



AI Engine Kernel and Graph Programming Guide (UG1079)

Overview

- Navigating Content by Design Process
- AI Engine Architecture Overview
- AI Engine Memory
- Tools

Single Kernel Programming

- AI Engine API
- Kernel Pragmas
- Kernel Compiler
- Kernel Simulation
- Kernel Inputs and Outputs

Introduction to Scalar and Vector Programming

- AI Engine API Overview
- Scalar Programming
- Vector Data Types
- Vector Registers
- Accumulator Registers
- Casting and Datatype Conversion
- Vector Initialization, Load, and Store
- Vector Arithmetic Operations
- Vector Reduction
- Bitwise Operations
- Data Comparison
- Data Reshaping
- Iterators
- Operator Overloading
- Floating-Point Operations
- Multiple Lanes Multiplication - sliding_mul
- Matrix Multiplication - mmul
- API Operation Examples

Kernel Optimization

- Loops
- Software Pipelining of Loops
- Chess Directives and C++ Attributes
- Scheduling Separator
- Parallel Streams Access
- Restrict Keyword
- Inline Keywords
- Profiling Kernel Code
- Using Vitis Unified IDE and Reports

Interface Considerations

- Data Movement Between AI Engines
- Buffer vs. Stream in Data Communication
- Free-Running AI Engine Kernel
- Runtime Parameter Specification
- AI Engine and PL Kernels Data Communication
- DDR Memory Access through GMIO

Example Designs Using the AI Engine API

- FIR Filter
- Matrix Multiplication

Introduction to Graph Programming

- Prepare the Kernels
- Creating a Data Flow Graph (Including Kernels)
- Recommended Project Directory Structure
- Compiling and Running the Graph from the Command Line

Memory and DMA Programming

- AI Engine Local Memory Access
- Tiling Parameters and Buffer Descriptors
- Tiling Parameters Specification
- Viewing Tiling Parameters in the Vitis IDE

Input and Output Buffers

- Buffer Port-Based Access
- Buffer Port Attributes and API

- Data Access Mechanisms
- Buffer Port Operations for Kernels
- Comparison between Buffer Ports and Windows
- Graph Conversion
- Kernel Conversion
- Kernel Code Conversion

Streaming Data API

- Data Access Mechanisms
- Stream Operations for Kernels

Runtime Graph Control API

- Graph Execution Control
- Runtime Parameter Specification
- Runtime Parameter Update/Read Mechanisms
- Runtime Graph Reconfiguration Using Control Parameters
- Runtime Parameter Support Summary

Specialized Graph Constructs

- Lookup Tables
- FIFO Depth
- Kernel Bypass
- Explicit Packet Switching
- Location Constraints
- Buffer Allocation Control
- C++ Kernel Class Support
- C++ Template Support
- Multicast Support
- Logical I/O Ports
- Conditional Ports
- Array of Graph Objects

AI Engine/Programmable Logic Integration

- Design Flow Using RTL Programmable Logic
- Design Considerations for Graphs Interacting with Programmable Logic
- AI Engine to PL Interface Performance
- AI Engine to PL Interface AXI Protocol
- AI Engine to PL Interface Text Input Format

Graph Programming Model

- Graph Topologies
- Programming Model Features
- Configuring input_plio/output_plio
- Configuring input_gmio/output_gmio
- Performance Comparison Between AI Engine/PL and AI Engine/NoC Interfaces

Single Kernel Programming using Intrinsic

- Intrinsic
- Introduction to Scalar and Vector Programming
- AI Engine Data Types
- Vector Registers
- Accumulator Registers
- Casting and Datatype Conversion
- Vector Initialization, Load, and Store
- Vector Register Lane Permutations
- Loops
- Floating-Point Operations

Design Analysis and Programming using Intrinsic

- Matrix Vector Multiplication
- Matrix Multiplication
- Multiple Kernels Coding Example: FIR Filter

Adaptive Data Flow Graph Specification Reference

- Return Code
- Graph Objects
- Input/Output Objects
- Connections
- Constraints

Using the Restrict Keyword in AI Engine Kernels

- Pointer Aliasing

[Strict Aliasing Rule](#)

[Restrict Keyword](#)

[Restrict Qualification](#)

[Undefined Behavior](#)

[Scope of Restrict Keyword in Inline Function](#)

[Benefits of Using the Restrict Keyword for Read/Modify/Write Loops](#)

[Derived Pointers](#)

[Summary](#)

Additional Resources and Legal Notices

[Finding Additional Documentation](#)

[Support Resources](#)

[References](#)

[Revision History](#)

[Please Read: Important Legal Notices](#)

Overview

AI Engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and AI technology such as machine learning (ML).

The AI Engine array supports three levels of parallelism:

Instruction Level Parallelism (ILP)

Through the VLIW architecture allowing multiple operations to be executed in a single clock cycle.

SIMD

Through vector registers allowing multiple elements (for example, eight) to be computed in parallel.

Multicore

Through the AI Engine array, allowing up to 400 AI Engines to execute in parallel.

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 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, access to local memory in itself and three neighboring AI Engines with the direction depending on the row it is in. It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA. Refer to the *Versal Adaptive SoC AI Engine Architecture Manual* (AM009) for specific details on the AI Engine array and interfaces.

While most standard C code can be compiled for the AI Engine, the code might need restructuring to take full advantage of the parallelism provided by the hardware. The power of an AI Engine is in its ability to execute a multiply-accumulate (MAC) operation using two vectors, load two vectors for the next operation, store a vector from the previous operation, and increment a pointer or execute another scalar operation in each clock cycle. Specialized functions called intrinsics allow you to target the AI Engine vector and scalar processors and provide implementation of several common vector and scalar functions, so you can focus on the target algorithm. In addition to its vector unit, an AI Engine also includes a scalar unit which can be used for non-linear functions and data type conversions.

An AI Engine program consists of a data flow graph (adaptive data flow graph) specification that is written in C++. This specification can be compiled and executed using the AI Engine compiler. An adaptive data flow (ADF) graph application consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. Kernels in the application can be compiled to run on the AI Engines, and are the fundamental building blocks of an ADF graph specification. The ADF graph is a modified **Kahn process network** with the AI Engine kernels operating in parallel. AI Engine kernels operate on data streams. These kernels consume input blocks of data and produce output blocks of data. Kernel behavior can be modified using static data or runtime parameter (RTP) arguments that can be either asynchronous or synchronous.

The following figure shows the conceptual view of the ADF graph and its interfaces with the processing system (PS), programmable logic (PL), and DDR memory. It consists of the following:

AI Engine

Each AI Engine is a VLIW processor containing a scalar unit, a vector unit, two load units, and a single store unit.

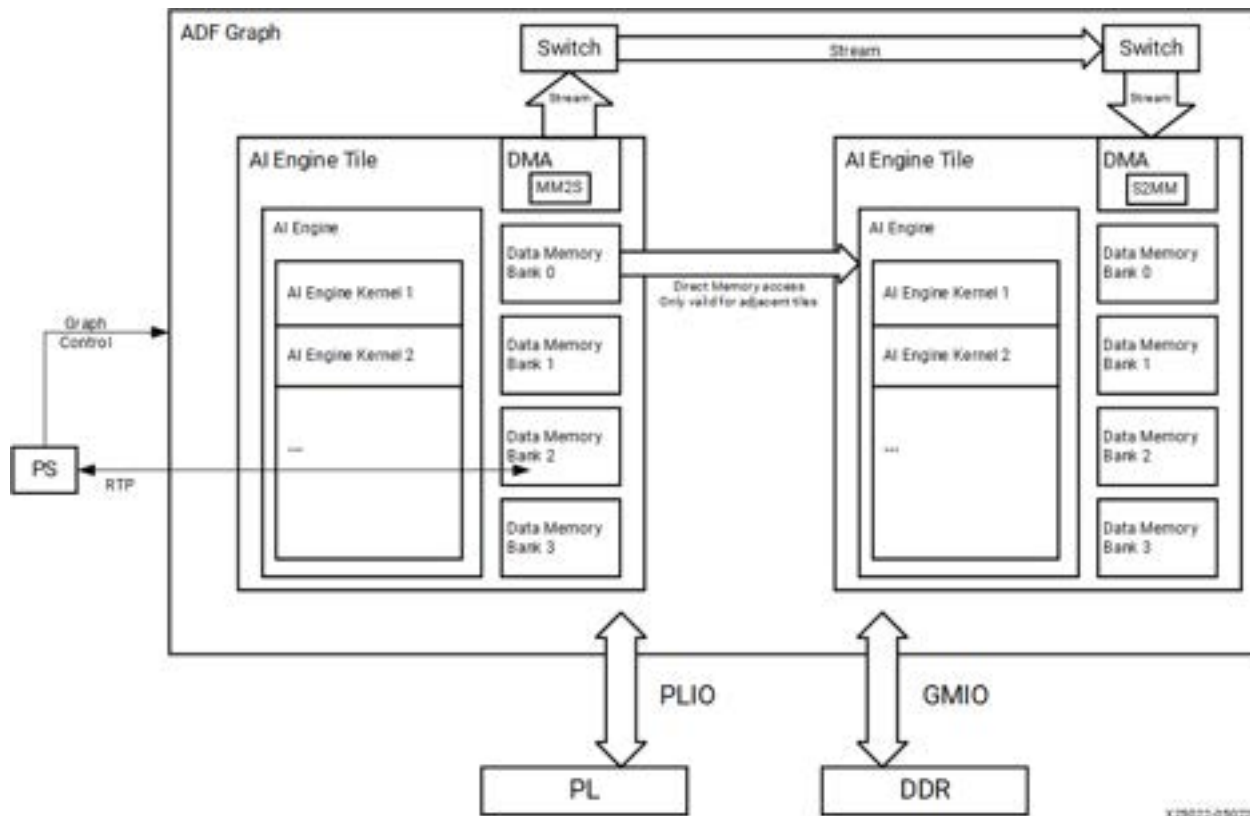
AI Engine Kernel

Kernels are written in C/C++ running in an AI Engine.

ADF Graph

The graph consists of a single AI Engine kernel or multiple AI Engine kernels connected by data streams and/or buffers. It interacts with the PL, global memory, and PS with specific constructs like PLIO (port attribute in the graph programming that is used to make stream connections to or from the programmable logic), GMIO (port attribute in the graph programming that is used to make external memory-mapped connections to or from the global memory), and RTP.

Figure: Conceptual Overview of the ADF Graph



X29022-090228

Recommended: For details about compiling and simulating graphs, and about the AI Engine hardware runtime and debug flows, refer to the *AI Engine Tools and Flows User Guide* ([UG1076](#)).

AI Engine Kernels

An AI Engine kernel is a C++ program written using specialized intrinsic functions. These intrinsic functions target the different functional units of the AI Engine processor, like the VLIW vector and scalar unit. The AI Engine kernel code is compiled using the AI Engine compiler that is included in the AMD Vitis™ core development kit. The AI Engine compiler compiles the kernels to produce an ELF file that is run on the AI Engine processors. Chapters 2 to 6 in this guide provides a high-level overview of kernel programming, tools, and documents that can be referenced for AI Engine kernel programming. In addition, these chapters provide details like scalar/vector programming, kernel optimization, interface considerations, and some examples.

AI Engine Graphs

[Introduction to Graph Programming](#) provides a brief overview of the AI Engine programming model, an introduction to ADF graphs, and information about compiling and simulating an AI Engine graph. [Streaming Data API](#) describes the APIs that are available for data communication between kernels.

Controlling the AI Engine Graph

[Runtime Graph Control API](#) describes the various control APIs available to control and update the AI Engine graphs at runtime. The graph control APIs can be used to initialize, run, update, and control the graph execution from an external controller and runs in the context of a platform. This platform can be a simulation-only platform, an extensible target platform which can be connected to the PL kernels, or a fixed platform for bare-metal applications.

[Specialized Graph Constructs](#) describes specialized graph constructs that are useful in modeling specific scenarios. It includes constructs like FIFO specification, explicit packet switching, and so on.

[AI Engine/Programmable Logic Integration](#) enumerates important points to consider when communicating programmable logic and AI Engine considerations to interface with the programmable logic, and goes into aspects of AI Engine programmable logic interface performance.

[Graph Programming Model](#) details more advanced topics in Adaptive Dataflow graph programming such as graph topologies, I/O specifications, including the supported graph topologies, I/O specifications like AI Engine to/from PL (PLIO), and AI Engine to/from NoC/DDR (GMIO).

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the [Design Hubs](#) page. You can also use the [Design Flow Assistant](#) to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

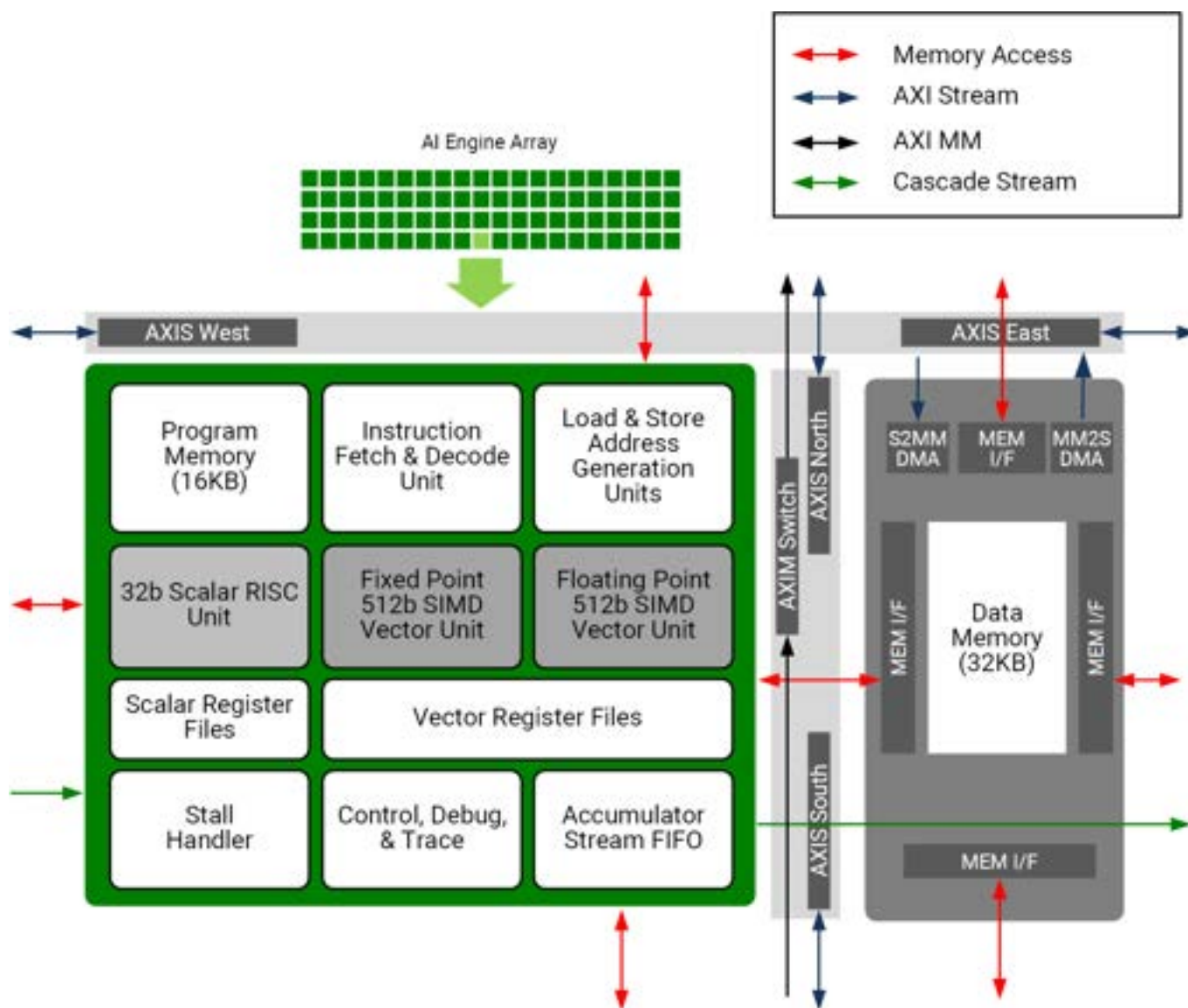
AI Engine Development

Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels.

AI Engine Architecture Overview

The AI Engine array consists of a 2D array of AI Engine tiles, where each AI Engine tile contains an AI Engine, memory module, and tile interconnect module. The AI Engine is a highly-optimized processor featuring a single-instruction multiple-data (SIMD) and very long instruction word (VLIW) instruction set architecture containing six functional units: scalar, vector, two load, one store, and one instruction fetch and decode. One VLIW instruction can support a maximum of two loads, one store, one scalar operation, one fixed-point or floating-point vector operation, and two move instructions. There is also a memory module available that is shared between its north, south, east, or west AI Engine neighbors, depending on the location of the tile within the array. An AI Engine can access its north, south, east, or west, and its own memory module.

Figure: AI Engine Tile Details



X24805-111120

Each AI Engine tile has an AXI4-Stream switch that is a fully programmable 32-bit AXI4-Stream crossbar. It supports both circuit-switched and packet-switched streams with back-pressure. Through MM2S DMA and S2MM DMA, the AXI4-Stream switch provides stream access to and from AI Engine data memory. The switch also contains two 16-deep 33-bit (32-bit data + 1-bit TLAST) wide FIFOs, which can be chained to form a 32-deep FIFO by circuit-switching the output of one of the FIFOs to the other FIFO's input.

More details on the AI Engine architecture can be found in *Versal Adaptive SoC AI Engine Architecture Manual* ([AM009](#)).

AI Engine Memory


Each AI Engine has 16 KB of program memory, which allows storing 1024 instructions of 128 bits each. The AI Engine instructions are 128 bits (maximum) wide and support multiple instruction formats, as well as variable length instructions to reduce the program memory size. Many instructions outside of the optimized inner loop can use the shorter formats.

Each AI Engine tile has eight data memory banks, where each memory bank (single bank) is a 256 word x 128-bit single-port memory (for a total of 32 KB). Each AI Engine can access three of the memories from neighboring tiles plus its own data memory for a total of 128 kB. The stack is a subset of the data memory. The default value for stack size and heap size is 1 KB. Heap size can be automatically computed and adjusted by the compiler when optimization level is larger than zero (`xlopt>=1` for the AI Engine compiler). Stack size and heap size can be changed using compiler options or constraints in the source code. Refer to the *AI Engine Tools and Flows User Guide* ([UG1076](#)) for more information about stack and heap size usage.

In a logical representation, the 128 KB memory can be viewed as one contiguous 128 KB block or four 32 KB blocks, and each block can be divided into four odd and four even banks. One even bank and one odd bank are interleaved to comprise a double bank. AI Engines on the edges of the AI Engine array have fewer neighbors and correspondingly less memory available.

Each memory port operates in 256-bit/128-bit vector register mode or 32-bit/16-bit/8-bit scalar register mode. The 256-bit port is created by an even and odd pairing of the memory banks. The 8-bit and 16-bit stores are implemented as read-modify-write instructions. Concurrent operation of all three ports is supported if each port is accessing a different bank.

Data stored in memory is in little endian format.

 **Recommended:** AMD recommends accessing data memory on a 128-bit boundary with vector operations.

Each AI Engine has a DMA controller that is divided into two separate modules, S2MM to store stream data to memory (32-bit data) and MM2S to write the contents of the memory to a stream (32-bit data). Each S2MM and MM2S has two independent data channels.

Tools

Vitis Integrated Design Environment

The AMD Vitis™ integrated design environment (IDE) can be used to target system programming of AMD devices including Versal devices. It supports development of single and multiple AI Engine kernel applications. The following features are available in the tool.

- An optimizing C/C++ compiler that compiles the kernels and graph code making all of the necessary connections, placements, and checks to ensure proper functioning on the device.
- The kernel code is compiled using the AI Engine compiler, which is included in the AMD Vitis™ core development kit. An option that specifies a platform that targets an AIE-ML / AIE-ML v2 (`--platform=<Path to a platform (XPFM) or hardware (XSA) specification>`) device is also necessary.
- A fast functional simulator that is useful in identifying errors in the design. This simulator is an ideal choice for testing, debugging, and verifying your AI Engine design because of the speed of iteration and the high level of data visibility it provides.
- A cycle-approximate simulator which models the timing and resources of the AI Engine array while using transaction-level SystemC models for the NoC, and DDR memory. This allows for quick performance analysis of your AI Engine applications and accurate estimation of the AI Engine resource use, with cycle-approximate timing information.
- A powerful debugging environment that works in both simulation and hardware environments. Various views are available, such as variables view, disassembly view, memory view, register view, and pipeline view.

Vitis Command Line Tools

Command line tools are available to build, simulate, and generate output files and reports.

- The `v++ -c --mode aie` command compiles kernels and graph code into ELF files that are run on the AI Engine processors.
- `aiesimulator` and `x86simulator` are tools for cycle approximate simulation and functional simulation, respectively.
- The Arm® cross compiler is provided for PS code compilation.
- The `v++ --link` command integrates the compiled Vitis PL and AIE components, and the hardware design.
- The Vitis IDE is available for report viewing and analysis of the output files and reports generated by the command line tools.

The *AI Engine Tools and Flows User Guide* ([UG1076](#)) contains a wealth of information on the design flow and tool usage.

Single Kernel Programming

An AI Engine kernel is a C++ program written using specialized intrinsic functions. These intrinsic functions target the different functional units of the AI Engine processor, like the VLIW vector and scalar unit. The AI Engine kernel code is compiled using the AI Engine compiler, which is included in the AMD Vitis™ core development kit. The AI Engine compiler takes the kernel code and produces ELF files that are run on the AI Engine processors.

The AI Engine supports specialized data types and functions for vector processing. By restructuring some scalar application code with these API functions and vector data types, one can create fast and efficient vectorized code. The compiler takes care of mapping functions to operations, performing register allocation and data movement, scheduling and generation of microcode. This microcode is efficiently packed into VLIW instructions.

The following chapters introduce the data types supported and registers available for use by the AI Engine kernel. In addition, the vector API functions that initialize, load, and store, as well as operate on the vector registers using the appropriate data types are also described.

To achieve the highest performance on the AI Engine, the primary goal of single kernel programming is to ensure that the usage of the vector processor approaches its theoretical maximum. Vectorization of the algorithm is important, but managing the vector registers, memory access, and software pipelining are also required. The programmer must strive to make the data for the new operation available while the current operation is executing because the vector processor is capable of an operation every clock cycle. Optimizations using software pipelining in loops are available using pragmas. For instance, when the inner loop has sequential or loop carried dependencies it might be possible to unroll an outer loop and compute multiple values in parallel. The following sections go over these concepts as well.

AI Engine API

The AI Engine API is a portable programming interface for AI Engine kernel programming. This API interface targets current and future AI Engine architectures. For AI Engine API documentation, see the *AI Engine API User Guide* ([UG1529](#)).

The interface provides parameterizable data types that enable generic programming and also implements the most common operations in a uniform way across the different data types. It is an easier programming interface as compared to using intrinsic functions. It is implemented as a C++ header-only library that gets translated into optimized intrinsic functions.

For an advanced user that needs programming with intrinsics, refer to *AI Engine Intrinsics User Guide* ([UG1078](#)).

The AI Engine API user guide is organized as follows:

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

Figure: AI Engine API User Guide



Kernel Pragmas

The AI Engine compiler supports dedicated directives for efficient loop scheduling. Additional pragmas for reducing memory dependencies and removing function hierarchy while further optimizing kernel performance is also included. Examples of the use of those pragmas can be found in this document.

A list of all pragmas and functions used in kernel coding is available in the *ASIP Programmer Chess Compiler User Manual*, which can be found in the [Versal AI Engines Secure Site](#).

Kernel Compiler

The AI Engine compiler (`v++ -c --mode aie`) is used to compile AI Engine kernel code. [Compiling an AI Engine Graph Application](#) in the *AI Engine Tools and Flows User Guide* ([UG1076](#)) describes in detail the AI Engine compiler usage, options available, input files that can be passed in, and expected output.

Kernel Simulation

To simulate an AI Engine kernel you need to write an AI Engine graph application that consists of a data-flow graph specification written in C++. This graph contains the AI Engine kernel, with test bench data being provided as input(s) to the kernel. The data output(s) from the kernel can be captured as the simulation output and compared against golden reference data. This specification can be compiled and executed using the AI Engine compiler (`v++ -c --mode aie`). The application can be simulated using the `aiesimulator`. For additional details on the `aiesimulator`, see [Simulating an AI Engine Graph Application](#) in the *AI Engine Tools and Flows User Guide* ([UG1076](#)).

Kernel Inputs and Outputs

AI Engine kernels operate on either streams or blocks of data. AI Engine kernels operate on data of specific types, for example, `int32` and `cint32`. A block of data used by a kernel is stored in a buffer. Kernels consume input streams and/or buffers, and produce output streams and/or buffers. Kernels access data streams in a sample-by-sample fashion. For additional details on stream APIs, see [Streaming Data API](#). For additional details on buffers, see [Input and Output Buffers](#).

AI Engine kernels can also have RTP ports which are accessible by the PS. For more information about RTP, see [Runtime Graph Control API](#).

Introduction to Scalar and Vector Programming

This section provides an overview of the key elements of kernel programming for scalar and vector processing elements. The details of each element and optimization skills are seen in following sections.

The following example demonstrates a `for` loop iterating through 512 `int32` elements. Each loop iteration pulls one element from each input buffer, multiplies them together and places the product in the output buffer. The `scalar_mul` kernel operates on two input buffers `input_buffer<int32>` and updates an output buffer `output_buffer<int32>`.

The iterators are used to read and write to the buffers outside the kernel. The code sample below does not use any vector registers and will be implemented on the scalar engine.

```
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>
using namespace adf;
void scalar_mul(input_buffer<int32>& __restrict data1,
               input_buffer<int32>& __restrict data2,
               output_buffer<int32>& __restrict out) {
    auto inIter1=aie::begin(data1);
    auto inIter2=aie::begin(data2);
    auto outIter=aie::begin(out);
    for(int i=0;i<512;i++) {
        int32 a=*inIter1++;
        int32 b=*inIter2++;
        int32 c=a*b;
        *outIter++=c;
    }
}
```

The following example is a vectorized version for the same kernel and will be implemented on the vector processor.

```

#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>
using namespace adf;
void vect_mul(input_buffer<int32>& __restrict data1,
              input_buffer<int32>& __restrict data2,
              output_buffer<int32>& __restrict out) {
    //iterator for vector of 8 elements
    auto inIter1=aie::begin_vector<8>(data1);
    auto inIter2=aie::begin_vector<8>(data2);
    auto outIter=aie::begin_vector<8>(out);
    for(int i=0;i<512/8;i++) chess_prepare_for_pipelining {
        //vector of 8 elements
        auto va=*inIter1++;
        auto vb=*inIter2++;

        //element-by-element multiplication
        auto vt=aie::mul(va,vb);
        *outIter++=vt.to_vector<int32>(0);
    }
}

```

The iterators returns a vector of eight int32 and stores them in variables named va and vb. These two variables are vector type variables and they are passed to the API function `aie::mul`. The result of the `aie::mul` function is stored in vt, which is an accumulator with data type `aie::accum<acc80,8>`. The accumulator is then converted by the *shift-round-saturate* function `to_vector` to a variable of `aie::vector<int32,8>` type. The result is then written to the output buffer. Additional details on the data types supported by the AI Engine are covered in the following sections.

The `__restrict` keyword used on the input and output parameters of the functions, allows for more aggressive compiler optimization by explicitly stating independence between data. For more information, see [Restrict Keyword](#).

`chess_prepare_for_pipelining` is a compiler pragma that explicitly directs kernel compiler to achieve optimized pipeline for the loop.

The scalar version of this example function needs 1045 cycles, while the vectorized and optimized version needs only 88 cycles. That means that there is more than ten times speedup for the vectorized version of the kernel. Vector processing itself would give 8x the throughput for int32 multiplication. However, with the loop optimizations done, it can get more than 10x.

To calculate the maximum performance for a given datapath, it is necessary to multiply the number of multiply-accumulates (MACs) per instruction with the clock frequency of the AI Engine kernel. For example, with 32-bit input vectors X and Z, the vector processor can achieve 8 MACs per instruction. Using the clock frequency for the slowest speed grade device results in:

8 MACs * 1 GHz clock frequency = 8 Giga MAC operations/second

The sections that follow describe in detail the various data types that can be used, registers available, and also the kinds of optimizations that can be achieved on the AI Engine using concepts like software pipelining in loops and keywords like `__restrict`.

Related Information

[Software Pipelining of Loops](#)

AI Engine API Overview

The AI Engine API is a portable programming interface for AI Engine accelerators. The *AI Engine API User Guide* ([UG1529](#)) indicates if the API is supported only on a specific architecture. The API are 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.

Usually, two header files are needed in kernel source code:

aie_api/aie.hpp

AI Engine main entry point.

aie_api/aie_adf.hpp

Graph buffer and stream interfaces.

AI Engine API provides a helper file to print `aie::vector` and `aie::mask` values in simulation when profiling is enabled:

- `aie_api/utils.hpp`: `aie::print` and `aie::print_matrix` functions are provided.

To support operator overloading on some operations, include header file `aie_api/operators.hpp` and use namespace `aie::operators`. For additional information, see [Operator Overloading](#).

A typical example of using the API for an AI Engine kernel is as follows.

```

#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>
using namespace adf;
void vec_incr(input_buffer<int32>& data, output_buffer<int32>& out) {

    //set all elements to 1
    aie::vector<int32,16> vec1=aie::broadcast(1);

    auto inIter=aie::begin_vector<16>(data);
    auto outIter=aie::begin_vector<16>(out);

    for(int i=0;i<16;i++) chess_prepare_for_pipelining {
        aie::vector<int32,16> vdata=*inIter++;

        //print vector in a line
        aie::print(vdata,true,"vdata=");

        //print vdata as a 2x8 matrix
        aie::print_matrix(vdata,8,"vdata matrix=");

        //increment each element in vdata by 1
        aie::vector<int32,16> vresult=aie::add(vdata,vec1);

        *outIter++=vresult;
    }
}

```

Running this code would display the following text in the console:

```

vdata=0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
vdata matrix=0 1 2 3 4 5 6 7
                8 9 10 11 12 13 14 15
vdata=16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
vdata matrix=16 17 18 19 20 21 22 23
                24 25 26 27 28 29 30 31
vdata=32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
vdata matrix=32 33 34 35 36 37 38 39
                40 41 42 43 44 45 46 47
vdata=48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
vdata matrix=48 49 50 51 52 53 54 55
                56 57 58 59 60 61 62 63
vdata=64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
vdata matrix=64 65 66 67 68 69 70 71
                72 73 74 75 76 77 78 79

...

vdata=208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
vdata matrix=208 209 210 211 212 213 214 215
                216 217 218 219 220 221 222 223
vdata=224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
vdata matrix=224 225 226 227 228 229 230 231
                232 233 234 235 236 237 238 239
vdata=240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
vdata matrix=240 241 242 243 244 245 246 247
                248 249 250 251 252 253 254 255

```

Vector data type and operations are covered in following sections.


Scalar Programming

The compiler and scalar unit enable the programmer to use standard C data types. The following table shows standard C data types with their precisions. All types, except float and double, support signed and unsigned prefixes.

Table: Scalar Data Types

Data Type	Precision	Comment
char	8-bit signed	
short	16-bit signed	
int	32-bit signed	Native support
long	64-bit signed	
float	32-bit	Emulated. Scalar processor does not contain a floating point unit (FPU).
double	64-bit	Emulated. Scalar processor does not contain a floating point unit (FPU).

It is important to remember that control flow statements such as branching are still handled by the scalar unit even in the presence of vector instructions. This concept is critical to maximizing the performance of the AI Engine.

 **Note:** GNU defined types like *complex* defined in `<complex.h>` are not supported in AI Engine hardware. GNU defined types work for x86 simulation but not for AI Engine simulation.

Vector Data Types

The two main vector types offered by the AI Engine API are vectors (`aie::vector`) and accumulators (`aie::accum`).

Vector

A vector represents a collection of elements of the same type which is transparently mapped to the corresponding vector registers supported on AI Engine architectures. Vectors are parameterized by the element type and the number of elements, and any combination that defines a 128 b/256 b/512 b/1024 b vector is supported, with 512 b being the default.

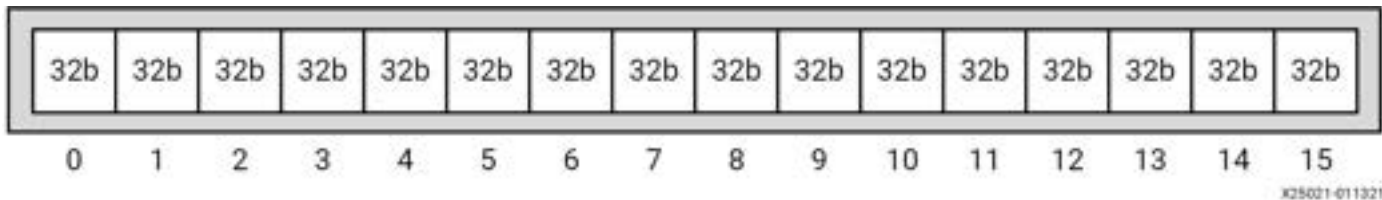
Table: Supported Vector Types and Sizes

Vector Types	Sizes ¹
int8	16/32/64/128
int16	8/16/32/64
int32	4/8/16/32
uint8	16/32/64/128
float	4/8/16/32
cint16	4/8/16/32
cint32	2/4/8/16
cfloat	2/4/8/16

1. The integer in *bold* is the native vector size for the data type supported in the AI Engine. For example, `aie::broadcast((cfloat){1,1})` is equivalent to `aie::broadcast<cfloat,8>((cfloat){1,1})`, where specifying `<cfloat,8>` is optional because it is the native vector size for the `cfloat` data type.

For example, `aie::vector<int32,16>` is a 16 element vector of integers with 32 bits. Each element of the vector is referred to as a *lane*. Using the smallest bit width necessary can improve performance by making good use of registers.

Figure: `aie::vector<int32,16>`



Complex integer and complex float data can be accessed with the member functions `real` and `imag` to extract the real and imagery parts of the data. The real part is stored in the low address, and the imaginary part is stored in the high address. For example:

```
cint8 ctmp={1,2};

// print real and imag values
printf("real=%d imag=%d\n",ctmp.real,ctmp.imag);

// store real and imag values
printf("ctmp mem storage=%llx\n",*(long long*)&ctmp);
cint32 ctmp2={3,4};
int32 *p_ctmp2=reinterpret_cast<int32*>(&ctmp2);

// print "real=3" and "imag=4"
printf("real=%d imag=%d\n",p_ctmp2[0],p_ctmp2[1]);
```

`aie::vector` and `aie::accum` have member functions to do type casting, data extraction and insertion, and indexing. These operations are covered in following sections.

Accumulator

An accumulator represents a collection of elements of the same class, typically obtained as a result of a multiplication operation, which is transparently mapped to the corresponding accumulator registers supported on each architecture. Accumulators commonly provide a large number of bits, allowing users to perform long chains of operations whose intermediate results might exceed the range of regular vector types. Accumulators are parameterized by the element type, and the number of elements. The native accumulation bits define the *minimum* number of bits and the AI Engine API maps different types to the nearest native accumulator type that supports the requirement. For example, `acc40` maps to `acc48` for the AI Engine architecture.

The following table lists the native accumulator data types that are supported by AI Engine API in kernel coding. For a complete list of data types supported in AI Engine API kernel coding, see [Basic Types](#) in the *AI Engine API User Guide (UG1529)*. For the kernel cascade interface and graph support, see [Stream Data Types](#).

Table: Supported Accumulator Types and Sizes

Type	acc48	cacc48	acc80	cacc80	accfloat	caccfloat
Native accumulation bits	48		80		32	

Vector Registers

All vector intrinsic functions require the operands to be present in the AI Engine vector registers. The following table shows the set of vector registers and how smaller registers are combined to form larger registers.

Table: Vector Registers

128-bit	256-bit	512-bit	1024-bit	
vrl0	wr0	xa	ya	N/A
vrh0				
vrl1	wr1			
vrh1				
vrl2	wr2	xb		yd (MSBs)
vrh2				

128-bit	256-bit	512-bit	1024-bit	
vrl3	wr3			
vrh3				
vcl0	wc0	xc	N/A	N/A
vch0				
vcl1	wc1			
vch1				
vdI0	wd0	xd	N/A	yd (LSBs)
vdh0				
vdI1	wd1			
vdh1				


The underlying basic hardware registers are 128-bit wide and prefixed with the letter `v`. Two `v` registers can be grouped to form a 256-bit register prefixed with `w`. `wr`, `wc`, and `wd` registers are grouped in pairs to form 512-bit registers (`xa`, `xb`, `xc`, and `xd`). `xa` and `xb` form the 1024-bit wide `ya` register, while `xb` and `xd` form the 1024-bit wide `yd` register. This means the `xb` register is shared between the `ya` and `yd` registers. `xb` contains the most significant bits (MSBs) for both `ya` and `yd` registers.

The vector register name can be used with the `chess_storage` directive to force vector data to be stored in a particular vector register. For example:

```
aie::vector<int32,8> chess_storage(wr0) bufA;
aie::vector<int32,8> chess_storage(WR) bufB;
```

When upper case is used in the `chess_storage` directive, it means register files (for example, any of the four `wr` registers). Lower case in the `chess_storage` directive means the specified register (for example, `wr0` in the previous code example) is used.

This Chess directive can be replaced with a C++ compliant directive: `[[chess::storage(<reg>)]]`.

 **Note:** Certain vector operations, such as MAC, require operands to come from a subset of registers, so a storage directive might clash with the operation. The `chess_storage` directive should be used with caution.

Vector registers are a valuable resource. If the compiler runs out of available vector registers during code generation, then it generates code to spill the register contents into local memory and read the contents back when needed. This consumes extra clock cycles.

The name of the vector register used by the kernel during its execution is shown for vector load/store and other vector-based instructions in the kernel microcode. This microcode is available in the disassembly view in Vitis IDE. For additional details on Vitis IDE usage, see [Using Vitis Unified IDE and Reports](#).

The `aie::vector` has member functions to support multiple operations on vectors. Some common operations include:

insert()

Updates the contents of a region of the vector using the subvector and returns a reference to the updated vector.

grow()

Returns a copy of the current vector in a larger vector. The contents of the new elements are undefined.

grow_replicate()

The vector is replicated multiple times in the returned larger vector.

extract()

Returns a subvector with the contents of a region of the vector.

push()

Shifts all elements in the vector up and writes the given value into the first position of the vector (the element in the last position of the vector is lost).

cast_to()

Reinterprets the current vector as a vector of the given type. The number of elements is automatically computed by the function.

set()

Updates the value of the element on the given index.

get()

Returns the value of the element on the given index.

operator[]

Returns a constant or non-constant reference object to the element on the given index.

```
aie::vector<int16,16> wv;
aie::vector<int16,8> vv0,vv1;

// insert content of vv0 to lower half of wv
wv.insert(0,vv0);

// insert content of vv1 to higher half of wv
wv.insert(1,vv1);

// grow() returns a vector of size 32
// returned vector is assigned to xv
// lower 16 values in xv is assigned the values from wv
aie::vector<int16,32> xv=wv.grow<32>(0);

// wv is replicated 4 times using grow_replicate()
// vector of size 64 is assigned to xv2
aie::vector<int16,64> xv2=wv.grow_replicate<64>();

int a = 100;
aie::vector<int32,4> v1,v2;

// set 0th element to a
v1[0]=a;


// another method to set 0th element to a
v1.set(a,0);
a=v1.get(0);

// operator[] is preferred
// Element extraction may be merged
// with the underlying operation with no cost in cycles
auto v3 = aie::add(v1[3], v2);

// Element extraction and add in different cycles
v3 = aie::add(v1.get(3), v2);

// cast wv to complex type
aie::vector<cint16,8> cv=wv.cast_to<cint16>();

// extract higher half from cv
aie::vector<cint16,4> cv0=cv.extract<4>(*idx=*/1);
```

 **Note:** The updates replace the content of a part of the vector register. If a vector operation tries to access the updated content, the compiler re-arranges the operations to ensure it operates on correct data. This can impact the performance of the kernel.

`aie::vector` has the member function `push` to update a vector by inserting a new element at the beginning of a vector and shifting the other elements by one.

```
aie::vector<int32,4> v1;
v1.push(100);
```

When defining and implementing a template function for an element type that is templated, standard C++ syntax requires that any variable of the template type calling its member functions be prepended `template` to the member function name. For example:

```
template<typename ELEMENT_TYPE> void func_test(){
    aie::vector<ELEMENT_TYPE,8> wv;
    aie::vector<ELEMENT_TYPE,16> xv=wv.template grow<16>(0);
    aie::vector<ELEMENT_TYPE,8> wv2=xv.template extract<8>(1);
}
```

Accumulator Registers

The accumulation registers are 384 bits wide and can be viewed as eight vector lanes of 48 bits each. The idea is to have 32-bit multiplication results and accumulate over those results without bit overflows. The 16 guard bits allow up to 2^{16} accumulations. The output of fixed-point vector MAC and MUL intrinsic functions is stored in the accumulator registers. The following table shows the set of accumulator registers and how smaller registers are combined to form large registers.

Table: Accumulator Registers

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

The accumulator registers are prefixed with the letters 'am'. Two of them are aliased to form a 768-bit register that is prefixed with 'bm'. The `to_vector` operation moves a value from an accumulator register to a vector register with any required shifting and rounding.

```
aie::accum<acc48,8> acc;

// shift right 10 bits from accumulator
aie::vector<int32,8> res=acc.to_vector<int32>(10); register to vector register
```

The `from_vector` operation is used to move a value from a vector register to an accumulator register with upshifting.

```
aie::vector<int32,8> v;
aie::accum<acc48,8> acc;

// shift left 10 bits
// from vector register to accumulator register
acc.from_vector(v, 10);
aie::print(acc,true,"acc value=");
```

Besides `from_vector()` and `to_vector()` functions, `aie::accum` class has the following member functions similar to `aie::vector`.

insert()

Updates the contents of a region of the accumulator using the values in the given native subaccumulator and returns a reference to the updated accumulator.

grow()

Returns a copy of the current accumulator in a larger accumulator. The value of the new elements is undefined.

extract()

Returns a subaccumulator with the contents of a region of the accumulator.

cast_to()

Reinterprets the current accumulator as an accumulator of the given type. The number of elements is automatically computed by the function.

```
int32 data[8]={1,2,3,4,5,6,7,8};
aie::vector<int32,8> v=aie::load_v<8>(data);
aie::accum<acc48,8> acc;

// shift left 0 bits
acc.from_vector(v, 0);

aie::accum<acc48,16> acc2=acc.grow<16>();
aie::print(acc2,true,"acc2 value=");
```

```
acc2.insert(1,acc);
aie::print(acc2,true,"acc2 value=");

// extract lower part, and cast to cacc48
aie::accum<cacc48,4> cacc1=acc2.extract<8>(0).cast_to<cacc48>();
aie::print(cacc1,true,"cacc1 value=");
```

Floating-point intrinsic functions do not have separate accumulation registers and instead return their results in a vector register. The following streaming data APIs can be used to read and write floating-point accumulator data from or to the cascade stream.

```
aie::vector<float,8> readincr_v<8>(input_cascade<accfloat> * str);
aie::vector<cfloat,4> readincr_v<4>(input_cascade<caccfloat> * str);
void writeincr(output_cascade<accfloat>* str, aie::vector<float,8> value);
void writeincr(output_cascade<caccfloat>* str, aie::vector<cfloat,4> value);
```

For additional details on stream APIs, see [Streaming Data API](#). For additional details on buffers, see [Input and Output Buffers](#).

Rounding and Saturation Modes

The `aie::set_rounding()` and `aie::set_saturation()` APIs are used to set the rounding and saturation modes of the `to_vector` operation.

The `aie::rounding_mode` includes:

floor

Truncate LSB, always round down (towards negative infinity). Default.

ceil

Always round up (towards positive infinity).

positive_inf

Rounds halfway towards positive infinity.

negative_inf

Rounds halfway towards negative infinity.

symmetric_inf

Rounds halfway towards infinity (away from zero).

symmetric_zero

Rounds halfway towards zero (away from infinity).

conv_even

Rounds halfway towards nearest even number.

conv_odd

Rounds halfway towards nearest odd number.

The `aie::saturation_mode` includes:

none

No saturation is performed and the value is truncated on the MSB side. Default.

saturate

Controls saturation. Saturation rounds an n-bit signed value in the range

$[- (2^{(n-1)}) : + 2^{(n-1)} - 1]$

. For example if n=8, the range would be [-128:127].

symmetric

Controls symmetric saturation. Symmetric saturation rounds an n-bit signed value in the range

$[- (2^{(n-1)} - 1) : + 2^{(n-1)} - 1]$

. For example if n=8, the range would be [-127:127].

The rounding and saturation settings are applied on the AI Engine tile. The `aie::get_rounding` and `aie::get_saturation` methods can be used to get the current rounding and saturation modes.

Following are some example codes setting and getting rounding and saturation modes:

```

aie::set_rounding(aie::rounding_mode::ceil);
aie::set_saturation(aie::saturation_mode::saturate);
aie::rounding_mode current_rnd=aie::get_rounding();
aie::saturation_mode current_sat=aie::get_saturation();
...
aie::set_rounding(aie::rounding_mode::floor);
aie::set_saturation(aie::saturation_mode::none);

```

Note: Rounding and saturation settings are sticky. This means that the AI Engine tile maintains the mode until it is changed. If a kernel has a runtime ratio of less than one such that it can share a tile with other kernels, and if the rounding and saturation modes matter to the kernel operation, then the kernel must set these modes in the kernel code rather than in a constructor or initialization function. This ensures that the rounding and saturation modes are guaranteed in the case that another kernel on the same tile uses different rounding or saturation mode values.

Casting and Datatype Conversion

Casting functions (`aie::vector_cast<DstT>(const Vec& v)` and `aie::vector.cast_to<DstT>()`) allow value casting between vector types with the same size in bits. Accumulator vector types have the casting function `aie::accum.cast_to<DstT>()`. Generally, using the smallest data type possible reduces register spillage and improve performance. For example, if a 48-bit accumulator (`acc48`) can meet the design requirements then it is preferable to use that instead of a larger 80-bit accumulator (`acc80`).

Note: The `acc80` vector data type occupies two neighboring 48-bit lanes.

```

aie::vector<int16,8> iv;
aie::vector<cint16,4> cv=iv.cast_to<cint16>();
aie::vector<cint16,4> cv2=aie::vector_cast<cint16>(iv);
aie::accum<cacc48,4> acc=aie::mul(cv,cv2);
aie::accum<acc64,4> acc2=acc.cast_to<acc64>();

```

Standard C++ casts can be also used. But the recommended ways of reading vectors from a buffer are as follows.

- Use `aie::load_v` and increment the scalar pointer by the number of elements in the vector.
- Using vector iterators.

Additional details about `aie::load_v` and iterators are covered in the following sections.

```

int16 coeff_buffer[16];

// cast to int32 and load
aie::vector<int32,8> coeff=aie::load_v<8>((int32*)coeff_buffer);

// create vector<int16,8> iterator
auto it = aie::begin_vector<8>(coeff_buffer);

// read first vector<int16,8>
aie::vector<int16,8> vec0=*it++;

// read second vector<int16,8>
aie::vector<int16,8> vec1=*it;

```

The API supports floating-point to fixed-point (`to_fixed()`) and fixed-point to floating-point (`to_float()`) conversions. The conversion functions (`to_float()` and `to_fixed()`) can be handled by either the vector or scalar engines depending on the function called.

Note: The AI Engine floating-point is not completely compliant with the IEEE standards. For more information about the exceptions, see *Versal Adaptive SoC AI Engine Architecture Manual (AM009)*.

```

int a=48;
float f1=aie::to_float(a);

// first argument is the value of f1
// second argument is the position of input decimal point
float f2=aie::to_float(a,2);

int b1=aie::to_fixed(f1);

```

```
// first argument is the value of f1
// second argument is the position of output decimal point
int b2=aie::to_fixed(f1,2);
aie::vector<float,32> fv;
aie::vector<int32,32> iv=aie::to_fixed<int32>(fv,2);
```

The vector engine offers two implementations of the `to_fixed()` functions: `safe`, which is the default, and `fast`. The `safe` implementation offers more strict data type checks. For the `fast` implementation, you can run the `v++ -c --mode aie` command with the `--fastmath` option, which lets `to_fixed()` choose the `fast` implementation.

The scalar engine offers three implementations for the `to_fixed()` function for floating point scalar operations:

Table: Scalar Floating-Point Operations Options

Compiler Option	Description
<code>--fastmath</code>	The <code>to_fixed</code> and floating-point comparison has two implementations. Fast <code>to_fixed</code> can have wrong results if the shift amount is greater than 1 while the <code>safe</code> version requires more cycles to complete. Fast floating-point comparison gives wrong results with <code>+0</code> and <code>-0</code> while the <code>safe</code> version is correct but takes more cycles.
<code>--fast-floats</code>	Floating point scalar operations, like add, subtract, multiply, and compare, can either be mapped on <code>vector floating point</code> or on <code>softfloat lib</code> . By default, <code>softfloat lib</code> implementation is chosen (<code>--fast-floats=false</code>). This takes quite a few cycles because it is emulated, but the vector floating-point processor can be used at the same time.
<code>--fast-nonlinearfloats</code>	Floating point non-linear scalar operations, like sine/cosine, sqrt, and inv, can either be mapped on <code>scalar non-linear function</code> or on <code>runtime lib (math.c)</code> . By default, <code>runtime lib</code> implementation is chosen (<code>--fast-nonlinearfloats=false</code>) which takes quite a few cycles because it is emulated.

Data can be moved from vector to accumulator using `aie::accum.from_vector()` or from accumulator to vector using `aie::accum.to_vector<DstT>()` with shifting and rounding, as shown in the following example.

```
aie::vector<int16,16> v;
aie::accum<acc32,16> acc;
acc.from_vector(v, 0);

aie::accum<acc32,16> acc2;
aie::vector<int16,16> v2;
v2 = acc2.to_vector<int16>(15);
```

The API supports vector conversions between data types:

- `aie::pack`: Returns a vector by converting each element into half number of bits.
- `aie::unpack`: Returns a vector by converting each element into twice the number of bits.

Following is an example code of data conversion and its example output:

```
aie::vector<int16,16> data;
aie::print(data,true,"data=");
aie::vector<int8,16> data_smaller=data.pack();
aie::vector<int16,16> data_larger=data_smaller.unpack();
aie::print(data_smaller,true,"smaller data=");
aie::print(data_larger,true,"larger data=");
//Example output:
//data=0 1 2 -32768 -4 -5 -6 32767 3 4 126 130 -8 -9 -300 0
//smaller data=0 1 2 0 -4 -5 -6 -1 3 4 126 -126 -8 -9 -44 0
//larger data=0 1 2 0 -4 -5 -6 -1 3 4 126 -126 -8 -9 -44 0
```

Vector Initialization, Load, and Store

Vector registers can be initialized, loaded, and saved in a variety of ways. For optimal performance, it is critical that the local memory that is used to load or save the vector registers be aligned on 16-byte boundaries.

Alignment

Applications can load from data memory (DM) into vector registers and store the contents of vector registers into DM. Memory instructions in the AI Engine that operate on vectors have alignment requirements. Therefore, functions are provided for both aligned and unaligned accesses.

The following functions are assumed to operate on pointers that have met the alignment requirements for a vector load or store of the size.

- `aie::load_v`
- `aie::store_v`

Note: The 128-bit or 256-bit vector load and store operations in the AI Engine require that data be aligned on 128 bits. If the aligned vector load and store operations are used on unaligned pointers, the resulting data might be incorrect.

The following functions are assumed to operate on pointers that have only aligned to the element of the vector.

- `aie::load_unaligned_v`
- `aie::store_unaligned_v`

For optimal performance, vector load and store must operate on memory that has met the vector operation alignment requirement. Unaligned accesses can incur additional overhead depending on the amount of misalignment.

Note: Kernel buffer interfaces ensure that internal buffers have the required alignment for vector loads.

The `alignas` standard C specifier can be used to ensure proper alignment of local memory. In the following example, `reals` is aligned to 16-byte boundary.

```
// align to 16 bytes boundary
// equivalent to "alignas(aie::vector<int16,8>)"
alignas(16) const int16 reals[8] =
    {32767, 23170, 0, -23170, -32768, -23170, 0, 23170};
```

The API has another way to specify vector alignment on specific vector type, for example:

```
alignas(aie::vector_ldst_align_v<int16, 8>) const int16 reals[8] =
    {32767, 23170, 0, -23170, -32768, -23170, 0, 23170};
```

The API provides a global constant value (`aie::vector_decl_align`) that can be used to align the buffer to a boundary that works for any vector size.

```
alignas(aie::vector_decl_align) static cint16 my_buffer[8]={0,0},{1,-1},{2,-2},{3,-3},{4,-4},{5,-5},{6,-6},
{7,-7}};
```

Note: The alignment requirement on AI Engine is 16 bytes. This requirement might change with future architectures; AMD recommends that you use `aie::vector_decl_align` for portability.

Initialization

You can initialize vector registers with elements undefined as all zeros, using data from local memory or using part of the values set from another register, or a combination of the above.

```
// contents undefined
aie::vector<int32,8> uv;

// all 0's
aie::vector<int32,8> nv = aie::zeros<int32,8>();

// all values are 100
aie::vector<int32,8> bv = aie::broadcast<int32,8>(100);

// concatenate vectors into larger vector
aie::vector<int32,16> cv=aie::concat(nv,bv);
```

```

/*re-interpret "reals" as "aie::vector<int32,8>*" pointer and load value from it
*/
aie::vector<int32,8> iv = *(reinterpret_cast<aie::vector<int32,8>*>(reals));

// load value pointed by "reals"
aie::vector<int32,8> iv2 = aie::load_v<8>((int32*)reals);

// create a new 512-bit vector with lower 256-bit set with "iv"
aie::vector<int32,16> sv = iv.grow<16>(0);

```

In the previous example:

aie::zeros

Creates vector with all elements 0.

aie::broadcast

Sets all elements of vector to a value.

aie::concat

Concatenates multiple vectors or accumulators with same type and size into a larger vector or accumulator.

The vector and accumulator classes have the member function `grow()` that creates a larger vector where only one part is initialized with the subvector (`iv` in above example) and the other parts are undefined. Here the function parameter `idx` indicates the location of the subvector within the output vector.

The static keyword applies to the vector data type as well. When applied, the default value is zero when not initialized and the value is kept between graph run iterations.

Load and Store

Load and Store From Buffer Interface

AI Engine APIs provide access methods to read and write data from data memory, streaming data ports, and cascade streaming ports which can be used by AI Engine kernels. For additional details on stream APIs, see [Streaming Data API](#). For additional details on buffers, see [Input and Output Buffers](#).

In the following example, the const circular iterator is used to read an `aie::vector<cint16,8>` vector. Similarly, `readincr_v<8>(cin)` is used to read a sample of `int16` data from the `cin` stream. `writeincr(cas_out, v)` is used to write data to a cascade stream output.

```

void func(input_buffer<cint16> &din,
          input_stream<int16> *cin,
          output_cascade<cacc48> *cas_out) {
    auto cirIter=aie::cbegin_vector_circular<8,512>(din.data());
    aie::vector<cint16,8> data=*cirIter++;
    aie::vector<int16,8> coef=readincr_v<8>(cin);
    aie::accum<cacc48,4> v;
    ...
    writeincr(cas_out, v);
}

```

Load and Store Using Pointers

Applications can load from DM into vector registers and store the contents of vector registers into DM. Memory instructions in the AI Engine that operate on vectors have alignment requirements. Functions are provided for both aligned and unaligned accesses:

aie::load_v

Load a vector of `Elms` size whose elements have type `T` (for example, `aie::load_v<Elms>(T*)`). The pointer is assumed to meet the alignment requirements for a vector load of this size.

aie::store_v

Store a vector of `Elms` size whose elements have type `T` (for example, `aie::store_v<Elms>(T*)`). The pointer is assumed to meet the alignment requirements for a vector store of this size.

aie::load_unaligned_v

Load a vector of `Elms` size whose elements have type `T`. The pointer is assumed to be aligned to `T`.

aie::store_unaligned_v

Store a vector of `Elms` size whose elements have type `T`. The pointer is assumed to be aligned to `T`.

```

alignas(aie::vector_decl_align) int16 delay_value[N]={...};
aie::vector<int16,8> va=aie::load_v<8>(delay_value);
aie::store_v(delay_value,va);
aie::vector<int16,8> vv=aie::load_unaligned_v<8>((int16*)scatter_value);
aie::store_unaligned_v((int16*)scatter_value,vv);

```

The compiler supports standard pointer de-referencing and pointer arithmetic for vectors. For using vector iterators to access memory, see [Iterators](#).

It is mandatory to use the buffer port in the kernel function prototype as inputs and outputs. However, in the kernel code, it is possible to use a direct pointer to load/store data.

```

void func(input_buffer<int16> &w_input, output_buffer<cint16> &w_output){
    .....
    aie::vector<int16,16> datain=aie::load_v<16>((int16*)w_input.data());
    aie::vector<cint16,8> dataout=datain.cast_to<cint16>();
    aie::store_v((cint16*)w_output.data(),dataout);
    .....
}

```

The buffer structure is responsible for managing buffer locks tracking buffer type (ping/pong) and this can add to the cycle count. This is especially true when load/store are out-of-order (scatter-gather). Using pointers can help reduce the cycle count required for load and store.

 **Note:** If using pointers to load and store data, it is the designer's responsibility to avoid out-of-bound memory access.

Load and Store Using Streams

Vector data can also be loaded from or stored in streams as shown in the following example.

```

void func(input_stream<int32> *s0, ...){
    for(...){
        int32 data0=readincr(s0); //32 bits load
        aie::vector<int32,4> data1=readincr_v<4>(s0); //128 bits load
        ...
    }
}

```

Load and Store with Virtual Resource Annotations

The AI Engine is able to perform several vector load or store operations per cycle. However, for the load or store operations to be executed in parallel, they must target different memory banks. In general, the compiler tries to schedule many memory accesses in the same cycle when possible but there are some exceptions. Memory accesses coming from the same pointer are scheduled on different cycles. If the compiler schedules the operations on multiple variables or pointers in the same cycle, memory bank conflicts can occur.

To avoid concurrent access to a memory with multiple variables or pointers, most memory access functions in the AI Engine API accept an enum value from `aie_dm_resource` defined as follows:

```

enum class aie_dm_resource {
    none,
    a,
    b,
    c,
    d,
    stack
};

```

Also, the compiler provides the following `aie_dm_resource` annotations to annotate different virtual resources. Accesses using types that are associated with the same virtual resource are not scheduled to access the resource at the same cycle.

```

__aie_dm_resource_a
__aie_dm_resource_b
__aie_dm_resource_c
__aie_dm_resource_d
__aie_dm_resource_stack

```

The following example shows how to annotate memory access to bind individual access to virtual resources and allow or avoid accessing

memories at the same cycle.

```
int __aie_dm_resource_a *A;
int *B;

aie::vector<int,8> v1 = aie::load_v<8>(A);

/* Following access can be scheduled on the same cycle
 * as the access to A since B is not annotated.
 */
aie::vector<int,8> v2 = aie::load_v<8>(B);

/* Following specific access to B
 * is annotated with the same virtual resource as A
 * so they cannot be scheduled on the same cycle.
 */
aie::vector<int,8> v3 = aie::load_v<8, aie_dm_resource::a>(B);

/* vector iterator of B
 * annotated with the same virtual resource as A
 * so they cannot be scheduled on the same cycle.
 */
auto it = aie::begin_vector<8, aie_dm_resource::a>(B);
aie::vector<int,8> v4 = *(++it);
```

For example, the following code is to annotate two arrays to the same `__aie_dm_resource_a` that guides the compiler to not access them in the same instruction. It shows two ways to load vectors, using `aie::load_v` and iterators.

```
aie::vector<int32,8> va[32];
aie::vector<int32,8> vb[32];
int32 __aie_dm_resource_a* __restrict p_va = (int32 __aie_dm_resource_a*)va;
int32 __aie_dm_resource_a* __restrict p_vb = (int32 __aie_dm_resource_a*)vb;
auto it_b=aie::begin_vector<8>(p_vb);

// access va, vb by p_va, it_b
aie::vector<int32,8> vc;
vc=aie::load_v<8>(p_va)+*it_b;
p_va+=8;
++it_b;
```

Avoid adding resource annotation in the kernel function signature. The following code gives an example of declaring pointers with resource annotations:

```
void kernel_top(input_buffer<int32> & __restrict data1,
               input_buffer<int32>& __restrict data2, ...) {
    auto w_data1 = (int32 __aie_dm_resource_a* __restrict)data1.data();
    auto w_data2 = (int32 __aie_dm_resource_b* __restrict)data2.data();
    auto pv = aie::begin_vector<8>(w_data1);
    auto pv2 = aie::begin_vector<8>(w_data2);
    auto va = *pv++;
    auto vb = *pv2++;
    ...
}
```

The following code is to annotate an array and a buffer to the same `__aie_dm_resource_a` that guides the compiler to not access them in the same cycle.

```
alignas(aie::vector_decl_align) static int32 coeff[256]={...};
void func(input_buffer<int32> & __restrict wa, ..... )
{
    aie::vector<int32,8> v_coeff=aie::load_v<8>((int32 __aie_dm_resource_a*)coeff);
    int32 __aie_dm_resource_a* __restrict p_wa = (int32 __aie_dm_resource_a*)wa.data();
    auto waIter = aie::begin_vector<8>(p_wa);
    aie::vector<int32,8> va;
    va = *waIter;
```



```
...
}
```

Vector Arithmetic Operations

The AI Engine API supports basic arithmetic operations on two vectors, or on a scalar and a vector (operation on the scalar and each element of the vector). It also supports addition or subtraction of a scalar or a vector on an accumulator. Additionally, it supports multiply-accumulate (MAC). These operations include:

aie::mul

Returns an accumulator with the element-wise multiplication of two vectors, or the product of a vector and a scalar value.

aie::negmul

Returns an accumulator with the negative of the element-wise multiplication of two vectors, or the negative of the product of a vector and a scalar value.

aie::mac

Multiply-add on vectors (or scalar) and accumulator.

aie::msc

Multiply-sub on vectors (or scalar) and accumulator.

aie::add

Returns a vector with the element-wise addition of two vectors, or adds a scalar value to each component of a vector, or adds scalar or vector on accumulator.

aie::sub

Returns a vector with the element-wise subtraction of two vectors, or subtracts a scalar value from each component of a vector. Or subtract scalar or vector on accumulator.

aie::saturating_add

Returns a vector with the element-wise addition of two vectors, or adds a scalar value to each component of a vector. It supports saturation mode.

aie::saturating_sub

Returns a vector with the element-wise subtraction of two vectors, or subtract a scalar value from each component of a vector. It supports saturation mode.

The vectors and accumulator must have the same size and their types must be compatible. For example:

```
aie::vector<int32,8> va,vb;
aie::accum<acc64,8> vm=aie::mul(va,vb);
aie::accum<acc64,8> vm2=aie::mul((int32)10,vb);
aie::vector<int32,8> vsub=aie::sub(va,vb);
aie::vector<int32,8> vadd=aie::add(va,vb);

// vsub2[i]=va[i]-10
aie::vector<int32,8> vsub2=aie::sub(va,(int32)10);

// vsub2[i]=10+va[i]
aie::vector<int32,8> vadd2=aie::add((int32)10,va);

aie::accum<acc64,8> vsub_acc=aie::sub(vm,(int32)10);
aie::accum<acc64,8> vsub_acc2=aie::sub(vm,va);
aie::accum<acc64,8> vadd_acc=aie::add(vm,(int32)10);
aie::accum<acc64,8> vadd_acc2=aie::add(vm,vb);

aie::accum<acc64,8> vmac=aie::mac(vm,va,vb);
aie::accum<acc64,8> vmsc=aie::msc(vm,va,vb);

// scalar and vector can switch placement
aie::accum<acc64,8> vmac2=aie::mac(vm,va,(int32)10);

// scalar and vector can switch placement
aie::accum<acc64,8> vmsc2=aie::msc(vm,(int32)10,vb);
```

Following code shows the difference between `aie::add` and `aie::saturating_add` on vector addition when saturation happens.

```

aie::tile::current().set_saturation(aie::saturation_mode::saturate);

aie::vector<int16, 16> v1 = aie::broadcast<int16, 16>(20000);
aie::vector<int16, 16> v2 = aie::broadcast<int16, 16>(20000);
aie::vector<int16, 16> result1 = aie::add(v1, v2);
printf("vector + vector = %d\n", result1.get(0));
//output: vector + vector = -25536

aie::vector<int16, 16> result_sat = aie::saturating_add(v1, v2);
printf("vector + vector saturate= %d\n", result_sat.get(0));
//output: vector + vector saturate= 32767

```

The AI Engine API supports arithmetic operations on a vector or accumulation of element-wise square, including:

aie::abs

Computes the absolute value for each element in the given vector.

aie::abs_square

Computes the absolute square of each element in the given complex vector.

aie::conj

Computes the conjugate for each element in the given vector of complex elements.

aie::neg

For vectors with signed types, returns a vector whose elements are the same as in the given vector but with the sign flipped. If the input type is unsigned, the input vector is returned.

aie::mul_square

Returns an accumulator of the requested type with the element-wise square of the input vector.

aie::mac_square

Returns an accumulator with the addition of the given accumulator and the element-wise square of the input vector.

aie::msc_square

Returns an accumulator with the subtraction of the given accumulator and the element-wise square of the input vector.

The vector and the accumulator must have the same size and their types must be compatible. For example:

```

aie::vector<int16,16> va;
aie::vector<cint16,16> ca;
aie::vector<int16,16> va_abs=aie::abs(va);
aie::vector<int32,16> ca_abs=aie::abs_square(ca);
aie::vector<cint16,16> ca_conj=aie::conj(ca);
aie::vector<int16,16> va_neg=aie::neg(va);
aie::accum<acc32,16> va_sq=aie::mul_square(va);

aie::vector<int32,8> vc,vd;
aie::accum<acc64,8> vm=aie::mul(vc,vd);

// vmac3[i]=vm[i]+vc[i]*vc[i];
aie::accum<acc64,8> vmac3=aie::mac_square(vm,vc);

// vmac3[i]=vm[i]-vd[i]*vd[i];
aie::accum<acc64,8> vmac3=aie::msc_square(vm,vd);

```

Operands can also be supported pre-multiplication operations. On some AI Engine architectures certain operations can be collapsed with the multiplication into a single instruction. For example:

```

aie::vector<cint16,16> ca,cb;
aie::accum<cacc48,16> acc=aie::mul(aie::op_conj(ca),aie::op_conj(cb));

```

For details about pre-multiplication operations, see [Pre-Multiplication Operations](#).

Pre-Multiplication Operations

AI Engine architectures offer multiplication instructions that can perform additional operations on the input arguments. Instead of adding one variant for each possible combination, the AI Engine API offers types that can wrap an existing vector, accumulator of element reference and be passed into the multiplication function. Then the API merges the operations into a single instruction or apply the operation on the vector

before the multiplication, depending on the hardware support.

The pre-multiplication operations are special empty operations that simply return the original objects they wrap. These include:

- `op_abs`
- `op_add`
- `op_conj`
- `op_max`
- `op_min`
- `op_none`
- `op_sub`
- `op_sign`
- `op_zero`

The following example performs an element-wise multiplication of the absolute of vector `a` and the conjugate of vector `b`.

```
aie::accum<cacc48,16> foo(aie::vector<int16,16> a, aie::vector<int16,16> b){
    aie::accum<cacc48,16> ret;
    ret = aie::mul(aie::op_abs(a), aie::op_conj(b));
    return ret;
}
```

The following code performs accumulation with dynamic zeroization of the accumulator and dynamic sign of the data vectors:

```
alignas(aie::vector_decl_align) int16 data[16]={0,-1,2,-3,4,-5,6,-7,8,-9,10,-11,12,-13,14,-15};
aie::vector<int16,16> a=aie::load_v<16>(data);
aie::vector<int16,16> b=aie::load_v<16>(data);
aie::accum<acc48,16> ret;
bool is_zero=true;
bool is_sign=true;
ret = aie::mac(aie::op_zero(ret,is_zero),aie::op_sign(a,is_sign), aie::op_sign(b,is_sign));

// print the "ret" values
aie::print(ret,true,"ret=");
```

Vector Reduction

The AI Engine API supports vector reduce operations on `aie::vector`.

`aie::reduce_add`

Returns sum of the elements in the input vector.

`aie::reduce_add_v`

Returns the sums of the elements in the input vectors. The sum of each input vector is stored in an element of the output vector.

`aie::reduce_max`

Returns the element from the input vector with the largest value.

`aie::reduce_min`

Returns the element from the input vector with the smallest value.

`aie::reduce_mul`

Returns multiplication of the elements in the input vector.

Vector reduction examples are as follows.

```
aie::vector<int16,8> iv,iv2,iv3,iv4;
int16 iv_add=aie::reduce_add(iv);
aie::vector<int16,8> iv_add_v4=aie::reduce_add_v(iv,iv2,iv3,iv4);
int16 iv_max=aie::reduce_max(iv);
int16 iv_min=aie::reduce_min(iv);
int16 iv_mul=aie::reduce_mul(iv);
```

Bitwise Operations


The following AI Engine APIs allow bitwise AND, OR, XOR, and NOT operations on a scalar value and all the elements of the input vector with

same type, or on two input vectors with same type and size.

- `aie::bit_and`
- `aie::bit_or`
- `aie::bit_xor`

The following AI Engine API allows bitwise NOT operations on the elements of the input vector.

- `aie::bit_not`

 **Note:** The `aie::bit_*` operations are emulated on a scalar processor in the AI Engine.

```
aie::vector<int16,8> iv,iv2;

// 0xf bit AND with each component of iv
aie::vector<int16,8> iv_bit_and=aie::bit_and((int16)0xf,iv);
aie::vector<int16,8> iv_bit_or=aie::bit_or((int16)0xf,iv);
aie::vector<int16,8> iv_bit_xor=aie::bit_xor((int16)0xf,iv);

// bit AND on each component of iv and iv2
aie::vector<int16,8> iv_bit_and2=aie::bit_and(iv,iv2);
aie::vector<int16,8> iv_bit_or2=aie::bit_or(iv,iv2);
aie::vector<int16,8> iv_bit_xor2=aie::bit_xor(iv,iv2);
aie::vector<int16,8> iv_not=aie::bit_not(iv);
```

The following AI Engine APIs shift all components of a vector right or left by specified number of bits.

- `aie::downshift`: Shifts each component of a vector arithmetically (signed extended) right by specific bits.

```
aie::vector<int16,8> iv_down=aie::downshift(iv,2);
```

- `aie::upshift`: Shifts each component left by specific bits. Filled by 0.

```
aie::vector<int16,8> iv_up=aie::upshift(iv,2);
```

- `aie::logical_downshift`: Shifts each component of a vector logically (zero extended) right by specific bits.

```
aie::vector<int16,8> iv_logic_down=aie::logical_downshift(iv,2);
```

Data Comparison

AI Engine API provides vector comparison operations, including:

- `aie::eq`
- `aie::neq`
- `aie::le`
- `aie::lt`
- `aie::ge`
- `aie::gt`

The vector comparison operations compare two vectors element by element or compare one scalar with one vector, and return a special type `aie::mask`. Each bit of `aie::mask` corresponds to one elementary comparison result.

```
aie::vector<int32,16> v1,v2;

// compare each element
// true if v1[i]<v2[i]
aie::mask<16> msk_lt=aie::lt(v1,v2);

// set bit 0 to be true
msk_lt.set(0);

// set bit 1 to be false
msk_lt.clear(1);
```

```

/*element-wise selection
 *select v2[i] if msk_lt[i] is true
 */select v1[i] if msk_lt[i] is false;
aie::vector<int32,16> v_s=aie::select(v1,v2,msk_lt);

```

The following API compares two input vectors, or one input vector and one scalar value element by element, and returns the difference between them if the first input element is larger than the second input element. Otherwise, it returns 0 for that element.

- **aie::maxdiff**

```

aie::vector<int16,16> v1,v2;

// vc[i]=(v1[i]>v2[i])?(v1[i]-v2[i]):0
auto vc=aie::maxdiff(v1,v2);

// vc[i]=(v1[i]>1)?(v1[i]-1):0
auto vc2=aie::maxdiff(v1,(int16)1);

// vc[i]=(2>v2[i])?(2>v2[i]):0
auto vc3=aie::maxdiff((int16)2,v2);

```

The following APIs compare all the elements of the two input vectors and return a bool value.

aie::equal

Returns whether all the elements of the two input vectors are equal. The vectors must have the same type and size.

aie::not_equal

Returns whether some elements in the two input vectors are not equal. The vectors must have the same type and size.

The following APIs are provided to choose the max or min value of two vectors (or one scalar and one vector) element by element. The type of the scalar and vectors must be same.

- **aie::max**
- **aie::min**

```

// vc[i]=max(v1[i], v2[i])
aie::vector<int32,16> vc=aie::max(v1,v2);

```

Data Reshaping

The AI Engine API provides operations to change the location of the elements within a vector, to extract subvector from larger vector, or to combine the elements from two or more vectors.


The following APIs are provided to retrieve half of the vector by specific patterns.

aie::filter_even

Returns a vector of half the size by: out = { v[0:step-1], v[2*step:3*step-1], ... }

aie::filter_odd

Returns a vector of half the size by: out = { v[step:2*step-1], v[3*step:4*step-1], ... }

 **Note:** The parameter step must be a power of two.

```

aie::vector<int32,16> xbuff;

// first parameter is the vector value
// second parameter is the step value
aie::vector<int32,8> result_e=aie::filter_even(xbuff,1);
aie::vector<int32,8> result_o=aie::filter_odd(xbuff,4);

```

The following figures show how elements are chosen by **aie::filter_even** and **aie::filter_odd**.

Figure: aie::filter_even

```
aie::vector<int32,8> result_e=aie::filter_even(xbuff, /*step=*/1);
```

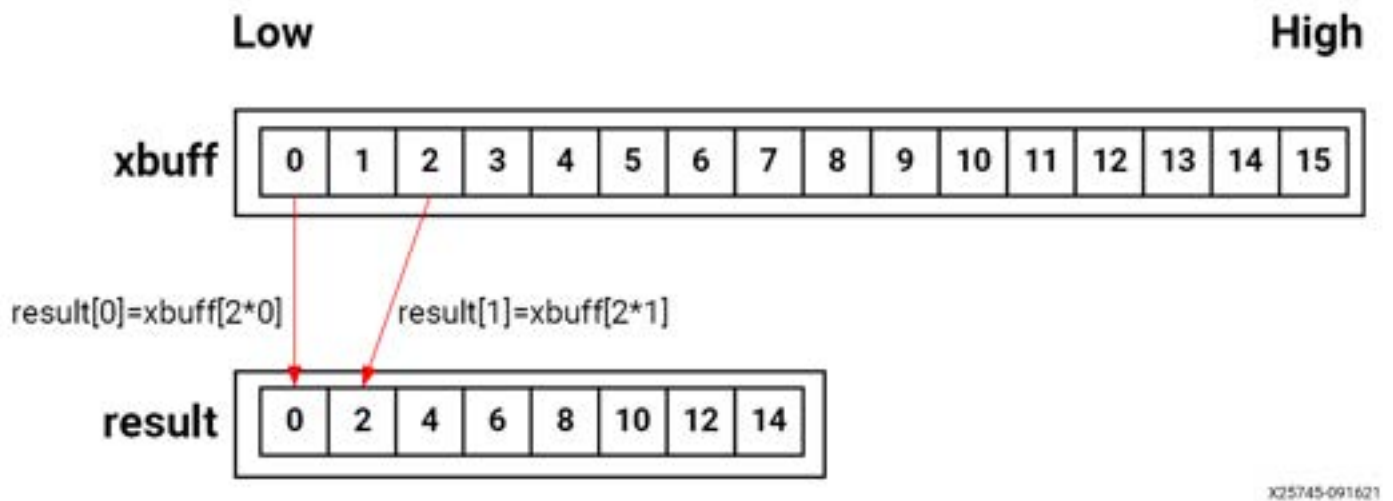
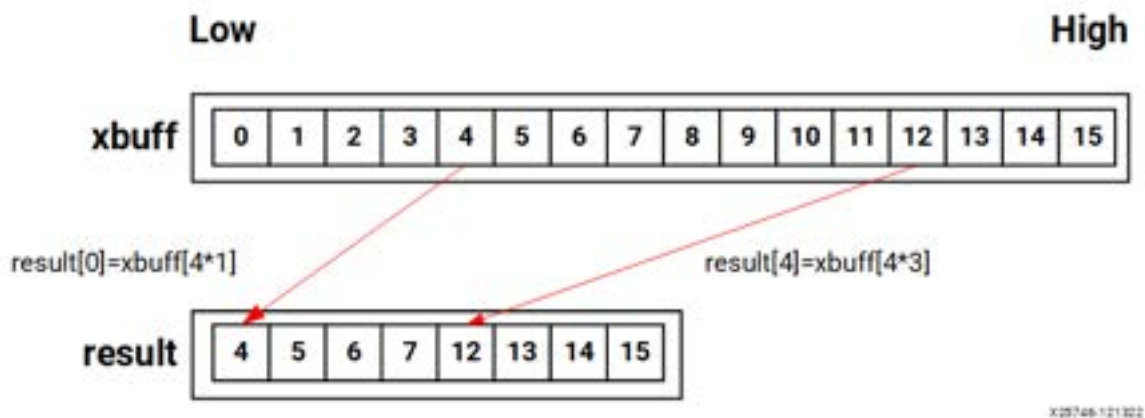


Figure: aie::filter_odd

```
aie::vector<int32,8> result_o=aie::filter_odd(xbuff, /*step=*/4);
```



`aie::select` can be used to choose elements from two vectors (or a scalar and a vector) to form a new vector. `aie::mask` is used for specifying the element to choose.

aie::select

Choose elements by each bit of mask value: `out[0] = {mask[0] == 0? a[0] : b[0], ...}`

```
aie::vector<int32,8> va,vb;
aie::mask<8> msk_value;
```

```
// vc[i]=(mask_value[i]==0)?va[i]:vb[i]
aie::vector<int32,8> vc=aie::select(va,vb,msk_value);
```

The AI Engine API provides following functions to shift vector elements by a specific number but maintain the vector size and element values.

aie::shuffle_down

Shift elements down by `n`: `out = {v[n], v[n+1], ..., v[Elms-1], undefined[0],...,undefined[n-1]}`. Higher elements of output(`out[n]`,`out[n+1]`,...) are undefined. "Elms" is the vector size.

aie::shuffle_up

Shift elements up by `n`: `out = {undefined[0],...,undefined[n-1],v[0], v[1], ...}`. Lower elements of output(`out[0]`,...,`out[n-1]`) are undefined.

aie::shuffle_down_rotate

Rotate elements down by `n`: `out = {v[n], v[n+1], ...,v[Elms-1], v[0], ...,v[n-1]}`.

aie::shuffle_up_rotate

Rotate elements up by n: out = {v[Elms-n], ..., v[Elms-1], v[0], ...,v[n-1]}.

aie::shuffle_down_fill

Shift elements down by n and fill with elements from a second vector: out = {v[n], v[n+1], ..., v[Elms-1], fill[0],...,fill[n-1]}. Higher elements of output(out[n],out[n+1],...) are filled with elements from second vector "fill"(fill[0],...,fill[n-1]).

aie::shuffle_up_fill

Shift elements up by n and fill with elements from a second vector: out = {fill[Elms-n],...,fill[Elms-1],v[0], v[1], ...}. Lower elements of output(out[0],...,out[n-1]) are filled with elements from second vector "fill"(fill[Elms-n],...,fill[Elms-1]).

```
aie::vector<int32,8> v,fill;

// v_d[0:4]=v[3:7]
// (v_d[5],v_d[6],v_d[7] are undefined)

auto v_d=aie::shuffle_down(v,3);

// v_u[3:7]=v[0:4]
// (v_u[0],v_u[1],v_u[2] are undefined)
auto v_u=aie::shuffle_up(v,3);

// v_d_r[0:4]=v[3:7],v_d_r[5:7]=v[0:2]
auto v_d_r=aie::shuffle_down_rotate(v,3);

// v_u_r[3:7]=v[0:4],v_u_r[0:2]=v[5:7]
auto v_u_r=aie::shuffle_up_rotate(v,3);

// v_d_fill[0:4]=v[3:7]
// v_d_fill[5:7]=fill[0:2]
auto v_d_fill=aie::shuffle_down_fill(v,fill,3);

// v_u_fill[3:7]=v[0:4]
// v_u_fill[0:2]=fill[5:7]
auto v_u_fill=aie::shuffle_up_fill(v,fill,3);
```

A vector can be reversed by:

aie::reverse

Reverse elements by: out = {v[Elms-1], ..., v[1], v[0]}

```
// v_rev[0:7]=v[7:0]
aie::vector<int32,8> v_rev=aie::reverse(v);
```


The AI Engine API provides functions to shuffle two input vectors and combine them into the larger output vectors. The two vectors must be the same type and size. The return type is `std::pair`.

aie::interleave_zip

Select and combine two vectors by this pattern and return a `std::pair` of vectors: out = { v1[0:step-1], v2[0:step-1], v1[step:2*step-1], v2[step:2*step-1], ... }

aie::interleave_unzip

Select and combine two vectors by this pattern and return a `std::pair` of vectors: out = { v1[0:step-1], v1[2*step:3*step-1], ..., v2[0:step-1], v2[2*step:3*step-1], ..., v1[step:2*step-1], v1[3*step:4*step-1], ..., v2[step:2*step-1], v2[3*step:4*step-1], ... }

 **Note:** The parameter `step` must be a power of two.

```
alignas(aie::vector_decl_align) int32 data[16]{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
aie::vector<int32,8> rva=aie::load_v<8>(data);
aie::vector<int32,8> rvb=aie::load_v<8>(data+8);
std::pair<aie::vector<int32,8>,aie::vector<int32,8>> rv=aie::interleave_zip(rva,rvb,4);
std::pair<aie::vector<int32,8>,aie::vector<int32,8>> rv2=aie::interleave_unzip(rva,rvb,2);

// rv.first=1 2 3 4 9 10 11 12
aie::print(rv.first,true,"rv.first=");

// rv.second=5 6 7 8 13 14 15 16
```

```
aie::print(rv.second,true,"rv.second=");

// rv2.first=1 2 5 6 9 10 13 14
aie::print(rv2.first,true,"rv2.first=");

// rv2.second=3 4 7 8 11 12 15 16
aie::print(rv2.second,true,"rv2.second=");
```

The following example takes two vectors with reals in `rva` and imaginary in `rvb` (with type `aie::vector<int32,8>`) and creates a new complex vector. It also shows how to extract real and imaginary parts back and compare them with the original values.

```
aie::vector<int32,8> rva,rvb;
auto rv=aie::interleave_zip(rva,rvb,1);
aie::vector<cint32,8> cv=aie::concat(rv.first.cast_to<cint32>(),rv.second.cast_to<cint32>());

auto [uva,uvb]=aie::interleave_unzip(rv.first,rv.second,1);

// return value as true
bool ret=aie::equal(rva,uva) && aie::equal(rvb,uvb);
```

The AI Engine API provides functions to transpose an input vector, which is interpreted as a matrix of the dimensions specified by Row and Col. The function returns a vector ordered as the transpose of the specified matrix. For Row or Col equals to 1, the input vector is returned.

- `aie::transpose`

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

// Row=4 & Col=4
auto va_t=aie::transpose(va,4,4);

// print va_t
// va_t=1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16
aie::print(va_t,true,"va_t=");
```

The AI Engine API provides functions to extract real or imaginary parts from complex scalar or vector data:

- `aie::real`: Gather and return the real part of a complex value or vector.
- `aie::imag`: Gather and return the imaginary part of a complex value or vector.

```
alignas(aie::vector_decl_align) int16 data[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
cint16 c1={1,2};
aie::vector<cint16,8> vc1=aie::load_v<8>((cint16*)data);
int16 c1_re=aie::real(c1);
aie::vector<int16,8> vc1_imag=aie::imag(vc1);
printf("c1_re=%d\n",c1_re);
//Example output: c1_re=1

aie::print(vc1_imag,true,"vc1_imag=");
//Example output: vc1_imag=2 4 6 8 10 12 14 16
```

Iterators

The AI Engine API provides iterators to access AI Engine memory. The AI Engine API provides iterators to iterate over both scalar and vector data types. In addition, it also provides circular iterators. The following table lists the different types of iterators provided by the AI Engine API.

Table: Iterators

Iterator	Description
<code>aie::begin</code>	Returns a scalar iterator.
<code>aie::cbegin</code>	Returns a scalar iterator with read-only access.

Iterator	Description
<code>aie::begin_circular</code>	Returns a scalar circular iterator. You can specify the circular buffer size.
<code>aie::cbegin_circular</code>	Returns a scalar iterator with read-only access.
<code>aie::begin_random_circular</code>	Returns a scalar circulator iterator with random access.
<code>aie::cbegin_random_circular</code>	Returns a scalar circulator iterator with random, read-only access.
<code>aie::begin_vector</code>	Returns a vector iterator. You can specify the size of the vector.
<code>aie::begin_restrict_vector</code>	Same as <code>aie::begin_vector</code> , but the return iterator is considered restrict.
<code>aie::cbegin_vector</code>	Returns a vector iterator with read-only access.
<code>aie::cbegin_restrict_vector</code>	Same as <code>aie::cbegin_vector</code> , but the return iterator is considered restrict.
<code>aie::begin_restrict_vector</code>	Returns a vector iterator that is a restrict type.
<code>aie::cbegin_restrict_vector</code>	Returns a vector iterator that is a restrict type with read-only access.
<code>aie::begin_vector_circular</code>	Returns a circular vector iterator. You can specify the elementary vector size and total circular buffer size.
<code>aie::cbegin_vector_circular</code>	Returns a circular vector iterator with read-only access.
<code>aie::begin_vector_random_circular</code>	Returns a circular iterator for the array described by the given address and size.
<code>aie::cbegin_vector_random_circular</code>	Returns a circular iterator for the array described by the given address and size, with read-only access.

These iterators can be divided into two categories: forward iterator and random access iterator. They all support the following:

- dereferencing `*`
- operator `++`
- operator `==`
- operator `!=`

Random access iterator supports more operators. For example, `aie::begin_vector` and `aie::begin_vector_random_circular` also support the following:

- operator `--`
- operator `+`
- operator `-`
- operator `+=`
- operator `-=`

For more information about the type of each iterator, see [Memory](#) in the *AI Engine API User Guide (UG1529)*.

One of the benefits of circular buffer is that it avoids out-of-bound memory access. Following are some examples of using iterator to access memory.

