



Introduction to the Versal ACAP AI Engine and to its programming model

Course Objective

After completing this course, you will be able to:

- Describe the Versal ACAP architecture
- Understand the data movement in the AI Engine
- Understand the AI Engine tile architecture
- Implement a system-level design for Versal® ACAPs with the Vitis™ tool flow
- Use a VCK5000 to run system-level designs

Agenda

- 01. Overview of the Versal ACAP Architecture
- 02. AI Engine Value Proposition
- 03. Introduction to the AI Engine Architecture
- 04. AI Engine Memory and Data Movement
- 05. Scalar and Vector Data Types
- 06. AI Engine APIs and Intrinsic Functions
- 07. Window and Streaming Data APIs
- 08. Versal ACAP Tool Flow
- 09. VCK5000 Platform
- 10. The Programming Model - Single Kernel
- 11. Introduction to the Adaptive Data Flow Graph
- 12. Vitis Analyzer
- 13. The Programming Model - Multiple Kernels Using Graphs
- 14. AIE DSP Library Overview
- 15. Application Partitioning on Versal ACAPs
- 16. Introduction to AI Engine APIs for Arithmetic Operations
- 17. AIE-ML Array Architecture



Overview of the Versal ACAP Architecture

Objectives

After completing this module, you will be able to:

- Describe the Versal® ACAP architecture at a high level
- Identify the various engines in the Versal® ACAP device
- Identify the different Versal® ACAP families

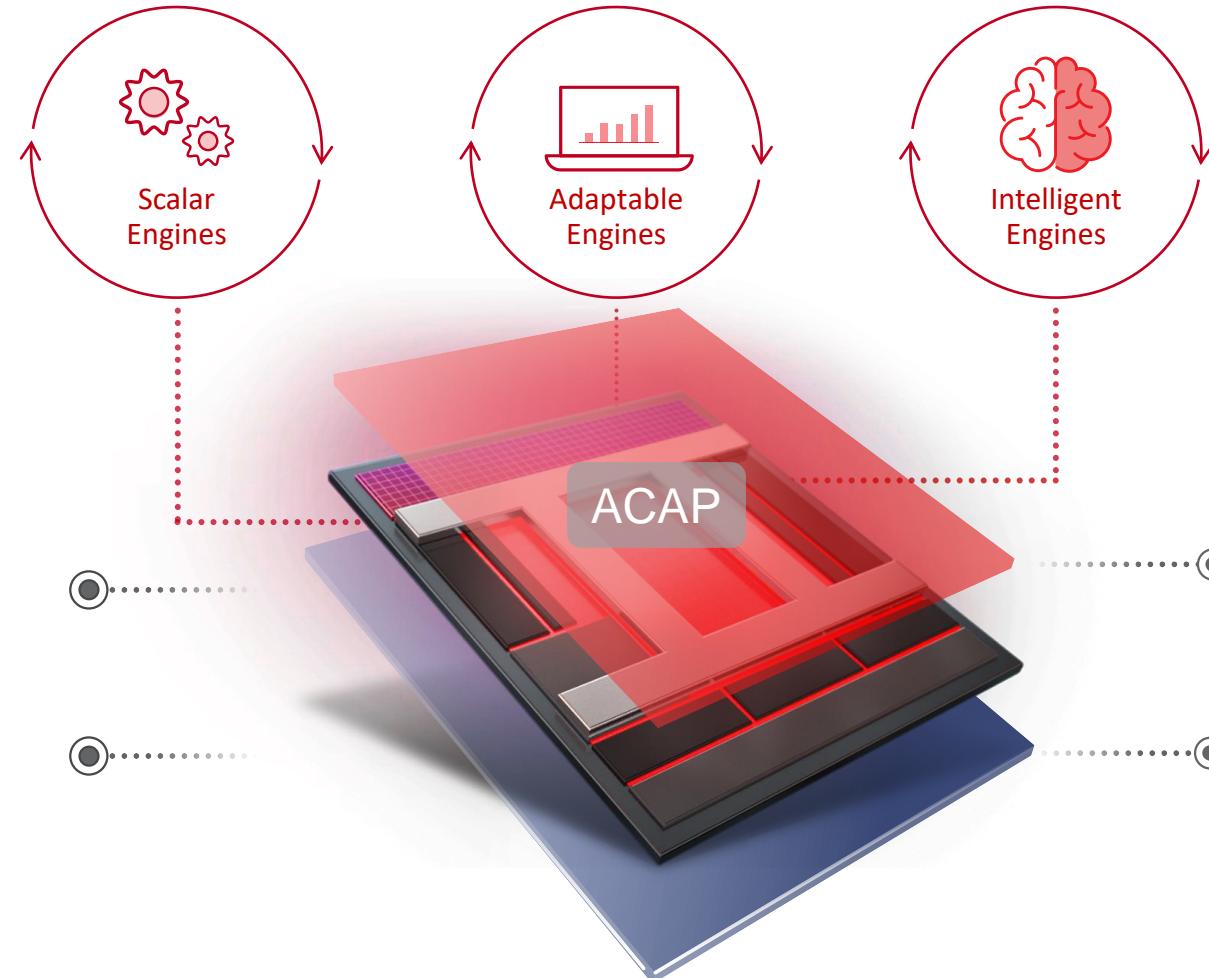
Adaptive Compute Acceleration Platform (ACAP)

COMPUTE ACCELERATION

ADAPTIVE

Future Proof for
New Algorithms

Support for Common
Standards



Customizable memory
hierarchies

NoC to move data

PLATFORM

Single Heterogeneous Platform
for HW/SW Engineers

Frameworks, Libraries, SW,
Runtime Stack

Versal Architecture: Overview



Scalar Engines

- Platform control
- Embedded edge compute



PCIe Gen5 & CCIX

- 2x PCIe and DMA bandwidth
- Cache coherency



DDR4 Memory

- 3200-DDR4, 4266-LPDDR4
- 2x bandwidth/pin



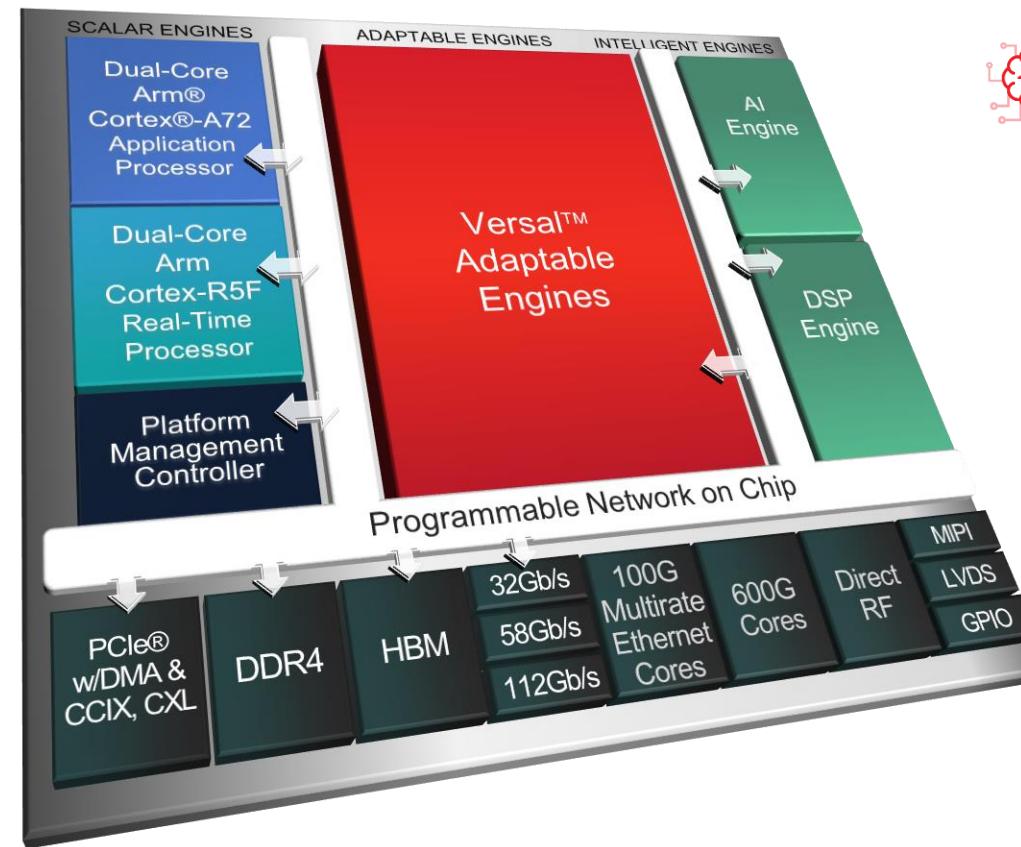
Transceiver Leadership

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



Adaptable Engines

- 2x compute density
- Voltage scaling for perf/watt



Intelligent Engines

- AI compute
- Diverse DSP workloads



Programmable NoC

- Guaranteed bandwidth
- Enables SW programmability



Protocol Engines

- 400G/600G cores
- Power optimized



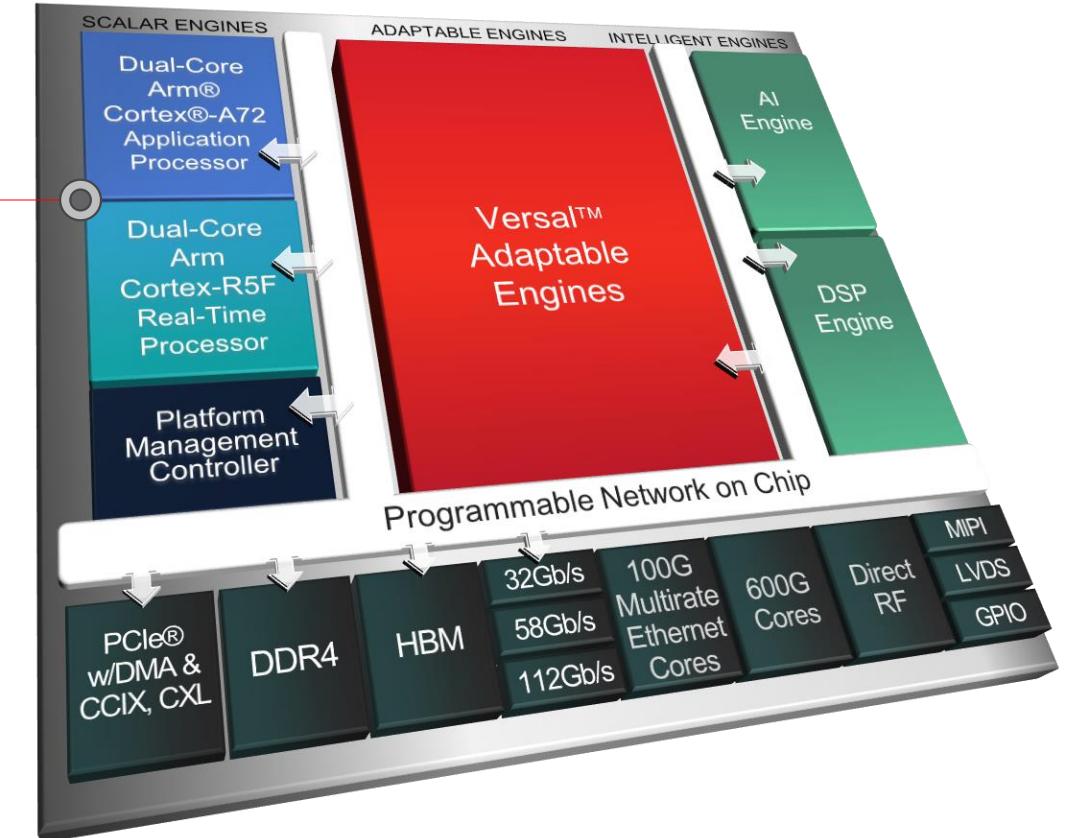
Programmable I/O

- Any interface or sensor
- Includes 3.2Gb/s MIPI

Scalar Engines

Scalar Engines for Platform Management

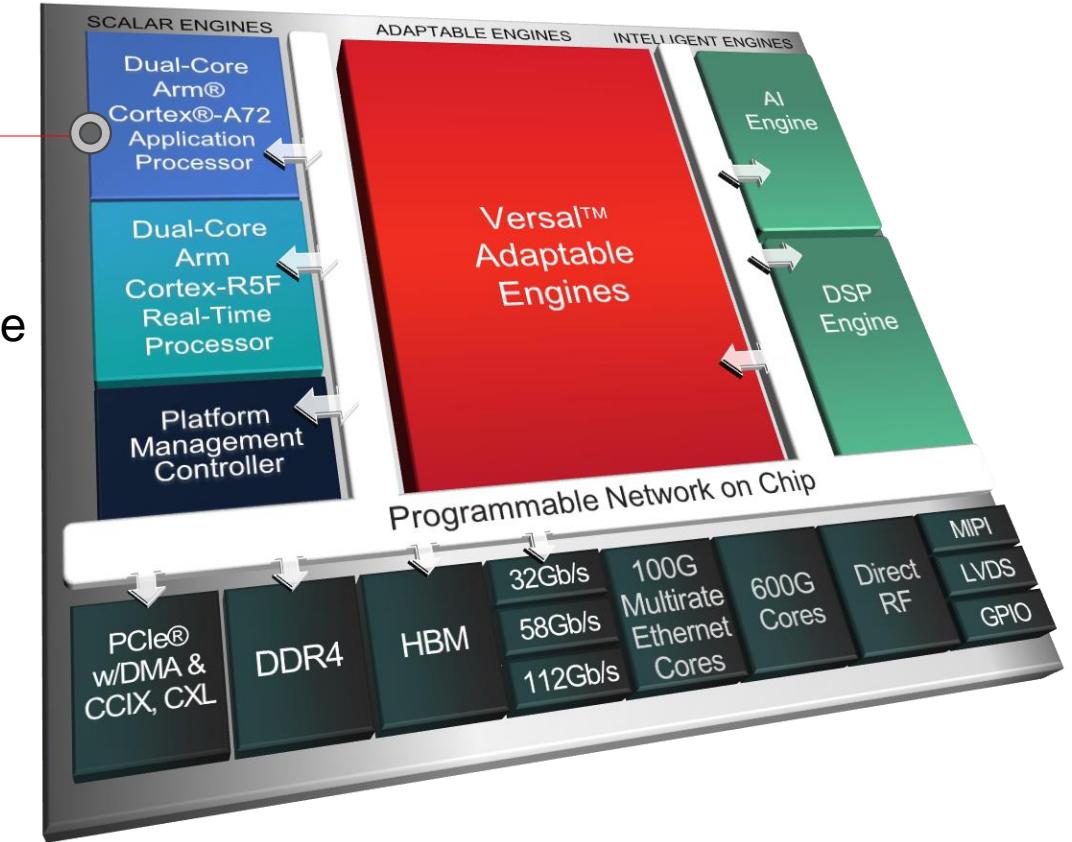
- Execute complex algorithms and decision making for autonomous systems
- Provide safety processing and redundancy for mission- and safety-critical applications
- Manage the entire platform
- Load each aspect of the ACAP and monitor status
- Support capability extension
 - PL-instantiated MicroBlaze™ processor



Scalar Engines

Application Processor Unit

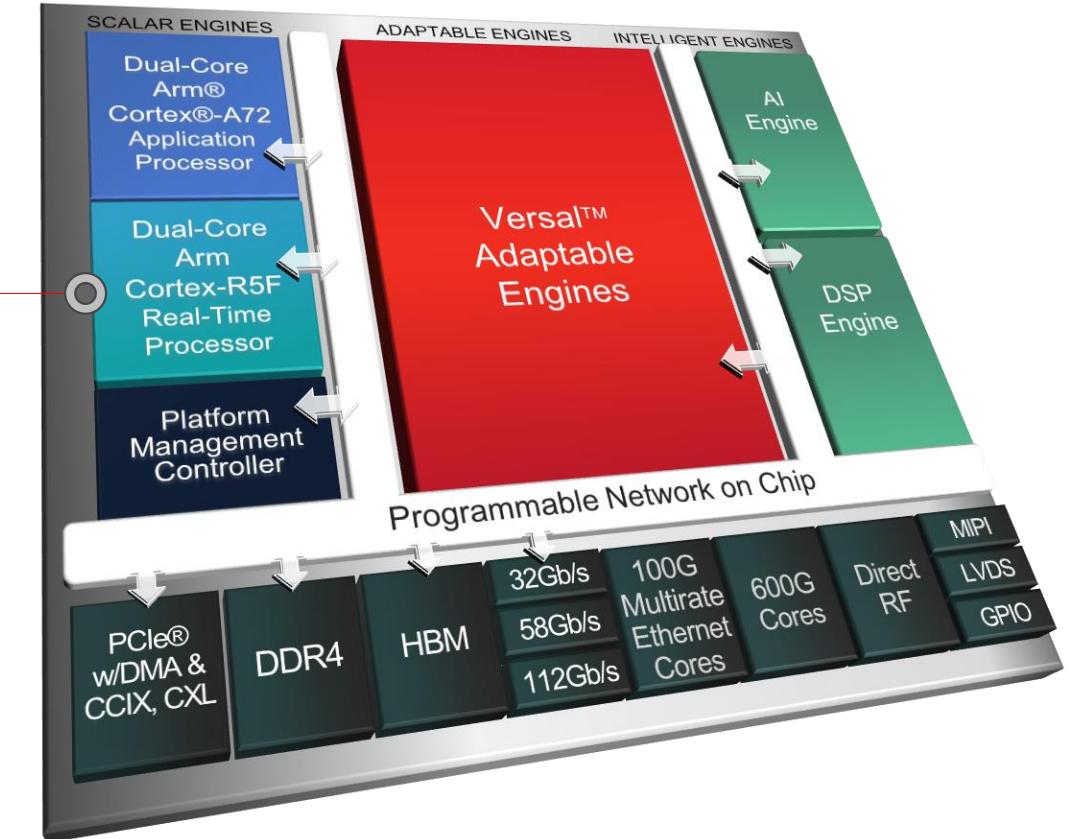
- Dual-core Arm Cortex-A72 application processor
- Up to 1.7 GHz speed – 2x single-threaded performance
- ARMv8 architecture
- Up in seconds
- Supports Linux and bare-metal



Scalar Engines

Real Time Processor Unit

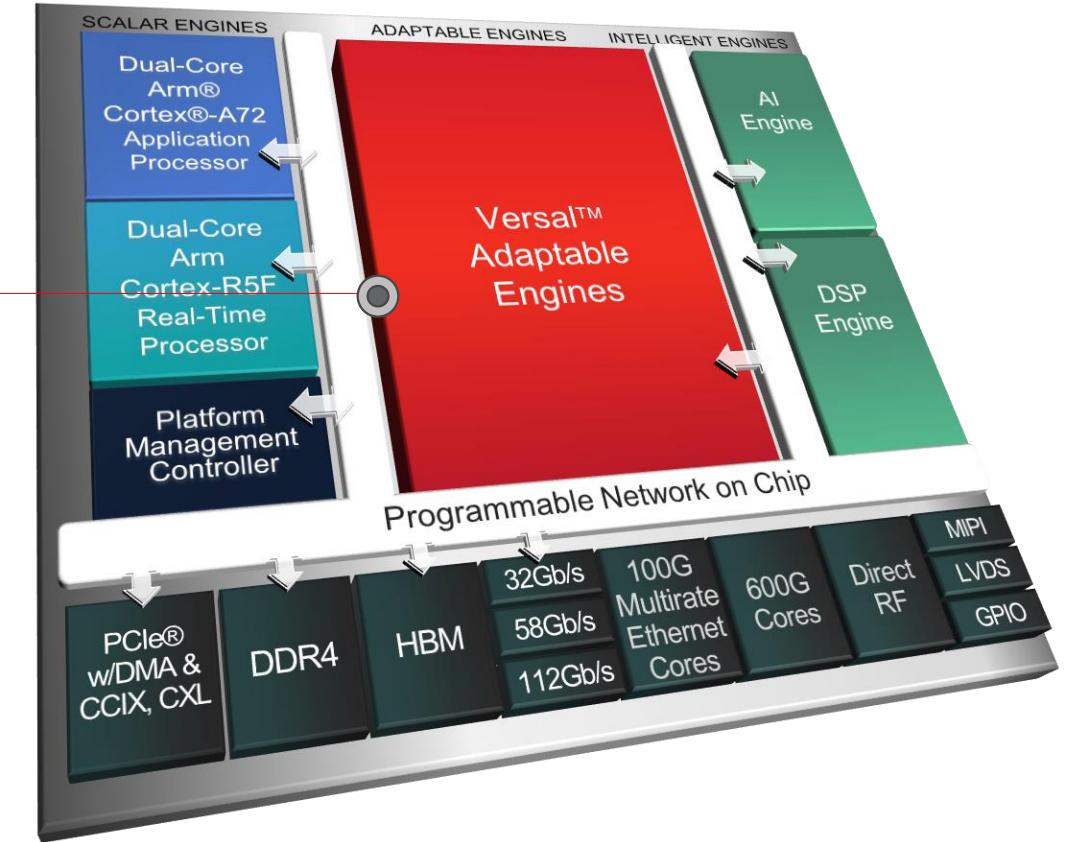
- Dual-core Arm Cortex-R5F real-time processor
- Functional safety
- Split mode for performance or lock step for safety
- Low latency, determinism, and real-time control for any application
- ASIL/SIL certifiable



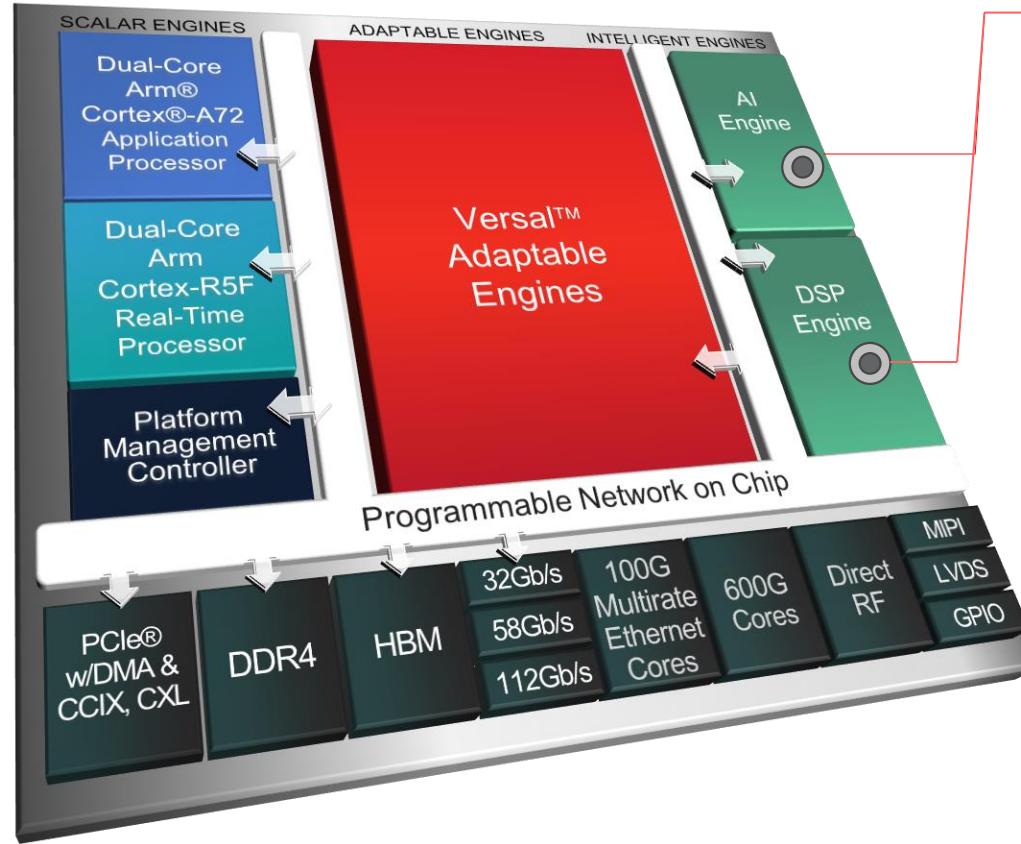
Adaptable Engines

Adaptable Hardware Engines – Programmable Logic

- Fine-grained parallel processing, data aggregation, and sensor fusion
- Programmable memory hierarchy to optimize compute efficiency



Intelligent Engines



Intelligent Engines for Diverse Compute

AI is everywhere

- Wired communications, automotive, and consumer markets

DSP Engines

- High-precision, floating-point computation support
- Offload additional functions for acceleration

AI Engines

- High throughput, low latency, deterministic, and power efficient
- Ideal for AI inference and advanced signal processing

Programmable NoC

Bridging Engines & Hard IP

High-bandwidth, Terabit Programmable NoC

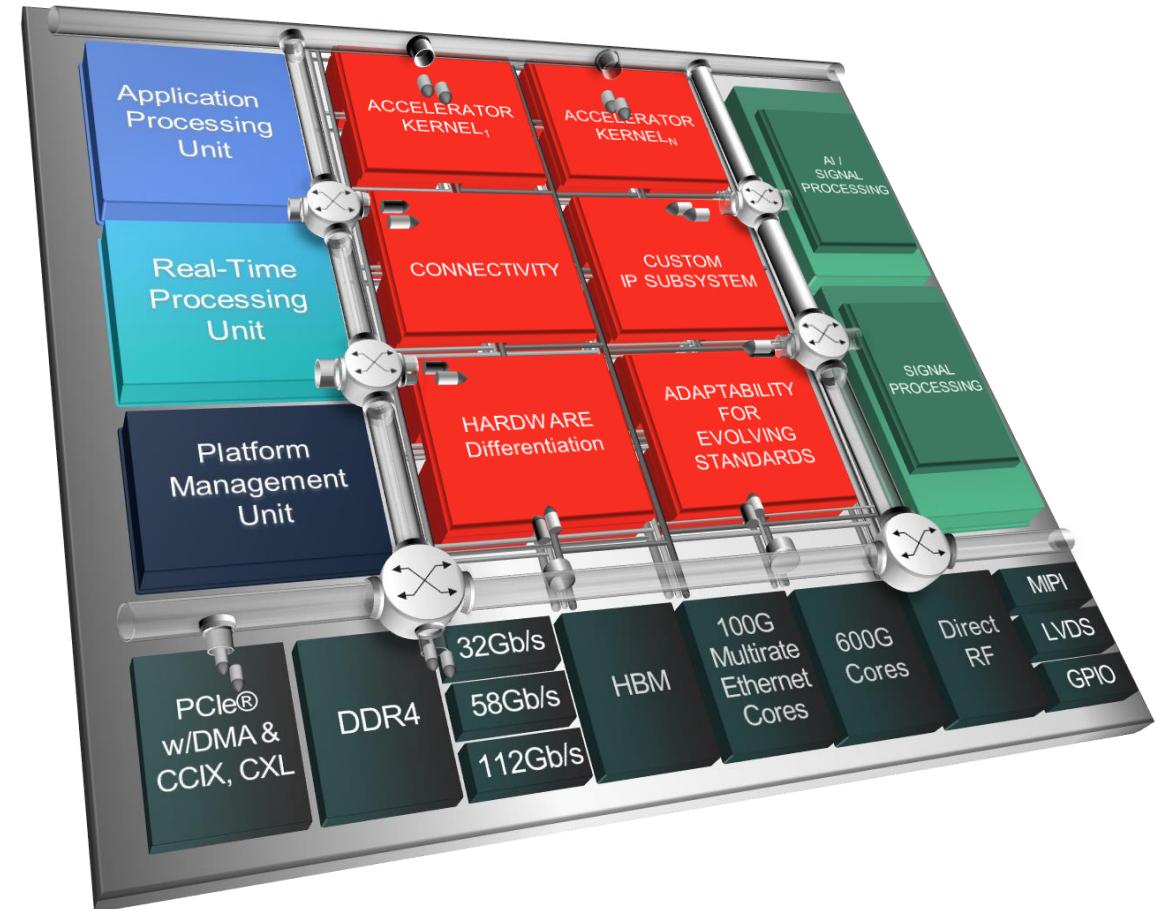
- Meet timing of critical paths
- Guaranteed QoS (bandwidth vs. latency)

Eases IP and Kernel Placement

- Simplifies connectivity of IP and peripherals
- Easily swap kernels at NoC port boundaries

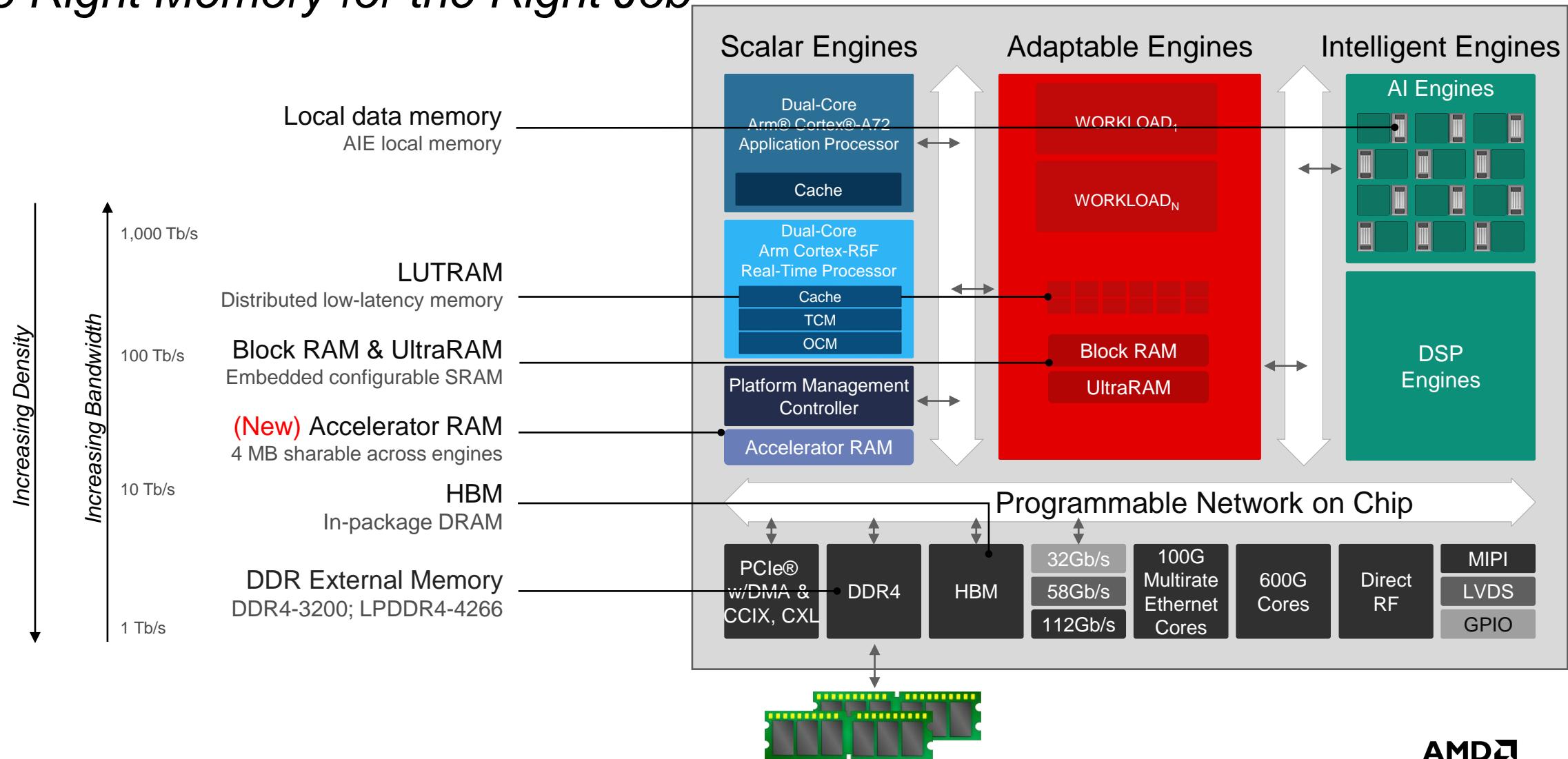
Programming Framework

- Memory-mapped access to all resources
- Built-in arbitration between engines and memory



Adaptable Memory Hierarchy

The Right Memory for the Right Job



AI Engine

Hardened Compute, Memory, and Interconnect

Array of VLIW / SIMD processors

- Running @ 1GHz+
- Versatile core for ML and other advanced DSP workloads

Massive array of interconnected cores

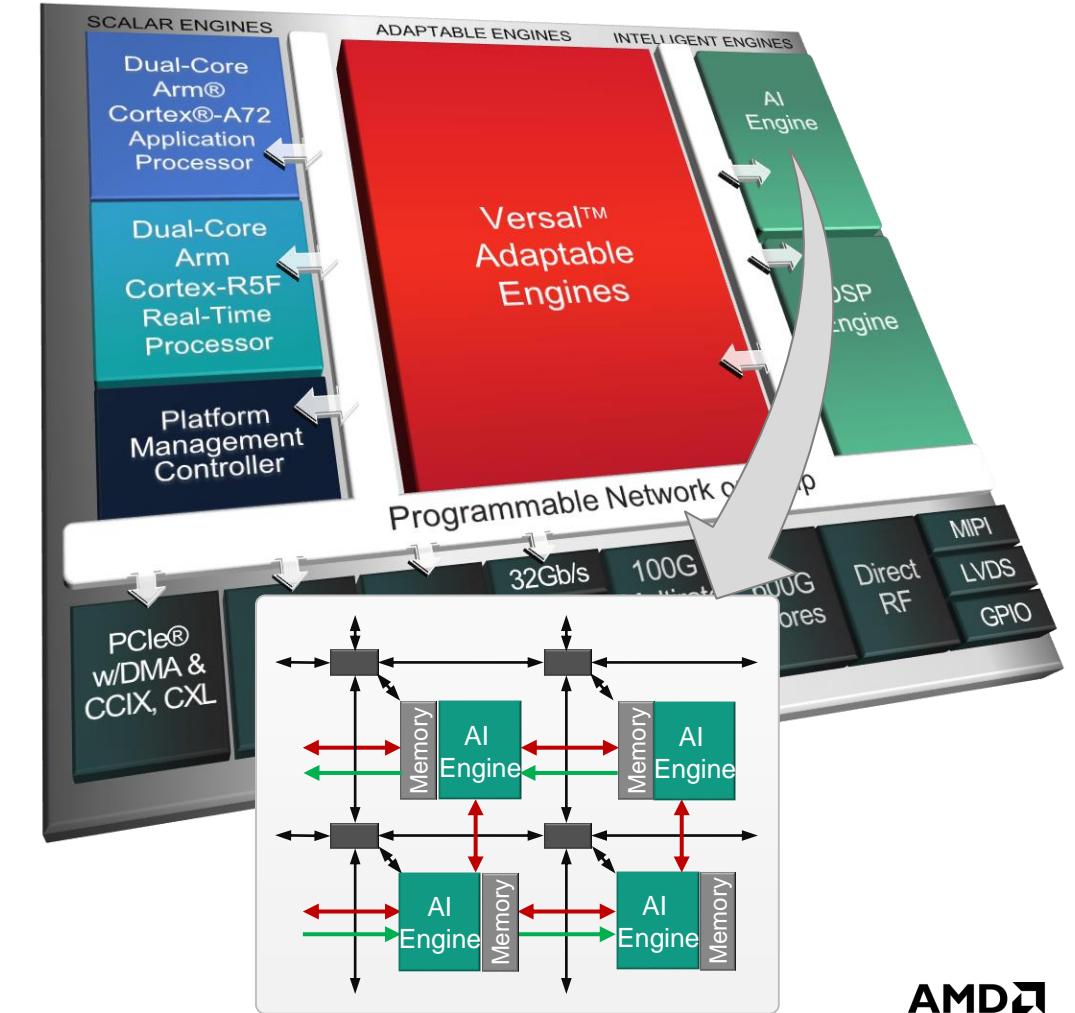
- Multiple tiles (10s to 100s) for scalable compute

TB/s of bandwidth to other engines

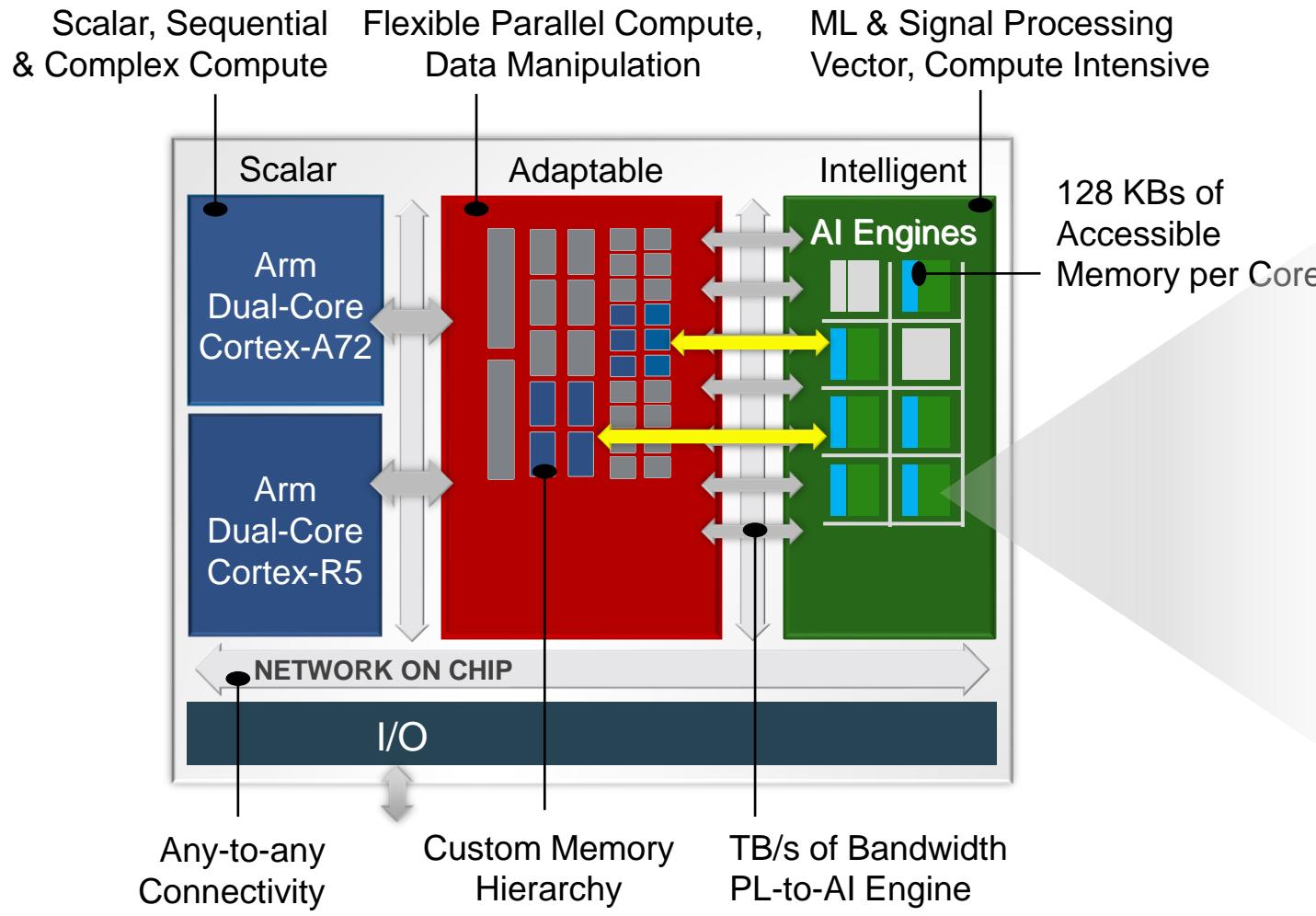
- Tile to tile
- Direct, massive throughput to adaptable hardware engines

Software Programmable

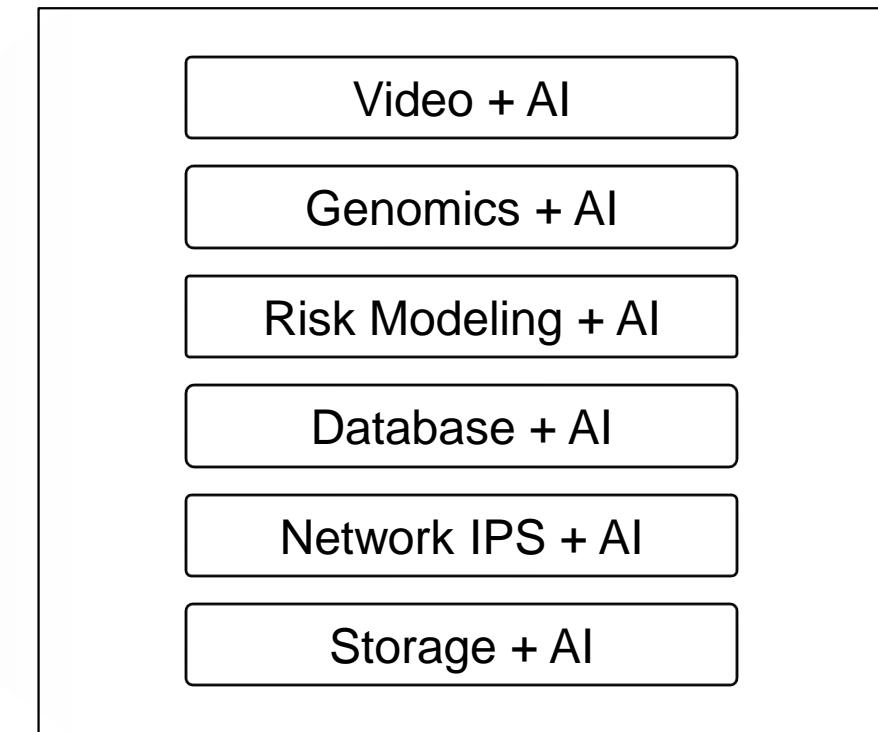
- C/C++ programmable, compile in minutes
- Libraries available



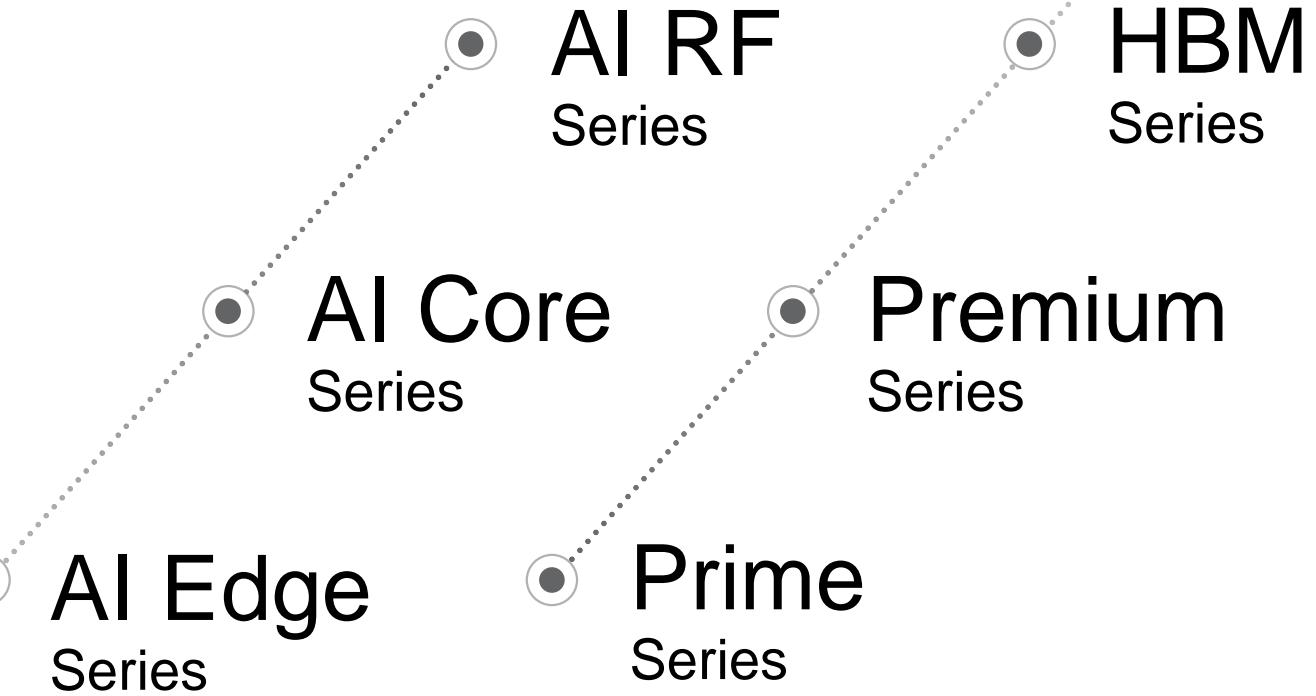
Hardware Adaptable: Accelerating the Whole Application



Heterogeneous Acceleration from Data Center to the Edge



AMD VERSAL



Versal ACAP Portfolio Overview

Resources & Capabilities	Prime Series	Premium Series	HBM Series	AI Edge Series	AI Core Series
Description	Signal processing and connectivity capabilities	High-end bandwidth series	Heterogeneous integration of HBM	Low Power AI for Real-Time Systems	Breakthrough AI Inference Throughput
LUT (K)	150-1,000	720-3,360	1,753-2,574	20-520	246-900
Distributed RAM (Mb)	4.6-31.2	22-103	54-79	0.6-15.9	7.5-27.5
Block RAM (Mb)	5.4-69.6	49-174	89-132	0.8-33.5	15.5-34
Ultra RAM (Mb)	43.6-190.4	453-2,549	366-541	6.8-129.9	58.8-130.2
Accelerator RAM (Mb)	0-32	-	-	0-32	0-32
DSP Engine	464-3,984	1,904-14,352	7,392-10,848	90-1,312	928-1,968
AI Engine	-	-	-	8-304	128-400
Serial Transceivers	8-48	48-168	88-128	0-44	8-44
Max. GT Bandwidth (Tb/s)	7.8	17.6	11.2	2.5	2.5
I/O	316-770	586-780	780	114-530	478-770
Memory Controllers	1-4	3-4	4	1-3	2-4
HBM (GB)	-	-	8-32	-	-

[Versal Architecture and Product Data Sheet: Overview](#)

Summary

- The Versal ACAP device contains:
 - Scalar Engines
 - Dual-core Arm Cortex-A72 application processor
 - Dual-core Arm Cortex-R5F real-time processor
 - Adaptable Engines
 - Programmable logic – millions of LUTs available for any workload
 - Extreme parallelism in compute
 - Intelligent Engines
 - AI Engines: High throughput, low latency, and power efficient
 - DSP Engines: High-precision floating point and low latency
- Whole application acceleration
 - Heterogeneous integration and acceleration through the Vitis™ unified software platform
- 6 Versal ACAP families



AI Engine Value Proposition

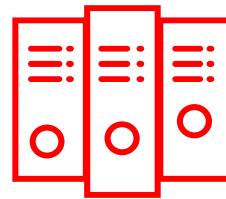
Motivation for AI Engine

AI IS EVERYWHERE

5G Wireless
Radio



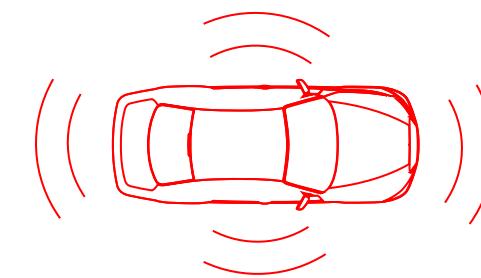
Data Center
Workloads



Smart
Factories



ADAS & AD



Machine
Learning



Technology Not
Scaling



Moore's Law / Dennard Scaling
Performance & Power Scaling
Traditional Single / Multi-core

Compute Density
Real Time Capability
Power Efficiency

Applications More
Demanding

Dynamic Markets Require Adaptable Compute Acceleration

Delivering Adaptable Compute Acceleration

	CPU Single → Multi-Core	GPU (Parallel)	ACAP	ASIC
SW Programmable	✓	✓	✓	✓
HW Adaptable	—	—	✓	—
Workload Flexibility	✓	✓	✓	—
High Throughput and Low Latency	—	—	✓	✓
Device / Power Efficiency	—	—	✓	✓

Development Time & Complexity

With ACAP AI Engine

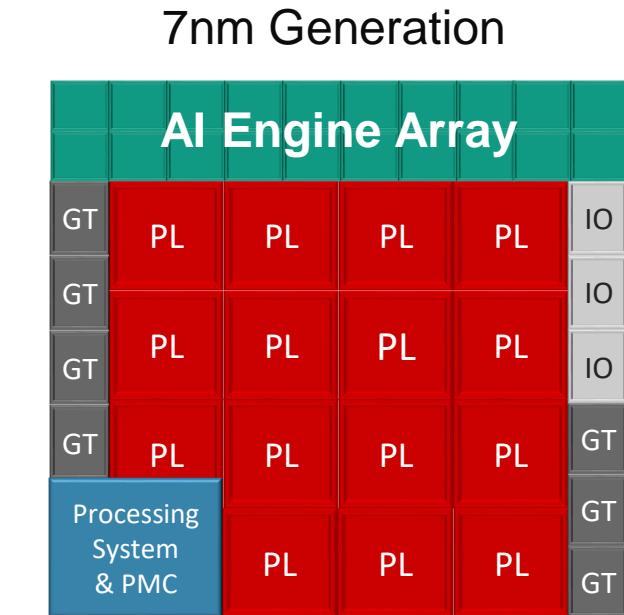
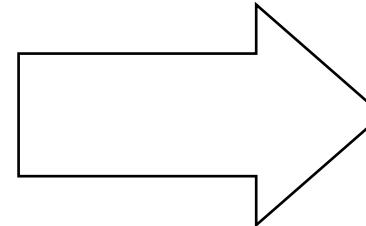
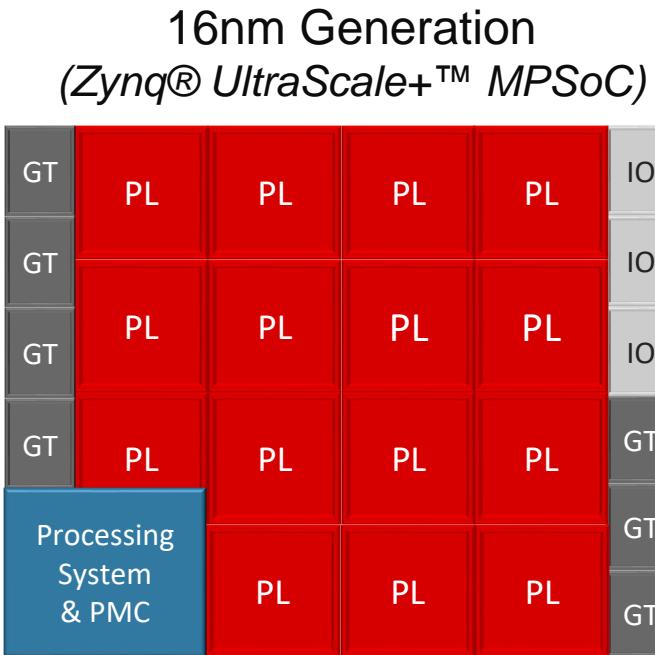
Weeks

Months

Years

Conceptual Device with High Compute

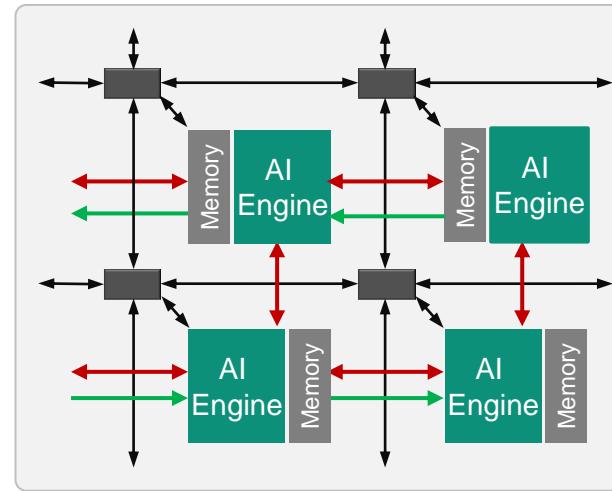
- Applications need (1) cost reduction, (2) power reduction, (3) more compute, (4) more programmability
- How does AMD-Xilinx meet the demands of these evolving applications?



Goal: Increase Compute Density and Silicon Efficiency

Introducing the AI Engine

**SW Programmable
Deterministic
Efficient**

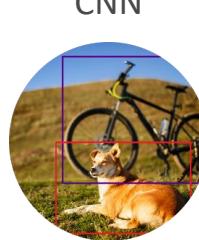


Target Applications

- 5G Wireless – Radio, Baseband
- Data Center - AI Inference
- ISM – Computer Vision, Genomics
- ADAS/AD – Embedded Vision
- A&D – Radar, Sonar

SIGNAL PROCESSING

ARTIFICIAL INTELLIGENCE



CNN



LSTM

MLP



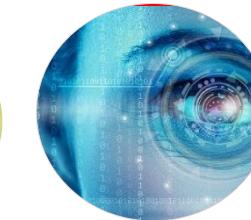
WIRELESS



WIRELESS



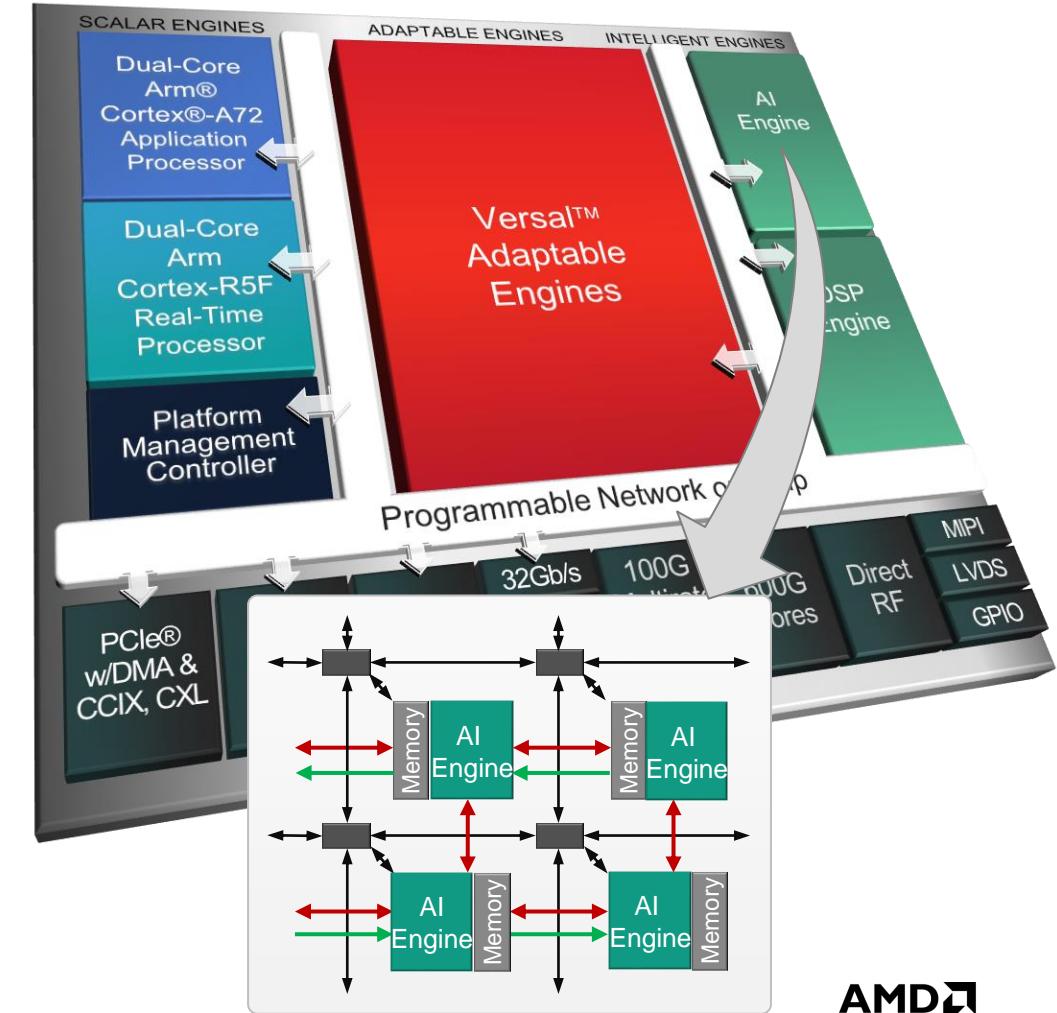
COMPUTER VISION



Adaptable. Intelligent.

AI Engine - Hardened Compute, Memory, and Interconnect

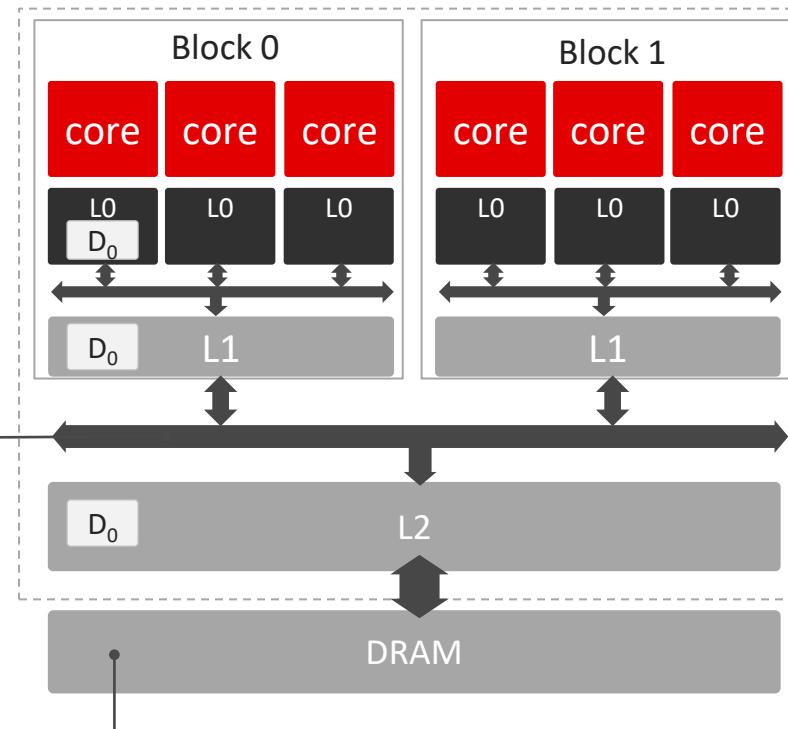
- Huge performance improvements vs. UltraScale+™ FPGAs
 - Up to 8x compute density @ 40% lower power
- 1GHz+ VLIW / SIMD vector processors
 - Versatile core for ML and other advanced DSP workloads
- Massive array of interconnected cores
 - Instantiate multiple tiles (10s to 100s) for scalable compute
- Terabytes/sec of interface bandwidth to other engines
 - Direct, massive throughput to adaptable HW engines
 - Implement core application with AI for “Whole App Acceleration”
- SW programmable for any developer
 - C programmable, compile in minutes
 - Library-based design for ML framework developers



AI Engine: Reinventing Multi-Core Compute

Traditional Multi-core

(cache-based architecture)



Fixed, shared Interconnect

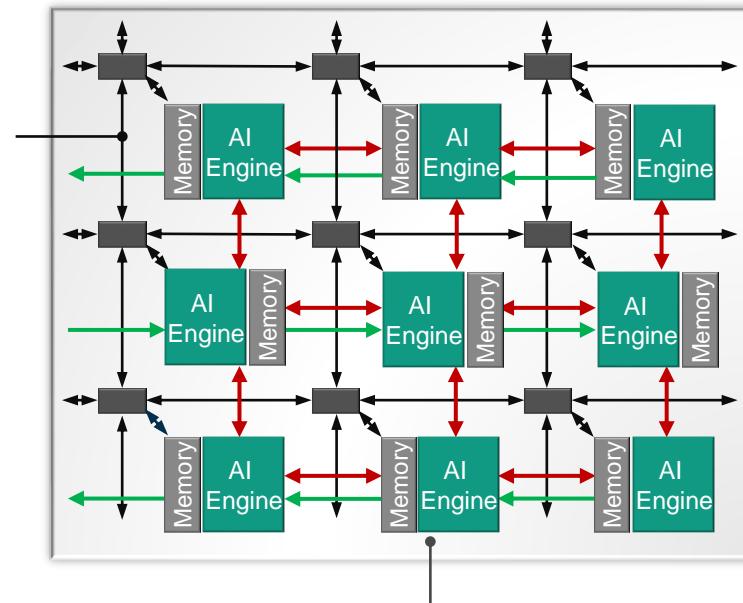
- Blocking limits compute
 - Timing not deterministic

Data Replicated

- Robs bandwidth
 - Reduces capacity

AI Engine Array

(intelligent engine)



Dedicated

- Non-blocking
 - Deterministic

Local, Distributed Memory

- No cache misses
 - Higher bandwidth
 - Less capacity required

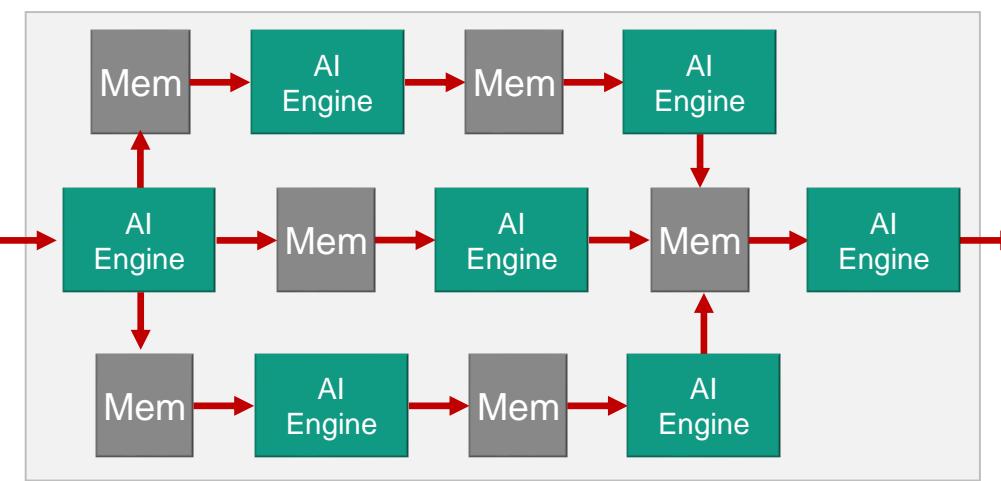
Data Movement Architecture

Memory Communication

Dataflow Pipeline



Dataflow Graph



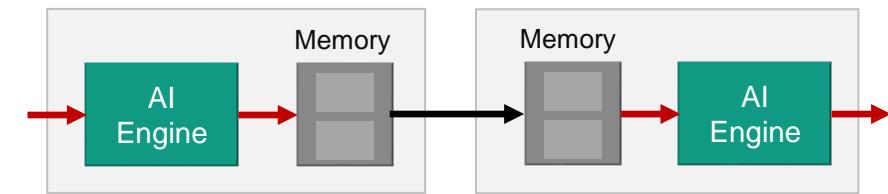
→ Memory Interface

→ Stream Interface

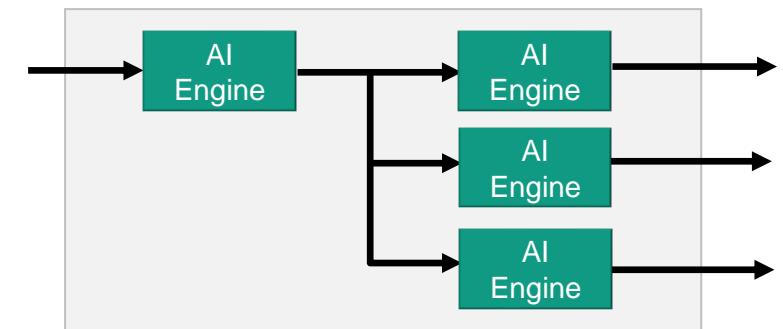
→ Cascade Interface

Streaming Communication

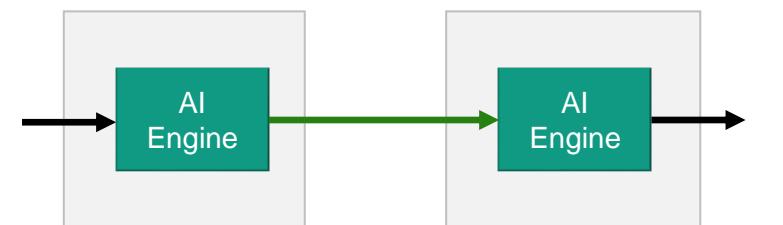
Non-Neighbor



Streaming Multicast

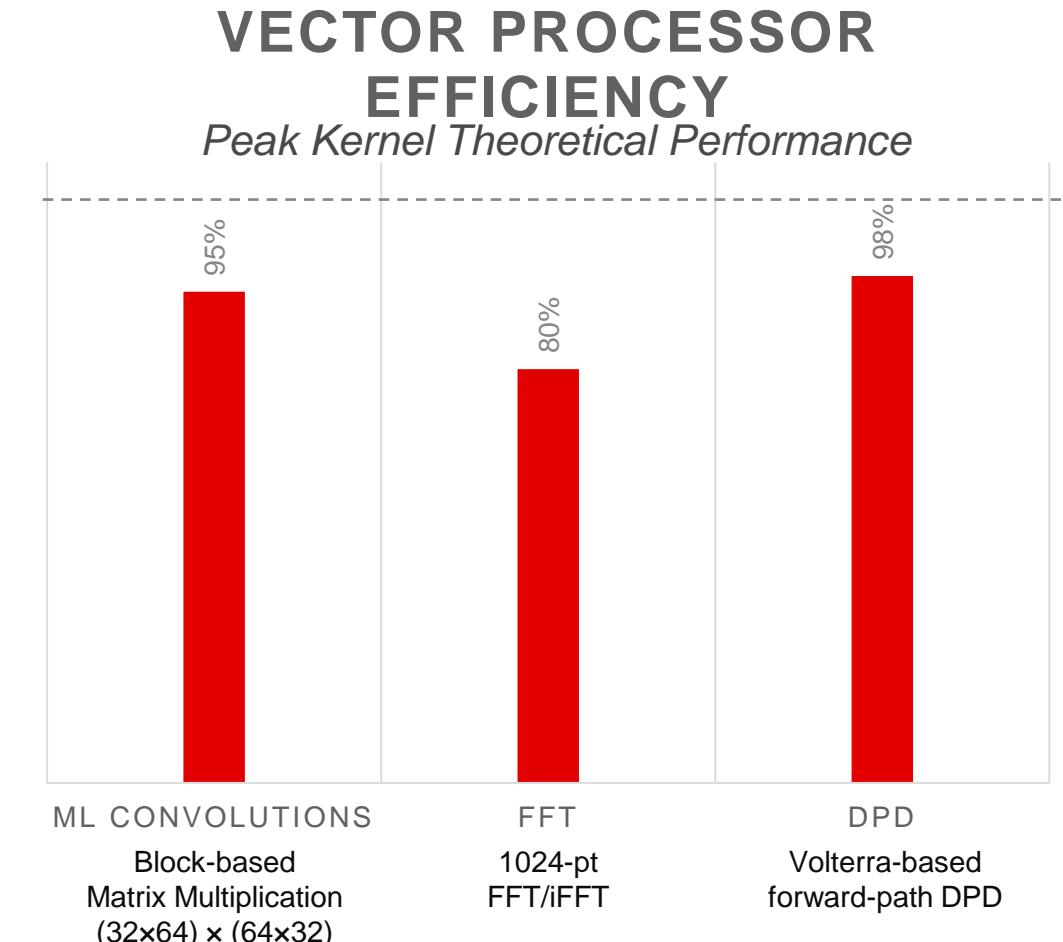
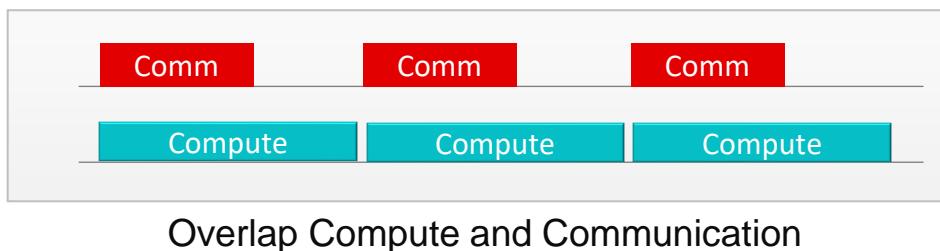


Cascade Streaming



AI Engine Delivers High Compute Efficiency

- Adaptable, non-blocking interconnect
 - Flexible data movement architecture
 - Avoids interconnect “bottlenecks”
- Adaptable memory hierarchy
 - Local, distributed, shareable = extreme bandwidth
 - No cache misses or data replication
 - Extend to PL memory (BRAM, URAM)
- Transfer data while AI Engine Computes

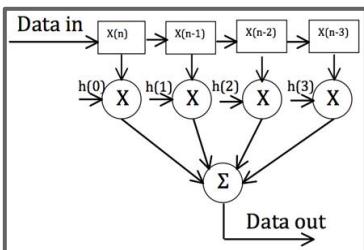


Representative AI Engine Algorithms

$$\begin{bmatrix} 0 & 2 & 5 & 2 \\ .4 & 0 & 0 & 0 \\ 0 & .5 & 0 & 0 \\ 0 & 0 & .6 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 18 \\ 2 \\ 1 \\ 1.2 \end{bmatrix}$$

Linear Algebra

Matrix-Matrix Multiplication
Matrix-Vector Multiplication



Convolution

FIR Filters
2-D Filters

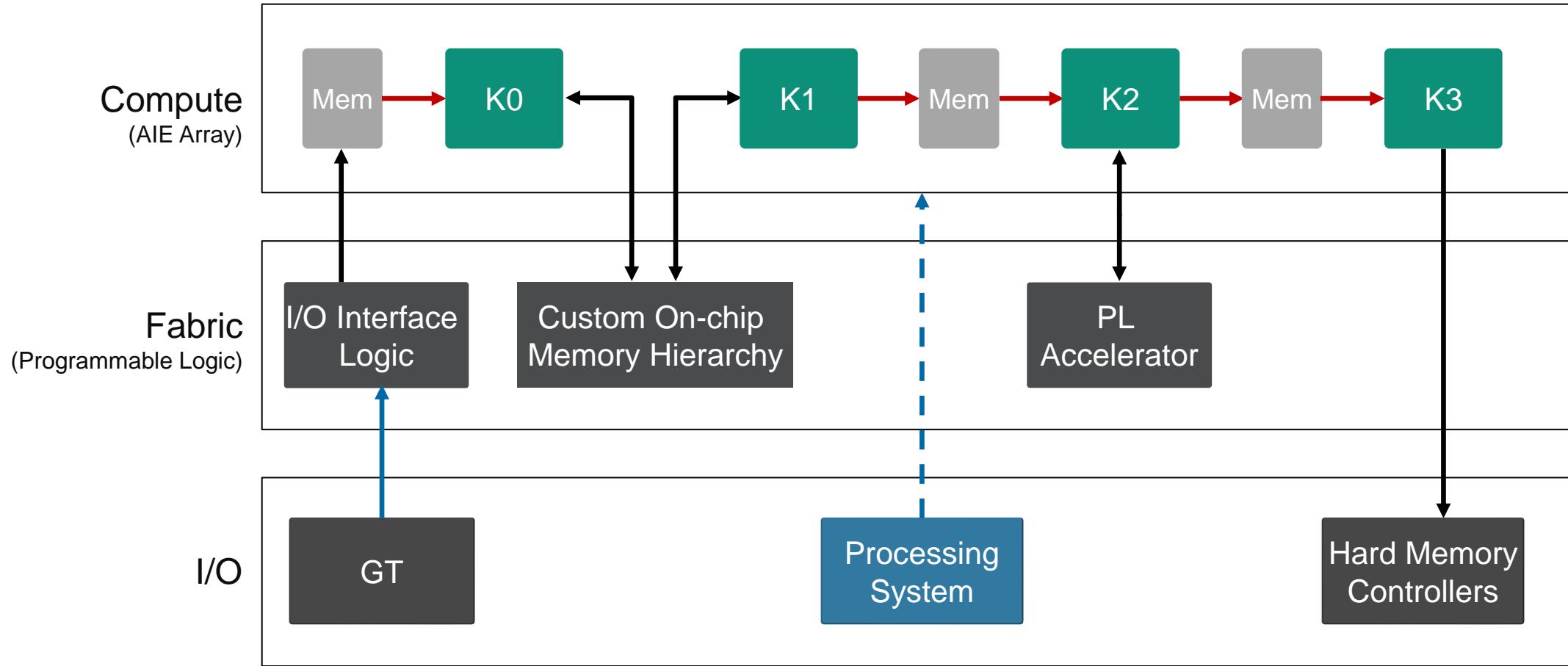
$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(\frac{x}{N})}$$

$$f(n) = \frac{1}{N} \sum_{x=0}^{N-1} F(x) e^{j2\pi(\frac{x}{N})}$$

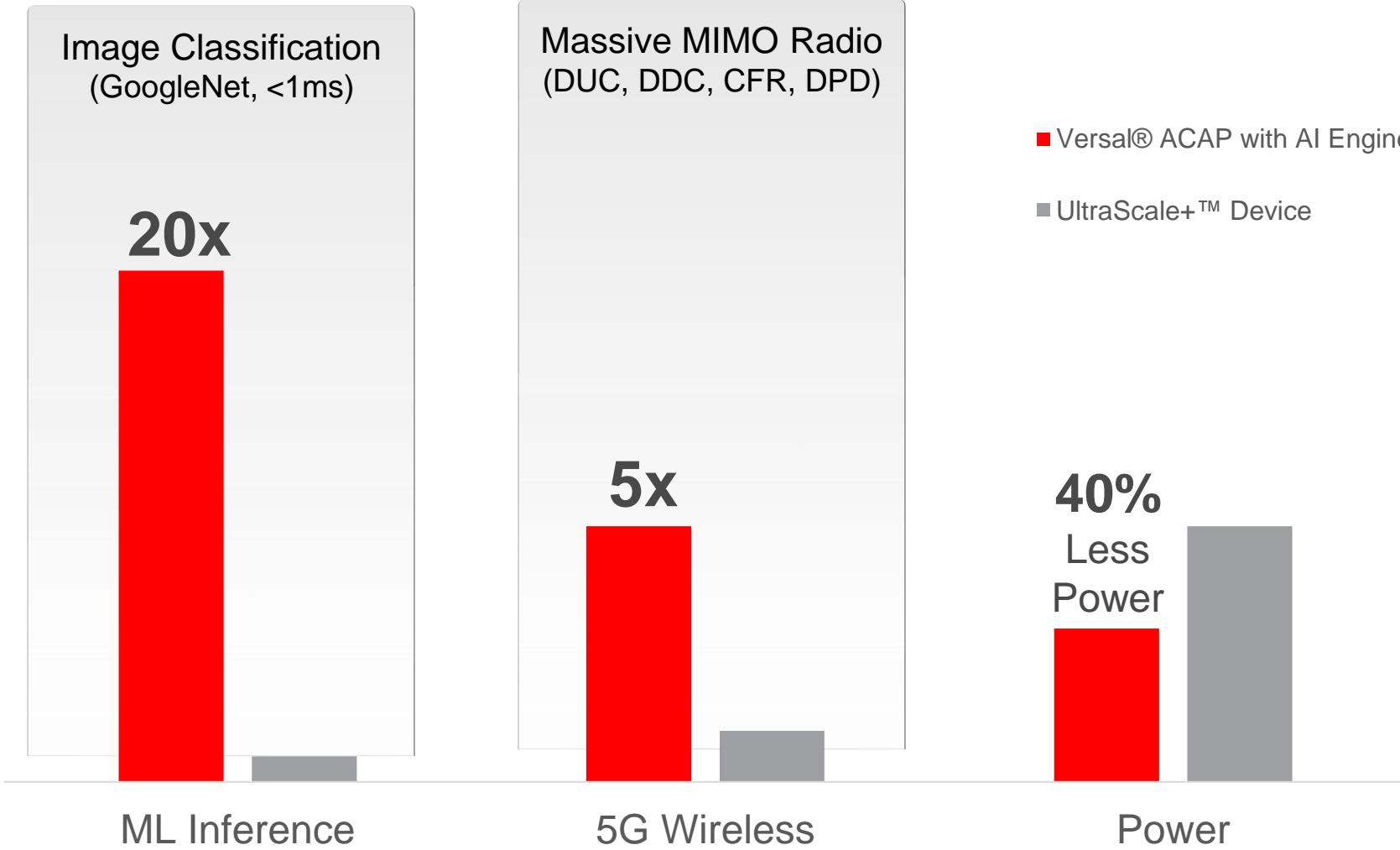
Transforms

Fast Fourier Transforms (FFTs/iFFTs)
Discrete Cosine Transforms (DCT)

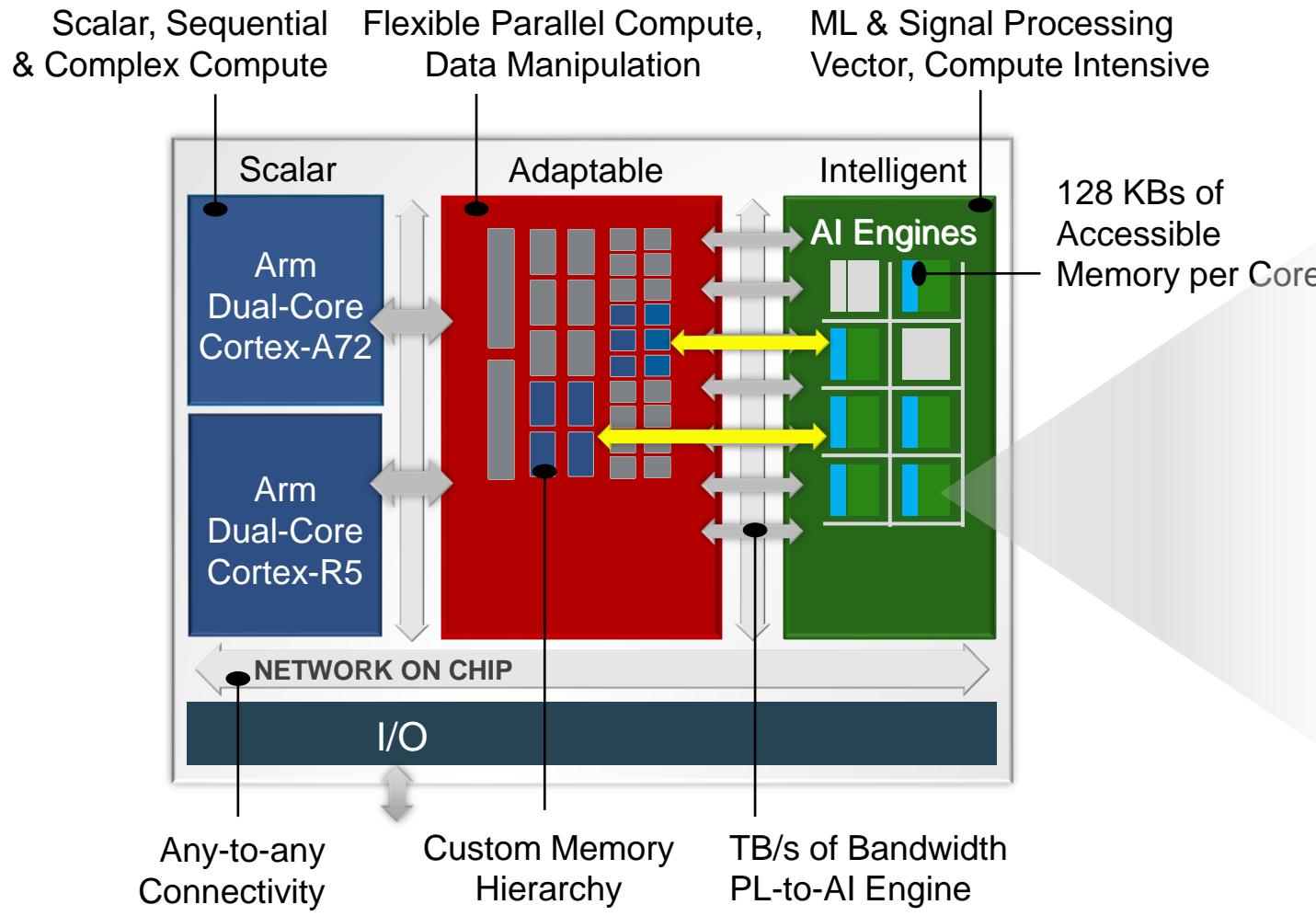
Data Movement Architecture: Device Integration



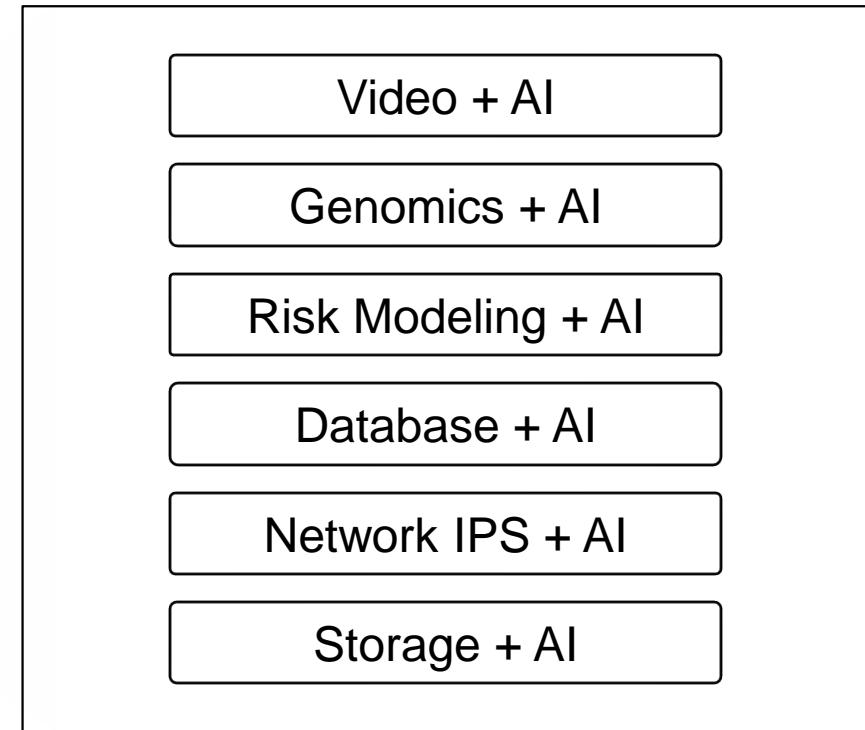
AI Engine Application Performance & Power Efficiency



Hardware Adaptable: Accelerating the Whole Application



**Heterogeneous Acceleration
from Data Center to the Edge**

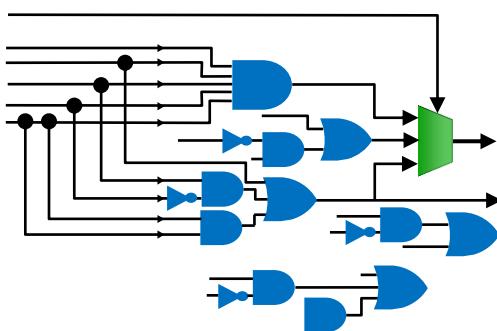


Intelligent Engines in Versal® ACAP → AI and DSP Engines

FPGA Fabric DSP

LUT and Memory

RTL Entry

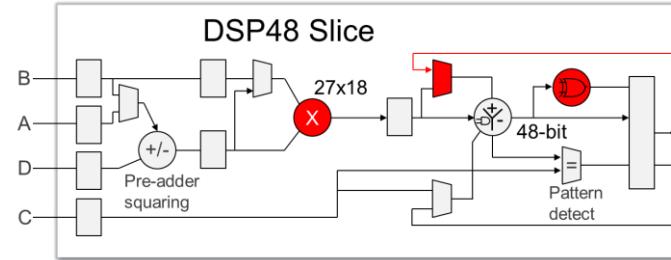


Increasing Capability

DSP48E2 Slice

Hardened MULT & ADDERS
 $ACC = ACC + (A \times B)$

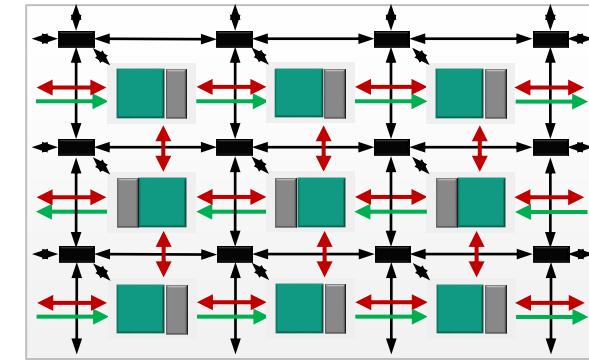
RTL Entry



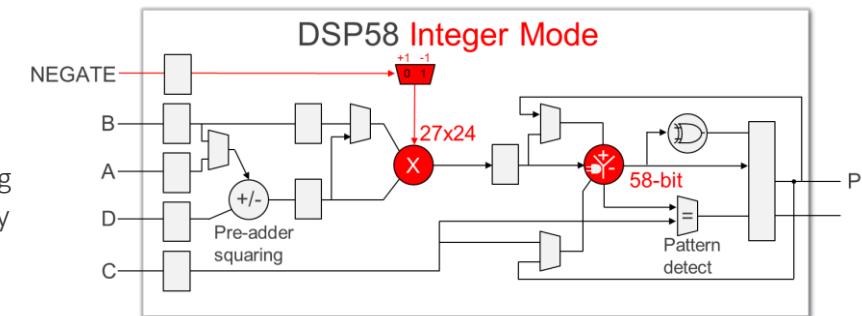
Increasing Capability

Increasing Capability

AI Engine⁽¹⁾ 2D Array
VLIW and SIMD Architecture
C/C++ Programmable



DSP Engine
Additional Features
RTL Entry



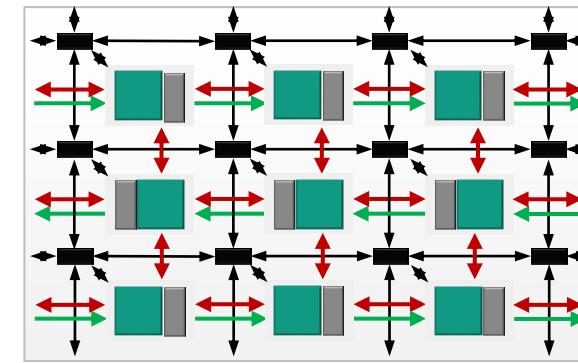
SPFP32 Native support

AI Engines and DSP Engines

- Why AI Engine?
 - Massive compute performance
 - S/W programmable (C/C++)
 - Fast compile – increase productivity
- Why DSP Engine?
 - Existing RTL/HLS IP usage
 - Additional features not available in AI Engine, e.g., 58-bit Logic Unit
 - Pre/Post processing to/from AI Engine
- Why both?
 - Versal® ACAP accelerates the complete application
 - AI Engine efficiency + PL flexibility (inc. DSP)

AI Engine⁽¹⁾ 2D Array

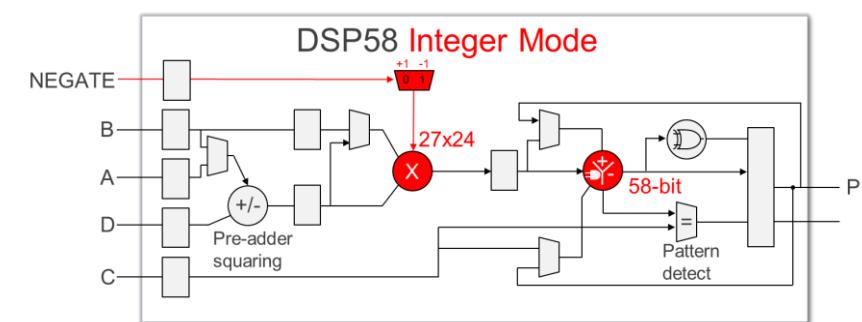
VLIW and SIMD Architecture
C/C++ Programmable



DSP Engine

Additional Features

RTL Entry



Designs with Versal® AI Core Device- DSP & PL v AI Engine

Key System Level Considerations

Design Partition at the system level – not in isolation (i.e., not DSP vs AIE)

Compute

- Compute/Function types
 - Multiply
 - Multiply & Accumulate
 - FFT
 - FIR etc

- Data Widths
 - AIE native 8/16/32 bit
 - DSP & PL – variable

- Vectorizable
 - Can the dataflow be vectorize

Memory

- Custom Mem Hierarchy
 - URAMs & BRAMs
 - LUTRAMs
 - AIE Data Memory
 - 32 & 128KB
 - Maybe external DDR

- Type of Mem Access
 - Random access
 - Structured v Unstructured Data
 - Data reuse or sharing

Throughput / Latency

- Throughput or Latency
 - AIE can provide both
 - Most applications lean towards one or the other
 - Decisions on partitioning likely to affect those parameters and need to be understood

Data Movement

- Data movement key factor
 - Data into and out of AI Engine array
 - Data around the array (Cascade, AXI etc.)
 - PL interface for Data
 - NoC for trace & debug

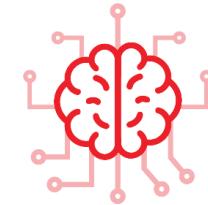
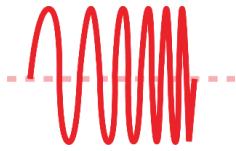
- Data Movement Libraries
 - Must simplify the problem for users via utilities and libraries

Power

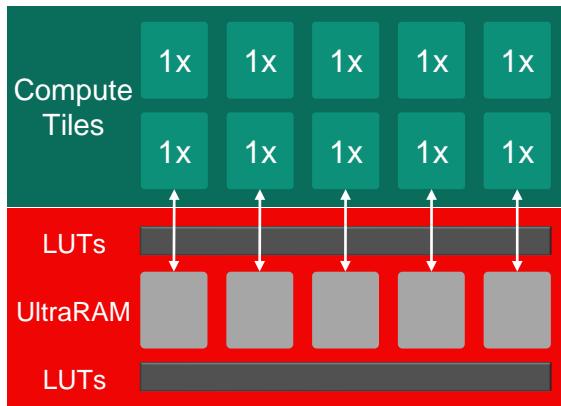
- AIE brings performance
 - Perf/watt
 - Perf/watt/mm²

Managing these key system level considerations enables Versal AI Core meet expectations and deliver the promised perf/watt

Intelligent Engines Optimized for Any AI Application



AIE Architecture



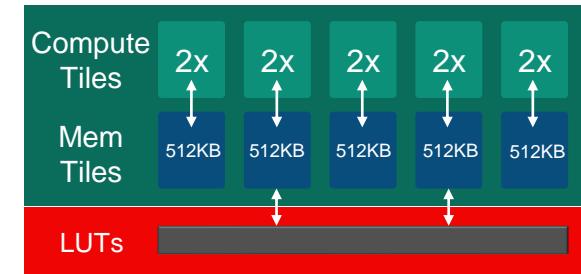
**2x INT8/16 OPs/Tile
4x INT4 OPs/Tile**

