Leveraging the IRON AI Engine API to Program the Ryzen™ AI NPU

@today Joe Melber, Jack Lo, Erika Hunhoff

@our team Muhammad Awad, Sam Bayliss, Kristof Denolf, Jeff Fifield, Andra Bisca, Yiannis Papadopoulos, Eddie Richter, Erwei Wang, André Rösti, Stephen Neuendorffer, Mario Ruiz Noguera, Ralph Wittig

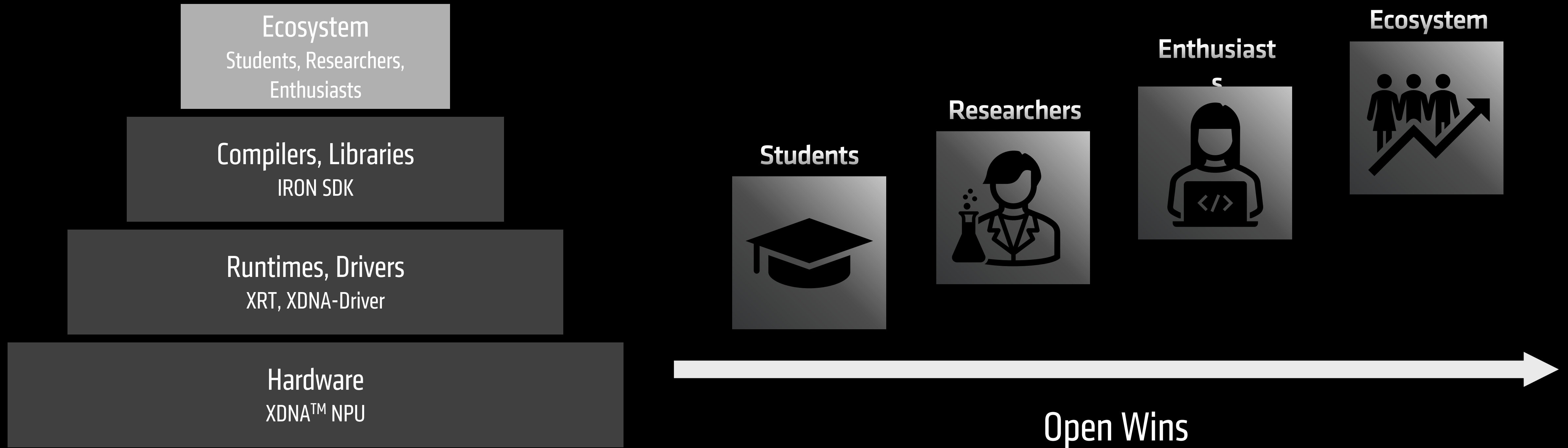
Research and Advanced Development (RAD)

March 23, 2026

[Public]

ENABLE DEVELOPERS EVERYWHERE

OPEN SOFTWARE LAYS THE FOUNDATION



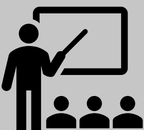
AMD together we advance_

[Public]

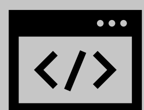
IRON NPU Development Kit Overview

Open-Source Fast Compiler

Close-to-Metal Python API



Programming Guides



Programming Examples

[Public]

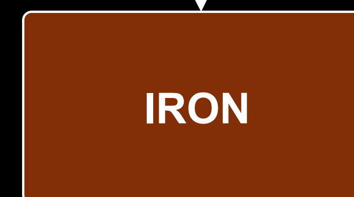
IRON: Close-to-Metal NPU Programming

- **IRON** is an open-source framework that lets developers unlock the ***full potential of AMD NPUs*** using:
  - Python APIs
  - Extensible compiler stack
  - Hardware-aware abstractions

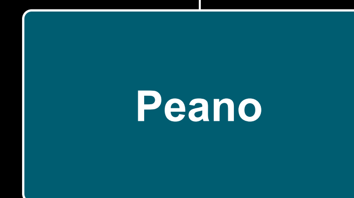
- **Core components**
  - ✓ **IRON Python API**
    Simplifies AIE programming with clean abstractions
  - ✓ **MLIR-AIE Dialect**
    Defines and lowers spatial operations to AIE cores via MLIR
  - ✓ **Peano Compiler**
    Optimized for fast **single-core AIE** compilation, leverages LLVM

```
import iron
@iron.jit
def func(a, b)
  ...
```

Python



MLIR



LLVM

AMD
together we advance_

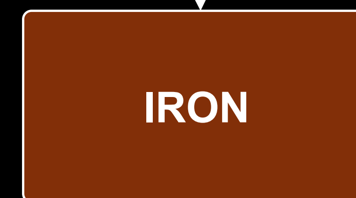
IRON: Close-to-Metal NPU Programming

- **IRON** is an open-source framework that lets developers unlock the ***full potential of AMD NPUs*** using:
  - Python APIs
  - Extensible compiler stack
  - Hardware-aware abstractions

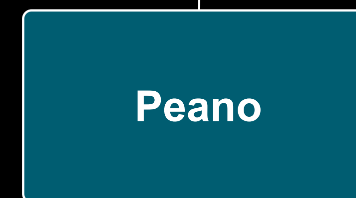
- **Core components**
  - ✓ **IRON Python API**
    Simplifies AIE programming with clean abstractions
  - ✓ **MLIR-AIE Dialect**
    Defines and lowers spatial operations to AIE cores via MLIR
  - ✓ **Peano Compiler**
    Optimized for fast **single-core AIE** compilation, leverages LLVM

```
import iron
@iron.jit
def func(a, b)
  ...
```

Python



MLIR



LLVM

AMD
together we advance_

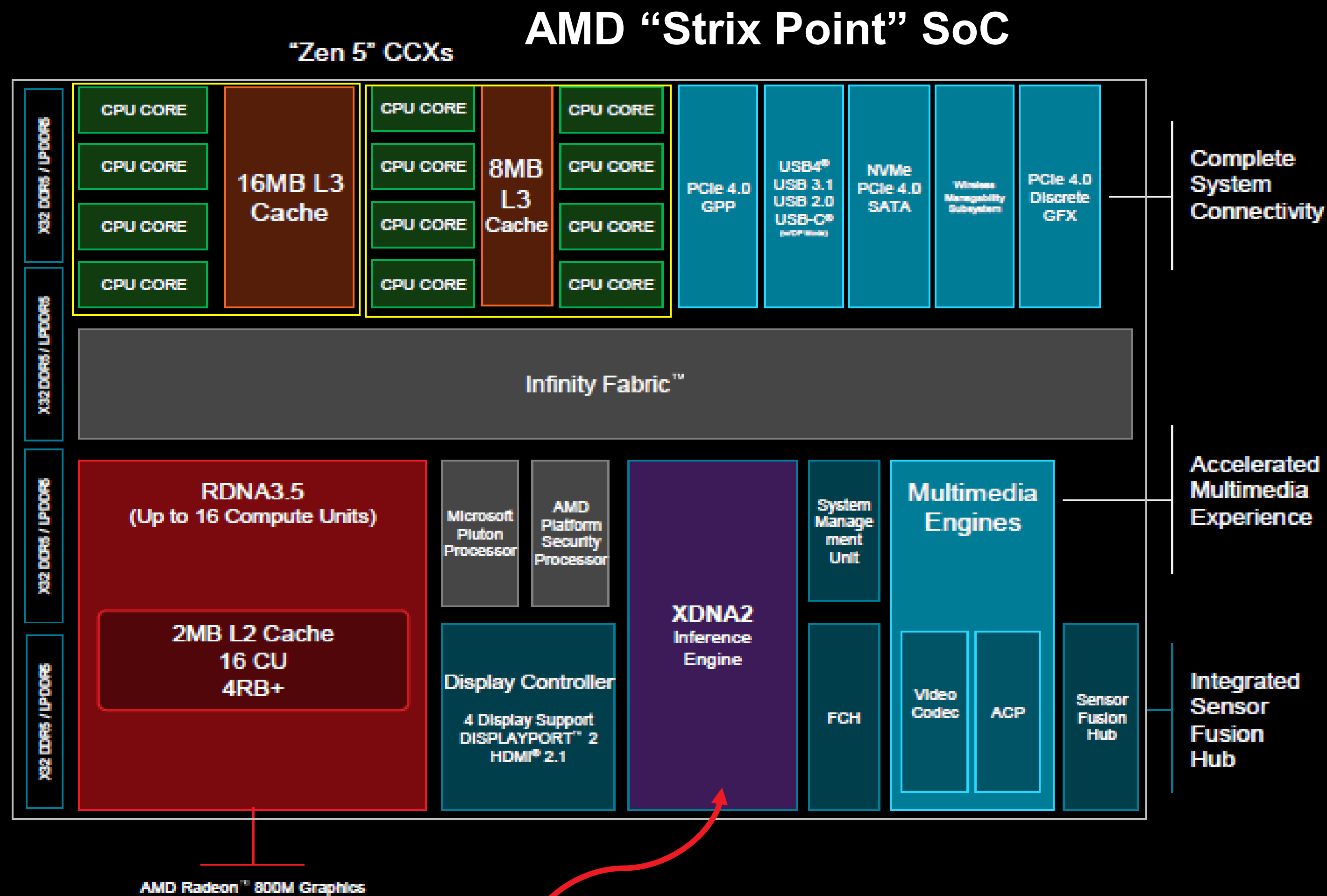
[Public]

Agenda

- Spatial Hardware Introduction
- Tile-by-Tile Deep Dive
- “Hello Worker” using IRON Python API
- Exercise 1:
  - Access to AUP Cloud
  - Run Your Frist Program!
- TODO

AMD
together we advance_

AMD Ryzen AI Architecture



Spatial Hardware

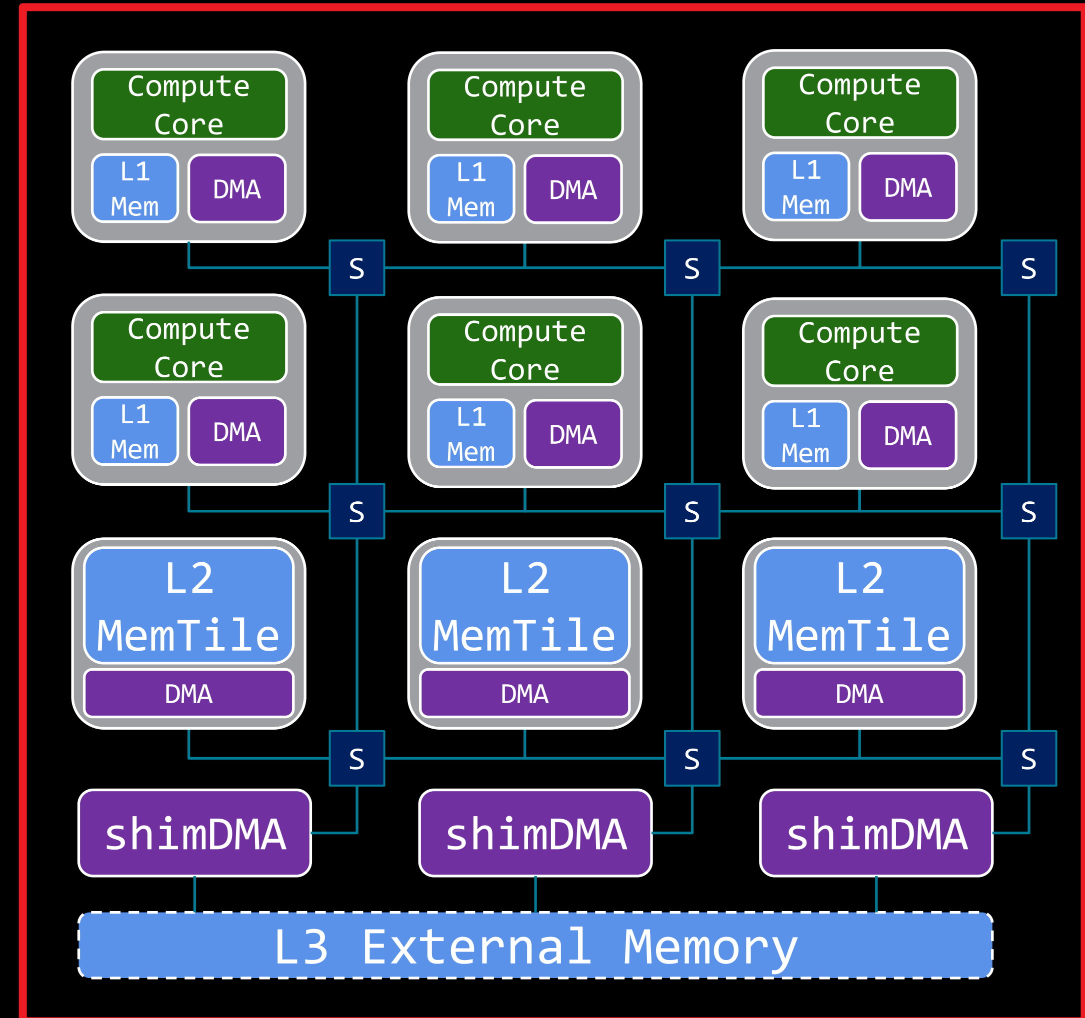
[Hot Chips '24:Brad Cohen and Mahesh Subramony, Next Generation "Zen 5" Core]

AMD together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The Neural Processing Unit (NPU)

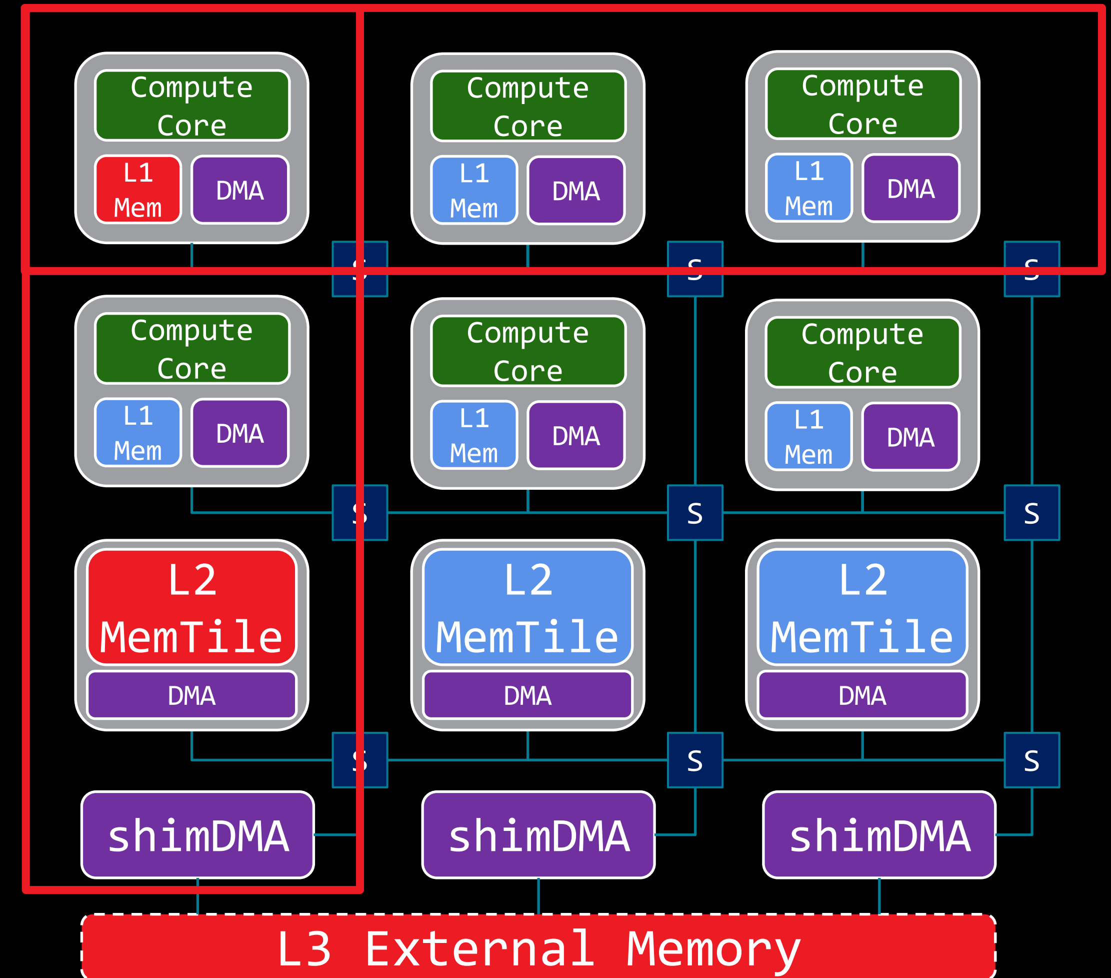


AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The Neural Processing Unit (NPU) has a **spatial architecture**

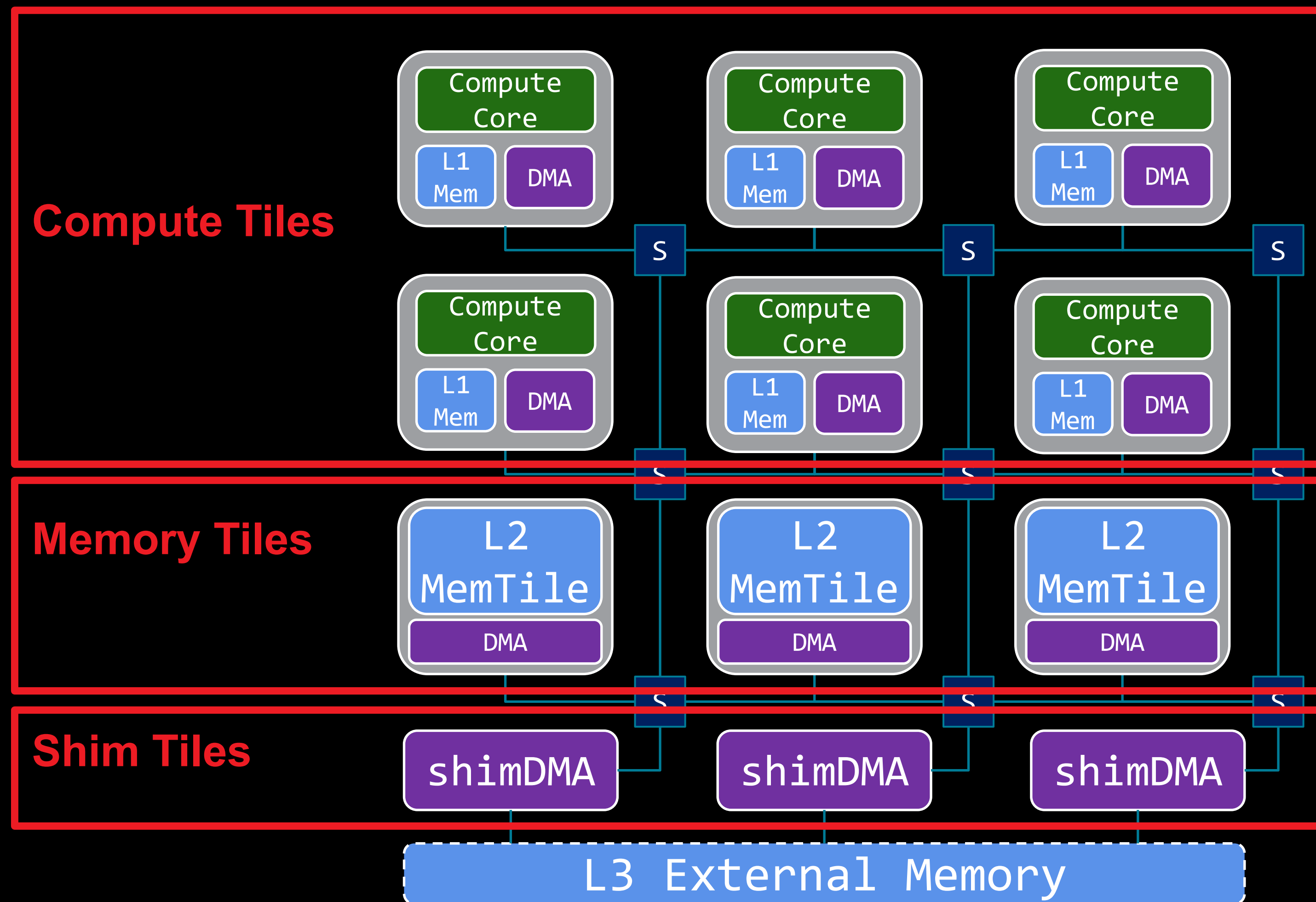


AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The Neural Processing Unit (NPU) has

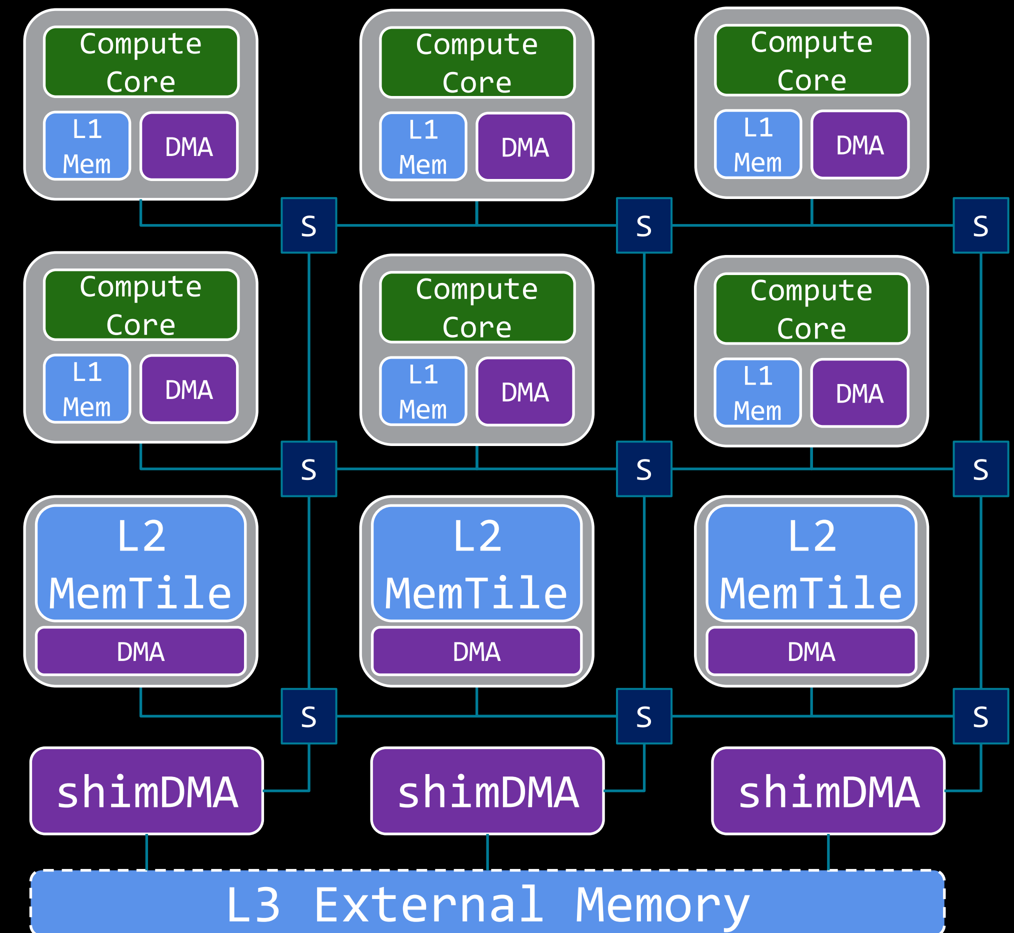


AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has



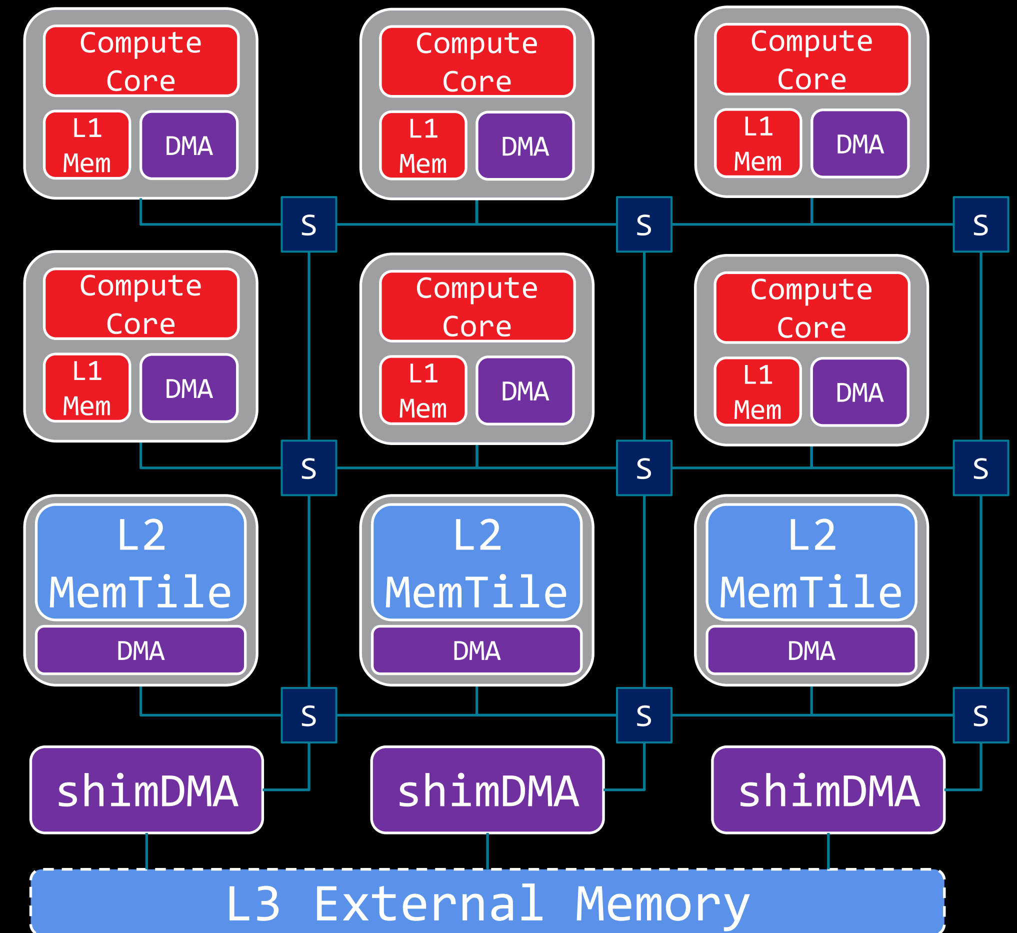
AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- **Spatially distributed compute and memory**



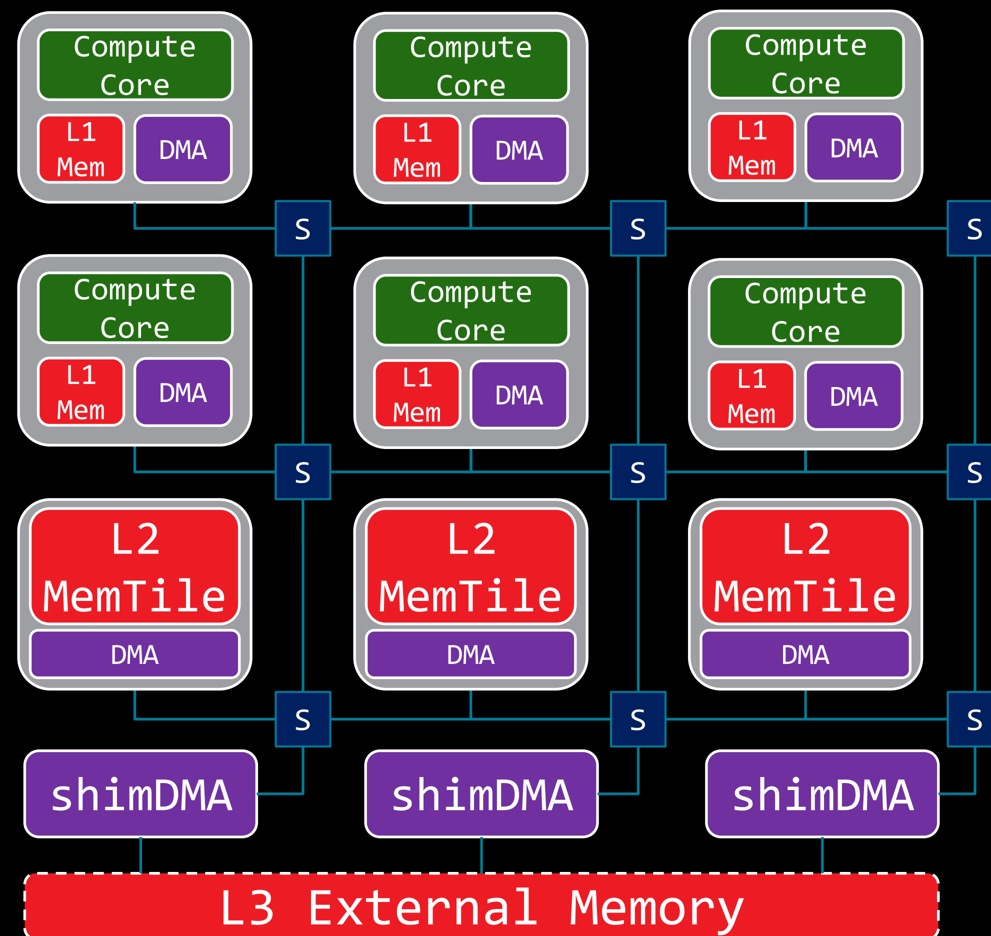
AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- **Multi-level memory hierarchy**



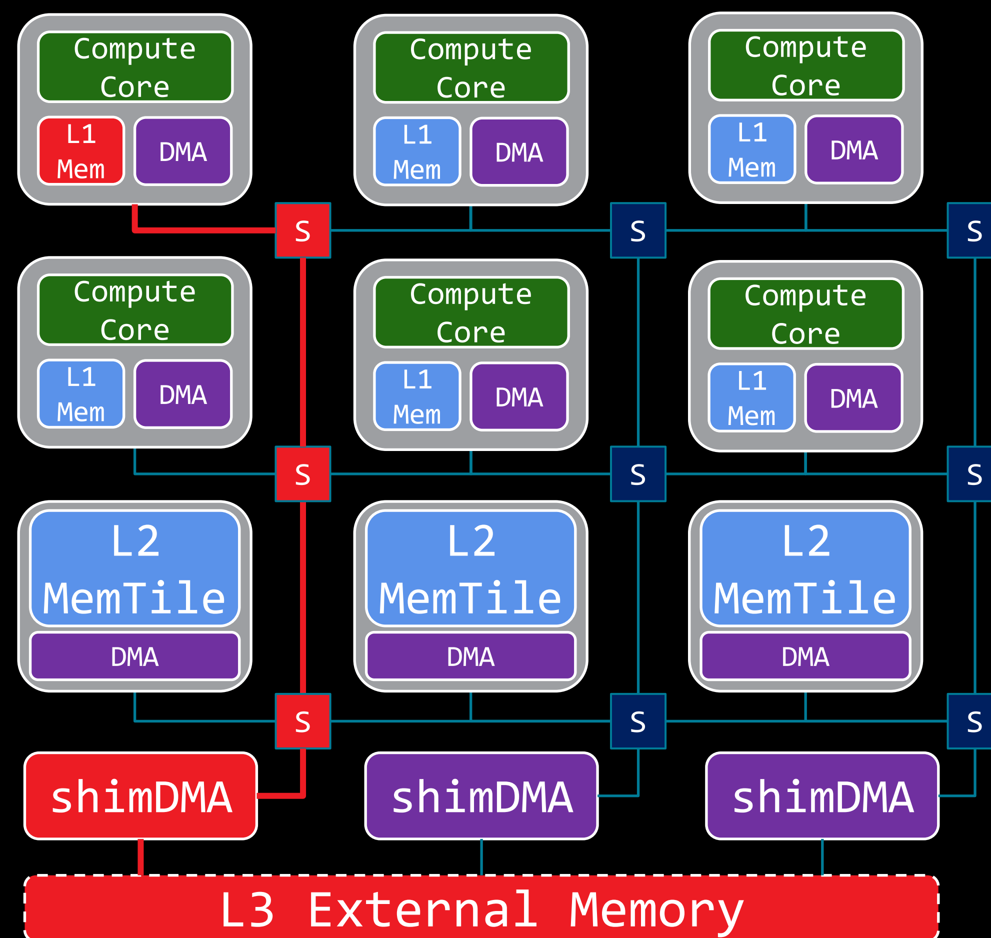
AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- **Explicitly scheduled, decoupled data movement**



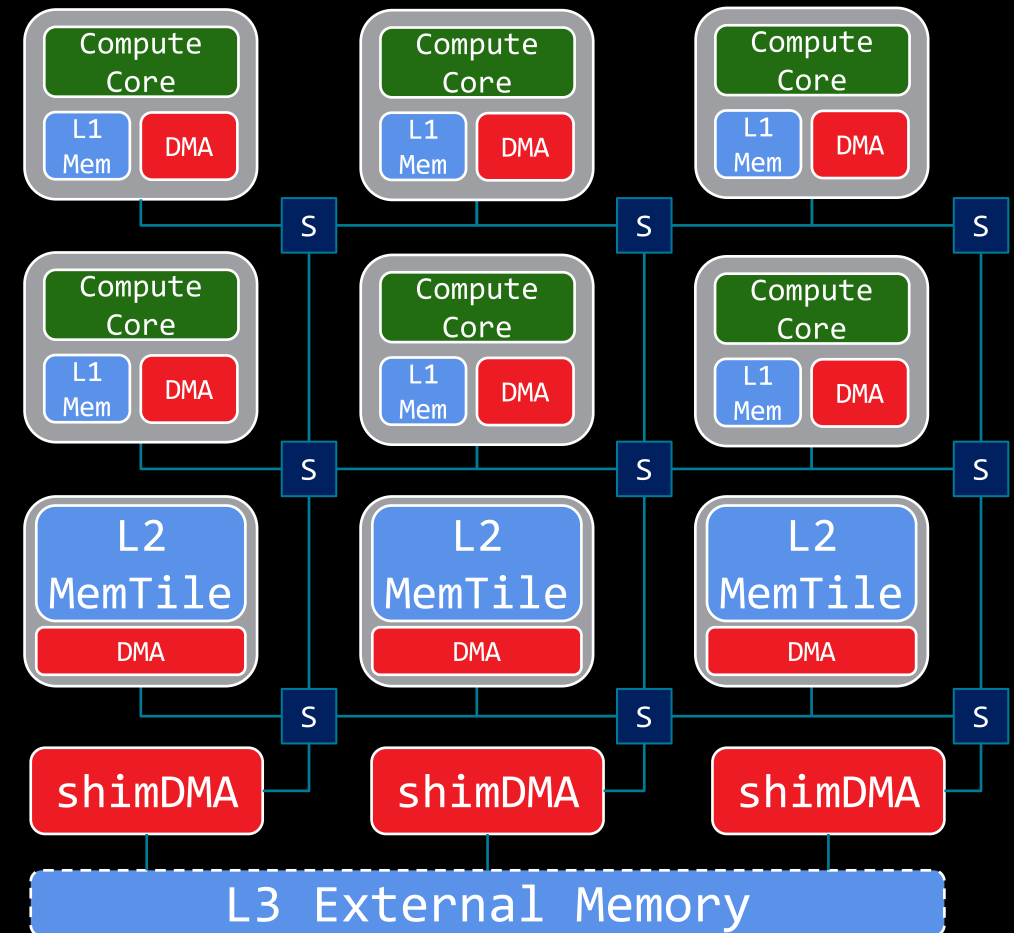
AMD
together we advance_

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

Data Movement Accelerators (DMA)