```
alignas(aie::vector_decl_align) int32 init_value[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
auto iIter = aie::begin(init_value);
for(int i=0;i<16;i++, iIter++){
    *iIter=*iIter+1;
}
```

```
alignas(aie::vector_decl_align) int32 init_value[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

// create vector iterator
// first value is [1,2,3,4]
auto pv=aie::begin_vector<4>(init_value);
```

```

// create const circular vector iterator
// first value is [1,2,3,4]
// total 16 elements
auto pv_c=aie::cbegin_vector_circular<4,16>(init_value);
for(;pv!=init_value+16;pv++){
    aie::vector<int32,4> buff=*pv;
    aie::print(*pv,true,"pv:");
}
for(int i=0;i<5;i++){

    // go back to start
    // if reach the end of the circular buffer
    aie::vector<int32,4> buff=*pv_c++;
}

// point to circular buffer size 16.
auto p=aie::begin_circular<16>(init_value);

for(int i=0;i<N;i++){

    // return back to init_value+0
    // if reaches init_value+16
    *p++=i;
}

__attribute__((noinline)) void pass_through(input_buffer<int> & __restrict in,
    output_buffer<int> & __restrict out
){
    auto inIter1=aie::cbegin_restrict_vector<8>(in);
    auto outIter=aie::begin_restrict_vector<8>(out);
    for(int i=0;i<32;i++) {
        *outIter++=*inIter++;
    }
}

```

Operator Overloading

The AI Engine API provides operator overloading for many operations. This feature can be used by including the header file `aie_api/operators.hpp` and using the `aie::operators` namespace. The included operators are as follows.

Operator +

Addition operator. It is equivalent to `aie::add`.

Operator +=

Addition assignment operator.

Operator -

Negation operator. It is equivalent to `aie::neg`.

Operator -

Subtraction operator. It is equivalent to `aie::sub`.

Operator -=

Subtraction assignment operator.

Operator !=

Not equal to comparison operator. It is equivalent to `aie::neq`.

Operator <

Less than comparison operator. It is equivalent to `aie::lt`.

Operator <=

Less than or equal comparison operator. It is equivalent to `aie::le`.

Operator ==

Equal to comparison operator. It is equivalent to `aie::eq`.

Operator >

Greater than comparison operator. It is equivalent to `aie::gt`.

Operator >=

Greater than or equal comparison operator. It is equivalent to `aie::ge`.

Operator <<

Bitwise left shift operator. It is equivalent to `aie::upshift`.

Operator >>

Bitwise right shift operator. It is equivalent to `aie::downshift`.

Operator &

Bitwise AND operation. It is equivalent to `aie::bit_and`.

Operator ^

Bitwise XOR operation. It is equivalent to `aie::bit_xor`.

Operator |

Bitwise OR operation. It is equivalent to `aie::bit_or`.

Operator ~

Bitwise NEG operation. It is equivalent to `aie::bit_not`.

An example code of using operators is as follows.

```
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>
#include <aie_api/operators.hpp>
using namespace aie::operators;

aie::vector<int32,8> va,vb;
aie::vector<int32,8> vadd=va+vb;
aie::vector<int32,8> vsub=va-vb;

// negation of each element
aie::vector<int32,8> vneg=-vb;
vadd+=vneg;
aie::print(va,true,"va=");
aie::print(vadd,true,"vadd=");

// compare each element of the vector correspondingly
auto msk_neq=(vadd!=va);

// mask bit equals 1 if vector element not equal
aie::print(msk_neq,true,"msk_neq=");

// bit not
auto vnot_va=~va;

// bit xor
auto vones=va ^ vnot_va;
aie::print(vones,true,"vones=");

// choose vadd if mask bit equals 0
auto vout=aie::select(vadd,vones,msk_neq);

aie::print(vout,true,"vout=");
```

One possible output of the previous code is as follows.

```
va=9 10 11 12 13 14 15 16
vadd=9 10 11 12 13 14 15 16
msk_neq=0 0 0 0 0 0 0 0
vones=-1 -1 -1 -1 -1 -1 -1 -1
```

```
vout=9 10 11 12 13 14 15 16
```

Floating-Point Operations

The scalar unit floating-point hardware support includes square root, inverse square root, inverse, absolute value, minimum, and maximum. It supports other floating-point operations through emulation. The `softfloat` library must be linked in for test benches and kernel code using emulation. For math library functions, the single precision float version must be used (for example, use `expf()` instead of `exp()`).

The AI Engine vector unit provides eight lanes of single-precision floating-point multiplication and accumulation. The unit reuses the vector register files and permute network of the fixed-point data path. Only one fixed-point or floating-point vector instruction per cycle can be performed.

Floating-point MACs have a latency of two-cycles, thus, using two accumulators in a ping-pong manner helps performance by allowing the compiler to schedule a MAC on each clock cycle.

```
aie::accum<accfloat,8> acc1=aie::zeros<accfloat,8>();
aie::accum<accfloat,8> acc2=aie::zeros<accfloat,8>();
aie::vector<float,8> va,vb;
auto ita=aie::begin_vector<8>(data1);
auto itb=aie::begin_vector<8>(data2);
auto ito=aie::begin(out);
for(int i=0;i<32;i++)
chess_prepare_for_pipelining
{
    va=*ita++;
    vb=*itb++;
    acc1=aie::mac(acc1,va,vb);
    va=*ita++;
    vb=*itb++;
    acc2=aie::mac(acc2,va,vb);
}
auto acc=aie::add(acc1,acc2);
auto sum=aie::reduce_add(acc.to_vector<float>(0));
*ito=(float)sum;
```

There is a scalar float divide function `aie::div`, but there is no divide vector function at this time. However, vector division can be implemented via an inverse and multiply as shown in the following example.

```
aie::vector<float,8> vf_div,vf1,vf1_inv,vf2;
vf1_inv=aie::inv(vf1);
vf_div=aie::mul(vf1_inv,vf2);
```

The following API functions support operations on a scalar or all elements of a vector.

- `aie::inv`
- `aie::sqrt`
- `aie::invsqrt`
- `aie::sin`
- `aie::cos`
- `aie::sincos`: Same as `sin` and `cos`, but performs both operations and returns a `std::pair` of vectors of result values. The first vector contains the sine values, the second contains the cosine values
- `aie::sincos_complex`: Same as `sincos`, but returns both values as the real and imaginary parts of a complex number (`cos` in the real part, `sin` in the imaginary part).

For `aie::sin`, `aie::cos`, and `aie::sincos`, the input can either be a float value in radians or an integer. The floating-point range is $[-\pi, \pi]$. Integer values are handled as a fixed-point input value in Q1.31 format scaled with $1/\pi$ (input value 2^{31} corresponds to π). In this case, only the upper 20 bits of the input value are used. According to input type, the returned value is either a float or a signed Q0.15 fixed-point format.

```
alignas(aie::vector_decl_align) static int16 dds_stored [16]={...};
aie::vector<cint16,8> dds=aie::load_v<8>((cint16*)dds_stored);
int32 phase_in;
auto [sin_,cos_] = aie::sincos(phase_in << 14) ;
cint16 scvalues={cos_,sin_};
dds.push(scvalues);
```

Multiple Lanes Multiplication - sliding_mul

AI Engine provides hardware support to accelerate a type of multiple lanes multiplication, called sliding multiplication. It allows multiple lanes to do MAC operations simultaneously, and the results are added to an accumulator. It especially works well with (but not limited to) finite impulse response (FIR) filter implementations.

These special multiplication structures or APIs are named `aie::sliding_mul*`. They accept coefficient and data inputs. Some variants of `aie::sliding_mul_sym*` allow pre-adding of the data input symmetrically before multiplication. These classes include:

- `aie::sliding_mul_ops`
- `aie::sliding_mul_x_ops`
- `aie::sliding_mul_y_ops`
- `aie::sliding_mul_xy_ops`
- `aie::sliding_mul_sym_ops`
- `aie::sliding_mul_sym_x_ops`
- `aie::sliding_mul_sym_y_ops`
- `aie::sliding_mul_sym_xy_ops`
- `aie::sliding_mul_sym_uct_ops`

For more information about these APIs and supported parameters, see the *AI Engine API User Guide* ([UG1529](#)).

For example, the `aie::sliding_mul_ops` class provides a parametrized multiplication that implements the following compute pattern.

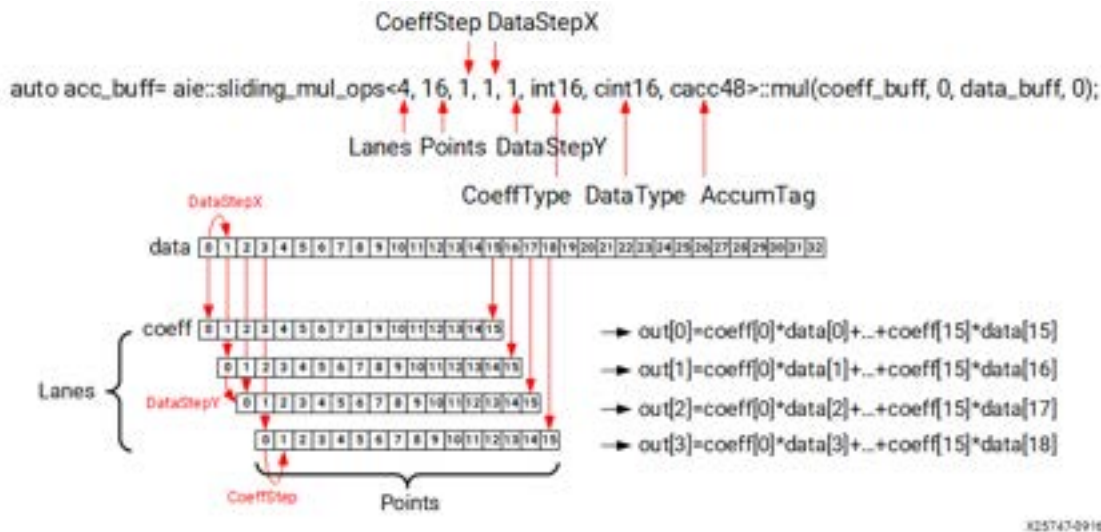
```
DSX = DataStepX
DSY = DataStepY
CS = CoeffStep
P = Points
L = Lanes
c_s = coeff_start
d_s = data_start
out[0] = coeff[c_s] * data[d_s + 0] + coeff[c_s + CS] * data[d_s + DSX] + ... + coeff[c_s + (P-1) * CS] *
data[d_s + (P-1) * DSX]
out[1] = coeff[c_s] * data[d_s + DSY] + coeff[c_s + CS] * data[d_s + DSY + DSX] + ... + coeff[c_s + (P-1) *
CS] * data[d_s + DSY + (P-1) * DSX]
...
out[L-1] = coeff[c_s] * data[d_s + (L-1) * DSY] + coeff[c_s + CS] * data[d_s + (L-1) * DSY + DSX] + ... +
coeff[c_s + (P-1) * CS] * data[d_s + (L-1) * DSY + (P-1) * DSX]
```

Table: Template Parameters

Parameter	Description
Lanes	Number of output elements.
Points	Number of data elements used to compute each lane.
CoeffStep	Step used to select elements from the coeff register. This step is applied to element selection within a lane.
DataStepX	Step used to select elements from the data register. This step is applied to element selection within a lane.
DataStepY	Step used to select elements from the data register. This step is applied to element selection across lanes.
CoeffType	Coefficient element type.
DataType	Data element type.
AccumTag	Accumulator tag that specifies the required accumulation bits. The class must be compatible with the result of the multiplication of the coefficient and data types (real/complex).

The following figure shows how to use the `aie::sliding_mul_ops` class and its member function, `mul`, to perform the sliding multiplication. It also shows how each parameter corresponds to the multiplication.

Figure: sliding_mul_ops Usage Example



Besides the `aie::sliding_mul*` classes, AI Engine API provides `aie::sliding_mul*` functions to do sliding multiplication and `aie::sliding_mac*` functions to do sliding multiplication and accumulation. These functions are simply helpers, that use the `aie::sliding_mul*_ops` classes internally and are provided for convenience. These include:

- `aie::sliding_mul`
- `aie::sliding_mac`
- `aie::sliding_mul_sym`
- `aie::sliding_mac_sym`
- `aie::sliding_mul_antisym`
- `aie::sliding_mac_antisym`
- `aie::sliding_mul_sym_uct`
- `aie::sliding_mac_sym_uct`
- `aie::sliding_mul_antisym_uct`
- `aie::sliding_mac_antisym_uct`

The following examples perform asymmetric sliding multiplications (template prototypes are in comments for quick reference).

```
constexpr unsigned Lanes = 8, Points = 8;
constexpr unsigned CoeffStep = 1;
constexpr unsigned DataStepX = 1, DataStepY = 1;
using CoeffType = int16;
using DataType = int16;
using AccumTag = acc48;

aie::vector<int16,16> va;
aie::vector<int16,64> vb0,vb1;
aie::accum<acc48,8> acc = aie::sliding_mul_ops<Lanes, Points, CoeffStep, DataStepX, DataStepY, CoeffType,
DataType, AccumTag>::mul(va, 0, vb0, 0);
acc = aie::sliding_mul_ops<Lanes, Points, CoeffStep, DataStepX, DataStepY, CoeffType, DataType,
AccumTag>::mac(acc, va, 8, vb1, 0);
auto vout=acc.to_vector<int32>(15);

constexpr unsigned coeff_start = 0;
constexpr unsigned data_start = 0;
aie::vector<int32,32> data_buff;
aie::vector<int32,8> coeff_buff;
aie::accum<acc80,8> acc_buff = aie::sliding_mul<Lanes, Points>(coeff_buff, coeff_start, data_buff,
data_start);
```

Following are symmetric sliding multiplication examples.

```
constexpr unsigned Lanes = 4, Points = 16;
constexpr unsigned CoeffStep = 1;
constexpr unsigned DataStepX = 1, DataStepY = 1;
using CoeffType = int16;
using DataType = cint16;
```

```
using AccumTag = cacc48;
constexpr unsigned coeff_start=0;
constexpr unsigned data_start=0;

aie::vector<cint16,16> data_buff;
aie::vector<int16,8> coeff_buff;
auto acc_buff = aie::sliding_mul_sym_ops<Lanes, Points, CoeffStep, DataStepX, DataStepY, CoeffType, DataType,
AccumTag>::mul_sym(coeff_buff, coeff_start, data_buff, data_start);

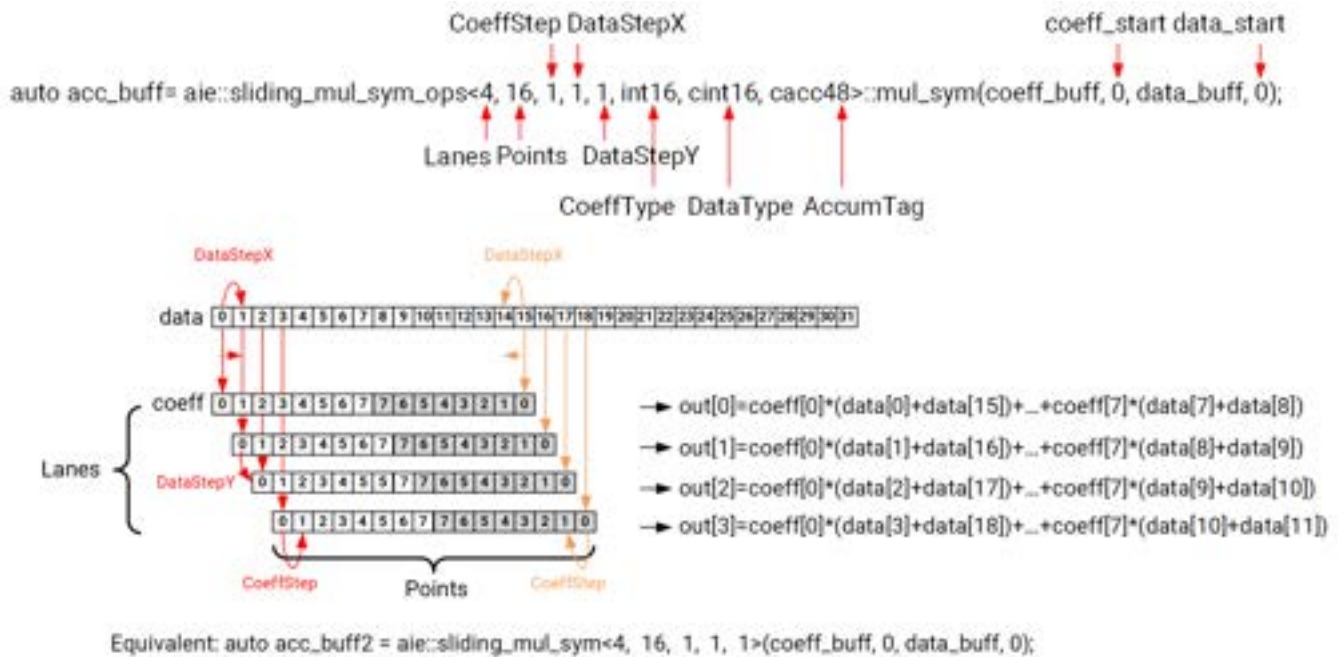
auto acc_buff2 = aie::sliding_mul_sym<Lanes, Points, CoeffStep, DataStepX, DataStepY>(coeff_buff,
coeff_start, data_buff, data_start);

constexpr unsigned ldata_start=0;
constexpr unsigned rdata_start=8;
aie::vector<cint16,16> ldata,rdata;
aie::vector<int16,8> coeff;
// symmetric sliding_mul using two data registers
auto acc = aie::sliding_mul_sym<Lanes, Points/2, CoeffStep, DataStepX, DataStepY>(coeff, coeff_start, ldata,
ldata_start, rdata, rdata_start);
```

Note: All registers in sliding multiplication must be considered circular. They go back to the start after they reach the end.

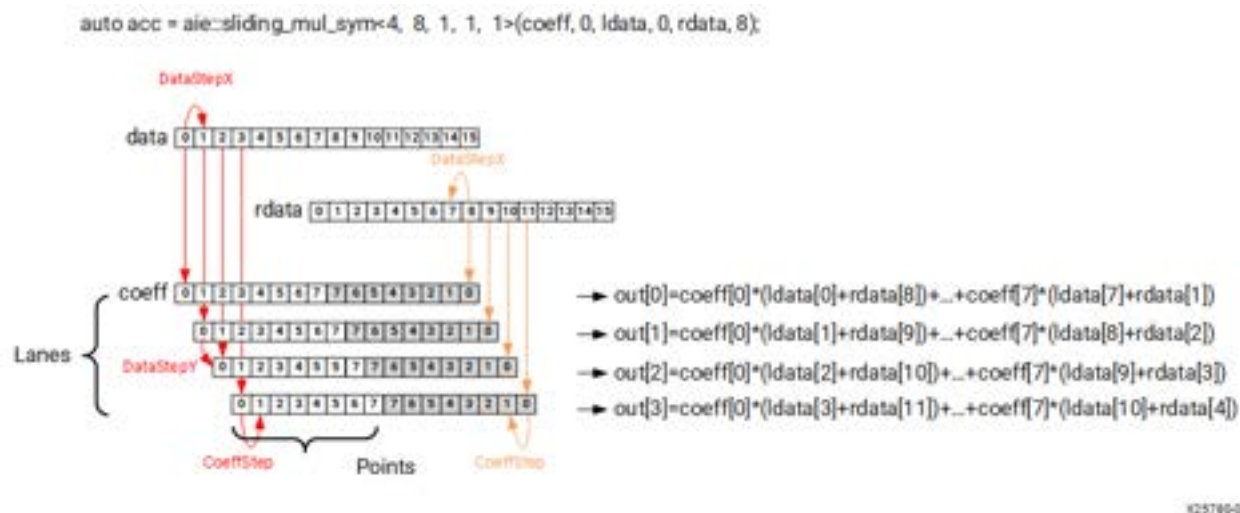
The following figures show how the previous symmetric examples are computed.

Figure: sliding_mul_sym_ops Usage Example



X25779-092921

Figure: sliding_mul_sym Function with Two Data Registers



Considerations When Using sliding_mul

Some restrictions include:

- Data width <=1024 bits, and Coefficient width <=256 bits
- Lanes * Points >= MACs per cycle for that type
- int8 is not supported in sliding_mul_sym_ops and sliding_mul_sym_uct_ops.

Matrix Multiplication - mmul

The AI Engine API encapsulates the matrix multiplication functionality in the `aie::mmul` class template. This class template is parametrized with the matrix multiplication shape ($M \times K \times N$), the data types and, optionally, the requested accumulation precision. For the supported shapes, see [Matrix Multiplication](#) in *AI Engine API User Guide* (UG1529).

It defines one function for the initial multiplication (`mul`) and one function for multiply-add (`mac`). `aie::mmul` objects can be initialized from vectors or accumulators so that they can be used in chained computations where partial results are sent over the cascade.

The resulting class defines a function that performs the multiplication and a data type for the result that can be converted to an accumulator/vector. The function interprets the input vectors as matrices as described by the shape parameters.

The following is a sample code to compute a $C(2 \times 64) = A(2 \times 8) * B(8 \times 64)$ matrix multiplication, using $2 \times 4 \times 8$ mode of `mmul`. One iteration of the loop does $C0(2 \times 8) = A0(2 \times 4) * B0(4 \times 8) + A1(2 \times 4) * B1(4 \times 8)$, where $A0$ is left half of A , $A1$ is right half of A , $B0$ is upper left 4×8 matrix of B , $B1$ is lower left 4×8 matrix of B , and $C0$ is leftmost 2×8 matrix of C .

The data for all matrices are assumed to be in row-major format in memory. Matrix A is read into a vector, per instructions. Thus, it requires some data filtering for `mmul`. $B0$ and $B1$ are read a row (eight elements) at a time. Four rows are combined for `mmul`. The indexes of two rows of $C0$ need to be calculated and two rows of $C0$ are written to memory separately.

Note: This example shows usage of `mmul`, which is not optimized for performance.

```
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include "aie_api/utils.hpp"

// For element mmul
const int M=2;
const int K=4;
const int N=8;

// Total matrix sizes
const int rowA=2;
const int colA=8;
const int colB=64;
const int SHIFT_BITS=0;
using namespace adf;
using MMUL = aie::mmul<M, K, N, int16, int16>;
void matmul_mmul(input_buffer<int16>& __restrict data0,
    input_buffer<int16>& __restrict data1, output_buffer<int16>& __restrict out){
    auto pa=aie::begin_vector<MMUL::size_A*2>(data0);
```



```

aie::vector<int16,MMUL::size_A*2> va=*pa;

// select left half matrix of A into va0
aie::vector<int16,MMUL::size_A> va0=aie::filter_even(va,4);

// select right half matrix of A into va1
aie::vector<int16,MMUL::size_A> va1=aie::filter_odd(va,4);

auto pb0=aie::begin_vector<8>(data1);
auto pb1=pb0+32;
aie::vector<int16,N> vb0_[4];
aie::vector<int16,N> vb1_[4];
aie::vector<int16,MMUL::size_C> vc;
auto pc=aie::begin_vector<8>(out);
for(int i=0;i<colB/N;i++)
chess_prepare_for_pipelining
{
    for(int j=0;j<4;j++){
        vb0_[j]=*pb0;
        pb0+=8;
        vb1_[j]=*pb1;
        pb1+=8;
    }
    MMUL m;
    m.mul(va0,aie::concat(vb0_[0],vb0_[1],vb0_[2],vb0_[3]));
    m.mac(va1,aie::concat(vb1_[0],vb1_[1],vb1_[2],vb1_[3]));
    vc=m.to_vector<int16>(SHIFT_BITS);//right shift SHIFT_BITS
    *pc=vc.extract<8>(0);
    pc+=8;
    *pc=vc.extract<8>(1);
    pc-=7;
    pb0-=31;
    pb1-=31;
}
}

```

API Operation Examples

The following example takes two vectors with reals in rva and imaginary in rvb (with type `aie::vector<int32,8>`) and creates a new complex vector, using the offsets to interleave the values as required.

```

aie::vector<int32,8> rva,rvb;
auto rv=aie::interleave_zip(rva,rvb,1);
aie::vector<cint32,8> cv=aie::concat(rv.first.cast_to<cint32>(),rv.second.cast_to<cint32>());

```

The following example shows how to extract real and imaginary portion of a vector cv with type `aie::vector<cint32,8>`.

```

aie::vector<cint32,8> cv;
aie::vector<int32,16> re_im=cv.cast_to<int32>();
aie::vector<int32,8> re=aie::filter_even(re_im,1);
aie::vector<int32,8> im=aie::filter_odd(re_im,1);

```

`aie::broadcast` can be used to set every element of a vector to a given value. The following example shows how to set all eight elements in a vector to a constant value.

```

// set all elements to 100
aie::vector<int32,8> v1=aie::broadcast<int32,8>(100);

```

The following example shows how to use `aie::broadcast` to set multiple values repeatedly in the vector.

```

alignas(aie::vector_decl_align) int16 init_data[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

// set 0 1 2 3 repeatedly into buff
aie::vector<int16,32> buff=aie::broadcast<cint32,8>(*(cint32*)init_data).cast_to<int16>();

```

The following example shows how to multiply each element in `rva` by the first element in `rvb`. This is efficient for a vector multiplied by constant value.

```
aie::vector<int32,8> rva,rvb;
aie::accum<acc80,8> acc = aie::mul(rva,rvb[0]);
```

The following examples show how to multiply each element in `rva` by its corresponding element in `rvb`.

```
aie::vector<int32,8> va,vb;
aie::accum<acc80,8> acc=aie::mul(va,vb);
```

The following examples show how to perform matrix multiplication for `int8 x int8` data types with `mmul` intrinsic, assuming that data storage is in row-major format.

```
// Z_{2x8} * X_{8x8} = A_{2x8}
aie::vector<int8,16> Z;
aie::vector<int8,64> X;
aie::mmul<2,8,8,int8,int8> m;
m.mul(Z,X);
```

```
// Z_{4x8} * X_{8x4} = A_{4x4}
aie::vector<int8,32> Z;
aie::vector<int8,32> X;
aie::mmul<4,8,4,int8,int8> m;
m.mul(Z,X);
```

For more information about vector lane permutations, refer to the *AI Engine Intrinsics User Guide* ([UG1078](#)).

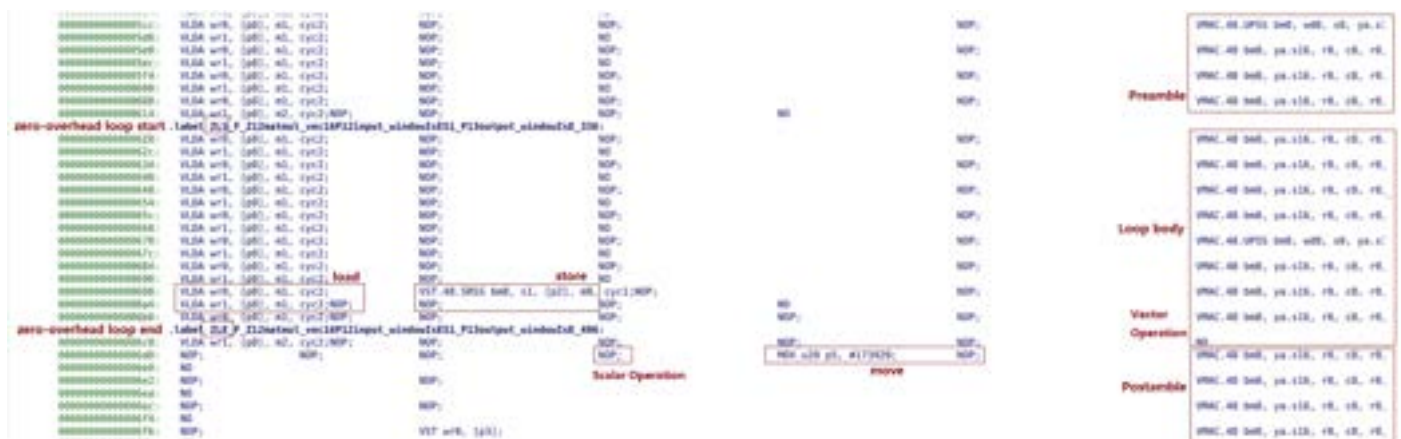
Kernel Optimization

Loops

The AI Engine has a zero-overhead loop structure that does not incur any branch control overhead for comparison and branching thus reducing the inner loop cycle count. Pipelining allows the compiler to add pre-amble and post-amble so that the instruction pipeline is always full during loop execution. With a pipelined loop, a new iteration can be started before the previous one ends to achieve higher throughput.

The following figure shows the assembly code of a zero-overhead loop. Two vector loads, one vector store, one scalar instruction, two data moves, and one vector instruction are shown in order in different slots.

Figure: Assembly Code of Zero-Overhead Loop



The following pragmas work together to direct the compiler to pipeline the loop and let it know that the loop will always be executed at least three times.

```
for (int i=0; i<N; i+=2)
    chess_prepare_for_pipelining
    chess_loop_range(3,)
```

The `chess_loop_range(<minimum>, <maximum>)` tells the compiler that the corresponding loop is executed at least `<minimum>` times,

and at most `<maximum>` times, where `<minimum>` and `<maximum>` are non-negative constant expressions, or can be omitted. When omitted, `<minimum>` defaults to 0, and `<maximum>` defaults to the maximum preset in the compiler. While `<maximum>` is not relevant for the pipeline implementation, `<minimum>` guides the pipeline implementation.

The `<minimum>` number defines how many loop iterations are executed at a minimum each time the loop is executed. The software pipeline is then tuned to allow at least that many iterations to execute in parallel if possible. It also determines that checking the boundaries for the loop is not necessary before the `<minimum>` number of iterations are executed.

The loop range pragma is not needed if the loop range is a compile time constant. In general, the AI Engine compiler reports the theoretical number best suited for optimum pipelining of an algorithm. If the range specification is not optimal, the compiler would issue a warning and suggest the optimal range. Towards that end, it is okay to initially set the `<minimum>` to one [`chess_loop_range(1,)`] and observe the theoretical best suited `<minimum>` being reported by the compiler.

```
Warning in "matmul_vec16.cc", line 10: (loop #39)
further loop software pipelining (to 4 cycles) is feasible with `chess_prepare_for_pipelining'
but requires a minimum loop count of 3
... consider annotating the loop with `chess_loop_range(3,)' if applicable,
... or remove the current `chess_loop_range(1,)` pragma
```

At this point, you can choose to update the `<minimum>` number to the reported optimum.

This second part of the pipeline implementation can be a reason for potential deadlocks in the AI Engine kernels if the actual `<minimum>` number of iterations is not reached. For this reason, you must ensure that the number of iterations is always at least the number specified in the `chess_loop_range` directive.

Loop carried dependencies impact the vectorization of code. If an inner loop dependency cannot be removed, one workaround would be to step out a level and manually unroll where there are (effectively) multiple copies of the inner loop running in parallel.

Try to avoid sequential load operations to fill a vector register completely before use. It is best to interleave loads with `aie::sliding_mul` functions, where the MAC and loads can be done in the same cycle.

```
buff.insert(3, readincr_v<4>(sig_in));
acc = aie::sliding_mul<4,8>(coe,0,buff,4);
writeincr(cascadeout,acc);
```

In certain use cases loop rotation, which rotates the instructions inside the loop, can be beneficial. Instead of loading data into a vector at the start of a loop, consider loading a block of data for the first iteration before the loop, and then for the next iteration near the end of the loop. This adds additional instructions but shortens the dependency length of the loop which helps to achieve an ideal loop with a potentially lower loop range.

```
// Load starting data for first iteration
aie::vector<cint16,16> buff = delay_line;

for (unsigned int i = 0; i < LSIZE; ++i)
chess_prepare_for_pipelining
chess_loop_range(4,)
{
    //template <unsigned Lanes, unsigned Points, int CoeffStep = 1, int DataStepX = 1, int DataStepY =
DataStepX, AccumElemBaseType AccumTag = accauto, Vector0r0p VecCoeff = void, Vector0r0p VecData = void>
    //auto sliding_mul (const VecCoeff &coeff, unsigned coeff_start, const VecData &data, unsigned data_start)
    buff.insert(2, readincr_v<4>(sig_in));
    acc = aie::sliding_mul<4,8>(coe,0,buff,0);
    writeincr(cascadeout,acc);

    buff.insert(3, readincr_v<4>(sig_in));
    acc = aie::sliding_mul<4,8>(coe,0,buff,4);
    writeincr(cascadeout,acc);

    buff.insert(0, readincr_v<4>(sig_in));
    acc = aie::sliding_mul<4,8>(coe,0,buff,8);
    writeincr(cascadeout,acc);

    buff.insert(1, readincr_v<4>(sig_in));
    acc = aie::sliding_mul<4,8>(coe,0,buff,12);
    writeincr(cascadeout,acc);
}
```

Loop Flattening and Unrolling

Loops can be flattened completely with the `chess_flatten_loop` pragma. This can be useful for small loops that are not optimally automated by the AI Engine compiler.

For loop flattening, the loop count can be determined by the compiler. In cases where the loop count cannot be determined by the tool automatically, you can set the loop count using the `chess_loop_count` pragma. For example:

```
for(int i=0;i<6;i++) chess_flatten_loop {...}
for(...) chess_loop_count(6) chess_flatten_loop {...}
```

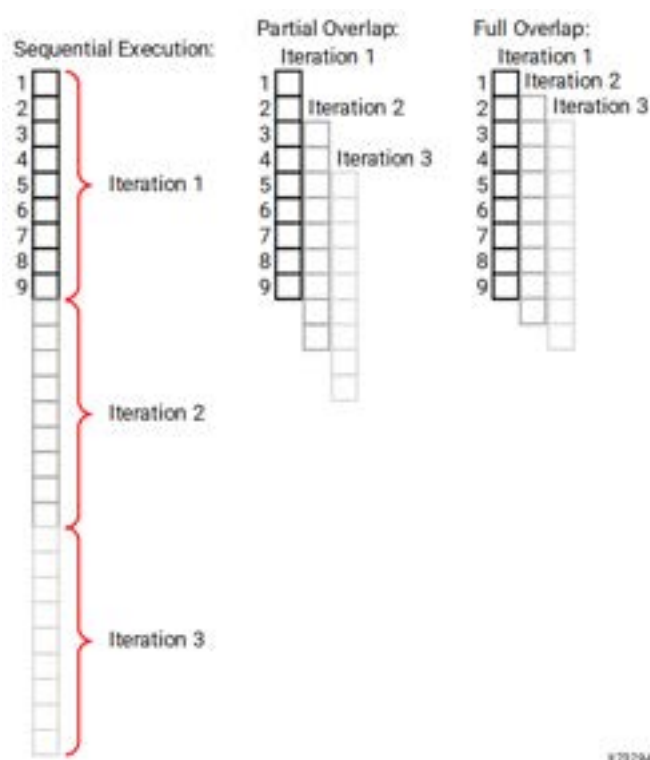
With `chess_unroll_loop(N)`, the loop body can be duplicated $N-1$ times, and the loop count is divided by N . The loop can also be completely unrolled by `chess_unroll_loop(*)`. The loop is unrolled and rewritten as a repeated sequence of similar independent statements.

With `chess_unroll_loop(N)`, an additional preamble loop is created. When the loop count is known at compile time, this preamble loop is fully unrolled. However, if the loop bound is not a compile-time constant but is guaranteed to be a multiple of N , use `chess_unroll_loop_assuming_multiple(N)` instead. This prevents the extra preamble loop, reducing program memory usage. Loop flattening is done in the final scheduling phase, such that the code generation, is still done based on the loop construct. Unlike loop flattening, loop unrolling duplicates iterations of code, and the duplicated codes can be compiled differently. This can be used to improve software pipelining of loops. But it can also pose a burden on scheduling when the unrolled loop count is large.

Software Pipelining of Loops

This section dives into software pipelining of loops. This is an important concept that enables the AI Engine to concurrently execute different parts of a program. For example, a loop that requires a total of nine cycles to execute through one iteration is shown in the following figure, where sequential execution all the way to a full overlap pipelining is illustrated.

Figure: Pipelining Example



Counting the cycles through each of these examples, it is clear that the sequential execution requires 27 cycles to fully execute the three loop iterations, while the partially overlapped pipeline requires 13 cycles, and the fully pipelined loop requires only 11 cycles. From a performance perspective, it is therefore desirable to have a fully overlapping pipeline. However, this is not always possible, because resource constraints, as well as inter-iteration loop dependencies can prevent a full overlap (see the following figure).

Figure: Dependencies in Pipelining



In this example, the program performs load A (2 x 256-bit) in cycle 2, load B (2 x 256-bit) in cycle 3, and in cycle 6 and 7 it executes operations on loop variable A. The remaining instructions of this iteration are of no importance with respect to the loop performance analysis. Cycles 2 and 3 of this loop iteration execute 4 x 256-bit load operations. The required four loads are executed in two cycles because the AI Engines can only execute two loads per cycle. This is called a resource constraint. If the loop containing this iteration is supposed to be pipelined, this constraint limits the overlap to no less than two cycles. Similarly, code dependencies between iterations shown in cycle 6 and 7 can prevent additional overlap. In this case, the next iteration of the loop requires the value of A to be updated before it can be used by the loop, thus, limiting the overlap.

The AI Engine compiler reports on each loop in the following form.

Note: The core compilation report can be found in `Work/aie/core_ID/core_ID.log` and the `-v` option is needed to generate the verbose report.

```
HW do-loop #397 in "testbench.cc", line 132: (loop #16) :
Critical cycle of length 2 : b67 -> b68 -> b67
Minimum length due to resources: 2
Scheduling HW do-loop #397
(algo 1a)      -> # cycles: 9
(modulo)      -> # cycles: 2 ok (required budget ratio: 1)
(resume algo)  -> after folding: 2 (folded over 4 iterations)
    -> HW do-loop #397 in "testbench.cc", line 132: (loop #16) : 2 cycles
NOTICE: loop #397 contains folded negative edges
NOTICE: postamble created
Removing chess_separator blocks (all)
```

In the AI Engine compiler report shown previously, the section `Critical cycle of length` provides feedback on code dependencies, while the `Minimum length due to resources` indicates minimum overlap requirement due to resource constraints. The `algo 1a` line states the total amount of cycles for a single iteration. Given these numbers, there are a maximum of five iterations active at a time creating the pipeline.

The AI Engine compiler reports these five overlapping iterations (the current iteration plus four folded iterations) in the `resume algo` line. In addition, it states the initiation interval (II), the number of cycles a single iteration has to execute before the following iteration is started, which is two in this example.

In general, it is sufficient to provide the directive `chess_prepare_for_pipelining` to instruct the compiler to attempt software pipelining. When the number of loop iterations is a compile time constant, the chess compiler creates the optimum software pipeline.

In the case of a dynamic loop range (defined by a variable start/end), the compiler requires additional information to create an effective pipeline loop structure. This is performed through the directive `chess_loop_range(<minimum>, <maximum>)`.

Note: If the number of cycles in the loop exceeds 64 cycles, pipelining can be disabled by the compiler for that loop. In this case, the compiler reports the following message.

```
(algo 1a)      -> # cycles: 167 (exceeds -k 64) -> no folding: 167
```

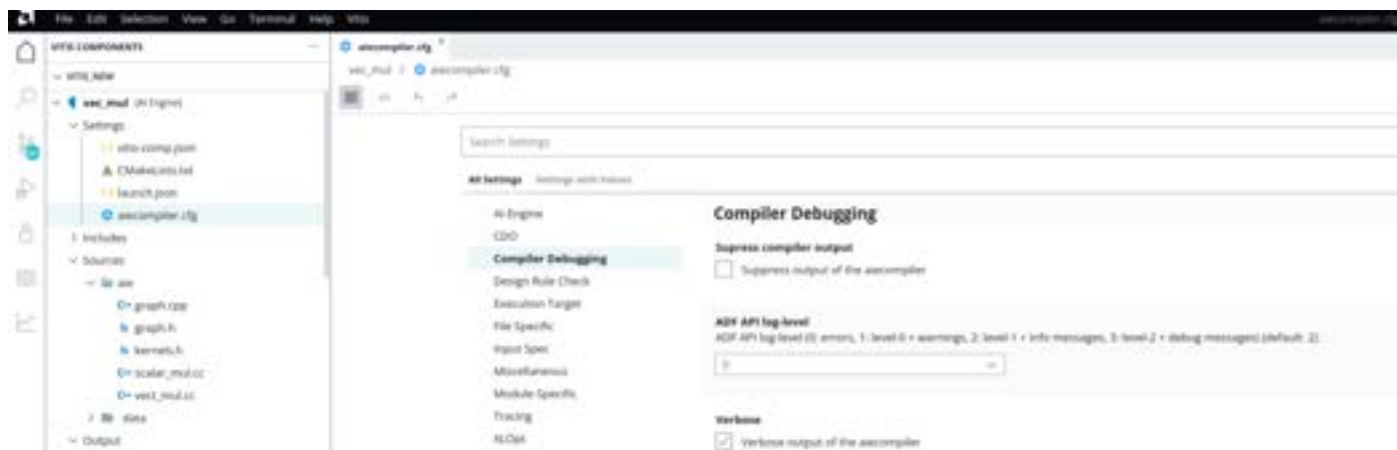
-> HW do-loop #511 in "xxxx", line 794: (loop #8): 167 cycles

To increase the threshold for loop pipelining, the following option can be added to AI Engine compiler:

```
--Xchess="main:backend.mist2.maxfoldk=200"
```

!! Important: To generate the detailed report, it is important to enable verbose mode in Vitis IDE as shown in the following figure or use the `--verbose` option at the command line, as in, `v++ -c --mode aie --aie.verbose`.

Figure: Enabling Verbose Mode in Debug Reports



A modulo scheduling report can be generated for modulo scheduled loops by specifying the option `--Xchess=main:backend.mist2.xargs=-ggraph` for the AI Engine compiler. The modulo scheduling report is available for software pipelined loop with the name `*_modulo.rpt` in `Work/aie/core_ID/Release/chesswork/<mangled_function_name>/*.rpt`, where `*` is the block name. The modulo scheduling report also contains the information about register live ranges for register files, which can be useful for finding inefficiencies in register assignment and can be improved by using `chess_storage`.

After compilation and linking is completed, you can open the compile log for an individual kernel in the Vitis IDE. For more information, see the *AI Engine Tools and Flows User Guide* (UG1076).

Related Information

[Loops](#)

Viewing Loop II in the Vitis IDE

When verbose (`-v`) mode of the AI Engine compiler is enabled, the Vitis IDE generates a summary table to show the pipeline results, with the following information:

- **TILE:** The AI Engine Tile coordinates of the kernel.
- **MIN II:** The minimum II that loop might try to achieve due to resources limitation.
- **ACTUAL II:** The actual II that the loop achieves.
- **SOURCE:** The loop source code.

An example loop II table is as follows:

Figure: Loop II



Chess Directives and C++ Attributes

In order to improve code readability, Chess loop directives can be written as C++ attributes in the AI Engine kernel.

Table: Chess Directive and Equivalent C++ Attribute

Chess Directive	C++ Attribute	Note
<code>chess_prepare_for_pipelining</code>	<code>[[chess::prepare_for_pipelining]]</code>	
<code>chess_loop_range(<minimum>, <maximum>)</code>	<code>[[chess::min_loop_count(<minimum>)]]</code> <code>[[chess::max_loop_count(<maximum>)]]</code>	In the Chess directive, the minimum and maximum loop ranges are specified in the same directive. In C++, two attributes are needed, one for minimum loop range and one for maximum loop range.
<code>chess_unroll_loop(N)</code>	<code>[[chess::unroll_loop]]</code>	Partial loop unrolling is not supported by the C++ attribute, hence N cannot be specified.
<code>chess_storage(<reg>)</code>	<code>[[chess::storage(<reg>)]]</code>	

Scheduling Separator

The AI Engine compiler can reorder data movements and expressions in a basic block. When data dependence is not seen by the compiler, or when you want to intentionally insert a sequential point in the code, the pragma `chess_separator_scheduler()` can be used to separate parts of the code, as follows:

```
func(...){
    //code block 1
    chess_separator_scheduler();
    //code block 2
}
```

The `chess_separator_scheduler(N)` pragma is another form of the `chess_separator_scheduler()` pragma, where N indicates additional N cycles between two blocks. N can be positive or negative. With a negative offset, you can allow a partial overlap (up to absolute N cycles) between two blocks.

For example, the compiler does not have knowledge about dependence between the different streams of the kernel. It can schedule different stream reads or writes in the same cycle. If any stream read or write stalls the kernel, it relies on an external source to supply or consume the data to or from the kernel. In the following example code, the stream write (to out) and read (from `receive_back`) can be scheduled in the same cycle.

```
void producer(output_stream<int32> *out, input_stream<int32> *receive_back){
    int32 data;
    ...
    writeincr(out,data); //schedule in the same cycle
    readincr(receive_back); //schedule in the same cycle
    ...
}
```

The above kernel will stall if there is no data read from stream `receive_back`. Thus, there will be no data sent to stream out. If an external source must receive data from producer before sending data to `receive_back`, the kernel stalls and cannot be recovered. To schedule the stream operations in different cycles, the `chess_separator_scheduler(N)` can be added, as follows:

```
void producer(output_stream<int32> *out, input_stream<int32> *receive_back){
    int32 data;
    ...
    writeincr(out,data);
    //Make sure read occurs after write and data is sent out before a possible stall
    chess_separator_scheduler(1);
    readincr(receive_back);
    ...
}
```

Parallel Streams Access

The AI Engine is able to use two streaming inputs or two streaming outputs in parallel.

To guide the tool to use parallel streams, use `aie_stream_resource_in` and `aie_stream_resource_out` annotations with different enumeration values, like `aie_stream_resource_in::a` and `aie_stream_resource_in::b` for input streams. For example:

```
void vect_mul(input_stream<int8>* __restrict data1, input_stream<int8>* __restrict data2,
              output_stream<int8>* __restrict out){
    while(true)
        chess_prepare_for_pipelining
        chess_loop_range(8,)
        {
            aie::vector<int8,16> va_int8=readincr_v<16,aie_stream_resource_in::a>(data1);
            aie::vector<int8,16> vb_int8=readincr_v<16,aie_stream_resource_in::b>(data2);
            auto vc=aie::mul(va_int8,vb_int8);

            // Avoid the write instruction to occur at the same cycle as the readincr of data1
            writeincr<aie_stream_resource_out::a>(out,vc.to_vector<int8>(0));

            va_int8=readincr_v<16,aie_stream_resource_in::a>(data1);
            vb_int8=readincr_v<16,aie_stream_resource_in::b>(data2);
            vc=aie::mul(va_int8,vb_int8);

            // This writeincr can be scheduled as soon as possible
            // as there is the __restrict keyword in function signature
            writeincr(out,vc.to_vector<int8>(0));
        }
}
```

Similarly, `aie_stream_resource_out::a` and `aie_stream_resource_out::b` can be used to denote two parallel output streams.

Recommended: When resource annotations are used on streams, use the annotation consistently on the same stream. Otherwise, the compiler scheduling might not follow the desired order of accesses with different annotations.

For example, the following stream operations are not recommended:

```
//input_stream<int32> * __restrict sin
.....
int32 a = readincr(sin); //access stream with none annotation
int32 b = readincr(sin);
v4int32 c_vect = readincr_v<4, aie_stream_resource_in::a>(sin); //access stream with different annotation
```

Instead, the following type of code can be used:

```
//input_stream<int32> * __restrict sin
.....
int32 a = readincr<aie_stream_resource_in::a>(sin); //access stream with annotation
int32 b = readincr<aie_stream_resource_in::a>(sin);
v4int32 c_vect = readincr_v<4, aie_stream_resource_in::a>(sin); //access stream with the same annotation
```

Restrict Keyword

The C++ standard provides a compiler extension `__restrict` for C standard pointer qualifier `restrict`, intended to allow more aggressive compiler optimization, by explicitly stating that no memory dependency will be caused by pointer aliasing. The compiler, by default, does not distinguish between different accesses of the same array. Thus, if an array is accessed in the pipeline, the compiler assumes that pointers might point to the same location, thereby incurring a higher interval between loops. This makes it essential in some situations to use a `__restrict` keyword to help guide the tool to achieve better performance. If a pointer is created with the `restrict` keyword, it is treated as a new object by the compiler. Pointers with the `restrict` keyword that point to the same location are treated independently by the compiler. The compiler can schedule the pointer access independently, which might impact the order of updates and cause undefined behavior. For detailed information about the concept of the `__restrict` keyword, see [Using the Restrict Keyword in AI Engine Kernels](#).

Inline Keywords

The AI Engine compiler supports the `always_inline` function attribute to eliminate function call overhead. An `inline` function is a function that is expanded in the line in which it is called. For larger functions, the overhead of the function call is insignificant compared to the amount of time the function takes to run. However, for smaller functions, the time needed to make the function call is greater than the time needed to execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

One advantage of using `inline` functions is that there is no function call overhead. Another potential advantage is that the compiler is able to perform context-specific optimizations which are not possible when the function stands in its own. The drawbacks are a larger register consumption, an increase of the size of the program, and the function cannot be profiled as there are no clear start and stop program counters to check.

The following keywords indicate the usage of the inlining functions to the compiler:

- `inline`: This keyword is an indicator that the AI Engine compiler is free to use an inline substitution or function call for any function marked `inline`.
- `inline __attribute__((always_inline))` or `__attribute__((always_inline))`: This forces the compiler to inline the function.
- `inline __attribute__((noinline))` or `__attribute__((noinline))`: This forces the compiler to always generate a function call.

Profiling Kernel Code

Every AI Engine has a 64-bit counter. The AI Engine API class `aie::tile` has method `cycles()` to read this counter value. For example:

```
aie::tile tile=aie::tile::current(); //get the tile of the kernel
unsigned long long time=tile.cycles();//cycle counter of the tile counter
```

The counter is continuously running. It is not limited by how many times you can read the counter. The value read back by the kernel can be written to memory, or it can be streamed out for further analysis. For example, to profile the latency of the code below, the counter value is read prior to the code being profiled, and again after the code has run:

```
aie::tile tile=aie::tile::current();
unsigned long long time1=tile.cycles(); //first time

for(...){...}

unsigned long long time2=tile.cycles(); //second time
long long time=time2-time1;
writeincr(out,time);
```

The latency of the loop in the kernel can then be examined in the host application by the second time minus the first time.

By comparing the data read back in between different executions of the kernel, or between different iterations of the loop, the data can be used to calculate latency. For example, the following code tries to get the latency of certain operations on an asynchronous buffer:

```
aie::tile tile=aie::tile::current();
for(...){//outer loop
    unsigned long long time=tile.cycles(); //read counter value
    writeincr(out,time);
    win_in.acquire();
    for(...){...} //inner loop
    win_in.release();
}
```

The latency of asynchronous buffer acquiring and release, plus the inner loop execution time can then be calculated by the second time minus the first time.

The counter value can also be written into data memory. The value can be read back by `printf` in simulation, or read back by host code in hardware. If the written value is not used by any other code, the `volatile` qualifier can be used to enforce the storage of the value of the counter. This qualifier ensures that the compiler optimizations do not eliminate this variable. For example:

```
static unsigned long long cycle_num[2];
aie::tile tile=aie::tile::current();
volatile unsigned long long *p_cycle=cycle_num;
*p_cycle=tile.cycles();//cycle_num[0]

for(...){...}

*(p_cycle+1)=tile.cycles();//cycle_num[1]
printf("cycles=%lld\n",cycle_num[1]-cycle_num[0]);
```

Using Vitis Unified IDE and Reports

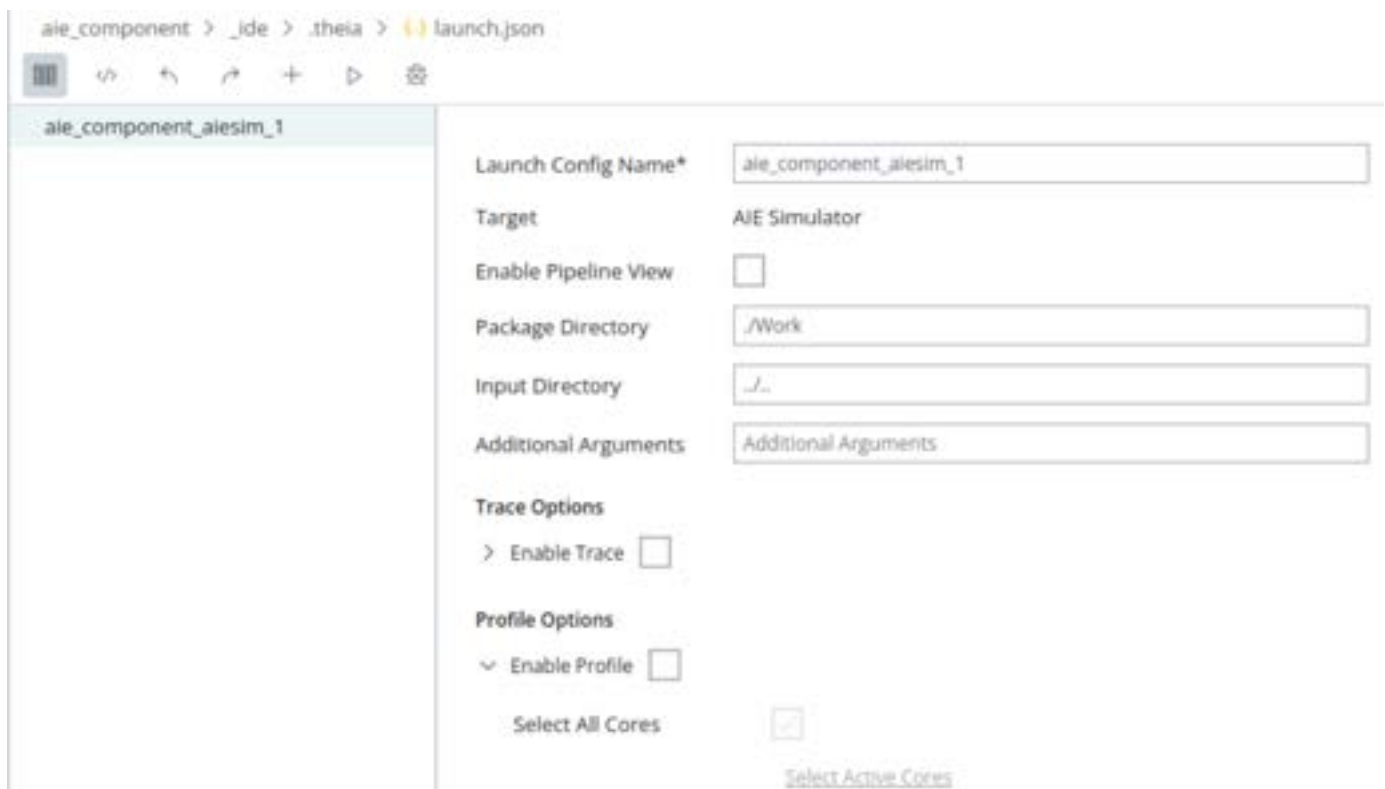
Vitis Unified IDE manages the AI Engine components and system project. The Vitis Unified IDE provides views for AI Engine kernel development, and it is essential in the kernel development, debugging, and analysis.

Vitis Unified IDE has a debug view which displays registers, variables, available breakpoints, variables to register/memory mapping, internal/external memory contents, Disassembly view, and Pipeline view for instructions.

When launching the simulation, if Enable Profile is selected in launch configurations, it will show the `printf` output in console. The Enable Trace check box in launch configurations is for generating event trace data which helps better understand when and how events such as memory stall and stream stall have occurred. Event trace is helpful in performance tuning.

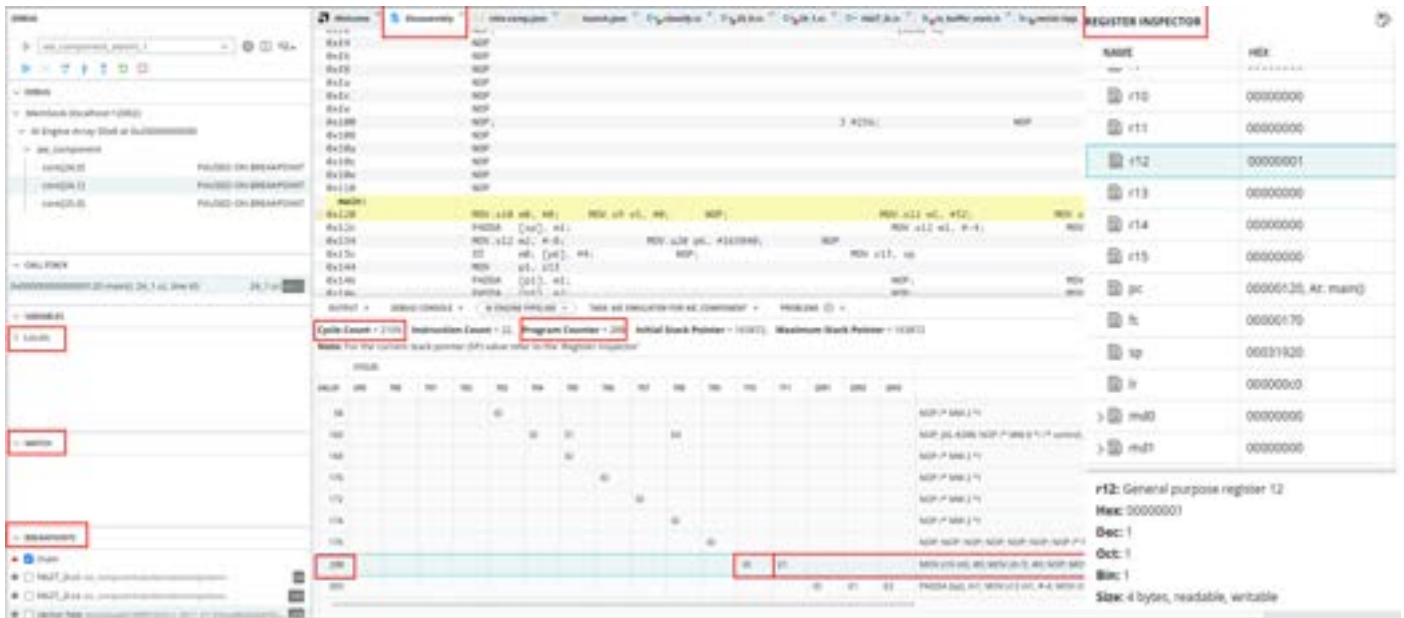
When launching the debug perspective, if Enable Pipeline View is selected in launch configurations, the AI ENGINE PIPELINE view can be shown.

Figure: Debug Configurations



In the debug perspective, debug commands enable you to resume, step into, and step over. The AI Engine source code is shown, and it is possible to set breakpoints by double-clicking lines. The windows Variables, Breakpoints, and Register Inspector are available to look into data memory or register status. The Disassembly view is helpful in understanding how instructions are used, especially how they are scheduled in the pipeline. The Pipeline view allows you to correlate instructions executed in a specific clock cycle with the labels in the Disassembly view.

Figure: Debug Perspective



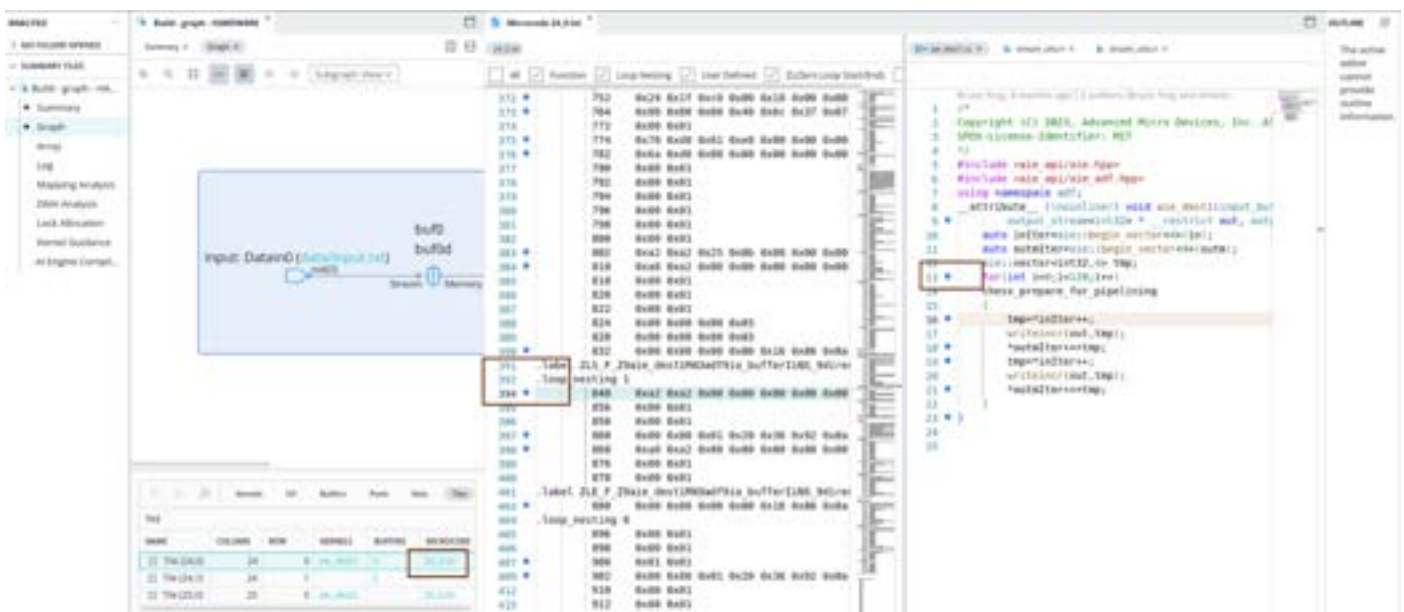
The generated code for an AI Engine (Col_Row.cc) includes the AI Engine kernels in the core and wrapper code. From the AI Engine wrapper code, you can step into the AI Engine kernel code by clicking multiple step-in buttons. Alternatively, you can open the AI Engine kernel source file from the design perspective, and set breakpoints in the file. Multiple views, such as the Disassembly view, Pipeline view, Memory Inspector, Register Inspector, and Variables view can be used for debug, and performance tuning.

Note: The number of breakpoints are limited to four for each tile. To set new breakpoints, beyond the number allowed, you must clear existing breakpoints. The tool will issue error messages if you try and set breakpoints beyond the number allowed.

The Disassembly view displays the compiler generated instruction target to the hardware. C/C++ source code can also be embedded between the lines for source code referencing. The instruction helps understand the compiled result, especially the loop pipelining result. By scrolling or stepping in the Disassembly view, the loop in the kernel can be found. The loop iterates from zero-overhead loop start (ZLS) to zero-overhead loop end (ZLE). It can be seen how load instructions and MAC instructions are placed to be pipelined. The preamble and postamble instructions are placed before and after the zero-overhead loop body to fill and flush the pipeline stages.

The Microcode view provides a static view of compiled kernel instructions. It can be opened via AI Engine components or links in Tile view in the analysis view in the Vitis IDE. By right-click the Microcode view, set the option to Enable AIE Cross Probing. Cross probing occurs between the Microcode view and source code.

Figure: Microcode View with Cross-Probing



Linker memory map reports for AI Engines cores can be found in Work/aie/core_ID/Release/core_id.map. This file lists the locations of the program and data memories by functions, static variables, and the software stack. From these reports, the stack size, program memory size, global buffers, and their sizes can be extracted. An XML version of the linker report (core_id.map.xml) can be generated by specifying the option `-Xchess=\"main:bridge.xargs=-fB\"` to the AI Engine compiler.

Work/<name>.aiecompile_summary is the compilation summary that can be opened in the Vitis IDE. Work/reports contains multiple reports for the graph compilation result such as, kernels and buffers mapping result. Refer to the *AI Engine Tools and Flows User Guide*

(UG1076) for additional information about AI Engine compiler outputs.

Interface Considerations

While single kernel programming focuses on vectorization of algorithm in a single AI Engine, multiple kernel programming considers several AI Engine kernels with data flowing between them.

The ADF graph can contain a single kernel or multiple kernels interacting with PS, PL, and global memory. Each AI Engine kernel has a runtime ratio. This number is computed as a ratio of the number of cycles taken by one invocation of a kernel (processing one block of data) to the cycle budget. The cycle budget for an application is typically fixed according to the expected data throughput and the block size being processed. The runtime ratio is specified as a constraint for every AI Engine kernel in the ADF graph.

The AI Engine compiler allocates multiple kernels into a single AI Engine if their combined total runtime ratio is less than one and multiple kernels fit in the AI Engine program memory, and if the total resource usage, like stream interface number, does not exceed the AI Engine tile limit. Alternatively, the compiler can allocate them into multiple AI Engines.

When programming for the AI Engine, it is important to note that each AI Engine has the capability to access two 32-bit AXI4-Stream inputs, two 32-bit AXI4-Stream outputs, one 384-bit cascade stream input, one 384-bit cascade stream output, two 256-bit data loads, and one 256-bit data store. However, due to the length of the instruction, not all of these operations can be performed during the same cycle.

To optimally use hardware resources, it is critical to understand the different methods available to transfer data between the ADF graph and PS, PL, and global memory, transfer data between kernels, balance the data movement, and minimize memory or stream stalls as much as possible which are covered in the following sections.

Data Movement Between AI Engines

Generally, there are two methods to transfer data between kernels—buffer or stream. When using the buffer, data transfers can be realized as ping-pong buffers or optionally, using a single buffer. The AI Engine tools will take care of buffer synchronization between the kernels.

Designers need to decide the buffer size and optionally buffer location between kernels when partitioning the application. If an overlap is needed between different buffers of the data, the AI Engine tools provide options for setting a margin for the buffer, that is, to copy the overlap of data automatically.

When using streams, the data movement involves two input as well as two output stream ports, along with one dedicated cascade stream input port and output port. Stream ports can provide 32-bit per cycle or, 128-bit per four cycles on each port. Stream interfaces are bidirectional and can read or write neighboring or non-neighboring AI Engines by stream ports. However, cascade stream ports are unidirectional and only provide a one-way access between the neighboring AI Engines.

Data Communication via AI Engine Data Memory

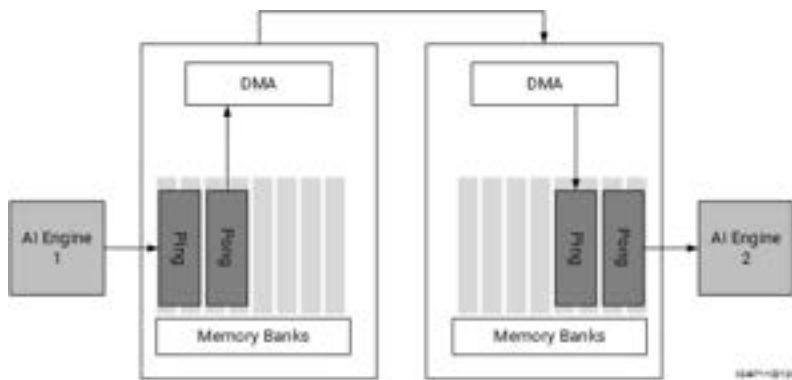
In the case where multiple kernels fit in a single AI Engine, communication between two or more consecutive kernels can be established using a common buffer in the local data memory of the AI Engine or in any of the three neighboring memories to which the AI Engine has direct access. In this case, only a single buffer is needed because the kernels execute one after another in a round-robin fashion.

For cases where the kernels are in separate but neighboring AI Engines, the communication can be carried out through the data memory module shared between the two neighboring AI Engine tile that use ping-pong buffers. These buffers can be on separate memory banks so access conflicts are avoided. The synchronization is done through locks. The input and output buffers for the AI Engine kernel are ensured to be ready by the locks associated with the buffers. In this type of communication, routing resources are saved and data transferring latency is eliminated because DMA and AXI4-Stream interconnect are not needed.

Data Communication via Memory and DMA

For non-neighboring AI Engines, similar communication can be established using the DMA in the memory module associated with each AI Engine. Ping-pong buffers in each memory module are used and synchronization is carried out with locks. There is increased communication latency as well as memory resources in comparison to shared memory communication.

Figure: Data Communication via Memory and DMA



Data Communication via AXI4-Stream Interconnect

AI Engines can directly communicate through the AXI4-Stream interconnect without any DMA and memory interaction. Data can be sent from one AI Engine to another or broadcast through the streaming interface. The data bandwidth of a streaming connection is 32-bit per cycle and built-in handshake and backpressure mechanisms are available.

The stream connection can be unicast or multicast. Note that in the case of multicast communication, the data is sent to all the destination ports at the same time and only when all destinations are ready to receive data.

Buffer vs. Stream in Data Communication

AI Engine kernels in the data flow graph operate on data streams that are infinitely long sequences of typed values. These data streams can be broken into separate blocks called buffers and processed by a kernel. Kernels consume input blocks of data and produce output blocks of data. An initialization function can be specified to run before the kernel starts processing input data. The kernel can read scalars or vectors from the memory, however, the valid vector length for each read and write operation must be multiple 128 bits. Buffers of input data and output data are locked for kernels before they are executed. Because the input data buffer needs to be filled with input data before the kernel can start, it increases latency compared to stream interface. The kernel can perform random access within a buffer of data and there is the ability to specify a margin for algorithms that require some number of bytes from the previous buffer.

Kernels can also access the data streams in a sample-by-sample fashion. Streams are used for continuous data and using blocking or non-blocking calls to read and write. Cascade stream only supports blocking access. The AI Engine supports two 32-bit stream input ports and two 32-bit stream output ports. Valid vector length for reading or writing data streams must be either 32 or 128 bits. Packet streams are useful when the number of independent data streams in the program exceeds the number of hardware stream channels or ports available.

A PLIO port attribute is used to make external stream connections that cross the AI Engine and PL boundary. The PLIO port can be connected to the AI Engine buffer via DMA S2MM or MM2S channels, or directly to AI Engine stream interfaces. Both of these connections (PL from/to buffer or stream) are limited by the stream interface of AI Engine tiles, that has a limit of 32 bits per cycle. However, for the buffer interface, the ping or pong buffer needs to be filled up before the kernel can start. Therefore, buffer interfaces from / to PL usually have a larger latency than a stream interface.

The following table summarizes the differences in buffer and stream connections between kernels.

Table: Buffer vs. Stream Connections

Connection	Margin	Packet Switching	Back Pressure	Lock	Max throughput by VLIW (per cycle)	Can be cast as a Source
Buffer	Yes	Yes	Yes ¹	Yes	2*256-bit load + 1*256-bit store	Yes
Stream	No	Yes	Yes	No	2*32-bit read + 2*32-bit write	Yes

1. Buffer back pressure, acquired or not, occurs on the whole buffer of data.

2. Considering all data are available at kernel boundary.

Graph code is C++ and available in a separate file from kernel source files. The compiler places the AI Engine kernels into the AI Engine array, taking care of the memory requirements and making all the necessary connections for data flow. Multiple kernels with low core usage can be placed into a single tile.

Free-Running AI Engine Kernel

The AI Engine kernel can be made to run continuously by using `graph::run(-1)`. This way the kernel will restart automatically after the last iteration is complete.

Note: `graph::run()` without an argument runs the AI Engine kernels for a previously specified number of iterations (which is infinity by

default if the graph is run without any arguments). If the graph is run with a finite number of iterations, for example, `mygraph.run(3)`; `mygraph.run()`; the second run call will also run for three iterations.

However, it requires input buffers and output buffers to be ready before it can start. Thus, it has a small overhead between kernel execution iterations. This section describes a method to construct a type of kernel that has zero overhead and runs forever. It is called the free-running AI Engine kernel.

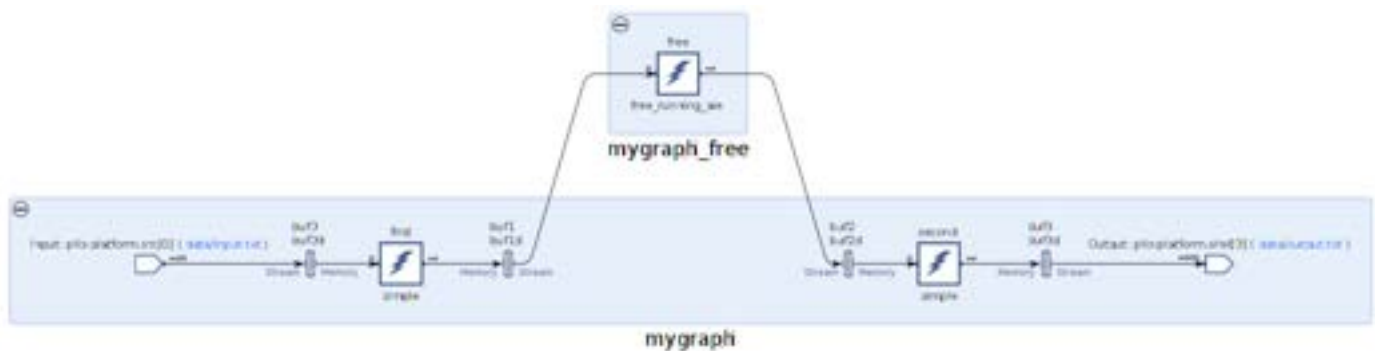
The free-running kernels can only have streaming interfaces. Loops with infinite iterations can be inside the kernel. For example:

```
void free_running_aie(input_stream<int32> *in, output_stream<int32> *out) {
    while(true){//for(;;) is acceptable for C++
        int32 tmp=readincr(in);
        chess_separator_scheduler();//make sure stream is flushed
        writeincr(out,tmp+1);
        chess_separator_scheduler();//make sure stream is flushed
    }
}
```

The free-running kernel must have its own graph defined. This graph must not have any other non-free-running kernels, because the graph never stops and non-free-running kernels will lose control after being started. The graph containing the free-running kernel must be a top-level graph that can be connected to other graphs, or it can be connected to PLIO or GMIO. A sample connection between the free-running graph and other graphs is shown as follows.

```
passingGraph mygraph;
freeGraph mygraph_free;
connect<> net0(mygraph.out1,mygraph_free.in);
connect<> net1(mygraph_free.out,mygraph.in2);
```

Figure: Free-Running Graph Connection



The free-running graph can be started using `mygraph_free.run(-1)` or automatically started after loading.

Runtime Parameter Specification

Using runtime parameters (RTP) is another way to pass data to the kernels. Two types of execution models for runtime parameters are supported.

1. Asynchronous parameters can be changed at any time by a controlling processor such as the Arm® processor. They are read each time a kernel is invoked. This means that the update of parameters occurs between different executions of the kernel, but it does not require the update to take place in a specific pattern. For example, these types of parameters are used as filter coefficients that change infrequently.
2. Synchronous parameters (triggering parameters) block a kernel from execution until these parameters are written by a controlling processor such as the Arm processor. Upon a write, the kernel reads the new updated value and executes once. After completion, it is blocked from executing until the parameter is updated again. This allows a different type of execution model from the normal streaming model, which can be useful for certain updating operations where blocking synchronization is important.

It is very important to understand that the RTP interaction between AI Engine kernels only happens in kernel execution boundaries. This means that the RTP output of the source kernel can only be read when the source kernel has completed its current iteration.

Note: RTP ports of AI Engine kernels will need to acquire lock and release lock before and after kernel execution, respectively. This will cause a small overhead for each kernel iteration. When thinking of partitioning the data into frames, the overhead must be taken into consideration according to system level performance requirements.

For more information about runtime parameter usage, refer to the *AI Engine Tools and Flows User Guide* (UG1076).

AI Engine and PL Kernels Data Communication

The AI Engine array interface contains modules to communicate between AI Engines and PL kernels using AXI4-Stream connections. Generally, PL interfaces produce or consume data through stream interfaces. Based on whether buffer or stream data is communicated by the AI Engine kernels, DMA and ping-pong buffers could be involved.

Note: PL kernels run at a lower frequency than AI Engine kernels. Data must cross the clock domains between the AI Engine clock and PL clock. The AMD Vitis™ environment handles the clock domain crossing (CDC) path automatically. It is recommended to run the PL kernel frequency as an integer factor of the AI Engine frequency, if possible. For instance, as ½ or ¼ of the AI Engine clock frequency.

DDR Memory Access through GMIO

The main data streams to and from the AI Engine are the AI Engine to PL streaming interface and GMIO, which is used to make external memory-mapped connections to or from the global memory. The interface between the PS and AI Engine has a low throughput and is ideal for configuration. The AI Engine-GMIO directly connects to the DDR memory through the AI Engine-NoC master unit (NMU).

The bandwidth of AI Engine GMIO is affected by the number of NMUs and DDR memory controllers used in the platform.

Related Information

[Configuring input_gmio/output_gmio](#)

Example Designs Using the AI Engine API

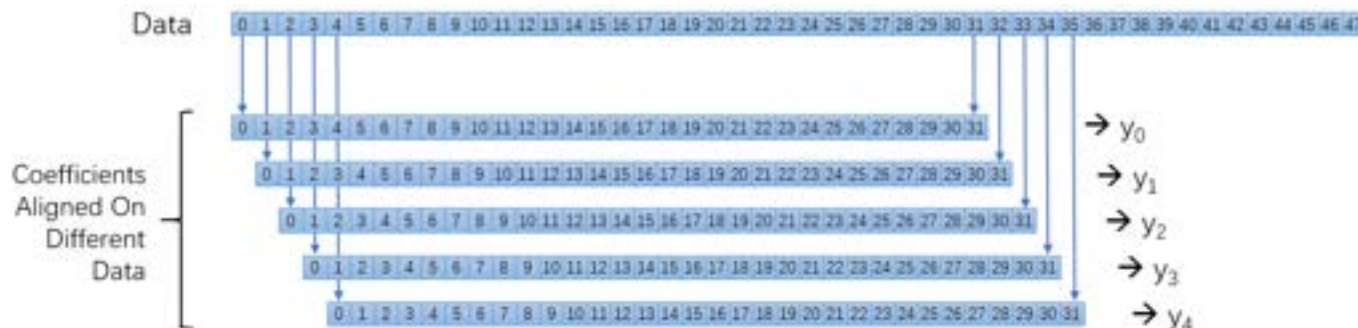
FIR Filter

Design Analysis

A finite impulse response (FIR) filter is described by the following equation, where x denotes the input, C denotes the coefficients, y denotes the output, and N denotes the length of the filter.

$$y_n = \sum_{k=0}^{N-1} C_k * x_{n+k}$$

Following is an example of a 32-tap filter.



Each output takes 32 multiplications. If you take `cint16` as the data type and coefficient type, it takes 4 cycles to compute a sample in a kernel, because each AI Engine can perform 8 MAC operations a cycle. If data is streaming from one stream port (32 bits), one data can produce one output (in the middle of processing).

So, the design is compute bound. You will see how to split the kernel into 4 cascaded kernels to process one sample per cycle.

Scalar Golden Reference

The AI Engine contains a scalar processor that can be used to implement scalar math operations, non-linear functions, and other general purpose operations. Sometimes it can be helpful to have a golden scalar reference version of code. But note that the scalar version of code takes much more time to run in simulation and hardware compared to the vectorized version.

The following provides example code for the scalar version of a 32-tap filter:

```
static cint16 eq_coef[32]={1,2},{3,4},...};

//keep margin data between different invocations of the graph
static cint16 delay_line[32];

__attribute__((noinline)) void fir_32tap_scalar(input_stream<cint16> * sig_in,
        output_stream<cint16> * sig_out){
```

```

for(int i=0;i<SAMPLES;i++){
    cycle_num[0]=tile.cycles();//cycle counter of the AI Engine tile
    cint64 sum={0,0};//larger data to mimic accumulator
    for(int j=0;j<32;j++){
        //auto integer promotion to prevent overflow
        sum.real+=delay_line[j].real*eq_coef[j].real-delay_line[j].imag*eq_coef[j].imag;
        sum.imag+=delay_line[j].real*eq_coef[j].imag+delay_line[j].imag*eq_coef[j].real;
    }
    sum=sum>>SHIFT;
    //produce one sample per loop iteration
    writeincr(sig_out,{(int16)sum.real,(int16)sum.imag});

    for(int j=0;j<32;j++){
        if(j==31){
            delay_line[j]=readincr(sig_in);
        }else{
            delay_line[j]=delay_line[j+1];
        }
    }
    cycle_num[1]=tile.cycles();//cycle counter of the AI Engine tile
    printf("cycle start=%d, cycle end=%d, total cycles=%d\n",cycle_num[0],cycle_num[1],(cycle_num[1]-
cycle_num[0]));

}

}

void fir_32tap_scalar_init()
{
    //initialize data
    for (int i=0;i<32;i++){
        int tmp=get_ss(0);
        delay_line[i]=*(cint16*)&tmp;
    }
};

```

Note:

- Function `fir_32tap_scalar_init` is used as an initialization function for the kernel, which will only be called once after `graph.run()`.
- Rounding and saturation modes are not supported in the scalar processor. They can be implemented via standard C operations, like shift.
- Tile counter is used for profiling the main loop of code.

From the profiling result, you can see that each sample (one sample per iteration) takes 2804 cycles. You can view the information under the Profile section in the Vitis IDE if you enable the option `--profile` during AI Engine simulation.

Note: The profiled cycle might vary when there are different compiler options, location constraints, etc. It might also vary between versions. But the concept of design analysis and performance analysis introduced here still applies.

For more information about graph construction and different kinds of profiling techniques, see [AI Engine Simulation-Based Performance Analysis](#) and [Performance Analysis of AI Engine Graph Application on Hardware](#) in *AI Engine Tools and Flows User Guide (UG1076)*.

Vectorized Version using a Single Kernel

The AI Engine naturally supports multiple lanes of MAC operations. For variations of FIR applications, the group of `aie::sliding_mul*` classes and functions introduced in [Multiple Lanes Multiplication - sliding_mul](#) can be used.

In this section, you will choose `aie::sliding_mul` and `aie::sliding_mac` functions with `Lanes=8` and `Points=8`. Both data and coefficient step sizes are 1, which is the default. For example, `acc = aie::sliding_mac<8,8>(acc,coe[1],0,buff,8)`; performs:

```

Lane 0: acc[0]=acc[0]+coe[1][0]*buff[8]+coe[1][1]*buff[9]+...+coe[1][7]*buff[15];
Lane 1: acc[1]=acc[1]+coe[1][0]*buff[9]+coe[1][1]*buff[10]+...+coe[1][7]*buff[16];
...
Lane 7: acc[7]=acc[7]+coe[1][0]*buff[15]+coe[1][1]*buff[16]+...+coe[1][7]*buff[22];

```

Notice that the data `buff` starts from different indexes in different lanes. It requires more than 8 samples (from `buff[8]` to `buff[22]`) to be ready before execution.

Since it has 32 taps, the FIR requires one `aie::sliding_mul<8,8>` operation and three `aie::sliding_mac<8,8>` operations to calculate eight lanes of output. The data buffer is updated from stream port by `buff.insert`.

The vectorized kernel code is as follows:

```
//keep margin data between different executions of graph
static aie::vector<cint16,32> delay_line;

alignas(aie::vector_decl_align) static cint16 eq_coef[32]={1,2},{3,4},...};

__attribute__((noinline)) void fir_32tap_vector(input_stream<cint16> * sig_in, output_stream<cint16> *
sig_out){
    const int LSIZE=(SAMPLES/32);
    aie::accum<cacc48,8> acc;
    const aie::vector<cint16,8> coe[4] =
{aie::load_v<8>(eq_coef),aie::load_v<8>(eq_coef+8),aie::load_v<8>(eq_coef+16),aie::load_v<8>(eq_coef+24)};
    aie::vector<cint16,32> buff=delay_line;
    for(int i=0;i<LSIZE;i++){
        //1st 8 samples
        acc = aie::sliding_mul<8,8>(coe[0],0,buff,0);
        acc = aie::sliding_mac<8,8>(acc,coe[1],0,buff,8);
        buff.insert(0,readincr_v<4>(sig_in));
        buff.insert(1,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<8,8>(acc,coe[2],0,buff,16);
        acc = aie::sliding_mac<8,8>(acc,coe[3],0,buff,24);
        writeincr(sig_out,acc.to_vector<cint16>(SHIFT));

        //2nd 8 samples
        acc = aie::sliding_mul<8,8>(coe[0],0,buff,8);
        acc = aie::sliding_mac<8,8>(acc,coe[1],0,buff,16);
        buff.insert(2,readincr_v<4>(sig_in));
        buff.insert(3,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<8,8>(acc,coe[2],0,buff,24);
        acc = aie::sliding_mac<8,8>(acc,coe[3],0,buff,0);
        writeincr(sig_out,acc.to_vector<cint16>(SHIFT));

        //3rd 8 samples
        acc = aie::sliding_mul<8,8>(coe[0],0,buff,16);
        acc = aie::sliding_mac<8,8>(acc,coe[1],0,buff,24);
        buff.insert(4,readincr_v<4>(sig_in));
        buff.insert(5,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<8,8>(acc,coe[2],0,buff,0);
        acc = aie::sliding_mac<8,8>(acc,coe[3],0,buff,8);
        writeincr(sig_out,acc.to_vector<cint16>(SHIFT));

        //4th 8 samples
        acc = aie::sliding_mul<8,8>(coe[0],0,buff,24);
        acc = aie::sliding_mac<8,8>(acc,coe[1],0,buff,0);
        buff.insert(6,readincr_v<4>(sig_in));
        buff.insert(7,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<8,8>(acc,coe[2],0,buff,8);
        acc = aie::sliding_mac<8,8>(acc,coe[3],0,buff,16);
        writeincr(sig_out,acc.to_vector<cint16>(SHIFT));
    }
    delay_line=buff;
}

void fir_32tap_vector_init()
{
    //initialize data
    for (int i=0;i<8;i++){
        aie::vector<int16,8> tmp=get_wss(0);
        delay_line.insert(i,tmp.cast_to<cint16>());
    }
};
```

- `alignas(aie::vector_decl_align)` can be used to ensure data is aligned for vector load and store.
- Each iteration of the main loop computes multiple samples. Consequently, the loop count is reduced.
- Data update, calculation and data write are interleaved in the code. Determining which portion of data buffer `buff` to read is controlled using `data_start` of `aie::sliding_mul`.
- For more information about supported data types and lane numbers for `aie::sliding_mul`, see *AI Engine API User Guide* (UG1529).

The initiation interval of the main loop should be identified. To locate the initiation interval of the loop:

1. Add the `-v` option to the AI Engine compiler to output a verbose report of kernel compilation.
2. Open the kernel compilation log, for example, `Work/aie/<COL_ROW>/<COL_ROW>.log`.
3. In the log, search keywords, such as `do-loop`, to find the initiation interval of the loop.

An example result follows:

```
HW do-loop #3651 in "/wrk/xcohdnobkup3/brucey/AIE_test_cases/ug1079/fir_32tap/fir_32tap_asym_1kernel/
aie/fir_32tap_vector.cc", line 21: (loop #3) :
critical cycle of length 121 : b428 -> b95 -> b96 -> b97 -> b98 -> b102 -> b115 -> b119 -> b120 -> b121
-> b122 -> b48 -> b49 -> b65 -> b73 -> b74 -> b75 -> b92 -> b99 -> b103 -> b104 -> b105 -> b106 -> b116
-> b123 -> b127 -> b128 -> b129 -> b130 -> b448 -> b215 -> b216 -> b217 -> b218 -> b50 -> b51 -> b175 ->
b176 -> b177 -> b178 -> b188 -> b195 -> b199 -> b200 -> b201 -> b202 -> b212 -> b219 -> b223 -> b224 ->
b225 -> b226 -> b236 -> b241 -> b67 -> b79 -> b80 -> b81 -> b93 -> b124 -> b131 -> b132 -> b133 -> b134
-> b52 -> b53 -> b155 -> b156 -> b157 -> b158 -> b165 -> b172 -> b179 -> b180 -> b181 -> b182 -> b419 ->
b69 -> b85 -> b86 -> b87 -> b498 -> b205 -> b206 -> b213 -> b220 -> b227 -> b228 -> b229 -> b230 -> b445
-> b111 -> b112 -> b113 -> b114 -> b118 -> b125 -> b135 -> b136 -> b137 -> b138 -> b54 -> b55 -> b159 ->
b160 -> b161 -> b162 -> b166 -> b173 -> b183 -> b184 -> b185 -> b186 -> b190 -> b221 -> b231 -> b232 ->
b233 -> b234 -> b428
minimum length due to resources: 128
scheduling HW do-loop #3651
(algo 2)      -> # cycles: 173 (exceeds -k 110)      -> no folding: 173
-> HW do-loop #3651 in "/proj/xbuils/SWIP/2025.1_0427_1909/installs/lin64/2025.1/Vitis/aietools/
include/adf/stream/me/stream_utils.h", line 258: (loop #3) : 173 cycles
```

where:

- The initiation interval of the loop is 173. This means that a sample is produced in roughly $173/32 \approx 5.4$ cycles.
 - The message `(exceeds -k 110) -> no folding` indicates that the scheduler is not attempting software pipelining because the loop cycle count exceeds a limit.
4. To override the loop cycle limit, add a user constraint, such as `--Xchess="fir_32tap_vector:backend.mist2.maxfoldk=200"` to the AI Engine compiler.

The example result is then as follows:

