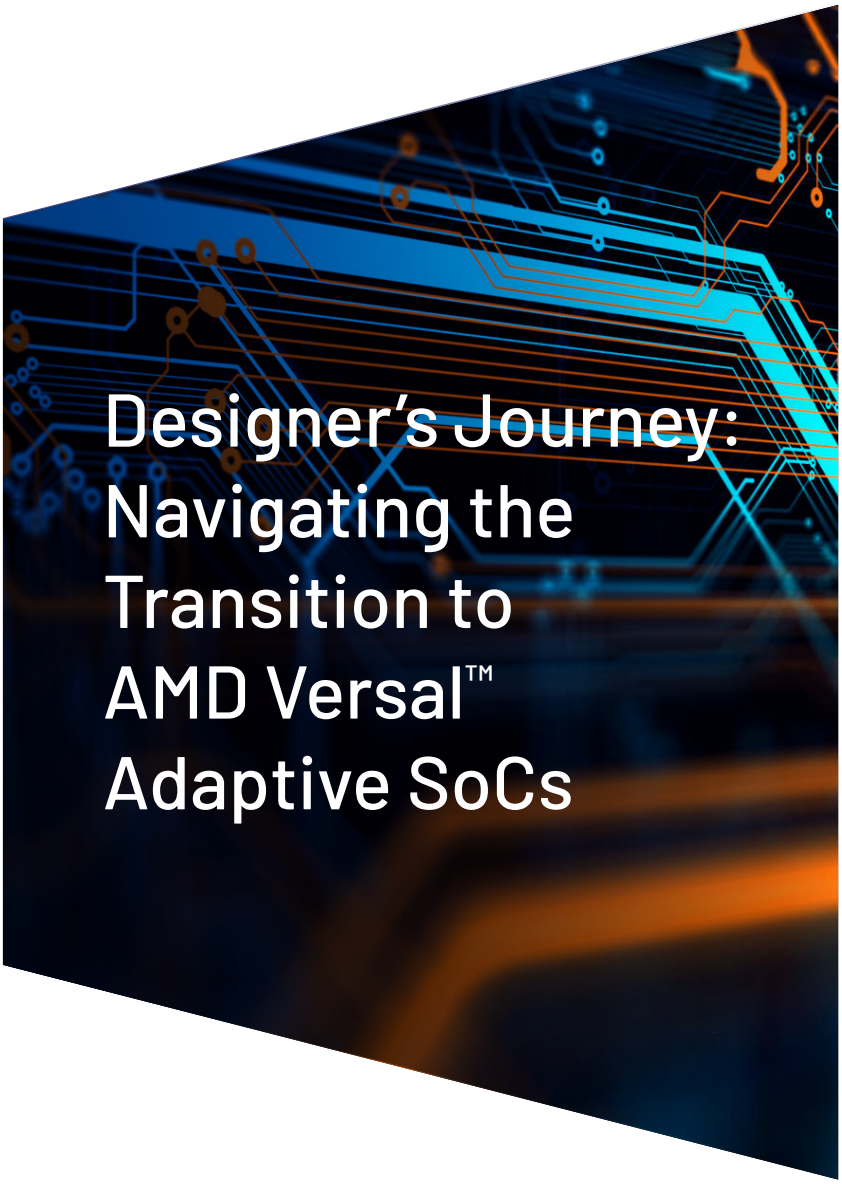
WHITE PAPER

To support ever-increasing computational challenges in radar, engineers must consider new innovative design approaches to meet real-time performance expectations.

HENRY GILSDORF
FPGA Development Engineer
Mercury Systems

MITZI WERLINE
Principal Technical Writer
Mercury Systems



Designer's Journey: Navigating the Transition to AMD Versal™ Adaptive SoCs

Ever-increasing data volumes, rising computation demands and real-time performance expectations can no longer be satisfied by traditional solutions. With the 2019 launch of the AMD Versal™ architecture, AMD has enabled an innovative approach for the next era of specialized computing. This highly dense, next-generation chip solution combines multiple types of processing elements that step beyond the current CPU/GPU/FPGA paradigm. Utilizing this technology, we can now solve the most advanced radar, cognitive EW and AI challenges — all on a single board.

