XILINX®

Beamforming Implementation on AI Engine

XAPP1352 (v1.0) January 11, 2021

Summary

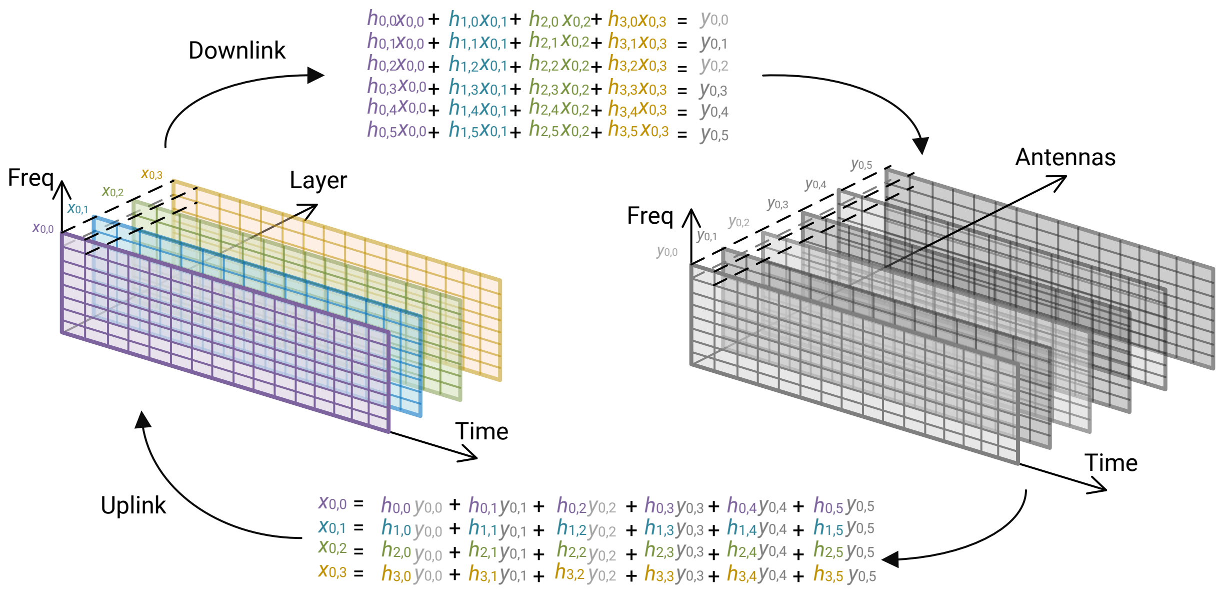
Multiple-input-multiple-output (MIMO) technology has been adopted by many wireless systems to exploit the spatial diversity of the channel. This application note shows an efficient implementation of beamforming functionality on AI Engine arrays in Xilinx® Versal™ AI Core devices. The proposed architecture consists of three kernels with slight differences and is scalable to various matrix sizes and throughput requirements. The design methodology is applicable to many use cases requiring high-throughput matrix multiplication including, but not limited to, 5G wireless communication.

Download the reference design files for this application note from the from the Xilinx® website. For detailed information about the design files, see Reference Design.

Introduction

5G wireless communication systems have enhanced multiple-input-multiple-output (MIMO) technology by employing a larger number of antennas for higher spectral efficiency than that of previous generations (3GPP Std TS 38.212). In MIMO systems, spatially uncorrelated data streams can be transmitted and received simultaneously in the same spectrum as if the communication channel were layered into many independent subchannels. The following figure shows the beamforming of an orthogonal frequency division multiplex (OFDM) system with four layers and six antennas.

Figure 1: **Beamforming in OFDM Systems**


Downlink
$h_{0,0}x_{0,0} + h_{1,0}x_{0,1} + h_{2,0}x_{0,2} + h_{3,0}x_{0,3} = y_{0,0}$
$h_{0,1}x_{0,0} + h_{1,1}x_{0,1} + h_{2,1}x_{0,2} + h_{3,1}x_{0,3} = y_{0,1}$
$h_{0,2}x_{0,0} + h_{1,2}x_{0,1} + h_{2,2}x_{0,2} + h_{3,2}x_{0,3} = y_{0,2}$
$h_{0,3}x_{0,0} + h_{1,3}x_{0,1} + h_{2,3}x_{0,2} + h_{3,3}x_{0,3} = y_{0,3}$
$h_{0,4}x_{0,0} + h_{1,4}x_{0,1} + h_{2,4}x_{0,2} + h_{3,4}x_{0,3} = y_{0,4}$
$h_{0,5}x_{0,0} + h_{1,5}x_{0,1} + h_{2,5}x_{0,2} + h_{3,5}x_{0,3} = y_{0,5}$
Freq
$x_{0,0}$ $x_{0,1}$ $x_{0,2}$ $x_{0,3}$
Layer
Time
Antennas
Freq
$y_{0,0}$ $y_{0,1}$ $y_{0,2}$ $y_{0,3}$ $y_{0,4}$ $y_{0,5}$
Time
Uplink
$x_{0,0} = h_{0,0}y_{0,0} + h_{0,1}y_{0,1} + h_{0,2}y_{0,2} + h_{0,3}y_{0,3} + h_{0,4}y_{0,4} + h_{0,5}y_{0,5}$
$x_{0,1} = h_{1,0}y_{0,0} + h_{1,1}y_{0,1} + h_{1,2}y_{0,2} + h_{1,3}y_{0,3} + h_{1,4}y_{0,4} + h_{1,5}y_{0,5}$
$x_{0,2} = h_{2,0}y_{0,0} + h_{2,1}y_{0,1} + h_{2,2}y_{0,2} + h_{2,3}y_{0,3} + h_{2,4}y_{0,4} + h_{2,5}y_{0,5}$
$x_{0,3} = h_{3,0}y_{0,0} + h_{3,1}y_{0,1} + h_{3,2}y_{0,2} + h_{3,3}y_{0,3} + h_{3,4}y_{0,4} + h_{3,5}y_{0,5}$
X24181-072920

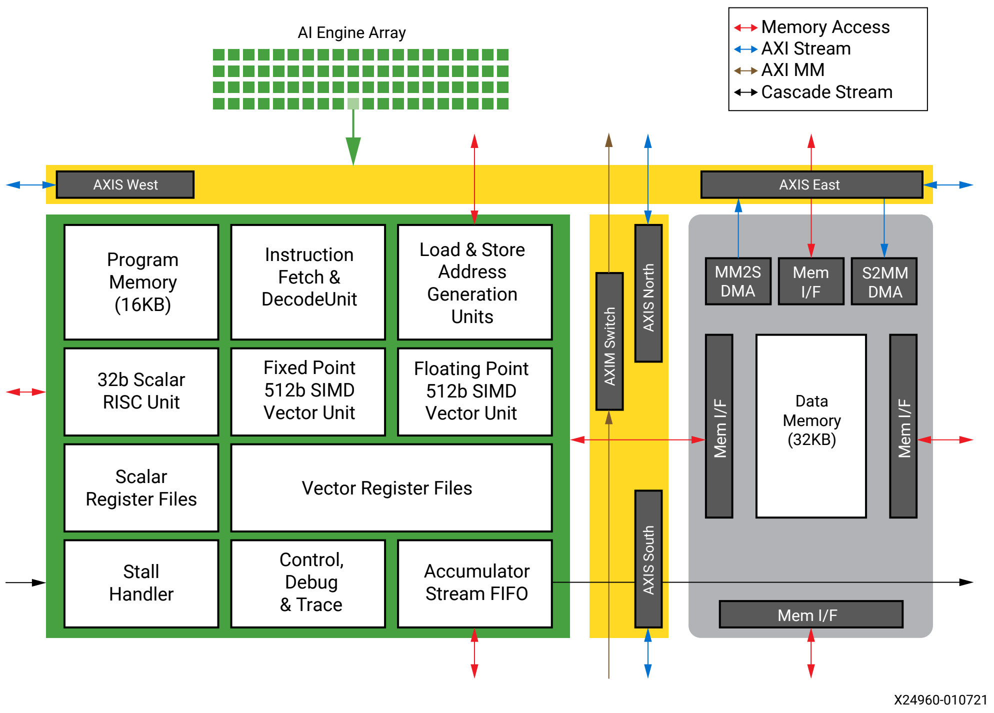
A wireless base station is made up of baseband and radio units, and its complexity is proportional to the number of layers and antennas, respectively. The beamforming module is located between the baseband and radio units and its complexity is proportional to the product of the number of layers and antennas. In 5G MIMO systems with a larger number of antennas to support more layers, the complexity of beamforming is as high as 320 times that of 4G LTE, and this has become one of the major challenges to system design.