AMD
together we advance_

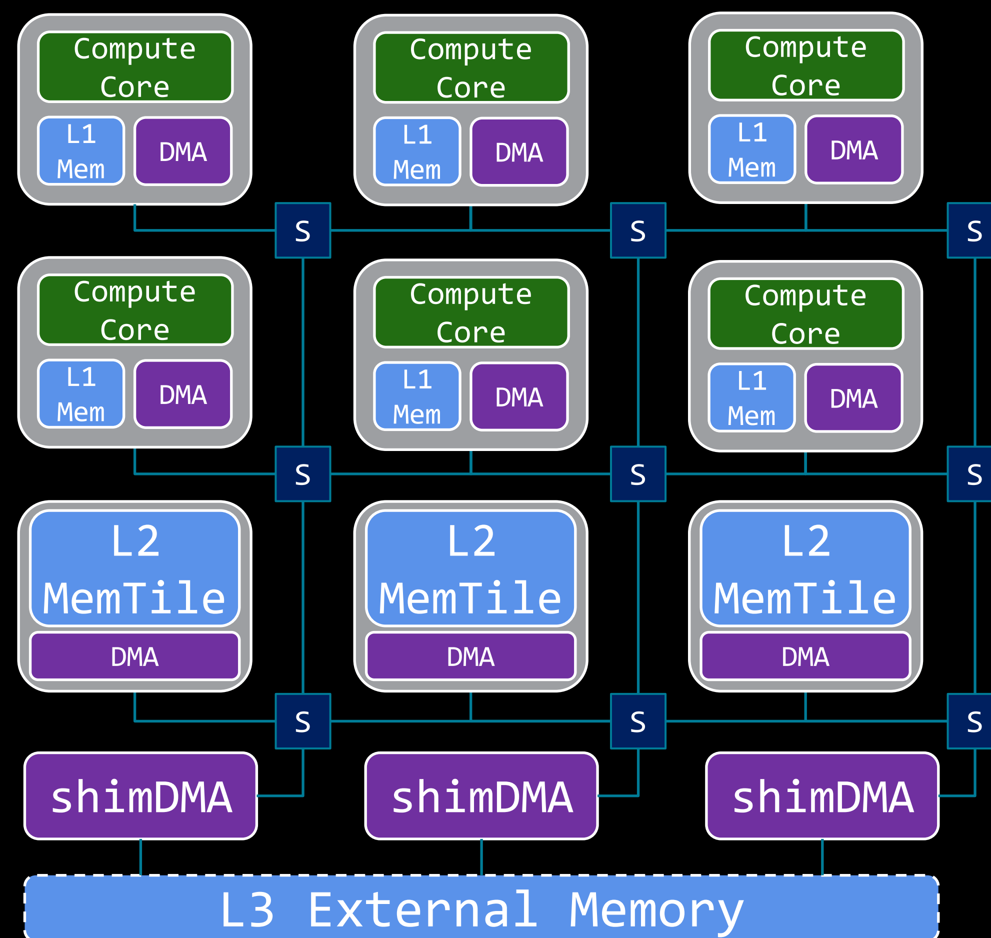
[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables



AMD
together we advance_

[Public]

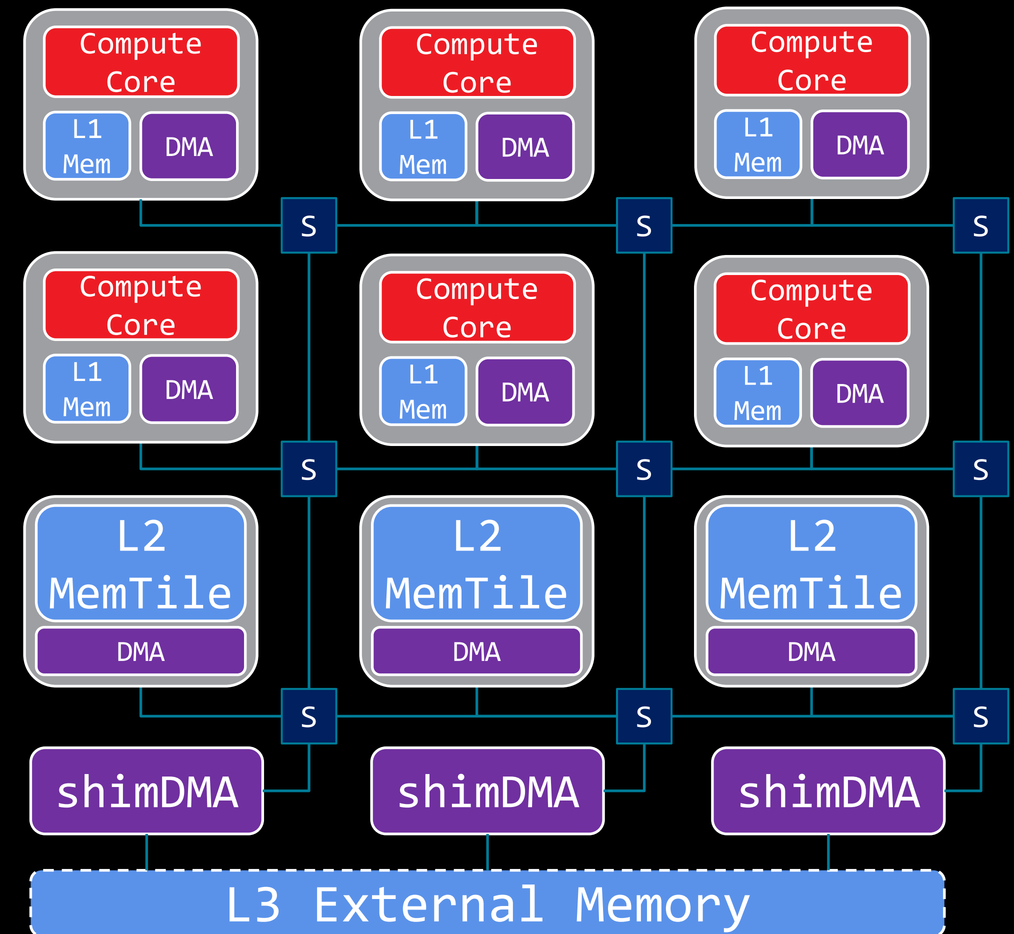
AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables

- **Independent tasks on each core**



AMD
together we advance_

[Public]

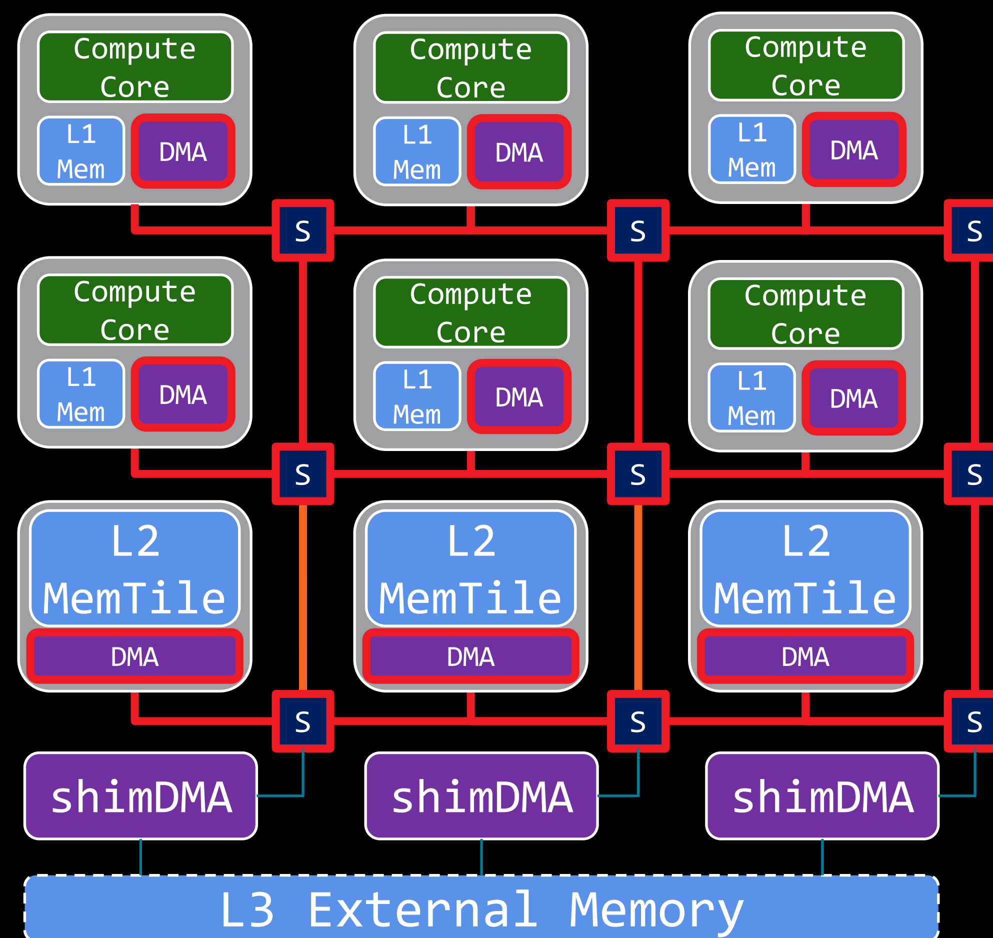
AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables

- Independent tasks on each core
- **Dataflow between tasks**



AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

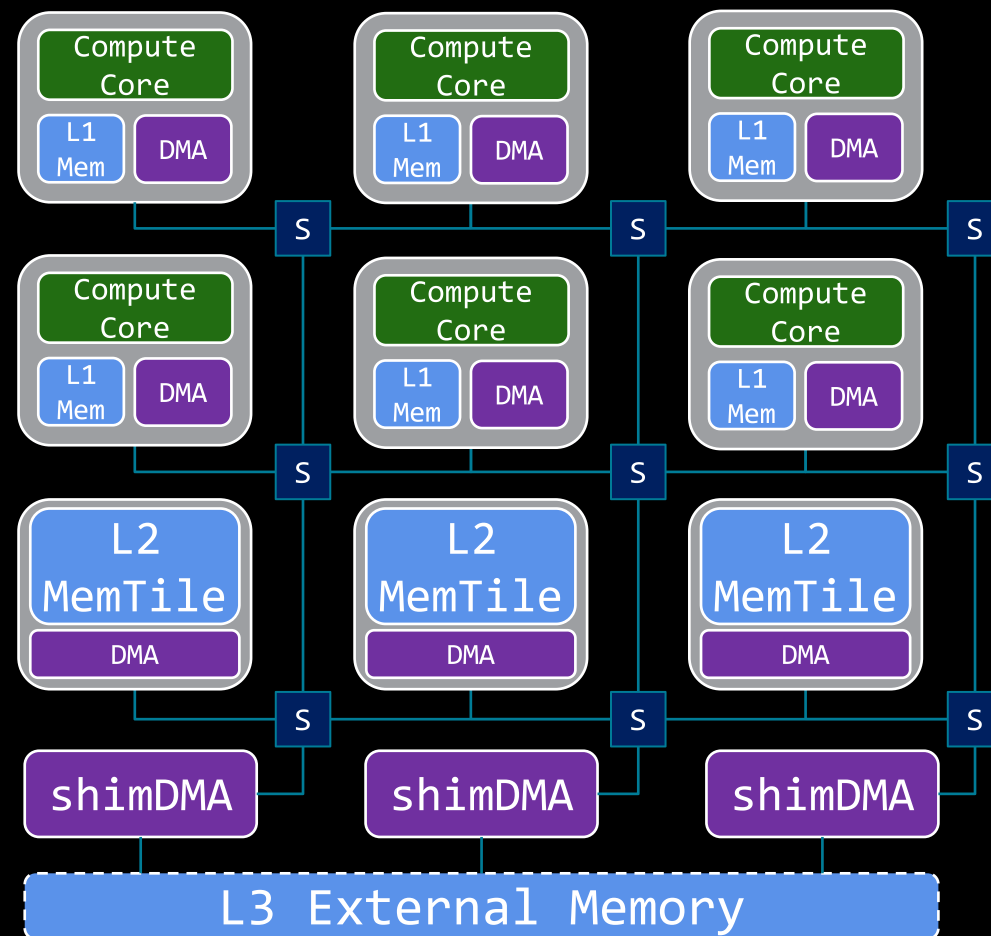
The AIE array has

- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables

- Independent tasks on each core
- Dataflow between tasks

The AIE array requires



AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

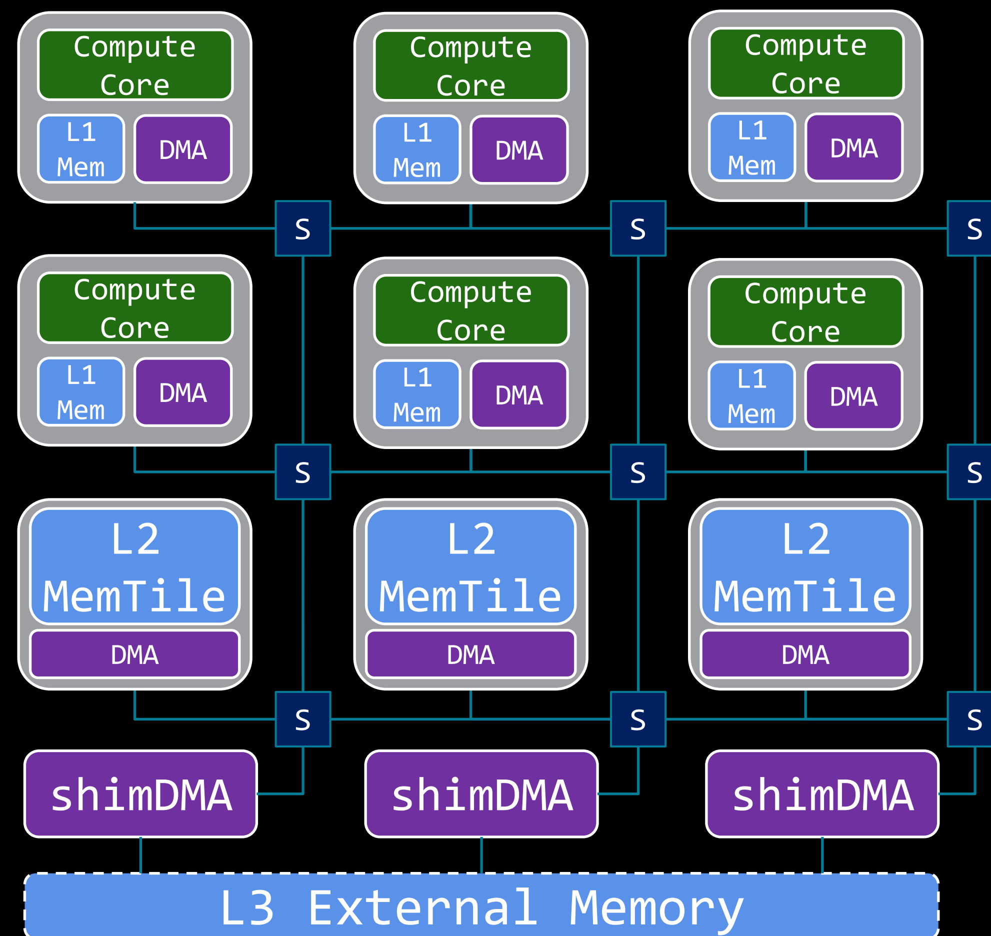
- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables

- Independent tasks on each core
- Dataflow between tasks

The AIE array requires

- Synchronization



AMD
together we advance_

[Public]

AMD XDNA: Modular and Scalable Architecture with Independent Concurrent Processing and Explicit Data Movement

The AIE array has

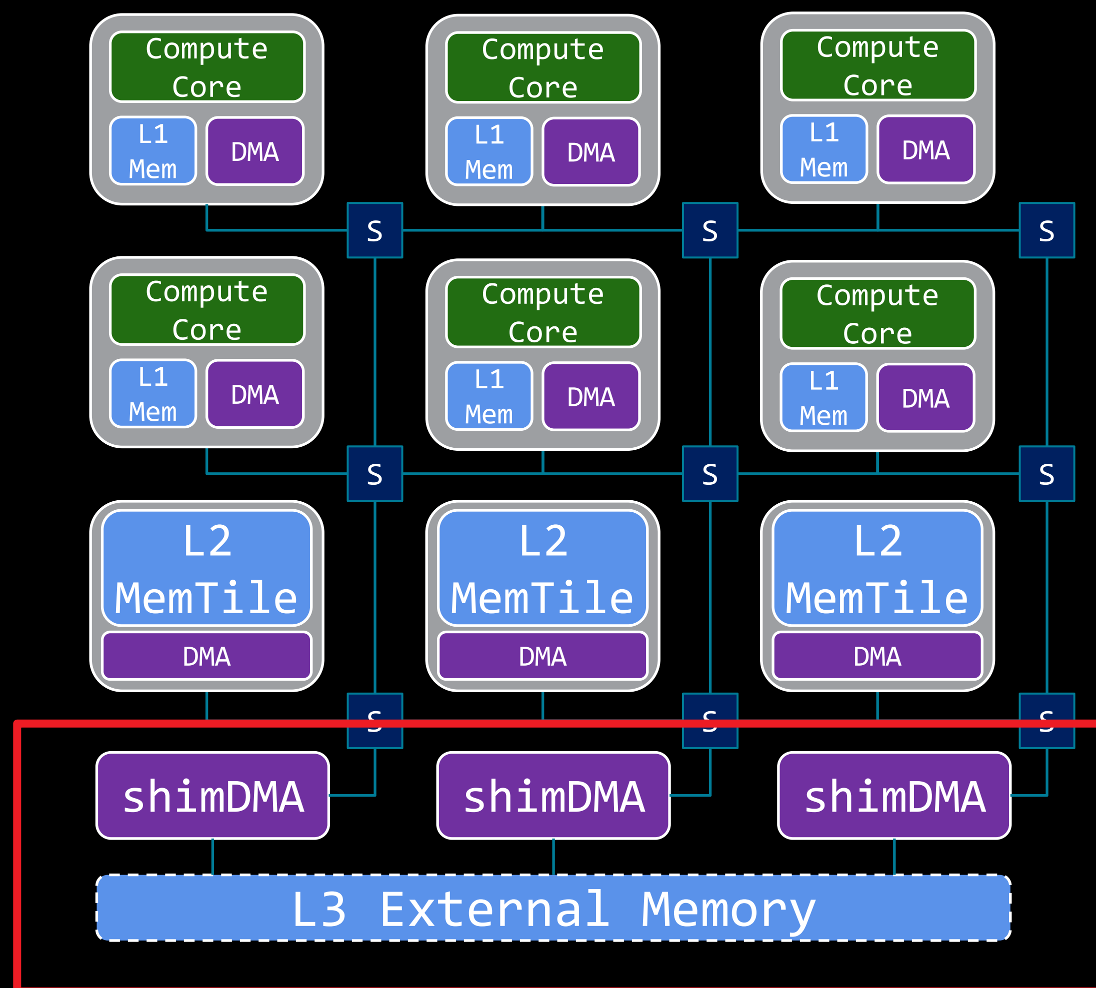
- Spatially distributed compute and memory
- Multi-level memory hierarchy
- Explicitly scheduled, decoupled data movement

The AIE array enables

- Independent tasks on each core
- Dataflow between tasks

The AIE array requires

- Synchronization
- **Runtime operations**



AMD
together we advance_

AMD XDNA™ 2 architecture leadership

AMD XDNA

20
AI Engine Tiles

10
NPU TOPS

Performance uplift

2x MACs per tile

1.6x on-chip memory

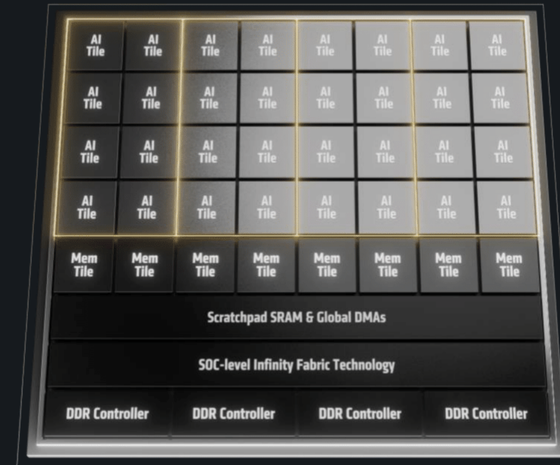
Block floating point

Enhanced non-linear support

AMD XDNA 2

32
AI Engine Tiles

50
NPU TOPS

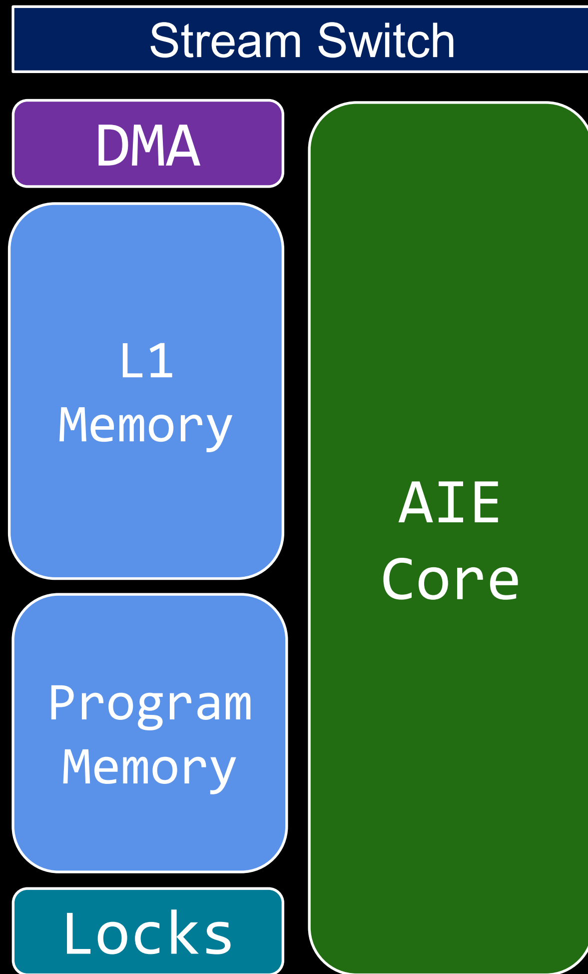


AMD Zen 5 Tech Day: https://www.anandtech.com/show/21469/amd-details-ryzen-ai-300-series-for-mobile-strix-point-with-rdna-35-igpu-xdna-2-npu/2

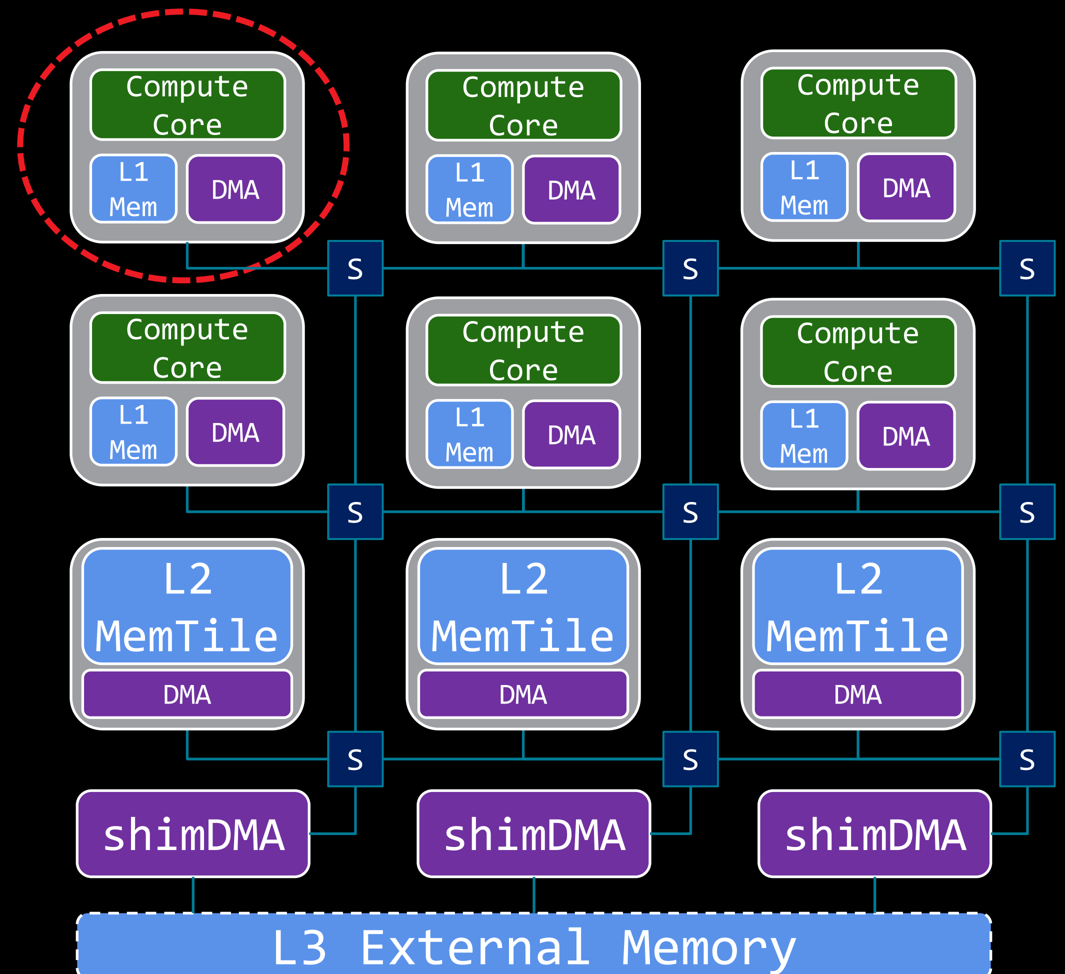
Tile-by-Tile Deep Dive

[Public]

AIE Compute Tile Architecture

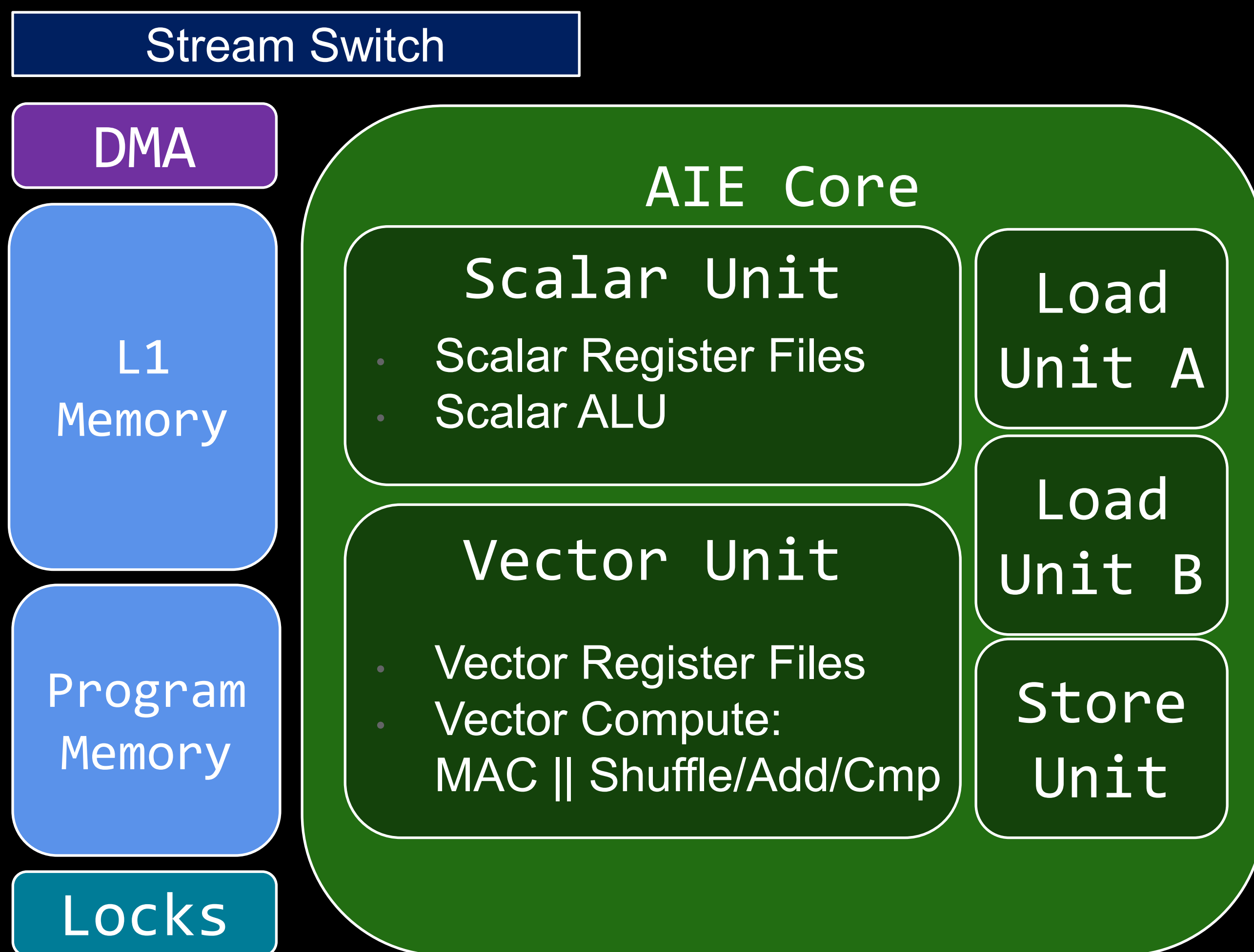


ComputeTile = tile(0, 3)



AMD
together we advance_

AIE Compute Tile Architecture: Compute Details



Precision 1	Precision 2	Acc Lanes	Bits per Acc Lane	MACs
int 8	int 4	32	32	512
int 8	int 8	32	32	256
int 16	int 8	32	32	128
int 16	int 16	32	32	64
int 32	int 16	16	64	32
int 32 [1]	int 32	16	64	16
bfloat 16 [3]	bfloat 16	16	SPFP 32 [2]	128

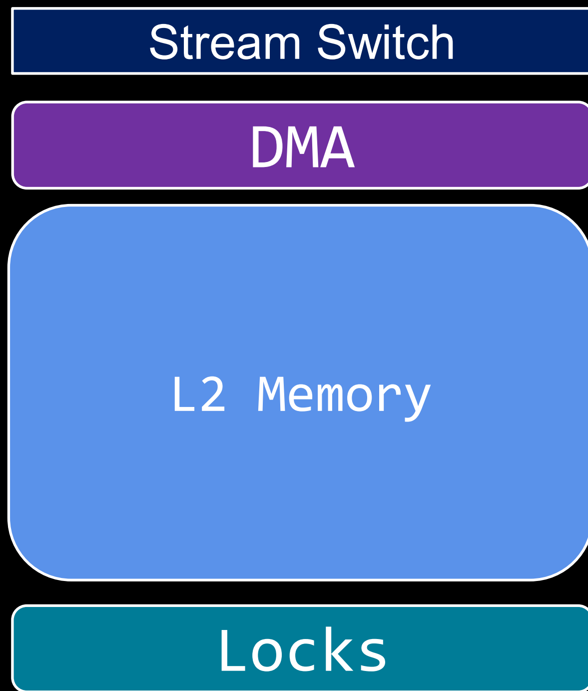
1.int32 x int32 can be emulated. The operation should have half the performance of int32 x int16 and there should be 16 multiplications per cycle.
2.Single precision floating point (SPFP) per the IEEE standard.
3.float32 x float32 can be emulated. Emulation deviates from the IEEE-754 standard.

