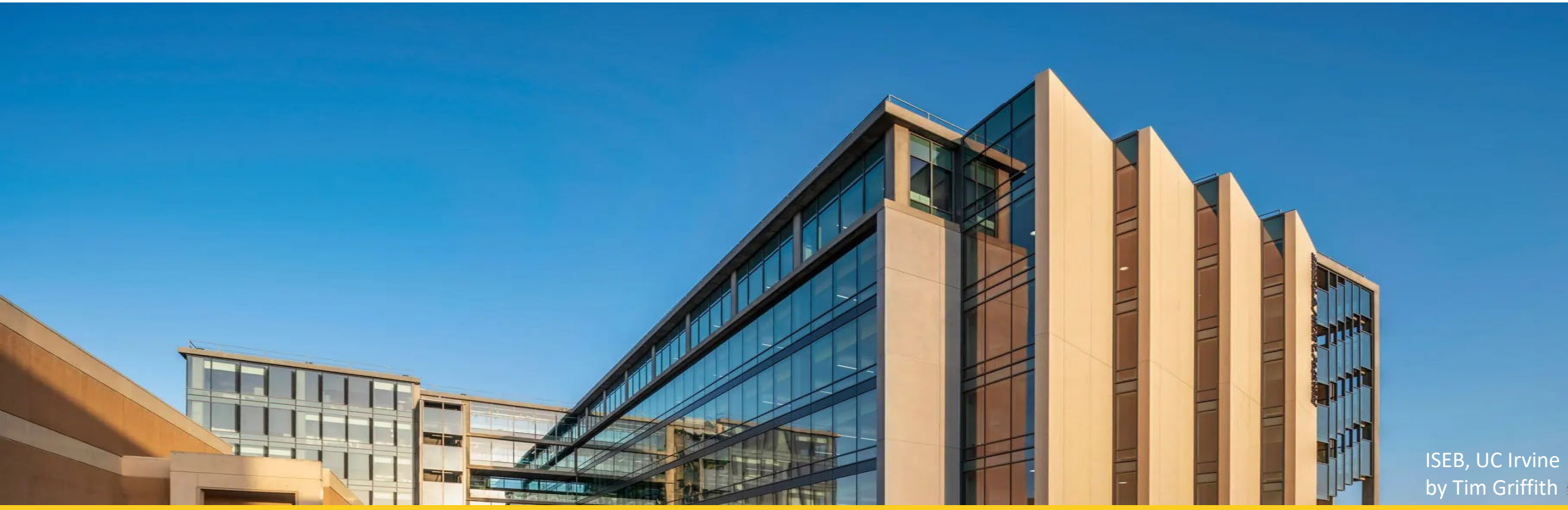
Lecture 6:

Deep Learning Accelerators

Sitao Huang, sitaoh@uci.edu

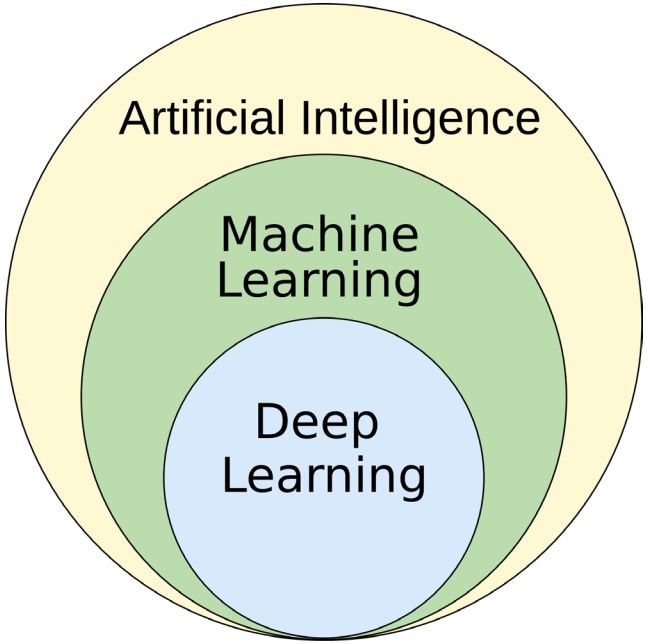
October 30, 2023

Sources: Prof. Deming Chen, UIUC ECE 527 “System-on-Chip Design”



ISEB, UC Irvine
by Tim Griffith

Machine Learning and Deep Learning



Machine Learning

- A data-oriented approach of building models
- Learns from data (training, update model parameters)
- Applied to unseen data (inference or testing, model makes predictions)
- Assume test data and training data follow same distribution
-

Deep Learning

- Using deep neural networks (DNNs) to build machine learning models

Why Machine Learning? Why Now?

Big Data

- Large unstructured data sets flood us everyday

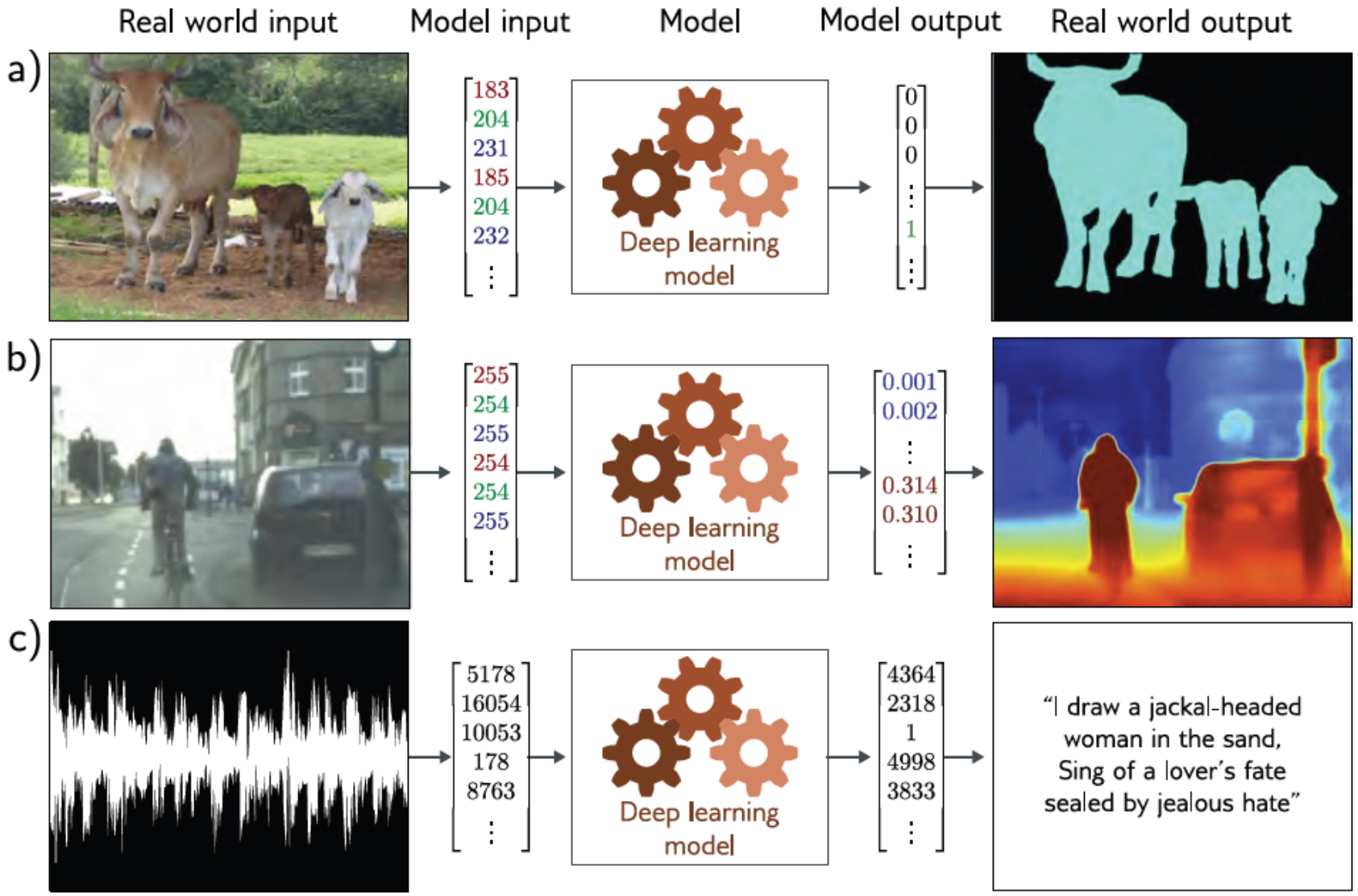
Data Science

- Extract knowledge/insight from data

Machine Learning

- For specific tasks, resembles human intelligence

Machine Learning



Overview Of The Machine Learning Process

- Training: Train the desired model, let machine learn intelligence
- Computationally intensive
- Huge amount of data, long training time
  - ImageNet: millions of training data, Weeks to train VGG16
- Platforms: GPUs, ASICs, etc.

- Inference: Infers things about new data based on its training
- Computationally intensive
- Real time -- Mobile device, IoTs
- Platforms: ASICs, GPUs, FPGAs, etc.

Classification

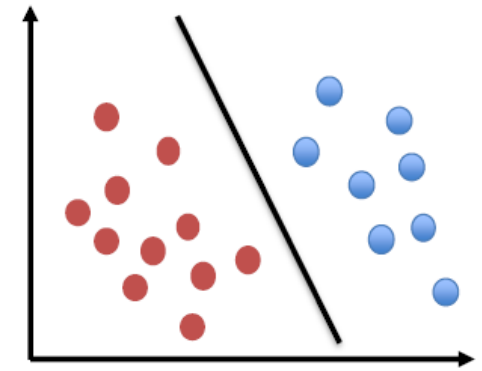
- Identify to which of a set of categories an observation belongs
  - Or rather, which category is the most dominant

Process overview

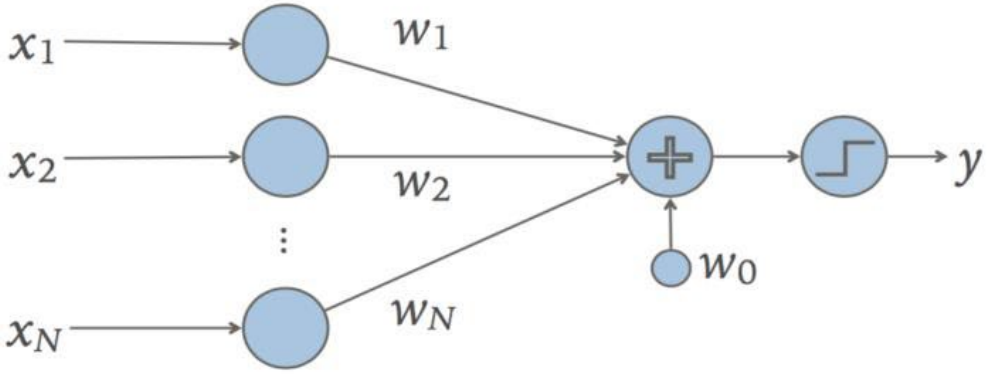
- Provide examples of classes
  - Training data (label)
- Make models for each class
  - Training process
- Assign each new input sample to a class
  - Classification

Linear Classifier

- Perceptron $y_i = sign(W^T \cdot X_i + b)$

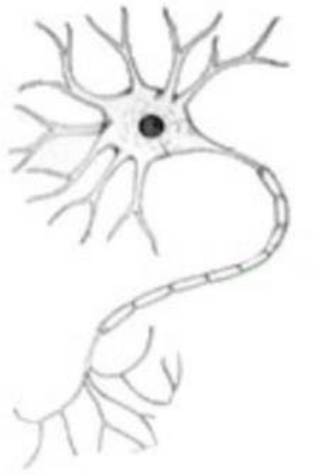


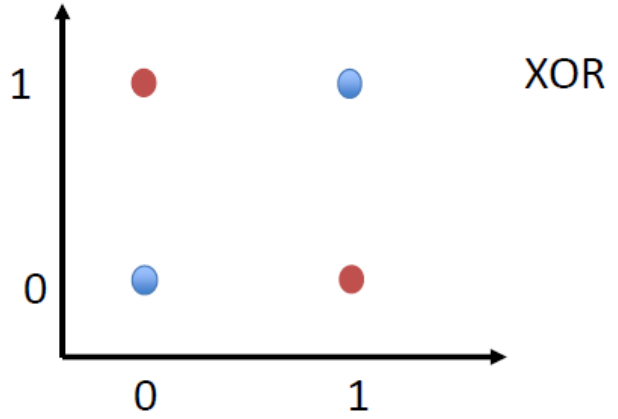
The perceptron



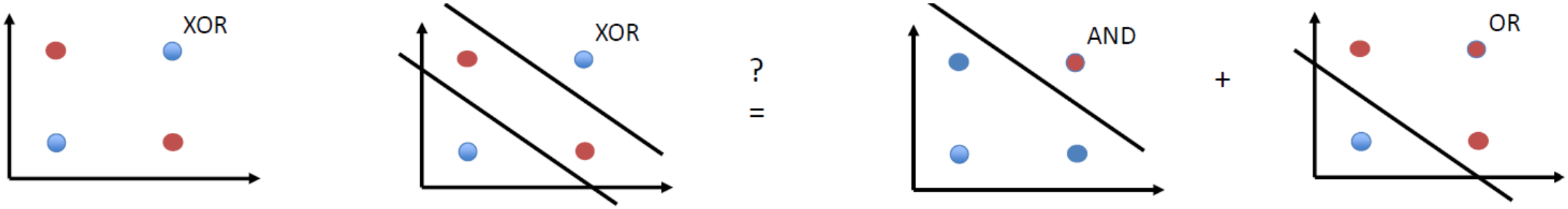
$\approx$

The neuron





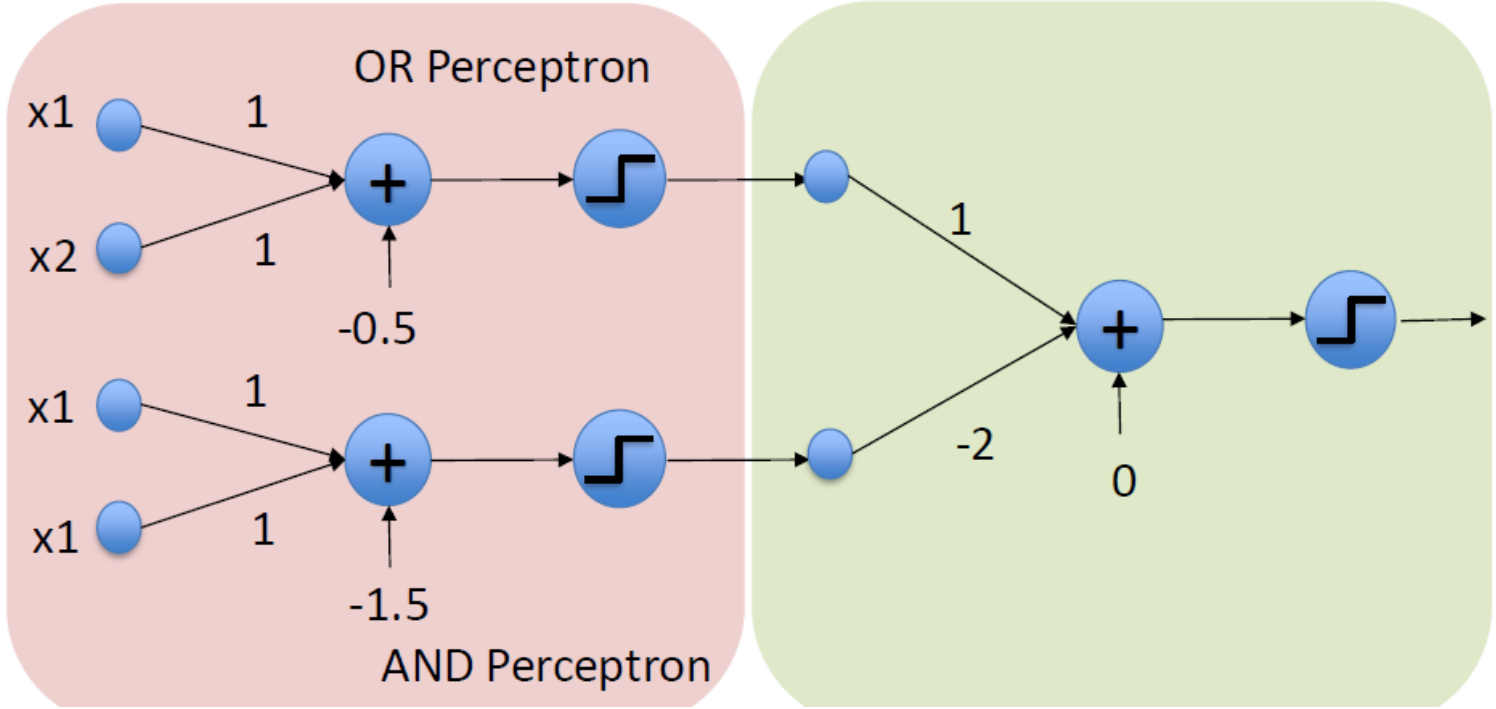
Multi-Layer Perceptron for Classification



A xor B = (A or B) – 2(A and B)

Nonlinear

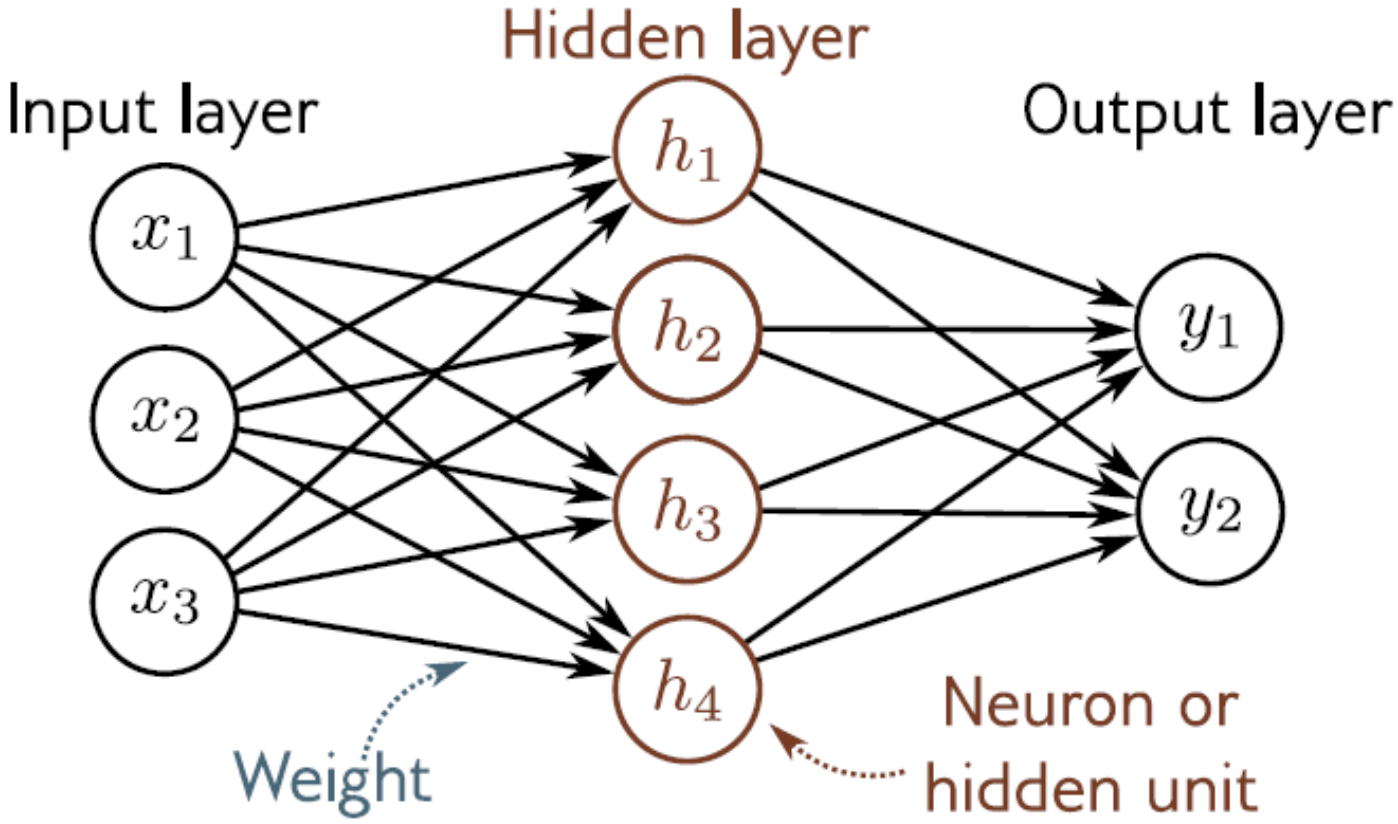
This is a Neural Network



Multi-layer Perceptron is a Neural Network

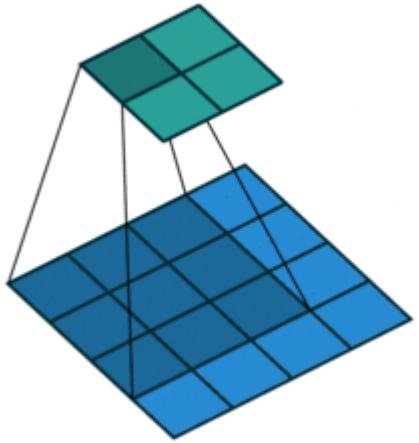
- Terminology:
- Input layer, Hidden layer(s), Output layer
- Deep Neural Network (DNN): more than one hidden layers

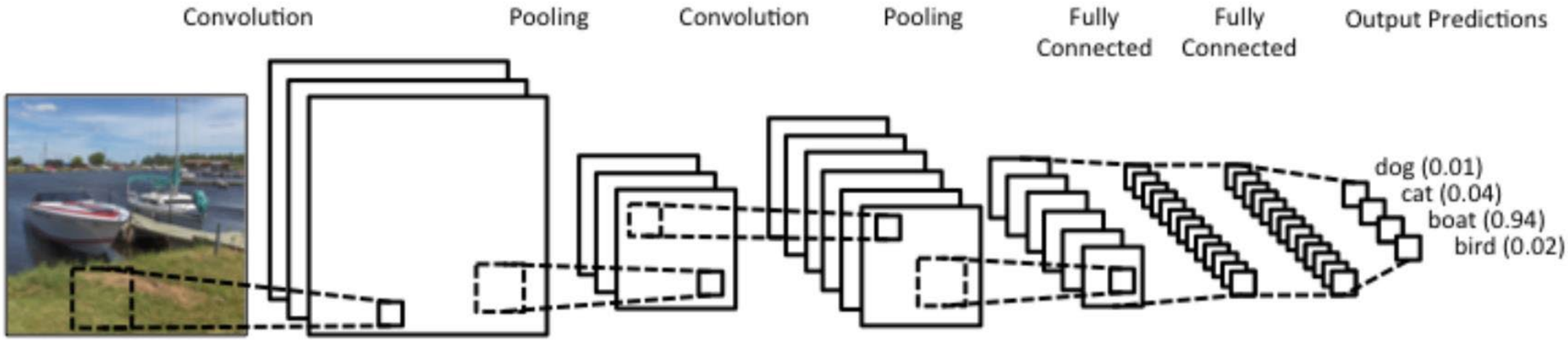
- X: Input feature, W: Weights (Filter), b: bias Y: Label



Convolutional Neural Network (CNN)

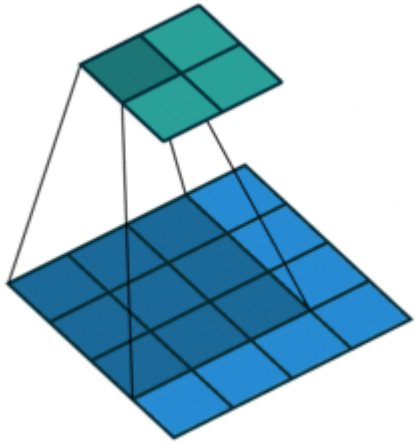
- Input is an image, 2D matrix (MLP is a vector)
- Linear function is convolution W⊛X (MLP is W X+b)
  - Learn each filter
- ▪ Advantage: Reduce number of weights, general better practice

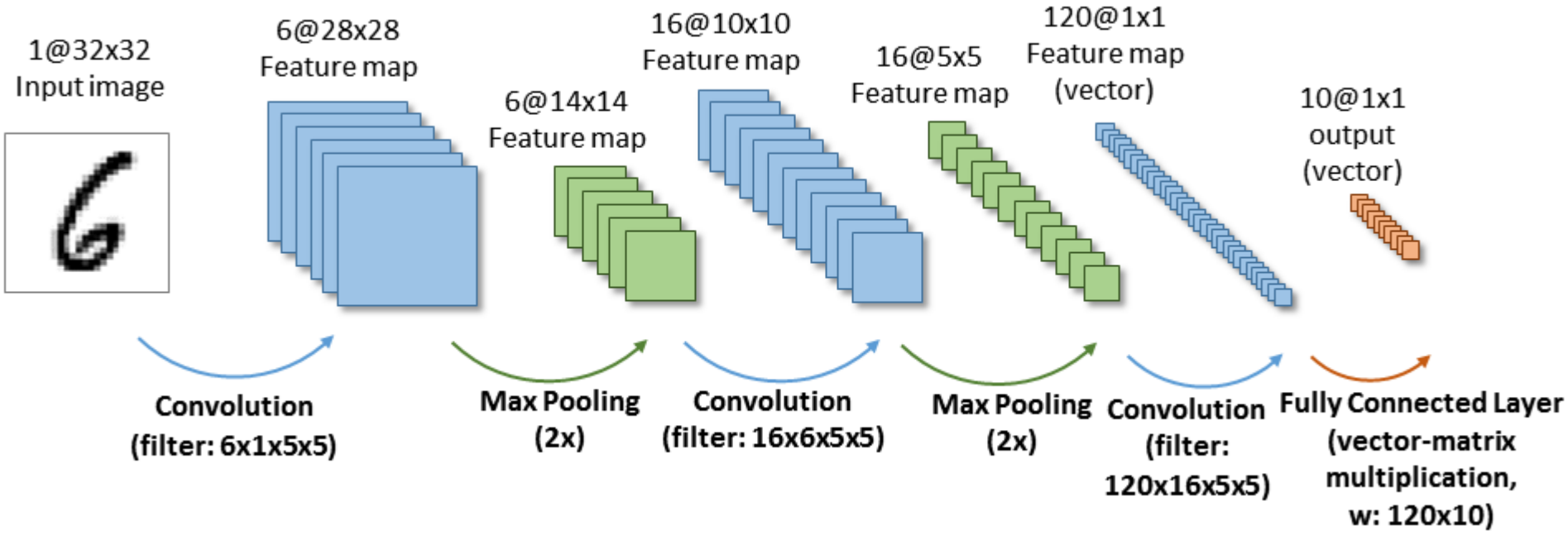




Convolutional Neural Network (CNN)

