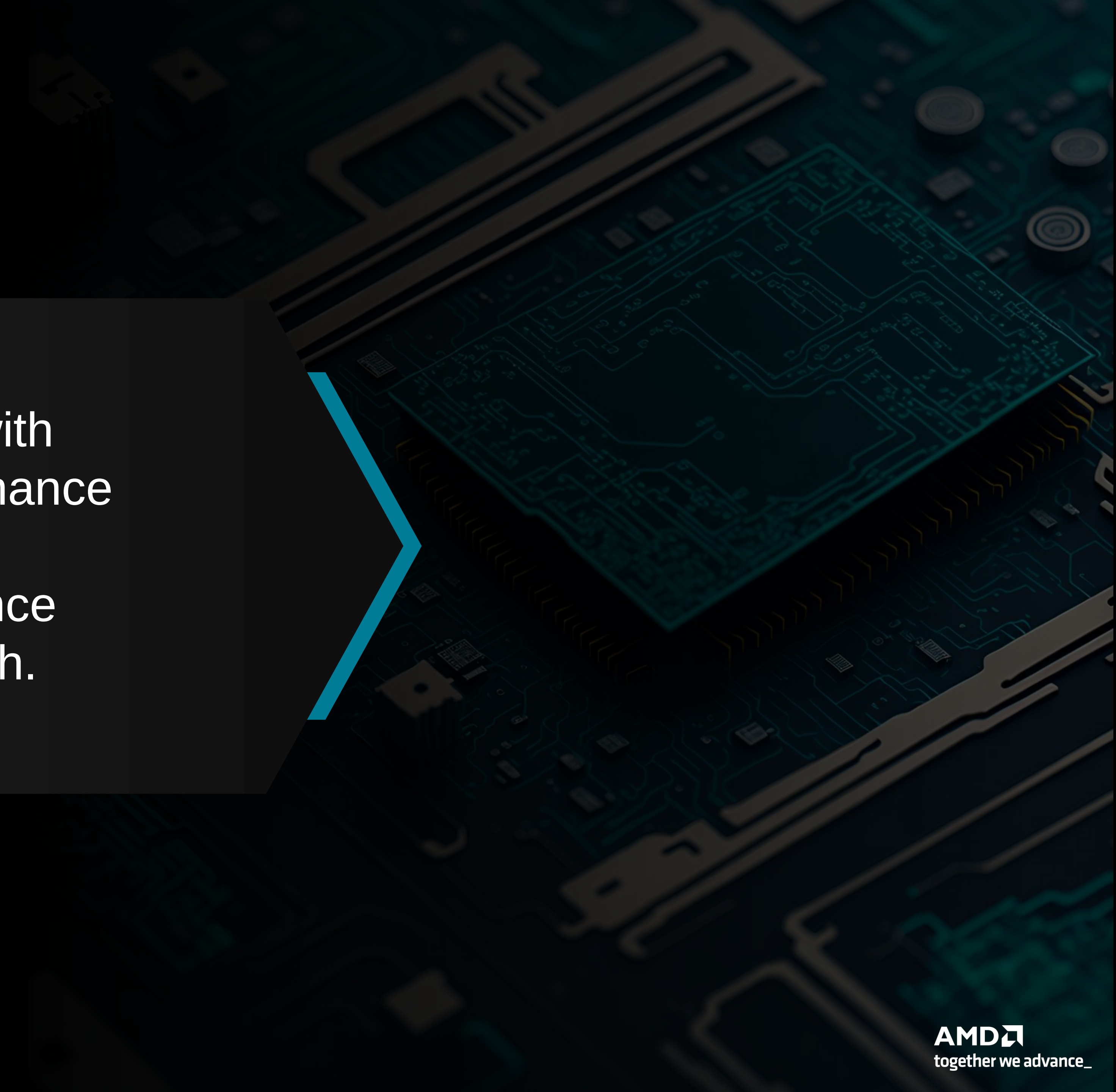
Introduction to FPGA, FINN and Brevitas

Dr. Mario Ruiz
AMD University Program

AMD
together we advance_

AUP Vision

Empower academics with AMD technology to enhance teaching and learning experiences and advance state-of-the-art research.

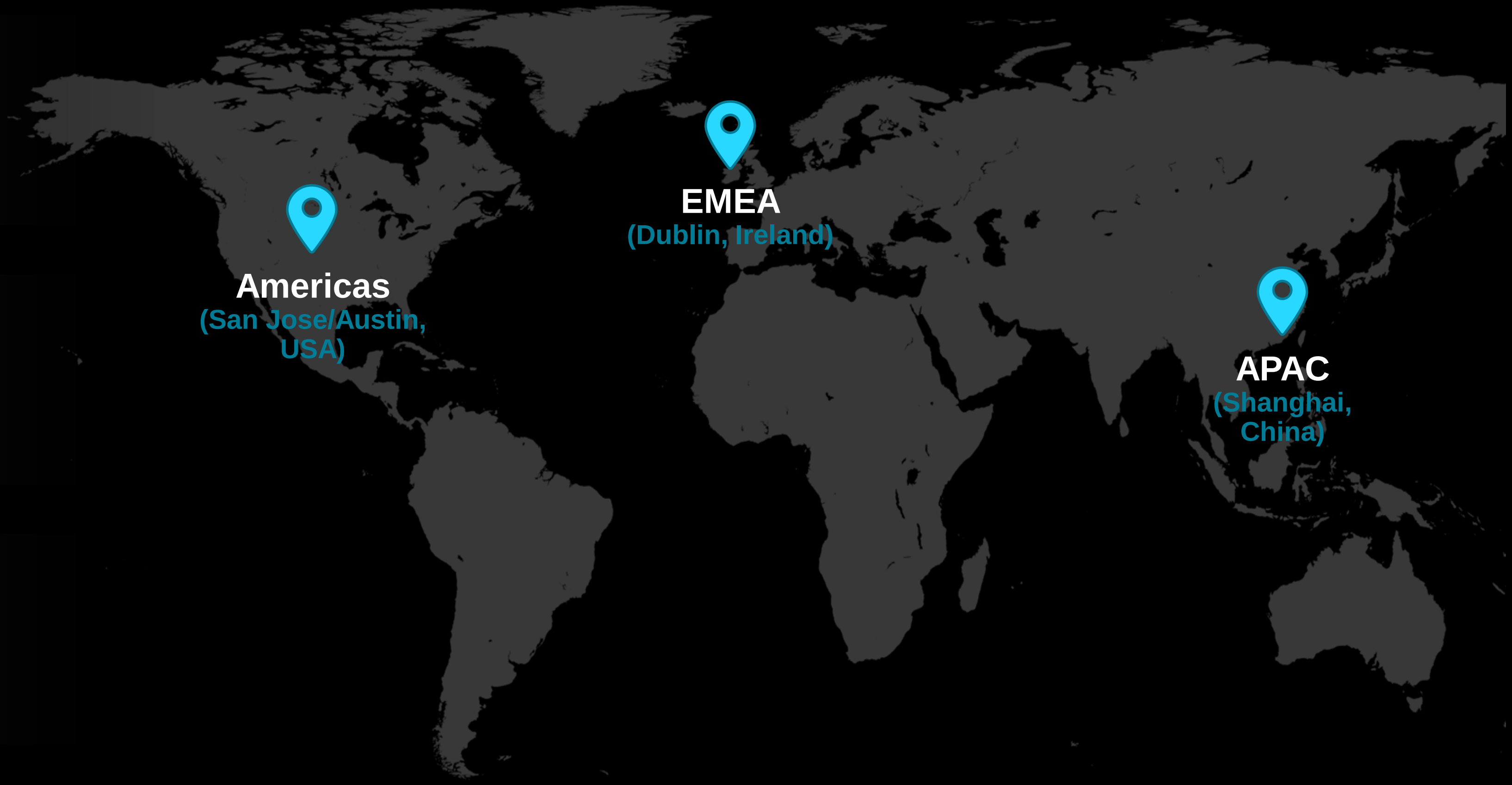


Our Team

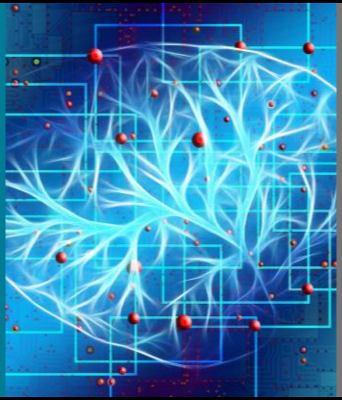
Dedicated world-wide technical team

Supporting High Performance and Adaptive Compute

25+ years experience working with academia



What We Offer



Research Programs

Donation Program

Teaching Resources

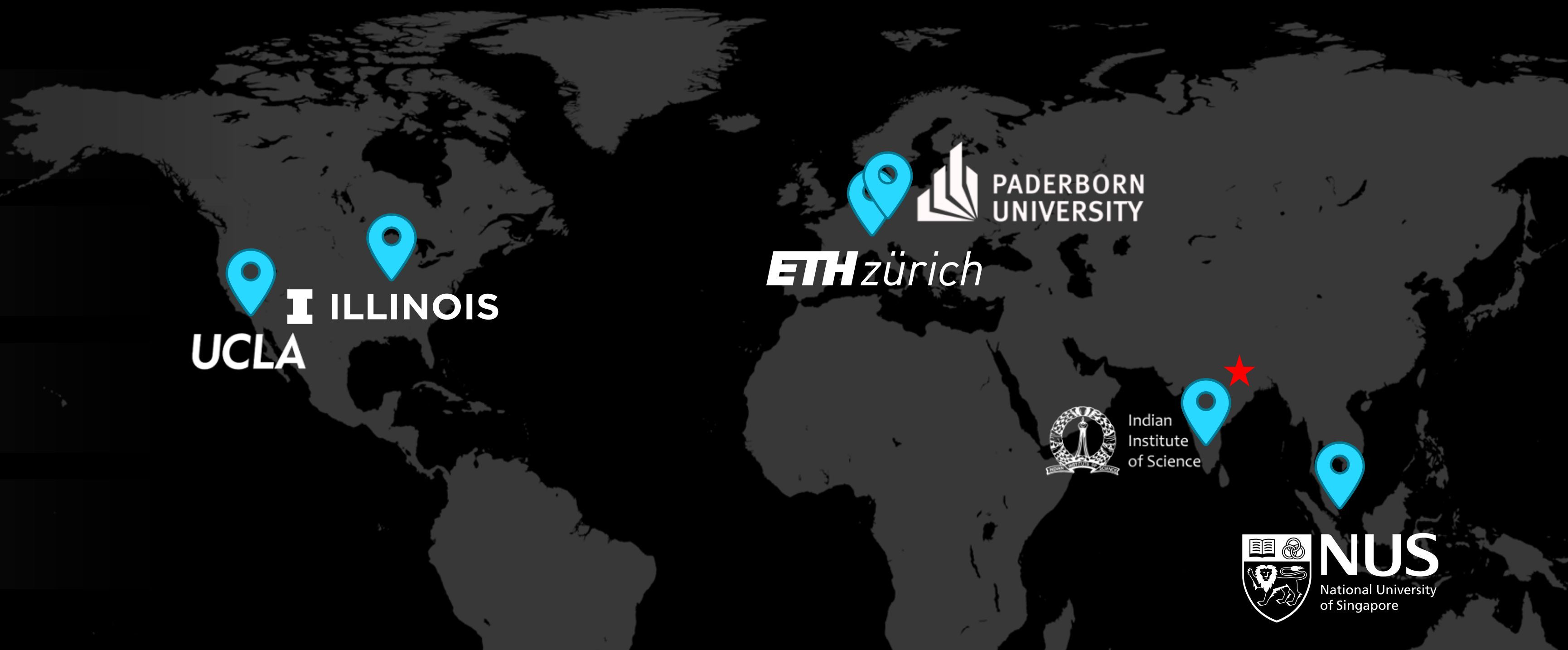
Training

Academic Solutions

Support

HACCs: Heterogeneous Accelerated Compute Clusters

- Remote access to Adaptive Compute hardware
- HACC user group meetings
- Access to AMD researchers
- Collaboration opportunities

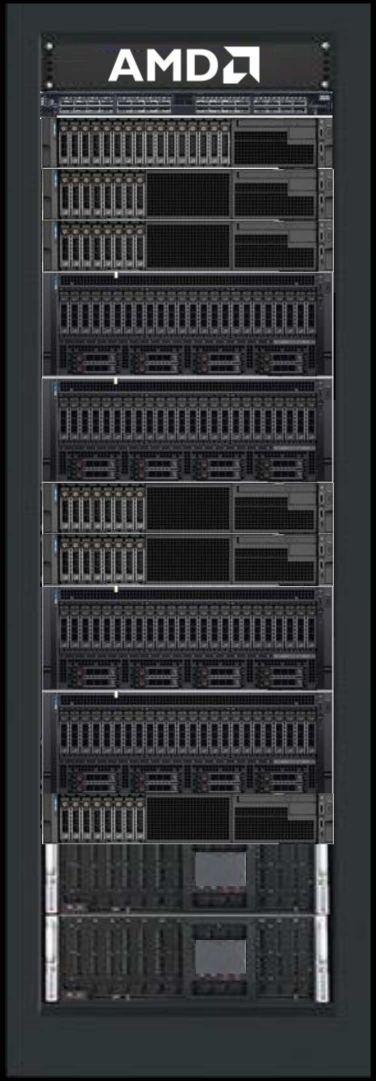

PADERBORN UNIVERSITY
ETH zürich
ILLINOIS
UCLA
Indian Institute of Science
NUS National University of Singapore

AMD EPYC | AMD INSTINCT | AMD ALVEO | AMD VERSAL

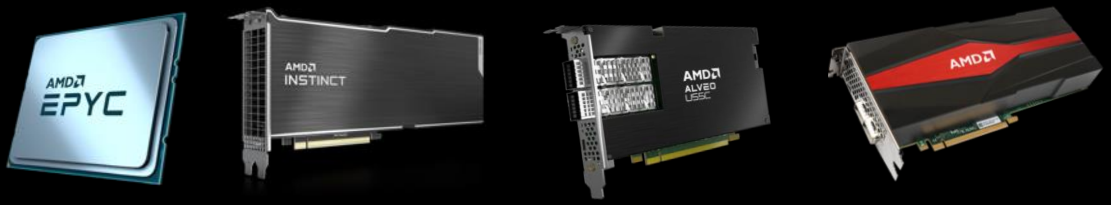
www.amd-haccs.io

★ Newest HACC at IISc, Bangalore

HACC Adaptive Computing Hardware



- HACC hardware consists of:
  - Compute and Alveo™ nodes (initially U250 and U280 with HBM)
  - Latest heterogeneous nodes (SMC 4124GS) include:
    - 2 EPYC™ 3rd generation CPUs
    - 4 AMD Instinct™ MI210 GPUs
    - 2 Alveo U55C FPGA with HBM
    - 2 VCK5000 Versal Adaptive SoC with AIEs
    - Run-time via AMD ROCm™, XRT
    - SW development via HIP, Vitis, frameworks
  - 100G network

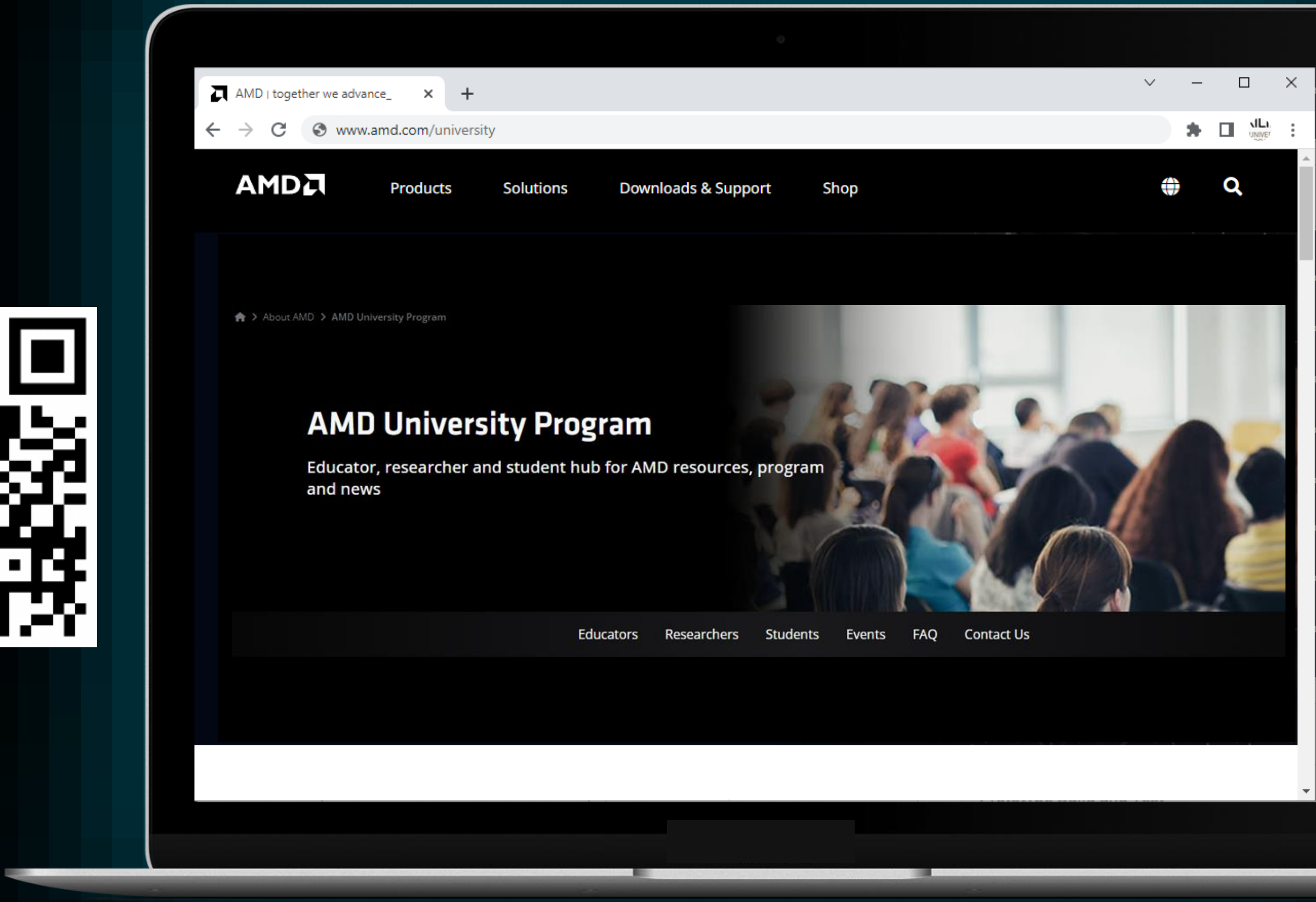


- Community hub for researchers
  - Support from in-house AMD research groups
  - Reproducible results & experiments

Contact Us

Visit our website to:

- Discover our research programs
- Access educational resources
- Submit a donation request
- Find training & other events



AMD University Program

Educator, researcher and student hub for AMD resources, program and news

Email us:

aup@amd.com

www.amd.com/AUP

What is Adaptive Computing?

Optimize for the Workload

Domain-Specific Architecture for your exact requirements, accelerating the whole application

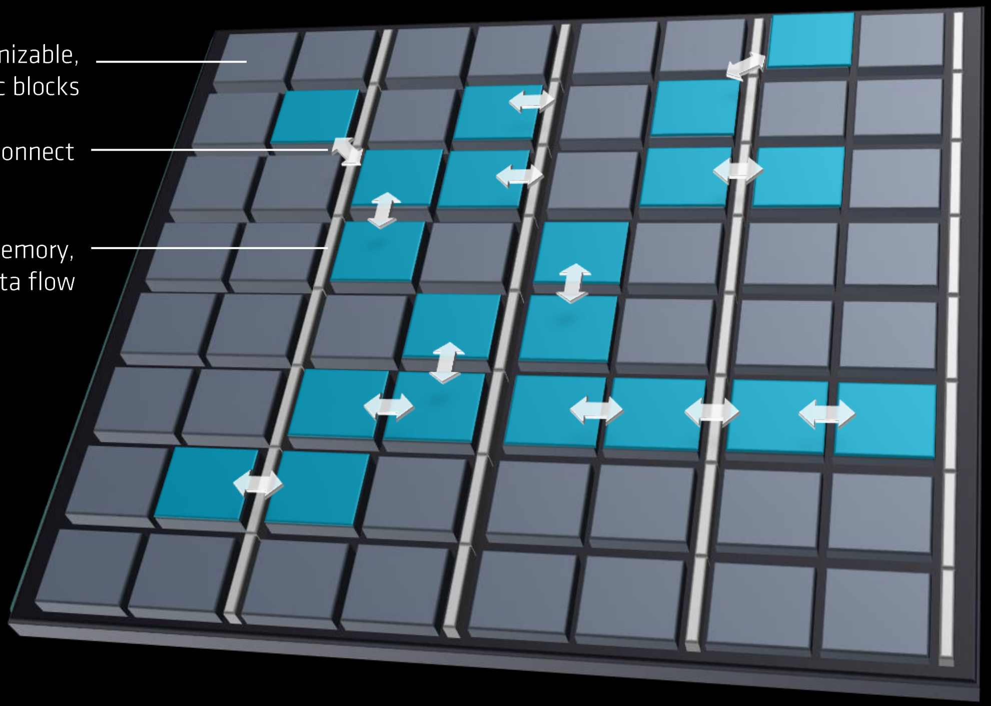
Adapt as Algorithms Change

Re-implement the silicon after deployment, adapting to evolving use cases

Accelerate Pace of Innovation

Keep pace with fast moving markets and rapid innovation cycles, e.g., AI algorithms

Adaptive Hardware (“FPGA”)
Conceptual Representation



Millions of customizable, configurable logic blocks

Flexible Interconnect

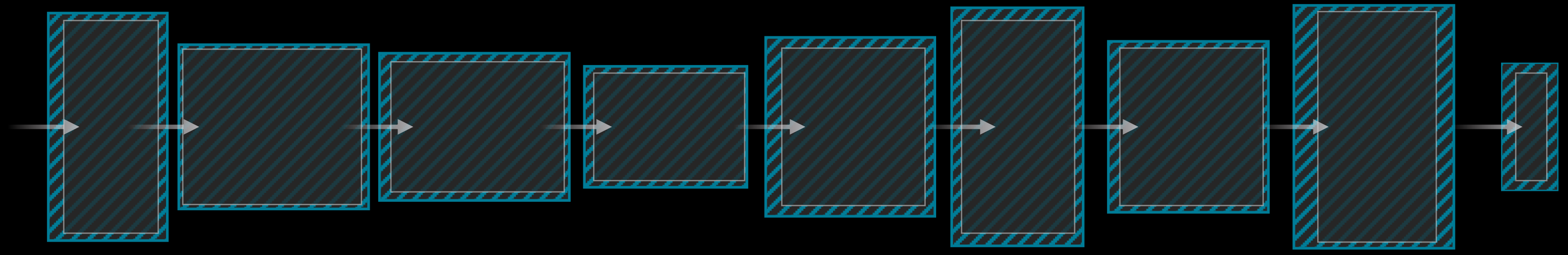
Adaptable memory, customize to data flow

Matching the Architecture to the Application

Custom Data Flow, Custom Memory Hierarchy, Custom Precision

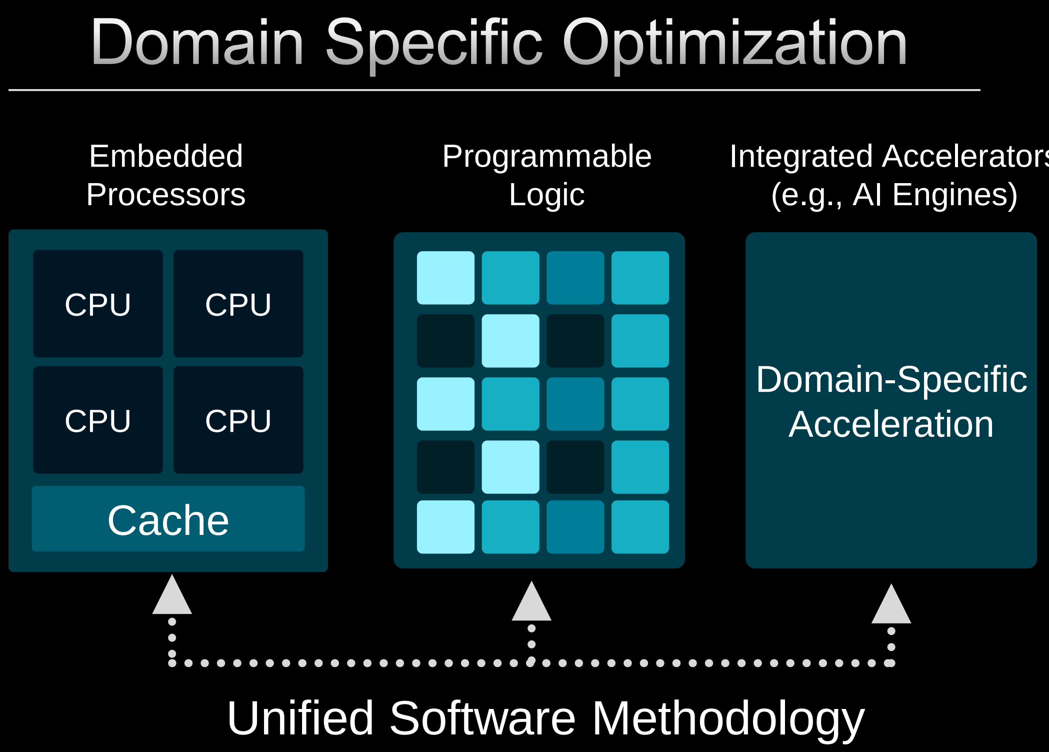
□ APPLICATION DOMAIN

▨ ADAPTIVE ARCHITECTURE (After Programming)



Evolution to Heterogeneous Platforms

- From FPGAs to adaptive SoCs → matching the engine to the workload
- Balancing diverse technologies for domain-specific requirements


Domain Specific Optimization
Embedded Processors
Programmable Logic
Integrated Accelerators (e.g., AI Engines)
CPU
CPU
CPU
CPU
Cache
Domain-Specific Acceleration
Unified Software Methodology

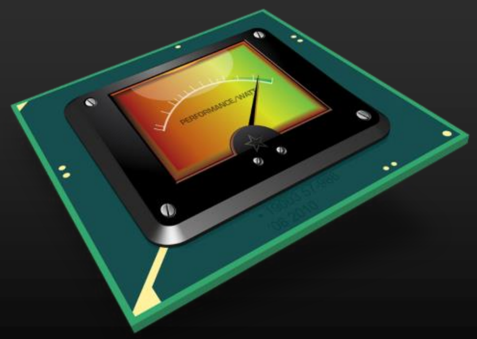
Field Programmable Gate Array (FPGA)

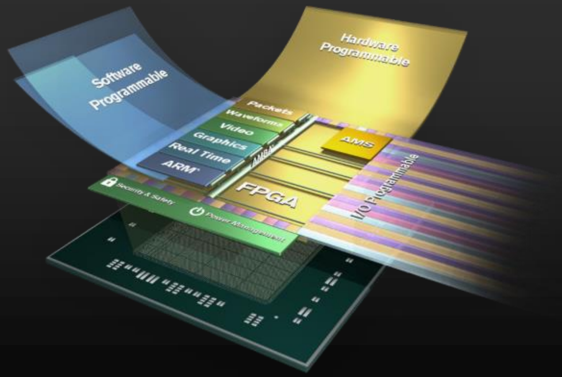
- Semiconductor devices
- Programmed and reprogrammed by a user
  - Configuration attributes manipulated after manufacturing
  - Matrix of configurable logic blocks (CLBs)
  - Dedicated specialized logic
  - Flexible programmable interconnects
- Ideal fit for many different workloads
  - Massive parallelism
- Hardware adaptability is a unique differentiator from CPUs and GPUs
- Invented in 1985

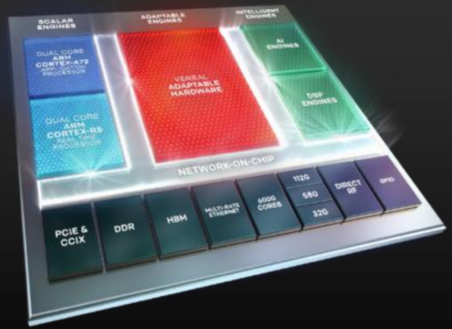
Applications

- Automotive
- Broadcast & Pro AV
- Consumer Electronics
- Data Center
- High Performance Computing and Data Storage
- Industrial
- Medical
- Video & Image Processing
- Wired Communications
- Wireless Communications

Core Adaptable Hardware Technologies







FPGAs

From high-bandwidth connectivity to massive compute engines

AMD SPARTAN AMD ARTIX AMD KINTEX AMD VIRTEX

SoCs

Multi-processing subsystem with Arm® cores and integrated FPGA logic

AMD ZYNQ

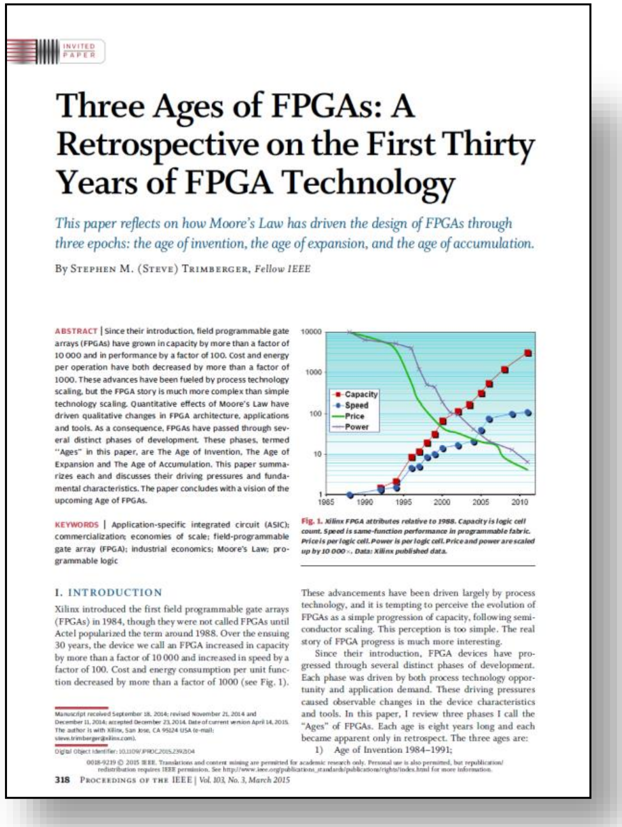
Adaptive SoCs

Adaptive Compute Acceleration Platforms for any application, any developer

AMD VERSAL

Three Ages of FPGAs

- A Retrospective on the First Thirty Years of FPGA Technology
- S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," in Proceedings of the IEEE, vol. 103, no. 3, pp. 318-331, March 2015, DOI: 10.1109/JPROC.2015.2392104



Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology

This paper reflects on how Moore's Law has driven the design of FPGAs through three epochs: the age of invention, the age of expansion, and the age of accumulation.

