AMD Versal™ Adaptive SoC AI Engine for DSP Architecture

Version: 1.0

Author: Adam Taylor

1. Contents

1. Contents 1
2. Change Log 2
3. Introduction 3
4. Core Maths of AI and DSP 3
5. Fixes and Floating Point Types 4
6. AMD Versal™ Adaptive SoC Architecture Introduction 5
7. AMD Versal™ Adaptive SoC Products 6
8. AMD Versal™ Adaptive SoC Architecture Generations 7
9. Deep Dive on the AMD Versal™ Adaptive SoC AI Engine 7
10. AI Engine Generations 9
11. Programming Concepts for the AIE 10
12. Developing AI Engine Applications 12
13. Why use the AI Engine for DSP-based Applications 13
14. Example AI Engine for DSP Applications 15

2. Change Log

Version	Notes
1.0	Initial issue

3. Introduction

Digital Signal Processing is critical to modern applications, from RF communications (where it is used for beamforming), to robotics, test equipment, and medical imaging.

As solutions addressing these markets are faced with competing demands, performance demands are increased (e.g. higher throughput). These performance demands add to the increasing pressure for a better power performance. Simply put, solution architects want to achieve more processing using less power to achieve that performance.

This technical note is going to explore how AMD Versal™ adaptive SoCs and its AI Engines are able to help solution developers achieve these demands.

We will examine the following aspects of using the Versal AI Engine for DSP platform:

1. How are mathematical operations for AI and DSP based upon the same foundational operation.
2. Introduce the Versal devices and the differences between generations.
3. Detailed look at the AI Engine structure.
4. Examine use cases for AI Engine in DSP applications.
5. Examine programming models to use the AI Engine for DSP in our applications.
6. Solution Example.

4. Core Maths of AI and DSP

While AI and DSP may initially seem like very different applications, the underlaying mathematics for many techniques are similar in that they are based upon mathematical operations on large arrays.

These operations include dot products, Finite Impulse Response, Fast Fourier Transforms and Matric Multiplication, and at the heart of these operations is the Multiply Accumulate (MAC). To get the best performance from this MAC, it is often best to have the data in closely coupled memory.

The simplest element of a MAC is the dot product. That is a MAC loop over two vectors of equal length.

$$a \cdot b = \sum_{i=1}^{N} a_i \, b_i$$

For a simple example: a = [1,2,3], b = [4,5,6] the dot product is a. b = 1.4 + 2.5 + 3.6 = 32.

The MAC is critical for DSP and AI, as for neural networks, many operations compute weighted sum, plus a bias e.g. dot(a,b)+bias.

For one-dimensional convolution, which is represented by the equation $y[n] = \Sigma_k x[n-k] \cdot h[k]$, the output is a dot product between a sliding window of the input X and the vector H (which is a predefined constant). Typical use cases of one-dimensional convolution is FIR filtering.

Two-dimensional convolution is typically used for image filtering and convolutional neural networks. Each output pixel is the result of a dot product of several local pixels and kernel weights. In effect, this results in many sliding dot products over the image.

To implement Fast Fourier Transforms (FFT), the DFT matrix is broken into a number of sparse stages called butterflies. This turns a large dot product calculation into a sequence of much smaller butterflies, which are implemented as smaller dot products, plus twiddle multiplies. This makes the FFT a sequence of dot products, multiplications, and additions.

While the dot product is key to many AI and DSP operations, working with real world signals requires many hundreds of millions of operations. As such, we need to be able to efficiently perform as many calculations as possible on a clock cycle.

This is where Single Instruction Multiple Data comes into its own. Single Instruction Multiple Data operates exactly as its name indicates- one instruction operates on many values in parallel instead of one value at a time. These parallel data values are often called vectors or lanes.

SIMD therefore enables an increase in throughput. We can get up to N x improvement for N lanes. As we have fewer fetch decode execute cycles, the performance per Watt is increased and as we are doing defined operations (loops and vector operations), the timing becomes more predictable.

This combination of the MAC and SIMD is critical for many advanced DSP and AI solutions, enabling them to achieve not only high performance, but also efficient power utilization.

5. Fixes and Floating Point Types

When implementing AI or DSP solutions, the types used for the mathematical operation such as the dot product defined above can have a large impact on performance and accuracy. At the highest level, we can group types into either fixed or floating point types.

Fixed point represents numbers as integers with an implied binary point. As such, you trade a dynamic range for tight control of precision, latency, and resources. Fixed point implementations map brilliantly onto FPGA fabrics and DSP slices, giving high throughput, low power, and deterministic behavior when your signal ranges are well understood.

In floating point, each number contains a mantissa and exponent, while fixed point vectors can be arbitrary sizes. Floating points are normally 32 bits. The ability to float the decimal point provides a wide dynamic range and easier algorithm development with fewer worries about scaling and overflow. However, the downside of floating point is a more complex implementation that can impact the performance.

