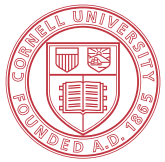
ASA-DAC 2018 Tutorial

Accelerating Deep Neural Networks on FPGAs

Zhiru Zhang[1], Deming Chen[2]

[1]: School of ECE, Cornell University
[2]: ECE, University of Illinois at Urbana-Champaign

1/22/2018



Cornell University

ILLINOIS

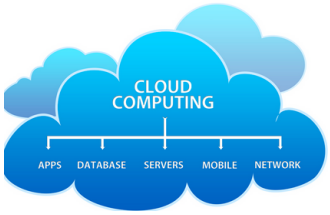
The Outline

- Introduction
- Fundamentals of Deep Neural Networks (DNNs)
  - Classification: Linear Classification, Multi-Layer Perceptron
  - Convolutional Neural Network (CNN)
  - Recurrent Neural Network (RNN): LSTM, GRU
- DNN Accelerations with FPGAs
- Concluding Remarks

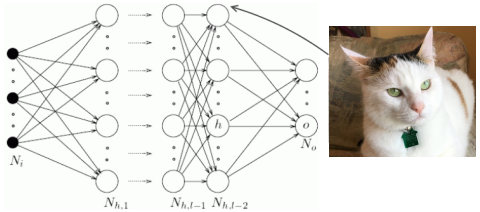
Why AI? Why Now?

- Big Data
  - Large unstructured data sets flood us everyday
  - Examples: Facebook, Google, …

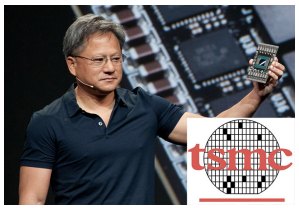
- Cloud Computing
  - Machine learning becomes affordable

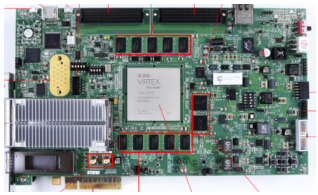


- Deep Neural Networks
  - Reaching or approaching human intelligence



- Hardware advances
  - GPU, FPGA, TPU, …





Overview of the Machine Learning Process

- Training: Train the desired ML model, let the machine **learn**
  - Computationally intensive
  - Huge amount of data, long training time
    - ImageNet: millions of training images. Weeks to train deep neural network models (e.g., VGG16)
  - GPUs, ASICs …
- Inference: Infers things about new data based on the trained model
  - Also computationally intensive
  - Real time requirement most of the times -- Mobile devices, IoTs
  - ASICs, GPUs, FPGAs …

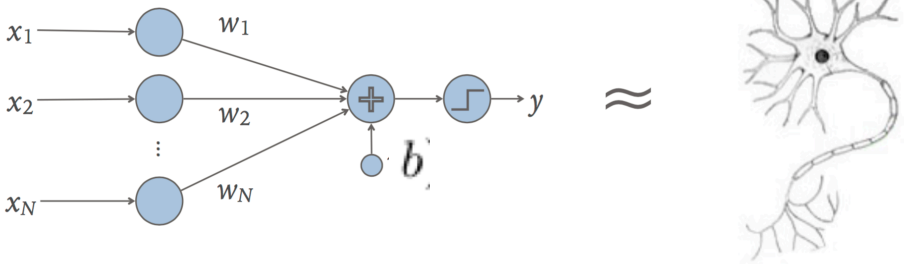
Classification

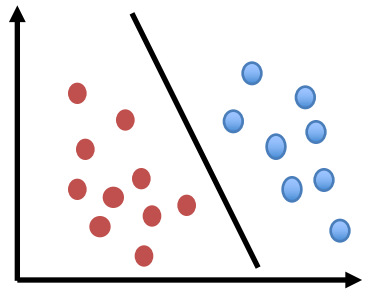
- Identify to which of a set of categories an observation belongs
  - Or rather, which category is the most dominant

- In general, it consists of the following steps
  - Provide examples of classes (**Dataset**, usually labeled for training/test)
  - Make models for distinguishing each class (**Training process**)
  - Assign a new input to a class (**Classification**)

Linear Classifier

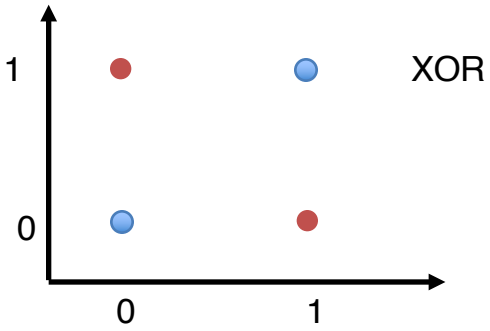
- Perceptron $y = sign(\sum_{i=1}^{n} w_i x_i + b)$



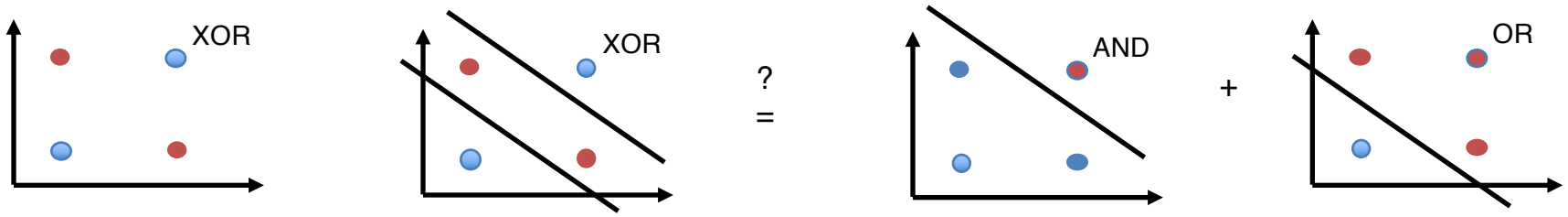


[Source: UIUC CS598PS Fall2017 https://courses.engr.illinois.edu/cs598ps/fa2017]

- How about non-linear separable?



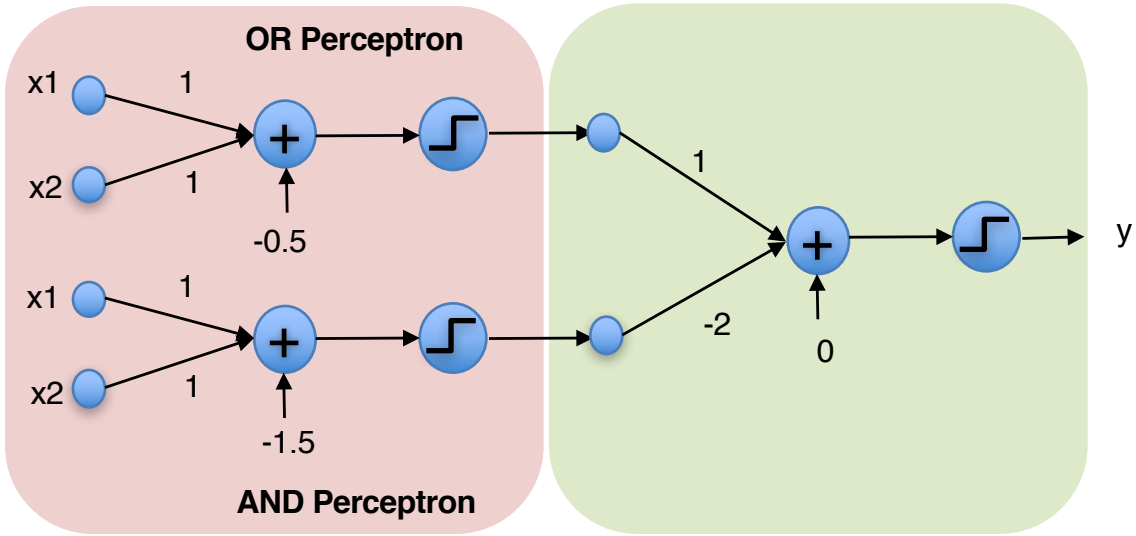
Multi-Layer Perceptron



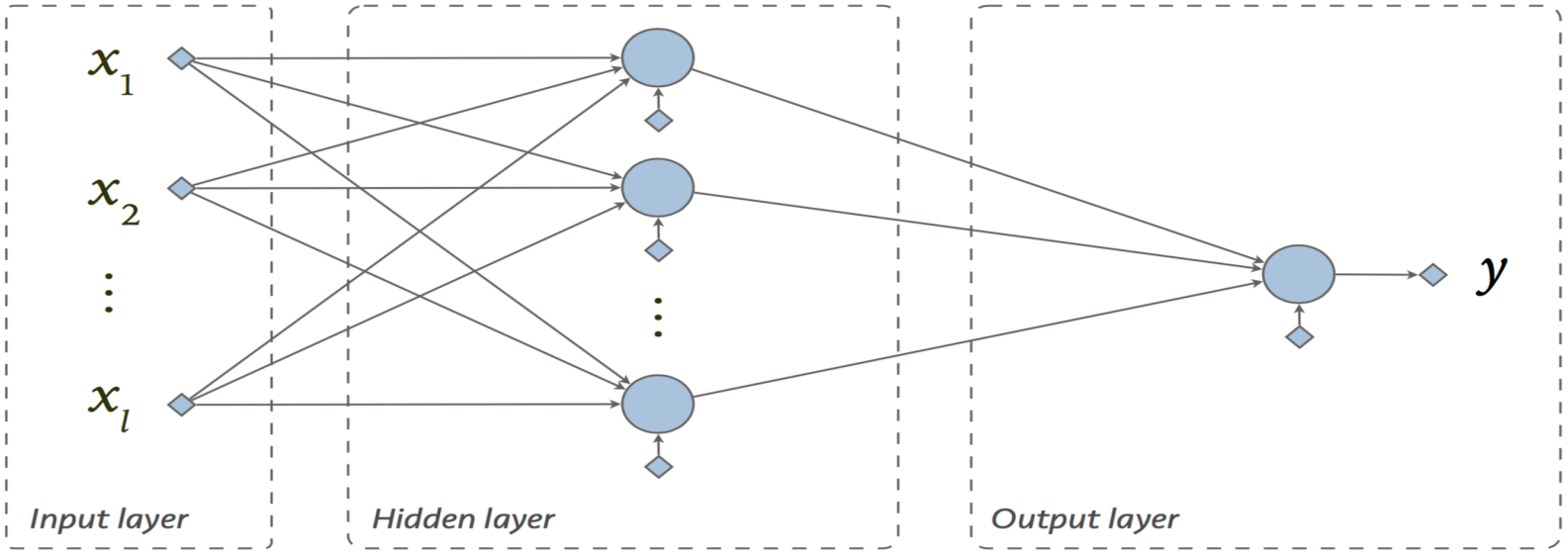
A xor B =
(A or B) – 2(A and B)

Nonlinear

This is a neural network!



Multi-layer Perceptron Is a Neural Network



[Source: UIUC CS598PS Fall2017 https://courses.engr.illinois.edu/cs598ps/fa2017]

- 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

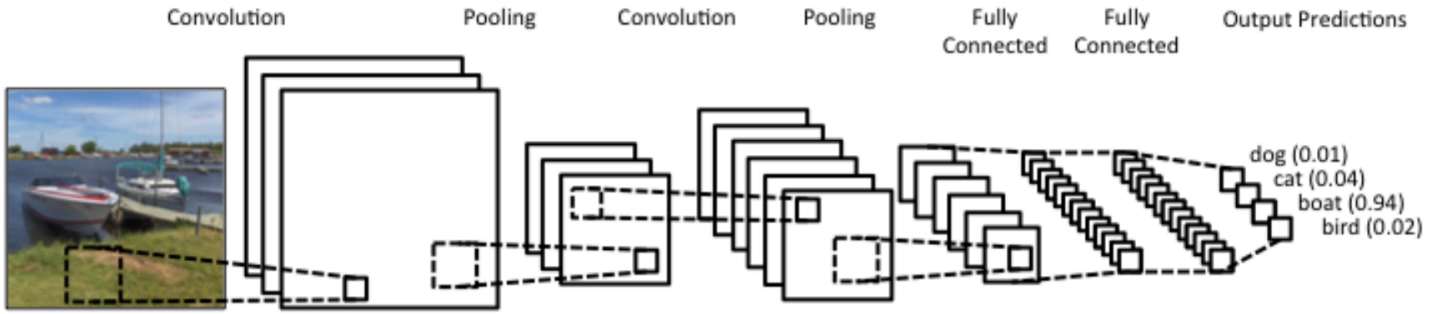
Different Types of Neural Networks

- There are many variations and complications of MLP → different neural networks
  - Number of layers, e.g., deep neural networks
  - Nodes and edges, e.g., CNN, RNN
  - Input features, e.g., vector, image, etc.

- Each variation can become a new structure -- active research area

Convolutional Neural Network (CNN)

- Input is an image, 2D matrix (MLP is a vector)
- A pipeline of connected layer



  - Each layer takes in a vector of 2D **feature maps** (fmaps), transforms them, and outputs a vector of new fmaps
  - **Convolutional** (conv) layers in the front; **Fully connected** (dense) layers in the back; **Pooling** (pool) layers reduce the size of fmaps

What about Time?!

- So far, all neural networks only consider the current input

$$y = g(x)$$

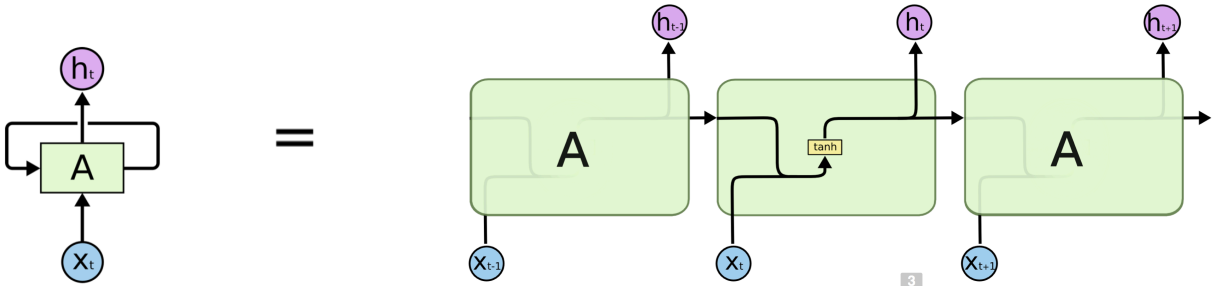
- We assume all inputs are independent; this may not be good assumption for certain tasks
  - Time sequence, e.g., speech, handwriting

- Input is a sequence -- how to learn the sequential information?
  - Make the neural network consider the previous output together with the current input

$$y_t = g(y_{t-1}, x_t)$$

Recurrent Neural Network (RNN)

- A class of neural networks with backward edges
  - Can learn sequential information



[Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs]

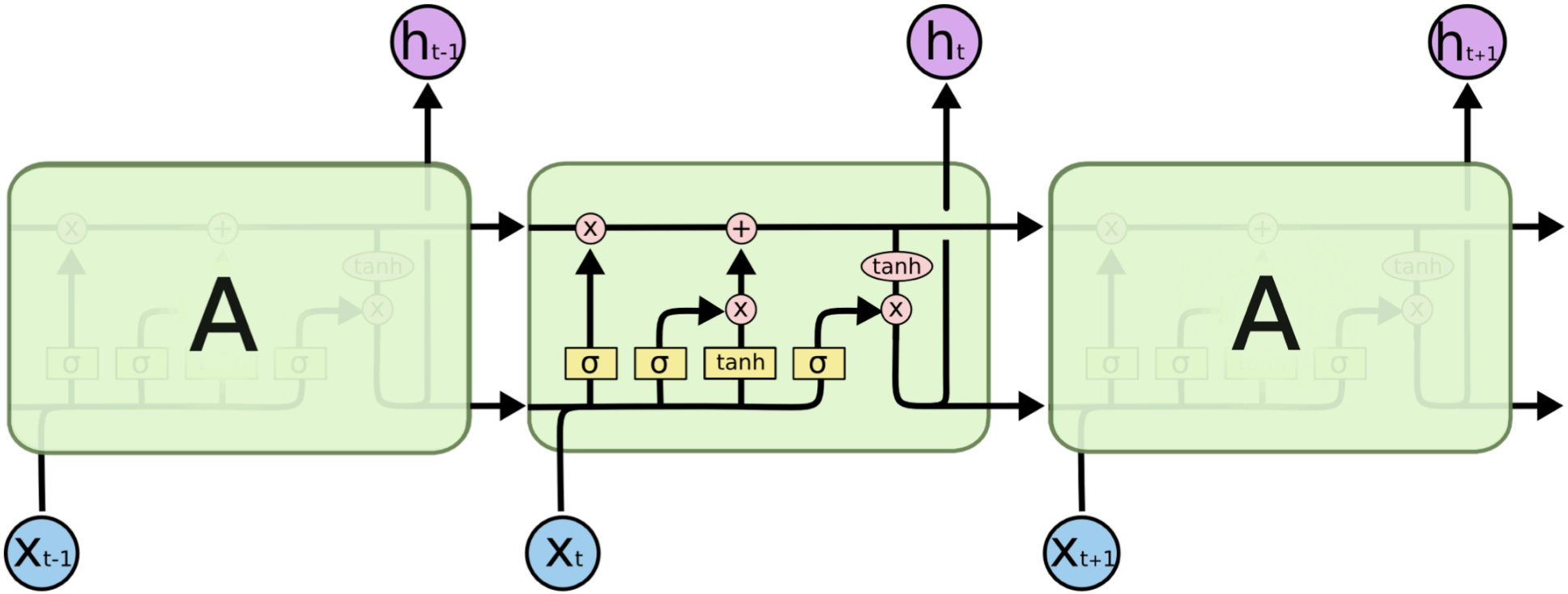
  - **Indirectly** factor in **all** previous inputs
  - Vanishing effect: The input long time ago has very little effect
  - Only short term memory, like myself :-)

Long Short Term Memory (LSTM)

- Can a neural network learn the patterns of both short-term and long-term memories, like human memory?

- What if
  - Instead of only using the current input (x_t) and previous output (y_{t-1}) as inputs, we also add a **memory input**?
  - If the memory brings information from a long time ago, it will **directly** affect the current output, i.e., achieving long term memory
  - We have a unique name to describe such a network: LSTM
  - LSTM effectively removed the vanishing effect of RNN

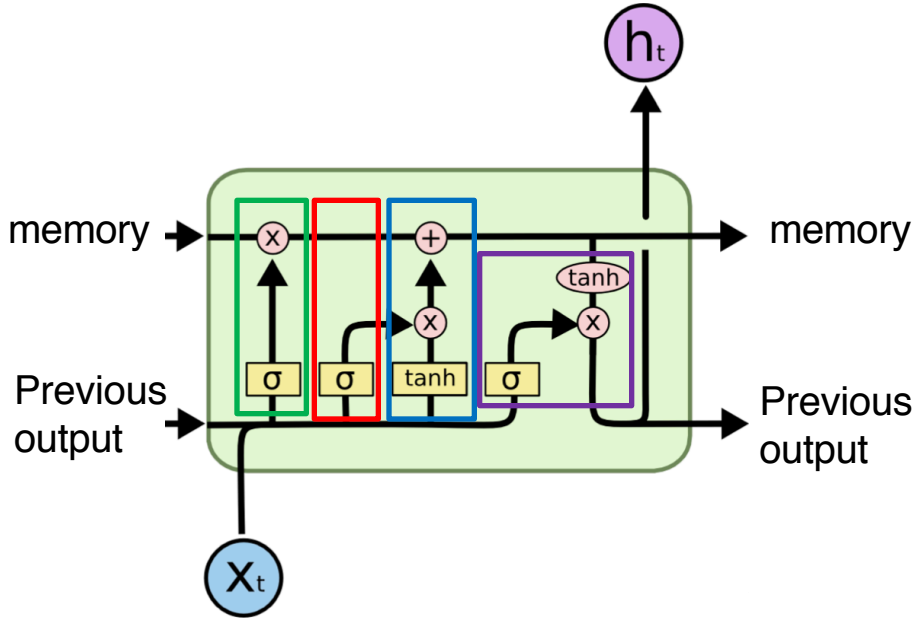
Long Short Term Memory (LSTM)



[Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs]

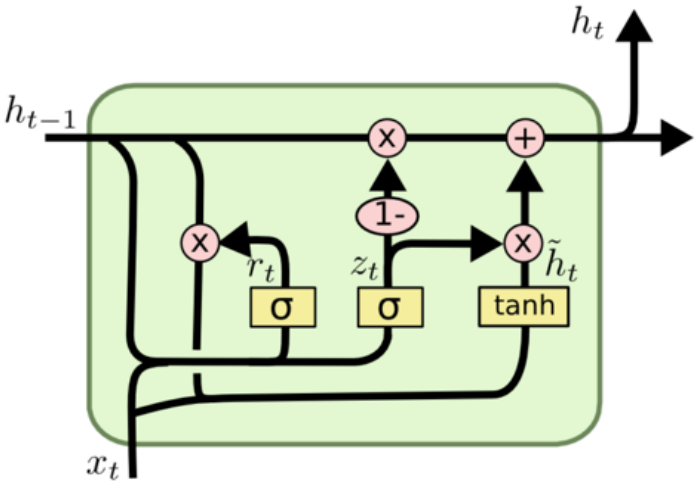
LSTM: A Closer Look

- Forget gate
- Update gates
  - Input gate
  - New candidate
- Output gate



Gated Recurrent Unit (GRU)

- A variant of LSTM
  - Combines the forget and input gates into a single “update gate”
  - Merges the memory state and output state
  - Simpler than LSTM, becoming more and more popular
  - For some data set, it has similar performance compared with LSTM

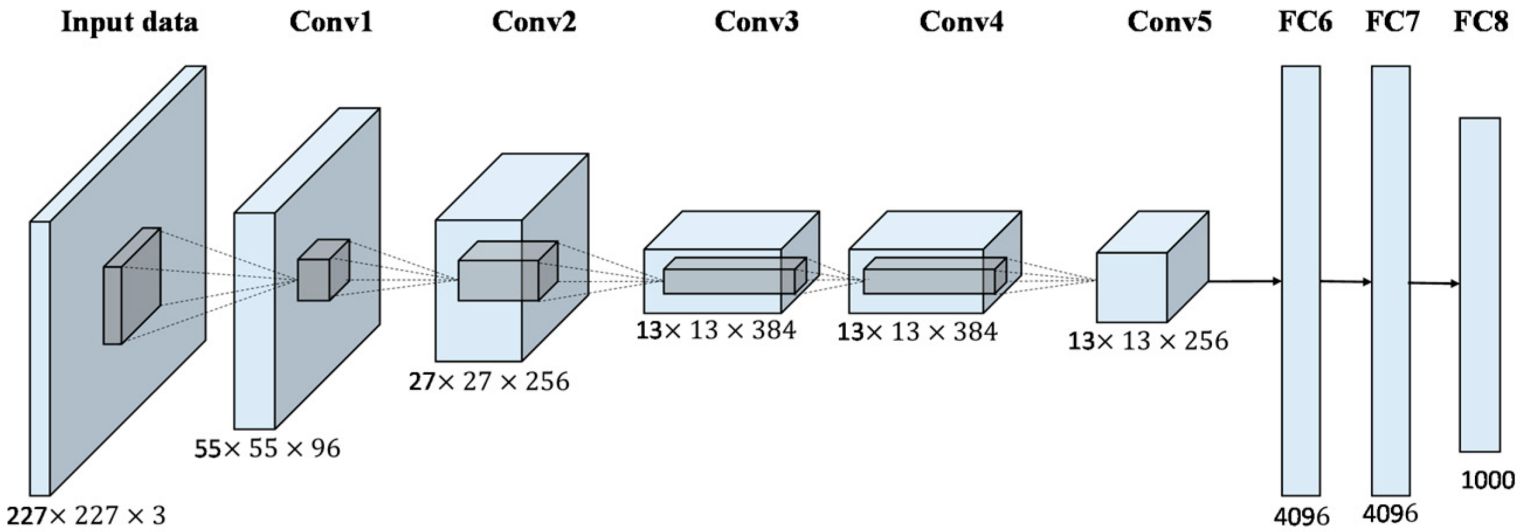


- x_t: input vector
- h_t: output vector
- z_t: update gate vector
- r_t: reset gate vector

[Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs]

Now, Let's Talk about Computation!