By Stephen M. (Steve) Trimberger, *Fellow IEEE*

ABSTRACT | Since their introduction, field programmable gate arrays (FPGAs) have grown in capacity by more than a factor of 10 000 and in performance by a factor of 100. Cost and energy per operation have both decreased by more than a factor of 1000. These advances have been fueled by process technology scaling, but the FPGA story is much more complex than simple technology scaling. Quantitative effects of Moore's Law have driven qualitative changes in FPGA architecture, applications and tools. As a consequence, FPGAs have passed through several distinct phases of development. These phases, termed "Ages" in this paper, are The Age of Invention, The Age of Expansion and The Age of Accumulation. This paper summarizes each and discusses their driving pressures and fundamental characteristics. The paper concludes with a vision of the upcoming Age of FPGAs.

KEYWORDS | Application-specific integrated circuit (ASIC); commercialization; economies of scale; field-programmable gate array (FPGA); industrial economics; Moore's Law; programmable logic

Fig. 1. *Xilinx FPGA attributes relative to 1988. Capacity is logic cell count. Speed is same-function performance in programmable fabric. Price is per logic cell. Power is per logic cell. Price and power are scaled up by 10 000×. Data: Xilinx published data.*

I. INTRODUCTION

Xilinx introduced the first field programmable gate arrays (FPGAs) in 1984, though they were not called FPGAs until Actel popularized the term around 1988. Over the ensuing 30 years, the device we call an FPGA increased in capacity by more than a factor of 10 000 and increased in speed by a factor of 100. Cost and energy consumption per unit function decreased by more than a factor of 1000 (see Fig. 1).

These advancements have been driven largely by process technology, and it is tempting to perceive the evolution of FPGAs as a simple progression of capacity, following semiconductor scaling. This perception is too simple. The real story of FPGA progress is much more interesting.

Since their introduction, FPGA devices have progressed through several distinct phases of development. Each phase was driven by both process technology opportunity and application demand. These driving pressures caused observable changes in the device characteristics and tools. In this paper, I review three phases I call the "Ages" of FPGAs. Each age is eight years long and each became apparent only in retrospect. The three ages are:

1) Age of Invention 1984–1991;

Manuscript received September 18, 2014; revised November 21, 2014 and December 11, 2014; accepted December 23, 2014. Date of current version April 14, 2015.
The author is with Xilinx, San Jose, CA 95124 USA (e-mail: steve.trimberger@xilinx.com).
Digital Object Identifier: 10.1109/JPROC.2015.2392104

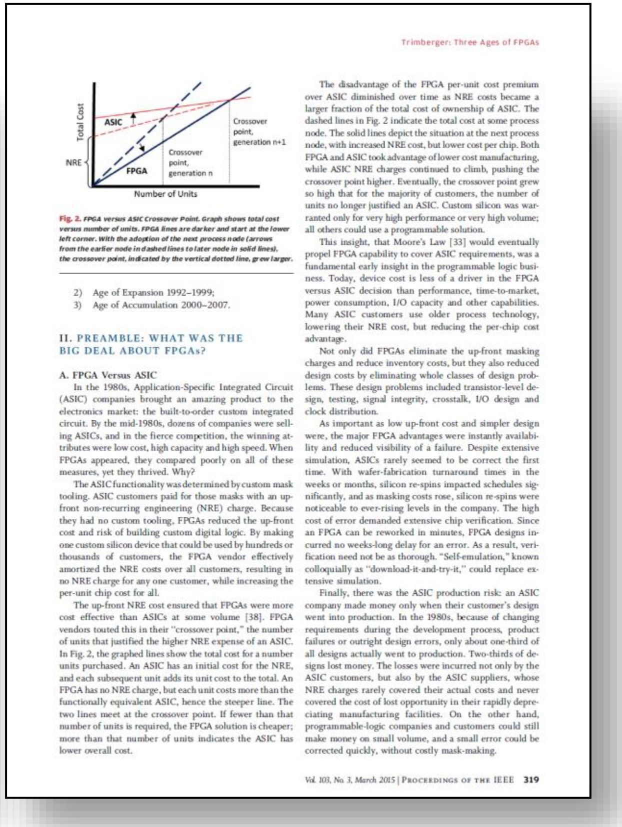

Total Cost
ASIC
NRE
FPGA
Crossover point, generation n+1
Crossover point, generation n
Number of Units

Fig. 2. *FPGA versus ASIC Crossover Point. Graph shows total cost versus number of units. FPGA lines are darker and start at the lower left corner. With the adoption of the next process node (arrows from the earlier node in dashed lines to later node in solid lines), the crossover point, indicated by the vertical dotted line, grew larger.*

2) Age of Expansion 1992–1999;
3) Age of Accumulation 2000–2007.

II. PREAMBLE: WHAT WAS THE BIG DEAL ABOUT FPGAs?

A. FPGA Versus ASIC

In the 1980s, Application-Specific Integrated Circuit (ASIC) companies brought an amazing product to the electronics market: the built-to-order custom integrated circuit. By the mid-1980s, dozens of companies were selling ASICs, and in the fierce competition, the winning attributes were low cost, high capacity and high speed. When FPGAs appeared, they compared poorly on all of these measures, yet they thrived. Why?

The ASIC functionality was determined by custom mask tooling. ASIC customers paid for those masks with an up-front non-recurring engineering (NRE) charge. Because they had no custom tooling, FPGAs reduced the up-front cost and risk of building custom digital logic. By making one custom silicon device that could be used by hundreds or thousands of customers, the FPGA vendor effectively amortized the NRE costs over all customers, resulting in no NRE charge for any one customer, while increasing the per-unit chip cost for all.

The up-front NRE cost ensured that FPGAs were more cost effective than ASICs at some volume [38]. FPGA vendors touted this in their "crossover point," the number of units that justified the higher NRE expense of an ASIC. In Fig. 2, the graphed lines show the total cost for a number units purchased. An ASIC has an initial cost for the NRE, and each subsequent unit adds its unit cost to the total. An FPGA has no NRE charge, but each unit costs more than the functionally equivalent ASIC, hence the steeper line. The two lines meet at the crossover point. If fewer than that number of units is required, the FPGA solution is cheaper; more than that number of units indicates the ASIC has lower overall cost.

The disadvantage of the FPGA per-unit cost premium over ASIC diminished over time as NRE costs became a larger fraction of the total cost of ownership of ASIC. The dashed lines in Fig. 2 indicate the total cost at some process node. The solid lines depict the situation at the next process node, with increased NRE cost, but lower cost per chip. Both FPGA and ASIC took advantage of lower cost manufacturing, while ASIC NRE charges continued to climb, pushing the crossover point higher. Eventually, the crossover point grew so high that for the majority of customers, the number of units no longer justified an ASIC. Custom silicon was warranted only for very high performance or very high volume; all others could use a programmable solution.

This insight, that Moore's Law [33] would eventually propel FPGA capability to cover ASIC requirements, was a fundamental early insight in the programmable logic business. Today, device cost is less of a driver in the FPGA versus ASIC decision than performance, time-to-market, power consumption, I/O capacity and other capabilities. Many ASIC customers use older process technology, lowering their NRE cost, but reducing the per-chip cost advantage.

Not only did FPGAs eliminate the up-front masking charges and reduce inventory costs, but they also reduced design costs by eliminating whole classes of design problems. These design problems included transistor-level design, testing, signal integrity, crosstalk, I/O design and clock distribution.

As important as low up-front cost and simpler design were, the major FPGA advantages were instantly availability and reduced visibility of a failure. Despite extensive simulation, ASICs rarely seemed to be correct the first time. With wafer-fabrication turnaround times in the weeks or months, silicon re-spins impacted schedules significantly, and as masking costs rose, silicon re-spins were noticeable to ever-rising levels in the company. The high cost of error demanded extensive chip verification. Since an FPGA can be reworked in minutes, FPGA designs incurred no weeks-long delay for an error. As a result, verification need not be as thorough. "Self-emulation," known colloquially as "download-it-and-try-it," could replace extensive simulation.

Finally, there was the ASIC production risk: an ASIC company made money only when their customer's design went into production. In the 1980s, because of changing requirements during the development process, product failures or outright design errors, only about one-third of all designs actually went to production. Two-thirds of designs lost money. The losses were incurred not only by the ASIC customers, but also by the ASIC suppliers, whose NRE charges rarely covered their actual costs and never covered the cost of lost opportunity in their rapidly depreciating manufacturing facilities. On the other hand, programmable-logic companies and customers could still make money on small volume, and a small error could be corrected quickly, without costly mask-making.

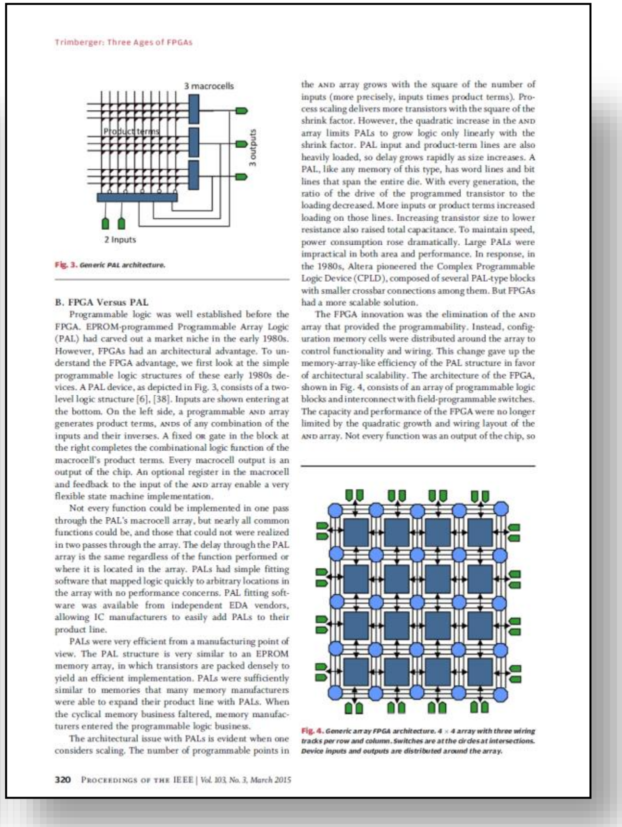


Fig. 3. *Generic PAL architecture.*

B. FPGA Versus PAL

Programmable logic was well established before the FPGA. EPROM-programmed Programmable Array Logic (PAL) had carved out a market niche in the early 1980s. However, FPGAs had an architectural advantage. To understand the FPGA advantage, we first look at the simple programmable logic structures of these early 1980s devices. A PAL device, as depicted in Fig. 3, consists of a two-level logic structure [6], [38]. Inputs are shown entering at the bottom. On the left side, a programmable AND array generates product terms, ANDs of any combination of the inputs and their inverses. A fixed OR gate in the block at the right completes the combinational logic function of the macrocell's product terms. Every macrocell output is an output of the chip. An optional register in the macrocell and feedback to the input of the AND array enable a very flexible state machine implementation.

Not every function could be implemented in one pass through the PAL's macrocell array, but nearly all common functions could be, and those that could not were realized in two passes through the array. The delay through the PAL array is the same regardless of the function performed or where it is located in the array. PALs had simple fitting software that mapped logic quickly to arbitrary locations in the array with no performance concerns. PAL fitting software was available from independent EDA vendors, allowing IC manufacturers to easily add PALs to their product line.

PALs were very efficient from a manufacturing point of view. The PAL structure is very similar to an EPROM memory array, in which transistors are packed densely to yield an efficient implementation. PALs were sufficiently similar to memories that many memory manufacturers were able to expand their product line with PALs. When the cyclical memory business faltered, memory manufacturers entered the programmable logic business.

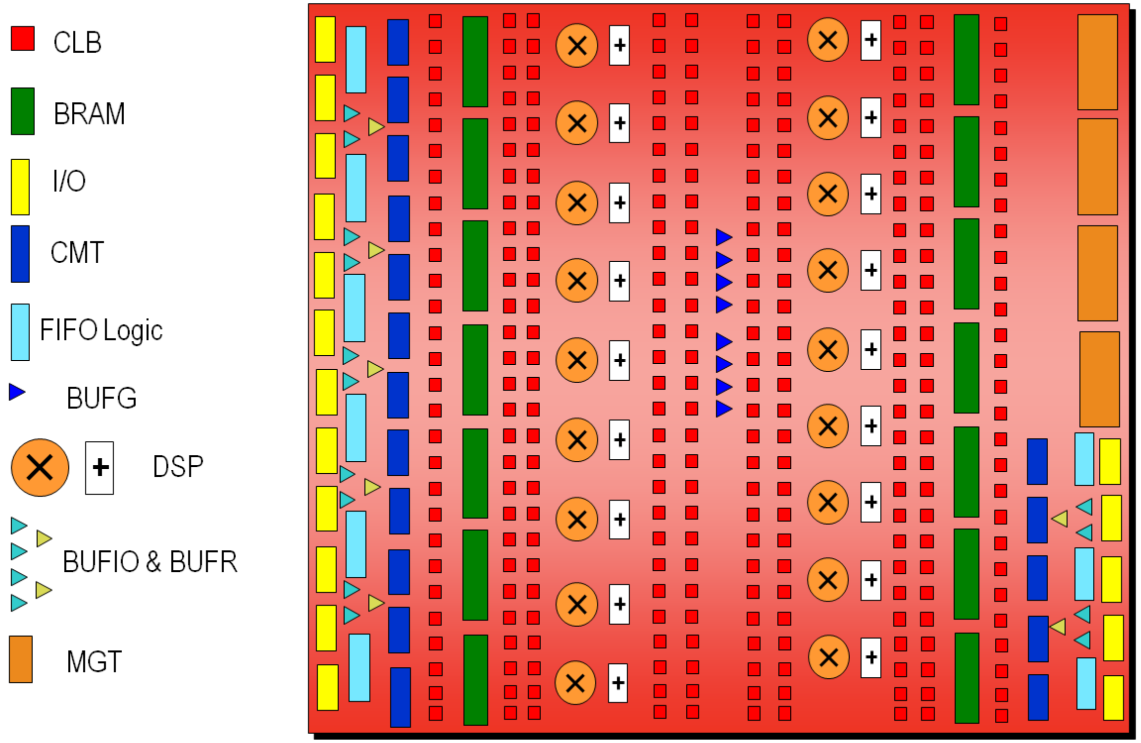
The architectural issue with PALs is evident when one considers scaling. The number of programmable points in the AND array grows with the square of the number of inputs (more precisely, inputs times product terms). Process scaling delivers more transistors with the square of the shrink factor. However, the quadratic increase in the AND array limits PALs to grow logic only linearly with the shrink factor. PAL input and product-term lines are also heavily loaded, so delay grows rapidly as size increases. A PAL, like any memory of this type, has word lines and bit lines that span the entire die. With every generation, the ratio of the drive of the programmed transistor to the loading decreased. More inputs or product terms increased loading on those lines. Increasing transistor size to lower resistance also raised total capacitance. To maintain speed, power consumption rose dramatically. Large PALs were impractical in both area and performance. In response, in the 1980s, Altera pioneered the Complex Programmable Logic Device (CPLD), composed of several PAL-type blocks with smaller crossbar connections among them. But FPGAs had a more scalable solution.

The FPGA innovation was the elimination of the AND array that provided the programmability. Instead, configuration memory cells were distributed around the array to control functionality and wiring. This change gave up the memory-array-like efficiency of the PAL structure in favor of architectural scalability. The architecture of the FPGA, shown in Fig. 4, consists of an array of programmable logic blocks and interconnect with field-programmable switches. The capacity and performance of the FPGA were no longer limited by the quadratic growth and wiring layout of the AND array. Not every function was an output of the chip, so

Fig. 4. *Generic array FPGA architecture. 4 × 4 array with three wiring tracks per row and column. Switches are at the circles at intersections. Device inputs and outputs are distributed around the array.*

FPGA: 7-Series Architecture

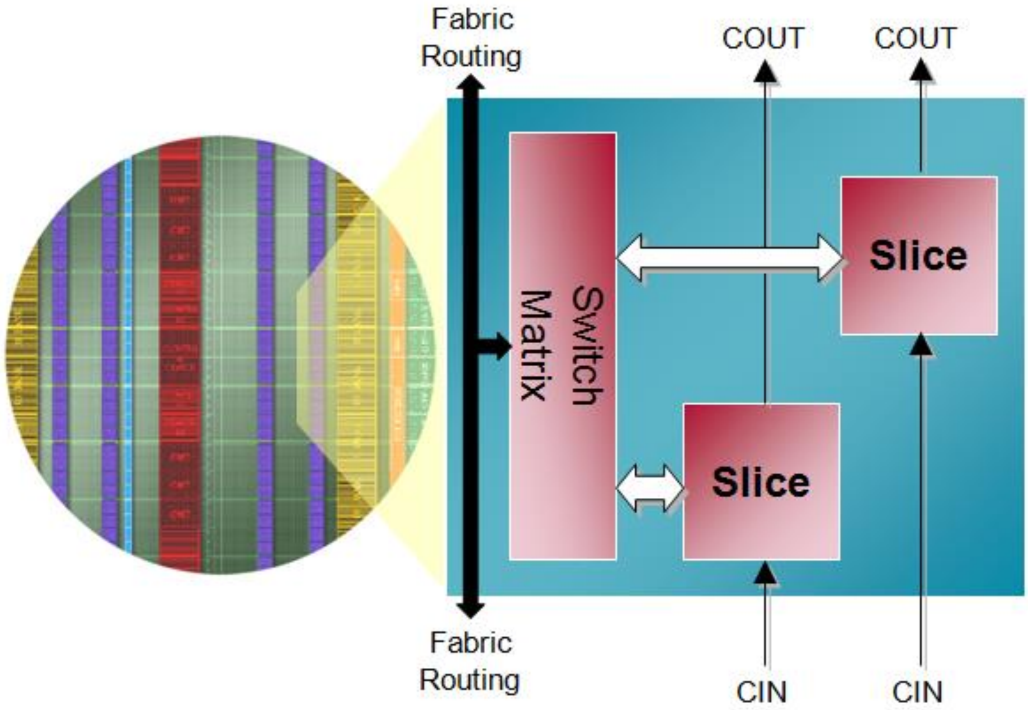
- Logic elements distributed on regular columns
  - Scalability from low-cost to high-performance
- High-speed IO
- Clock management
- Interconnect matrix
  - Routing resources



Artix-7 Architecture Overview

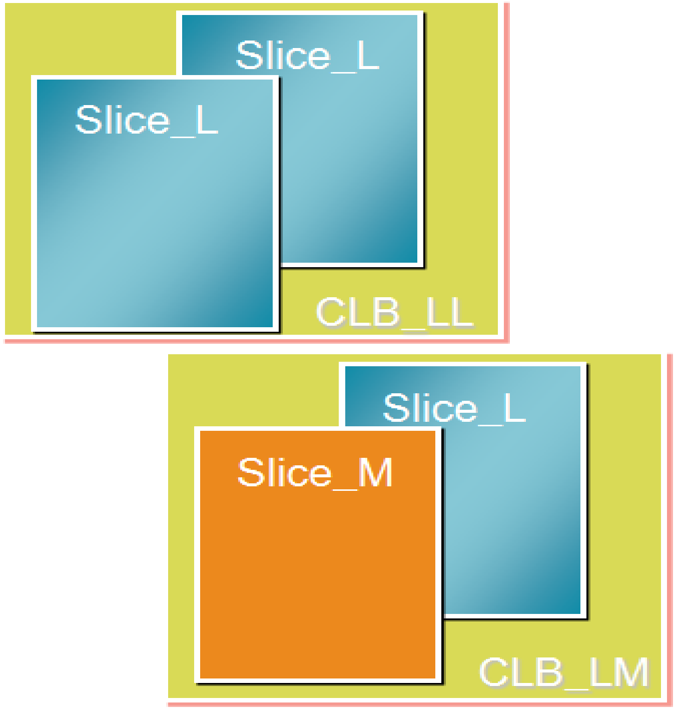
Configurable Logic Block (CLB)

- Primary resource for design in AMD FPGAs
  - Combinatorial functions
  - Flip-flops
- CLB contains two slices
- Connected to switch matrix for routing to other FPGA resources
  - Carry chain runs vertically



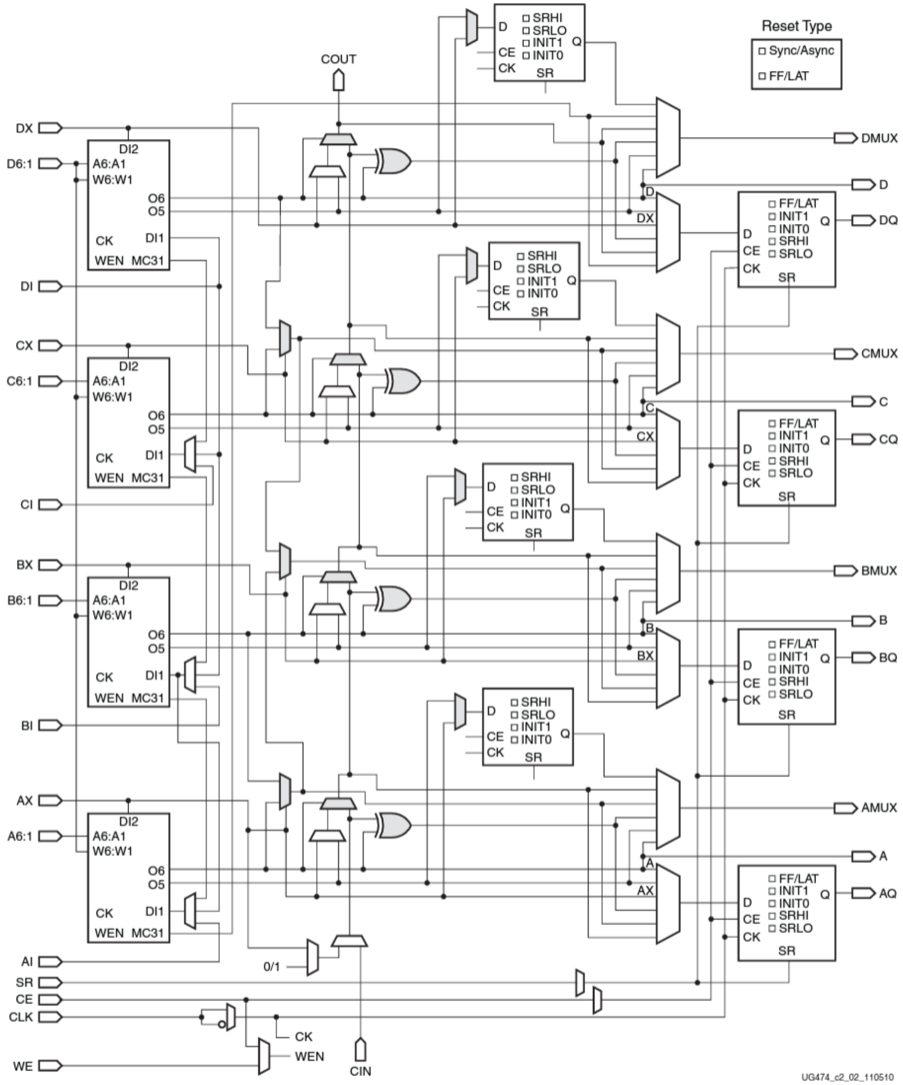
Two Types of CLB Slices

- SLICEM: Full slice
  - Can be used for logic, memory and shift register LUT
  - Has wide multiplexers and carry chain
- SLICEL: Logic and arithmetic only
  - LUT can only be used for logic (not memory)
  - Has wide multiplexers and carry chain



Slice Resource

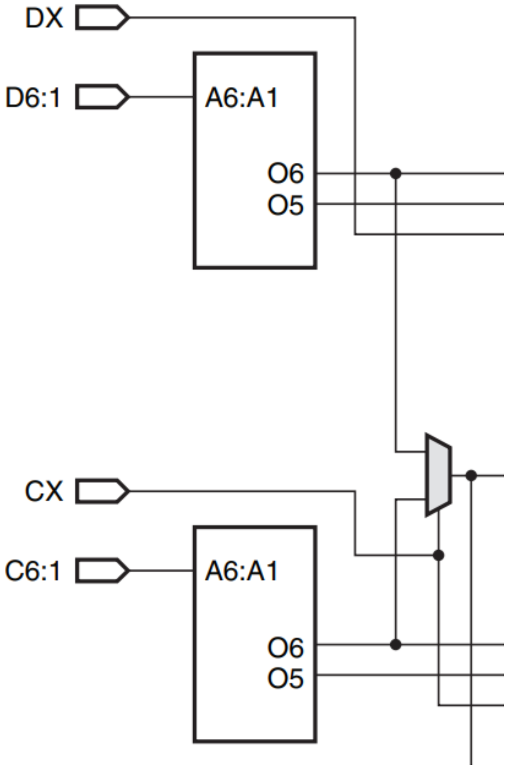
- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
    - Four additional flip-flops
- The implementation tool will pack multiple slices in the same CLB if certain rules are followed



