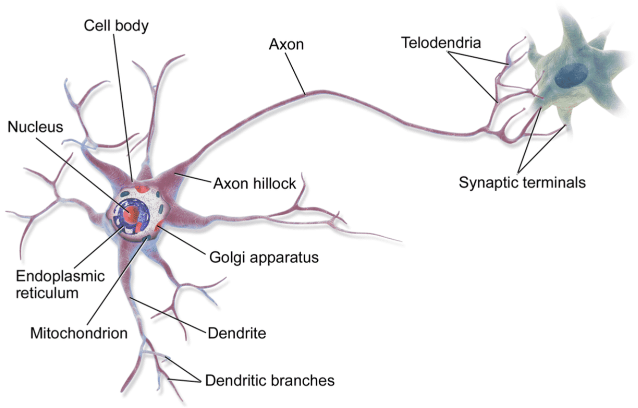
Fotakis Tzanis
Electrical & Computer Engineering School
Technical University of Crete

Analysis and Design Methodology of Convolutional Neural Networks mapping on Reconfigurable Logic

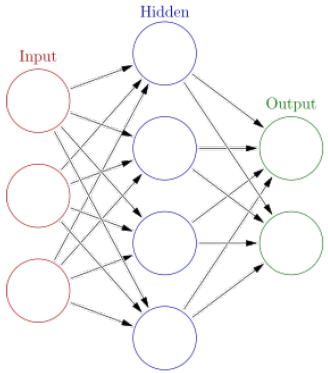
Diploma Thesis

Table of Contents

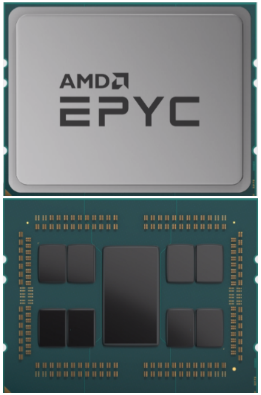
What are Neural Networks?



Representation of Artificial Neural Networks



ANNs on CPUs

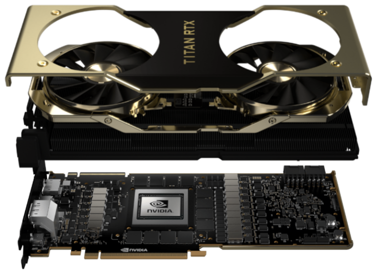


AMD Epyc 7002 series chip

+ Easy development
+ High clock frequency
+ Advanced Vector Extensions (AVX)
+ Streaming SIMD Extensions (SSE)

- Costly
- Non-scalable
- Energy inefficient
- Traditional Low Bandwidth Memory - up to 50 GB/s

ANNs on GPUs

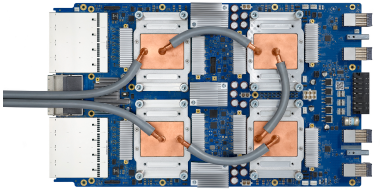


NVIDIA Titan RTX card

+ Relatively easy development
+ Very high parallelism - Thousands of cores
+ Specialized Tensor Cores
+ Vector processing - Streaming Multiprocessors
+ High Bandwidth Memory - up to 750 GB/s
+ Multiple GPUs in a system

- Very power hungry
- Costly to scale up
- Increased latency

ANNs on ASICs


Google TPU v3

+ Best parallelism
+ Lowest power & energy consumption
+ High Bandwidth Memory
- Extremely expensive to design and produce
- Serve a single purpose
- Can become deprecated fast - AI field is still developing

ANNs on FPGAs

FORTH QFDB

+ Flexible
+ Low power & energy consumption
+ Low latency
+ Standalone

- Difficult to develop
- Constrained resources

What is Reconfigurable Logic

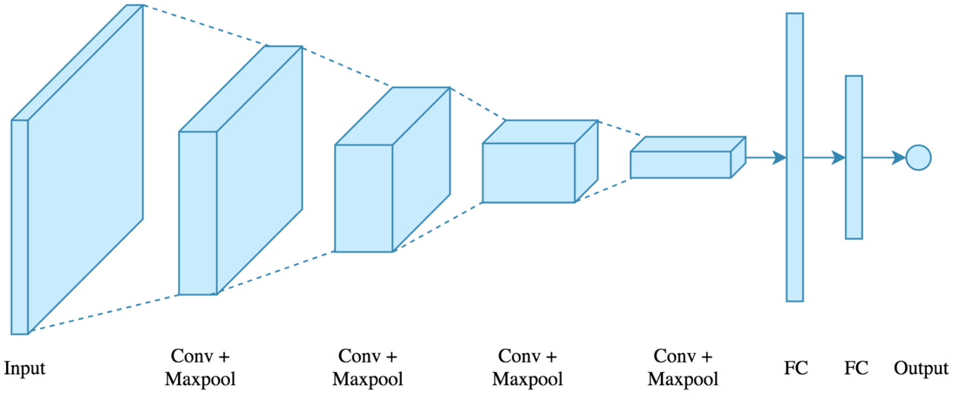
- Look-Up Tables (LUTs)
- Flip-Flops (FFs)
- Block-RAM (BRAM)
- Ultra-RAM (URAM)
- Digital Signal Processing (DSP) blocks
- Hard Processor cores (SoC/MPSoC devices)
- DDR & HBM (modern FPGAs)

Neural Networks

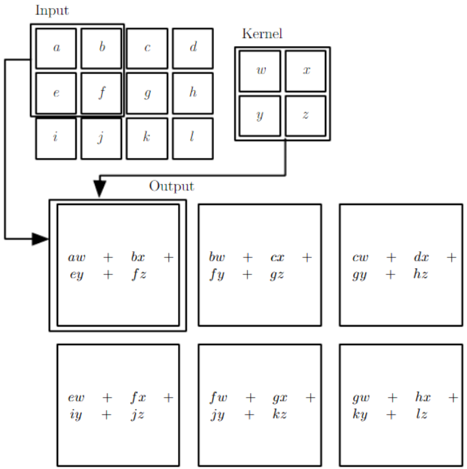
ANN Architectures

- Multiclass Perceptron
- Autoencoder
- Convolutional
- Recurrent
- Long short-term memory

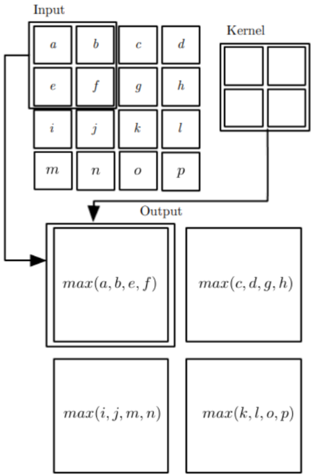
Typical Convolutional Neural Network



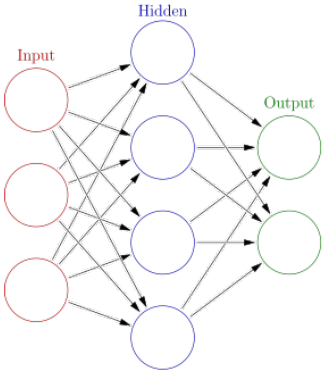
The Convolution Layer



The Max-Pooling Layer

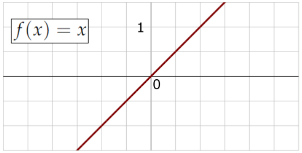


The Fully-Connected Layer

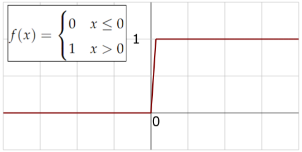


$\mathrm{Output}(i) = \mathrm{Bias}(i) + \sum_{j=1}^{N} \mathrm{Input}(j) * \mathit{Weight}(i,j)$, for i = 1, 2, ..., M