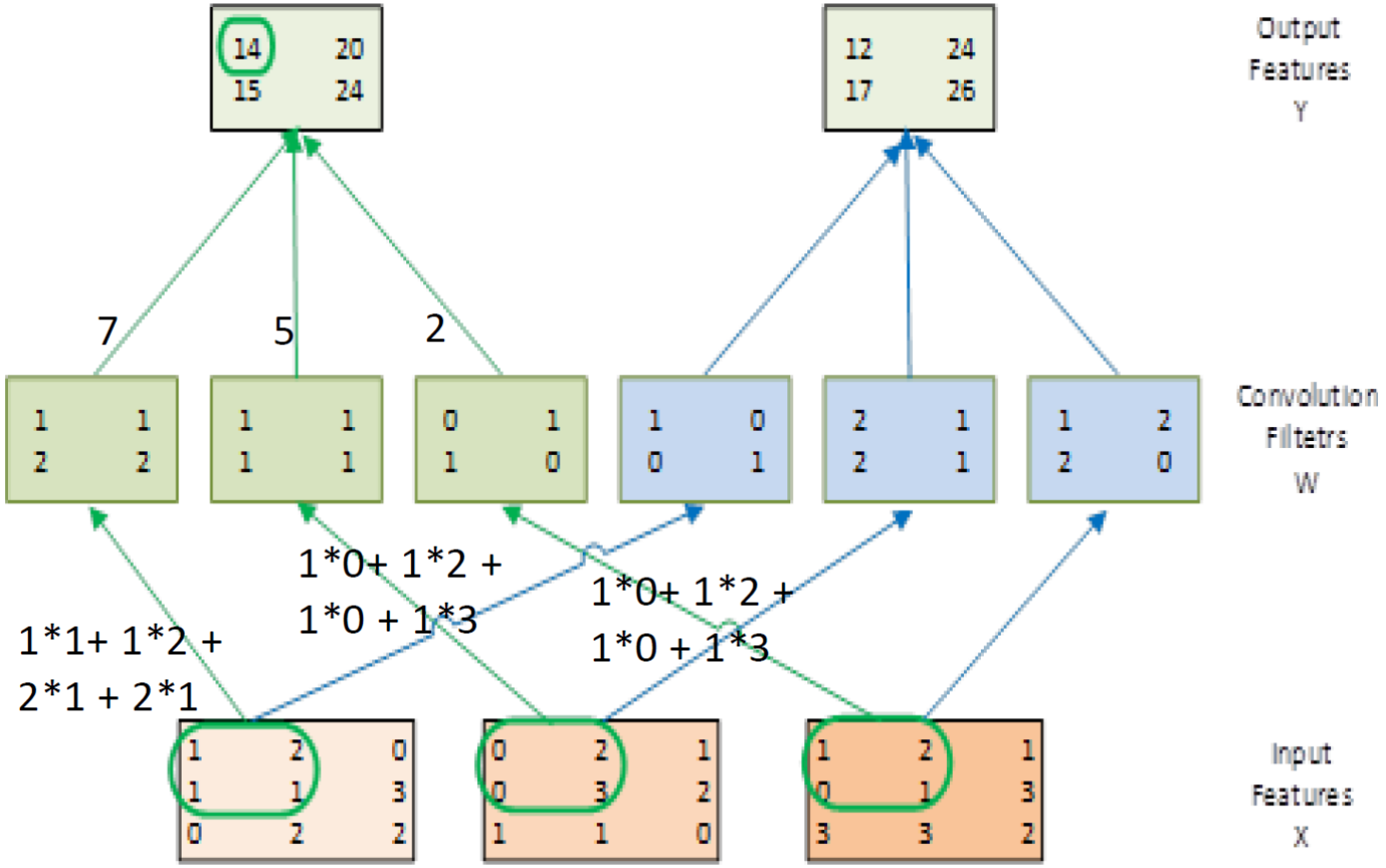
- Input is an image, 2D matrix (MLP is a vector)
- Linear function is convolution W⊛X (MLP is W X+b)
  - Learn each filter
- ▪ Advantage: Reduce number of weights, general better practice





Example of CNN

- 3 input feature maps, each is 3x3
- 2 output features, each is 2x2
- 6 Convolutional filters
  - Window size 2x2
- All the pixels in the same image shares filter
  - Total 2x2x6 = 24 weights
- If is MLP:
  - 27 x 8 = 216 weights



FPGA-based DNN Inference Works

- Representative FPGA-based DNN Inference Works
  - **HLS Optimization**
    - [FPGA'15] Optimizing FPGA-based CNN Accelerator
  - **Winograd, OpenCL for FPGA flow**
    - [FPGA'17] OpenCL Deep Learning Accelerator
  - **Low Bit-width Neural Networks**
    - [FPGA'17] Accelerating Binarized CNN
  - **Real-time CNN Inference in Embedded Systems**
    - [DAC'19] Real-time Object Detection with SoC FPGAs

Recent FPGA-based DNN Inference Works

HLS Optimization

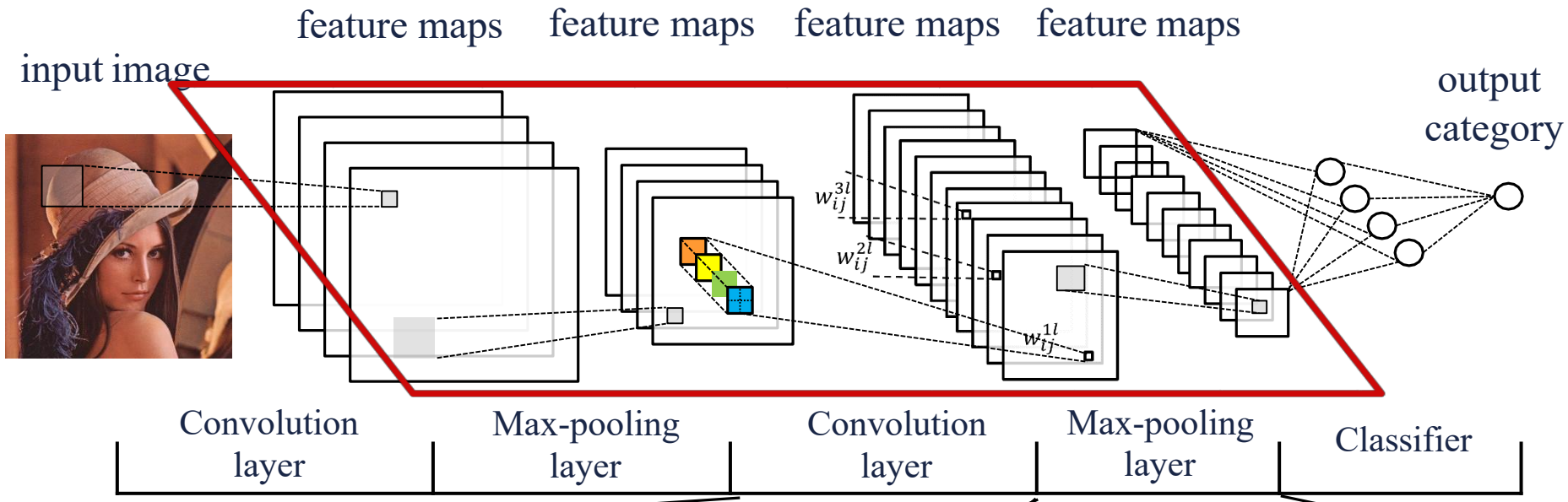
ISFPGA 2015

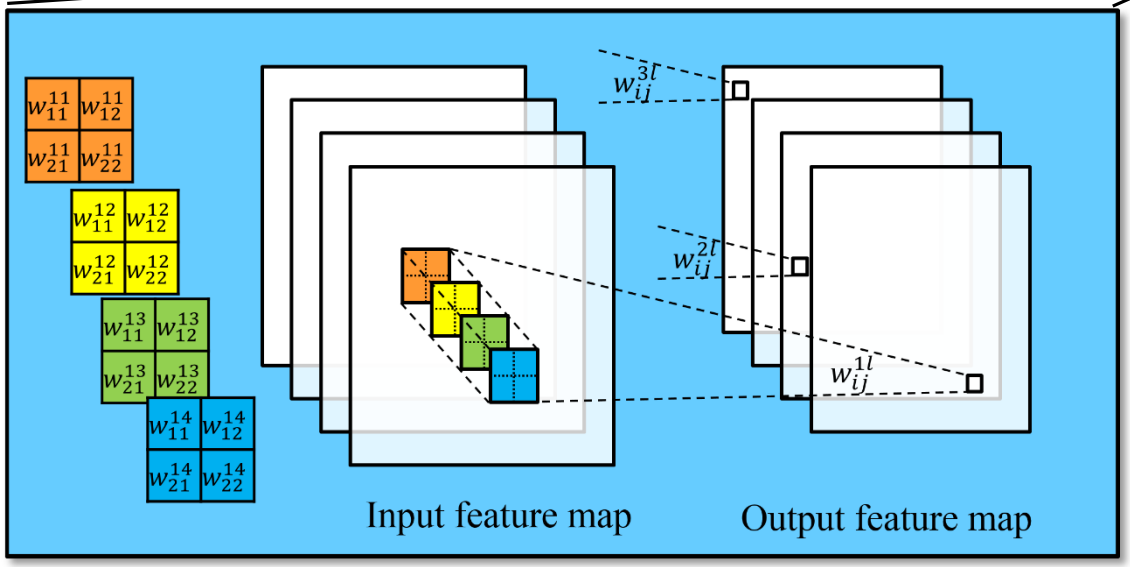
Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Network

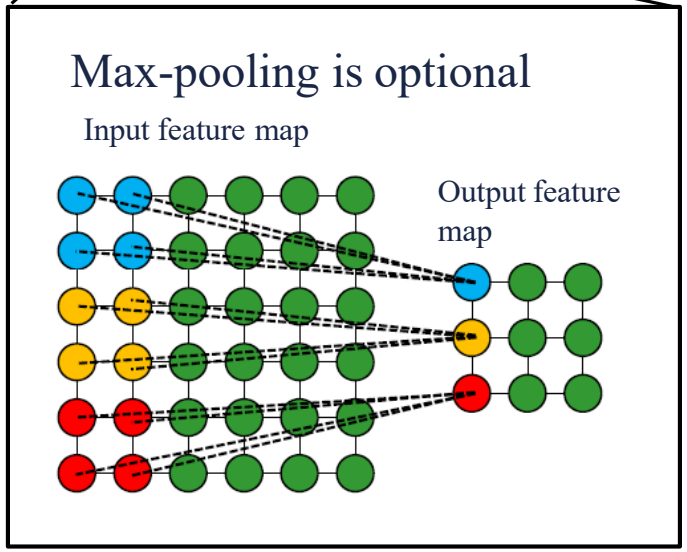
Chen Zhang[1], Peng Li[3], Guangyu Sun[1,2], Yijin Guan[1], Bingjun Xiao[3], Jason Cong[1,2,3]

[1]Peking University
[2]PKU/UCLA Joint Research Institution
[3]University of California, Los Angeles

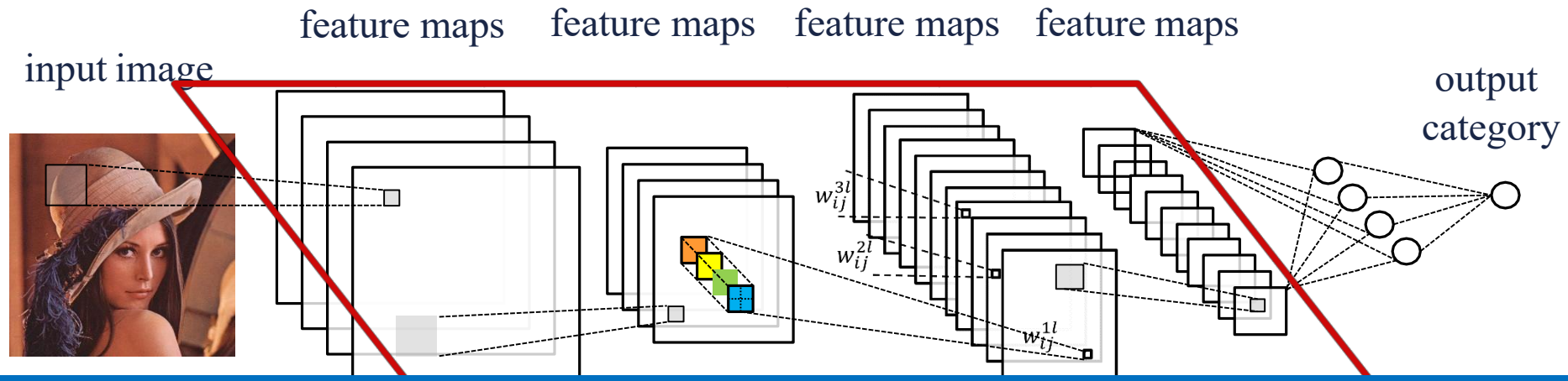
Convolutional Neural Network





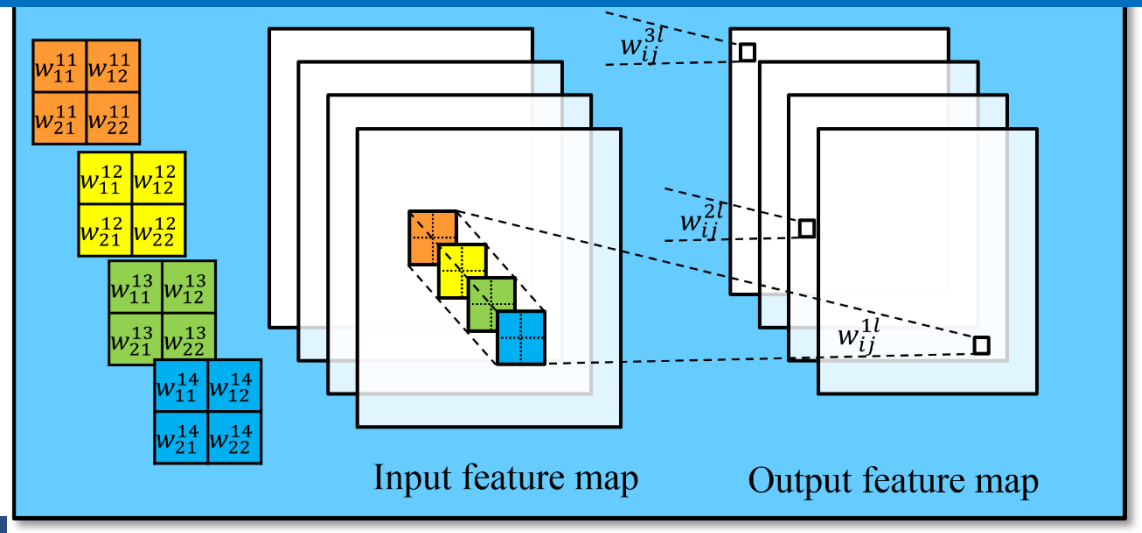


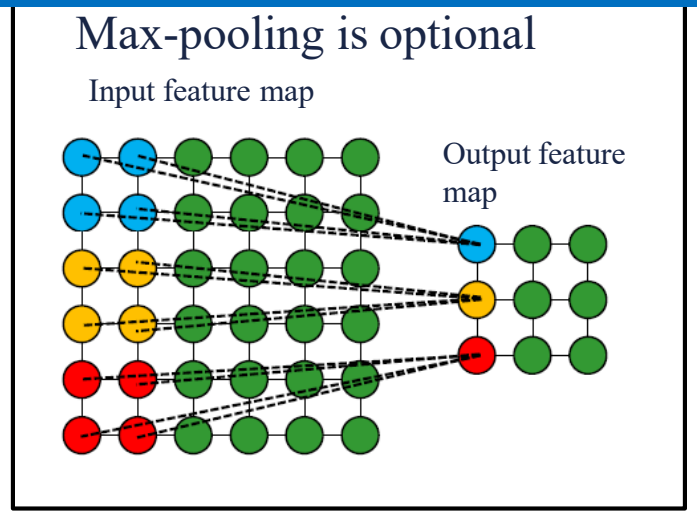
Convolutional Neural Network



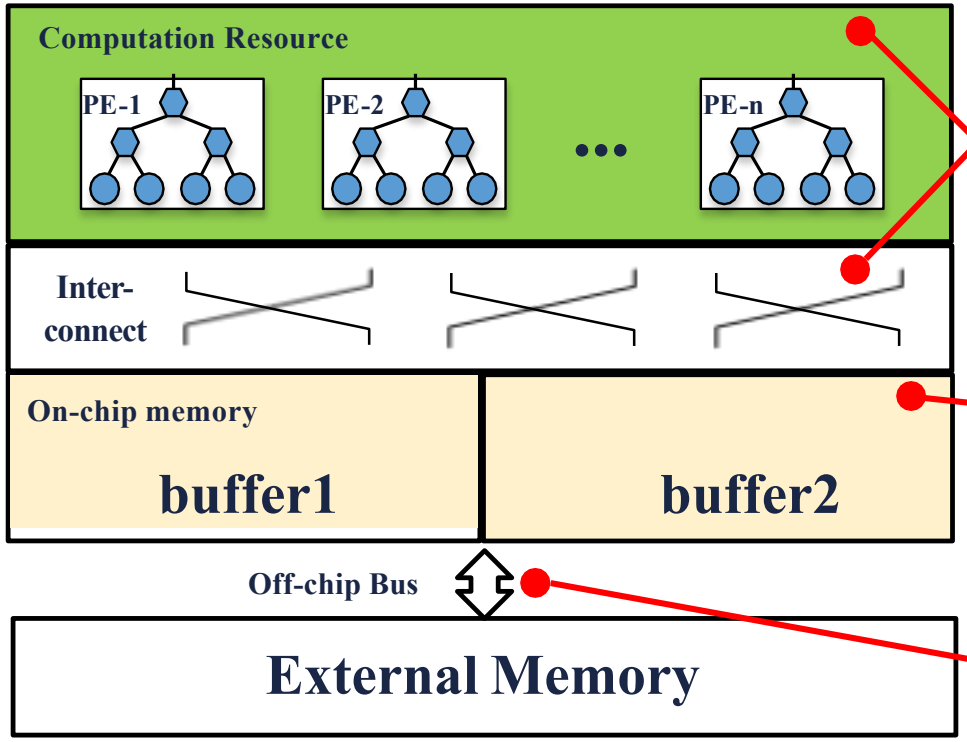
Convolutional layers account for over 90% computation

1 A. Krizhevsky, etc. Imagenet classification with deep convolutional neural networks. NIPS 2012.
2 J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. ICANN 2014





Hardware Computing on FPGA



Challenge: Resrc. Util.

Solution: Unroll & Pipeline

+

Challenge: BRAM Limits

Solution: Loop Tiling

+

Challenge: BW Limits

Solution: Data reuse

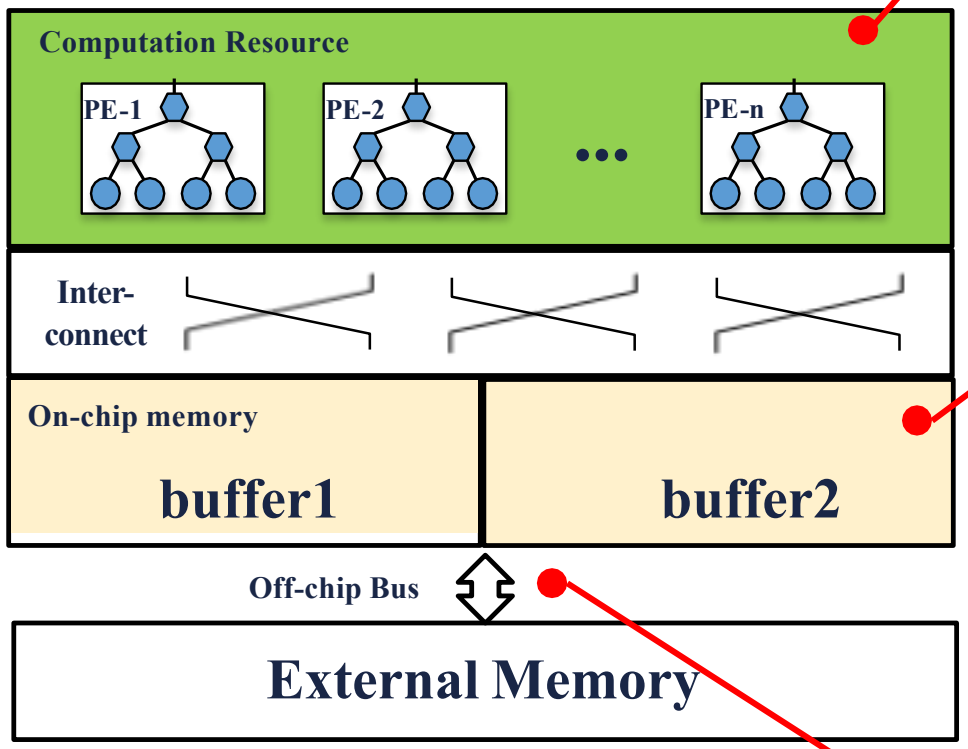
=

Challenge: Co-optimization,
-- match 'computation' & 'communication'

Solution: Roofline Model

Main Contribution

FPGA design in Roofline Model



Computational Perf.

