AI Engine Programming: A Kahn Process Network Evolution

WP552 (v1.0) July 20, 2023

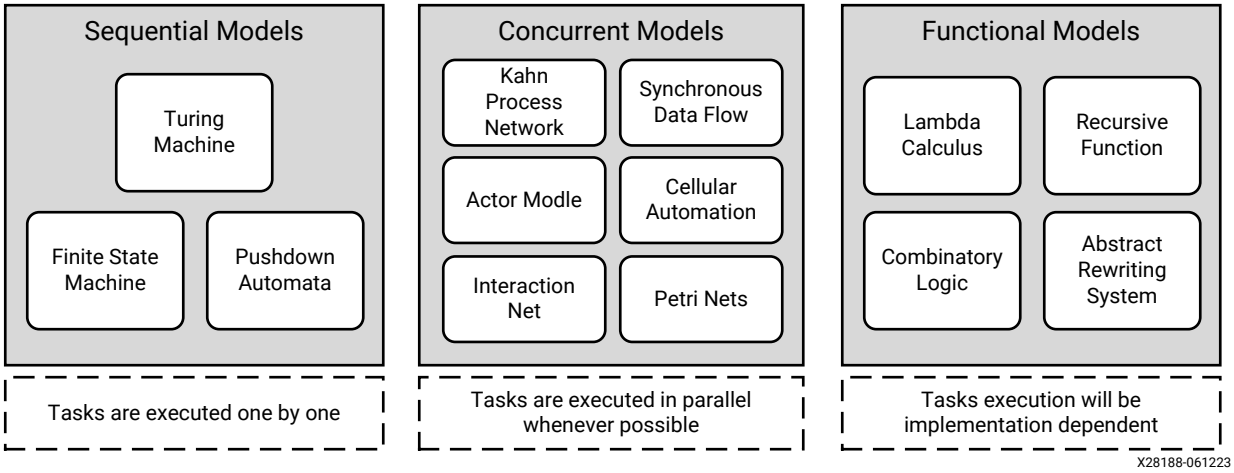
Abstract

This white paper explores how the AI Engine graph programming model is defined based on the Kahn process network (KPN). The KPN model helps to make the data flow parallel, which improves the overall performance of the system. Programming the AI Engine array requires a thorough understanding of the algorithm to be implemented, the capabilities of the AI Engines, and the overall data flow between individual functional units. AI Engine kernels are functions that run on an AI Engine and form the fundamental building blocks of a data flow graph specification. The data flow graph is a KPN with deterministic behavior. This white paper also includes an example design to illustrate a data flow graph with four AI Engine kernels that form the fundamental building blocks of a data flow graph specification. This example also demonstrates a data flow stall in the design and provides a solution.

Introduction

The KPN is widely used as a distributed programming model to run tasks in parallel whenever possible. This white paper describes how the AI Engine uses the KPN model for graph programming. There are various types of computation models based on the target architecture such as central processing unit (CPU), graphics processing unit (GPU), FPGA, and AI Engine programming. The following figure shows the models of computation classified as sequential, concurrent, and functional models.

Figure 1: **Models of Computation**

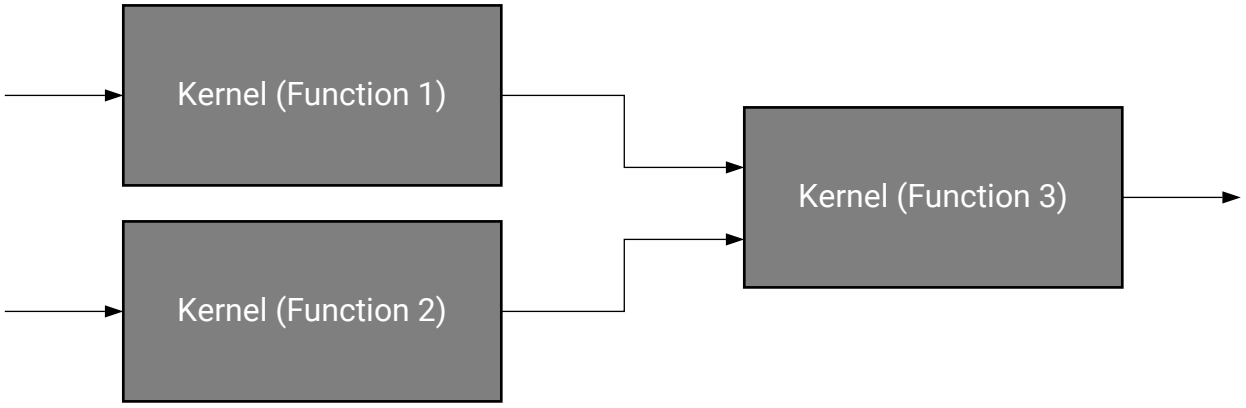


In sequential models, tasks are executed one after another or sequentially. In concurrent models, tasks are executed in parallel whenever possible. In functional models, tasks are implementation dependent, such as targeting a specific architecture, such as a GPU or the programmable logic in FPGAs. The focus of this white paper is the computation model of AI Engine programming. This model can be used to guide the programmer when writing the program that targets the AI Engine architecture. The aim is to fully leverage the computing power of the AI Engine by understanding its programming model. As the complexity of computational tasks have become more challenging, the standard processor has proven insufficient in performing these tasks efficiently. In response, various computational architectures have evolved to address this shortcoming such as CPUs, GPUs, application specific processors, etc.

Kahn Process Network

The KPN is a distributed model of computation proposed by Gilles Kahn in 1974 as a general-purpose scheme developed for parallel programming, which laid a foundation for the data flow model. In the KPN, the components represent the functions (or kernels) and the connections represent the data flow as shown in the following figure.

Figure 2: **Functions and Data Flow**



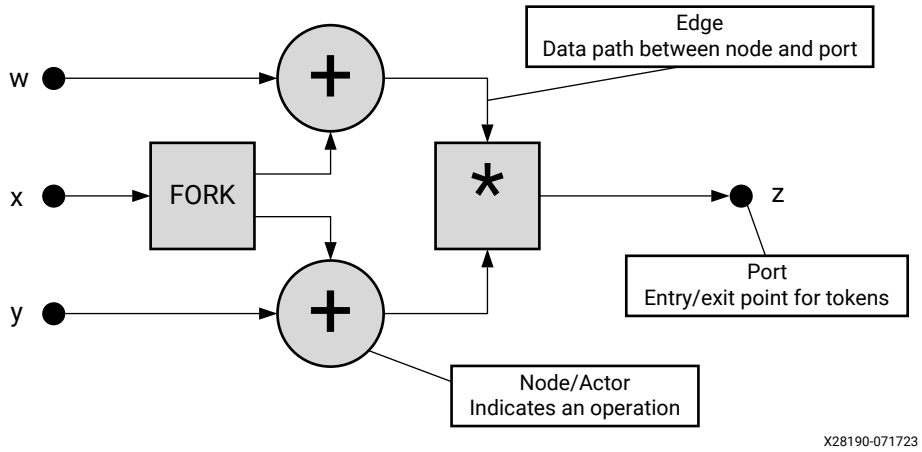
The kernel (function 3) reads data from two kernels (function 1 and 2). If there are no data available at any one of the kernels, the read stalls the process, which blocks the kernel (function 3). The process can continue only when sufficient data (tokens) are available. Writing the data (function 1 and 2) to the process (function 3) is non-blocking, which means it is always successful while writing and a stall does not occur. Due to these characteristics, the data flow network is deterministic in nature. This is a deterministic process communication through first in and first out (FIFO) channels. This model is proven for modeling embedded systems, signal processing systems, high-performance computing, data flow programming languages, and other computational tasks.