AIE-ML AIE API Manual: https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aie_api/doc/

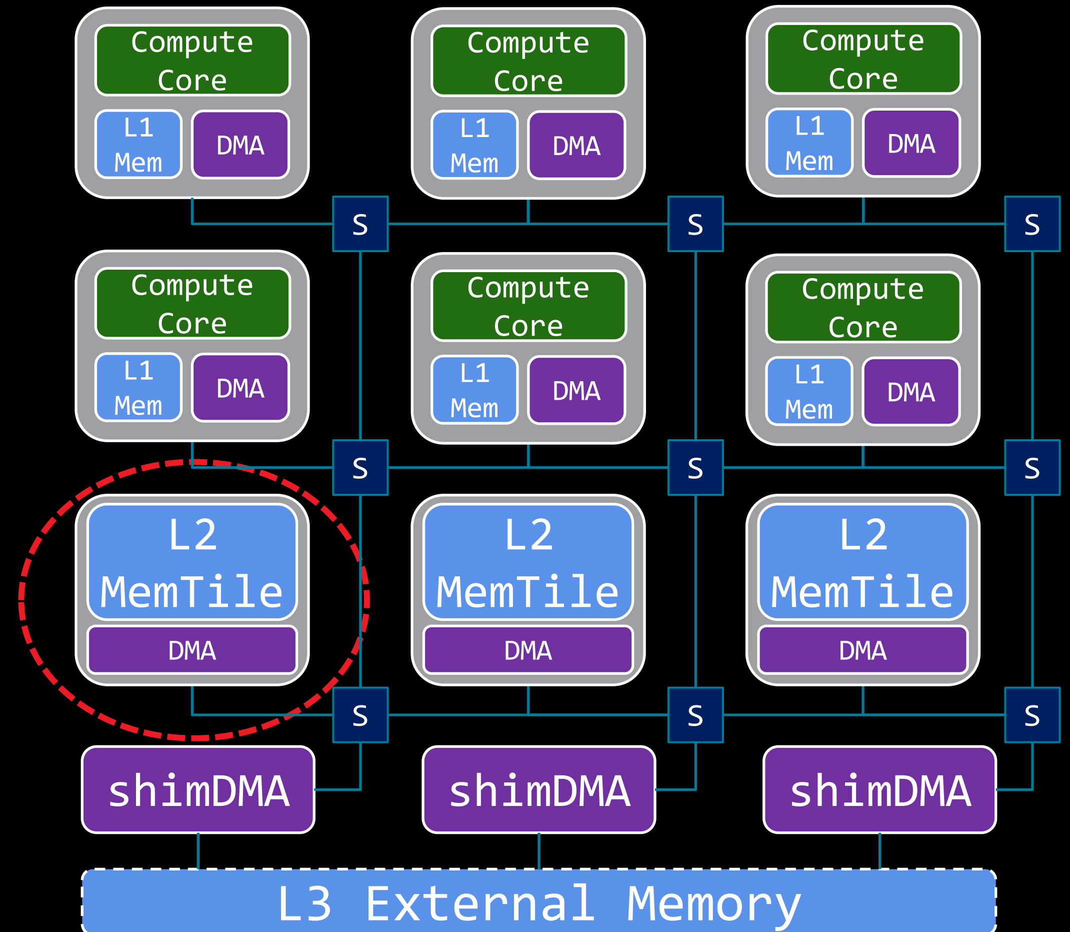
AMD
together we advance_

[Public]

AIE Mem Tile Architecture



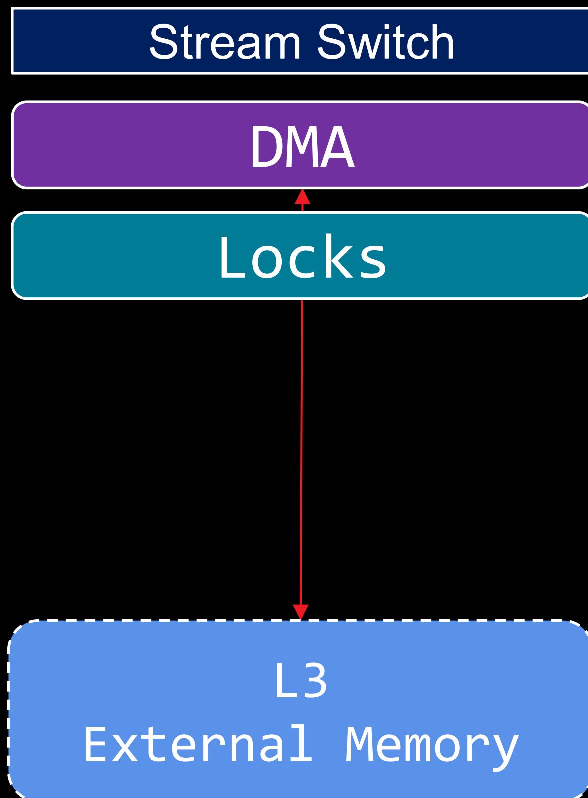
MemTile = tile(0, 1)



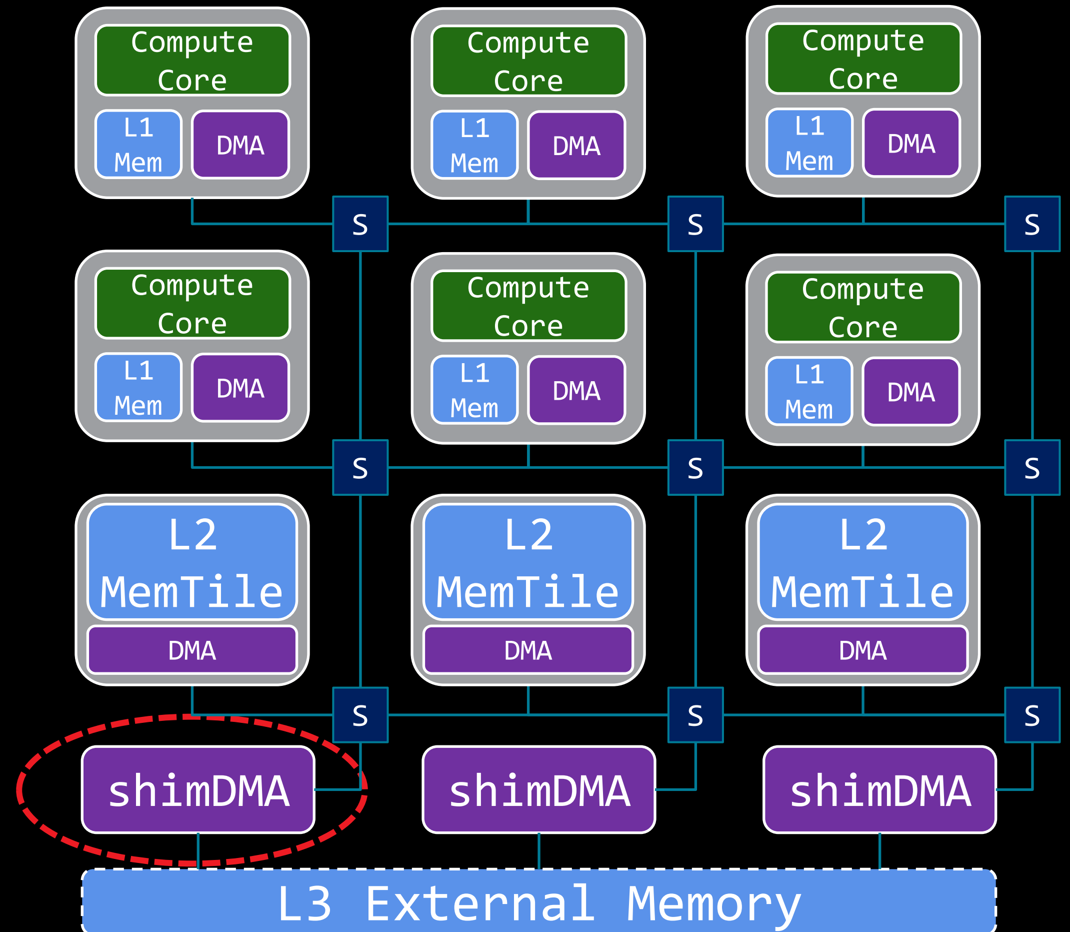
AMD
together we advance_

[Public]

AIE Shim Tile Architecture



ShimTile = tile(0, 0)



AMD
together we advance_

IRON “Hello Worker”

[Public]

IRON API Options

More Abstraction

Less Abstraction

```
...

@iron.jit(is_placed=False)
def example(input):
    ...
```

Primary focus of this tutorial: IRON Python API

```
...
with mlir_mod_ctx() as ctx:
    @device(AIEDevice.npu1_1col)
    def device_body():
        ...
```

Autogenerated Python Bindings + Convenience Extensions (e.g., placed)

```
module {
  aie.device(npu2) {
    %shim_noc_tile_0_0 = aie.tile(0, 0)
    %tile_0_2 = aie.tile(0, 2)
    ...
```

MLIR Dialects (e.g., AIE, AIEX)

AMD
together we advance_

[Public]

IRON “Hello Worker”

```
from aie.iron import Program, Runtime, Worker, Buffer
from aie.iron.placers import SequentialPlacer
from aie.iron.device import NPU1Col1, Tile
from aie.iron.controlflow import range_

import aie.iron as iron
```

Include modules defining IRON libraries

AMD
together we advance_

[Public]

IRON "Hello Worker"

```python
from aie.iron import Program, Runtime, Worker, Buffer
from aie.iron.placers import SequentialPlacer
from aie.iron.device import NPU1Col1, Tile
from aie.iron.controlflow import range_

import aie.iron as iron

@iron.jit(is_placed=False)
def example():
    data_size = input.numel()
    data_ty = np.ndarray[(data_size,), np.dtype[input.dtype]]
    buff = Buffer(data_ty,initial_value=np.array(range(data_size), dtype=np.int32))

    # Task for the worker to perform
    def core_fn(local_buff):
        for i in range_(data_size):
            local_buff[i] = 0

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [buff], placement=Tile(0, 2), while_true=False)
```

JIT decorator indicating IRON design targeting an AMD NPU

- *Worker()* represents a compute tile in the AIE array
- *core_fn* is the task executed by a Worker
- Workers can be hand placed, or their placement can be left to the compiler (e.g. a placer)

AMD
together we advance_

[Public]

IRON “Hello Worker”

```
from aie.iron import Program, Runtime, Worker, Buffer
from aie.iron.placers import SequentialPlacer
from aie.iron.device import NPU1Col1, Tile
from aie.iron.controlflow import range_

import aie.iron as iron

@iron.jit(is_placed=False)
def example():
    data_size = input.numel()
    data_ty = np.ndarray[(data_size,), np.dtype[input.dtype]]
    buff = Buffer(data_ty,initial_value=np.array(range(data_size), dtype=np.int32))

    # Task for the worker to perform
    def core_fn(local_buff):
        for i in range_(data_size):
            local_buff[i] = 0

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [buff], placement=Tile(0, 2), while_true=False)


    # Runtime operations to start workers and move data to/from external memory
    rt = Runtime()
    with rt.sequence() as (_):
        rt.start(my_worker)
```

Runtime() starts workers and data movement to/from external memory

AMD
together we advance_

[Public]

IRON "Hello Worker"

```
from aie.iron import Program, Runtime, Worker, Buffer
from aie.iron.placers import SequentialPlacer
from aie.iron.device import NPU1Col1, Tile
from aie.iron.controlflow import range_

import aie.iron as iron

@iron.jit(is_placed=False)
def example():
    data_size = input.numel()
    data_ty = np.ndarray[(data_size,), np.dtype[input.dtype]]
    buff = Buffer(data_ty,initial_value=np.array(range(data_size), dtype=np.int32))

    # Task for the worker to perform
    def core_fn(local_buff):
        for i in range_(data_size):
            local_buff[i] = 0

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [buff], placement=Tile(0, 2), while_true=False)


    # Runtime operations to start workers and move data to/from external memory
    rt = Runtime()
    with rt.sequence() as (_):
        rt.start(my_worker)

# Create the program from the device type and runtime
my_program = Program(iron.get_current_device(), rt)
# Place program (assign resources on the device) and generate MLIR
return my_program.resolve_program(SequentialPlacer())
```

Components assigned to a *Program():* design, device and runtime

AMD
together we advance_

IRON “Hello Worker”

```
…

@iron.jit(is_placed=False)
def example():
    …

def main():
    # Define tensor shape and data type
    tensor_size = 48
    data_type = np.int32

    # Construct an input tensor and an output zeroed tensor
    input = iron.arange(tensor_size, dtype=data_type, device="npu")

    # JIT-compile the kernel then launches the kernel with the given arguments.
    example(input)

    # Exit because the example has no output.
    print("\nPASS!\n")
    sys.exit(0)

if __name__ == "__main__":
    main()
```

“main” host program

Future calls to the *example* kernel will reuse the compiled design and loaded code objects

AMD
together we advance_

[Public]

IRON "Hello Worker" – Runtime Input

```
...

@iron.jit(is_placed=False)
def example():
    ...

def main():
    # Define tensor shape and data type
    tensor_size = 48
    data_type = np.int32

    # Construct an input tensor and an output zeroed tensor
    input = iron.arange(tensor_size, dtype=data_type, device="npu")

    # JIT-compile the kernel then launches the kernel with the given arguments.
    example(input)

    # Exit because the example has no output.
    print("\nPASS!\n")
    sys.exit(0)

if __name__ == "__main__":
    main()
```

Signature of JIT function matches call

Call the JIT function

AMD
together we advance_

IRON “Hello Worker” – Runtime Input

```
...

@iron.jit(is_placed=False)
def example(input):
    ...
    # Runtime operations to start workers and move data to/from external memory
    rt = Runtime()
    with rt.sequence(tensor_size) as (in):
        ..

def main():
    # Define tensor shape and data type
    tensor_size = 48
    data_type = np.int32

    # Construct an input tensor and an output zeroed tensor
    input = iron.arange(tensor_size, dtype=data_type, device="npu")

    # JIT-compile the kernel then launches the kernel with the given arguments.
    example(input)

    # Exit because the example has no output.
    print("\nPASS!\n")
    sys.exit(0)

if __name__ == "__main__":
    main()
```


AMD
together we advance_

AUP Cloud Ryzen AI cluster Login Information

- Login to JupyterHub via web browser
  - https://aupcloud.io/aipc-<*id*> – provided separately
  - Token: provided separately

AMD
together we advance_

[Public]

JupyterHub login page

aupcloud.io/aipc-<id>

AMD University Program

Sign into the AUP Cloud

Unique token:

<provided-token-goes-here>

Token

Sign in

Welcome to the Ryzen™ AI AUP Cloud

This page allows you to spawn a server for a period of time and run a number of stacks via JupyterLab on Ryzen AI enabled devices.

To access it you should have been given a unique token, these tokens will allow you to login until their expiration date.
While the token is active any data that you write into the **/notebooks** directory will persist.
Once the token expires, your data could be deleted at anytime.
The AUP Remote Learning Lab is maintained by the AMD University Program (AUP) team in Ireland.

Login Issues

If you get the log in error **Invalid username or password**:

- If this is the first time trying to login, make sure the token is correct.
- If you logged in before, your token has likely expired.
- Make sure the URL you entered is correct, e.g., check for the port

For more information or to request an access token please email: AMD University Program.

Useful Links

- Ryzen™ AI
- Riallto Documentation
- Riallto GitHub Repo
- Riallto Discussion Forum
- MLIR-AIE - An MLIR-based toolchain for AMD AI Engine-enabled devices.

AMD University Program

Server Options

Logout and end your session

Select your desired environment (ROCm or MLIR-AIE or Ryzen AI SW or Riallto) from the dropdown below:

mlir-aie:latest

0 of 4 server slots are currently in use

Run my server for (minutes): 20

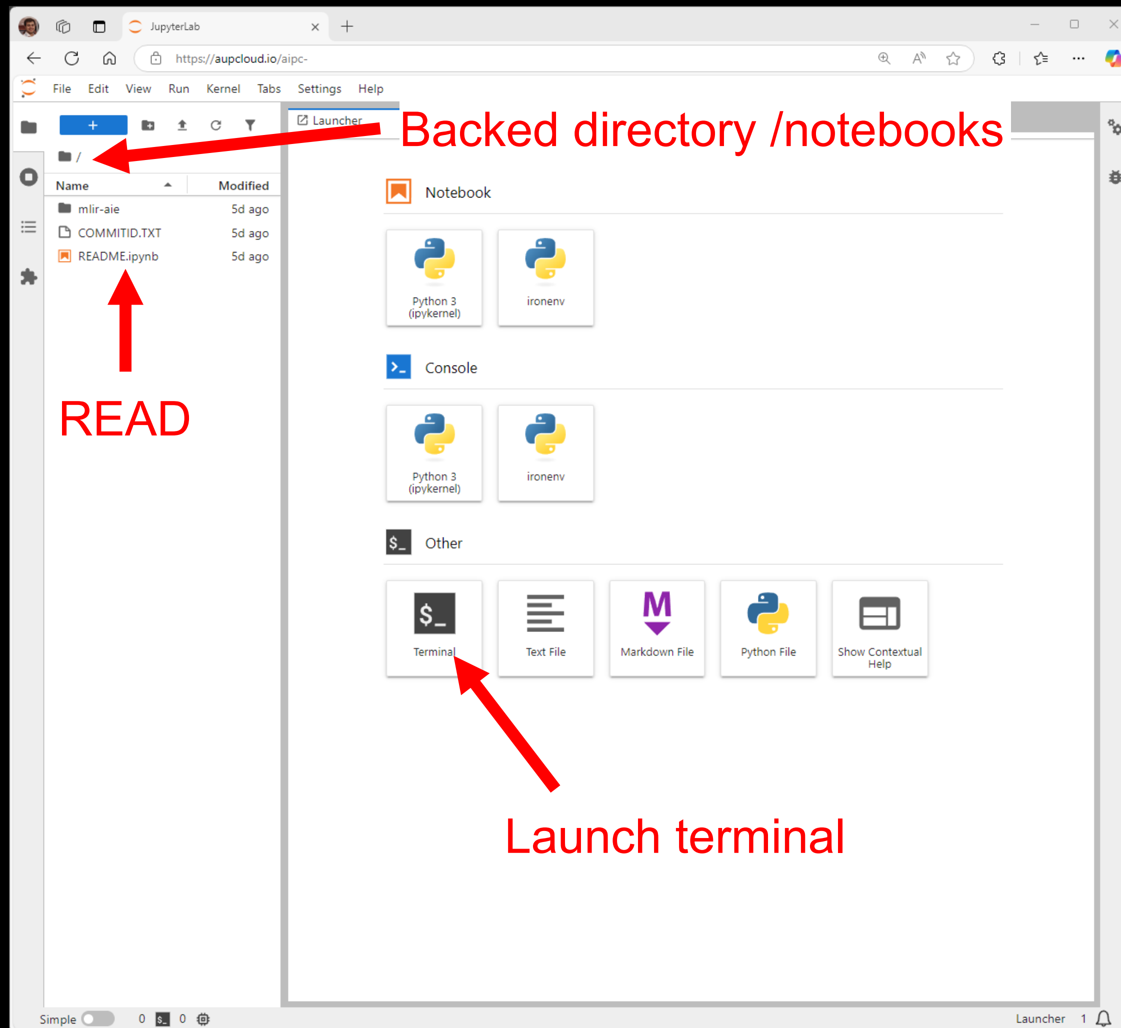
Your token expires on **31 December 2030 at 23:59:59**.

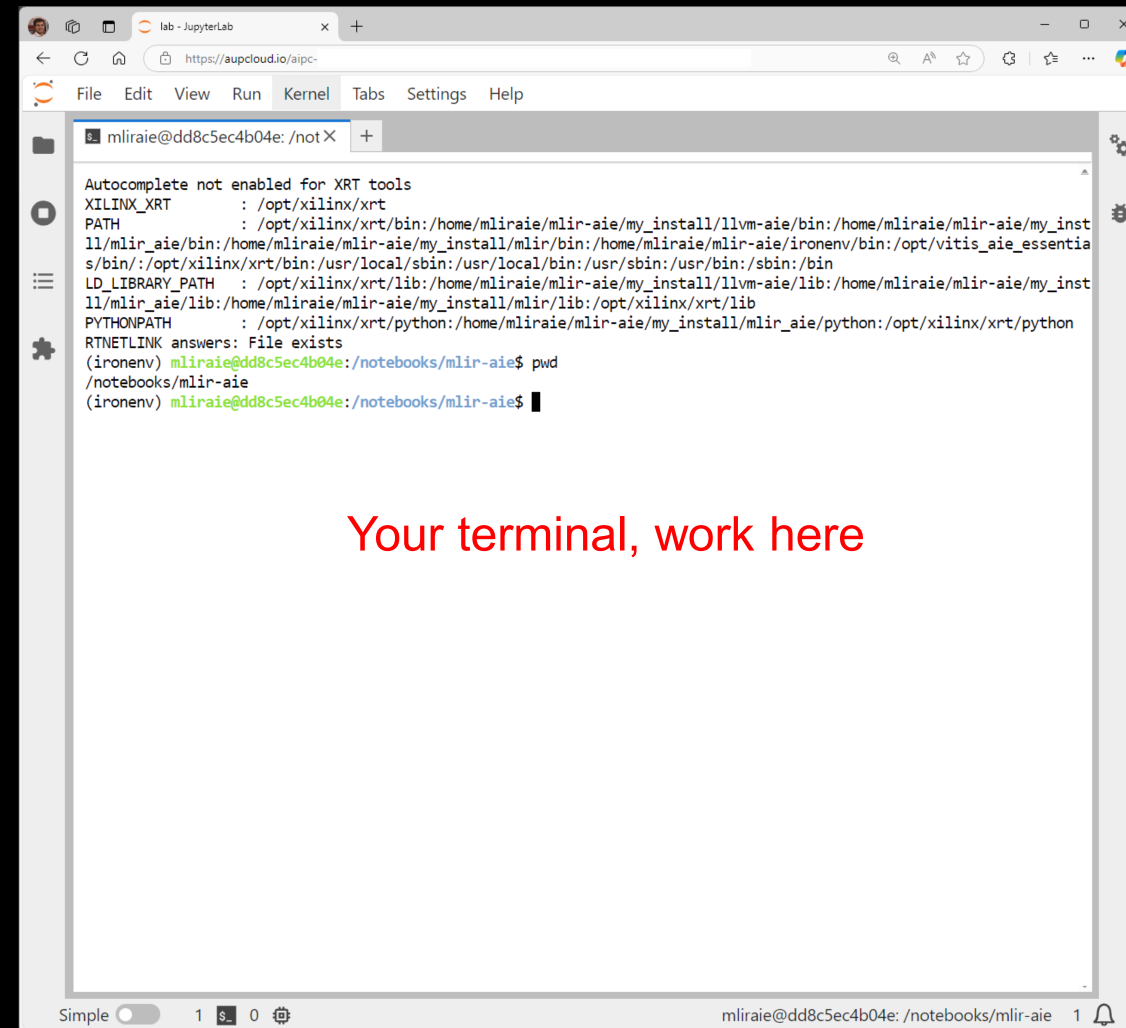
Start

AMD together we advance_

[Public]

JupyterHub working with IRON

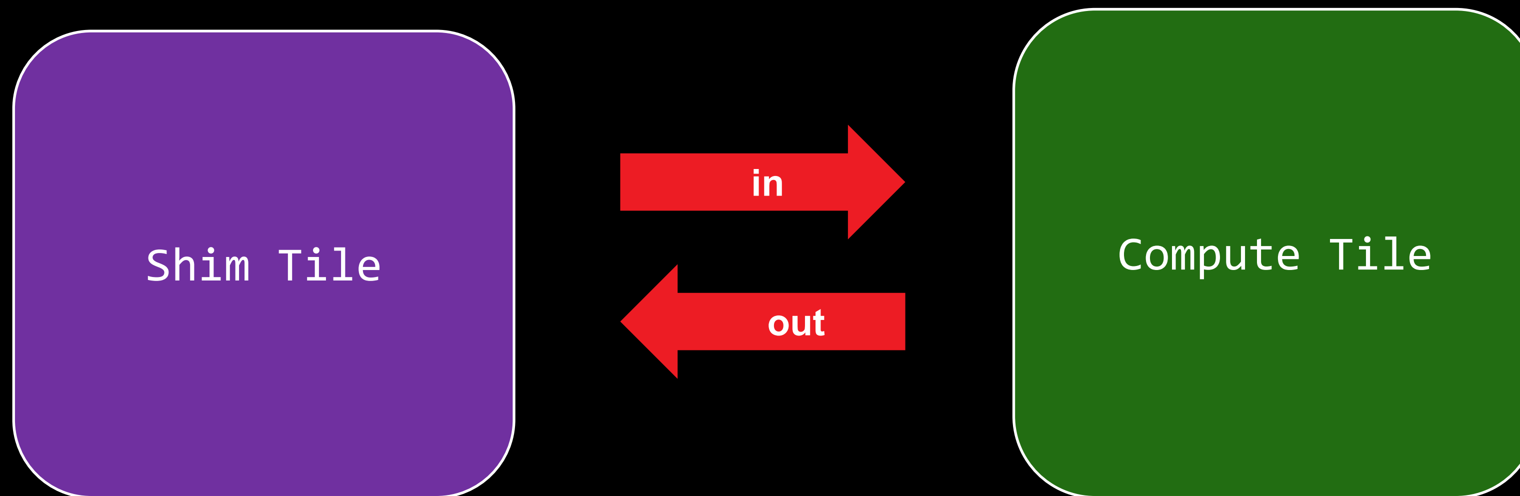




AMD
together we advance_

[Public]

Exercise 1: Run Your First Program



Memcpy: out = in

AMD
together we advance_

Exercise 1: Run Your First Program

Navigate in your browser to:

https://github.com/Xilinx/mlir-aie/tree/main/programming_examples/getting_started/00_memcpy

- This is a parallel parameterized design that uses multiple shim DMA channels in N columns

- Exercise:
  - Profile the NPU bandwidth with memcpy
  - Follow the README in the directory

Objective

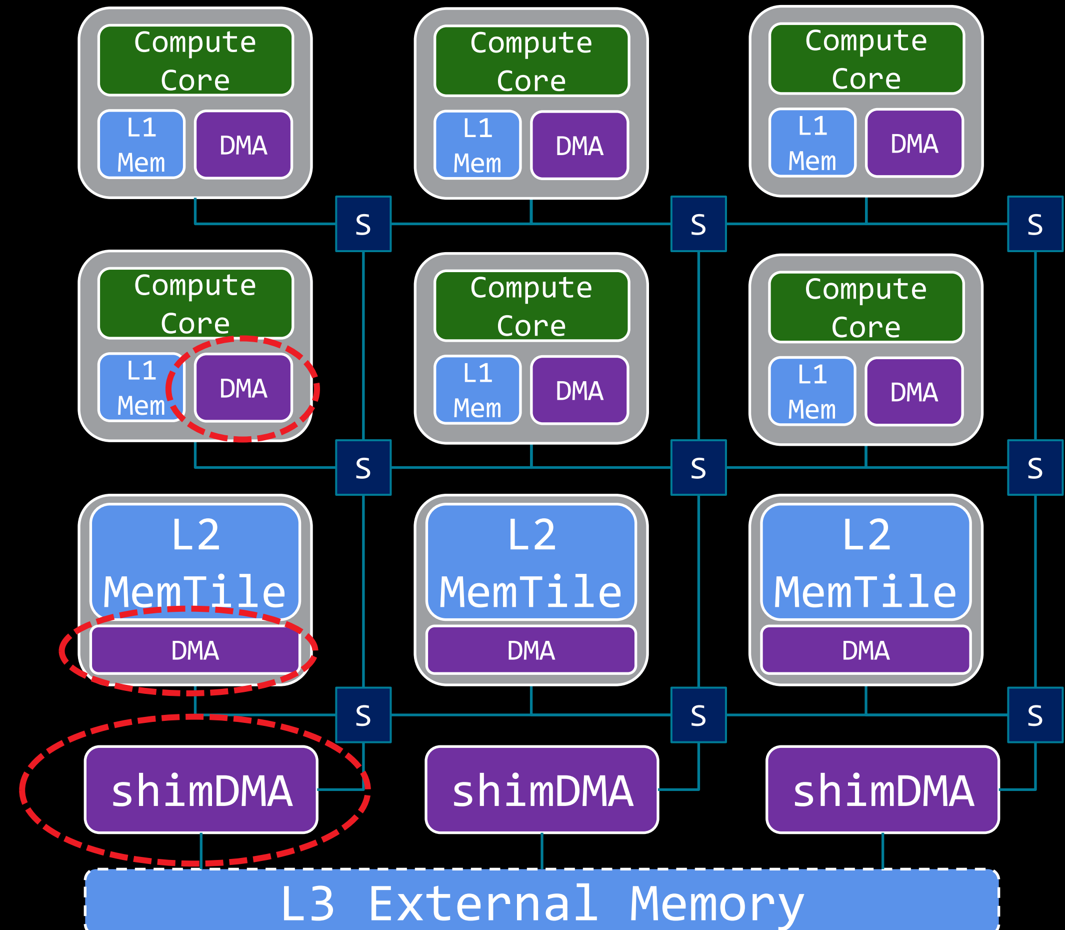
- Understand the structure of a full NPU dataflow application using IRON
- Measure and report the peak memory bandwidth achieved
- Experiment with parameters (columns, channels, data size) to study how design choices affect performance
- If you have extra time, peek at the other “Getting Started” examples

AMD
together we advance_

Data Movement with ObjectFifo

[Public]

Spatial Compute = Data Movement a Main Programming Task

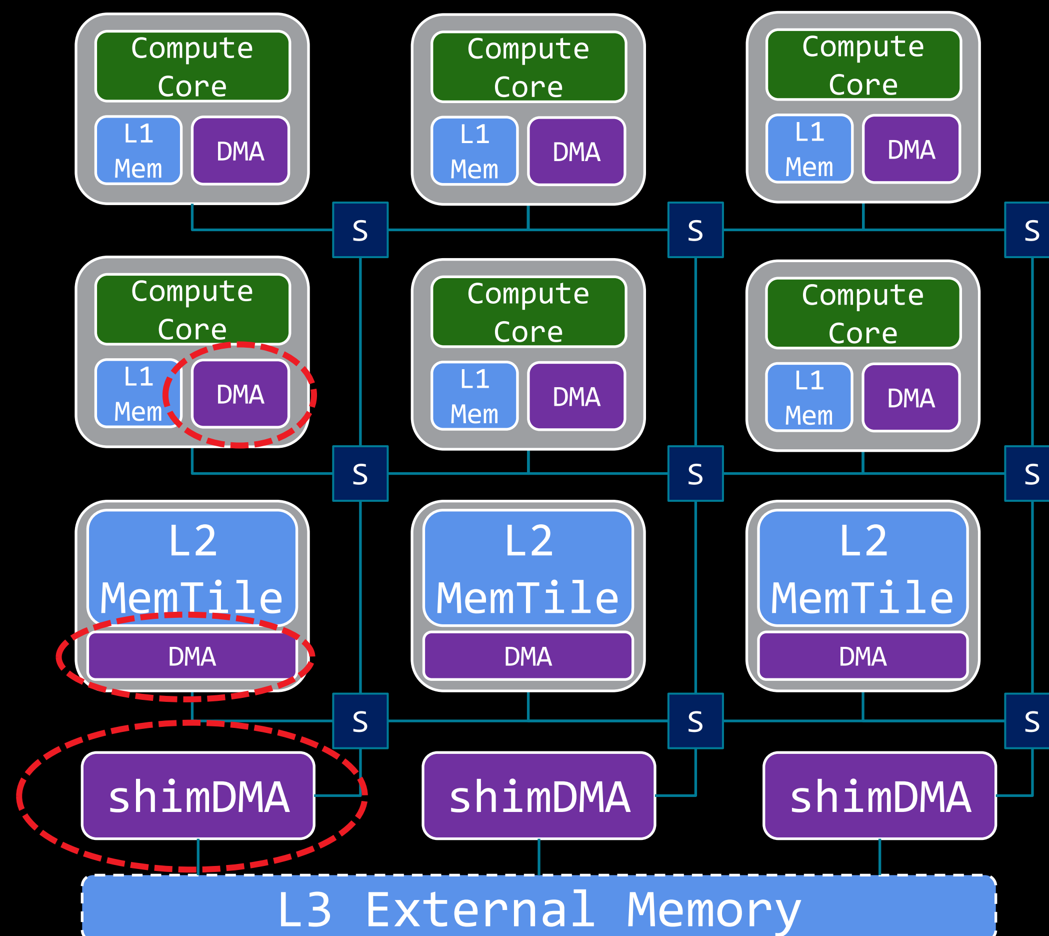


AMD
together we advance_

[Public]

Spatial Compute = Data Movement a Main Programming Task

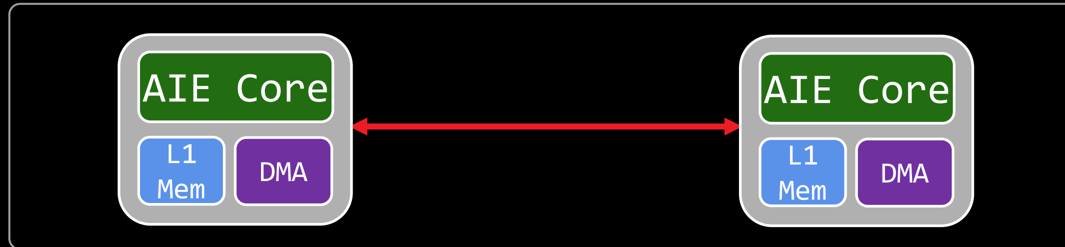
DMA Feature	Compute Tile DMA	MemTile DMA	ShimDMA
Max Addressing Dimension	3D	4D	3D
Zero-padding	N/A	Yes	N/A
Num Channels (R / W)	2 / 2	6 / 6	2 / 2
Num Buffer Descriptors	16	48	16
Num Semaphore Locks	16	64	16



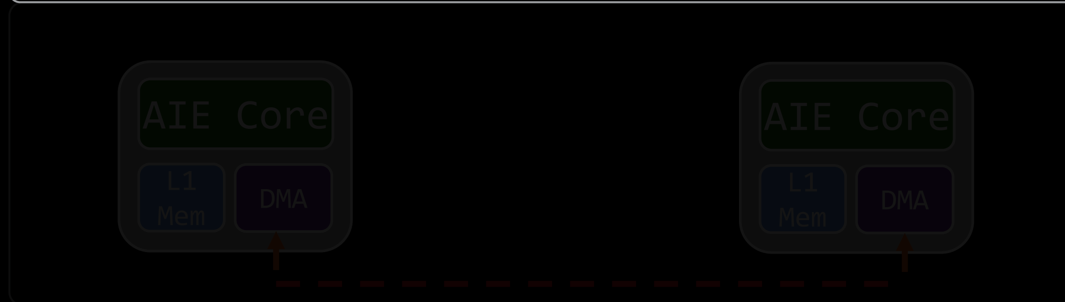
AMD
together we advance_

[Public]

Data Movement Scenarios



Peer Mem-To-Mem
Shared Memory



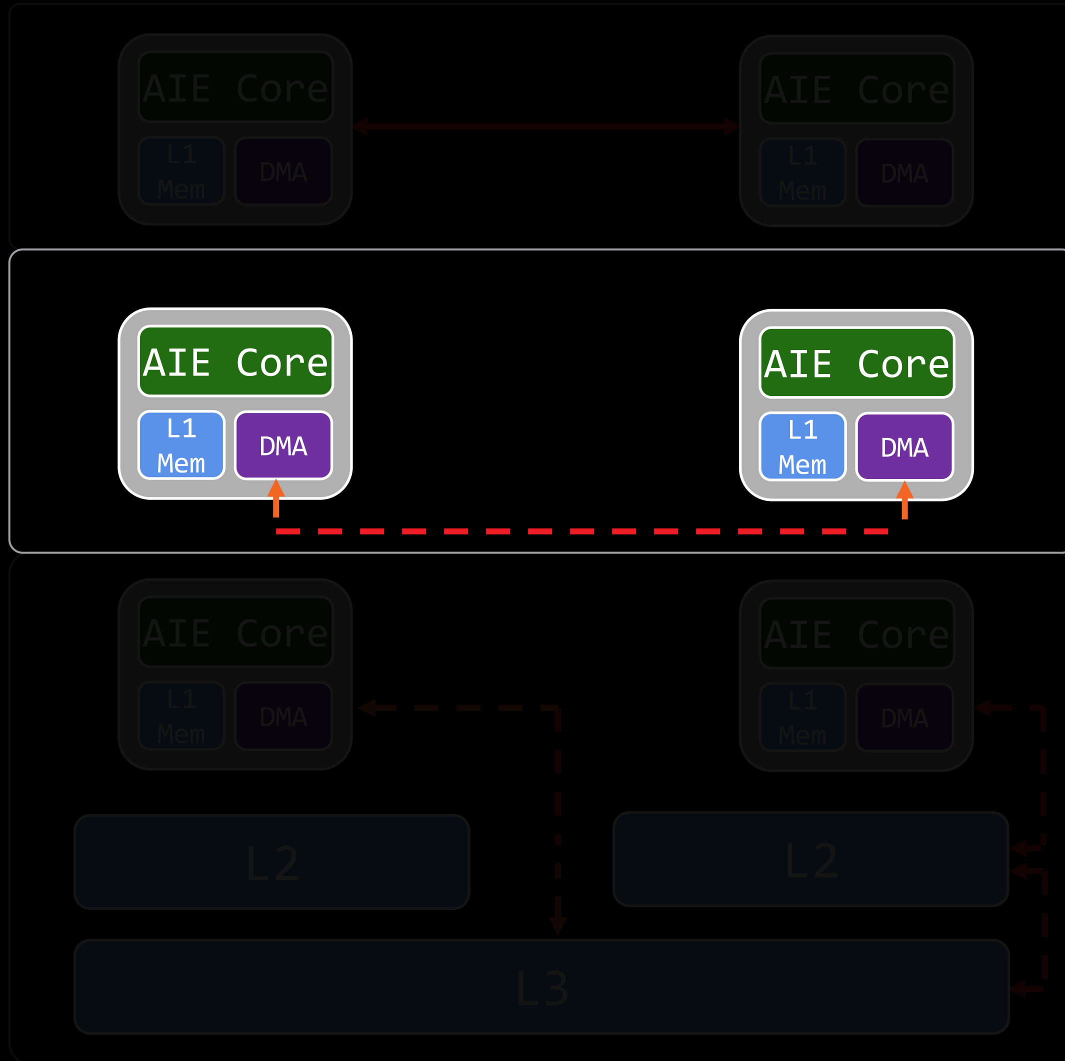
Peer Mem-To-Mem
DMAs + Routing

Hierarchical Mem-To-Mem
DMAs + Routing

AMD
together we advance_

[Public]