- Consider AlexNet – a relatively "old" CNN architecture
  - Pooling layers not shown



[Source: http://www.mdpi.com:8080/2072-4292/9/8/848]

Why Is Deep Learning Compute Intensive?

- Notice how much compute is involved in AlexNet:
  - 1.44T floating-point operations
  - **90% in convolution layers**
- Notice how data need to be loaded:
  - 61M weights
  - **97% in fully connected layers**
    - Not necessarily true in new networks, where the portion of dense layers is reduced

Layers	Weight (M)	FP OPs (M)
Conv1	0.03	210.83
Conv2	0.31	447.90
Conv3	0.89	299.04
Conv4	0.66	224.28
Conv5	0.44	149.52
FC1	37.76	75.50
FC2	16.79	33.55
FC3	4.10	8.19

Understanding Deep Learning Computations

- AlexNet is not state of the art
- Many new networks are deeper and bigger!
  - Even more computation needed

	AlexNet	VGG	GoogleNet	ResNet
Year	2012	2014	2014	2015
Top-5 err	16.4%	7.3%	6.7%	3.57%
Layers	8	19	22	152
# of Conv	5	16	21	151
# of FC	3	3	1	1
Model Size	233MB	548MB	51MB	230MB

Compute Acceleration for DNNs

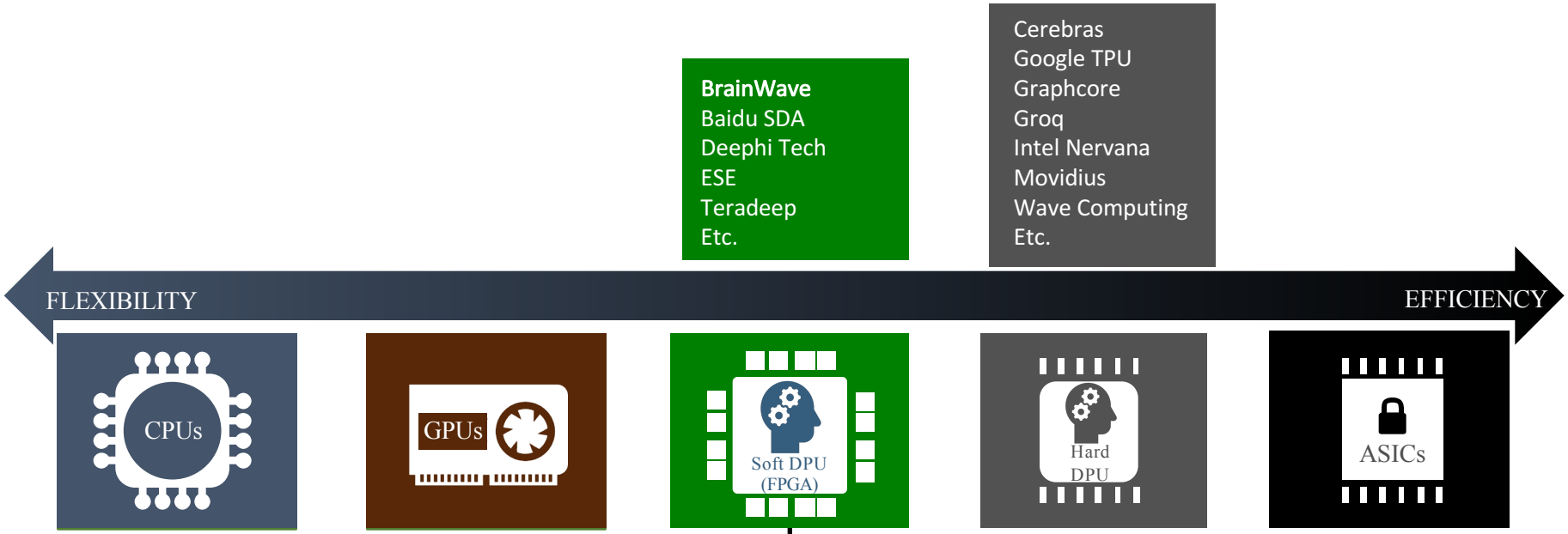
- **Current practice**
  - DNNs training and inference are done on clusters of CPUs/GPUs
  - Computation is typically done in floating-point with large batch sizes

- **Limitations**
  - CPUs/GPUs have high power consumption and poor energy efficiency, and are unsuitable in many settings (mobile devices, IoT, drones)
  - Large batch sizes are unfeasible for real-time apps due to latency

Compute Acceleration for DNNs

- Emerging solutions
  - **Hardware specialization**: use special-purpose DNN processing units optimized for a small batch size
  - **Data quantization**: use low-precision fixed-point or customized floating-point types
  - **Model compression**: reduce redundancy in weights and connections
  - **Algorithmic transformation**: FFT, Winograd, etc.

Industry Landscape of DNN Acceleration



Key advantages of soft DNN processing units (DPUs)

Performance

- Excellent inference performance at low batch sizes
- Ultra-low latency serving on modern DNNs
  - >10X lower than CPUs and GPUs
- Scale to many FPGAs in single DNN service

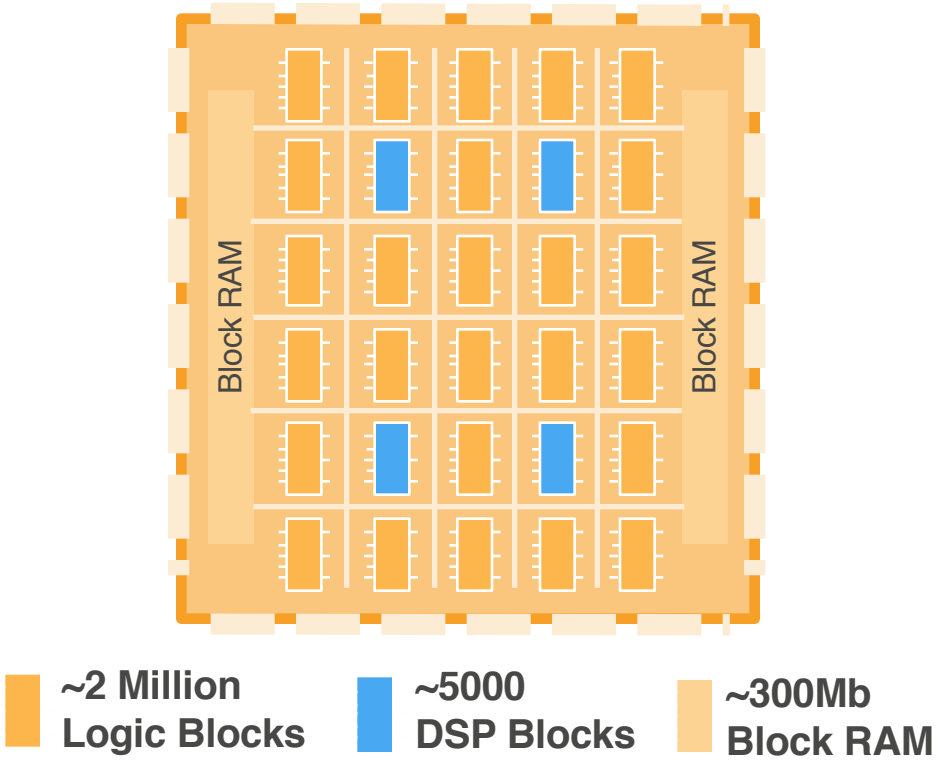
Flexibility

- FPGAs ideal for adapting to rapidly evolving ML
- CNNs, LSTMs, MLPs, reinforcement learning, feature extraction, decision trees, etc.
- Inference-optimized numerical precision
- Exploit sparsity, deep compression for larger, faster models

[source: Materials adapted from Microsoft BrainWave presentation at HotChips'2017]

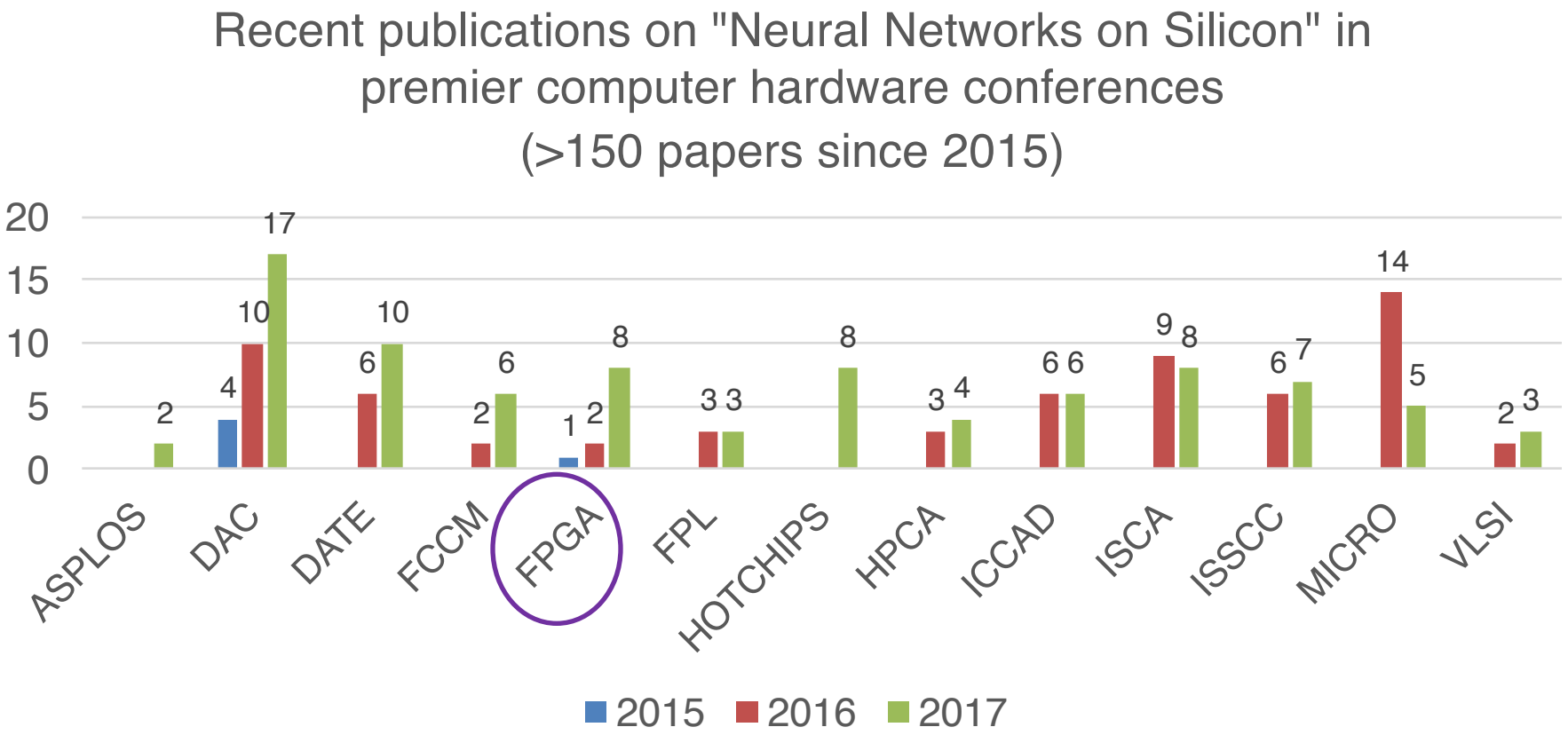
FPGA as a Programmable Accelerator

- Massive amount of fine-grained parallelism
  - Highly parallel and/or deeply pipelined
  - Distributed data/control dispatch
- Silicon configurable to fit the application
  - Compute the exact algorithm at the desired level of numerical accuracy
- Performance/watt advantage over CPUs & GPUs



AWS Cloud F1 FPGA instance:
Xilinx UltraScale+ VU9P
[Figure source: David Pellerin, AWS]

A (Hyper-)Active Body of Research



Data source: https://github.com/fengbintu/Neural-Networks-on-Silicon

This tutorial focuses on FPGA-based DNN inference accelerators by discussing a few representative work

CNN ACCELERATION

Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

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

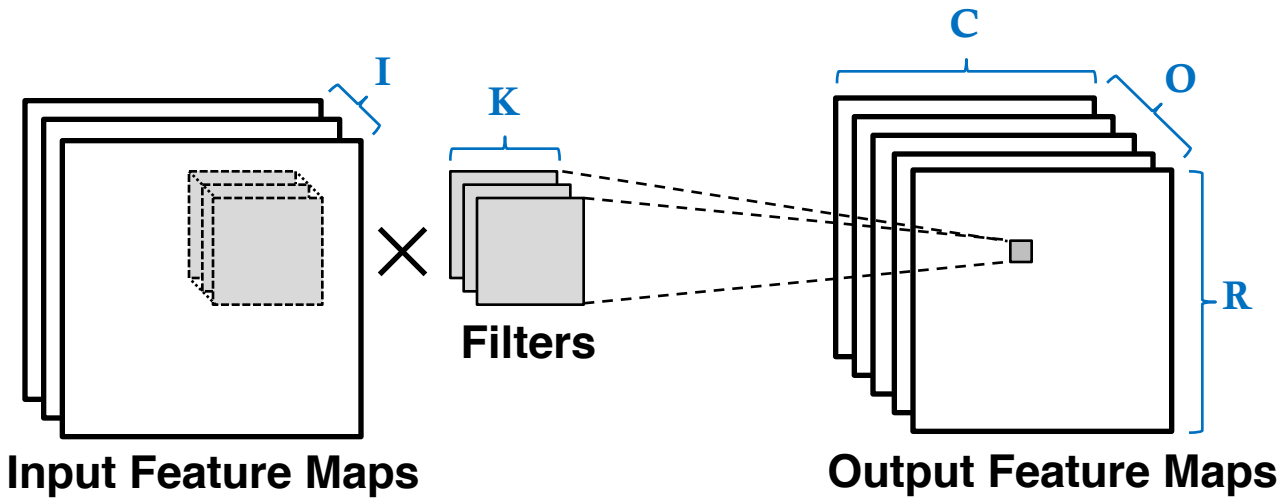
[1]Center for Energy-Efficient Computing and Applications, Peking University
[2]Computer Science Department, University of California, Los Angeles
[3]PKU/UCLA Joint Research Institute in Science and Engineering

FPGA'15, Feb 2015

Main Contributions

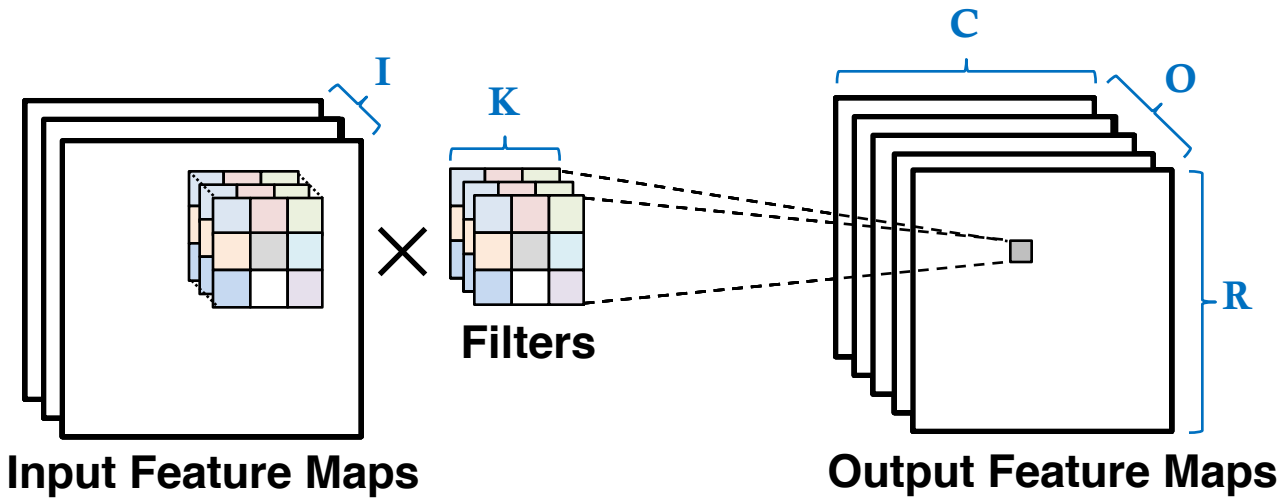
1. Analysis of the different sources of parallelism in the convolution kernel of a CNN
2. Quantitative performance modeling of the hardware design space using the Roofline method
3. Design and implementation of a CNN accelerator for FPGA using Vivado HLS, evaluated on AlexNet

Convolutional Layer Revisited



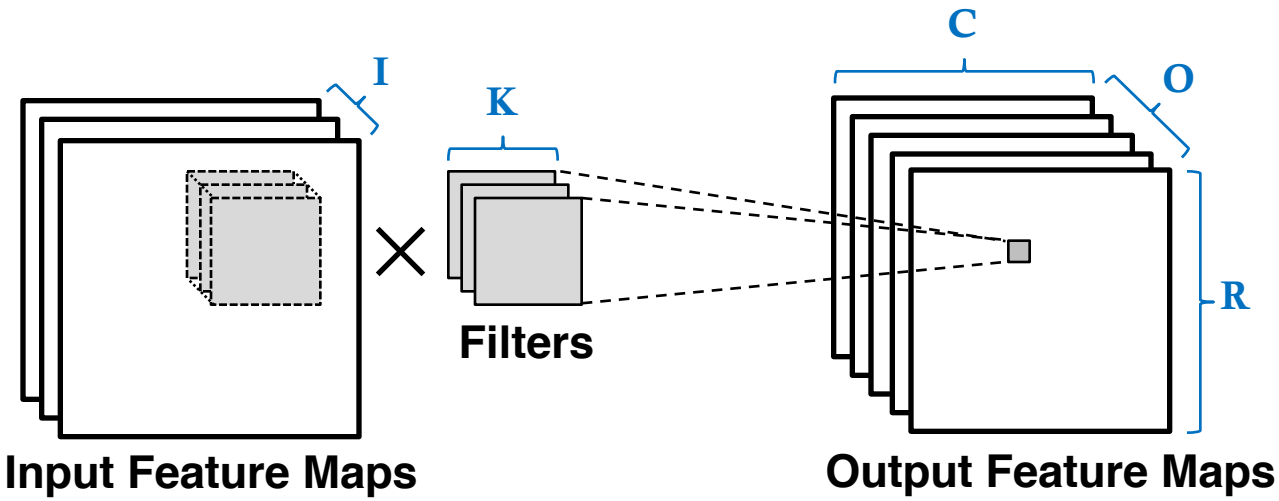
- An output pixel is connected to its neighboring region on each input feature map
- All pixels on an output feature map use the same filter weights

Parallelism in the Convolutional Layer



- Four main sources of parallelism
  1. Across input feature maps
  2. Across output feature maps
  3. Across different output pixels (i.e. filter positions)
  4. Across filter pixels

Parallelism in the Code




```
1 for (row=0; row<R; row++) {
2    for (col=0; col<C; col++) {
3       for (to=0; to<O; to++) {
4          for (ti=0; ti<I; ti++) {
5             for (ki=0; ki<K; ki++) {
6                for (kj=0; kj<K; kj++) {
                    output_fm[to][row][col] +=
                    weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];
}}}}}}
```

Loop over output pixels

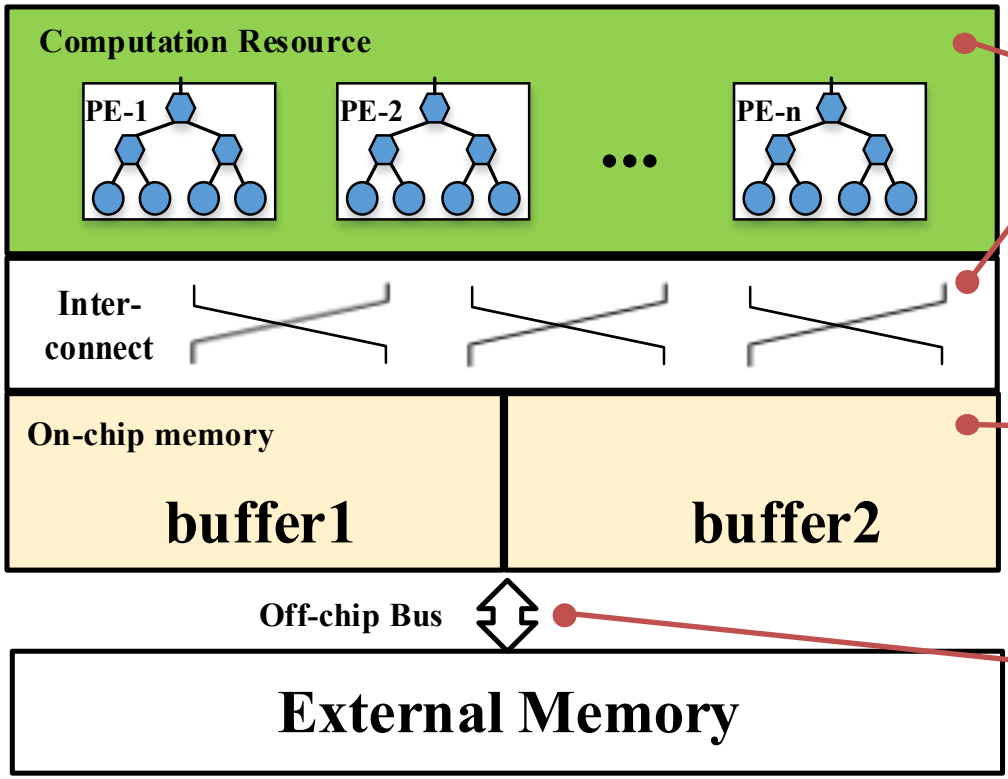
Loop over output feature maps

Loop over input feature maps

Loop over filter pixels

Challenges for FPGA

- We can't just unroll all the loops due to limited FPGA resources
- Must choose the right code transformations to exploit the parallelism in a resource efficient way