UG474_c2_02_110510

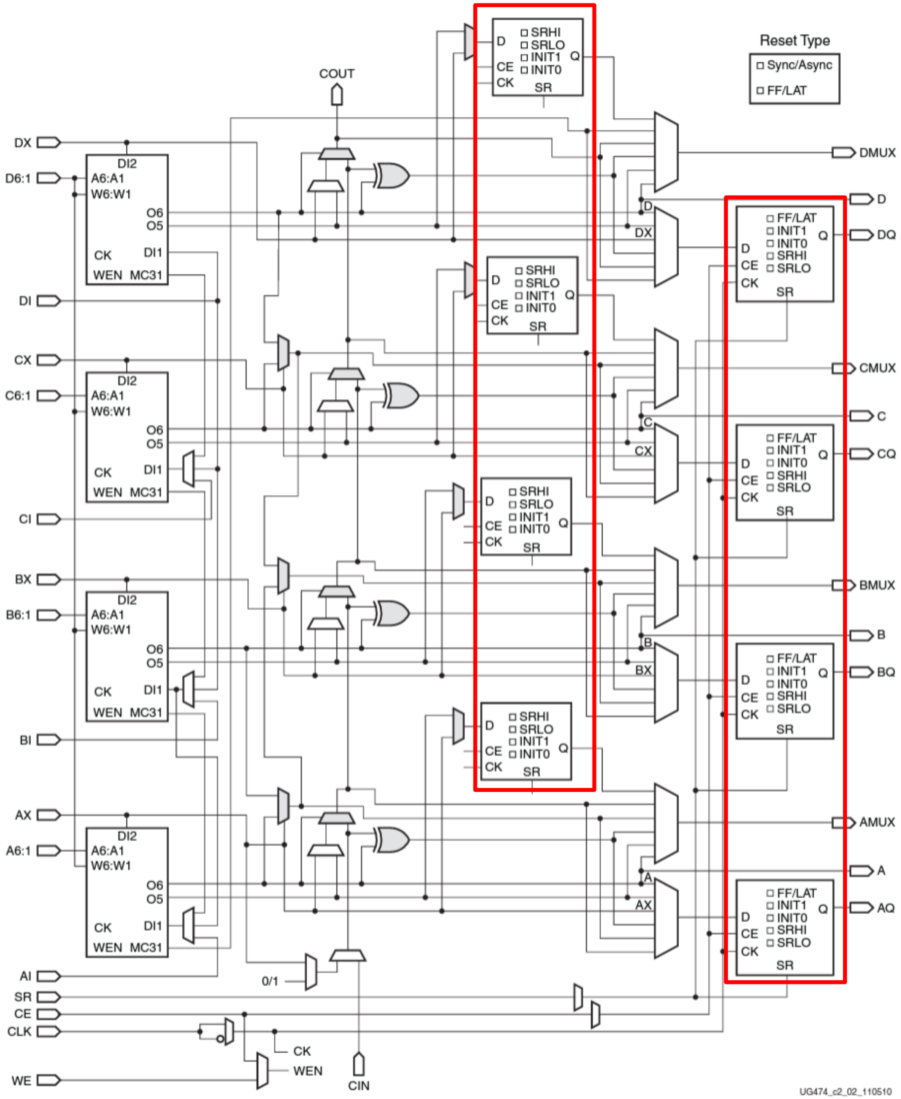
6-Input LUT with Dual Output

- LUTs can be two 5-input LUTs with common input
  - Minimal speed impact to a 6-input LUT
  - One or two outputs
- Any combinatorial function of six variables or two functions of five variables



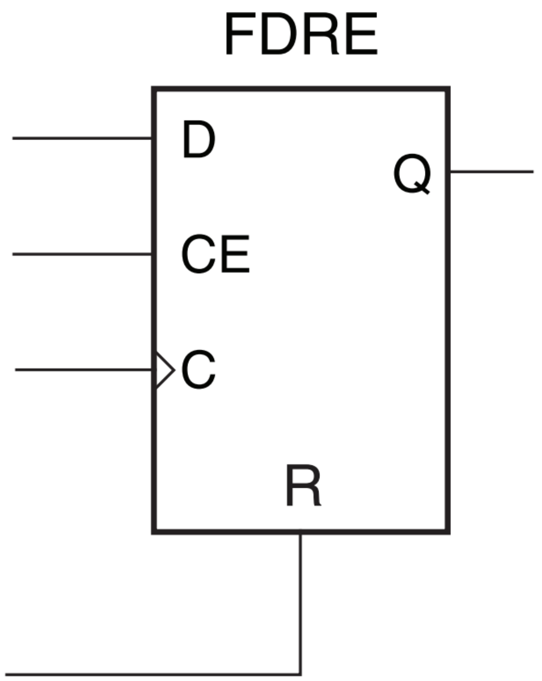
Slice Flip-Flops and Flip-Flop/Latches

- Each slice has four flip-flop/latches (FF/L)
  - Can be configured as either flip-flops or latches
- Each slice also has four flip-flops (FF)



Slice Flip-Flop Capabilities

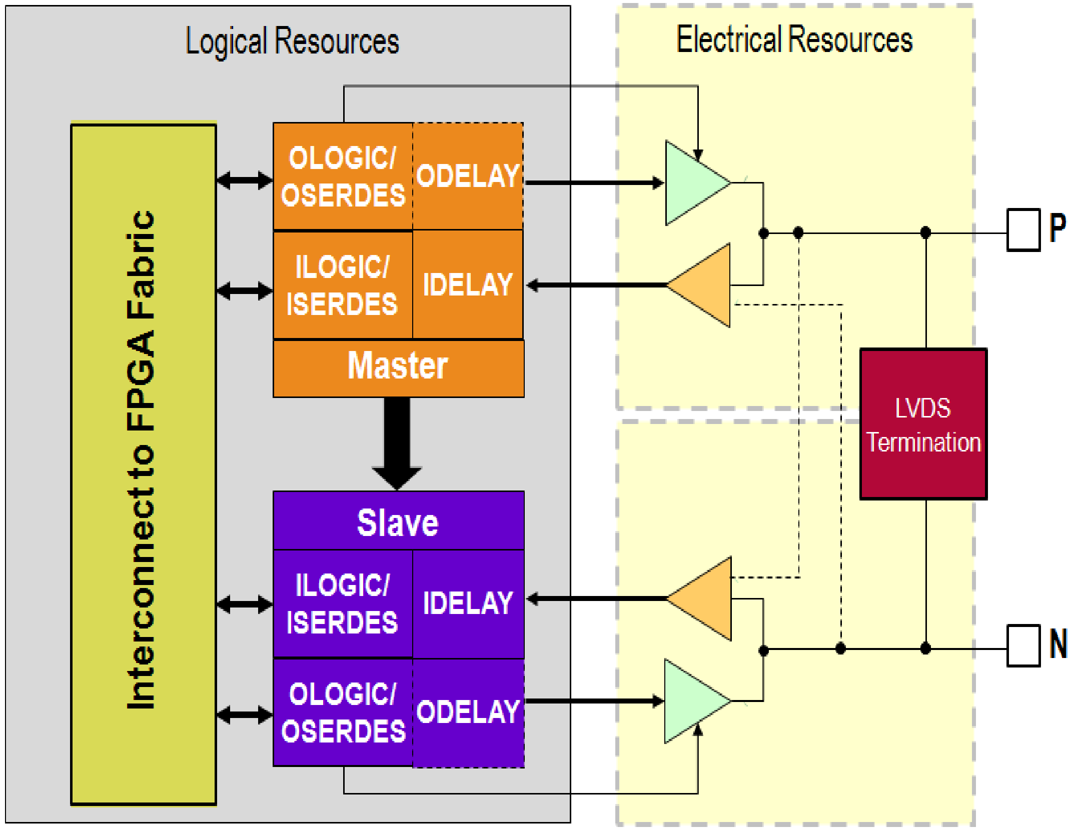
- All flip-flops are D type
  - Q output
- All flip-flops have a single clock input (CK)
- All flip-flops have an active high chip enable (CE)
- All flip-flops have an active high SR input
  - Input can be synchronous or asynchronous
  - Sets the flip-flop value to a pre-determined



UG474_c3_05_102910

7-Series FPGA I/O

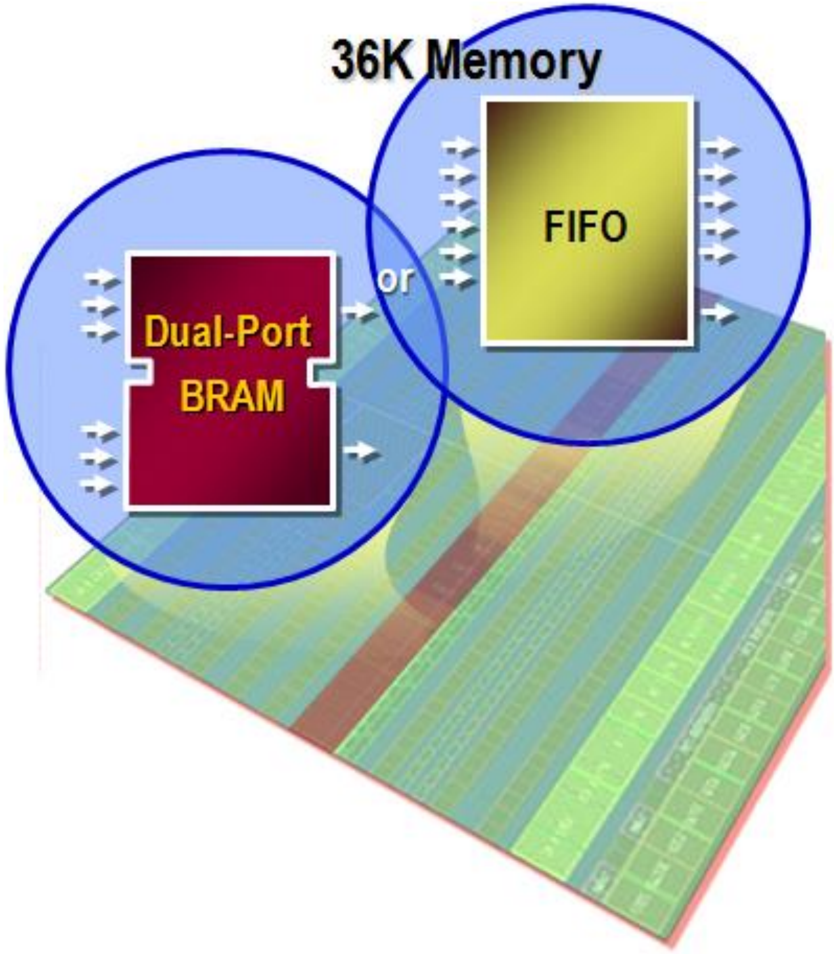
- Wide range of voltages
  - 1.2V to 3.3V operation
- Wide I/O standards support
  - Single ended and differential
  - Referenced voltage inputs
  - 3-state capability
- Very high performance
  - Up to 1600 Mbps LVDS
  - Up to 1866 Mbps single-ended for DDR3
- Easy memory interfacing
  - Hardware support for QDRII+ and DDR3
- Digitally controlled impedance
- Power reduction features



AMD
together we advance_

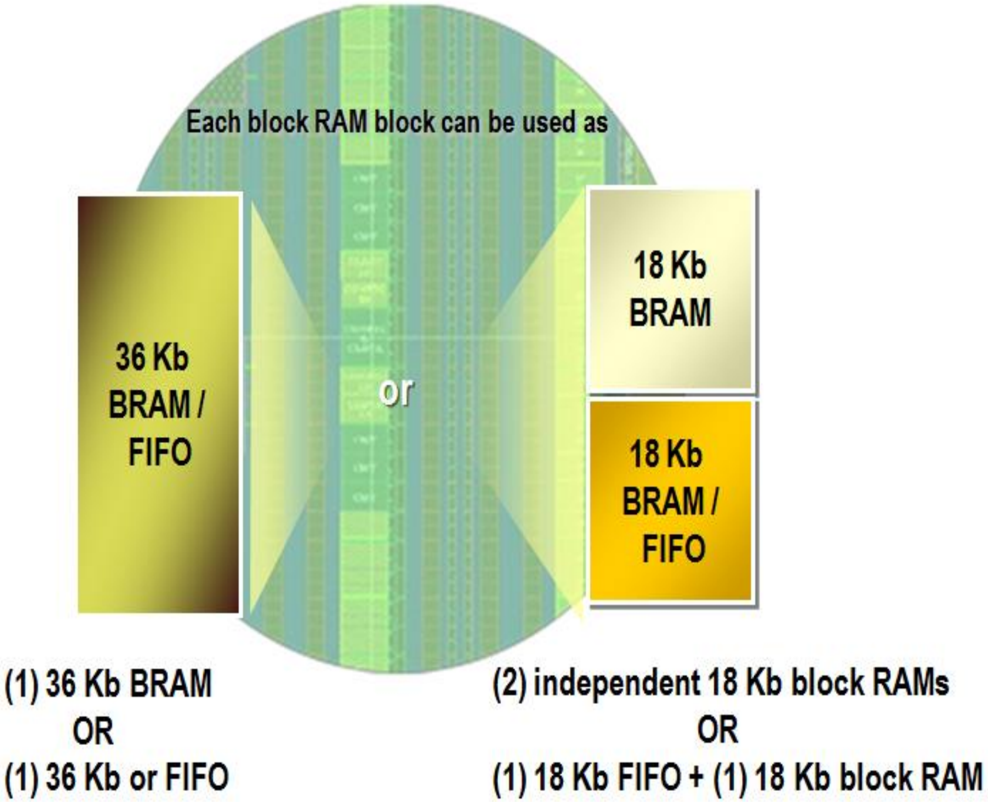
7-Series Block RAM and FIFO

- Fully synchronous operation
  - Outputs are latched
- Optional internal pipeline register
  - Higher frequency operation
- Two independent ports access common data
  - Individual address, clock, write enable, clock enable
  - Independent data widths for each port

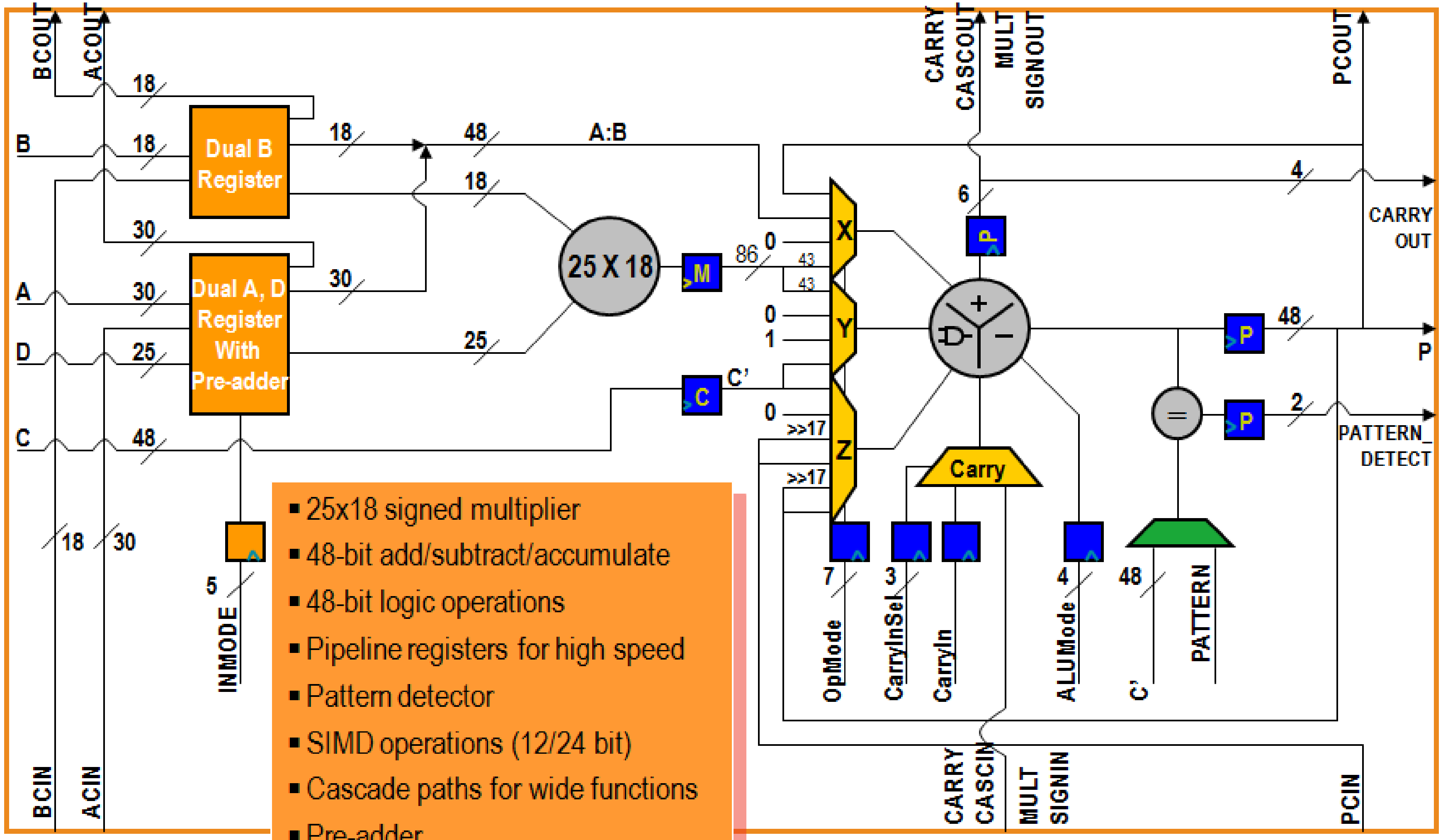


7-Series Block RAM and FIFO

- Multiple configuration options
  - True dual-port, simple dual-port, single-port
- Integrated cascade logic
- Byte-write enable in wider configurations
- Integrated control for fast and efficient FIFOs
- Integrated 64/72-bit Hamming error correction


Each block RAM block can be used as
36 Kb BRAM / FIFO
or
18 Kb BRAM
18 Kb BRAM / FIFO
(1) 36 Kb BRAM
OR
(1) 36 Kb or FIFO
(2) independent 18 Kb block RAMs
OR
(1) 18 Kb FIFO + (1) 18 Kb block RAM

7-Series DSP48E1 Slice

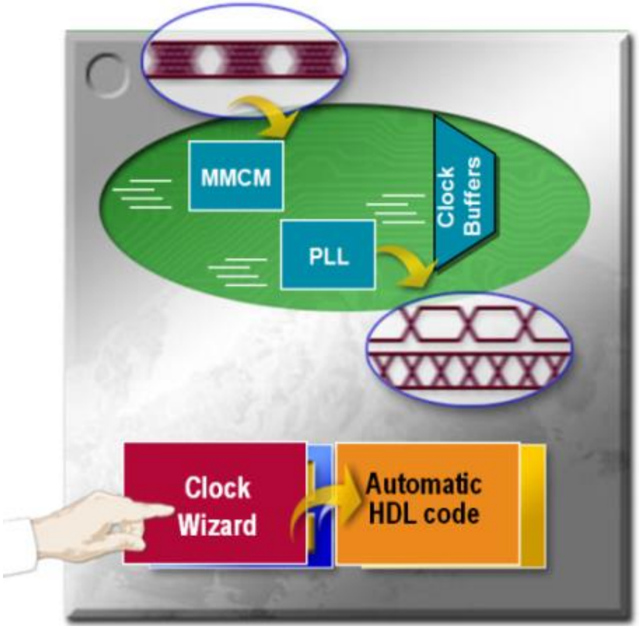


- 25x18 signed multiplier
- 48-bit add/subtract/accumulate
- 48-bit logic operations
- Pipeline registers for high speed
- Pattern detector
- SIMD operations (12/24 bit)
- Cascade paths for wide functions
- Pre-adder

AMD
together we advance_

7-Series FPGAs Clock Management

- Global clock buffers
  - High fanout clock distribution buffer
- Low-skew clock distribution
  - Regional clock routing
- Clock regions
  - Each clock region is 50 CLBs high and spans half the device
- Clock management tile (CMT)
  - One Mixed-Mode Clock Managers (MMCMs) and one Phase Locked Loop (PLL) in each Clock
  - Performs frequency synthesis, clock de-skew, and jitter-filtering
  - High input frequency range



Programming Model

Hardware Description Languages (HDL)

- Verilog
- VHDL
- System Verilog
- Closer to the metal
  - Low level abstraction
  - Describe the behaviour

AMD Vivado

High-Level Synthesis (HLS)

- C/C++
- High level of abstraction
  - Write algorithms
- Vitis HLS generates the architecture
  - Guided by user directives

AMD Vitis

VHLD/Verilog counter

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity counter is
    Port ( clk: in std_logic;
           rst: in std_logic;
           cout: out std_logic_vector(3 downto 0)
     );
end counter;
architecture rtl of counter is
signal counter_up: std_logic_vector(3 downto 0);
begin
    process(clk)
    begin
    if(rising_edge(clk)) then
        if(rst='1') then
            counter_up <= x"0";
        else
            counter_up <= counter_up + x"1";
        end if;
    end if;
    cout <= counter_up;
    end process;
end rtl;
```

Verilog

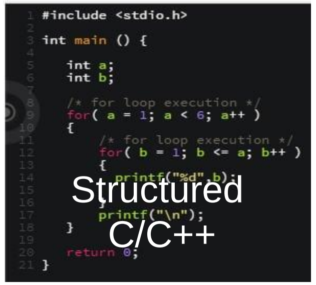
```
module counter(
    input clk,
    input rst,
    output reg [7:0] count
    );

 always @(posedge(clk)) begin
     if (rst)
        count <= 0;
     else
        count <= count + 1;
    end
endmodule
```

AMD
together we advance_

Vitis HLS Vector addition

```
void vadd(const int* in1, // Read-Only Vector 1
          const int* in2, // Read-Only Vector 2
          int* out,       // Output Result
          int elements    // Number of elements
          ) {
// Simple vector addition kernel.
vadd1:
    for (int i = 0; i < elements; i++) {
        out[i] = in1[i] + in2[i];
    }
}
```

What is AMD Vitis™ HLS and HLS Benefits



```
#include <stdio.h>

int main () {

    int a;
    int b;

    /* for loop execution */
    for( a = 1; a < 6; a++ )
    {
        /* for loop execution */
        for( b = 1; b <= a; b++ )
        {
          printf("%d",b);

