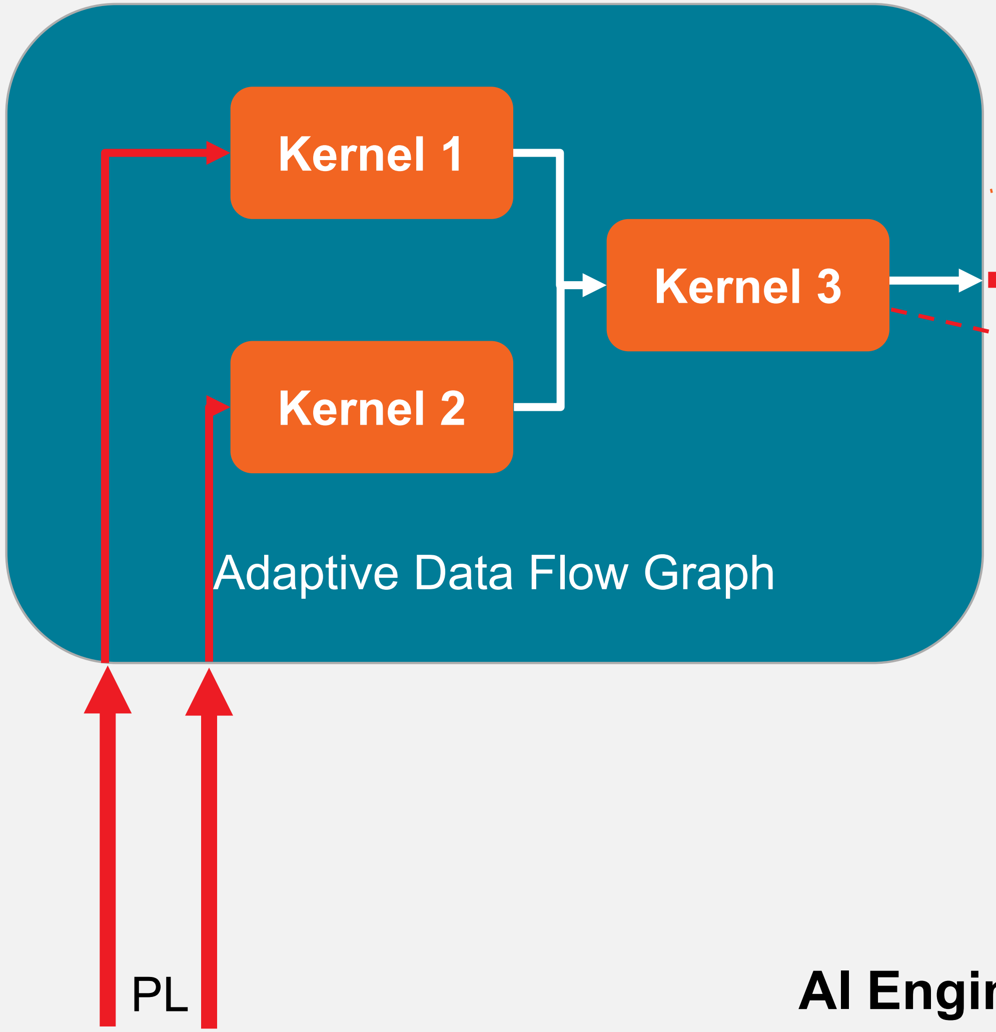
AI Engine Programming: Kernels and Graphs

2024.1

AMD
together we advance_

Programming the AI Engine



Adaptive Data Flow Graph (ADF)

- Written in C++
- Describes the kernel connectivity

C/C++ {</>}

AI Engine Kernels

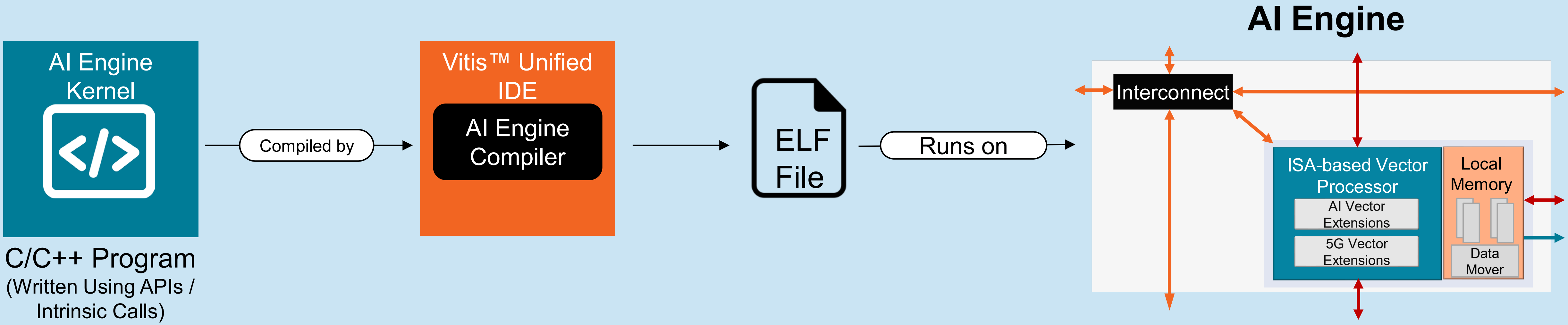
- C/C++ computation function
- I/O buffer or stream arguments for graph connectivity
- Compiled for one AI Engine
- Multiple kernels can run on a single AI Engine

PL

AI Engine Application

AMD
together we advance_

AI Engine Kernel Programming



Recommended:

- Use AI Engine APIs for your designs
- Usage of intrinsics must only be considered for situations where performance is required (not covered by AI Engine APIs)

Programming the AI Engine Kernel

AI Engines are an array of VLIW processors with SIMD vector units

Programming the AI Engine array requires understanding of:

- Algorithm to be implemented
- Capabilities of the AI Engines
- Overall data flow between individual functional units

AI Engine array supports three levels of parallelism:

SIMD

Multiple elements to be computed in parallel

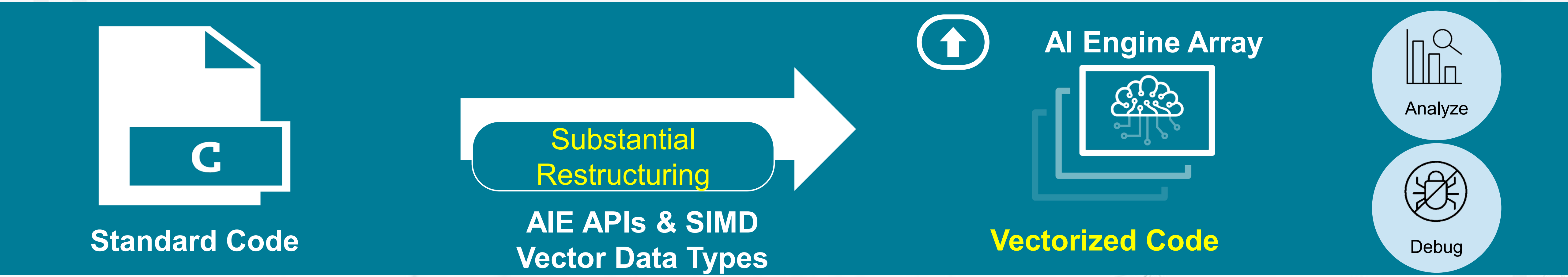
Instruction-level Parallelism

Multiple instructions to be executed in a single clock cycle

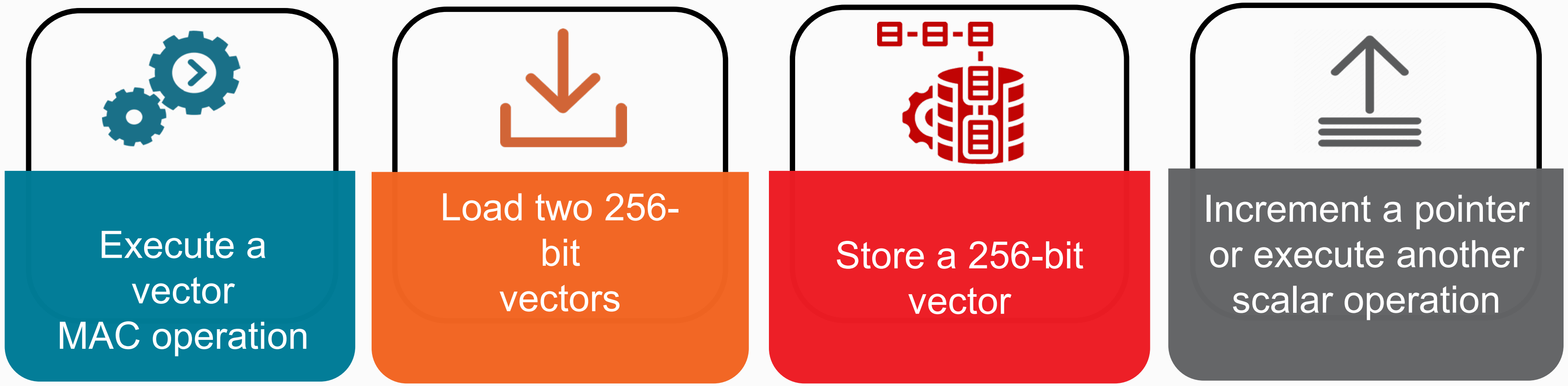
Multicore

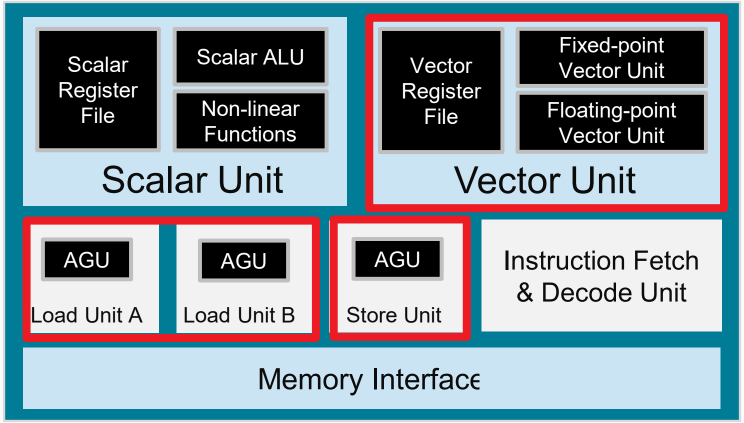
Multiple AI Engines can execute in parallel

Programming the AI Engine Kernel



Power of an AI Engine is its ability to:





AI Engine compiler does not perform any auto or pragma-based vectorization

For more information on scalar and vector data types, see the ***AI Engine Kernel and Graph Programming Guide* (UG1079)**

AI Engine APIs (Based on C++)

AI Engine API – A portable programming interface for AI Engine kernel programming

AI Engine API interface:

- Targets current and future AI Engine architectures
- Provides parameterizable data types that enable generic programming
- Implements the most common operations in a uniform way
- Easier programming interface as compared to using intrinsic functions

AI Engine APIs are implemented as a C++ header-only library that is translated into optimized intrinsic functions

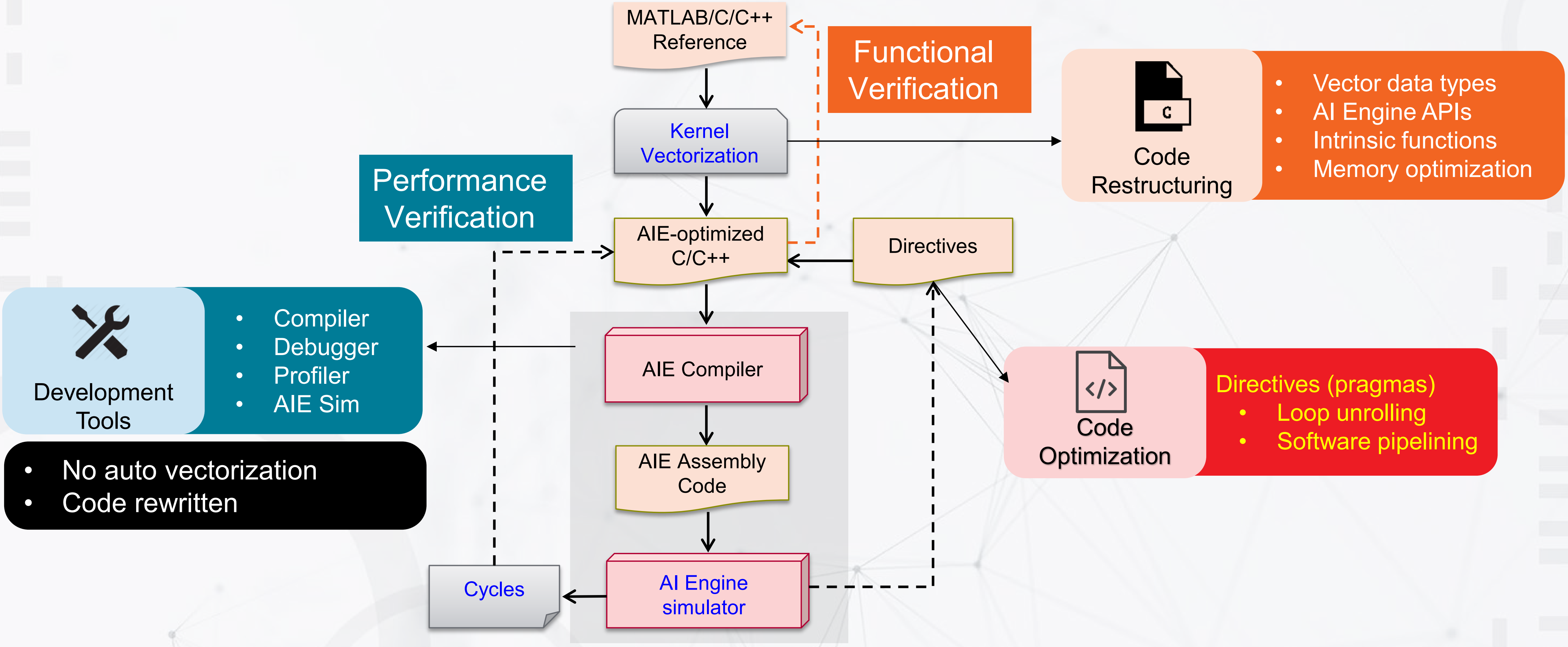
```
#include "aie_api/aie.hpp"
#include "aie_api/aie_adf.hpp"
```

C++

For more details: *AI Engine API User Guide* (UG1529)

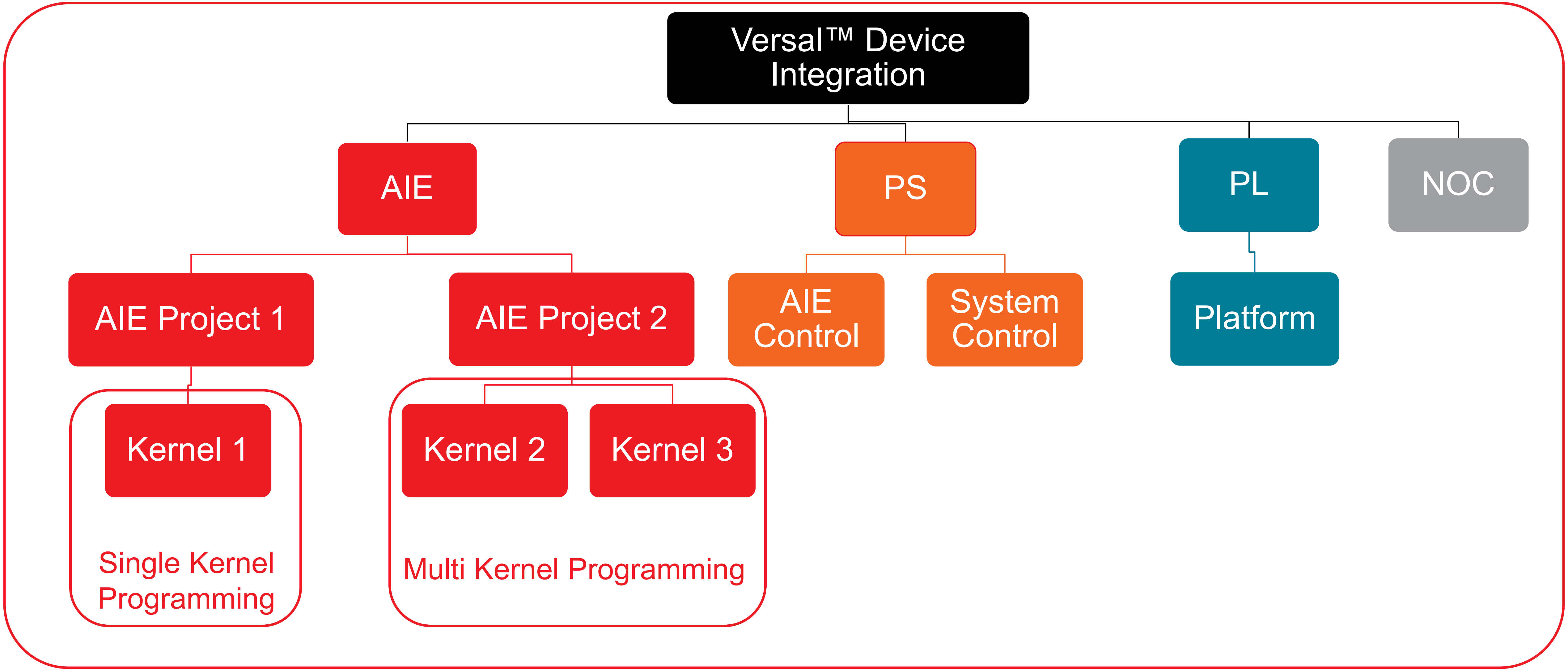
For advanced users that need programming with intrinsics: *AI Engine Intrinsics User Guide* (UG1078)

AI Engine Programming Flow



AMD together we advance_

Single and Multiple Kernel Programming



Kernel Function Program

Example: Single kernel function “weighted sum” using scalar data types

```
#include <adf.h>

void weighted_sum_with_margin(input_buffer<int32,extents<256>,margin<8>> & in,
                               output_buffer<int32,extents<256>> & out)
{
        auto inIter=aie::begin_random_circular(in);
        auto outIter=aie::begin_random_circular(out);

        inIter+=8;

        for (unsigned i = 0; i < WLEN; i++)
        {
                int32 val;
                int32 wsum = 0;

                for (unsigned j = 1; j <= 8; j++)
                {
                        val=*inIter++;
                        wsum = wsum + (j * val);
                }
                *outIter++=wsum;
        }
}
```

Adaptive Data Flow library

A kernel is a C/C++ function using special I/O and scalar data types — it will be launched automatically by a scheduler depending on data arrival

Buffer port iterators to access data

Scalar data types for computations

Computation function

Kernel Function: Code Restructuring and Directives

```
#include <adf.h>
#include "../include.h"
#include "aie_api/aie.hpp"
#include "aie_api/aie_adf.hpp"
using namespace adf;

alignas(aie::vector_decl_align) int32 weights[8] =    { 1, 2, 3, 4, 5, 6, 7, 8};

void vectorized_weighted_sum_with_margin(input_buffer<int32,extents<256>,margin<8>> & restrict in,
                                         output_buffer<int32,extents<256>> & restrict out)
{
        auto inIter=aie::begin_vector<8>(in);
        auto outIter=aie::begin_vector<8>(out);
        aie::vector<int32,8> coeffs = aie::load_v<8>(weights);

        aie::vector<int32,16> data;
        aie::accum<acc80,4> acc;
        aie::vector<int32,8> dataout;
        data.insert(0,*inIter++);

        for(unsigned i=0; i<WLEN/16; i++)
                chess_prepare_for_pipelining
                chess_loop_range(4,32)
        {
                data.insert(1, *inIter++);
                acc = aie::sliding_mul<4,8>(coeffs, 0, data, 1);
                dataout.insert(0,acc.to_vector<int32>());

                acc = aie::sliding_mul<4,8>(coeffs, 0, data, 5);
                dataout.insert(1,acc.to_vector<int32>());
                *outIter++=dataout;

                data.insert(0, *inIter++);
                acc = aie::sliding_mul<4,8>(coeffs, 0, data, 9);
                dataout.insert(0,acc.to_vector<int32>());

                acc = aie::sliding_mul<4,8>(coeffs, 0, data, 13);
                dataout.insert(1,acc.to_vector<int32>());
                *outIter++=dataout;
        }
}
```

Adaptive Data Flow library

AI Engine API headers

A kernel is a C/C++ function using special I/O and vector data types — it will be launched automatically by a scheduler depending on some events

Buffer port iterators to access data

Vector data types for vectorized computations

Directives to help in scheduling for optimal performance

AI Engine APIs to perform vectorized computation

Example: Scalar and Vector Programming

Scalar Programming	Vector Programming

Scalar Programming

```
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;
    }
}
```

Vector Programming

```
void vect_mul(input_buffer<int32>& __restrict data1,
              input_buffer<int32>& __restrict data2,
              output_buffer<int32>& __restrict out)
{
    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<NUM_SAM/8;i++)
        chess_prepare_for_pipelining
        {
            auto va=*inIter1++;
            auto vb=*inIter2++;
            auto vt=aie::mul(va,vb);

            *outIter++=vt.to_vector<int32>(0);
        }
}
```

Cycles: 1055

Cycles: 99

Performance: 10x speed up, 8x throughput (int 32 multiplication)

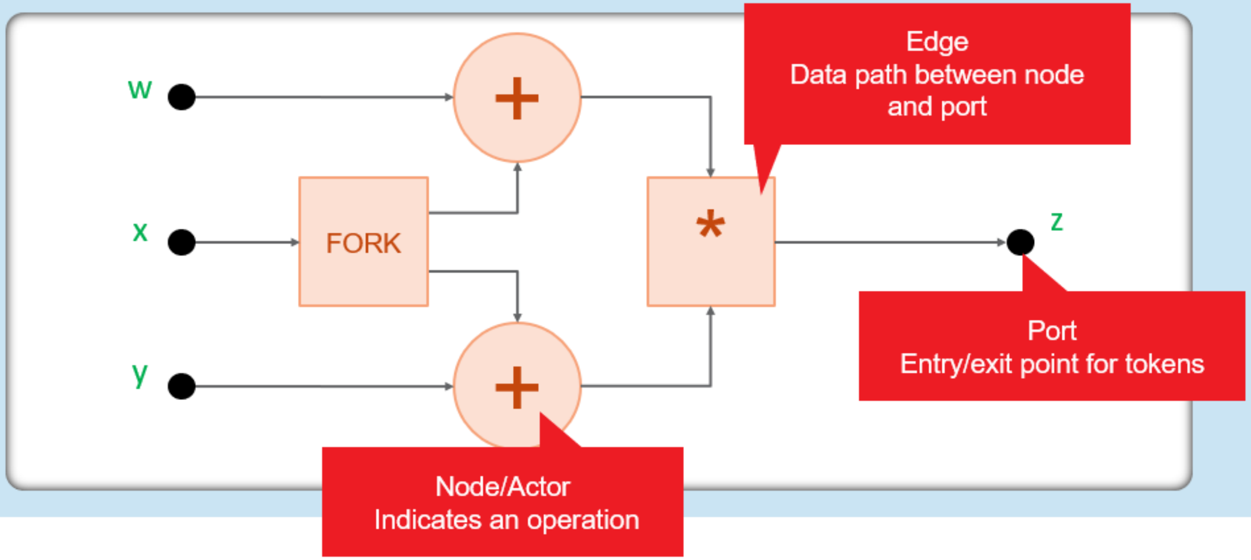
Example: X and Z → 32-bit input vectors

No. of GMACs @ 1 GHz (8 MACs * 1 GHz clock frequency) = 8 GMAC operations/second

Adaptive Data Flow Graph

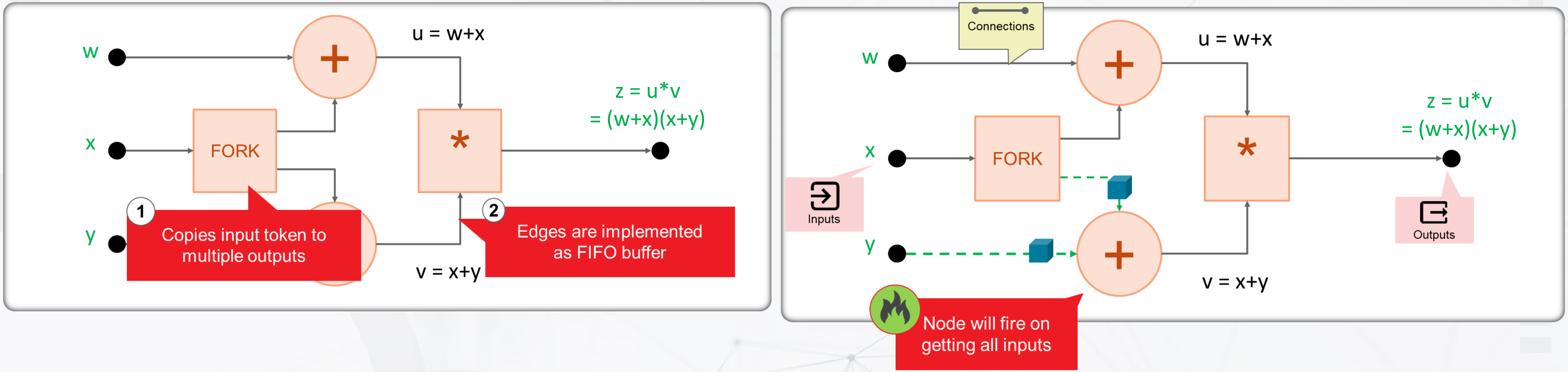
- Written in C++
- Can be compiled and executed using the AI Engine compiler
- Consists of nodes and edges:
  - Nodes [+] represent compute kernel functions
  - Edges [*] represent data connections
- It is a modified Kahn process network with the AI Engine kernels operating in parallel

A representation of how inputs are processed to generate outputs



Adaptive Data Flow Graph

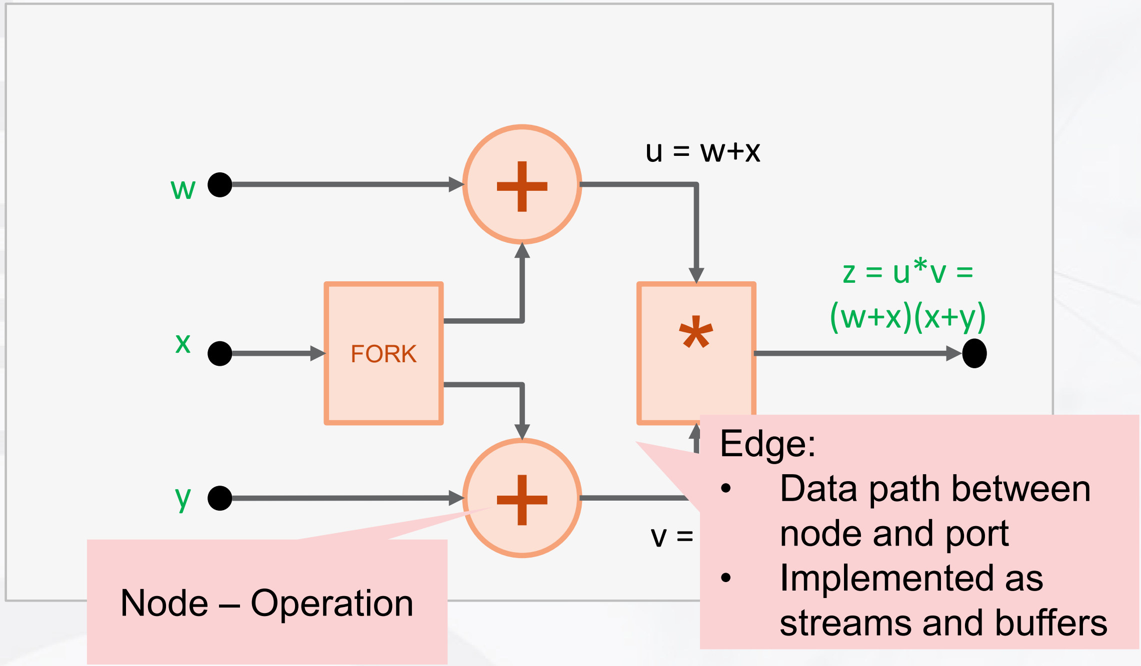
Indicates processing sequence, parallelism, and data dependence



AMD
together we advance_

Adaptive Data Flow Programming for AI Engine

Data Flow Graph Example



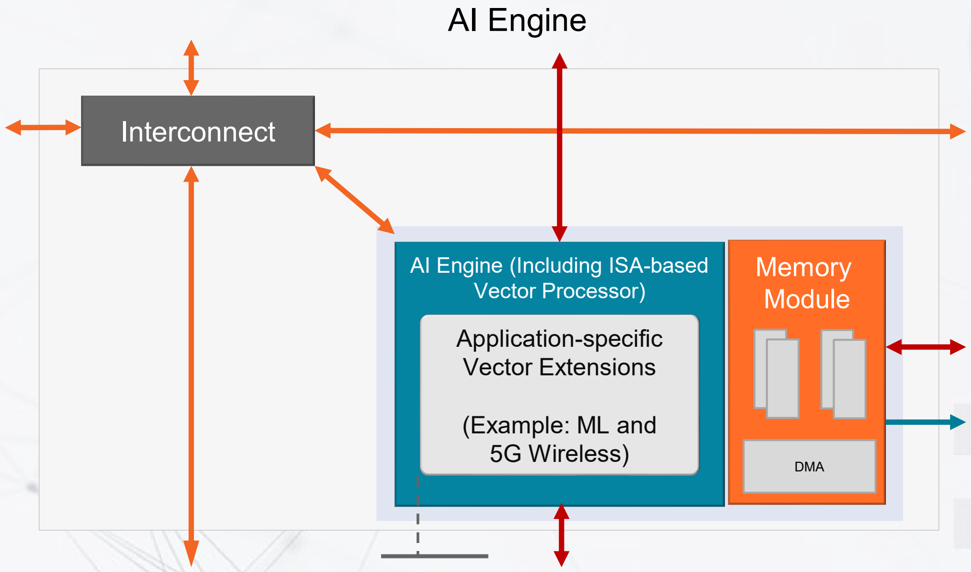
Data Flow Graph

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

Algorithm:

```
u = w+x;
v = x+y;
z = u*v = (w+x)(x+y);
```

Program – series of sequential operations



AI Engine

- Implements nodes/kernels
- Operations with multiple operators
- Can have multiple kernels
- Limitation: instruction memory 16 KB
- 10s to 100s of AI Engines
- AI Engines either computing or waiting for data

Describing a Single Kernel Environment in Graph

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

class FirstGraph : public adf::graph
{
private:
  adf::kernel k;

public:
  adf::input_plio pl_in;
  adf::output_plio pl_out;

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

    k = adf::kernel::create(vectorized_weighted_sum_with_margin);

    adf::connect< >(pl_in.out[0], k.in[0]);
    adf::connect< >(k.out[0], pl_out.in[0]);

    adf::source(k) = "aie_kernel/weighted_sum.cc";

    adf::runtime<ratio>(k) = 0.9;

  }
};
```

Adaptive Data Flow library

AI Engine application described as a graph: Class declaration

Single kernel declaration

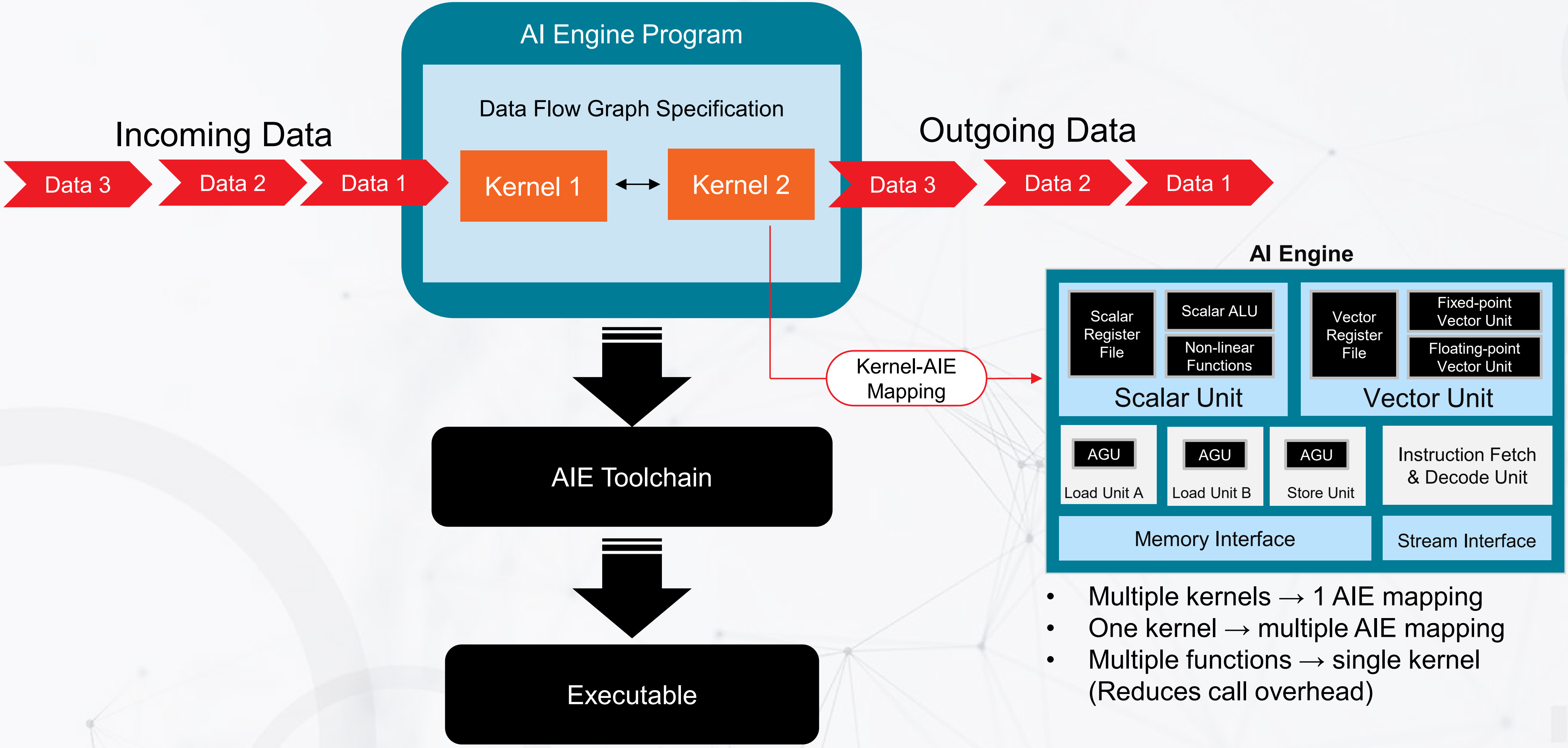
I/Os of the graph

Creation of a virtual platform:
- Input test vector file
- Output vector file
- Connection of the graph

The constructor of the graph describes all the connections and some other parameters

For single kernel programming this section is very simple

Multiple Kernel Programming



- Multiple kernels → 1 AIE mapping
- One kernel → multiple AIE mapping
- Multiple functions → single kernel (Reduces call overhead)

Data Flow Graph for Multiple Kernels

Derived from adf::graph

Loaded once on reset or loaded dynamically

Edges will be replaced by streams and buffer

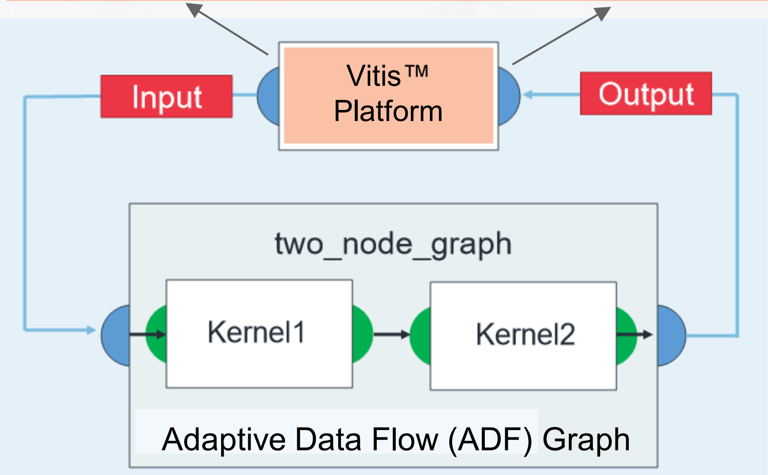
Use blocking read and non-blocking write

Model extensions for synchronized data, double buffers, etc.

Schedule, interleave, parallelize the processes in any order

- C/C++ kernels
- OpenCL™ kernels
- RTL kernels
- Target platform using Vitis compiler

I/O can connect to:



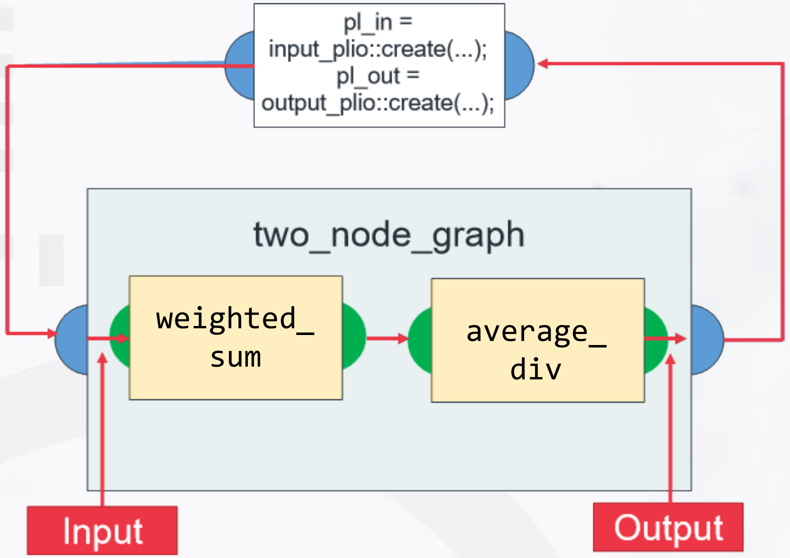
Explicitly represents:

- Compute block
- Data flow

Only risk: Deadlock due to blocking read/write

AMD together we advance_

Graph Code for Multiple Kernels



Define application graph class in *header file*

Add Adaptive Data Flow (ADF) library and include the kernel function prototypes

Define graph class
Use objects defined in the adf name space

Declare the top-level ports to the graph class

Declare the kernels

```
#include <adf.h>                 1
#include "kernels.h"

using namespace adf;

class simpleGraph : public       2
       adf::graph {
      public:
// Declare external ports         3
     input_plio in;
     output_plio out;

// Declare kernels                4
     adf::kernel k1;
     adf::kernel k2;
  }
```


AMD
together we advance_

Kernel Instantiation and Connections in graph.h

Kernels:

Ordinary C/C++ functions

- Return void
- Use special data types

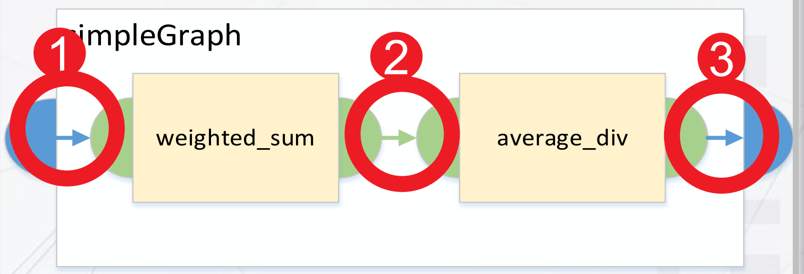
kernel::create

To instantiate the first and second C++ kernel objects

Example: Two-node application implemented on AI Engines

```
class simpleGraph : public adf :: graph {

public :
  input_plio in;
  output_plio out;
  // Declare kernels
        adf::kernel k1;
        adf::kernel k2;
 simpleGraph() {
   // Bind a function to each of the declared kernels
   k1  =  adf::kernel::create(weighted_sum);
   k2  =  adf::kernel::create(average_div);
 // create nets to connect kernels and IO ports
     adf::connect<> net0 (pl_in.out[0], k1.in[0]);   1
     adf::connect<>net1 (k1.out[0], k2.in[0]);       2
     adf::connect<>net2 (k2.out[0], pl_out.in[0]);   3
```

Create connection between ports and kernels



Connectivity information added in a data flow graph

Ports are referred to by indices

AMD together we advance_

Runtime Ratio Constraint

AI Engine Utilization Attribute

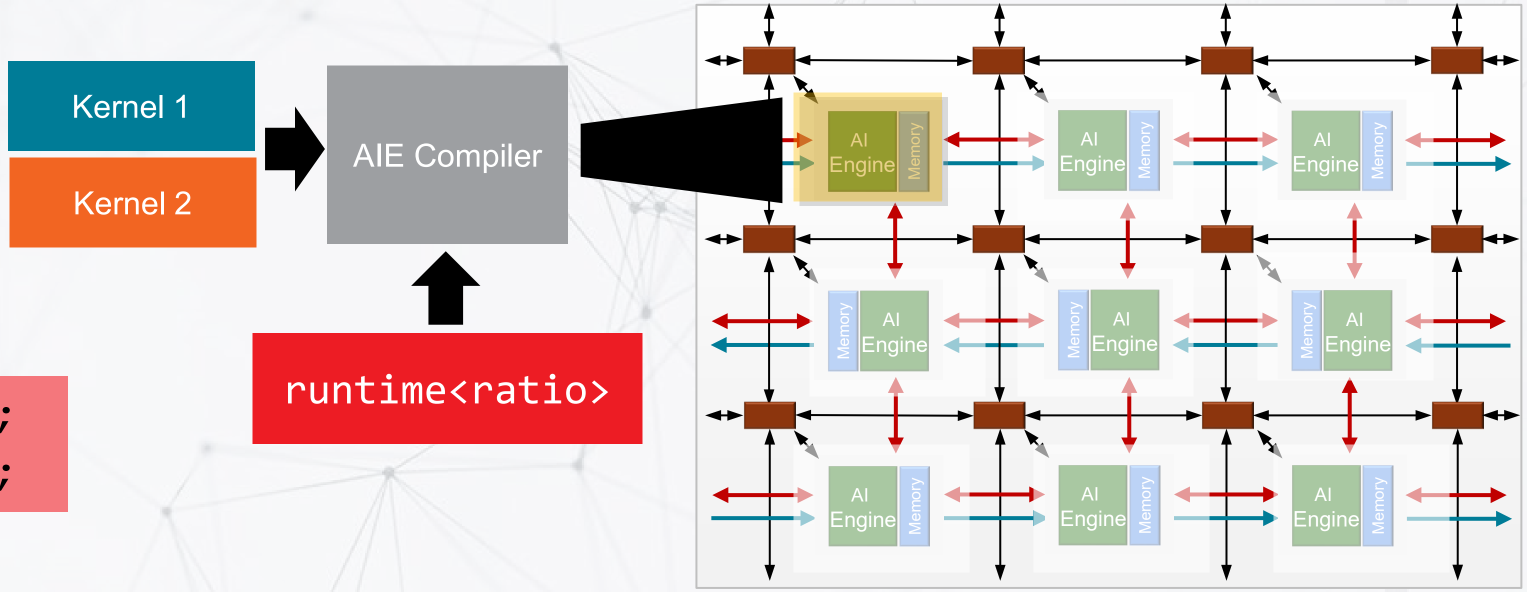
Cycle budget for an application is typically fixed

Ratio of the kernel runtime compared to cycle budget (between 0 and 1)

Cycle budget and function runtime can be affected when changing frame size

Sets specific AI Engine usage for a kernel

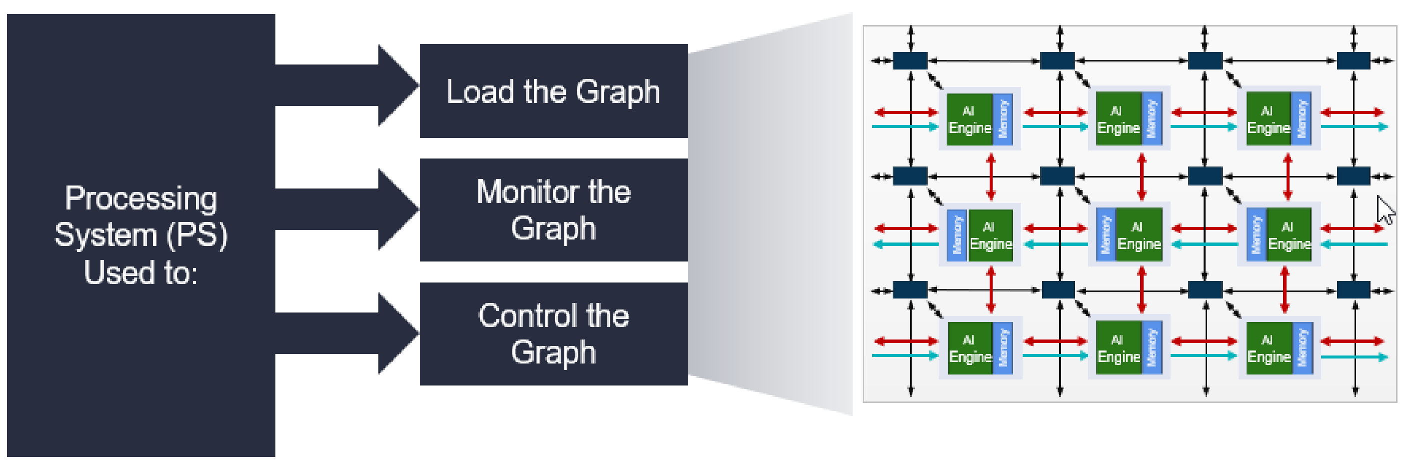
Ratio of the number of cycles taken by one invocation




```
adf::runtime<ratio>(k1)  = 0.1;
adf::runtime<ratio>(k2)  = 0.1;
```


AMD together we advance_

Graph Control API



Graph base class provides a number of API methods

- Control initialization and execution of the graph

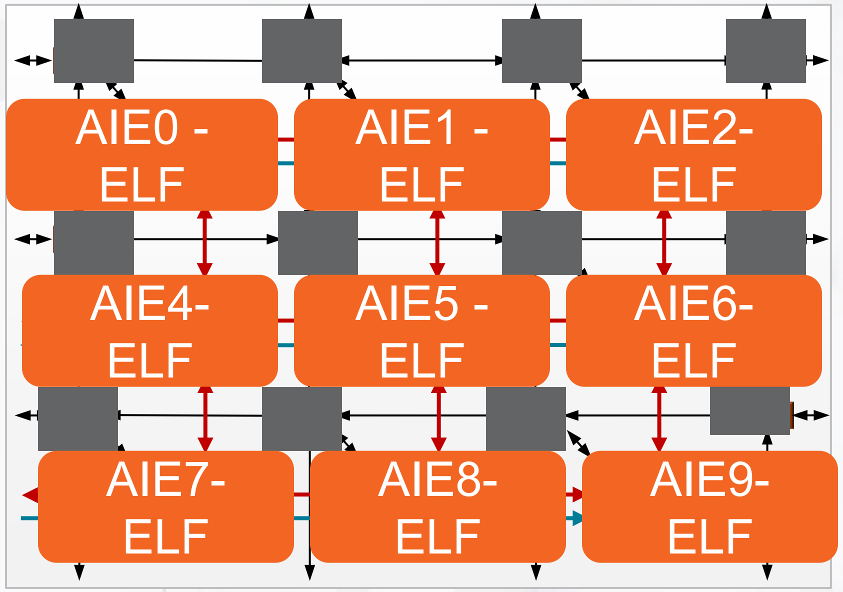
```
simpleGraph mygraph;
int main(void) {
  mygraph.init();
  mygraph.run(1);
  mygraph.end();
  return 0;
}
```

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


AMD together we advance_

Graph Control API

```
simpleGraph mygraph;

int main(void) {

  mygraph.init();

  mygraph.run(1);

  mygraph.end();

  return 0;

}
```



Initialize the AI Engines

init() method loads the graph to the AI Engine array at pre-specified AI Engine tiles

- Loads ELF binaries for each AI Engine
- Configures the stream switches for routing
- Configures the DMAs for I/O
- Leaves the processors in a disabled state

Run the AIE Programs

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

End AIE Programs

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

- Disables the graph execution

Graph Control API: Iterative Control

API wait()

Calling ***run*** back-to-back without an intervening *wait* to finish that run can have an unpredictable effect

- **run() API** modifies the loop bounds of the active processors of the graph

If there is not enough data for the input, the simulation will hang

Specify #iterations in the main program

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

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

Graph Control API: Timed Control

Timed execution model is suitable for testing multi-rate graphs

There are variants of the wait() and end() APIs specifying a cycle timeout

This is the number of AI Engine cycles before disabling the processors and returning

Blocking condition does not depend on any graph termination event

Graph can be in an **arbitrary state**

Specify #cycles in the main program

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

Describing a Multiple Kernel Environment in a Graph

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

class simpleGraph : public adf::graph
{
private:
        adf::kernel k1;
        adf::kernel k2;

public:
        input_plio pl_in;
        output_plio pl_out;

simpleGraph()
   {
   pl_in = input_plio::create("PLIO_In0", plio_32_bits, INPUT_FILE, );
   pl_out = output_plio::create("PLIO_Out0", plio_32_bits, OUTPUT_FILE, );

        k1 = kernel::create(weighted_sum_with_margin);
        k2 = kernel::create(average_div);

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

        adf::source(k1) = "kernels/weighted_sum.cc";
        adf::source(k2) = "kernels/average_div.cc";

        runtime<ratio>(k1) = 0.1;
        runtime<ratio>(k2) = 0.1;
   }
};
```

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

2 Step 2: Define your graph class

3 Step 3: Declare the top-level ports

4 Step 4: Specify the input & output files to the I/O ports

5 Step 5: Declare the kernels

6 Step 6: Connect the ports for the kernels k1 and k2

7 Step 7: Attach the source to the kernels

8 Step 8: Sets the AI Engine utilization

Test Bench Description

A graph object **mygraph** is declared using a pre-defined graph class called **simpleGraph**

In the main application, this graph object is:

- Initialized
- Run
- Terminated

init	run	end
Load the graph	Execute the graph	End the graph

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

2 simpleGraph mygraph;

int main(void) {
  mygraph.init();   3
  mygraph.run();    4
  mygraph.end();    5
  return 0;
}
```


AMD together we advance_

Recommended Project Directory Structure

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

- There should be no dependencies in the order that the header files are to be included

Resources

- Overview • AI Engine Kernel and Graph Programming Guide (UG1079) • Reader • AMD Technical Information Portal

AMD