Table 1: **Complexity Comparison Between 4G LTE and 5G NR Carriers**

Carrier Type	4G LTE	5G NR
Channel Bandwidth	20 MHz	100 MHz
Number of Antennas	8	64
Number of Layers	2	16
Radio Complexity	Normalized to 1	40x
Baseband Complexity	Normalized to 1	40x
Beamforming Complexity	Normalized to 1	320x

The Xilinx AI Engine is designed for intensive compute in various applications including, but not limited to, 5G wireless. One AI Engine tile consists of one AI Engine, 32KB data memory, and two DMA engines for automatic data transportation. Every AI Engine is equipped with a vector processor that is capable of 32 real-by-real 16-bit multiply-and-accumulate (MAC) operations in one clock cycle. The memory access unit inside the AI Engine reads 512 bits operands and writes 256 bits computation results every clock cycle to match the capability of the vector processor. In a single Versal™ AI Core device, there are hundreds of AI Engine tiles interconnected through cascading buses, AXI streams, and shared local memory according to the dataflow defined by the user at compilation time. For more detailed information about AI Engines, see *Xilinx AI Engine and Their Applications* (WP506).

Figure 2: **Block Diagram of an AI Engine Tile**



Using traditional programmable devices, 5G NR beamformers are built with thousands of DSPs and tens of thousands of look-up tables (LUTs) and flip-flops (FFs). It can easily take several months to develop such complicated systems. This application note shows that the same functionality can be built on tens of AI Engine tiles with three kernels that can be coded in the C programming language within days. The AI Engine design is guaranteed to run at a minimum of 1 GHz on Versal™ AI Core devices without the need to worry about timing closure. When the system specification changes, thanks to the scalability of the AI Engine design, only slight modifications to the C code are required.

Features

This application note proposes a method to implement wideband beamforming functionality on the AI Engine with the following features:

- A generic framework for matrix multiplication that covers a wide range of matrix sizes and throughput requirements.
- A scalable architecture that only needs a small number of kernels to be developed.
- An example submatrix multiplication kernel design that fits into one AI Engine tile and achieves 85% overall efficiency with low latency.

Beamforming Formulation

As shown in Figure 1, beamforming can be described as linear operations. In the downlink, the transmit signal on each antenna is a weighted summation of the layers, and in the uplink the equalized signal on each layer is a linear combination of the signals received on the antennas.

Write the vector of layers on subcarrier k as $X_k = [x_{k,0}, x_{k,1}, x_{k,2}, ..., x_{k,M-1}]^T$, where M is the number of streams; and the vector of frequency-domain transmit signal on the antennas as $Y_k = [y_{k,0}, y_{k,1}, y_{k,2}, ..., y_{k,N-1}]^T$, where N is the number of antennas, and $[]^T$ is vector transpose operation. Beamforming can be formulated as a matrix multiplication:

$$[Y_k, Y_{k+1}, ..., Y_{k+L-1}]_{N \times L} = H_{N \times M}[X_k, X_{k+1}, ..., X_{k+L-1}]_{M \times L} \qquad \textit{Equation 1}$$

where H is a complex N x M matrix often known as beamforming coefficient, and L is the number of subcarriers sharing the same coefficient matrix H. Similarly, in the uplink the beamforming function can be written as

$$[X_k, X_{k+1}, ..., X_{k+L-1}]_{M \times L} = H_{M \times N}[Y_k, Y_{k+1}, ..., Y_{k+L-1}]_{N \times L} \qquad \textit{Equation 2}$$

The preceding two equations suggest that both downlink and uplink beamforming can be formulated as matrix multiplications. In the downlink, the matrix dimension is (N x M) times (M x L), which requires N x M x L complex multiplication and addition (CMAC) operations. In the uplink, though the matrix dimension becomes (M x N) times (N x L), the number of CMACs is also (N x M x L).

According to the definition of OFDM, the time duration of one OFDM symbol with occupied bandwidth B is equal to the inverse of the subcarrier spacing, which is given by K/B, where K is the number of subcarriers. For downlink beamforming, (K/L) matrix multiplications of Equation 1 must be performed in (K/B) time, so the number of CMACs in one second is given by:

$$(N \times M \times L) \times (K/L)/(K/B) = N \times M \times B \qquad \textit{Equation 3}$$

In 3GPP OFDM systems with cyclic prefix and some subcarriers not requiring beamforming, the number of CMACs given by the preceding equation is higher than the minimum requirement but is desirable for system dimensioning purpose.

In a 100 MHz 5G system of 64 antennas and 32 layers, downlink beamforming requires as high as (64 x 32 x 100e6) = 204.8G CMACs per second. At a 400 MHz clock, two DSP58s can compute 400M CMACs in one second, and it takes (204.8G / 400M x 2) = 1024 DSP58s to implement the same functionality.

In Versal™ AI Core devices, one AI Engine is capable of 8G CMACs per second. For the previous example, assuming 80% runtime ratio of the AI Engines, it takes (204.8G / 8G / 80%) = 32 AI Engines to implement the matrix multiplication, that is, one AI Engine is equivalent to 32 DSP58 blocks.