        }
        printf("\n");
    }

    return 0;
}
```

Structured C/C++

RTL Code

Automated C/C++ to RTL Conversion

Allows Significantly Faster Design Iterations

Significantly Accelerates Simulation – Important For Wireless, Video Applications

AI on FPGA

DNNs and their Potential

1. Requires little domain expertise
2. NNs are a “universal approximation function”
3. If you make it big enough and train it long enough
   - Can outperform humans and existing algorithms on specific tasks

Will not only increasingly replace other algorithms, but also…

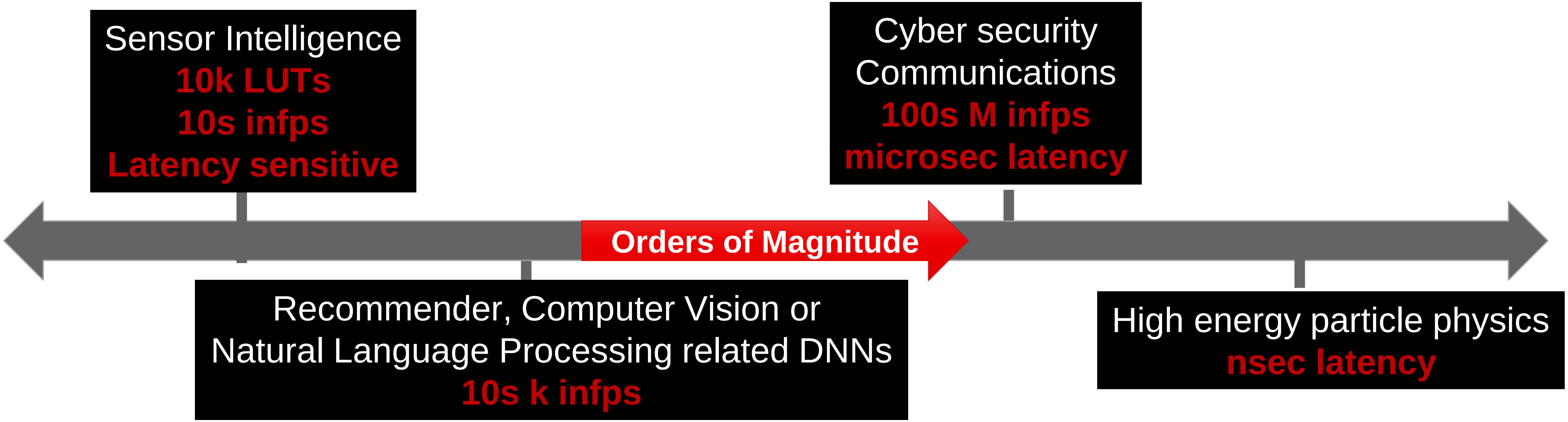
Nature, Oct 2021

… solve previously unsolved problems

- ChatGPT, Copilot
- Stable diffusion
- Protein folding

Stable Diffusion Prompt: "Pencil sketch of an international group of semiconductor research scientists, studio Ghibli"

Spectrum of ML use case with very different requirements



Sensor Intelligence
10k LUTs
10s infps
Latency sensitive

Recommender, Computer Vision or Natural Language Processing related DNNs
10s k infps

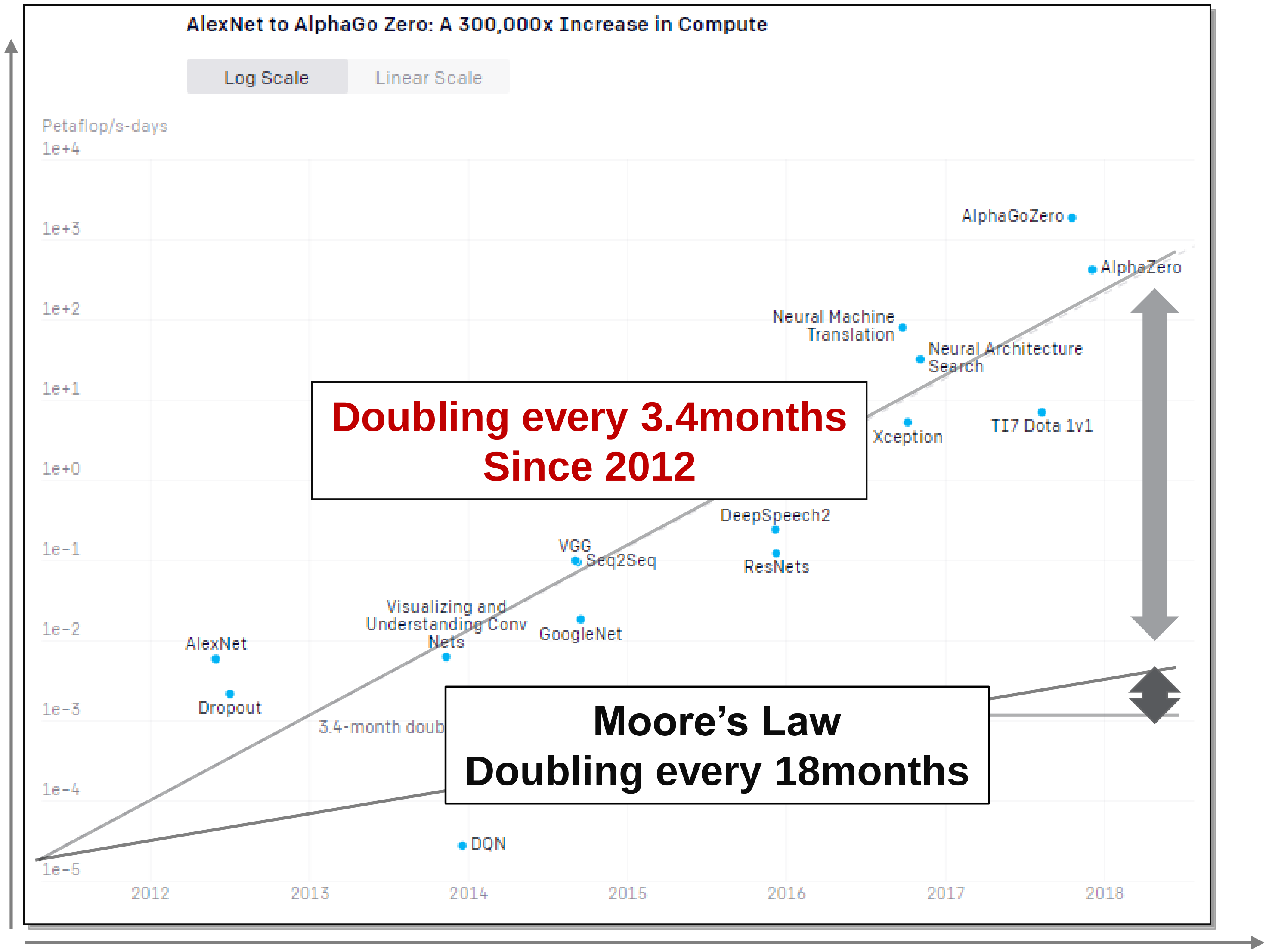
Orders of Magnitude

Cyber security
Communications
100s M infps
microsec latency

High energy particle physics
nsec latency

infps = inferences per second

DNN Compute Requirements are Outpacing Moore’s Law



Doubling every 3.4months Since 2012

Moore’s Law Doubling every 18months

Compute Requirements Petaflops/day (log scale)

Time

Source: https://blog.openai.com/ai-and-compute

Innovation is needed to provide the necessary performance scalability

Specialization Is #1 Industry Approach to Achieve Performance Scalability and Energy Efficiency

GRAPHCORE

KNUPATH

habana

WAVE COMPUTING

Movidius

Specialized Architectures
DPU
(Deep Learning Processors)

Wafer Scale Computing

Quantum Computing

In-memory computing

GPUs

3D Stacking

Analog computing

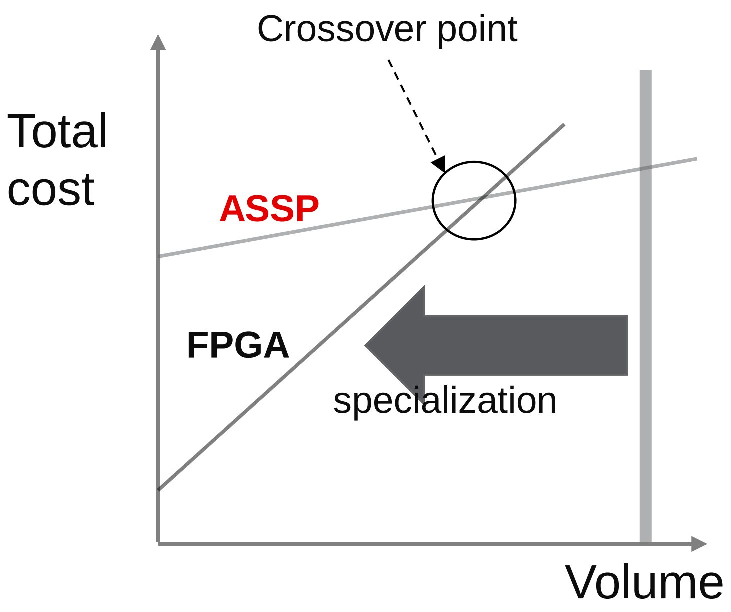
nervana

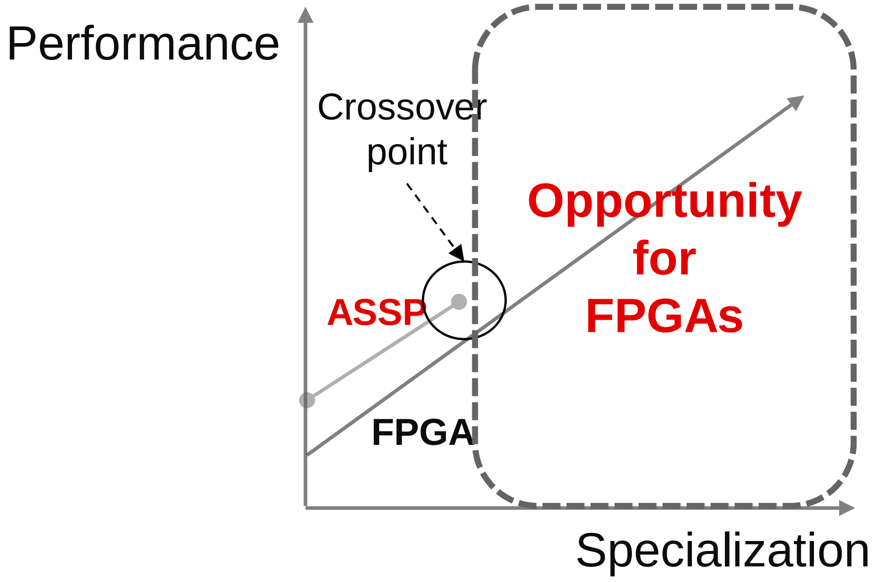
Alibaba.com

aws

AMD
together we advance_

Adaptive Computing or Dedicated Silicon for DPUs





- Programmable interconnect
- Programmable precision
- Programmable operations
- …

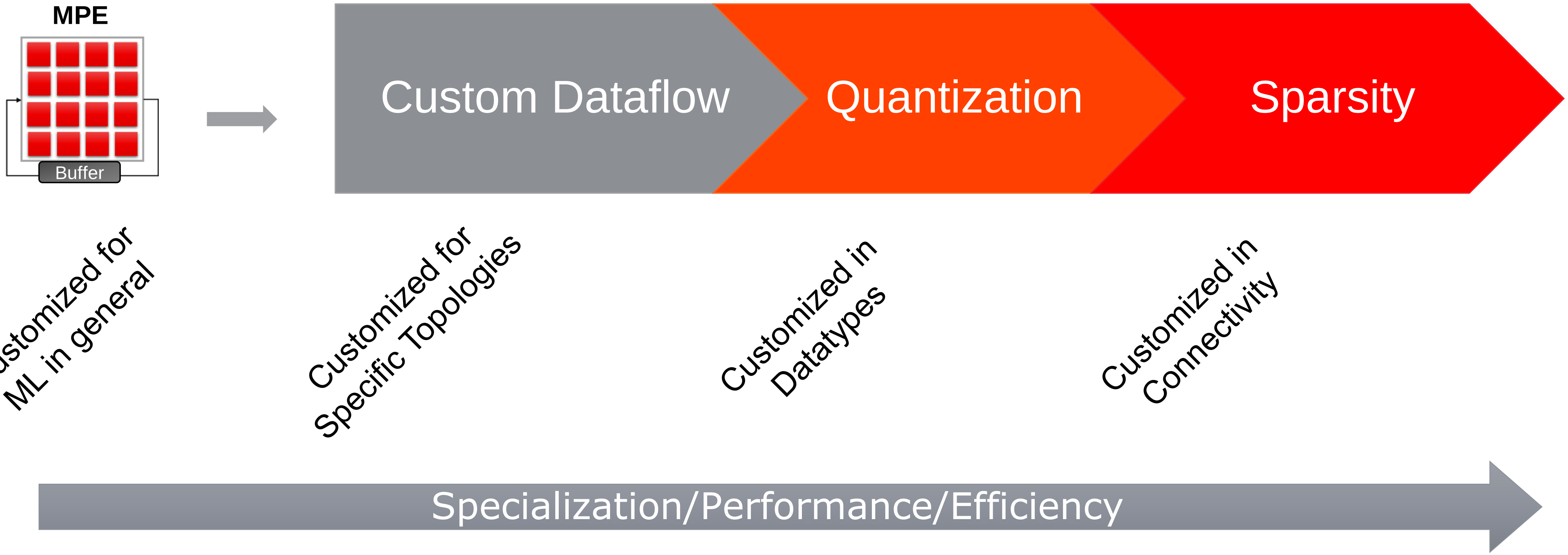
- With increasing specialization of the device, potential sales volume decreases
  - Hard to amortize the increasing NRE costs involved in building ASSPs
  - FPGAs become more attractive
- Increasing specialization scales performance for both ASSPs and FPGAs

- **The opportunity for FPGAs lies in their ability to specialize**

ASSP: Application Specific Standard Product

Vitis AI - ML in general

Customization levels on Adaptive Computing



Custom Dataflow

Quantization

Sparsity

Customized for ML in general

Customized for Specific Topologies

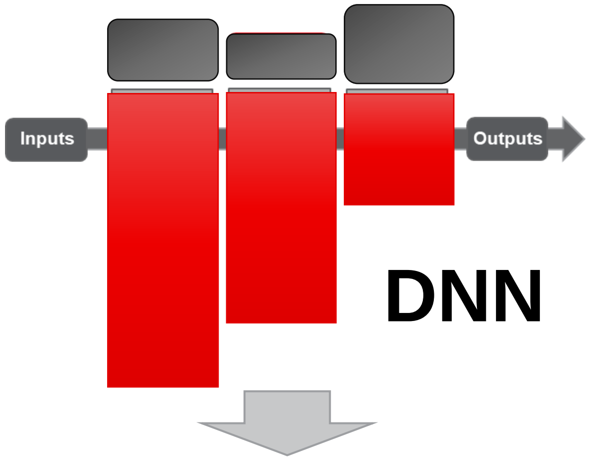
Customized in Datatypes

Customized in Connectivity

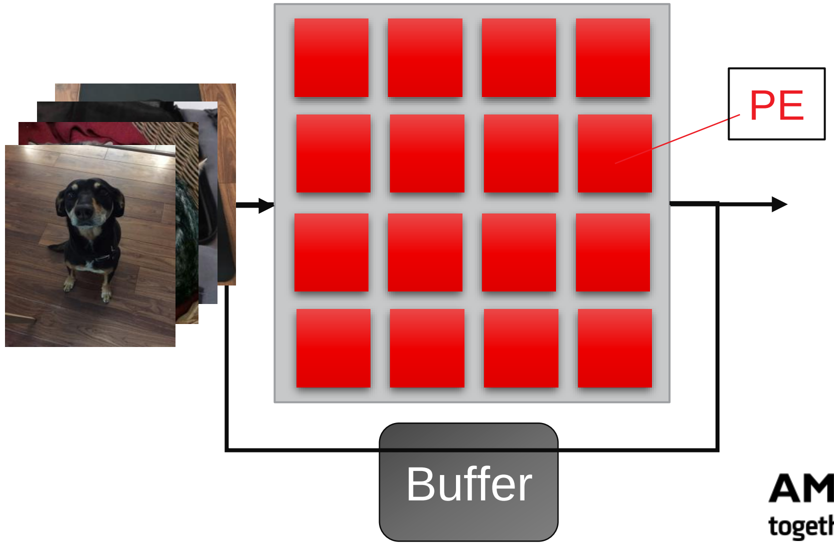
Specialization/Performance/Efficiency

Popular Approach: Matrix of Processing Engines (MPEs) Specializing for AI in general

- Popular layer-by-layer compute
- Batching to achieve high compute efficiency
  - At latency cost (latency ~ batch size)
- Specialized processing engines
  - Operators
  - ALU types
    - tensor-, matrix- or vector-based



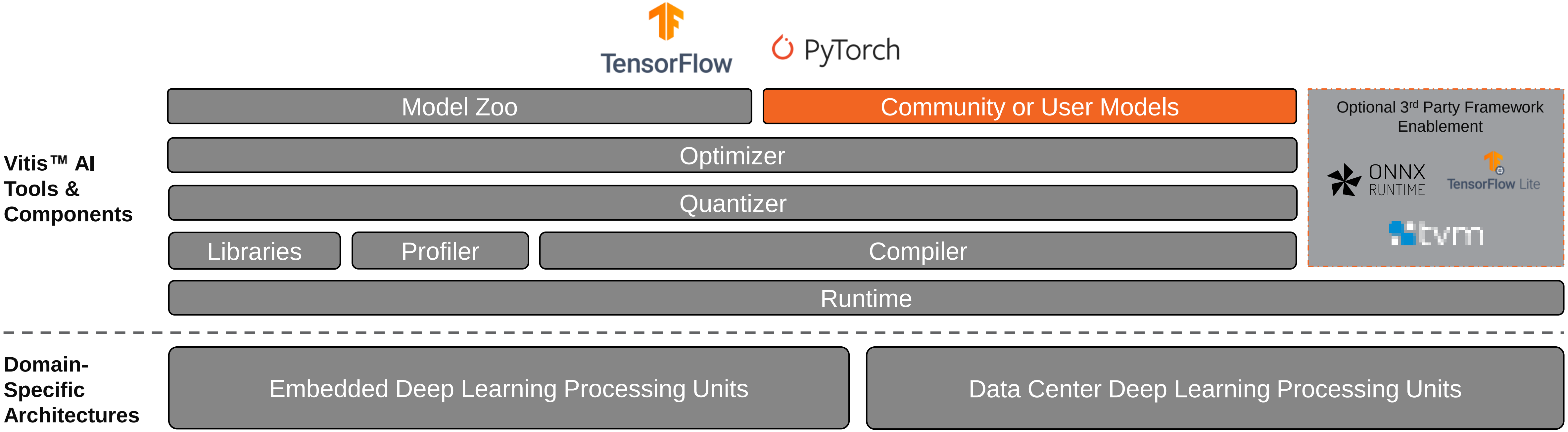
Matrix of Processing Engines (MPE)
“layer-by-layer” compute



- **Customized for ML in general**
  - **Designed to run any DNN**
- **Works really well for computer vision and natural language processing (10s kinfps)**
- **Popular approach: Vitis AI (FPGA or AIE) as well as majority of AI accelerators**

AMD Vitis™ AI Integrated Development Environment

A Complete AI Stack for Adaptable AMD Targets


TensorFlow
PyTorch
Vitis™ AI Tools & Components
Model Zoo
Community or User Models
Optional 3rd Party Framework Enablement
ONNX RUNTIME
TensorFlow Lite
tvm
Optimizer
Quantizer
Libraries
Profiler
Compiler
Runtime
Domain-Specific Architectures
Embedded Deep Learning Processing Units
Data Center Deep Learning Processing Units

Supported AMD Targets

VCK190

ZCU102

ZCU104

Kria™ K26 SOM

U50/C/LV

U200 / U250

U280

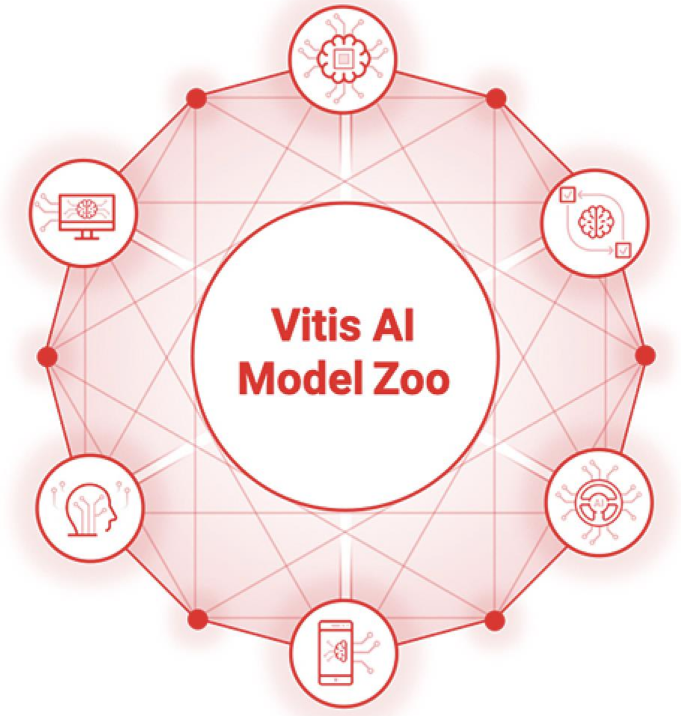
VCK5000

Your Platform

AI Model Zoo – Expanding to Diverse AI Applications

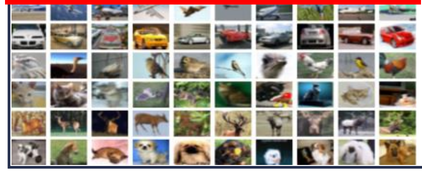
- A comprehensive AI model repository
  - Open and free to download for any user
  - State-of-the-art models from Pytorch, TF & TF2
  - Retrainable, appliable to various data set & scenario
  - Deployable on AMD FPGA and Versal Adaptive SoC
- New models in each release

PyTorch TensorFlow TensorFlow 2.0



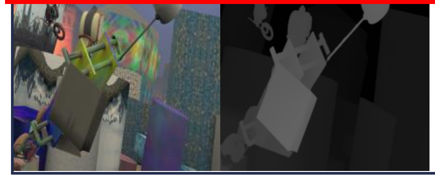
Extensive Application Coverage

Classification



- Inception
- Mobilenet
- Resnet
- VGG
- EfficientNet
- MLPerf ResNet50
- OFA ResNet
- Vision Transformer
- Car Type classification
- Car Color classification

IndustrialVision/Robotics

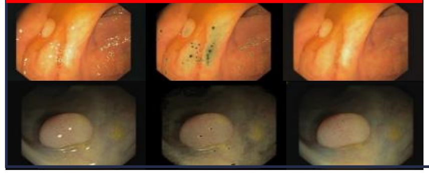


- FADNet
- PSMNet
- PMG
- Superpoint
- HFNet

Detection

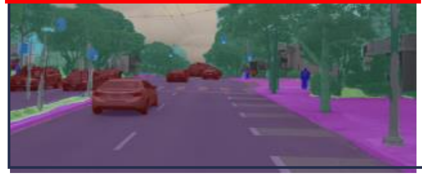
- ssd_mobilenet
- Yolov3
- Yolov4
- YoloX
- Refinedet
- EfficientDet
- Pointpillars
- Centerpoint
- CLOCs
- Pointpainting
- Multi-taskv3
- OFA-Yolo

Medical Image



- RCAN
- SESR
- OFA-RCAN
- DRUnet
- SSR
- C2D2lite

Segmentation



- ENet
- Semantic FPN
- Salsanext
- Salsanextv2
- SOLO
- Mobilenetv2
- 2D-Unet
- FPN-ResNet18
- Unet-Chaos-CT
- HardNet
- Sa-Gate

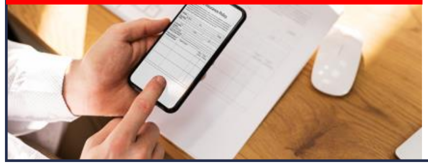
NLP

- Bert-base
- Sentiment detection
- Customer satisfaction
- Open-information-extraction

Video Analytics

- Face Recognition
- Face Quality
- Face ReID
- Person ReID
- FairMOT
- FaceMask Detection
- MoveNet

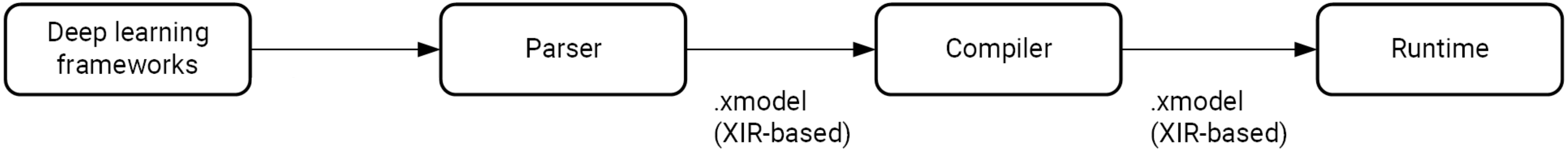
Text-OCR



- Textmountain, OCR

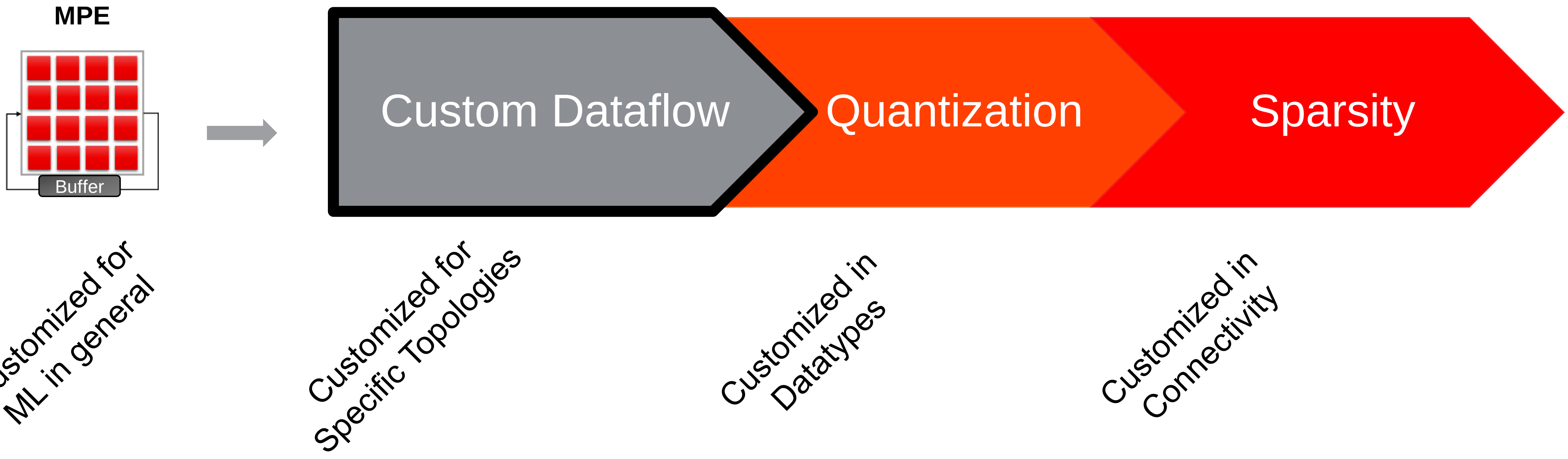
Compiling for DPU - an XIR-based Toolchain

- Xilinx Intermediate Representation (XIR)
  - Graph-based intermediate representation of the AI algorithms
  - Designed for compilation and efficient deployment of the DPU on the FPGA platform.
- XIR-based compilation flow
  - First, transform the input models to XIR format
  - Breaks up computing graph to subgraphs
  - Execute DPU subgraph to a compiled xmodel file



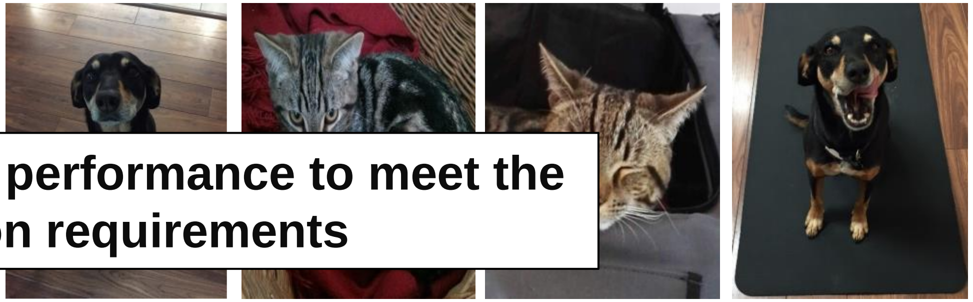
Techniques for Further Specialization with **Adaptive Compute Architectures**

Specialization beyond MPEs

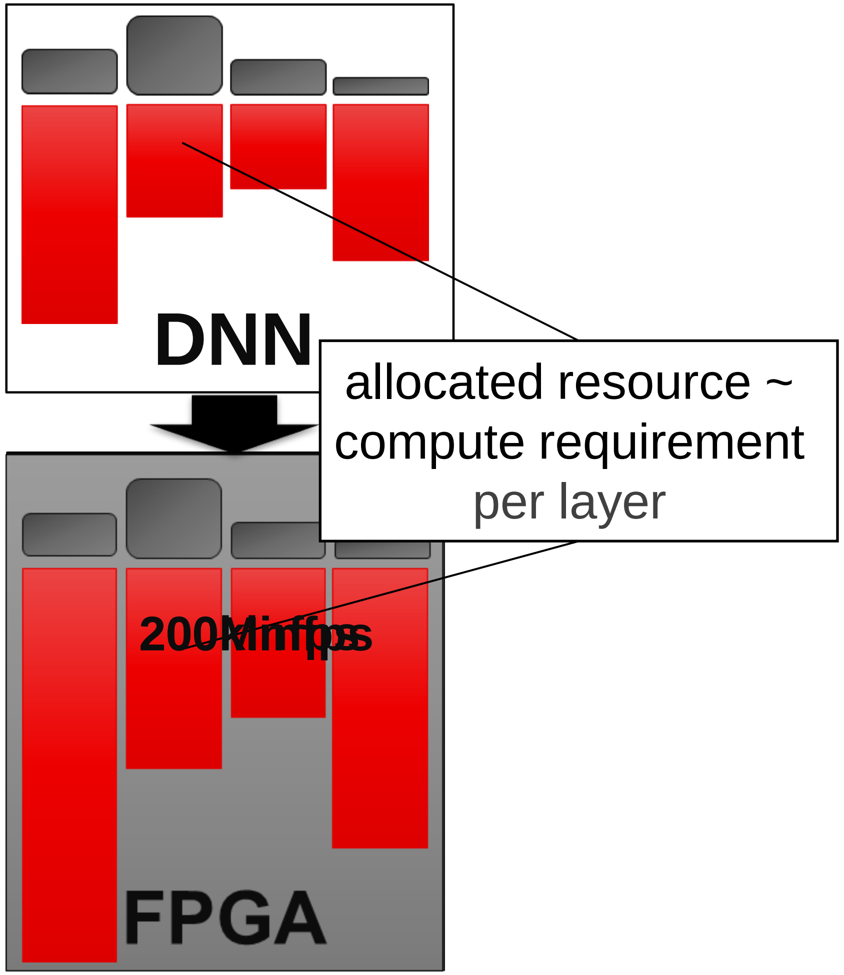

MPE
Buffer
Custom Dataflow
Quantization
Sparsity
Customized for ML in general
Customized for Specific Topologies
Customized in Datatypes
Customized in Connectivity

Dataflow - Specializing for Individual Topologies

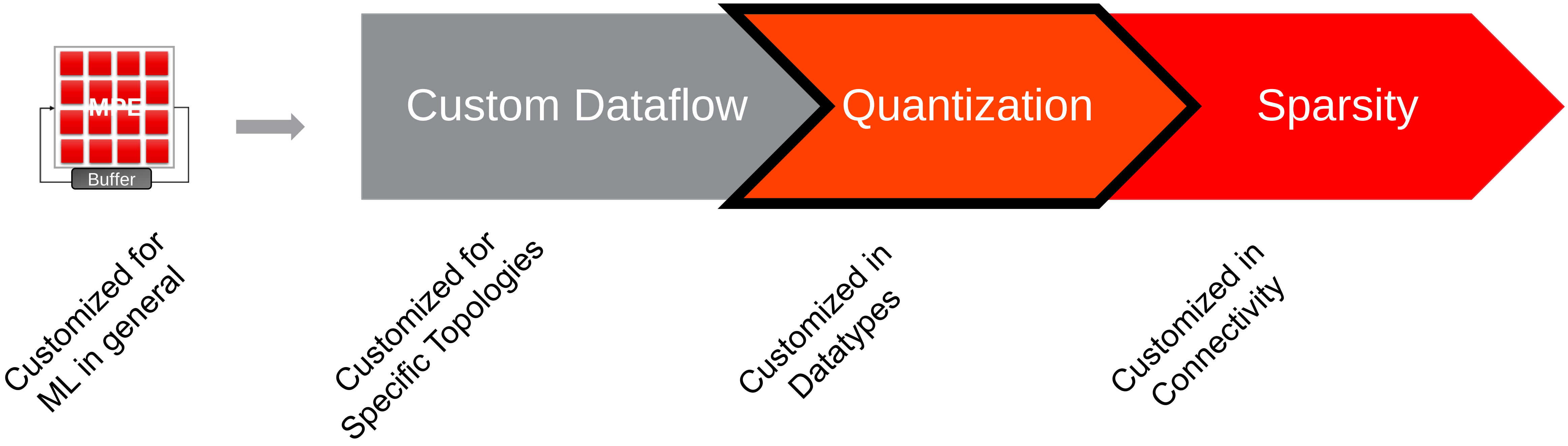
- Hardware instantiates the topology as a dataflow architecture
  - Customize everything to the **specifics of the given DNN**, any operation, any connectivity