Limited compute and routing resources

Technique: loop unrolling, pipelining, interchange

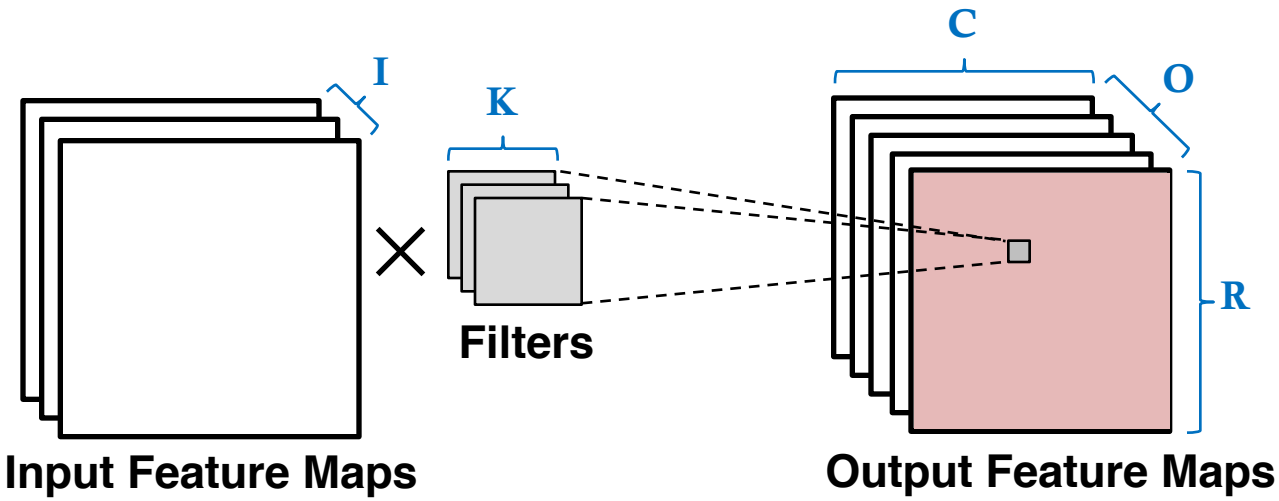
Limited on-chip storage

Technique: loop tiling

Limited off-chip bandwidth

Technique: data reuse, compute-memory balance

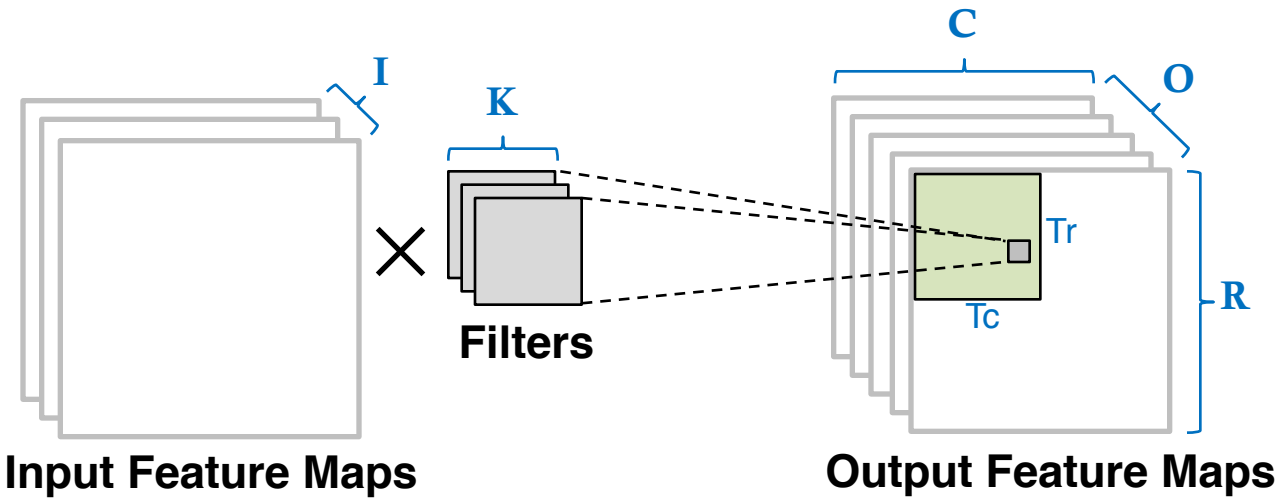
Loop Tiling




```
1 for (row=0; row<R; row++) {
2     for (col=0; col<C; col++) {
3         for (to=0; to<O; to++) {
4             for (ti=0; ti<I; ti++) {
5                 for (ki=0; ki<K; ki++) {
6                     for (kj=0; kj<K; kj++) {
                          output_fm[to][row][col] +=
                          weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];
}}}}}}
```

Loop over **pixels** in an output map

Loop Tiling




```
1 for (row=0; row<R; row+=Tr) {
2   for (col=0; col<C; col+=Tc) {
3     for (to=0; to<O; to++) {
4       for (ti=0; ti<I; ti++) {
5         for (trr=row; trr<min(row+Tr, R); trr++) {
6           for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
```

Loop over different **tiles**

Loops over **pixels** in each tile

Offloading just the inner loops requires only a small portion of the data to be stored on FPGA chip

Code with Loop Tiling

```
1 for (row=0; row<R; row+=Tr) {
2     for (col=0; col<C; col+=Tc) {
3         for (to=0; to<O; to+=Tm) {
4             for (ti=0; ti<I; ti+=Tn) {
                  // software: write output feature map
                  // software: write input feature map + filters

5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
7                         for (too=to; too<min(to+To, O); too++) {
8                             for (tii=ti; tii<(ti+Ti, I); tii++) {
9                                 for (ki=0; ki<K; ki++) {
10                                    for (kj=0; kj<K; kj++) {
                                          output_fm[too][trr][tcc] +=
                                          weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
                  }}}}}}

                  // software: read output feature map
}}}}
```

CPU Portion

Maximize memory reuse

FPGA Portion

Maximize computational performance

Optimizing for Memory Reuse

```
1 for (row=0; row<R; row+=Tr) {
2     for (col=0; col<C; col+=Tc) {
3         for (to=0; to<O; to+=Tm) {
              // software: write output feature map
4             for (ti=0; ti<I; ti+=Tn) {
                  // software: write input feature map + filters

5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
7                         for (too=to; too<min(to+To, O); too++) {
8                             for (tii=ti; tii<(ti+Ti, I); tii++) {
9                                 for (ki=0; ki<K; ki++) {
10                                    for (kj=0; kj<K; kj++) {
                                          output_fm[too][trr][tcc] +=
                                          weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
                  }}}}}}
              }
              // software: read output feature map
}}}
```

Move these to the outer loop, reusing the output feature map to reduce data transfer

Optimizing for On-Chip Performance

```
5           for (trr=row; trr<min(row+Tr, R); trr++) {
6             for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
7               for (too=to; too<min(to+To, O); too++) {
8                 for (tii=ti; tii<(ti+Ti, I); tii++) {
9                   for (ki=0; ki<K; ki++) {
10                    for (kj=0; kj<K; kj++) {
                        output_fm[too][trr][tcc] +=
                        weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Optimizing for On-Chip Performance

```
9           for (ki=0; ki<K; ki++) {
10            for (kj=0; kj<K; kj++) {
5               for (trr=row; trr<min(row+Tr, R); trr++) {
6                 for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
7                   for (too=to; too<min(to+To, O); too++) {
8                     for (tii=ti; tii<(ti+Ti, I); tii++) {
                        output_fm[too][trr][tcc] +=
                        weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Reorder these two loops to the top level

Optimizing for On-Chip Performance

```
9           for (ki=0; ki<K; ki++) {
10             for (kj=0; kj<K; kj++) {
5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
                          #pragma HLS pipeline
7                         for (too=to; too<min(to+To, O); too++) {
8                             for (tii=ti; tii<(ti+Ti, I); tii++) {
                                  output_fm[too][trr][tcc] +=
                                  weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Optimizing for On-Chip Performance

```
9           for (ki=0; ki<K; ki++) {
10             for (kj=0; kj<K; kj++) {
5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
                          #pragma HLS pipeline
7                         for (too=to; too<min(to+To, O); too++) {
                             #pragma HLS unroll
8                            for (tii=ti; tii<(ti+Ti, I); tii++) {
                                output_fm[too][trr][tcc] +=
                                weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
             }}}}}}
```

Optimizing for On-Chip Performance

```
9           for (ki=0; ki<K; ki++) {
10             for (kj=0; kj<K; kj++) {
5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(tcc+Tc, C); tcc++) {
                          #pragma HLS pipeline
7                         for (too=to; too<min(to+To, O); too++) {
                             #pragma HLS unroll
8                            for (tii=ti; tii<(ti+Ti, I); tii++) {
                                #pragma HLS unroll
                                output_fm[too][trr][tcc] +=
                                weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Number of cycles to execute the above loop nest

$$\approx K \times K \times Tr \times Tc + L \approx Tr \times Tc \times K^2$$

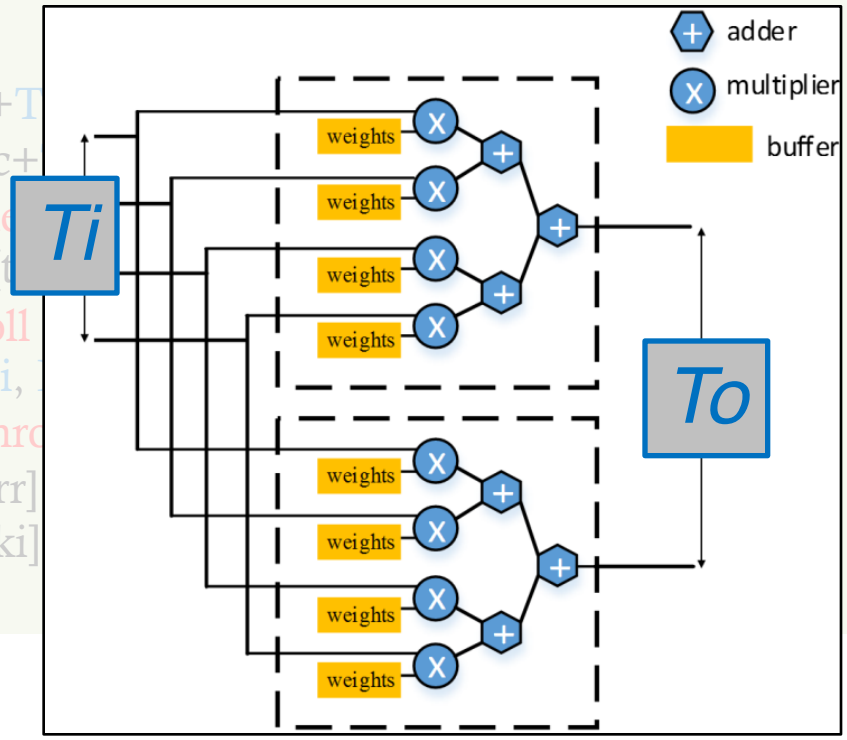
L is the pipeline depth (# of pipeline stages, II=1)

Optimizing for On-Chip Performance

Generated Hardware

Performance and size of each PE determined by tile factors *Ti* and *To*

Number of data transfers determined by Tr and Tc



Design Space Complexity

- **Challenge:** Number of available optimizations present a huge space of possible designs
  - What is the optimal loop order?
  - What tile size to use for each loop?
- Implementing and testing each design by hand will be slow and error-prone
  - Some designs will exceed the on-chip compute/memory capacity
- **Solution:** Performance modeling + automated design space exploration

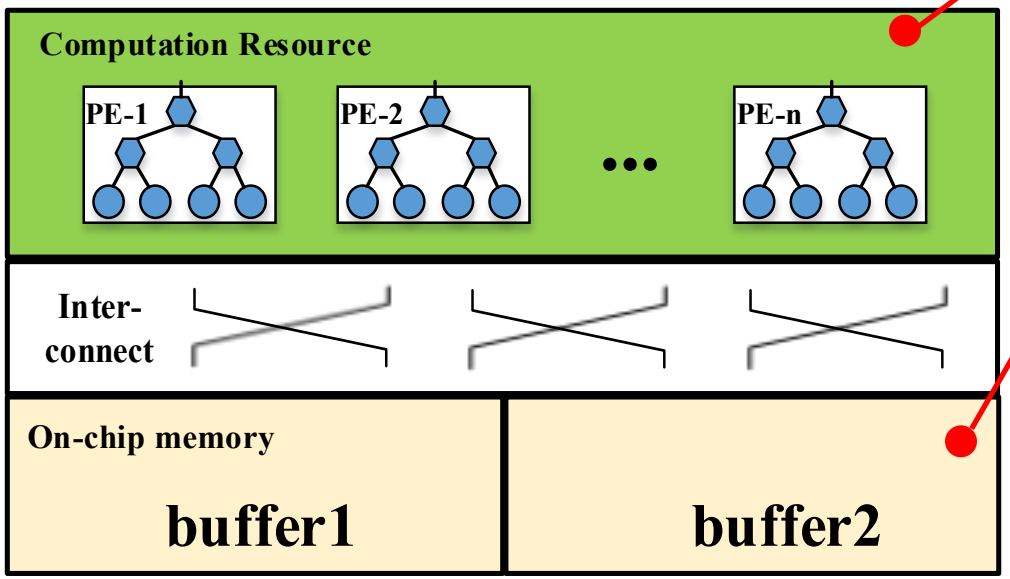
Performance Modeling

- We calculate the following design metrics:
  - **Total number of operations (FLOP)**
    - Depends on the CNN model parameters
  - **Total external memory access (Byte)**
    - Depends on the CNN weight and activation size
  - **Total execution time (Sec)**
    - Depends on the hardware architecture (e.g., tile factors To and Ti)
    - Ignore resource constrains for now

Performance Modeling

- Total FP operations $\approx 2 \times O \times I \times R \times C \times K^2$
- Execution time = Number of Cycles × Clock Period
  - Number of cycles $\approx \left\lceil \frac{O}{To} \right\rceil \times \left\lceil \frac{I}{Ti} \right\rceil \times \left\lceil \frac{R}{Tr} \right\rceil \times \left\lceil \frac{C}{Tc} \right\rceil \times (Tr \times Tc \times K^2)$
    $\approx \left\lceil \frac{O}{To} \right\rceil \times \left\lceil \frac{I}{Ti} \right\rceil \times R \times C \times K^2$
- External memory accesses $= ai \times Bi + aw \times Bw + ao \times Bo$
  - Size of input fmap buffer: $Bi = Ti \times (Tr+K-1)(Tc+K-1)$ with stride=1
  - Size of output fmap buffer: $Bo = To \times Tr \times Tc$
  - Size of weight buffer: $Bw = To \times Tr \times K^2$
  - External access times: $ao = \left\lceil \frac{O}{To} \right\rceil \times \left\lceil \frac{R}{Tr} \right\rceil \times \left\lceil \frac{C}{Tc} \right\rceil$, $ai = aw = \left\lceil \frac{I}{Ti} \right\rceil \times ao$

Performance Modeling



Computational Throughput

$$= \frac{\text{Total number of operations}}{\text{Total execution time}}$$

GFLOP/Sec

Computation To Communication (CTC) Ratio

$$= \frac{\text{Total number of operations}}{\text{Total external memory access}}$$

FLOP / (Byte Accessed)

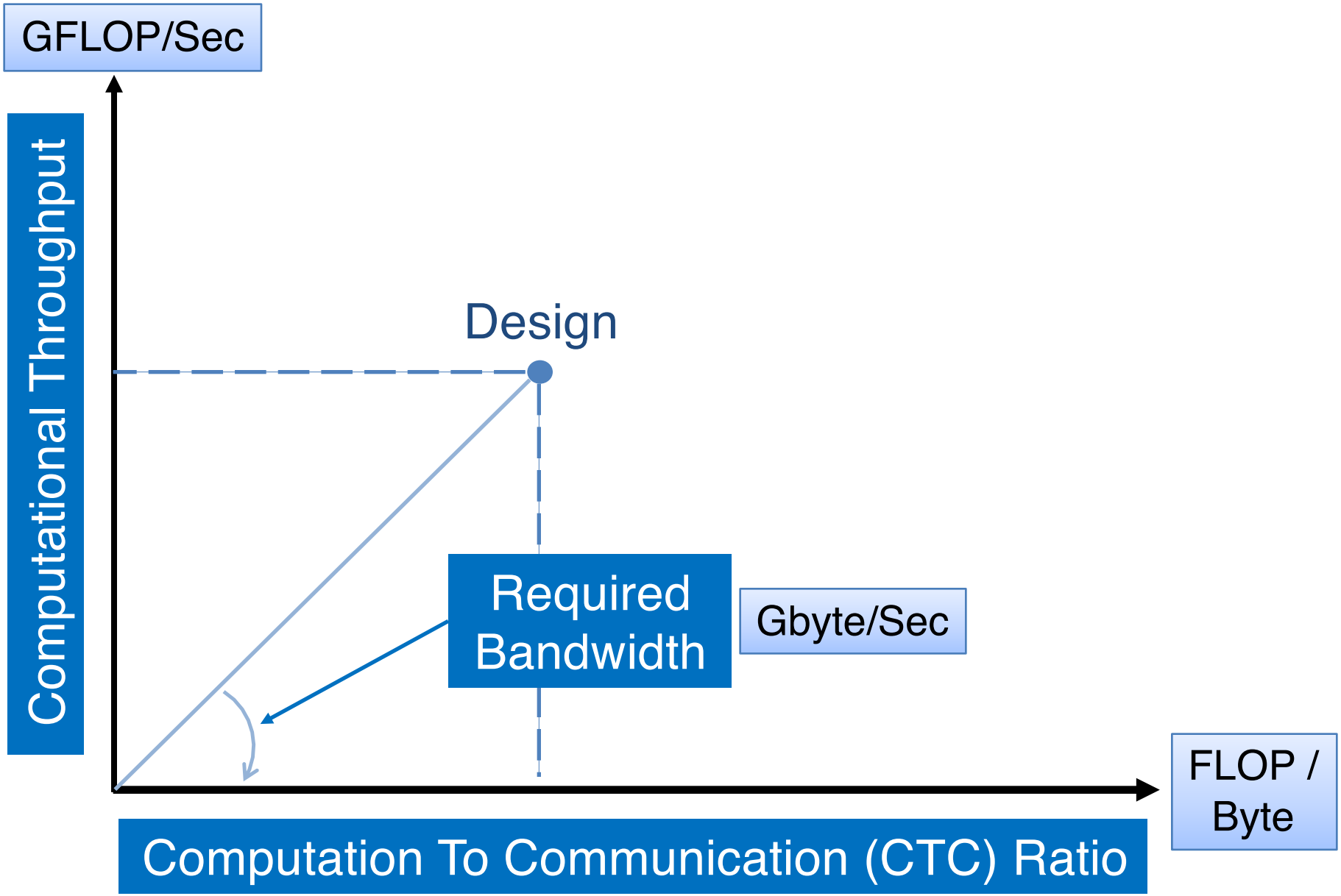
Required Bandwidth

$$= \frac{\text{Computational Throughput}}{\text{CTC Ratio}}$$

GByte/Sec

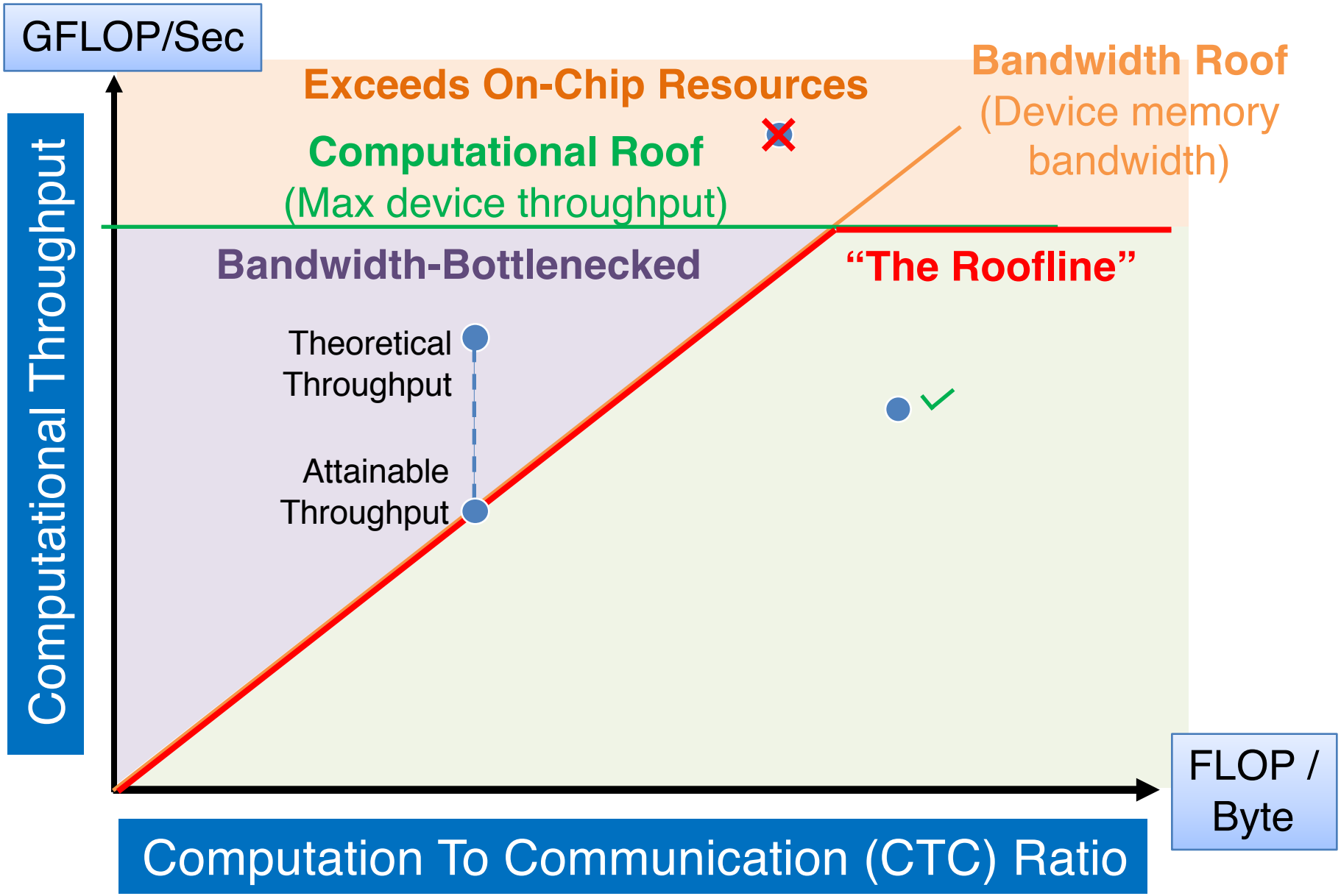
$$\frac{GByte}{Sec} = \frac{GFLOP/Sec}{FLOP/(Byte\ Accessed)}$$

Roofline Method [1]



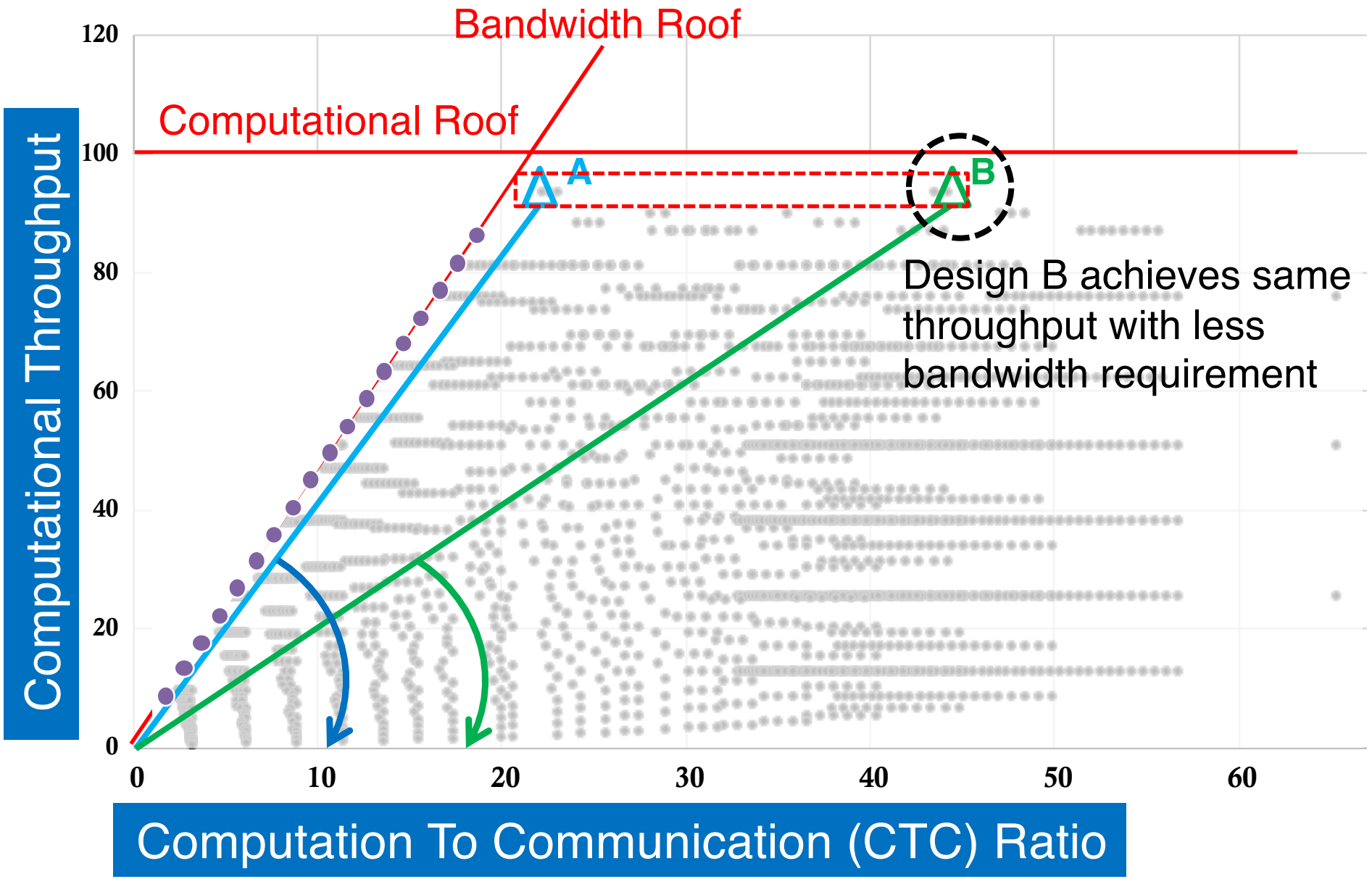
[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