AMD VERSAL™ ADAPTIVE SOC IS:

- 43x faster than today's fastest CPUs.
- 3x faster than today's fastest GPUs.
- Up to 20x faster than today's fastest FPGAs.[1]

Today's processing challenges typically fall into one of three categories, all if which can be addressed by the the AMD Versal™ platform.

Processing Challenges	Processing Solutions	AMD Versal™ platform
Scalar Processing: Complex algorithms, widely branching decision trees, broad library sets.	CPUs	Arm® Cortex™ processors
Vector Processing: Large-scale, parallel computations on high data volumes. Used for signal processing and artificial intelligence applications.	DSPs and GPUs	Adaptable Intelligence Engines (AIE)
Programmable Logic Processing: Real-time, low/no latency, and highly-customizable user applications.	FPGAs	FPGA with Network on Chip (NoC)

Mercury has benefited greatly from close collaboration with AMD during the Versal™ platform development, allowing us to bring a deployable Versal device to the market early — the new SCFE6931 Dual AMD Versal™ AI Core adaptive SoC Processing Board.

This white paper follows a Mercury design engineering team's journey toward AMD Versal™ AI Core development methodologies. By starting simply, our team was able to better understand the tools and technology behind the AMD Versal™ architecture before taking on more complex implementations.

The following engineer-to-engineer designer's journey is intended to assist other development teams as they adopt the Versal™ adaptive SoC design.

1. Source: Versal: The First Adaptive Compute Acceleration Platform (ACAP AMD Versal™ WP505 (v1.1.1)

THE JOURNEY BEGINS

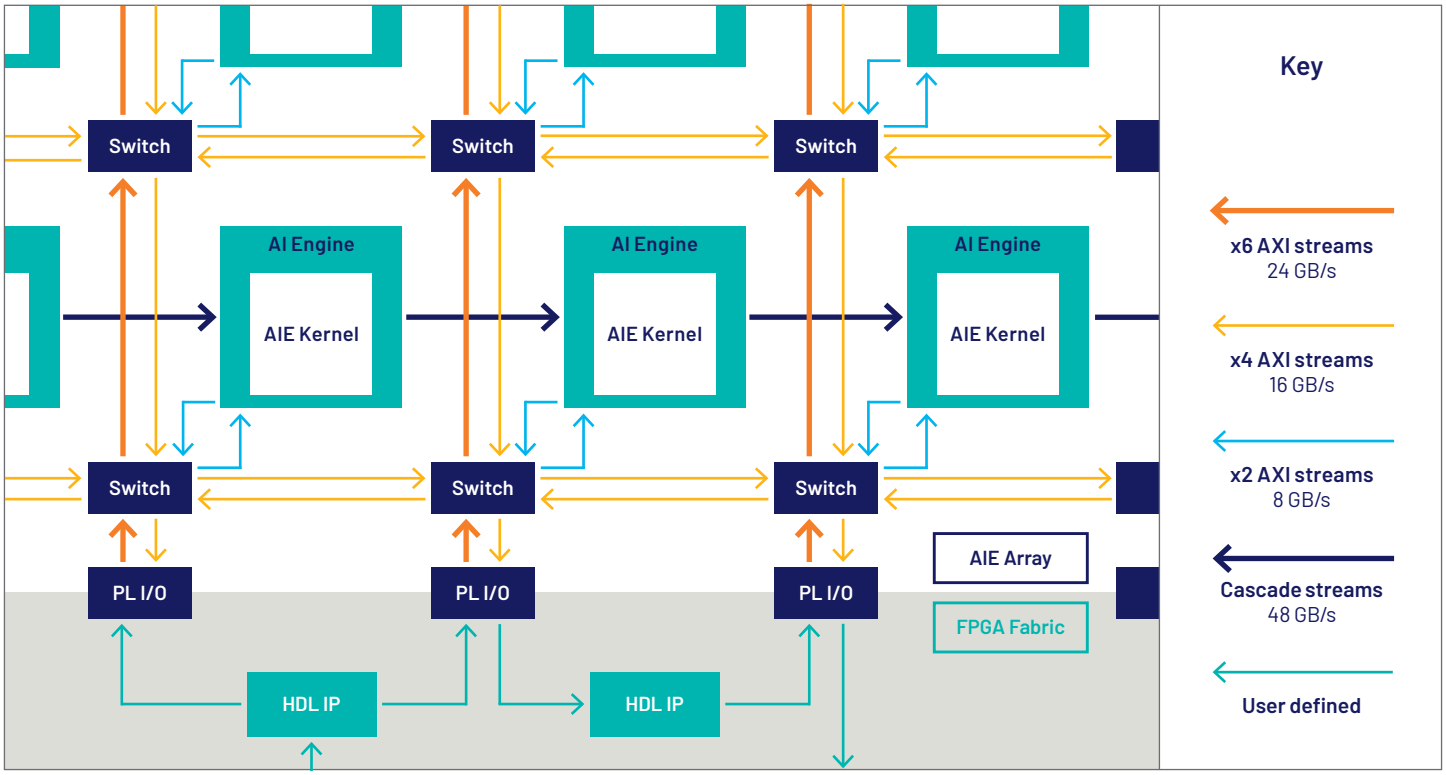
Adopting the AMD Versal™ AI Core device seemed challenging for our team at first. Having a primary background in traditional FPGA and DSP development, the idea of programming AIE processors using high-level languages was unfamiliar to us. In addition, we did not yet understand the available methods of defining the dataflow into and out of the AIE array. To dispel our worries, we decided to start small and build up our experience with AI.

