Lecture: Deep Compression Accelerators

- Topics: EIE accelerator, Cnvlutin, SCNN
- Project ideas on google doc

NFU Operation Data Structures

All weights handled by green PE

Weights skipped in that column

Virtual Weight	$W_{0,0}$	$W_{8,0}$	$W_{12,0}$	$W_{4,1}$	$W_{0,2}$	$W_{12,2}$	$W_{0,4}$	$W_{4,4}$	$W_{0,5}$	$W_{12,5}$	$W_{0,6}$	$W_{8,7}$	$W_{12,7}$
Relative Row Index	0	1	0	1	0	2	0	0	0	2	0	2	0
Column Pointer	0	3	4	6	6	8	10	11	13				

Start of each column

$$\vec{a} \begin{pmatrix} 0 & 0 & a_2 & 0 & a_4 & a_5 & 0 & \end{pmatrix} \times$$

$$\begin{array}{l} PE0 \\ PE1 \\ PE2 \\ PE3 \end{array} \begin{pmatrix} w_{0,0} & 0 & \mathbf{w_{0,2}} & 0 & w_{0,4} & w_{0,5} & w_{0,6} & 0 \\ 0 & w_{1,1} & \mathbf{0} & w_{1,3} & 0 & 0 & w_{1,6} & 0 \\ 0 & 0 & \mathbf{w_{2,2}} & 0 & w_{2,4} & 0 & 0 & w_{2,7} \\ 0 & w_{3,1} & \mathbf{0} & 0 & 0 & w_{0,5} & 0 & 0 \\ 0 & w_{4,1} & \mathbf{0} & 0 & w_{4,4} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0} & w_{5,4} & 0 & 0 & 0 & w_{5,7} \\ 0 & 0 & \mathbf{0} & 0 & w_{6,4} & 0 & w_{6,6} & 0 \\ w_{7,0} & 0 & \mathbf{0} & w_{7,4} & 0 & 0 & w_{7,7} & 0 \\ w_{8,0} & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & w_{8,7} \\ w_{9,0} & 0 & \mathbf{0} & 0 & 0 & 0 & w_{9,6} & w_{9,7} \\ 0 & 0 & \mathbf{0} & 0 & w_{10,4} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{w_{11,2}} & 0 & 0 & 0 & 0 & w_{11,7} \\ w_{12,0} & 0 & \mathbf{w_{12,2}} & 0 & 0 & w_{12,5} & 0 & w_{12,7} \\ w_{13,0} & w_{13,2} & \mathbf{0} & 0 & 0 & 0 & w_{13,6} & 0 \\ 0 & 0 & \mathbf{w_{14,2}} & w_{14,3} & w_{14,4} & w_{14,5} & 0 & 0 \\ 0 & 0 & \mathbf{w_{15,2}} & w_{15,3} & 0 & w_{15,5} & 0 & 0 \end{pmatrix} \quad \boldsymbol{b} \begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \\ -b_8 \\ -b_9 \\ b_{10} \\ -b_{11} \\ -b_{12} \\ b_{13} \\ b_{14} \\ -b_{15} \end{pmatrix} \overset{ReLU}{\Rightarrow} \begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \\ 0 \\ 0 \\ b_{10} \\ 0 \\ 0 \\ b_{13} \\ b_{14} \\ 0 \end{pmatrix}$$

EIE Details

- Each PE handles a subset of neurons (data layout determines broadcast vs. reduction)
- Non-zero activations are broadcast along with their id
- Based on that id, the column pointer array (32KB SRAM) is accessed
- Accordingly, the weight and displacement are accessed – Sparse Matrix Unit that stores 4-bits for weight and 4-bits for displacement look-ups – this is an SRAM array that is 64 bits wide (locality) and 128KB in size
- One codebook look-up required (Quantization, but no Huffman Encoding)
- Each multiplication operation is added to the partial sum for that neuron (stored in a 64-entry destination activation register, spills into a 2KB activation SRAM, batching to reduce SRAM reads/writes)
- Activation queue in each PE that helps deal with sporadic load imbalance
- Central Control Unit receives leading non-zero activations from PEs and broadcasts them

EIE Details

- 4 1.15ns stages to update one activation: codebook look-up and address accumulation; output activation read and input activation multiply; shift and add; output activation write
- 16-entry queue yields ~90% efficiency
- Each PE is 9mW and 0.6 mm^2 ; SRAMs account for 59% power and 93% area
- 64 PEs at 800 MHz ➔ 102 GOP/s (mult and add)
- With 10x weight sparsity and 3x activation sparsity, amounts to 3 TOP/s
- About 1-2 orders of magnitude faster than competing CPUs and GPUs
- DDN specs for rough comparison:

  Throughput = 16 x 500 x 800 MHz = 6.4 TOP/s

  Power = 16 W (@ 600 MHz)

  Area = 68 mm^2 (38.4 mm^2 for EIE)

  Storage = 36 MB eDRAM (8MB SRAM for EIE) (EIE needs fewer chips)

EIE Take-Home

- EIE better for mobile (fewer chips, lower energy)
  DDN better for datacenter (higher throughput, SIMD)

- Prior comparison was with peak throughput; note that zero-weights are more prevalent in FC layers, not in CONV layers; so, EIE energy benefit may not be that high

- Worth pondering a hybrid arch that does DDN in CONV layers (for high thruput) and EIE in FC layers (for low energy)