For a MIMO system with M layers and N antennas, the matrix multiplication of Equation 1 can be divided into (N/u) sub-matrix multiplication chains, each of which consists of (M/v) sub-matrix multiplication units that handle (u-by-v) times (v-by-L) matrix multiplication each. More specifically, Equation 1 can be written as follows:

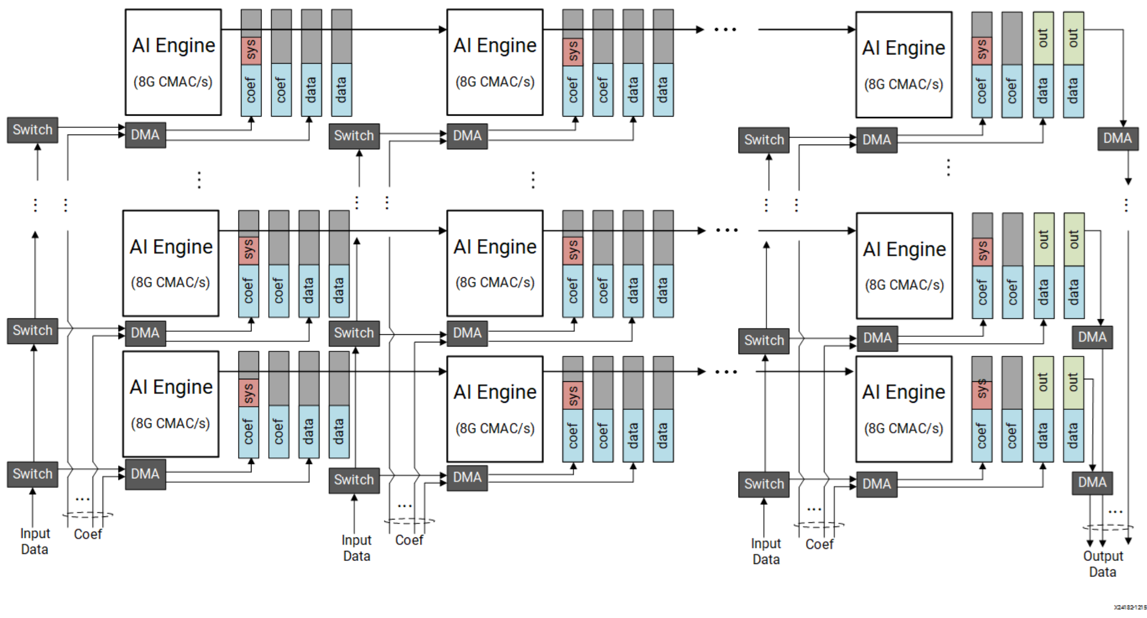
$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ \dots \\ Y_{N/(u-1)} \end{bmatrix} = \begin{bmatrix} H_{0,0} & H_{0,1} & \dots & H_{0,M/v-1} \\ H_{1,0} & H_{1,1} & \dots & H_{1,M/v-1} \\ H_{2,0} & H_{2,1} & \dots & H_{2,M/v-1} \\ \dots & \dots & \dots & \dots \\ H_{N/u-1,0} & H_{N/u-1,1} & \dots & H_{N/u-1,M/v-1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \dots \\ X_{M/v-1} \end{bmatrix} \qquad \textit{Equation 4}$$

where Y_k is a matrix of (u-by-L) complex entries, X_m is a matrix of (v-by-L) complex elements, $H_{k,m}$ is a (u-by-v) complex matrix, and they must satisfy

$$Y_k = H_{k,0}X_0 + H_{k,1}X_1 + \dots + H_{k,M/v-1}X_{M/v-1} \qquad \textit{Equation 5}$$

The preceding equation has a unified submatrix multiplication dimension of (u-by-v) times (v-by-L) that can be implemented on a single AI Engine. A chain of (M/v) AI Engines can be cascaded to accumulate the partial CMAC results for the final output.

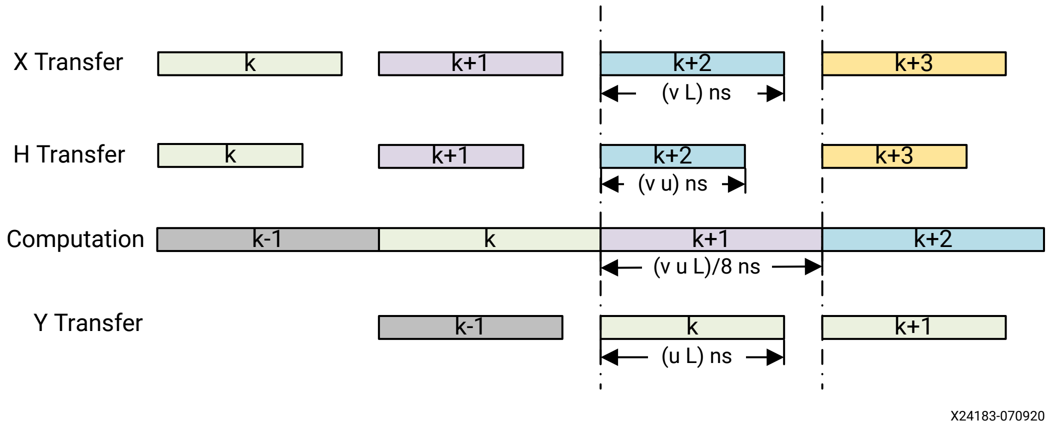
Figure 3: **Matrix Multiplication on AI Engine Array**



The previous figure shows a possible implementation of Equation 4 on the AI Engine array. Each AI Engine handles (u-by-v) times (v-by-L) matrix multiplication, and the cascading bus connects the accumulation register of one AI Engine to another to form a full-precision accumulation chain. Every row of AI Engines implements Equation 5 for a given k, and the output matrix Y_k is written into the memory of the last AI Engine tile in a row. The input to each AI Engine is provided by programmable logic and consists of a (u-by-v) matrix $H_{k,m}$ and a (v-by-L) data matrix

X_m, both of which are stored in the local memory by DMAs while the AI Engine is computing the product of the previous H and X in the double buffer. All AI Engines in one column of the array share the same data matrix X_m, so one input data stream can be multicast to all of them through AXI switches built in the AI Engine array. Note that the DMAs and AXI switches are configured by Xilinx tools automatically according to the dataflow defined by the user at compilation time.

Figure 4: **Timing Diagram of Pipelining in One AI Engine**



In one AI Engine, data transfer and computation are pipelined for high throughput. As shown in the previous figure, the time needed for data transfer and computation depends on the design parameters u, v, and L. To achieve 100% MAC efficiency in the AI Engine, the time needed for data transfer should not exceed that of computation, which means the following equation must hold.

$$(vuL/8) \geq \max((vL), (vu), (uL)) \qquad \textit{Equation 6}$$

The solution is:

$$(u \geq 8), (v \geq 8), (L \geq 8) \qquad \textit{Equation 7}$$

In 3GPP LTE and 5G NR systems, the minimum value of L is 12, and setting (u = v = 8) is a good strategy to make sure the overall throughput is not limited by data transfer bandwidth.

Moreover, the input and output AXI buses will carry time-multiplexed data of v and u channels. Limited by the throughput of each AXI bus at 1 Gs/s, the maximum signal bandwidth is upper bounded by the sample rate of each data channel given by

$$B \leq 1 \text{ GSPS}/\max(u, v) \leq 125 \text{ MHz} \qquad \textit{Equation 8}$$

All 5G NR carriers transmitted in FR1 frequency bands (below 7.125 GHz) are within 100 MHz, which fit in the above bandwidth range nicely. When there are multiple carriers and the total bandwidth is over 100 MHz, multiple instances of beamforming modules are used to meet the throughput requirement.

Larger u and v values reduce the number of AI Engines, however, the total amount of compute given by Equation 3 must be satisfied, that is:

$$\mathrm{ceil}(M/u) \times \mathrm{ceil}(N/v) \times (8\mathrm{G\ CMACs}) \geq NMB \qquad \textit{Equation 9}$$

After simplification:

$$uv \leq (8\ \mathrm{GHz}/B) \qquad \textit{Equation 10}$$

Equation 10 gives an upper bound of the product of u and v, which can be translated into a lower bound of the number of AI Engines or the overall MAC efficiency of each AI Engine. For 100 MHz 5G NR carriers where B=100 MHz, the upper bound is u v ≤ (8000/100) = 80. The selection of (u = v = 8) represents a MAC efficiency of the vector processor in a single AI Engine of (8 × 8/80) = 80% and is optimal for many wireless systems where the numbers of antennas and layers are both integer multiples of eight.

