ECE 6775
High-Level Digital Design Automation
Fall 2023

DNN Acceleration on FPGAs

Cornell University

CSL

Announcements

- Lab 4 on BNN acceleration
  - Use the latest zip file from the class folder

- Start forming teams for the final project
  - 3 or 4 students per team

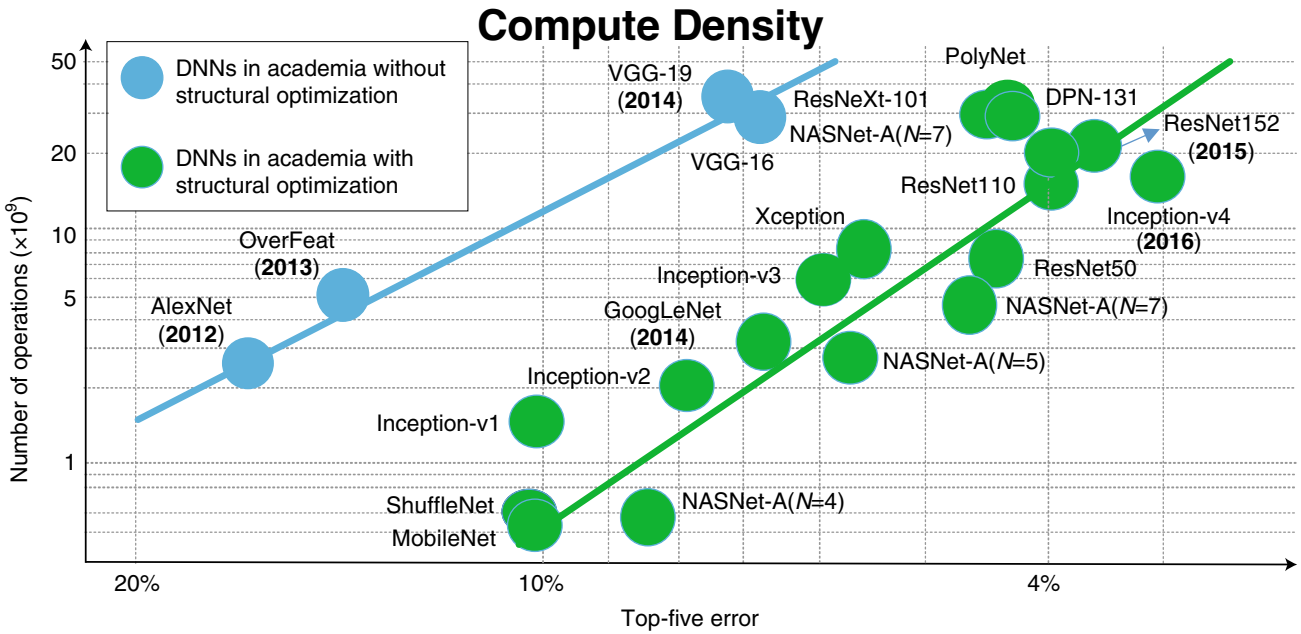
Course Roadmap

- **Background**
  - Introduction
  - Hardware specialization
  - Algorithm basics
- **High-level synthesis**
  - C-based synthesis for FPGAs
  - Front-end compilation
  - Scheduling
  - Resource sharing
  - Pipelining
- **Advanced topics**
  - Deep learning acceleration
  - Domain-specific programming

Agenda

- CNN acceleration on FPGAs
- Overview of BNN acceleration

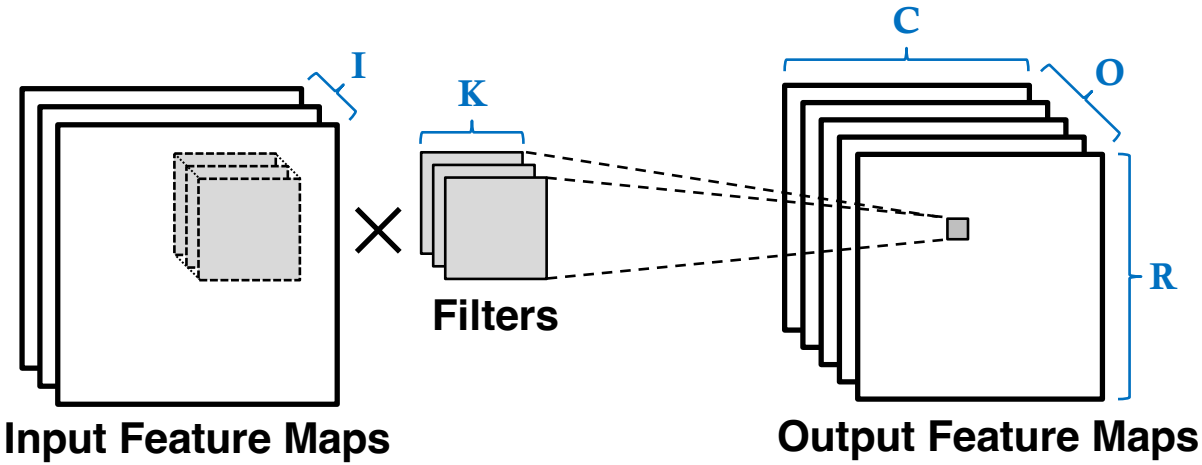
Modern ML Models are Computationally Expensive



X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi. **Scaling for Edge Inference of Neural Networks**. *Nature Electronics*, vol 1, Apr 2018.

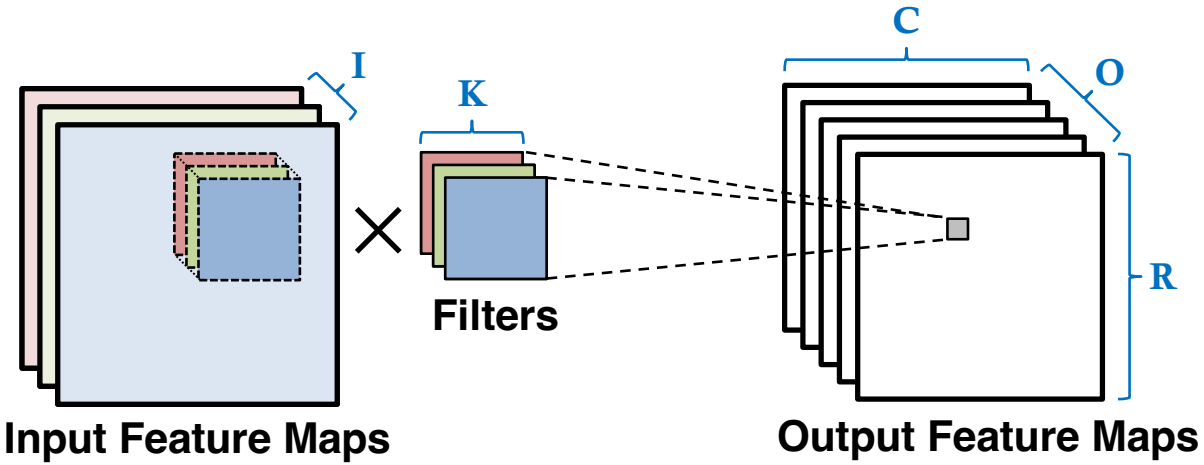
- For example, ResNet50, a 70-layer CNN model performs 7.7 billion operations required to classify one image