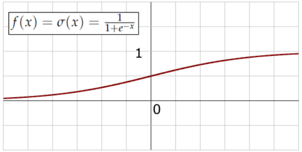
Activation Functions



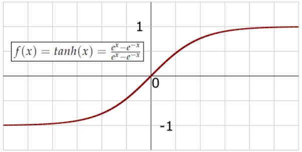
Identity



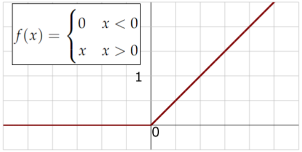
Binary step



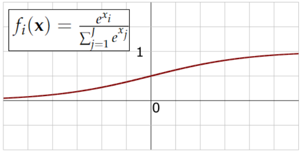
Sigmoid



tanh

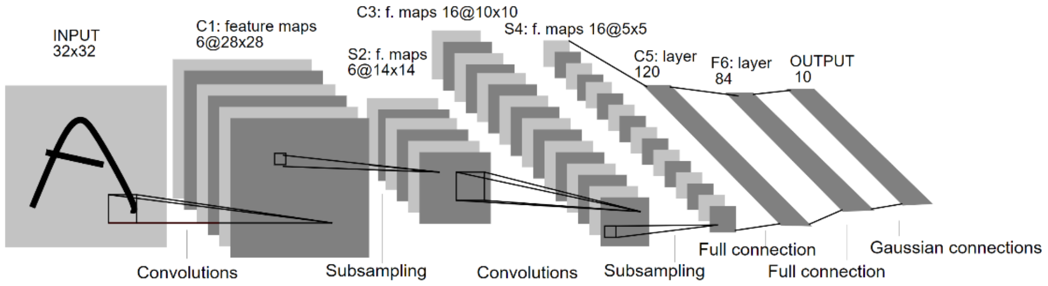


ReLU

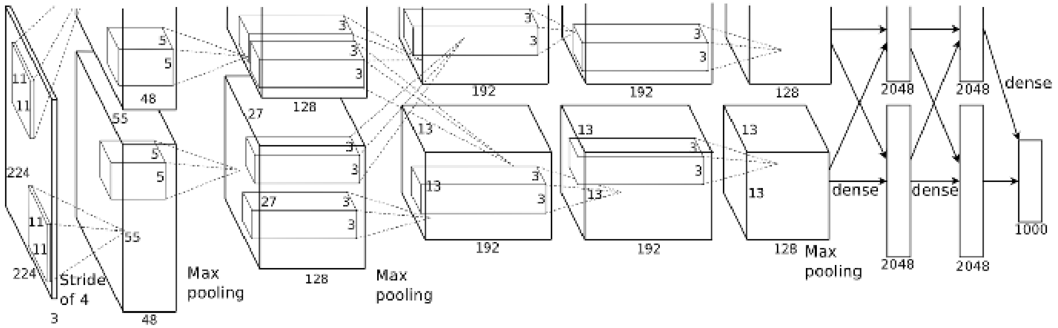


Softmax

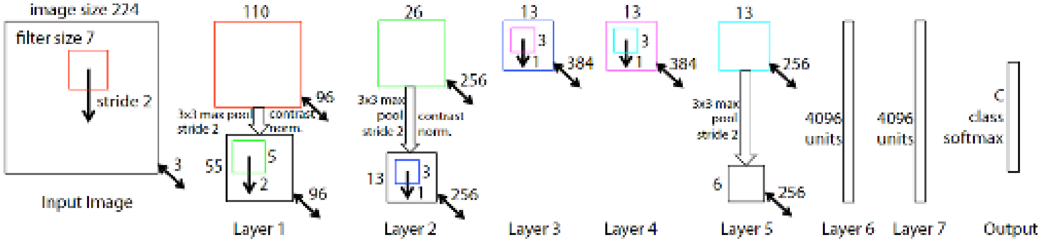
CNN Architectures: LeNet-5



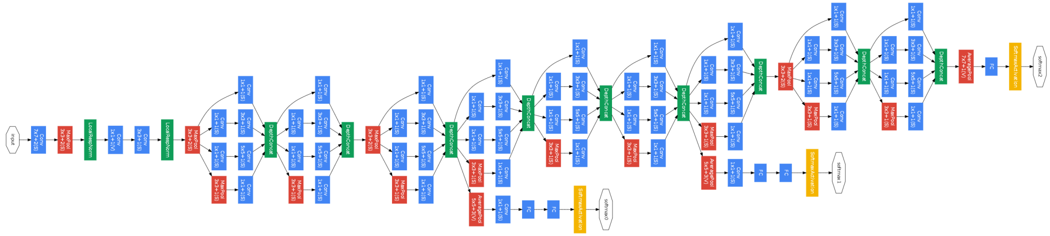
CNN Architectures: AlexNet



CNN Architectures: ZFNet



CNN Architectures: GoogLeNet / Inception



CNN Architectures: VGGNet

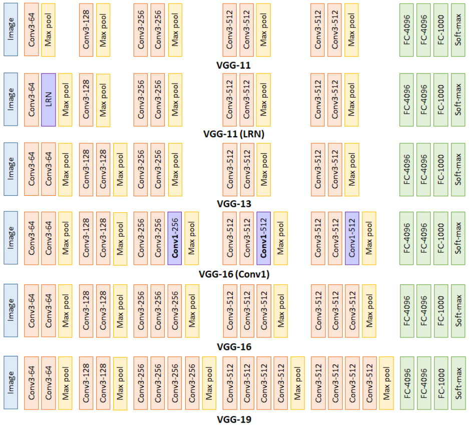
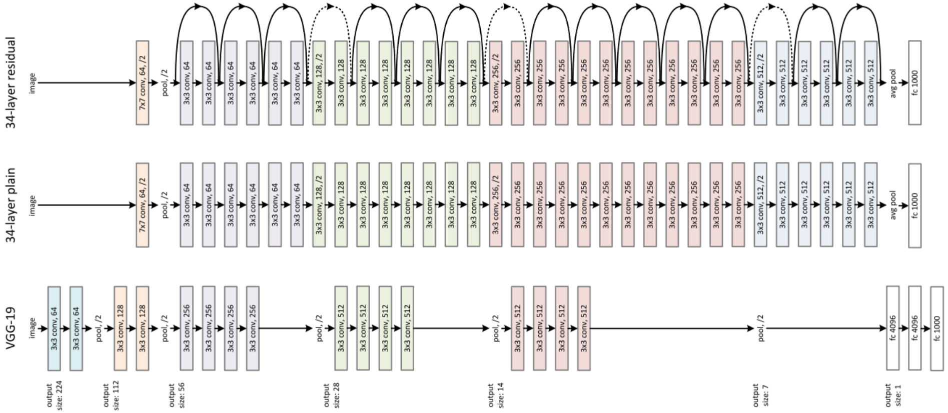


Image Conv3-64 Max pool Conv3-128 Max pool Conv3-256 Conv3-256 Max pool Conv3-512 Conv3-512 Max pool Conv3-512 Conv3-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-11
Image Conv3-64 LRN Max pool Conv3-128 Max pool Conv3-256 Conv3-256 Max pool Conv3-512 Conv3-512 Max pool Conv3-512 Conv3-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-11 (LRN)
Image Conv3-64 Conv3-64 Max pool Conv3-128 Conv3-128 Max pool Conv3-256 Conv3-256 Max pool Conv3-512 Conv3-512 Max pool Conv3-512 Conv3-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-13
Image Conv3-64 Conv3-64 Max pool Conv3-128 Conv3-128 Max pool Conv3-256 Conv3-256 Conv1-256 Max pool Conv3-512 Conv3-512 Conv1-512 Max pool Conv3-512 Conv3-512 Conv1-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-16 (Conv1)
Image Conv3-64 Conv3-64 Max pool Conv3-128 Conv3-128 Max pool Conv3-256 Conv3-256 Conv3-256 Max pool Conv3-512 Conv3-512 Conv3-512 Max pool Conv3-512 Conv3-512 Conv3-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-16
Image Conv3-64 Conv3-64 Max pool Conv3-128 Conv3-128 Max pool Conv3-256 Conv3-256 Conv3-256 Conv3-256 Max pool Conv3-512 Conv3-512 Conv3-512 Conv3-512 Max pool Conv3-512 Conv3-512 Conv3-512 Conv3-512 Max pool FC-4096 FC-4096 FC-1000 Soft-max
VGG-19

	Number of Parameters (millions)	Top-5 Error Rate (%)
VGG-11	133	10.4
VGG-11 (LRN)	133	10.5
VGG-13	133	9.9
VGG-16 (Conv1)	134	9.4
VGG-16	138	8.8
VGG-19	144	9.0

CNN Architectures: ResNet



Related Work

Software Frameworks

TensorFlow PyTorch

Keras Caffe

Software Frameworks: TensorFlow

- By Google
- Most popular
- Lower-level - Much coding - High Configurability
- Python, Javascript, C++, C#, Java, Go & Julia interfaces
- Targeted for production
- Static computation graph - Efficient but less flexible
- CPU, GPU & TPU acceleration
- Server, Mobile & Embedded platforms

Software Frameworks: PyTorch

PyTorch

- By Facebook
- Based on Torch
- CPU & GPU acceleration
- Dynamically updated graph
- Targeted for prototyping & research
- Contains many pre-trained models

Software Frameworks: Keras

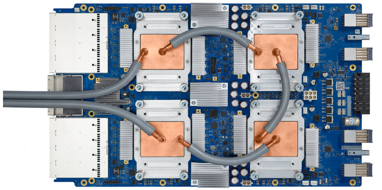
- Higher-level
- Back-ends: TensorFlow, Theano & CNTK
- Easy huge models - Less configurable
- Targeted for learning & prototyping

Software Frameworks: Caffe

- By Berkley, University of California
- Written in C++ - Python interface
- CPU & GPU acceleration
- Not all Neural Networks supported
- Caffe2 merged with PyTorch

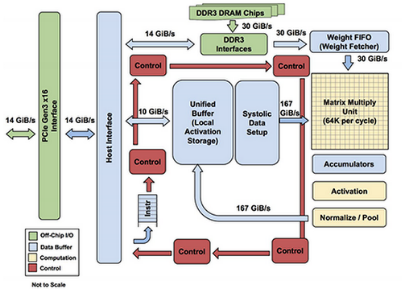
Hardware Frameworks

Hardware Frameworks: Google TPU


Google TPU v3

- By Google since 2015
- Used with TensorFlow
- Accelerate 95% of their AI needs
- Publicly available on Google Compute Engine since 2017
- Initially only inference & 8-bit fixed-point
- 200x compared to Intel Haswell CPU
- 70x compared to NVIDIA K80 GPU
- Version 2 and above add training & floating-point

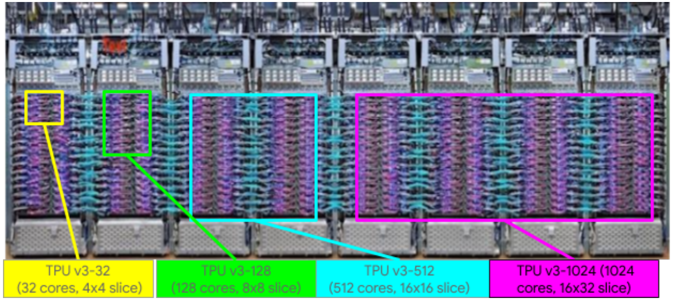
Hardware Frameworks: Google TPU v1 Block Diagram



TPU v1 Block Diagram

- Operate on matrices - GPUs operate on vectors
- 2x Matrix Multiplier Units (MXUs)
- MXU based on systolic arrays
- 16k MAC ops/cycle
- Up to 128GB HBM (TPU v3)
- 92 TOPS

Hardware Frameworks: Google TPU v3 Pods



- 2048 TPU Cores
- 32TB HBM
- 92 PFLOPS
- Suitable for very large models (weeks/months of training) - no custom operations

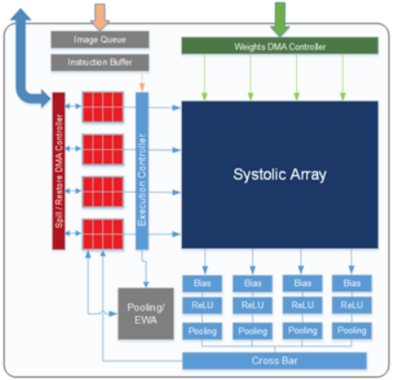
FPGA Frameworks: Xilinx CHaiDNN

- Released in February 2018 - Targeted for CNN inference
- 6-bit & 8-bit fixed-point - variable through layers
- Similar to single-precision floating-point
- Dynamic fixed-point Quantization & Xilinx Quantizer
- 128-1024 Double-Pumped DSPs - Up to 700MHz - URAM supported
- Unsupported layers added via Software
- Fully-Connected & SoftMax layers implemented through Software
- Hardware & Software layers run in parallel
- PetaLinux - Caffe framework
- DietCHai for smaller MPSoCs

FPGA Frameworks: Xilinx DPU

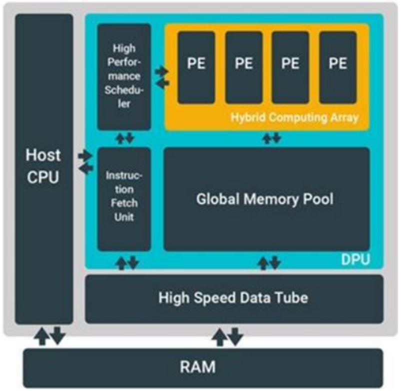
- Released in February 2019 - Replaces CHaiDNN - Still in development
- Targeted for CNN inference - 8-bit fixed-point
- On-Chip memory utilized as buffer
- All layers are hardware accelerated
- Up to four DPU cores in a single DPU IP
- Double Data Rate / Double-Pumped DSPs
- 512-4096 operations per cycle per core