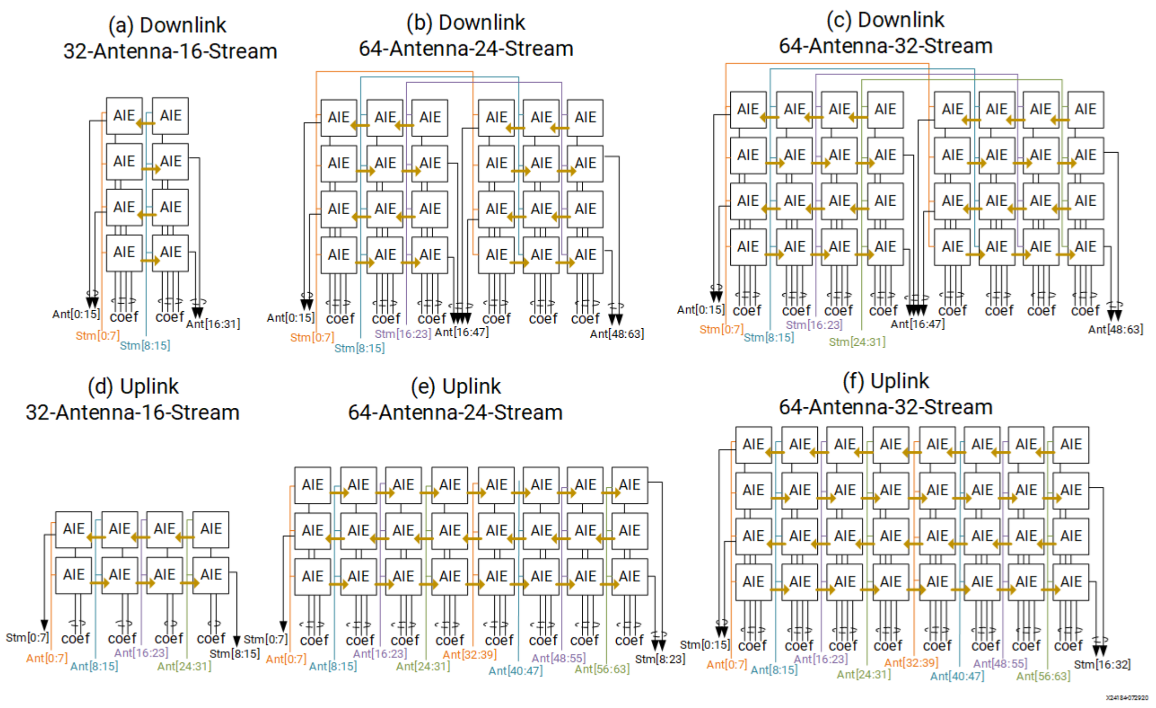
Case Studies

The generic matrix multiplication architecture proposed in this application note is scalable to various beamforming configurations. For illustration purposes, a 5G NR 100 MHz system is studied in this section and the following figure shows the beamformer implementation of several use cases. All the designs are based on a highly scalable architecture that can be built with a small number of kernels differing in the input and output interfaces only:

1. Each AI Engine performs (8 × 8) times (8 × 12) submatrix multiplication, that is, u = v = 8, L = 12.
2. The throughput of every AXI4-Stream for the input and output data is 8 × 100=800 MSPS, which is 80% of the capacity of 1 GSPS.
3. The throughput of every AXI4-Stream for the coefficients is 8 × 8/(12 × 1/100 MHz) = 533 MSPS.

Because the memory access of every AI Engine is within its own tile, no conflict with other AI Engines is expected.

Figure 5: **5G NR 100 MHz Beamforming Implementation on AI Engine**



In narrowband systems, Equation 10 indicates a higher upper bound on the product (u v) and therefore more freedom in the selection of system parameters. One strategy is to select (u = v ≤ sqrt(8G/B)) such that downlink and uplink can share the same set of kernels. For an LTE 20 MHz system of 64 antennas and 16 streams, B=20 MHz, it is possible to select (u = v = 16) and construct the beamformer as those shown in Figure 6 (a) and (b).

Another strategy is to combine the AXI streams of data and coefficient into one, which is equivalent to changing the parallel transfer of X and H to serial. To keep the vector processor fully occupied, they must satisfy

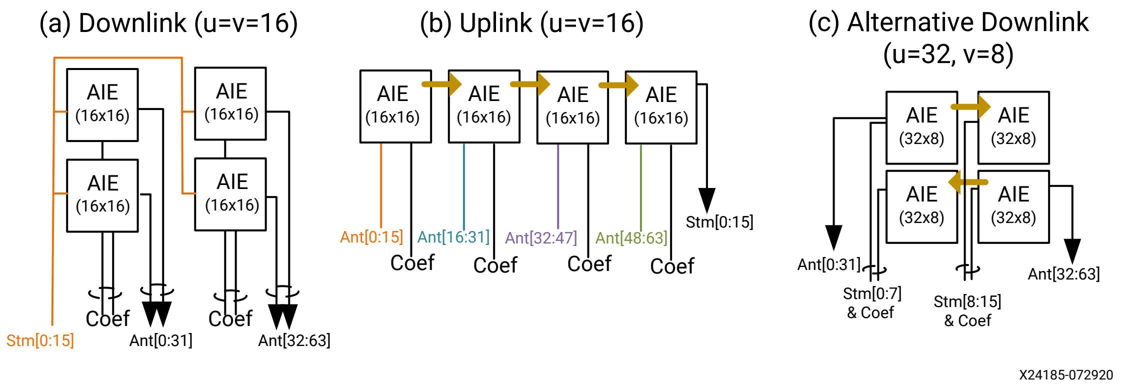
$$uvL/8 \geq uv + vL \qquad \textit{Equation 11}$$

In 3GPP systems where L=12, Equation 11 can be simplified to

$$u \geq 24 \qquad \textit{Equation 12}$$

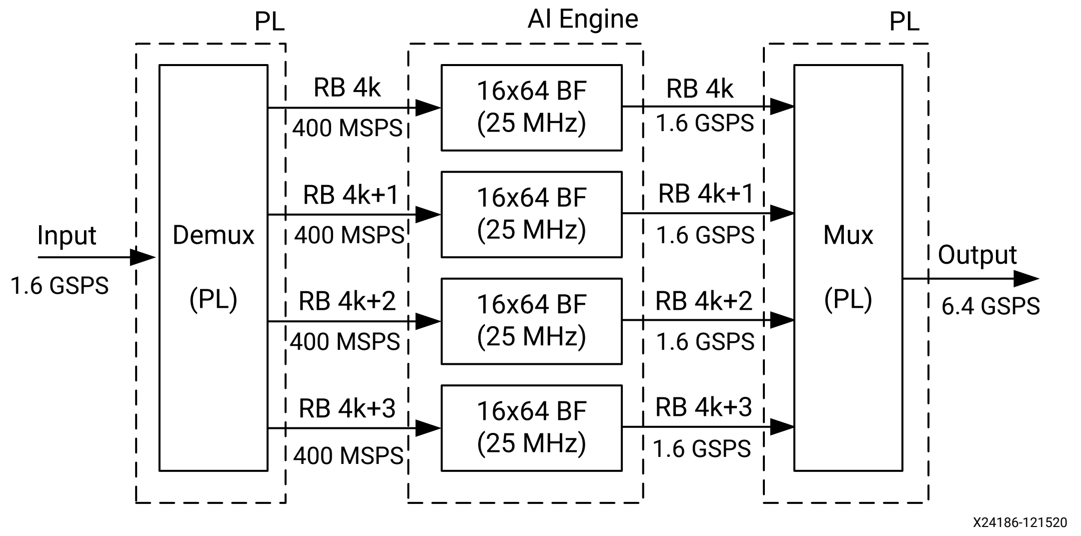
Figure 6 (c) shows an alternative downlink beamforming architecture constructed with AI Engines handling (u = 32) antennas and (v = 8) layers each. However, because the number of layers is less than 32 in this case, the uplink will have a different set of kernels as shown in Figure 6 (b).