Convolutional Layer



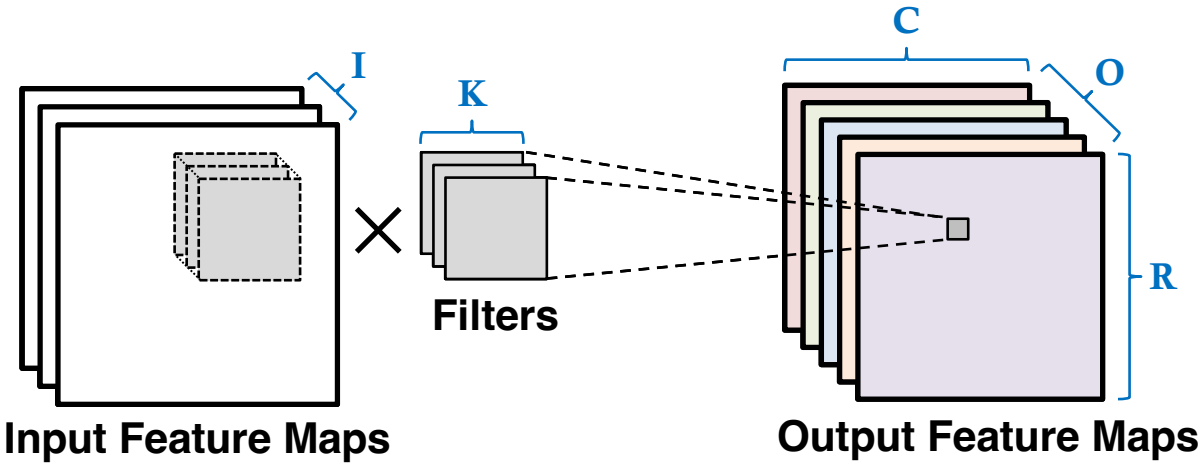
- An output pixel is connected to its neighboring region on each input feature map (fmap)
- All pixels on an output feature map use the same filter weights

Parallelism in the Convolutional Layer



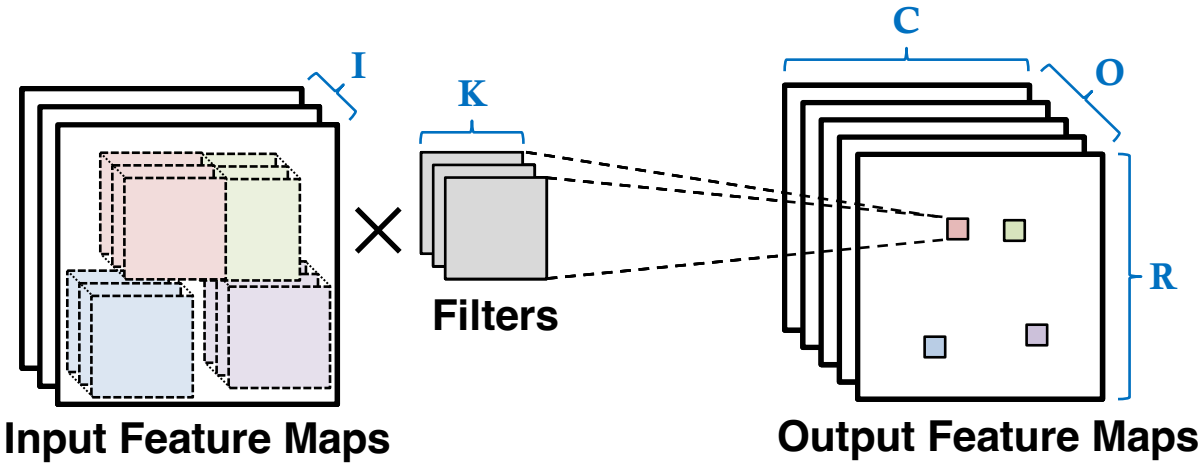
- Four main sources of parallelism
  1. **Across input feature maps (i.e., input channels)**

Parallelism in the Convolutional Layer



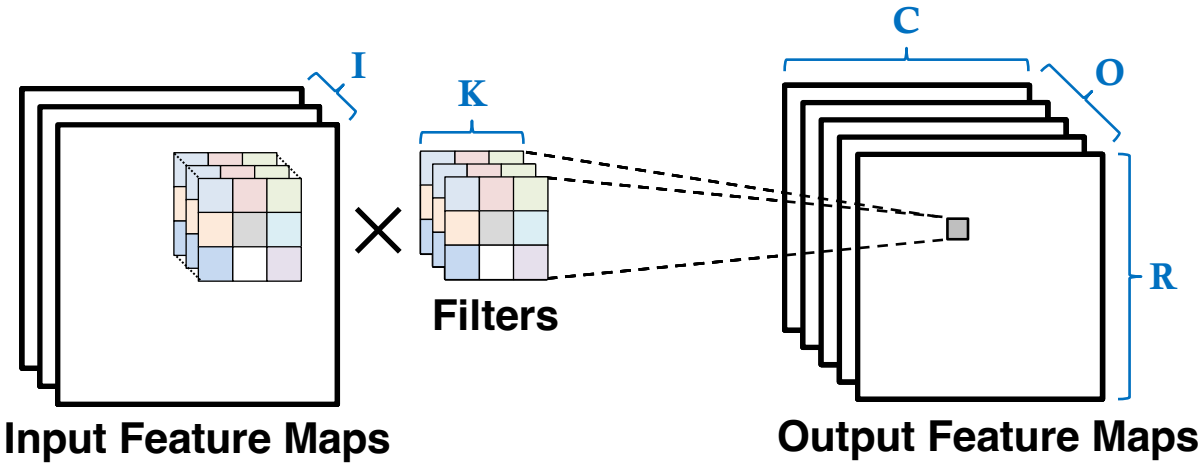
- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. **Across output feature maps (i.e., output channels)**

Parallelism in the Convolutional Layer



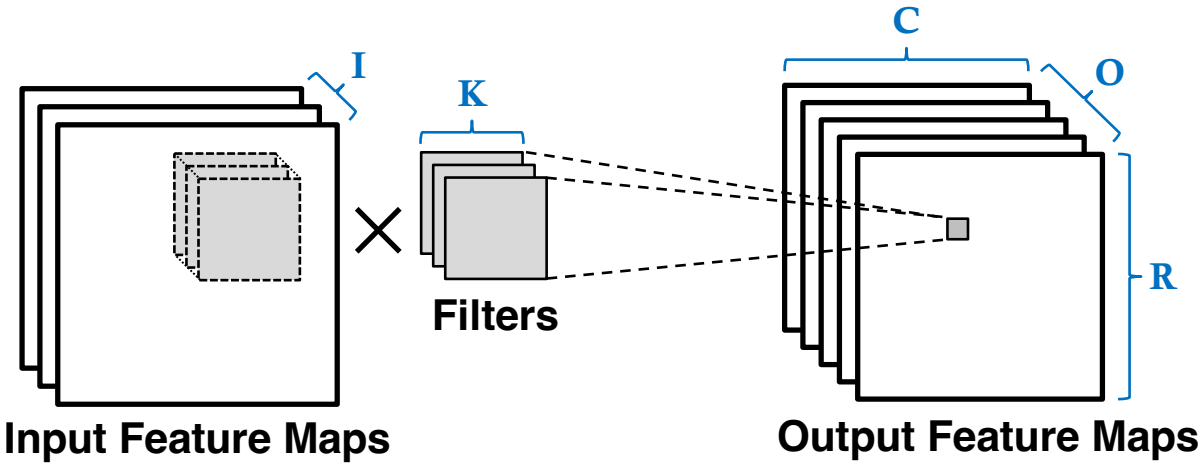
- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. Across output feature maps (i.e., output channels)
  3. **Across different output pixels (i.e., filter positions)**

Parallelism in the Convolutional Layer



- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. Across output feature maps (i.e., output channels)
  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 channels

Loop over input channels

Loop over filter pixels

S is the stride

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, UCLA
[3]PKU/UCLA Joint Research Institute in Science and Engineering