The signal processing systems are modeled using the KPN where infinite streams of data are processed by executing in sequential or parallel forms based on the given tasks.

Data Flow Graph

The data flow graph indicates the processing sequence (or precedence), parallelism, and data dependence. The following figure shows a representation of a data flow graph that has inputs that are processed to generate outputs. The three inputs are w, x, and y. In the figure, the inputs w and y are going to separate nodes.

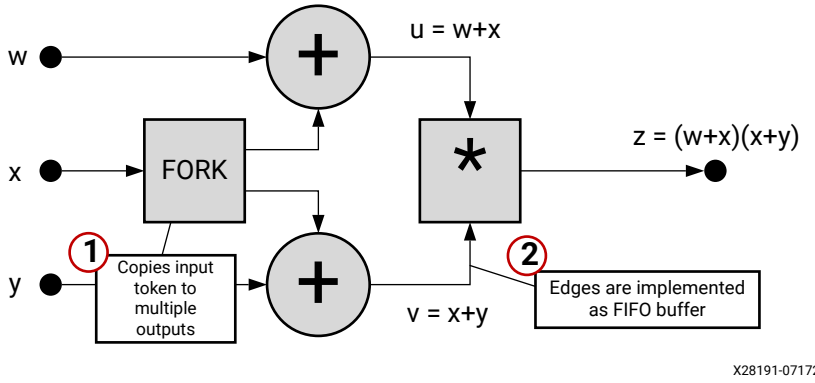
Figure 3: **Data Flow Graph Representation**



Edges indicate the path that the data takes to or from *actors* or *ports*. Ports indicate the points at which tokens enter or leave actors or the graph. The data flow graph shows the processing sequence, parallelism, and data dependence.

In the following figure, the fork replicates the input token across multiple outputs. The output of the nodes (edges) is implemented as FIFO buffers.

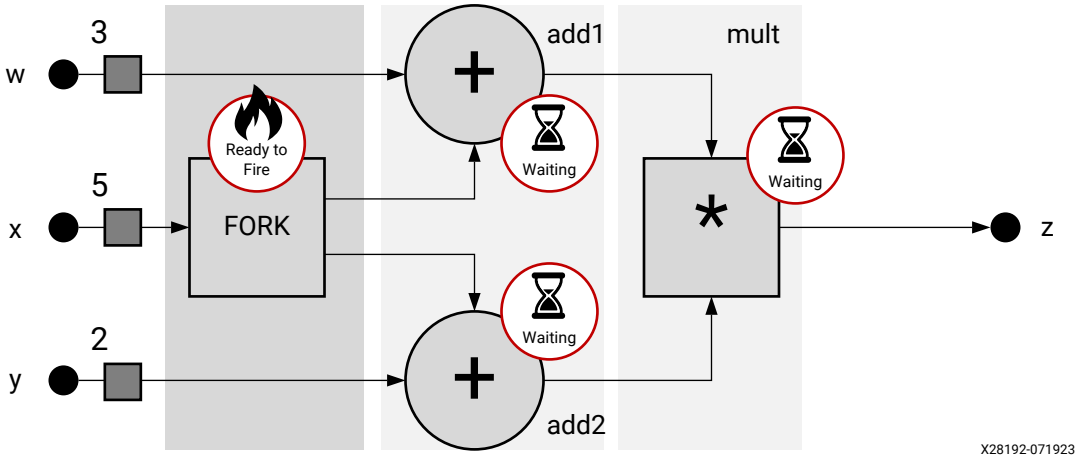
Figure 4: **Fork Replicates the Input Token Across Multiple Outputs**



Nodes (or actors) fire only when a single token is present on every input to the node. If at least one input is missing a token, the node is blocked. Each token is removed from the edge of each input after firing the node.

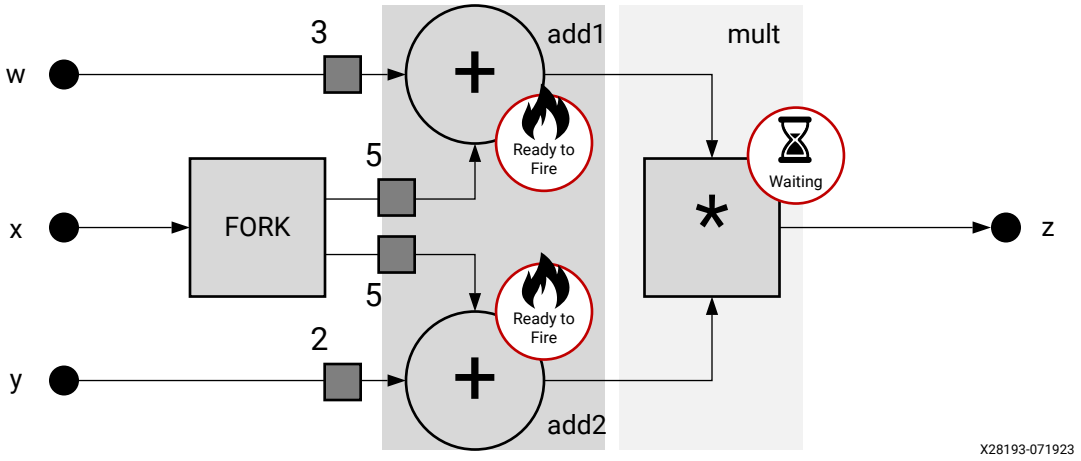
In the following figure, the values for w, x, and y are 3, 5, and 2, respectively. The x value is passed to the fork. The fork is ready to fire, while the nodes add1, add2, and mult are waiting for all inputs.

Figure 5: **Input Tokens are Available for Fork (Node/Actor) - Ready to Fire**



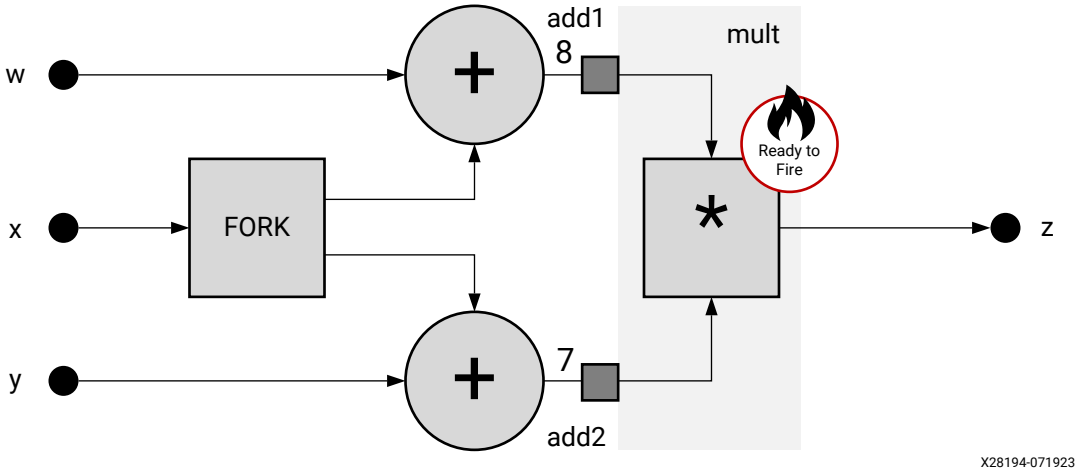
In the following figure, the x value 5 has been copied to nodes add1 and add2. The w value 3 and y value 2 are passed to nodes add1 and add2, respectively, from the input nodes. The node mult waits for its input tokens until add1 and add2 are both ready to fire and provide them. The output of node add1 is 8 (that is, 3 + 5). The output of node add2 is 7 (that is, 2 + 5). These output edges are implemented as FIFOs.