$$= \frac{total\ number\ of\ operations}{execution\ cycles}$$

GFLOP/S

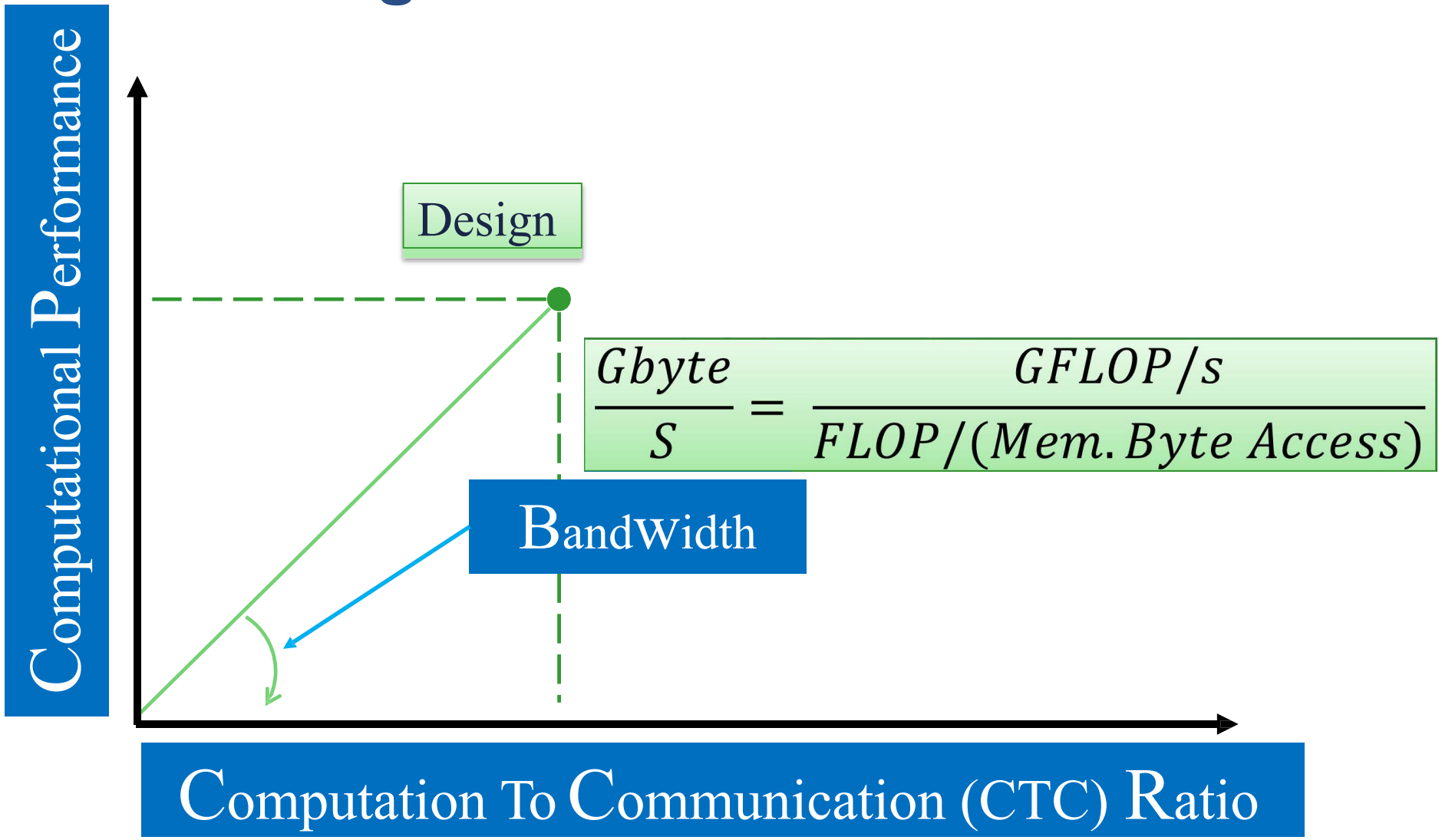
Computation To Communication Ratio

$$= \frac{Total\ number\ of\ operations}{Amount\ of\ external\ data\ access}$$

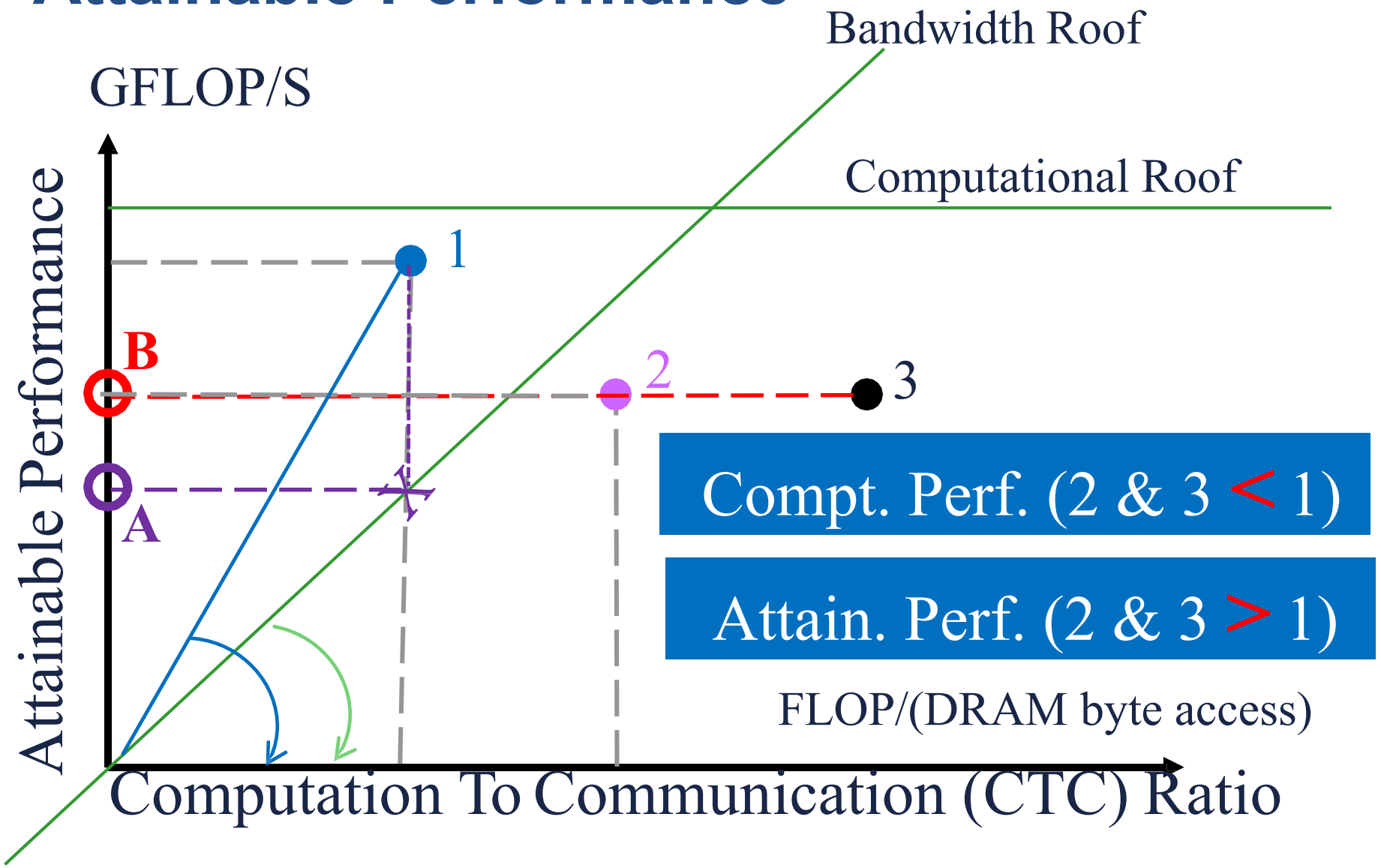
FLOP / (Mem. Byte Access)

Gbyte/S

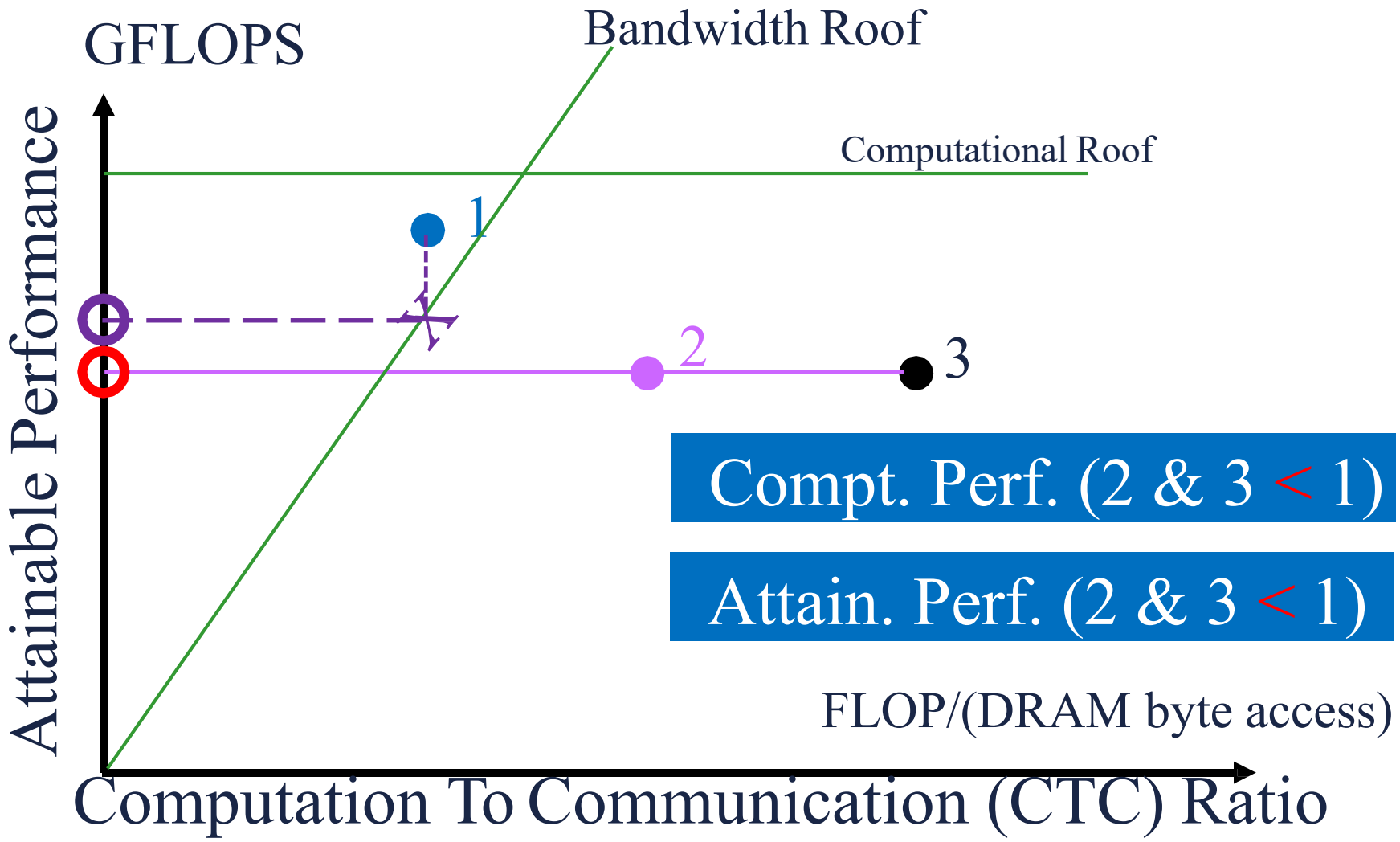
FPGA design in Roofline Model



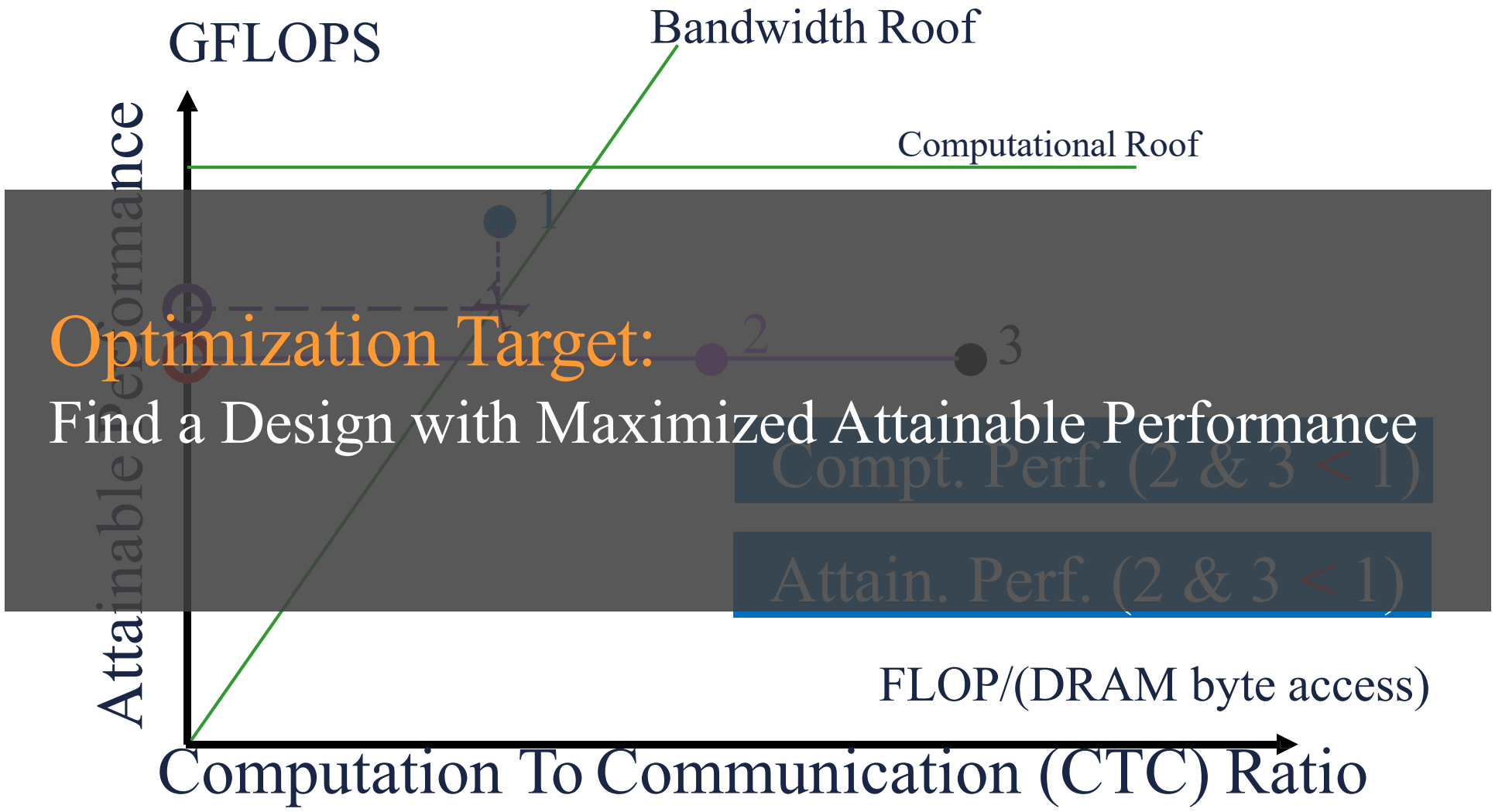
Attainable Performance



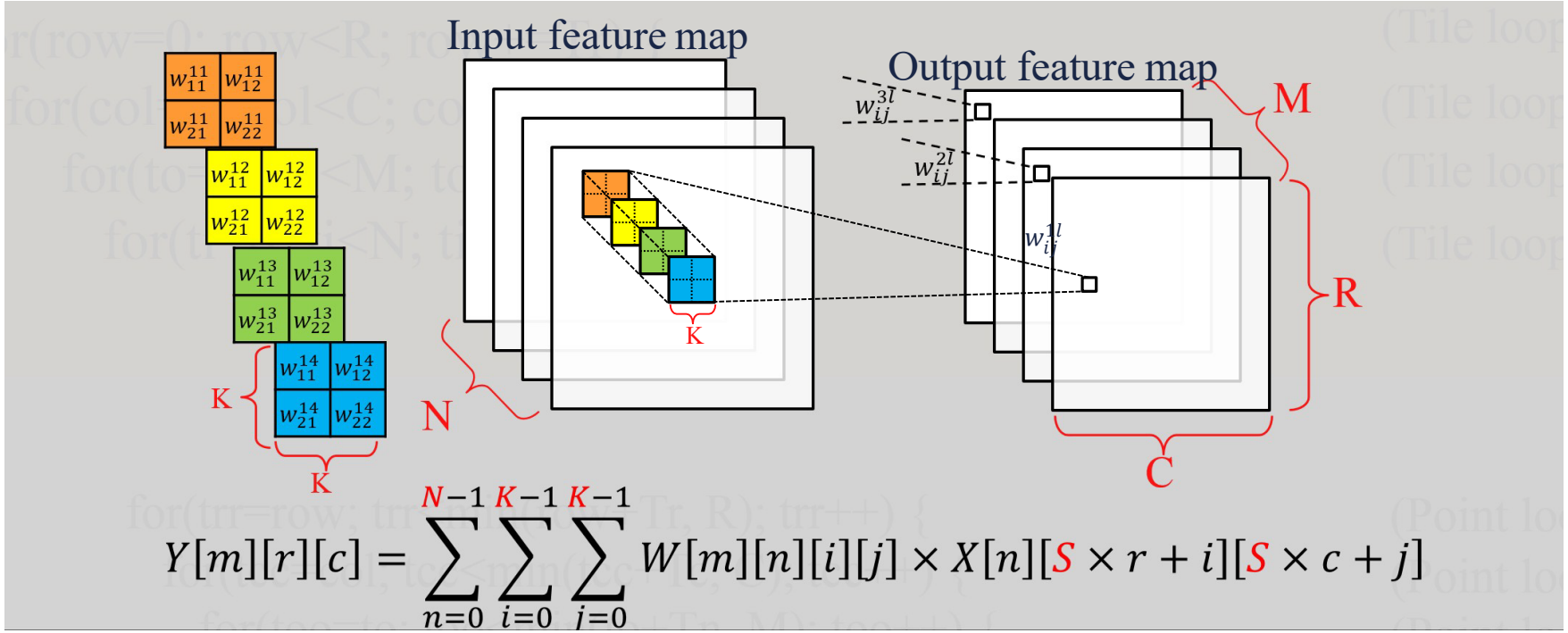
Attainable Performance



Attainable Performance



Loop tiling



$$Y[m][r][c] = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} W[m][n][i][j] \times X[n][S \times r + i][S \times c + j]$$

```
1  for(row=0; row<R; row++) {
2     for(col=0; col<C; col++) {
3        for(to=0; to<M; to++) {
4           for(ti=0; ti<N; ti++) {
5              for(i=0; i<K; i++) {
6                 for(j=0; j<K; j++) {
                     output_fm[to][row][col] +=
                     weights[to][ti][i][j]*input_fm[ti][S*row+i][S*col+j];
}}}}}}
```

R, C, M, N, K, S are all configuration parameters of the convolutional layer

Loop tiling

```
1  for(row=0; row<R; row+=Tr) {                                   (Tile loop)
2    for(col=0; col<C; col+=Tc) {                                 (Tile loop)
3      for(to=0; to<M; to+=Tm) {                                  (Tile loop)
4        for(ti=0; ti<N; ti+=Tn) {                                (Tile loop)
```

Off-chip Data Transfer: Memory Access Optimization

On-chip Data: Computation Optimization

```
5          for(trr=row; trr<min(row+Tr, R); trr++) {              (Point loop)
6            for(tcc=col; tcc<min(tcc+Tc, C); tcc++) {            (Point loop)
7              for(too=to; too<min(to+Tn, M); too++) {            (Point loop)
8                for(tii=ti; tii<min(ti+Tn, N); tii++) {          (Point loop)
9                  for(i=0; i<K; i++) {                           (Point loop)
10                   for(j=0; j<K; j++) {                         (Point loop)
                       output_fm[too][trr][tcc] +=
                       weights[too][tii][i][j]*input_fm[tii][S*trr+i][S*tcc+j];
             }}}}}}
}}}}
```

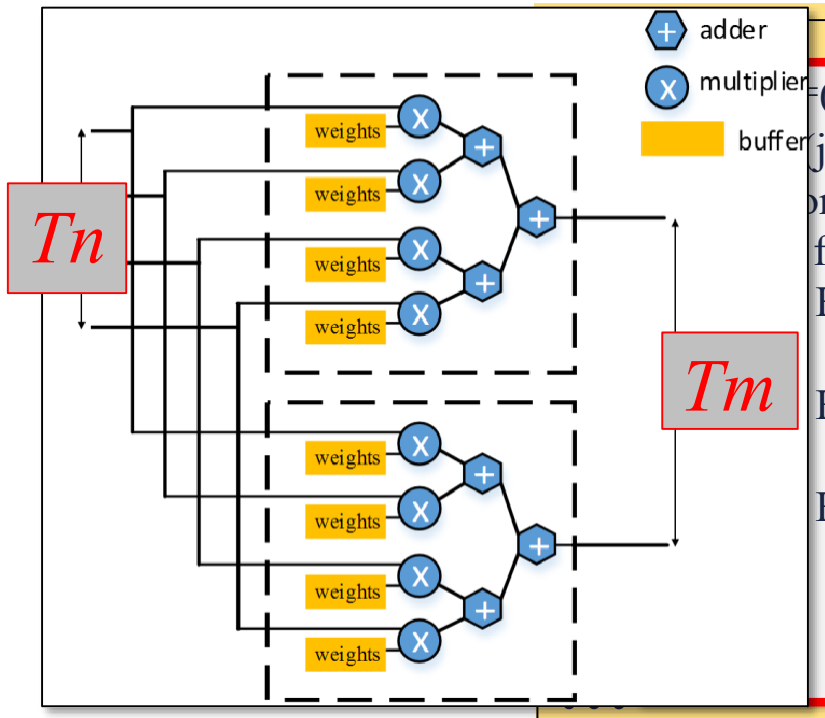
Computation Optimization

(Tile loops)

```
5for(trr=row; trr<min(row+Tr, R); trr++)                    (Point loop)
6     for(tcc=col; tcc<min(tcc+Tc, C); tcc++)               (Point loop)
7        for(too=to; too<min(to+Tn, M); too++)              (Point loop)
8           for(tii=ti; tii<(ti+Tn, N); tii++)              (Point loop)
9              for(i=0; i<K; i++) {
10                for(j=0; j<K; j++) {
                     output_fm[too][trr][tcc] +=
                     weights[too][tii][i][j]*input_fm[tii][S*trr+i][S*tcc+j];
    }}}}}}
```

```
5              for(i=0; i<K; i++) {
6               for(j=0; j<K; j++) {
7        for(trr=row; trr<min(row+Tr, R); trr++) {
8           for(tcc=col; tcc<min(tcc+Tc, C); tcc++) {
#pragma HLS pipeline
9             for(too=to; too<min(to+Tm, M); too++) {
#pragma HLS UNROLL
10              for(tii=ti; tii<(ti+Tn, N); tii++) {
#pragma HLS UNROLL
                   output_fm[too][trr][tcc] +=
                   weights[too][tii][i][j]*input_fm[tii][S*trr+i][S*tcc+j];
             }}}}}}
```

Computation Optimization



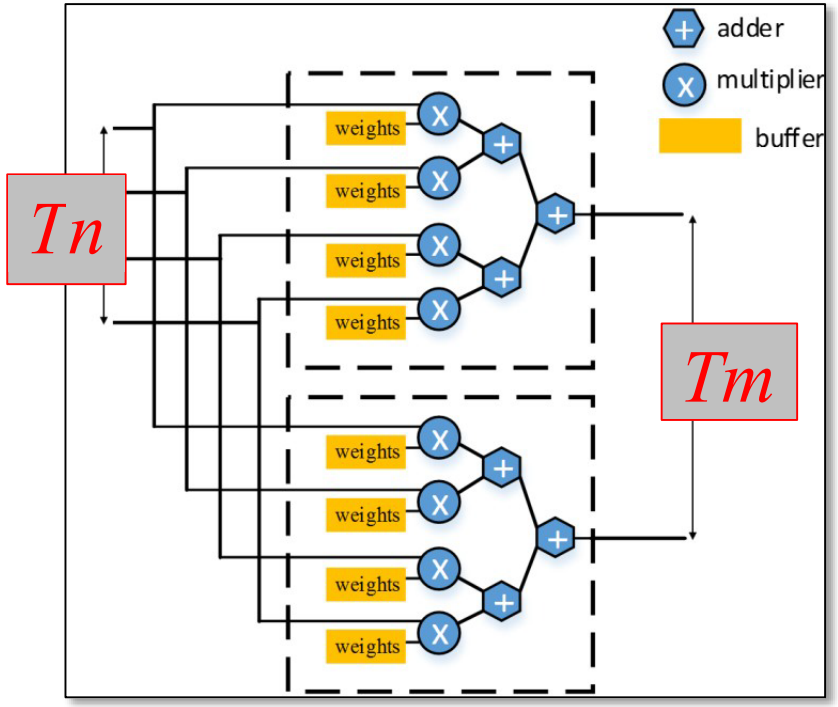

```
(Tile loops)
=0; i<K; i++) {
j=0; j<K; j++) {
or(trr=row; trr<min(row+Tr, R); trr++) {
 for(tcc=col; tcc<min(tcc+Tc, C); tcc++) {
HLS pipeline
  for(too=to; too<min(to+Tm, M); too++) {
HLS UNROLL
    for(tii=ti; tii<(ti+Tn, N); tii++) {
HLS UNROLL
      output_fm[too][trr][tcc] +=
      weights[too][tii][i][j]*input_fm[ti][S*trr+i][S*tcc+j];
  }}}}}}
```

$$\text{Computational Performance} = \frac{total\ number\ of\ operations}{execution\ cycles}$$

$$\text{execution cycles} = \frac{R}{Tr} \times \frac{C}{Tc} \times \left\lceil \frac{M}{Tm} \right\rceil \times \left\lceil \frac{N}{Tn} \right\rceil \times (K \times K \times Tc \times Tr + P)$$

$$\approx \left\lceil \frac{M}{Tm} \right\rceil \times \left\lceil \frac{N}{Tn} \right\rceil \times R \times C \times K \times K$$

Computational Performance



	Design 1	Design 2	Design 3	Design 4
Output (Tm)	5	10	20	30
Input (Tn)	5	10	20	15
DSPs	125	500	2000	2250

Design Space

Computation Engine:

Constraints for CNN configurations:

$Tm \in (Integer, 1 < Tm < M)$ N=128

$Tn \in (Integer, 1 < Tn < N)$ N=192

Constraints for FPGA resource:

$Tn \times Tm \in (Integer, 1 < Tm \times Tn < \#\ of\ PE)$

# of PE = 450

Legal Solutions of Tm & Tn:

2097

×

Communication:

of memory access methods

Legal Solutions:

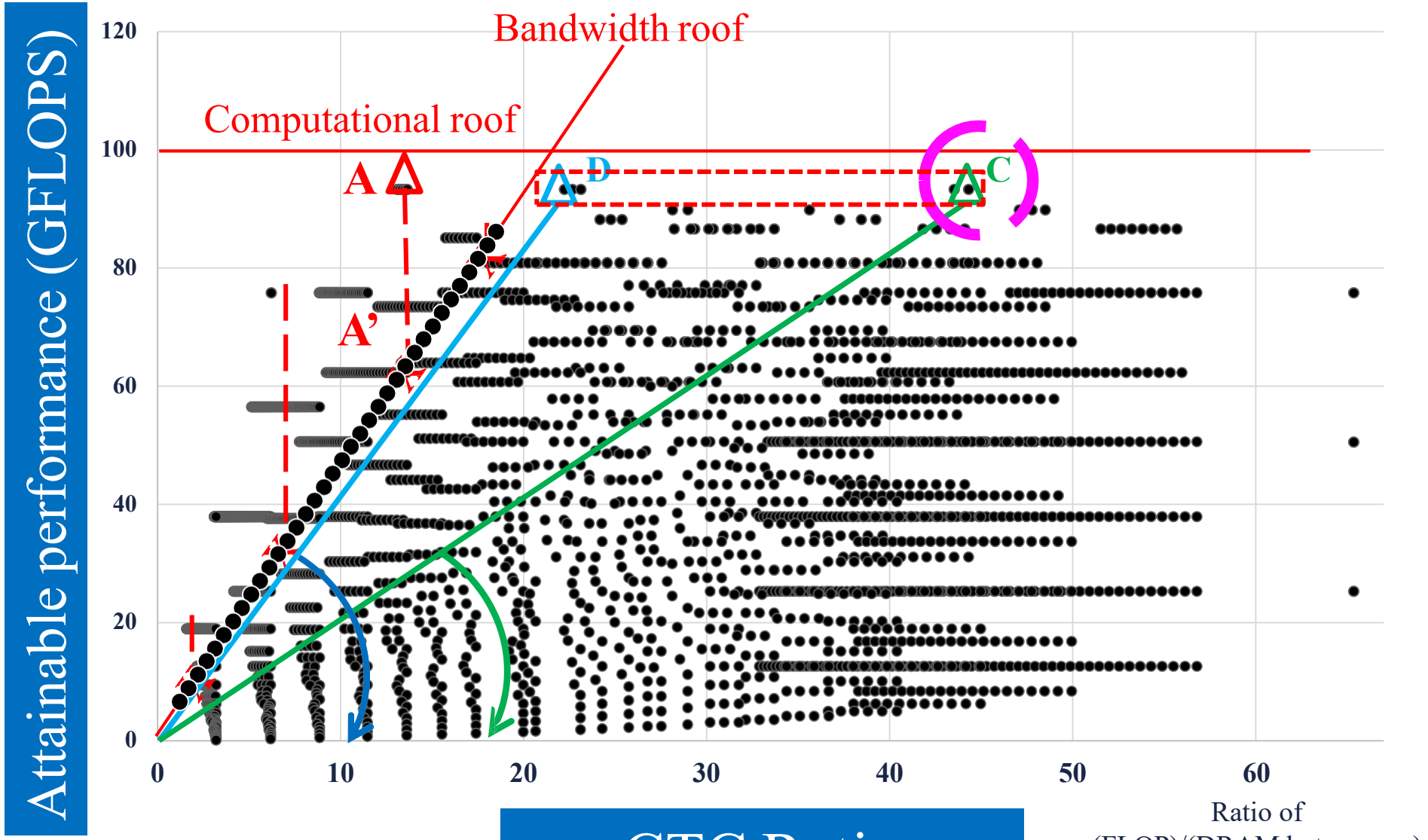
3

=

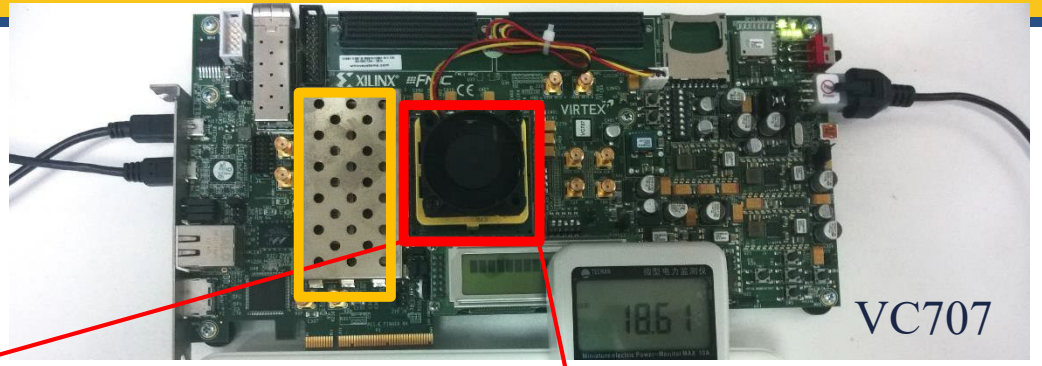
Total Legal Solutions:

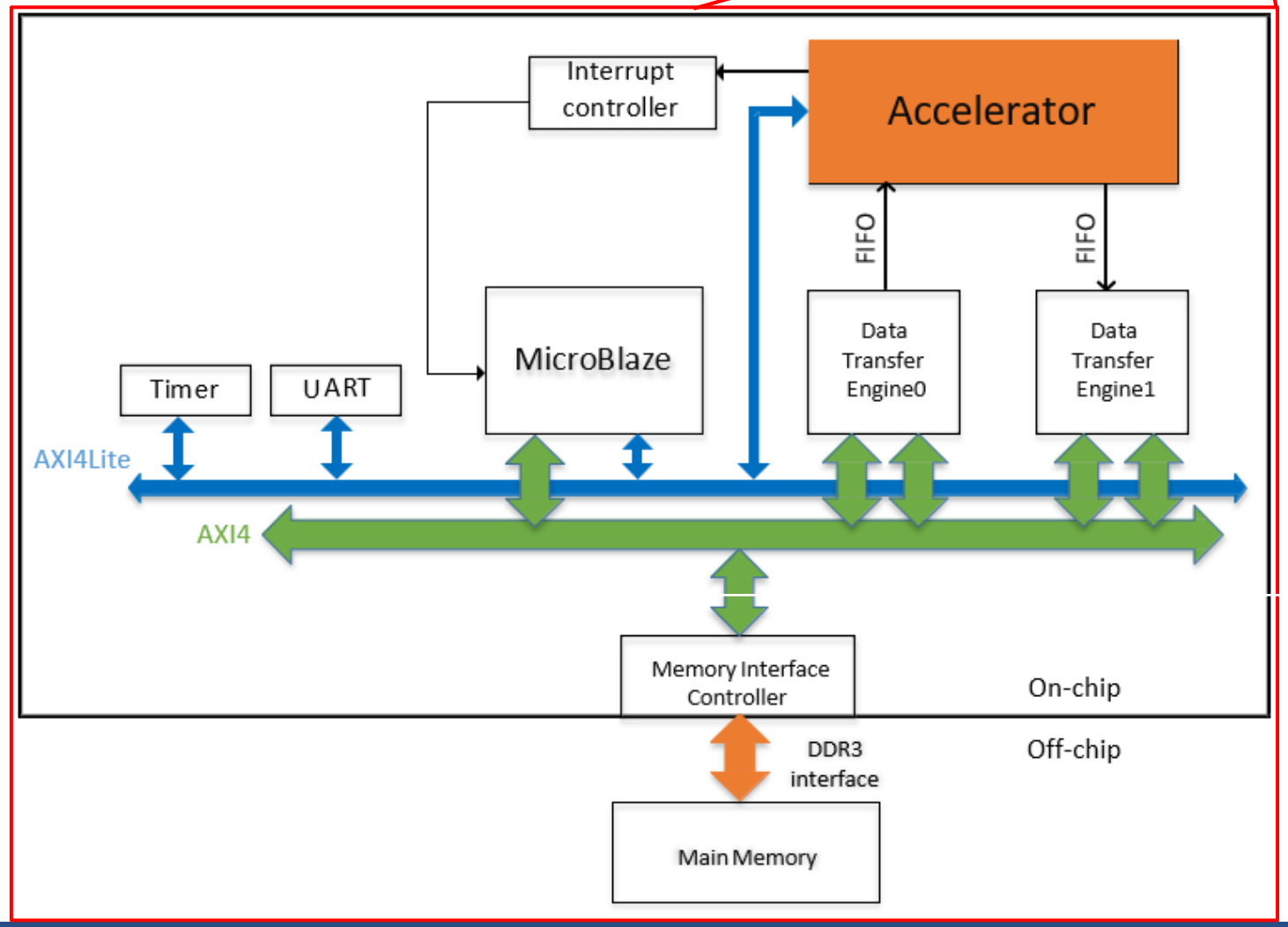
6291

Design Space Exploration



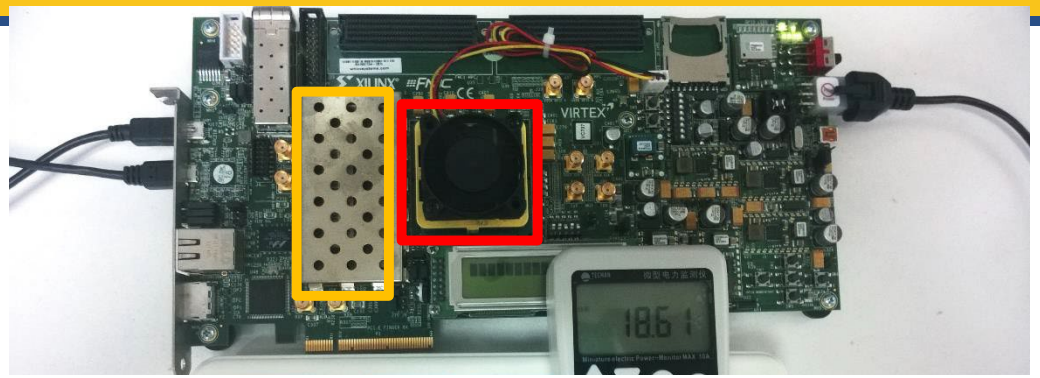
System Overview





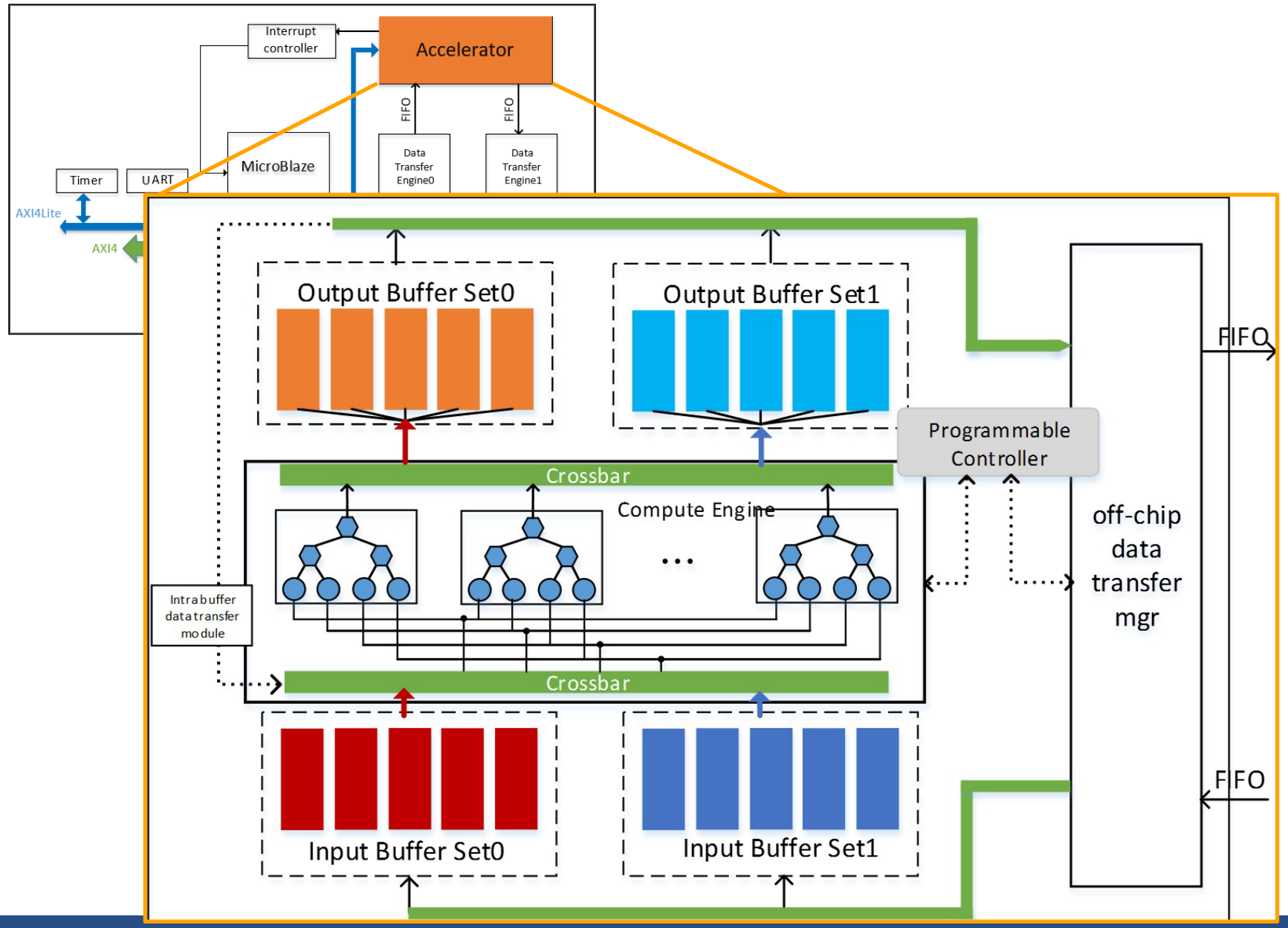
Vivado 2013.4
Vivado HLS 2013.4

System Overview

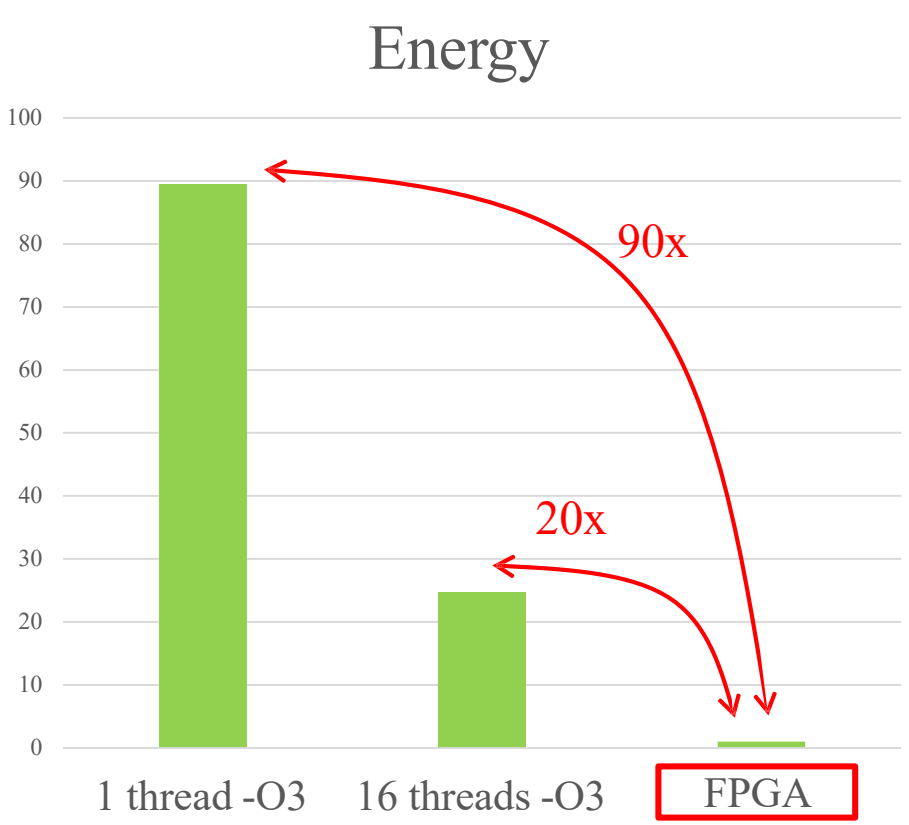


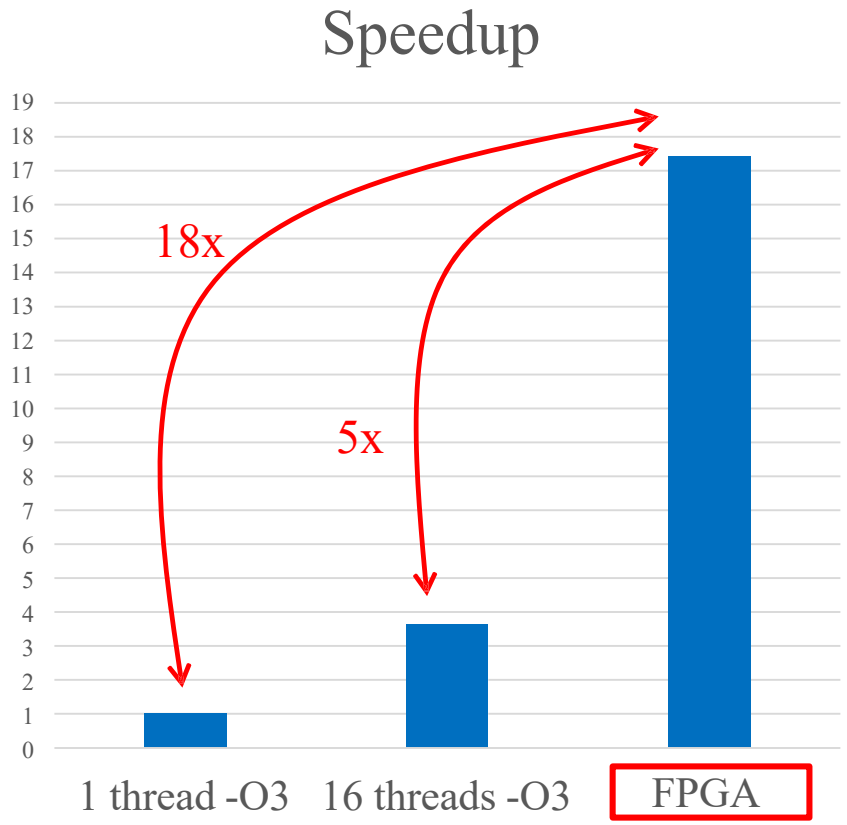
	Estimated	On-board run	Difference
Layer 1	7.32 ms	7.67 ms	~5%
Layer 2	5.11 ms	5.35 ms	~5%
Layer 3	3.42 ms	3.79 ms	~9%
Layer 4	2.59 ms	2.88 ms	~10%
Layer 5	1.73 ms	1.93 ms	~10%
Overall	20.17 ms	21.61 ms	~7%

Accelerator



Experimental Results: vs. CPU



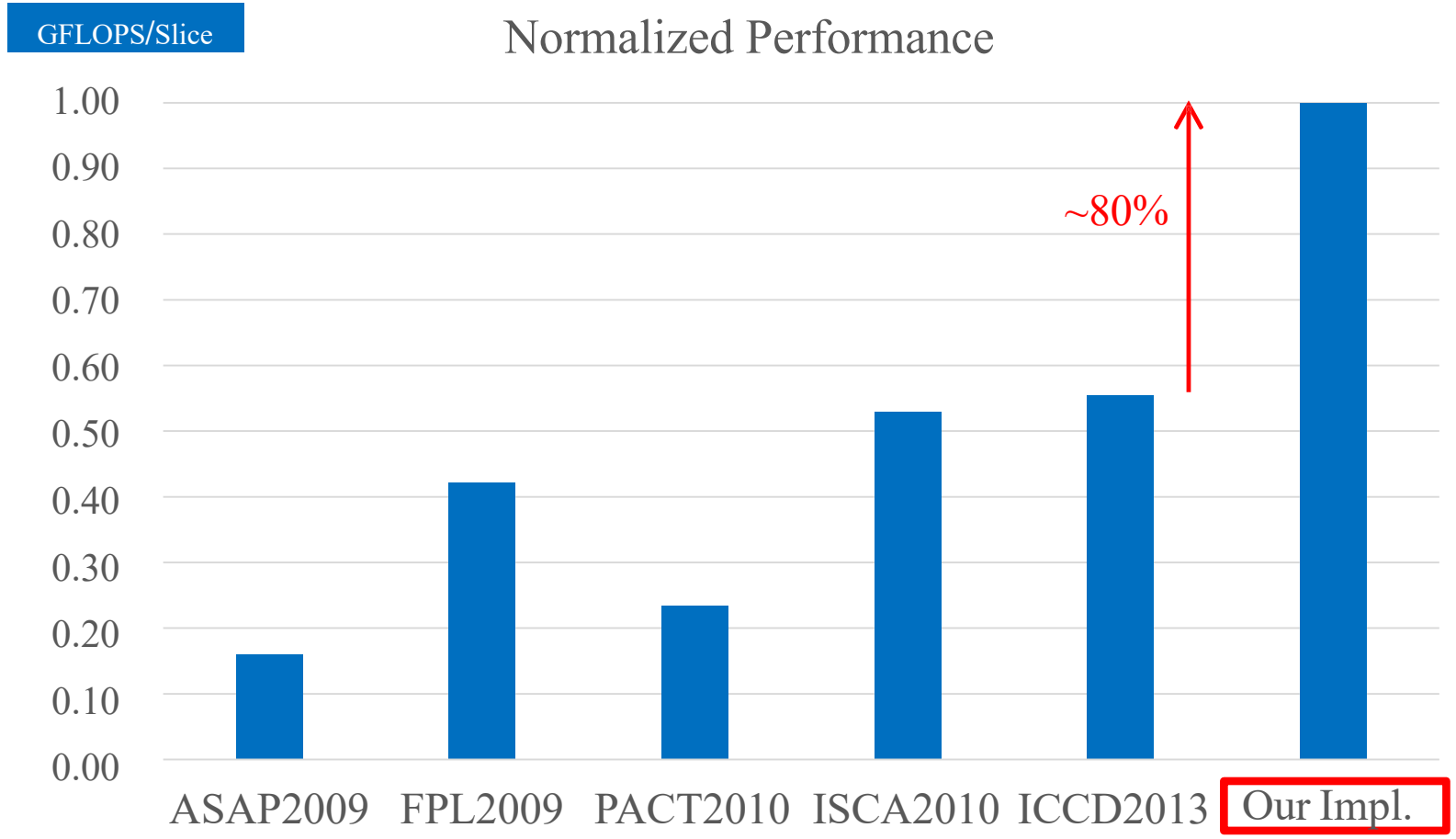


CPU	Xeon E5-2430 (32nm)	16 cores	2.2 GHz	gcc 4.7.2 –O3 OpenMP 3.0
FPGA	Virtex7-485t (28nm)	448 PEs	100MHz	Vivado 2013.4 Vivado HLS 2013.4

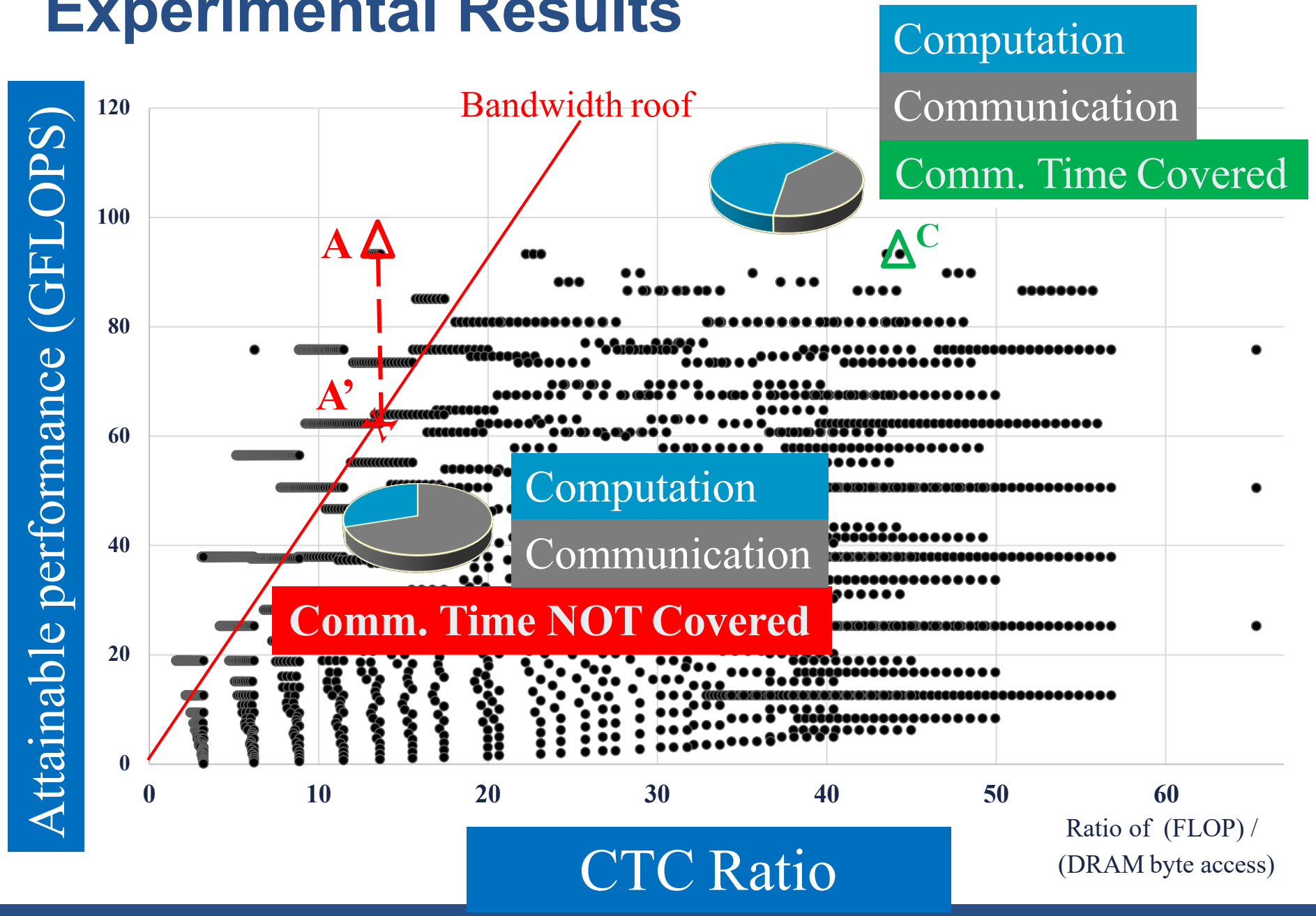
Experimental Results: vs. Other FPGAs

FPL2009	ASAP2009	PACT2010	ISCA2010	ICCD2013	Our Impl.
Virtex 4	Virtex 5	Virtex 5	Virtex 5	Virtex 6	Virtex 7
125MHz	115MHz	125MHz	200MHz	150MHz	100MHz
5.25 GOPS	6.74 GOPS	7 GOPS	16 GOPS	17 GOPS	61.6 GOPS

Experimental Results: vs. Other FPGAs



Experimental Results



Conclusions for this work

- An accelerator for convolutional neural network

- **Contribution:**
  - Accurate Analytical model for computation & communication
  - Find the best solution with roofline model

- **Result:**
  - On-board run implementation
  - ~3.5x better performance over other FPGA implementations
  - ~80% performance/area improvement

Today's Lecture

- Representative FPGA-based DNN Inference Works
  - **HLS Optimization**
    - [FPGA'15] Optimizing FPGA-based CNN Accelerator
  - **Winograd, OpenCL for FPGA flow**
    - [FPGA'17] OpenCL Deep Learning Accelerator
  - **Low Bit-width Neural Networks**
    - [FPGA'17] Accelerating Binarized CNN
  - **Real-time CNN Inference in Embedded Systems**
    - [DAC'19] Real-time Object Detection with SoC FPGAs