Roofline Method [1]

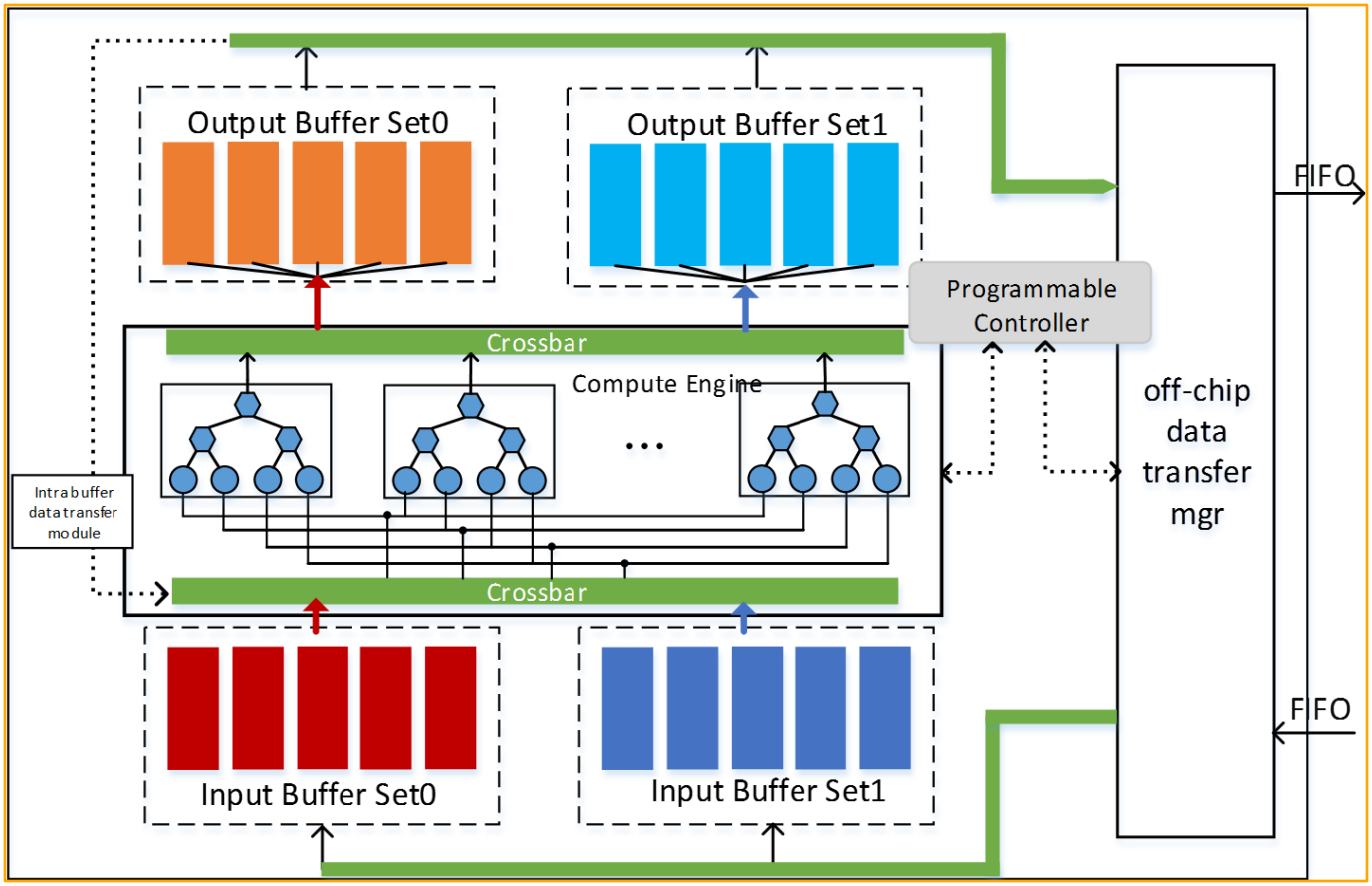


[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

Design Space Exploration



Hardware Implementation



- All values in floating-point
- Only handles Conv layers, not dense or pooling
- Virtex7-485t
- 100MHz
- 61.6 GOPS
- 18.6 W power

Experimental Evaluation

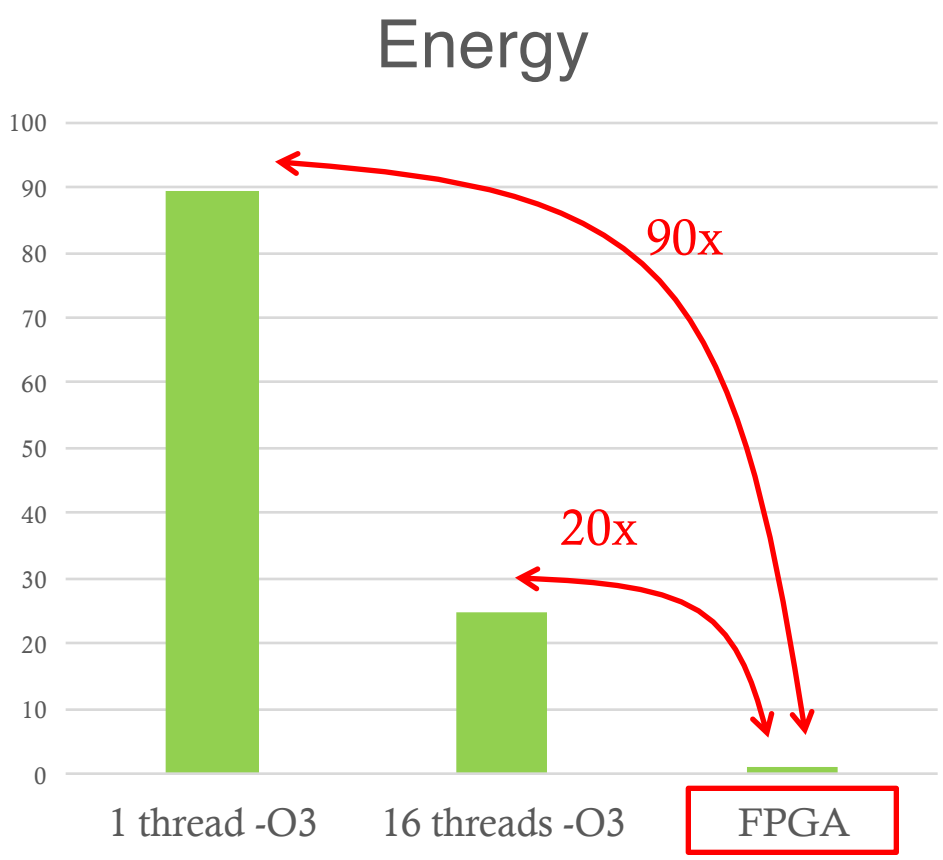
- The accelerator design (for AlexNet) is implemented with Vivado HLS on the VC707 board containing a Virtex7 FPGA
- Hardware resource utilization

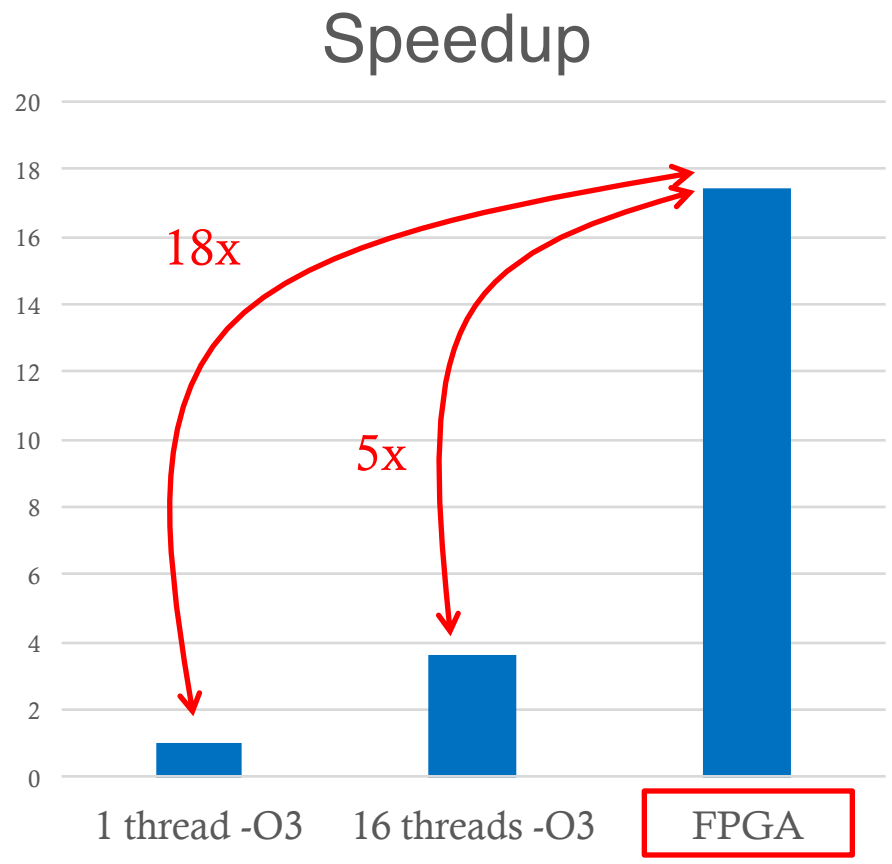
Resource	DSP	BRAM	LUT	FF
Used	2240	1024	186251	205704
Available	2800	2060	303600	607200
Utilization	80%	50%	61%	34%

- Comparison of theoretical and real performance

Layer 2	Theoretical	Measured	Difference
	5.11ms	5.25ms	~5%

Experimental Results





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

Going Deeper with Embedded FPGA Platform for Convolutional Neural Network

Jiantao Qiu[1,2], Jie Wang[1], Song Yao[1,2], Kaiyuan Guo[1,2], Boxun Li[1,2], Erjin Zhou[1], Jincheng Yu[1,2], Tianqi Tang[1,2], Ningyi Xu[3], Sen Song[2,4], Yu Wang[1,2], Huazhong Yang[1,2]

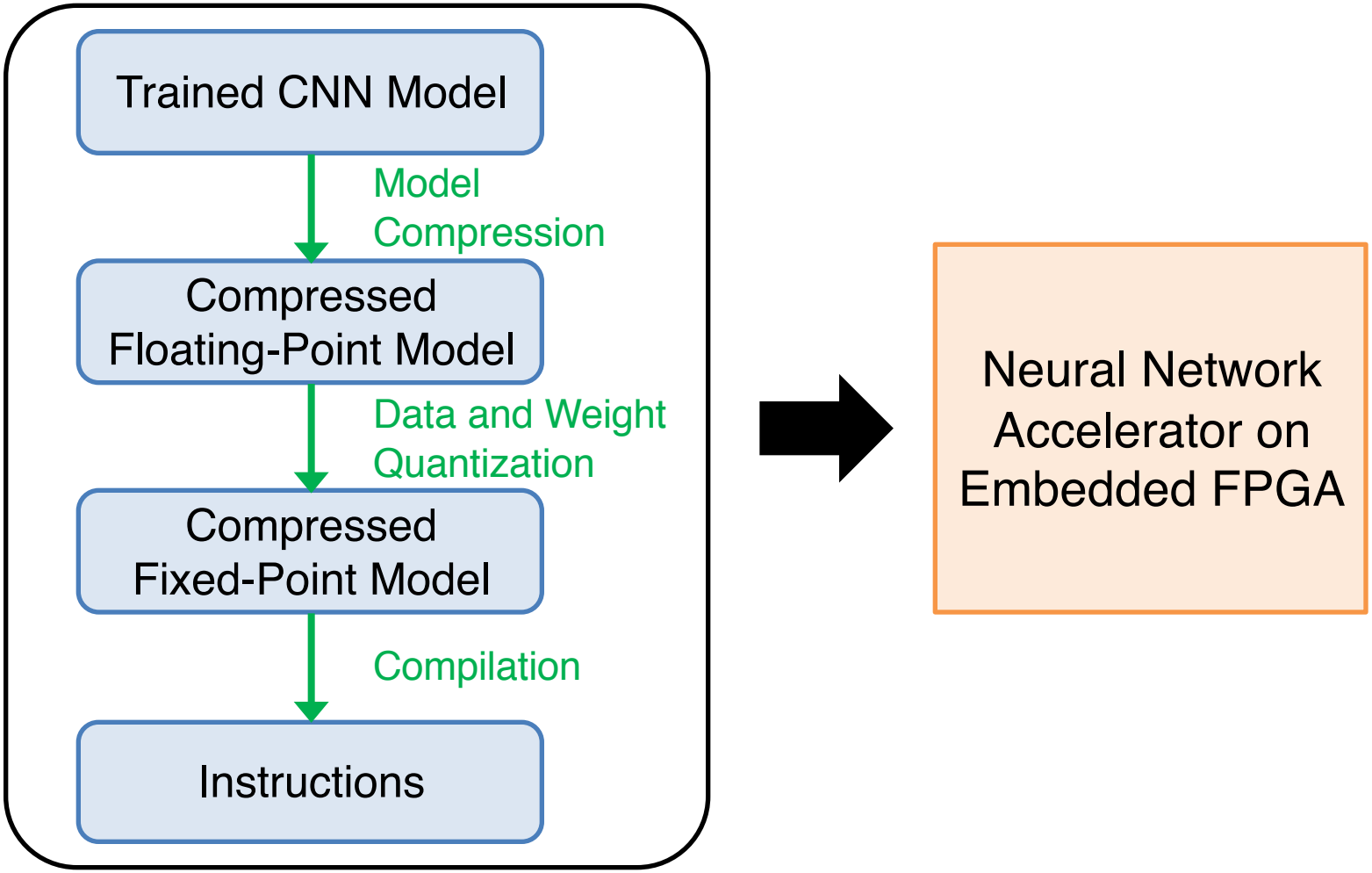
[1]Department of Electronic Engineering, Tsinghua University
[2]Center for Brain-Inspired Computing Research, Tsinghua University
[3]Hardware Computing Group, Microsoft Research Asia
[4]School of Medicine, Tsinghua University

FPGA'16, Feb 2016

Main Contributions

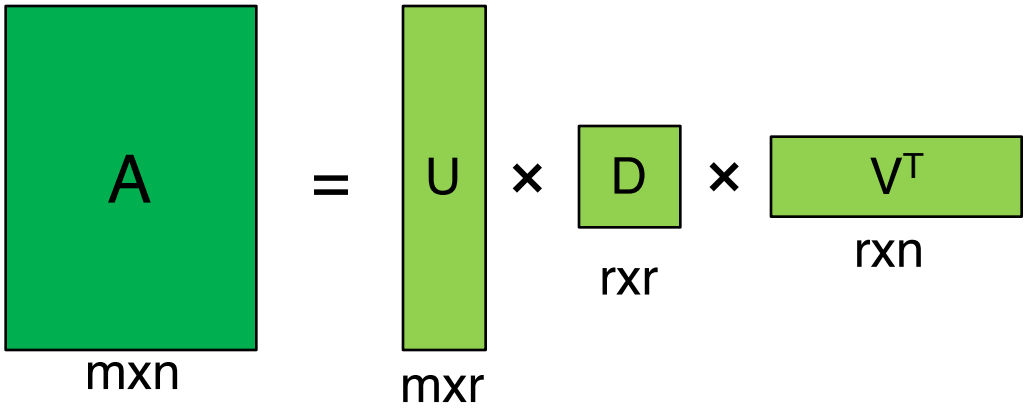
1. Automated flow for dynamic-precision data quantization
2. Application of singular value decomposition (SVD) to the dense layer weights, greatly reducing model size
3. A reconfigurable processing engine which can target both convolutional and dense layer
4. Design and implementation of a CNN accelerator for *embedded* FPGA, evaluated on the VGG-16 network

Overall Acceleration Flow



Model Compression

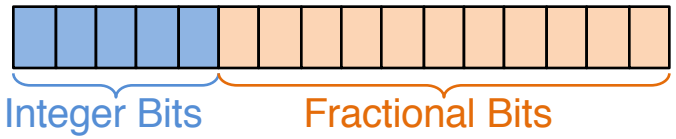
- The authors employ singular value decomposition (SVD) to compress the dense layer weights



Network	Matrix Size (layer FC6)	Total Weights	Top-5 Accuracy
VGG-16	25088×4096	138.4M	88.00%
VGG-16-SVD	25088×500 + 500×4096	50.2M	87.96%

Weight and Data Quantization

- Quantizing a CNN model involves many choices:
  - Fixed-point bitwidth
  - Integer vs. fractional width (precision)
  - Weight bits vs. activation bits
- The authors make use of **dynamic precision** fixed-point
  - The # of fractional vs. integer bits is configurable in hardware, and can change across different layers
  - Total bitwidth remains the same



High precision, low range

Low precision, high range

Dynamic Precision Fixed-Point

- Since the precision is configurable, we need an algorithm to find the optimal precision for each layer
- Precise problem formulation:

$$f_l = \underset{f_l}{\operatorname{argmin}} \sum |W_{float} - W_{quantized}(f_l)|$$

- To find the best f_l the authors use a greedy heuristic:
  - Use the dynamic range of the weights in a layer to choose an f_l which can represent that range
  - Explore neighboring values of f_l until a local minimum is found

Quantization Experiments

Network	VGG16				
Data Bits	Single-float	16	8	8	8
Weight Bits	Single-float	16	8	8	8 or 4†
Data Precision	-	Fixed	Fixed*	Dynamic	Dynamic
Weight Precision	-	Fixed	Fixed	Dynamic	Dynamic
Top-1 Accuracy	**68.1%**	68.0%	28.2%	66.6%	**67.0%**
Top-5 Accuracy	**88.0%**	87.9%	49.7%	87.4%	**87.6%**

* Different number format for Conv and Dense

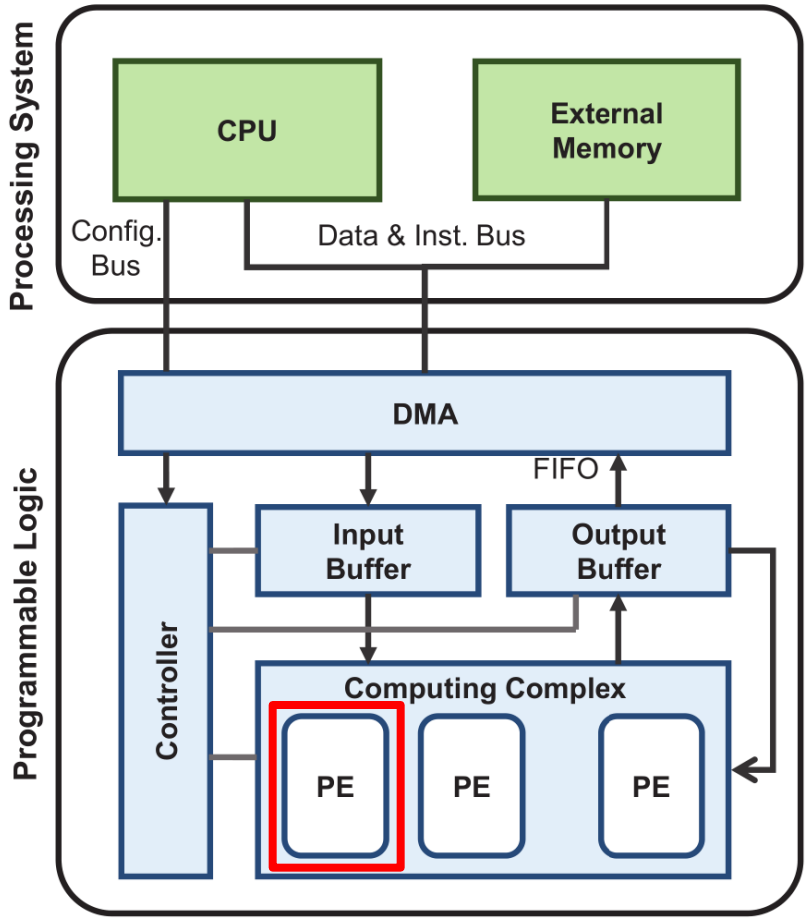
† 8 bits for Conv, 4 bits for Dense

Dynamic fixed-point greatly outperforms static

Network	VGG16-SVD		
Data Bits	Single-float	16	8
Weight Bits	Single-float	16	8 or 4†
Data Precision	-	Dynamic	Dynamic
Weight Precision	-	Dynamic	Dynamic
Top-1 Accuracy	**68.0%**	64.6%	64.1%
Top-5 Accuracy	**88.0%**	86.7%	86.3%