Commonly used fixed and floating point types in AI and DSP applications include:

- **Int4** — 4-bit integer for ultra-compact fixed-point quantization.
- **Int8** — 8-bit integer commonly used fixed point quantization.
- **Int16** — 16-bit integer often used in many DSP applications.
- **CInt16** — Complex fixed point: 16-bit real + 16-bit imaginary per sample.
- **CInt32** — Complex fixed point: 32-bit real + 32-bit imaginary per sample.
- **FP32** — 32-bit floating point often called single precisions.
- **Bfloat16 (BF16)** — 16-bit float with FP32-like exponent and shorter mantissa.

- **FP16** — 16-bit float often called half precision smaller range/precision than FP32.
- **FP8** — 8-bit float extreme compression, represents the exponent and mantissa in either 4,3 or 5,2 format plus sign.

Exactly which type to use depends on the needs of the application for accuracy, performance, and power efficiency.

6. AMD Versal™ Adaptive SoC Architecture Introduction

Fabricated on TMSC 7nm silicon, the AMD Versal™ adaptive SoC architecture fuses application-class processors, network on-chip (NoC), programmable logic, and domain-specific accelerators into a single platform. At the heart is the Control, Interfaces & Processing System (CIPS), which brings together Arm® applications and real-time processors, cache-coherent interconnect, platform management/boot, and AXI connectivity into the NoC and PL. This fusion enables a coherent, well-managed control plane for the entire device, while providing a tightly integrated high performance processing solution.

Connectivity within the device is provided by a high-bandwidth Network-on-Chip (NoC) which spans the device, making DDR memory and peripherals system-wide resources. To support different applications, determinism, and latencies, the NoC supports several QoS classes, low-latency, isochronous, and best-effort.

The Programmable Logic (PL) delivers next-generation CLBs, abundant BRAM/URAM, and DSP58 slices which support SIMD, complex math, and FP32 for custom data paths and tight I/O coupling. This makes the Versal adaptive SoC architecture ideal for packet handling, protocol adaptation, and DSP pipelines.

Many devices within the AMD Versal adaptive SoC range include AI Engines (AIE/AIE-ML). These AI engines are VLIW/SIMD vector processors with local memories and high-throughput streaming, well suited to signal processing, imaging, communications, and ML inference. They natively support common fixed-point and floating-point formats and integrate via the NoC and AXI, letting you partition algorithms between software-defined AI Engine graphs and PL kernels to hit performance, latency, and power targets.

Complementing AI Engines, the PL's DSP Engines provide deterministic MAC-dense building blocks for oversampling filters, channelization, modulation, and control-loop math—great for offloading pre/post-processing around AI Engine kernels or standing alone where hardware pipelines win on latency.

Integrated hardware peripherals round out the platform. This includes hardened DDR memory controllers accessed through the NoC, multi-rate Ethernet, PCIe/CPM DMA, high-speed GTs (32/58/112 Gb/s), XPHY for DDR/MIPI and source-synchronous interfaces, plus video decode, crypto, USB, SD/eMMC, and more—so most board-level data movement is solved with built-ins rather than soft IP.

Together, CIPS for control, NoC for predictable data movement, PL and DSP Engines for custom hardware, the AI Engine for software-defined vector compute, and rich integrated peripherals form a cohesive and composable platform. The remainder of this note will show how to map real DSP workloads onto the AI Engine and PL using AMD Vivado™ tools and Vitis™ software platform flow to achieve higher throughput and lower power than PL-only implementations.

7. AMD Versal™ Adaptive SoC Products

AMD Versal devices are available in a range of different series, while built on a common backbone: a hardened processing system connected with a Network-on-Chip, surrounded by programmable logic and rich I/O.

What changes from series-to-series is the application emphasis: some devices are optimized for compute and AL acceleration, some for embedded inference at low power, some for sheer I/O bandwidth, and others for compute with massive on-package memory. These different product series are Versal AI Core, Versal AI Edge, Versal Prime, Versal Premium, and Versal High-Bandwidth Memory adaptive SoCs.

Versal AI Edge devices are the natural choice when the problem lives beside sensors and cameras, and you care about milliwatts and nanoseconds. They integrate AI Engine-ML blocks and on-chip XRAM to enable inference and pre/post-processing to the edge. In practice, that means robotics, smart cameras, and industrial analytics pipelines where you want to keep data local, minimize DDR traffic, and still have enough PL fabric to massage pixels, fuse sensors, or run control loops without breaking the power budget.

Versal AI Core devices are optimized for compute and AI Acceleration. Here, the AI Engine array also includes support for commonly used DSP Types ideal for filters, FFTs, beamforming.

Versal Prime devices sit in the middle as the “do-everything” platform. This is intended for solutions where the developers need a broad mix of logic, memory, and I/O without AI acceleration. It is an ideal target for embedded control with pockets of acceleration, protocol bridging, and gateway-style designs that touch lots of interfaces.