Recent FPGA-based DNN Inference Works

Winograd, OpenCL for FPGA flow

ISFPGA 2017

An OpenCL Deep Learning Accelerator on Arria 10

Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, Gordon R. Chiu

Intel Corporation
Toronto, Canada

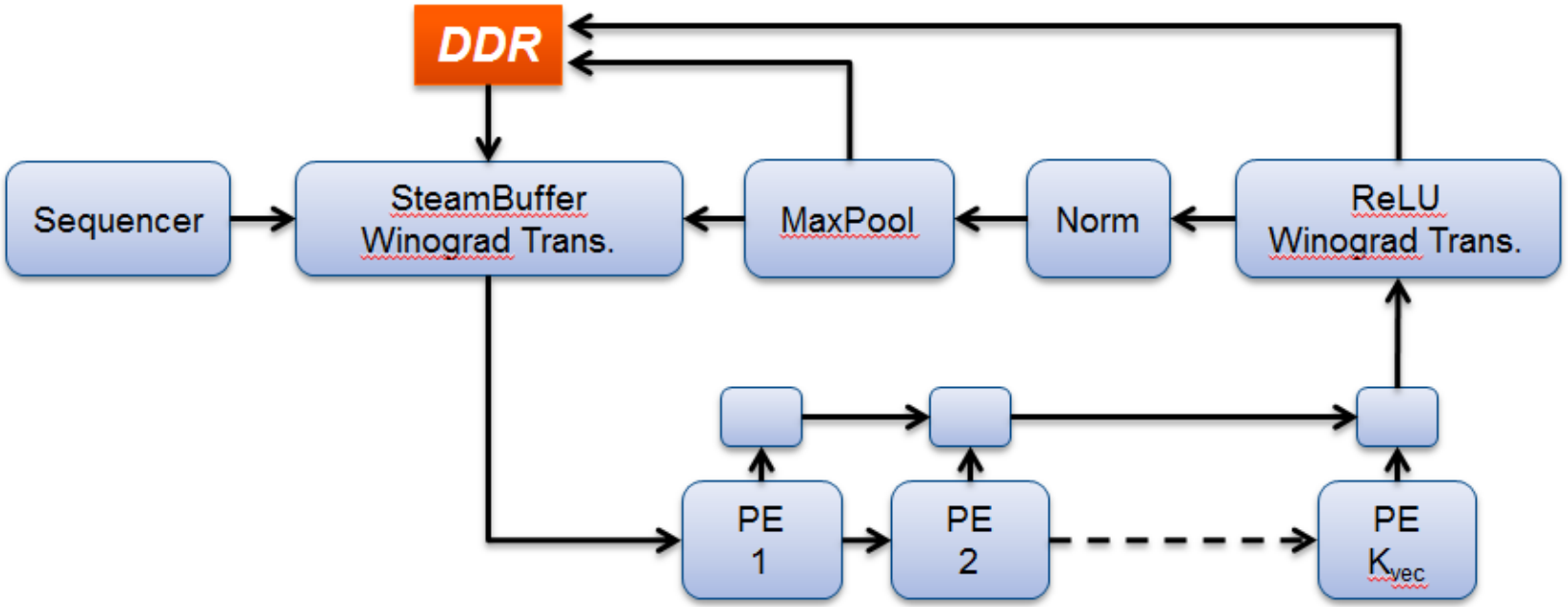
Highlights

- A CNN (AlexNet) accelerator written in OpenCL (kernels connected with pipes)
  - 10x better performance than previous FPGA implementation
  - Comparable performance/Watt with Titan X
- Reduce the required DRAM bandwidth by an order-of-magnitude
- Improve DSP efficiency (DSP efficiency: 62.6% ~ 99.8%) and usage
- Accurate analytical model

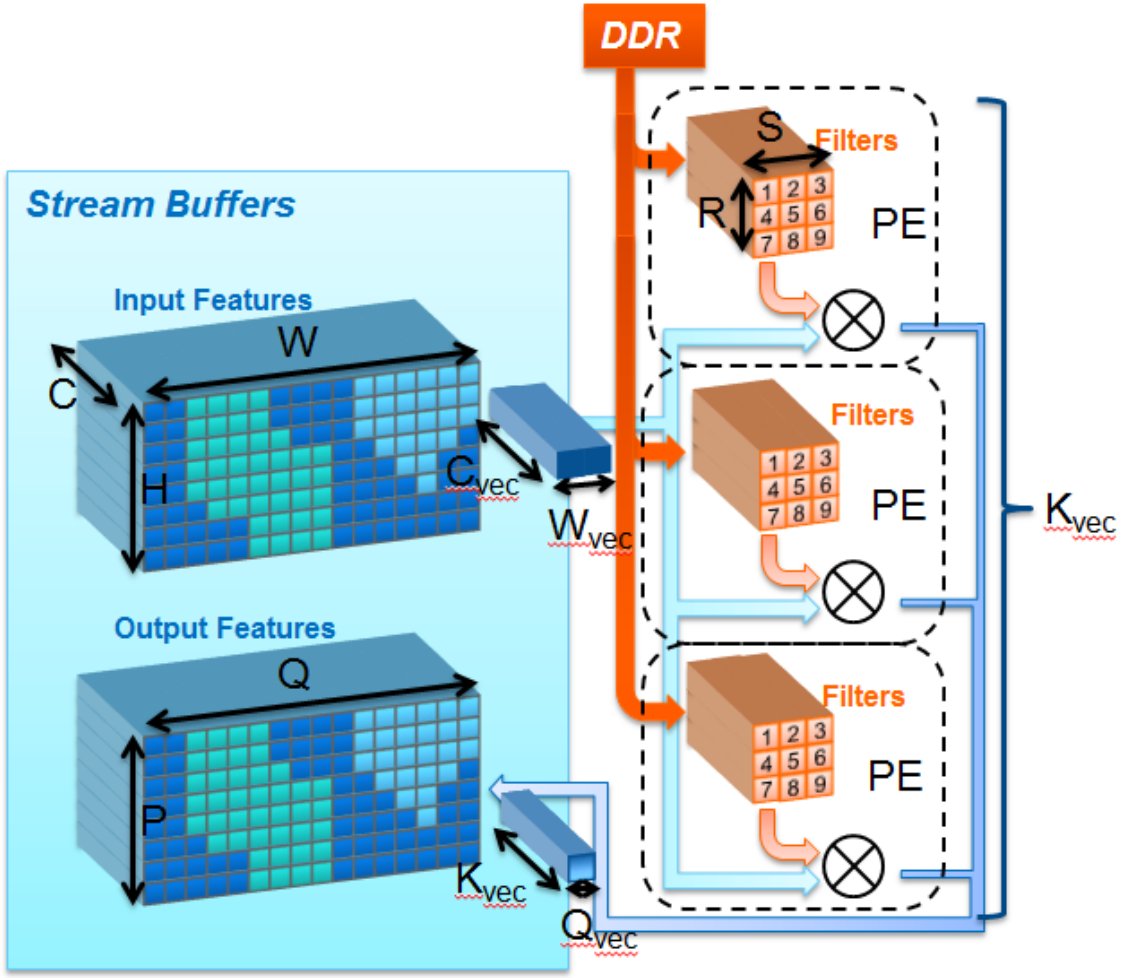
Techniques:

- On-chip stream buffer
- Vectorization approach
- Winograd Transform in conv. Layers
- Shared Exponent FP16

DLA (Deep Learning Accelerator)

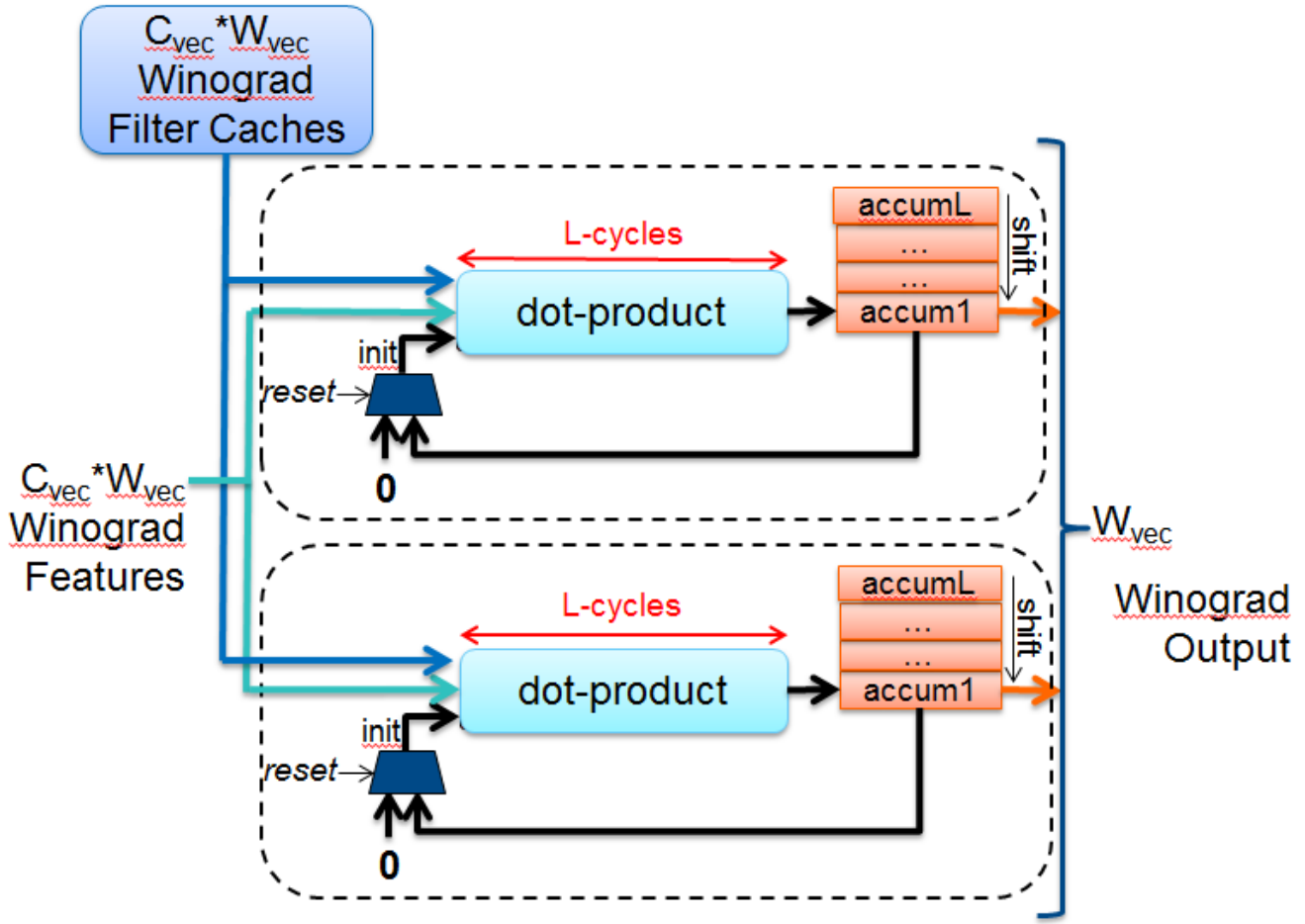


Vectorization



PE Structure

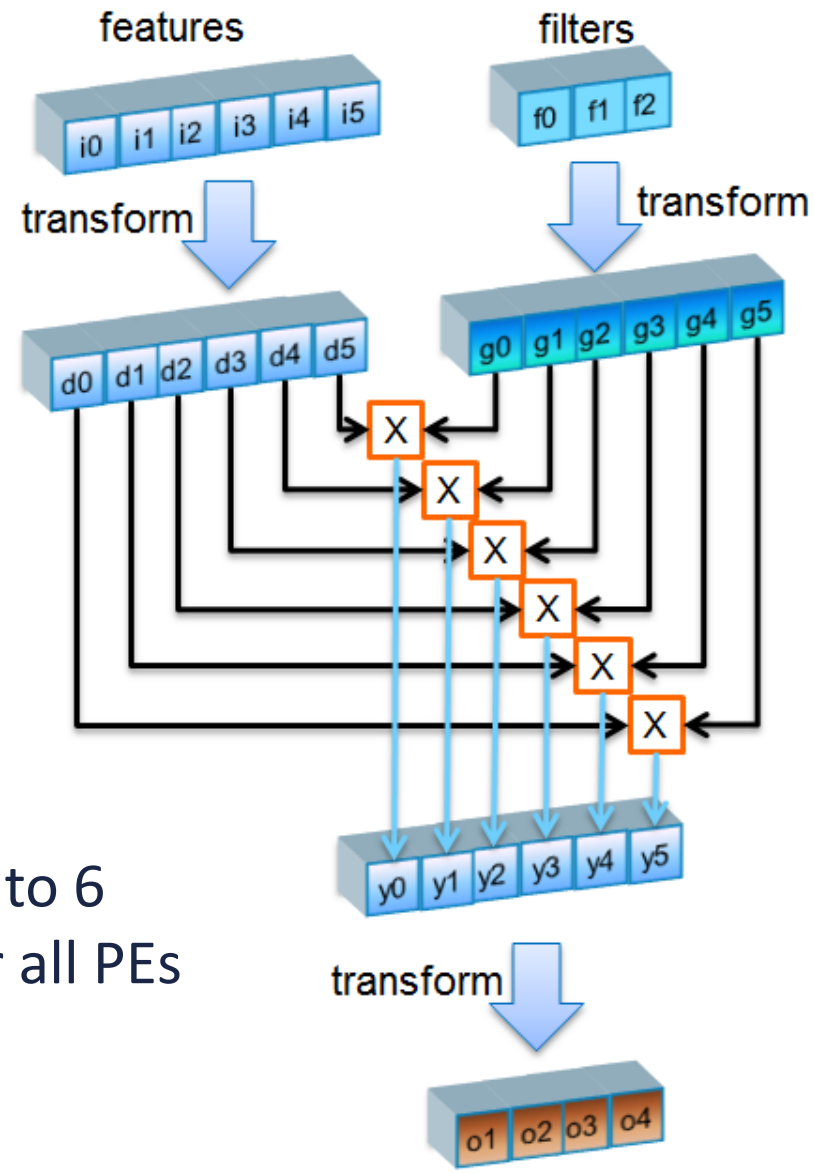
(when calculating convolutional layer)



Winograd Flow

(convolutional layers only)

$$o_0 = (f_0, f_1, f_2) \cdot (i_0, i_1, i_2)$$
$$o_1 = (f_0, f_1, f_2) \cdot (i_1, i_2, i_3)$$
$$o_2 = (f_0, f_1, f_2) \cdot (i_2, i_3, i_4)$$
$$o_3 = (f_0, f_1, f_2) \cdot (i_3, i_4, i_5)$$



- Reducing number of MAC operations from 12 to 6
- Transformation only need to be done once for all PEs

Convolution with Winograd and FFT

- Winograd Transform
  - Computing convolution with a smaller number of floating-point operations (multiplications)
    - Trading multiplication with addition
  - Good for small kernel sizes (3x3, 4x4)
- FFT
  - Convolution in time domain == elementwise multiplication in frequency domain

$$y(t) = h(t) * x(t) \xleftrightarrow{FT} Y(j\omega) = X(j\omega)H(j\omega)$$

  - Fast FFT algorithms and implementation
  - Good for big kernel sizes

Winograd Transform

- Winograd
  - Computing convolution with a smaller number of floating-point operations (multiplications)
- Example:
  - 1D convolution [d0, d1, d2, d3] with filter [g0, g1, g2]

$$\begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

6 multiplications

More details (1/3)

- Winograd documented the following algorithm

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

More details (2/3)

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

- Denote $g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$
  $d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$

$$[m0, m1, m2, m3] = (Gg) \odot (B^T d)$$

4 multiplications

where

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Only shifting and negation

More details (3/3)

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

- To compute the final output

$$Y = A^T \left[(Gg) \odot (B^T d) \right] \quad \text{where} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

No multiplication in A again

- Total 4 multiplication,
  - Compared to 1D convolution, saved 4/6 = 1.5
- For 2D convolution: window size 3x3, output size 2x2

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A$$

Multiplications:

- Standard convolution : 3x3x2x2 = 36
- Winograd: 4x4 = 16
- Saved 36/16 = 2.25

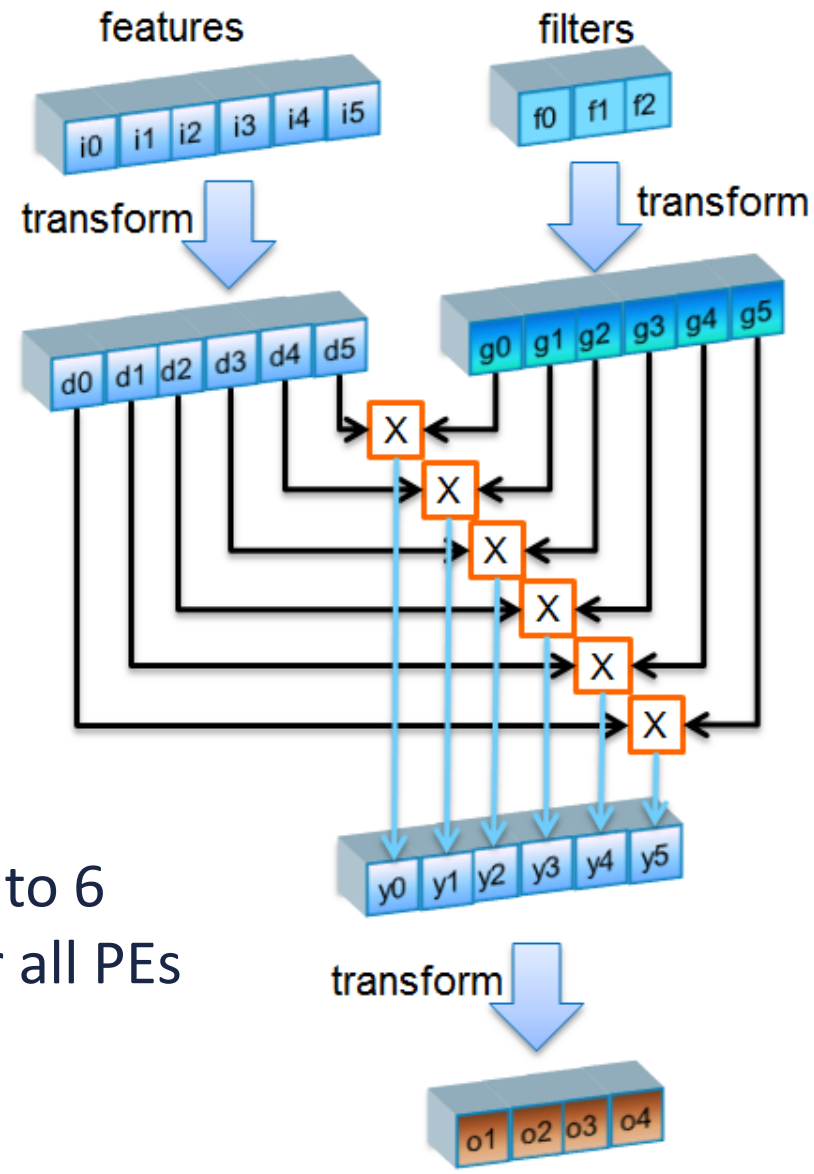
Notes on Winograd Algorithm

- Works well when window size is small.
  - 2x2, 3x3
  - Trading addition with multiplication
  - When window size becomes big, addition become dominating.

Winograd Flow

(convolutional layers only)

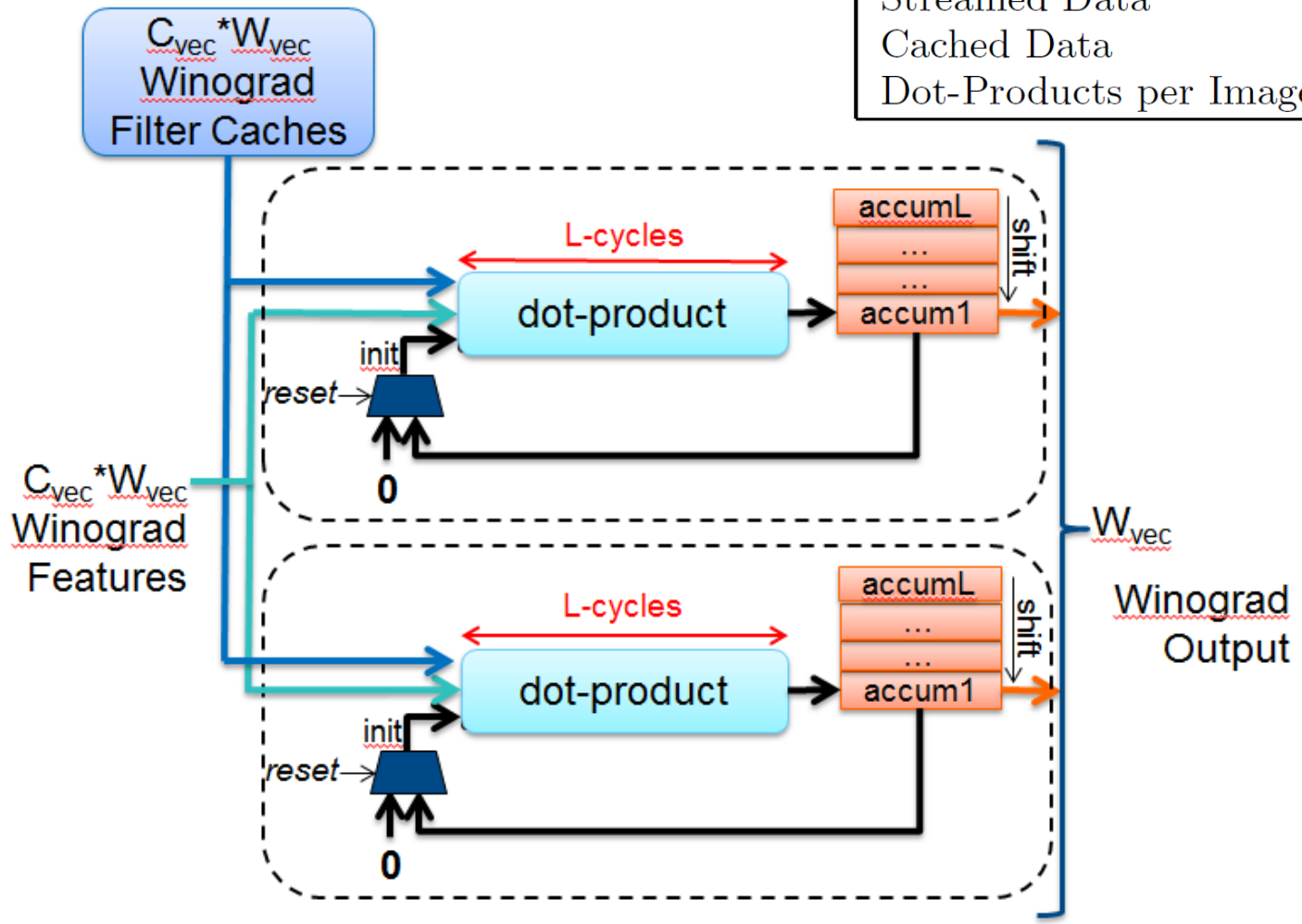
$$o_0 = (f_0, f_1, f_2) \cdot (i_0, i_1, i_2)$$
$$o_1 = (f_0, f_1, f_2) \cdot (i_1, i_2, i_3)$$
$$o_2 = (f_0, f_1, f_2) \cdot (i_2, i_3, i_4)$$
$$o_3 = (f_0, f_1, f_2) \cdot (i_3, i_4, i_5)$$



- Reducing number of MAC operations from 12 to 6
- Transformation only need to be done once for all PEs

Configuration

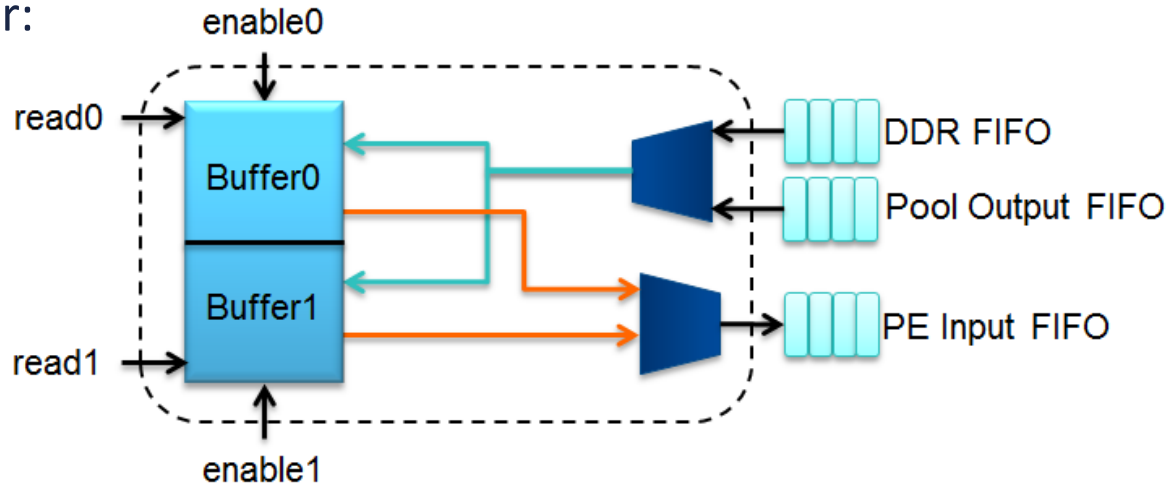
Configuration	Convolution	Fully-Connected
Winograd Transformation	Yes	No
Batch Size	1	S_{batch}
Streamed Data	Features	Filters
Cached Data	Filters	Features
Dot-Products per Image	W_{vec}	W_{vec}/N