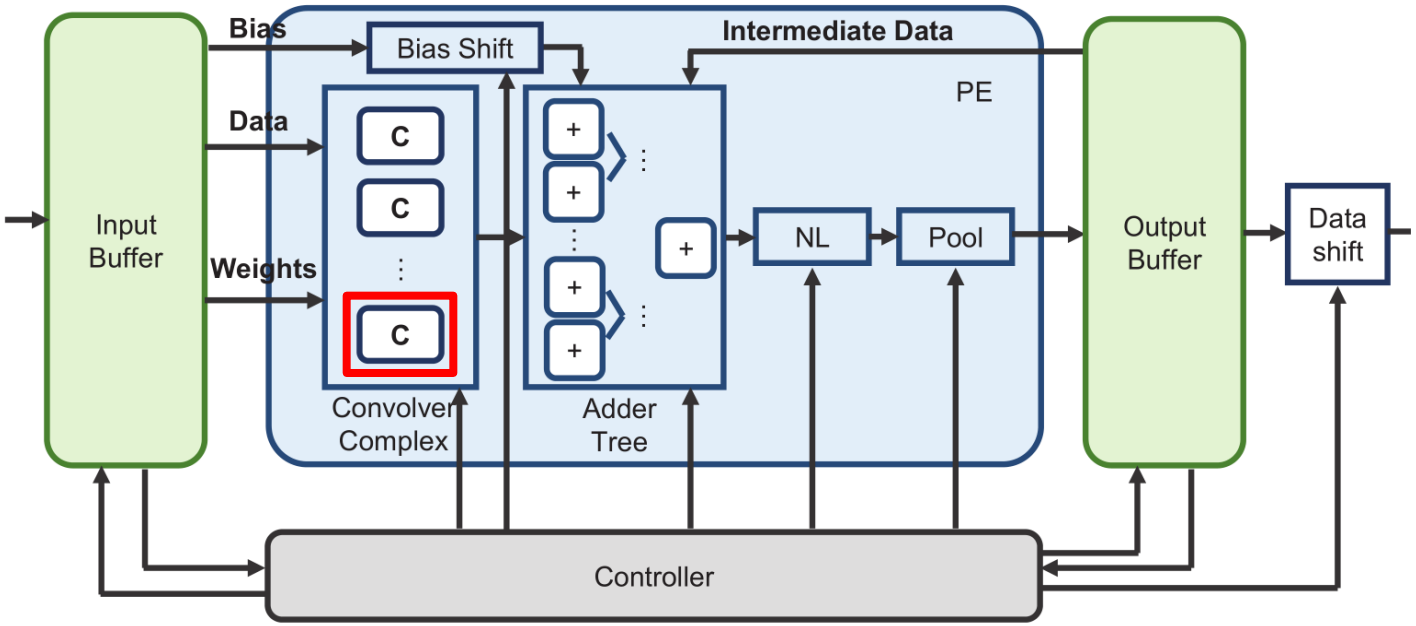
Dynamic fixed-point + SVD together loses a few% of accuracy

Hardware System Design



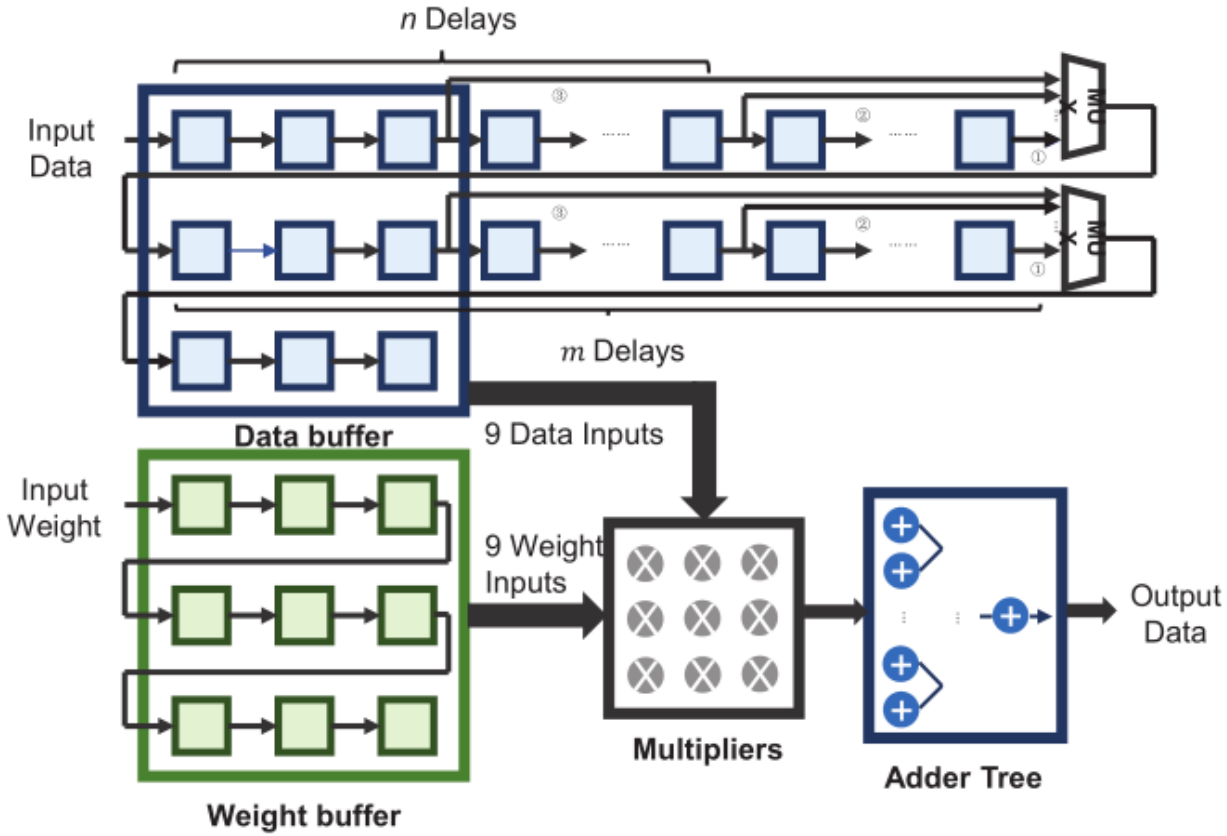
- 16-bit dynamic fixed-point
- Processing System
  - CPU issues instructions to accelerator, and prepares data
  - Instructions allow the system to handle different networks
- Programmable Logic
  - Computing Complex + On-Chip Buffers + Controller + DMA
  - Small number of complex PEs

PE Architecture



- Multiple PEs → inter-output parallelism
- Multiple convolvers → intra-output parallelism
- Bias Shift and Data Shift handle dynamic fixed-point
- NL and Pool handle non-linearities and pooling

Convolver Design



- Optimized for 3x3 convolution (VGG-16 only uses 3x3 convs)
- Line buffering for convolutional data reuse
- Supports dense layers using configurable line buffer delays

Comparison vs. Other Systems

Layers	FPGA	CPU	GPU	GPU (Embedded)
All Conv layers (GOP/s)	187.8	98.3	1986	76.8
All Dense layers (GOP/s)	1.2	17.2	40.1	1.4
Total (GOP/s)	**137.0**	**97.2**	**1784**	**67.8**
Power (W)	**9.6**	**135**	**250**	**9.0**
Energy Efficiency (GOP/s/W)	**14.3**	**0.7**	**7.1**	**7.5**

- FPGA uses 16-bit fixed point, 2 PEs, and 64 convolvers per PE
- FPGA does well for Conv layers, but poorly in Dense layers due to limited memory bandwidth
- Compared to FPGA (Xilinx Zynq ZC706), the GPU (NVIDIA K40) has 13x higher performance but consumes 26x more power

Comparison vs. Previous Paper

	Zhang 2015	This Paper
Platform	Virtex7 VX485t	Zynq XC7Z045
Clock (MHz)	100	150
Quantization	32-bit float	16-bit fixed
Logic Utilization	186K (61%)	183K (84%)
DSP Utilization	2240 (80%)	780 (89%)
BRAM Utilization	1024 (50%)	486 (87%)
Total GOP in Network	1.33	30.8
Performance (GOP/s)	61.6	137.0
Power (W)	18.6	9.6
Energy Efficiency (GOP/J)	3.3	14.2

Model compression, quantization, and hardware reuse enables **higher performance** with **fewer resources** and **lower power**

An OpenCL Deep Learning Accelerator on Arria 10

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

Intel Corporation (formerly Altera)
Toronto, Canada

FPGA'17, Feb 2017

Main Contributions

- Reducing external memory bandwidth usage by:
  1. Storing all intermediate feature maps in on-chip buffers
  2. Image batching for dense layers
- Optimizing the convolution arithmetic using the Winograd Transformation
- A compute-bound implementation on Arria 10 whose energy efficiency matches the Titan X GPU

OpenCL Programming

```
void sum (float* a,
          float* b,
          float* c)
{
   for (i = 0; i < 100; i++)
      c[i] = a[i] * b[i];
}
```

Conventional C++

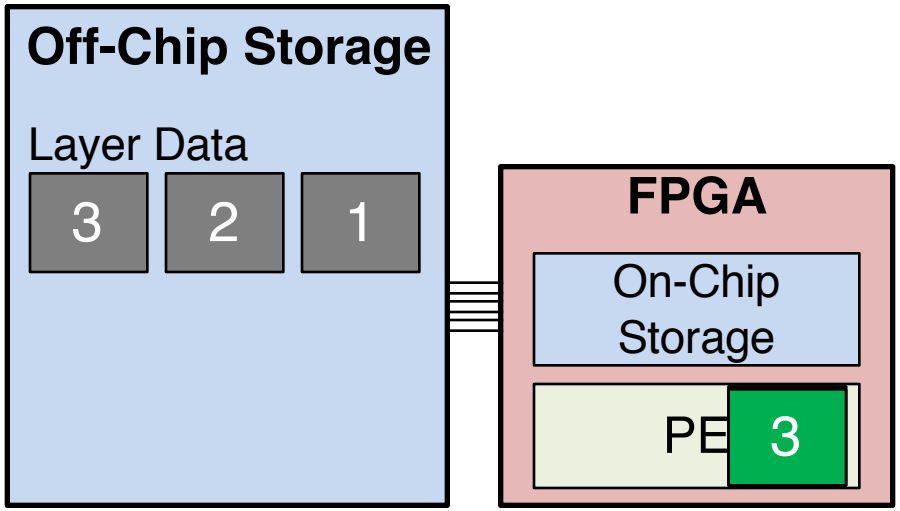
```
kernel
void sum (global float* a,
          global float* b,
          global float* c)
{
   int gid = get_global_id(0);
      c[gid] = a[gid] * b[gid];
}
```

OpenCL

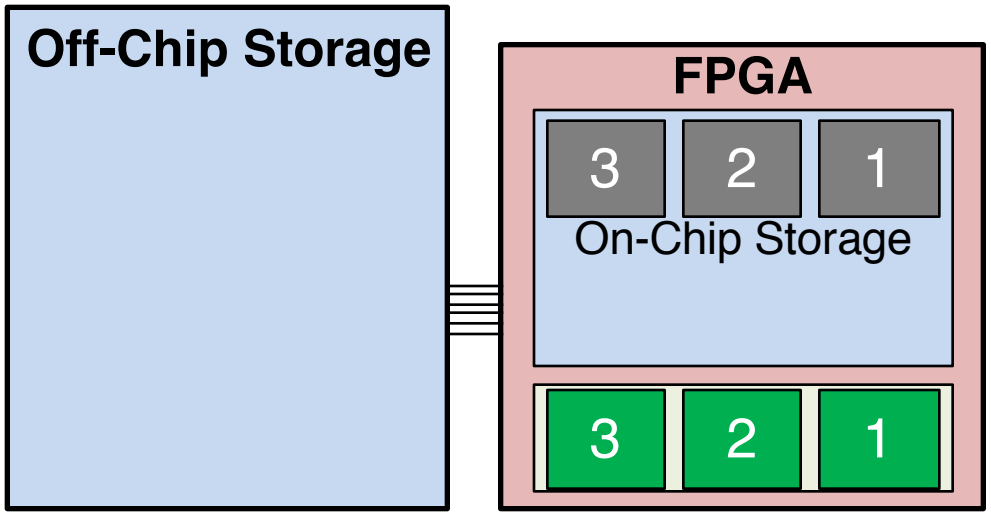
- The FPGA'15 paper used C++ with **Xilinx Vivado HLS** to generate RTL
- Sequential programming model using loops
- Inter-loop-iteration parallelism is implicit (requires unrolling)

- This work used OpenCL and **Intel FPGA SDK** to generate RTL
- Parallel programming model using multithreaded kernels; where inter-iteration parallelism is explicit – each thread obtains an independent ID

Data Placement

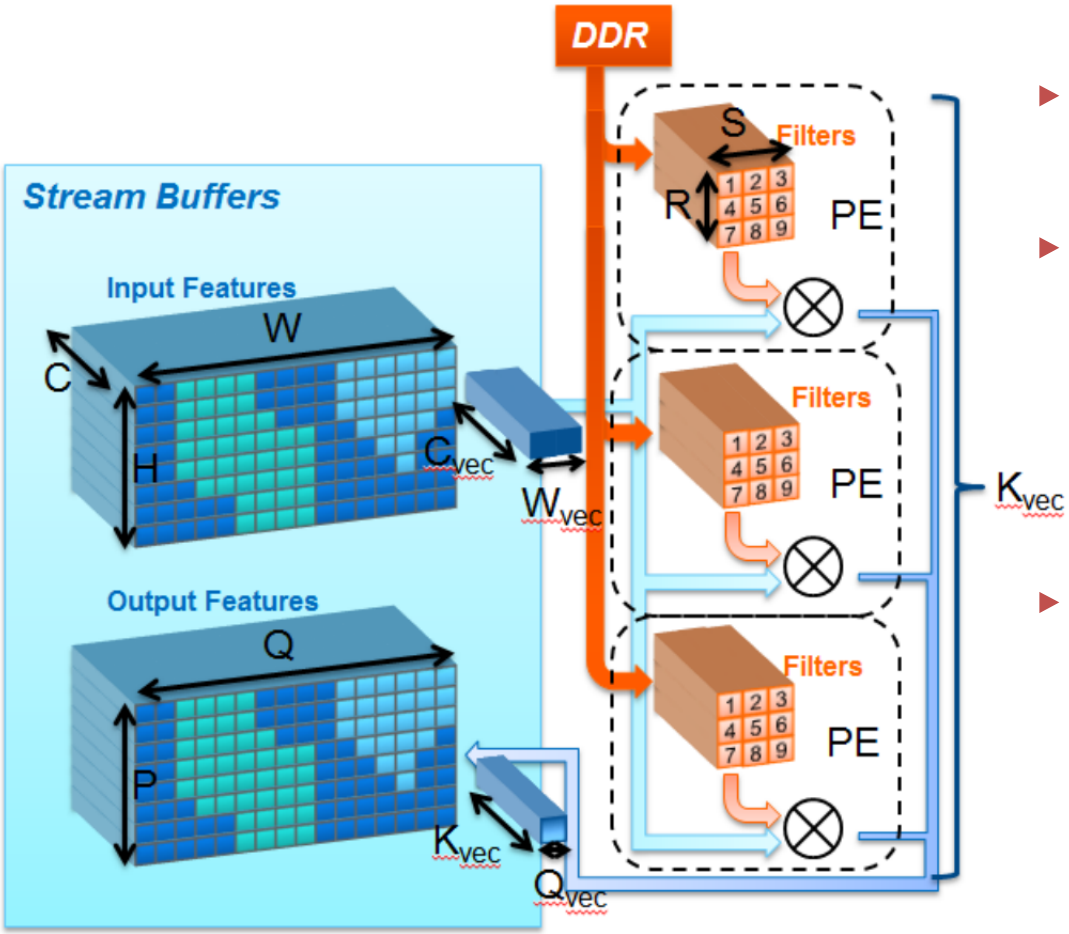


- Previous papers stored layer data off-chip
- Insufficient on-chip storage to hold all data for a layer (input+output)



- This work uses Arria 10 FPGA device
- Enough storage to keep data on-chip (for conv layers in AlexNet)
- Use double-buffering to store input+output

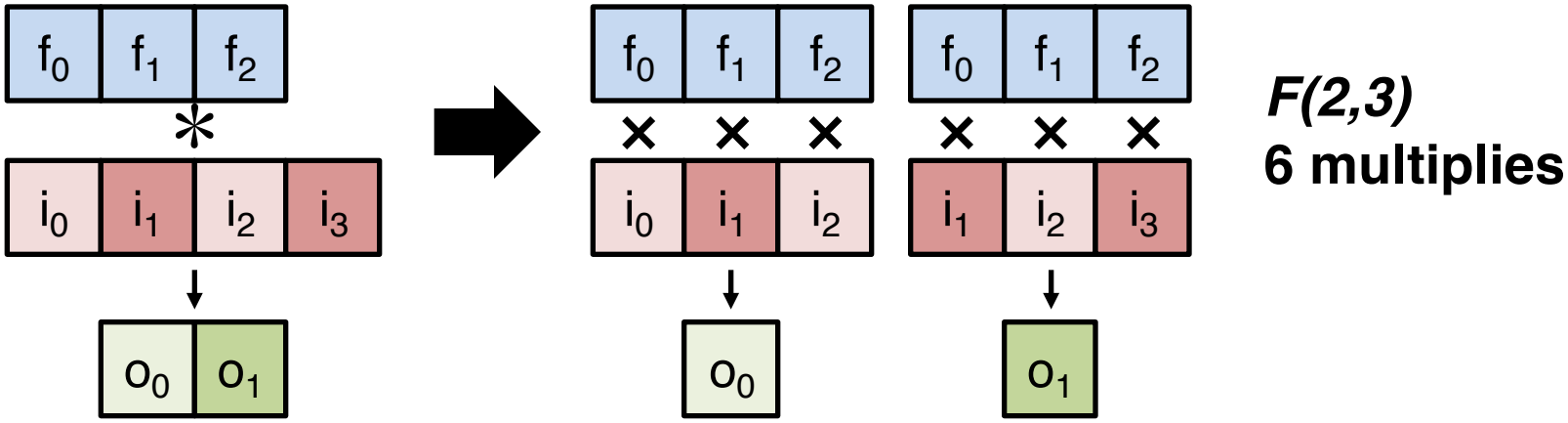
Parallelizing Convolutions



- First paper (FPGA'15) had tile sizes To, Ti, Tr, Tc
- Similarly this paper uses vectorization sizes
  K_{vec} → To
  C_{vec} → Ti
  W_{vec} → Tc
- This paper decomposes a 2D conv into horizontal 1D convs, so Tr is effectively 1

Arithmetic Optimizations

- On FPGA, DSP blocks (used for fixed-point multiplies) are typically the bottlenecked resource
- Consider a 1-dimensional convolution with output length 2 and filter length 3, denoted *F(2,3)*

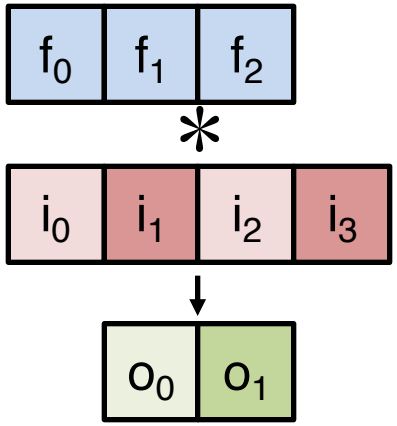


- In the 70s, Shmuel Winograd proved that *F(m,r)* can be computed with a lower bound of only *m+r−1* multiplies [1]

[1] S. Winograd, **Arithmetic Complexity of Computations,** SIAM, Jan 1, 1980

Winograd Transform

Naïve Approach



$$\begin{pmatrix} o_0 \\ o_1 \end{pmatrix} = \begin{pmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$$

$$= \begin{pmatrix} i_0 f_0 + i_1 f_1 + i_2 f_1 \\ i_1 f_0 + i_2 f_1 + i_3 f_2 \end{pmatrix}$$

6 unique multiplies

Winograd Approach

$$\begin{pmatrix} o_0 \\ o_1 \end{pmatrix} = \begin{pmatrix} y_0 + y_1 + y_2 \\ y_1 - y_2 - y_3 \end{pmatrix}$$

Each $y_i = d_i g_i$

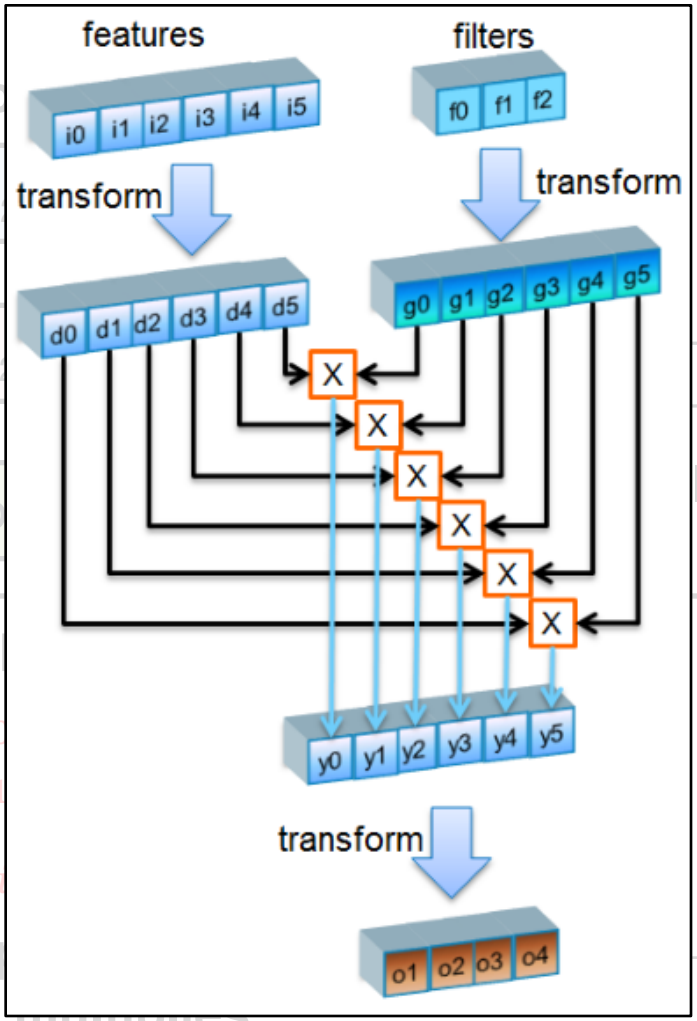
1 multiply per y_i
4 unique multiplies

$$d_0 = i_0 - i_2 \qquad d_3 = i_1 - i_3$$
$$d_1 = i_1 + i_2 \qquad d_2 = i_2 - i_1$$

$$g_0 = f_0 \qquad g_3 = f_2$$
$$g_1 = \frac{f_0 + f_1 + f_2}{2} \qquad g_2 = \frac{f_0 - f_1 + f_2}{2}$$

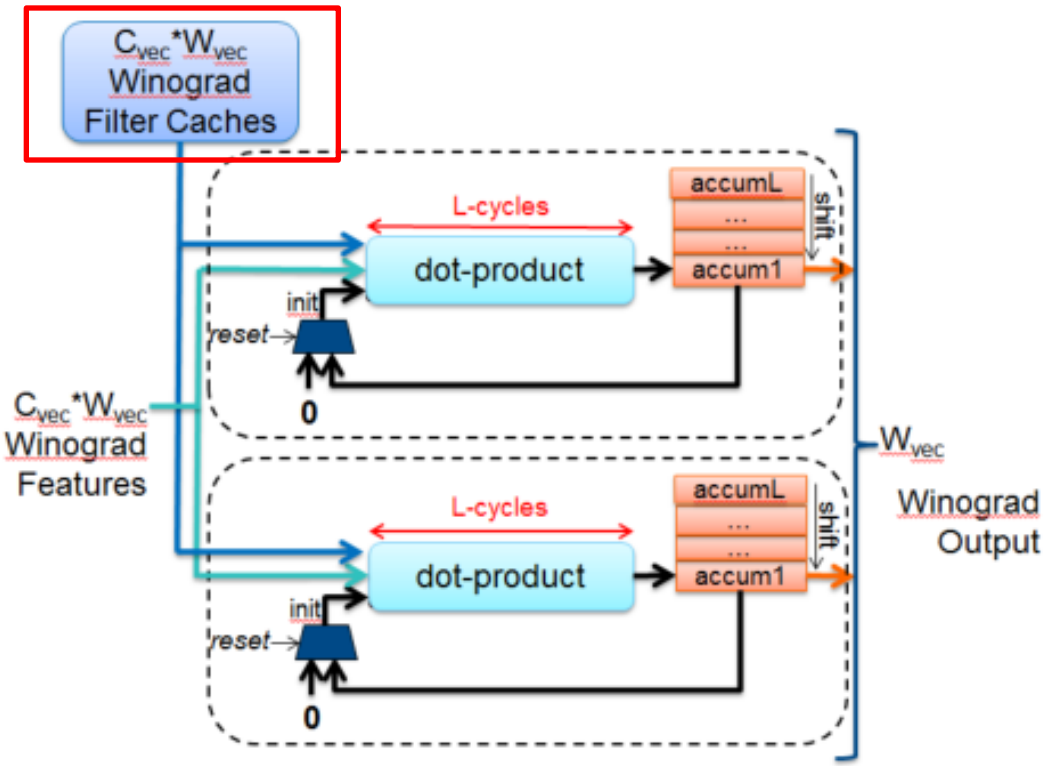
d_i and g_i are linearly mapped from i_i and f_i

