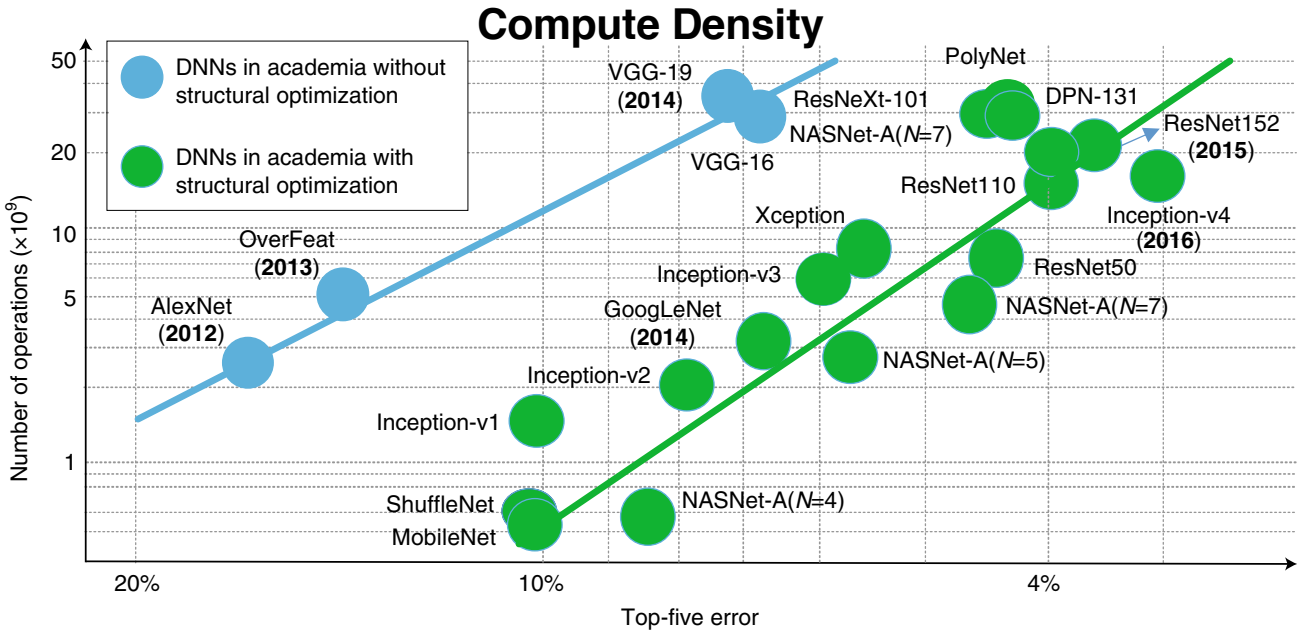
ECE 5997
Hardware Accelerator Design & Automation
Fall 2021

DNN Acceleration on FPGAs

Cornell University

CSL

Modern DNNs 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.

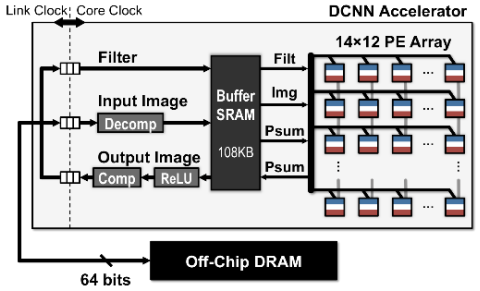
- **DNNs require enormous amount of compute**
  - For example, ResNet50 (70 layers) performs 7.7 billion operations required to classify one image

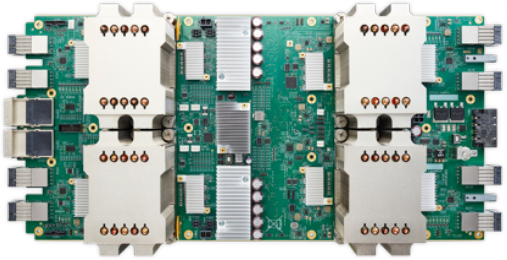
DNNs are Moving to Dedicated Hardware

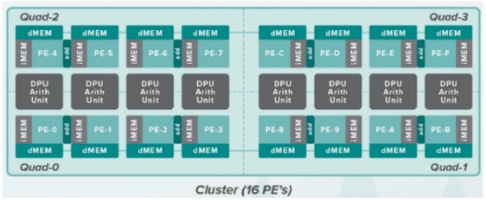
Blue Chips
Apple
Google
Intel
Microsoft
…

Startups
Cambricon
Cerebras
Deephi (→ Xilinx)
Graphcore
…

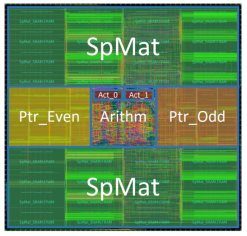
Academia
EIE/ESE [Han ISCA'16, FPGA'17]
Eyeriss [Chen ISCA'16]
FINN [Umuroglu FPGA'17]
Minerva [Reagen ISCA'16]
…







CGRA Cluster of 16 PEs with 8 Arithmetic Units and 16KB data RAM



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]

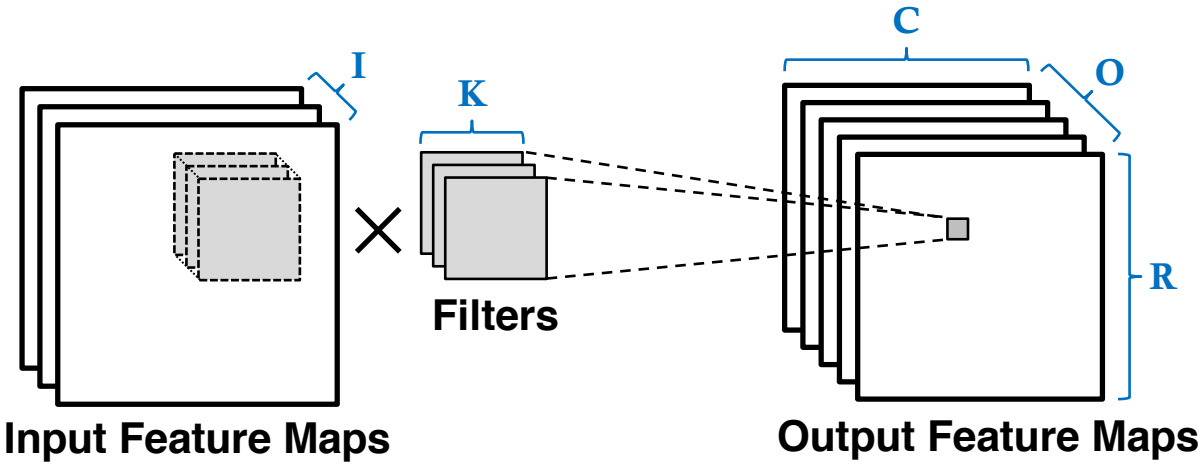
[1]Center for Energy-Efficient Computing and Applications, Peking University
[2]Computer Science Department, University of California, Los Angeles
[3]PKU/UCLA Joint Research Institute in Science and Engineering

FPGA'15, Feb 2015

Main Contributions

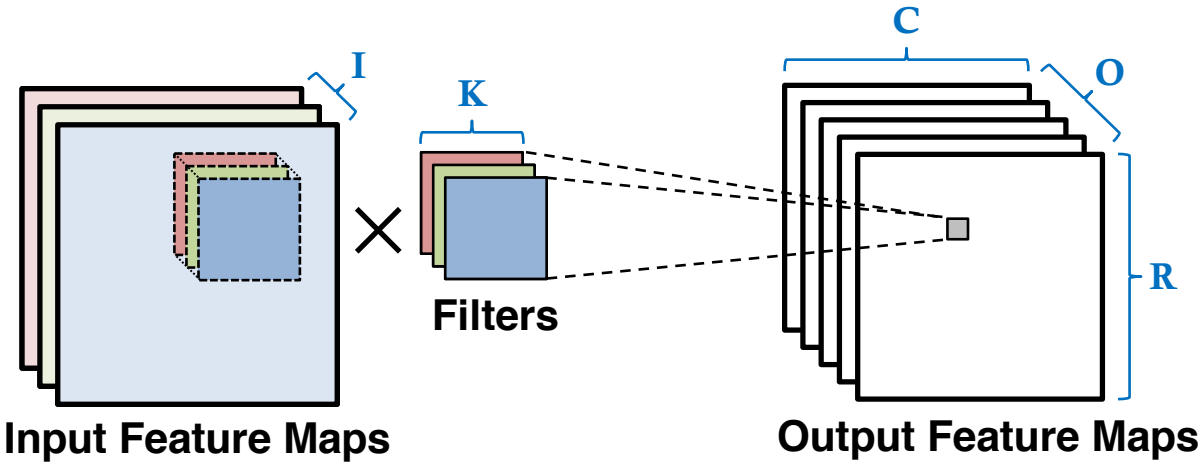
1. Analysis of the different sources of parallelism in the convolution kernel of a CNN
2. Quantitative performance modeling of the hardware design space using the Roofline method
3. Design and implementation of a CNN accelerator for FPGA using Vivado HLS, evaluated on AlexNet

Convolutional Layer



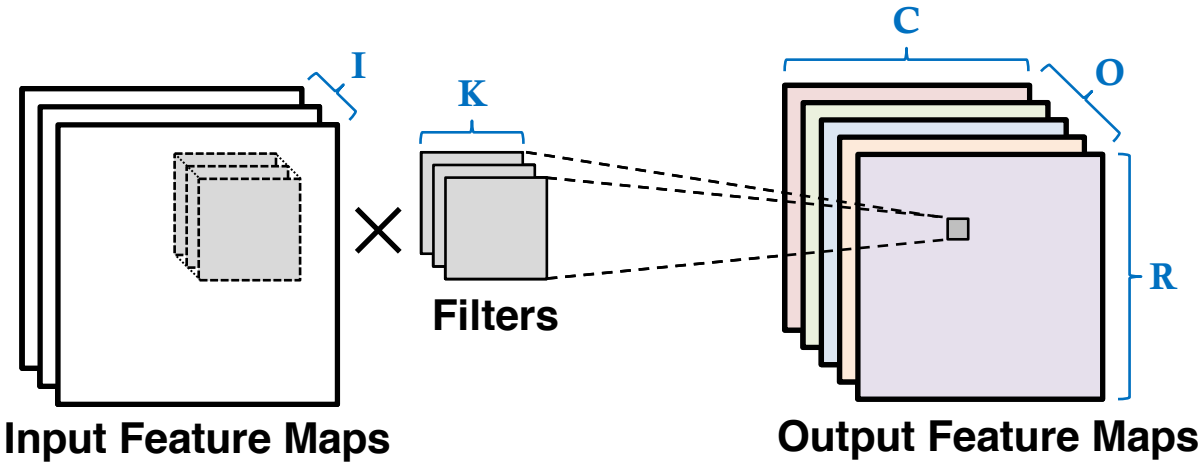
- An output pixel is connected to its neighboring region on each input feature map
- All pixels on an output feature map use the same filter weights

Parallelism in the Convolutional Layer



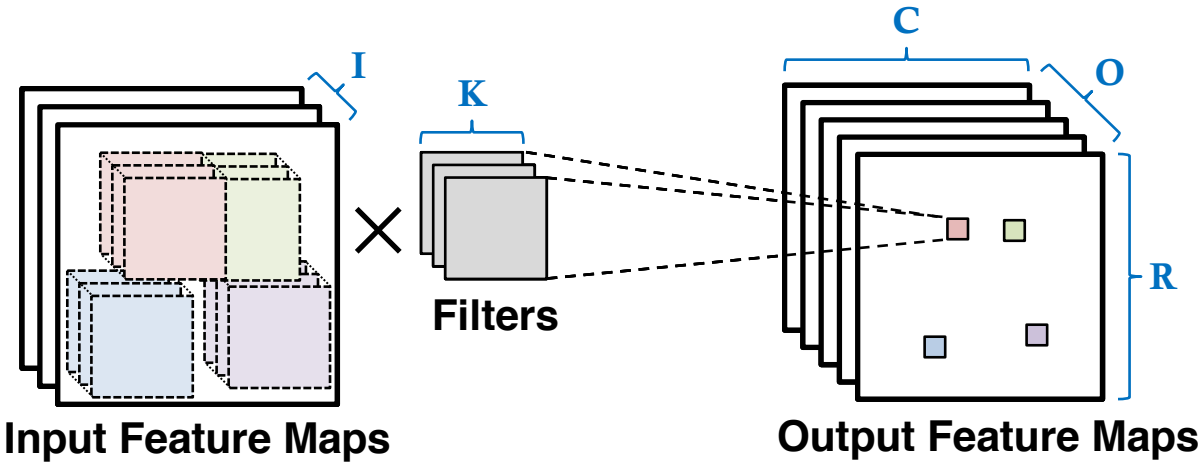
- Four main sources of parallelism
  1. **Across input feature maps**

Parallelism in the Convolutional Layer



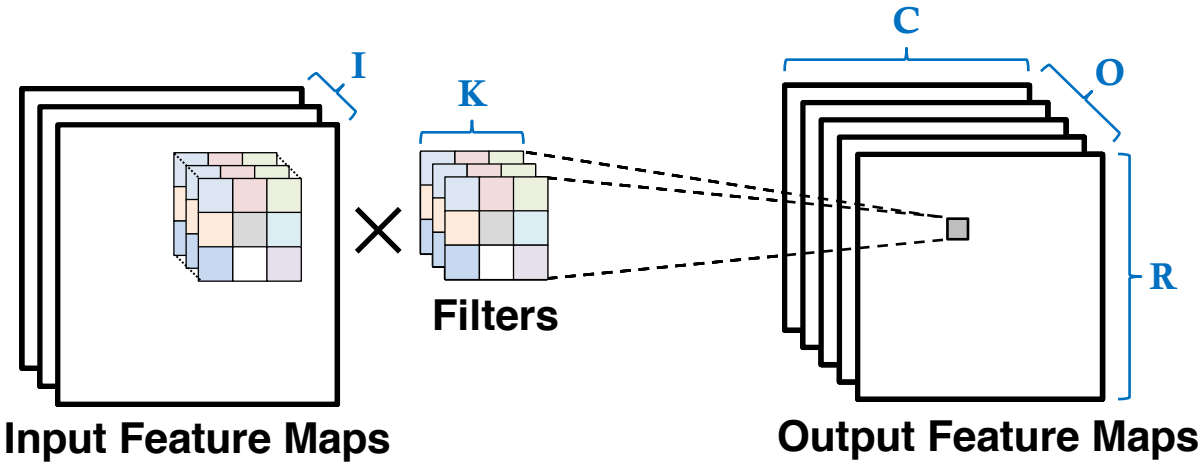
- Four main sources of parallelism
  1. Across input feature maps
  2. **Across output feature maps**

Parallelism in the Convolutional Layer



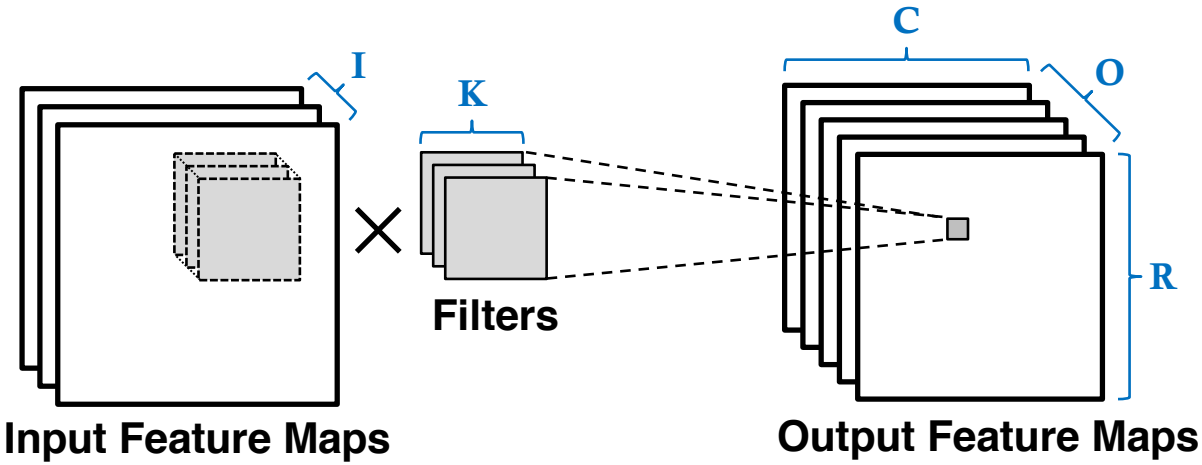
- Four main sources of parallelism
  1. Across input feature maps
  2. Across output feature maps
  3. **Across different output pixels (i.e. filter positions)**

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

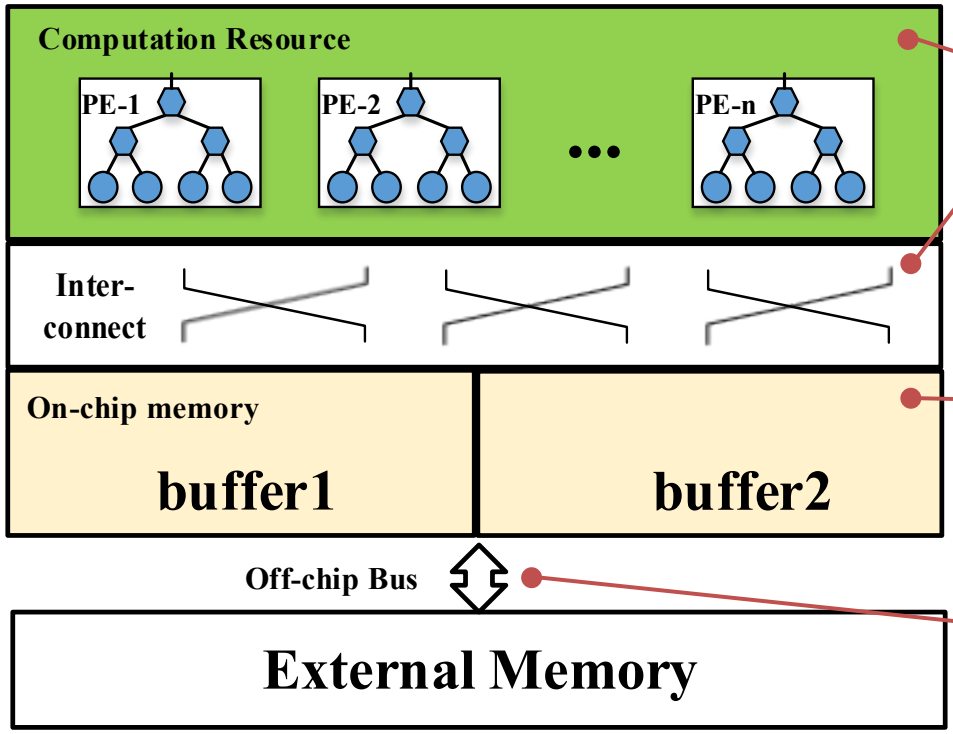
Loop over input feature maps

Loop over filter pixels

S is the stride

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

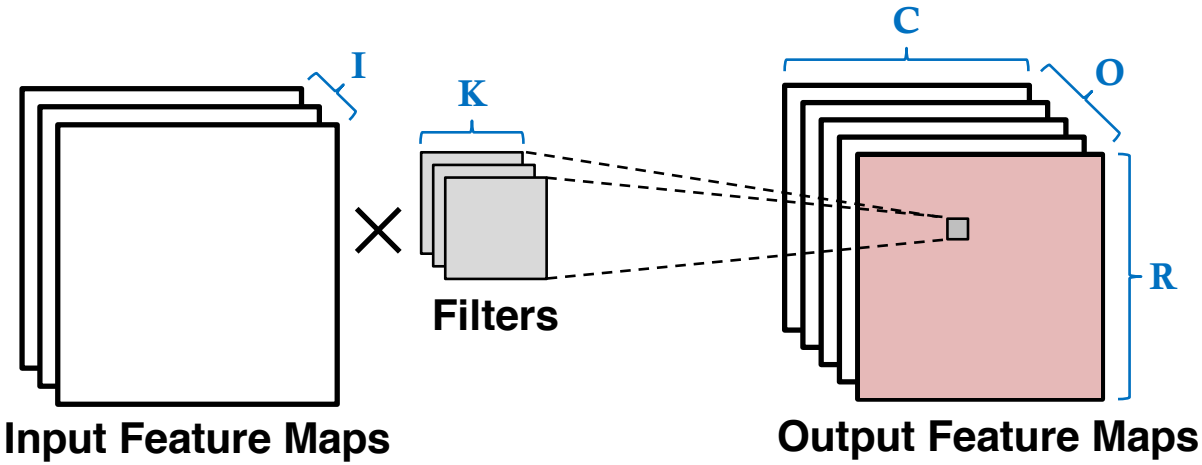
Limited on-chip storage

Technique: loop tiling

Limited off-chip bandwidth

Technique: data reuse, compute-memory balance

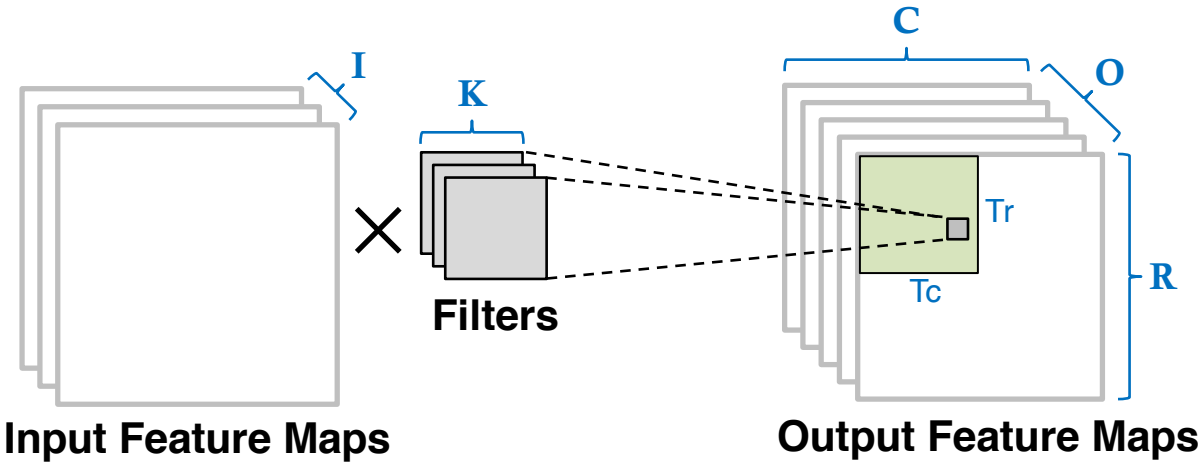
Loop Tiling




```
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(col+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) {
           // 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(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

Maximize memory reuse

FPGA Portion

Maximize computational performance

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];
            }}}}}}
```

Number of cycles to execute the above loop nest

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

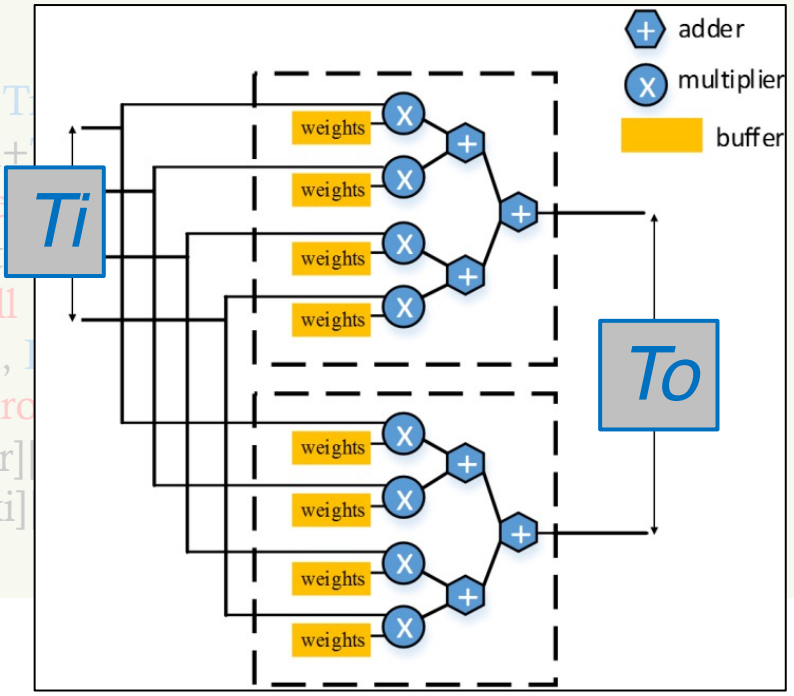
L is the pipeline depth (# of pipeline stages, II=1)

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

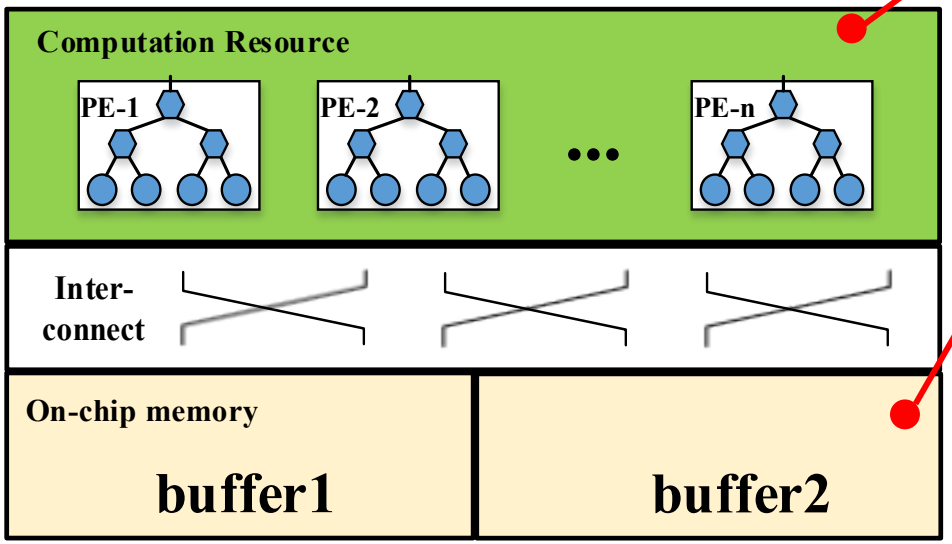
Performance Modeling

- 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 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: $Bw = To\times Tr\times K^2$
  - 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)

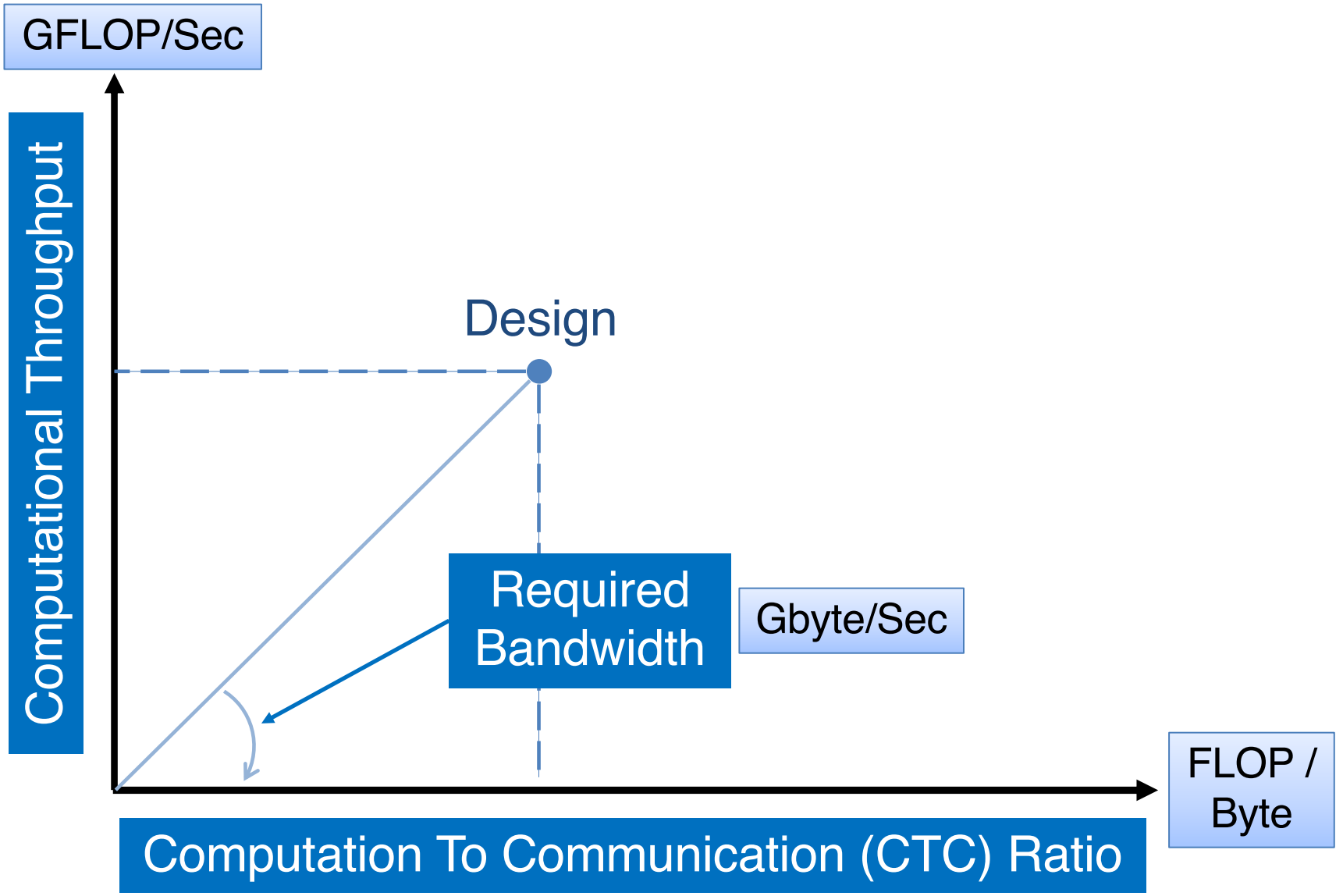
Required Bandwidth

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

GByte/Sec

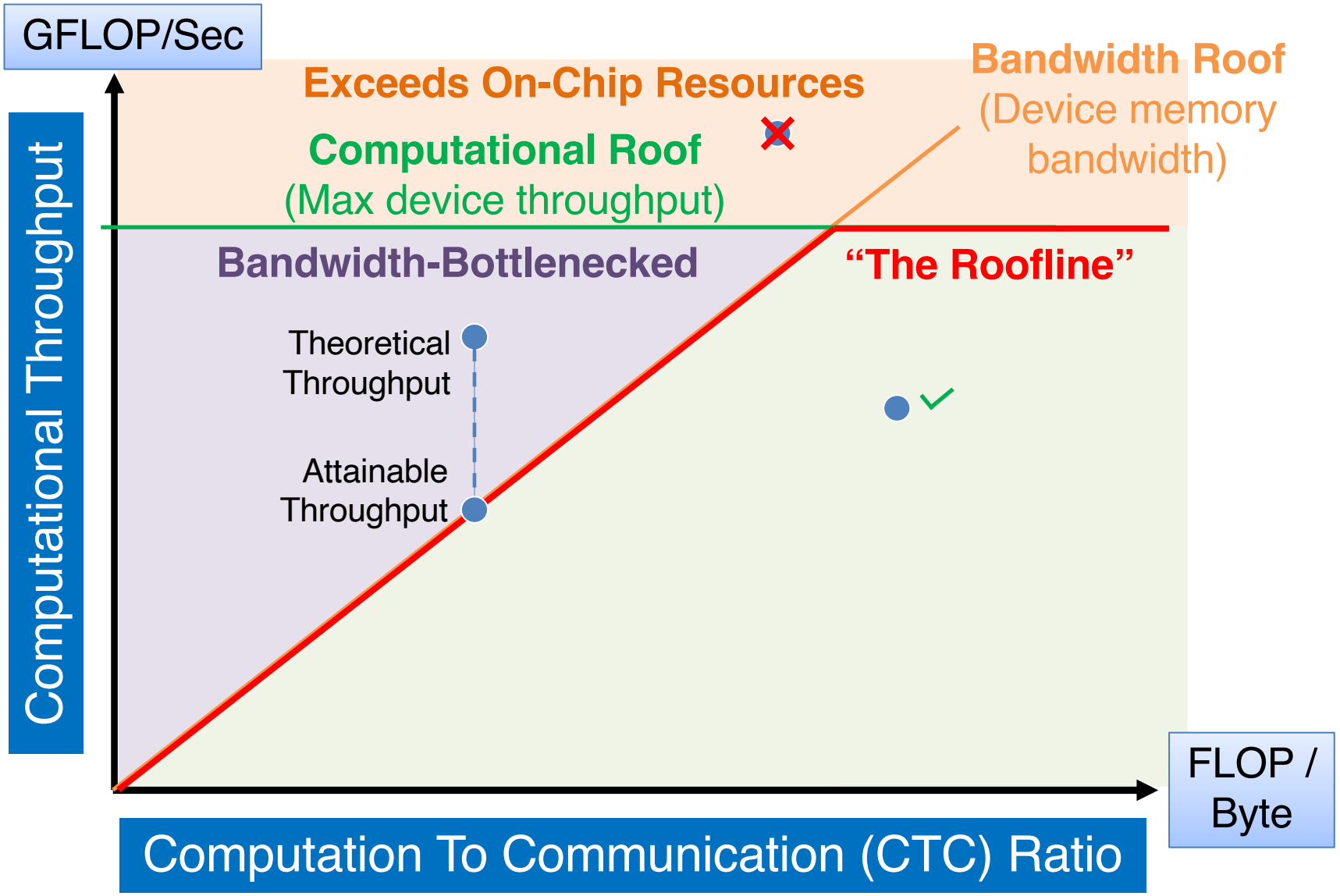
$$\frac{GByte}{Sec} = \frac{GFLOP/Sec}{FLOP/(Byte\ Accessed)}$$

Roofline Method [1]



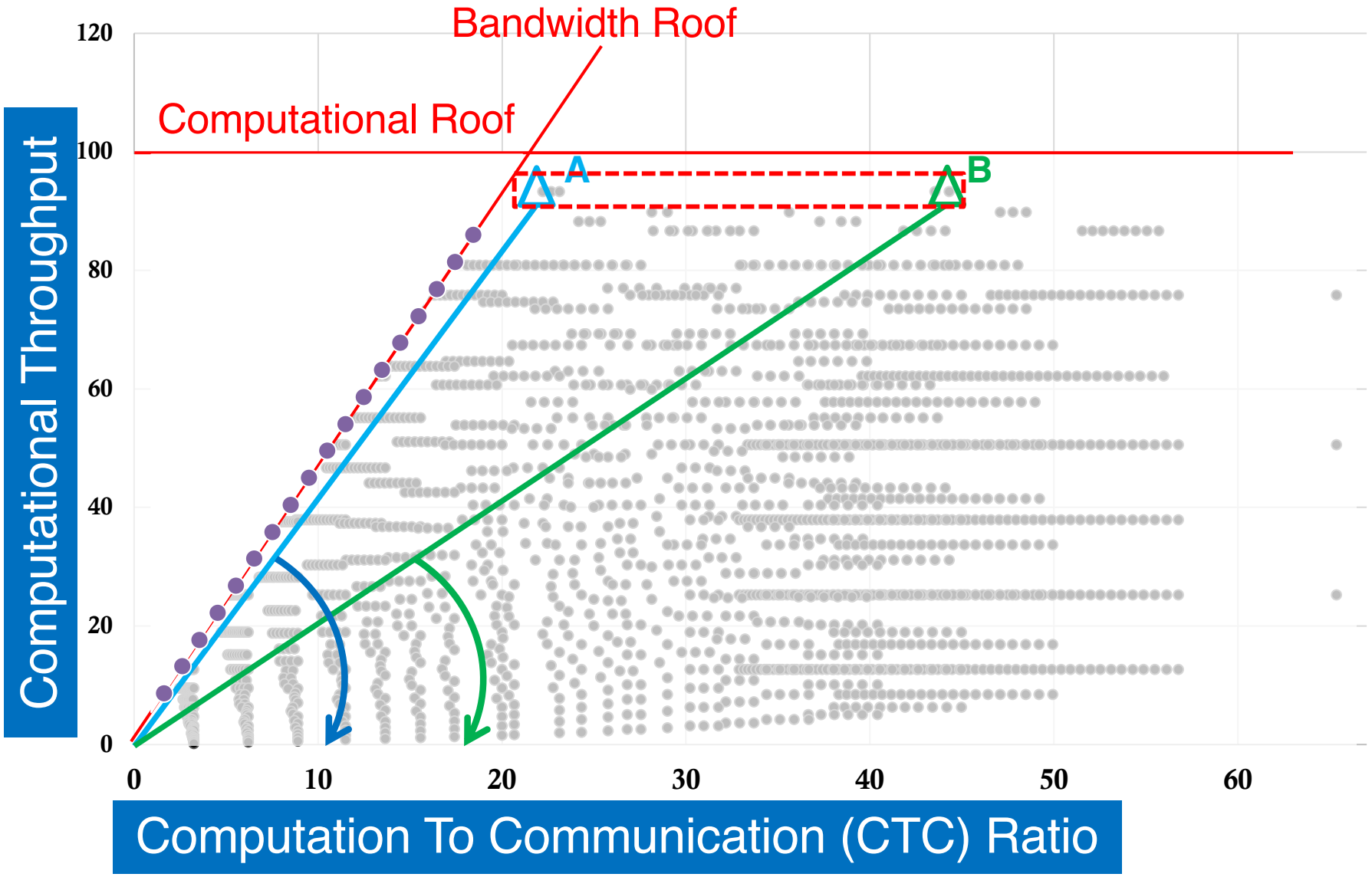
[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

Roofline Method [1]

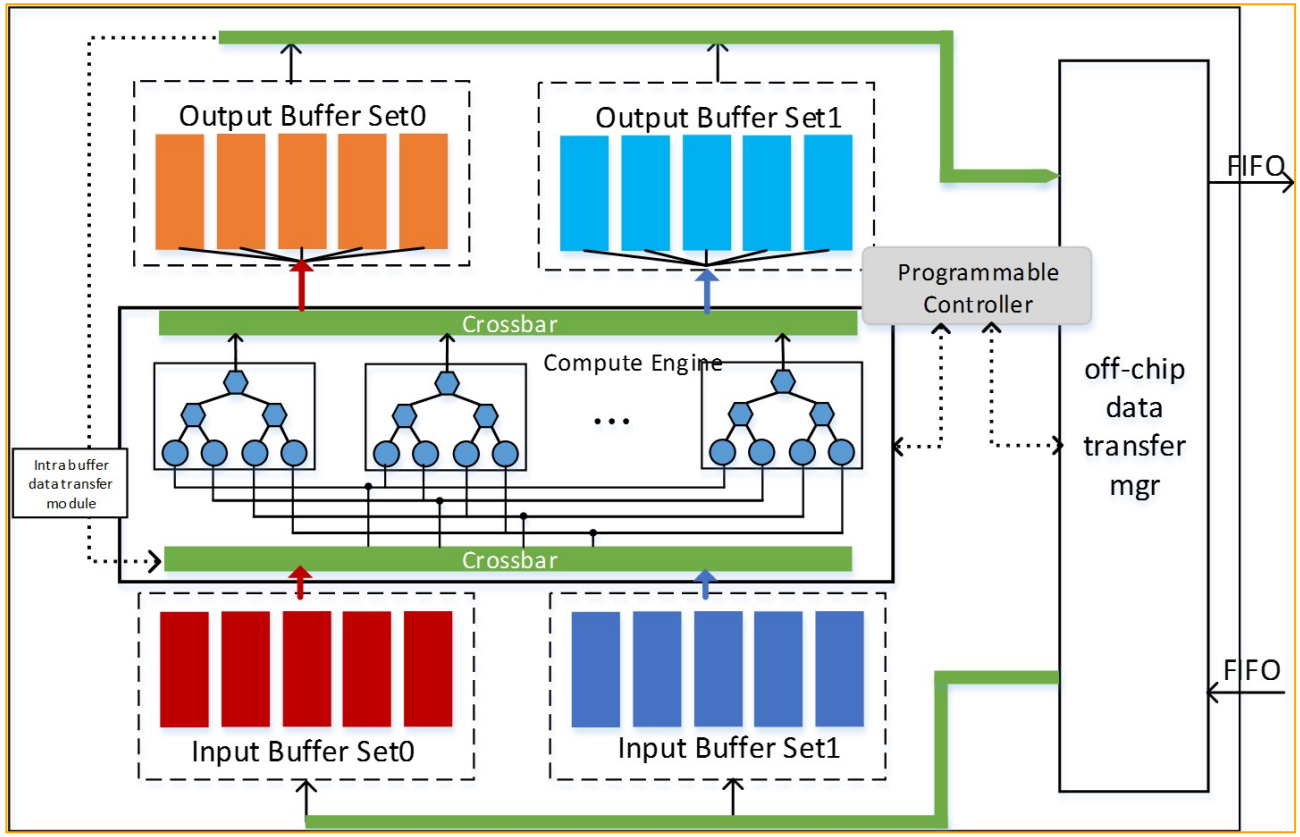


[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

Design Space Exploration with Roofline

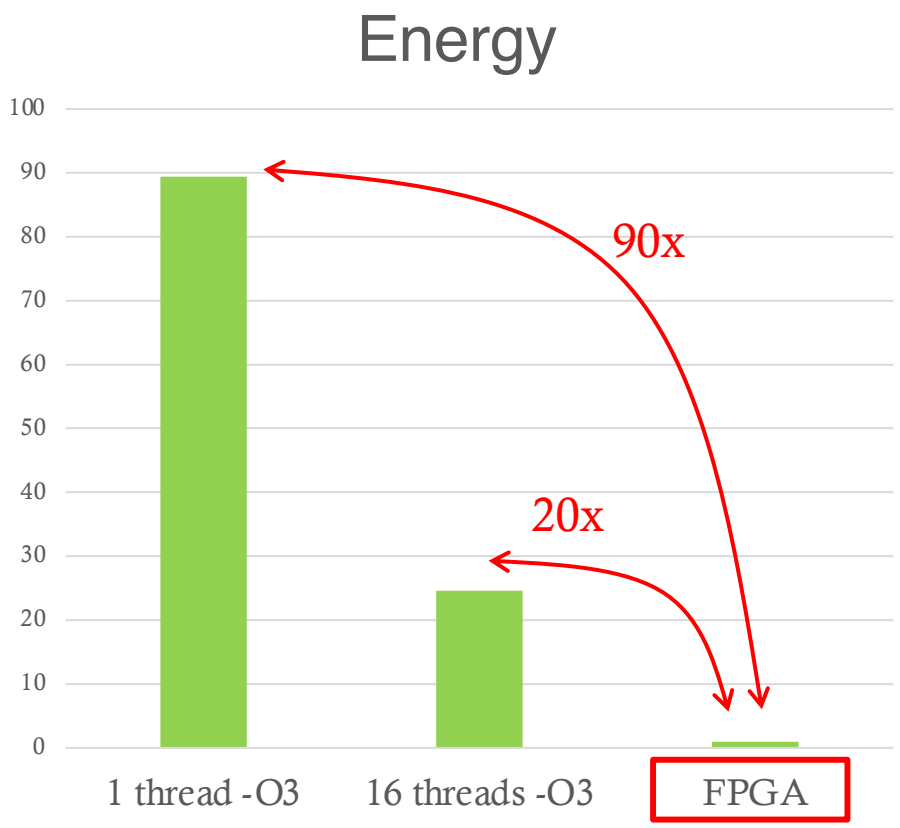


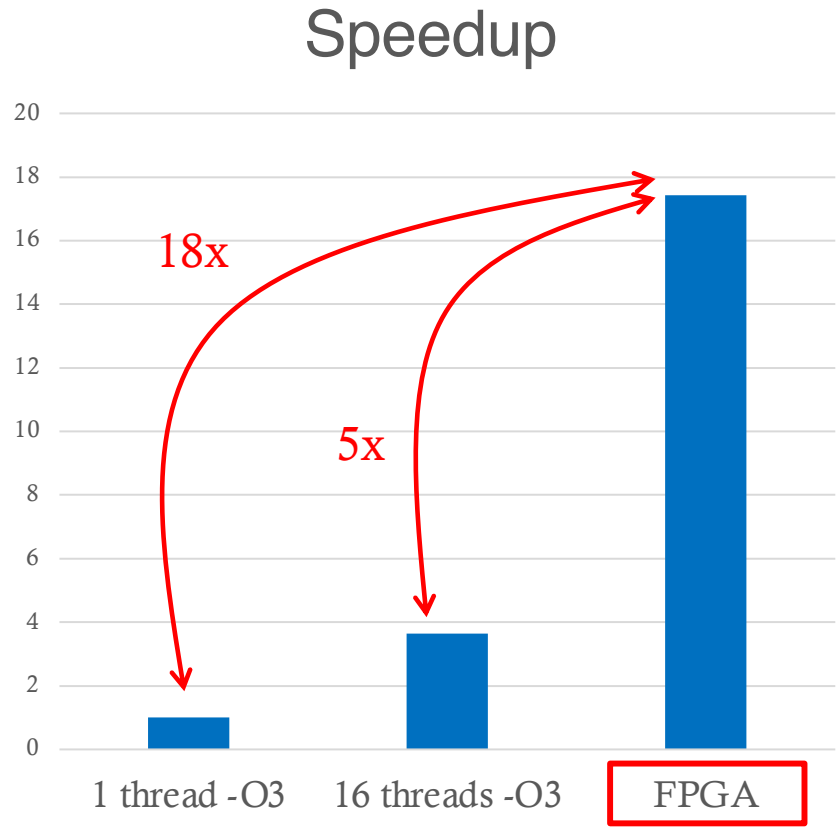
Hardware Implementation



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

Experimental 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

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++

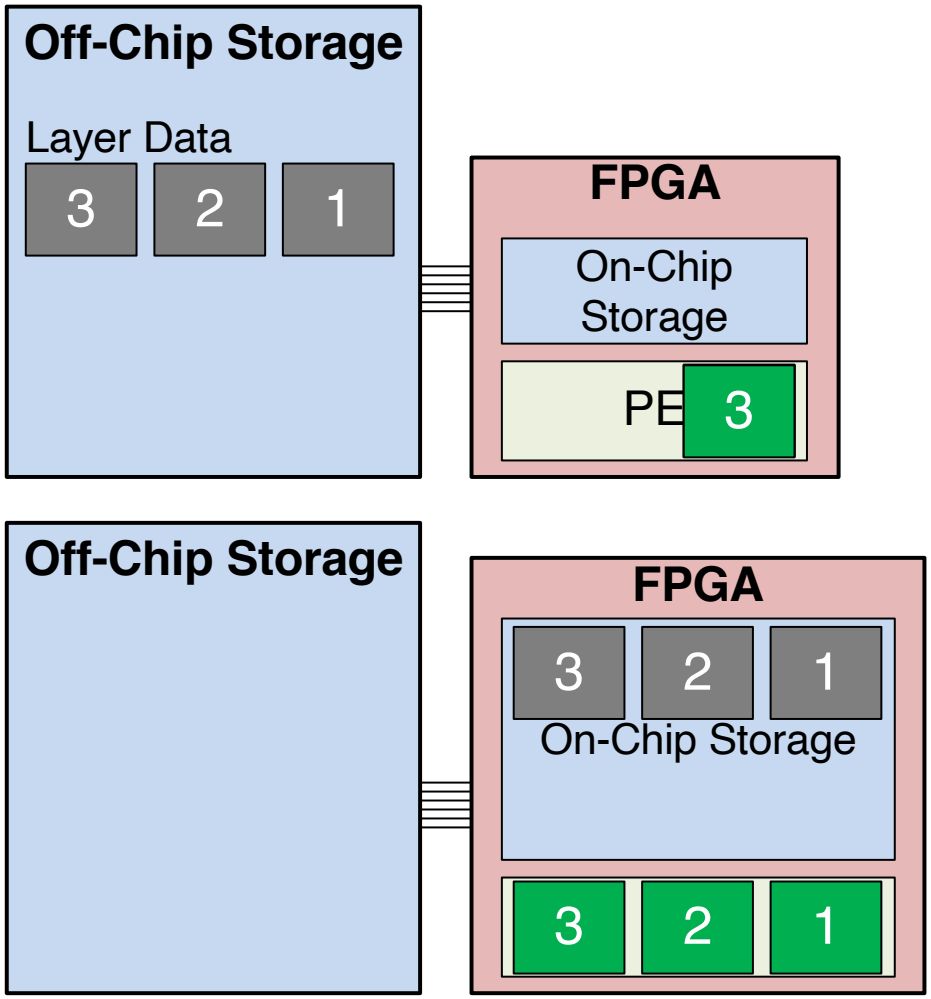
```
kernel
void sum (global float* a,
          global float* b,
          global float* c)
{
    int gid = get_global_id(0);
        c[gid] = a[gid] * b[gid];
}
```

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

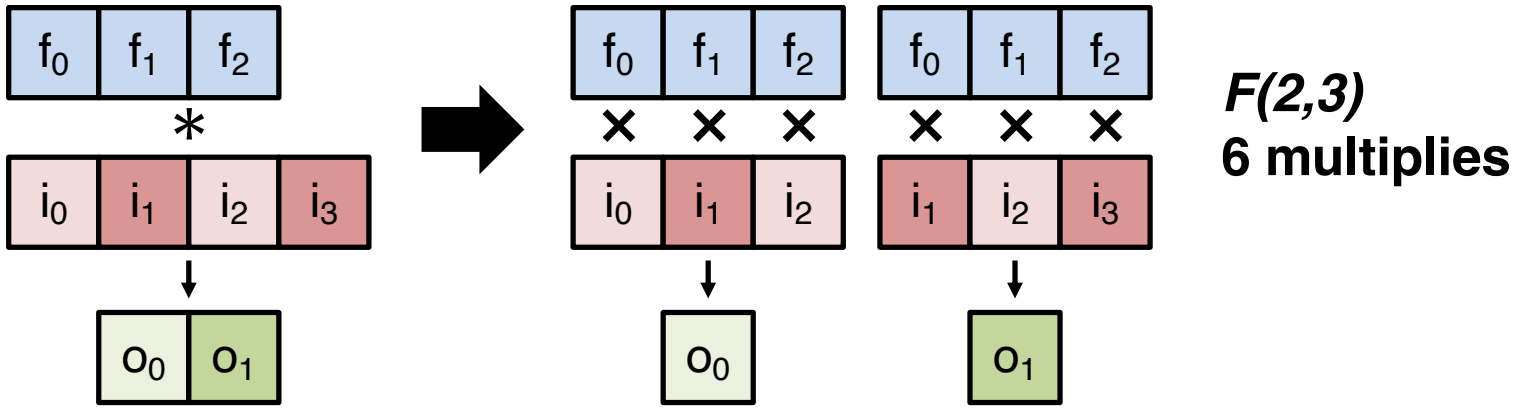


- Previous papers stored layer data off-chip
- Insufficient on-chip storage to hold all data for a layer (input+output)

- 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

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)*

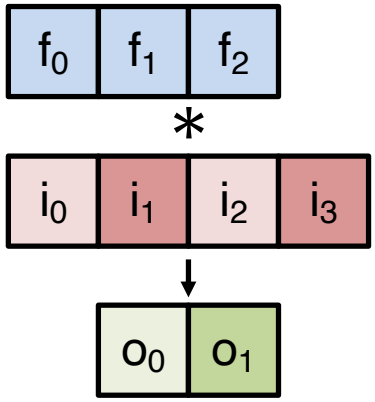


- In the 70s, Shmuel Winograd proved that *F(m,r)* can be computed with a lower bound of only *m+r–1* multiplies [1]

[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

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

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)

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

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

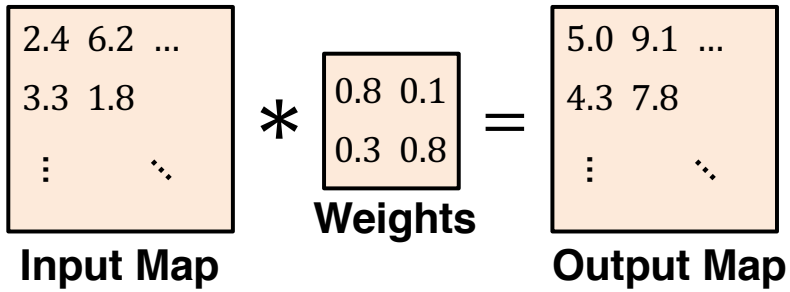
- **ML research papers:**
  - BinaryConnect [NIPS] Dec 2015
  - BNN [arXiv] Mar 2016
  - Ternary-Net [arXiv] May 2016
  - XNOR-Net [ECCV] Oct 2016
  - HWGQ [CVPR] Jul 2017
  - LR-Net [arXiv] Oct 2018
  - Many more!

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

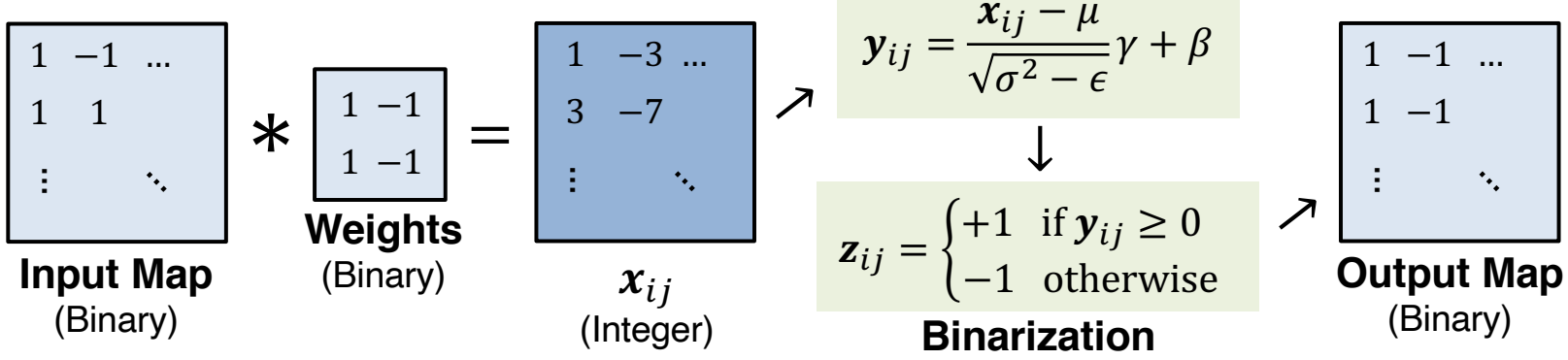
CNN



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

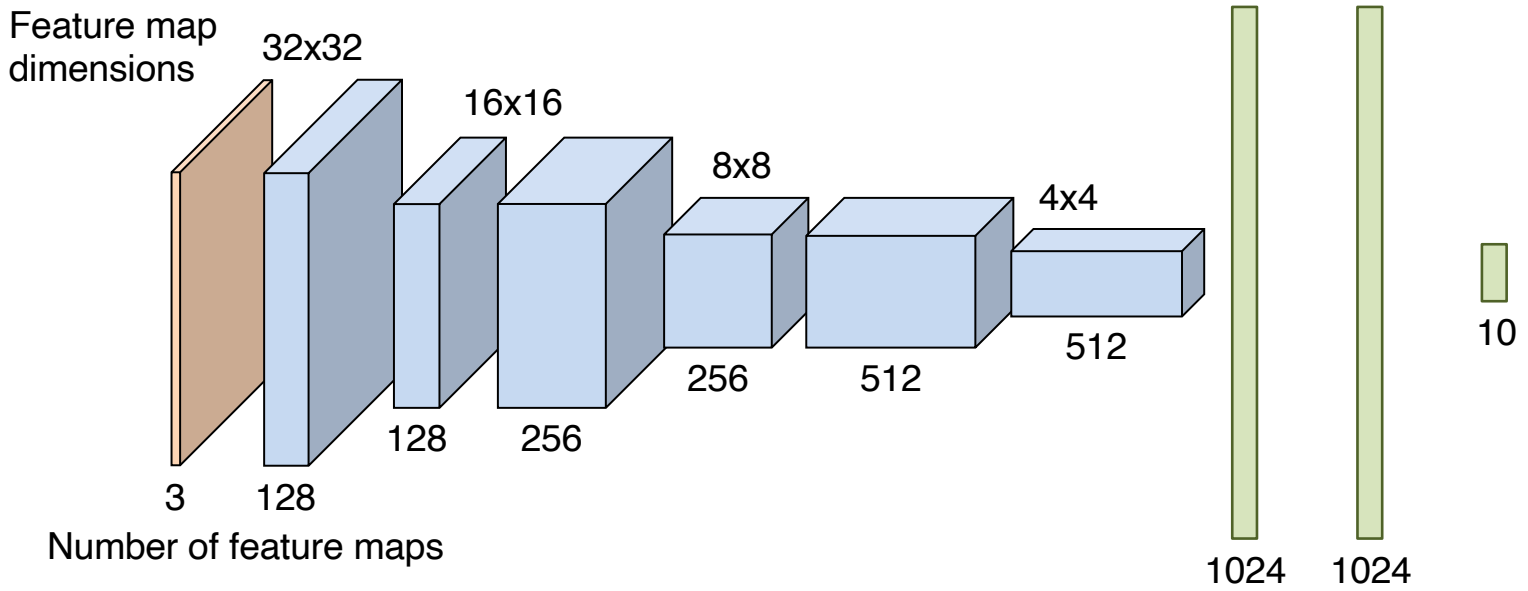
b_1	b_2	b_1 XOR b_2
0	0	0
0	1	1
1	0	1
1	1	0

- 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

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

BNN CIFAR-10 Architecture [2]



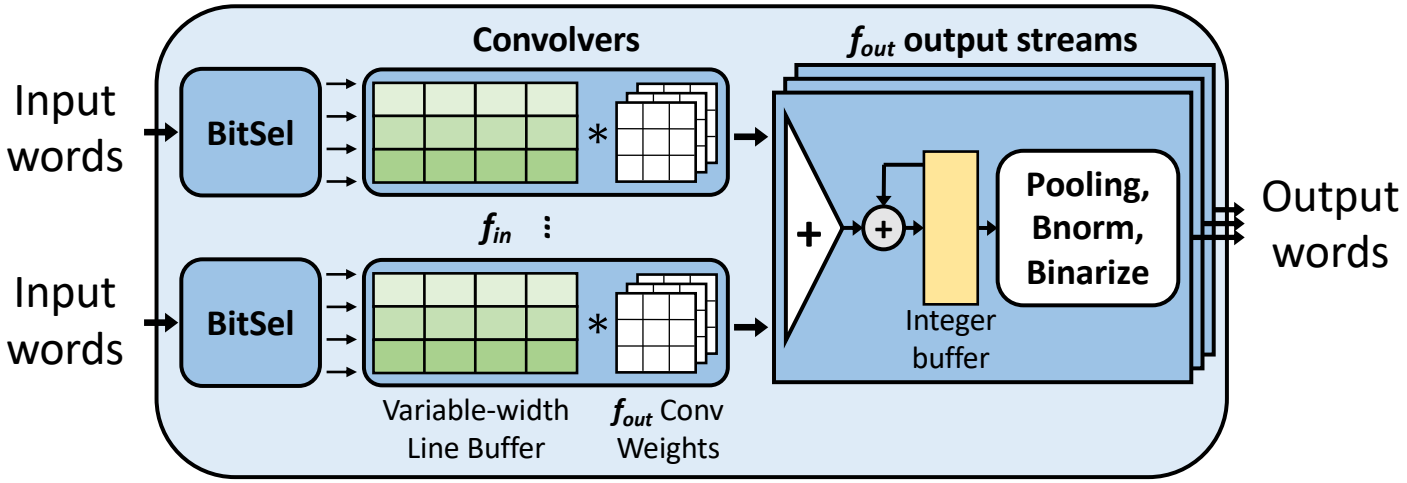
- 6 conv layers, 3 dense layers, 3 max pooling layers
- All conv filters are 3x3
- First conv layer takes in floating-point input
- 13.4 Mbits total model size (after hardware optimizations)

[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.

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

```
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

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.

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)
  - Authors of the following papers
    - Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks (FPGA'15, PKU-UCLA)
    - An OpenCL Deep Learning Accelerator on Arria 10 (FPGA'17, Intel)