Versal Premium devices are about moving data very fast, providing a very high lane counts, and the highest performance SerDes, coupled with hardened networking/crypto feature set. These devices are intended for packet processing, switching, and secure transport at scale.

Versal HBM devices solve the memory starvation problem that many adaptive compute solutions face. By bringing High-Bandwidth Memory onto the package and coupling it into the NoC and PL, they keep wide dataflows fed without the latency and contention of external DRAM. Workloads like large FFTs, table look-ups, packet buffers, and analytics that keep revisiting big datasets all benefit because the memory system stops being the bottleneck.

Alongside the different Versal adaptive SoC architecture series, some also have different generations that introduce new capabilities and performance.

8. AMD Versal™ Adaptive SoC Architecture Generations

The range of AMD Versal adaptive SoCs provide a number of different devices available across two generations. To identify the most appropriate device and generation for the challenge at hand, we must first understand the different generations and devices.

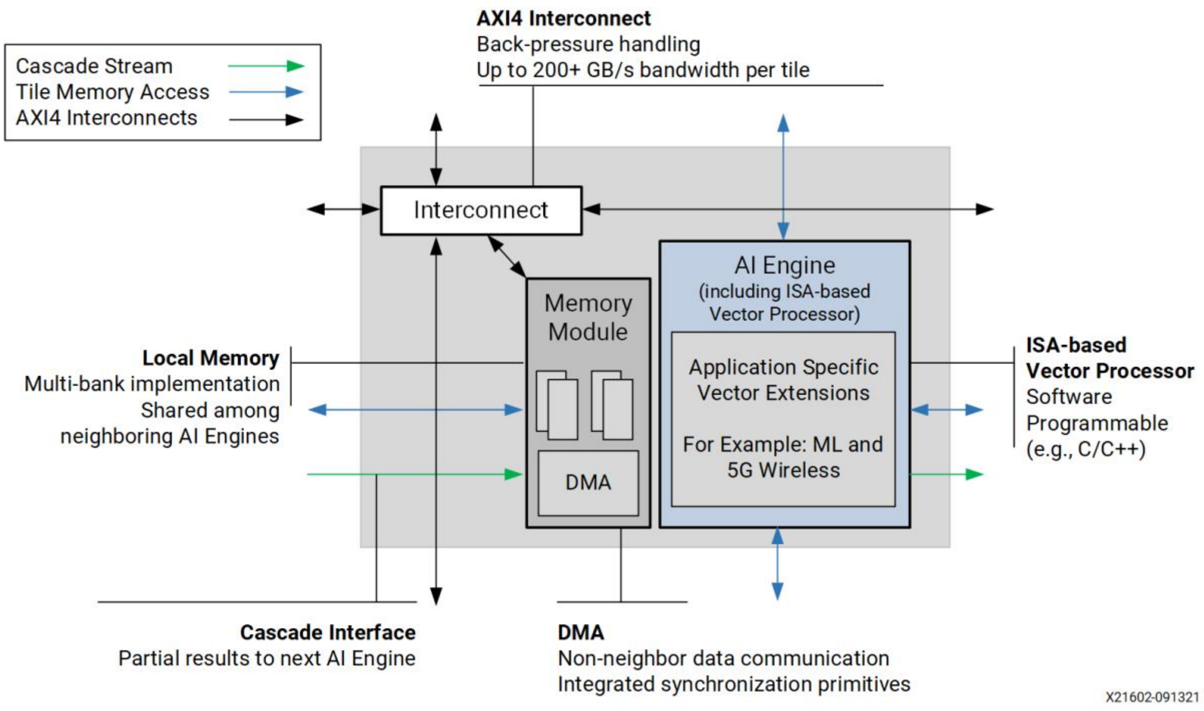
Generation 1.
The first generation of Versal devices brings together a heterogeneous compute fabric described above with the common core being the CIPS, PL, and NoC. Depending on the family, devices add AI Engines: either the original AI Engine or the first-generation AIE-ML Gen-1 spans multiple families, Versal AI Core, Versal AI Edge (Gen 1), Versal Premium, Versal Prime, and Versal HBM adaptive SoCs giving designers a consistent architecture with different mixes of acceleration and I/O.

Generation 2.
The second generation updates the processing system, moving to Arm® Cortex®-A78AE for applications and Arm Cortex-R52 for real-time control. Another major difference is the introduction of second-generation ML-centric AI Engines (AIE-ML v2). These engines provide higher inference efficiency and throughput at the edge.

9. Deep Dive on the AMD Versal™ Adaptive SoC AI Engine

At the heart of several Versal devices sits a two-dimensional array of AI Engine tiles. Each tile combines a VLIW/SIMD processor, tightly-coupled local memories, and on-tile interconnect for streaming, configuration, and debug. The array connects to the rest of the device via a dedicated AI Engine array interface into the NoC (memory-mapped) and directly into the PL (streaming).