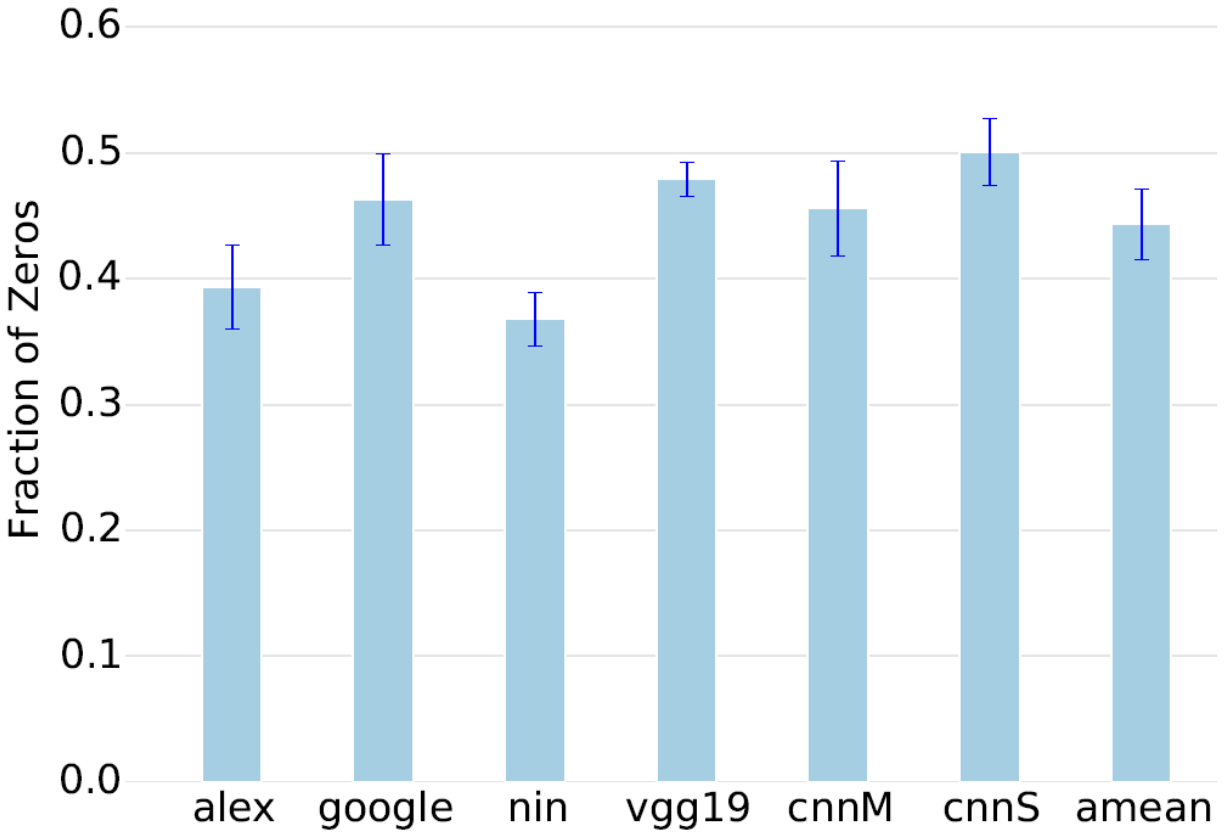
Cnvlutin

A few take-homes from the EIE discussion. First, note that they didn't do Huffman encoding – how much more complexity does that add to each PE? EIE's throughput is relatively low, but it could easily be extended in one of two ways. One is to add more ports per buffer in each PE so we could work on multiple ops simultaneously. The other is to add more PEs and do more neurons in parallel, but perhaps do this with lower storage per PE. Analyzing this trade-off space could be a decent project.

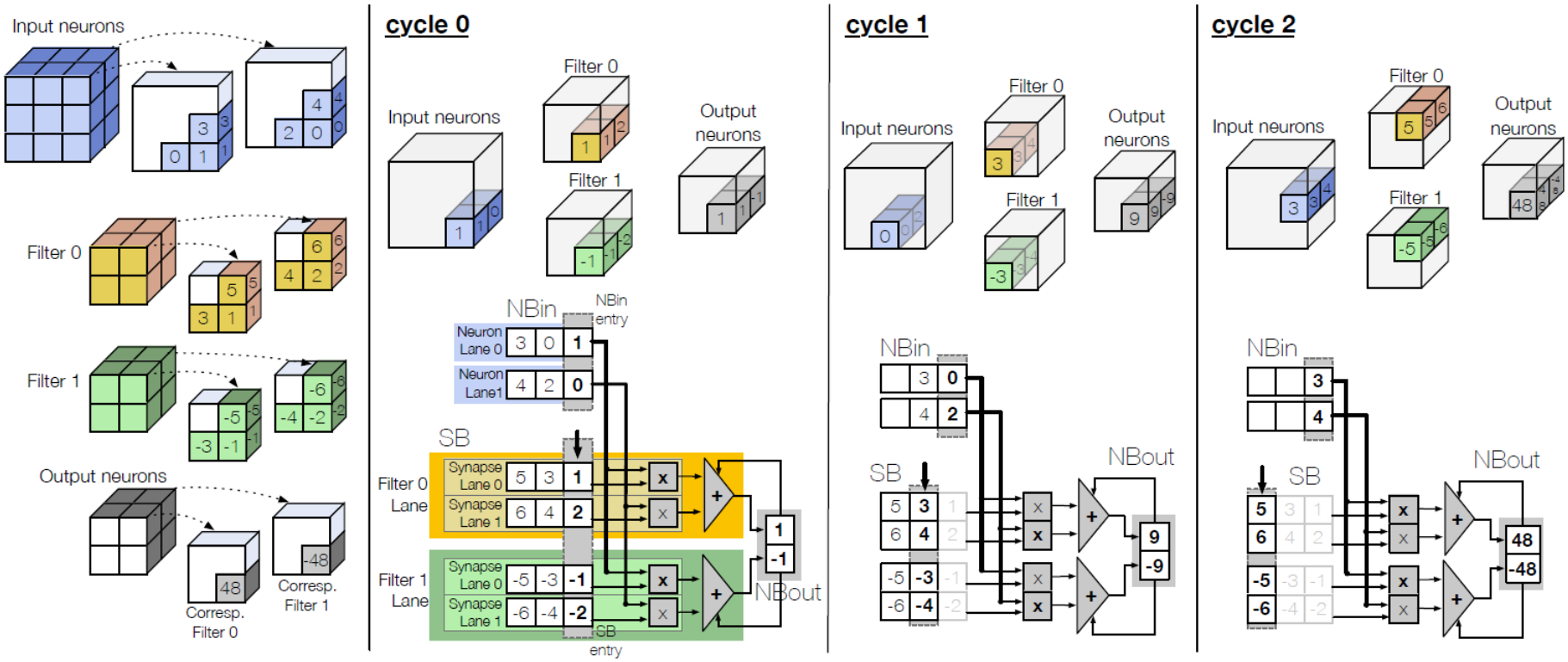
Recall that EIE compressed the weights and then also avoided computations when input values are zero. Managing sparse weights required non-trivial complexity, and also cost us the ability to do SIMD (but it was able to compress a large network into a single chip). Cnvlutin takes a different approach – it only focuses on zero input values and this is integrated into DaDianNao, so they continue to get high throughput. But unlike other works in this space, the improvement is only 1.5x and this has no impact on being able to place large networks on a single chip. On the next slide, you can see that nearly half of all neuron outputs are zero (thanks to the ReLU activation function), so this is a significant opportunity.