FPGA Frameworks: Xilinx DPU v1



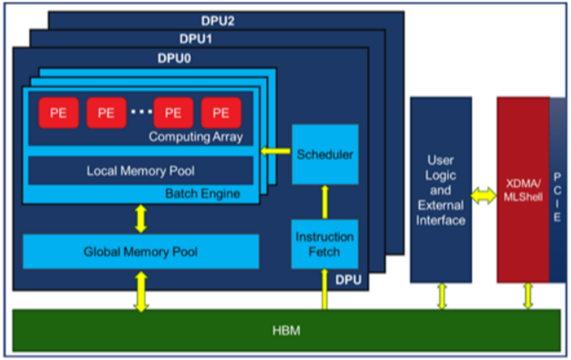
- 96x16 DSP Systolic Array

FPGA Frameworks: Xilinx DPU v2



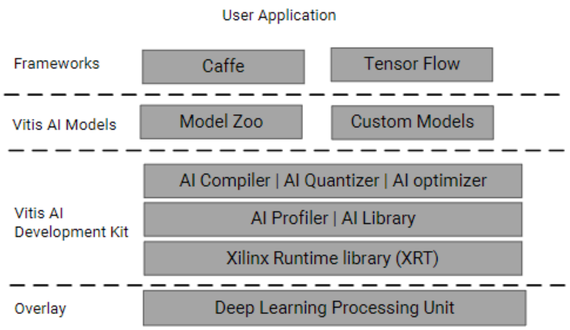
- Hybrid Computing Array
- Processing Elements - based on fine grained building blocks (multipliers, adders, accumulators)
- Deep Pipeline

FPGA Frameworks: Xilinx DPU v3



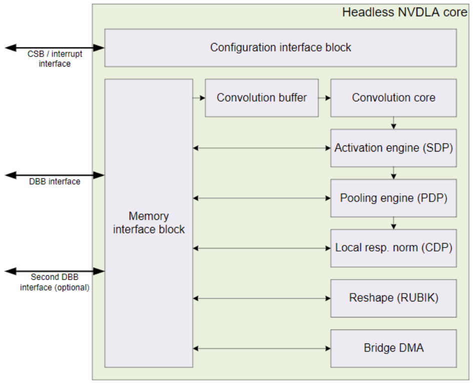
- Multiple Batch Engines
- Multiple DPU Cores

FPGA Frameworks: Xilinx Vitis AI



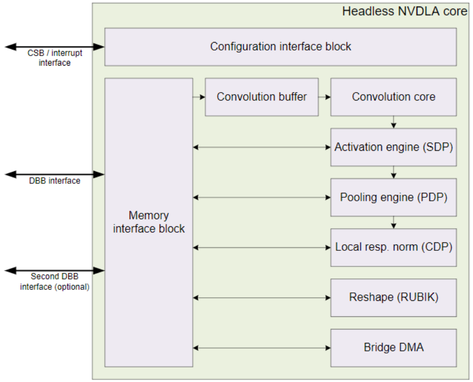
- Released in December 2019
- High-level abstraction
- AI inference applications
- Based on Xilinx DPU
- Optimized IPs, tools & libraries
- PetaLinux
- Instruction optimization - Vitis AI Compiler
- Vitis AI Quantizer - 8-bit fixed point parameters

FPGA Frameworks: NVIDIA Deep Learning Accelerator (NVDLA)



- Released in Q3 2017
- Free & Open architecture
- Goal to standardize inference DL accelerator development
- Headless implementation: Manager is the main system processor
- Headed implementation: Manager is a companion microcontroller
- Modular & Highly customizable
- Suitable for both FPGAs & ASICs

FPGA Frameworks: NVIDIA Deep Learning Accelerator (NVDLA)



- Binary and 4-bit integer up to 64-bit floating-point
- Convolution core
- Single Data processor
- Planar Data processor
- Channel Data processor
- Data Reshape Engine
- Memory-to-Memory or Pass-Through

Theoretical Modeling & Robustness Analysis

PyTorch, C/C++, MATLAB

PyTorch $\longrightarrow$ pure Python $\longrightarrow$ C/C++ & MATLAB

- Replicate PyTorch functionality
- Evaluation using PyTorch
- PyTorch pre-build & pretrained AlexNet as a reference
- 2500 images of Kaggle cats & dogs database
- Better understanding of underlining algorithms
- Explore Hardware implementation opportunities
- Quantization techniques $\longrightarrow$ reduce memory footprint
- Algorithmic optimizations
- Minimize hardware resources and optimize performance using various tools

Algorithms

- Convolution Layer
- Max-Pooling Layer
- Fully-Connected Layer
- ReLU
- SoftMax

Memory Footprint

- Classic hardware architectures $\longrightarrow$ Compute bound
- ASICs & FPGAs $\longrightarrow$ Memory bound
- Highest benefit: Memory requirements fit into BRAM (order of MBs)
- Otherwise, external memory used $\longrightarrow$ latency & IO stalls
- Goal: Minimize memory footprint & bandwidth

Memory Footprint: AlexNet Parameters (float32)

Layer	**#Parameters**	**Footprint**	**Memory (%)**
Conv1	$64 * 3 * 11 * 11 = 23232$	92.92KB	0.04
Conv2	$192 * 64 * 5 * 5 = 307200$	1.22MB	0.5
Conv3	$384 * 192 * 3 * 3 = 663552$	2.65MB	1.09
Conv4	$256 * 384 * 3 * 3 = 884736$	3.53MB	1.45
Conv5	$256 * 256 * 3 * 3 = 589824$	2.35MB	0.97
FC1	$9216 * 4096 = 37748736$	150.99MB	61.79
FC2	$4096 * 4096 = 16777216$	67.10MB	27.46
FC3	$4096 * 1000 = 4096000$	16.38MB	6.70
Total	61090496	244.36MB	100

Memory Footprint: AlexNet Data Stages (float32)

Layer	**#Data**	**Footprint**	**Memory (%)**
Image	$3 * 224 * 224 = 150528$	150.52KB	6.07
Conv1	$64 * 55 * 55 = 193600$	774.40KB	31.22
MaxPool1	$64 * 27 * 27 = 46656$	186.62KB	7.52
Conv2	$192 * 27 * 27 = 139968$	559.87KB	22.57
MaxPool2	$192 * 13 * 13 = 32448$	129.79KB	5.23
Conv3	$384 * 13 * 13 = 64896$	259.58KB	10.46
Conv4	$256 * 13 * 13 = 43264$	173.05KB	6.98
Conv5	$256 * 13 * 13 = 43264$	173.05KB	6.98
MaxPool3	9216	36.86KB	1.49
FC1	4096	16.38KB	0.66
FC2	4096	16.38KB	0.66
FC3	1000	4KB	0.16
Total	682856	2.48MB	100

Memory Footprint Reduction

- Data type bit-width shortening (float64-float16)
- Simpler data types (fixed-point/integers)
- Binary
- Quantization
- Quantization aware training
- Compression
- K-Means clustering
- Second Level Codebook

Memory Footprint Reduction

Trading accuracy to performance.

Memory Footprint Reduction: Evaluation

- Baseline: PyTorch pre-trained pre-build AlexNet model
- Inferencing 2500 pre-transformed Kaggle cats & dogs images
- Top-1 error rate
- MATLAB implementation used
- PyTorch & C/C++ do not support half-floating point

Memory Footprint Reduction: Floating Point

Convert 32-bit floats to their closest representation.

Tool	**Data type**	**Top-1 Error rate (%)**	**Avg. inference time (sec)**
PyTorch	float64	0	0.091
PyTorch	float32	0	0.034
MATLAB	float64	0	6.624
MATLAB	float32	0	8.162
MATLAB	float16	0.36	147.480

Memory Footprint Reduction: Fixed Point Parameters

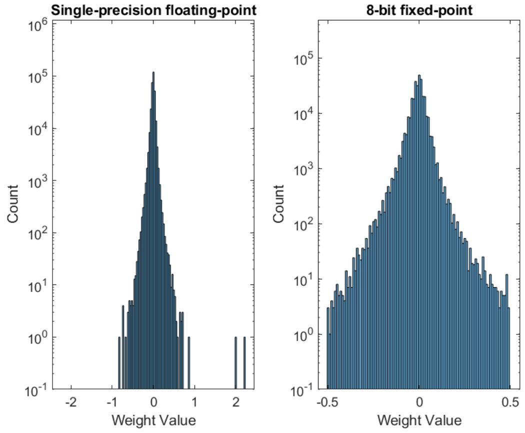
- Convert 32-bit float number sets to fixed-point
- Select best radix-point position to most accurately represent the number set
- Use same scale factor on whole set
- Every layer has its own scale factor

$$\mathrm{Position} = \mathrm{argmin}_{i=0}^{W}\left[\frac{\sum_{j=1}^{size(S)} |S_j - \mathrm{FixPtConvert}(S_j, W, i)|}{size(S)}\right]$$

Memory Footprint Reduction: Fixed Point Parameters

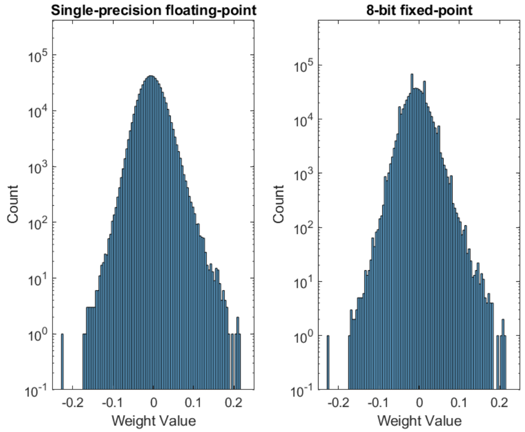
Tool	Data type	Top-1 Error rate (%)	Avg. inference time (sec)
MATLAB	fixed64	0	7.318
MATLAB	fixed32	0	7.692
MATLAB	fixed16	22	6.650
MATLAB	fixed14	28.44	6.813
MATLAB	fixed12	36.24	6.797
MATLAB	fixed10	77.07	6.929
MATLAB	fixed8	100	6.312