At a high level, the AIE array is similar to a GPU in that it consists of hundreds of vector processors. Each AIE processor can perform up to eight complex multiplications per cycle and has its own memory scratch pad for temporary storage of work. Data inputs and outputs are AXI4-Streams and can flow from the programmable logic into multiple AIE processors before being output from the AIE array. These functions executed by the AIEs are called kernels[2], and a single AIE can share its time between different kernels.

THE POWER OF THE AMD VERSAL™ ARCHITECTURE IN A READY-TO-RUN, PROVEN AND TESTED PLATFORM

Jump-start development with the Model 8258 low-cost 6U VPX platform to build, run and debug applications on the SCFE6931 Dual AMD Versal™ adaptive SoC processing module. Providing power and cooling to match the SCFE6931 in a small desktop footprint, the chassis allows access to all required front-panel interfaces and the optional rear-panel connectors to support 100 GigE. Mercury's Navigator® FPGA design kit (FDK) and board support package (BSP) complete the preconfigured development platform.

AIE Array Diagram



Each AIE processor has two physical stream inputs and outputs, however, these interfaces can be multiplexed to accommodate a higher number of "virtual streams."

2. Kernel: A small unit of execution that performs a clearly defined function and that can be executed in parallel.

LEARNING BY EXAMPLE

We selected a problem to solve using AIE and created a small test application consisting of a single kernel. This kernel would perform a common DSP function: *beamforming*.

We began by studying the AIE architecture manual before coding the test beamformer kernel in C++. This kernel would take in multiple AXI4-Streams for element data and weights, producing an output stream of a single complex beam.

For the first design, we settled on two input streams of interleaved element samples, with another input stream for weights. These streams were continuously read into double-buffered memory within the AIE. The initial C code for the kernel function looked like this:

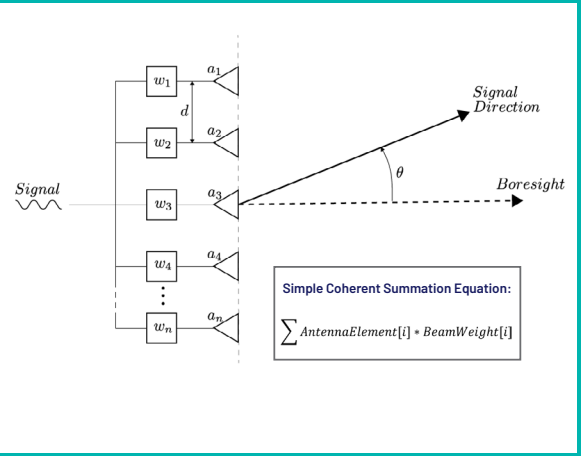
```
void singleKernelBeamformer (
        input_window_cint16 * restrict elementsEven,
        input_window_cint16 * restrict elementsOdd,
        input_window_cint16 * restrict weights,
        output_window_cint16 * restrict beam
) {

    // Define vector registers
    v8cint16 vector_elements;
    v8cint16 vector_weights;
    v8cacc48 vector_accumulator = null_v8cacc48();

    // For each vector of 8 samples
    for (unsigned i = 0; i < 8; i++) {
        window_readincr(weights, vector_weights);

        // Multiply-accumulate each element with it's weight
        for (unsigned j = 0; j < 8; j=j+2) {
            // Even Elements
            window_readincr(elementsEven, vector_elements);
            vector_accumulator = mac8(vector_accumulator, xset_w(0,vect
            // Odd Elements
            window_readincr(elementsOdd, vector_elements);
            vector_accumulator = mac8(vector_accumulator, xset_w(0,vect
        }
    }

    // Shift-Round-Saturate accumulator and write to output
    window_writeincr(beam, srs(vector_accumulator, 17));

}
```

We initially chose the input data width to be 64 receive elements, as this represents a common beamforming application. However, we soon discovered that routing 64 streams to a single AIE was not feasible.

As you will see in the next section, we overcame this obstacle by interleaving our element samples into two streams.

WHAT IS BEAMFORMING?

Beamforming — also referred to as spatial filtering — is a signal processing technique used in sensor arrays for directional signal transmission or reception. This is achieved by combining elements in an antenna array in such a way that signals at particular angles experience constructive interference while others experience destructive interference.

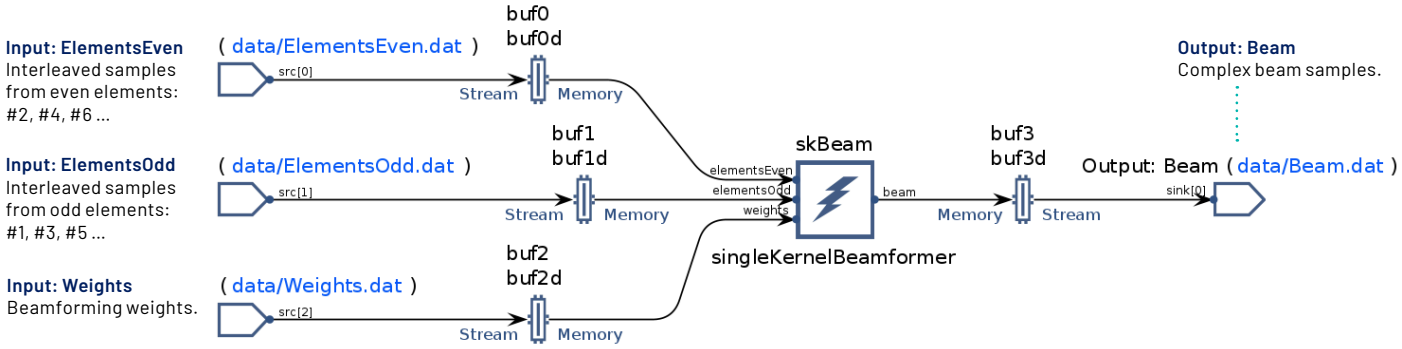


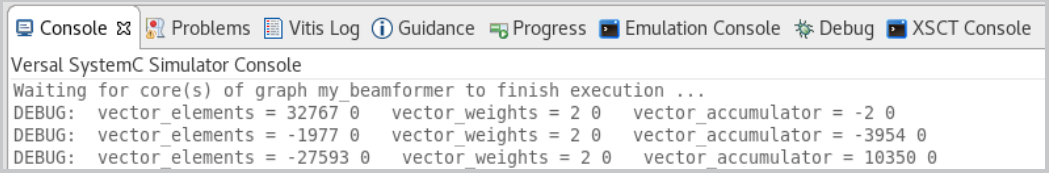
"The appearance of U.S. Department of Defense (DoD) visual information does not imply or constitute DoD endorsement."

IMPLEMENTING THE DESIGN

Each AIE application consists of a dataflow graph[3] that describes a set of kernels and their associated inputs, outputs and interconnections. For standalone simulation of an AIE graph, these input and output ports reference test vector files. We began testing our first application by generating test element data and weights with MATLAB®. These test vectors represented the input data that would normally flow from the programmable logic fabric.

Initial Design: Simple Beamformer Kernel Stream Ports Diagram



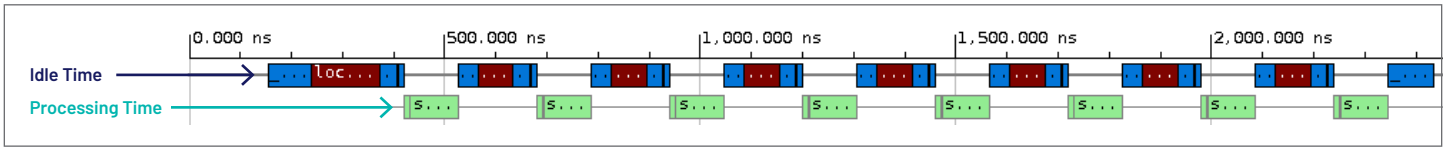


As shown here in the debug screen, we added debug printf() statements inside our C++ kernel code to print the inputs and calculation results for the first few elements.

REVIEWING OUR INITIAL AIE DESIGN

After simulating our AIE kernel, we used the AMD Vitis™ analyzer tool to display the trace data generated. This timeline display allowed us to see the activity of each AIE in the array and how effectively it was being utilized.

As shown in the AMD Vitis™ analyzer screen below, our first kernel spent a substantial amount of its time idle. This is because the AIE was able to compute the output beam faster than the I/O throughput rate.

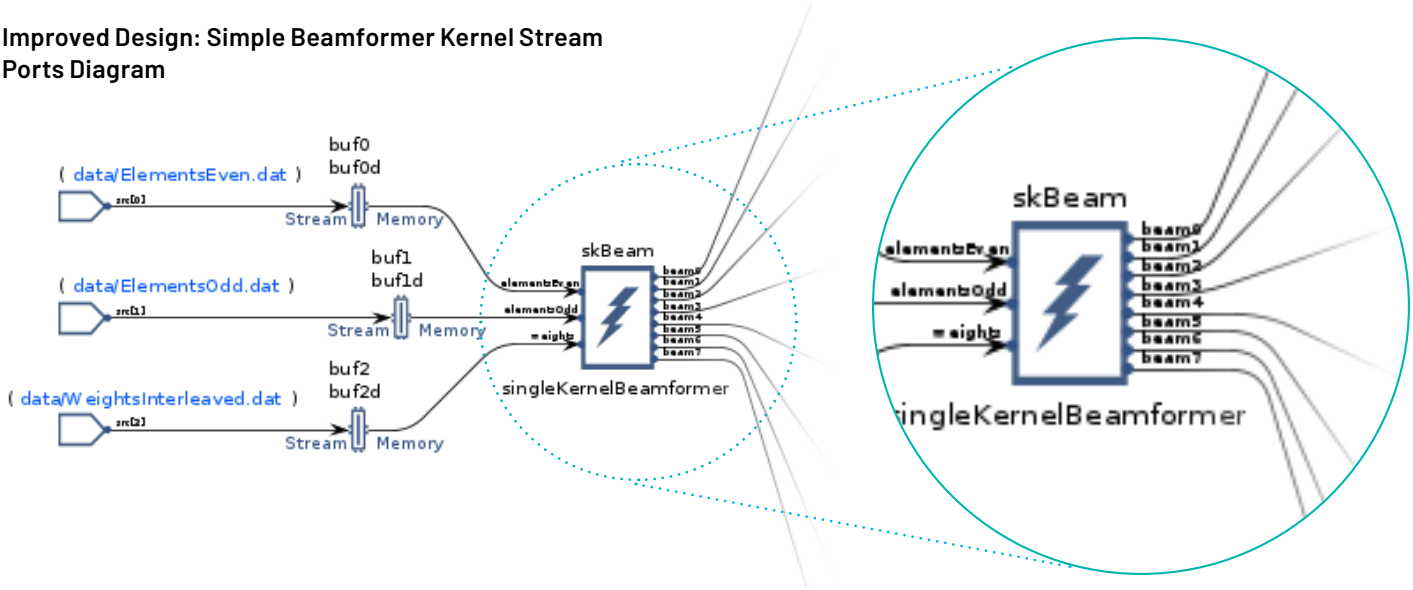


3. Dataflow graph: A dataflow graph is how an AI Engine application is described at the highest level. Dataflow graphs consist of nodes and edges where nodes represent compute kernel functions and edges represent data connections.

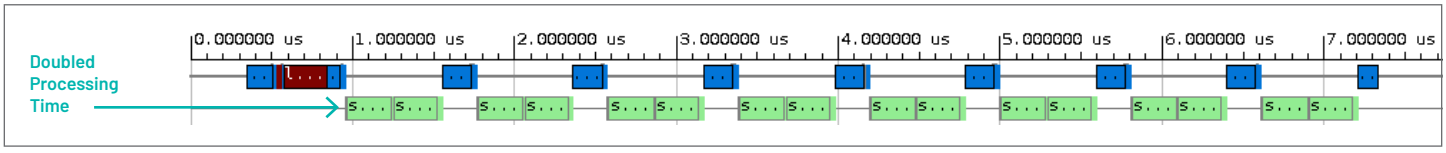
INCREASING OUR UTILIZATION

To make better use of the processor's time, we experimented with increasing the number of beam outputs to discover how throughput would be affected.

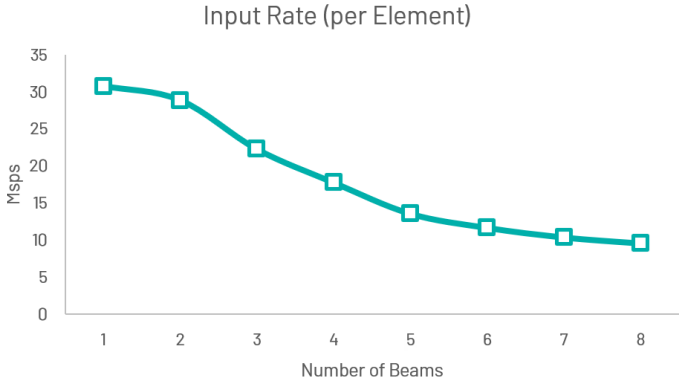
Improved Design: Simple Beamformer Kernel Stream Ports Diagram

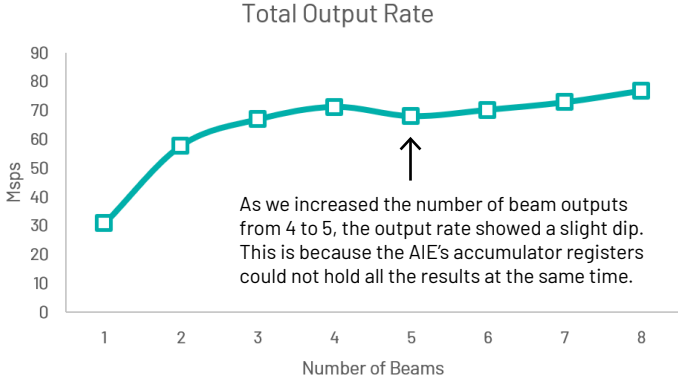


As shown below, the single AIE processor is now tasked more than twice as efficiently. By reusing the element data with more sets of weights to produce more beams, we greatly increased our efficiency. However, this also meant that the input throughput was reduced because we were now CPU bound.



Charting the throughput for designs with different numbers of beam outputs illustrates the trade-offs that should be considered when designing applications.



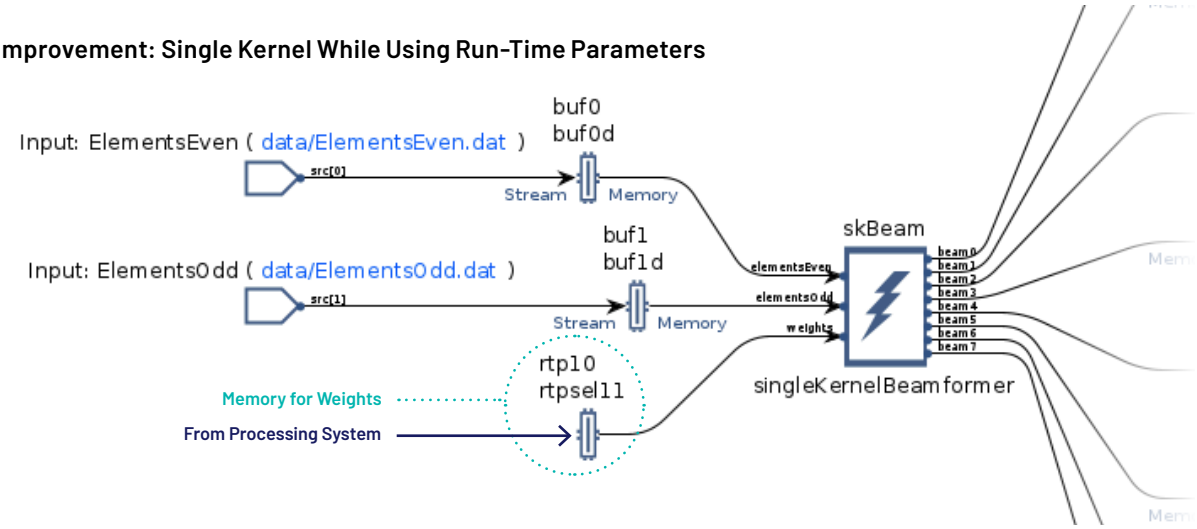


EXPERIMENTING WITH PARAMETERS

At this point in the design, the beamforming kernel received its weights from an input AXI4-Stream. Since these weights did not need to be updated frequently, we found the opportunity to further improve the kernel by using run-time parameters (RTPs). RTPs can be single values or entire arrays that are passed from either the processing system (PS) or another kernel.

Using RTPs to store weights alongside the kernel replaced the need for them to be streamed from the Programmable Logic (PL), simplifying the design. This approach can improve design throughput by reducing the amount of data streams contending for routing resources within the AIE array.

Further Improvement: Single Kernel While Using Run-Time Parameters



Kernel weights delivered using RTP

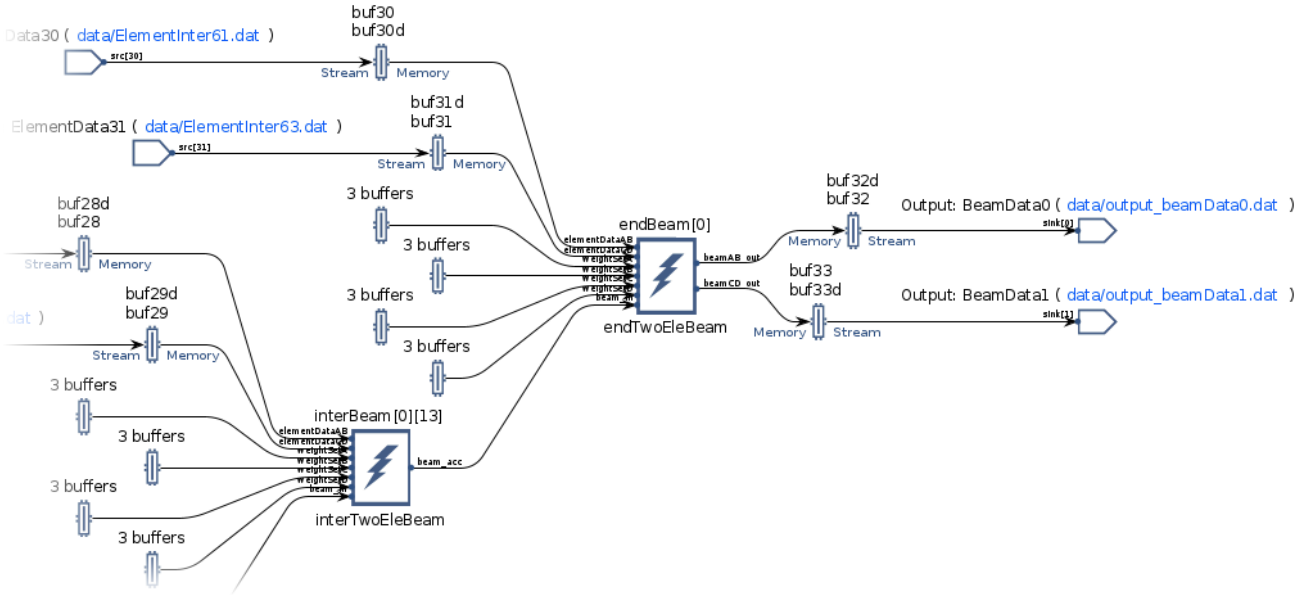
```
void singleKernelBeamformer (
        input_window_cint16 * restrict elementsEven,
        input_window_cint16 * restrict elementsOdd,
        const cint16 (&weights)[512],
```

MULTI-KERNEL GRAPH

So far, we have explored several example AIE kernels. But what about larger applications? To effectively use the AIE array, designers must consider how to divide their application into multiple kernels that work together.

To demonstrate this, we created a graph with 16 kernels where each of the kernels computes part of the input elements. The intermediate results are passed to the next kernel in the AIE array through a cascade path. The last kernel finishes the calculations and outputs the data to the FPGA fabric.

Multi-Kernel Graph



HIGH-PERFORMANCE APPLICATIONS

For the most demanding applications, designers should consider how to structure graphs so they can scale efficiently across many AIEs. The physical location of kernels and I/O interfaces is also important. A good starting point is to map the dataflow of the application, as this will guide the other aspects of the AIE design.

Input data should flow directly upward from the logic fabric through the AIE array. This is because the AIE array's AXI4-Stream interconnect is non-symmetrical, with more paths traveling north than any other direction (see the AIE Array Diagram on page 3).

If one of the input streams is broadcast to many kernels, it will occupy more routing as it branches out to each of the destinations.

Within the application, designers should take advantage of the cascade path to forward data between kernels when possible. To transfer low-bandwidth data, designers should consider using RTPs, which can be transferred both between kernels as well as the processing system. These techniques will reduce the total number of data streams and make the application more flexible and easier to implement.

CONCLUSION

Necessity is most assuredly the mother of invention. Today's exploding data volumes, combined with the increasing need for energy efficiency, require a new generation of processing solutions. The AMD Versal™ adaptive SoC meets those demands. Now a single, hardened, heterogeneous silicon chip provides the computational performance of multiple devices while using much less energy.

The landscape has changed and the journey has just begun toward more complex, secure and purpose-built solutions and systems for the next generation in aerospace and defense capabilities.

Acknowledgments

The authors would like to recognize the valuable contributions and support given by **Kok Lee**, **Berk Adanur**, and **Don Stickels**.

About Mercury

Mercury Systems is a technology company that delivers mission-critical processing power to the edge to solve the most pressing aerospace and defense challenges. Combining technologies and expertise developed for more than 40 years, the Mercury Processing Platform offers customers a unique advantage to unleash breakthrough capabilities. It spans the full breadth of signal processing—from RF front end to the human-machine interface—enabling customers to turn data into decisions with standard products and custom solutions from silicon to system scale. Mercury's products and solutions are deployed in more than 300 programs and across 35 countries. The company is headquartered in Andover, Massachusetts, and has 23 locations worldwide. To learn more, visit mrcy.com. (Nasdaq: MRCY)

Corporate Headquarters

50 Minuteman Road
Andover, MA 01810 USA
+1 978.967.1401 tel
+1 866.627.6951 tel
+1 978.256.3599 fax

International Headquarters
Mercury International

Avenue Eugène-Lance, 38
PO Box 584
CH-1212 Grand-Lancy 1
Geneva, Switzerland
+41 22 884 51 00 tel

Learn more about the SCFE6931
mrcy.com/acap

Collaborate with us on your Versal solution:
mrcy.com/contactus