Cnvlutin

- Only targets sparsity in activations

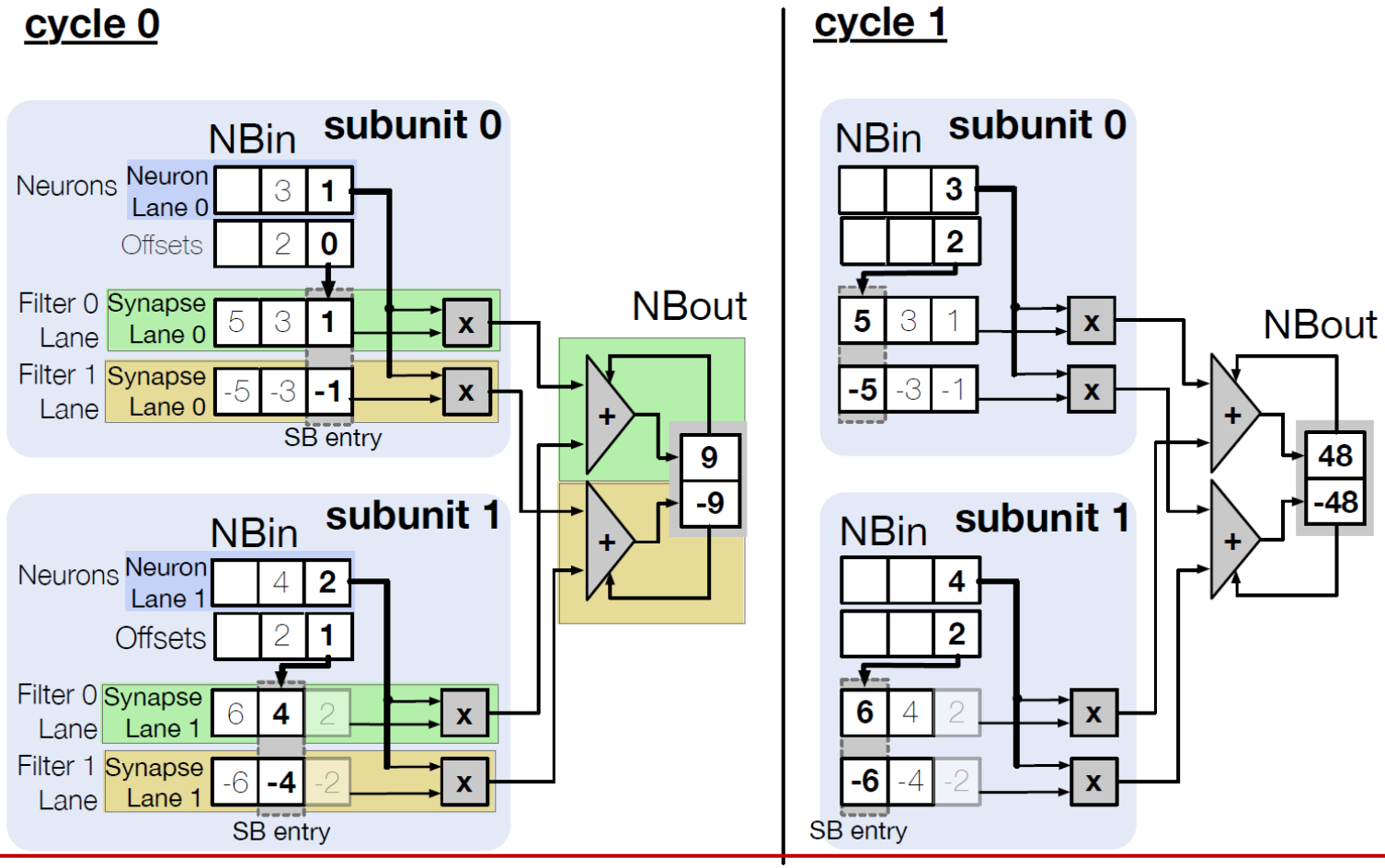


Baseline Example



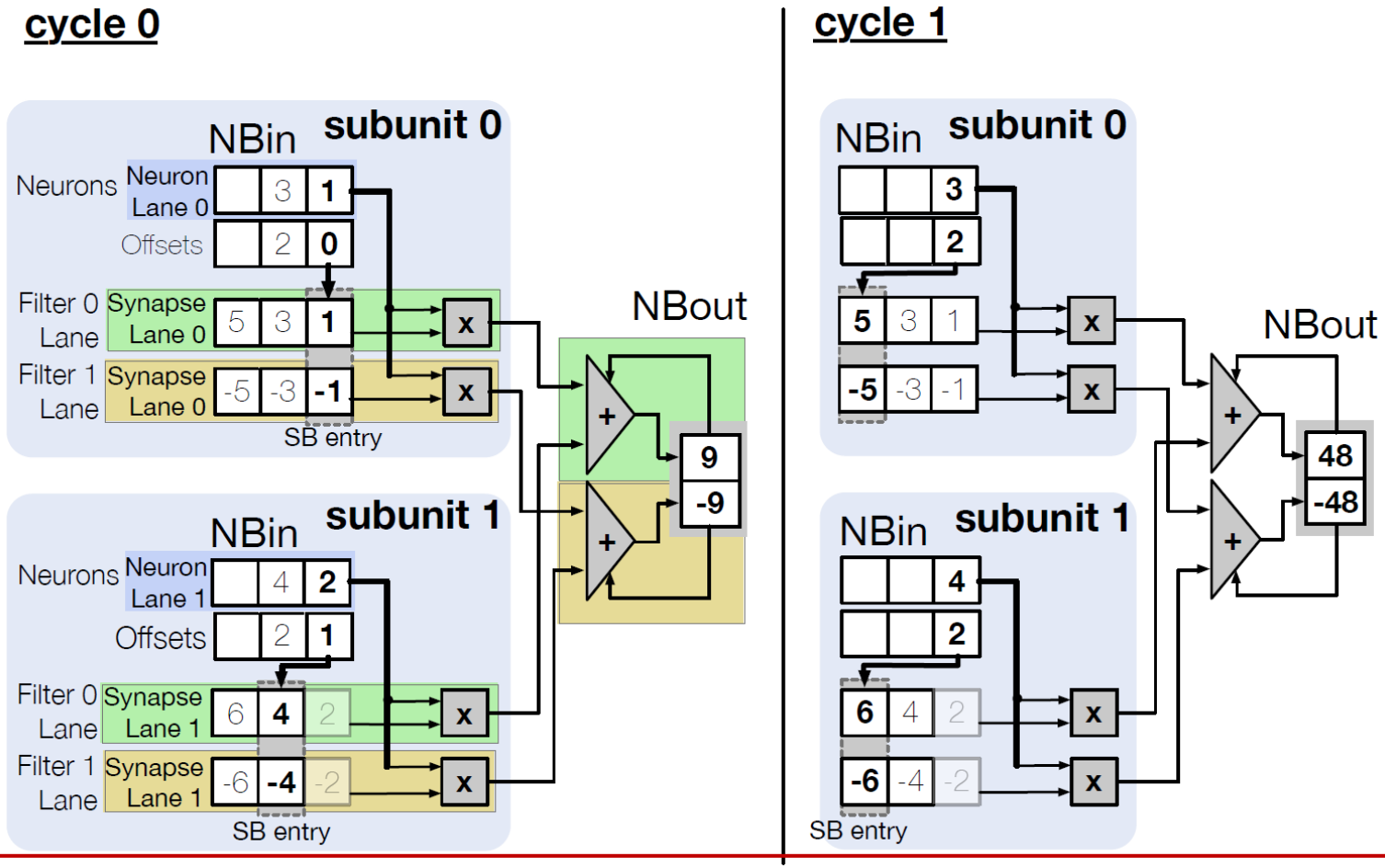
This figure lays out an example convolution in DaDianNao. The blue input neurons are being convolved with the green and orange filters. DaDianNao has 16 neuron lanes, where each neuron is dealing with one filter (lane 0 deals with the orange filter and lane 1 deals with the green filter). Within each filter, all 16 inputs are processed in lock-step in synapse lanes 0-15. If any of the inputs, let's say input 1 is zero, synapse lane 1 goes