Memory Footprint Reduction: Fixed Point Parameters



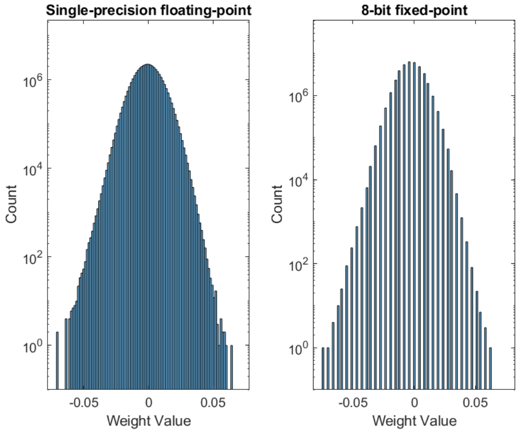
Histogram limits significantly altered

Memory Footprint Reduction: Fixed Point Parameters



Significant spiking

Memory Footprint Reduction: Fixed Point Parameters



Severe subsampling

Memory Footprint Reduction: Fixed Point Parameters

Use Mean Squared Error (MSE)

$$Position = argmin_{i=0}^{32}\left[\frac{\sum_{j=1}^{size(S)} |S_j - FixPtConvert(S_j, W, i)|^2}{size(S)}\right]$$

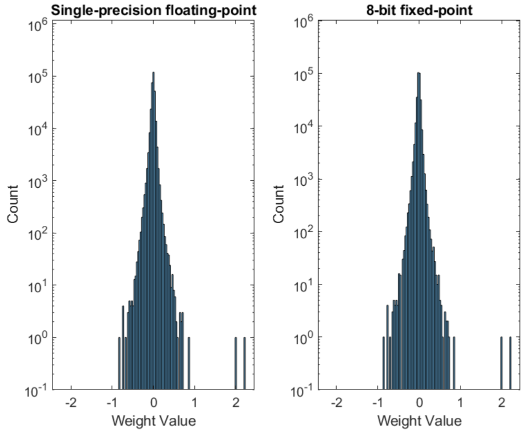
Use Mean Quarted Error (MQE)

$$Position = argmin_{i=0}^{32}\left[\frac{\sum_{j=1}^{size(S)} |S_j - FixPtConvert(S_j, W, i)|^4}{size(S)}\right]$$

Memory Footprint Reduction: Fixed Point MQE

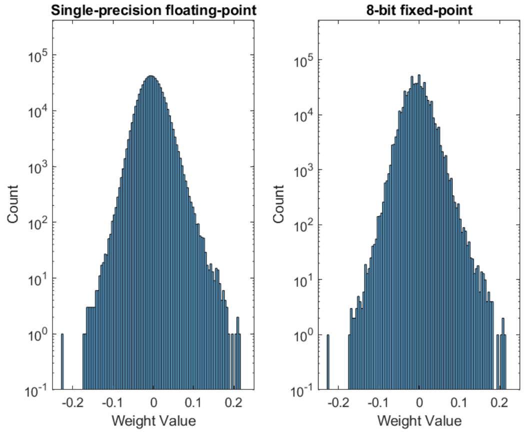
Tool	Data type	Top-1 Error rate (%)
MATLAB	fixed64	0
MATLAB	fixed32	0
MATLAB	fixed16	4.42
MATLAB	fixed14	17.59
MATLAB	fixed12	48.11
MATLAB	fixed10	86.91
MATLAB	fixed8	99.3

Memory Footprint Reduction: Fixed Point Parameters



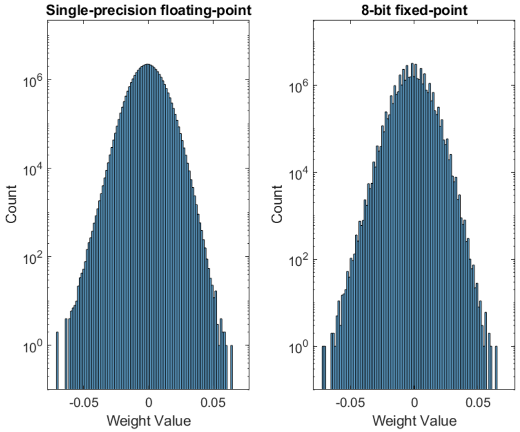
Histogram limits identical

Memory Footprint Reduction: Fixed Point Parameters



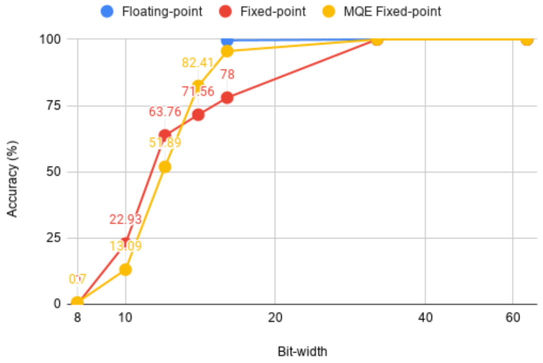
Slight spiking

Memory Footprint Reduction: Fixed Point Parameters



No subsampling, but spiking

Memory Footprint Reduction: All data types tested



- Accuracy degradation is expected
- Model dependent
- Training dependent

Memory Footprint Reduction: Fixed Point Activations

- Floating-point arithmetic scales automatically
- Fixed-point activations may overflow
- Quantize every layer outputs
- Keep the upper N most significant bits to retain accuracy
- Finding on runtime the uppermost on is computationally intensive $\longrightarrow$ any fixed-point benefits get obsolete

Memory Footprint Reduction: Fixed Point Activations

Calculate optimal activation scale factor per layer.

$$\mathrm{Theoretical}_{\mathrm{bitWidth}} = \mathrm{input}_{\mathrm{bitWidth}} + \mathrm{weight}_{\mathrm{bitWidth}} + \lceil \log_2 \#\mathrm{Additions} \rceil$$

Memory Footprint Reduction: Fixed Point Activations

Layer	Theoretical bit-width	Practical bit-width
Input	8	8
Conv1	25	17
Conv2	27	14
Conv3	27	15
Conv4	28	15
Conv5	28	17
FC1	30	17
FC2	28	17
FC3	28	17

- Theoretical worse case scenario significantly differs from practical
- Inference 2000 images, find maximum absolute valued activation per layer
- Maximum theoretical bit-width is 30-bits $\longrightarrow$ all activations fit in 32-bit integers before quantization

Memory Footprint Reduction: Fixed Point Activations

Layer	Weights	Bias	Output
Input	-7	-	-
Conv1	-7	-5	-2
Conv2	-5	-7	0
Conv3	-7	-7	3
Conv4	-8	-6	5
Conv5	-9	-5	10
FC1	-10	-10	15
FC2	-10	-9	19
FC3	-9	-9	23

Optimal scale factor per layer.

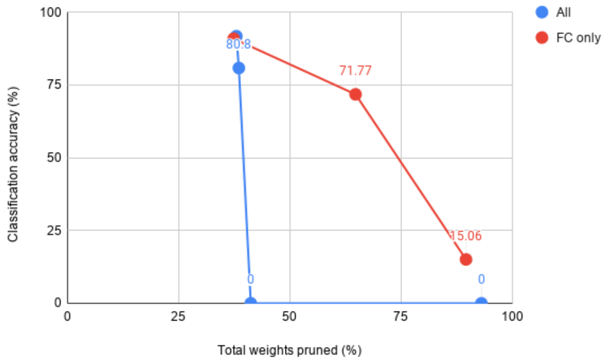
Weight Pruning

- Biological brain phase from birth until mid-20s
- Network compression
- Weak weights get pruned $\rightarrow w \epsilon [-\mathrm{f}, \mathrm{f}] = 0$, f: pruning factor
- Calculations can get skipped
- Higher memory & power efficiency & inference performance
- Accuracy-Performance tradeoff
- Weight pruning amount varies per network
- Global pruning factor: Not a good idea!

Weight Pruning

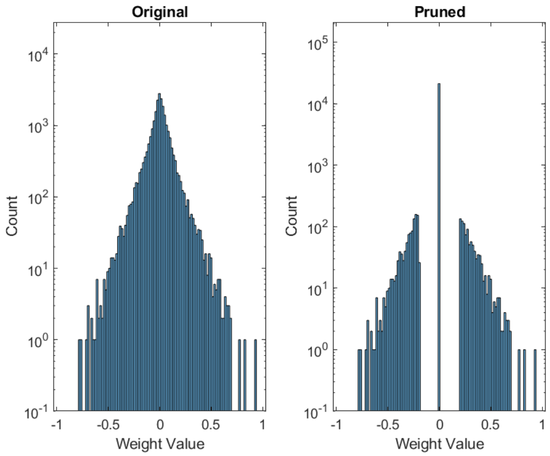
Layer	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7
Conv1 (%)	7.15	13.66	91.3	91.3	0	0	0
Conv2 (%)	13.82	26.9	95.83	95.83	0	0	0
Conv3 (%)	13.54	26.63	98.62	98.62	0	0	0
Conv4 (%)	15.32	29.99	93.14	93.14	0	0	0
Conv5 (%)	15.55	30.53	94.02	94.02	0	0	0
FC1 (%)	41.23	41.23	41.23	94.48	41.23	71.89	96.61
FC2 (%)	36.69	36.69	36.69	90.61	36.69	62.52	90.61
FC3 (%)	27.27	27.27	27.27	89.68	27.27	47.74	75.56
Total (%)	37.97	38.54	41.22	93.11	37.38	64.79	89.65
Accuracy (%)	91.74	80.8	0	0	90.87	71.77	15.06

Weight Pruning



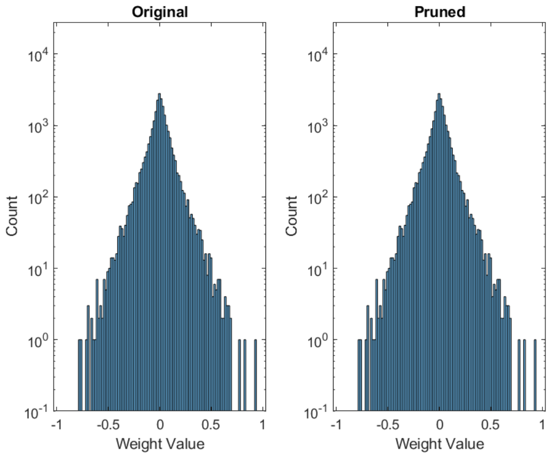
- Less Pruning $\longrightarrow$ Higher Accuracy
- Convolution layers more prone to error
- Pruning also denoises - Low valued weights act as noise

Weight Pruning



- Aggressive pruning
- High concentration of zeroes
- High compression factor
- Severe absence of near-to-zero valued weights

