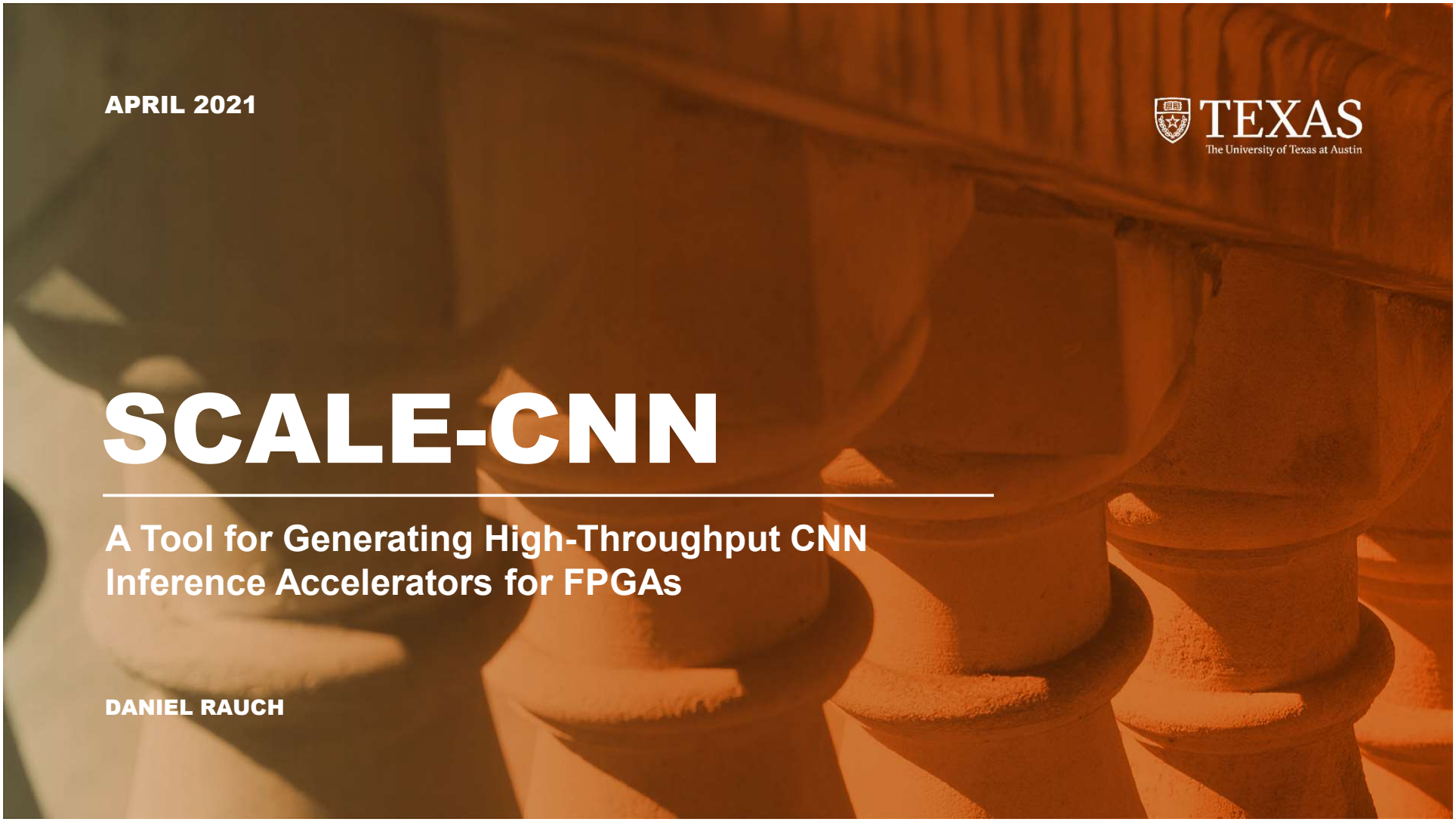


APRIL 2021

TEXAS
The University of Texas at Austin

SCALE-CNN

A Tool for Generating High-Throughput CNN Inference Accelerators for FPGAs

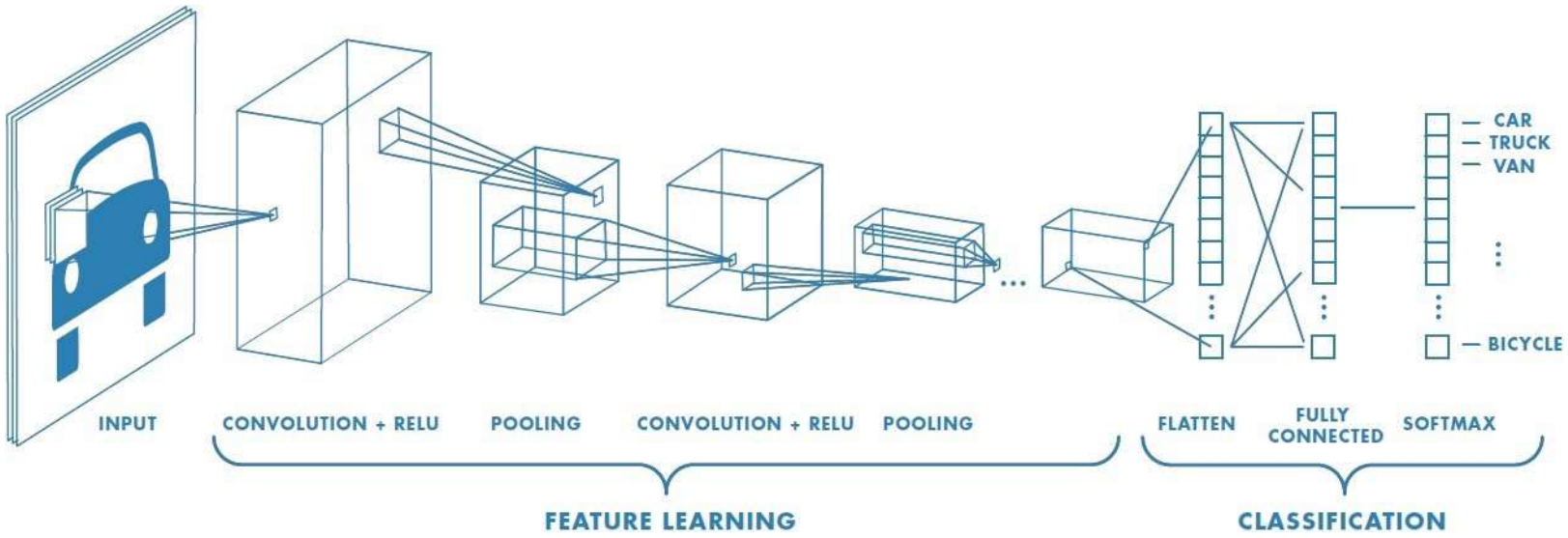
DANIEL RAUCH

Outline

- Background / Motivation
- Scale-CNN Introduction
- HLS
- Layer Design
- Network Design
- Results
- Future Work

Background - CNNs

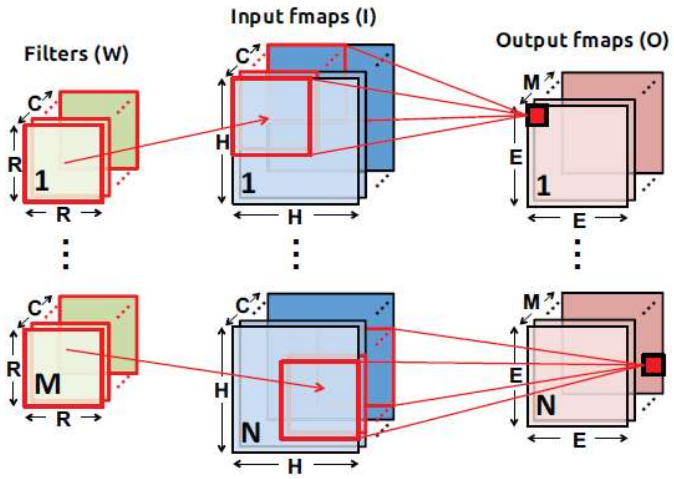
Convolutional Neural Networks (CNNs)



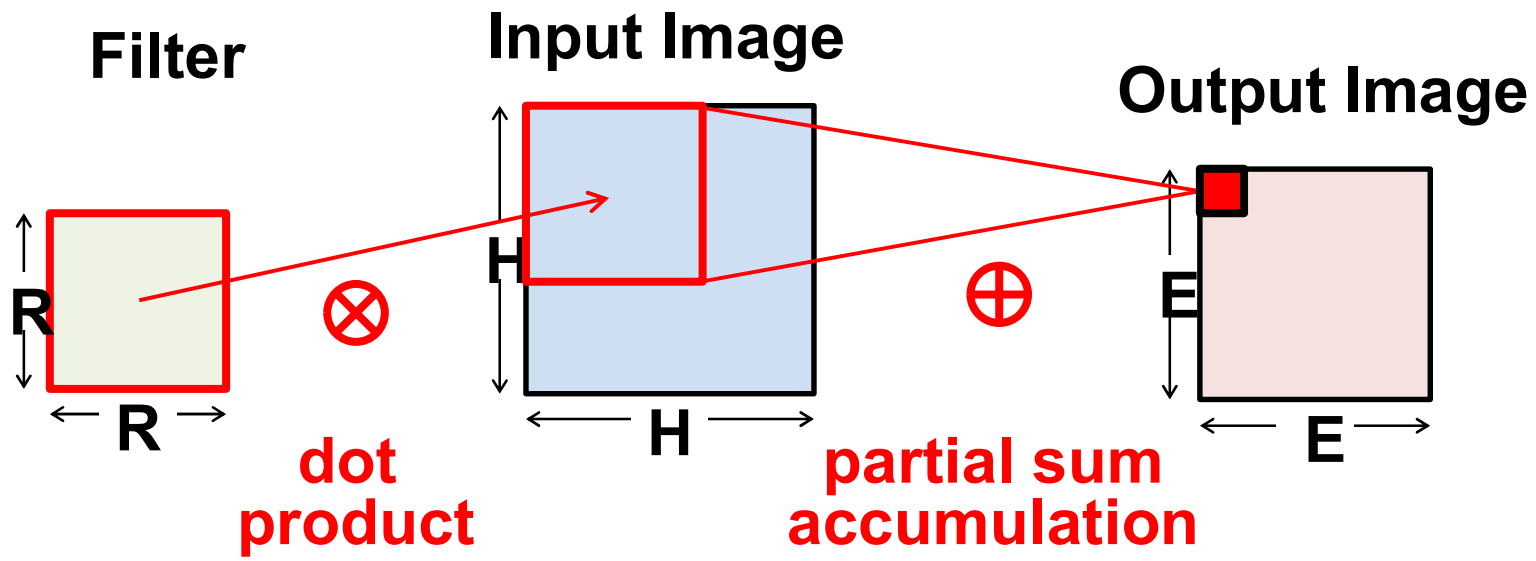
(Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53)

Convolution Layers

- 3D input / output feature maps
- 4D weights (collection of 3D filters)
- Each filter is “slid” across every possible position of the inputs
- One output for each filter/position combo
- # output channels = # filters



Convolution Layers



Convolution Layer - Pseudocode

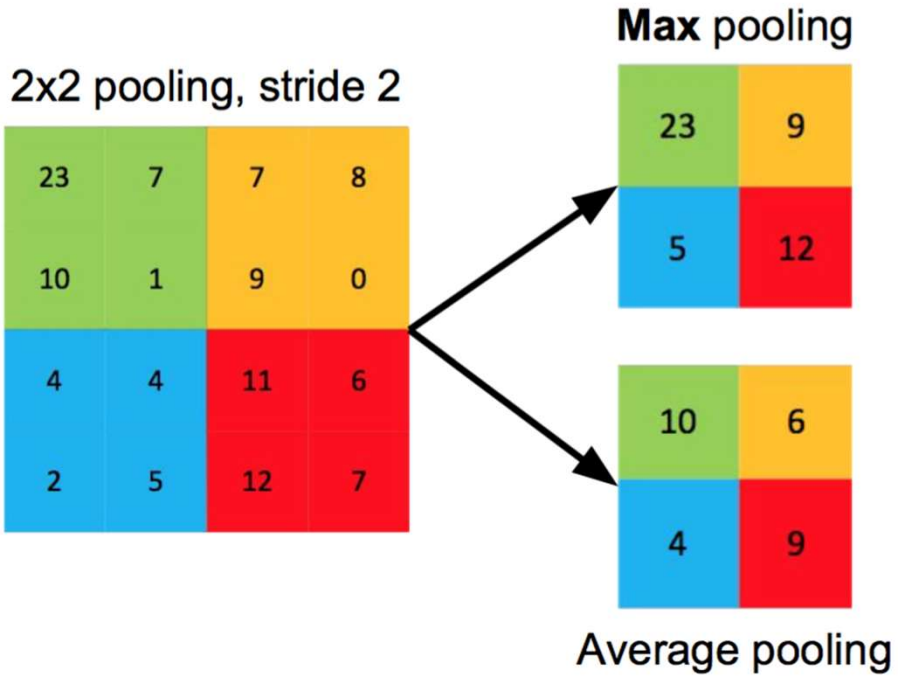
```
for(m=0; m<M; m++) { // Output Channels (Filters)
  for(c=0; c<C; c++) { // Input Channels
    for(y=0; y<H; y++) {  // Feature Map Y
      for(x=0; x<H; x++) {  // Feature Map X
        for(j=0; j<R; j++) {  // Filter Y
          for(i=0; i<R; i++) {  // Filter X
            O[m][x][y] += W[m][c][i][j] * I[c][x][y]}}}}}}}
```