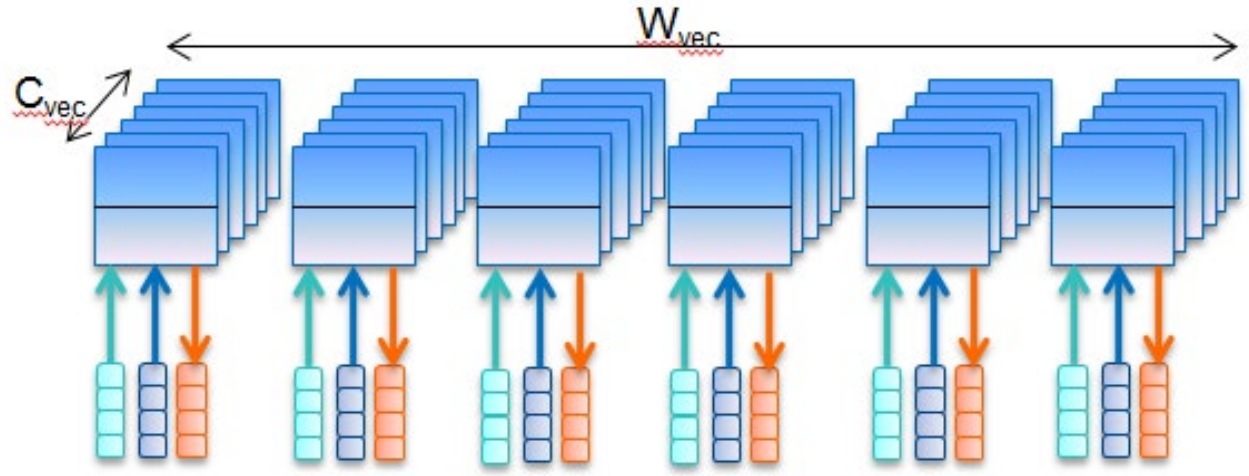
Stream Buffer

On-chip scratchpad memory (double buffer)

A single stream buffer:



The array of stream buffers:



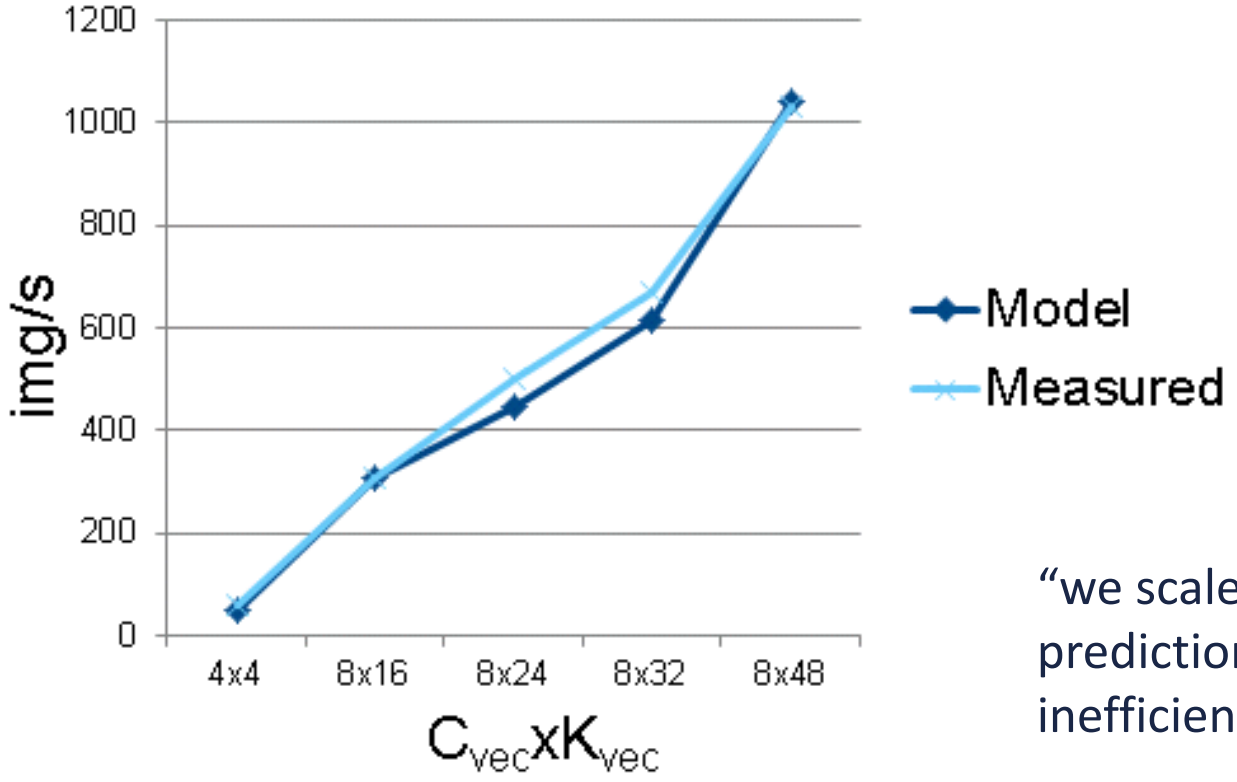
Shared Exponent FP16

- Using FP16 (instead of FP32) can significantly reduce the resource requirement of each PE
- However, FP16 is not natively supported
- Solution: within a group of dot-products, use the shared exponent, convert fractions to fixed-point numbers (18bits), perform multiplication in fixed-point, and convert back to 10bit fraction in FP16.
- "no impact to accuracy was seen to the top-1 and top-5 error rate (56% and 79% respectively) between our shared exponent implementation and 32-bit floating point"

FP16 config	ALMs	Reg
Half-type	10.7K	26K
Shared Exponent	3.3K	10.6K

Analytical Model

- Performance model and resource usage model
- Refer to paper for details
- Accuracy:



"we scale down the model img/s predictions by 16% to account for inefficiencies in data transfer"

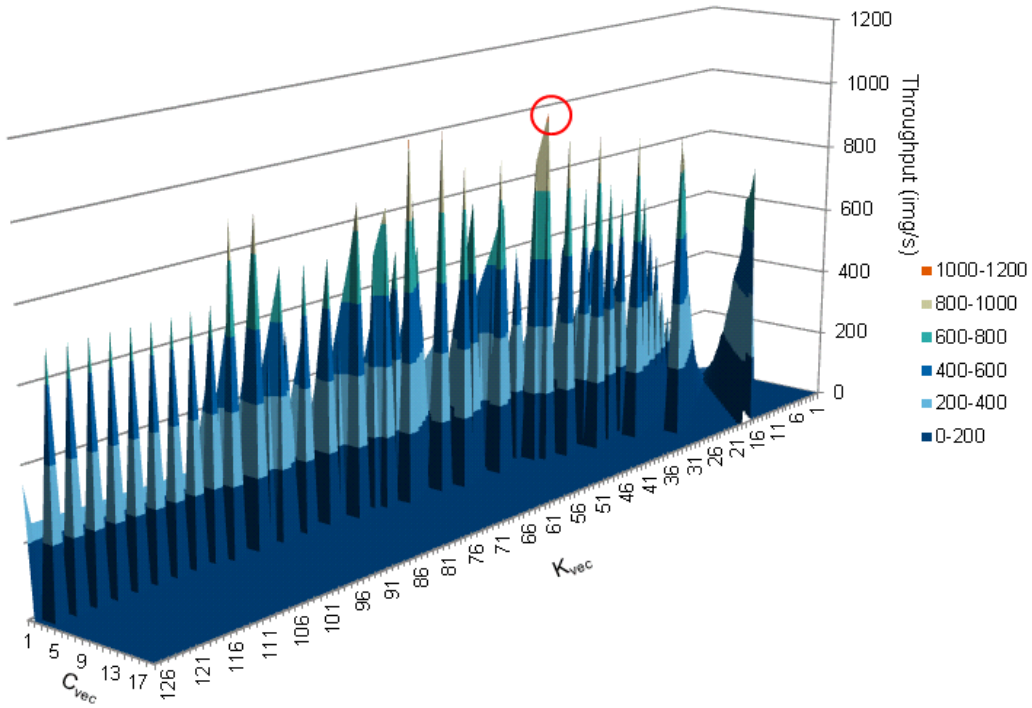
Results

Layer	Eff. GFLOPS	Act. GFLOPS	Eff.
Conv1	2,308	1,154	82.9%
Conv2	1,740	870	62.5%
Conv3	1,960	980	72.4%
Conv4	1,960	980	72.4%
Conv5	1,743	871	62.6%
Fc6	1,389	1,389	99.8%
Fc7	1,386	1,386	99.6%
Fc8	1,378	1,378	99.0%

Table 2: The average GFLOPS achieved of convolutional and fully-connected layers and DSP efficiency when using an 8×48 configuration. Shows both effective GFLOPS (Eff. GFLOPS) due to Winograd and actual GFLOPS (Act. GFLOPS).

ALMs	Reg	M20K	DSPs	Freq.
246K (58%)	681K	2487 (92%)	1476 (97%)	303 MHz

Table 4: Resource usage and clock frequency on the Arria 10 1150 device, for an 8×48 configuration running at 303MHz.



Design space exploration
(Opt.: $C_{vec} = 8$; $K_{vec} = 48$)

Results (cont.)

Stratix V (28nm)[16]	KU060 (20nm) [20]	DLA (20nm)
72.4 GOPS	165 GOPS	1382 GFLOPS

Improvement comes from better DSP efficiency and higher DSP utilization.

	img/s	Watts (W brd)	Peak Ops	img/s/W
DLA (20nm)	1020	45	1.3TFLOPS	23
KU060 (20nm)	104	25	3.6TOPS	4
TitanX (28nm)	5120	227	6.1TFLOPS	23
M4 (28nm)	1150	58	2.2TFLOPS	20

Takeaways

- Improve resource efficiency (DSPs, on-chip memory, memory bandwidth, etc.)
- Using OpenCL to describe hardware (?)

Today's Lecture

- Representative FPGA-based DNN Inference Works
  - **HLS Optimization**
    - [FPGA'15] Optimizing FPGA-based CNN Accelerator
  - **Winograd, OpenCL for FPGA flow**
    - [FPGA'17] OpenCL Deep Learning Accelerator
  - **Low Bit-width Neural Networks**
    - [FPGA'17] Accelerating Binarized CNN
  - **Real-time CNN Inference in Embedded Systems**
    - [DAC'19] Real-time Object Detection with SoC FPGAs

Recent FPGA-based DNN Inference Works

Low Bit-width Neural Networks

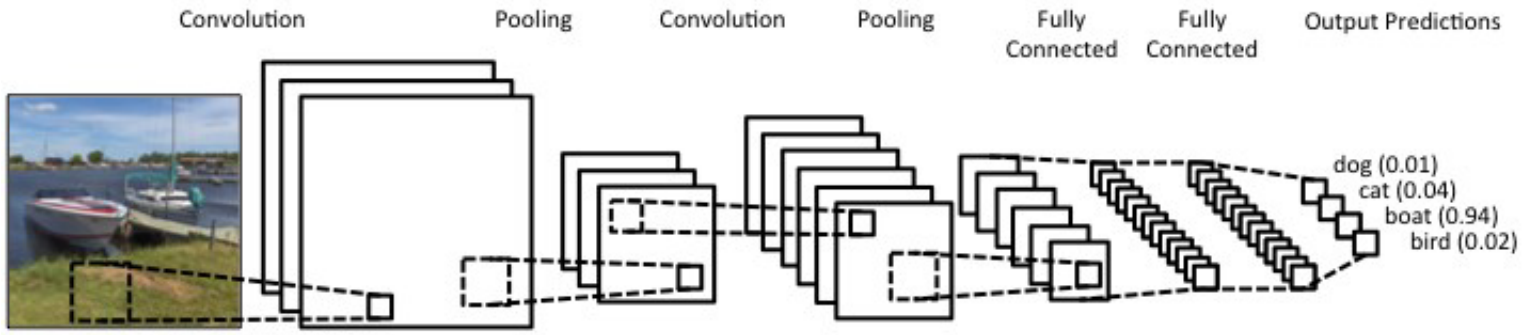
ISFPGA 2017

Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs

Ritchie Zhao[1], Weinan Song[2], Wentao Zhang[2], Tianwei Xing[3], Jeng-Hau Lin[4], Mani Srivastava[3], Rajesh Gupta[4], Zhiru Zhang[1]

1 Electrical and Computer Engineering, Cornell University
2 Electronics Engineering and Computer Science, Peking University
3 Electrical Engineering, University of California Los Angeles
4 Computer Science and Engineering, University of California San Diego

CNN Architecture



- Basics:
  - **Convolutional** (conv) layers in the front
  - **Fully connected** (dense) layers in the back
  - **Pooling** layers reduce the size of **feature maps** (fmaps)

- Enormous computational and memory requirements
  - VGG-19 Network**:** 140 million floating-point parameters and 15 billion floating-point operations per image [1]

[1] K. Simonyan and A. Zisserman. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. *arXiv:1409.15568*, Apr 2015.

CNNs on FPGA?

- **Challenges**
  - CNN weights and fmaps don't fit in on-chip memory
  - Difficult for FPGAs to compete on floating-point throughput
  - Deep learning frameworks for CPU/GPU (e.g. Theano, Caffe, TensorFlow)

- **Opportunities**
  - **Energy efficiency** → deep learning on embedded platforms
  - **Networked FPGA cloud** → overcome performance limitations through scale (MSR Catapult)
  - **Hardware CNN optimizations**
    - Data reordering and tiling (Zhang FPGA'15)
    - Dynamic fixed-point quantization (Qui FPGA'16)
    - Sparse model compression (Han ISCA'16, FPGA'17)

Very Low Precision CNNs

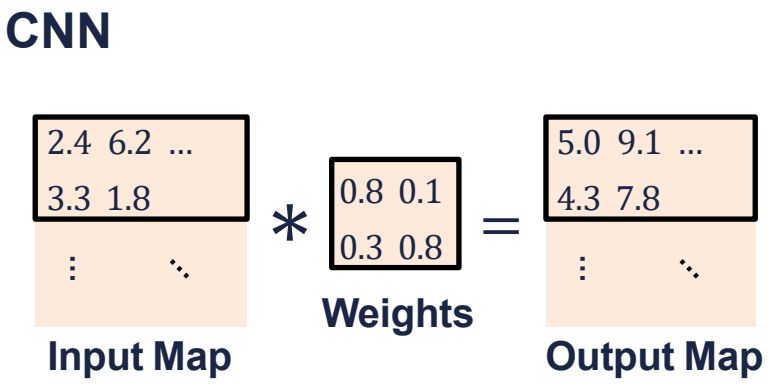
- **ML Research Papers:**

– BinaryConnect	[NIPS]	Dec 2015	
This Model – BNN	[arXiv]	Mar 2016	Near state-of-the-art on MNIST, CIFAR-10 and SVHN
– Ternary-Net	[arXiv]	May 2016	
– XNOR-Net	[ECCV]	Oct 2016	
– Local Binary CNNs	[arXiv]	Aug 2016	Within 3% of state-of-the-art on ImageNet

- **Our Paper:**
  - FPGA implementation of BNN[2] for CIFAR-10 inference
  - New model optimizations and hardware structures
  - Open-source HLS code available

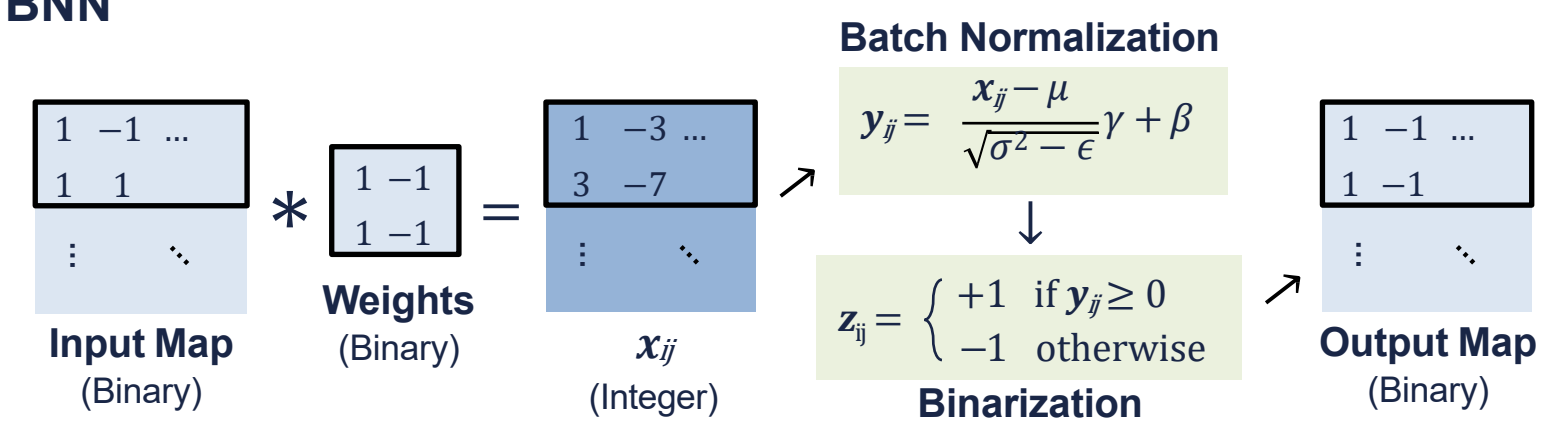
[2] M. Courbariaux et al. **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**. *arXiv:1602.02830*, Feb 2016.

Binarized Neural Networks (BNN)

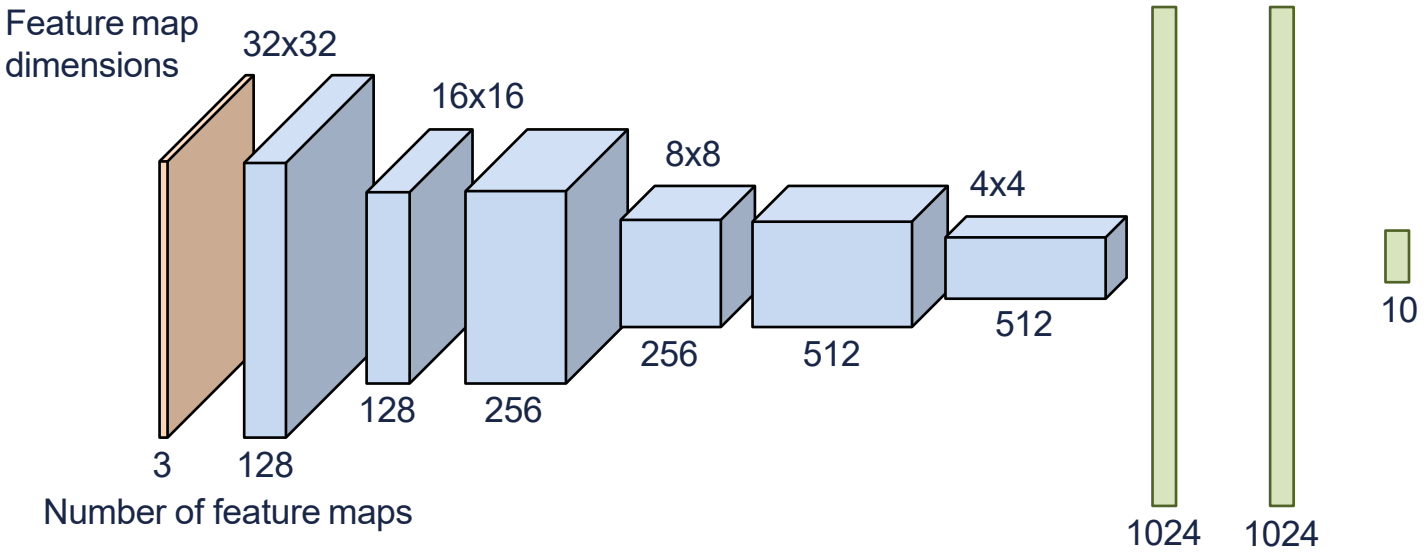


Key Differences

1. Inputs are binarized (−1 or +1)
2. Weights are binarized (−1 or +1)
3. Results are binarized after **batch normalization**



BNN CIFAR-10 Architecture [2]



- 6 conv layers, 3 dense layers, 3 max pooling layers
- All conv filters are 3x3
- First conv layer takes in floating-point input
- **13.4 Mbits total model size** (after hardware optimizations)

[2] M. Courbariaux et al. **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**. *arXiv:1602.02830*, Feb 2016.

Advantages of BNN

1. Floating point ops replaced with binary logic ops

b_1	b_2	$b_1 \times b_2$
$+1$	$+1$	$+1$
$+1$	-1	-1
-1	$+1$	-1
-1	-1	$+1$

b_1	b_2	$b_1 \text{ XOR } b_2$
0	0	0
0	1	1
1	0	1
1	1	0

- Encode {+1,−1} as {0,1} → multiplies become XORs
- Conv/dense layers do dot products → XOR and popcount
- Operations can map to LUT fabric as opposed to DSPs

2. Binarized weights may reduce total model size

- Fewer bits per weight may be offset by having more weights

BNN vs CNN Parameter Efficiency

Architecture	Depth	Param Bits (Float)	Param Bits (Fixed-Point)	Error Rate (%)
ResNet [3] (CIFAR-10)	164	51.9M	13.0M*	11.26
BNN [2]	9	-	13.4M	11.40

* Assuming each float param can be quantized to 8-bit fixed-point

- **Comparison:**
  - Conservative assumption: ResNet can use 8-bit weights
  - BNN is based on VGG (less advanced architecture)
  - BNN seems to hold promise!

2 M. Courbariaux et al. **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**. *arXiv:1602.02830*, Feb 2016.
3 K. He, X. Zhang, S. Ren, and J. Sun. **Identity Mappings in Deep Residual Networks.** *ECCV 2016.*

BNN Hardware Optimizations

Optimizations

1. Quantized the input image and batch norm parameters
2. Removed additive biases (no effect on accuracy)
3. Simplified batch norm and pooling computation

Accuracy Impact

BNN Model	Test Error
Claimed in paper [2]	11.40%
Python out-of-the-box [2]	11.58%
C++ optimized model	**11.19%**
Accelerator	**11.19%**

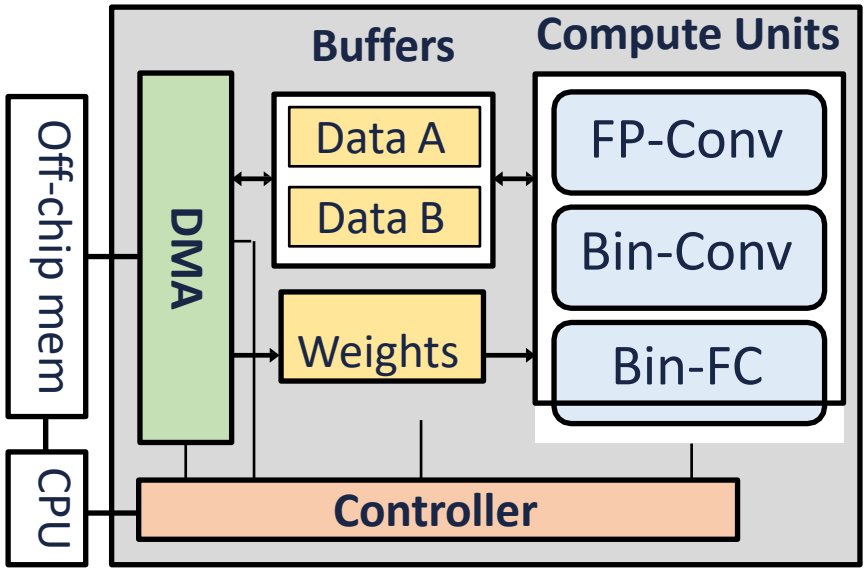
[2] M. Courbariaux et al. **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**. *arXiv:1602.02830*, Feb 2016.