Each AI Engine tile comprises three blocks: the AI Engine itself (scalar + vector data paths under a VLIW scheduler), the AI Engine memory module, and the tile interconnect. This forms the basic compute, memory, switch which is replicated across the two-dimensional array, to implement the desired functionality.

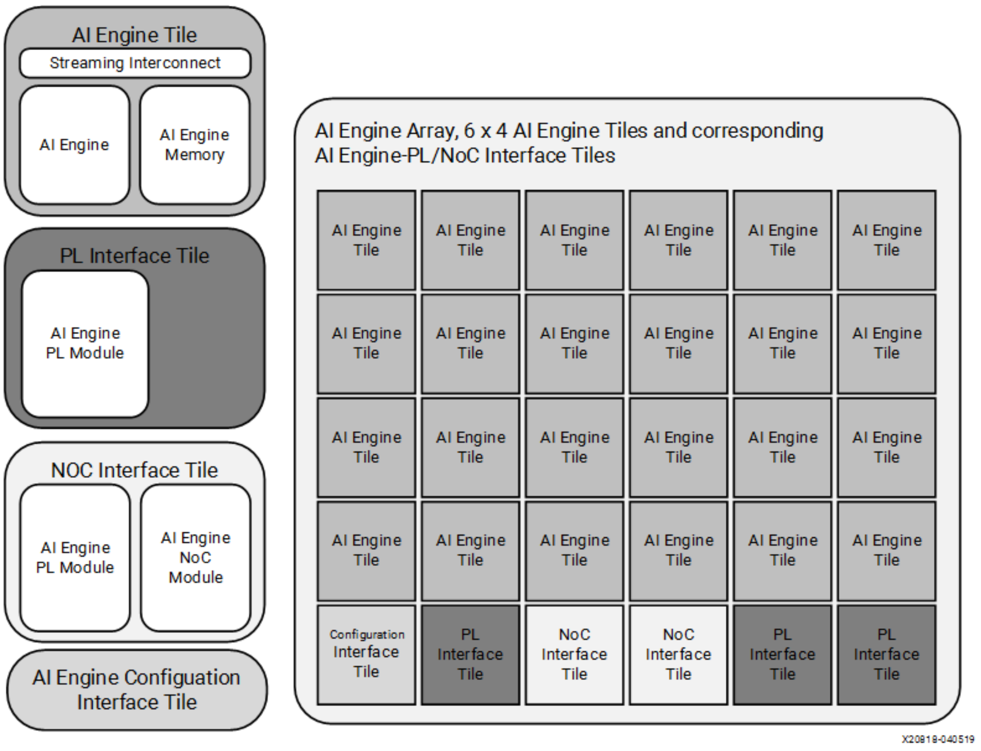

Cascade Stream
Tile Memory Access
AXI4 Interconnects
AXI4 Interconnect
Back-pressure handling
Up to 200+ GB/s bandwidth per tile
Interconnect
Memory Module
DMA
AI Engine
(including ISA-based Vector Processor)
Application Specific Vector Extensions
For Example: ML and 5G Wireless
Local Memory
Multi-bank implementation
Shared among neighboring AI Engines
ISA-based Vector Processor
Software Programmable (e.g., C/C++)
Cascade Interface
Partial results to next AI Engine
DMA
Non-neighbor data communication
Integrated synchronization primitives
X21602-091321

On-array connectivity

AI Engine tiles are able to link north/south/east/west through a lightweight stream switch for dataflow between neighbours; in parallel, a memory-mapped AXI network spans the array so external AXI masters via the NoC can reach tile memories and control/status registers. This dual-fabric streaming for dataflow, AXI for control/config is what makes application implementation practical.

Getting data in/out

To achieve the highest throughput possible by using the AI Engine tiles, we need to able to get data into and out of the AI Engine tile array efficiently. To achieve this, interface tiles are used at the boundary, which are capable of connecting to with the NoC, Programmable Logic, or Configuration Interface.



It is through the NoC interface tiles that we are able to access system memories such as DDR memory or Acceleration RAM (XRAM) within the CIPS.

Each AI Engine tile pairs its VLIW/SIMD core with banked local data memory for parallel access, a tile-level DMA to move data efficiently, and a 16-lock hardware semaphore unit for safe producer-consumer interfacing between tiles. This combination gives a deterministic data flow which can leverage back pressure to control producer-consumer flow.

All memories and registers in AI Engines and memory modules are AXI4 memory-mapped, so software on PS can boot, configure, and inspect the array. That same path enables profiling and trace control exposed by the tools.

10. AI Engine Generations

The Versal AI Engine is a tiled VLIW+SIMD processor array with local memories and stream/DMA fabric between tiles. It is designed for deterministic deeply-pipelined signal processing and inference graphs: you place kernels tile-by-tile, move data explicitly, and get predictable throughput.

AI Engines provide wide SIMD on classic fixed-point and full single-precision floating point. Vector lanes natively cover 8/16/32-bit integers (including complex forms) and real/complex FP32, and as such, AI Engines excel at numerically sensitive DSP (FIR, FFT, beamforming) as well as fixed-point ML primitives.