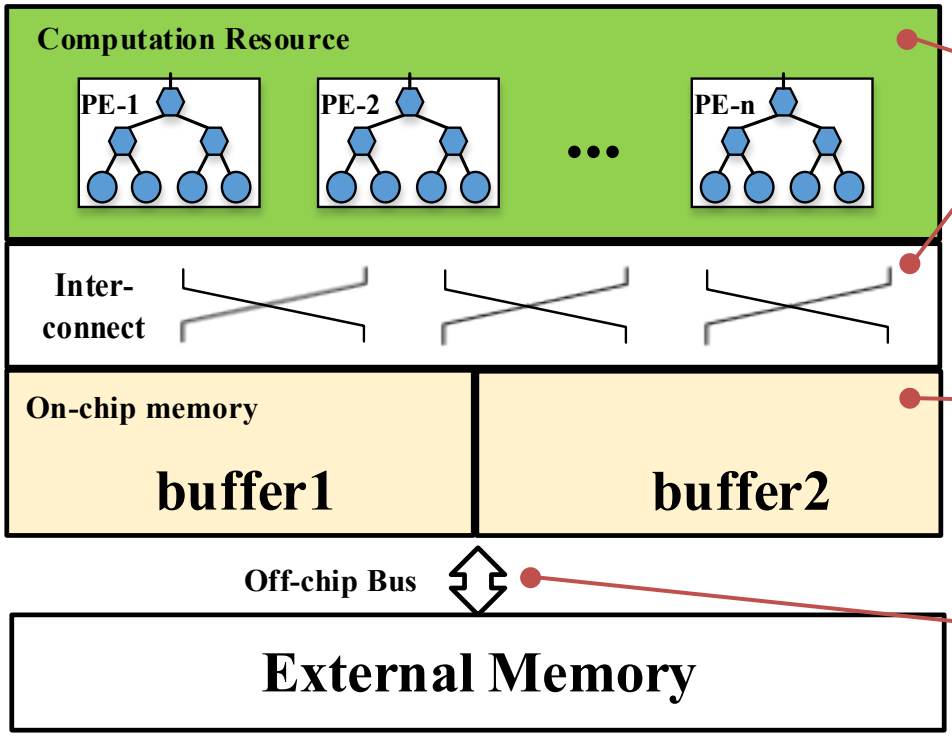
FPGA'15, Feb 2015

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

Challenges to FPGA Acceleration

- 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

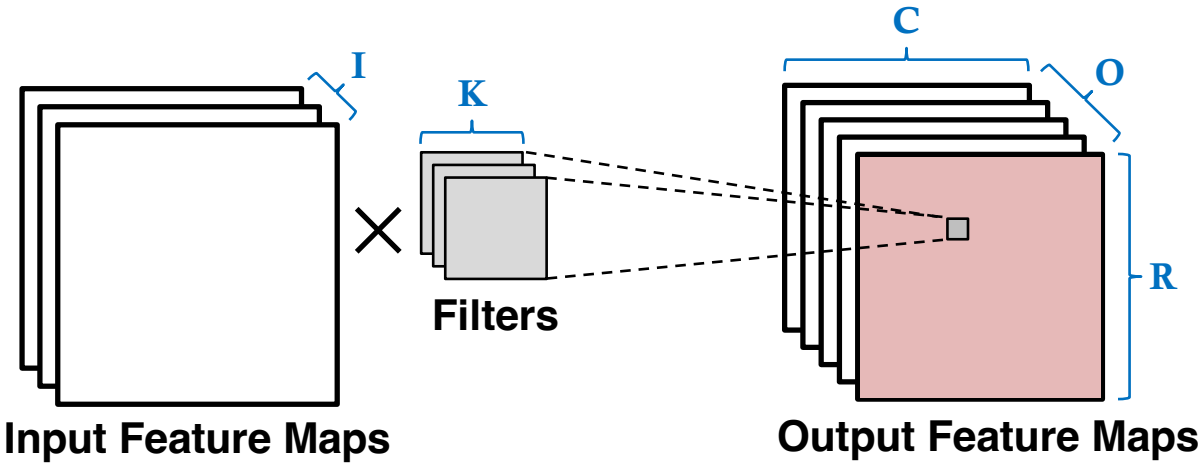
Limited on-chip storage

Technique: loop tiling

Limited off-chip bandwidth

Technique: data reuse, compute-memory balance

Loop Tiling

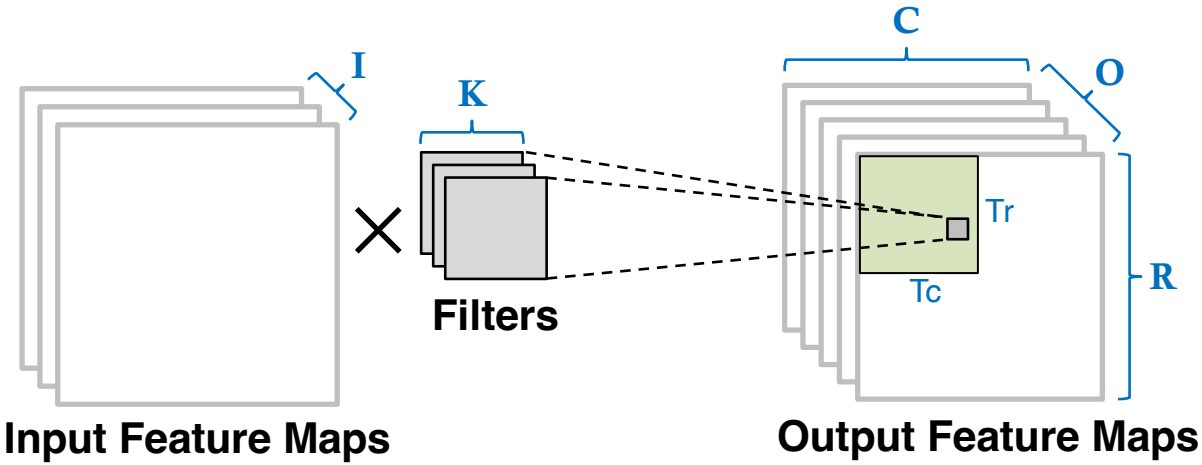



```
1 for (row=0; row<R; row++) {
2    for (col=0; col<C; col++) {
3       for (to=0; to<O; to++) {
4          for (ti=0; ti<I; ti++) {
5             for (ki=0; ki<K; ki++) {
6                for (kj=0; kj<K; kj++) {
                    output_fm[to][row][col] +=
                    weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];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

Loop over **pixels** in an output fmap

Loop Tiling

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




```
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(col+Tc, C); tcc++) {
7             for (ki=0; ki<K; ki++) {
8               for (kj=0; kj<K; kj++) {
                  output_fm[to][trr][tcc] +=
                  weights[to][ti][ki][kj]*input_fm[ti][S*trr+ki][S*tcc+kj];
          }}}}}}
}}}}
```

Loop over different **tiles**

Loops over **pixels** in each tile

Code with Loop Tiling

Tile sizes: rows (Tr), columns (Tc), input channels (Ti), and output channels (To)

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

5             for (trr=row; trr<min(row+Tr, R); trr++) {
6                for (tcc=col; tcc<min(col+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

FPGA Portion

Optimizing for On-Chip Performance