Bias, Activation, Batchnorm

$$\mathbf{O}[n][m][x][y] = \text{Activation}(\mathbf{B}[m] + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} \mathbf{I}[n][k][Ux+i][Uy+j] \times \mathbf{W}[m][k][i][j]),$$

- Per-filter *biases* (also learned offline)
- Activation function (most commonly ReLU)
- Can also include batch normalization step (per-filter mean and variance) (optional)

Pooling Layers



Background – CNN Accelerators

CNN Accelerators

- High degree of parallelism makes convolution layers ideal for acceleration
- Many works on CNN acceleration in the past decade
  - GPUs
  - ASICs
  - FPGAs

DianNao (ASPLOS 2014)

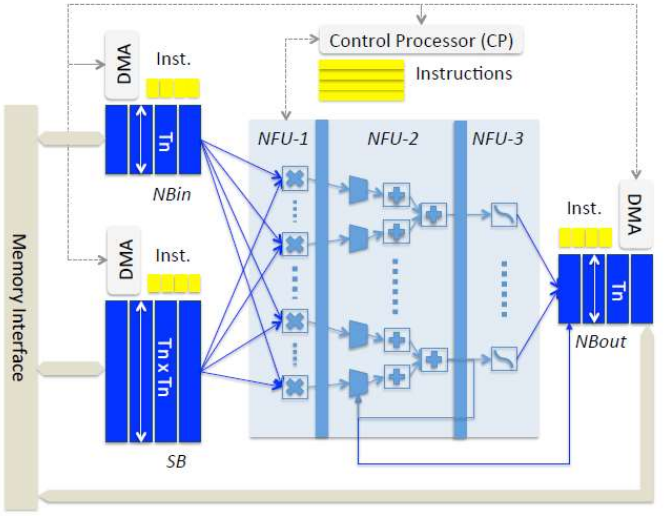
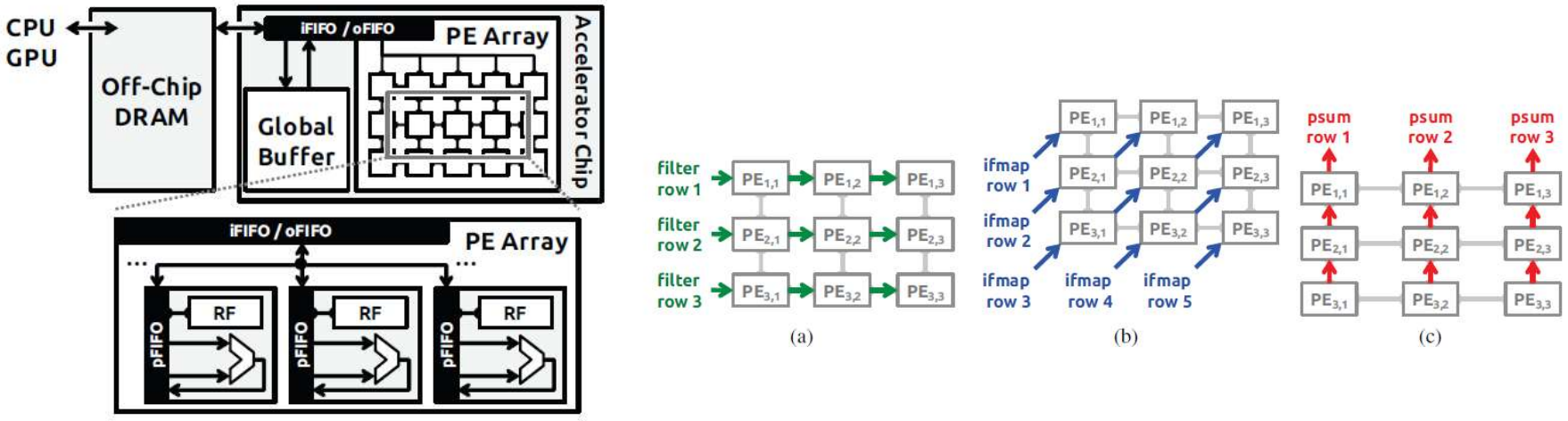


Figure 11. *Accelerator.*

```
for (int yy = 0; yy ¡ Nyin; yy += Ty)  {
  for (int xx = 0; xx ¡ Nxin; xx += Tx)  {
    for (int nnn = 0; nnn ¡ Nn; nnn += Tnn)  {
      // — Original code — (excluding nn, ii loops)
      int yout = 0;
      for (int y = yy; y < yy + Ty; y += sy) { // tiling for y;
        int xout = 0;
        for (int x = xx; x < xx + Tx; x += sx) { // tiling for x;
          for (int nn = nnn; nn < nnn + Tnn; nn += Tn) {
            for (int n = nn; n < nn + Tn; n++)
              sum[n] = 0;
            // sliding window;
            for (int ky = 0; ky < Ky; ky++)
              for (int kx = 0; kx < Kx; kx++)
                for (int ii = 0; ii < Ni; ii += Ti)
                  for (int n = nn; n < nn + Tn; n++)
                    for (int i = ii; i < ii + Ti; i++)
                      // version with shared kernels
                      sum[n] += synapse[ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
                      // version with private kernels
                      sum[n] += synapse[yout][xout][ky][kx][n][i]}
                                * neuron[ky + y][kx + x][i];
            for (int n = nn; n < nn + Tn; n++)
              neuron[yout][xout][n] = non_linear_transform(sum[n]);
          } xout++; } yout++;
} } } }
```

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269-284, New York, NY, USA, 2014. Association for Computing Machinery.

Eyeriss (ISCA 2016)

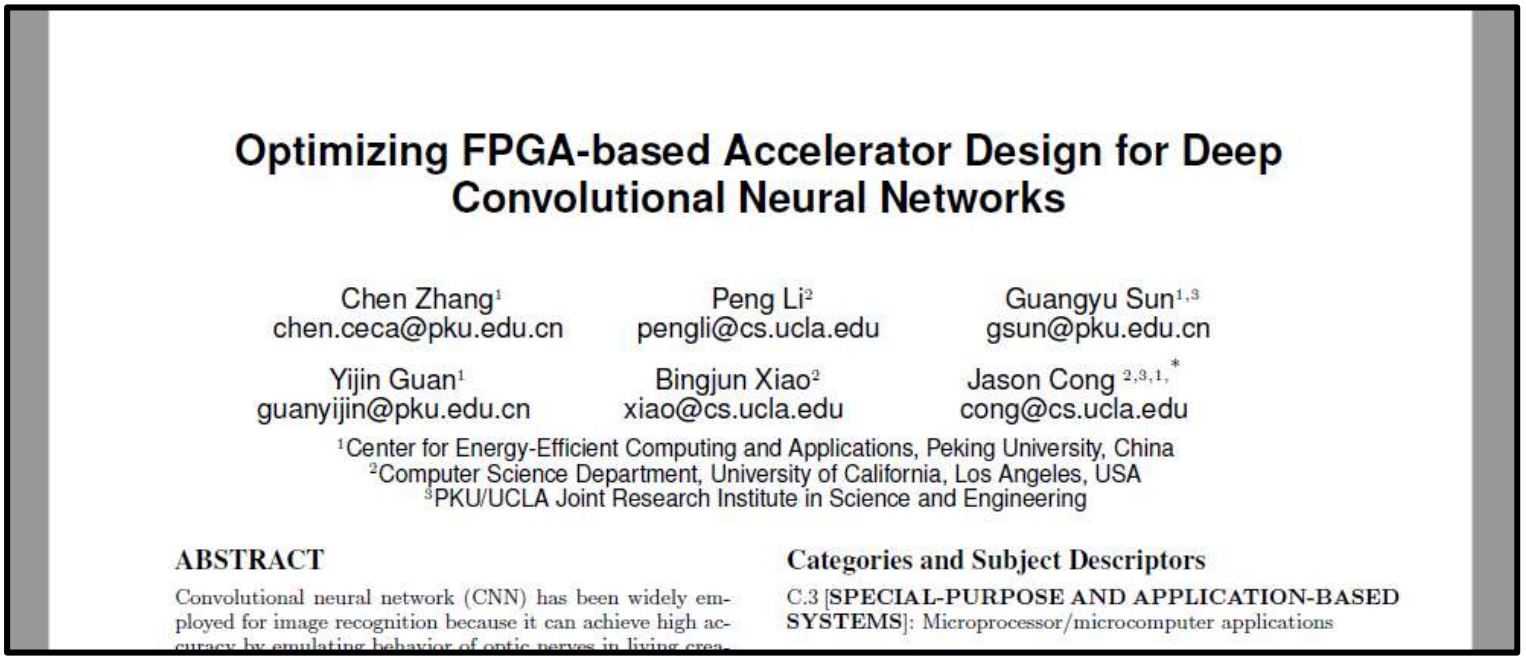


Chen, Y.H., Emer, J. and Sze, V., 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *Proc. Int'l Symp. on Computer Architecture*.

Why FPGAs for CNN accelerators?

- ASICs:
  - Pros: Best performance per watt
  - Cons: Not reprogrammable, costly
- GPUs:
  - Pros: Fully, easily reprogrammable
  - Cons: Less performant, not a dedicated accelerator
- FPGAs:
  - Bridge the gap between ASICs and GPUs
  - Can reprogram to dedicated accelerators for different networks

FPGA-Based Accelerators



Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

Chen Zhang[1]
chen.ceca@pku.edu.cn

Peng Li[2]
pengli@cs.ucla.edu

Guangyu Sun[1,3]
gsun@pku.edu.cn

Yijin Guan[1]
guanyijin@pku.edu.cn

Bingjun Xiao[2]
xiao@cs.ucla.edu

Jason Cong[2,3,1,*]
cong@cs.ucla.edu

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

ABSTRACT

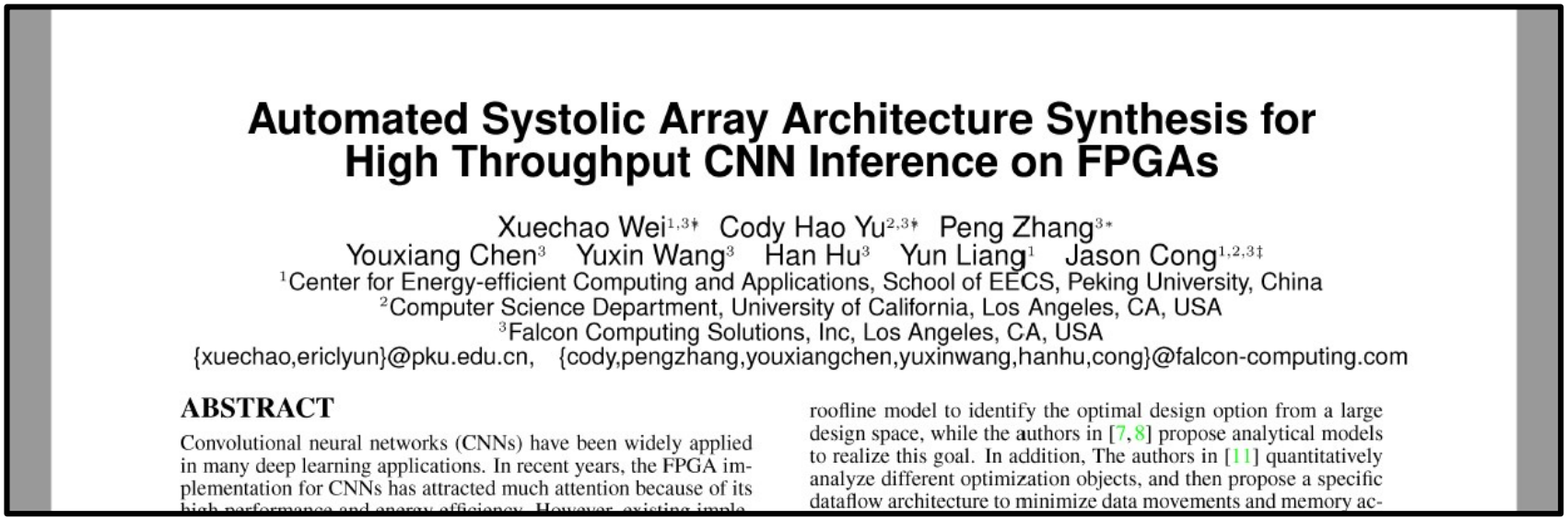
Convolutional neural network (CNN) has been widely employed for image recognition because it can achieve high accuracy by emulating behavior of optic nerves in living crea-

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Microprocessor/microcomputer applications