```
(resume algo)      -> after folding: 156 (folded over 1 iterations)
-> HW do-loop #3651 in "/proj/xbuils/SWIP/2025.1_0427_1909/installs/lin64/2025.1/Vitis/aietools/
include/adf/stream/me/stream_utils.h", line 258: (loop #3) : 156 cycles
```

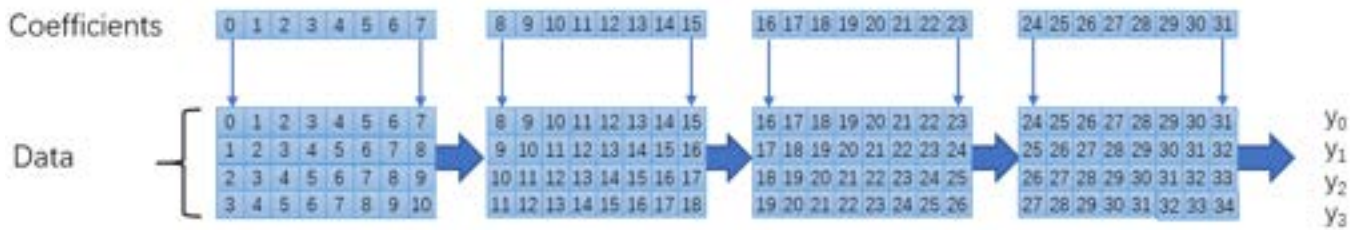
where, the software requires approximately $156/32=4.9$ cycles to produce a sample.

Note: The exact number of cycles might fluctuate slightly based on the specific compiler settings and the version of the compiler being used. However, the analysis techniques described in this section remain relevant and applicable regardless of these variations.

Vectorized Version using Multiple Kernels

With one input data per cycle, as estimated in [Design Analysis](#), you must run four kernels in parallel to get the best throughput. One such solution is to divide 32 coefficients into four kernels, where data is broadcast to four kernels. The partial accumulation results from kernels can be cascaded to produce result in the last kernel. The implementation is shown in the following figure:

Figure: Data Broadcast to 4 Kernels



Note that part of the initial data is discarded in subsequent kernels. For example, the second kernel will discard the first eight inputs. Four lanes and eight points are chosen for `aie::sliding_mul`. Data reads and writes are interleaved with computation. The first kernel code is as follows:

```
alignas(aie::vector_decl_align) static cint16 eq_coef0[8]={1,2},{3,4},...};

//For storing data between graph iterations
static aie::vector<cint16,16> delay_line;
__attribute__((noinline)) void fir_32tap_core0(input_stream<cint16> * sig_in,
        output_cascade<cacc48> * cascadeout){
    const cint16_t * restrict coeff = eq_coef0;
    const aie::vector<cint16,8> coe = aie::load_v<8>(coeff);

    aie::vector<cint16,16> buff = delay_line;
    aie::accum<cacc48,4> acc;
    const unsigned LSIZE = (SAMPLES/4/4); // assuming samples is integer power of 2 and greater than 16

    main_loop:for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    {
        //8 MAC produce 4 partial output
        buff.insert(2,readincr_v<4>(sig_in));
        acc = aie::sliding_mul<4,8>(coe,0,buff,0);
        writeincr(cascadeout,acc);

        //8 MAC produce 4 partial output
        buff.insert(3,readincr_v<4>(sig_in));
        acc = aie::sliding_mul<4,8>(coe,0,buff,4);
        writeincr(cascadeout,acc);

        buff.insert(0,readincr_v<4>(sig_in));
        acc = aie::sliding_mul<4,8>(coe,0,buff,8);
        writeincr(cascadeout,acc);

        buff.insert(1,readincr_v<4>(sig_in));
        acc = aie::sliding_mul<4,8>(coe,0,buff,12);
        writeincr(cascadeout,acc);
    }
    delay_line = buff;
}

void fir_32tap_core0_init(){
    // Drop samples if not first block
    int const Delay = 0;
    for (int i = 0; i < Delay; ++i){
        get_ss(0);
    }
    //initialize data
    for (int i=0;i<8;i++){
        int tmp=get_ss(0);
        delay_line.set(*(cint16*)&tmp,i);
    }
};
```

Note:

- `__attribute__((noinline))` is optional to keep function hierarchy.
- `chess_prepare_for_pipelining` is optional as the tools can do automatic pipelining.
- Each `aie::sliding_mul<4,8>` is multiplying four lanes eight points MAC and the partial result is sent through the cascade chain to the next kernel.
- Data buff is read starting from `data_start` parameter of `aie::sliding_mul`. The kernel code goes back to the beginning when it reaches the end in a circular fashion.

The compilation report can be found in `Work/aie/<COL_ROW>/<COL_ROW>.log` and the `-v` option is needed to generate the verbose report. In the log, search keywords, such as `do-loop`, to find the initiation interval of the loop. In the following example log file, you can see that the initiating interval of the loop is 16:

```
(resume algo) -> after folding: 16 (folded over 1 iterations)
-> HW do-loop #128 in ".../Vitis/<VERSION>/aietools/include/adf/stream/me/stream_utils.h", line 1192: (loop
#3) : 16 cycles
```

★ **Tip:** Get the last reported cycle for the loop in the log file.

The kernel code above takes roughly $16 \text{ (cycles)} / 16 \text{ (partial results)} = 1 \text{ cycle}$ to produce a partial output.

The other three kernels are similar. The second kernel code is as follows:

```
alignas(aie::vector_decl_align) static cint16 eq_coef2[8]={17,18},{19,20},...};

//For storing data between graph iterations
alignas(aie::vector_decl_align) static aie::vector<cint16,16> delay_line;

__attribute__((noinline)) void fir_32tap_core1(input_stream<cint16> * sig_in, input_cascade<cacc48> *
cascadein,
        output_cascade<cacc48> * cascadeout){
    const aie::vector<cint16,8> coe = aie::load_v<8>(eq_coef1);
    aie::vector<cint16,16> buff = delay_line;
    aie::accum<cacc48,4> acc;
    const unsigned LSIZE = (SAMPLES/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
        chess_prepare_for_pipelining
        {
            //8 MAC produce 4 partial output
            acc = readincr_v4(cascadein);
            buff.insert(2,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,0);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(3,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,4);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(0,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,8);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(1,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,12);
            writeincr(cascadeout,acc);
        }
    delay_line = buff;
}

void fir_32tap_core1_init()
{
    // Drop samples if not first block
    int const Delay = 8;
    for (int i = 0; i < Delay; ++i){
        get_ss(0);
    }
}
```

```

}
//initialize data
for (int i=0;i<8;i++){
    int tmp=get_ss(0);
    delay_line.set(*(cint16*)&tmp,i);
}
};

```

The third kernel code is as follows:

```

alignas(aie::vector_decl_align) static cint16 eq_coef2[8]={33,34},{35,36},...};

//For storing data between graph iterations
alignas(aie::vector_decl_align) static aie::vector<cint16,16> delay_line;
__attribute__((noinline)) void fir_32tap_core2(input_stream<cint16> * sig_in, input_cascade<cacc48> *
cascadein,
    output_cascade<cacc48> * cascadeout){
    const aie::vector<cint16,8> coe = aie::load_v<8>(eq_coef2);
    aie::vector<cint16,16> buff = delay_line;
    aie::accum<cacc48,4> acc;
    const unsigned LSIZE = (SAMPLES/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
        chess_prepare_for_pipelining
        {
            //8 MAC produce 4 partial output
            acc = readincr_v4(cascadein);
            buff.insert(2,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,0);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(3,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,4);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(0,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,8);
            writeincr(cascadeout,acc);

            acc = readincr_v4(cascadein);
            buff.insert(1,readincr_v<4>(sig_in));
            acc = aie::sliding_mac<4,8>(acc,coe,0,buff,12);
            writeincr(cascadeout,acc);
        }
    delay_line = buff;
}

void fir_32tap_core2_init(){
    // Drop samples if not first block
    int const Delay = 16;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
    //initialize data
    for (int i=0;i<8;i++){
        int tmp=get_ss(0);
        delay_line.set(*(cint16*)&tmp,i);
    }
};

```

The last kernel code is as follows:

```

alignas(aie::vector_decl_align) static cint16 eq_coef3[8]={49,50},{51,52},...};

```

```

//For storing data between graph iterations
alignas(aie::vector_decl_align) static aie::vector<cint16,16> delay_line;
__attribute__((noinline)) void fir_32tap_core3(input_stream<cint16> * sig_in, input_cascade<cacc48> *
cascadein,
    output_stream<cint16> * data_out){
    const aie::vector<cint16,8> coe = aie::load_v<8>(eq_coef3);
    aie::vector<cint16,16> buff = delay_line;
    aie::accum<cacc48,4> acc;
    const unsigned LSIZE = (SAMPLES/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    {
        //8 MAC produce 4 output
        acc = readincr_v4(cascadein);
        buff.insert(2,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<4,8>(acc,coe,0,buff,0);
        writeincr(data_out,acc.to_vector<cint16>(SHIFT));

        acc = readincr_v4(cascadein);
        buff.insert(3,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<4,8>(acc,coe,0,buff,4);
        writeincr(data_out,acc.to_vector<cint16>(SHIFT));

        acc = readincr_v4(cascadein);
        buff.insert(0,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<4,8>(acc,coe,0,buff,8);
        writeincr(data_out,acc.to_vector<cint16>(SHIFT));

        acc = readincr_v4(cascadein);
        buff.insert(1,readincr_v<4>(sig_in));
        acc = aie::sliding_mac<4,8>(acc,coe,0,buff,12);
        writeincr(data_out,acc.to_vector<cint16>(SHIFT));
    }
    delay_line = buff;
}
void fir_32tap_core3_init()
{
    // Drop samples if not first block
    int const Delay = 24;
    for (int i = 0; i < Delay; ++i){
        get_ss(0);
    }
    //initialize data
    for (int i=0;i<8;i++){
        int tmp=get_ss(0);
        delay_line.set(*(cint16*)&tmp,i);
    }
};

```

The last kernel writes results to the output stream using `acc.to_vector<cint16>(SHIFT)`.

Each kernel takes one cycle to produce a partial output. When they are working simultaneously, the system performance is one cycle to produce one output, which meets the design goal.

For more information about graph construction, stream broadcast, DMA FIFO insertion, profiling in simulation and hardware, design stall and deadlock analysis that can be met in system design, see *AI Engine Tools and Flows User Guide* ([UG1076](#)).

Matrix Multiplication

The AI Engine API provides a `aie::mmul` class template for a vector-based matrix multiplication. Multiple intermediate matrix multiplication results are accumulated to give the final result. For more details on the supported matrix multiplication shapes ($M \times K \times N$) and data types, see [Matrix Multiplication](#) in the *AI Engine API User Guide* ([UG1529](#)).

The `aie::mmul` operations `mul` and `mac` accept row-major format data for the vector-based matrix multiplication. Then for the `mac` operation of `aie::mmul`, arrange the data by $M \times K$ or $K \times N$. This data shuffling can be done either in the PL or AI Engine.

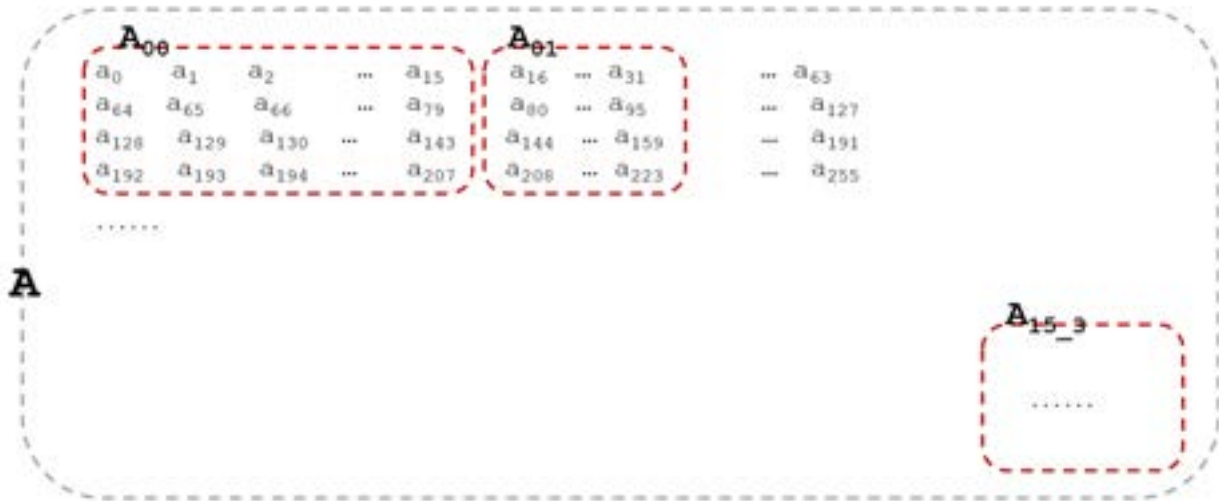
This section gives an example of $A(64 \times 64) \times B(64 \times 64)$ matrix multiplication. The data type is `int8 x int8`. The matrix

multiplication shape $4 \times 16 \times 8$ is chosen for `aie::mmul` operations.

The input data is assumed to be in row-major format. The data is input to the matrix multiplication kernel as 4×16 matrix and 16×8 matrix. Prior to the matrix multiplication kernel, the input data is shuffled.

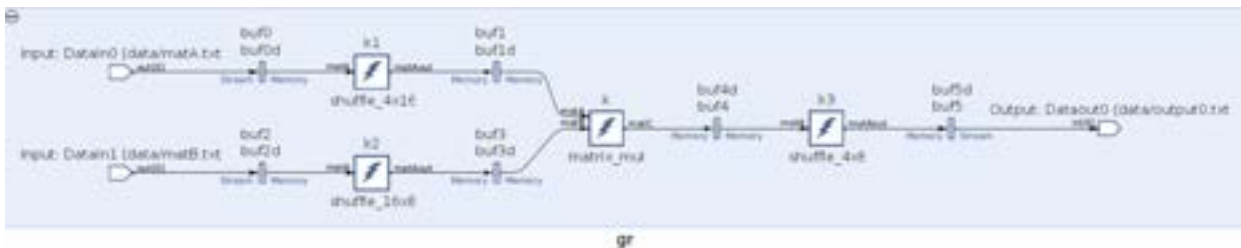
For example, before shuffling, matrix A (64×64) is stored in memory with `a0, a1, ..., a63, a64, ..., a4096` in order. The `aie::mmul` operations will use shapes 4×16 . The matrix A is partitioned into smaller matrix sized 4×16 . For the smaller matrix `A00`, `a0` to `a15`, `a64` to `a79`, `a128` to `a143`, and `a192` to `a207` should be fetched sequentially for `aie::mmul`. So, the purpose of the data shuffle is to put `a0` to `a15`, `a64` to `a79`, `a128` to `a143`, and `a192` to `a207` into continuous storage for the matrix multiplication kernel. This data shuffling is shown in the following figure.

Figure: Data Shuffling for Matrix A



Similarly, the output data is shuffled. The following figure shows the graph of the design.

Figure: Matrix Multiplication Kernel Graph



Vectorized Matrix Multiplication

The following is an example of a matrix multiplication with a $(64 \times 64) \times (64 \times 64)$ matrix size, and `int8 x int8` data type. The matrix multiplication shape is $4 \times 16 \times 8$. The input data is reshaped for matrix multiplication.

```
const int SHIFT=10;

//For element mmul
const int M=4;
const int K=16;
const int N=8;

//Total matrix sizes
const int rowA=64;
const int colA=64;
const int colB=64;

//mmul numbers
const int num_rowA=rowA/M;
const int num_colA=colA/K;
const int num_colB=colB/N;
```

```

void matrix_mul(input_buffer<int8> & __restrict matA, input_buffer<int8> & __restrict matB,
output_buffer<int8> & __restrict matC){
    using MMUL = aie::mmul<M, K, N, int8, int8>;

    const int8* __restrict pA=(int8*)matA.data();
    const int8* __restrict pB=(int8*)matB.data();
    int8* __restrict pC=(int8*)matC.data();
    //For profiling only
    unsigned cycle_num[2];
    aie::tile tile=aie::tile::current();
    cycle_num[0]=tile.cycles();//cycle counter of the AI Engine tile

    for (unsigned i = 0; i < num_rowA; i++) { //for output row number of element matrix
        for (unsigned j = 0; j < num_colB; j++) { //for output col number of element matrix
            const int8 * __restrict pA1 = pA + ( i * num_colA + 0 ) * MMUL::size_A;
            const int8 * __restrict pB1 = pB + ( 0 * num_colB + j ) * MMUL::size_B;

            aie::vector<int8, MMUL::size_A> A0 = aie::load_v<MMUL::size_A>(pA1); pA1 += MMUL::size_A;
            aie::vector<int8, MMUL::size_B> B0 = aie::load_v<MMUL::size_B>(pB1); pB1 += MMUL::size_B * num_colB;


            MMUL C00; C00.mul(A0, B0);

            for (unsigned k = 0; k < num_colA-1; k++) {
                A0 = aie::load_v<MMUL::size_A>(pA1); pA1 += MMUL::size_A;
                B0 = aie::load_v<MMUL::size_B>(pB1); pB1 += MMUL::size_B * num_colB;
                C00.mac(A0, B0);
            }

            aie::store_v(pC, C00.template to_vector<int8>(SHIFT)); pC += MMUL::size_C;
        }
    }
    //For profiling only
    cycle_num[1]=tile.cycles();//cycle counter of the AI Engine tile
    printf("start=%d,end=%d,total=%d\n", cycle_num[0], cycle_num[1], cycle_num[1]-cycle_num[0]);
}

```

The profiled result shows that the loop takes around 3254 cycles. In total, this is $64 \times 64 \times 64 = 262144$ multiplications on $\text{int8} \times \text{int8}$ data type, which is $262144/3254 \approx 80$ $\text{int8} \times \text{int8}$ MAC operations per cycle.

 **Note:** The exact number of cycles might fluctuate slightly based on the specific compiler settings and the version of the tool being used. However, the analysis techniques described in this section remain relevant and applicable regardless of these variations.

Data Shuffling Kernel

Because `aie::mmul` accepts row-major format vector data for shape of matrix multiplication, it might require data shuffling in PL or AI Engine with raw data for performance. This section assumes that the original data is row-major format for whole matrices. It shuffles the data to match the shape $4 \times 16 \times 8$ used in the matrix multiplication.

The following kernel code shuffles data for matrix A, with a target shape 4×16 :

```

//element matrix size
const int M=4;
const int N=16;

//Total matrix sizes
const int rowA=64;
const int colA=64;

void shuffle_4x16(input_buffer<int8> & __restrict matA, output_buffer<int8> & __restrict matAout){

    const int sizeA=M*N;
    auto pV=aie::begin_vector<16>((int8*)matA.data());
    auto pOut=aie::begin_vector<sizeA>((int8*)matAout.data());

    aie::vector<int8,sizeA> mm;
    for(int i=0;i<rowA/M;i++){

```



```

    for(int j=0;j<colA/N;j++){
        for(int k=0;k<M;k++){
            mm.insert(k,*pV);
            pV=pV+4;
        }
        *pOut+=mm;
        pV=pV-15;
    }
    pV=pV+12;
}
}

```

The following is an example of code used to shuffle data for matrix B, with a target shape 16*8:

```

//element matrix size
const int M=16;
const int N=8;

//Total matrix sizes
const int rowA=64;
const int colA=64;

void shuffle_16x8(input_buffer<int8> & __restrict matA, output_buffer<int8> & __restrict matAout){

    const int sizeA=M*N;
    auto pV=aie::begin_vector<16>((int8*)matA.data());
    auto pOut=aie::begin_vector<16>((int8*)matAout.data());

    aie::vector<int8,16> sv1,sv2;
    for(int i=0;i<rowA/M;i++){
        for(int j=0;j<colA/N/2;j++){
            for(int k=0;k<M/2;k++){
                sv1=*pV;
                pV=pV+4;
                sv2=*pV;
                pV=pV+4;
                auto mm=aie::interleave_zip(sv1,sv2,8);
                *pOut=mm.first;
                pOut+=8;
                *pOut=mm.second;
                pOut+=7;
            }
            pOut+=8;
            pV-=63;
        }
        pV+=60;
    }
}

```

The following is an example of code used to shuffle data for matrix C, with an input shape 4*8:

```

//element matrix size
const int M=4;
const int N=8;

//Total matrix sizes
const int rowA=64;
const int colA=64;

void shuffle_4x8(input_buffer<int8> & __restrict matA, output_buffer<int8> & __restrict matAout){
    const int sizeA=M*N;
    auto pV=aie::begin_vector<sizeA>((int8*)matA.data());
    auto pOut=aie::begin_vector<sizeA>((int8*)matAout.data());

    aie::vector<int8,sizeA> mm1,mm2,mm3,mm4;

```

```

for(int i=0;i<rowA/M;i++){
    for(int j=0;j<colA/N/4;j++){
        mm1=*pV++;
        mm2=*pV++;
        mm3=*pV++;
        mm4=*pV++;
        auto mm12=aie::interleave_zip(mm1,mm2,8);
        auto mm34=aie::interleave_zip(mm3,mm4,8);
        auto mm1234_low=aie::interleave_zip(mm12.first,mm34.first,16);
        auto mm1234_high=aie::interleave_zip(mm12.second,mm34.second,16);
        *pOut=mm1234_low.first;
        pOut=pOut+2;
        *pOut=mm1234_low.second;
        pOut=pOut+2;
        *pOut=mm1234_high.first;
        pOut=pOut+2;
        *pOut=mm1234_high.second;
        pOut=pOut+5;
    }
    pOut=pOut+6;
}
}
}

```

Introduction to Graph Programming

An AI Engine program must include a *Data Flow Graph Specification* written in C++. An adaptive data flow (ADF) graph is a network with a single AI Engine kernel or multiple AI Engine kernels connected by data streams. The graph can interact with the programmable logic (PL), global memory, and/or the host processor using specific constructs. `input_plio` and `output_plio` port objects can be used to establish stream connections to or from the programmable logic, `input_gmio` and `output_gmio` port objects can be used to establish memory-mapped connections to or from the global memory, and RTP (Runtime Parameter) objects can be used to setup and control parameters required by the kernels during graph execution.

A graph can have multiple kernels, input and output ports. The graph connectivity, which is equivalent to the nets in a data flow graph is either between the kernels, between kernel and input ports, or between kernel and output ports, and can be configured as a connection. A graph executes when all the data samples equaling the buffer or stream of data expected by the kernels in the graph become available, and produces data samples equaling the buffer or stream of data expected at the output of all the kernels in the graph.

As described in [C++ Template Support](#) you can use template classes or functions for writing the AI Engine graph or kernels. The application can be compiled and executed using the AI Engine tool chain. This chapter provides an introduction to writing an AI Engine program.


The example that is used in this chapter can be found as a template example in the AMD Vitis™ environment when creating a new AI Engine project.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

Prepare the Kernels

Kernels are computation functions that form the fundamental building blocks of the data flow graph specifications. Kernels are declared as ordinary C/C++ functions that return `void` and can use special data types as arguments. Each kernel must be defined in its own source file. This organization is recommended for reusability and faster compilation. Furthermore, the kernel source files should include all relevant header files to allow for independent compilation.

 **Note:** For the AI Engine kernel to use the AI Engine API, include the following file in the kernel source code:

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

It is recommended that a header file (`kernels.h` in this documentation) declares the function prototypes for all kernels used in a graph. An example is as follows.

```

#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H

void simple(adf::input_buffer<int16> & __restrict in,
            adf::output_buffer<int16> & __restrict out);

```

```
#endif
```

In the example, the `#ifndef` and `#endif` are present to ensure that the include file is only included once, which is good C/C++ practice. The `kernels.cc` file is the implementation file for the simple function. The kernel implementation uses two `int 16` variables `in_t` and `out_t`, which point to the underlying values of input and output buffer respectively using the `data()` member function.

```
/* A simple kernel */
#include <adf.h>
#include "kernels.h"

void simple(adf::input_buffer<int16> & __restrict in,
           adf::output_buffer<int16> & __restrict out)
{
    int16* __restrict in_t = in.data();
    int16* __restrict out_t = out.data();
    for (unsigned i=0; i<NUM_SAMPLES; i++) {
        *out_t++ = 1 + *in_t++;
    }
}
```

Related Information

[Input and Output Buffers](#)

[Streaming Data API](#)

Creating a Data Flow Graph (Including Kernels)

This following process describes how to construct data flow graphs in C++.

1. Define your application graph class in a separate header file (for example `project.h`). First, add the Adaptive Data Flow (ADF) header (`adf.h`) and include the kernel function prototypes. The ADF library includes all the required constructs for defining and executing the graphs on AI Engines.

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

2. Define your graph class by using the objects which are defined in the `adf` namespace. All user graphs are derived from the class `graph`.

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

class simpleGraph : public adf::graph {
private:
    adf::kernel first;
    adf::kernel second;
};
```

This is the beginning of a graph class definition that declares two kernels (`first` and `second`).

3. Add some top-level input/output objects, `input_plio` and `output_plio`, to the graph.

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

class simpleGraph : public adf::graph {
private:
    adf::kernel first;
    adf::kernel second;
public:
    adf::input_plio in;
    adf::output_plio out;
};
```

4. Use the `kernel::create` function to instantiate the `first` and `second` C++ kernel objects using the functionality of the C function

simple.

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

class simpleGraph : public adf::graph {
private:
    adf::kernel first;
    adf::kernel second;
public:
    adf::input_plio in;
    adf::output_plio out;
    simpleGraph() {
        first = adf::kernel::create(simple);
        second = adf::kernel::create(simple);
    }
};
```

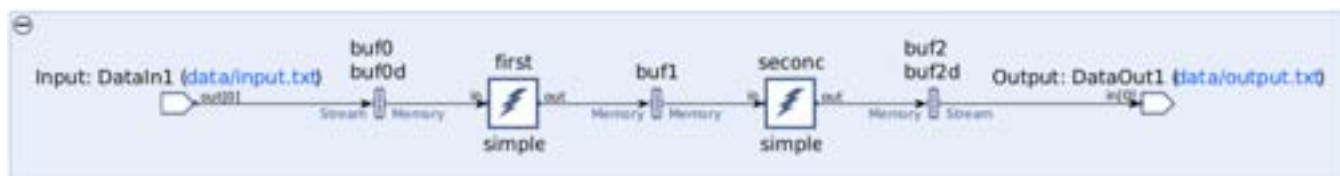
5. Configure input/output objects with specified PLIO width and input/output files, and add the connectivity information, which is equivalent to the nets in a data flow graph. In this description, input/output objects are referenced by indices. The first input buffer or stream argument in the `simple` function is assigned index 0 in an array of input ports (`in`). Subsequent input arguments take ascending consecutive indices. The first output buffer or stream argument in the `simple` function is assigned index 0 in an array of output ports (`out`). Subsequent output arguments take ascending consecutive indices.

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

class simpleGraph : public adf::graph {
private:
    adf::kernel first;
    adf::kernel second;
public:
    adf::input_plio in;
    adf::output_plio out;

    simpleGraph() {
        first = adf::kernel::create(simple);
        second = adf::kernel::create(simple);

        in = adf::input_plio::create(plio_32_bits, "data/input.txt");
        out = adf::output_plio::create(plio_32_bits, "data/output.txt");
        adf::connect(in.out[0], first.in[0]);
        adf::connect(first.out[0], second.in[0]);
        adf::connect(second.out[0], out.in[0]);
        adf::dimensions(first.in[0]) = {128};
        adf::dimensions(first.out[0]) = {128};
        adf::dimensions(second.in[0]) = {128};
        adf::dimensions(second.out[0]) = {128};
    }
};
```



This figure represents the graph connectivity specified in the previous graph code. Graph connectivity can be viewed when you open the compilation results in the Vitis IDE. For more information, see [Viewing AI Engine Compilation Summary Results](#) in the *Vitis Reference Guide* (UG1702). As shown in the previous figure, the input port from the top level is connected into the input port of the first kernel, the output port of the first kernel is connected to the input port of the second kernel, and the output port of the second kernel is connected to the output exposed to the top level. The first kernel executes when 128 bytes of data (32 complex samples) are collected in a buffer from an external source. This is specified using a `dimensions(first.in[0])={128}` construct. Likewise, the second kernel executes

when its input buffer has valid data being produced as the output of the first kernel. Finally, the output of the second kernel is connected to the top-level output port and the dimensions (second.out[0])={128} specifies the number of samples of data kernels produced upon termination.

buf0 and buf0d are ping pong buffers allocated for the first kernel input buffer. Similarly, buf2 and buf2d are ping pong buffers allocated for the second kernel output buffer. Notice that buf1 which is the buffer output from the first kernel to the second kernel is not a ping pong buffer. This is because both the first and second kernel buffers are placed in a single AI Engine tile where they will execute sequentially. See [Runtime Ratio](#) for more information.

- Set the source file and tile usage for each of the kernels. The source file kernel.cc contains kernel first and kernel second source code. Then the ratio of the function runtime compared to the cycle budget, known as the runtime ratio, and must be between 0 and 1. The cycle budget is the number of instruction cycles a function can take to either consume data from its input (when dealing with a rate limited input data stream), or to produce a block of data on its output (when dealing with a rate limited output data stream). This cycle budget can be affected by changing the block sizes.

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

class simpleGraph : public adf::graph {
private:
    adf::kernel first;
    adf::kernel second;
public:
    adf::input_plio in;
    adf::output_plio out;

    simpleGraph(){

        first = adf::kernel::create(simple);
        second = adf::kernel::create(simple);

        in = adf::input_plio::create(plio_32_bits, "data/input.txt");
        out = adf::output_plio::create(plio_32_bits, "data/output.txt");

        adf::connect(in.out[0], first.in[0]);
        adf::connect(first.out[0], second.in[0]);
        adf::connect(second.out[0], out.in[0]);
        adf::dimensions(first.in[0]) = {128};
        adf::dimensions(first.out[0]) = {128};
        adf::dimensions(second.in[0]) = {128};
        adf::dimensions(second.out[0]) = {128};

        adf::source(first) = "kernels.cc";
        adf::source(second) = "kernels.cc";

        adf::runtime<ratio>(first) = 0.1;
        adf::runtime<ratio>(second) = 0.1;

    }
};
```

 **Note:** See [Runtime Ratio](#) for more information.

- Define a top-level application file (for example project.cpp) that contains an instance of your graph class.

```
#include "project.h"

simpleGraph mygraph;

int main(void) {
    adf::return_code ret;
    mygraph.init();
    ret=mygraph.run(<number_of_iterations>);
    if(ret!=adf::ok){
        printf("Run failed\n");
        return ret;
    }
}
```

```

ret=mygraph.end();
if(ret!=adf::ok){
    printf("End failed\n");
    return ret;
}
return 0; //Must have return statement
}


```

!! Important: By default, the `mygraph.run()` option specifies a graph that runs forever. The AI Engine compiler generates code to execute the data flow graph in a perpetual while loop. To limit the execution of the graph for debugging and test, specify the `mygraph.run(<number_of_iterations>)` in the graph code. The specified number of iterations can be one or more.

!! Important: The main function must have a return statement. Otherwise, the AI Engine compiler will error out.

ADF APIs have return enumerate type `return_code` to show the API running status.

The main program is the driver for the graph. It is used to load, execute, and terminate the graph.

 **Note:** Kernel code must be written in such a way that no name clashes occur when two kernels get assigned to the same core.

Related Information

[Runtime Graph Control API](#)


Runtime Ratio

The runtime ratio is a user-specified constraint that allows the AI Engine tools the flexibility to put multiple AI Engine kernels into a single AI Engine, if their total runtime ratio is less than one. The runtime ratio of a kernel can be computed using the following equation.

$\text{runtime ratio} = (\text{cycles for one run of the kernel}) / (\text{cycle budget})$

The cycle budget is the number of cycles allowed to run one invocation of the kernel, which depends on the system throughput requirement. Cycles for one run of the kernel can be estimated in the initial design stage. For example, if the kernel contains a loop that can be well pipelined, and each cycle is capable of handling that amount of data, then the cycles for one run of the kernel can be estimated by the following.

synchronization of synchronous buffers + function initialization + loop count * cycles of each iteration of the loop + preamble and postamble of the loop

 **Note:** synchronization of synchronous buffers + function initialization takes tens of cycles, depending on the interface numbers. This needs to be taken into account when targeting for performance.

Cycles for one run of the kernel can also be profiled in the `aiesimulator` when vectorized code is available.

If multiple AI Engine kernels are put into a single AI Engine, they run in a sequential manner, one after the other, and they all run once with each iteration of `graph::run`, unless there is a multi-rate processing. This means the following.

- If the AI Engine runtime percentage (specified by the runtime constraint) is allocated for the kernel in each iteration of `graph::run` (or on an average basis, depending on the system requirement), the kernel performance requirement can be met.
- For a single iteration of `graph::run`, the kernel takes no more percentage than that specified by the runtime constraint. Otherwise, it might affect other kernels' performance that are located in the same AI Engine.
- Even if multiple kernels have a summarized runtime ratio less than one, they are not necessarily put into a single AI Engine. The mapping of an AI Engine kernel into an AI Engine is also affected by hardware resources. For example, there must be enough program memory to allow the kernels to be in the same AI Engine, and also, stream interfaces must be available to allow all the kernels to be in the same AI Engine.
- When multiple kernels are put into the same AI Engine, resources might be saved. For example, the buffers between the kernels in the same AI Engine are single buffers instead of ping-pong buffers.
- Increasing the runtime ratio of a kernel does not necessarily mean that the performance of the kernel or the graph is increased, because the performance is also affected by the data availability to the kernel and the data throughput in and out of the graph. A pessimistically high runtime ratio setting might result in inefficient resource utilization.
- Low runtime ratio does not necessarily limit the performance of the kernel to the specified percentage of the AI Engine. For example, the kernel can run immediately when all the data is available if there is only one kernel in the AI Engine, no matter what runtime ratio is set.
- Kernels in different top-level graphs can not be put into the same AI Engine, because the graph API needs to control different graphs independently.
- Set the runtime ratio as accurately as possible, because it affects not only the AI Engine to be used, but also the data communication routes between kernels. It might also affect other design flows, for example, the power estimation.

Recommended Project Directory Structure

The following directory structure and coding practices are recommended for organizing your AI Engine projects to provide clarity and reuse.

- All adaptive data flow (ADF) graph class definitions, that is, all the ADF graphs that are derived from graph class `adf::graph`, must be located in a header file. Multiple ADF graph definitions can be included in the same header file. This class header file should be included in the main application file where the actual top-level graph is declared in the file scope.
- There should be no dependencies on the order that the header files are included. All header files must be self-contained and include all the other header files that they need.
- There should be no file scoped variable or data-structure definitions in the graph header files. Any definitions (including `static`) must be declared in a separate source file that can be identified in the `header` property of the kernel where they are referenced (see [Lookup Tables](#)).
- Only one source file is allowed to be specified as kernel source via `adf::source`. When sub-functions are defined as a library in separate source files (for example, `util.hpp` and `util.cpp`), the workaround is to:
 - Include all the library source files in a header file (for example, `lib.hpp`):

```
#pragma once
#include <util.hpp>
#include <util.cpp>
```

- And then include the header file `lib.hpp` in the kernel source file.

Related Information

[Creating a Data Flow Graph \(Including Kernels\)](#)

[Lookup Tables](#)

Compiling and Running the Graph from the Command Line

1. To compile your graph, execute the following command (see [Compiling an AI Engine Graph Application](#) in *AI Engine Tools and Flows User Guide* ([UG1076](#)) for more details).

```
v++ -c --mode aie --target hw --config ./config.cfg project.cpp
```

The program is called `project.cpp`. The AI Engine compiler reads the input graph specified, compiles it to the AI Engine array, produces various reports, and generates output files in the `Work` directory.

2. After parsing the C++ input into a graphical intermediate form expressed in JavaScript object notation (JSON), the AI Engine compiler performs resource mapping, scheduling analysis, and maps kernel nodes in the graph to the processing cores in the AI Engine array and data buffers to memory banks. The JSON representation is augmented with mapping information. Each AI Engine also requires a schedule of all the kernels mapped into it.

The input graph is first partitioned into groups of kernels to be mapped to the same core.

The output of the mapper can also be viewed as a tabular report in the file `project_mapping_analysis_report.txt`. This reports the mapping of nodes to processing cores and data buffers to memory banks. Inter-processor communication is appropriately double-banked as ping-pong buffers.

3. The AI Engine compiler allocates the necessary locks, memory buffers, DMA channels, 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/C++ program for each core is compiled to produce loadable ELF files. The AI Engine compiler also generates control APIs to control the graph initialization, execution and termination from the main application and a simulator configuration script `scsim_config.json`. These are all stored within the `Work` directory under various sub-folders (see [Compiling an AI Engine Graph Application](#) in *AI Engine Tools and Flows User Guide* ([UG1076](#)) for more details).
4. After the compilation of the AI Engine graph, the AI Engine compiler writes a summary of compilation results called `<graph-file-name>.aiecompile_summary` to be viewed in the Vitis IDE. The summary contains a collection of reports, and diagrams reflecting the state of the AI Engine application implemented in the compiled build. The summary is written to the working directory of the AI Engine compiler as specified by the `--work_dir` option, which defaults to `./Work`.

To open the AI Engine compiler summary, use the following command:

```
vitis -a ./Work/graph.aiecompile_summary
```


5. To run the graph, execute the following command (see [Simulating an AI Engine Graph Application](#) in *AI Engine Tools and Flows User Guide* ([UG1076](#)) for more details).

```
aiesimulator --pkg-dir=./Work
```

This starts the SystemC-based simulator with the control program being the `main` application. The graph APIs which are used in the control program configure the AI Engine array including setting up static routing, programming the DMAs, loading the ELF files onto the individual cores, and then initiates AI Engine array execution.

At the end of the simulation, the output data is produced in the directory `aiesimulator_output` and it should match the reference data.

The graph can be loaded at device boot time in hardware or through the host application. Details on deploying the graph in hardware and the flow associated with it is described in detail in [Building and Running the System](#) in the *Data Center Acceleration using Vitis (UG1700)*.

 **Note:** Only AI Engine kernels that have been modified are recompiled in subsequent compilations of the AI Engine graph. Any un-modified kernels will not be recompiled.

Memory and DMA Programming

The AI Engine processor can leverage local memory for efficient data access:

AI Engine memory

The AI Engine processor has on-tile data memory, providing efficient access within the same processing unit. Additionally, it can access data memory from neighboring tiles to the north, south, and west (or east), expanding its reach and enabling efficient data flow within a larger AI Engine array.

This memory can be addressed linearly within dedicated address ranges. Additionally, they support multidimensional addressing for more complex access patterns. Direct Memory Access (DMA) controllers manage these memories automatically. These DMAs offer various programmable features to optimize memory access, including:

Table: DMA Features

DMA Feature	Description	AI Engine Tile DMA ¹
Maximum Addressing Dimension	Maximum dimension for accessing data available to the DMA depending on the type of memory. Depending on the application, you can access data in a uni-dimensional (1D), bi-dimensional (2D, as in a gray-scale image).	2D
Packet-ID	Feature that is used by AXI4-Stream switch to drive packets to their destination based on the Packet ID. See Explicit Packet Switching .	Supported
Number of Buffer Descriptors	Lists the total number of buffer descriptors available in the memory. A Buffer Descriptor describes a DMA transfer. Each buffer descriptor contains all information needed for a DMA transfer. They are used by the DMA to specify the read/write access schemes to the memory.	16
Number of Locks	List the total number of locks available in the memory. These locks are used by the DMA to handle synchronization for Buffer Descriptor usage. They are used to organize read and write access to the memory.	16
1. Used to access local buffers.		

The AI Engine tiling parameters enable you to program the DMAs for the AI Engine Memory Module. They can be declared and defined in the ADF graph and connected to the appropriate kernel inputs or outputs. For more information on tiling parameters, see [Tiling Parameters and Buffer Descriptors](#).

AI Engine Local Memory Access

The buffers used in the various kernels are automatically inferred at the graph level as ping-pong buffers located in the AI Engine data memory. From the kernel point of view, the buffers are accessed through the address generator units of the processor and are managed by the kernel. Input/output buffers can be filled and flushed by other components of the device using streams that are handled internally by a DMA. In order to simplify kernel addressing, the user may require the data to be stored in a specific way in the data memory. For instance, a tensor transposition might be needed to simplify matrix multiplication at the kernel level. You can define specific access patterns for this type of memory. Defining tiling parameters enables flexible addressing for local memories.

The code snippet demonstrates the API for utilizing the AI Engine data memory to perform a matrix multiplication where matrix B has been transposed:

```
class MatrixMultiply : public graph {
```



```

public:
input_port inA,inB;
output_port outC;

kernel MatMult;

MatrixMultiply() {

    // kernel
    MatMult = kernel::create(ClassicMatMult);
    source(MatMult) = "src/matmult.cpp";
    runtime<ratio>(MatMult) = 0.9;

    // Connect Input A to MatMult Kernel
    connect(inA, MatMult.in[0]);
    dimensions(MatMult.in[0]) = {NColsA,NRowsA};

    // Connect Input B to MatMult Kernel
    connect(inB, MatMult.in[1]);
    dimensions(MatMult.in[1]) = {NRowsB,NColsB};
    /* tiling parameters to transpose matrix */
    write_access(MatMult.in[1]) = adf::tiling(...);

    // Connect MatMult Kernel to Output
    connect(MatMult.out[0],outC);
    dimensions(MatMult.out[0]) = {NColsC,NRowsC};
};
};

```

The AI Engine compiler examines the connections in the constructor to determine the correct number of buffers at the input and the output ports of the kernel. The buffer sizes are parametrized by:

```
dimensions(MatMult.in[0]) = {NColsA,NRowsA};
```

This size definition can also be set directly within the kernel.

Tiling Parameters and Buffer Descriptors

DMA transfers are managed by buffer descriptors located in AI Engine memory.

These buffer descriptors handle 1D and 2D memory addressing, multiple iterations, lock ID, and buffer descriptor chaining.

Buffer descriptors address generation can be complex and hard to maintain over time. Hence, higher-level address generation is supported by the AI Engine compiler when you create and configure tiling parameters objects in the graph. When data is transferred, it is based on the configuration settings of the tiling parameter object associated with the memory. Data transfers occur on a tile basis, that can be as small as a 1x1 element, that are regularly extracted from or written to a memory space. The structure `tiling_parameters` is defined as follows:

```

struct tiling_parameters
{
    std::vector<uint32_t> buffer_dimension;
    std::vector<uint32_t> tiling_dimension;
    std::vector<int32_t> offset;
    std::vector<traversing_parameters> tile_traversal;
    int packet_port_id = -1;
    std::vector<uint32_t> boundary_dimension;
};

```

The members of this structure are:

buffer_dimension

Buffer dimensions in the memory element type (for example, AI Engine memory). `buffer_dimension[0]` is contiguous in memory and has the fastest access. When this member is not specified, the dimensions of the associated memory object are used. The AI Engine memory can access data in the first and second dimensions.

tiling_dimension

Tiling dimensions of the data transfer in buffer. The tiling dimension of AI Engine memory can access data in the first and second dimensions.

offset

Multidimensional offset with respect to the starting element in the buffer, assuming the buffer dimension is specified.

tile_traversal

Vector of traversing_parameters. `tile_traversal[i]` represents the *i*-th loop of inter-tile traversal, where *i*=0 represents most inner loop and *i*=N-1 represents most outer loop. `tile_traversal` structure is detailed the section below.

packet_port_id

Multiple connections can go through a single port that are previously merged through a `pktmerge` block or split afterward with a `pktsplit` block. This member represents the output port ID of the connected `pktsplit` or the input port ID of the connected `pktmerge`. If this member is set to a specific id, the data transfer will only happen if the incoming or outgoing data block ID matches this ID.

boundary_dimension

Real data boundary dimension.

A key member of the tiling parameter is the `tile_traversal` vector that describes how the buffer is accessed. The structure `traversing_parameters` is defined as follows:

```
struct traversing_parameters
{
    uint32_t dimension;
    uint32_t stride;
    uint32_t wrap;
};
```

The members of this structure are:

dimension

The buffer dimension on which this traversing loop applies. It could be the 0 or first dimension. The stride and wrap members of this structure are applied in the specified dimension.

stride

Represents the distance in terms of buffer element data type between consecutive inter-tile traversal in this dimension.

wrap

Number of tiles to access in this dimension.

When the stride value is lower than the tile size in one or more dimensions, the tiles overlap naturally in that dimension.

!! Important: All generated addresses are 32-bit aligned. The requirements for accessing data that is less than or equal to 16 bits are:

- 16-bit data are accessed as pairs.
 - 8-bit data are accessed as fours.
 - 4-bit data are accessed as eights.
-

Tiling Parameters Specification

Tiling parameters are specified in the graph and associated to an input or output port of an AI Engine memory DMA. In the following example, where six tiles (4x3 samples) must be written to a 12x8 sample buffer by kernel k1, and where six tiles (7x2 samples) must be read out of the same buffer by kernel k2:

Figure: Write Scheme

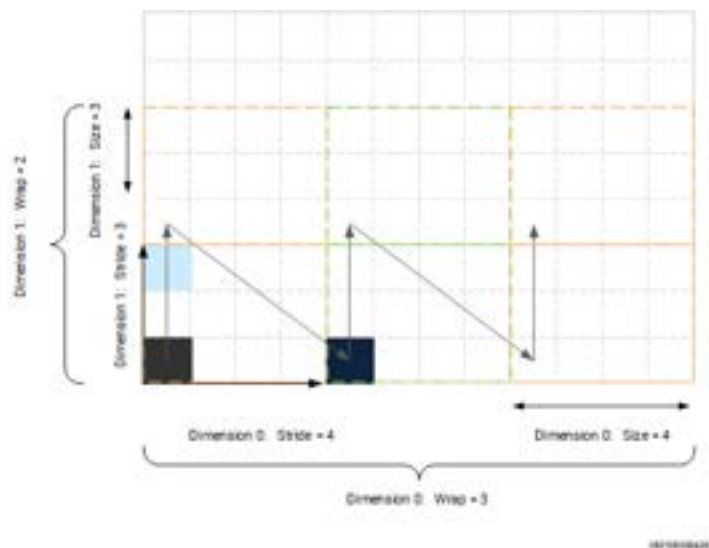
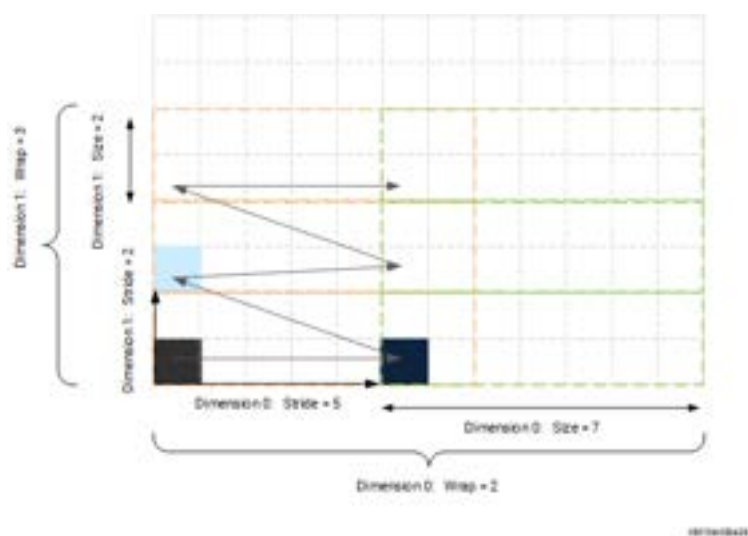


Figure: Read Scheme



```
kernel k1, k2;

mygraph()
{
    k1 = kernel::create(func1);
    k2 = kernel::create(func2);

    connect n1(k1.out[0], k2.in[0]);

    read_access(k1.out[0]) = tiling({
        .buffer_dimension={12,8},
        .tiling_dimension={4,3},
        .offset={0,0},
        .tile_traversal = {{.dimension=1, .stride=3, .wrap=2},
            {.dimension=0, .stride=4, .wrap=3}}});


    read_access(k2.in[0]) = tiling({
        .buffer_dimension={12,8},
        .tiling_dimension={7,2},
        .offset={0,0},
        .tile_traversal = {{.dimension=0, .stride=5, .wrap=2},
            {.dimension=1, .stride=2, .wrap=3}}});
};
```

With this construct, the two kernels communicate through two ping-pong buffers. A DMA transfer is automatically added by the AI Engine

compiler between a ping-pong output buffer on the *k1* side and a ping-pong input buffer on the *k2* side. Within each tile the data are read/written with dimension 0 as the inner-loop, but the tile selection follows the `tile_traversal` vector specification.

On the *k1* side the MM2S DMA accesses the tiles column-wise as per the `tile_traversal` vector starts with dimension 1, which is followed by dimension 0.

On the *k2* side the S2MM DMA accesses the tiles row-wise as per the `tile_traversal` vector which starts with dimension 0. The read access overlaps in dimension 0 as the specified stride in this dimension is less than the tile size.

 **Note:** DMA data access is based on buffer descriptors. All tiling parameters that you specify in the graph are translated into one or multiple buffer descriptor parameter sets. The tile itself added to the tile traversal parameters can require many buffer descriptors. The compiler could run out of hardware resources and issue an error:

```
Failed to allocate buffer descriptors for TG.G1.mtxin due to insufficient number of available buffer descriptors.
```

Tiling Parameters Examples

This section uses a graph that contains two kernels that are directly connected to show how data is written to or read from the AI Engine memory by tiling parameters. The graph code is:

```
class mygraph: public graph {
public:
    kernel first,second;
    input_plio in;
    output_plio out;

    mygraph() {
        first=kernel::create(producer);
        source(first) = "producer.cc";
        second=kernel::create(consumer);
        source(second) = "consumer.cc";

        in=input_plio::create("Datain0", plio_64_bits, "data/input.txt");
        out=output_plio::create("Dataout0", plio_64_bits, "data/output.txt");

        runtime<ratio>(first)=0.9;
        runtime<ratio>(second)=0.9;
        connect(in.out[0], first.in[0]);
        connect(first.out[0], second.in[0]);
        connect(second.out[0], out.in[0]);
        dimensions(first.in[0])={256};
        dimensions(first.out[0])={256};
        dimensions(second.in[0])={16,16};
        dimensions(second.out[0])={16,16};

        read_access(first.out[0]) = tiling({.buffer_dimension={256}, .tiling_dimension={256}, .offset={0} });
        write_access(second.in[0]) = tiling({.buffer_dimension={256}, .tiling_dimension={256}, .offset={0} });
    }
};
```

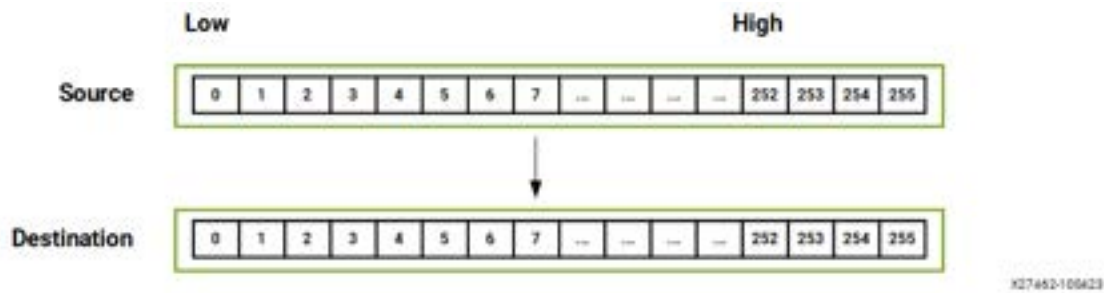
Following are examples of different tiling configurations. You can see how source data is transferred to destination, how data is stored in row basis in memory, and how the configurations can be applied on `write_access` or `read_access` by default.

In the graphics *Source* represents data organization within the *first* output buffer, and *Destination* represents data organization within the *second* input buffer.

1D Linear Transferring From Source to Destination

1D Linear transferring from source to destination using three equivalent lines of code (any one applies):

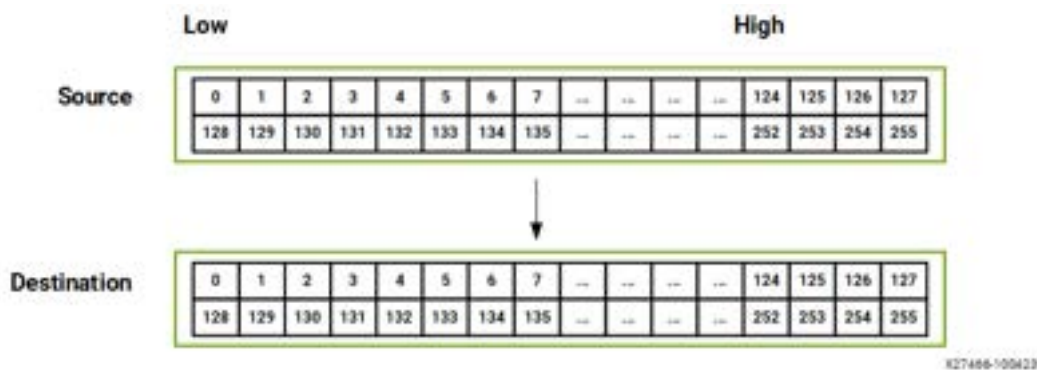
```
tiling({.buffer_dimension={256}, .tiling_dimension={256}, .offset={0} });
tiling({.buffer_dimension={256}, .tiling_dimension={256}, .offset={0},
        .tile_traversal = {{.dimension=0, .stride=256, .wrap=1}} });
tiling({.buffer_dimension={256}, .tiling_dimension={1}, .offset={0},
        .tile_traversal = {{.dimension=0, .stride=1, .wrap=256}} });
```

Figure: Example of 1D Linear Transferring From Source to Destination

2D Linear Transferring from Source to Destination

2D Linear transferring from source to destination using two equivalent lines of code (any one applies):

```
tiling({.buffer_dimension={128,2}, .tiling_dimension={128,2},
        .offset={0,0} });
tiling({.buffer_dimension={128,2}, .tiling_dimension={128,1}, .offset={0,0},
        .tile_traversal = {{.dimension=0, .stride=128, .wrap=1},
                           {.dimension=1, .stride=1, .wrap=2}} });
```

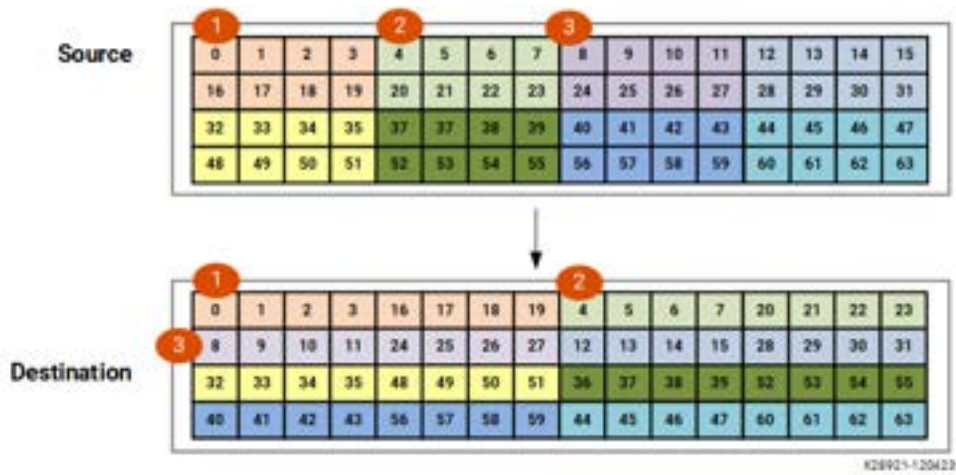
Figure: 2D Linear

2D Matrix Transferring to 4x2 Sub Matrices

2D Matrix transferring to transfer 4x2 sub matrices. The numbers show how data are stored in source and transferred to destination with one such tiling parameter:

```
read_access(first.out[0]) = tiling({.buffer_dimension={16,4},
        .tiling_dimension={4,2}, .offset={0,0},
        .tile_traversal = {{.dimension=0, .stride=4, .wrap=4},
                           {.dimension=1, .stride=2, .wrap=2}} });
```

Figure: Example of 2D Matrix Transferring to 4x2 Sub Matrices



2D Matrix Transpose Using Tiling Parameters

Tiling parameters can be used to handle a matrix transpose. A matrix transpose creates a new matrix wherein the rows correspond to the columns of the original matrix.

Because the DMAs can only generate addresses for 32-bit aligned data, transpose results will be exact for 32-bit (int32, uint32, cint32, cint16, float, cfloat) or higher data widths.

The transpose result is only partial and will need further steps for 16-bit, 8-bit and 4-bit data.

For 16-bit data, matrix elements have to be taken 2x1 when performing the transpose. For 8-bit data, the elements must be taken 4x1. For data widths lower than 32 bits, matrix transpose can be a two step process, with the first partial transpose achieved using the tiling parameters on the memory tile data, and the next step of the final transpose of the partial results in the kernel.

In the following code, the kernels are passthrough functions that copy the input data into the output without any modification. The transpose is handled by the DMA through the read access tiling parameter setting:

```
class Transpose : public adf::graph
{
public:
    adf::kernel k32_1,k16_1,k8_1;
    adf::kernel k32_2,k16_2,k8_2;

    adf::input_plio plin32,plin16,plin8;
    adf::output_plio plout32,plout16,plout8;

    Transpose()
    {

        // Kernel Creation
        k32_1 = adf::kernel::create(Passthrough<int32,Nrows,Ncolumns>);
        k16_1 = adf::kernel::create(Passthrough<int16,Nrows,Ncolumns>);
        k8_1 = adf::kernel::create(Passthrough<int8,Nrows,Ncolumns>);
        k32_2 = adf::kernel::create(Passthrough<int32,Nrows,Ncolumns>);
        k16_2 = adf::kernel::create(Passthrough<int16,Nrows,Ncolumns>);
        k8_2 = adf::kernel::create(Passthrough<int8,Nrows,Ncolumns>);

        adf::source(k32_1) = "Passthrough.cpp";
        adf::runtime<ratio>(k32_1) = 0.9;
        adf::source(k32_2) = "Passthrough.cpp";
        adf::runtime<ratio>(k32_2) = 0.9;

        ...

        // PLIO Creation
        plin32 = adf::input_plio::create("input64_32", adf::plio_64_bits,
            "data/Input_32.csv", 625);
        plout32 = adf::output_plio::create("output64_32", adf::plio_64_bits,
            "data/Output_32.csv", 625);

        ...

        // Connections
```

```

adf::connect (plin32.out[0],k32_1.in[0]);
adf::connect(k32_1.out[0],k32_2.in[0]);
adf::connect(k32_2.out[0],plout32.in[0]);
...

// Tiling Parameters
read_access(k32_1.out[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},
        .tiling_dimension={Ncolumns,Nrows},.offset={0,0}});
write_access(k32_2.in[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},
        .tiling_dimension={1,1},.offset={0,0},
        .tile_traversal = {{.dimension=1,.stride=1,.wrap=Nrows},
            {.dimension=0, .stride=1, .wrap=Ncolumns}}});

read_access(k16_1.out[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},.tiling_dimension={Ncolumns,Nrows},.offset={0,0}});
write_access(k16_2.in[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},
        .tiling_dimension={2,1},.offset={0,0},
        .tile_traversal = {{.dimension=1,.stride=1,.wrap=Nrows},
            {.dimension=0, .stride=2, .wrap=Ncolumns/2}}});

read_access(k8_1.out[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},
        .tiling_dimension={Ncolumns,Nrows},.offset={0,0}});
write_access(k8_2.in[0]) =
    adf::tiling({.buffer_dimension={Ncolumns,Nrows},
        .tiling_dimension={4,1},.offset={0,0},
        .tile_traversal = {{.dimension=1,.stride=1,.wrap=Nrows},
            {.dimension=0, .stride=4, .wrap=Ncolumns/4}}});

};
};

```

It can be seen that for the 16-bit case the data are taken 2x1 (.tiling_dimension={2,1}) so that the size of each tiling is 32 bits. If the size of the tiling is not a multiple of 32 bits, you will get an error message generated by the AI Engine compiler:

```

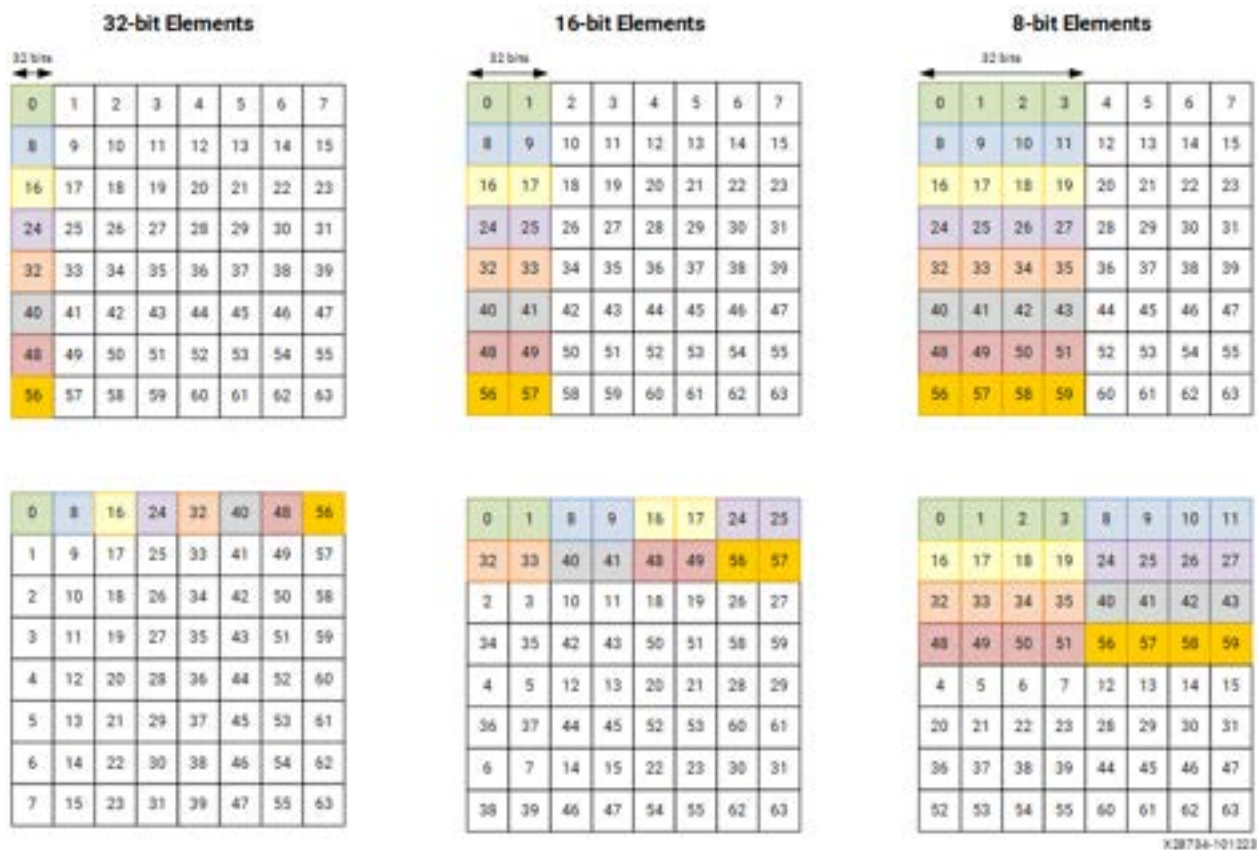
Failed in converting tiling parameters to buffer descriptors for port T.mtx16.out[0]: In dimension 0,
stepsize 8 and element size 16 (in bytes) cannot be adjusted to 32 bit word address. In dimension 1, stepsize
1 and element size 16 (in bytes) cannot be adjusted to 32 bit word address.

```

A step size of 1 with a 16-bit data width will generate 16-bit aligned addresses which is not supported.

The following figure provides a visualization of the effect of this transpose function with 32-bit, 16-bit and 8-bit data:

Figure: 32-bit, 16-bit and 8-bit Results For An 8x8 Matrix



In this example, all matrices get transposed in various ways depending on data bitwidth. The color coding helps understand how the data are shuffled around. The effect on the 32-bit data is clearly a transpose function. 16-bit and 8-bit data end up with a partial transpose where multiple data on each row (contiguous on the column dimension) are kept contiguous on the column dimension. This can be used if you have multiple matrices that are interleaved in the column dimension. For example, for 16-bit data you could have two matrices (or 4, 6, 8, ...) where element (0,0) of the first matrix is followed by the exact same element of the second matrix, and so on for element (0,1). If the bit width of the concatenation of each element coming from all matrices is a multiple of 32 bits, then they can all be transposed at once using tiling parameters.

Viewing Tiling Parameters in the Vitis IDE

After compiling an AI Engine graph, the tiling parameters settings are summarized in the Tiling Parameters table in the Vitis IDE. An example command to open the compile summary is as follows:

```
vitis -a Work/graph.aiecompile_summary
```

The Tiling Parameters table displays:

Figure: Tiling Parameters Table



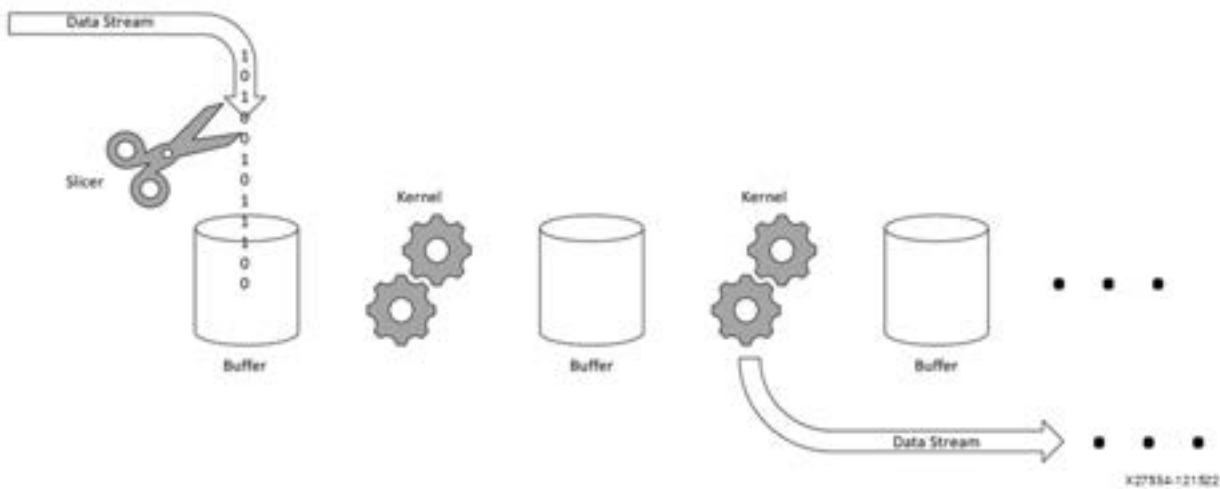
Input and Output Buffers

Input and output buffers represent a block of data that is stored contiguously on a tile's physical memory, and that can be used by kernels in a graph. The origin of this data can be either other kernels that produce them, or they can come from the PL through AI Engine array interface. The buffer port can be allocated in the physical memory of the tile where the kernel executes or in the physical memory of accessible adjacent tiles.

When a kernel has a buffer port on its input side, it waits for the buffer to be fully available before it starts execution. The kernel can access the contents of the buffer port either randomly or in a linear fashion. Conversely the kernel can write a block (frame) of data to the local memory that can be used by other kernels after it has finished execution.

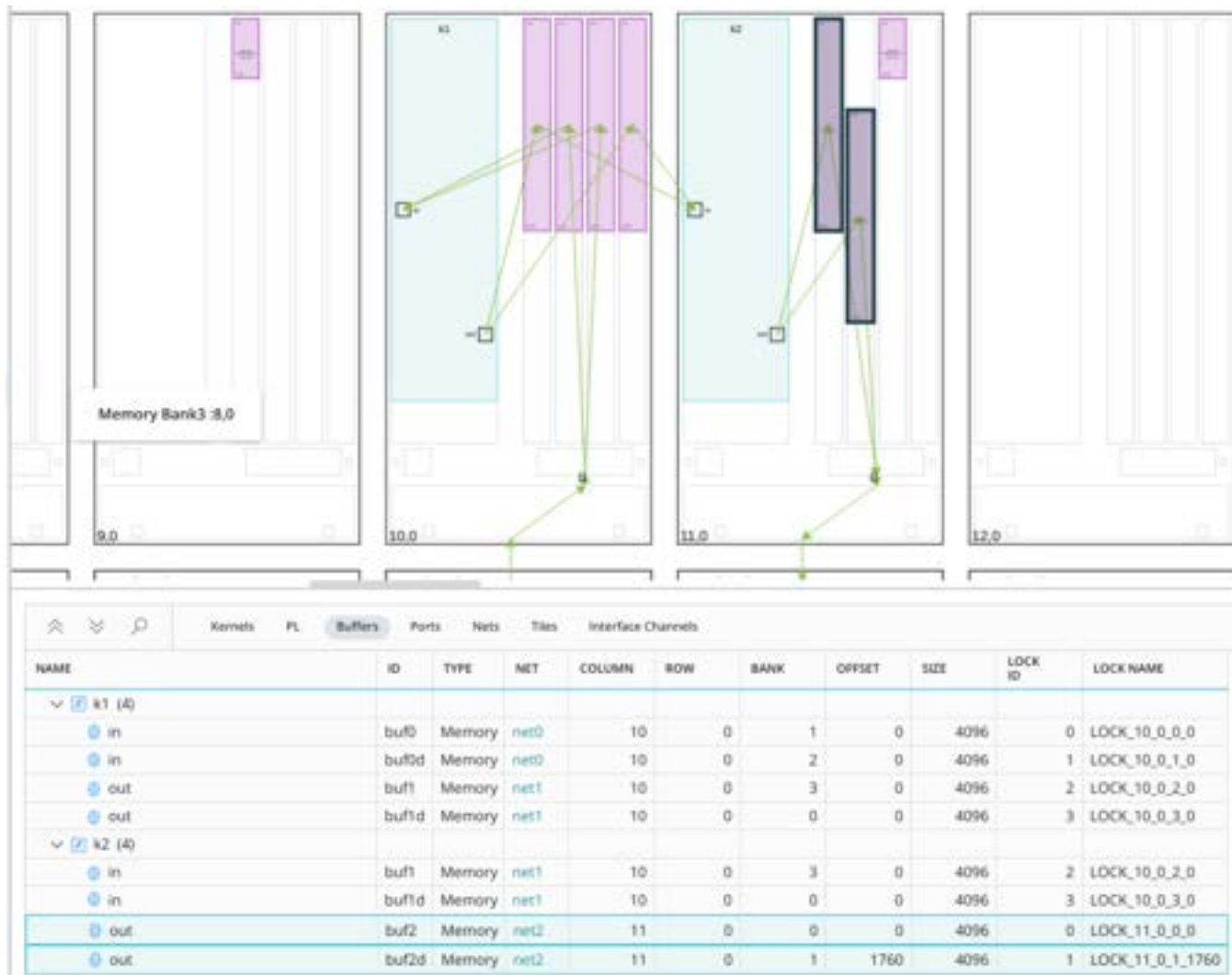
When the source of a buffer is a stream, this stream is sliced into contiguous blocks which are stored one by one into buffers, as illustrated below.

Figure: Data Stream Slicing Into Buffers



An example of kernel buffer port that resides at local tile memory is shown in the following figure. The kernel k1 is in tile (10,0) and input buffer port is allocated in the local memory of the same tile (10,0).

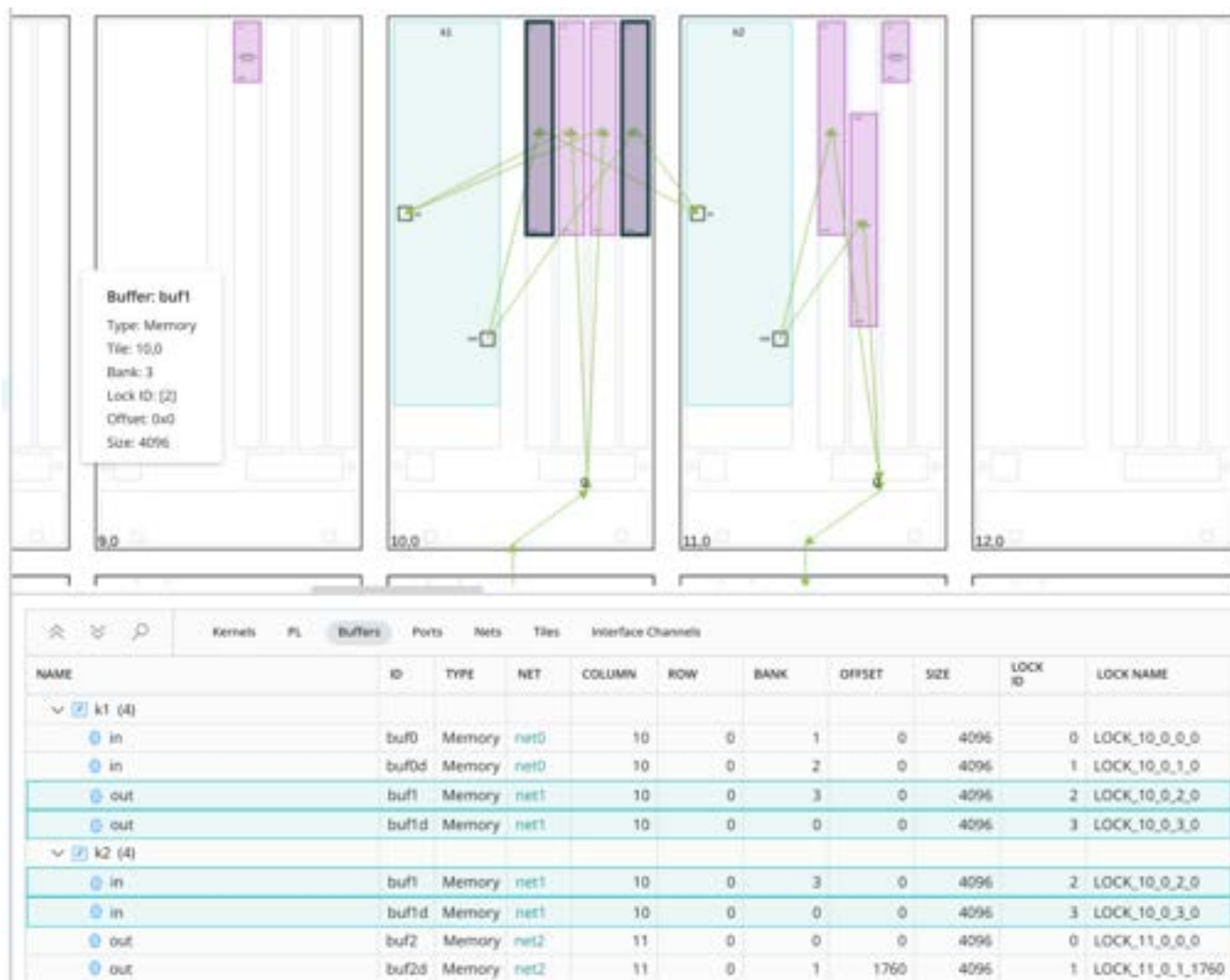
Figure: Kernel Buffer Port in Local Tile Memory



An example of a kernel buffer port that is allocated in a neighboring tile memory is shown in the following figure. The kernel k2 is in tile (11,0) and input buffer port at neighbor tile (10,0).

Note: Buffers can be placed automatically by the tools as directed with user-specified design constraints. A single buffer port (including both Ping and Pong buffers) must be placed in a single tile. For AI Engine devices, a single buffer port is no larger than 32 KB.

Figure: Kernel Buffer Port in Neighboring Tile Memory



Buffer Port-Based Access

Buffer ports provide a way for a kernel to operate on a block of data. Buffer ports operate in a single direction (e.g., input or output). The view that a kernel has of incoming blocks of data is called an input buffer. Input buffers are defined by a type. The type of data contained within that buffer needs to be declared before the kernel can operate on it.

The view that a kernel has of outgoing blocks of data is called an output buffer. These are defined by a type. The example below shows a declaration of a kernel named `simple`. The `simple` kernel has an input buffer named `in`, which contains complex integers where both the real and the imaginary parts are each 16-bit wide signed integers. The `simple` kernel also has an output buffer named `out`, which contains 32-bit wide signed integers.

```
void simple(input_buffer<cint16> & in,
           output_buffer<int32> & out);
```

The example below shows input and output buffer port sizes declaration using the `adf::extents` template parameter.

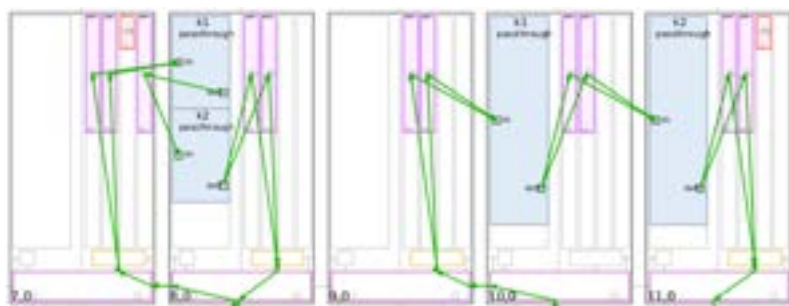
```
void simple(input_buffer<cint16, adf::extents<INPUT_SAMPLE_SIZE>> & in,
           output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out);
```

These buffer data structures are inferred by the AI Engine compiler from the data flow graph connections and are declared in the wrapper code implementing the graph control. The kernel functions merely operate on pointers to the buffer data structures that are passed to them as arguments. There is no need to declare these buffer data structures in the data flow graph or kernel program.

When two kernels (`k1`, `k2`) communicate through buffers (the output buffer of `k1` is connected to the input buffer of `k2`) the compiler attempts to place them into tiles that can share at least an AI Engine memory module.

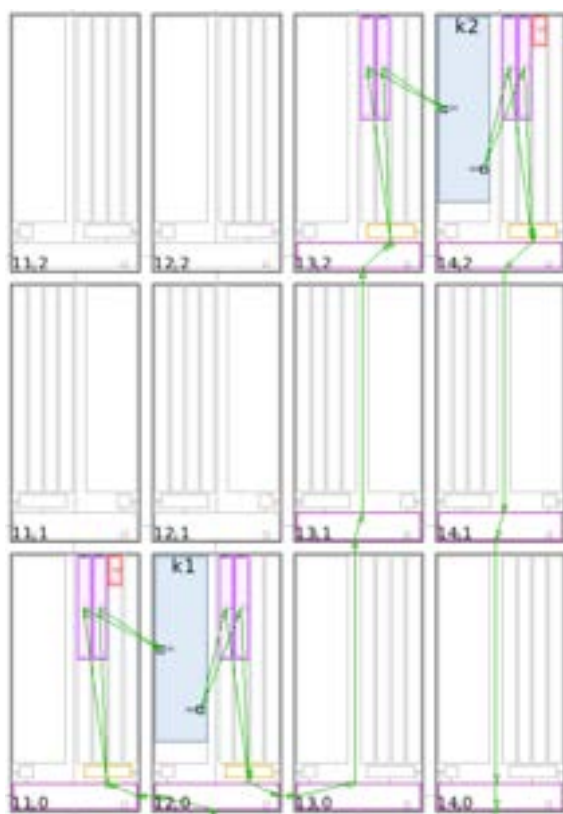
- If the two kernels are located on the same tile, the compiler uses a single memory area to communicate because they are not executed simultaneously (see k1 and k2 in tile (8,0) and the single shared memory block in (7,0) in the following figure). Because the execution of multiple kernels within an AI Engine is sequential, access conflicts are avoided when using the same memory area.
- If the two kernels are placed in different tiles sharing an AI Engine memory module, the compiler will infer a ping-pong buffer, allowing the two kernels to write and read at the same time but not to the same memory area (see k1 in tile (10,0), k2 in tile (11,0) and the shared buffer implemented as a ping-pong buffer in (10,0) in the following figure).

Figure: Same Tile and Memory Sharing Placement Example



- If your system performance can handle it, you can switch this ping-pong buffer into single buffering by applying the `single_buffer(<port>)` constraint to the kernel ports.
- If the two kernels are placed in distant tiles, the compiler will automatically infer a ping-pong buffer at the output of k1, and another one at the input k2. The two ping-pongs are connected with a DMA which will automatically copy the content of the output buffer of k1 onto the input buffer of k2 using a data stream.

Figure: Distant Tiles Placement Example



- When multiple buffers/streams converge onto a kernel, the various paths can have very different latencies, which can potentially lead to a deadlock. To avoid this kind of problem, you can insert a FIFO between the two kernels. The compiler will generate the same type of architecture as the distant tile case, except that a FIFO is inserted in the middle of the stream connection.

Buffer Port Attributes and API

Attributes on buffer ports that can be configured are the buffer port's size, margin, locking protocol (synchronous or asynchronous), addressing

mode (linear or circular), and single buffer versus ping-pong-buffer. Of these attributes, margin, locking protocol and addressing mode are specified in the kernel function signature. Single buffer versus. ping-pong buffer is specified in the graph.

The size of a buffer port can be specified either in the kernel signature or in the graph constructor. If the kernel function signature does *not* specify the size or if it is specified to be `adf::inherited_extent`, then the size must be specified in the graph using the `dimensions()` API. The `dimensions()` API is used in the constructor of the graph and configures buffer port size in terms of the number of samples. `adf::inherited_extent`, represents the size of the buffer which is inferred from the context, for example, it is inferred from the specification in the graph.

The buffer port margin represents the number of samples copied from the end of one block to the beginning of the next one. If a margin parameter is specified, the total memory allocated is the sum of sample size and margin size, multiplied by template data type in number of bytes ($(\text{sample size} + \text{margin size}) * \text{sizeof}(\text{data type})$). In the case of a circular output buffer, the margin can be specified as `adf::inherited_margin`. In this situation, the margin size is specified by the sink connected to this output port.

An example function prototype specifying margin size follows:

```
simple(input_buffer<int32, adf::extents<adf::inherited_extent>,
      adf::margin<MARGIN_SIZE>> & in0,
      output_buffer<int32> & out0);
```

Note:

- Buffer port types define their dimensions and margin in number of samples of data. To determine the actual data size stored in physical memory, you need to multiply the total number of samples with supported data type in bytes.
 - Buffer size and margin size must be a multiple of 16 bytes.
 - Constructs related to I/O buffers are defined in the `adf` namespace. Therefore, these constructs need to be qualified with `adf::`, unless you include `using namespace adf;`
-

Data Access Mechanisms

Synchronous Buffer Port Access

A kernel reads from its input buffers and writes to its output buffers. By default, the synchronization that is required to wait for an input buffer of data is performed before entering the kernel. The synchronization that is required to provide an empty output buffer is also performed before entering the kernel. There is no synchronization needed for synchronous buffer ports within the kernel to read or write the individual samples of data after the kernel has started execution.

Buffer port size can be declared via `dimensions()` API or with kernel function prototype.

Option 1

Configure with `dimensions()` API in graph.

```
connect netN(in.out[0], k.in[0]);
dimensions(k.in[0])={INPUT_SAMPLE_SIZE};
```

Option 2

Configure with kernel function prototype where function prototypes are declared in kernel header files and which is referenced in graph code.

Graph code specifies connection.

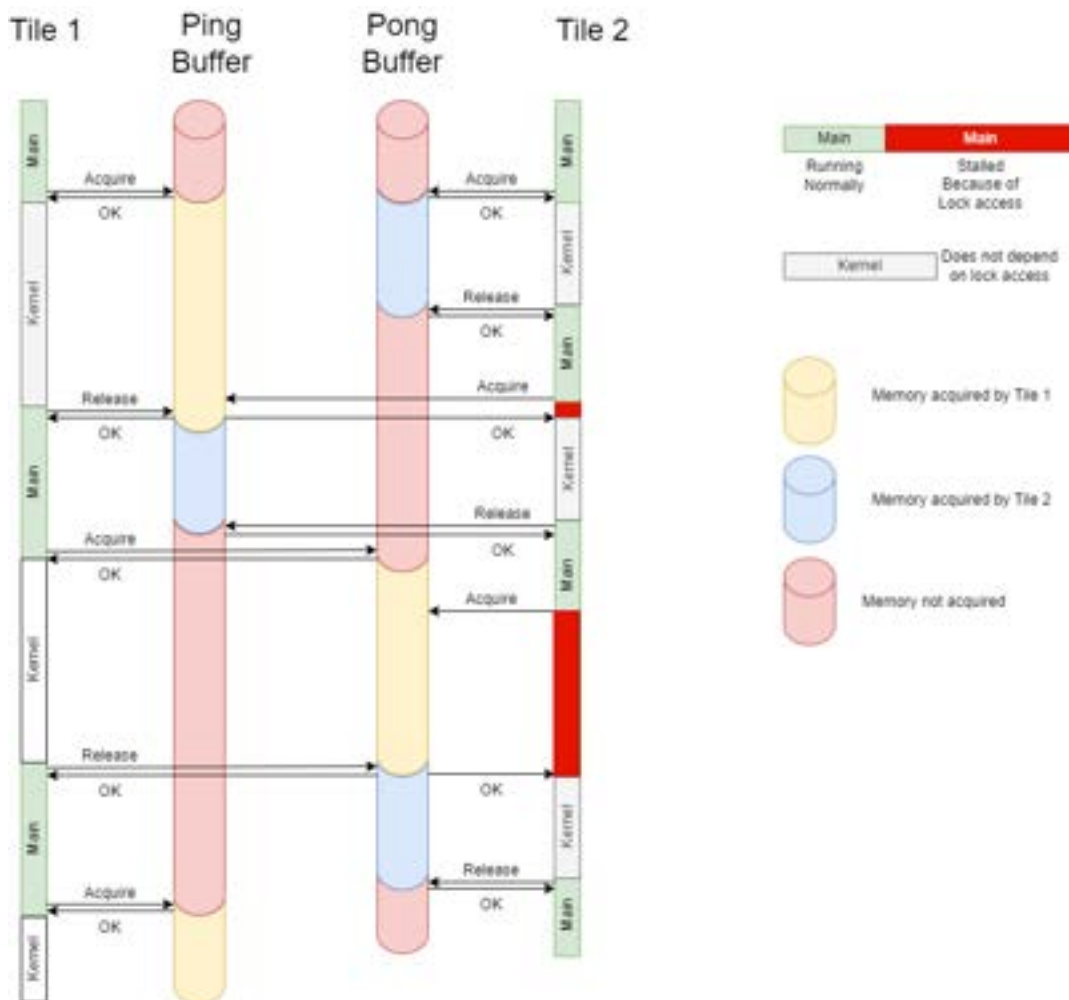
```
connect netN(in.out[0], k.in[0]);
```

Kernel code specifies data type and buffer size.

```
void simple(input_buffer<int32, adf::extents<INPUT_SAMPLE_SIZE>> & in,
            output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out);
```

In the following example, a kernel located in tile 1 uses a ping-pong buffer for writing, and the kernel located in tile 2, which is adjacent to tile 1, uses the same ping-pong buffer for reading. The two kernels and two main functions do not have the same execution time leading to some processor stalling during the runtime. The overall mechanism is that kernel 1 writes onto the ping buffer while kernel 2 reads from pong buffer. In the following figure, iteration kernel 1 writes onto the pong buffer, and kernel 2 reads from the ping buffer.