```
5           for (trr=row; trr<min(row+Tr, R); trr++) {
6              for (tcc=col; tcc<min(col+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(col+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(col+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(col+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(col+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];
            }}}}}}
```

Parallelize across input (Ti) and output (To) channels

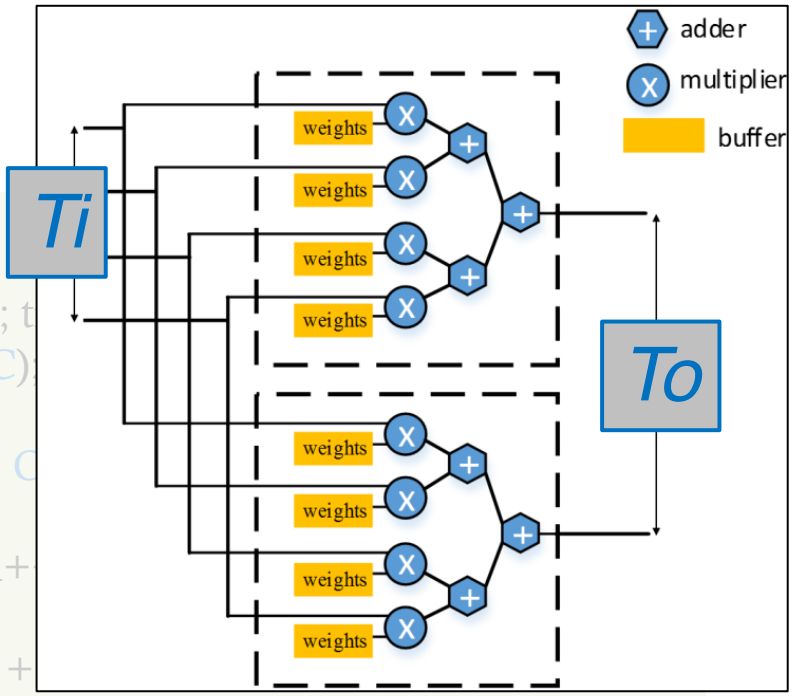
Number of cycles to execute the above loop nest

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

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

Optimizing for On-Chip Performance