Weight Pruning



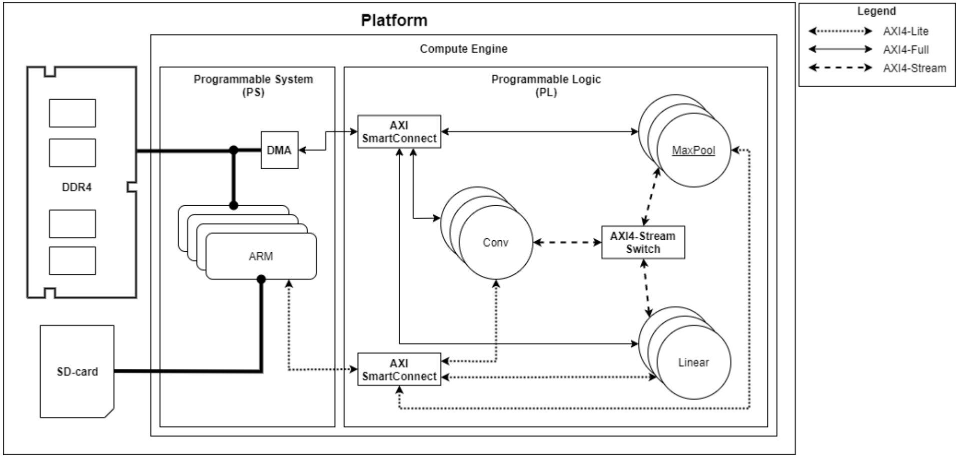
- Fine-tuned pruning
- Normal concentration of zeroes
- No discontinuation

Architecture Design

The platform

- Targeted for FPGAs
- Flexible & Versatile $\rightarrow$ easy transfer
- Scalable $\rightarrow$ multi-FPGA platforms
- Expandable $\rightarrow$ Easy adding of new layer types & accelerators
- Capable of running various CNN models
- Easy experimentation & development
- Minor to no code changes

Platform Block Diagram



Platform: Non-Volatile Memory

- Storage Medium
- Network model configurations
- Weights & Biases
- Class labels
- Input data, e.g. Images
- SD card - M.2 SSD (QFDB)
- External storage devices via Ethernet & JTAG

Platform: Volatile Memory

- Main system memory: DDR
- DDR loaded using PS part
- No global BRAM module
- Accelerators: BRAM caches
- Accelerators responsible to load their BRAM & locate data on DDR
- BRAM caches are private to their accelerator

Platform: Compute Engine

- Both PS & PL part utilized
- Bulk of computation on PL part through hardware accelerators
- Sophisticated work on PS part $\longrightarrow$ Initialization, Data loading, Input data preprocessing, accelerator configuration & scheduling
- Network layers both software & hardware
- Convolution, Max-Pooling & Fully-Connected accelerators
- Accelerator driver only knows how to configure it
- Driver is reusable

Platform: I/O

- Memory-Mapped I/O (MMIO)
- Streaming (AXI4-stream)
- BRAM MMIO - Not used

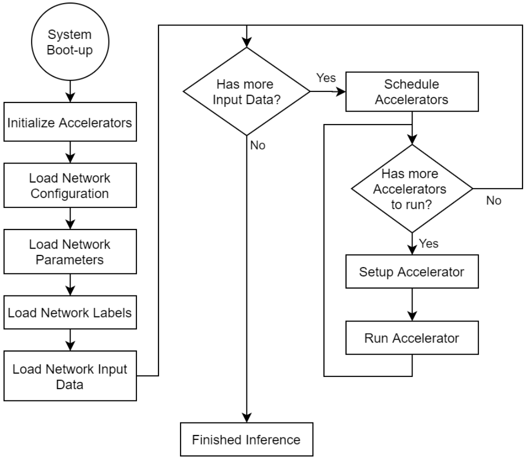
Port bit-width	MMIO avg. cycles	Streaming avg. cycles
32-bit	62700922	65611580
128-bit	15761270	16201797

40MB data - 40KB bursts

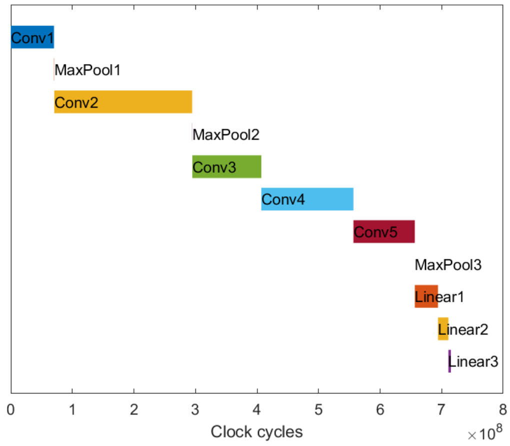
Platform: Software

- Accelerator drivers - Abstract form, every accelerator implements same functions
- Scheduler
- Application Logic
- User Interface

Platform: Software Flowchart

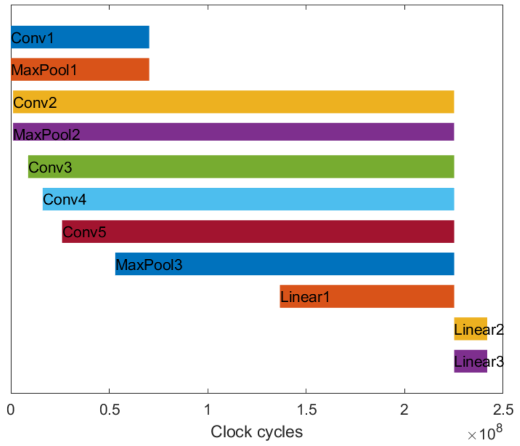


Platform: Serial Scheduler



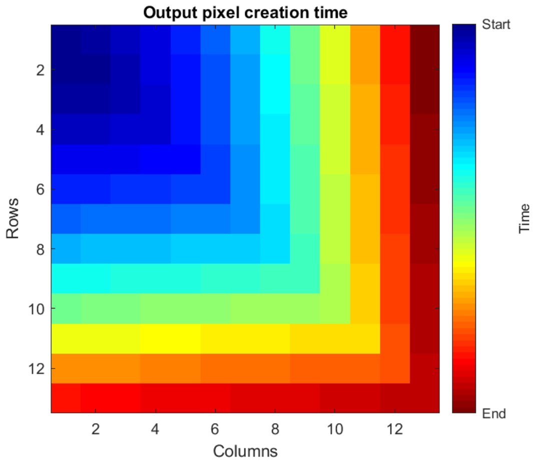
- Simple
- Best suited for debugging & validating
- About 90% of total inference time is consumed by convolution layers
- Possibility for deadlocks

Platform: Layer Pipelining Scheduler



- A layer gets its input as soon as its previous generates a single output
- Accelerator instances needed as many as there are in the model
- Almost 3x speedup
- Decreases latency & increases throughput
- Accelerators need to support pipelining
- Relatively complex

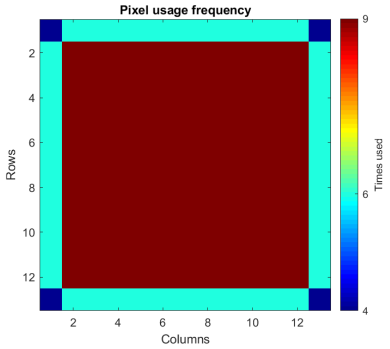
Output Pixel Creation Time

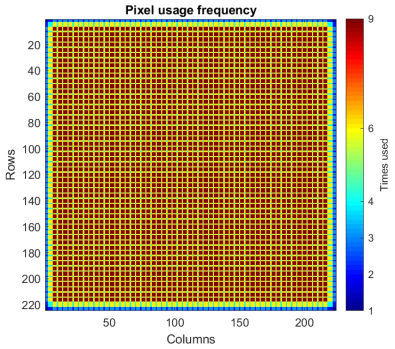


- Outputs generated in specific order to become useful inputs
- Convolution layers use cubes of inputs
- Max-Pooling layers use squares of inputs

Pixel Usage Frequency

Larger BRAM caches required





Platform: Multi-Inference Scheduler

- Multiple accelerator instances
- Multiple inferences in parallel
- Increases batch size & throughput
- Possibility for deadlocks

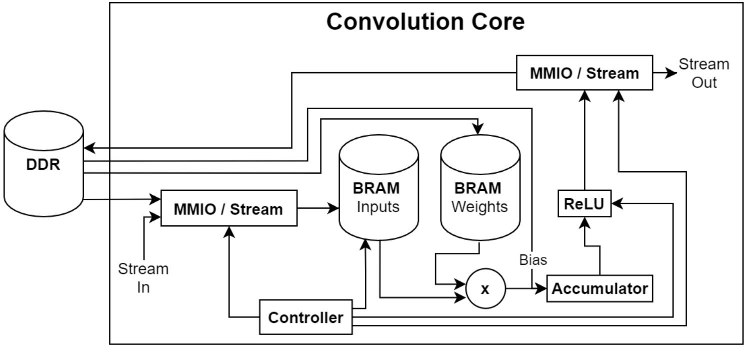
Platform: Image-Pipelining Scheduler

- Combination of Layer Pipelining & Multi-Inference
- Every layer handles a different input image
- Decreases latency & increases throughput
- Possibility for deadlocks

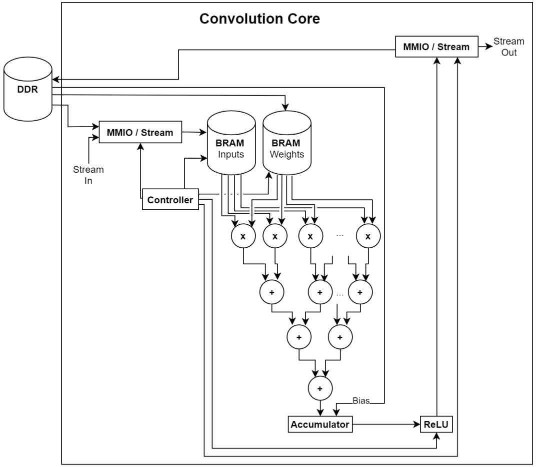
Accelerator Architectures

- Two versions: Simple serial & High performance
- Many have been tested
- Convolution layer
- Max-Pooling layer
- Fully-Connected layer