Figure 6: **Input Tokens are Available for Add1 and Add2 (Node/Actor) - Ready to Fire**



In the following figure, the output tokens from add1 and add2 arrive at node mult, which then performs the multiplication and provides the output as 56 (that is, 8×7). This output then goes to port z.

Figure 7: **Input Tokens are Available for Mult (Node/Actor) - Ready to Fire**

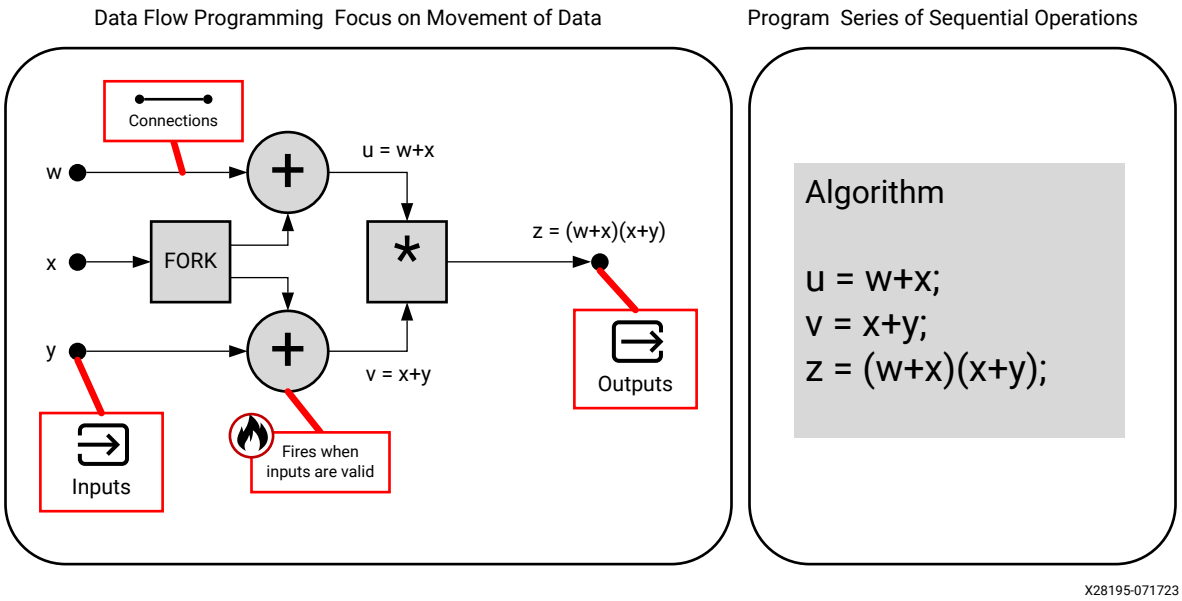


Data Flow Programming

This section describes how the data flow graph relates to data flow programming. Usually, a program is modeled as a series of sequential operations.

Data flow programming emphasizes the movement of data, which includes a series of connections, explicitly defined edges that connect nodes. The node fires as soon as all the input tokens are valid. Contrast this view to the algorithm representation shown on the right in the following figure.

Figure 8: **Data Flow Programming**



Although both the data flow graph on the left and the algorithmic form on the right represent the same function, the data flow graph illustrates the parallelism explicitly, while the algorithm form on the right hides the parallelism and appears sequential. For example, u and v can be computed in parallel provided inputs w, x, and y are all available. This is clear from the data flow graph but less so from the algorithm form.

In the figure in the Kahn Process Network section, assume function 1 is faster compared to function 2. Even though function 1 is faster compared to function 2, function 3 must wait for the input from both functions. This means there is no improvement to the performance overall by having a data flow model for these functions. It is important to accelerate the complete system rather than an individual function to achieve the performance improvement as a whole system. Data flow languages are inherently parallel, which works well in large decentralized systems.

AI Engine – Adaptive Data Flow Programming

This section describes how data flow programming works for the AI Engine. Nodes (or actors) indicate some type of operation. Nodes or kernels are implemented in AI Engines, which perform the operations, but not strictly as a single operator as shown in Figure 8: Data Flow Programming. An AI Engine can contain multiple kernels that can perform several operations.

KPN edges indicate the path that data takes to or from actors or ports. Edges are implemented as I/O streams, cascade I/O streams, streams or direct memory access (DMA) FIFOs, and local tile memory buffers in the AI Engine tile architecture.

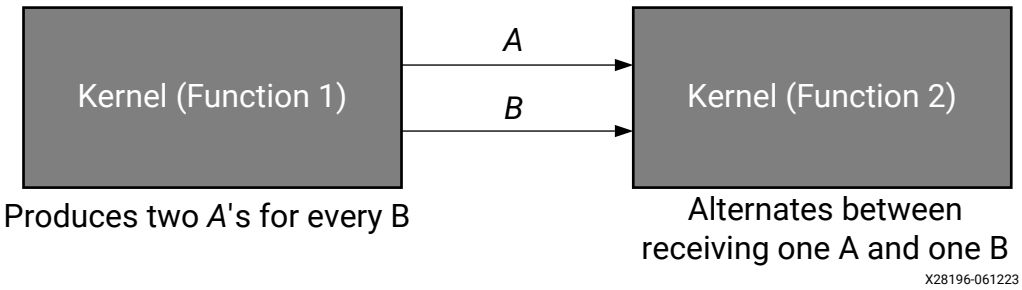
The connection between the KPN nodes (AI Engine kernels) in an AI Engine design are made through the C++ adaptive data flow (ADF) graph program. This code establishes the data flow graph wiring between KPN nodes (AI Engine kernels), identifies any large memory buffers required for those nodes and any I/Os to the graph.

The execution schedule is determined by the graph and the availability of input data and output resources:

- There is no instruction pointer to firing the AI Engines. Each tile fires and executes its kernel function once all input data is available, like in a KPN.
- There are many execution units available—10s to 100s of AI Engines based on the device. Some, none, or all of these engines might execute in parallel depending on the nature of the data flow graph that interconnects them together.
- All AI Engines are either computing or waiting for their input data, like in a KPN.
    - The AI Engine compiler takes inputs (data flow graph and kernels) and produces executable applications for running on an AI Engine device. The AI Engine compiler allocates the necessary resources such as locks, memory buffers, and DMA channels and descriptors, and generates routing information for mapping the graph onto the AI Engine array. It synthesizes a main program for each core that schedules all the kernels on the cores and implements the necessary locking mechanism and data copy among buffers.

In the following figure, function 1 generates two A's for every B. On average, function 2 consumes twice as many A's than B's. It might not always be A and B. It can be A for some time and B for another time. To handle this scenario, the data/token need to be accumulated to process later. In some cases, if this accumulation is for longer cycles, this might stall the system and affect performance. Based on the design requirements, the difficulties can vary. A few ways to overcome some of these challenges are by adding FIFOs to accumulate the data, program the kernel in such a way to improve the performance by using the multiple AI Engines, and other optimization techniques. It is important to understand the deadlock problem and use the proper techniques to solve it.