Reduced data movement

Reduced AI PL Footprint

- Optimized for signal processing and ML
- Flexibility for high performance DSP applications
- Native support for INT8, INT16, INT32, FP32

AIE-ML Architecture



- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16
- Fine grained sparsity HW optimization
- Enhanced FFT & complex math support

Versal® AIE-ML offers 2x AI Performance per Watt

AI Engine Architecture Summary

- Compute Efficiency
 - 8x increase in compute density
 - At ~40% lower power
- Heterogeneous Architecture
 - High-throughput, low-latency
 - PL flexibility – accelerate the complete application
 - Custom memory hierarchy
- Multiple Applications
 - Data Center: AI inference
 - Comms: wireless 5G – radio (DFE), baseband, Wired – cable access
 - ADAS/AD: embedded vision
- Software Programmable
 - C/C++
 - Optimized libraries
- Two versions of AI Engines
 - AIE optimized for DSP
 - AIE-ML optimized for ML



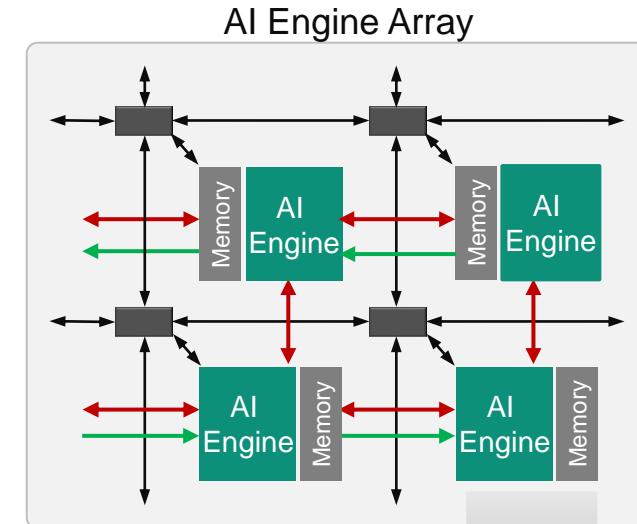
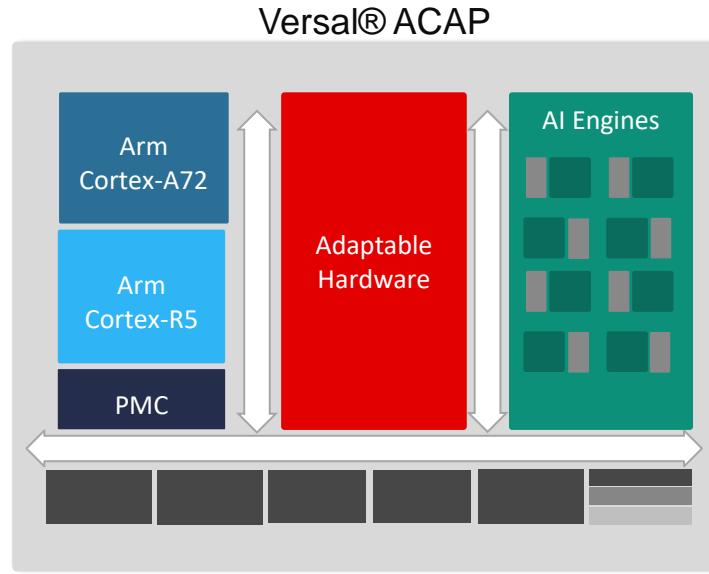
Introduction to the AI Engine Architecture

Objectives

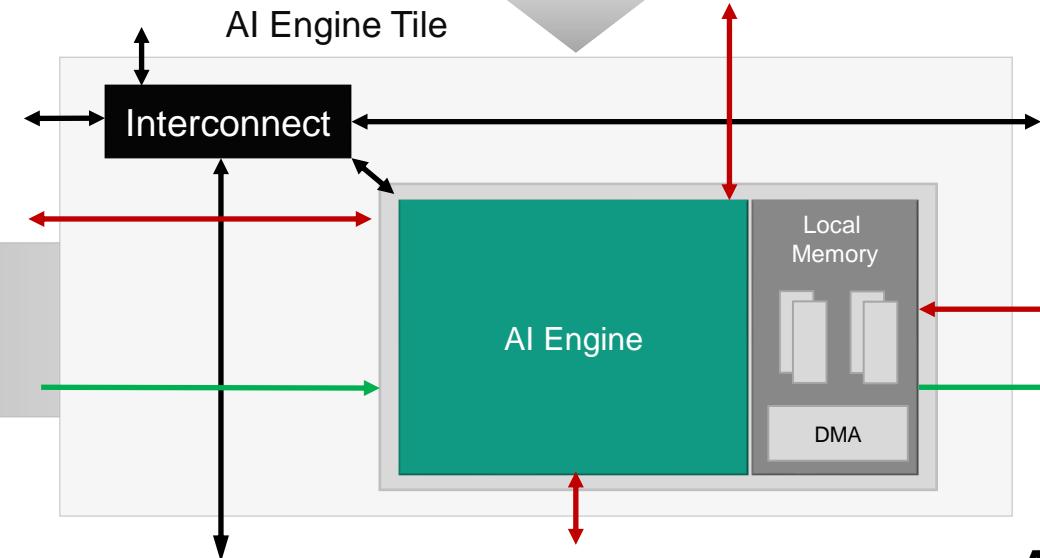
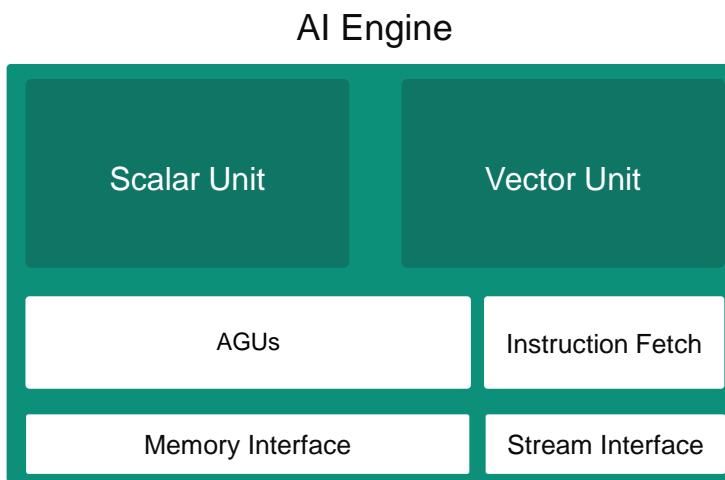
After completing this module, you will be able to:

- Understand the AI Engine terminology
- Understand the multiple levels of parallelism
- Understand the scalar and vector processor
- Understand the AI Engine module interfaces

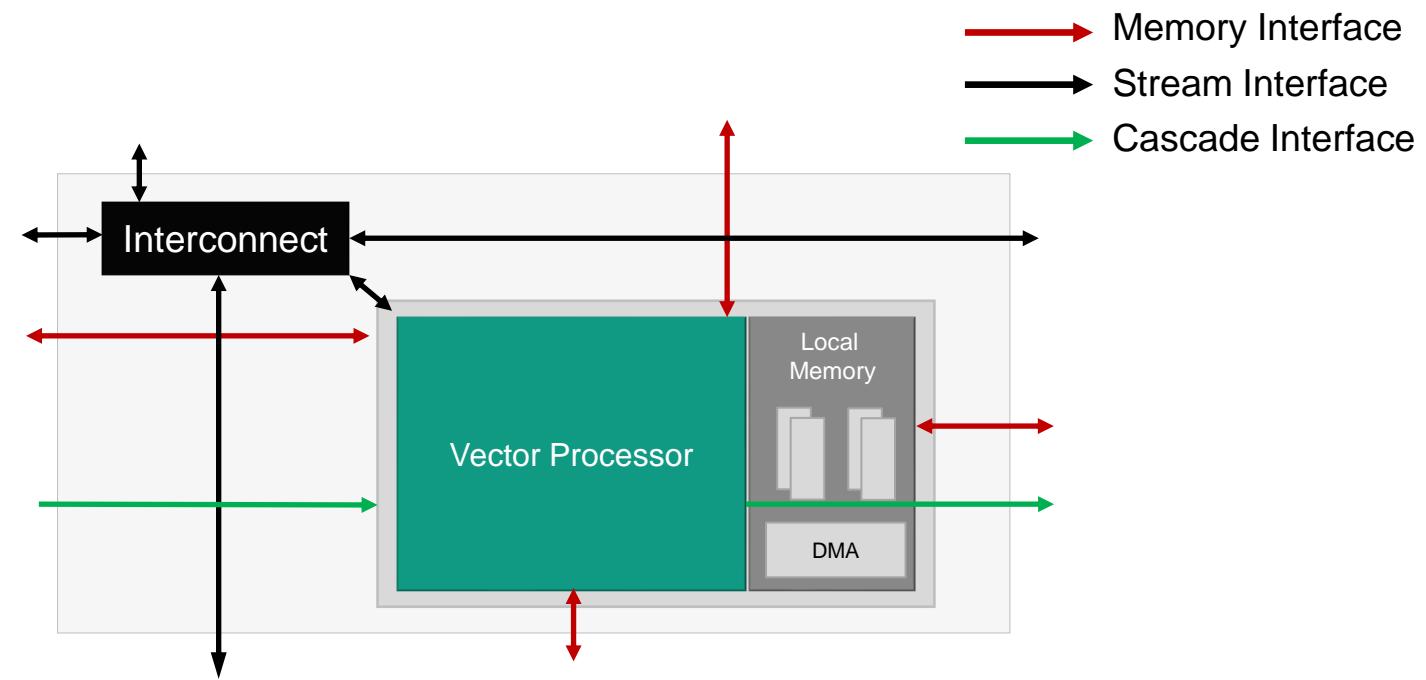
AI Engine: Terminology



—————→ Memory Interface
 —————→ Stream Interface
 —————→ Cascade Interface



AI Engine: Tile-based Architecture



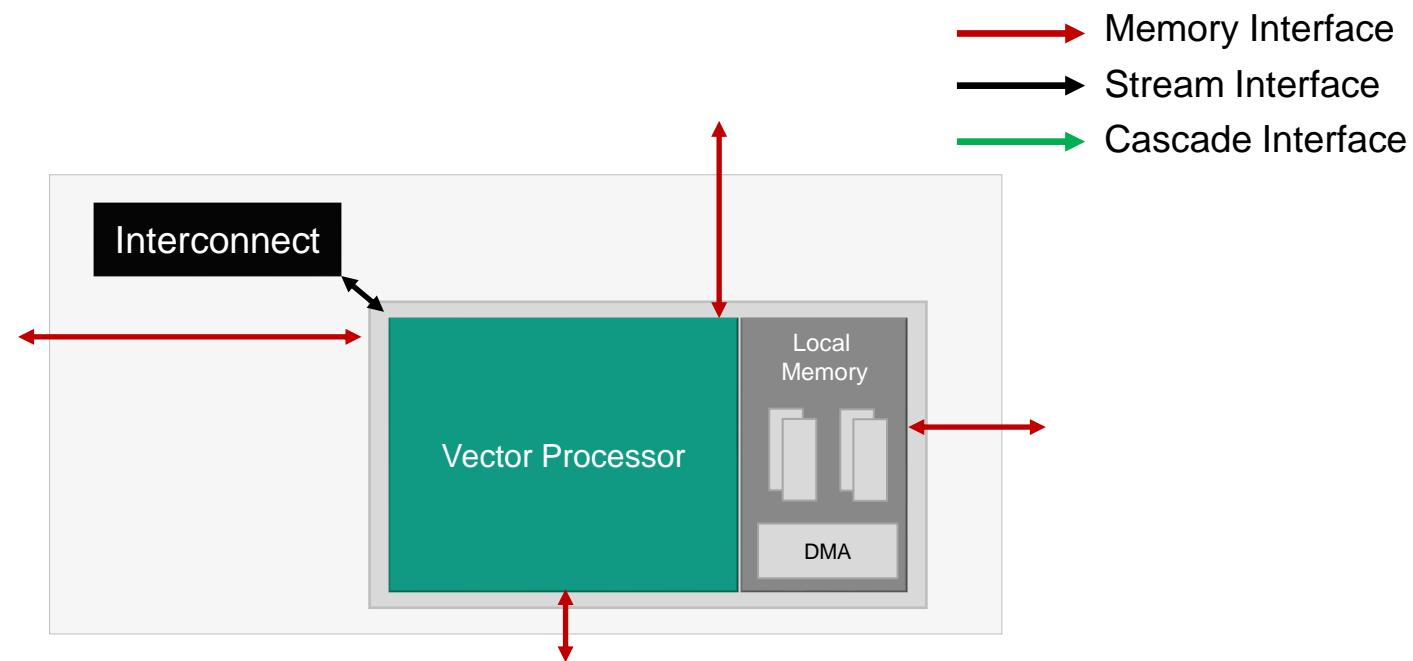
AI Engine: Tile-based Architecture

Local Memory

- Back-pressure handling
- Up to 200+ GB/s bandwidth per tile
- Access to north, south, east or west

DMA

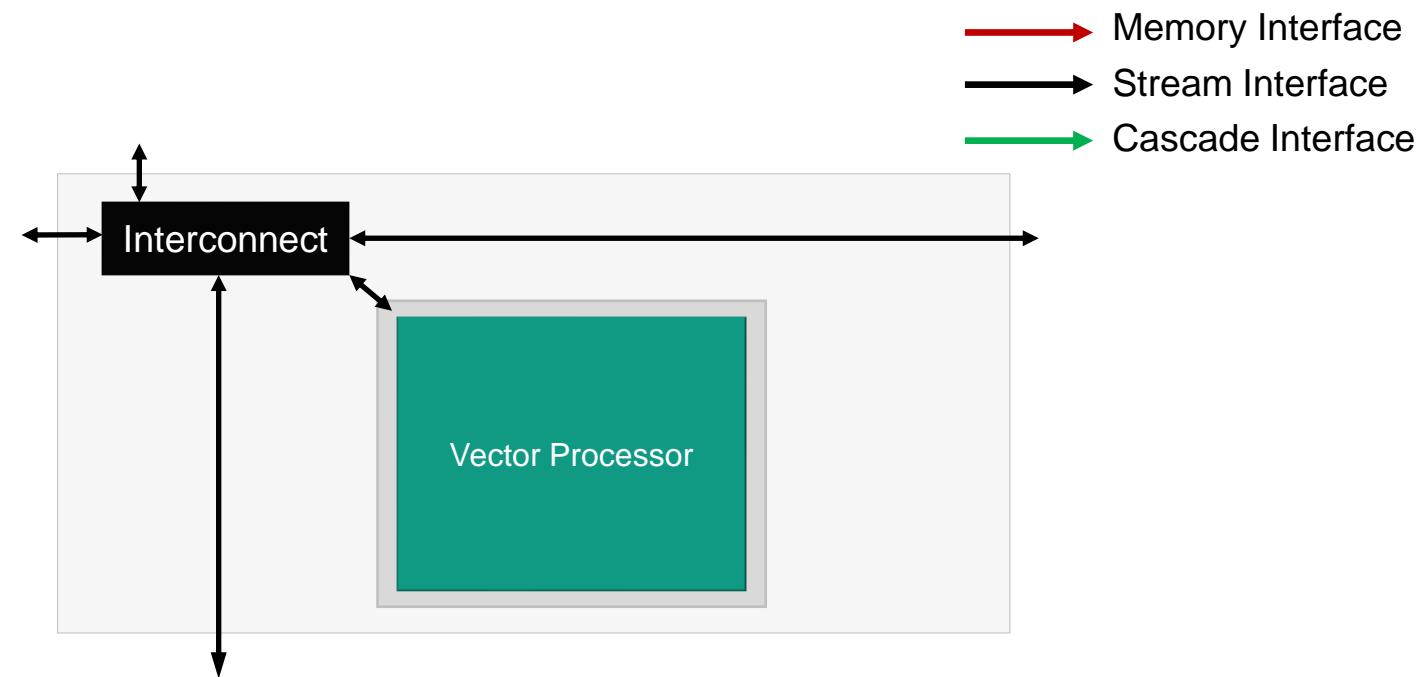
- Non-neighbor data communication
- Via Interconnect
- Integrated synchronization primitives



AI Engine: Tile-based Architecture

Tile Interconnect

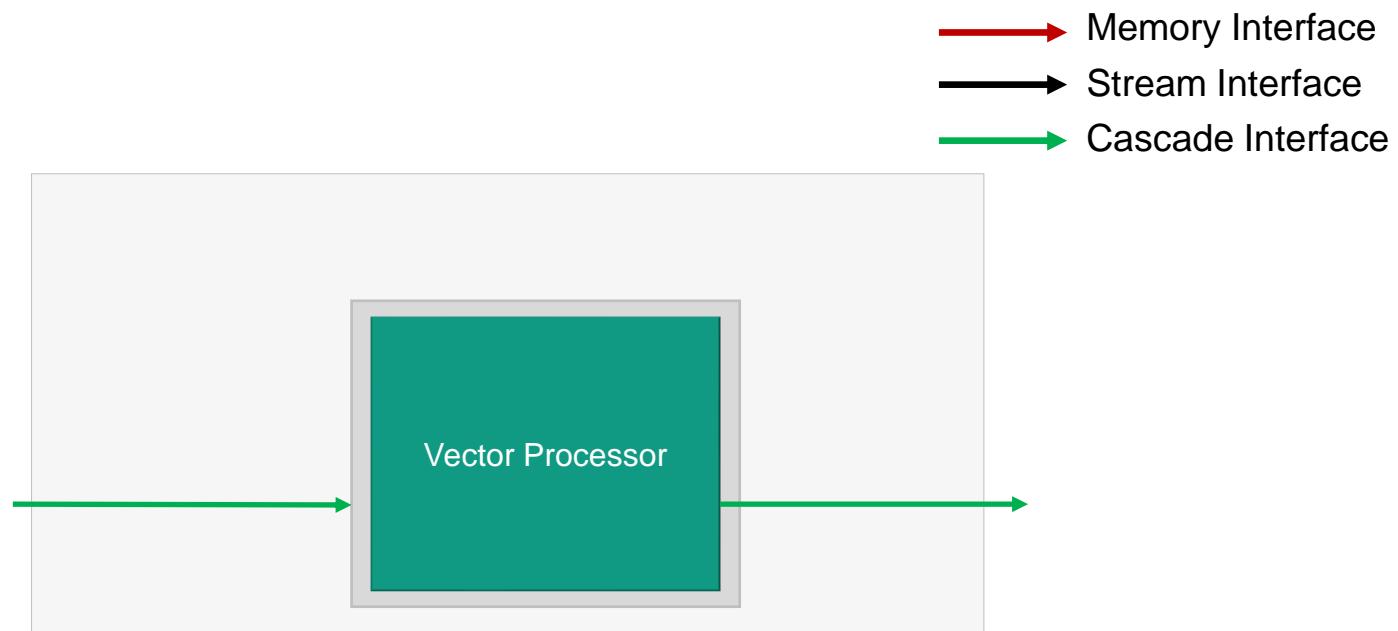
- AXI4 Interconnect
- Back-pressure handling
- Up to 200+ GB/s bandwidth per tile



AI Engine: Tile-based Architecture

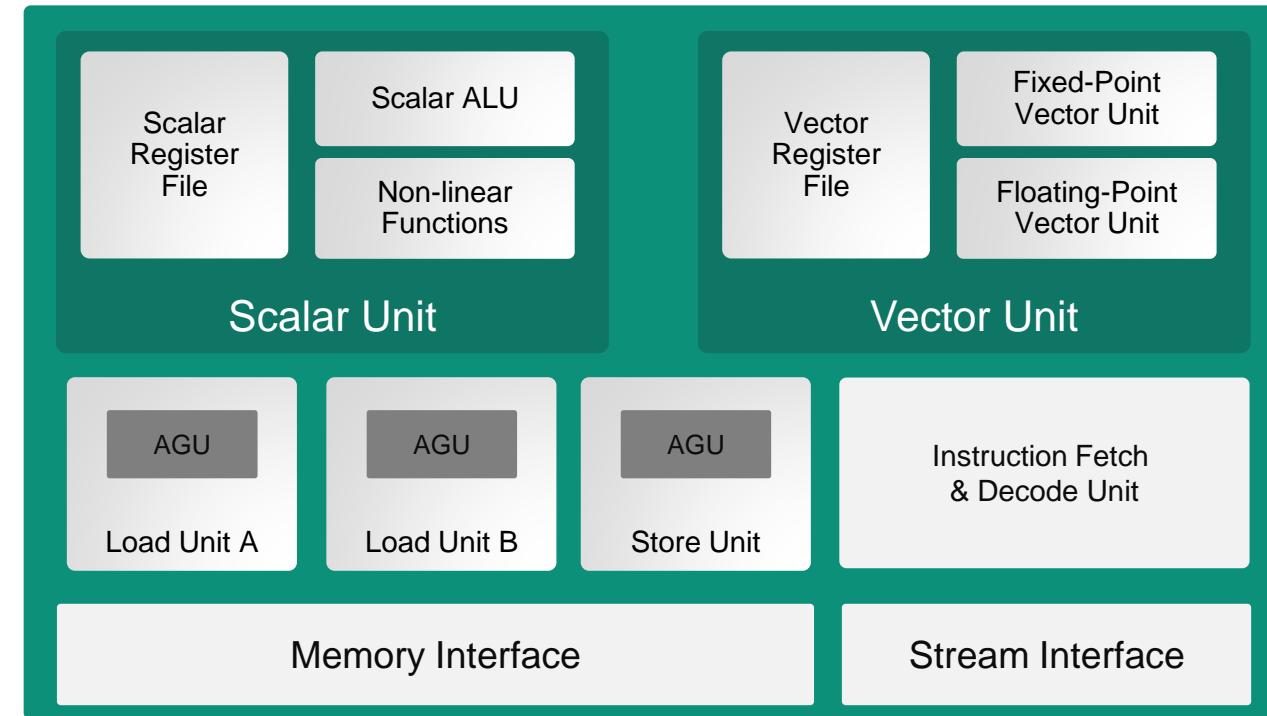
Cascade Interface

- Partial results to next AI Engine
- 384-bit wide
- 2-word deep FIFO



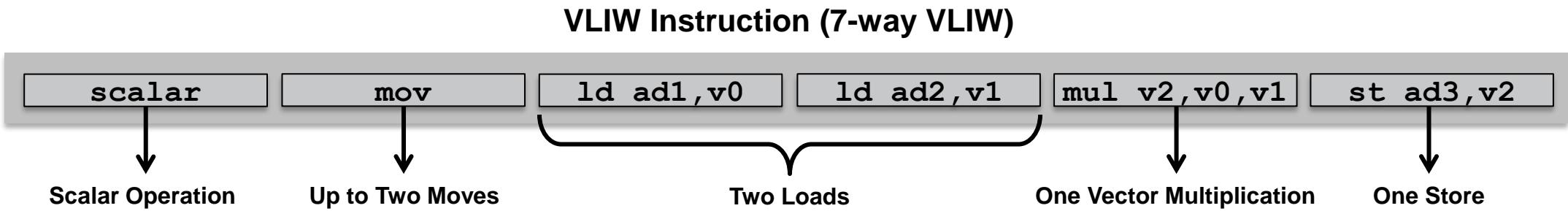
AI Engine

- Is a highly-optimized processor
- Scalar Processor Unit
- Vector Processor Unit
 - Features SIMD and VLIW processor
 - Fixed-point and floating-point precision
- 2x load and 1x store Address Generator Unit
- Instruction Fetch & Decode Unit
- Memory Interface
- Stream Interface



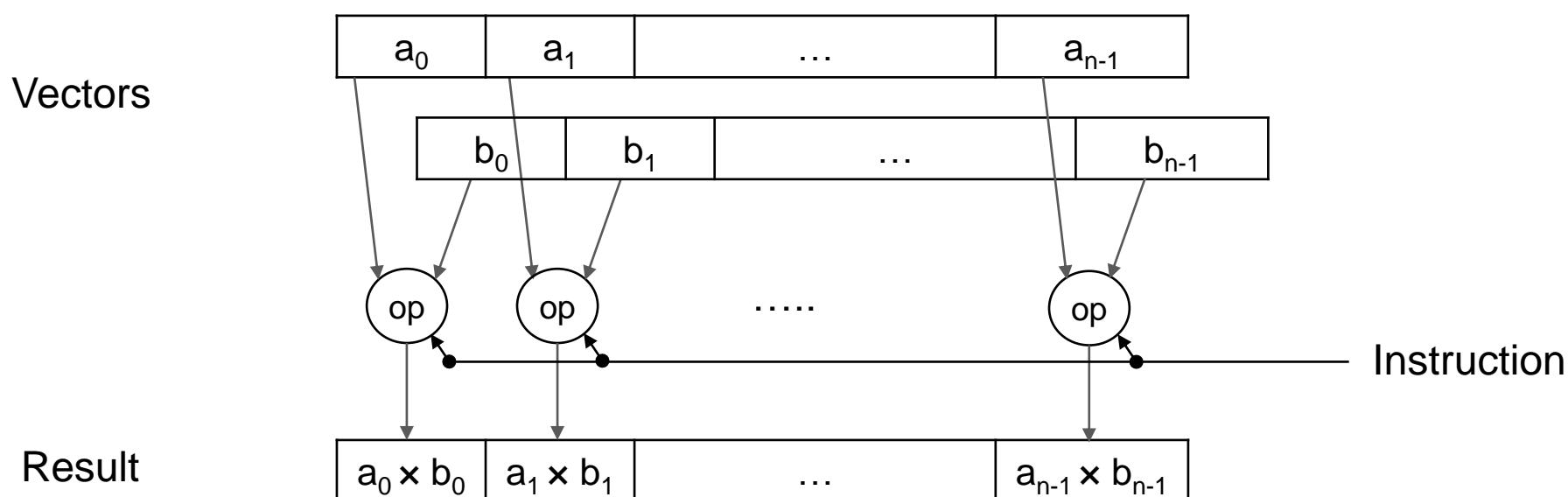
Multiple Levels of Parallelism – ILP

- Instruction-level Parallelism (ILP)
 - Very Long Instruction Word (VLIW)
 - Up to 128-bit wide
 - Up to 7 instructions issued in one cycle
 - 2x loads
 - 1x store
 - Up to 2 move
 - Scalar operation
 - Vector operation



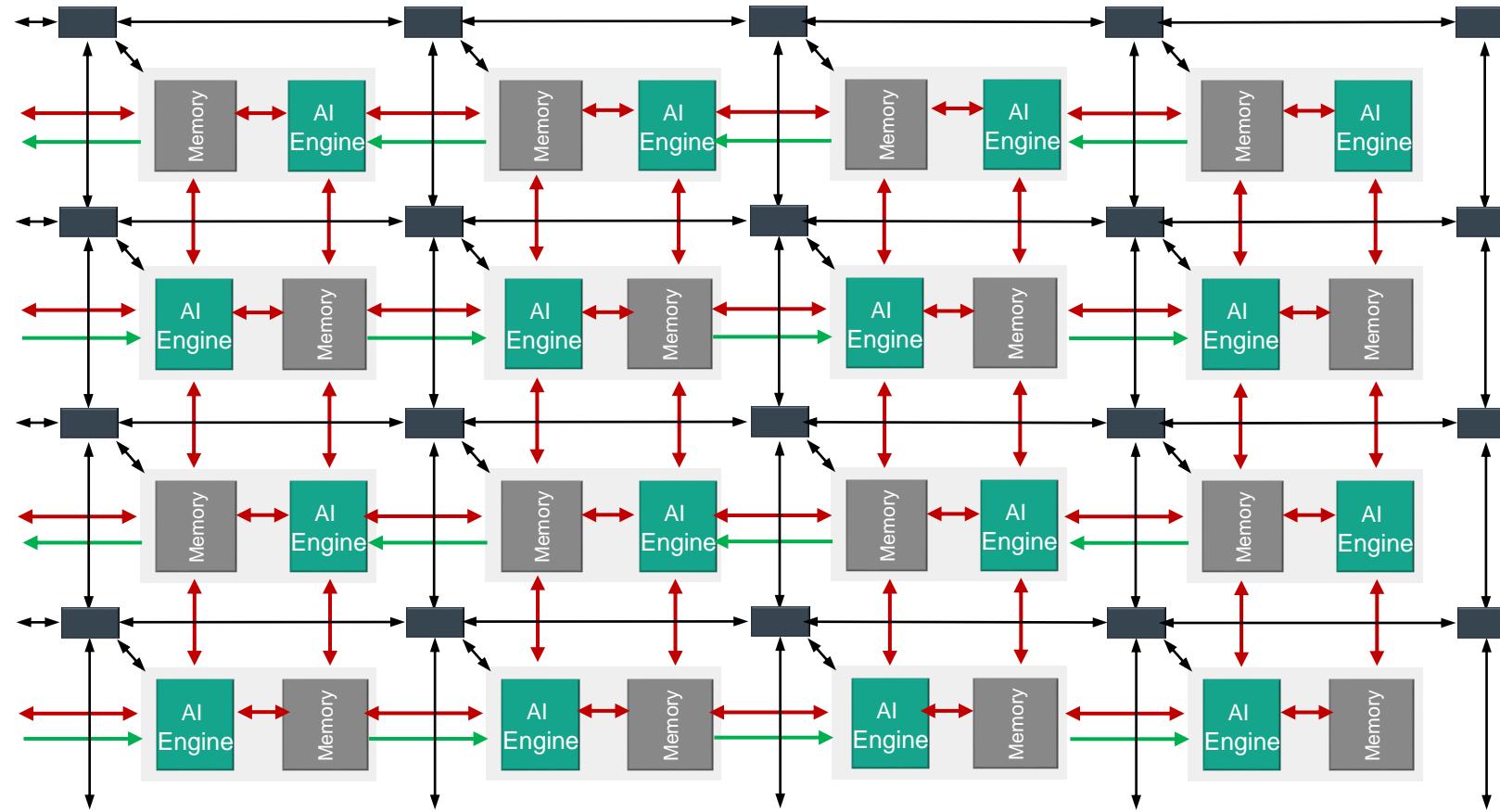
Multiple Levels of Parallelism – DLP

- Data-level Parallelism (DLP)
 - Vector data path
 - Single Instruction Multiple Data (SIMD)
 - Operation applies to each element
 - Support for multiple datatypes



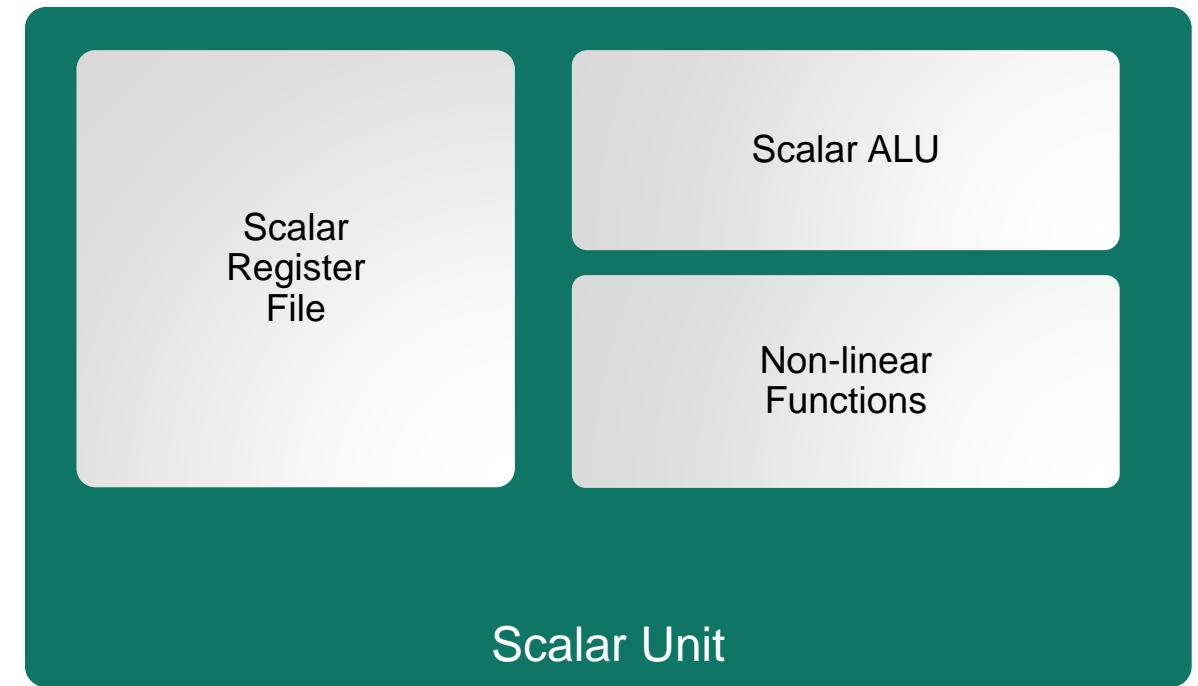
Multiple Levels of Parallelism – Multicore

- Multicore array
 - From 8 up to 400 independent AI Engine tiles



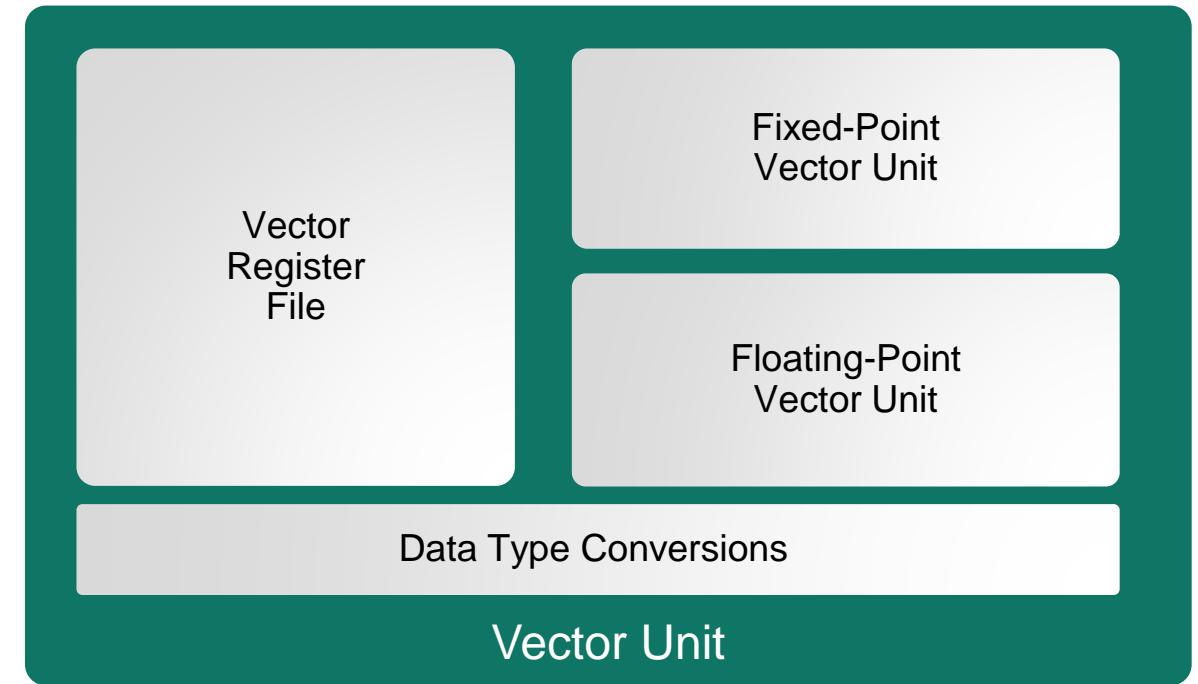
AI Engine – Scalar Unit

- RISC processor
 - Scalar & Special Register File
 - Scalar ALU
 - Non-linear Functions
 - Data type conversion
- Standard C programming

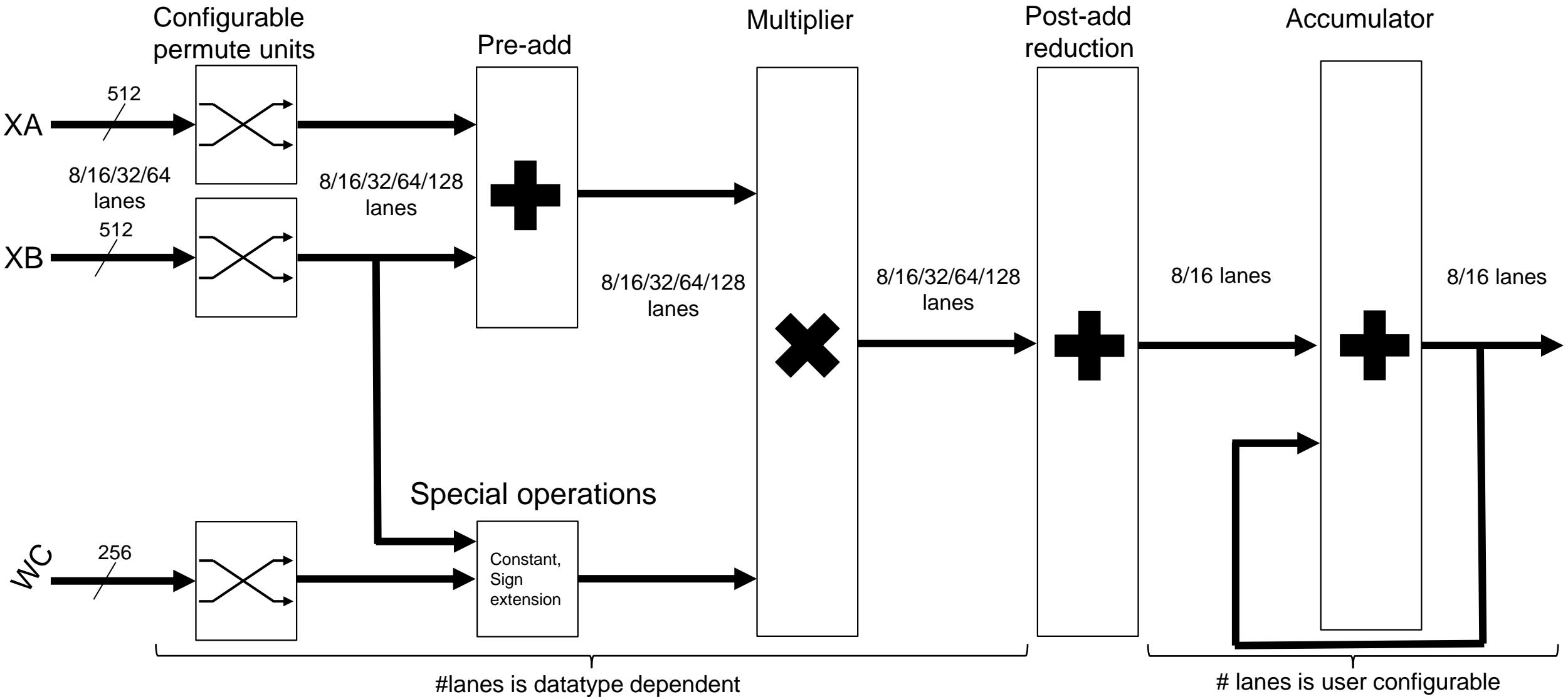


AI Engine – Vector Unit

- Vector and Accumulator Register File
- Fixed-Point Vector Unit
- Floating-point Vector Unit
- Single Instruction Multiple Data (SIMD)
- Multi-precision support
- Special vector data types



Fixed-Point Vector Unit – Multiply Accumulator (MAC) Path



Fixed-Point Vector Unit – Upshift & Shift Round Saturate Paths

Upshift Path

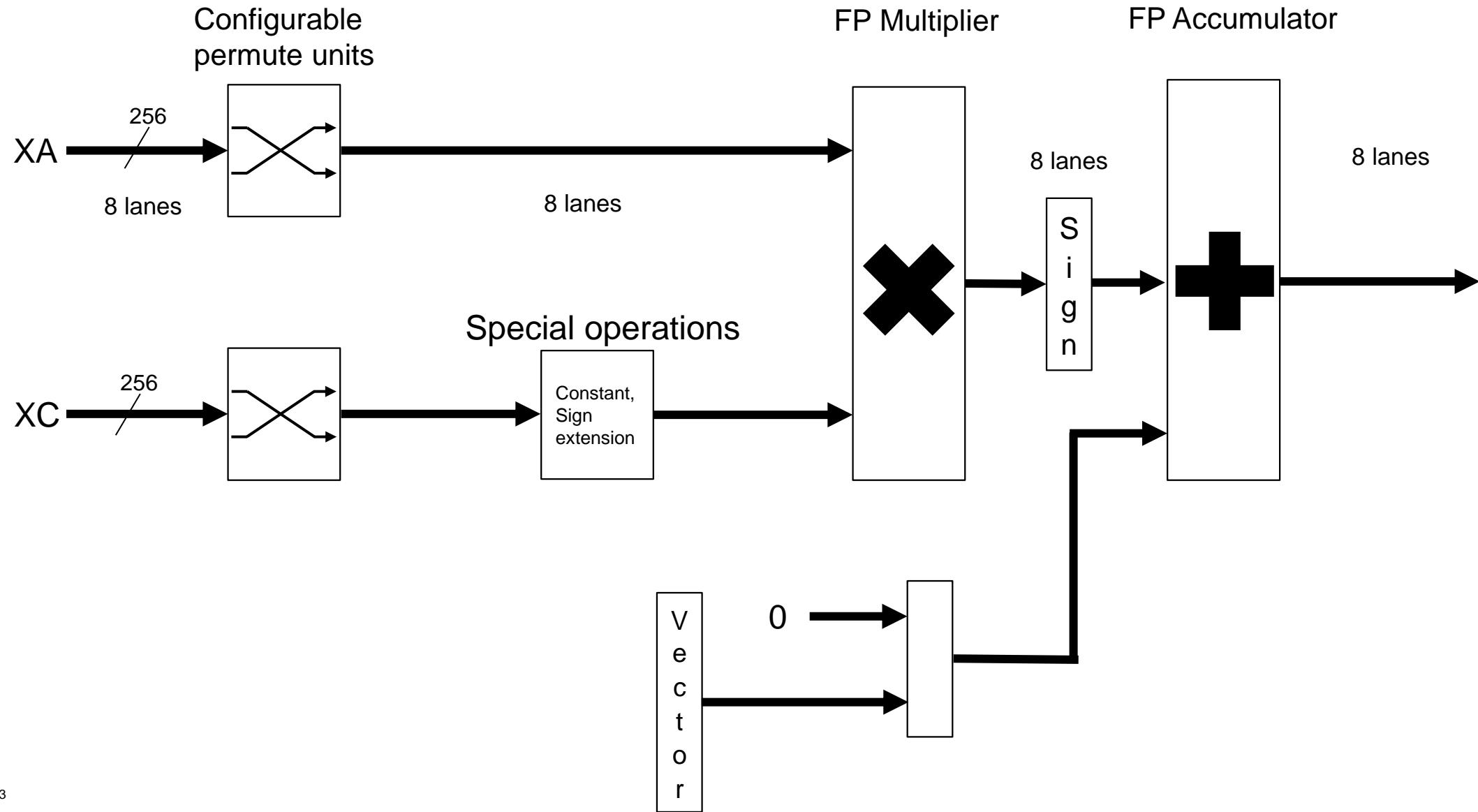
- Transfers vectors to accumulators
 - Lower to higher bit-width precision
- Reads data from:
 - Permute units or
 - Vector register
- Left-shifts
- Feeds it to the accumulator registers

Shift Round Saturate Path (SRS)

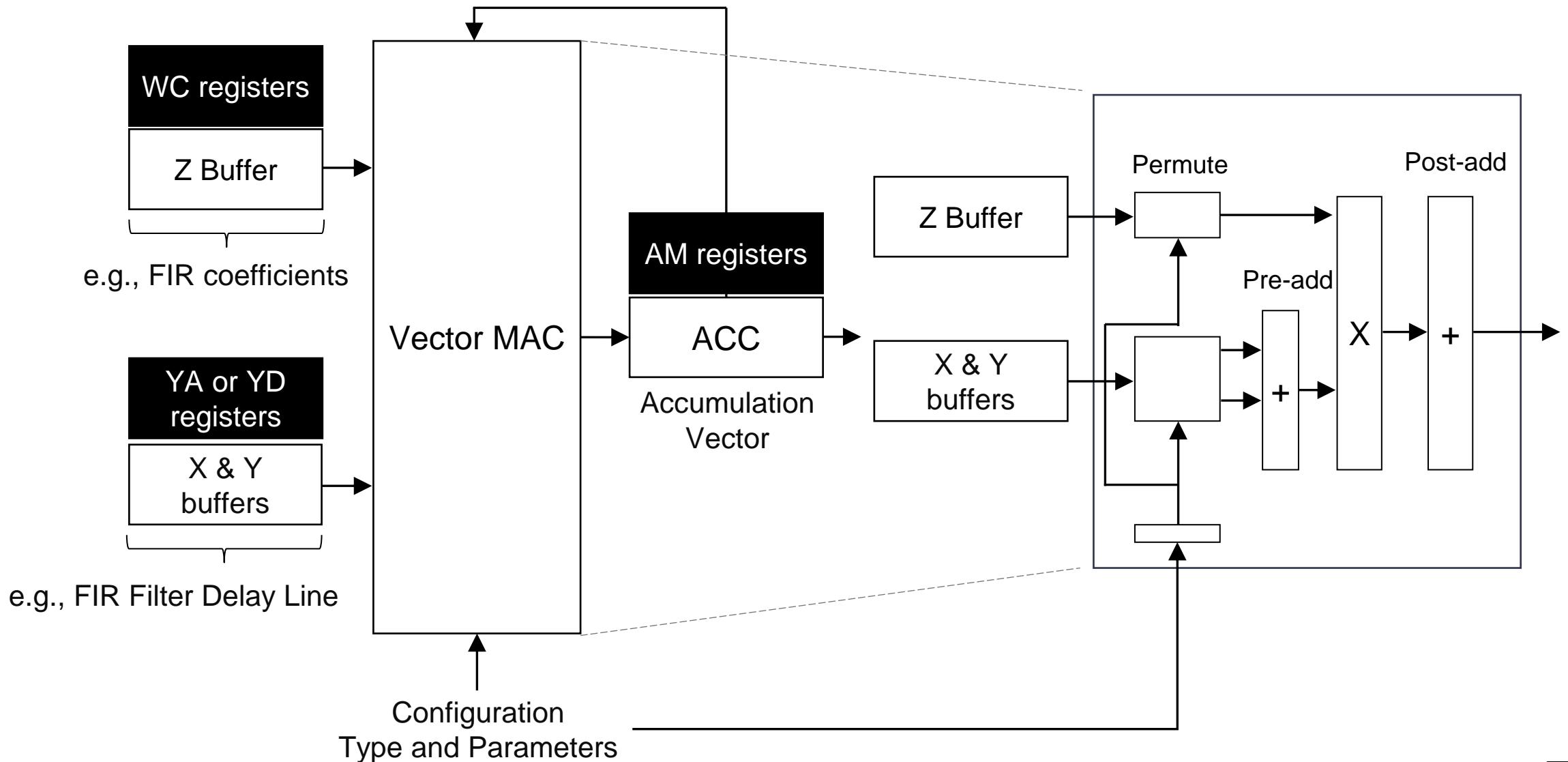
- Transfers accumulators to vectors
 - Higher to lower bit-width precision
- Reads from the accumulator registers and stores
 - To vector registers or
 - To data memory
- Data is right shifted on a lane-by-lane basis
- Control register S determines the shift amount
- Saturation and rounding control register
 - Q and R fields to influence its behavior
- Support for various rounding modes
 - PosInf, NegInf, SymInf
 - SymZero, ConvEven, and ConvOdd

Check out [AM009](#) for more information

Floating-Point Vector Unit – Multiply Accumulator (MAC) Path



Multiply Accumulate (MAC) operations



Vector and Accumulator Register File

128-bit	256-bit	512-bit	1024-bit	
vrl0				
vrh0	wr0			
vrl1		xa		N/A
vrh1	wr1			
vrl2			ya	
vrh2	wr2			
vrl3		xb		
vrh3	wr3			
vcl0				
vch0	wc0			
vcl1		xc	N/A	N/A
vch1	wc1			
vdl0				
vdh0	wd0			
vdl1		xd	N/A	yd(LSBs)
vdh1	wd1			

Vector Registers

384-bit	768-bit
aml0	bm0
amh0	
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

Accumulator Registers

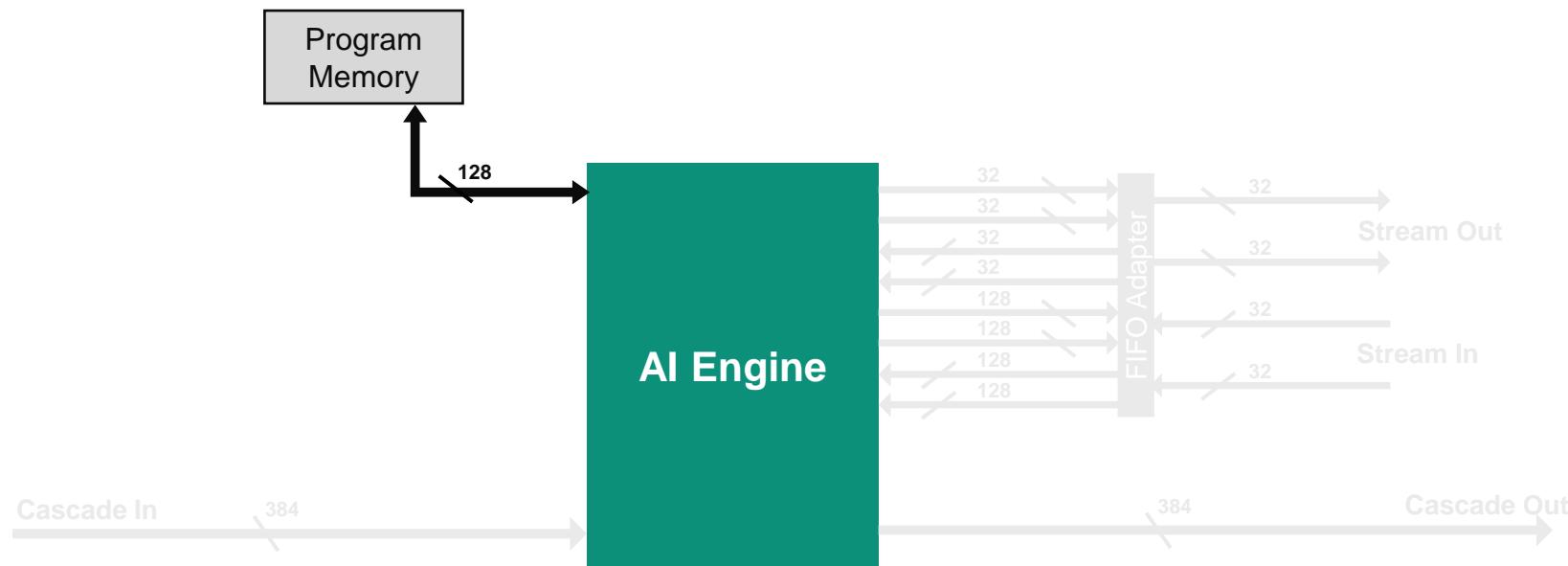
Multi-precision Support

X Operand (1024-bit)	Z Operand (256-bit)	Output	Number of GMACs @ 1 GHz
8 real	8 real	48 real	128
16 real	8 real	48 real	64
16 real	16 real	48 real	32
16 real	16 complex	48 complex	16
16 complex	16 real	48 complex	16
16 complex	16 complex	48 complex	8
16 real	32 real	48/80 real	16
16 real	32 complex	48/80 complex	8
16 complex	32 real	48/80 complex	8
16 complex	32 complex	48/80 complex	4
32 real	16 real	48/80 real	16
32 real	16 complex	48/80 complex	8
32 complex	16 real	48/80 complex	8
32 complex	16 complex	48/80 complex	4
32 real	32 real	80 real	8
32 real	32 complex	80 complex	4
32 complex	32 real	80 complex	4
32 complex	32 complex	80 complex	2
32 SPFP	32 SPFP	32 SPFP	8

There are 128 x 8-bit single multipliers (post-added and accumulated into 16 or 8 accumulator lanes)

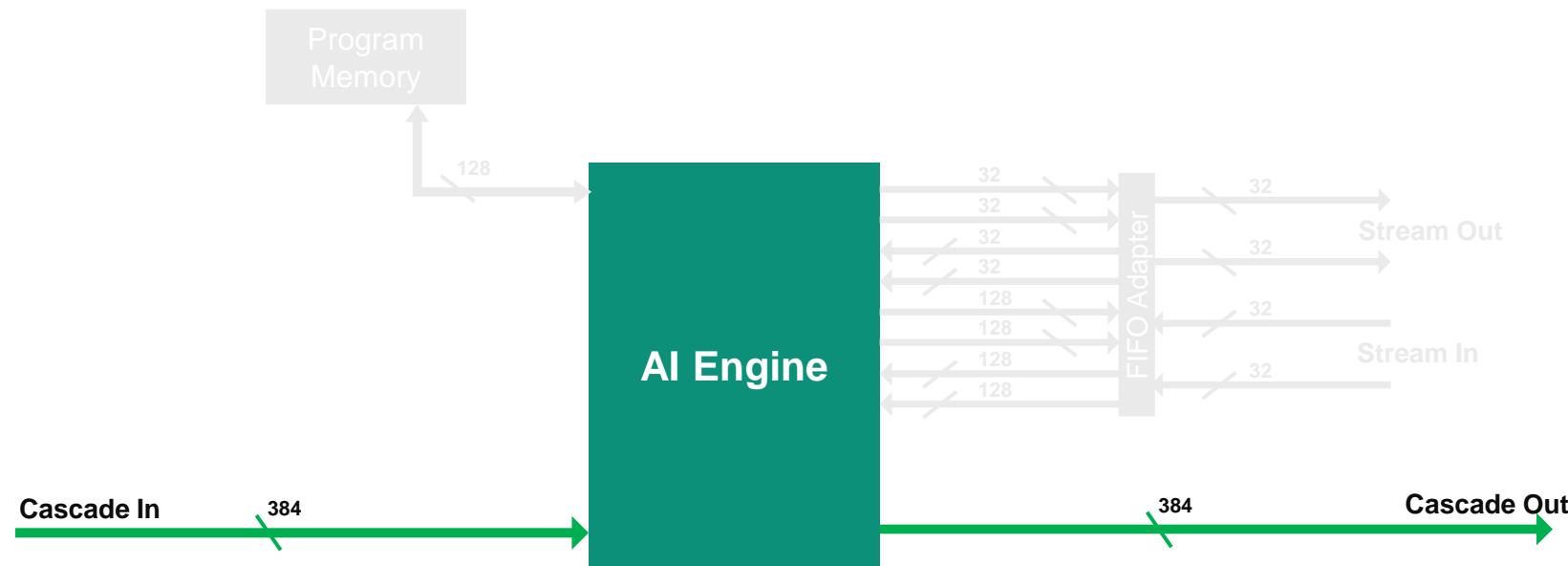
AI Engine: Interfaces – Program Memory Interface

- 128-bit wide
- New instruction per cycle
- 16KB program memory size
 - Compress instructions
- Instruction width of 32, 64, 96, 128 bits



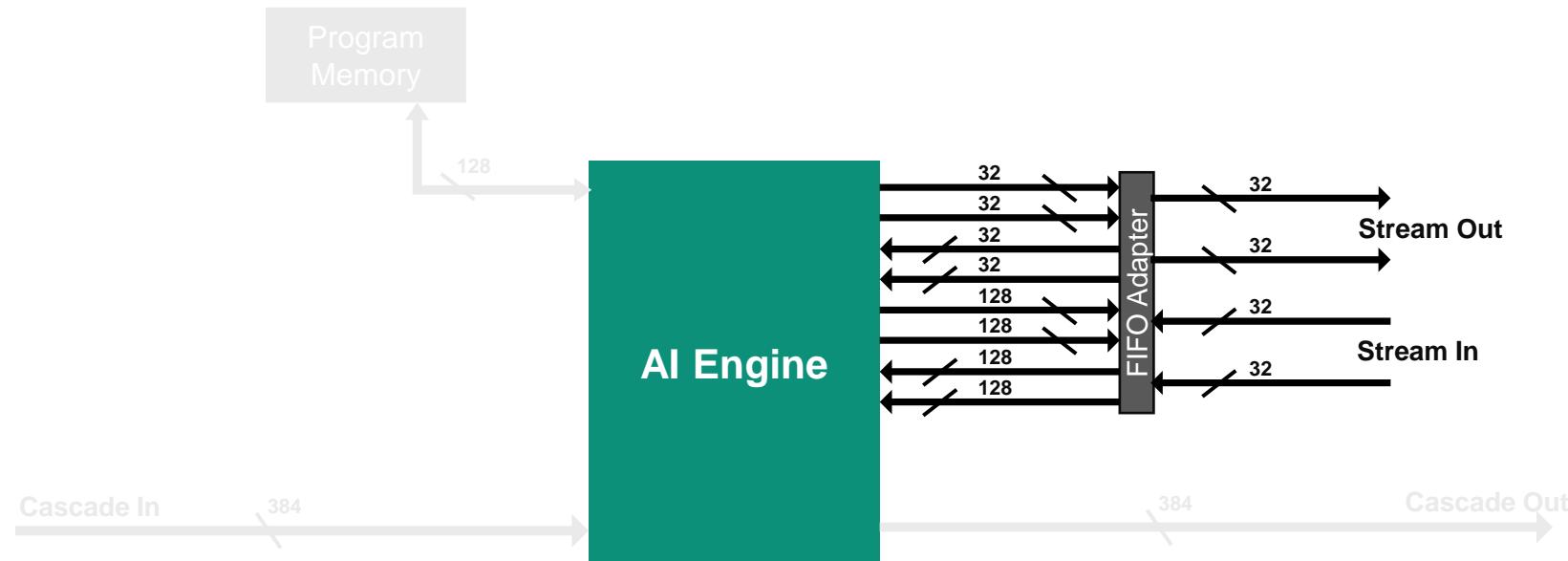
AI Engine: Interfaces – Cascade Stream

- Forwards 384-bit accumulator data from one AI Engine (input) to a neighbor AI Engine (output)
- Two-deep, 384-bit FIFO on I/O streams



AI Engine: Interfaces – Stream Interface

- Direct AXI-Stream
- Two incoming 32-bit data streams
- FIFO of four 32-bit words deep
- Two outgoing 32-bit data stream
- FIFO of four 32-bit words deep



AI Engine: Other Interfaces

- Data Memory Interface
- Locks
- Core Debug
- Stall handling
- Event
- Tile Timer
- Execution Trace

Check out [AM009](#) for more information

Summary

- The AI Engine consists of:
 - 512-bit SIMD (single instruction, multiple data) vector units, both fixed-point and floating-point vector units
 - 16 KB program memory
 - 32-bit scalar RISC processor
 - 256-bit load (x2) and store units with individual address generation units (AGUs)
- The AI Engine supports multiple levels of parallelism
 - Instruction-level parallelism (ILP)
 - Data-level parallelism (DLP)
 - Many core



AI Engine Memory and Data Movement

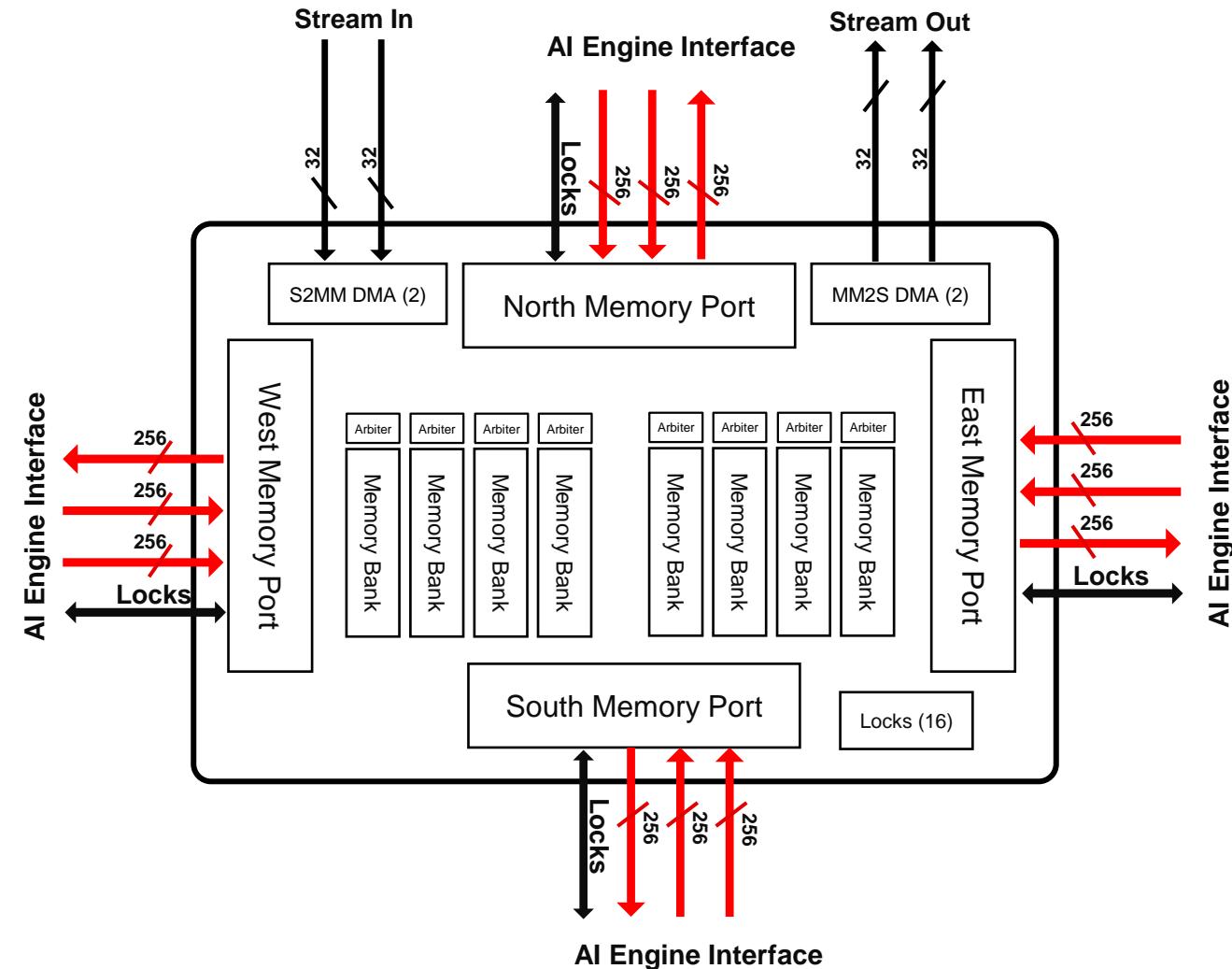
Objectives

After completing this module, you will be able to:

- Describe the memory module architecture for the Versal® AI Engine
- Describe the memory access structure for the AI Engine
- Describe the data movement between:
 - AI Engine to AI Engine
 - AI Engine to AI Engine via memory and DMA
 - AI Engine to AI Engine via AXI4-Stream Interconnect
 - AI Engine to programmable logic (PL)

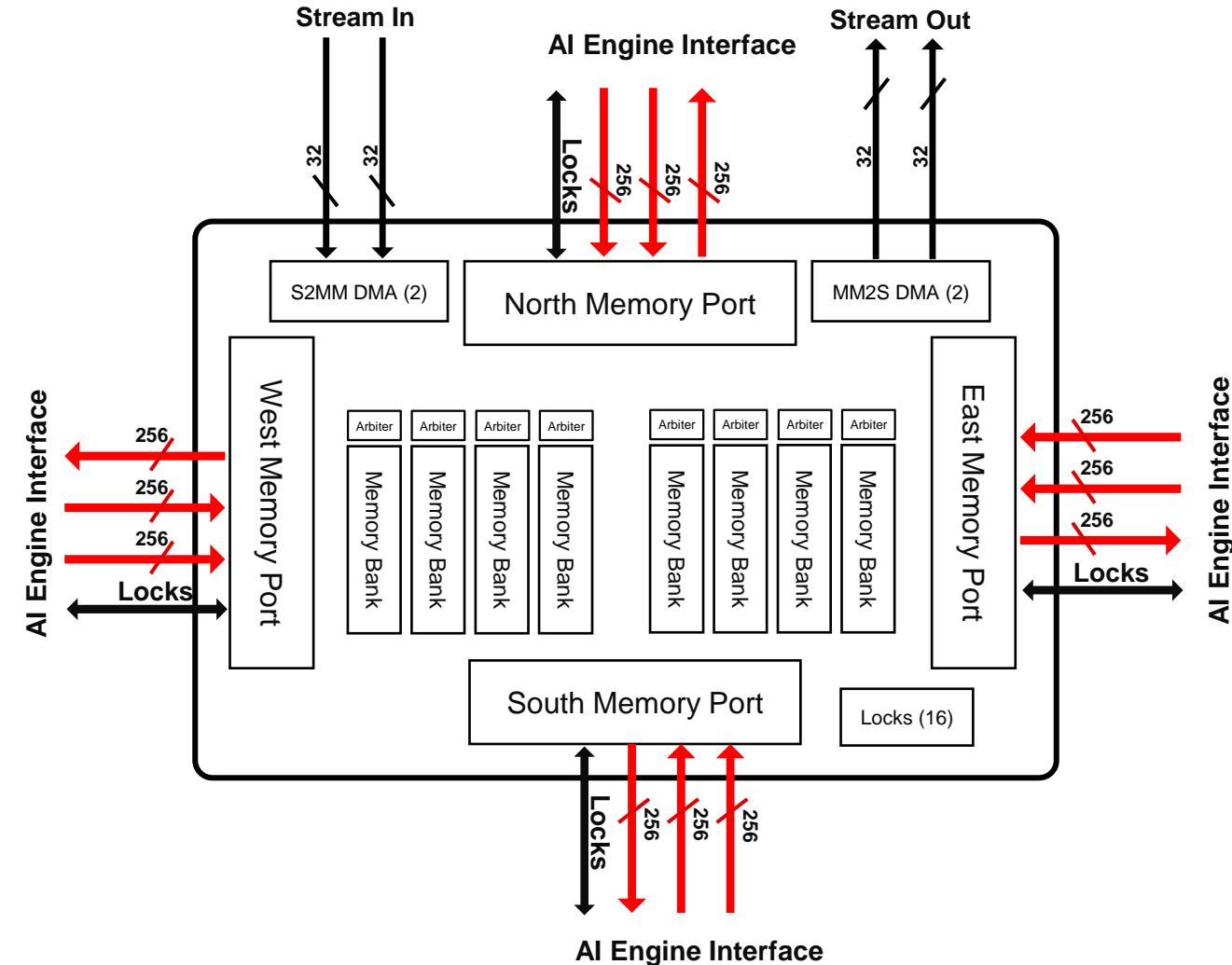
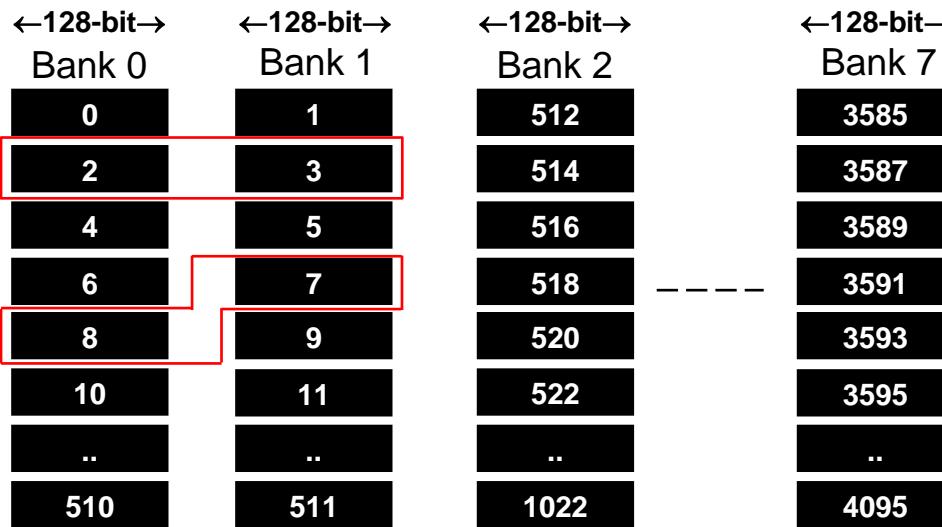
AI Engine Memory Module

- 32 KB of local memory
- Eight memory banks:
 - 4 KB each
 - 256-word x 128-bit single-port memory
 - Write enable for each 32-bit word
 - Banks [0-1]: ECC protection
 - Banks [2-7]: Parity check
 - ECC protection is a 1-bit error detector/corrector and 2-bit error detector per 32-bit word
- Local, distributed memory
 - No cache misses
 - Higher bandwidth



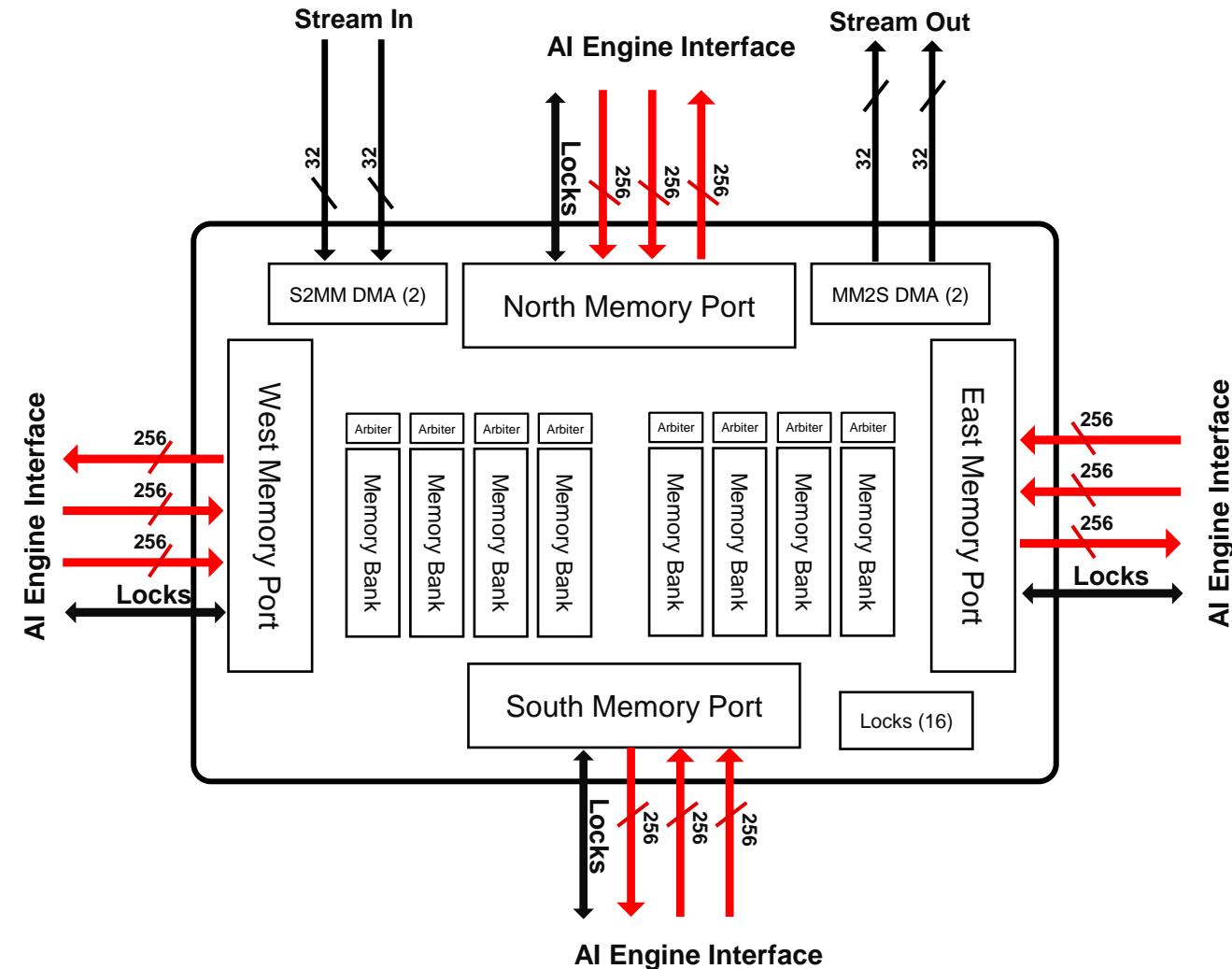
AI Engine Memory Module – Memory Banks

- 8 Memory banks
- Banks bundled in pairs even-odd
 - 256-bit read/write
- Continuous addressing on the bank pair
- LSB and MSB can be in either bank



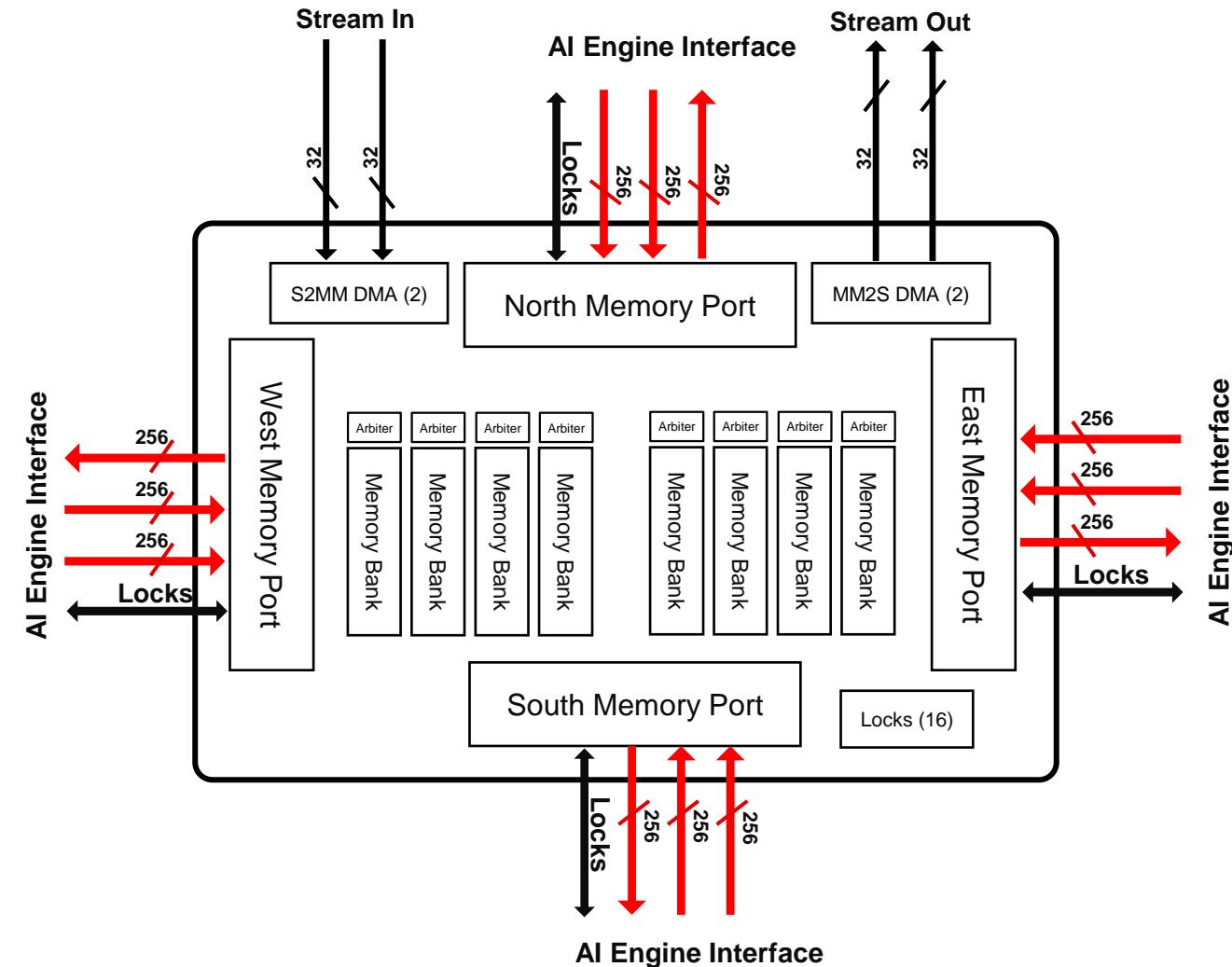
AI Engine Memory Module – Memory Arbitration

- Round-robin to avoid starving any requester
- Handles a new request every cycle
- Only one requestor at a time allowed to access the memory per bank



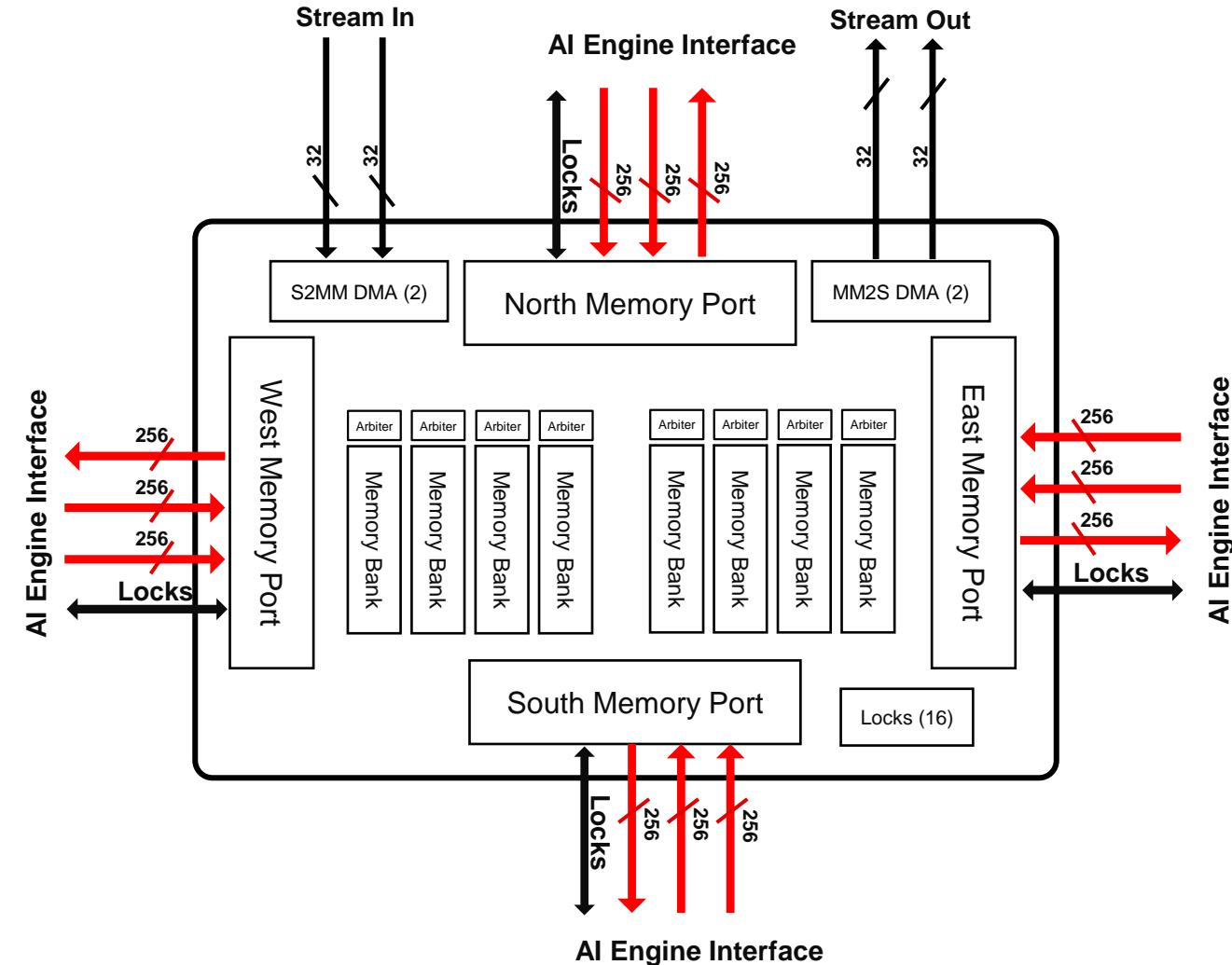
AI Engine Memory Module - DMA Controller

- Two incoming and two outgoing streams to the stream switches
- Divided into two modules
 - S2MM → store 32-bit streams of data to memory
 - MM2S → write the contents of the memory to a 32-bit stream
- Waits four clock cycles (128-bit) to access memory
- Has access to the 16 buffer descriptors
- Has access to the 16 locks
- Non neighboring memory access



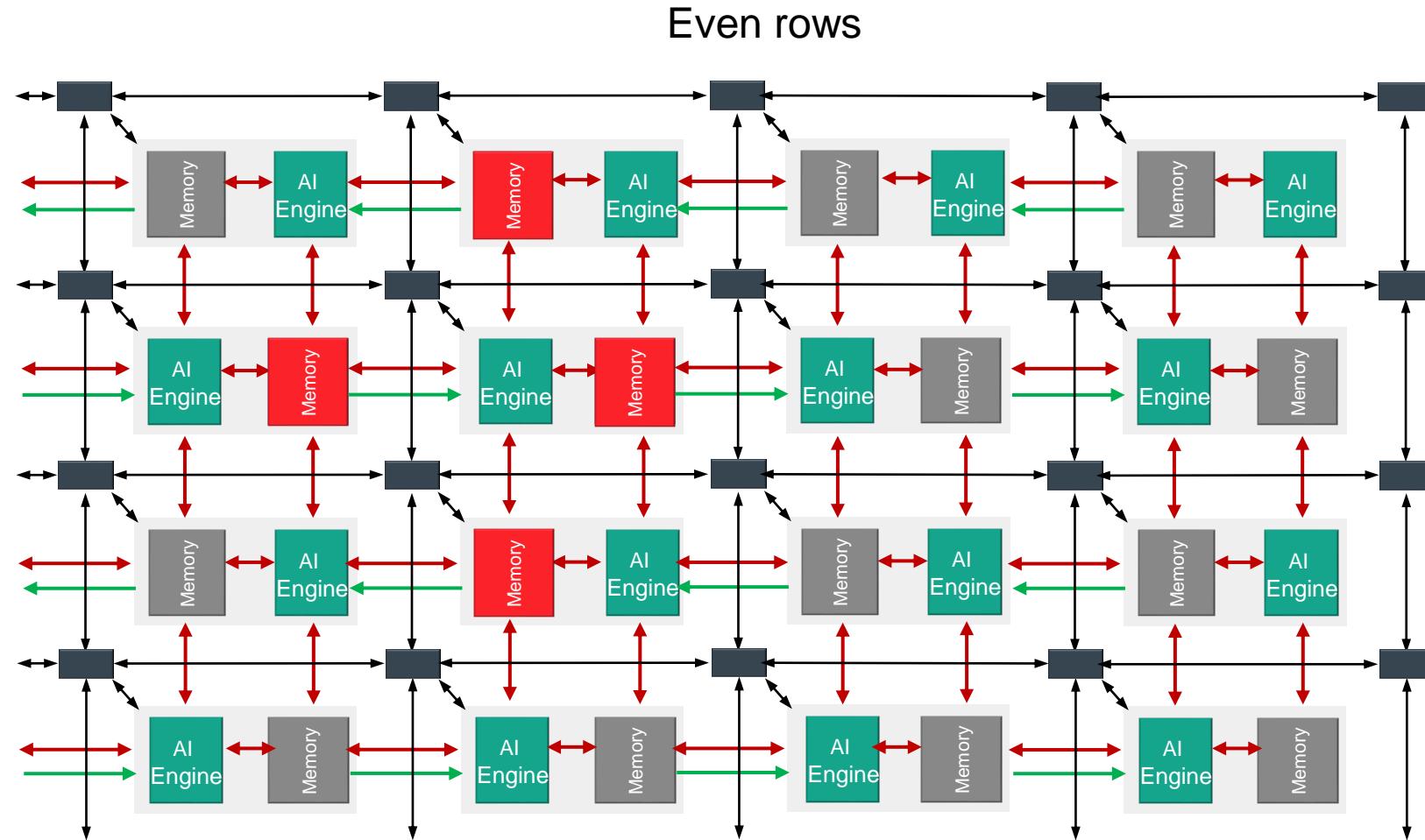
AI Engine Memory Module - Locks

- Helps achieve synchronization among the AI Engines, tile DMA, and external memory-mapped AXI4 interface master
- Sixteen hardware locks with a binary value for each AI Engine memory module lock
- One arbitrator for each lock
- Handles lock requests from
 - AI Engines
 - Local DMA controller
 - Memory-mapped AXI4



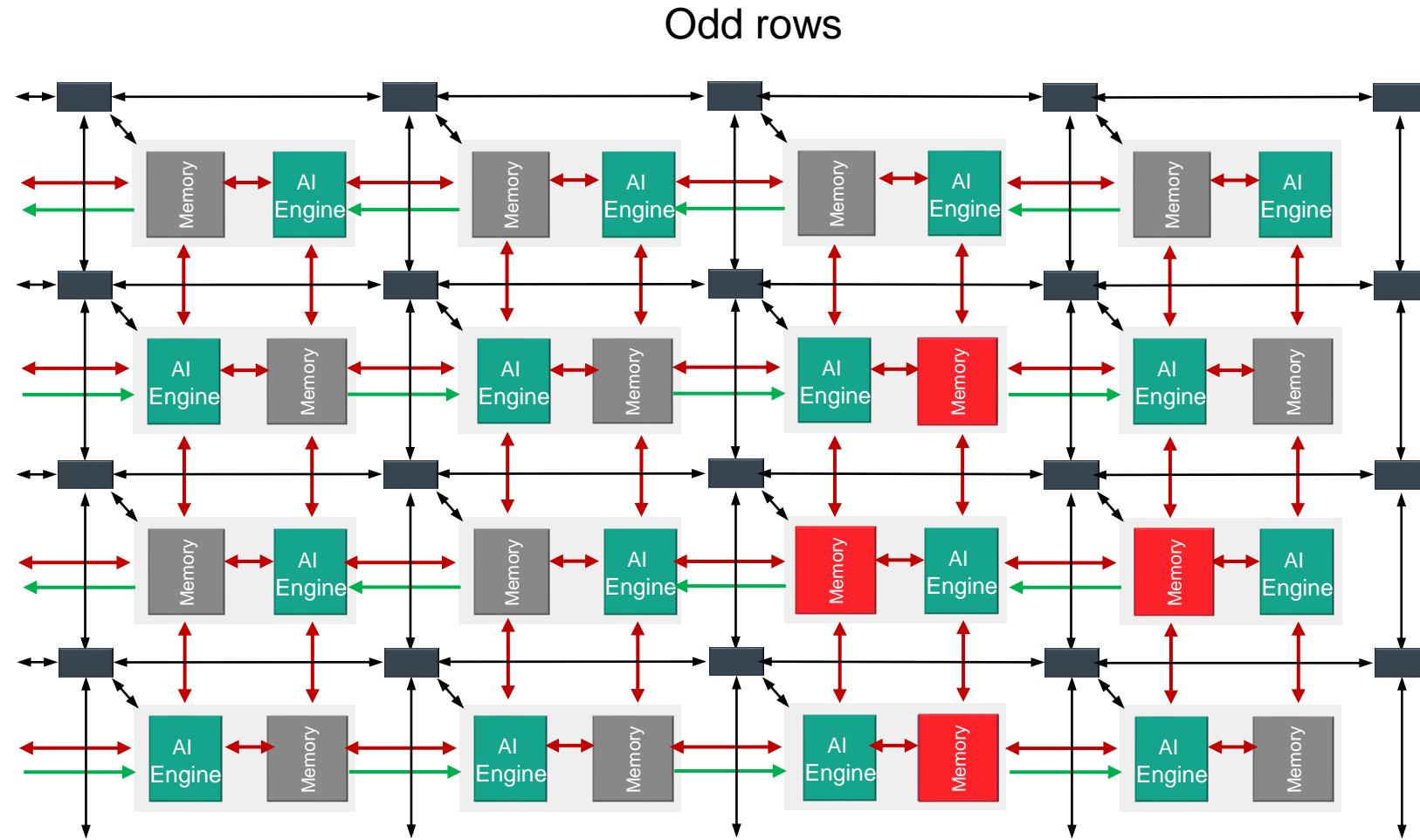
Memory Accesses in the AI Engine Array

- Each AI Engine can access four memory modules
 - Local
 - North
 - South
 - West or East, row dependant
- Diagonals do not have the same memory access structure



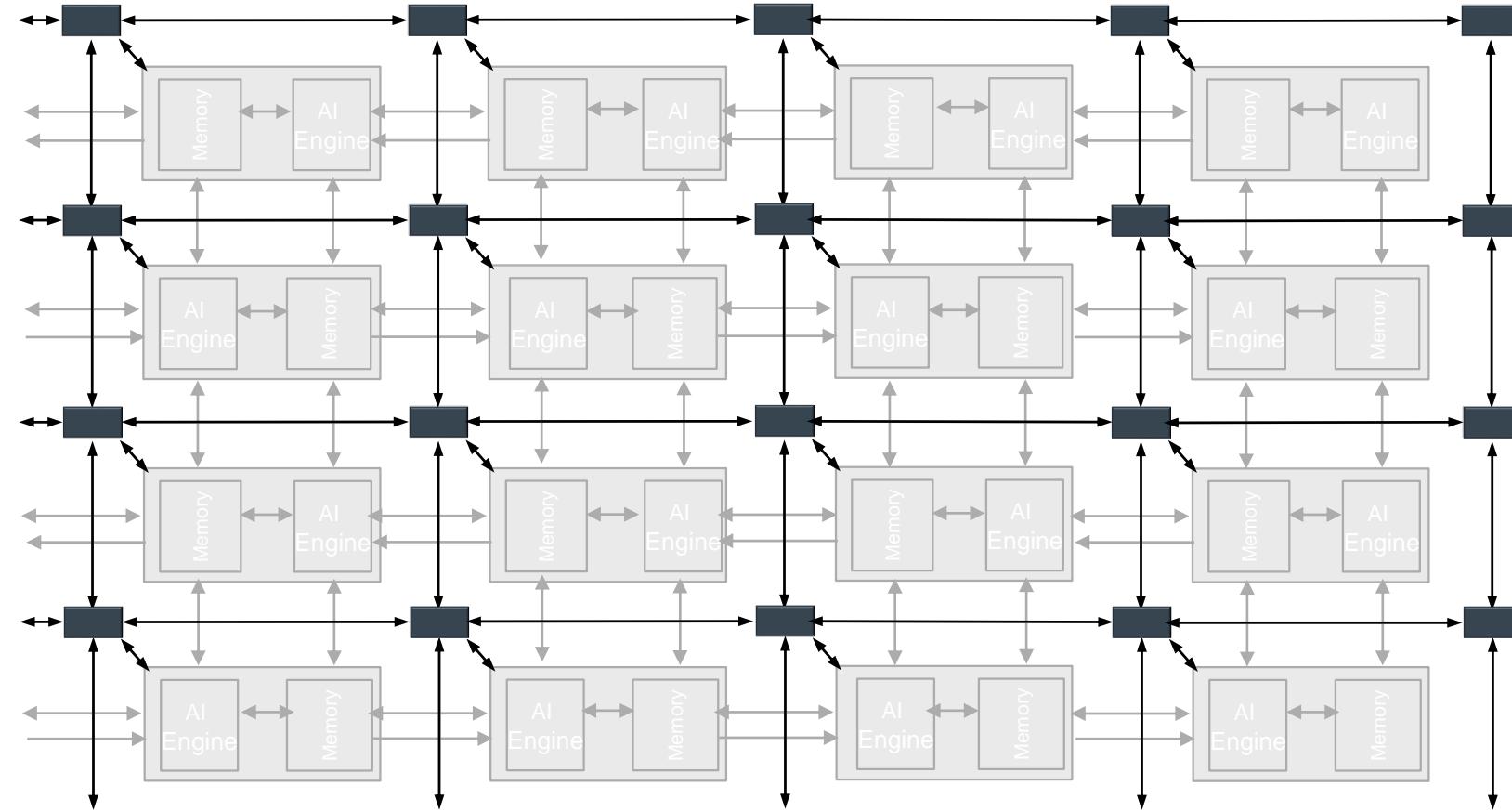
Memory Accesses in the AI Engine Array

- Each AI Engine can access four memory modules
 - Local
 - North
 - South
 - West or East, row dependant
- Diagonals do not have the same memory access structure