Data Movement Scenarios



Peer Mem-To-Mem
Shared Memory

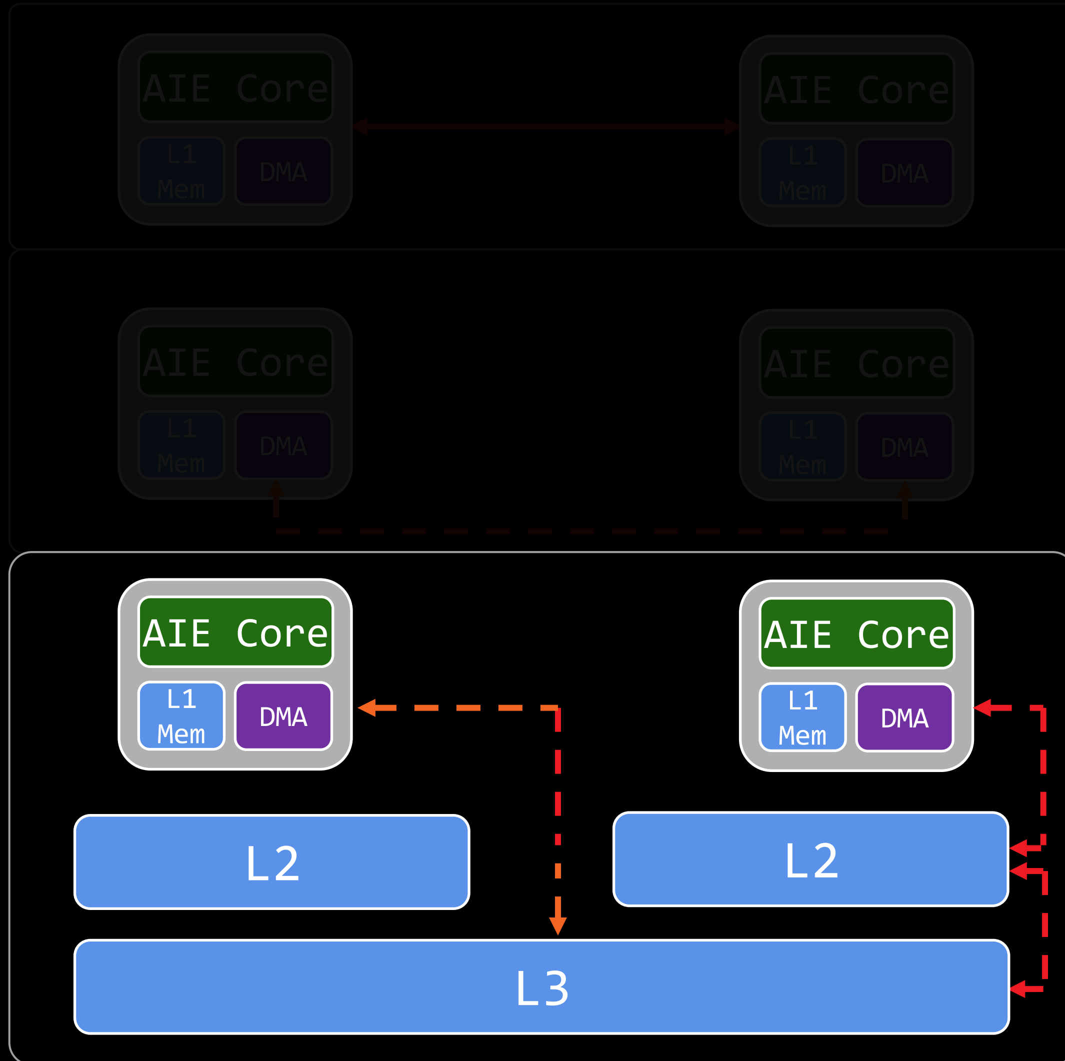
Peer Mem-To-Mem
DMAs + Routing

Hierarchical Mem-To-Mem
DMAs + Routing

AMD
together we advance_

[Public]

Data Movement Scenarios



Peer Mem-To-Mem
Shared Memory

Peer Mem-To-Mem
DMAs + Routing

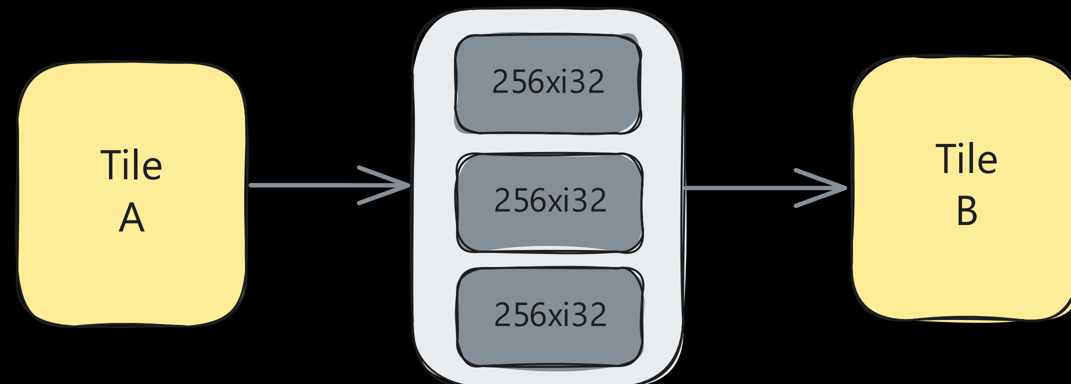
Hierarchical Mem-To-Mem
DMAs + Routing

AMD
together we advance_

[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive

```
class ObjectFifo:
        def __init__(
                self,
                obj_type,
                depth = 2,
                name,
        )
```

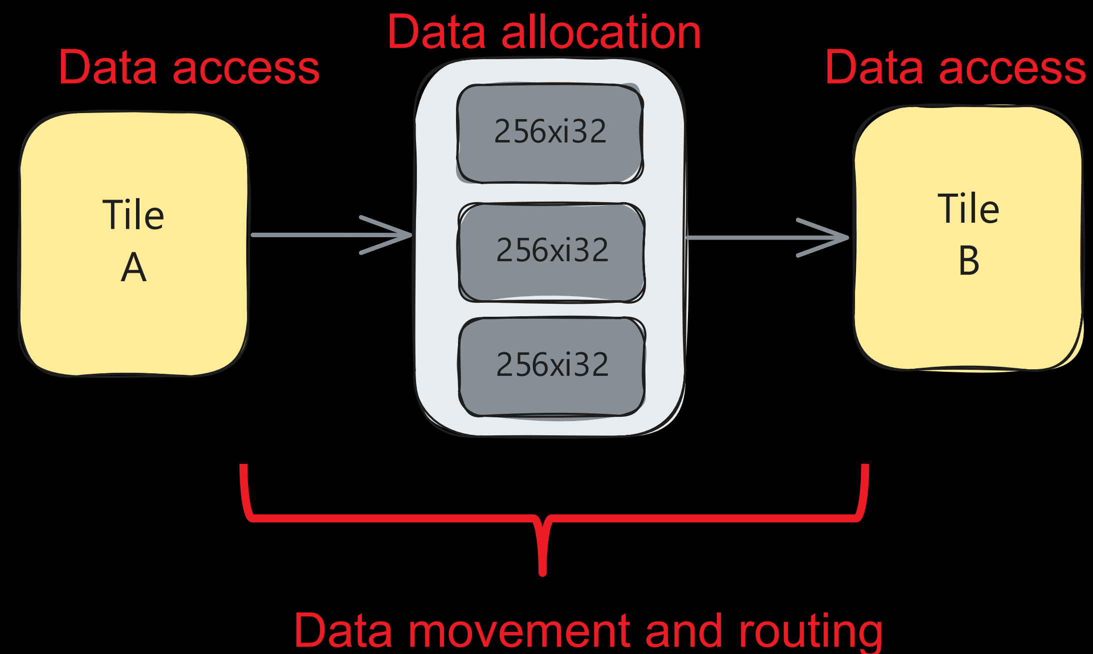


AMD
together we advance_

[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive

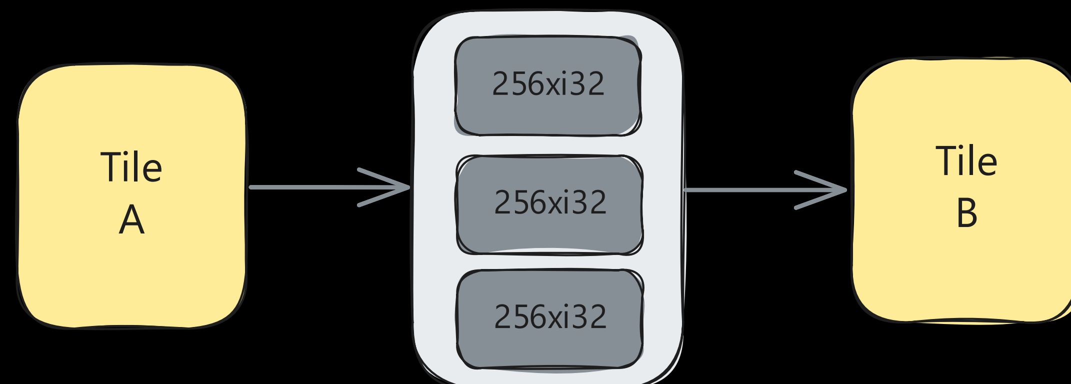
```
class ObjectFifo:
        def __init__(
                self,
                obj_type,
                depth = 2,
                name,
        )
```



AMD
together we advance_

[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive

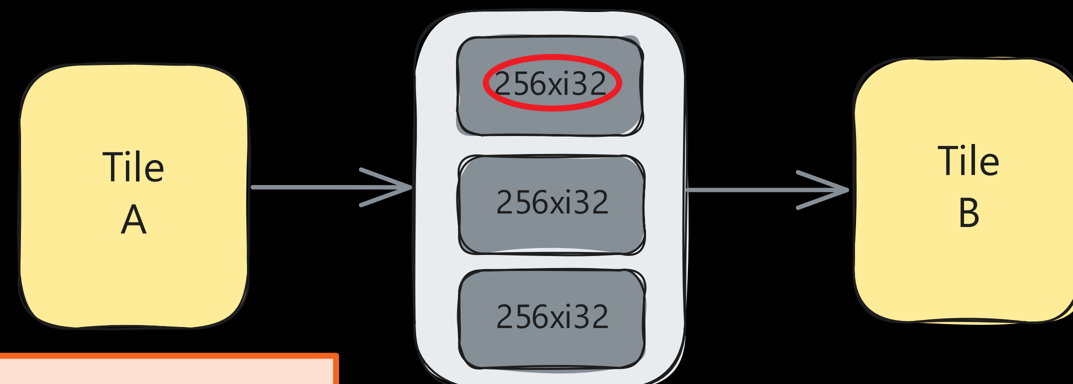



```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],3,"objfifo0")
A = Worker(..., fn_args=[of0.prod(),] )
B = Worker(..., fn_args=[of0.cons(),] )
```


AMD
together we advance_

[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive



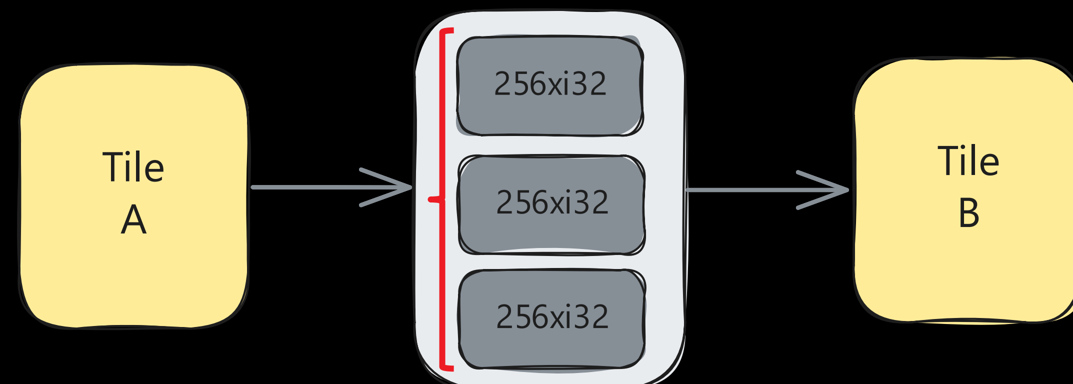
- Layout of an *ObjectFifo* object (tensor-like)

```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],3,"objfifo0")
A = Worker(..., fn_args=[of0.prod(),] )
B = Worker(..., fn_args=[of0.cons(),] )
```


AMD
together we advance_

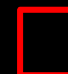
[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive



How many objects in a *ObjectFifo* (depth)

```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],3,"objfifo0")
A = Worker(..., fn_args=[of0.prod(),] )
B = Worker(..., fn_args=[of0.cons(),] )
```

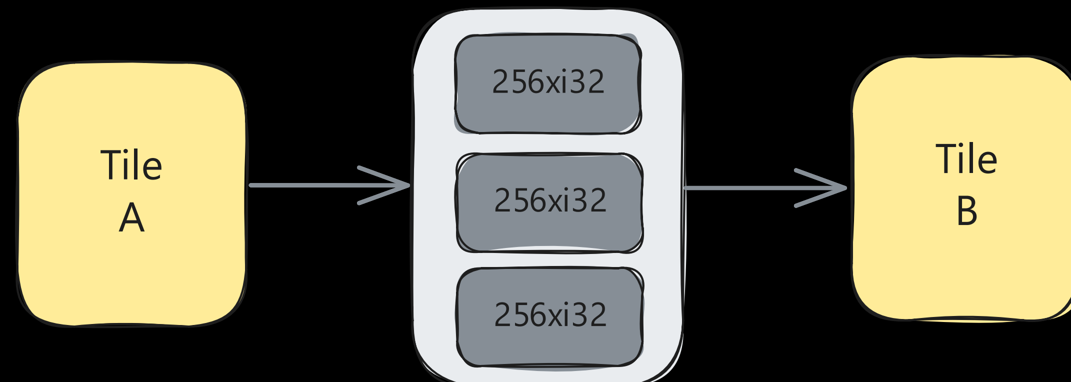


AMD
together we advance_

[Public]

ObjectFifo is the High-Level IRON Data Movement Primitive

```
class ObjectFifoHandle:
        def __init__(
                self,
                of: ObjectFifo
                is_prod: bool,
        )
```



Worker A produces objects
Worker B consumes objects

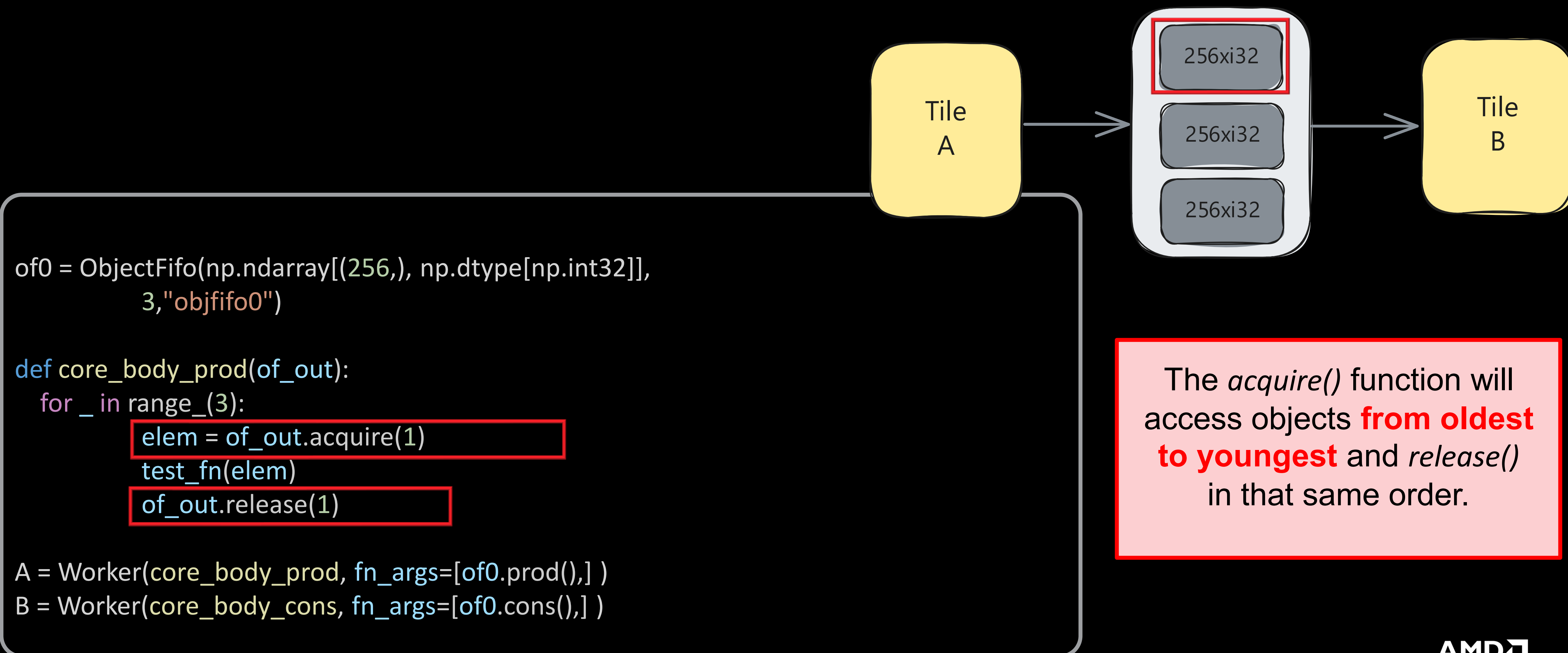
```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],3,"objfifo0")
A = Worker(..., fn_args=[of0.prod(),] )
B = Worker(..., fn_args=[of0.cons(),] )
```

ObjectFifoHandles

AMD
together we advance_

[Public]

Synchronized Accesses to the ObjectFifo Enables a Deadlock-Free Schedule



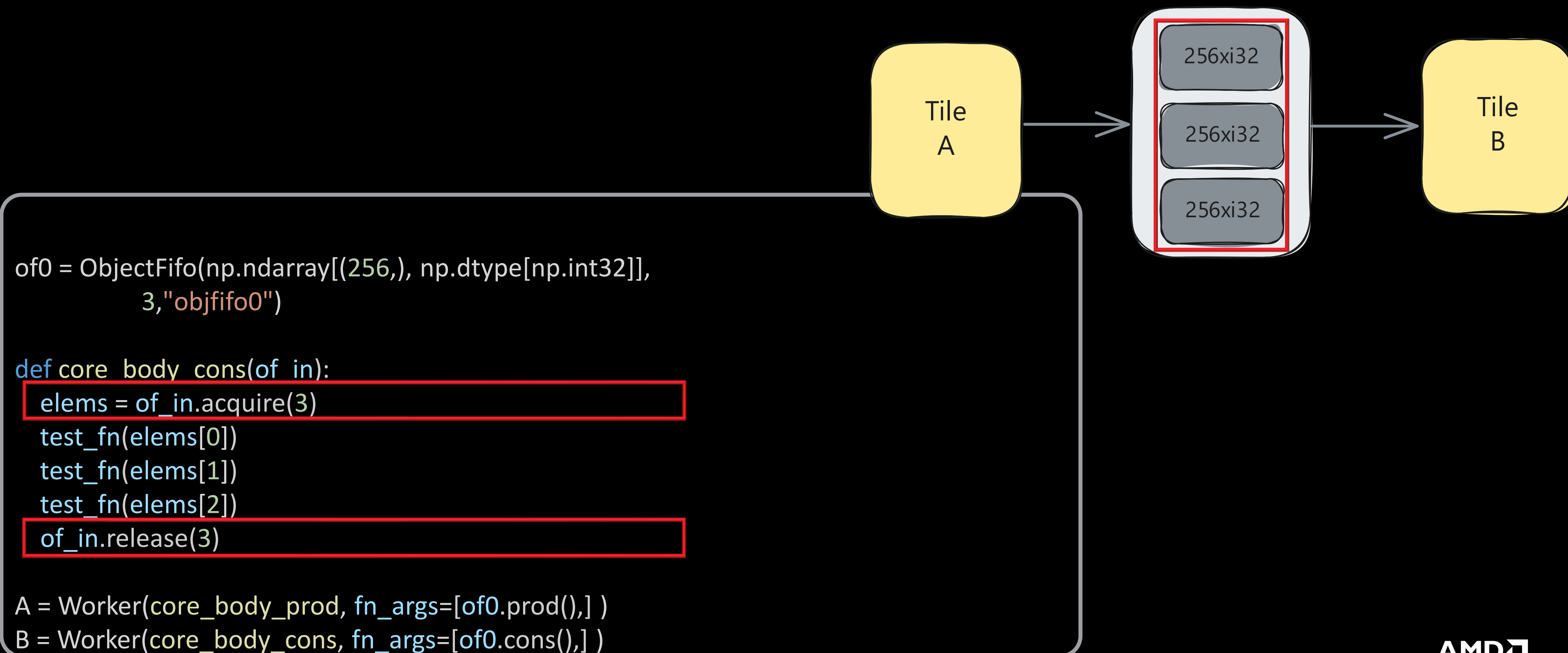

```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],
                 3,"objfifo0")

def core_body_prod(of_out):
   for _ in range_(3):
                elem = of_out.acquire(1)
                test_fn(elem)
                of_out.release(1)

A = Worker(core_body_prod, fn_args=[of0.prod(),] )
B = Worker(core_body_cons, fn_args=[of0.cons(),] )
```

The *acquire()* function will access objects **from oldest to youngest** and *release()* in that same order.

AMD
together we advance_

[Public]

Synchronized Accesses to the ObjectFifo Enables a Deadlock-Free Schedule

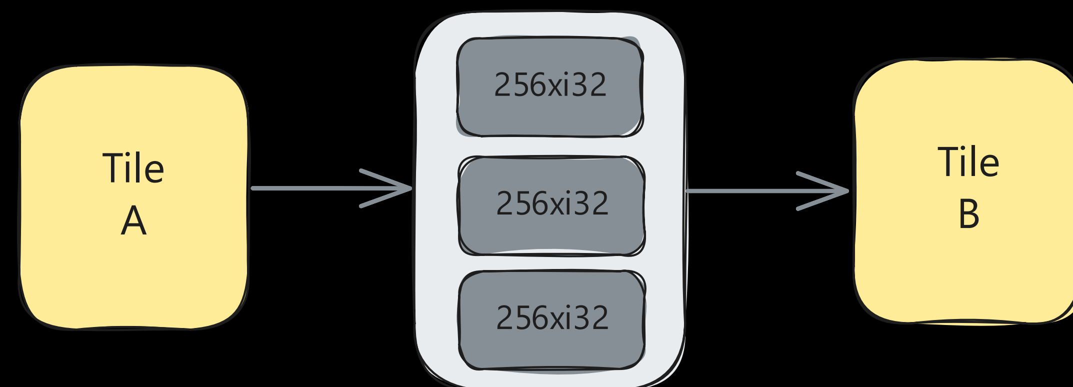


AMD
together we advance_

[Public]

Reusing Data in an ObjectFifo: Sliding Window

```
def core_body_cons(of_in):
    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)

    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)
    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(2)
```



Window

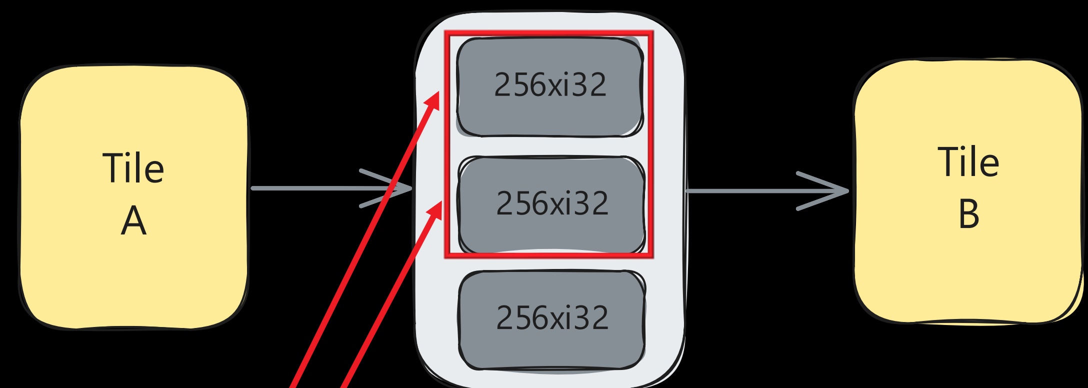
elems[0]

elems[1]

AMD
together we advance_

Reusing Data in an ObjectFifo: Sliding Window

```
def core_body_cons(of_in):
    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)

    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)
    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(2)
```



Window

elems[0]

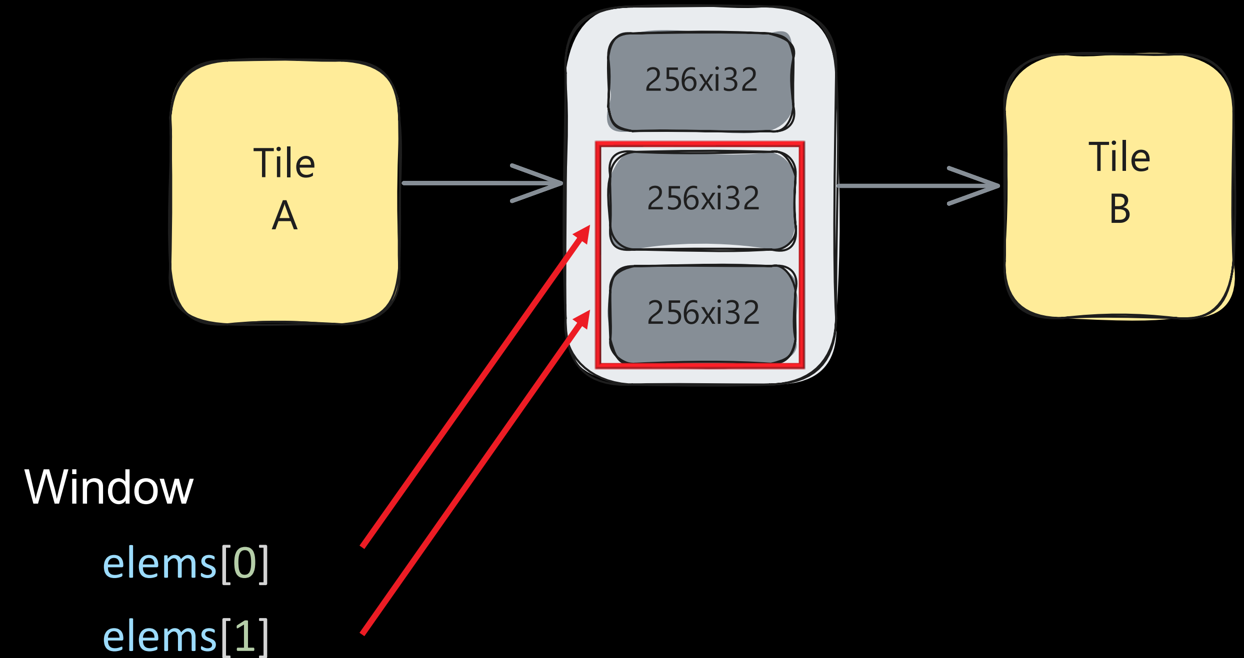
elems[1]

AMD
together we advance_

[Public]

Reusing Data in an ObjectFifo: Sliding Window

```
def core_body_cons(of_in):
    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)

    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)
    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(2)
```

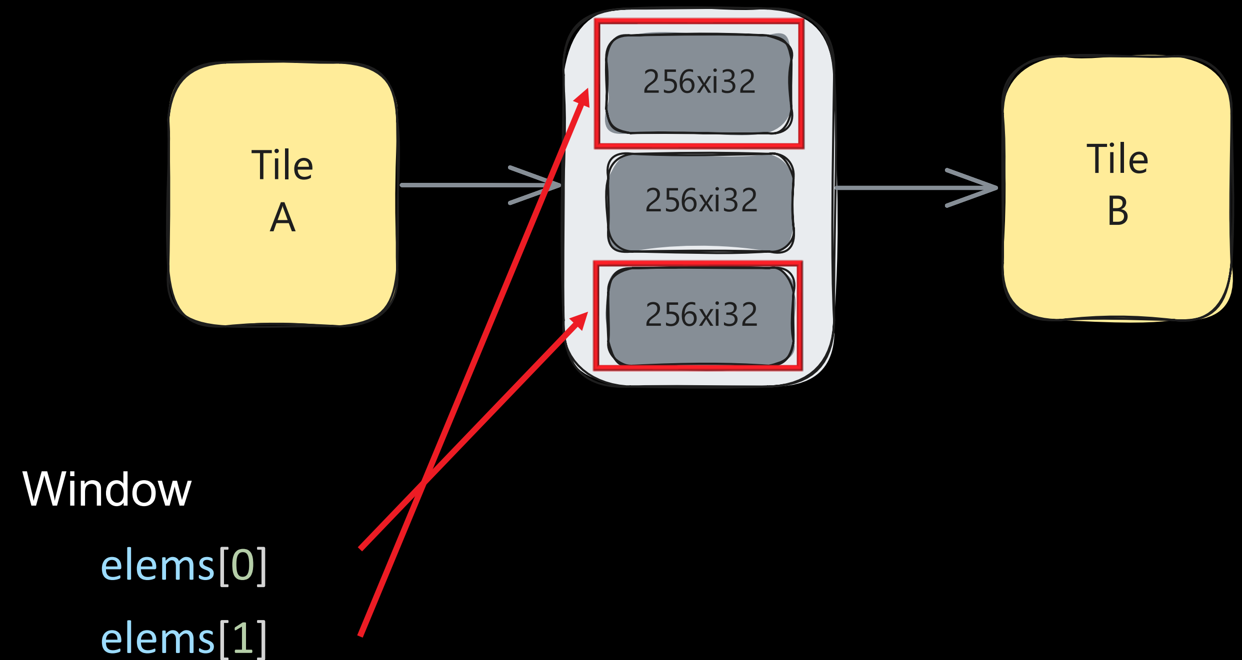


AMD
together we advance_

[Public]

Reusing Data in an ObjectFifo: Sliding Window

```
def core_body_cons(of_in):
    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)

    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(1)
    ...

    elems = of_in.acquire(2)
    test_fn(elems[0])
    test_fn(elems[1])
    of_in.release(2)
```



AMD
together we advance_

[Public]

Exercise 2: mini_tutorial Part 1

Navigate in your browser to https://github.com/Xilinx/mlir-aie/tree/main/programming_guide/mini_tutorial

- Contains a mini tutorial with exercises for learning IRON data movement concepts
- Follow the README in the mini_tutorial directory
- Do exercise 1; we'll come back later for the other exercises
- This example uses `iron.jit` for runtime
  - We have already configured the host-side buffer management for the input & the JIT signature for the function

AMD
together we advance_

Answer:

```python
def aie2p(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
    of_out = ObjectFifo(tile_ty, name="out")

    # Task for the core to perform
    def core_fn(of_in, of_out): # Use of_in rather than buff_in
        elem_in = of_in.acquire(1) # Acquire 1 object from of_in
        elem_out = of_out.acquire(1)
        for i in range_(num_elements):
            elem_out[i] = elem_in[i]
        of_in.release(1) # Release 1 object from of_in
        of_out.release(1)

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

    # To/from AIE-array runtime data movement
    rt = Runtime()
    with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
        rt.start(my_worker)
        rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
        rt.drain(of_out.cons(), c_out, wait=True)

def main():
    ...

    # Add input0 as an argument to the exercise_1 function call
    aie2p(input0, output)
```


AMD
together we advance_

Answer:

```
def aie2p(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
    of_out = ObjectFifo(tile_ty, name="out")

    # Task for the core to perform
    def core_fn(of_in, of_out): # Use of_in rather than buff_in
        elem_in = of_in.acquire(1) # Acquire 1 object from of_in
        elem_out = of_out.acquire(1)
        for i in range_(num_elements):
            elem_out[i] = elem_in[i]
        of_in.release(1) # Release 1 object from of_in
        of_out.release(1)

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

    # To/from AIE-array runtime data movement
    rt = Runtime()
    with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
        rt.start(my_worker)
        rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
        rt.drain(of_out.cons(), c_out, wait=True)

def main():
    ...

    # Add input0 as an argument to the exercise_1 function call
    aie2p(input0, output)
```


AMD
together we advance_

[Public]

Answer:

```
def aie2p(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
    of_out = ObjectFifo(tile_ty, name="out")

    # Task for the core to perform
    def core_fn(of_in, of_out): # Use of_in rather than buff_in
        elem_in = of_in.acquire(1) # Acquire 1 object from of_in
        elem_out = of_out.acquire(1)
        for i in range_(num_elements):
            elem_out[i] = elem_in[i]
        of_in.release(1) # Release 1 object from of_in
        of_out.release(1)

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

    # To/from AIE-array runtime data movement
    rt = Runtime()
    with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
        rt.start(my_worker)
        rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
        rt.drain(of_out.cons(), c_out, wait=True)

def main():
    ...

    # Add input0 as an argument to the exercise_1 function call
    aie2p(input0, output)
```


AMD
together we advance_

[Public]

Answer:

```python
def aie2p(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
    of_out = ObjectFifo(tile_ty, name="out")

    # Task for the core to perform
    def core_fn(of_in, of_out): # Use of_in rather than buff_in
        elem_in = of_in.acquire(1) # Acquire 1 object from of_in
        elem_out = of_out.acquire(1)
        for i in range_(num_elements):
            elem_out[i] = elem_in[i]
        of_in.release(1) # Release 1 object from of_in
        of_out.release(1)

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

    # To/from AIE-array runtime data movement
    rt = Runtime()
    with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
        rt.start(my_worker)
        rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
        rt.drain(of_out.cons(), c_out, wait=True)

def main():
    ...

    # Add input0 as an argument to the exercise_1 function call
    aie2p(input0, output)
```


AMD
together we advance_

[Public]

Answer:

```
def aie2p(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
    of_out = ObjectFifo(tile_ty, name="out")

    # Task for the core to perform
    def core_fn(of_in, of_out): # Use of_in rather than buff_in
        elem_in = of_in.acquire(1) # Acquire 1 object from of_in
        elem_out = of_out.acquire(1)
        for i in range_(num_elements):
            elem_out[i] = elem_in[i]
        of_in.release(1) # Release 1 object from of_in
        of_out.release(1)

    # Create a worker to perform the task
    my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

    # To/from AIE-array runtime data movement
    rt = Runtime()
    with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
        rt.start(my_worker)
        rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
        rt.drain(of_out.cons(), c_out, wait=True)

def main():
    ...

    # Add input0 as an argument to the exercise_1 function call
    aie2p(input0, output)
```