Winograd Transform



With the Winograd Transform, a fixed-length convolution becomes a dot product

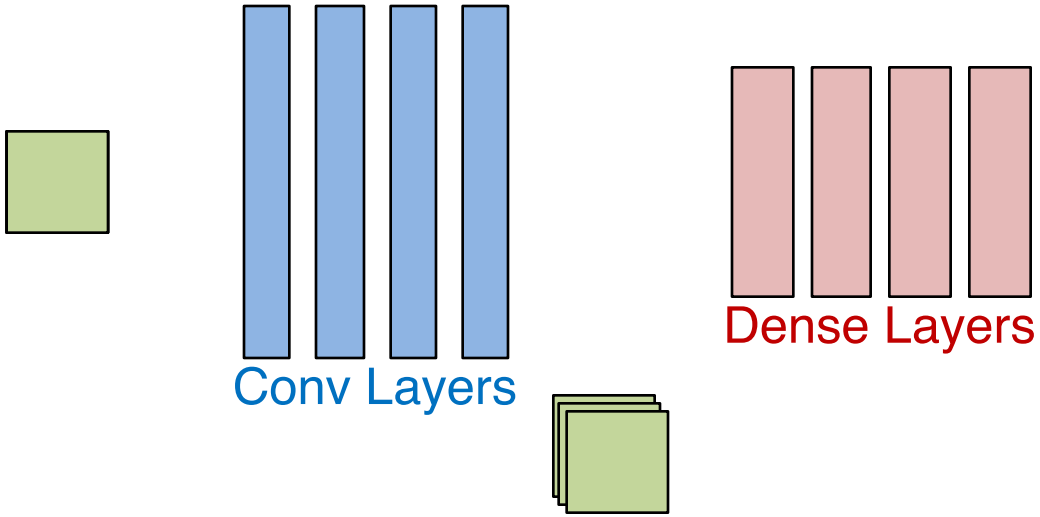
PE Execution (Convolution)



- Due to the use of 1D Winograd, the PEs in the DLA are 1D dot product engines
- For Conv, (transformed) filters are cached to exploit data reuse over an image

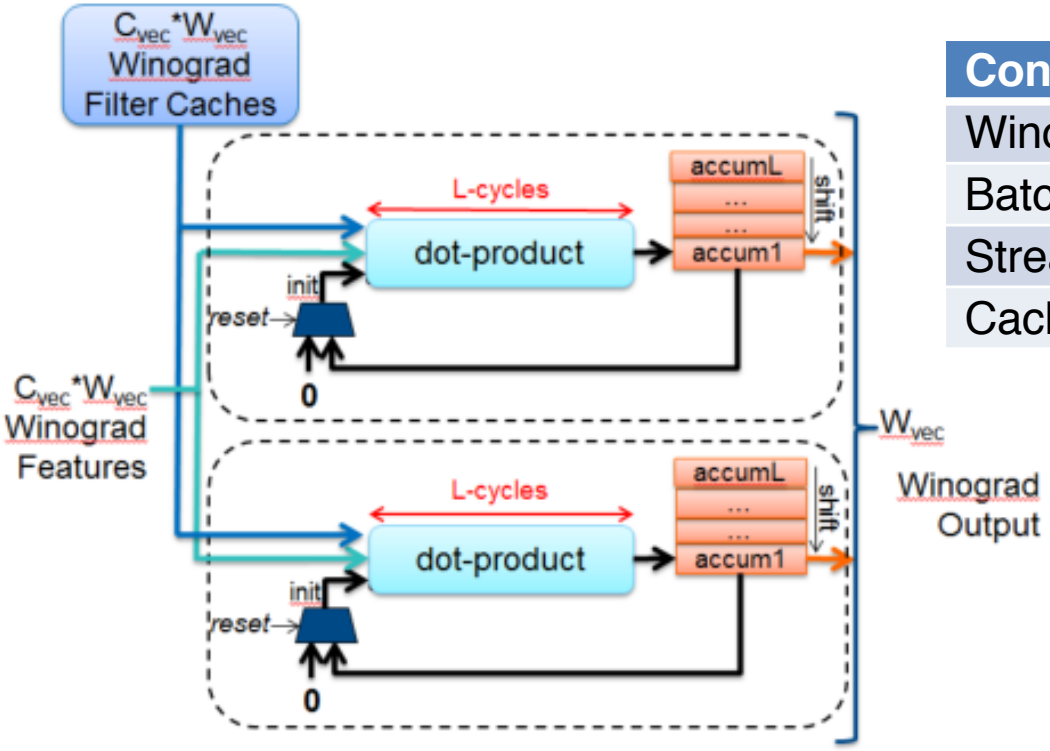
Handling the Dense Layers

- For Dense layers there is no filter reuse in one image, so the authors instead **batch S_{batch} images together**



- Images go through Conv layers one-by-one, results stored off-chip
- After S_{batch} images, the whole batch goes through Dense layers

PE Execution (Dense)



Configuration	Conv	Dense
Winograd?	Yes	No
Batch Size	1	S_{batch}
Streamed Data	Features	Filters
Cached Data	Filters	Features

- We reuse the same dot-product engine for the Dense layers
- Do not need the Winograd transform
- Opposite to the Conv layers, features are cached and filters are streamed over them

Experimental Evaluation

Platform	img/s	Power (W)	Energy Efficiency (img/s/W)
Arria 10 DLA (20nm)	1020	45	22.7
Nvidia Titan X (28nm)	5120	227	22.6
Nvidia M4 (28nm)	1150	58	19.8

Benchmark app is AlexNet

- Results show FPGAs can compete with GPUs in energy efficiency
- Titan X numbers ignore communication overhead and use random data instead of real images (highly optimistic)

Comparison to Previous Papers

	Zhang 2015	Qiu 2016	This Paper
Platform	Virtex7 VX485t	Zynq XC7Z045	Arria 10 1150
Clock (MHz)	100	150	303
Quantization	32-bit float	16-bit fixed	16-bit fixed
Performance (GOP/s)	61.6	137.0	1382
Power (W)	18.6	9.6	45
Energy Efficiency (GOP/J)	3.3	14.2	30.7

- Massive increase in performance due to Winograd Transform and storing all features on-chip
- Among the first to break the TeraOP/s barrier on FPGA

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

FPGA'17, Feb 2017

Main Contributions

- Implementation of binarized neural networks (BNNs) [1] on an embedded FPGA platform
- New model optimizations and hardware structures for BNNs
- Open-source HLS code available on GitHub

[1] 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.

CNNs with Reduced Numerical Precision

- Hardware architects widely apply fixed-point optimization for CNN acceleration
  - **Motivation**: both neural nets and image/video apps naturally tolerate small amounts of noise
  - **Approach**: take a trained floating-point model and apply quantization
    - 16 or 8-bit fixed-point have been shown to be practical

- Can we go even lower by training a reduced-numerical-precision CNN from the ground up?

Aggressively Quantized CNNs

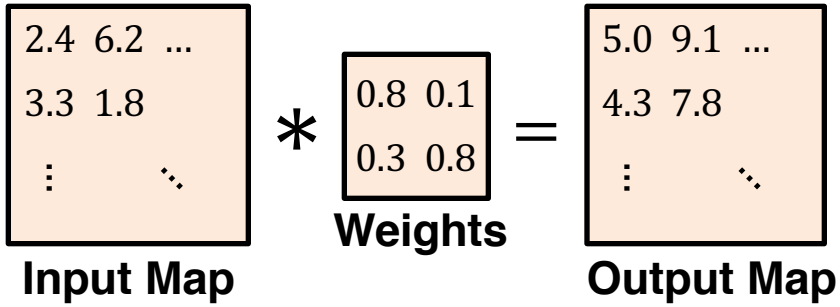
- **ML research papers:**
  - BinaryConnect [NIPS] Dec 2015
  - BNN [arXiv] Mar 2016
  - Ternary-Net [arXiv] May 2016
  - XNOR-Net [ECCV] Oct 2016
  - HWGQ [CVPR] Jul 2017
  - LR-Net [arXiv] Oct 2018
  - Many more!

Near state-of-the-art on MNIST, CIFAR-10 and SVHN at time of publication

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

CNN vs. BNN

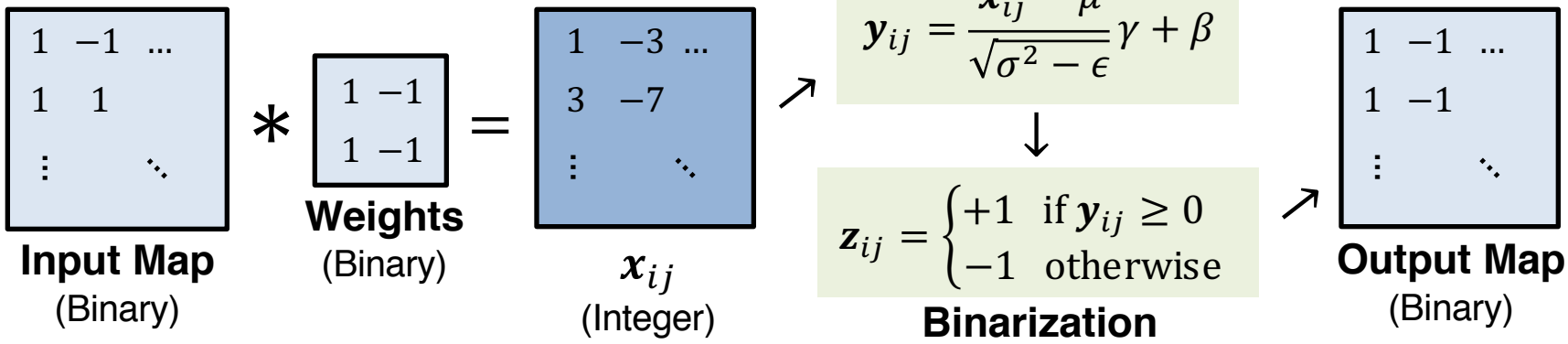
CNN



Key Differences

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

BNN



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 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

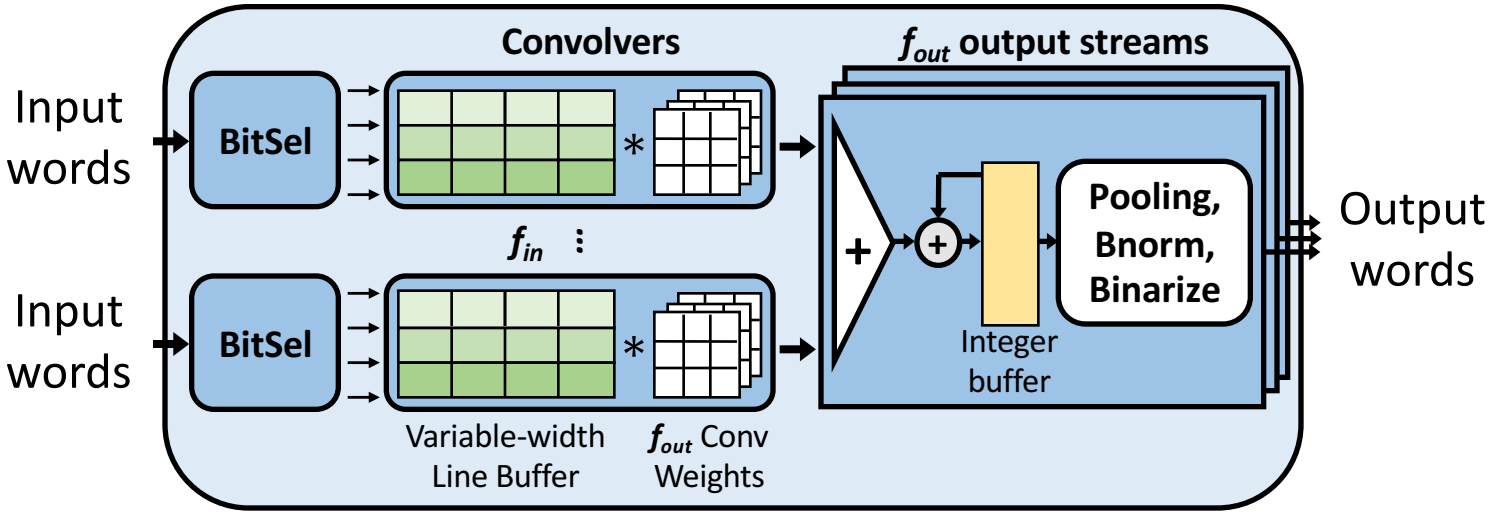
2. Binarized weights may reduce total model size

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

BNN Accelerator Design Goals

- **Target low-power embedded applications**
  - Design must be resource efficient to fit a small device
  - Execute layers sequentially on a single module
- **Optimize for batch size of 1**
  - Store all feature maps on-chip
    - Binarization makes feature maps smaller
  - Weights are streamed from off-chip storage
- **Synthesize RTL from C++ source**

BNN Accelerator Architecture



Challenges	Our Solution
Many diverse sources of parallelism (across/within images, feature maps, subword)	Highly parallel and pipelined architecture with a parameterized number of convolvers
Design must handle various layer types with different sized feature maps	Novel variable-width line buffer ensure pipeline is fully utilized
Slow interface between accelerator and general-purpose memory system	Careful memory layout and BitSel unit enable word-by-word data processing, instead of pixel-by-pixel

BNN HLS Design

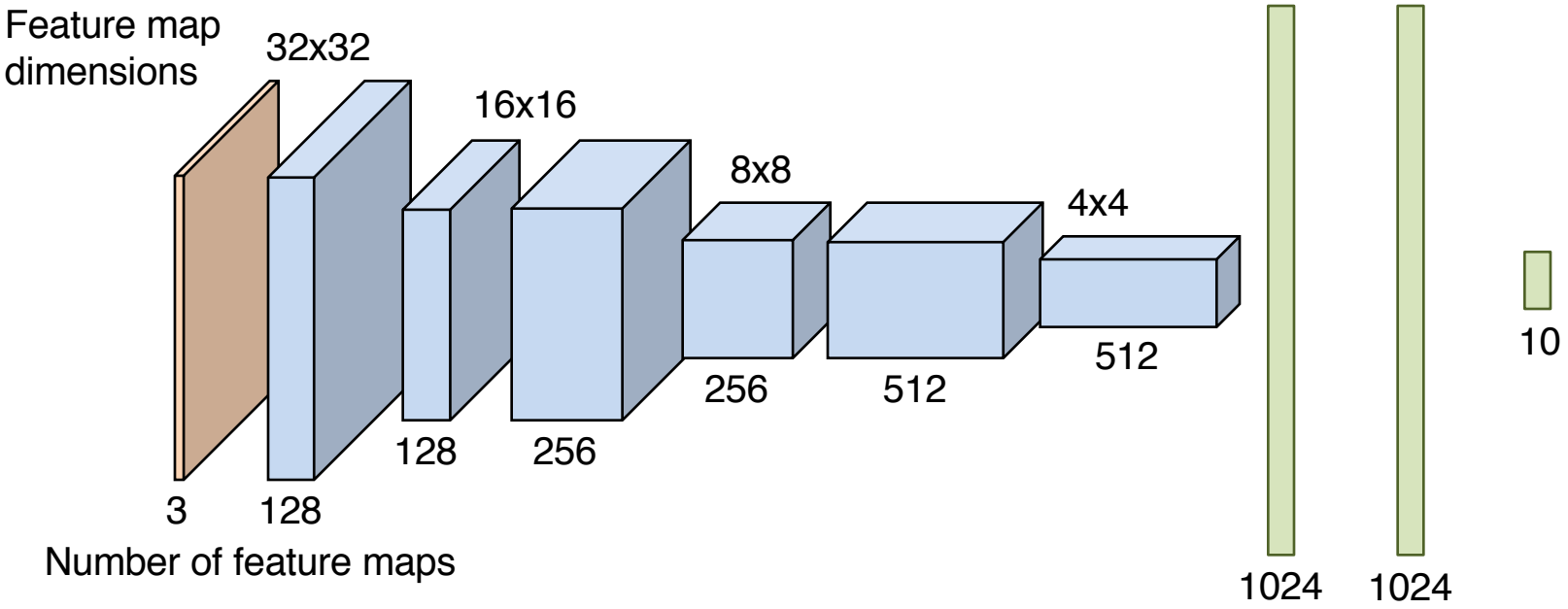
- **User writes and tests in C++**
  - CPU-FPGA interface automatically synthesized (by Xilinx SDSoC)
  - Significant reduction in verification time
    - BNN RTL takes days to simulate
- **Design effort**
  - One PhD student in 10 weeks

```
1  VariableLineBuffer linebuf;
2  ConvWeights wts;
3  IntegerBuffer outbuf;
4
5  for (i = 0; i < n_input_words; i++) {
6    #pragma HLS pipeline
7
8    // read input word, update linebuffer
9    WordType word = input_data[i];
10   BitSel(linebuf, word, input_width);
11
12   // update the weights each time we
13   // begin to process a new fmap
14   if (i % words_per_fmap == 0)
15     wts = weights[i / words_per_fmap];
16
17   // perform conv across linebuffer
18   for (c = 0; c < LINE_BUF_COLS; c++) {
19   #pragma HLS unroll
20     outbuf[i % words_per_fmap][c] +=
21       conv(c, linebuf, wts);
22   }
23 }
```

HLS code for part of convolver unit

https://github.com/cornell-zhang/bnn-fpga

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.

FPGA Implementation

Misc. HW Optimizations

1. Quantized the input image and batch norm params
2. Removed additive biases
3. Simplified batch norm computation

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%**

FPGA: ZedBoard with Xilinx Zynq-7000

mGPU: Jetson TK1 embedded GPU board

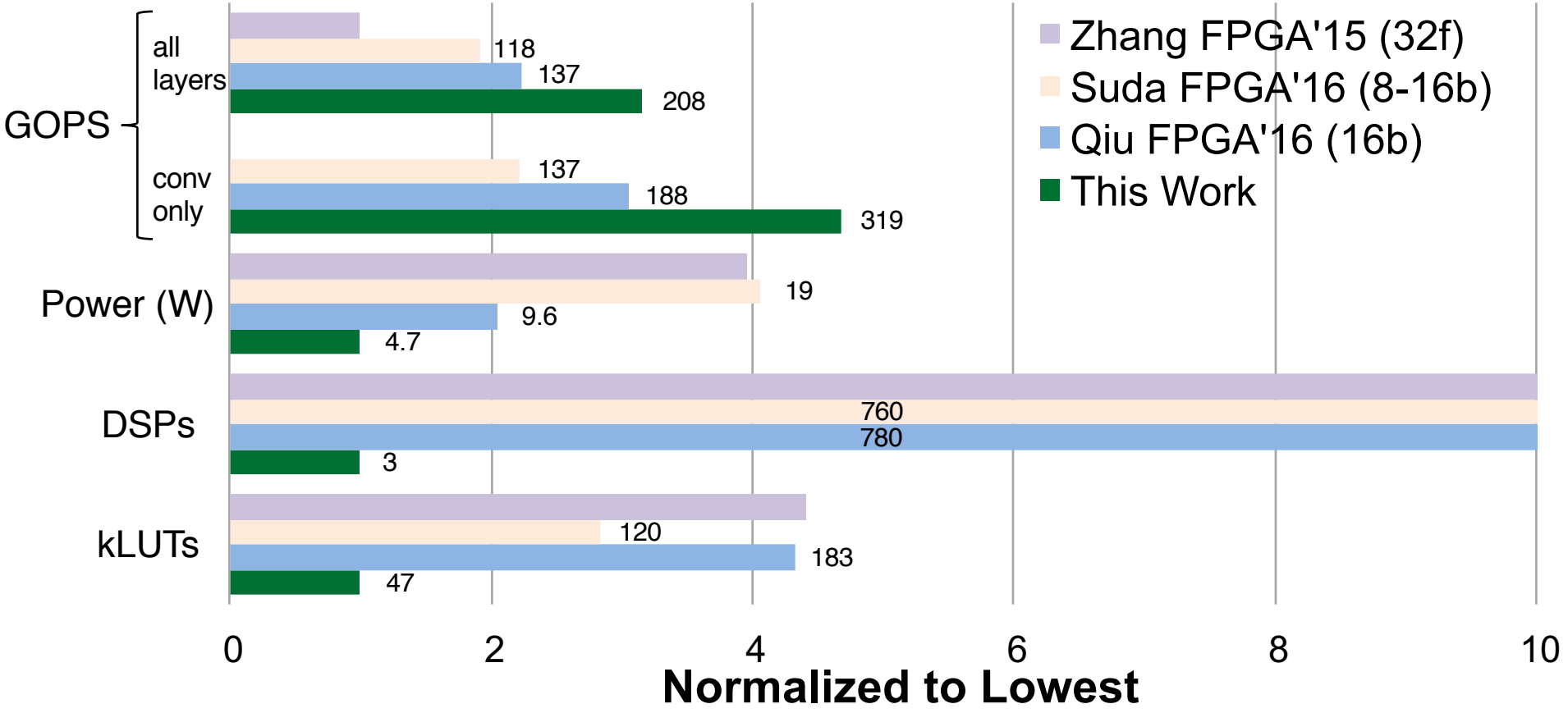
CPU: Intel Xeon E5-2640 multicore processor

GPU: NVIDIA Tesla K40 GPU

	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
Image/sec/Watt	3.1	0.71	5.8	**36**

R. Zhao et al., **Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs**. International Symposium on Field-Programmable Gate Arrays, Feb. 2017.

Comparison with CNNs on FPGA



- Performance metric is GigaOps Per Second (GOPS)
- Fair comparison is difficult
  - Binary vs. High-Precision
  - On-chip storage size differs across devices
  - But it shows the binary ops we can squeeze out of an FPGA board

BNN HLS Source Code

- **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

Quantized Networks on ImageNet

ImageNet Top-5 Accuracy		Weight		
		Binary (1b)	Ternary (b)	Full (32b)
Activation	1b	73.2% XNOR-Net [1] (ResNet-18: 89.2%)	-	-
	2b	85.9% HWGQ [2] (ResNet-50: 93%)	-	-
	32b	82.3% LR-Net [3] (ResNet-18: 89.2%)	84.8% LR-Net [3] (ResNet-18: 89.2%)	96.9% Inception V4 [4]

Closing the accuracy gap for high-resolution images: Ternary weights paired with higher precision activations are showing promise

[1] M. Rastegari, et al. XNOR-net: ImageNet classification using binary convolutional neural networks. *ECCV* 2016.
[2] Z. Cai, et al. Deep learning with low precision by half-wave Gaussian quantization. CVPR *20*17.
[3] O. Shayar, et al. Learning discrete weights using the local reparameterization Trick. *arXiv* 2017.
[4] C. Szegedy, et al. Inception-v4, Inception-ResNet and the impact of residual connections on learning. *AAAI* 2017.