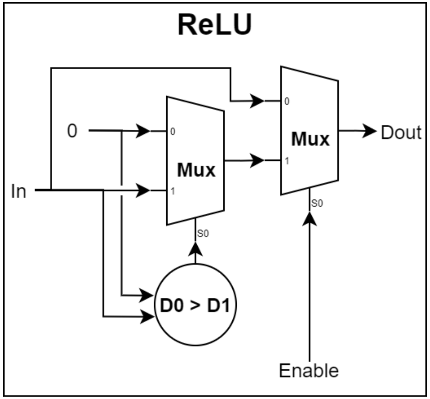
Convolution Accelerator



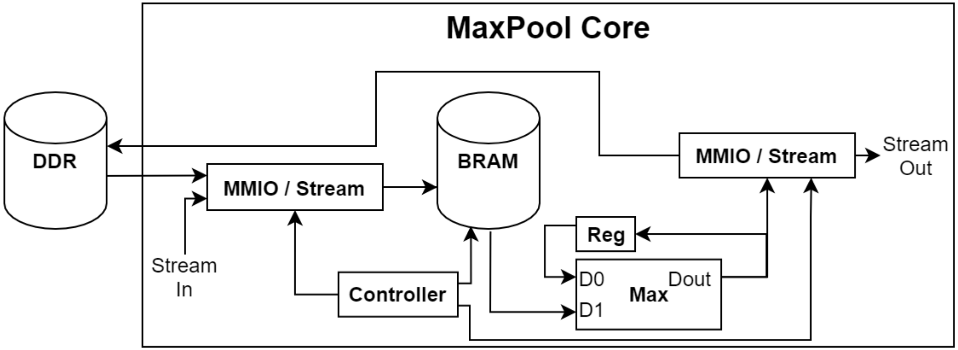
Convolution Accelerator



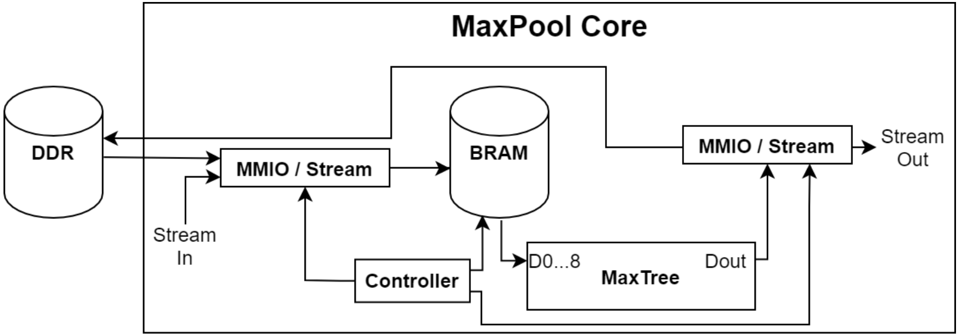
ReLU component



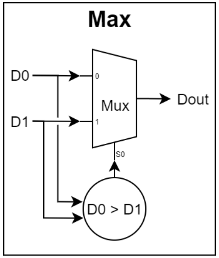
Max-Pooling Accelerator

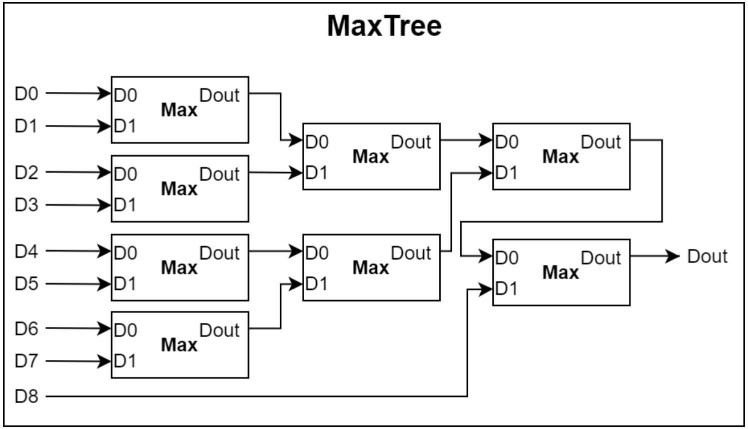


Max-Pooling Accelerator

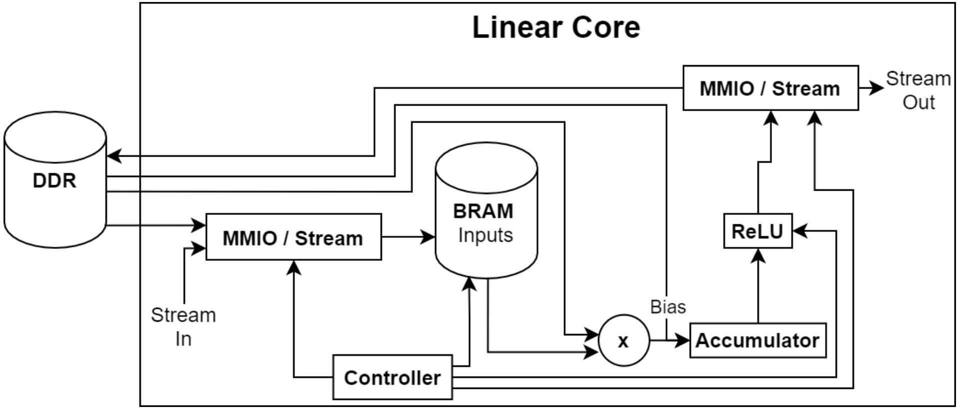


Max & Max-Tree components

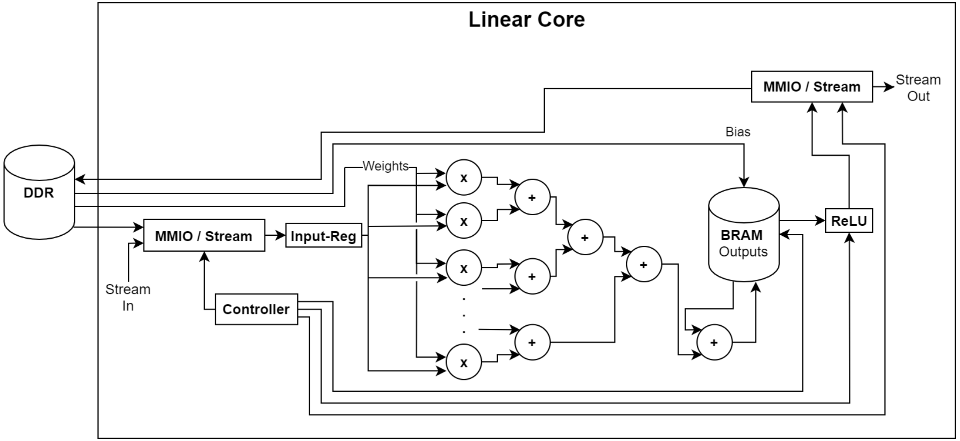




Fully-Connected Accelerator

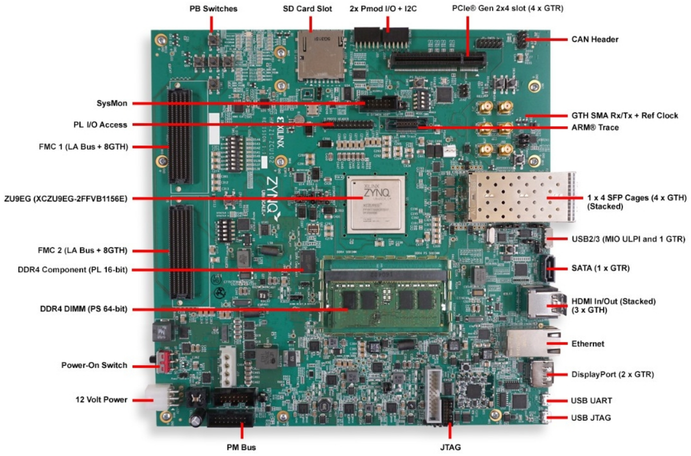


Fully-Connected Accelerator

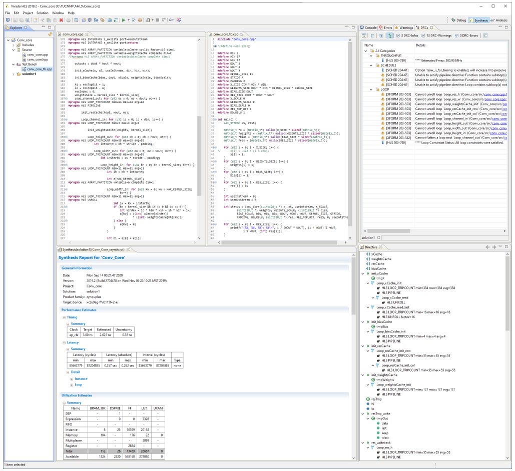


FPGA Implementation

Xilinx ZCU102 Evaluation Kit



Tools Used: Xilinx Vivado HLS



conv_core.cpp

```
#pragma HLS INTERFACE s_axilite port=useOutStream
#pragma HLS INTERFACE s_axilite port=return

#pragma HLS ARRAY_PARTITION variable=xCache cyclic factor=16 dim=1
#pragma HLS ARRAY_PARTITION variable=weightsCache complete dim=2
//#pragma HLS ARRAY_PARTITION variable=biasCache complete dim=1

    outputs = dout * hout * wout;

    init_xCache(x, xS, useInStream, din, hin, win);

    init_biasCache(bias, dout, xScale, weightsScale, biasScale);

    hi = resTopBit + 1;
    lo = resTopBit - 6;
    resIndex = 0;
    weightsSize = kernel_size * kernel_size;
    Loop_channel_out: for (u32 oc = 0; oc < dout; oc++) {
#pragma HLS LOOP_TRIPCOUNT min=64 max=64 avg=64
#pragma HLS PIPELINE

        init_resCache(hout, wout, oc);

        Loop_channel_in: for (u32 ic = 0; ic < din; ic++) {
#pragma HLS LOOP_TRIPCOUNT min=3 max=3 avg=3

            init_weightsCache(weights, kernel_size);

            Loop_height_out: for (u32 oh = 0; oh < hout; oh++) {
#pragma HLS LOOP_TRIPCOUNT min=55 max=55 avg=55
                int instartH = oh * stride - padding;

                Loop_width_out: for (u32 ow = 0; ow < wout; ow++) {
#pragma HLS LOOP_TRIPCOUNT min=55 max=55 avg=55
                    int instartW = ow * stride - padding;

                    Loop_height_in: for (u32 kh = 0; kh < kernel_size; kh++) {
#pragma HLS LOOP_TRIPCOUNT min=11 max=11 avg=11
                        int ih = kh + instartH;

                        int a[MAX_KERNEL_SIZE];
#pragma HLS ARRAY_PARTITION variable=a complete dim=1

                        Loop_width_in: for (u32 kw = 0; kw < MAX_KERNEL_SIZE;
                                kw++) {
#pragma HLS LOOP_TRIPCOUNT min=11 max=11 avg=11
#pragma HLS UNROLL
                            int iw = kw + instartW;
                            if (kw < kernel_size && ih >= 0 && iw >= 0) {
                                int xIndex = ic * hin * win + ih * win + iw;
                                a[kw] = ((int) xCache[xIndex])
                                        * ((int) weightsCache[kh][kw]);
                            } else {
                                a[kw] = 0;
                            }
                        }

                        int b1 = a[0] + a[1];
```