Accelerator Design Goals

- **Target low-power embedded SoC**
  - Design must be resource efficient to fit the FPGA
  - Leverage resource-sharing across layers (execute layers sequentially on a single module)
- **Store all feature maps on-chip**
  - Binarization makes feature maps much smaller
- **Use Xilinx SDSoC to generate RTL from C++ source**

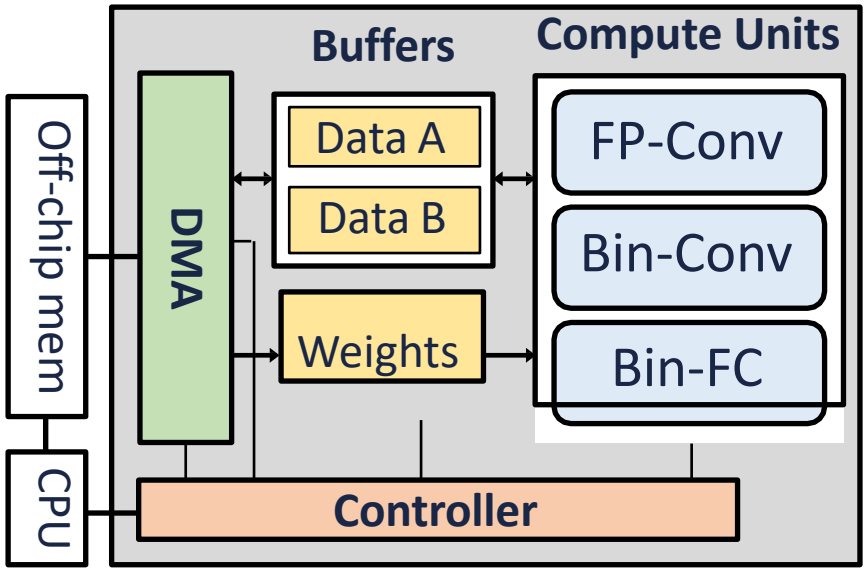
Accelerator Architecture

- **Data buffers (A and B)**
  - Stores feature maps
  - Alternately read from one and write to the other
- **Weight buffer**
  - Store weights and batch norm parameters
  - Reads from off-chip memory

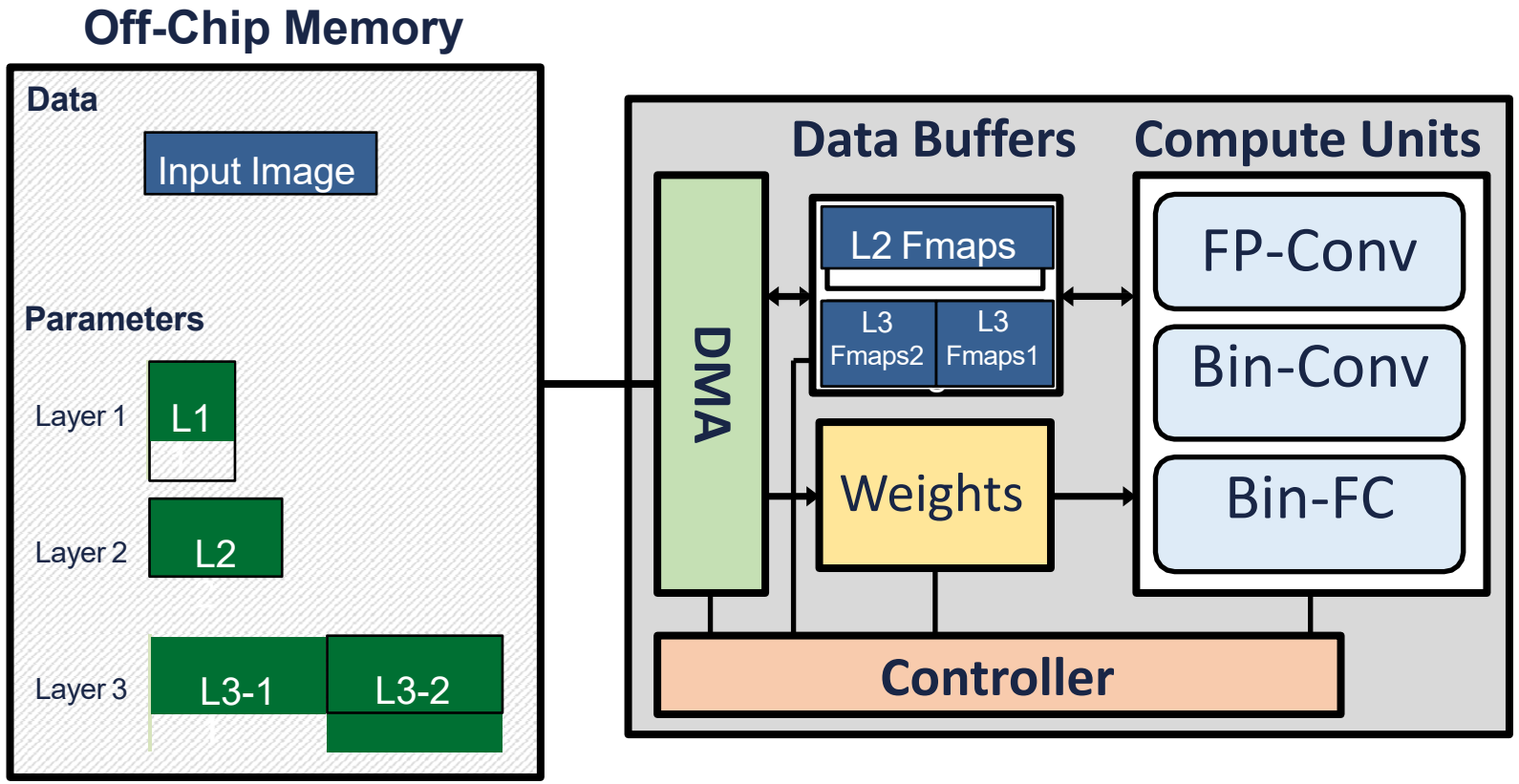


Accelerator Architecture

- **3 compute units**
  - FP-conv → input conv
  - Bin-conv → binary conv
  - Bin-FC → binary dense
- **DMA and controller**
  - Automatically generated by SDSoC



Accelerator Execution Example

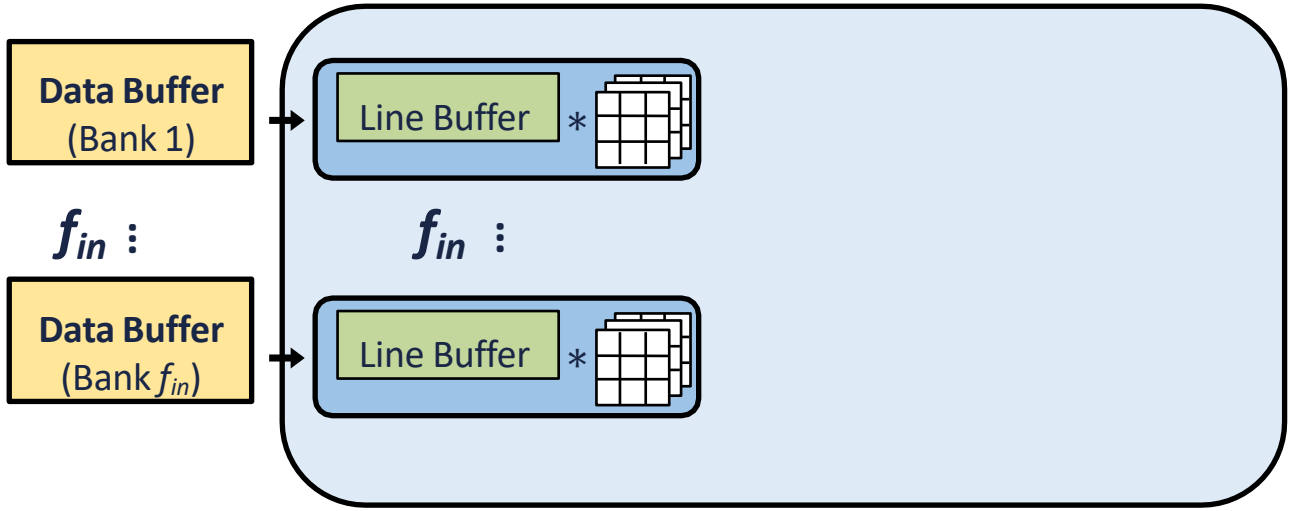


Bin-Conv Unit

- Responsible for the binary conv layers, which take up most of the runtime

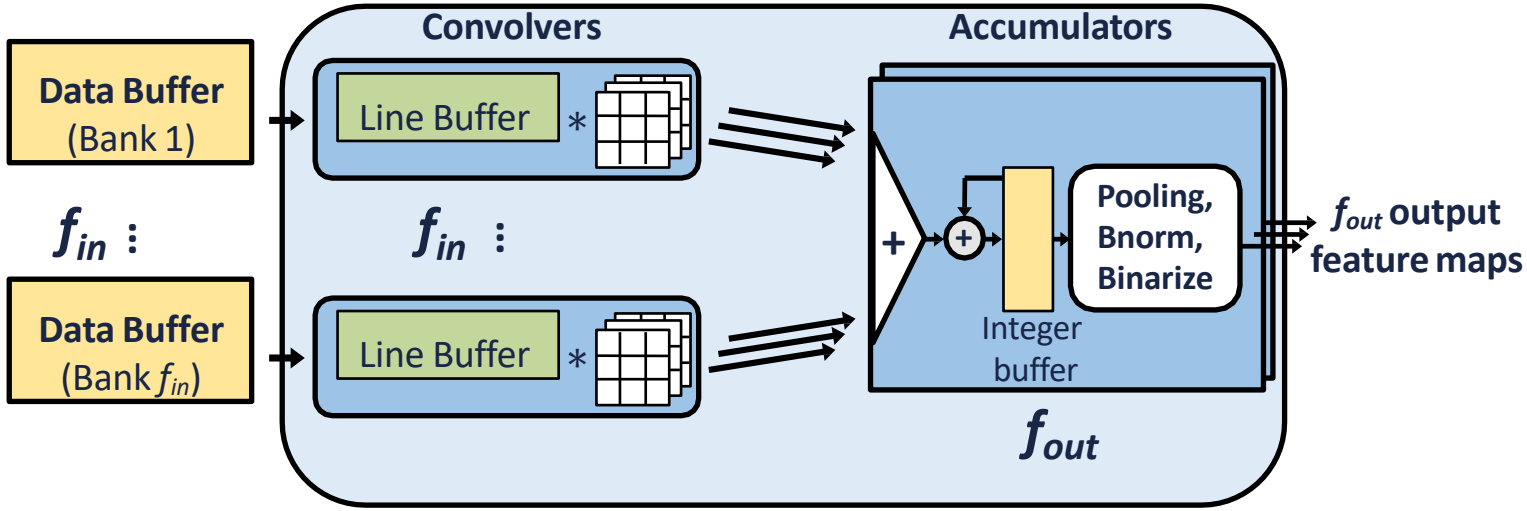
- **Design Goals:**
  - Configurable for different feature map widths and different numbers of feature maps in a layer
  - Scalable performance
  - Exploits parallelism across input/output feature maps and within the pixels of each feature map

Bin-Conv Unit – Input Parallelization



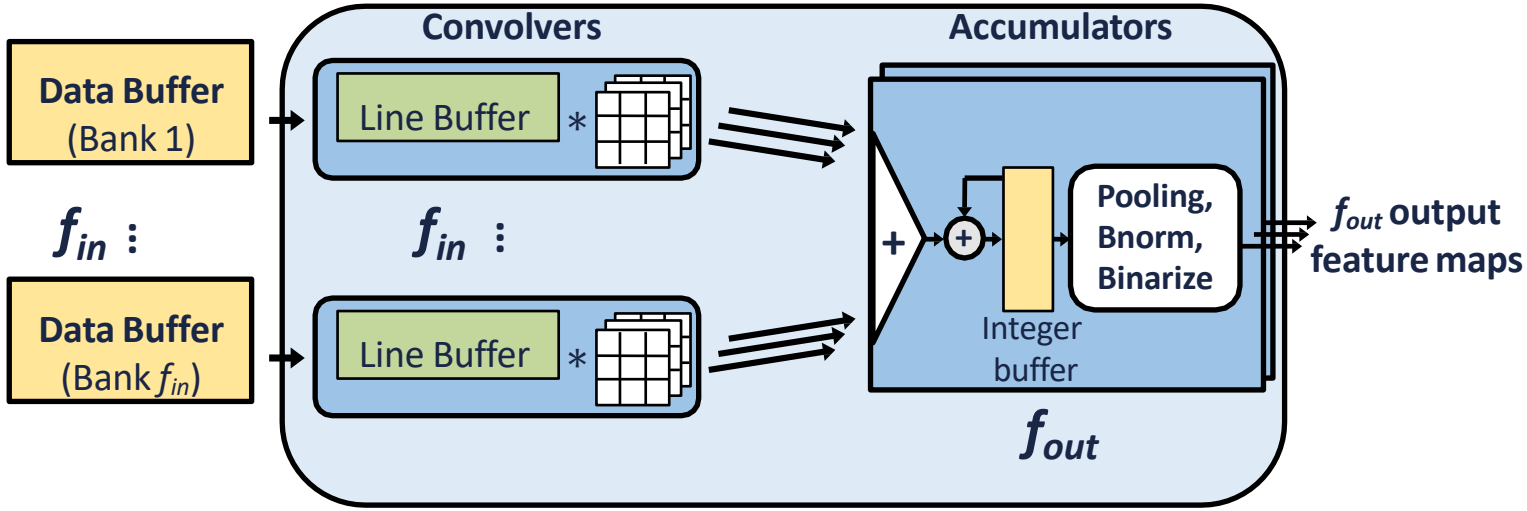
- **f_{in} is the input parallelization factor**
  - Read f_{in} words from the data buffer each cycle
  - The words go to f_{in} **convolvers**
  - Each convolver contains a line buffer and convolution logic

Bin-Conv Unit – Output Parallelization



- **f_{out} is the output parallelization factor**
  - The convolvers generate partial conv sums for f_{out} new maps
  - Accumulate f_{out} output feature maps in parallel
  - Completed feature maps are pushed through pooling, batch norm, and binarization logic

Bin-Conv Unit – Pixel Parallelization



- **The word size is the pixel parallelization factor**
  - Pixels (bits) in a word are processed in parallel
  - **Variable-width line buffer** can be configured for different feature map widths

f_{in} – f_{out} tradeoff

- f_{in} /f_{out} control how many input/output feature maps are processed in parallel
  - f_{in} increases the number of line buffers
  - f_{out} increases the number of integer feature map buffers, pooling units, and batch norm units
  - Increasing f_{in} should be more area efficient

- In the implementation, fix f_{out}= 1 and allow f_{in} to vary

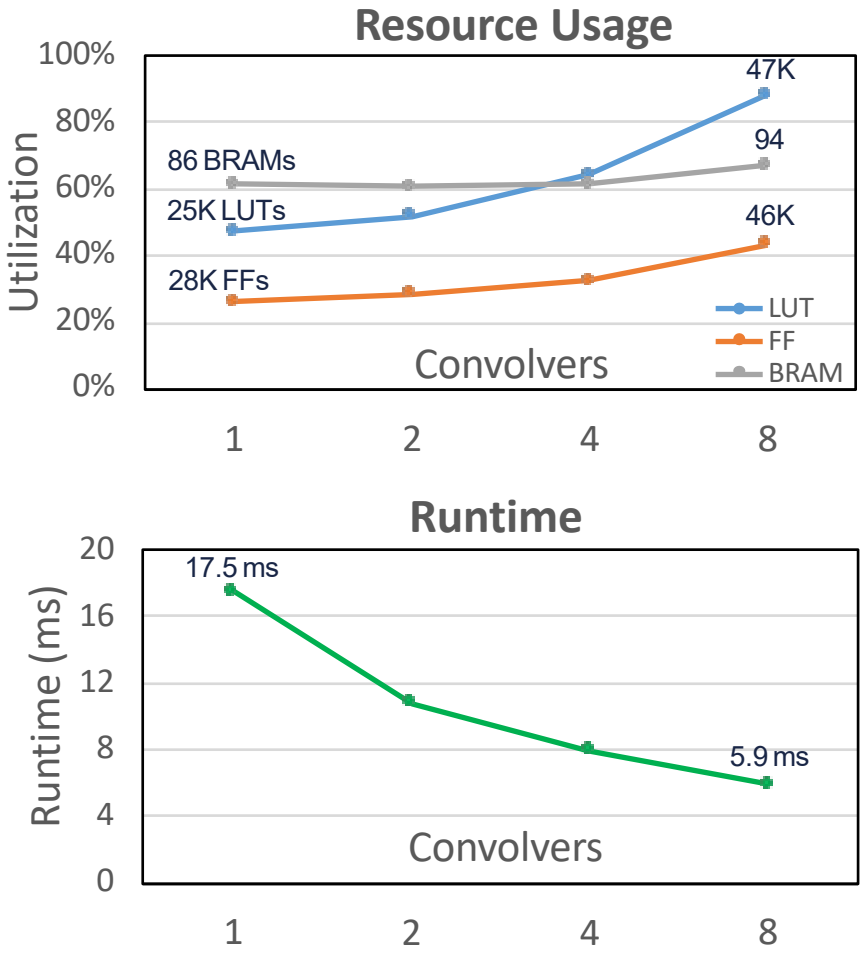
Experiment Setup

- **Target platform:** Zedboard with XC7Z020 FPGA
  - 53K LUTs, 106K FFs
  - 40 BRAMs, 220 DSPs

- **Measurement**
  - Power: Physical meter
  - Resource: Post-route report

- **Platform Comparisons:**
  - **CPU:** Intel Xeon 8-core 2.6GHz
  - **GPU:** NVIDIA Tesla K40
  - **mGPU:** NVIDIA Jetson TK1 embedded board

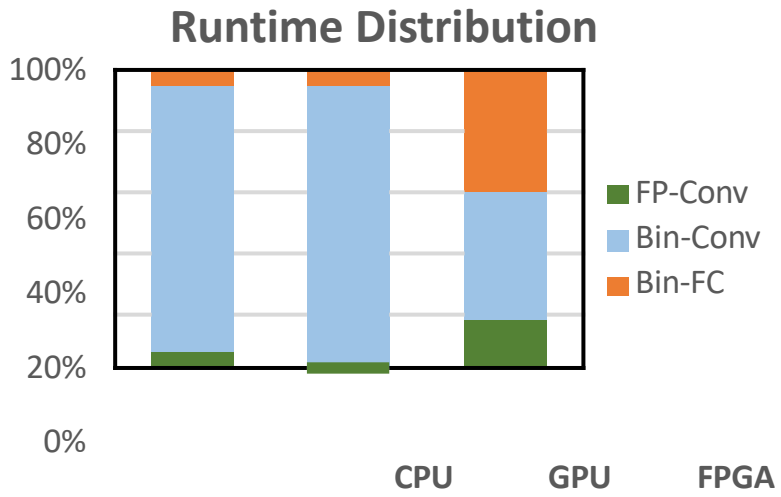
Resource Usage and Scalability



- **Resource:**
  - LUT and FF usage scaled with # of convolvers
  - BRAM and DSP were mostly the same
- **Runtime:**
  - Scales with # of convolvers, some overhead
- **Final Design:**
  - 88% LUT utilization
  - 5.9ms per image
  - 143MHz

Performance Comparison

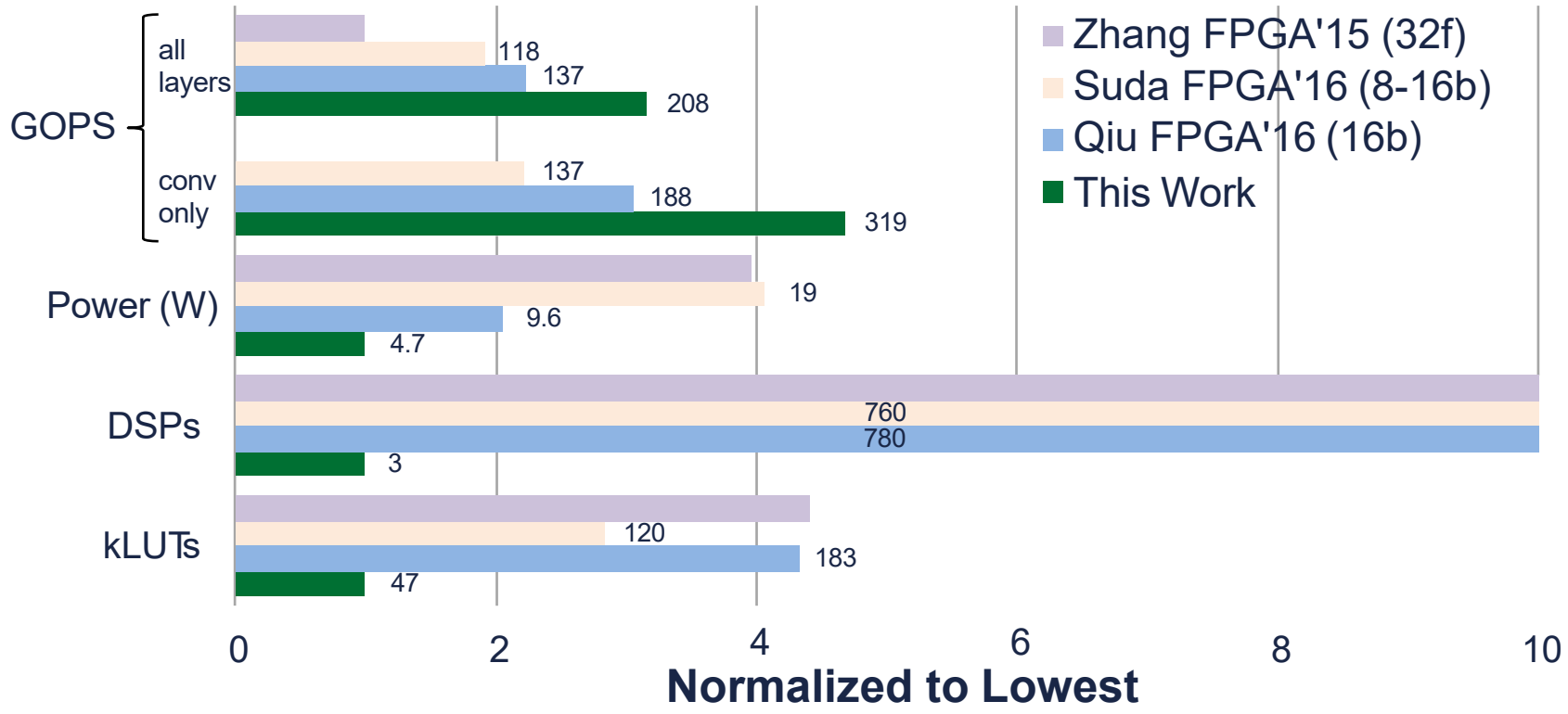
	mGPU	CPU	GPU	FPGA
Runtime per Image (ms)	90	14.8	0.73	5.94
Speedup	1x	6x	120x	15x
Power (W)	3.6	95	235	4.7
Energy Efficiency	3.1	0.71	5.8	36



- **vs. mGPU and CPU**: significant improvement in performance and energy efficiency
- **vs. GPU**: 8x slower but 6x more energy efficient

- Binary conv layers see the most speedup on FPGA
  - First (floating point) conv layer is not heavily parallelized
  - Dense layers are bound by memory throughput

Comparison with CNNs on FPGA



- Comparing CNN and BNN GOPS is not apples-to-apples…
  - But it shows the binary ops we can squeeze out of an FPGA board
  - Our device is considerably smaller than the other works' here

Conclusions for this work

- **Takeaways:**
  - Low precision CNNs on FPGA show great promise
  - There is room for algorithmic improvement in binarized CNNs

- **Code:** https://github.com/cornell-zhang/bnn-fpga
  - '**master**' branch is the debug build (larger area)
  - '**optimized**' branch is the paper build

- **Further Improvements:**
  - '**conv1x1**' branch is a modified BNN with 60% model size reduction and negligible accuracy loss

Today's Lecture

- Representative FPGA-based DNN Inference Works
  - **HLS Optimization**
    - [FPGA'15] Optimizing FPGA-based CNN Accelerator
  - **Winograd, OpenCL for FPGA flow**
    - [FPGA'17] OpenCL Deep Learning Accelerator
  - **Low Bit-width Neural Networks**
    - [FPGA'17] Accelerating Binarized CNN
  - **Real-time CNN Inference in Embedded Systems**
    - [DAC'19] Real-time Object Detection with SoC FPGAs

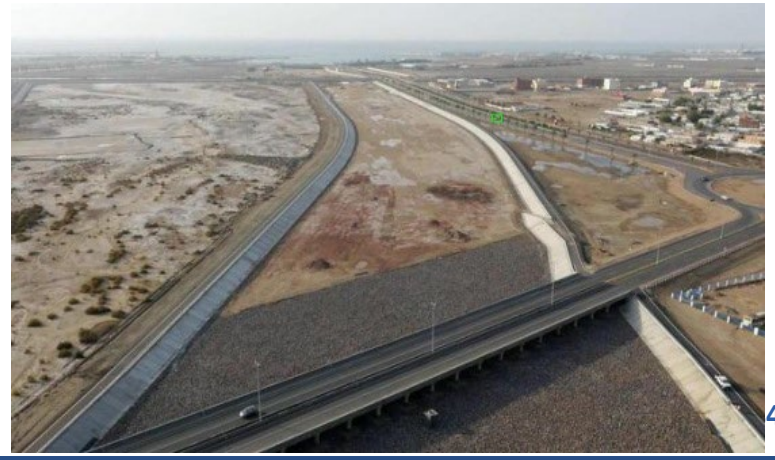
Case Study: Real-Time Object Detection with SoC FPGAs

Task: Object detection in images (videos) captured by drones (DJI dataset)

- Each frame has one single object with a bounding box
- To detect the same single object from training data







DAC System Design Contest (DAC-SDC)

FPGA platforms:

- **PYNQ-Z1 board (ZYNQ-7000)**
- Ultra96 board (ZYNQ UltraScale+)
- KV 260 board (ZYNQ UltraScale+, more DSPs)

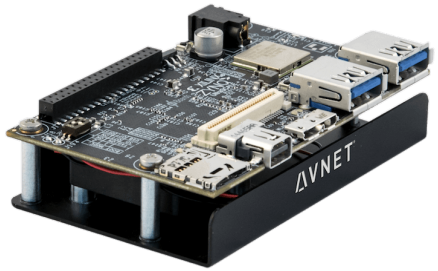
Logic	53200 LUTs
DSP	220 Slices
Memory	630 KB of RAM 512 MB DDR3
Board Power	**2.1W – 2.4W**

GPU Platform:

