C-Brain: A Deep Learning Accelerator that Tames the Diversity of CNNs through Adaptive Data-level Parallelization

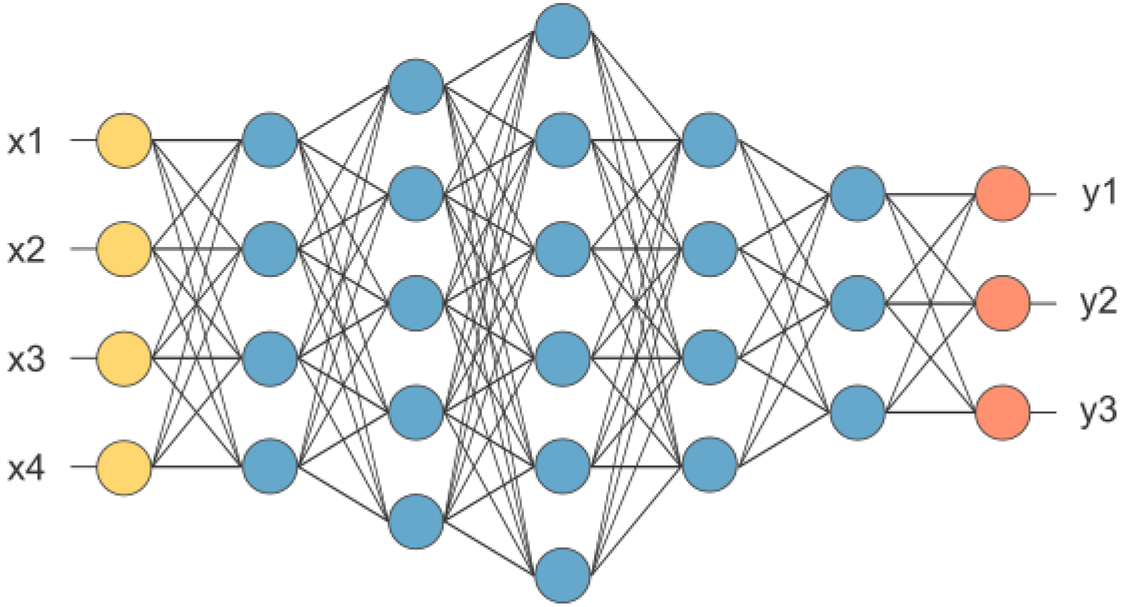
Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, Xiaowei Li State Key Laboratory of Computer Architecture Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China

Presented by:

Ryan Barton
17M38035

What we’ll cover

- What is a Convolutional Neural Network (CNN)?
- Accelerators: problem statement and paper introduction
- Data-parallelization scheme
  - Kernel-level parallelism
  - Adaptivity, regardless of NN topology & hardware
- Putting the “petal to the metal” – performance and energy evaluation

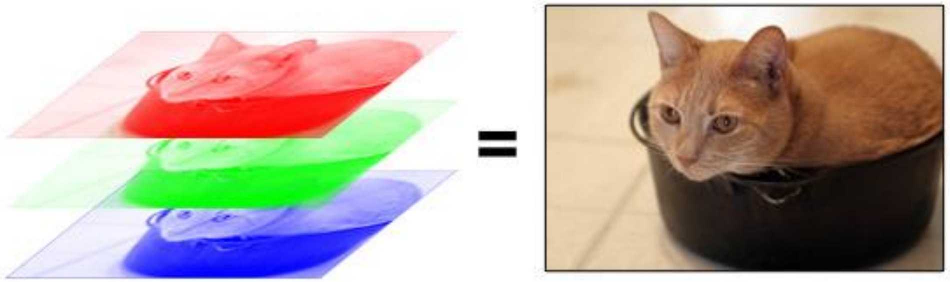


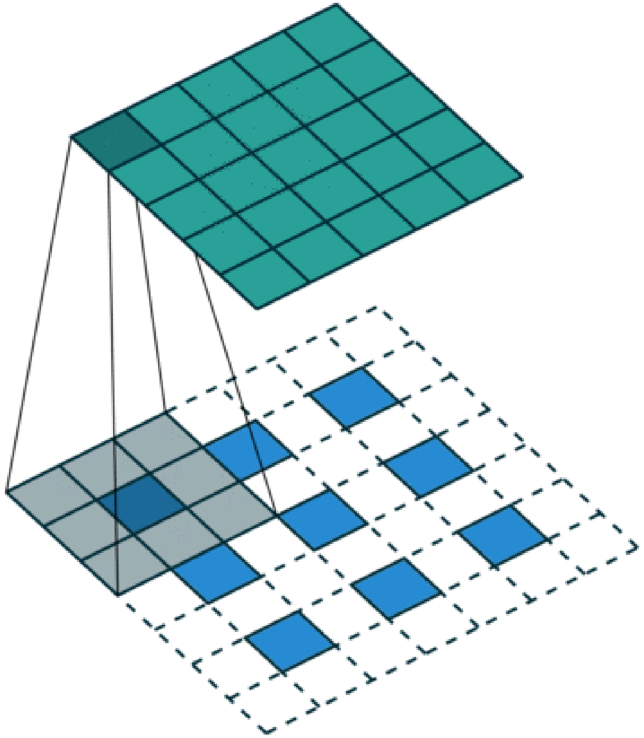
Convolutional Neural Network (CNN)

- A Deep Learning, feed-forward, neural network known for its success in image recognition (think Facebook’s tagging algorithm)
- General idea: make series of reductions of an image, analyze its fundamental properties, and arrive at a result
- 3 types of layers:
  - Convolutional layers
  - Pooling layers
  - Fully connected layers
- Our example CNN: is input an X or an O?

Convolutional layer

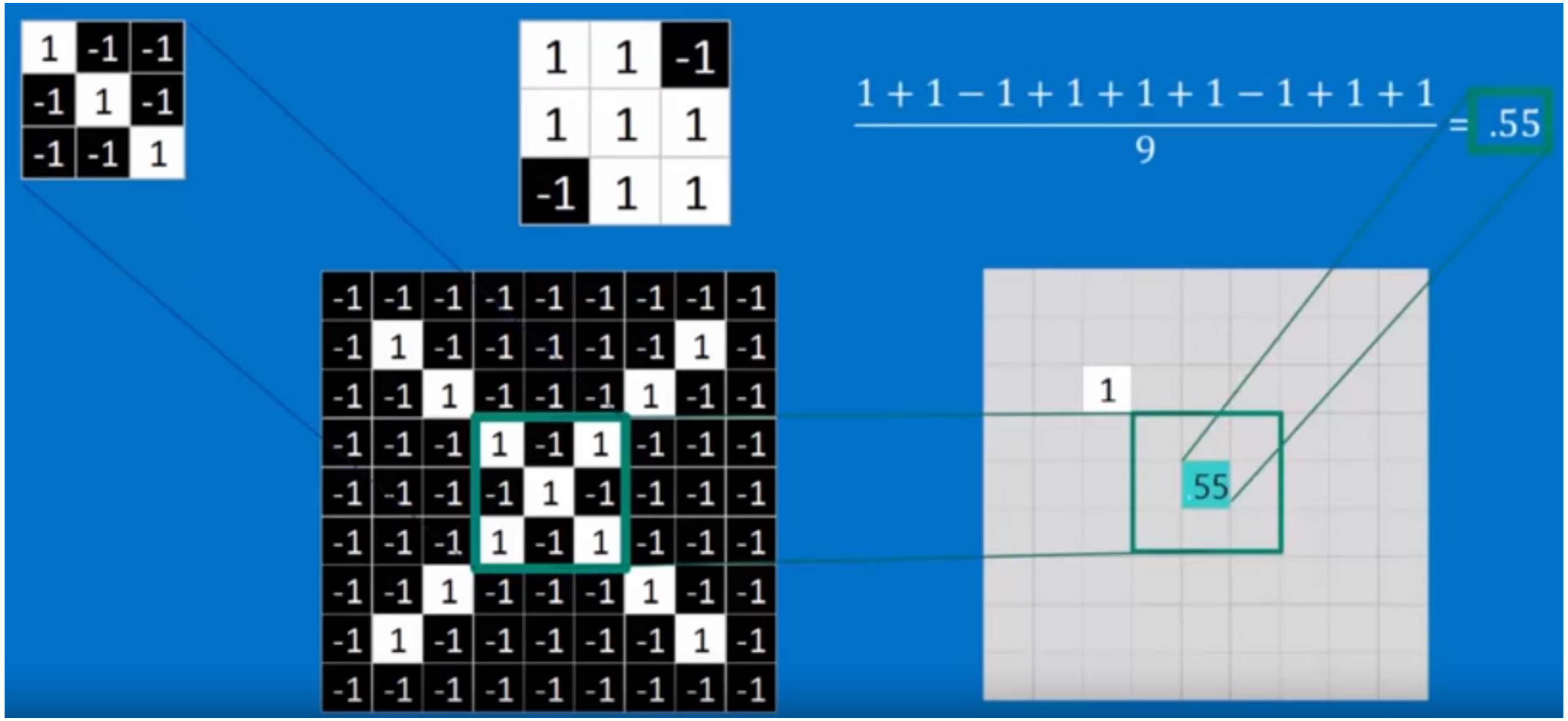
- Input: image of n x n pixels.
  - 3D stack of layers called **features** (ex. RGB, lines)
- Output: smaller image of values.
  - A **map** showing how well that feature is represented throughout original image.
  - In our example, values 0 <= c <= 1
- What is a convolution?
  - The act of sliding a **kernel** (window) k x k pixels across an image, and looking for something. Called **stride**.
  - Usually a matrix of parameters the NN is trying to learn