```
9           for (ki=0; ki<K; ki++) {
10            for (kj=0; kj<K; kj++) {
5               for (trr=row; trr<min(row+Tr, R); t
6                 for (tcc=col; tcc<min(col+Tc, C)
                    #pragma HLS pipeline
7                   for (too=to; too<min(to+To, C
                      #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];
          }}}}}}
```



Generated Hardware

Performance and size of the accelerator 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

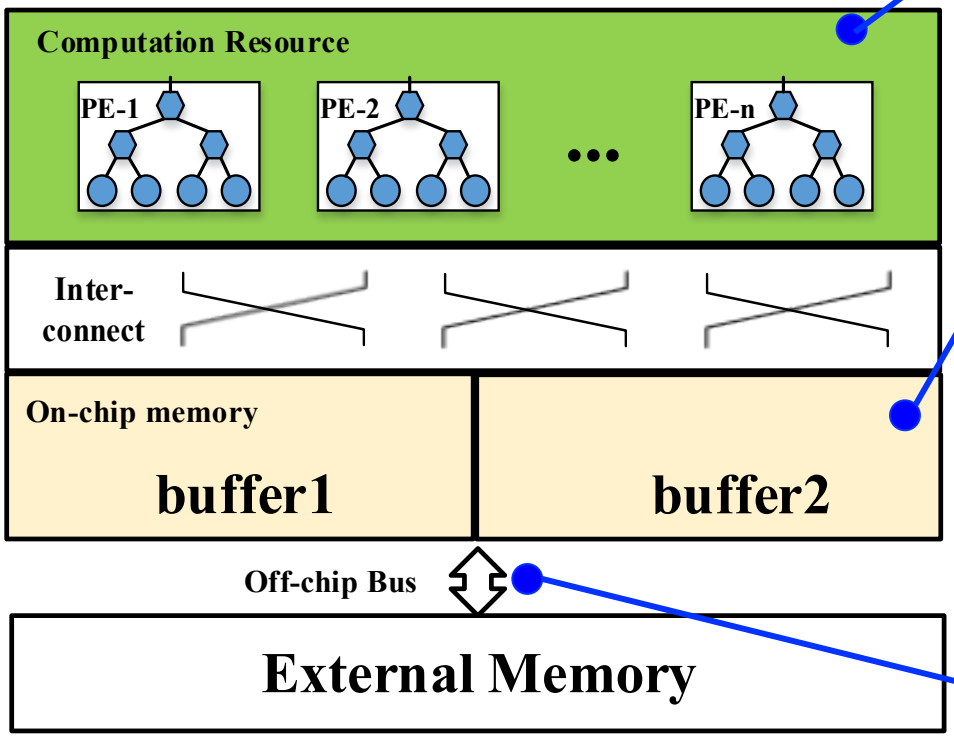
Performance Modeling

- 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 constraints for now

Performance Modeling

- Total operations FLOPS $\approx 2\times O\times I\times R\times C\times K^2$
- 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: $Bw = Ti\times To\times K^2$
  - External access times: $ao = \left\lceil\frac{O}{To}\right\rceil\times\left\lceil\frac{R}{Tr}\right\rceil\times\left\lceil\frac{C}{Tc}\right\rceil$, $ai = aw = \left\lceil\frac{I}{Ti}\right\rceil\times ao$
    - Input features and weights are reused across multiple output tiles

Performance Modeling using Roofline



Computational Throughput

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

(OPs / Sec)

Operational Intensity* (OI)

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

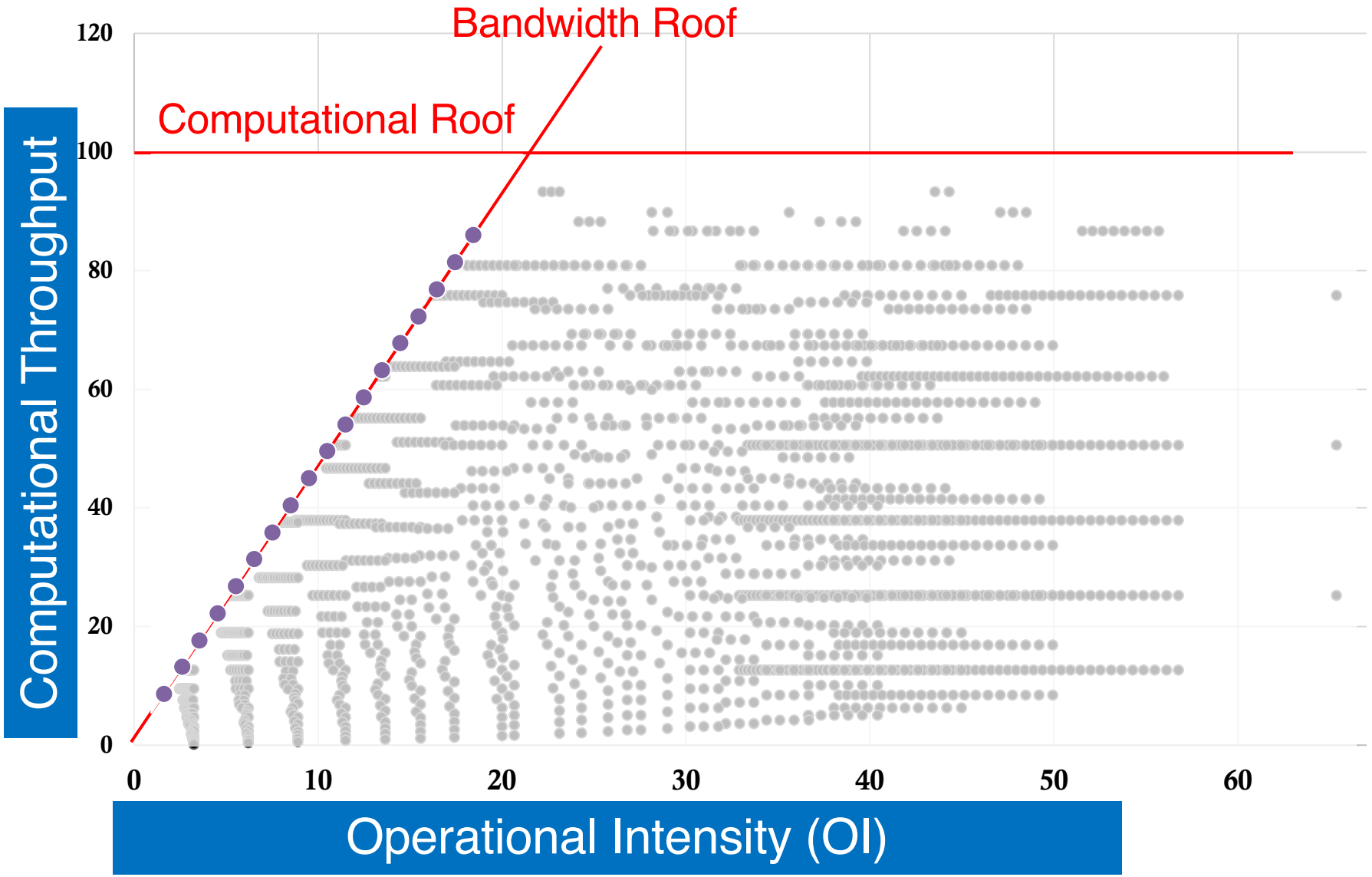