AMD
together we advance_

Answer:

```
def aie2p(input, output):
  ...
  # Dataflow with ObjectFifos
  of_in = ObjectFifo(tile_ty, name="in") # Add input ObjectFifo
  of_out = ObjectFifo(tile_ty, name="out")

  # Task for the core to perform
  def core_fn(of_in, of_out): # Use of_in rather than buff_in
    elem_in = of_in.acquire(1) # Acquire 1 object from of_in
    elem_out = of_out.acquire(1)
    for i in range_(num_elements):
      elem_out[i] = elem_in[i]
    of_in.release(1) # Release 1 object from of_in
    of_out.release(1)

  # Create a worker to perform the task
  my_worker = Worker(core_fn, [of_in.cons(), of_out.prod()]) # Use of_in rather than buff_in

  # To/from AIE-array runtime data movement
  rt = Runtime()
  with rt.sequence(tile_ty, tile_ty) as (a_in, c_out): # Add input buffer to runtime sequence
    rt.start(my_worker)
    rt.fill(of_in.prod(), a_in) # Fill of_in from a_in input
    rt.drain(of_out.cons(), c_out, wait=True)

def main():
  ...

  # Add input0 as an argument to the exercise_1 function call
  aie2p(input0, output)
```


AMD
together we advance_

Your First Complete Program

[Public]

Runtime Support

More Abstraction

```
...
@iron.jit(is_placed=False)
def example(input):
   ...
```

JIT Runtime Support

```
import pyxrt
```

```
#include "xrt/xrt_bo.h"
#include "xrt/xrt_device.h"
#include "xrt/xrt_kernel.h"
```

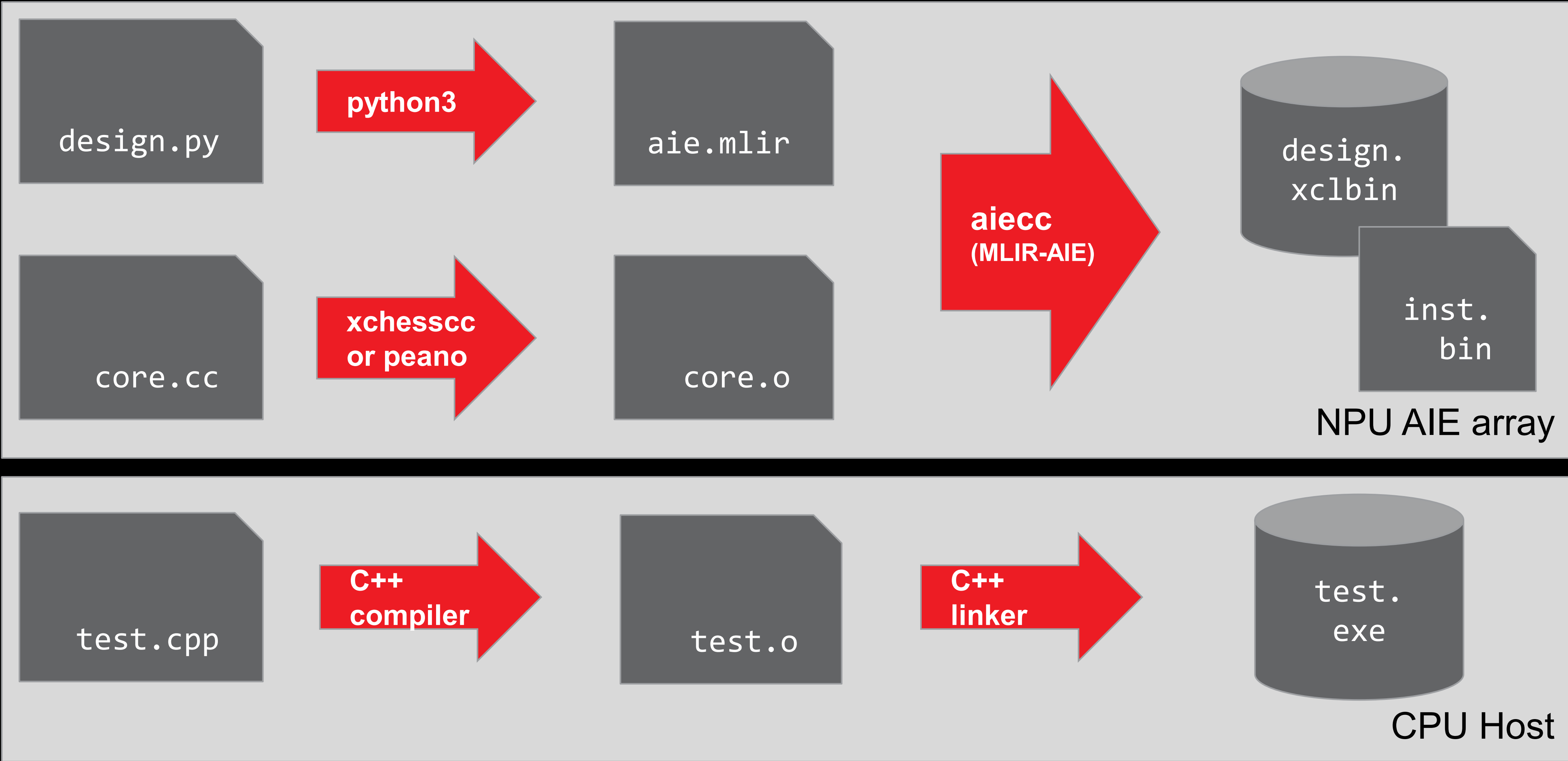
```
from aie.utils import DefaultNPURuntime
```

Separate Test Programs + Makefiles (python or C++)

Less Abstraction

AMD
together we advance_

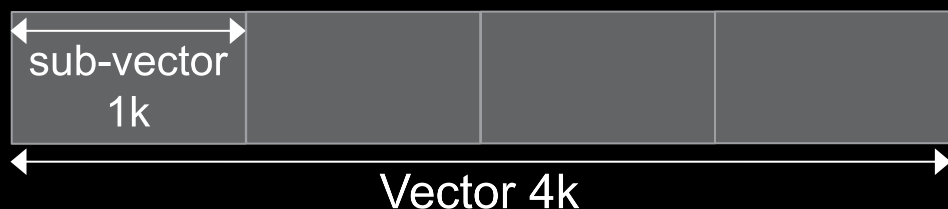
Understanding the SDK: From Source Code to Binaries



AMD
together we advance_

[Public]

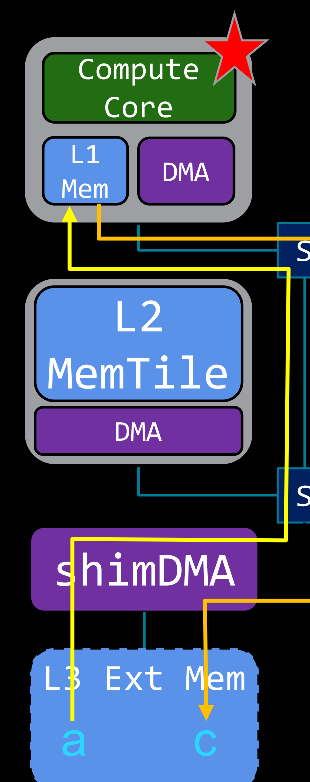
vector_scalar_mul: Data Movement and Compute Steps
c = a*factor



- Read "a" data from external memory and
- Push data onto streaming interconnect with Shim DMA
- Pull data off streaming interconnect with AIE tile DMA

- Vector scale operation in the AIE Core

- Push data from L1 tile memory onto streaming interconnect with AIE tile DMA
- Pull data off streaming interconnect with Shim DMA
- Write "c" back to DDR

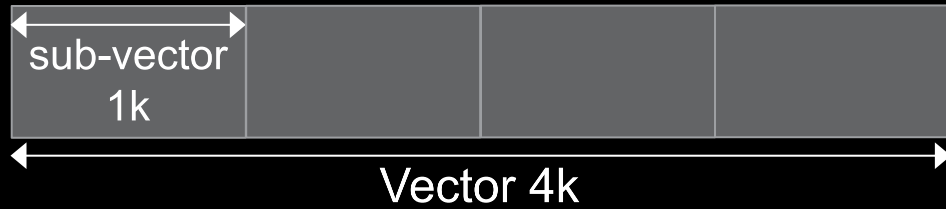


AMD
together we advance_

[Public]

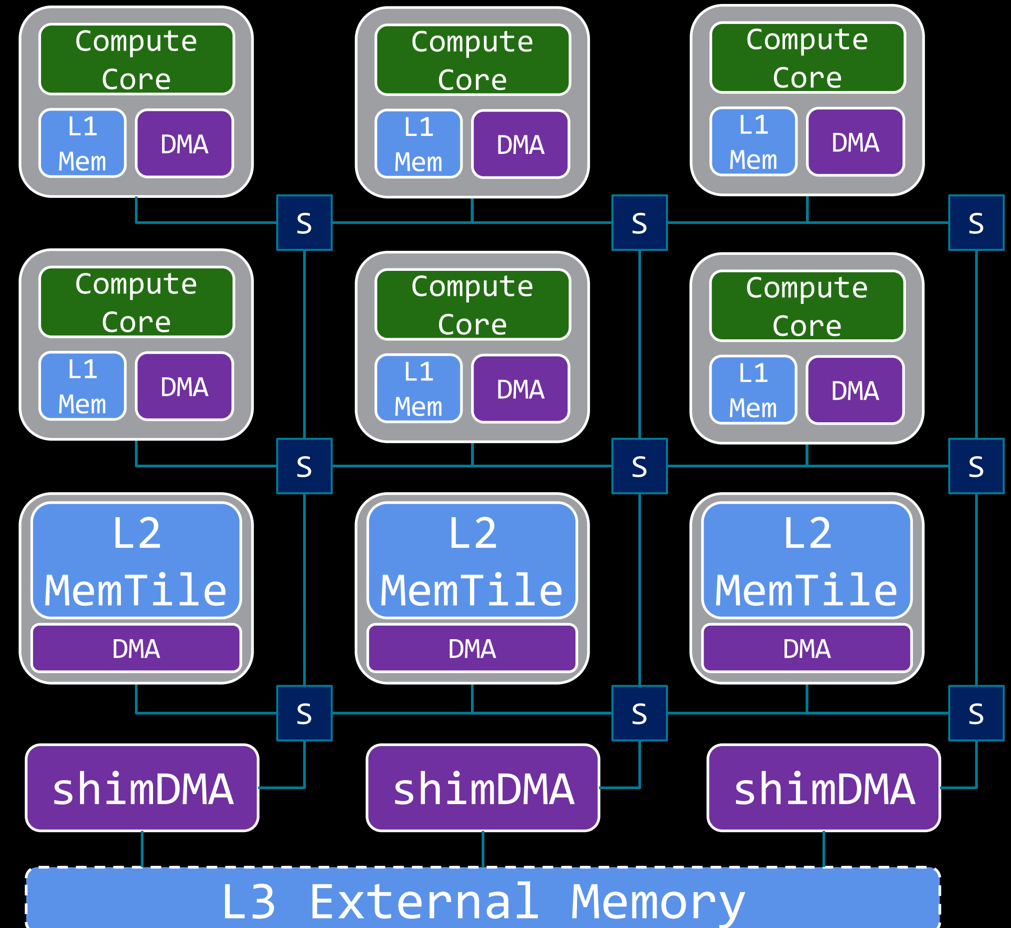
vector_scalar_mul

c = a*factor



AIE Core Code (Scalar Version)

```
void scale_scalar(int32_t *a, int32_t *c,
             int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```

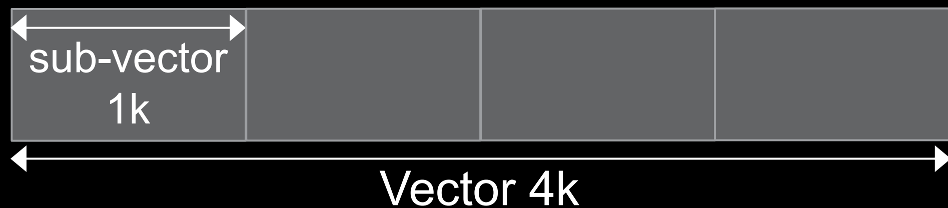


AMD
together we advance_

[Public]

vector_scalar_mul

c = a*factor



AIE Core Code (Scalar Version)

```
void scale_scalar(int32_t *a, int32_t *c,
            int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```

AIE Array Code

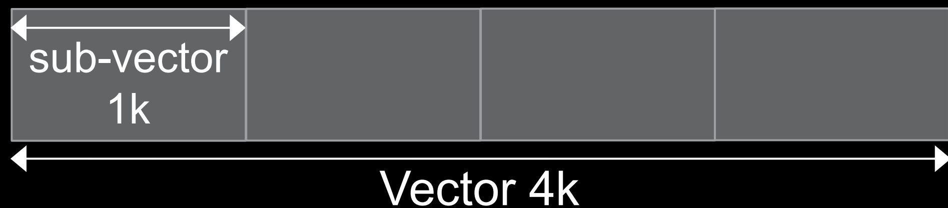
```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar”, “scale.o",
            [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
            elem_in = of_in.acquire(1)
            elem_out = of_out.acquire(1)
            scale_fn(elem_in, elem_out, elem_factor, n)
            of_in.release(1)
            of_out.release(1)
    of_factor.release(1)

worker = Worker(core_body, fn_args=[of_in.cons(), of_factor.cons(),
                    of_out.prod(), scale])

rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
        rt.start(worker)
        rt.fill(of_in.prod(), A)
        rt.fill(of_factor.prod(), F)
        rt.drain(of_out.cons(), C, wait=True)

return Program(dev, rt).resolve_program(SequentialPlacer())
```


AMD
together we advance_

[Public]

vector_scalar_mul

c = a*factor



AIE Core Code (Scalar Version)

```
void scale_scalar(int32_t *a, int32_t *c,
             int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```

AIE Array Code

```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar", "scale.o",
          [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
            elem_in = of_in.acquire(1)
            elem_out = of_out.acquire(1)
            scale_fn(elem_in, elem_out, elem_factor, n)
            of_in.release(1)
            of_out.release(1)
    of_factor.release(1)

worker = Worker(core_body, fn_args=[of_in.cons(), of_factor.cons(),
                    of_out.prod(), scale])

rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
        rt.start(worker)
        rt.fill(of_in.prod(), A)
        rt.fill(of_factor.prod(), F)
        rt.drain(of_out.cons(), C, wait=True)

return Program(dev, rt).resolve_program(SequentialPlacer())
```

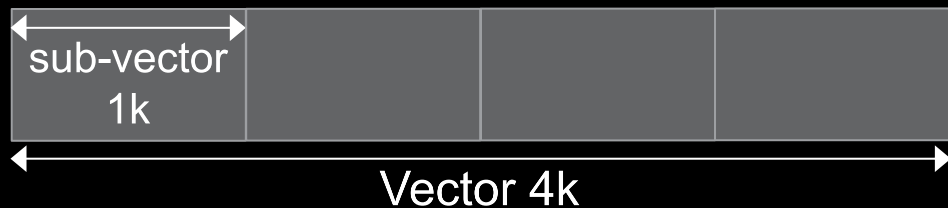
Resources and Declarations

AMD
together we advance_

[Public]

vector_scalar_mul

c = a*factor



AIE Core Code (Scalar Version)

```
void scale_scalar(int32_t *a, int32_t *c,
              int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```

AIE Array Code

```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar", "scale.o",
           [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
              elem_in = of_in.acquire(1)
              elem_out = of_out.acquire(1)
              scale_fn(elem_in, elem_out, elem_factor, n)
              of_in.release(1)
              of_out.release(1)
    of_factor.release(1)

worker = Worker(core_body, fn_args=[of_in.cons(), of_factor.cons(),
                      of_out.prod(), scale])

rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
        rt.start(worker)
        rt.fill(of_in.prod(), A)
        rt.fill(of_factor.prod(), F)
        rt.drain(of_out.cons(), C, wait=True)

return Program(dev, rt).resolve_program(SequentialPlacer())
```

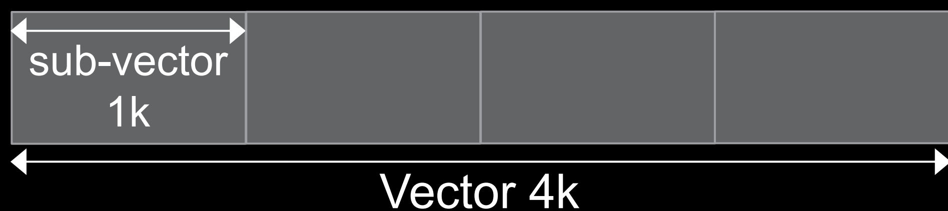
Compute Core Definitions

AMD
together we advance_

[Public]

vector_scalar_mul

c = a*factor



AIE Core Code (Scalar Version)

```
void scale_scalar(int32_t *a, int32_t *c,
            int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```

AIE Array Code

```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar", "scale.o",
           [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
            elem_in = of_in.acquire(1)
            elem_out = of_out.acquire(1)
            scale_fn(elem_in, elem_out, elem_factor, n)
            of_in.release(1)
            of_out.release(1)
    of_factor.release(1)

worker = Worker(core_body, fn_args=[of_in.cons(), of_factor.cons(),
                    of_out.prod(), scale])

rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
        rt.start(worker)
        rt.fill(of_in.prod(), A)
        rt.fill(of_factor.prod(), F)
        rt.drain(of_out.cons(), C, wait=True)

return Program(dev, rt).resolve_program(SequentialPlacer())
```

Runtime Sequence

AMD
together we advance_

[Public]

Exercise 3: vector_scalar_mul

- Navigate to Section 3 of our Programming Guide
  - (https://github.com/Xilinx/mlir-aie/tree/main/programming_guide/section-3)
  - On your local mlir-aie install, change the directory to mlir-aie/programming_guide/section-3 folder

Steps:

1. Follow the instructions in Section 3 for building and running this design
2. Familiarize yourself with the binaries generated and the additional source files
3. Challenge: Modify the example to add a scalar to the vector rather than multiply
   - Do not forget to modify the host code verification!

AMD
together we advance_

Tracing and Performance Analysis

[Public]

Fast and Efficient NPU Designs

- Measuring performance is key to building efficient designs
- Different ways to measure performance:
  - Latency
  - Throughput
  - Energy efficiency
  - Hardware utilization
- Primary methods used in IRON:
  - Timers
  - Tracing
- Area of ongoing research & development

AMD
together we advance_

[Public]

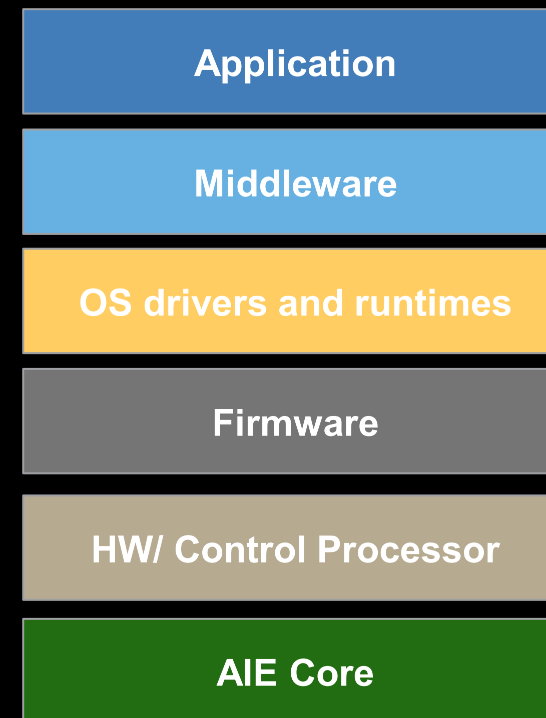
Fast and Efficient NPU Designs with Timers

"Wall clock" time is a helpful metric to measure the overall performance of an application

- Gives upper bounds AIE application
  - Include software stack overhead like OS, kernel drivers, and communication overheads

```
auto start = std::chrono::high_resolution_clock::now();
auto run = kernel(bo_instr, instr_v.size(), bo_inout0, bo_inout1, bo_inout2);
run.wait();
auto stop = std::chrono::high_resolution_clock::now();

bo_inout2.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

Application Time



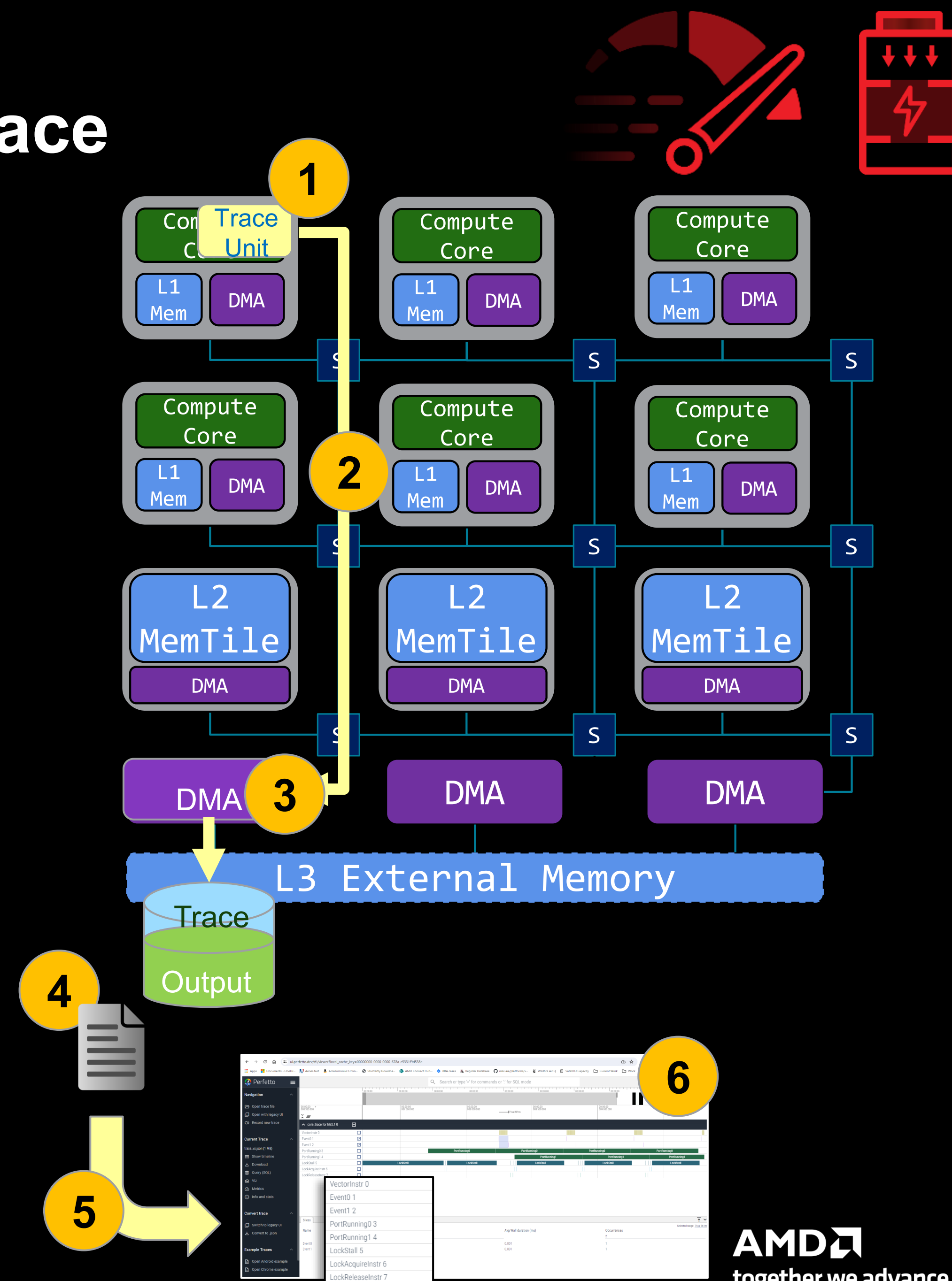
AMD
together we advance_

[Public]

Fast and Efficient NPU Designs with Trace

- Visualize cycle-accurate low-level AIE events
- What you can trace (over 100 events for core tile)
  - Kernel start (event0) / Kernel end (event1)
  - Data movement: active port transfers
  - Stalls: due to memory/locks/stream backpressure
  - Vector instructions
  - Complete list: https://xilinx.github.io/mlir-aie/AIEXDialect.html (search EventAIE)
- How to enable trace in IRON
  1. Configure trace units (events)
  2. Route trace packets through stream switches
  3. Configure Shim DMA to write trace packets to DDR
  4. Read buffer and write to file
  5. Parse file to generate JSON waveform file
  6. View json file in a visualizer (e.g., Perfetto)

IRON makes trace integration easy and scalable!



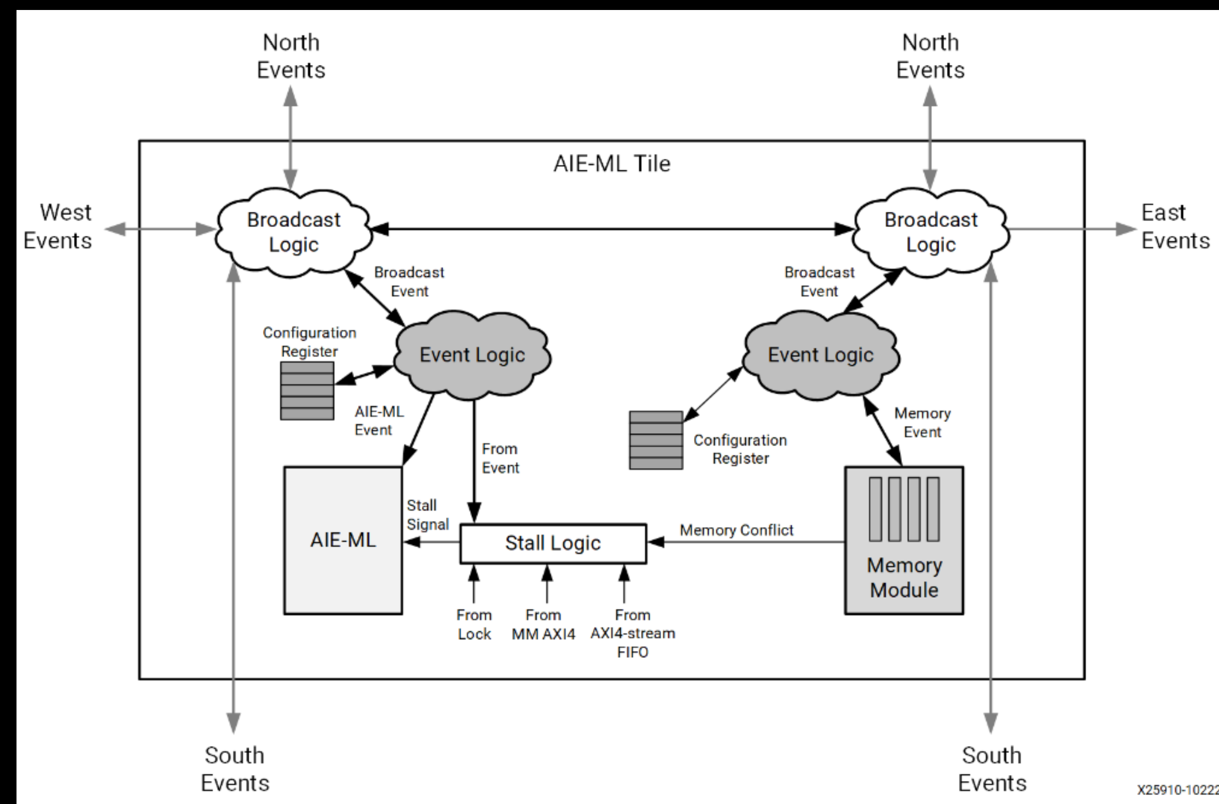
AMD
together we advance_

[Public]

NPU Trace Units

- What are trace units?
  - Dedicated hardware blocks across all tile types:
    - Core, L1 Memory, MemTile, ShimTile
  - Can monitor up to 8 user-selected events (from 100+)
  - Generate trace packets based on events

- Trace has minimal performance impact
  - Monitoring and packet generation are non-intrusive
  - But:
    - Packet routing uses stream switch resources
    - Data movement to DDR (L3) uses ShimDMA channels



AMD
together we advance_

[Public]

Enabling Trace in IRON: Code Integration & Dataflow (1/2)

IRON code to enable trace (unplaced design)

aie2.py

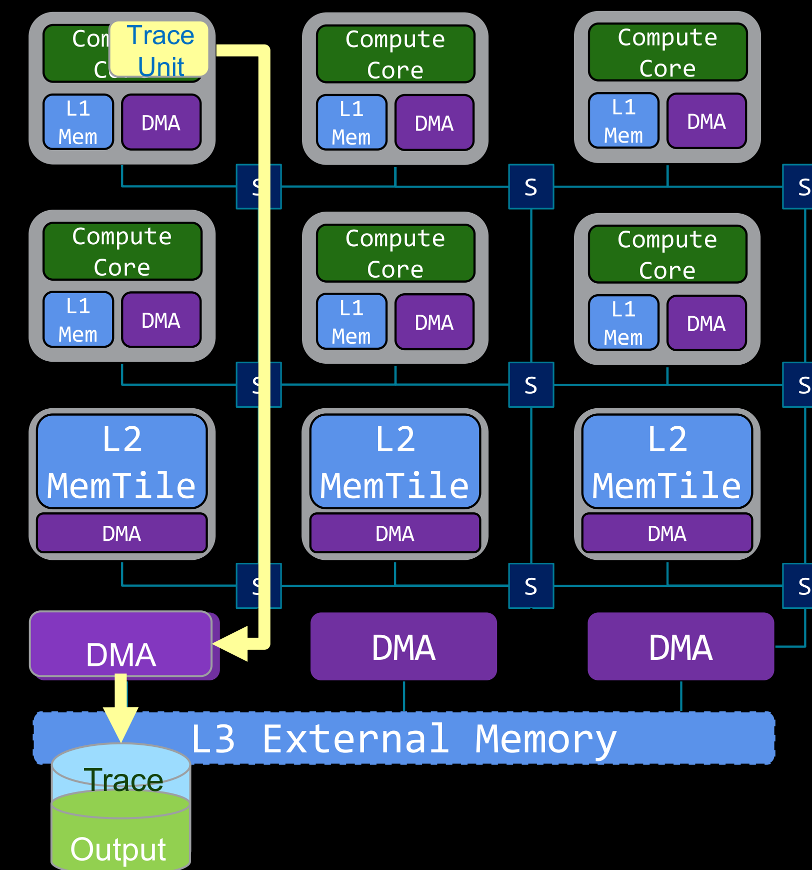
```
my_workers = []
for i in range(n_cores):
    workers.append(
        Worker(...
...

rt = Runtime()
with rt.sequence(tensor_ty, tensor_ty) as (A, C):
    rt.enable_trace(trace_size, workers=my_workers)
```

Configure trace for all workers in array

test.cpp

```
...
if (trace_size > 0) {
    test_utils::write_out_trace(
        ((char *)bufOut) + IN_SIZE,
        trace_size, vm["trace_file"]
        .as<std::string>());
}
```

Writes trace data to file



AMD
together we advance_

[Public]

Enabling Trace in IRON: Code Integration & Dataflow (2/2)

IRON code to enable trace (unplaced design) with selected events

aie2.py

```
...

rt = Runtime()
with rt.sequence(tensor_ty, tensor_ty) as (A, C):
    rt.enable_trace(
        trace_size=trace_size,
        workers=my_workers,
        trace_offset= trace_offset,
        coretile_events = [
            trace_utils.CoreEvent.INSTR_EVENT_0,
            trace_utils.CoreEvent.INSTR_EVENT_1,
            trace_utils.CoreEvent.INSTR_VECTOR,
            trace_utils.CoreEvent.MEMORY_STALL,
            trace_utils.CoreEvent.STREAM_STALL,
            trace_utils.CoreEvent.LOCK_STALL,
            trace_utils.CoreEvent.ACTIVE,
            trace_utils.CoreEvent.DISABLED]
    )
    ...
```

Config Trace and ShimDMA

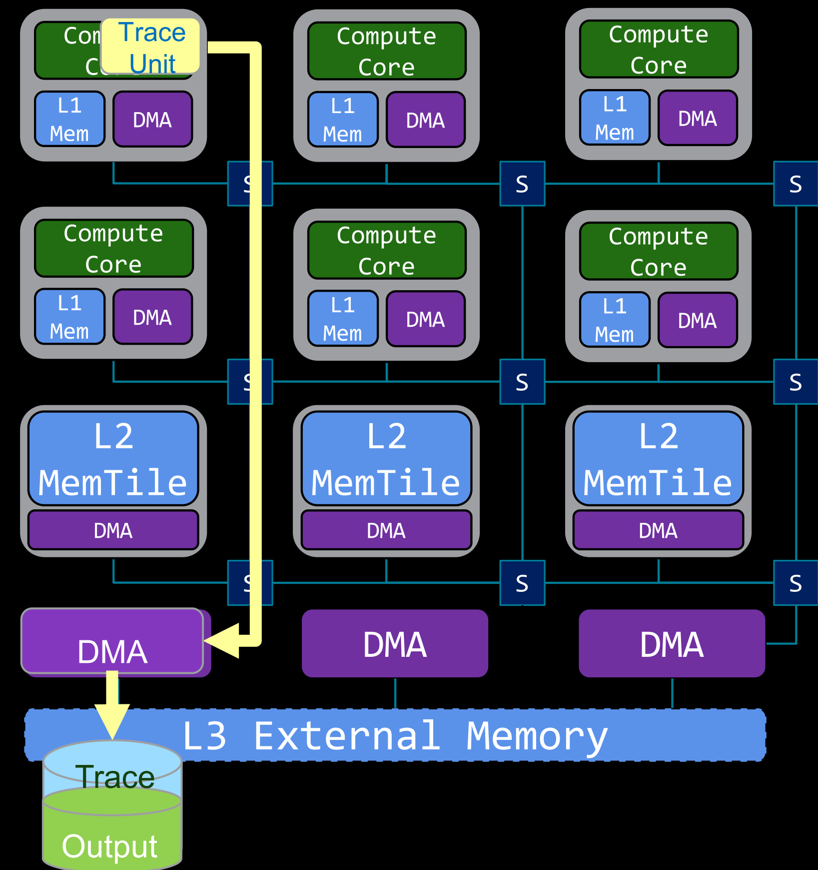
Buffer offset

XRT buffer select

Selected events

test.cpp

```
...
```

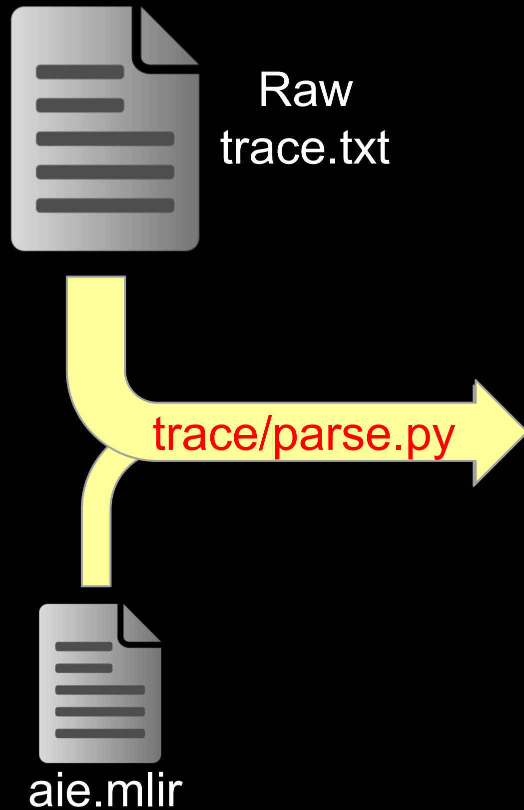


AMD
together we advance_

[Public]