Figure: Lock Mechanism For Synchronous Ping-pong Buffer Access



The kernel's buffer lock mechanism is handled in the tiles main function. The kernel starts only when all input and output buffers are locked for reading and writing respectively. The minimum latency for a lock acquisition is seven clock cycles if the buffer is ready to be acquired. If it's already locked by another kernel, it stalls until it becomes available (indicated in red in the figure).

You can see in the diagram that the lock acquisition occurs alternatively on the ping then pong buffer. The selection of the ping or pong buffer is automatic, no user decision is needed at this point.

When a synchronous buffer port is used, the lock on the output buffer of the kernel is released after the kernel execution has finished. This buffer can then either be acquired by its consumer kernel or transferred by DMA to its destination (destination such as a PLIO).

!! Important: The synchronous output buffer will be acquired and released during every iteration of the kernel, regardless of the number of samples written into the buffer by the kernel.

Synchronous Buffer Port With Margin Configuration

Option 1

Configure with `dimensions()` API in graph.

```
k = kernel::create(simple);
connect netN(in.out[0], k.in[0]);
dimensions(k.in[0]) = {INPUT_SAMPLE_SIZE};
dimensions(k.out[0]) = {OUTPUT_SAMPLE_SIZE};
```

Kernel `k`'s function prototype specifies margin size.

Note: The margin cannot be specified in the graph. The margin must be set in the kernel signature within the template parameters of the buffer.

```
simple(input_buffer<int32, adf::extents<adf::inherited_extent>, adf::margin<MARGIN_SIZE>> & in0,
      output_buffer<int32> & out0);
```

Option 2

Configure with kernel function prototype.

```
k = kernel::create(simple);
connect netN(in.out[0], k.in[0]);
```

Kernel k's function prototype specifies input buffer size plus margin size and output buffer size.

```
simple(input_buffer<int32, adf::extents<INPUT_SAMPLE_SIZE>, adf::margin<MARGIN_SIZE>> & in0,
      output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0);
```

The buffer ports are designed to be accessed sequentially. The kernel program reads the buffer port and starts from the first position of samples. Therefore, a useful model is that of a current position, which can be advanced or rolled back on reads or writes. On starting a kernel, the current position is always in the correct position. For example, the current position for an input buffer port for a filter is on the first sample to restore the delay line. It could be an older sample in the case of filters requiring overlap of incoming data samples, in which case the connection needs to be declared using the overlap or margin as described above. Similarly, the current position for an output buffer is on the first sample to send to the next block, irrespective of whether that block requires a duplication of older samples. The kernel is free to manipulate this current position and it is not necessary that this position is at the end of the block when the kernel completes. Buffer ports are implemented as linear buffers. Circular buffer port types are also supported. For more information, see [Linear and Circular Addressing of Buffer Ports](#).

Buffer Port Multicasting

The AI Engine compiler does not limit the buffer ports to a one-to-one connection. In certain circumstances the same output buffer can be used by other kernels to perform various tasks. You can connect a producer to as many consumers as needed. The AI Engine compiler will automatically infer a MM2S DMA to read the output buffer and as many S2MM DMAs as there are consumers to write to their respective input buffers.

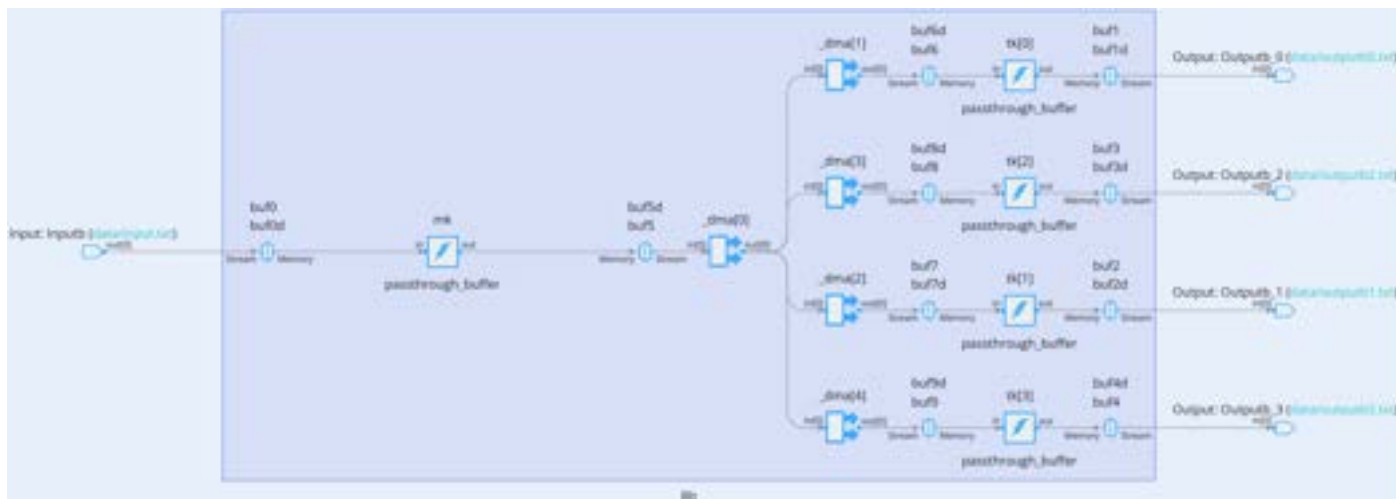
```
private:
adf::kernel mk;
adf::kernel tk0,tk1,tk2,tk3;
...
connect net0 ( mk.out[0] , tk0.in[0] );
connect net1 ( mk.out[0] , tk1.in[0] );
connect net2 ( mk.out[0] , tk2.in[0] );
connect net3 ( mk.out[0] , tk3.in[0] );
...
dimensions(tk0.in[0]) = {128};
dimensions(tk1.in[0]) = {128};
dimensions(tk2.in[0]) = {128};
dimensions(tk3.in[0]) = {128};
```

Kernel function prototypes:

```
tk0(input_buffer<int32, adf::extents<adf::inherited_extent>> & in0,
     output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0);
tk1(input_buffer<int32, adf::extents<adf::inherited_extent>,
     adf::margin<32>> & in0,
     output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0);
tk2(input_buffer<int32, adf::extents<adf::inherited_extent>,
     adf::margin<64>> & in0,
     output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0);
tk3(input_buffer<int32, adf::extents<adf::inherited_extent>> & in0,
     output_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0);
```

The input buffer to kernels tk0, tk1, tk2, and tk3 are served at the same time. This is because the output buffer of the kernel mk is read only once. The slight delay variation is only due to the different AXI4-Stream path taken to route from the maker to the various takers in the AI Engine array.

Figure: One Kernel Serving Four Kernels



In the code, the same kernel output is connected to four different kernel inputs. The AI Engine compiler adds DMAs between kernels so that the content of buf5(d) buffer can be copied into the other buffers using the AXI4-Stream interconnect network.

Buffer Port Connection for Multirate Processing

In some designs the size of the output buffer port of a kernel can be different than the size of the input buffer port of the next kernel. In that case, the connection declaration will contain the size of the output port and the size of the input buffer port. For example:

```
connect net0 (kernel0.out[0], kernel1.in[0]);
dimensions(kernel0.out[0]) = {128};
dimensions(kernel1.in[0]) = {192};
```

In the above example, kernel0 writes 128 samples and kernel1 expects 192 samples will be written to memory. In such scenarios, the AI Engine compiler performs multirate analysis. In this case, the compiler will specify that kernel1 should run twice while kernel0 will run three times. You can specify the repetition count for these kernels in the graph manually, as follows:

```
repetition_count(kernel0) = 3;
repetition_count(kernel1) = 2;
```

Just as you can multicast an output buffer port to multiple input buffer ports in the graph (automatic DMA insertion mechanism), you can also perform multirate processing in this specific use case:

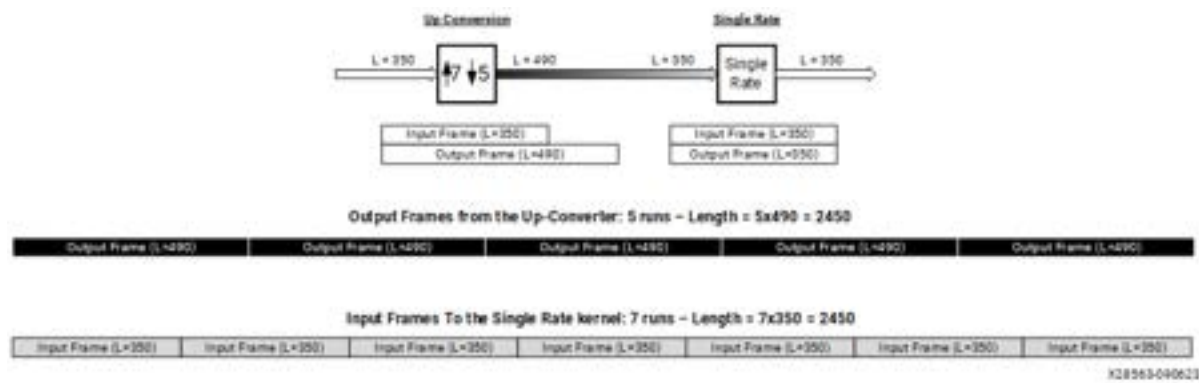
```
connect net0 ( kernel0.out[0] , kernel1.in[0] );
connect net1 ( kernel0.out[0] , kernel2.in[0] );
dimensions(kernel0.out[0]) = {128};
dimensions(kernel1.in[0]) = {64};
dimensions(kernel2.in[0]) = {192};
```

In this example, the AI Engine compiler automatically detects that kernel0 should run three times, kernel1 should run six times, and kernel2 should run twice for one graph iteration, `graph.run(1)`. These repetition counts can also be specified manually in the graph. The following is a graphical example with a rational up-converter (7/5) followed by a single rate kernel. Input frame length is 350 for both, but the output frame length of the up-converter is 490.

```
connect ( datain , upconv.in[0] );
connect ( upconv.out[0] , singlerate.in[0] );
connect ( singlerate.out[0] , dataout );

// If the connection is buffer based dimensions can be specified
// in the graph
dimensions(upconv.in[0]) = {350};
dimensions(upconv.out[0]) = {490};
dimensions(singlerate.in[0]) = {350};
dimensions(singlerate.out[0]) = {350};

// If the connections are stream based, repetitions count must be specified
repetition_count(upconv) = 5;      // LCM(350,490)/490 = 5
repetition_count(singlerate) = 7;  // LCM(350,490)/350 = 7
```


Figure: Up-Converter Followed by a Single Rate Kernel

In the preceding figure, the up-conversion kernel must be run five times, and the single rate kernel must be run seven times to produce and consume the same number of samples between the two kernels.

In this example the output frame length of the up-converter is larger than the length of the input frame of the single rate kernel. This means that the up-converter will write over the ping and pong buffers (or other more complex schemes) in a single iteration.

Asynchronous Buffer Port Access

In some situations, if you are not consuming a buffer port worth of data on every invocation of a kernel, or if you are not producing a buffer port worth of data on every invocation, then you can control the buffer synchronization by declaring the kernel port using `async` to declare the `async` buffer port in kernel function prototype. The example below illustrates that the kernel `simple` uses:

ifm
Synchronous input buffer port.

wtS
Asynchronous input buffer port.

ofm
Asynchronous output buffer port.

The declaration below informs the compiler to omit synchronization of the buffer named `wtS` upon entry to the kernel. You must use buffer port synchronization member function shown *inside* the kernel code before accessing the buffer port using read/write iterators/references, as shown below.

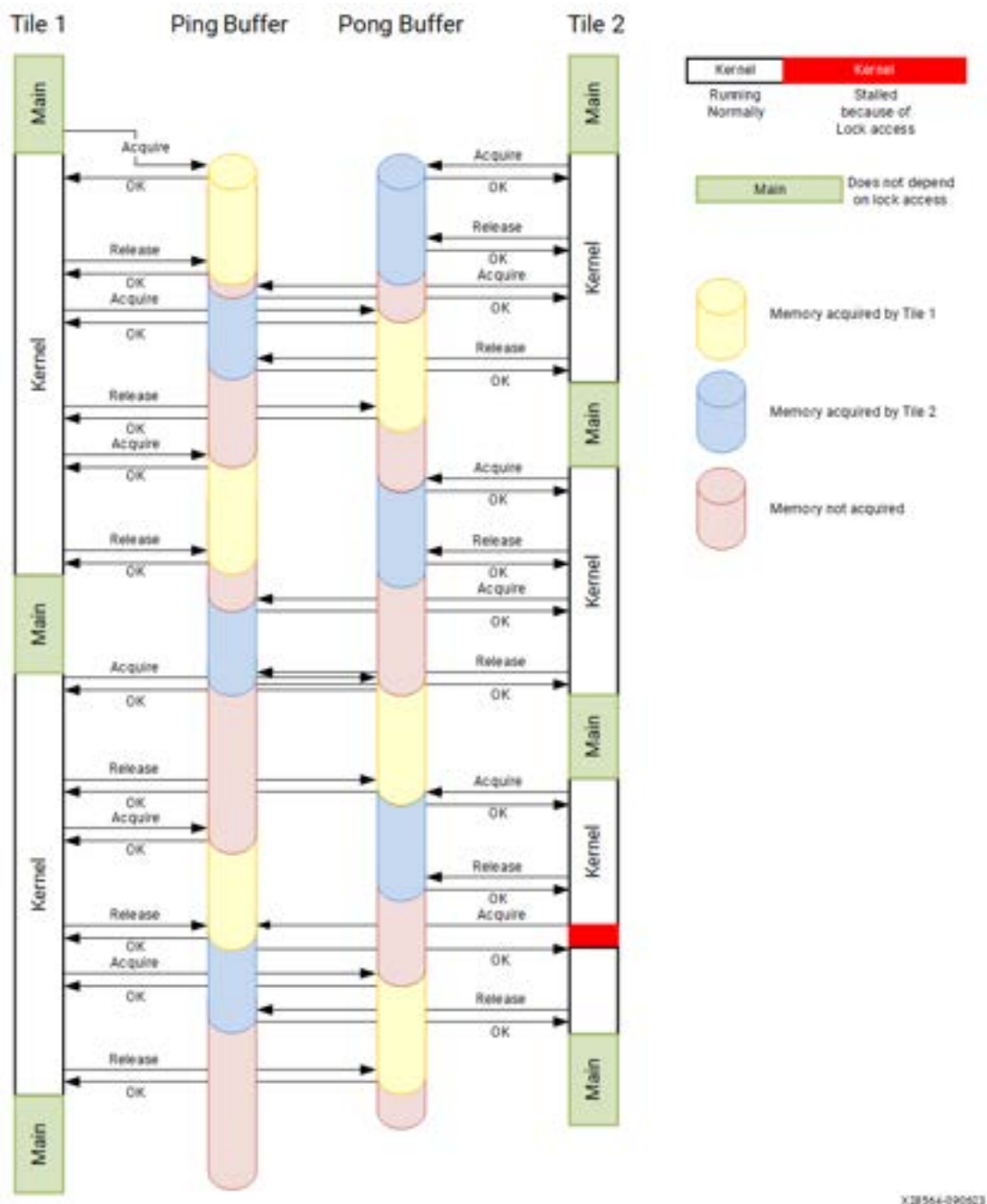
```
void simple(adf::input_buffer<uint8>& ifm, adf::input_async_buffer<uint8>& wtS,
adf::output_async_buffer<uint8>& ofm)
{
    ...
    wtS.acquire(); // acquire lock unconditionally inside the kernel
    if (<somecondition>) {
        ofm.acquire(); // acquire output buffer conditionally
    }
    ... // do some computation
    wtS.release(); // release input buffer port inside the kernel
    if (<somecondition>) {
        ofm.release(); // release output buffer port conditionally
    }
    ...
};
```

The `acquire()` member function of the buffer object `wtS` performs the appropriate synchronization and initialization to ensure that the buffer port object is available for read or write. This function keeps track of the appropriate buffer pointers and locks to be acquired internally, even if the buffer port is shared across AI Engine processors and can be double buffered. This function can be called unconditionally or conditionally under dynamic control and is potentially a blocking operation. It is your responsibility to ensure that the corresponding `release()` member function is executed sometime later (possibly even in a subsequent kernel call) to release the lock associated with that buffer object. Incorrect synchronization can lead to a deadlock in your code.

!! Important: Operations on asynchronous buffer should be done after the buffer is acquired. For example, declare the buffer iterator after the `acquire()` API.

In the following example, the kernel located in tile 1 requests a lock acquisition (write access) three times per each run. The kernel located in tile 2 requests a lock acquisition (read access) twice per each run.

Figure: Lock Mechanism for Asynchronous Ping-pong Buffer Access



The lock acquisition and release is a kernel-only process. The main function is not taking care of the buffer synchronization; buffer synchronization is the user responsibility. Kernel in tile 1 requests three times the access to the ping pong buffer and tile 2 only twice. In order to balance the number of accesses, tile 1 should be run twice, and tile 2 should be run three times per iteration.

As seen in the figure, the lock acquisition occurs alternatively on the ping then pong buffer. The buffer choice is automatic. No user decision is needed at this point.

The minimum latency for lock acquisition is seven clock cycles during which the kernel is stalled. If the buffer is not available for acquisition, the kernel is stalled for a longer time (as indicated in red in the figure) until the buffer is available. Depending on the application, there might be time intervals where the ping and/or the pong buffer might not be locked at all.

For asynchronous buffer port, the buffer port of the kernel is acquired and released explicitly by the `acquire` and `release` APIs. The asynchronous output buffer can be released anytime inside the kernel by the `release` API, no matter how many samples are written into the buffer by the kernel. After the port is released, the asynchronous output buffer can be acquired by its consumer kernel or can be transferred by DMA to its destination, such as PLIO.

Consider a system with one producer AI Engine kernel and one consumer AI Engine kernel, communicating via asynchronous buffers. Initially, there are two empty buffers between the producer and the consumer.

From the *producer's* perspective:

Each time the producer wants to write data to a buffer, it must first call the `acquire` API. When acquired, the buffer is owned by the producer, which can read from or write to it as needed. After finishing the operation—either in the same iteration or later—it must call the `release` API to

release the buffer. Once released, the buffer becomes available to the consumer, increasing the count of full buffers. If both buffers are full, any subsequent `acquire` call by the producer will block until an empty buffer becomes available.

From the *consumer's* perspective:

The consumer must also call the `acquire` API before accessing a buffer. After acquiring, it owns the buffer and can read from or write to it. Once finished, it calls the `release` API to release the buffer, making it available for the producer again and increasing the count of empty buffers. If both buffers are empty, the consumer will be stalled upon trying to acquire a buffer, until one becomes full.

In this system, *PLIO* or *GMIO* can also act as producers or consumers. Data exchange between PLIO/GMIO and the AI Engine is managed by DMA, which handles buffer availability transparently. Data can only be sent or received when the corresponding buffer is ready (that is, empty for writing or full for reading).

Buffer Port Data Types

Table: Supported Buffer Port Data Types

Input Buffer Port Types	Output Buffer Port Types
<code>input_buffer<int8></code>	<code>output_buffer<int8></code>
<code>input_buffer<int16></code>	<code>output_buffer<int16></code>
<code>input_buffer<int32></code>	<code>output_buffer<int32></code>
<code>input_buffer<int64></code>	<code>output_buffer<int64></code>
<code>input_buffer<uint8></code>	<code>output_buffer<uint8></code>
<code>input_buffer<uint16></code>	<code>output_buffer<uint16></code>
<code>input_buffer<uint32></code>	<code>output_buffer<uint32></code>
<code>input_buffer<uint64></code>	<code>output_buffer<uint64></code>
<code>input_buffer<cint16></code>	<code>output_buffer<cint16></code>
<code>input_buffer<cint32></code>	<code>output_buffer<cint32></code>
<code>input_buffer<float></code>	<code>output_buffer<float></code>
<code>input_buffer<cfloat></code>	<code>output_buffer<cfloat></code>

Access Data From a Buffer Port

Buffer Port Iterators

For buffer port data access, an iterator is supported to perform read or write operations. Additionally, vector based iterators are supported in buffer ports.

An iterator is an object that can iterate over data in a buffer and provide access to individual data. Buffer port objects support multiple iterator objects depending on the buffer port type.

Scalar Iterators

Iterator `begin`

A bidirectional scalar iterator used to return an iterator pointing to the first sample of the container.

Iterator `cbegin`

A bidirectional scalar iterator that points to the first `const` content. It cannot be used to modify the contents it points to.

Iterator `begin_circular`

Same as `iterator begin` but is used for circular buffer port.

Iterator `cbegin_circular`

Same as `iterator begin_circular` but points to `const` content.

Iterator `begin_random_circular`

Same as `begin_circular` iterator and can be used to access samples at an arbitrary offset position relative to the sample they point to.

Iterator `cbegin_random_circular`

Same as `begin_random_circular` iterator but cannot be used to modify the contents it points to.

Vector Iterators

Iterator `begin_vector`

A bidirectional vector iterator used to return an iterator pointing to the first vector sample of the container.

Iterator `cbegin_vector`

A bidirectional vector iterator that points to `const` content. it cannot be used to modify the contents it points to.

Iterator `begin_vector_circular`

Same as iterator `begin_vector` but is used for circular buffer port.

Iterator `cbegin_vector_circular`

Same as iterator `begin_vector_circular` but points to 'const' content.

Iterator `begin_vector_random_circular`

Same as iterator `begin_vector_circular` and can be used to access samples at an arbitrary offset position relative to the sample they point to. The arbitrary offset is in unit of configured vector size.

Iterator `cbegin_vector_random_circular`

Same as iterator `begin_vector_random_circular` but cannot be used to modify the contents it points to.

Table: List of Supported Buffer Port Vector Iterators

Iterator API	Description
<code>auto begin_vector</code>	Random scalar iterator; not supported for circular buffer ports.
<code>auto cbegin_vector</code>	Same as <code>begin_vector</code> but only for read access.
<code>auto begin_vector_circular</code>	Forward scalar circular iterator.
<code>auto cbegin_vector_circular</code>	Same as <code>begin_vector_circular</code> but only for read access.
<code>auto begin_vector_random_circular</code>	Random scalar circular iterator.
<code>auto cbegin_vector_random_circular</code>	Same as <code>begin_vector_random_circular</code> but only for read access.

Examples

Example of `begin` Iterator

```
#include "aie_api/aie.hpp"
void simple(adf::input_buffer<cint16, adf::extents<BUFFER_SIZE>> & in, adf::output_buffer<cint16,
adf::extents<BUFFER_SIZE>> & out)
{
    // Use scalar iterator to traverse data
    auto pIn = aie::begin(in);
    auto pOut = aie::begin(out);

    // For loop to go through all data from input_buffer via iterator
    for (unsigned i=0; i<(BUFFER_SIZE); i++)
    {
        *pOut++ = *pIn++;
    }
}
```

Example of `begin_vector` Iterator

```
#define VECTOR_SIZE 8
void simple(adf::input_buffer<cint16, adf::extents<BUFFER_SIZE>> & in, adf::output_buffer<cint16,
adf::extents<BUFFER_SIZE>> & out)
{
    // Use vectoriterator to traverse data
    auto pIn = aie::begin_vector<VECTOR_SIZE>(in);
    auto pOut = aie::begin_vector<VECTOR_SIZE>(out);
```

```

// For loop to go through all data from input_buffer via iterator
for (unsigned i=0; i<(BUFFER_SIZE/VECTOR_SIZE); i++)
{
    *pOut++ = *pIn++;
}
}

```

Example of cbegin Iterator

```

void simple(adf::input_buffer<cint16, adf::extents<BUFFER_SIZE>> & in, adf::output_buffer<cint16,
adf::extents<BUFFER_SIZE>> & out)
{
    // Use scalar iterator to traverse data
    auto pIn = aie::cbegin(in);
    auto pOut = aie::begin(out);

    // For loop to go through all data from input_buffer via iterator
    for (unsigned i=0; i<(BUFFER_SIZE); i++)
    {
        *pOut++ = *pIn++;
    }
}

```

 **Note:** cbegin iterator is a read-only iterator. The AI Engine compiler errors out if the pOut iterator is declared as a cbegin iterator.

Example of cbegin_vector Iterator

```

#define VECTOR_SIZE 8
void simple(adf::input_buffer<cint16, adf::extents<BUFFER_SIZE>> & in, adf::output_buffer<cint16,
adf::extents<BUFFER_SIZE>> & out)
{
    // Use vector iterator to traverse data
    auto pIn = aie::cbegin_vector<VECTOR_SIZE>(in);
    auto pOut = aie::begin_vector<VECTOR_SIZE>(out);

    // For loop to go through all data from input_buffer via vector iterator
    // The buffer contains (BUFFER_SIZE/VECTOR_SIZE) vectors
    for (unsigned i=0; i<(BUFFER_SIZE/VECTOR_SIZE); i++)
    {
        *pOut++ = *pIn++;
    }
}

```

Example of begin_random_circular Iterator

```

void simple(adf::input_circular_buffer<cint16, adf::extents<BUFFER_SIZE>> & in,
adf::output_circular_buffer<cint16, adf::extents<BUFFER_SIZE>> & out)
{
    // Use scalar iterator to traverse data
    auto pIn = aie::begin_random_circular(in);
    auto pOut = aie::begin_random_circular(out);

    // Position the pointer at the middle of the buffer
    pIn += BUFFER_SIZE/2;
    // Copies the second half, then the first half of the buffer onto the output
    for (unsigned i=0; i<(BUFFER_SIZE); i++)
    {
        *pOut++ = *pIn++;
    }
}

```

Example of begin_vector_random_circular Iterator

```
#define VECTOR_SIZE 8
void simple(adf::input_circular_buffer<cint16, adf::extents<BUFFER_SIZE>> & in,
adf::output_circular_buffer<cint16, adf::extents<BUFFER_SIZE>> & out)
{
    // Use vector iterator to traverse data
    auto pIn  = aie::begin_vector_random_circular<VECTOR_SIZE>(in);
    auto pOut = aie::begin_vector_random_circular<VECTOR_SIZE>(out);

    // Position the pointer at the end of the buffer
    pIn += BUFFER_SIZE/VECTOR_SIZE;
    // Copies the input buffer onto the output buffer
    for (unsigned i=0; i<(BUFFER_SIZE/VECTOR_SIZE); i++)
    {
        *pOut++ = *pIn++;
    }
}
```

- Circular buffer ports must use circular iterators. Linear buffer ports can use linear iterators or circular iterators.

Warning: If there are two kernels connected by buffers (output from the first kernel, and input to the second kernel), the kernels should have either linear or circular addressing if the two kernels are mapped to the same tile. If the buffers are not configured for linear or circular addressing, *x86 Simulation* will give correct simulation output while *AI Engine Simulation* will be wrong due to some optimization in memory organization that are performed in hardware implementation.

- Use vector iterators to access VECTOR_SIZE samples for each iteration. Where VECTOR_SIZE is 4 (128 bits), 8 (256 bits), 16 (512 bits), and 32 (1024 bits) for this specific example where the data-type is cint16 (32 bits).
- Use random iterators when iterators need to be moved more than one step at a time in either direction.

Reading and Writing Data

There are several ways of reading and writing data to a one dimensional buffer port.

- Using a raw pointer. Be aware that this mechanism must not be used with circular buffer ports.

```
void simple(adf::input_buffer<int32> & in, adf::output_buffer<int32> & out) {
    int32 * pin = in.data();
    int32 * pout = out.data();
    for (int i = 0; i < BUFFER_SIZE; i++) {
        *pout++ = *pin++;
    }
    ...
}
```

- Using a scalar iterator.

```
void simple(adf::input_buffer<int32> & in, adf::output_buffer<int32> & out) {
    auto pin = aie::begin(in);
    auto pout = aie::begin(out);
    for (int i = 0; i < BUFFER_SIZE; i++) {
        *pout++ = *pin++;
    }
    ...
}
```

- Using a vector iterator.

```
void simple(adf::input_buffer<int32> & in, adf::output_buffer<int32> & out) {
    auto pin = aie::begin_vector<VECTOR_SIZE>(in);
    auto pout = aie::begin_vector<VECTOR_SIZE>(out);
    for (int i = 0; i < BUFFER_SIZE/VECTOR_SIZE; i++) {
        *pout++ = *pin++;
    }
    ...
}
```

Using Input and Output Buffer as Intermediate Storage

After acquiring an input or output buffer but before releasing it, the buffer is owned by the kernel. The kernel can be responsible to read or write to the buffer by pointer or iterator without conflicting the data. The following code shows an example of an asynchronous output buffer being used for temporary storage between iterations:

```
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include <aie_api/utils.hpp>
const int BUFFER_SIZE=1024;
const int VECTOR_SIZE=16;
const int TOTAL_N=2;
static int iteration=0;
__attribute__((noinline)) void accumulation(adf::input_buffer<int32, extents<BUFFER_SIZE>> & __restrict in1,
    adf::input_buffer<int32, extents<BUFFER_SIZE>> & __restrict in2,
    adf::output_async_buffer<int32, extents<BUFFER_SIZE>> & __restrict out
){
    auto pin1 = aie::begin_vector<VECTOR_SIZE>(in1);
    auto pin2 = aie::begin_vector<VECTOR_SIZE>(in2);

    if(iteration==0){
        out.acquire();
        //must be done after lock acquisition
        auto pout = aie::begin_vector<VECTOR_SIZE>(out);
        for (int i = 0; i < BUFFER_SIZE/VECTOR_SIZE; i++) {
            *pout++ = aie::add(*pin1++, *pin2++);
        }
        iteration++;
    }else{
        auto pout=aie::begin_vector<VECTOR_SIZE>(out); //lock acquired
        for (int i = 0; i < BUFFER_SIZE/VECTOR_SIZE; i++) {
```

```

        auto tmp = aie::add(*pin1++, *pin2++);
        auto tmp2=*pout;
        *pout++ = aie::add(tmp2, tmp);
    }
    iteration++;
}
if(iteration==TOTAL_N){
    iteration=0;
    out.release();
}
}

```

Linear and Circular Addressing of Buffer Ports

Buffer ports can be addressed in a linear or circular manner. In linear addressing mode, data is addressed linearly and has no wrap around. In circular addressing mode, data is addressed circularly and has wrap around. Kernels with margin in the same tile are recommended to use circular buffer ports. Port connections with non-zero margin require copying the margin data at the consumer port after the consumer kernel returns. However, if the producer and consumer are on the same core and you use circular buffer ports, then margin copy can be avoided, which improves performance. Data in a regular buffer port can be accessed with a pointer obtained from the `data()` API, or using linear or circular iterators. Linear iterators have lower overhead than circular iterators.

Note:

- Circular addressing mode is supported only in one dimension buffer ports.
- The iterator to access the data in a circular buffer port must be a circular iterator.
- Circular buffer port can only be declared in the function prototype.

Circular Input Buffer

The following example declares the kernel function `k2` with the input circular buffer `in0` that operates on `int32` data type and an output stream that operates on data type `int32` that is named `out0`.

```

void k2(input_circular_buffer<int32, adf::extents<INPUT_SAMPLE_SIZE>, adf::margin<MARGIN_SIZE>> & in0,
output_stream<int32> *out0)
{
    auto in0Iter = aie::begin_circular(in0);
    for (int ind = 0; ind < (INPUT_SAMPLE_SIZE + MARGIN_SIZE); ++ind)
    {
        writeincr(out0, *in0Iter++);
    }
}

```

Circular Output Buffer

Declare kernel function `k1` with input stream operates on data type `int32` that is named `in0` and circular 1d output buffer operates on data type `int32` that is named `out0`.

```

void k1(input_stream<int32> *in0, output_circular_buffer<int32, adf::extents<OUTPUT_SAMPLE_SIZE>> & out0)
{
    auto out0Iter = aie::begin_circular(out0);
    for (int ind = 0; ind < OUTPUT_SAMPLE_SIZE; ++ind)
    {
        *out0Iter++ = readincr(in0);
    }
}

```

!! Important: When kernels are located on the same tile in the AI Engine, memory addressing optimizations become possible. However, it is important to note that some restrictions might arise due to this optimization. For instance, if `K1` and `K2` are situated in the same tile and are connected through an output-to-input buffer, they must use the same type of addressing. This means they can either opt for linear or circular addressing, but they cannot have a mixture of both. However, it is worth noting that this limitation will not be identified during X86 simulation because it only emulates functional aspects of AI Engine tiles and memory. For more details on X86 simulation models, see [Limitations](#) in *AI Engine Tools and Flows User Guide (UG1076)*.

Asynchronous Circular Buffer

Declare kernel function k3 with asynchronous circular input one dimension buffer that operates on data type `int32` with buffer size specified in graph and margin size `MARGIN_SIZE` that is named `in0` and output stream that operates on data type `int32` that is named `out0`.

```
void k3(input_async_circular_buffer<int32, adf::extents<adf::inherited_extent>, adf::margin<MARGIN_SIZE>>
&in0, output_stream<int32> *out0)
{
    in0.acquire();
    auto in0Iter = aie::begin_circular(in0);
    for (int ind = 0; ind < INPUT_SAMPLE_SIZE + MARGIN_SIZE; ++ind)
    {
        writeincr(out0, *in0Iter++);
    }

    in0.release();
}
```

Buffer Port Operations for Kernels

Moving the Current Read/Write Position Forward

In the following description, `input_buffer<TYPE>` stands for any of the allowed input buffer port data types. Likewise, `output_buffer<TYPE>` stands for any of the allowed output buffer port data types.

Table: Moving the Current Read/Write Position Forward

Purpose	Input Buffer Port Type	Output Buffer Port Type
To advance the current read/write position	<p>Option 1, Using an iterator:</p> <pre>void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie::begin(in); TYPE data = *pIn++; ... }</pre> <p>Option 2, Using the <code>data()</code> API:</p> <pre>void simple_1(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { const TYPE* pIn=in.data(); TYPE data = pIn[index++]; ... }</pre>	<p>Option 1, Using an iterator:</p> <pre>void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pOut= aie::begin(out); TYPE data; *pOut++ = data; ... }</pre> <p>Option 2, Using the <code>data()</code> API:</p> <pre>void simple_1(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { const TYPE* pOut =(TYPE*)out.data(); pOut[index++] = data; ... }</pre>
To advance the current read/write position by four times the underlying buffer port type.	<pre>#define VECTOR_SIZE 4 void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie::begin_vector<VECTOR_SIZE>(in); v4TYPE data = *pIn++; ... }</pre>	<pre>#define VECTOR_SIZE 4 void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pOut = aie::begin_vector<VECTOR_SIZE>(out); v4TYPE data; *pOut++ = data; ... }</pre>
To advance the current read/write position by eight times the underlying buffer port type.	<pre>#define VECTOR_SIZE 8 void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie::begin_vector<VECTOR_SIZE>(in); ... }</pre>	<pre>#define VECTOR_SIZE 8 void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pOut = aie::begin_vector<VECTOR_SIZE>(out); ... }</pre>

Purpose	Input Buffer Port Type	Output Buffer Port Type
	<pre>v8TYPE data = *pIn++; ...</pre>	<pre>v8TYPE data; *pOut++ = data; ...</pre>

Moving the Current Read/Write Position Backward

In the following description, `input_buffer<TYPE>` and `input_circular_buffer<TYPE>` stands for any of the allowed input buffer port data types. Likewise, `output_buffer<TYPE>` and `output_circular_buffer<TYPE>` stands for any of the allowed output buffer port data types.

Table: Moving the Current Read/Write Position Backward

Purpose	Input Buffer Port Type	Output Buffer Port Type
To decrease the current read/write position.	<pre>void simple(input_circular_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie::begin_random_circular(in); ... TYPE data = *pIn--; ... }</pre>	<pre>void simple(input_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pOut = aie::begin_random_circular(out); TYPE data; ... *pOut-- = data; ... }</pre>
To decrement the current read/write position by four times the underlying buffer port type.	<pre>#define VECTOR_SIZE 4 void simple(input_circular_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie:: begin_vector_random_circular <VECTOR_SIZE>(in); ... v4TYPE data = *pIn--; ... }</pre>	<pre>#define VECTOR_SIZE 4 void simple(input_buffer<TYPE> & in, output_circular_buffer<TYPE> & out) { auto pOut = aie:: begin_vector_random_circular <VECTOR_SIZE>(out); v4TYPE data; ... *pOut-- = data; ... }</pre>
To decrement the current read/write position by eight times the underlying buffer port type.	<pre>#define VECTOR_SIZE 8 void simple(input_circular_buffer<TYPE> & in, output_buffer<TYPE> & out) { auto pIn = aie:: begin_vector_random_circular <VECTOR_SIZE>(in); ... v8TYPE data = *pIn--; ... }</pre>	<pre>#define VECTOR_SIZE 8 void simple(input_buffer<TYPE> & in, output_circular_buffer<TYPE> & out) { auto pOut = aie:: begin_vector_random_circular <VECTOR_SIZE>(out); v8TYPE data; ... *pOut-- = data; ... }</pre>


Comparison between Buffer Ports and Windows

In AMD Vitis™ tools, version 2022.1 and previous versions, the memory interface between kernels was called *window*. Buffer ports are similar to window ports, but differ as outlined below. Buffer ports provide a way for a kernel to operate on a block of core memory.

Table: Window and Buffer Differences

Windows	Buffers
Sizes and margins are in <i>bytes</i> .	Sizes and margins are in <i>samples</i> .

Windows	Buffers
Sizes can be defined only in <i>graph</i> using the connect statement.	Sizes can be defined either in <i>graph</i> using dimensions statement, or in <i>kernel</i> signature as compile time constant.
Margins can be defined only in <i>graph</i> using the connect statement.	Margins can be defined only in <i>kernel</i> signature as compile time constant.
Locking mechanism is specified in <i>graph</i> using connect construct.	Locking mechanism is specified in <i>kernel</i> signature.
C++14 is enough.	Buffer ports use C++17 fold expressions.
Support of <i>circular</i> addressing only.	Default addressing is <i>linear</i> . It can be set to <i>circular</i> .
For data access, the iteration operation is tied directly with the action reading or writing API, such as <code>window_readincr()</code> , and <code>window_writeincr()</code> .	A reference pointer or an iterator are supported to perform read or write operations.
One-dimensional only	Multi-dimensional. If buffer dimension is omitted in the kernel signature, it will be supposed one-dimensional.

 **Note:** Limitations on buffer port size, window size and margin size depend on hardware constraints, hence, there is no difference between buffer port and window in terms of minimum size.

Graph Conversion


Connect statements with buffer port to buffer port do not require template parameters specifying the size. Use `dimensions()` API in graph to specify number of samples (not bytes) for all input/output buffer ports. Buffer port size (number of samples, not bytes) can also be specified using kernel function prototypes when `dimensions()` APIs in graph are not used.

Window-Based Connect Example

```
connect<window<EQ24_INPUT_SAMPLES*4, EQ24_INPUT_MARGIN*4> > (in.out[0], eq24.in[0]);
connect<window<HB27_2D_INPUT_SAMPLES*4, HB_2D_INPUT_MARGIN*4> > (eq24.out[0], hb27.in[0]) ;
connect<window<HB27_2D_INPUT_SAMPLES*4, HB_2D_INPUT_MARGIN*4> > (eq24.out[1], hb27.in[1]) ;
connect<window<CE_INPUT_SAMPLES*4, CE_INPUT_MARGIN*4> > (hb27.out[0], ce.in[0]) ;
```

Buffer Port-Based Connect and Dimensions Example

```
connect(in.out[0], eq24.in[0]) ;
connect(eq24.out[0], hb27.in[0]) ;
connect(eq24.out[1], hb27.in[1]) ;
connect(hb27.out[0], ce.in[0]);
adf::dimensions(eq24.in[0]) = { EQ24_INPUT_SAMPLES };
adf::dimensions(hb27.in[0]) = { HB27_2D_INPUT_SAMPLES };
adf::dimensions(hb27.in[1]) = { HB27_2D_INPUT_SAMPLES };
adf::dimensions(eq24.out[0]) = { EQ24_OUTPUT_SAMPLES };
adf::dimensions(eq24.out[1]) = { EQ24_OUTPUT_SAMPLES };
adf::dimensions(hb27.out[0]) = { HB27_2D_OUTPUT_SAMPLES };
adf::dimensions(ce.in[0]) = { CE_INPUT_SAMPLES };
```

 **Note:** Margin size can be specified only in kernel function prototypes. Kernel Conversion section contains kernel function prototype with margin examples. The example above illustrates only graph code updates.

Kernel Conversion

When migrating from window based kernels to buffer port based kernels, users need to determine the buffer port addressing type for each kernel. Window data types are implemented as circular buffers; however, buffer port types are implemented as linear buffers unless declared as circular buffer port types. Criteria selecting addressing mode are listed below.

Criteria to select circular address mode are as follows:

- Both kernels must be located in the same tile, and they must communicate with each other directly.
- A margin size greater than 0 is used.
- Kernel needs to iterate over data cyclically.

Criteria to select linear address mode are as follows:

- Kernel location is not guaranteed to be within the same tile.
- The kernel does not require any margin.
- Extra margin copy does impact design performance.

For an explanation of the differences between linear and circular buffer ports, see [Linear and Circular Addressing of Buffer Ports](#).

Examples of Kernel Conversion Function Prototypes

Examples of kernel conversions follows:

Example 1

Window-based function prototype:

```
void k1(input_window_cint16 * __restrict inputw_l, input_window_cint16 * __restrict inputw_r,
output_window_cint16 * __restrict output);
```

One dimension circular buffer port function prototype:

```
void k1_buffer_port(input_circular_buffer <cint16, adf::extents<adf::inherited_extent>,
adf::margin<MARGIN_SIZE> > & __restrict inputw_l, input_circular_buffer <cint16,
adf::extents<adf::inherited_extent>, adf::margin<MARGIN_SIZE> > & __restrict inputw_r, output_circular_buffer
<cint16, adf::extents<adf::inherited_extent>> & __restrict output);
```


Example 2

Window-based function prototype:

```
void k2(input_window_cint16 * __restrict input_cb0, input_window_cint16 * __restrict input_cb1,
input_window_cint16 * __restrict input_cb2, input_window_cint16 * __restrict input_cb3, input_window_cint16 *
__restrict input_cb4, output_window_cint16 * __restrict output_cb);
```

One dimension buffer port function prototype:

```
void k2_buffer_port(input_buffer<cint16> & __restrict input_cb0, input_buffer<cint16> & __restrict input_cb1,
input_buffer<cint16> & __restrict input_cb2, input_buffer<cint16> & __restrict input_cb3,
input_buffer<cint16> & __restrict input_cb4, output_buffer<cint16> & __restrict output_cb);
```

 **Note:** When the inherited size is specified as buffer port size in the function prototype, the actual buffer port size is specified using the `dimensions()` API in graph.

Kernel Code Conversion

Buffer ports do not support read, read advancing, read decrementing, write, and write advancing APIs.

[Access Data From a Buffer Port](#) illustrates multiple methods for accessing data with pointer, iterator, and vector referencing.

If the kernel algorithm uses a vector, vector referencing with vector iterator is recommended to use the vector processor for best and optimized performance.

If the kernel algorithm uses a scalar data type, iterators are recommended to simplify the code, making it portable and safer compared to pointer referencing.

Examples of Reading From Input

Window read operations need to convert to input buffer port iterator reads. Vector-based operations need to use a vector iterator. Iterator incrementing/decrementing corresponds to `window_readincr()` and `window_writeincr()` operations.

Window-based input example:

```
input_window_cint16 * restrict inputw_l;
```

```
...
window_incr_v8(inputw_r,3);
v8cint16 vdata;
window_readincr(inputw_l, vdata);
window_read(inputw_l, vdata);
window_decr_v8(inputw_l, 1);
```

Buffer port based input:

```
adf::input_circular_buffer<cint16, adf::extents<adf::inherited_extent>, adf::margin<MARGIN_SIZE>> &
__restrict inputw_l;
auto inputw_lItr = aie::begin_vector_random_circular<8>(inputw_l);

inputw_lItr += 3;

v8cint16 vdata;
vdata = *inputw_lItr++;
vdata = *inputw_lItr;
inputw_lItr--;
```

Window-based output example:

```
output_window_cint16 * __restrict outputw;
const unsigned output_samples = GET_NUM_SAMPLES(outputw);
acc0 = mac4_sym_ct(acc0, lbuff, 14, 0x6420, 2, rbuff, 0, 5, coe, 8, 0x0000, 1);
window_writeincr(outputw, srs(acc0, shift));
```


Buffer port based output:

```
output_circular_buffer<cint16> & __restrict outputw;
const unsigned output_samples = outputw.size();
auto outputwItr = aie::begin_vector_random_circular<8>(outputw);

acc0 = mac4_sym_ct(acc0, lbuff, 14, 0x6420, 2, rbuff, 0, 5, coe, 8, 0x0000, 1);
aie::vector<cint16, 4> v1 = srs(acc0, shift);
*outputwItr++ = v1;
```

Streaming Data API

Data flow graph kernels operate on data streams that are infinitely long sequences of typed values. These data streams can be broken into separate blocks and these blocks are processed by a kernel. Kernels consume input blocks of data and produce output blocks of data. Kernels can also access the data streams in a sample-by-sample fashion. The data access API in these two cases are described in this chapter.

 **Note:** The data movement APIs described in this chapter apply to both vector and scalar, signed and unsigned data. However, note that the AI Engine architecture supports unsigned integer *vector* arithmetic only for the 8-bit data types `aie::vector<uint8, 16>`, `aie::vector<uint8, 32>`, `aie::vector<uint8, 64>`, `aie::vector<uint8, 128>`. But for *scalar* arithmetic, all standard C unsigned integer data types `unsigned char(uint8)`, `unsigned short(uint16)`, `unsigned int(uint32)`, `unsigned long long(uint64)` are supported.

Data Access Mechanisms

Stream-Based Access

With a stream-based access model, the kernels receive an input stream or an output stream of typed data as an argument. Each access to these streams is synchronized, i.e., reads stall if the data is not available in the stream and writes stall if the stream is unable to accept new data.

An AI Engine supports two 32-bit input stream ports with `id=0` or `1` and two 32-bit output stream ports with `id=0` or `1`. This ID is supplied as an argument to the stream object constructors. The AI Engine compiler automatically allocates the input and output stream port IDs from left to right in the argument list of a kernel. Multiple kernels mapped to the same AI Engine are not allowed to share stream ports unless the streams are packet switched (see [Explicit Packet Switching](#)).

```
public:
    input_plio din;
```

```

output_plio dout;
adf::kernel k0,k1;
...
connect <stream> (din.out[0], k1.in[0]);
connect <stream> (k1.out[0], k2.in[0]);
connect <stream> (k2.out[0], dout.in[0]);

```

There is also a direct stream communication channel between the accumulator register of one AI Engine and the physically adjacent core, called a cascade. The cascade stream is connected within the AI Engine array in a snake-like linear fashion from AI Engine processor to processor.

```
connect <cascade> (k1.out[1], k2.in[1]);
```

The stream data structures are automatically inferred by the AI Engine compiler from data flow graph connections, and are automatically declared in the wrapper code implementing the graph control. The kernel functions merely operate on pointers to stream data structures that are passed to them as arguments. There is no need to declare these stream data structures in data flow graph or kernel program.

Stream Connection for Multi-Rate Processing


Multi-rate analysis is not an easy task when it comes to streams and packet-streams connections if there is no user specification. Use constraints to specify how many samples the kernel will read from the input stream or pktstream input, and how many samples the kernel will write to the stream or pktstream output, as shown below:

```

// constraint to specify samples per iteration for stream/pktstream ports to support multirate connections
constraint<uint32_t> samples_per_iteration(adf::port<adf::input>& p);
constraint<uint32_t> samples_per_iteration(adf::port<adf::output>& p);

```

The `constraint` keyword needs the sample datatype as a template value and the function `samples_per_iteration` is applied to the input or the output of the kernel. The related stream can be connected to another stream of a buffer.

 **Note:** The multi-rate analysis pass calculates the rate for only those stream/pktstream ports for which `adf::samples_per_iteration (>0)` is specified.

Stream Operations for Kernels

Stream Data Types

Each of the data types in the table can be read or written from the AI Engine as either scalars or in vector groups. However, there are certain restrictions on valid groupings based on the bus data width supported on the AI Engine to programmable logic interface ports or through the stream-switch network. The valid combinations for AI Engine kernels are vector bundles totaling up to 32-bits or 128-bits. The accumulator data types are only used to specify cascade-stream connections between adjacent AI Engines. Its valid groupings are based on the 384-bit wide cascade channel between two processors.

Input and output Cascade data types can be used in the context of AI Engine APIs or ADF APIs. AMD recommends the use of AI Engine APIs because it supports a larger range of lanes. To use AI Engine APIs, include `#include <aie_adf.hpp>` in the kernel source code.

ADF APIs support a limited number of lanes. To use ADF APIs, include `#include <adf.h>` in the kernel source code. ADF APIs are used for advanced kernel programming using intrinsic calls.

Table: Stream Data Types

Input Stream Types	Output Stream Types
<code>input_stream<int8></code>	<code>output_stream<int8></code>
<code>input_stream<int16></code>	<code>output_stream<int16></code>
<code>input_stream<int32></code>	<code>output_stream<int32></code>
<code>input_stream<int64></code>	<code>output_stream<int64></code>
<code>input_stream<uint8></code>	<code>output_stream<uint8></code>
<code>input_stream<uint16></code>	<code>output_stream<uint16></code>
<code>input_stream<uint32></code>	<code>output_stream<uint32></code>
<code>input_stream<uint64></code>	<code>output_stream<uint64></code>

Input Stream Types	Output Stream Types
input_stream<cint16>	output_stream<cint16>
input_stream<cint32>	output_stream<cint32>
input_stream<float>	output_stream<float>
input_stream<cfloat>	output_stream<cfloat>

Table: Cascade Accumulator Data Types

Input Cascade Types	Output Cascade Types	Lanes in ADF API (adf.h)	Lanes in AIE API (aie_adf.hpp)
input_cascade<acc48>	output_cascade<acc48>	8	8/16/32/64/128
input_cascade<cacc48>	output_cascade<cacc48>	4	4/8/16/32/64
input_cascade<acc80>	output_cascade<acc80>	4	4/8/16/32/64
input_cascade<cacc80>	output_cascade<cacc80>	2	2/4/8/16/32
input_cascade<accfloat>	output_cascade<accfloat>	8	4/8/16/32
input_cascade<caccfloat>	output_cascade<caccfloat>	4	2/4/8/16
input_cascade<int8>	output_cascade<int8>	32	16/32/64/128
input_cascade<uint8>	output_cascade<uint8>	32	16/32/64/128
input_cascade<int16>	output_cascade<int16>	16	8/16/32/64
input_cascade<int32>	output_cascade<int32>	8	4/8/16/32
input_cascade<cint16>	output_cascade<cint16>	8	4/8/16/32
input_cascade<cint32>	output_cascade<cint32>	4	2/4/8/16
input_cascade<cfloat>	output_cascade<cfloat>	4	2/4/8/16
input_cascade<float>	output_cascade<float>	8	4/8/16/32

Reading and Advancing an Input Stream

AI Engine Operations

The following operations read data from the given input stream and advance the stream on the AI Engine. Because there are two input stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values from the stream can be read one at a time or as a vector. In the latter case, unless all values are present, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. The cascade connection reads all accumulator values in parallel.

```
//Scalar operations
//#include<aie_adf.hpp> or #include<adf.h>
int32 readincr(input_stream<int32> *w);
uint32 readincr(input_stream<uint32> *w);
cint16 readincr(input_stream<cint16> *w);
float readincr(input_stream<float> *w);
cfloat readincr(input_stream<cfloat> *w);

//AIE API Operations to read vector data which supports more vector lanes
//#include<aie_adf.hpp>
aie::vector<int8,16> readincr_v<16>(input_stream<int8> *w);
aie::vector<uint8,16> readincr_v<16>(input_stream<uint8> *w);
aie::vector<int16,8> readincr_v<8>(input_stream<int16> *w);
aie::vector<cint16,4> readincr_v<4>(input_stream<cint16> *w);
aie::vector<int32,4> readincr_v<4>(input_stream<int32> *w);
aie::vector<cint32,2> readincr_v<2>(input_stream<cint32> *w);
```

```

aie::vector<float,4> readincr_v<4>(input_stream<float> *w);

template<typename T,int N>
aie::accum<T,N> readincr_v<N>(input_cascade<T> *w);
template<typename T,int N>
aie::vector<T,N> readincr_v<N>(input_cascade<T> *w);

//ADF API Operations to read vector data which supports limited vector lanes
#include<adf.h>
v8acc48 readincr_v8(input_cascade<acc48>* str);
v4acc80 readincr_v4(input_cascade<acc80>* str);
v8accfloat readincr_v8(input_cascade<accfloat>* str);
v4cacc48 readincr_v4(input_cascade<cacc48>* str);
v2cacc80 readincr_v2(input_cascade<cacc80>* str);
v4caccfloat readincr_v4(input_cascade<caccfloat>* str);
v32int8 readincr_v32(input_cascade<int8>* str);
v32uint8 readincr_v32(input_cascade<uint8>* str);
v16int16 readincr_v16(input_cascade<int16>* str);
v8cint16 readincr_v8(input_cascade<cint16>* str);
v8int32 readincr_v8(input_cascade<int32>* str);
v4cint32 readincr_v4(input_cascade<cint32>* str);
v8float readincr_v8(input_cascade<float>* str);
v4cfloat readincr_v4(input_cascade<cfloat>* str);

```

For the supported data types and lanes in AI Engine API `readincr_v<N>`, see [Stream Data Types](#).

Note: To indicate the end of the stream, the `readincr` API can be used with a `TLAST` argument as shown below.

```
int32 readincr(input_stream<int32> *w, bool &tlast);
```

Writing and Advancing an Output Stream

AI Engine Operations

The following operations write data to the given output stream and advance the stream on the AI Engine. Because there are two output stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values can be written to the output stream one at a time or as a vector. In the latter case, until all values are written, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. Cascade connection writes all values in parallel.

```

//Scalar operations
#include<aie_adf.hpp> or #include<adf.h>
void writeincr(output_stream<int32> *w, int32 v);
void writeincr(output_stream<int64> *w, int64 v);
void writeincr(output_stream<uint32> *w, uint32 v);
void writeincr(output_stream<cint16> *w, cint16 v);
void writeincr(output_stream<cint32> *w, cint32 v);
void writeincr(output_stream<float> *w, float v);
void writeincr(output_stream<cfloat> *w, cfloat v);

//AIE API Operations to read vector data which supports more vector lanes
#include<aie_adf.hpp>
void writeincr(output_stream<int8> *w, aie::vector<int8,16> &v);
void writeincr(output_stream<uint8> *w, aie::vector<uint8,16> &v);
void writeincr(output_stream<int16> *w, aie::vector<int16,8> &v);
void writeincr(output_stream<cint16> *w, aie::vector<cint16,4> &v);
void writeincr(output_stream<int32> *w, aie::vector<int32,4> &v);
void writeincr(output_stream<cint32> *w, aie::vector<cint32,2> &v);
void writeincr(output_stream<float> *w, aie::vector<float,4> &v);


template<typename T,int N>
void writeincr(output_cascade<T> *w, aie::accum<T,N> &v);
template<typename T,int N>
void writeincr(output_cascade<T> *w, aie::vector<T,N> &v);

```



```
//ADF API Operations to read vector data which supports limited vector lanes
//#include<adf.h>
void writeincr(output_cascade<acc48>* str,v8acc48 value);
void writeincr(output_cascade<acc80>* str,v4acc80 value);
void writeincr(output_cascade<cacc48>* str,v4cacc48 value);
void writeincr(output_cascade<cacc80>* str,v2cacc80 value);
void writeincr(output_cascade<accfloat>* str,v8accfloat value);
void writeincr(output_cascade<caccfloat>* str,v4caccfloat value);
void writeincr(output_cascade<int8>* str,v32int8 value);
void writeincr(output_cascade<uint8>* str,v32uint8 value);
void writeincr(output_cascade<int16>* str,v16int16 value);
void writeincr(output_cascade<cint16>* str,v8cint16 value);
void writeincr(output_cascade<int32>* str,v8int32 value);
void writeincr(output_cascade<cint32>* str,v4cint32 value);
void writeincr(output_cascade<float>* str,v8float value);
void writeincr(output_cascade<cfloat>* str,v4cfloat value);
```

For the supported data types and lanes in AI Engine API `writeincr`, see [Stream Data Types](#).

 **Note:** To indicate the end of stream, the `writeincr` API is used with a `TLAST` argument as shown below.

```
void writeincr(output_stream<int32> *w, int32 value, bool tlast);
```

Packet Stream Operations

Table: Supported Packet Stream Data Types

Input Stream Types	Output Stream Types
input_pktstream	output_pktstream

Two additional stream data types are provided to characterize streaming data that consists of packetized interleaving of several different streams. These data types are useful when the number of independent data streams in your program exceeds the number of hardware stream channels or ports available. This mechanism is described in more detail in [Explicit Packet Switching](#).

Packet Stream Reading and Writing

A data packet consists of a one word (32-bit) packet header, followed by some number of data words where the last data word has the `TLAST` field signaling the end-of-packet. The following operations are used to read and advance input packet streams and write and advance output packet streams.

```
int32 readincr(input_pktstream *w);
int32 readincr(input_pktstream *w, bool &tlast);

void writeincr(output_pktstream *w, int32 value);
void writeincr(output_pktstream *w, int32 value, bool tlast);
```

The API with `TLAST` argument help to read or write the end-of-packet condition if the packet size is not fixed.

If the packet header word has sideband `TLAST` equals to true, this signals that the packet is complete and there is no more data with the packet, that is, the packet has only the packet header.

Packet Processing

The first 32-bit word of a packet must always be a packet header, which encodes several bit fields as shown in the following table.

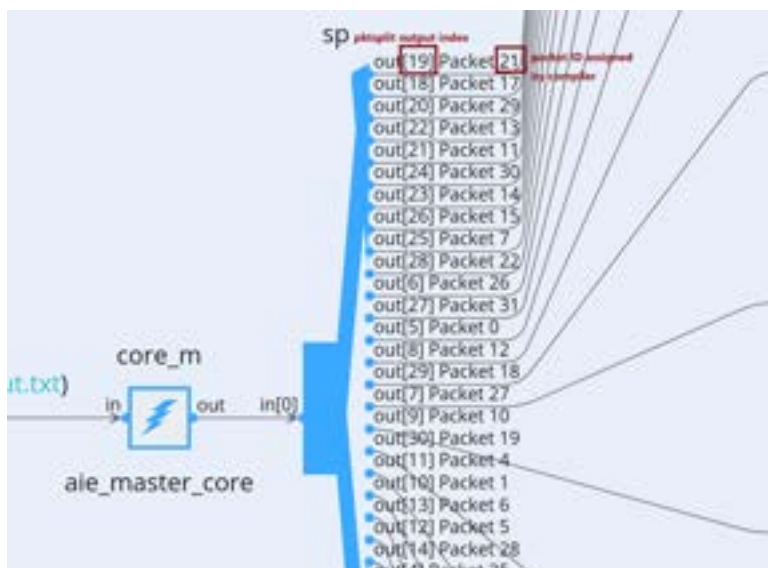
Table: Packet Bit Fields

Bits	Field
4-0	Packet ID
11-5	7'b0000000
14-12	Packet Type

Bits	Field
15	1'b0
20-16	Source Row
27-21	Source Column
30-28	3'b000
31	Odd parity of bits[30:0]

The packet ID is assigned by the compiler based on routing requirements. In the graph, the `pktsplit` output index or `pktmerge` input index might be different with the assigned packet ID. For example, the following figure shows a compilation result of a `pktsplit`:

Figure: Graph View of `pktsplit` with Packet ID



From the above graph view, the index of the `pktsplit` output is shown first, followed by the corresponding packet ID. So, inside the AI Engine kernel, the packet ID should be queried by the `getPacketid` API to make sure that the code is valid across different compilations.

The packet type can be any 3-bit pattern that you want to insert to identify the type of packet. The source row and column denote the AI Engine tile coordinates from where the packet originated. By convention, source row and column for packets originating in the programmable logic (PL) is -1,-1.

It is your responsibility to construct and send an appropriate packet header at the beginning of every packet. On the receive side, the packet header needs to be received and decoded before reading the data.

The following operations help to assemble or disassemble the packet header in the AI Engine kernel.

```
void writeHeader(output_pktstream *str, unsigned int pktType, unsigned int ID);
void writeHeader(output_pktstream *str, unsigned int pktType, unsigned int ID, bool tlast);

uint32 getPacketid(input_pktstream *w, int index);
uint32 getPacketid(output_pktstream *w, int index);
```

The `writeHeader` API allows a packet header to be assembled with a given packet ID and packet type, while the packet ID should be queried inside the kernel with the `getPacketid` API. The source row and column are inserted automatically using the coordinates of the AI Engine tile where this API is executed.

The `getPacketid` API allows the compiler assigned packet ID to be queried on the input or output packet stream data structure. The index argument refers to the split or merge branch edge in the graph specification.

The example below uses the `getPacketid` API to query the packet ID on an output packet stream. This ID can be written to the output packet stream using the `writeHeader` API.

```
void aie_core1(...,output_pktstream *out){
    //Get ID from output pktstream, index=0
    uint32 ID=getPacketid(out,0);
    //Generate header for output
    writeHeader(out,pktType,ID);
```

.....

See [Explicit Packet Switching](#) for more examples on different graphs and kernel code.

!! Important: The `writeHeader()` and `getPacketid()` APIs are not supported in PL kernels.

Runtime Graph Control API

This chapter describes the control APIs that can be used to initialize, run, update, and control the graph execution from an external controller. This chapter also describes how runtime parameters (RTP) can be specified in the input graph specification that affect the data processing within the kernels and change the control flow of the overall graph synchronously or asynchronously.

Graph Execution Control

In AMD Versal™ Adaptive SoCs with AI Engines, the processing system (PS) can be used to dynamically load, monitor, and control the graphs that are executing on the AI Engine array. Even if the AI Engine graph is loaded once as a single bitstream image, the PS program can be used to monitor the state of the execution and modify the runtime parameters of the graph.

The graph base class provides a number of API methods to control the initialization and execution of the graph that can be used in the `main` program. The user `main` application used in simulating the graph does not support `argc`, `argv` parameters.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

Basic Iterative Graph Execution

The following graph control API shows how to use graph APIs to initialize, run, wait, and terminate graphs for a specific number of iterations. A graph object `mygraph` is declared using a pre-defined graph class called `simpleGraph`. Then, in the `main` application, this graph object is initialized and run. The `init()` method loads the graph to the AI Engine array at prespecified AI Engine tiles. This includes loading the ELF binaries for each AI Engine, configuring the stream switches for routing, and configuring the DMAs for I/O. It leaves the processors in a disabled state. The `run()` method starts the graph execution by enabling the processors. The `run` API is where a specific number of iterations of the graph can be run by supplying a positive integer argument at run time. This form is useful for debugging your graph execution.

```
#include "project.h"
simpleGraph mygraph;

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

The API `wait()` is used to wait for the first run to finish before starting the second run. `wait` has the same blocking effect as `end` except that it allows re-running the graph again without having to re-initialize it. Calling `run` back-to-back without an intervening `wait` to finish that run can have an unpredictable effect because the `run` API modifies the loop bounds of the active processors of the graph.

Graph Iteration

A graph can have multiple kernels, input and output ports. The graph connectivity, which is equivalent to the nets in a data flow graph is either between the kernels, between kernel and input ports, or between kernel and output ports, and can be configured as a connection. A graph runs for an iteration when it consumes data samples equal to the buffer or stream of data expected by the kernels in the graph, and produces data samples equal to the buffer or stream of data expected at the output of all the kernels in the graph.

Finite Execution of Graph

For finite graph execution, the graph state is maintained across the `graph.run(n)`. The AI Engine is not reinitialized and memory contents are not cleared after `graph.run(n)`. In the following code example, after the first run of three invocations, the AI Engine `main()` wrapper code is left in a state where the kernel will start with the pong buffer in the next run (of ten iterations). The ping-pong buffer selector state is left as-is. `graph.end()` does not clean up the graph state (specifically, does not re-initialize global variables), nor clean up stream switch configurations. It merely exits the core-main. To re-run the graph, you must reload the PDI/XCLBIN.

!! Important: A `graph.wait()` must be followed by either a `graph.run()` or `graph.resume()` prior to a `graph.end()`. Failing to do so means that a graph could wait forever, and `graph.end()` never executes.

```
#include "project.h"
simpleGraph mygraph;

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

Related Information

Graph Objects

Infinite Graph Execution

The following graph control API shows how to run the graph infinitely.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init(); // load the graph
    mygraph.run(); // start the graph and return to main
    .....
    return 0;
}
```

A graph object `mygraph` is declared using a pre-defined graph class called `simpleGraph`. Then, in the `main` application, this graph object is initialized and run. The `init()` method loads the graph to the AI Engine array at prespecified AI Engine tiles. This includes loading the ELF binaries for each AI Engine, configuring the stream switches for routing, and configuring the DMAs for I/O. It leaves the processors in a disabled state. The `run()` method starts the graph execution by enabling the processors. This graph runs forever because the number of iterations to be run is not provided to the `run()` method.


`graph::run()` without an argument runs the AI Engine kernels for a previously specified number of iterations (which is infinity by default if the graph is run without any arguments). If the graph is run with a finite number of iterations, for example, `mygraph.run(3); mygraph.run()` the second run call also runs for three iterations. Use `graph::run(-1)` to run the graph infinitely regardless of the previous run iteration setting.

Parallel Graph Execution

In the previous API methods, only the `wait()` and `end()` methods are blocking operations that can block the `main` application indefinitely. Therefore, if you declare multiple graphs at the top level, you need to interleave the APIs suitably to execute the graphs in parallel, as shown.

```
#include "project.h"
simpleGraph g1, g2, g3;

int main(void) {
    g1.init(); g2.init(); g3.init();
    g1.run(<num-iter>); g2.run(<num-iter>); g3.run(<num-iter>);
    g1.end(); g2.end(); g3.end();
    return 0;
}
```

 **Note:** Each graph should be started (`run`) only after it has been initialized (`init`). Also, to get parallel execution, all the graphs must be started (`run`) before any graph is waited upon for termination (`end`).


Timed Execution

In multi-rate graphs, all kernels need not execute for the same number of iterations. In such situations, a timed execution model is more suitable for testing. There are variants of the `wait` and `end` APIs with a positive integer that specifies a cycle timeout. This is the number of AI

Engine cycles that the API call blocks before disabling the processors and returning. The blocking condition does not depend on any graph termination event. The graph can be in an arbitrary state at the expiration of the timeout.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run();
    mygraph.wait(10000); // wait for 10000 AI Engine cycles
    mygraph.resume();    // continue executing
    mygraph.end(15000);  // wait for another 15000 cycles and terminate
}
```

 **Note:** The API `resume()` is used to resume execution from the point it was stopped after the first timeout. `resume` only resets the timer and enables the AI Engines. Calling `resume` after the AI Engine execution has already terminated has no effect.

Runtime Parameter Specification

The data flow graphs shown until now are defined completely statically. However, in real situations you might need to modify the behavior of the graph based on some dynamic condition or event. The required modification could be in the data being processed, for example a modified mode of operation or a new coefficient table, or it could be in the control flow of the graph such as conditional execution or dynamically reconfiguring a graph with another graph. Runtime parameters (RTP) are useful in such situations. Either the kernels or the graphs can be defined to execute with parameters. Additional graph API are also provided to update or read these parameter values while the graph is running.

Two types of runtime parameters are supported. The first is the asynchronous or sticky parameters which can be changed at any time by a controlling processor such as the Processing System (PS). After complete update, they are read each time a kernel is invoked. These types of parameters can be used as filter coefficients that change infrequently.

Synchronous or triggering parameters are the other type of supported runtime parameters. A kernel that requires a triggering parameter does not execute until these parameters are written by a controlling processor. Upon a write, the kernel executes once, reading the new updated value. After completion, the kernel is blocked from executing until the parameter is updated again. This allows a different type of execution model from the normal streaming model, which can be useful for certain updating operations where blocking synchronization is important. Runtime parameters can either be scalar values or array values. A `graph.update()` API is used for RTP update.


Specifying Runtime Data Parameters

Parameter Inference

If an integer scalar value appears in the formal arguments of a kernel function, then that parameter becomes a runtime parameter. In the following example, the arguments `select` and `result_out` are runtime parameters.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H
    void simple_param(input_buffer<int32> &in, output_buffer<int32> &out, int32 select, int32 &result_out);
#endif
```

Runtime parameters are processed as ports alongside those created by streams and buffers. Both scalar and array of scalar data types can be passed as runtime parameters. The valid scalar data types supported are `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`.

 **Note:** Structs and pointers cannot be passed as runtime parameters. Every AI Engine has a different memory space and pointers have an inconsistent meaning when passed between AI Engines. Furthermore, the PS and AI Engines have different pointer representations. Because structs can contain pointers as members, the passing of structs as runtime parameters is not supported.

In the following example, an array of 32 integers is passed as a parameter into the `filter_with_array_param` function.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H

    void filter_with_array_param(input_buffer<cint16> & in, output_buffer<cint16> * out, const int32
(&coefficients)[32]);

#endif
```

Implicit ports are inferred for each parameter in the function argument, including the array parameters. The following table describes the type of port inferred for each function argument.

Table: Port Type per Parameter

Formal Parameter	Port Class
T	Input
const T	Input
T &	Inout
const T &	Input
const T (&)[...]	Input
T(&)[...]	Inout

From the table, you can see that when the AI Engine cannot make externally visible changes to the function parameter, an input port is inferred. When the formal parameter is passed by value, a copy is made, so changes to that copy are not externally visible. When a parameter is passed with a `const` qualifier, the parameter cannot be written, so these are also treated as input ports.

When the AI Engine kernel is passed a parameter reference and it is able to modify it, an inout port is inferred and can be used to allow reading back of results from the control processor.

Note: The inout port is a port that a kernel itself can read or write. But, from the graph, the inout port can only behave as an output port. Therefore, the inout port can only be read by `graph::read()`. The inout port *cannot* be updated by `graph::update()`.

Note: If a kernel wants to accept an input runtime parameter, modify its value, and pass back this modified value via an output runtime parameter, then the variable has to appear twice in the arg list, once as an input and once as an inout, for example, `kernel_function(int32 foo_in, int32 &foo_out)`.

Parameter Hookup

Both input and inout runtime parameter ports can be connected to corresponding hierarchical ports in their enclosing graph. This is the mechanism that parameters are exposed for runtime modification. In the following graph, an instance is created of the previously defined `simple_param` kernel. This kernel has two input ports, one output port and one inout port. The first argument to appear in the argument list, `in[0]`, is an input buffer. The second argument is an output buffer. The third argument is a runtime parameter (it is not a buffer or stream type) and is inferred as an input parameter, `in[1]`, because it is passed by value. The fourth argument is a runtime parameter and is inferred as an inout parameter, `inout[0]`, because it is passed by reference.

Note: The different port types `in`, `out`, and `inout` for a kernel belong to their own categories, and all start from index 0 in the graph.

In the following graph definition, a `simple_param` kernel is instantiated and buffers are connected to `in[0]` and `out[0]` (the input and output buffers of the kernel). The input runtime parameter is connected to the graph input port, `select_value`, and the inout runtime parameter is connected to the graph inout port, `result_out`.

```
class parameterGraph : public graph {
private:
    kernel first;

public:
    input_port select_value;
    input_plio in;
    output_plio out;
    inout_port result_out;
    parameterGraph() {
        first = kernel::create(simple_param);
        .....
        connect(in.out[0], first.in[0]);
        connect(first.out[0], out.in[0]);
        connect<parameter>(select_value, first.in[1]); //default sync rtp input
        connect<parameter>(first.inout[0], result_out); //default async rtp output
    }
};
```

An array parameter can be hooked up in the same way. The compiler automatically allocates space for the array data so that it is accessible from the processor where this kernel gets mapped.

```

class arrayParameterGraph : public graph {
private:
    kernel first;

public:
    input_port coeffs;
    input_plio in;
    output_plio out;
    arrayParameterGraph() {
        first = kernel::create(filter_with_array_param);
        .....
        connect(in.out[0], first.in[0]);
        connect(first.out[0], out.in[0]);
        connect<parameter>(coeffs, first.in[1]);
    }
};

```

Input Parameter Synchronization

The default behavior for input runtime parameters ports is triggering behavior. This means that the parameter plays a part in the rules that determine when a kernel could execute. In this graph example, the kernel only executes when three conditions are met:

- A valid buffer of 32 bytes of input data is available
- An empty buffer of 32 bytes is available for the output data
- A write to the input parameter takes place

In triggering mode, a single write to the input parameter allows the kernel to execute once, setting the input parameter value on every individual kernel call.

There is an alternative mode to allow input kernels parameters to be set asynchronously. To specify that parameters update asynchronously, use the `async` modifier when connecting a port.

```
connect<parameter>(param_port, async(first.in[1]));
```

When a kernel port is designated as asynchronous, it no longer plays a role in the firing rules for the kernel. When the parameter is written once, the value is observed in subsequent firings. At any time, the PS can write a new value for the runtime parameter. That value is observed on the next and any subsequent kernel firing.

Inout Parameter Synchronization

The default behavior for inout runtime parameters ports is asynchronous behavior. This means that the parameter can be read back by the controlling processor or another kernel, but the producer kernel execution is not affected. For synchronous behavior from the `inout` parameter where the kernel blocks until the parameter value is read out on each invocation of the kernel, you can use a `sync` modifier when connecting the `inout` port to the enclosing graph as follows.

```
connect<parameter>(sync(first.inout[1]), param_port);
```

Runtime Parameter Update/Read Mechanisms

This section describes the mechanisms to update or read back the runtime parameters. For these types of applications, it is usually better not to specify an iteration limit at compile time to allow the cores to run freely and monitor the effect of the parameter change.

Parameter Update/Read Using Graph APIs

In default compilation mode, the `main` application is compiled as a separate control thread which needs to be executed on the PS in parallel with the graph executing on the AI Engine array. The `main` application can use update and read APIs to access runtime parameters declared within the graphs at any level. This section describes these APIs using examples.

Synchronous Update/Read

The following code shows the `main` application of the `simple_param` graph described in [Specifying Runtime Data Parameters](#).

```
#include "param.h"
parameterGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run(2);

    mygraph.update(mygraph.select_value, 23);
    mygraph.update(mygraph.select_value, 45);

    mygraph.end();
    return 0;
}
```

In this example, the graph `mygraph` is initialized first and then run for two iterations. It has a triggered input parameter port `select_value` that must be updated with a new value for each invocation of the receiving kernel. The first argument of the `update` API identifies the port to be updated and the second argument provides the value. Several other forms of update APIs are supported based on the direction of the port, its data type, and whether it is a scalar or array parameter.

If the program is compiled with a fixed number of test iterations, then for triggered parameters the number of update API calls in the `main` program must match the number of test iterations, otherwise the simulation could be waiting for additional updates. For asynchronous parameters, the updates are done asynchronously with the graph execution and the kernel uses the old value if the update was not made. If additionally, the previous graph was compiled with a synchronous inout parameter, then the update and read calls must be interleaved as shown in the following example.

```
#include "param.h"
parameterGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();
    mygraph.run(2);

    mygraph.update(mygraph.select_value, 23);
    mygraph.read(mygraph.result_out, result0);
    mygraph.update(mygraph.select_value, 45);
    mygraph.read(mygraph.result_out, result1);

    mygraph.end();
    return 0;
}
```

In this example, it is assumed that the graph produces a scalar result every iteration through the inout port `result_out`. The `read` API is used to read out the value of this port synchronously after each iteration. The first argument of the `read` API is the graph inout port to be read back and the second argument is the location where the value will be stored (passed by reference).

The synchronous protocol ensures that the read operation will wait for the value to be produced by the graph before sampling it and the graph will wait for the value to be read before proceeding to the next iteration. This is why it is important to interleave the update and read operations.

Asynchronous Update/Read

When an input parameter is specified with asynchronous protocol, the kernel execution waits for the first update to happen for parameter initialization. However, an arbitrary number of kernel invocations can take place before the next update. This is usually the intent of the asynchronous update during application deployment. However, for debugging, `wait` API can be used to finish a predetermined set of iterations before the next update as shown in the following example.

```
#include "param.h"
asyncGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();

    mygraph.update(mygraph.select_value, 23);
```



```

mygraph.run(5);
mygraph.wait();
mygraph.update(mygraph.select_value, 45);
mygraph.run(15);
mygraph.end();
return 0;
}

```

In the previous example, after the initial update, five iterations are run to completion followed by another update, then followed by another set of 15 iterations. If the graph has asynchronous inout ports, that data can also be read back immediately after the wait (or end).

Another template for asynchronous updates is to use timeouts in wait API as shown in the following example.

```

#include "param.h"
asyncGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();
    mygraph.run();
    mygraph.update(mygraph.select_value, 23);
    mygraph.wait(10000);
    mygraph.update(mygraph.select_value, 45);
    mygraph.resume();
    mygraph.end(15000);
    return 0;
}

```

In this example, the graph is set up to run forever. However, after the run API is called, it still blocks for the first update to happen for parameter initialization. Then, it runs for 10,000 cycles (approximately) before allowing the control thread to make another update. The new update takes effect at the next kernel invocation boundary. Then the graph is allowed to run for another 15,000 cycles before terminating.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

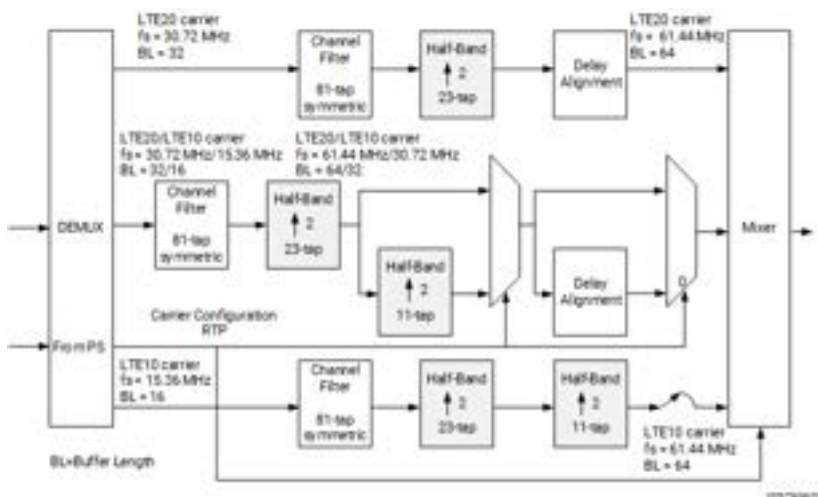
Runtime Graph Reconfiguration Using Control Parameters

The runtime parameters are also used to switch the flow of data within the graph and provide alternative routes for processing dynamically. The most basic version of this type of processing is a kernel bypass that allows the data to be processed or pass-through based on a runtime parameter (see [Kernel Bypass](#)). This can be useful, for example, in multi-modal applications where switching from one mode to another requires bypassing a kernel.

Bypass Control Using Runtime Parameters

The following figure shows an application supporting two channels of signal data, where one is split into two channels of lower bandwidth while the other must continue to run undisturbed. This type of dynamic reconfiguration is common in wireless applications.

Figure: Dynamic Reconfiguration of 2 LTE20 Channels into 1 LTE20 and 2 LTE10 Channels



In the figure, the first channel processes LTE20 data unchanged, while the middle channel is dynamically split into two LTE10 channels. The

control parameters marked as carrier configuration RTP are used to split the data processing on a block boundary. When the middle channel is operating as an LTE20 channel, the 11-tap half-band kernel is bypassed. However, when the bandwidth of the middle channel is split between itself and the third channel forming two LTE10 channels, both of them need a 3-stage filter chain before the data can be mixed together. This is achieved by switching the 11-tap half-band filter back into the flow and reconfiguring the mixer to handle three streams of data instead of two.

★ **Tip:** The delay alignment kernels are needed to balance the sample delay when mixing LTE20 and LTE10 signals and must also be part of the control flow due to dynamic switching between the two.

The top-level input graph specification for the above application is shown in the following code.

```
class lte_reconfig : public graph {
private:
    kernel demux;
    kernel cf[3];
    kernel interp0[3];
    kernel interp1[2];
    bypass bphb11;
    kernel delay ;
    kernel delay_byp ;
    bypass bpdelay ;
    kernel mixer ;
public:
    input_port in;
    input_port fromPS;
    output_port out ;

    lte_reconfig() {
        // demux also handles the control
        demux = kernel::create(demultiplexor);
        connect<>(in, demux.in[0]);
        connect< parameter >(fromPS, demux.in[1]);

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

        // instantiate all channel kernels
        for (int i=0;i<3;i++) {
            cf[i] = kernel::create(fir_89t_sym);
            source(cf[i]) = "kernels/fir_89t_sym.cc";
            runtime<ratio>(cf[i]) = 0.12;
        }
        for (int i=0;i<3;i++) {
            interp0[i] = kernel::create(fir_23t_sym_hb_2i);
            source(interp0[i]) = "kernels/hb23_2i.cc";
            runtime<ratio>(interp0[i]) = 0.1;
        }
        for (int i=0;i<2;i++) {
            interp1[i] = kernel::create(fir_11t_sym_hb_2i);
            source(interp1[i]) = "kernels/hb11_2i.cc";
            runtime<ratio>(interp1[i]) = 0.1;
        }
        bphb11 = bypass::create(interp1[0]);
        mixer = kernel::create(mixer_dynamic);
        source(mixer) = "kernels/mixer_dynamic.cc";
        runtime<ratio>(mixer) = 0.4;
        delay = kernel::create(sample_delay);
        source(delay) = "kernels/delay.cc";
        runtime<ratio>(delay) = 0.1;
        delay_byp = kernel::create(sample_delay);
        source(delay_byp) = "kernels/delay.cc";
        runtime<ratio>(delay_byp) = 0.1;
        bpdelay = bypass::create(delay_byp) ;

        // Graph connections
        for (int i=0; i<3; i++) {
            connect<>(demux.out[i], cf[i].in[0]);
```

```
    connect< parameter >(demux.inout[i], cf[i].in[1]);
}
connect< parameter >(demux.inout[3], bphb11.bp);
connect< parameter >(demux.inout[5], negate(bpdelay.bp)) ;
for (int i=0;i<3;i++) {
    connect<>(cf[i].out[0], interp0[i].in[0]);
    connect< parameter >(cf[i].inout[0], interp0[i].in[1]);
}
// chan0 is LTE20 and is output right away
connect<>(interp0[0].out[0], delay.in[0]);
connect<>(delay.out[0], mixer.in[0]);
// chan1 is LTE20/10 and uses bypass
connect<>(interp0[1].out[0], bphb11.in[0]);
connect< parameter >(interp0[1].inout[0], bphb11.in[1]);
connect<>(bphb11.out[0], bpdelay.in[0]);
connect<>(bpdelay.out[0], mixer.in[1]);
// chan2 is LTE10 always
connect<>(interp0[2].out[0], interp1[1].in[0]);
connect< parameter >(interp0[2].inout[0], interp1[1].in[1]);
connect<>(interp1[1].out[0], mixer.in[2]);
//Mixer
connect< parameter >(demux.inout[4], mixer.in[3]);
connect<>(mixer.out[0], out);
};
};
```

The bypass specification is coded as a special encapsulator over the kernel to be bypassed. The port signature of the bypass matches the port signature of the kernel that it encapsulates. It also receives a runtime parameter to control the bypass: 0 for no bypass and 1 for bypass. The control can also be inverted by using the negate function as shown.

The bypass parameter port of this graph is an ordinary scalar runtime parameter and can be driven by another kernel or by the Arm® processor using the interactive or scripted mechanisms described in [Runtime Parameter Update/Read Mechanisms](#). This can also be connected hierarchically by embedding it into an enclosing graph.

Runtime Parameter Support Summary

This section summarizes the AI Engine runtime parameter (RTP) support status.

Table: AI Engine RTP Default and Support Status

AI Engine RTP (from/to PS)	Input		Output	
	Synchronous	Asynchronous	Synchronous	Asynchronous
Scalar	Default	Supported	Supported	Default
Array	Default	Supported	Supported	Default

Code snippets for RTP connections from or to the PS:

```
//Synchronous RTP, default for input
connect<parameter>(fromPS, first.in[0]);
//Synchronous RTP
connect<parameter>(fromPS, sync(first.in[0]));
//Asynchronous RTP
connect<parameter>(fromPS, async(first.in[0]));
//Asynchronous RTP, default for output
connect<parameter>(second.inout[0], toPS);
//Asynchronous RTP
connect<parameter>(async(second.inout[0]), toPS);
//Synchronous RTP
connect<parameter>(sync(second.inout[0]), toPS);
```

Specialized Graph Constructs

This chapter describes several graph constructs that help when modeling specific scenarios.

Lookup Tables

Static File-Scoped Tables

Kernel functions can use private, read-only data structures that are accessed as file-scoped variables. The compiler allocates a limited amount of static heap space for such data. As an example, consider the following header file (`user_parameter.h`).

```
#ifndef USER_PARAMETER_H
#define USER_PARAMETER_H

#include <adf.h>
static int16 lutarray[8] = {1,2,3,4,5,6,0,0} ;

#endif
```

This header file can be included in the kernel source file and the lookup table can be accessed inside a kernel function directly. The `static` modifier ensures that the array definition is local to this file. The AI Engine compiler allocates this array in static heap space for the processor where this kernel is used.

```
#include <aie_api/aie.hpp>
#include "user_parameter.h"

void simple_lut(adf::input_buffer<int16> &in, adf::output_buffer<int16> &out){
    aie::vector<int16,32> sbuff;
    aie::vector<int16,8> coeffs=aie::load_v<8>((int16*)lutarray);
    auto inIter=aie::begin_vector<32>(in);
    sbuff=*inIter++;
    auto acc = aie::sliding_mul<8,16>(coeffs, 0, sbuff, 0);
    auto outIter=aie::begin_vector<8>(out);
    *outIter+=acc.to_vector<int16>(0);
}
```

Global Graph-Scoped Tables

While the previous example only includes an eight entry look-up table accessed as a global variable, many other algorithms require much larger look-up tables. Because AI Engine local memory is at a premium, it is much more efficient for the AI Engine compiler to manage the look-up table explicitly for specific kernels than to leave a large amount of stack or heap space on every processor. Such tables should *not* be declared static in the kernel header file.

```
#ifndef USER_PARAMETER_H
#define USER_PARAMETER_H

#include <adf.h>

int16 lutarray[8] = {1,2,3,4,5,6,0,0} ;

#endif
```

The kernel source continues to include the header file and use the table as before. But, now you must declare this table as `extern` in the graph class header and use the `parameter::array(...)` function to create a parameter object explicitly in the graph. You also need to attach this parameter object to the kernel as shown in the following code:

```
#include <adf.h>
extern int16 lutarray[8];
class simple_lut_graph : public adf::graph {
public:
    adf::kernel k;
    adf::parameter p;

    simple_lut_graph() {
        k = adf::kernel::create(simple);
        p = adf::parameter::array(lutarray);
    }
};
```

```

    adf::connect(p,k);
    ...
}
}

```

Including this explicit specification of the look-up table in the graph description ensures that the compiler is aware of the requirement to reserve a suitably sized piece of memory for the look-up table when it allocates memory for kernel input and output buffers.