- Benefits:
  - Improved efficiency
  - Low fixed latency
- Scale performance & resources to meet the application requirements
  - If resources allow, we can completely unfold to create a circuit that inferences at clock speed and thereby meet these new throughput requirements



Dataflow can scale performance to meet the application requirements

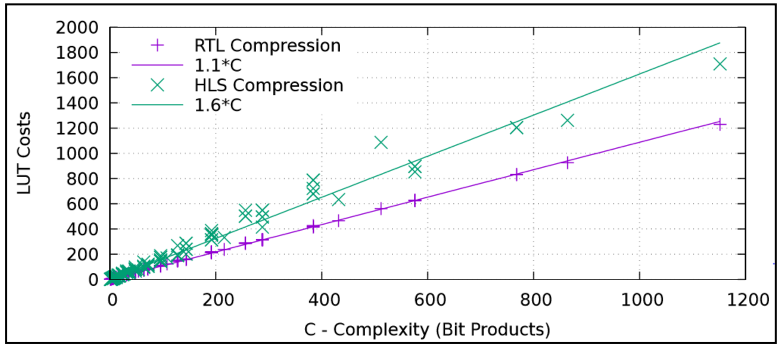


Specialization beyond MPEs


MPE
Buffer
Custom Dataflow
Quantization
Sparsity
Customized for ML in general
Customized for Specific Topologies
Customized in Datatypes
Customized in Connectivity

Customizing Arithmetic to Minimum Precision

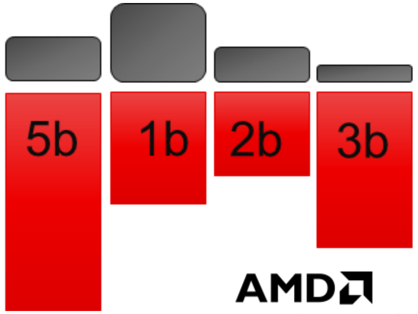
- Popular approach which reduces bits in the data representation of weights and activations while preserving accuracy
- Reducing precision shrinks hardware cost/ scales performance
  - Instantiate n-times more compute within the same fabric, thereby scale performance n-times
- Reduces memory footprint
  - NN model can stay on-chip => no memory bottlenecks
- With dataflow: every layer has dedicated compute resources, we can mix and match precision across layers
  - Exploit custom arithmetic at a greater degree than MPEs



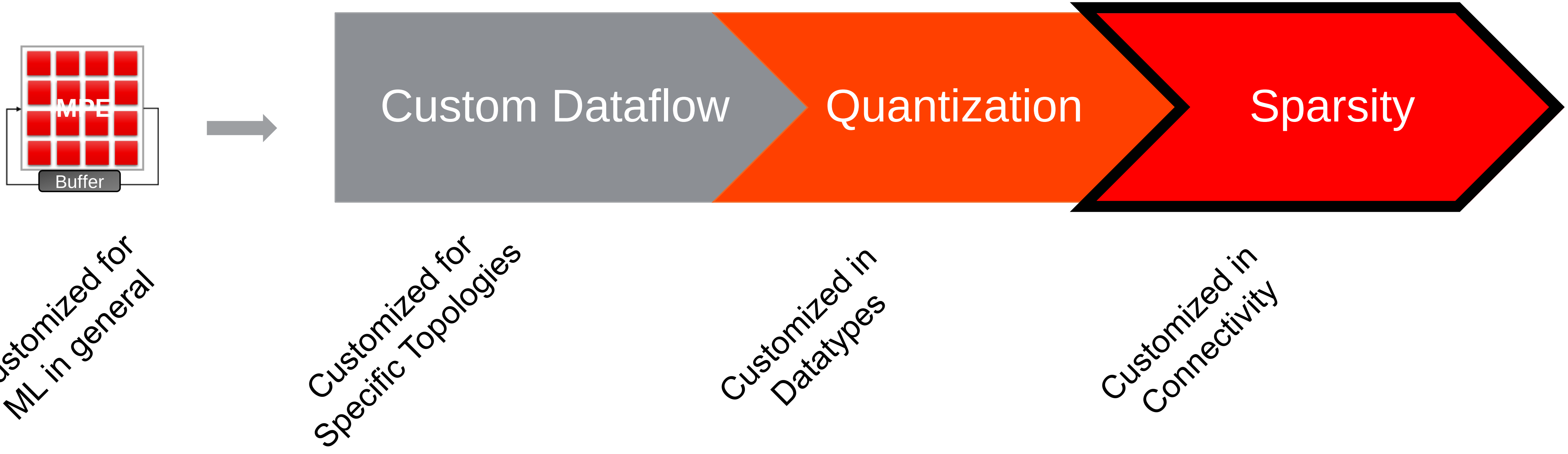
C= f(size of accumulator, size of weight,size of activation)

Precision	Model size [MB] (ResNet50)
1b	3.2
8b	25.5
32b	102.5

Reducing precision saves resources/ scales performance, and reduces memory
However, it requires quantization support in the training software



Specialization beyond MPEs



Custom Dataflow

Quantization

Sparsity

Customized for ML in general

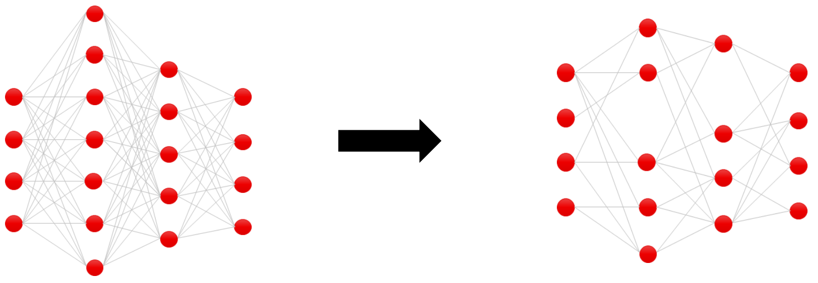
Customized for Specific Topologies

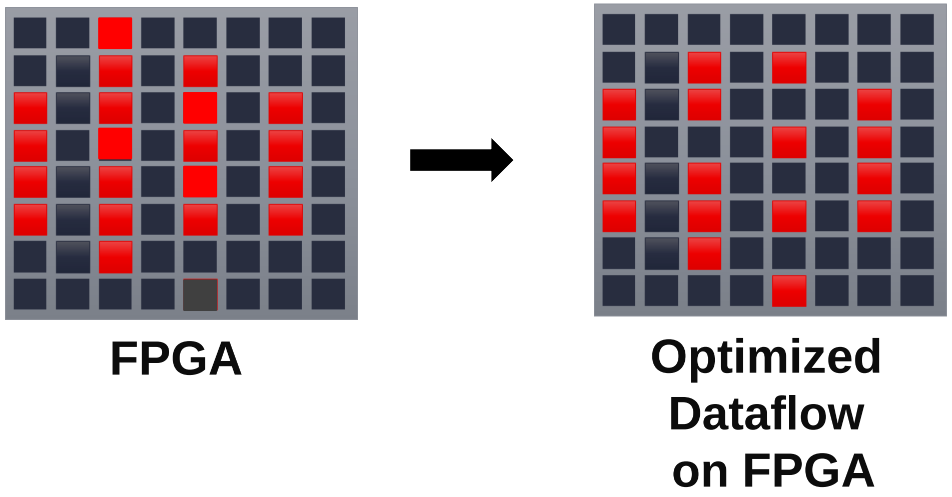
Customized in Datatypes

Customized in Connectivity

Sparsity

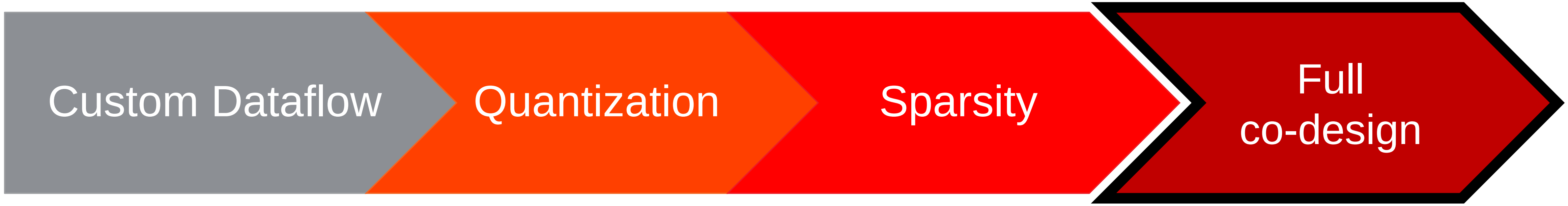
- DNNs are naturally sparse
- Sparse topologies result in irregular compute patterns which are difficult to accelerate on vector- or matrix-based execution units
- With streaming dataflow architectures, where every neuron and synapse is represented in the hardware, we can fully exploit this





Taking it to the Extreme: **LogicNets**

Specialization beyond MPEs



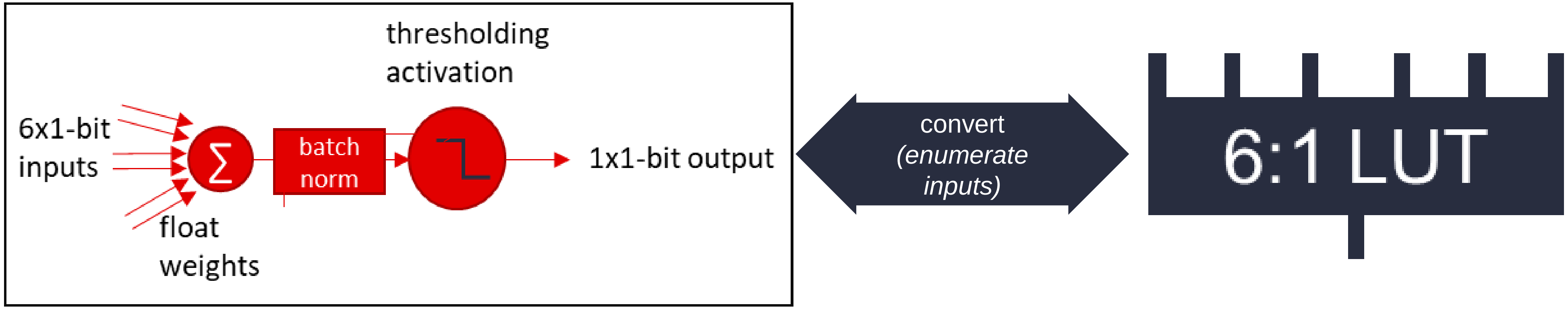
Customized for Specific Topologies

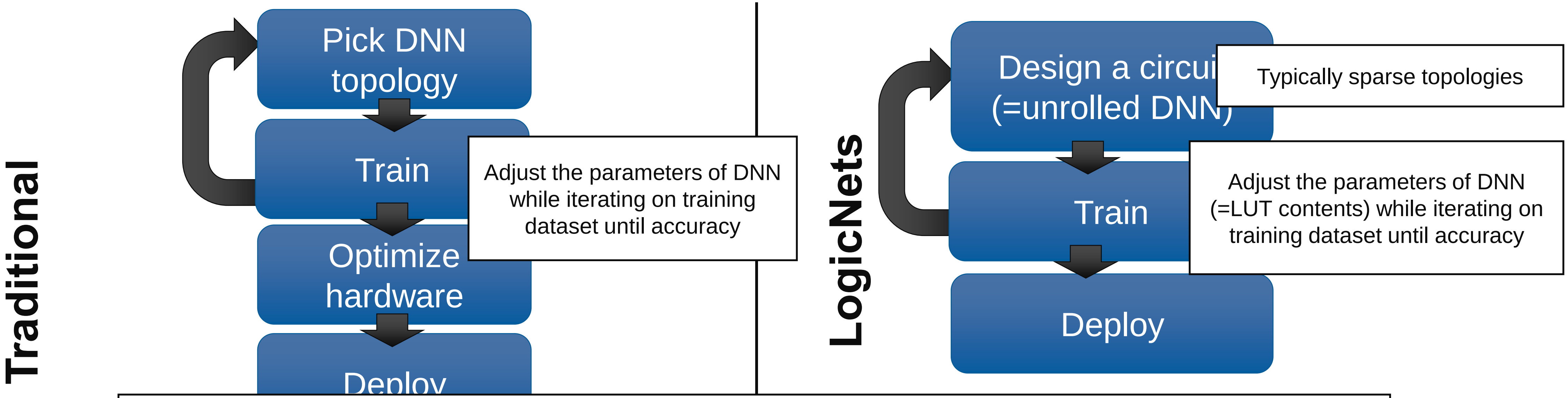
Customized in datatypes

Customized in connectivity

Specialize the DNN to suit hardware

LogicNets with Adaptive Computing





Adjust the parameters of DNN while iterating on training dataset until accuracy

Typically sparse topologies

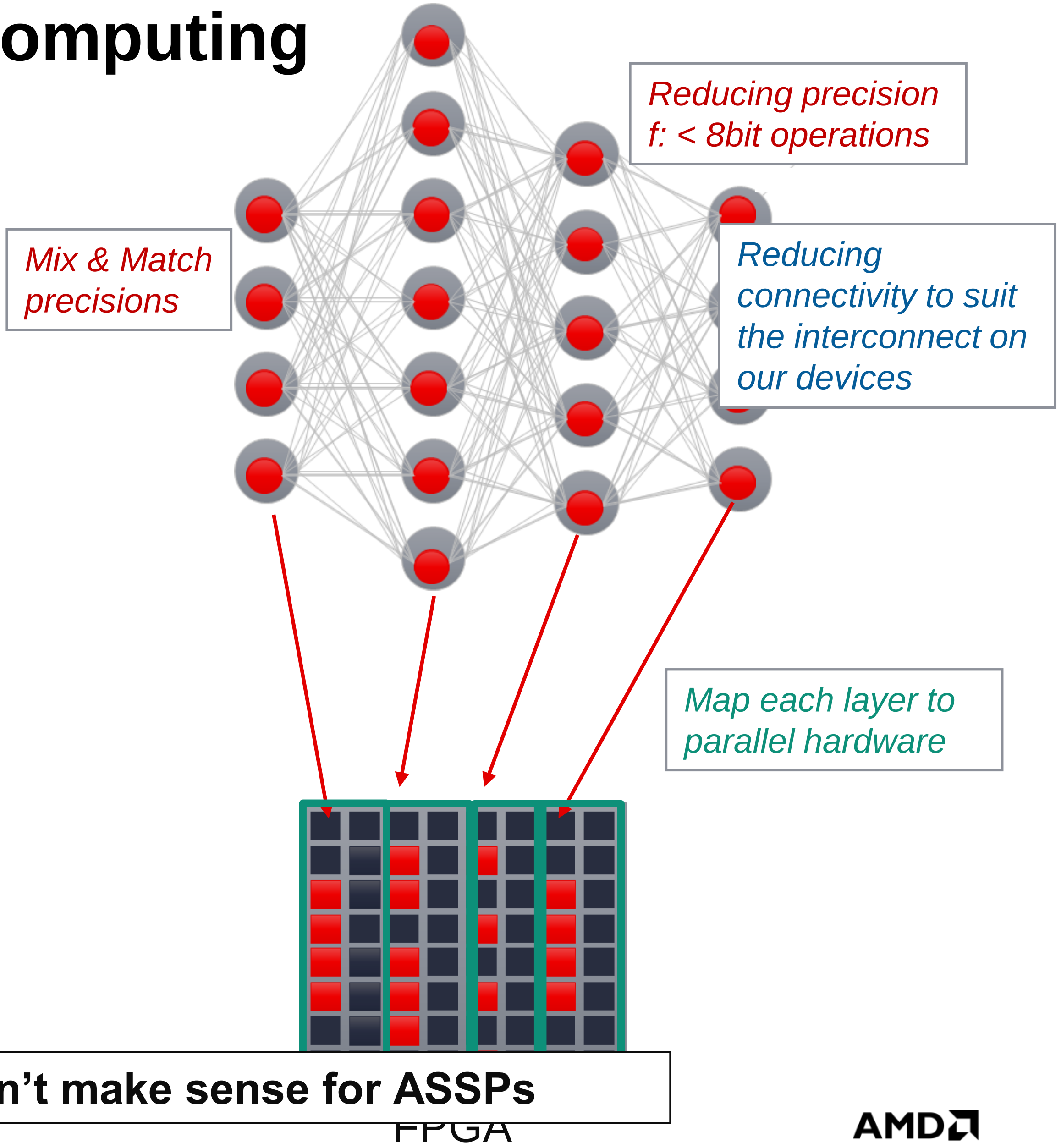
Adjust the parameters of DNN (=LUT contents) while iterating on training dataset until accuracy

Maximum performance by design (classification at clock rate) [5]

Compared to unrolled DF: sparse to suit the interconnect

Unique Opportunity for Adaptive Computing

- FPGAs can scale DNN performance through extreme specialization
- Reduced precision arithmetic
  - Arbitrary bitwidth
  - Mix & match bitwidths between layers
- Fine-grained sparsity
- Scalable, layer-parallel streaming dataflow


Reducing precision f: < 8bit operations

Mix & Match precisions

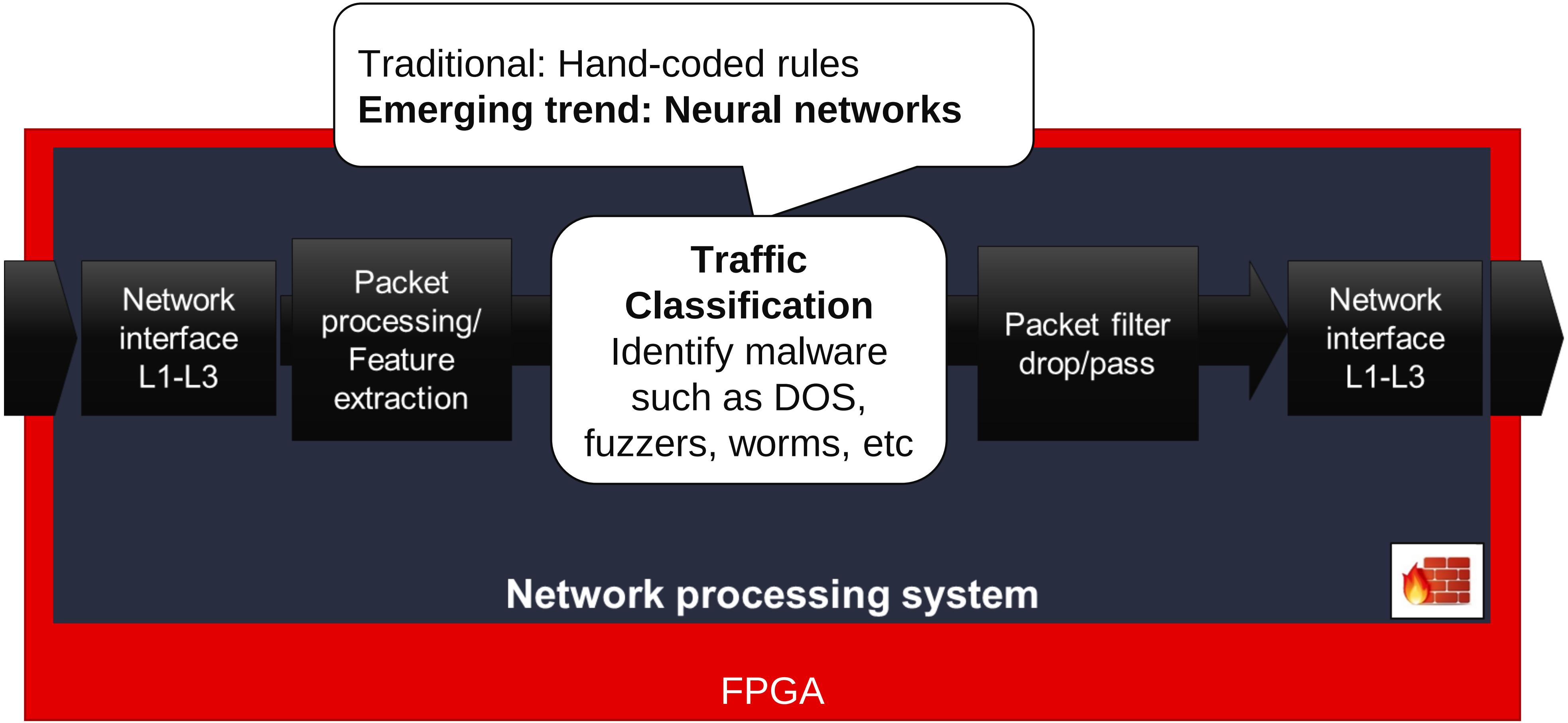
Reducing connectivity to suit the interconnect on our devices

Map each layer to parallel hardware

High degree of specialization doesn't make sense for ASSPs

How much do we get out of the different specializations?

Deep Network Intrusion Detection System (NIDS)



Traditional: Hand-coded rules
Emerging trend: Neural networks

Network interface L1-L3
Packet processing/ Feature extraction
Traffic Classification
Identify malware such as DOS, fuzzers, worms, etc
Packet filter drop/pass
Network interface L1-L3
Network processing system
FPGA

Goal: Implement **NN-based traffic classifier** delivering 100G **line-rate** throughput = 150 Mips
Latency sensitive (buffer 10s of MB/msec)

Dataset: UNSW-NB15 Moustafa, Nour, and Jill Slay. "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)." 2015 military communications and information systems conference (MilCIS). IEEE, 2015.

Results – Implementations

Specialization

	Generic ML architecture	Specialized for topology 2 different folding factors	Fully Co-designed
	MPE (VitisAI)	**FINN (fold 8, fold 1)**	**LogicNet**
Topology / #layers / #OPs	MLP / 3 / 92KOPs	MLP / 3 / 92KOPs	**Circuit** / 4 / 15.4KOPs
#inputs / neuron	64	64	**7**
#bits / weight & activation	8b & 8b	2b & 2b	2b & 32b
Accuracy	92.3%	91.9%	91.3%

Same network

Reduced precision

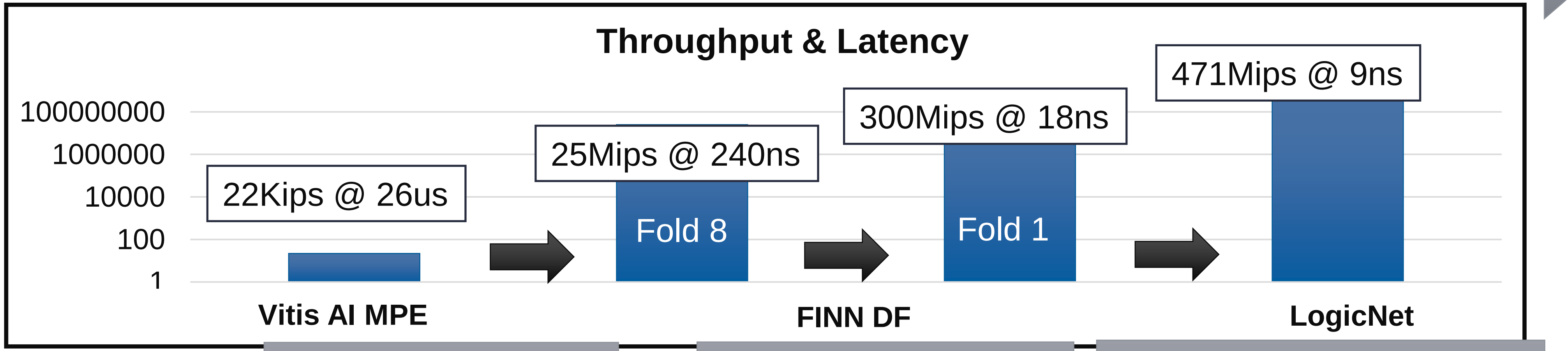
Sparsity

New sparse topology with 7 Inputs/LUT 6x less OPs

4 implementations with varying degree of specialization

Results – Throughput and Latency

Performance



Performance scaling And latency reduction through specialization ~1000x

Further unrolling ~8x

DF unrolling reduces latency further

Full co-design ~ 2x sparse connectivity suits the interconnect

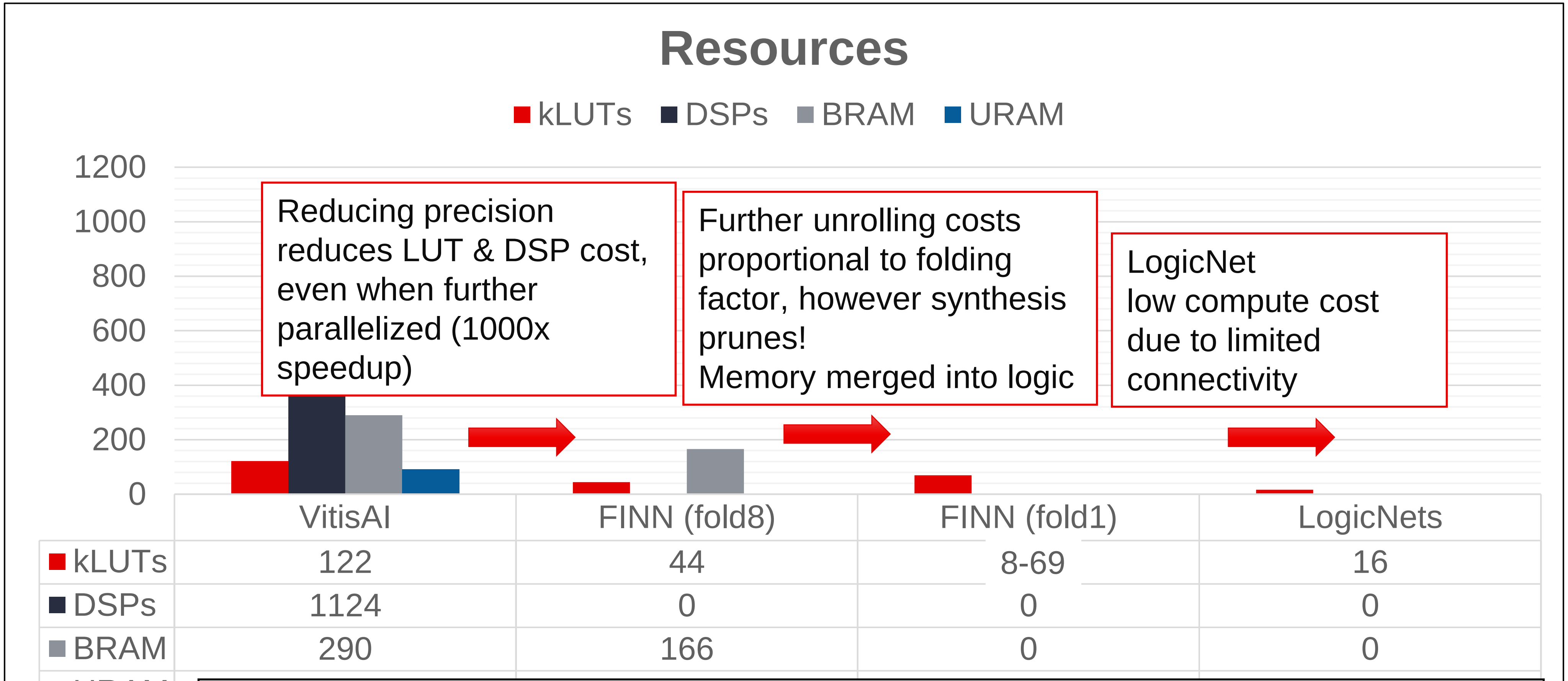
Sparse topology reduces #pipeline stages => latency

Classification ~ clock rate

Specialization scales performance and reduces latency by orders of magnitude if application is amenable

Resource Cost - Compute, Memory

Efficiency



Reducing precision reduces LUT & DSP cost, even when further parallelized (1000x speedup)

Further unrolling costs proportional to folding factor, however synthesis prunes!
Memory merged into logic

LogicNet
low compute cost due to limited connectivity

	VitisAI	FINN (fold8)	FINN (fold1)	LogicNets
🟥 kLUTs	122	44	8-69	16
■ DSPs	1124	0	0	0
▨ BRAM	290	166	0	0
🟦 URAM				

Customizing arithmetic, sparse implementations and learned circuits greatly reduce resources and improve device efficiency

Deep Network Intrusion Detection System (NIDS) Results

- This example illustrates the trade-offs between specialization and performance and efficiency

- Custom arithmetic is effective to **scale performance** and **dataflow to reduce latency**
  - If application is amenable, custom arithmetic can meet extreme throughput requirements such as in NIDS

- Reduced precision, fine-granular sparsity & learned circuits can **shrink the resource** requirements despite speedup

- These are some of the opportunities which make most sense to exploit with FPGAs

General Introduction to FINN

Project Mission and Key Techniques

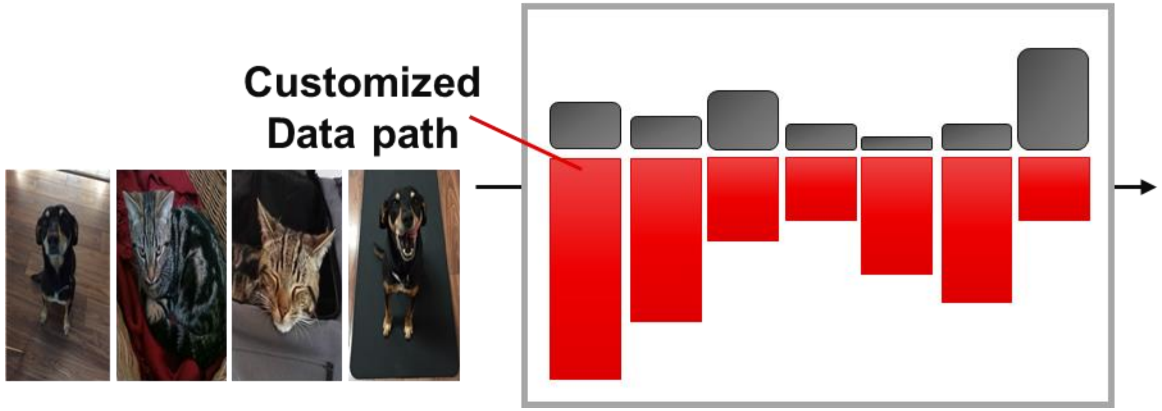
FINN – Project Mission

FINN

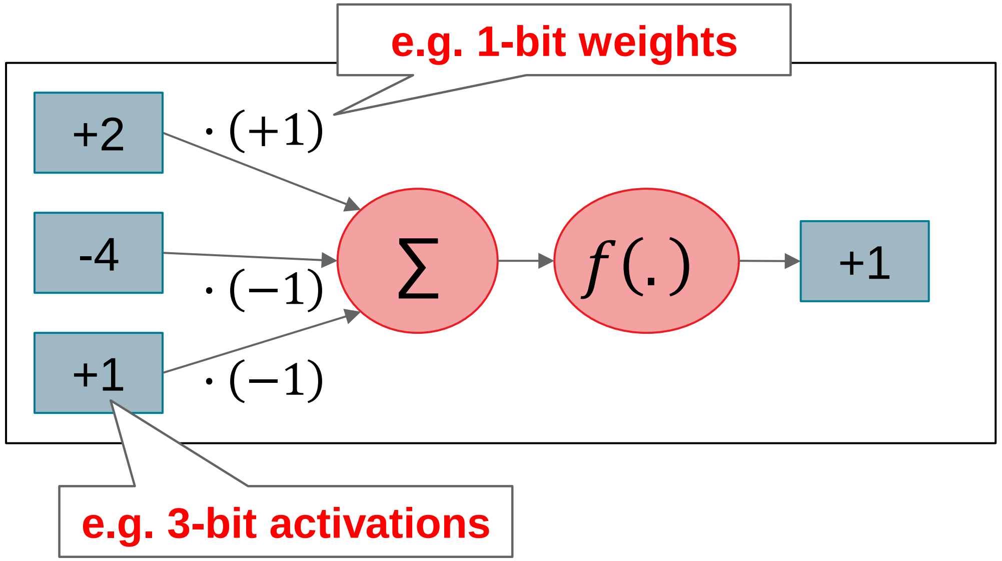
- Custom Specialization
  - for creating **high-throughput, ultra-low-latency** DNN inference engines
- End-to-End
  - flow for the ***easy*** creation of **specialized hardware architectures** for FPGAs