Figure 9: **Data Need to Be Accumulated**



The following table lists the comparisons between the KPN and AI Engine terminologies.

Table 1: **KPN and AI Engine Terminology**

Terminology	KPN	AI Engine
Node/actor	Represents the processes (functions).	AI Engine kernel: the processes (node/actor) are implemented as kernels in the AI Engine.
Tokens/inputs	Input data to the node/actor.	Input data to the AI Engine kernel.
Edge	Edges indicate the path that the data takes to or from actors or ports. The output of the nodes (edges) is implemented as FIFO buffers.	Edges are implemented as I/O streams, cascade I/O streams, streams or DMA FIFOs, and local tile memory buffers in the AI Engine tile architecture.
Firing	Nodes (or actors) fire only when a single token is present on every input to the node.	The AI Engine compiler manages the firing based on the availability of the input token (input window size) and the availability of the buffers.

Table 1: **KPN and AI Engine Terminology** *(cont'd)*

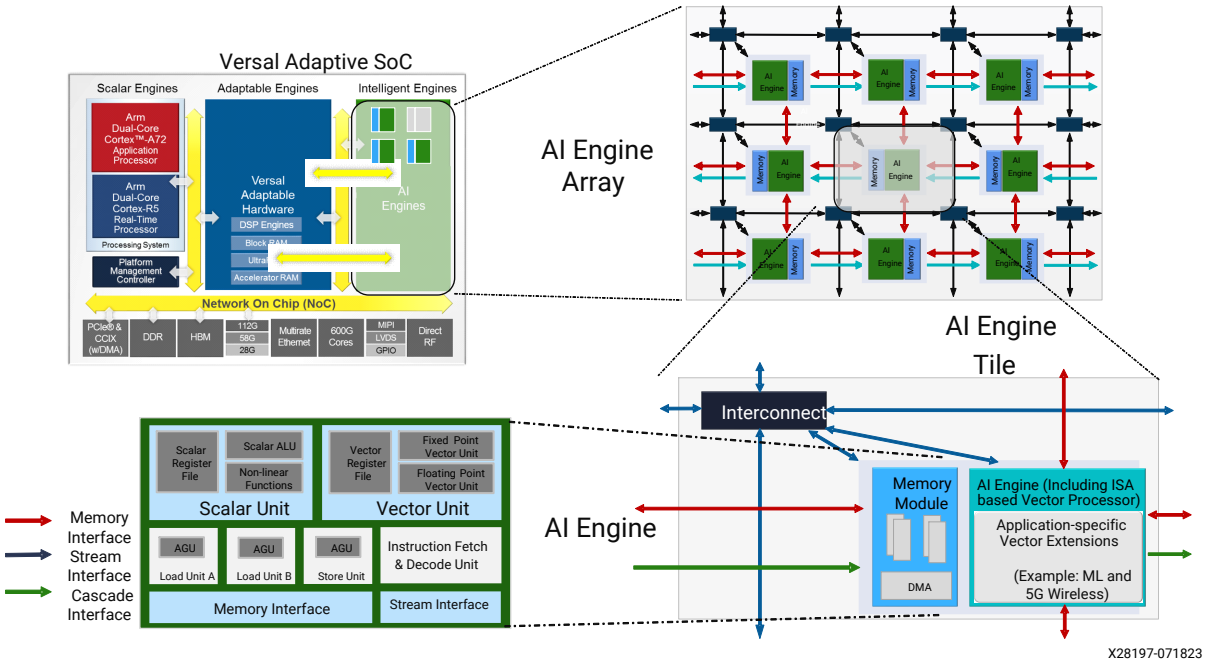
Terminology	KPN	AI Engine
Blocking	Reading is blocked if a node/actor (process) tries to read from an empty input.	For memory communication, kernels are stalled if the kernel is waiting for the buffer to be filled. With stream or cascade communication, the sink kernel can stall if the source is not producing the samples. This is taken care of by the AI Engine compiler. Locks: The AI Engine compiler allocates the necessary locks, memory buffers, and DMA channels and descriptors, and generates routing information for mapping the graph onto the AI Engine array. It synthesizes a main program for each core that schedules all the kernels on the cores and implements the necessary locking mechanism and data copy among buffers. The C program for each core is compiled using the Synopsys Single Core Compiler to produce loadable ELF files. The buffer structure is responsible for managing buffer locks tracking buffer type (ping/pong). The input and output buffers for the AI Engine kernel are ensured to be ready by the locks associated with the buffers.

In some scenarios, the data flow programming can be challenging for certain algorithms because scheduling can lead to stalling the process.

AI Engine Architecture

AI Engines provide multiple levels of parallelism including instruction-level and data-level parallelism. Instruction-level parallelism includes a scalar operation, up to two moves, two vector reads (loads), one vector write (store), and one vector instruction that can be executed. In total, a 7-way very long instruction word (VLIW) instruction per clock cycle. Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis. Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, and access to local memory in any of three neighboring directions. It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines, the programmable logic (PL), or the DMA.

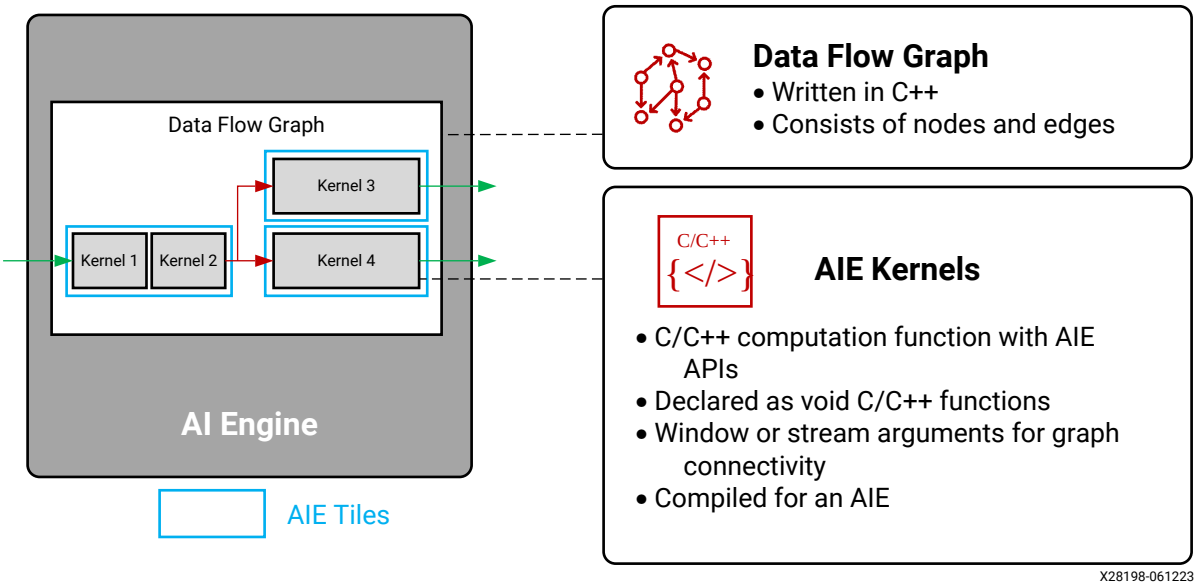
Figure 10: **Overview of AI Engine Architecture**



AI Engine Kernel and Graph Programming

The kernels/functions needed to accelerate are programmed using C++ with AI Engine APIs that target the VLIW scalar and vector processors of the AI Engine. The AI Engine kernel code is compiled using the AI Engine compiler (aiecompiler) that is included in the AMD Vitis™ development kit. The AI Engine compiler compiles the kernels to produce executable and linkable format (ELF) files that are run on the AI Engine processors. The kernels can be mapped to the independent AI Engines based on the algorithm to perform the parallel computation.