AI Engine: AXI-Stream Interconnect

- Data movement between non-neighboring tiles
- Dedicated interconnect
 - Back-pressure handling
 - Deterministic



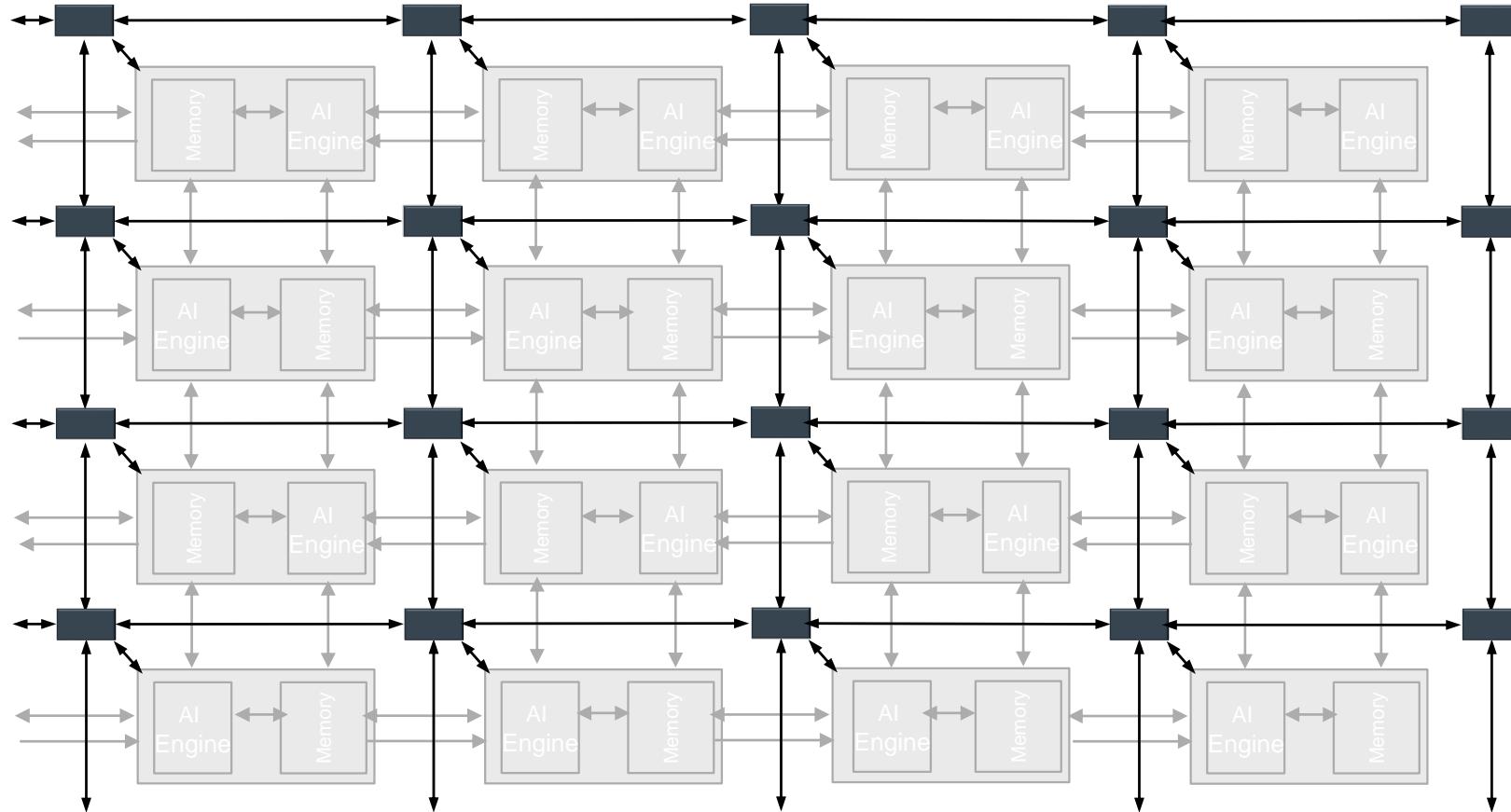
AI Engine: AXI-Stream Interconnect

Circuit Switching

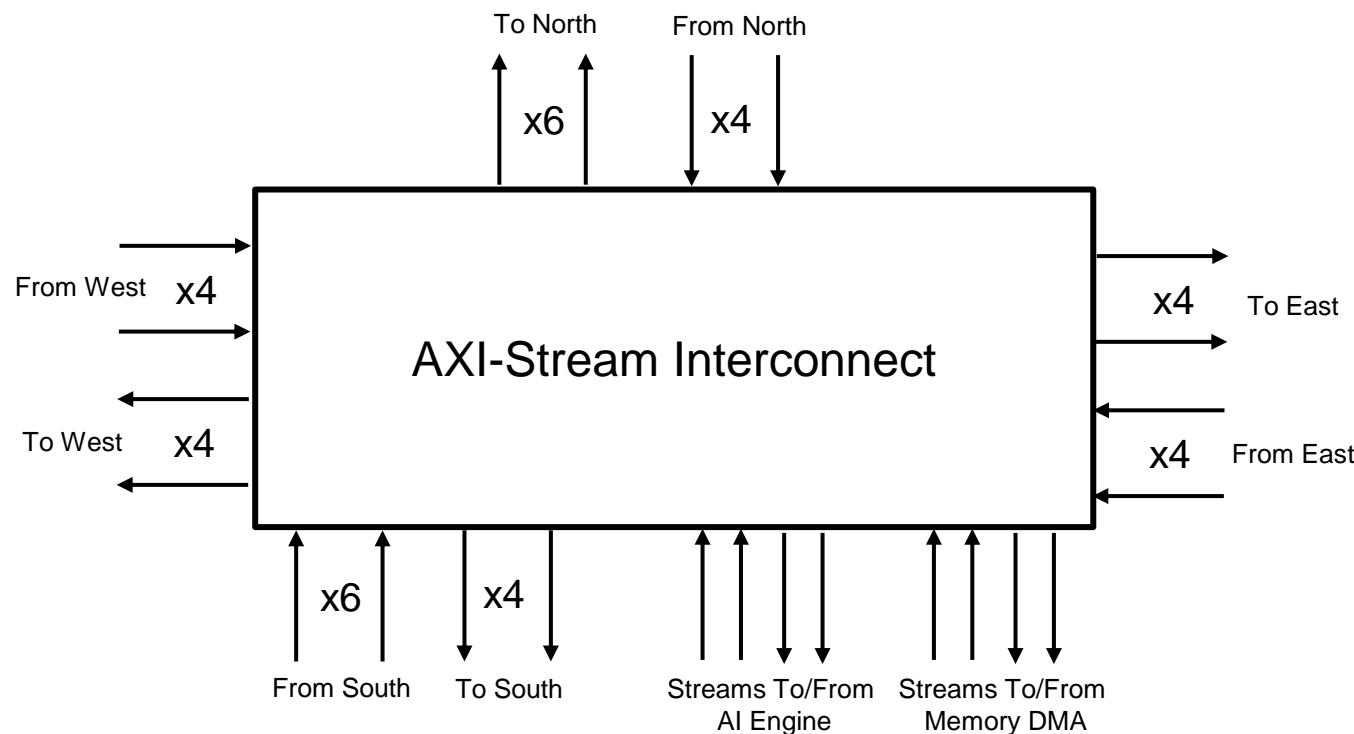
- Point to point within the AI Engine array or beyond
- Deterministic

Packet Switching

- Time sharing, interleaved communication
- Not deterministic
- Examples:
 - Packetized data
 - Header words



AI Engine: AXI-Stream Interconnect

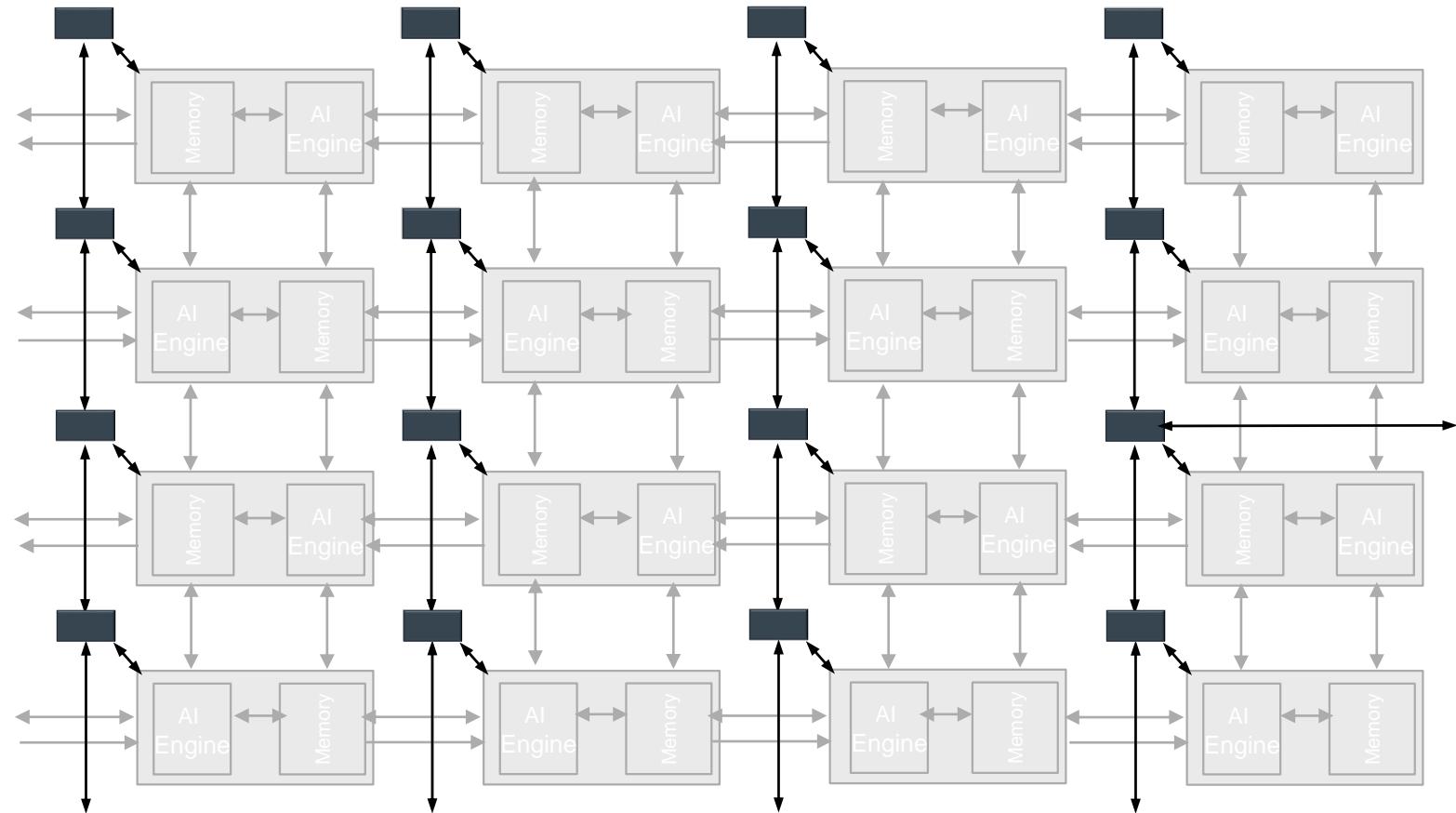


Direction	Streams	Bit-width	Bandwidth per stream	Aggregated Bandwidth
To North	6			24 GB/s
To South				
To West	4	32-bit	4 GB/s	16 GB/s
To East				
AIE Tile Connected				
To AIE				
From AIE				
To DMA	2	32-bit	4 GB/s	8 GB/s
From DMA				

AIE clock @ 1GHz

AI Engine: Memory Mapped AXI4 Interconnect

- Independent interconnect
- Connection from the NOC to AIE
- Allows external access to the AIE
- Not designed to carry the bulk of the data movement
- 32-bit address
- 32-bit data
- Maximum bandwidth 1.5GB/s



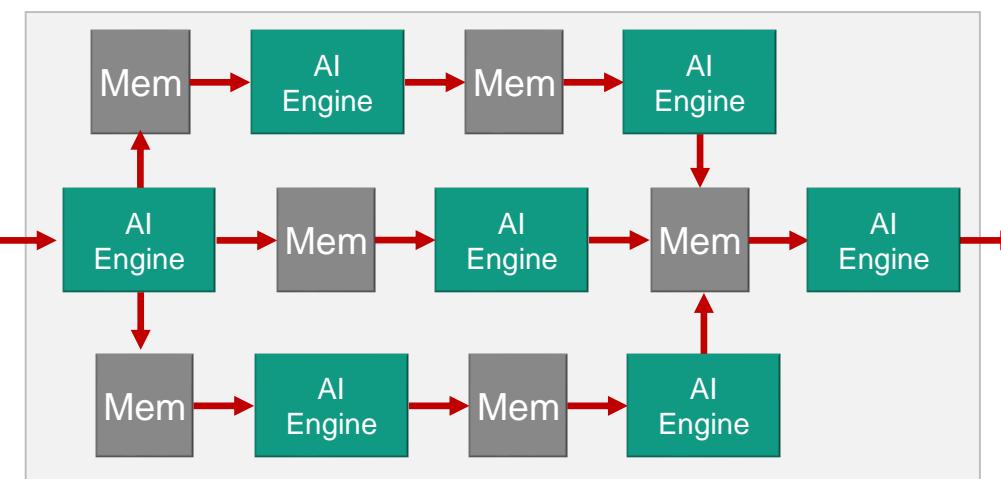
Data Movement Architecture

Memory Communication

Dataflow Pipeline



Dataflow Graph



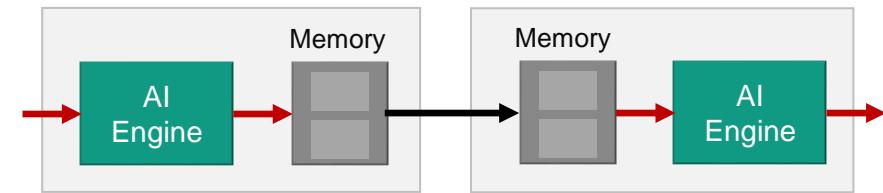
→ Memory Interface

→ Stream Interface

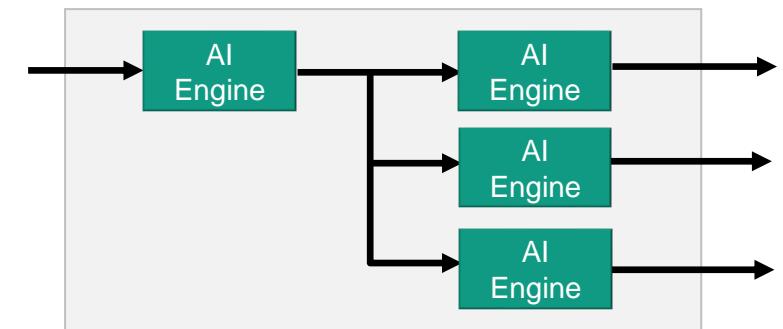
→ Cascade Interface

Streaming Communication

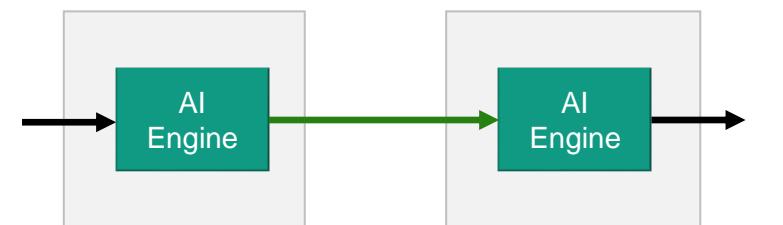
Non-Neighbor



Streaming Multicast



Cascade Streaming



AIE to AIE Data Communication via Shared Memory

Dataflow
Pipeline

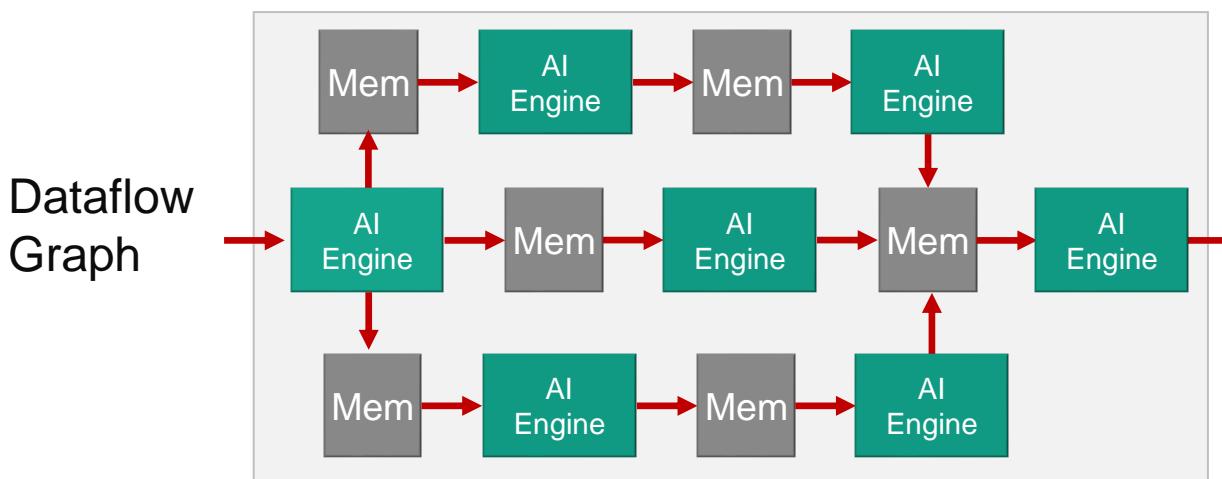


One-dimensional Pipelined Communication
Between Neighboring AI Engine Tiles

- Memory Interface
- Stream Interface
- Cascade Interface

AIE to AIE Data Communication via Shared Memory

Multi-dimensional Pipelined Communication
Between Neighboring AI Engine Tiles

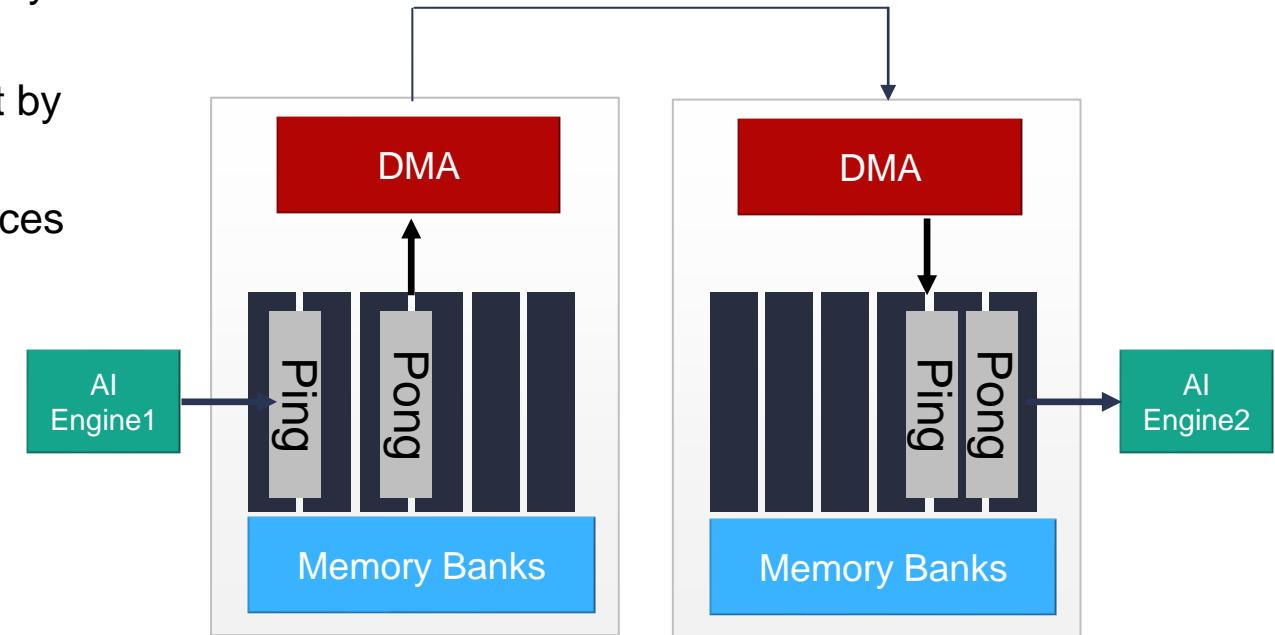


- Memory Interface
- Stream Interface
- Cascade Interface

- When multiple kernels fit in a single AI Engine:
Communication is established using a common buffer in the shared memory
- When kernels are in neighboring AI Engines:
Communication is through the shared memory module
- Communication between the two AI Engines can use ping and pong buffers on separate memory banks to avoid access conflicts
- Synchronization is done through locks
- DMA and AXI4-Stream interconnect are not needed

AIE to AIE Data Communication via Memory and DMA

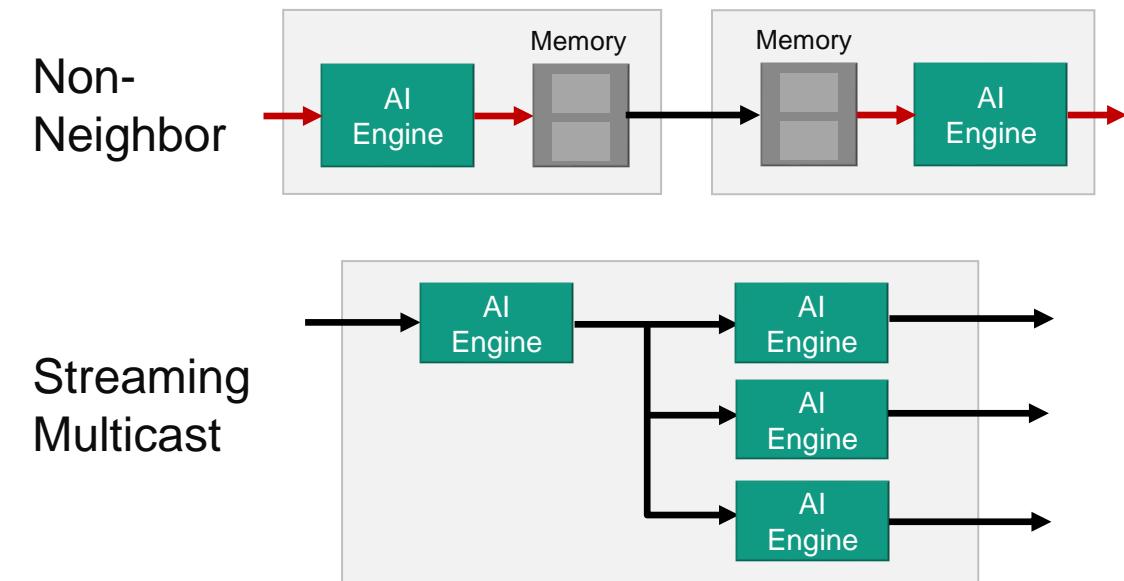
- **For non-neighboring AI Engine tiles:**
Communication is established using DMA in the memory module of each AI Engine tile
- Synchronization of the ping-pong buffers is carried out by the locks
- Increased communication latency and memory resources



- Memory Interface
- Stream Interface
- Cascade Interface

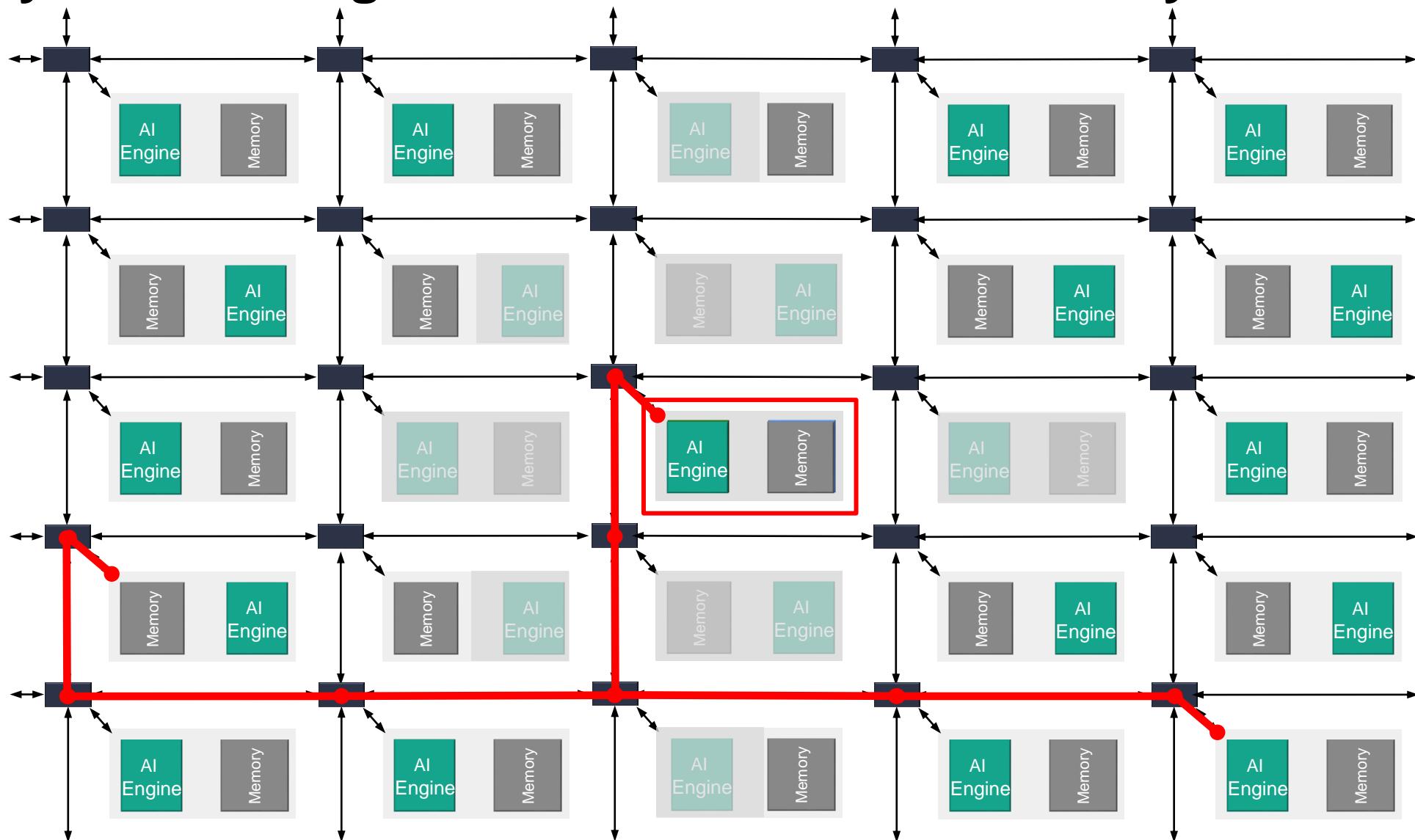
AIE to AIE Data Communication via AXI4-Stream Interconnect

- Data can be sent through the streaming interface in a serial fashion or using a multicast communication approach
- Built-in handshake and backpressure mechanisms give a feel for how data is moved around the array and the available options



→ Memory Interface
→ Stream Interface
→ Cascade Interface

Many More AI Engines and Memories in the Array

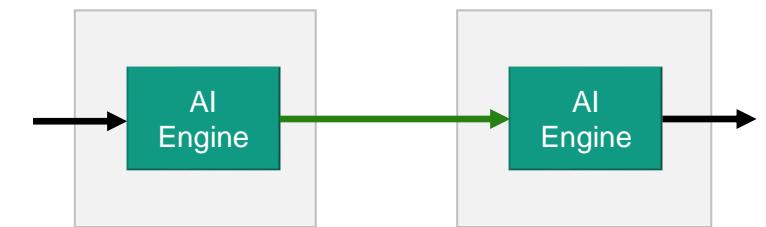


AIE to AIE Data Communication via Cascade Stream

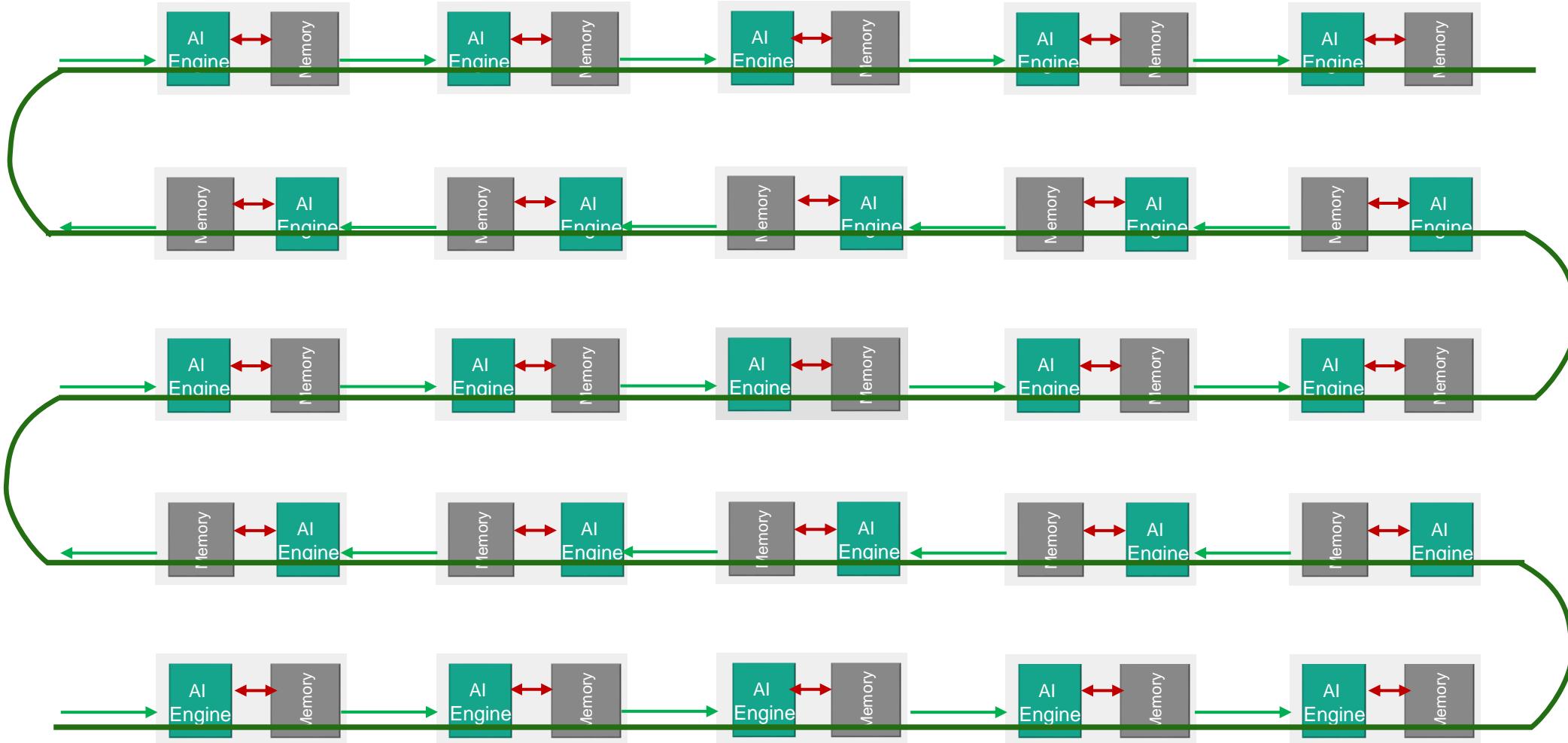
- Cascade Stream
 - Intermediate accumulation result from one AI Engine to the next

→ Memory Interface
→ Stream Interface
→ Cascade Interface

Cascade
Streaming

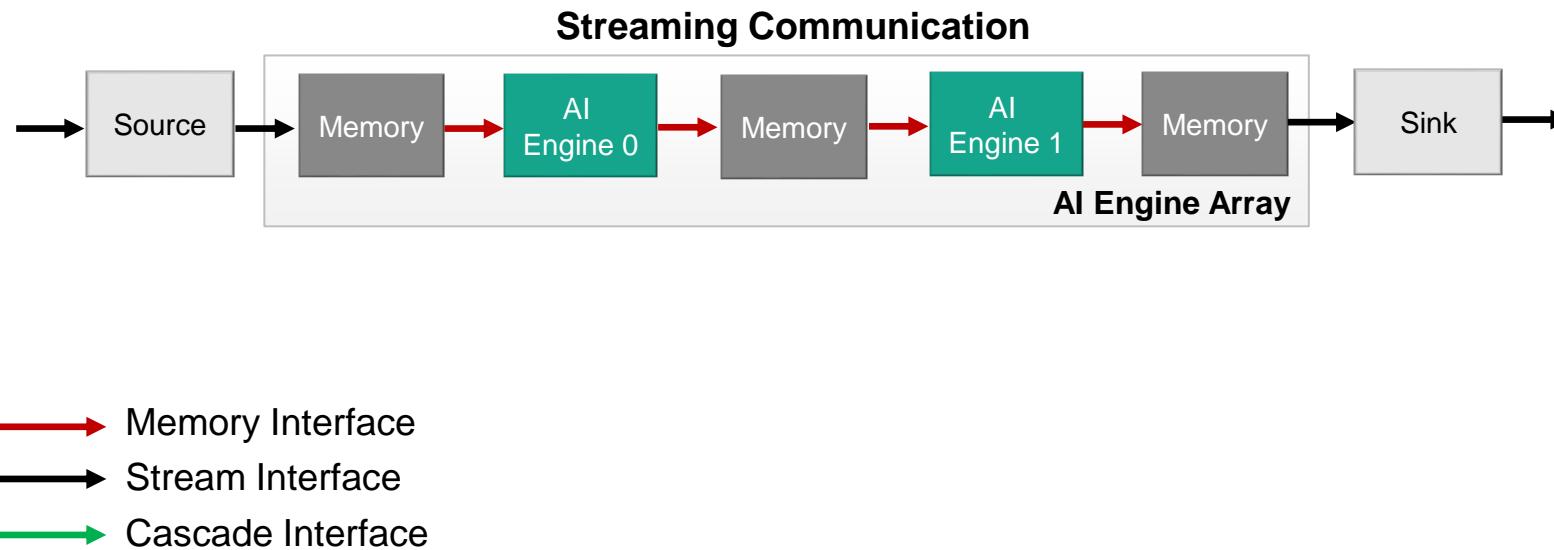


Cascade Stream Data Communication



AIE to PL Data Communication via Shared Memory

- PL block consumes data via the stream interface
- FIFO receives the PL stream and converts it into an AI Engine stream
- AI Engine stream is routed to the AI Engine destination function
- AI Engine and PL can communicate using DMA in the AI Engine tile
- DMA moves the stream into a memory block that is neighboring the consuming AI Engine



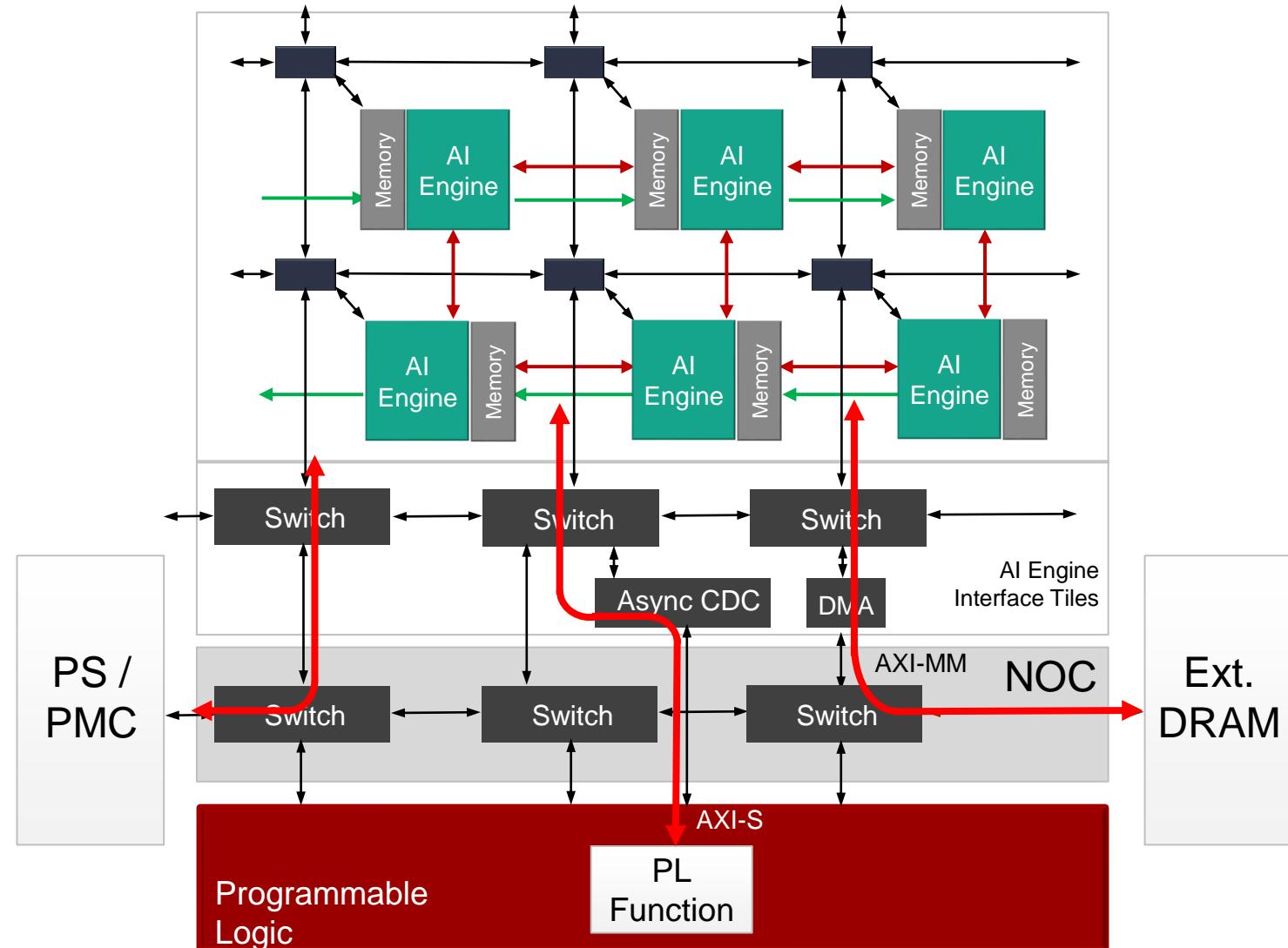
AI Engine Integration with Versal ACAP

Versal ACAP offers:

- Intelligent Engines (AIE)
- Adaptable Engines (PL)
- Scalar Engines (PS)

AIE array can communicate with:

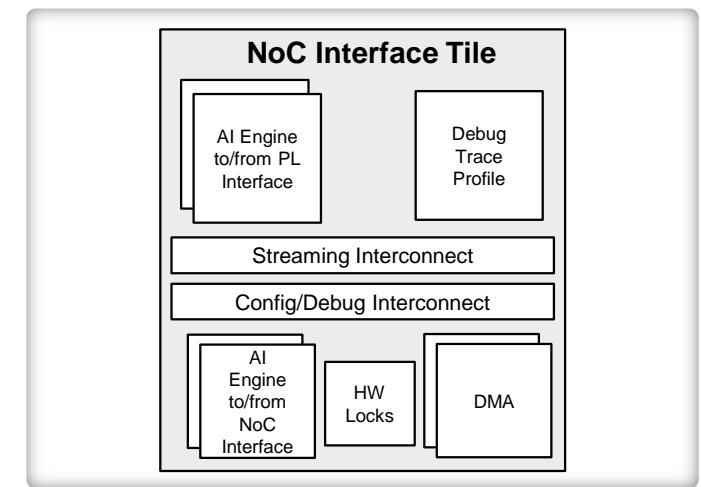
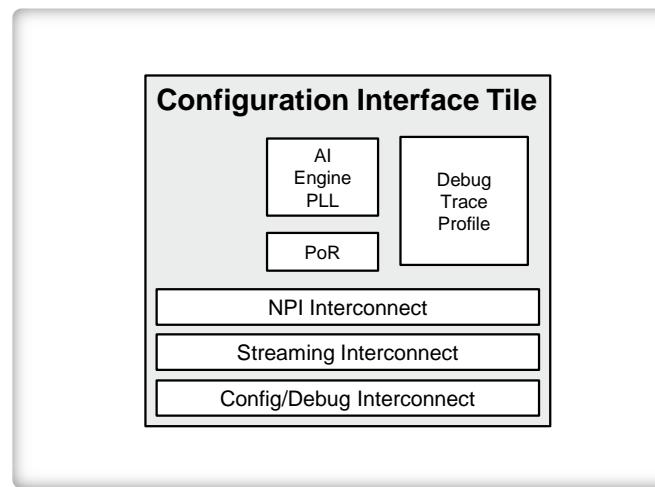
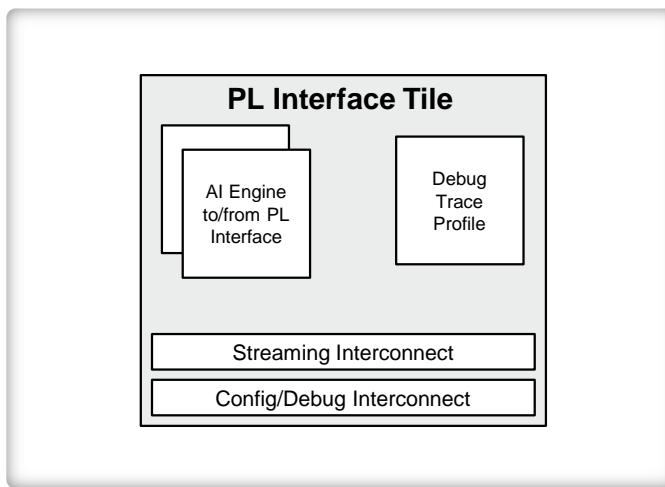
- PL via AXI4-Streams
- PS via the NoC
- DRAM via NoC



AI Engine Array Interface Tiles

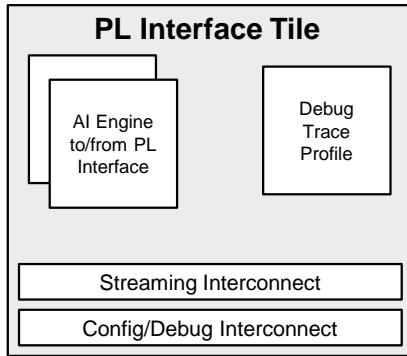
Three types of AI Engine interface tiles

- One-to-one correspondence for every column of the AI Engine array
- Form a row and move memory-mapped AXI4 and AXI4-Stream data horizontally and vertically up
- Based on a modular architecture with device-specific final composition

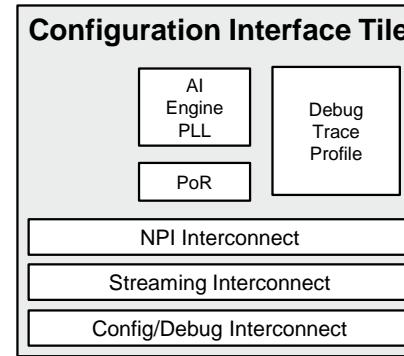


AI Engine Array Interface Tiles

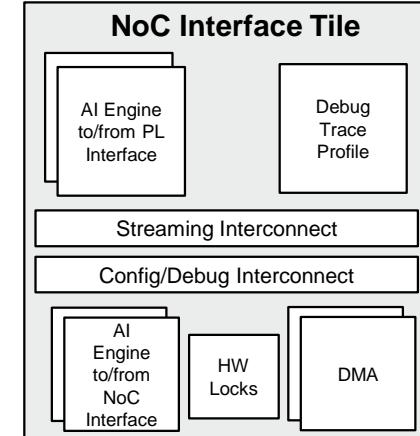
- AXI4-Stream switch
- Memory-mapped AXI4 interface
- AI Engine to PL stream interface
- Control, debug, and trace unit



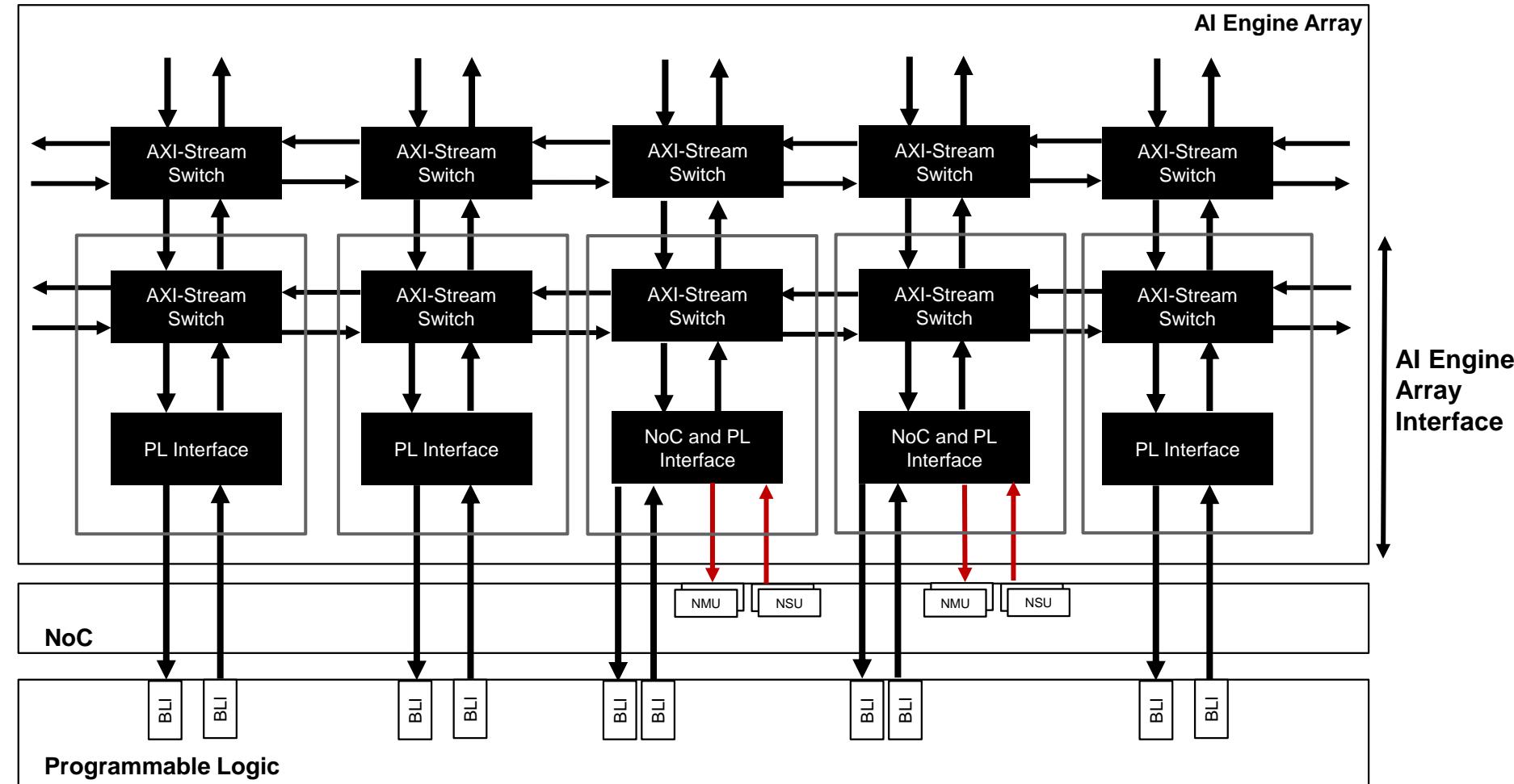
- PLL for AI Engine clock generation
- Power-on-reset (POR) unit
- Interrupt generation unit
- DFX logic
- NoC peripheral interconnect (NPI) unit
- AI Engine array global registers



- All the PL module blocks
- NoC modules with interfaces to the NoC master unit (NMU) and NoC slave unit (NSU)
 - Bi-directional NoC streaming interface
 - Array interface DMA

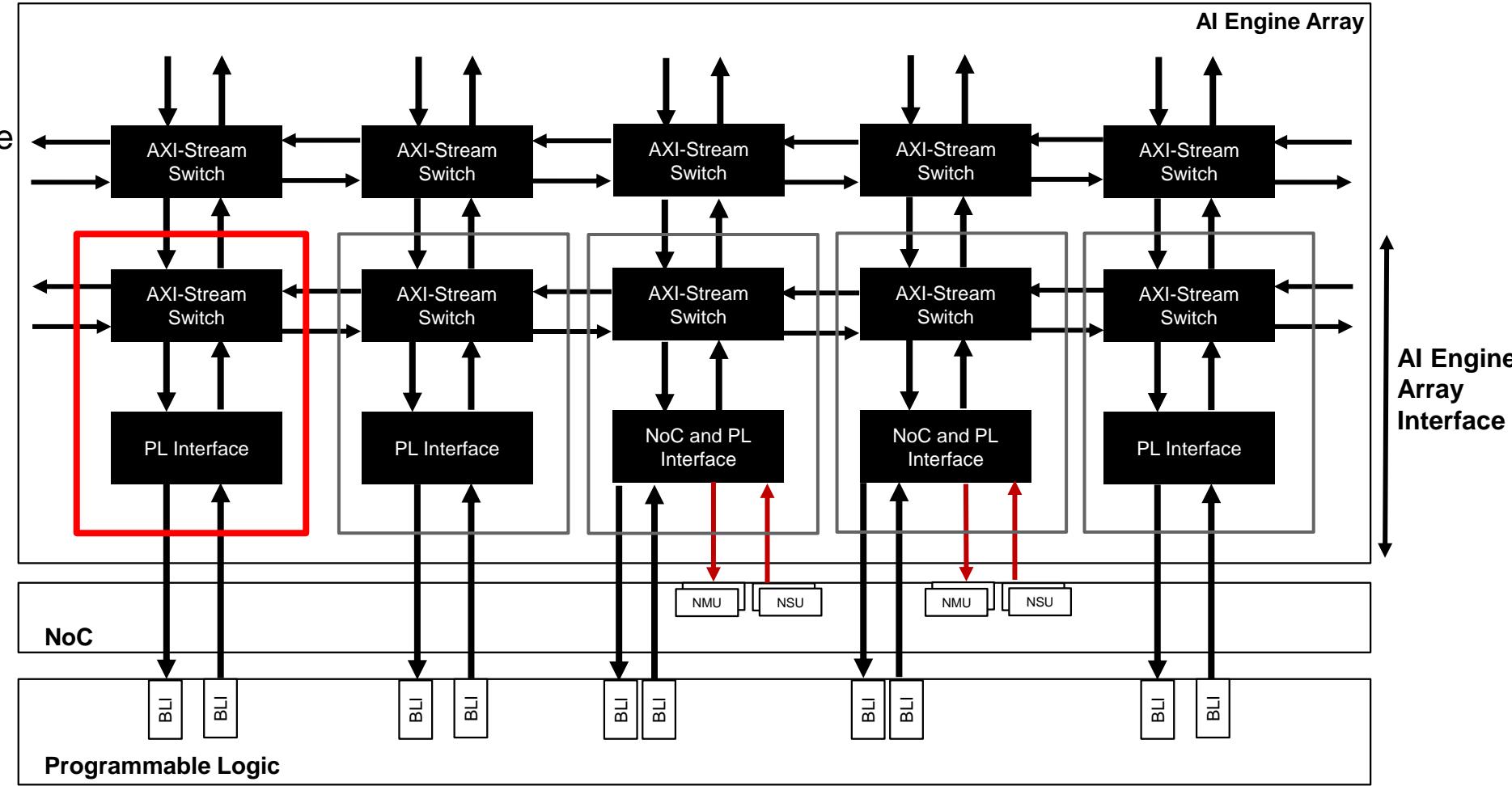
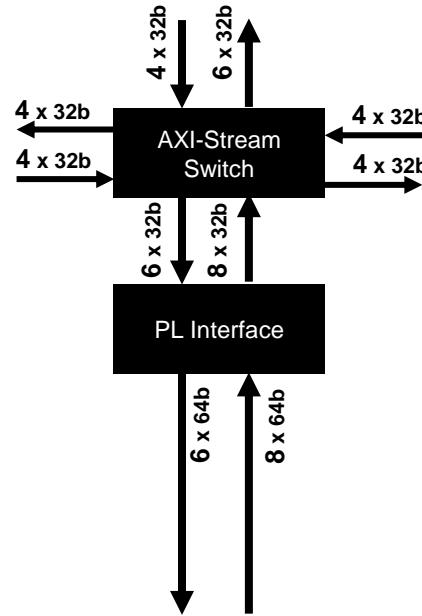


NoC / PL Communication with AI Engine Array



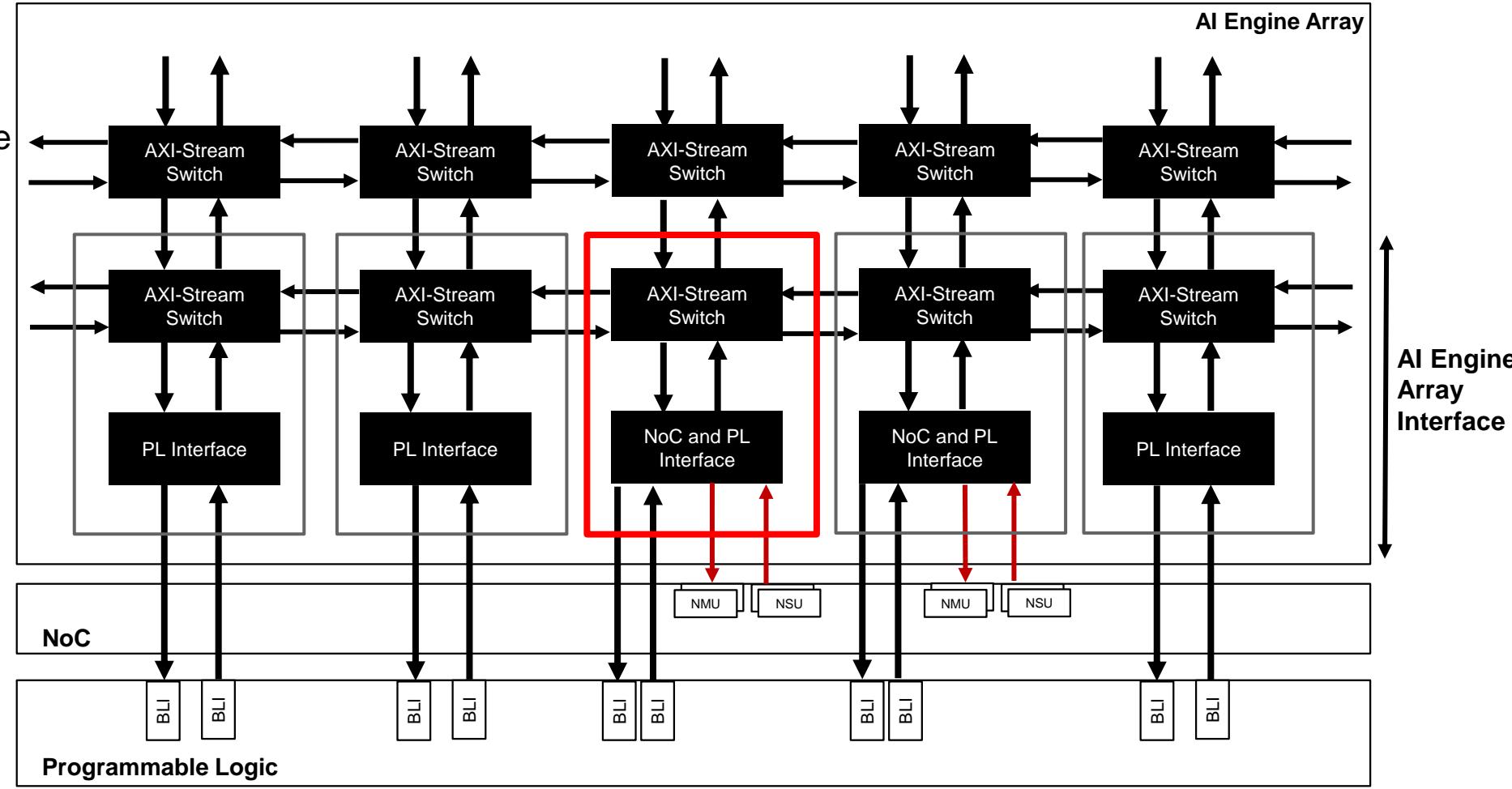
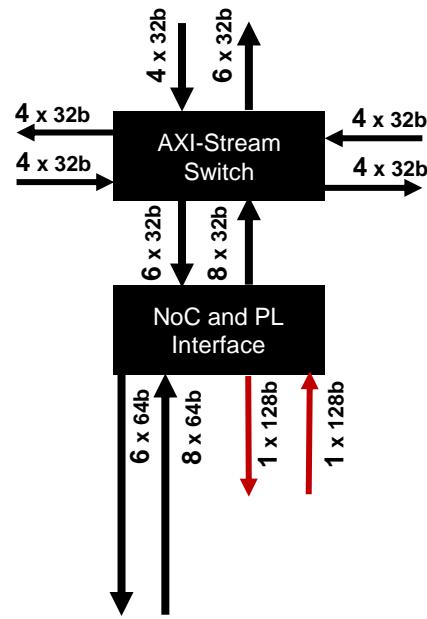
HNoC / PL Communication with AI Engine Array

- PL Interface Tile
- Connects to
 - Boundary Logic Interface
 - 6 @ 500 MHz
 - 2 @ 300 MHz



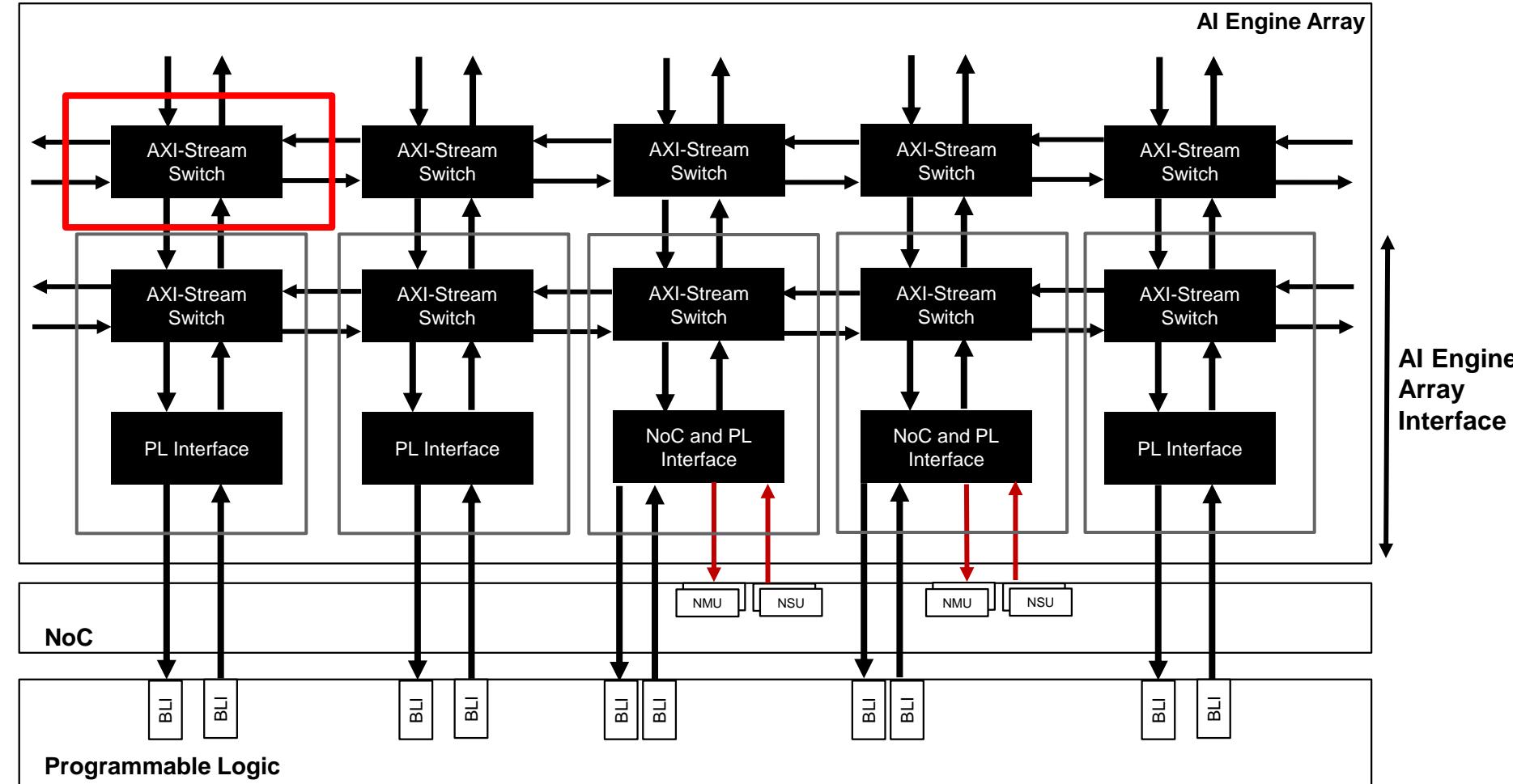
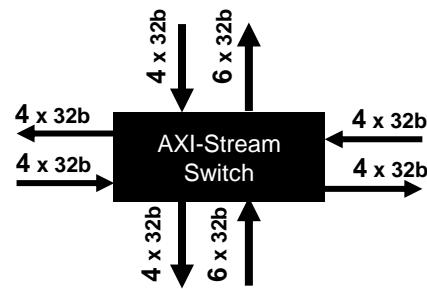
HNoC / PL Communication with AI Engine Array

- NoC Interface Tile
- Connects to
 - Boundary Logic Interface
 - 6 @ 500 MHz
 - 2 @ 300 MHz
 - NoC Master Unit
 - NoC Slave Unit



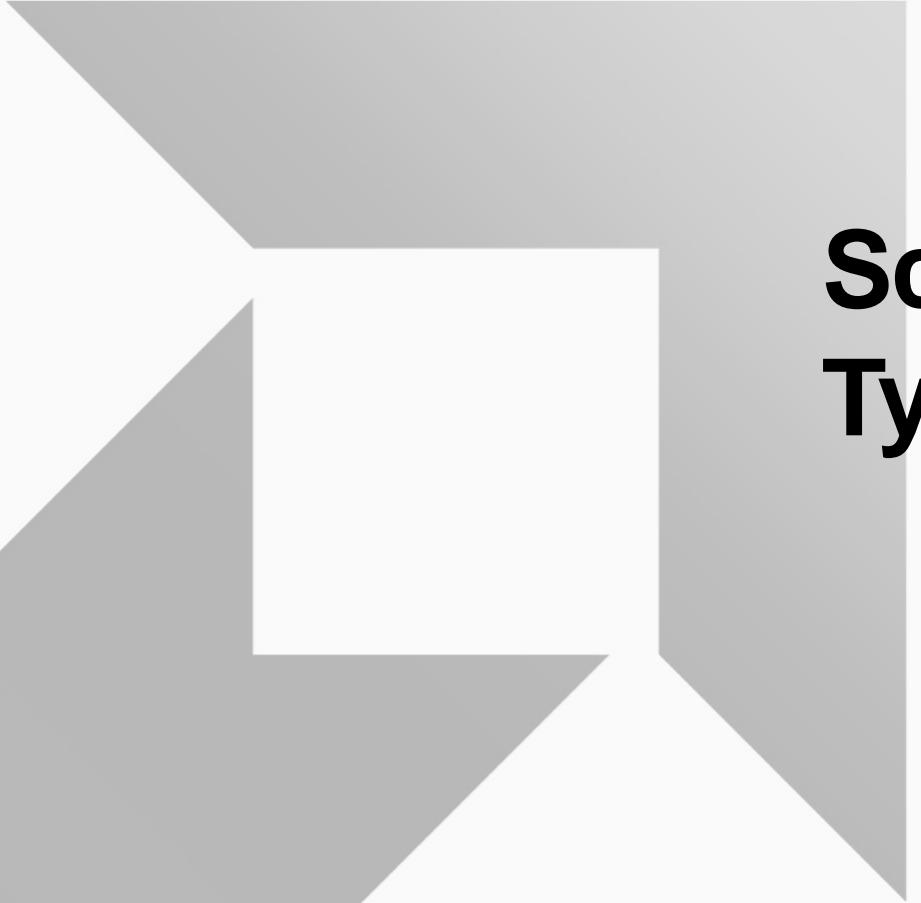
HNoC / PL Communication with AI Engine Array

- AXI-Stream Switch
- Connects to AIE tiles



Summary

- The AI Engine memory module contains:
 - Eight memory banks
 - Two input streams to memory map (S2MM) DMA
 - Two memory-map to output DMA streams (MM2S)
 - Hardware synchronization module (locks)
- Data communications can happen between:
 - AI Engine tile to AI Engine tile via shared memory
 - AI Engine tile to AI Engine tile via memory and DMA
 - AI Engine tile to AI Engine tile via the AXI4-Stream interconnect
 - AI Engine to programmable logic (PL)



Scalar and Vector Data Types

Objectives

After completing this module, you will be able to:

- Describe the functional blocks of scalar units
- Identify the supported vector data types for Versal® AI Engines and the AI Engine API
- Describe AI Engine support for high-width registers to allow SIMD instructions

Scalar Processor Unit - Scalar Programming

- Supports all standard C scalar data types
- Program control flow determined by scalar instructions
- No logical operations defined on vector types

Data Type	Precision	Comment
char	8-bit signed	-
short	16-bit signed	-
int	32-bit signed	Native support
long long	64-bit signed	-
float	32-bit	Emulated using SoftFloat library
double	64-bit	Scalar proc does not contain FPU

Vector Processor Unit - Overview of Vector Data Types

Elements

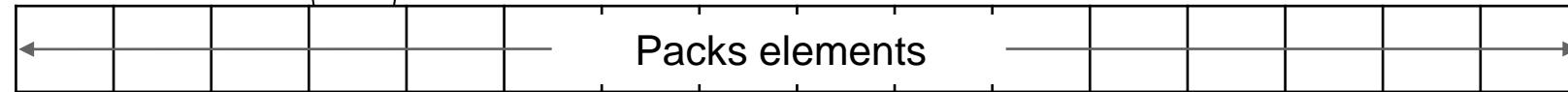
Real

Real

Imaginary

8-bit, 16-bit, 32-bit
real or complex elements

Vector Data Type



128-bit / 256-bit / 512-bit / 1024-bit data types

Vector Datapath: 4, 8, 16, 32, 64 or 128 lanes

Vector Data Types

The AI Engine vector unit supports:

- Integers and complex integers
- Real and complex single-precision floating-point numbers
- Accumulator vector data types
- Intrinsic functions operate using these vector data types

Vector Data Types – Vector Registers

{ } – Mandatory
[] – Optional

v{NumLanes} [c]{[u]int|float} {SizeofElement}

NumLanes

Number of output lanes

Element data type

int denotes integer

float denotes floating point

c denotes complex elements

u denotes unsigned, only
uint8

Size of Element

Element bitwidth

8-bit, 16-bit, 32-bit

Does not apply for float

Data width of vector data types: 128/256/512/1024-bit

Vector Data Types – Accumulator Registers

{ – Mandatory
[] – Optional

v{NumLanes} [c]acc {SizeofElement}

NumLanes

Number of output lanes
Either 4, 8 or 16

Element data type

acc denotes accumulator
c denotes complex accumulators

Size of Element

Accumulator bitwidth
48/80-bit
No floating-point accumulator

Data width of accumulator data types: 320/364-bit or 640/784-bit

AI Engine API – Based on C++

- Common API across AIE architectures
- Parameterizable data types
- Higher level programming abstraction than intrinsics
- Implemented as a C++ header-only library
- Templates for datatypes

```
#include "aie_api/aie.hpp"  
#include "aie_api/aie_adf.hpp"
```



```
aie::vector<Type, NumberOfLanes>  
aie::accum<Type, NumberOfLanes>
```



Supported Data Types for AIE

- Vector and accumulator types have naming conventions
- Supported vector types and sizes

	int8	uint8	int16	int32	float	cint16	cint32	cfloat
Lanes	16/32/64/128		8/16/32/64		4/8/16/32		2/4/8/16	

- Supported accumulator types and sizes

	acc48	acc80	cacc48	cacc80
Lanes	8/16	2/4/8	4/8	2/4
Accumulation	48b	80b	48b	80b

List of Supported Data Types for AIE

Vector Data Types

	128-bit	256-bit	512-bit	1024-bit
Complex	v2cint32	v4cint32	v8cint32	v16cint32
	v4cint16	v8cint16	v16cint16	v32cint16
	v2cfloat	v4cfloat	v8cfloat	v16cfloat
Real	v4int32	v8int32	v16int32	v32int32
	v8int16	v16int16	v32int16	v64int16
	v16int8	v32int8	v64int8	v128int8
	v4float	v8float	v16float	v32float

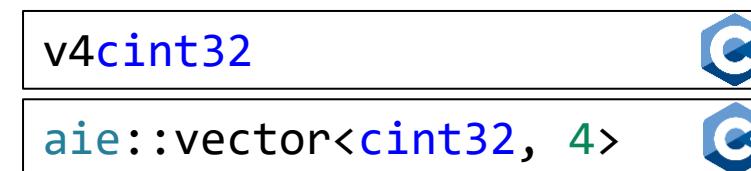
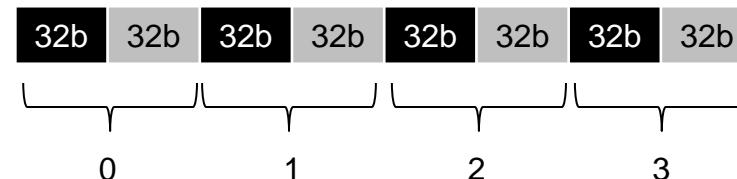
Accumulator Data Types

	320/384-bit	640/768-bit
Complex	v2cacc80	v4cacc80
	v4cacc48	v8cacc48
Real	v4acc80	v8acc80
	v8acc48	v16acc48

Vector Data Type

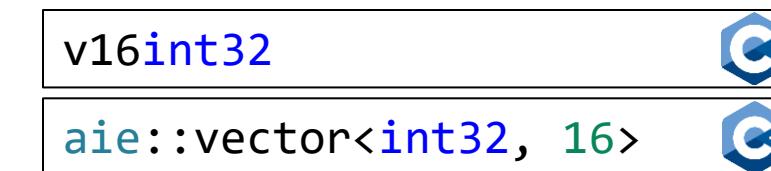
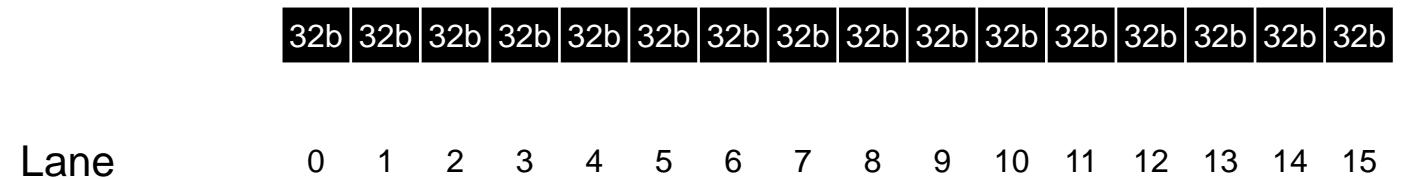
4-lane complex vector

- 32-bit samples
- 256-bit width



16-lane real vector

- 32-bit samples
- 512-bit width

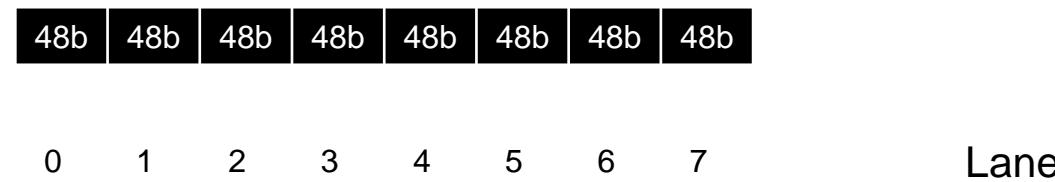


Real
Imaginary

Accumulator Data Type

8-lane real accumulator

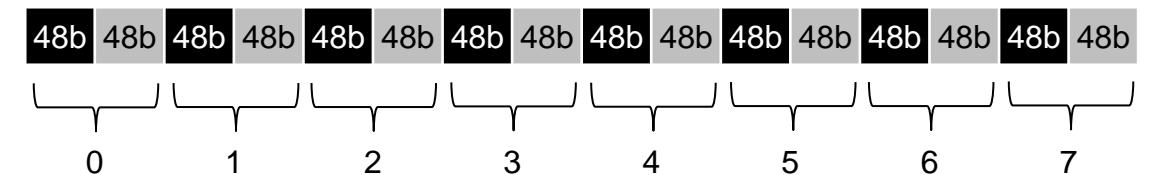
- 48-bit samples
- 384-bit width



```
v8acc48          C  
aie::accum<acc48, 8>  C++
```

8-lane complex accumulator

- 96-bit samples
- 768-bit width



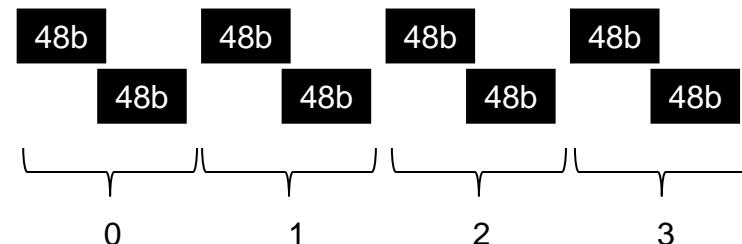
```
v8cacc48          C  
aie::accum<cacc48, 8>  C++
```

Real
Imaginary

Accumulator Data Type

4-lane real accumulator

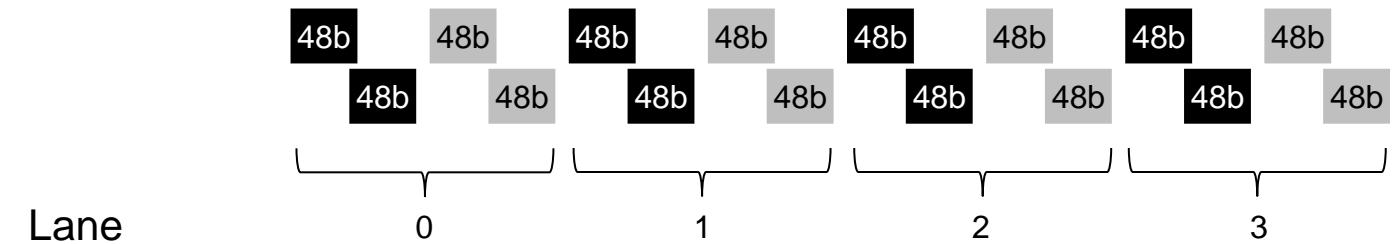
- 80-bit samples
- 320-bit width



```
v4acc80          C
aie::accum<acc80, 4> C++
```

4-lane complex accumulator

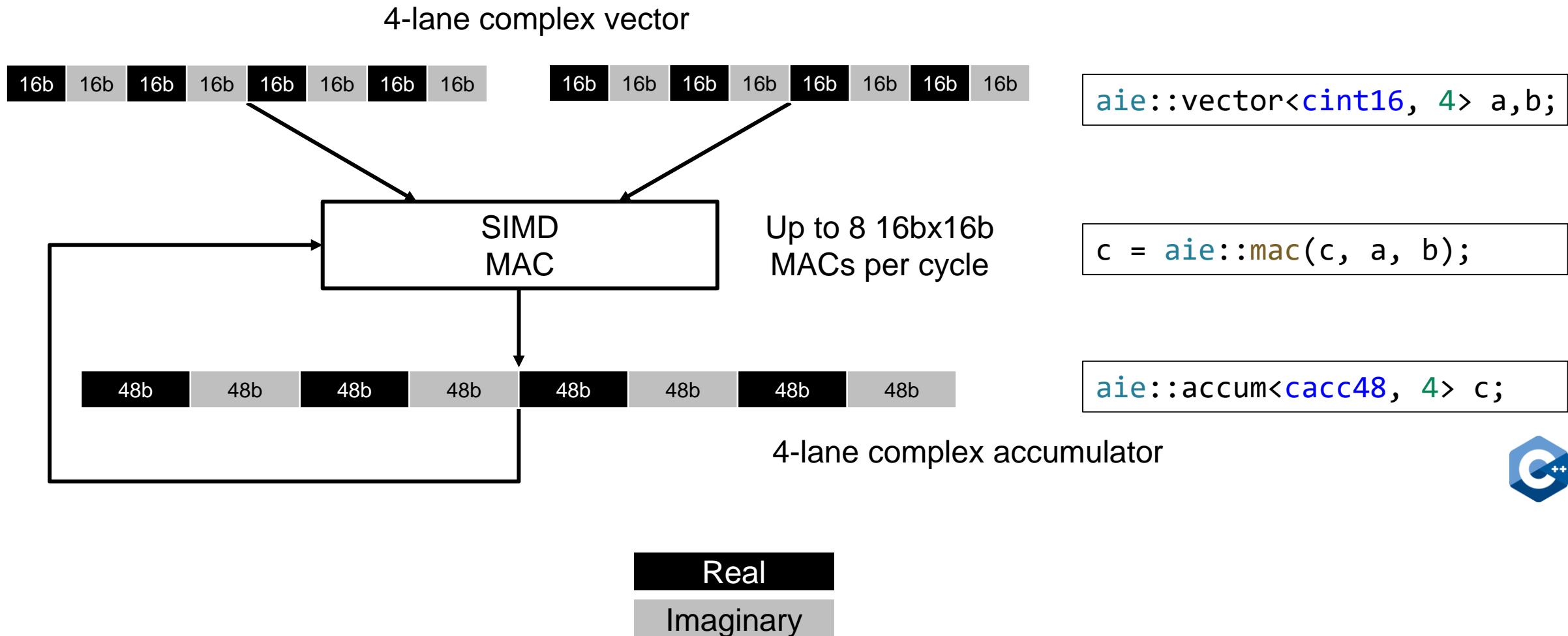
- 160-bit samples
- 640-bit width



```
v4cacc80         C
aie::accum<cacc80, 4> C++
```

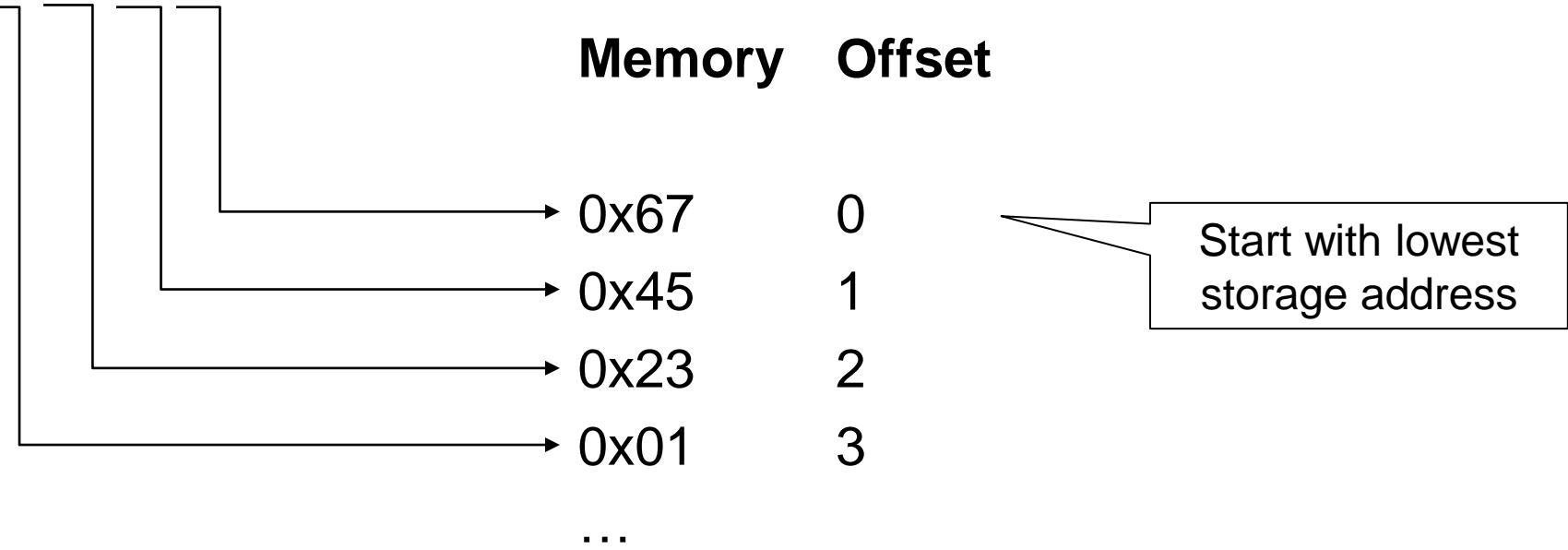
Real
Imaginary

Accumulator and Vector Types



Endianness

Example Data: 0x01234567

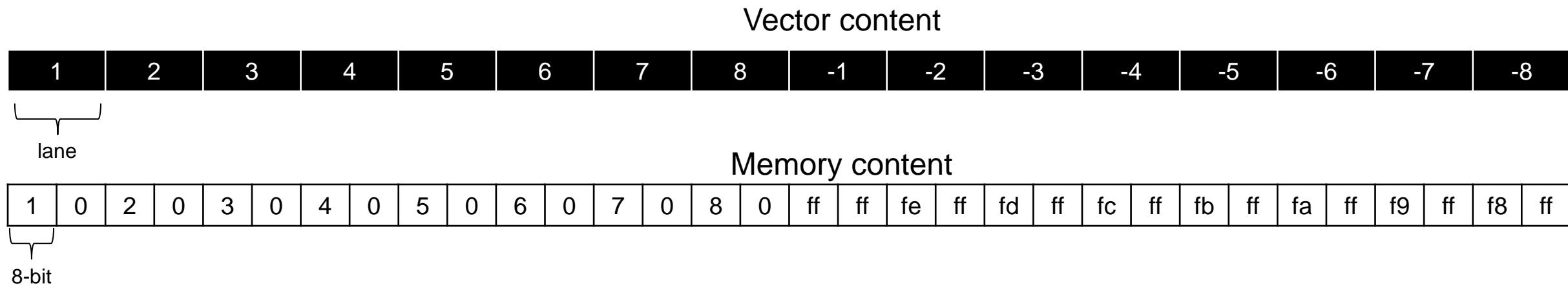


Vectors are stored in little-endian format

Endianness - Example

- Data and vectors have little-endian format
 - Data is stored in Least Significant Byte (LSB) first
 - Vector is stored in lowest element index first

v16int16 
aie::vector<int16, 16> 



Vector Register File

- High-width registers enabling SIMD instructions
- Incrementally grouped for higher widths

128-bit	256-bit	512-bit	1024-bit
vrl0	wr0	xa	N/A
vrh0			
vrl1	wr1		
vrh1			
vrl2	wr2		ya
vrh2		xb	
vrl3	wr3		yd(MSBs)
vrh3			
vcl0	wc0		
vch0		xc	N/A
vcl1	wc1		N/A
vch1			
vdl0	wd0		
vdh0		xd	N/A
vdl1	wd1		yd(LSBs)
vdh1			

Accumulator Register File

- Accumulator registers (am)
- Store results of vector data path
- 8 vector lanes of 48-bit
- 32-bit multiplication results and accumulation without bit overflow

384-bit	768-bit
aml0	bm0
amh0	
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

Register Management with Large Vectors – Examples

v4cint32	
aie::vector<cint32, 4>	

v16int32	
aie::vector<int32, 16>	

v8acc48	
aie::accum<acc48, 8>	

2 x 128-bit vector registers

128-bit	256-bit
vrl0	
vrh0	wr0

4 x 128-bit vector registers

128-bit	256-bit	512-bit
vrl0		
vrh0	wr0	
vrl1		
vrh1	wr1	xa

2 x 384-bit accumulator registers

384-bit	768-bit
aml0	
amh0	bm0

AI Engine uses multiple registers for wide vector and accumulator types

Register Management with Large Vectors – Register Spilling

- Vector registers are a limited resource
 - Register spilling occurs when the compiler runs out of registers
 - Vectors will be stored in memory
 - Extra instructions, thus less overall utilization
 - Microcode with register name is show in the disassembly view

Debugger Vector Register Views

Vector Buffer	Memory	Configuration	Regular Load	Wide Load
XA = WR1::WR0	512b	vector<cint16,16>	vector<cint16,4>	vector<cint16,8>

```
00000000000000001218: VLDA wr0, [p2], m0, cycl
00000000000000001226: VLDA wr1, [p2];
00000000000000001234: NO
00000000000000001236: NO
00000000000000001238: NO
00000000000000001240: NOP;
00000000000000001244: VLDA wr0, [p2], m0, cycl
00000000000000001252: NOP;
00000000000000001260: NOP;
00000000000000001268: VLDA wr1, [p2];
00000000000000001280: NOP;
00000000000000001288: NOP;
00000000000000001296: NOP;
00000000000000001304: NOP;
00000000000000001312: VLDA wr0, [p2], m0, cycl
```

128-bit	256-bit
vrl0	wr0
vrh0	
vrl1	wr1
vrh1	
vrl2	wr2
vrh2	
vrl3	wr3
vrh3	
vcl0	wc0
vch0	
vcl1	wc1
vch1	
vdl0	wd0
vdh0	
vdl1	wd1
vdh1	

Summary

- The AI Engine compiler and processor provide support for scalar programming using standard C/C++ types
- Vector data types pack multiple scalar data elements into a wider vector
- Vector datatypes can be defined with intrinsics or C++ template
- The AI Engine uses high-width registers for allowing SIMD instructions
- Data and vectors have little-endian format
 - Data is stored in LSB first
 - Vector is stored in lowest element index first



AI Engine APIs and Intrinsic Functions

Objectives

After completing this module, you will be able to:

- Describe what intrinsic functions are
- Describe the available vector manipulating functions and their operations
- Identify the AI Engine APIs for various vector manipulations
- Describe the advanced MAC intrinsic functions
- List the available application-specific intrinsic functions

What are Intrinsic Functions?

- Built-in functions
- Implementation is handled by the compiler
- Compiler has deep knowledge of intrinsic functions
- Treated as inline function, no call overhead
- Better integration and optimization
 - Produces efficient microcode
- Typically used to implement vectorization and parallelization

```
acc0 = mul16(xbuff, 0, 0x11101110, 16, 0x3120, zbuff, 0, 0x44440000, 2, 0x3210);
```

AI Engine – Intrinsic Functions

- C intrinsic functions invoke vector operations
- The compiler takes care of:
 - Mapping each intrinsic function to a sequence of assembly instructions
 - Register allocation and data movement
 - Automatic scheduling
- Intrinsic functions are higher level than assembly programming
 - However, knowledge of the architecture is required

Intrinsic Functions - Example

- Specialized data types and intrinsics are available for vector programming
- Provides control and predictable timing for execution
- Intrinsics are also available to move data between the scalar and vector processors
- Examples:
 - Casting
 - Extract sub vector

```
v8int16  vec = *a;
v4cint16 res = as_v4cint16(vec);
```

vec



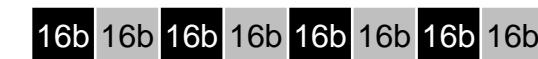
res



buf



Returns a v4cint16



Real

Imaginary

Intrinsic Functions - Documentation

The screenshot shows a web browser displaying the **AI Engine Intrinsics User Guide** (AIE r2p22). The URL is https://www.xilinx.com/htmldocs/xilinx2022_2/aiengine_intrinsics/intrinsics/group__intrinsics.html. The page title is "AI Engine Intrinsics User Guide". The left sidebar contains navigation links for "Overview", "Related Pages", and "API Reference". Under "API Reference", the "Intrinsics" section is selected, showing a list of intrinsic categories: Application Specific Intrinsics, Load/Store Operations, Scalar Operations, Vector Conversions, Vector Operations, Configuration, Full Lane Addressing Scheme, Initialization, MAC intrinsics, Reduced Lane Addressing Scheme, Vector Arithmetic, Vector Compares, Vector Lane Selection, and Native stream access.

Check out [UG 1078](#) for more information

AI Engine API – Based on C++

- Common API across AIE architectures
- Parameterizable data types
- Higher level programming abstraction than intrinsics
- Implemented as a C++ header-only library
- Templates for datatypes

```
#include "aie_api/aie.hpp"  
#include "aie_api/aie_adf.hpp"
```



```
aie::vector<Type, NumberOfLanes>  
aie::accum<Type, NumberOfLanes>
```



AI Engine API - Documentation

The screenshot shows a web browser displaying the AI Engine API User Guide (AIE) 2022.2. The page title is "AI Engine API User Guide (AIE) 2022.2". The navigation bar includes links for "Overview", "Related Pages", and "API Reference", along with a search bar. The main content area is titled "Overview" and contains the following text: "AIE API is a portable programming interface for AIE accelerators. It is implemented as a C++ header-only library that provides types and operations that get translated into efficient low-level intrinsics. The API also provides higher-level abstractions such as iterators and multi-dimensional arrays." On the left, there is a sidebar with a tree view of the API Reference, listing categories like Changelog, API Reference (Basic Types, Memory, Initialization, Arithmetic, Comparison, Reduction, Reshaping, Floating-point Conversion, Elementary Functions, Matrix Multiplication, Fast Fourier Transform (FFT), Special Multiplications, Operator Overloading, Interoperability with Adaptive Data Flow (ADF) Graph Abstractions), and a link to the Changelog.