unused in all filter lanes. As shown on the next slide, Cnvlutin uses the same overall DaDianNao organization, but reshuffles the lanes and wiring to exploit zero inputs.

Cnvlutin Example



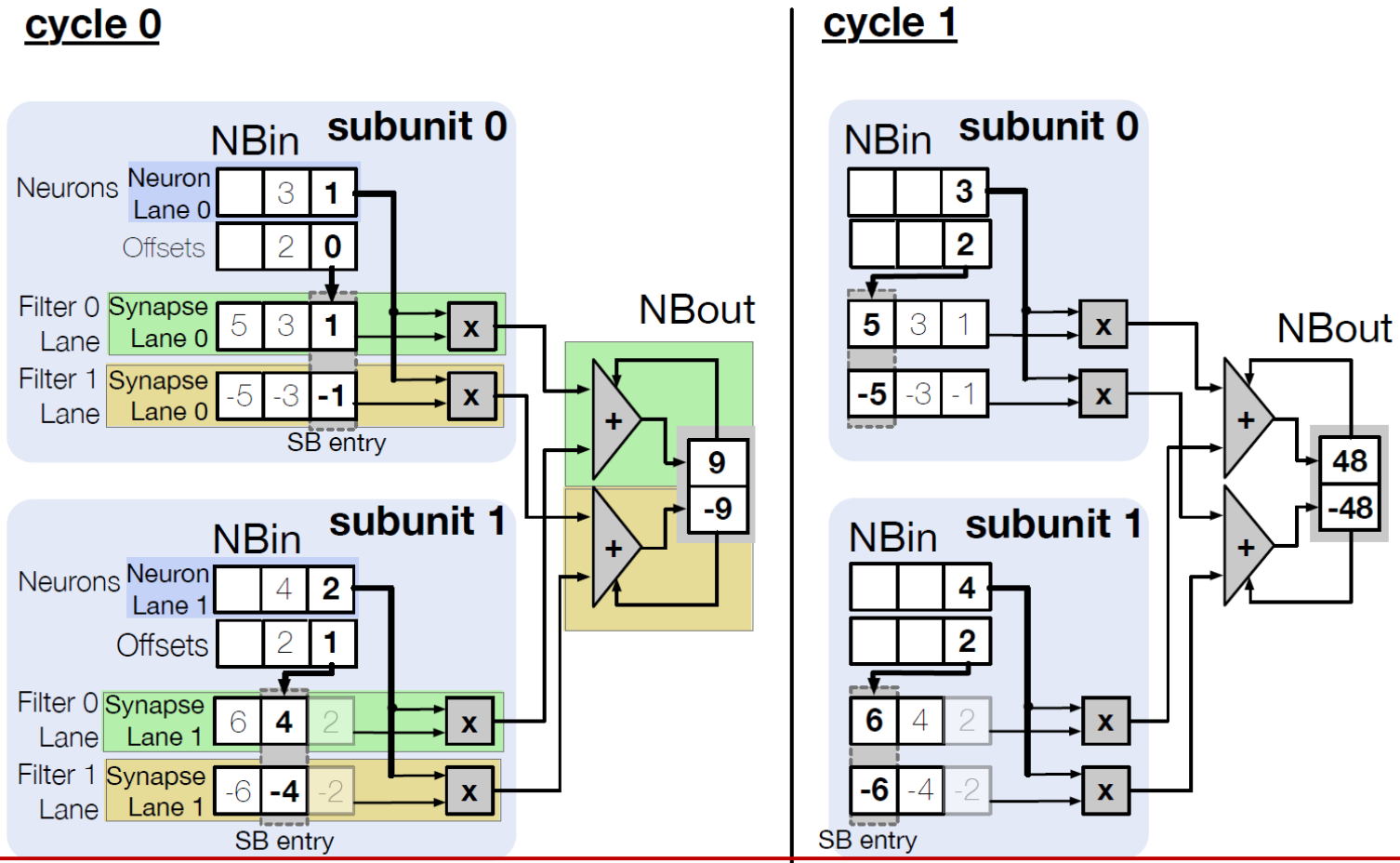
We see here that all the synapse lane 0s (from all the filters) are aggregated in subunit 0 because they all share the same input value. If the input value is zero, they skip ahead and move on to the next input. All we care about is that the output of each lane be added to the correct neuron. So all the green lanes are eventually feeding the green tree of adders.