Figure 11: **Graph and Kernel Programming**



The data flow concept should be applied to the design specification to obtain the benefit of parallel execution whenever possible. It is also important to make sure the AI Engine kernels are running faster to obtain the benefit of the data flow programming. See Data Flow Programming.

AI Engine Design Example

This example design uses four AI Engine kernels:

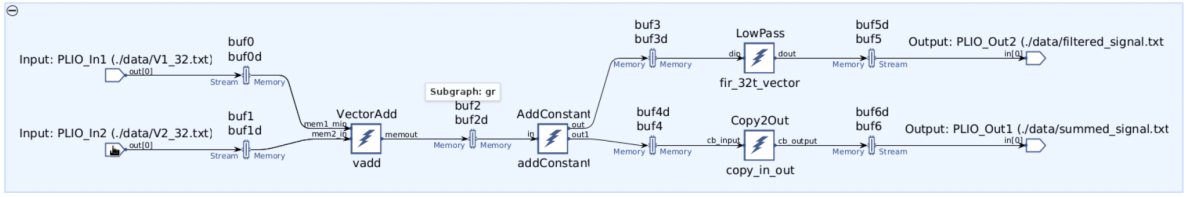
- Vector addition (vadd)
- Add constant value (addConstant)
- Copy the add constant to the output (copy_in_out)
- FIR filter (fir_32)

This example uses two versions of the FIR filter (fir_32), one scalar code, and another vector code. This helps to showcase the importance of the kernel performance and the benefit of using the data flow programming in the design (parallel computation).

Figure 12: **AI Engine Kernels**

The following figure shows the actual implementation of the AI Engine ADF in the Vitis analyzer tool. In this figure, the boxes with the lightning symbols represent the kernels. The buffers are represented as bufx and bufxd (ping and pong buffers), for example, buf0 (ping) and buf0d (pong). The input and output ports are represented as PLIO_In1, PLIO_In2, PLIO_Out1, and PLIO_Out2, respectively.

Figure 13: **Graph View in Vitis Analyzer Tool**



The next sections illustrate how the design is executed based on the KPN concept.

Frame 1

The first set of data started writing to the buffers (ping buffer – buf0/buf1) and the vadd kernel (node/actor) is waiting for the input token to be ready.

Figure 14: **Frame 1**

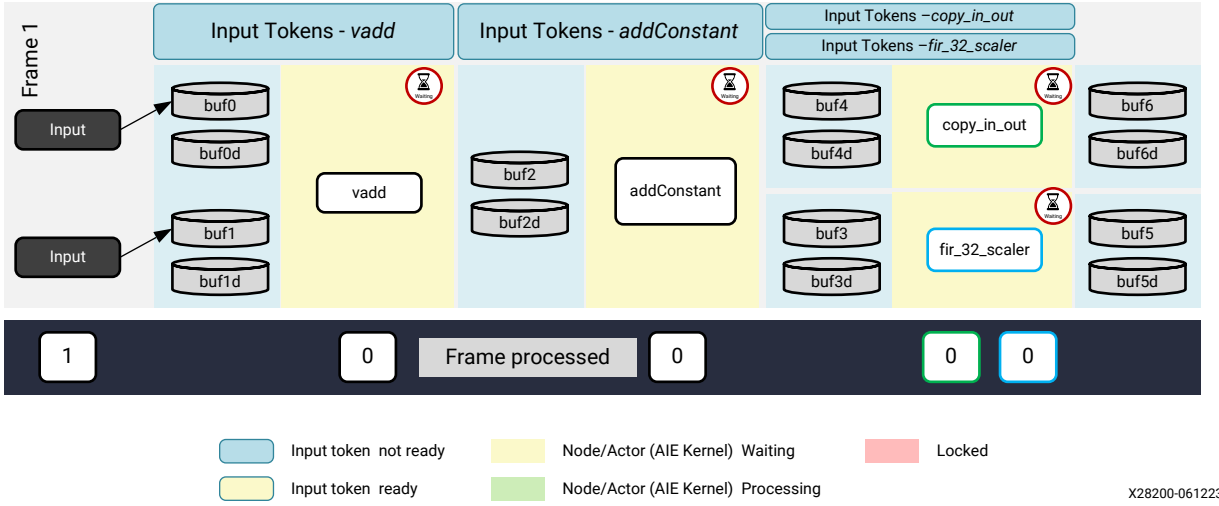


Table 2: **Sending the Frame 1 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-

Frame 2

The second set of data started writing to the buffers (pong buffer – buf0d/buf1d) and the vadd kernel (node/actor) started processing frame 1 as the input token is ready.

Figure 15: **Frame 2**

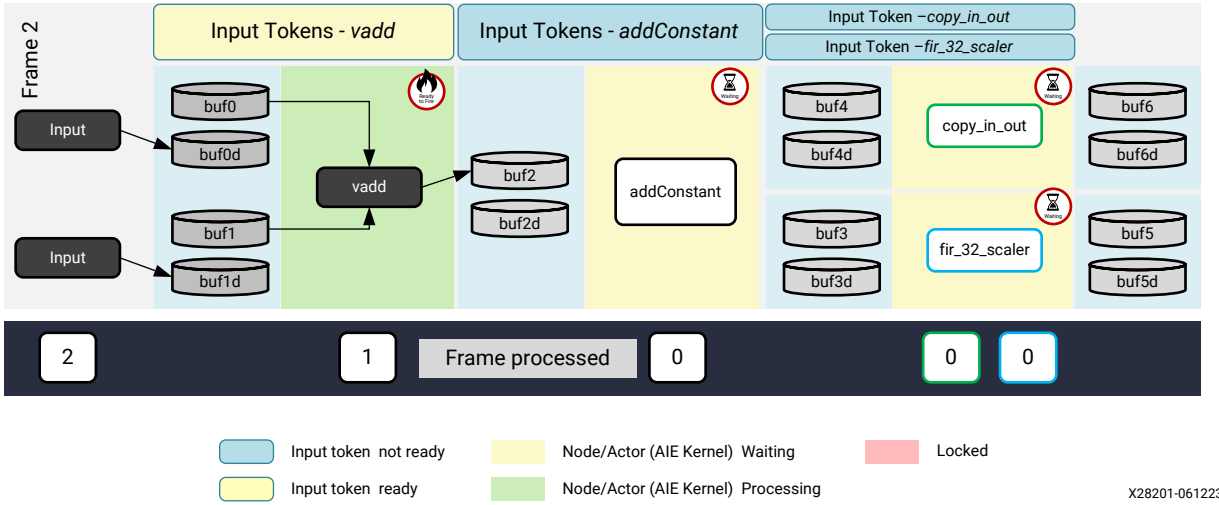


Table 3: **Sending the Frame 2 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-

Frame 3

The third set of data started writing to the buffers (ping buffer – buf0/buf1) and the vadd kernel (node/actor) started processing frame 2. The addConstant kernel (node/actor) started processing frame 1 as the input token is ready.