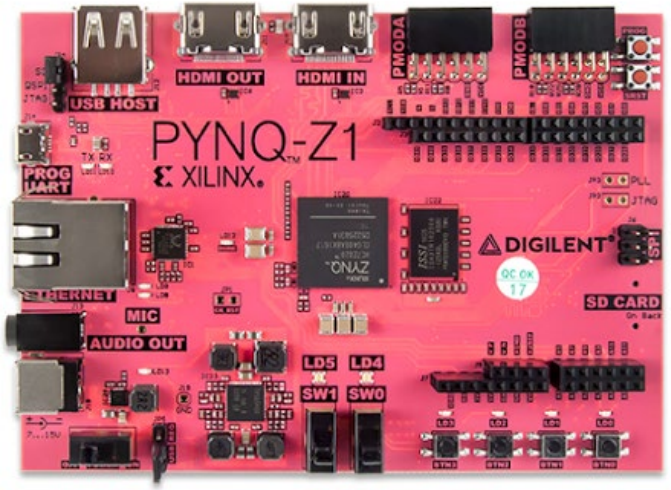
- NVIDIA Jetson Nano board

NVIDIA Jetson Nano

AMD KV 260



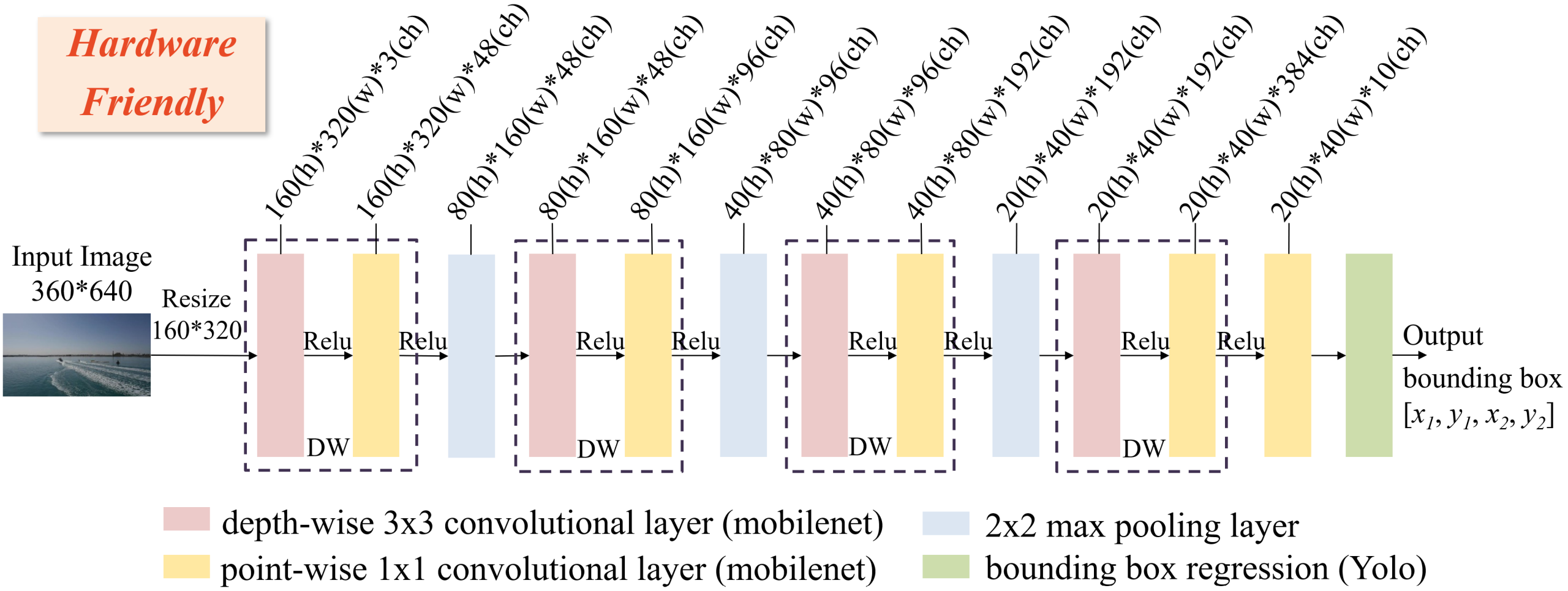
Ultra96



PYNQ-Z1

Proposed NN model – customized from MobileNet [1] & YOLO [2]

Hardware Friendly



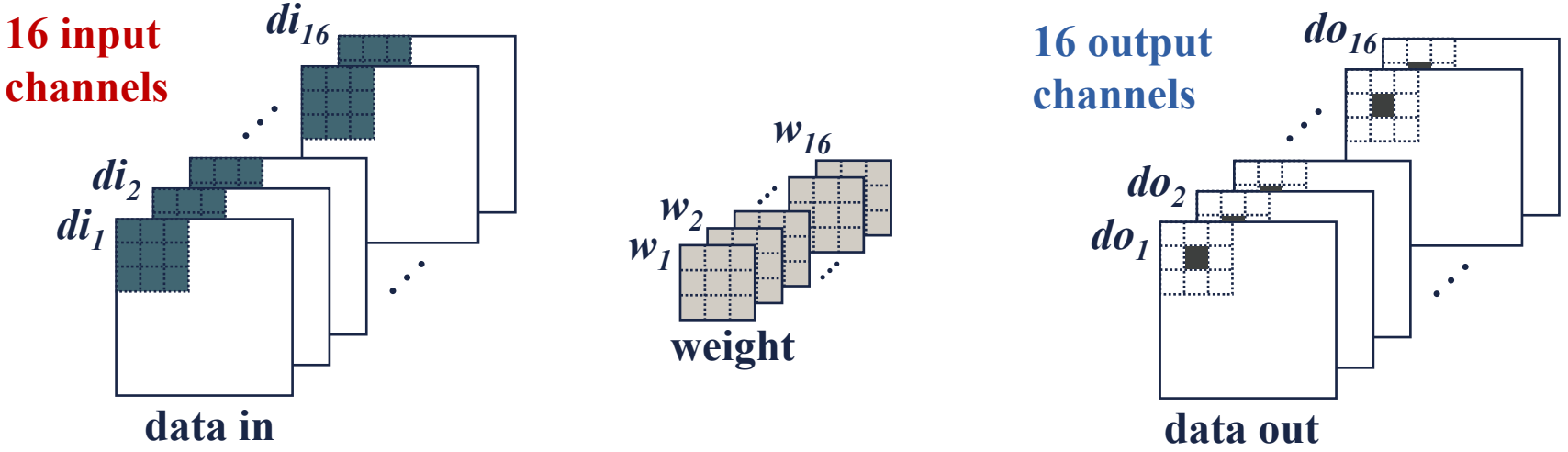
[1] *Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).*
[2] *Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." arXiv preprint (2017).*

Customized FPGA Implementation

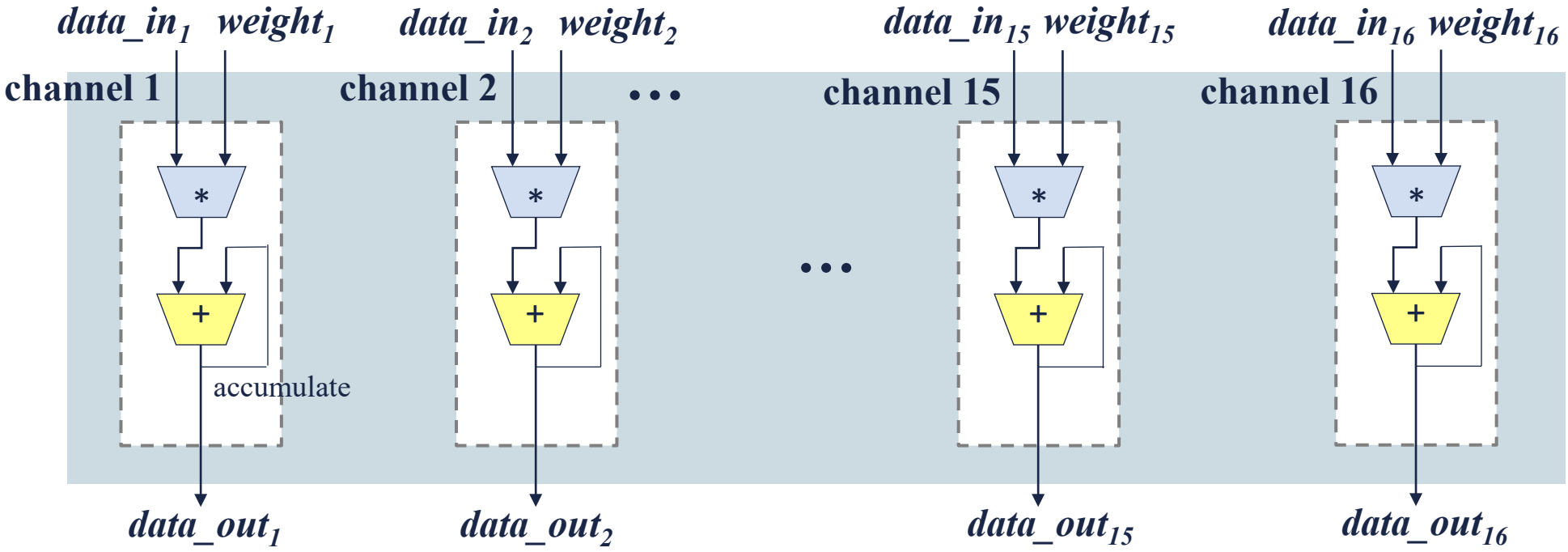
- **High-Level Synthesis** to generate Verilog design
- **Quantization**: 8-bit fixed point weights, 16-bit fixed point feature map
- **IP Design & IP Reuse**
  - One instance per IP
  - One IP is reused by all the layers of the same type
- **Data Reuse**
  - IP level and feature map level
- **Latency Optimization**
  - Image and feature map partition
  - Fine-grained buffer schedule and IP pipeline

All IPs	Resources	
	LUT	DSP
Image Norm	1200	0
Depth-wise Conv 3x3	5000	18
Point-wise Conv 1x1	12000	128
Max Pooling	3500	4
Set Bias	800	4
ReLU	1000	4
Data Loading	800	24
Bounding Box Regression	15000	35

Depth-Wise Convolutional 3x3 IP

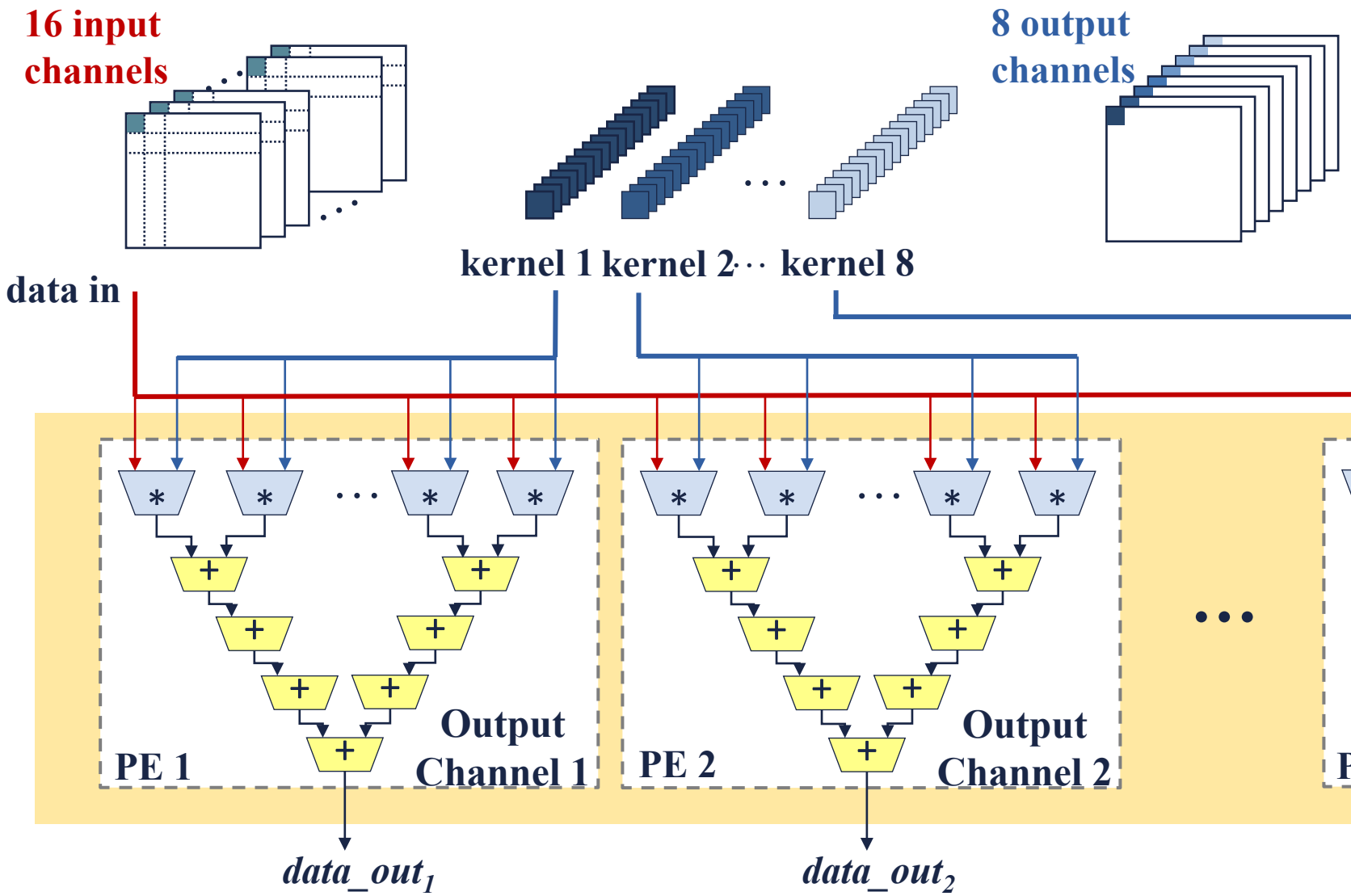


Parallelism: **16**
Multipliers: **16**
Adders: **16**
DSPs: **16**



Accumulate 3x3 times for one output

Point-Wise Convolutional 1x1 IP



Parallelism: **16*8 = 128**
Multipliers: **16*8 = 128**
Adders: **15*8 = 120**
DSPs: **16*8 = 128**

Overall System Diagram

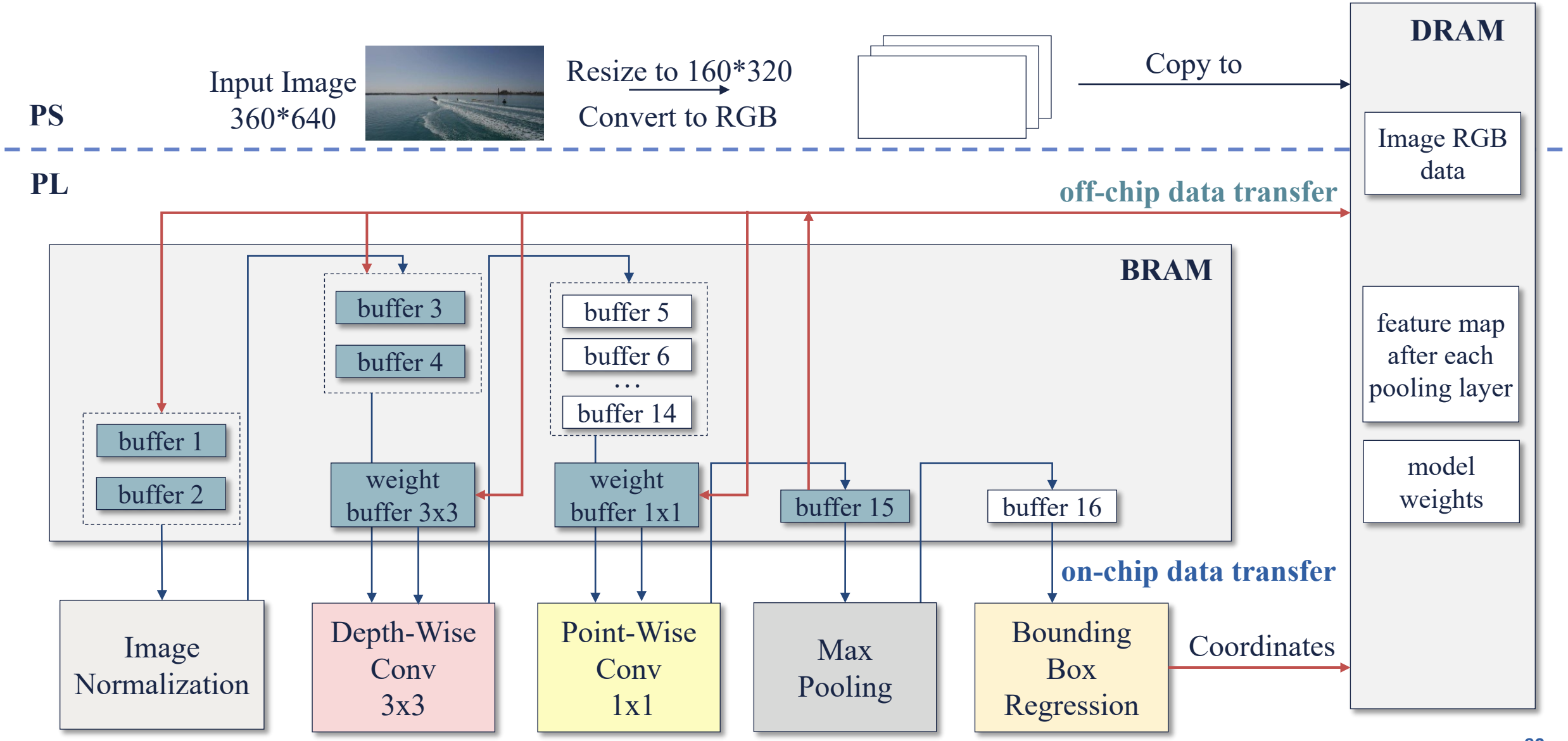
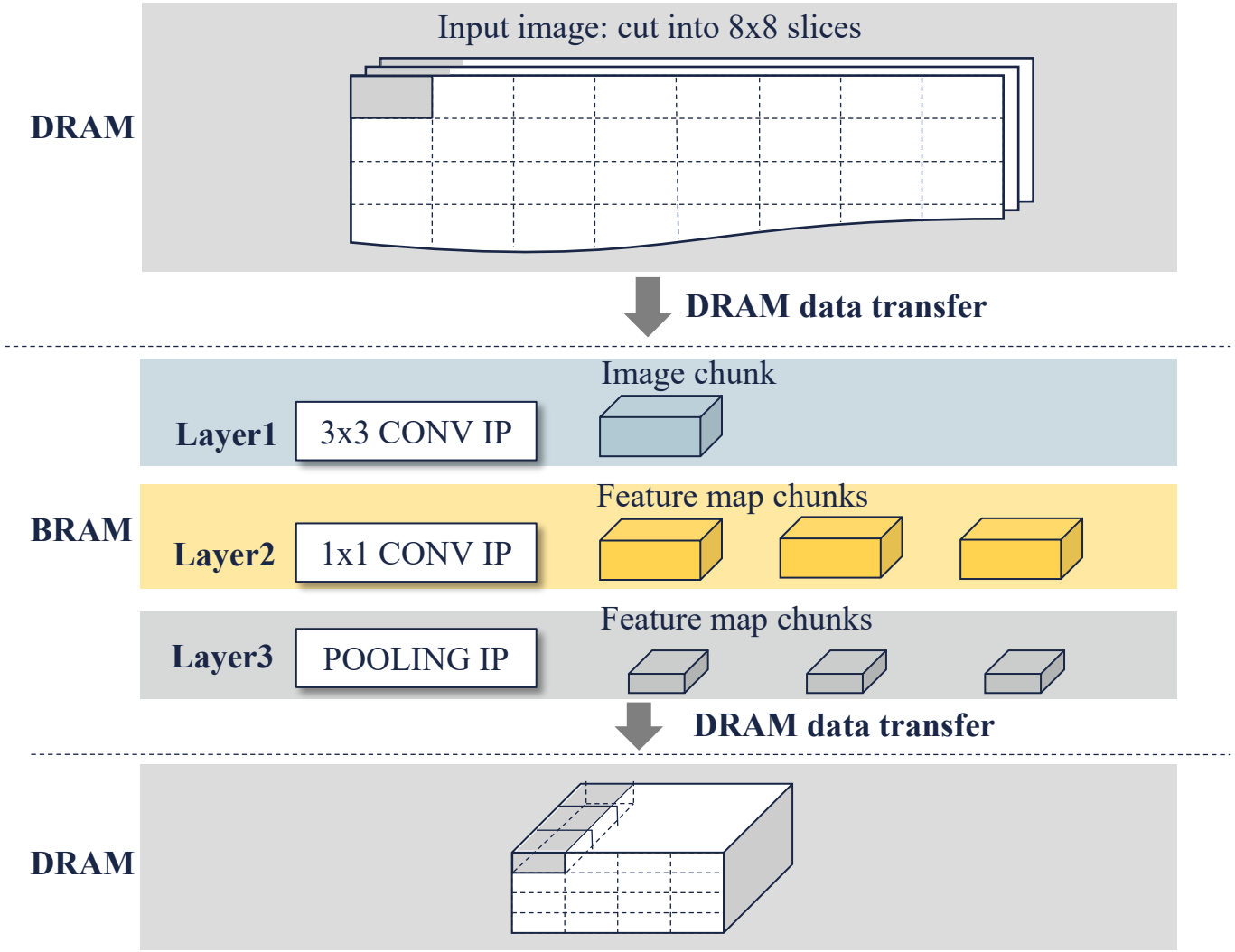
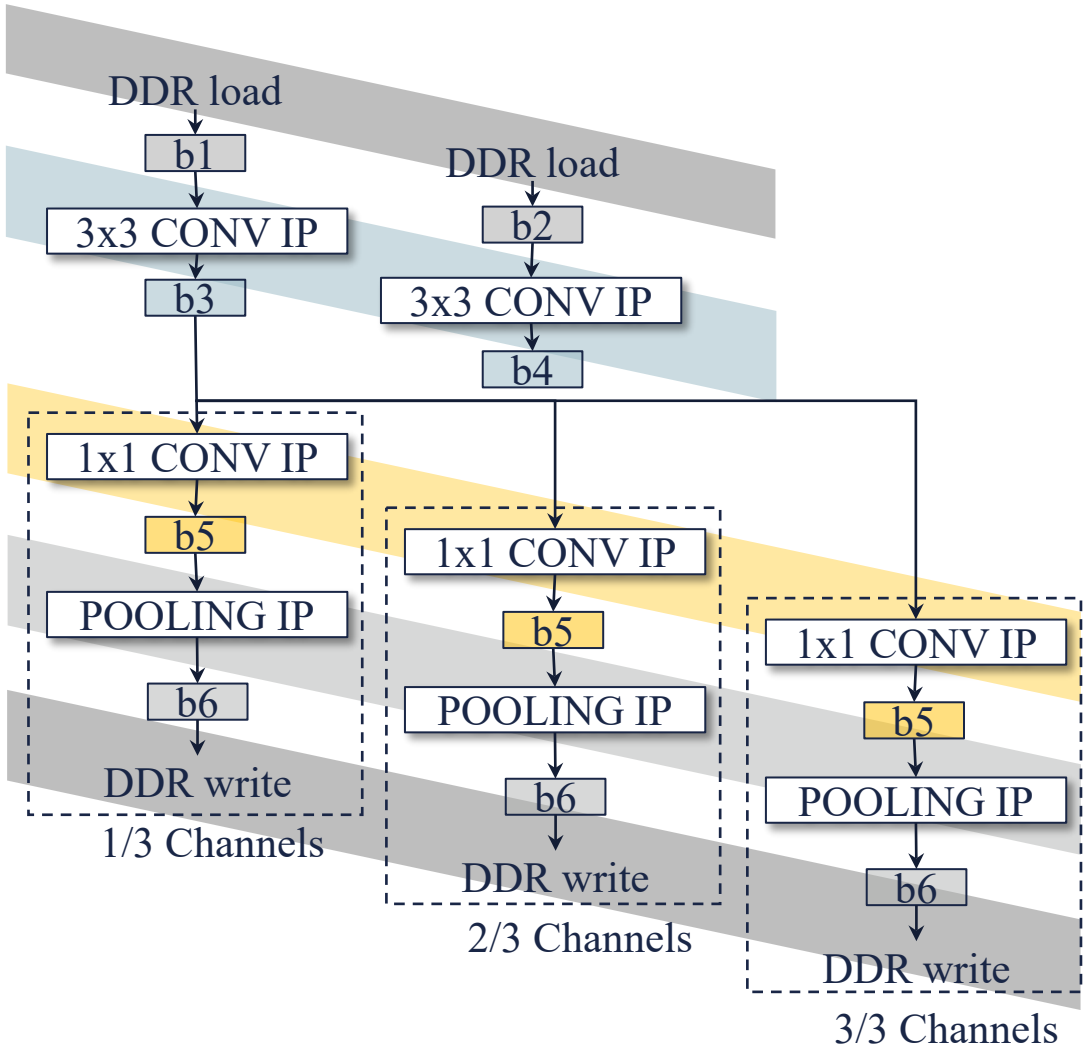


Image partition, fine grained buffer scheduling, IP pipeline



DRAM data transfer pattern



Fine Grained IP/Buffer Scheduling

Experimental Results (Won the 3rd place in 2018)

iSmart team, SkyNet (And 1st place in 2019)

NN Structure	IoU		Latency (ms)		Speed (FPS)		Power (W)		Energy (Wh)		Resource Utilization			
	GPU	FPGA	100MHz	150MHz	100MHz	150MHz	100MHz	150MHz	100MHz	150MHz	LUT	BRAM	DSP	FF
12layer 64ch	26.6%	26.3%	13	--	20	--	2.35	--	--	--	75.7%	77.5%	59.7%	15%
12layer 128ch	38.6%	38.9%	34	-	20	--	2.35	--	--	--	76.7%	94.3%	76.2%	17%
12layer 256ch	45.6%	45.5%	120	--	8.3	--	2.35	--	--	--	77.5%	95.4%	81.8%	18%
12layer 384ch	**57.3%**	**57.3%**	**175**	**131**	**5.7**	**7.6**	**--**	**2.59**	**--**	**5.12**	**76.7%**	**95.4%**	**84.2%**	**22%**
14layer 512ch	60.4%	60.4%	256	194	3.7	5.0	--	2.82	--	8.13	76.7%	95.4%	84.2%	23%

- C. Hao, et al. FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. DAC 2019.
- https://github.com/TomG008/SkyNet

DAC System Design Contest (DAC-SDC) 2018

A live demo where it tracks a toy train (2018)

Our design won 3rd place in 2018
(and 1st place in 2019)

EECS 298:
System-on-Chip Design

Sitao Huang, sitaoh@uci.edu