- Open Source
  - for full **transparency and flexibility** to adapt to end user applications and
  - for easy customer interactions

Two Key Techniques for Customization in FINN

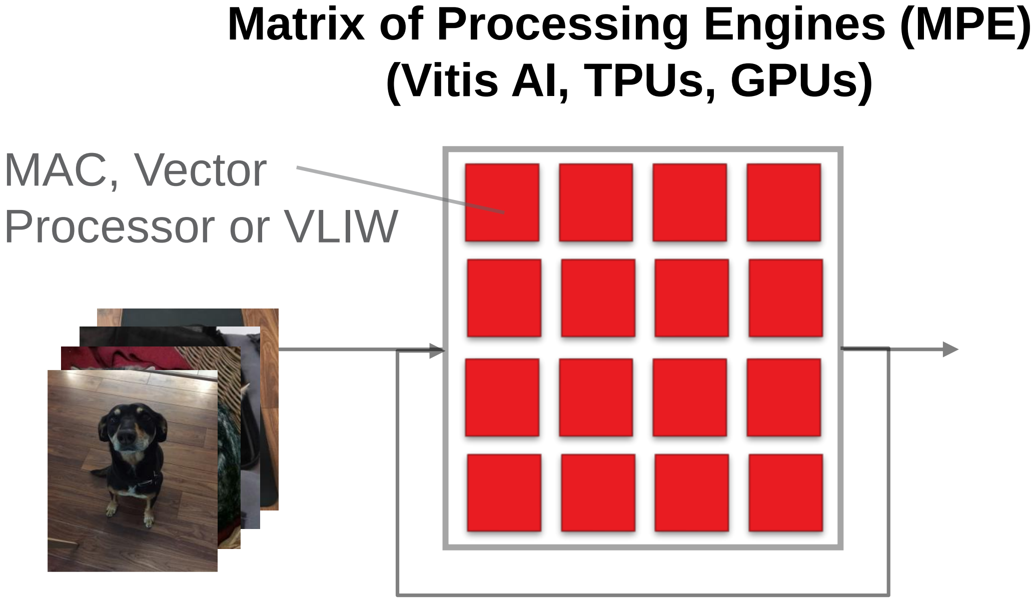
Streaming Dataflow Architectures for FPGAs

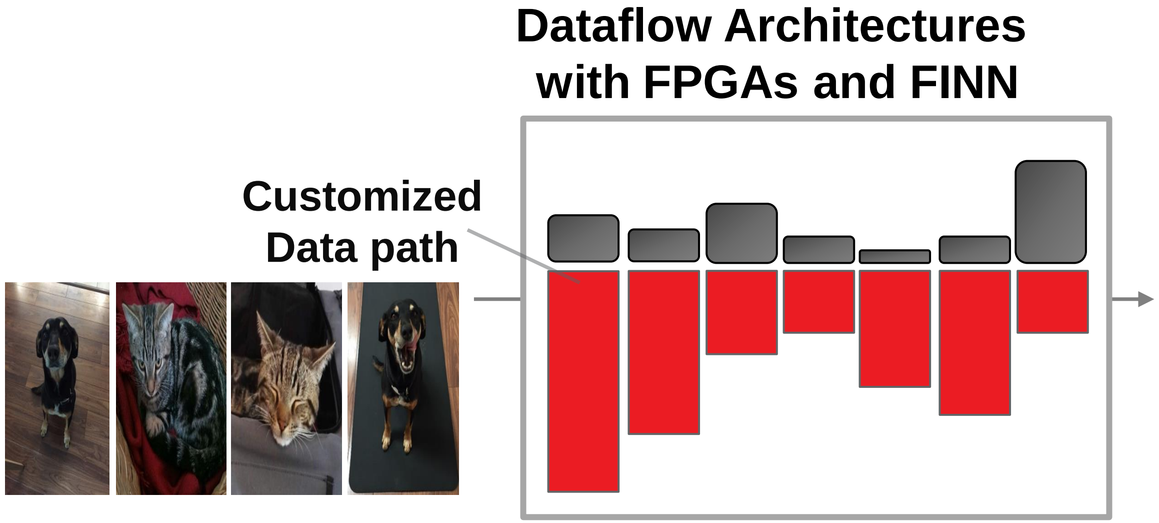


Custom Precision: Few-bit Weights and Activations



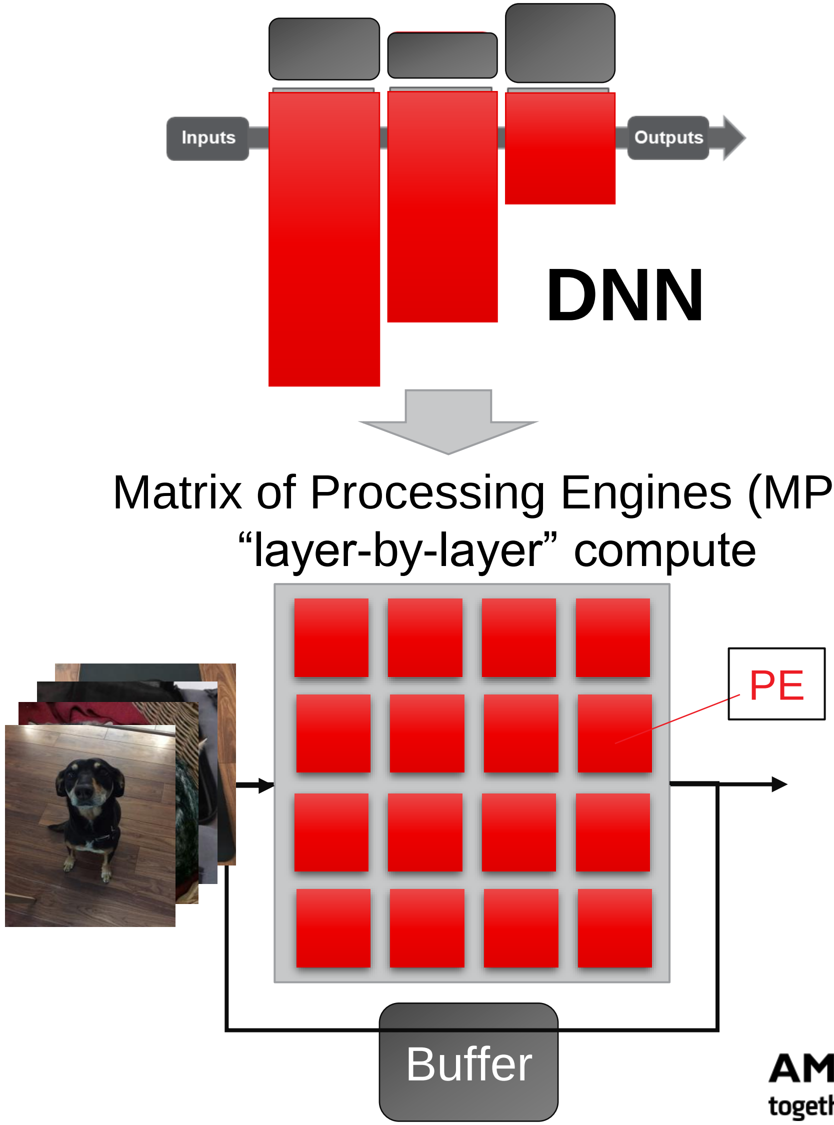
Customized Dataflow Processing versus More Generic Architectures


Matrix of Processing Engines (MPE)
(Vitis AI, TPUs, GPUs)
MAC, Vector Processor or VLIW



Matrix of Processing Engines (MPEs) Specializing for AI in General

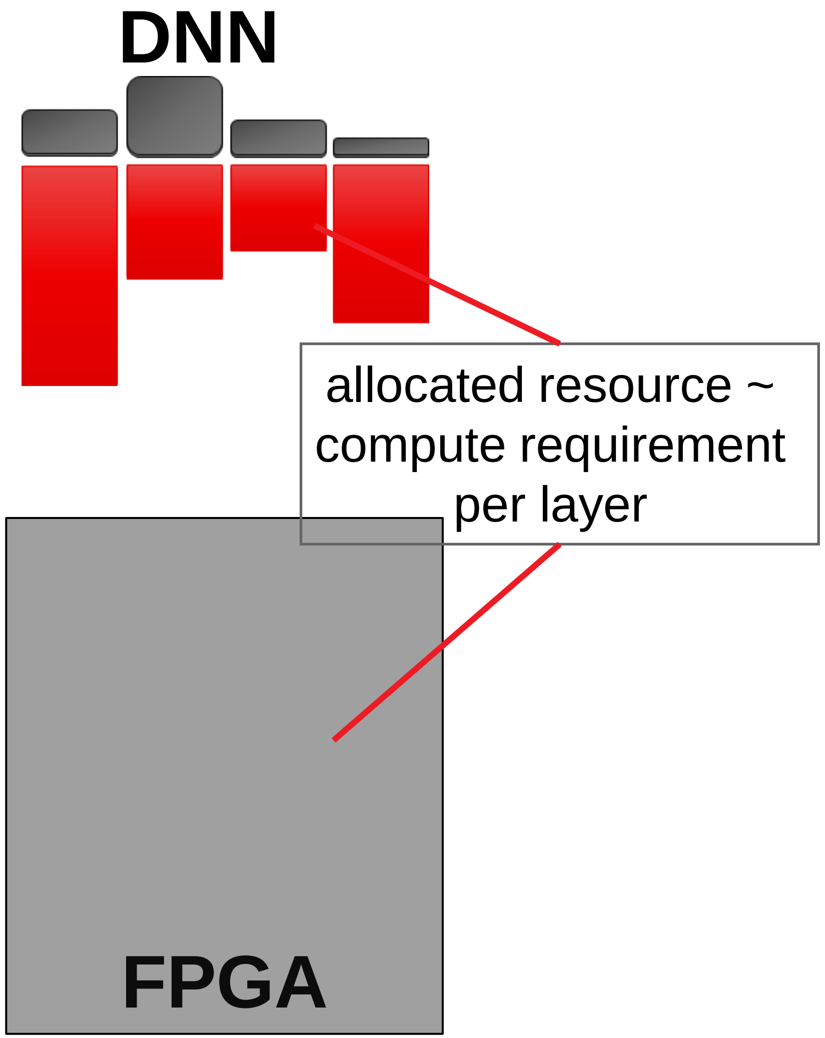
- Popular layer-by-layer compute
- Batching to achieve high compute efficiency
  - At latency cost (latency ~ batch size)
- Customized for ML in general
  - Designed to run any DNN
  - Specialized processing engines
    - Operators
    - ALU types
- Works really well for computer vision and natural language processing
- Popular approach: Vitis AI (FPGA or AIE) as well as majority of AI accelerators



Matrix of Processing Engines (MPE)
“layer-by-layer” compute

Dataflow - Specializing for Individual Topologies

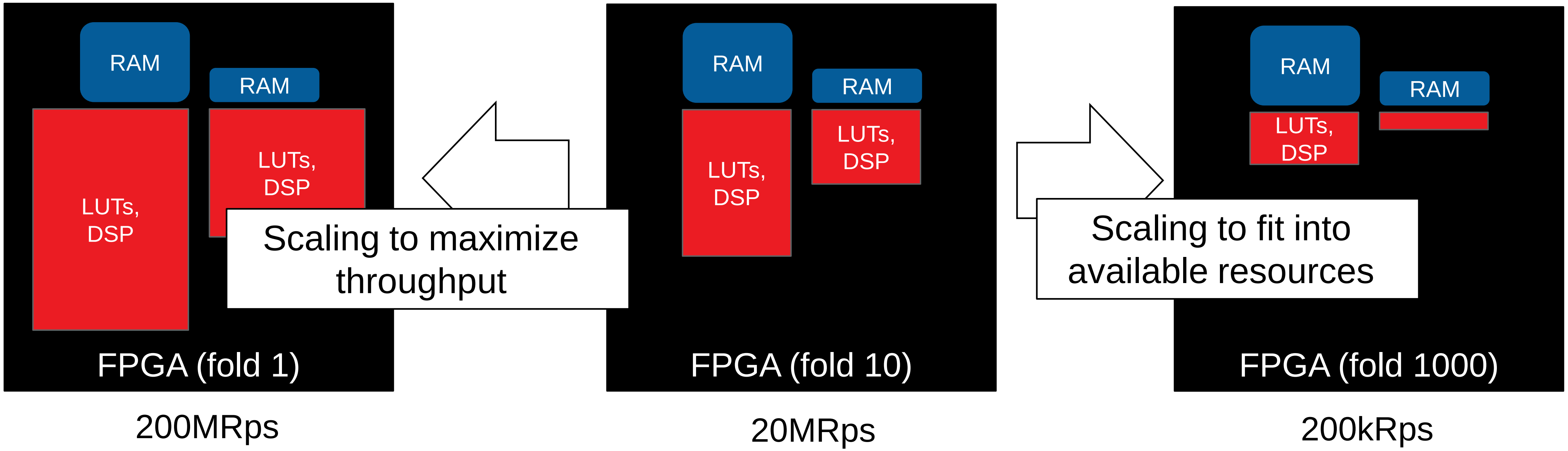
- Hardware instantiates the topology as a dataflow architecture
- Customize everything to the specifics of the given DNN, any operation, any connectivity
- Benefits
  - Improved efficiency
  - Low fixed latency
- Scale performance and resources to meet the application requirements



Dataflow can scale performance to meet the application requirements

Dataflow Processing:

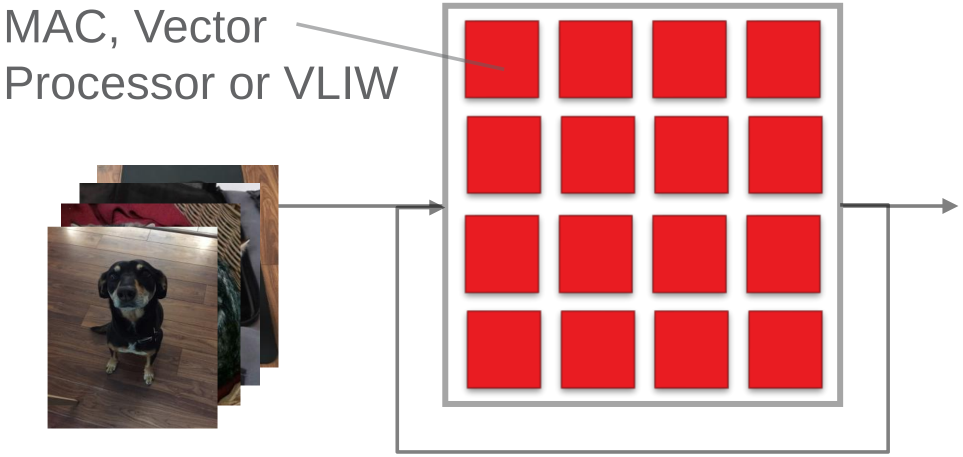
Scaling to Meet Performance and Resource Requirements



1. Scale performance and resources to meet the application requirements
2. If resources allow, we unfold completely, creating a circuit for inference at clock speed

Customized Dataflow Processing versus More Generic Architectures

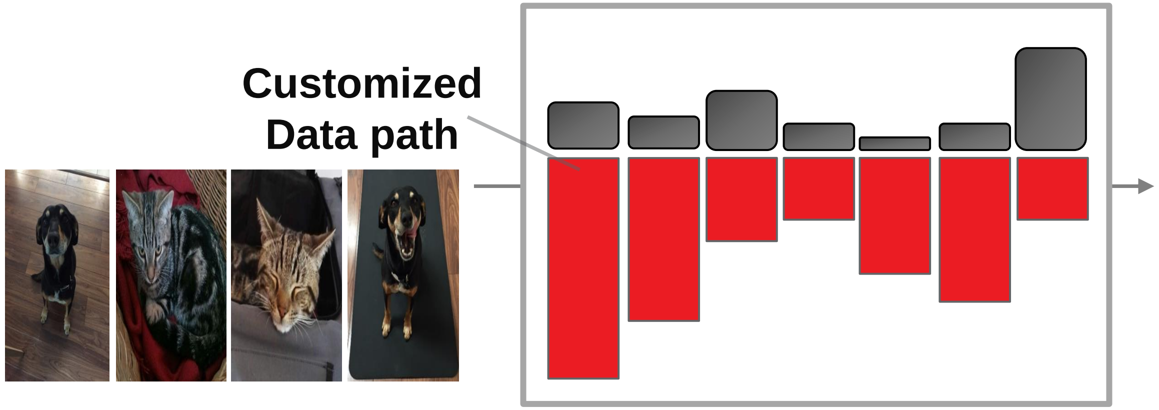
Matrix of Processing Engines (MPE) (Vitis AI, TPUs, GPUs)



MAC, Vector Processor or VLIW

- Customized for typical DNN operations
  - e.g., multiply accumulate
- Lower throughput (~10KRps)
- Flexibility through programming
- Applications: CV, Speech

Dataflow Architectures with FPGAs and FINN

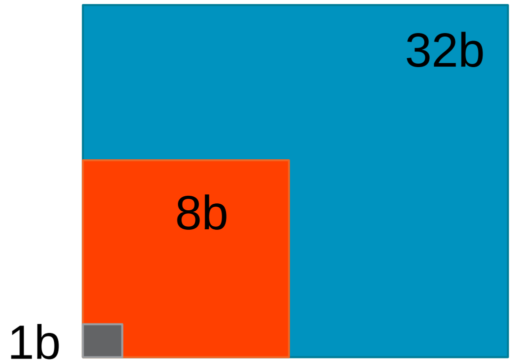


Customized Data path

- Customized/adapted for specific DNN topologies
- Streaming interfaces
- Specialization -> higher efficiency
- Lower latency (no intermediate buffering)
- Higher throughput (~100MRps)
- Flexibility through reconfiguration
- Applications: radio, networking, material science, particle physics – smaller DNNs

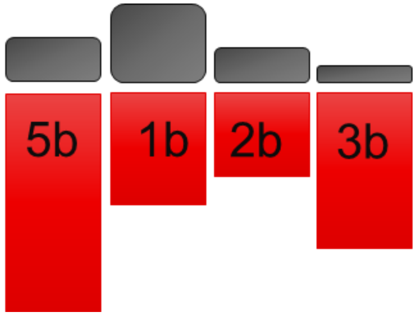
Quantization

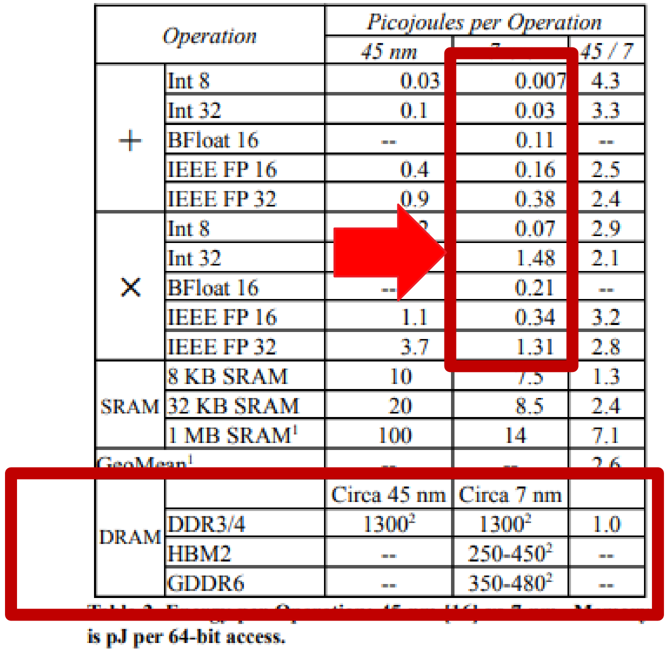
- Reducing precision shrinks hardware cost/scales performance
  - For integer datatypes, LUT cost proportional to both bitwidths in weight and activations (e.g., `INT8` : `INT1` ≈ 70×)
  - n-times more compute fits into the same fabric, thereby, scaling performance n-times or shrinking hardware cost accordingly



- Energy
  - Faster execution or smaller footprint → less energy ($E = P \cdot time$)
  - Using reduced precision operators saves energy
  - Reduces memory footprint
    - ResNet50 @ 32b: 102.5 MB, ResNet50 @ 2b: 6.4 MB
    - NN model can stay on-chip → no external memory access → saves energy

Precision	Model size [MB] (ResNet50)
1b	3.2
8b	25.5
32b	102.5



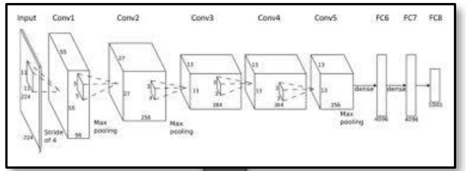


Operation		Picojoules per Operation		
		45 nm	7 nm	45 / 7
+	Int 8	0.03	0.007	4.3
	Int 32	0.1	0.03	3.3
	BFloat 16	--	0.11	--
	IEEE FP 16	0.4	0.16	2.5
	IEEE FP 32	0.9	0.38	2.4
×	Int 8	0.2	0.07	2.9
	Int 32	[illegible]	1.48	2.1
	BFloat 16	--	0.21	--
	IEEE FP 16	1.1	0.34	3.2
	IEEE FP 32	3.7	1.31	2.8
SRAM	8 KB SRAM	10	7.5	1.3
	32 KB SRAM	20	8.5	2.4
	1 MB SRAM[1]	100	14	7.1
GeoMean[1]		--	--	2.6
DRAM		Circa 45 nm	Circa 7 nm	
	DDR3/4	1300[2]	1300[2]	1.0
	HBM2	--	250-450[2]	--
	GDDR6	--	350-480[2]	--

Table 2. Energy per Operation: 45 nm [16] vs 7 nm. Memory is pJ per 64-bit access.

The FINN Framework

FINN Framework: From DNN to FPGA Deployment



Brevitas
Training in PyTorch
Algorithmic optimizations

- Train or even learn reduced precision DNNs
- Library of standard layers
- Pretrained examples

FINN Compiler
Hardware Architecture
Build

- Perform optimizations
- Assemble parameterized HLS/RTL modules
- Generate a DNN hardware IP

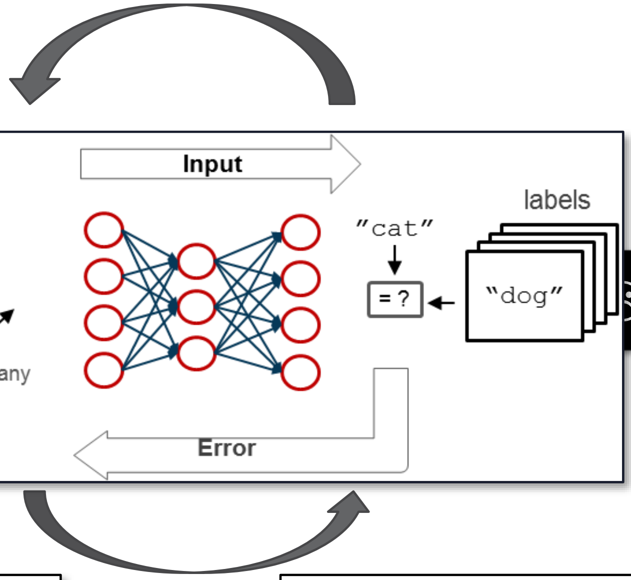
Deployment

- Embed the DNN IP into an infrastructure design
- Generate a Python run-time
- Enable integration with your application
- System integration available for some embedded and Alveo platforms, including HACC

Brevitas:
A PyTorch Library for Quantization-Aware Training

add quantization
resize layers
change hyperparameters
retrain

FP32



INT

Precision
Preset or learned

Scaling Factors
Granularities, strategies and constraints

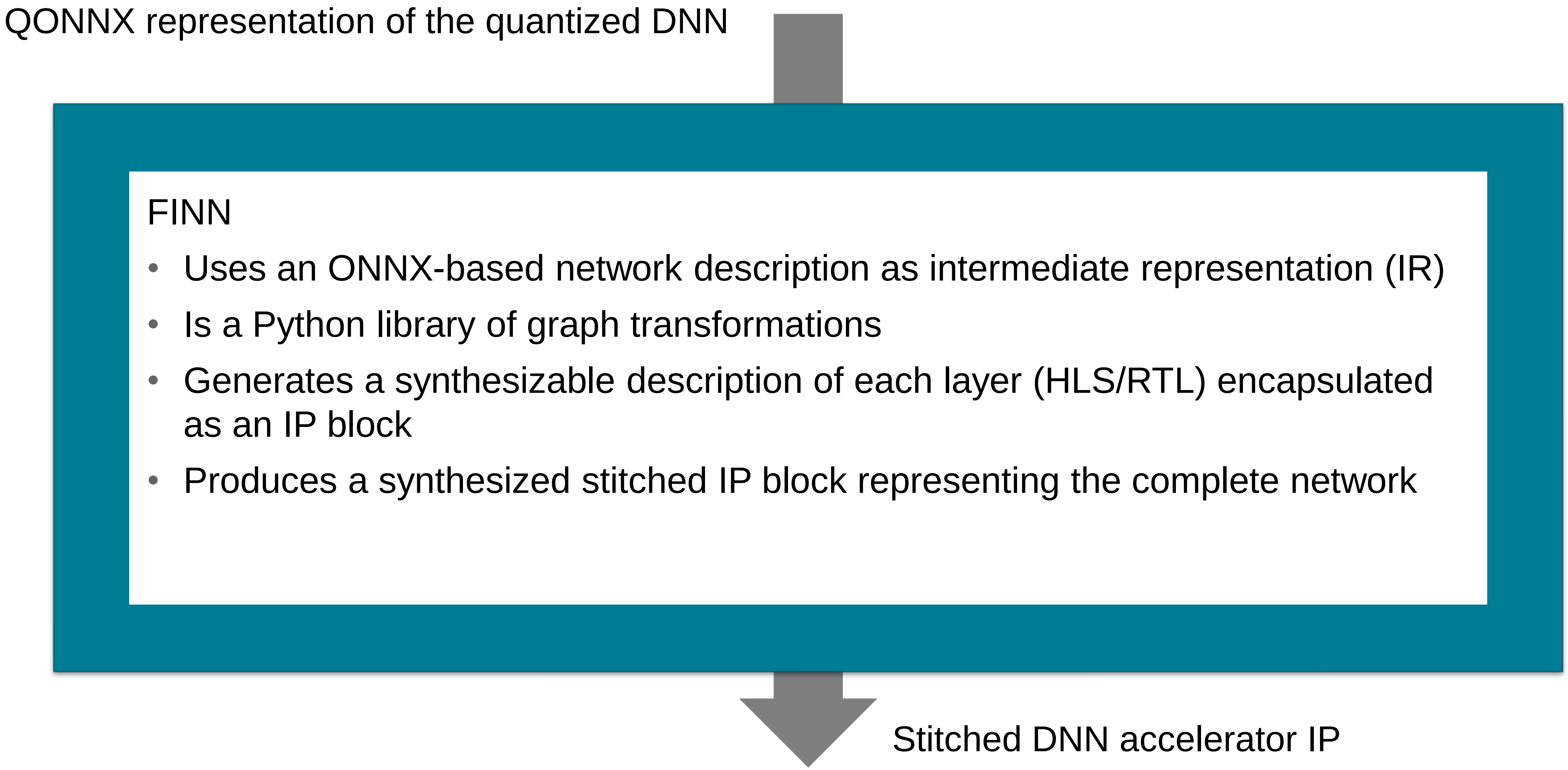
Target Tensors
Weights, activations, accumulators

Export to ONNX
To import into the FINN compiler

https://github.com/Xilinx/brevitas

FINN Compiler

Transform DNN into Custom Dataflow Architecture



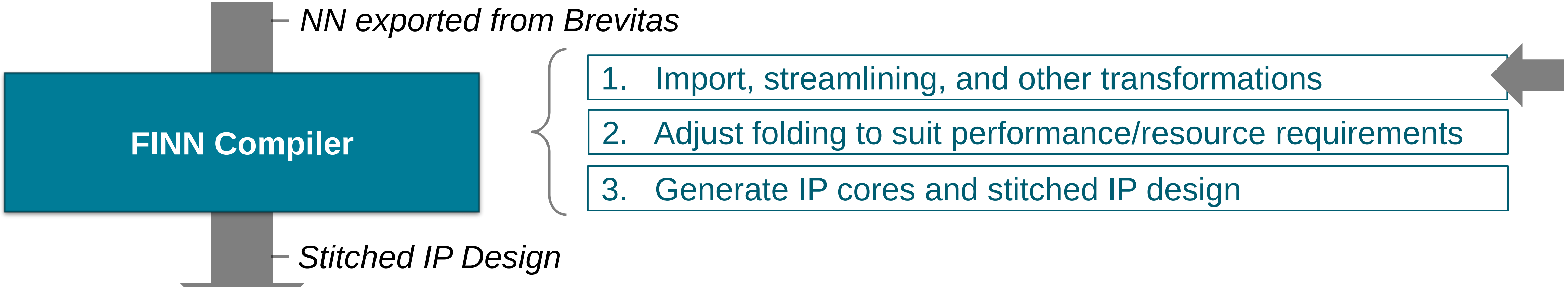
QONNX representation of the quantized DNN

FINN

- Uses an ONNX-based network description as intermediate representation (IR)
- Is a Python library of graph transformations
- Generates a synthesizable description of each layer (HLS/RTL) encapsulated as an IP block
- Produces a synthesized stitched IP block representing the complete network

Stitched DNN accelerator IP

FINN Compiler - Network preparation



NN exported from Brevitas

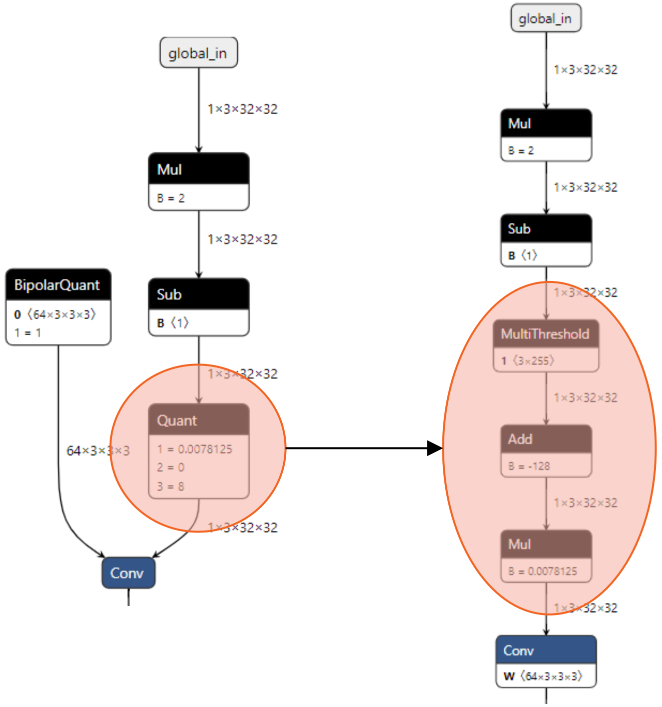
FINN Compiler

1. Import, streamlining, and other transformations
2. Adjust folding to suit performance/resource requirements
3. Generate IP cores and stitched IP design

Stitched IP Design

QONNX

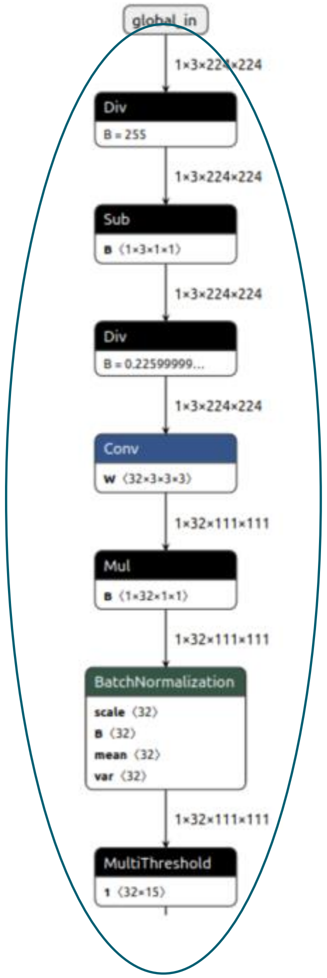
- Directly exported from Brevitas
- Input format to FINN compiler
- Quantization operator(s)
    - Quant, BipolarQuant, Trunc
- No tensor annotations

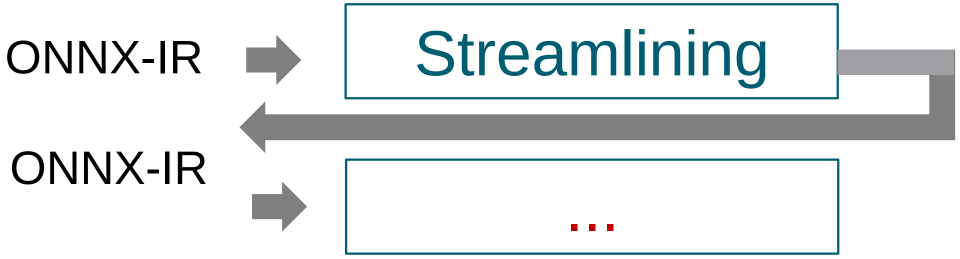


FINN-ONNX

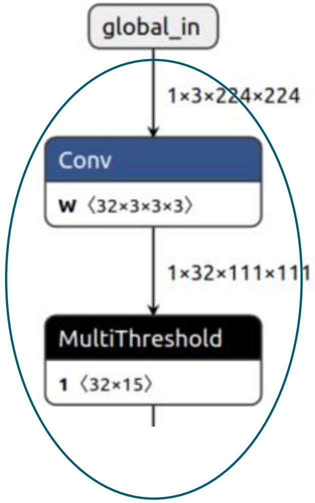
- Previously used as input format
- IR in the FINN compiler
- MultiThreshold to represent activation quantization
- Custom datatype annotations on tensor

FINN Passes - ONNX Graph Transformations



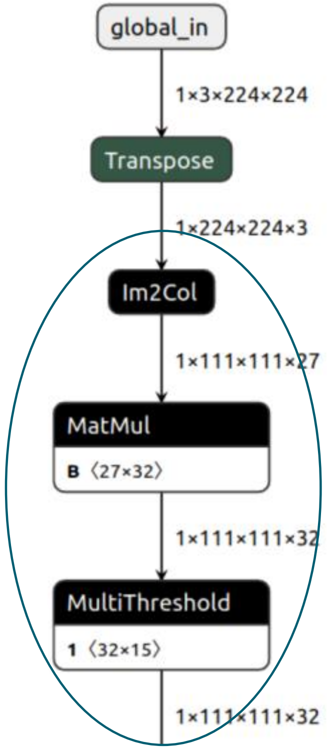


Streamline()



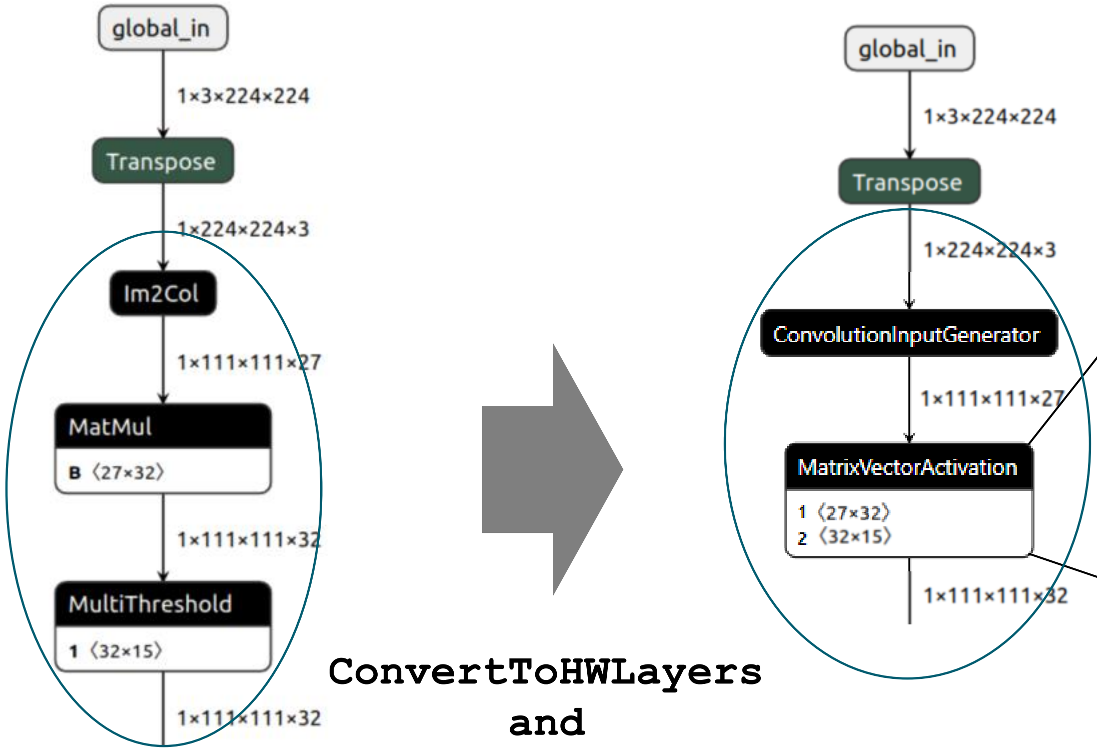
simplified

LowerConvsToMatMul()

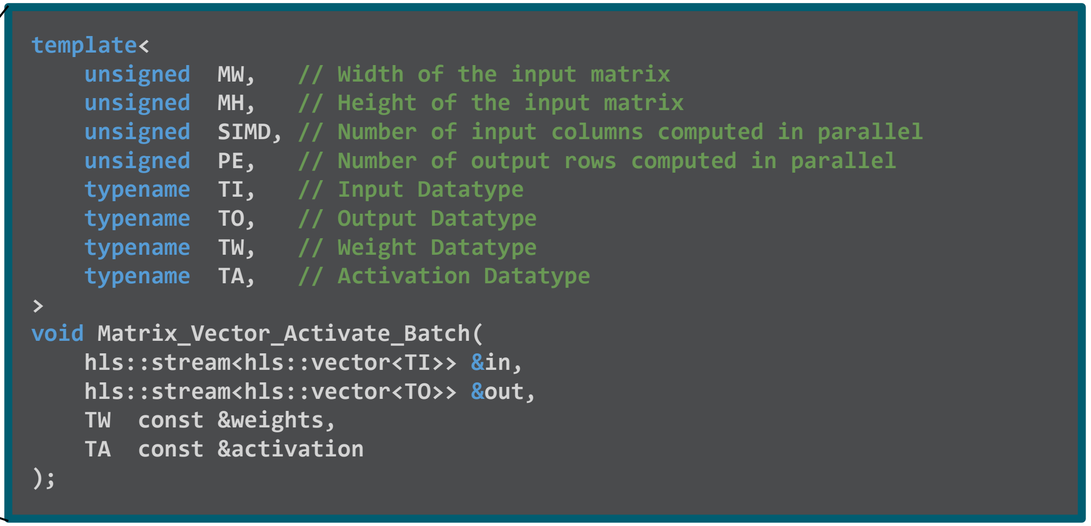


Convolutions as Im2Col + MatMul

FINN Passes - ONNX Graph Transformations



ConvertToHWLayers
and
SpecializeLayers

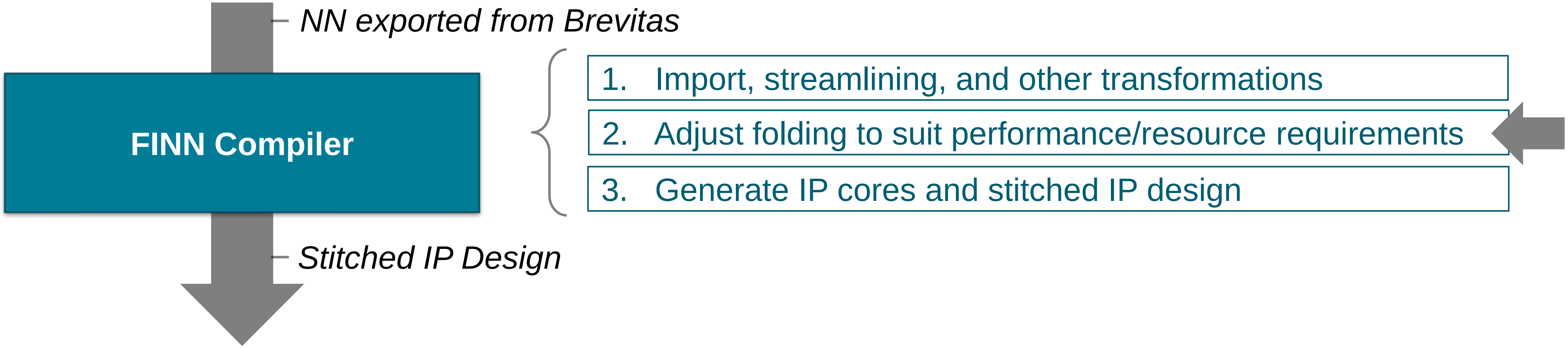