conv_core_tb.cpp

```
#include "conv_core.hpp"

//#define MASK 0xFF

#define DIN 3
#define HIN 17
#define WIN 17
#define DOUT 2
#define HOUT 3
#define WOUT 3
#define KERNEL_SIZE 11
#define STRIDE 4
#define PADDING 2
#define X_SIZE DIN * HIN * WIN
#define WEIGHTS_SIZE DOUT * DIN * KERNEL_SIZE * KERNEL_SIZE
#define BIAS_SIZE DOUT
#define RES_SIZE DOUT * HOUT * WOUT
#define X_SCALE 0
#define WEIGHTS_SCALE 0
#define BIAS_SCALE 0
#define RES_TOP_BIT 8
#define DO_RELU 1

int main() {
    AXI_STREAM xS, resS;

    matrix_t *x = (matrix_t*) malloc(X_SIZE * sizeof(matrix_t));
    matrix_t *weights = (matrix_t*) malloc(WEIGHTS_SIZE * sizeof(matrix_t));
    matrix_t *bias = (matrix_t*) malloc(BIAS_SIZE * sizeof(matrix_t));
    matrix_t *res = (matrix_t*) malloc(RES_SIZE * sizeof(matrix_t));

    for (u32 i = 0; i < X_SIZE; i++) {
//      x[i] = -128 + (i % 256);
        x[i] = 1;
    }
    for (u32 i = 0; i < WEIGHTS_SIZE; i++) {
        weights[i] = 1;
    }
    for (u32 i = 0; i < BIAS_SIZE; i++) {
        bias[i] = 1;
    }
    for (u32 i = 0; i < RES_SIZE; i++) {
        res[i] = 0;
    }

    int useInStream = 0;
    int useOutStream = 0;

    int status = Conv_Core((uint128_t *) x, xS, useInStream, X_SCALE,
            (uint128_t *) weights, WEIGHTS_SCALE, (uint128_t *) bias,
            BIAS_SCALE, DIN, HIN, WIN, DOUT, HOUT, WOUT, KERNEL_SIZE, STRIDE,
            PADDING, DO_RELU, (uint128_t *) res, RES_TOP_BIT, resS, 0, useOutStre

    for (u32 i = 0; i < RES_SIZE; i++) {
        printf("(%d, %d, %d): %d\n", i / (HOUT * WOUT), (i / WOUT) % HOUT,
                i % WOUT, (int) res[i]);
    }
```

Console | Errors | Warnings | DRCs — ☑ 3 DRC-Infos ☑ 13 DRC-Warnings ☑ 0 DRC-Errors

Name	Details
All Categories	
THROUGHPUT	
[HLS 200-789]	**** Estimated Fmax: 380.95 MHz
SCHEDULE	
[SCHED 204-61]	Option 'relax_ii_for_timing' is enabled, will increase II to preserve
[SCHED 204-65]	Unable to satisfy pipeline directive: Function contains subloop(s)
[SCHED 204-65]	Unable to satisfy pipeline directive: Function contains subloop(s)
[SCHED 204-65]	Unable to satisfy pipeline directive: Loop contains subloop(s) not
LOOP	
[XFORM 203-503]	Cannot unroll loop 'Loop_res_h' (Conv_core/src/conv_core.cpp:
[XFORM 203-503]	Cannot unroll loop 'Loop_res_w' (Conv_core/src/conv_core.cpp:
[XFORM 203-503]	Cannot unroll loop 'Loop_weightsCache_init' (Conv_core/src/co
[XFORM 203-503]	Cannot unroll loop 'Loop_resCache_init_row' (Conv_core/src/co
[XFORM 203-503]	Cannot unroll loop 'Loop_resCache_init_col' (Conv_core/src/con
[XFORM 203-503]	Cannot unroll loop 'Loop_channel_in' (Conv_core/src/conv_core
[XFORM 203-503]	Cannot unroll loop 'Loop_height_out' (Conv_core/src/conv_core
[XFORM 203-503]	Cannot unroll loop 'Loop_width_out' (Conv_core/src/conv_core.
[XFORM 203-503]	Cannot unroll loop 'Loop_height_in' (Conv_core/src/conv_core.c
[XFORM 203-542]	Cannot flatten a loop nest 'Loop_channel_out' (Conv_core/src/c
[HLS 200-790]	**** Loop Constraint Status: All loop constraints were satisfied.

Synthesis(solution1)(Conv_Core_csynth.rpt)

Synthesis Report for 'Conv_Core'

General Information

Date:	Mon Sep 14 00:21:47 2020
Version:	2019.2 (Build 2704478 on Wed Nov 06 22:10:23 MST 2019)
Project:	Conv_core
Solution:	solution1
Product family:	zynquplus
Target device:	xczu9eg-ffvb1156-2-e

Performance Estimates

Timing — Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.00 ns	2.625 ns	0.38 ns

Latency — Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
85663779	87204885	0.257 sec	0.262 sec	85663779	87204885	none

Detail: Instance, Loop

Utilization Estimates

Summary

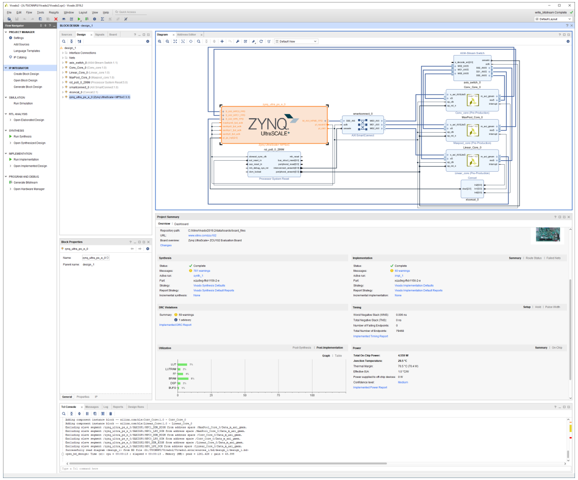
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	0	0	3398	-
FIFO	-	-	-	-	-
Instance	8	25	10399	20158	-
Memory	104	-	176	22	0
Multiplexer	-	-	-	3089	-
Register	-	-	2884	-	-
Total	112	26	13459	26667	0
Available	1824	2520	548160	274080	0

Directive

```
[] xCache
[] weightsCache
[] resCache
[] biasCache
init_xCache
    tmpX
    Loop_xCache_init
        # HLS LOOP_TRIPCOUNT min=384 max=384 avg=384
        # HLS PIPELINE
        Loop_xCache_read
            # HLS UNROLL
    Loop_xCache_read_last
        # HLS LOOP_TRIPCOUNT min=16 max=16 avg=16
        # HLS UNROLL factor=16
init_biasCache
    tmpBias
    Loop_biasCache_init
        # HLS LOOP_TRIPCOUNT min=4 max=4 avg=4
        # HLS PIPELINE
init_resCache
    Loop_resCache_init_row
        # HLS LOOP_TRIPCOUNT min=55 max=55 avg=55
        # HLS PIPELINE
        Loop_resCache_init_col
            # HLS LOOP_TRIPCOUNT min=55 max=55 avg=55
init_weightsCache
    tmpWeights
    Loop_weightsCache_init
        # HLS LOOP_TRIPCOUNT min=121 max=121 avg=121
        # HLS PIPELINE
resTmp
hi
lo
resTmp_write
    tmpOut
        data
        last
        keep
        tdest
res_writeback
    Loop_res_h
        # HLS LOOP_TRIPCOUNT min=55 max=55 avg=55
        # HLS PIPELINE
```

- Now Vitis HLS
- High-level design using C/C++, SystemC, OpenCL
- Generates VHDL & Verilog HDL designs
- Directives
- C/C++ testbench
- C/RTL Cosimulation
- Synthesis Report

Tools Used: Xilinx Vivado IDE



Project Summary

Repository path: C:/Xilinx/Vivado/2019.2/data/boards/board_files
URL: www.xilinx.com/zcu102
Board overview: Zynq UltraScale+ ZCU102 Evaluation Board

Synthesis

Status:	Complete
Messages:	781 warnings
Active run:	synth_1
Part:	xczu9eg-ffvb1156-2-e
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Implementation

Status:	Complete
Messages:	60 warnings
Active run:	impl_1
Part:	xczu9eg-ffvb1156-2-e
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Summary: 59 warnings, 1 advisory

Timing

Worst Negative Slack (WNS):	0.006 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	79468

Power

Total On-Chip Power:	4.559 W
Junction Temperature:	29.5 °C
Thermal Margin:	70.5 °C (70.4 W)
Effective θJA:	1.0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

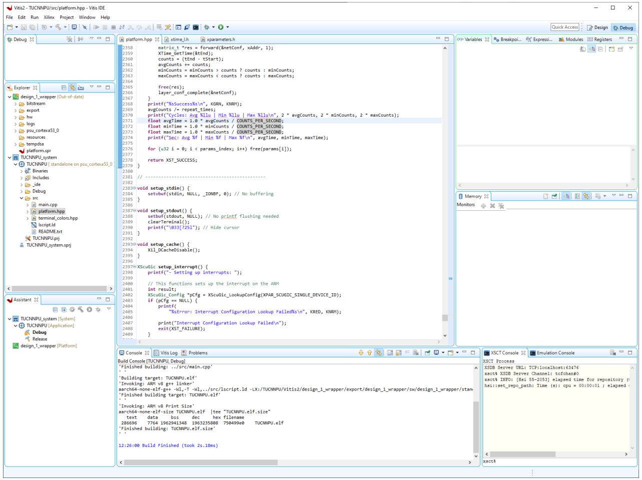
Utilization (Post-Implementation)