(FPGA '15)

FPGA-Based Accelerators



Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs

Xuechao Wei[1,3†] Cody Hao Yu[2,3†] Peng Zhang[3*]
Youxiang Chen[3] Yuxin Wang[3] Han Hu[3] Yun Liang[1] Jason Cong[1,2,3‡]
[1]Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, CA, USA
[3]Falcon Computing Solutions, Inc, Los Angeles, CA, USA
{xuechao,ericlyun}@pku.edu.cn, {cody,pengzhang,youxiangchen,yuxinwang,hanhu,cong}@falcon-computing.com

ABSTRACT

Convolutional neural networks (CNNs) have been widely applied in many deep learning applications. In recent years, the FPGA implementation for CNNs has attracted much attention because of its high performance and energy efficiency. However, existing imple-

roofline model to identify the optimal design option from a large design space, while the authors in [7, 8] propose analytical models to realize this goal. In addition, The authors in [11] quantitatively analyze different optimization objects, and then propose a specific dataflow architecture to minimize data movements and memory ac-

(DAC ’17)

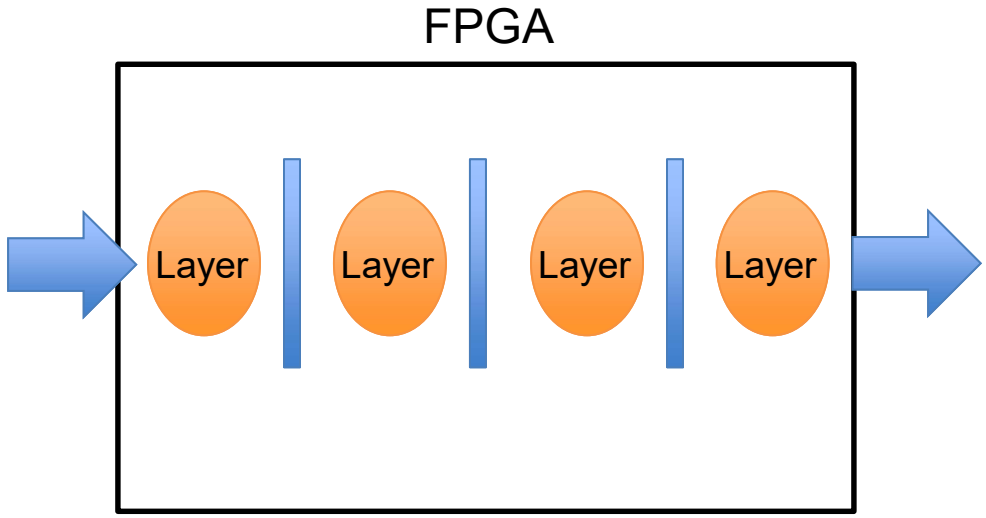
Motivation

Limitations with Existing Accelerators

- Only work on one layer at a time
  - Same hardware resources used for each layer
- Require slow off-chip memory accesses during inference
  - Primary bottleneck for a naively-designed accelerator (memory wall problem)
  - Systolic arrays, tiling maximize locality & data reuse to minimize these accesses

My Idea

- Create a network pipeline on FPGA
- Each layer is a stage, works on separate inference at a time
- No off-chip memory accesses except at beginning and end



Memory Requirements

- Per-layer memory requirements of Tiny Darknet w/ FP16:
- Input fmaps require double-buffering for pipelined execution
- Requires ~126 Mb total

#	Type	Input Dims			Filter size	Filter storage (Mb)	Input fmaps storage (Mb)
		H	W	C			
0	conv	224	224	3	3	0.007	4.59
1	max	224	224	16	-	-	24.50
2	conv	112	112	16	3	0.07	6.13
3	max	112	112	32	-	-	12.25
4	conv	56	56	32	1	0.008	3.06
5	conv	56	56	16	3	0.28	1.53
6	conv	56	56	128	1	0.03	12.25
7	conv	56	56	16	3	0.28	1.53
8	max	56	56	128	-	-	12.25
9	conv	28	28	128	1	0.06	3.06
10	conv	28	28	32	3	1.13	0.77
11	conv	28	28	256	1	0.13	6.13
12	conv	28	28	32	3	1.13	0.77
13	max	28	28	256	-	-	6.13
14	conv	14	14	256	1	0.25	1.53
15	conv	14	14	64	3	4.50	0.38
16	conv	14	14	512	1	0.50	3.06
17	conv	14	14	64	3	4.50	0.38
18	conv	14	14	512	1	1.00	3.06
19	conv	14	14	128	1	1.95	0.77
20	(final)	14	14	1000	-	-	5.98
					Total:	**15.819**	**110.11**

Is 126 Mb a lot?

Virtex-7 FPGAs

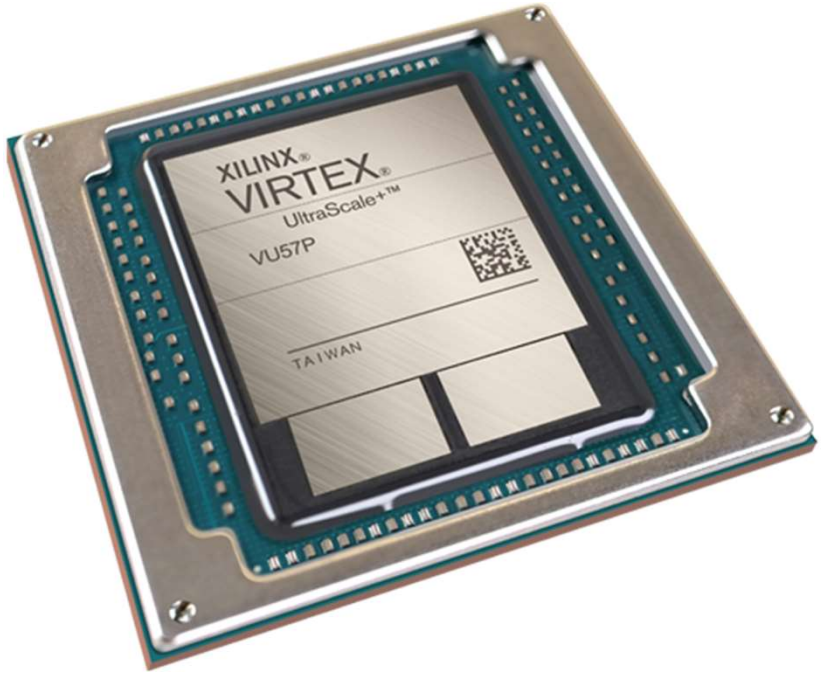
		Optimized for Highest System Performance and Capacity (1.0V)										
	Part Number	XC7V585T	XC7V2000T	XC7VX330T	XC7VX415T	XC7VX485T	XC7VX550T	XC7VX690T	XC7VX980T	XC7VX1140T	XC7VH580T	XC7VH870T
Logic Resources	Slices	91,050	305,400	51,000	64,400	75,900	86,600	108,300	153,000	178,000	90,700	136,900
	Logic Cells	582,720	1,954,560	326,400	412,160	485,760	554,240	693,120	979,200	1,139,200	580,480	876,160
	CLB Flip-Flops	728,400	2,443,200	408,000	515,200	607,200	692,800	866,400	1,224,000	1,424,000	725,600	1,095,200
Memory Resources	Maximum Distributed RAM (Kb)	6,938	21,550	4,388	6,525	8,175	8,725	10,888	13,838	17,700	8,850	13,275
	Block RAM/FIFO w/ ECC (36 Kb each)	795	1,292	750	880	1,030	1,180	1,470	1,500	1,880	940	1,410
	Total Block RAM (Kb)	28,620	46,512	27,000	31,680	37,080	42,480	52,920	54,000	67,680	33,840	50,760
Clocking	CMTs (1 MMCM + 1 PLL)	18	24	14	12	14	20	20	18	24	12	18
I/O Resources	Maximum Single-Ended I/O	850	1,200	700	600	700	600	1,000	900	1,100	600	300
	Maximum Differential I/O Pairs	408	576	336	288	336	288	480	432	528	288	144
	DSP Slices	1,260	2,160	1,120	2,160	2,800	2,880	3,600	3,600	3,360	1,680	2,520

(https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf)

Largest Virtex-7 FPGA only has 66 Mb Block RAM
(largest Xilinx FPGAs until mid-2010s, 28nm)

Xilinx Ultrascale / Ultrascale+

- High-end FPGAs introduced in mid-2010s
- 20nm / 16nm FinFET
- Largest ones have 100s Mb on-chip RAM
- Pipelined CNN design now feasible

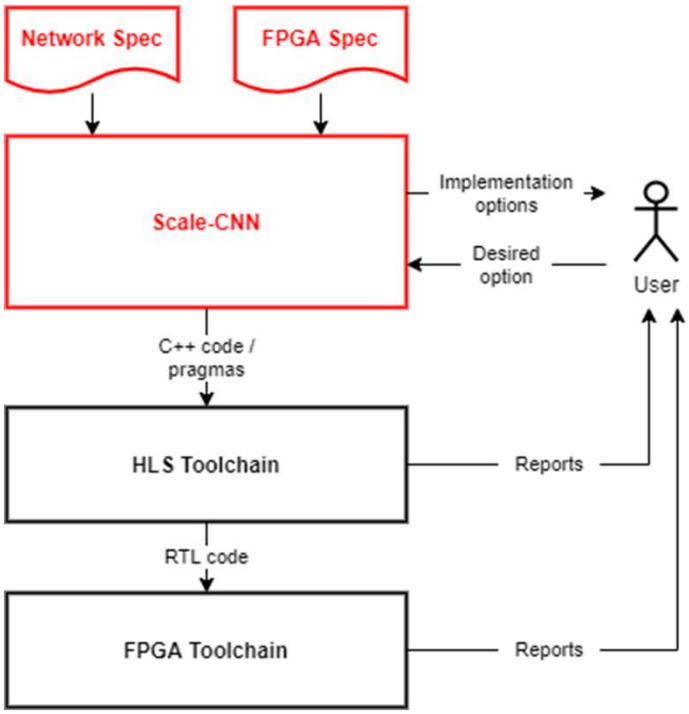


Scale-CNN Introduction

Scale-CNN

- Tool that generates high-throughput CNN inference accelerators for FPGAs
- Targets Xilinx Ultrascale+ FPGAs
- For one network, generates multiple design points with different cost/performance
  - Allows designer to choose best design point for their application

Scale-CNN Use Model



Tiny Darknet

- Small CNN optimized for minimal memory requirements
- Simple CONV layers with regularity in dimensions

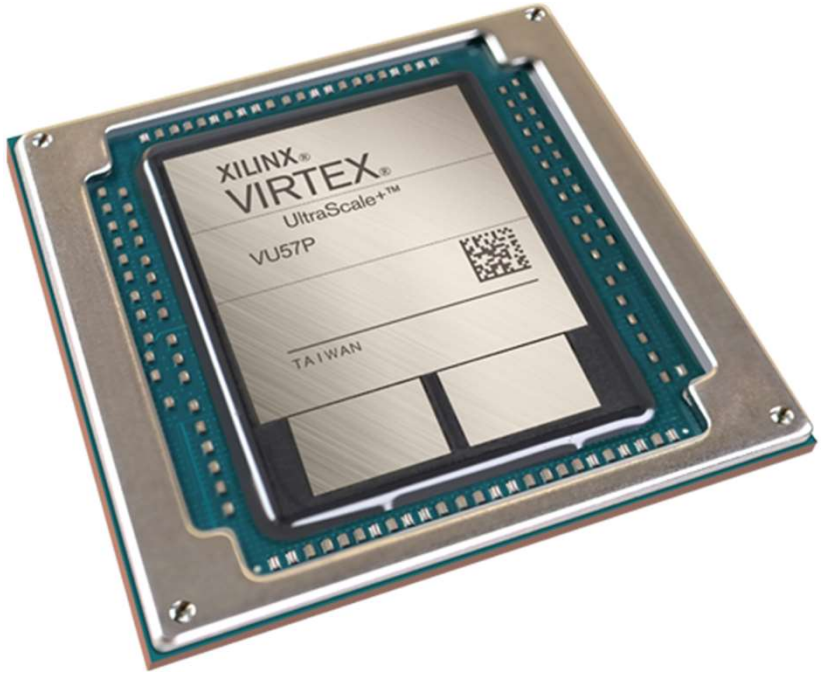
```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   224 x 224 x   3   ->   224 x 224 x  16
    1 max          2 x 2 / 2   224 x 224 x  16   ->   112 x 112 x  16
    2 conv     32  3 x 3 / 1   112 x 112 x  16   ->   112 x 112 x  32
    3 max          2 x 2 / 2   112 x 112 x  32   ->    56 x  56 x  32
    4 conv     16  1 x 1 / 1    56 x  56 x  32   ->    56 x  56 x  16
    5 conv    128  3 x 3 / 1    56 x  56 x  16   ->    56 x  56 x 128
    6 conv     16  1 x 1 / 1    56 x  56 x 128   ->    56 x  56 x  16
    7 conv    128  3 x 3 / 1    56 x  56 x  16   ->    56 x  56 x 128
    8 max          2 x 2 / 2    56 x  56 x 128   ->    28 x  28 x 128
    9 conv     32  1 x 1 / 1    28 x  28 x 128   ->    28 x  28 x  32
   10 conv    256  3 x 3 / 1    28 x  28 x  32   ->    28 x  28 x 256
   11 conv     32  1 x 1 / 1    28 x  28 x 256   ->    28 x  28 x  32
   12 conv    256  3 x 3 / 1    28 x  28 x  32   ->    28 x  28 x 256
   13 max          2 x 2 / 2    28 x  28 x 256   ->    14 x  14 x 256
   14 conv     64  1 x 1 / 1    14 x  14 x 256   ->    14 x  14 x  64
   15 conv    512  3 x 3 / 1    14 x  14 x  64   ->    14 x  14 x 512
   16 conv     64  1 x 1 / 1    14 x  14 x 512   ->    14 x  14 x  64
   17 conv    512  3 x 3 / 1    14 x  14 x  64   ->    14 x  14 x 512
   18 conv    128  1 x 1 / 1    14 x  14 x 512   ->    14 x  14 x 128
   19 conv   1000  1 x 1 / 1    14 x  14 x 128   ->    14 x  14 x1000
   20 avg                       14 x  14 x1000   ->  1000
   21 softmax                                        1000
   22 cost                                           1000
```