Convolution example

- Define feature
  - Any property of X. Let's say the top left slant.
  - White pixel = 1, black pixel -1
- In kernel, compare each pixel in feature to those in image
  - Perform dot product and divide by # of pixels in feature.

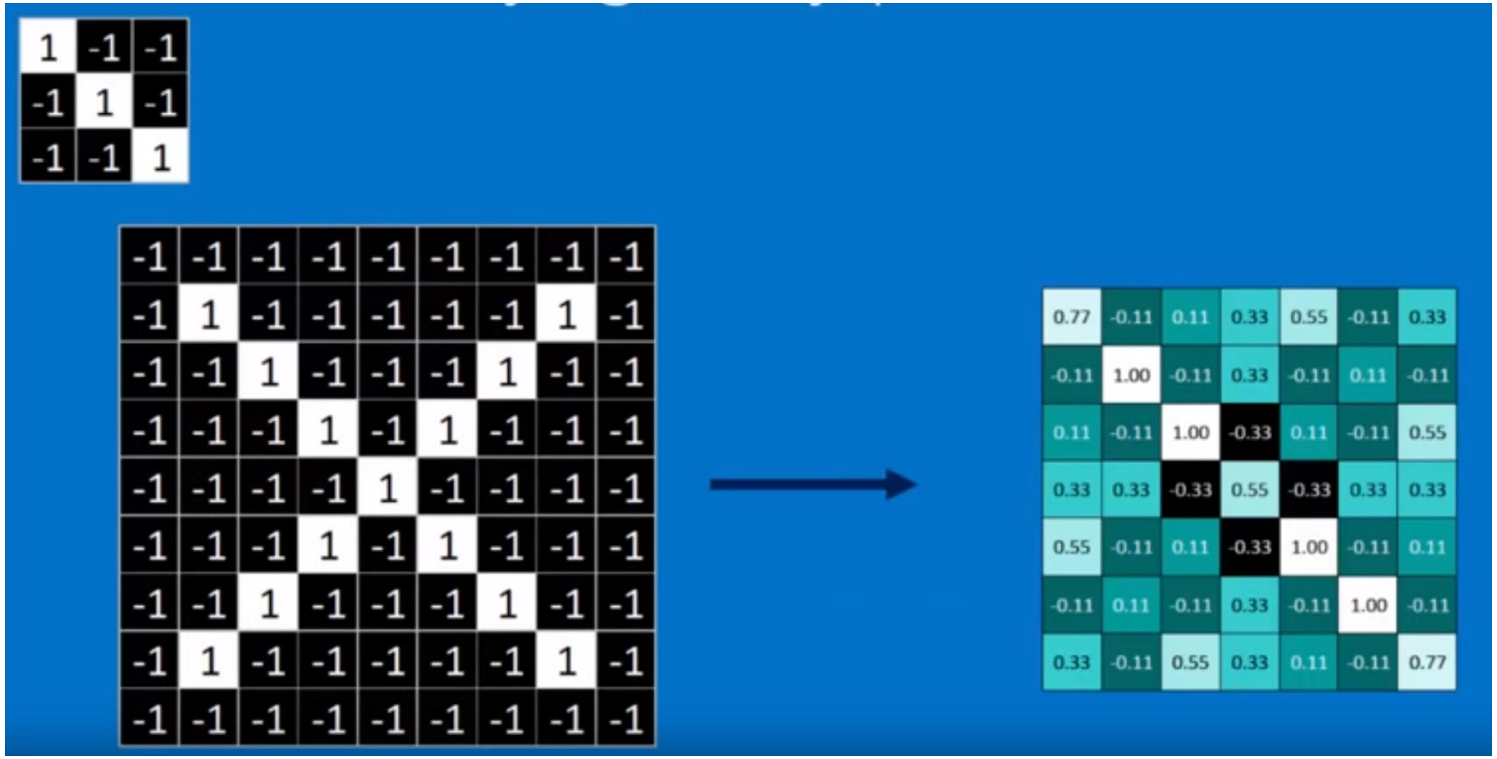


Images for this example courtesy of Brandon Rohrer

http://brohrer.github.io/how_convolutional_neural_networks_work.html

Convolution example cont.

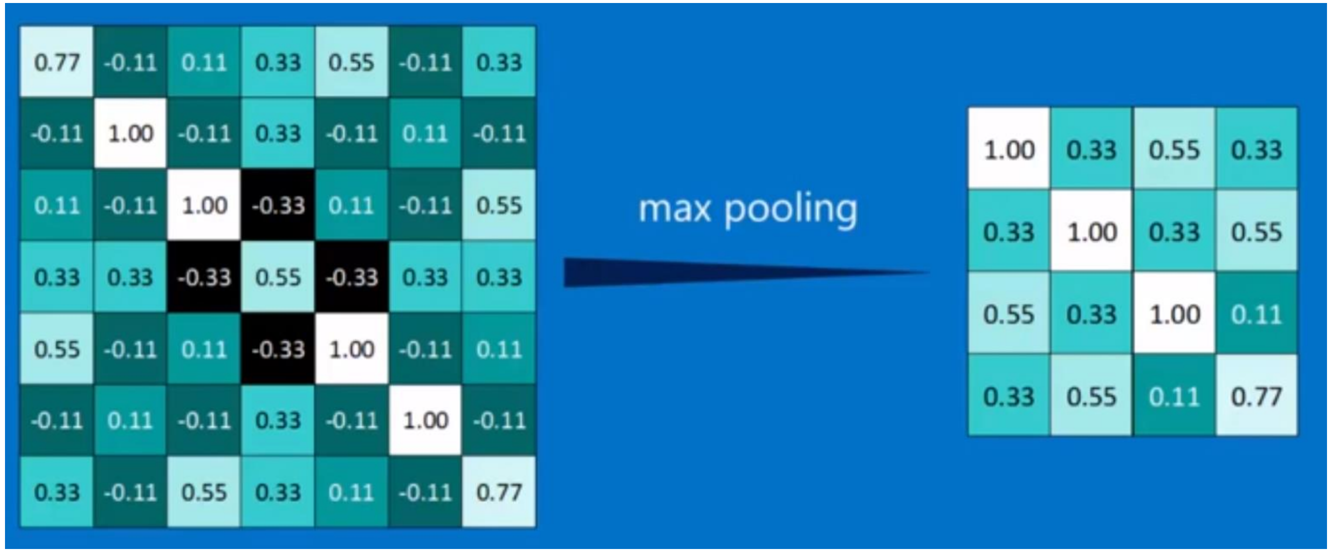
- After iterating over the entire image, below we get our feature map



*Aside: On Instagram, this is known as a Box Blur.