RNN ACCELERATION

A Representative Work on LSTM

ESE: Efficient Speech Recognition Engine for Sparse LSTM on FPGA

Song Han[1,2], Junlong Kang[2], Huizi Mao[1], Yiming Hu[3], Xin Li[2], Yubin Li[2], Dongliang Xie[2], Hong Luo[2], Song Yao[2], Yu Wang[2,3], Huazhong Yang[2,3] and Bill Dally[1,4]

Stanford University[1], DeePhi[2], Tsinghua University[3], NVIDIA[4]

FPGA'17, Feb 2017

Recurrent Neural Networks Applications

speech recognition

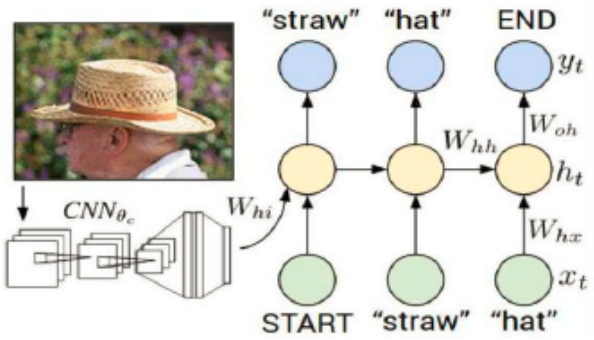
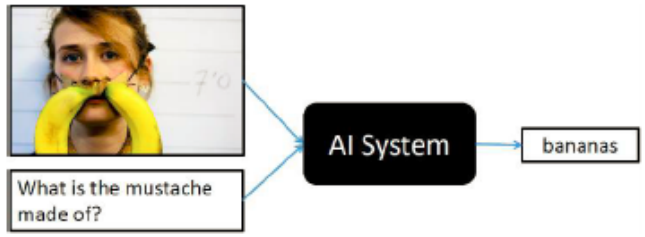


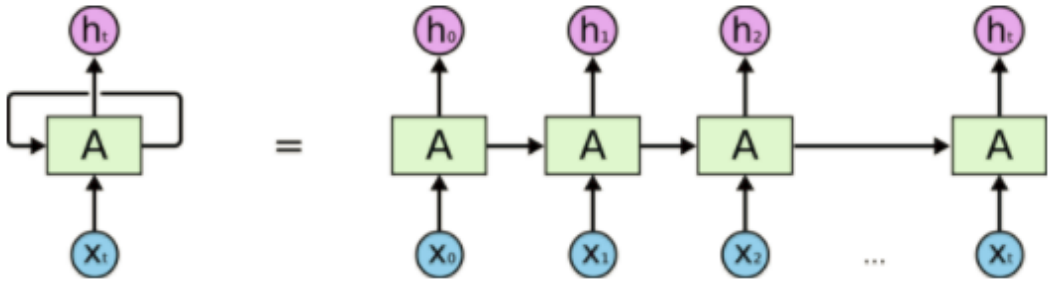
image caption

machine translation



visual question answering

Comparing CNN and RNN

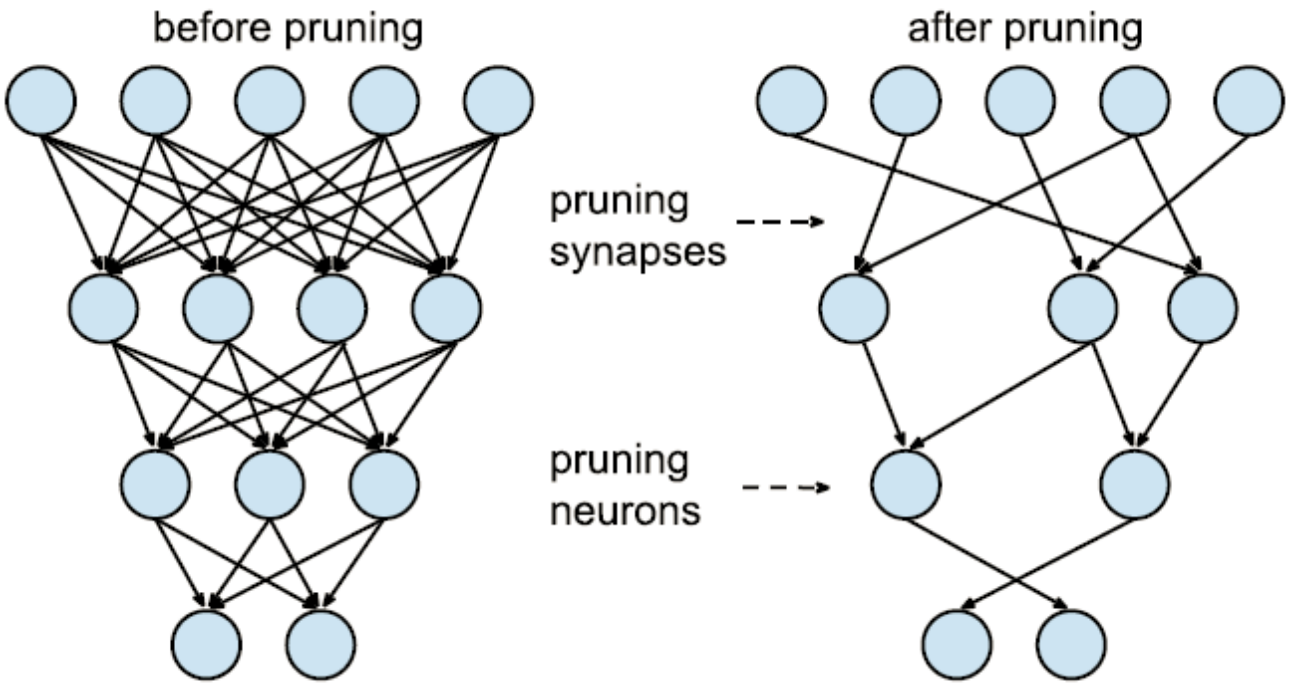


An unrolled recurrent neural network.

(Christopher Olah, "understanding LSTM")

- CNN: weights shared in space
- RNN/LSTM: weights shared in time
- => Produces complicated data dependency
- => Making parallelization difficult

Pruning Review

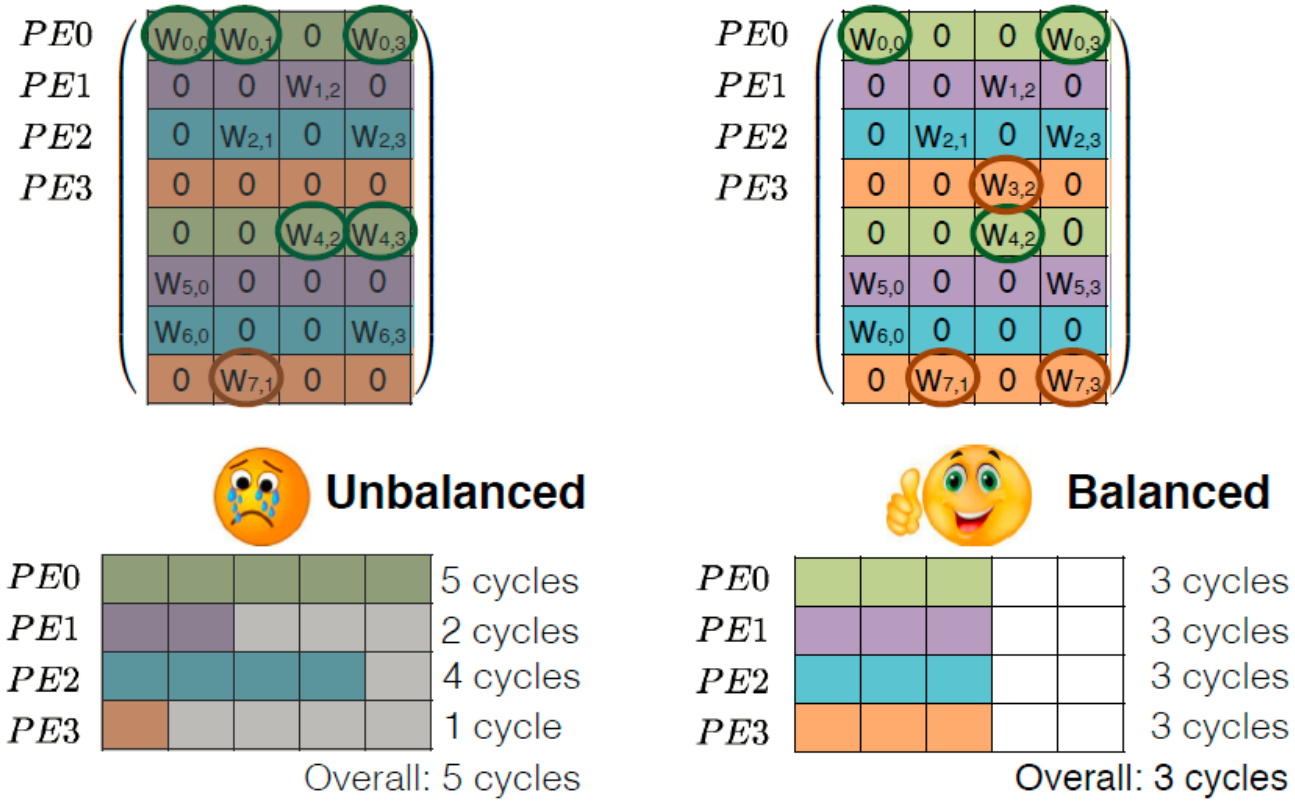


Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS'15

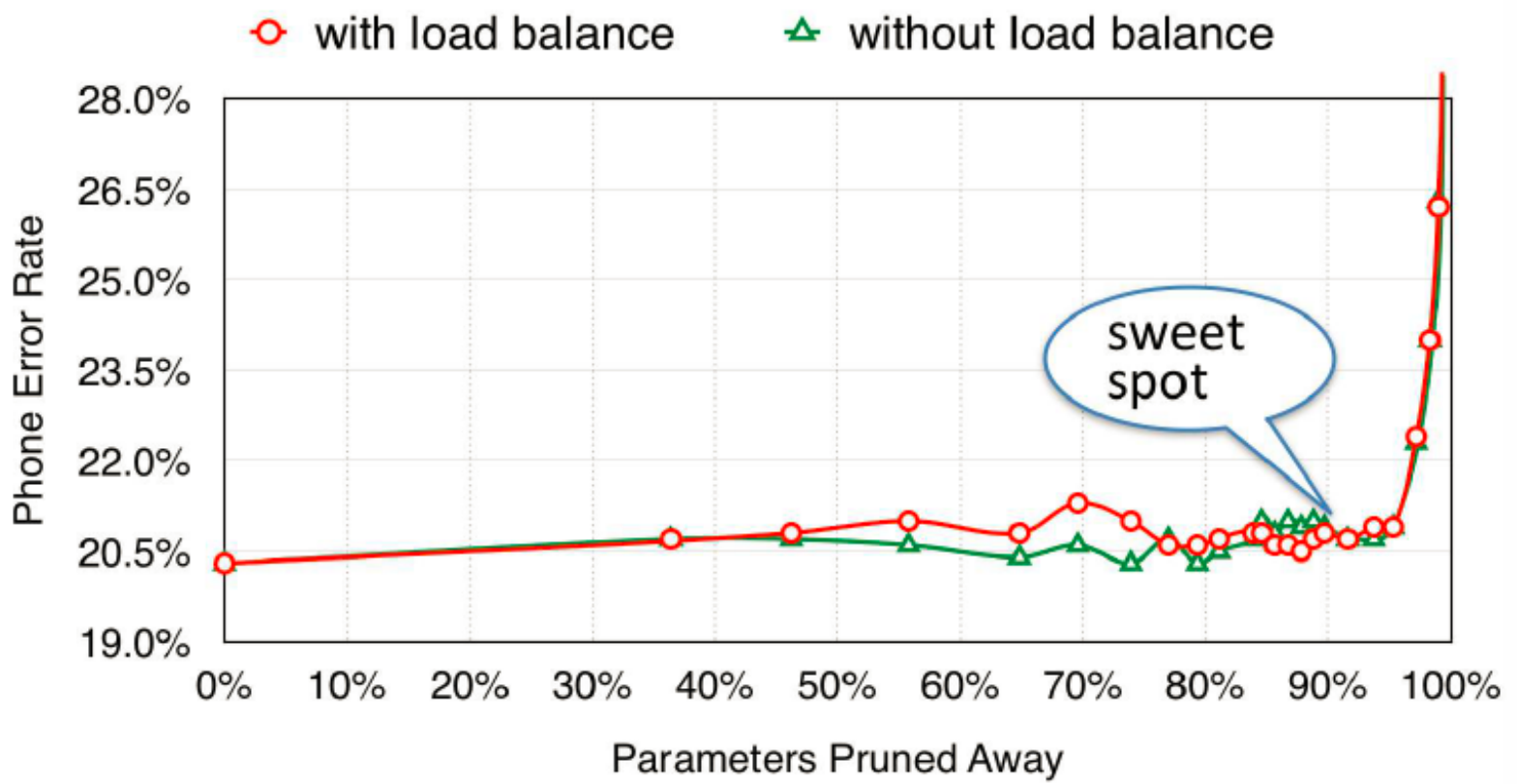
Proposed Techniques

- **Compression**
  - Load Balance-Aware Pruning
- **Quantization**
  - Quantize weights to 12bits fixed point numbers
- **Scheduling**
  - Overlap Computation and Memory Reference

Load Balance Aware Pruning



Accuracy vs. Sparsity



Weight Quantization

Word Error Rate

Networks	WER
32bit floating original network	20.3%
32bit floating pruned network	20.7%
16bit fixed pruned network	20.7%
12bit fixed pruned network	20.7%
8bit fixed pruned network	84.5%

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$
$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$
$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$
$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$
$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$
$$m_t = o_t \odot h(c_t) \quad (6)$$
$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t m_t	*N/A*
STATE	INITIAL		STATE_1			STATE_2			STATE_3	STATE_4			STATE_5	STATE_6

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{ox}	W_{ir}	W_{fr}	W_{or}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}	
		P	P	P	P	P	P	P	P	*N/A*		P	P	
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x	
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t m_t	*N/A*
STATE	**INITIAL**		**STATE_1**		**STATE_2**				**STATE_3**	**STATE_4**			**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t m_t	*N/A*
STATE	INITIAL		STATE_1		STATE_2				STATE_3	STATE_4			STATE_5	STATE_6

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$ h_t	o_t m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$ h_t	o_t m_t	*N/A*
STATE	**INITIAL**	**STATE_1**					**STATE_2**		**STATE_3**	**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$ h_t	o_t m_t	*N/A*
STATE	**INITIAL**		**STATE_1**				**STATE_2**		**STATE_3**	**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}	
		P	P	P	P	P	P	P	P	*N/A*		P	P	
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x	
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**			**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{ox}	W_{ir}	W_{fr}	W_{or}	W_{ox}	W_{or}	N/A			W_{ym}		W_{ix}
		P	P	P	P	P	P	P	P	N/A			P		P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	N/A			N/A		x
Computation	N/A		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	N/A		N/A		y_t
	N/A		N/A		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t	m_t	N/A
STATE	INITIAL		STATE_1			STATE_2			STATE_3	STATE_4			STATE_5		STATE_6

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$
$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$
$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$
$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$
$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$
$$m_t = o_t \odot h(c_t) \quad (6)$$
$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}		N/A	W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P		N/A	P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}		N/A	N/A	x
Computation	N/A		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	N/A	N/A	y_t
	N/A		N/A		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t	m_t	N/A
STATE	INITIAL		STATE_1			STATE_2			STATE_3		STATE_4		STATE_5		STATE_6

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{ox}	W_{ir}	W_{fr}	W_{or}	W_{ox}	W_{or}		*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P		*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}		*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*		*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**		**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{ox}	W_{ir}	W_{fr}	W_{or}	W_{ox}	W_{or}	*N/A*			W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*			P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*			*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*		*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t	m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**			**STATE_5**		**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*	W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*	P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*	*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$ h_t	o_t	m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*		W_{ym}	W_{ix}
		P	P	P	P	P	P	P	P	*N/A*		P	P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*		*N/A*	x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*	*N/A*	y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$ h_t	o_t	m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**		**STATE_5**	**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Scheduling

Memory

spMM

Elt-wise

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$
$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$
$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$
$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$
$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$
$$m_t = o_t \odot h(c_t) \quad (6)$$
$$y_t = W_{ym}m_t \quad (7)$$

Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{ox}	W_{ir}	W_{fr}	W_{or}	W_{ox}	W_{or}	*N/A*			W_{ym}		W_{ix}
		P	P	P	P	P	P	P	P	*N/A*			P		P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*			*N/A*		x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*		*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t	m_t	*N/A*
STATE	**INITIAL**		**STATE_1**			**STATE_2**			**STATE_3**	**STATE_4**			**STATE_5**		**STATE_6**

Sparse matrix-vector multiplication by *SpMV*

Accumulate operations by *Adder Tree*

Element-wise multiplication by *ElemMul*

Fetch data for the next operation

N/A Idle state

Speedup and Energy Efficiency

Plat.	ESE on FPGA (ours)									CPU		GPU	
Matrix	Matrix Size	Sparsity (%)[1]	Compres. Matrix (Bytes)[2]	Theoreti. Comput. Time (μs)	Real Comput. Time (μs)	Total Operat. (GOP)	Real Perform. (GOP/s)	Equ. Operat. (GOP)	Equ. Perform. (GOP/s)	Real Comput. Time (μs)		Real Comput. Time (μs)	
										Dense	Sparse	Dense	Sparse
W_{ix}	1024×153	11.7	18304	2.9	**5.36**	0.0012	218.6	0.010	1870.7	**1518.4**[3]	**670.4**	**34.2**	**58.0**
W_{fx}	1024×153	11.7	18272	2.9	**5.36**	0.0012	218.2	0.010	1870.7				
W_{cx}	1024×153	11.8	18560	2.9	**5.36**	0.0012	221.6	0.010	1870.7				
W_{ox}	1024×153	11.5	17984	2.8	**5.36**	0.0012	214.7	0.010	1870.7				
W_{ir}	1024×512	11.3	59360	9.3	**10.31**	0.0038	368.5	0.034	3254.6	**3225.0**[4]	**2288.0**	**81.3**	**166.0**
W_{fr}	1024×512	11.5	60416	9.4	**10.01**	0.0039	386.3	0.034	3352.1				
W_{cr}	1024×512	11.2	58880	9.2	**9.89**	0.0038	381.2	0.034	3394.5				
W_{or}	1024×512	11.5	60128	9.4	**10.04**	0.0038	383.5	0.034	3343.7				
W_{ym}	512×1024	10.0	52416	8.2	**15.66**	0.0034	214.2	0.034	2142.7	1273.9	611.5	124.8	63.4
Total	**3248128**	**11.2**	**364320**	**57.0**	**82.7**	**0.0233**	**282.2**	**0.208**	**2515.7**	**6017.3**	**3569.9**	**240.3**	**287.4**

	ESE	CPU		GPU	
		Dense	Sparse	Dense	Sparse
Latency	82.7us	6017us	3569us	240us	287us
Power	41W	111W	38W	202W	136W
Performance	2.9x	0.039	0.067	1x	0.84
Energy Efficiency	14.3x	0.071	0.355	1x	1.25
Compression Ratio	20x	1	10	1x	10

A Representative Work on LRCN

High-Performance Video Content Recognition with LRCN for FPGA

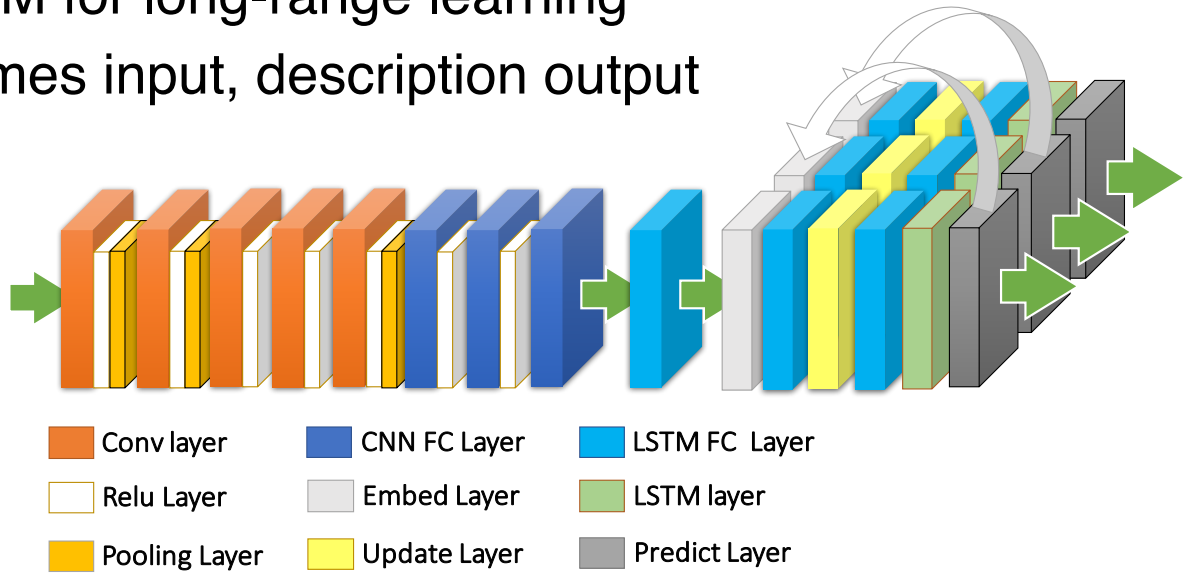
Xiaofan Zhang[1], Xinheng Liu[1], Anand Ramachandran[1], Chuanhao Zhuge[1], Shibin Tang[2], Peng Ouyang[2], Zuofu Cheng[3], Kyle Rupnow[3], and Deming Chen[1,3]

1: University of Illinois at Urbana-Champaign
2:Tsinghua University
3: Inspirit IoT, Inc.

FPL'17, Sep 2017

LRCN

- Long-term Recurrent Convolutional Network (LRCN) is a video recognition and description architecture which combines convolutional layers and long-range temporal recursion
  - 18 layers in CNN (Conv, Relu, Pooling, FC)
  - 7 layers in RNN (FC, Embed, Update, Predict)
  - LSTM for long-range learning
  - Frames input, description output



Design Challenge (1)

- **Computation complexity**

The memory and computational complexity of LRCN are very high for FPGA implementation.

2.22 GOPs with 86.56 million weight data (346.24MB)

Layers	# of Weights (M)	Giga Floating-point Ops	# of Input data(K)	# of Output data (K)
Conv1	0.03	0.21	150.53	290.40
Conv2	0.31	0.45	69.98	186.62
Conv3	0.89	0.30	43.26	64.90
Conv4	0.66	0.22	64.90	64.90
Conv5	0.44	0.15	64.90	43.26
FC1	37.76	0.08	9.22	4.10
FC2	16.79	0.03	4.10	4.10
FC3	4.10	0.01	4.10	1.00
RNN: 15 iterations	25.61	0.77	1.00	0.015
Total	**86.56**	**2.22**	**411.99**	**659.29**

Different layer types in LRCN show different characteristics regarding computation and memory requirements.

Layer	Comp.	Memory
CONV	60.06%	2.69%
FC	5.29%	67.73%
RNN	34.65%	29.58%

Design Challenge (2)

- **Unclear allocation scheme & partitioning, tiling factors**

An optimal implementation of the overall design requires careful resource allocation among the various loops.