Model	Top-1	Top-5	Ops	Size
AlexNet	57.0	80.3	2.27 Bn	238 MB
Darknet Reference	**61.1**	**83.0**	**0.81 Bn**	28 MB
SqueezeNet	57.5	80.3	2.17 Bn	4.8 MB
Tiny Darknet	58.7	81.7	0.98 Bn	**4.0 MB**

(https://pjreddie.com/darknet/tiny-darknet/)

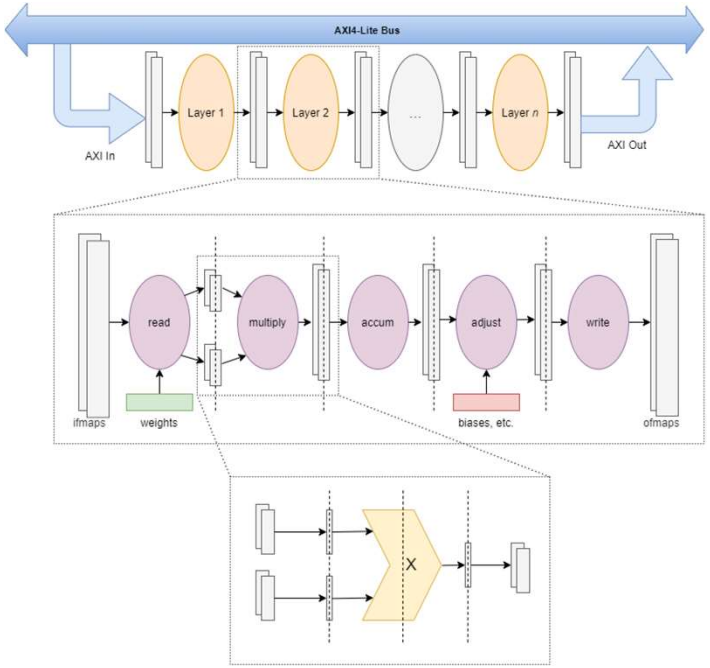
Ultrascale+ FPGAs

- Enough on-chip memory to store all weights and feature maps on-chip
- Contain high-capacity "UltraRAMs" (URAMs) with 288 Kb storage
  - Traditional Block RAMs (BRAMs) only 18 Kb



Scale-CNN Architecture

Scale-CNN Architecture



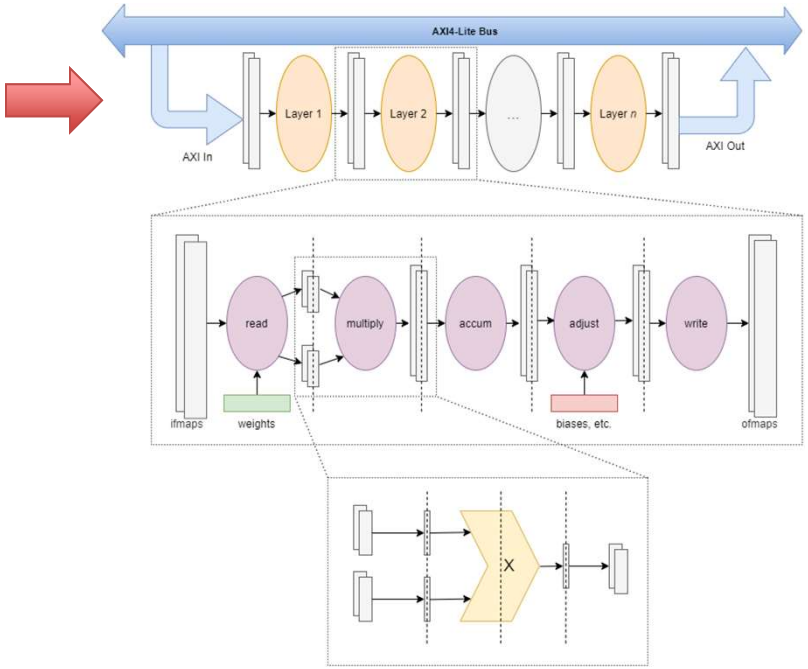
High-level network pipeline

Mid-level layer pipeline

Low-level function pipeline

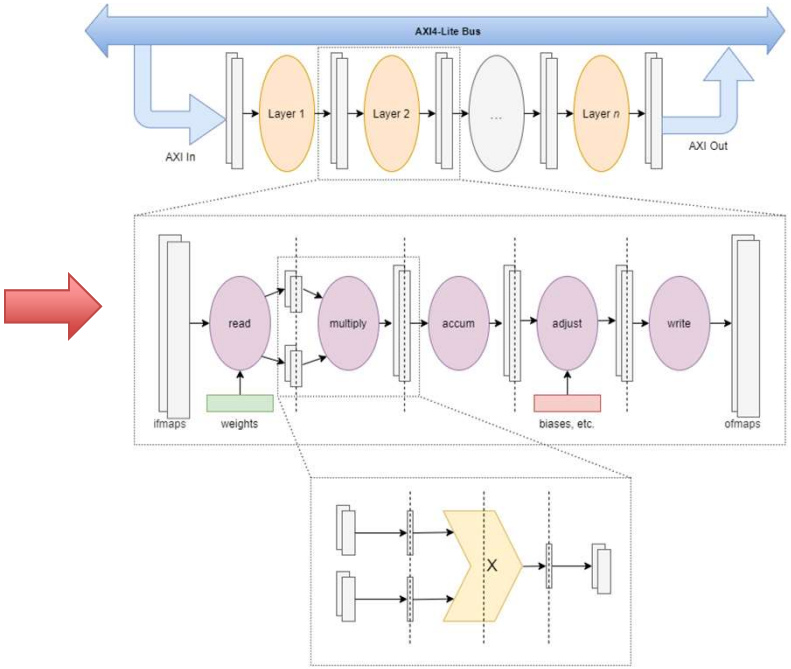
Scale-CNN Architecture

- Highest level: Network pipeline
- Each layer is one stage, works on different inferences in parallel
- Beginning and end connect network to AXI4-Lite bus



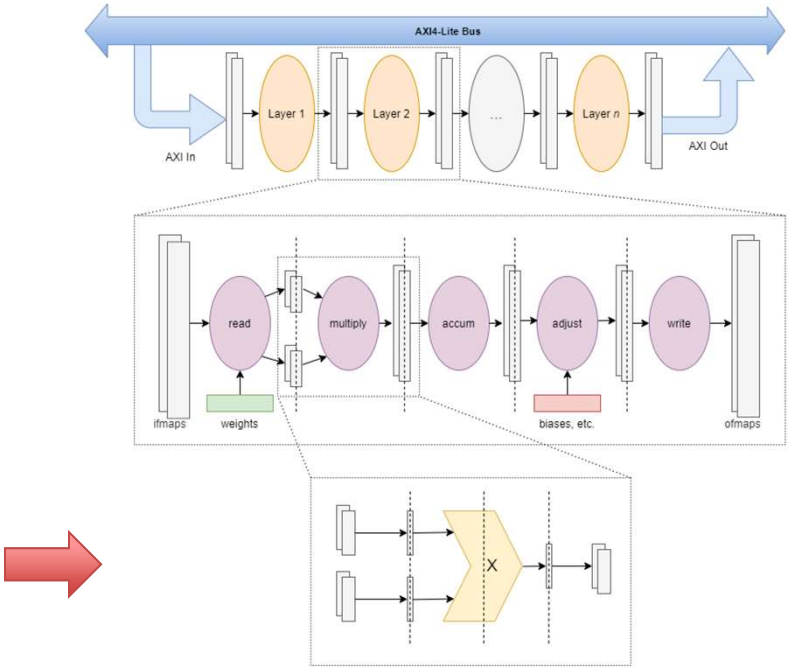
Scale-CNN Architecture

- Middle level: Layer pipeline
- Each stage does one part of the computation for one or more output fmap elements
- Stages work on different outputs in parallel



Scale-CNN Architecture

- Lowest level: Function pipeline
- Traditional pipeline where each stage is one clock cycle



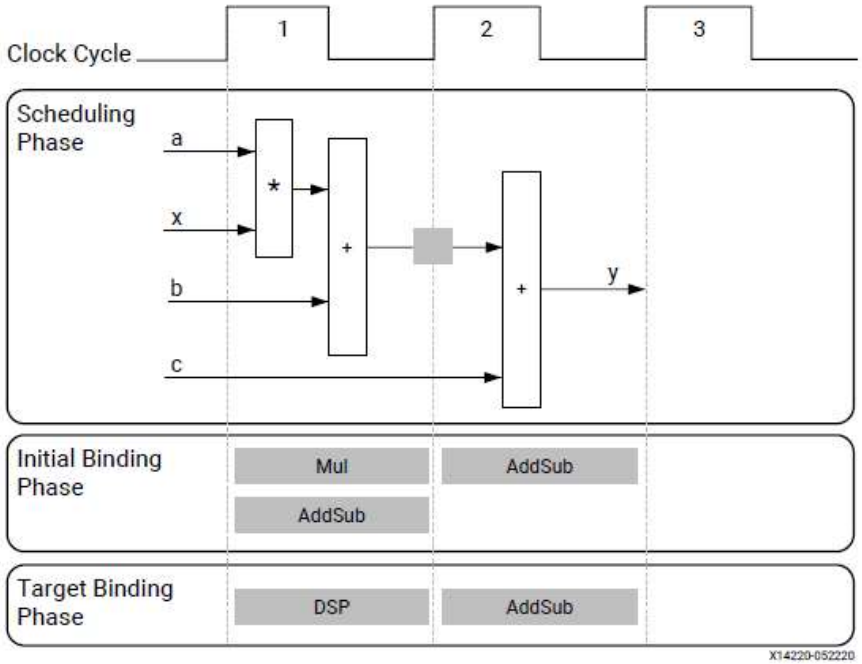
HLS
(High-Level Synthesis)

High-Level Synthesis

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

- C/C++ code → RTL code
- Respects data dependencies
- Optimizes hardware for high throughput / low latency

Figure 1: **Scheduling and Binding Example**



Why HLS?