AIE-ML (first-generation ML engines) shifts the center of gravity toward inference. It adds ML-oriented data types most notably bfloat16 (BF16) and low-precision integer vectors down to INT4 to leverage more efficient quantized CNN/transformer blocks while still composing DSP-style graphs. Critically, AIE-ML removes the native FP32 vector pipeline present in AI Engines, and in its place, float support is provided via BF16-based emulation / accumulation in the API.

AIE-ML v2 (second-generation ML engines) builds on that with higher per-tile compute and better performance per Watt for inference. It expands native types beyond AIE-ML: FP16 and FP8 are supported in the vector unit, along with microscaling (MX) formats MX9, MX6, MX4 where lanes share an exponent for block-floating efficiency. There's dedicated block-floating matrix-multiply in the API that accumulates in FP32 (accfloat), making modern attention and Matrix Multiply (GEMM)-heavy models more efficient at the edge.

11. Programming Concepts for the AIE

To program an AI Engine, we need to be able to define not only the sequence which is executed by a particular AI Engine tile (the kernel), but also how the AI Engine tiles flow data between each other.

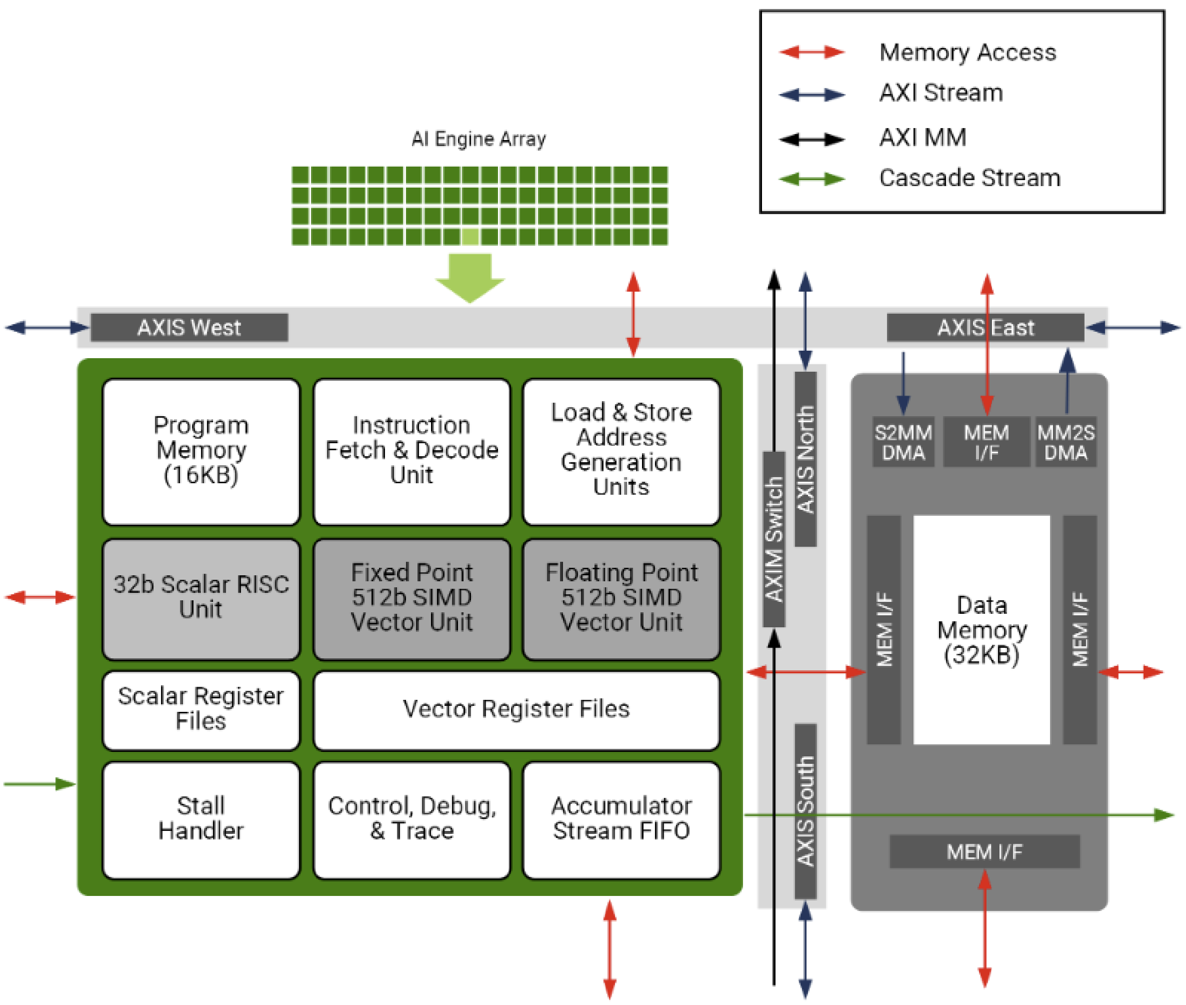
First we should understand when to use the AIE and when to use the PL. AIE suits vectorizable, compute-dense streaming DSP such as multi-rate FIRs, FFT/IFTT channelizers, beamforming/MIMO, linear algebra, and AIE-ML inference; these workloads map cleanly to SIMD and benefit from local tile memory.

The PL suits ultra-low-latency, bit-level or control-heavy logic and very high I/O rates such as packet parsing, protocol framing/de-framing, encoders/decoders, scramblers, custom interfaces, and wide fixed-point pipelines close to SERDES.

Before we discuss how the AIE Kernel is programmed, we will examine how the AIE Tiles are connected to stream data between each other to implement the desired overall functionality.

Connecting together AI Engine Tile kernels is called creating a graph, and is based on a distributed model of computing proposed in 1974 by Gilles Kahn, called a Khan Process Network (KPN). In the Khan Process Network, tasks within it are executed in parallel whenever possible. Like many concepts, the KPN is implementation independent, which also makes it ideal for implementation within a heterogeneous system.

Within a KPN, the components represent the kernel while the connections between the components represent the data flow. These are often called edges in KPN terminology. Mapping this back to the AI Engine, the components represent an AI Engine tile executing a kernel. Connections between the components is achieved by the AI Engine tile interfaces, typically the lightweight streaming protocol which connects the AI Engine tiles.



When working with Versal devices, KPN networks are referred to as data flow graphs. These graphs are captured using a C++ Adaptive Data Flow (ADF) graph program. Within this graph, the execution schedule is determined by the graph and the available of input data and of course output resources.

This means the AI Engine tiles within the design are in one of two states, executing or awaiting input data. Common concepts used in ADF graph programs are:

- Node – This is the AI Engine Kernel.
- Token – This is the input data to the AI Engine.
- Edge – Edges are implemented as AI Engine interfaces (e.g. IO Stream or DMA).
- Firing – Execution of the AI Engine Kernel this is managed by the AI Engine compiler based on input data availability.
- Blocking – Kernels are stalled if waiting for the buffers to be filled or the source kernel is not providing data.

The AI Engine Kernel is declared as C/C++ functions leveraging the AI Engine API. The resultant application is compiled by the AI Engine compiler to create ELF files for all of the kernels, along with partitioning, placement and routing defined by the ADF.

This is provided to the development environment as a compiled library libadf.a along with the necessary metadata. This compiled graph can then be integrated within the larger system design using AMD Vitis™ software platform.

12. Developing AI Engine Applications

There are two practical routes to program the Versal AI Engine, and they are deliberately complementary.

In the Vitis software platform, the AI Engine C++ ADF Graph serves as the description of the pipeline. Kernels are written in standard C/C++, and the graph defines how samples move either as framed windows for deterministic staging or as continuous streams while making interfaces explicit.

PLIO connects the graph to the AXI4-Stream in the programmable logic. GMIO moves data to and from external memory through the NoC, and cascade links enable multi-tile accumulations for constructs such as long FIRs and beamformers. When you build, the AI Engine compiler elaborates the graph, places and routes kernels, configures tile DMAs and locks, and emits the graph container (libadf.a with metadata).

The Vitis software platform system linker (v++ --link) then packages that container, any PL kernels, and the platform into a single device image (PDI).

Verification is done in two passes: software emulation for very fast functional checks using file I/O, followed by aiesim for tile-accurate timing and back-pressure behavior. The Vitis software platform analyzer closes the loop by exposing stalls and utilisation, so buffer sizes, window lengths, and NoC choices can be optimized for the application. At run time, the processing system controls the graph through XRT/ADF including start/stop, runtime parameter updates, event profiling, which enables us to fine tune behavior without a rebuild.

The alternative approach is to use Vitis Model Composer integrates the same technology into MathWorks MATLAB® and Simulink® for teams who begin with algorithms and golden vectors.

The signal chain is drawn with AI Engine blocks and co-simulated against existing MATLAB testbenches, which keeps verification assets intact while you settle questions of frame size, fixed-versus floating-point, and quantization.

With a single export, Vitis Model Composer generates the identical ADF graph and kernel sources and hands them to the Vitis software platform.

Following the export from Vitis Model Composer, the flow is unchanged. The aiecompiler builds the graph, v++ links the system, the same simulators validate timing, and the same runtime APIs control deployment.

Both routes can take a design from concept-to-device. The choice often depends upon the developing teams experience. The Vitis software platform suits software-oriented teams comfortable with C/C++, version control, and CI/CD, offering explicit control and clean system integration. Model Composer suits algorithm-focused teams working in MATLAB/Simulink, accelerating iteration with block-diagram modelling and code generation. As each path leverages the same linking and analysis flow, both approaches enable high performance solutions leveraging the AI Engine.

13. Why use the AI Engine for DSP-based Applications

Traditionally we have implemented DSP processing within programmable logic, leveraging the capabilities provided by DSP, Block RAM and CLB elements. While this enables low-latency and deterministic implementations it may not provide the most optimal solution for performance and performance / MHz / W.

The Versal AI Engine provides developers with a VILW SMID specialized vector processor, which has been optimized to implement the mathematics commonly used within AI and DSP algorithms. Coupled with the closely coupled memory and streaming / DMA interconnects, this enables the construction of DSP chains implementing filters, FFTs, and beamformers to be composed tile-by-tile with bounded latency and fewer external “glue” resources than would be implemented when a discrete PL solution was implemented.

In contrast, DSP58 slices are powerful arithmetic blocks that provide the ability for SIMD, complex math, FP32 within the PL fabric, but they rely on separately provisioned PL memories (BRAM/URAM) and control to form long pipelines. This can add routing and buffering overheads at scale. This programmable logic implementation performance is dependent on the implementation style selected by the developer.

Leveraging the AI Engine therefore has several advantages over the use of PL and discrete logic implementation using IP blocks. When working with a PL implementation, the engineering must start from scratch and determine the data types, quantization, implement and verify the design using an HDL simulator.

Based around KPN, the AI Engine tile array enables a simple graph to be created for the desired DSP algorithm to be implemented. The closely coupled memory provides for fast storage of the operands while the streaming interface allows multiple AI Engine kernels to be connected to together without the need to achieve timing closure.

When it comes to developing DSP applications, AI Engine tiles also align with the data types commonly used in DSP applications (e.g. Int8/Int16, CInt16/CInt32, and FP32). The AIE-ML extends this with Int4 and BFloat16 for bandwidth and energy efficient processing.

The on-device NoC provides quality-of-service classes, Low-Latency and Isochronous among them to move streams predictably between PS, PL, DDRMC, and the AI Engine array. The NoC therefore acts as enabler for high-rate, multi-stage DSP graphs. Finally, XRAM (Acceleration RAM) offers a low-latency staging buffer for AIE/AIE-ML and the RPU, giving designs a convenient scratch space for ping-pong buffers and burst absorption.

The AI Engine and its DSP capabilities therefore provide the developer with a wider range of tools with which to address the challenges presented for performance and power efficiency.

Aspect	AI Engine Tile (AIE / AIE-ML)	DSP58 in PL
Compute model	VLIW core with SIMD units per tile; designed for pipelined DSP/AI kernels.	Fixed-function arithmetic slice with SIMD/complex/FP32 modes.
Local memory & DMA	Shared local SRAM on each tile with integrated interconnect/DMA for windows/streams—minimizes external buffering.	No embedded scratchpad or DMA; long chains typically add BRAM/URAM/FIFOs and control in PL.
System fabric	NoC with QoS (Low-Latency, Isochronous, Best-Effort) for predictable movement between PS/PL/DDR/AIE.	Reaches memory/IO via PL interconnect; QoS guarantees come from design discipline rather than fabric classes.
Staging buffers	XRAM (4 MB) provides low-latency staging for AIE/AIE-ML and RPU.	Uses PL memories (BRAM/URAM) as designed by the user.
Native data types	Int8/Int16, CInt16/CInt32, FP32; AIE-ML adds Int4, BFloat16 for low-precision, high-throughput inference/DSP blending.	Supports fixed/floating operations inside the slice, including FP32 and complex modes.
Pipeline scalability	Tile-by-tile composition with deterministic behavior and fewer external inter-stage resources.	Scales by replicating slices and adding fabric for buffering/scheduling; routing pressure grows with stage count.
Throughput/latency QoS	Fabric QoS classes enable bounded latency paths (Low-Latency/Isochronous) for streams.	Depends on user-implemented interconnect and arbitration policies.
Tool flow	Vitis software platform ADF (and optionally Model Composer) for software-driven graph design, profiling, and deployment.	Vivado tool-centric RTL/block design; DSP58 used as IP/macro with user-built control/data path.
Time-to-performance & power	Typically shorter bring-up; dynamic power often reduced when bulk DSP moves into the AI Engine (software-developed kernels).	Powerful for custom data paths and unique bit-level tricks; power/effort scale with the amount of surrounding fabric.
Best-fit use cases	Multi-stage filtering, FFTs/OFDM, beamforming, modem chains, and mixed DSP/AI pipelines with tight latency budgets.	Fine-grained arithmetic inside bespoke PL architectures; glue logic around the AI Engine; interface adapters; specialized kernels.

14. Example AI Engine for DSP Applications

Across beam forming, EW, satcom, robotics, test gear, and medical imaging, Versal AI Engines (AIE) turn heavy streaming DSP kernels into tile-by-tile, deterministic pipelines.