Check out [UG 1529](#) for more information

Vector Management Functions

- Load and Store
 - Perform vector initialization
- Vector Conversions
 - Shift-round-saturate
 - Upshift
- Lane Insertion and Extraction
 - Extract
 - Update

Loading and Storing Vectors

- Vector and accumulator types are stored in memory
 - Static allocation
 - Allocation on the stack
- Compiler supports pointer dereferencing and pointer arithmetic
 - No intrinsic needed
- Vectors are initialized by casting a pointer from a statically initialized array
- All vector loads and stores must be aligned to 128-bit boundaries
 - Stalls the processor if the data pointers are aligned to 256-bit loads from the same memory bank
- Accumulators are padded when transferred to memory
 - To align with 128-bit boundaries
- Each AI Engine supports simultaneous loads and stores
 - Support for **two** independent 128-bit or 256-bit **loads** in each cycle
 - Support for **one** 128-bit or 256-bit **store** in each cycle
- Data and vectors have little-endian format

Vector Initialization and Loading

Real

Imaginary

Initializing
vector
values of
`x_a`

```
// vector initialization values
alignas(16) int32 x_a[8] = {0,1,2,3,4,5,6,7};
```

`x_v` vector
values are
again
copied to `x`

```
// cast a pointer to initialize vector
v4cint32 *x_v = (v4cint32 *)x_a;
v4cint32 x = *x_v;
```

`x` = [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7]

`x_a` has
been type
cast and
the `x_v`
vector
values are
initialized

`alignas` standard C specifier can be used to ensure proper alignment of local memory

Memory Alignment – AI Engine API

- Functions are provided for both aligned and unaligned accesses

	Load	Store
Aligned access	aie::load_v	aie::store_v
Unaligned access	aie::load_unaligned_v	aie::store_unaligned_v

Unaligned accesses can incur additional overhead depending on the amount of misalignment

Memory Alignment – AI Engine API

- Buffers are aligned using standard C/C++ facilities such as alignas

```
alignas(aie::vector_decl_align) int16 buffer[N]; // Generic alignment declaration
```

```
alignas(aie::vector_ldst_align_v<cint16, 8>) int16 reim[16]; // Type specific alignment
```

Global constant values used to align the buffer to a boundary that works for any vector size

Memory Bank Conflict – AI Engine APIs

- AIE Compiler
 - Evenly distribute buffers from communication primitives
 - Schedule many accesses in the same instruction when possible
 - Provides type annotations to associate memory accesses to virtual resources
- Bind individual accesses to a virtual resource
 - Most memory access functions accept an attribute from `aie_dm_resource`

```
void fn(int __aie_dm_resource_a * A, int * B, *D, int __aie_dm_resource_a * C) {  
  
    aie::vector<int,8> v1 = aie::load_v<8>(A); // Access from A and C are bound to the same virtual resource so they  
    aie::vector<int,8> v2 = aie::load_v<8>(B); // are never scheduled on the same instruction. B is not annotated so  
    aie::vector<int,8> v3 = aie::load_v<8>(C); // its memory accesses can be scheduled in the same instruction with  
                                                // accesses to A or C.  
    auto v4 = aie::load_v<8, aie_dm_resource::b>(D); // Memory access can be annotated in load operation  
    ...  
}
```

Accesses using types that are associated to the same virtual resource will not be scheduled in the same instruction

Vector Initialization and Loading

- Various Ways to Initialize Vector Values
- Casting Intrinsics

```
// Vector Initialization Values
alignas(16) const int32 reals[8] = {1, 2, 3, 4, 5, 6, 7, 8};

// ways to initialize
v8int32 v;                                // uninitialized
v8int32 uv = undef_v8int32();    // initialized as undefined
v8int32 nv = null_v8int32();    // initialized as zeros
v8int32 iv = *(v8int32 *) reals; // initialized with data from local memory
v16int32 sv = xset_w(0, iv);   // init. with half values from another register
```

Declaration and Initialization – AI Engine API

- Initialization constructs

```
aie::vector<int16, 16> v;                                // Default constructor -> value undefined
aie::vector<int16, 16> v2 = v1;                            // Initialized with another vector
aie::vector<int16, 16> v3 = aie::add(v1, v2);            // initialized with an operation

aie::vector<int16, 32> w = aie::concat(v1, v2, ...);      // Vectors are concatenated to a larger vector

// Initialization with identical values
aie::vector<int16, 16> v0 = aie::zeros<int16, 16>();      // Initialized with all zeros
aie::vector<int16, 16> va = aie::broadcast<int16, 16>((int16)a); // Initialized with all a
```

- Valid vector and accumulators

```
aie::accum<acc80, 8> acc1;                            // Default constructor -> value undefined
aie::accum<cacc48, 8> acc2(aie::zeros <cacc48, 8>()); // Initialized with zeros
```

Interface Access Examples – AI Engine API

```
bool tlast;
// interface: input_window<cint16> *in
//      -> Reads 8x cint16 samples and increments the pointer of the same amount
aie::vector<cint16, 8> v = window_readincr_v<8>(in);

// interface: input_window<cint16> *in
//      -> Reads 16x int16 samples and does not change the pointer
//      -> Next read will start at the same position
aie::vector<int16, 16> v = window_read_v<16>(in);

// interface: output_window<cint32> *out
//      -> Writes 2x cint32 samples and changes the pointer to write at next position afterwards
window_writeincr(out, (aie::vector<cint32, 2>) data );

// interface: input_stream<cint16> *in
//      -> Reads 4x cint16 samples and tlast value
aie::vector<cint16,4> v = readincr_v<4>(in,tlast);

// interface: output_stream<int16> *in -> Writes 8x int16 samples and gets ready for the next vector
writeincr<8>(in, aie::vector<int16, 8> v);
```

Vector Initialization and Loading

- Initialize with half values from another register
- Valid vector set intrinsic functions: wset_v, xset_v, xset_w, yset_v, yset_w, and yset_x

[T]set_[R]

[T] indicates the target vector register to be set:

- w for W register (256-bit)
- x for X register (512-bit)
- y for Y register (1024-bit)

[R] indicates where the source value comes from:

- v for V register (128-bit)
- w for W register (256-bit)
- x for X register (512-bit)

```
v16int32 sv = xset_w(0, iv);
```

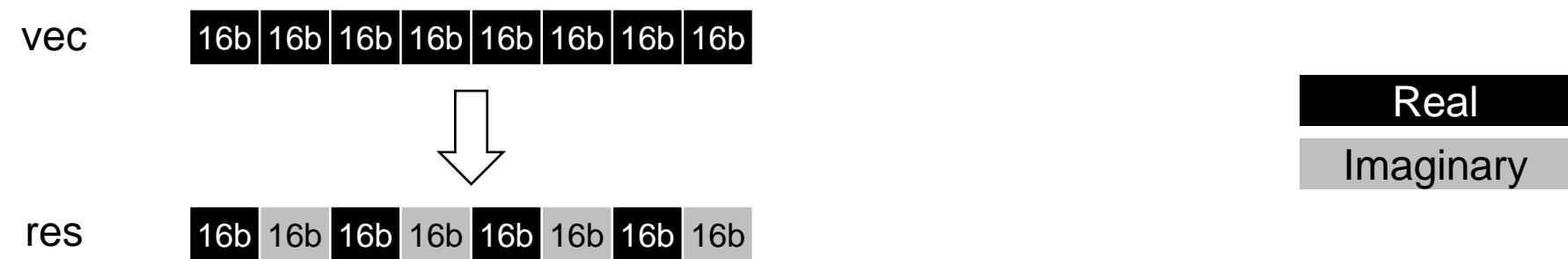
[R] width is smaller than [T] width

Vector Initialization and Loading – Casting Intrinsics

- Library of casting functions allows type casting
 - Convert any type to any other similarly sized type
 - No assembly instructions, no cycles lost

```
v8int16  vec = *a;  
v4cint16 res = as_v4cint16(vec);
```

```
aie::vector<int16, 8> vec = *a;  
aie::vector<cint16, 4> res = vec.cast_to<cint16>();
```



Casting Between Vector and Accumulator

- Shift-round-saturate (srs) instructions are used to move values from accumulators to vectors
 - Requires a reduction in precision
- In fixed-point arithmetic
 - Shifting (right)
 - Rounding
 - Saturation
- Applied by using the srs family of intrinsics
- Casting a vector to an accumulator requires shifting (left)
- Each lane performs a separate shifting, rounding, and saturation depending on the parameters

SRS Rounding & Saturation – AI Engine API

- Rounding and saturation are enabled using configuration bits
- When rounding is set, round to nearest
- When rounding is cleared, function truncates

Rounding Mode	Value
floor	0
ceil	1
positive_inf	2
negative_inf	3
symmetric_inf	4
symmetric_zero	5
conv_even	6
conv_odd	7

Saturation Mode	Value
none	0
truncate	1
symmetric	2

Move Accumulator to Vector - Example

- 80-bit Accumulator

- Produces 32-bit or 64-bit vectors

```
v8int32 srs(v8acc80 a, int shft)  
v8int64 lsrs(v8acc80 a, int shft)
```

- 48-bit Accumulator

- Produces 16-bit or 32-bit vectors

```
v8int16 srs(v8acc48 a, int shft)  
v8int32 lsrs(v8acc48 a, int shft)
```

- AIE API

```
aie::accum<acc48, 8> acc  
aie::vector<int16, 8> v = acc.to_vector<int16>(shift)
```

Shift-Round-Saturate (srs) Code Example

```
int16_t a_data[8] = { 5000, 192, 0,0,0,0,0,0};  
int16_t b_data[8] = { 4000, 5, 0,0,0,0,0,0};
```

```
v8int16 *a = (v8int16 *)a_data;  
v8int16 *b = (v8int16 *)b_data;
```

Setting rounding and saturation

```
set_rnd(); set_sat();  
v8int16 r_s = srs(mul(*a,*b), 8); // = {32767=2^15-1 saturation to 16 bit signed  
// 4=[3.75] round to nearest integer}
```

Output for 1st
and 2nd element
of the array

$$\frac{5000 \cdot 4000}{2^8} = 78125$$

$$\frac{192 \cdot 5}{2^8} = 3.75$$

Cleared rounding and saturation

```
clr_rnd(); clr_sat();  
v8int16 nr_ns = srs(mul(*a,*b), 8); // = {12589=mod(78125,2^15 )  
//, 3 no rounding = lose LSBs}
```

Move Vector to Accumulator: Upshift (ups) Instructions

- Upshift (ups) instructions move vectors to accumulators
- Requires an adjustment in precision
- Fixed-point Arithmetic
 - Left-shift operation aligns the decimal point between the two representations
- Shift parameter is in the range of -1 to 62, which is encoded as 0..63 in the instruction
- Rounding and saturation are enabled using configuration bits
 - When rounding is set, round to nearest
 - When rounding is cleared, function truncates

```
v8acc48 ups(v8int16 v, shift);
```

```
aie::vector<int16, 8> v;  
aie::accum<acc48, 8> acc;  
acc.from_vector(v, shift);
```

Naming Conventions: srs and ups

srs

Function Name	Conversion Type
bsrs	acc48 → int8
ubsrs	acc48 → uint8
srs	acc48 → int16 acc80 → int32
lsrs	acc48 → int32 acc80 → int64

ups

Function Name	Conversion Type
ups	int8 → acc48 int16 → acc48 int32 → acc80
lups	int32 → acc48 int64 → acc80

Standard srs	Interleaved srs_ilv
o0 = srs(acc0)	o0 = srs_ilv (acc0)
o1 = srs (acc1)	o1 = srs_ilv (acc4)
o2 = srs (acc2)	o2 = srs_ilv (acc1)
o3 = srs (acc3)	o3 = srs_ilv (acc5)
o4 = srs (acc4)	o4 = srs_ilv (acc2)
o5 = srs (acc5)	o5 = srs_ilv (acc6)
o6 = srs (acc6)	o6 = srs_ilv (acc3)
o7 = srs (acc7)	o7 = srs_ilv (acc7)

List of Intrinsics

Shift-Round-Saturate Operations	Upshift Operations
v16int8 bsrs(v16acc48 a, int shift)	v8acc48 ups(v8int16 a, int shft)
v16uint8 ubsrs(v16acc48 a, int shift)	v4cacc48 ups(v4cint16 a, int shft)
v8int16 srs(v8acc48 a, int shift)	v8acc48 ups(v8int16 a, int shft)
v4cint16 srs(v4cacc48 a, int shift)	v4cacc48 ups(v4cint16 a, int shft)
v16int16 srs(v16acc48 a, int shift)	v16acc48 ups(v4int32 a, int shft)
v8cint16 srs(v8cacc48 a, int shift)	v8cacc48 ups(v2cint32 a, int shft)
v4int32 srs(v4acc80 a, int shift)	v4acc80 ups(v4int32 a, int shft)
v2cint32 srs(v2cacc80, int shift)	v2cacc80 ups(v2cint32 a, int shft)
v8int32 srs(v8acc80 a, int shift)	v8acc80 ups(v8int32 a, int shft)
v4cint32 srs(v4cacc80 a, int shift)	v4cacc80 ups(v4cint32 a, int shft)

srs and ups intrinsics may map to more than one assembly instruction

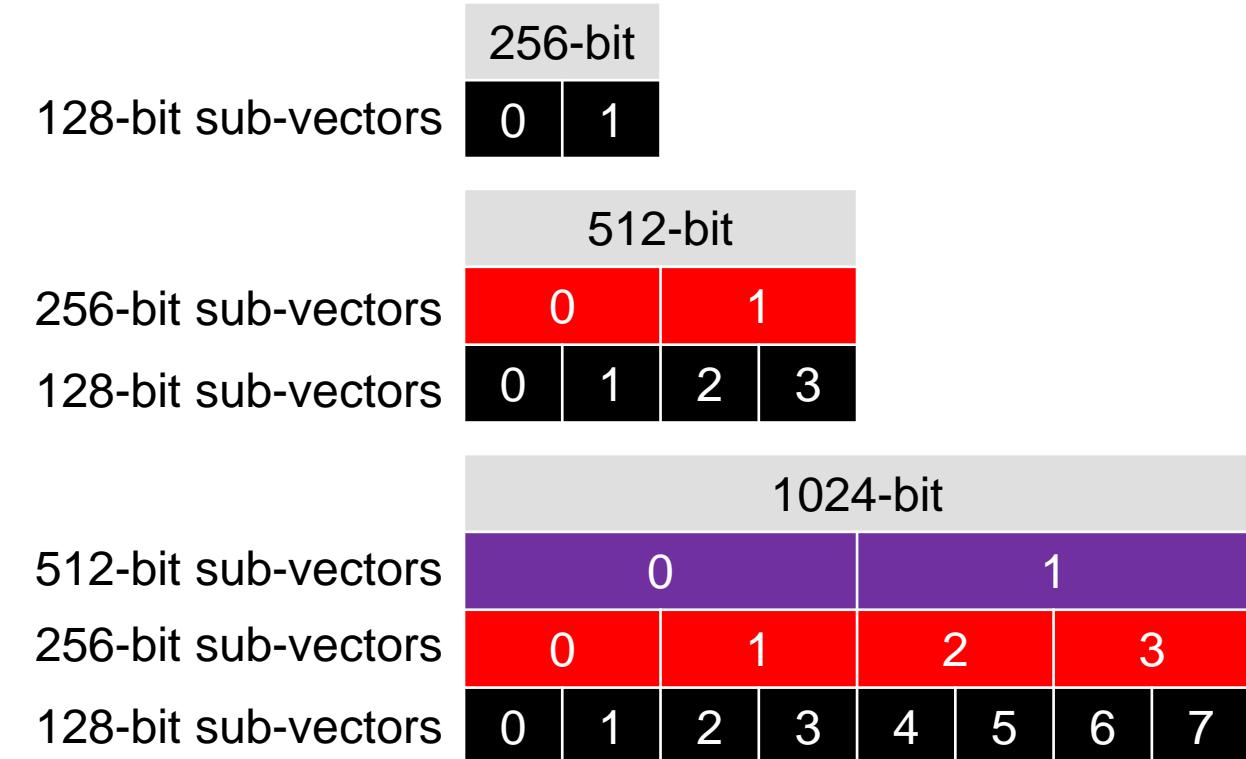
Sub-vector Manipulation

Vector

- A vector can be seen as the concatenation of smaller sub-vectors
- Sub-vector size: 128-bit, 256-bit, 512-bit
- Each sub-vector is identified with an index starting from 0

Manipulation of a Sub-vector

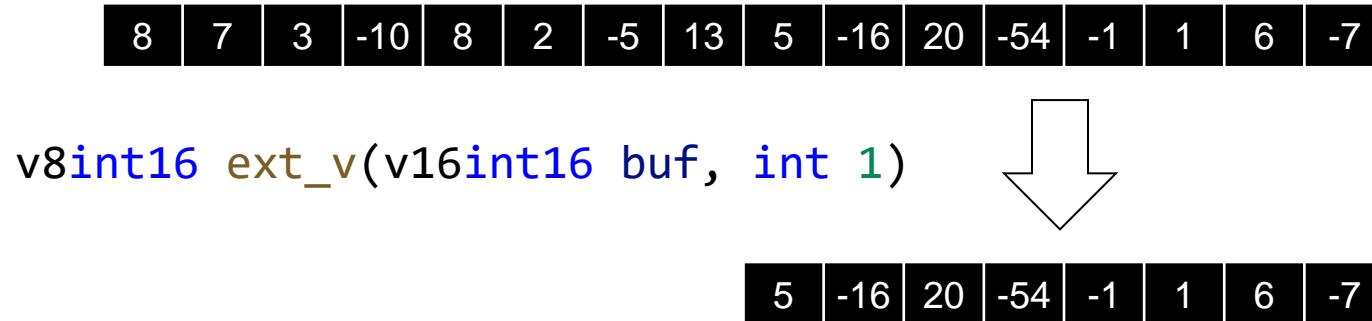
- A sub-vector can be extracted using the method **extract**
- A sub-vector can be updated using the method **insert**



Accumulators can also be manipulated with extract and insert

Extract from Vector

- Extract portions of a vector
 - `ext_v()`
 - `ext_w()`
 - `ext_x()`
- Example:
 - Extract 8 lanes



Extract from Accumulator

- Extract portions of an accumulator
 - ext_hi: upper half
 - ext_lo: lower half
- Examples

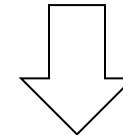
```
v8acc48 ext_lo(v16acc48 buf)  
v4acc80 ext_hi(v8acc80 buf)
```

Extraction – AI Engine API

- Same function regardless underlying bits
 - `.extract<lanes>(index)`

8	7	3	-10	8	2	-5	13	5	-16	20	-54	-1	1	6	-7
---	---	---	-----	---	---	----	----	---	-----	----	-----	----	---	---	----

```
aie::vector<int16, 16> vout;  
aie::vector<int16, 8> v = vout.extract<8>(1);
```



5	-16	20	-54	-1	1	6	-7
---	-----	----	-----	----	---	---	----

Update

- Vector update intrinsic functions allow for the substitution of the lanes within a vector value
- Update
 - upd_lo: lower half
 - upd_hi: upper half
- Update
 - upd_v(buf, N, val): Allows the replacement of a 128-bit subset of a vector
 - upd_w(buf, N, val): Allows the replacement of a 256-bit subset of a vector
 - upd_x(buf, N, val): Allows the replacement of a 512-bit subset of a vector
 - buf is the vector to update
 - N is the sub-vector to update on buf
 - val is the new sub-vector

Update (upd_lo / upd_hi) - Example

- Update the lower half of hl with l

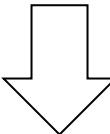
hl

8	7	3	-10	8	2	-5	13	5	-16	20	-54	-1	1	6	-7
---	---	---	-----	---	---	----	----	---	-----	----	-----	----	---	---	----

l

35	4	-58	12	68	-3	-44	12
----	---	-----	----	----	----	-----	----

v16int16 upd_lo(v16int16 hl, v8int16 l)



35	4	-58	12	68	-3	-44	12	5	-16	20	-54	-1	1	6	-7
----	---	-----	----	----	----	-----	----	---	-----	----	-----	----	---	---	----

Update (Insert) – AI Engine API

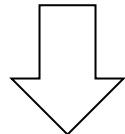
- Update the lower half of $h1$ with l and m

$h1$ 8 | 7 | 3 | -10 | 8 | 2 | -5 | 13 | 5 | -16 | 20 | -54 | -1 | 1 | 6 | -7

l 35 | 4 | -58 | 12

m 68 | -3 | -44 | 12

`h1.insert(0, concat(l,m))`



35 | 4 | -58 | 12 | 68 | -3 | -44 | 12 | 5 | -16 | 20 | -54 | -1 | 1 | 6 | -7

Fine-grain Data Selection – Shuffle

- Builds the output vector as a selection of the input vector lanes

```
v16int32 shuffle16 ( v16int32 xbuff, int xstart, unsigned int xoffsets, unsigned int xoffsets_hi)
```

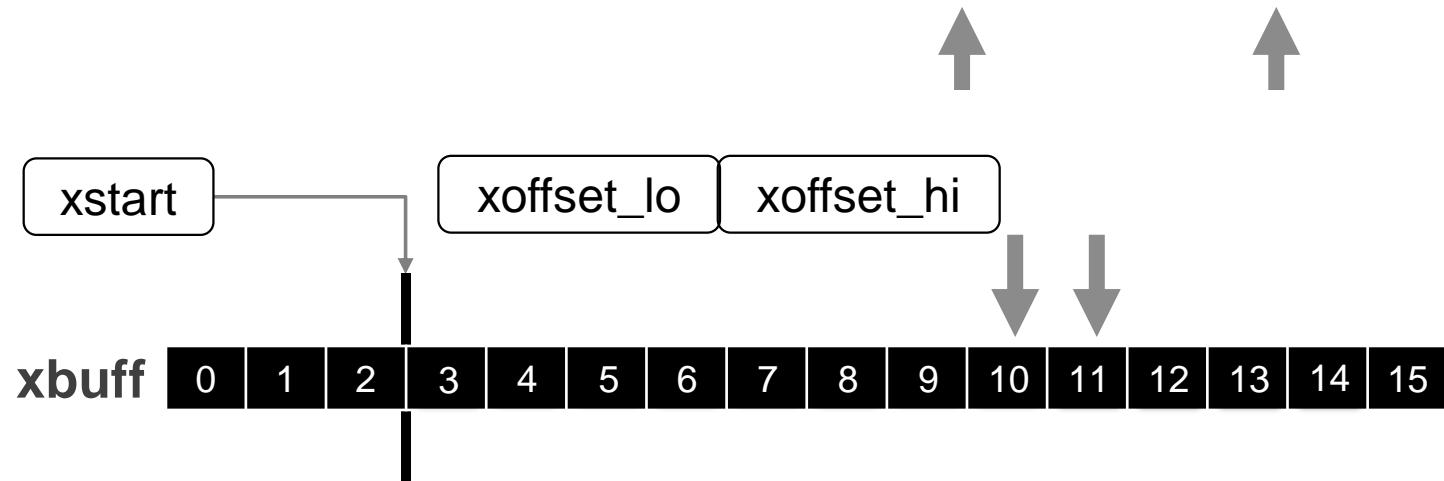
Input/Output	Type	Comments
return	v16int32	Return value of each lane is the result of a shuffle between lanes of xbuff where the result of lane 0 goes to lane 0 of the output.
xbuff	v16int32	Input buffer of 16 elements with 32-bit precision.
xstart	int	Starting position offset applied to all lanes of input from X buffer.
xoffsets	unsigned int	4b offset for each lane in the xbuffer. LSB applies to first lane.
xoffsets_hi	unsigned int	4b offset for each lane in the xbuffer. LSB applies to eighth lane.

Takes advantage of the data selection and permute logic of the AIE vector processor architecture

Shuffle Example on v16int32

v16int32 32b 32b

```
shuffle16(buf, 3, 0x01234567, 0x89ABBA98)
```



Resulting index is
xstart + offset
mod length(xbuff)

Data Shuffling – AI Engine API

- Data shuffling is performed through three different schemes:
- Returns a vector whose content are the same but shifted down/up by n
- Elements do not wrap, new elements are undefined

```
shuffle_down(vec, n);  
shuffle_up(vec, n);
```

- In this version, new elements are filled from second vector

```
shuffle_down_fill(vec, fill, n);  
shuffle_up_fill(vec, fill, n);
```

- In this version, elements wrap around

```
shuffle_down_rotate(vec, n);  
shuffle_up_rotate(vec, n);
```

Multiplication Operations

- Multiply (mul)
 - Performs element-by-element multiplication and returns an accumulator type
- Multiply Accumulate (mac)
 - Performs element-by-element multiplication and accumulation
 - Current value of the accumulator must be provided, and it returns an accumulator type
- Multiply Subtract (msc)
 - Performs element-by-element multiplication and subtract
 - Current value of the accumulator must be provided, and it returns an accumulator type

Multiply (mul) - Examples

- Multiply two vectors and place the result in an accumulator

	Multiplication Operations	Cycles	Utilization
32 Bit	v4acc80 mul (v4int32 a, v4int32 b)	1	50%
	v8acc80 mul (v8int32 a, v8int32 b)	1	100%
	v2cacc80 mul (v2cint32 a, v2cint32 b)	1	100%
	v4cacc80 mul (v4cint32 a, v4cint32 b)	2	100%
16 Bit	v8acc48 mul (v8int16 a, v8int16 b)	1	25%
	v16acc48 mul (v16int16 a, v16int16 b)	1	50%
	v4cacc48 mul (v4cint16 a, v4cint16 b)	1	50%
	v8cacc48 mul (v8cint16 a, v8cint16 b)	1	100%

- Multiply Accumulate/Subtract variants are very similar
 - Accumulator register is provided

Multiply (mac/msc) - Examples

- Multiply Accumulate/Subtract variants are very similar to mul
 - Accumulator register is provided

v8acc48 mac (v8acc48 a, v8int16 x, v8int16 y)

$\rightarrow a + x \cdot y$

v8acc48 msc (v8acc48 a, v8int16 x, v8int16 y)

$\rightarrow a - x \cdot y$

Advanced MAC Intrinsics

- Operations are described by using “lanes” and “columns”
- Lanes
 - Outputs generated in parallel
- Columns
 - Number of multiplications per output lane
 - Each of the multiplication results being added together
 - Multiplications per lane is determined by the precision of data and coefficients

Advanced MAC Intrinsics - Example

- **mac8** with 16-bit data and 16-bit coefficients
 - 8 lanes
 - 4 columns
- 32 MACs/cycle
- **mul4** with 32-bit data and coefficients
 - 4 lanes
 - 2 columns
- 8 MACs/cycle

```
acc0 += z00*x00 + z01*x01 + z02*x02 + z03*x03  
acc1 += z10*x10 + z11*x11 + z12*x12 + z13*x13  
acc2 += z20*x20 + z21*x21 + z22*x22 + z23*x23  
acc3 += z30*x30 + z31*x31 + z32*x32 + z33*x33  
acc4 += z40*x40 + z41*x41 + z42*x42 + z43*x43  
acc5 += z50*x50 + z51*x51 + z52*x52 + z53*x53  
acc6 += z60*x60 + z61*x61 + z62*x62 + z63*x63  
acc7 += z70*x70 + z71*x71 + z72*x72 + z73*x73
```

```
acc0 += z00*x00 + z01*x01  
acc1 += z10*x10 + z11*x11  
acc2 += z20*x20 + z21*x21  
acc3 += z30*x30 + z31*x31
```

Application-specific Intrinsics

- Intrinsics designed to assist with the implementation of a specific application
 - Digital pre-distortion
 - Direct digital synthesis (DDS) interpolation
 - Fast Fourier Transform (FFT)
 - Peak cancellation crest factor reduction (PC-CFR)

Summary

- Intrinsic functions target the AIE Vector processor
- Three types of vector management operations
 - Load and store
 - Element conversion
 - Lane insertion and extraction
- Types of multiplication functions include mul, mac, and msc
- Application-specific functions are also available



Window and Streaming Data APIs

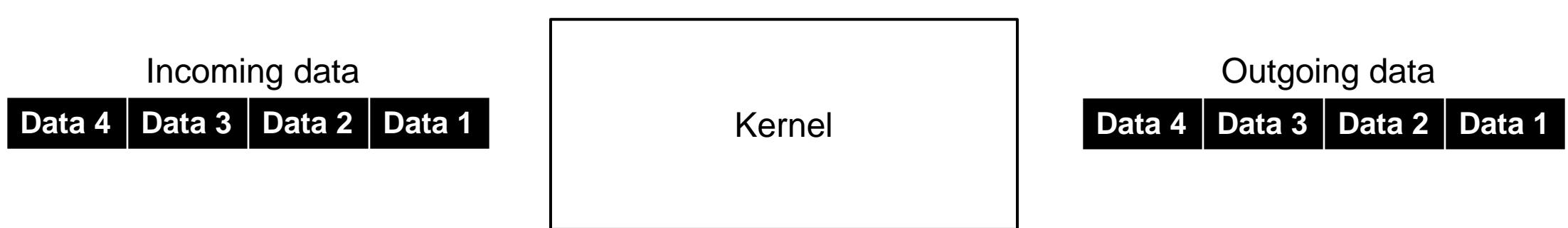
Objectives

After completing this module, you will be able to:

- Describe the window and streaming APIs
- Identify the different window operations for kernels
- Describe using overlapping data for the continuous stream of data
- Describe various data movement use cases

Dataflow Kernels

- Data streams are an infinity long sequences of typed values
- Data streams can be broken into blocks
- Kernel consumes input blocks of data
- Kernel produces output blocks of data



Data Movement APIs

- Support both vector and scalar, signed and unsigned data
 - AI Engine supports unsigned integer vector arithmetic only for the 8-bit data types
- The data movement can be
 - Window-based access
 - Stream-based access

Window-based Access

Input Window

- Kernel's view of incoming blocks of data
- Defined by type

Output Window

- Kernel's view of outgoing blocks of data
- Defined by type

```
input_window<uint16> myFirstWindow;
```

```
output_window<int32> myOtherWindow;
```

Incoming data

Data 4	Data 3	Data 2	Data 1
--------	--------	--------	--------

Kernel

Outgoing data

Data 4	Data 3	Data 2	Data 1
--------	--------	--------	--------

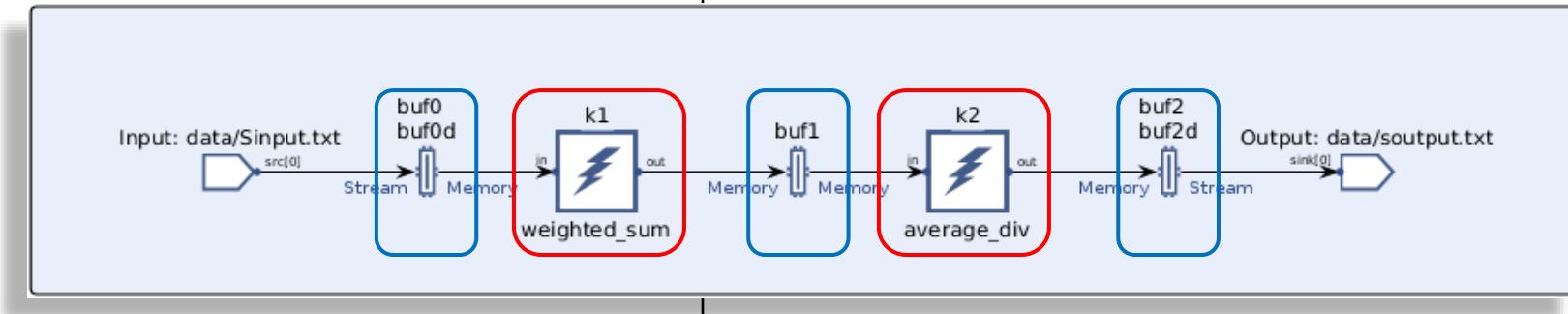
Window-based Access – Adaptive Dataflow Graph

```

class FirstGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio pl_in;
    output_plio pl_out;

    GraphScalar() {
        //create kernels
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        //create nets to connect kernels and IO ports
        connect<window<1024>> net0 (pl_in.out[0], k1.in[0]);
        connect<window<1024>> net1 (k1.out[0], k2.in[0]);
        connect<window<1024>> net2 (k2.out[0], pl_out.in[0]);
    }
};

```



More details about ADF Graphs in a few slides

Window-access – Example: 8 sample weighted sum

```
void weighted_sum(input_window<int32> *in,
                  output_window<int32> *out){
    for(unsigned int i=0; i< WLEN; i++){
        int32 val;
        int32 wsum = 0;
        for(unsigned int j=0; j< 8; j++){
            window_readincr(in, val);
            wsum = wsum + (j * val);
        }
        window_writeincr(out, wsum);
        window_decr(in, 7);
    }
}
```

C function is using the special window type parameters as pointers

Inside this kernel function, a C API is used to access data in the window

Scalar example

Supported Window Data Types

Input Window Types	Output Window Types
input_window<int8>	output_window<int8>
input_window<int16>	output_window<int16>
input_window<int32>	output_window<int32>
input_window<int64>	output_window<int64>
input_window<uint8>	output_window<uint8>
input_window<uint16>	output_window<uint16>
input_window<uint32>	output_window<uint32>
input_window<uint64>	output_window<uint64>
input_window<cint16>	output_window<cint16>
input_window<cint32>	output_window<cint32>
input_window<float>	output_window<float>
input_window<cfloat>	output_window<cfloat>

Window-access functions to read and write

Description	Input Windows	Output Windows
Reads data from an input window	window_read()	window_write()
Moves the current pointer position forward	window_incr()	window_incr()
Reads and advances an input window	window_readincr()	window_writeincr()
Moves current read or write position backward	window_decr()	window_decr()

Full List of APIs in [UG1076](#): Versal ACAP AI Engine Programming Environment

Window-access operations – move current position forward

Input Window Type	Output Window Type
void window_incr(<input_window_type> *w, int count);	void window_incr(<output_window_type> *w, int count);
void window_incr_v4(<input_window_type> *w, int count);	void window_incr_v4(<output_window_type> *w, int count);
void window_incr_v8(<input_window_type> *w, int count);	void window_incr_v8(<output_window_type> *w, int count);
void window_incr_v16(<input_window_type> *w, int count);	void window_incr_v16(<output_window_type> *w, int count);
void window_incr_v32(<input_window_type> *w, int count);	void window_incr_v32(<output_window_type> *w, int count);
void window_incr_v64(<input_window_type> *w, int count);	void window_incr_v64(<output_window_type> *w, int count);

Full List of APIs in [UG1076: Versal ACAP AI Engine Programming Environment](#)

Window-access operations – move current position backward

Input Window Type	Output Window Type
void window_decr(<input_window_type> *w, int count);	void window_decr(<output_window_type> *w, int count);
void window_decr_v4(<input_window_type> *w, int count);	void window_decr_v4(<output_window_type> *w, int count);
void window_decr_v8(<input_window_type> *w, int count);	void window_decr_v8(<output_window_type> *w, int count);
void window_decr_v16(<input_window_type> *w, int count);	void window_decr_v16(<output_window_type> *w, int count);
void window_decr_v32(<input_window_type> *w, int count);	void window_decr_v32(<output_window_type> *w, int count);
void window_decr_v64(<input_window_type> *w, int count);	void window_decr_v64(<output_window_type> *w, int count);

Full List of APIs in [UG1076: Versal ACAP AI Engine Programming Environment](#)

Typical Use from a Kernel

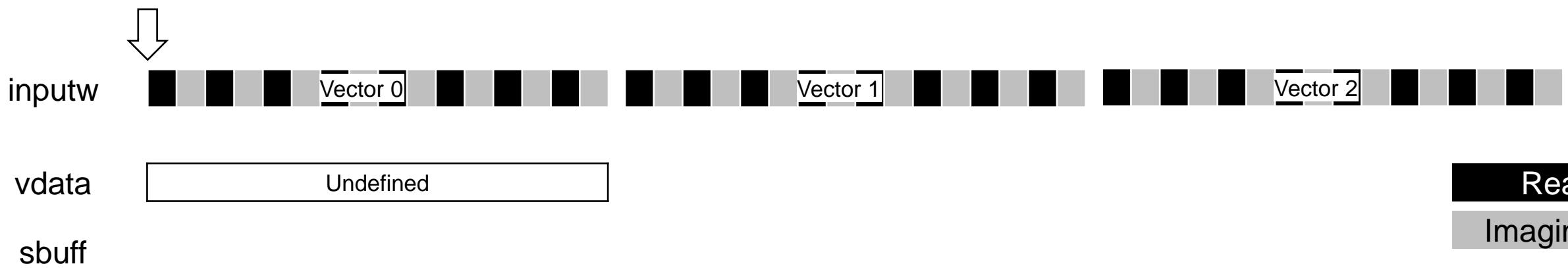
```
void kernel(input_window<uint16> *inputw,  
           output_window<uint16> *outputw) {  
    v8uint16 vdata;  
    window_readincr(inputw, vdata);  
    sbuf = upd_w(sbuf, 0, vdata);  
    // Do math with sbuf  
    window_readincr(inputw, vdata);  
    sbuf = upd_w(sbuf, 1, vdata);  
    ...  
}
```

Window data type pointers

- input_window<uint16>
- output_window<uint16>

Vector data type

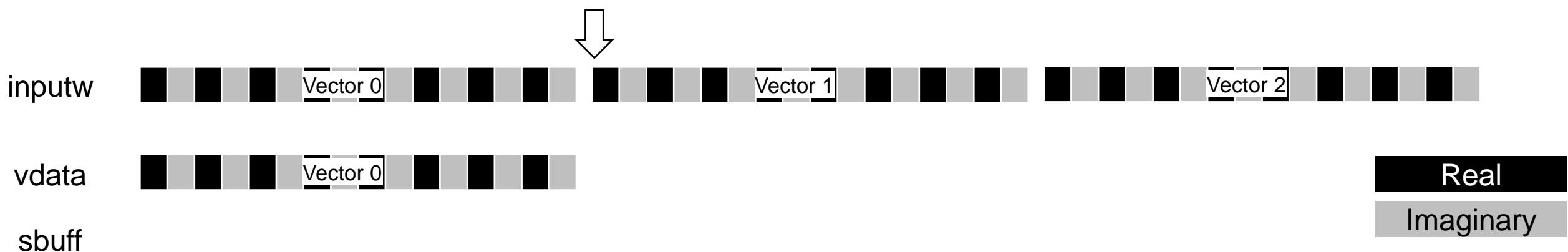
- v8uint16 vdata



Typical Use from a Kernel

```
void kernel(input_window<uint16> *inputw,  
           output_window<uint16> *outputw) {  
    v8uint16 vdata;  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 0, vdata);  
    // Do math with sbuff  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 1, vdata);  
    ...  
}
```

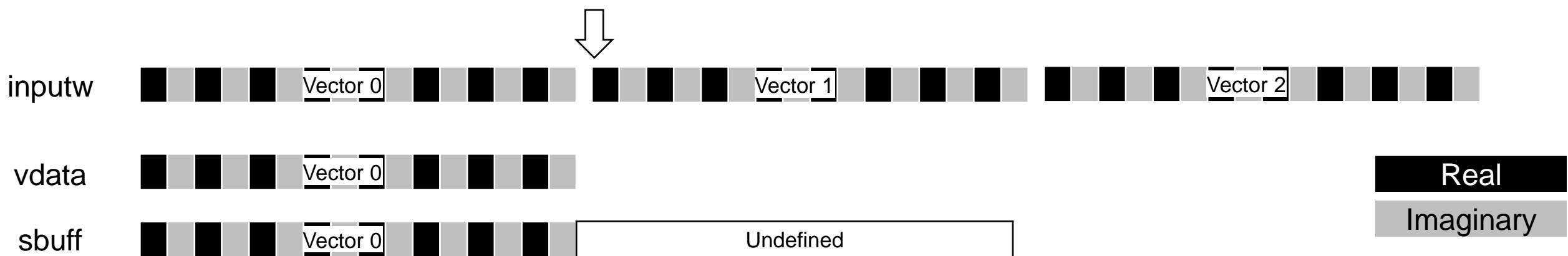
Read eight complex elements
Advance the pointer to next vector



Typical Use from a Kernel

```
void kernel(input_window<uint16> *inputw,  
           output_window<uint16> *outputw) {  
    v8uint16 vdata;  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 0, vdata);  
    // Do math with sbuff  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 1, vdata);  
    ...  
}
```

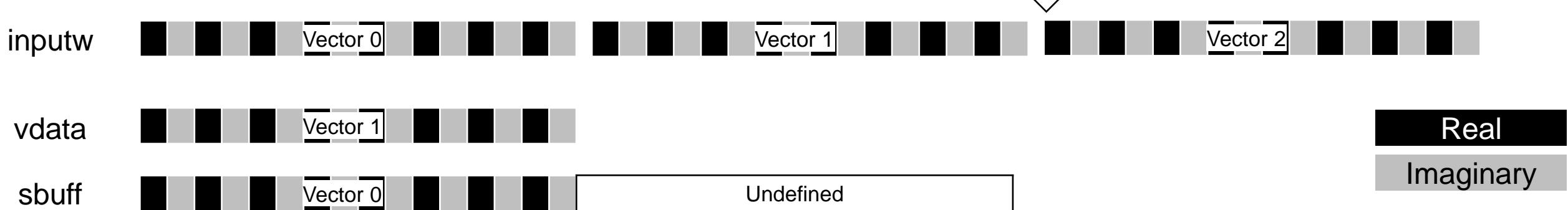
Update lower index of sbuff with Vector 0



Typical Use from a Kernel

```
void kernel(input_window<uint16> *inputw,  
           output_window<uint16> *outputw) {  
    v8uint16 vdata;  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 0, vdata);  
    // Do math with sbuff  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 1, vdata);  
    ...  
}
```

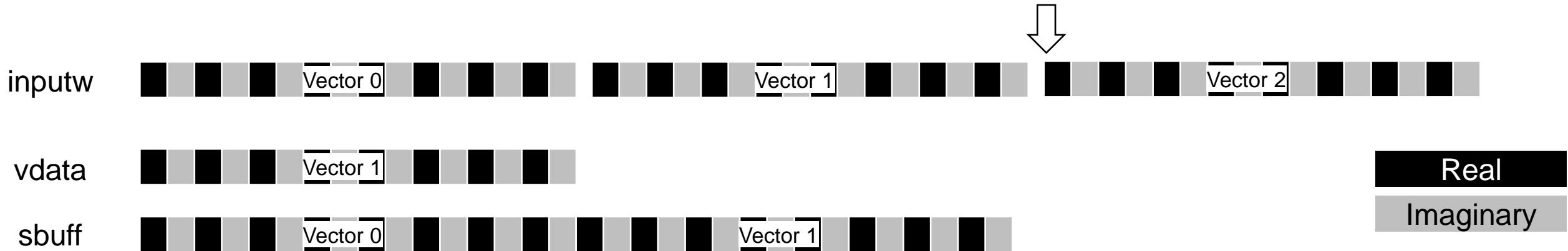
Read eight complex elements
Advance the pointer to next vector



Typical Use from a Kernel

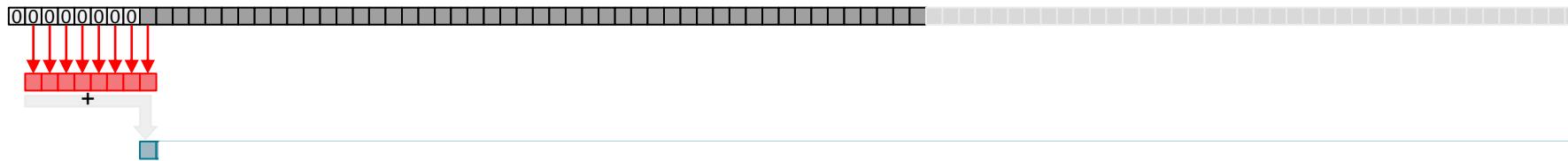
```
void kernel(input_window<uint16> *inputw,  
           output_window<uint16> *outputw) {  
    v8uint16 vdata;  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 0, vdata);  
    // Do math with sbuff  
    window_readincr(inputw, vdata);  
    sbuff = upd_w(sbuff, 1, vdata);  
    ...  
}
```

Update upper index of sbuff with Vector 1



Window Access to Data – Overlap

- Filtering application needs a continuous stream of data



- When data is **decomposed to multiple frames**:

- First block of data is prepended with zeros



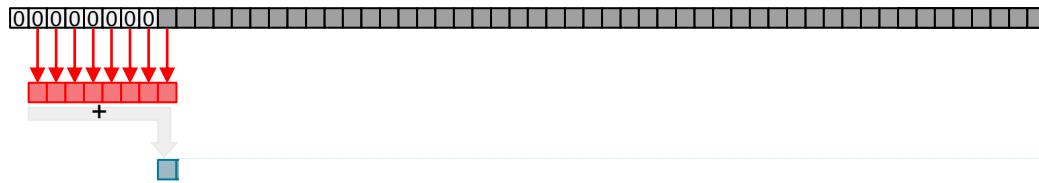
- Second block of data is also **prepended with zeros**



ERROR !

Window Access to Data – Overlap

- First execution is correct

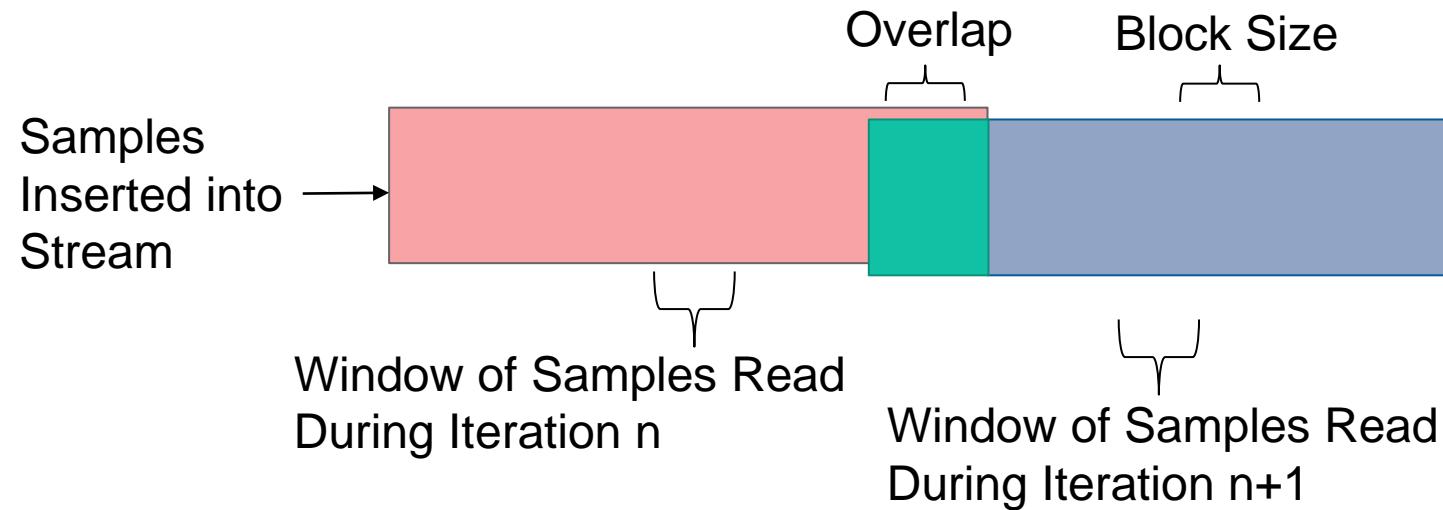


- Prior to second execution, the **last samples** of first input have to be **copied**



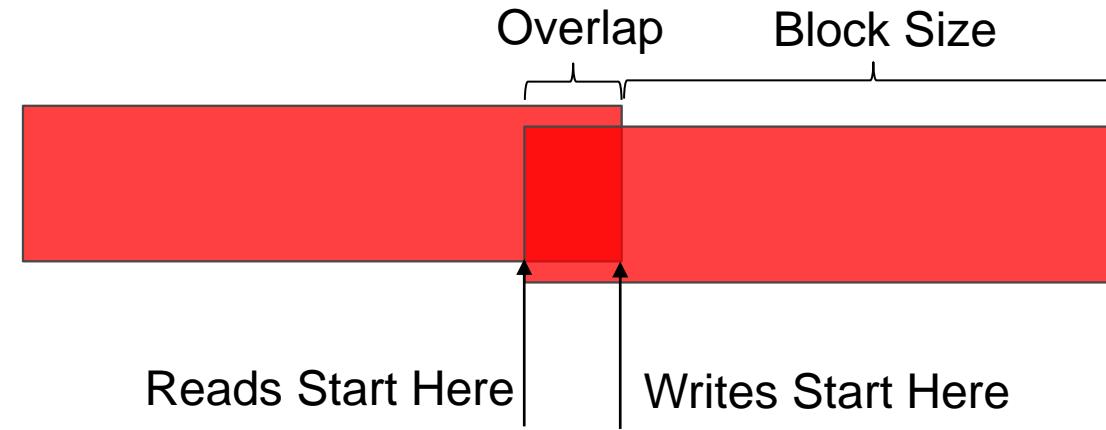
Window Access to Data – Overlap

- Requires read access to previous data samples in addition to new data samples
- Overlap section is automatically copied in between two kernel calls
- This overlap is sometimes referred as margin



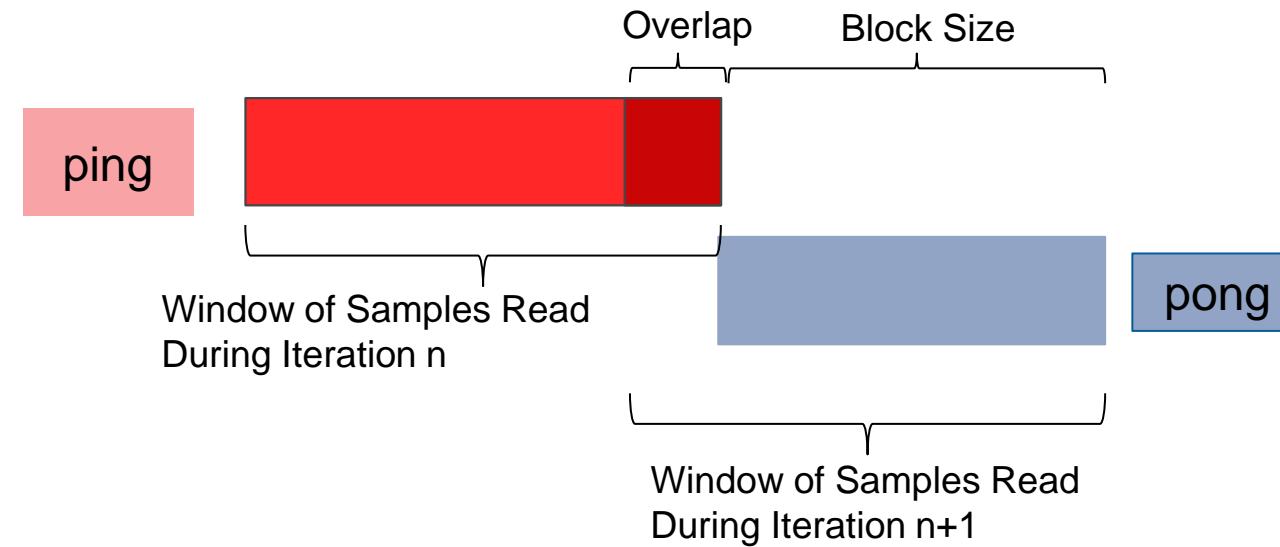
Window Access to Data – Specifying at Kernel Level

- Kernels are provided with a start pointer for both inputs and outputs
- Helper functions are available to read and write data and manipulate the pointer



Window Access to Data – Handling Overlap

- If producer and consumer are on different AI Engines, ping-pong buffers are used



Summary of Kernel and Window Interactions

- When a kernel starts, its input and output windows have always been set correctly

Input windows start on overlap
and move to new data

API functions hide details of
addressing underlying
structure

Output windows start in
position to write new data

Incoming data

Data 4 | Data 3 | Data 2 | Data 1

Kernels

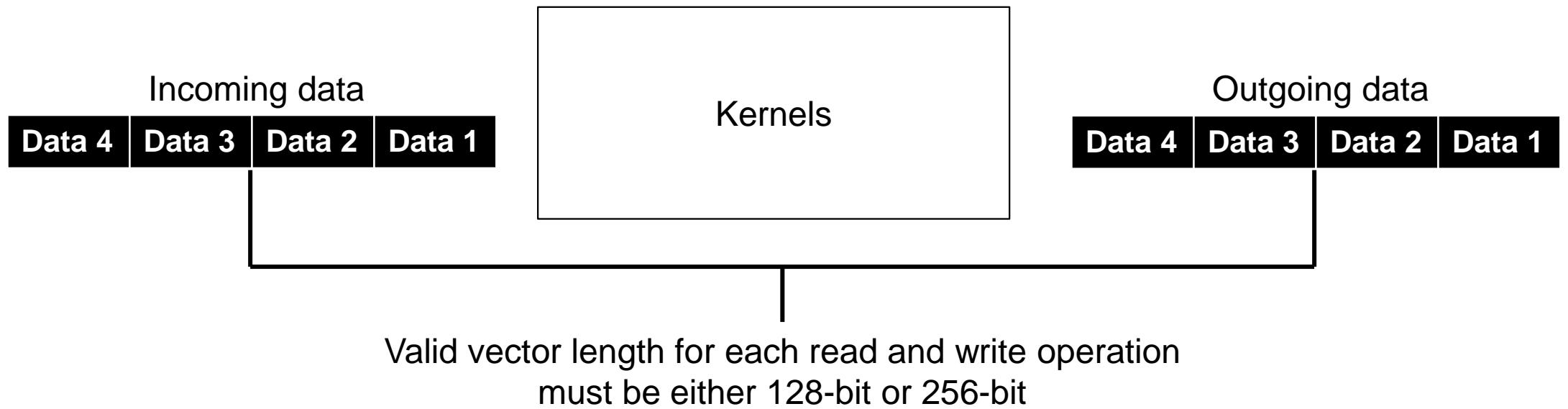
Outgoing data

Data 4 | Data 3 | Data 2 | Data 1

AIE Compiler still generates
efficient code

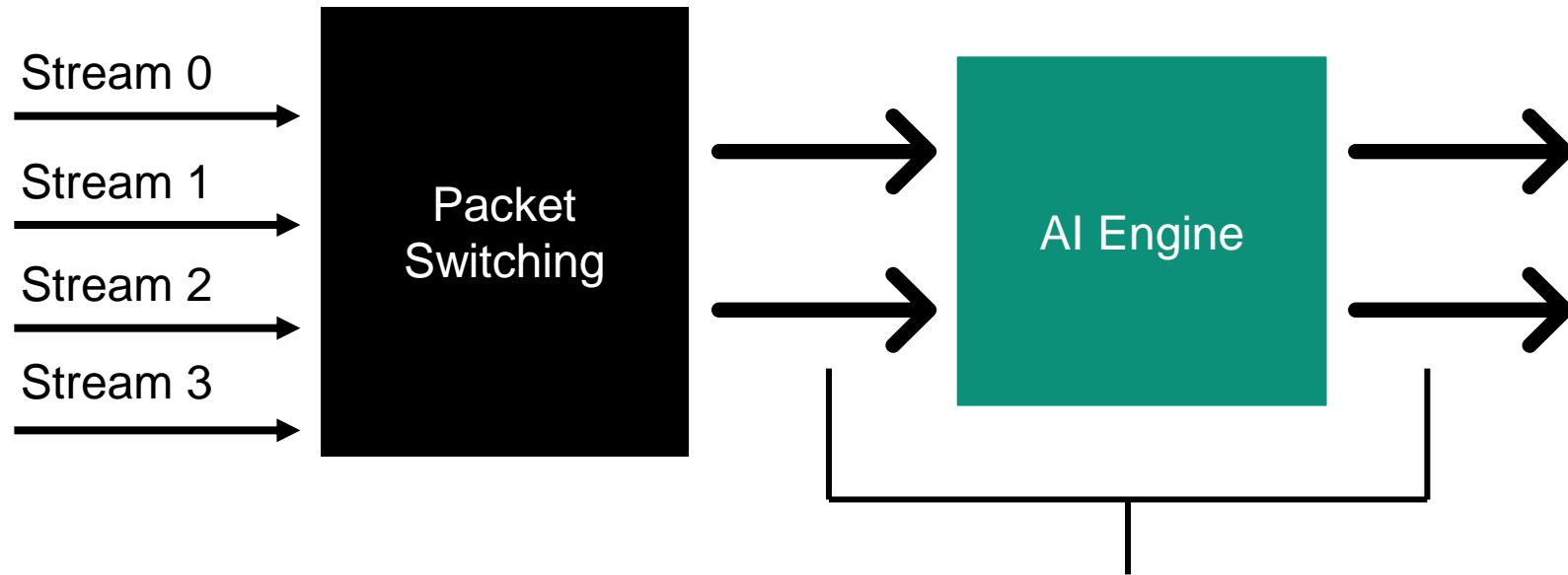
Window in Data Communication

- Windows of input data and output buffer are locked for kernels before they are executed
- Kernel can perform random access within a window of data



Stream in Data Communication

- Kernels perform sequential access
 - Blocking or non-blocking
- Cascade stream only supports blocking access



Valid vector length for each read
and write operation
must be either **32-bit or 128-bit**

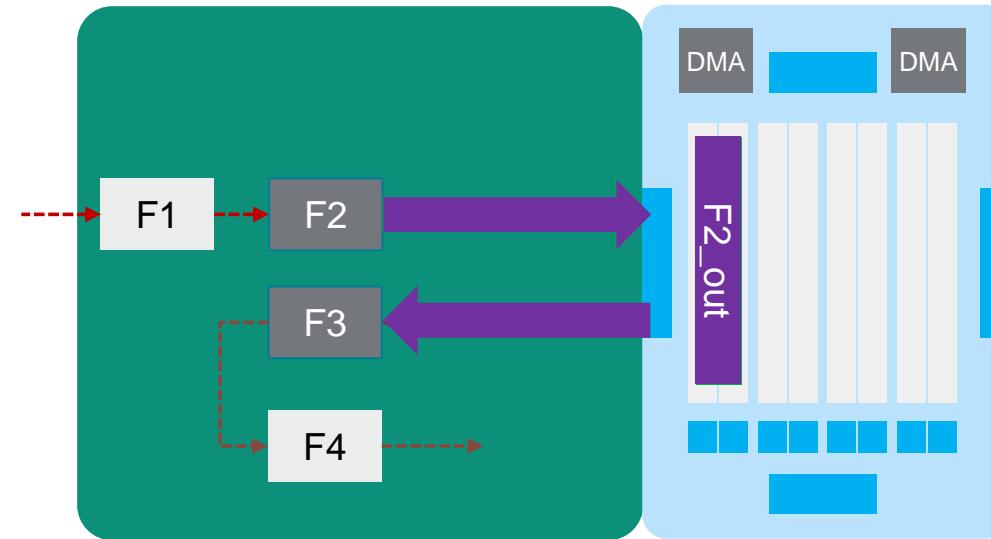
Window vs. Stream in Data Communication

Connection	Margin	Packet Switching	Back Pressure	Lock	Max. Throughput by VLIW (Per Cycle)	Multicast as a Source
Window	Yes	Yes ¹	No	Yes	2*256-bit load + 1*256-bit store	Broadcast
Stream	No	Yes ¹	Yes	No	2*32-bit read + 1*32-bit write or 1*32-bit read + 2*32-bit write	Yes

¹ Packet switching is only supported between AI Engine kernels and PL kernels

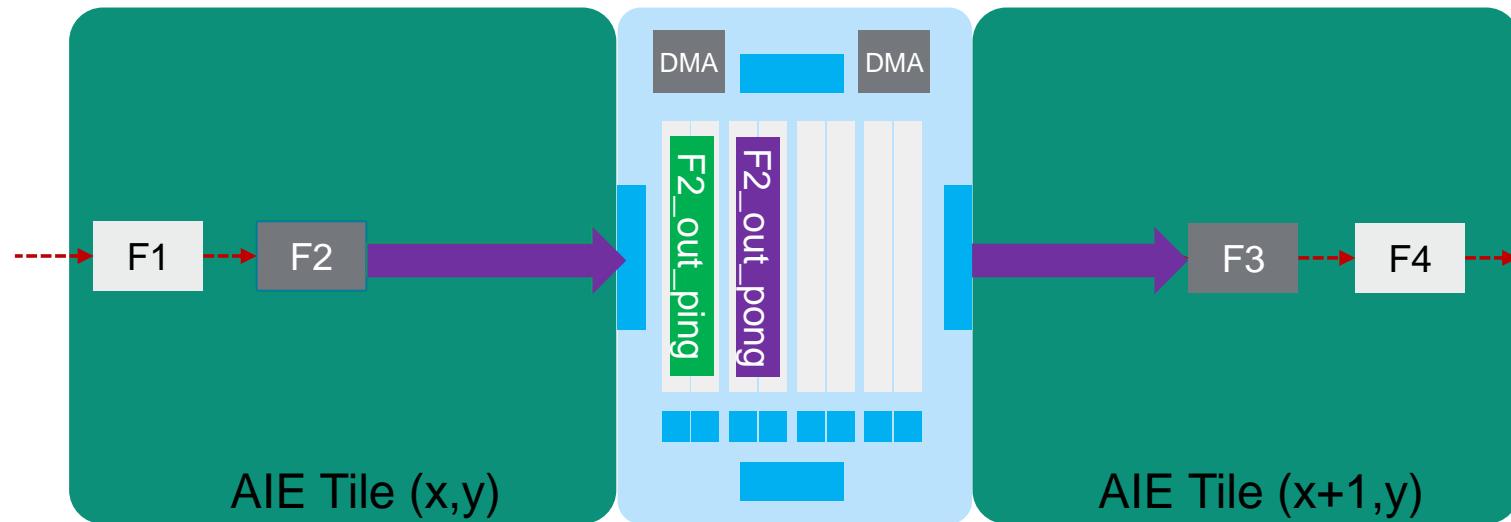
Multiple Kernel Communication on One AI Engine Tile

- Multiple kernels can fit on a single AI Engine tile
 - Depending on data rate, compute, and memory requirements
- Kernels run sequentially
 - Allowing communication via a common buffer in shared memory



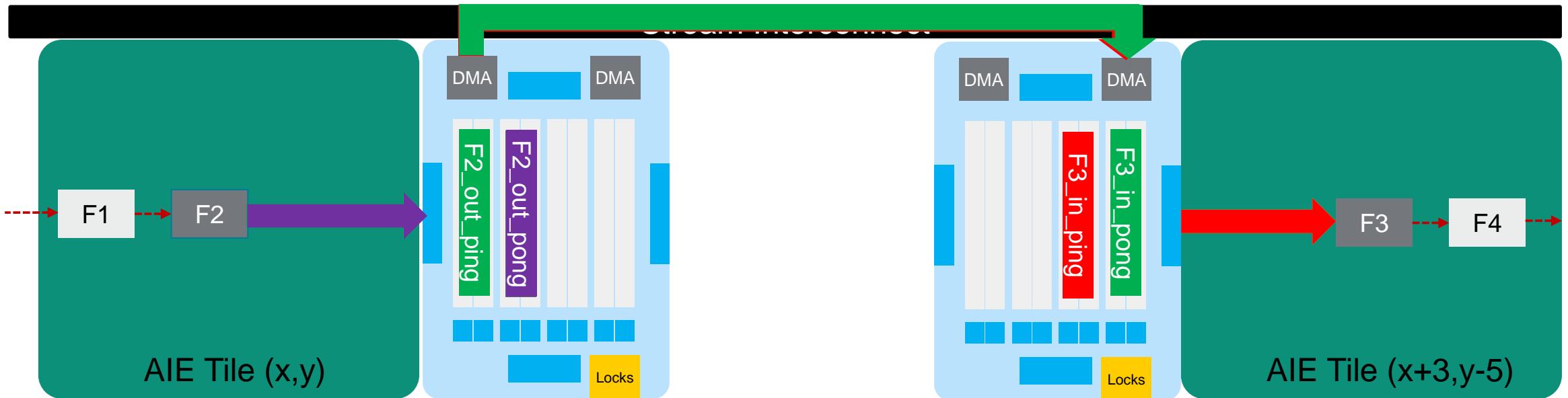
Communication Between Neighboring AI Engines

- Kernels on neighboring AI Engine can communicate via shared memory
 - While the AI Engines are working in parallel
- Communication is established using ping and pong buffers on separate memory banks
- Locks are used for synchronization
- In case of overlap between frames, data must be copied between calls
 - Overlap section is automatically copied in between two kernel calls
- DMA and stream interconnects are NOT required



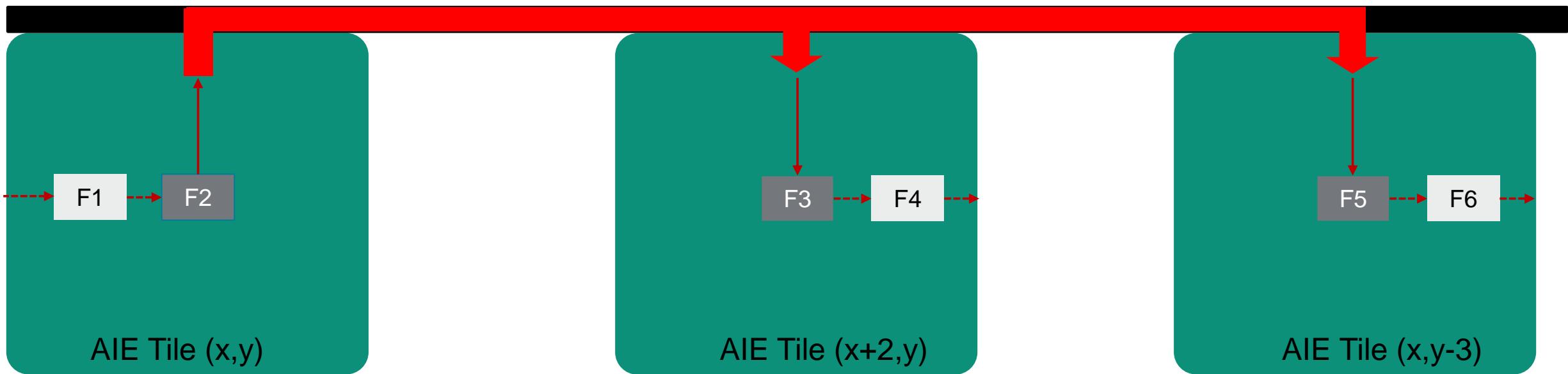
Communication via Memory Modules, DMA, and Streams

- Non-neighbor communication
 - Uses DMA and stream interconnect functionality of the AI Engine tile architecture
- Increased communication latency and double memory resources
 - Ping pong buffers for each AIE Tile
- Less bandwidth with respect to local memory



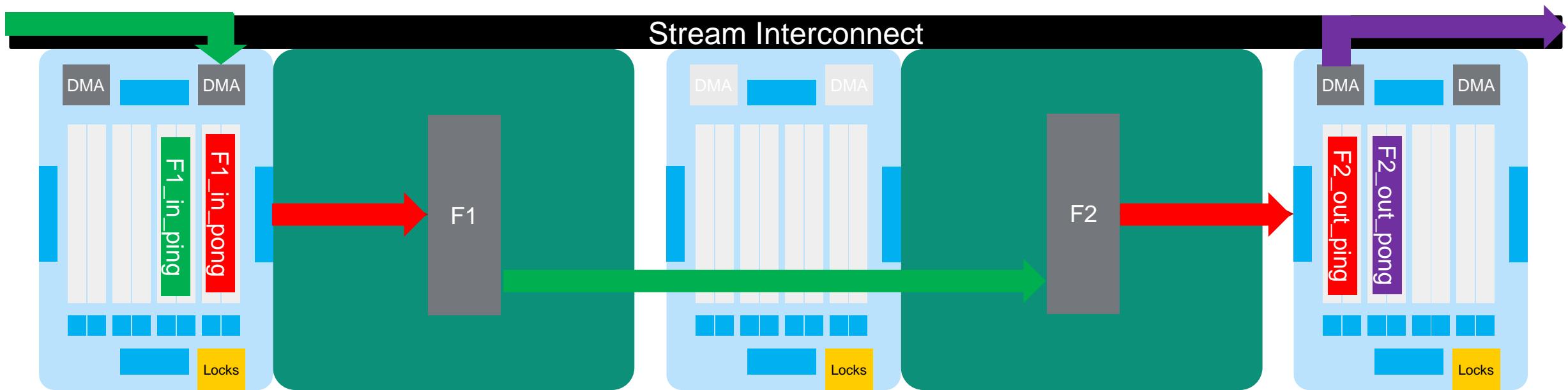
Multicast Communication Using Direct Stream Access

- F2 writes directly to stream interface
 - Broadcasts information to multiple AI Engines
- F3 and F5 read directly from stream interface



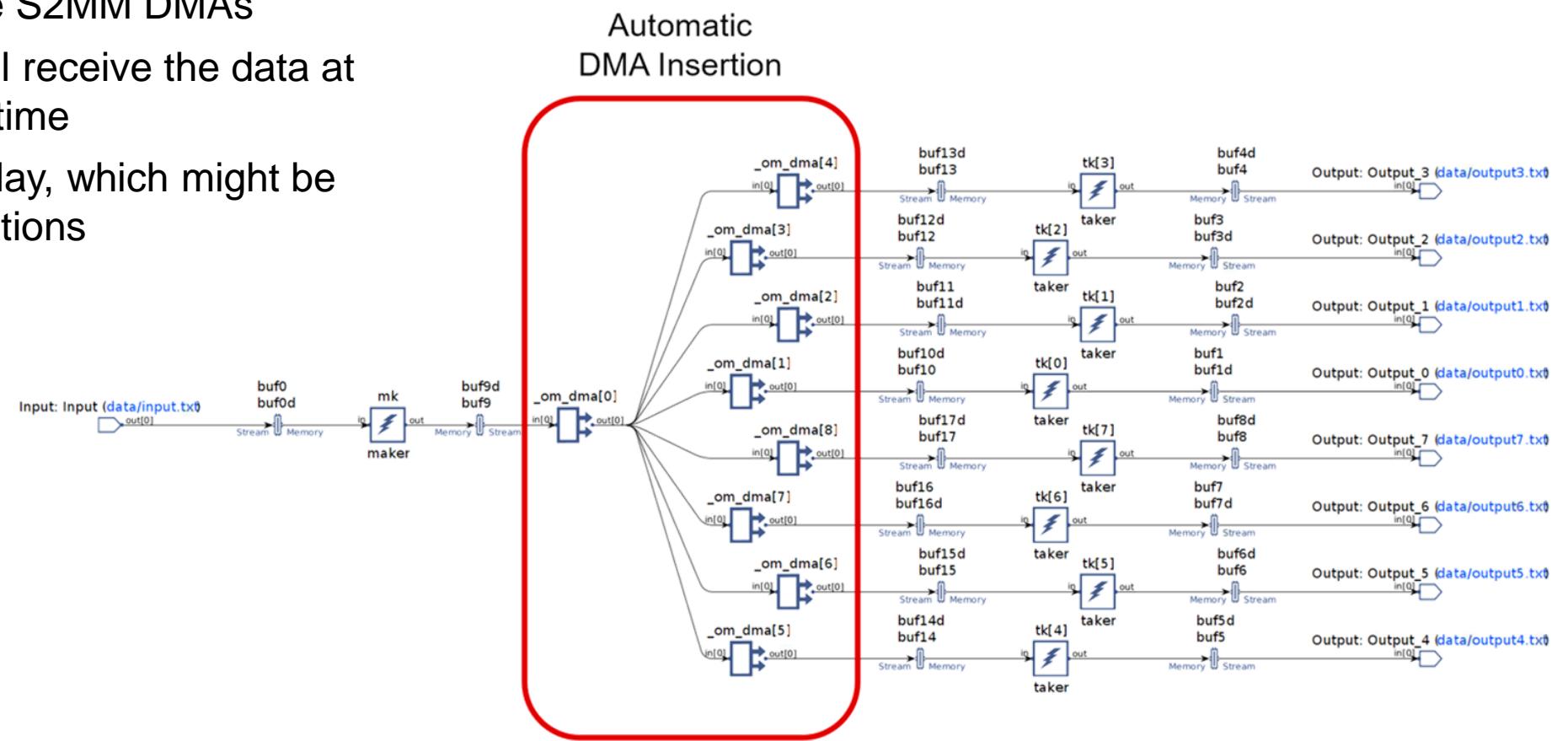
Cascade Stream

- Distribute compute workload by pipelining multiple AI Engines
- Communication can be done using cascade stream which travels from tile to tile horizontally
- Transfer accumulator values between AI Engines



Support for Broadcast Windows

- Windows can be broadcasted with automatic insertion of DMAs
- 1x MM2S and multiple S2MM DMAs
- All the destinations will receive the data at approximate at same time
- Modulo the routing delay, which might be different for all destinations



Summary

- An incoming block of data is called an input window
- An outgoing block of data is called an output window
- Various window operations are available to read and access the window data, such as
 - `window_read()`
 - `window_incr()`
 - `window_readincr()`
 - `window_decr()`
- When overlap data is required, the overlap section can be copied automatically using the available APIs
- Streams allow to access block of data sequentially



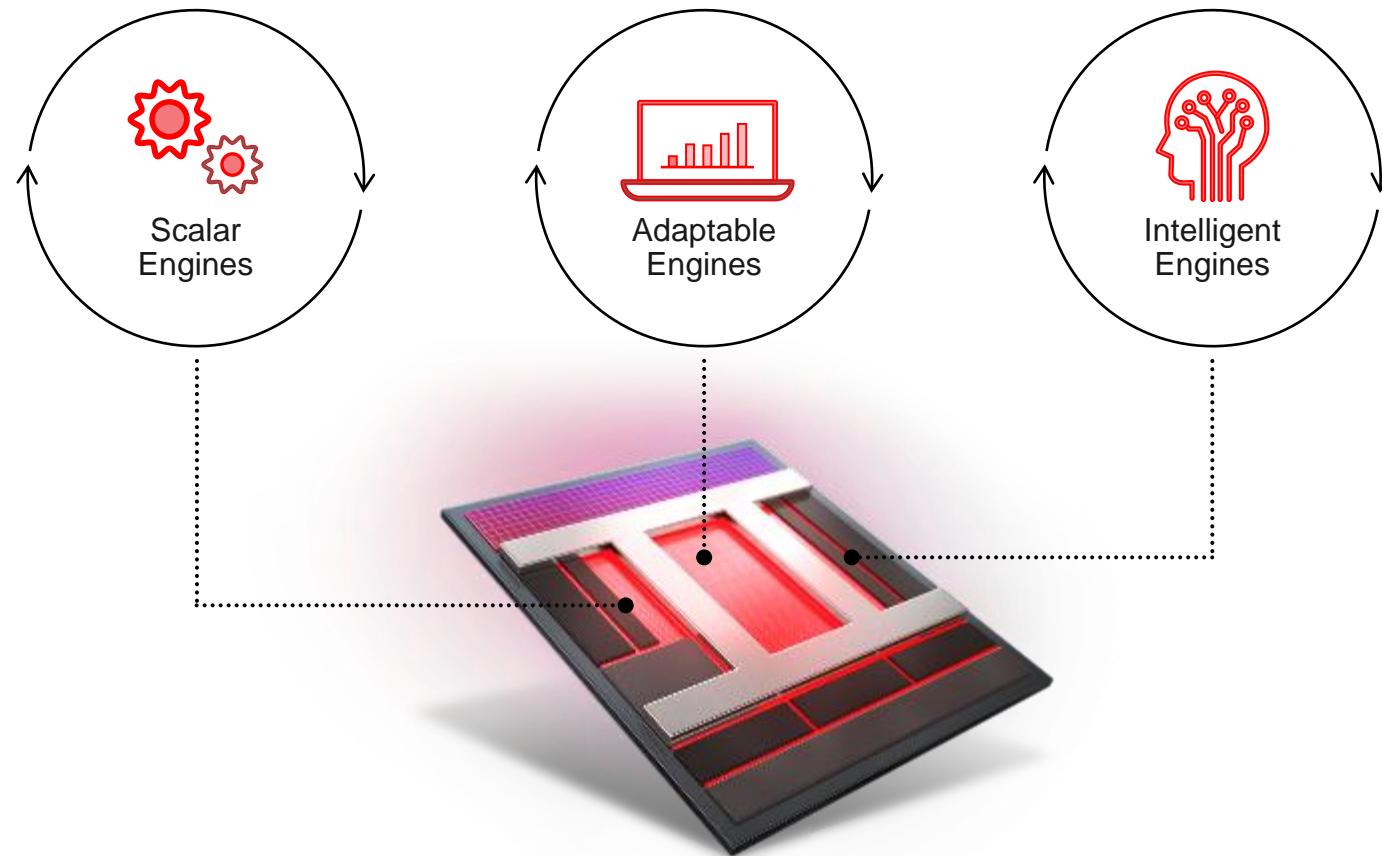
Versal ACAP: Tool Flow

Objectives

After completing this module, you will be able to do:

- Describe the Vitis™ Unified Software platform
- Describe the Vitis tool flow for Versal® devices
- Describe the full application acceleration flow for the Vitis platform
- Enumerate the toolchain components for Versal AI Engine programming

Adaptive Compute Acceleration Platform (ACAP)



Heterogenous Compute Platform

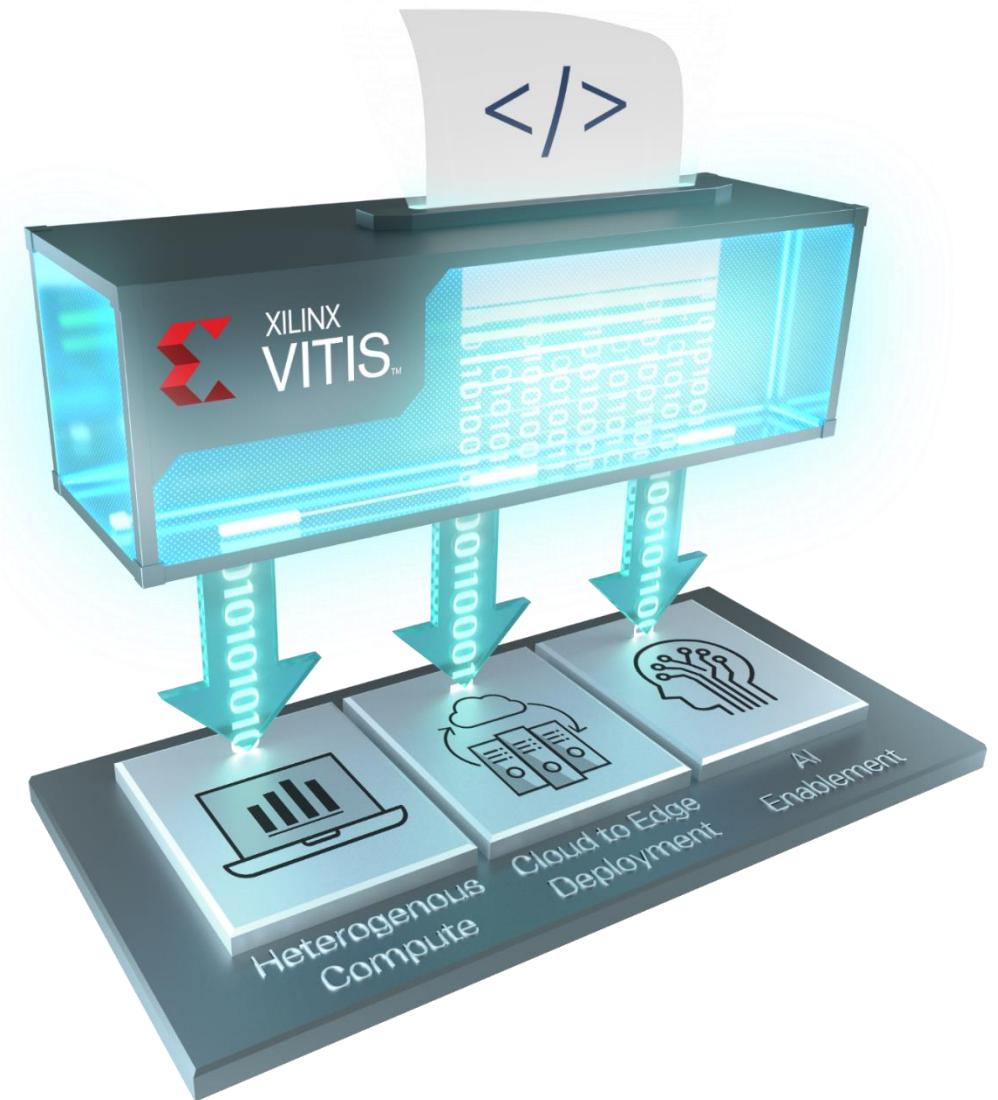
Vitis Unified Software Platform

- Adaptive computing
- Easy programmability of Versal ACAP hardware
- Support for open standards
- Software
 - Heterogeneous environment
 - Portability: edge to cloud



Vitis

Unified Software Platform

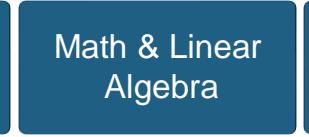


Vitis Unified Solution Stack

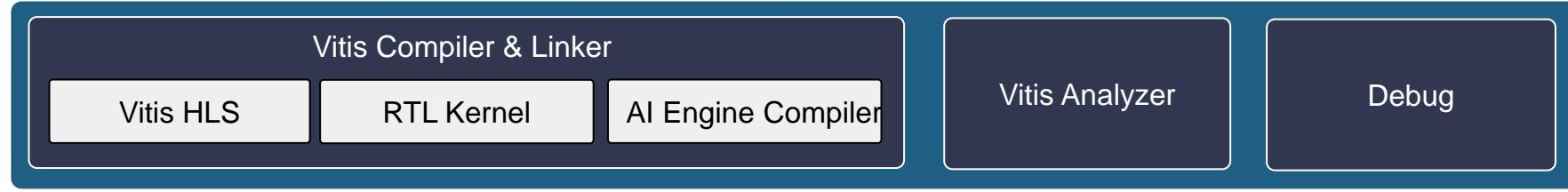
Domain-specific development environments



Vitis accelerated libraries



Vitis core development kit



Vitis target platform



Zynq-7000



Alveo Cards



Kria SOM



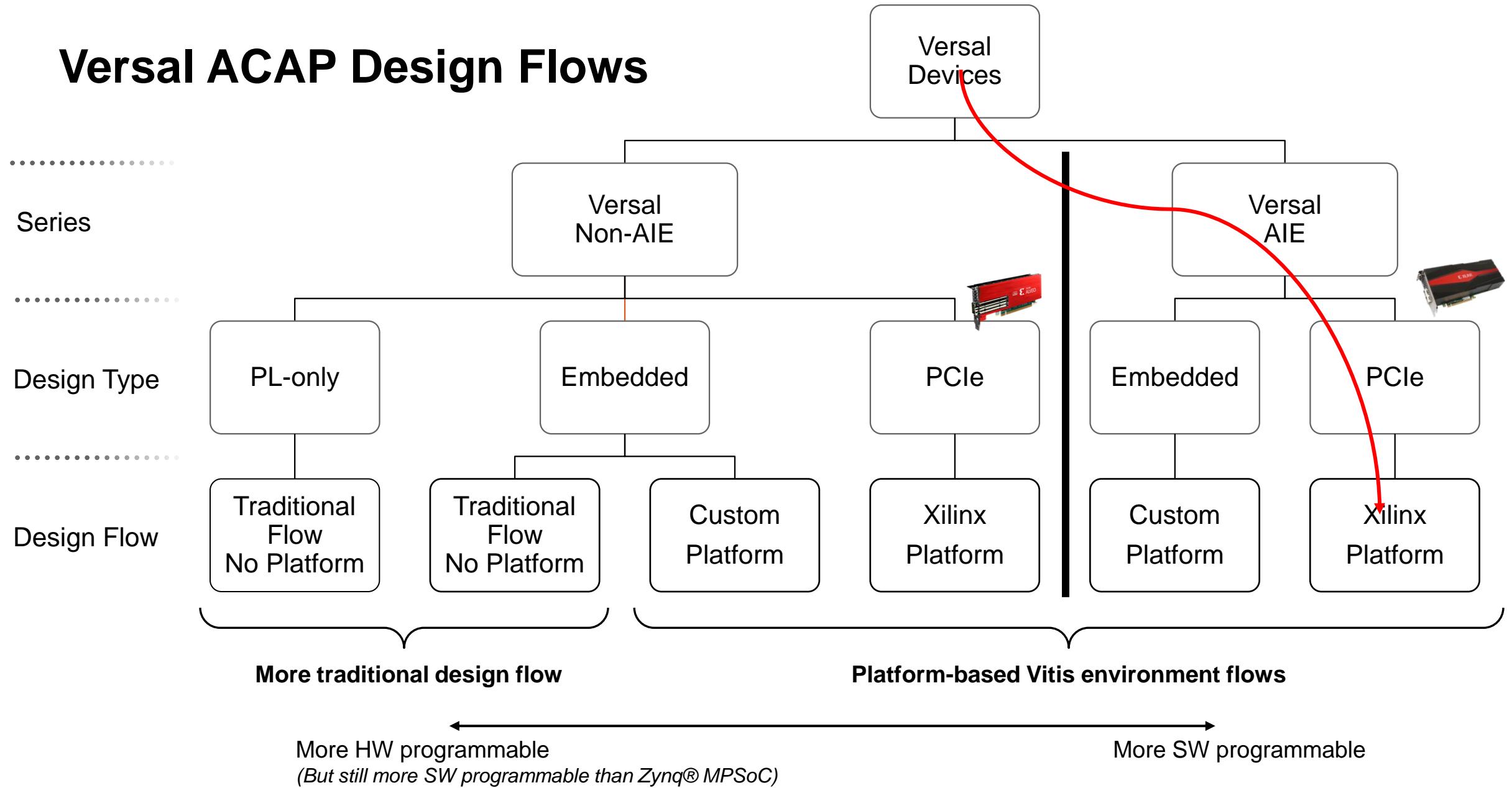
Varium Cards



Versal Cards

For Heterogenous Compute, Edge to Cloud

Versal ACAP Design Flows



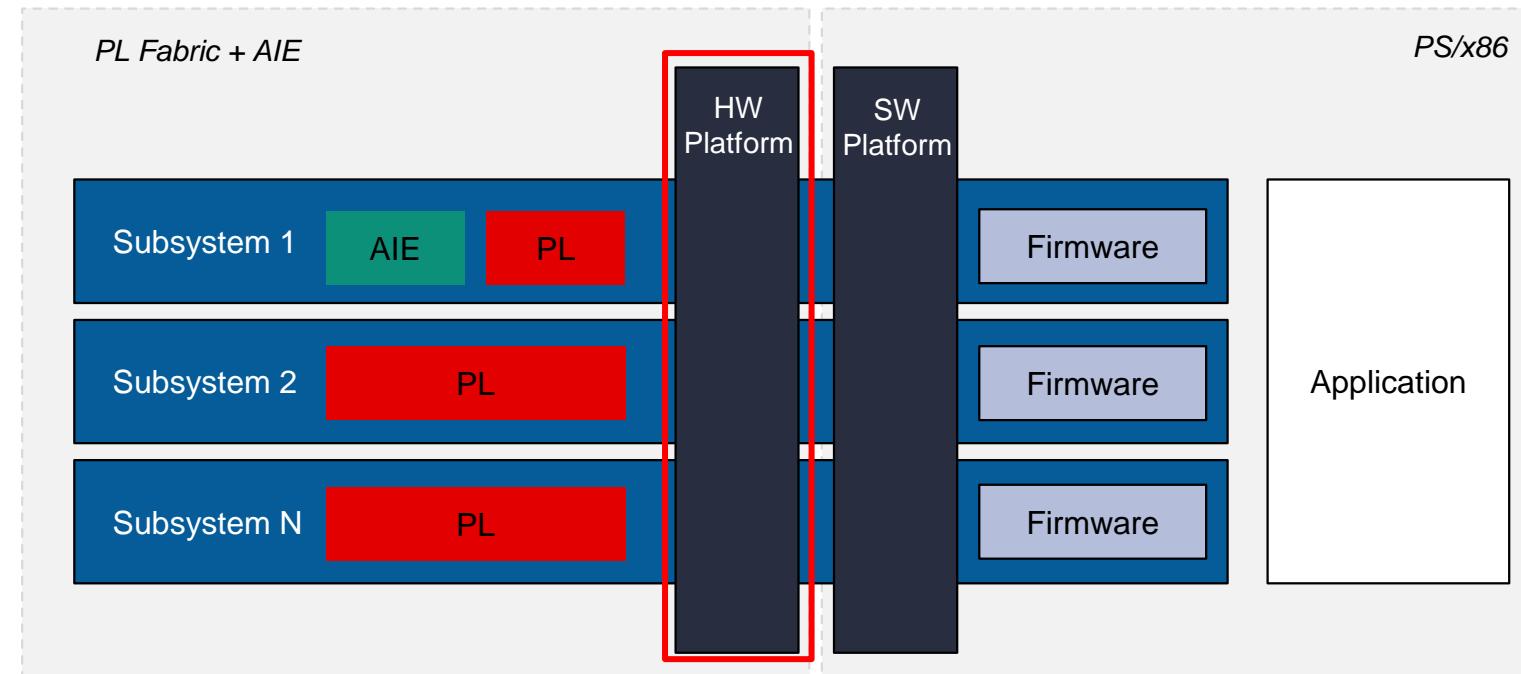
Platforms and Subsystem

- Approach for building a Versal system using the Vitis
- Divide-and-conquer approach
- Platform
 - Base hardware and software architecture and application context
 - Includes
 - External memory interfaces
 - Input/Output interfaces
 - PCIe connectivity
 - Software runtime
- Software Platform
 - Application runs on either PS or x86
- Hardware Platform
 - User code targets PL and AIE
- Subsystem
 - Combination of host code, PL kernels and AIE graphs

Platforms and Subsystem

Platform

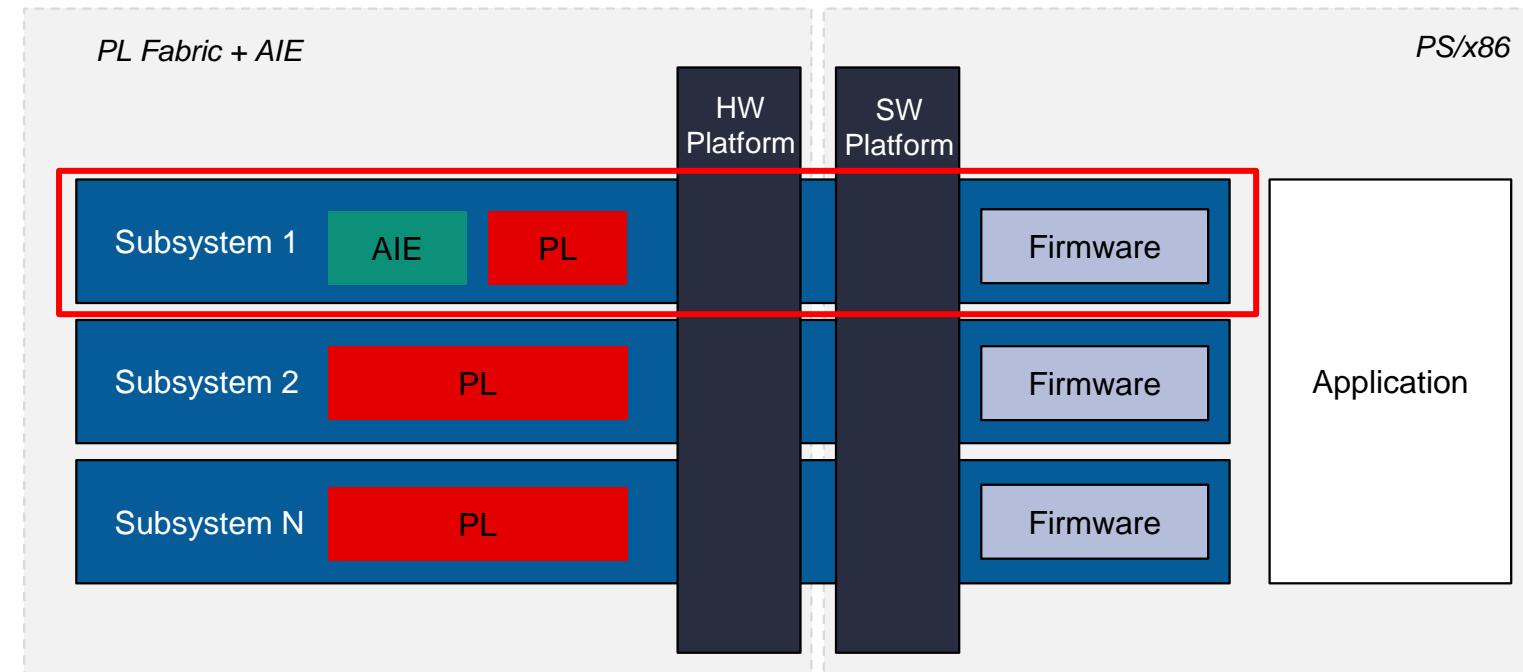
- Insulates developers from low-level details
- Enables focus on application development (SW, PL, or AIE)
- IPI block design containing essential IPs
- Generated using the PetaLinux tool suite