- Allows designer to specify higher-level design behavior without providing specific implementation details
- Implementation details are decoupled from code that describes behavior
- Useful for generating multiple implementations of the same thing

HLS Pragmas & Directives

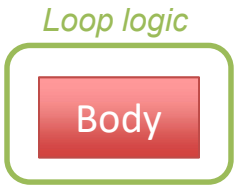
- Specify implementation details of design
- *Pragmas* live inside the code (preprocessor directives)
  - Preferred when optimization doesn't change
  - E.g.: `#pragma HLS pipeline`
- *Directives* are separate, stated as TCL commands
  - Preferred when optimization differs between design points
  - E.g.: `set_directive_unroll [LOOP_NAME] -factor 2`

HLS Pragmas & Directives

- Many optimizations supported by Vitis HLS
- Main ones used by Scale-CNN:
  - Loop Unrolling
  - Array Partitioning
  - Array Reshaping
  - Pipelining
  - Dataflow Pipelining
  - Binding

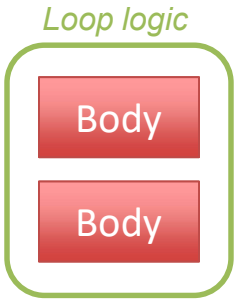
Loop Unrolling

```
for (int i = 0; i < N; i++)
{
    foo(i);
}
```



One iteration at a time

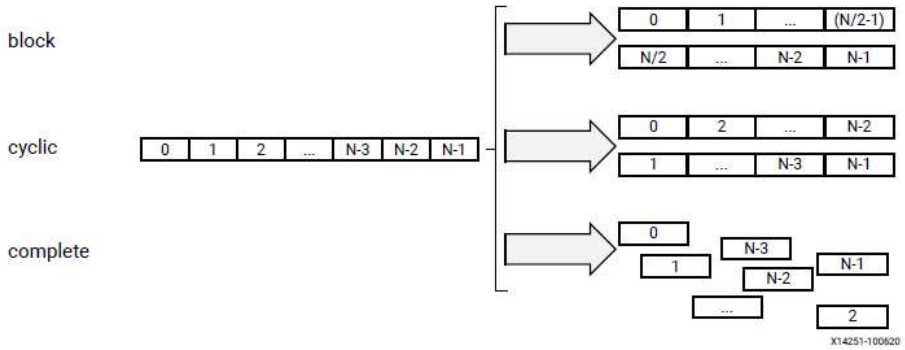

```
for (int i = 0; i < N; i += 2)
{
    foo(i);
    foo(i+1);
}
```



Multiple iterations in parallel

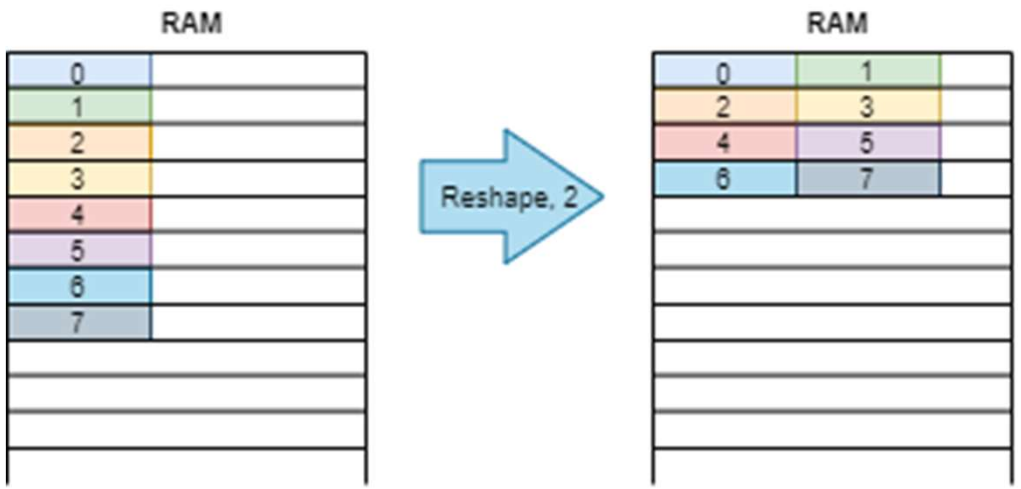
Array Partitioning

- Splits arrays into multiple RAMs
- Enables multiple array elements to be accessed each cycle
- Needed to support unrolling loops that access arrays
- Complete partitioning: Put all array elements in FFs



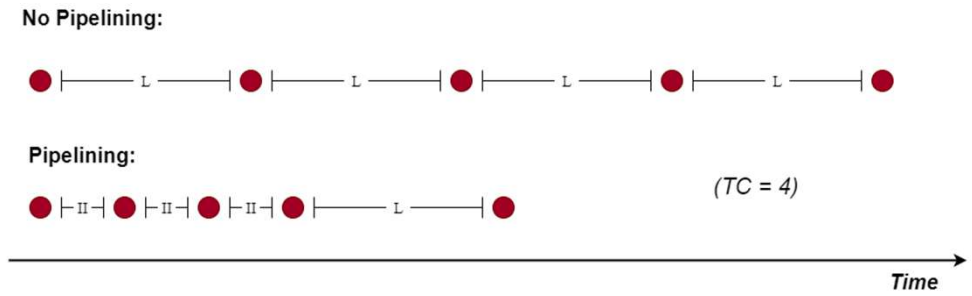
Array Reshaping

- Default: One array element per RAM row
- Packs multiple elements into same RAM row
- Useful for 72-bit wide UltraRAMs holding 16-bit data
- Extension of array partitioning, allows reading multiple words per cycle



Pipelining

- Breaks loop body into stages, enabling multiple iterations to execute in parallel
- *TC:* Trip Count (# iterations)
  L: Iteration latency
  II: Initiation Interval
- Ideal scenario: *II* = 1
- Total loop execution time:
  - *TC* * *L* (no pipelining)
  - (*TC*-1) * *II* + *L* (with pipelining)



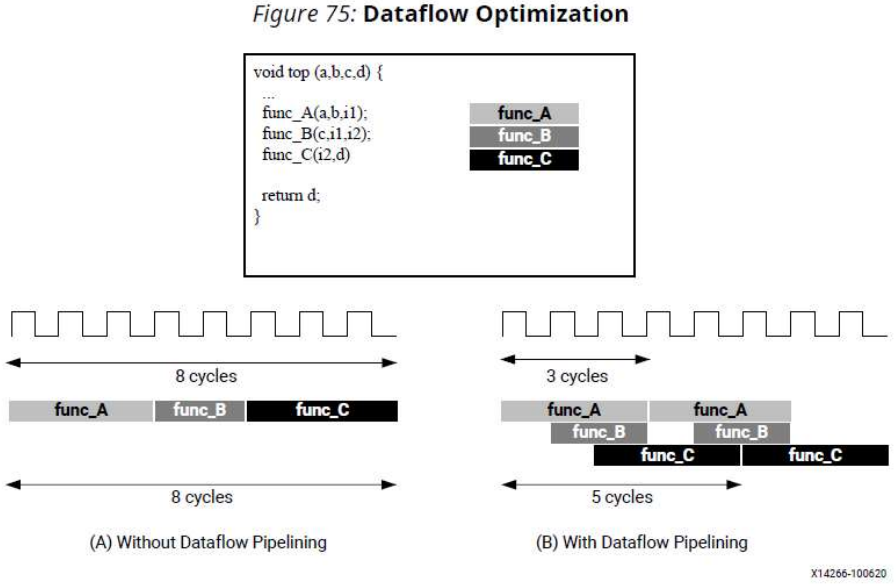
Pipelining + Loop Unrolling

- *Pipelining*: Multiple iterations in same loop body instance
- *Loop unrolling*: Multiple loop body instances
- Can partially unroll the loop and pipeline each instance
- Analogy: Multi-lane drive thru

Dataflow Pipelining

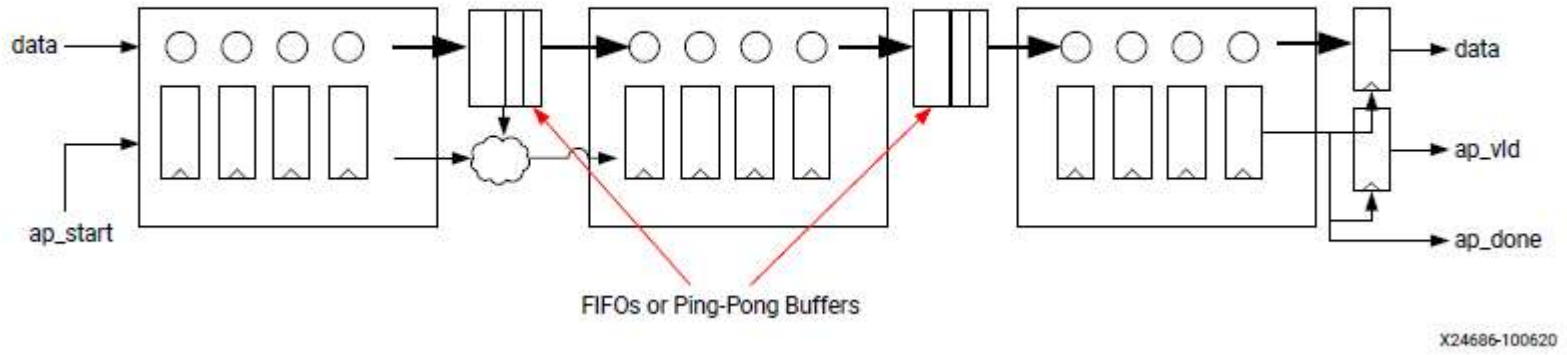
- Coarse-grained pipeline, each stage is a task
- Stages take multiple cycles
- Different stages have different latencies
- II = Latency of longest stage
- L = Sum of stage latencies

Figure 75: **Dataflow Optimization**



Dataflow Pipelining

Figure 76: **Structure Created During Dataflow Optimization**



- Ping-pong buffers prevent next iteration from overwriting previous iteration's data while it is being used
- Flow control is decentralized, managed independently at stage boundaries

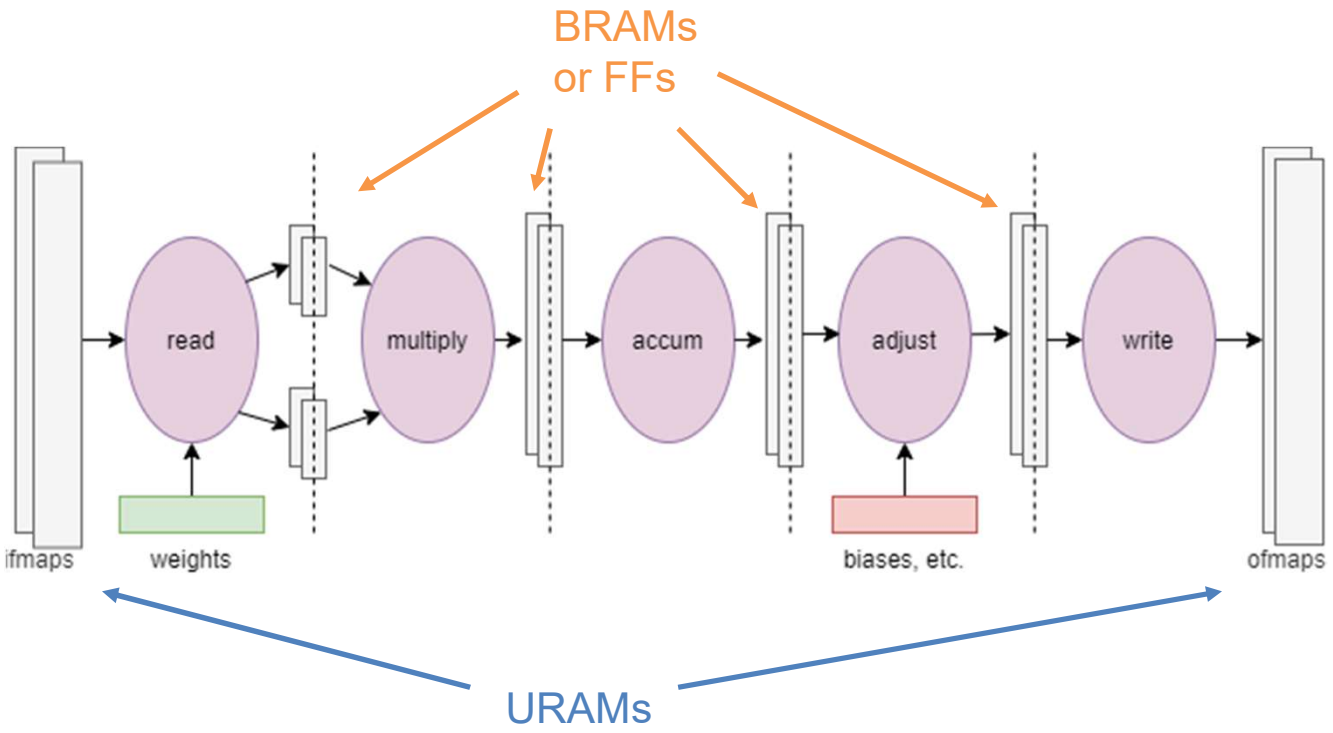
Binding

- Specifies an implementation for array storage (*bind_storage*) or arithmetic operation (*bind_op*)
- Array storage:
  - Primitive type: BRAM, URAM, Distributed RAM
  - Ports: 1 R/W, 2 R/W, 2R + 2W, 1 R/W + 1 R, etc.
- Arithmetic operation:
  - All LUTs
  - 1 DSP, fewer LUTs
  - 2 DSPs, even fewer LUTs
- If unspecified, tool makes decisions on its own using heuristics
  - But tool does not know higher-level design goals