Following is an example graph code for constraining LUTs to different banks to avoid conflicts:

```

const int size=1024;
extern int16 lnr_lutab[size*2*2];
extern int16 lnr_lutcd[size*2*2];

class adaptive_graph : public adf::graph
{
public:
    adf::input_plio in;
    adf::output_plio out;
    adf::kernel k;

    adaptive_graph(){
        k = adf::kernel::create(linear_approx);
        adf::source(k) = "linear_approx.cc";
        in=adf::input_plio::create("Datain10", adf::plio_64_bits, "data/input1.txt");
        out=adf::output_plio::create("Dataout1", adf::plio_64_bits, "data/output1.txt");
        adf::runtime<ratio>(k) = 0.8;

        auto buf_lnr_ab = adf::parameter::array(lnr_lutab);
        auto buf_lnr_cd = adf::parameter::array(lnr_lutcd);
        adf::connect(buf_lnr_ab,k);
        adf::connect(buf_lnr_cd,k);
        adf::location<adf::parameter>(buf_lnr_ab)={ adf::address(8,1,0x8000) };//optional
        adf::location<adf::parameter>(buf_lnr_cd)={ adf::address(8,1,0xC000) };//optional
        adf::location<adf::buffer>(k.out[0])={ adf::address(8,0,0x8000),adf::address(8,0,0xC000) };//optional
        adf::location<adf::stack>(k)={ adf::address(8,1,0x4000) };//optional
        adf::location<adf::buffer>(k.in[0])={ adf::address(8,0,0x0000), adf::address(8,0,0x4000) };//optional

        adf::connect(in.out[0], k.in[0]);
        adf::connect(k.out[0], out.in[0]);
        adf::dimensions(k.in[0])={1024};//elements
        adf::dimensions(k.out[0])={1024};//elements
    }
};

```

Shared Graph-Scoped Tables

Sometimes, the same table definition is used in multiple kernels. Because the AI Engine architecture is a distributed address-space architecture, each processor binary image that executes such a kernel needs to have that table defined in its own local memory. To get the correct graph linkage spread across multiple processors, you must declare the tables as `extern` within the kernel source file as well as the graph class definition file. Then, the actual table definition needs to be specified in a separate header file that is attached as a property to the kernel as shown below.

```

#include <adf.h>
extern int16 lutarray[8];
class simple_lut_graph : public adf::graph {
public:
    kernel k;
    parameter p;

    simple_lut_graph() {
        k = kernel::create(simple);
        p = parameter::array(lutarray);
        connect(p,k);

        std::vector<std::string> myheaders;
    }
};

```

```

myheaders.push_back("./user_parameter.h");
headers(k) = myheaders;
location<parameter>(p)={address(0,1,0x1000)};
...
}
}

```

This ensures that the header file that defines the table is included in the final binary link wherever this kernel is used without causing re-definition errors.

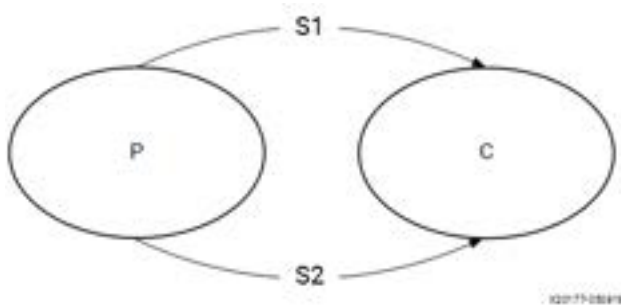
Note: Large lookup tables (>32 KB) are not supported.

Note: Shared data either has to be managed explicitly as runtime parameters or declared at the file scope, which is shared across all kernels mapped to the same AI Engine.

FIFO Depth

The AI Engine architecture uses stream data extensively for DMA-based I/O, for communicating between two AI Engines, and for communicating between the AI Engine and the programmable logic (PL). This raises the potential for a resource deadlock when the data flow graph has reconvergent data paths. If the pipeline depth of one path is longer than the other, the producer kernel can stall and might not be able to push data into the shorter path because of back pressure. At the same time, the consumer kernel is waiting to receive data on the longer path due to the lack of data. If the order of data production and consumption between two data paths is different, a deadlock can happen even between two kernels that are directly connected with two data paths. The following figure illustrates the paths.

Figure: Producer and Consumer Kernels with Reconvergent Streams



If the producer kernel is trying to push data on stream S1 and runs into back pressure while the consumer kernel is still trying to read data from stream S2, a deadlock occurs. A general way to fix this situation is to create more buffering in the paths that have back pressure in the source code by using a `fifo_depth` constraint on a connection.

```

p = kernel::create(producer);
c = kernel::create(consumer);
connect s1(p.out[0], c.in[0]);
connect s2(p.out[1], c.in[1]);
fifo_depth(s1) = 20;
fifo_depth(s2) = 10;

```

Note: The `fifo_depth()` constraint is only valid on stream and buffer type kernel connections. It is not available on cascade stream type connections, because there is a two deep 512-bit wide FIFO on both the input and output cascade streams that allows storing up to four values between AI Engines.

Stream Switch FIFO

The AI Engine has two 32-bit input AXI4-Stream interfaces and two 32-bit output AXI4-Stream interfaces. Each stream is connected to a FIFO both on the input and output side, allowing the AI Engine to have a four word (128-bit) access per four cycles, or a one word (32-bit) access per cycle on a stream. A `fifo_depth()` constraint specification below 40 allocates FIFOs from the stream switch. The following is an example of a FIFO allocation on the stream switch requesting a `fifo_depth(32)`.

Figure: FIFO Allocation on Stream Switch



DMA FIFO

A `fifo_depth()` constraint specification above 40 allocates FIFOs from memory, known as DMA FIFOs. The following is an example of a FIFO allocation for a request of `fifo_depth(128)` bytes which is allocated in memory.

Figure: DMA FIFO Allocation



Note: TLAST is dropped when data goes through DMA FIFO.

Note: Write to DMA FIFO must be continuous or multiple of 4 words (4 words = 128 bits).

You can also specify the type of FIFO allocated, whether stream switch or DMA, as well as their locations. More information can be found in [FIFO Location Constraints](#).

AI Engine Tile DMA Performance

In high throughput use cases where the AI Engine and PL throughput is close to maximum, when using a DMA FIFO, and the PL communicates with the DMA FIFO in an asynchronous PL to AI Engine clock relationship, the read side must occasionally wait for data due to nature of a single DMA FIFO. This can lead to slightly lower than 100% throughput on the AI Engine. Some of the recommended ways to avoid the small loss in throughput are as follows.

- Choose a `fifo_depth` constraint of less than or up to 40 at the AI Engine-PL boundaries on streaming connections with a slack of 40 or less.
- Add a small asynchronous FIFO in the PL to shift the alignment into the AI Engine clock domain.
- Use a synchronous PL clock to the AI Engine. Use a 128-bit AXI4-Stream interface from the PL and use a PL clock at integer multiples of the AI Engine frequency.

Kernel Bypass


A bypass encapsulator construct is used to execute a kernel conditionally. The control of the bypass is done through a runtime parameter.

The bypass runtime control input `bp` : 0 for no bypass and 1 for bypass. In addition to the control parameter, the external connections of a bypassed kernel or a graph are directed to the external ports of the bypass construct itself. Internally, the bypass construct is connected to the bypassed kernel or the graph automatically by the compiler.

The `negate` modifier can be used to denote that the input control is negated. By default, the control is synchronous RTP. To use asynchronous RTP, use the `async` modifier.

The following example shows the required coding.

```
input_port control;
input_port in;
output_port out;
bypass b;
kernel k;
k = kernel::create(filter);
dimensions(k.in[0])={32};
dimensions(k.out[0])={32};
...
b = bypass::create(k);
connect<parameter> (control, async(negate(b.bp)));
connect(in, b.in[0]);
connect(b.out[0], out);
```

 **Note:** For the bypass to work correctly, a one-to-one correspondence between the input and output buffer ports is required, both in type and size.

Explicit Packet Switching

Just as multiple AI Engine kernels can share a single processor and execute in a interleaved manner, multiple stream connections can be shared on a single physical channel. This mechanism is known as Packet Switching. The AI Engine architecture and compiler work together to provide a programming model where up to 32 stream connections can share the same physical channel.

The Explicit Packet Switching feature allows fine-grain control over how packets are generated, distributed, and consumed in a graph computation. Explicit Packet Switching is typically recommended in cases where many low bandwidth streams from common PL source can be distributed to different AI Engine destinations. Similarly, many low bandwidth streams from different AI Engine sources to a common PL destination can also take advantage of this feature. Because a single physical channel is shared between multiple streams, you minimize the number of AI Engine to PL interface streams used.

A packet stream can be created from one AI Engine kernel to multiple destination kernels, from multiple AI Engine kernels to a single destination kernel, or between multiple AI Engine kernels and multiple destination kernels.

This section describes graph constructs to create packet-switched streams explicitly in the graph, and provide multiple examples on packet switching use cases.

Packet Switching Graph Constructs

Packet-switched streams are essentially multiplexed data streams that carry different types of data at different times. Packet-switched streams do not provide deterministic latency due to the potential for resource contention with other packet-switched streams. The multiplexed data flows in units of packets with a 32-bit packet header and a variable number of payload words. A header word needs to be sent before the actual payload and the TLAST signal is required on the last word of the packet. Two new data types called `input_pktstream` and `output_pktstream` are introduced to represent the multiplexed data streams as input to or output from a kernel, respectively. More details on the packet headers and data types can be found in [Packet Stream Operations](#).

To explicitly control the multiplexing and de-multiplexing of packets, two new templated node classes, `pktsplit<n>` and `pktmerge<n>`, are added to the ADF graph library. A node instance of class `pktmerge<n>` is a `n:1` multiplexer of `n` packet streams producing a single packet stream. A node instance of class `pktsplit<n>` is a `1:n` de-multiplexer of a packet stream producing `n` different packet streams. The maximum number of allowable packet streams is 32 on a single physical channel ($n \leq 32$).

A kernel can receive packets of data either as buffers of data or as `input_pktstream`. And a kernel can send packets of data either as buffers of data or as `output_pktstream`.

To connect from ports, local buffers or packet streams to ports, local buffers or packet streams, use a `connect` construct, such as:

```
connect (<SOURCE>.out[0], <DEST>.in[0]);
```

When a kernel receives packets of data as a buffer of data, the header and TLAST are dropped prior to the kernel receiving the buffer of data. If the kernel writes an output buffer of data, the packet header and TLAST are automatically inserted, when the buffer is transferred by DMA to a packet stream.

However, if the kernel receives `input_pktstream` of data, the kernel needs to process the packet header and TLAST, in addition to the

packet data. Similarly, if the kernel sends an `output_pktstream` of data, the kernel needs to insert the packet header and TLAST, in addition to the packet data into the output stream.

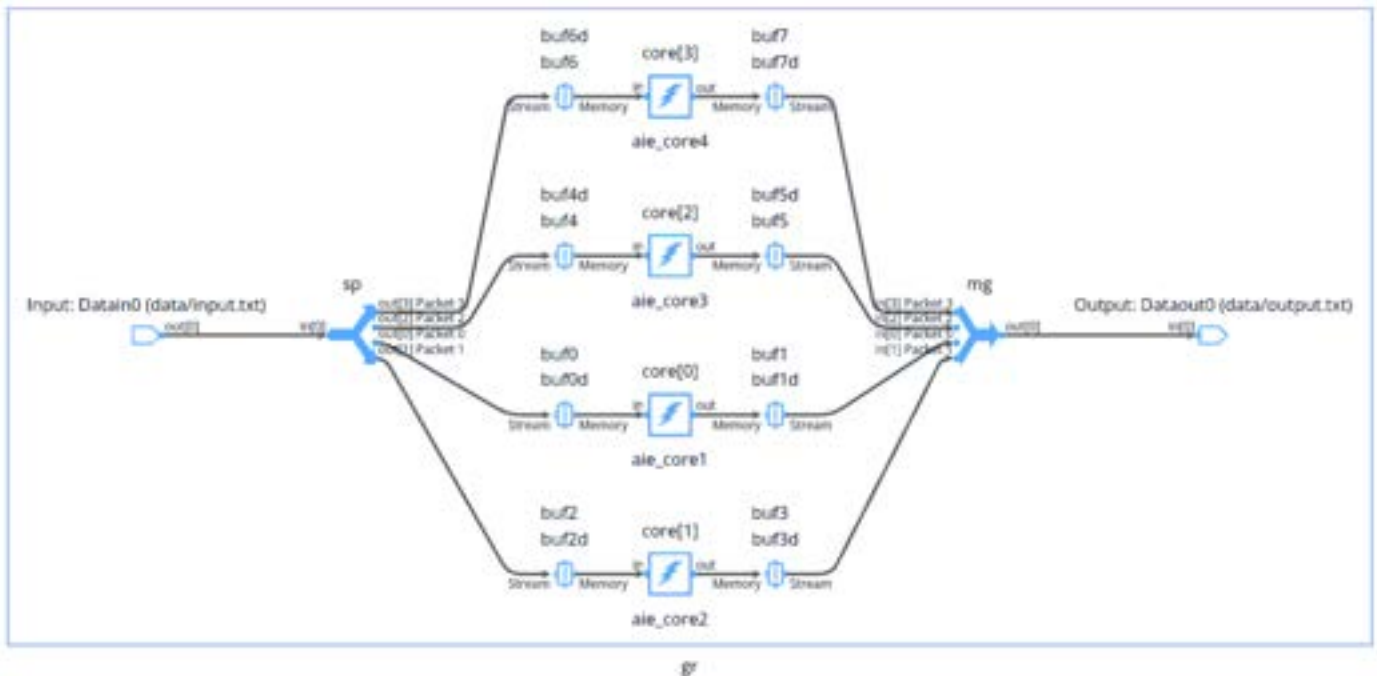
These concepts are illustrated in the following example.

```
class ExplicitPacketSwitching: public adf::graph {
private:
    adf:: kernel core[4];
    adf:: pktsplit<4> sp;
    adf:: pktmerge<4> mg;
public:
    adf::input_plio in;
    adf::output_plio out;
    mygraph() {
        core[0] = adf::kernel::create(aie_core1);
        adf::source(core[0]) = "aie_core1.cpp";

        .....
        sp = adf::pktsplit<4>::create();
        mg = adf::pktmerge<4>::create();
        for(int i=0;i<4;i++){
            adf::connect(sp.out[i], core[i].in[0]);
            adf::connect(core[i].out[0], mg.in[i]);
        }
        adf::connect(in.out[0], sp.in[0]);
        adf::connect(mg.out[0], out.in[0]);
    }
};
```

The graph has one input PLIO port and one output PLIO port. The input packet stream from the PL is split four ways and input to four different AI Engine kernels. The output streams from the four AI Engine kernels are merged into one packet stream which is output to the PL. The Vitis IDE Graph view of the code is shown as follows.

Figure: Graph View



When the AI Engine kernel uses buffer port input, the packet stream to the buffer is automatically decoded and packet data are transferred to the buffer via DMA. When the AI Engine kernel uses buffer port output and the buffer is connected to packet stream, the packet header is inserted automatically and the last sample of the buffer data is enabled with TLAST to denote the completion of the packet.

To read data from a packet stream in the AI Engine, the `input_pktstream` interface is used. However, this data is read as an integer value by default. If other data types are to be read, the data can first be read as an integer value and then cast to the desired data type.

Following is an example kernel code that accepts and transfers floating-point data type:

```
const uint32 pktType=0;
```

```

void aie_core1_float(input_pktstream *in,output_pktstream *out){
    readincr(in);//read header and discard
    uint32 ID=getPacketid(out,0);//for output pktstream
    writeHeader(out,pktType,ID); //Generate header for output

    bool tlast;
    for(int i=0;i<4;i++){
        int32 tmp=readincr(in,tlast);//read data as integer type
        float tmp_f=reinterpret_cast<float&>(tmp);//Reinterpret memory as float
        tmp_f+=1.0f;
        writeincr(out,tmp_f,i==3);//TLAST=1 for last word
    }
}

```

The input data should also be in integer format for simulation. For example, if float values 0.0f, 1.0f, 2.0f and 3.0f are sent to the kernel, they are converted into integer values in the input data file:

```

0
1065353216
1073741824
1077936128

```

After the kernel execution, the float output values are 1.0f, 2.0f, 3.0f and 4.0f. In a simulation output data file, the output values are in integer format, as follows:

```

1065353216
1073741824
1077936128
1082130432

```

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

Example of Packet Switching between PL and AI Engine

The PL to AI Engine interface allows multiple low bandwidth PL sources to use packet switching to distribute data to different destinations in the AI Engine. The same interface can also merge multiple AI Engine packet streams and transmit them to the PL.

To view the packet header format, see [Packet Processing](#) in the *AI Engine Kernel and Graph Programming Guide* ([UG1079](#)). If a packet originates from the PL, it is the PL's responsibility to generate the correct packet header and data with the TLAST field set to true for the last sample of the packet. Conventionally, packets created in the programmable logic can have a row and column initialization of -1,-1, which indicates that the packet is not possible from inside AI Engine.

If the PL receives packets from the AI Engine, it needs to decode the header to determine the packet's origin and treat it accordingly. For example, based on the decoded packet ID, the packet data is dispatched to the correct destinations.

The graph from the previous section is used as an example to illustrate packet switching connections between the PL and the AI Engine:

```

class PLPacketGraph: public adf::graph {
private:
    adf:: kernel core[4];
    adf:: pktsplit<4> sp;
    adf:: pktmerge<4> mg;
public:
    adf::input_plio in;
    adf::output_plio out;
    mygraph() {
        core[0] = adf::kernel::create(aie_pktstream_core1);
        core[1] = adf::kernel::create(aie_pktstream_core2);
        core[2] = adf::kernel::create(aie_pktstream_core3);
        core[3] = adf::kernel::create(aie_pktstream_core4);
        adf::source(core[0]) = "aie_pktstream_core1.cpp";
        adf::source(core[1]) = "aie_pktstream_core2.cpp";
        adf::source(core[2]) = "aie_pktstream_core3.cpp";
        adf::source(core[3]) = "aie_pktstream_core4.cpp";

        in=input_plio::create("Datain0", plio_32_bits, "data/input.txt");
        out=output_plio::create("Dataout0", plio_32_bits, "data/output.txt");
    }
}

```

```

    sp = adf::pktsplit<4>::create();
    mg = adf::pktmerge<4>::create();
    for(int i=0;i<4;i++){
        adf::runtime<ratio>(core[i]) = 0.9;
        adf::connect(sp.out[i], core[i].in[0]);
        adf::connect(core[i].out[0], mg.in[i]);
    }
    adf::connect(in.out[0], sp.in[0]);
    adf::connect(mg.out[0], out.in[0]);
}
};

```

An AI Engine kernel code example is as follows.

```

const uint32 pktType=0;
void aie_pktstream_core1(input_pktstream *in,output_pktstream *out){
    readincr(in);//read header and discard because only the correct packet arrives
    uint32 ID=getPacketid(out,0);//for output pktstream, index always =0
    writeHeader(out,pktType,ID); //Generate header for output

    bool tlast;
    for(int i=0;i<8;i++){
        int32 tmp=readincr(in,tlast);
        tmp+=1;
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}

```

The PL kernel doesn't have a helper function, such as `getPacketid()`, to extract a specific destination's packet ID. To obtain the correct packet ID, the compiled report files, `Work/temp/packet_ids_c.h` and `Work/temp/packet_ids_v.h`, can be used in C/C++ or Verilog source files.

For example, the generated `Work/temp/packet_ids_c.h` report for the above graph is:

```

#define Datain0_0 0
#define Datain0_1 1
#define Datain0_2 2
#define Datain0_3 3
#define Dataout0_0 0
#define Dataout0_1 1
#define Dataout0_2 2
#define Dataout0_3 3

```

The macro `Datain0_0` connects the PL to the 0th index of the `pktsplit` output, while the macro `Datain0_1` connects it to the first index. It's worth noting that the macro name remains the same across different compilations unless there is a change in the graph structure. However, the macro value (such as the packet ID) can differ among compilations.

Based on these macro names, the following example HLS helper function can be written for the PL kernel as shown:

```

#include "packet_ids_c.h"

static const unsigned int pktType=0;
static const int PACKET_NUM=4; //How many kernels do packet switching
static const int PACKET_LEN=8; //Length for a packet

static const unsigned int packet_ids[PACKET_NUM]={Datain0_0, Datain0_1, Datain0_2, Datain0_3}; //macro values
are generated in packet_ids_c.h

ap_uint<32> generateHeader(unsigned int pktType, unsigned int ID){
#pragma HLS inline
    ap_uint<32> header=0;
    header(4,0)=ID;
    header(11,5)=0;
    header(14,12)=pktType;
    header[15]=0;
    header(20,16)=-1;//source row
}

```

```

        header(27,21)=-1;//source column
        header(30,28)=0;
        header[31]=header(30,0).xor_reduce()?(ap_uint<1>)0:(ap_uint<1>)1;
        return header;
    }
    void hls_packet_sender(.....){
        for(unsigned int iter=0;iter<num;iter++){
            for(int i=0;i<PACKET_NUM;i++){//Iterate on PL kernels that do packet switching
                unsigned int ID=packet_ids[i]; //get packet ID from AIE compilation
                ap_uint<32> header=generateHeader(pktType,ID); //packet header
                ap_axiu<32,0,0,0> tmp;
                tmp.data=header;
                tmp.keep=-1;
                tmp.last=0;
                out.write(tmp);//write packet header
                for(int j=0;j<PACKET_LEN;j++){ //generate packet data
                    .....
                }
            }
        }
    }

```

For more examples on packet switching between the PL and the AI Engine, see [Vitis Tutorials: AI Engine Development: Packet Switching](#).

Example of Packet Switching from an AI Engine to Multiple AI Engines

If the performance allows, AI Engine kernels can be used to dispatch packets to multiple destinations. This section specifically addresses the AI Engine kernel code necessary to manage the packets, particularly the packet header. To illustrate this process, the following graph is considered:

```

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

class mygraph: public adf::graph {
private:
    adf:: kernel core[4],core_m;

    adf:: pktsplit<4> sp;
public:
    adf::input_plio in;
    adf::output_plio out[4];
    mygraph() {
        core[0] = adf::kernel::create(aie_core1);
        core[1] = adf::kernel::create(aie_core2);
        core[2] = adf::kernel::create(aie_core3);
        core[3] = adf::kernel::create(aie_core4);
        core_m = adf::kernel::create(aie_master_core);
        adf::source(core[0]) = "aie_core1.cpp";
        adf::source(core[1]) = "aie_core2.cpp";
        adf::source(core[2]) = "aie_core3.cpp";
        adf::source(core[3]) = "aie_core4.cpp";
        adf::source(core_m) = "aie_master_core.cpp";
        adf::runtime<ratio>(core_m) = 0.9;
        adf::repetition_count(core_m)=4;
        adf::location<adf::kernel>(core_m)=adf::tile(0,0);

        in=adf::input_plio::create("Datain0", adf::plio_32_bits, "data/input.txt");
        sp = adf::pktsplit<4>::create();
        for(int i=0;i<4;i++){
            out[i]=adf::output_plio::create("Dataout"+std::to_string(i), plio_32_bits, "data/
output"+std::to_string(i)+".txt");
            adf::runtime<ratio>(core[i]) = 0.9;
            adf::repetition_count(core[i])=1;
            adf::connect<adf::pktstream > (sp.out[i], core[i].in[0]);
            adf::connect<> (core[i].out[0], out[i].in[0]);
        }

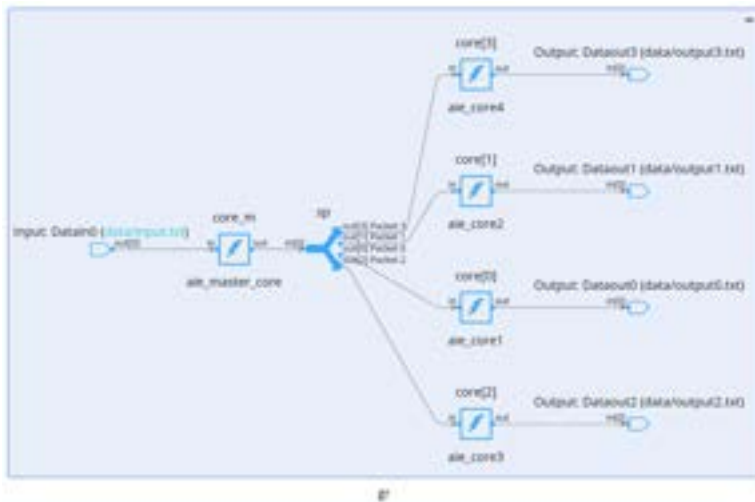
        adf::connect<> (in.out[0], core_m.in[0]);
        adf::connect<adf::pktstream> (core_m.out[0], sp.in[0]);
    }
}

```

```
};
```

The graph view of the above graph is as follows:

Figure: Graph View of AI Engine to Multiple AI Engines



Below is the code for the AI Engine kernel responsible for sending packets:

```
#include <aie_api/aie.hpp>
const uint32 pktType=0;

void aie_master_core(input_stream<int32> *in,output_pktstream *out){
    int32 user_id=readincr(in); //this is somehow specified by user
    uint32 ID=getPacketid(out,user_id);//get packet ID from user specified destination
    writeHeader(out,pktType,ID); //Generate header for output
    for(int i=0;i<8;i++){
        int32 tmp=readincr(in);
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}
```

The pktsplit connection is established at compile time. It is fixed and cannot be modified. You can use the `getPacketid()` API to extract the packet ID from a specific index of the pktsplit output. This ID is automatically generated during compilation and can be used to transfer the packet from a source to a target connection in the graph.

Note: Only packets matching the extracted packet ID reach the intended receiver kernel. When received, the packet's header can be accessed and discarded as necessary.

Following is an example code of the receiver kernel:

```
#include <aie_api/aie.hpp>

void aie_core1(input_pktstream *in,output_stream<int32> *out){
    readincr(in);//read header and discard

    bool tlast;
    for(int i=0;i<8;i++){
        int32 tmp=readincr(in,tlast);
        tmp+=1;
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}
```

Example on Packet Switching from Multiple AI Engines to an AI Engine

AI Engine kernels can be used to merge packets from different sources before sending the data to destinations like PL kernels. This section covers the AI Engine kernel code that manages packets and their headers. The following graph is used as an example:

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

```

class mergegraph: public adf::graph {
private:
    adf:: kernel core[4],core_m;

    adf:: pktmerge<4> mg;
public:
    adf::input_plio in[4];
    adf::output_plio out;
    mergegraph() {
        core[0] = adf::kernel::create(aie_core1);
        core[1] = adf::kernel::create(aie_core2);
        core[2] = adf::kernel::create(aie_core3);
        core[3] = adf::kernel::create(aie_core4);
        core_m = adf::kernel::create(aie_combine_core);
        adf::source(core[0]) = "aie_core1.cpp";
        adf::source(core[1]) = "aie_core2.cpp";
        adf::source(core[2]) = "aie_core3.cpp";
        adf::source(core[3]) = "aie_core4.cpp";
        adf::source(core_m) = "aie_combine_core.cpp";
        adf::runtime<ratio>(core_m) = 0.9;
        adf::repetition_count(core_m)=4;

        out=output_plio::create("Dataout0", plio_32_bits, "data/output.txt");

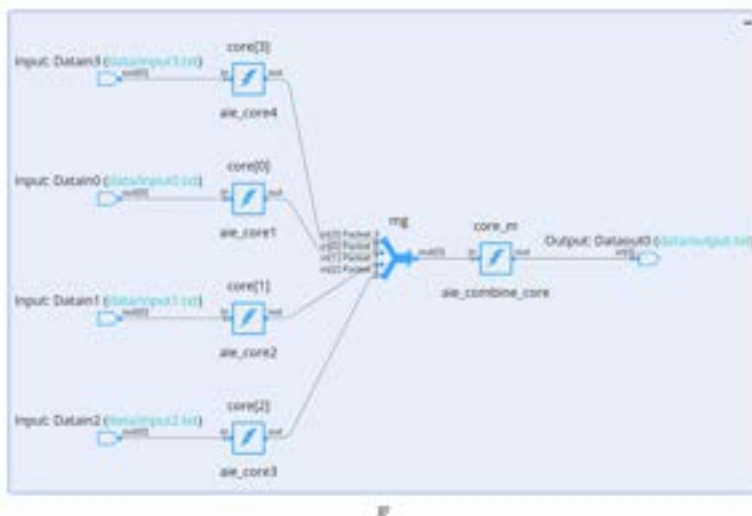
        mg = adf::pktmerge<4>::create();
        for(int i=0;i<4;i++){
            in[i]=adf::input_plio::create("Datain"+std::to_string(i), plio_32_bits, "data/
input"+std::to_string(i)+".txt");
            adf::runtime<ratio>(core[i]) = 0.9;
            adf::repetition_count(core[i])=1;
            adf::connect (in[i].out[0], core[i].in[0]);
            adf::connect<adf::pktstream > (core[i].out[0], mg.in[i]);
        }

        adf::connect<adf::pktstream> (mg.out[0], core_m.in[0]);
        adf::connect<> (core_m.out[0], out.in[0]);
    }
};

```

The graph view of the above graph is as follows:

Figure: Graph View of Multiple AI Engines to AI Engine



The code for the combining AI Engine kernels is as follows:

```
#include <aie_api/aie.hpp>
```

```

const uint32 pktType=0;
const int PACKET_NUM=4;
static uint32 ID_TABLE[PACKET_NUM];
static int iteration=0;
void aie_combine_core(input_pktstream *in,output_stream<int32> *out){
    if(iteration==0){
        for(int i=0;i<PACKET_NUM;i++){
            uint32 packet_id=getPacketid(in,i);
            ID_TABLE[packet_id%PACKET_NUM]=i;
            printf("merge input index=%d, compiler assigned packet ID=%d\n",i,packet_id);
        }
        iteration++;
    }

    int32 header=readincr(in);
    uint32 ID=header & 0x1f;//packet id from header
    uint32 index=ID_TABLE[ID];//get merge input index, which is fixed by graph code.

    for(int i=0;i<8;i++){
        int32 tmp=readincr(in);
        if(index==0){//operate based on the index of the merge input
            ...
        }else if(index==1){
            ...
        }else if(index==2){
            ...
        }else if(index==3){
            ...
        }
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}

```

The connection of the pktmerge is predetermined by the graph code. However, the packet ID for each pktmerge input might change with subsequent compilations. Therefore, the kernel code constructs a table called ID_TABLE that maps the packet ID to an index. This index can then be used to perform operations and is fixed, provided the graph connection remains unchanged.

Following is an example code of the producer kernel:

```

#include <aie_api/aie.hpp>

const uint32 pktType=0;

void aie_core2(input_stream<int32> *in,output_pktstream *out){
    uint32 ID=getPacketid(out,0); //index always =0 in the kernel code
    writeHeader(out,pktType,ID); //Generate header for output

    for(int i=0;i<8;i++){
        int32 tmp=readincr(in);
        tmp+=2;
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}

```

For all producer kernels, the index parameter in the getPacketid API for output_pktstream is always set to 0, as shown in the above code snippet

Packet Split and Merge Connections

Multiple streams can share routing by connecting pktmerge to pktsplit. When buffers are transferred via pktmerge connected to pktsplit, each pktmerge.in[i] is routed to the corresponding pktsplit.out[i]. The in-degree of pktmerge should be equal to the out-degree of pktsplit. An example graph code is as follows:

```

for (int i=0; i<WAYS; i++) {
    connect<>(plioIn[i].out[0], kOut[i].in[0]);
    connect<>(kOut[i].out[0], merge.in[i]);
}

```

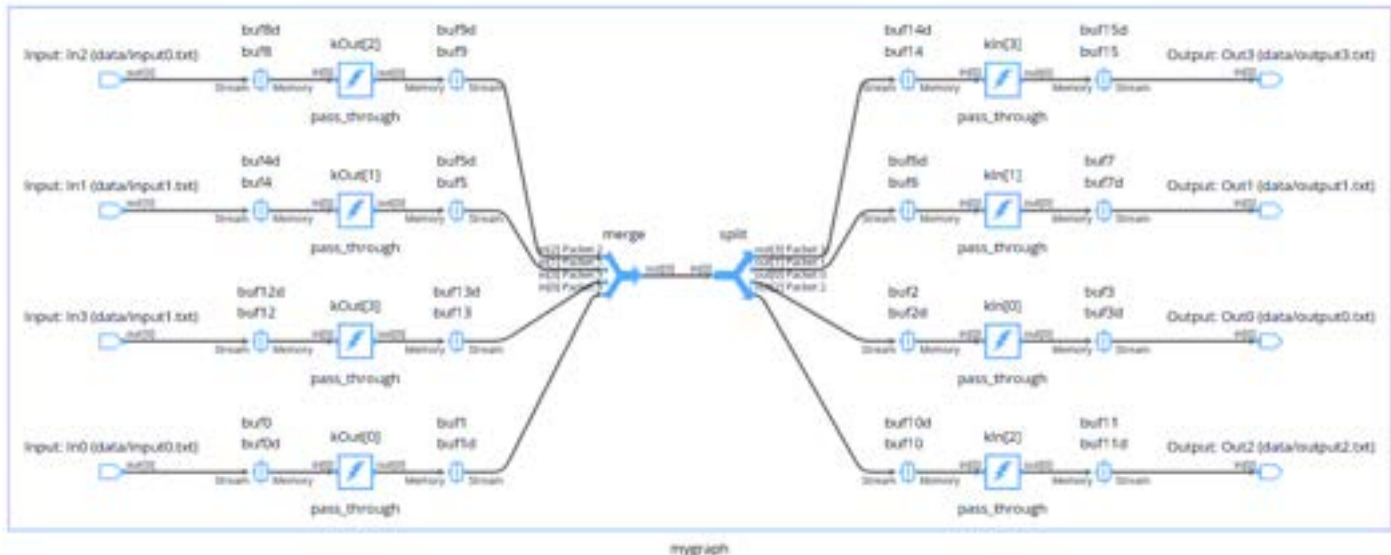
```

connect<>(split.out[i], kIn[i].in[0]);
connect<>(kIn[i].out[0], plioOut[i].in[0]);
}
connect<> (merge.out[0], split.in[0]);

```

The graph view is shown in the following figure.

Figure: Pktmerge to Pktsplit Graph View



Packet Split and Merge Sizes

Currently, packet switching up to 32 streams is supported. A maximum 32 to 1 pktmerge, and 1 to 32 pktsplit are supported. Using packet switching with large fanout/fanin (16/32 streams) can be resource expensive, and care should be taken when using these in designs.

Recommended: AMD recommends that you avoid unnecessarily splitting packets and immediately merging them again. This adds complexity on the router solution.

Packet Split and Merge Broadcast

When streams pass through pktmerge to pktsplit, broadcast is supported from pktsplit. In this situation, pktmerge.in[i] is broadcast to the corresponding multiple destinations that connect to pktsplit.out[i]. Following is an example code, where merge.in[WAYS-1] is broadcast to split.out[WAYS-1] and split.out[WAYS]:

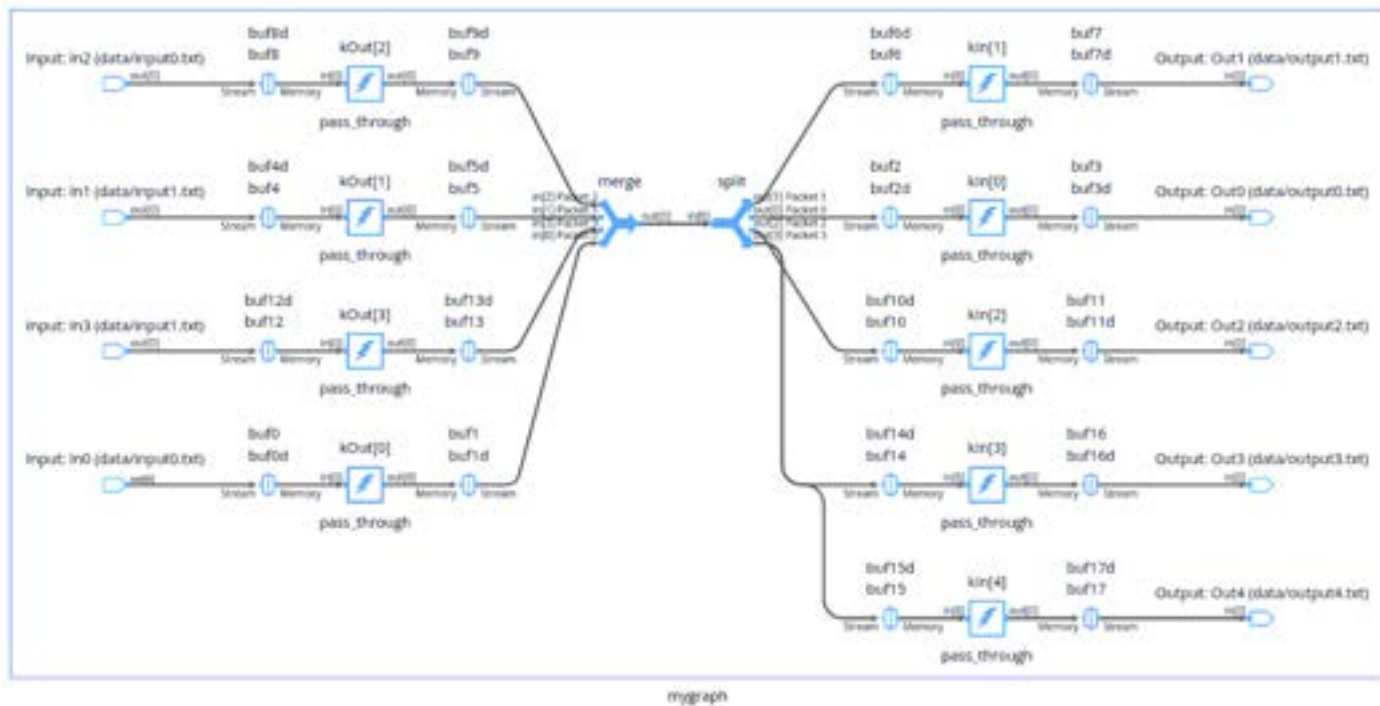
```

for (int i=0; i<WAYS; i++) {
    connect<>(plioIn[i].out[0], kOut[i].in[0]);
    connect<>(kOut[i].out[0], merge.in[i]);
    connect<>(split.out[i], kIn[i].in[0]);
    connect<>(kIn[i].out[0], plioOut[i].in[0]);
}
connect<>(split.out[WAYS-1], kIn[WAYS].in[0]);
connect<>(kIn[WAYS].out[0], plioOut[WAYS].in[0]);
connect<> (merge.out[0], split.in[0]);

```

The graph view is shown in the following figure.

Figure: Pktmerge to Pktsplit Graph View with Broadcast



Packet Switching and the aiesimulator

Explicit packet switching is supported by the `aiesimulator`. Consider the example of the previous graph that expects packet switched data from the PL; the data is split inside the AI Engine and sent to four AI Engine kernels. On the output side the four kernel outputs are merged into one output stream to the PL.

The input data file from the PL contains all the packet switched data from the PL for the four AI Engine kernels in the previous example. It contains the data for different kernels, packet by packet. Each packet of data is for one iteration of an AI Engine kernel. The data format is as follows.

```
2415853568
0
1
2
3
4
5
6
TLAST
7
```

Here, 2415853568 is 0x8fff0000 in hex format. The five least significant bits are the packet ID, 0 in this case. The last data in the packet has the keyword TLAST, which denotes the last data for the input of the kernel.

Note: Integer values only are accepted as a packet header ID for the PL packet inputs to the `aiesimulator`.

You can construct the header for each packet manually, or write helper functions to generate the header. The AI Engine compiler generates a packet switching report file `Work/reports/packet_switching_report.json` that lists the packet IDs used in the graph. In addition it also generates `Work/temp/packet_ids_c.h` and `Work/temp/packet_ids_v.h` header files that can be included in your C or Verilog kernel code.

Location Constraints

Kernel Location Constraints

When building large graphs with multiple subgraphs, it is sometimes useful to control the exact mapping of kernels to AI Engines, either relative to other kernels or in an absolute sense. The AI Engine compiler provides a mechanism to specify location constraints for kernels, which when used with the C++ template class specification, provides a powerful mechanism to create a robust, scalable, and predictable mapping of your graph onto the AI Engine array. It also reduces the choices for the mapper to try, which can considerably speed up the mapper. Consider the following graph specification:

```
#include <adf.h>
#include "kernels.h"
#define NUMCORES (COLS*ROWS)
```

```

template <int COLS, int ROWS, int STARTCOL, int STARTROW>
class indep_nodes_graph1 : public adf::graph {
public:
    adf::kernel kr[NUMCORES];
    adf::port<input> datain[NUMCORES] ;
    adf::port<output> dataout[NUMCORES] ;

    indep_nodes_graph1() {
        for (int i = 0; i < COLS; i++) {
            for (int j = 0; j < ROWS; j++) {
                int k = i*ROWS + j;
                kr[k] = adf::kernel::create(mykernel);
                adf::source(kr[k]) = "kernels/kernel.cc";
                adf::runtime<ratio>(kr[k]) = 0.9;
                adf::location<adf::kernel>(kr[k]) = adf::tile(STARTCOL+i, STARTROW+j);
            }
        }
        for (int i = 0; i < NUMCORES; i++) {
            adf::connect(datain[i], kr[i].in[0]);
            adf::connect(kr[i].out[0], dataout[i]);
        }
    };
};

```

The template parameters identify a COLS x ROWS logical array of kernels (COLS x ROWS = NUMCORES) that are placed within a larger logical device of some dimensionality starting at (STARTCOL, STARTROW) as the origin. Each kernel in that graph is constrained to be placed on a specific AI Engine. This is accomplished using an absolute location constraint for each kernel placing it on a specific processor tile. For example, the following declaration would create a 1 x 2 kernel array starting at offset (3,2). When embedded within a 4 x 4 logical device topology, the kernel array is constrained to the top right corner.

```
indep_nodes_graph1<1,2,3,2> mygraph;
```

!! Important: Earlier releases used `location<absolute>(k)`, function to specify kernel constraints and `proc(x,y)` function to specify a processor tile location. These functions are now deprecated. Instead, use `adf::location<adf::kernel>(k)` to specify the kernel constraints and `adf::tile(x,y)` to identify a specific tile location.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

Buffer Location Constraints

The AI Engine compiler tries to automatically allocate buffers for buffers, lookup tables, and run-time parameters in the most efficient manner possible. However, you might want to explicitly control their placement in memory. Similar to the kernels shown previously in this section, buffers inferred on a kernel port can also be constrained to be mapped to specific tiles, banks, or even address offsets using location constraints, as shown in the following example.

```

#include <adf.h>
#include "kernels.h"
#define NUMCORES (COLS*ROWS)

template <int COLS, int ROWS, int STARTCOL, int STARTROW>
class indep_nodes_graph2 : public adf::graph {
public:
    adf::kernel kr[NUMCORES];
    adf::port<input> datain[NUMCORES] ;
    adf::port<output> dataout[NUMCORES] ;

    indep_nodes_graph2() {
        for (int i = 0; i < COLS; i++) {
            for (int j = 0; j < ROWS; j++) {
                int k = i*ROWS + j;
                kr[k] = adf::kernel::create(mykernel);
                adf::source(kr[k]) = "kernels/kernel.cc";
                adf::runtime<ratio>(kr[k]) = 0.9;
            }
        }
    };
};

```

```

adf::location<adf::kernel>(kr[k]) = adf::tile(STARTCOL+i, STARTROW+j); // kernel location
adf::location<adf::buffer>(kr[k].in[0]) =
    { adf::address(STARTCOL+i, STARTROW+j, 0x0),
      adf::address(STARTCOL+i, STARTROW+j, 0x2000) }; // double buffer location
adf::location<adf::stack>(kr[k]) = adf::bank(STARTCOL+i, STARTROW+j, 2); // stack location
adf::location<adf::buffer>(kr[k].out[0]) = adf::location<adf::kernel>(kr[k]); // relative buffer
location
}
}

for (int i = 0; i < NUMCORES; i++) {
    adf::connect(datain[i], kr[i].in[0]);
    adf::connect(kr[i].out[0], dataout[i]);
}
};
};

```

In the previous code, the location of double buffers at port `kr[k].in[0]` is constrained to the specific memory tile address offsets that are created using the `address(col, row, offset)` constructor. Furthermore, the location of the system memory (including the sync buffer, stack and static heap) for the processor that executes kernel instance `kr[k]` is constrained to a particular bank using the `bank(col, row, bankid)` constructor. Finally, the tile location of the buffers connected to the port `kr[k].out[0]` is constrained to be the same tile as that of the kernel instance `kr[k]`. Buffer location constraints are applied on kernel buffer ports.

The second example shows how a buffer constraint (for PING and PONG buffers) can be added to a constraints file, and the constraints file specified with `--constraints` option of the AI Engine compiler.

```

{
  "PortConstraints": {
    "gr.k[0].in[0]": {
      "buffers": [{
        "column": 16,
        "row": 1,
        "bankId": 0,
        "offset": 16320
      }, {
        "column": 16,
        "row": 1,
        "bankId": 3,
        "offset": 0
      }]
    }
  }
}

```

If there is an RTP port for a graph, there is an additional selector word that can be optionally constrained as follows (for PING, PONG and selector word):

```

{
  "PortConstraints": {
    "gr.fir24.in[1]": {
      "buffers": [{
        "column": 17,
        "row": 1,
        "bankId": 0,
        "offset": 16320
      }, {
        "column": 17,
        "row": 1,
        "bankId": 3,
        "offset": 0
      }, {
        "column": 18,
        "row": 1,
        "bankId": 0,
        "offset": 16224
      }]
    }
  }
}

```

```

    }
  }
}

```

!! Important: Using location constraint constructors and equality relations between them, you can make fine-grain mapping decisions that the compiler must honor. However, you must be careful because it is easy to create constraints that are impossible for the compiler to satisfy. For example, the compiler cannot allow two buffers to be mapped to the same address offset.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

FIFO Location Constraints

The AI Engine compiler tries to allocate FIFOs in the most efficient manner possible. However, you might want to explicitly control their placement in memory, as shown in the following example. This constraint is useful to preserve the placement of FIFO resources between runs of the AI Engine compiler.

Note the following considerations for FIFO constraints.

- If FIFO constraints are used, the entire depth of the FIFO must be constrained. It is not possible to constrain a portion of the FIFO and leave the rest for the compiler to add.
- If FIFO constraints are added to branching nets, the FIFO constraint should be added to each point-to-point net. If you want to share stream switch FIFOs or DMA FIFOs before the branch, this can be achieved by duplicating the FIFO type and location on each point-to-point net.
- The constraint can be used without a location to specify the desired type of FIFO without specifying a location or depth.

The following example shows how a FIFO constraint can be used in a graph file.

```

two_node_graph() {
  loop0 = adf::kernel::create(loopback_stream);
  loop1 = adf::kernel::create(loopback_stream);
  loop2 = adf::kernel::create(loopback_stream);

  adf::source(loop0) = "loopback_stream.cc";
  adf::source(loop1) = "loopback_stream.cc";
  adf::source(loop2) = "loopback_stream.cc";

  adf::connect net0 (in0, loop0.in[0]);
  adf::connect net1 (loop0.out[0], loop1.in[0]);
  adf::connect net2 (loop1.out[0], loop2.in[0]);
  adf::connect net3 (loop2.out[0], out0);

  adf::runtime(loop0) = 0.9;
  adf::runtime(loop1) = 0.9;
  adf::runtime(loop2) = 0.9;

  adf::fifo_depth(net1) = 32;
  adf::fifo_depth(net2) = 48;

  adf::location< adf::fifo >(net1) = {adf::ss_fifo(adf::shim_tile, 16 , 0, 1),
adf::ss_fifo(adf::shim_tile,17,0,0)};
  adf::location< adf::fifo >(net2) = adf::dma_fifo(adf::aie_tile, 8, 0, 0x0000, 48);
};

```

The second example shows how a FIFO constraint can be added to a constraints file.

```

{
  "PortConstraints": {
    "fifo_locations_records": {
      "dma_fifos": {
        "r1": {
          "tile_type": "core",
          "row": 0,
          "column": 0,
          "size": 16,
          "offset": 8,
          "bankId": 2

```

```
    },
    "r2": {
      "tile_type": "core",
      "row": 0,
      "column": 1,
      "size": 16,
      "offset": 9
    },
    "r4": {
      "tile_type": "mem",
      "row": 2,
      "column": 4,
      "size": 16,
      "offset": 6,
      "bankId": 2
    }
  },
  "stream_fifos": {
    "r3": {
      "tile_type": "shim",
      "row": 1,
      "column": 3,
      "channel": 1
    }
  }
},
"mygraph.k2.in[0]": {
  "fifo_locations": ["r1", "r2", "r3"]
},
"mygraph.k4.in[0]": {
  "fifo_locations": ["r1", "r2", "r4"]
}
}
```

PLIO and GMIO Location Constraints

In AI Engine graphs, when defining external stream connections that cross the AI Engine to the programmable logic (PL) boundary via PLIOs or to the NoC (PL) via GMIOs, it is beneficial to specify both the interface tile column and the exact channel within the interface tile. This specification can impact throughput and latency, optimizing the overall performance.

When setting these constraints, it is essential to select columns in the AI Engine that have direct connections to the PLIO or GMIO shim interface. This ensures optimal functionality and data flow through the AI Engine interfaces.

By default, the AI Engine compiler automatically assigns a channel in the PLIO or GMIO shim interface for data routing. However, you can explicitly specify a channel for further control using the following constraints:

```
location_constraint shim(int column)
location_constraint shim(int column,int channel)
```

The available channel ranges for GMIO and PLIO differ based on the data direction:

Table: Channel Ranges

	Input Channel Range	Output Channel Range
GMIO	0, 1	0, 1
PLIO	0 through 7	0 through 5

Following is an example of how to constrain the location of kernels and interfaces within an AI Engine graph using C++:

```
template <int COL,int ROW>
class SimpleGMIOGraph : public adf::graph
{

private:
public:
```

```
adf::kernel k1,k2;

adf::input_gmio gmin;
adf::output_gmio gmout;
adf::input_plio plin;
adf::output_plio plout;

SimpleGMIOGraph()
{

    k1 = adf::kernel::create(passthrough_buffer<1024>);
    adf::source(k1) = "passthrough.cpp";
    adf::runtime<ratio>(k1) = 0.9;
    adf::location<adf::kernel>(k1) = adf::tile(COL, ROW);

    k2 = adf::kernel::create(passthrough_buffer<1024>);
    adf::source(k2) = "passthrough.cpp";
    adf::runtime<ratio>(k2) = 0.9;
    adf::location<adf::kernel>(k2) = adf::tile(COL+1, ROW);

    gmin = adf::input_gmio::create("gmin", 64, 4000);
    gmout = adf::output_gmio::create("gmout", 64, 4000);
    adf::location<adf::GMIO>(gmin) = adf::shim(COL);
    adf::location<adf::GMIO>(gmout) = adf::shim(COL);

    plin = adf::input_plio::create("plin", adf::plio_64_bits, "data/Input_64.txt", 625);
    plout = adf::output_plio::create("plout", adf::plio_64_bits,"data/Output.txt", 625);
    adf::location<adf::PLIO>(plin) = adf::shim(COL+1);
    adf::location<adf::PLIO>(plout) = adf::shim(COL+1);

    // Connections
    adf::connect(gmin.out[0], k1.in[0]);
    adf::connect(k1.out[0],gmout.in[0]);
    adf::connect(plin.out[0], k2.in[0]);
    adf::connect(k2.out[0], plout.in[0]);
};

};
```

Area Location Constraints

Area location constraints direct the compiler to contain nodes to a custom location in the array. Properties to specify on an area group are described in the following table.

Table: Area Group Properties

Property	Description
group	<p>Specify the collection of group. Each group can be:</p> <ul style="list-style-type: none">• tile-type: Specify the tile-type for the group. Supported tile-types are <code>aie_tile</code>, <code>shim_tile</code>, or <code>memory_tile</code>.• column_min: Column index for lower left corner of the group.• row_min: Row index for lower left corner of the group.• column_max: Column index for upper right corner of the group.• row_max: Row index for upper right corner of the group.
contain_routing	<p>A boolean value that when specified true ensures all routing, including nets between nodes contained in the <code>nodeGroup</code>, is contained within the area group. Default: false.</p>
exclusive_routing	<p>A boolean value that when specified true ensures all routing, excluding nets between nodes from the <code>nodeGroup</code>, is excluded from the area group. Default: false.</p>

Property	Description
exclusive_placement	A boolean value that when specified true prevents all nodes not included in the nodeGroup from being placed within the area group bounding box. Default: false.

The following examples show how an area location constraint can be applied in a graph file.

```
class testGraph1: public adf::graph {

private:
    adf::kernel first;
    adf::kernel second;
public:
    testGraph1() {
        first = adf::kernel::create(simple1);
        second = adf::kernel::create(simple2);
        adf::connect(first.out[0], second.in[0]);
        adf::source(first) = "src/kernels/kernels.cc";
        adf::source(second) = "src/kernels/kernels.cc";
        adf::runtime<adf::ratio>(first) = 0.1;
        adf::runtime<adf::ratio>(second) = 0.1;

        // Create area group with some valid ranges.
        adf::location<adf::graph>(*this) = adf::area_group({{adf::aie_tile, 0, 0, 1, 7}, {adf::shim_tile,
0, 0, 1, 0}});
    }

};

class testGraph2: public adf::graph {

private:
    adf::kernel first;
    adf::kernel second;
public:
    testGraph2() {
        first = adf::kernel::create(simple1);
        second = adf::kernel::create(simple2);
        adf::connect(first.out[0], second.in[0]);
        adf::source(first) = "src/kernels/kernels.cc";
        adf::source(second) = "src/kernels/kernels.cc";
        adf::runtime<adf::ratio>(first) = 0.1;
        adf::runtime<adf::ratio>(second) = 0.1;

        // Explicitly specify contain_routing, exclusive_routing, exclusive_placement.
        adf::location<adf::graph>(*this) = adf::area_group({{adf::aie_tile, 0, 0, 1, 7}, {adf::shim_tile,
0, 0, 1, 0}}, true, false, true);
    }

};
```

The following table clarifies whether the nodes and nets can access the resources of the area group given various combinations of the flags. The use cases presented in the table as columns are illustrated in the figure that follows. Two important things to consider when applying the rules from the following table.

- Broadcast nets that are driven from one node to several destination nodes, are considered as individual point-to-point nets
- Any FIFOs on a net adhere to the same conditions as the net. For example, if a net is fully contained in an area group (both driver and receiver are contained in the area group) and the contain_routing flag is used, then the following table indicates that the net routing must be fully contained in the area group. Similarly, any FIFOs on that net, must also be placed within the boundary of the area group.

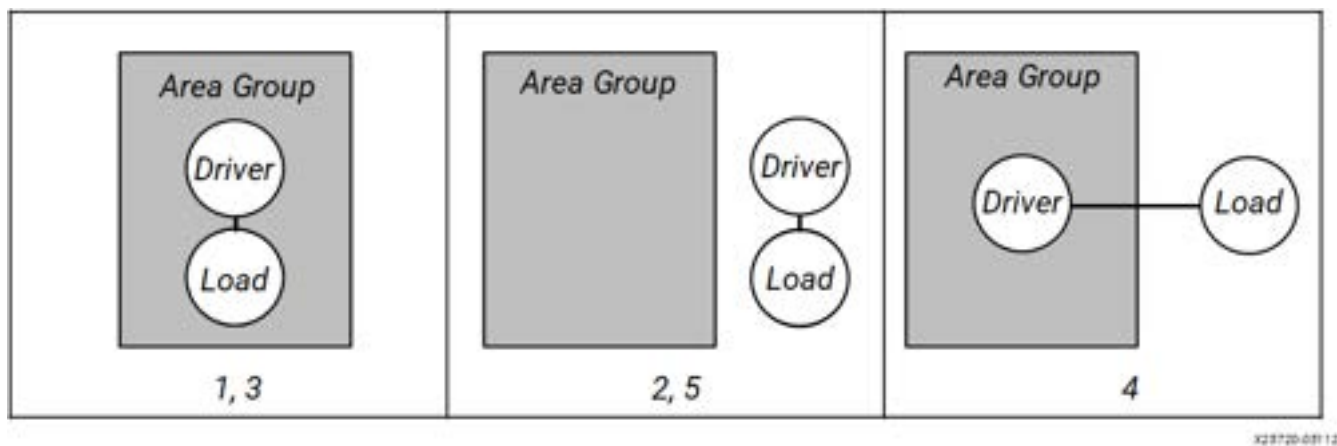
Table: Permission to use Area Group Resources with the Stated Conditions

contain_routing	exclusive_routing	exclusive_placement	Rules
-----------------	-------------------	---------------------	-------

contain_routing	exclusive_routing	exclusive_placement	Placement
False	False	False	Mayt
False	False	True	Mayt Not
False	True	False	Mayt Not
False	True	True	Mayt Not
True	False	False	Mayt
True	False	True	Mayt Not
True	True	False	Mayt Not
True	True	True	Mayt Not

An illustration of the use cases is shown in the following figure.

Figure: Use Cases



Hierarchical Constraints

When creating complex graphs with multiple subgraph classes, or multiple instances of the same subgraph class, the location constraints described previously can also be applied to each kernel instance or kernel port instance individually at the point of subgraph instantiation instead of the definition. In this case, you need to specify the graph qualified name of that kernel instance or kernel port instance in the constraint as shown in the following example. Also, make sure that the kernels or their ports being constrained as described previously are defined to be public members of the subgraph.

```
class ToplevelGraph : public adf::graph {
public:
    indep_nodes_graph1<1,2,3,2> mygraph;
    adf::port<adf::input> datain[2] ;
    adf::port<adf::output> dataout[2] ;

    ToplevelGraph() {
        for (int i = 0; i < 2; i++) {
            adf::connect(datain[i], mygraph.datain[i]);
            adf::connect(mygraph.dataout[i], dataout[i]);


            // hierarchical constraints
            adf::location<adf::stack>(mygraph.kr[i]) = adf::bank(3, 2+i, 2);
            adf::location<adf::buffer>(mygraph.kr[i].out[0]) = adf::location<adf::kernel>(mygraph.kr[i]);
        }
    }
};
```



```

    }
};
};
};

```

 **Note:** You can recirculate the previous design placement in your next compilation. This significantly reduces the mapper run time. When the compiler runs, it generates a placement constraints file in the Work directory. This constraint file can be specified on the command line for the next iteration.

```
v++ -c --mode aie --constraints Work/temp/graph_aie_mapped.aiecst src/graph.cpp
```

To reuse FIFO location constraints, use `aie_routed.aiecst` file.

```
v++ -c --mode aie --constraints Work/temp/graph_aie_routed.aiecst src/graph.cpp
```

Relative Constraints

Kernel, PLIO, GMIO and shared buffer objects can be placed relative to each other. These types of constraints are called relative constraints. The constraints allow for placement of a destination node relative to the source node, and is specified as a row or column offset in 2-D grid. You should specify either the row offset or column offset. If the row offset or column offset is not specified, it is "don't care."

The `adf::relative_offset` API allows you to specify the desired column and row offset. For example:


```

k = adf::kernel::create(matrix_mul);
k1 = adf::kernel::create(shuffle_4x16);
in0=adf::input_plio::create("Datain0", adf::plio_64_bits, "data/matA.txt");

adf::location<adf::kernel>(k1) = adf::location<adf::kernel>(k) + adf::relative_offset({.col_offset = 1,
    .row_offset= 1});
adf::location<adf::PLIO>(in0) = adf::location<adf::kernel>(k) + adf::relative_offset({.col_offset = 0});

adf::shared_buffer<int32> mtxA,mtxB;
adf::location<adf::buffer>(mtxA) = adf::location<adf::buffer>(mtxB) + adf::relative_offset({.col_offset = 1,
    .row_offset=1});
adf::location<adf::kernel>(adf::first) = adf::location<adf::buffer>(mtxA) + adf::relative_offset({.col_offset
    = 1});

```

 **Note:** Shared buffers only support relative column offsets with other types.

Buffer Allocation Control

The AI Engine compiler allocates the desired number of buffers for each memory connection. There are several different cases.

- Lookup tables are always allocated as single buffers because they are expected to be read-only and private to a kernel. No locks are needed to synchronize lookup table accesses because they are expected to be accessed in an exclusive manner.
- Buffer connections are usually assigned double buffers if the producer and consumer kernels are mapped to different processors or if the producer or the consumer is a DMA. This enables the two agents to operate in a pipelined manner using ping-pong synchronization with two locks. The AI Engine compiler generates this synchronization in the respective processor main functions.
- If the producer and consumer kernels are mapped to the same processor, then the buffer connection is given only one buffer and no lock synchronization is needed because the kernels are executed sequentially.
- Runtime parameter connections can be assigned double buffers (default) along with a selector word to choose the next buffer to be accessed.

Runtime parameter connections can also be assigned single buffers. Sometimes, with buffer connections, it is desirable to use only single buffer synchronization instead of double buffers. This is useful when the local data memory is at a premium and the performance penalty of using a single buffer for data transfer is not critical. This can be achieved using the `single_buffer(port<T>&)` constraint.

```

single_buffer(first.in[0]); //For buffer input or RTP input
single_buffer(first.inout[0]); //For RTP output

```

C++ Kernel Class Support

The AI Engine compiler supports C++ kernel classes. The following example shows how to set filter coefficients and the number of samples of a FIR filter class through a constructor. The C++ kernel class allows internal states for each kernel instance to be encapsulated within the

corresponding class object. In the following code, you can see an example of this where the filter coefficients (`coeffs`) are specified through the constructor. This resolves the problem of using file scope variable, global variable, or static function scope variable to store the internal states of a C function kernel. When multiple instances of such a kernel are mapped to the same core, the internal state variables are shared across multiple instances and cause conflicts.

```
//fir.h
#pragma once
#include "adf.h"
#define NUM_COEFFS 12
class FIR
{
private:
    int32 coeffs[NUM_COEFFS];
    int32 tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(const int32(&coefficients)[NUM_COEFFS], uint32 samples);
    void filter(adf::input_buffer<int32> &in, adf::output_buffer<int32> &out);
    static void registerKernelClass()
    {
        REGISTER_FUNCTION(FIR::filter);
    }
};
```

You are required to write the static `void registerKernelClass()` method in the header file. Inside the `registerKernelClass()` method, you need to call the `REGISTER_FUNCTION` macro. This macro is used to register the class run method to be executed on the AI Engine core to perform the kernel functionality. In the preceding example `FIR::filter` is registered using this macro. The kernel class constructor and run method should be implemented inside a separate source file. The implementation of a run method of a kernel class is the same as writing a kernel function described in previous chapters.

```
//fir.cpp
//implementation in this example is not optimized and is for illustration purpose
#include "fir.h"
#include <aie_api/aie.hpp>

FIR::FIR(const int32(&coefficients)[NUM_COEFFS], uint32 samples)
{
    for (int i = 0; i < NUM_COEFFS; i++)
        coeffs[i] = coefficients[i];

    for (int i = 0; i < NUM_COEFFS; i++)
        tapDelayLine[i] = 0;

    numSamples = samples;
}

void FIR::filter(adf::input_buffer<int32> &in, adf::output_buffer<int32> &out){
    auto inIter=aie::begin(in);
    auto outIter=aie::begin(out);
    for (int i = 0; i < numSamples; i++){
        for (int j = NUM_COEFFS-1; j > 0; j--){
            tapDelayLine[j] = tapDelayLine[j - 1];
        }
        tapDelayLine[0] = *inIter++;
        int32 y = 0;
        for (int j = 0; j < NUM_COEFFS; j++){
            y += coeffs[j] * tapDelayLine[j];
        }
        *outIter++=y;
    }
}
```

//graph.h

```

#pragma once
#include "adf.h"
#include "fir.h"
class mygraph : public graph
{
public:
    adf::input_plio in1, in2;
    adf::output_plio out1, out2;
    adf::kernel k1, k2;
    mygraph(){
        in1=adf::input_plio::create("Datain1",adf::plio_32_bits,"data/input1.txt");
        in2=adf::input_plio::create("Datain2",adf::plio_32_bits,"data/input2.txt");
        out1=adf::output_plio::create("Dataout1",adf::plio_32_bits,"data/output1.txt");
        out2=adf::output_plio::create("Dataout2",adf::plio_32_bits,"data/output2.txt");
        k1 = adf::kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80,
-391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 }), 8);
        adf::runtime<adf::ratio>(k1) = 0.9;
        adf::source(k1) = "aie/fir.cpp";
        k2 = adf::kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319,
-78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 }), 8);
        adf::runtime<adf::ratio>(k2) = 0.9;
        adf::source(k2) = "aie/fir.cpp";

        adf::connect(in1.out[0], k1.in[0]);
        adf::connect(in2.out[0], k2.in[0]);
        adf::connect(k1.out[0], out1.in[0]);
        adf::connect(k2.out[0], out2.in[0]);

        adf::dimensions(k1.in[0])={8};
        adf::dimensions(k2.in[0])={8};
        adf::dimensions(k1.out[0])={8};
        adf::dimensions(k2.out[0])={8};
    }
};

```

For a kernel class with a non-default constructor, you can specify the constructor parameter values in the arguments of `kernel::create_object`, when creating a representation of a kernel instance. In the previous example, two FIR filter kernels (k1 and k2) are created using `kernel::create_object<FIR>`. k1 has filter coefficients { 180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 } and k2 has filter coefficients { -21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 }. Both of them consume eight samples for each invocation.

The following code was generated by the AI Engine compiler. The two FIR kernel objects are instantiated with the proper constructor parameters.

```

//Work/aie/<COL_ROW>/src/<COL_ROW>.cc
...
FIR i4({180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504}, 8);
FIR i5({-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066, 18539}, 8);

int main(void) {
    ...
    // Kernel call : i4:filter
    i4.filter(window_buf0_buf0d_i[0],window_buf2_buf2d_o[0]);
    ...
    // Kernel call : i5:filter
    i5.filter(window_buf1_buf1d_i[0],window_buf3_buf3d_o[0]);
    ...
}

```

A kernel class can have a member variable occupying a significant amount of memory space that might not fit into data memory. The location of the kernel class member variable can be controlled. The AI Engine compiler supports array reference member variables that allow the compiler to allocate or constrain the memory space while passing the reference to the object.

```

//fir.h
#pragma once
#include "adf.h"

```

```

#define NUM_COEFFS 12
class FIR
{
private:
    int32 (&coeffs)[NUM_COEFFS];
    int32 tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples);
    void filter(adf::input_buffer<int32> &in, adf::output_buffer<int32> &out);
    static void registerKernelClass()
    {
        REGISTER_FUNCTION(FIR::filter);
        REGISTER_PARAMETER(coeffs);
    }
};

//fir.cpp
#include "fir.h"
FIR::FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples)
    : coeffs(coefficients)
{
    for (int i = 0; i < NUM_COEFFS; i++)
        tapDelayLine[i] = 0;

    numSamples = samples;
}

void FIR::filter(adf::input_buffer<int32> &in, adf::output_buffer<int32> &out)
{
    ...
}

```

The previous example shows a slightly modified version of the FIR kernel class. Here, member variable `coeffs` is a `int32 (&)[NUM_COEFFS]` data type. The constructor initializer `coeffs(coefficients)` initializes `coeffs` to the reference to an array allocated externally to the class object. To let the AI Engine compiler know that the `coeffs` member variable can be relocated in the mapper stage of the compilation, you must use `REGISTER_PARAMETER` to register an array reference member variable inside the `registerKernelClass`. The use of `kernel::create_object` to create a representation of a FIR kernel instance and to specify the initial value of the constructor parameters is the same as in the previous example. See the following code.

```

//graph.h
...
class mygraph : public adf::graph
{
...
    mygraph()
    {
        k1 = adf::kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80, -391, -720, -834, -478, 505,
2063, 3896, 5535, 6504 }), 8);
        ...
        k2 = adf::kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319, -78, -511, 977, -610, -844,
2574, -2754, -1066, 18539 }), 8);
        ...
    }
};

```

The following code was generated by the AI Engine compiler. The memory spaces for `int32 i4_coeffs[12]` and `int32 i5_coeffs[15]` are outside the kernel object instances and are passed into the FIR objects by reference.

```

//Work/aie/<COL_ROW>/src/<COL_ROW>.cc
int32 i4_coeffs[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504};
FIR i4(i4_coeffs, 8);
int32 i5_coeffs[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066, 18539};

```

```

FIR i5(i5_coeffs, 8);

int main(void) {
    ...
    // Kernel call : i4:filter
    i4.filter(window_buf0_buf0d_i[0],window_buf2_buf2d_o[0]);
    ...
    // Kernel call : i5:filter
    i5.filter(window_buf1_buf1d_i[0],window_buf3_buf3d_o[0]);
    ...
}

```

Because the memory space for an array reference member variable is allocated by the AI Engine compiler, the location constraint can be applied to constrain the memory location of these arrays, as shown in the following example code. The REGISTER_PARAMETER macro allows kernel::create_object to create a parameter handle for an array reference member variable, like k1.param[0] and k2.param[0], and the location<parameter> constraint can be applied.

```

//graph.h
...
class mygraph : public adf::graph
{
    ...
    mygraph()
    {
        k1 = adf::kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80, -391, -720, -834, -478, 505,
2063, 3896, 5535, 6504 })), 8);
        ...
        k2 = adf::kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319, -78, -511, 977, -610, -844,
2574, -2754, -1066, 18539 })), 8);
        ...

        adf::location<adf::parameter>(k1.param[0]) = adf::address(...);
        adf::location<adf::parameter>(k2.param[0]) = adf::bank(...);
    }
};

```

The C++ kernel class header files and the C++ kernel function template (see [C++ Template Support](#)) should not contain single-core specific intrinsic APIs and pragmas. This is the same programming guideline as writing regular C function kernels. This is because these header files are included in the graph header file and can be cross-compiled as part of the PS program. The Arm® cross-compiler cannot understand single-core intrinsic APIs or pragmas. Single-core specific programming content must be kept inside the source files.

C++ Template Support

A template is a powerful tool in C++. By passing the data type as a parameter, you eliminate the need to rewrite code to support different data types. Templates are expanded at compile time, like macros. The difference is that the compiler performs type checking before template expansion. The source code contains template functions and class definitions, but the compiled code can contain multiple copies of same function or class. Type parameters, non-type parameters, default arguments, scalar parameters, and template parameters can be passed to a template, where the compiler instantiates the function or class accordingly.

- Support for general C++ template features.
- Supported data types (T) and connection types between kernels:
 - int8
 - uint8
 - int16
 - uint16
 - cint16
 - int32
 - uint32
 - cint32
 - int64
 - uint64
 - float
 - float16
 - bfloat8
 - float8
 - mx9
- The compiler does not support pre-compiled headers for template kernels.

!! Important: acc48 and cacc48 data types are not supported in template stream connections.

Function Templates

Function template source code defines a generic function that can be used for different data types. Example function template:

```
// add.h
```

```
template<typename ELEMENT_TYPE, int FACTOR, size_t NUM_SAMPLES> void add(adf::input_buffer<ELEMENT_TYPE> &in,
adf::output_buffer<ELEMENT_TYPE> &out);
```

```
// add.cpp
```

```
#include <aie_api/aie.hpp>
template<typename ELEMENT_TYPE, int FACTOR, size_t NUM_SAMPLES> void add(adf::input_buffer<ELEMENT_TYPE> &in,
adf::output_buffer<ELEMENT_TYPE> &out){
    auto inIter=aie::begin(in);
    auto outIter=aie::begin(out);
    for (int i=0; i<NUM_SAMPLES; i++){
        ELEMENT_TYPE value = *inIter++;
        value += FACTOR;
        *outIter++=value;
    }
}
```

```
// graph.h
```

```
mygraph()
{
    k[0] = adf::kernel::create(add<int32, 6, 8>);
    k[1] = adf::kernel::create(add<int16, 3, 8>);
    for (int i=0; i<NUM_KERNELS; i++)
    {
        adf::runtime<adf::ratio>(k[i]) = 0.3;
        adf::source(k[i]) = "src/add.cpp";
    }

    adf::connect(in[0], k[0].in[0]);
    adf::connect(k[0].out[0], out[0]);

    adf::connect(in[1], k[1].in[0]);
    adf::connect(k[1].out[0], out[1]);

    adf::dimensions(k[0].in[0])={8};
```

```

    adf::dimensions(k[0].out[0])={8};
    adf::dimensions(k[1].in[0])={8};
    adf::dimensions(k[1].out[0])={8};
}

```

where:

- add.h defines a template add() function.
- add.cpp defines the code for the template add() function.
- graph.h uses the template add() function within mygraph class.

Class Templates

Like function templates, class templates are useful when a class defines an object that is independent of a specific data type. Example class template:

```

// fir.h
...
template<size_t NUM_COEFFS, typename ELEMENT_TYPE> class FIR
{
private:
    ELEMENT_TYPE (&coeffs)[NUM_COEFFS];
    ELEMENT_TYPE tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(ELEMENT_TYPE(&coefficients)[NUM_COEFFS], uint32 samples);

    void filter(input_buffer<ELEMENT_TYPE> &in, output_buffer<ELEMENT_TYPE> &out);

    //user needs to write this function to register necessary info
    static void registerKernelClass()
    {
        REGISTER_FUNCTION(FIR::filter);
        REGISTER_PARAMETER(coeffs);
    }
}

// fir.cpp
...
template<size_t NUM_COEFFS, typename ELEMENT_TYPE> FIR<NUM_COEFFS,
ELEMENT_TYPE>::FIR(ELEMENT_TYPE(&coefficients)[NUM_COEFFS], uint32 samples):coeffs(coefficients)
{
    ...
}

template<size_t NUM_COEFFS, typename ELEMENT_TYPE> void
FIR<NUM_COEFFS,ELEMENT_TYPE>::filter(adf::input_buffer<ELEMENT_TYPE> &in, adf::output_buffer<ELEMENT_TYPE>
&out)
{
    ...
}

// graph.h
...
mygraph()
{
    k1 = adf::kernel::create_object<FIR<12, int32>>(std::vector<int>({ 180, 89, -80, -391, -720, -834, -478,
505, 2063, 3896, 5535, 6504 })), 8);
    adf::runtime<adf::ratio>(k1) = 0.1;
    adf::source(k1) = "src/fir.cpp";
    adf::headers(k1) = { "src/fir.h" };

    k2 = adf::kernel::create_object<FIR<15, int32>>(std::vector<int>({ -21, -249, 319, -78, -511, 977, -610,

```

```
-844, 2574, -2754, -1066, 18539, 0, 0, 0 }}, 8);  
adf::runtime<ratio>(k2) = 0.1;  
adf::source(k2) = "src/fir.cpp";  
adf::headers(k2) = { "src/fir.h" };  
...  
}
```

where:

- fir.h defines a class template where class FIR is declared.
- fir.cpp contains class FIR implementation and the class FIR member function filter implementation.
- graph.h demonstrates the template class FIR instantiation within the mygraph class.

Multicast Support

Various multicast scenarios are supported in the graph, such as from a buffer to multiple buffers, from stream to multiple streams, from input_plio to multiple buffers, etc. This section lists the supported types of multicast from a single source to multiple destinations. For additional details on input_plio/output_plio, and input_gmio/output_gmio, see [Graph Programming Model](#).

Table: Multicast Support Scenarios

Scenario #	Source	Destination 1	Destination 2	Support
1	AI Engine Buffer	AI Engine Buffer	AI Engine Buffer	Supported
2	AI Engine Buffer	AI Engine Buffer	AI Engine Stream	Supported
3	AI Engine Buffer	AI Engine Buffer	output_plio/output_gmio	Supported
4	AI Engine Buffer	AI Engine Stream	AI Engine Stream	Supported
5	AI Engine Buffer	AI Engine Stream	output_plio/output_gmio	Supported
6	AI Engine Buffer	output_plio/output_gmio	output_plio/output_gmio	Supported
7	AI Engine Stream	AI Engine Buffer	AI Engine Buffer	Supported
8	AI Engine Stream	AI Engine Buffer	AI Engine Stream	Supported
9	AI Engine Stream	AI Engine Buffer	output_plio/output_gmio	Supported
10	AI Engine Stream	AI Engine Stream	AI Engine Stream	Supported
11	AI Engine Stream	AI Engine Stream	output_plio/output_gmio	Supported
12	AI Engine Stream	output_plio/output_gmio	output_plio/output_gmio	Supported
13	input_plio/input_gmio	AI Engine Buffer	AI Engine Buffer	Supported
14	input_plio/input_gmio	AI Engine Buffer	AI Engine Stream	Not Supported
15	input_plio/input_gmio	AI Engine Buffer	output_plio/output_gmio	Not Supported
16	input_plio/input_gmio	AI Engine Stream	AI Engine Stream	Supported
17	input_plio/input_gmio	AI Engine Stream	output_plio/output_gmio	Not Supported
18	input_plio/input_gmio	output_plio/output_gmio	output_plio/output_gmio	Not Supported

Note:

- All source and destination buffers in the multicast connections are required to have the same size to stay in a single rate environment.
- If all sources and destinations do not have the same size, the compiler automatically switches to multirate processing. The compiler determines the number of times the kernels need to be executed per iteration.
- Buffer multicast is realized by the tool by adding DMA to source and destination buffers.
- Each connection between the source and destination is blocking. Any destination blocks the multicast if it is not ready to accept data.
- RTP and packet switching are not covered in this section.
- If the multicast type is supported, the destination number is not limited if it can fit into the hardware.

When multiple streams are connected to the same source, the data is sent to all the destination ports at the same time and is only sent when all destinations are ready to receive data. This might cause stream stall or design hang if the FIFO depth of the stream connections are not deep enough.

The following multicast example shows scenario number 10 from above table. Source and both destinations are stream.

In this graph.h code snippet, two sub-graphs named `_graph0` and `_graph1` are defined within the top graph named `top_graph`. This ensures that sending data to all destination ports at the same time.

```
class _graph0: public adf::graph {
private:
    adf::kernel kr;
public:
    adf::port<input> instream;
    adf::port<output> ostream;
    _graph0() {
        kr = adf::kernel::create(compute0);
        adf::runtime<ratio>(kr) = 0.9;
        adf::source(kr) = "compute0.cc";
        adf::connect<adf::stream> n0(instream, kr.in[0]);
        adf::connect<adf::stream> n1(kr.out[0], ostream);
    }
};

class _graph1: public adf::graph {
private:
    adf::kernel kr;
public:
    adf::port<input> instream;
    adf::port<output> ostream;
    _graph1() {
        kr = adf::kernel::create(compute1);
        adf::runtime<ratio>(kr) = 0.9;
        adf::source(kr) = "compute1.cc";
        adf::connect<adf::stream> n0(instream, kr.in[0]);
        adf::connect<adf::stream> n1(kr.out[0], ostream);
    }
};

class top_graph: public adf::graph {
private:

public:
    _graph0 g0;
    _graph1 g1;
    adf::input_plio instream;
    adf::output_plio ostream0;
    adf::output_plio ostream1;
    top_graph()
    {
        instream = adf::input_plio::create("aie_broadcast_0_S_AXIS",
            adf::plio_32_bits,
            "data/input.txt");
        ostream0 = adf::output_plio::create("aie_graph0_ostream",
            adf::plio_32_bits,
            "data/output0.txt");
        ostream1 = adf::output_plio::create("aie_graph1_ostream",
            adf::plio_32_bits,
            "data/output1.txt");

        adf::connect<adf::stream> n0(instream.out[0], g0.instream);
        adf::connect<adf::stream> n1(instream.out[0], g1.instream);
        adf::connect<adf::stream> n2(g0.ostream, ostream0.in[0]);
        adf::connect<adf::stream> n3(g1.ostream, ostream1.in[0]);
    }
};
```

In this graph.cpp code snippet, the graph calls are invoked from the top graph so all sub-graphs are receiving the same data at the same time.

```
top_graph top_g;

#ifdef (__AIESIM__) || defined(__X86SIM__)
int main () {
    top_g.init();
    top_g.run(3);
    top_g.wait();
    top_g.end();
    return 0;
}
#endif
```

Logical I/O Ports

When designing a subgraph, it might not be feasible to know beforehand how an I/O port will be connected in the final system. It is probable that a subgraph output will be linked to an output_plio or output_gmio port. To overcome this issue, the subgraph designer can define logical I/O ports like port<input>, port<inout>, and port<output> graph objects. This ensures that you can specify the inputs and outputs of the subgraph while enabling the end-user to determine how those ports will be physically connected in the system. These objects can establish links between kernels within a graph and across levels of hierarchy in your specifications that include platforms, graphs, and subgraphs.

Related Information

[port<T>](#)

[Port Combinations](#)

Conditional Ports

Certain graphs design might require the instantiation of a port based on a template parameter. For example if you want to use the same templated graph with or without a port that will be connected to a cascade in or a cascade out kernel port, then you will need to use this conditional port feature.

The syntax to conditionally instantiate a port or a sub-graph is as follows:

```
typename std::conditional<SEL,T1,T2>::type Object
```

Where,

- T1 is an input_port, output_port, kernel, or user-defined graph.
- T2 is a dummy type as std::tuple<> or int.
- Object is the name of the object that will be instantiated.

The syntax to conditionally instantiate an array of ports or sub-graphs is as follows:

```
typename std::array<T3,N>::type Object
```

Where,

- T3 is an input_port, output_port, kernel, or user-defined graph.
- N is the number of items in the array.
- Object is the name of the object that will be instantiated.

Here is an example of conditional port instantiation:

```
#include "adf.h"

using namespace adf;

void k0(input_stream<int32> *, output_stream<int32> *);
void k0_cascin(input_stream<int32> *, output_stream<int32> *, input_cascade<acc48>*);
void k0_cascout(input_stream<int32> *, output_stream<int32> *, output_cascade<acc48>*);
void k0_cascin_cascout(input_stream<int32> *, output_stream<int32> *, input_cascade<acc48>*,
```

```

output_cascade<acc48>*);

template<bool HAS_CASCADE_IN, bool HAS_CASCADE_OUT>
struct SubGraph: public graph
{
    input_port strmIn;
    input_port strmOut;
    kernel k1;
    typename std::conditional<HAS_CASCADE_IN, input_port, int>::type cascIn;
    typename std::conditional<HAS_CASCADE_OUT, output_port, int>::type cascOut;

    SubGraph()
    {
        if constexpr (HAS_CASCADE_IN && HAS_CASCADE_OUT)
        {
            k1 = adf::kernel::create(k0_cascIn_cascOut);
        }
        else if constexpr (HAS_CASCADE_IN)
        {
            k1 = adf::kernel::create(k0_cascIn);
        }
        else if constexpr (HAS_CASCADE_OUT)
        {
            k1 = adf::kernel::create(k0_cascOut);
        }
        else
        {
            k1 = adf::kernel::create(k0);
        }
        adf::connect(strmIn, k1.in[0]);
        adf::connect(k1.out[0], strmOut);
        if constexpr (HAS_CASCADE_IN)
        {
            adf::connect(cascIn, k1.in[1]);
        }
        if constexpr (HAS_CASCADE_OUT)
        {
            adf::connect(k1.out[1], cascOut);
        }
        adf::source(k1) = "kernels.cc";
        adf::runtime<ratio>(k1) = 0.6;
    }
};

```

In this example, the four functions have different ports. One of these functions is instantiated in the graph depending on the template parameters. All functions have one input stream and one output stream, however, the cascade input and output ports are optional. The two template parameters are used in the `std::conditional` to create, or not, the cascade ports at the sub-graph level. They are also used to specify which function is instantiated in the sub-graph, and to connect the kernel ports to the right sub-graph port.

Following is another example where a complete sub-graph is instantiated and is not dependent on a template parameter:

```

#include "adf.h"

template<int ID>
void f0(input_stream<int32> *, output_stream<int32> *);

struct Sub0: public graph
{
    input_port _in0;
    input_port _out0;
    adf::kernel _k0;
    Sub0()
    {
        _k0 = adf::kernel::create(f0<0>);
    }
};

```

```

    adf::connect(_in0, _k0.in[0]);
    adf::connect(_k0.out[0], _out0);
    adf::runtime<adf::ratio>(_k0) = 0.9;
    adf::source(_k0) = "k0.cpp";
}
};

```

```

struct Sub1: public graph
{
    adf::input_port _in0;
    adf::input_port _out0;
    adf::kernel _k0;
    Sub1()
    {
        _k0 = adf::kernel::create(f0<1>);
        adf::connect(_in0, _k0.in[0]);
        adf::connect(_k0.out[0], _out0);
        adf::runtime<adf::ratio>(_k0) = 0.8;
        adf::source(_k0) = "k0.cpp";
    }
};

```

```

template<int ID>
struct MyGraph: public graph
{
    adf::input_plio _plioI;
    adf::output_plio _plio0;

    constexpr static bool hasSub0() {return ID & 0x1;}
    constexpr static bool hasSub1() {return ID & 0x2;}

    typename std::conditional<hasSub0(), Sub0, int >::type _sub0;
    typename std::conditional<hasSub1(), Sub1, int >::type _sub1;

    MyGraph()
    {
        _plioI = adf::input_plio::create("plio_I"+std::to_string(ID), adf::plio_32_bits,
"input"+std::to_string(ID)+".txt");
        _plio0 = adf::output_plio::create("plio_0"+std::to_string(ID), adf::plio_64_bits,
"output"+std::to_string(ID)+".txt");
        if constexpr (hasSub0() && hasSub1())
        {
            adf::connect(_plioI.out[0], _sub0._in0);
            adf::connect(_sub0._out0, _sub1._in0);
            adf::connect(_sub1._out0, _plio0.in[0]);
        }
        else if constexpr (hasSub0())
        {
            adf::connect(_plioI.out[0], _sub0._in0);
            adf::connect(_sub0._out0, _plio0.in[0]);
        }
        else if constexpr (hasSub1())
        {
            adf::connect(_plioI.out[0], _sub1._in0);
            adf::connect(_sub1._out0, _plio0.in[0]);
        }
    }
};

```

Depending on the template parameter ID, the graph MyGraph will contain either one or both of the sub-graphs Sub0 and Sub1. If ID=0, no sub-graph is instantiated, and the compiler will error out. The two lines containing the `std::conditional` are used to conditionally instantiate the two sub-graphs. The connection of the kernels to the I/Os is also driven by the template parameter ID.

Array of Graph Objects

A graph can contain multiple ports of the same type depending on some template parameter. This case is handled efficiently by the standard C vector notation, for example, `input_port in[5];`.