The core computations share similar loop structures, but the good choices of partitioning & tiling factors vary from layer to layer

- **Limited on-chip memory & memory access bandwidth**

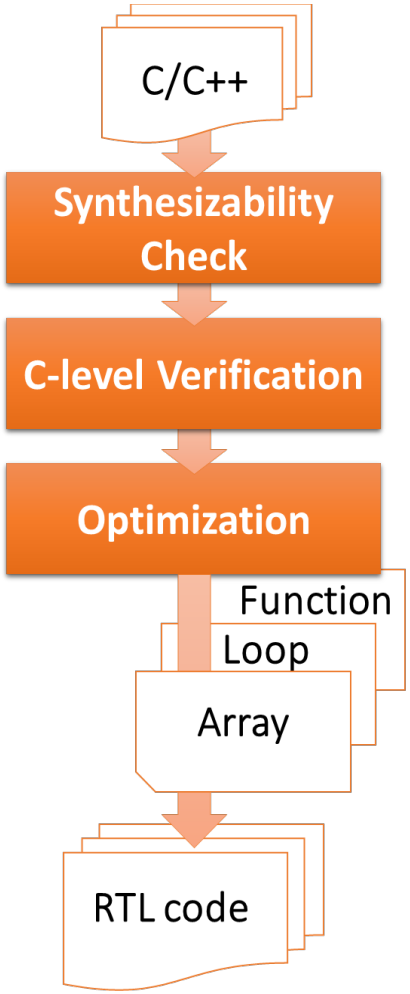
Not enough on-chip memory for storing weight & bias

External memory bandwidth becomes a bottleneck

HLS for LRCN

- Functions – reconstruction
- Loops – loop pipelining, loop unrolling and loop reordering
- Arrays – intermediate and input data partitioning
- Tradeoff – memory, DSP, FF, LUT resources and performance
- Computation/memory – overlapping and latency balancing

LRCN output: A group of young men playing a game of soccer



Resource Allocation!

- **REsource ALlocation Management (REALM)**

  - Analyze resource allocation among the layers
  - Determine the most efficient allocation to minimize total network latency given limitations in FPGA resource

$$Latency = \alpha \sum_i \frac{C_i}{R_i} \quad (1)$$

$$R_{total} = \sum_i R_i \quad (2)$$

$$\left[\sum_i \left(\sqrt{\frac{C_i}{R_i}}\right)^2\right]\left[\sum_i \left(\sqrt{R_i}\right)^2\right] \geq \left[\sum_i \sqrt{C_i}\right]^2 \quad (3)\ \text{(Cauchy inequality)}$$

$$\sum_i \frac{C_i}{R_i} \geq \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \quad (4)\ \text{(Simplify 3)}$$

$$Latency_{min} = \alpha \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \quad (5)\ \text{(Use (1) and (4))}$$

(6) REALM : $\frac{R_i}{R_j} = \frac{\sqrt{C_i}}{\sqrt{C_j}}$ satisfies (4) with equality

(To see this, use $R_i = \sqrt{C_i}\frac{R_{total}}{\sum_j \sqrt{C_j}}$ and plug into LHS of $\sum_i \frac{C_i}{R_i} = \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i}$)

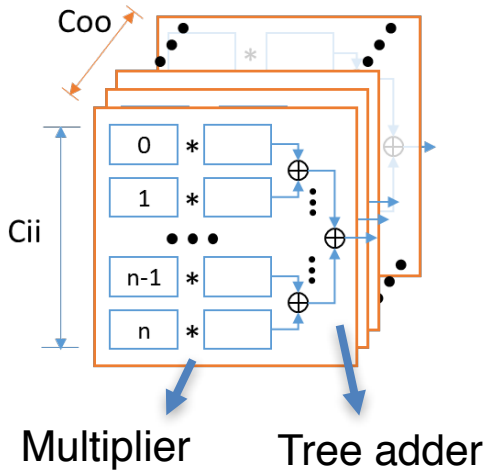
Computational demand of layer i is C_i
Resource consumed by that layer is R_i
Resource limitation R_{total}

Guideline provided by REALM

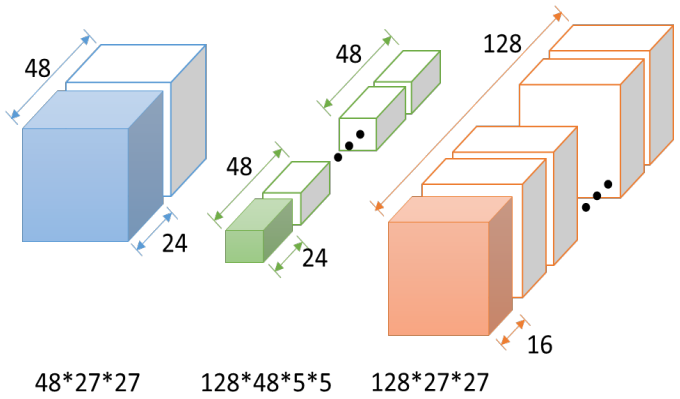
$$\frac{R_i}{R_j} = \frac{\sqrt{C_i}}{\sqrt{C_j}}$$

HLS IP for ML

- One IP is responsible for computing one tile in a layer
- IP consists of Coo multiply-accumulate units of dimension Cii each



Configurable in Coo & Cii dimensions



A complete convolutional layer is built using the IP

BLUE, GREEN, RED blocks make up one tile

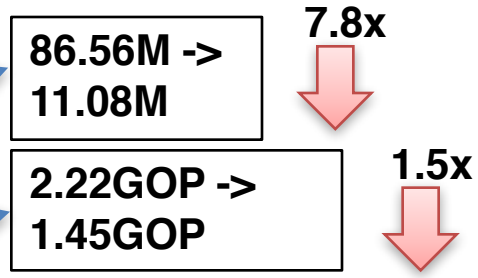
Other Techniques

- **Network pruning**: The number of output nodes in FC1 and FC3 are pruned from 4096 to 256. Two LSTM layers with 1000 hidden units each are converted to one LSTM layer with 256 hidden units.
  - 1% accuracy loss after retraining
- **Fixed point**: Use the fixed-point bit-widths: 16-bit for intermediate variables and 12-bit for weight & bias.
- **FIFO and Ping-Pong buffers**: Use FIFOs to pre-fetch some weights first. Ping-pong buffers are also applied at the input of each layer.

Methodology - Pruning

The number of output nodes in FC1 and FC3 are pruned from 4096 to 256. Two LSTM layers with 1000 hidden units each are converted to one LSTM layer with 256 hidden units.

Layers	# of Weights (M)	Complexity (GOP)	# of Input data(K)	# of Output data (K)
Conv1	0.03	0.21	150.53	290.40
Conv2	0.31	0.45	69.98	186.62
Conv3	0.89	0.30	43.26	64.90
Conv4	0.66	0.22	64.90	64.90
Conv5	0.44	0.15	64.90	43.26
FC1	2.36	0.005	9.22	0.26
FC2	0.07	0.0001	0.26	0.26
FC3	0.26	0.0005	0.26	1.00
RNN: 15 iterations	6.06	0.77	1.00	0.015
Total	**11.08**	**1.45**	**404.31**	**651.62**



Methodology - Quantization

Use the fixed-point bit-widths with different Q values (fractional bits)
16-bit for intermediate variables and 12-bit for weight & bias

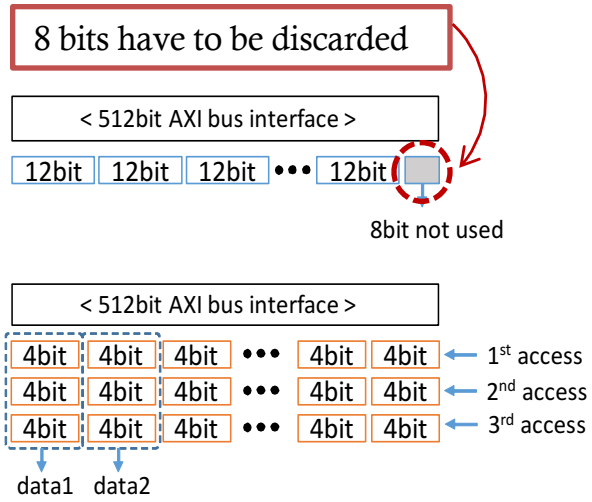
Layers	Output data (total bits, frac. bits)	Weight and Bias data (total bits, frac. bits)
Conv1	16, 4	12, 11
Conv2	16, 7	12, 11
Conv3	16, 8	12, 11
Conv4	16, 9	12, 11
Conv5	16, 10	12, 11
FC1	16, 11	12, 11
FC2	16, 11	12, 11
FC3	16, 11	12, 11
LSTM	16, 11	12, 11

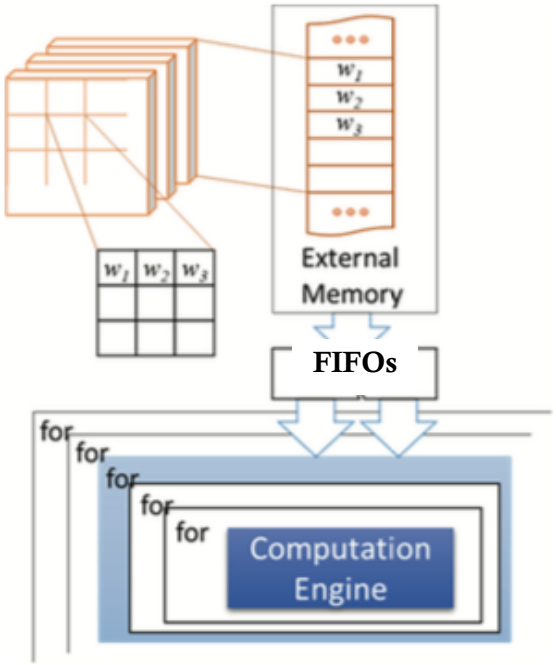
Accuracy after re-training

Network	Accuracy
LRCN–original (AlexNet+2 LSTM layers)	43%
LRCN–pruned, fixed-point (AlexNet + 1 LSTM layer) implemented on FPGA	42%

Methodology – Memory Access

FPGA accesses DDR memory via AXI bus with 512-bit interface
Memory organization: multi-dim -> linear sequence





Use FIFOs & Ping-pong buffers to hide memory access latency

To fully use the bus interface, collect bits from three bus accesses

System Implementation

- Front-end streams the image frames over the internet.
- At the back-end, the host PC receives the frames and launches the LRCN kernel implemented on the FPGA.

Internet
(TCP/IP)

Front-end, Tegra TK1 and webcam

Back-end, Xilinx VC709
FPGA board in host PC

Results

- **Performance and power comparisons between the CPU, GPU and FPGA versions**

	Frequency	TDP	Lithography
Xilinx Virtex-7	100MHz	-	28nm
Nvidia K80	562MHz	300W	28nm
Intel E5 2630-v2	2600MHz	95W	22nm

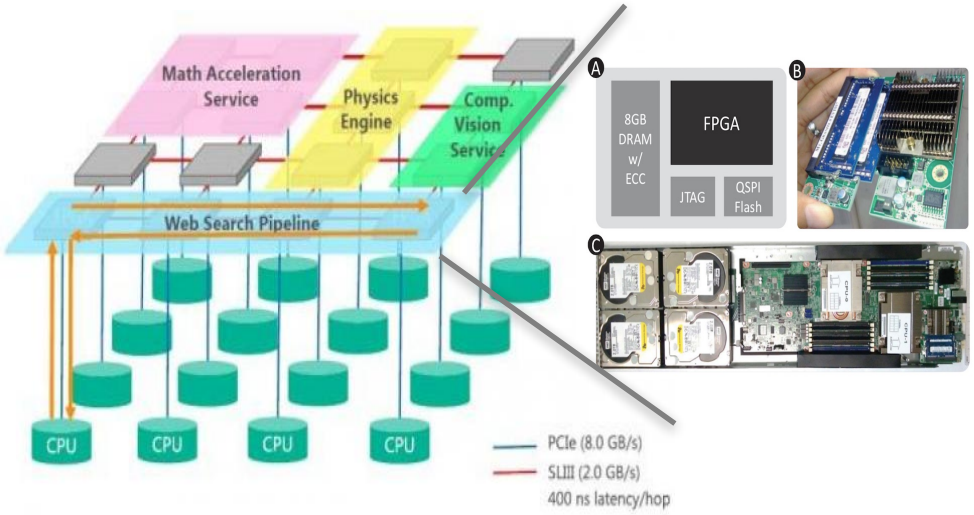
	Latency (Second)	Speedup	Real Power	Energy Efficiency
This work	0.040		23.6W	0.94 J/pic
NVidia K80	0.124	3.10X	133W	16.49 J/pic
Intel Xeon E5-2630 (Optimized)	0.190	4.75X	88W	16.72 J/pic

> 17.5X lower energy

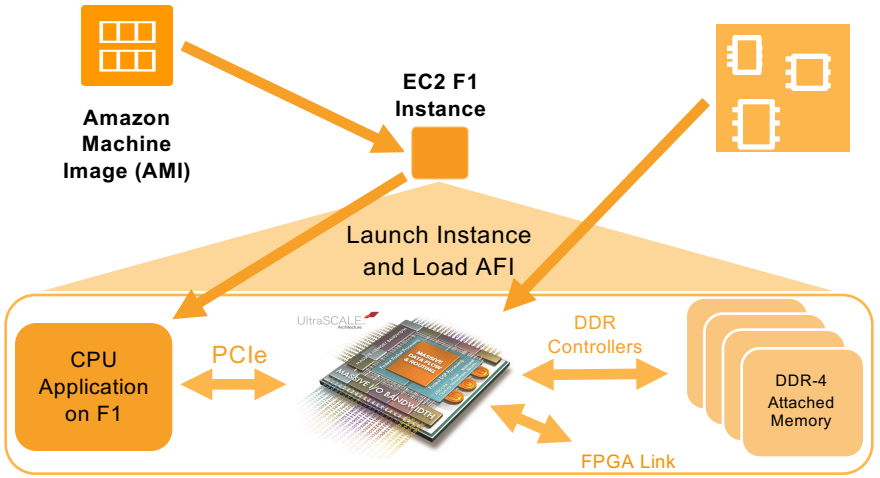
CONCLUDING REMARKS

Best of Times for FPGA-Based Computing

- Growing demand for energy-efficient compute acceleration
  - Hardware specialization required for a rich set of compute-intensive apps in edge devices and datacenters



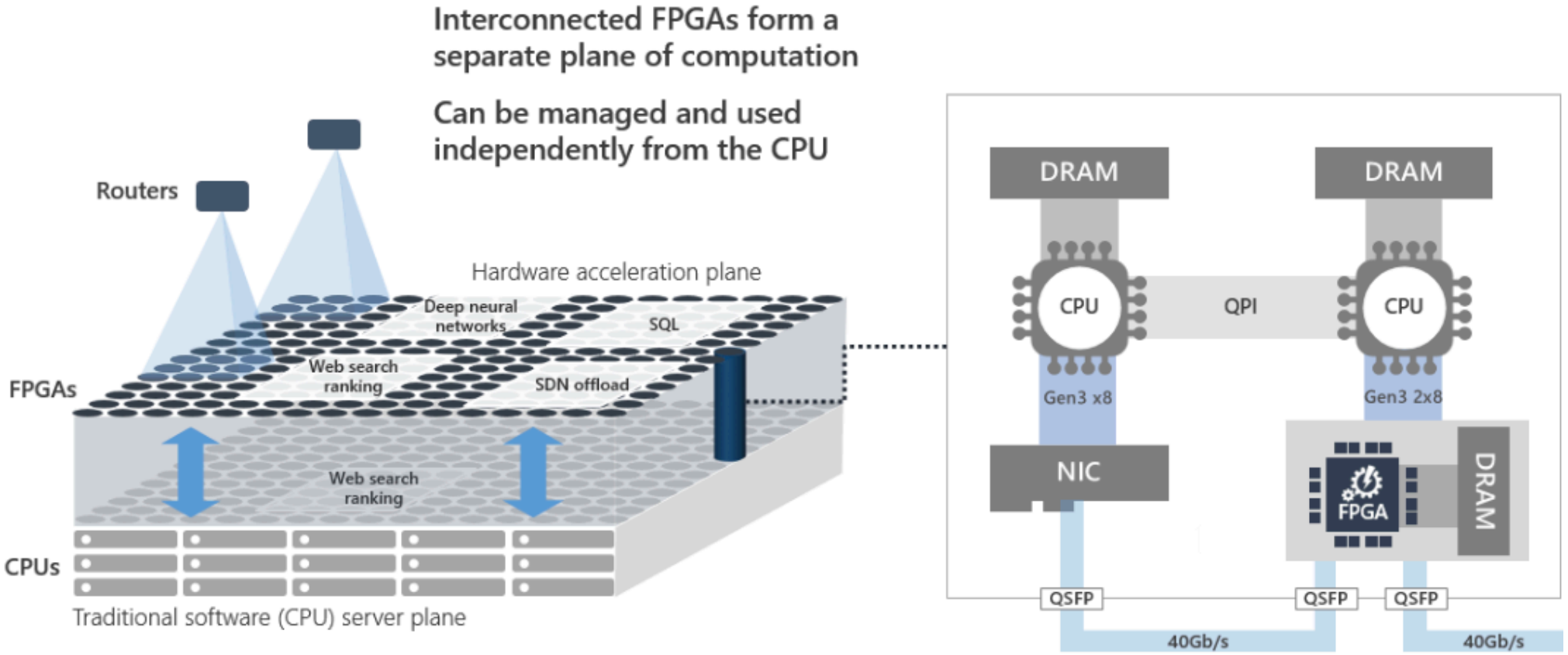
Microsoft Project Catapult



Amazon EC2 F1 instances

Example: Microsoft Deploying FPGAs in Datacenter

- FPGAs massively deployed in Microsoft datacenters to accelerate various web, database, and AI services
  - e.g., project BrainWave achieved ~40Teraflops on large LSTM using Intel Stratix 10 FPGAs



source: www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave

"Worst of Times" Also?

Software programmability is paramount

- Target of acceleration is moving fast!
  - Deep learning algorithms & frameworks are evolving at a breathtaking pace

Imagenet classification with deep convolutional neural networks
A Krizhevsky, I Sutskever, GE Hinton - Advances in neural ..., 2012 - papers.nips.cc
... Thus far, our results have improved as we have made our network **larger** and trained it longer but we still have many orders of magnitude to go in order to match the infero-temporal pathway of the human visual system. ... **ImageNet**: A **Large-Scale** Hierarchical Image Database. ...
Cited by 15127 Related articles All 95 versions Cite Save

Very deep convolutional networks for large-scale image recognition
K Simonyan, A Zisserman - arXiv preprint arXiv:1409.1556, 2014 - arxiv.org
Abstract: In this work we investigate the effect of the **convolutional network** depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of **networks** of increasing depth using an architecture with **very** small (3x3)
Cited by 6274 Related articles All 14 versions Cite Save

Deep **residual learning** for image recognition
K He, X Zhang, S Ren, J Sun - ... of the IEEE conference on computer ..., 2016 - cv-foundation.org
Abstract Deeper neural networks are more difficult to train. We present a **residual learning** framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as **learning residual** functions with reference
Cited by 3659 Related articles All 20 versions Cite Save More

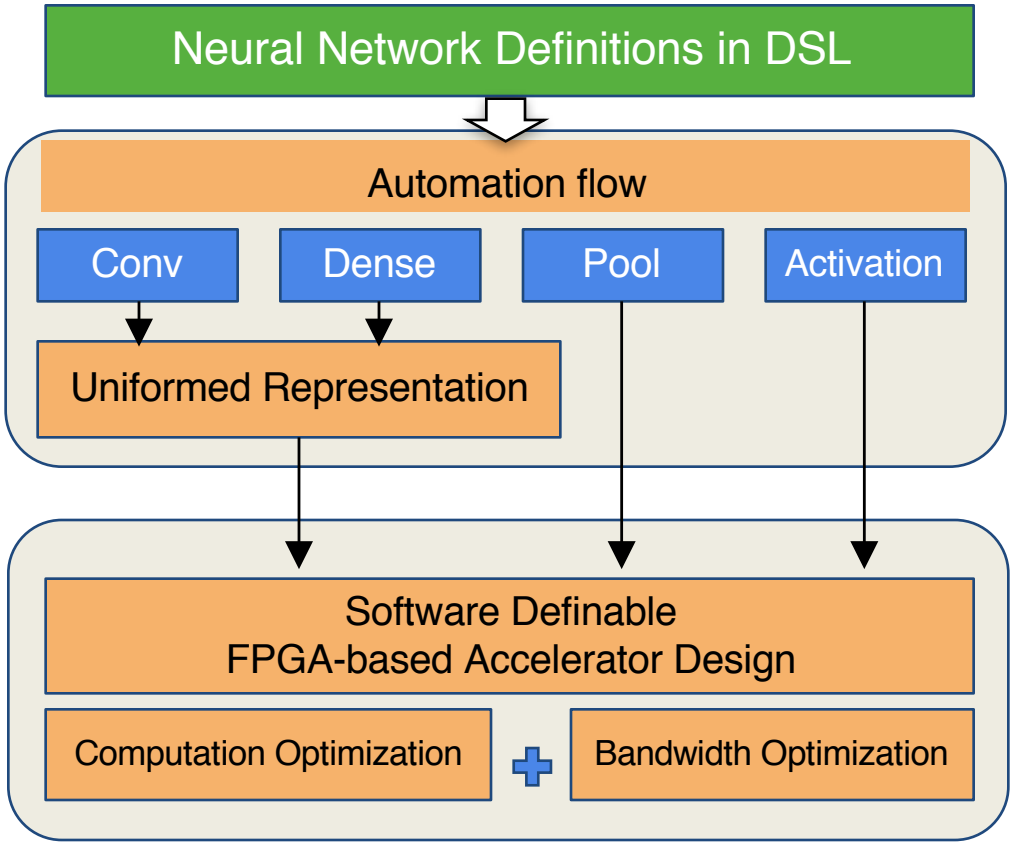
Challenges of FPGA-Based DNN Acceleration

- Challenges intrinsic to DNNs:
  - Network structures vary across different applications (e.g., AlexNet, ResNet, DenseNet, MobileNets)
  - Different layers have distinct compute/memory access patterns
    - Conv layers are compute bound, whereas dense layers are memory bound

- FPGA design is very hard for software developers
  - Efficient allocation and utilization of heterogeneous resources (LUTs, DSPs, RAMs)
  - Timing closure
  - Performance and functional debugging
  - ...

Need for Domain-Specific Programming

- **Ideally**, DNN developers should use their familiar programming interface with a domain-specific language (DSL) without considering hardware details, and the optimized accelerator is automatically generated



Takeaway Points

- Challenges & opportunities are abound for deep learning acceleration with FPGAs
- Algorithm-hardware co-design is the “secrete” sauce for designing a highly efficient soft DPU
  - Reduced-precision DNNs on FPGA show great promise in delivering high performance and energy efficiency
- Software programmability is vital with HLS being a key enabler

~~“HARD”~~ **From Software to**

“SOFT” **Hardware**

Parallelism (dependence)	**Resource** (FU, memory, I/O)	**Data Organization** (access pattern)
Parallelization	Pipelining	Data Reuse
		HW/SW Partition

Additional Useful Resources

- Recent papers on neural networks on silicon
  - https://github.com/fengbintu/Neural-Networks-on-Silicon
- Tutorial on hardware architectures for DNNs
  - http://eyeriss.mit.edu/tutorial.html
- Landscape of neural network inference accelerators
  - https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator
- Most cited deep learning papers (since 2012)
  - https://github.com/terryum/awesome-deep-learning-papers

Acknowledgements

- This tutorial contains/adapts materials developed by
  - **Ritchie Zhao** (PhD student at Cornell)
  - **Wei Zuo and Ashutosh Dhar** (PhD students at UIUC)
  - Authors of the following papers
    - Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks (FPGA'15, PKU-UCLA)
    - Going Deeper with Embedded FPGA Platform for Convolutional Neural Network (FPGA'16, Tsinghua-MSRA)
    - An OpenCL Deep Learning Accelerator on Arria 10 (FPGA'17, Intel)
    - Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks (ICCAD'16, PKU-UCLA-Falcon)
    - ESE: Efficient Speech Recognition Engine for Sparse LSTM on FPGA (FPGA'17, Stanford-DeePhi-Tsinghua-NVidia)
    - High-Performance Video Content Recognition with LRCN for FPGA (FPL'17, UIUC-Inspirit-Tsinghua)

END