Cnvlutin Example



So the input to each subunit is a stream of non-zero inputs and an offset field that indicates how many zeroes we skipped over (it's actually the index of the non-zero value). This field is used to figure out which synapse should be used for this computation. To keep the overhead low, the offset is fixed to 4 bits. So we're essentially dealing with 16-entry batches at a time.

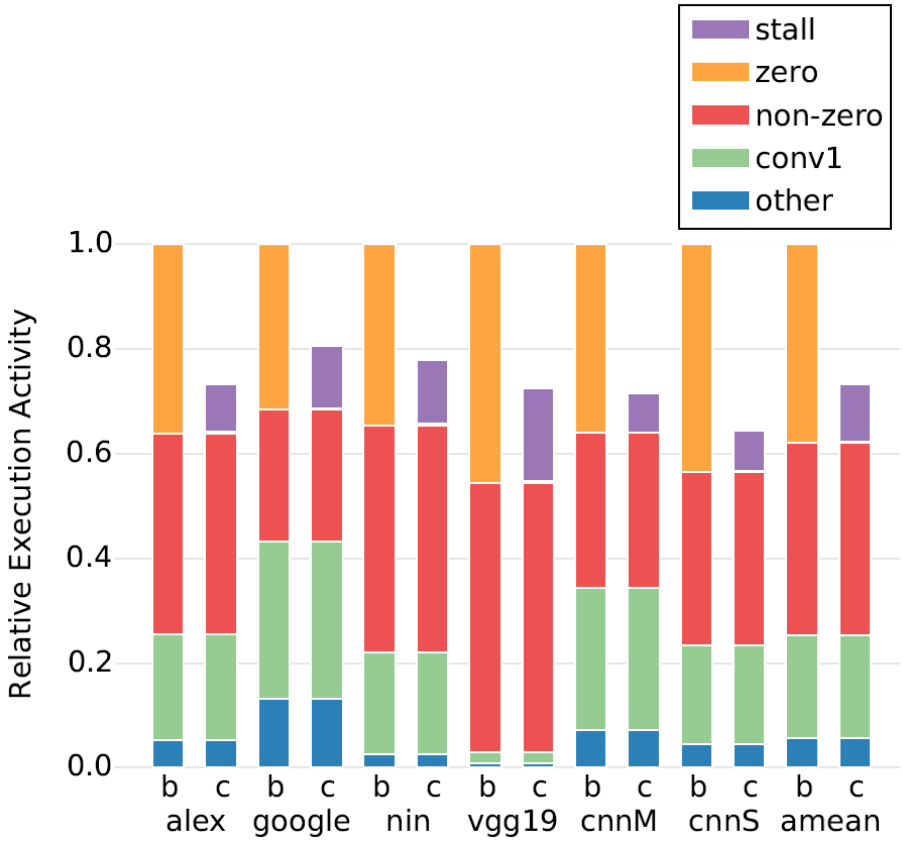
Cnvlutin Example

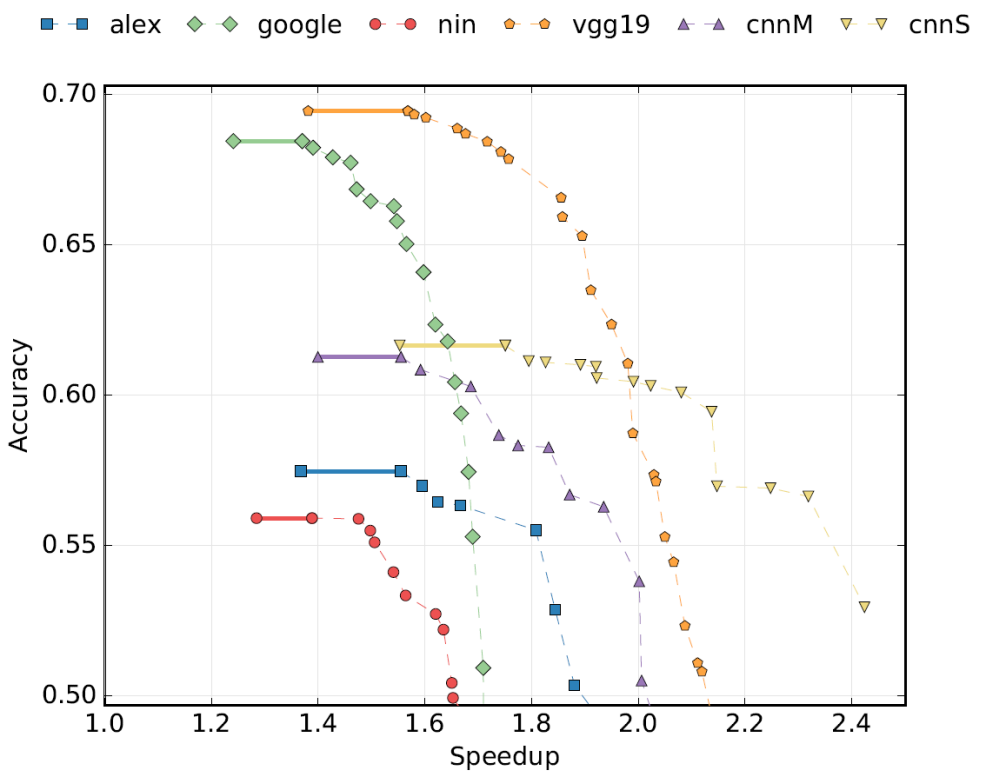


In that batch of 16, some subunits may finish their work in 6 cycles, some may take as long as 11 cycles, i.e., there will inevitably be some load imbalance. We will be limited by the worst-case, so in the above example, we'll move on to the next batch of inputs after 11 cycles. In the baseline, we would have taken 16 cycles to execute the same batch of inputs. Hence, we see about 1.37x speedup.