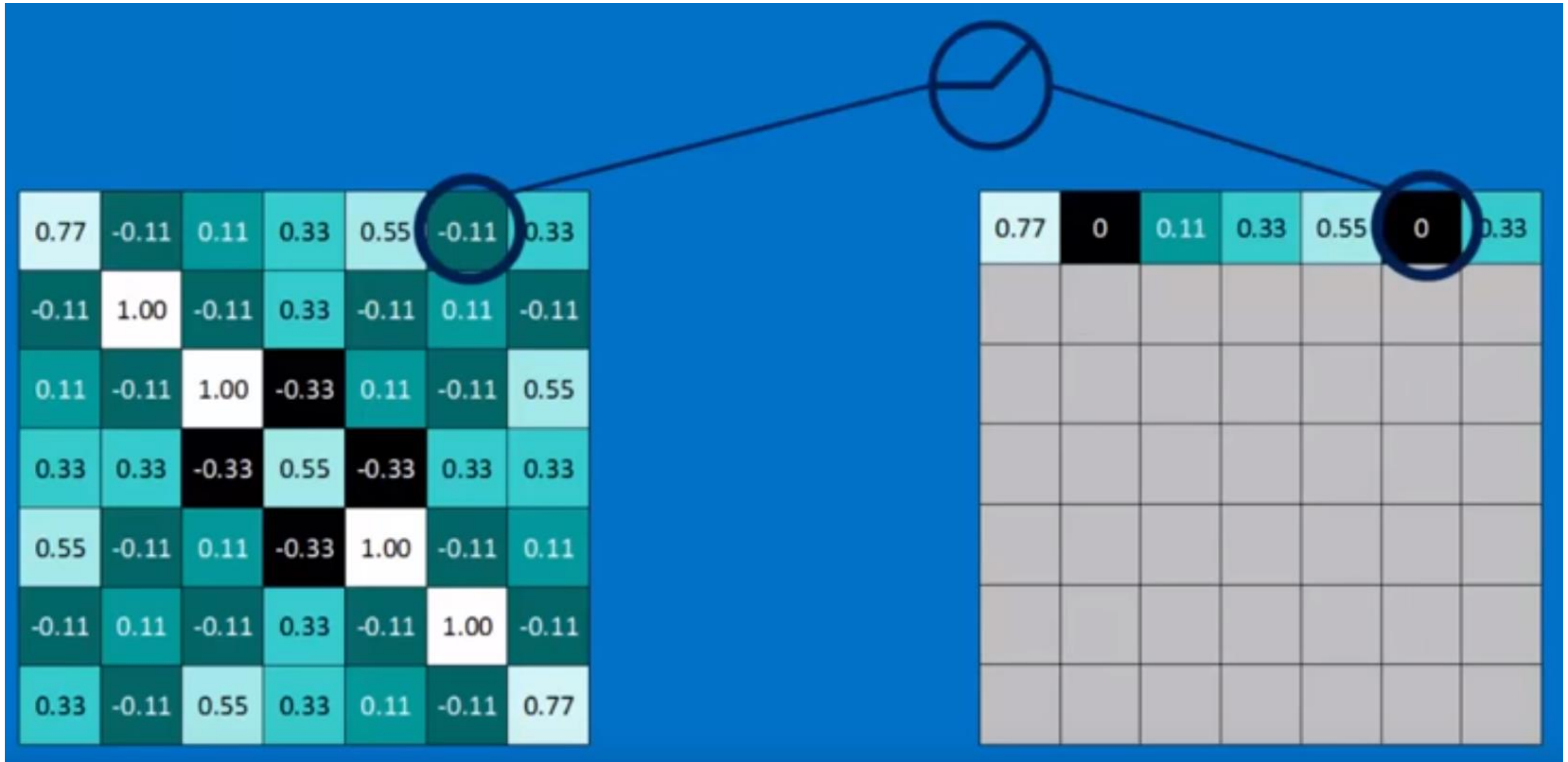
Pooling layer

- Input: convolutional layer
- Output: even smaller image containing max values of input layer
- Like convolution, pick kernel and stride
- Calculate maximum value, insert value p into new image



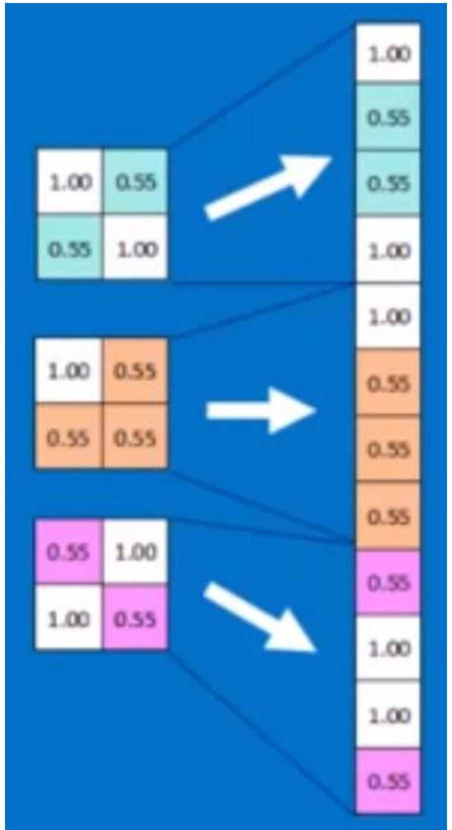
Trick: Normalization (Linear Rectified Units (ReLU))

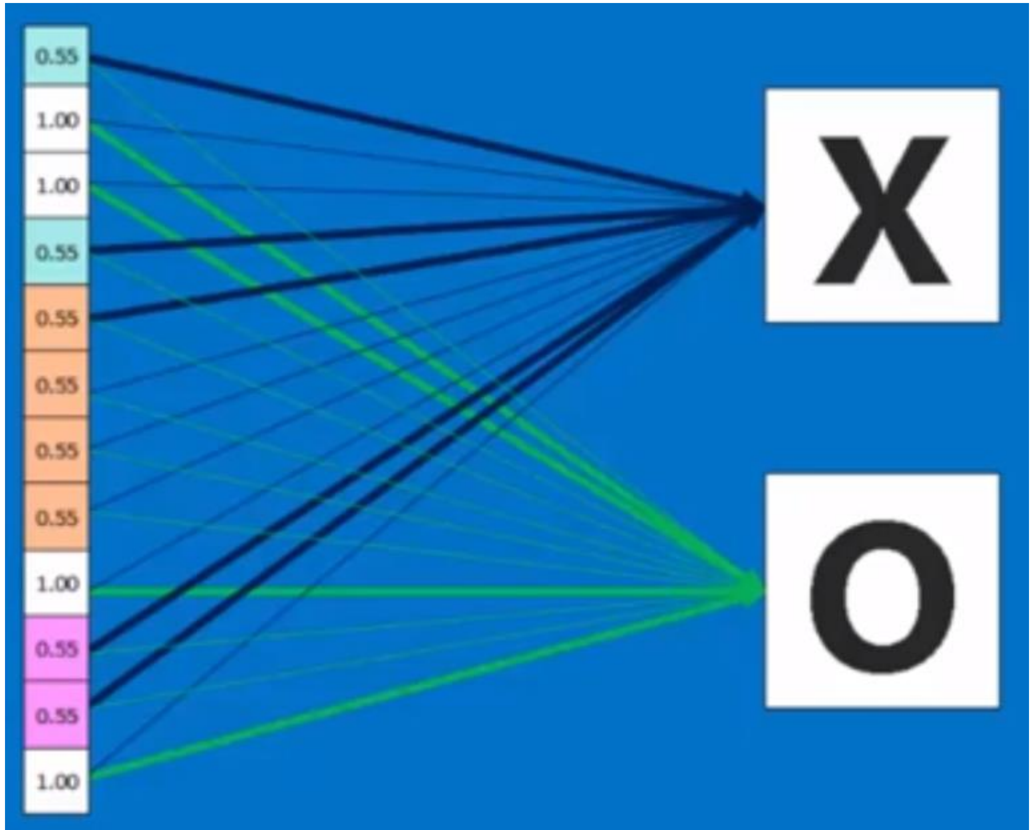
- Input: convolution layer or pooling layer
- Output: same image, with all c and p > 0.
- This keeps math consistent throughout the network



Fully Connected layer

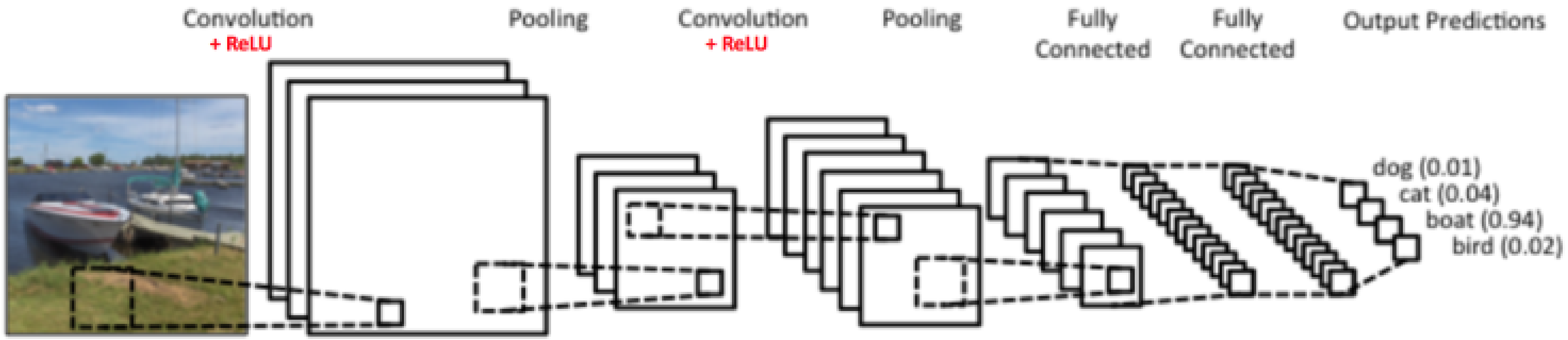
- All neurons in a layer L1 is connected to all neurons in layer L2.
- Basically, each neuron has a say in the final result (X or O?).





Bringing it all together

These layer operations can be combined in any order (generally speaking).



Back propagation works in same way as other NNs, with gradient descent.

In CNNs there are potentially many steps, so indeed they're computational beasts!

Accelerating CNNs

- How to make CNNs  faster
  - Parallelizing:
    - Output layer creation
    - Inner-kernel operations (without buffers for data re-use)
    - Memory bandwidth utilization (between layers)
    - Using special hardware (FPGAs)

  - However, these attempts consistently ignore:
    - Data reuse – too much data!
    - Network topology – too specific!
    - Hardware overreliance – too power hungry & costly!

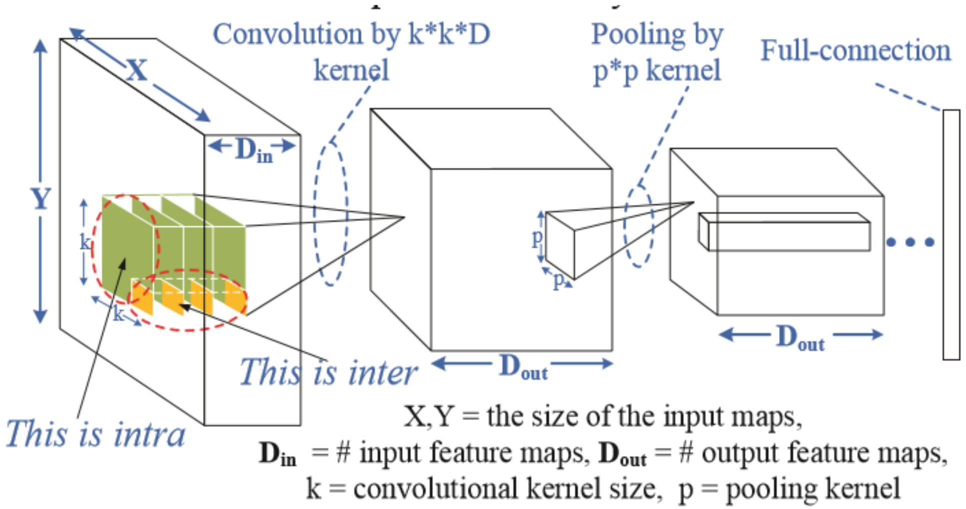
C-Brain Introduction

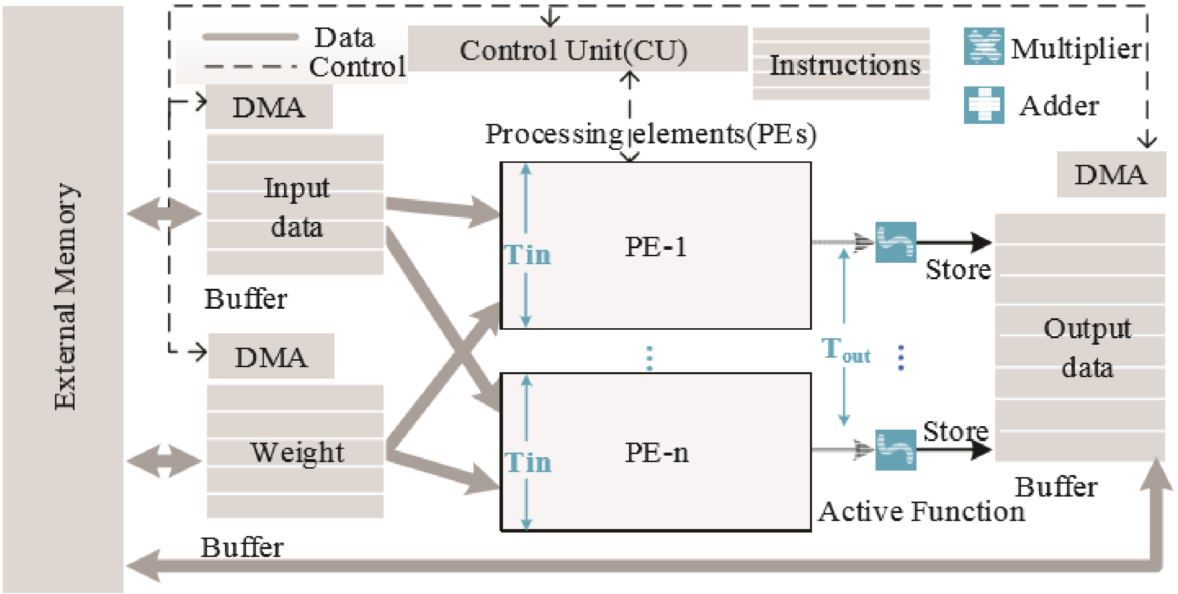
- This paper tries to solve these problems by proposing a 2-pronged **software-based** approach
  1. Kernel partitioning scheme
     - Inter- and intra- kernel parallelization, by splitting and transferring kernel data intelligently
     - Pros and cons to both styles, so a hybrid approach is desired
  2. Adaptiveness scheme
     - Generalizing inter- and intra- kernel strategy with –any – network topology or hardware

       Tested on 4 main NNs: Alexnet, GoogleNet, VGG, and NIN

Inter-kernel Parallelization

- Goal: efficiently transfer data in one kernel **k * k** across several input layers from memory to the Processing Elements (PEs)
- Result: load pairs into input buffer, compute **k * k** operations, sum them up, load number into output buffer





Inter-kernel Parallelization (2) Direct Insert

- Problem: Parallelization is limited by dimensions of $\mathbf{D_{in}}$ and $\mathbf{D_{out}}$.
- Ideal case: input map size well matches size $\mathbf{T_{in}}$

PROS: if layers can be inserted in PEs well, then super fast

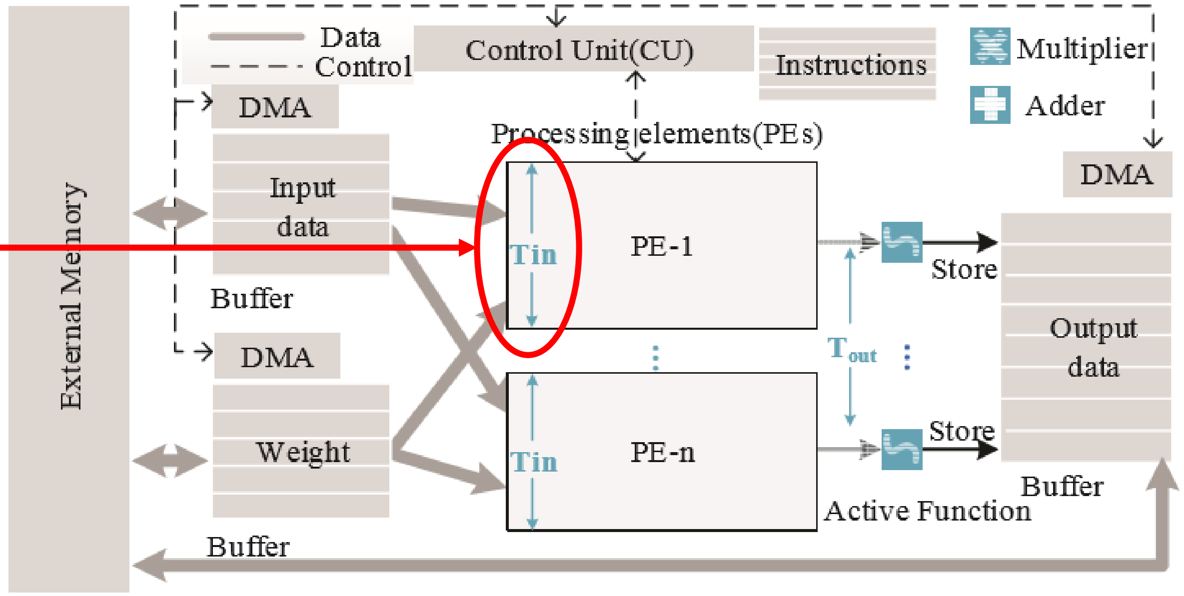
CONS: if PEs really underestimate or overestimate # of layers,
either we use too few resources, or wait unnecessarily for time on PE.

AlexNet example

C1 = 3 layers
1 layer assigned per PE

if Tin = 16, we have 16 PEs

16-3 = 13 PEs not used.
Waste of resources!



Intra-kernel Parallelization

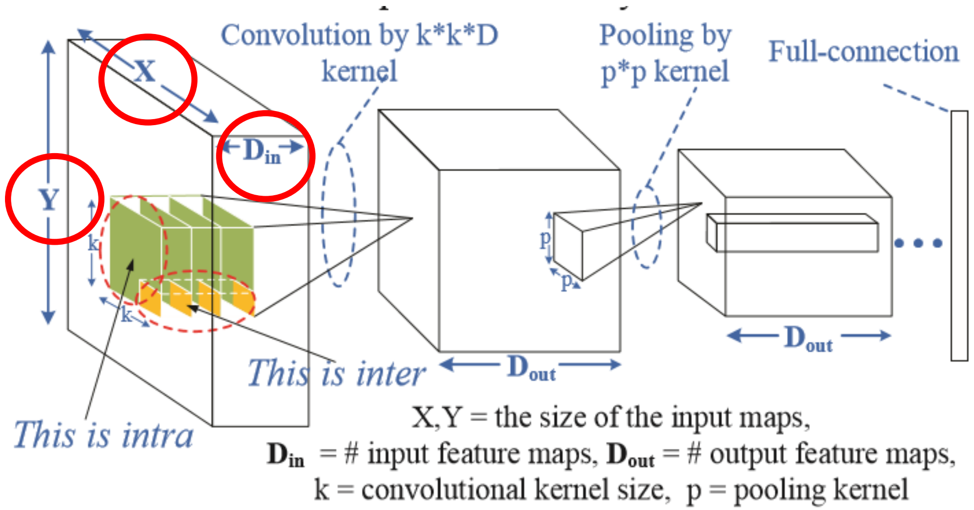
- Goal: efficiently transfer data from several kernels **k_1 k_2 … k_n** across one input layer from memory into the Pes
- In CNNs, layer size **X * Y** almost always > layer depth **D_{in}.** So intra- is more efficient than inter-

- Strategies:

  1. Data unrolling

  2. Sliding window

  3. 2D PEs



Intra-kernel (2) Data Unrolling

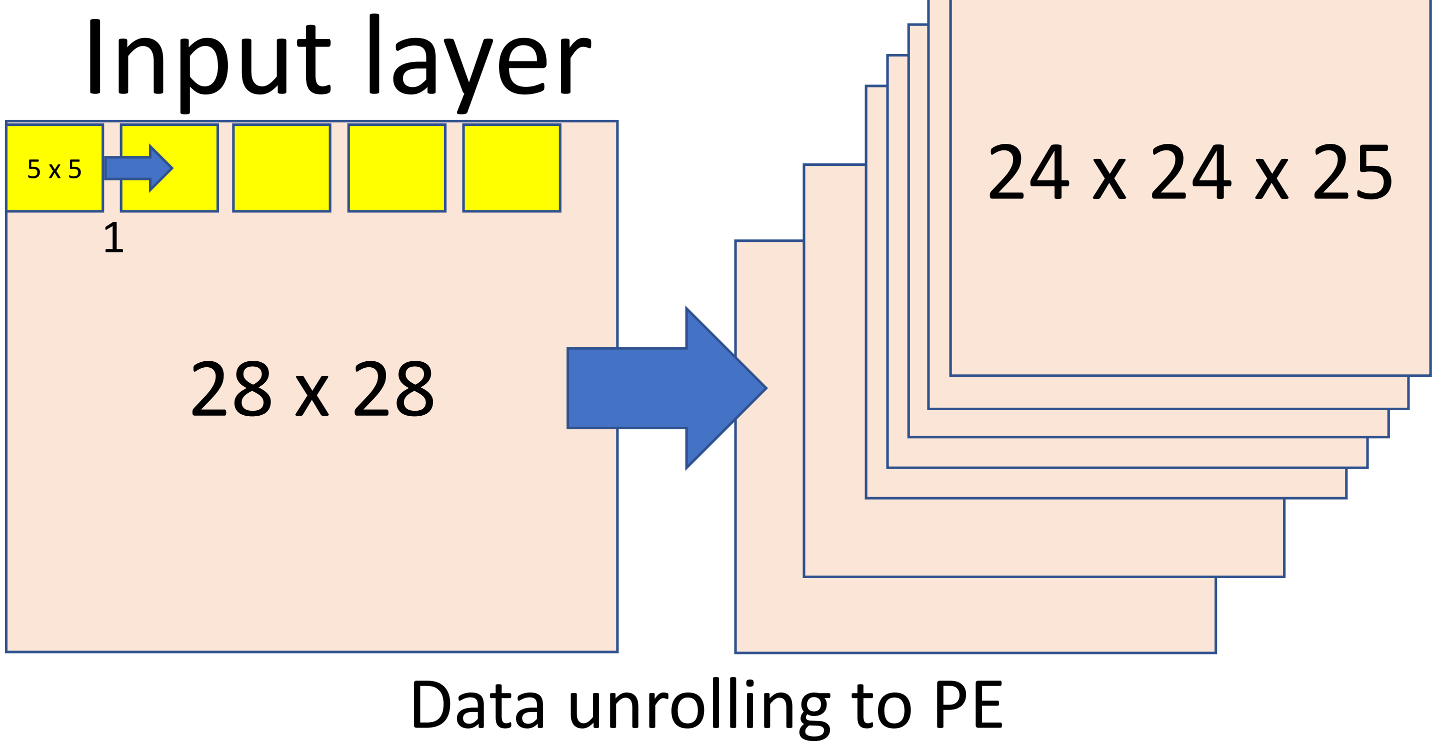
- Involves unrolling (doing all kernel operations on a given layer) in 1-fell swoop on a PE.
- Example:

  28 x 28 pixel layer

  5 x 5 pixel kernel

  stride of 1 pixel

- While great (and super efficient) in theory, data duplicates everywhere!



Intra-kernel (3) Data Unrolling cont.

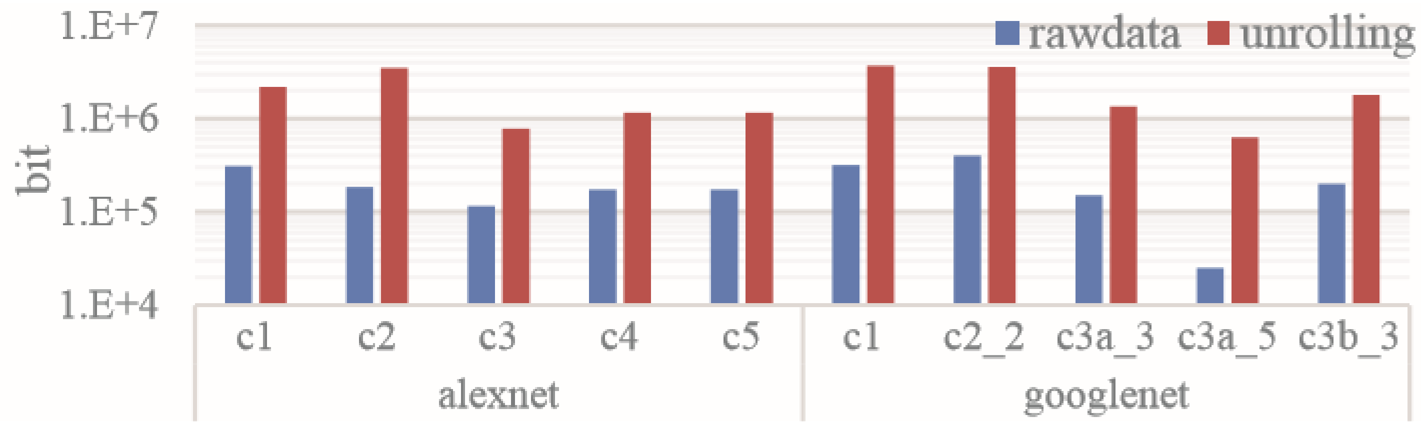
Example:
28 x 28 pixel layer
5 x 5 pixel kernel
stride of 1 pixel

Data increase by factor of T, given input layer X*Y, kernel k, stride s

$$T = \frac{((X-k)/s+1)\times((Y-k)/s+1)\times k\times k}{X\times Y}$$

= 4.22x raw input size

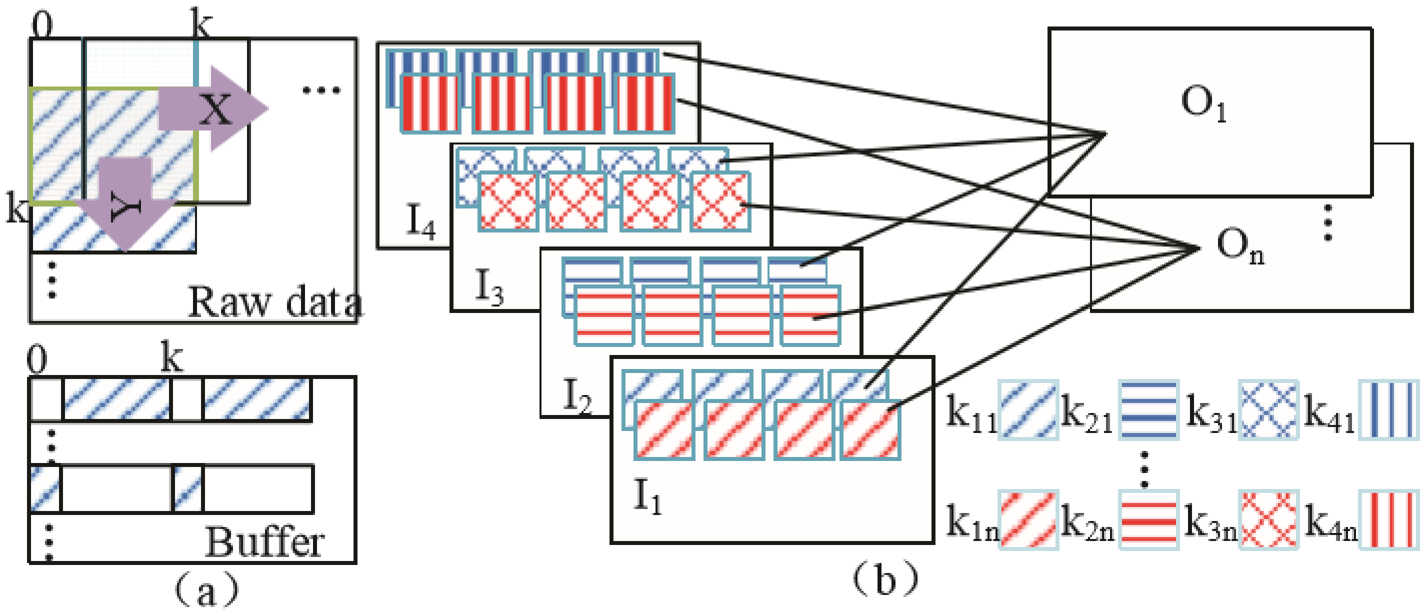
Data duplication
rose by factor of
9x ~ 18.9x
on AlexNet and
GoogleNet



We'll tackle this later!

Intra-kernel (4) Sliding Window

- Only good when kernel size = stride (k=s)
  - In most cases, k > s
- This special case avoids the data overlap & duplication we saw before

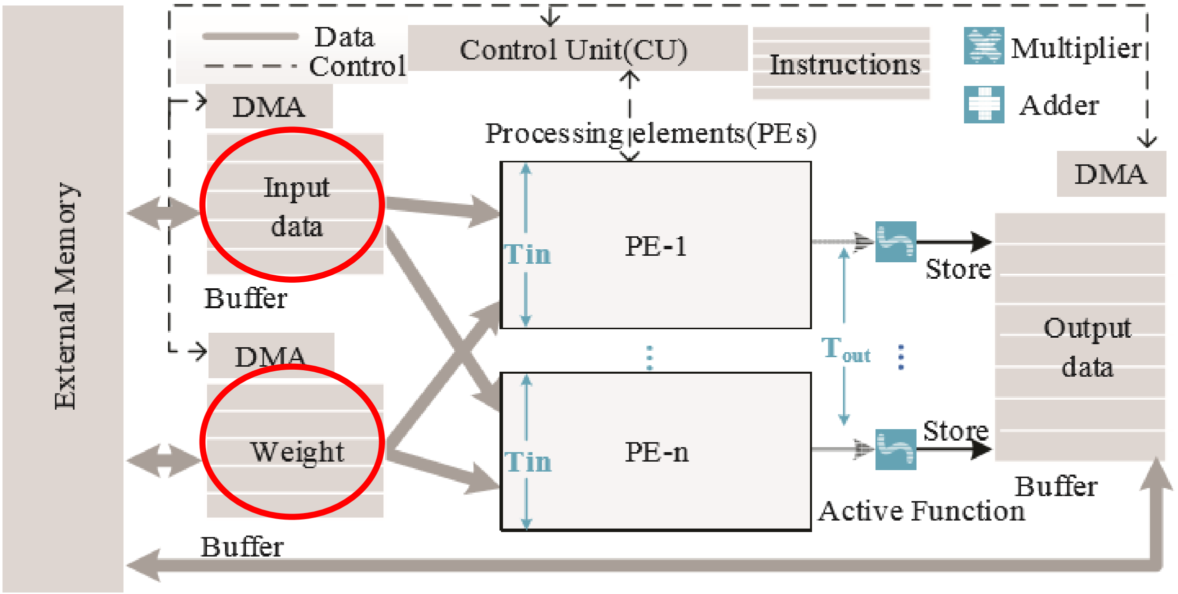


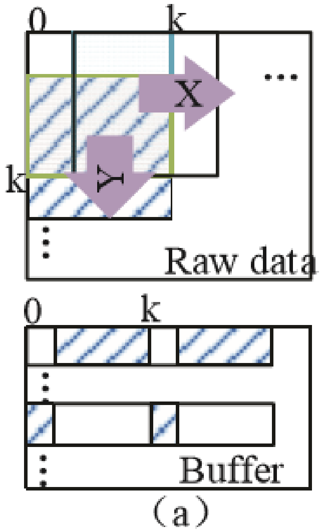
Intra-kernel (5) 2D-PEs

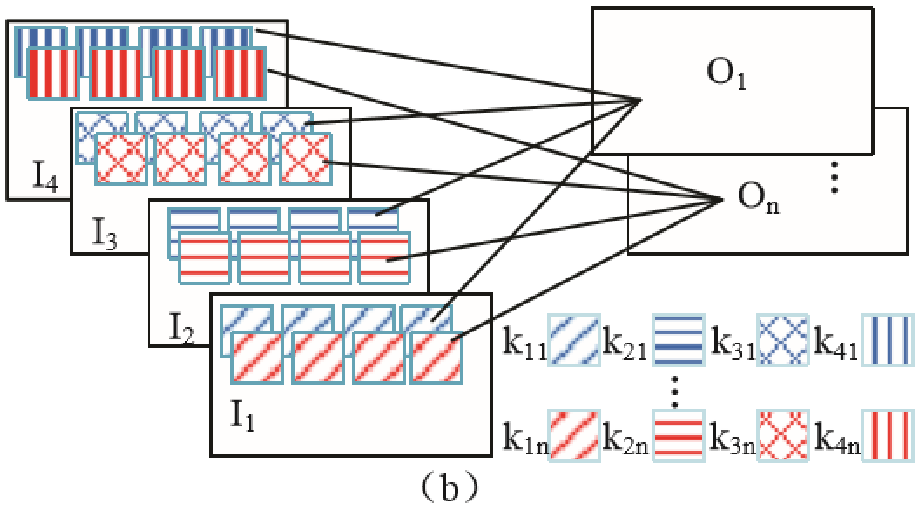
- The best solution for that pesky data overlap/duplication
- Flexible system where we can store consistently-accessed input data OR weight **in buffer**, rather than external memory

PROS: Lowered bus traffic considerably. More power efficient, too.

CONS: Layers vary in kernel size and parameters, so making sure everything is aligned in PEs is hard







k_{11} can be stored in buffer while PE cycles through all kernels in I1.

OR

I_1 can be stored in buffer while PE cycles through all weights k_{11}~k_n.

Hybrid (inter- & intra-)

How can we use inter- and intra-kernel parallelization intelligently?

…Kernel-Partitioning!

Given k x k >> T_{in}, and s < k x k

$$g = ceil(k / s), \quad k_s = s$$

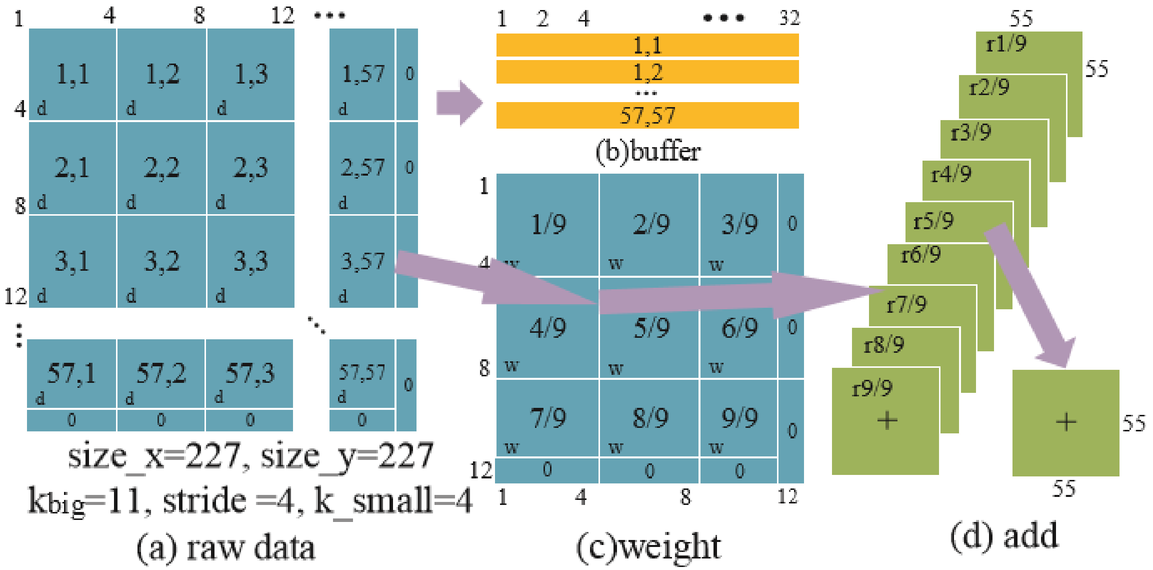
g = # of kernel partitions

k_s = kernel partition stride

In this example, we've convolved a large image of 228x228 to just 9 images of size 55 x 55, **all on PEs**

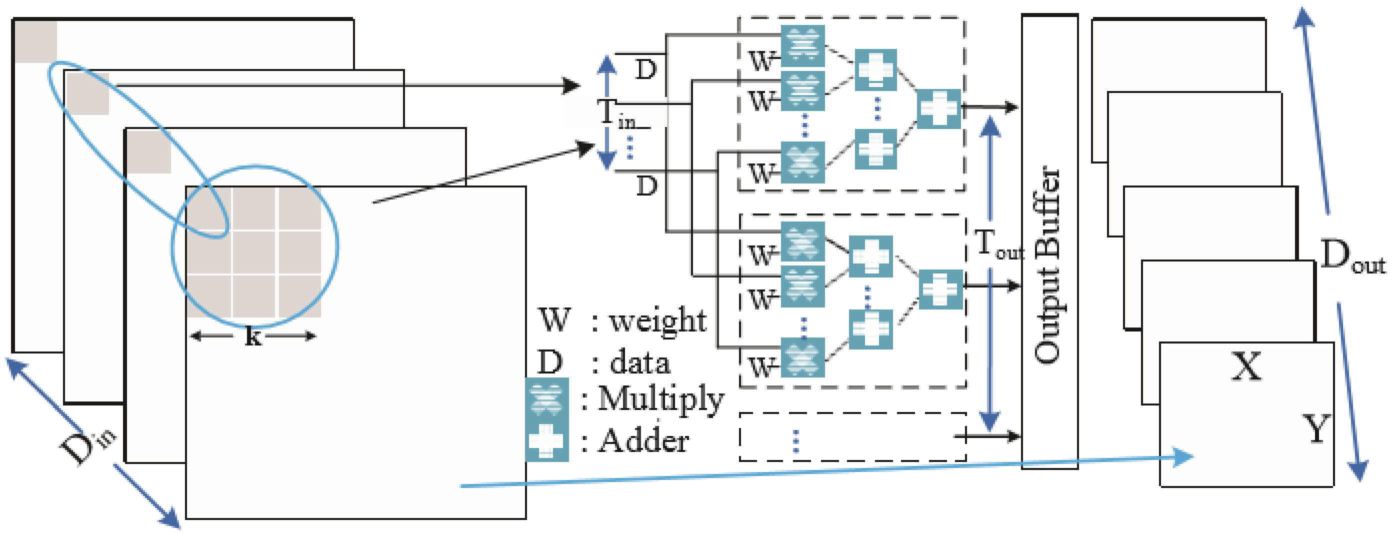
Input:
k:kernel, s:stride, k_s:kernel after partition, g:the groups of partion, T_{in}:the number multiplier in a PE, $size_x$,$size_y$:the size of output maps, G=g×g

```
For i=1:G
   load w_{i/G} to PE
      FOR j_x = i%g:(i%g+size_x), r_x = 1:size_x
      FOR j_y=(i/g+1):(i/g+size_y), ry=1:size_y
         Mapping d(j_x,j_y) to PE
         Calcuate
         IF i=1, THEN store the result to buffer as a pixel located at
             (r_x,r_y) of output map r_{i/G}(in Fig. 5d)
         ELSE reload pixel(r_x,r_y)of r_{(i-1)/G} from buffer, add the MAC
             result to it, then store the sum as a pixel(r_x,r_y) of r_{i/G}
END END END END
```



Furthering the mapping scheme for Kernel-Partitioning

- In particular, how to better use inter-kernel parallelization
  - Recall inter- tends to ignore data reuse between kernel and layer
- Striding kernel tends to reuse data
  - Instead of computing whole kernel before striding,
    do partial sums 1/(k x k) <u>then</u> stride
- Partial sums all sent to output buffer, ready to be added after entire image is complete. Extra store-and-sum operations better than many buffer loads.



Partial sums result in:
$X * Y * D_{out} * k * k$ more stores

But…

$(D_{in}/T_{in}) * X * Y * D_{out} * k * k$ <u>less loads</u>

Kernel-Partitioning Summary

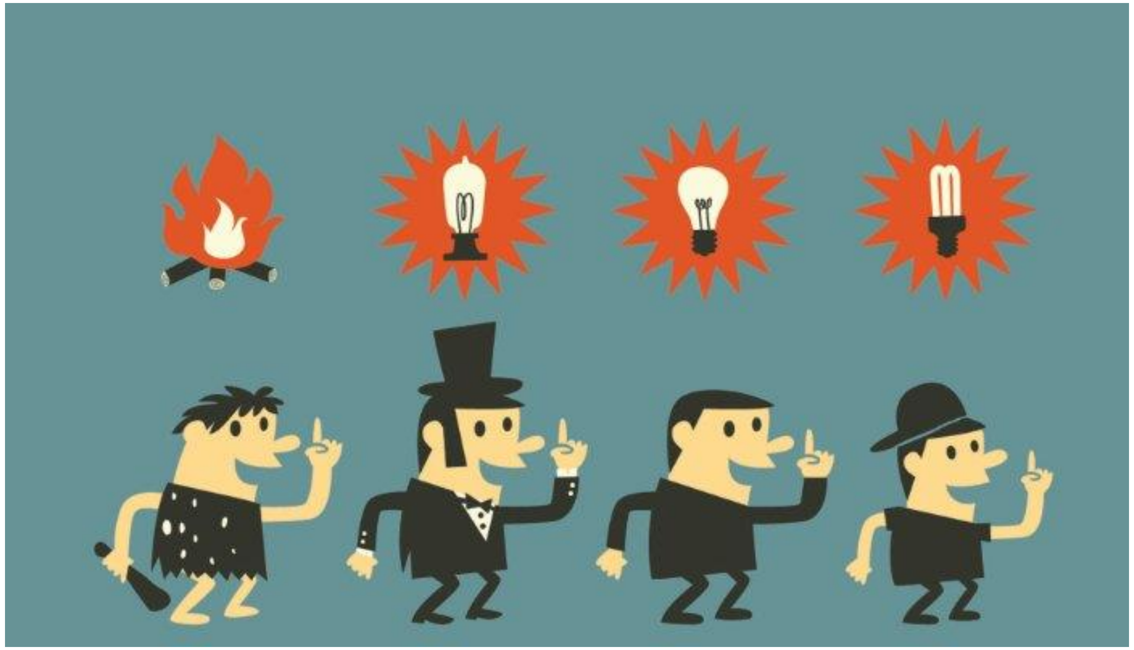
Table 1. Parallelization scheme comparison

scheme	Suited layer characteristic	Advantages
Inter	Large #input maps and small kernel	Implement easily
Intra	Kernel = stride	Less memory traffic
partition	Big kernel or small #input maps	Both of above