Figure 6: **LTE 20 MHz 64 Antenna 16 Stream Beamforming Implementation on AI Engine**



When there is a special need for u and v to be certain values, it is possible to construct a wideband beamformer with multiple instances of narrow-band ones at the cost of additional programmable logic (PL) for data multiplexing and demultiplexing. The following figure shows an example of 5G NR 100 MHz beamformer made up of four instances of 25 MHz beamforming units shown in the previous figure. Though the total number of AI Engines is the same as that of a single instance of wideband beamformer, the demux block at 1.6 GSPS and mux block at 6.4 GSPS will require considerable logic resources.

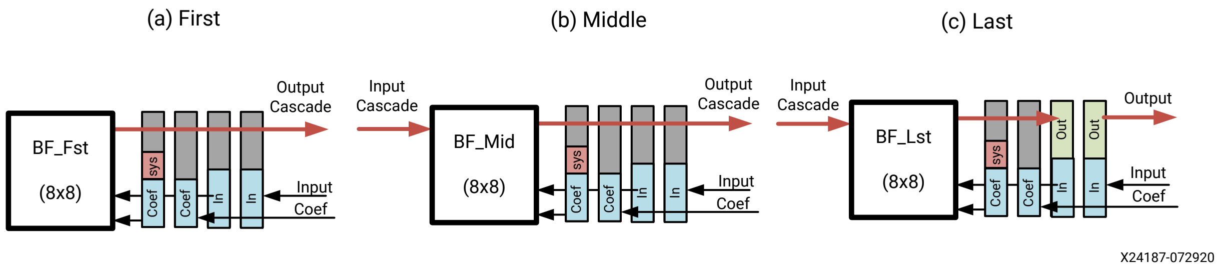
Figure 7: **5G NR 100 MHz Beamformer Using Four 25 MHz Beamforming Units**



Kernel Design Details

One feature of the proposed beamforming architecture is that only a small number of kernels is required by various system configurations. For instance, in 5G NR 100 MHz systems all the beamformers shown in Figure 5 can be built with three kernels, as shown in the following figure. Depending on the location of the kernels in the cascading chain, they are named first, middle, and last. All the kernels implement (8 x 8) times (8 x 12) matrix multiplication and only differ in the input and output interfaces. The first kernel in the cascading chain does not have cascading input, while the last one writes the output to local memory instead of the cascading bus.

Figure 8: **Three Kernels Required by 5G NR Beamforming**



Every beamforming kernel performs eight MAC4 operations on one column of eight inputs {x0, x1, x2, ..., x7} to compute 8 outputs {y0, y1, y2, ..., y7}. Each MAC4 operation takes eight coefficients and two inputs, and stores the result in a register of 384 bits. Two accumulation registers are allocated for {y0, ..., y3} and {y4, ..., y7}, respectively. At the end of computation, the partial summations are sent to the next AI Engine for further accumulation, or output to local memory after shift, round, and saturation. The following figure illustrates this process, which repeats L times until all the subcarriers sharing the same coefficient matrix have been processed.

Figure 9: **Two `mul4` Operations on Input Data x0 and x1**

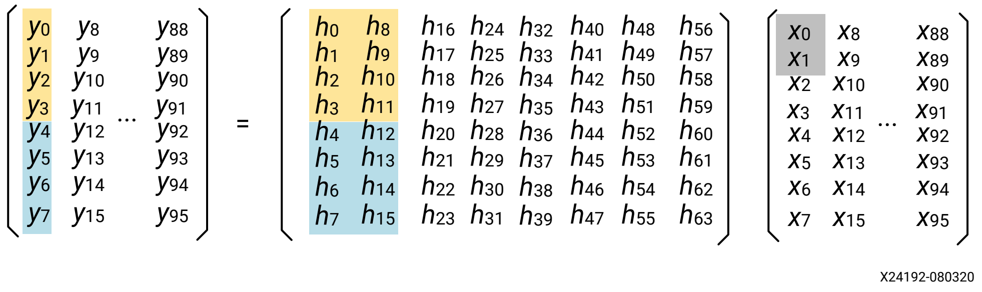


Figure 10: **Two `mac4` Operations on Input Data x_2 and x_3**

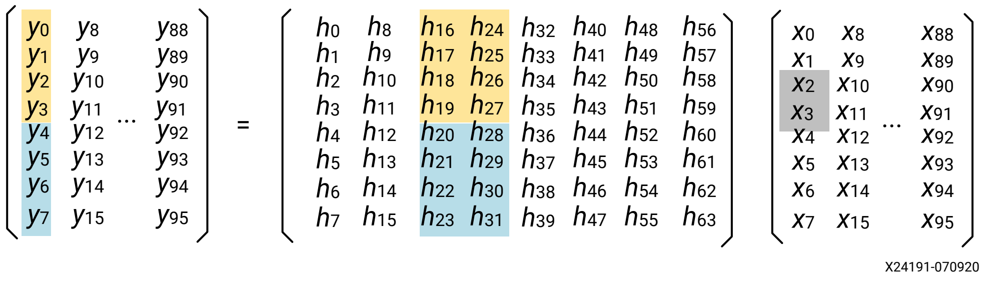


Figure 11: **Two `mac4` Operations on Input Data x_4 and x_5**

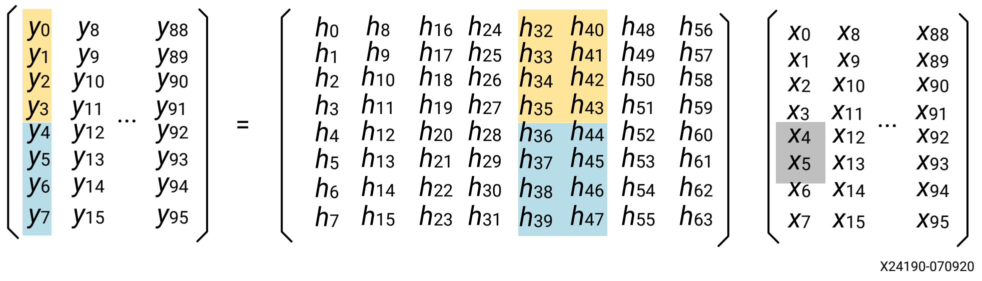
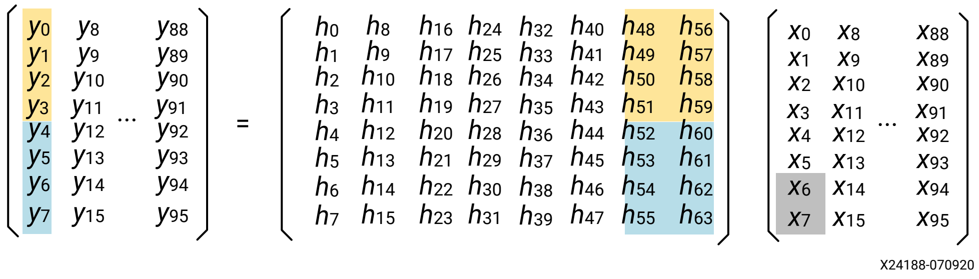
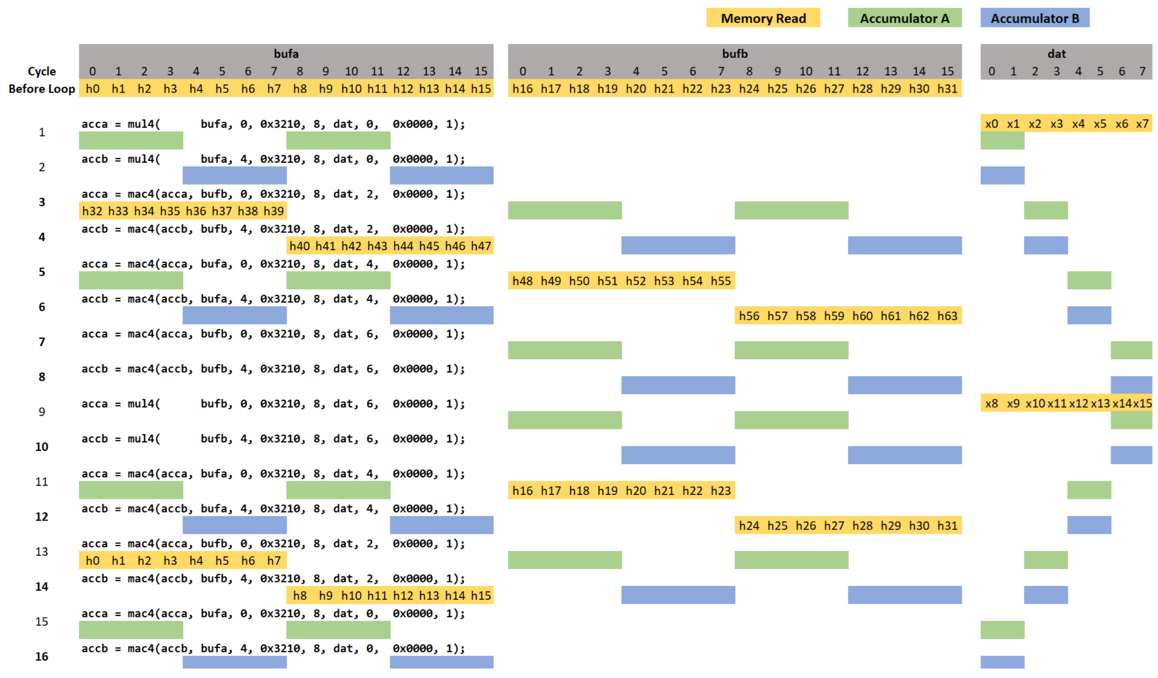


Figure 12: **Two `mac4` Operations on Input Data x_6 and x_7**



The following figure is a timing diagram of the inner loop of the `bf8x8_fst` kernel. Before the loop, registers `bufa` and `bufb` are initialized with the first half of coefficients {h0, h1, ..., h31}. One column of input data {x0, x1, ... x7} is loaded into the register `dat`. During the first four clock cycles of the loop, in parallel to the MAC operations on the first half of coefficients, the second half is read into the registers. At clock cycle 7, {y0, y1, y2, y3} are computed and sent to the next AI Engine via the cascading bus, followed by {y4, y5, y6, y7}. From clock cycle 9 to 16, the computation of the next 8 data is performed in reverse order; the MAC operations start from the second half of the coefficients already available in the registers, and then the first half is loaded at cycle 13 and 14. The inner loop takes 16 clock cycles, during which 16 `mul4`/`mac4` operations, 10 memory loads, and four cascading bus pushes are executed in parallel. The vector processor is fully occupied without any idle cycle. For L subcarriers, the inner loop runs for L/2 iterations.

Figure 13: **Timing Diagram of Inner Loop of `bf8x8_fst` Kernel**


Memory Read
Accumulator A
Accumulator B
bufa
bufb
dat
Cycle
Before Loop
h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27 h28 h29 h30 h31
x0 x1 x2 x3 x4 x5 x6 x7
h32 h33 h34 h35 h36 h37 h38 h39
h40 h41 h42 h43 h44 h45 h46 h47
h48 h49 h50 h51 h52 h53 h54 h55
h56 h57 h58 h59 h60 h61 h62 h63
x8 x9 x10 x11 x12 x13 x14 x15
h16 h17 h18 h19 h20 h21 h22 h23
h24 h25 h26 h27 h28 h29 h30 h31
h0 h1 h2 h3 h4 h5 h6 h7
h8 h9 h10 h11 h12 h13 h14 h15

```
1   acca = mul4(      bufa, 0, 0x3210, 8, dat, 0,  0x0000, 1);
2   accb = mul4(      bufa, 4, 0x3210, 8, dat, 0,  0x0000, 1);
3   acca = mac4(acca, bufb, 0, 0x3210, 8, dat, 2,  0x0000, 1);
4   accb = mac4(accb, bufb, 4, 0x3210, 8, dat, 2,  0x0000, 1);
5   acca = mac4(acca, bufa, 0, 0x3210, 8, dat, 4,  0x0000, 1);
6   accb = mac4(accb, bufa, 4, 0x3210, 8, dat, 4,  0x0000, 1);
7   acca = mac4(acca, bufb, 0, 0x3210, 8, dat, 6,  0x0000, 1);
8   accb = mac4(accb, bufb, 4, 0x3210, 8, dat, 6,  0x0000, 1);
9   acca = mul4(      bufb, 0, 0x3210, 8, dat, 6,  0x0000, 1);
10  accb = mul4(      bufb, 4, 0x3210, 8, dat, 6,  0x0000, 1);
11  acca = mac4(acca, bufa, 0, 0x3210, 8, dat, 4,  0x0000, 1);
12  accb = mac4(accb, bufa, 4, 0x3210, 8, dat, 4,  0x0000, 1);
13  acca = mac4(acca, bufb, 0, 0x3210, 8, dat, 2,  0x0000, 1);
14  accb = mac4(accb, bufb, 4, 0x3210, 8, dat, 2,  0x0000, 1);
15  acca = mac4(acca, bufa, 0, 0x3210, 8, dat, 0,  0x0000, 1);
16  accb = mac4(accb, bufa, 4, 0x3210, 8, dat, 0,  0x0000, 1);
```

The kernel `bf8x8_mid` reads the partial summation from the previous AI Engine before starting the first MAC operation. In the following C code, the intrinsic `get_scd()` loads the data on the cascading bus into an accumulation register, and the intrinsic `mac4()` resumes the accumulation without wasting any clock cycles.

```
acca = mac4(getc_scd(), bufa, 0, 0x3210, 8, dat, 0, 0x0000, 1);
accb = mac4(getc_scd(), bufa, 4, 0x3210, 8, dat, 0, 0x0000, 1);
```

The kernel `bf8x8_lst` writes the final computation result into local memory. The vector {y0, y1, ..., y7} is 256 bits can be written into memory in one clock cycle if the data come from a 768-bit 8-lane accumulation register. Because every `mac4` operation only updates four lanes, the intrinsics `ext_lo`, `ext_hi`, `upd_lo`, and `upd_hi` are needed. The first four instructions of the loop are shown in the following for comparison with those of other kernels:

```
acc = upd_lo(acc, mac4(getc_scd(),  bufa, 0, 0x3210, 8, dat, 0,0x0000, 1));
acc = upd_hi(acc, mac4(getc_scd(),  bufa, 4, 0x3210, 8, dat, 0,0x0000, 1));
acc = upd_lo(acc, mac4(ext_lo(acc), bufb, 0, 0x3210, 8, dat, 2,0x0000, 1));
acc = upd_hi(acc, mac4(ext_hi(acc), bufb, 4, 0x3210, 8, dat, 2,0x0000, 1));
```

Graph Design Details

The dataflow of an AI Engine design is defined by a C++ class referred to as the graph design. In the beamforming reference design, one cascading chain is defined in the subgraph `bfCascadingChain`. The template parameters `xoff` and `yoff` define the coordinate of the leftmost AI Engine and `len` specifies the length of the cascading chain in terms of the number of AI Engines. Because every AI Engine needs two inputs for data and coefficients, respectively, the whole chain has two `len` inputs and only one output.

```
template <int xoff, int yoff, int len>
class bfCascadeChain: public graph {
private:
 kernel core[len];
public:
 port<input> din[len];
 port<input> cin[len];
 port<output> out;
 bfCascadeChain() {
 ...
 } ;
}; // end of class bfCascadeChain
```

With this subgraph, beamforming designs can be constructed by instantiating several cascading chains of certain lengths. For a 5G NR 100 MHz system with 64 antennas and 32 layers, downlink beamforming can be described as eight cascading chains of length 4, and uplink as four chains of length 8. Some example graph C++ code is shown in the following table.

Downlink

```
//---------------------------------------
// DL 64 Antenna 32 Layer
//---------------------------------------
template <int xoff, int yoff>
class DL64A32L: public graph {
private:
bfCascadeChain<xoff, yoff+0, 4> bf0;
 bfCascadeChain<xoff, yoff+1, 4> bf1;
 bfCascadeChain<xoff, yoff+2, 4> bf2;
 bfCascadeChain<xoff, yoff+3, 4> bf3;
 bfCascadeChain<xoff+4, yoff+0, 4> bf4;
 bfCascadeChain<xoff+4, yoff+1, 4> bf5;
 bfCascadeChain<xoff+4, yoff+2, 4> bf6;
 bfCascadeChain<xoff+4, yoff+3, 4> bf7;
public:
 port<input> din[4];
 port<input> cin[32];
 port<output> out[8];
 DL64A32L(){
 ...
 };
}; // end of DL64A32L
```

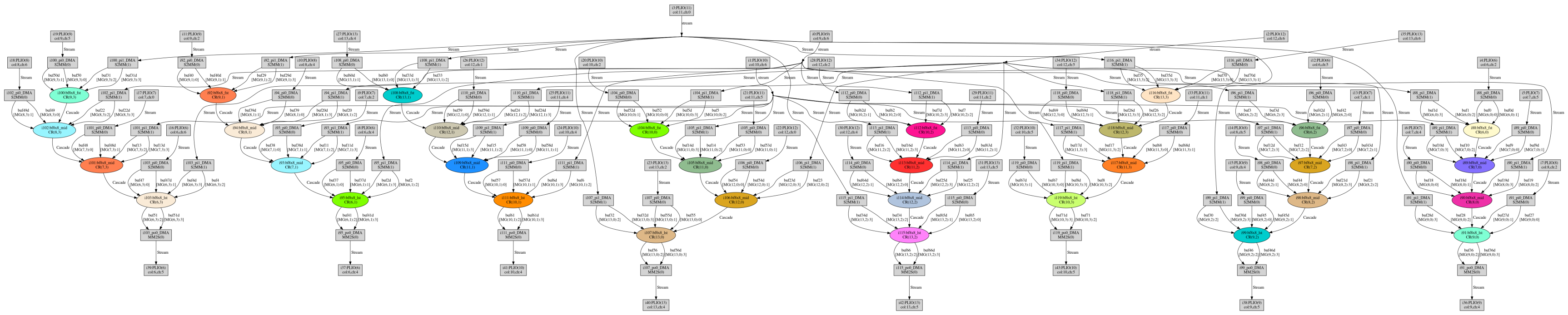
Uplink

```
//---------------------------------------
//  UL 64 Antenna 32 Layer
//---------------------------------------
template <int xoff, int yoff>
class UL64A32L: public graph {
private:
  bfCascadeChain<xoff, yoff+0, 8> bf0;
  bfCascadeChain<xoff, yoff+1, 8> bf1;
  bfCascadeChain<xoff, yoff+2, 8> bf2;
  bfCascadeChain<xoff, yoff+3, 8> bf3;




public:
  port<input> din[8];
  port<input> cin[32];
  port<output> out[4];

  UL64A32L(){
         ...
  };
};  // end of UL64A32L
```

Xilinx tools compile the graph design and automatically generate block diagrams to represent the compilation result. The following two figures show one example using the 64-antenna 32-layer 100 MHz beamforming reference design. Every colored bubble represents one AI Engine, and gray boxes are DMAs and memories used by the design. Xilinx tools automatically configure the DMAs, AXI switches, and PL-AI Engine interfaces for the graph design.

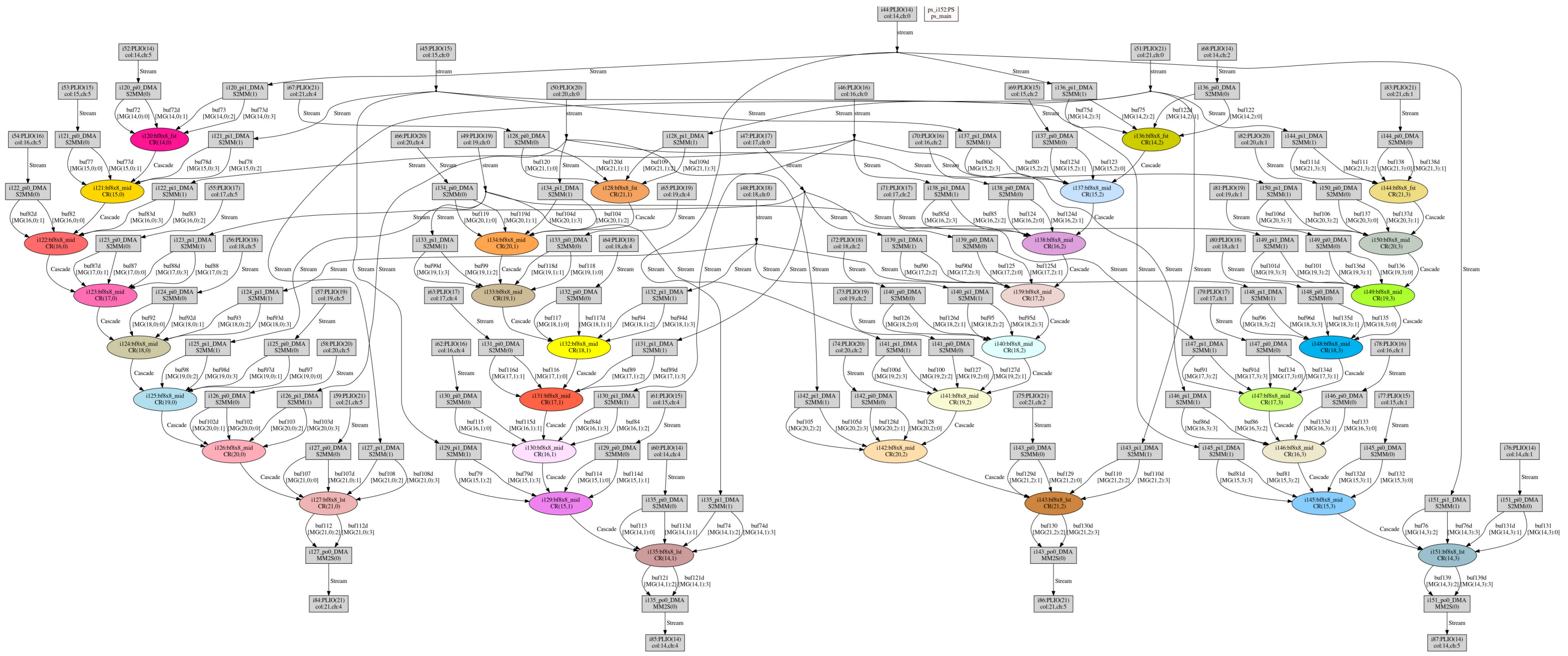
Compilation Result of 5G NR 100 MHz 64-Antenna 32-Layer Beamforming Reference Design

Figure 14: **Downlink**



Send Feedback

Figure 15: **Uplink**

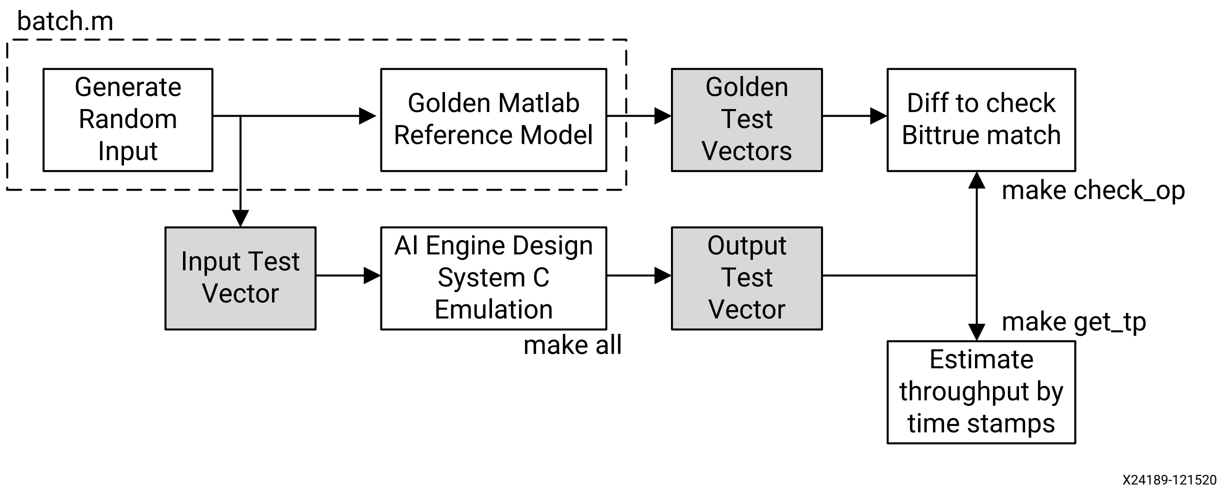


Design Validation

AI Engine designs can be simulated for functional verification and throughput validation before integration with programmable logic. The following figure shows the validation workflow for the AI Engine beamforming reference design.

Random input test vectors are generated by a MATLAB® script and golden test data are computed by the MATLAB reference model. AI Engine designs are compiled and tested in a System C simulation environment (AI Engine simulator) using the input test vectors. The AI Engine simulation results are stored in data files that record the output samples along with their time stamps. The time duration from the first output sample to the last can be measured by the time stamps, and the number of output samples can be counted. Their quotient gives an estimate of throughput.

Figure 16: **AI Engine Design Validation Workflow**



The beamforming reference design has 12 output AXI streams, each of which contains the data of eight antennas or eight layers. A makefile is included in the design to compare all 12 outputs with the golden test vectors generated by MATLAB. Table 2 (a) shows a bit-true match with the reference output for the AI Engine beamforming design. The makefile also estimates the throughput from all output files. Table 2 (b) shows that all 12 output AXI streams achieve the target throughput of 800 MSPS with more than 5% margin.

Table 2: **Example Design Validation Result**

(a) Functional Verification	(b) Throughput Estimation
`> make check_op`	`> make get_tp`
`DLBF 0 - diff=0 -`	`DLBF 0 Throughput= 856.493 Msps (>800Msps)`
`DLBF 1 - diff=0 -`	`DLBF 1 Throughput= 855.901 Msps (>800Msps)`
`DLBF 2 - diff=0 -`	`DLBF 2 Throughput= 856.512 Msps (>800Msps)`
`DLBF 3 - diff=0 -`	`DLBF 3 Throughput= 856.436 Msps (>800Msps)`
`DLBF 4 - diff=0 -`	`DLBF 4 Throughput= 855.939 Msps (>800Msps)`
`DLBF 5 - diff=0 -`	`DLBF 5 Throughput= 855.901 Msps (>800Msps)`
`DLBF 6 - diff=0 -`	`DLBF 6 Throughput= 856.493 Msps (>800Msps)`
`DLBF 7 - diff=0 -`	`DLBF 7 Throughput= 855.901 Msps (>800Msps)`
`ULBF 0 - diff=0 -`	`ULBF 0 Throughput= 855.977 Msps (>800Msps)`
`ULBF 1 - diff=0 -`	`ULBF 1 Throughput= 855.92 Msps (>800Msps)`
`ULBF 2 - diff=0 -`	`ULBF 2 Throughput= 855.901 Msps (>800Msps)`
`ULBF 3 - diff=0 -`	`ULBF 3 Throughput= 855.92 Msps (>800Msps)`

Reference Design

Download the reference design files for this application note from the from the Xilinx® website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 3: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices	Versal™ AI Core
Source code provided?	Yes
Source code format (if provided)	MATLAB® script, AI Engine C code, and Makefile script
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	No
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	No
Test bench provided for functional and timing simulation?	No
Test bench format	C code
Simulator software and version	Vitis™ 2020.2
SPICE/IBIS simulations	No
Implementation	
Implementation software tool(s) and version	Vitis™ 2020.2
Static timing analysis performed?	No
Hardware Verification	
Hardware verified?	Yes
Platform used for verification	Xilinx VCK190

Conclusion

5G NR massive MIMO systems require intensive computation for matrix multiplication. It is shown that 100 MHz 64-antenna 32-layer 5G NR beamforming can be implemented on 64 AI Engines with three kernels with slight differences, while the same functionality needs 2048 DSP58 blocks and hundreds of thousands of LUTs and FFs to be implemented on programmable logic. The proposed matrix multiplication implementation on AI Engines has a flexible and scalable architecture applicable to a wide range of use cases including, but not limited to, 5G wireless.

References

These documents provide supplemental material useful with this guide:

1. *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; NR; Multiplexing and channel coding (Release 15)* (3GPP Std TS 38.212 V15.0.0)
2. *Xilinx AI Engine and Their Applications* (WP506)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
01/11/2021 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

Copyright