Visualizing Trace with Perfetto

- Step 1: Use trace parser (trace/parse.py) to convert raw trace.txt file to waveform JSON (trace.json)
- Step 2: View results (trace.json) in web browser (http://ui.perfetto.dev)



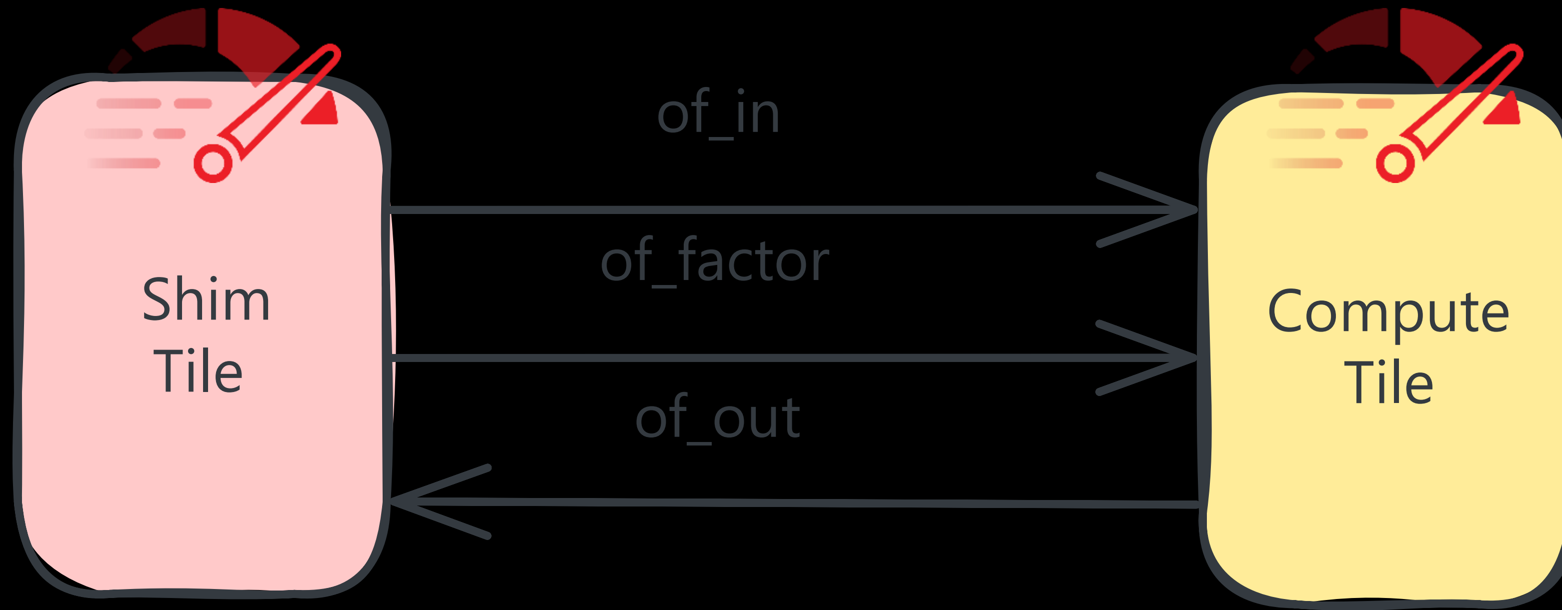
VectorInstr 0
Event0 1
Event1 2
PortRunning0 3
PortRunning1 4
LockStall 5
LockAcquireInstr 6
LockReleaseInstr 7

NOTE: units in microseconds should be interpreted as clock cycles

AMD
together we advance_

[Public]

Exercise 4: Measuring Performance with vector_scalar_mul

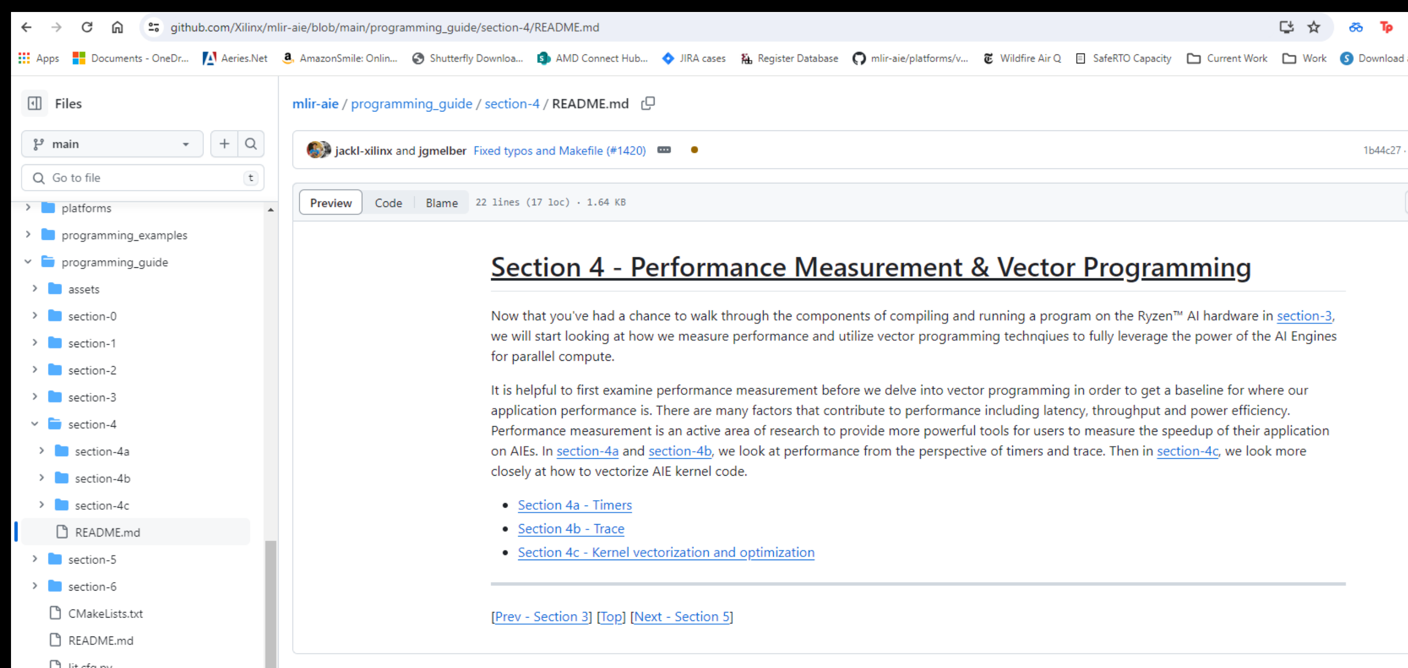


AMD
together we advance_

Running Timers and Trace Exercises on Hardware

Navigate to Section 4 of the Programming Guide

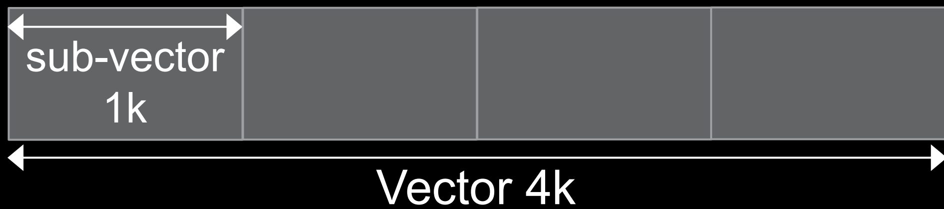
- https://github.com/Xilinx/mlir-aie/tree/main/programming_guide/section-4
- On your local mlir-aie install, change directory to mlir-aie/programming_guide/section-4/section-4b folder
- Follow the instructions in Section 4b
- Note: use "*make use_placed=1 trace*" for this example



AMD
together we advance_

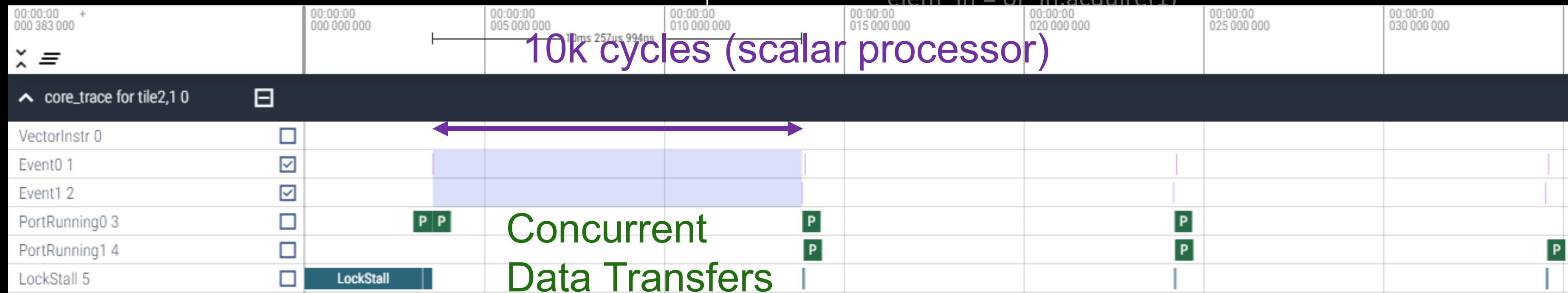
vector_scalar_mul

c = a*factor



AIE Array Code

```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar", "scale.o",
          [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
            elem_in = of_in.acquire(1)
```




```
rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
        rt.start(worker)
        rt.fill(of_in.prod(), A)
        rt.fill(of_factor.prod(), F)
        rt.drain(of_out.cons(), C, wait=True)

return Program(dev, rt).resolve_program(SequentialPlacer())
```

AIE Core Code (Scalar Version)

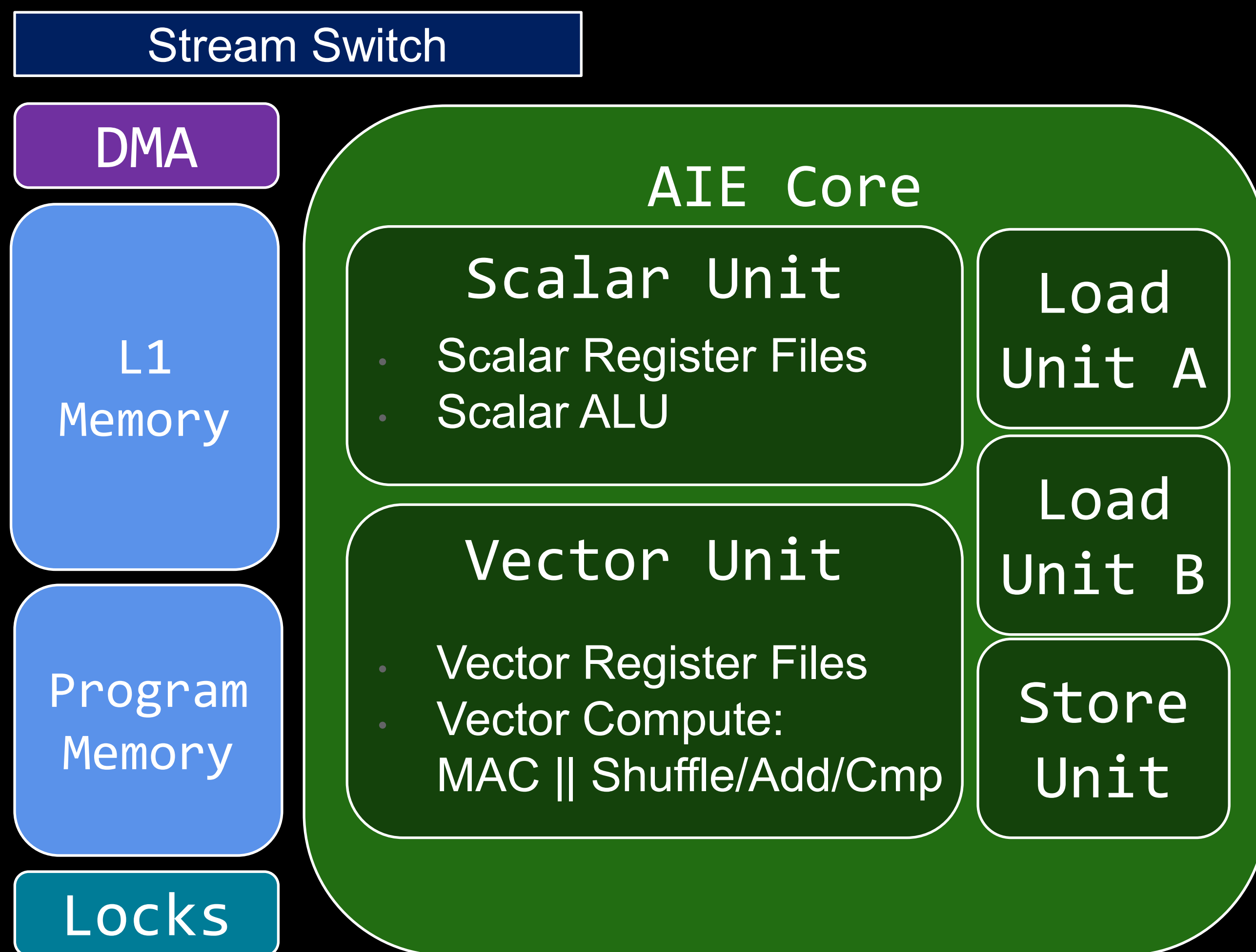
```
void scale_scalar(int32_t *a, int32_t *c,
          int32_t factor) {

 for (int i = 0; i < 1024; i++) {
  c[i] = factor * a[i];
 }
}
```


AMD
together we advance_

Vectorizing on AIE

[Public]

AIE Compute Tile: Compute Details



Precision 1	Precision 2	Acc Lanes	Bits/Acc Lane	MACs
int 8	int 4	32	32	512
int 8	int 8	32	32	256
int 16	int 8	32	32	128
int 16	int 16	32	32	64
int 32	int 16	16	64	32
int 32 [1]	int 32	16	64	16
bfloat 16 [3]	bfloat 16	16	SPFP 32 [2]	128

1.int32 x int32 can be emulated. The operation should have half the performance of int32 x int16 and there should be 16 multiplications per cycle.
2.Single precision floating point (SPFP) per the IEEE standard.
3.float32 x float32 can be emulated. Emulation deviates from the IEEE-754 standard.

AIE-ML AIE API Manual: https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aie_api/doc/

AMD
together we advance_

AIE Compute Tile
Zero-Overhead SW Programmable VLIW Vector Processor

```
void processing_int32(int32_t *in, int32_t *out, int32_t parameter) {

 for (int i = 0; i < iLoopBound; i++) {
  for (int j = 0; j < jLoopBound; j++) {
   aie::vector< int32_t, 16> vectorOfData = aie::load_v<16>(in++);
   aie::accum<acc64, vec_factor> cout =
                        aie::vectorProcessing(vectorOfData, parameter);
   aie::store_v(out++,cout);
  }
 }

}
```

Zero counter overhead on nested loop

Zero pointer increment overhead

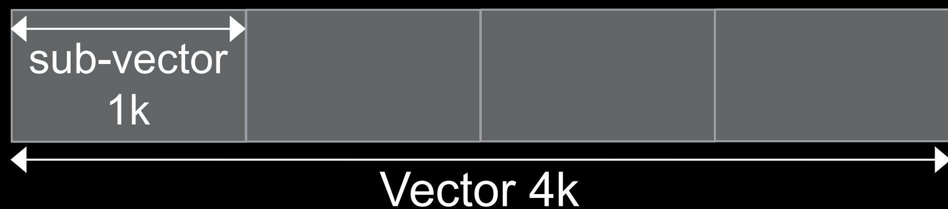
Enable back-to-back vector ops

AIE-ML AIE API Manual: https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aie_api/doc/

AMD
together we advance_

[Public]

Vectorizing vector_scalar_mul: c = a*factor



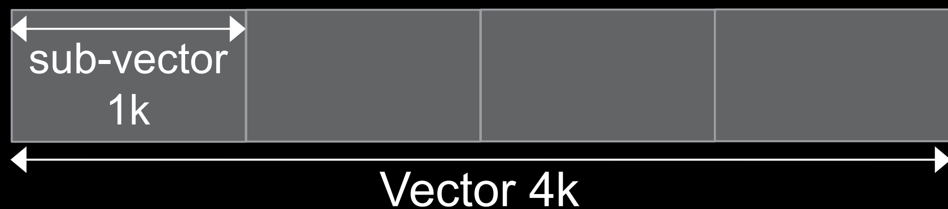
AIE Core Code (Scalar Version)

```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

  for (int i = 0; i < 1024; i++) {
    c[i] = factor * a[i];
  }
}
```

1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64

AMD
together we advance_

Vectorizing vector_scalar_mul: c = a*factor



AIE Core Code (Vectorizing WIP)

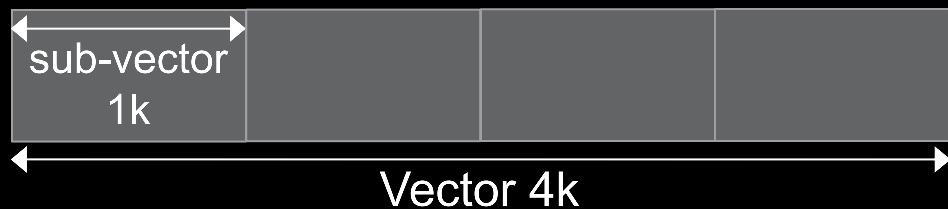
```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

  for (int i = 0; i < 1024/16; i++) {
    aie::accum<acc64, 16> cout = aie::mul(a, factor);
  }
}
```

1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64

https://xilinx.github.io/aie_api/topics.html

AMD
together we advance_

Vectorizing vector_scalar_mul: c = a*factor



AIE Core Code (Vectorizing WIP)

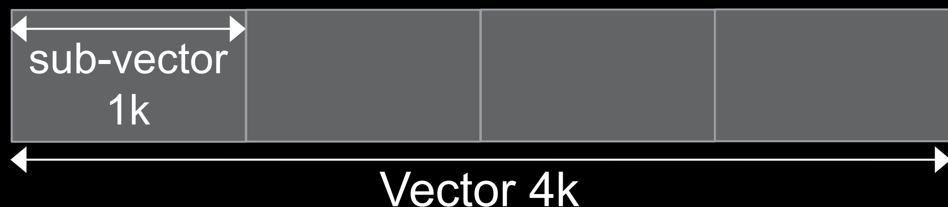
```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

  for (int i = 0; i < 1024/16; i++) {
    aie::vector< int32_t, 16> a0 = aie::load_v<16>(a);
    a += 16;
    aie::accum<acc64, 16> cout = aie::mul(a0, factor);
  }
}
```

1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64
2. Load input into vector register and increment input pointer

AMD
together we advance_

[Public]

Vectorizing vector_scalar_mul: c = a*factor



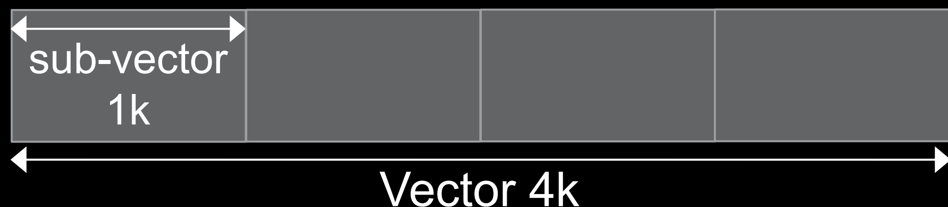
AIE Core Code (Vectorizing WIP)

```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

  for (int i = 0; i < 1024/16; i++) {
    aie::vector< int32_t, 16> a0 = aie::load_v<16>(a);
    a += 16;
    aie::accum<acc64, 16> cout = aie::mul(a0, factor);
    aie::store_v(c,cout.to_vector<int32_t>(0));
    c += 16;
  }
}
```

1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64
2. Load input into vector register and increment input pointer
3. Cast and store the result from the accumulator into the output and increment the output pointer

AMD
together we advance_

Vectorizing vector_scalar_mul: c = a*factor



AIE Core Code (Vectorizing WIP)

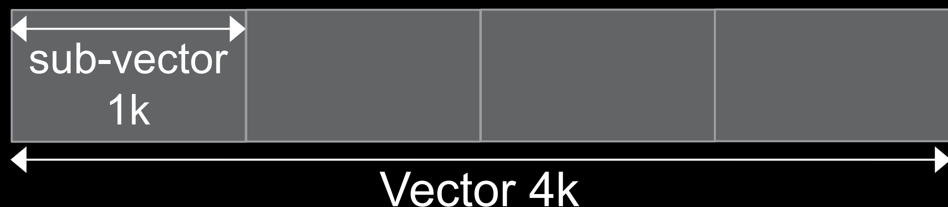
```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

 AIE_PREPARE_FOR_PIPELINING
 AIE_LOOP_MIN_ITERATION_COUNT(64)
 for (int i = 0; i < 1024/16; i++) {
   aie::vector< int32_t, 16> a0 = aie::load_v<16>(a);
   a += 16;
   aie::accum<acc64, 16> cout = aie::mul(a0, factor);
   aie::store_v(c,cout.to_vector<int32_t>(0));
   c += 16;
 }
}
```

1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64
2. Load input into vector register and increment input pointer
3. Cast and store the result from the accumulator into the output and increment the output pointer
4. Add optional pragmas to further optimize design

AMD
together we advance_

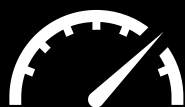
[Public]

Vectorizing vector_scalar_mul: c = a*factor



AIE Core Code (Vectorized)

```
void scale_int32(int32_t *a, int32_t *c,
                 int32_t factor) {

  AIE_PREPARE_FOR_PIPELINING
  AIE_LOOP_MIN_ITERATION_COUNT(64)
  for (int i = 0; i < 1024/16; i++) {
    aie::vector< int32_t, 16> a0 = aie::load_v<16>(a);
    a += 16;
    aie::accum<acc64, 16> cout = aie::mul(a0, factor);
    aie::store_v(c,cout.to_vector<int32_t>(0));
    c += 16;
  }
}
```

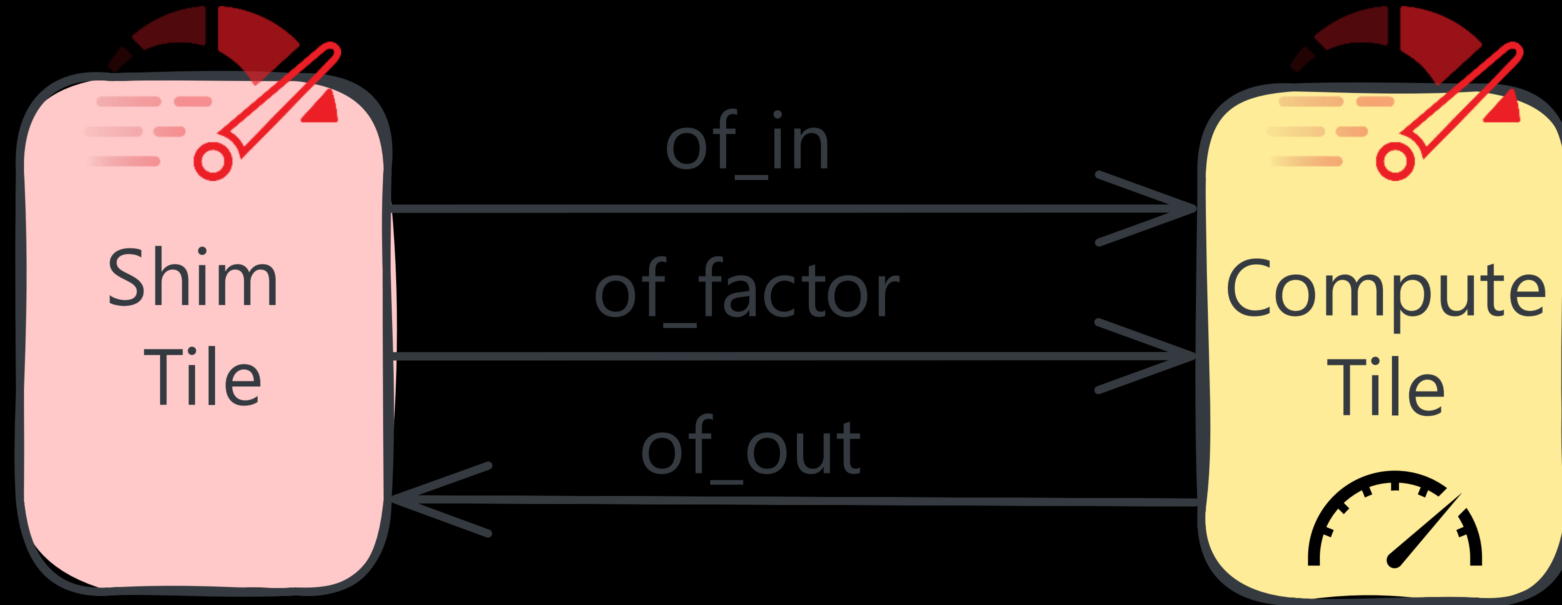


1. Find vector operation, its vector and accumulator size:
   mul/mac, int32 x int32 → vector size 16
   → acc type 64
2. Load input into vector register and increment input pointer
3. Cast and store the result from the accumulator into the output and increment the output pointer
4. Add optional pragmas to further optimize design

AMD
together we advance_

[Public]

Exercise 5: Tracing Vectorized vector_scalar_mul

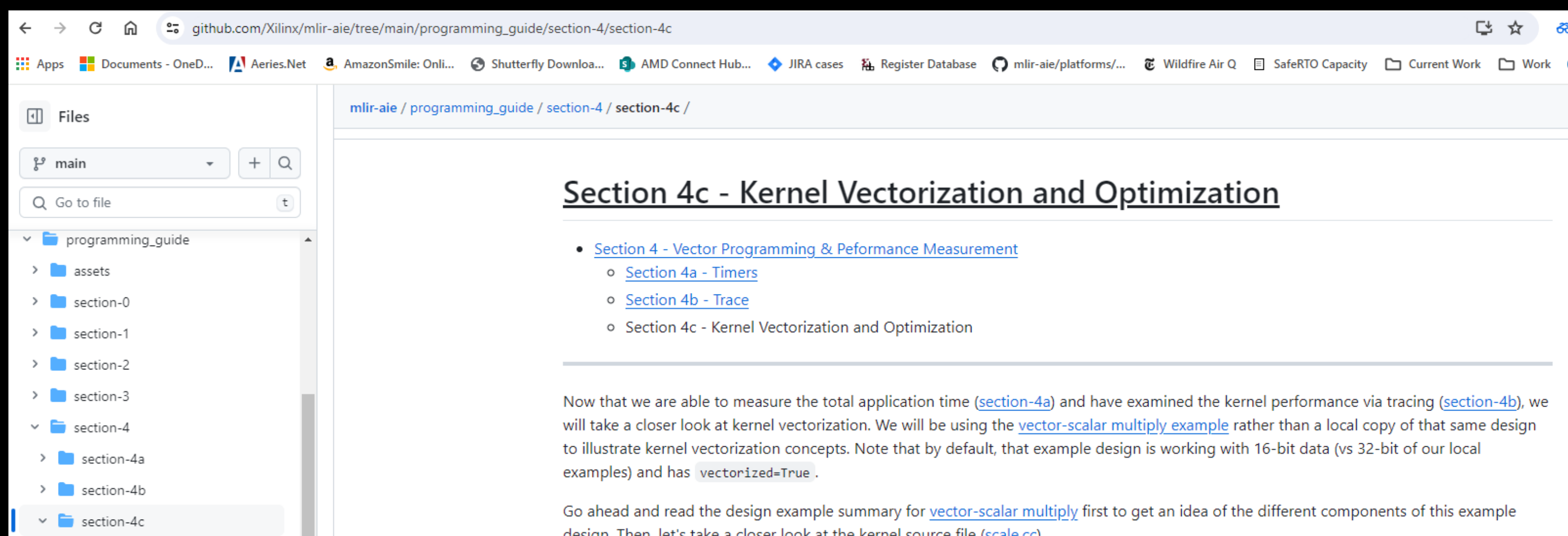


AMD
together we advance_

[Public]

Running Vectorization Exercises on Hardware

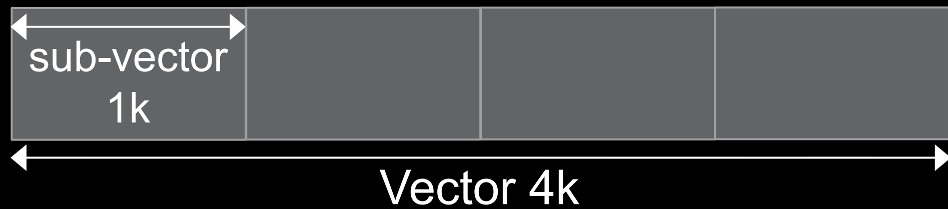
- Navigate now to Section 4c of our Programming Guide
  - (https://github.com/Xilinx/mlir-aie/tree/main/programming_guide/section-4/section-4c)
  - On your local mlir-aie install, change directory to mlir-aie/programming_guide/section-4/section-4c folder
  - Follow the instructions in Section-4c up to Vectorization Exercises
- NOTE – We will be working with programming_examples/basic/vector_scalar_mul example directly
  - Edit programming_examples/basic/vector_scalar_mul/vector_scalar_mul.py to set vectorized=False
  - Edit aie_kernels/aie2/scale.cc to comment/ uncomment pragmas
  - Use "*make use_placed=1 trace*" for this example



AMD
together we advance_

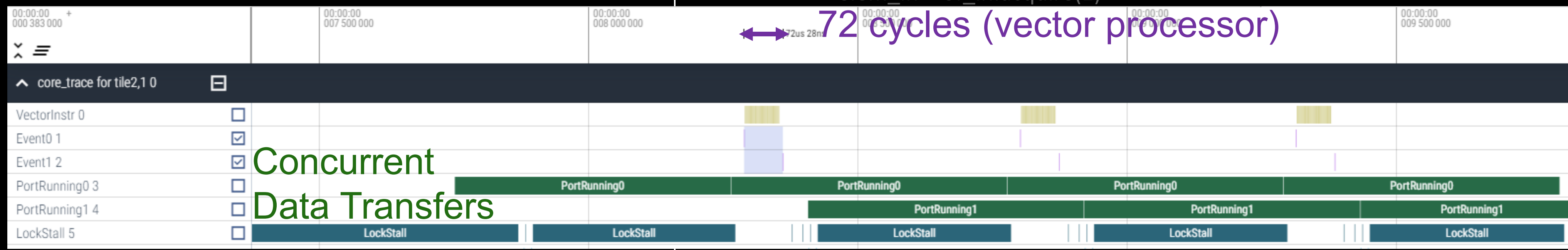
[Public]

Vectorizing vector_scalar_mul: c = a*factor



AIE Array Code

```
def my_vector_scalar(dev, vector_size):
scale = Kernel(f"scale_scalar", "scale.o",
          [tile_ty, tile_ty, scalar_ty, np.int32],)

of_in = ObjectFifo(tile_ty, name="in")
of_factor = ObjectFifo(scalar_ty, name="infactor")
of_out = ObjectFifo(tile_ty, name="out")

def core_body(of_in, of_factor, of_out, scale_fn):
    elem_factor = of_factor.acquire(1)
    for _ in range_(N_div_n):
            elem_in = of_in.acquire(1)

rt = Runtime()
with rt.sequence(tensor_ty, scalar_ty, tensor_ty) as (A, F, C):
orker)
n.prod(), A)
actor.prod(), F)
f_out.cons(), C, wait=True)

dev, rt).resolve_program(SequentialPlacer())
```



AIE Core Code (Vector Version)

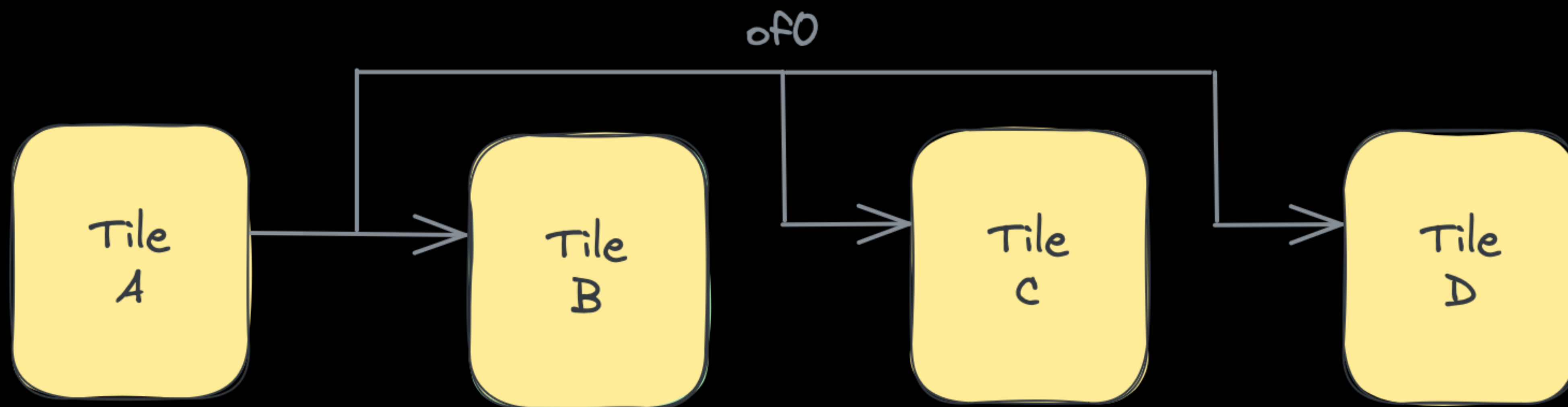
```
void scale_int32(int32_t *a, int32_t *c, int32_t factor) {
 event0();
 for (int i = 0; i < 1024/16; i++) {
  aie::vector< int32_t, 16> a0 = aie::load_v<16>(a);
  a += 16;
  aie::accum<acc64, 16> cout = aie::mul(a0, factor);
  aie::store_v(c,cout.to_vector<int32_t>(0));
  c += 16;
 }
 event1();
}
```


AMD
together we advance_

Dataflow and Larger Designs

[Public]

Broadcast with ObjectFifo



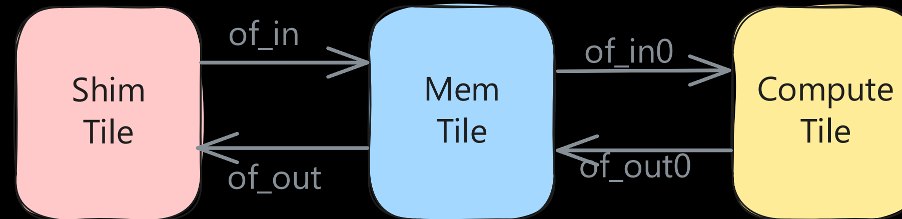

```
of0 = ObjectFifo(np.ndarray[(256,), np.dtype[np.int32]],3,"objfifo0")
A = Worker(..., fn_args=[of0.prod(),] )
B = Worker(..., fn_args=[of0.cons(),] )
C = Worker(..., fn_args=[of0.cons(),] )
D = Worker(..., fn_args=[of0.cons(),] )
```


AMD
together we advance_

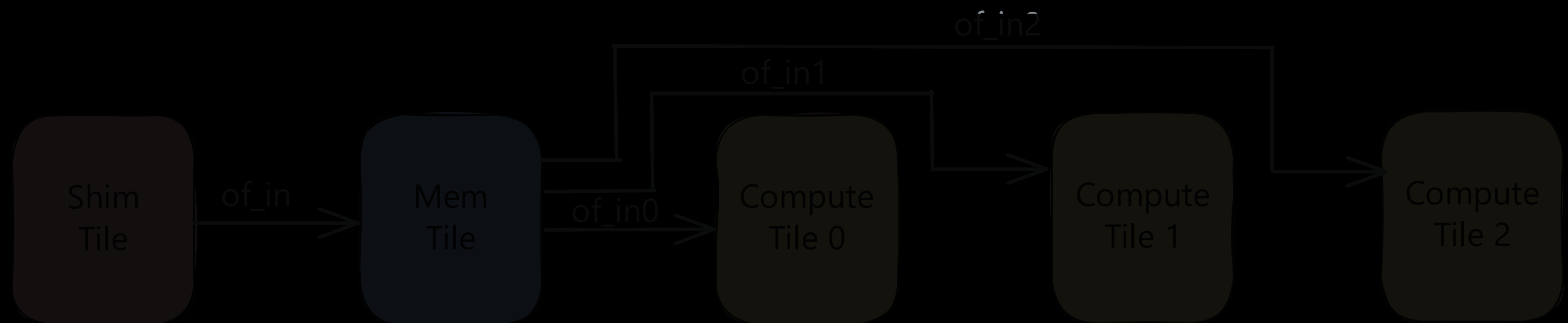
[Public]

Some ObjectFifo Patterns

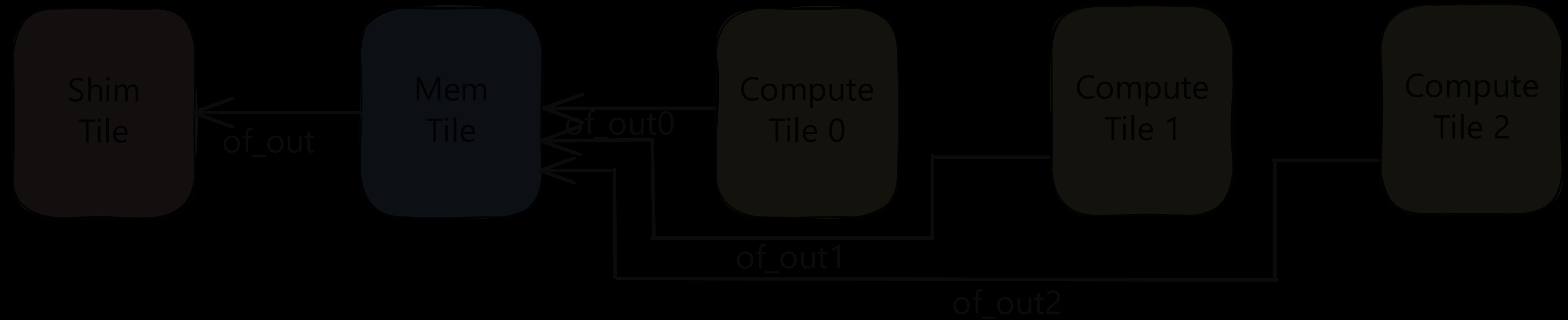
- forward() -- Hierarchical Data Movement



- split() from MemTile



- join() in MemTile



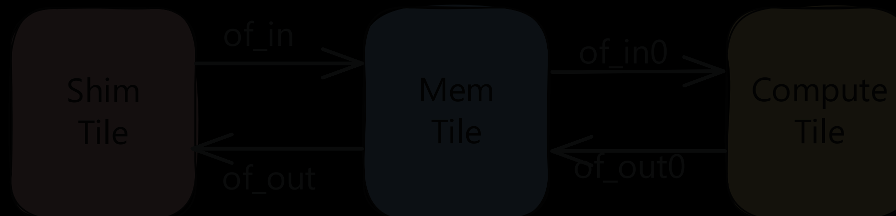
Checkout the programming guide for more patterns and details!