(OPs / Byte Accessed)

Memory Bandwidth

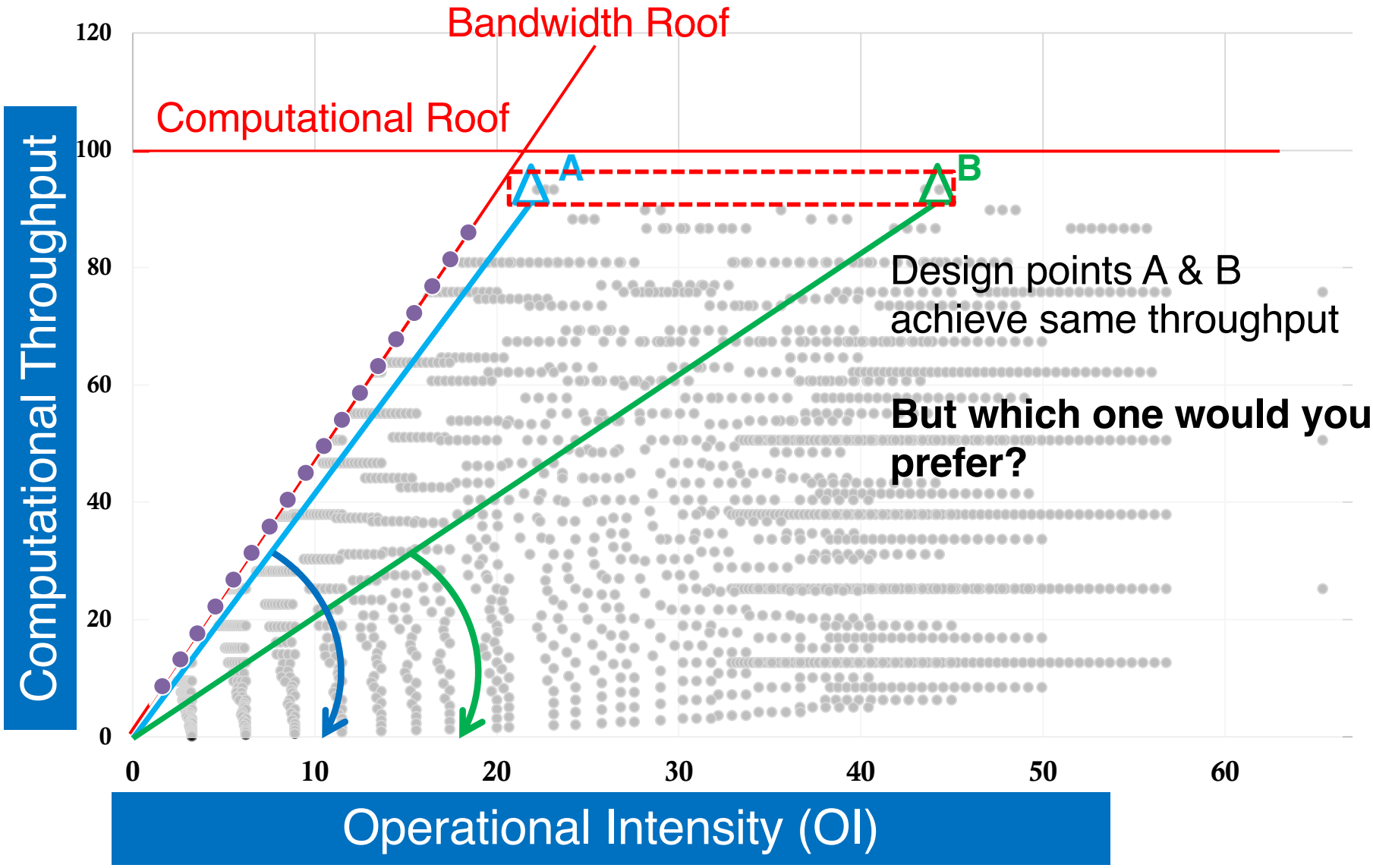
(Bytes / Sec)

* OI is also known as computation to communication ratio (CTC) or arithmetic intensity (AI)

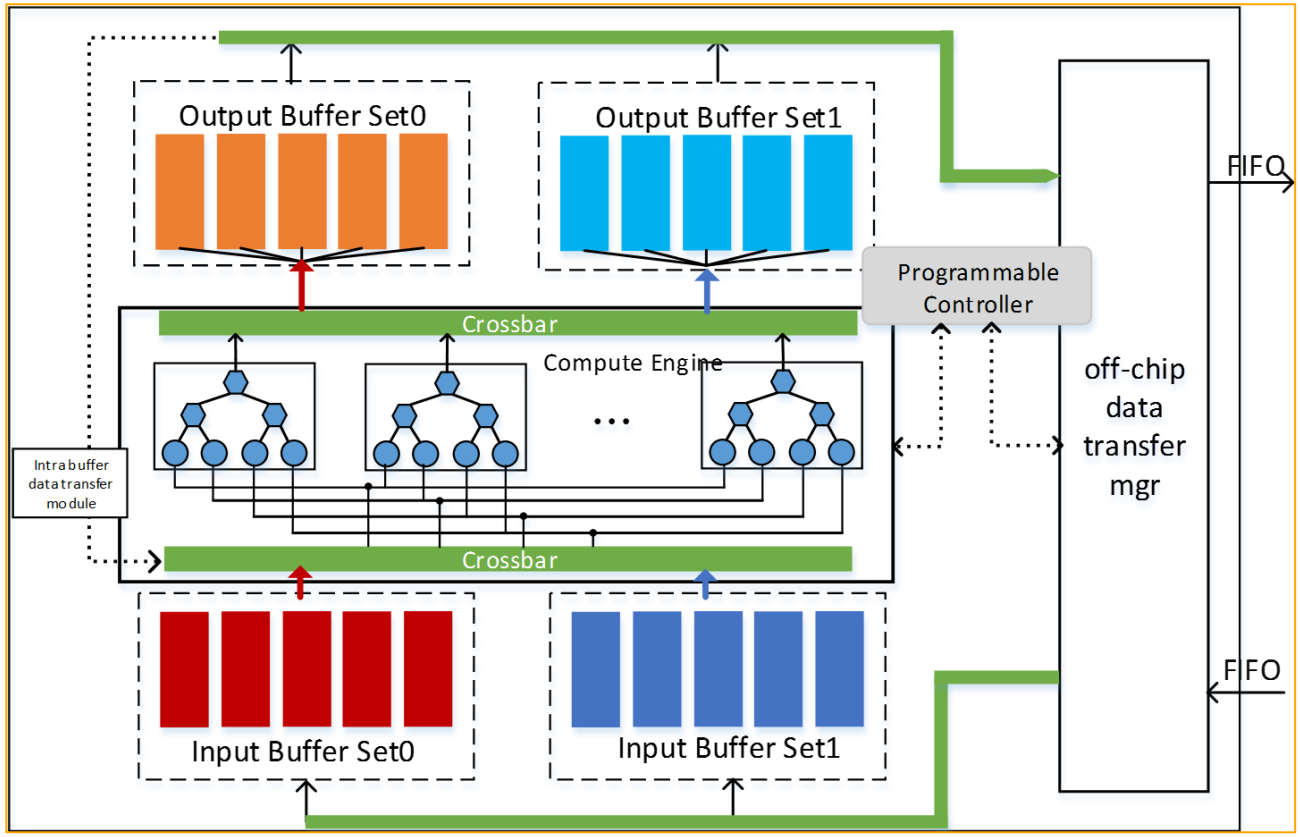
Design Space Exploration with Roofline



Design Space Exploration with Roofline



Hardware Implementation



- All values in floating-point
- Only handles Conv layers, not dense or pooling
- Target FPGA: Virtex7-485t (28nm)

FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations

Yichi Zhang[1], Junhao Pan[2], Xinheng Liu[2], Hongzheng Chen[1], Deming Chen[2], Zhiru Zhang[1]

[1]Electrical and Computer Engineering, Cornell
[2]Electrical & Computer Engineering, UIUC

FPGA'21, Feb/Mar 2021

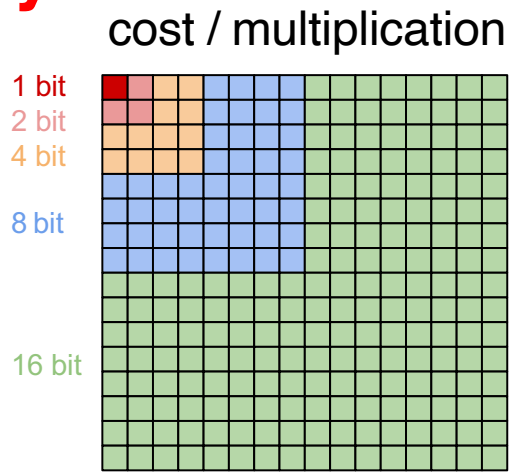
CNNs with Reduced Numerical Precision

- Hardware architects widely quantization to improve the efficiency of CNN execution
  - **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?

Quantization Improves Efficiency

Reducing bitwidth by a factor of 2

- **1/4** of multiplication power cost
- **1/2** of addition power cost
- **1/2** of SRAM area/power cost

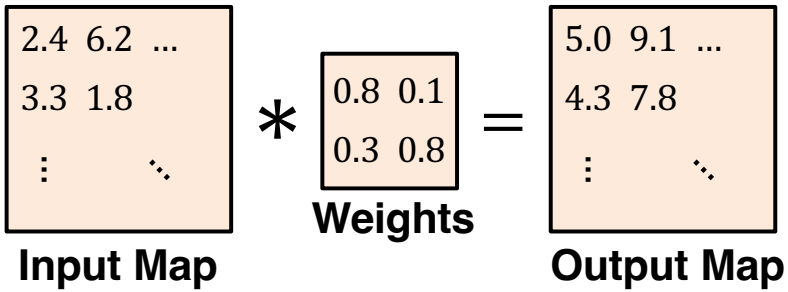


	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32	FP32	15.7	1x	-	-
	FP16	FP32	125	8x	-	-
A100	FP32	FP32	19.5	1x	-	-
	TF32	FP32	156	8x	312	16x