If you want this set of ports to be conditionally instantiated, the compiler will reject this notation. In that case, you must use C++ `std::array` notation:

```
std::array<T,N> Object;
```

Where,

- T is the type of the object `input_port/output_port`, `input_plio/output_plio`, `input_gmio/output_gmio`, `sub-graph`, `kernel`, and so on.
- N is the number of elements of the array.
- Object is the name of the array to instantiate.

The following example shows how to declare various objects in arrays:

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

void f0(input_stream<int32> *, output_stream<int32> *);

constexpr int getNum(int id) {return id == 1? 2: 3;}

template<int ID>
struct Sub0: public graph
{
    std::array<adf::input_port,getNum(ID)> _ins;
    std::array<adf::output_port,getNum(ID)> _outs;
    std::array<adf::kernel,getNum(ID)> _ks;
    Sub0()
    {
        for (auto &k: _ks)
        {
            k = adf::kernel::create(f0);
            adf::runtime<adf::ratio>(k) = 0.9;
            adf::source(k) = "k0.cpp";
        }
        for (int ind = 0; ind < _ins.size(); ++ind)
        {
            adf::connect(_ins[ind], _ks[ind].in[0]);
            adf::connect(_ks[ind].out[0], _outs[ind]);
        }
    }
};

template<int ID>
struct MyGraph: public graph
{
    std::array<input_plio , getNum(ID)> _plioIs;
    std::array<output_plio, getNum(ID)> _plioOs;
    Sub0<ID> _sub;

    MyGraph()
    {
        for (int ind = 0; ind < getNum(ID); ++ind)
        {
            char iName[40];
            char oName[40];
            sprintf(iName, "data/i_%d_%d.txt", ID, ind);
            sprintf(oName, "data/o_%d_%d.txt", ID, ind);
        }
    }
};
```

```

    _plioIs[ind] = adf::input_plio::create(adf::plio_32_bits, iName);
    _plioOs[ind] = adf::output_plio::create(adf::plio_32_bits, oName);

    adf::connect(_plioIs[ind].out[0], _sub._ins[ind]);
    adf::connect(_sub._outs[ind], _plioOs[ind].in[0]);
}
}
};


```

AI Engine/Programmable Logic Integration

When you are ready to consider interfacing to the programmable logic (PL), you need to make a decision on the platform you want to interface with. A platform is a fully contained image that defines both the hardware (XSA) as well as the software (bare metal, Linux, or both). The XSA contains the hardware description of the platform, which is defined in the AMD Vivado™ Design Suite, and the software is defined with the use of a bare-metal setup, or a Linux image defined through PetaLinux. Depending on the needs of your application you might decide to use an example reference platform provided by AMD, or a custom platform created by your organization.

AMD recommends interfacing to the PLIO port attributes which represent external stream connections that cross the AI Engine-PL boundary. PLIO represents an ADF graph interface to the PL. This PL could be, for example, a PL kernel, a platform IP representing a signal source or sink, or it could be a data mover to interface the ADF graph to memory.

Alternatively interface connections can also be GMIO port attributes which represent external memory-mapped connections to or from the global memory. Further details on these attributes can be found in [Graph Programming Model](#).

 **Note:** The AI Engine Engine Graph with PL kernels (HLS or RTL kernels) can be co-simulated using the Vitis Hardware Emulation flow.

Design Flow Using RTL Programmable Logic

RTL blocks are not supported inside the ADF graph. Communication between the RTL blocks and the ADF graph requires PLIO interfaces. In the following example, `interpolator` and `classify` are AI Engine kernels. The `interpolator` AI Engine kernel streams data to a PL RTL block, which, in turn, streams data back to the AI Engine `classify` kernel.

```

class clipped : public graph {
private:
    kernel interpolator;
    kernel classify;

public:
    input_plio in;
    output_plio out;
    output_plio clip_in;
    input_plio clip_out;

    clipped() {
        in = input_plio::create("DataIn1", plio_32_bits,"data/input.txt");
        out = output_plio::create("DataOut1", plio_32_bits,"data/output.txt");
        clip_out = input_plio::create("clip_out",plio_32_bits,"data/input1.txt");
        clip_in = output_plio::create("clip_in", plio_32_bits,"data/output1.txt");

        interpolator = kernel::create(fir_27t_sym_hb_2i);
        classify      = kernel::create(classifier);

        connect(in.out[0], interpolator.in[0]);
        connect(interpolator.out[0], clip_in.in[0]);
        connect(clip_out.out[0], classify.in[0]);
        connect(classify.out[0], out.in[0]);

        std::vector<std::string> myheaders;
        myheaders.push_back("include.h");

        adf::headers(interpolator) = myheaders;
        adf::headers(classify) = myheaders;

        source(interpolator) = "kernels/interpolators/hb27_2i.cc";
        source(classify)     = "kernels/classifiers/classify.cc";
    }
};

```

```

        runtime<ratio>(interpolator) = 0.8;
        runtime<ratio>(classify) = 0.8;
    };
};

```

`clip_in` and `clip_out` are ports to and from the `polar_clip` PL RTL kernel which is connected to the AI Engine kernels in the graph. For example, the `clip_in` port is the output of the `interpolator` AI Engine kernel that is connected to the input of the `polar_clip` RTL kernel. The `clip_out` port is the input of the `classify` AI Engine kernel and the output of the `polar_clip` RTL kernel.

RTL Blocks and AI Engine

The following example shows application code.

```


#include "graph.h"

clipped clipgraph;

#if defined(__AIESIM__) || defined(__X86SIM__)
int main(int argc, char ** argv) {
    clipgraph.init();
    clipgraph.run();
    clipgraph.end();
    return 0;
}
#endif

```

To make the `aiesimulator` work, you must create input test bench files related to the RTL kernel. `data/output_interp.txt` is the test bench input to the RTL kernel. The `aiesimulator` generates the output file from the `interpolator` AI Engine kernel. The `data/input_classify.txt` file contains data from the `polar_clip` kernel which is input to the AI Engine `classify` kernel.

 **Note:** PLIO can have an optional attribute, PL clock frequency, which is 100 for the `polar_clip`.

RTL Blocks in Hardware Emulation and Hardware Flows

RTL kernels are fully supported in hardware emulation and hardware flows. You need to add the RTL kernel as an `nk` option and link the interfaces with the `sc` option, as shown in the following code. If necessary, adjust any clock frequency using `freqHz`. The following is an example of a Vitis configuration file.

```

[connectivity]
nk=mm2s:1:mm2s
nk=s2mm:1:s2mm
nk=polar_clip:1:polar_clip
sc=mm2s.s:ai_engine_0.DataIn1
sc=ai_engine_0.clip_in:polar_clip.in_sample
sc=polar_clip.out_sample:ai_engine_0.clip_out
sc=ai_engine_0.DataOut1:s2mm.s
[clock]
freqHz=100000000:polar_clip.ap_clk

```

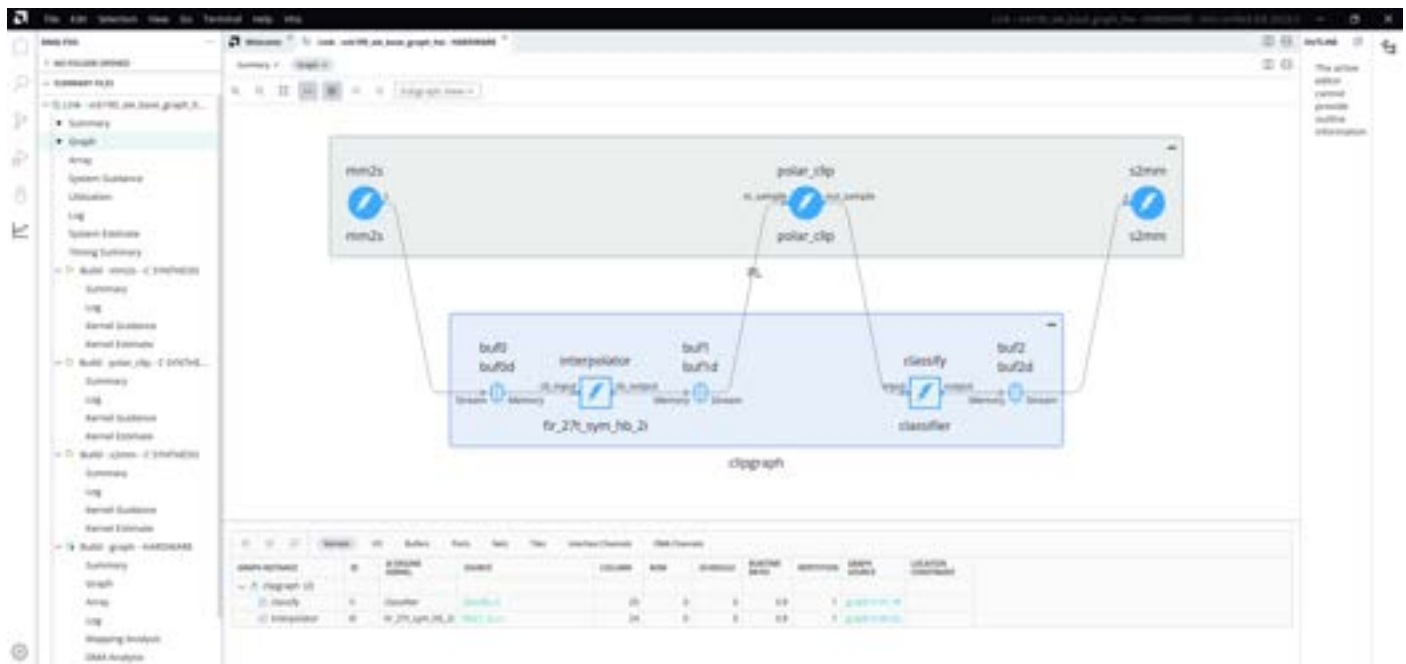
For more information on RTL kernels and the Vitis flow, see [RTL Kernel Development](#) in the *Embedded Design Development Using Vitis (UG1701)*.

PL Kernels Inside AI Engine Graph

After system linking with AI Engine graph and PL Kernels, the system diagram of AI Engine graph and PL Kernels connections can be shown inside AI Engine graph view. Open the linker summary in the Vitis IDE with the following command:

```
vitis -a <USER_NAME>.xsa.link_summary
```

Figure: PL Kernels Inside AI Engine Graph

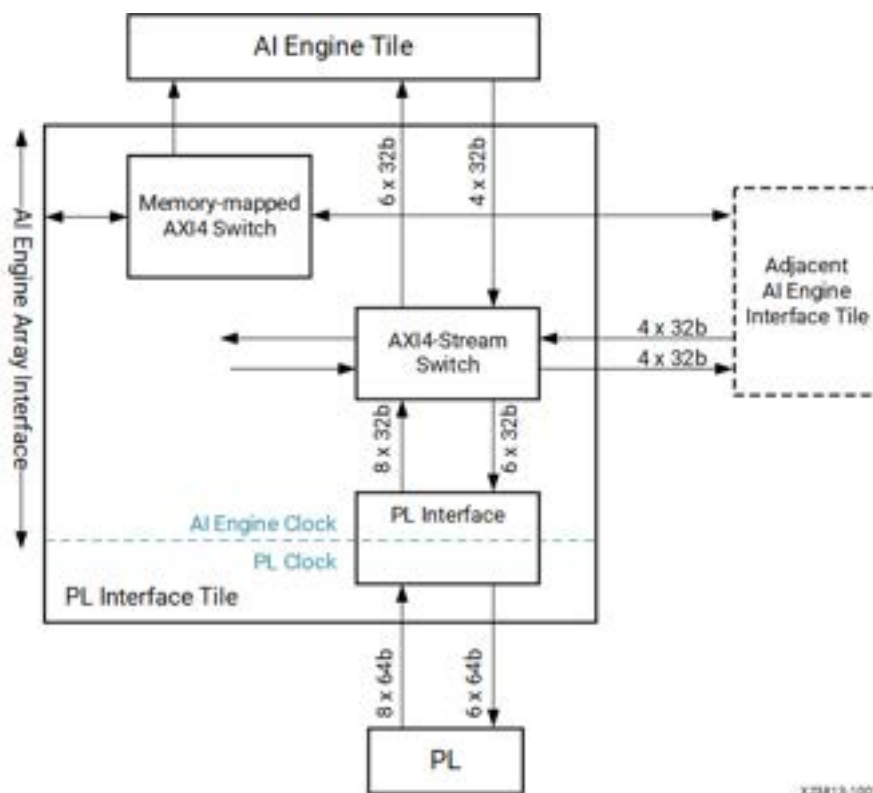


Design Considerations for Graphs Interacting with Programmable Logic

The AI Engine array is made up of AI Engine tiles and AI Engine array interface tiles on the last row of the array. The types of interface tiles include AI Engine-PL and AI Engine-NoC.

Knowledge of the PL interface tile, which interfaces and adapts the signals between the AI Engines and the PL region, is essential to take full advantage of the bandwidth between AI Engines and the PL. The following figure shows an expanded view of a single PL interface tile.

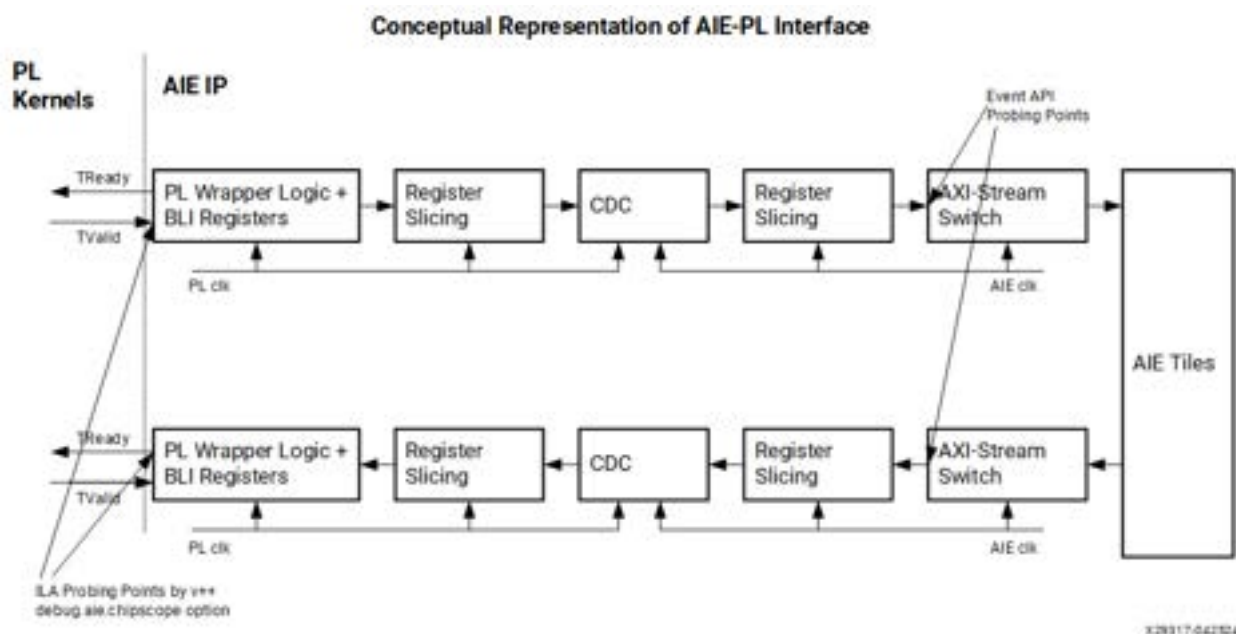
Figure: AI Engine-PL Interface Tile



Note: Notice the interface tile supports two different clock domains (AI Engine clock and PL clock), and a predefined number of streaming channels available to connect from the AI Engine tile to a specific PL interface tile.

Following is a conceptual representation of the AI Engine-PL interface interacting with PL and AI Engine tiles:

Figure: Conceptual Representation of AIE-PL Interface



Notice the CDC path between PL and AI Engine. The latency of the path can vary when PL frequency or phase changes.

Generally, the higher the frequency of the PL, the lower the latency in absolute time. And the higher the frequency of the PL, the higher the throughput or sample rate of the PL kernels. It is important to plan the PL clocks for low latency applications and high speed designs, based on the AI Engine-to-PL rate matching or any other requirements.

When using event APIs to do profiling, the probing points are inside the AXI4-Stream switch box of the AI Engine-PL interface. However, if using `--debug.aie.chipscope` option of `v++`, the ILA probing points will be on the PL wrapper logic. Thus, there will be multiple cycle differences between the two methods when measuring the latency of the AI Engine graph.

Also, the path inside the AI Engine-PL interface including the AXI4-Stream switch box has the capability of buffering. The `tready` signal from AI Engine will be asserted after the device is booted, that is, even before the AI Engine graph is run by host code. So, if PL kernel starts transferring data to AI Engine, it fills all the buffers inside the AI Engine-PL interface, until back pressure occurs.

PL Interface Tile Capabilities

The AI Engine clock can run at up to 1 GHz for -1L speed grade devices, or higher, for -2 and -3 speed grade devices. The default width of a stream channel is 32 bits. Because this frequency is higher than the PL clock frequency, it is always necessary to perform a clock domain crossing to the PL region, for example, to either one-half or a quarter of the AI Engine clock frequency.

Recommended: Though not required, AMD recommends running the PL kernel with a frequency where the AI Engine frequency is an integer multiple of the PL kernel frequency.

For C++ HLS PL kernels, choose an appropriate target frequency depending on the complexity of the algorithm implemented. The `--hls.clock` option can be used in the Vitis compiler when compiling HLS C/C++ into Xilinx object (XO) files.

AI Engine-to-PL Rate Matching

The AI Engine runs at 1 GHz (or more, depending on the speed grade) and each interface channel can read or write with a 64-bit data width per cycle. In contrast, a PL kernel can run at 500 MHz (half the frequency of the AI Engine), while consuming a larger bit width. Rate matching is concerned with balancing the throughput from the producer to the consumer, and is used to ensure that neither of the processes creates a bottleneck with respect to the total performance. The following equation shows the rate matching for each channel:

Frequency AI Engine × Data AI Engine per cycle = Frequency PL × Data PL per cycle

The following table shows a PL rate matching example for a 32-bit channel written to each cycle by the AI Engine at 1 GHz for -1L speed grade devices. As shown, the PL IP has to consume two times the data at half the frequency or four times the data at one quarter of the frequency.

Table: Frequency Response of AI Engine Compared to PL Region

AI Engine		PL	
Frequency	Data per Cycle	Frequency	Data per Cycle
1 GHz	32-bit	500 MHz	64-bit
		250 MHz	128-bit

Because the need to match frequency and adjust data-path width is well understood by the Vitis compiler (`v++`), the tool automatically extracts the port width from the PL kernel, the frequency from the clock specification, and introduces an upsizer/downsizer to temporarily store the data

exchanged between the AI Engine and the PL regions to manage the rate match.

To avoid deadlocks, it is important to ensure that if multiple channels are read or written between the AI Engine and the PL, the data rate per cycle is concurrently achieved on all channels. For example, if one channel requires 32 bits, and the second 64 bits, the AI Engine code must ensure that both channels are written adequately to avoid back pressure or starvation on the channel. Additionally, to avoid deadlock, writing/reading from the AI Engine and reading/writing in the PL code must follow the same chronological order.

The number of interfaces used in the graph function definition for the PL defines the number of AXI4-Stream interfaces. Each argument results in the creation of a separate stream.

AI Engine to PL Interface Performance

Versal AI Core Series devices include an AI Engine array with the following column categories.

PL column

provides PL stream access. Each column supports eight 64-bit slave channels for streaming data into the AI Engine and six 64-bit master channels for streaming data to the PL.

NoC column

provides connectivity between the AI Engine array and the NoC. These interfaces can also connect to the PL.


To instruct the AI Engine compiler to select higher frequency interfaces, use the `--pl-freq=<number>` to specify the clock frequency (in MHz) for the PL kernels. The default value is one quarter of the AI Engine frequency and the maximum supported value is a half of the AI Engine frequency, the values depending on the speed grade. Following are examples:

- Option to enable an AI Engine to PL frequency of 300 MHz for all AI Engine to PL interfaces:

```
--pl-freq=300
```

- To set a different frequency for a specific PLIO interface use the following code to set it in the ADF graph.

```
adf::PLIO *<input>= new adf::PLIO(<logical_name>, <plio_width>, <file>, <FreqMHz>);
```

 **Note:** The following information applies to the AI Engine device architecture documented in *Versal Adaptive SoC AI Engine Architecture Manual (AM009)*.

The AI Engine to PL AXI4-Stream channels use boundary logic interface (BLI) connections that include optional BLI registers with the exception of the slave channels 3 and 7. The two slave channels channel 3 and channel 7 are slower interfaces. The performance of the data transfer between the AI Engine and PL depends on whether the optional BLI registers are enabled or not.

For less timing-critical designs, all eight channels can be used without using the BLI registers. PL timing can still be met in this case. However, for higher frequency designs, only the six fast channels (0,1,2,4,5,6) can be used and the timing paths from the PL must be registered, using the BLI registers.

To control the use of BLI registers across the AI Engine to PL channels, use the `--pl-register-threshold=<number>` compiler option, specified in MHz. The default value is 1/8 of the AI Engine frequency based on speed grade. Following is an example:

- `--pl-register-threshold=125`

The compiler will map any PLIO interface with an AI Engine to PL frequency higher than this setting (125 MHz in this case) to high-speed channels with the BLI registers enabled. If the PLIO interface frequency is not higher than the `pl-register-threshold` value then any of the AI Engine to PL channels will be used.

In summary, if `pl-freq < pl-register-threshold` all eight channels can be used unregistered. If `pl-freq > pl-register-threshold` only the six fast channels can be used, with registering. `pl-register-threshold` is a way to control the threshold frequency beyond which only fast channels can be used (with registering).

AI Engine to PL Interface AXI Protocol

AI Engine to PL AXI4-Stream interfaces support a subset of the AXI4-Stream protocol (per the [AMBA 4 AXI4-Stream Protocol Specification](#)).

Internally, AI Engine to PL AXI4-Stream interfaces are 64-bit channels physically. And 128-bit occupies two adjacent physical channels. This imposes additional requirements on sending data timely between AI Engine and PL via AXI4-Stream interfaces. This section focuses on TLAST and TKEEP requirements of the interfaces to send data without stall.


TLAST is required for a 64-bit stream between the AI Engine and PL if single 32-bit words are sent. AI Engine to PL 32-bit stream interfaces are automatically internally up-sized to 64-bit interfaces by the AI Engine compiler. When sending 32-bit stream data (to or from the PL from the AI Engine), single 32-bit words without TLAST are held in the interface until a second 32-bit word arrives to complete a 64-bit up-sizing. The solution is to assert TLAST for the single 32-bit data. The data will be pushed into AI Engine without stall.

When using 64-bit and 128-bit interfaces, it is valid to send data without TLAST. Then, even number of data are sent without stall. TLAST can be used depending on the need. When TLAST is asserted, TKEEP can be used together with TLAST to send arbitrate number of 32-bit data for 64-bit and 128-bit interfaces. TKEEP must be set correctly, either -1 (all bits are 1) or partial 32-bit words are enabled, for example 0x0F. The lower parts of the data should be asserted if only partial data is valid.

The following table summarizes TLAST and TKEEP usage to send an odd number of 32-bit words without stall for AI Engine to PL AXI4-Stream interfaces:

Table: TLAST and TKEEP Usage for Sending Odd Number of 32-bit Words to AI Engine

Interface Bit Width	TLAST	TKEEP	Note
32-bit	1	-1	32-bit word to send without stall.
64-bit	1	0x0F	Lowest 32-bit word to send without stall.
128-bit	1	0x000F	Lowest 32-bit word to send without stall.
128-bit	1	0x00FF	Lowest two 32-bit word to send without stall.
128-bit	1	0x0FFF	Lowest three 32-bit word to send without stall.

 **Note:** It is not valid to send only the highest partial data to AI Engine from PL interfaces. For example, setting TLAST=1 & TKEEP=0xF0 for sending only the highest 32-bit word is not valid.

The last odd number of 32-bit word from AI Engine to PL will also stall without asserting TLAST, even with 32-bit AI Engine to PL interface. To make sure the last odd number is pushed out to PL, assert TLAST inside AI Engine kernel by:

```
writeincr(out,value,true); //"true" to assert TLAST
```

AI Engine to PL Interface Text Input Format

The TXT file, which is used to provide data that represents a PLIO port or packet stream interface, has requirements that should be followed. For more details, see [TXT File Format](#) found in *AI Engine Tools and Flows User Guide* ([UG1076](#)).

Graph Programming Model

[Introduction to Graph Programming](#) briefly introduced the AI Engine programming model with I/O objects support. This chapter continues the discussion and describes other variations in detail.

Graph Topologies

Overview

Graphs can be programmed in a variety of topologies depending on the needs of your application. You can take advantage the flexibility of C++ to describe the graph topologies. Graphs can be embedded into other graphs as sub-graphs and this can be repeated many times, limited only by the performance of the machine on which you compile. Multiple graphs can be connected sequentially or in parallel, connecting themselves to simple kernels or I/Os.

Single Kernel Graph

The simplest basic graph is a single kernel that is instantiated in a class that inherits from the `adf::graph` class with specified inputs and outputs.

```
class SimplestGraph: public adf::graph {
private:
    adf::kernel k;

public:
    adf::port<input> din;
    adf::port<output> dout;
```

```
SimplestGraph() {
    k = adf::kernel::create(passthrough);
    adf::source(k) = "passthrough.cpp";
    adf::runtime<ratio>(k) = 0.9;
    adf::connect(din, k.in[0]);
    adf::connect(k.out[0], dout);
    dimensions(k.in[0]) = {FRAME_LENGTH};
    dimensions(k.out[0]) = {FRAME_LENGTH};
};
};
```

This simple graph above can be embedded into another graph. Data input file Input_64.txt represents the data from a PLIO connection to the embedded graph input. Data output file Output1.txt represents the data from the embedded graph output to the PLIO connection. The Simple graph is a sub-graph within the TestGraph.

```
class TestGraph: public adf::graph {
public:
    adf::input_plio plin1;
    adf::output_plio plout1;

    SimplestGraph Simple;

    TestGraph()
    {
        plin1 = adf::input_plio::create("input1",adf::plio_64_bits,"data/Input_64.txt",500);
        adf::connect(plin1.out[0],Simple.din);

        plout1 = adf::output_plio::create("output1",adf::plio_64_bits,"data/Output1.txt",500);
        adf::connect(Simple.dout,plout1.in[0]);
    };
};
```

The test bench program instantiates the test graph and calls the control commands to initialize run and end the graph.

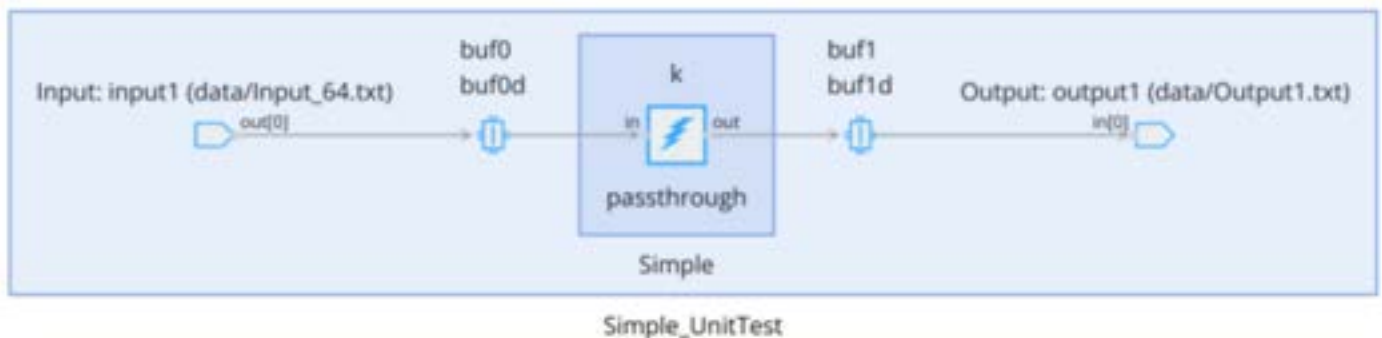
```
#include "graph.h"

TestGraph UnitTest;

int main(int argc, char ** argv) {
    UnitTest.init();
    UnitTest.run(NFRAMES*NITERATIONS);
    UnitTest.end();
    return 0;
}
```

The resulting graph display in the Vitis IDE:

Figure: Graph View



Linking Multiple Sub-Graphs

You can link several sub-graphs to create a larger test or application graph. You can develop graphs as entities independent from the I/Os, and embed these graphs in other graphs that contain the necessary PLIO and GMIO definitions to connect the graph to the external world. This allows you to independently test your graph and connect it to other graphs to build larger applications.

The benefit of this methodology is that you do not need to know the size of the buffers or the type of connection within the sub-graph. This allows you to independently develop the graphs and the kernels within the graph. When compiling the larger test or application graph the compiler will automatically adapt various I/O port types with the necessary hardware interposers. For example, as long as the connect statement which connects a stream PLIO input of a graph to a buffer input of another graph is specified in the test graph, the compiler will automatically connect the stream PLIO output port of one graph to the buffer input port of another graph.

```
class TestMoreComplexGraph: public adf::graph {
public:
    adf::input_plio plin;
    adf::output_plio plout;

    SplitGraph AddedGraph;
    MergeGraph AnotherGraph;
    SimplestGraph Simple;

    TestMoreComplexGraph()
    {
        plin = adf::input_plio::create("input2",adf::plio_64_bits,"data/Input_64.txt",500);
        adf::connect(plin.out[0],AddedGraph.din);

        adf::connect(AddedGraph.dout[0],Simple.din);
        adf::connect(Simple.dout,AnotherGraph.din[0]);
        adf::connect(AddedGraph.dout[1],AnotherGraph.din[1]);

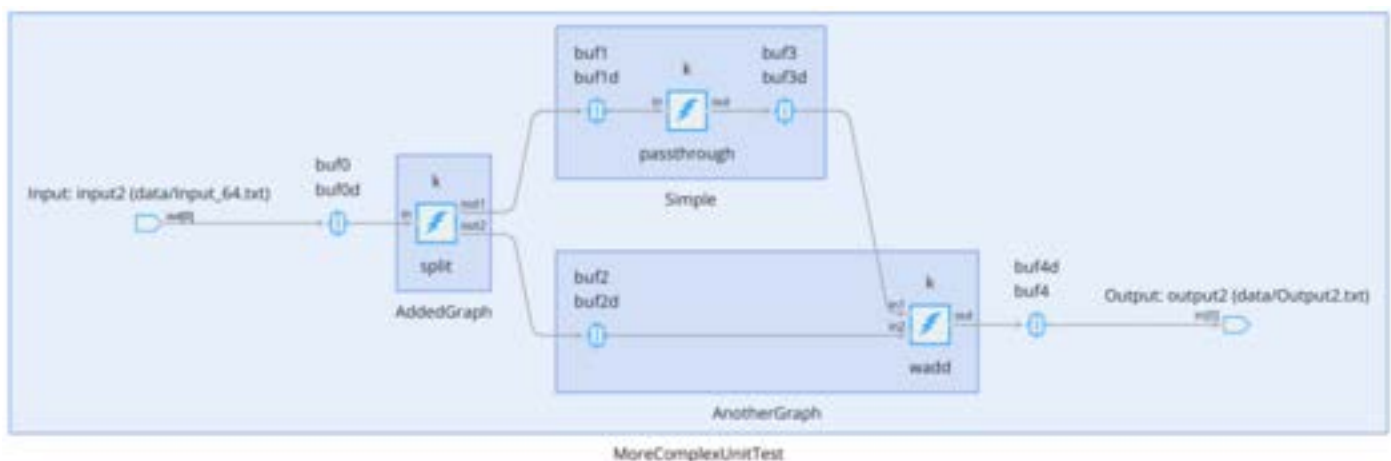
        plout = adf::output_plio::create("output2",adf::plio_64_bits,"data/Output2.txt",500);
        adf::connect(AnotherGraph.dout,plout.in[0]);
    };
};

TestMoreComplexGraph MoreComplexUnitTest;

int main(int argc, char ** argv) {

    MoreComplexUnitTest.init();
    MoreComplexUnitTest.run(NFRAMES*NITERATIONS);
    MoreComplexUnitTest.end();
    return 0;
}
```

Figure: Graph View



You do not need to change anything in graph Simple when it is connected to AddedGraph and AnotherGraph. It can be error prone as soon as you start to have many sub-graphs. To avoid errors, it is recommended that SimplestGraph graph definition can be in its own file. Any modifications to the graph file will change the behavior of UnitTest and MoreComplexUnitTest.

Hierarchical Graphs

A graph can contain a mix of kernels and sub-graphs. If you want to instantiate multiple kernels they can be declared as an array or with specific names. In the following example a template parameter specifies the number of kernels that will be declared as an array:

```
template <int NK>
class MultiKernelGraph: public adf::graph {
private:
    adf::kernel k[NK];

public:
    adf::port<input> din;
    adf::port<output> dout;

    MultiKernelGraph() {
        for(int i=0;i<NK;i++)
        {
            k[i] = adf::kernel::create(passthrough);
            adf::source(k[i]) = "kernels.cpp";

            adf::runtime<ratio>(k[i]) = 0.9;
        }
        adf::connect(din, k[0].in[0]);
        for(int i=0;i<NK-1;i++)
            adf::connect(k[i].out[0], k[i+1].in[0]);
        adf::connect(k[NK-1].out[0], dout);
    };
};

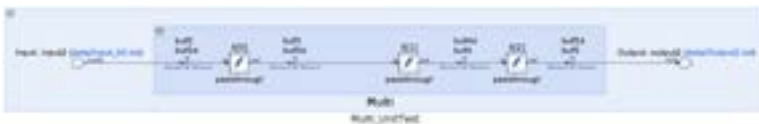
class TestGraphMulti: public adf::graph {
public:
    adf::input_plio plin;
    adf::output_plio plout;

    MultiKernelGraph<3> Multi;

    TestGraphMulti()
    {
        plin = adf::input_plio::create("input2",adf::plio_64_bits,"data/Input_64.txt",500);
        adf::connect(plin.out[0],Multi.din);

        plout = adf::output_plio::create("output2",adf::plio_64_bits,"data/Output2.txt",500);
        adf::connect(Multi.dout,plout.in[0]);
    };
};
```

Figure: Graph View



As seen earlier, a graph can contain kernels and sub-graphs. The test bench .cpp code can instantiate multiple graphs and run them independently.

```
TestGraphSimple Simple_UnitTest;
TestGraphMulti Multi_UnitTest;

int main(int argc, char ** argv) {

    Simple_UnitTest.init();
    Multi_UnitTest.init();

    Simple_UnitTest.run(NFRAMES*NITERATIONS);
```

```

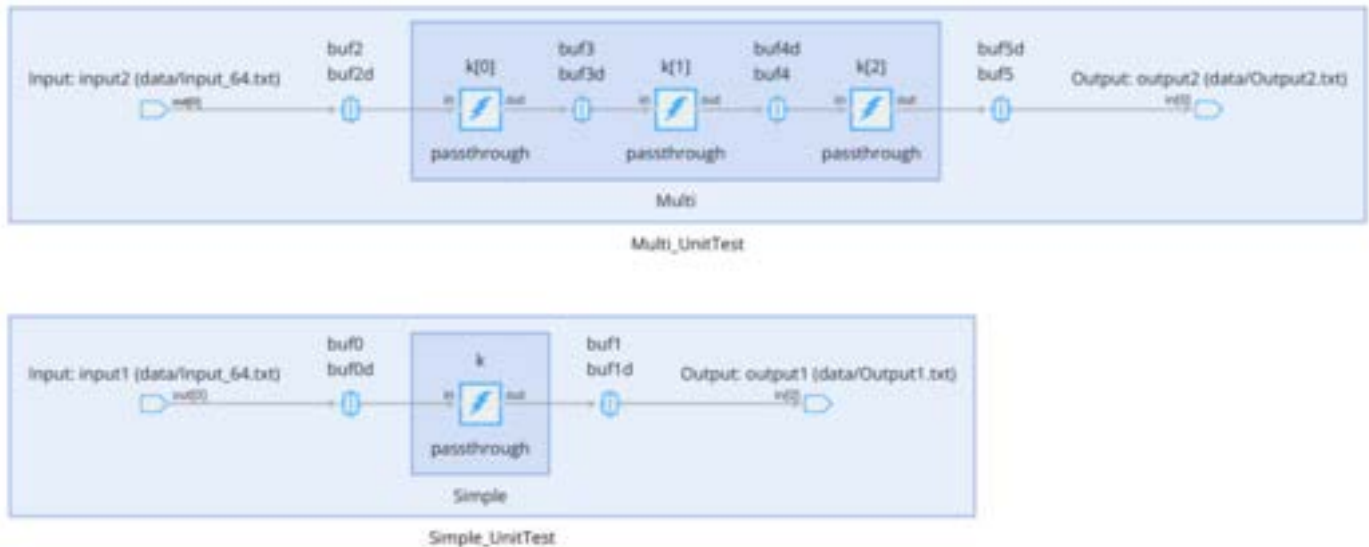
Multi_UnitTest.run(NFRAMES*NITERATIONS);

Simple_UnitTest.end();
Multi_UnitTest.end();
return 0;
}

```

The Vitis IDE displays two independent graphs on the graph view.

Figure: Multiple Graphs



These two graphs can also be instantiated in a third graph that is instantiated in the test bench:

```

class TestGraph: public adf::graph {
public:
    adf::input_plio plin1, plin2;
    adf::output_plio plout1, plout2;

    SimplestGraph Simple;
    MultiKernelGraph Multi;

    TestGraph()
    {
        plin1 = adf::input_plio::create("input1", adf::plio_64_bits, "data/Input_64.txt", 500);
        adf::connect(plin1.out[0], Simple.din);

        plout1 = adf::output_plio::create("output1", adf::plio_64_bits, "data/Output1.txt", 500);
        adf::connect(Simple.dout, plout1.in[0]);

        plin2 = adf::input_plio::create("input2", adf::plio_64_bits, "data/Input_64.txt", 500);
        adf::connect(plin2.out[0], Multi.din);

        plout2 = adf::output_plio::create("output2", adf::plio_64_bits, "data/Output2.txt", 500);
        adf::connect(Multi.dout, plout2.in[0]);

    };
};

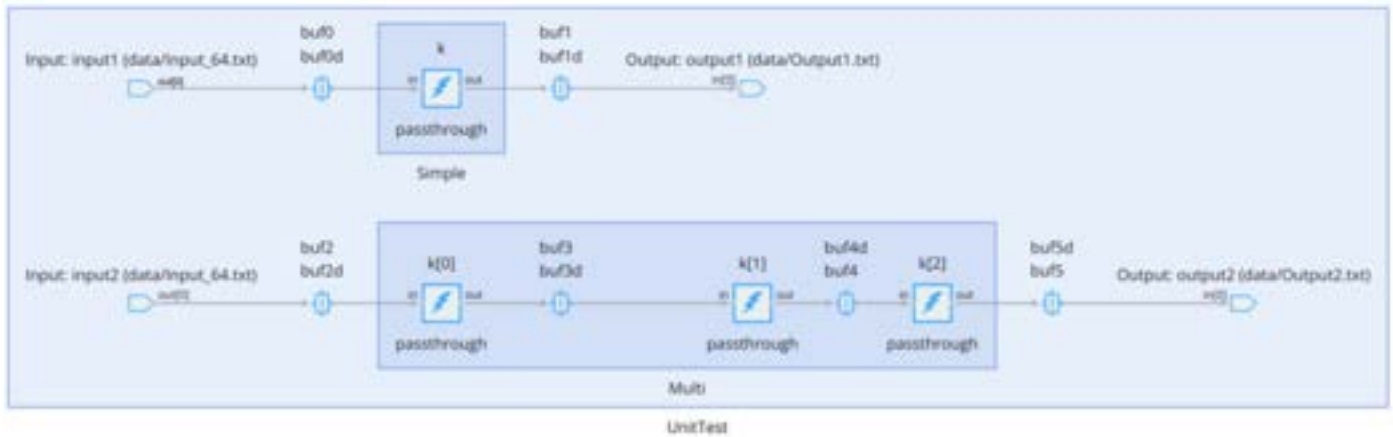
TestGraph UnitTest;

int main(int argc, char ** argv) {

    UnitTest.init();
    UnitTest.run(NFRAMES*NITERATIONS);
    UnitTest.end();
    return 0;
}

```

}

Figure: Composite Graph View

Much more complex graphs can be created including sequential and parallel kernels, and graphs where the data flow is split and merged multiple times throughout the graph, using buffers and streams to transfer data through the AI Engine array.

```

VeryComplexGraph() {
    // Declare Kernels
    for(int i=0;i<2;i++)
    {
        simple[i] = adf::kernel::create(passthrough);
        adf::source(simple[i]) = "kernels.cpp";
        adf::runtime<ratio>(simple[i]) = 0.9;
    }

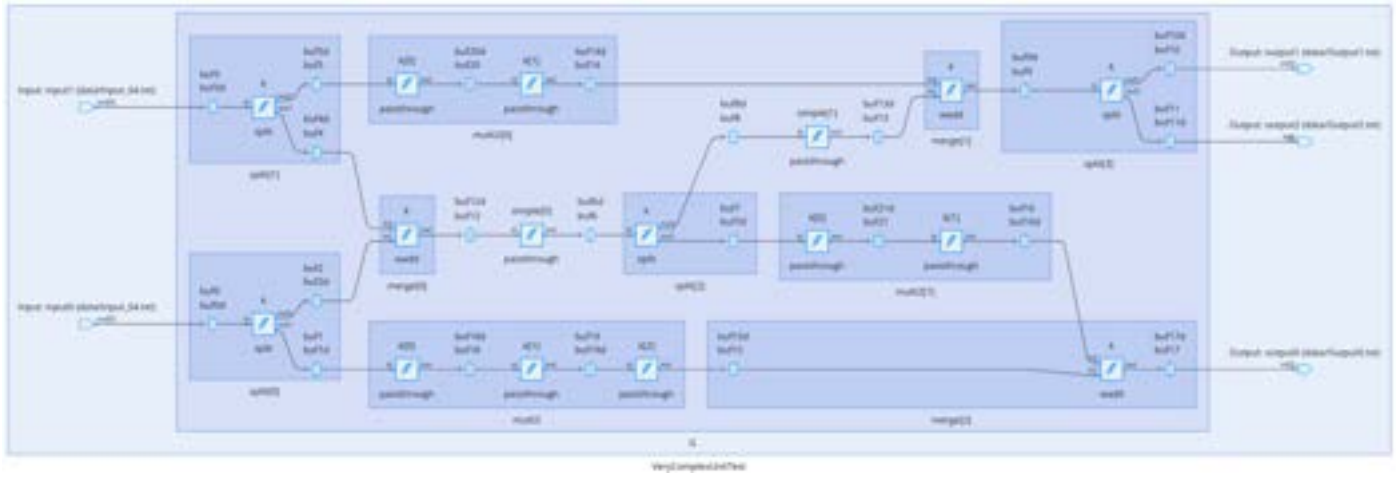
    // Connections
    // Inputs
    connect(din[0],split[0].din);
    connect(din[1],split[1].din);

    // Outputs
    connect(merge[2].dout,dout[0]);
    connect(split[3].dout[0],dout[1]);
    connect(split[3].dout[1],dout[2]);

    // Internal connections
    connect(split[0].dout[0],multi3.din);
    connect(multi3.dout,merge[2].din[0]);
    connect(split[0].dout[1],merge[0].din[0]);
    connect(split[1].dout[0],merge[0].din[1]);
    connect(merge[0].dout,simple[0].in[0]);
    connect(simple[0].out[0],split[2].din);
    connect(split[2].dout[0],multi2[1].din);
    connect(multi2[1].dout,merge[2].din[1]);
    connect(split[2].dout[1],simple[1].in[0]);
    connect(simple[1].out[0],merge[1].din[0]);
    connect(split[1].dout[1],multi2[0].din);
    connect(multi2[0].dout,merge[1].din[1]);
    connect(merge[1].dout,split[3].din);
};

```

Figure: Complex Graph View



In this graph view, a complex dataflow diverges and converges in multiple points. If a part of the dataflow needs to go in and out of the PL, you will see the corresponding output ports and input ports that you connect to your PL kernels during the link phase.

Recursive Graphs

One feature of C++ is recursive templates. Graphs can be created recursively by taking advantage of the recursive templates.

```
template<int Level>
class RecursiveGraph: public adf::graph {
private:
    adf::kernel k;
    RecursiveGraph<Level-1> RG;

public:
    adf::port<input> din;
    adf::port<output> dout;

    RecursiveGraph() {
        k = adf::kernel::create(passthrough);
        adf::source(k) = "kernels.cpp";
        adf::runtime<ratio>(k) = 0.9;

        adf::connect(din, k.in[0]);
        adf::connect(k.out[0], RG.din);
        adf::connect(RG.dout, dout);
    };
};

template<>
class RecursiveGraph<1>: public adf::graph {
private:
    adf::kernel k;

public:
    adf::port<input> din;
    adf::port<output> dout;

    RecursiveGraph() {
        k = adf::kernel::create(passthrough);
        adf::source(k) = "kernels.cpp";
        adf::runtime<ratio>(k) = 0.9;

        adf::connect(din, k.in[0]);
        adf::connect(k.out[0], dout);
    };
};
```

The class `RecursiveGraph` is parametrized with the template parameter `Level` which represents the depth of the recursive function. Within the `RecursiveGraph` class, an instantiation of the same class is performed as `RecursiveGraph<Level-1> RG;` with the template

```
class TestRecursiveGraph: public adf::graph {
public:
    adf::input_plio plin;
    adf::output_plio plout;

    RecursiveGraph<5> Recursive;

    TestRecursiveGraph()
    {
        plin = adf::input_plio::create("input",adf::plio_64_bits,"data/Input_64.txt",500);
        adf::connect(plin.out[0],Recursive.din);

        plout = adf::output_plio::create("output",adf::plio_64_bits,"data/Output.txt",500);
        adf::connect(Recursive.dout,plout.in[0]);
    };
};
```

```
TestRecursiveGraph RecursiveUnitTest;

int main(int argc, char ** argv) {

    RecursiveUnitTest.init();
    RecursiveUnitTest.run(NFRAMES*NITERATIONS);
    RecursiveUnitTest.end();
    return 0;
}
```

Figure: Recursive Graph and Sequential Multi-kernel Graph View



The Vitis libraries targeting AI Engine are currently split into two levels:

Basic kernels

Graphs that use multiple Level 1 kernels with parameters specified through templates. The following examples contain PL data movers and host code that uses Level 2 functions in hardware.

9/24/25, 9:28 PM

throughput of the overall system. The system settings are shown below:

```
#include <adf.h>
#include "fir_sr_sym_graph.hpp"
#include "fir_interpolate_hb_graph.hpp"

// Filter parameters
#define DATA_TYPE cint16
#define COEFF_TYPE int16

#define FIR_LEN_CHAN 151
#define SHIFT_CHAN 15
#define ROUND_MODE_CHAN 0
#define AIES_CHAN 5

#define FIR_LEN_HB 43
#define SHIFT_HB 15
#define ROUND_MODE_HB 0

#define WINDOW_SIZE 1024

// Simulation parameters
#define NUM_ITER 8
```

The graph instantiating the filter sub-graph is shown below:

```
namespace dsplib = xf::dsp::aie;

class FirGraph: public adf::graph
{
private:
    // Channel Filter coefficients
    std::vector<int16> chan_taps = std::vector<int16>{
        -17, -65, -35, 34, -13, -6, 18, -22,
        18, -8, -5, 18, -26, 26, -16, -1,
        21, -36, 40, -31, 8, 21, -46, 59,
        -53, 26, 13, -54, 81, -83, 56, -6,
        -54, 102, -122, 101, -43, -38, 116, -164,
        161, -102, 1, 114, -204, 235, -190, 74,
        83, -231, 319, -310, 193, 5, -229, 406,
        -468, 380, -147, -174, 487, -684, 680, -437,
        -10, 553, -1030, 1262, -1103, 474, 596, -1977,
        3451, -4759, 5660, 26983};

    // HalfBand Filter coefficients
    std::vector<int16> hb_taps = std::vector<int16>{
        23, -63, 143, -281, 503, -845, 1364, -2173,
        3557, -6568, 20729, 32767};

    using channel = xf::dsp::aie::fir::sr_sym::
    fir_sr_sym_graph<DATA_TYPE, COEFF_TYPE, FIR_LEN_CHAN, SHIFT_CHAN, ROUND_MODE_CHAN, WINDOW_SIZE, AIES_CHAN>;

    using halfband = xf::dsp::aie::fir::interpolate_hb::
    fir_interpolate_hb_graph<DATA_TYPE, COEFF_TYPE, FIR_LEN_HB, SHIFT_HB, ROUND_MODE_HB, WINDOW_SIZE>;

public:
    adf::port<input> in;
    adf::port<output> out;

    // Constructor - with FIR graph classes initialization
    FirGraph(){

        channel chan_FIR(chan_taps);
        halfband hb_FIR(hb_taps);

        // Margin gets automatically added within the FIR graph class.
```

```
// Margin equals to FIR length rounded up to nearest multiple of 32 Bytes.
adf::connect(in, chan_FIR.in[0]);
adf::connect(chan_FIR.out[0], hb_FIR.in[0]);
adf::connect(hb_FIR.out[0], out);
};
};
```

As usual the connections to the external world are specified in another graph to allow for more flexibility in this graph use:

```
class TopGraph : public adf::graph
{
public:
    adf::input_plio in;
    adf::output_plio out;

    FirGraph F;

    TopGraph()
    {
        in = adf::input_plio::create("128 bits read in",
                                     adf::plio_128_bits,"data/input_128b.txt", 250);
        out = adf::output_plio::create("128 bits read out",
                                       adf::plio_128_bits,"data/output_128b.txt", 250);

        adf::connect (in.out[0],F.in);
        adf::connect (F.out, out.in[0]);
    };
};
```

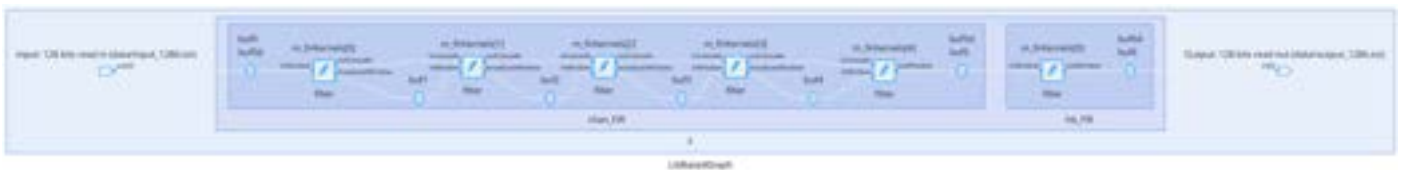
And, finally the test bench used to launch the simulation is shown below:

```
TopGraph LibBasedGraph;

int main(void) {
    LibBasedGraph.init() ;
    LibBasedGraph.run(NUM_ITER) ;
    LibBasedGraph.end() ;
    return 0 ;
}
```

The graph view in the Vitis IDE is shown below:

Figure: DPSLib Graph View



As you can see `chan_FIR` is built on five kernels (`AIES_CHAN = 5`) which are grouped in a single graph, and `hb_FIR` is based on a single kernel in its own sub-graph.

!! Important: Kernels and sub-graphs must be part of the graph class. There is no support for conditional declaration or pointer-based declaration. In the same way input/output ports should be also member of the graph class.

Programming Model Features

The AI Engine programming model has the following features.

- All I/O objects (input_plio/output_plio and input_gmio/output_gmio) are public members of graph class. Looping through I/O objects is supported. This makes I/O objects construction easier and less error prone. For example:

```
class mygraph : public graph
{
public:
    // Declare N kernels.
    kernel k[N];
    // Declare N input_gmio(s) and output_gmio(s)
    input_gmio gmioIn[N];
    output_gmio gmioOut[N];

    mygraph()
    {
        // Create multiple input_gmio(s)/output_gmio(s) and connections
        for (int i=0; i<N; i++)
        {
            gmioIn[i] = input_gmio::create("gmioIn" + std::to_string(i), 64, 1);
            gmioOut[i] = output_gmio::create("gmioOut" + std::to_string(i), 64, 1);
            connect(gmioIn[i].out[0], k[i].in[0]);
            connect(k[i].out[0], gmioOut[i].in[0]);

            dimensions(k[i].in[0]) = {FRAME_LENGTH};
            dimensions(k[i].out[0]) = {FRAME_LENGTH};
        }
    }
};
```

Note: While not required, you can provide a logical name when creating the input_plio, output_plio, input_gmio and output_gmio objects. If not provided, the AI Engine compiler will automatically set a unique logical name.

- The programming model provides ease of use for connections within the design. With the input_plio, output_plio, input_gmio and output_gmio objects in the local scope, connect calls are not required to provide a unique connection name.
- The programming model supports the direction property of input_plio, output_plio, input_gmio and output_gmio objects. The direction property is determined within object construction.

An example of the programming model that addresses these previously listed features is as follows.

```
/* A graph with both PLIO and GMIO */
class mygraph : graph
{
public:

    /* Classes support input_gmio/output_gmio and input_plio/output_plio */
    input_gmio gm_in;
    output_gmio gm_out;

    input_plio pl_in;
    output_plio pl_out;

    kernel k1, k2;
    mygraph()
    {
        k1 = kernel::create(...);
        k2 = kernel::create(...);

        /* create() API for PLIO and GMIO objects support same arguments specified as in global scope */
        gm_in = input_gmio::create("GMIO_In0", /*const std::string& logical_name*/
                                   64,          /*size_t burst_length*/
                                   1            /*size_t bandwidth*/);

        gm_out = output_gmio::create("GMIO_Out0", 64, 1);

        pl_in = input_plio::create("PLIO_In0",      /* std::string logical_name */
                                   plio_32_bits,    /* enum plio_type plio_width */
                                   "data/input.txt", /* std::string data_file */
```

```

                250.0                /* double frequency */ );

    pl_out = output_plio::create("PLIO_Out0", plio_32_bits, "data/output.txt", 250.0);

    /* Each input_gmio/output_gmio and input_plio/output_plio supports 1 port per direction */
    connect(gm_in.out[0], k1.in[0]);
    connect(k1.out[0], gm_out.in[0]);

    connect(pl_in.out[0], k2.in[0]);
    connect(k2.out[0], pl_out.in[0]);

    location<GMIO>(gm_in) = shim(col);
    location<GMIO>(gm_out) = shim(col, ch_num); /* not supported */


    location<PLIO>(pl_in) = shim(col);
    location<PLIO>(pl_out) = shim(col, ch_num);
}


};

```

The previous code snippet demonstrates:

- input_plio, output_plio, input_gmio, and output_gmio are public objects within the graph.
- input_plio, and output_plio classes are inherited from the PLIO class.
- input_gmio, and output_gmio classes are inherited from the GMIO class.
- input_plio, output_plio, input_gmio, and output_gmio object constructions contain the direction property.
- connect call does not require unique name.
- Location constraints can be applied to the input_plio, output_plio, input_gmio, and output_gmio objects directly with the exception of GMIO channel constraints which are not supported in this release.

 **Note:** GMIO shim tile location constraint with DMA channel is not supported.

 **Note:** Host application calling profiling APIs need reference object's public members, mygraph.gm_in, mygraph.gm_out, mygraph.pl_in, mygraph.pl_out. For additional details about profiling APIs, see [Performance Analysis of AI Engine Graph Application during Simulation](#) in *AI Engine Tools and Flows User Guide* (UG1076).

An example of a profile call from programming model is as follows.

```

event::handle handle0 = event::start_profiling(mygraph.gm_in,
event::io_stream_start_to_bytes_transferred_cycles, sizeIn*sizeof(cint16));

```

!! Important: The input_plio, output_plio, input_gmio, and output_gmio objects must have a unique name. If a name is not unique, the following error message is issued:

```

ERROR: [aiecompiler 77-4551] The logical name DataIn1 of node i6 conflicts with the logical/qualified name of node i0.

```

Configuring input_plio/output_plio

An input_plio/output_plio object can be configured to make external stream connections that cross the AI Engine to programmable logic (PL) boundary. This situation arises when a hardware platform is designed separately and the PL blocks are already instantiated inside the platform. This hardware design is exported from the Vivado tools as a package XSA and it should be specified when creating a new project in the AMD Vitis™ tools using that platform. The XSA contains a logical architecture interface specification that identifies which AI Engine I/O ports can be supported by the platform. The following is an example interface specification containing stream ports (looking from the AI Engine perspective).

Table: Example Logical Architecture Port Specification

AI Engine Port	Annotation	Type	Direction	Data Width	Clock Frequency (MHz)
S00_AXIS	Weight0	stream	slave	32	300
S01_AXIS	DataIn0	stream	slave	32	300
M00_AXIS	Dataout0	stream	master	32	300

This interface specification describes how the platform exports two stream input ports (slave port on the AI Engine array interface) and one stream output port (master port on the AI Engine array interface). An `input_plio/output_plio` attribute specification is used to represent and connect these interface ports to their respective destination or source kernel ports in data flow graph.

The following example shows how the `input_plio/output_plio` attributes shown in the previous table can be used in a program to read input data from a file or write output data to a file. The width and frequency of the `input_plio/output_plio` port are also provided in the PLIO constructor.

```
adf::input_plio wts = adf::input_plio::create("Weight0", adf::plio_32_bits, "inputwts.txt", 300);
adf::input_plio din = adf::input_plio::create("Datain0", adf::plio_32_bits, "din.txt", 300);
adf::output_plio out = adf::output_plio::create("Dataout0", adf::plio_32_bits, "dout.txt", 300);
```

When simulated, the input weights and data are read from the two supplied files and the output data is produced in the designated output file in a streaming manner.

When a hardware platform is exported, all the AI Engine to PL stream connections are already routed to specific physical channels from the PL side.

Wide Stream Data Path PLIO

Typically, the AI Engine array runs at a higher clock frequency than the internal programmable logic. The `--pl-freq` option can be set to specify the frequency at which the PL blocks are expected to run. To balance the throughput between AI Engine and internal programmable logic, it is possible to design the PL blocks for a wider stream data path (64-bit, 128-bit), which is then sequentialized automatically into a 32-bit stream on the AI Engine stream network at the AI Engine to PL interface crossing.

The following example shows how wide stream `input_plio/output_plio` attributes can be used in a program to read input data from a file or write output data to a file.

```
adf::output_plio pl_out = adf::output_plio::create("TestLogicalNameOut", adf::plio_128_bits, "data/
output.txt");
adf::input_plio pl_in = adf::input_plio::create("TestLogicalNameIn", adf::plio_128_bits, "data/input.txt");
...
adf::connect(pl_in.out[0], kernel_first.in[0]);
adf::connect(kernel_last.out[0], pl_out.in[0]);
```

In the previous example, two 128-bit PLIO attributes is declared: one for input and one for output. The `input_plio` and `output_plio` are then hooked up to the graph in the usual way. Data files specified in the `input_plio/output_plio` attributes are then automatically opened for reading the input or writing the output respectively.

When simulating `input_plio/output_plio` with data files, the data should be organized to accommodate both the width of the PL block as well as the data type of the connecting port on the AI Engine block. For example, a data file representing 32-bit PL interface to an AI Engine kernel expecting `int16` should be organized as two columns per row, where each column represents a 16-bit value. As another example, a data file representing 64-bit PL interface to an AI Engine kernel expecting `cint16` should be organized as four columns per row, where each column represents a 16-bit real or imaginary value. The same 64-bit PL interface feeding an AI Engine kernel with `int32` port would need to organize the data as two columns per row of 32-bit real values. The following examples show the format of the input file for the previously mentioned scenarios.

64-bit PL interface feeding AI Engine kernel expecting `cint16`

input file:

```
0 0 0 0
1 1 1 1
2 2 2 2
```

64-bit PL interface feeding AI Engine kernel expecting `int32`

input file:

```
0 0
1 1
2 2
```


With these wide PLIO attribute specifications, the AI Engine compiler generates the AI Engine array interface configuration to convert a 64-bit or 128-bits data into a sequence of 32-bit words. The AXI4-Stream protocol followed with all PL IP blocks ensures that partial data can also be sent on a wider data path with the appropriate strobe signals describing which words are valid.

Related Information

[Adaptive Data Flow Graph Specification Reference](#)

Configuring input_gmio/output_gmio

A `input_gmio` or `output_gmio` object is used to make external memory-mapped connections to or from the global memory. These connections are made between an AI Engine graph and the logical global memory ports of a hardware platform design. The platform can be a base platform from AMD or a custom platform that is exported from the Vivado tools as an AMD device support archive (XSA) package. AI Engine tools support mapping the `input_gmio` or `output_gmio` ports to the tile DMA, one to one. It does not support mapping multiple `input_gmio/output_gmio` ports to one tile DMA channel. There is a limit on the number of `input_gmio/output_gmio` ports supported for a given device. For example, the XCVC1902 device on the VCK190 board has 16 AI Engine to NoC master units (NMU) in total. For each AI Engine to NMU, it supports two MM2S and two S2MM channels. So, there can be at most 32 AI Engine GMIO inputs, and 32 AI Engine GMIO outputs. Note that it can be further limited by the existing hardware platform.

 **Note:** GMIO channel constraints should not be used for AI Engine compilation.

While developing data flow graph applications on top of an existing hardware platform, you need to know what global memory ports are exported by the underlying XSA and their functionality. In particular, any input or output ports exposed on the platform are recorded within the XSA and can be viewed as a logical architecture interface.

Programming Model for AI Engine–DDR Memory Connection

The `input_gmio/output_gmio` port attribute can be used to initiate AI Engine–DDR memory read and write transactions in the PS program. This enables data transfer between an AI Engine and the DDR controller through APIs written in the PS program. The following example shows how to use GMIO APIs to send data to an AI Engine for processing and retrieve the processed data back to the DDR through the PS program.

```
graph.h

class mygraph: public adf::graph
{
private:
    adf::kernel k_m;

public:
    adf::output_gmio gmioOut;
    adf::input_gmio gmioIn;
    mygraph()
    {
        k_m = adf::kernel::create(weighted_sum_with_margin);
        gmioOut = adf::output_gmio::create("gmioOut", 64, 1000);
        gmioIn = adf::input_gmio::create("gmioIn", 64, 1000);

        adf::connect(gmioIn.out[0], k_m.in[0]);
        adf::connect(k_m.out[0], gmioOut.in[0]);

        dimensions(k_m.in[0]) = {256};
        dimensions(k_m.out[0]) = {256};

        adf::source(k_m) = "weighted_sum.cc";
        adf::runtime<adf::ratio>(k_m) = 0.9;
    };
};
```

graph.cpp

```
myGraph gr;
int main(int argc, char ** argv)
{
    const int BLOCK_SIZE=256;
    int32 *inputArray=(int32*)GMI0::malloc(BLOCK_SIZE*sizeof(int32));
    int32 *outputArray=(int32*)GMI0::malloc(BLOCK_SIZE*sizeof(int32));

    // provide input data to AI Engine in inputArray
    for (int i=0; i<BLOCK_SIZE; i++) {
        inputArray[i] = i;
    }

    gr.init();

    gr.gmioIn.gm2aie_nb(inputArray, BLOCK_SIZE*sizeof(int32));
```



```

    gr.gmioOut.aie2gm_nb(outputArray, BLOCK_SIZE*sizeof(int32));

    gr.run(8);

    gr.gmioOut.wait();

    // can start to access output data from AI Engine in outputArray
    ...

    GMI0::free(inputArray);
    GMI0::free(outputArray);
    gr.end();
}

```

This example declares two I/O objects: `gmioIn` represents the DDR memory space to be read by the AI Engine, and `gmioOut` represents the DDR memory space to be written by the AI Engine. The constructor specifies the logical name of the GMIO, burst length (that can be 64, 128, or 256 bytes) of the memory-mapped AXI4 transaction, and the required bandwidth (in MB/s).

```

gmioOut = adf::output_gmio::create("gmioOut", 64, 1000);
gmioIn  = adf::input_gmio::create("gmioIn", 64, 1000);

```

The application graph (`myGraph`) has an input port (`myGraph::gmioIn`) connecting to the processing kernels. The kernels produce data to the output port (`myGraph::gmioOut`) producing the processed data from the kernels. The following code connects the input port of the graph and connects to the output port of the graph.

```

adf::connect(gmioIn.out[0], k_m.in[0]);
adf::connect(k_m.out[0], gmioOut.in[0]);
dimensions(k_m.in[0]) = {256};
dimensions(k_m.out[0]) = {256};

```

Inside the main function, two 256-element `int32` arrays are allocated by `GMI0::malloc`. The `inputArray` points to the memory space to be read by the AI Engine and the `outputArray` points to the memory space to be written by the AI Engine. In Linux, the virtual address passed to `GMI0::gm2aie_nb`, `GMI0::aie2gm_nb`, `GMI0::gm2aie` and `GMI0::aie2gm` must be allocated by `GMI0::malloc`. After the input data is allocated, it can be initialized.

```

const int BLOCK_SIZE=256;
int32 *inputArray=(int32*)GMI0::malloc(BLOCK_SIZE*sizeof(int32));
int32 *outputArray=(int32*)GMI0::malloc(BLOCK_SIZE*sizeof(int32));

```

`gr.gmioIn.gm2aie_nb()` is used to initiate memory-mapped AXI4 transactions for the AI Engine to read from DDR memory spaces. The first argument in `gr.gmioIn.gm2aie_nb()` is the pointer to the start address of the memory space for the transaction. The second argument is the transaction size in bytes. The memory space for the transaction must be within the memory space allocated by `GMI0::malloc`. Similarly, `gr.gmioOut.aie2gm_nb()` is used to initiate memory-mapped AXI4 transactions for the AI Engine to write to DDR memory spaces. `gr.gmioOut.gm2aie_nb()` or `gr.gmioOut.aie2gm_nb()` is a non-blocking function in a sense that it returns immediately when the transaction is issued, that is, it does not wait for the transaction to complete. By contrast, `gr.gmioIn.gm2aie()` or `gr.gmioOut.aie2gm()` behaves in a blocking manner.


In this example, assuming in one iteration, the graph consumes 32 `int32` data from the input port and produces 32 `int32` data to the output port. To run eight iterations, the graph consumes 256 `int32` data and produces 256 `int32` data. The corresponding memory-mapped AXI4 transactions are initiated using the following code, one `gr.gmioIn.gm2aie_nb()` call to issue a read transaction for eight-iteration worth of data, and one `gr.gmioOut.aie2gm_nb()` call to issue a write transaction for eight-iteration worth of data.

```

gr.gmioIn.gm2aie_nb(inputArray, BLOCK_SIZE*sizeof(int32));
gr.gmioOut.aie2gm_nb(outputArray, BLOCK_SIZE*sizeof(int32));

```

`gr.run(8)` is also a non-blocking call to run the graph for eight iterations. To synchronize between the PS and AI Engine for DDR memory read/write access, you can use `gr.gmioOut.wait()` to block PS execution until the GMIO transaction is complete. In this example, `gr.gmioOut.wait()` is called to wait for the output data to be written to `outputArray` DDR memory space.

 **Note:** The memory is non-cachable for GMIO in Linux.

After that, the PS program can access the data. When PS has completed processing, the memory space allocated by `GMI0::malloc` can be released by `GMI0::free`.

```

GMI0::free(inputArray);

```

```
GMI0::free(outputArray);
```

The `input_gmio/output_gmio` APIs can be used in various ways to perform different level of control for read/write access and synchronization between the AI Engine, PS, and DDR memory. Either `input_gmio::gm2aie`, `output_gmio::aie2gm`, `input_gmio::gm2aie_nb` or `output_gmio::aie2gm_nb` can be called multiple times to associate different memory spaces for the same `input_gmio/output_gmio` object during different phases of graph execution. Different `input_gmio/output_gmio` objects can be associated with the same memory space for in-place AI Engine–DDR read/write access. Blocking versions of `input_gmio::gm2aie` and `output_gmio::aie2gm` APIs themselves are synchronization point for data transportation and kernel execution. Calling `input_gmio::gm2aie` (or `output_gmio::aie2gm`) is equivalent to calling `input_gmio::gm2aie_nb` (or `output_gmio::aie2gm_nb`) followed immediately by `output_gmio::wait`. The following example shows the combination of the aforementioned use cases.

```
myGraph gr;

int main(int argc, char ** argv)
{
    const int BLOCK_SIZE=256;
    // dynamically allocate memory spaces for in-place AI Engine read/write access
    int32* inoutArray=(int32*)GMI0::malloc(BLOCK_SIZE*sizeof(int32));
    gr.init();

    for (int k=0; k<4; k++)
    {
        // provide input data to AI Engine in inoutArray
        for(int i=0;i<BLOCK_SIZE;i++){
            inoutArray[i]=i;
        }

        gr.run(8);
        for (int i=0; i<8; i++)
        {
            gr.gmioIn.gm2aie(inoutArray+i*32, 32*sizeof(int32)); //blocking call to ensure transaction data
is read from DDR to AI Engine
            gr.gmioOut.aie2gm_nb(inoutArray+i*32, 32*sizeof(int32));
        }
        gr.gmioOut.wait();

        // can start to access output data from AI Engine in inoutArray
        // ...
    }
    GMI0::free(inoutArray);
    gr.end();
    return 0;
}
```

In the previous example, the two GMIO objects `gmioIn` and `gmioOut` are using the same memory space allocated by `inoutArray` for in-place read and write access.

Without knowing data flow dependency among the kernels inside the graph, and to ensure write-after-read for the `inoutArray` memory space, the blocking version `gr.gmioIn.gm2aie()` is called to ensure transaction data is copied from DDR memory to AI Engine local memory before issuing a write transaction to the same memory space in `gr.gmioOut.aie2gm_nb()`.

```
gr.gmioIn.gm2aie(inoutArray+i*32, 32*sizeof(int32)); //blocking call to ensure transaction data is read from
DDR to AI Engine
gr.gmioOut.aie2gm_nb(inoutArray+i*32, 32*sizeof(int32));
```

`gr.gmioOut.wait()` is to ensure that data has been migrated to DDR memory. After it is done, the PS can access output data for post-processing.

The graph execution is divided into four phases in the for loop, for `(int k=0; k<4; k++)`. `inoutArray` can be re-initialized in the for loop with different data to be processed in different phases.

Hardware Emulation and Hardware Flows

`input_gmio/output_gmio` is not only used with the AI Engine simulator, but can also work in hardware emulation and hardware flows. To allow it to work in hardware emulation and hardware flows, add the following code to `graph.cpp`.

```
#if !defined(__AIESIM__) && !defined(__X86SIM__)
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"
// Create XRT device handle for ADF API

char* xclbinFilename = argv[1];
auto dhd1 = xrtDeviceOpen(0);//device index=0
xrtDeviceLoadXclbinFile(dhd1,xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhd1, uuid);

adf::registerXRT(dhd1, uuid);
#endif
```

Using the guard macro `__AIESIM__` and `__X86SIM__`, the same version of `graph.cpp` can work for the AI Engine simulator, `x86simulator`, hardware emulation, and hardware flows. Note that the preceding code should be placed before calling the graph or the GMIO ADF APIs. At the end of the program, close the device using the `xrtDeviceClose()` API.

```
#if !defined(__AIESIM__) && !defined(__X86SIM__)
xrtDeviceClose(dhd1);
#endif
```

To compile the code for hardware flow, see [Programming the PS Host Application](#) in *AI Engine Tools and Flows User Guide* ([UG1076](#)).

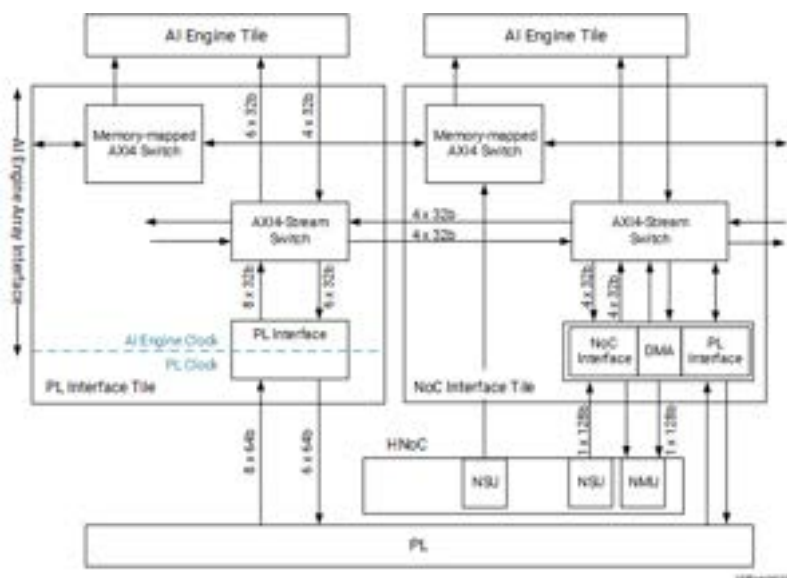
Performance Comparison Between AI Engine/PL and AI Engine/NoC Interfaces

The AI Engine array interface consists of the PL and NoC interface tiles. The AI Engine array interface tiles manage the two following high performance interfaces.

- AI Engine to PL
- AI Engine to NoC

The following image shows the AI Engine array interface structure.

Figure: AI Engine Array Interface Topology




One AI Engine to PL interface tile contains eight streams from the PL to the AI Engine and six streams from the AI Engine to the PL. The following table shows one AI Engine to PL interface tile capacity.

Table: AI Engine Array Interface to PL Interface Bandwidth Performance

Connection Type	Number of Connections	Data Width (bits)	Clock Domain	Bandwidth per Connection (GB/s)	Aggregate Bandwidth (GB/s)
-----------------	-----------------------	-------------------	--------------	---------------------------------	----------------------------

Connection Type	Number of Connections	Data Width (bits)	Clock Domain	Bandwidth per Connection (GB/s)	Aggregate Bandwidth (GB/s)
PL to AI Engine array interface	8	64	PL (500 MHz)	4	32
AI Engine array interface to PL	6	64	PL (500 MHz)	4	24

 **Note:** All bandwidth calculations in this section assume a nominal 1 GHz AI Engine clock for a -1L speed grade device at VCCINT = 0.70V with the PL interface running at half the frequency of the AI Engine as an example.

The exact number of PL and NoC interface tiles is device-specific. For example, in the VC1902 device, there are 50 columns of AI Engine array interface tiles. However, only 39 array interface tiles are available to the PL interface. Therefore, the aggregate bandwidth for the PL interface is approximately:

- 24 GB/s * 39 = 0.936 TB/s from AI Engine to PL
- 32 GB/s * 39 = 1.248 TB/s from PL to AI Engine

The number of array interface tiles available to the PL interface and total bandwidth of the AI Engine to PL interface for other devices and across different speed grades is specified in *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* (DS957).

The input_gmio/output_gmio attribute uses DMA in the AI Engine to NoC interface tile. The DMA has two 32-bit incoming streams from the AI Engine and two 32-bit streams to the AI Engine. In addition, it has one 128-bit memory mapped AXI master interface to the NoC NMU.

The performance of one AI Engine to NoC interface tile is shown in the following table.

Table: AI Engine to NoC Interface Tile Bandwidth Performance

Connection Type	Number of connections	Bandwidth per connection (GB/s)	Aggregate Bandwidth (GB/s)
AI Engine to DMA	2	4	8
DMA to NoC	1	16	16
DMA to AI Engine	2	4	8
NoC to DMA	1	16	16

The exact number of AI Engine to NoC interface tiles is device-specific. For example, in the VC1902 device, there are 16 AI Engine to NoC interface tiles. So, the aggregate bandwidth for the NoC interface is approximately:

- 8 GB/s * 16 = 128 GB/s from AI Engine to PL
- 8 GB/s * 16 = 128 GB/s from PL to AI Engine

When accessing DDR memory, the integrated DDR memory controller (DDRMC) number in the platform limits the performance of DDR memory read and write. For example, if all four DDRMCs in a VC1902 device are fully used, the hard limit to access DDR memory is as follows.

- 3200 Mb/s * 64 bit * 4 DDRMCs / 8 = 102.4 GB/s

The performance of input_gmio/output_gmio accessing DDR memory through the NoC is further restricted by the NoC lane number in the horizontal and vertical NoC, inter-NoC configurations, and QoS. Note that DDR memory read and write efficiency is largely affected by the access pattern and other overheads. For more information about the NoC, memory controller use, and performance numbers, see the *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

For a single connection from the AI Engine or to the AI Engine, both input_plio/output_plio and input_gmio/output_gmio have a hard bandwidth limit of 4 GB/s. Some advantages and disadvantages for choosing input_plio/output_plio or input_gmio/output_gmio are shown in the following table.

Table: Comparison of input_plio/output_plio vs input_gmio/output_gmio

	input_plio/output_plio	input_gmio/output_gmio
Advantages	<ul style="list-style-type: none"> • Number of AI Engine to PL interface streams are larger, hence larger aggregate bandwidth • No interference between different stream connections • Supports packet switching 	<ul style="list-style-type: none"> • No PL resource required • No timing closure requirement

	input_plio/output_plio	input_gmio/output_gmio
Disadvantages	<ul style="list-style-type: none"> • Congestion risk if there are too many stream connections in a region of the device • Timing closure required for achieving best performance 	<ul style="list-style-type: none"> • Less input_gmio/output_gmio ports available • Aggregate bandwidth is lower • Multiple input_gmio/output_gmio ports competing for bandwidth

Data Throughput Estimate in Hardware

When a system does not meet performance requirements, the root cause could be one of the following:

- Some kernels of the graph cannot keep pace with the input/output sample rate, leading to a global performance degradation.
- The throughput at the AI Engine array interface does not meet the performance requirements for the following reasons:
 - Maximum theoretical throughput is not sufficient.
 - Real throughput obtained is drastically lower than expected.

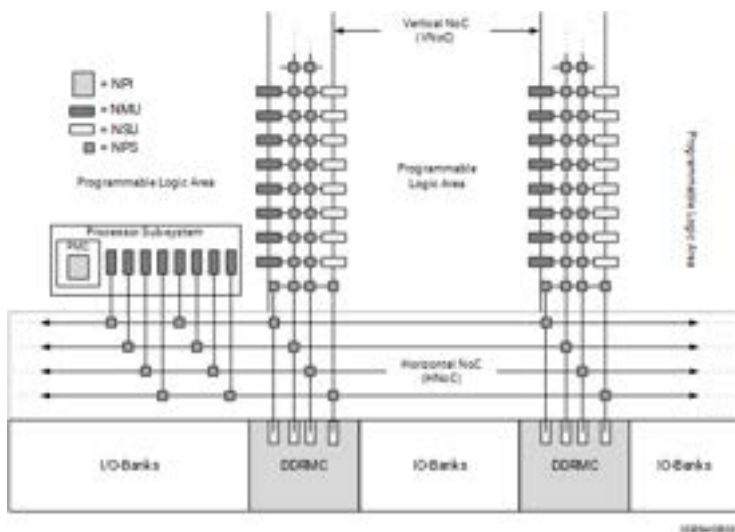
The *Versal Adaptive SoC AI Engine Architecture Manual* (AM009) lists the maximum achievable bandwidth of the various interfaces.

For the PLIOs, it depends on the frequency, bitwidth, and efficiency of the PL kernels. For the GMIOs, it depends on the processor, the DDR memory access pattern, and also on the traffic on the various NoC lanes.

A lower interface throughput does not always mean that the PL or DDR throughput is too low. It might be the graph itself that cannot keep pace of the sample rate, transmitting backpressure to the data source.

The number of DDR interfaces of the device can be: two in the VC1352, and four in the VC1802 and VC1902. The number of DDR chips on the board (DDR4, LPDDR4, ...) can be less than the maximum supported by the device. Furthermore, the platform that is built upon the device mounted on a board might support fewer memories than the number that is actually mounted on the board. All these parameters have a direct impact on the maximum throughput achievable by the system you want to implement.

Figure: NoC Block Diagram



Each DDR interface has four ports which are connected through switches to the four independent lanes of the Horizontal NoC and also to the Vertical NoC just above. The Processor System has also two (APU + RPU) connections on each HNoC lane. The VNoC has only two lanes going up the device, so the traffic coming from the four incoming lanes are combined to go over the two lanes. Each lane has two streams, one in each direction, and each stream has a maximum throughput of 14 GB/s.

On the VCK 190, there are four DDR ports which are connected to the HNoC and to the four VNoCs above. Among these four ports, only three are declared in the base platform provided by AMD, thus limiting the available bandwidth to the AI Engine. There are 16 NMUs/NSUs on the VC1902, each one is capable of 16 GB/s of throughput in each direction. We know that the top HNoC is fed by the four VNoCs, leading to a maximum of 8x14GB/s which is much less than the theoretical 16x16 GB/s potential throughput using the 16 NMUs/NSUs.

You may be interested in the input or output throughput, or the overall throughput. The interface may be the PLIO or the GMIO to access the programmable logic or the DDR or the processing system. In the figure above you can see that multiple DDR interfaces are connected to an Horizontal Network on Chip (HNoC), which is connected to the PS and multiple Vertical NoCs. On the top an HNoC is connected to the different VNoCs on one side and to the AI Engine Array on the other side.

Estimating the throughput (in Hardware) of the design at the AI Engine interfaces are possible using different methods, including:

- Timers
- Events
- Profiling
- Event trace
- NoC profiling (GMIO only)

Throughput Measurement Using Timers

Throughput measurement using timers is one of the easiest way to measure the throughput at the AI Engine Array interface coarsely. The goal is to measure the time interval in between the input data transfer start and end.

A host code is generally decomposed in different parts:

- Open the device.
- Load the `xclbin` (bitstream for the PL and executables for the AIE array).
- Declare the source and destination buffers.
- Run the graph.
- Launch the graph.
- Launch input transfers.
- Launch output transfers.
- Wait for the end of input transfers.
- Wait for the end of output transfers.
- Wait for the end of the graph.
- Free all buffers.

When using timers to evaluate interface throughput, the reset of the timer should be as close as possible to the beginning of the transfer. Also, the timer stop should be as close as the effective end of the transfer. The best ordering of these actions depends on the throughput that you want to estimate.

For input transfers, timer fences should be just before input transfer launch and just after the wait of the end of the input transfers. If graph processing is really fast, you can launch output transfers before input ones.

Declare the source and destination buffers as follows:

- Launch the graph.
- Launch output transfers.
- Reset timer.
- Launch input transfers.
- Wait for the end of input transfers.
- Stop timer.
- Wait for the end of output transfers.

If interested in the overall system throughput, timer fences should be set right before the input transfers and right after waiting for the end of the output transfers:

- Launch the graph.
- Launch output transfers.
- Reset timer.
- Launch input transfers.
- Wait for the end of output transfers.
- Stop timer.
- Wait for the end of output transfers (should be safe to skip this stage).

Once you have the time interval, knowing the amount of data allows you to compute directly the throughput of your system.

Throughput Measurements Using Event APIs

This method is more precise than the previous one as you use the events to automatically count the clock ticks in between the start of the transfers to the time the interface is idle. So, you do not have to precisely position the `start_profiling` and the `stop_profiling` instruction. You only have to set them before the graph run and input data transfer for the start, and after the graph ends for the end. Again, knowing the amount of data that has been transferred allows you to more easily compute the average throughput of the system.

Throughput Measurements using Event Trace

Using event trace for throughput measurements requires that you compile the AI Engine application using AI Engine compiler option `--event-trace=runtime` and run the complete system in hardware using XRT or XSDB flow to extract event trace.

After the trace data are extracted they can be visualized within `vitis -a`. An example of trace data can be seen here:

You can position markers where you want to compute the data throughput. Marker can be precisely positioned on signal edges using icons

← → and a new marker can be added with the icon + at the exact position of the mobile marker.

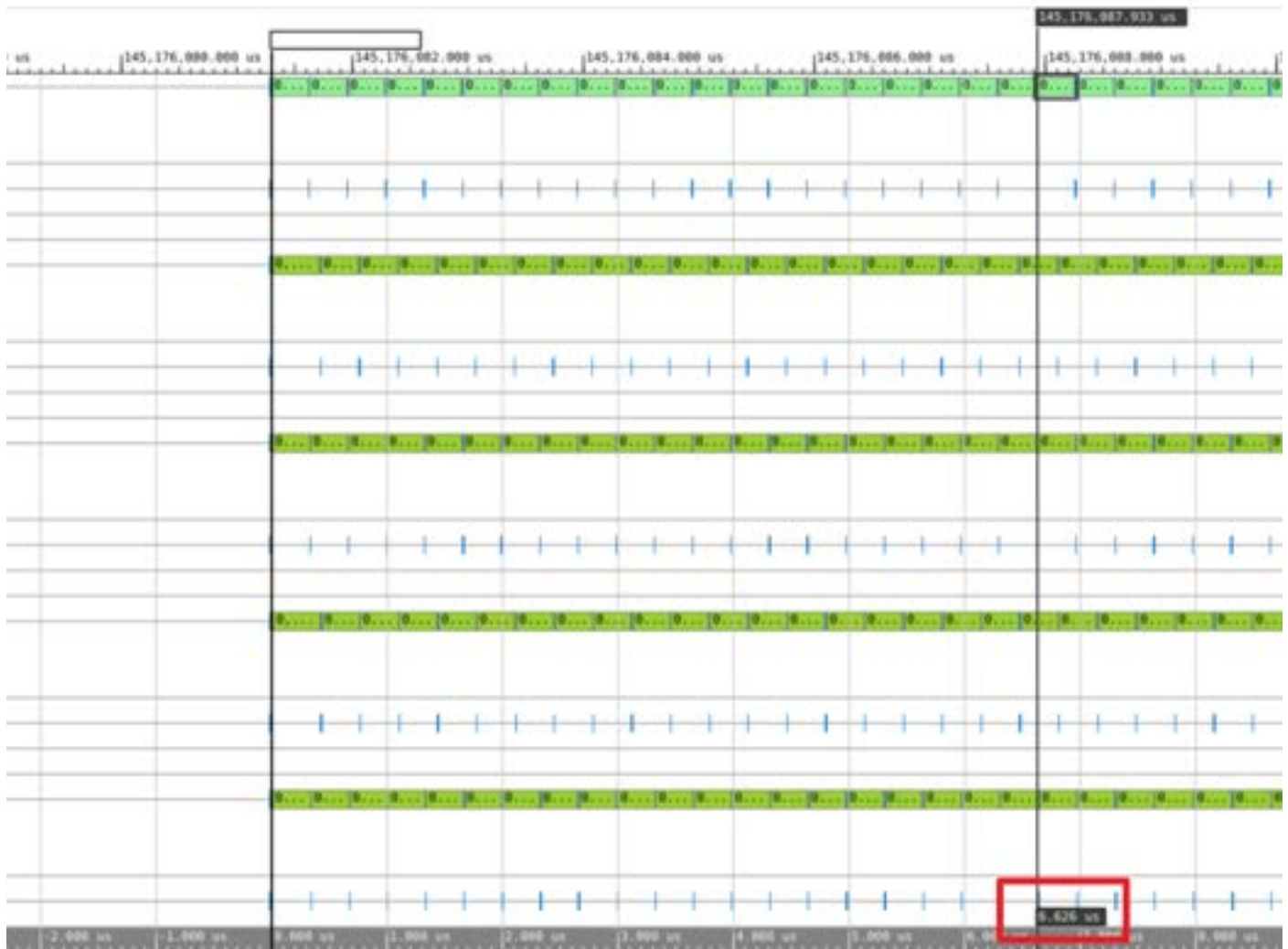
In order to have the most precise result in throughput computation, markers should be positioned at the beginning of the kernel frame run. The added marker should be positioned at the beginning of the first frame you want to take into account, and the mobile one on the first frame after the last one you want to take into account for the throughput computation:

Figure: First Frame and Last Frame Marker Position



Clicking on the flag of the fixed marker which is on the first frame edge will define it as the origin of time, and relative time is displayed on the time scale and at the bottom of all other markers:

Figure: Relative Time Displayed on Time Scale and at Bottom of all Other Markers



Knowing the number of samples processed by each run of the kernel and the number of frames processed allows the user to compute the average throughput on this set of frames with the duration in between the first and last frame processing time.

Single Kernel Programming using Intrinsics

⚠ CAUTION! It is strongly recommended that you use AI Engine APIs for your designs. Usage of intrinsics must only be considered for situations where the stringent performance needs of the design require capabilities that are not covered by the AI Engine API. For example, the

AI Engine API does not currently support functionality provided by some intrinsics such as `fft_data_incr` and `cyclic_add`. While AI Engine APIs support and abstract the main permute use cases, not all permute capabilities are covered. Using intrinsics might allow you to close the performance gap required by your design.

An AI Engine kernel is a C/C++ program which is written using native C/C++ language and specialized intrinsic functions that target the VLIW scalar and vector processors. The AI Engine kernel code is compiled using the AI Engine compiler that is included in the AMD Vitis™ core development kit. The AI Engine compiler compiles the kernels to produce ELF files that are run on the AI Engine processors.

For more information on intrinsic functions, see the *AI Engine Intrinsics User Guide* ([UG1078](#)). AI Engine compiler and simulator are covered in the first few sections of this chapter.

AI Engine supports specialized data types and intrinsic functions for vector programming. By restructuring the scalar application code with these intrinsic functions and vector data types as needed, you can implement the vectorized application code. The AI Engine compiler takes care of mapping intrinsic functions to operations, vector or scalar register allocation and data movement, automatic scheduling, and generation of microcode that is efficiently packed in VLIW instructions.

The following sections introduce the data types supported and registers available for use by the AI Engine kernel. In addition, the vector intrinsic functions that initialize, load, and store, as well as operate on the vector registers using the appropriate data types are also described. To achieve the highest performance on the AI Engine, the primary goal of single kernel programming is to ensure that the usage of the vector processor approaches its theoretical maximum. Vectorization of the algorithm is important, but managing the vector registers, memory access, and software pipelining are also required. The programmer must strive to make the data for the new operation available while the current operation is executing because the vector processor is capable of an operation every clock cycle. Optimizations using software pipelining in loops is available using pragmas. For instance, when the inner loop has sequential or loop carried dependencies it might be possible to unroll an outer loop and compute multiple values in parallel. The following sections go over these concepts as well.

Intrinsics

For intrinsic function documentation, see the *AI Engine Intrinsics User Guide* ([UG1078](#)).

The intrinsic function documentation is organized by the different types of operations and data types. The function calls at the highest level are organized in the documentation as follows:

Load /Store Operations

About pointer dereferencing and pointer arithmetic, as well as operations on streams.

Scalar Operations

Operations on integer and floating-point scalar values, configuration, conversions, addressing, locks and events.

Vector Conversions

Handling of vectors with different sizes and precisions.

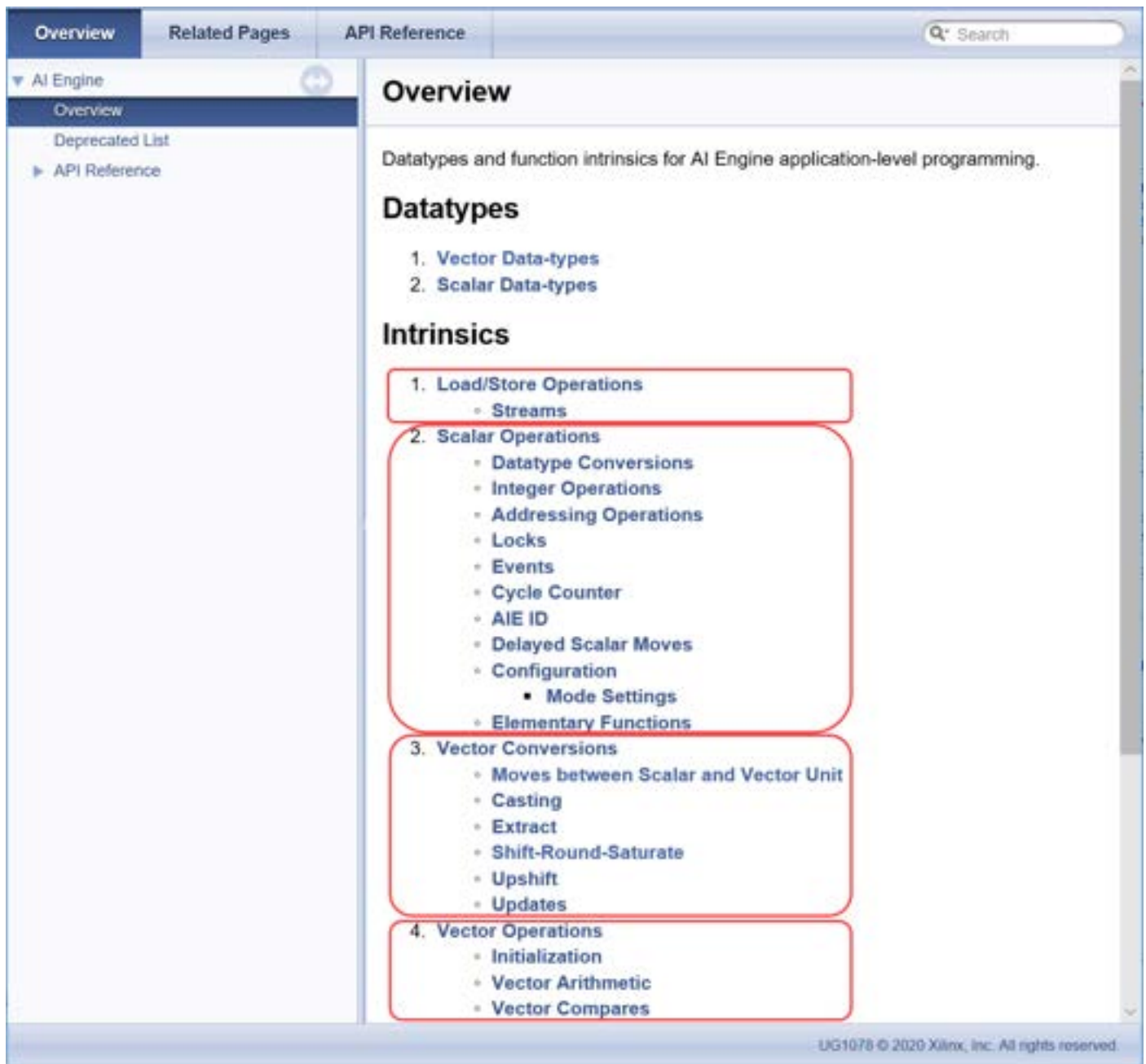
Vector Operations

Arithmetic operations performed on vectors.

Application Specific Intrinsics

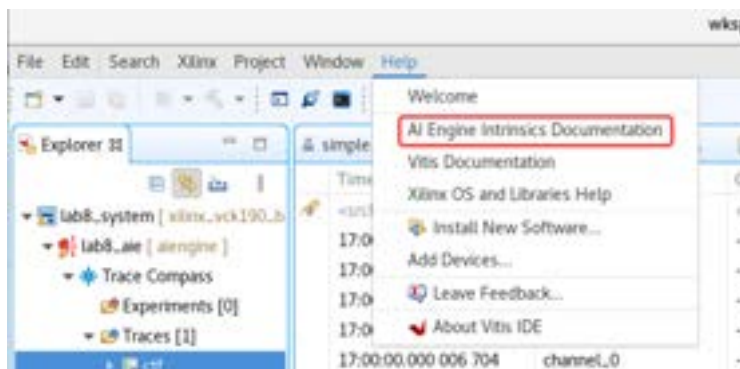
Intrinsics assisting in the implementation of a specific application.

Figure: Intrinsic Function Documentation



The Versal Adaptive SoC AI Engine Intrinsic Documentation is also available through the AMD Vitis™ IDE.

Figure: AI Engine Intrinsic Documentation through the Vitis IDE



Introduction to Scalar and Vector Programming

This section provides an overview of the key elements of kernel programming for scalar and vector processing elements. The details of each element and optimization skills will be seen in following sections.

The following example uses only the scalar engine. It demonstrates a for loop iterating through 512 int32 elements. Each loop iteration

performs a single multiply of int32 a and int32 b storing the result in c and writing it to an output buffer. The scalar_mul kernel operates on two input blocks (buffer) of data input_buffer<int32> and produces an output buffer of data output_buffer<int32>.

Buffers are accessed through scalar and vector iterators. For additional details on the buffer APIs, see [Streaming Data API](#).