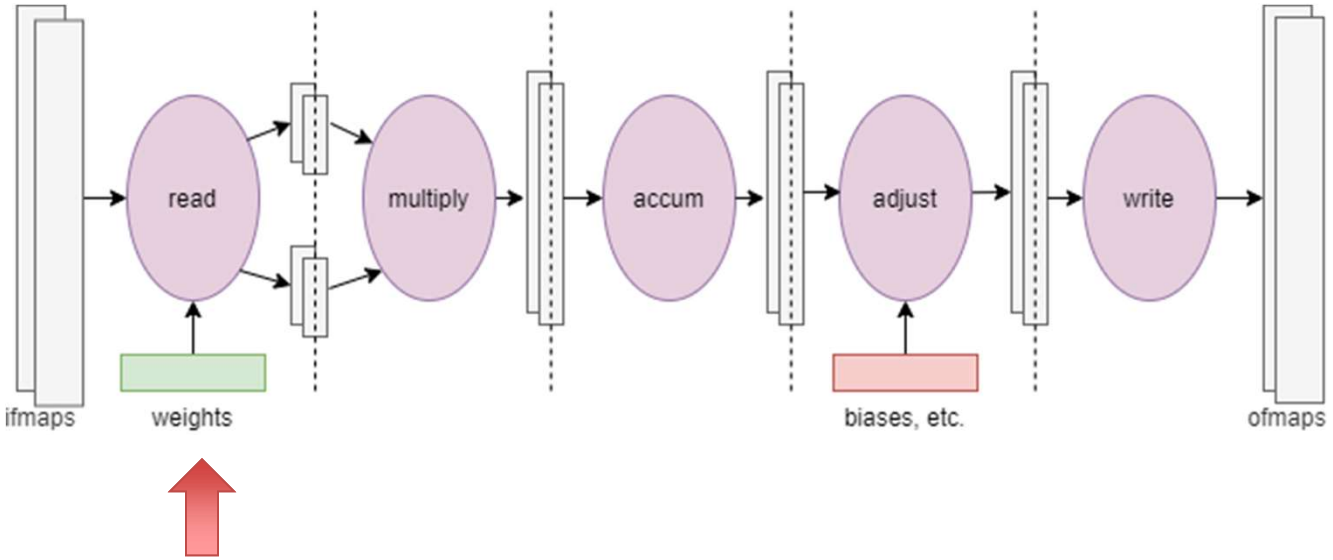
Layer Design

CONV Layer Design

Dataflow Pipeline

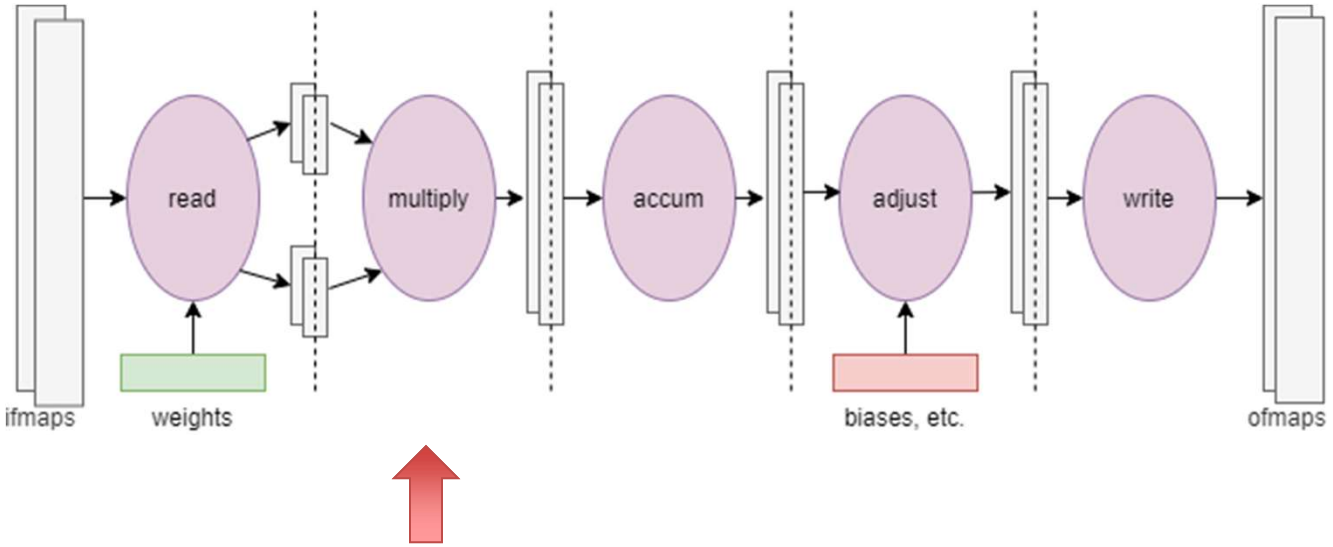


CONV Layer Design



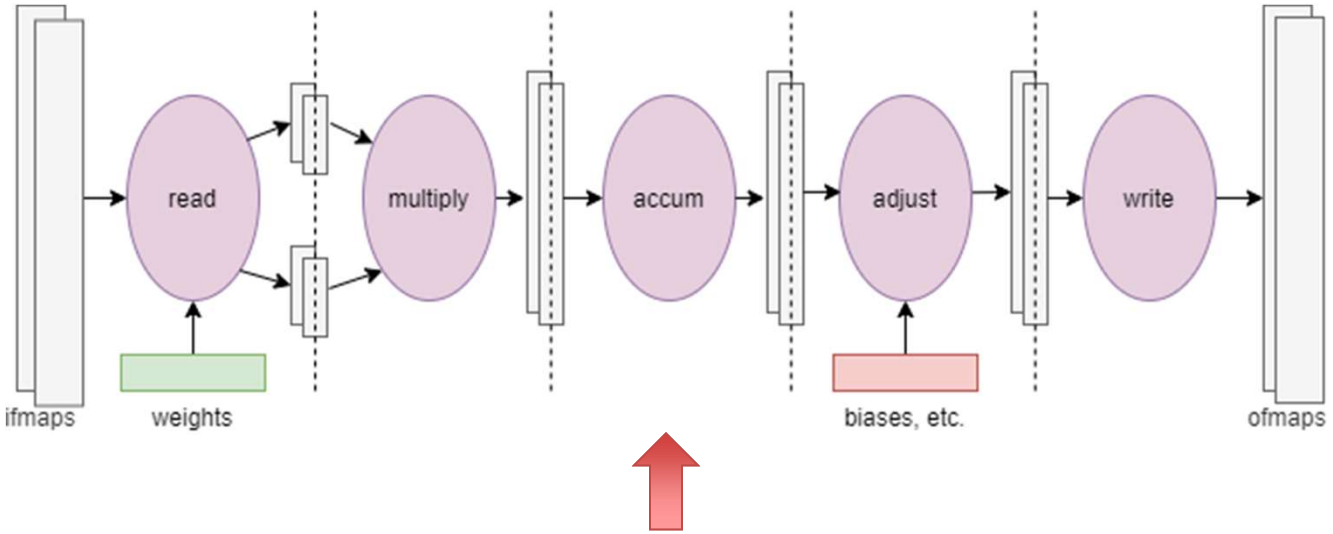
read:
Reads one convolution window of inputs + one filter

CONV Layer Design



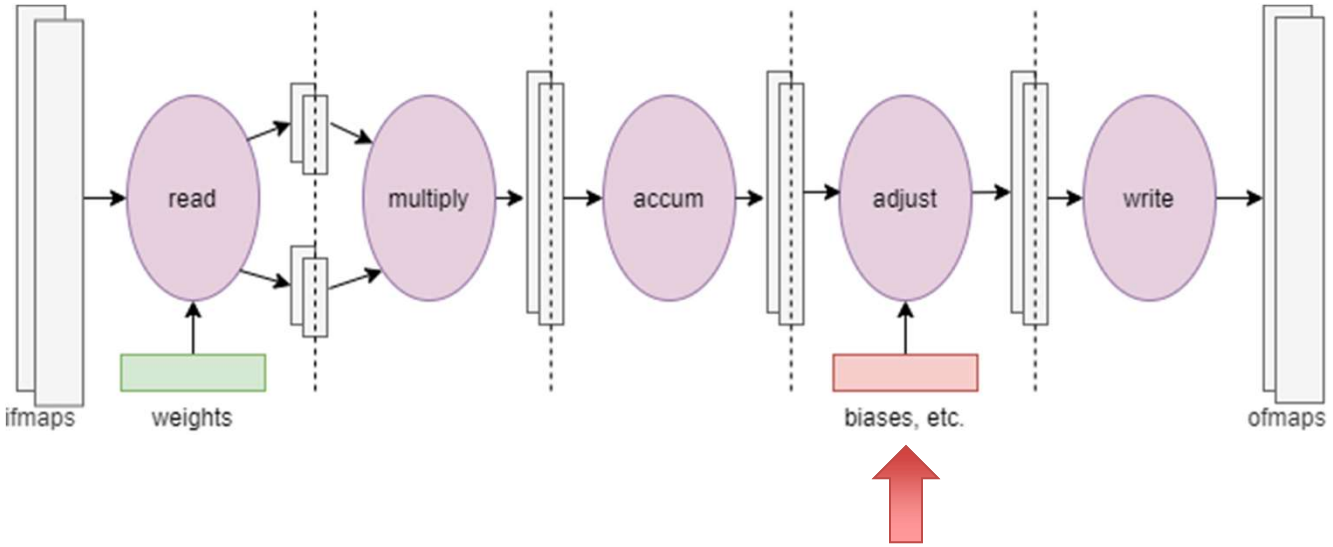
multiply:
Multiplies each ifmap element with corresponding weight element

CONV Layer Design



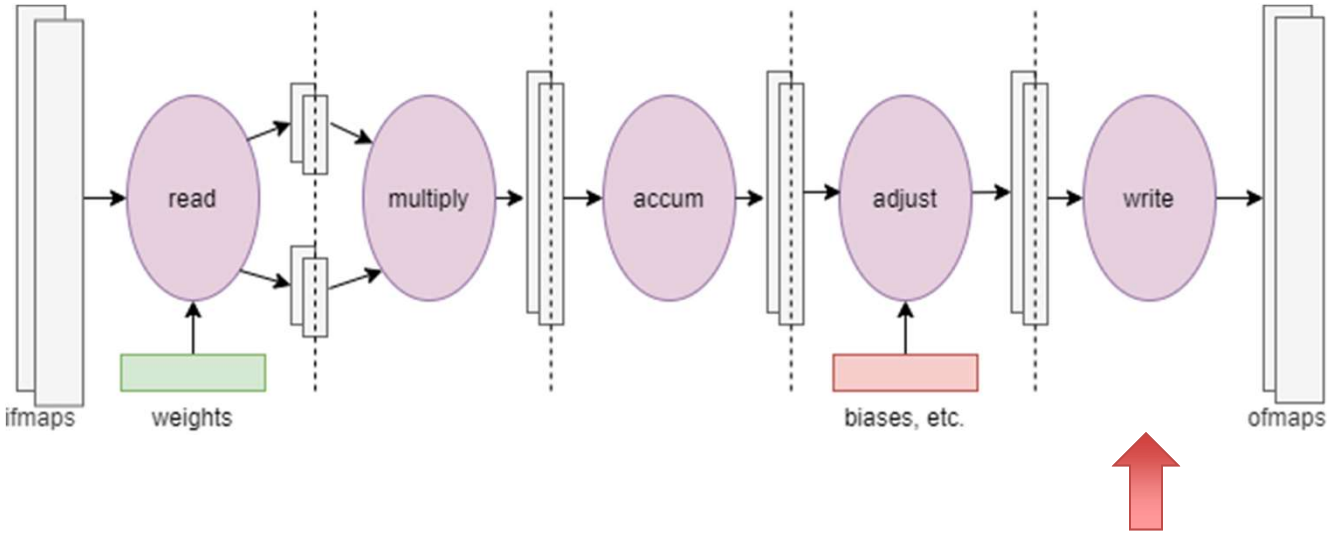
accum:
Accumulates products from *multiply* stage into a sum

CONV Layer Design



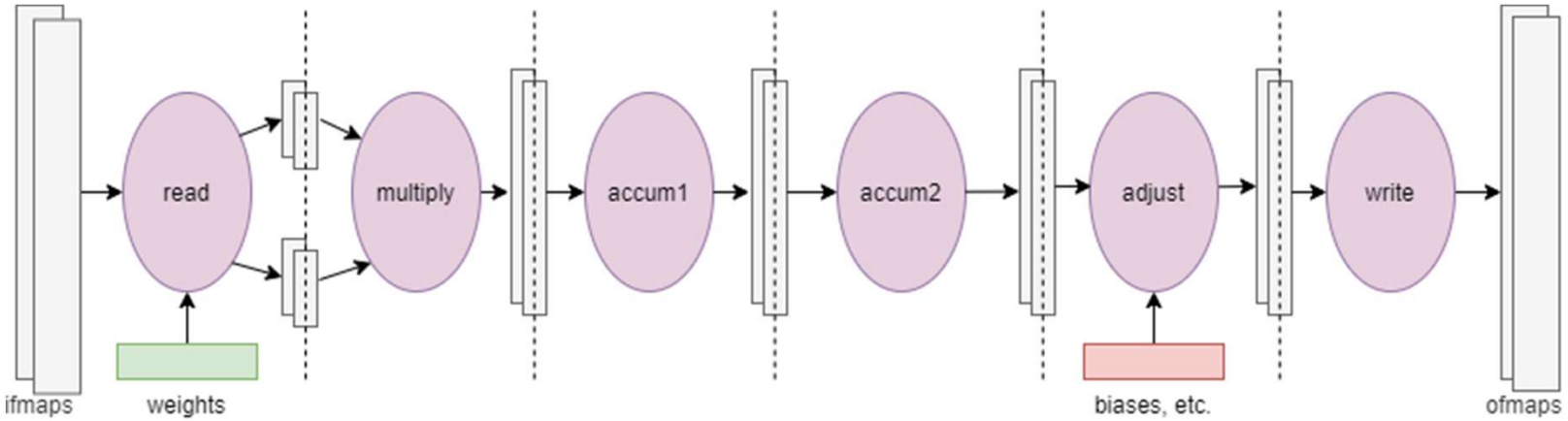
adjust:
Applies batch normalization (optional), bias, activation (ReLU)