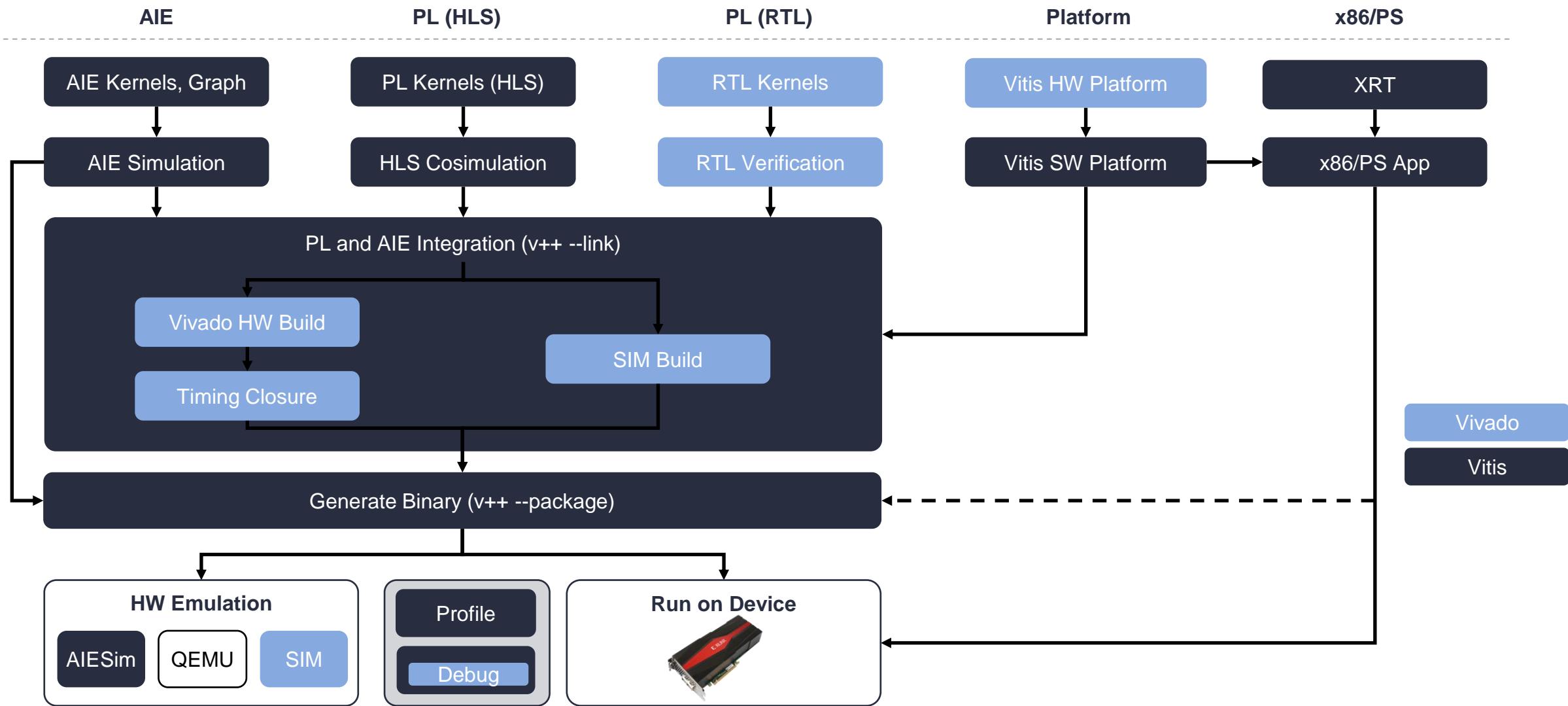
Platforms and Subsystem

Subsystem

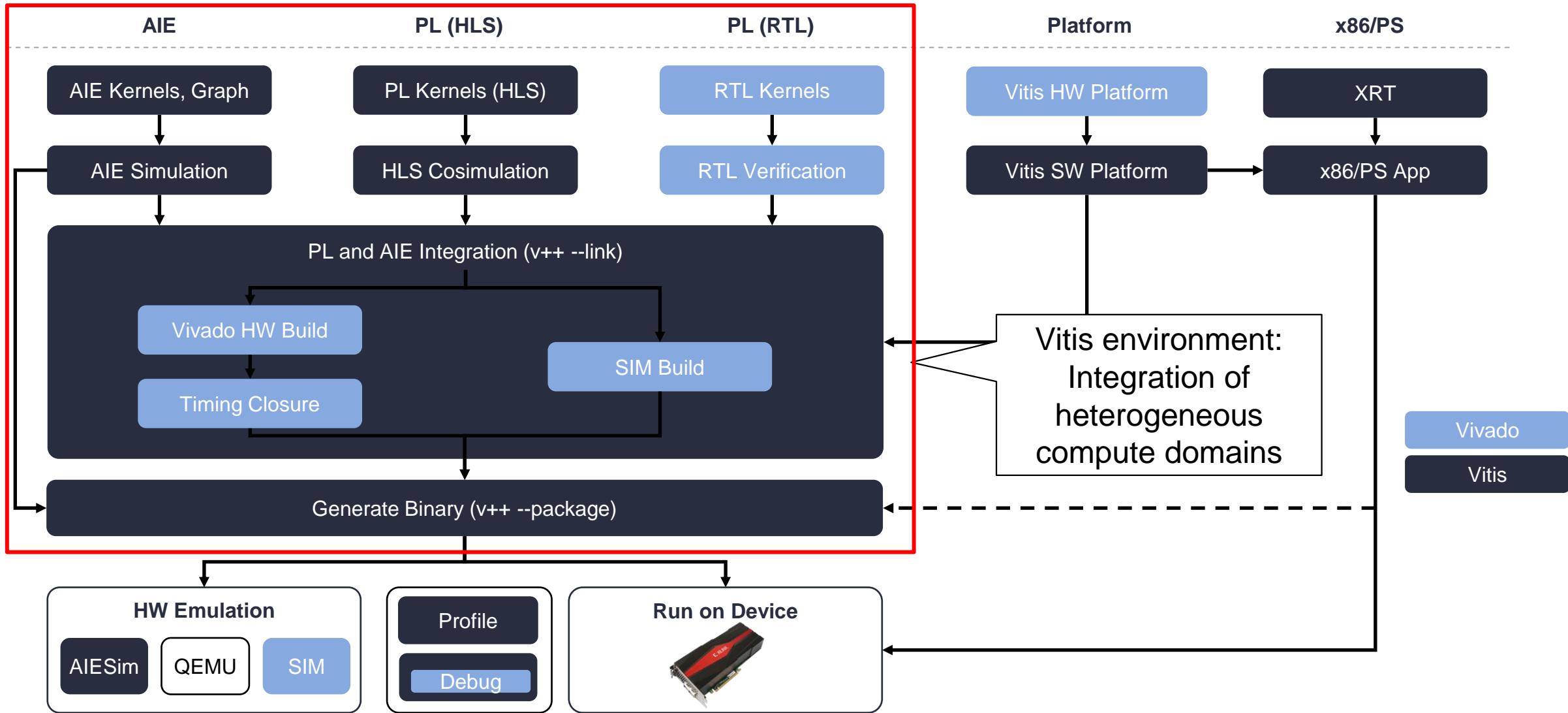
- Functions under host supervision
- Developed and tested independently
- Interacts via shared memory
- v++ --link
 - PL components
 - AIE interfaces
- v++ --package
 - AIE graph
 - PS software



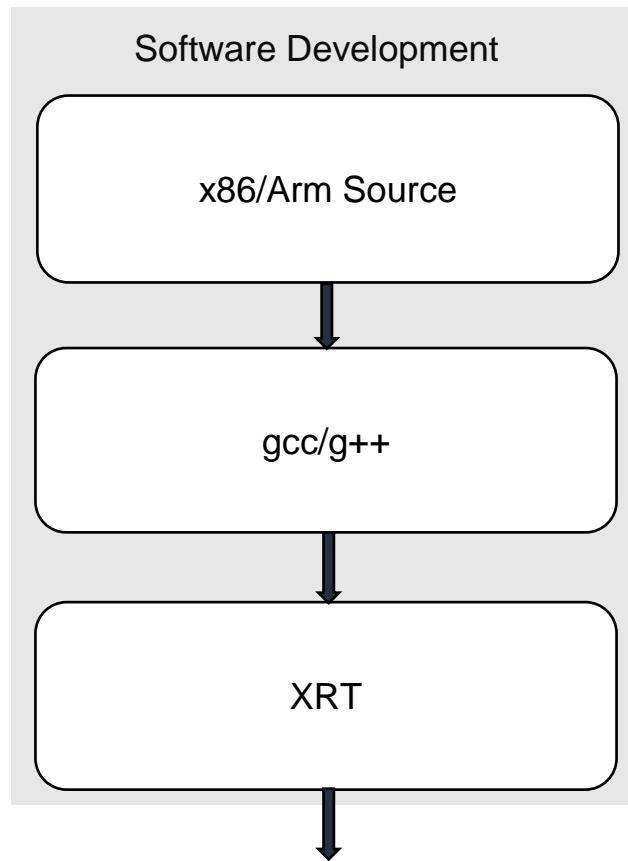
Vitis Tool Flow for Versal Devices



Vitis Tool Flow for Versal Devices



Vitis Environment: Full Application Acceleration Flow

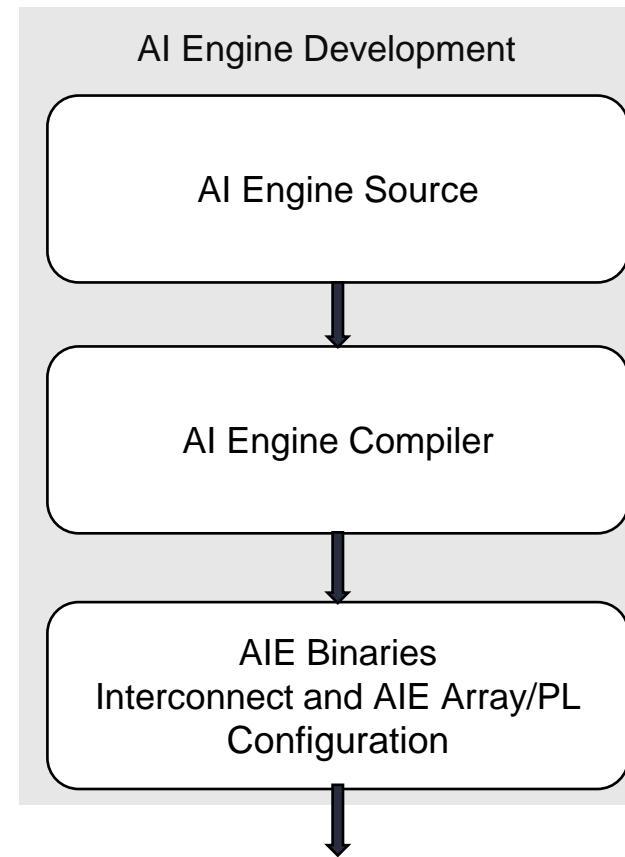


- **Executable file**
 - Compiled with gcc
 - Linked against Xilinx RunTime
 - User APIs
 - Driver
 - Runs on
 - PS for embedded platforms
 - x86 for PCIe platforms

Vitis Environment: AI Engine Development

libadf.a

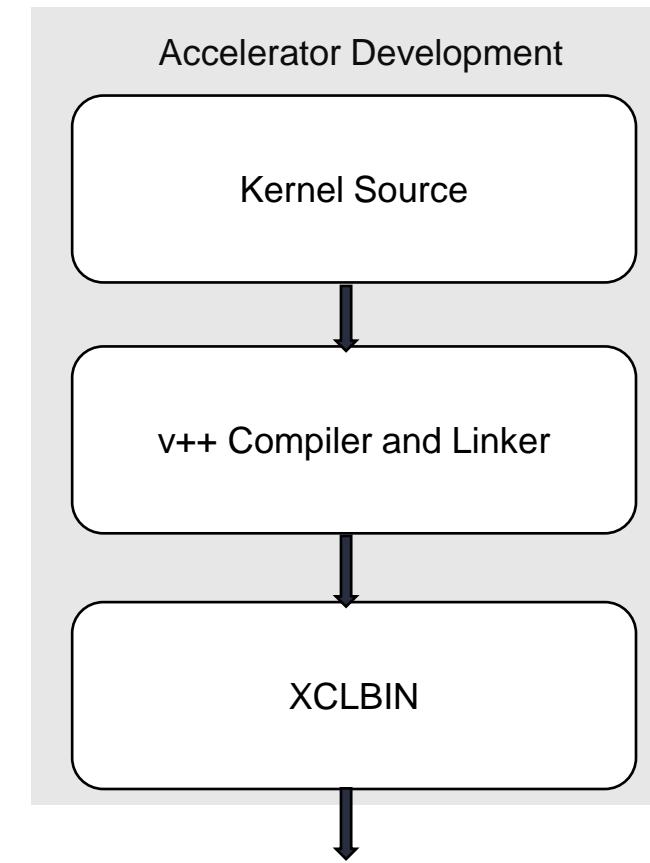
- Contains:
 - ELF and CDO files
 - Tool-specific data
 - Metadata



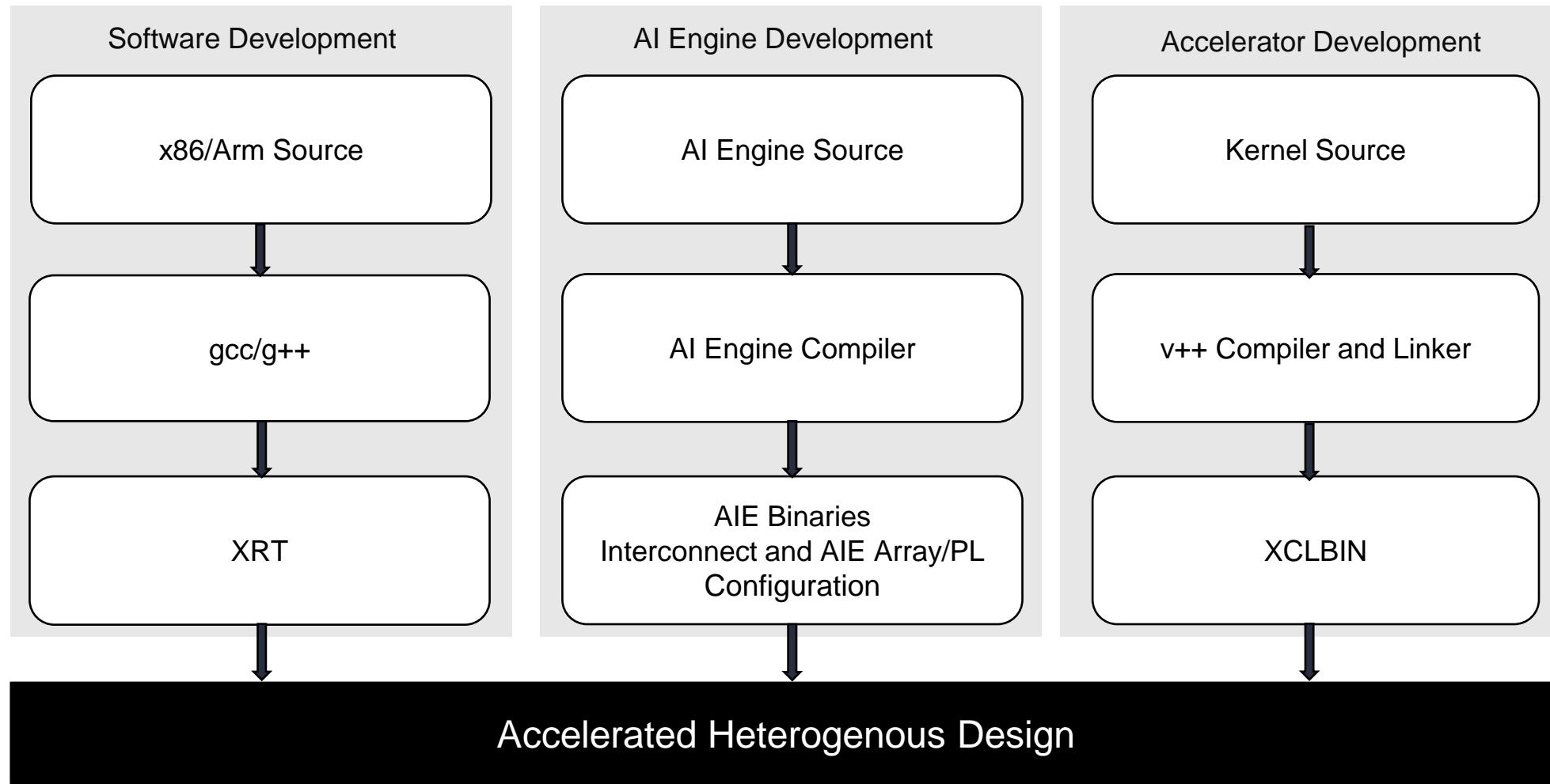
Vitis Environment: AI Engine Development

xclbin file

- Binary container
- PL kernels
 - C/C++
 - RTL
- Compiled and linked with v++ (Vitis)



Vitis Environment: AI Engine Development



Simulation Flows

Simulation Tool Flow	Functional Debug	Performance Analysis and Debug	Source-level Debug	Design Development Stage Usage
x86 Simulator	Yes	No	Yes	AI Engine kernel and graph debug
AI Engine Simulator	Yes	Yes (this simulation steps through the AI Engine assembly code and is useful in performance analysis as well as optimization)	Allows stepping through the AI Engine compiler-generated assembly code, which aids in code optimization; however, source-level visibility could be limited due to compiler optimization	AI Engine graph performance debug
Vitis Software Emulation	Yes	No	Yes	System-level emulation and functional debug
Vitis Hardware Emulation	Yes	Yes	Possible; however, provides limited source-level visibility due to compiler optimization	System-level emulation and performance debug

Simulation Models

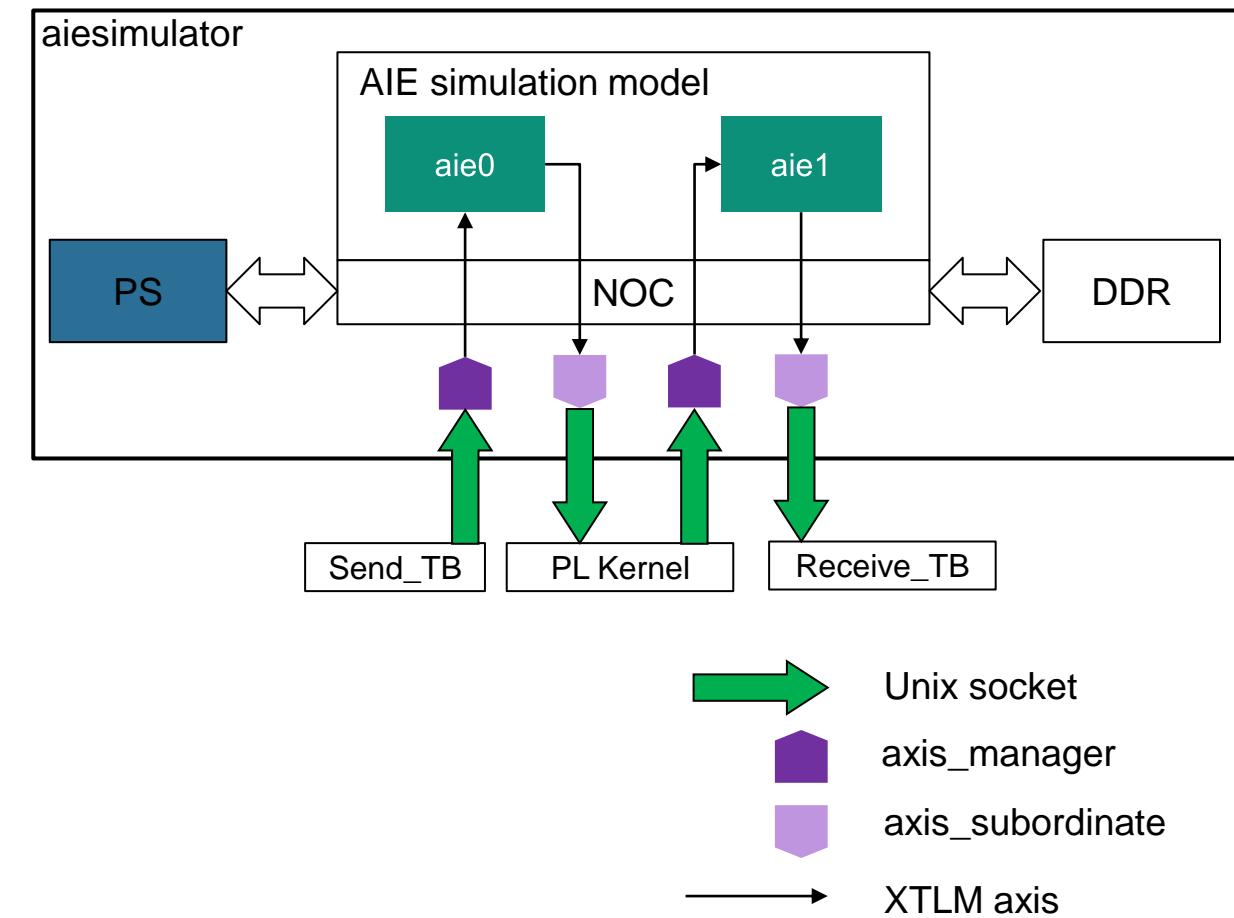
Simulation Tool Flow	AI Engine Kernels	PL Kernels	PL Platform	NoC/DDR Model	PS Model
x86 Simulator	x86 threads	C/SystemC	N/A	N/A	N/A
AI Engine Simulator	SystemC	SystemC	N/A	SystemC	N/A
Vitis Software Emulation	x86 threads	SystemC, RTL, external traffic generators	N/A	N/A	QEMU
Vitis Hardware Emulation	SystemC	SystemC, RTL, external traffic generators	RTL/SystemC	SystemC	QEMU

Simulation Features

Simulation Tool Flow	Trace	Profile	Host Application	Test Bench Support	AI Engine Data Flow Visibility
x86 Simulator	No	No	Through the main() function in graph.cpp	File based	Yes (via the snapshot feature)
AI Engine Simulator	Yes	Yes	Through the main() function in graph.cpp	File based	Yes (via the Trace view in the Vitis analyzer)
Vitis Software Emulation	No	No	Through the main() function in the host application, targeting bare-metal or Linux-XRT	File based External traffic generators	Yes (via the snapshot feature)
Vitis Hardware Emulation	Yes	Yes	Through the main() function in the host application, targeting bare-metal or Linux-XRT	File based External traffic generators	Yes (via the Trace view in the Vitis analyzer)

External Traffic Generators

- Flexible Traffic Generators
 - Inserted easily in simulation and emulation flows
- Simulate PL kernels with AI Engines
- Supports:
 - C++
 - Python
 - HDL
- Provides a method for platforms and applications to inject traffic onto the I/O during simulation
- Requires both the inclusion of:
 - Streaming I/O kernels or IP in your design
 - Use of Python/C++/C code
- Support for
 - AXI4-Stream
 - AXI3/AXI4 memory mapped



Summary

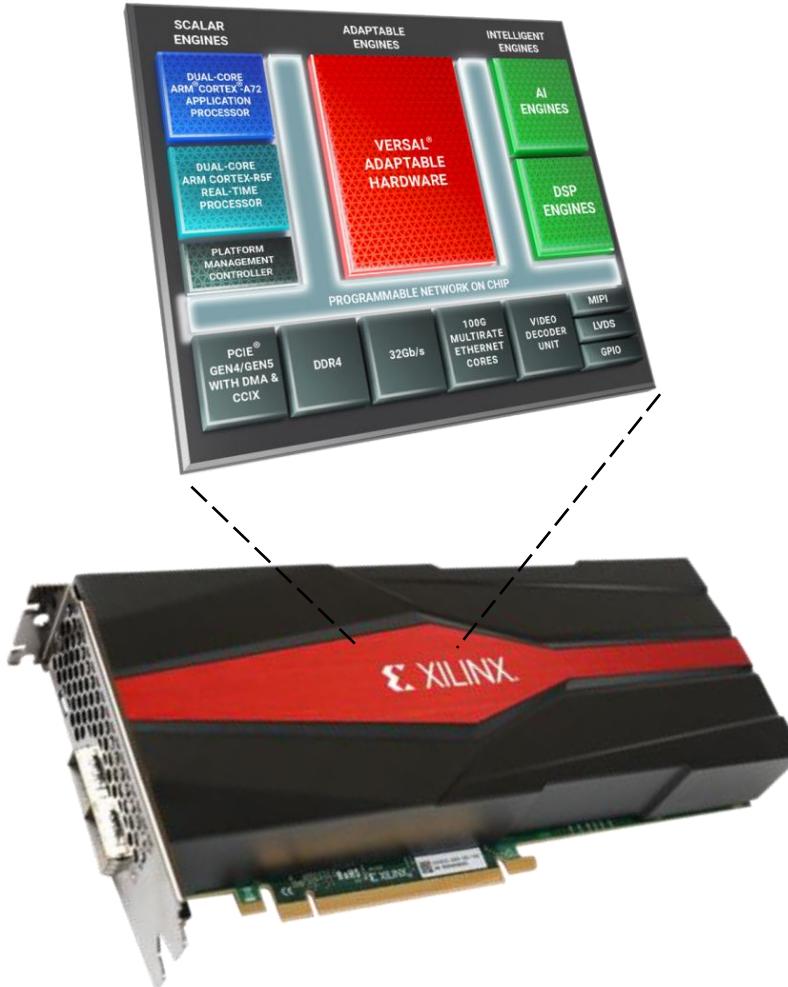
- Vitis unified software platform for performing software development
 - Highly customizable environment supports Xilinx-specific tools for compiling, debugging, profiling, etc.
- All types of developers
 - Hardware designers (Vivado Design Suite users)
 - Firmware and software developers (Vitis platform users)
 - AI/ML developers (Vitis AI environment users)



VCK5000 Versal Development Card

AMD
together we advance_

VCK5000 Development Card



5G Radio & Beamforming

Data Center Compute

Artificial Intelligence

Signal Processing

Video Processing for Smart Cities

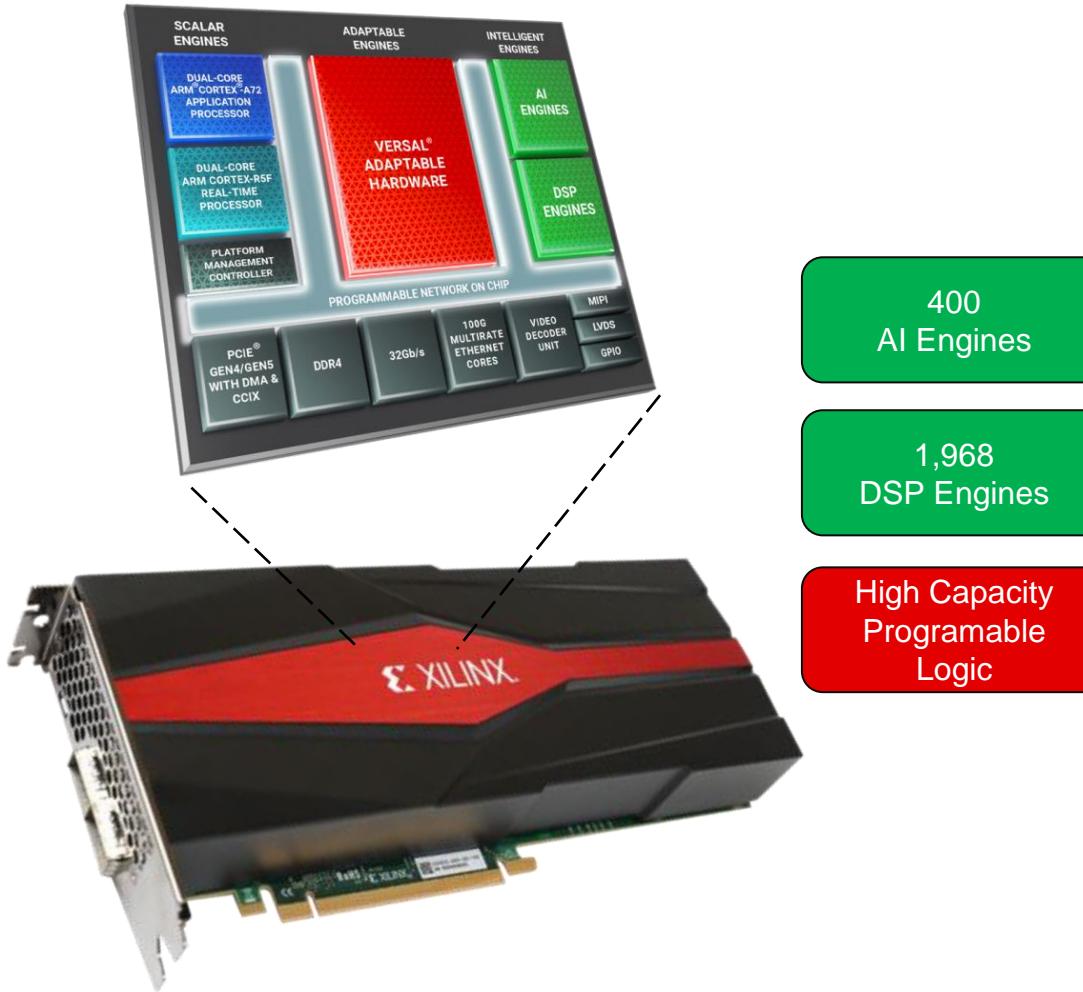
Medical Image Processing

Radar Processing

Wireless Test Equipment

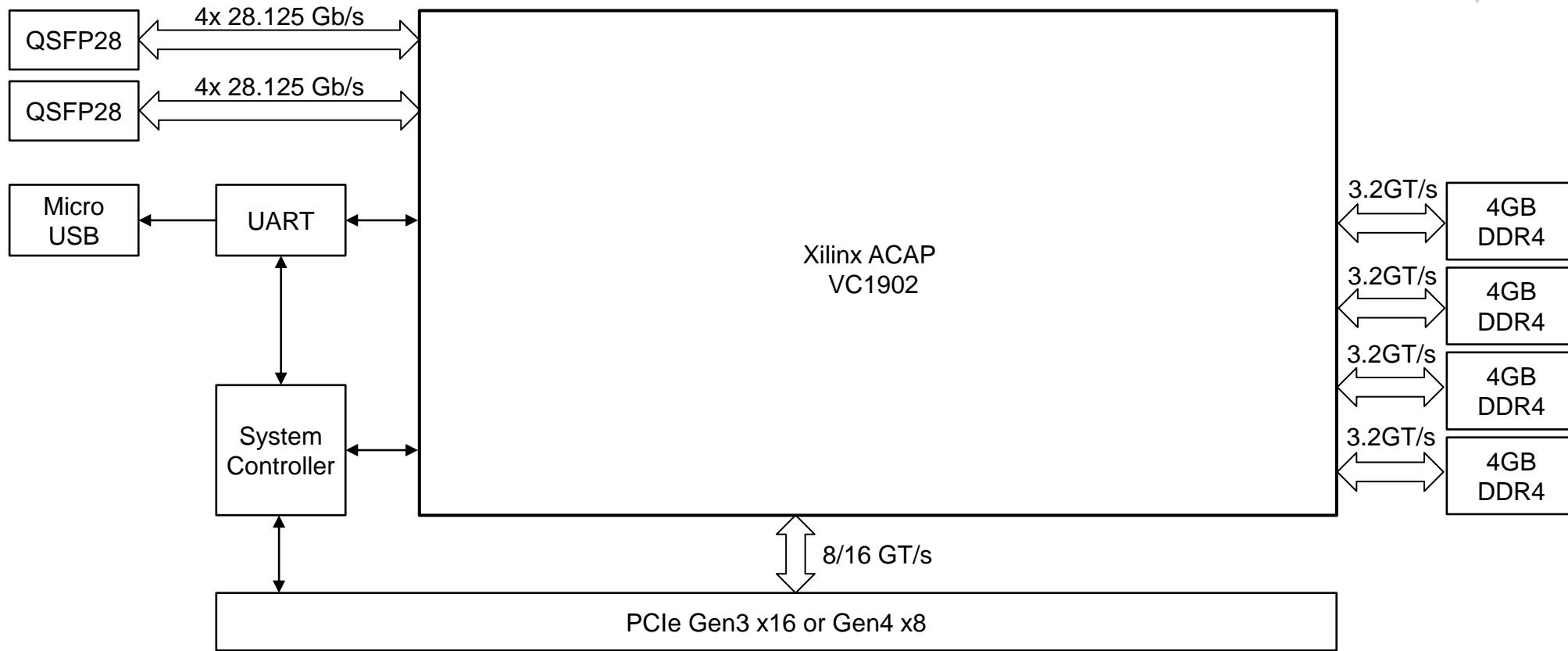
<https://www.xilinx.com/products/boards-and-kits/vck5000.html>

VCK5000 Development Card



Key Specifications		VCK5000
Off-chip Memory Capacity	16 GB	
Off-chip Total Bandwidth	102.4 GB/s	
Internal SRAM Capacity	23.9 MB	
Internal SRAM Total Bandwidth	23.5 TB/s	
Look-up Tables (LUTs)	899,840	
Peak INT8 TOPS	145	
AIE Array Max Frequency	1.25 GHz	
PCI Express	Gen3x16 / Gen4x8	
Network Interfaces	2x QSFP28 (100GbE)	
Dimensions	FHFL Dual Slot	

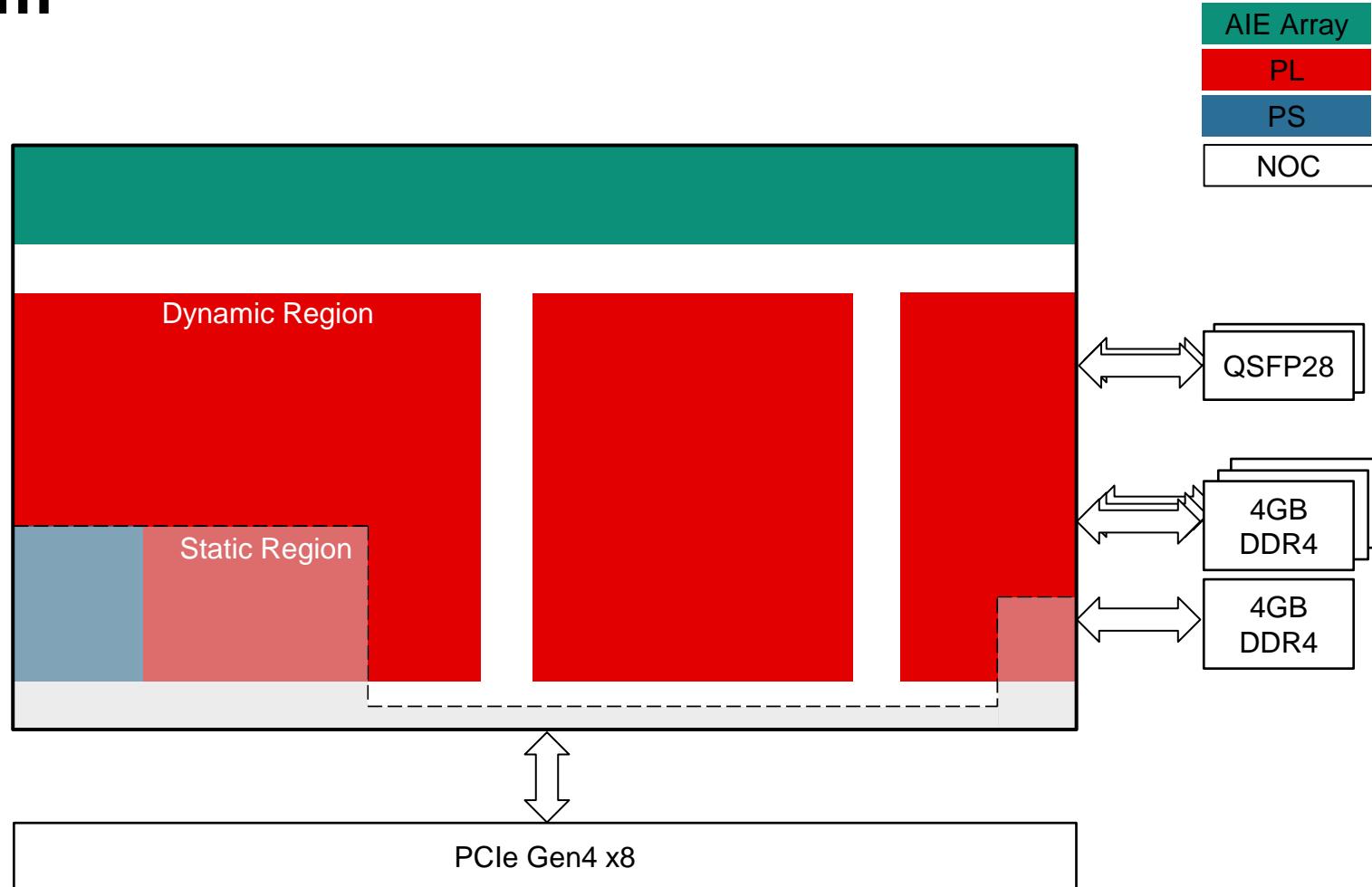
VCK5000 Connectivity and Features



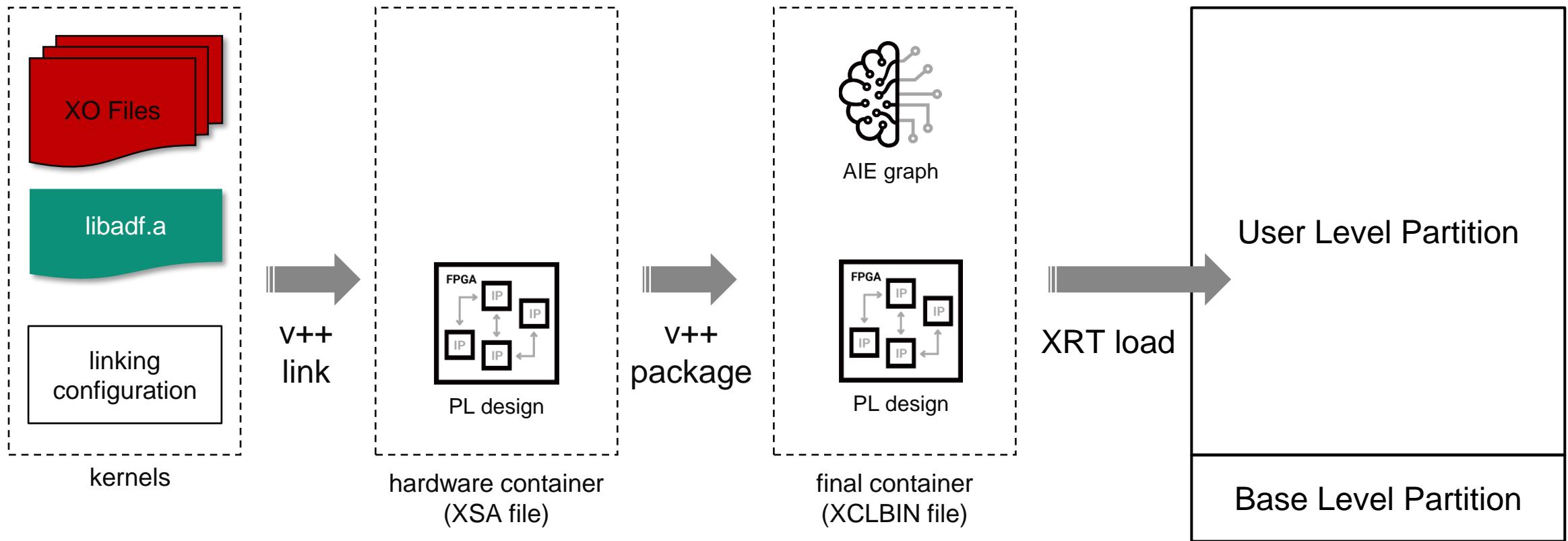
Find out more information on [VCK5000 Versal Development Card](#)

VCK5000 Gen4 x8 Platform

- Static Region
 - Base Level Partition (BLP)
 - PCIe link for host to card connectivity
 - Data movement between host and card memory
 - 4GB DDR4
- Dynamic Region
 - User Level Partition (ULP)
 - Region for PL kernels
 - AIE Array @ 1.25 GHz
 - 12GB DDR4
 - 128KB PLRAM
 - 2x QSFP28
 - 2x scalable clocks
 - Default: 300 & 500 MHz



VCK5000 Hardware Development with Vitis



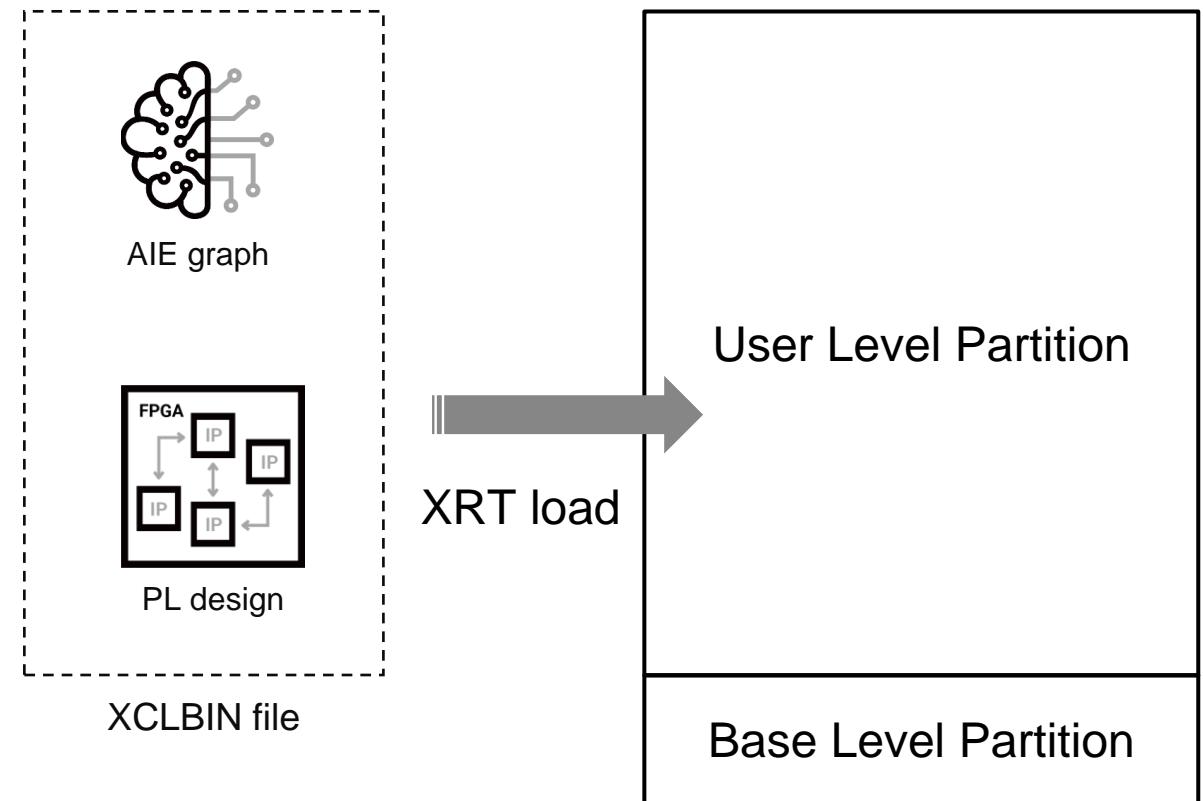
v++ is the Vitis equivalent to g++

VCK5000 Software Development

- Host code runs on x86
- Unified XRT based programming model
 - C & C++ APIs
 - g++ compiler
- Data movement and job execution
- AIE kernels are controlled via the PL

The Embedded Run Time (ERT) runs in the PS

- Job scheduling



Other Versal ACAP Boards & Kits Portfolio

Versal® Prime Series Eval Kit

VMK180



- Featuring signal processing and connectivity
- Serving the broadest applicability across multiple markets
- Featuring VM1802

Versal AI Core Series Eval Kit

VCK190



- Featuring AI and DSP acceleration engines
- Optimized for cloud, networking, and autonomous applications
- Featuring VC1902

Versal Premium Series Eval Kit

VPK120



- Featuring power-optimized Hard IP
- Optimized for cloud, networking, and test & measurement applications
- Featuring VP1202



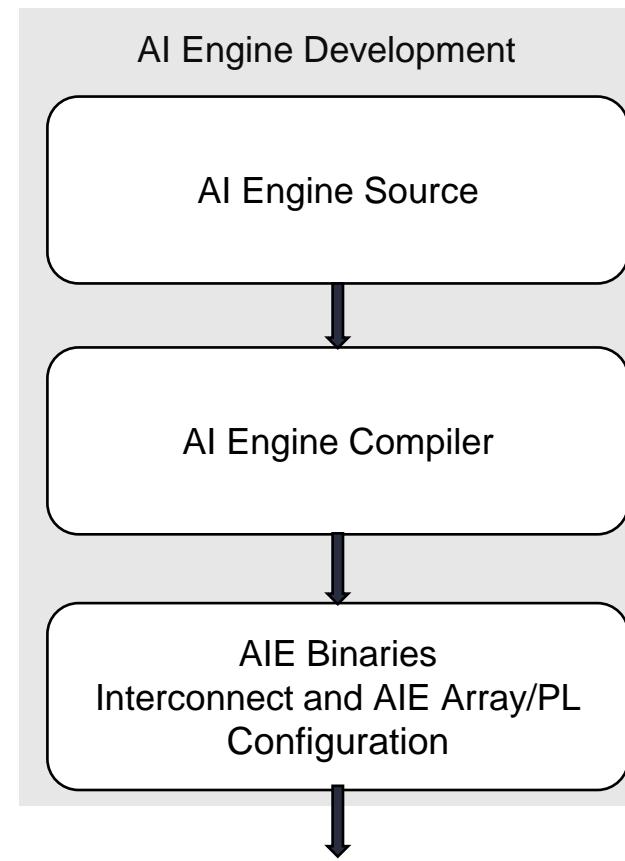
The Programming Model: Single Kernel

Objectives

After completing this module, you will be able to:

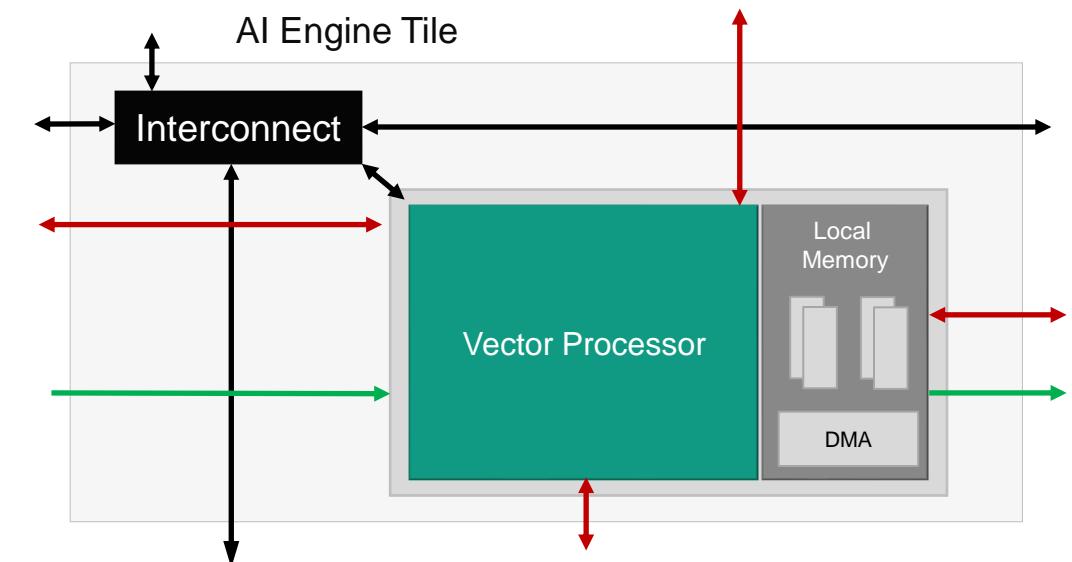
- Describe the full application acceleration flow using heterogeneous engines
- Describe the Versal® AI Engine kernel programming flow for programming and building a single kernel

Vitis Environment: AI Engine Development



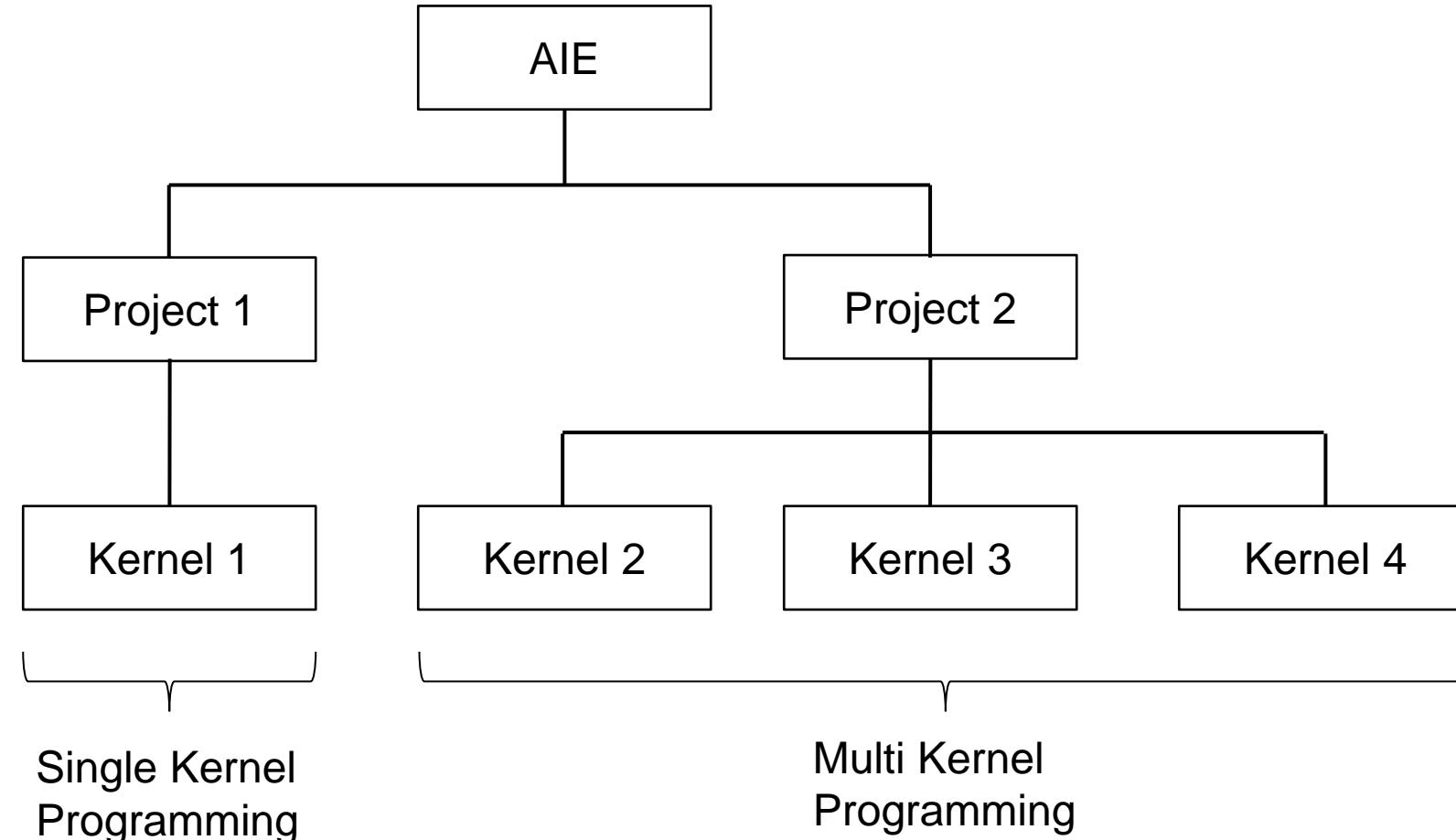
AI Engine Programming

- AI Engine Kernel
 - C/C++ Program
 - Written Using AI Engine API/ or Intrinsic Calls
 - Targets the AIE Tile (VLIW/SIMD)
 - Compiled by the aiecompiler
 - ELF file runs on the AI Engine Tile
- Recommendations
 - Use AI Engine APIs for your designs
 - Consider intrinsics only where performance is required (not covered by AI Engine APIs)



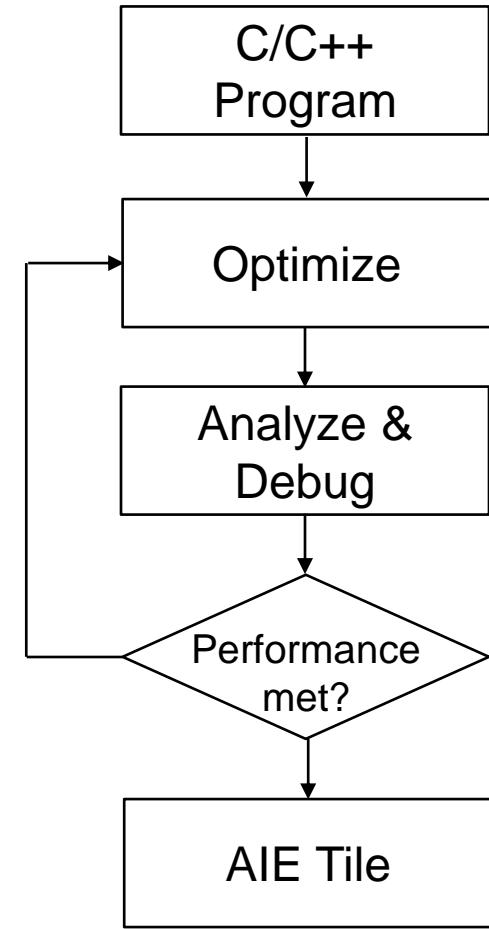
→ Memory Interface
→ Stream Interface
→ Cascade Interface

AI Engine Programming – Flow chart



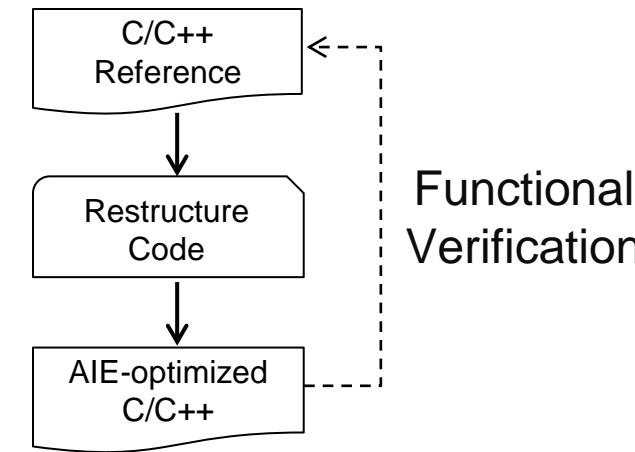
Single Kernel Programming

- AI Engine programs for each kernel
- Optimize
 - Vector data types
 - AI Engine APIs
 - AI Engine intrinsic functions
 - Window function APIs
- Analyze & Debug
- Repeat until performance is achieved



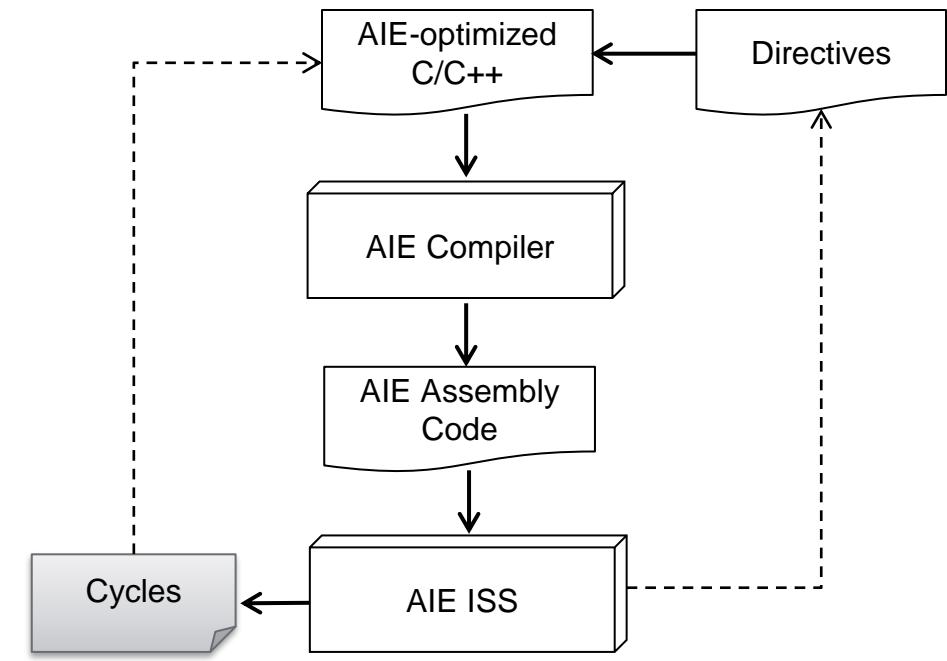
AI Engine Kernel Programming Flow – Functional Verification

- Restructure code
 - Kernel vectorization
 - Vector datatypes
 - AI Engine APIs
 - Intrinsic functions
 - Memory optimization
- Run functional simulation



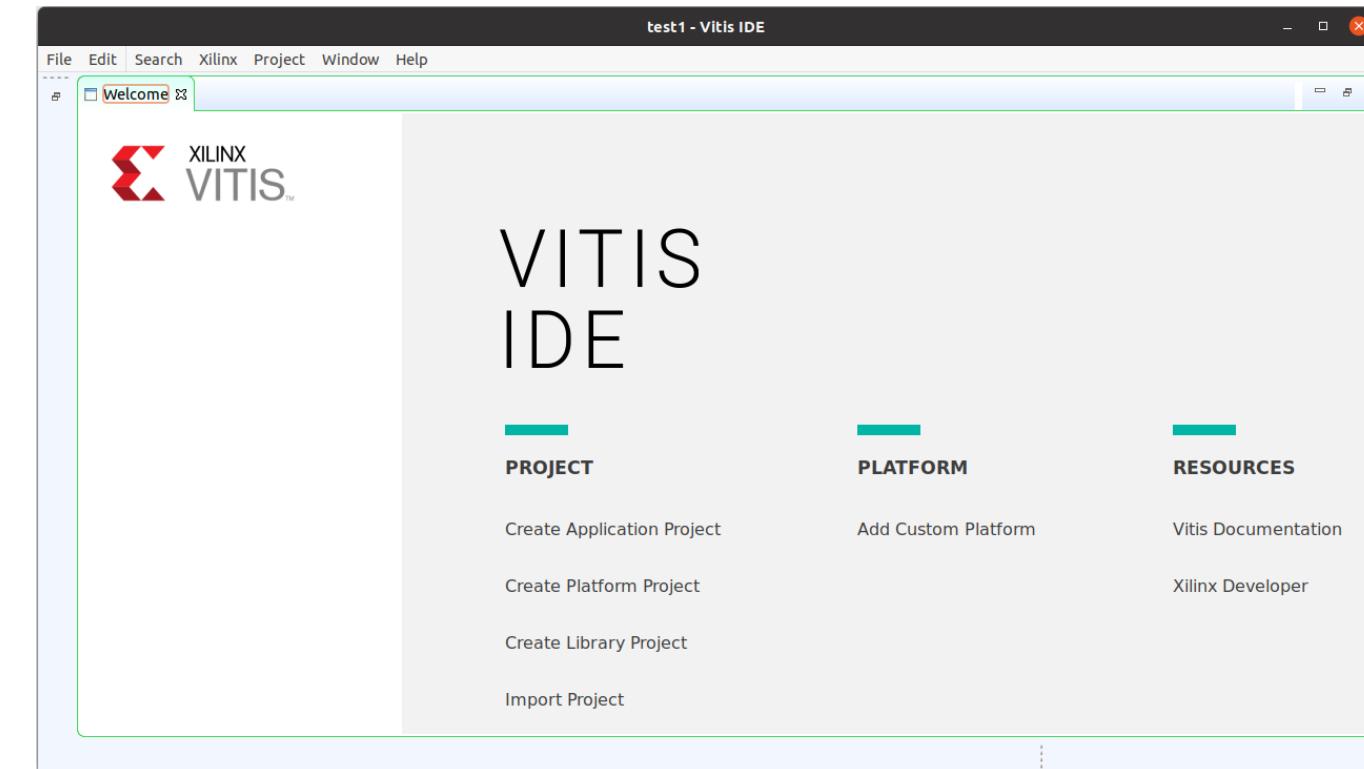
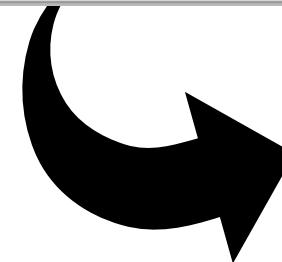
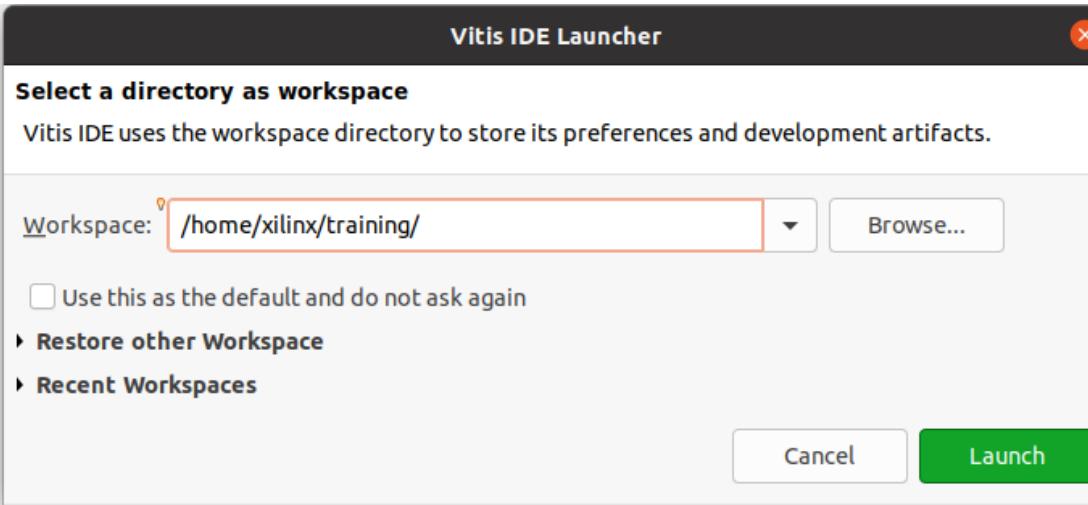
AI Engine Kernel Programming Flow – Performance Verification

- Development tools
 - aiecompiler
 - Debugger
 - Profiler
 - Instruction Set Simulation (ISS)
- Directives
 - Loop unrolling
 - Software pipeline
 - Helps achieving performance



Starting the Vitis IDE

- Start the Vitis IDE and choose a “workspace” that will contain your projects



Vitis IDE

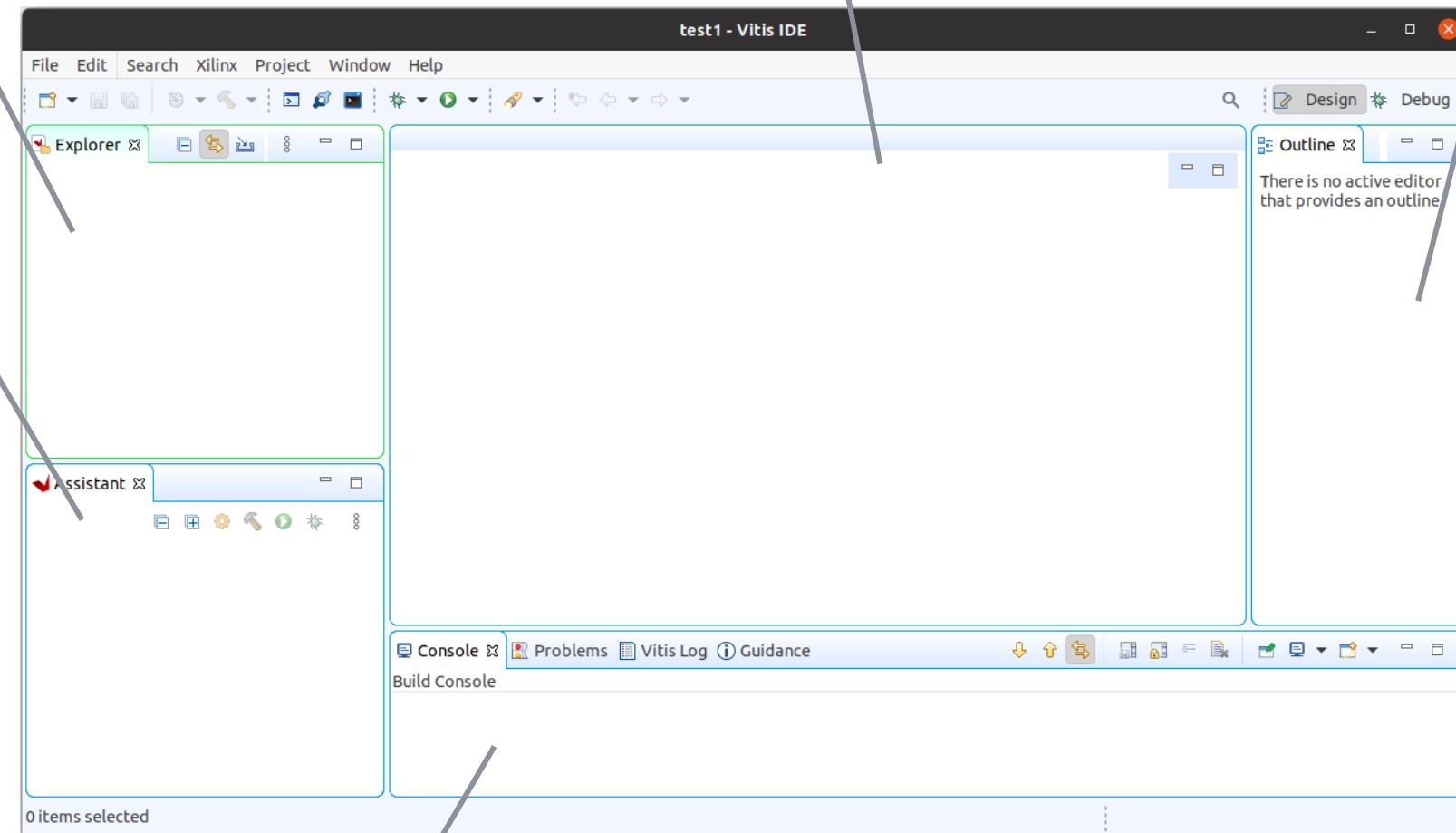
Editor, project viewer, ...

Explorer: Contains project directory and files

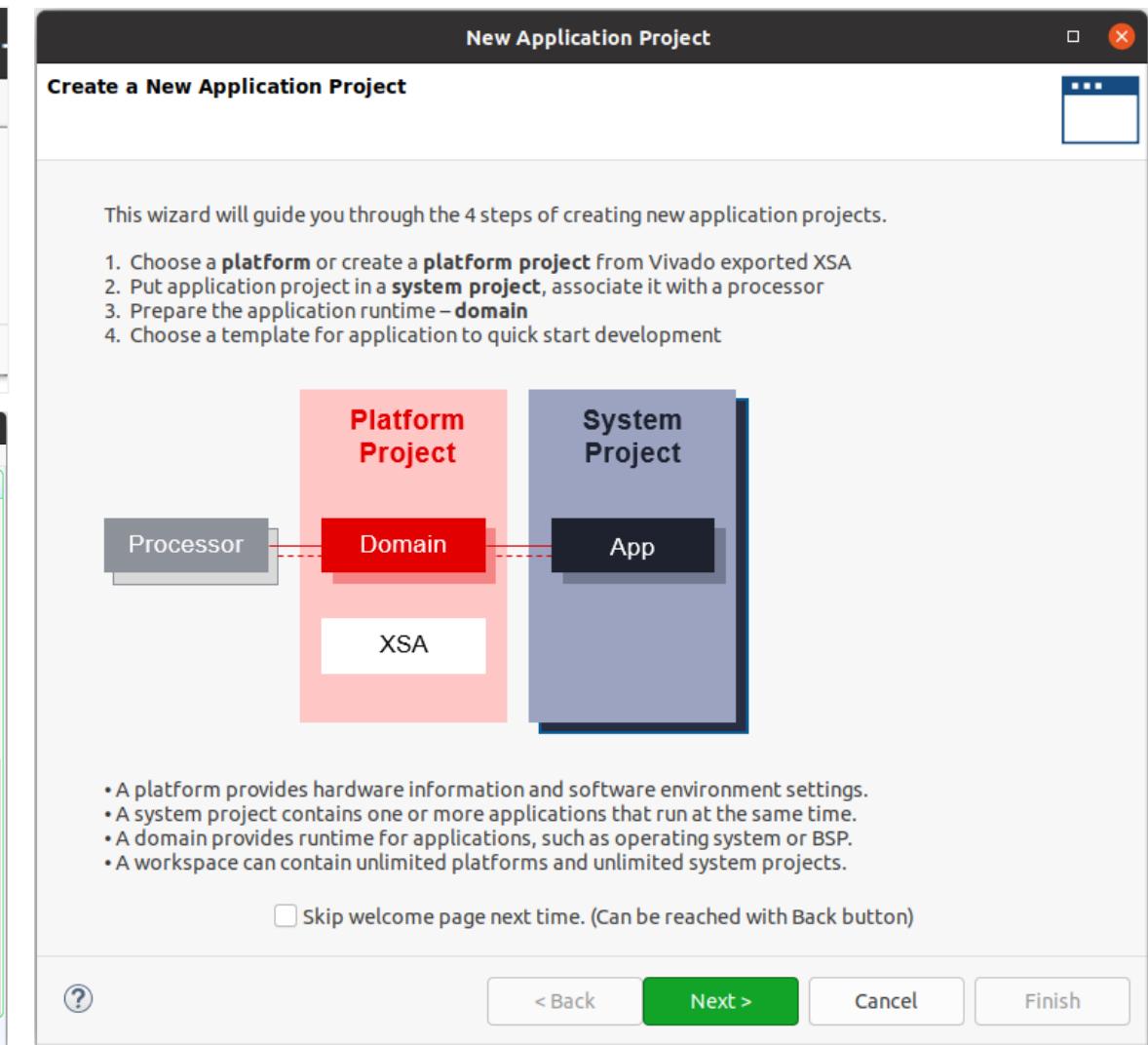
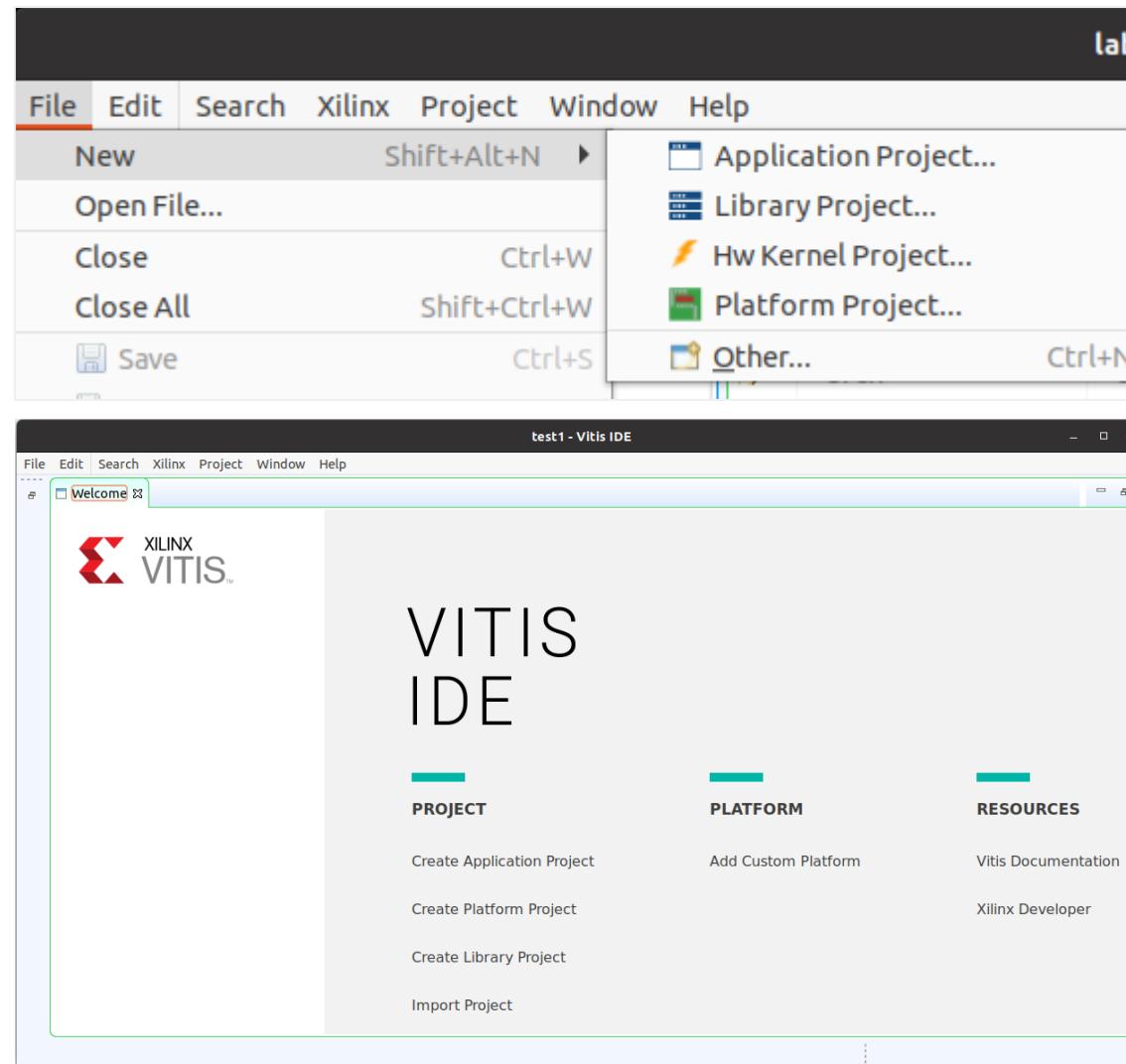
Assistant: Shortcuts for important reports and views

Outline: Important features of data displayed in the editor

Console, compiler errors and warning, guidance, ...

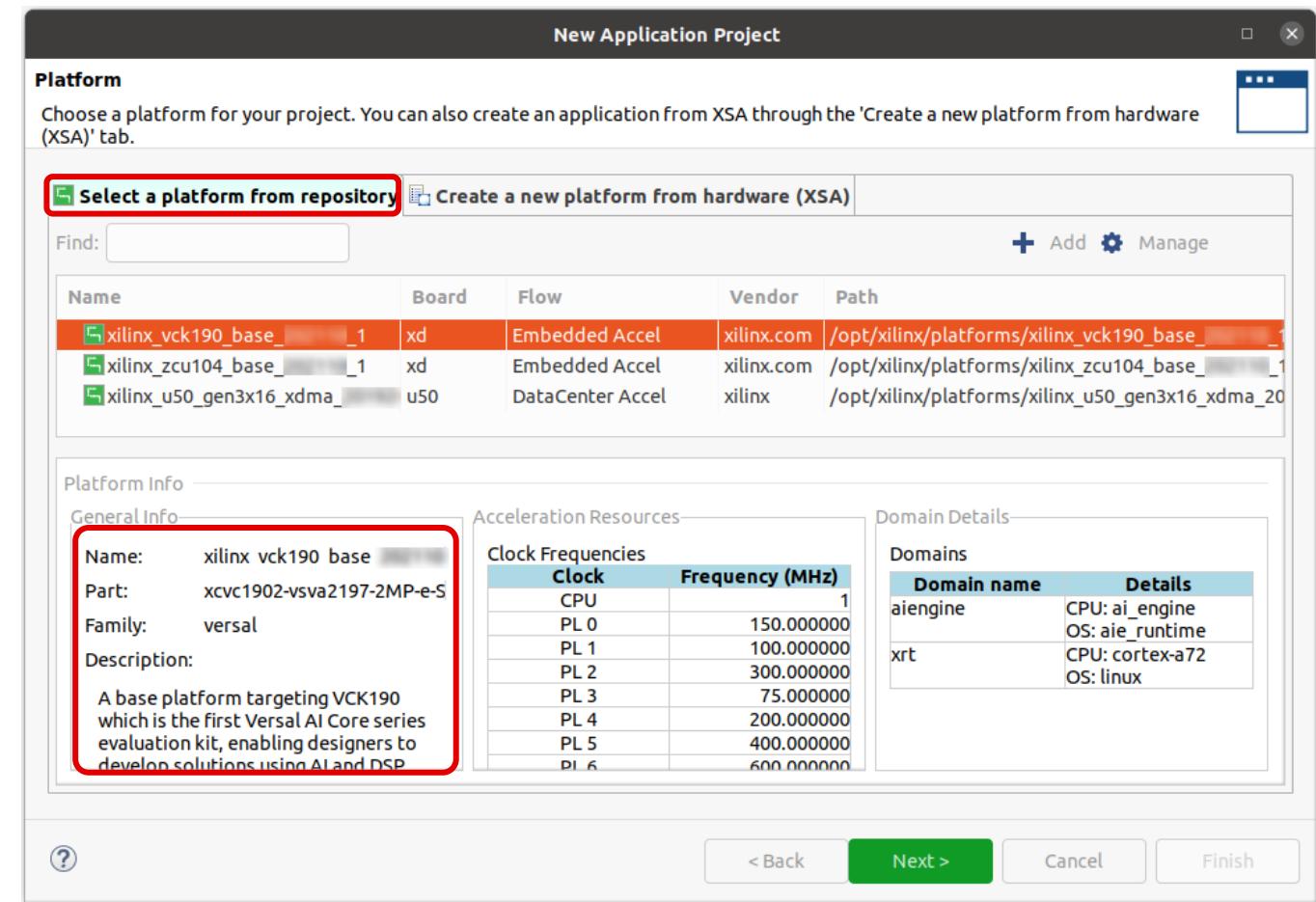


Project Creation



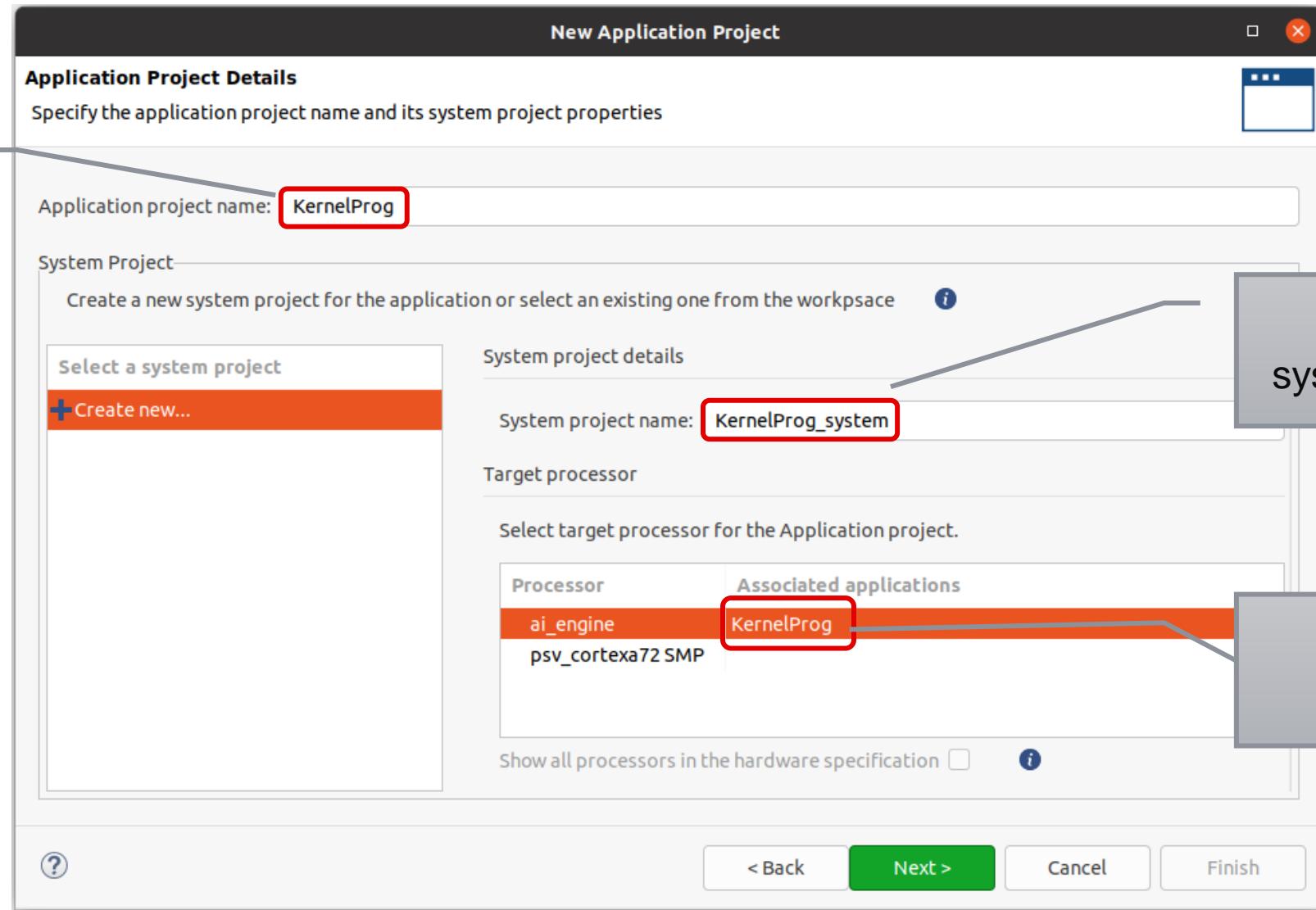
Platform Selection

- Number of platforms are available
- New one can be created
- Platform info includes:
 - Name and description
 - Available clocks
 - Domains available:
 - AI Engine
 - PS (CPU, OS)



Application Project Name Specification

Project name
(user input)



Auto-generated
system project name

Modifiable

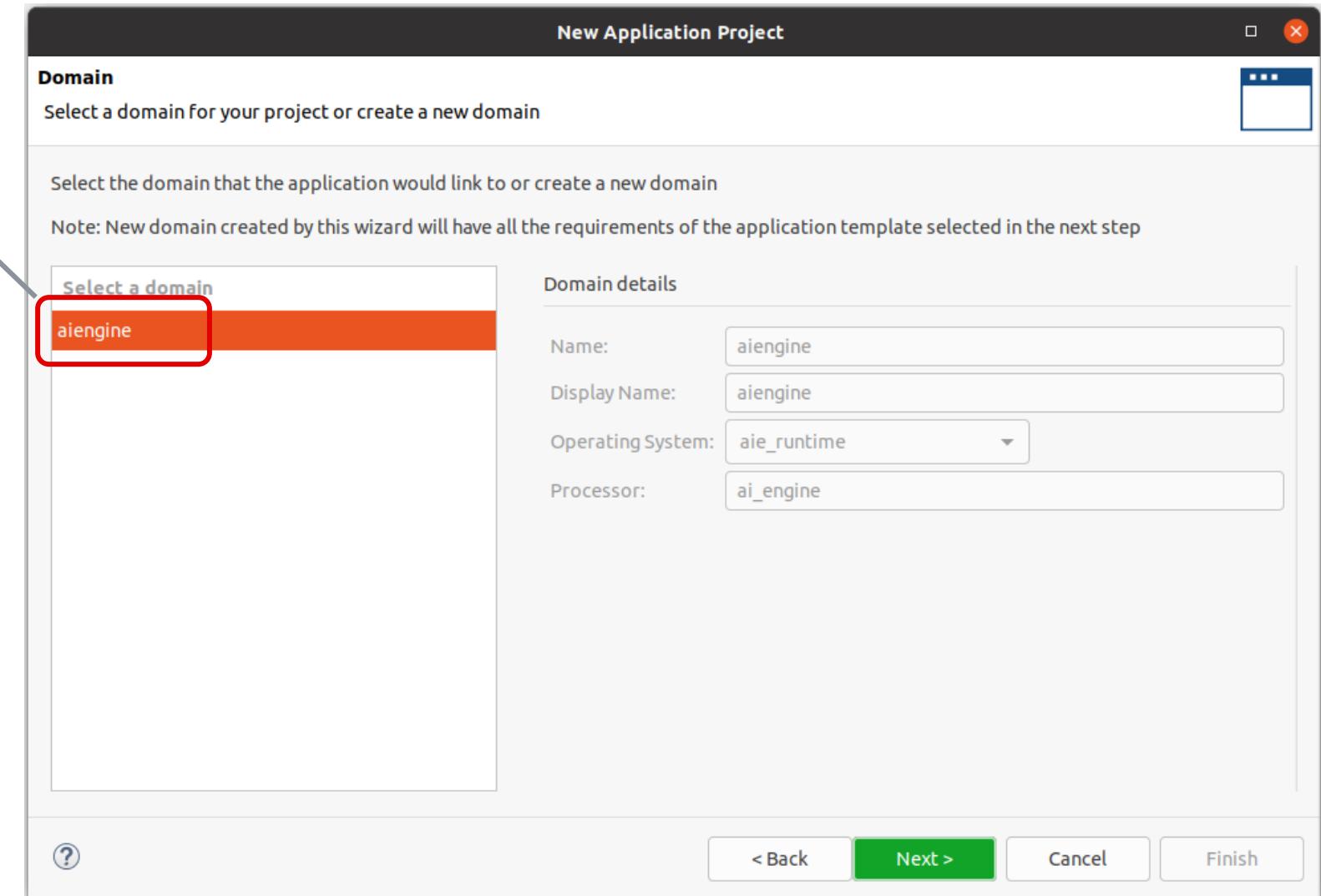
User-selectable
domain

*AI Engine Domain
in this case*

Domain Selection

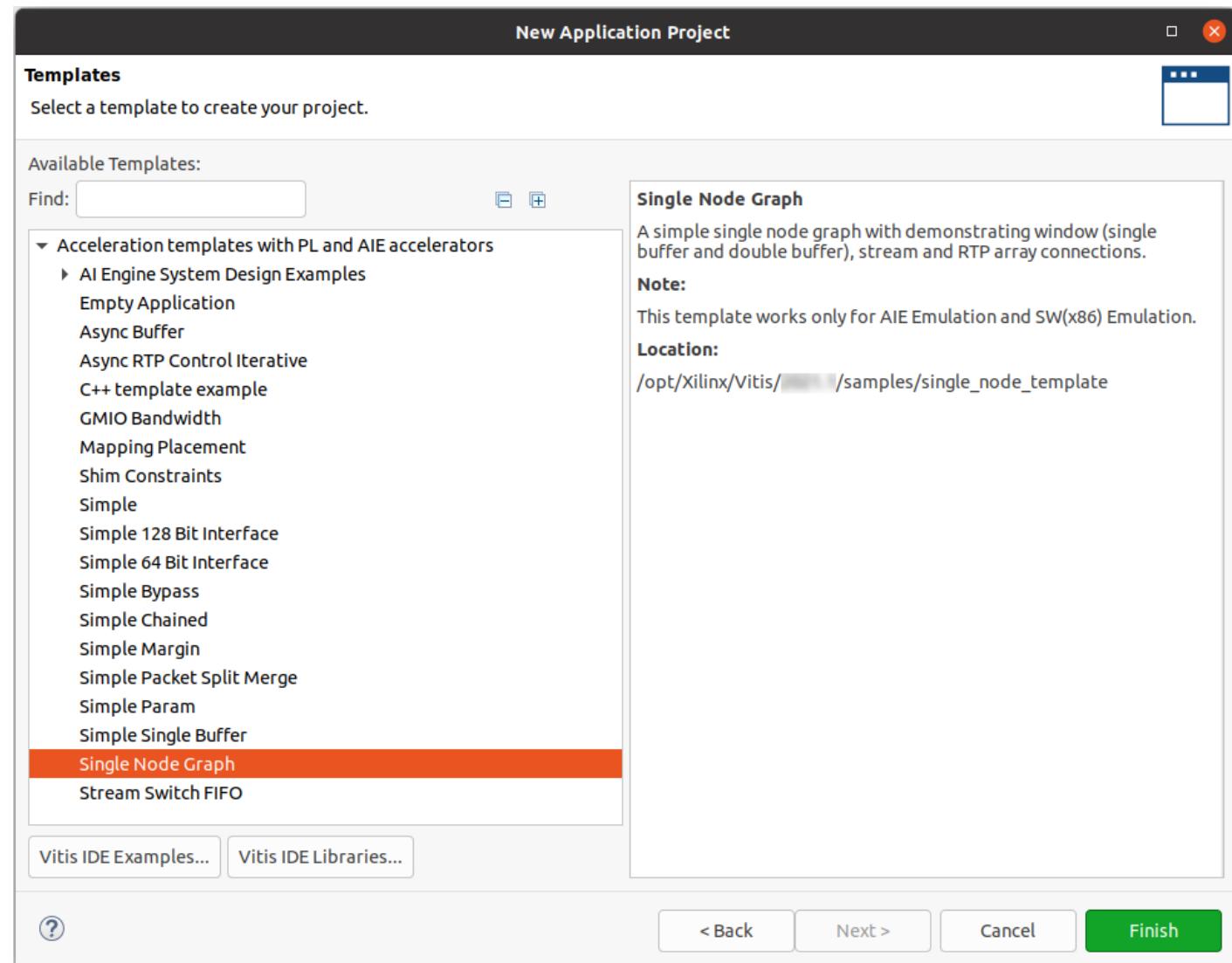
Domain selection

*Currently a single
AI Engine domain
is available*



Template Selection

- Various templates available
- Feature examples
- “Single Node Graph”
 - Single kernel programming example