```
void scalar_mul(input_buffer<int32> & data1,
               input_buffer<int32> & data2,
               output_buffer<int32> & out){
    auto pin1 = aie::begin(data1);
    auto pin2 = aie::begin(data2);
    auto pout = aie::begin(out);
    for(int i=0; i<512; i++)
    {
        int32 a=*pin1++;
        int32 b=*pin2++;
        int32 c=a*b;
        *pout++ = c;
    }
}
```

The following example is a vectorized version for the same kernel.

```
void vect_mul(input_buffer<int32> & __restrict data1,
             input_buffer<int32> & __restrict data2,
             output_buffer<int32> & __restrict out){
    auto pin1 = aie::begin_vector<8>(data1);
    auto pin2 = aie::begin_vector<8>(data2);
    auto pout = aie::begin_vector<8>(out);
    for(int i=0; i<64; i++)
        chess_prepare_for_pipelining
        {
            aie::vector<int32, 8> va=*pin1++;
            aie::vector<int32, 8> vb=*pin2++;
            aie::accum<acc80, 8> vt=mul(va, vb);
            aie::vector<int32, 8> vc=srs(vt, 0);
            *pout++ = vc;
        }
}
```

Note the data types vector<int32, 8> and accum<acc80, 8> are used in the previous kernel code. The buffer API begin_vector<8> returns an iterator that will iterate over vectors of 8 int32s and stores them in variables named va and vb. These two variables are vector type variables and they are passed to the intrinsic function mul which outputs vt which is a accum<acc80, 8> data type. The accum<acc80, 8> type is reduced by a *shift round saturate* function srs that allows a vector<int32, 8> type, variable vc, to be returned and then written to the output buffer. Additional details on the data types supported by the AI Engine are covered in the following sections.

The __restrict keyword used on the input and output parameters of the vect_mul function, allows for more aggressive compiler optimization by explicitly stating independence between data.

chess_prepare_for_pipelining is a compiler pragma that directs kernel compiler to achieve optimized pipeline for the loop.

The scalar version of this example function takes 1055 cycles while the vectorized version takes only 99 cycles. As you can see there is more than 10 times speedup for vectorized version of the kernel. Vector processing itself would give 8x the throughput for int32 multiplication but has a higher latency and would not get 8x the throughput overall. However, with the loop optimizations done, it can get close to 10x. The sections that follow describe in detail the various data types that can be used, registers available, and also the kinds of optimizations that can be achieved on the AI Engine using concepts like software pipelining in loops and keywords like __restrict.

AI Engine Data Types

The AI Engine scalar unit supports signed and unsigned integers in 8, 16, and 32-bit widths, along with some single-precision floating-point for specific operations.

The AI Engine vector unit supports integers and complex integers in 8, 16, and 32-bit widths, along with real and complex single-precision floating-point numbers. It also supports accumulator vector data types, with 48 and 80-bit wide elements. Intrinsic functions such as absolute value, addition, subtraction, comparison, multiplication, and MAC operate using these vector data types. Vector data types are named using a convention that includes the number of elements, real or complex, vector type or accumulator type, and bit width as follows:

```
aie::vector<[c]([u]int|float)(SizeofElement), NumLanes>
aie::accum<[c](acc|accfloat)(SizeofElements), NumLanes>
```

Optional specifications include:

NumLanes

Denotes the number of elements in the vector which can be 2, 4, 8, 16, 32, 64, or 128.

c

Denotes complex data with real and imaginary parts packed together.

int

denotes integer vector data values.

float

Denotes single precision floating-point values.

acc

Denotes accumulator vector data values.

u

Denotes unsigned. Unsigned only exists for int8 vectors.

SizeofElement

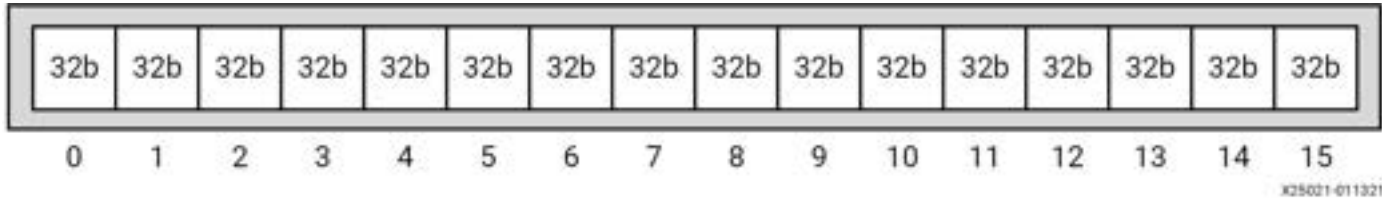
Denotes the size of the vector data type element.

- 1024-bit integer vector types are vectors of 8-bit, 16-bit, or 32-bit vector elements. These vectors have 16, 32, 64, or 128 lanes.
- 512-bit integer vector types are vectors of 8-bit, 16-bit, 32-bit, or 64-bit vector elements. These vectors have 4, 8, 16, 32, or 64 lanes.
- 256-bit integer vector types are vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit vector elements. These vectors have 1, 2, 4, 8, 16, or 32 lanes.
- 128-bit integer vector types are vectors of 8-bit, 16-bit, or 32-bit vector elements. These vectors have 2, 4, 8, or 16 lanes.
- Accumulator data types are vectors of 80-bit or 48-bit elements These vectors have 2, 4, 8, or 16 lanes.

The total data-width of the vector data-types can be 128-bit, 256-bit, 512-bit, or 1024-bit. The total data-width of the accumulator data-types can be 320/384-bit or 640/768-bit.

For example, v16int32 is a sixteen element vector of integers with 32 bits. Each element of the vector is referred to as a *lane*. Using the smallest bit width necessary can improve performance by making good use of registers.

Figure: aie::vector<int32,16>



Vector Registers

All vector intrinsic functions require the operands to be present in the AI Engine vector registers. The following table shows the set of vector registers and how smaller registers are combined to form large registers.

Table: Vector Registers

128-bit	256-bit	512-bit	1024-bit	
vrl0	wr0	xa	ya	N/A
vrh0				
vrl1	wr1			
vrh1				
vrl2	wr2	xb		yd (msbs)
vrh2				
vrl3	wr3			
vrh3				

128-bit	256-bit	512-bit	1024-bit	
vcl0	wc0	xc	N/A	N/A
vch0				
vcl1	wc1			
vch1				
vdI0	wd0	xd	N/A	yd (lsbs)
vdh0				
vdI1	wd1			
vdh1				

The underlying basic hardware registers are 128-bit wide and prefixed with the letter v. Two v registers can be grouped to form a 256-bit register prefixed with w. wr, wc, and wd registers are grouped in pairs to form 512-bit registers (xa, xb, xc, and xd). xa and xb form the 1024-bit wide ya register, while xd and xb form the 1024-bit wide yd register. This means the xb register is shared between ya and yd registers. xb contains the most significant bits (MSBs) for both ya and yd registers.

The vector register name can be used with the `chess_storage` directive to force vector data to be stored in a particular vector register. For example:

```
aie::vector<int32,8> chess_storage(wr0) bufA;
aie::vector<int32,8> chess_storage(WR) bufB;
```

When upper case is used in the `chess_storage` directive, it means register files (for example, any of the four wr registers), whereas lower case in the directive means just a particular register (for example, wr0 in the previous code example) will be chosen.

Vector registers are a valuable resource. If the compiler runs out of available vector registers during code generation, then it generates code to spill the register contents into local memory and read the contents back when needed. This consumes extra clock cycles.

The name of the vector register used by the kernel during its execution is shown for vector load/store and other vector-based instructions in the kernel microcode. This microcode is available in the disassembly view in Vitis IDE. For additional details on Vitis IDE usage, see [Using Vitis Unified IDE and Reports](#).

Many intrinsic functions only accept specific vector data types but sometimes not all values from the vector are required. For example, certain intrinsic functions only accept 512-bit vectors. If the kernel code has smaller sized data, one technique that can help is to use the `concat()` intrinsic to concatenate this smaller sized data with an undefined vector (a vector with its type defined, but not initialized).

For example, the `lmul8` intrinsic only accepts a `v16int32` or `v32int32` vector for its `xbuff` parameter. The intrinsic prototype is:

```
v8acc80 lmul8 (      aie::vector<int32,16>  xbuff,
                  int      xstart,
                  unsigned int  xoffsets,
                  aie::vector<int32,8>      zbuff,
                  int      zstart,
                  unsigned int  zoffsets
                )
```

The `xbuff` parameter expects a 16 element vector (`v16int32`). In the following example, there is an eight element vector (`aie::vector<int32,8>`) `rva`. The `concat()` intrinsic is used to upgrade it to a 16 element vector. After concatenation, the lower half of the 16 element vector has the contents of `rva`. The upper half of the 16 element vector is uninitialized due to concatenation with the undefined `aie::vector<int32,8>` vector.

```
int32 a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
aie::vector<int32,8> rva = *((aie::vector<int32,8>*)a);
acc = lmul8(concat(rva,aie::zeros<int32,8>()),0,0x76543210,rvb,0,0x76543210);
```

For more information about how vector-based intrinsic functions work, refer to [Vector Register Lane Permutations](#).

Accumulator Registers

The accumulation registers are 384 bits wide and can be viewed as eight vector lanes of 48 bits each. The idea is to have 32-bit multiplication results and accumulate over those results without bit overflows. The 16 guard bits allow up to 2^{16} accumulations. The output of fixed-point vector MAC and MUL intrinsic functions is stored in the accumulator registers. The following table shows the set of accumulator registers and

how smaller registers are combined to form large registers.

Table: Accumulator Registers

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

The accumulator registers are prefixed with the letters 'am'. Two of them are aliased to form a 768-bit register that is prefixed with 'bm'. The shift-round-saturate `srs()` intrinsic is used to move a value from an accumulator register to a vector register with any required shifting and rounding.

```
aie::vector<int32,8> res = srs(acc, 8); // shift right 8 bits, from accumulator register to vector register
```

The upshift `ups()` intrinsic is used to move a value from an vector register to an accumulator register with upshifting:

```
aie::accum<acc48,8> acc = ups(v, 8); //shift left 8 bits, from vector register to accumulator register
```

The `set_rnd()` and `set_sat()` intrinsics are used to set the rounding and saturation mode of the accumulation result, while `clr_rnd()` and `clr_sat()` intrinsics are used to clear the rounding and saturation mode, that is to truncate the accumulation result.

Note that only when operations are going through the shift-round-saturate data path, the shifting, rounding, or saturation mode will be effective. Some intrinsics only use the vector pre-adder operations, where there will be no shifting, rounding, or saturation mode for configuration. Such operations are adds, subs, abs, vector compares, or vector selections/shuffles. It is possible to choose MAC intrinsics instead to do subtraction with shifting, rounding, or saturation mode configuration. The following code performs subtraction between `va` and `vb` with `mul` instead of `sub` intrinsics.

```
v16cint16 va, vb;
int32 zbuff[8]={1,1,1,1,1,1,1,1};
aie::vector<int32,8> coeff=(v8int32*)zbuff;
aie::accum<acc48,8> acc = mul8_antisym(va, 0, 0x76543210, vb, 0, false, coeff, 0, 0x76543210);
aie::vector<int32,8> res = srs(acc,0);
```

Floating-point intrinsic functions do not have separate accumulation registers and instead return their results in a vector register. The following streaming data APIs can be used to read and write floating-point accumulator data from or to the cascade stream.

```
aie::vector<float,8> readincr_v<8>(input_cascade<accfloat> * str);
aie::vector<cfloat,4> readincr_v4(input_cascade<caccfloat> * str);
void writeincr_v8(output_cascade<accfloat>* str, v8float value);
void writeincr(output_cascade<caccfloat>* str, v4cfloat value);
```

For more information about the window and streaming data APIs, refer to [Input and Output Buffers](#)

The data size in memory is aligned to the next power of 2 (here 64b for `acc48`), hence it is best to use `sizeof` to determine the position on the elements. The following code is an example to print accumulator vector registers.

```
aie::accum<acc48,8> vacc; //cascade value
const int SIZE_ACC48=sizeof(aie::accum<acc48,8>)/8;
for(int i=0;i<8;i++){//8 number
    int8 *p=(int8*)&vacc+SIZE_ACC48*i;//point to start of each acc48
    printf("acc value[%d]=0x",i);
    for(int j=5;j>=0;j--){//print each acc48 from higher byte to lower byte
        printf("%02x",*(p+j));
    }
}
```


```
printf("\n");
}
```

The output is as follows.

```
acc value[0]=0x000000000000
acc value[1]=0x000000000001
acc value[2]=0x000000000002
acc value[3]=0x000000000003
acc value[4]=0x000000000004
acc value[5]=0x000000000005
acc value[6]=0x000000000006
acc value[7]=0x000000000007
```

Casting and Datatype Conversion


Casting intrinsic functions (`as_[Type]()`) allow casting between vector types or scalar types of the same size. The casting can work on accumulator vector types too. Generally, using the smallest data type possible will reduce register spillage and improve performance. For example, if a 48-bit accumulator (`acc48`) can meet the design requirements then it is preferable to use that instead of a larger 80-bit accumulator (`acc80`).

 **Note:** The `acc80` vector data type occupies two neighboring 48-bit lanes.

Standard C casts can be also used and works identically in almost all cases as shown in the following example.

```
aie::vector<int16,8> iv;
aie::vector<cint16,4> cv=as_v4cint16(iv);
aie::vector<cint16,4> cv2=*(aie::vector<cint16,4>*)&iv;
aie::accum<acc80,8> cas_iv;
aie::accum<acc48,8> cas_cv=as_v8cacc48(cas_iv);
```

There is hardware support built-in for floating-point to fixed-point (`float2fix()`) and fixed-point to floating-point (`fix2float()`) conversions. For example, the fixed-point square root, inverse square root, and inverse are implemented with floating-point precision and the `fix2float()` and `float2fix()` conversions are used before and after the function.

 **Note:** The AI Engine floating-point is not completely compliant with the IEEE standards. For more information about the exceptions, see *Versal Adaptive SoC AI Engine Architecture Manual* ([AM009](#)).

The scalar engine is used in this example because the square root and inverse functions are not vectorizable. This can be verified by looking at the function prototype's input data types:


```
float _sqrtf(float a) //scalar float operation
int sqrt(int a,...) //scalar integer operation
```

Note that the input data types are scalar types (`int`) and not vector types (`vint`).

The conversion functions (`fix2float`, `float2fix`) can be handled by either the vector or scalar engines depending on the function called.

Note the difference in data return type and data argument types:

```
float fix2float(int n,...) //Uses the scalar engine
v8float fix2float(aie::vector<int32,8> ivec,...) //Uses the vector engine
```

 **Note:** For `float2fix`, there are two types of implementations, `float2fix_safe` (default) and `float2fix_fast` with the `float2fix_safe` implementation offering a more strict data type check. You can define the macro `FLOAT2FIX_FAST` to make `float2fix` choose the `float2fix_fast` implementation.

Vector Initialization, Load, and Store

Vector registers can be initialized, loaded, and saved in a variety of ways. For optimal performance, it is critical that the local memory that is used to load or save the vector registers be aligned on 16-byte boundaries.

Alignment

The `alignas` standard C specifier can be used to ensure proper alignment of local memory. In the following example, the `reals` is aligned to 16 byte boundary.

```
alignas(16) const int32 reals[8] =
```

```
{32767, 23170, 0, -23170, -32768, -23170, 0, 23170};
//align to 16 bytes boundary, equivalent to "alignas(v4int32)"
```

Initialization

The following functions can be used to initialize vector registers as undefined, all 0's, with data from local memory, or with part of the values set from another register and the remaining part are undefined. Initialization using the `undef_type()` initializer ensures that the compiler can optimize regardless of the undefined parts of the value.

```
v8int32 v;
v8int32 uv = undef_v8int32(); //undefined
v8int32 nv = null_v8int32(); //all 0's
v8int32 iv = *(v8int32 *) reals; //re-interpret "reals" as "v8int32" pointer and load value from it
v16int32 sv = xset_w(0, iv); //create a new 512-bit vector with lower 256-bit set with "iv"
```

In the previous example, vector set intrinsic functions `[T] set_[R]` allow creating a vector where only one part is initialized and the other parts are undefined. Here `[T]` indicates the target vector register to be set, `w` for W register (256-bit), `x` for X register (512-bit), and `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), and `x` for X register (512-bit). Note that `[R]` width is smaller than `[T]` width. The valid vector set intrinsic functions are, `wset_v`, `xset_v`, `xset_w`, `yset_v`, `yset_w`, and `yset_x`.

The `static` keyword applies to the vector data type as well. The default value is zero when not initialized and the value is kept between graph run iterations.

Load and Store

Load and Store from Vector Registers

The compiler supports standard pointer de-referencing and pointer arithmetic for vectors. Post increment of the pointer is the most efficient form for scheduling. No special intrinsic functions are needed to load vector registers.

```
v8int32 * ptr_coeff_buffer = (v8int32 *)ptr_kernel_coeff;
v8int32 kernel_vec0 = *ptr_coeff_buffer++; // 1st 8 values (0 .. 7)
v8int32 kernel_vec1 = *ptr_coeff_buffer;    // 2nd 8 values (8 .. 15)
```

Load and Store From Memory

AI Engine APIs provide access methods to read and write data from data memory, streaming data ports, and cascade streaming ports which can be used by AI Engine kernels. For additional details on streaming APIs, see [Streaming Data API](#). In the following example, the `window_readincr_v8(din)` API is used to read a window of complex int16 data into the data vector. Similarly, `readincr_v8(cin)` is used to read a sample of int16 data from the `cin` stream. `writeincr(cas_out, v)` is used to write data to a cascade stream output.

```
void func(input_window_cint16 *din,
          input_stream_int16 *cin,
          output_cascade<cacc48> *cas_out){
    v8cint16 data=window_readincr_v8(din);
    v8int16 coef=readincr_v8(cin);
    v4cacc48 v;
    ...
    writeincr(cas_out, v);
}
```

Load and Store Using Pointers

It is mandatory to use the window API in the kernel function prototype as inputs and outputs. However, in the kernel code, it is possible to use a direct pointer reference to read/write data.

```

void func(input_window_int16 *w_input,
          output_window_cint16 *w_output){
    .....
    v16int16 *ptr_in  = (v16int16 *)w_input->ptr;
    v8cint16 *ptr_out = (v8cint16 *)w_output->ptr;
    .....
}

```

The window structure is responsible for managing buffer locks tracking buffer type (ping/pong) and this can add to the cycle count. This is especially true when load/store are out-of-order (scatter-gather). Using pointers can help reduce the cycle count required for load and store.

 **Note:** If using pointers to load and store data, it is the designer's responsibility to avoid out-of-bound memory access.

Load and Store Using Streams

Vector data can also be loaded from or stored in streams as shown in the following example.

```

void func(input_stream_int32 *s0, input_stream_int32 *s1, ...){
    for(...){
        data0=readincr(s0);
        data1=readincr(s1);
        ...
    }
}

```

For more information about streaming data APIs, see [Streaming Data API](#).

Load and Store with Virtual Resource Annotations

AI Engine is able to perform several vector load or store operations per cycle. However, in order for the load or store operations to be executed in parallel, they must target different memory banks. In general, the compiler tries to schedule many memory accesses in the same cycle when possible, but there are some exceptions. Memory accesses coming from the same pointer are scheduled on different cycles. If the compiler schedules the operations on multiple variables or pointers in the same cycle, memory bank conflicts can occur.

To avoid concurrent access to a memory with multiple variables or pointers, the compiler provides the following `aie_dm_resource` annotations to annotate different virtual resources. Accesses using types that are associated with the same virtual resource are not scheduled to access the resource at the same cycle.

```

__aie_dm_resource_a
__aie_dm_resource_b
__aie_dm_resource_c
__aie_dm_resource_d
__aie_dm_resource_stack

```

For example, the following code is to annotate two arrays to the same `__aie_dm_resource_a` that guides the compiler to not access them in the same instruction.

```

v8int32 va[32];
v8int32 vb[32];
v8int32 __aie_dm_resource_a* restrict p_va = (v8int32 __aie_dm_resource_a*)va;
v8int32 __aie_dm_resource_a* restrict p_vb = (v8int32 __aie_dm_resource_a*)vb;
//access va, vb by p_va, p_vb
v8int32 vc;
vc=p_va[i]+p_vb[i];

```

The following code is to annotate an array and a window buffer to the same `__aie_dm_resource_a` that guides the compiler to not access them in the same instruction.


```

void func(input_window_int32 * __restrict wa, .....
    v8int32 coeff[32];
    input_window_int32 sa;
    v8int32 __aie_dm_resource_a* restrict p_coeff = (v8int32 __aie_dm_resource_a*)coeff;
    input_window_int32 __aie_dm_resource_a* restrict p_wa = (input_window_int32 __aie_dm_resource_a*)&sa;
    window_copy(p_wa, wa);
    v8int32 va;
    va=window_readincr_v8(p_wa);//access wa by p_wa

```

Update, Extract, and Shift

To update portions of vector registers, the `upd_v()`, `upd_w()`, and `upd_x()` intrinsic functions are provided for 128-bit (v), 256-bit (w), and 512-bit (x) updates.

Note: The updates overwrite a portion of the larger vector with the new data while keeping the other part of the vector alive. This alive state of the larger vector persists through multiple updates. If too many vectors are kept unnecessarily alive, register spillage can occur and impact performance.

Similarly, `ext_v()`, `ext_w()`, and `ext_x()` intrinsic functions are provided to extract portions of the vector.

To update or extract individual elements, the `upd_elem()` and `ext_elem()` intrinsic functions are provided. These must be used when loading or storing values that are not in contiguous memory locations and require multiple clock cycles to load or store a vector. In the following example, the 0th element of vector `v1` is updated with the value of `a` - which is 100.

```

int a = 100;
v4int32 v1 = upd_elem(undef_v4int32(), 0, a);

```

Another important use is to move data to the scalar unit and do an inverse or sqrt. In the following example, the 0th element of vector `vf` is extracted and stored in the scalar variable `f`.

```

v4float vf;
float f=ext_elem(vf,0);
float i_f=invsqrt(f);

```

The `shft_elem()` intrinsic function can be used to update a vector by inserting a new element at the beginning of a vector and shifting the other elements by one.

Accumulator registers can be updated from vector registers by `ups` intrinsic function. And accumulator registers can also be half updated by `upd_hi` and `upd_lo` intrinsic functions.

```

//From v16int32 to v16acc48
v16int32 v;
v16acc48 acc = upd_lo(acc, ups(ext_w(v, 0), 0));
acc = upd_hi(acc, ups(ext_w(v, 1), 0));

```

Vector Register Lane Permutations

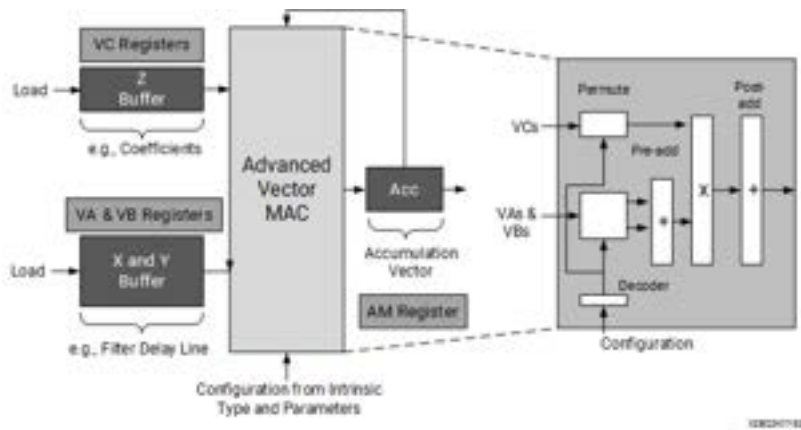
The AI Engine fixed-point vector units datapath consists of the following three separate and largely independently usable paths:

- Main MAC datapath
- Shift-round-saturate path
- Upshift path

The main multiplication path reads values from vector registers, permutes them in a user controllable fashion, performs optional pre-adding, multiplies them, and after some post-adding accumulates them to the previous value of the accumulator register.

While the main datapath stores to the accumulator, the shift-round-saturate path reads from the accumulator registers and stores to the vector registers or the data memory. In parallel to the main datapath runs the upshift path. It does not perform any multiplications but simply reads vectors, upshifts them and feeds the result into the accumulators. For details on the fixed-point and floating-point data paths refer to *Versal Adaptive SoC AI Engine Architecture Manual* (AM009). Details on the intrinsic functions that can be used to exercise these data paths can be found in the *AI Engine Intrinsics User Guide* (UG1078).

As shown in the following figure, the basic functionality of MAC data path consists of vector multiply and accumulate operations between data from the X and Z buffers. Other parameters and options allow flexible data selection within the vectors and number of output lanes and optional features allow different input data sizes and pre-adding. There is an additional input buffer, the Y buffer, whose values can be pre-added with those from the X buffer before the multiplication occurs. The result from the intrinsic is added to an accumulator.

Figure: Functional Overview of the MAC Data Path

The operation can be described using *lanes* and *columns*. The number of lanes corresponds to the number of output values that will be generated from the intrinsic call. The number of columns is the number of multiplications that will be performed per output lane, with each of the multiplication results being added together. For example:

```
acc0 += z00*(x00+y00) + z01*(x01+y01) + z02*(x02+y02) + z03*(x03+y03)
acc1 += z10*(x10+y10) + z11*(x11+y11) + z12*(x12+y12) + z13*(x13+y13)
acc2 += z20*(x20+y20) + z21*(x21+y21) + z22*(x22+y22) + z23*(x23+y23)
acc3 += z30*(x30+y30) + z31*(x31+y31) + z32*(x32+y32) + z33*(x33+y33)
```

In this case, four outputs are being generated, so there are four lanes and four columns for each of the outputs with pre-addition from the X and Y buffers.

The parameters of the intrinsics allow for flexible data selection from the different input buffers for each lane and column, all following the same pattern of parameters. The following section introduces the data selection (or data permute) schemes with detailed examples that include shuffle and select intrinsics. Details around the mac intrinsic and its variants are also discussed in the following sections.

Data Selection

AI Engine intrinsic functions support various types of data selection. The details around the shuffle and select intrinsic are as follows.

Data Shuffle

The AI Engine shuffle intrinsic function selects data from a single input data buffer according to the start and offset parameters. This allows for flexible permutations of the input vector values without needing to rearrange the values. `xbuff` is the input data buffer, with `xstart` indicating the starting position offset for each lane in the `xbuff` data buffer and `xoffset` indicating the position offset applied to the data buffer. The shuffle intrinsic function is available in 8, 16, and 32 lane variants (`shuffle8`, `shuffle16`, and `shuffle32`). The main permute for data (`xoffsets`) is at 32-bit granularity and `xsquare` allows a further 16-bit granularity mini permute after main permute. Thus, the 8-bit and 16-bit vector intrinsic functions can have additional square parameter- for more complex permutations.

For example, a `shuffle16` intrinsic has the following function prototype.

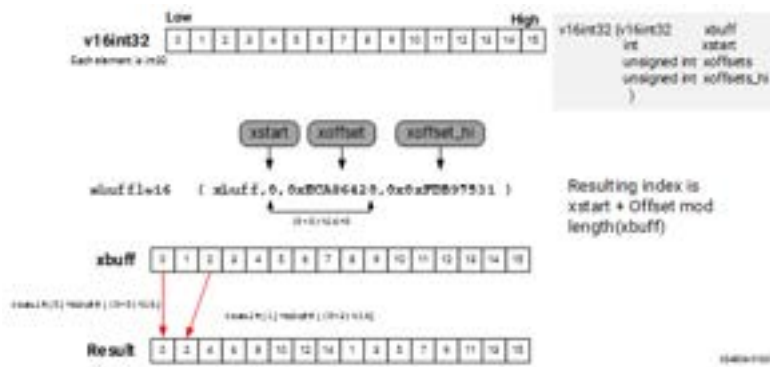
```
v16int32 shuffle16 (          v16int32 xbuff,
                    int xstart,
                    unsigned int xoffsets,
                    unsigned int xoffsets_hi
                    )
```

The data permute performs in 32 bits granularity. When the data size is 32 bits or 64 bits, the start and offsets are relative to the full data width, 32 bits or 64 bits. The lane selection follows the regular lane selection scheme.

```
f: result [lane number] = (xstart + xbuff [lane number]) Mod input_samples
```

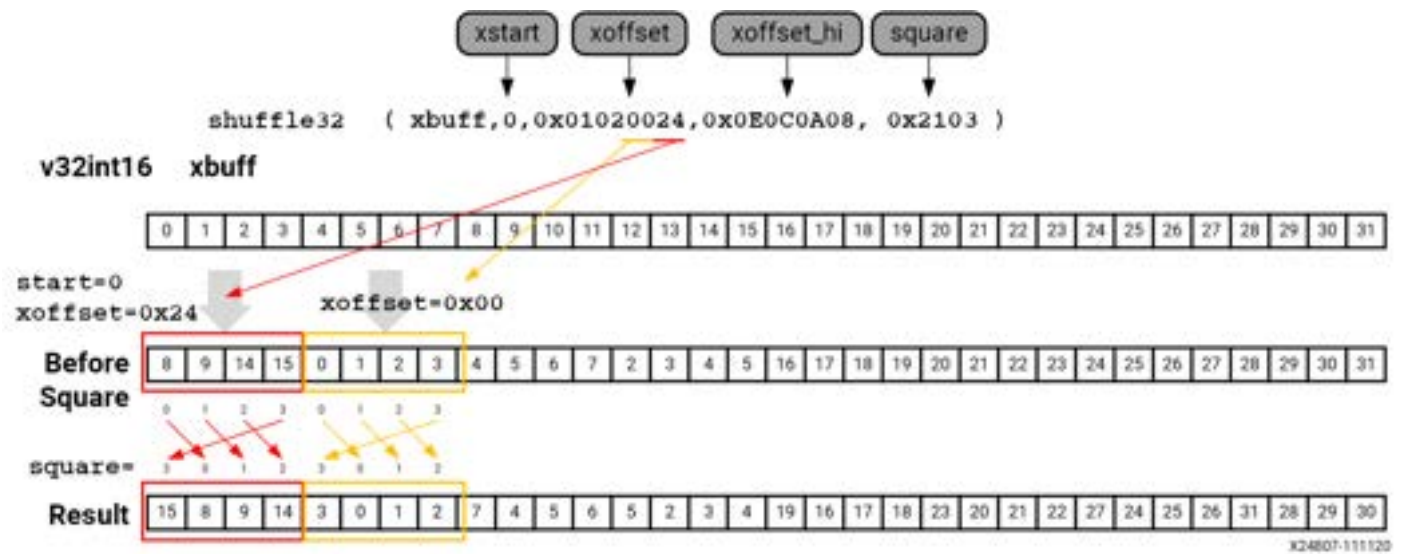
The following example shows how shuffle works on the `v16int32` vector. `xoffset` and `xoffset_hi` have 4 bits for each lane. This example moves the even and odd elements of the buffer into lower and higher parts of the buffer.

Figure: Data Shuffle on int32 Type



When data permute is on 16 bits data, the intrinsic function includes another parameter, xsquare, allowing flexibility to perform data selection in each 4 x 16 bits block of data. The xoffset comes in pairs. The first hex value is an absolute 32 bits offset and picks up 2 x 16 bits values (index, index+1). The second hex value is offset from first value + 1 (32 bits offset) and picks up 2 x 16 bits values. For example, 0x00 selects index 0, 1, and index 2, 3. 0x24 selects index 8, 9, and index 14, 15. Following is a shuffle example on the v32int16 vector.

Figure: Data Shuffle on int16 Type



Data Select

The select intrinsic selects between the first set of lanes or the second one according to the value of the select parameter. If the lane corresponding bit in select is zero, it returns the value in the first set of lanes. If the bit is one, it returns the value in the second set of lanes. For example, a select16 intrinsic function has the following function prototype.

```
v16int32 select16 (
    v16int32 xbuff,
    int xstart,
    unsigned int xoffsets,
    unsigned int xoffsets_hi,
    v16int32 ybuff,
    int ystart,
    unsigned int yoffsets,
    unsigned int yoffsets_hi
)
```

For each bit of select (from low to high), it will select a lane either from xbuff (if the select parameter bit is 0) or from ybuff (if the select parameter bit is 1). Data permute on the resulting lane of xbuff or ybuff is achieved by a shuffle with corresponding bits in xoffsets or yoffsets. Following is the pseudo C-style code for select.

```
for (int i = 0; i < 16; i++){
    idx = f( xstart, xoffsets[i]); //i'th 4 bits of offsets
    idy = f( ystart, yoffsets[i]);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

For information about how `f` works in previous code, refer to the regular lane selection scheme equation listed at the beginning of this section. When working on the `int16` data type, the `select` intrinsic has an additional `xsquare` parameter which allows a further 16-bit granularity mini permute after main permute. For example, a `select32` intrinsic function has the following function prototype.

```
v32int16 select32      (unsigned int select,
                        v64int16 xbuff,
                        int xstart,
                        unsigned int xoffsets,
                        unsigned int xoffsets_hi,
                        unsigned int xsquare,
                        int ystart,
                        unsigned int yoffsets,
                        unsigned int yoffsets_hi,
                        unsigned int ysquare
                        )
```

Following is the pseudo C-style code for `select`.

```
for (int i = 0; i < 32; i++){
    idx = f( xstart, xoffsets[i], xsquare);
    idy = f( ystart, yoffsets[i], ysquare);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

The following example uses `select32` to interleave first 16 elements of A and B (A first).

```
int16 A[32]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31};
int16 B[32]={32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
             48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63};
v32int16 *pA=(v32int16*)A;
v32int16 *pB=(v32int16*)B;
v32int16 C = select32(0xAAAAAAAA, concat(*pA,*pB),
                     0, 0x03020100, 0x07060504, 0x1100,
                     32, 0x03020100, 0x07060504, 0x1100);
```

The output C for the previous code is as follows.

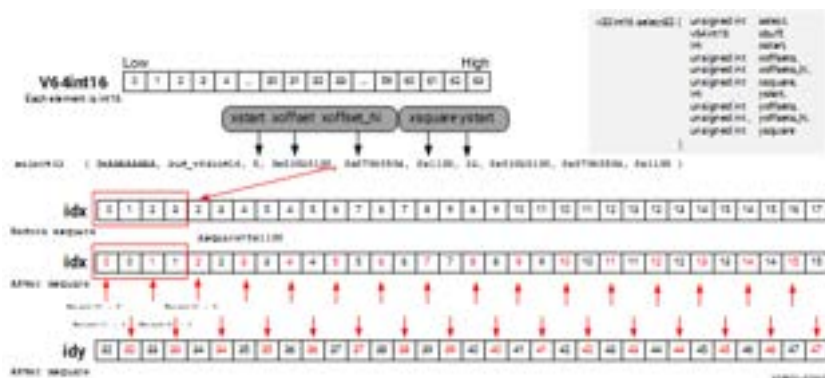
```
{0,32,1,33,2,34,3,35,4,36,5,37,6,38,7,39,8,40,9,41,10,42,11,43,12,44,13,45,14,46,15,47
}
```

This can also be done using the `shuffle32` intrinsic.

```
v32int16 C = shuffle32(concat(*pA,*pB),
                       0, 0xF3F2F1F0, 0xF7F6F5F4, 0x3120);
```

The following figure shows how the previous `select32` intrinsic works.

Figure: Data Select on int16 Type



MAC Ininsics

MAC intrinsics perform vector multiply and accumulate operations between data from two buffers, the X and Z buffers, with the other parameters and options allowing flexibility (data selection within the vectors, number of output lanes) and optional features (different input data sizes, pre-adding, etc). There is an additional input buffer, the Y buffer, whose values can be pre-added with those from the X buffer before the multiplication occurs. The result from the intrinsic is added to an accumulator.

The parameters of the intrinsics allow for flexible data selection from the different input buffers for each lane and column, all following the same pattern of parameters. A starting point in the buffer is given by the (x/y/z) start parameter which selects the first element for the first row as well as first column. To allow flexibility for each lane, (x/y/z) offsets provides an offset value for each lane that will be added to the starting point. Finally, the (x/y/z) step parameter defines the step in data selection between each column based on the previous position. It is worth noticing that when the ystep is not specified in the intrinsic it will be the symmetric of the xstep.

Main permute granularity for x/y and z buffers is 32 bits and 16 bits, respectively. Complex numbers are considered as one entity for the permute (for example, cint16 as 32 bits for permute). Parameter zstart must be a compile time constant. 8-bit and 16-bit permute granularity in x/y and 8-bit permute granularity in z have certain limitations as addressed towards the end of this section. The following sections covers the different data widths and explains the result of the MAC intrinsic on these data widths.

MAC on 32x32 bits

The following figure shows how start, offsets, and step work on the cint16 data type.

Figure: MAC4 on cint16 x cint16 Type



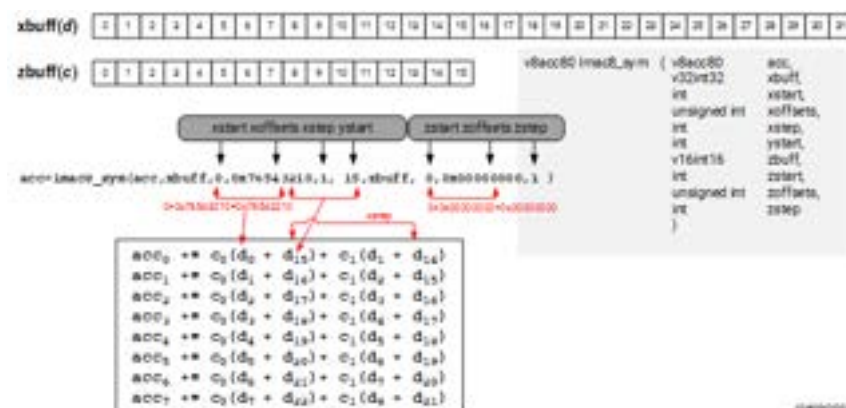
mac4 has four output lanes. The first column of data is selected by adding xstart to every 4 bits of xoffsets. The subsequent column of data is selected by adding xstep to its previous column.

The coefficients of mac4 are chosen similarly by zstart, zoffset, and zstep.

MAC on 32x16 bits

An example of MAC with pre-adding is as follows. With pre-adding, the data from X buffer can be added by itself, or the data from X buffer and Y buffer can be added. The start, offsets, and step parameters work similar as previous example. There is a ystart parameter for Y buffer or another data from X buffer. The step parameter works reversely for Y or another data from X buffer.

Figure: LMAC8_SYM on int32 x int16 Type



MAC on 16x16 bits

An example of MAC with int16 X buffer and int16 Z buffer is as follows. Note that the permute granularity for X buffer is 32 bits. The start and step parameters are always in terms of data type granularity. To get to a 16-bit index, you need to multiply them by 2.

The xoffsets parameter comes as a pair. The first hex value is an absolute 32 bits offset and picks up 2 x 16 bits values (index, index+1) in the even row. The second hex value is offset from first hex value plus 1 (32 bits offset) and picks up 2 x 16 bits values in the odd row. So the hex value 0x24 in xoffsets selects index 8, 9 for even row and index 14, 15 for odd row from xbuff:

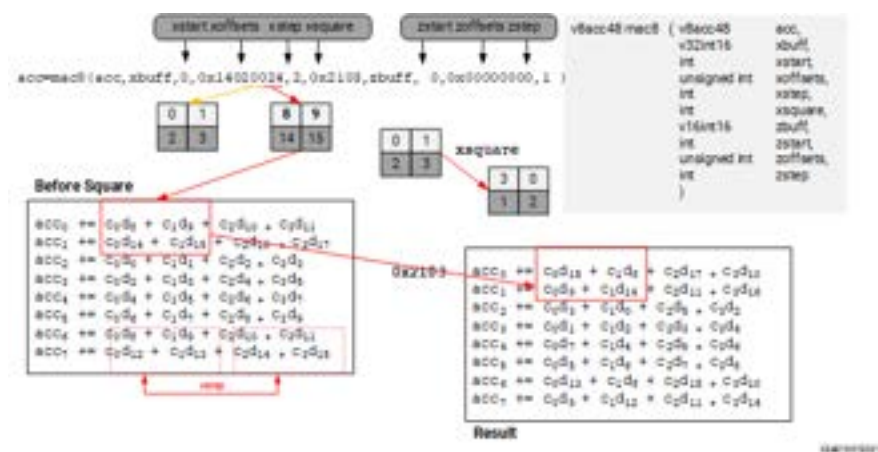
even: $2 * 4 \rightarrow$ get indices [8, 9]
 odd: $2 * (2 + 4 + 1) \rightarrow$ get indices [14, 15]

Similarly, the hex value 0x00 in xoffsets selects index 0, 1 for even row and index 2, 3 for odd row from xbuff.

There is another xsquare parameter to perform 16 bits granularity twiddling after the main permute. It will give additional contribution to the index in a 2 by 2 matrix recurring across the 8x4 matrix compute given by MUL8 in int16 x int16 mode.

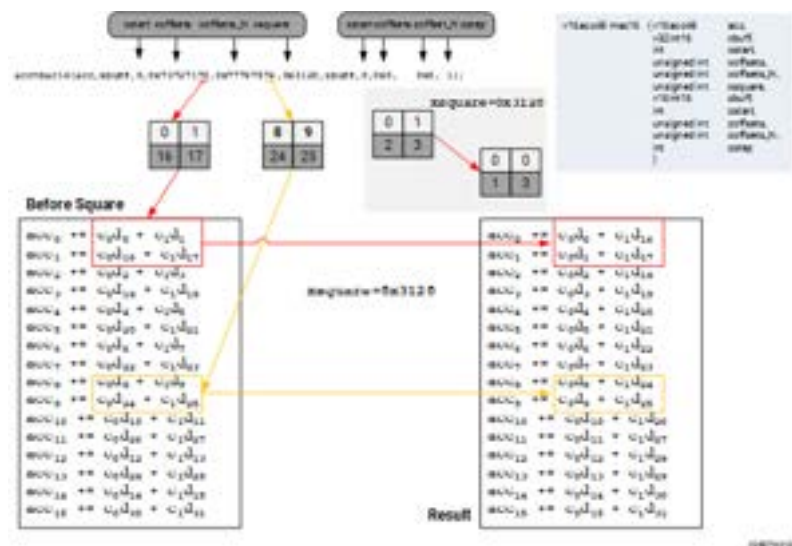
For example, xsquare value 0x2103 (see from lower hex value to higher hex value) puts index 3, 0 in the even row and index 1, 2 in the odd row. How the xsquare parameter works can be seen in the center of the following figure.

Figure: MAC8 on int16 x int16 Type



The following figure is an example of mac16 intrinsic of int16 and int16.

Figure: MAC16 on int16 x int16 Type



MAC on 8x8 bits

The following figures show MAC with int8 X buffer and int8 Z buffer. The first figure shows how data is permuted and the second figure shows how coefficients are permuted. Note that the permute granularity for X buffer and Z buffer are 32 bits and 16 bits, respectively. The xoffsets parameter comes in pair. The first hex value is an absolute 32 bits offset and pick up 4 x 8 bits values (index, index+1, index+2, index+3). The second hex value is offset from the first value + 1 (32 bits offset) and picks up 4 x 8 bits values. For example, 0x00 selects index 0, 1, 2, 3 as well as 4, 5, 6, 7, and 0x24 selects index 16, 17, 18, 19 as well as 28, 29, 30, 31.

There is another xsquare parameter to do 8 bits granularity twiddling after main permute. How xsquare parameter works in this example can be seen in the center of the following figure.

The start (xstart, zstart) and step (xstep, zstep) parameters are always in terms of data type granularity. Hence, a value of 2 for 16 bits is $2 * 16$ bits away, while a value of 2 for 8 bits is $2 * 8$ bits away. The step parameter applies to the next block of selected data. So, if a

pair of offset parameters select a 2 * 2 block, the step applies to the next 2 * 2 block. The step added to the index value must be aligned to the permute granularity (32 bits for data, 16 bits for coefficient). For example, when working with 8-bit data, xstep needs to be multiples of four. When working with 8-bit coefficient, zstep needs to be multiples of two. The following two figures show how step works for data and coefficients.

Note that for the coefficient in int8 * int8 types, the 2 * 2 index block is duplicated to construct a 4 * 2 block. See how index 0, 1, 2, and 3 are duplicated in [Figure 2](#).

Figure: MAC8 on int8 x int8 Type (X Part)

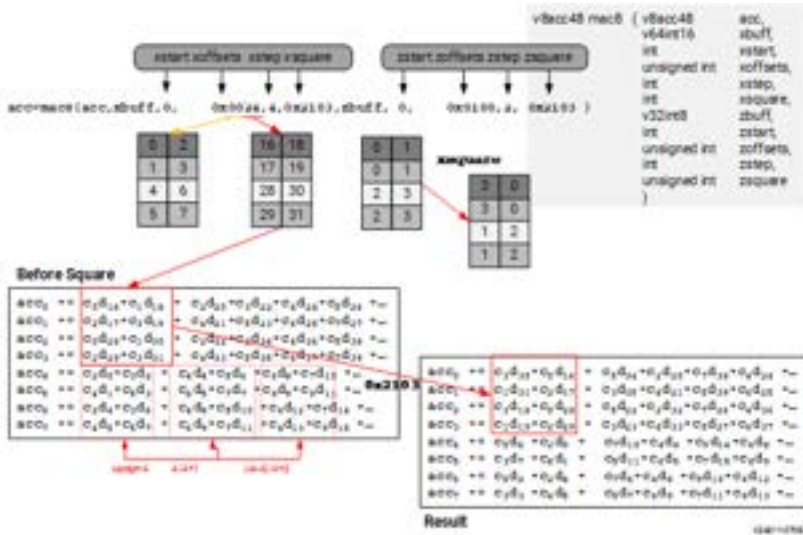
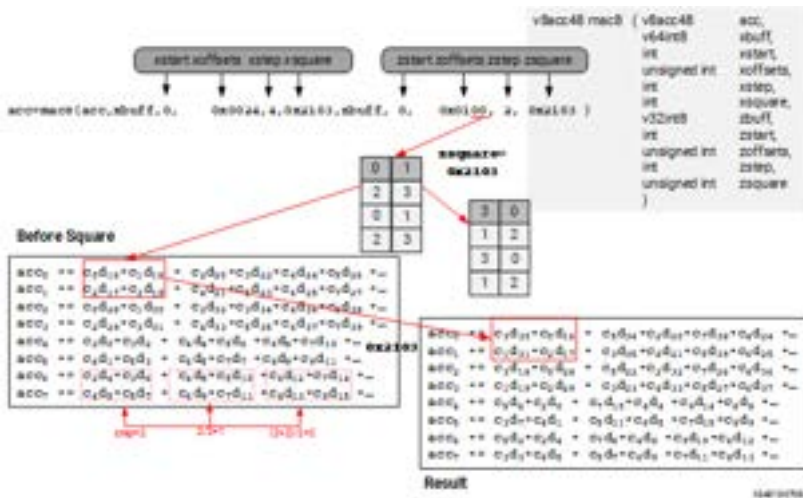


Figure: MAC8 on int8 x int8 Type (Z Part)



Options

There are rich sets of MAC intrinsic with additional operations like pre-adding, pre-subtraction, and conjugation. The naming convention for the vector MAC intrinsics is as follows. Optional characteristics are shown in [] and mandatory ones in {}.

[l]{mac|msc|mul|negmul}{2|4|8|16}[_abs|_max|_min|_maxdiff][_conj][{_sym|_antisym}[_ct|_uct]][_c|_cc|_cn|_nc]

Every operation will either be a multiplication, initializing an accumulator, or a MAC operation which accumulates to a running accumulator of 2, 4, 8, or 16 lanes.

l

Denotes that an accumulator with 80-bit lanes is used for the operation.

sym and antisym

Indicates the use of pre-adding and pre-subtraction respectively.

max, min, and maxdiff

Indicates the pre-selection of lanes in the xbuff based on the maximum, minimum, or maximum difference value.

abs

Indicates the pre-computation of the absolute value in the `xbuff`.

ct

Used for partial pre-adding and pre-subtraction (separate selection for the data input from X for the final column).

uct

Used for unit center optimization for certain types of FIR filters. Refer to the *AI Engine Intrinsic User Guide* (UG1078) for more information.

n and c

Used to indicate that the complex conjugate will be used for one of the input buffers with complex values:

c

The only 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.

conj

Indicates that the complex conjugate of Z will be used when multiplying the data input from Y.

Data Permute and MAC Examples

The following example takes two vectors with reals in `rva` and imaginary in `rvb` (with type `v8int32`) and creates a new complex vector, using the offsets to interleave the values as required.

```
v8cint32 cv = as_v8cint32(select16(0xaaaa, concat(rva, rvb),
    0, 0x03020100, 0x07060504, 8, 0x30201000, 0x70605040));
```

The following example shows how to extract real and imaginary portion of a vector `cv` with type `v8cint32`.

```
v16int32 re_im = shuffle16(as_v16int32(cv), 0, 0xECA86420, 0xFDB97531);
v8int32 re = ext_w(re_im, 0);
v8int32 im = ext_w(re_im, 1);
```

Shuffle intrinsic functions can be used to reorder the elements in a vector or set all elements to the same value. Some intrinsic functions operate only on larger registers but it is easy to use them for smaller registers. The following example shows how to implement a function to set all four elements in a vector to a constant value.

```
v4int32 v2 = ext_v(shuffle16(xset_v(0, v1), 0, 0, 0), 0);
```

The following example shows how to multiply each element in `rva` by the first element in `rvb`. This is efficient for a vector multiplied by constant value.

```
v8acc80 acc = lmul8(concat(rva, undef_v8int32()), 0, 0x76543210, rvb, 0, 0x00);
```

The following examples show how to multiply each element in `rva` by its corresponding element in `rvb`.

```
acc = lmul8(concat(rva, undef_v8int32()), 0, 0x76543210, rvb, 0, 0x76543210);
acc = lmul8(upd_w(undef_v16int32(), 0, rva), 0, 0x76543210, rvb, 0, 0x76543210);
```

The following examples show how to do matrix multiplication for `int8 x int8` data types with `mul` intrinsic, assuming that data storage is row based.

```
//Z_{2x8} * X_{8x8} = A_{2x8}
mul16(Xbuff, 0, 0x11101110, 16, 0x3120, Zbuff, 0, 0x44440000, 2, 0x3210);
//Z_{4x8} * X_{8x4} = A_{4x4}
mul16(Xbuff, 0, 0x00000000, 8, 0x3120, Zbuff, 0, 0xCC884400, 2, 0x3210);
```

If the kernel has multiple `mul` or `mac` intrinsics, try to keep the `xoffsets` and `zoffsets` parameters constant across uses and vary the `xtsart` and `zstart` parameters. This will help prevent configuration register spills on stack.

Loops

The following figure shows the assembly code of a zero-overhead loop. Note that two vector loads, one vector store, one scalar instruction, two data moves, and one vector instruction are shown in order in different slots.

[illegible]

```
for (int i=0; i<N; i+=2)
    chess_prepare_for_pipelining
    chess_loop_range(3,)
```

The loop range pragma is not needed if the loop range is a compile time constant. In general, the AI Engine compiler reports the theoretical number best suited for optimum pipelining of an algorithm. If the range specification is not optimal, the compiler would issue a warning and suggest the optimal range. Towards that end, it is okay to initially set the `<minimum>` to one [`chess_loop_range(1,)`] and observe the theoretical best suited `<minimum>` being reported by the compiler.

When looping through data, to increment or decrement by a specific offset, use the `cyclic_add` intrinsic function for circular buffers. The `fft_data_incr` intrinsic function enables the iteration of the pointer that is the current target of the butterfly operation. Using these functions can save multiple clock cycles over coding the equivalent functionality in standard C. Depending on the data types, you might need to cast parameters and return types.

The following example uses `fft_data_incr` intrinsic when operating on a matrix of real numbers.

```
pC = (v8float*)fft_data_incr( (v4cfloat*)pC, colB_tiled, pTarget);
```

Try to avoid sequential load operations to fill a vector register completely before use. It is best to interleave loads with MAC intrinsic functions, where the current MAC and next load can be done in the same cycle.

```
acc = mul4_sym(lbuff, 4, 0x3210, 1, rbuff, 11, coeff, 0, 0x0000, 1);
lbuff = upd_w(lbuff, 0, *left);
acc = mac4_sym(acc, lbuff, 8, 0x3210, 1, rbuff, 7, coeff, 4, 0x0000, 1);
```

In certain use cases loop rotation, which rotates the instructions inside the loop, can be beneficial. Instead of loading data into a vector at the start of a loop, consider loading a block of data for the first iteration before the loop, and then for the next iteration near the end of the loop. This will add additional instructions but shorten the dependency length of the loop which helps to achieve an ideal loop with a potentially lower loop range.

```
// Load starting data for first iteration
sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7

for ( int l=0; l<LSIZE; ++l )
chess_loop_range(5,)
chess_prepare_for_pipelining
{
    sbuff = upd_w(sbuff, 1, window_readincr_v8(cb_input)); // 8..15
    acc0 = mul4_sym(      sbuff,5 ,0x3210,1 ,12 ,coe,4,0x0000,1);

    sbuff = upd_w(sbuff, 2, window_readdecr_v8(cb_input)); // 16..23
    acc0 = mac4_sym(acc0,sbuff,1 ,0x3210,1 ,16,coe,0,0x0000,1);
    acc1 = mul4_sym(      sbuff,5 ,0x3210,1 ,20,coe,0,0x0000,1);
    window_writeincr(cb_output, srs(acc0, shift));
    // Load data for next iteration
    sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7
    acc1 = mac4_sym(acc1,sbuff,9,0x3210,1,16,coe,4,0x0000,1);
    window_writeincr(cb_output, srs(acc1, shift));
}
```

Floating-Point Operations

The scalar unit floating-point hardware support includes square root, inverse square root, inverse, absolute value, minimum, and maximum. It supports other floating-point operations through emulation. The `softfloat` library must be linked in for test benches and kernel code using emulation. For math library functions, the single precision float version must be used (for example, use `expf()` instead of `exp()`).

The AI Engine vector unit provides eight lanes of single-precision floating-point multiplication and accumulation. The unit reuses the vector register files and permute network of the fixed-point data path. In general, only one vector instruction per cycle can be performed in fixed-point or floating-point.

Floating-point MACs have a latency of two-cycles, thus, using two accumulators in a ping-pong manner helps performance by allowing the compiler to schedule a MAC on each clock cycle.

```
acc0 = fpmac( acc0, abuff, 1, 0x0, bbuff, 0, 0x76543210 );
acc1 = fpmac( acc1, abuff, 9, 0x0, bbuff, 0, 0x76543210 );
```

There are no divide scalar or vector intrinsic functions at this time. However, vector division can be implemented via an inverse and multiply as shown in the following example.

```
invpi = upd_elem(invpi, 0, inv(pi));
acc = fpmul(concat(acc, undef_v8float()), 0, 0x76543210, invpi, 0, 0);
```

A similar implementation can be done for the vectors `sqrt`, `invsqrt`, and `sincos`.

Design Analysis and Programming using Intrinsics

⚠ CAUTION! It is strongly recommended that you use AI Engine APIs for your designs. Usage of intrinsics must only be considered for

situations where the stringent performance needs of the design require capabilities that are not covered by the AI Engine API. For example, the AI Engine API does not currently support functionality provided by some intrinsics such as `fft_data_incr` and `cyclic_add`. While AI Engine APIs support and abstract the main permute use cases, not all permute capabilities are covered. Using intrinsics may allow you to close the performance gap required by your design.

AI Engines provide high compute density through large amount of VLIW and SIMD compute units by connecting with each other through innovative memory and AXI4-Stream networks. When targeting an application on AI Engine, it is important to evaluate the compute needs of the AI Engine and data throughput requirements. For example, how the AI Engine interacts with PL kernels and external DDR memory. After the compute and data throughput requirements can be met for AI Engine, the next step involves divide and conquer methods to map the algorithm into the AI Engine array. In the divide and conquer step, it is necessary to understand vector processor architecture, memory structure, AXI4-Stream, and cascade stream interfaces. This step is usually iterated multiple times. At the same time, each single AI Engine kernel is optimized and the graph is constructed and optimized iteratively. AI Engine tools are used to simulate and debug AI Engine kernels and the graph. The graph is then integrated with PL kernels, GMIO, and PS to perform system level verification and performance tuning. In this chapter, the divide and conquer method to map the algorithm into data flow diagrams (DFD) is briefly introduced. Single kernel programming and multiple kernels programming examples are provided to illustrate how to do kernel partitioning by the compute and memory bound, single kernel vectorization and optimization, and streaming balancing between different kernels.

Note: The references to `input_window` and `output_window` in this appendix are being deprecated. It is recommended that you use `input_buffer` and `output_buffer` instead.

Matrix Vector Multiplication

The following matrix vector multiplication example focuses on a single AI Engine kernel vectorization. It implements the following matrix vector multiplication equation.

$$C(64 \times 1) = A(64 \times 16) * B(16 \times 1)$$

The example assumes that the data for the matrices is stored in column based form and data type for the matrices A and B is `int16`.

```
c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 + a448*b7 + ...
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 + a449*b7 + ...
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 + a450*b7 + ...
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 + a451*b7 + ...
...
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 + a508*b7 + ...
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 + a509*b7 + ...
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 + a510*b7 + ...
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 + a511*b7 + ...
```

Kernel Coding Bounds

In this example, a total of 16 `int16` x `int16` multiplications are required per output value. As the matrix C consists of 64 values, a total of $16 * 64 = 1024$ multiplications is required to complete one matrix multiplication. Given that 32 16-bit multiplications can be performed per cycle in an AI Engine, the minimum number of cycles required for the matrix multiplication is $1024/32 = 32$. The summation of the individual terms comes without additional cycle requirements because the addition can be performed together with the multiplication in a MAC operation. Hence the compute bound for the kernel is:

Compute bound = 32 cycles / invocation

Next, analyze the memory accesses bound for the kernel. If it is going to fully use the vector unit MAC performance, 32 16-bit multiplications are performed per cycle. Vector b can be stored in the vector register because it is only $16 * 16\text{-bit} = 256$ bits. It does not need to be fetched from the AI Engine data memory or tile interface for each MAC operation. Considering data "a" needed for computation, it needs $32 * 16\text{-bit} = 512$ bits data per cycle. The stream interface only supports $2 * 32$ bit per cycle and hence fetching data from memory can be considered. It allows two 256 bits loads per cycle which matches the MAC performance. Thus, if two 256 bits loads are performed each cycle, the memory bound for the kernel is:

Memory bound = 32 cycles / invocation

Note that compute bound and memory bound are the theoretical limits of the kernel realization. It does not take into account the function overhead outside the main computation loop. When the kernel is only part of the graph, it might be relieved due to bandwidth limitation of other kernels or lower system performance requirements.

Vectorization

For a complicated vector processing algorithm, starting with a scalar version is recommended because it is also helpful as a golden reference for verifying the accuracy. The scalar version for matrix multiplication is shown as follows.

```
void matmul_scalar(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){ //A[M,N], B[N,1], C[M,1]. M=64, N=16
```

```

for(int i=0; i<M; i++){
    int temp = 0 ;
    for(int j=0; j<N; j++){
        temp += window_read(matA)*window_readincr(matB) ;
        window_incr(matA,64); //Jump of 64 elements to access the next element of the same row
    }
    window_writeincr(matC,(int16_t)(temp>>15)) ;
    window_incr(matA,1); //Jump of one element for moving to the next row.
}
}

```

Note that in the previously shown code, matA is stored in the column base and matB is a circular buffer to the kernel. It can be read continuously by window_readincr for computing different rows of output because it will loop back to the start of the buffer.

There are total 64 outputs (M=64), and each output needs 16 (N=16) multiplications. When choosing MAC intrinsics to do vector processing, for the data type int16 * int16, select lane 4, 8, 16 to do the equation. These are illustrated in following figure.

Figure: Lane Selection

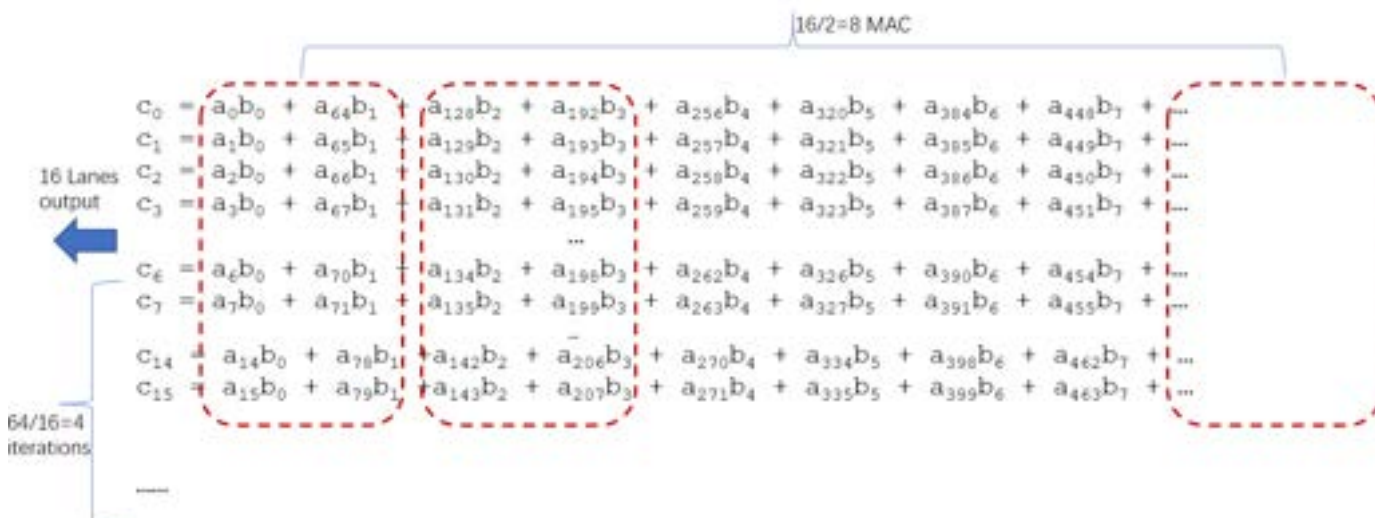
$$\begin{aligned}
 c_0 &= a_0b_0 + a_{64}b_1 + a_{128}b_2 + a_{192}b_3 + a_{256}b_4 + a_{320}b_5 + a_{384}b_6 + a_{448}b_7 + \dots \\
 c_1 &= a_1b_0 + a_{65}b_1 + a_{129}b_2 + a_{193}b_3 + a_{257}b_4 + a_{321}b_5 + a_{385}b_6 + a_{449}b_7 + \dots \\
 c_2 &= a_2b_0 + a_{66}b_1 + a_{130}b_2 + a_{194}b_3 + a_{258}b_4 + a_{322}b_5 + a_{386}b_6 + a_{450}b_7 + \dots \\
 c_3 &= a_3b_0 + a_{67}b_1 + a_{131}b_2 + a_{195}b_3 + a_{259}b_4 + a_{323}b_5 + a_{387}b_6 + a_{451}b_7 + \dots \\
 &\vdots \\
 c_6 &= a_6b_0 + a_{70}b_1 + a_{134}b_2 + a_{198}b_3 + a_{262}b_4 + a_{326}b_5 + a_{390}b_6 + a_{454}b_7 + \dots \\
 c_7 &= a_7b_0 + a_{71}b_1 + a_{135}b_2 + a_{199}b_3 + a_{263}b_4 + a_{327}b_5 + a_{391}b_6 + a_{455}b_7 + \dots \\
 &\vdots \\
 c_{14} &= a_{14}b_0 + a_{78}b_1 + a_{142}b_2 + a_{206}b_3 + a_{270}b_4 + a_{334}b_5 + a_{398}b_6 + a_{462}b_7 + \dots \\
 c_{15} &= a_{15}b_0 + a_{79}b_1 + a_{143}b_2 + a_{207}b_3 + a_{271}b_4 + a_{335}b_5 + a_{399}b_6 + a_{463}b_7 + \dots
 \end{aligned}$$

Note that the main difference between 4, 8, and 16 lanes MAC is how the data is consumed. If you assume that the data is stored by column, then 16 lanes MAC might be the best option, because only two parts of continuous data needs to be loaded for the MAC operation, – a_0 to a_{15} and a_{64} to a_{79} . a_0 to a_{15} are 256 bits, which allows one load to load the value into vector register.

To allow two loads to occur at the same cycle, a_0 to a_{15} and a_{64} to a_{79} are required to be in separate data banks. The data needs to be divided column by column into two separate buffers to the kernel. That is to say, a_0 to a_{63} are in the first buffer, a_{64} to a_{127} are in the second buffer, a_{128} to a_{191} are in the first buffer again, and so on.

By vectorization, the matrix multiplication can have a loop with $64/16=4$ iterations and each iteration of the loop contains eight MAC operations. Every iteration of the loop produces 16 output data. This is illustrated in the following figure.

Figure: Vectorization



The mac16() intrinsic function to be used has the following interface.

```

v16acc48 mac16( v16acc48 acc,
    v32int16 xbuff,
    int xstart,
    unsigned int xoffsets,
    unsigned int xoffsets_hi,
    unsigned int xsquare,
    v16int16 zbuff,
    int zstart,
    unsigned int zoffsets,
    unsigned int zoffsets_hi,
    int zstep
)

```

The buffers contain parameters (start, offsets, square, and step) to compute the indexing into the buffers (vector registers). For details about the lane addressing scheme with these parameters, see [MAC Intrinsics](#).

Coding with MAC intrinsics can be seen in the following section.

Coding with Intrinsics

You have analyzed how the function will be mapped into the AI Engine vector processor. Now have a look at the first version of the vectorized code.

```

inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB, v16acc48 &acc, int i){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,64);
    acc = mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot product

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16 outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            mac16_sub(matA,buf_matB,acc,j);
        }
        window_writeincr(matC,srs(acc,15));
        window_incr(matA,16);
    }
}


```

In the main function `matmul_vec16`, the loop produces 16 output data per iteration. In the outer loop body, there is an inner loop with eight iterations. In each iteration of the inner loop, an inline function `mac16_sub` is called. In the inline function, there is a `mac16` operation, with two loads of data for the MAC operation.

Inside `mac16_sub()`, `buf_matA` is declared as local variable and `buf_matB` and `acc` are declared as local variables in the main function. They are passed between functions by reference (or pointer). This ensures that only one identical vector exists for each variable. The function has one parameter that is used in the `mac16()` intrinsic as follows and this specific intrinsic (`i=0`) has been introduced in [MAC Intrinsics](#).

```
acc = mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
```

At the end of each iteration of the loop, window pointer for the data is incremented by 16 (that is 16 rows for the matrix).

 **Note:** While in the example, `inline` is used to guide the tool to remove the boundary of a function and `inline __attribute__((always_inline))` can be used to force removal of the boundary of the function, sometimes it is helpful to retain the boundary of a function using `__attribute__((noinline)) void func(...)`. Note that inlining or not can affect program memory usage and program optimization.

The compiled code for the kernel can be found in the disassembly view in the debug perspective of the AMD Vitis™ IDE. Note that a graph is

needed for compiling the kernel with AI Engine tools. For more understanding about the assembly code in disassembly view, refer to [Using Vitis Unified IDE and Reports](#). For additional details on graph coding and Vitis IDE usage, refer to the *AI Engine Tools and Flows User Guide* (UG1076).

Figure: Assembly Code for the Loop

Note that the compiler automatically unrolls the inner loop and pipelines the outer loop. From the previous assembly code for the loop, each iteration requires 19 cycles. However, with one window interface of data (matA), the minimum cycle number required for eight MACs must be 16 (two loads of data per MAC). This degradation of performance is caused by unbalanced window pointer increment at the end of the loop. This can be resolved by pairing the last increment with the last MAC operation. The optimized code is as follows.

```
inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB, v16acc48 &acc, int i,int incr_num){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,incr_num);
    acc = mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot product

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16 outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            int incr_num=(j==14)?80:64;
            mac16_sub(matA,buf_matB,acc,j,incr_num);
        }
        window_writeincr(matC,srs(acc,15));
    }
}
```

Note that the function mac16_sub has a new parameter incr_num. This parameter is for the pointer increment, which is different for the last function call in the inner loop. This increment number 80 for the last function call is to ensure that data in the next 16 rows is selected in the next iteration of the outer loop. Now the assembled code for the loop is as shown in following figure.

Figure: Optimized Assembly Code for the Loop

An iteration of the loop requires 16 cycles. This means that the compute bound for this kernel is $16 \times 4 = 64$ cycles per invocation. As seen in the

previous section, the theoretical limit is 32 cycles per invocation. That is eight cycles for an iteration of the loop, which means that eight MAC operations must be compacted into eight cycles. Depending on the system performance requirements, this can be achieved by splitting the data input column by column into two window buffers, `matA_0` and `matA_1`. The data of the two windows is first to be read into two `v16int16` vectors and concatenated into one `v32int16` vector to be used in the `mac16` intrinsic. The code for the kernel is as follows.

```
inline void mac16_sub_loads(input_window_int16* matA_0, input_window_int16* matA_1, v16int16 &buf_matB,
v16acc48 &acc, int i, int incr_num){
    v16int16 buf_matA0 = window_read_v16(matA_0);
    window_incr(matA_0,incr_num);
    v16int16 buf_matA1 = window_read_v16(matA_1);
    window_incr(matA_1,incr_num);
    acc = mac16(acc,concat(buf_matA0,buf_matA1),0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16* __restrict matA_0,
input_window_int16* __restrict matA_1,
input_window_int16* __restrict matB,
output_window_int16* __restrict matC){
    v16int16 buf_matB = window_read_v16(matB);
    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16 outputs
        chess_prepare_for_pipelining
        {
            v16acc48 acc=null_v16acc48();
            for(int j=0;j<16;j+=2){
                int incr_num=(j==14)?80:64;
                mac16_sub_loads(matA_0,matA_1,buf_matB,acc,j,incr_num);
            }
            window_writeincr(matC,srs(acc,15));
        }
}
```

Note that two `v16int16` vectors, `buf_matA0` and `buf_matA1`, are defined and concatenated for the `mac16` intrinsic. Also note that `chess_prepare_for_pipelining` is added for the loop and `__restrict` keyword for the window interfaces to ensure that the loop is pipelined and window operations can be well optimized.

!! Important: The `__restrict` keyword cannot be used freely. Before using it, refer to [Using the Restrict Keyword in AI Engine Kernels](#).

The assembly code for the version of two window loads in a cycle is as follows.

Figure: Assembly Code for Two Window Loads a Cycle

Matrix Multiplication

The following matrix multiplication example implements the equation:

$$C (64 \times 2) = A (64 \times 8) * B (8 \times 2)$$

The example assumes that the data for the matrices is stored in column major form and the data type for the matrices A and B is `int16`.

The first output column is computed as follows.

```
c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 + a448*b7
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 + a449*b7
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 + a450*b7
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 + a451*b7
...
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 + a508*b7
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 + a509*b7
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 + a510*b7
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 + a511*b7
```

The second output column is computed as follows.

```

c64 = a0*b8 + a64*b9 + a128*b10 + a192*b11 + a256*b12 + a320*b13 + a384*b14 + a448*b15
c65 = a1*b8 + a65*b9 + a129*b10 + a193*b11 + a257*b12 + a321*b13 + a385*b14 + a449*b15
c66 = a2*b8 + a66*b9 + a130*b10 + a194*b11 + a258*b12 + a322*b13 + a386*b14 + a450*b15
c67 = a3*b8 + a67*b9 + a131*b10 + a195*b11 + a259*b12 + a323*b13 + a387*b14 + a451*b15
...
c124 = a60*b8 + a124*b9 + a188*b10 + a252*b11 + a316*b12 + a380*b13 + a444*b14 + a508*b15
c125 = a61*b8 + a125*b9 + a189*b10 + a253*b11 + a317*b12 + a381*b13 + a445*b14 + a509*b15
c126 = a62*b8 + a126*b9 + a190*b10 + a254*b11 + a318*b12 + a382*b13 + a446*b14 + a510*b15
c127 = a63*b8 + a127*b9 + a191*b10 + a255*b11 + a319*b12 + a383*b13 + a447*b14 + a511*b15

```

Kernel Coding Bounds

In this example, a total of 1024 int16 x int16 multiplications are required for computing 128 output value. Given that 32 16-bit multiplications can be performed per cycle in an AI Engine, the compute bound for the kernel is as follows.

Compute bound = 32 cycles / invocation

Matrix B can be stored in the vector register because it is only 16*16-bit = 256 bits. It does not need to be fetched from the AI Engine data memory or tile interface for each MAC operation. Considering the data "a" needed for computation, there are total 64*8*2=1024 bytes to be fetched from memory. Given that AI Engine allows two 256 bits (32 bytes) loads per cycle, the memory bound for the kernel is as follows.

Memory bound = 1024 / (2*32) = 16 cycles / invocation

It is seen that the compute bound is larger than the memory bound. Hence the purpose of vectorization can be to achieve the theoretical limit of MAC operations in the vector processor.

Vectorization

The scalar reference code for this matrix multiplication example is shown as follows. Note that the data is stored in columns.

```

void matmul_mat8_scalar(input_window_int16* matA,
                        input_window_int16* matB,
                        output_window_int16* matC){

    for(int i=0; i<M; i++){//M=64
        for(int j=0; j<L; j++){//L=2
            int temp = 0 ;
            for(int k=0; k<N; k++){//N=8
                temp += window_read(matA)*window_readincr(matB);//B is circular buffer, size
N*L
                window_incr(matA,64); //Jump of 64 elements to access the next element of the
same row
            }
            window_write(matC,(int16_t)(temp>>15)) ;
            window_incr(matC,64); //Jump to the next column
        }
        window_incr(matA,1); //Jump of one element for moving to the next row.
        window_incr(matC,1); //Jump to the next row
    }
}

```

As analyzed in the previous example, mac16 intrinsic is the best choice for computing 16 lanes together because 16 int16 from a column can be loaded at once. To compute 16 output data in a column, four mac16 operations are needed. The same data in vector "a" is used twice to compute the data for two output columns. Thus, two columns of data can be loaded and two mac16 used for accumulations to the two output columns. These two loads and two MACs are repeated four times to get the results of two output columns. This method is shown in the following pseudo-code.

```

C_[0:15,0] = A_[0:15,0:1]*B_[0:1,0]
C_[0:15,1] = A_[0:15,0:1]*B_[0:1,1]

C_[0:15,0] += A_[0:15,2:3]*B_[2:3,0]
C_[0:15,1] += A_[0:15,2:3]*B_[2:3,1]

C_[0:15,0] += A_[0:15,4:5]*B_[4:5,0]
C_[0:15,1] += A_[0:15,4:5]*B_[4:5,1]

```



```

C_[0:15,0] += A_[0:15,6:7]*B_[6:7,0]
C_[0:15,1] += A_[0:15,6:7]*B_[6:7,1]

```

In the previous code, each "" denotes a MAC operation. `C_[0:15,0]` and `C_[0:15,1]` denote two output columns that are accumulated separately. `A_[0:15,0:1]` denotes the column 0 and 1, and each column has 16 elements. `B_[0:1,0]` denotes column 0 with 2 elements. There will be a loop for the code in the real vectorized code because there are 64 output rows. The `mac16` intrinsic function to be used has the following interface.

```

v16acc48 mac16 (      v16acc48      acc,
                    v64int16      xbuff,
                    int      xstart,
                    unsigned int   xoffsets,
                    unsigned int   xoffsets_hi,
                    unsigned int   xsquare,
                    v16int16      zbuff,
                    int      zstart,
                    unsigned int   zoffsets,
                    unsigned int   zoffsets_hi,
                    int      zstep
)

```

The buffers contain parameters (start, offsets, square, and step) to compute the indexing into buffers (vector registers). For details about the lane addressing scheme with these parameters, see [MAC Intrinsics](#).

Note that the `mac16` intrinsic function prototype is different with the one introduced in the previous matrix vector multiplication example. The `xbuff` here is `v64int16` which allows two sets of data to be stored and used in an interleaved way.

Coding with MAC intrinsics can be seen in the following section.

Coding with Intrinsics

You have analyzed how the function will be mapped into the AI Engine vector processor. Now have a look at the vectorized code.

