**Lecture 19:**

Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2019

Today

- **We will discuss the workload of <u>evaluating</u> deep neural networks (performing "inference")**

  - **This lecture will be heavily biased towards concerns of DNNs that process images (to be honest, because that is what your instructor knows best)**

  - **Which admittedly is not the majority of DNN evaluation in the world right now (text processing, speech, ads, etc.)**
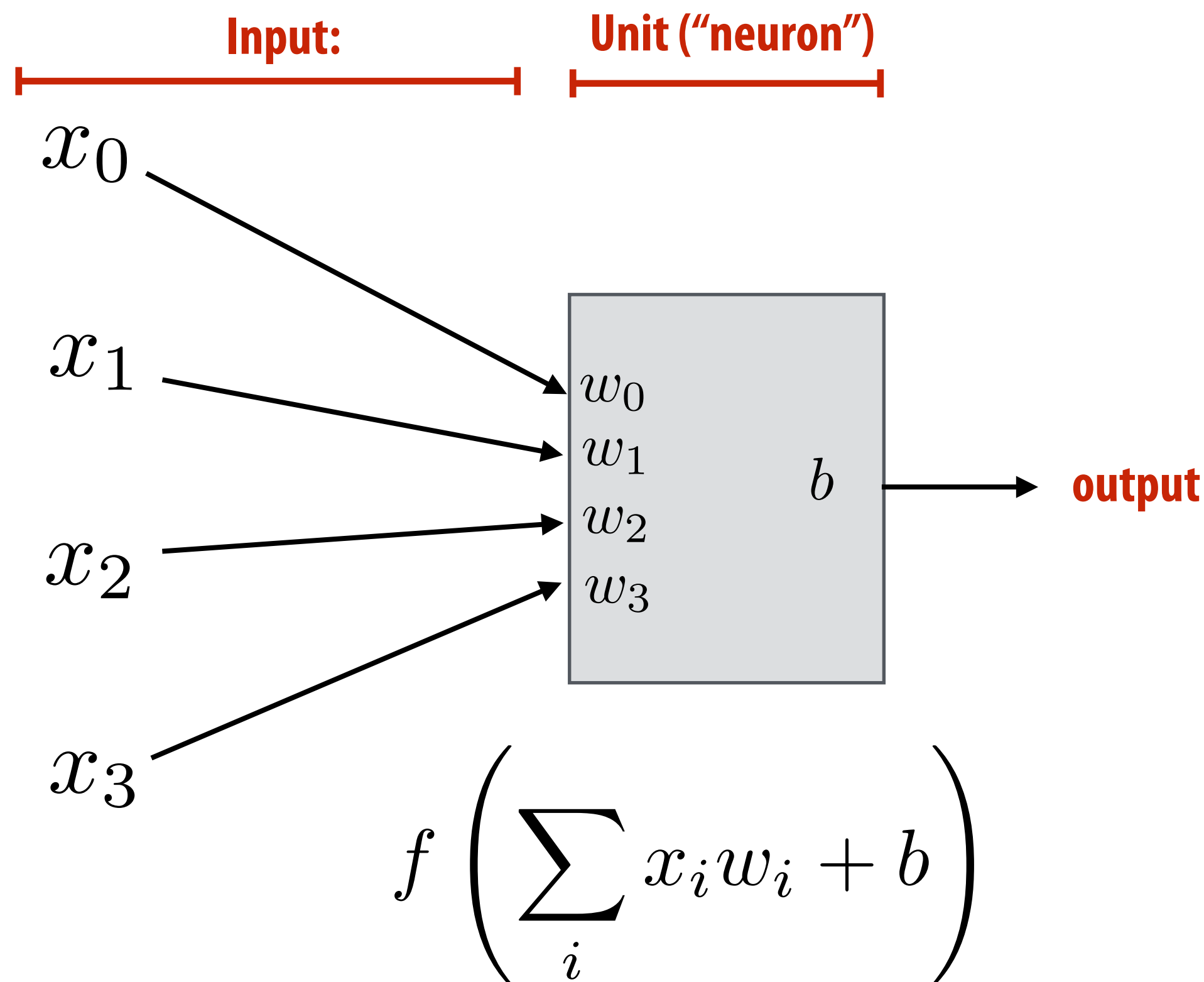
Consider the following expression
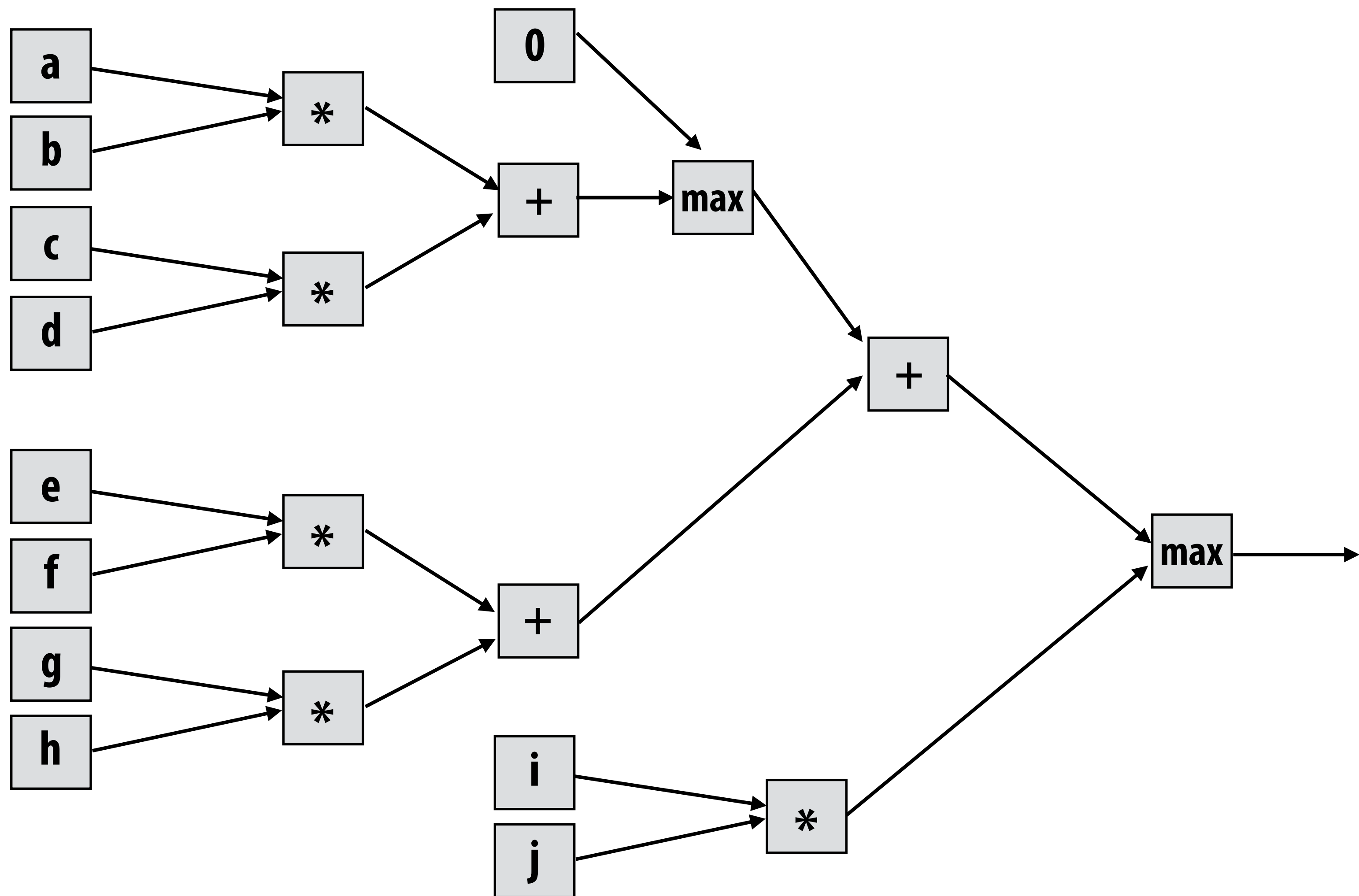

```
max( max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j)
```

What is a deep neural network?

A basic unit:

Unit with *n* inputs described by *n+1* parameters (weights + bias)

Input: Unit ("neuron")

x_0

x_1

x_2

x_3

w_0
w_1
w_2
w_3

b

output

$$f\left(\sum_i x_i w_i + b\right)$$

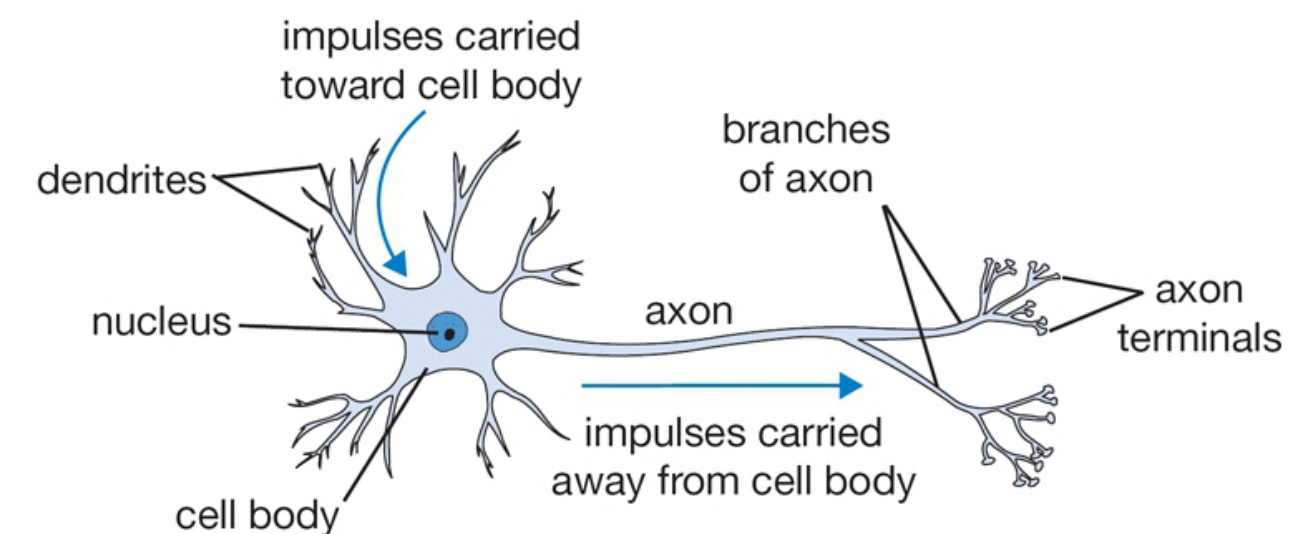**Example: rectified linear unit (ReLU)**

$f(x) = max(0, x)$

Basic computational interpretation:

It is just a circuit!
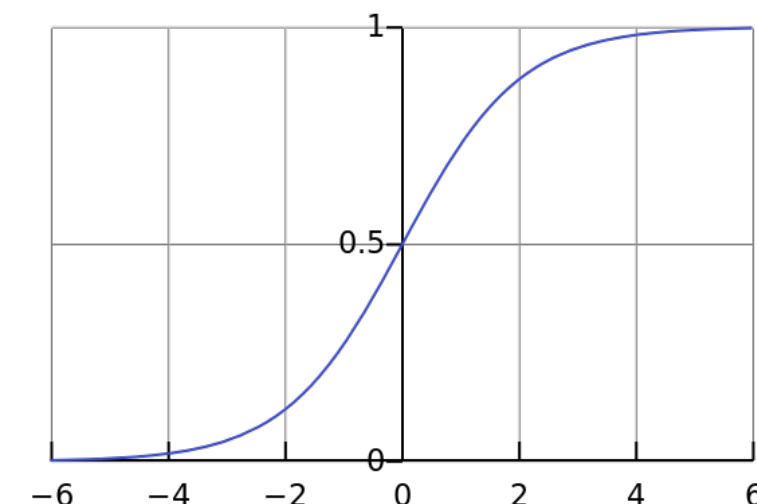
Biological inspiration:

unit output corresponds loosely to activation of neuron

impulses carried toward cell body

dendrites

nucleus

cell body

axon

impulses carried away from cell body
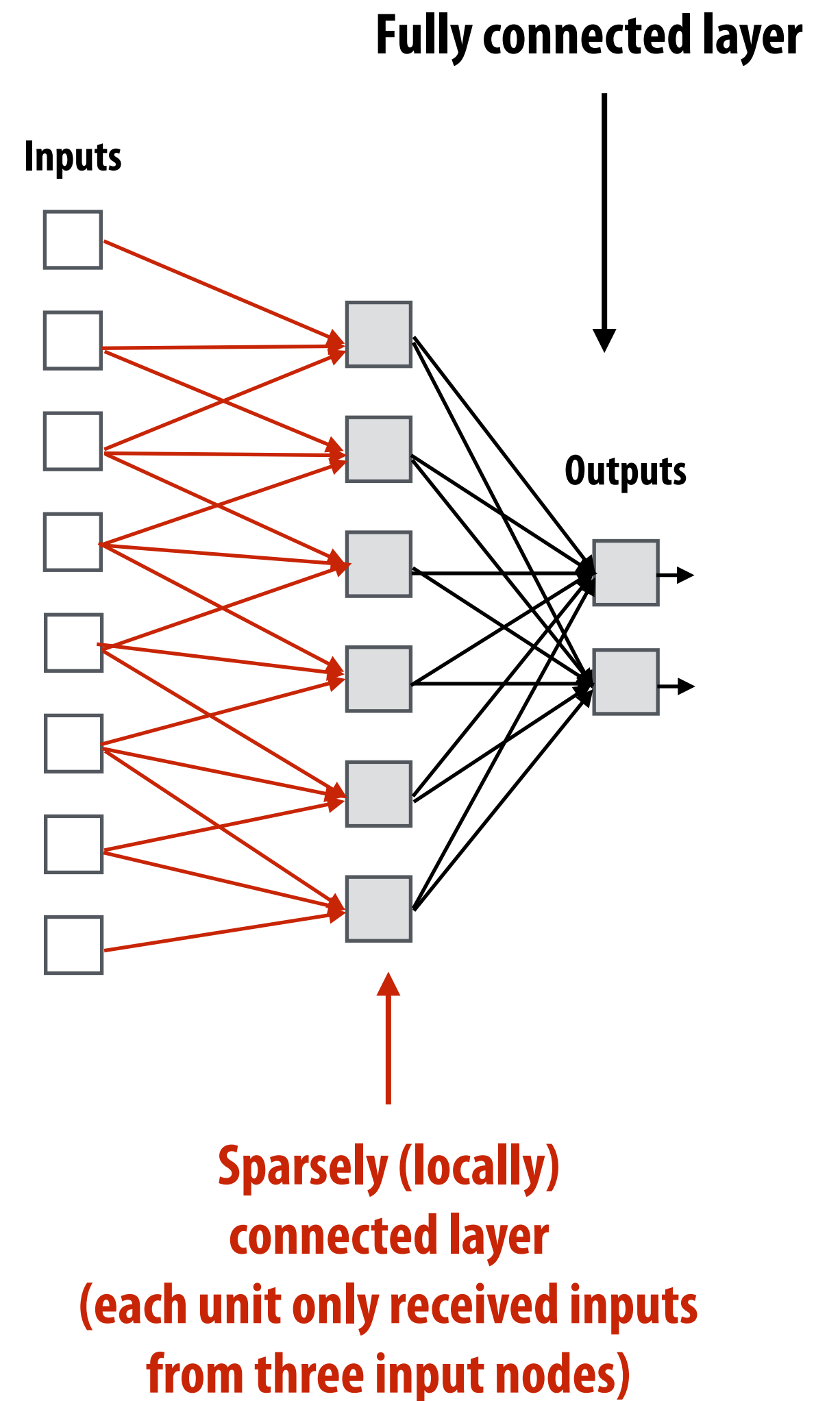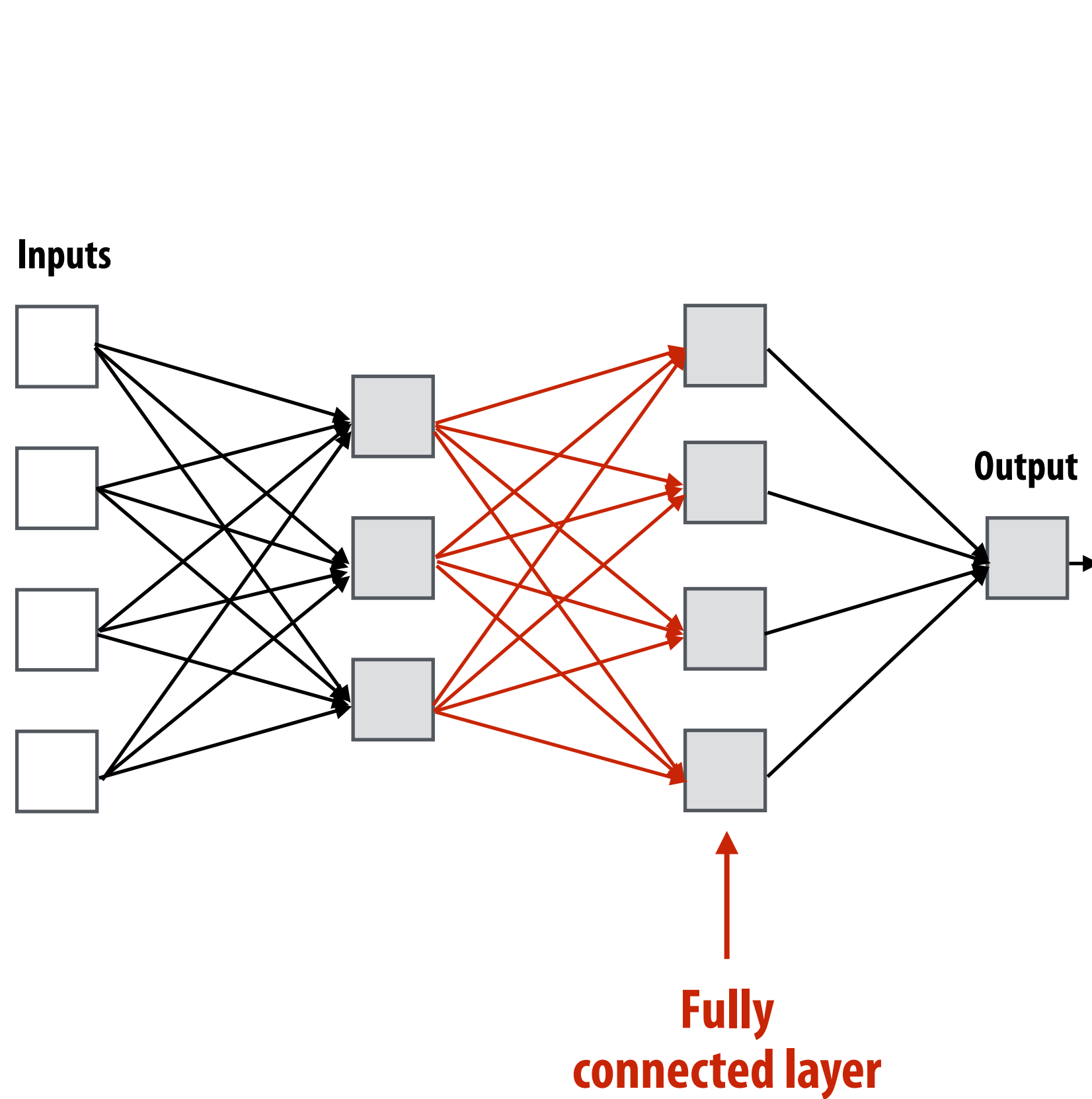
branches of axon

axon terminals

Machine learning interpretation:

binary classifier: interpret output as the probability of one class

$$f(x) = \frac{1}{1 + e^{-x}}$$

Deep neural network: topology

Inputs

Output

Fully connected layer

Inputs

Outputs

Fully connected layer

Sparsely (locally) connected layer (each unit only received inputs from three input nodes)
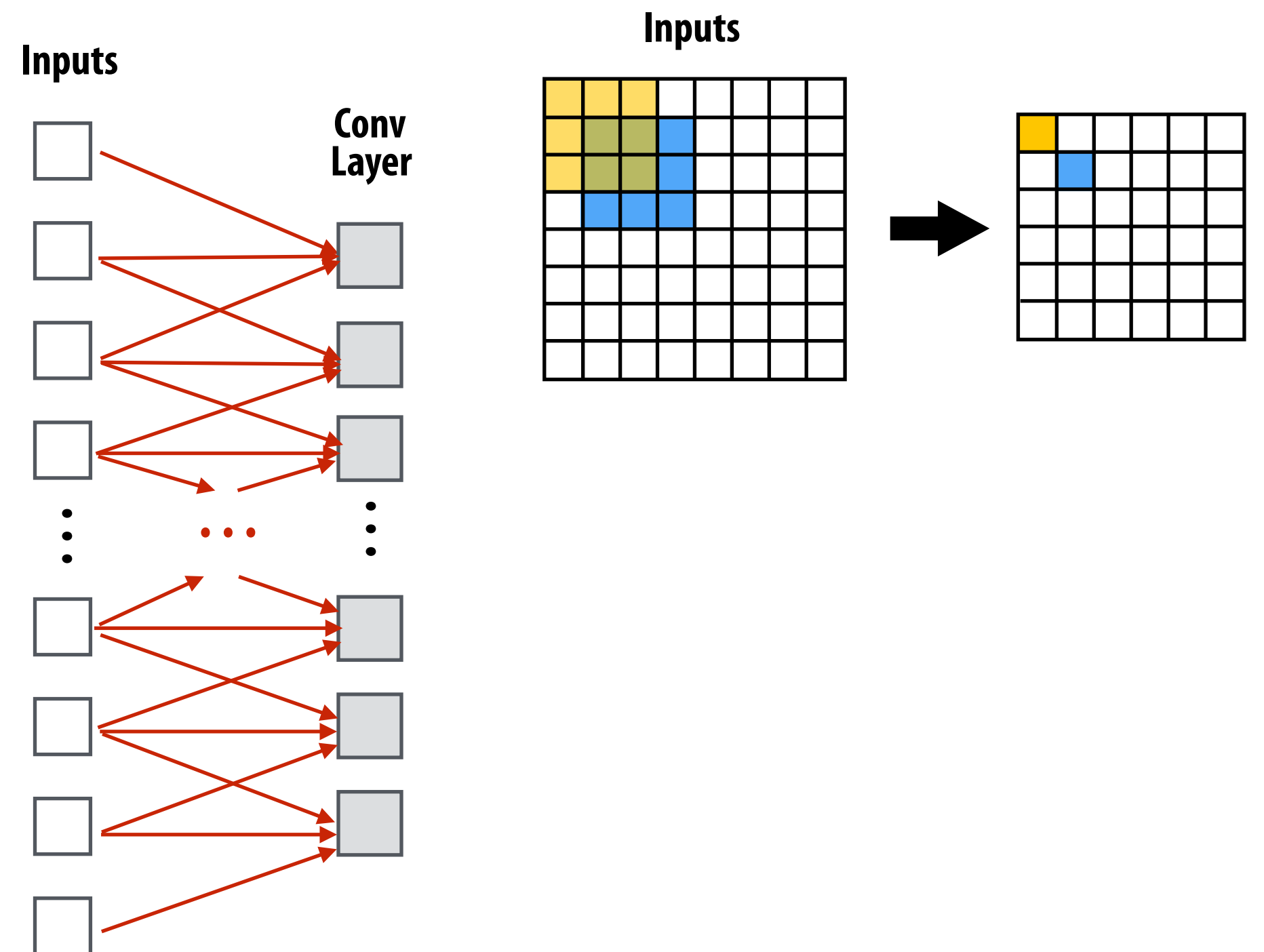
Recall image convolution (3x3 conv)

```
int WIDTH = 1024;

int HEIGHT = 1024;

float input[(WIDTH+2) * (HEIGHT+2)];

float output[WIDTH * HEIGHT];


float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};
```


Inputs

Conv Layer

Inputs

```
for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```

Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):

(note: network illustration above only shows links for a 1D conv:
a.k.a. one iteration of `ii` loop)

Strided 3x3 convolution

Inputs


```
int WIDTH = 1024;
int HEIGHT = 1024;
int STRIDE = 2;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[(WIDTH/STRIDE) * (HEIGHT/STRIDE)];



float weights[] = {1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9,
                   1.0/9, 1.0/9, 1.0/9};


for (int j=0; j<HEIGHT; j+=STRIDE) {
  for (int i=0; i<WIDTH; i+=STRIDE) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++) {
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[(j/STRIDE)*WIDTH + (i/STRIDE)] = tmp;
  }
}
```

Inputs
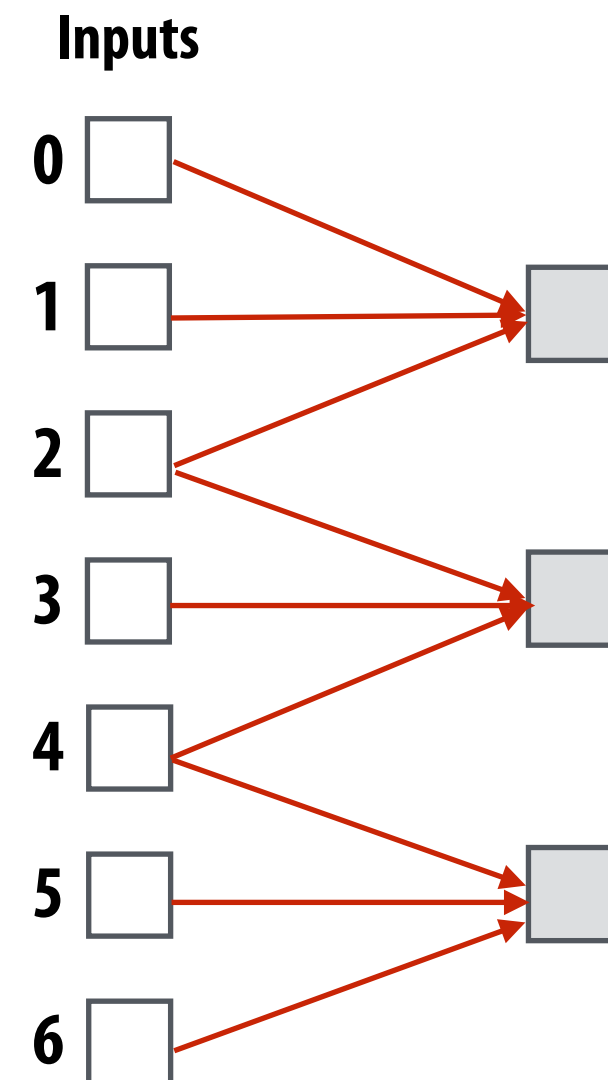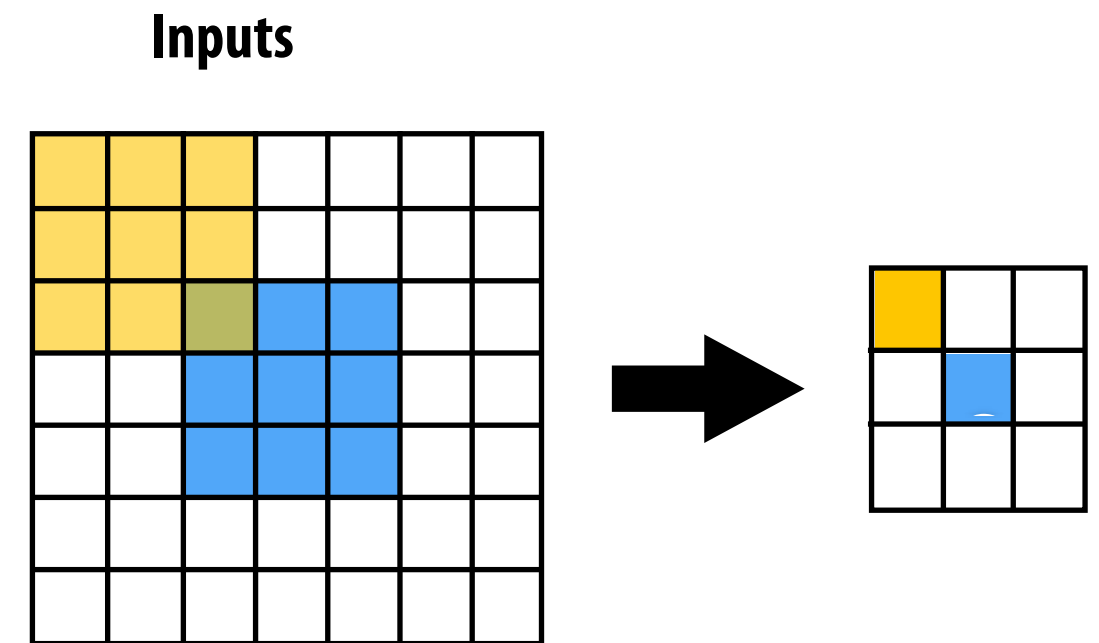
Convolutional layer with stride 2
(0,1,2) , (2,3,4) , (4,5,6) , ...

What does convolution using these filter weights do?

$$\begin{bmatrix} .111 & .111 & .111 \\ .111 & .111 & .111 \\ .111 & .111 & .111 \end{bmatrix}$$

"Box blur"

Original

Blurred
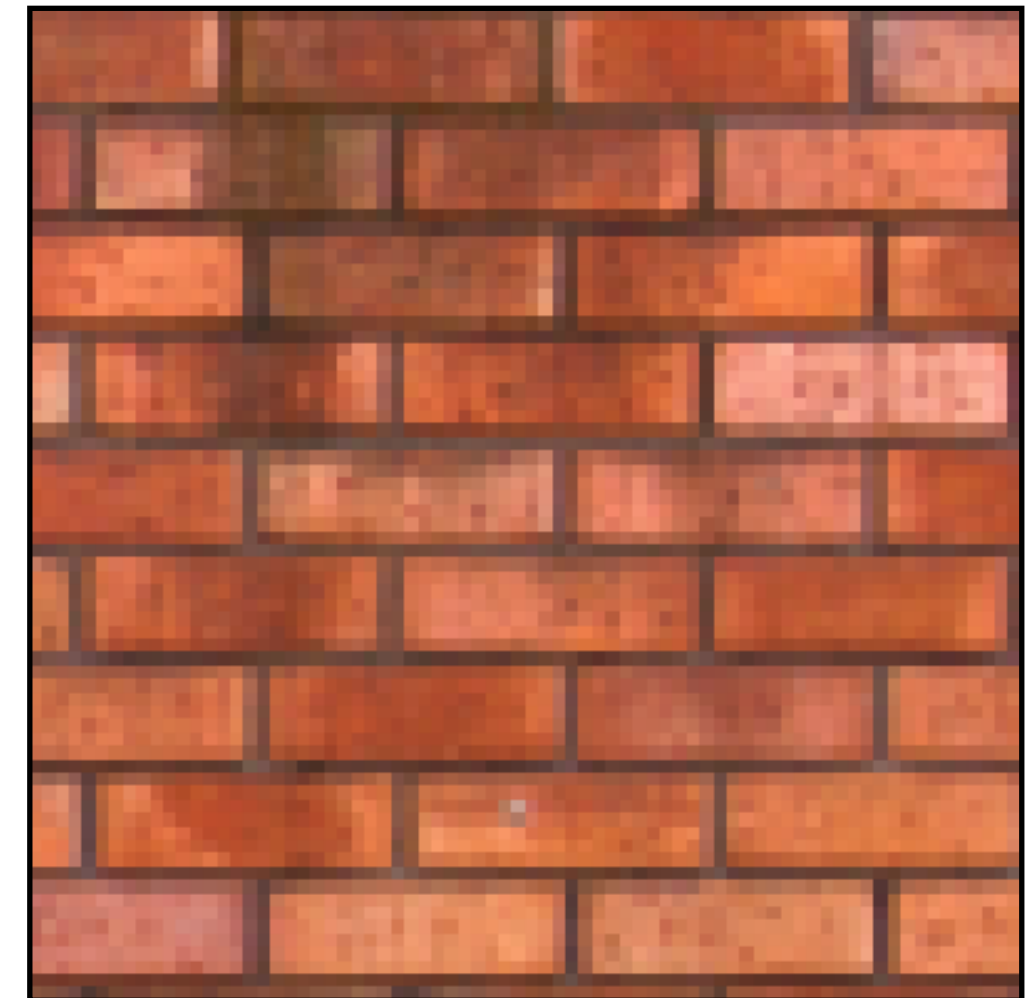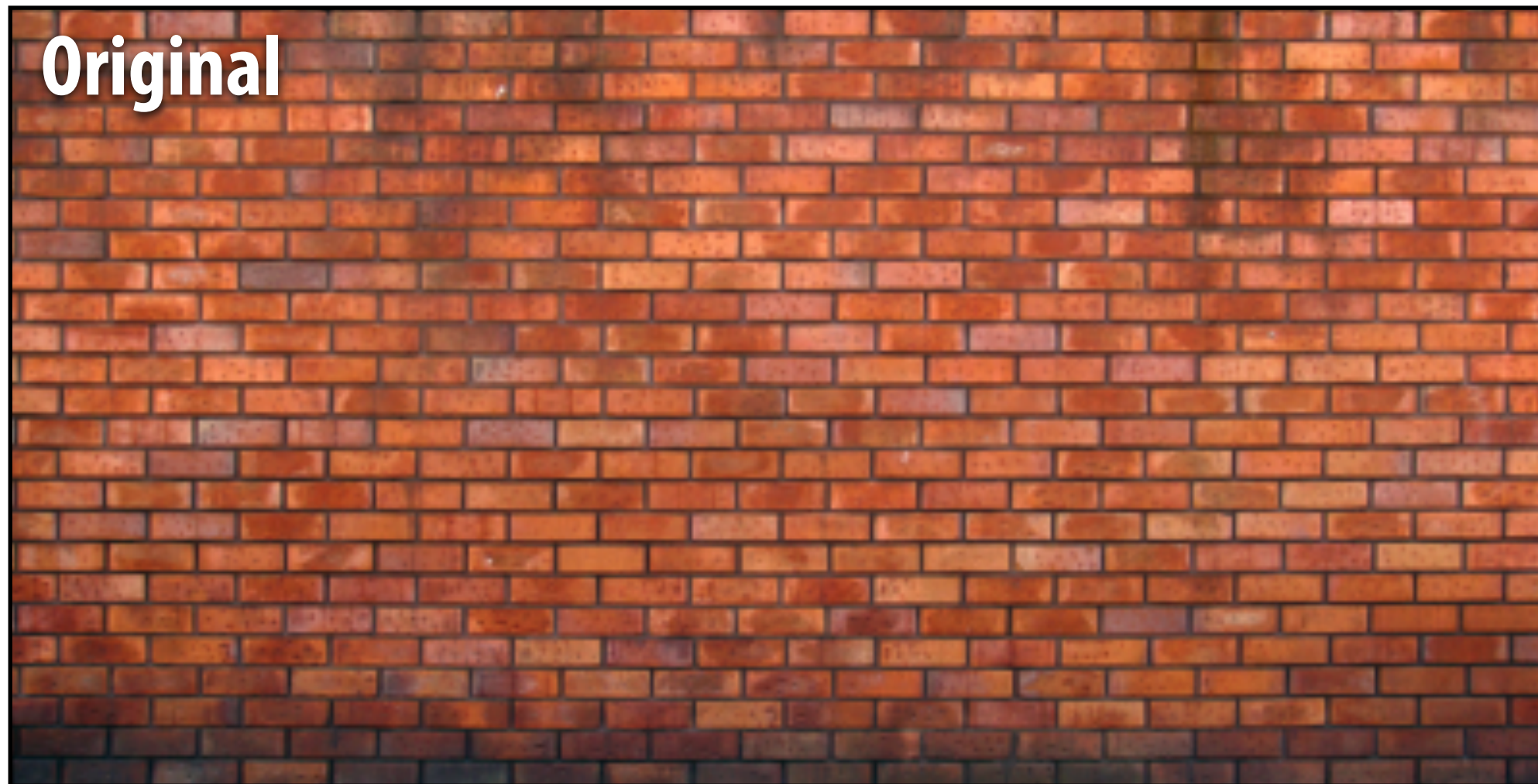
What does convolution using these filter weights do?

$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

"Gaussian Blur"

Original

Blurred
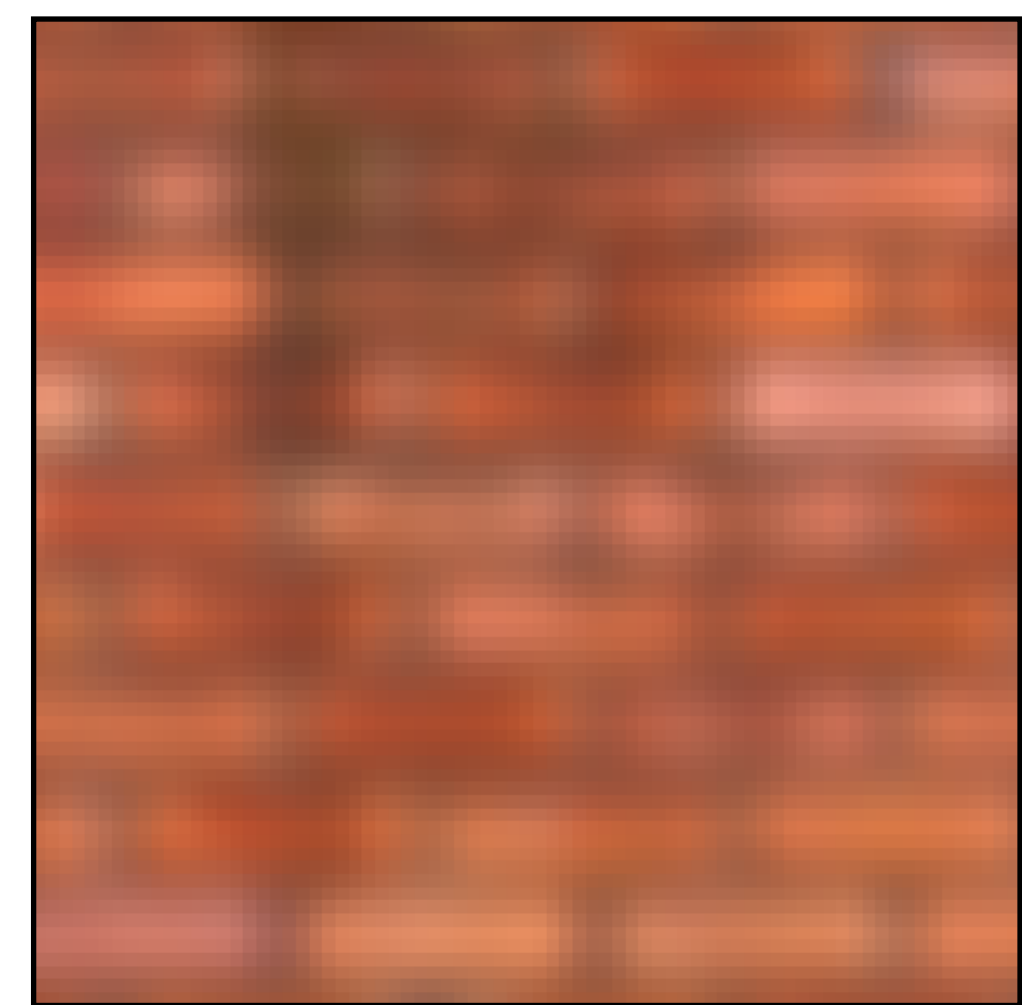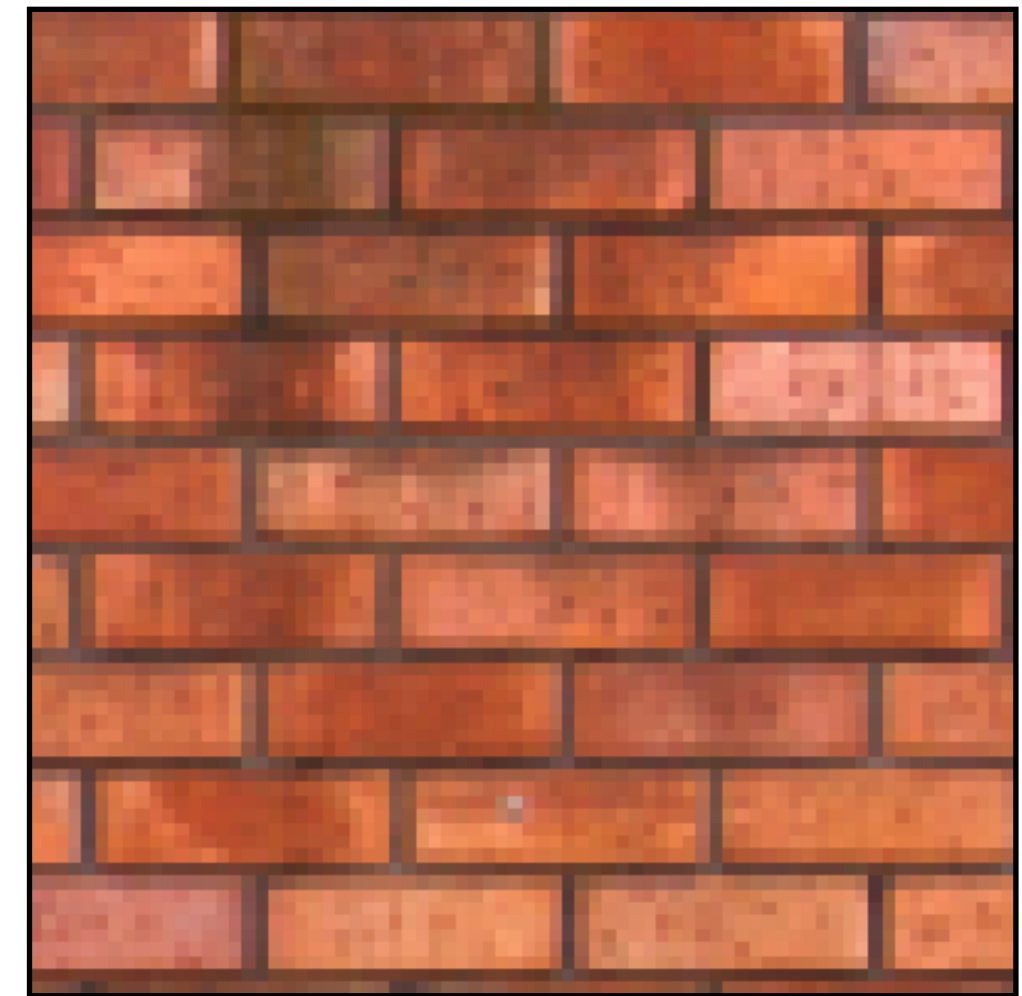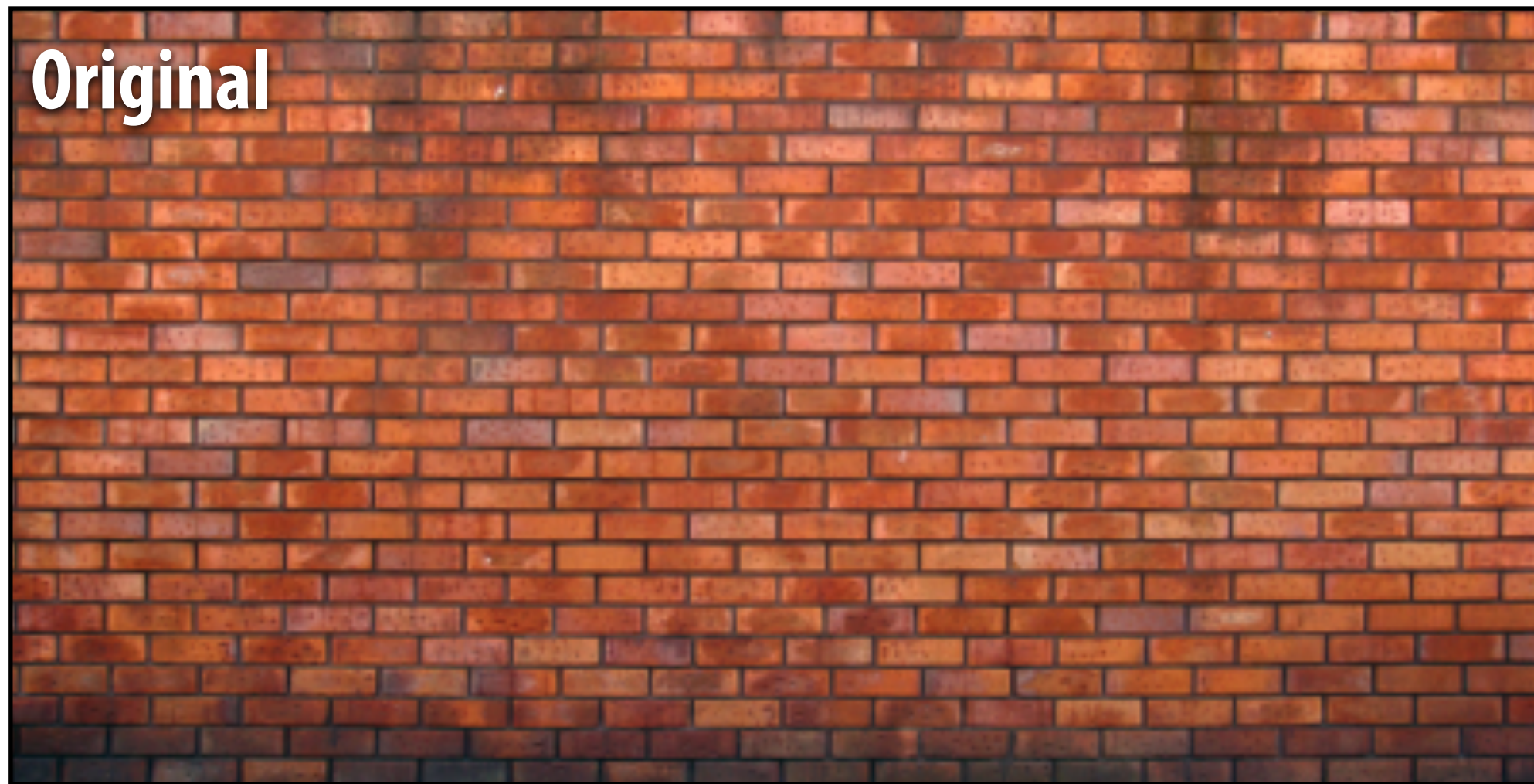
What does convolution with these filters do?

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Extracts horizontal gradients

Extracts vertical gradients

Gradient detection filters
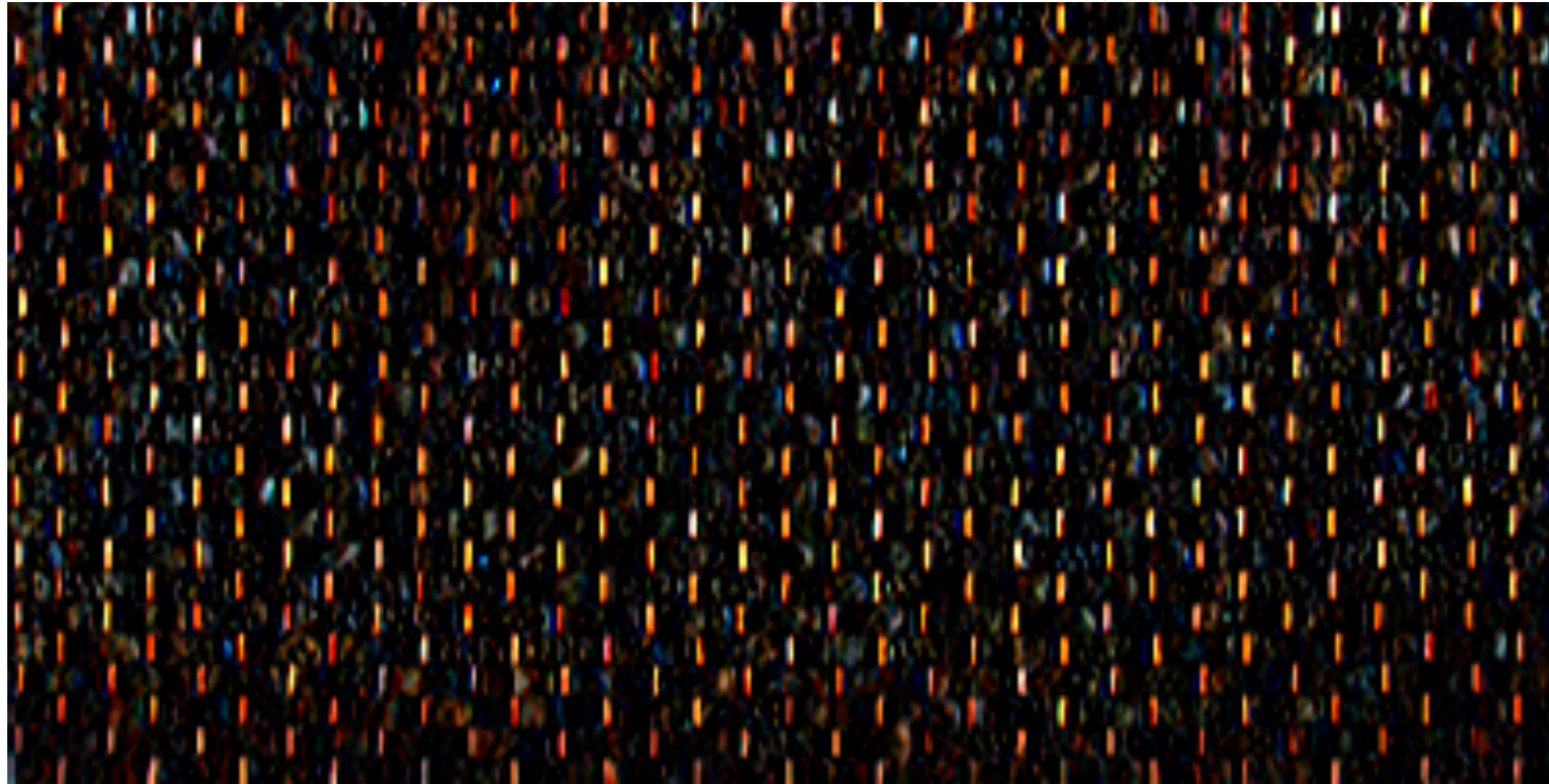
Horizontal gradients
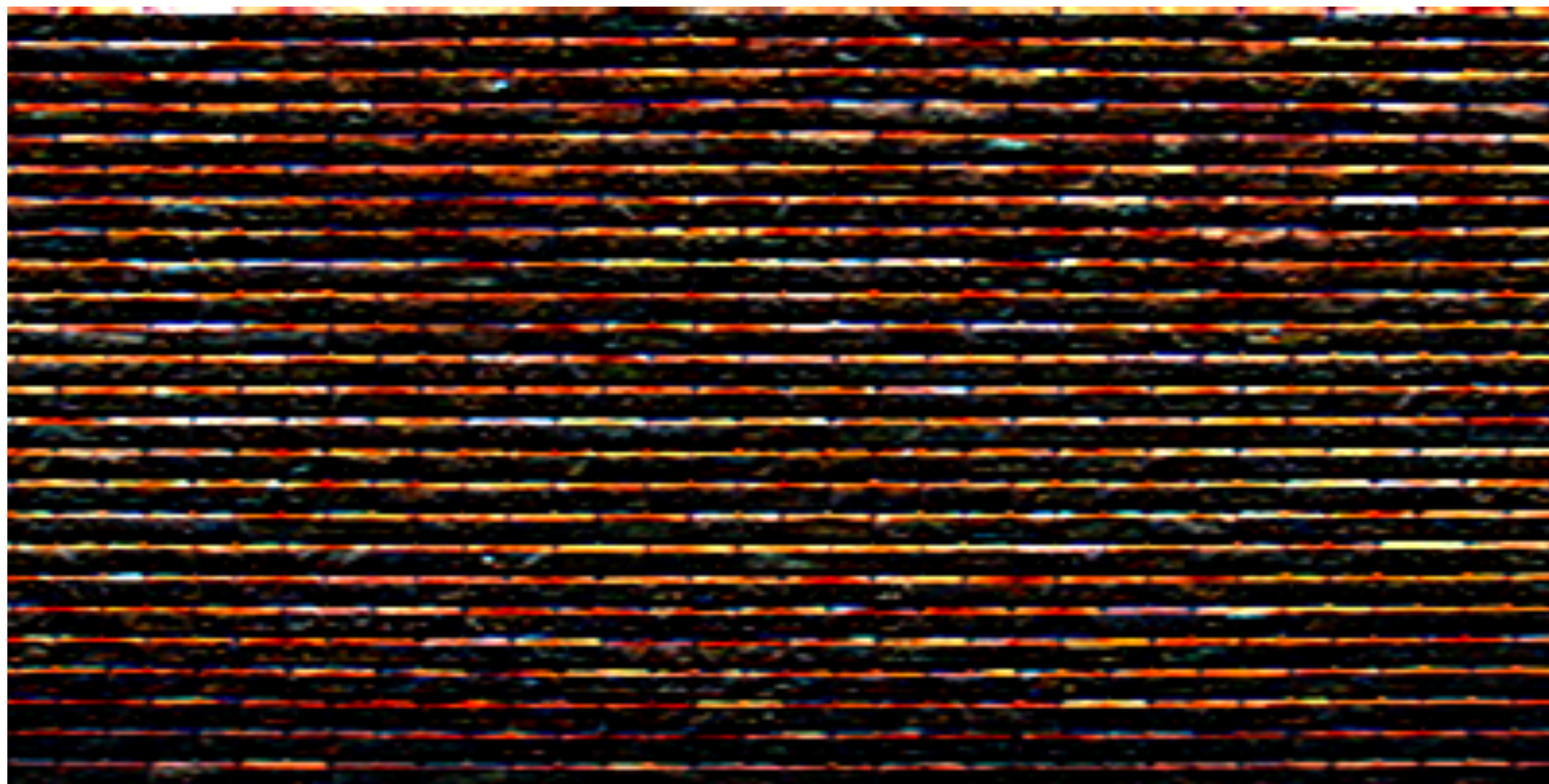
Vertical gradients

Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image

Applying many filters to an image at once

Input: image (single channel):
W x H
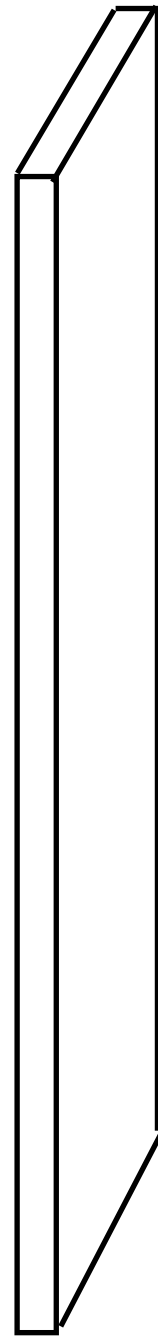
3x3 spatial convolutions on image
3x3 x num_filters weights

...

Output: filter responses
W x H x num_filters

...
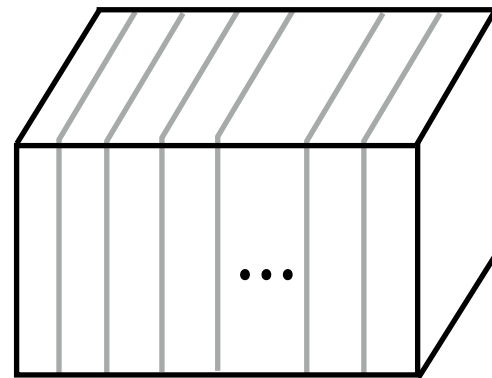
Each filter described by unique
set of 3x3 weights
(each filter "responds" to
different image phenomena)
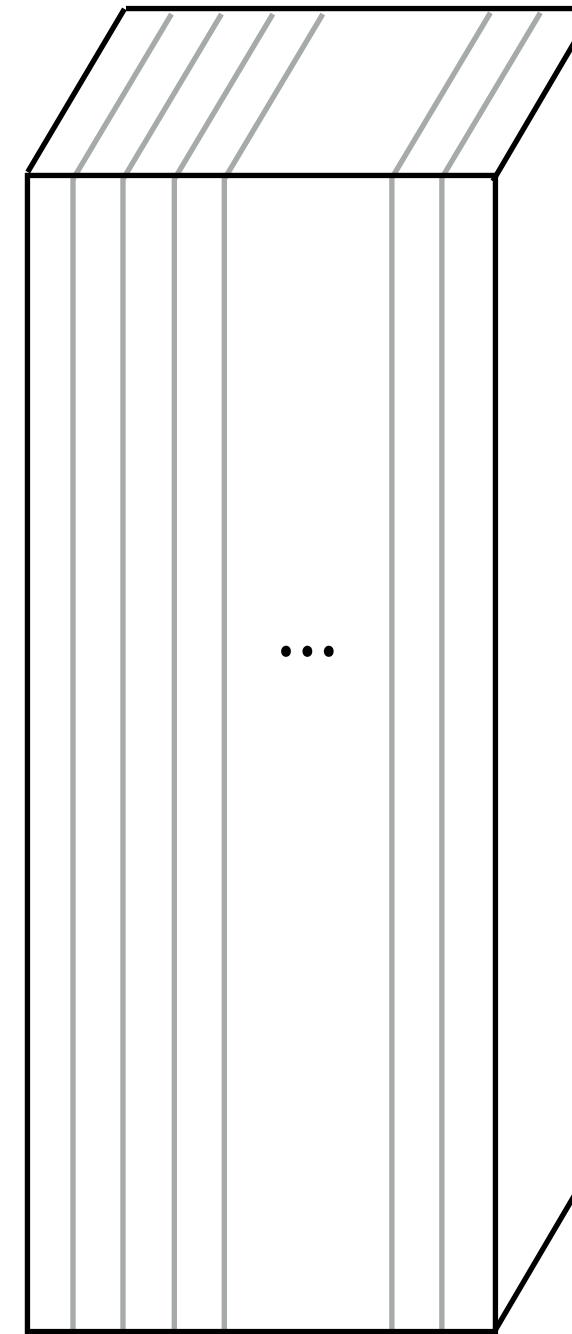
Filter response maps
(num_filters of them)

Applying many filters to an image at once

Input RGB image (W x H x 3)

**96 11x11x3 filters
(operate on RGB)**
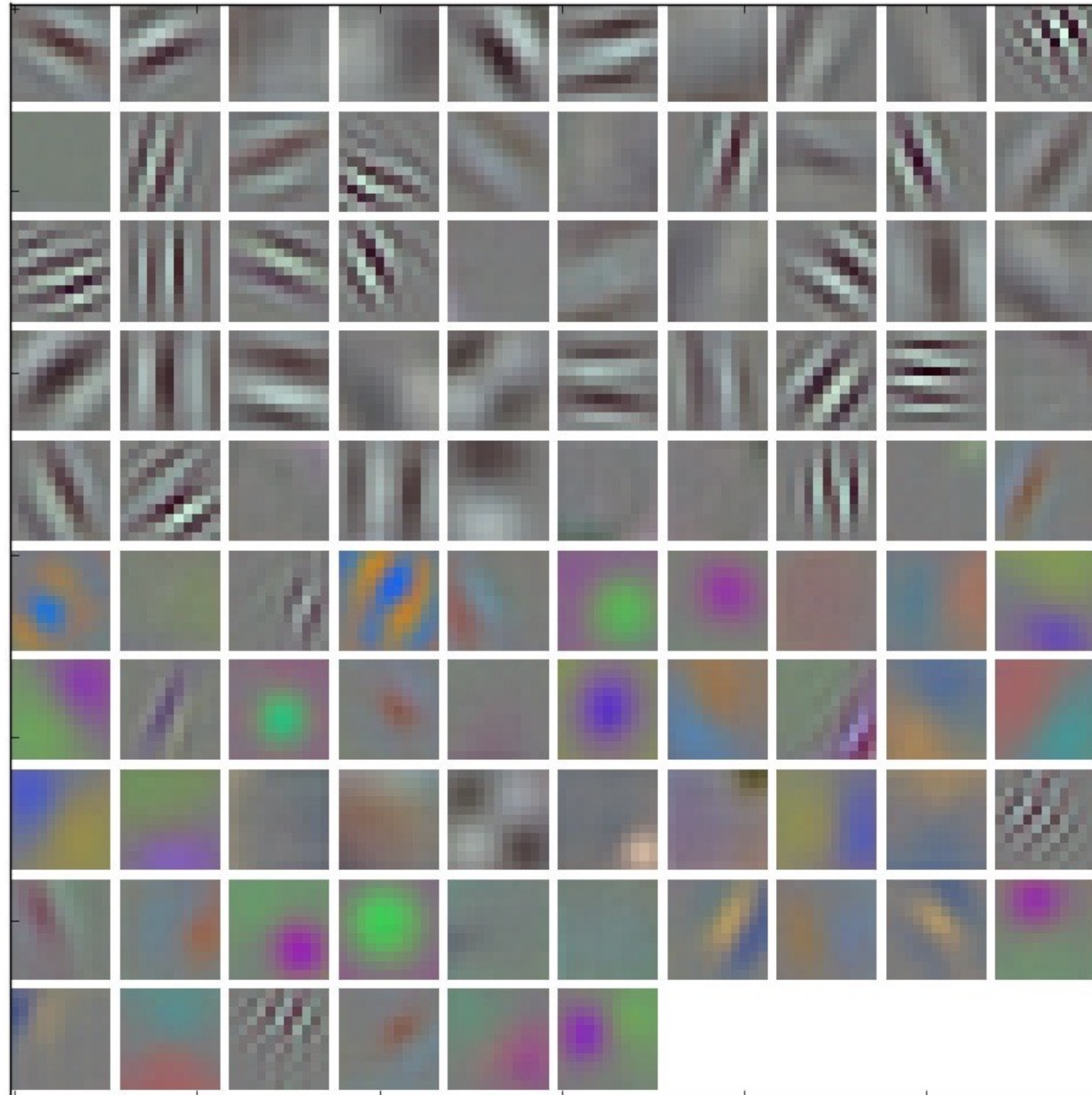
96 responses (normalized)
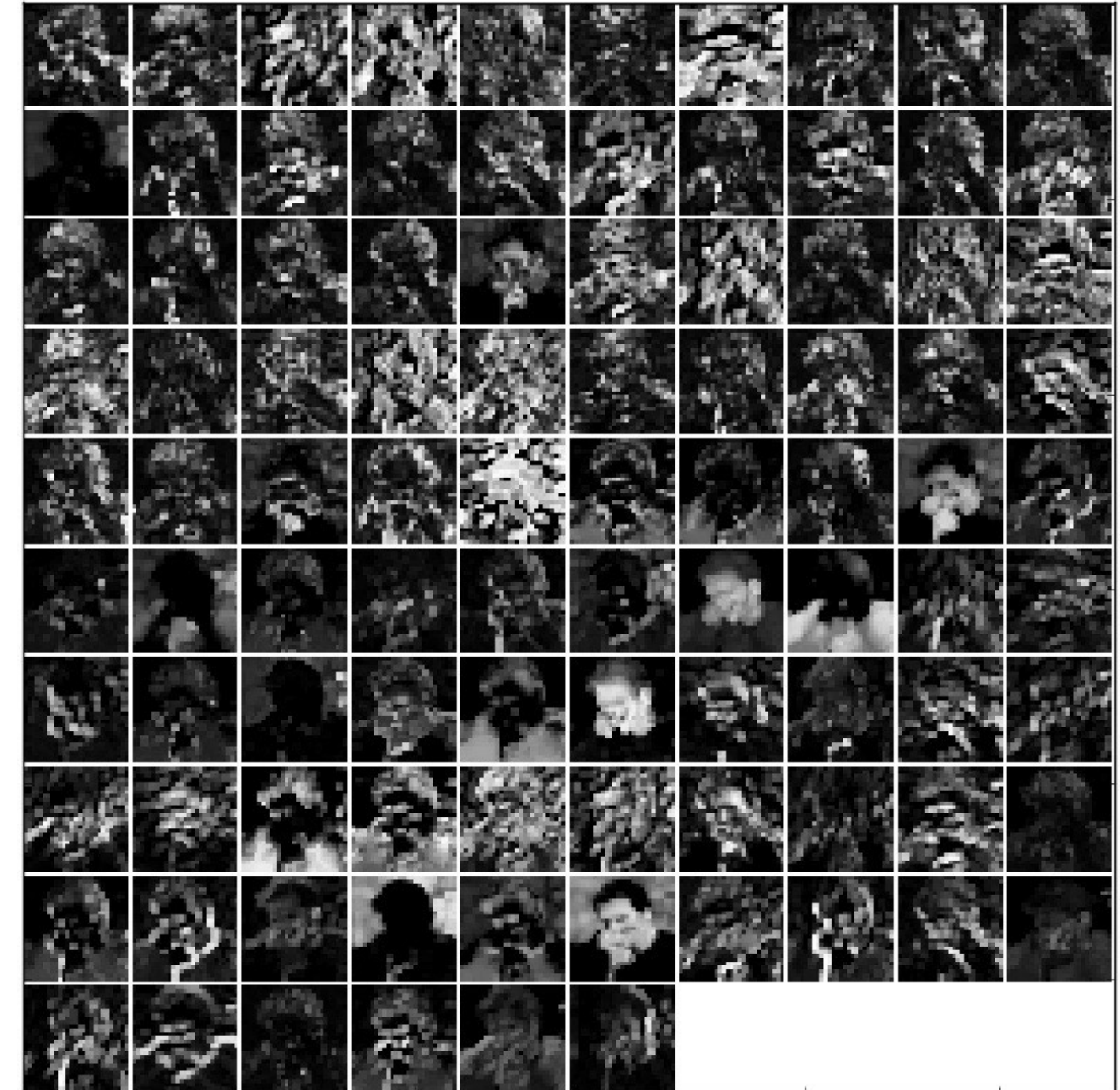
Adding additional layers

**Input: image
(single channel)
W x H**

**3x3 spatial convolutions
3x3 x num_filters weights**

Conv

**Output: filter responses
W x H x num_filters**

ReLU

**post ReLU
W x H x num_filters**

Pool

**(max response
in 2x2 region)**

**post pool
W/2 x H/2 x num_filters**

**Each filter described by
unique set of weights
(responds to different
image phenomena)**

Filter responses

**Note data reduction as a
result of pooling**

Example: "AlexNet" object detection network

Sequences of conv + reLU + pool (optional) layers

Example: AlexNet [Krizhevsky12]: 5 convolutional layers + 3 fully connected layers
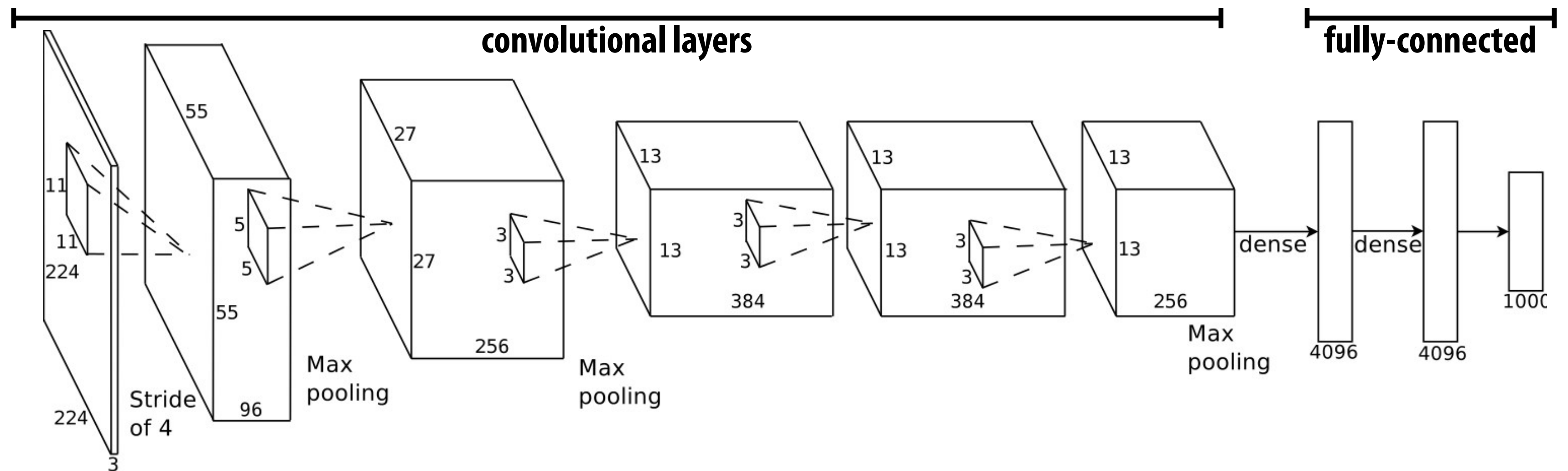
Another example: VGG-16 [Simonyan15]: 13 convolutional layers

input: 224 x 224 RGB	conv/reLU: 3x3x128x256	conv/reLU: 3x3x512x512
conv/reLU: 3x3x3x64	conv/reLU: 3x3x256x256	conv/reLU: 3x3x512x512
conv/reLU: 3x3x64x64	conv/reLU: 3x3x256x256	conv/reLU: 3x3x512x512
maxpool	maxpool	maxpool
conv/reLU: 3x3x64x128	conv/reLU: 3x3x256x512	fully-connected 4096
conv/reLU: 3x3x128x128	conv/reLU: 3x3x512x512	fully-connected 4096
maxpool	conv/reLU: 3x3x512x512	fully-connected 1000
	maxpool	soft-max
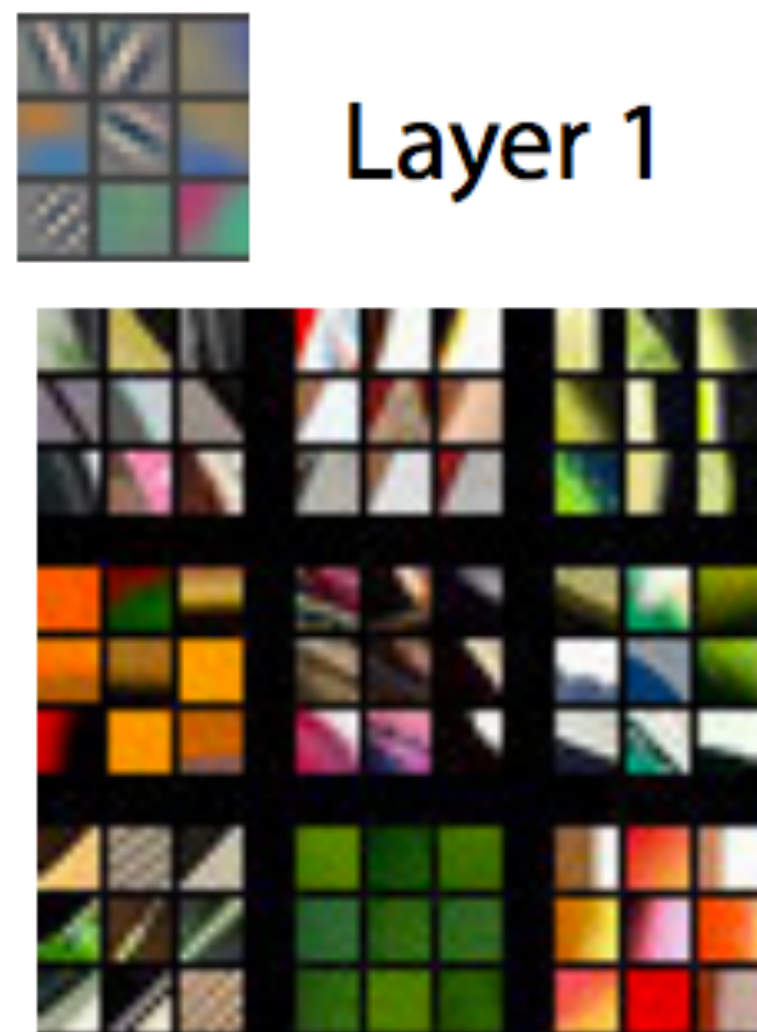
Why deep?

Left: what pixels in image patch trigger the response
Right: images that generate strongest response for filters at each layer

Layer 1

Layer 2

Layer 3

[image credit: Zeiler 14]
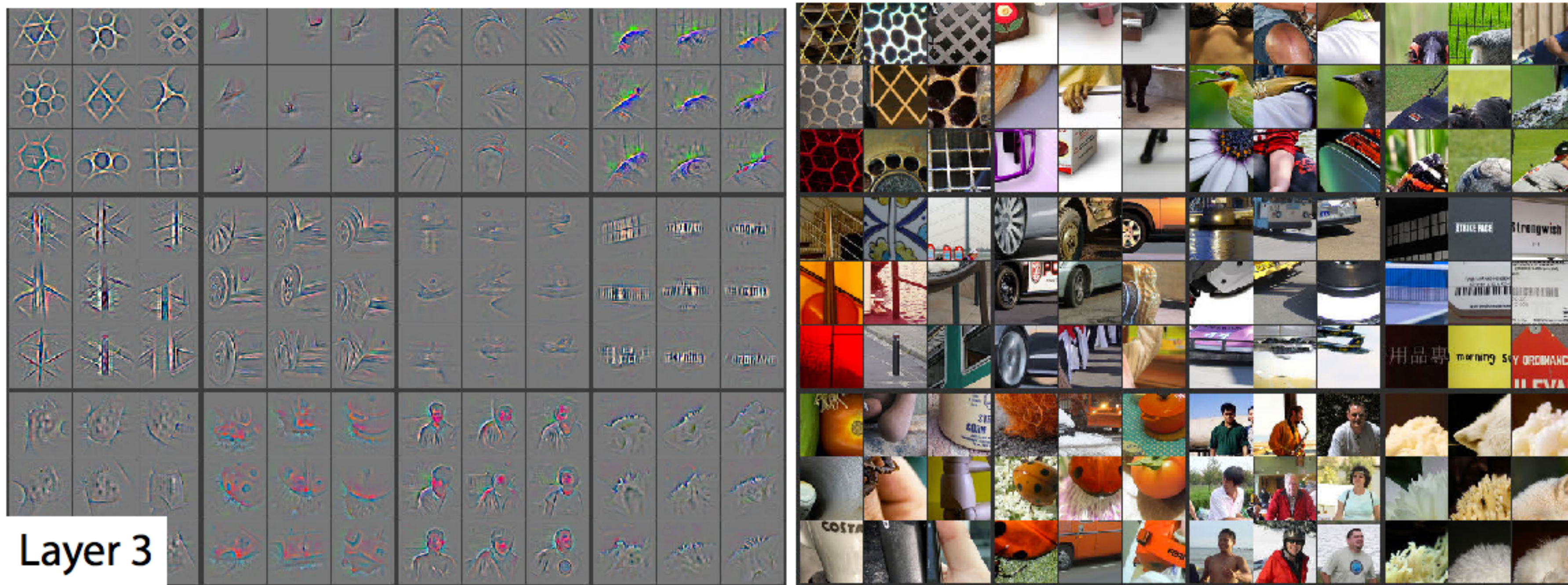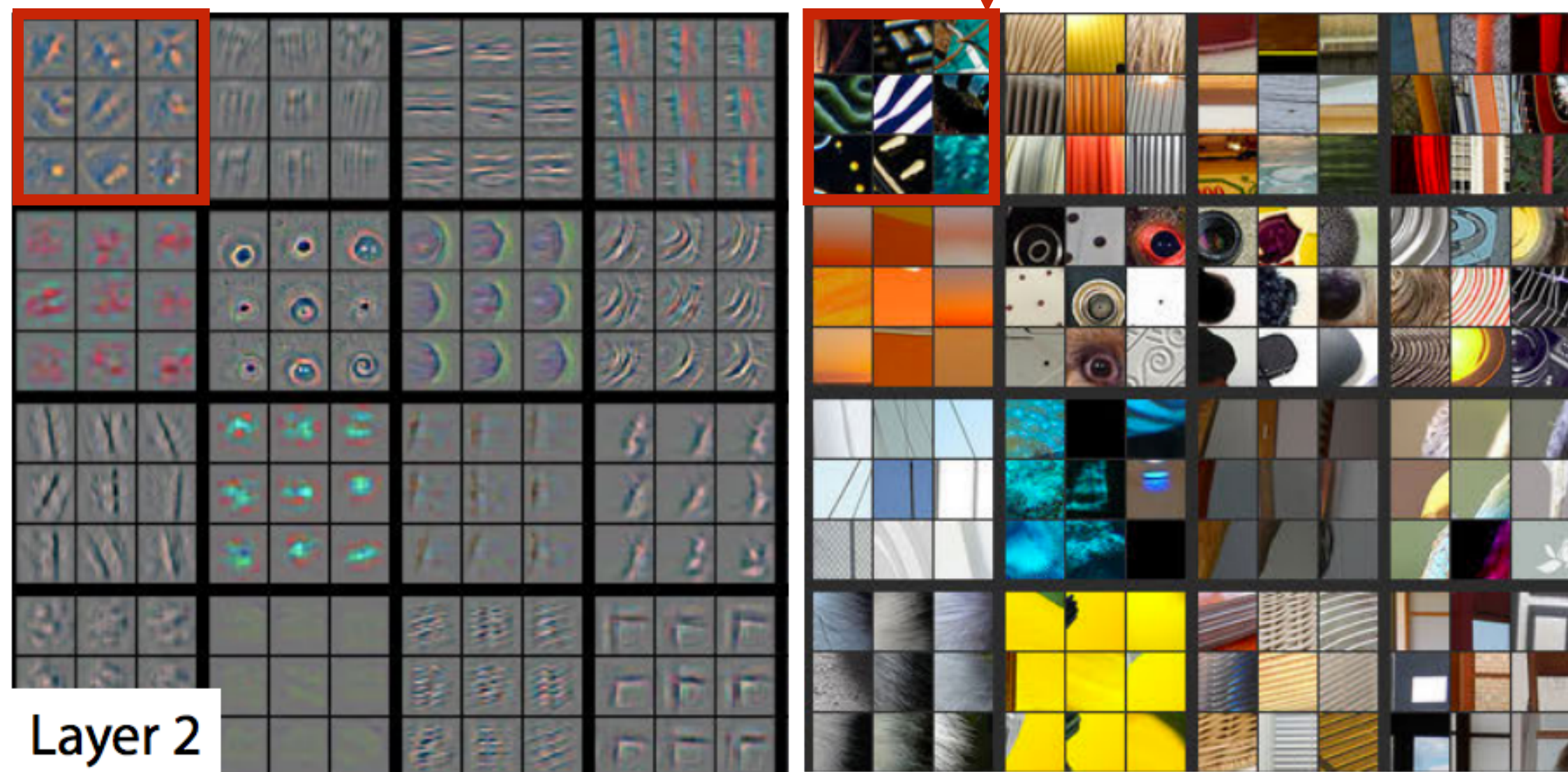
Why deep?
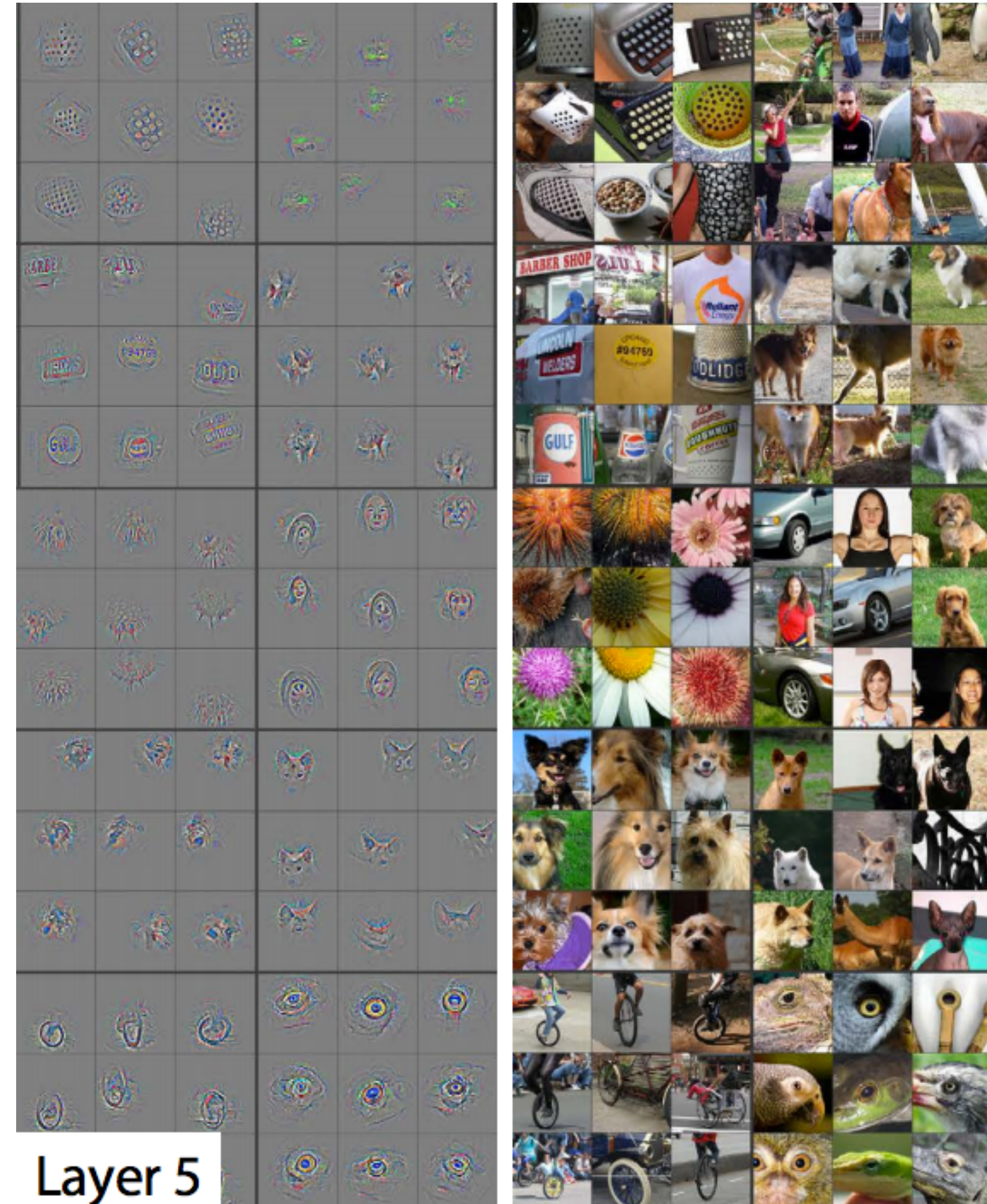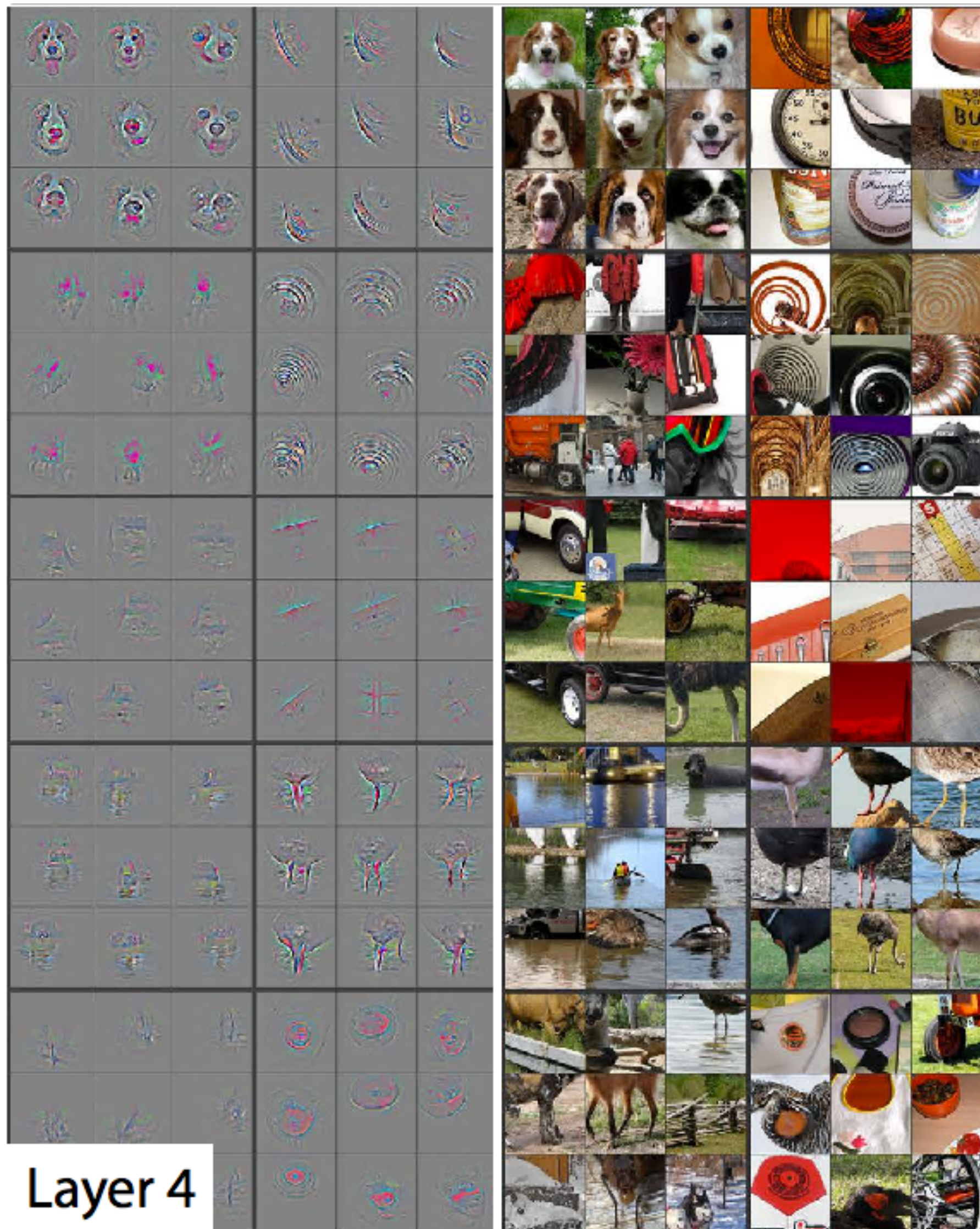
Layer 4

Layer 5

More recent image understanding networks
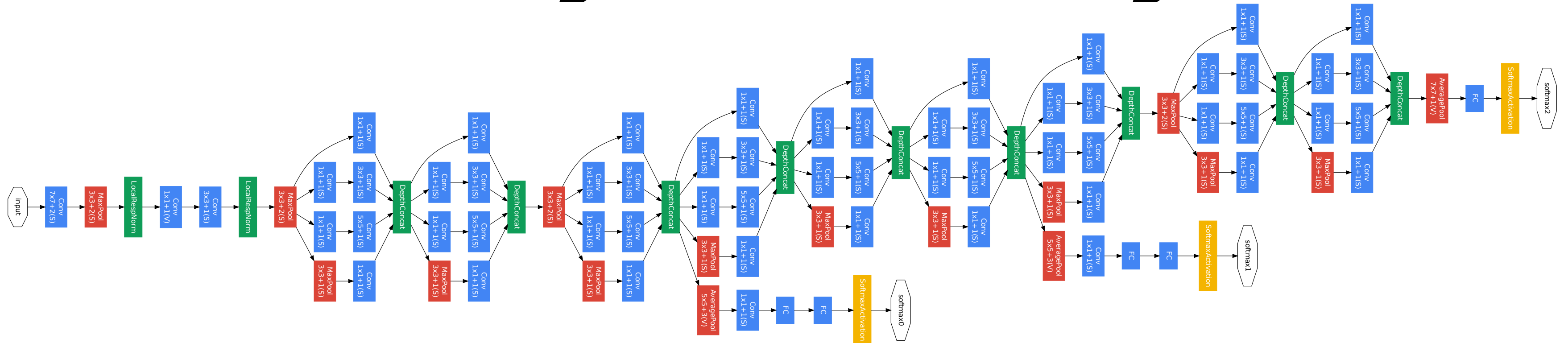
Inception (GoogleLeNet)

ResNet (34 layer version)

Convolution network

Upsampling network
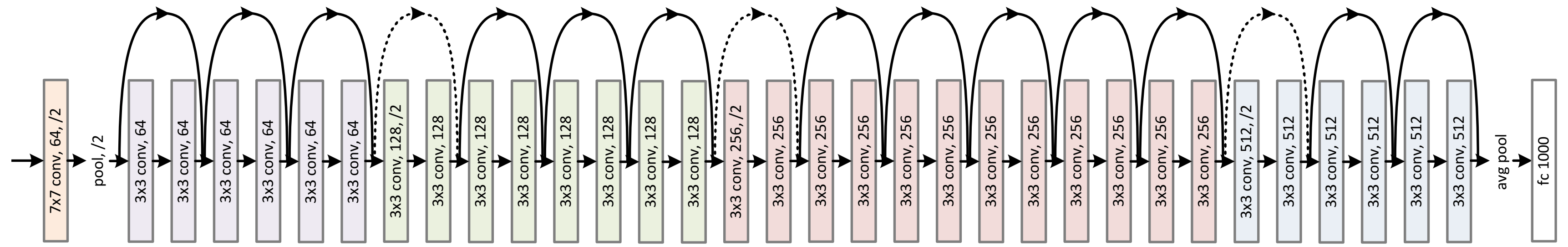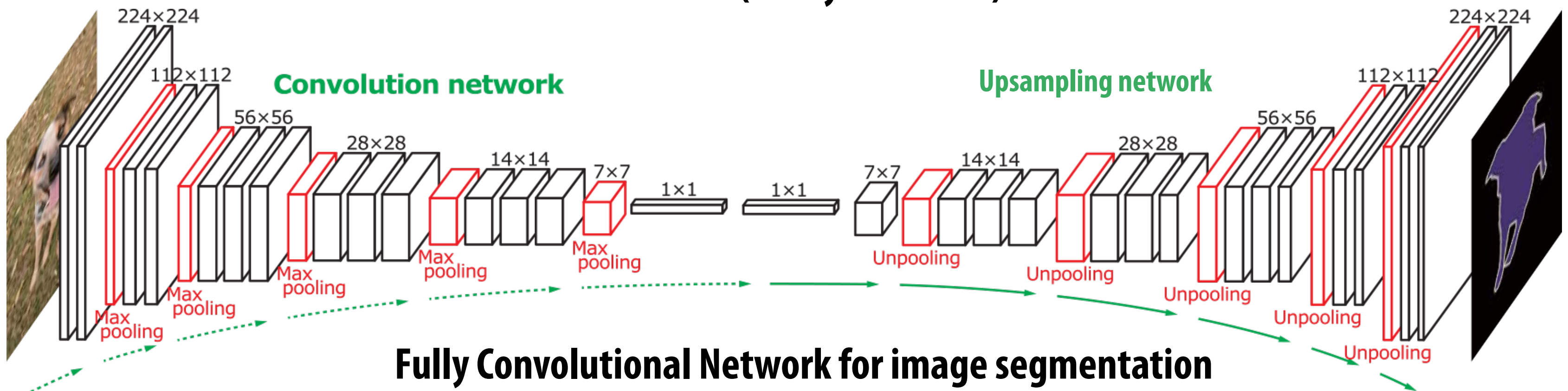
Fully Convolutional Network for image segmentation

Deep networks learn useful representations

- **Multi-scale learning of useful features for the task at hand**
  - Example on previous slides: subparts detectors emerged in network for object classification

- **But wait… how did you learn the values of all the weights?**
  - For today, assume the weights are given (today is about evaluating deep networks, not training them)

Efficiently implementing convolution layers

Dense matrix multiplication

```
float A[M][K];
float B[K][N];
float C[M][N];


// compute C += A * B
#pragma omp parallel for
for (int j=0; j<M; j++)
  for (int i=0; i<N; i++)
    for (int k=0; k<K; k++)
      C[j][i] += A[j][k] * B[k][i];
```
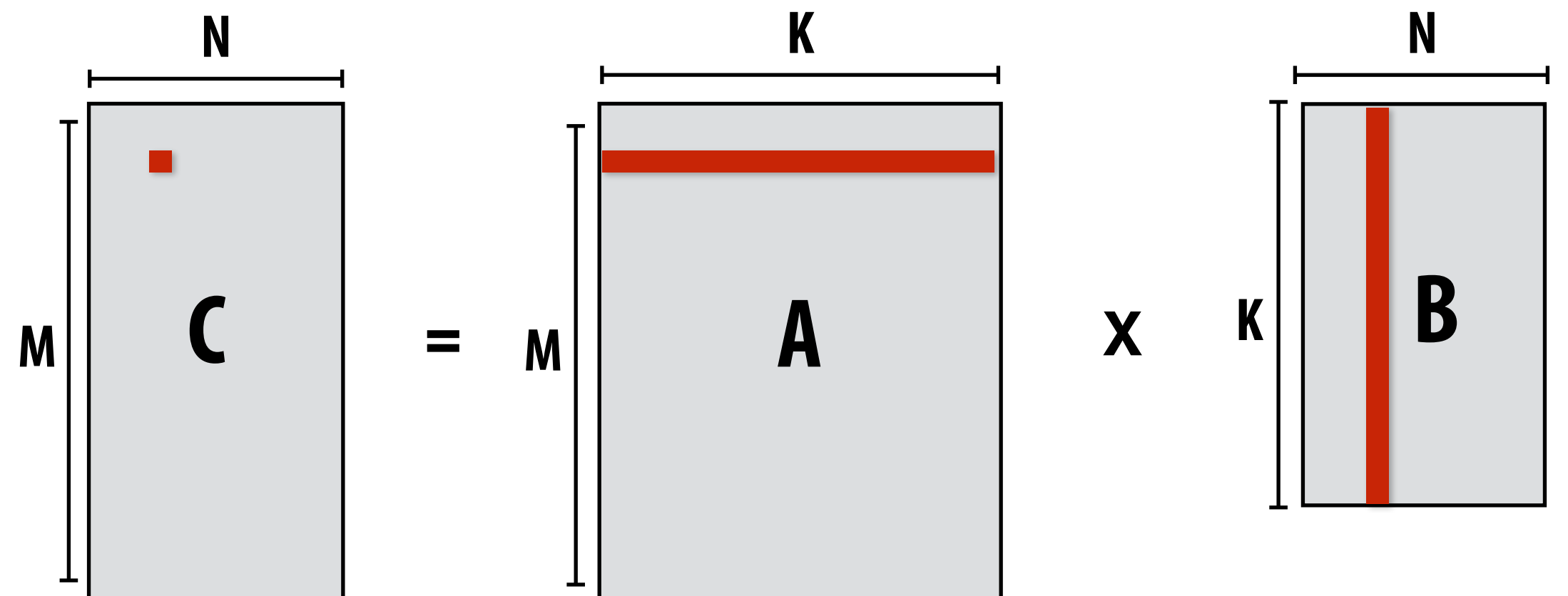

What is the problem with this implementation?

Low arithmetic intensity (does not exploit temporal locality in access to A and B)

Blocked dense matrix multiplication

```
float A[M][K];
float B[K][N];
float C[M][N];


// compute C += A * B
#pragma omp parallel for
for (int jblock=0; jblock<M; jblock+=BLOCKSIZE_J)
  for (int iblock=0; iblock<N; iblock+=BLOCKSIZE_I)
    for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
      for (int j=0; j<BLOCKSIZE_J; j++)
        for (int i=0; i<BLOCKSIZE_I; i++)
          for (int k=0; k<BLOCKSIZE_K; k++)
            C[jblock+j][iblock+i] += A[jblock+j][kblock+k] * B[kblock+k][iblock+i];
```
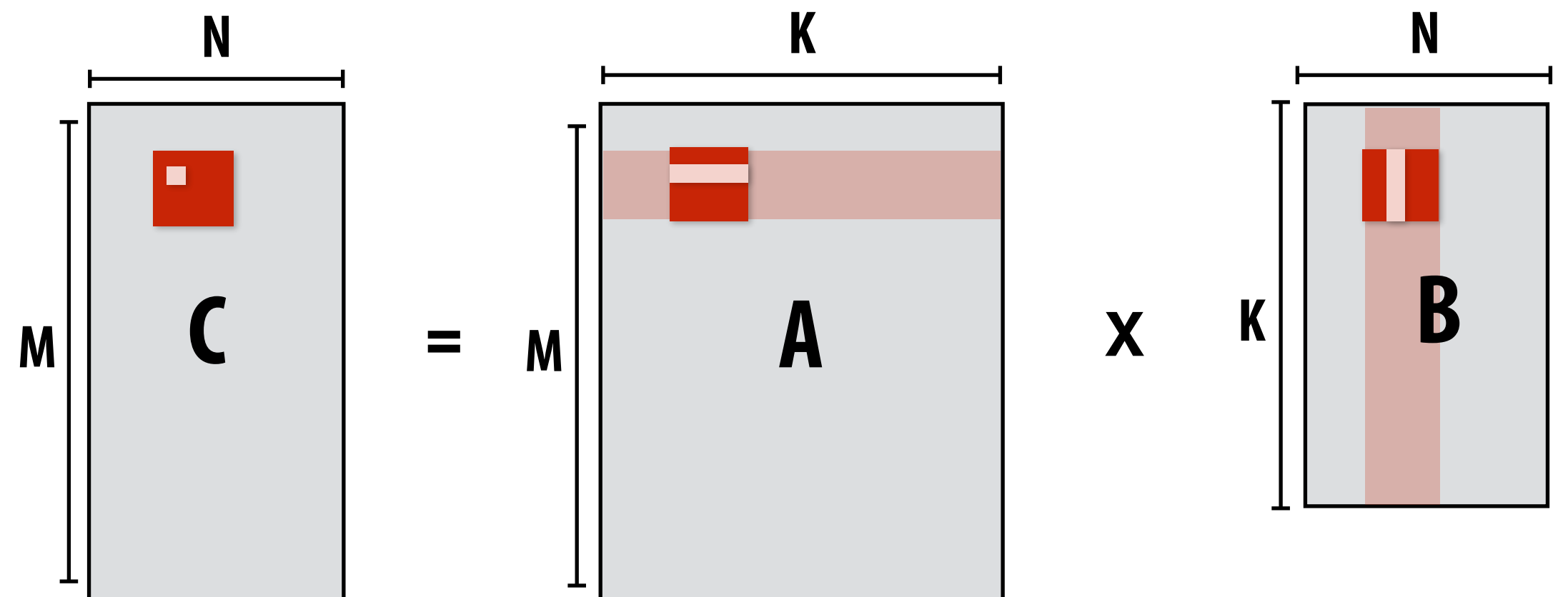

Idea: compute partial result for block of C while required blocks of A and B remain in cache (Assumes BLOCKSIZE chosen to allow block of A, B, and C to remain resident)

Self check: do you want as big a BLOCKSIZE as possible? Why?

Hierarchical blocked matrix mult

Exploit multiple levels of memory hierarchy

```
float A[M][K];

float B[K][N];

float C[M][N];


// compute C += A * B

#pragma omp parallel for

for (int jblock2=0; jblock2<M; jblock2+=L2_BLOCKSIZE_J)

  for (int iblock2=0; iblock2<N; iblock2+=L2_BLOCKSIZE_I)

    for (int kblock2=0; kblock2<K; kblock2+=L2_BLOCKSIZE_K)

      for (int jblock1=0; jblock1<L1_BLOCKSIZE_J; jblock1+=L1_BLOCKSIZE_J)

        for (int iblock1=0; iblock1<L1_BLOCKSIZE_I; iblock1+=L1_BLOCKSIZE_I)

          for (int kblock1=0; kblock1<L1_BLOCKSIZE_K; kblock1+=L1_BLOCKSIZE_K)

            for (int j=0; j<BLOCKSIZE_J; j++)

              for (int i=0; i<BLOCKSIZE_I; i++)

                for (int k=0; k<BLOCKSIZE_K; k++)

                    ...
```

Not shown: final level of "blocking" for register locality…

Blocked dense matrix multiplication (1)

Consider SIMD parallelism within a block
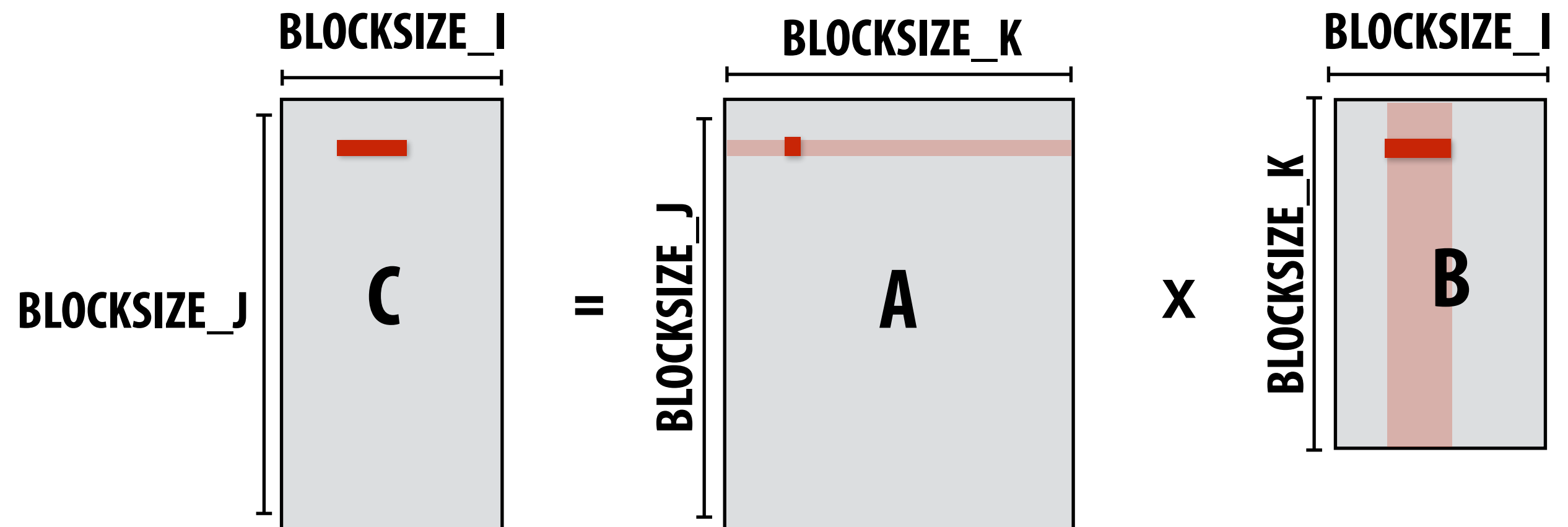

```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
    for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
        simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
        for (int k=0; k<BLOCKSIZE_K; k++) {
            // C = A*B + C
            simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in vector register
            simd_muladd(A_val, vec_load(&B[kblock+k][iblock+i]), C_accum);
        }
        vec_store(&C[jblock+j][iblock+i], C_accum);
    }
}
```

Vectorize i loop

Good: also improves spatial locality in access to B

Bad: working set increased by SIMD_WIDTH, still walking over B in large steps

Blocked dense matrix multiplication (2)
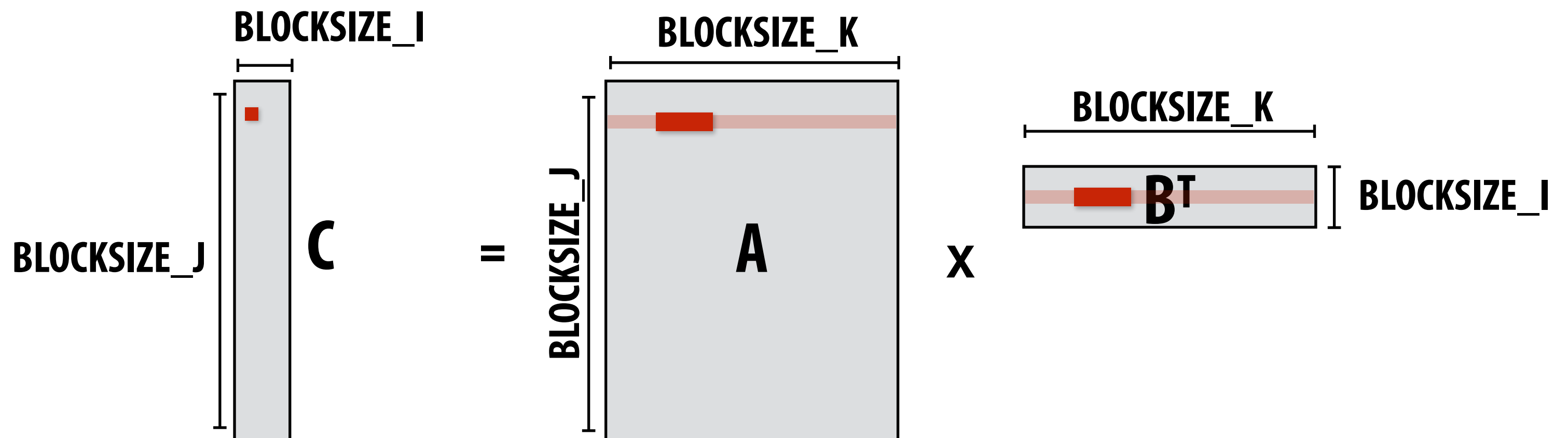

```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
       float C_scalar = C[jblock+j][iblock+i];
       // C_scalar += dot(row of A,row of B)
       for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
         C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][kblock+k]);
       }
       C[jblock+j][iblock+i] = C_scalar;
   }
```

Assume *i* dimension is small. Previous vectorization scheme (1) would not work well.

Pre-transpose block of B (copy block of B to temp buffer in transposed form)

Vectorize innermost loop

Blocked dense matrix multiplication (3)
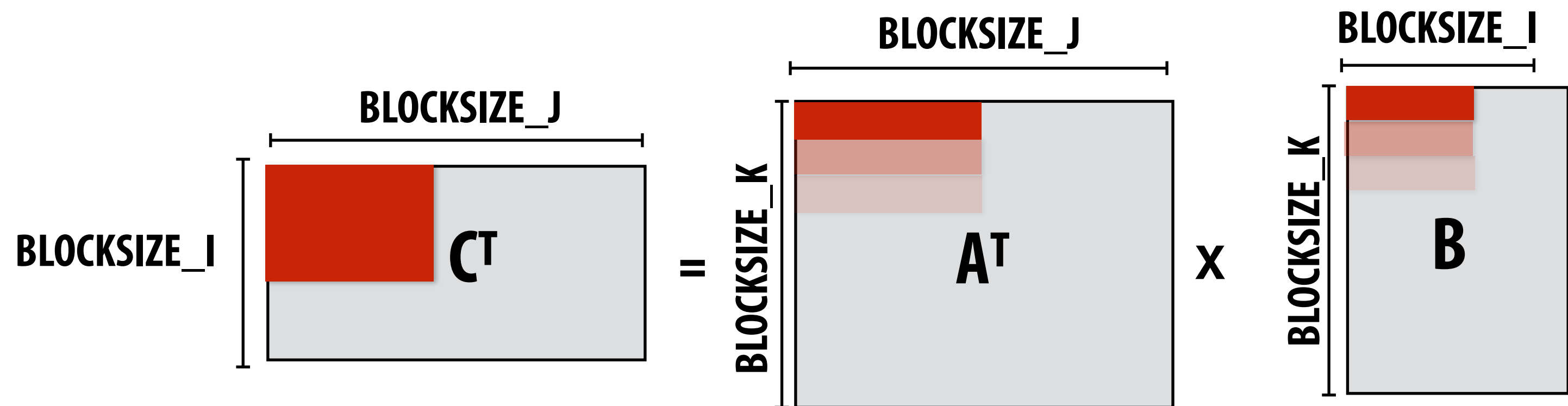

```
// assume blocks of A and C are pre-transposed as Atrans and Ctrans
for (int j=0; j<BLOCKSIZE_J; j+=SIMD_WIDTH) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {

      simd_vec C_accum[SIMD_WIDTH];
      for (int k=0; k<SIMD_WIDTH; k++)    // load C_accum for a SIMD_WIDTH x SIMD_WIDTH chunk of C^T
        C_accum[k] = vec_load(&Ctrans[iblock+i+k][jblock+j]);

      for (int k=0; k<BLOCKSIZE_K; k++) {
        simd_vec bvec = vec_load(&B[kblock+k][iblock+i]);
        for (int kk=0; kk<SIMD_WIDTH; kk++)  // innermost loop items not dependent
            simd_muladd(vec_load(&Atrans[kblock+k][jblock+j], splat(bvec[kk]), C_accum[kk]);
      }

      for (int k=0; k<SIMD_WIDTH; k++)
        vec_store(&Ctrans[iblock+i+k][jblock+j], C_accum[k]);
   }
}
```

Convolution as matrix-vector product

Construct matrix from elements of input image

X_{00}	X_{01}	X_{02}	X_{03}	...			
X_{10}	X_{11}	X_{12}	X_{13}	...			
X_{20}	X_{21}	X_{22}	X_{23}	...			
X_{30}	X_{31}	X_{32}	X_{33}	...			
...	...	...	...				

O(N) storage overhead for filter with N elements

Must construct input data matrix

$$3\text{x}3 = 9$$

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & x00 & x01 & 0 & x10 & x11 \\
& & & & & & & & \\
& & & \dots & & & & & \\
& & & & & & & &
\end{bmatrix}
\begin{bmatrix}
w_0 \\
w_1 \\
\vdots \\
w_8
\end{bmatrix}
$$

WxH

Note: 0-pad matrix

3x3 convolution as matrix-vector product

Construct matrix from elements of input image

X00	X01	X02	X03	...			
X10	X11	X12	X13	...			
X20	X21	X22	X23	...			
X30	X31	X32	X33	...			
...	...	...	...				

O(N) storage overhead for filter with N elements

Must construct input data matrix

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & x00 & x01 & 0 & x10 & x11 \\
0 & 0 & 0 & x00 & x01 & x02 & x10 & x11 & x12 \\
0 & 0 & 0 & x01 & x02 & x03 & x11 & x12 & x13 \\
& & & & \cdots & & & & \\
x00 & x01 & x02 & x10 & x11 & x12 & x20 & x21 & x22 \\
& & & & \cdots & & & &
\end{bmatrix}
\begin{bmatrix}
w_0 \\ w_1 \\ \vdots \\ w_8
\end{bmatrix}
$$

9

WxH

Note: 0-pad matrix

Multiple convolutions as matrix-matrix mult

X_{00}	X_{01}	X_{02}	X_{03}	...			
X_{10}	X_{11}	X_{12}	X_{13}	...			
X_{20}	X_{21}	X_{22}	X_{23}	...			
X_{30}	X_{31}	X_{32}	X_{33}	...			
...	...	...	...				

9

num filters

WxH

| 0 | 0 | 0 | 0 | x00 | x01 | 0 | x10 | x11 |

| 0 | 0 | 0 | x00 | x01 | x02 | x10 | x11 | x12 |

| 0 | 0 | 0 | x01 | x02 | x03 | x11 | x12 | x13 |

...

| x00 | x01 | x02 | x10 | x11 | x12 | x20 | x21 | x22 |

...

$$\begin{bmatrix} w_{00} & w_{01} & w_{02} & \cdots & w_{0N} \\ w_{10} & w_{11} & w_{12} & \cdots & w_{0N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{80} & w_{81} & w_{82} & \cdots & w_{8N} \end{bmatrix}$$

Multiple convolutions on multiple input channels
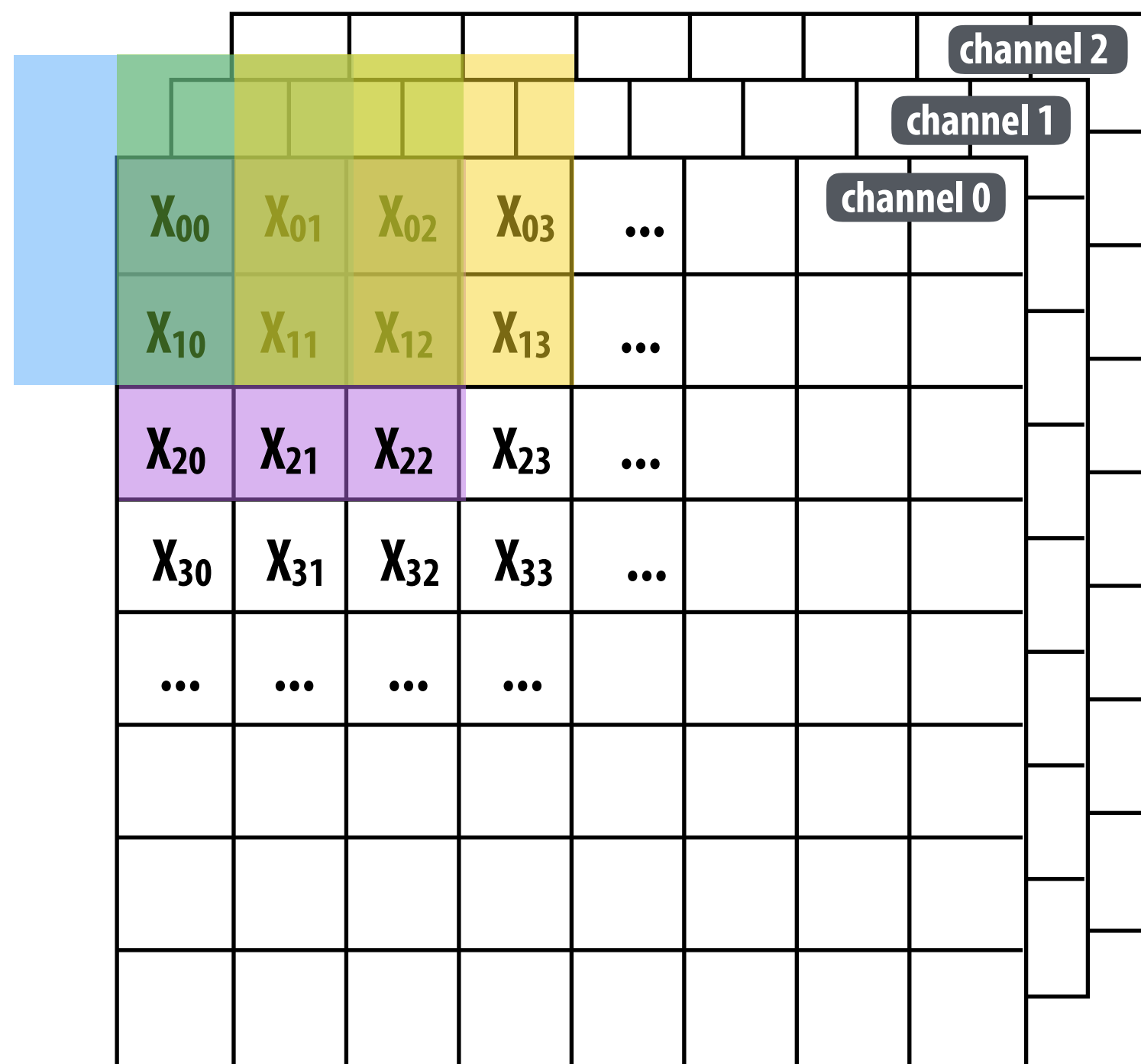
For each filter, sum responses over input channels

Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data
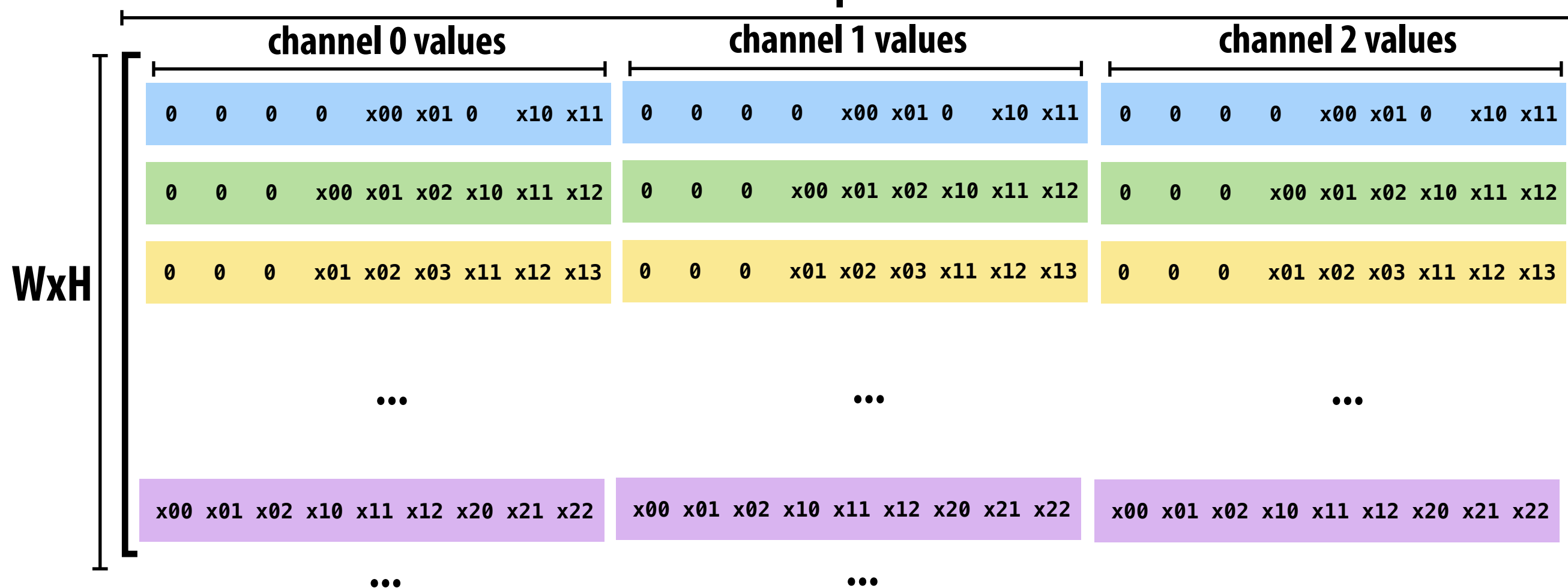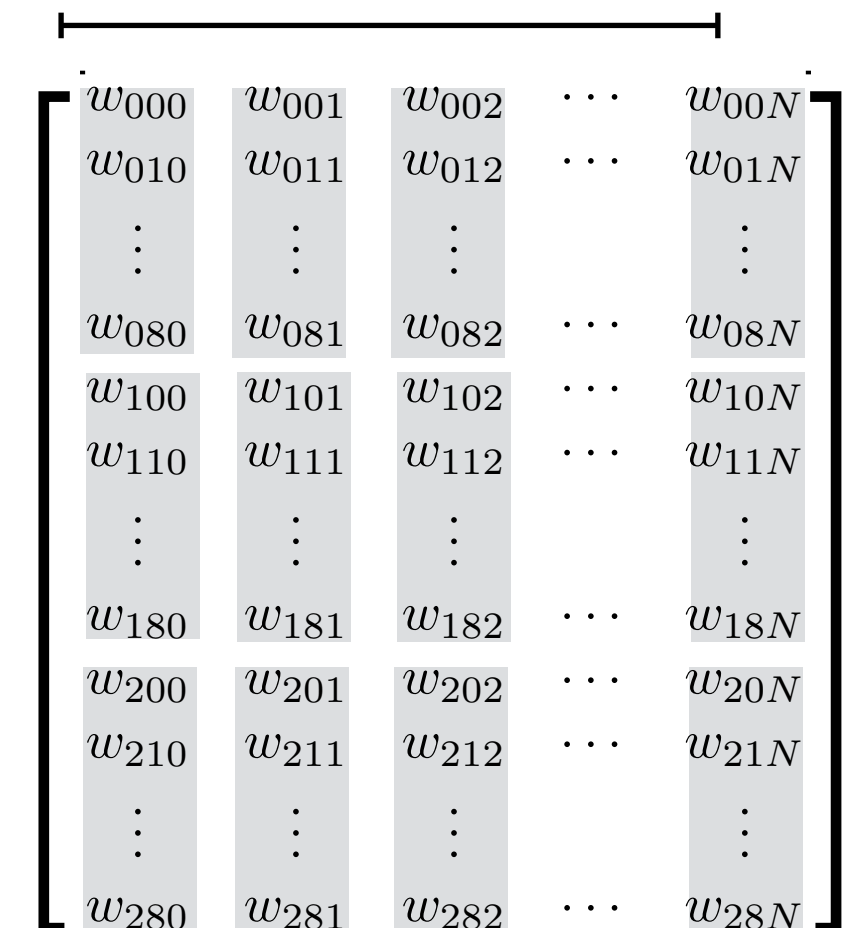
Direct implementation of conv layer

```
float input[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];

float output[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];

float layer_weights[LAYER_NUM_FILTERS][LAYER_CONVY][LAYER_CONVX][INPUT_DEPTH];


// assumes convolution stride is 1
for (int img=0; img<IMAGE_BATCH_SIZE; img++)
   for (int j=0; j<INPUT_HEIGHT; j++)
      for (int i=0; i<INPUT_WIDTH; i++)
         for (int f=0; f<LAYER_NUM_FILTERS; f++) {
            output[img][j][i][f] = 0.f;
            for (int kk=0; kk<INPUT_DEPTH; kk++)          // sum over filter responses of input channels
               for (int jj=0; jj<LAYER_FILTER_Y; jj++)    // spatial convolution (Y)
                  for (int ii=0; ii<LAYER_FILTER_X; ii+)  // spatial convolution (X)
                     output[img][j][i][f] += layer_weights[f][jj][ii][kk] * input[img][j+jj][i+ii][kk];
         }
```

Seven loops with significant input data reuse: reuse of filter weights (during convolution), and reuse of input values (across different filters)

Avoids O(N) footprint increase by avoiding materializing input matrix
In theory loads O(N) times less data (potentially higher arithmetic intensity… but matrix mult is typically compute-bound)
But must roll your own highly optimized implementation of complicated loop nest.

Convolutional layer in Halide

```cpp
int in_w, in_h, in_ch = 4;             // input params: assume initialized

Func in_func;                          // assume input function is initialized

int num_f, f_w, f_h, pad, stride;      // parameters of the conv layer

Func forward = Func("conv");
Var x, y, z, n;                        // n is minibatch dimension

// This creates a padded input to avoid checking boundary
// conditions while computing the actual convolution
f_in_bound = BoundaryConditions::repeat_edge(in_func, 0, in_w, 0, in_h);

// Create buffers for layer parameters
Halide::Buffer<float> W(f_w, f_h, in_ch, num_f)
Halide::Buffer<float> b(num_f);

// domain of summation for filter with W x H x in_ch
RDom r(0, f_w, 0, f_h, 0, in_ch);

// Initialize to bias
forward(x, y, z, n) =  b(z);
forward(x, y, z, n) += W(r.x, r.y, r.z, z) *
                       f_in_bound(x*stride + r.x – pad, y*stride + r.y – pad, r.z, n);
```

Consider scheduling this seven-dimensional loop nest!

Different layers of a single DNN may benefit from unique scheduling strategies

Throughput: Input-Specialized Schedules
(relative to best-on-average schedule)

VGG-16 conv layers

[Figure credit: Mullapudi et al. 2016]

**Notice sizes of weights and activations in this network:
(and consider SIMD widths of modern machines). Ug!**

Optimization of Manually-Authored Schedules

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

LENSBLUR

MAXFILTER

NL MEANS

Schedule development Time (minutes)

= Programmer 1   = Programmer 2   = Auto-s

Many efforts to automatically schedule key DNN operations

Tensor Comprehensions

Halide IR → Range Inference and Specialization

Polyhedral Transformations ↔ Polyhedral IR

C Reference Impl. → CUDA Kernel → ATen Expr.

LLVM IR → CUDA Module → libTHC.so

Exec

Deep Learning Compiler

nGraph
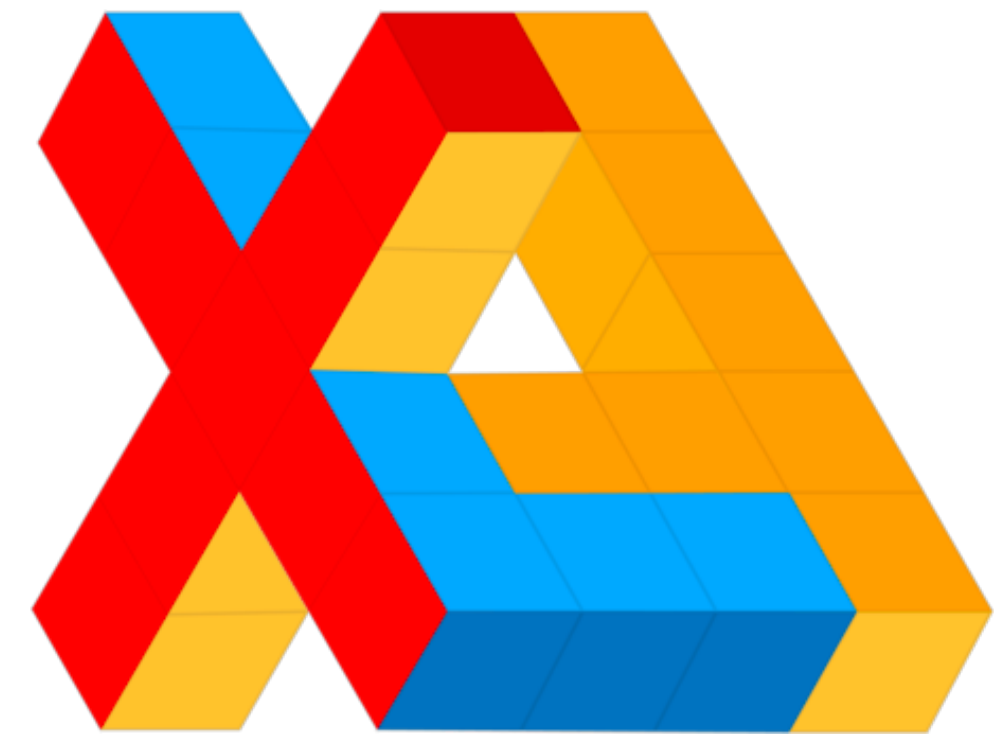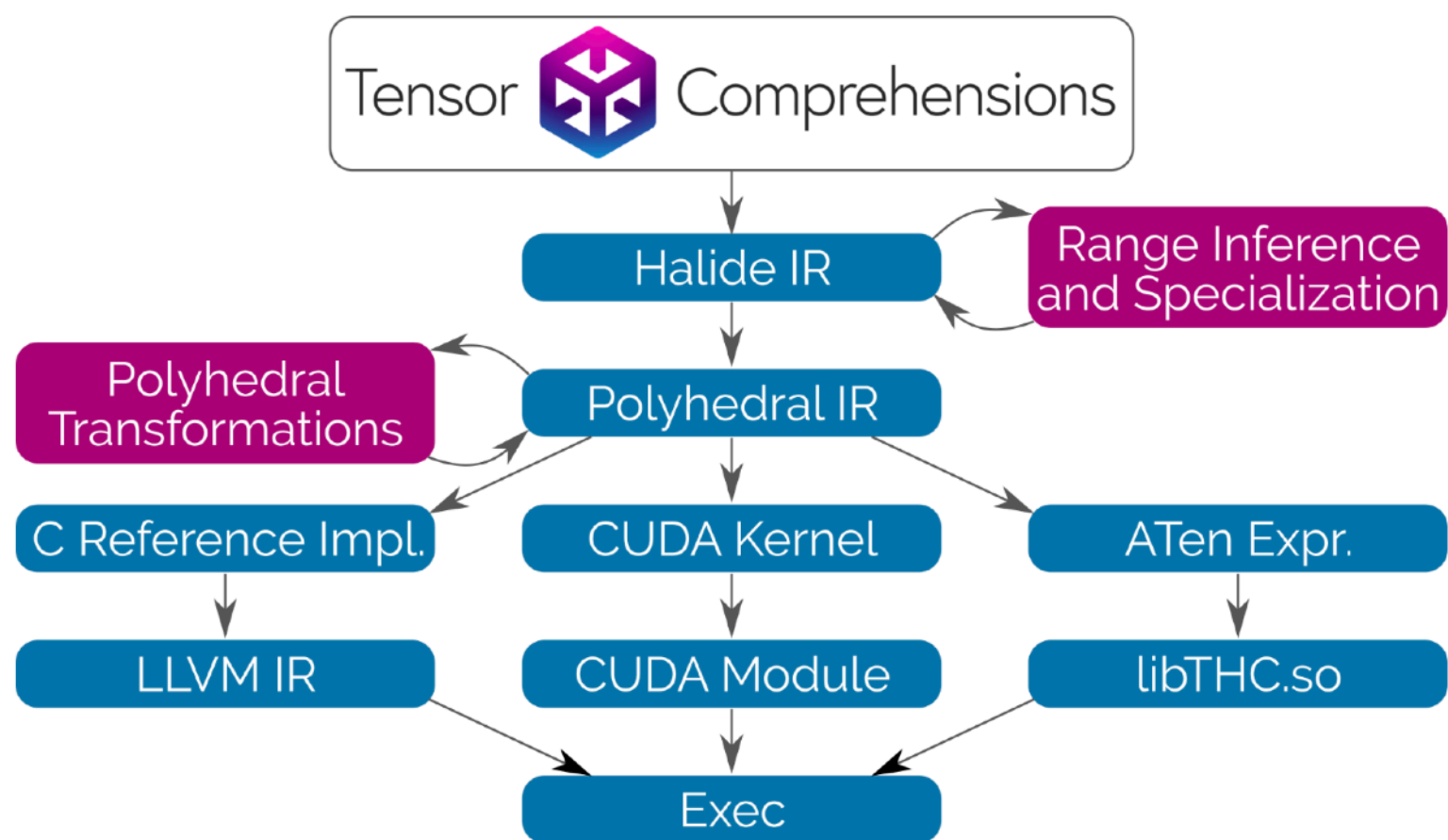
tvm Open Deep Learning Compiler Stack

| license | Apache 2.0 | build | passing |

Documentation | Contributors | Community | Release Notes

TVM is a compiler stack for deep learning systems. It is designed to close the gap between the productivity-focused deep learning frameworks, and the performance- and efficiency-focused hardware backends. TVM works with deep learning frameworks to provide end to end compilation to different backends. Checkout the tvm stack homepage for more information.
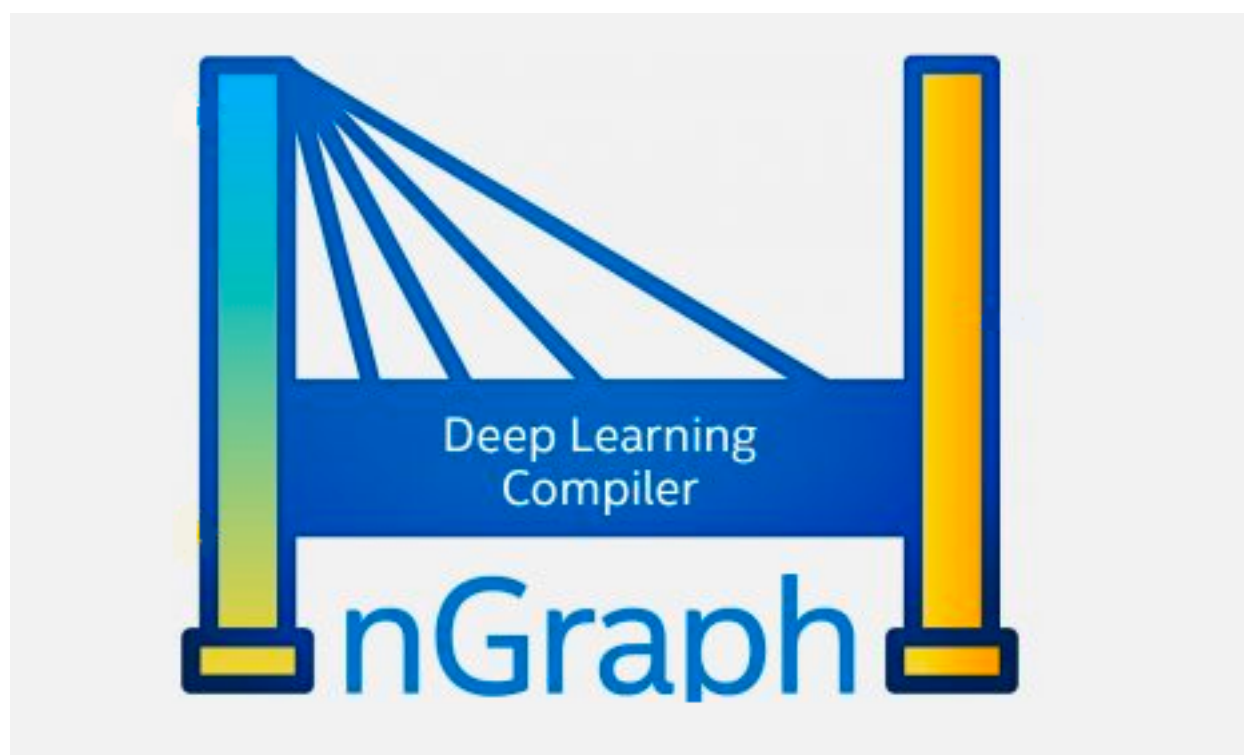
NVIDIA TensorRT

Programmable Inference Accelerator

Halide autoscheduler produces efficient DNN layer implementations (for CPUs)

Use large database of programs to **learn to predict** performance of Halide program+schedule. Then search for best performing programs.

Resnet50 throughput in images/sec (higher is better)
16 threads, AVX-512, batch size 1

[Adams et al. 2019]
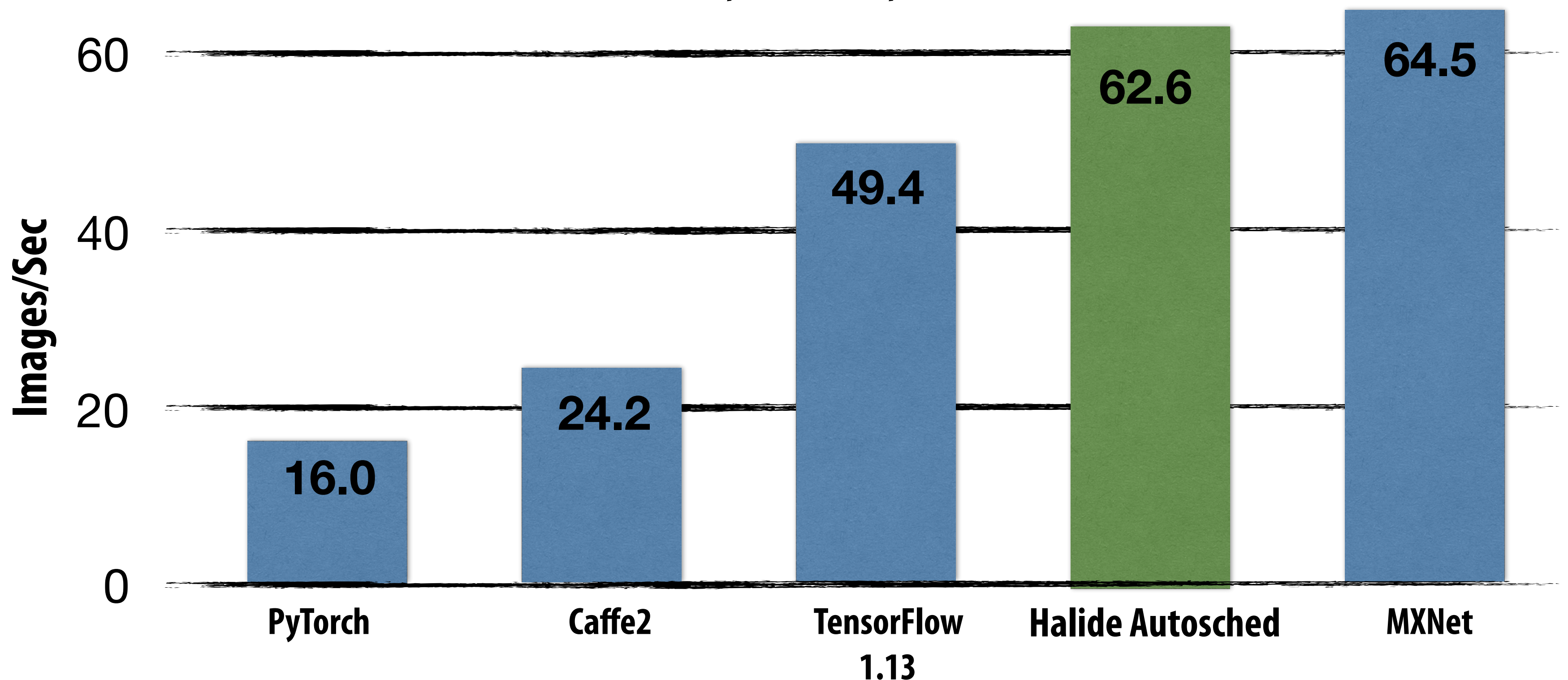
Reminder: energy cost of data access

Significant fraction of energy expended moving data to processor ALUs

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

Estimates for 45nm process
[Source: Mark Horowitz]

1   10

Reducing network footprint

- **Early DNN designs: large storage cost for model parameters**
    - AlexNet model: ~200 MB
    - VGG-16 model: ~500 MB
    - ResNet-50: 102 MB
    - Inception-v3: 91 MB

- **In many modern DNNs, activations (intra-layer intermediate buffers) require more storage than weights**
    - So bandwidth is often due to reading/writing intermediates

Is there an opportunity for compression?

"Pruning" (sparsifying) a network

x_0

x_1

x_2

x_3

w_0
w_1
b
w_2
w_3
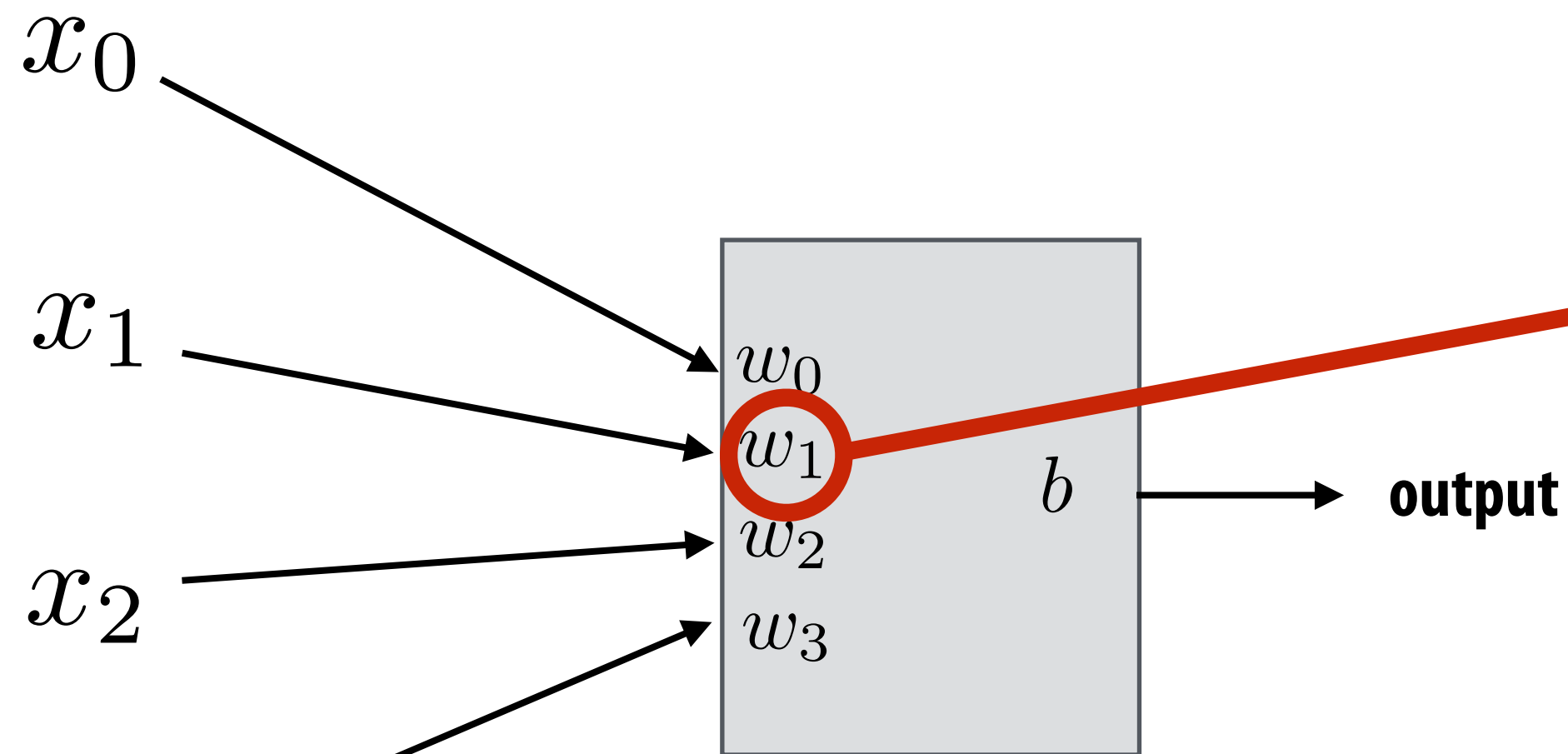
output

If weight is near zero, then corresponding input has little impact on output of neuron.

$$f\left(\sum_i x_i w_i + b\right)$$

$$f(x) = max(0, x)$$

"Pruning" (sparsifying) a network

x_0

x_1

w_0

b → output

w_2

x_2

w_3

x_3
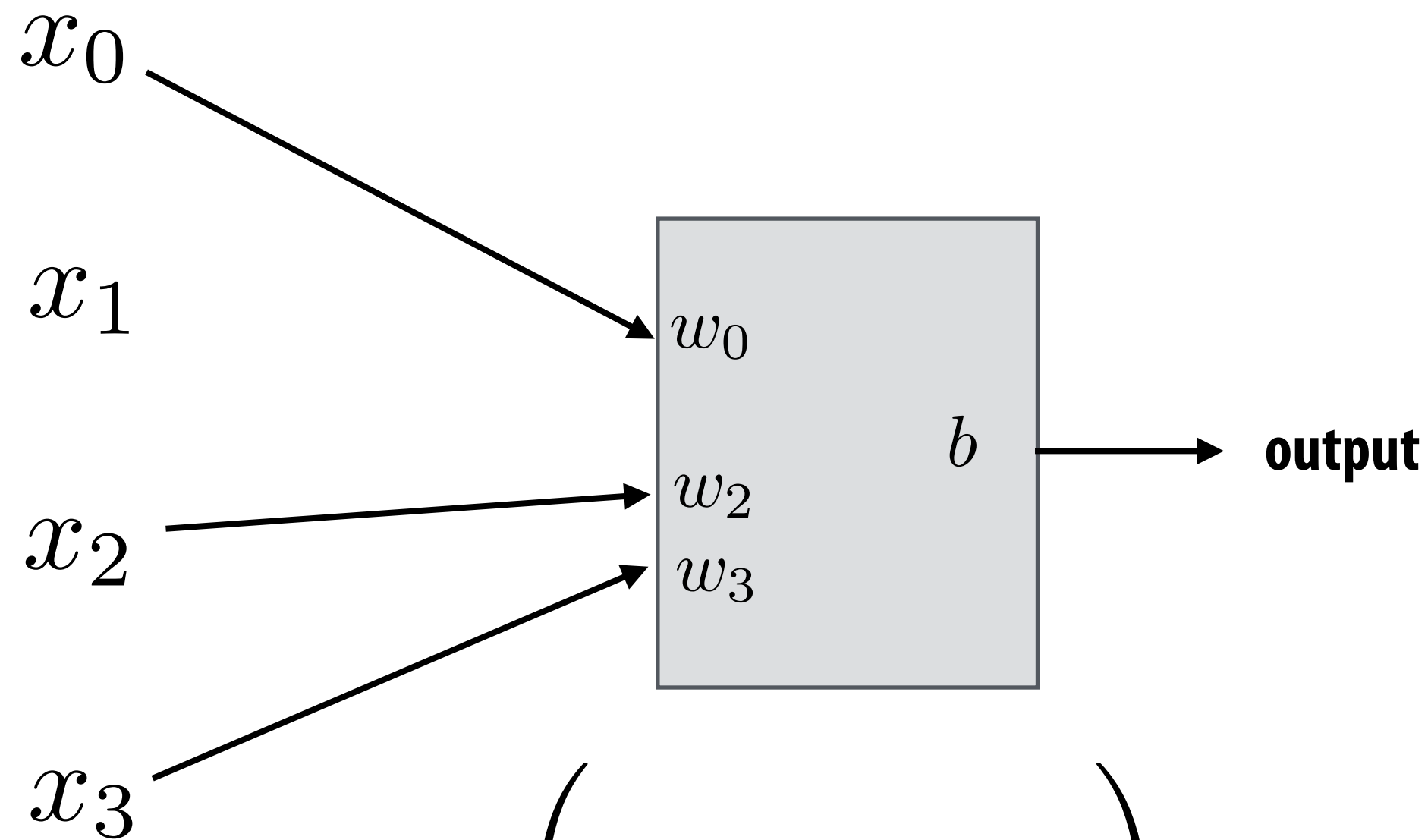
$$f\left(\sum_i x_i w_i + b\right)$$

$$f(x) = max(0, x)$$

Idea: prune connections with near zero weight

Remove entire units if all connections are pruned.

Representing "sparsified" networks

Step 1: prune low-weight links (iteratively retrain network, then prune)

- **Store weight matrices in compressed sparse row (CSR) format**

```
Indices    1    4    9   ...
Value     1.8  0.5  2.1
```

| 0 | 1.8 | 0 | 0 | 0.5 | 0 | 0 | 0 | 0 | 1.1 | ... |

Reduce storage over head of indices by delta encoding them to fit in 8 bits

```
Indices    1    3    5   ...
Value     1.8  0.5  2.1
```

Efficiently storing the surviving connections

Step 2: Weight sharing: make surviving connections share a small set of weights
- **Cluster weights via k-means clustering**
- **Compress weights by only storing index of assigned cluster (lg(k) bits)**
- **This is a form of lossy compression**

weights
(32 bit float)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

cluster →

cluster index
(2 bit uint)

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

centroids

3:	2.00
2:	1.50
1:	0.00
0:	-1.00
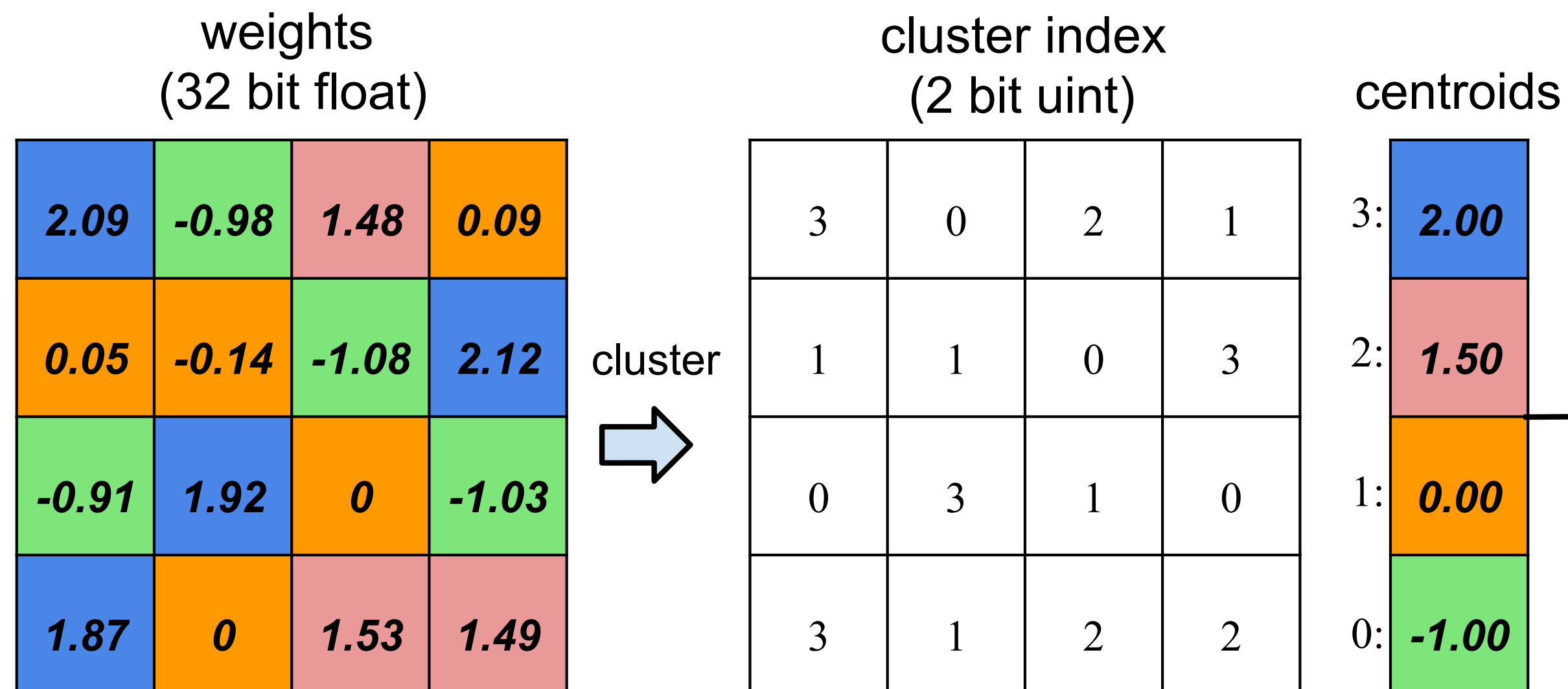
Step 3: Huffman encode quantized weights and CSR indices (lossless compression)

VGG-16 sparsification

Large savings in fully connected layers due to combination of pruning, quantization, Huffman encoding *

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%(13×)	6.4	4.1	5	3.1	3.2% (**31**×)	2.05% (**49**×)

P = connection pruning (prune low weight connections)

Q = quantize surviving weights (using shared weights)

H = Huffman encode

ImageNet Image Classification Performance

	Top-1 Error	Top-5 Error	Model size	
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	49×

* Benefits of automatic pruning apply mainly to fully connected layers, but unfortunately many modern networks are dominated by costs of convolutional layers

Compressing weights (and activations)

- **Many efforts to use low precision values for DNN weights and intermediate activations**

- **In the extreme case: 1-bit**

XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

Mohammad Rastegari[†], Vicente Ordonez[†], Joseph Redmon[*], Ali Farhadi[†*]

Allen Institute for AI[†], University of Washington[*]
{mohammadr,vicenteor}@allenai.org
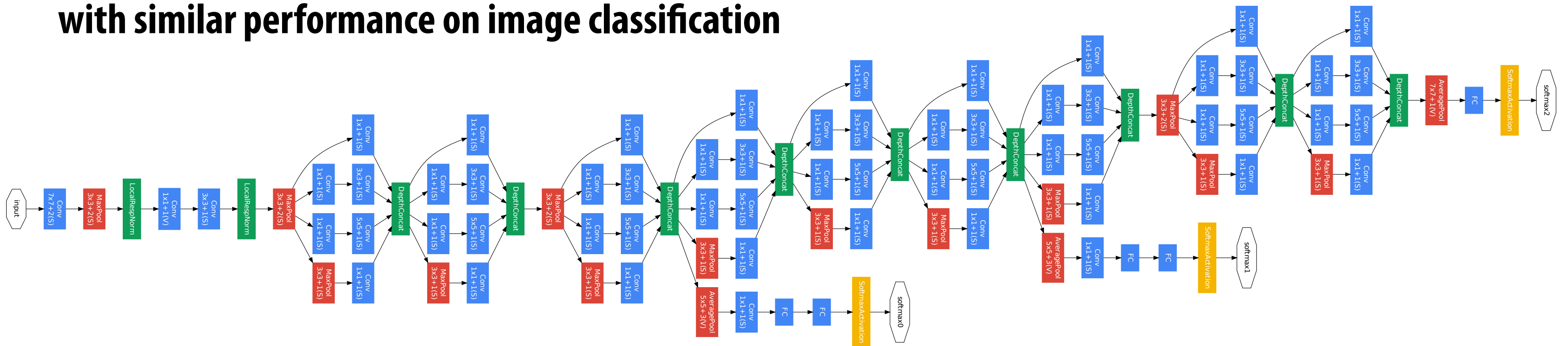{pjreddie,ali}@cs.washington.edu

Abstract. We propose two efficient approximations to standard convolutional neural networks: Binary-Weight-Networks and XNOR-Networks. In Binary-Weight-Networks, the filters are approximated with binary values resulting in $32\times$ memory saving. In XNOR-Networks, both the filters and the input to convolutional layers are binary. XNOR-Networks approximate convolutions using primarily binary operations. This results in $58\times$ faster convolutional operations (in terms of number of the high precision operations) and $32\times$ memory savings. XNOR-Nets offer the possibility of running state-of-the-art networks on CPUs (rather than GPUs) in real-time. Our binary networks are simple, accurate, efficient, and work on challenging visual tasks. We evaluate our approach on the ImageNet classification task. The classification accuracy with a Binary-Weight-Network version of AlexNet is the same as the full-precision AlexNet. We compare our method with recent network binarization methods, BinaryConnect and BinaryNets, and outperform these methods by large margins on ImageNet, more than 16% in top-1 accuracy. Our code is available at: http://allenai.org/plato/xnornet.

This a great example of non-domain-specific vs. domain-specific approach to innovation

Leveraging domain-knowledge: more efficient topologies (aka better algorithm design)

- **Original DNNs for image recognition where over-provisioned**
  - Large filters, many filters

- **Modern DNNs designs are hand-designed to be sparser**

  SqueezeNet: [Iandola 2017] Reduced number of parameters in AlexNet by 50x, with similar performance on image classification

Inception v1 (GoogleLeNet) — 27 total layers, 7M parameters

ResNet (34 layer version)

Modular network designs

Inception v4

Softmax — Output: 1000

Dropout (keep 0.8) — Output: 1536

Avarage Pooling — Output: 1536

3 x Inception-C — Output: 8x8x1536

Reduction-B — Output: 8x8x1536

7 x Inception-B — Output: 17x17x1024

Reduction-A — Output: 17x17x1024

4 x Inception-A — Output: 35x35x384

Stem — Output: 35x35x384

Input (299x299x3) — 299x299x3

A block

Filter concat

1x1 Conv (96) · Avg Pooling · 1x1 Conv (96) · 3x3 Conv (96) · 1x1 Conv (64) · 3x3 Conv (96) · 3x3 Conv (96) · 1x1 Conv (64)

Filter concat

B block

Filter concat

1x1 Conv (128) · Avg Pooling · 1x1 Conv (384) · 1x1 Conv (256) · 1x7 Conv (224) · 1x7 Conv (192) · 7x1 Conv (256) · 1x7 Conv (224) · 7x1 Conv (224) · 1x7 Conv (192) · 1x1 Conv (192)

Filter concat

Inception stem

Filter concat — 35x35x384

3x3 Conv (192 V)   MaxPool (stride=2 V)

Filter concat — 71x71x192

3x3 Conv (96 V)

1x7 Conv (64)

7x1 Conv (64)

1x1 Conv (64)

3x3 Conv (96 V)

1x1 Conv (64)

Filter concat — 73x73x160

3x3 MaxPool (stride 2 V)   3x3 Conv (96 stride 2 V)

3x3 Conv (64) — 147x147x64

3x3 Conv (32 V) — 147x147x32

3x3 Conv (32 stride 2 V) — 149x149x32

Input (299x299x3) — 299x299x3

ResNet

	VGG-19	34-layer plain	34-layer residual

VGG-19

output size: 224
- image
- 3x3 conv, 64
- 3x3 conv, 64

output size: 112
- pool, /2
- 3x3 conv, 128
- 3x3 conv, 128

output size: 56
- pool, /2
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256

output size: 28
- pool, /2
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512

output size: 14
- pool, /2
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512

output size: 7
- pool, /2

output size: 1
- fc 4096
- fc 4096
- fc 1000

34-layer plain

- image
- 7x7 conv, 64, /2
- pool, /2
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 128, /2
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 256, /2
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 512, /2
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
- avg pool
- fc 1000

34-layer residual

- image
- 7x7 conv, 64, /2
- pool, /2
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 64
- 3x3 conv, 128, /2
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 128
- 3x3 conv, 256, /2
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 256
- 3x3 conv, 512, /2
- 3x3 conv, 512
- 3x3 conv, 512
- 3x3 conv, 512
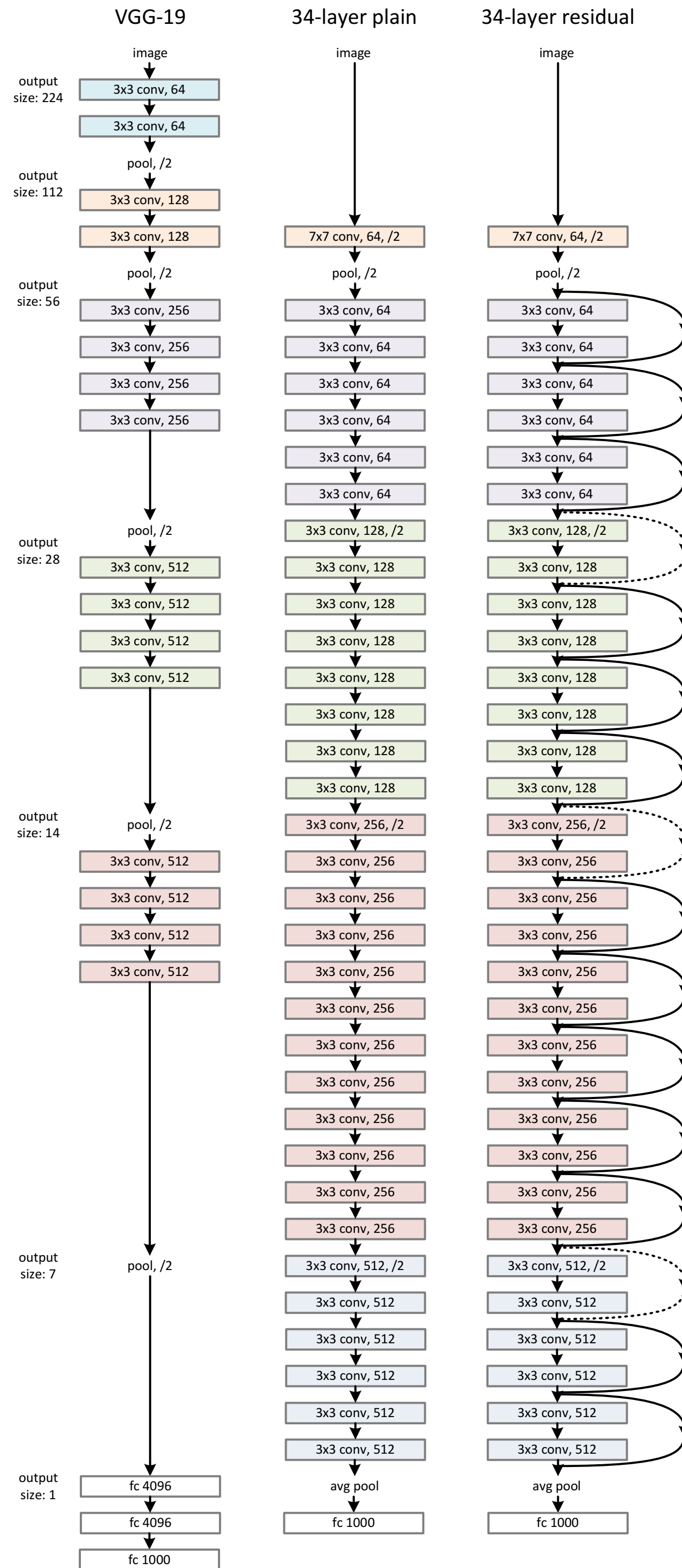- 3x3 conv, 512
- 3x3 conv, 512
- avg pool
- fc 1000
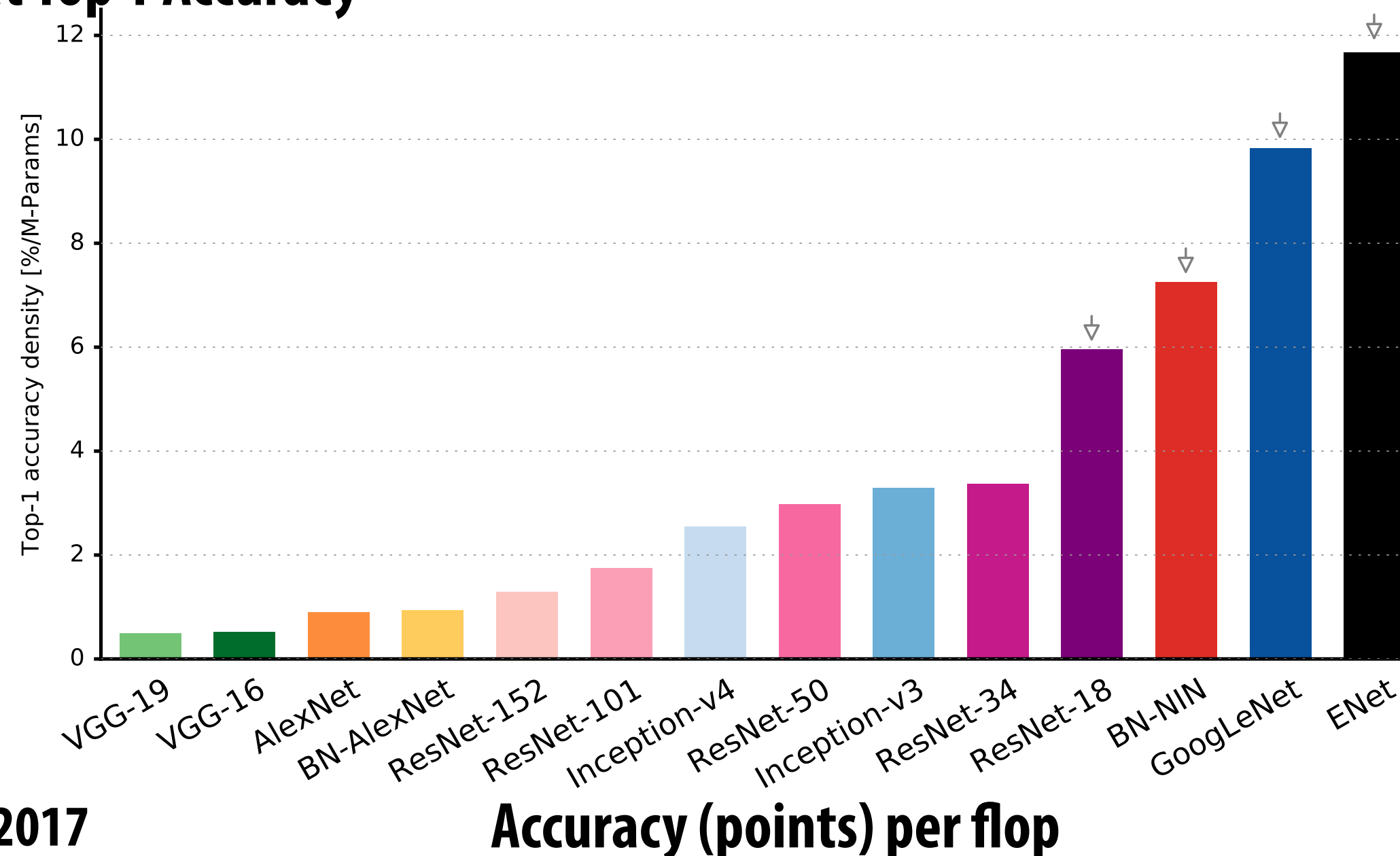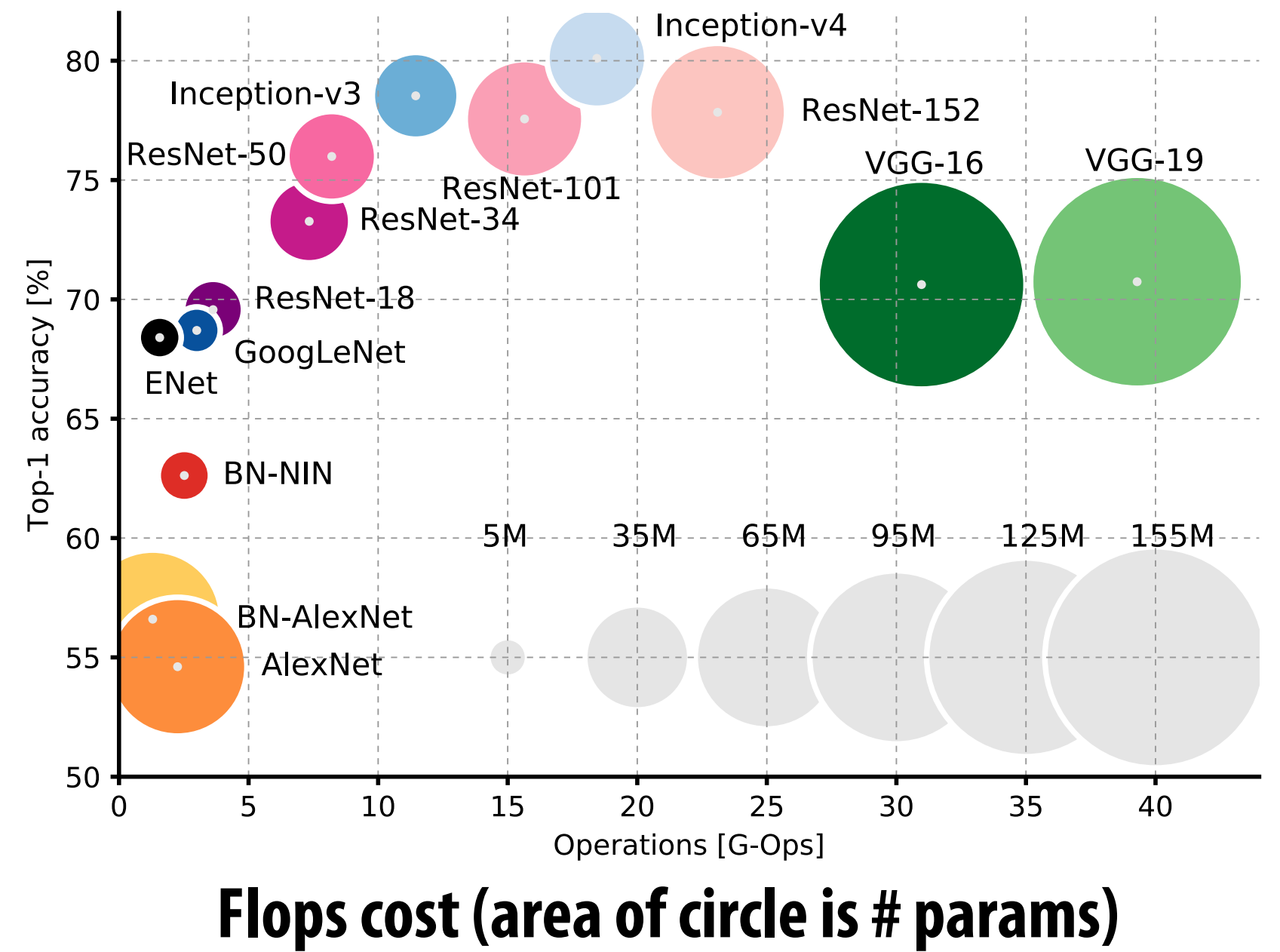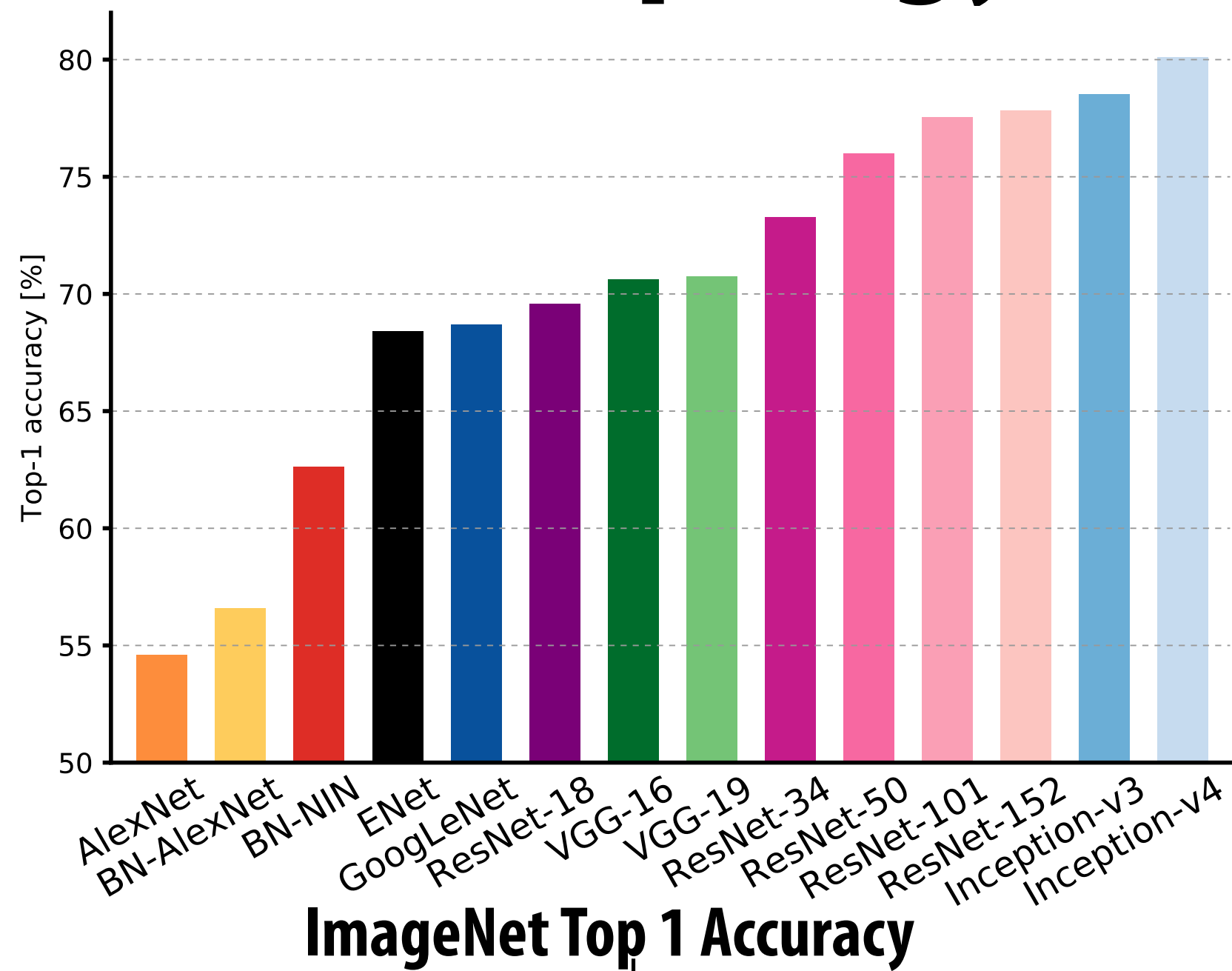

Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

Effect of topology innovation

ImageNet Top 1 Accuracy

Flops cost (area of circle is # params)

Accuracy (points) per flop

Improving accuracy/cost (image classification)

2014 → 2017   ~ 25x improvement in cost at similar accuracy

	ImageNet Top-1 Accuracy	Num Params	Cost/image (MADDs)	
VGG-16	71.5%	138M	15B	[2014]
GoogleNet	70%	6.8M	1.5B	[2015]
ResNet-18	73% *	11.7M	1.8B	[2016]
MobileNet-224	70.5%	4.2M	0.6B	[2017]

* 10-crop results (ResNet 1-crop results are similar to other DNNs in this table)

Depthwise separable convolution

Main idea: factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:
- NUM_CHANNELS 3x3x1 convolutions for each input channel
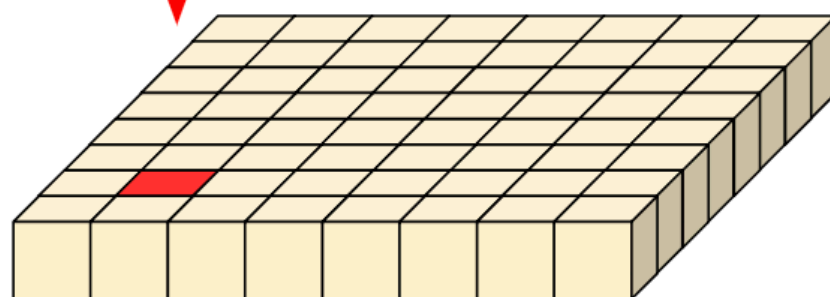- And NUM_FILTERS 1x1xNUM_CHANNELS convolutions to combine the results
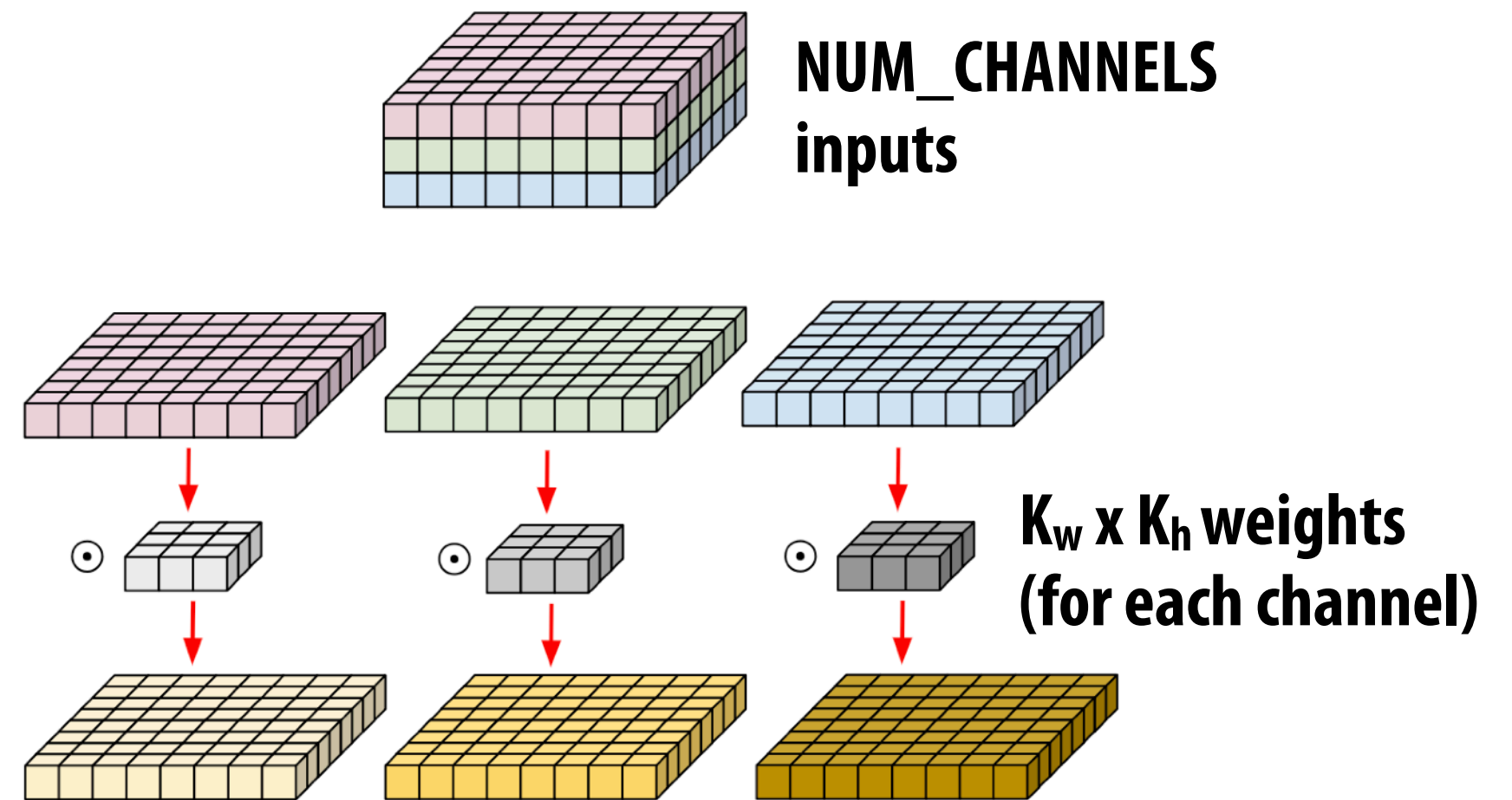
Convolution Layer

Depthwise Separable Conv Layer
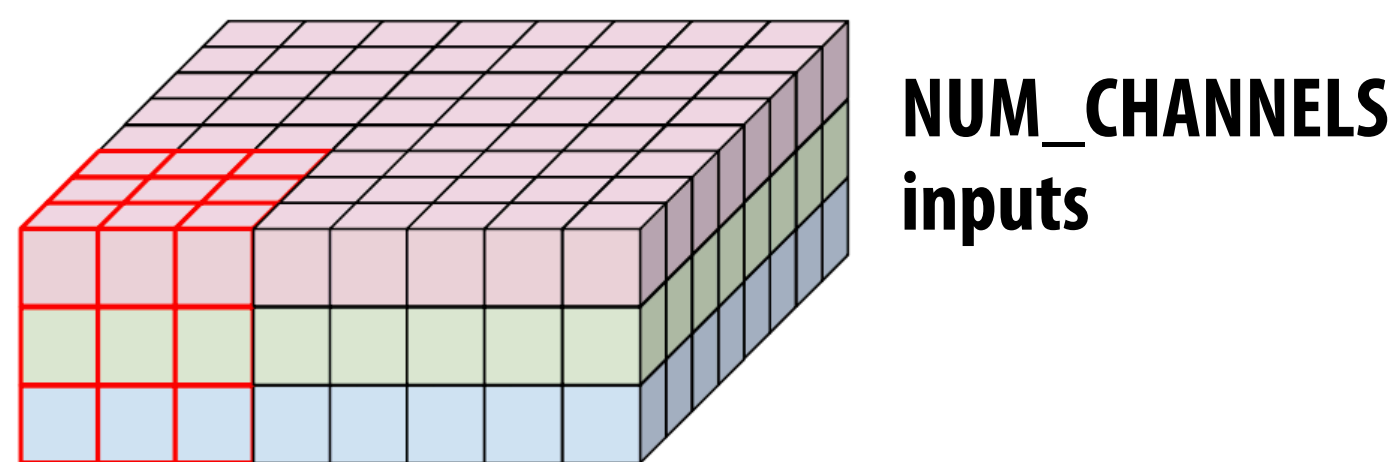
NUM_CHANNELS inputs
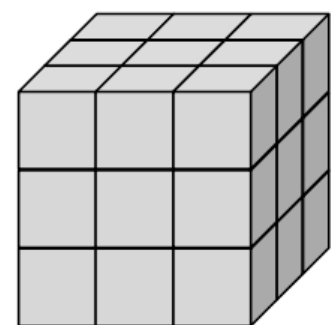
K_w x K_h x NUM_CHANNELS weights (for each filter)

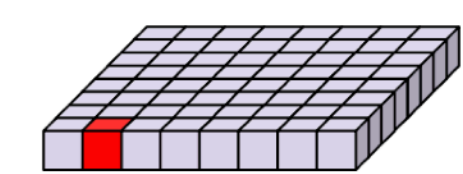K_w x K_h x NUM_CHANNELS work per output pixel (per filter)

NUM_CHANNELS inputs

K_w x K_h weights (for each channel)

results of filtering each of NUM_CHANNELS independently

NUM_CHANNELS weights (for each filter)

NUM_CHANNELS work per output pixel (per filter)

Image credit: Eli Bendersky
https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

MobileNet

Factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:

- **NUM_CHANNELS 3x3x1 convolutions for each input channel**
- **And NUM_FILTERS 1x1xNUM_CHANNELS convolutions to combine the results**

3x3 Conv
BN
ReLU

3x3 Depthwise Conv
BN
ReLU
1x1 Conv
BN
ReLU
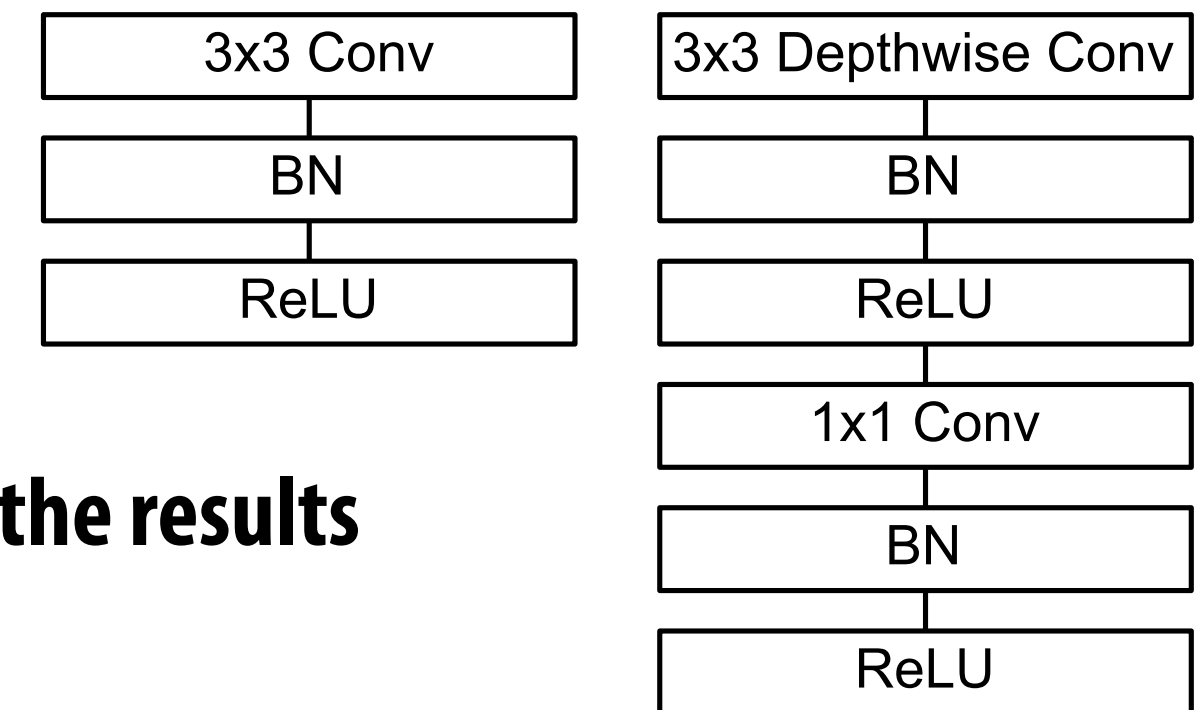
Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Image classification (ImageNet) Comparison to Common DNNs

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Image classification (ImageNet) Comparison to Other Compressed DNNs

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60

Value of improving DNN topology

- **Increasing overall accuracy on a task (often primary goal of CV/ML papers)**

- **Increasing accuracy/unit cost**

- **What is cost of evaluating DNN?**

    - **Number of ops (often measured in multiply adds)**

    - **Bandwidth!**

        - **Loading model weights + loading/storing intermediate activations**

        - **Careful! Certain layers are bandwidth bound, e.g., batch norm**

Depthwise separable convolutions add additional batch norm operations to network (after each step of depthwise conv layer)

Implication: number of ops can be a poor predictor of run time of network (too small to utilize processor, bandwidth bound, etc.)!

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Model optimization techniques

- **Manually designing better models**

  - **Common parameters: depth of network, width of filters, number of filters per layer, convolutional stride, etc.**

- **Good scheduling of performance-critical operations (layers)**

  - **Loop blocking/tiling, fusion**

  - **Typically optimized manually by humans (but significant research efforts to automate scheduling)**

- **Compressing models**

  - **Lower bit precision**

  - **Automatic sparsification/pruning**

- **Automatically discovering efficient model topologies (architecture search)**

DNN architecture search

- **Learn an efficient DNN topology along with associated weights**
- **Example: progressive neural architecture search [Liu et al. 18]**
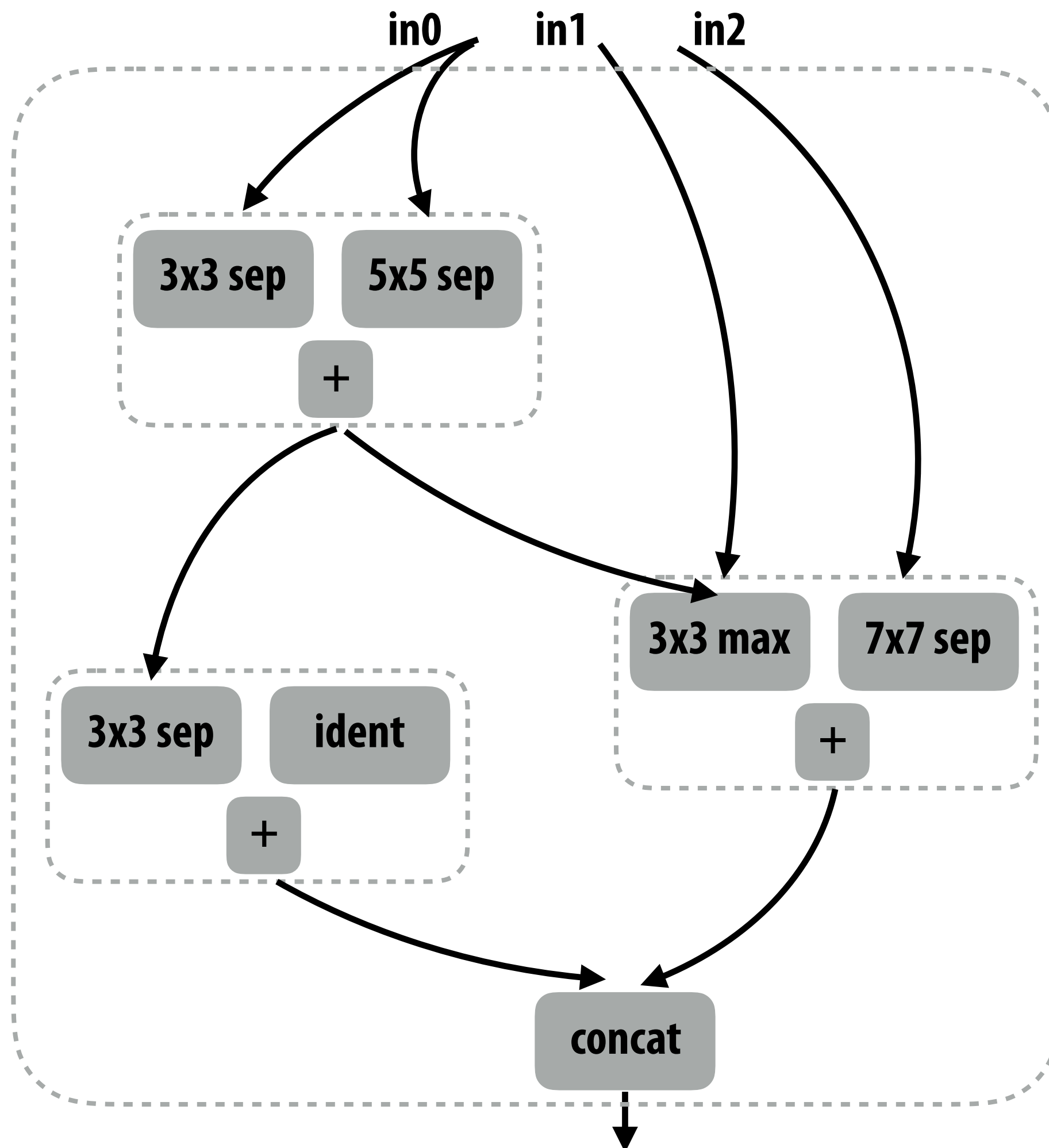
"Block" = (input1, input2, op1, op2)
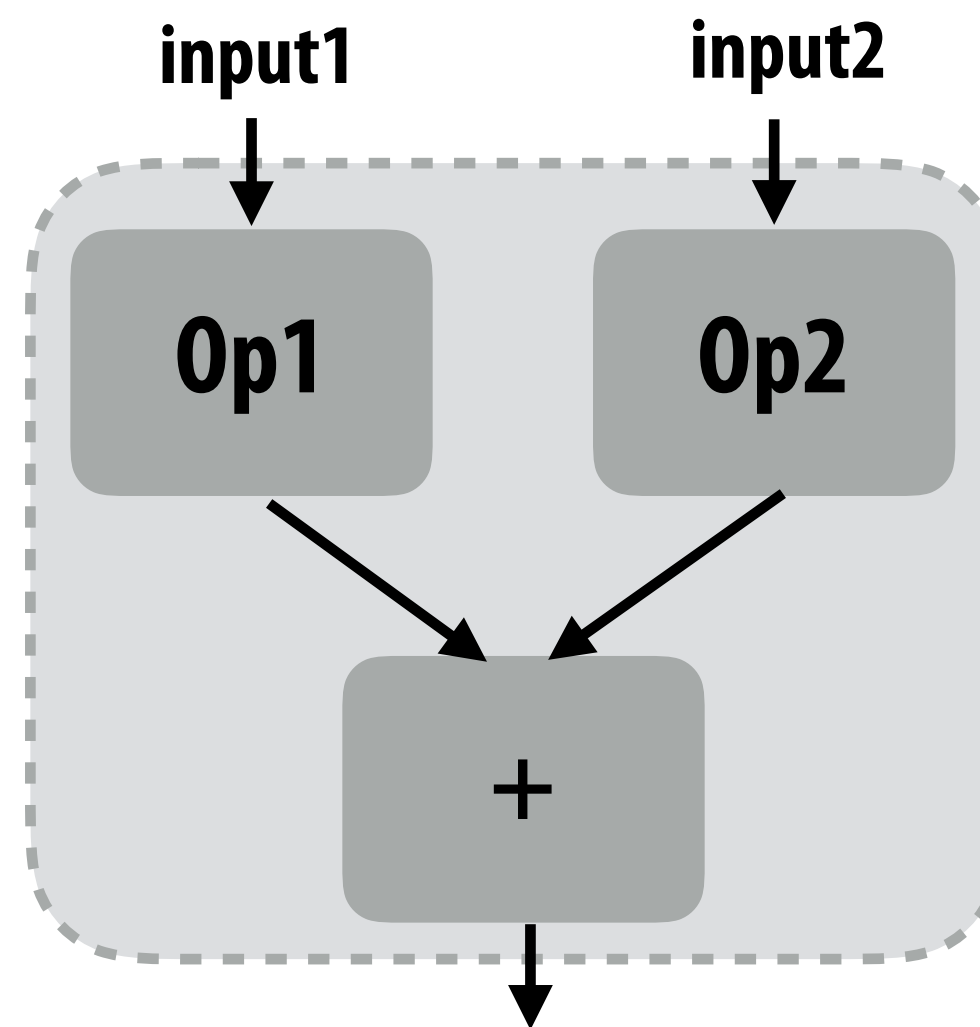
Eight possible operations:

3x3 depthwise-separable conv	identity
5x5 depthwise-separable conv	3x3 average pool
7x7 depthwise-separable conv	3x3 max pool
1x7 followed by 7x1 conv	3x3 dilated conv

Architecture search space

Cells are DAGs of _B_ blocks

DNNs are sequences of _N_ cells

in0   in1   in2

3x3 sep   5x5 sep

+

3x3 sep   ident

3x3 max   7x7 sep

+

+

concat

Cells have one output, can receive input from all prior cells

Cell 1

Cell 2

Cell N
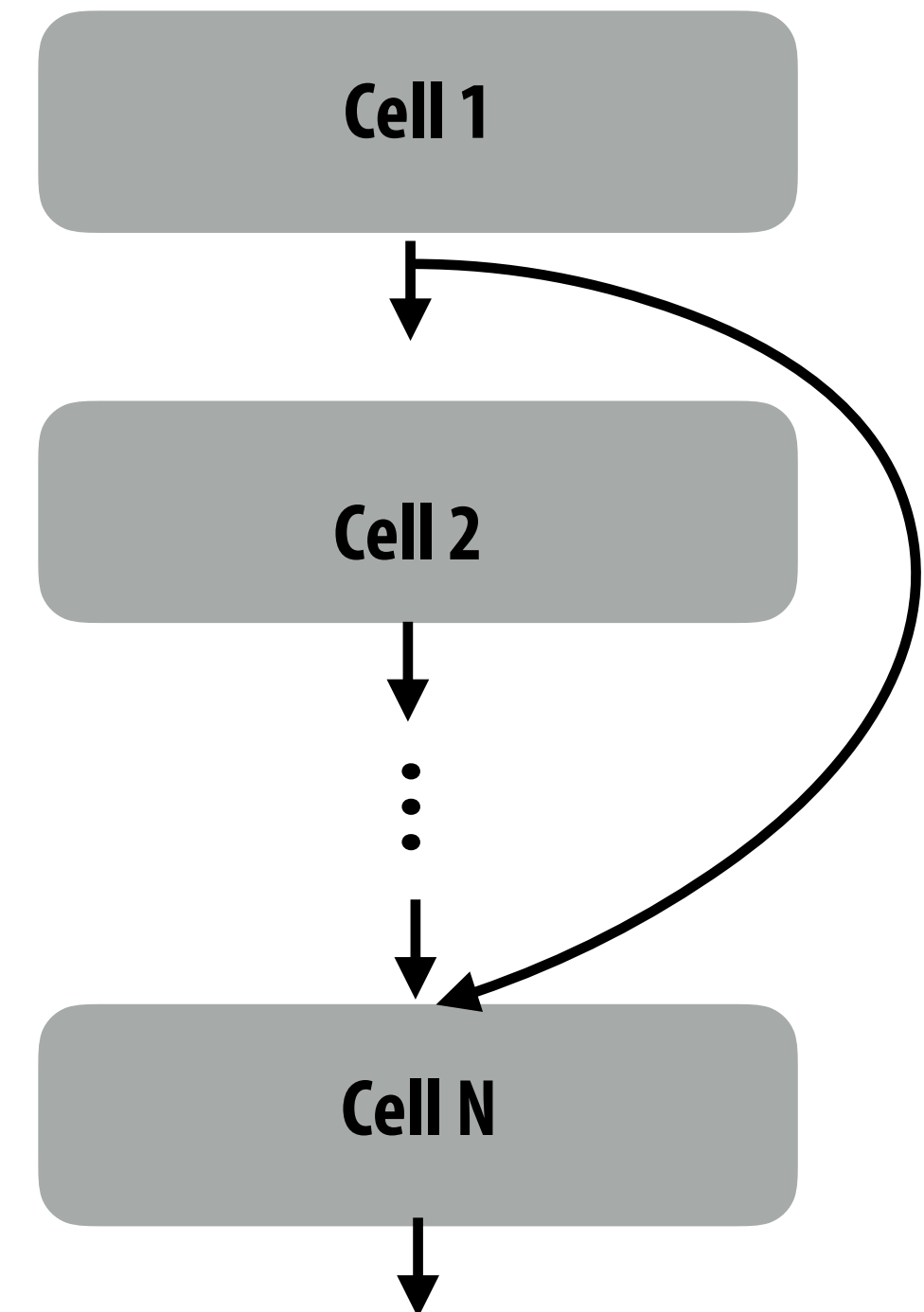
Progressive neural architecture search results

- **Automatic search was able to find model architectures that yielded similar/ better accuracy to hand designed models (and comparable costs)**

Model	Params	Mult-Adds	Top-1	Top-5
MobileNet-224 [14]	4.2M	569M	70.6	89.5
ShuffleNet (2x) [37]	5M	524M	70.9	89.8
NASNet-A ($N = 4, F = 44$) [41]	5.3M	564M	74.0	91.6
AmoebaNet-B ($N = 3, F = 62$) [27]	5.3M	555M	74.0	91.5
AmoebaNet-A ($N = 4, F = 50$) [27]	5.1M	555M	74.5	92.0
AmoebaNet-C ($N = 4, F = 50$) [27]	6.4M	570M	75.7	92.4
PNASNet-5 ($N = 3, F = 54$)	5.1M	588M	74.2	91.9

- **Forms of architecture search implemented by Cloud-based ML hosting services (user provides training data, service searches for good model)**

Google Cloud
Cloud AutoML BETA

Amazon SageMaker
Build, train, and deploy machine learning models at scale

Why might a GPU be a good platform for DNN evaluation?

Deep neural networks on GPUs

- **Many high-performance DNN implementations target GPUs**
  - High arithmetic intensity computations (computational characteristics similar to dense matrix-matrix multiplication)
  - Benefit from flop-rich architectures
  - Highly-optimized library of kernels exist for GPUs (cuDNN)
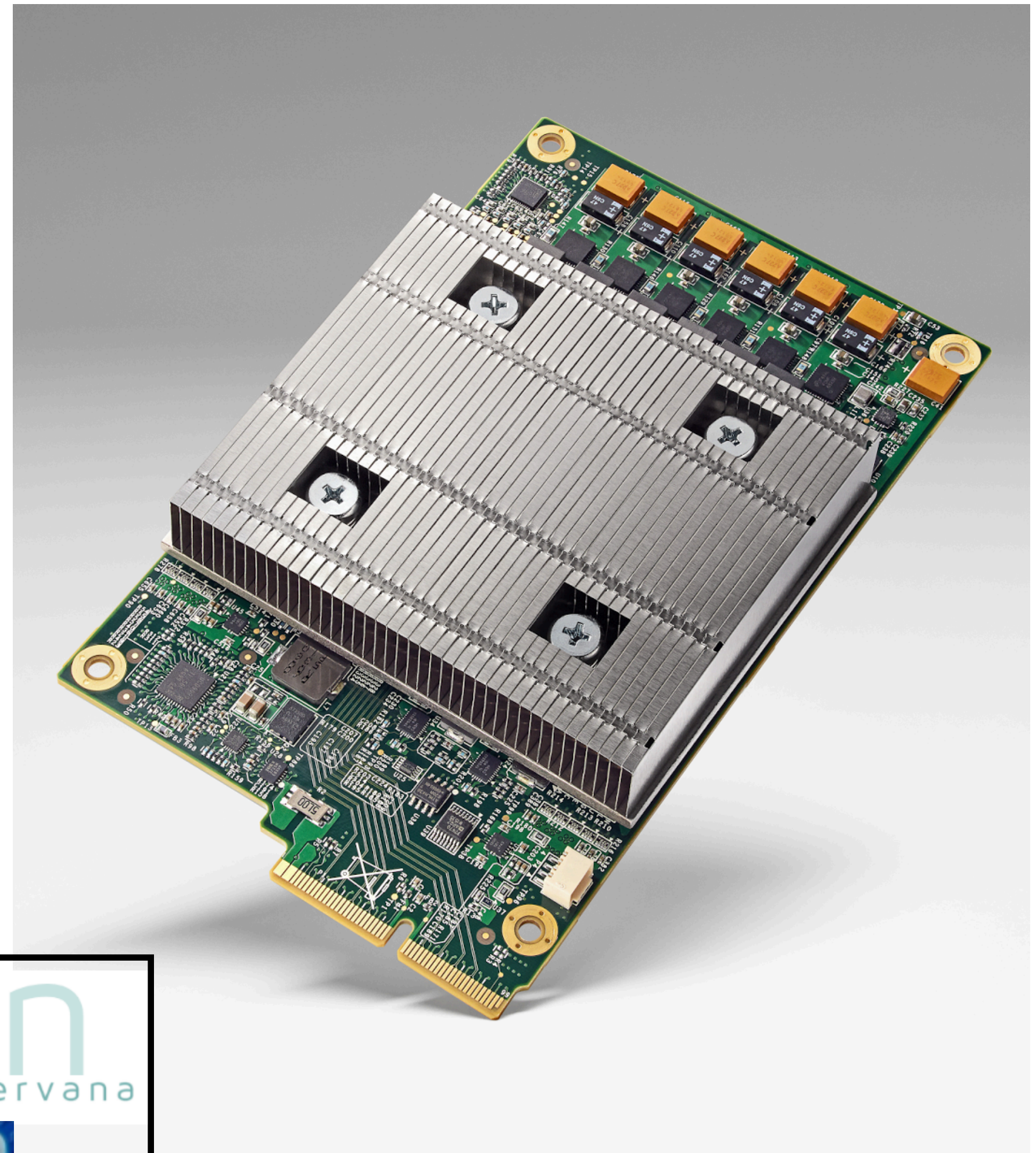   - Most CPU-based implementations use basic matrix-multiplication-based formulation (good implementations could run faster!)

Facebook's Big Sur

Why might a GPU be a sub-optimal platform for DNN evaluation?

Increasing efficiency through specialization

Example: Google's Tensor Processing Unit (TPU)
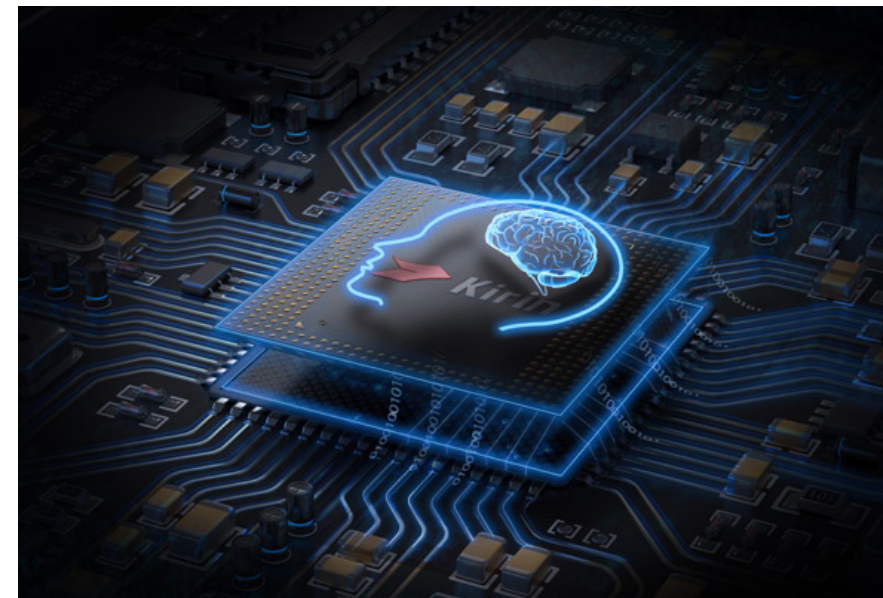Accelerates deep learning operations in Google
datacenter

Intel has announced
Lake Crest ML accelerator
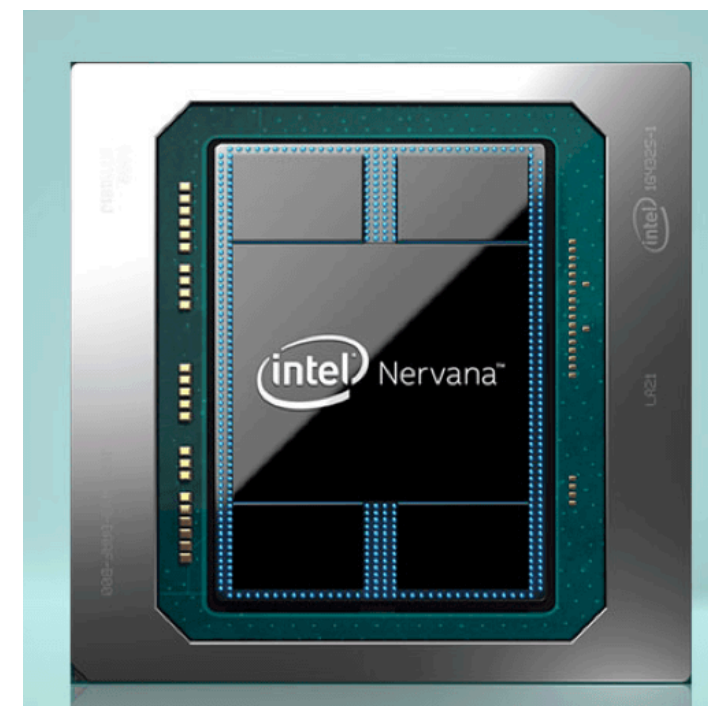(formerly called Nervana)
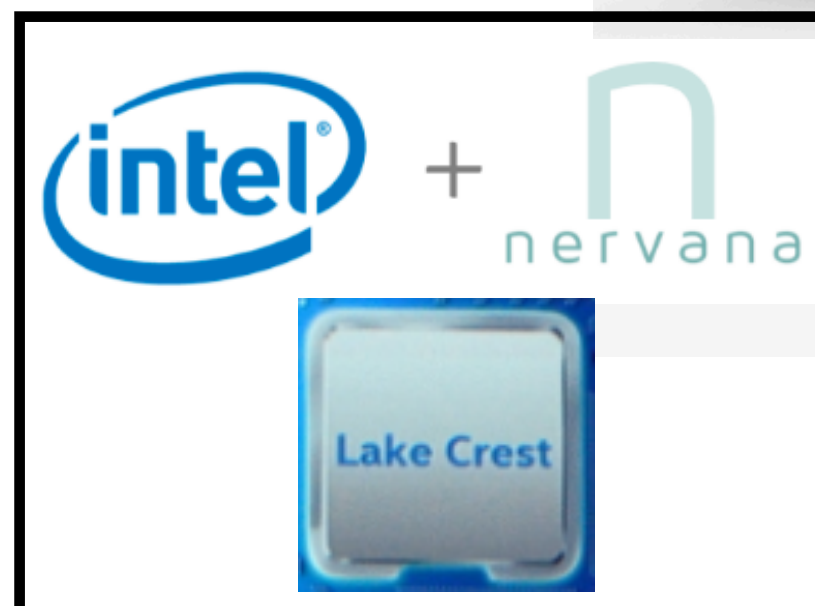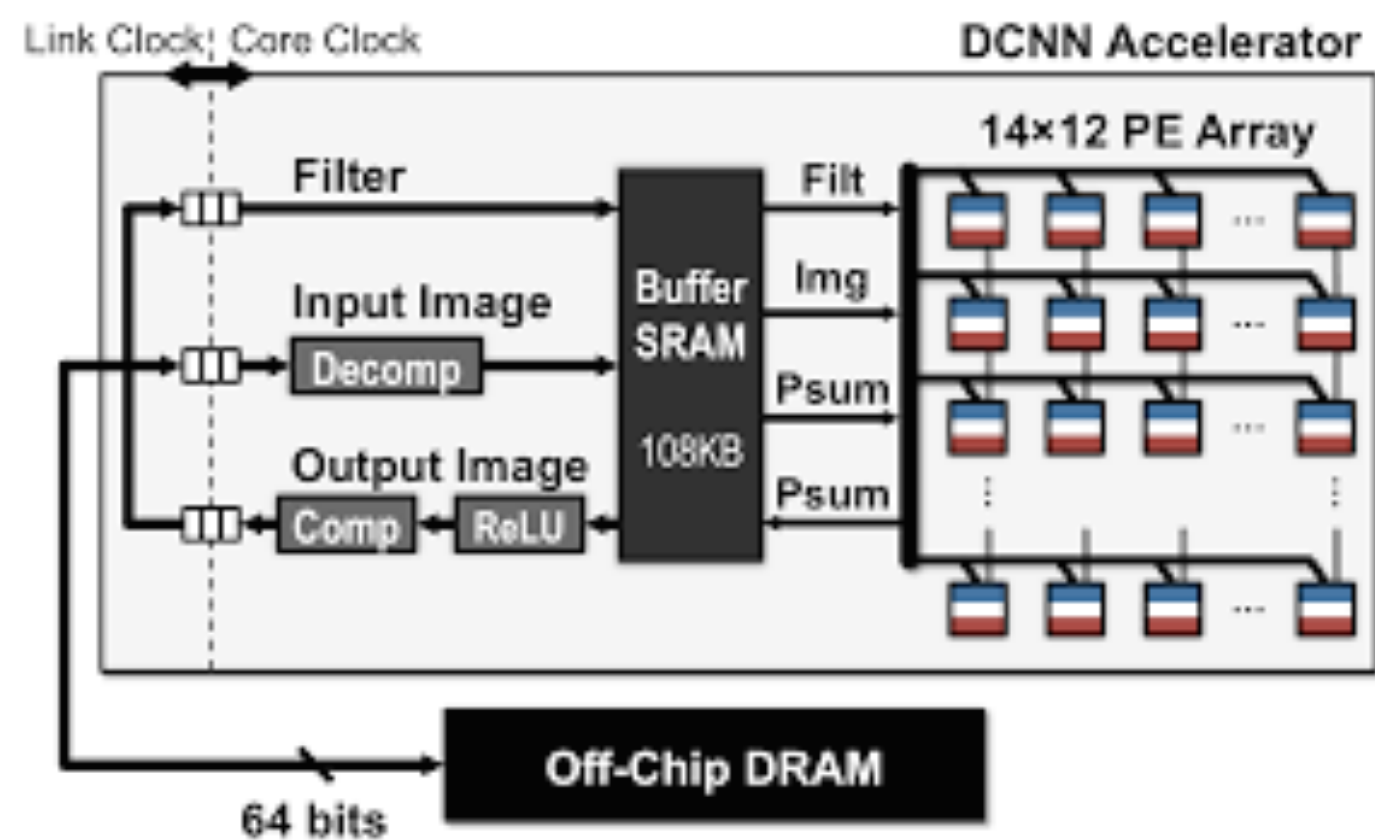
Hardware acceleration for DNNs
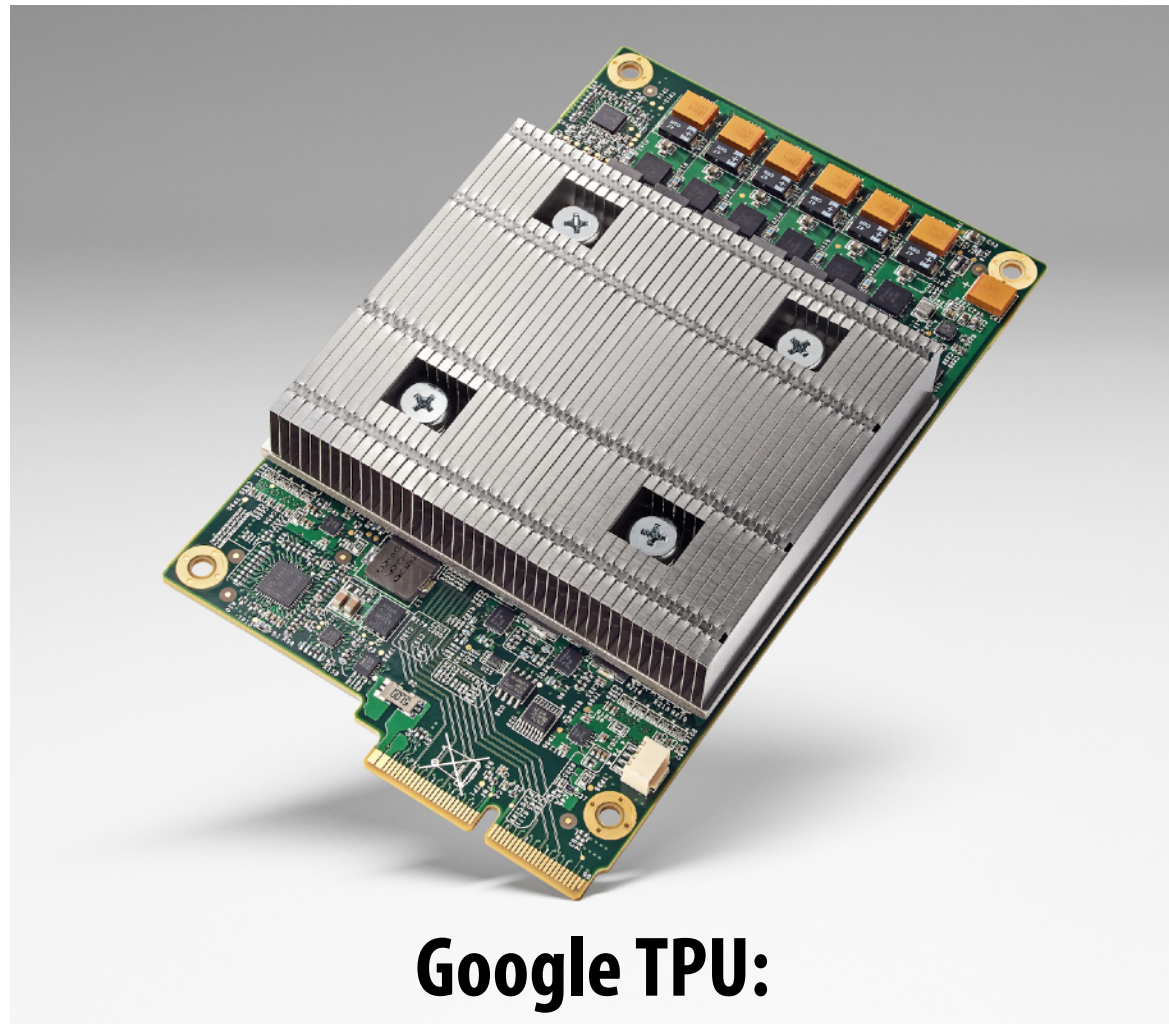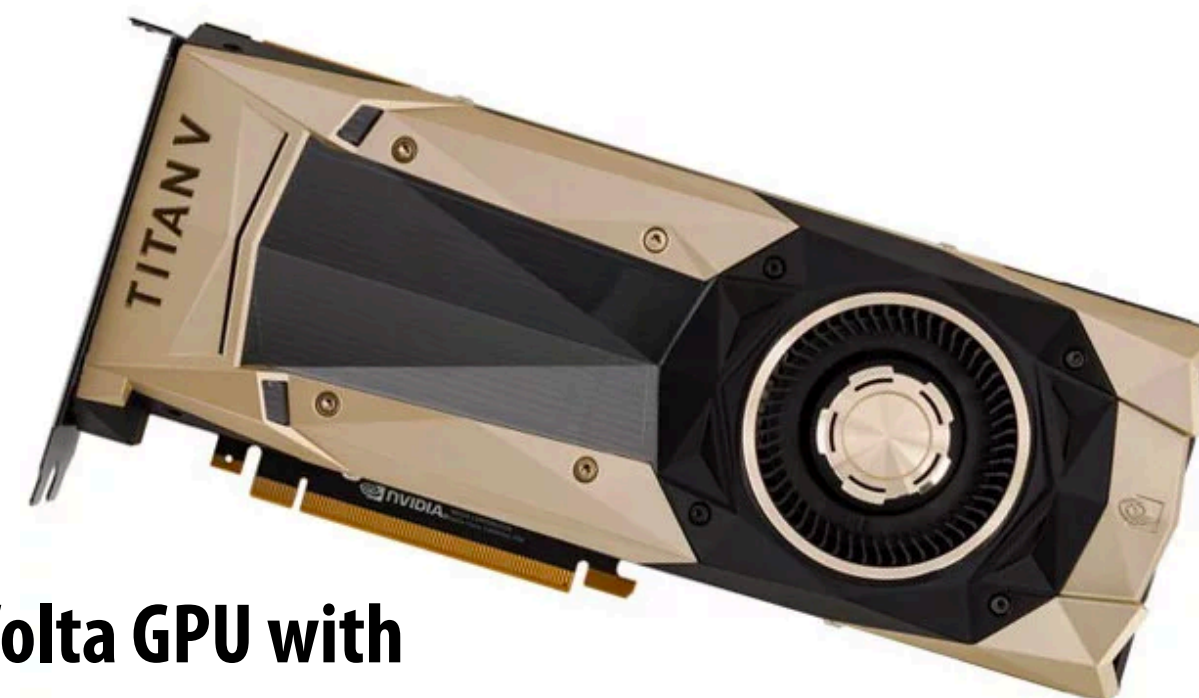

Google TPU:

Huawei Kirin NPU

Apple Neural Engine

Intel Lake Crest
Deep Learning Accelerator


MIT Eyeriss

Volta GPU with
Tensor Cores

And many more…

IC Giants	Intel, Qualcomm, Nvidia, Samsung, AMD, Apple, Xilinx, IBM, STMicroelectronics, NXP, MediaTek, HiSilicon	12
Cloud/HPC	Google, Amazon_AWS, Microsoft, Aliyun, Tencent Cloud, Baidu, Baidu Cloud, HUAWEI Cloud, Fujitsu	9
IP Vendors	ARM, Synopsys, Imagination, CEVA, Cadence, VeriSilicon	6
Startups in China	Cambricon, Horizon Robotics, DeePhi, Bitmain, Chipintelli, Thinkforce	6
Startups Worldwide	Cerebras, Wave Computing, Graphcore, PEZY, KnuEdge, Tenstorrent, ThinCI, Koniku, Adapteva, Knowm, Mythic, Kalray, BrainChip, AImotive, DeepScale, Leepmind, Krtkl, NovuMind, REM, TERADEEP, DEEP VISION, Groq, KAIST DNPU, Kneron, Vathys, Esperanto Technologies	26

Modern NVIDIA GPU
(Volta)

Recall: properties of GPUs

- **"Compute rich": packed densely with processing elements**
  - **Good for compute-bound applications**

- **Good, because dense-matrix multiplication and DNN convolutional layers (when implemented properly) are compute bound**

- **But recall cost of instruction stream processing and control in a programmable processor:**

Note: these figures are estimates for a CPU:

Clock and Control 24%

Data supply 28%

Arithmetic 6%

Instruction supply 42%
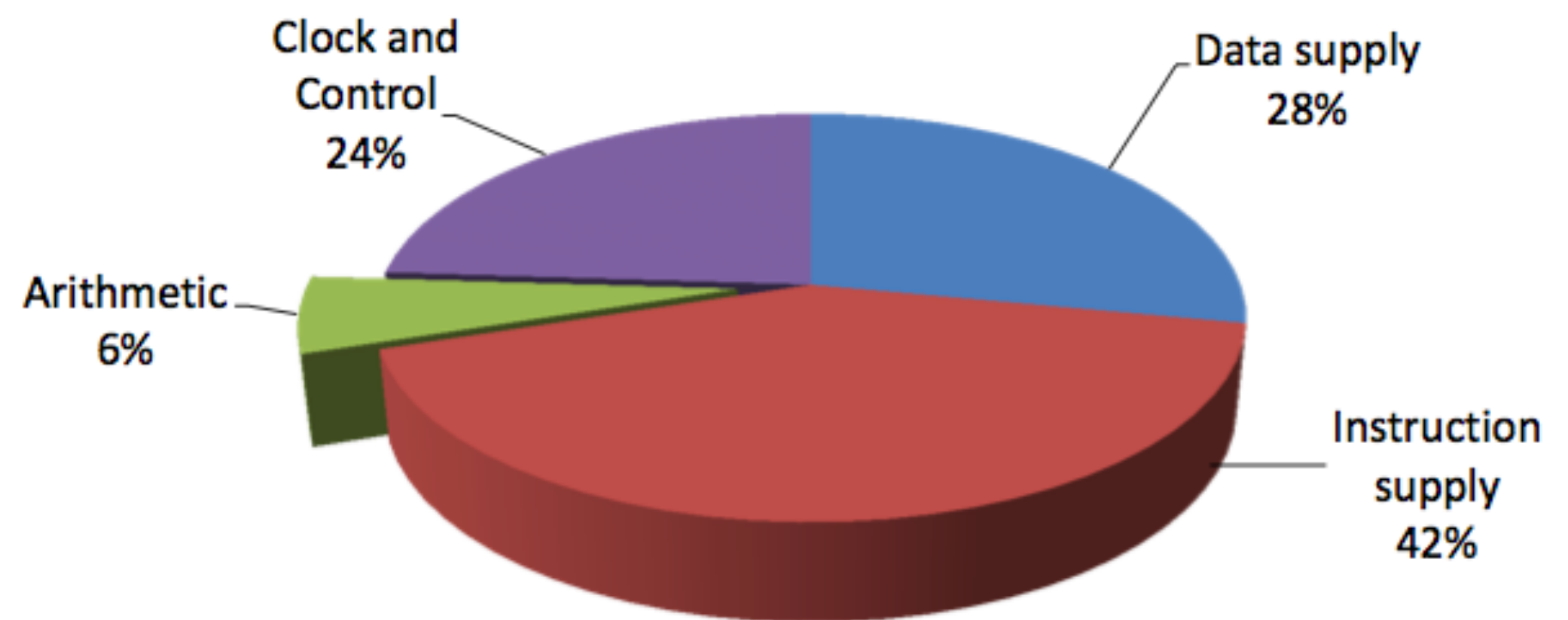
Efficient Embedded Computing [Dally et al. 08]
[Figure credit Eric Chung]

One solution: more complex instructions

- **Fused multiply add (ax + b)**

- **4-component dot product x = A dot B**

- **4x4 matrix multiply**

  - **AB + C for 4x4 matrices A, B, C**

- **Key principle: amortize cost of instruction stream processing across many operations of a single complex instruction**

Volta GPU

Single instruction to perform 2x4x4x4 + 4x4 ops

Each SM core has:

64 fp32 ALUs (mul-add)

32 fp64 ALUs

8 "tensor cores"

Execute 4x4 matrix mul-add instr

A x B + C for 4x4 matrices A,B,C

A, B stored as fp16, accumulation with fp32 C

GV100 GPU has 80 SM cores:

5,120 fp32 mul-add ALUs

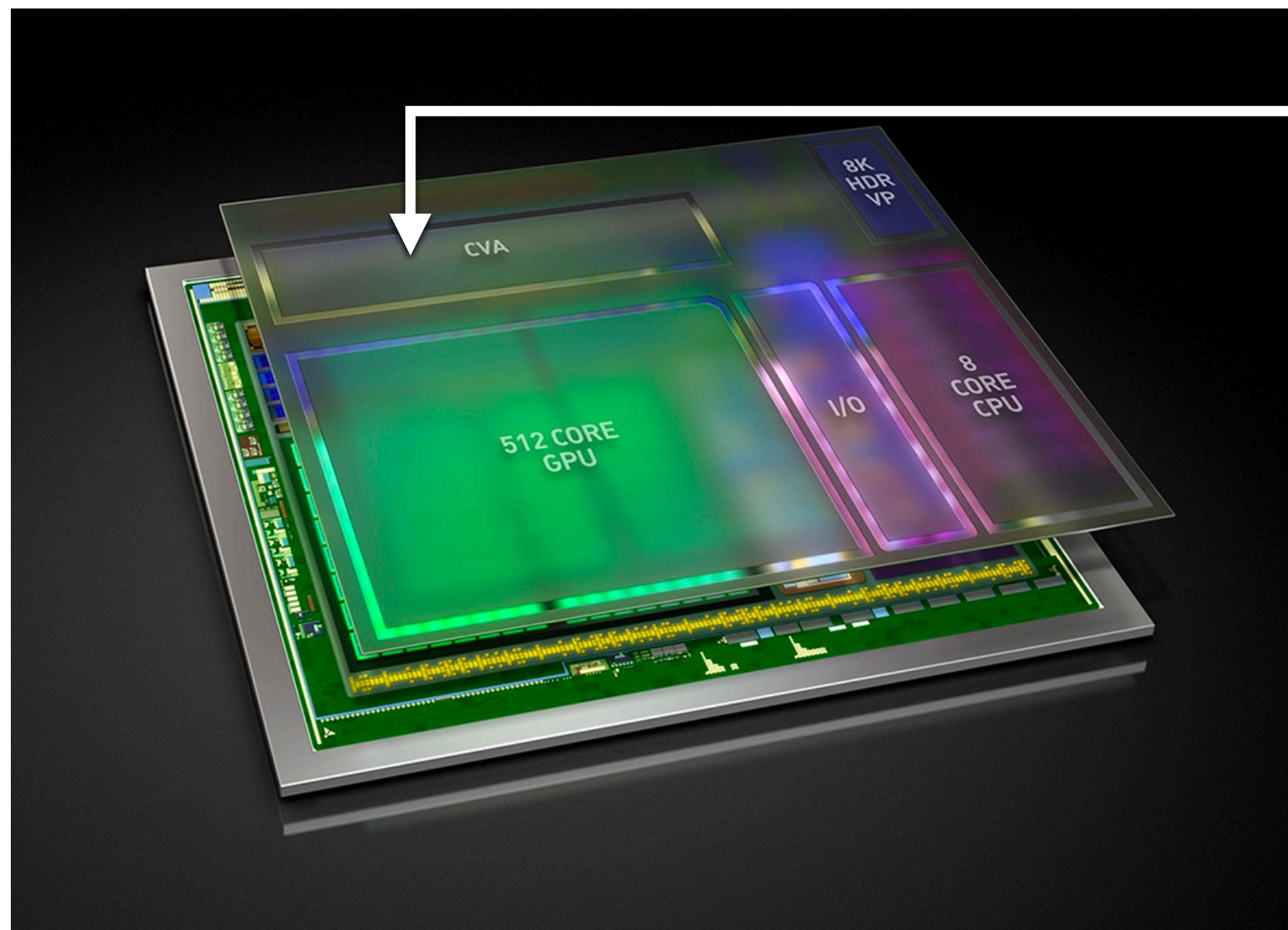**640 tensor cores**

6 MB of L2 cache

1.5 GHz max clock

= 15.7 TFLOPs fp32

= 125 TFLOPs (fp16/32 mixed) in tensor cores

Efficiency estimates *

- Estimated overhead of programmability (instruction stream, control, etc.)

  - Half-precision FMA (fused multiply-add)   2000%

  - Half-precision DP4 (vec4 dot product)   500%

 - Half-precision MMA (matrix-matrix multiply + accumulate)   27%

NVIDIA Xavier (SoC for automotive domain)

Features a Computer Vision Accelerator (CVA), a custom module for deep learning acceleration (large matrix multiply unit)
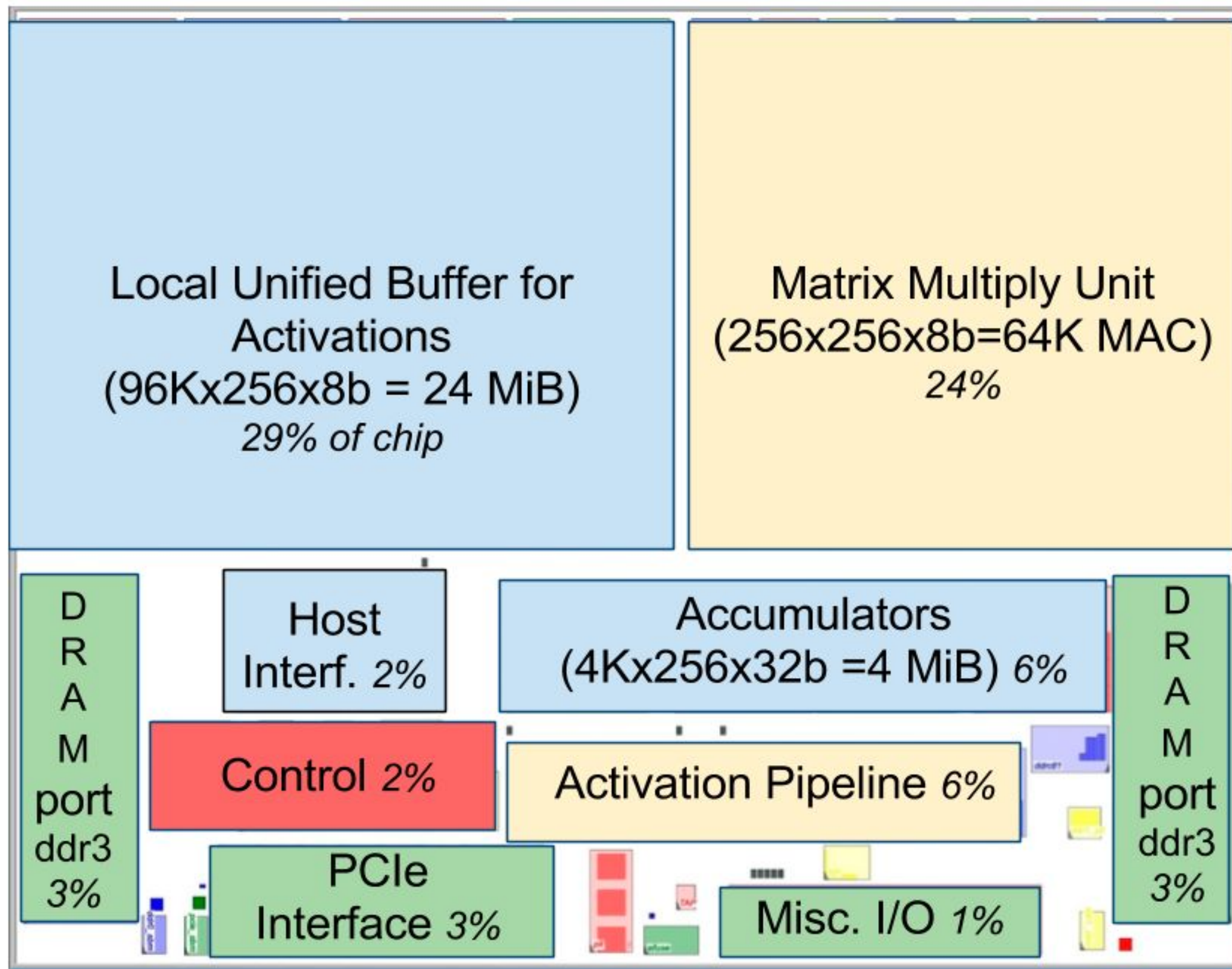
But only 2x more efficient than Volta MMA instruction despite being highly specialized component. (includes optimization of gating multipliers if either operand is zero)

* Estimates by Bill Dally using academic numbers, SysML talk, Feb 2018

Google TPU
(version 1)

Google's TPU
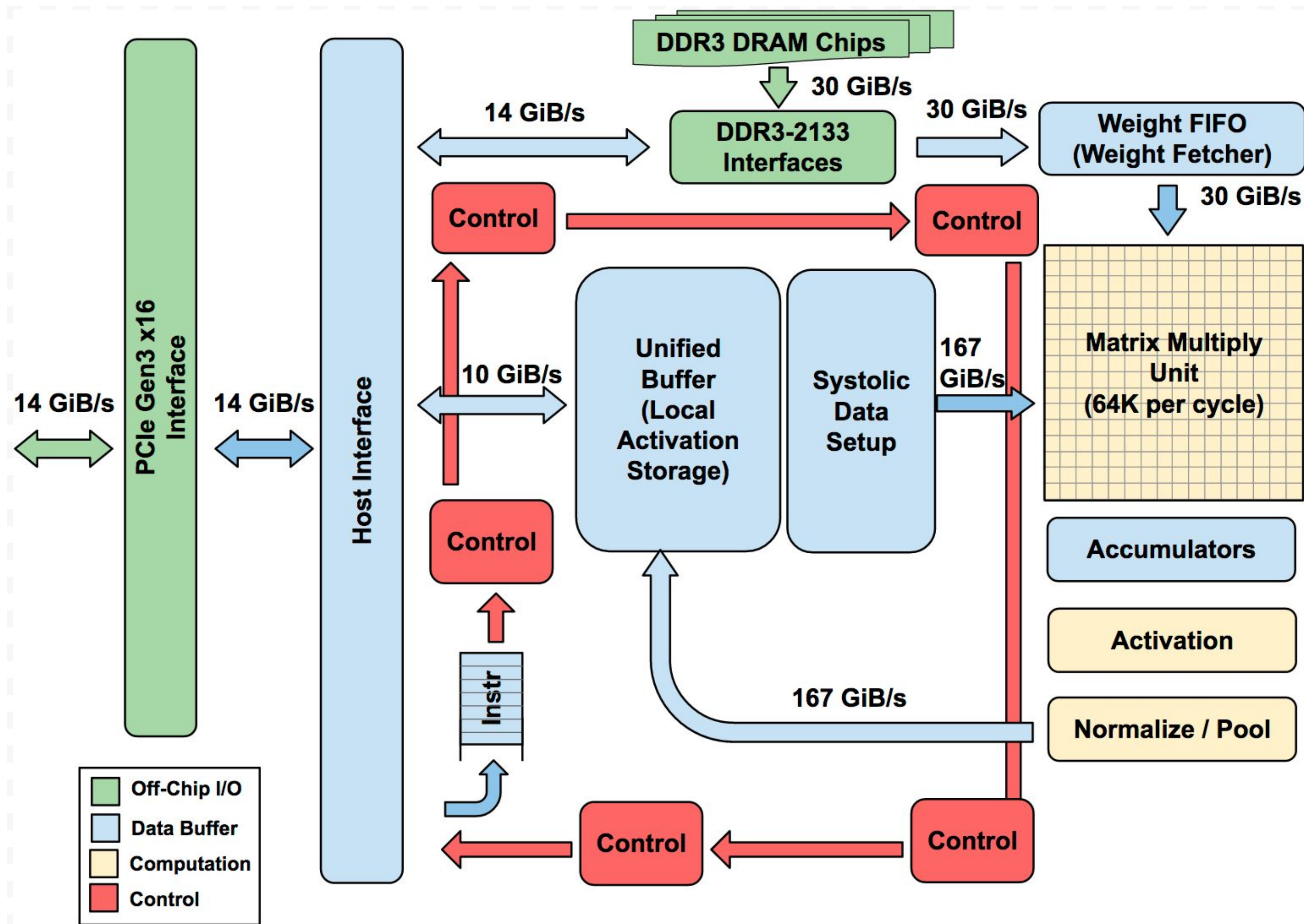
TPU area proportionality

Compute ~ 30% of chip
Note low area footprint of control
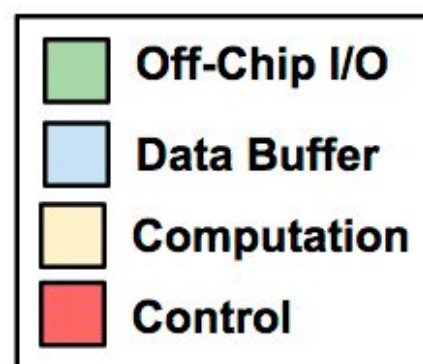
Key instructions:
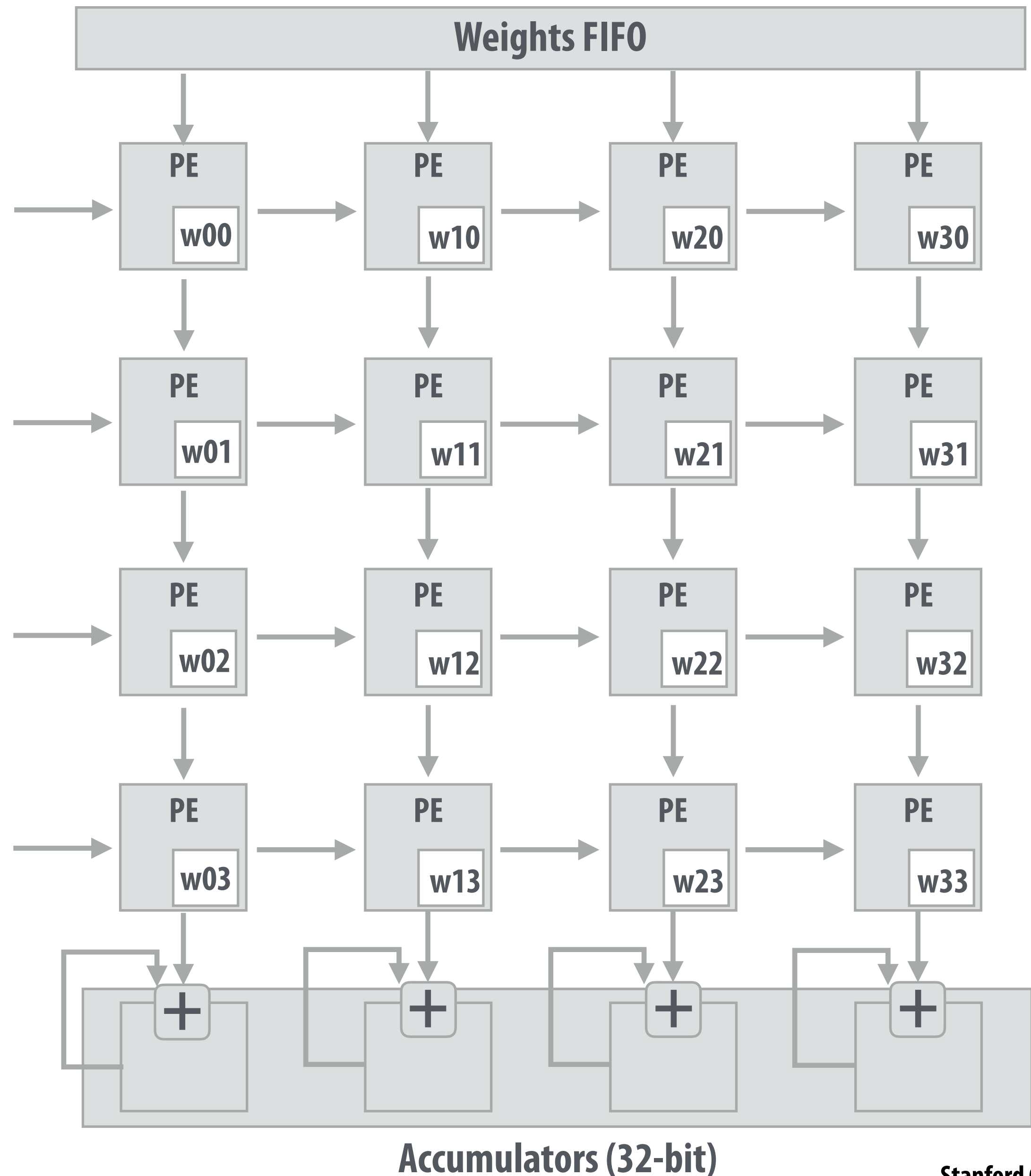   read host memory
   write host memory
   read weights
   matrix_multiply / convolve
   activate

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

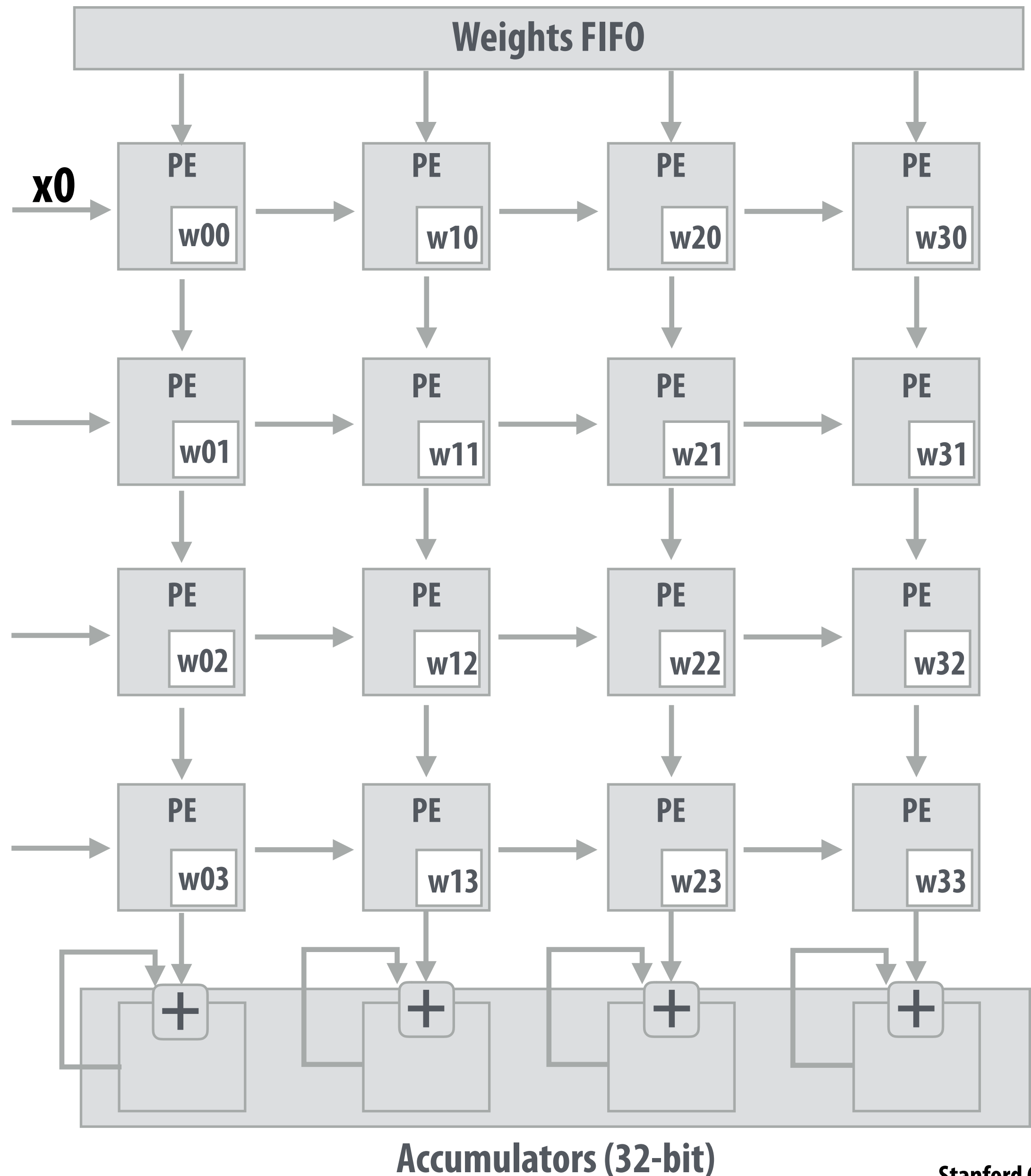| PE w00 | PE w10 | PE w20 | PE w30 |
PE w01	PE w11	PE w21	PE w31
PE w02	PE w12	PE w22	PE w32
PE w03	PE w13	PE w23	PE w33

+ + + +

Accumulators (32-bit)

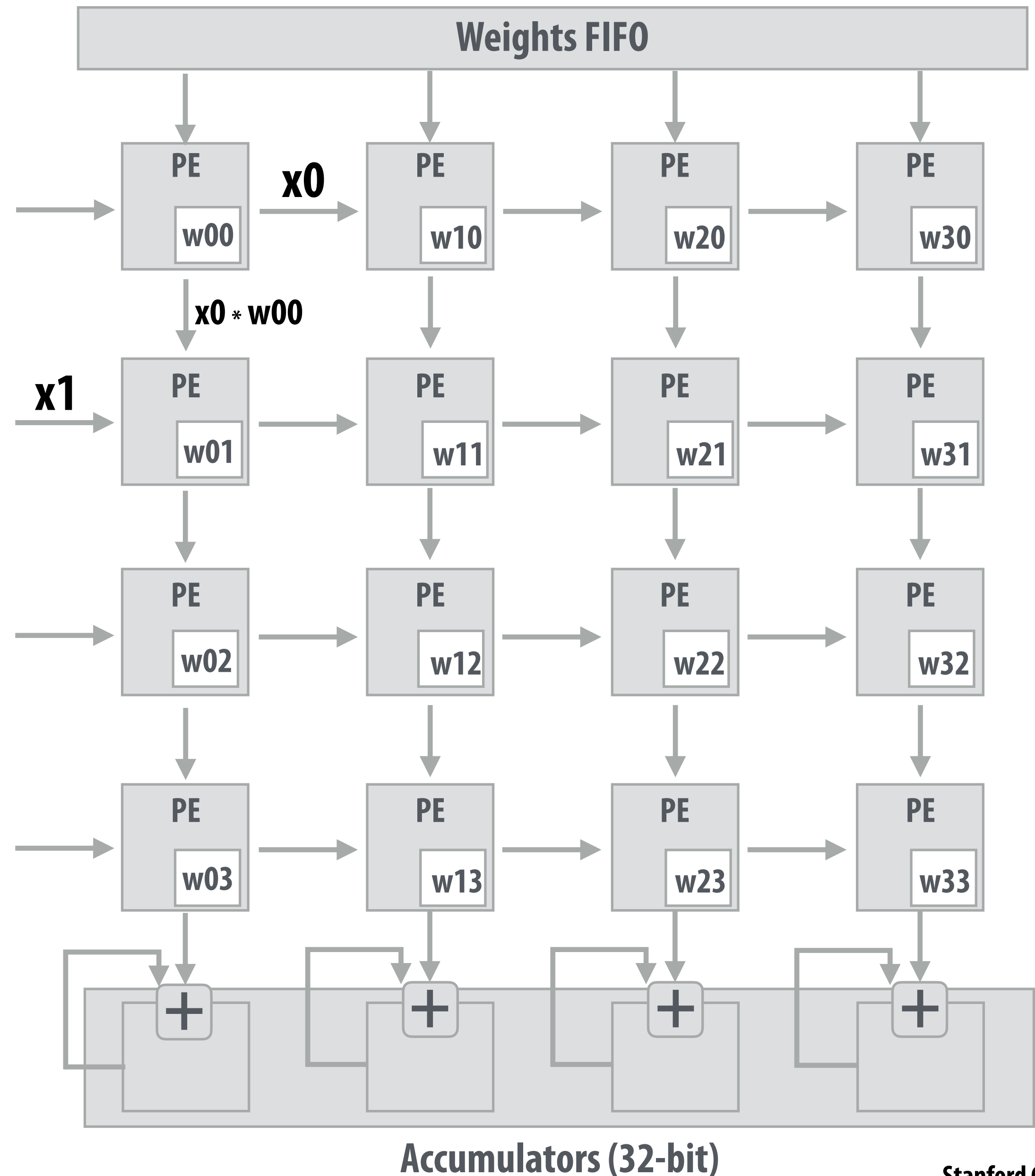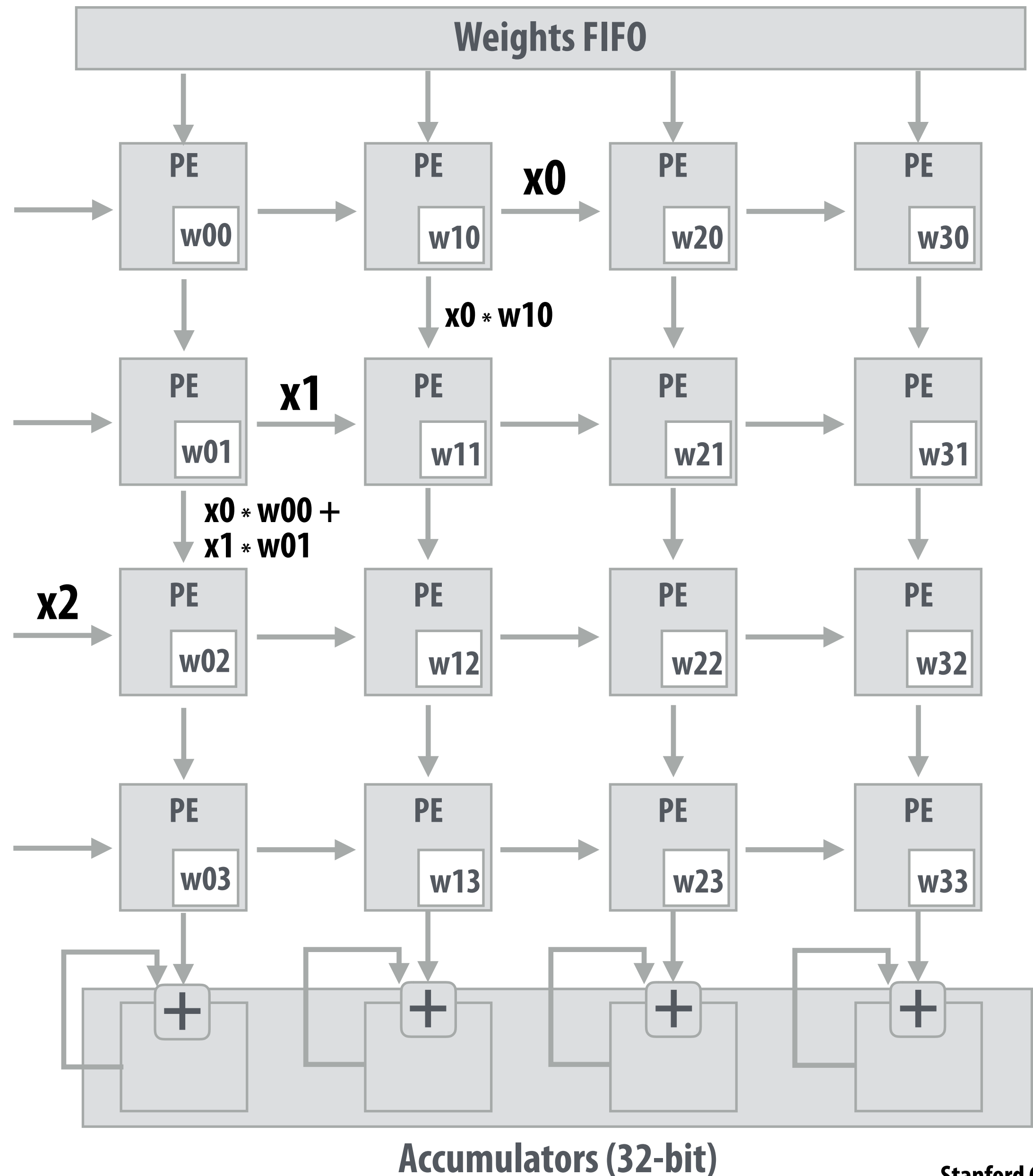# Systolic array

(matrix vector multiplication example: $y=Wx$)

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

PE w00 — x0 → PE w10 → PE w20 → PE w30

x0 * w00

x1 → PE w01 → PE w11 → PE w21 → PE w31

PE w02 → PE w12 → PE w22 → PE w32

PE w03 → PE w13 → PE w23 → PE w33

+ + + +

Accumulators (32-bit)

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

| PE w00 | PE w10 | x0 | PE w20 | PE w30 |

x0 * w10

| PE w01 | x1 | PE w11 | PE w21 | PE w31 |

x0 * w00 + x1 * w01

x2

| PE w02 | PE w12 | PE w22 | PE w32 |

| PE w03 | PE w13 | PE w23 | PE w33 |

+ + + +

Accumulators (32-bit)

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

| PE w00 | PE w10 | PE w20 | **x0** | PE w30 |

$x0 * w20$

| PE w01 | PE w11 | **x1** | PE w21 | PE w31 |

$x0 * w10 +$
$x1 * w11$

x2

| PE w02 | PE w12 | PE w22 | PE w32 |

$x0 * w00 +$
$x1 * w01 +$
$x2 * w02 +$

x3

| PE w03 | PE w13 | PE w23 | PE w33 |

$+$ $+$ $+$ $+$

Accumulators (32-bit)

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

PE w00   PE w10   PE w20   PE w30

x0 ∗ w30

PE w01   PE w11   PE w21   **x1**   PE w31

x0 ∗ w20 +
x1 ∗ w21

PE w02   PE w12   **x2**   PE w22   PE w32

x0 ∗ w10 +
x1 ∗ w11 +
x2 ∗ w12 +

PE w03   **x3**   PE w13   PE w23   PE w33

x0 ∗ w00 +
x1 ∗ w01 +
x2 ∗ w02 +
x3 ∗ w03

+   +   +   +

Accumulators (32-bit)

Systolic array

(matrix vector multiplication example: $y=Wx$)

Weights FIFO

| PE | x30 | PE | x20 | PE | x10 | PE |
| w00 | | w10 | | w20 | | w30 |

x30 * w00   x20 * w10   x10 * w20   x00 * w30

x31 → PE x21 PE x11 PE x01 PE
     w01   w11   w21   w31

x20 * w20 +   x10 * w20 +   x00 * w20 +
x21 * w21   x11 * w21   x01 * w21

x22 → PE x12 PE x02 PE   PE
     w02   w12   w22   w32

x10 * w10 +   x00 * w10 +
x11 * w11 +   x01 * w11 +
x12 * w12 +   x02 * w12 +

x13 → PE x03 PE   PE   PE
     w03   w13   w23   w33

x00 * w00 +
x01 * w01 +
x02 * w02 +
x03 * w03

Accumulators (32-bit)

Notice: need multiple 4x32bit accumulators to hold output columns

Building larger matrix-matrix multiplies
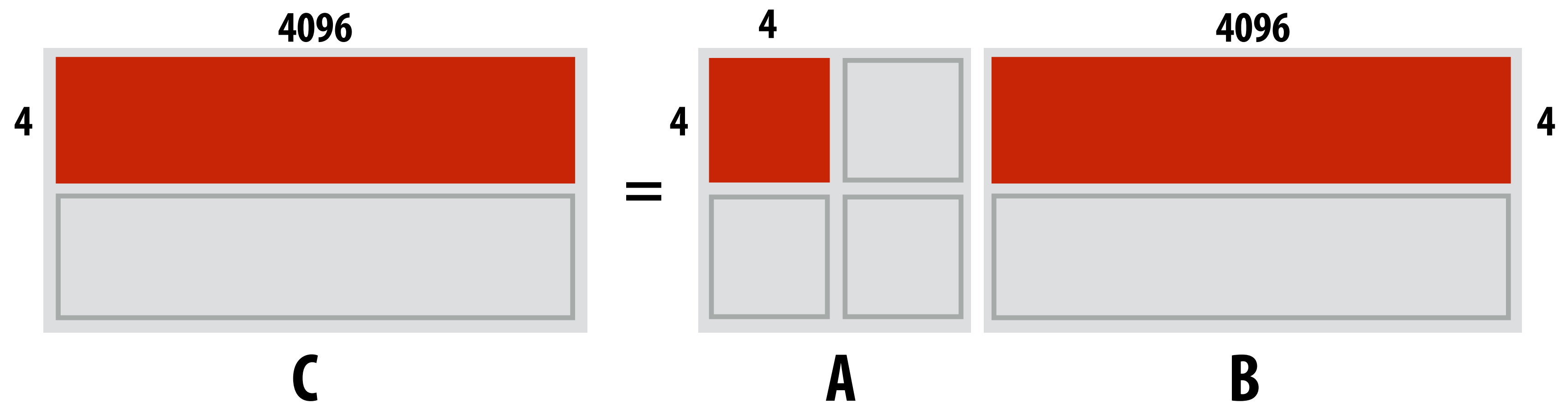
Example: A = 8x8, B= 8x4096, C=8x4096

Assume 4096 accumulators

Building larger matrix-matrix multiplies

Example: A = 8x8, B= 8x4096, C=8x4096

C = A B

Assume 4096 accumulators

Building larger matrix-matrix multiplies

Example: A = 8x8, B= 8x4096, C=8x4096

C

A

B

Assume 4096 accumulators

Building larger matrix-matrix multiplies

Example: A = 8x8, B= 8x4096, C=8x4096
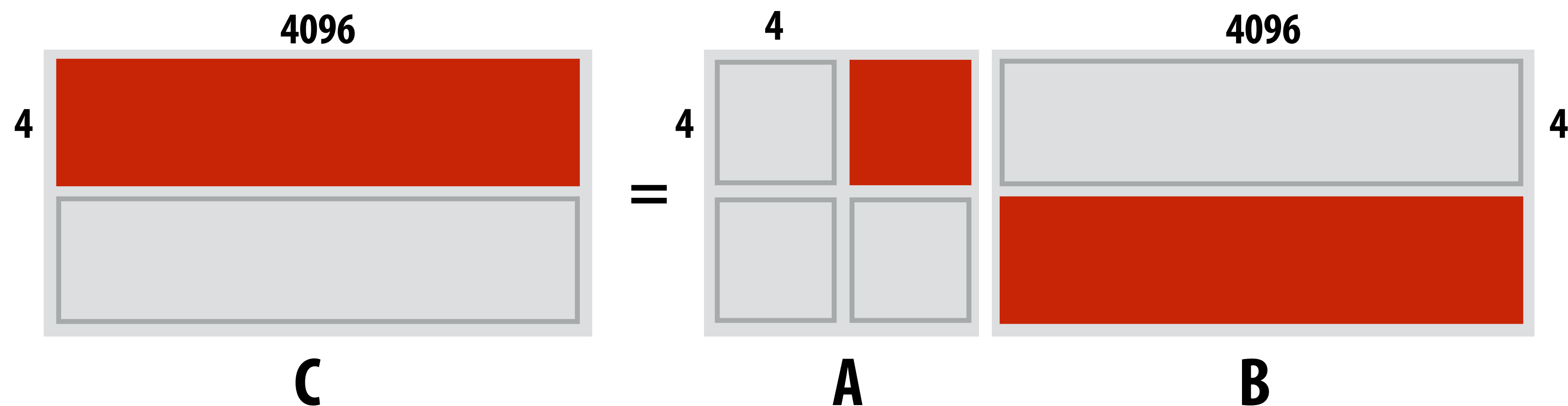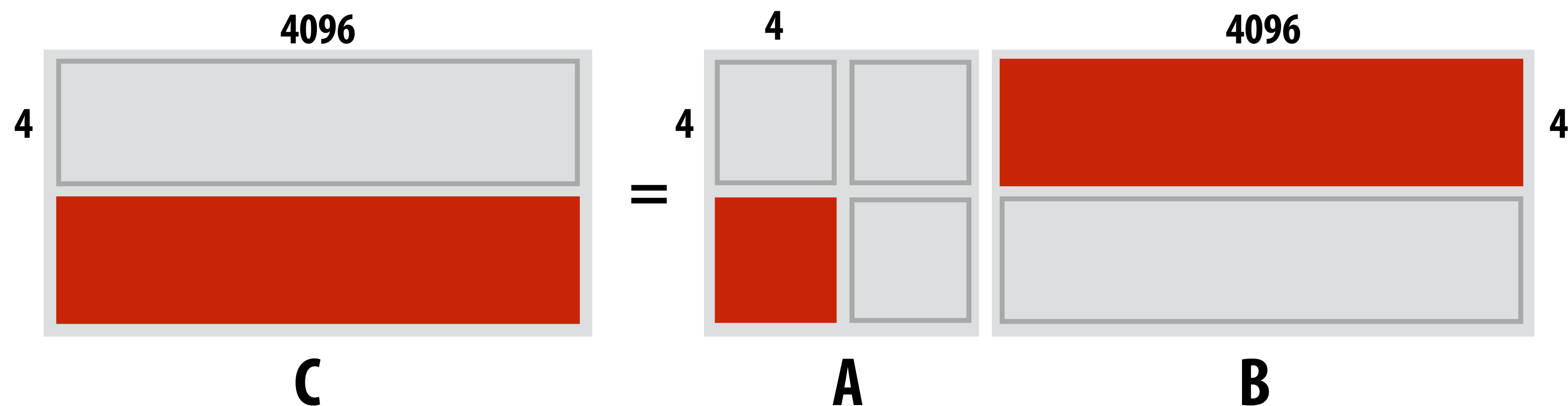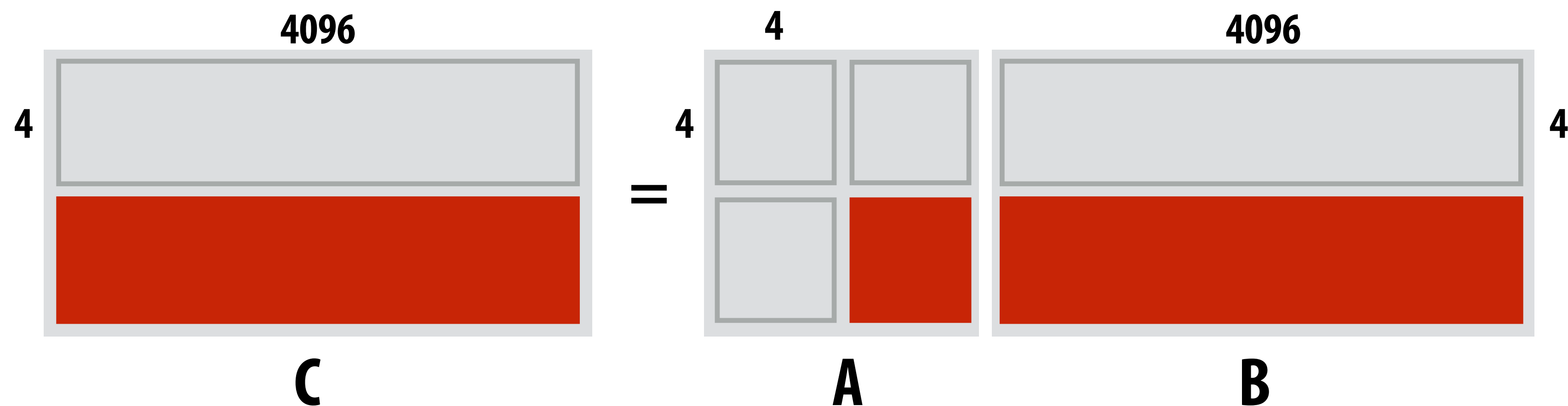
Assume 4096 accumulators

TPU Performance/Watt
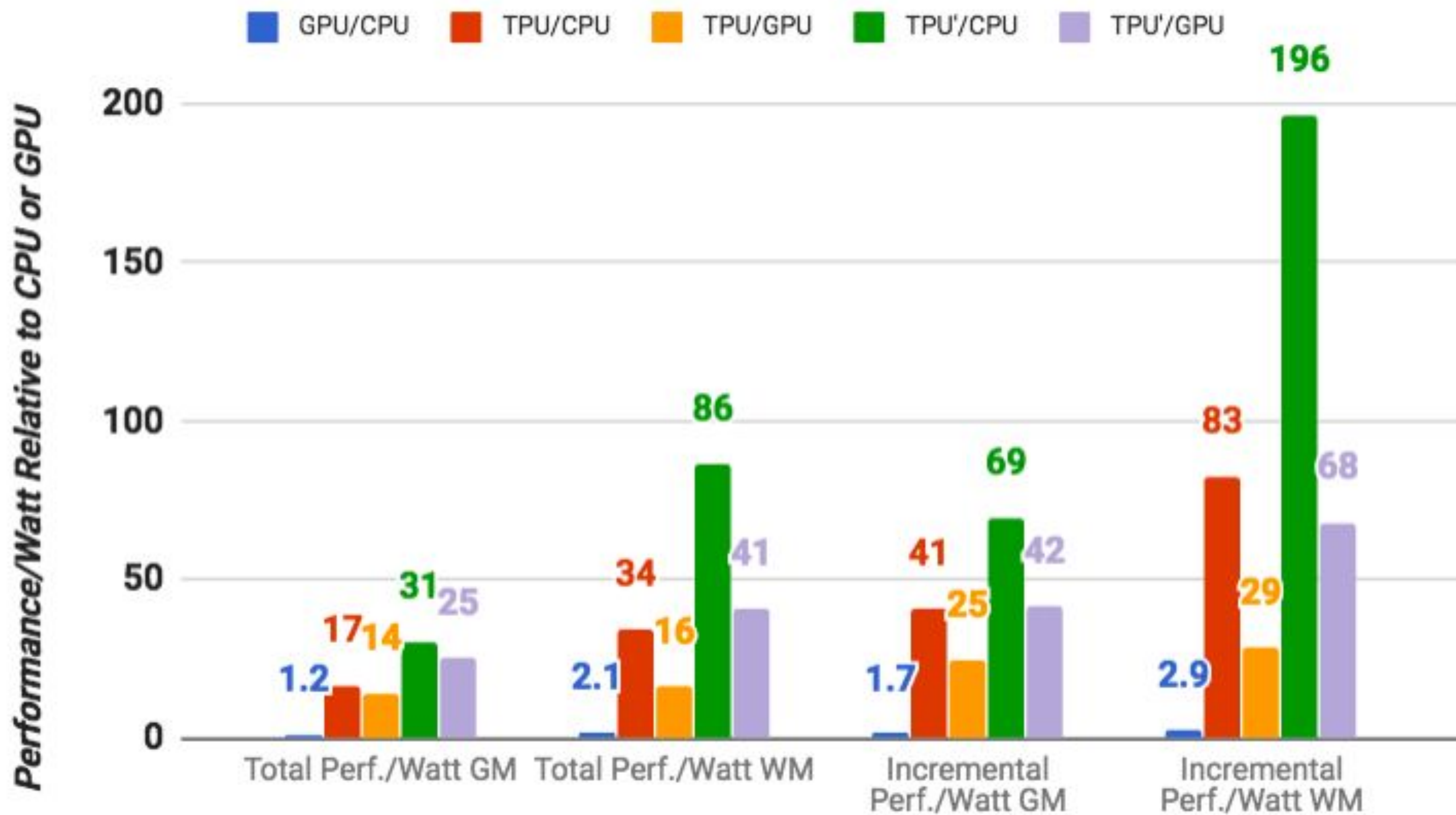
GM = geometric mean over all apps
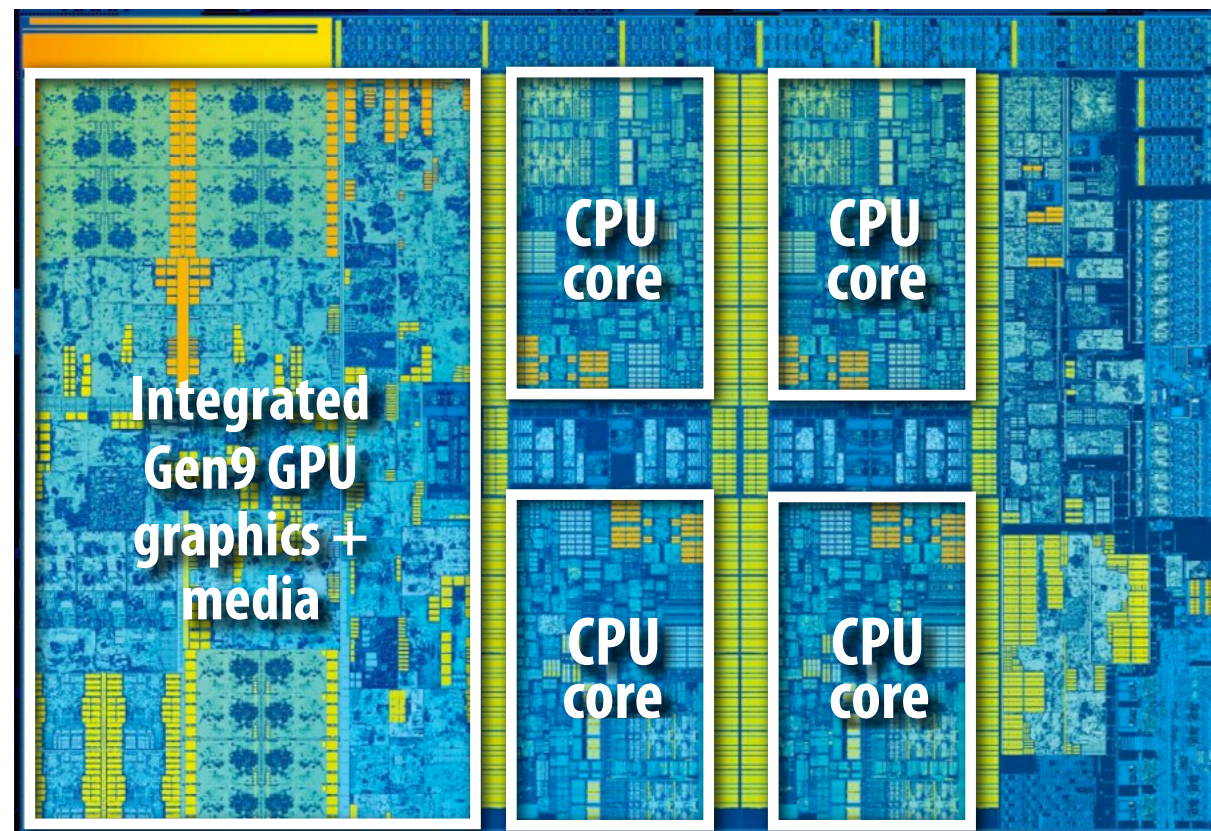WM = weighted mean over all apps

total = cost of host machine + CPU
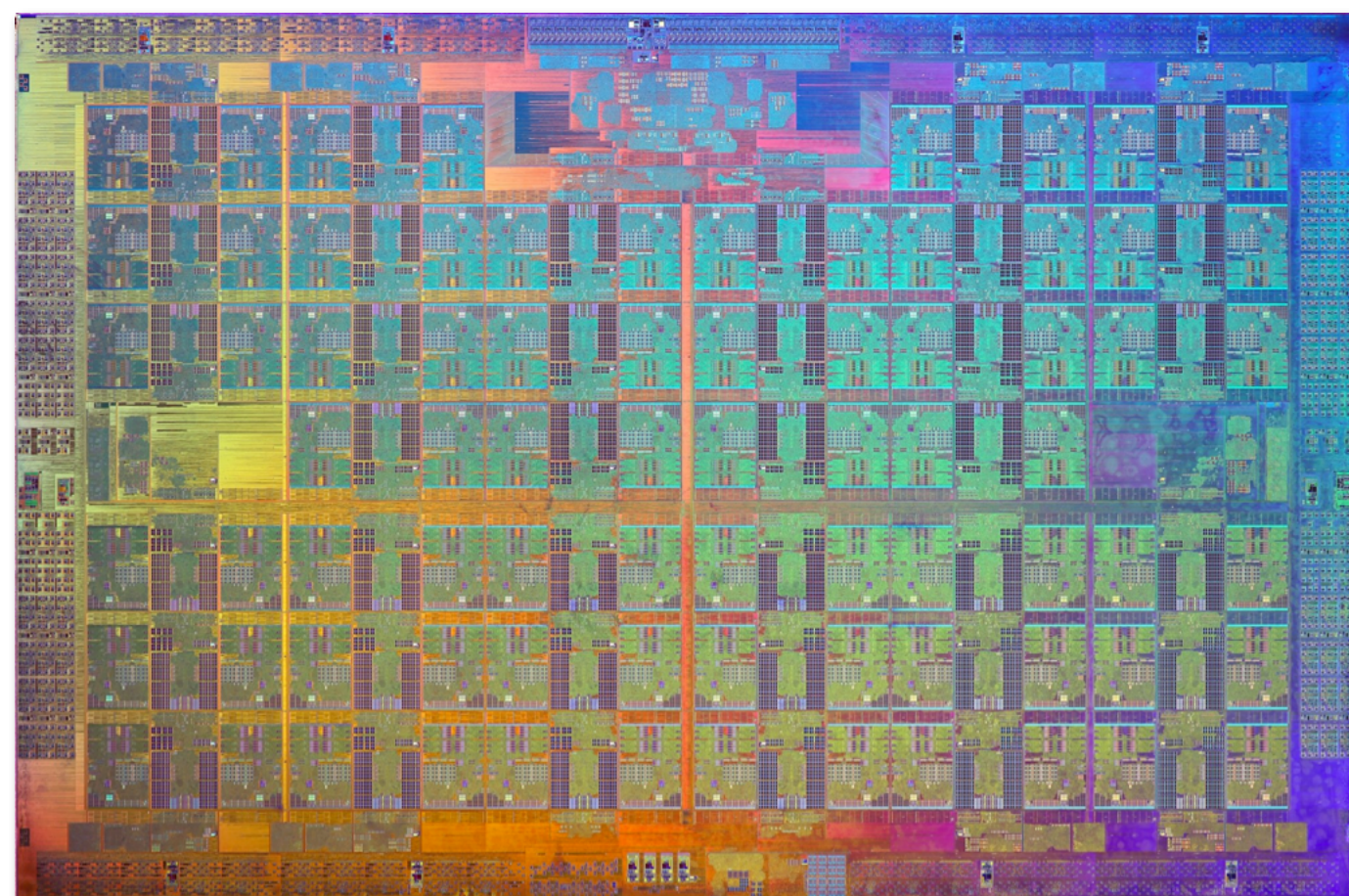incremental = only cost of TPU

Course Wrap Up

😆 😎 😂 💃 🕺

(Students)

For the foreseeable future, the primary way to obtain higher performance computing hardware is through a combination of increased parallelism and hardware specialization.
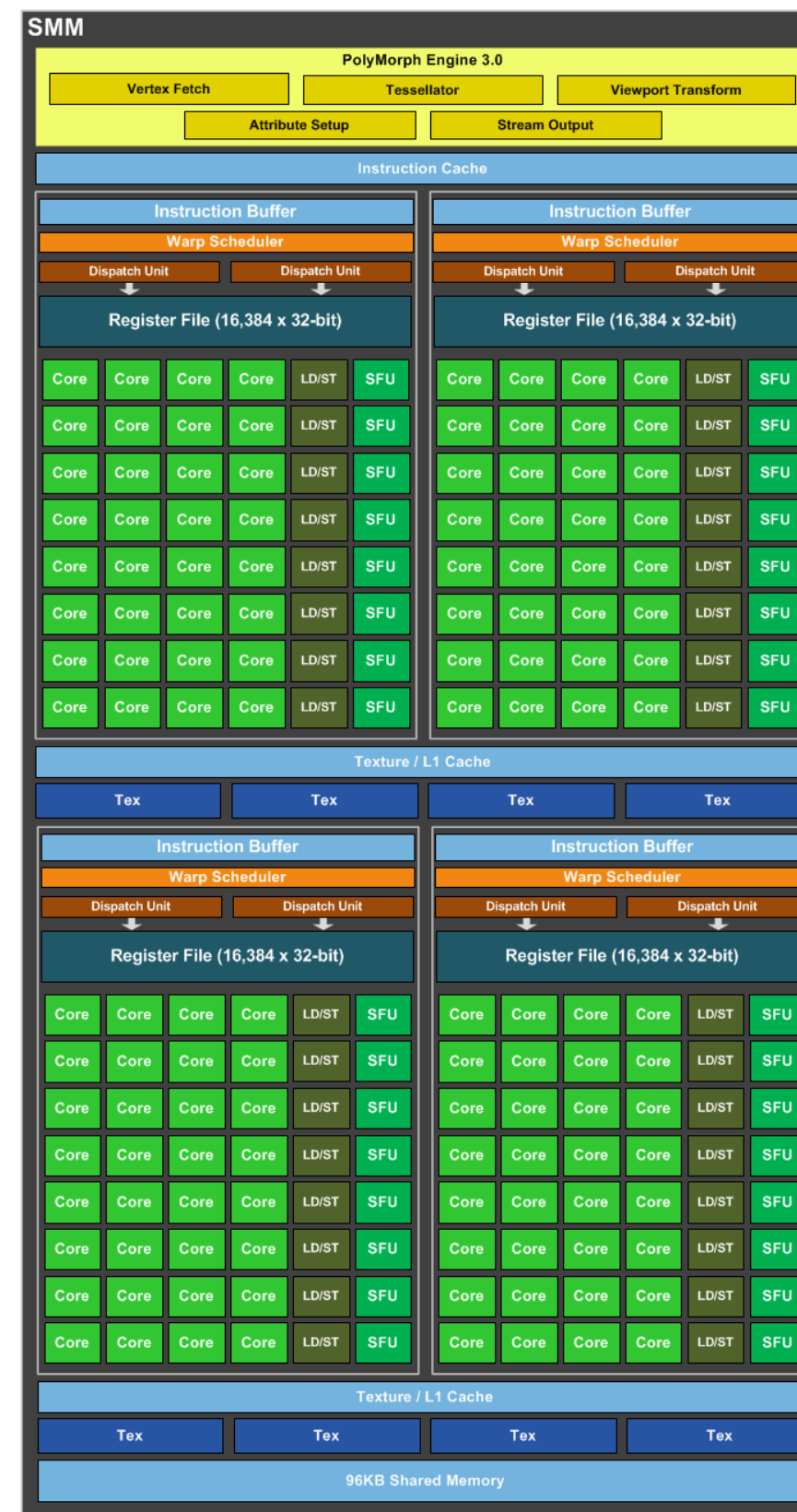

Intel Core i7 CPU + integrated GPU and media

**Intel Xeon Phi
72 cores, 16-wide SIMD, 4-way multi-threading**

Figure 3: GM204 SMM Diagram (GM204 also features 4 DP units per SMM, which are not depicted in this diagram)
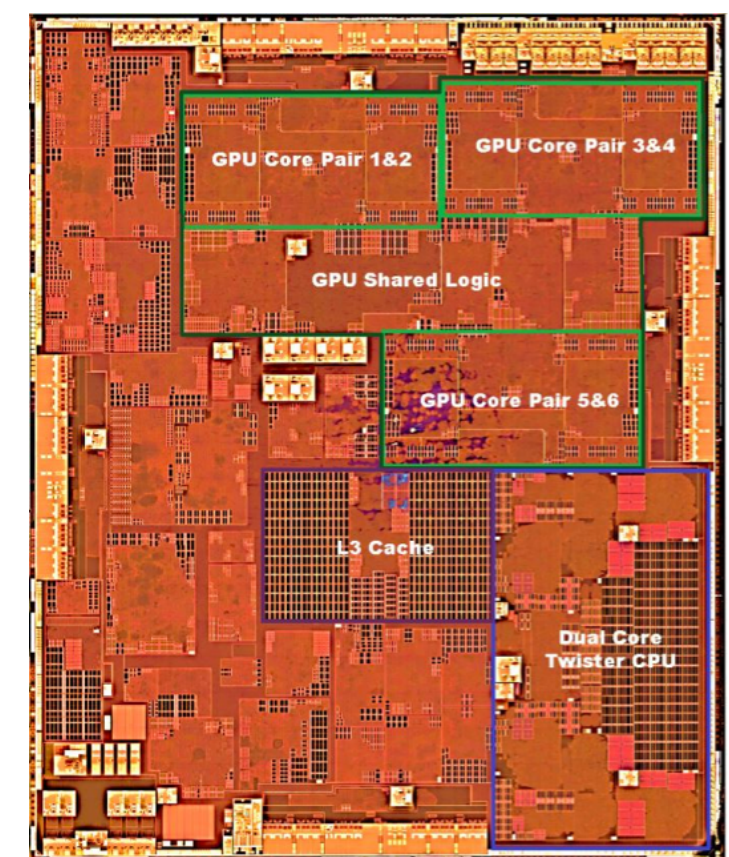
**NVIDIA Maxwell GPU
(single SMM core)
32 wide SIMD
2048 CUDA/core threads per SMM**

**FPGA
(reconfigurable logic)**

**Apple A9
Heterogeneous SoC
multi-core CPU + multi-
core GPU + media ASICs**

Today's software is surprisingly inefficient compared to the capability of modern machines

A lot of performance is currently left on the table (increasingly so as machines get more complex, and parallel processing capability grows)

Extracting this performance stands to provide a notable impact on many compute-intensive fields (or, more importantly enable new applications of computing!)

Given current software programming systems and tools, understanding how a parallel machine works is important to achieving high performance.

A major challenge going forward is making it simpler for programmers to extract performance on these complex machines.

This is very important given how exciting (and efficiency-critical) the next generation of computing applications are likely to be.

Key issues we have addressed in this course

Identifying parallelism

(or conversely, identifying dependencies)

Efficiently scheduling parallelism

1. Achieving good workload balance

2. Overcoming communication constraints:

Bandwidth limits, dealing with latency, synchronization

Exploiting data/computation locality = efficiently managing state!

3. Scheduling under heterogeneity (using the right processor for the job)

We discussed these issues at many scales and in many contexts

Heterogeneous mobile SoC

Single chip, multi-core CPU

Multi-core GPU

CPU+GPU connected via bus

Clusters of machines

Large scale, multi-node supercomputers

Key issues we have addressed in this course

Abstractions for thinking about efficient code

Data parallel thinking

Functional parallelism

Transactions

Tasks

How throughput-oriented hardware works

Multiple cores, hardware-threads, SIMD

Specialization

After taking this course, you are ready to try undergraduate research in parallel computing!

Why research (or independent study)?

- You will learn <u>way more</u> about a topic than in any class.

- You think your undergrad friends are very smart? Come hang out with Stanford Ph.D. students! (you get to work side-by-side with them and with faculty). Imagine what level you might rise to.

- It's way more fun to be on the cutting edge. Industry might not even know about what you are working on. (imagine how much more valuable you are if you can teach them)

- It widens your mind as to what is possible.

Example: what my own Ph.D. students are working on these days...

- **Generating efficient code from image processing or deep learning DSLs, and compiling these applications directly to FGPAs**

- **Designing a new shading language for future real-time 3D graphics pipelines (collaboration with NVIDIA)**

- **Parallel computing platforms that make it simpler and more efficient to analyzing large video collections (Scanner project: "Spark for video")**

- **Designing programming models for querying video collections (e.g, find frames with "three people around a table" or where DNN1 disagrees with DNN2)**

- **Designing more efficient DNNs to accelerate image processing on video**

- **Parallel rendering using 1000's of CPU cores in the cloud**

- **Analyzing 230,000 hours of news video for biases in representation.**

Thanks for being a great class!

Good luck on your finals!

p.s. See you in a week!