	FP16	FP32	312	16x	624	32x
	BF16	FP32	312	16x	624	32x
	FP16	FP16	312	16x	624	32x
	INT8	INT32	624	32x	1248	64x
	INT4	INT32	1248	64x	2496	128x
	BINARY	INT32	4992	256x	-	-
	IEEE FP64		19.5	1x	-	-

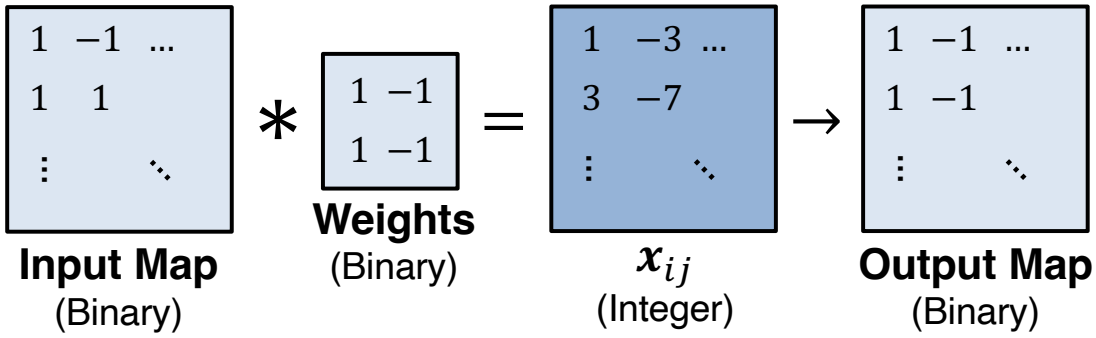
Reference: NVIDIA Ampere Architecture Whitepaper Table 3

Binarization: Pushing Quantization to the Extreme

CNN



BNN



Key Differences

1. Inputs are binarized (–1 or +1)
2. Weights are binarized (–1 or +1)
3. MAC becomes XNOR+Popcount

BNNs are well suited for FPGAs (rich in LUTs)

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

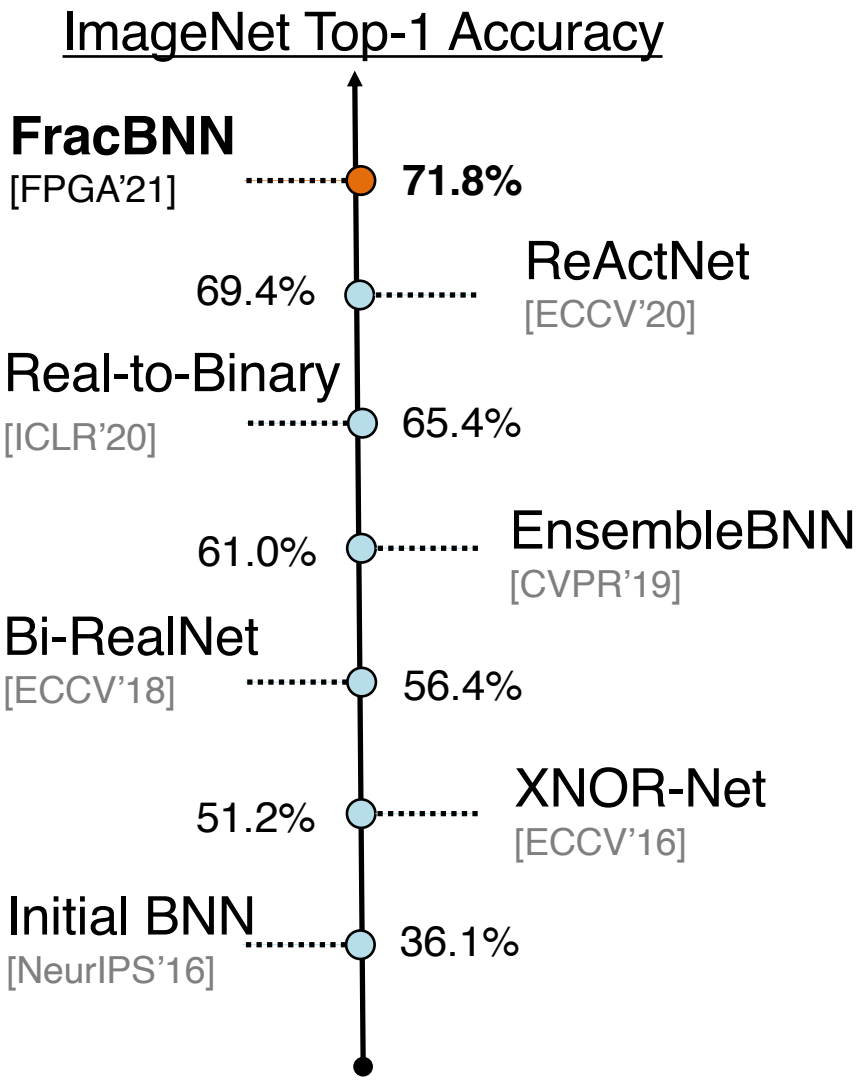
b_1	b_2	b_1 XNOR b_2
0	0	1
0	1	0
1	0	0
1	1	1

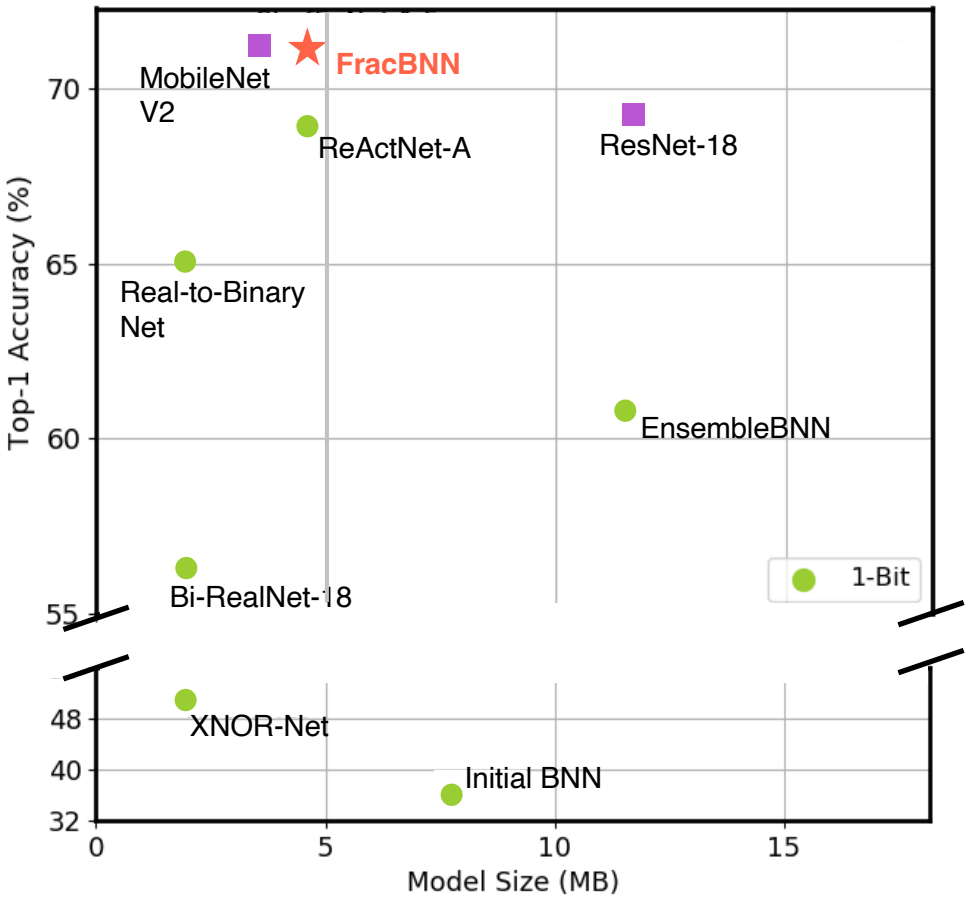
- Encode {–1, +1} as {0,1} → multiplies become XNORs
- Conv/dense layers do dot products → XNOR and popcount
- Operations can map to LUT fabric as opposed to DSPs

2. Binarized weights may reduce total model size

- But note that fewer bits per weight may be offset by having more weights

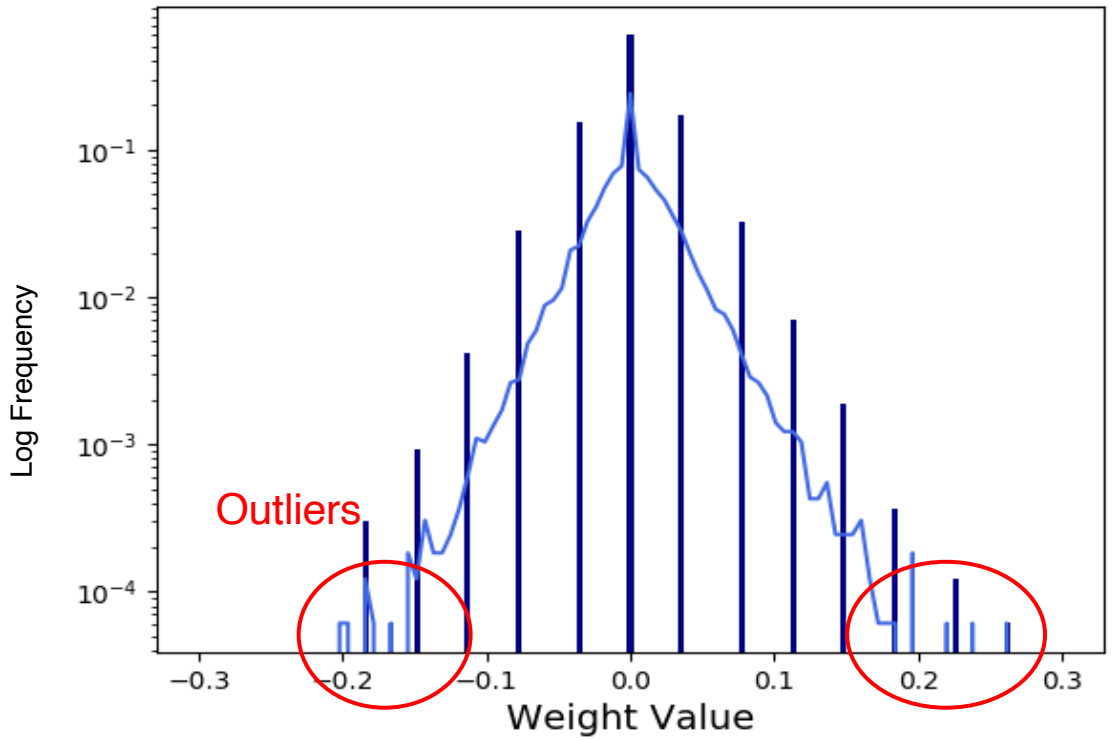
Evolution of BNN Accuracy



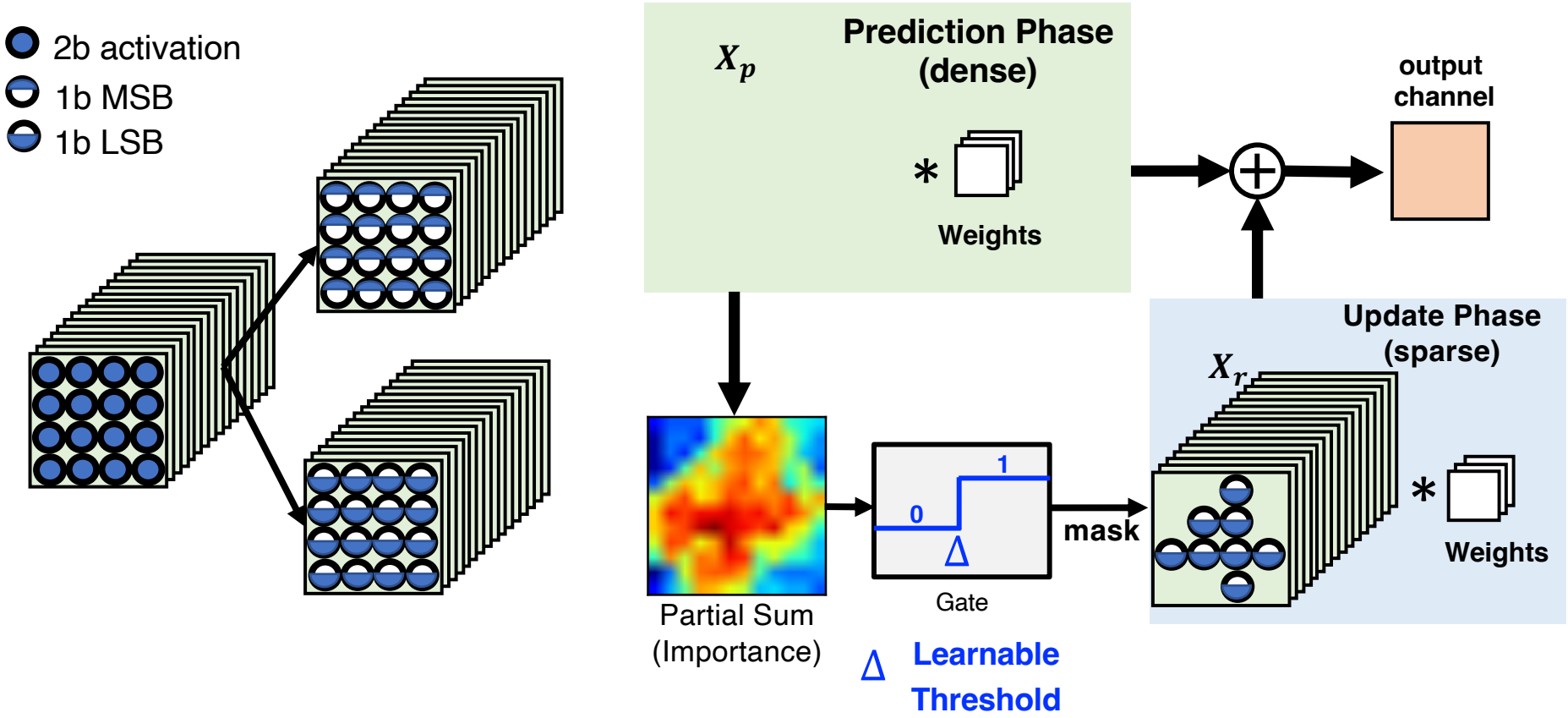


The Outlier Problem in Quantization

- DNN weights and activations are typically distributed in a bell curve
  - Most values are close to zero, but rare *outliers* are large
  - Outliers stretch the quantization grid, resulting in poor resolution

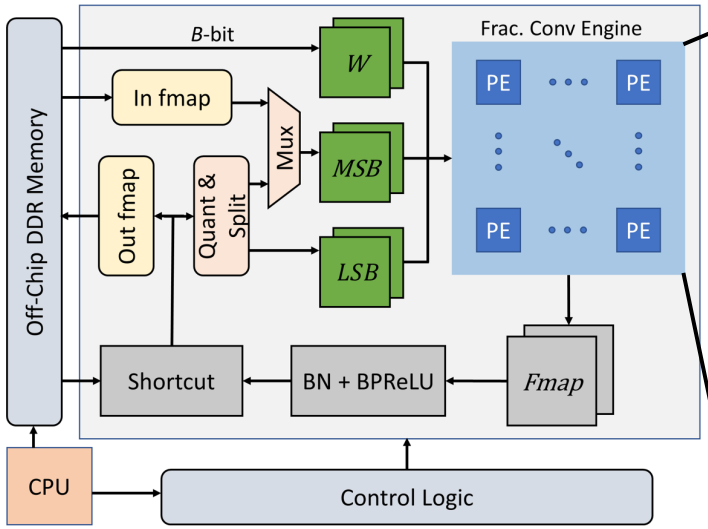


Fractional Activation: Dual-Precision Scheme



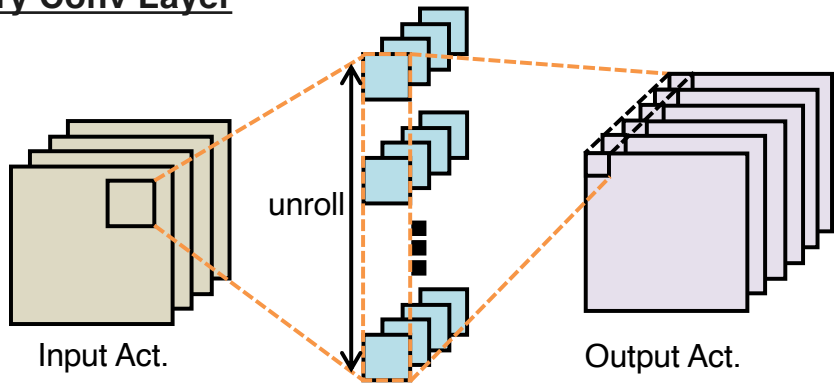
Exploits input-dependent characteristics
Avoids overly aggressive quantization that degrades accuracy

FracBNN Accelerator Architecture on FPGA

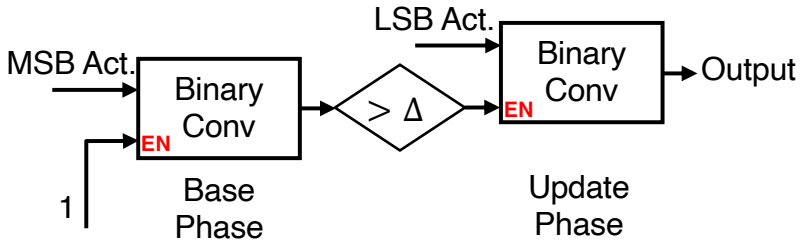


Board: Xilinx Ultra96 v2

Binary Conv Layer



Fractional Conv



All layers share the same conv engine using binary MACs (Implemented in HLS)

FracBNN Hardware Evaluation

Evaluation on CIFAR-10

	Device	Bits (W/A)	Freq. (MHz)	Top-1 (%)	Frame Rate
ReBNet	Zynq ZC702	1/1	200	86.98	2000
Initial BNN	Zynq 7Z020	1/1	143	88.8	168.4
FracBNN	Zynq ZU3EG	1/1.4	**250**	**89.1**	**2806.9**

Evaluation on ImageNet

	Device	Bits (W/A)	Freq. (MHz)	Top-1/Top-5 (%)	Frame Rate
FINN-R	Zynq ZU3EG	1/2	220	50.3/-	200
Synetgy	Zynq ZU3EG	4/4	**250**	68.3/88.1	41.1
FracBNN	Zynq ZU3EG	1/1.4	**250**	**71.8/90.1**	48.1

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

Acknowledgements

- This tutorial contains/adapts materials developed by
  - Ritchie Zhao (Cornell ECE PhD, now Microsoft)
  - Yichi Zhang (PhD Student, Cornell ECE)
  - Authors of the following papers
    - Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks (FPGA'15, PKU-UCLA)
    - FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations (FPGA'21, Cornell-UIUC)