AMD
together we advance_

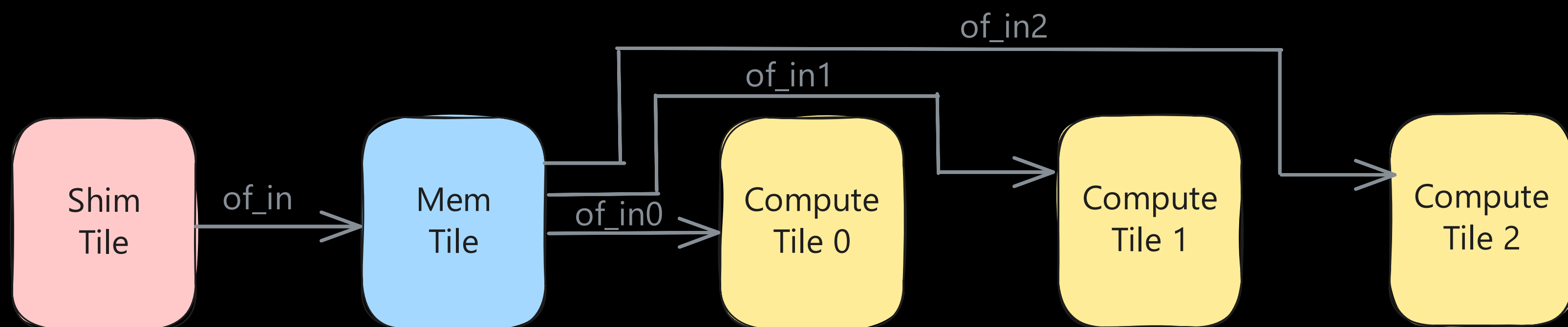
[Public]

Some ObjectFifo Patterns

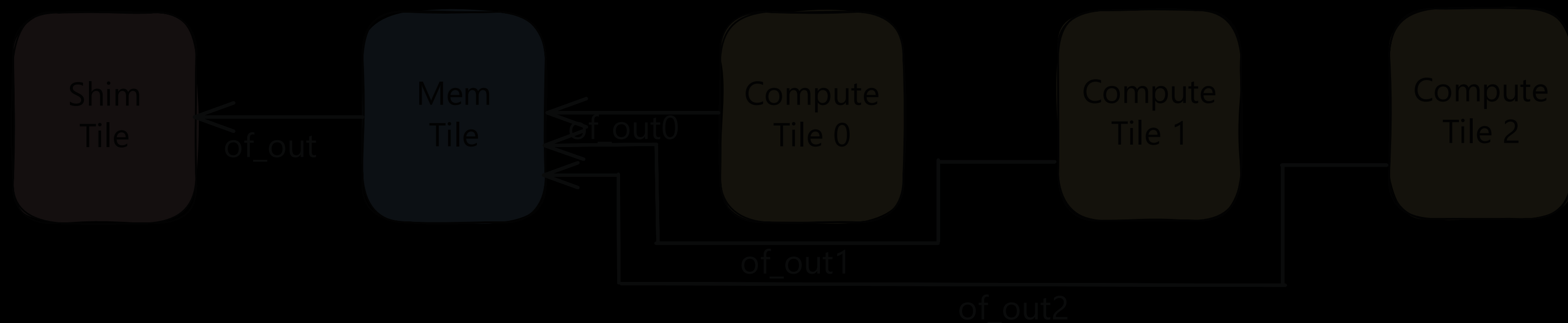
- forward() -- Hierarchical Data Movement



- split() from MemTile



- join() in MemTile

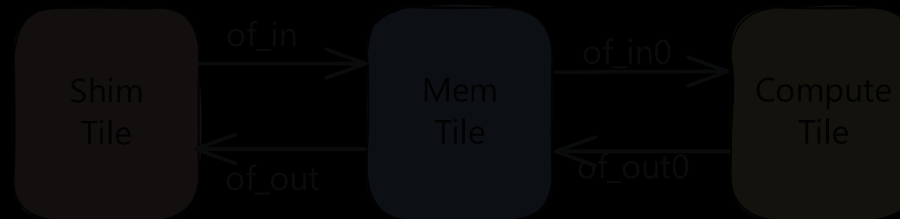


Checkout the programming guide for more patterns and details!

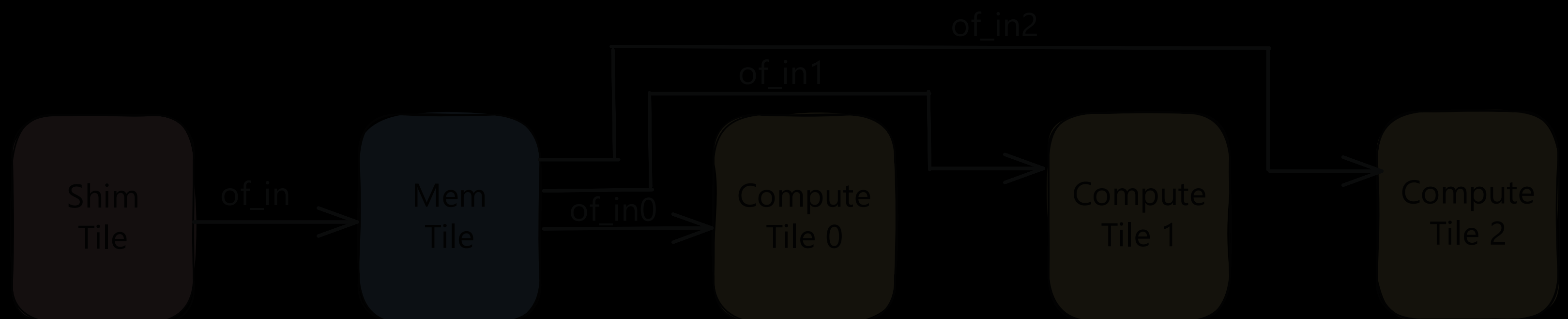
AMD
together we advance_

Some ObjectFifo Patterns

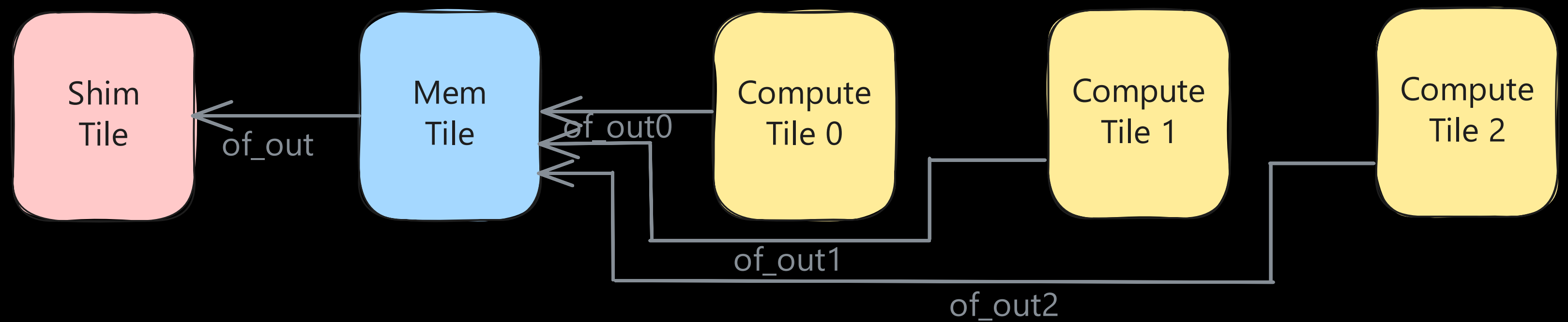
- forward() -- Hierarchical Data Movement



- split() from MemTile



- join() in MemTile



Checkout the programming guide for more patterns and details!

AMD
together we advance_

More Information

Checkout the programming guide for
more patterns and details!

IRON API Docs:
https://xilinx.github.io/mlir-aie/python/html/

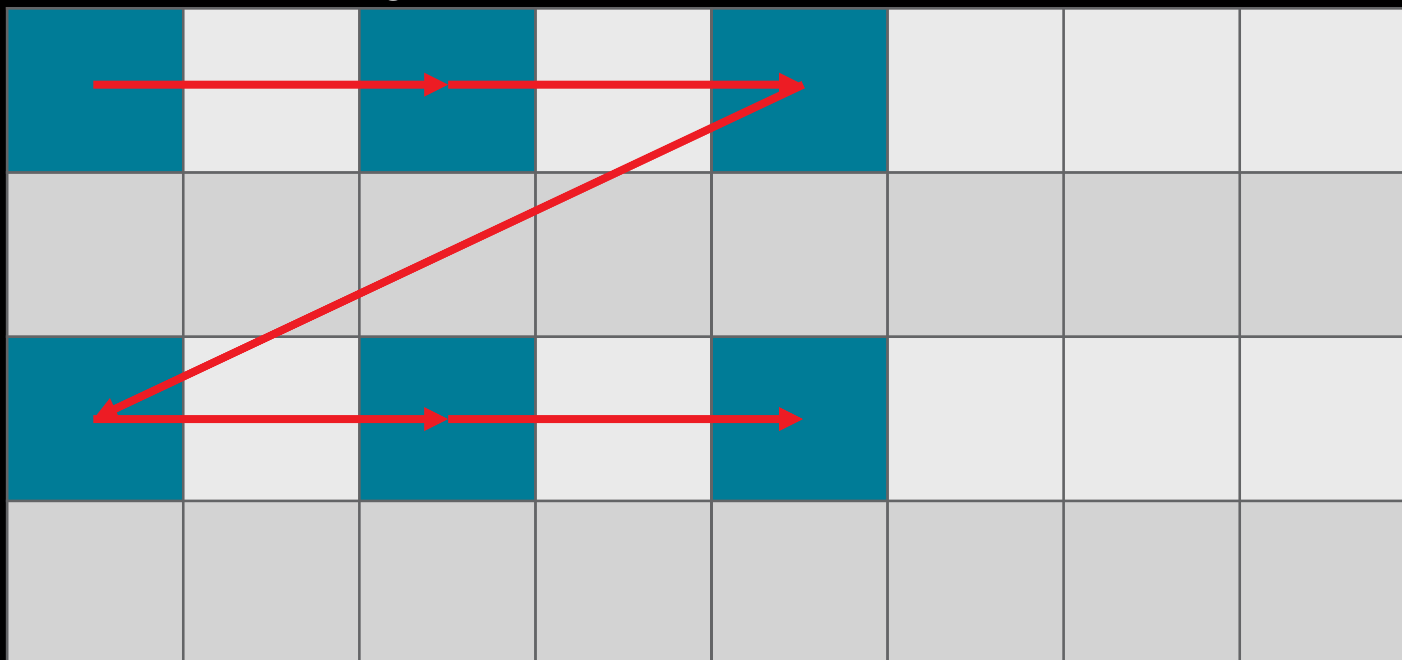
View Source Code:
mlir-aie/python/iron

AMD
together we advance_

[Public]

Data Layout Transformations on the Fly

```
of0 = ObjectFifo(np.ndarray[(4, 8), np.dtype[int32]], 3, "objfifo0",
                 dims_to_stream = [(8, 1), (4, 8),])
```




```
int *buffer;
for(int dim1 = 0; dim1 < 2; dim1++)   // size_1
  for(int dim0 = 0; dim0 < 3; dim0++) // size_0
    // access/store element at/to index:
    buffer[
      dim1 * 16  // stride_1
      + dim0 * 2 // stride_0
    ];
```


AMD
together we advance_

[Public]

Exercise 6: mini_tutorial Part 2

Navigate in your browser to https://github.com/Xilinx/mlir-aie/tree/main/programming_guide/mini_tutorial

- Contains a mini tutorial with exercises for learning IRON data movement concepts
- Follow the README in the mini_tutorial directory
- Do exercise 2

AMD
together we advance_

[Public]

Answer:

```
def exercise_2(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in")

    # Split of_in into of_ins
    of_ins = of_in.cons().split(
            of_offsets,
            obj_types=[tile_ty] * n_workers,
            names=[f"in{worker}" for worker in range(n_workers)],
        )

    of_out = ObjectFifo(tile_ty, name="out")

    # Split of_in into of_ins
    of_outs = of_out.prod().join(
            of_offsets,
            obj_types=[tile_ty] * n_workers,
            names=[f"in{worker}" for worker in range(n_workers)],
        )

    # Create workers to perform the tasks
    workers = []
    for worker in range(n_workers):
      workers.append(Worker(core_fn, [of_ins[worker].cons(), of_outs[worker].prod()]))
```


AMD
together we advance_

[Public]

Answer:

```
def exercise_2(input, output):
    ...
    # Dataflow with ObjectFifos
    of_in = ObjectFifo(tile_ty, name="in")

    # Split of_in into of_ins
    of_ins = of_in.cons().split(
            of_offsets,
            obj_types=[tile_ty] * n_workers,
            names=[f"in{worker}" for worker in range(n_workers)],
        )

    of_out = ObjectFifo(tile_ty, name="out")

    # Split of_in into of_ins
    of_outs = of_out.prod().join(
            of_offsets,
            obj_types=[tile_ty] * n_workers,
            names=[f"in{worker}" for worker in range(n_workers)],
        )

    # Create workers to perform the tasks
    workers = []
    for worker in range(n_workers):
        workers.append(Worker(core_fn, [of_ins[worker].cons(), of_outs[worker].prod()]))
```


AMD
together we advance_

[Public]

Example Vector Designs

Basic

Design name	Data type	Description
Vector Scalar Add	i32	Adds 1 to every element in vector
Vector Scalar Mul	i32	Returns a vector multiplied by a scale factor
Vector Vector Add	i32	Returns a vector summed with another vector
Vector Vector Modulo	i32	Returns vector % vector
Vector Vector Multiply	i32	Returns a vector multiplied by a vector
Vector Reduce Add	bfloat16	Returns the sum of all elements in a vector
Vector Reduce Max	bfloat16	Returns the maximum of all elements in a vector
Vector Reduce Min	bfloat16	Returns the minimum of all elements in a vector
Vector Exp	bfloat16	Returns a vector representing e^x of the inputs
DMA Transpose	i32	Transposes a matrix with the Shim DMA using npu_dma_memcpy_nd
Matrix Scalar Add	i32	Returns a matrix multiplied by a scalar
Single core GEMM	bfloat16	A single core matrix-matrix multiply
Multi core GEMM	bfloat16	A matrix-matrix multiply using 16 AIEs with operand broadcast. Uses a simple "accumulate in place" strategy
GEMV	bfloat16	A vector-matrix multiply returning a vector

ML Operators

Design name	Data type	Description
Eltwise Add	bfloat16	An element by element addition of two vectors
Eltwise Mul	i32	An element by element multiplication of two vectors
ReLU	bfloat16	Rectified linear unit (ReLU) activation function on a vector
Softmax	bfloat16	Softmax operation on a matrix
Conv2D	i8	A single core 2D convolution for CNNs
Conv2D+ReLU	i8	A Conv2D with a ReLU fused at the vector register level

AMD
together we advance_

Larger Example Designs

Vision Pipelines

Design name	Data type	Description
Vision Passthrough	i8	A simple pipeline with just one passThrough kernel. This pipeline mainly aims to test whether the data movement works correctly to copy a greyscale image.
Color Detect	i32	This multi-kernel, multi-core pipeline detects colors in an RGBA image.
Edge Detect	i32	A multi-kernel, multi-core pipeline that detects edges in an image and overlays the detection on the original image.
Color Threshold	i32	A multi-core data-parallel implementation of color thresholding of a RGBA image.

ML Designs

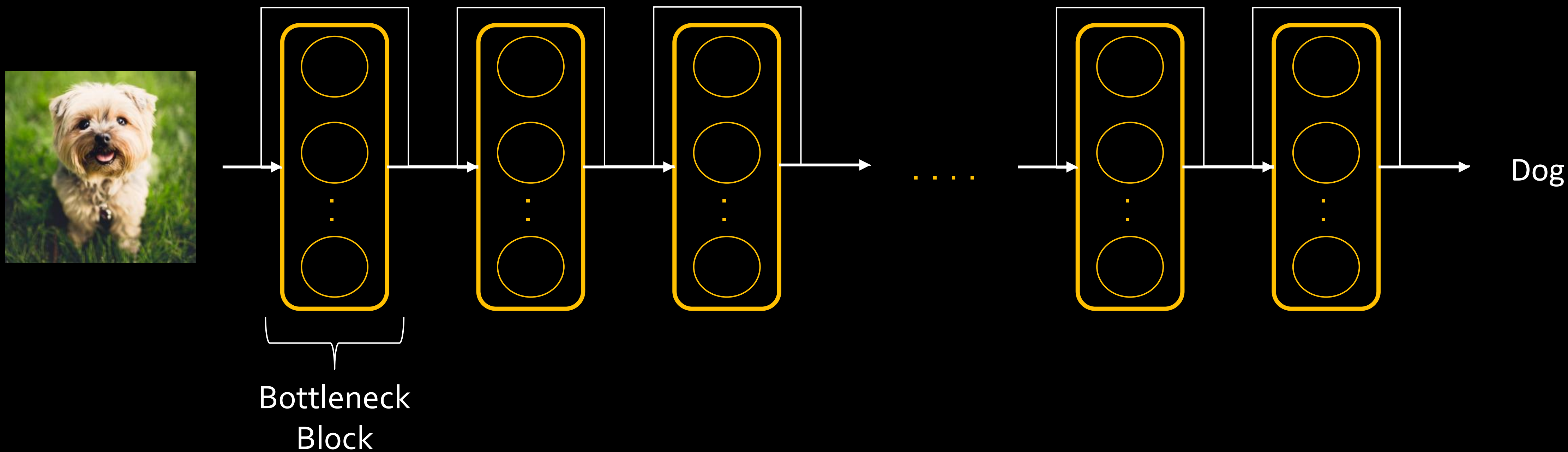
Design name	Data type	Description
bottleneck	ui8	A Bottleneck Residual Block is a variant of the residual block that utilizes three convolutions, using 1x1, 3x3, and 1x1 filter sizes, respectively. The implementation features fusing of multiple kernels and dataflow optimizations, highlighting the unique architectural capabilities of AI Engines
resnet	ui8	ResNet with offloaded conv2_x layers. The implementation features depth-first implementation of multiple bottleneck blocks across multiple NPU columns.
mobilenet	ui8	MobileNet v3. The implementation features depth-first implementation of multiple bottleneck blocks utilizing all 32 tiles in a Strix/ Strix Halo device. Currently uses scalar convolution kernels, with vectorized kernels coming soon.

AMD
together we advance_

[Public]

Design 1: ResNet: Residual Network on Whole NPU Array

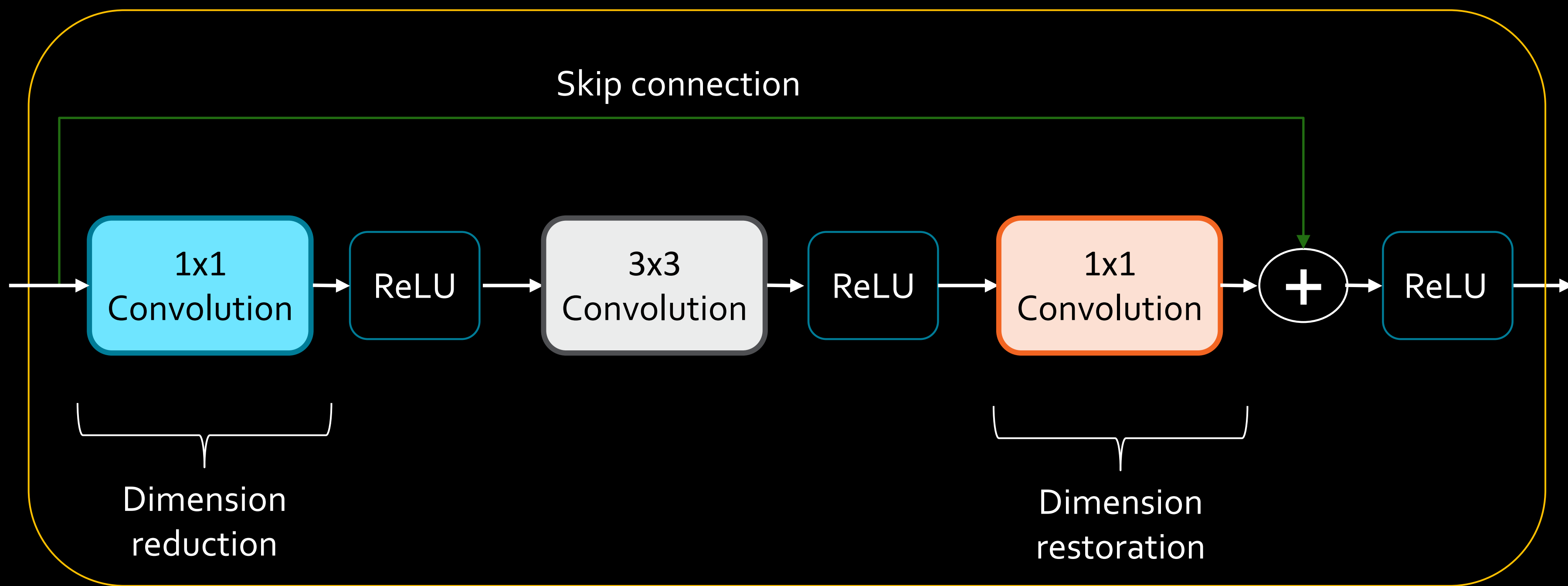
https://github.com/Xilinx/mlir-aie/tree/test-tutorial/programming_examples/ml/resnet



AMD
together we advance_

[Public]

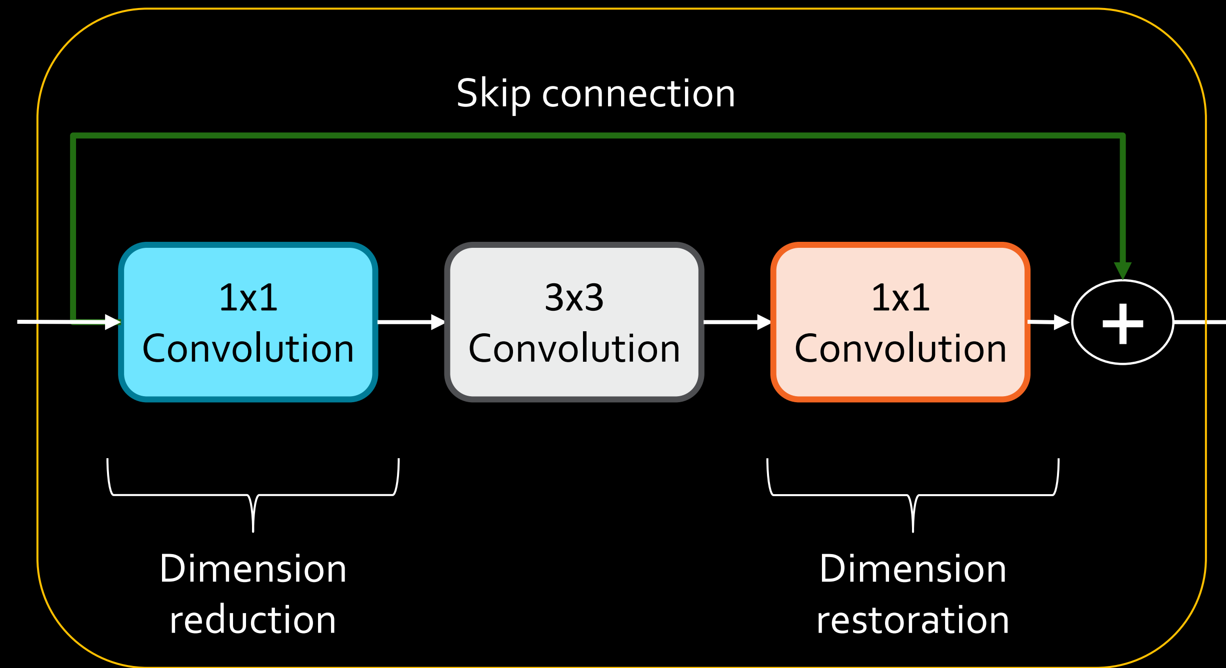
Bottleneck Block



AMD
together we advance_

[Public]

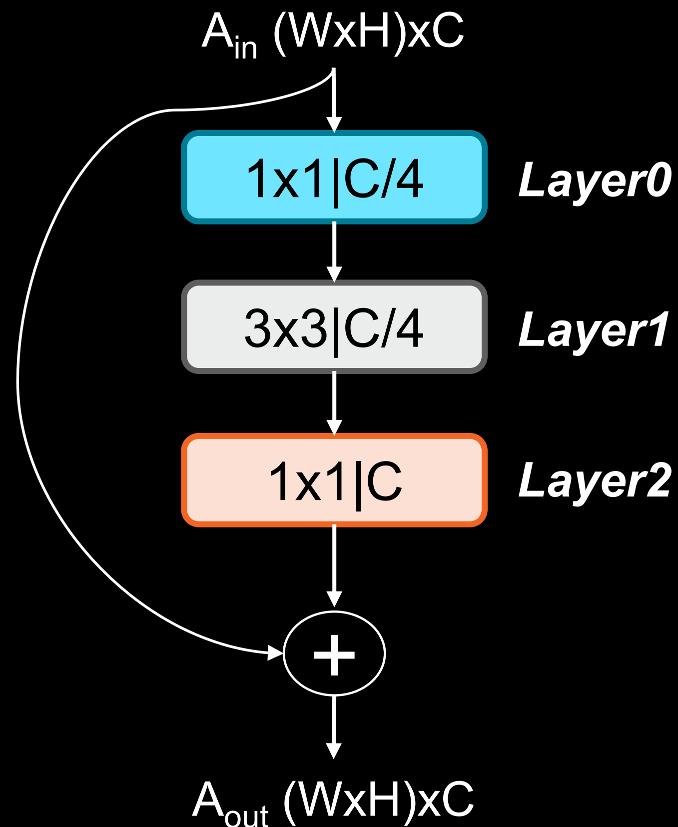
Bottleneck Block: Kernel Fusion

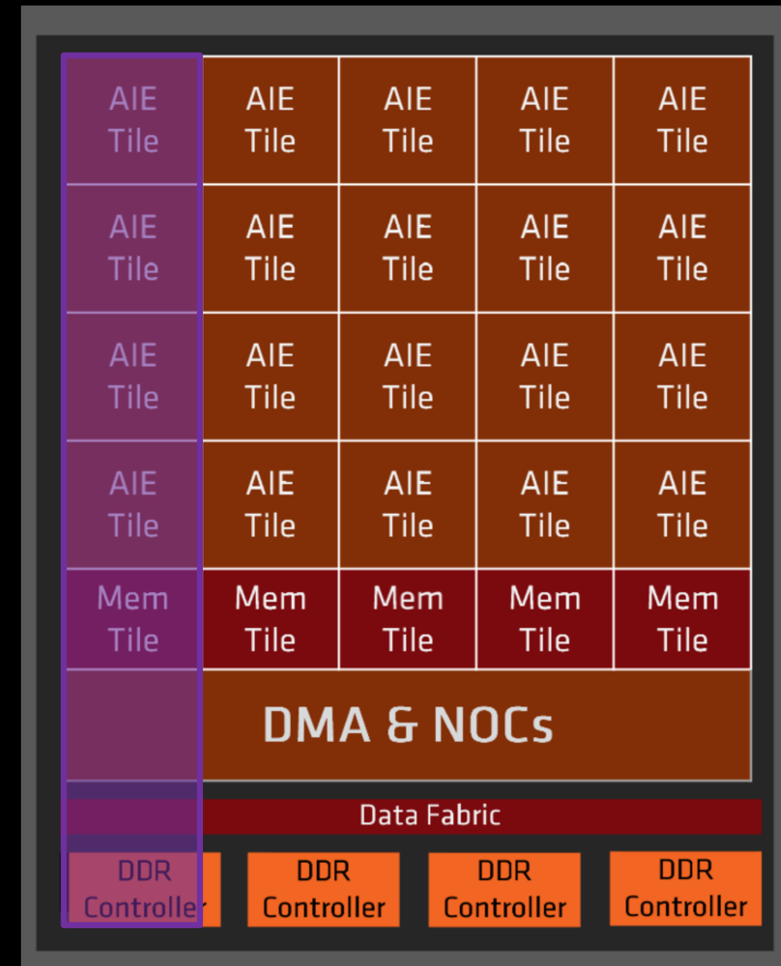


AMD
together we advance_

[Public]

Mapping 1 Bottleneck on 1 NPU Column

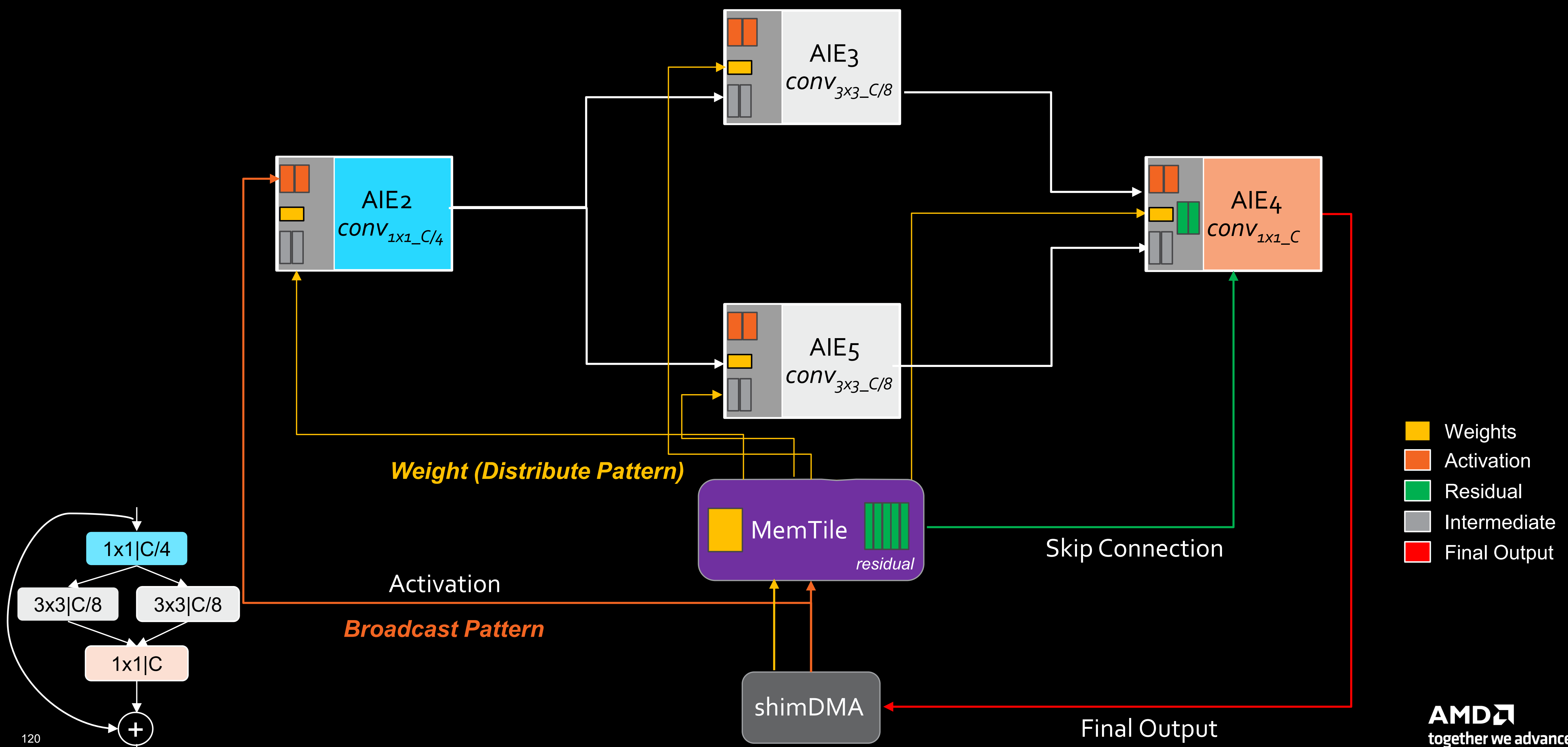




AMD
together we advance_

[Public]