LUT	7%
LUTRAM	2%
FF	4%
BRAM	8%
DSP	2%
BUFG	1%

```
Adding component instance block -- xilinx.com:hls:Conv_Core:1.0 - Conv_Core_0
Adding component instance block -- xilinx.com:hls:Linear_Core:1.0 - Linear_Core_0
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP1/HPC1_DDR_HIGH from address space /MaxPool_Core_0/Data_m_axi_gmem.
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP1/HPC1_LPS_OCM from address space /MaxPool_Core_0/Data_m_axi_gmem.
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP0/HPC0_DDR_HIGH from address space /Conv_Core_0/Data_m_axi_gmem.
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP0/HPC0_LPS_OCM from address space /Conv_Core_0/Data_m_axi_gmem.
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP2/HP0_DDR_HIGH from address space /Linear_Core_0/Data_m_axi_gmem.
Excluding slave segment /zynq_ultra_ps_e_0/SAXIGP2/HP0_LPS_OCM from address space /Linear_Core_0/Data_m_axi_gmem.
Successfully read diagram <design_1> from BD file <X:/TUCNNPU/Vivado2/Vivado2.srcs/sources_1/bd/design_1/design_1.bd>
open_bd_design: Time (s): cpu = 00:00:13 ; elapsed = 00:00:13 . Memory (MB): peak = 1261.426 ; gain = 43.398
```

- VHDL & Verilog
- IP Integrator Tool
- Vivado HLS RTL designs
- Synthesis, Implementation & Download RTL designs
- RTL Simulators & Integrated Logic Analyzer IPs

Tools Used: Xilinx SDK/Vitis IDE



```
        matrix_t *res = forward(&netConf, xAddr, 1);
        XTime_GetTime(&tEnd);
        counts = (tEnd - tStart);
        avgCounts += counts;
        minCounts = minCounts > counts ? counts : minCounts;
        maxCounts = maxCounts < counts ? counts : maxCounts;

        free(res);
        layer_conf_complete(&netConf);
    }
    printf("%sSuccess%s\n", KGRN, KNRM);
    avgCounts /= repeat_times;
    printf("Cycles: Avg %llu | Min %llu | Max %llu\n", 2 * avgCounts, 2 * minCounts, 2 * maxCounts);
    float avgTime = 1.0 * avgCounts / COUNTS_PER_SECOND;
    float minTime = 1.0 * minCounts / COUNTS_PER_SECOND;
    float maxTime = 1.0 * maxCounts / COUNTS_PER_SECOND;
    printf("Sec: Avg %f | Min %f | Max %f\n", avgTime, minTime, maxTime);

    for (u32 i = 0; i < params_index; i++) free(params[i]);

    return XST_SUCCESS;
}

// ---------------------------------------------

void setup_stdin() {
    setvbuf(stdin, NULL, _IONBF, 0); // No buffering
}

void setup_stdout() {
    setbuf(stdout, NULL); // No printf flushing needed
    clearTerminal();
    printf("\033[?25l"); // Hide cursor
}

void setup_cache() {
    Xil_DCacheDisable();
}

XScuGic setup_interrupt() {
    printf("- Setting up interrupts: ");

    // This functions sets up the interrupt on the ARM
    int result;
    XScuGic_Config *pCfg = XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
    if (pCfg == NULL) {
        printf(
            "%sError: Interrupt Configuration Lookup Failed%s\n", KRED, KNRM);

        print("Interrupt Configuration Lookup Failed\n");
        exit(XST_FAILURE);
    }
```

Build Console [TUCNNPU, Debug]

```
'Finished building: ../src/main.cpp'
' '
'Building target: TUCNNPU.elf'
'Invoking: ARM v8 g++ linker'
aarch64-none-elf-g++ -Wl,-T -Wl,../src/lscript.ld -LX:/TUCNNPU/Vitis2/design_1_wrapper/export/design_1_wrapper/sw/design_1_wrapper/stan
'Finished building target: TUCNNPU.elf'
' '
'Invoking: ARM v8 Print Size'
aarch64-none-elf-size TUCNNPU.elf  |tee "TUCNNPU.elf.size"
   text    data     bss     dec     hex filename
 286696    7764 1962941348 1963235808 750499e0    TUCNNPU.elf
'Finished building: TUCNNPU.elf.size'
' '


12:26:00 Build Finished (took 2s.18ms)
```

XSCT Process

```
XSDB Server URL: TCP:localhost:63476
xsct% XSDB Server Channel: tcfchan#0
xsct% INFO: [Hsi 55-2053] elapsed time for repository (
hsi::set_repo_path: Time (s): cpu = 00:00:01 ; elapsed 
xsct%
```

- Vitis IDE integrates SDK, SDAccel, SDSoC tools
- C/C++ IDE
- Application development for PS part
- PetaLinux & FreeRTOS
- Download bitstreams
- Debugging tools

Results

Compared Platforms: CPU

Intel i7 4710MQ

Cores / Threads	4/8
Max Turbo Frequency	3.5GHz
TDP	47W
Max Memory Bandwidth	25.6GB/s
Lithography	22nm

Compared Platforms: GPU

NVIDIA RTX-2060 Super 8GB

CUDA Cores	2176
Tensor Cores	32
GPU Memory	8GB GDDR6
Boost Clock	1650 MHz
Memory Interface	256-bit
Memory Bandwidth	448GB/s
Power Consumption	175W

Compared Platforms: FPGA

Xilinx CHaiDNN

PL/DSP Clock Frequency	250/500 MHz
LUT Usage	59.1%
FF Usage	27.66%
BRAM Usage	74.12%
DSP Usage	53.65%

Compared Platforms: FPGA

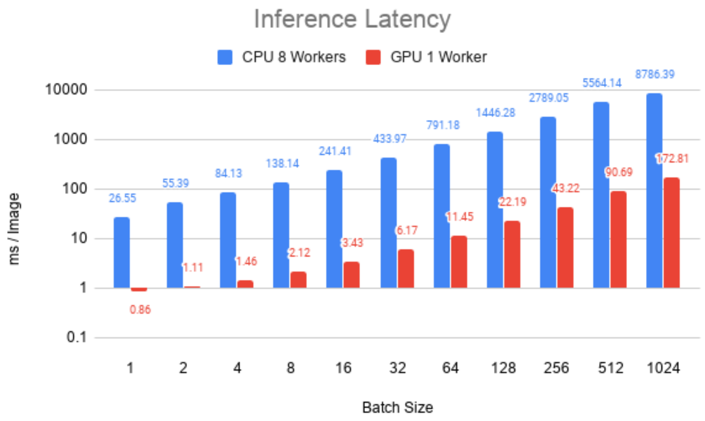
Proposed Platform

Clock Frequency (MHz)	300MHz
LUT Usage	7.34%
LUTRAM Usage	2.05%
FF Usage	4.03%
BRAM Usage	7.51%
DSP Usage	1.9%

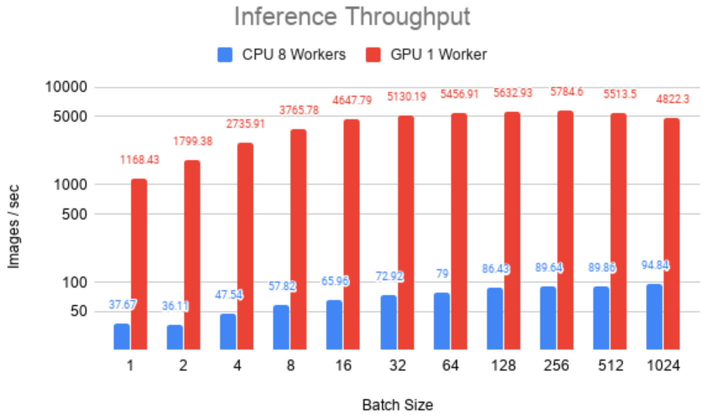
CPU & GPU Performance

- Inference 2500 images
- Use all worker & batch-size combinations
- PyTorch pre-built pre-trained AlexNet

CPU & GPU Performance: Latency



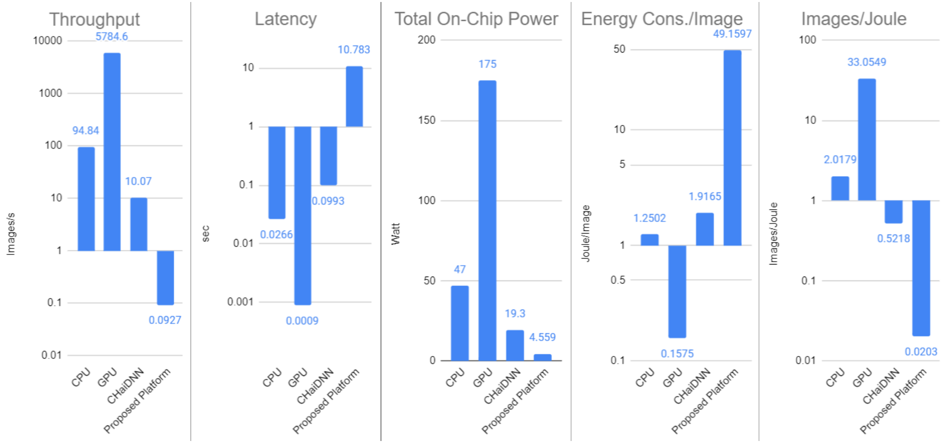
CPU & GPU Performance: Throughput



Final Performance

	CPU	GPU	CHaiDNN	Proposed Platform
Clock Frequency (MHz)	3500	1650	250/500	300
Throughput (Images/s)	94.84	5784.6	10.07	0.0927
Throughput Speedup	1x	60.9933x	0.1062x	0.001x
Latency (s)	0.0266	0.0009	0.0993	10.783
Latency Speedup	1x	29.5556x	0.2679x	0.0025x
Total On-Chip Power (Watt)	47	175	19.3	4.559
Power Efficiency	1x	0.2686x	2.4352x	10.3093x
Energy Cons./Image (Joule)	1.2502	0.1575	1.9165	49.1597
Energy Efficiency	1x	7.9378x	0.6523x	0.0254x
Images/Joule	2.0179	33.0549	0.5218	0.0203

Final Performance



Conclusions & Future Work

Conclusions

- Neural Networks need hardware acceleration
- Proposed platform provides an easy and structured methodology for scalable & expandable accelerator implementation
- Memory reduction is a necessity
- Further development $\longrightarrow$ higher performance

Future Work

- Quantization: better classification accuracy, K-Means clustering, Lloyd’s, Pair and Quad compression, and Second Level Codebook
- Integrating the pooling layer into the convolution layer
- Pruning enabled accelerators
- Systolic arrays as their main compute engine
- Multiple accelerator instances

Future Work

- Layer-Pipelining
- Bigger FPGA devices & multiple interconnected FPGAs (FORTH QFDB & CRDB).
- Monte Carlo Dropout for increased confidence of classification results
- Designs using VHDL & Verilog for resource & performance optimization
- CPU-FPGA partitioning

Thank You!

Any Questions?