```

void matmul_mat8(input_window_int16* matA,
                 input_window_int16* matB,
                 output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB);

    v64int16 buf_matA = undef_v64int16();
    buf_matA=upd_w(buf_matA,0,window_read_v16(matA));
    window_incr(matA,64);
    buf_matA=upd_w(buf_matA,1,window_read_v16(matA));
    window_incr(matA,64);

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16 outputs
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        v16acc48 acc0=null_v16acc48();//For first output column
        v16acc48 acc1=null_v16acc48();//For second output column

        acc0 = mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,0,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,2,window_read_v16(matA));
        window_incr(matA,64);
        acc1 = mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,8,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,3,window_read_v16(matA));
        window_incr(matA,64);

        acc0 = mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,2,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,0,window_read_v16(matA));
        window_incr(matA,64);
        acc1 = mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,10,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,1,window_read_v16(matA));
        window_incr(matA,64);
    }
}

```

```

acc0 = mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,4,0x0,0x0,1);
buf_matA=upd_w(buf_matA,2>window_read_v16(mata));
window_incr(mata,64);
acc1 = mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,12,0x0,0x0,1);
buf_matA=upd_w(buf_matA,3>window_read_v16(mata));
window_incr(mata,80);//point to next 16 rows

acc0 = mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,6,0x0,0x0,1);
window_write(matC,srs(acc0,15));
window_incr(matC,64);
buf_matA=upd_w(buf_matA,0>window_read_v16(mata));
window_incr(mata,64);
acc1 = mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,14,0x0,0x0,1);
window_write(matC,srs(acc1,15));
window_incr(matC,80);//point to next 16 rows
buf_matA=upd_w(buf_matA,1>window_read_v16(mata));
window_incr(mata,64);
}
}

```

In the previous code, buf_matB is for matrix B and it is loaded outside the loop. buf_matA is for matrix A and two sets of A are stored in lower and higher parts. When mac16 has the value "0" for xstart, the lower part of buf_matA is used. When mac16 has the value "32" for xstart, the higher part of buf_matA is used. acc0 and acc1 are the accumulated values for two output columns.

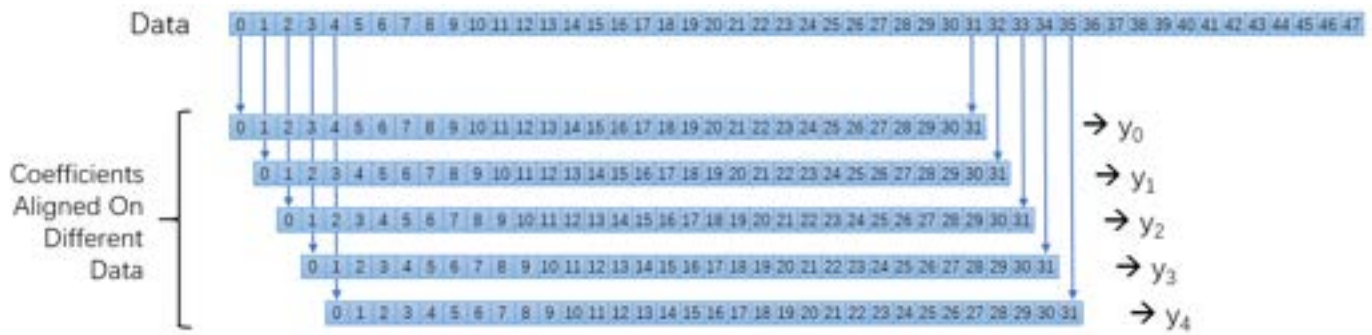
Note that buf_matA is preloaded before the loop. In the loop, the loads with window buffer pointer increment, MAC operations and the stores are interleaved. To understand how the mac16() intrinsic works, refer to [MAC Intrinsics](#). The assembled code for the loop is as shown in following figure.

Figure: Assembly Code for the Loop



In the previous equation, N denotes the taps to be used to calculate each output. The calculation process when a 32 taps filter is used as an example is shown in the following figure. int16 complex types for data and coefficient are also used as an example.

Figure: FIR Filter



1 Gbps Implementation with Cascade Stream

The AI Engine vector unit supports 8 MACs per cycle for cint16 multiply-accumulate cint16 types. If a four lane implementation of mul4/mac4 intrinsics is adopted, then there will be two complex operations on each lane.

$$\begin{cases} y_0 = c_0 \cdot d_0 + c_1 \cdot d_1 \\ y_1 = c_0 \cdot d_1 + c_1 \cdot d_2 \\ y_2 = c_0 \cdot d_2 + c_1 \cdot d_3 \\ y_3 = c_0 \cdot d_3 + c_1 \cdot d_4 \end{cases}$$

16 mac4() are needed for computing four outputs because each output requires 32 complex MACs. This means, to compute four outputs, 16 cycles using an AI Engine are required. So the sample rate of an AI Engine (assuming it runs at 1 GHz) would be as follows.

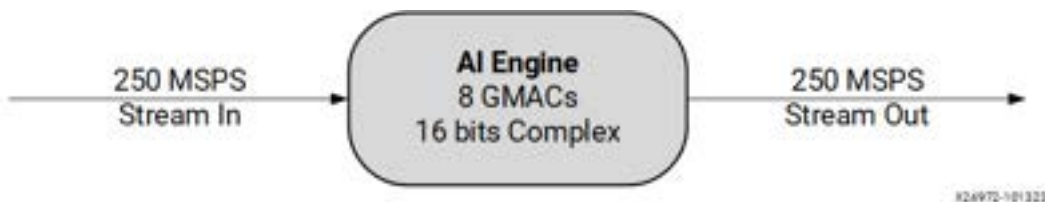
$$4 \text{ Gbps} / 16 = 0.25 \text{ Gbps} = 250 \text{ Mbps}$$

This calculates the compute bound of an AI Engine. However, the memory bound to see if that sample rate can be met still needs to be considered. Assume that only one stream input and one stream output are used for data transfer and the coefficients are stored in the AI Engine internal memory. The stream interface of an AI Engine supports 32 bits per cycle. It is capable of transferring one sample of data every cycle. Thus, the sample rate from the data transferring view is as follows.

$$1 \text{ sample/cycle} * 1 \text{ GHz} = 1 \text{ Gbps}$$

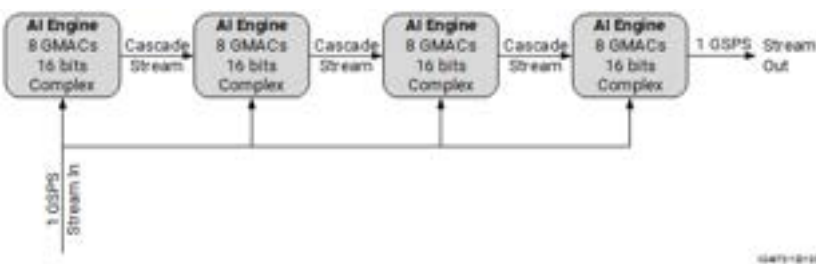
It is larger than compute bound, which is 250 Mbps. So an AI Engine implementation will operate at 250 Mbps.

Figure: One AI Engine FIR Filter Realization



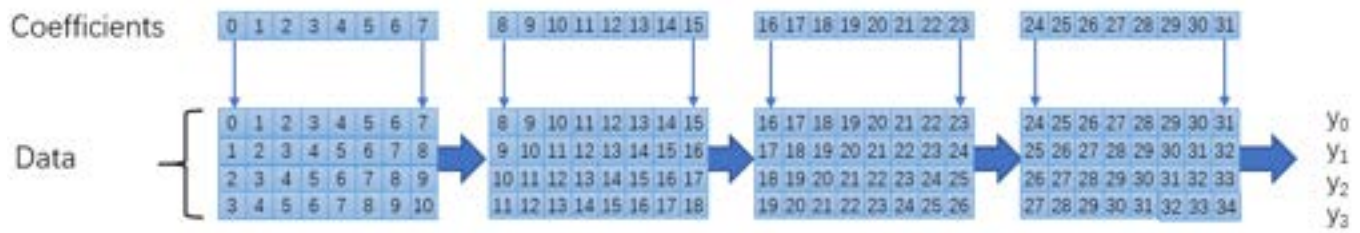
Based on the calculations, it is possible to achieve 1 Gbps via a stream input and output stream interface. If the MAC operations of a single kernel implementation are split into four kernels, $4 * 250 \text{ Mbps} = 1 \text{ Gbps}$, compute throughput can be achieved. Those four kernels are connected through cascade streaming. Therefore, the AI Engine compute bound matches AI Engine interface throughput.

Figure: 1 Gbps Implementation with Four Cascaded Kernels



Coding with Intrinsics

The four kernels in the 1 Gbps implementation can have different sets of coefficients and cascade streams between them. An implementation is shown in the following figure.

Figure: Four Kernels with Split Coefficient and Cascade Stream

Input data flows from stream to these four kernels. However, the second kernel will discard the first eight input data. The third kernel will discard the first 16 input data. Similarly, the fourth kernel will discard the first 24 input data.

The code for the first kernel is as follows.

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core0(
    input_stream_cint16 * sig_in,
    output_cascade_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef0;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;
    v4cacc48 acc;
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
        chess_prepare_for_pipelining
        chess_loop_range(4,)
        {
            acc = mul4(buff, 0 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 2, readincr_v4(sig_in));
            acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = mul4(buff, 4 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 3, readincr_v4(sig_in));
            acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 10, 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = mul4(buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 10 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 0, readincr_v4(sig_in));
            acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = mul4(buff, 12 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 1, readincr_v4(sig_in));
            acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);
        }
}
```

```

    delay_line = buff;
}

void fir_32tap_core0_init()
{
    // Drop samples if not first block
    int const Delay = 0;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }

};

```

Note that the function, `fir_32tap_core0_init`, is going to be the initialization function for the AI Engine kernel, `fir_32tap_core0`, which is only executed once at the kernel start. The purpose of this initialization function is to discard the unnecessary samples to align the input stream.

Similarly, the function, `fir_32tap_core1_init`, is going to be the initialization function for the AI Engine kernel, `fir_32tap_core1`, in the following codes. Same applies for the initialization functions, `fir_32tap_core2_init` and `fir_32tap_core3_init`.

The second kernel code is as follows.

```

#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core1(
    input_stream_cint16 * sig_in,
    input_cascade_cacc48 * cascadein,
    output_cascade_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef1;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;
    v4cacc48 acc;
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout, acc);

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 10, 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout, acc);

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 10 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 6, 0x0000, 1);
    }
}

```

```

    writeincr_v4(cascadeout,acc);

    acc = readincr_v4(cascadein);
    acc = mac4(acc, buff, 12 , 0x3210, 1,  coe, 0, 0x0000, 1);
    acc = mac4(acc, buff, 14 , 0x3210, 1,  coe, 2, 0x0000, 1);
    buff = upd_v(buff, 1, readincr_v4(sig_in));
    acc = mac4(acc, buff, 0 , 0x3210, 1,  coe, 4, 0x0000, 1);
    acc = mac4(acc, buff, 2 , 0x3210, 1,  coe, 6, 0x0000, 1);
    writeincr_v4(cascadeout,acc);
}
delay_line = buff;
}

void fir_32tap_core1_init()
{
    // Drop samples if not first block
    int const Delay = 8;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};

```

The third kernel is similar to the second one. The last kernel is as follows.

```

#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core3(
    input_stream_cint16 * sig_in,
    input_cascade_cacc48 * cascadein,
    output_stream_cint16 * data_out)
{
    const cint16_t * __restrict coeff = eq_coef3;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;

    v4cacc48 acc;

    set_rnd(rnd_pos_inf);
    set_sat();
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 0 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 2 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc = mac4(acc, buff, 4 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 4 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc = mac4(acc, buff, 8 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 10, 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));
    }
}

```

```

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 10 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 1, readincr_v4(sig_in));
        acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));
    }
    delay_line = buff;
}

void fir_32tap_core3_init()
{
    // Drop samples if not first block
    int const Delay = 24;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};

```

The graph code is as follows.

```

#include <adf.h>
#include "kernels.h"
using namespace adf;
class firGraph : public graph {
public:
    kernel k0,k1,k2,k3;
    input_port in0123;
    output_port out;
    firGraph()
    {
        k0 = kernel::create(fir_32tap_core0);
        runtime<ratio>(k0) = 0.9;
        source(k0) = "fir_32tap_core0.cpp";
        connect<stream> n0(in0123,k0.in[0]);

        k1 = kernel::create(fir_32tap_core1);
        runtime<ratio>(k1) = 0.9;
        source(k1) = "fir_32tap_core1.cpp";
        connect<stream> n1(in0123,k1.in[0]);
        connect<cascade> (k0.out[0],k1.in[1]);

        k2 = kernel::create(fir_32tap_core2);
        runtime<ratio>(k2) = 0.9;
        source(k2) = "fir_32tap_core2.cpp";
        connect<stream> n2(in0123,k2.in[0]);
        connect<cascade> (k1.out[0],k2.in[1]);

        k3 = kernel::create(fir_32tap_core3);
        runtime<ratio>(k3) = 0.9;
        source(k3) = "fir_32tap_core3.cpp";
        connect<stream> n3(in0123,k3.in[0]);
        connect<cascade> (k2.out[0],k3.in[1]);
        connect<stream> (k3.out[0],out);
    }
};

```

```

        initialization_function(k0) = "fir_32tap_core0_init";
        initialization_function(k1) = "fir_32tap_core1_init";
        initialization_function(k2) = "fir_32tap_core2_init";
        initialization_function(k3) = "fir_32tap_core3_init";
    };
};

```

The kernels connected through cascade streams are expected to operate synchronously. Conflicts in cascade streams can stall the kernels. Loops in the kernels must have input data available to run smoothly. Hence it is important that the input stream arrives at the appropriate time for each kernel. The input stream stall (if any) can be resolved by adding a large enough FIFO to the net connecting to the AI Engine kernels. For example:

```

fifo_depth(n0)=175;
fifo_depth(n1)=150;
fifo_depth(n2)=125;
fifo_depth(n3)=100;

```

Note that different FIFO depths are specified in the previous example to prevent auto FIFO merge which can occur when a common FIFO depth is used for all nets.

For the purpose of saving FIFO resources, individual FIFO depths can be set by looking at when the event `CORE_INSTREAM_WIDE` occurs for each kernel. The earlier the event occurs, the deeper the FIFO needs to be. For example:

```

fifo_depth(n0)=45;
fifo_depth(n1)=33;
fifo_depth(n2)=23;
fifo_depth(n3)=10;

```

For additional details about coding on graph, refer to the *AI Engine Tools and Flows User Guide* ([UG1076](#)).

Adaptive Data Flow Graph Specification Reference

Unless otherwise stated, all classes and their member functions belong to the `adf` name space.

Return Code

ADF APIs have defined return codes to indicate success or different kinds of failures in the `adf` namespace.

```

enum return_code
{
    ok = 0,
    user_error,
    aie_driver_error,
    xrt_error,
    internal_error,
    unsupported
};

```

The following defines the different return codes:

ok

success

user error

Possible invalid argument or use of the API in an unsupported way

aie_driver_error

If the AI Engine driver returns errors, the graph API returns this error code.

xrt_error

If XRT returns errors, the graph API returns this error code.

internal error

This means something is wrong with the tool; contact AMD support.

unsupported

Unsupported feature or unsupported scenario.

Graph Objects

graph

This is the main graph abstraction exported by the ADF tools. All user-defined graphs should be inherited from `class graph`.

Scope

All instances of those user-defined graph types that form part of a user design must be declared in global scope, but can be declared under any name space.

Member Functions

```
virtual return_code init() ;
```

This method loads and initializes a precompiled graph object onto the AI Engine array using a predetermined set of processor tiles. Currently, no relocation is supported. All existing information in the program memory, data memory, and stream switches belonging to the tiles being loaded is replaced. The loaded processors are left in a disabled state.

```
virtual return_code run();
virtual return_code run(unsigned int num_iterations);
```

This method enables the processors associated with a graph to start execution from the beginning of their respective main programs. Without any arguments, the graph will run forever. The API with arguments can set the number of iterations for each run differently. This is a non-blocking operation on the PS application.

```
virtual return_code end();
virtual return_code end(unsigned int cycle_timeout);
```

The end method is used to wait for the termination of the graph. A graph is considered to be terminated when all its active processors exit their main thread and disable themselves. This is a blocking operation for the PS application. This method also cleans up the state of the graph such as forcing the release of all locks and cleaning up the stream switch configurations used in the graph. The end method with cycle timeout terminates and cleans up the graph when the timeout expires rather than waiting for any graph related event. Attempting to run the graph after end without re-initializing it can give unpredictable results.

```
virtual return_code wait();
virtual return_code wait(unsigned int cycle_timeout);
```

```
virtual return_code resume();
```

The wait method is used to pause the graph execution temporarily without cleaning up its state so that it can be restarted with a run or resume method. The wait method without arguments is useful when waiting for a previous run with a fixed number of iterations to finish. This can be followed by another run with a new set of iterations. The wait method with cycle timeout pauses the graph execution when the timeout expires counted from a previous run or resume call. This should only be followed by a resume to let the graph to continue to execute. Attempting to run after a wait with cycle timeout call can lead to unpredictable results because the graph can be paused in an unpredictable state and the run can restart the processors from the beginning of their main programs.

```
virtual return_code update(input_port& pName, <type> value);

virtual return_code update(input_port& pName, const <type>* value, size_t size);
```

These methods are various forms of runtime parameter update APIs that can be used to update scalar or array runtime parameter ports. The port name is a fully qualified path name such as `graph1.graph2.port` or `graph1.graph2.kernel.port`. The `<type>` can be one of `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`, `cfloat`. For array runtime parameter updates, a size argument specifies the number of elements in the array to be updated. This size must match the RTP array size defined in the graph, meaning that the full RTP array must be updated at one time.

```
virtual return_code read(input_port& pName, <type>& value);
```

```
virtual return_code read(input_port& pName, <type>* value, size_t size);
```

These methods are various forms of runtime parameter read APIs that can be used to read scalar or array runtime parameter ports. The port name is a fully qualified path name such as `graph1.graph2.port` or `graph1.graph2.kernel.port`. The `<type>` can be one of `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`, `cfloat`. For array runtime parameter reads, a `size` argument specifies the number of elements in the array to be read.

kernel

This class represents a single node of the graph. User-defined graph types contain kernel objects as member variables that wrap over some C function computation mapped to the AI Engine array.

Scope

`kernel` objects can be declared in class scope as member variables in a user-defined graph type (i.e., inside a class that inherits from `graph`).

`kernel` objects must be initialized by assignment in the graph constructor.

Member Functions

```
static kernel & create( function );
```

The static `create` method creates a kernel object from a C kernel function. It automatically determines how many input ports and output ports each kernel has and their appropriate element type. Any other arguments in a kernel function are treated as runtime parameters or lookup tables, passed to the kernel on each invocation. Runtime parameters are passed by value, while lookup tables are passed by reference each time the kernel is invoked by the compiler generated static-schedule.

```
kernel & operator()(...)
```

Takes one or more parameter objects as arguments. The number of parameter arguments must match the number of non-buffer formal arguments in the kernel function used to construct the `kernel`. When used in the body of a graph constructor to assign to a `kernel` member variable, the operator ensures that updated parameter arguments are passed to the kernel function on every invocation.

Member Variables

```
std::vector<port<input>> in;
```

This variable provides access to the logical inputs of a kernel, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th input port (buffer, stream, or rtp) declared in the kernel function arguments.

```
std::vector<port<output>> out;
```

This variable provides access to the logical outputs, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th output port (buffer or stream) declared in the kernel function arguments.

```
std::vector<port<inout>> inout;
```

This variable provides access to the logical inout ports, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th inout port (rtp) declared in the kernel function arguments.

`port<T>`

Scope

Objects of type `port<T>` are port objects that can be declared in class scope as member variables of a user-defined graph type (i.e., member variables of a class that inherits from `graph`), or they are defined implicitly for a kernel according to its function signature. The template parameter `T` can be one of `input`, `output`, or `inout`.

Aliases

`input_port` is an alias for the type `port<input>`.
`output_port` is an alias for the type `port<output>`.
`inout_port` is an alias for the type `port<inout>`.

Purpose

Used to connect between kernels within a graph and across levels of hierarchy in customer specification containing platform, graphs, and subgraphs.

Operators

```
port<T>& negate(port<T>&)
```

When applied to a destination port within a connection, this operator inverts the Boolean semantics of the source port to which it is connected. Therefore, it has the effect of converting a 0 to 1 and 1 to 0.

```
port<T>& async(port<T>&)
```

When applied to a destination RTP port within a connection, this operator specifies an *asynchronous* update of the destination port's RTP buffer from the source port that it is connected to or from the external control application if the source is a graph port left unconnected. Therefore, the receiving kernel does not wait for the value for each invocation, rather it uses previous value stored in the corresponding buffer.

```
port<T>& sync(port<T>&)
```

When applied to a source RTP port within a connection, this operator specifies a *synchronous* read of the source port's RTP buffer from the destination port that it is connected to or from the external control application if the destination is a graph port left unconnected. Therefore, the receiving kernel waits for a new value to be produced for each invocation of the producing kernel.

Related Information

[Logical I/O Ports](#)

parameter

The `parameter` class contains two static member functions to allow you to associate globally declared variables with kernels.

Member Functions

```
static parameter & array(X)
```

Wrap around any extern declaration of an array to capture the size and type of that array variable.

```
static parameter & scalar(Y)
```

Wrap around any extern declaration of a scalar value (including user defined structs).

bypass

This class is a control flow encapsulator with data bypass. It wraps around an individual node or subgraph to create a bypass data path based on a dynamic control condition. The dynamic control is coded as a runtime parameter port `bp` (with integer value 0 or 1) that controls whether the input buffer (or stream) data will flow into the kernel encapsulated by the bypass (`bp=0`) or whether it will be directly bypassed into the output buffer (or stream) (`bp=1`).

Scope

`bypass` objects can be declared in class scope as member variables in a user-defined graph type (that is, inside a class that inherits from `graph`).

`bypass` objects must be initialized by assignment in the graph constructor.

Member Functions

```
static bypass & create( kernel );
```

The static `create` method creates a bypass object around a given kernel object. The number of inputs and outputs of the bypass are inferred automatically from the corresponding ports of the kernel.

Graph Objects for Packet Processing

The following predefined object classes in the `adf` namespace are used to define the connectivity of packet streams.

```
template <int nway> class pktsplit { ... }
template <int nway> class pktmerge { ... }
```

Scope

Objects of type `pktsplit<n>` and `pktmerge<n>` can be declared as member variables in a user-defined graph type (i.e., inside a class that inherits from `graph`). The template parameter `n` must be a compile-time constant positive integer denoting the `n`-way degree of split or merge. These objects behave like ordinary nodes of the graph with input and output connections, but are only used for explicit packet routing.

Member Functions

```
static pktsplit<nway> & create();
static pktmerge<nway> & create();
```

The static `create` method for these classes work in the same way as `kernel create` method. The degree of split or merge is already specified in the template variable declaration.

Member Variables

```
std::vector<port<input>> in;
```

This variable provides access to the logical inputs of the node. There is only one input for `pktsplit` nodes. For `pktmerge` nodes the `i`'th index selects the `i`'th input port.

```
std::vector<port<output>> out;
```

This variable provides access to the logical outputs of the node. There is only one output for `pktmerge` nodes. For `pktsplit` nodes the `i`'th index selects the `i`'th output port.

Input/Output Objects

input_gmio/output_gmio

This class represents the global memory (DDR) resource management and data transfer between AI Engine and global memory (DDR). The `input_gmio` object manages data transfer from global memory (DDR) to AI Engine or read from global memory (DDR) operation. The `output_gmio` object manages data transfer from AI Engine to global memory or write to global memory (DDR) operation.

Base Class Member Functions

```
static void* malloc(size_t size);
```


The `malloc` method allocates contiguous physical memory space and returns the corresponding virtual address. It accepts a parameter, `size`, to specify how many bytes to be allocated. If successful, the pointer to the allocated memory space is returned. `nullptr` is returned in the event of a failure.

```
static void free(void* address);
```

The `free` method frees memory space allocated by `GMI0::malloc`.

```
return_code wait();
```

The `wait` method blocks until all the previously issued transactions are complete. This method is only applicable for `GMI0` objects connected to AI Engine.

 **Note:** The `input_gmio` and `output_gmio` classes are derived from `GMIO` base class. Functions `malloc()`, `free()`, and `wait()` are declared in the `GMIO` base class.

input_gmio Member Functions

```
create(std::string logical_name, size_t burst_length, size_t bandwidth);
```

The above port specification is used to connect DDR memory to AI Engine kernels. `logical_name` is the name of the port. The `burst_length` is the length of DDR memory burst transaction (can be 64, 128 or 256 bytes), and the `bandwidth` is the average expected throughput in MB/s.

```
create(size_t burst_length, size_t bandwidth);
```

The above port specification is used to connect DDR memory to AI Engine kernels. The `burst_length` is the length of DDR memory burst transaction (can be 64, 128 or 256 bytes), and the `bandwidth` is the average expected throughput in MB/s.

```
return_code gm2aie_nb(const void* address, size_t transaction_size);
```

The `gm2aie_nb` method initiates a DDR to AI Engine transfer. The memory space for the transaction is specified by the address pointer and the `transaction_size` parameter (in bytes). The transaction memory space needs to be within the total memory space allocated by the `GMIO::malloc` method. It is a non-blocking function in that it does not wait for the read transaction to complete.

```
return_code gm2aie(void* address, size_t transaction_size);
```

The `gm2aie` method is a blocking version of `gm2aie_nb`. It blocks until the AI Engine–DDR read transaction completes.

output_gmio Member Functions

```
create(std::string logical_name, size_t burst_length, size_t bandwidth);
```

The above port specification is used to connect AI Engine kernels to DDR memory. `logical_name` is the name of the port. The `burst_length` is the length of DDR memory burst transaction (can be 64, 128 or 256 bytes), and the `bandwidth` is the average expected throughput in MB/s.

```
create(size_t burst_length, size_t bandwidth);
```


The above port specification is used to connect AI Engine kernels to DDR memory. The `burst_length` is the length of DDR memory burst transaction (can be 64, 128 or 256 bytes), and the `bandwidth` is the average expected throughput in MB/s.

```
return_code aie2gm_nb(void* address, size_t transaction_size);
```

The `aie2gm_nb` method initiates an AI Engine to DDR transfer. The memory space for the transaction is specified by the address pointer and the `transaction_size` parameter (in bytes). The transaction memory space needs to be within the total memory space allocated by the `GMIO::malloc` method. It is a non-blocking function in that it does not wait for the write transaction to complete.

```
return_code aie2gm(void* address, size_t transaction_size);
```

The `aie2gm` method is a blocking version of `aie2gm_nb`. It blocks until the AI Engine–DDR write transaction completes.

 **Note:** It is recommended using `GMIO::malloc()` and `GMIO::free()` to manage DDR memory resources.

input_plio/output_plio

These classes represent the I/O port specification used to connect AI Engine kernels to the external platform ports representing programmable logic.

Member Functions

```
create(plio_type plio_width, std::string datafile);
```

The above `input_plio/output_plio` port specification is used to represent a single 32-bit, 64-bit, or 128-bit input or output AXI4-Stream

port at the AI Engine array interface. The `datafile` is an input or output file path that sources input data or receives output data for simulation purposes. This data could be captured separately during simulation run.

```
create(std::string logical_name, plio_type plio_width, std::string datafile);
```

The above `input_plio/output_plio` port specification is used to represent a single 32-bit, 64-bit, or 128-bit input or output AXI4-Stream port at the AI Engine array interface. Here the `plio_width` can be one of `plio_32_bits` (default), `plio_64_bits`, or `plio_128_bits`. The `logical_name` must be the same as the annotation field of the corresponding port as presented in the logical architecture interface specification.

```
create(std::string logical_name, plio_type plio_width, std::string datafile, double frequency);
```

The above `input_plio/output_plio` port specification is used to represent a single 32-bit, 64-bit, or 128-bit input or output AXI4-Stream port at the AI Engine array interface. Here `plio_width` can be one of `plio_32_bits` (default), `plio_64_bits`, or `plio_128_bits`. The frequency of the `input_plio/output_plio` port can also be specified as part of the constructor.

```
create(std::string logical_name, plio_type plio_width, std::string datafile, double frequency, bool hex);
```

The above `input_plio/output_plio` boolean port specification is used to indicate if the contents in the input data file are in hex or binary formats.

The data in the data files must be organized according to the bus width of the `input_plio/output_plio` attribute (32, 64, or 128) per line as well as the data type of the graph port it is connected to. For example, a 64-bit `input_plio` feeding a kernel port with data type `int32` requires file data organized as two columns. However, the same 64-bit `input_plio` feeding to a kernel port with data type `cint16` requires the data to be organized into four columns, each representing a 16-bit real or imaginary part of the complex data type.

Connections

The following template object constructors specify different types of connections between ports. Each of them support the appropriate overloading for input/output/inout ports. Specifying the connection object name while creating a connection is optional, but it is recommended for better debugging.

Connection Constructor Templates

```
connect [name](portA, portB)
```

Connects two kernel ports.

- `portA` can be a stream port output, a cascade output or an I/O-buffer output.
- `portB` can be a stream port input, a cascade input or an I/O-buffer input

Cascade ports must be connected together, defined as follows:

```
connect [name] (cascade out, cascade in)
```

Table: Available Connections

	Stream Input	I/O-Buffer Input	Cascade Input
Stream Output	Yes	Yes	N/A
I/O-Buffer Output	Yes	Yes	N/A
Cascade Output	N/A	N/A	Yes

```
connect [name](portA, portB)
```

Connects between hierarchical ports between different levels of hierarchy.

```
connect [name](parameter, portB)
```

Connects a parameter port to a kernel port.

```
connect [name](LUT, kernel)
```

Connects a LUT parameter array object to a kernel.

Port Combinations

The port combinations used in the constructor templates are specified in the following table.

Table: Port Combinations

PortA	PortB	Comment
port <output>	port <output>	Connect a kernel output to a parent graph output
port <output>	port <input>	Connect a kernel output to a kernel input
port <output>	port <inout>	Connect an output of a kernel or a subgraph to an inout port of another kernel or a subgraph
port <input>	port <input>	Connect a graph input to a kernel input
port <input>	port <inout>	Connect an input of a parent graph to an inout port of a child subgraph or a kernel
port <inout> >	port <input>	Connect an inout port of a parent graph or a kernel to an input of another kernel or a subgraph
port <inout>	port <output>	Connect an inout port of a subgraph or a kernel to an output of a parent graph
parameter&	kernel&	Connect an initialized parameter variable to a kernel ensuring that the compiler allocates space for the variable in the memory around the kernel

Related Information

[Logical I/O Ports](#)

Constraints

Constraints are user-defined properties for graph nodes that provide additional information to the compiler.

`constraint<T>`

This template class is used to build scalar data constraints on kernels, connections, and ports.

Scope

A constraint must appear inside a user graph constructor.

Member Functions

```
constraint<T> operator=(T)
```

This overloaded equality operator allows you to assign a value to a scalar constraint.

Constructors


The default constructor is not used. Instead the following special constructors are used with specific meaning.

```
constraint<std::string>& initialization_function(kernel&)
```

This constraint allows you to set a specific initialization function for each kernel. The constraint expects a string denoting the name of the initialization function. Where multiple kernels are packed on a core, each initialization function packed on the core is called exactly once. No kernel functions are scheduled until all the initialization functions packed on a core are completed.

An initialization function cannot return a value and cannot have input/output arguments, that is, the function prototype must be as follows.

```
void init_function_name(void)
```

 **Note:** The initialization function is called only once when the first `graph::run` API is called.

This function can be used to initialize global variables and set or clear rounding and saturation modes. It cannot use buffer or stream APIs to access memory or stream interfaces, but stream intrinsics (for example, `get_ss()`) can be used.

```
constraint<float>& runtime<ratio>(kernel&)
```

This constraint allows you to set a specific core usage fraction for a kernel. This is computed as a ratio of the number of cycles taken by one invocation of a kernel (processing one block of data) to the cycle budget. The cycle budget for an application is typically fixed according to the expected data throughput and the block size being processed.

```
constraint<std::string>& source(kernel&)
```

This constraint allows you to specify the source file containing the definition of each kernel function. A source constraint must be specified for each kernel.

```
constraint<int>& fifo_depth(connect&)= [<depth> | (depth)]
```

This constraint allows you to specify the amount of slack to be inserted on a streaming connection to allow deadlock free execution.

```
void single_buffer(port<T>&)
```

This constraint allows you to specify single buffer constraint on a buffer port. By default, a buffer port is double buffered.

```
template <typename _XILINX_ADF_T=api_impl::variant>
constraint<_XILINX_ADF_T> initial_value(adf::port<adf::input>& p);
```

This constraint allows you to set the initial value for an asynchronous AI Engine input runtime parameter port. It allows the destination kernel to start asynchronously with the specified initial value. You can set both scalar and array runtime parameters using this constraint.

Example scalar: `initial_value(k.in[1])=6;`

Example array: `initial_value(k.in[2])={1,2,3};`

```
constraint<int> stack_size(adf::kernel& k);
```

This constraint allows you to set stack size for individual kernel.

```
constraint<int> heap_size(adf::kernel& k);
```

This constraint allows you to set heap size for individual kernel.

```
constraint< std::vector<T>>
```

This template class is used to build vector data constraints on kernels, connections, and ports.

Scope

Constraint must appear inside a user graph constructor.

Member Function

```
constraint<std::vector<T> > operator=(std::vector<T>)
```

Constraint must appear inside a user graph constructor.

Constructors

The default constructor is not used. Instead the following special constructors are used with specific meaning.

```
constraint <std::vector<std::string > >& headers (kernel&)
```

This constraint allows you to specify a set of header files for a kernel that define objects to be shared with other kernels and hence have to be included once in the corresponding main program. The kernel source file would instead include an `extern` declaration for that object.

Mapping Constraints

The following functions help to build various types of constraints on the physical mapping of the kernels and buffers onto the AI Engine array.

Scope

A constraint must appear inside a user graph constructor.

Kernel Location Constructors

```
location_constraint tile(int col, int row)
```

This location constructor points to a specific AI Engine tile located at specified column and row within the AI Engine array. The column and row values are zero based, where the zero'th row is counted from the bottom-most row with a compute processor and the zero'th column is counted from the left-most column. The previously used constructor `proc(col, row)` is now deprecated.

```
location_constraint location<kernel> (kernel&)
```

This constraint provides a handle to the location of a kernel so that it can be constrained to be located on a specific tile or co-located with another kernel using the following assignment operator.


Buffer Location Constructors

```
location_constraint address(int col, int row, int offset)
```

This location constructor points to a specific data memory address offset on a specific AI Engine tile. The offset address is relative to that tile starting at zero with a maximum value of 32768 (32K).

```
location_constraint bank(int col, int row, int bankid)
```

This location constructor points to a specific data memory bank on a specific AI Engine tile. The bank ID is relative to that tile and can take values 0, 1, 2, 3.

 **Note:** The hardware view is 8 banks of 128-bit width. The software view is 4 banks of 256 width.

```
location_constraint offset(int offset_value)
```

This location constructor specifies data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.

```
location_constraint location<buffer> (port<T>&)
```

This location constructor provides a handle to the location of a buffer attached to an input, output, or inout port of a kernel. It can be used to constrain the location of the buffer to a specific address or bank, or to be on the same tile as another kernel, or to be on the same bank as another buffer using the following assignment operator. It is an error to constrain two buffers to the same address. This constructor only applies to buffer kernel ports.

```
location_constraint location<stack> (kernel&)
```

This location constructor provides a handle to the location of the system memory (stack and heap) of the AI Engine where the specified kernel is mapped. This provides a mechanism to constrain the location of the system memory with respect to other buffers used by that kernel.

!! Important: The stack location offset must be in multiples of 32 bytes.

```
location_constraint location<parameter> (parameter&)
```

This location constructor provides a handle to the location of the parameter array (for example, a lookup table) declared within a graph.

Bounding Box Constructor

```
location_constraint bounding_box(int column_min, int row_min, int column_max, int row_max)
```

This bounding box constructor specifies a rectangular bounding box for a graph to be placed in AI Engine tiles, between columns from `column_min` to `column_max` and rows from `row_min` to `row_max`. Multiple bounding box location constraints can be used in an initializer list

to specify an irregular shape bounding region.

Operator Functions

```
location_constraint& operator=(location_constraint)
```

This operator expresses the equality constraint between two location constructors. It allows various types of absolute or relative location constraints to be expressed.

The following example shows how to constrain a kernel to be placed on a specified AI Engine tile.

```
location<kernel>(k1) = tile(3,2);
```

The following template shows how to constrain the location of double buffers attached to a port that are to be placed on a specific address or a bankid. At most, two elements should be specified in the initializer list to constrain the location of the double banks. Furthermore, if these buffers are read or written by a DMA engine, then they must be on the same tile.

```
location<buffer>(port1) = { [address(c,r,o) | bank(c,r,id)] , [address(c,r,o) | bank(c,r,id)] };
```

The following template shows how to constrain the location of a parameter lookup table or the system memory of a kernel to be placed on a specific address or a bankid.

```
location<parameter>(param1) = [address(c,r,o) | bank(c,r,id)];
```

```
location<stack>(k1) = [address(c,r,o) | bank(c,r,id)];
```

The following example shows how to constrain two kernels to be placed relatively on the same AI Engine. This forces them to be sequenced in topological order and be able to share memory buffers without synchronization.

```
location<kernel>(k1) = location<kernel>(k2);
```

The following example shows how to constrain a buffer, stack, or parameter location to be on the same tile as that of a kernel. This ensures that the buffer, or parameter array can be accessed by the other kernel k1 without requiring a DMA.

```
location<buffer>(port1) = location<kernel>(k1);
```

```
location<stack>(k2) = location<kernel>(k1);
```

```
location<parameter>(param1) = location<kernel>(k1);
```

The following example shows how to constrain a buffer, stack, or parameter location to be on the same bank as that of another buffer, stack, or parameter. When two double buffers are co-located, this constrains both the ping buffers to be on one bank and both the pong buffers to be on another bank.

```
location<buffer>(port2) = location<buffer>(port1);
```

```
location<parameter>(param1) = location<buffer>(port1);
```

The following example shows how to constraint a graph to be placed within a bounding box or a joint region among multiple bounding boxes.

```
location<graph>(g1) = bounding_box(1,1,2,2);
```

```
location<graph>(g2) = { bounding_box(3,3,4,4), bounding_box(5,5,6,6) };
```

The following example shows how to constrain FIFO locations using the DMA FIFO with `aie_tile/memory_tile/shim_tile`, tile column number, tile row number, memory address, and size as input parameters, and/or the stream switch FIFO with `aie_tile/memory_tile/shim_tile`, tile column number, tile row number, and FIFO identifier as input parameters.

```
location(net0) = { dma_fifo(tile_type0, col0, row0, address0, size0), ss_fifo(tile_type1, col1, row1, fifo_id1), ... }
```

Non-Equality Function

```
void not_equal(location_constraint lhs, location_constraint rhs)
```

This function expresses $lhs \neq rhs$ for the two `location_constraint` parameters `lhs` and `rhs`. It allows relative non-collocation constraint to be specified. The `not_equal` buffer constraint only works for single buffers. This constraint should not be used with double buffers. The following example shows how to specify two kernels, `k1` and `k2`, should not be mapped to the same AI Engine.

```
not_equal(location<kernel>(k1), location<kernel>(k2));
```

The following example shows how to specify two buffers, `port1` and `port2`, should not be mapped to the same memory bank.

```
not_equal(location<buffer>(port1), location<buffer>(port2));
```

Stamp and Repeat Constraint

The following example shows how to constraint a graph to be placed within a bounding box. In this case, the `tx_chain0` graph is the reference and its objects will be placed first and stamped to graph `tx_chain1` and `tx_chain2`. The number of rows for all identical graphs (reference plus stamp-able ones) must be the same, and must begin and end at the same parity of row, meaning if the reference graph's bounding box begins at an even row and ends at an odd row, then all of the stamped graphs must follow the same convention. This limitation occurs because of the mirrored tiles in AI Engine array. In one row the AI Engine is followed by a memory group, and in the next row the memory group is followed by an AI Engine within a tile.

```
location<graph>(tx_chain0) = bounding_box(0,0,3,3);
location<graph>(tx_chain1) = bounding_box(4,0,7,3);
location<graph>(tx_chain2) = bounding_box(0,4,3,7);

location<graph>(tx_chain1) = stamp(location<graph>(tx_chain0));
location<graph>(tx_chain2) = stamp(location<graph>(tx_chain0));
```

Other Constraints

Multi-Rate Designs

Some designs implement kernels which are based on various data frame size. The data can be transferred through streams and/or buffers. The repetition count is the number of times a kernel is launched during one iteration of the graph:

```
repetition_count(k) = N;
```

This sets `N` as the repetition count for the kernel `k`.

JSON Constraints

The constraints JSON file can contain one or more of the following sections:

NodeConstraints

Constrain graph nodes, such as kernels

PortConstraints

Constrain kernel ports and params

GlobalConstraints

Specify global constraints, i.e., constraints that are not associated with a specific object

Node Constraints

The `NodeConstraints` section is used to constrain graph nodes. Constraints are grouped by node, such that one or more constraints can be specified per node.

Syntax

```
{
  "NodeConstraints": {
    "<node name>": {
```

```

    <constraint>,
    <constraint>,
    ...
  }
}
}
<node name> ::= string
<constraint> ::= tile
    | shim
    | reserved_memory
    | colocated_nodes
    | not_colocated_nodes
    | colocated_reserved_memories
    | not_colocated_reserved_memories

```

Example

```

{
  "NodeConstraints": {
    "mygraph.k1": {
      "tile": {
        "column": 2,
        "row": 1
      },
      "reserved_memory": {
        "column": 2,
        "row": 1,
        "bankId": 3,
        "offset": 4128
      }
    },
    "mygraph.k2": {
      "tile": {
        "column": 2,
        "row": 2
      }
    }
  }
}

```

Node Names

Nodes must be specified by their fully qualified name, for example: <graph name>.<kernel name>.

In the following example, the graph name is myGraph and the kernel name is k1. The fully specified node name is myGraph.k1.

```

class graph : public adf::graph {
private:
    adf::kernel k1;
public:
    my_graph() {
        k1 = kernel::create(kernel1);
        source(k1) = "src/kernels/kernel1.cc";
    }
};
graph myGraph;

```

Whenever this kernel is referenced in the constraints JSON file it must be named myGraph.k1, as shown in the various examples throughout this document.

Tile Constraint

This constrains a kernel to a specific tile located at a specified column and row within the array. The column and row values are zero based, where the zeroth row is counted from the bottom-most compute processor and the zero-th column is counted from the left-most column.

Syntax


```
"tile": {
  "column": integer,
  "row": integer
}
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k1": {
      "tile": {
        "column": 2,
        "row": 1
      }
    }
  }
}
```

Shim Constraint

This constrains a node (PLIO or GMIO) to a specific AI Engine array interface, which is specified by column and channel. The column and channel are zero based. The channel is optional, and if omitted, the compiler selects the optimal channel.

 **Note:** PLIOs cannot be placed in every column. The availability of columns is device dependent. For example, columns 0-5 cannot be used for PLIO for the xcvc1902-vsua2197-2MP-e-S device. Refer to the relevant device data sheet for more information.

Syntax

```
"shim": {
  "column": integer,
  "channel": integer (optional)
}
```


Example

```
{
  "NodeConstraints": {
    "plioOut1": {
      "shim": {
        "column": 0,
        "channel": 1
      }
    },
    "plioOut2": {
      "shim": {
        "column": 1
      }
    }
  }
}
```

Reserved Memory Constraint

This constrains the location of system memory (stack and heap) for a kernel to a specific address on a specific tile. The address can be specified in one of two different ways:

- Column, row, bankId and offset, where the tile is specified by column, row. The bankId is relative to the tile and can be set to either 0, 1, 2, or 3. The offset address is relative to the bankId, and can be set to any value from 0 to 8192 which corresponds to the maximum number of bytes in a bank.
- Column, row, and bankId, where the bankId is relative to the tile and can be set to either 0, 1, 2, or 3.

 **Note:** The hardware views of each AI Engine memory is comprised of 8 banks of 128-bit width data. The aiecompiler views of each AI Engine memory is comprised of 4 banks of 256-bit width data.

Syntax

```
"reserved_memory": <bank_address>
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer,
  "offset": integer
}
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer
}
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k1": {
      "reserved_memory": {
        "column": 2,
        "row": 1,
        "bankId": 3,
        "offset": 4128
      }
    },
    "mygraph.k2": {
      "reserved_memory": {
        "column": 1,
        "row": 1,
        "bankId": 3
      }
    }
  }
}
```

Colocated Nodes Constraint

The colocated nodes constraint requires two or more kernels to be on the same tile and forces sequencing of the kernels in a topological order. It also allows them to share memory buffers without synchronization.

Syntax

```
"colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
```

```

        "colocated_nodes": ["mygraph.k1"]
    }
}

```

Non-colocated Nodes Constraint

This constrains two or more kernels to not be on the same tile.

Syntax

```

"not_colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string

```

Example

```

{
  "NodeConstraints": {
    "mygraph.k2": {
      "not_colocated_nodes": ["mygraph.k1"]
    }
  }
}

```

Colocated Reserved Memories Constraint

This constrains a kernel location to be on the same tile as that of one or more stacks. This ensures that the stacks can be accessed by the kernel without requiring a DMA.

Syntax

```

"colocated_reserved_memories": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string

```

Example

```

{
  "NodeConstraints": {
    "mygraph.k2": {
      "colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}

```

Non-colocated Reserved Memories Constraint

This constrains a kernel location so that it will not be on the same tile as the AI-Engine stack memory.

Syntax

```

"not_colocated_reserved_memories": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string

```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
      "not_collocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

Relative Constraints

Kernel, PLIO and GMIO objects can be placed relative to each other. These types of constraints are called relative constraints. The X and Y offsets indicates the relative column and row offsets between the source node and the relative node. Either X offset or Y offset should be specified. If X offset or Y offset is not specified, it is "don't care".

Syntax

```
{
  "NodeConstraints": {
    "<source node>" : {
      "relative_nodes": ["<destination node>:(X=-3,Y=-2)", "<destination node>:(X=1)", "<destination node>:(Y=2)", ...]
    }
  }
}

<source node> ::= string
<destination node> ::= string
```

Example

```
{
  "NodeConstraints": {
    "gr.k00": {
      "relative_nodes": ["gr.k01:(X=1,Y=1)", "gr.k10:(X=2,Y=2)", "gr.k11:(X=3,Y=-3)"]
    },
    "DatainA0": {
      "relative_nodes": ["Dataout00:(X=0)", "Dataout01:(X=-1)", "Dataout10:(X=-2)", "Dataout11:(X=3)"]
    },
    "gr.k00": {
      "relative_nodes": ["DatainA0:(X=0)", "DatainA1:(X=1)", "DatainB0:(X=2)", "DatainB1:(X=-3)"]
    },
    "DatainA0": {
      "relative_nodes": ["gr.k00:(X=0)"]
    }
  }
}
```

Port Constraints

Port constraints are specified in the PortConstraints section. Constraints are grouped by port, such that one or more constraints can be specified per port.

Syntax

```
{
  "PortConstraints": {
    "<port name>": {
      <constraint>[,
      <constraint>...]
    }
  }
}
```



```

}
<port name> ::= string
<constraint> ::= buffers
                | colocated_nodes
                | not_colocated_nodes
                | colocated_ports
                | not_colocated_ports
                | exclusive_colocated_ports
                | colocated_reserved_memories
                | not_colocated_reserved_memories

```

Example

```

{
  "PortConstraints": {
    "mygraph.k1.in[0]": {
      "colocated_nodes": ["mygraph.k1"]
    },
    "mygraph.k2.in[0]": {
      "colocated_nodes": ["mygraph.k2"]
    },
    "mygraph.p1": {
      "buffers": [{
        "column": 2,
        "row": 1,
        "bankId": 2
      }]
    }
  }
}

```

Port Names

Ports must be specified by their fully qualified name: <graph name>.<kernel name>.<port name>. In the following example, the graph name is myGraph, the kernel name is k1, and the kernel has two ports named in[0] and out[0] (as specified in kernel1.cc). The fully specified port names are then myGraph.k1.in[0] and myGraph.k1.out[0].

```

class graph : public adf::graph {
private:
    adf::kernel k1;
public:
    my_graph() {
        k1 = kernel::create(kernel1);
        source(k1) = "src/kernels/kernel1.cc";
    }
};
graph myGraph;


```

Anytime either of these ports are referenced in the constraints JSON file, they must be named myGraph.k1.in[0] and myGraph.k1.out[0], as shown in the various examples throughout this document.

Buffers Constraint

This constrains a data buffer to a specific address on a specific tile. The data buffer can be attached to an input, output, or inout port of a kernel or param (e.g., a lookup table). The address can be specified in one of three different ways:

- Column, row, and offset, where the tile is specified by column and row and the offset address is relative to the tile, starting at zero with a maximum value of 32768 (32k).
- Column, row, and bankId, where the bank ID is relative to the tile and can take values 0, 1, 2, 3.
- Offset, that can be between zero and 32768 (32k) and is relative to the tile allocated by the compiler.

 **Note:** One or two buffers can be constrained for a port.

Syntax

```
"buffers": [<address>, <(optional) address>]
<address> ::= <offset_address> | <bank_address> | <offset_address>
<tile_address> ::= {
  "column": integer,
  "row": integer,
  "offset": integer
}
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer
}
<offset_address> ::= {
  "offset": integer
}
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.out[0]": {
      "buffers": [{
        "column": 2,
        "row": 2,
        "offset": 5632
      }, {
        "column": 2,
        "row": 2,
        "offset": 4608
      }]
    },
    "mygraph.k1.out[0]": {
      "buffers": [{
        "column": 2,
        "row": 3,
        "bankId": 2
      }, {
        "column": 2,
        "row": 3,
        "bankId": 3
      }]
    },
    "mygraph.p1": {
      "buffers": [{
        "offset": 512
      }]
    }
  }
}
```

Colocated Nodes Constraint

This constrains a port (i.e., the port buffer) location to be on the same tile as that of one or more kernels. This ensures that the data buffer can be accessed by the other kernels without requiring a DMA.

Syntax

```
"colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k1.in[0]": {
      "colocated_nodes": ["mygraph.k1"]
    },
    "mygraph.k2.in[0]": {
      "colocated_nodes": ["mygraph.k2"]
    }
  }
}
```

Not Colocated Nodes Constraint

This constrains a port (i.e., the port buffer) location to not be on the same tile as that of one or more kernels.

Syntax

```
"not_colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_colocated_nodes": ["mygraph.k1"]
    }
  }
}
```

Colocated Ports Constraint

This constrains a ports buffer location to be on the same bank as that of one or more other port buffers. When two double buffers are co-located, this constraints both of the ping buffers to be on one bank and both of the pong buffers to be on another bank.

Syntax

```
"colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}
```

Not Colocated Ports Constraint

This constrains a port buffer location to not be on the same bank as that of one or more other port buffers.

Syntax

```

"not_colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string

```

Example

```

{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}

```

Exclusive Colocated Ports Constraint

This constrains a port buffer location to be exclusively on the same bank as that of one or more other port buffers, meaning that no other port buffers can be on the same bank.

Syntax

```

"exclusive_colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string

```

Example

```

{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "exclusive_colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}

```

Colocated Reserved Memories Constraint

This constrains a ports buffer location to be on the same bank as that of one or more stacks.

Syntax

```

"colocated_reserved_memories": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string

```

Example

```

{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}

```

Not Colocated Reserved Memories Constraint

This constrains a ports buffer location to not be on the same bank as that of one or more stacks.

Syntax

```
"not_collocated_reserved_memories": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example


```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_collocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

FIFO Constraint

This constrains a FIFO to a specific tile located at a specified column and row within the array. The tile can be an AI Engine tile, memory tile, or interface tile. The bankId is optional; if omitted, the compiler selects an optimal bank.

The FIFO constraint can be specified in one of the following ways:

- Column, row, bankID, offset and size, where the tile is specified by column, row, bankID, and the offset address is relative to the bankID, and the size starting at zero with a maximum value of 8188 32-bit words of a bank.
- Column, row, and bankId, where the bank ID is relative to the tile and can take values 0, 1, 2, or 3.

 **Note:** The hardware view is 8 banks of 128-bit width but the software view is 4 banks of 256 width.

The following code block shows the syntax.

```
"PortConstraints": [<fifo list>]
<fifo list> ::= <fifo type>[, <fifo type>...]
<fifo type> ::= <dma_fifos> | <stream_fifos>
<dma_fifos> ::= <aie_tile>
<aie_tile> ::= {
  "fifo_id": string,
  "tile_type": "core",
  "column": integer,
  "row": integer,
  "size": integer,
  "offset": integer,
  "bankId": integer (optional)
}
<stream_fifos> ::= {
  "fifo_id": string,
  "tile_type": "shim",
  "column": integer,
  "row": integer,
  "channel": integer
}
```

The following code shows an example.

```
{
  "PortConstraints": {
    "fifo_locations_records": {
      "dma_fifos": {
        "r1": {
          "tile_type": "core",
          "row": 0,
          "column": 0,
          "size": 16,
          "offset": 8,
          "bankId": 2
        }
      }
    }
  }
}
```

```

    },
    "r2": {
      "tile_type": "core",
      "row": 0,
      "column": 1,
      "size": 16,
      "offset": 9
    },
    "r4": {
      "tile_type": "mem",
      "row": 2,
      "column": 4,
      "size": 16,
      "offset": 6,
      "bankId": 2
    }
  },
  "stream_fifos": {
    "r3": {
      "tile_type": "shim",
      "row": 1,
      "column": 3,
      "channel": 1
    }
  }
},
"mygraph.k2.in[0]": {
  "not_colocated_nodes": ["mygraph.k1"],
  "fifo_locations": ["r1", "r2", "r3"]
},
"mygraph.k4.in[0]": {
  "fifo_locations": ["r1", "r2", "r4"]
}
}
}

```

Global Constraints

Global constraints are specified in the GlobalConstraints section.

Syntax

```

{
  "GlobalConstraints": {
    <constraint>[,
    <constraint>...]
  }
}
<constraint> ::= areaGroup
                | IsomorphicGraphGroup

```

Example

```

{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "root_area_group",
      "nodeGroup": ["mygraph.k1", "mygraph.k2"],
      "tileGroup": ["(2,0):(2,3)"],
      "shimGroup": ["0:3"]
    },
    "isomorphicGraphGroup": {
      "name": "isoGroup1",

```

```

    "referenceGraph": "clipGraph0",
    "stampedGraphs": ["clipGraph1", "clipGraph2"]
  }
}
}

```

Area Group Constraint

The area group constraint specifies a range of tile and/or shim locations to which a group of one or more nodes can be mapped. The area group constraint can be specified with the following properties.

groups

Specify the collection of groups. The groups can be:

tile-type

Specify the tile-type for the group. Supported tile-types are `aie_tile`, `shim_tile`, or `memory_tile`.

column_min

Column index for lower left corner of the group.

row_min

Row index for lower left corner of the group.

column_max

Column index for upper right corner of the group.

row_max

Row index for upper right corner of the group.

contain_routing

A boolean value that when specified true ensures all routing, including nets between nodes contained in the `nodeGroup`, is contained within the area group.

exclude_routing

A boolean value that when specified true ensures all routing, excluding nets between nodes from the `nodeGroup`, is excluded from the area Group.

exclude_placement


A boolean value that when specified true prevents all nodes not included in `nodeGroup` from being placed within the area group bounding box.

An AI Engine tile or memory tile range is in the form of `(column, row):(column, row)`, where the first tile is the bottom left corner and the second tile the upper right corner. The column and row values are zero based, where the zeroth row is counted from the bottom-most compute processor and the zeroth column is counted from the left-most column.

A shim range is in the form of `(column):(column)`, where the first value is the left-most column and the second value the right-most column. The column is zero based, where the zeroth row is counted from the bottom-most compute processor and the zeroth column is counted from the left-most column. The shim range also allows an optional channel to be specified, for example, `(column, channel):(column, channel)`.

The area group is used to exclude a range on the device from being used by the compiler for mapping and optionally routing as follows.

- To exclude a range from router, set `nodeGroup` and set `exclude_placement` to true.
- To exclude a range from mapper, set `nodeGroup` and set `exclude_routing` to true.
- To include all nets within a `nodeGroup` for router and mapper, set `nodeGroup` and set `contain_routing` to true.

 **Note:** There can be any number of area group constraints in the Global Constraints section, as long as each constraint has a unique name.

Syntax

```

"areaGroup": {
  "name": string,

  "nodeGroup": [<node list>], (*optional)
  "tileGroup": [<tile list>], (*optional)
  "shimGroup": [<shim list>], (*optional)
  "contain_routing": bool, (*optional)
  "exclude_routing": bool, (*optional)
  "exclude_placement": bool, (*optional)
}

```

```

<node list> ::= <node name>[,<node name>...]

<tile array> ::= <tile value>[,<tile value>...]
<tile value> ::= <tile range> | <tile address>
<tile range> ::= "<tile address>[:<tile address>]"
<tile address> ::= (<column>, <row>)

<shim array> ::= <shim value>[,<shim value>...]
<shim value> ::= <shim range> | <shim address>
<shim range> ::= "<shim address>[:<shim address>]"
<shim address> ::= (<column>[,<channel>])

<node name> ::= string
<column> ::= integer
<row> ::= integer
<channel> ::= integer

```

Example

```

{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "mygraph_area_group",
      "nodeGroup": ["mygraph.k1", "mygraph.k2"],
      "tileGroup": ["(2,0):(2,3)"],
      "shimGroup": ["0:3"]
    }
  }
}

```

Example of Exclude

```

{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "mygraph_excluded_area_group",

      "tileGroup": ["(3,0):(4,3)"],
      "shimGroup": ["3:4"],
      "exclude_routing": true
    }
  }
}

```

IsomorphicGraphGroup Constraint

The isomorphicGraphGroup constraint is used to specify isomorphic graphs that are used in the stamp and repeat flow.

Syntax

```

"isomorphicGraphGroup": {
  "name": string,
  "referenceGraph": <reference graph name>,
  "stampedGraphs": [<stamped graph name list>]
}

```

Example

```

"isomorphicGraphGroup": {
  "name": "isoGroup",

```



```

    "referenceGraph": "tx_chain0",
    "stampedGraphs": ["tx_chain1", "tx_chain2", "tx_chain3"]
}

```

General Description

The stamp and repeat feature of the AI Engine compiler can be used when the same graph has multiple instances that can be constrained to the same geometry in AI Engines. There are two main advantages to using this feature when the same graph is instantiated multiple times.

Small variation in performance

All graphs will have very similar throughput because buffers and kernels are mapped identically with respect to each other. Throughput might not be exactly identical due to differences in routing. However, it will be much closer than when stamping is not used.

Smaller run time of AI Engine compiler

Because the AI Engine compiler only solves a reference graph instead of the entire design, run time required will be significantly less than the default flow.

Capabilities and Limitations

If required, you are allowed to stamp multiple different graphs. For example, if a design contains four instances of a graph called tx_chain and four instances of rx_chain, then both sets of graphs can be independently stamped. This feature is only supported for designs which have one or more sets of isomorphic graphs, with no interaction between the different isomorphic graph sets. All reference and stamped graphs must have area group constraints. You must declare identical size area groups for each instance of the graph that needs to be stamped. All area groups must be non-overlapping. For example:

```

"areaGroup": {
  "name": "ant0_cores",
  "nodeGroup": ["tx_chain0*"],
  "tileGroup": ["(0,0):(3,3)"]
},
"areaGroup": {
  "name": "ant1_cores",
  "nodeGroup": ["tx_chain1*"],
  "tileGroup": ["(0,4):(3,7)"]
},

```

Note: The node group must contain all node instances in the graphs to be stamped. Pattern matching can be used as in shown in the previous example.

You must declare an `isomorphic_graph_group` in the constraints file that specifies the reference graph and the stamped graphs. For example:

```

"isomorphicGraphGroup": {
  "name": "isoGroup",
  "referenceGraph": "tx_chain0",
  "stampedGraphs": ["tx_chain1", "tx_chain2"]
}
,

```

In this case, the tx_chain0 graph is the reference and its objects will be placed first and stamped to graph tx_chain1 and tx_chain2. Area groups must follow these rules for number of rows: the number of rows for all identical graphs (reference + stamp-able ones) must be the same, and must begin and end at the same *parity* of row, meaning if the reference graph's tileGroup begins at an even row and ends at an odd row, then all of the stamped graphs must follow the same convention. This limitation occurs because of the mirrored tiles in AI Engine array. In one row, the AI Engine is followed by a memory group and in the next row the memory group is followed by an AI Engine within a tile.

Recommended: To add absolute, co-location, or other constraints to your design, only add constraints to the reference graph and all these constraints will automatically be applied to the stamped graphs.

!! Important: Only top-level graphs can be stamped. You cannot instantiate a single graph at the top level and stamp graphs at a lower level of a hierarchy.

Using the Restrict Keyword in AI Engine Kernels

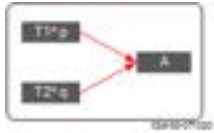
The use of restrict keyword (`__restrict`) is permitted in the AI Engine kernel C++ code. This appendix highlights AMD recommendations for

using the restrict keyword in the context of AI Engine kernel code.

Pointer Aliasing

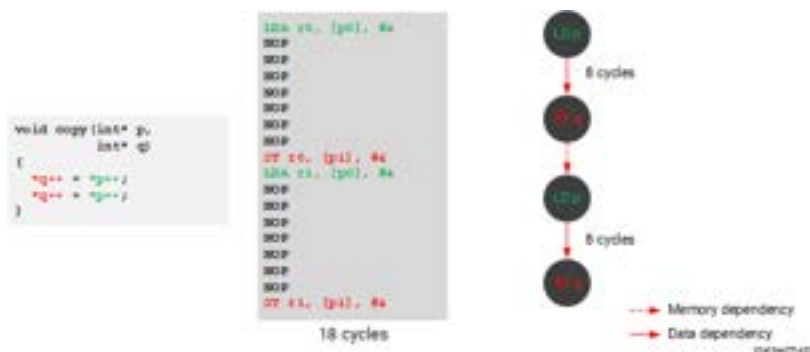
Pointer aliasing refers to the situation where the same memory location can be accessed using different pointer names. The strict aliasing rule in C/C++ means that pointers are assumed not to alias if they point to fundamentally different types. Aliasing introduces strong constraints on program execution order. The following shows the aliasing of p and q.

Figure: Pointer Aliasing



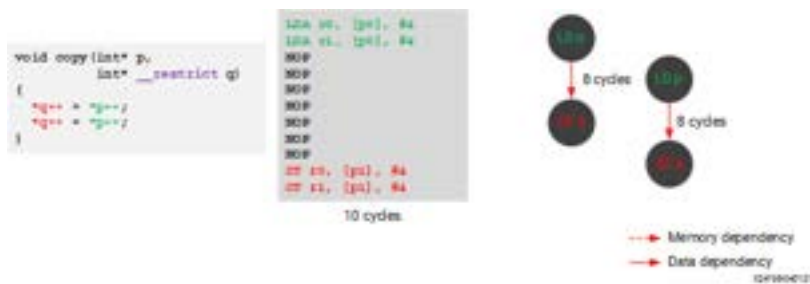
The following is an example of pointer aliasing, in which both the pointers p and q point to the same address. The assembly language code produced by the compiler is shown in the middle column, and the operations and clock cycles are shown on the right.

Figure: Aliasing Code Example



By adding the restrict keyword into this code example, the compiler can optimize the resulting assembly language to increase parallelization of the operations in hardware. The following example shows that using the restrict keyword to prevent aliasing uses fewer clock cycles to complete the same operation.

Figure: Use of Restrict Keyword to Avoid Aliasing



Memory Dependencies

Memory dependencies in the code can limit the kinds of optimizations attempted by the compiler. For example in the following code, xyz and pointers p and q might be unrelated. However, within the function code both pointer p and pointer q point to same global variable xyz. The compiler must guarantee the correct execution under both these conditions. Due to these kinds of memory dependencies the compiler needs to be conservative and limit optimizations.

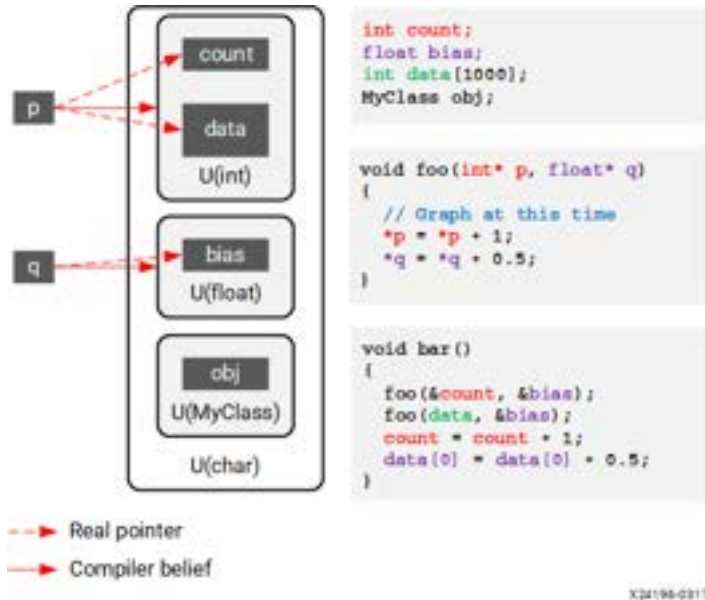
Figure: Unrelated Pointers

```
int xyz;
void foo(int *p, int *q)
{
    *p=xyz;
    *q=xyz;
}
```

Strict Aliasing Rule

The strict aliasing rule dictates that pointers are assumed not to alias if they point to fundamentally different types, except for `char*` and `void*` which can alias to any other data type. This is shown in the following graphic which shows the object universes and the associated pointers.

Figure: Object Universes



Pointers are associated with a type universe U(T)

T is the template and in the preceding graphic the various templates are shown, including an `int` universe and a `float` universe; there is also a `MyClass` universe per design. Additionally there is a `char` universe that includes all universes by default.

Universes do not alias

Pointer `p` can only point to any address within the `int` universe whereas pointer `q` can only point to any address within the `float` universe. Because of this pointer `p` and pointer `q` cannot be aliased.

Derived pointers point to the original universe

Pointers derived from a restrict pointer are considered restrict pointers and point to the same restricted memory region. See [Derived Pointers](#).

char* universe contains all universes

A `char` pointer can point to any variable in all universes.

For two pointers of the same type, as in the following, where both `p` and `q` are `int`, the compiler is conservative and aliasing is applied, resulting in loss of performance.

Figure: Loss of Performance




For two pointers of different types, as in the following example, where `p` is an `int` and `q` is `float`, the compiler applies the strict aliasing rule and an undefined behavior occurs if aliasing exists.

Figure: Two Pointers of Different Types



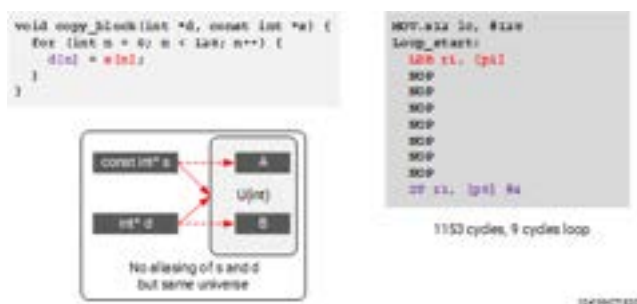
Restrict Keyword

The restrict keyword is mainly used in pointer declarations as a type qualifier for pointers. It does not add any new functionality. It allows you to tell the compiler about a potential optimization. Using `__restrict` with a pointer informs the compiler that the pointer is the only way to access the object pointed at, and the compiler does not need to perform any additional checks.

 **Note:** If a programmer uses the restrict keyword and violates the above condition, undefined behavior can occur.

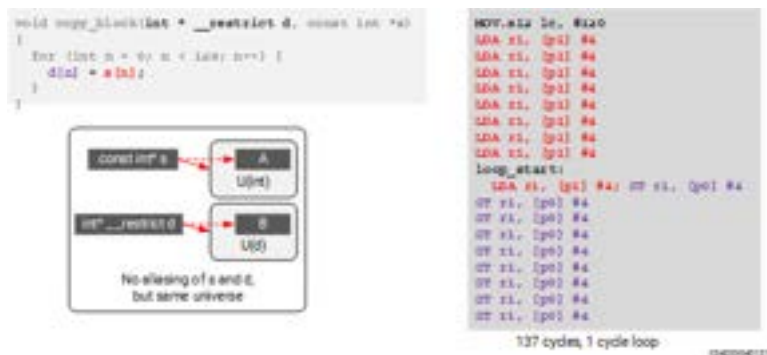
The following is another example with pointers that, by default, have no aliasing.

Figure: No Aliasing Example



Apply the restrict keyword for performance improvement. The following example shows no memory dependencies with other pointers.

Figure: No Memory Dependencies with Other Pointers



Restrict Qualification

The C standard provides a specific pointer qualifier, `__restrict`, intended to allow more aggressive compiler optimization by explicitly stating data independence between whatever the pointer references and all other variables. For example:

```
int a; // global variable
void foo(int* __restrict p, int* q)
{
    for (...) { ... *p += a + *q; ...}
}
```

Now the analysis of `foo` can proceed with the knowledge that `*p` does not denote the same object as `*q` and `a`. So, `a` and `*q` can now be loaded once, before the loop.

Currently, the compiler front end does not disambiguate between different accesses to the same array. So, when updating one element of an array, it assumes that the complete array has changed value. The `__restrict` qualifier can be used to override this conservative assumption. This is useful when you want to obtain multiple independent pointers to the same array.

```
void foo(int A[])
```

```

{
  int* __restrict rA = A; // force independent access
  for (int i = ...)
    rA[i] = ... A[i];
}


```

In this example, the `__restrict` qualifier allows software pipelining of the loop; the next array element can already be loaded, while the previous one must still be stored. To maximize the impact of the `__restrict` qualifier, the compiler front end, by default, inserts a `chess_copy` operation in the initializer, as if was written:

```
int* __restrict rA = chess_copy(A);
```

This is needed to keep both pointers distinct within the optimizer (for example, no common subexpression elimination). This behavior can be disabled for the AI Engine compiler by means of the option `-mllvm -chess-implicit-chess_copy=false`. So, the `chess_copy` creates two pointers, while `__restrict` informs the compiler not to consider any mutual dependencies between the stores/loads through these pointers. For `__restrict` pointers having a local scope, the mutual independence assumption only holds during the lifetime of the `__restrict` pointer.

Pointers derived from a `__restrict` pointer (such as `rA+1` or through pointer intrinsics) keep the restriction, that is, they are considered to point to the same restricted memory region.

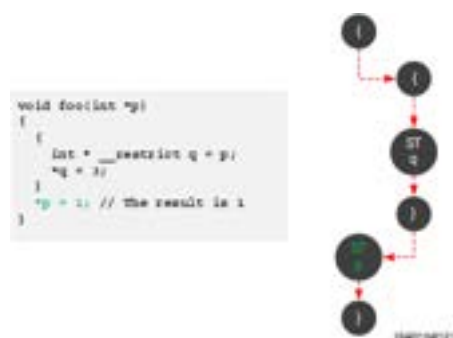
 **Note:** Details of `chess_copy` is available from the *ASIP Programmer Chess Compiler User Manual*, which can be found in the [AI Engine Lounge](#).

Undefined Behavior

Using the `restrict` keyword improves performance as shown in the previous topic. However, there are issues if the keyword is used inappropriately. The `__restrict` child pointers *must* be used in a different block-level scope than the parent pointers, such as pointer `p` and `q` as shown in the following example.

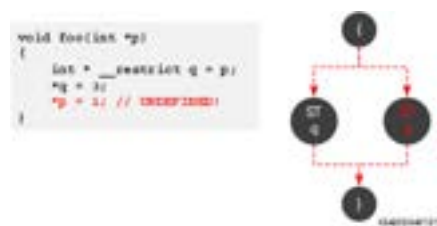
Working Example 1

Figure: Use of Restrict Keyword



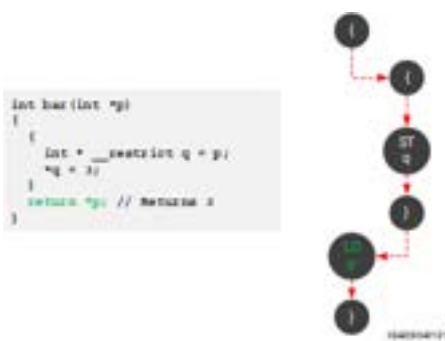
Use of parent pointers in the same scope might break the `__restrict` contract which produces an undefined behavior, such as pointers `p` and `q` in the following example.

Figure: Undefined Behavior

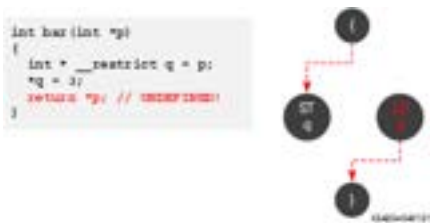


Working Example 2

This can also happen during the load operation, as shown in the green text (`return *p;`) in the following figure.

Figure: Load Operation

The undefined behavior occurs when the restrict pointers are used within the same scope, such as pointers `p` and `q` in the following example.

Figure: Restrict Pointers in Same Scope

Working Example with Inline Function

The following code shows the working inline function call, in which pointer `p` and pointer `q` are used in different scopes.

Figure: Inline Function Calls

The undefined behavior occurs when the restrict pointers are used within the same scope, such as pointers `p` and `q` in the following example.

Figure: Inline Function Calls in Same Scope

Scope of Restrict Keyword in Inline Function

When there are no other accesses within the scope, declaring the restrict pointer has no performance benefits.

Figure: Working Example with No Performance Benefits

```

inline int read(int *p) {
    int * __restrict q = p;
    return *q;
}

inline void write(int *p, int d) {
    int * __restrict q = p;
    *q = d;
}

int foo(int *p1, int *p2) {
    write(p1, 4);
    return read(p2);
}

```

154279421

In a special case, you can have non-aliasing accesses, as in the following example. Here the parent pointer, *p*, is used but points to a different location and therefore this is acceptable.

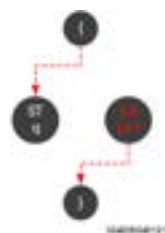
Figure: Special Case—Non-aliasing Accesses

```

int bar(int *p)
{
    int * __restrict q = p;
    *q = 3;
    return *(q+1); // OK!
}

```

--- Memory dependency
 --- Data dependency



154279421

Benefits of Using the Restrict Keyword for Read/Modify/Write Loops

The following example works without the restrict keyword, but has poor performance.

Figure: Example Without Restrict Keyword

```

void inc(int* p)
{
    for (int n = 0; n < 128; n++)
    {
        p[n] = p[n] + 1;
    }
}

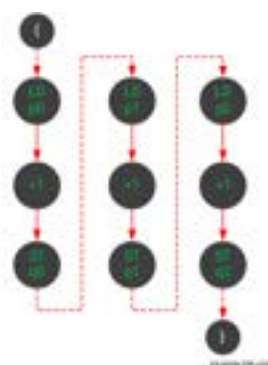
```

```

MOV.x16 rc, #128
loop_start:
    LDA r1, [p0], #4
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    ADD r0, r1, #1
    ST r0, [p0], #4

```

10 cycles/iteration



154279421

Adding the restrict keyword allows every iteration to access a different location where there is no aliasing between iterations (`__restrict`) and aliasing within iterations preserved by data dependency. The increased parallelization results in improved performance.

Figure: Add Restrict Keyword

```

void inc(int* p)
{
    int * __restrict q = p;
    for (int n = 0; n < 128; n++)
    {
        q[n] = q[n] + 1;
    }
}

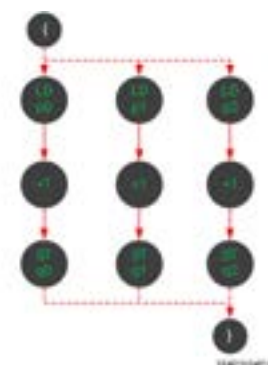
```

```

loop_start:
    LDA r1, [p0], #4
    ST r0, [p0], #4
    ADD r0, r1, #1

```

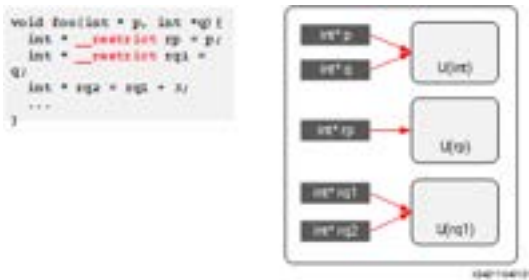
1 cycle/iteration



154279421

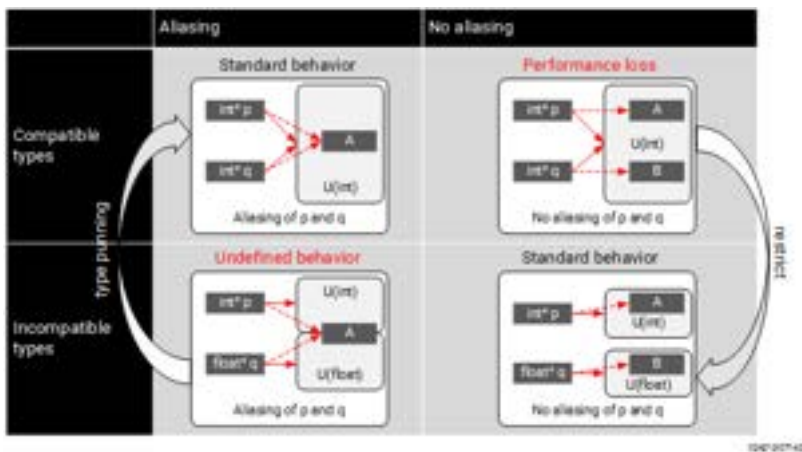
Derived Pointers

Pointers derived from a restrict pointer are considered restrict pointers and point to the same restricted memory region, as shown in the following example, where *rq2*, derived from *rq1* (defined as a restrict pointer) is also a restrict pointer and points to the same universe.

Figure: Pointers to Same Restricted Memory Region

Summary

Proper use of the restrict keyword (`__restrict`) in AI Engine kernel programming can result in performance gains and eliminate undefined behaviors in your code. However, be aware that when assigned to the same scope, the restrict pointers might result in undefined behavior in your design.

Figure: Restrict Keyword Use Summary

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to <https://docs.amd.com>.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select Help > Documentation and Tutorials.
- On Windows, click the Start button and select Xilinx Design Tools > DocNav.
- At the Linux command prompt, enter docnav.

Note: For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and

address frequently asked questions. To access the Design Hubs:

- In DocNav, click the Design Hubs View tab.
- Go to the [Design Hubs](#) web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Support](#).

References

These documents provide supplemental material useful with this guide:

1. *Versal Adaptive SoC AI Engine Architecture Manual* ([AM009](#))
2. *AI Engine Tools and Flows User Guide* ([UG1076](#))
3. *AI Engine Intrinsic User Guide* ([UG1078](#))
4. *AI Engine-ML Kernel and Graph Programming Guide* ([UG1603](#))
5. *Data Center Acceleration using Vitis* ([UG1700](#))
6. *Embedded Design Development Using Vitis* ([UG1701](#))
7. *Vitis Reference Guide* ([UG1702](#))
8. *AI Engine API User Guide* ([UG1529](#))
9. *Vitis Model Composer User Guide* ([UG1483](#))
10. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* ([PG313](#))
11. *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* ([DS957](#))

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/04/2025 Version 2025.1	
Memory and DMA Programming	Added chapter on local tile memory and DMA programming.
AI Engine Local Memory Access	Added section on local tile memory access.
Tiling Parameters and Buffer Descriptors	Added section on tiling parameters and buffer descriptors on an input or output port of an AIE memory DMA.
Tiling Parameters Specification	Added section to describe tiling parameter specification in the graph with regard to an input or output port of an AIE memory DMA.
Viewing Tiling Parameters in the Vitis IDE	Updated figure to show tile memory module tiling parameters table in Vitis IDE.
PLIO and GMIO Location Constraints	Added section on PLIO and GMIO location constraints.
11/28/2024 Version 2024.2	
Synchronous Buffer Port Access	Clarified the behavior of locks on a synchronous output buffer, during multiple iterations of a kernel.
Circular Output Buffer	Clarified buffer addressing limitations when multiple kernels are mapped to the same tile.
Global Graph-Scoped Tables	Added this topic.
Explicit Packet Switching	Clarified that packet stream can be created from one or more AI Engine kernels to one or more multiple destination AI Engine kernels.
Packet Switching Graph Constructs	Clarified that data read data from a packet stream is read as an integer value by default, but that can then can be cast to a different data type if desired.

Section	Revision Summary
Example of Packet Switching between PL and AI Engine , Example of Packet Switching from an AI Engine to Multiple AI Engines , and Example on Packet Switching from Multiple AI Engines to an AI Engine	Added examples of how to set up packet switching connections between PL and AI Engine.
Logical I/O Ports	Added topic that explains the usage of the graph port objects.
06/05/2024 Version 2024.1	
General	Removed <code>--disable-multirate</code> option on the AI Engine compiler.
	Changed <code>output_stream</code> to <code>output_cascade</code> , and changed <code>input_stream</code> to <code>input_cascade</code> .
	Changed <code>writeincr_v<n></code> to <code>writeincr</code> .
	Updated to reflect that <code>bfloat16</code> is supported only on AI Engine ML devices.
	Removed unsupported data types: <code>cacc32</code> , <code>aac40</code> , <code>cacc40</code> , <code>acc56</code> , <code>cacc56</code> , <code>acc72</code> , <code>cacc72</code> , <code>accfloat</code> , and <code>caccfloat</code> .
Scalar Programming	Added note that GNU defined types work for x86 simulation but not for AI Engine simulation.
Chess Directives and C++ Attributes	Provides Chess directives and the equivalent C++ attribute.
Parallel Streams Access	Added recommendation that when resource annotations are used on streams, to use the annotation consistently on the same stream.
Data Reshaping	Added that the AI Engine API provides functions to extract real or imaginary parts from complex scalar or vector data.
Casting and Datatype Conversion	Added details about the API supporting conversions between vector data types.
Packet Processing	Added <code>getPacketId</code> API example.
Packet Switching Graph Constructs	Added examples to illustrate packet merge, and packet split APIs. Also, added an example of using floating point data.
Examples	Added warning note that two kernels connected by buffers should have both linear or circular addressing if the two kernels are in the same tile.
Throughput Measurements Using Event APIs	Changed <code>end_profiling</code> to <code>stop_profiling</code> .
Vector Arithmetic Operations	Added <code>aie::saturating_add</code> , and <code>aie::saturating_sub</code> . Added example code to show the difference between <code>aie::add</code> and <code>aie::saturating_add</code> .
Design Considerations for Graphs Interacting with Programmable Logic	Added Conceptual Representation of AIE-PL Interface.
Event API	Moved Event API section from Appendix C in UG1079 to Appendix A in UG1076.
12/04/2023 Version 2023.2	
General	Updated to reflect the usage of the new unified AMD Vitis™ IDE.
Casting and Datatype Conversion	Added more information about scalar floating-point operations options.
Inline Keywords	Added <code>inline</code> keyword explanation.

Section	Revision Summary
Input and Output Buffers	Specified maximum single buffer port size.
Asynchronous Buffer Port Access	Added important note that operations on asynchronous buffer should be done after the buffer is acquired.
Stream Data Types	Added supported cascade accumulator data types.
10/18/2023 Version 2023.2	
Buffer Port Connection for Multirate Processing	Added code example to accompany the Up-Converter Followed by a Single Rate Kernel figure.
Viewing Loop II in the Vitis IDE	Added new section.
Using Vitis Unified IDE and Reports	Updated for the Vitis unified IDE.
Design Flow Using RTL Programmable Logic	Added the PL Kernels Inside AI Engine Graph section.
AI Engine to PL Interface Text Input Format	Updated section with new examples.
Examples	Added the Using Input and Output Buffer as Intermediate Storage section.
Creating a Data Flow Graph (Including Kernels)	Added important note that the main function must have a return statement. Otherwise, <code>aiecompiler</code> will error out.
Introduction to Scalar and Vector Programming, AI Engine Data Types, and Vector Registers	Updated instances of windows to input and output buffers.
06/23/2023 Version 2023.1	
General	Editorial update; no technical update.
05/23/2023 Version 2023.1	
General	Editorial update; no technical update.
05/16/2023 Version 2023.1	
General	Updated window to buffer, as in <code>input_window</code> to <code>input_buffer</code> and <code>output_window</code> to <code>output_buffer</code> .
Vector Data Types	Added <code>uint16</code> and <code>uint32</code> to Supported Vector Types and Sizes table.
Input and Output Buffers	Added chapter to introduce the input and output buffers. These buffers are a replacement for windows, which are deprecated in 2023.1.
Introduction to Scalar and Vector Programming	Updated code examples to use buffers throughout the chapter.
AI Engine API Overview	Added <code>aie::print_matrix</code> .
Iterators	Added <code>aie::begin_restrict_vector</code> and <code>aie::cbegin_restrict_vector</code> .
Example Designs Using the AI Engine API	Updated for the 2023.1 release and to use buffers.
Recommended Project Directory Structure	Added guidance that only one source file is allowed to be specified as kernel source via <code>adf::source</code> .
Streaming Data API	Updated for the 2023.1 release.
Reading and Advancing an Input Stream and Writing and Advancing an Output Stream	Added note about how to indicate the end of the stream, with command example.
Packet Split and Merge Connections	Added new section.
Multicast Support	Added note to indicate that the compiler supports multirate processing for multicast connections.

Section	Revision Summary
Conditional Ports	Added section to describe the compiler feature that allows for conditional ports on template functions.
Array of Graph Objects	Added section to describe how to conditionally instantiate an array of graph objects.
Graph Programming Model	Updated code snippets, and screenshots throughout chapter.
Data Throughput Estimate in Hardware	Added new section to describe ways to measure interface throughput of the AI Engine in hardware.
Throughput Measurement Using Timers	Added new section to describe the way to measure interface throughput using timers.
Throughput Measurements Using Event APIs	Added new section to describe the way to measure interface throughput using events.
Throughput Measurements using Event Trace	Added new section.
Connection Constructor Templates	Added new section that describes how to specify templated connect statements in a graph..
Design Analysis and Programming using Intrinsic	Added note about <code>input_window</code> and <code>output_window</code> being deprecated, and when these are found in code examples in this appendix, to use <code>input_buffer</code> and <code>output_buffer</code> instead.
10/19/2022 Version 2022.2	
Document title	Changed title to <i>AI Engine Kernel and Graph Programming Guide</i> .
Document-wide changes	Added graph programming content from <i>AI Engine Tools and Flows User Guide</i> (UG1076).
Vector Arithmetic Operations	Updated examples syntax.
Rounding and Saturation Modes	Changed <code>truncate</code> to <code>saturate</code> .
Casting and Datatype Conversion	Updated the conversion functions (<code>to_float()</code> and <code>to_fixed()</code>) example.
Operator Overloading	Updated operators example code.
Data Reshaping	Updated to include <code>aie::vector<int32,8></code> .
Matrix Multiplication	Updated section for the release.
Window-Based Access	Made clarifying updates.
Stream-Based Access	Added Stream Connection for Multi-Rate Processing subsection.
Relative Constraints	New topic.
MAC on 16x16 bits	Made clarifying updates.
Relative Constraints	New topic.
Graph Topologies	New section.
Other Constraints	Added <code>async_repetition</code> constraint.
05/25/2022 Version 2022.1	
Vector Data Types	Highlighted data type sizes supported natively by the AI Engine.
Vector Registers	Added the <code>grow_replicate</code> function on <code>aie::vector</code> .
Loops	Added Loop Flattening and Unrolling section.
Scheduling Separator	Added information about scheduling separator pragma.

Section	Revision Summary
Parallel Streams Access	Added information about accessing two input and/or output streams in parallel.
Profiling Kernel Code	Added information about profiling kernel code using cycles() API.
Runtime Parameter Specification	Removed section about AI Engine-to-AI Engine runtime parameter support.
Data Communication via AXI4-Stream Interconnect	Updated the section.
Buffer vs. Stream in Data Communication	Updated section to reflect window multicast support.
Example Designs Using the AI Engine API	New chapter. Added new FIR Filter and Matrix Multiplication example designs.
Update, Extract, and Shift	Added information about using the update API on accumulators.
11/10/2021 Version 2021.2	
Scalar Processing Unit	Updated for AI Engine API.
AI Engine Memory	Added information about heap and stack size.
AI Engine API	New section.
Introduction to Scalar and Vector Programming	Updated for AI Engine API.
AI Engine API Overview	New section.
Vector Arithmetic Operations	
Vector Reduction	
Bitwise Operations	
Data Comparison	
Data Reshaping	
Iterators	
Operator Overloading	
Multiple Lanes Multiplication - sliding_mul	
Matrix Multiplication - mmul	
API Operation Examples	
Loops	Updated information.
Floating-Point Operations	Updated for AI Engine API.
Single Kernel Programming using Intrinsics	Appendix describing programming using intrinsics.
Design Analysis and Programming using Intrinsics	Appendix describing design analysis and programming using intrinsics.
07/19/2021 Version 2021.1	
Accumulator Registers	Added information about print acc value, as well as streaming data APIs.
Casting and Datatype Conversion	Added a note about the AI Engine floating-point.
Initialization	Added information about the static keyword.
Load and Store with Virtual Resource Annotations	Added new section.
Buffer vs. Stream in Data Communication	Added information.

Section	Revision Summary
DDR Memory Access through GMIO	Removed information related to PL GMIO.
Mapping Algorithm onto the AI Engine	Clarified description.
Coding with Intrinsics	Added information about the <code>(always_inline)</code> attribute.
02/04/2021 Version 2020.2	
Initial release.	N/A

Please Read: Important Legal Notices

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

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

© Copyright 2021-2025 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Versal, Vitis, Vivado, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.