Cnvlutin Results

- Inputs are organized as bricks of 16 entries, requiring a 4-bit offset (25% overhead or 1MB) (4.5% area overhead for the entire chip)
- 7% lower power than the baseline (lower activity to read SB)
- Speedup: 1.37x (1.52x if we can prune neurons sufficiently close to zero)





Summary

- Relatively easy to eliminate zero neurons from a data-parallel accelerator (Cnvlutin) – relatively small speedups

- Some data-parallelism is lost when eliminating zero synapses (EIE); but the benefit of dealing with lower synaptic storage is worth it (the workload fits on a chip)

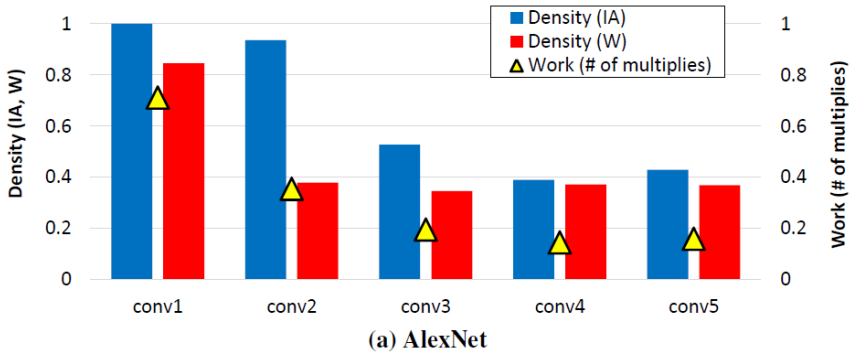
SCNN

There have been a number of accelerators that try to exploit sparsity. This ISCA'17 paper appears to be the gold standard now. It's also a very well-written paper. Unlike most prior work, SCNN targets sparsity in both weights and activations. The graphs in the next slide quantify the degree of sparsity in weights/activations and the number of saved multiplications we can expect, on a per-layer basis. The early layers have little sparsity, but as we get deeper, about 80% of the multiplications can be eliminated.

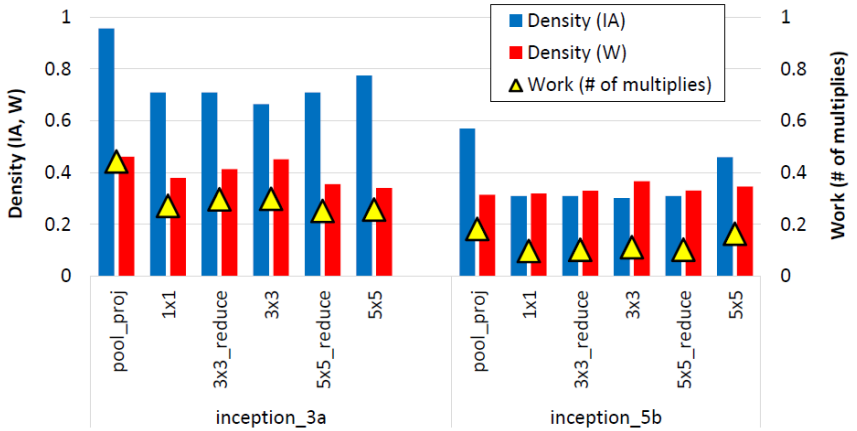
SCNN

- First work to target sparsity in activations and weights

(Cambricon-X and EIE do sparse weights; Cnvlutin and Eyeriss do sparse activations)



(a) AlexNet



(b) GoogLeNet

Loop Nesting

This is a helpful way to look at all the required computations. This is a 7-deep nested loop. The problem here is to find the most efficient way to complete this work. This may be done by re-ordering the loops, tiling the loops, splitting the loops across parallel units, etc.

```
for n = 1 to N          Do all images
  for k = 1 to K        Do all output feature maps
    for c = 1 to C      Do all input feature maps
      for w = 1 to W    Do all horizontal pixels in output fmap
        for h = 1 to H  Do all vertical pixels in output fmap
          for r = 1 to R    Do all horizontal pixels in kernel
            for s = 1 to S  Do all vertical pixels in kernel
              out[n][k][w][h] +=
                in[n][c][w+r-1][h+s-1] *
                filter[k][c][r][s];
```

Figure 3: 7-dimensional CNN loop nest.

Short-hand: $N \rightarrow K \rightarrow C \rightarrow W \rightarrow H \rightarrow R \rightarrow S$

SCNN Approach

After trying a number of options, they settle on the loop order specified on the last slide. The also assume no batching of images (N=1); note that Google and Microsoft both use heavy batching in their datacenters.

- For any architecture, must find the right way to order these loops and distribute the loops across PEs, such that reuse is maximized and data movement is minimized
- After experimentation, they pick a nesting/dataflow that is best for exploiting sparsity

  (They assume N=1, which may be questionable)
- They observe that convs are bigger bottlenecks than FC; they observe that feature map data movement is the biggest bottleneck; hence input-stationary

  (actually, a combo of input-stationary and row-stationary)

The rationale for their loop order is that input feature maps in conv layers are a much bigger bottleneck. So once we've fetched a pixel value, it must be reused as much as possible without additional data movement. This leads to an "input-stationary" dataflow. As we'll see shortly, they also make sure that a row of weights is heavily reused – similar to Eyeriss row-stationary.

SCNN Dataflow

The overall SCNN architecture is shown on the next slide. Let's first describe the dataflow without regard to sparsity. The computational unit has a grid of multipliers. It ends up being a 4x4 grid of 16-bit multipliers. In every step, we provide I inputs and F weights. All pair-wise multiplications are performed.

- Each PE has a matrix of multipliers
- This matrix is fed with I inputs and F filter weights
- The matrix performs a *Cartesian Product* – every I is multiplied with every F
- These results are then fed to an array of adders that keep aggregating partial sums – need a router and an index generator to make sure each product is sent to the right adder (multiple pixels may be buffered at adder)
- The input stays and a new vector of weights is provided
- Once an output feature map entry is computed, it is written to the output buffer
- Then they bring in the next set of input feature map values
- Lots of PEs; each handles part of the input feature map; aggregation required for boundary elements (halos)

SCNN Dataflow

The products must be added to corresponding output neuron values. With simple calculations, we can figure out which output neuron. The result is sent through a routed network to that (24-bit) accumulator. The accumulator array has buffers to keep track of partial sums.