Starting Point

Three projects created:

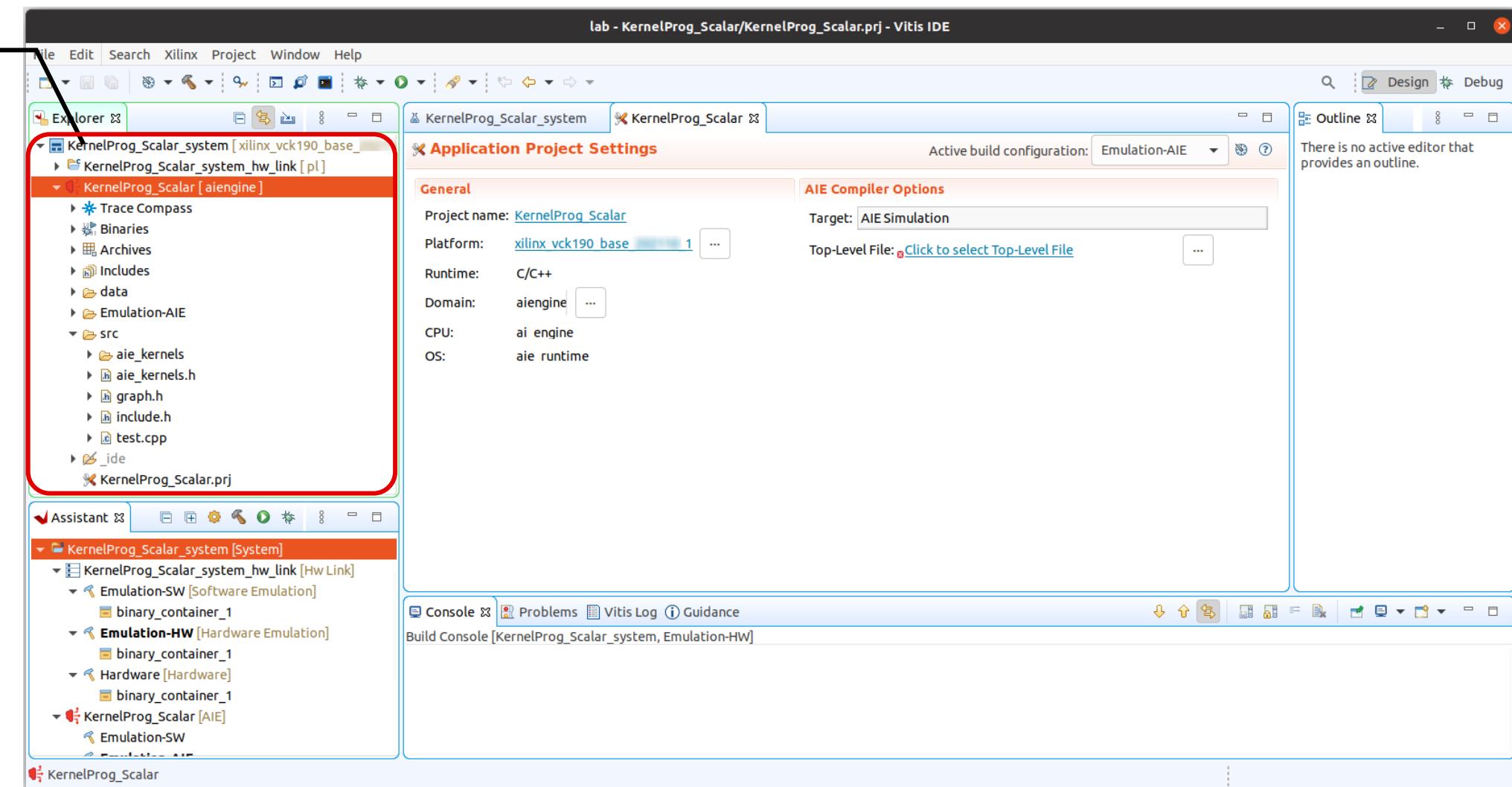
- System
- AI Engine
- Hardware link

System Project

In charge of global
(AI Engine + PS + PL)
application simulation
and implementation

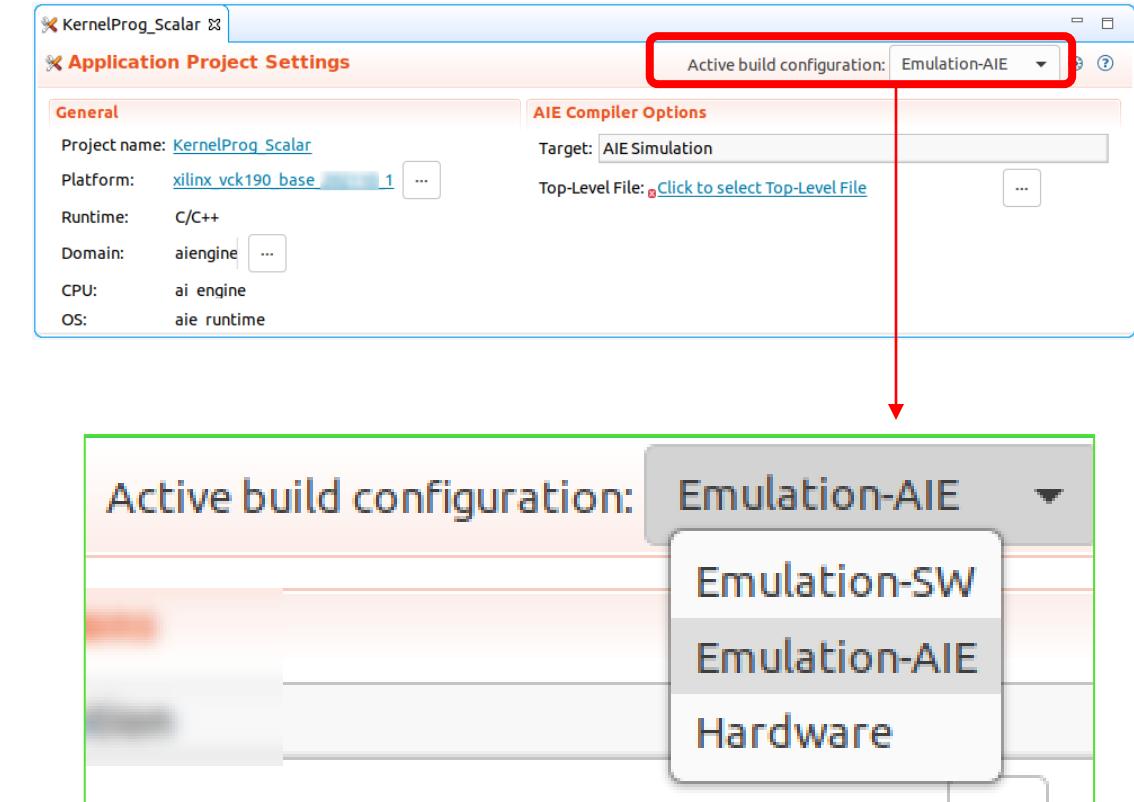
Hardware Link

In charge of linking
The AI Engine and PL
for simulation
and implementation



AI Engine Project View – Build Configurations

- Emulation-SW
 - Pure functional simulation
 - AIE program is compiled by the AI Engine compiler and runs in the x86 simulator
- Emulation-AIE
 - Verifies hardware functional correctness
 - AIE program is compiled by the AI Engine compiler and runs in the AI Engine SystemC simulator (aiesimulator)
- Hardware
 - Hardware build compiled by the AI Engine compiler for use in the actual device



Vitis IDE Debugging for AI Engine

- Debugging AIE kernels
 - Pipeline view
 - Vector Register View
 - Internal Memory view East, West North South

F

R

M

Disassembly

Enter location here

```
00000000000000452: NO
00000000000000454: NO
00000000000000460: weighted_sum_with_margin:
00000000000000460: PADD0 [p1], #24; MOV.s9 r9, #5; PADD0 [p0], #8; MOV.s12 r8, #3; MOV.s8 m0, #
00000000000000464: MOV.s10 r1, #-28; MOV.s9 r11, #2; NOP; MOV.s12 r13, #1; MOV.ch0, r1
0000000000000046C: LDA cb2, [p1], #12; LDB p2, [p0], #16; NOP; MOV.s12 r12, #6; MOV.ch1, r
00000000000000478: LDA cb1, [p0], #12; MOV.s9 r10, #7; NOP; MOV.s12 lc, #63; MOV.ch2, r1
00000000000000484: LDA cs1, [p0];
00000000000000488: LDA cs2, [p1], #-28; NOP; NOP; MOV.p3, p
0000000000000048C: LDA p1, [p1]; NOP; NOP; MOV.Q20 ls, #1312; NO
00000000000000490: LDA p2, [p0]; NOP; NOP; MOV.Q20 le, #1408; NO
00000000000000494: NO
00000000000000498: NO
000000000000004A2: NO
000000000000004A4: NO
000000000000004A8: NO
000000000000004B0: NO
000000000000004B2: NO
000000000000004B4: NO
000000000000004B6: NO
000000000000004B8: NO
000000000000004B9: NO
000000000000004BC: PADD0 [p2], m0, cyc1; NOP; NO
000000000000004C4: NO
```

1010 0101 wc0	00000000000000000000000000000000	0	WC0 register
1010 0101 WC1	00000000000000000000000000000000	0	WC1 register
1010 0101 wd0	00000000000000000000000000000000	0	WD0 register

Address	0 - 3	4 - 7	8 - B	C - F
000002000C880000	00000000	00000000	00000000	00000000
000002000C880010	00000000	00000000	00000000	00000000
000002000C880020	00000000	00000000	00000000	00000000

Kernel Programming: Code Restructuring and Directives

```

#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess_prepare_for_pipelineing
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input) );
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));

        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}

```

- Kernel program

Kernel Programming: Code Restructuring and Directives

```
#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess_prepare_for_pipelining
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input) );
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));

        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}
```

- Adaptive Data Flow library

Kernel Programming: Code Restructuring and Directives

```
#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess_prepare_for_pipelineing
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input) );
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));

        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}
```

- Kernel
 - C/C++ function
 - Special I/O and vector data types.
 - Launched automatically by a scheduler depending on some events
 - Operates on data streams

Kernel Programming: Code Restructuring and Directives

```
#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess_prepare_for_pipelineing
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input) );
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));

        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}
```

- Vector data types for vectorized computations
- Multiple lanes of data processed independently
- Predefined in the AIE tools

Kernel Programming: Code Restructuring and Directives

```
#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess_prepare_for_pipelineing
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input));
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}
```

- C Window API to access data
- Used by kernel to operate on data windows
 - Input window and Output window

Kernel Programming: Code Restructuring and Directives

```

#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
    input_window<cint16> * w_input, output_window<cint16> * w_output){
v16int16 coeffs;
v32cint16 sbuff = undef_v32cint16();
for (unsigned i = 0; i < 12 ; i++)
    coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
const unsigned LSIZE = (samples / 4);
for ( unsigned i=0; i<LSIZE; i+=2)
chess_loop_range(2,)
chess_prepare_for_pipelineing
{
    v4cacc48 acc;
    sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
    sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
    sbuff = upd_w(sbuff, 2, window_read_v8(w_input));
    acc = mul4_sym(    sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
    acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
    window_writeincr(w_output, srs(acc, 0));

    acc = mul4_sym(    sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
    acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
    window_writeincr(w_output, srs(acc, 0));
    window_decr_v8(w_input,1);
}

```

- AI Engine intrinsics to perform vectorized computation
- E.g., mul and mac with pre addition

Kernel Programming: Code Restructuring and Directives

```
#include <adf.h>
void fir_16taps_symm(const unsigned samples, const int32 (&taps_in)[16],
                      input_window<cint16> * w_input, output_window<cint16> * w_output){
    v16int16 coeffs;
    v32cint16 sbuff = undef_v32cint16();
    for (unsigned i = 0; i < 12 ; i++)
        coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
    const unsigned LSIZE = (samples / 4);
    for ( unsigned i=0; i<LSIZE; i+=2)
        chess_loop_range(2,)
        chess prepare for pipelining
    {
        v4cacc48 acc;
        sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
        sbuff = upd_w(sbuff, 2, window_read_v8(w_input) );
        acc = mul4_sym(      sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));

        acc = mul4_sym(      sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
        acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
        window_writeincr(w_output, srs(acc, 0));
        window_decr_v8(w_input,1);
    }
}
```

- Directives
- Help with scheduling for optimal performance
- chess_loop_range
 - Minimum loop execution
- chess_prepare_for_pipelining
 - Compiler will try to fill processor pipeline with useful instructions

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

- Adaptive Data Flow library

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

    FirstGraph(){
        pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
        pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

        k = kernel::create(fir_16taps_symm);

        connect< window<128> > net0 (pl_in.out[0], k.in[0]);
        connect< window<128> > net1 (k.out[0], pl_out.in[0]);

        source(k) = "kernels/k.cc";
        runtime<ratio>(k) = 0.1;
    }
};
```

- AI Engine application described as dataflow graph
- The constructor of the graph describes all the connections and some other parameters
- For single kernel programming this section is very simple

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};

};
```

- Single kernel based graph

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

- Graph Input/output ports
- For multiple I/O ports define them as array. E.g.,

```
input_plio pl_in[N];
output_plio pl_out[N];
```

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};

};
```

Creation of source and sink

1. External port name
2. Predefine connection
3. I/O vector file for emulation
4. Optional, PL frequency

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

- Define the connection of elements in the graph
- Index of the array is the position of the kernel argument
- Independent index for input and output

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

- Specify the kernel function

Describing the Kernel Environment

```
#include <adf.h>
using namespace adf;
#include "kernels.h"

class FirstGraph : public graph {
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

FirstGraph(){
    pl_in = input_plio::create("In", plio_32_bits, "in.txt", 250.0);
    pl_out = output_plio::create("Out", plio_32_bits, "out.txt", 250.0);

    k = kernel::create(fir_16taps_symm);

    connect< window<128> > net0 (pl_in.out[0], k.in[0]);
    connect< window<128> > net1 (k.out[0], pl_out.in[0]);

    source(k) = "kernels/k.cc";
    runtime<ratio>(k) = 0.1;
}
};
```

- Define kernel source code
- Define kernel budget

Testbench

```
#include "project.h"
using namespace adf;

FirstGraph mygraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {

    mygraph.init();
    mygraph.run(4);
    mygraph.wait(10000);
    mygraph.resume();
    mygraph.update();
    mygraph.end();

    return 0;
}
#endif
```

Testbench

```
#include "project.h"
using namespace adf;
```

```
FirstGraph mygraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {

    mygraph.init();
    mygraph.run(4);
    mygraph.wait(10000);
    mygraph.resume();
    mygraph.update();
    mygraph.end();

    return 0;
}
#endif
```

- Adaptive Data Flow library

Testbench

```
#include "project.h"
using namespace adf;

FirstGraph mygraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {

    mygraph.init();
    mygraph.run(4);
    mygraph.wait(10000);
    mygraph.resume();
    mygraph.update();
    mygraph.end();

    return 0;
}
#endif
```

- Instance of your AIE graph

Testbench

```
#include "project.h"
using namespace adf;

FirstGraph mygraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {

    mygraph.init();
    mygraph.run(4);
    mygraph.wait(10000);
    mygraph.resume();
    mygraph.update();
    mygraph.end();

    return 0;
}
#endif
```

- Macro guard
- Only compile code when simulating

Testbench

```
#include "project.h"
using namespace adf;

FirstGraph mygraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {

    mygraph.init();
    mygraph.run(4);
    mygraph.wait(10000);
    mygraph.resume();
    mygraph.update();
    mygraph.end();

    return 0;
}
#endif
```

- Simulation control
- Init graph
- Run N times
- Wait X cycles
- Resume execution
- Update
- End simulation

Kernel Development and Validation: Vitis IDE

- Testbench required for:
 - Detailed cycle counts
 - Profiling
 - Low-level analysis
- Vitis IDE provides
 - In context simulation
 - Full AI Engine array access and PL connections

Example: Scalar and Vector Programming

Scalar Programming

```
void scalar_mul(input_window<int32>* __restrict data0,
                input_window<int32>* __restrict data1,
                output_window<int32>* __restrict out){
    for(unsigned int = 0; i < 512; i++){
        int32 a = window_readincr(data0);
        int32 b = window_readincr(data1);
        int32 c = a * b;
        window_writeincr(out, c);
    }
}
```

Cycles: 1053

The **`__restrict`** keyword allows for more aggressive compiler optimization by stating the independence between data

Vector Programming

```
void vector_mul(input_window<int32>* __restrict data0,
                input_window<int32>* __restrict data1,
                output_window<int32>* __restrict out){
    for(unsigned int = 0; i < 64; i++)
        chess_prepare_for_pipelineing
    {
        aie::vector<int32, 8> va = window_readincr_v<8>(data0);
        aie::vector<int32, 8> vb = window_readincr_v<8>(data1);
        aie::accum<acc80, 8> vt = aie::mul(va, vb);
        window_writeincr(out, vt.to_vector<int32>(0));
    }
}
```

Cycles: 98



- 10x speed up
- 8x throughput (mul)
- Pipeline optimization

Kernel Functional and Performance Validation

- Kernel Development
 - Single node development template
 - Or your own single node project
 - Required for profiling and low-level analysis
- Kernel Validation
 - Simple connection of the graph to the environment
 - In-context, full AI Engine array access, PL-connection, ...
 - Debug at the kernel level
- Single Kernel Optimization
 - Check out the documentation

Summary

- An AI Engine kernel is a C/C++ program written using AI Engine APIs or specialized intrinsic calls that target the very long instruction word (VLIW) vector and scalar processors
- The Vitis IDE tool can be used for single kernel development
- Kernel verification is performed using the ADF graph specification



The Programming Model: Introduction to the Adaptive Data Flow (ADF) Graph

Objectives

After completing this module, you will be able to:

- Identify various computation models
- Describe the basics of a data flow graph model for a concurrent computation model
- Describe graph input specifications, such as the number of platforms and ports and the Adaptive Data Flow (ADF) library

Models of Computation

Sequential Models

- Tasks are executed one by one
 - Turing Machine
 - Finite State Machine
 - Pushdown Automata

Functional Models

- Task execution will be implementation dependent
 - Lambda Calculus
 - Recursive Function
 - Combinatory Logic
 - Abstract Rewriting System

Concurrent Models

- Tasks are executed in parallel whenever possible
 - **Kahn Process Network**
 - Synchronous Data Flow
 - Actor Model
 - Cellular Automaton
 - Interaction Net
 - Petri Nets

Data Flow Graph - A representation of how inputs are processed

w



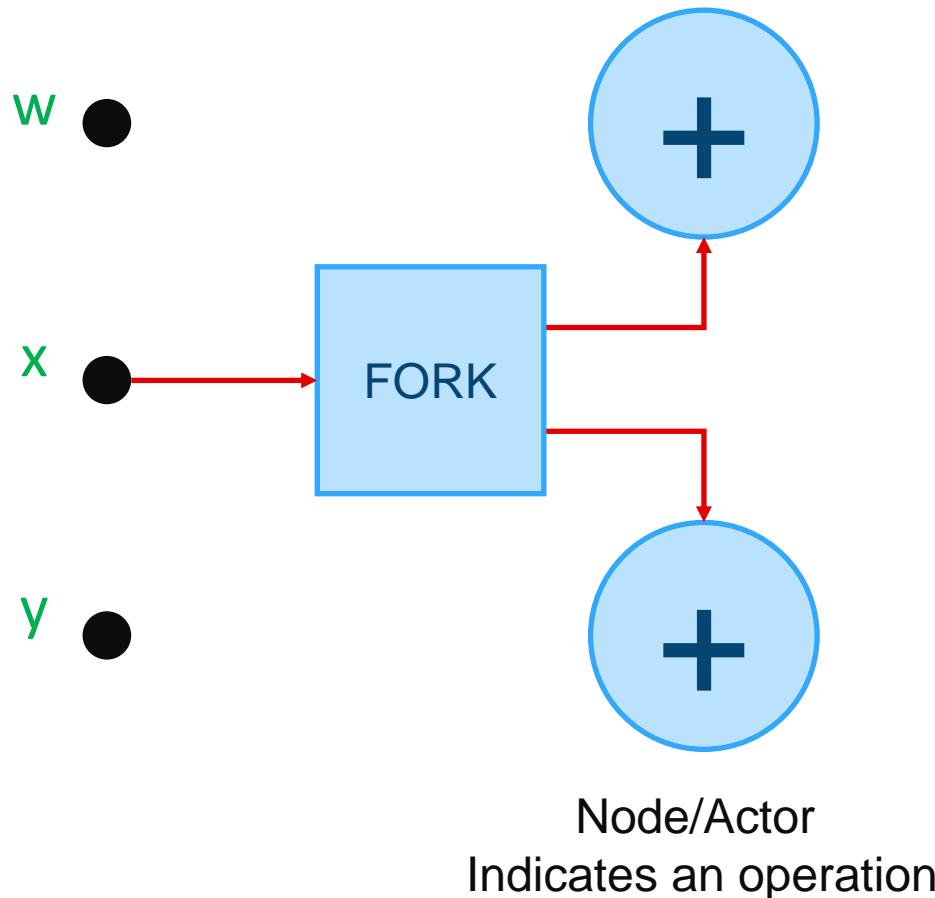
x



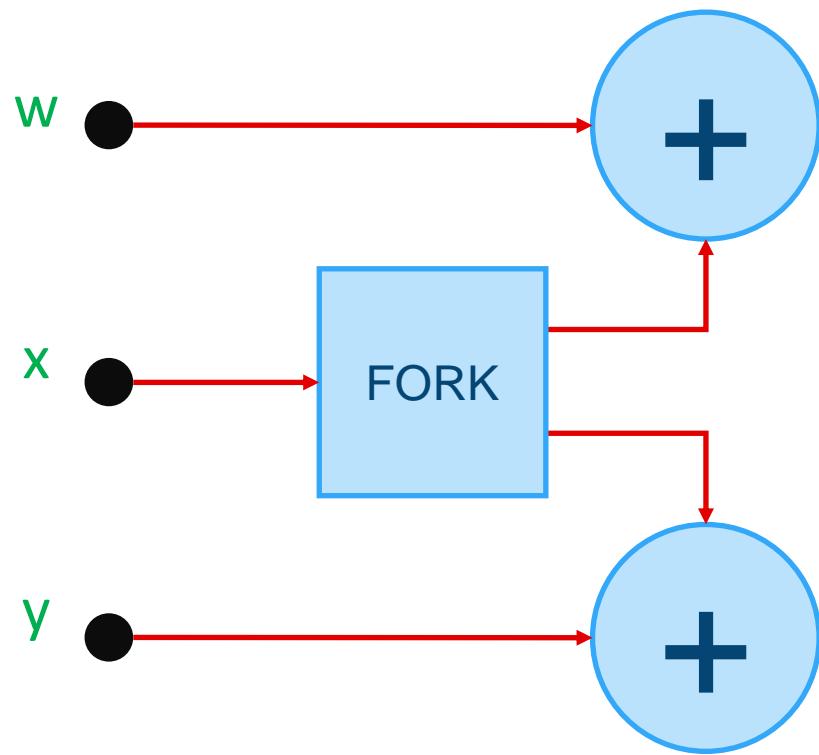
y



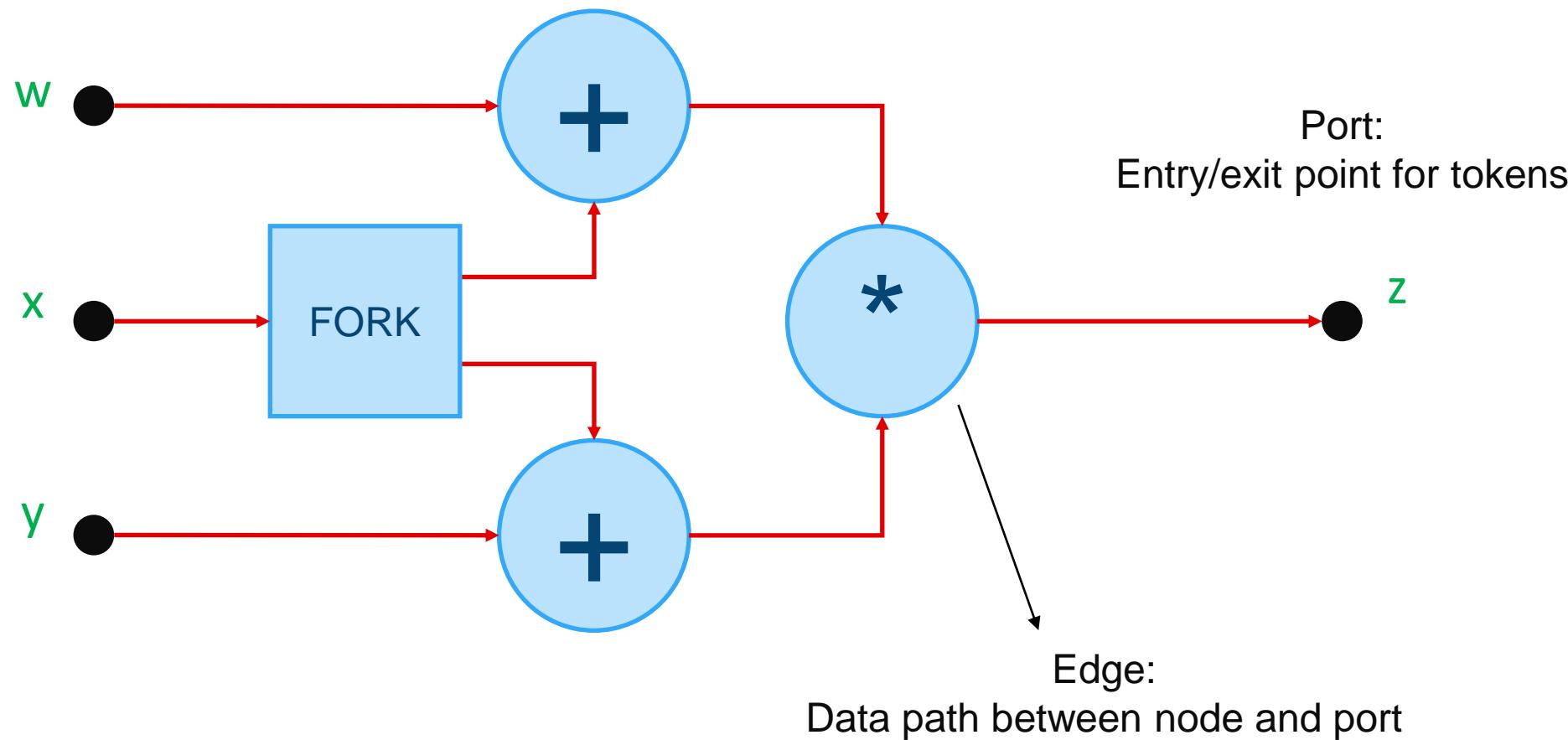
Data Flow Graph - A representation of how inputs are processed



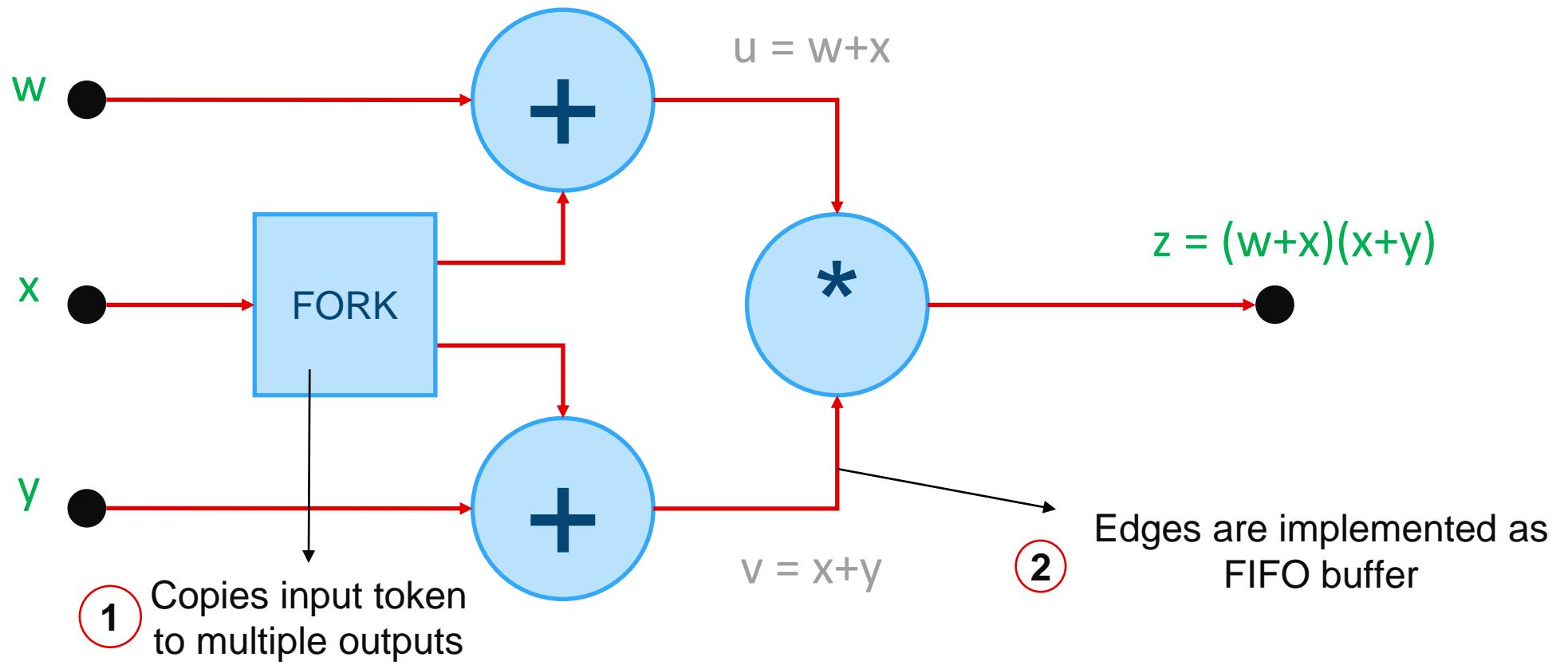
Data Flow Graph - A representation of how inputs are processed



Data Flow Graph - A representation of how inputs are processed

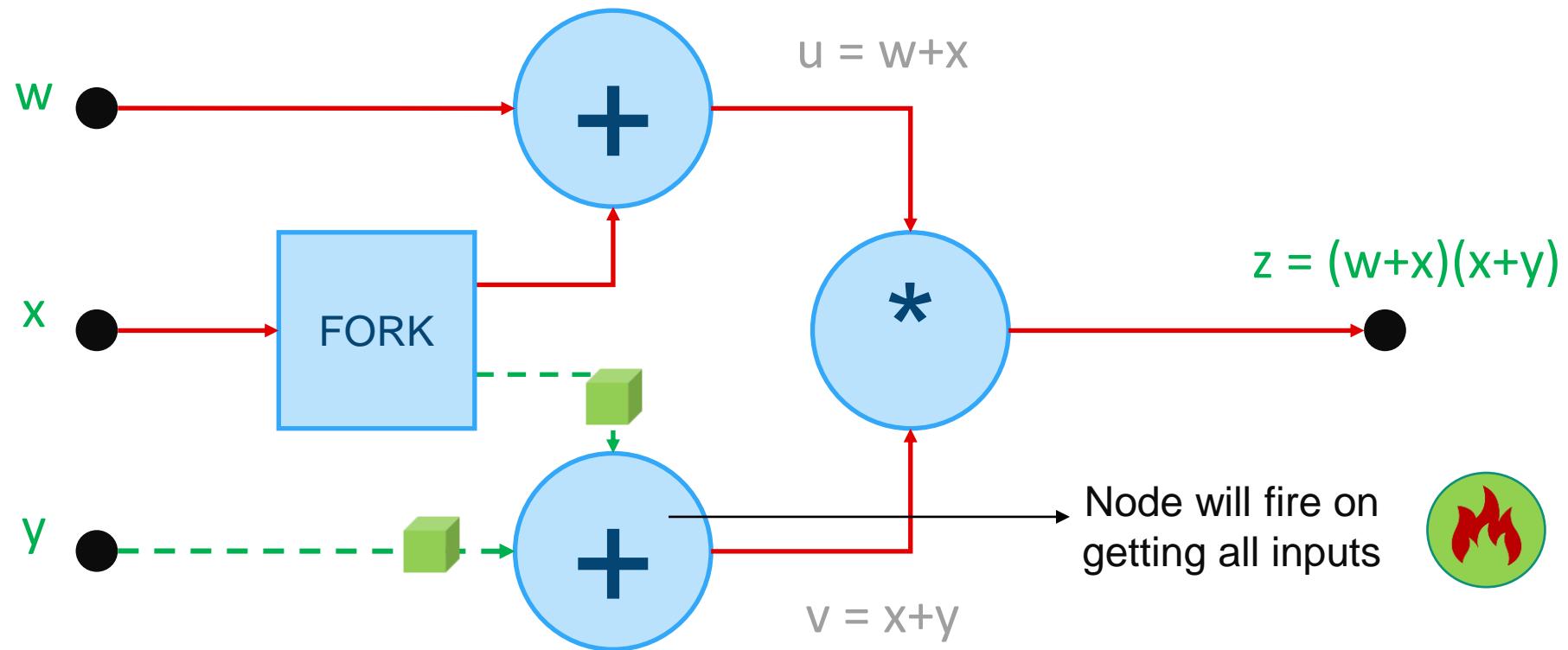


Data Flow Graph - A representation of how inputs are processed

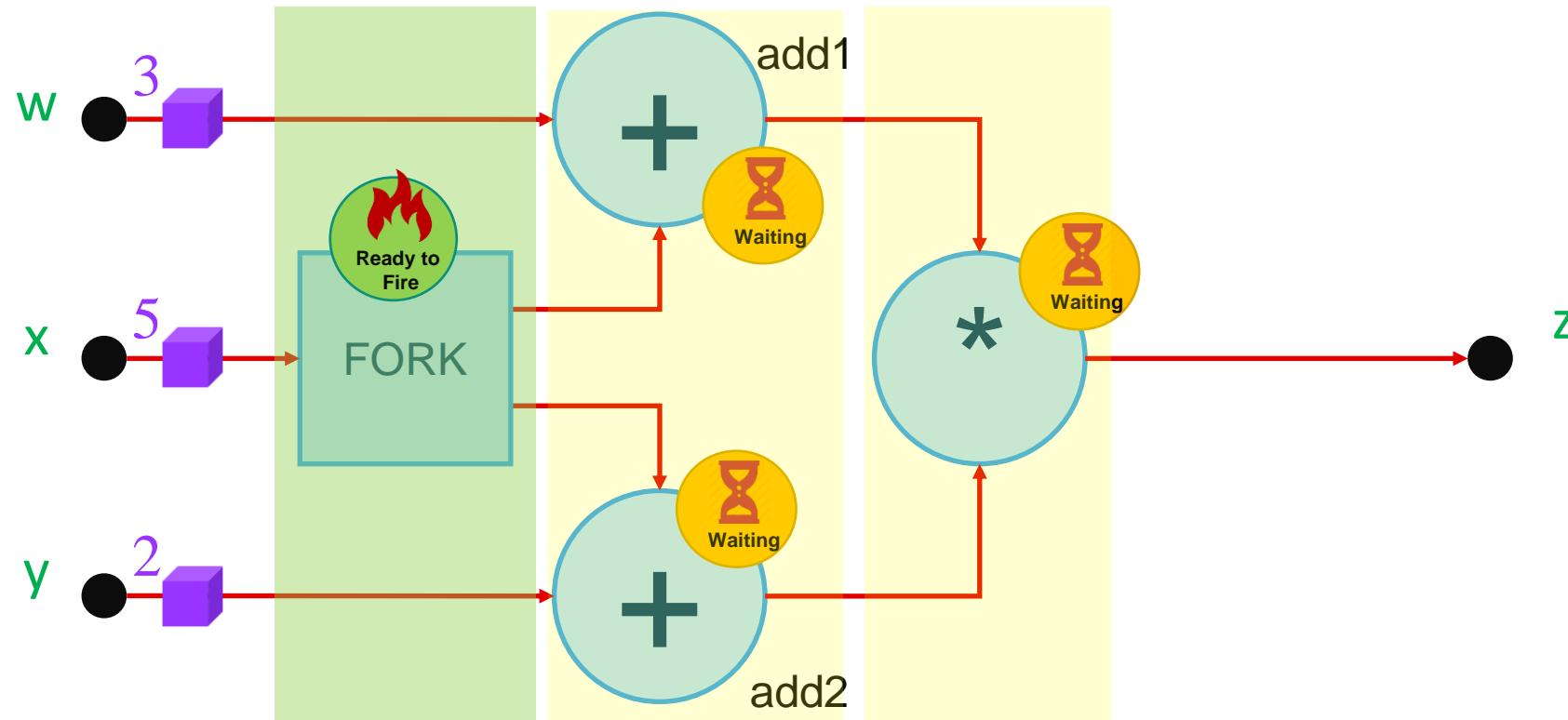


Indicates processing sequence, parallelism, and data dependence

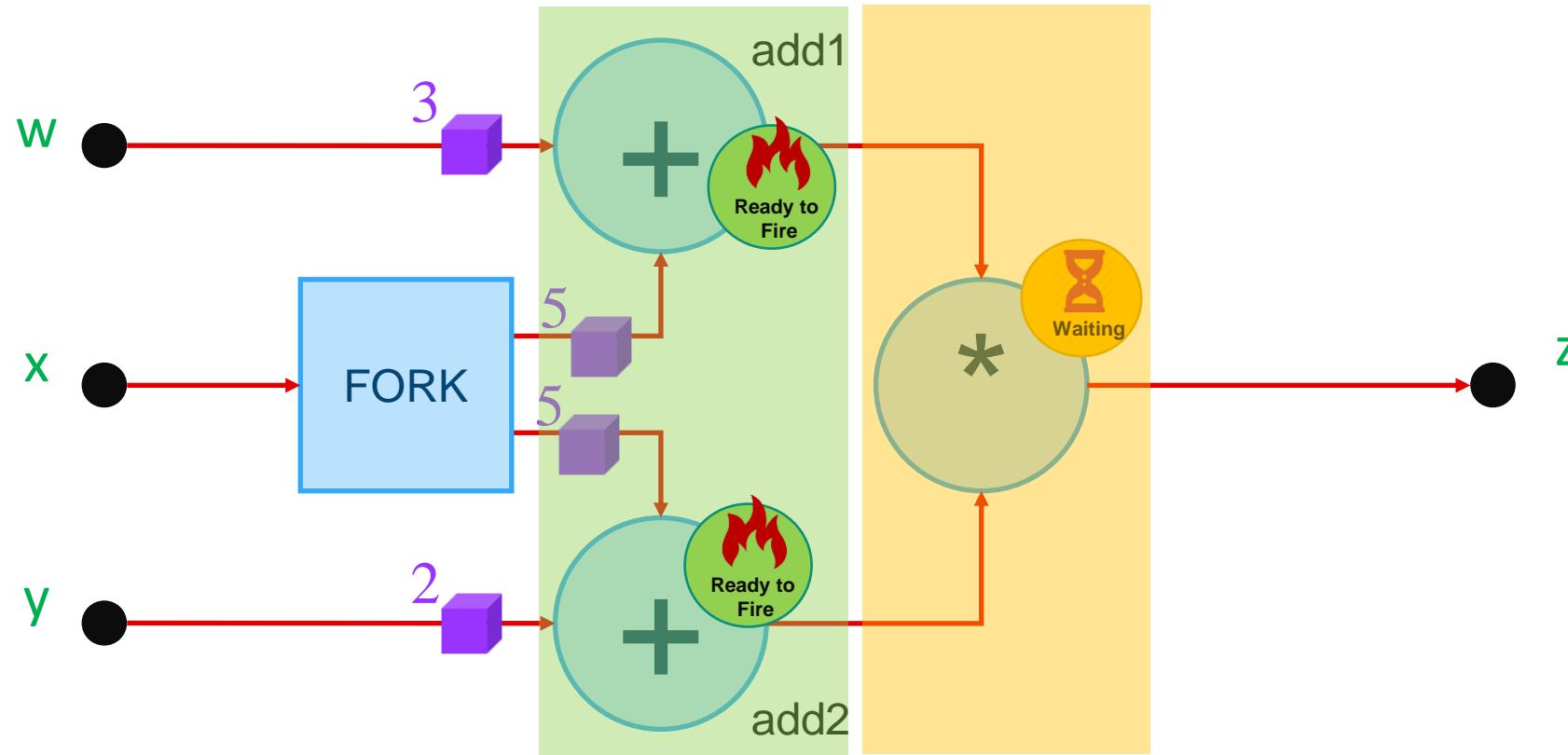
Data Flow Graph - A representation of how inputs are processed



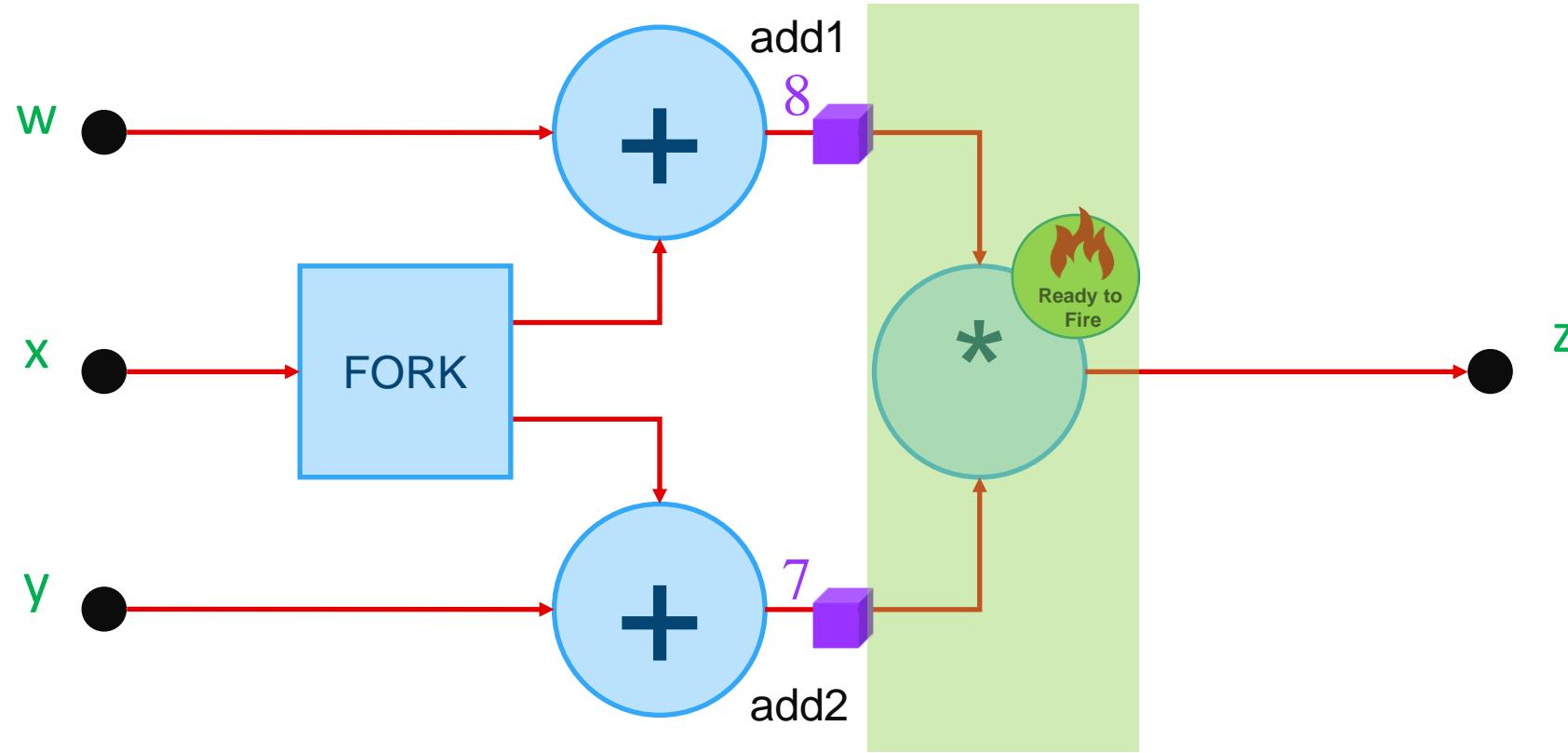
Data Flow Graph - Example



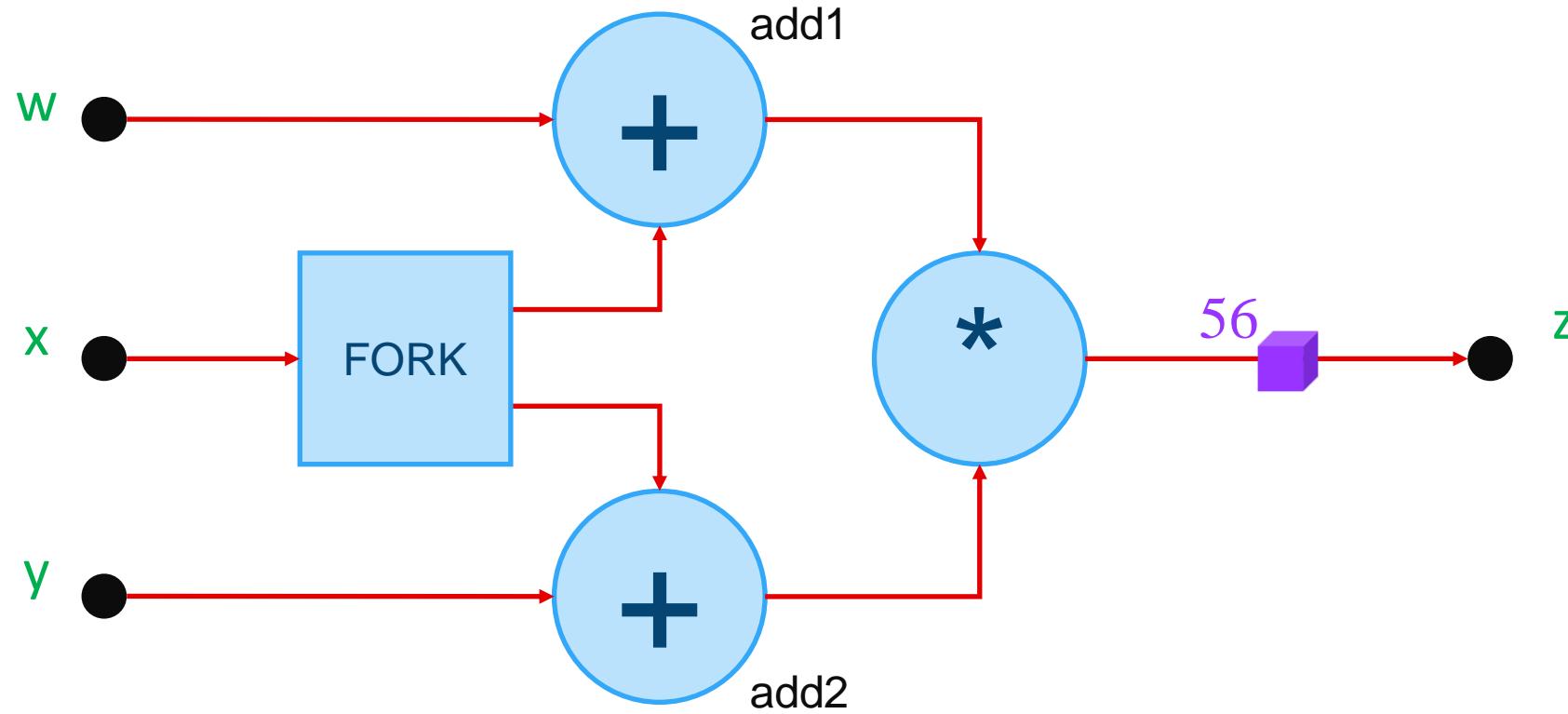
Data Flow Graph - Example



Data Flow Graph - Example

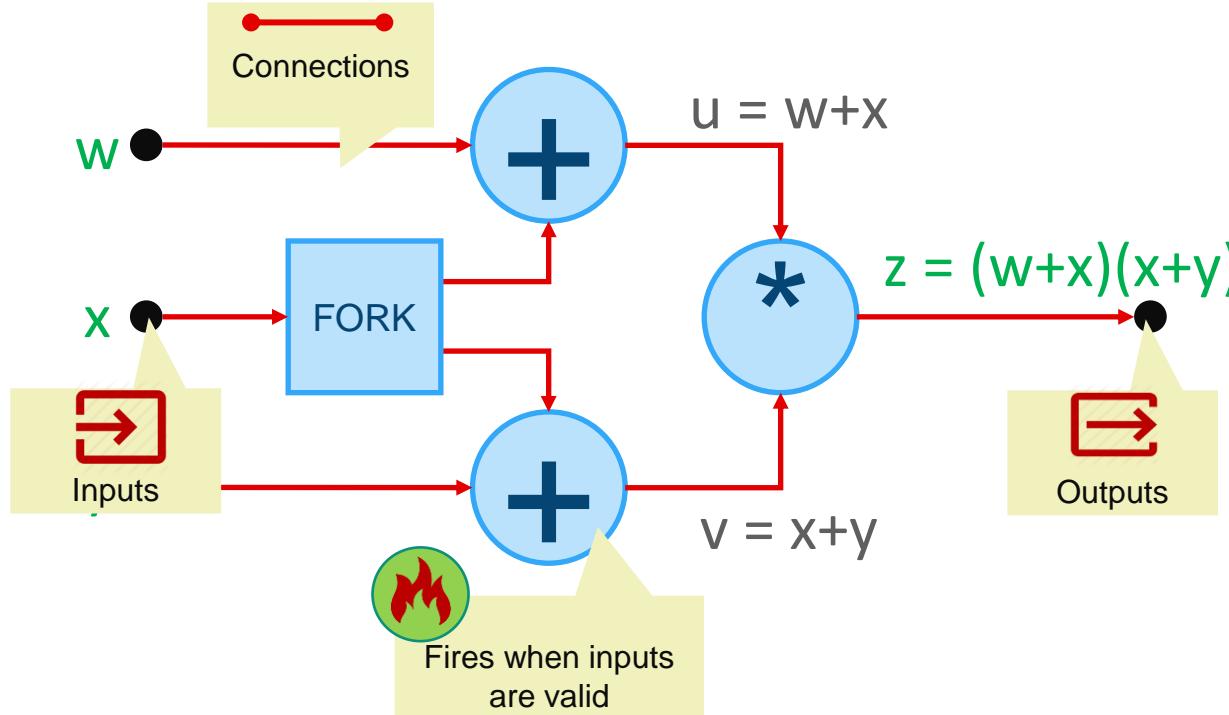


Data Flow Graph - Example



Data Flow Programming

Data flow programming – Focus on movement of data



Program – Series of sequential operations

Algorithm:

$$u = w+x;$$

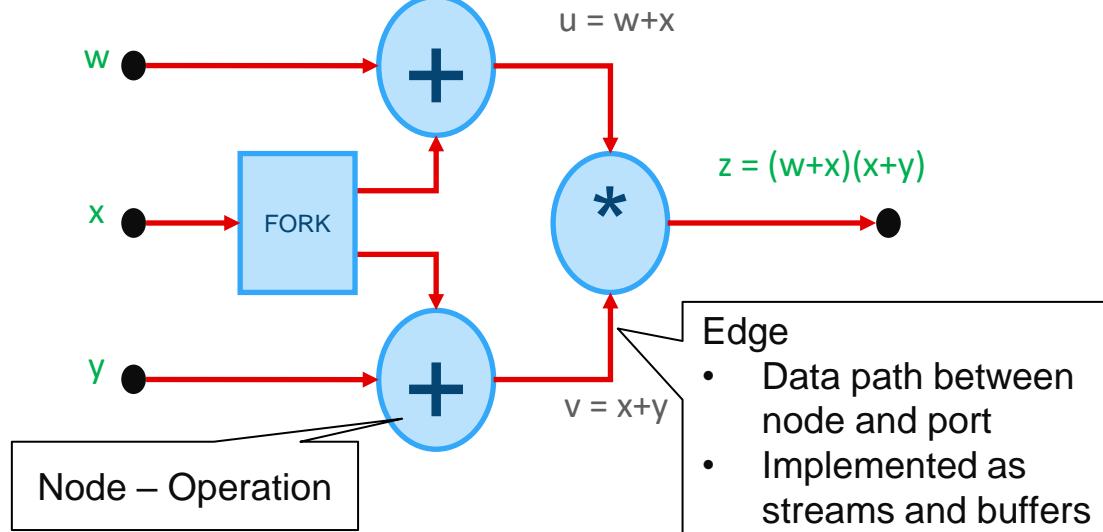
$$v = x+y;$$

$$z = (w+x)(x+y);$$

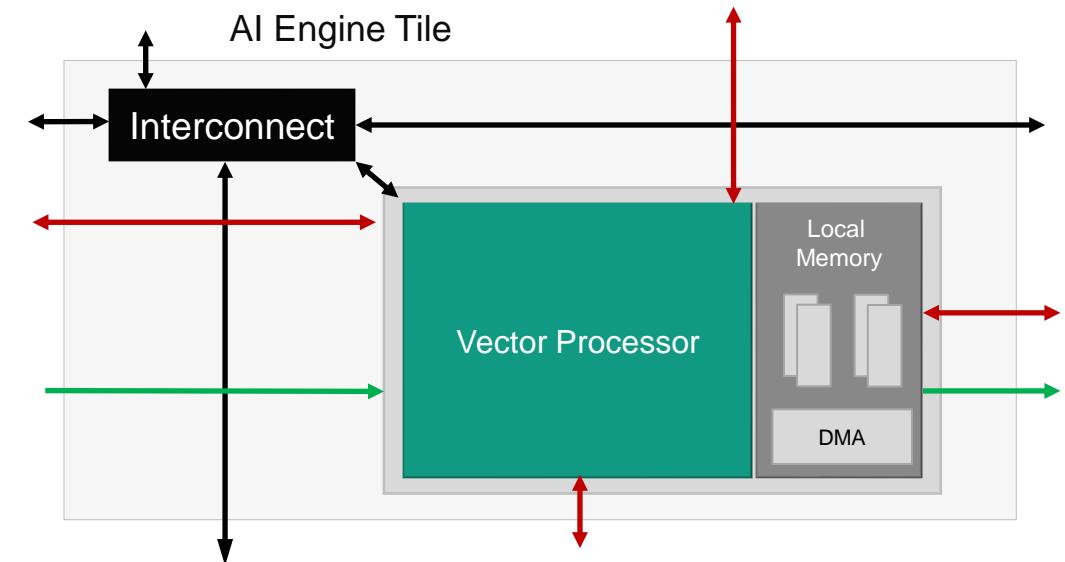
Data Flow Languages Inherently Parallel Ideal for Large Decentralized Systems

Data Flow Programming For AI Engines

- Data Flow Graph
 - Determines execution schedule
 - No “instruction pointer” to fire the AI Engines

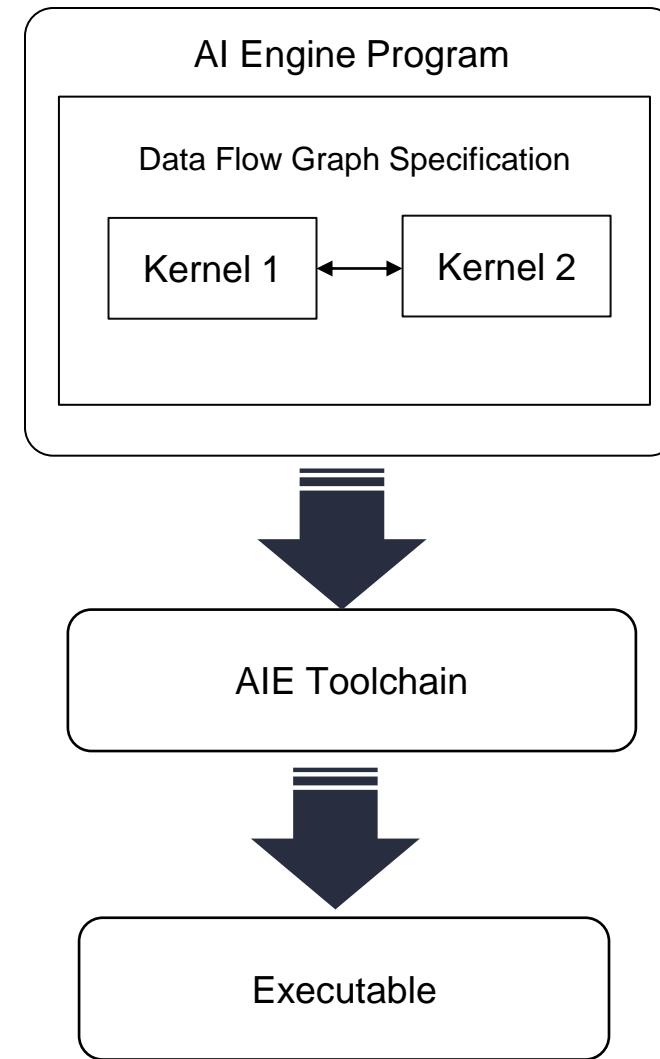


- AI Engine
 - Implements nodes/kernels
 - Operations with multiple operators
 - Can have multiple kernels
 - Instruction memory 16 KB
 - 8 to 400 AI Engine Tiles
 - AI Engines either computing or waiting for data

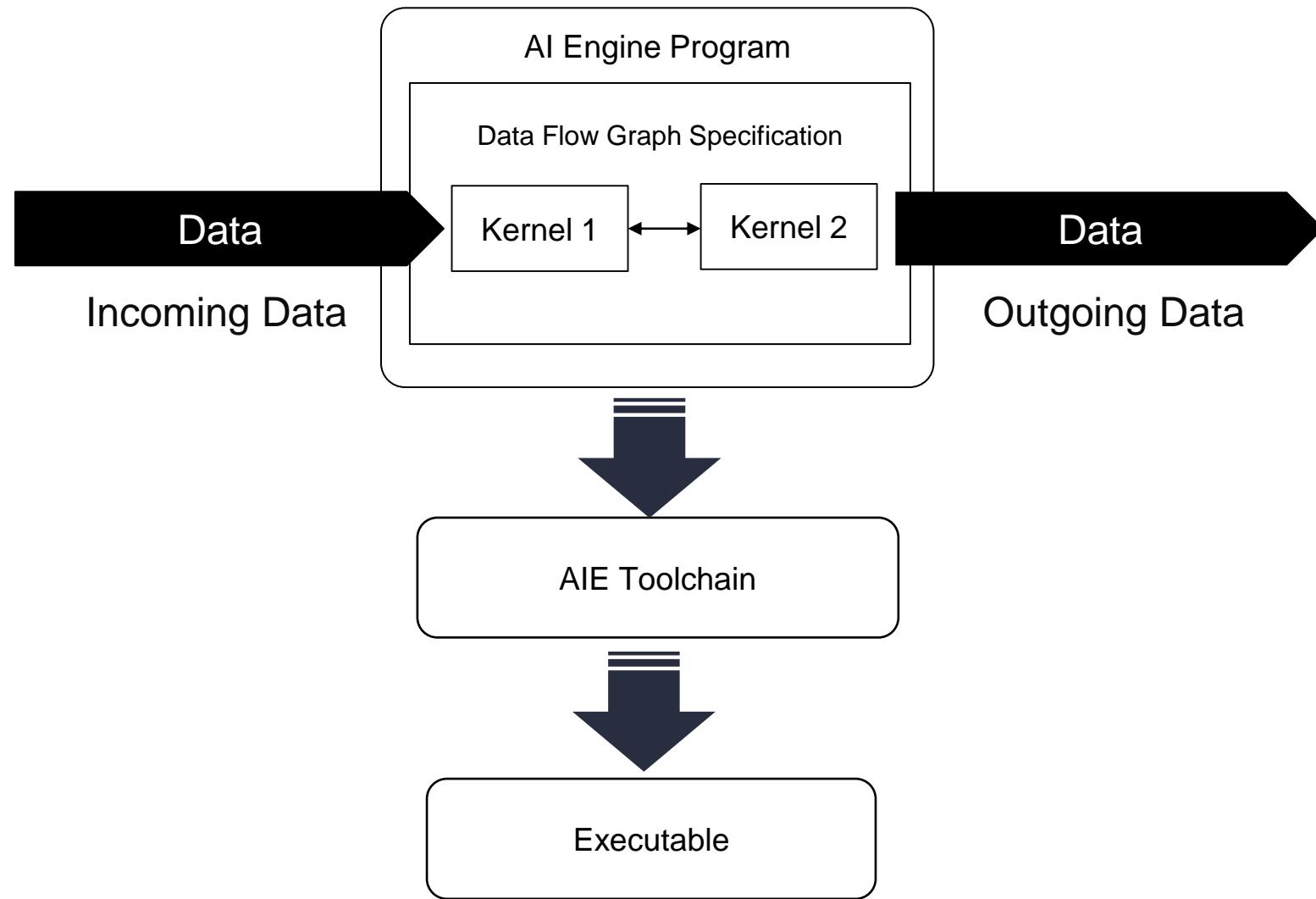


AI Engine Programming: Kernels

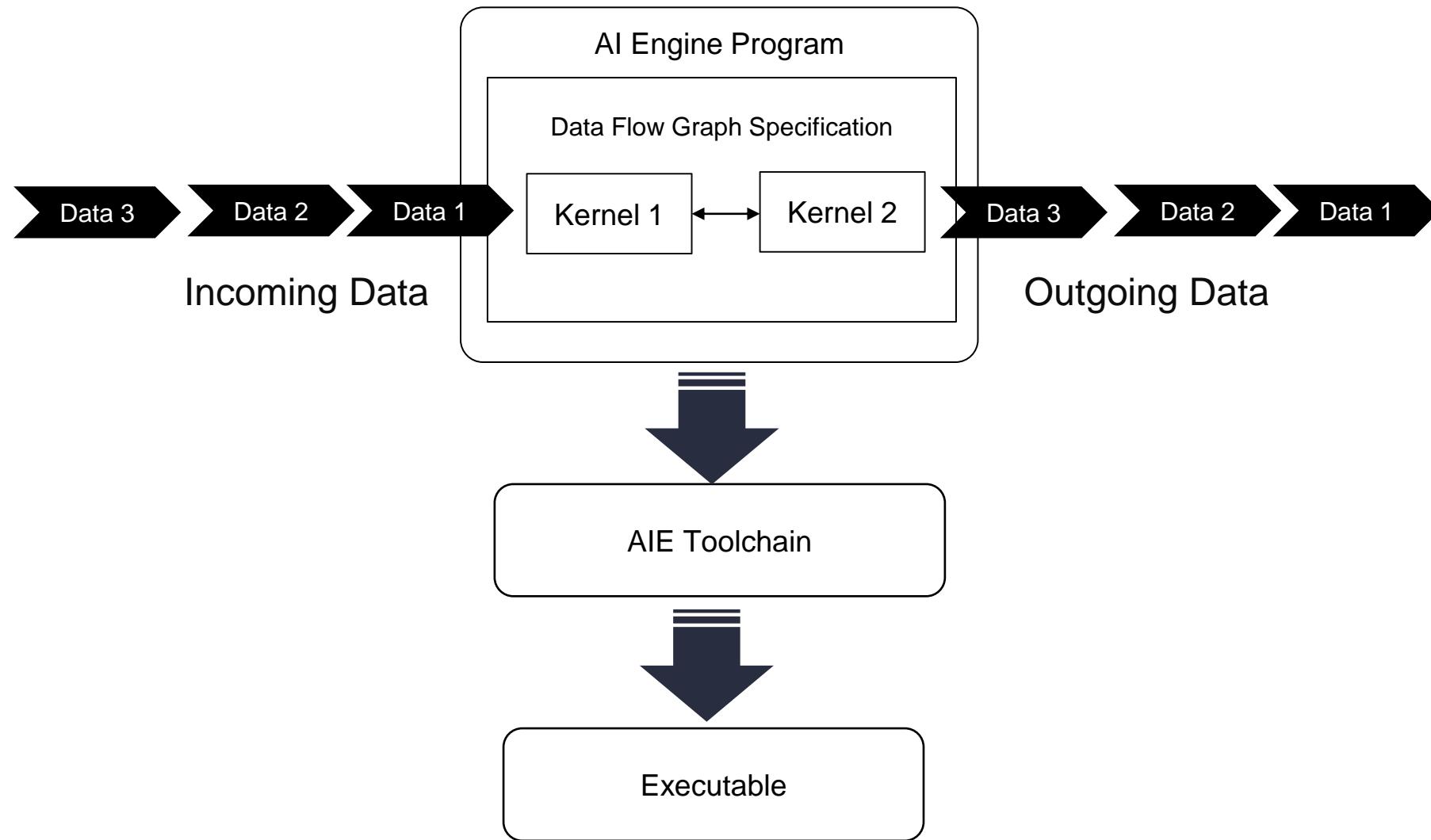
- Written in C/C++
- Declared as C/C++ functions that return void
- Unitary operation
- Frame-based communication using windows
- Sample-based communication using streams
- Leverage both Scalar and Vector processor



AI Engine Programming: Kernels

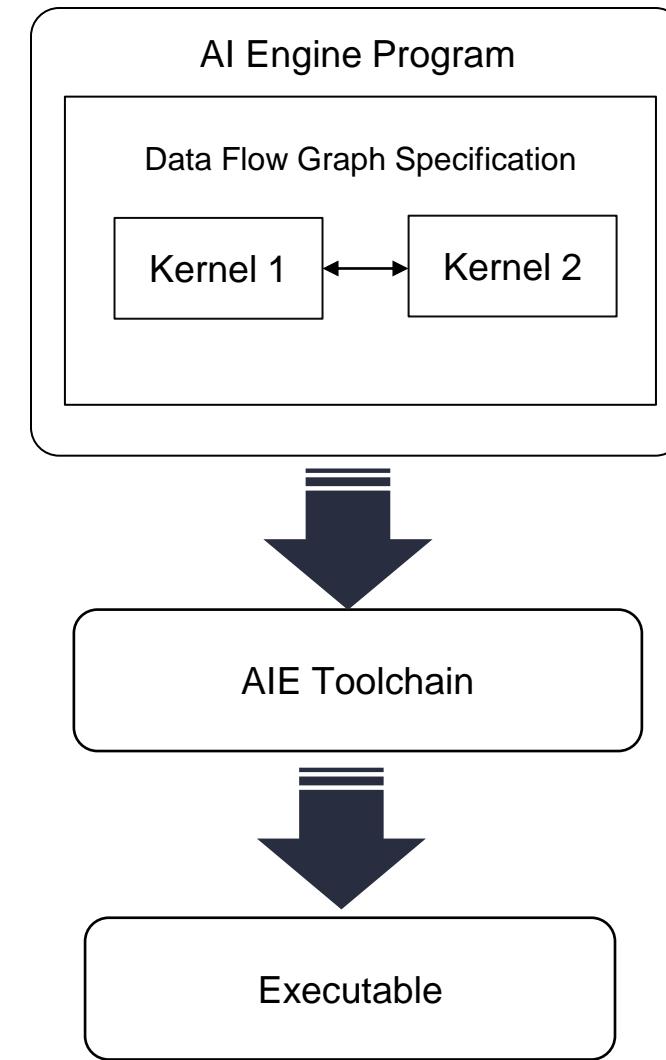


AI Engine Programming: Kernels



AI Engine Programming: Kernels

- Map multiple kernels onto one AIE Tile
 - Runtime ratio
- Split one kernel into multiple AIE is a manual process
 - Can leverage cascade stream
- Multiple functions can be mapped to single kernel
 - Reduces call overhead



Graph Input Specification Terminology

Platforms

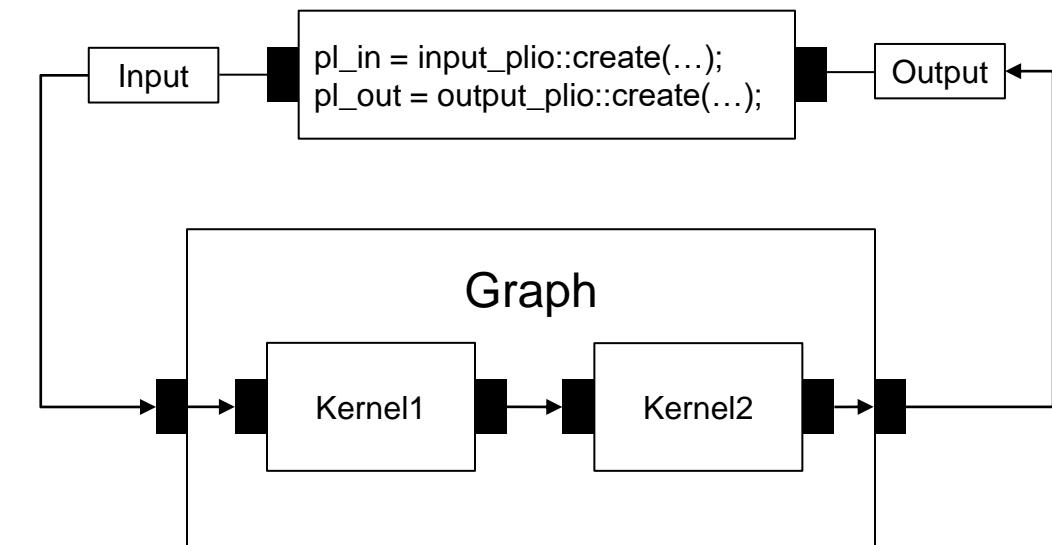
- Connects a data flow graph to a chosen target
- Specified for:
 - Simulation
 - Emulation
 - Actual hardware

Ports

- Input and output ports of the design
- Can have runtime parameters

Graph

- Defines all the required constructs for defining and executing graphs on AI Engines

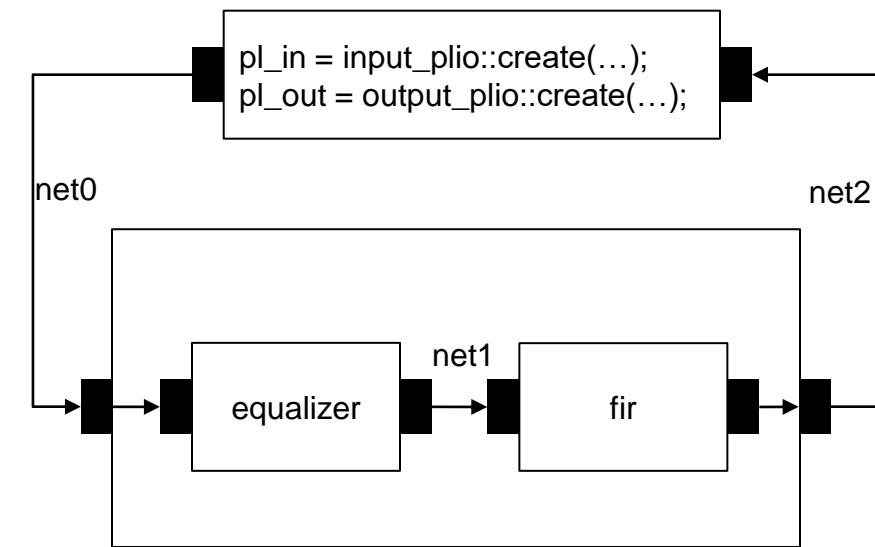


The Programming Model – Platform

```
pl_in = input_plio::create(plio_32_bits, "input.txt");
pl_out = output_plio::create(plio_32_bits, "output.txt");

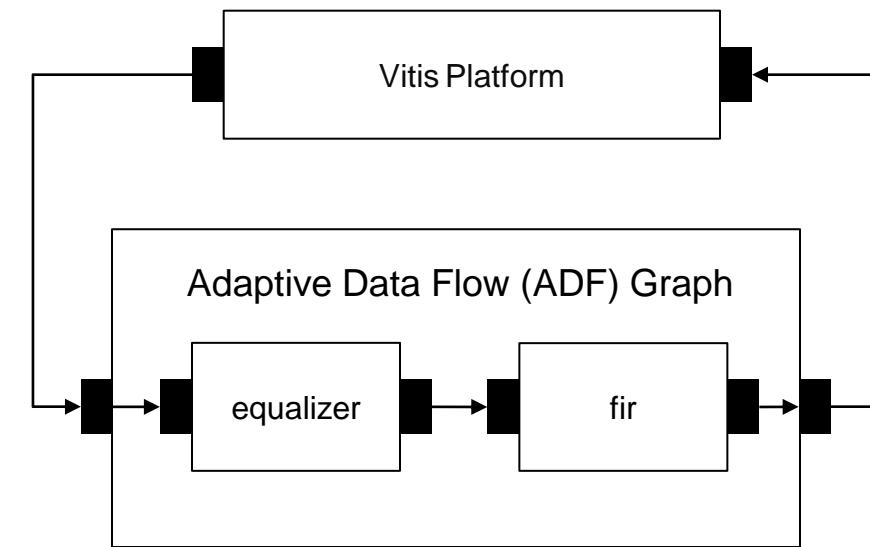
k1 = kernel::create(equalizer);
k2 = kernel::create(fir);

adf::connect<window<128>> net0(pl_in.out[0], k1.in[0]);
adf::connect<window<128>> net1(k1.in[0], k2.in[0]);
adf::connect<window<128>> net2(k2.out[0], pl_out.in[0]);
```



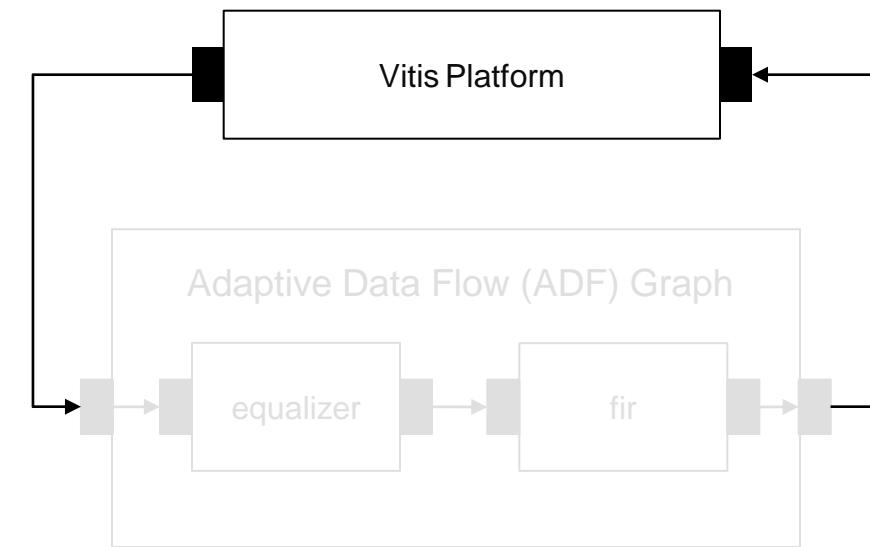
The Programming Model – Data Flow Graph

- Vitis Platform
 - For hardware builds
- I/O can connect to
 - C/C++ kernels
 - OpenCL kernels
 - RTL kernels
- Target platform using Vitis compiler



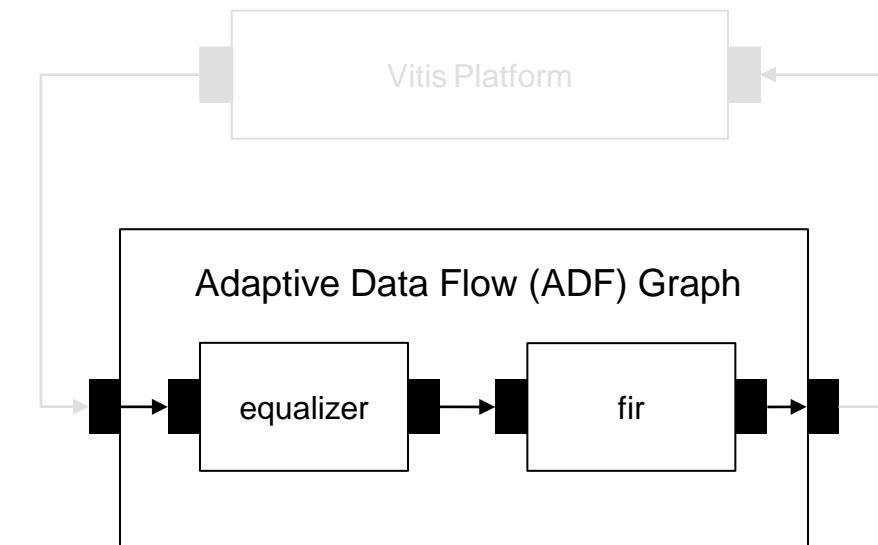
The Programming Model – Data Flow Graph

- Vitis Platform
 - For hardware builds
- I/O can connect to
 - C/C++ kernels
 - OpenCL kernels
 - RTL kernels
- Target platform using Vitis compiler



The Programming Model – Data Flow Graph

- Adaptive Data Flow (ADF) Graph
 - Derived from `adf::graph`
- Loaded once on reset or loaded dynamically
- Edges will be replaced by streams and buffer
- Use blocking read and non-blocking write
 - Potential deadlocks when using blocking read/write
- Model extensions for synchronized data, double buffers, etc
- Schedule, interleave, parallelize the processes in any order



The Programming Model – Kernel and Graph

- One kernel per source file
 - Reusability
 - Faster compilation
- Include all header files
 - Allows independent compilation
- Declare function prototype in an independent header file

```
#include <adf.h>
void weighted_sum(input_window<int32> * in,
                   output_window<int32> * out) {
    for (unsigned i = 0; i < NUM_SAMPLES; i++) {
        int32 val;
        int32 wsum = 0;
        window_decr(in, 7);
        for (unsigned j = 1; j <= 8; j++) {
            window_readincr(in, val);
            wsum = wsum + (j * val);
        }
        window_writeincr(out, wsum);
    }
}
```

The Programming Model – Kernel and Graph

- Each kernel invocation requires an input window (or stream) of data
- Each kernel invocation produces an input window (or stream) of data
- `.init()`
 - Loads the graph
- `.run()`
 - Execute the graph forever
 - Can pass number of invocations
- `.end()`
 - Terminate graph

```
#include "kernels.h"
#include "graph.h"

// instantiate ADF to compute weighted moving
average
GraphScalar Sgraph;

// initialize and run the data flow graph
int main(void) {
    Sgraph.init();
    Sgraph.run();
    Sgraph.end();
    return 0;
}
```

Summary

- Nodes or kernels are computation functions
 - Fundamental building block of the data flow graph specification
- Edges are the output from the nodes that can be implemented by streams and buffers
- In an Adaptive Data Flow (ADF) graph
 - Compute (nodes or kernels) blocks and the data flow (edges) are explicitly defined
 - Loaded once on reset
- Each kernel should be defined in its own source file



Vitis Analyzer

AMD
together we advance_

Objectives

After completing this module, you will be able to:

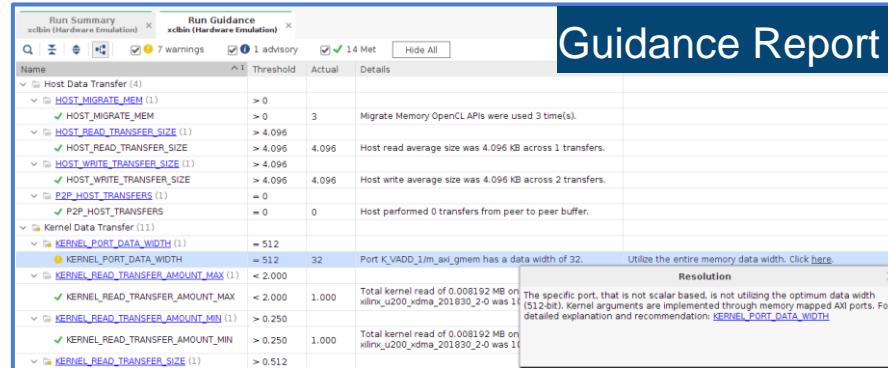
- Analyze the reports generated by the AI Engine tools that are part of the Vitis™ core development kit
- Use the reports to identify system performance and kernel performance bottlenecks

Vitis Reports

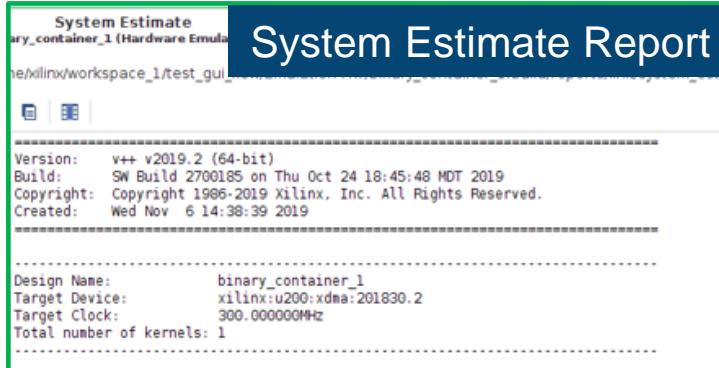
- Vitis generates various reports and logs
- Vitis HLS Compiler
- Vivado® Design Suite
- AIE Tools

Vitis Reports

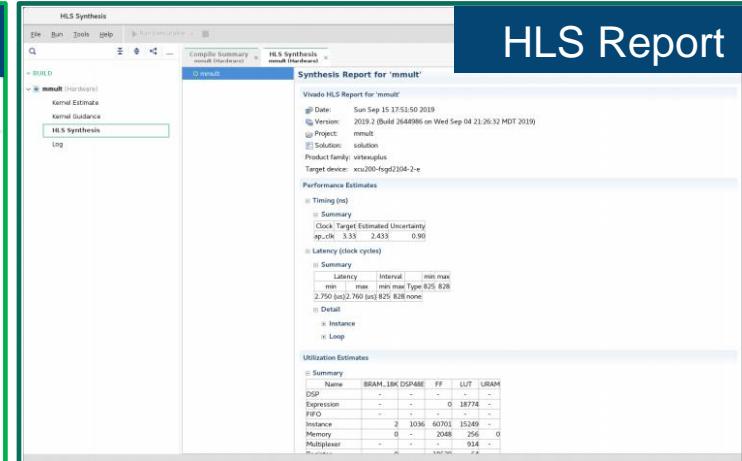
Guidance Report



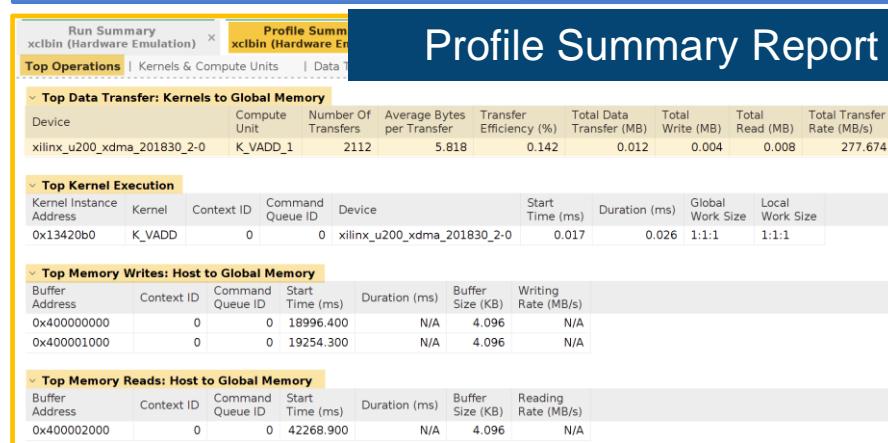
System Estimate Report



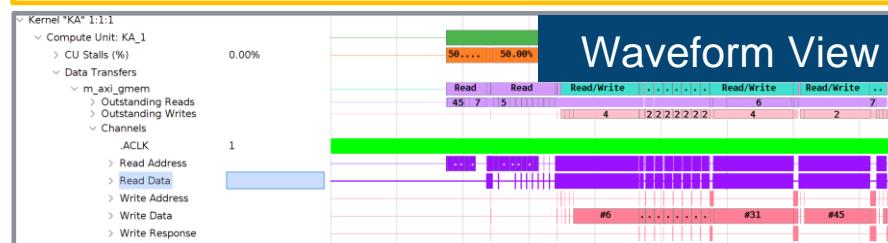
HLS Report



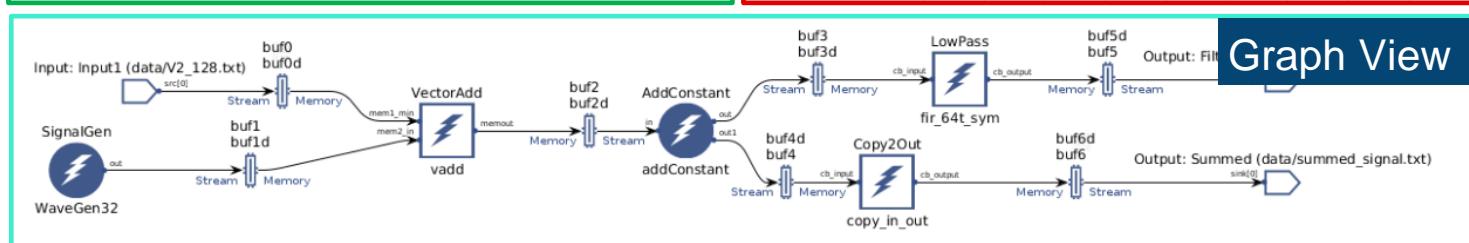
Profile Summary Report



Waveform View



Graph View

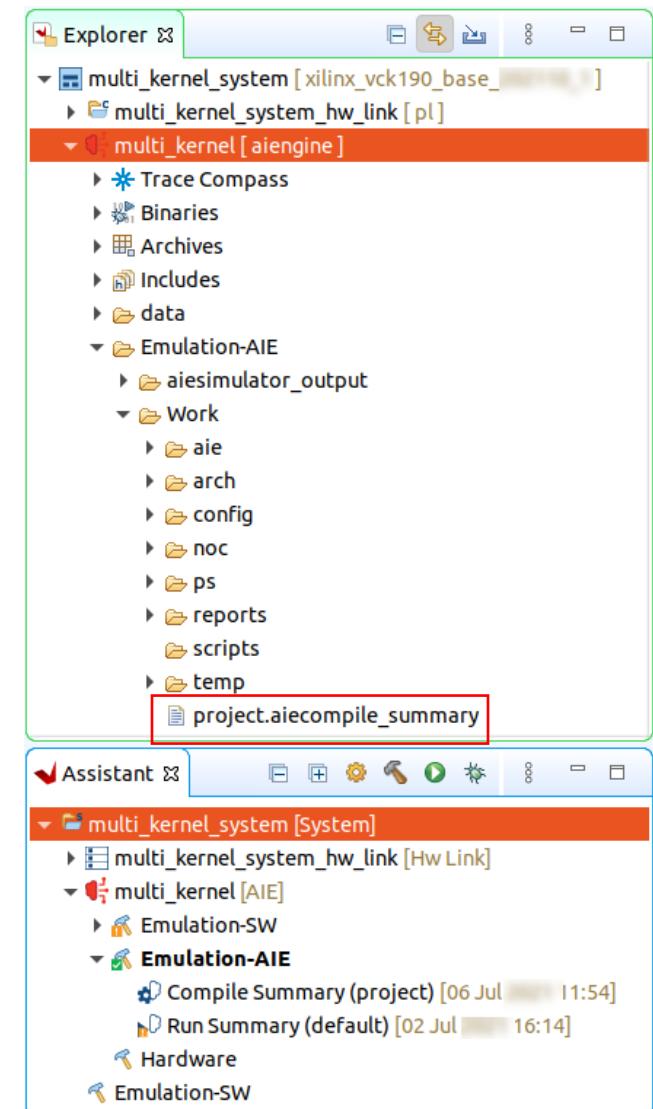


Vitis Analyzer

- View and analyze reports
 - Build reports
 - Application run reports
 - Run reports generated by XRT
- Compile Summary Report
 - For individual kernels
- Link Summary Report
 - For FPGA binary (XCLBIN)
- Run Summary Report
 - For profiling data
 - Related to application execution

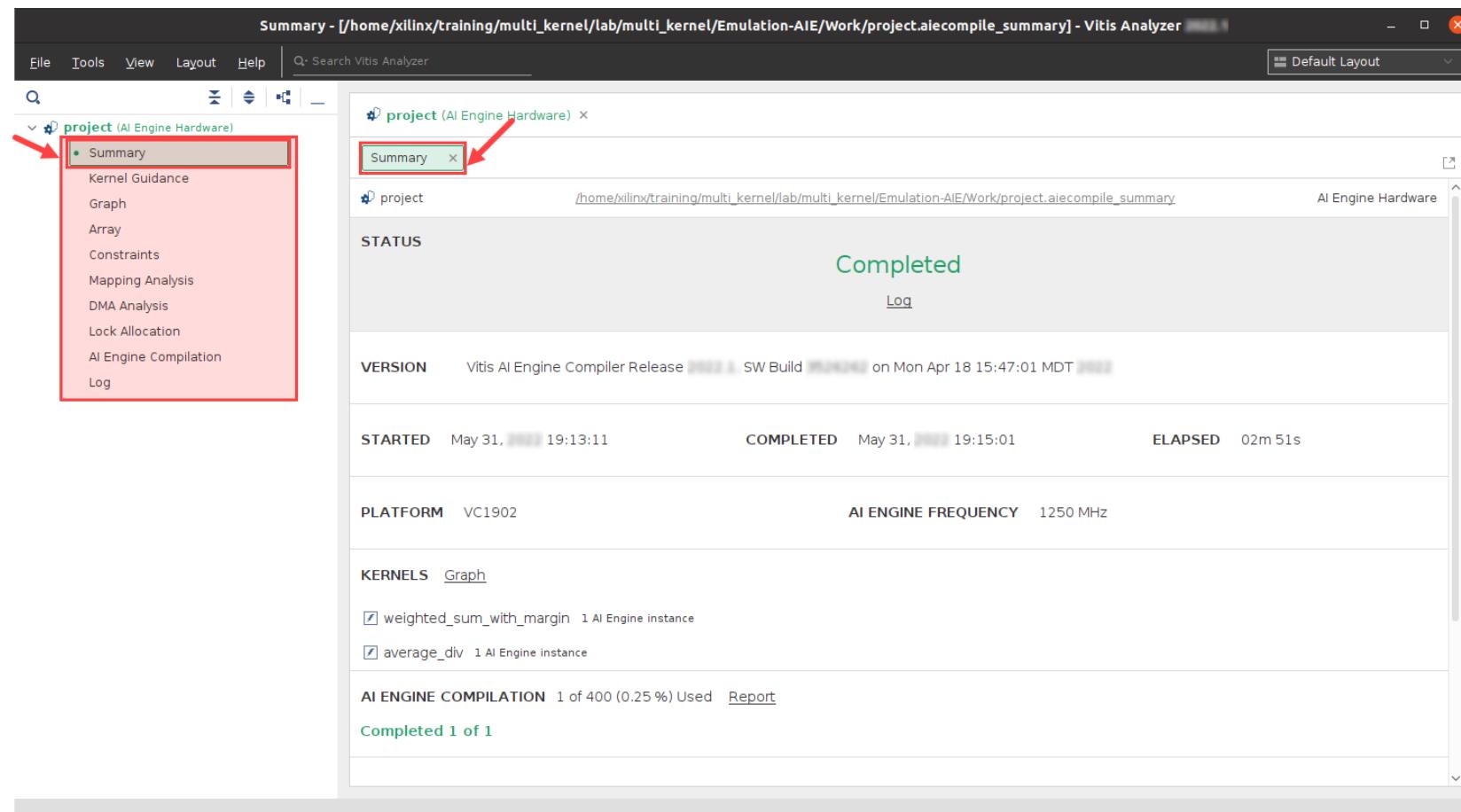
Vitis Analyzer – Compile Summary

- Compile Summary report generated after the compilation of the AI Engine graph
- AI Engine compiler writes a summary of compilation results:
 - project.aiecompile_summary
- Reports include:
 - Summary
 - Kernel Guidance
 - Graph
 - Array
 - Constraints
 - Mapping Analysis
 - DMA Analysis
 - Lock Allocation
 - Log
 - AI Engine Compilation



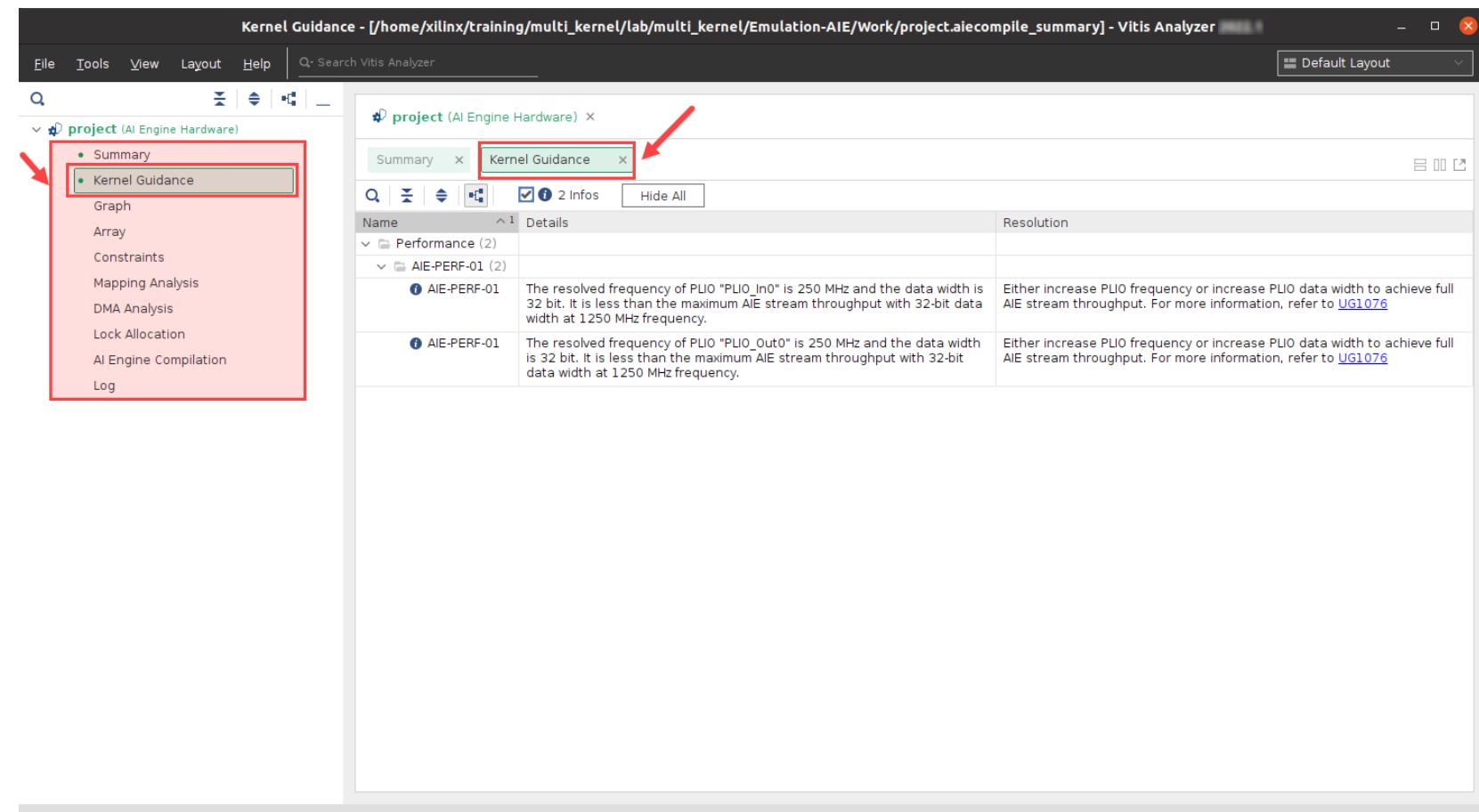
Compile Report: Summary

- Contains collection of reports and diagrams
- Written to the working directory of the aiecompiler as specified by the --workdir option



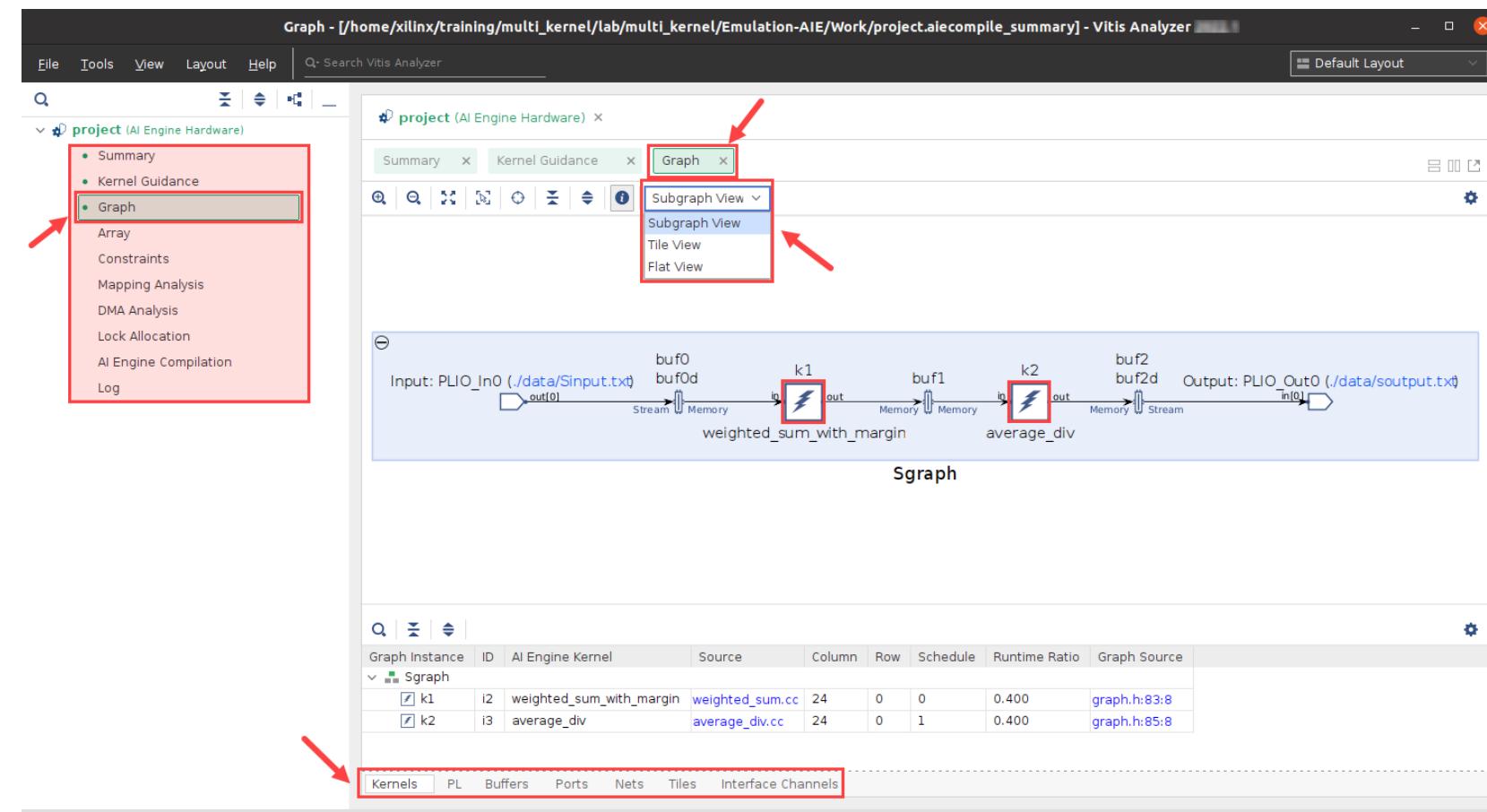
Compile Report: Kernel Guidance

- Messages to provide guidance on kernel optimization



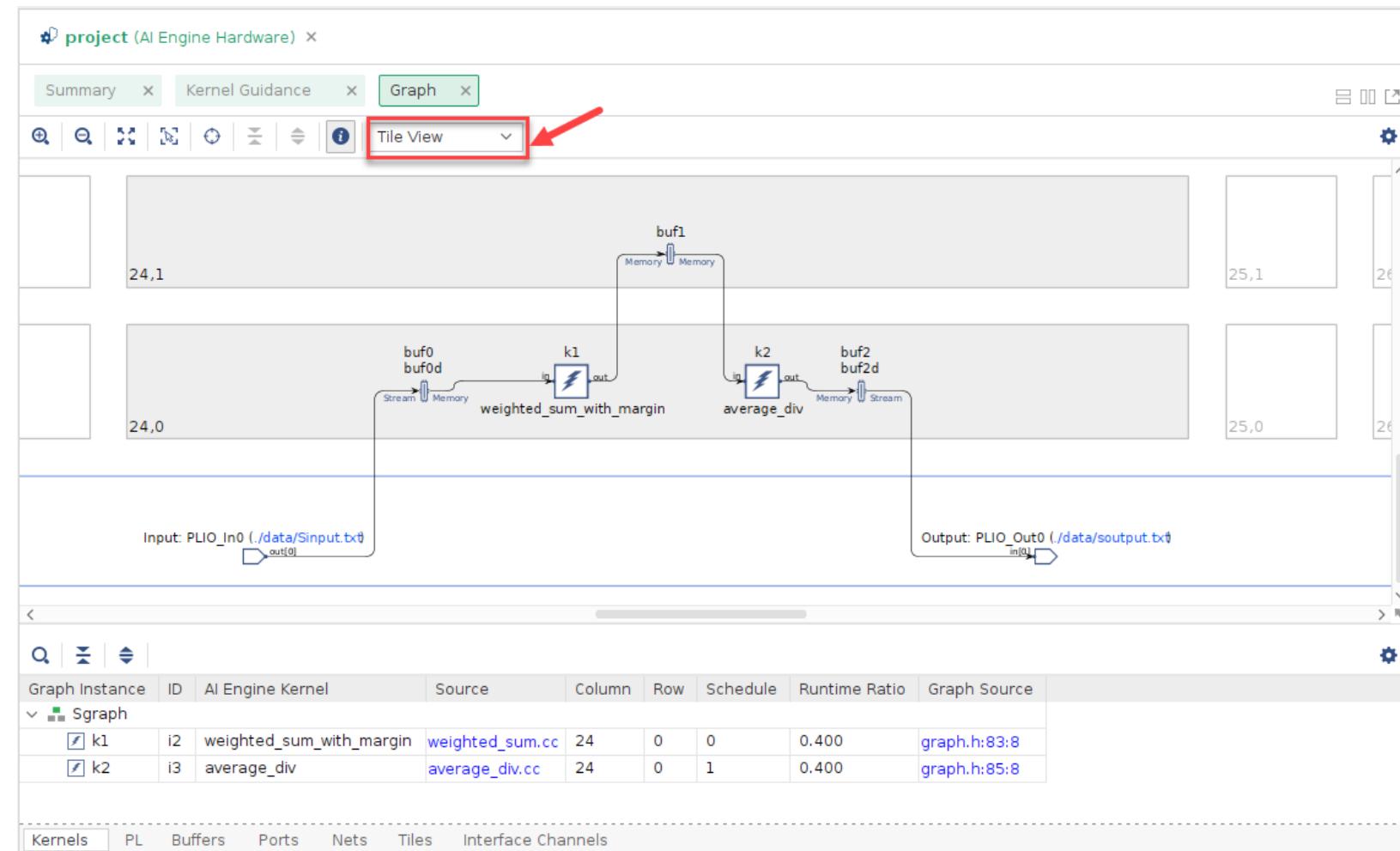
Compile Report: Graph View

- Graph provides a flow diagram of the AI Engine graph
- Shows all the kernels defined in the AI Engine graph
- Layouts used to view the graph:
 - **Subgraph View**
 - Tile View
 - Flat View



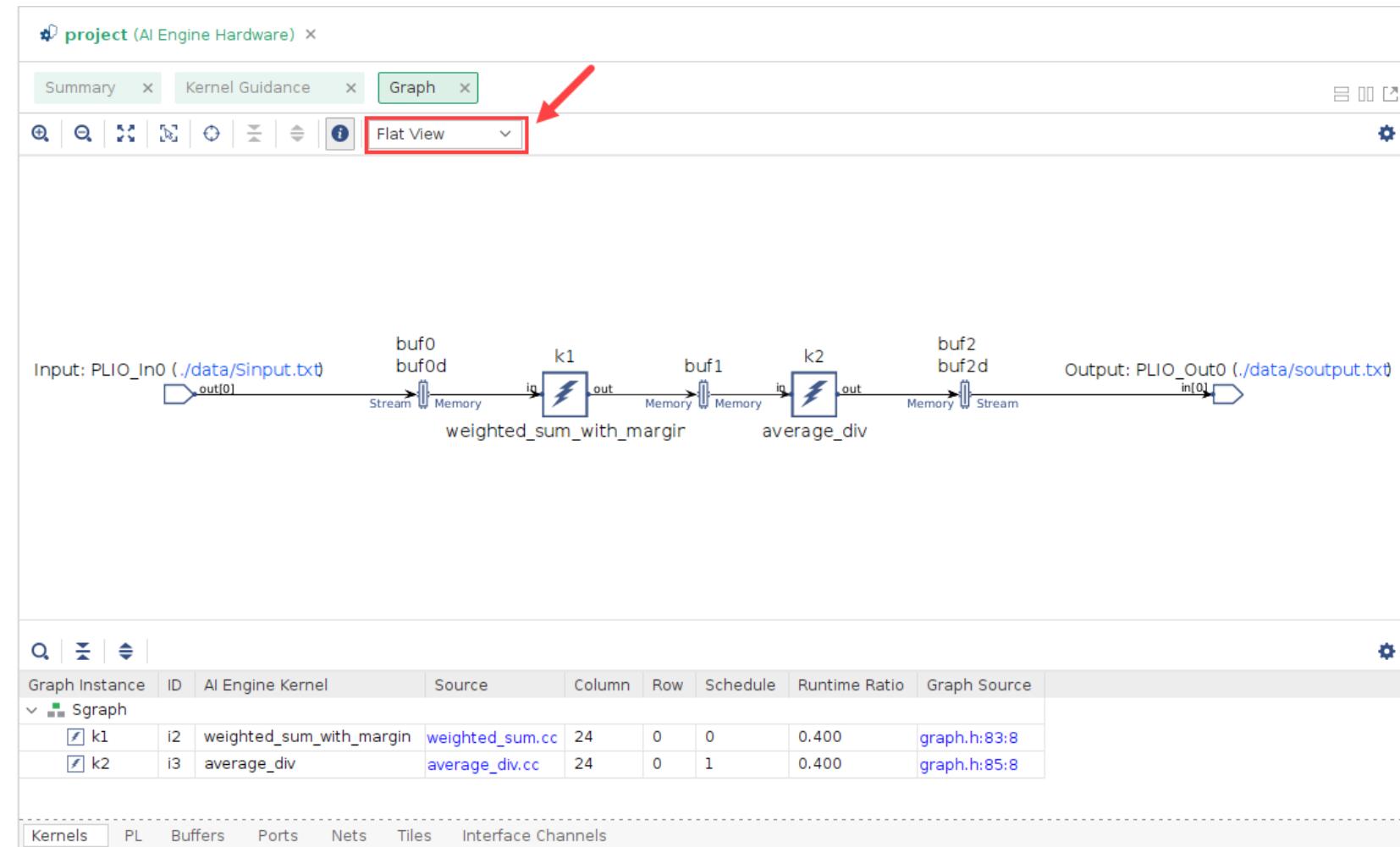
Compile Report: Graph View

- Graph provides a flow diagram of the AI Engine graph
- Shows all the kernels defined in the AI Engine graph
- Layouts used to view the graph:
 - Subgraph View
 - **Tile View**
 - Flat View



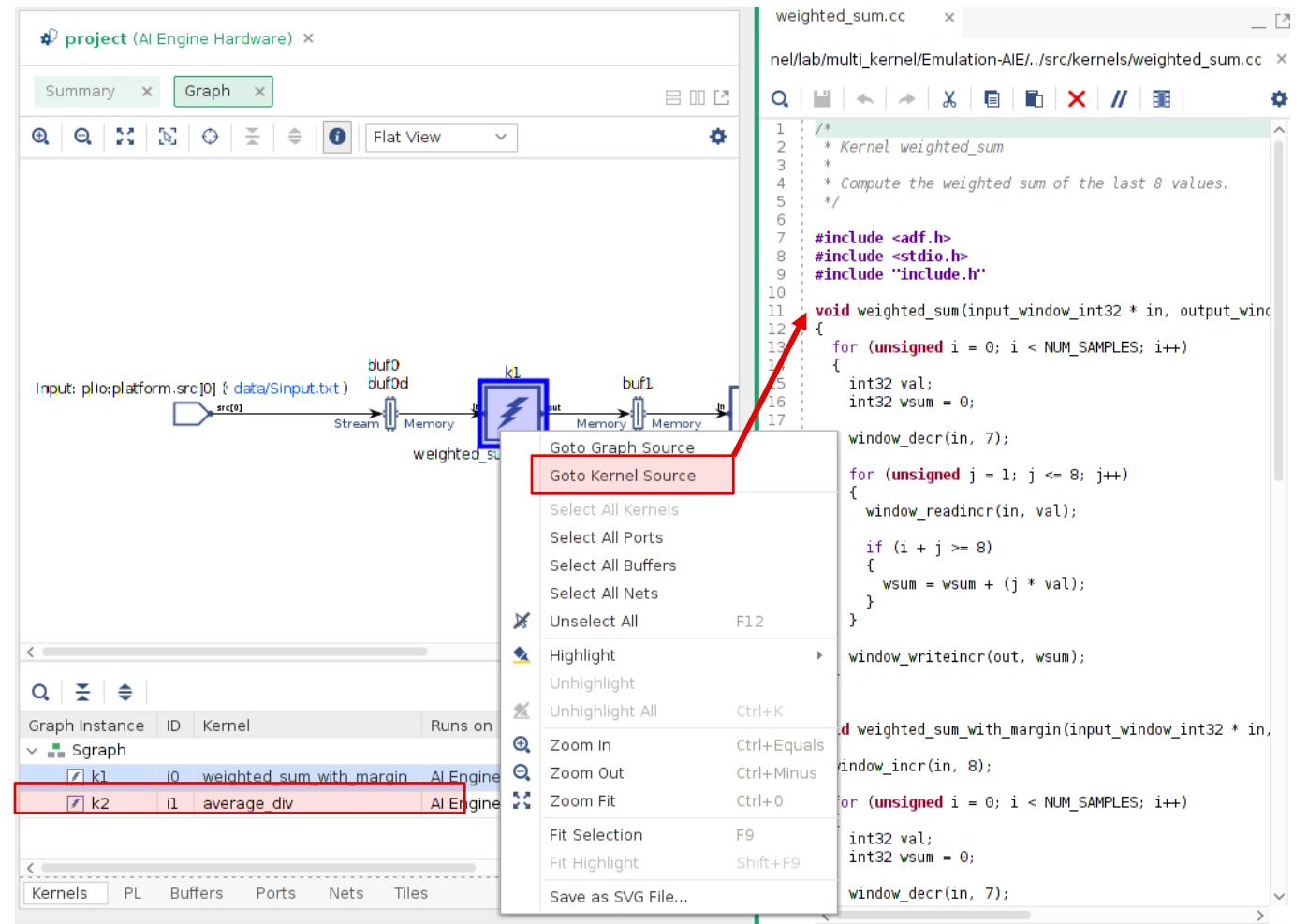
Compile Report: Graph View

- Graph provides a flow diagram of the AI Engine graph
- Shows all the kernels defined in the AI Engine graph
- Layouts used to view the graph:
 - Subgraph View
 - Tile View
 - **Flat View**



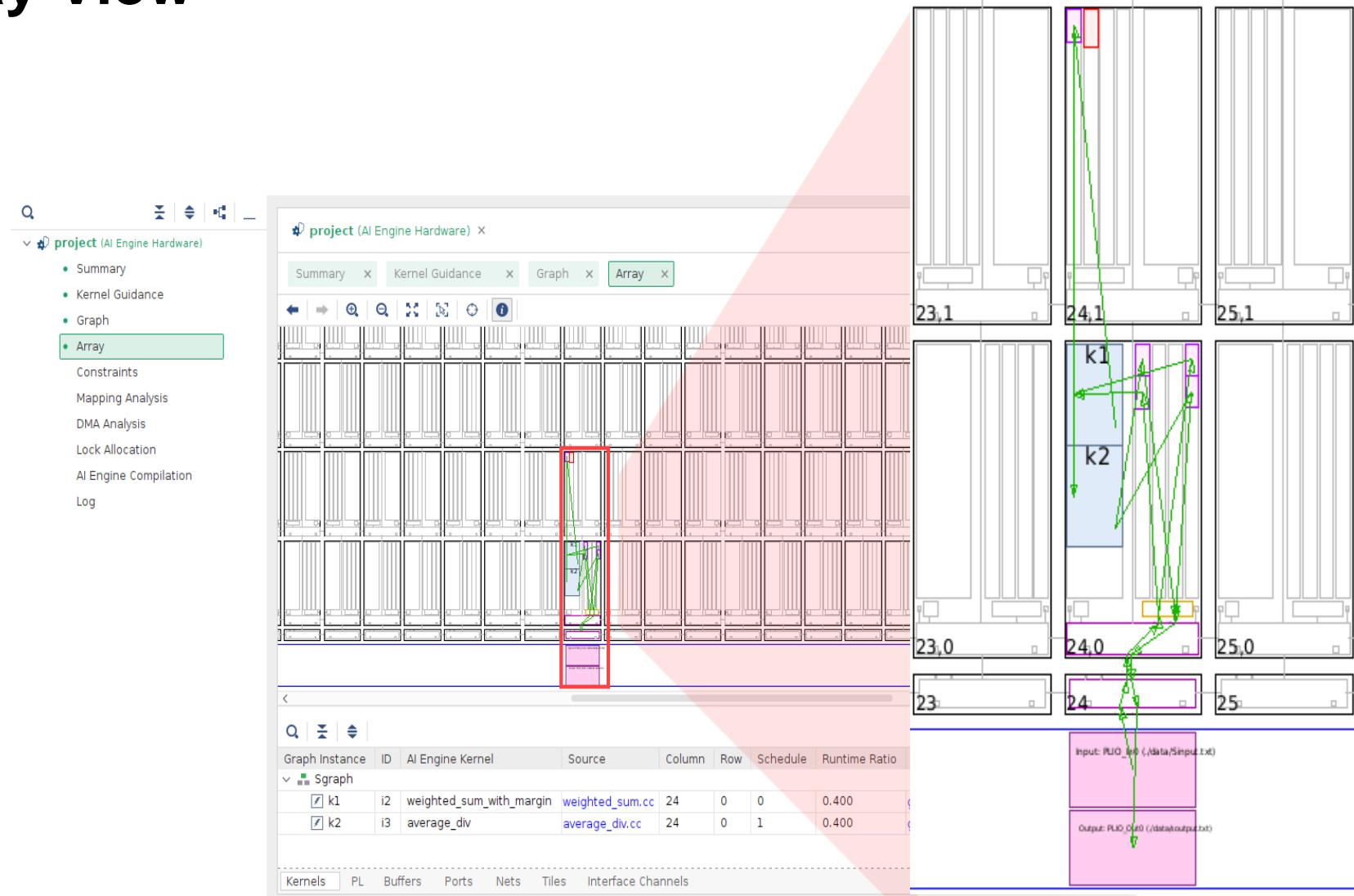
Compile Report: Graph View

- Cross probing
 - Access to kernel source
 - Access to graph source



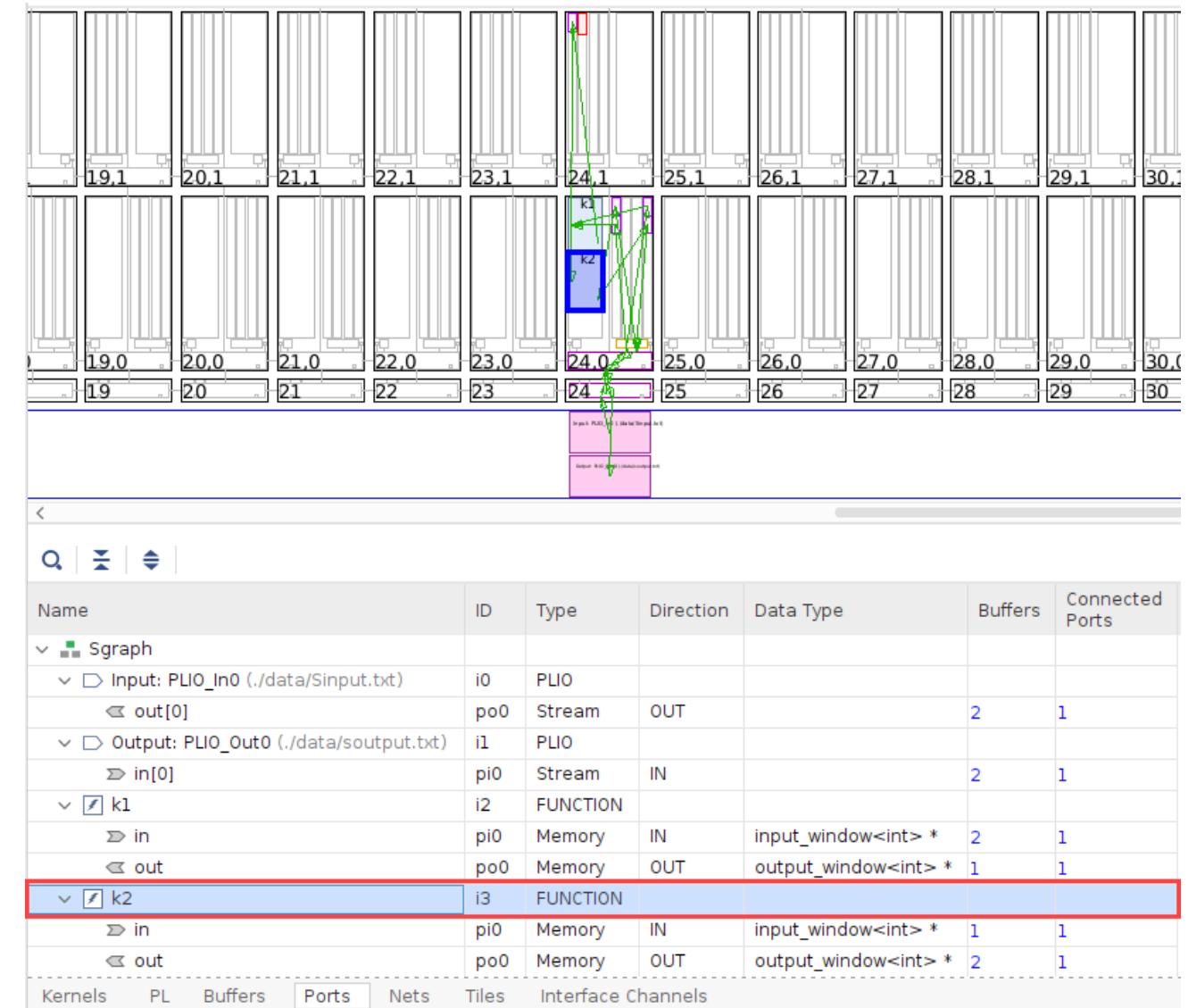
Compile Report: Array View

- Shows the complete AI Engine array
- Graph kernels and connections placed in context of array



Compile Report: Array View

- Choosing the objects in the array shows the details in the table below



Compile Report: Constraints

- Provides details for port constraints

The screenshot shows a software interface for managing port constraints. On the left, there is a tree view with the following structure:

- Port Constraints (1)
 - Sgraph.k1/in

The item "Sgraph.k1/in" is highlighted with a blue selection bar. To the right of the tree view is a detailed panel for the selected port constraint:

Port Constraint
Display 'in' constraint details

Port: ➤ Sgraph.k1/in

Below this, another panel shows colocated ports:

- Colocated Ports (1)
 - ↳ Sgraph.k2/out

Compile Report: Mapping Analysis

- Displays the text report: `project_mapping_analysis_report.txt`

project (AI Engine Hardware)

Summary Kernel Guidance Graph Array Constraints Mapping Analysis

/home/xilinx/training/multi_kernel/lab/multi_kernel/Emulation-AIE/Work/reports/project_mapping_analysis_report.txt

Mapping Analysis Report

Acronym List

- * CR(x,y) - Core <column, row>
- * MG(x,y):b - MemoryGroup <column, row> : Bank
- * MT(x,y):b - MemoryFile <column, row> : Bank

Block Mapping Report:

Block:Function Name	CR(x,y)/IO(x)	Schedule	Utilization	Variable Name	Graph Name
io:PLIO	IO(24)			pl_in	Sgraph
il:PLIO	IO(24)			pl_out	Sgraph
i2:weighted_sum_with_margin	CR(24,0)	0	0.400	k1	Sgraph
i3:average_div	CR(24,0)	1	0.400	k2	Sgraph

Port Mapping Report:

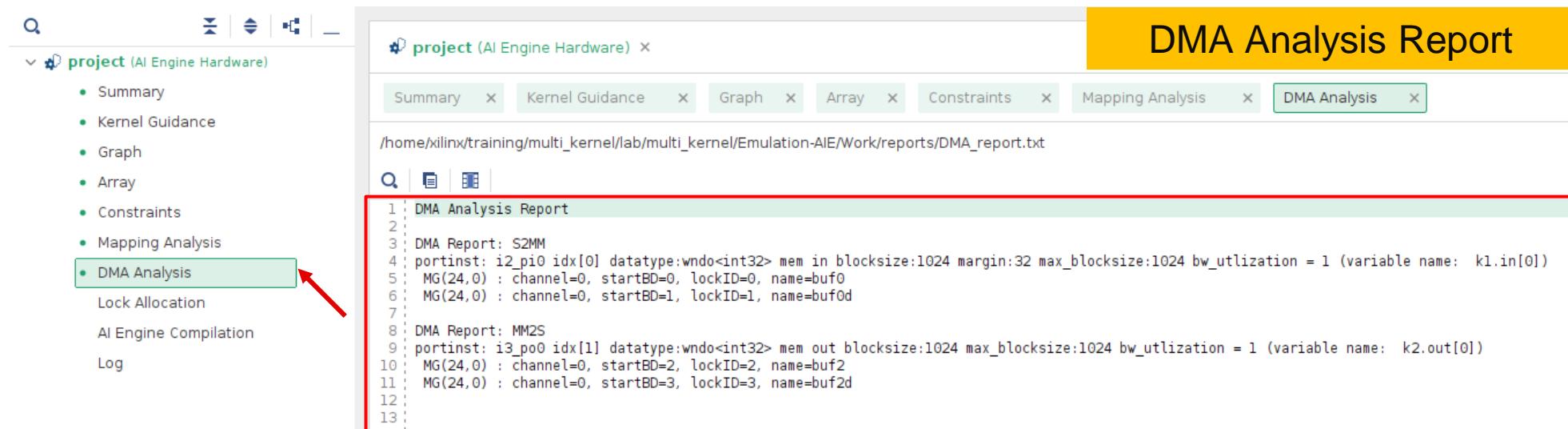
PortName	Dir	PrtType	Buftype	Buffer Name	MG(x,y):b	Addr	Size	Variable Name	Graph Name
i2_pio	in	window	mem	buf0	MG(24,0):0	0x00420	1056	k1.in[0]	Sgraph
i2_pio	in	window	mem	buf0d	MG(24,0):3	0x06000	1056	k1.in[0]	Sgraph
il_pio	in	stream	mem	buf2	MG(24,0):0	0x00000	1024	pl_out.in[0]	Sgraph
il_pio	in	stream	mem	buf2d	MG(24,0):3	0x06420	1024	pl_out.in[0]	Sgraph
io_pio0	out	stream	mem	buf0	MG(24,0):0	0x00420	1056	pl_in.out[0]	Sgraph
io_pio0	out	stream	mem	buf0d	MG(24,0):3	0x06000	1056	pl_in.out[0]	Sgraph
i2_pio	out	window	mem	buf1	MG(24,1):0	0x00000	1024	k1.out[0]	Sgraph
i3_pio0	out	window	mem	buf2	MG(24,0):0	0x00000	1024	k2.out[0]	Sgraph
i3_pio0	out	window	mem	buf2d	MG(24,0):3	0x06420	1024	k2.out[0]	Sgraph
i3_pio0	in	window	mem	buf1	MG(24,1):0	0x00000	1024	k2.in[0]	Sgraph

Memory Bank Report:

MG(x,y):b	Buffer Name	Addr	Size	Accessing Block	CR(x,y)/IO(x)
MG(24,0):0	buf2	0x00000	1024	i1	IO(24)
MG(24,0):0	buf2	0x00000	1024	i3:average_div	CR(24,0)
MG(24,0):0	buf0	0x00420	1056	i2:weighted_sum_with_margin	CR(24,0)
MG(24,0):0	buf0	0x00420	1056	i0	IO(24)
MG(24,0):3	buf0d	0x06000	1056	i2:weighted_sum_with_margin	CR(24,0)
MG(24,0):3	buf0d	0x06000	1056	i0	IO(24)
MG(24,0):3	buf2d	0x06420	1024	i1	IO(24)
MG(24,0):3	buf2d	0x06420	1024	i3:average_div	CR(24,0)
MG(24,1):0	buf1	0x00000	1024	i2:weighted_sum_with_margin	CR(24,0)
MG(24,1):0	buf1	0x00000	1024	i3:average_div	CR(24,0)
MG(24,1):1	sysmem3	0x02000	1224	main	CR(24,0)

Compile Report: DMA Analysis

- Displays the text report: **DMA_report.txt**
- Summary of DMA accesses from the graph
- DMA generated by the compilers which include the details of:
 - Tile
 - Channel
 - Data type



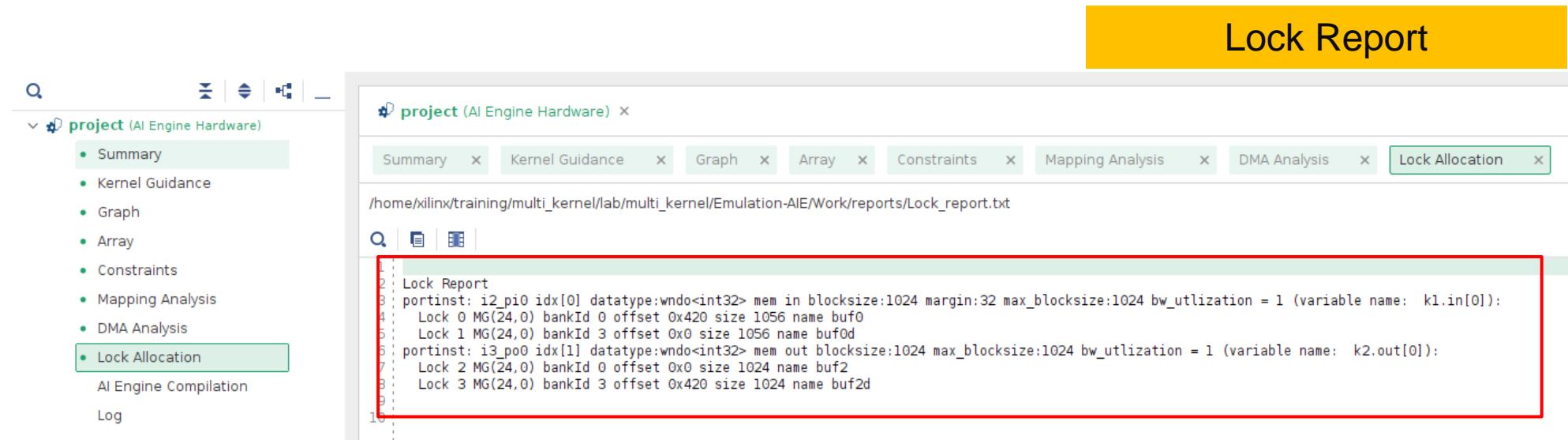
The screenshot shows the Xilinx IDE interface with the following details:

- Project Tree:** Under the project node, the "DMA Analysis" item is highlighted with a green box and has a red arrow pointing to it.
- Toolbars and Menus:** Standard Xilinx toolbars and menus are visible at the top.
- Tab Bar:** The "DMA Analysis" tab is selected in the tab bar.
- Content Area:** The main content area displays the text of `DMA_report.txt`. The text content is as follows:

```
1 DMA Analysis Report
2
3 DMA Report: S2MM
4 portinst: i2_pio idx[0] datatype:wndo<int32> mem in blocksize:1024 margin:32 max_blocksize:1024 bw_utilization = 1 (variable name: k1.in[0])
5 MG(24,0) : channel=0, startBD=0, lockID=0, name=buf0
6 MG(24,0) : channel=0, startBD=1, lockID=1, name=buf0d
7
8 DMA Report: MM2S
9 portinst: i3_poo idx[1] datatype:wndo<int32> mem out blocksize:1024 max_blocksize:1024 bw_utilization = 1 (variable name: k2.out[0])
10 MG(24,0) : channel=0, startBD=2, lockID=2, name=buf2
11 MG(24,0) : channel=0, startBD=3, lockID=3, name=buf2d
12
13
```

Compile Report: Lock Allocation

- Displays the text report: **lock_report.txt**
- Lists DMA locks on port instances



The screenshot shows the Vivado IDE interface with the 'Lock Report' tab selected. The left sidebar shows project navigation with 'Lock Allocation' highlighted. The main window displays the contents of 'Lock_report.txt' with several lines of text describing DMA locks on port instances.

```
1: Lock Report
2: portinst: i2_pi0 idx[0] datatype:wndo<int32> mem in blocksize:1024 margin:32 max_blocksize:1024 bw_utilization = 1 (variable name: k1.in[0]):
3:   Lock 0 MG(24,0) bankId 0 offset 0x420 size 1056 name buf0
4:   Lock 1 MG(24,0) bankId 3 offset 0x0 size 1056 name buf0d
5: portinst: i3_po0 idx[1] datatype:wndo<int32> mem out blocksize:1024 max_blocksize:1024 bw_utilization = 1 (variable name: k2.out[0]):
6:   Lock 2 MG(24,0) bankId 0 offset 0x0 size 1024 name buf2
7:   Lock 3 MG(24,0) bankId 3 offset 0x420 size 1024 name buf2d
```

Vitis Analyzer – Run Summary

- **run_summary** report is generated when the application has been properly configured
- **aiesimulator --profile**
 - For simulator to collect profiling data

The screenshot shows the Vitis Analyzer interface with two tabs open: 'test (AI Engine Hardware)' and 'default (AI Engine Simulation)'. The 'default' tab is active, displaying a summary of a completed run. The summary includes:

- STATUS:** Completed
- VERSION:** [redacted]
- STARTED:** June 01, [redacted]
- COMPLETED:** June 01, [redacted]
- ELAPSED:** 04m 22s
- PLATFORM:** VC1902
- AI ENGINE FREQUENCY:** 1250 MHz
- COMMAND LINE:**

```
aiesimulator
--pkg-dir ./Work
--aiearch-version aie
--i=..
--dump-vcd=foo
--profile
```

Vitis Analyzer – Run Summary

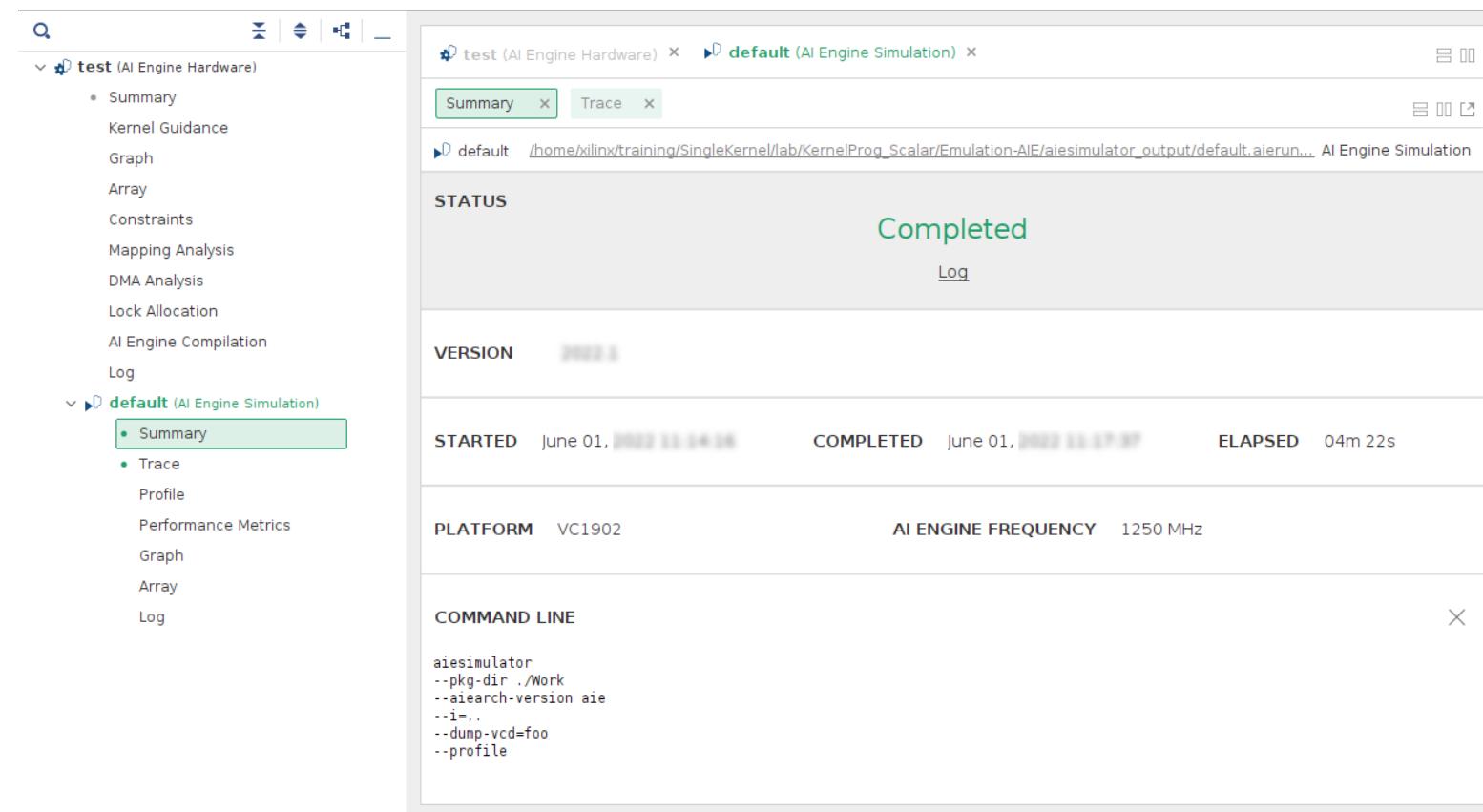
During simulation of the AIE graph:

- AI Engine simulator captures performance and activity metrics
- Writes the report to the output directory **./aiesimulator_output**

aiesimulator run_summary is named default.aierun_summary

Reports include:

- Summary
- Trace
- Profile
- Performance Metrics
- Graph
- Array
- Log



Vitis Analyzer – Trace View

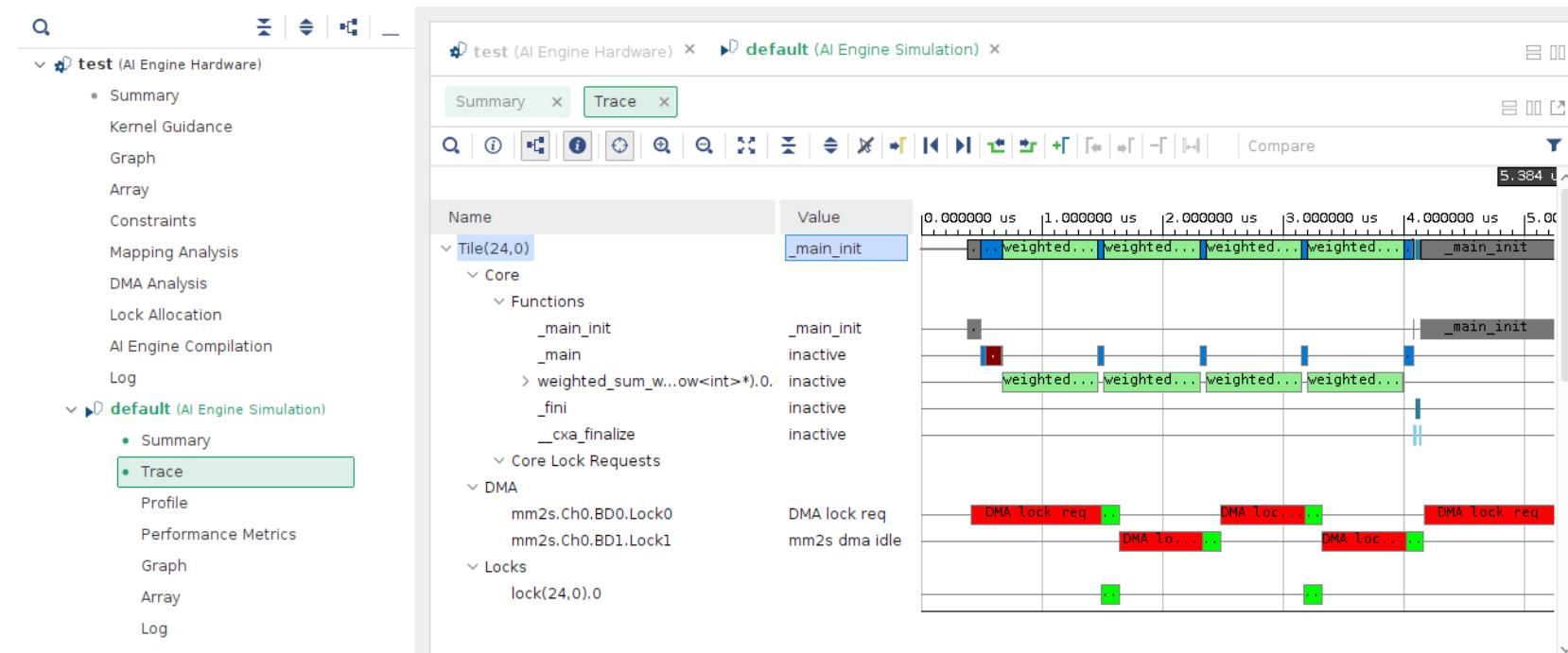
- Trace view gives information on what runs on each tile

Helps debugging:

- Missing or mismatching locks
- Buffer overruns
- Incorrect programming of DMA buffers

Event trace provides:

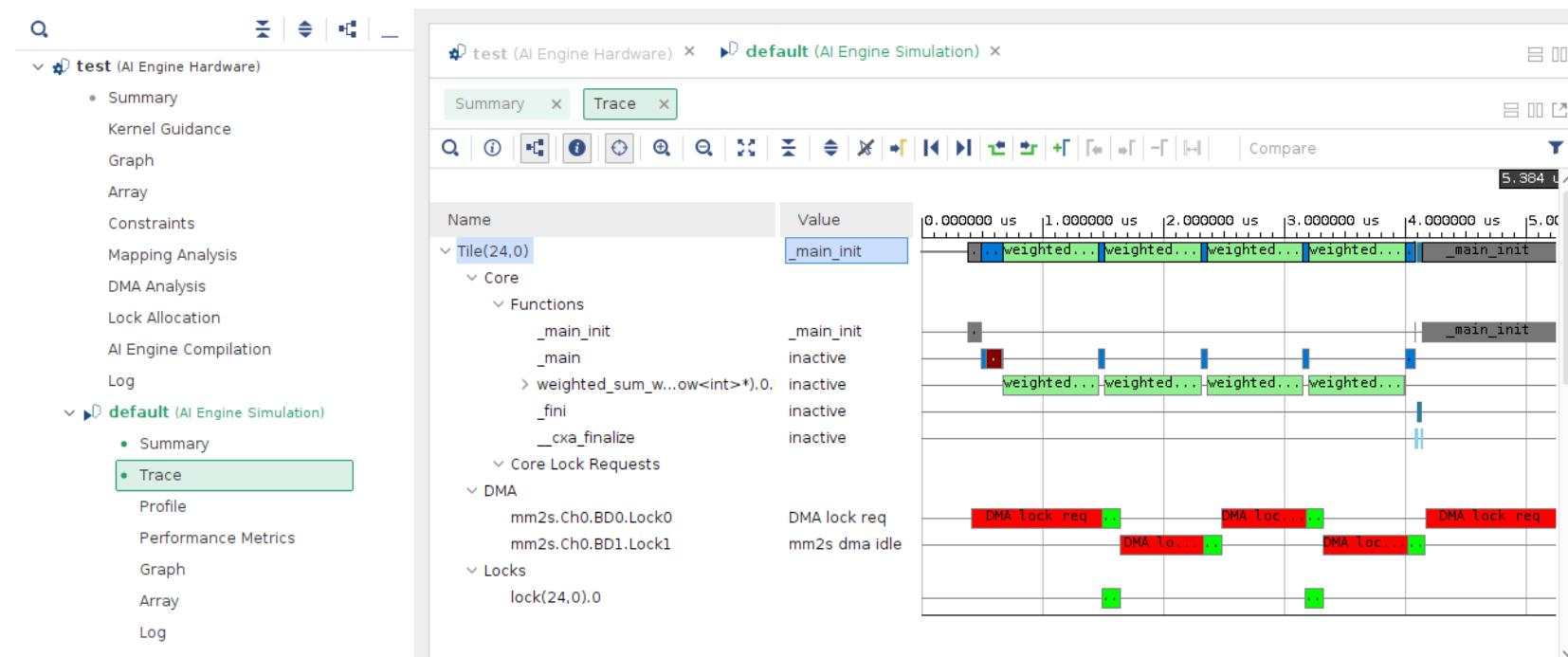
- System-level traces for program events
- Provides direct support for generation, collection, and streaming of hardware events as a trace



Vitis Analyzer – Trace View

Features of the Trace Report

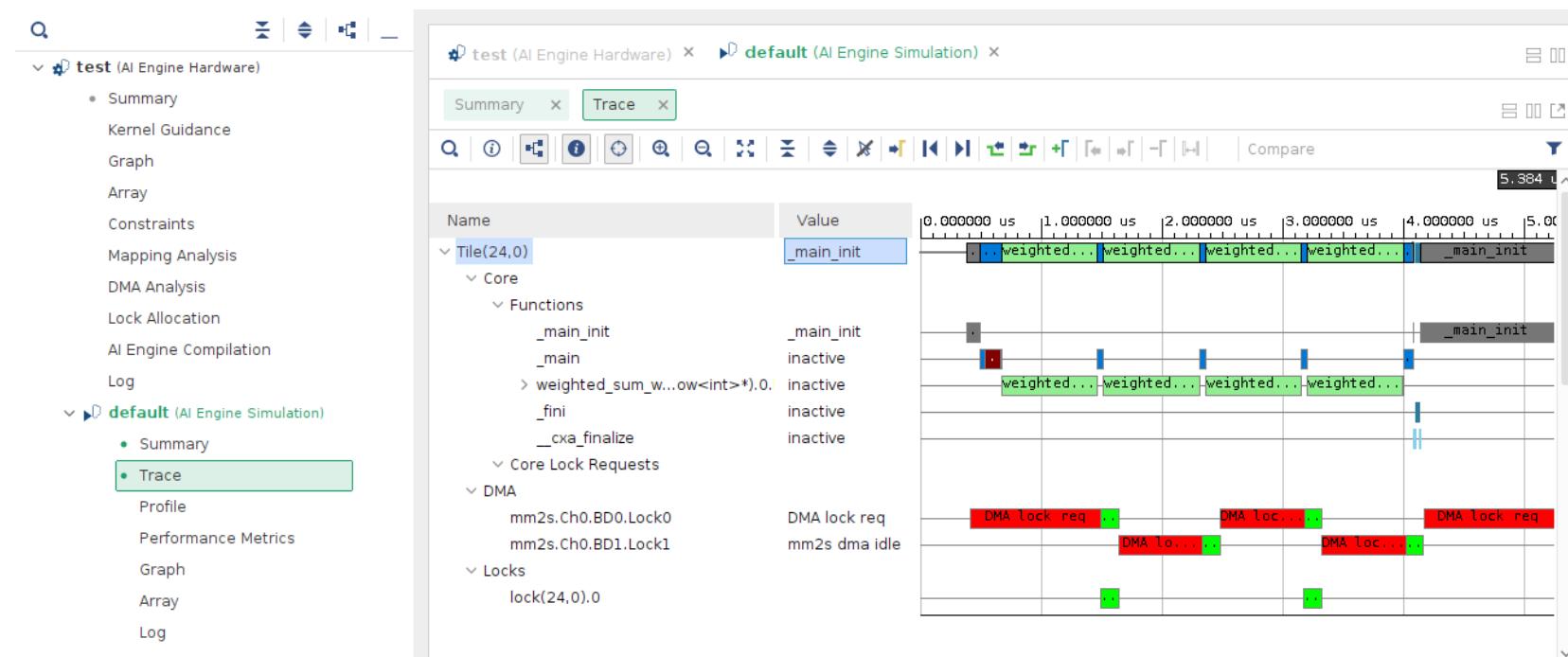
- Each tile is reported
 - DMA
 - Locks
 - I/O
- Separate timeline for each kernel mapped to a core
- Shows kernel status
 - Blue: executing
 - Red: stalled
- Lock IDs help to identify how cores and DMAs interact with one another



Vitis Analyzer – Trace View

Features of the Trace Report

- Lock section shows the activities of the locks in the tile
- If a lock is not released, a red bar extends through the end of simulation time
- Data view shows the data flowing through stream switch network with slave entry points and master exit points at each hop
- Useful in finding:
 - Routing delays
 - Network congestion



Vitis Analyzer – Profile

- Provides overall performance of the application
- Grouped into categories
- Useful for system-level performance tuning and debug

The screenshot shows the Vitis Analyzer interface with two projects open:

- test (AI Engine Hardware)**: This project is collapsed. Its contents include:
 - Summary
 - Kernel Guidance
 - Graph
 - Array
 - Constraints
 - Mapping Analysis
 - DMA Analysis
 - Lock Allocation
 - AI Engine Compilation
 - Log
- default (AI Engine Simulation)**: This project is expanded, showing its sub-components:
 - Summary
 - Trace
 - Profile** (highlighted in green)
 - Performance Metrics
 - Graph
 - Array
 - Log

The **Profile** tab is selected for the **default** project. The main pane displays the following summary information:

```

Processor: ::tl.aie_logical.aie_xtlm.math_engine.array.tile_24_1.cm.proc.iss
Total cycle count: 6,246
Report cycle count: 4,695
Total instruction count: 4,418
Total size in program memory: 1,782
Command: ::tl.aie_logical.aie_xtlm.math_engine.array.tile_24_1.cm.proc.iss profile
  
```

Below this, the **Profile Details** pane provides more detailed profiling information:

```

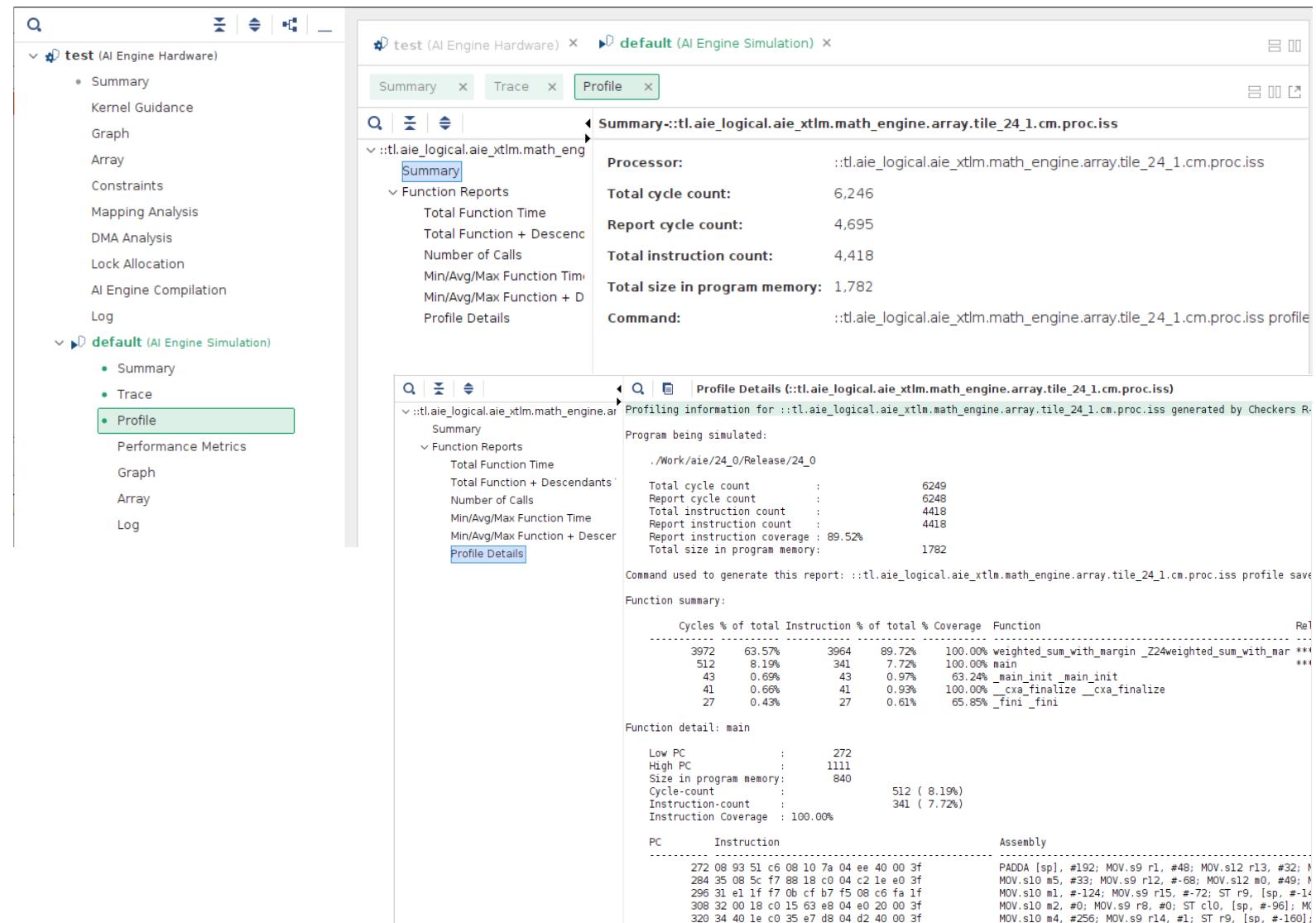
Profiling information for ::tl.aie_logical.aie_xtlm.math_engine.array.tile_24_1.cm.proc.iss generated by Checkers R...
Program being simulated:
./Work/aie/24_0/Release/24_0
  
```

It lists various metrics such as total cycle count, report cycle count, total instruction count, report instruction count, and report instruction coverage.

At the bottom, there is a table for function summaries and a detailed view of the `main` function's assembly code.

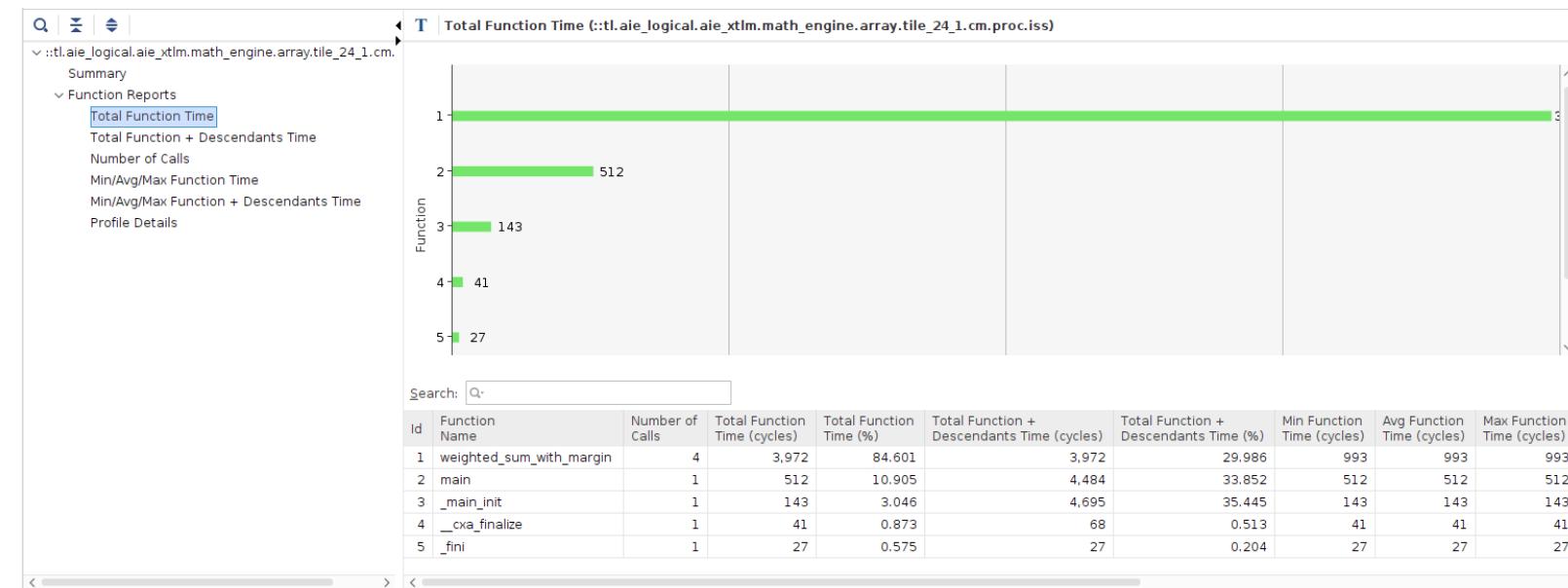
Vitis Analyzer – Profile

- System performance is presented in terms of:
 - Latency
 - Throughput
- Sub-optimal system performance forces you to examine and control:
 - Mapping and buffer packing
 - Stream and packet switch allocation
 - Interaction with neighboring processors
 - External interfaces



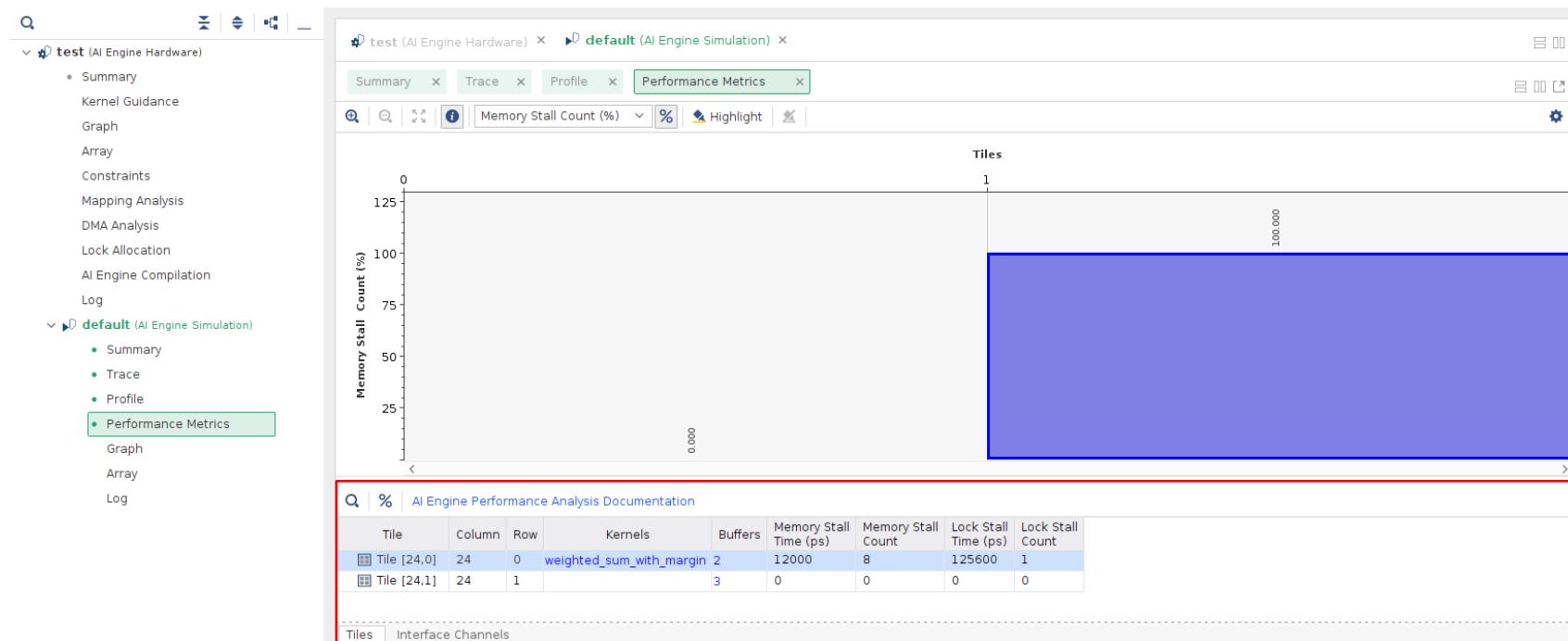
Vitis Analyzer – Profile

- Shows Total Function Time:
Total cycles that the function used in running the graph
- Useful in determining where time is being spent in a function and helps with potential optimization or debug



Vitis Analyzer – Performance Metrics

- Shows memory stall time in ns
- Shows memory stall time in percent
- Shows memory stall count



Summary

- Vitis development environment generates various profiling reports
- These reports can be viewed and analyzed using the Vitis analyzer
- AI Engine tool generates the following reports:
 - Compile Summary
 - Kernel Guidance
 - Graph View
 - Array View
 - Constraints
 - Mapping Analysis
 - DMA Analysis
 - Lock Allocation, AI Engine Compilation, and Log
 - Run Summary
 - Summary, Graph, Array, and Log



The Programming Model: Multiple Kernels Using Graphs

Objectives

After completing this module, you will be able to:

- Describe the Adaptive Data Flow (ADF) graph
- Describe the declaration of a graph, including the port, kernel, and connection details
- List the various attributes for the ports, kernels, and connections
- Describe graph control APIs such as `init`, `run`, `wait`, and `end`

The Programming Model – Kernel and Graph

- Kernel function
- Computation function
- Data flow graph specification

```
#include <adf.h>
void weighted_sum(input_window<int32> *in,
                   output_window<int32> *out){
    for(unsigned int i=0; i< WLEN; i++){
        int32 val;
        int32 wsum = 0;
        for(unsigned int j=0; j< 8; j++){
            window_readincr(in, val);
            wsum = wsum + (j * val);
        }
        window_writeincr(out, wsum);
        window_decr(in, 7);
    }
}
```

The Programming Model – Kernel and Graph

- Graph
- Header file
- Connects kernels and sub-graphs

```
#include "kernels.h"
#include <adf.h>

class FirstGraph : public graph{
private:
    kernel k;
public:
    input_plio pl_in;
    output_plio pl_out;

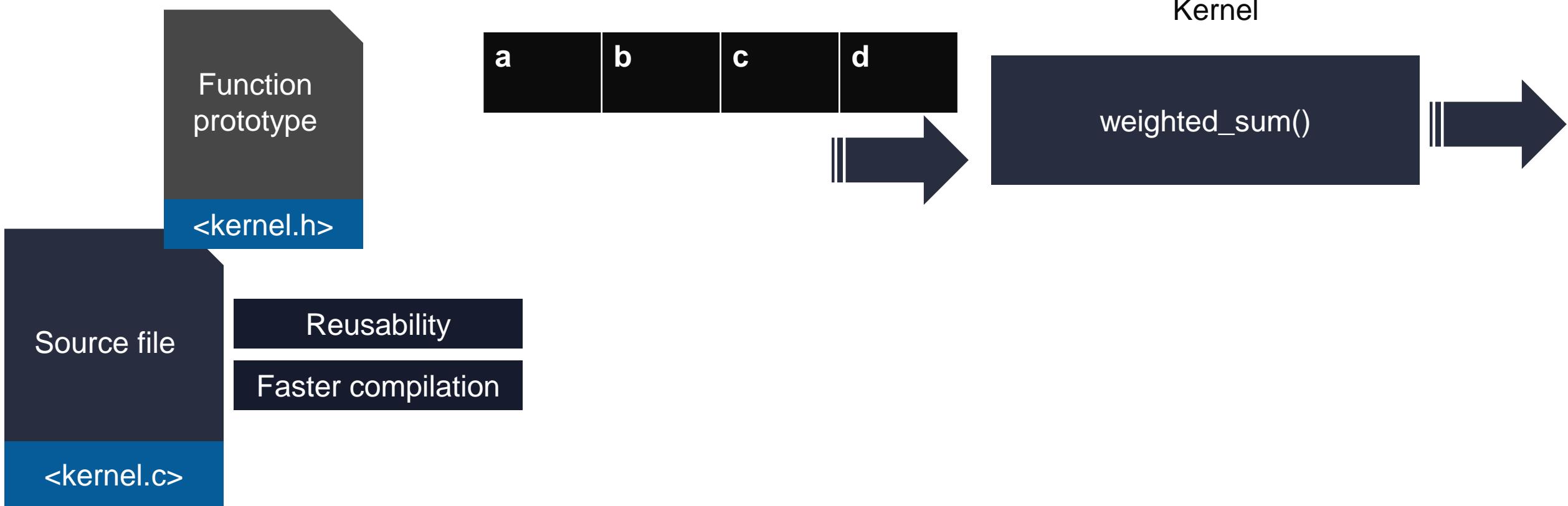
    FirstGraph() {
        pl_in = input_plio::create("PLIO_In", plio_32_bits, INPUT_FILE, 250.0);
        pl_out = output_plio::create("PLIO_Out", plio_32_bits, OUTPUT_FILE, 250.0);

        k = kernel::create(weighted_sum);

        connect< window<WLEN*NBYTES,WMARGIN*NBYTES> > net0 (pl_in.out[0], k.in[0]);
        connect< window<WLEN*NBYTES> > net1 (k.out[0], pl_out.in[0]);

        source(k) = "aie_kernels/weighted_sum.cc";
        runtime<ratio>(k) = 0.1;
    }
};
```

The Programming Model – Kernel



C/C++ functions

The Programming Model – Graph

- Define application graph class in header file

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;
};
```

The Programming Model – Graph

- Define application graph class in header file
- Add Adaptive Data Flow (ADF) library and include the kernel function prototypes

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;
};
```

The Programming Model – Graph

- Define application graph class in header file
- Add Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Define graph class, use objects defined in the adf name space

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;
};
```

The Programming Model – Graph

- Define application graph class in header file
- Add Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Define graph class, use objects defined in the adf name space
- Declare the top-level ports to the graph class

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;
};
```

The Programming Model – Graph

- Define application graph class in header file
- Add Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Define graph class, use objects defined in the adf name space
- Declare the top-level ports to the graph class
- Declare the kernels

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;
};
```

Kernel Instantiation in graph.h

- Kernels
 - Ordinary C/C++ functions
 - Return void
 - Use special data types
- To instantiate the first and second C++ kernel objects
- Example: Two-node application implemented on AI Engines

```
#include <adf.h>
#include "kernels.h"
using namespace adf;

class simpleGraph : public graph {
public:

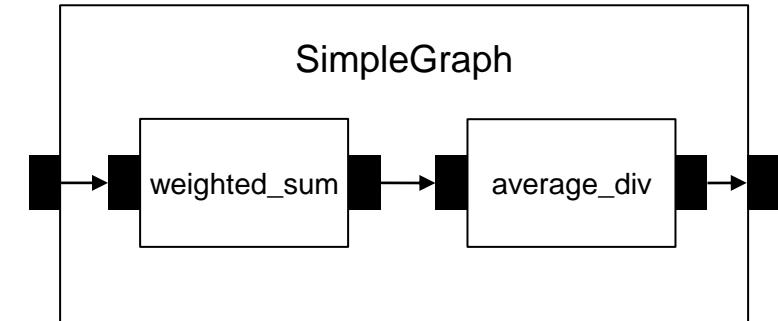
    // Declare external ports
    input_plio in;
    output_plio out;

    // Declare kernels
    adf::kernel k1;
    adf::kernel k2;

    simpleGraph() {
        // Bind a function to each of the kernels
        k1 = adf::kernel::create(weighted_sum);
        k2 = adf::kernel::create(average_div);
    };
};
```

Connecting Kernels in graph.h

- Connectivity information added in a data flow graph
 - Create connections between ports
- Ports are referred to by indices



```
simpleGraph () {
    // Bind a function to each of the declared kernels
    k1 = adf::kernel::create(weighted_sum);
    k2 = adf::kernel::create(average_div);

    // create nets to connect kernels and IO ports
    adf::connect<adf::window<128>> net0(pl_in.out[0], k1.in[0]);
    adf::connect<adf::window<128>> net1(k1.out[0], k2.in[0]);
    adf::connect<adf::window<128>> net2(k2.out[0], pl_out.in[0]);
}
```

Kernel Connection Semantics

- Bytes of data
 - Number of samples depend on datatype
 - Window size should be a multiple of 16 bytes
 - Matches the 128-bit width of the memory banks
- Margin
 - Optional second template parameter
 - Identifies the overlap (in bytes) from one block of data to the next

```
connect< window<bytes, margin> > net0 (pl_in.out[0], k1.in[0]);
```

Kernel Connection Semantics - Example

```
connect< window<128, 32> > net0 (pl_in.out[0], k1.in[0]);
```

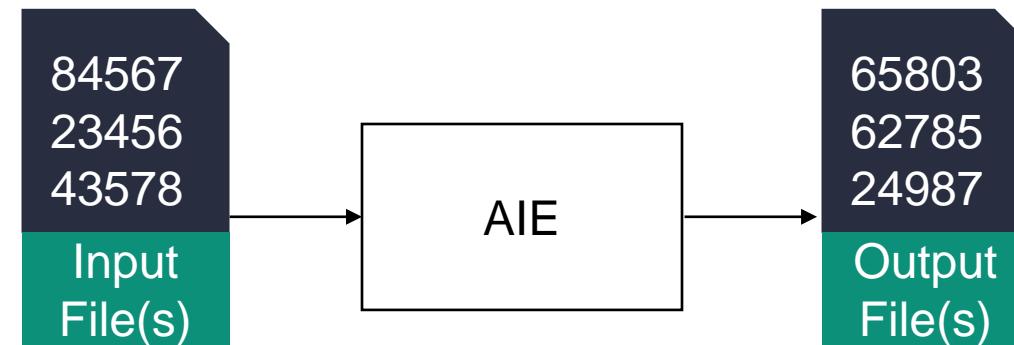
- Consume: 128-Byte
- Last sample 32-Byte
- Total buffer size: 160-Byte

Specifying Kernel Sources in graph.h

- Attributes can be attached to any logical object in the graph:
- Kernels
 - Source used to implement the functionality of the kernel
- Ports
 - An enumeration of attributes of each platform, starting with all the inputs and followed by the outputs
- Connections
 - I/O attribute specification
 - FileIO
 - GMIO
 - PLIO

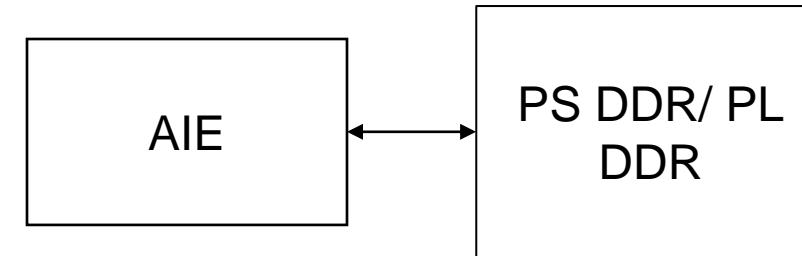
Specifying Kernel Sources in graph.h - FileIO

- Represents the I/O port attribute specification
- Used to connect an external file to a graph input or output for simulation purposes



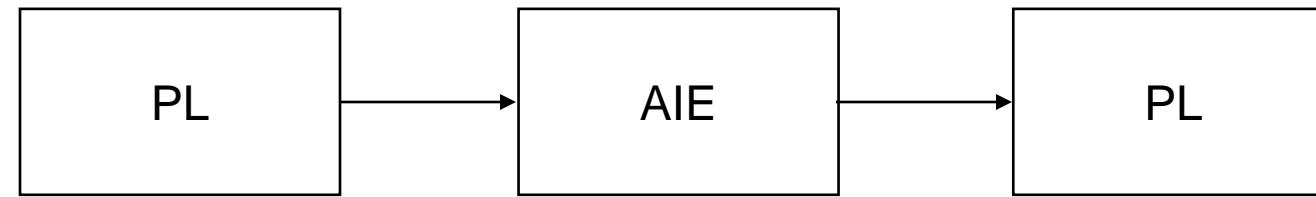
Specifying Kernel Sources in graph.h - GMIO

- Represents the I/O port attribute specification
- Used to connect graph kernels to the external virtual platform ports
- Connect AI Engine kernels with the DDR or connect PL blocks with the DDR



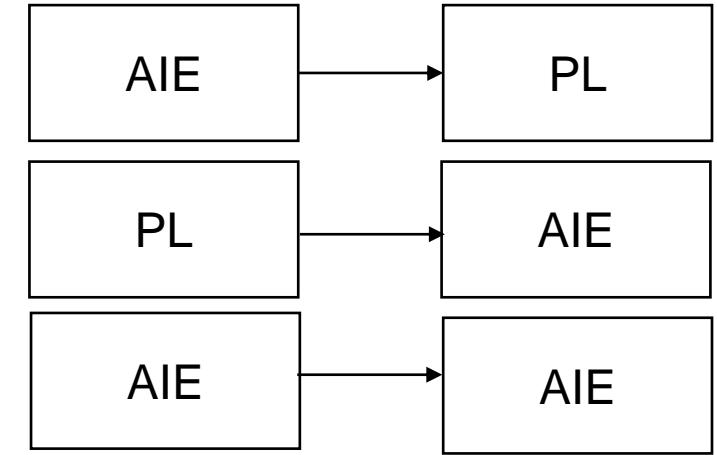
Specifying Kernel Sources in graph.h - PLIO

- Represents the I/O port attribute specification
- Used to make external stream connections that cross the AI Engine to programmable logic (PL)



Specifying Kernel Sources in graph.h

- All sources and destinations are supported as connections
 - PL kernel to an AIE kernel
 - AIE kernel to AIE kernel
 - AIE kernel to PL kernel
 - Stream or Windows
 - Similar syntax



```
// specify kernel sources
source(adder) = "module/add.cpp";
// specify the connections
connect<window<32>, stream>(prod.out[0], adder.in[0]);
connect<window<32>, stream>(prod.out[1], adder.in[1]);
connect<stream, window<32> >(adder.out[0], cons.in[0]);
```

runtime<ratio>

- Specific AI Engine usage fraction for a kernel
- Ratio of the number of cycles taken by one invocation of a kernel to the cycle budget
- Cycle budget for an application is typically fixed
- Ratio of the kernel runtime compared to cycle budget (between 0 and 1)
- Cycle budget and function runtime can be affected when changing frame size
- Compiler decides whether to combine multiple kernels onto a single AI Engine
 - Based on runtime ratio

```
simpleGraph () {  
    runtime<ratio>(k1) = 0.4;  
    runtime<ratio>(k2) = 0.4;  
}
```

location<kernel>

- The compiler maps kernels to AIE Tiles automatically
- Users can map manually kernel to AIE tiles with *location<kernel>*
- Relative or absolute placement
- For large graphs reduces the choices for the mapper to try

```
simpleGraph () {  
    location<kernel>(k1) = tile(COL, ROW);  
    location<kernel>(k2) = tile(COL+1, ROW);  
}
```

Graph Control API

- In embedded platforms Processing System (PS) Used to:
 - Load the Graph
 - Monitor the Graph
 - Control the Graph
- For the VCK5000 these do not apply

Graph Control API

- Graph base class provides a number of API methods
- Control initialization and execution of the graph

```
class simpleGraph : public adf::graph {  
public:  
    // Declare external ports  
    input_plio in;  
    output_plio out;  
    // Declare kernels  
    adf::kernel k1;  
    adf::kernel k2;  
};
```

```
simpleGraph mygraph;  
int main(void) {  
    mygraph.init();  
    mygraph.run(1);  
    mygraph.end();  
    return 0;  
}
```

Graph Control API

- init() method loads the graph to the AI Engine array at pre-specified AI Engine tiles
- Loads ELF binaries for each AI Engine
- Configures the stream switches for routing
- Configures the DMAs for I/O
- Leaves the processors in a disabled state

```
simpleGraph mygraph;
int main(void) {
    mygraph.init();
    mygraph.run(1);
    mygraph.end();
    return 0;
}
```

Graph Control API

- run() method starts the graph execution by enabling the processors

```
simpleGraph mygraph;
int main(void) {
    mygraph.init();
    mygraph.run(1);
    mygraph.end();
    return 0;
}
```

Graph Control API

- end() method waits until all the processors reach the end of their respective (local) main execution
- Disables the graph execution

```
simpleGraph mygraph;
int main(void) {
    mygraph.init();
    mygraph.run(1);
mygraph.end();
    return 0;
}
```

Graph Control API: Iterative Control

- API wait()
- Used to wait for the first run to finish before starting the second run
- Has the same blocking effect as end
- Allows re-running the graph again without having to re-initialize it

```
int main(void) {
    mygraph.init();
    mygraph.run(3); // start 3 iterations
    mygraph.wait(); // wait for iterations to finish
    mygraph.run(10); // start 10 iterations
    mygraph.end(); // wait for iterations to finish
                    // and terminate graph
    return 0;
}
```

Graph Control API: Iterative Control

- Calling run back-to-back without an intervening wait to finish that run can have an unpredictable effect
- run() API modifies the loop bounds of the active processors of the graph
- If there is not enough data for the input, the simulation will hang

```
int main(void) {
    mygraph.init();
    mygraph.run(3); // start 3 iterations
    mygraph.wait(); // wait for iterations to finish
    mygraph.run(10); // start 10 iterations
    mygraph.end(); // wait for iterations to finish
                    // and terminate graph
    return 0;
}
```

Graph Control API: Timed Control

- Timed execution model is suitable for testing multi-rate graphs
- There are variants of the wait() and end() APIs with a positive integer specifying a cycle timeout
- This is the number of AI Engine cycles that the API call will block before disabling the processors and returning
- Blocking condition does not depend on any graph termination event
- Graph can be in an arbitrary state at the expiration of the timeout

```
int main(void) {
    mygraph.init();
    mygraph.run();          // start the graph
    mygraph.wait(1000);     // wait for 1000 cycles
    mygraph.resume();       // continue executing
    mygraph.end(1500);      // wait for 1500 cycles more and terminate graph
    return 0;
}
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports
- Step 4: Specify the input & output files to the I/O ports

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum_with_margin);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports
- Step 4: Specify the input & output files to the I/O ports
- **Step 5: Declare the kernels**

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports
- Step 4: Specify the input & output files to the I/O ports
- Step 5: Declare the kernels
- Step 6: Connect the ports for the kernels k1 and k2

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32> > net0 (in.out[0], k1.in[0]);
        connect<window<128> > net1 (k1.out[0], k2.in[0]);
        connect<window<128> > net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports
- Step 4: Specify the input & output files to the I/O ports
- Step 5: Declare the kernels
- Step 6: Connect the ports for the kernels k1 and k2
- Step 7: Attach the source to the kernels

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- Step 1: Add the Adaptive Data Flow (ADF) library and include the kernel function prototypes
- Step 2: Define your graph class
- Step 3: Declare the top-level ports
- Step 4: Specify the input & output files to the I/O ports
- Step 5: Declare the kernels
- Step 6: Connect the ports for the kernels k1 and k2
- Step 7: Attach the source to the kernels
- Step 8: Set the AI Engine utilization as 0.1

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class simpleGraph : public graph {
private:
    kernel k1;
    kernel k2;
public:
    input_plio in;
    output_plio out;
    simpleGraph() {
        in = input_plio::create("In", plio_32_bits, in_file, 250.0);
        out = output_plio::create("Out", plio_32_bits, out_file, 250.0);
        k1 = kernel::create(weighted_sum);
        k2 = kernel::create(average_div);
        connect<window<128, 32>> net0 (in.out[0], k1.in[0]);
        connect<window<128>> net1 (k1.out[0], k2.in[0]);
        connect<window<128>> net2 (k2.out[0], out.in[0]);
        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";
        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
    }
};
```

Complete Example Description

- A graph object mygraph is declared
 - Instance of simpleGraph
- In the main application, this graph object is:
 - Initialized
 - Run
 - Terminated

```
// project.cpp
#include "graph.h"

simpleGraph mygraph;
int main(void) {
    mygraph.init();
    mygraph.run();
    mygraph.end();
    return 0;
}
```

Single Kernel vs. Multiple Kernel Programming

Single Kernel Programming

- Create separate AI Engine kernel programs
- Use
 - Various vector data types
 - AI Engine APIs
 - AI Engine intrinsics
 - Window function APIs, ...
- Analyze and debug kernel code
 - Compile, simulate, profile, ...

Multiple Kernel Programming

- Create multiple kernel AI Engine projects
- ADF graph-based programming
 - Modular, hierarchical graph definition
 - Instantiation of AI Engine memories, streams, ...
 - Analyze and debug
 - Dataflow, function scheduling, ...

Recommended Project Directory Structure and Coding Practices

Do

- All the ADF graphs must be located in a header file
- Multiple ADF graph definitions can be included in the same header file
- Class header file should be included in the main application
- All headers must be self-contained
- Include all the other necessary header files

Do not

- There should be no dependencies in the order that the header files are to be included
- There should be no file-scoped variables or data structure definitions in the graph header files
- No need to declare kernels under extern "C" {...}. If doing so, it must adhere to the following conditions:
 - Kernel function declaration is wrapped with extern "C"
 - Extern "C" must be wrapped with #ifdef __cplusplus

Summary

- The Adaptive Data Flow (ADF) graph can connect to ports on:
 - C/C++, OpenCL kernels, and RTL kernels
 - A target platform using the Vitis compiler
- The `runtime<ratio>` constraint allows you to set a specific AI Engine usage fraction for a kernel
- In the main application, a graph object can be initialized, then run, and finally terminated via APIs
 - `init`
 - `run`
 - `end`



AIE DSP Library Overview

Objectives

After completing this module, you will be able to:

- Describe the Versal® AI Engine DSP library
- Identify various components in the AI Engine DSP library
- Install and use the AI Engine DSP library in your application

What Is the AI Engine DSP Library?



Configurable library of kernels that can be used to develop applications on Versal ACAP AI Engines



Open-source library for DSP applications



Kernels are coded in C++ and intrinsics give access to AI Engine vector processing capabilities



Kernels can be combined to construct graphs for developing complex designs



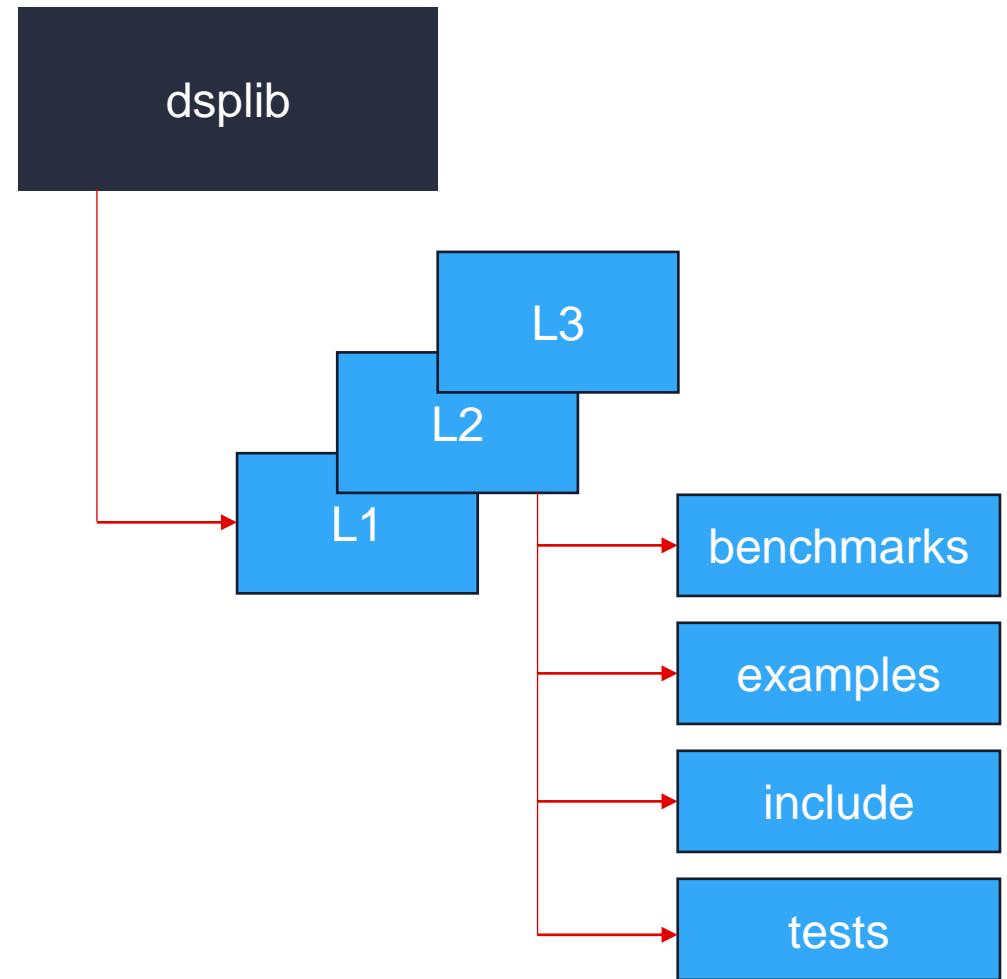
Example design is also provided with this library



Use the library element's graph as the entry point

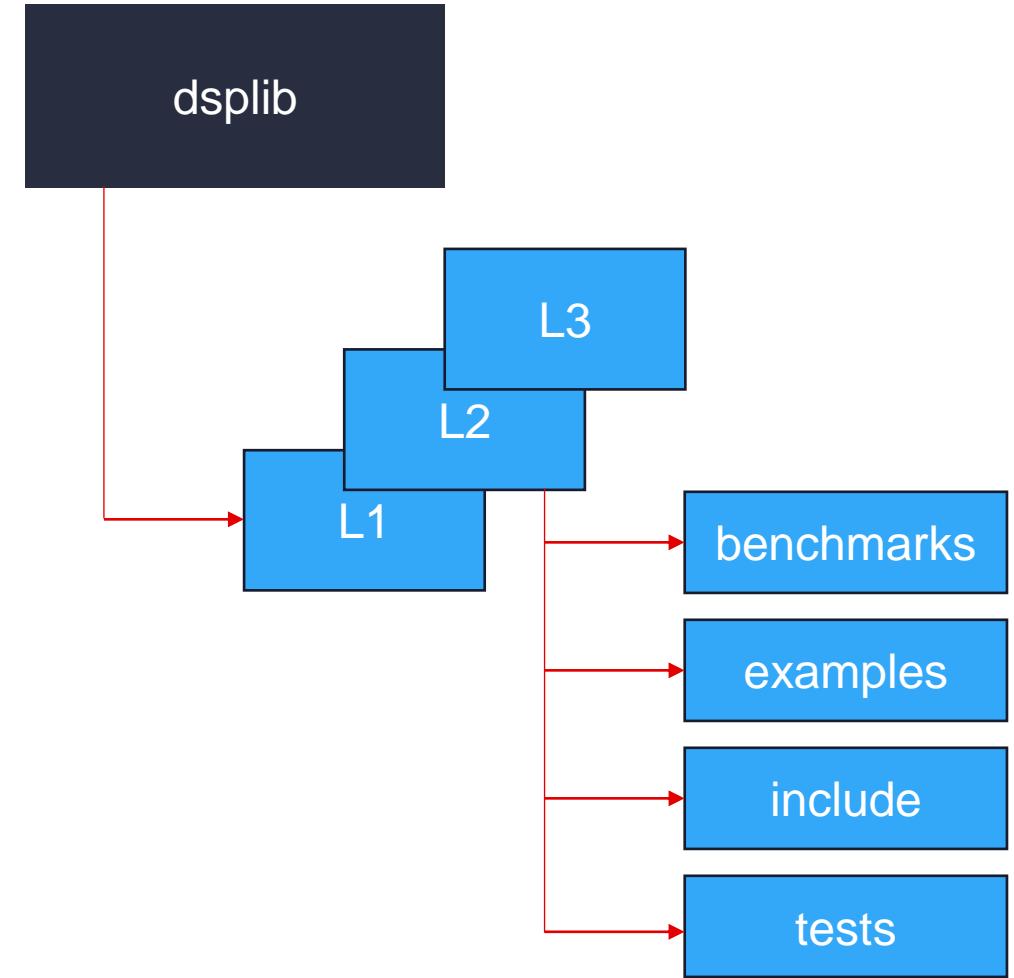
Parameterized Solution Functions

- Directories L1, L2, and L3 correspond to AI Engine kernels, AI Engine graphs, and drivers for each function
- Inclusion of an L2 element rather than an L1 element is recommended
- L3 directory not yet available



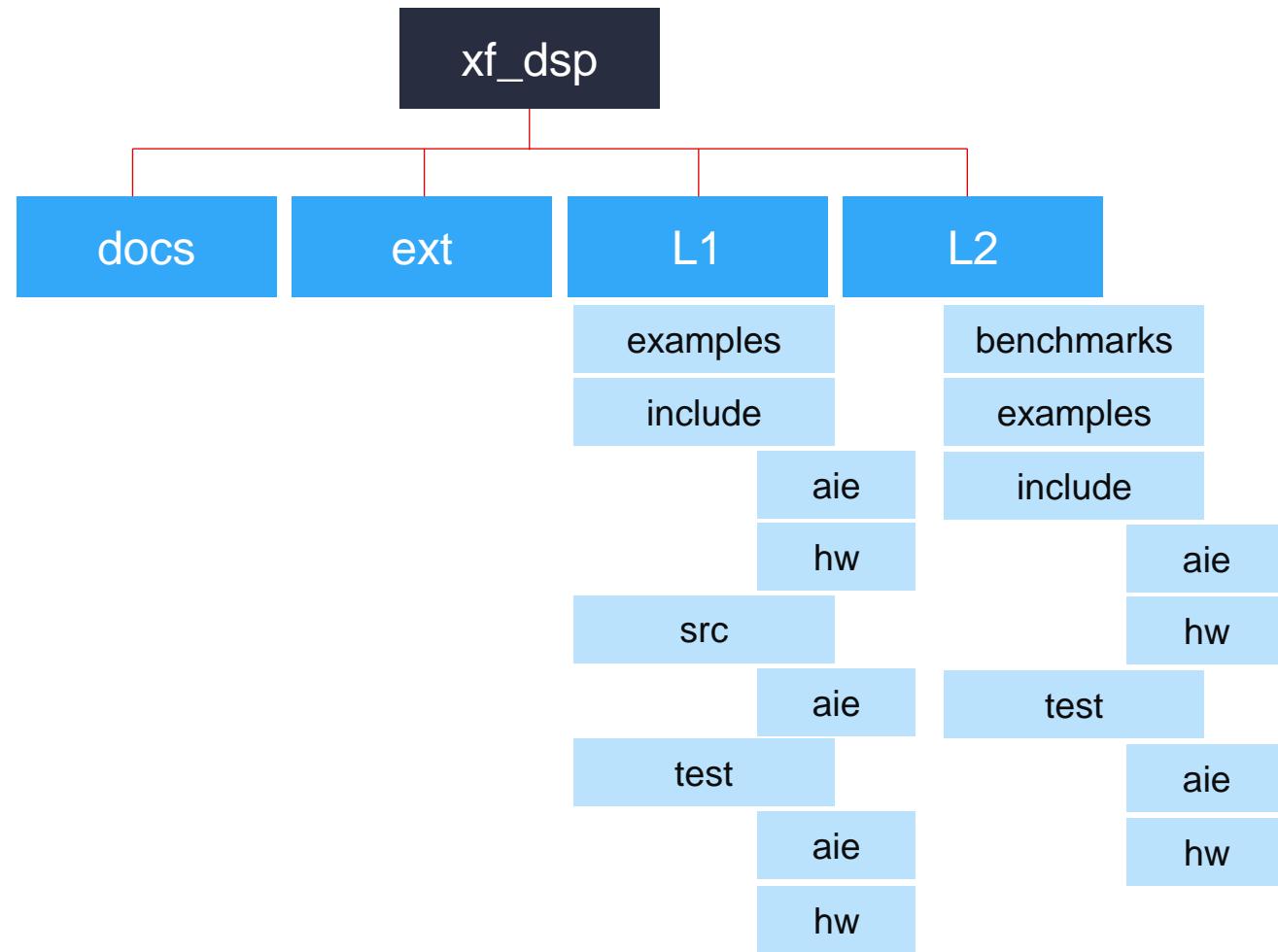
Parameterized Solution Functions

Description	Directory/Subdirectory
Graph class declarations and constants	L2/include/aie
Kernel class definitions, .cpp files, and corresponding .hpp files	L1/src/aie and L1/include/aie
Test bench for the library element	L2/tests/aie/<library_element>
Additional test bench files	L1/tests/aie/inc
Reference model graph classes	L2/tests/aie/common/inc
Reference model kernel classes	L1/tests/aie/inc/ and L1/tests/aie/src
Example wrapper designs	L2/examples



AIE DSPLib Organization

- *examples* Directory
 - Demonstrates the use of the library elements
- *include/hw* Directory
 - Header files for declarations and constants
- *L1/src/aie* Directory
 - .cpp files corresponding to the .hpp files
- *tests/aie* Directory
 - Include and source directories



For more information check out [AIE DSP Library User Guide](#)

Using the AI Engine DSP Library

- Download library from GitHub
 - https://github.com/Xilinx/Vitis_Libraries/tree/master/dsp
- Set the environment variable
 - `export DSPLIB_ROOT=<vitis-libraries-install/dsp>`
- Use the following options in the aiecompiler command to provide the path
 - `-include=$DSPLIB_ROOT/L2/include/aie`
 - `-include=$DSPLIB_ROOT/L1/include/aie`
 - `-include=$DSPLIB_ROOT/L1/src/aie`

Using Library Elements within User-defined Graphs

- Library element to include in your graph: L2 directory, subgraph
- To include a single rate asymmetrical FIR filter:
 - Include `fir_sr_asym_graph.hpp` from the *L2/include/aie* folder
- For `test`, `test.hpp`, and `test.cpp`:
 - Instantiates a parameterized graph *L2/examples/fir_129t_sym*

DSPLib Components

- Direct Digital Synthesis (DDS)/Mixer
- FFT/iFFT
 - (Inverse) Fast Fourier Transform
 - Single channel, decimation in time implementation
 - Configurable point size, data type
- Filters
 - Several variants of Finite Impulse Response (FIR) filters
- Matrix Multiply
 - General Matrix Multiply (GEMM) solution

Filters Considerations

- Data is packetized into windows
- Each window is extended by a margin
- To maximize performance, the window size should be set to the maximum
- Complex multiple factor decision
 - Data movement
 - Latency

FIR Filter templates

Description	Function
Single rate, asymmetrical	dsplib::fir::sr_asym::fir_sr_asym_graph
Single rate, symmetrical	dsplib::fir::sr_sym::fir_sr_sym_graph
Interpolation asymmetrical	dsplib::fir::interpolate_asym::fir_interpolate_asym_graph
Decimation, halfband	dsplib::fir::decimate_hb::fir_decimate_hb_graph
Interpolation, halfband	dsplib::fir::interpolate_hb::fir_interpolate_hb_graph
Decimation, asymmetric	dsplib::fir::decimate_asym::fir_decimate_asym_graph
Interpolation, fractional, asymmetric	dsplib::fir::interpolate_fract_asym::fir_interpolate_fract_asym_graph
Decimation, symmetric	dsplib::fir::decimate_sym::fir_decimate_sym_graph

Supported Combinations of Data Type and Coefficient Type

		Data Type					
		int16	cint16	int32	cint32	float	cfloat
Coefficient Type	int16	Supported	Supported	Supported	Supported	3	3
	cint16	1	Supported	1	Supported	3	3
	Int32	2	2	Supported	Supported	3	3
	cint32	1, 2	2	1	Supported	3	3
	float	3	3	3	3	Supported	Supported
	cfloat	3	3	3	3	3	Supported

1. Complex coefficients are not supported for real-only data types
2. Coefficient type of higher precision than data type is not supported
3. Mix of float and integer types is not supported

Parameters Supported by FIR Filters (1)

Parameter Name	Type	Description	Range
TP_FIR_LEN	unsigned int	Number of taps	4 to 240
TP_RND	unsigned int	Round mode	0 = truncate or floor 1 = ceiling (round up) 2 = positive infinity 3 = negative infinity 4 = symmetrical to infinity 5 = symmetrical to zero 6 = convergent to even 7 = convergent to odd
TP_SHIFT	unsigned int	Number of bits to shift accumulation down by before output	0 to 61
TT_DATA	typename	Data type	int16, cint16, int32, cint32, float, cfloat
TT_COEFF	typename	Coefficient type	int16, cint16, int32, cint32, float, cfloat

Parameters Supported by FIR Filters (2)

Parameter Name	Type	Description	Range
TP_INPUT_WINDOW_SIZE	unsigned int	Number of samples in the input window	Must be a multiple of the number of lanes used (typically 4 or 8); no enforced range, but large windows will result in mapper errors due to excessive RAM use
TP_CASC_LEN	unsigned int	Number of cascaded kernels to use for this FIR	1 to 9; defaults to 1 if not set
TP_DUAL_IP	unsigned int	Use dual inputs (may increase throughput for symmetrical and halfband filters by avoiding load contention by using a second RAM bank for input)	Range 0 (single input), 1 (dual input); defaults to 0 if not set
TT_USE_COEFF_RELOAD	unsigned int	Enable reloadable coefficient feature; additional 'coeff' RTP port will appear on the graph	0 (no reload), 1 (use reloads); defaults to 0 if not set
TT_NUM_OUTPUTS	unsigned int	Number of fir output ports	>1

Vitis Unified Software Platform to Run an Example Design

- Set the environment variable DSPLIB_ROOT
 - `export DSPLIB_ROOT=<vitis-libraries-install/dsp>`
- Import the desired project
 - Example: L2/examples/fir_129t_sym/vitis
- Select the target platform
 - VCK5000 or VCK190
- Set the build configuration as Emulation-AIE and build the project
- Run the simulation and verify the results

Summary

- DSPLib is a configurable library of kernels that can be used to develop applications on Versal ACAP AI Engines
- To install DSPLib:
 - Create an environment variable DSPLIB_ROOT
 - Set the variable to the location of the extracted library directory
 - Use the variable in the Vitis IDE



Application Partitioning on Versal ACAPs

Objectives

- After completing this module, you will be able to:
- Describe what application partitioning is and how it accelerates an application
- Apply application partitioning to your application based on the different compute engines available in Versal® ACAPs

Models of Computation

Sequential Models

- Tasks are executed one by one
 - Turing Machine
 - Finite State Machine
 - Pushdown Automata

Functional Models

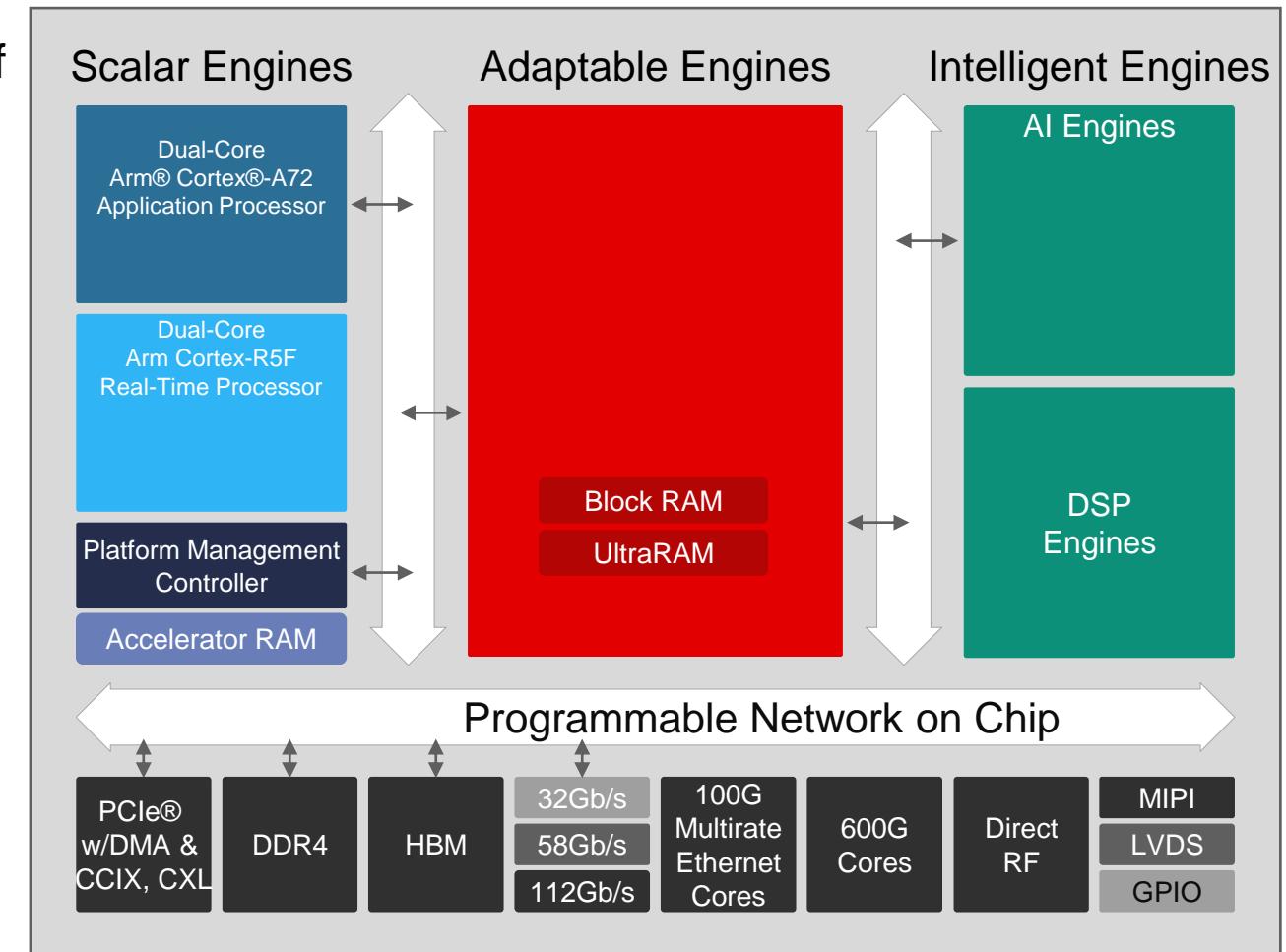
- Task execution will be implementation dependent
 - Lambda Calculus
 - Recursive Function
 - Combinatory Logic
 - Abstract Rewriting System

Concurrent Models

- Tasks are executed in parallel whenever possible
 - **Kahn Process Network**
 - Synchronous Data Flow
 - Actor Model
 - Cellular Automaton
 - Interaction Net
 - Petri Nets

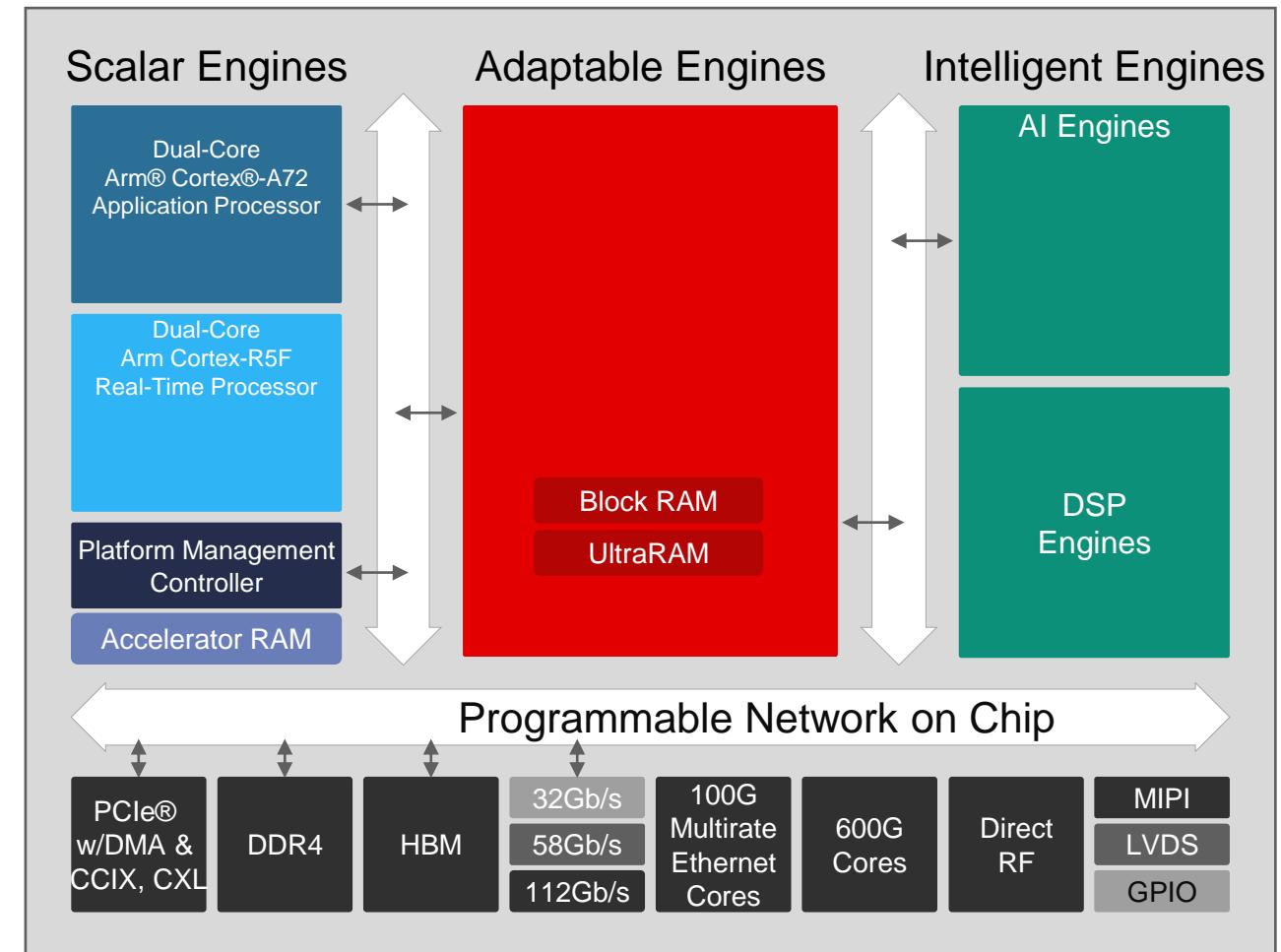
Versal ACAP Supports All Models of Computation

- Combines different engine types with a wealth of connectivity and communication capability
 - NoC enables seamless memory-mapped access
- Supports all the models of computation with the help of different engines
 - Scalar Engines
 - Sequential Models
 - Adaptable Engines
 - Any model of computation
 - AI Engine Tile
 - Sequential MoC
 - AI Engine Array
 - Concurrent MoC
 - Synchronous Data Flow



Versal Adaptive Compute Acceleration Platform

- Adaptable Engines
 - Programmable hardware
 - Bit-level compute
- Scalar Engines
 - Provide benefits of an ARM® processor
 - Control the ACAP
 - Provide compute at the edge
- Intelligent Engines
 - Low latency
 - High throughput
 - Variable precision output
 - AI
 - 5G
 - DSP processing



Heterogeneous compute platform

Why Application Partitioning?

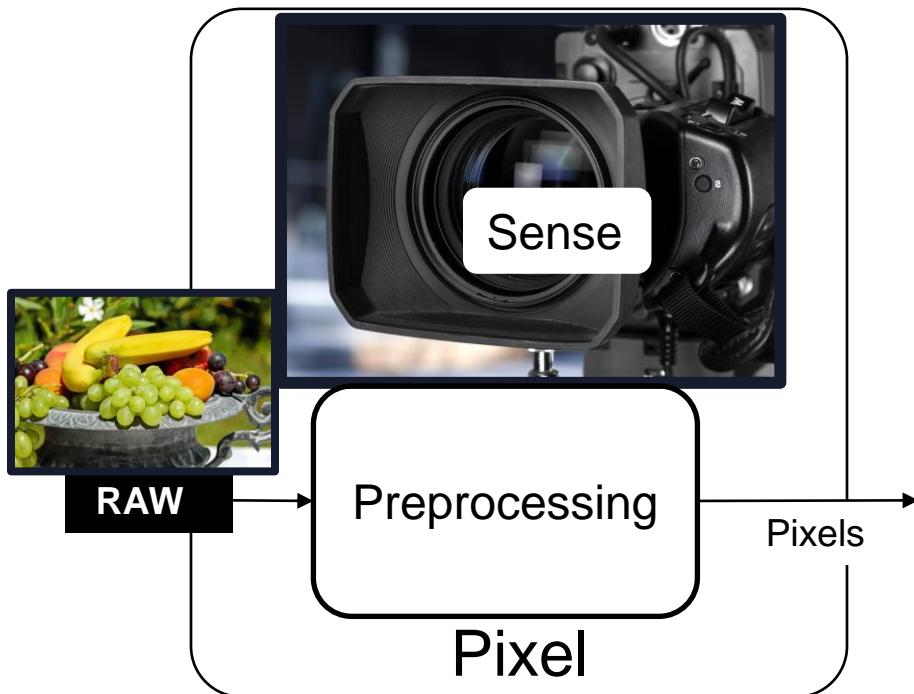
- Different tasks have different compute requirements
- Tasks can be executed in the different compute engines
- Partition the application
 - Parallel performance
 - Faster implementation
 - Pipeline execution

Application Partitioning on the Versal ACAP

- Scalar Engine
 - System control tasks
- Adaptable Engine + NoC
 - Programmable hardware
 - Provides the acceleration
 - Algorithms that allow parallelization
 - Partition application into workloads
- AI Engine
 - Compute-intensive tasks
 - Leverages ACAP adaptable memory hierarchy for AIE partitioning
 - Vector processing

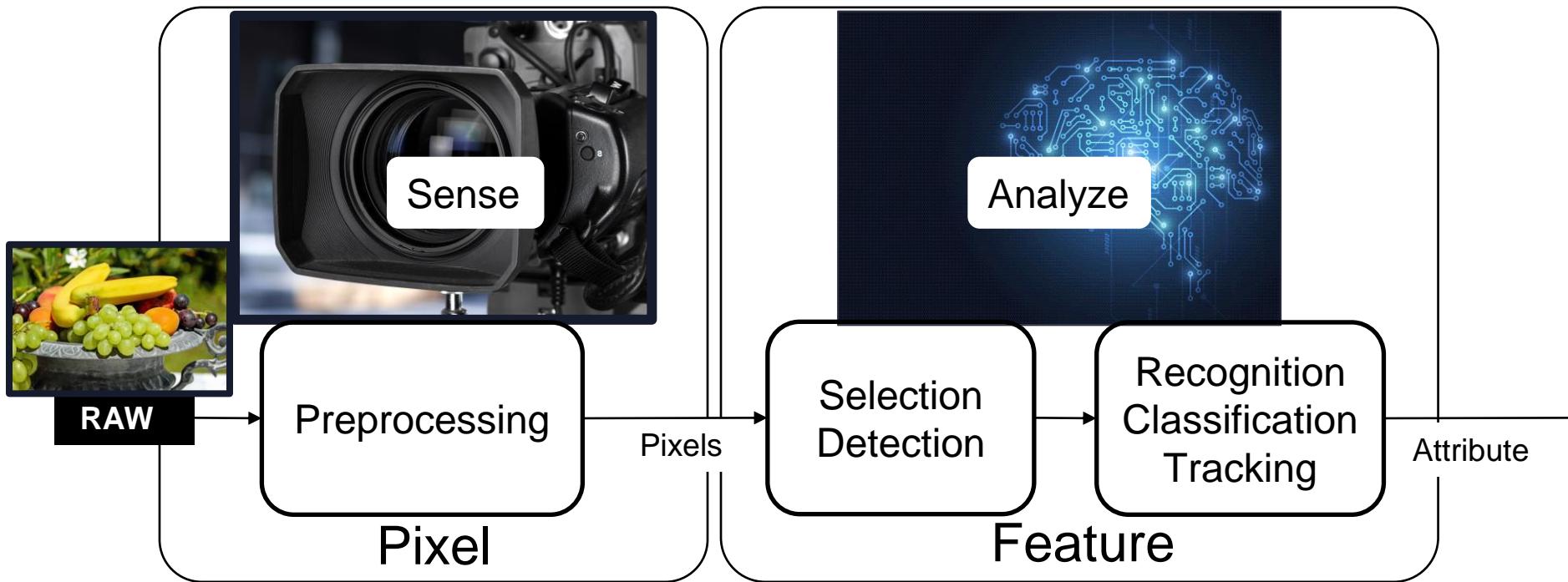
The key is to correctly partition the application

Example: Computer Vision



Step 1 Sense of capture the raw data

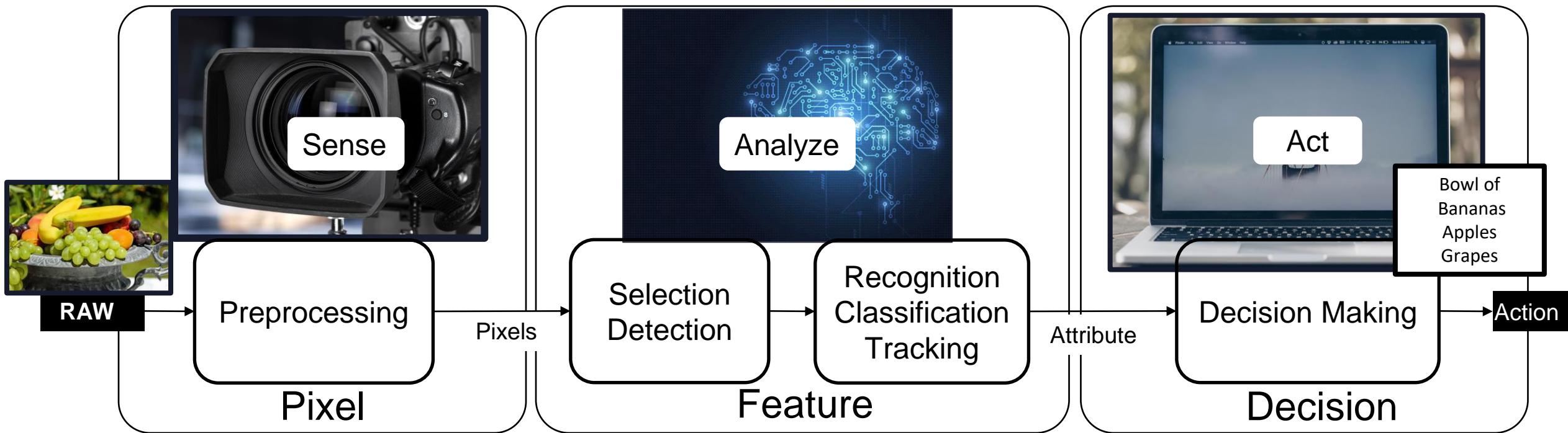
Example: Computer Vision



Step 1 Sense of capture the raw data

Step 2 Analyze data

Example: Computer Vision



Step 1 Sense of capture the raw data

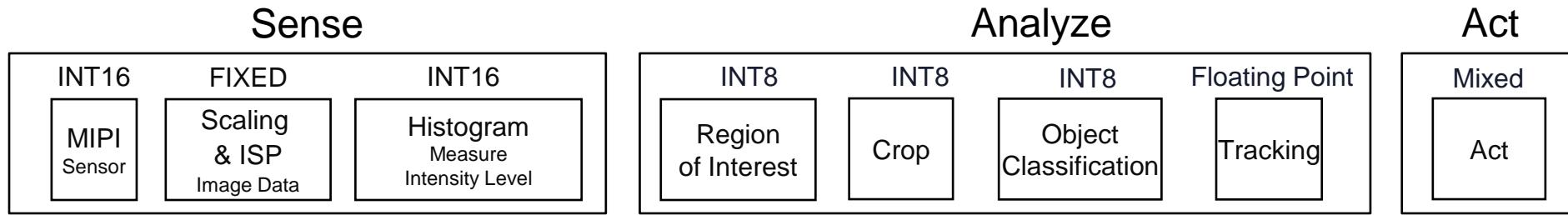
Step 2 Analyze data

Step 3 Make a decision based on the analysis

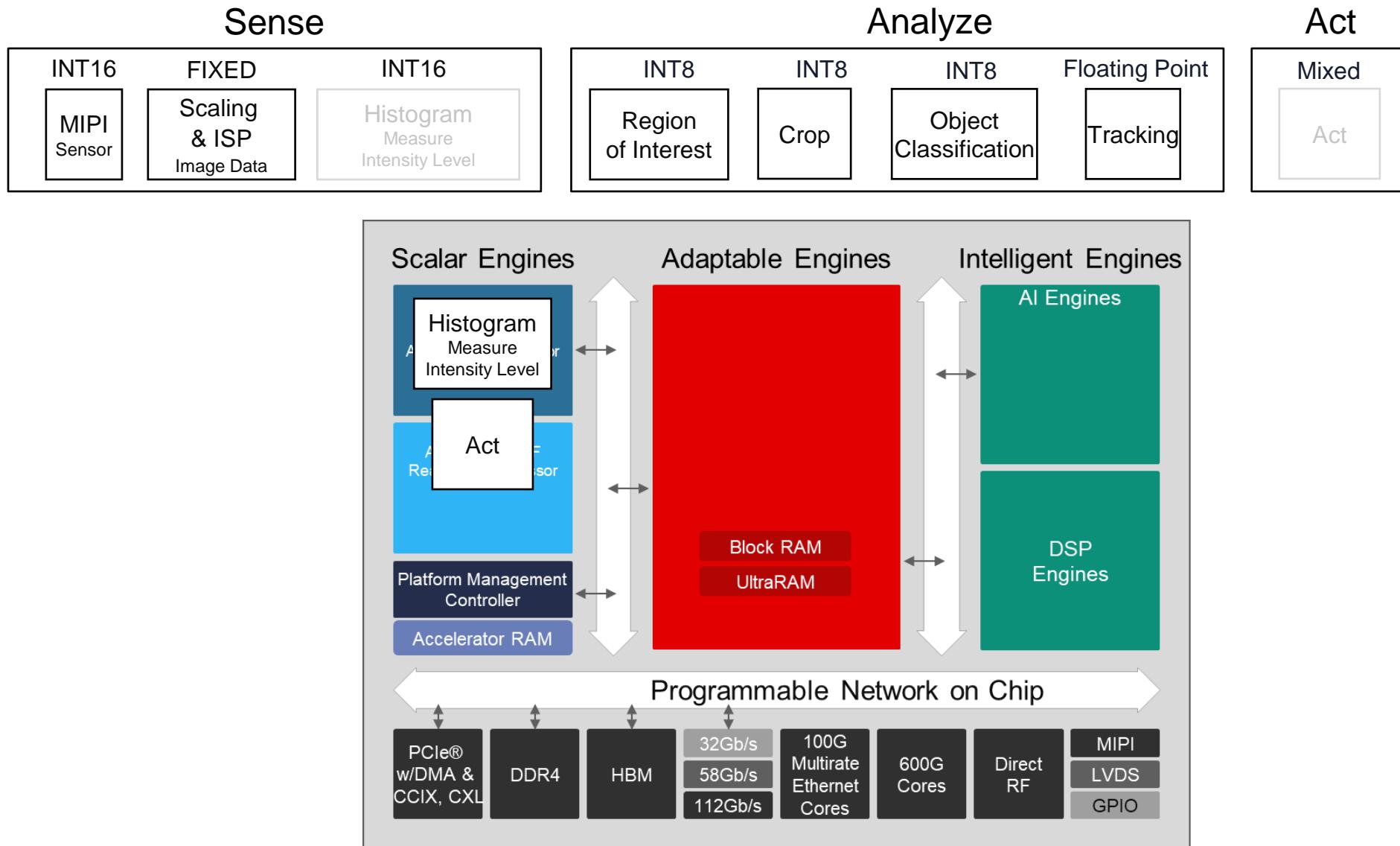
Computer vision pipelines are heterogeneous

Mapping Application Components to the Heterogenous ACAP

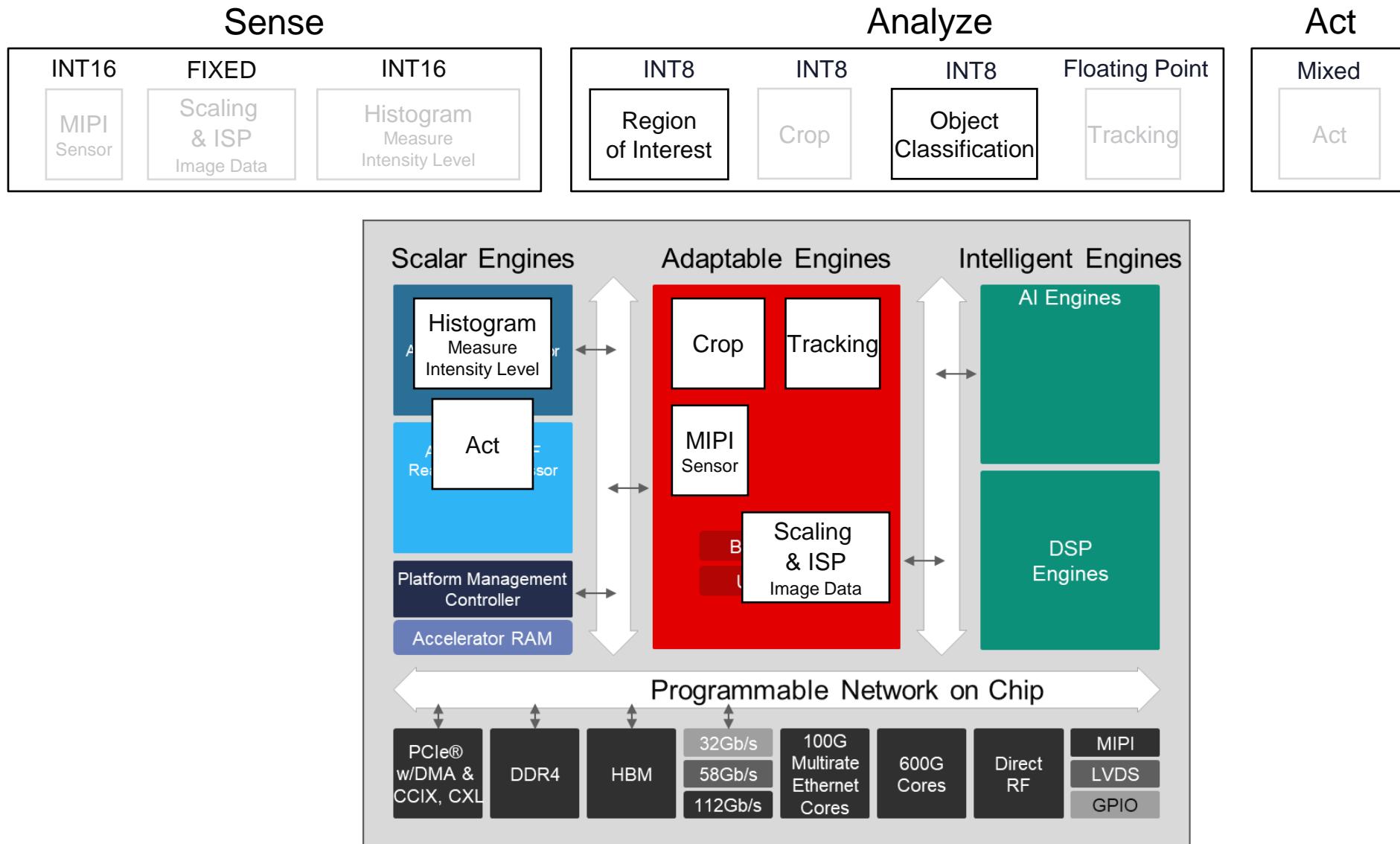
- Correct map the sub-task to each domain
- User needs to understand the application and ACAP



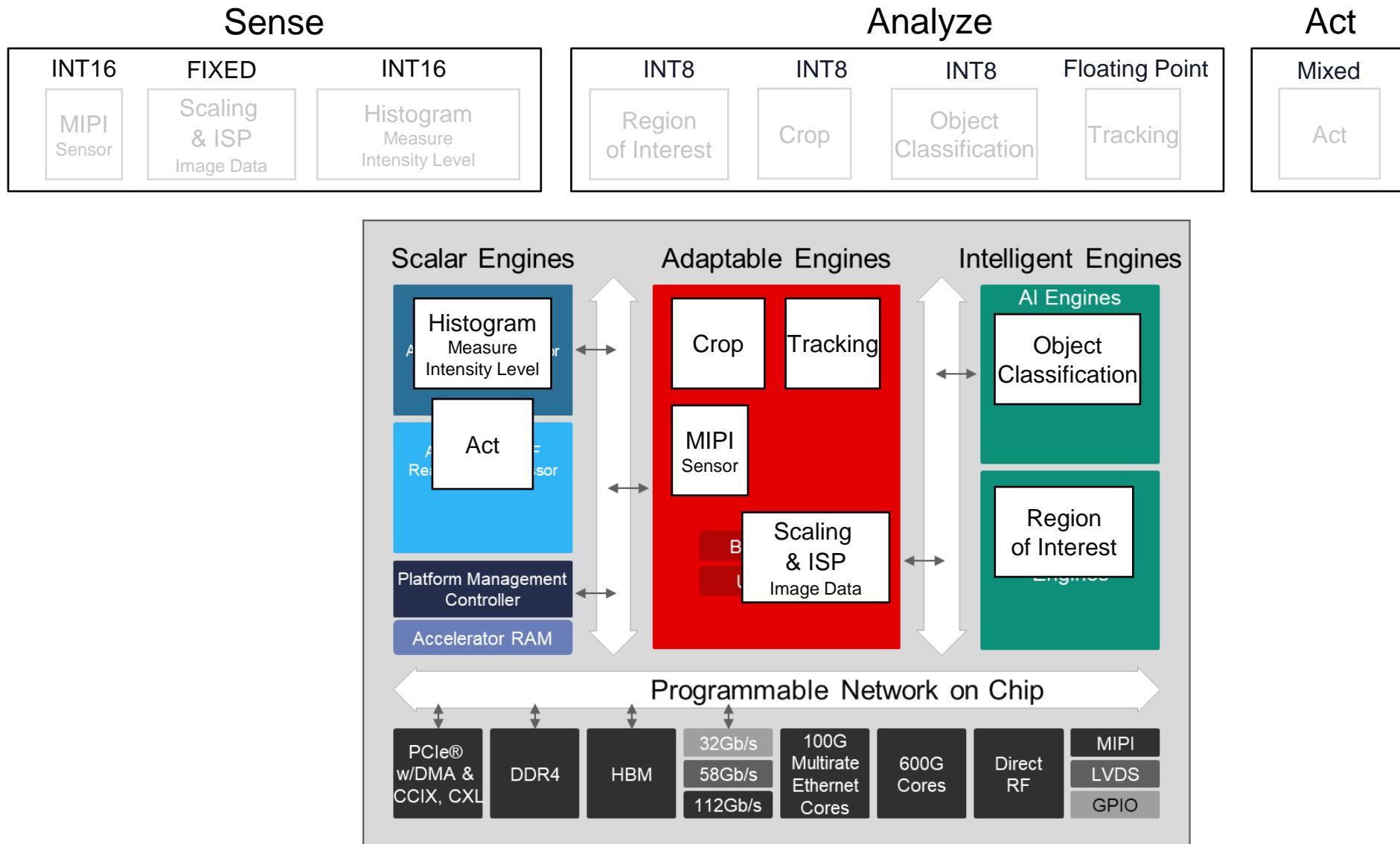
Mapping Application Components to the Heterogenous ACAP



Mapping Application Components to the Heterogenous ACAP



Mapping Application Components to the Heterogenous ACAP



Application Partitioning – Memory Aspects

- External DDR memory
 - Very large memory
 - Good for long vector read/write operations
 - DDR4-3200 max throughput (25.6 GBps)
 - Accessed using the NoC
 - 16GBps per connection
 - Penalty for random accesses
- On-Chip memory
 - Small memories
 - Block RAM
 - UltraRAMs
 - Used as cache between the DDR and AI Engine
 - Efficient memory shuffling
- AI Engine Memory Tile
 - Tiny memories
 - 4 banks x 8KB
 - Access to the AI Engine memory tile happens through the AIE/PL interface
 - AI Engine tile has direct access
 - 2x load and 1x store per cycle
 - Very high bandwidth, if data kept in the Array

Custom memory hierarchy to maximize performance

Summary

- Application partitioning allows
 - Different tasks from the application to be executed in different Versal ACAP engines
 - To fully utilize the performance of the engines based on your tasks
- Computation and data movement requirement are key when partition



Introduction to AI Engine APIs for Arithmetic Operations

Objectives

After completing this module, you will be able to:

- Describe the Versal® AI Engine APIs operations
 - Arithmetic
 - Comparison
 - Reduction
 - Advanced MAC operations

Arithmetic Operations – AI Engine APIs

- The AIE API provides a set of functions that implements arithmetic operations on vector types
- Supported operand combinations are
 - Vector/Vector
 - Value/Vector
 - Vector element references/Vector
- Example:

```
void add(int32 * __restrict out,
         const int32 * __restrict in1,
         const int32 * __restrict in2,
         unsigned count){
    for (unsigned int i = 0; i < count; i += 8) {
        aie::vector<int32, 8> vec = aie::add(aie::load_v<8>(in1 + i),
                                              aie::load_v<8>(in2 + i));
        aie::store_v(out, vec);
    }
}
```



Arithmetic Operations – AI Engine APIs

- Operations that include a multiplication return an accumulator
- The API defines a default accumulation for each combination of types
 - Users can specify a larger number of accumulation bits by explicitly passing an accumulator tag

```
// Default accumulation will be used
auto acc = aie::mul(v1, v2);
// 64b accumulation, at least, will be used
auto acc = aie::mul<acc64>(v1, v2);
// For multiply-add operations, the API uses
// the same accumulation as the given
// accumulator (cannot be overridden)
auto acc2 = aie::mac(acc, v1, v2);
```



One Operand – Arithmetic Operations

Function	Operation	Constraint Type
abs	$(v_i) \rightarrow (v_i)$	Real types: scalar or vector
abs_square	$(re_i+im_i) \rightarrow (a_i = re_i^2+im_i^2)$	Complex types, vector
conj	$(re_i+i.im_i) \rightarrow (re_i-i.im_i)$	Complex types, scalar or vector
mul_square	$(v_i) \rightarrow (a_i = v_i^2)$	Real or complex vector
neg	$(v_i) \rightarrow (-v_i)$	Signed types, scalar or vector

```
aie::vector<int32,16> v1, v2, o1;
aie::vector<cint16,16> c1;
int32 t1, t2;

o1 = neg(v1);
t2 = abs(t1);
auto acc1 = abs_square(c1);
auto acc2 = mul_square(v2);
```



Two Operands – Arithmetic Operations

Function	Operation	Constraint Type
add	$(u_i), (v_i) \rightarrow (a_i = u_i + v_i)$	Any type, scalar or vector, u vector, or accumulator
sub	$(u_i), (v_i) \rightarrow (a_i = u_i - v_i)$	Any type, scalar or vector, u vector, or accumulator
mul	$(u_i), (v_i) \rightarrow (a_i = u_i * v_i)$	Complex types, scalar or vector
mac	$(a_i), (u_i), (v_i) \rightarrow (a_i = a_i + u_i * v_i)$	Real or complex vector
mac_square	$(a_i), (v_i) \rightarrow (a_i = a_i + v_i * v_i)$	Real or complex vector
msc	$(a_i), (u_i), (v_i) \rightarrow (a_i = a_i - u_i * v_i)$	Real or complex vector
msc_square	$(a_i), (v_i) \rightarrow (a_i = a_i - v_i * v_i)$	Real or complex vector
negmul	$(u_i), (v_i) \rightarrow (a_i = -u_i * v_i)$	Real or complex vector

Two Operands – Arithmetic Operations: aie:add()

- Same API with various combinations of inputs
- Element-wise addition of the input accumulator and vector
 - Acc aie:**add**(**const** Acc & acc, **const** Vec & v)
- Addition of a value to all the elements of the input accumulator
 - Acc aie:**add**(**const** Acc & acc, E a)
- Addition of a value to all the elements of the input vector
 - The type of the value and the type of the vector elements must be the same
 - Acc aie:**add**(**const** Vec & v, E a)
- Element-wise addition of the two input vectors
 - The type of the vectors must be the same
 - Acc aie:**add**(**const** Vec & v1, **const** Vec & v2)

Bitwise Operations

Return a vector with the bitwise operation

- bit_and: Element-by-element bitwise AND between a vector and a scalar
- bit_or: Element-by-element bitwise OR between a vector and a scalar
- bit_xor: Element-by-element bitwise XOR between a vector and a scalar

Function	Operation	Type Constraint
bit_and	$S, (v_i) \rightarrow (a \& s_i)$	a and v of same type
bit_or	$S, (v_i) \rightarrow (a s_i)$	a and v of same type
bit_xor	$S, (v_i) \rightarrow (a ^ s_i)$	a and v of same type

```
aie::vector<int32, 16> v1,o1;
aie::vector<int16, 8> v2,v3,o2,o3;
int32 b1, b2;

o1 = bit_and(b1, v1);
o2 = bit_or(b2, v2);
o3 = bit_xor(b2, v3);
```

Comparison Operations

- Minimum and maximum computation
 - Operated between vectors or a scalar and a vector of real type only
- Comparison operators: eq, neq, le, lt, ge, gt
 - Both vectors
 - Scalar and vector
 - Vector and scalar
 - Output: mask
- aie:equal()
 - Compares the two input vectors of the same type and returns equal or not

```
aie::mask<16> m1, m2;
aie::vector<int32,16> v1,v2,v3,c1,c2;
int32 t1,t2;

c1 = max(v1,v2); // With 2 vectors: element-by-element max extraction
c2 = min(t1,v3); // With 1 scalar and 1 vector
m1 = ge(v1,t2); // With 1 vector and 1 scalar → mask output
m2 = eq(v2,v3); // With 2 vectors → mask output
```

Reduction Operations

Function	Description
reduce_add	Returns the sum of the elements within a vector
reduce_max	Returns the maximum value within a vector (real data)
reduce_min	Returns the minimum value within a vector (real data)

```
alignas(16) int16 Init[16] = { 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
aie::vector<int16,16> v = aie::load_v<8>(Init);

// Return type is the same as input vector
int16 Min = aie::reduce_min(v); // Returns Min = 1
int16 Max = aie::reduce_max(v); // Returns Max = 16
int16 Sum = aie::reduce_add(v); // Returns Sum = 136
```



Summary

- AI Engine API-supported arithmetic operations:
 - `aie::abs()`, `aie::abs_square()`, `aie::add()`, `aie::conj()`, `aie::mac()`, `aie::mac_square()`
 - `aie::msc()`, `aie::msc_square()`, `aie::mul()`, `aie::mul_square()`, `aie::sub`
- Bitwise operations:
 - `aie::bit_and()`, `aie::bit_or()`, `aie::bit_xor()`
- Comparison operations:
 - `aie::eq()`, `aie::neq()`, `aie::le()`, `aie::lt()`, `aie::ge()`, `aie::gt()`
- Reduction operations:
 - `aie::reduce_add()`, `aie::reduce_min()`, `aie::reduce_max()`

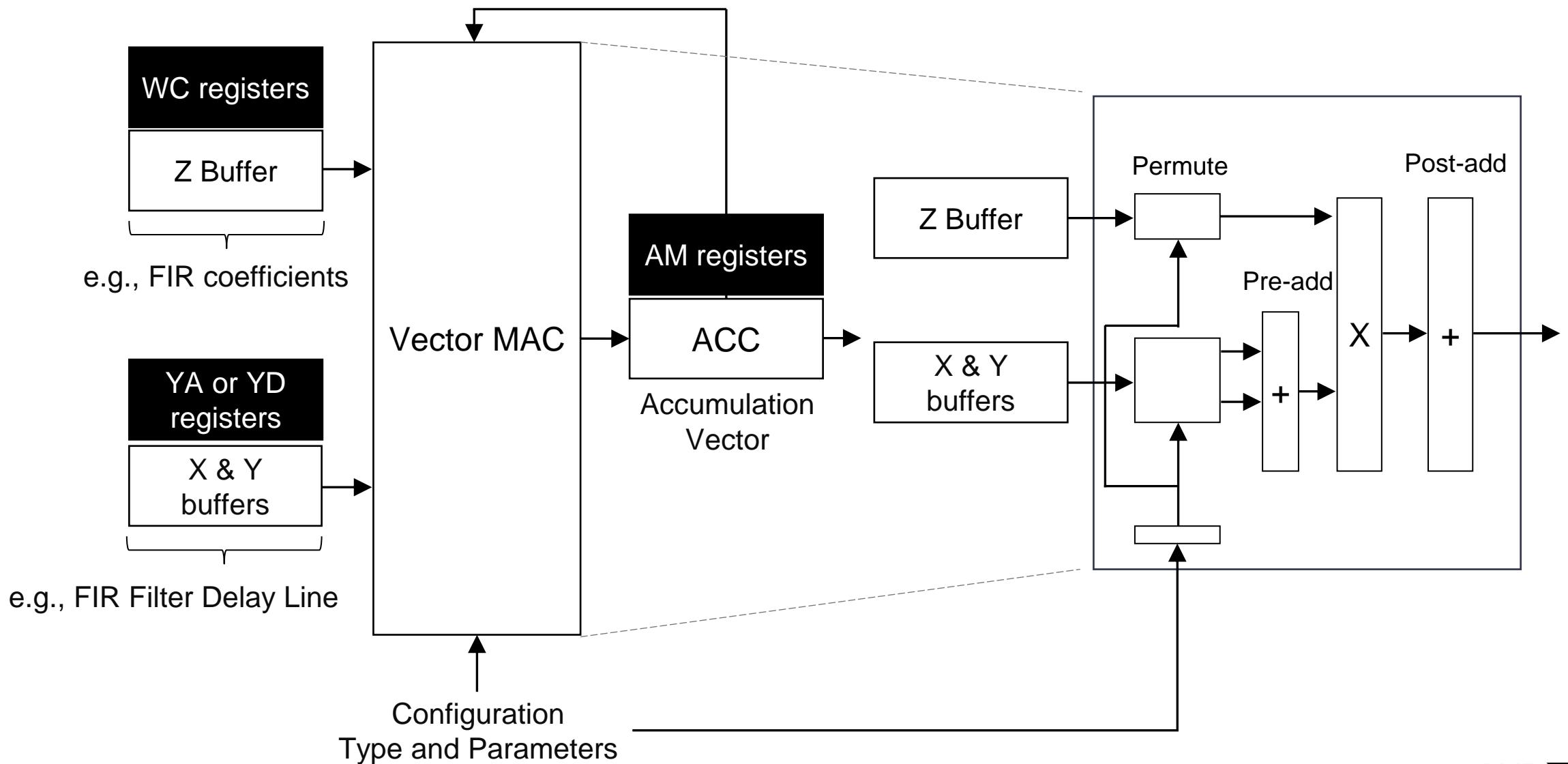
Appendix

- Intrinsic functions for advanced users

Advanced MAC Intrinsics

- Allow more flexibility on lane selection
- Operations are described by using “lanes” and “columns”
- Lanes
 - Outputs generated in parallel
- Columns
 - Number of multiplications per output lane
 - Each of the multiplication results being added together
 - Multiplications per lane is determined by the precision of data and coefficients

Full Lane Addressing Scheme for Vector Operations



Advanced MAC Intrinsics - Example

- **mac8** with 16-bit data and 16-bit coefficients
 - 8 lanes
 - 4 columns
- 32 MACs/cycle
- **mul4** with 32-bit data and coefficients
 - 4 lanes
 - 2 columns
- 8 MACs/cycle

```
acc0 += z00*x00 + z01*x01 + z02*x02 + z03*x03  
acc1 += z10*x10 + z11*x11 + z12*x12 + z13*x13  
acc2 += z20*x20 + z21*x21 + z22*x22 + z23*x23  
acc3 += z30*x30 + z31*x31 + z32*x32 + z33*x33  
acc4 += z40*x40 + z41*x41 + z42*x42 + z43*x43  
acc5 += z50*x50 + z51*x51 + z52*x52 + z53*x53  
acc6 += z60*x60 + z61*x61 + z62*x62 + z63*x63  
acc7 += z70*x70 + z71*x71 + z72*x72 + z73*x73
```

```
acc0 += z00*x00 + z01*x01  
acc1 += z10*x10 + z11*x11  
acc2 += z20*x20 + z21*x21  
acc3 += z30*x30 + z31*x31
```

Advanced MAC Intrinsics Syntax

- `[I] {mul|mac}{2|4|8|16}[_sym|_antisym[_ct|_uct]][_c|_nc|_cn|_cc]`
 - 'I' – accumulator with 80-bit lanes is used for the operation; otherwise, 48-bit lanes
 - 'mul' – multiplication with initialization of accumulators
 - 'mac' – multiplication and addition with running accumulator value
 - 2, 4, 8, or 16 “processing lanes”; i.e., parallel outputs
 - 'sym' and 'antisym' – use pre-adding and pre-subtraction, respectively
 - 'ct' – partial pre-adding and pre-subtraction
 - 'uct' – upshifting, unit (no multiplication) center tap
 - 'n' and 'c' – used to indicate that the complex conjugate will be used for one of the input buffers with complex values
 - 'c' : only the complex input buffer will be conjugated
 - 'cn' : complex conjugate of X (or XY if pre-adding is used) buffer
 - 'nc' : complex conjugate of Z buffer
 - 'cc' : complex conjugate of both X (or XY if pre-adding is used) and Z buffers

Advanced Multiplication MAC Intrinsics – Float

- Advanced float vector multiplication operations contain the fully configurable **fpmac_conf** and some convenient wrappers
- fpmac_conf** function is for fully configurable real multiply-accumulate for single-precision, floating-point vectors
- This capability is introduced to allow flexibility and implement operations on conjugates

```
v4cfloat fpmac_conf ( v4cfloat acc,  
                      v16float xbuf,  
                      int xstart,  
                      unsigned int xoffs,  
                      v4cfloat zbuf,  
                      int zstart,  
                      unsigned int zoffs,  
                      bool ones,  
                      bool abs,  
                      unsigned int addmode,  
                      unsigned int addmask,  
                      unsigned int cmpmode,  
                      unsigned int & cmp  
)
```

Advanced Multiplication MAC Intrinsic – Integer

- Used to perform advanced multiply accumulate (MAC) operations
- Available advanced integer MAC intrinsics

Vector MAC	Vector MAC with pre-adding and conjugation
Vector MAC combined with vector comparisons	Vector MAC with pre-adding and upshifting
Vector MAC with partial pre-adding	Vector MAC with pre-subtraction
Vector MAC with partial pre-subtraction	Vector MAC with pre-subtraction and conjugation
Vector MAC with pre-adding	Vector MAC with pre-subtraction and upshifting

Advanced Self-Multiplication MAC Intrinsic Functions

- Self-multiplication MAC intrinsic functions used for buffer self-multiplication
- 16/32-bit self-multiplication intrinsic functions are available for complex as well as real type vectors
- 16/32-bit complex self-multiplication intrinsic functions can have the following types:
 - No buffer conjugation
 - X and Y buffer conjugation
 - X buffer conjugation
 - Y buffer conjugation

AI Engine Intrinsic User Guide

- Overview
- Deprecated List
- API Reference
 - Data Types
 - Intrinsics
 - Application Specific Intrinsics
 - Load/Store Operations
 - Scalar Operations
 - Vector Conversions
 - Vector Operations
 - Configuration
 - Full Lane Addresssing Scheme
 - Initialization
 - MAC intrinsics
 - Multiplication
 - Self-Multiplication
 - Advanced
 - 16 bit x 16 bit
 - 16-bit Complex x 16-bit Complex
 - No buffer conjugation.

```
v4cacc48 mac4 ( v4cacc48 acc,
                  v32cint16 xbuff,
                  int xstart,
                  unsigned int xoffsets,
                  int xstep,
                  int ystart,
                  unsigned int yoffsets,
                  int ystep
                )
```

Complex multiply-accumulate intrinsic function .

```
acc0 += x00*y00 + x01*y01
acc1 += x10*y10 + x11*y11
acc2 += x20*y20 + x21*y21
acc3 += x30*y30 + x31*y31
```

Parameters

Input/Output	Type	Comments
return	v4cacc48	Returned accumulation vector (4 x cint48 lanes)
acc	v4cacc48	Incomming accumulation vector (4 x cint48 lanes)
xbuff	v32cint16	Input buffer of 32 elements of type cint16
xstart	int	Starting position offset applied to all lanes of input from X buffer
xoffsets	unsigned int	4b offset for each lane in the xbuffer. LSB apply to first lane
xstep	unsigned int	Step between each column for selection in the xbuffer
ystart	int	Starting position offset applied to all lanes of input from xbuffer for the second input
yoffsets	unsigned int	4b offset for each lane in the xbuffers. LSB apply to first lane
ystep	unsigned int	Step between each column for selection in the xbuffers

16-bit complex self-multiplication intrinsics; no buffer conjugation

Reduced Lane Addressing Scheme

- Intrinsics that operate on vectors but do not perform a multiplication
 - Follow a reduced or modified lane selection scheme
 - Such operations are adds, subs, abs, vector compares, or vector selections/shuffles
- Operate mostly on fixed-point numbers
 - Except float select and shuffles because no arithmetic is performed
- Floating-point arithmetic is always done in the floating-point datapath
- These intrinsics perform vector comparisons between data from two buffers with the other parameters and options to allow flexibility

Reduced Lane Addressing Scheme

- Legend:
 - RA: Regular selection scheme
 - RA_16: Selection scheme used for 16-bit numbers
 - na: Not implemented at intrinsics level
 - Fpdp: Floating-point datapath

	i32	i16	ci32	ci16	float	cfloat
select/shuffle	RA	RA_16. Offsets are relative to 32-bit. Start is relative to 16-bit but must be multiple of 2. Square relative to 16-bit.	RA. Start and offsets are relative to a full ci32 (64-bit).	RA. Real and Imag are never split.	RA	RA. Start and offsets are relative to full cfp32 (64-bit)
add/sub	RA	RA_16. Offsets are relative to 32-bit. Start is relative to 16-bit but must be multiple of 2. Square relative to 16-bit.	RA. Start and offsets are relative to a full ci32. (64-bit).	RA_16 modified. Start is doubled to represent a full ci16. (32-bit). Offset follows the RA_16 scheme. 16-bit permute is disabled.	fpdः	fpdः
abs	RA	RA_16. Offsets are relative to 32-bit. Start is relative to 16-bit but must be multiple of 2. Square relative to 16-bit.	na	Na	fpdः	fpdः
cmp	RA	RA_16. Offsets are relative to 32-bit. Start is relative to 16-bit but must be multiple of 2. Square relative to 16-bit.	na	na	fpdः	fpdः

Advanced Float Vector Arithmetic Operations

- Select elements and take absolute value of elements from a real single-precision, floating-point vector
 - `v8float fpabs (v32float xbuf, int xstart, unsigned int xoffs)`
 - `v8float fpabs (v16float xbuf, int xstart, unsigned int xoffs)`
- Parameters:
 - `xbuf`: Input buffer
 - `xstart`: Starting offset for all lanes of X
 - `xoffs`: 4-bit per lane; additional lane-dependent offset for X
- Returns vector with the absolute values

AI Engine Intrinsic User Guide

Overview

Deprecated List

API Reference

- Data Types
- ▼ Intrinsics
 - Application Specific Intrinsics
 - Load/Store Operations
 - Scalar Operations
 - Vector Conversions
 - ▼ Vector Operations
 - Configuration
 - Full Lane Addressing Scheme
 - Initialization
 - MAC intrinsics
 - Reduced Lane Addressing Scheme
 - ▼ Vector Arithmetic
 - ▼ Advanced
 - **Float**
 - Integer
 - Simple
 - Vector Comparisons
 - Vector Lane Selection
 - Native stream access

Overview

Advanced Float Vector Arithmetic Operations. The lane selection scheme is explained after each intrinsic. Note that these intrinsics are a wrapper over the fully configurable fpmac_conf base function, explained here. More information on fpmac_conf can be found here.

Functions

v8float `fpabs` (v8float `xbuf`, int `xstart`, unsigned int `xoffs`)

Take absolute value for single precision real floating point vectors. More...

v8float `fpabs` (v16float `xbuf`, int `xstart`, unsigned int `xoffs`)

Take absolute value for single precision real floating point vectors. More...

v8float `fpabs` (v8float `xbuf`)

Take absolute value for single precision real floating point vectors. More...

v8float `fpabs` (v32float `xbuf`, int `xstart`, unsigned int `xoffs`)

Take absolute value for single precision real floating point vectors. More...

v8float `fpadd` (v8float `acc`, v8float `xbuf`, int `xstart`, unsigned int `xoffs`)

Add for single precision real floating point vectors. More...

v8float `fpadd` (v8float `acc`, v16float `xbuf`, int `xstart`, unsigned int `xoffs`)

Add for single precision real floating point vectors. More...

v8float `fpadd` (v8float `acc`, v8float `xbuf`)

Add for single precision real floating point vectors. More...

These intrinsics are a wrapper over the fully configurable fpmac_conf base function

Advanced Integer Vector Arithmetic Operations

- Function **add16** performs an addition between lanes of xbuff

Overview **Related Pages** **API Reference**

▶ **Scalar Operations**

- ▶ Vector Conversions
- ▶ **Vector Operations**
- ▶ Full Lane Addressing Scheme
- ▶ Initialization
- ▶ MAC Intrinsicss
- ▶ Reduced Lane Addressing Scheme
- ▶ **Vector Arithmetic**
- ▶ Advanced
- ▶ **Float**
- ▶ **Integer**
- abs16
- abs16
- abs32
- abs32
- add16
- add16**
- add16
- add16
- add16
- add16
- add32
- add32
- add32
- add8
- add8
- add8
- add8
- sub16
- sub16
- sub16
- sub16

**v16int32 add16 (v16int32 xbuff,
 int xstart,
 unsigned int xoffsets,
 unsigned int xoffsets_hi,
 int ystart,
 unsigned int yoffsets,
 unsigned int yoffsets_hi
)**

Performs an addition between lanes of xbuff.

```
for (int i = 0; i < 16; i++)
    idx = f( xstart, xoffsets[i] );
    idy = f( ystart, yoffsets[i] );
    o[i] = x[idx] + x[idy]

xoffsets, xoffsets_hi, yoffsets, yoffsets_hi have 8 offset values each. 4 bits per offset.
For Example: for v16int32 output type, idx for output_lane_0 = f(xstart,xoffsets[0])
For Example: for v16int32 output type, idx for output_lane_15 = f(xstart,xoffsets_hi[7])
In case of v32int16, 1 offset is used for 2 adjacent lanes.
For more information on how the function f() selects data from the buffers refer to Lane selection note below.
```

Parameters

Input/Output	Type	Comments
return	v16int32	Value of each lane is the result of an addition between lanes of xbuff where the result of lane 0 goes to lane 0 of the output.
xbuff	v16int32	Input buffer of 16 elements with 32-bit precision
xstart	int	Starting position offset applied to all lanes of input from X buffer
xoffsets	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to first lane
xoffsets_hi	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to 8th lane
ystart	int	Starting position offset applied to all lanes of input from xbuffer for the second input
yoffsets	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to first lane
yoffsets_hi	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to 8th lane

Advanced Vector Compare Functions

- Vector max/min
 - Performs comparison
 - Returns the result of the comparison as a lane in the output vector
- Vector maxdiff
 - Performs an integer subtraction between the lanes of X/Y
 - Returns the maximum between zero and the subtraction result as a lane in the output vector
- Vector Comparison
 - Performs a comparison between lanes
 - Returns the result of the comparison as a bit in the return word

Advanced vector compare functions allow flexible lane selection before the comparison

Advanced Vector Compare Function Examples

```
v8float fpmax ( v8float      acc,  
                  v32float     xbuf,  
                  int          xstart,  
                  unsigned int xoffs  
)
```

Float Vector Max/Min

```
v16int32 maxdiff16 ( v32int32    xbuff,  
                      int          xstart,  
                      unsigned int xoffsets,  
                      unsigned int xoffsets_hi,  
                      int          ystart,  
                      unsigned int yoffsets,  
                      unsigned int yoffsets_hi  
)
```

Float Vector Maxdiff

```
unsigned int fpge ( v8float      acc,  
                     v32float     xbuf,  
                     int          xstart,  
                     unsigned int xoffs  
)
```

Unsigned Integer Vector Comparison

Advanced Vector Lane Selection

- Select
 - Selects between the first set of lanes or the second one according to the value in 'select'
- Shuffle
 - Selects from a single input according to the start/offset computation

Advanced vector lane selection can be performed for float and integer types

Advanced Vector Lane Selection: Select

- **fpselect16** function performs a floating-point selection between lanes of xbuff

▼ API Reference	
▶ Data Types	
▼ Intrinsics	
▶ Application Specific Intrinsics	
▶ Load/Store Operations	
▶ Scalar Operations	
▶ Vector Conversions	
▼ Vector Operations	
▶ Full Lane Addressing Scheme	
▶ Initialization	
▶ MAC Intrinsics	
Reduced Lane Addressing Scheme	
▶ Vector Arithmetic	
▶ Vector Compares	
▼ Vector Lane Selection	
▼ Advanced	
▼ Float	
fpselect16	
fpselect16	
fpselect16	
fpselect8	
fpselect8	
fpselect8	
fpselect8	
fpshuffle16	
fpshuffle16	
fpshuffle8	
fpshuffle8	
▶ Integer	
▶ Simple	

```
v16float fpselect16 ( unsigned int select,
                      v32float      xbuff,
                      int           xstart,
                      unsigned int  xoffsets,
                      unsigned int  xoffsets_hi,
                      int           ystart,
                      unsigned int  yoffsets,
                      unsigned int  yoffsets_hi
)
```

Input/Output	Type	Comments
return	v16float	Value of each lane is the result of a floating point selection between lanes of xbuff where the result of lane 0 goes to lane 0 of the output.
select	unsigned int	Value of each bit selects from the value to be placed in the corresponding vector position
xbuff	v32float	Input buffer of 32 elements with single precision
xstart	int	Starting position offset applied to all lanes of input from X buffer
xoffsets	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to first lane
xoffsets_hi	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to 8th lane
ystart	int	Starting position offset applied to all lanes of input from xbuffer for the second input
yoffsets	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to first lane
yoffsets_hi	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to 8th lane

Advanced Vector Lane Selection: Shuffle

- **fpshuffle16** function performs a floating-point shuffle between lanes of xbuff

```
v16float fpshuffle16 ( v32float      xbuff,
                      int          xstart,
                      unsigned int xoffsets,
                      unsigned int xoffsets_hi
)
```

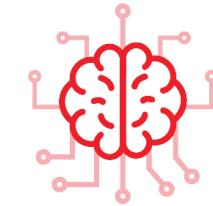
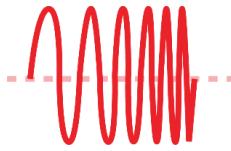
Input/Output	Type	Comments
return	v16float	Value of each lane is the result of a floating point shuffle between lanes of xbuff where the result of lane 0 goes to lane 0 of the output.
xbuff	v32float	Input buffer of 32 elements with single precision
xstart	int	Starting position offset applied to all lanes of input from X buffer
xoffsets	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to first lane
xoffsets_hi	unsigned int	4b offset for each lane, applied to the xbuffer. LSB apply to 8th lane



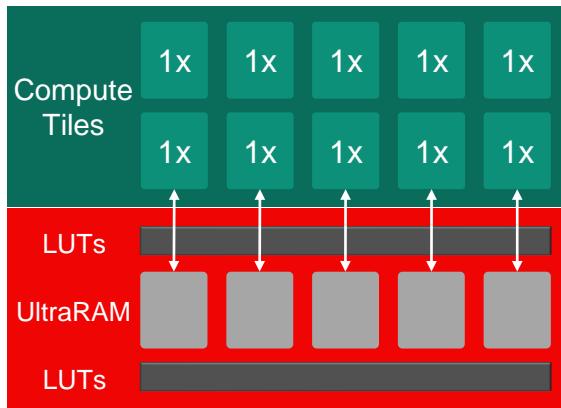
AIE-ML Architecture

AMD
together we advance_

Intelligent Engines Optimized for Any AI Application

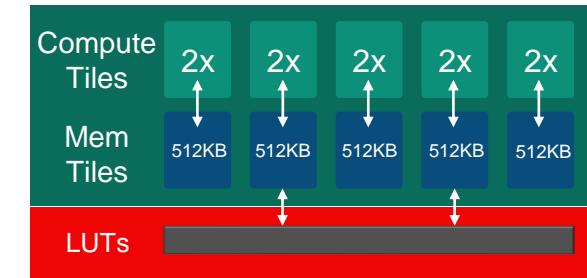


AIE Architecture



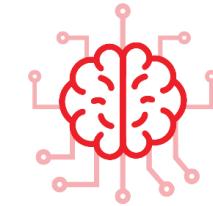
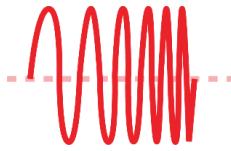
- Optimized for signal processing and ML
- Flexibility for high performance DSP applications
- Native support for INT8, INT16, INT32, FP32

AIE-ML Architecture

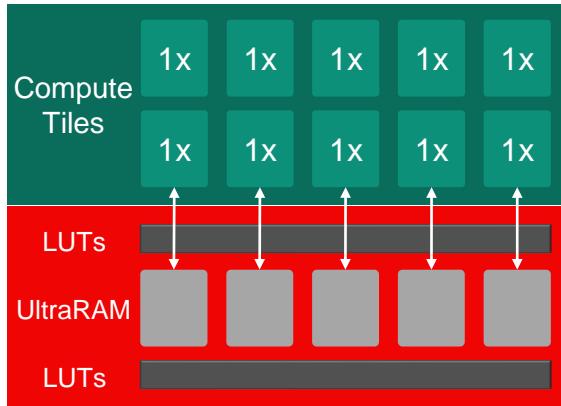


- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16
- Fine grained sparsity HW optimization
- Enhanced FFT & complex math support

Intelligent Engines Optimized for Any AI Application



AIE Architecture



- Optimized for signal processing and ML
- Flexibility for high performance DSP applications
- Native support for INT8, INT16, INT32, FP32

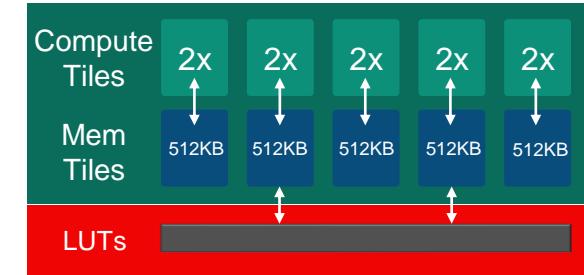
OPS / Tile

256	INT4	1024
256	INT8	512
64	INT16	128
No native support		BFLOAT16 256
16	INT32	32*
16	FP32	51*

KB / Tile

32	Data Memory	64
16	Program Memory	16

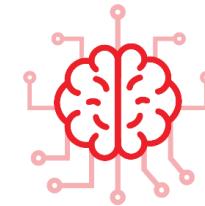
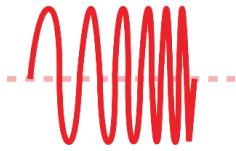
AIE-ML Architecture



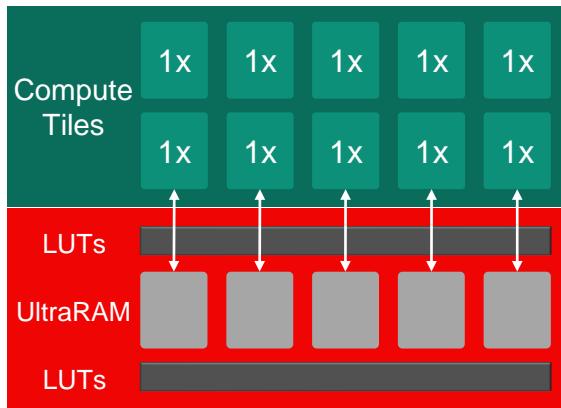
- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16
- Fine grained sparsity HW optimization
- Enhanced FFT & complex math support

*Via software emulation (FP32 not IEEE-754 compliant)

Intelligent Engines Optimized for Any AI Application



AIE Architecture



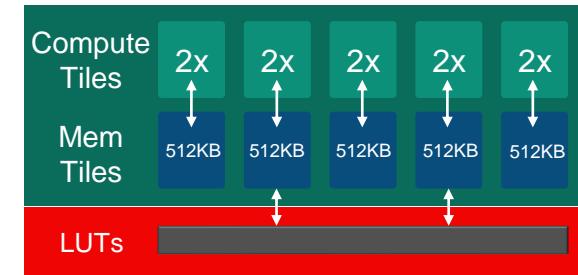
**2x INT8/16 OPs/Tile
4x INT4 OPs/Tile**

Reduced data movement

Reduced AI PL Footprint

- Optimized for signal processing and ML
- Flexibility for high performance DSP applications
- Native support for INT8, INT16, INT32, FP32

AIE-ML Architecture



- Optimized for ML Inference Applications
- Maximum AI/ML compute with reduced footprint
- Native support for INT4, INT8, INT16, bfloat16
- Fine grained sparsity HW optimization
- Enhanced FFT & complex math support

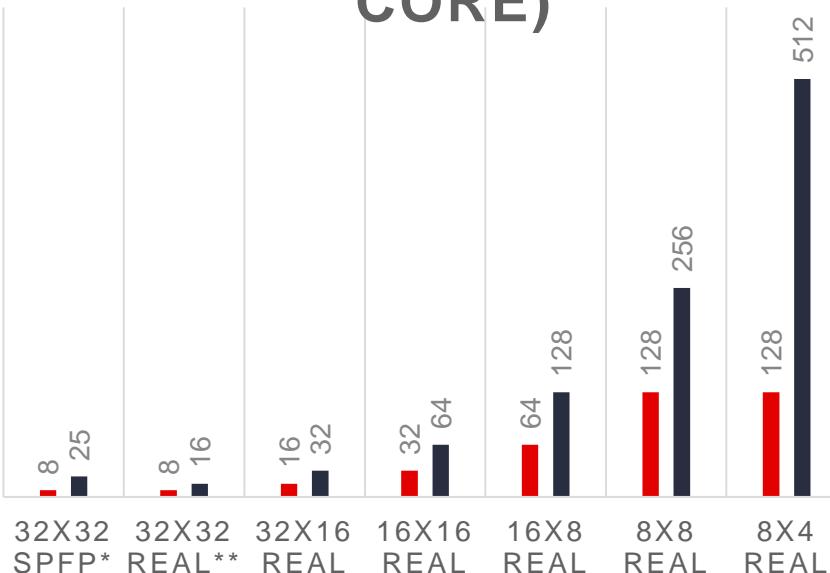
Versal® AIE-ML offers 2x AI Performance per Watt

Multi-Precision Support

█ AIE
█ AIE-ML

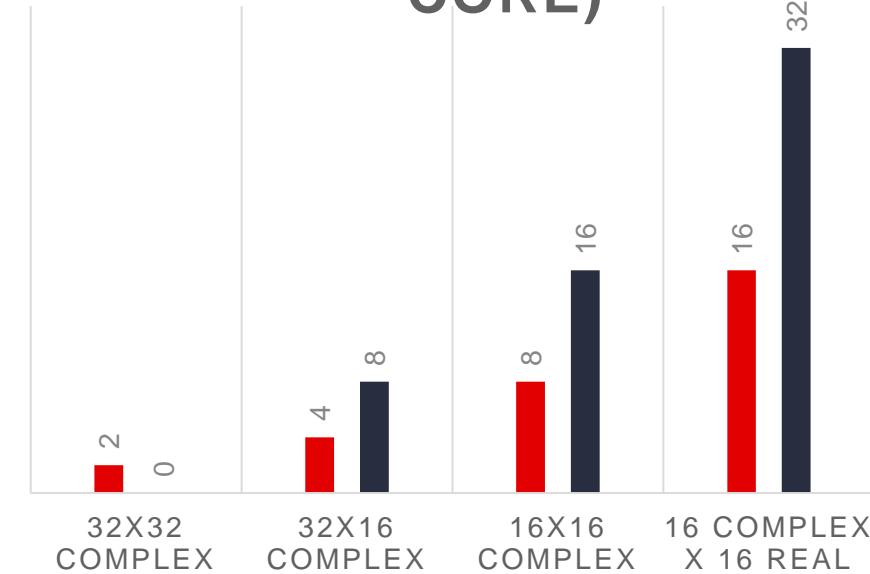
Real Data Types

MACS / CYCLE (PER CORE)



Complex Data Types

MACS / CYCLE (PER CORE)

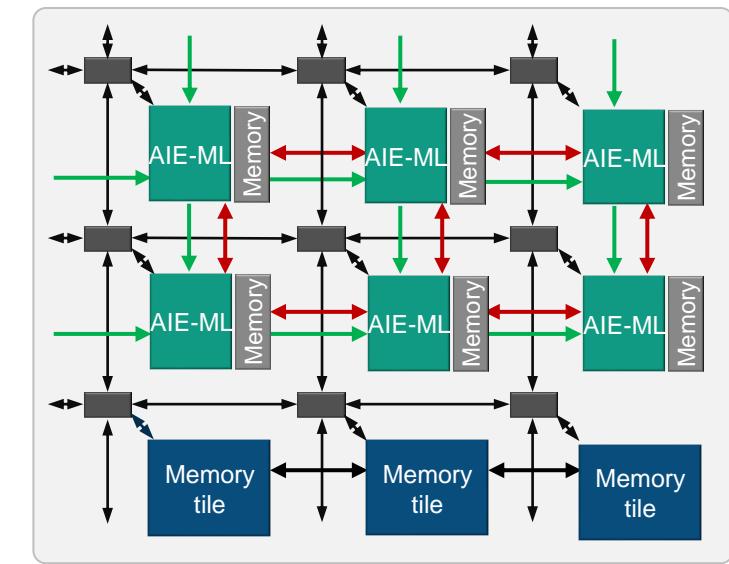


* 32 SPFP x 32 SPFP emulated using BFLOAT16. Not IEEE-754 compliant. Low accuracy mode

**32 Real x 32 Real is emulated

AIE-ML Array

- AIE-ML Array includes Memory tiles at the bottom of the array
 - 512 KB
 - Accessible via Interconnect
- All lines of compute tiles are identical in AIE-ML
 - No snake shape of the cascade interface
 - Always from left to right and top to bottom
 - Local memory is always East
 - Accessible neighbor memories are always North, South and West
- Interfaces with PL and NoC stays the same
 - However, no direct access to the AIE-ML tiles
 - Interface through the AIE-ML memory tiles

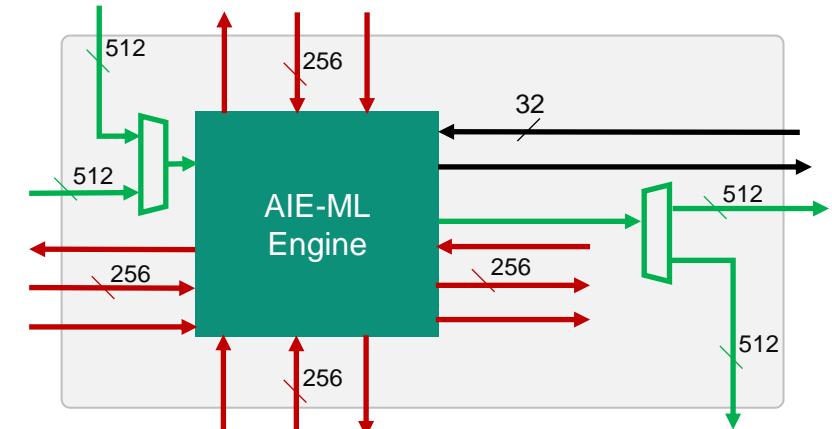


→ Memory Interface
→ Stream Interface
→ Cascade Interface

For more information check out [AM020](#)

AIE-ML Interfaces

- AIE-ML tile has 2 directions cascade
 - Input: North and West
 - Output: South and East
- 512-bit width cascade interface
 - 16 lanes of 32-bit data
 - 8 lanes of 64-bit data
- Direct AXI4-Stream interface
 - 1 x 32-bit input
 - 1 x 32-bit output



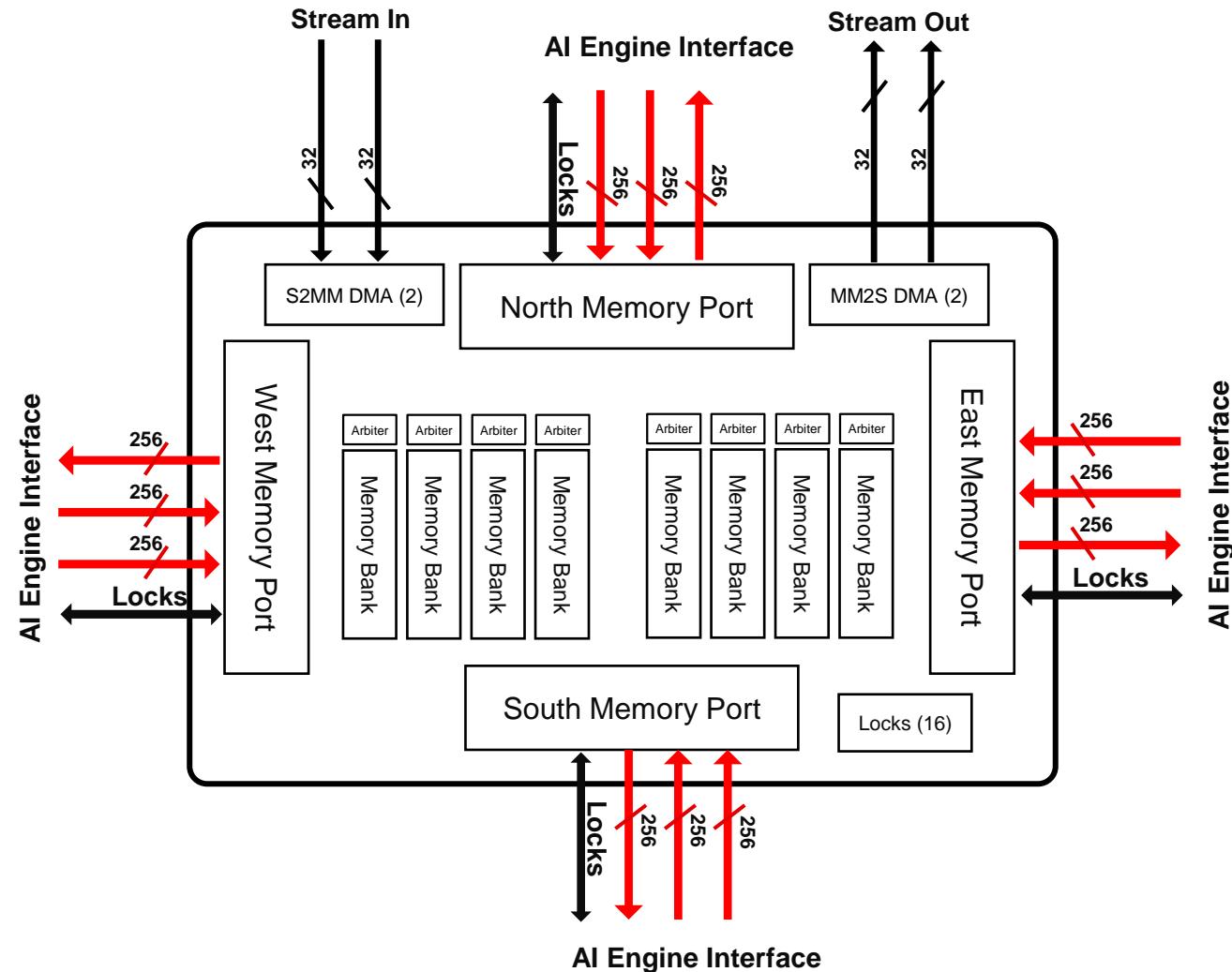
→ Memory Interface
→ Stream Interface
→ Cascade Interface

AIE-ML Tile

- 256 (int8 x int8) multipliers
- Native support for INT4 and BFLOAT16
- No native support for INT32
 - Emulated via software
- No native support for SPFP32
 - Can be emulated using BFLOAT16 (not IEEE-754 compliant).
- Support for 2:4 sparsity
- No support for 5G specific extensions
- No support for scalar non-linear functions
 - For example sqrt, Sin, Cos, and InvSqrt
 - Floating point / integer conversions

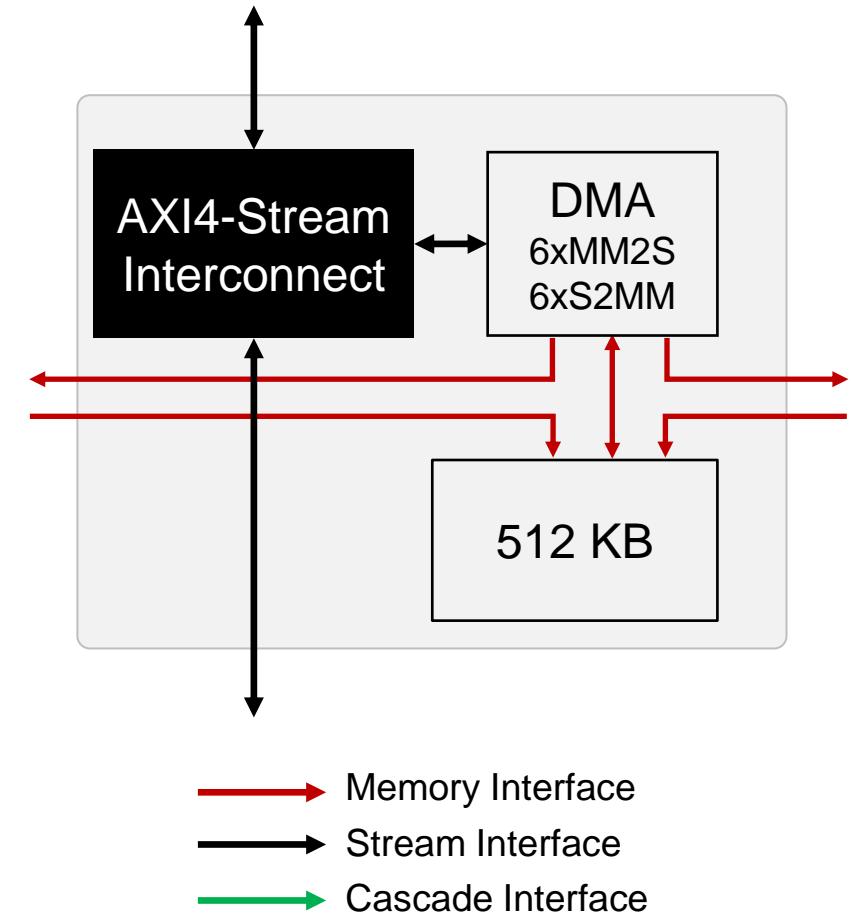
Memory Module

- 64KB of local memory
- Eight memory banks
 - 8KB each
 - 512-word x 128-bit single port memory
 - Interleaved to form 4 groups
- DMA features
 - Address generation to support 4D addressing
 - S2MM finish on TLAST and out-of-order packets
 - Compression (MM2S) and decompression (S2MM)
- Semaphore locks
 - 6-bit unsigned lock state



AIE-ML Memory Tile

- 512KB memory
- Arranged into 16 banks, per bank
 - 9 read interfaces
 - 9 write interfaces
 - Arbitration
- MM2S and S2MM with 6 channels each
 - 32-bit stream interfaces
 - 128-bit memory interfaces
 - Support 4D tensor address generation
 - Supports compression and decompression of the data
- AXI4-Stream interconnect with North and South
- 1 or 2 rows at the bottom of the array



For more information check out [AM020](#)

AI Engine vs AI Engine-ML summary

	AI Engine	AI Engine-ML
Array structure	Checkerboard	All lines identical
Cascade Interface	384-bit wide Horizontal Direction	512-bit wide Horizontal and Vertical Directions
Tile Stream Interfaces	2x 32-bit In and 2x 32-bit Out	1x 32-bit In and 1x 32-bit Out
Memory Load/Store (per cycle)	512/256-bit	512/256-bit
Int8 * Int8 Multipliers	128	256
Int4 * Int8 Multipliers	128	512
Format Native Support	int8/16/32, cint16/32, SPFP32	int4/8/16, cint16, BFloat
Non-Linear Functions in scalar processor	Yes	No
Tile Local Memory	32KB	64KB
Tile Local Memory DMA	-	Support for 4D addressing modes S2MM finish on TLAST and out-of-order packets Compression/Decompression
Local Memory Locks	Boolean	Semaphore
Memory Tiles	No	Yes (512KB)

Disclaimer & Attribution

Timelines, roadmaps, and/or product release dates shown in these slides are plans only and subject to change.

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

©2023 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Athlon, CDNA, EPYC, Infinity Fabric Radeon, RDNA, ROCm, Ryzen, Ryzen Threadripper, Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Vitis, Virtex, and Zynq and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft is registered trademark of Microsoft Corporation in the US and other jurisdictions. SPEC®, SPECCrate®, SPECint and SPEC CPU® are registered trademarks of the Standard Performance Evaluation Corporation. See www.spec.org for more information. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

AMD