Self adaptiveness

- Truth about CNNs:
  - Surface layers: small # input maps, big kernels
  - Deeper layers: large # input maps, small kernels

  ** Due to more and more feature abstractions
  - Thus there is a need to adapt to the changing structure as we venture deep
- Solution: Algorithm to best choose which type of kernel parallelism is best in a given point of the CNN
- 2 adaptive versions were tested:
  - Adpa1- original (limited) inter-kernel parallelism
  - Adpa 2- improved inter-kernel mapping



Given a NN layer:
1: IF k=s and $k \neq 1$, THEN select intra-kernel parallelism
2: ELSE-IF $D_{in} < T_{in}$,THEN select kernel-partition
3: ELSE Select inter-kernel parallelism
4:IF(paralellism scheme of nextlayer is inter-kernel), store in inter-order(Din,X,Y in Fig. 1)
5:ELSE Stroe in intra-order(X,Y,Din in Fig. 1)
Move to next NN layer

Performance evaluation: Speedup

System specs:

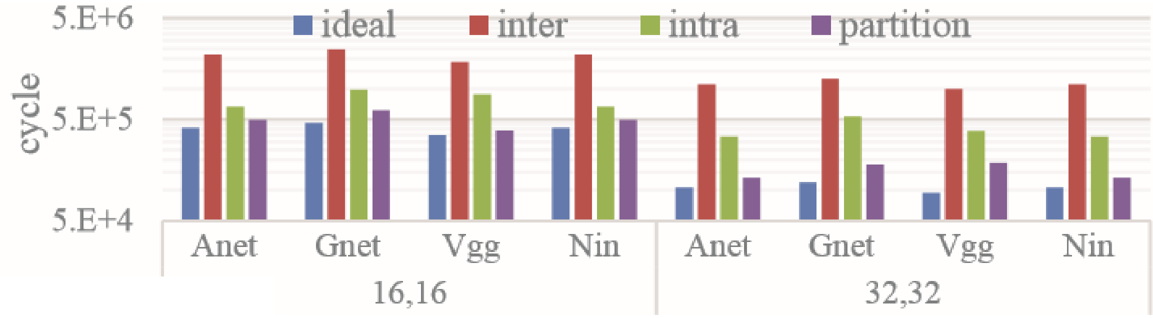
- Verilog-based CNN accelerator
- Synopsys Design Compiler

Neural Net specs:

- Pre-trained CNNs with fixed accuracies
- Only forward propagation
- Data recorded were cycles of simulation

Network	Alexnet	google net	VGG	NiN
Conv1 detail	3,11,4,96	3,7,2,64	3,3,1,64	3,11,4,96
#conv layers	5	57	16	12
Kernel types	11,5,3	7,5,3,1	3	11,5,3,1

name	bandwidth	size	opertation	cycle
PE	16-16,32-32	16bit	mulitplication	1
InOut-buf	16,32	2M Byte	add	1
Weight-buf	256,1024	1M Byte	load	1
Bias_buf	16,32	4K Byte	store	1

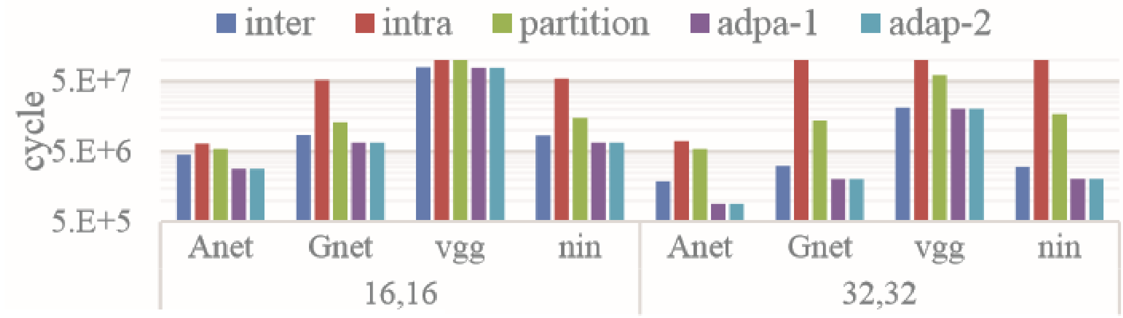


Comparison of execution time of layer Conv-1

Outperforms Intel Xeon 2.2GHz by whopping 696.88x max

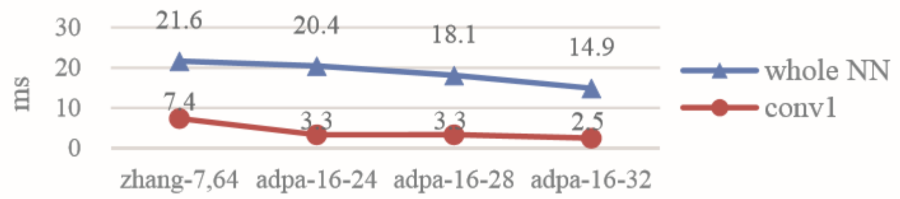
Table 4 Performance comparied to CPU(ms)

	CPU	adap-16-16	speedup	adap-32-32	speedup
Anet	376.50	2.83	133.02x	0.91	414.58x
Gnet	1418.8	6.69	212.11x	2.04	696.88x
Vgg	10071.71	77.51	129.94 x	20.41	493.44x
Nin	553.43	6.72	82.35 x	2.05	269.77x



Performance comparison

Outperforms Zhang-7-64's FPGA (circa 2015) by 2.22x on Conv1 1.20x whole network



Performance evaluation: Energy Consumption

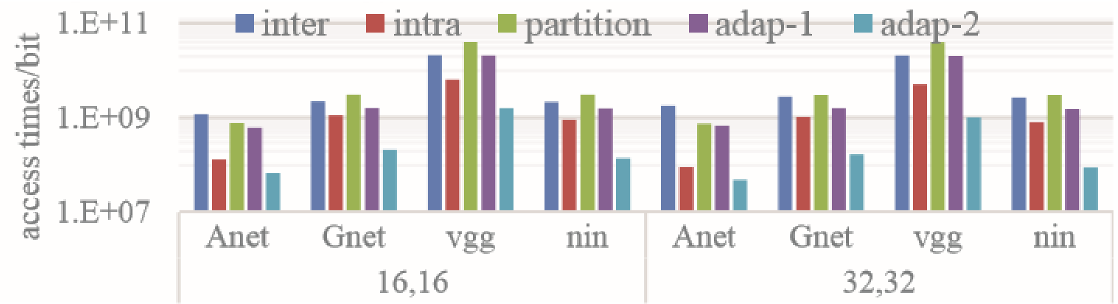
System specs:
- Verilog-based CNN accelerator
- Synopsys Design Compiler

Neural Net specs:
- Pre-trained CNNs with fixed accuracies
- Only forward propagation
- Data recorded were cycles of simulation

Network	**Alexnet**	**google net**	VGG	NiN
Conv1 detail	3,11,4,96	3,7,2,64	3,3,1,64	3,11,4,96
#conv layers	5	57	16	12
Kernel types	11,5,3	7,5,3,1	3	11,5,3,1

name	bandwidth	size	opertation	cycle
PE	16-16,32-32	16bit	mulitplication	1
InOut-buf	16,32	2M Byte	add	1
Weight-buf	256,1024	1M Byte	load	1
Bias_buf	16,32	4K Byte	store	1

Best result: Adpa2 90.13% memory traffic reduction



Buffer traffic comparison

Thus, Adpa2 also achieved 47.1% energy reduction

Table 5 PEs Energy reduction (%)

	inter(base)	intra	partition	adap-1	adap-2
Alexnet	0.00	32.85	40.23	47.77	47.71
Googlenet	0.00	9.66	22.77	31.48	31.40
VGG	0.00	-44.72	-8.61	3.00	2.89

Conclusion

- Achieved a generalized, flexible, CNN accelerator that outperforms several current accelerators on popular CNNs
- Uses a variety of innovative data-parallel schemes
- Highly adaptive, which allows it to maintain speedups and save energy, no matter what network, or what layers within a network

Thank you!