```
template<
    unsigned  MW,   // Width of the input matrix
    unsigned  MH,   // Height of the input matrix
    unsigned  SIMD, // Number of input columns computed in parallel
    unsigned  PE,   // Number of output rows computed in parallel
    typename  TI,   // Input Datatype
    typename  TO,   // Output Datatype
    typename  TW,   // Weight Datatype
    typename  TA,   // Activation Datatype
>
void Matrix_Vector_Activate_Batch(
    hls::stream<hls::vector<TI>> &in,
    hls::stream<hls::vector<TO>> &out,
    TW  const &weights,
    TA  const &activation
);
```

Corresponding to
finn-hlslib function call
or finn-rtllib module

Optimization, lowering, code generation... are all transformations

FINN Hardware Folding

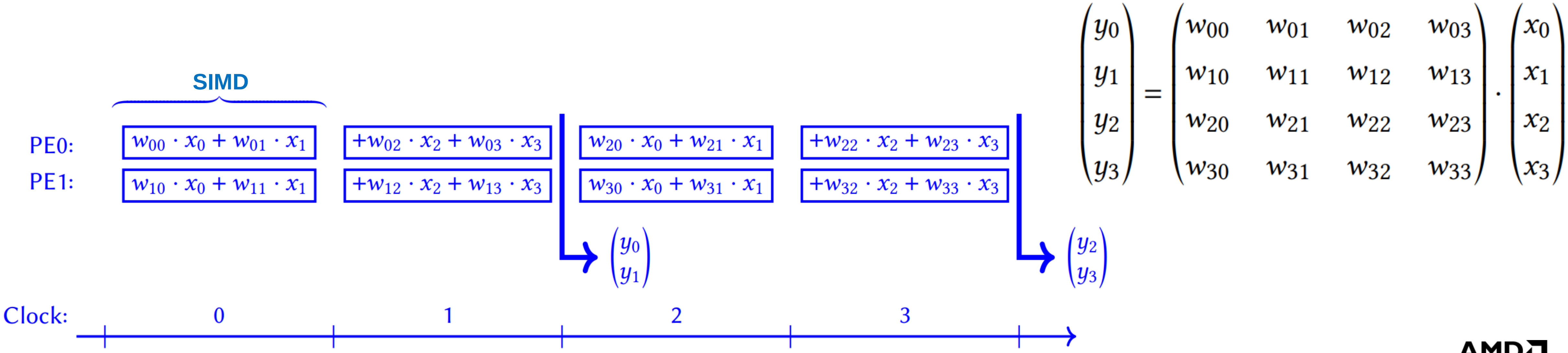


NN exported from Brevitas

FINN Compiler

1. Import, streamlining, and other transformations
2. Adjust folding to suit performance/resource requirements
3. Generate IP cores and stitched IP design

Stitched IP Design

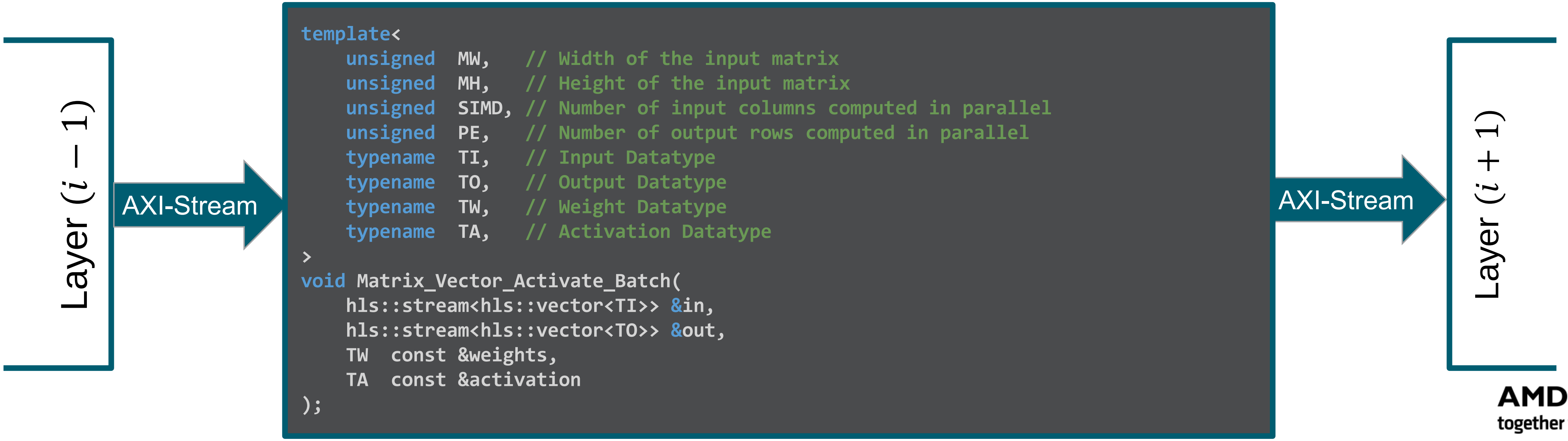


$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

SIMD
PE0:
PE1:
$w_{00} \cdot x_0 + w_{01} \cdot x_1$
$w_{10} \cdot x_0 + w_{11} \cdot x_1$
$+w_{02} \cdot x_2 + w_{03} \cdot x_3$
$+w_{12} \cdot x_2 + w_{13} \cdot x_3$
$w_{20} \cdot x_0 + w_{21} \cdot x_1$
$w_{30} \cdot x_0 + w_{31} \cdot x_1$
$+w_{22} \cdot x_2 + w_{23} \cdot x_3$
$+w_{32} \cdot x_2 + w_{33} \cdot x_3$
$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix}$
$\begin{pmatrix} y_2 \\ y_3 \end{pmatrix}$
Clock:
0
1
2
3

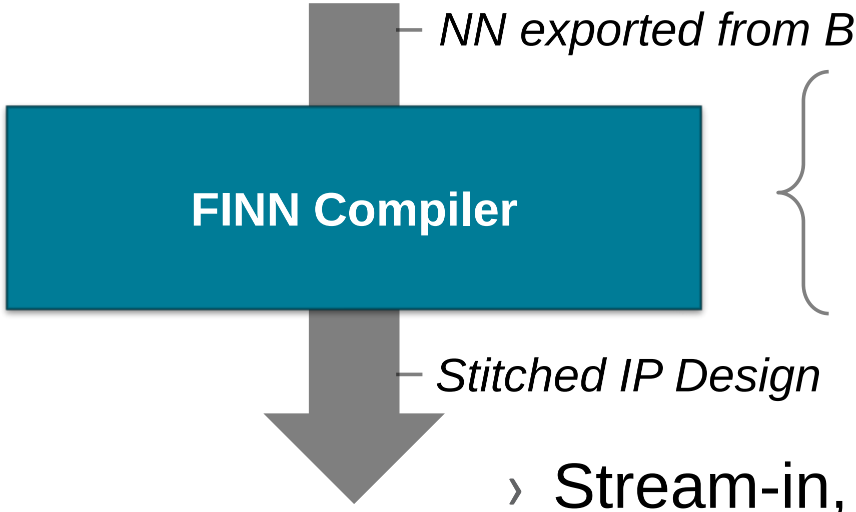
FINN HLS/RTL Library - Parameterizable Kernel Library

- Kernels representing individual layers, a.k.a. Operators
- Flexible parametrization as for
  - Degree of parallelism (output channels, input channels, kernel dimensions …)
  - Datatypes (INT8, ternary, INT2, …)
  - Behaviour (activation function)
- Instantiated and stitched by FINN compiler with AXI-Stream data path
- Implemented as synthesizable C++ (Vitis HLS) or SystemVerilog



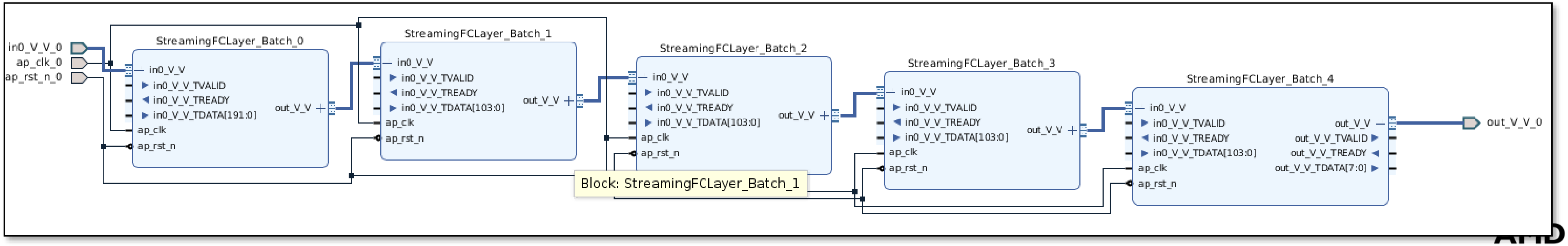
```
template<
    unsigned  MW,   // Width of the input matrix
    unsigned  MH,   // Height of the input matrix
    unsigned  SIMD, // Number of input columns computed in parallel
    unsigned  PE,   // Number of output rows computed in parallel
    typename  TI,   // Input Datatype
    typename  TO,   // Output Datatype
    typename  TW,   // Weight Datatype
    typename  TA,   // Activation Datatype
>
void Matrix_Vector_Activate_Batch(
    hls::stream<hls::vector<TI>> &in,
    hls::stream<hls::vector<TO>> &out,
    TW  const &weights,
    TA  const &activation
);
```

FINN Compiler: IP Generation Flow



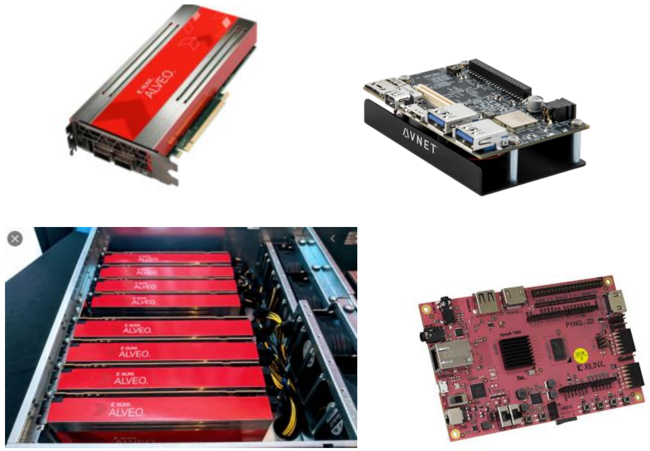
1. Import, streamlining, and other transformations
2. Adjust folding to suit performance/resource requirements
3. Generate IP cores and stitched IP design

- Stream-in, stream-out FPGA IP block
  - Easy "bump-in-the-wire" integration into streaming systems
  - Simple data movement, fully deterministic



Deployment with PYNQ™ for Python Productivity

```
# instantiate the accelerator
accel = models.cnv_w2a2_cifar10()
# generate an empty numpy array to use as input
dummy_in = np.empty(accel.ishape_normal, dtype=np.uint8)
# perform inference and get output
dummy_out = accel.execute(dummy_in)
```



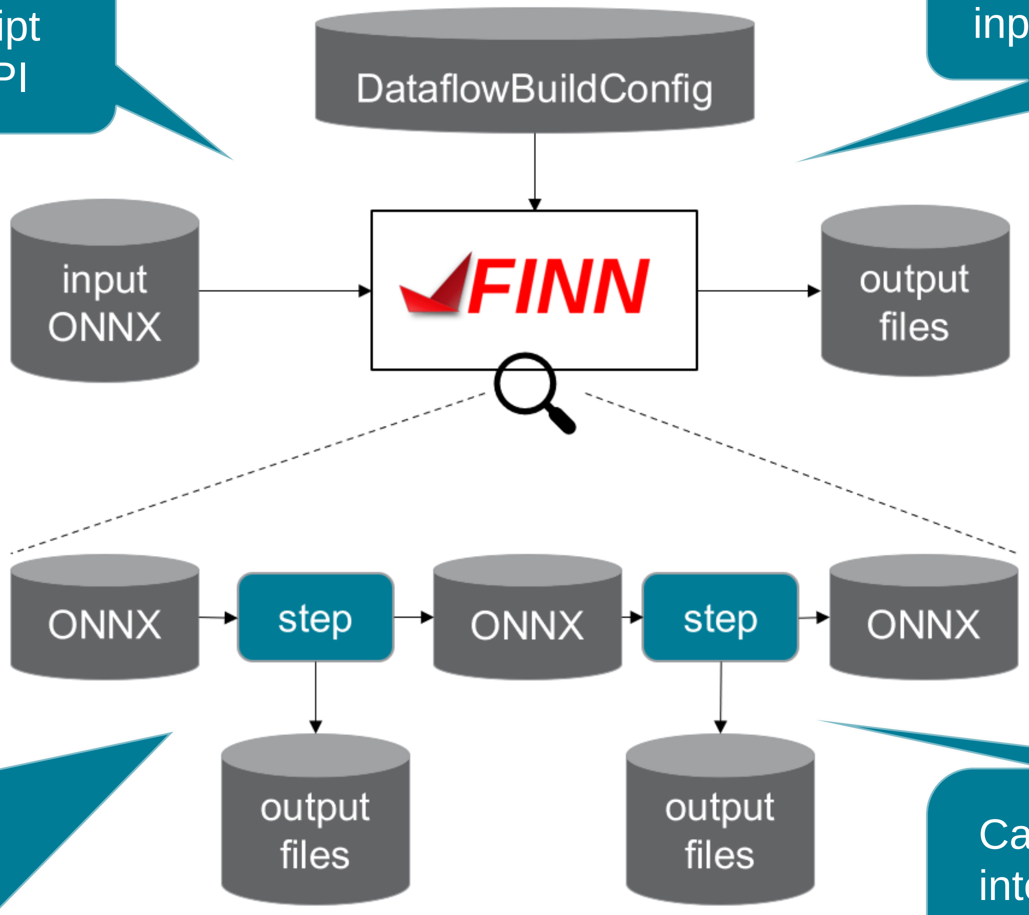
- Use PYNQ-provided Python abstractions and drivers
- User provides NumPy array input, calls driver, retrieves NumPy array output
  - Internally use PYNQ DMA driver to wr/rd NumPy arrays into I/O streams

https://github.com/Xilinx/PYNQ
https://github.com/Xilinx/finn-examples

FINN Dataflow Build Mode

FINN flow = Python script making calls to FINN API

Produce output files from input ONNX and config



Consists of a sequence of steps, each step

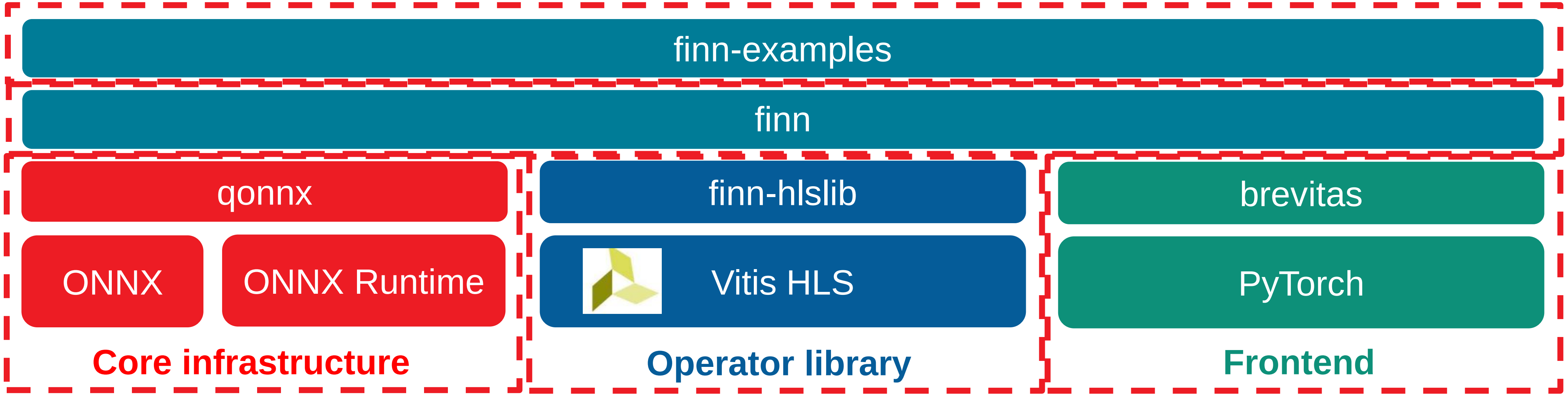
- is a Python function with a standardized interface
- consumes and produces ONNX
- may produce other files
- may be standard or custom
- may have config-dependent behavior

Can be resumed from intermediate steps

ONNX files act as checkpoints

FINN Infrastructure and Workflow

The FINN Ecosystem and Software Stack



finn-examples

finn

qonnx

ONNX

ONNX Runtime

Core infrastructure

finn-hlslib

Vitis HLS

Operator library

brevitas

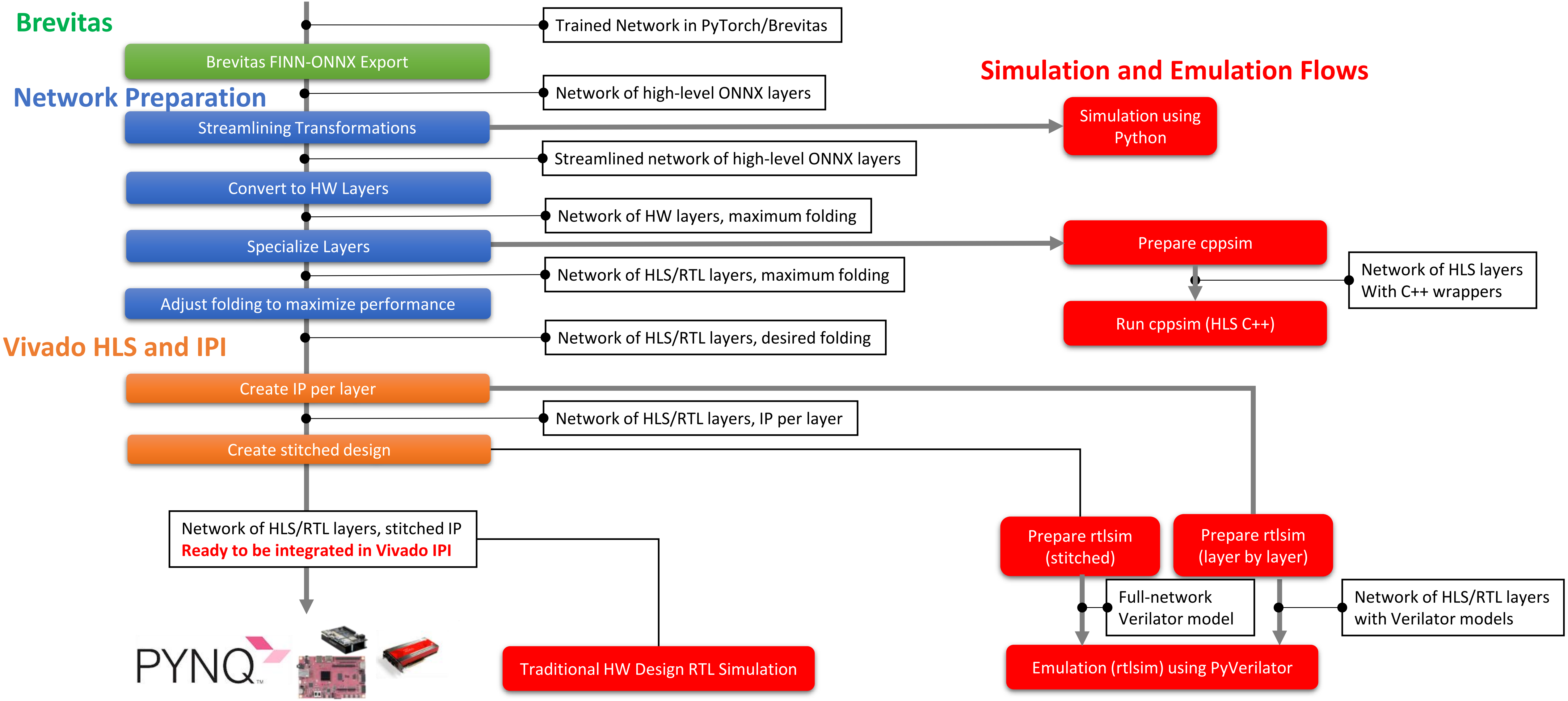
PyTorch

Frontend

FINN project landing page: https://xilinx.github.io/finn

- Quick Start, Documentation, Examples (Jupyter Notebooks)
- Links to Repos

A FINN End-to-End Flow



Brevitas

Trained Network in PyTorch/Brevitas

Brevitas FINN-ONNX Export

Network Preparation

Network of high-level ONNX layers

Streamlining Transformations

Streamlined network of high-level ONNX layers

Convert to HW Layers

Network of HW layers, maximum folding

Specialize Layers

Network of HLS/RTL layers, maximum folding

Adjust folding to maximize performance

Vivado HLS and IPI

Network of HLS/RTL layers, desired folding

Create IP per layer

Network of HLS/RTL layers, IP per layer

Create stitched design

Network of HLS/RTL layers, stitched IP
Ready to be integrated in Vivado IPI

Traditional HW Design RTL Simulation

Simulation and Emulation Flows

Simulation using Python

Prepare cppsim

Network of HLS layers
With C++ wrappers

Run cppsim (HLS C++)

Prepare rtlsim (stitched)

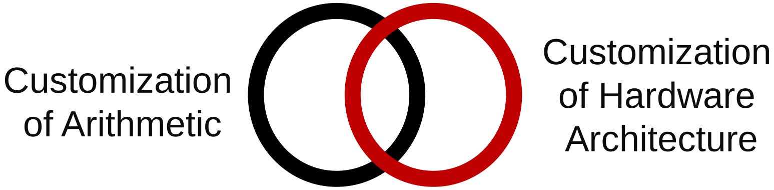
Prepare rtlsim (layer by layer)

Full-network Verilator model

Network of HLS/RTL layers with Verilator models

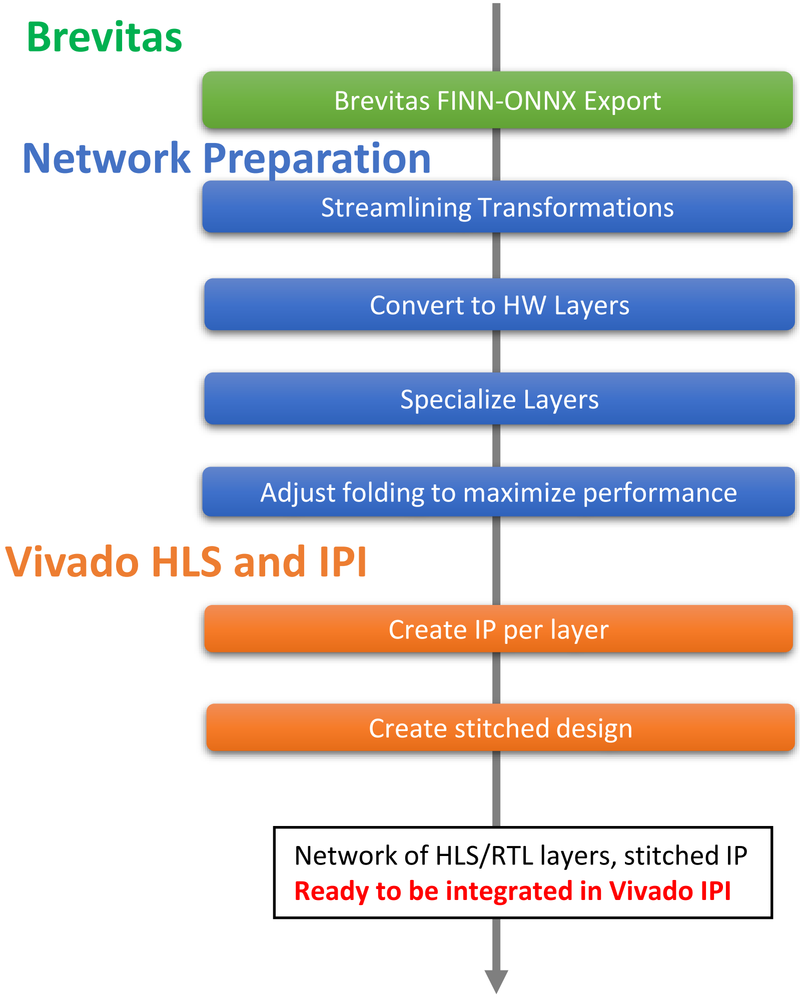
Emulation (rtlsim) using PyVerilator

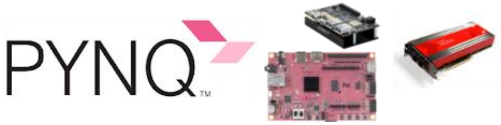
FINN Workflow


Customization of Arithmetic
Customization of Hardware Architecture

FINN and Brevitas can be used as co-design tools to implement your DNN use case on an FPGA.

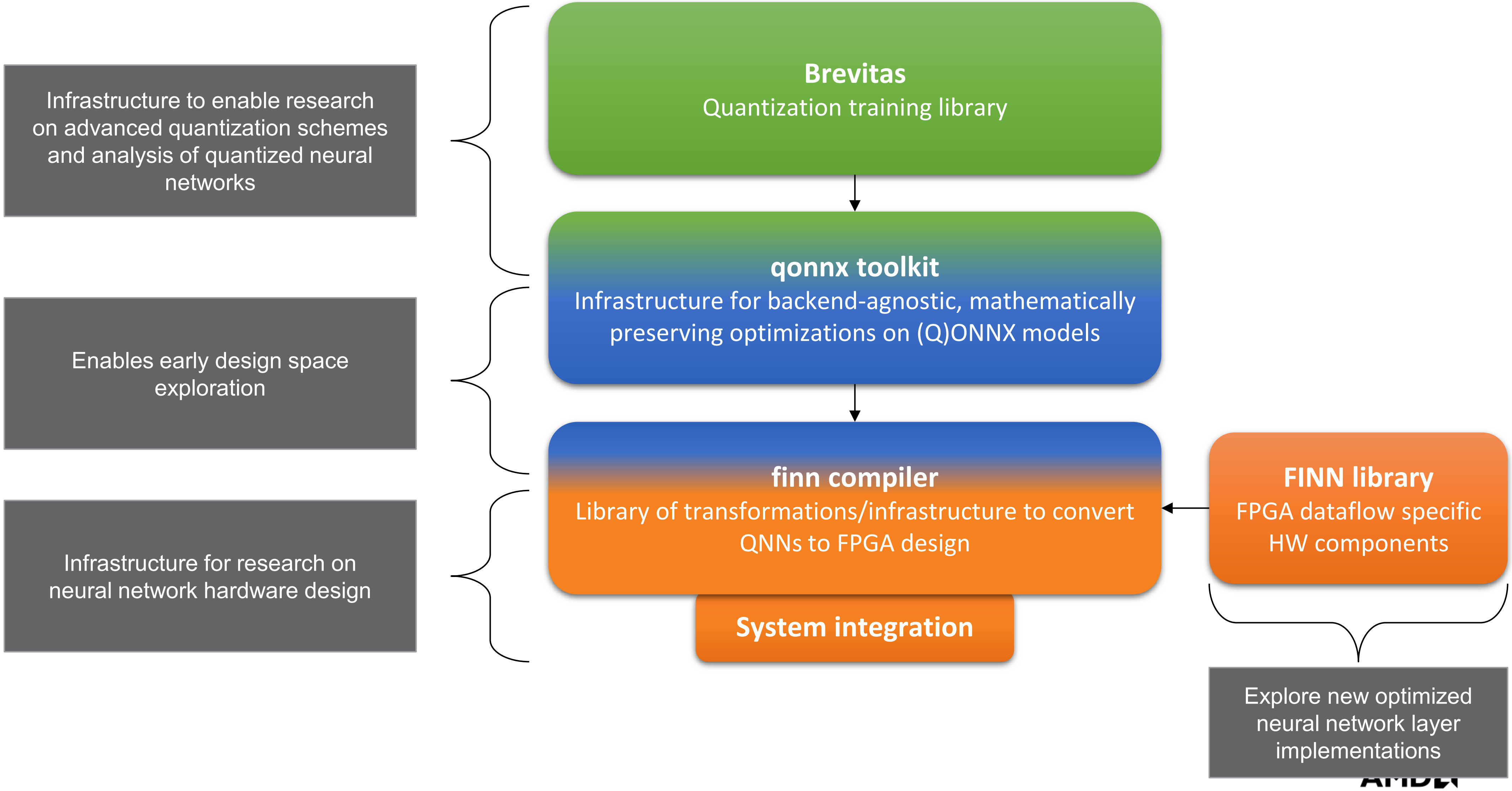
- Train a quantized neural network in PyTorch using Brevitas
- Converting trained QNN to Vivado IP
- Fine-tune model to meet resource/performance targets
- Integrate generated IP into a larger design


Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Streamlining Transformations
Convert to HW Layers
Specialize Layers
Adjust folding to maximize performance
Vivado HLS and IPI
Create IP per layer
Create stitched design
Network of HLS/RTL layers, stitched IP
Ready to be integrated in Vivado IPI



But you can leverage the infrastructure beyond that…

Research in the FINN Ecosystem



Infrastructure to enable research on advanced quantization schemes and analysis of quantized neural networks

Enables early design space exploration

Infrastructure for research on neural network hardware design

Brevitas
Quantization training library

qonnx toolkit
Infrastructure for backend-agnostic, mathematically preserving optimizations on (Q)ONNX models

finn compiler
Library of transformations/infrastructure to convert QNNs to FPGA design

System integration

FINN library
FPGA dataflow specific HW components

Explore new optimized neural network layer implementations

Status and Outlook

Status Summary

- **Open-Source Adoption**
  - ~2k+ GitHub stars summarized across repos
  - 250k+ Brevitas downloads
  - ~200k QONNX downloads
  - 17k+ FINN compiler downloads
- **Academic Results**
  - ACM TRETS 2020, FPL'2020, DFT'2019 Best Paper awards
  - 1000+ citations on original paper
- **University Classes on computer architecture for ML with FINN**
  - Stanford, UNC Charlotte, NTNU in Norway, EPFL in Switzerland
  - Regular tutorials, also available on YouTube: https://www.youtube.com/watch?v=zw2aG4PhzmA
- **Business units providing customer support**
  - Lead engineering team: Custom and Strategic Engineering, Dublin

> *"The FINN toolset is showing **huge potential using it in upcoming SICK products**.*
> *It is **easy to use** and with an **extraordinary performance** and very promising results.*
> *In the future, flexible implementations of ML in our products with FINN can be a great advantage and even replace static architectures as they are currently used.*
> *Thanks to the FINN team for the great cooperation"*
> *– Sick AG*

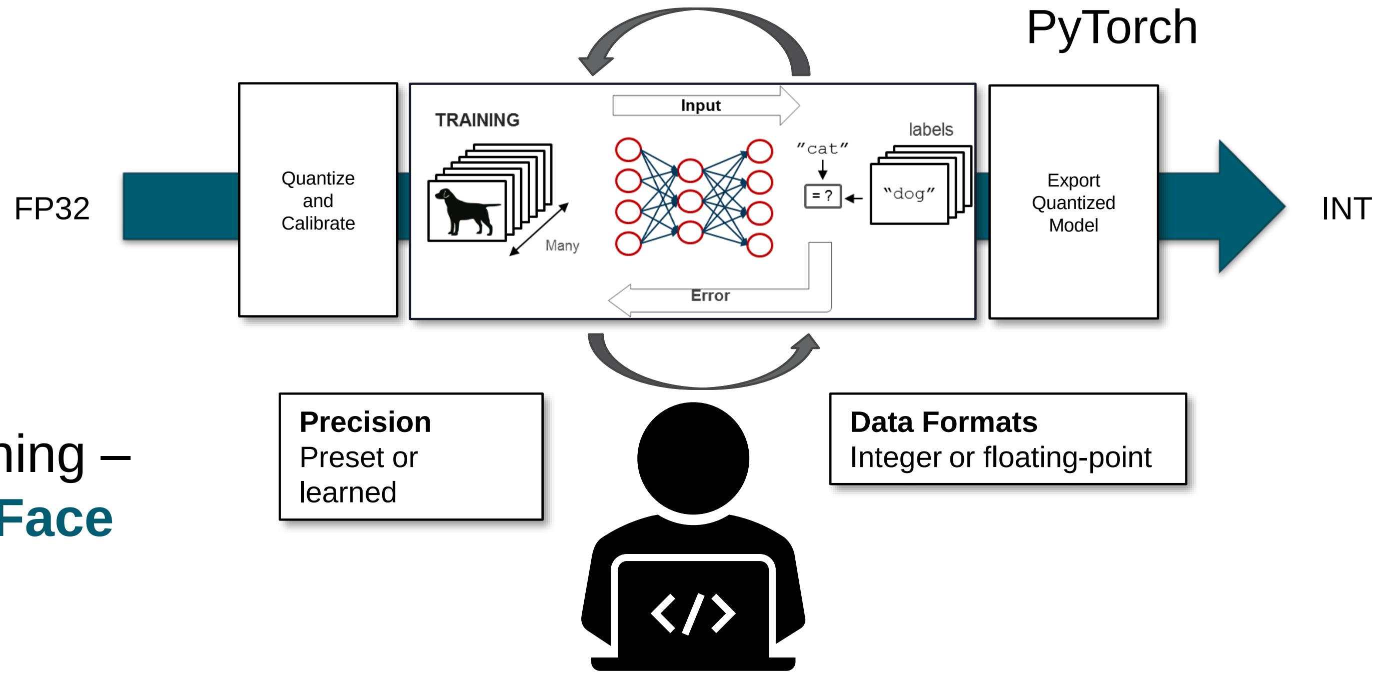
https://github.com/Xilinx/brevitas
https://github.com/Xilinx/finn
https://github.com/Xilinx/finn-hlslib
https://github.com/Xilinx/finn-examples
https://github.com/fastmachinelearning/qonnx

FINN Layer Support

Layer	Current Support	Outlook
GEMM	✓	
Conv1D and Conv2D	✓	
- Dense	✓	
- Depthwise	✓	
- Separable (pointwise)	✓	
Elementwise (add**,** sub**)**	✓	others easily doable
Activation	ReLU, SeLU	
BatchNorm	✓ (absorbed by streamlining)	
Pooling	✓	
Scale	✓ (absorbed by streamlining)	
Concat	✓	
Reshape	✓ (must be streamlinable)	
Transpose	✓ (must be streamlinable)	
Clip by Value	✓ (absorbed by streamlining)	
TransposeConv2D	✓	optimized version (WIP)
UpSample	✓	
DownSample	✓	

Brevitas Updates

- Targets the **entire AMD product range**
- First-class support for **integer datatypes**
  - prototype support for minifloats (e.g., FP8)
- Supports **PTQ and QAT**
- Out of the box support for distributed training – (e.g., DDP, interoperability with **HuggingFace Accelerate** (PP))
- Interoperability with **HuggingFace Transformers**



Precision
Preset or learned

Data Formats
Integer or floating-point

https://github.com/Xilinx/brevitas

FINN Compiler Updates

FINN v0.10.1 Release

- **Refactoring** of operator instantiation infrastructure
  - FINN compiler used to assume that hardware blocks are synthesized from HLS code
  - New class hierarchy to facilitate integration of RTL components
  - Provide users with an interface to override the compiler's choice for HLS vs. RTL implementation on a per-layer basis
- **RTL component** library optimizing the implementations of critical layers
  - Efficient implementation of 4-bit and 8-bit compute leveraging DSP slices
  - Efficient implementation of multi-level thresholding
  - Eradication of (regularly long) HLS synthesis times for layers with an RTL option
- **Compiler optimization pass** for accumulator and weight bit width minimization
- **Added board support** in system integration flow
  - **RFSoC 4x2** and **U55C** (contributed by University of Paderborn)

FINN Technical Roadmap: Capabilities

- Operator Hardening
  - **Revised RTL Thresholding by binary search**
    - Ingestion of fp32 inputs
  - DSP-enabled Generalized Datatype Support
    - Efficient higher-precision integer compute: **`int4, int8`**, …, `int16`
    - Small standard floating-point formats: `float16`, `bfloat16`
    - Custom MiniFloats: `fp4` – `fp8`
  - Internal clock pumping of DSP datapaths to increase their operational density
    We are aiming at a standard operational frequency around 500 MHz
- New Operators
  - Optimized **transposed convolution**
  - Fallback float layers to mitigate streamlining limits

FINN Technical Roadmap: Ease of Use

- **FINN Library**
  - Refactoring of streamed layer interfaces
    - Packed flat `ap_uint<W>` → explicit `hls::vector<T, N>`
  - Combining HLS and RTL components into one FINN Library
- **FINN Examples**
  - MobileNet-v1 and VGG10-RadioML **with efficient DSP compute**
  - New example: German Traffic Sign Recognition Benchmark

FINN-examples v0.0.7 Release

Resources

- https://github.com/Xilinx/brevitas
  https://github.com/Xilinx/finn
- https://github.com/Xilinx/finn-hlslib
- https://github.com/Xilinx/finn-examples
- https://github.com/fastmachinelearning/qonnx

- https://amd.com/aup

Q & A

COPYRIGHT AND DISCLAIMER

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 releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or 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.

AMD