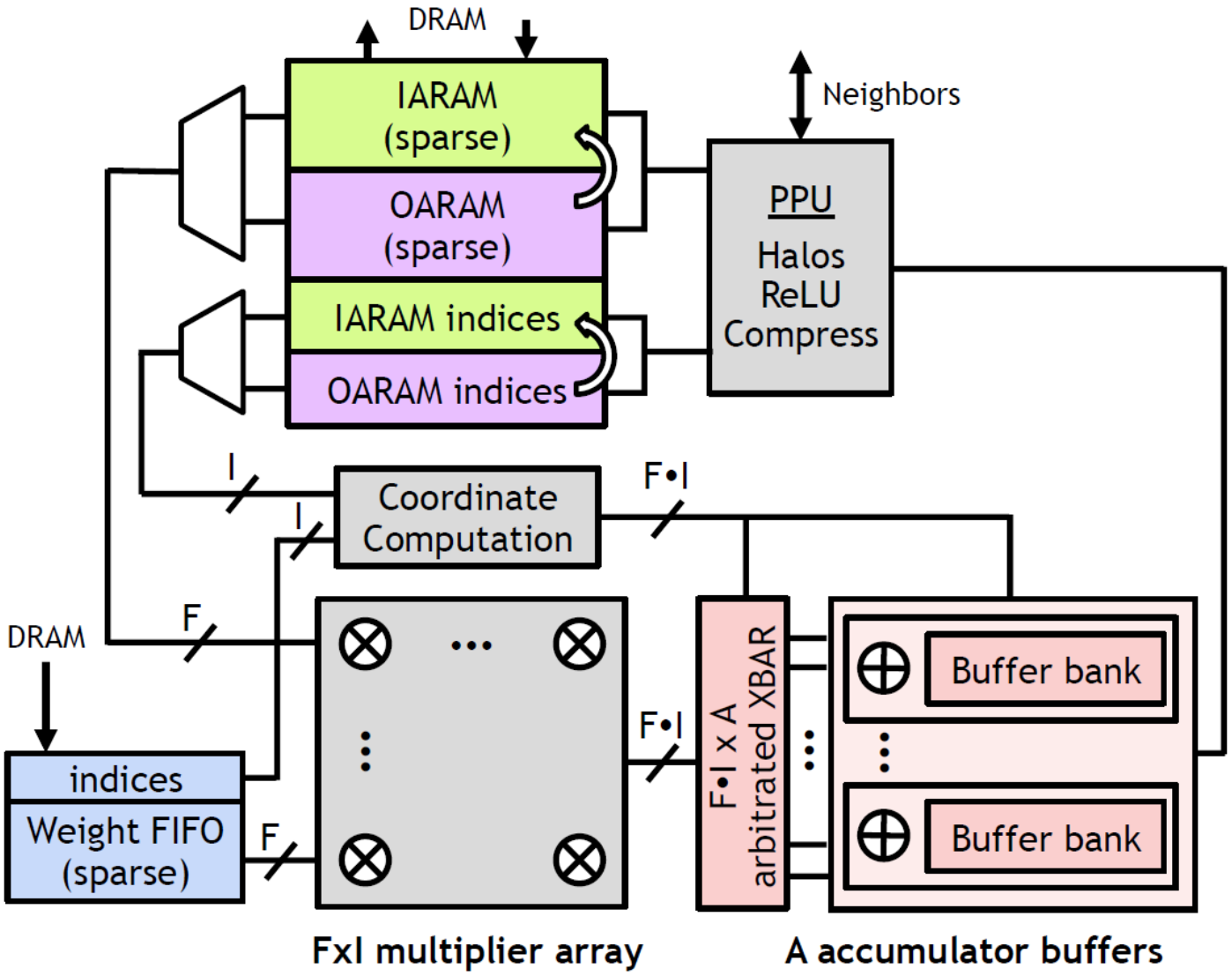
- Each PE has a matrix of multipliers
- This matrix is fed with I inputs and F filter weights
- The matrix performs a *Cartesian Product* – every I is multiplied with every F
- These results are then fed to an array of adders that keep aggregating partial sums – need a router and an index generator to make sure each product is sent to the right adder (multiple pixels may be buffered at adder)
- The input stays and a new vector of weights is provided
- Once an output feature map entry is computed, it is written to the output buffer
- Then they bring in the next set of input feature map values
- Lots of PEs; each handles part of the input feature map; aggregation required for boundary elements (halos)

SCNN Dataflow

The same inputs are used multiple times (with different weight vectors) to maximize reuse. The chip has 64 such PEs. Each PE is asked to work on part of the feature map. Some aggregations will be required across PEs.

- Each PE has a matrix of multipliers
- This matrix is fed with I inputs and F filter weights
- The matrix performs a *Cartesian Product* – every I is multiplied with every F
- These results are then fed to an array of adders that keep aggregating partial sums – need a router and an index generator to make sure each product is sent to the right adder (multiple pixels may be buffered at adder)
- The input stays and a new vector of weights is provided
- Once an output feature map entry is computed, it is written to the output buffer
- Then they bring in the next set of input feature map values
- Lots of PEs; each handles part of the input feature map; aggregation required for boundary elements (halos)

PE Design


DRAM
IARAM (sparse)
OARAM (sparse)
IARAM indices
OARAM indices
Neighbors
PPU
Halos
ReLU
Compress
I
I
Coordinate Computation
F•I
DRAM
F
indices
Weight FIFO (sparse)
F
F•I
F•I x A arbitrated XBAR
Buffer bank
Buffer bank
FxI multiplier array
A accumulator buffers

Managing Sparsity

When inputs are provided as sparse vectors, the only difference is that every element has an index. So we need some additional calculations in the mult-matrix to figure out where every product must be sent. The scatter accumulators will also see load imbalance. 32 adders are provided for the 16 mults to reduce this imbalance.

- The PE is fed with dense vectors (I and F); each element also has an index; when multiplications are performed, the index is also calculated
- The product is sent to the appropriate adder (buffered in case of conflicts); the array vector is twice as large to reduce conflicts (4x4 mult and 32 adders)
- FC layer can only use 4 of the mults in the PE
- Overheads: index generation, routed network, more scatter accumulators

Chip Details

We see that the multiplier array is a small area overhead. The accumulator buffers and the network are bigger overheads. They use a baseline (DCNN) that is best described as output-stationary. It also has optimizations (DCNN-opt) that stores activations in DRAM in compressed format and can reduce energy by gating zero-multiplications.