CONV Layer Design



write:
Writes elements to output feature map array

Multiple Accumulation Stages

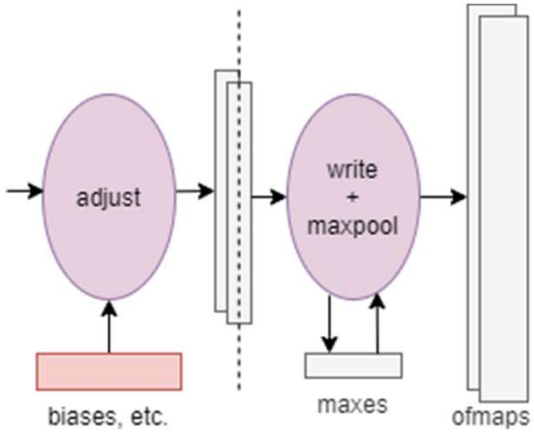


Accumulation can be split up into multiple stages
Goal: accumulation stage should never be the bottleneck
Scale-CNN dynamically chooses # stages per layer implementation

Layer Fusing

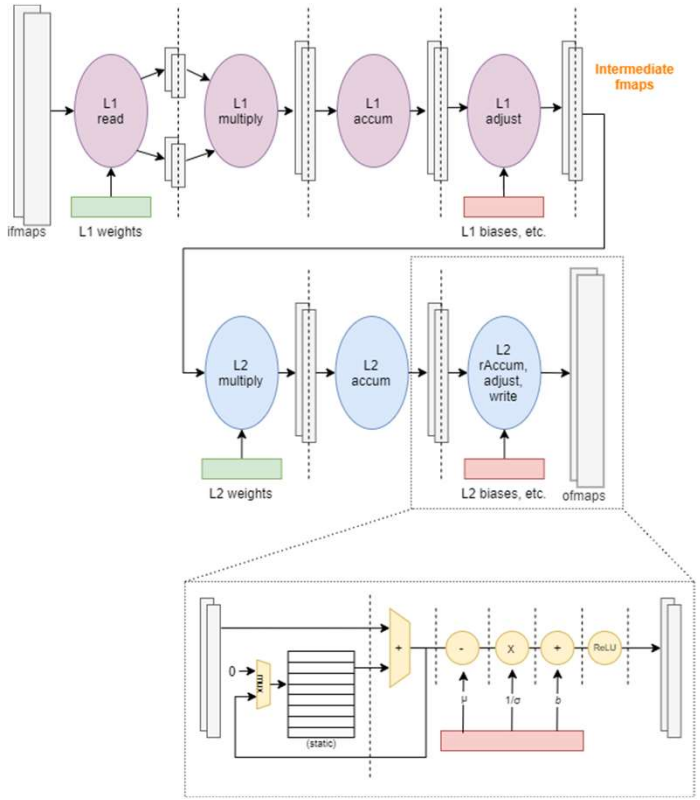
CONV-MAX Layers

- Fused CONV – Maxpool layer
- *Write* stage augmented with running maximum for each output element
- Only writes to ofmaps every P^2 iterations
  - (P = pooling factor)
- Iteration order over input feature maps is changed



CONV-CONV Layers

- Limited support for fusing CONV layers together
- Requires second layer to have 1x1 filters
  - Convenient for Tiny Darknet – many such layers

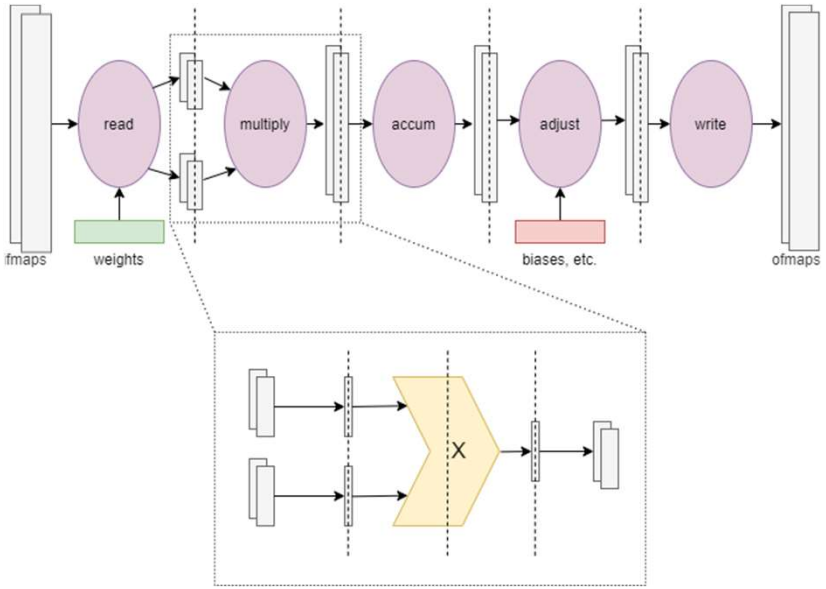


Memory Requirements with Fusing

#	Type	Input Dims			Filter size	Filter storage (Mb)	Input fmaps storage (Mb)		
		H	W	C			no fusing	conv-max	conv-max / conv-conv
0	conv	224	224	3	3	0.007	4.59	4.59	4.59
1	max	224	224	16	-	-	24.50	**0**	**0**
2	conv	112	112	16	3	0.07	6.13	6.13	6.13
3	max	112	112	32	-	-	12.25	**0**	**0**
4	conv	56	56	32	1	0.008	3.06	3.06	3.06
5	conv	56	56	16	3	0.28	1.53	1.53	1.53
6	conv	56	56	128	1	0.03	12.25	12.25	**0**
7	conv	56	56	16	3	0.28	1.53	1.53	1.53
8	max	56	56	128	-	-	12.25	**0**	**0**
9	conv	28	28	128	1	0.06	3.06	3.06	3.06
10	conv	28	28	32	3	1.13	0.77	0.77	0.77
11	conv	28	28	256	1	0.13	6.13	6.13	**0**
12	conv	28	28	32	3	1.13	0.77	0.77	0.77
13	max	28	28	256	-	-	6.13	**0**	**0**
14	conv	14	14	256	1	0.25	1.53	1.53	1.53
15	conv	14	14	64	3	4.50	0.38	0.38	0.38
16	conv	14	14	512	1	0.50	3.06	3.06	**0**
17	conv	14	14	64	3	4.50	0.38	0.38	0.38
18	conv	14	14	512	1	1.00	3.06	3.06	**0**
19	conv	14	14	128	1	1.95	0.77	0.77	0.77
20	(final)	14	14	1000	-	-	5.98	5.98	5.98
					Total:	**15.819**	**110.11**	**54.98**	**30.48**

Layer Implementations

Base Case

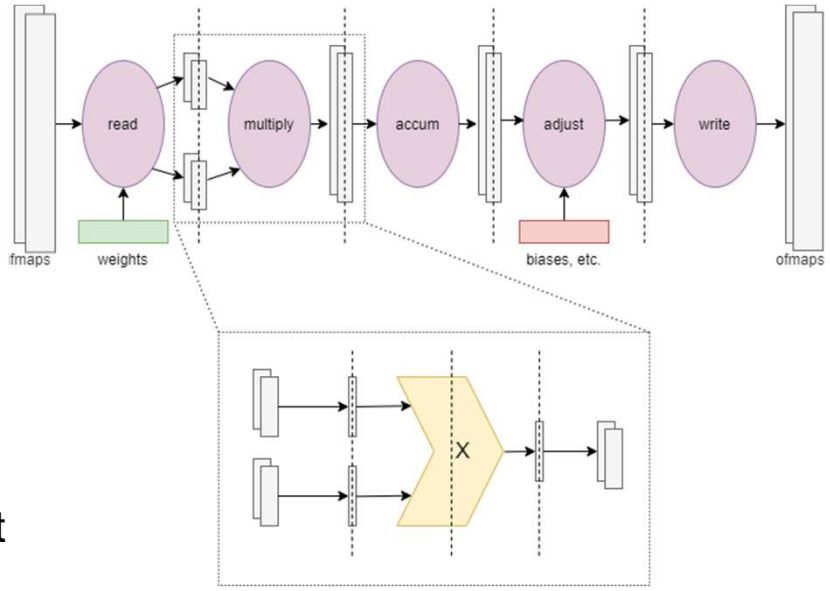


One element at a time

One element at a time

How to make this faster?

For each output element:

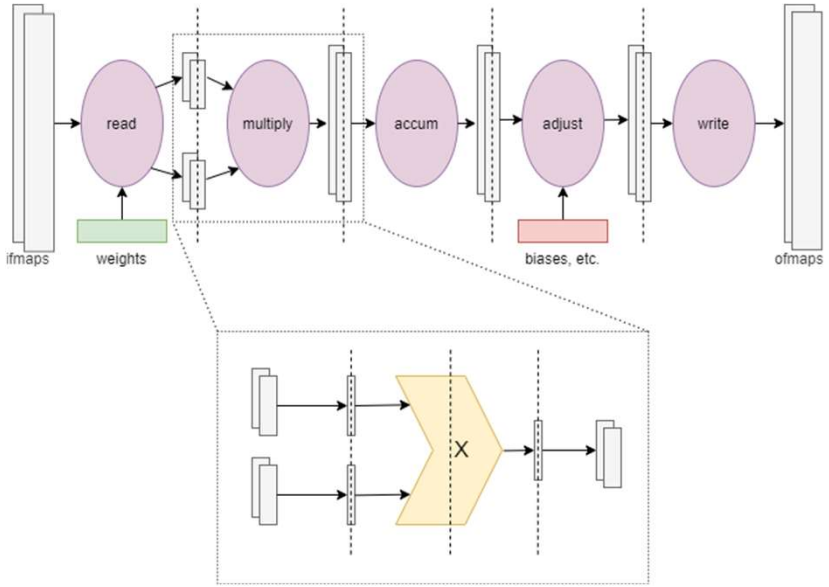


Iterates over:
- Output width
- Output height
- Output channels

For each input element required to compute one output element:

Iterates over:
- Filter width
- Filter height
- Input channels

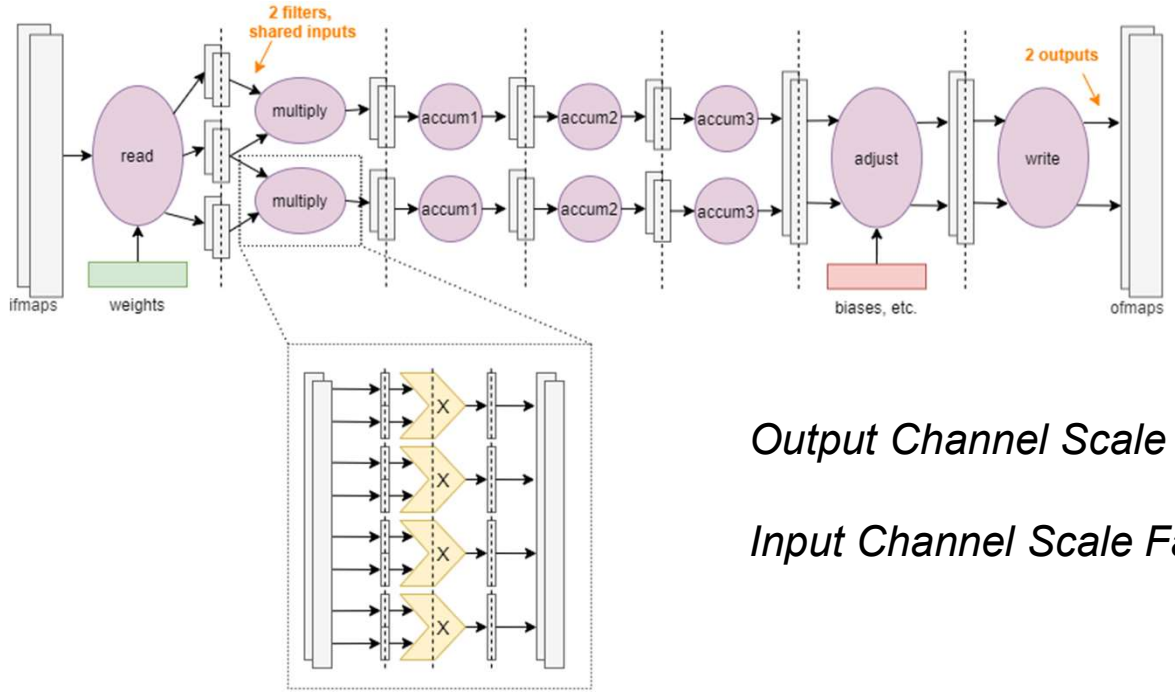
Loop Unrolling



Unroll the **output channel** dimension (output channel scaling): Reduces the ***trip count*** of the layer pipeline

Unroll the **input channel** dimension (input channel scaling): Reduces the ***initiation interval*** of the layer pipeline by reducing the stage latencies