Figure 16: **Frame 3**

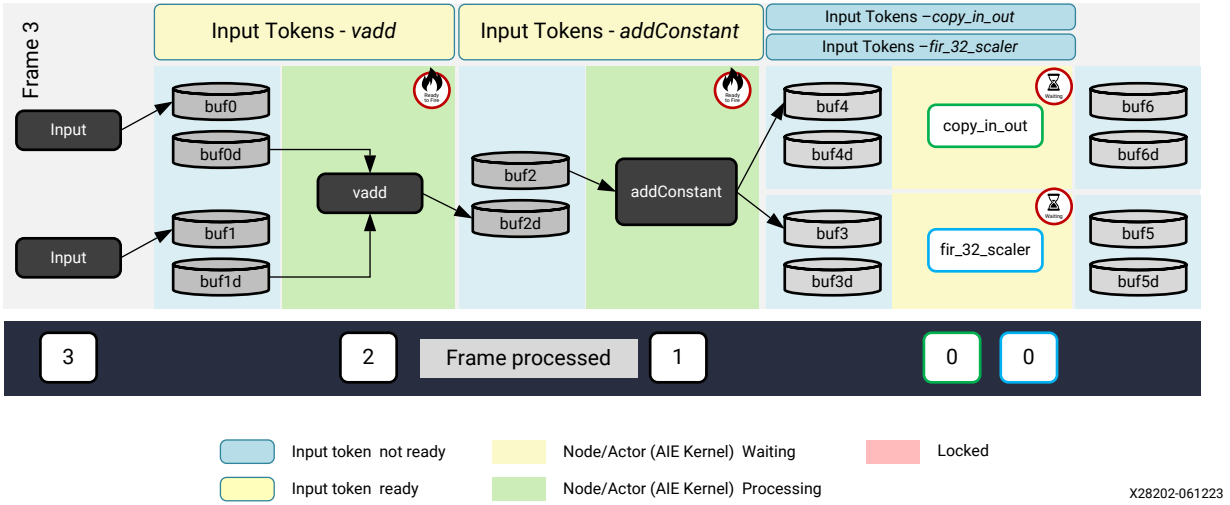


Table 4: **Sending the Frame 3 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 3	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Fill	-	Waiting	-	-	Waiting	-	-

Frame 4

The fourth set of data started writing to the buffers (pong buffer – buf0d/buf1d) and the vadd kernel (node/actor) started processing frame 3. The addConstant kernel (node/actor) started processing frame 2. The fir_32 kernel (node/actor) and copy_in_out kernel (node/actor) started processing frame 1 as the input token is ready.

***Note*:** The fir_32 and copy_in_out kernels are running parallel as the data flow graph is made in such a way to obtain the benefit of parallelism.

Figure 17: **Frame 4**

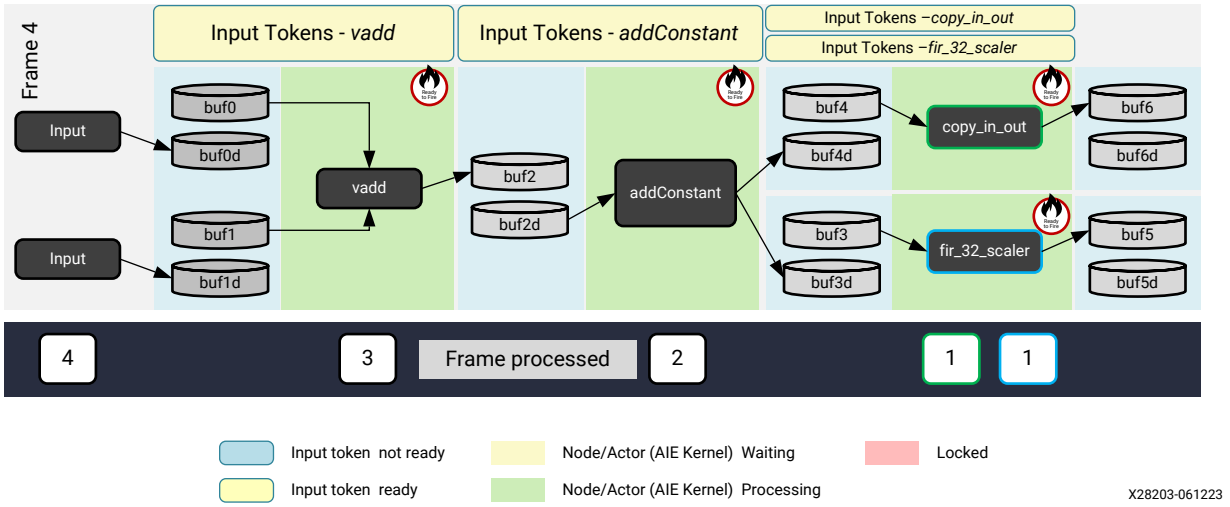


Table 5: **Sending the Frame 4 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-

Table 5: **Sending the Frame 4 – Tokens/Kernels Status** *(cont'd)*

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
Frame 3	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Fill	-	Waiting	-	-	Waiting	-	-
Frame 4	Token ready Frame 3	Fill	Processing (Frame 3)	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Token ready for Frame 1	Fill	Processing (Frame 1)	Fill	-	Processing (Frame 1)	Fill	-

Frame 5

The fifth set of data started writing to the buffers (ping buffer – buf0/buf1) and the vadd kernel (node/actor) started processing frame 4. The addConstant kernel (node/actor) must wait to process frame 2 because it cannot write to the buffer that is used by the fir_32 kernel (node/actor) for processing frame 1. This generates the memory lock for the input tokens. The copy_in_out kernel starts processing frame 2 as the token is ready.

Figure 18: **Frame 5**

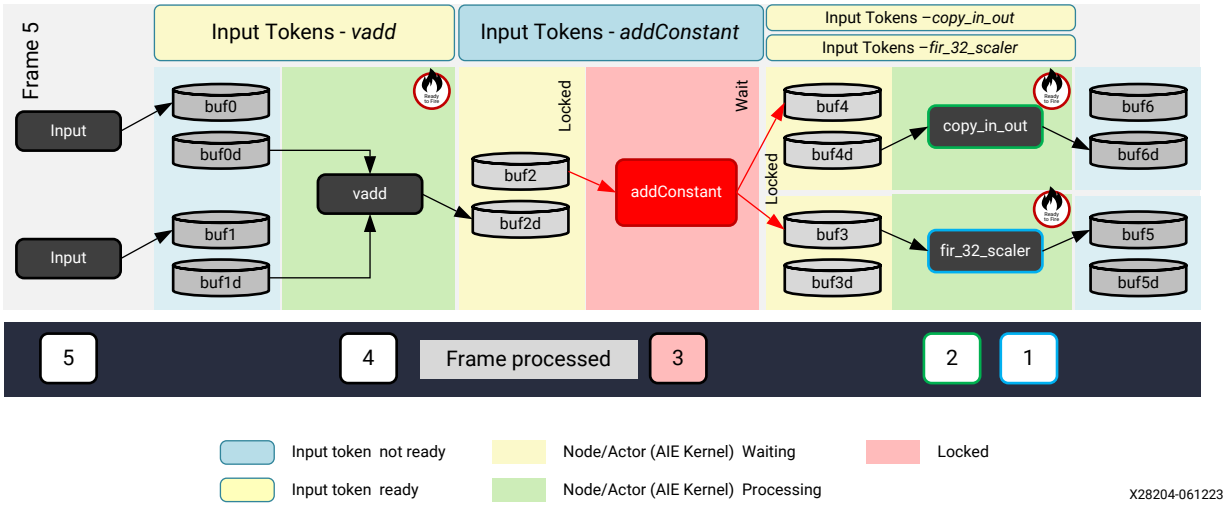


Table 6: **Sending the Frame 5 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 3	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Fill	-	Waiting	-	-	Waiting	-	-