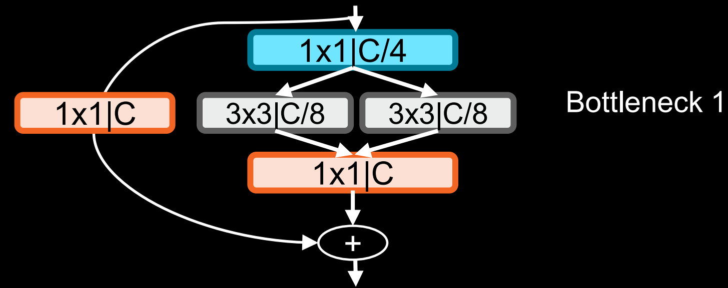
Mapping 1 Bottleneck on 1 NPU Column: Spatial Dataflow Mapping



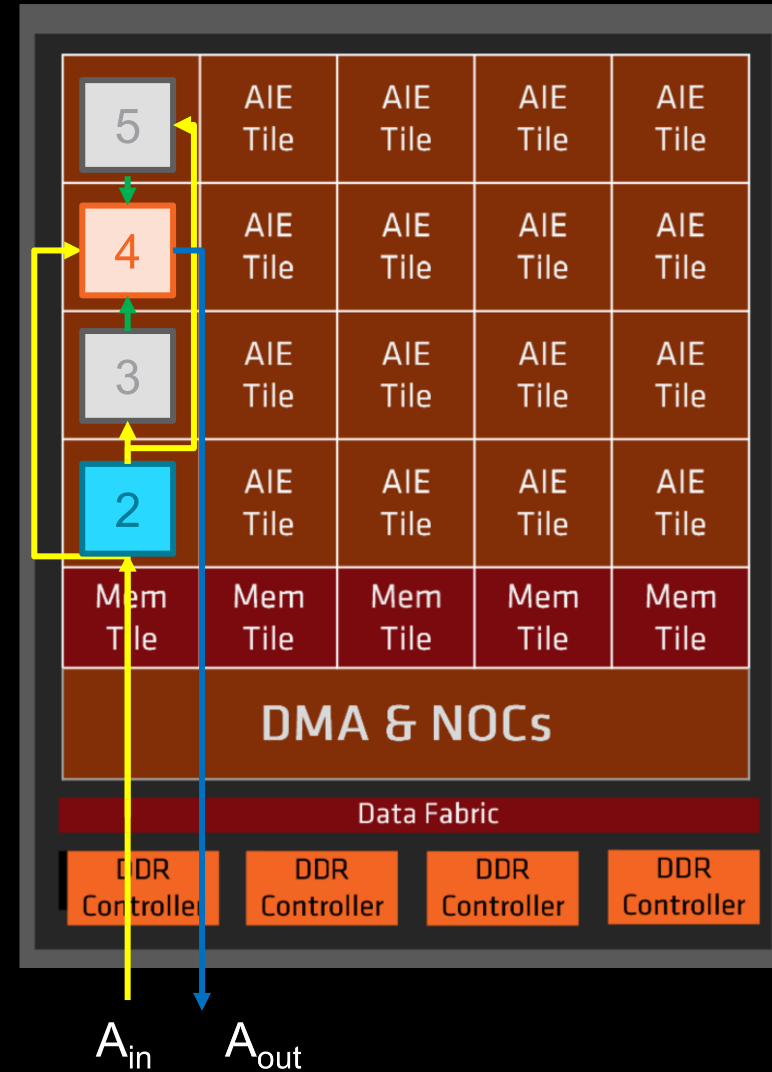
AMD
together we advance_

[Public]

Mapping 1 Bottleneck on 1 NPU Column



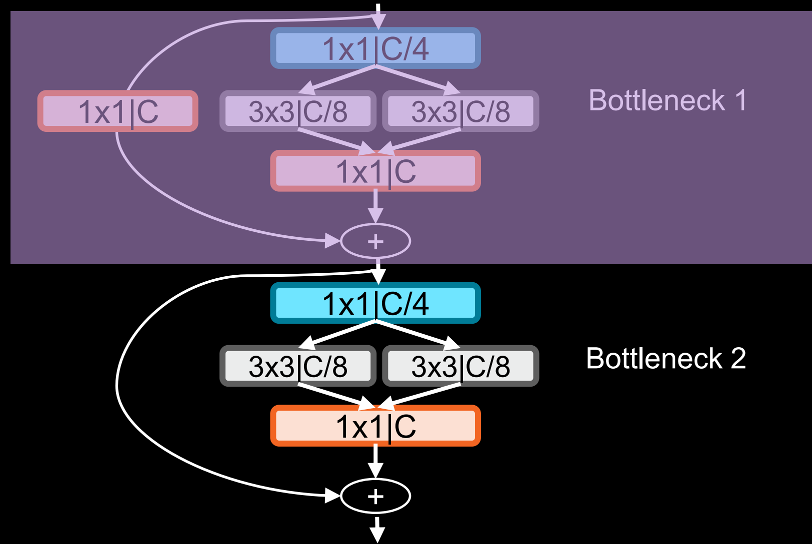
Bottleneck 1

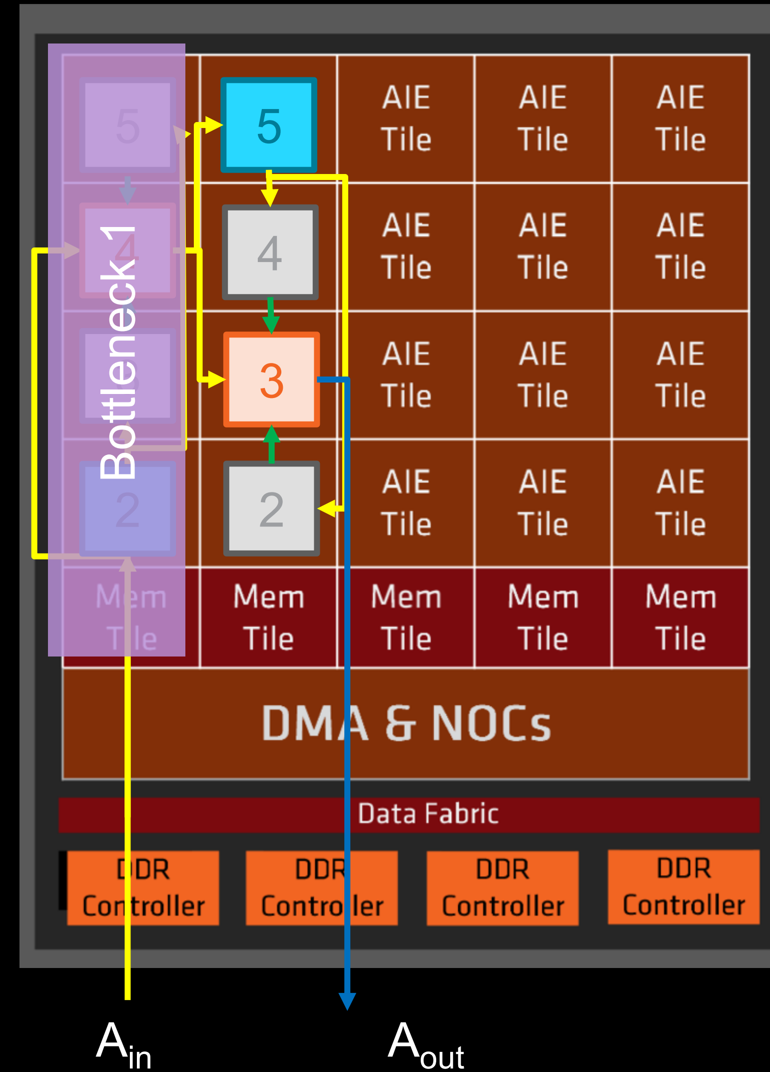


AMD
together we advance_

[Public]

Mapping 2 Bottleneck on 2 NPU Column

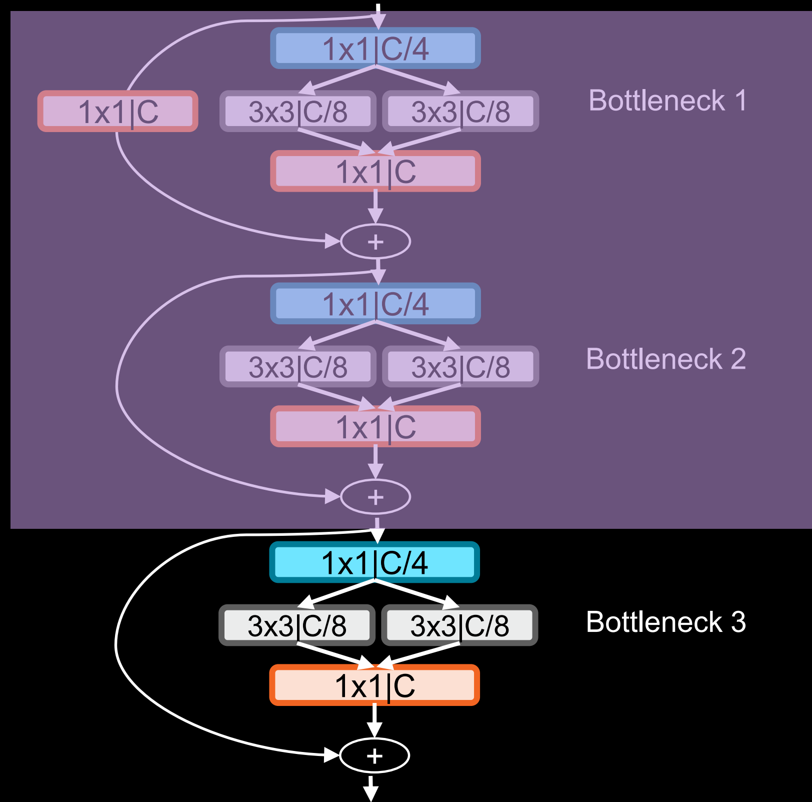


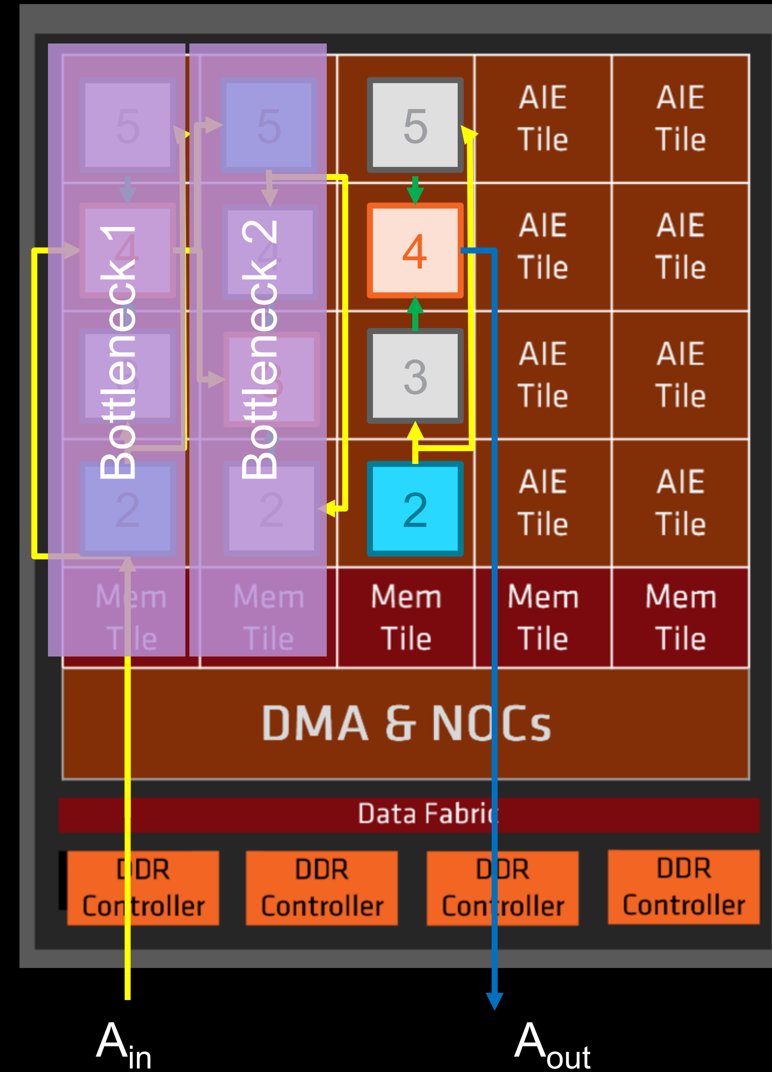


AMD
together we advance_

[Public]

Mapping 3 Bottleneck on 3 NPU Column

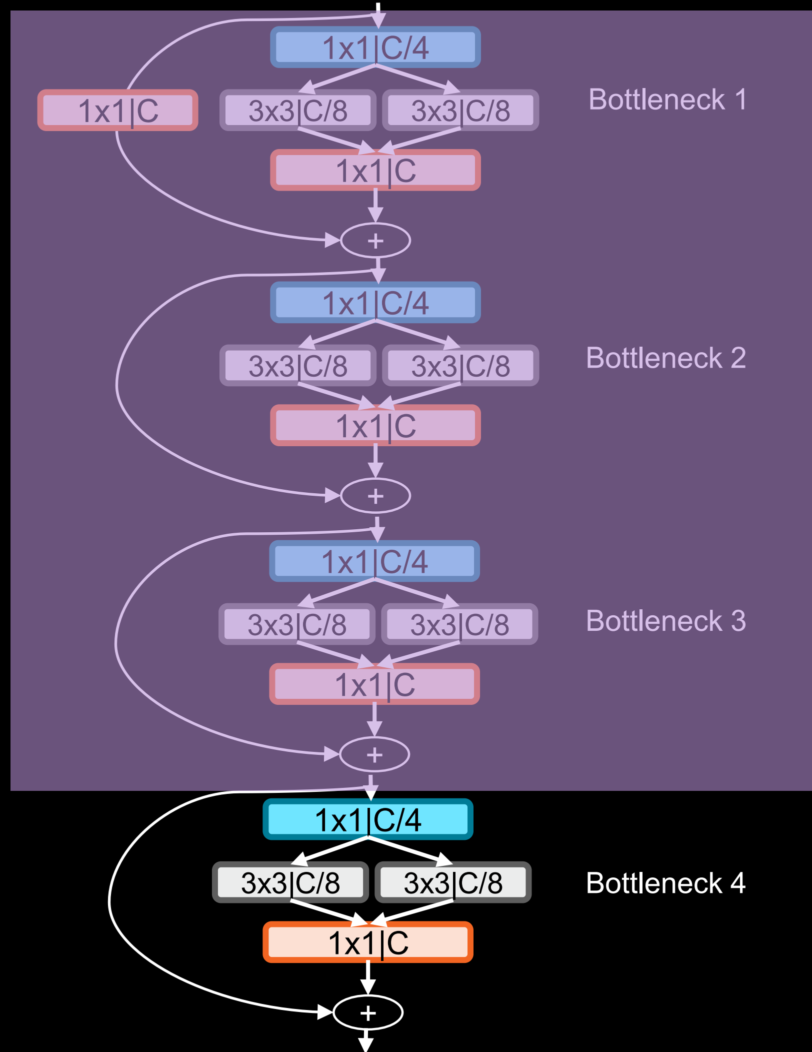


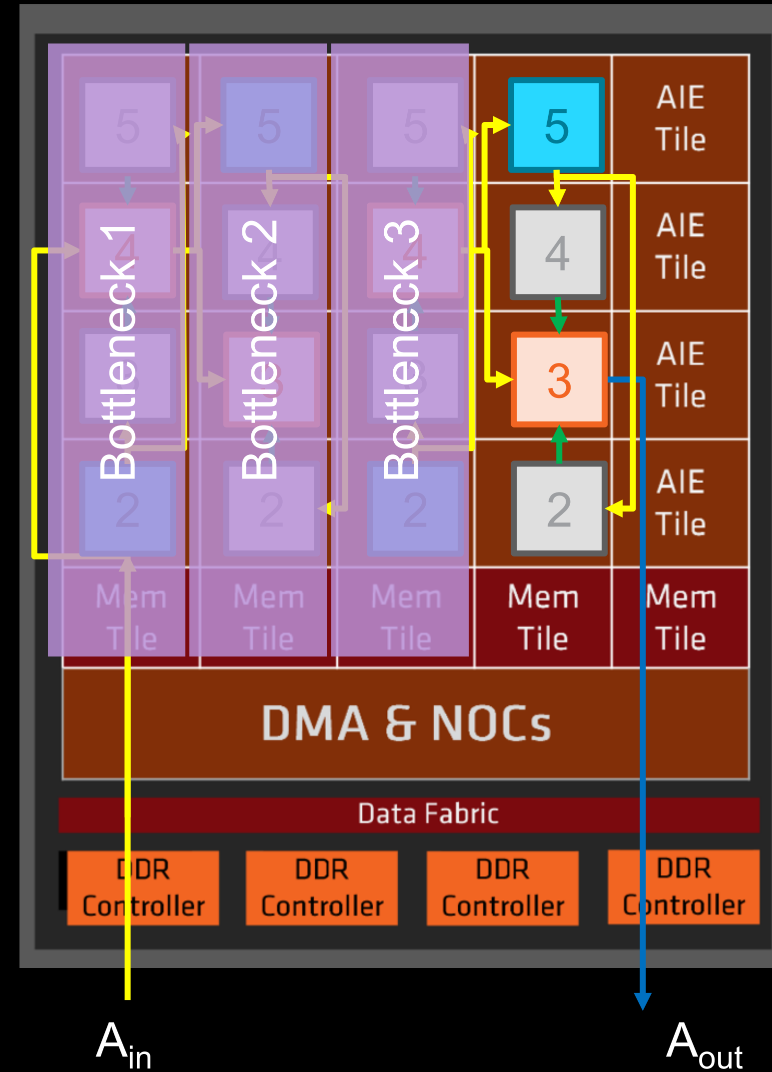


AMD
together we advance_

[Public]

Mapping 4 Bottleneck on 4 NPU Column



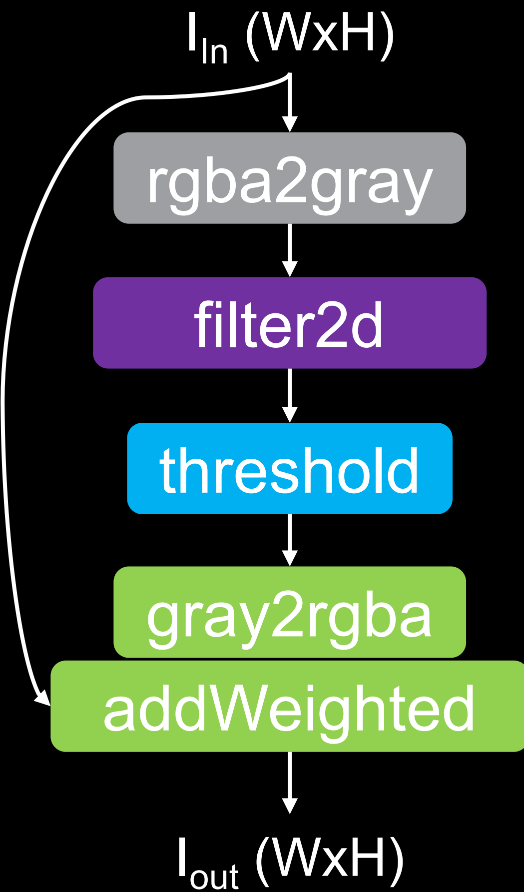


AMD
together we advance_

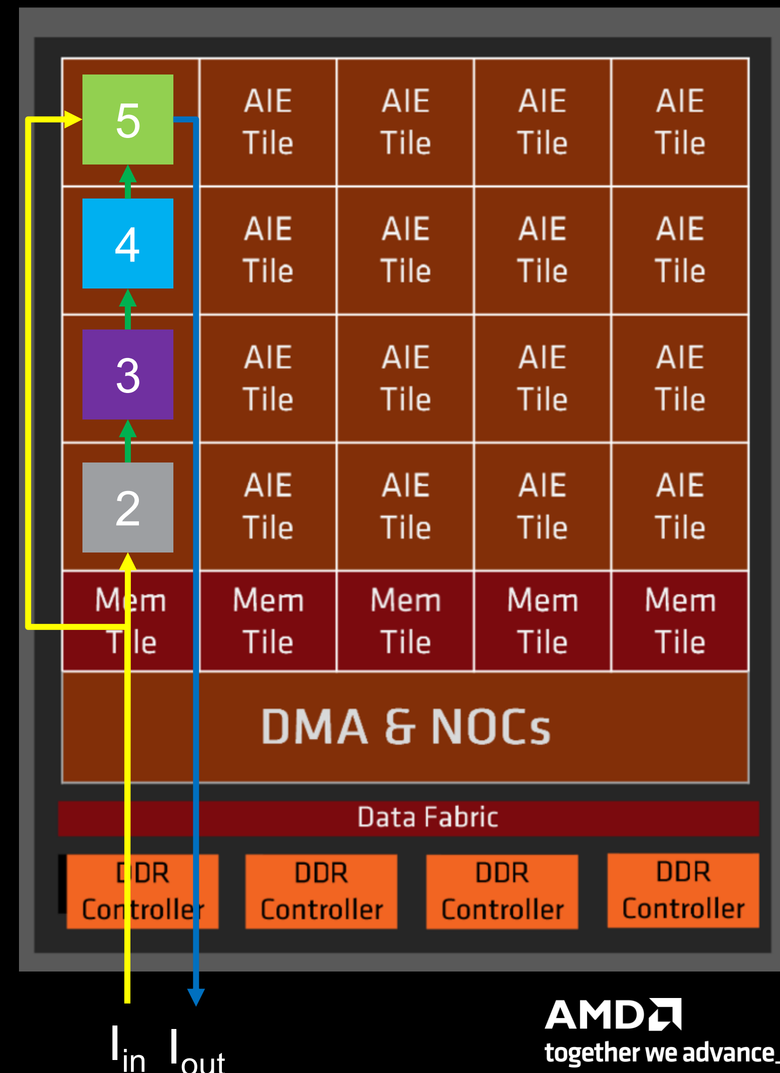
[Public]

Design 2: Edge Detect Vision Pipeline on 1 NPU Column

https://github.com/Xilinx/mlir-aie/tree/test-tutorial/programming_examples/vision/edge_detect




```
def edgeDetect():
@device(AIEDevice.ipu)
def deviceBody():

# AIE Core Function declarations


# AIE-array data movement with object fifos
# input RGBA broadcast + memtile for skip
inOF_L3L2 = ObjectFifo(lB_ty, 2, "inOF_L3L2")
inOF_L2L1 = inOF_L3L2.cons(7).forward("inOF_L2L1")

# output RGBA
outOF_L1L2 = ObjectFifo(lB_ty, 2, "outOF_L1L2")
outOF_L2L3 = outOF_L1L2.cons().forward("outOF_L2L3")


# between computeTiles
OF_2to3 = ObjectFifo(lB_ty, 4, "OF_2to3")
OF_3to4 = ObjectFifo(lB_ty, 2, "OF_3to4")
OF_4to5 = ObjectFifo(lB_ty, 2, "OF_4to5")
OF_5to5 = ObjectFifo(lB_ty, 1, "OF_5to5")

# Tasks for cores to perform

# Workers to run the tasks

# Runtime sequence
```

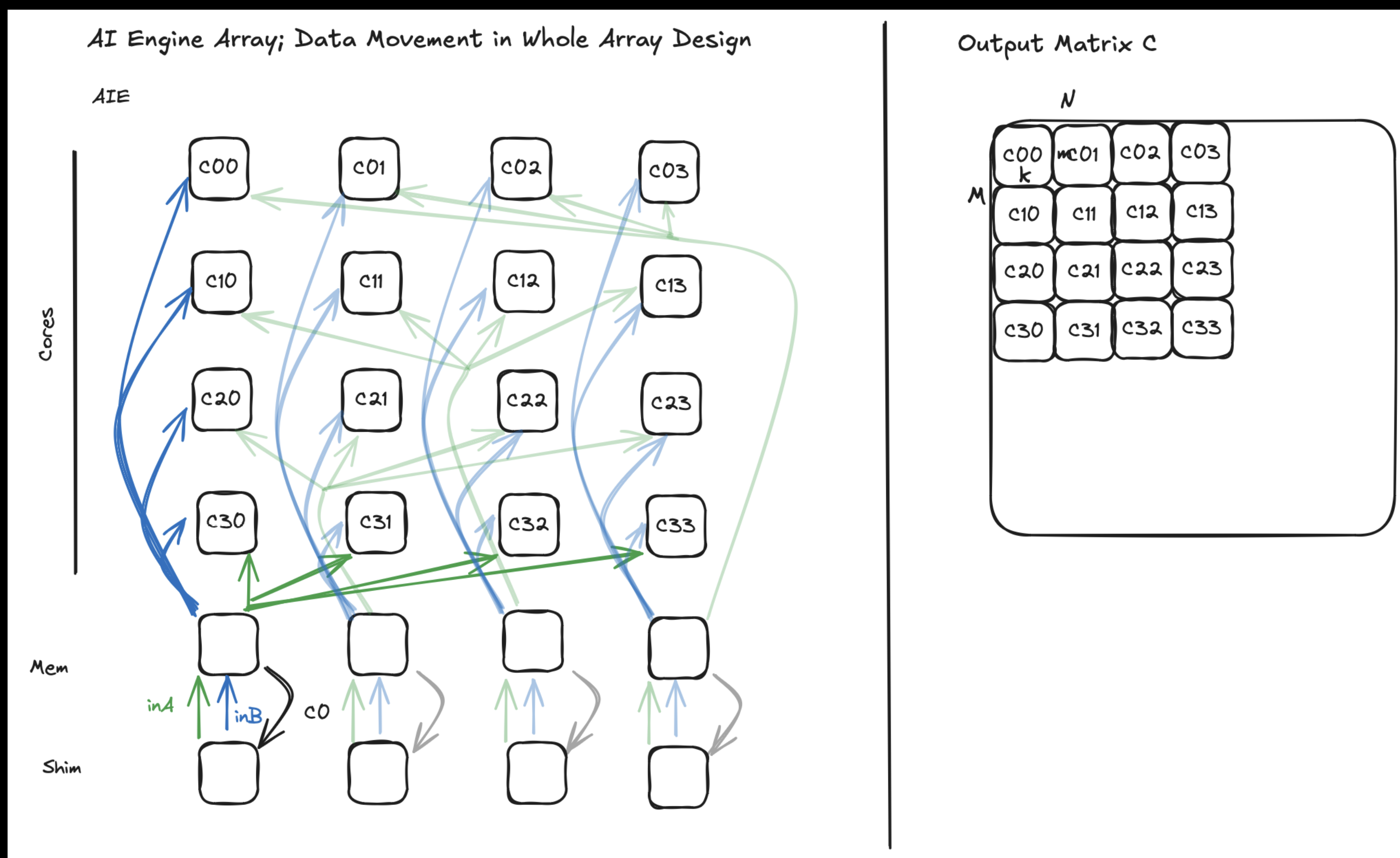


AMD
together we advance_

[Public]

Design 3: Matrix Multiplication on Whole NPU Array

https://github.com/Xilinx/mlir-aie/tree/main/programming_examples/basic/matrix_multiplication/whole_array



AMD
together we advance_

Wrapping Up

[Public]

Continued Learning

- Open-source repositories:
  - IRON + mlir-aie: https://github.com/Xilinx/mlir-aie
  - Peano: https://github.com/Xilinx/llvm-aie
- Papers:
  - Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface – *FCCM 2025*
  - AMD XDNA™ NPU in Ryzen™ AI Processors – *IEEE Micro 2024*
- Readily available AIE hardware:
  - Phoenix, Hawk, Strix, Krackan, Gorgon Point
  - Recommended hardware:
    - Phoenix Point Mini PC: Minisforum EM780 : AMD Ryzen™ 7 7840U
    - Hawk Point Mini PC: Minisforum UM890 Pro : AMD Ryzen™ 9 8945HS
    - Strix Point Mini PC: Minisforum AI X1 Pro : AMD Ryzen™ AI 9 HX 370
    - Krackan Point Mini PC: ASRock 4x4 BOX-AI350 : AMD Ryzen™ AI 7 350

README License

IRON API and MLIR-based AI Engine Toolchain

open pull requests 40 | bug issues 13 open | downloads@latest-wheels 11k | contributors 64

This project emphasizes fast, open-source toolchains for NPU devices including LLVM-based code generation. IRON contains a close-to-metal toolkit that empowers performance engineers to create fast and efficient designs for Ryzen™ AI NPUs powered by AI Engines. It provides Python APIs that enable developers to harness the unique architectural capabilities of AMD's NPUs. However, this project is not intended to represent an end-to-end compilation flow for all application designs---it is designed to complement, not replace, mainstream NPU tooling for inference like the AMD Ryzen™ AI Software Platform. Targeting researchers and enthusiasts, IRON is designed to unlock the full potential of NPUs for a wide range of workloads, from machine learning to digital signal processing and beyond. This repository includes programming guides and examples demonstrating the APIs. Additionally, the Peano component extends the LLVM framework by adding support for the AI Engine processor as a target architecture, enabling integration with popular compiler frontends such as `clang`. Developers can leverage the AIE API header library to implement efficient vectorized AIE core code in C++ that can be compiled by Peano.

AMD
together we advance_

[Public]

IRON Library

🔗 https://github.com/amd/IRON

- IRON repository for NPU developers
- Optimized operator library
- Example model implementation:
  - Llama 3.2 1B

💪 - IRON: Unlocking the Full Potential of NPUs - 💪

Discord | release v0.0.0 | downloads 0 | Small Benchmark/Test Suite passing | PRs welcome | license Apache | code style black

AMD XDNA 2

World's most powerful NPU for Next Gen AI PCs

Up to 5x Compute Capacity
Enhanced AI Tiles enabling 2x multitasking

Up to 2x Power Efficiency
Architectural advancements for generative AI workloads

IRON is an open-source & close-to-metal Python API enabling fast and efficient execution on AMD Ryzen™ AI NPUs. It relies on language bindings around the MLIR-AIE dialect.

The IRON Python API for Ryzen™ AI NPUs is described in the following paper:

> E. Hunhoff, J. Melber, K. Denolf, A. Bisca, S. Bayliss, S. Neuendorffer, J. Fifield, J. Lo, P. Vasireddy, P. James-Roxby, E. Keller. "Efficiency, Expressivity, and Extensibility in a Close-to-Metal NPU Programming Interface". In 33rd IEEE International Symposium On Field-Programmable Custom Computing Machines, May 2025.

🎯 Operator Dashboard

Section	Description	Datatype	AIE2	AIE2P	Status	Design Example
Element-wise Add	Element-wise addition kernel	bfloat16	✓	✓	🟢	iron/operators/elementwise_add/
Element-wise Mul	Element-wise multiplication kernel	bfloat16	✓	✓	🟢	iron/operators/elementwise_mul/
GEMM	General Matrix Multiplication kernel	bfloat16	✓	✓	🟢	iron/operators/gemm/
GEMV	General Matrix-Vector Multiplication kernel	bfloat16	✓	✓	🟢	iron/operators/gemv/
GQA	Grouped Query Attention kernel (Single pipeline)	bfloat16		✓	🟢	iron/operators/mha/

[Public]

Triton for AMD NPU:

🔗 https://github.com/amd/Triton-XDNA

README | MIT license

Triton-XDNA

An experimental open-source project demonstrating compiler-driven kernel generation for AMD XDNA NPUs using Triton and MLIR-AIR.

Triton-XDNA provides an end-to-end compilation flow that lowers standard Triton kernels directly to AMD NPU hardware — no prebuilt kernel libraries required. It bridges Triton's high-level parallel programming model with AMD's MLIR-AIR/AIE compilation stack, producing XRT-compatible binaries for AMD AI Engine architectures (AIE2 and AIE2P).

How it works

Triton kernels are first lowered to compact Linalg compute graphs via triton-shared, then tiled and mapped onto parallel NPU cores using the MLIR Transform dialect, and finally compiled through MLIR-AIR and MLIR-AIE to produce device binaries.

```
Triton kernel (@triton.jit)
  -> triton-shared (Linalg)
    -> MLIR Transform dialect (tiling, bufferization, vectorization)
      -> MLIR-AIR / MLIR-AIE
        -> XRT binary (aie.xclbin)
```

AMD
together we advance_

[Public]

Exercise 7: Programming Examples

Basic

Design name	Data type	Description
Vector Scalar Add	i32	Adds 1 to every element in vector
Vector Scalar Mul	i32	Returns a vector multiplied by a scale factor
Vector Vector Add	i32	Returns a vector summed with another vector
Vector Vector Modulo	i32	Returns vector % vector
Vector Vector Multiply	i32	Returns a vector multiplied by a vector
Vector Reduce Add	bfloat16	Returns the sum of all elements in a vector
Vector Reduce Max	bfloat16	Returns the maximum of all elements in a vector
Vector Reduce Min	bfloat16	Returns the minimum of all elements in a vector
Vector Exp	bfloat16	Returns a vector representing e^x of the inputs
DMA Transpose	i32	Transposes a matrix with the Shim DMA using npu_dma_memcpy_nd
Matrix Scalar Add	i32	Returns a matrix multiplied by a scalar
Single core GEMM	bfloat16	A single core matrix-matrix multiply
Multi core GEMM	bfloat16	A matrix-matrix multiply using 16 AIEs with operand broadcast. Uses a simple "accumulate in place" strategy
GEMV	bfloat16	A vector-matrix multiply returning a vector

ML Operators

Design name	Data type	Description
Eltwise Add	bfloat16	An element by element addition of two vectors
Eltwise Mul	i32	An element by element multiplication of two vectors
ReLU	bfloat16	Rectified linear unit (ReLU) activation function on a vector
Softmax	bfloat16	Softmax operation on a matrix
Conv2D	i8	A single core 2D convolution for CNNs
Conv2D+ReLU	i8	A Conv2D with a ReLU fused at the vector register level

AMD
together we advance_

Copyright and Disclaimer

- ©2026 Advanced Micro Devices, Inc. All rights reserved.
- AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.
- The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.
- THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION

AMD