Automotive
Automotive LiDAR pushes multi-gigabit sensor streams that reward a clean split: keep the sensor PHY (MIPI/SLVS-EC), timestamping, and safety monitors in the PL, and let AIE handle the heavy math. Tiles run windowing and de-chirp, batched 1D/2D FFTs for range, Doppler, multi-return accumulation, and phase-based beamforming across the receiver array. Downstream stages, CFAR/peak picking, motion compensation, and fast clustering/voxel filtering, streaming tile-to-tile to produces a clean, latency-bounded point cloud. The result is a deterministic frame timing and higher throughput per watt than a PL-only build, with the PS free to fuse LiDAR, radar, and camera for tracking and SLAM

Beam Forming
Phased-array beam forming is a natural fit for AI Engines because the core operations fractional-delay FIRs, complex phase rotation, and weighted summation across elements map directly to vector MACs with data kept close in on-tile memory. AI Engine tiles hold per-beam weight vectors and slide sample windows via tile DMA, passing partial sums through cascade/stream ports so the array behaves like a deterministic, deeply pipelined summation tree. For direction-of-arrival heads such as MUSIC or ESPRIT, covariance accumulation (outer products) and narrowband FFTs are staged tile-by-tile, while PL logic handles high-rate ADC interfacing, decimation, and calibration injection without disturbing the compute graph.

Electronic Warfare
EW receivers benefit from wideband channelization and fast retuning under load. AI Engines deliver both by distributing polyphase filter banks and large FFTs across tiles, using hardware locks to coordinate producers and consumers without software spin loops. Pulse compression and detection chains FFT-domain convolution, cross-correlation, and CFAR-style statistics stream deterministically from one stage to the next, letting threat libraries or matched-filter coefficients be swapped at runtime via scalar control. Instantaneous frequency and phase estimation (short-window STFTs, vectorized arctan/unwrap) exploit SIMD lanes for high sample-rate I/Q math, while PL implements gating/blanking and exotic RF digitizer links.

Military / Satellite Communications
Baseband workloads such as pulse shaping (RRC), resampling, carrier/timing loops, MIMO equalization, and OFDM/iFFT pipelines can be arranged as a tiled AI Engine graph to guarantee frame-to-frame throughput and latency. In beam-hopped or phased-array systems, per-beam precoding and combining become matrix–vector streams, each tile holds a sub-matrix and applies updates deterministically each frame, enabling rapid beam schedule changes without rebuilding the PL. Symbol demapping and soft-information generation are vector reductions that sit well on the AI Engine, with the NoC moving traffic to DDR/XRAM or handing off to PL-based FEC engines and framers at line rate.

Robotics
Real-time autonomy blends perception with control, so deterministic latency matters as much as raw TOPS. AIE tiles run multi-rate filtering (FIR/IIR), quaternion and rotation math, and EKF/Gaussian update steps as short, predictable kernels that meet control deadlines without thrashing external memory. Vision and ranging pre-processing, pyramid filters, block-matching, local FFTs for feature

extraction are streamed through the array to minimize DRAM bandwidth, while the PL handles image/LiDAR/IMU PHYs and timestamping. Audio/ultrasonic pipelines (beamforming, envelope detection, matched filtering) likewise chain tile-to-tile, giving consistent end-to-end latency for sensing and actuation loops.

Test Equipment (e.g., 5G Test Radios)
Test sets need flexible waveforms, clean spectra, and accurate measurements, often at high bandwidths. AI Engine graphs can build and analyze multi-carrier OFDM (resource-grid mapping → iFFT → clipping/CFR → DUC) and execute spectrum/cyclostationary analysis and EVM pipelines with deterministic frame timing. Digital predistortion and channel emulation, tap-heavy FIRs, memory polynomials, and fast-convolution blocks, partition naturally across tiles, letting you sweep models or impairments through scalar control without touching the PL. Framing and high-speed interfaces (JESD/CPRI/eCPRI) stay in PL, while the AI Engine delivers the baseband math at throughput-per-watt levels that keep thermals and power in check.

Medical Imaging (CT/MRI Reconstruction)
Reconstruction algorithms are dominated by streaming linear algebra and FFTs, which the AI Engine array handles as locality-friendly stages. CT pipelines perform 1D/2D/3D FFTs, filtering, and filtered back-projection as streaming reductions, distributing angles/slices across tiles to meet real-time targets. MRI pipelines benefit from gridding/NUFFT and complex multiply-accumulate steps arranged tile-by-tile; parallel MRI methods (SENSE/GRAPPA) use coil-wise complex ops and small dense linear algebra blocks that are SIMD-friendly. By keeping working sets in on-tile SRAM and moving data via tile DMA and stream ports, the AI Engine reduces DDR churn and delivers stable, repeatable latency, key for clinical throughput and image quality.

Conclusion

The AMD Versal™ AI Engine (AIE) family delivers excellent DSP performance with software-level flexibility, achieving higher throughput, lower power, and reduced PL resource use compared to traditional logic-based designs. With the unified Vivado Vitis flow, engineers can rapidly prototype, optimize, and deploy advanced DSP algorithms across Versal devices, accelerating innovation while cutting development time.