Table 6: **Sending the Frame 5 – Tokens/Kernels Status** *(cont'd)*

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
Frame 4	Token ready Frame 3	Fill	Processing (Frame 3)	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Token ready for Frame 1	Fill	Processing (Frame 1)	Fill	-	Processing (Frame 1)	Fill	-
Frame 5	Fill	Token ready Frame 4	Processing (Frame 4)	-	Locked	Waiting	Locked	Token ready Frame 2	Locked	Token ready Frame 2	Processing (Frame 2)	-	Fill	Processing (Frame 1)	Fill	-

Frame 6

The sixth set of data started writing to the buffers (pong buffer – buf0d/buf1d) and the vadd kernel (node/actor) must wait to process frame 4 because the buffers are locked due to the addConstant kernel (node/actor), which must wait to process frame 3. The fir_32 kernel (node/actor) is still processing frame 1. The copy_in_out kernel must wait to process frame 3 because the fir_32 kernel (node/actor) is still processing frame 1, which locks the buffers (buf3 and buf4). Even though the copy_in_out kernel can run parallel, but the fir_32 is running slow, which generated the lock to the buffers (buf3/buf4).

Figure 19: **Frame 6**

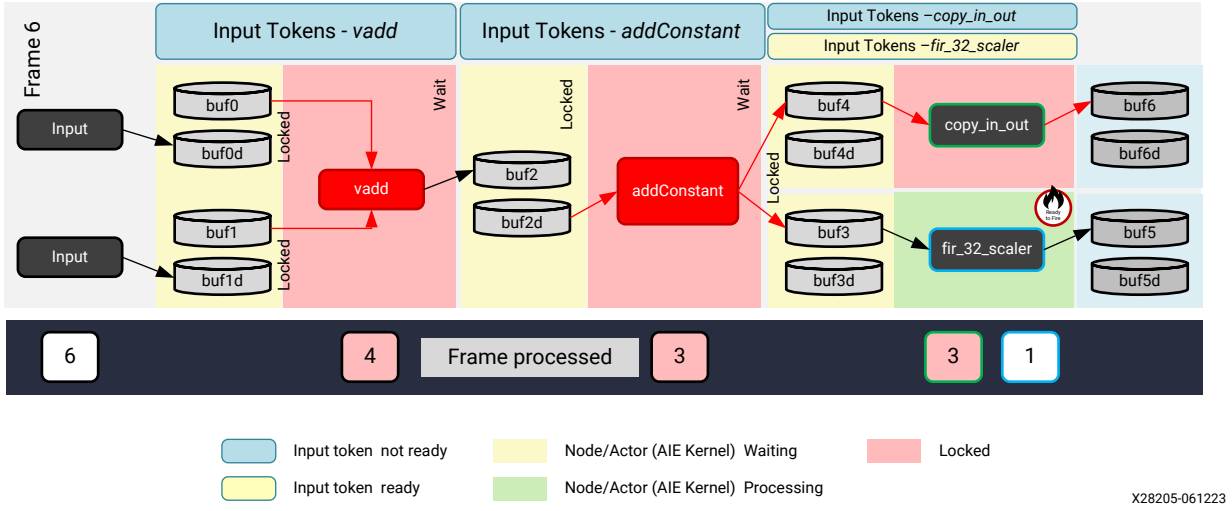


Table 7: **Sending the Frame 6 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-

Table 7: **Sending the Frame 6 – Tokens/Kernels Status** *(cont'd)*

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
Frame 3	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Fill	-	Waiting	-	-	Waiting	-	-
Frame 4	Token ready Frame 3	Fill	Processing (Frame 3)	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Token ready for Frame 1	Fill	Processing (Frame 1)	Fill	-	Processing (Frame 1)	Fill	-
Frame 5	Fill	Token ready Frame 4	Processing (Frame 4)	-	Locked	Waiting	Locked	Token ready Frame 2	Locked	Token ready Frame 2	Processing (Frame 2)	-	Fill	Processing (Frame 1)	Fill	-
Frame 6	Locked	Fill	Waiting	Locked	Locked	Waiting	Locked	Token ready Frame 2	Locked	Locked	Waiting	Fill	-	Processing (Frame 1)	Fill	-

Frame 7

The seventh set of data cannot be written to the buffer due to the lock. This is because the vadd is waiting to process frame 4. The fir_32 kernel (node/actor) is still processing frame 1.

Figure 20: **Frame 7**

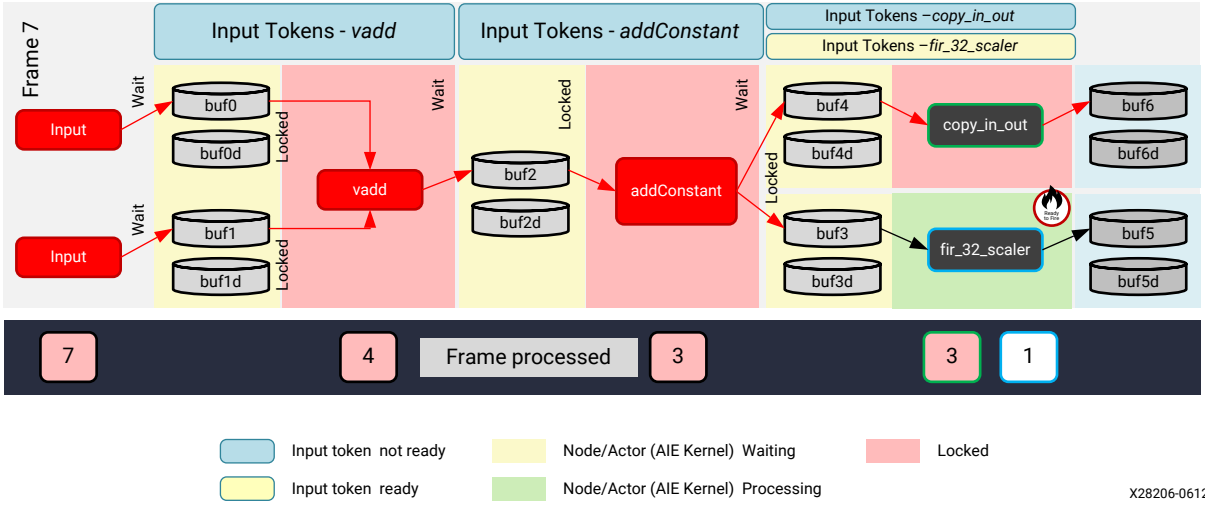


Table 8: **Sending the Frame 7 – Tokens/Kernels Status**

KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
AI Engine	Buffer (ping/pong)		Vadd	Buffer (ping/pong)		addConstant	Buffer (ping/pong)		Buffer (ping/ pong)		copy_in_out	Buffer (ping/ pong)		fir_32	Buffer (ping/ pong)	
	buf0/ buf1	buf0d/ buf1d		buf2	buf2d		buf3	buf3d	Buf4	buf4d		buf6	buf6d		buf5	buf5d
Frame 1	Fill	-	Waiting	-	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 2	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Waiting	-	-	-	-	Waiting	-	-	Waiting	-	-
Frame 3	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Processing (Frame 1)	Fill	-	Fill	-	Waiting	-	-	Waiting	-	-