PE Parameter	Value
Multiplier width	16 bits
Accumulator width	24 bits
IARAM/OARAM (each)	10KB
Weight FIFO	50 entries (500 B)
Multiply array ($F \times I$)	4×4
Accumulator banks	32
Accumulator bank entries	32
SCNN Parameter	**Value**
# PEs	64
# Multipliers	1,024
IARAM + OARAM data	1MB
IARAM + OARAM indices	0.2MB

PE Component	Size	Area (mm^2)
IARAM + OARAM	20 KB	0.031
Weight FIFO	0.5 KB	0.004
Multiplier array	16 ALUs	0.008
Scatter network	16×32 crossbar	0.026
Accumulator buffers	6 KB	0.036
Other	—	0.019
Total	—	0.123
Accelerator total	64 PEs	7.9

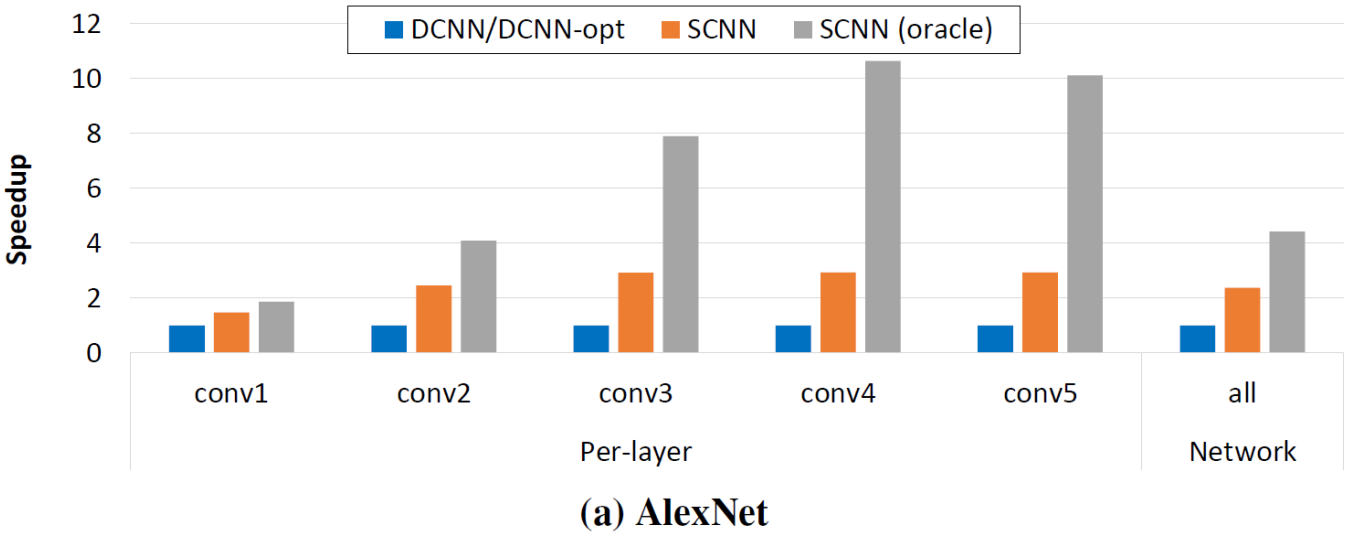
The table below shows that the accumulator buffers cause most of the area increase, relative to the baseline.

1 GHz, 2 TOps/s

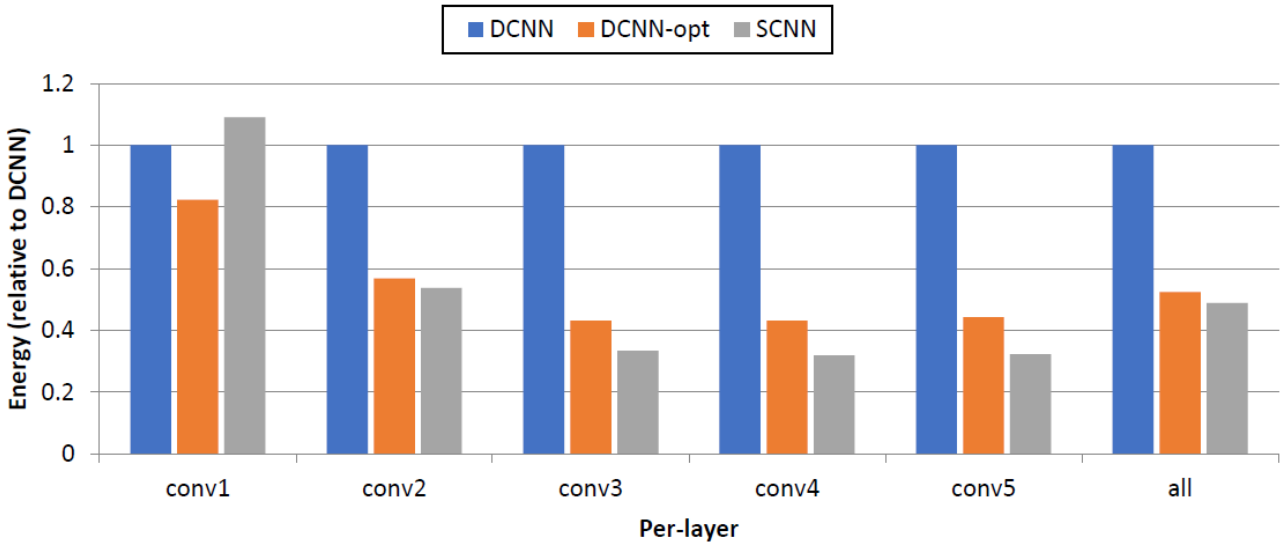
	# PEs	# MULs	SRAM	Area (mm^2)
DCNN	64	1,024	2MB	5.9
DCNN-opt	64	1,024	2MB	5.9
SCNN	64	1,024	1MB	7.9

Results

The DCNN-opt does a pretty good job on energy, but it is unable to speed things up. SCNN is able to beat the baseline by about 2.5x in both performance and energy. Could do better if all the multipliers were busy every cycle.



(a) AlexNet



(a) AlexNet

- About 2.5x speedup
- Some inefficiency because PEs are underutilized and because of load imbalance
- Turns out that DCNN-opt is very effective at reducing energy (DRAM activ. comp and power-gating for zero-mults)

References

- "Learning both Weights and Connections for Efficient Neural Network", S. Han et al., Proceedings of NIPS, 2015
- "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding", S. Han et al., ICLR, 2016
- "EIE: Efficient Inference Engine on Compressed Deep Neural Network", S. Han et al., Proceedings of ISCA, 2016
- "Deep Compression and EIE", S. Han, talk at GPU Tech Conf, 2016
- "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing", J. Albericio et al., Proceedings of ISCA, 2016
- "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks," A. Parashar et al., ISCA 2017