Channel Scaling (Loop Unrolling)



Output Channel Scale Factor (OCSF) = 2

Input Channel Scale Factor (ICSF) = 4

Layer Implementations

- Each layer implementation characterized by its two scale factors
- ICSF must be factor of # input channels
- OCSF must be factor of # output channels
- One implementation for each permutation of {factors(# input chans), factors(# output chans)}

Layer Implementations

- Example: 3 input channels, 8 output channels
- Possible layer implementations:

i1_o1	*i3_o1*
i1_o2	*i3_o2*
i1_o4	*i3_o4*
i1_o8	*i3_o8*

- More scaling = better performance, higher cost

Layer Implementation Results

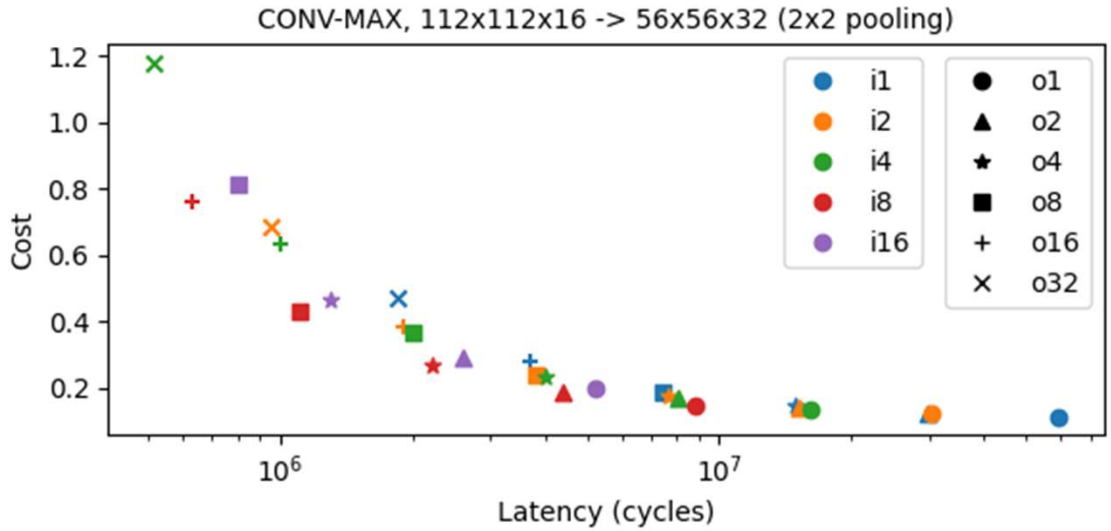
Cost Function

- Need a cost function to compare layer implementations
- HLS tool reports individual resource utilizations, need to reduce this to a single quantity
- Scale-CNN uses *sum-of-percentages* cost:

$$Cost = \sum_i \frac{r_i}{R_i}$$

- r_i = Amount of resource *i* utilized
- R_i = Amount of resource *i* available
- $i \in$ {LUTs, FFs, DSPs, BRAMs, URAMs}

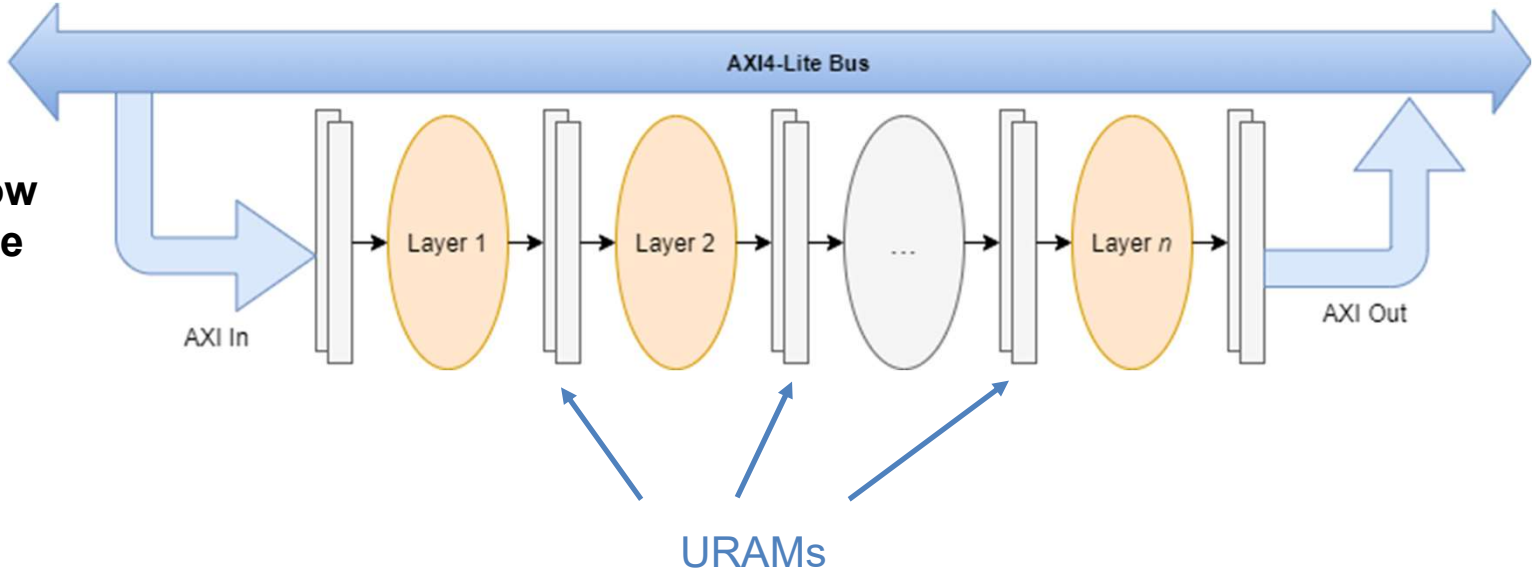
Layer Implementation Results



Network Design / Implementations

Network Design

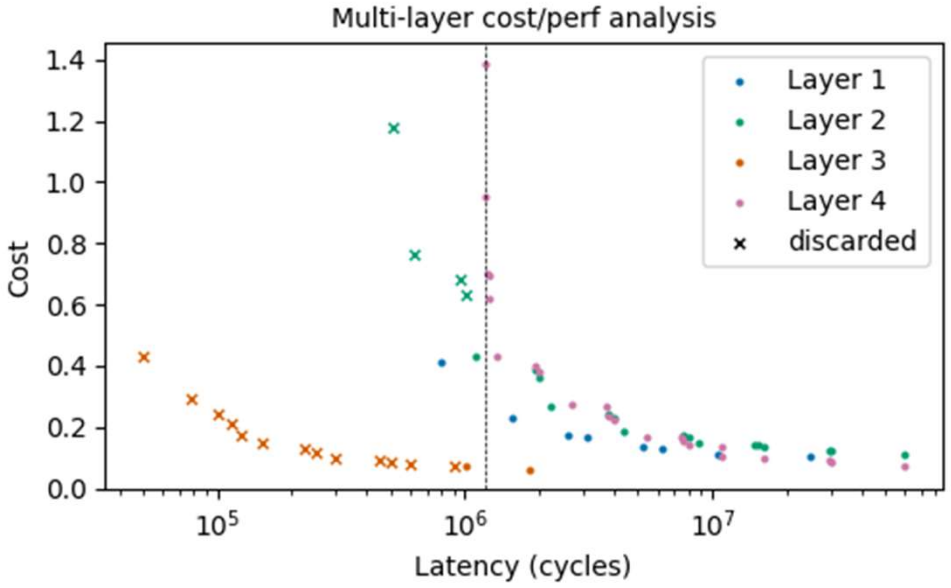
Dataflow Pipeline



Network Implementations

- For each layer:
  - Generate all possible implementations
  - Synthesize all of them to get cost/perf stats
  - Filter out those that are not Pareto optimal
- "Mix-and-match" layer implementations to make network implementations
  - Choose one implementation for each layer
  - How to choose intelligently?

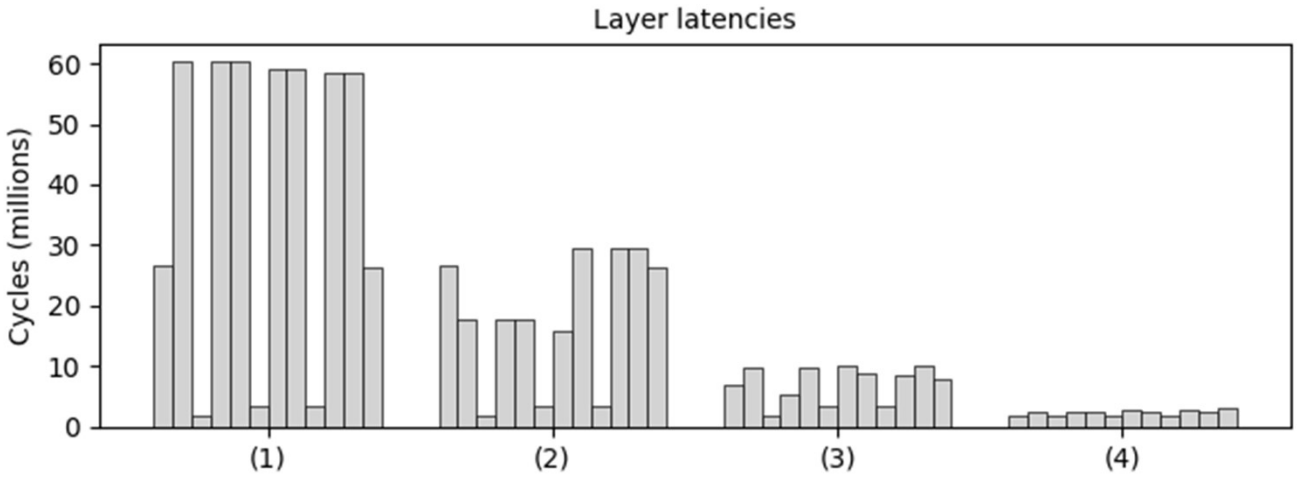
Analyzing Layer Options



Layer Selection Algorithm

1. Start with the slowest implementation of each layer
   - This is the first network design point
2. Find the layer with the highest latency. This is the bottleneck
3. Switch this layer to the next fastest implementation
   - This is a new network design point
4. Repeat 2-3 until the bottlenecking layer is already using the fastest possible implementation
5. At the end, discard any points that are estimated to exceed available FPGA resources

Layer Selection Algorithm



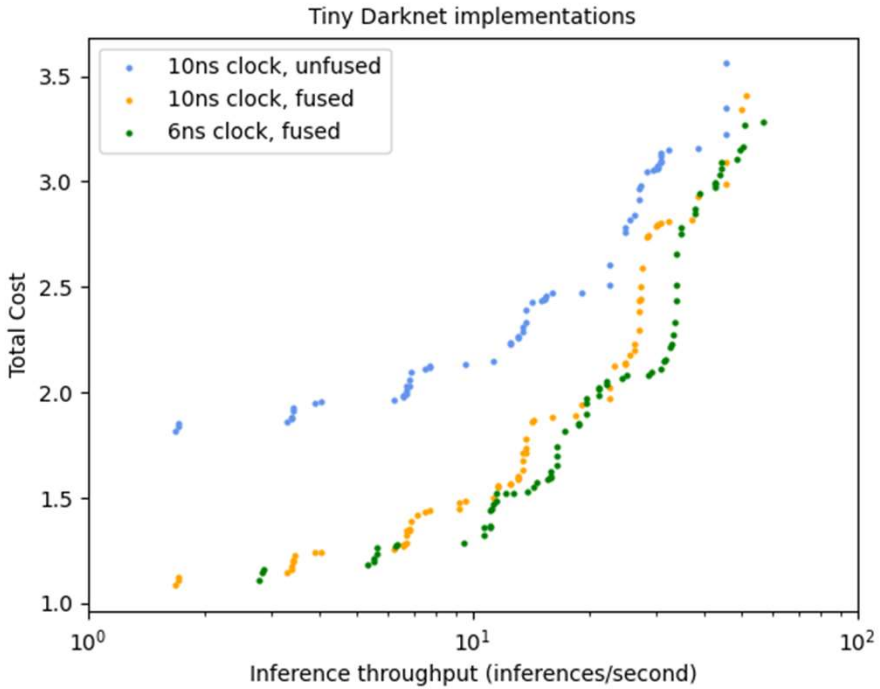
Layer latencies for four different implementations of Tiny Darknet:
(1) slowest possible, (4) fastest possible, (2,3) in the middle

Layer Selection Algorithm

- Similar to reducing the critical path of a circuit
- Ensures additional resources are spent in the right places to improve throughput
- Maximizes resource efficiency by balancing stage latencies

Network Results

Network Results



Setup

- Tiny Darknet (3 configs)
- Smallest Virtex Ultrascale+ FPGA (XCVU3P)

Future Work

- Actually implement and run on physical FPGA
  - Integrate in higher-level coprocessing system
  - Compare performance/accuracy/power to other works
- Support more layer types / networks

Questions?