Table 8: **Sending the Frame 7 – Tokens/Kernels Status** *(cont'd)*

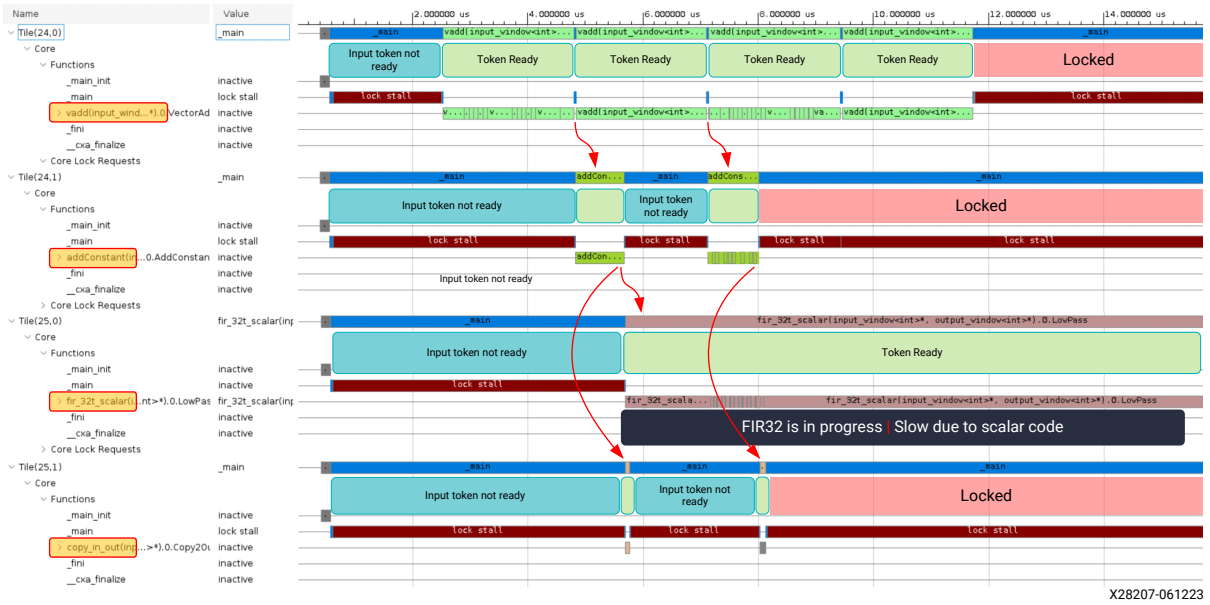
KPN Terminology	Input Token for Vadd		Node/Actor	Input Token for addConstant		Node/Actor	Input Token for fir_32		Input Token for copy_in_out		Node/ Actor	Port		Node/ Actor	Port	
Frame 4	Token ready Frame 3	Fill	Processing (Frame 3)	Fill	Token ready Frame 2	Processing (Frame 2)	Token ready Frame 1	Fill	Token ready for Frame 1	Fill	Processing (Frame 1)	Fill	-	Processing (Frame 1)	Fill	-
Frame 5	Fill	Token ready Frame 4	Processing (Frame 4)	-	Locked	Waiting	Locked	Token ready Frame 2	Locked	Token ready Frame 2	Processing (Frame 2)	-	Fill	Processing (Frame 1)	Fill	-
Frame 6	Locked	Fill	Waiting	Locked	Locked	Waiting	Locked	Token ready Frame 2	Locked	Locked	Waiting	Fill	-	Processing (Frame 1)	Fill	-
Frame 7 (Wait)	Locked	Locked	Waiting	Locked	Locked	Waiting	Locked	Token ready Frame 2	Locked	Locked	Waiting	Fill	-	Processing (Frame 1)	Fill	-

In this case, the fir_32 is implemented using the scalar processor, which is very slow to execute. Implementing the fir_32 using the vector processor solves this issue and works much faster. The key takeaway is not only that the proper data flow improves the performance, but the kernel performance has an impact on the overall system.

The following figures show the event trace view of the scalar code and vector code designs.

In the following figure, the kernel addConstant performs two frames but the kernel fir (fir_32t_scalar) is still processing frame 1. This causes the locks to be generated respectively to the previous buffers and also leads to a kernel stall.

Figure 21: **Event Trace – Scalar Code Design**



As explained in the paragraph following Table 8, the kernel fir has been replaced with the vector version (fir_32t_vector), which performs faster compare to the scalar version. As a result, the locks are prevented.

Figure 22: **Event Trace – Vector Code Design**

Performance Benchmark

This section provides a performance benchmark for ultrafast ultrasonic imaging.

The performance benchmark results are presented in frames per second (fps). There are two application results shown in the following tables, one for small parts imaging and the other for abdominal imaging. The software environment used for the AMD Versal™ adaptive SoC is Vitis software platform 2021.1 and the compute unified device architecture (CUDA) for the GPU. The linear and matched filter interpolation results are presented for both floating point 32 and for integer 16. As these numbers show, the Versal adaptive SoC significantly outperforms the GPU by 44X for linear interpolation for integer and 27X for floating point. For the spline interpolation, the performance is a staggering 91X over the GPU.

Table 9: **Small Parts Ultrasound Imaging**

	Versal Adaptive SoC VCK190	GPU-RTX 2070	PC i7
Linear interpolation	1101 fps	~40 fps	~1 fps
Matched filter Catmull-Rom Spline interpolation	365 fps	~4 fps	~0.006 fps
Linear interpolation (int16)	4406 fps	~100 fps	~1 fps
Matched filter interpolation (int16)	1461 fps	~15 fps	~0.006 fps

Table 10: **Abdominal Imaging**

	Versal Adaptive SoC VCK190	GPU-RTX 2070	PC i7
Linear interpolation	482 fps	~20 fps	~0.25 fps
Matched filter Catmull-Rom Spline interpolation	160 fps	~1 fps	~0.0015 fps
Linear interpolation (int16)	1920 fps	~90 fps	~0.25 fps
Matched filter interpolation (int16)	640 fps	~10 fps	~0.0015 fps

Conclusion

Graph programming can be seen as the same as defining the KPN model while AI Engine kernel programming is purely based on the C++ vector programming with AI Engine APIs. The benefits of data flow programming help to program the graph to use the multiple AI Engines to perform parallel computation efficiently to achieve the maximum benefit from the architecture.

References

These documents provide supplemental material useful with this guide:

1. Kahn Process Networks: https://en.wikipedia.org/wiki/Kahn_process_networks

2. Model of Computation:
   - https://en.wikipedia.org/wiki/Model_of_computation
   - https://cs.brown.edu/people/jsavage/book/pdfs/ModelsOfComputation.pdf
3. Communication-aware_mapping_of_KPN_applications_onto_heterogeneous_MPSoCs: https://ieeexplore.ieee.org/document/6241671
4. *Versal Adaptive SoC AI Engine Architecture Manual* (AM009)
5. *AI Engine Kernel and Graph Programming Guide* (UG1079)
6. *AI Engine Tools and Flows User Guide* (UG1076)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/20/2023 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright