Lecture 19:

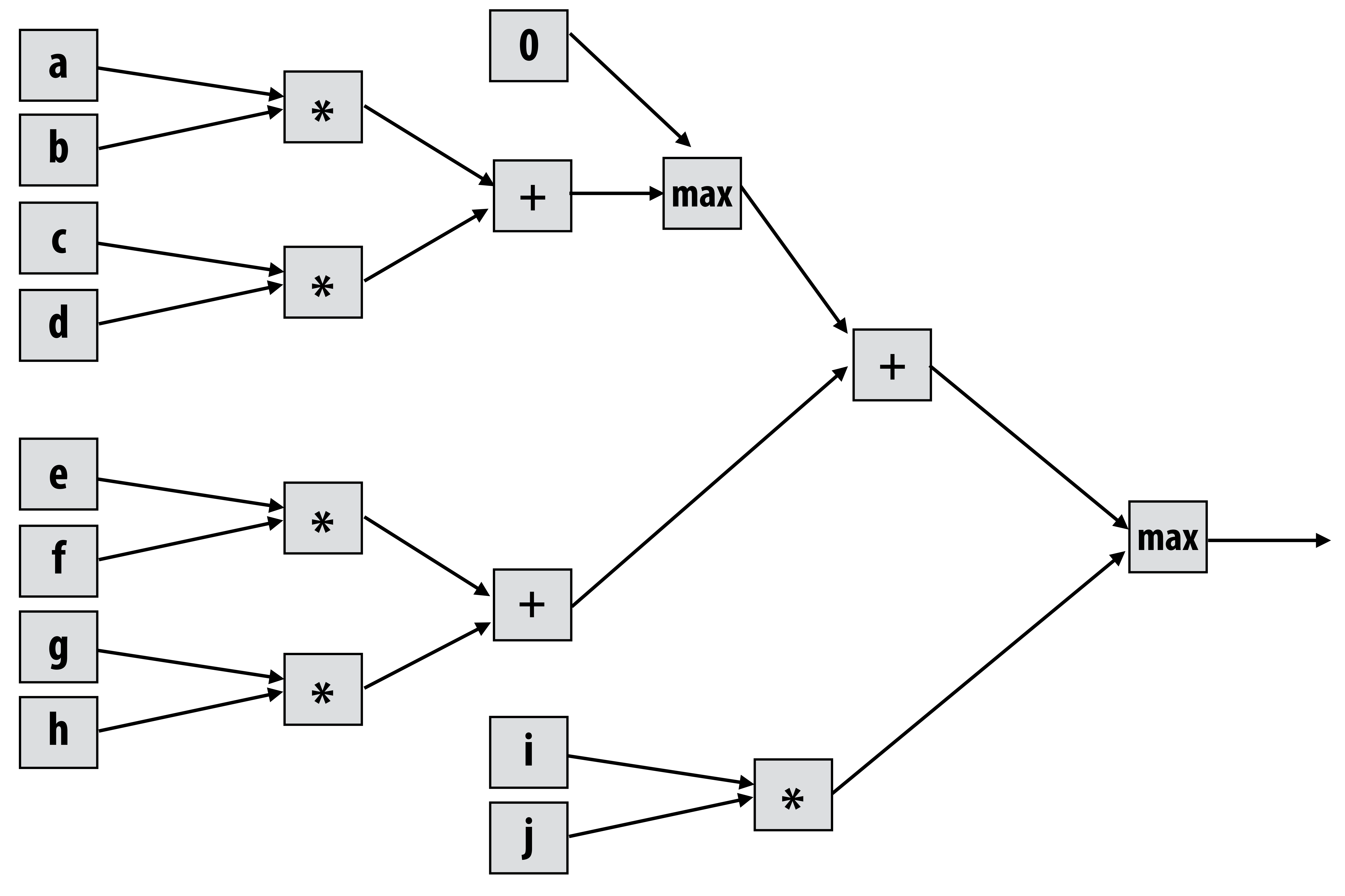
Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2021

Today

- We will discuss the workload of evaluating 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)
  - But, image processing is not the application driving the majority of DNN evaluation in the world right now (its text processing, speech, ads, etc.)

Consider the following expression

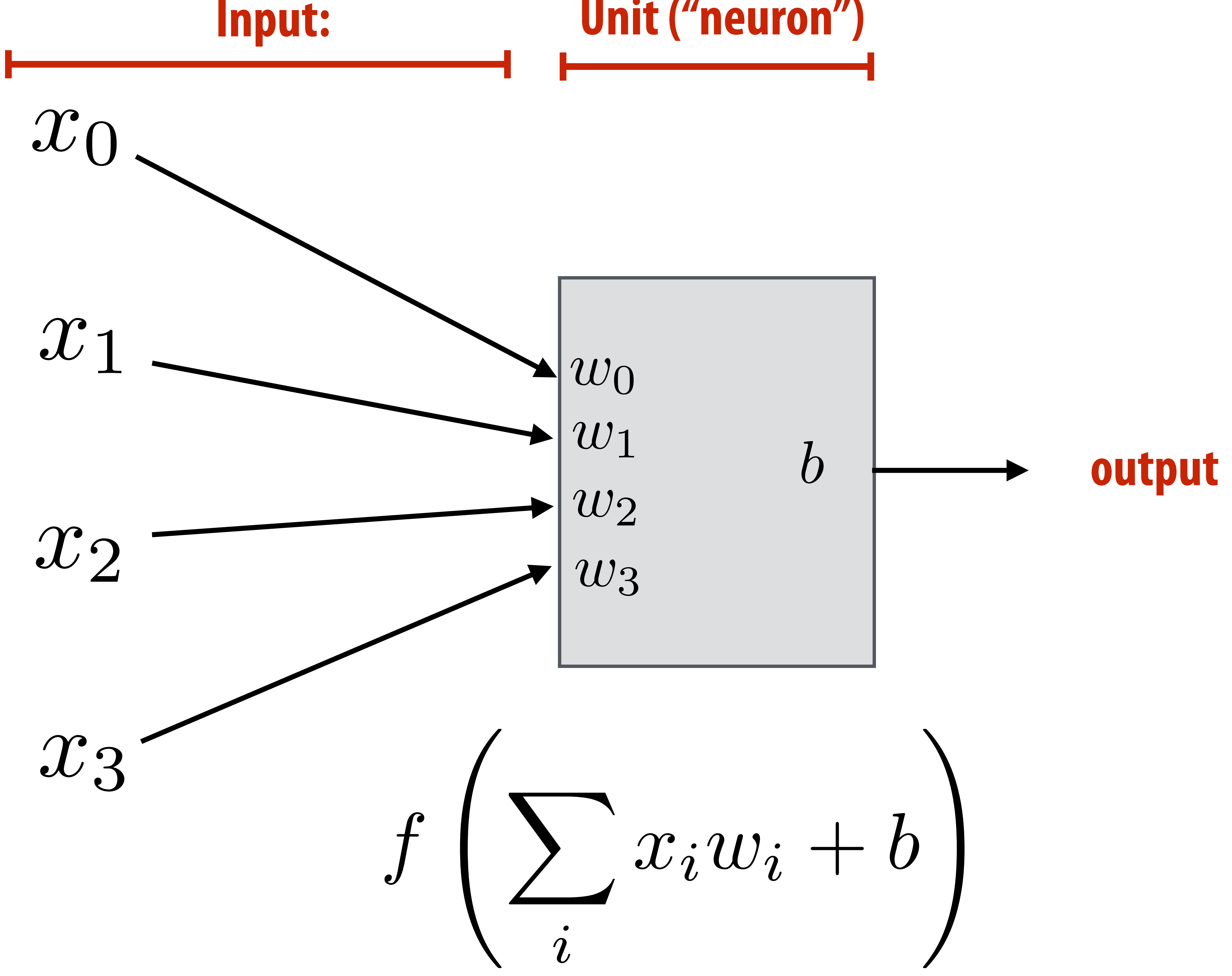



```
max( max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j)
```

What is a deep neural network?

A basic unit:

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



Example: rectified linear unit (ReLU)

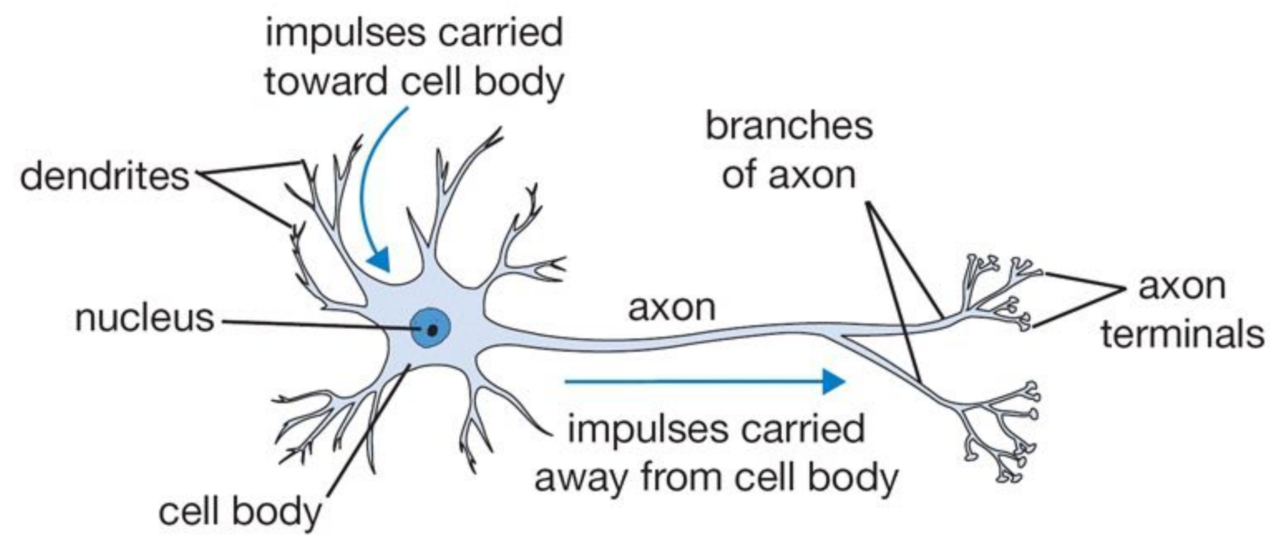
$f(x) = max(0, x)$

Basic computational interpretation:

It is just a circuit!

Biological inspiration:

unit output corresponds loosely to activation of neuron

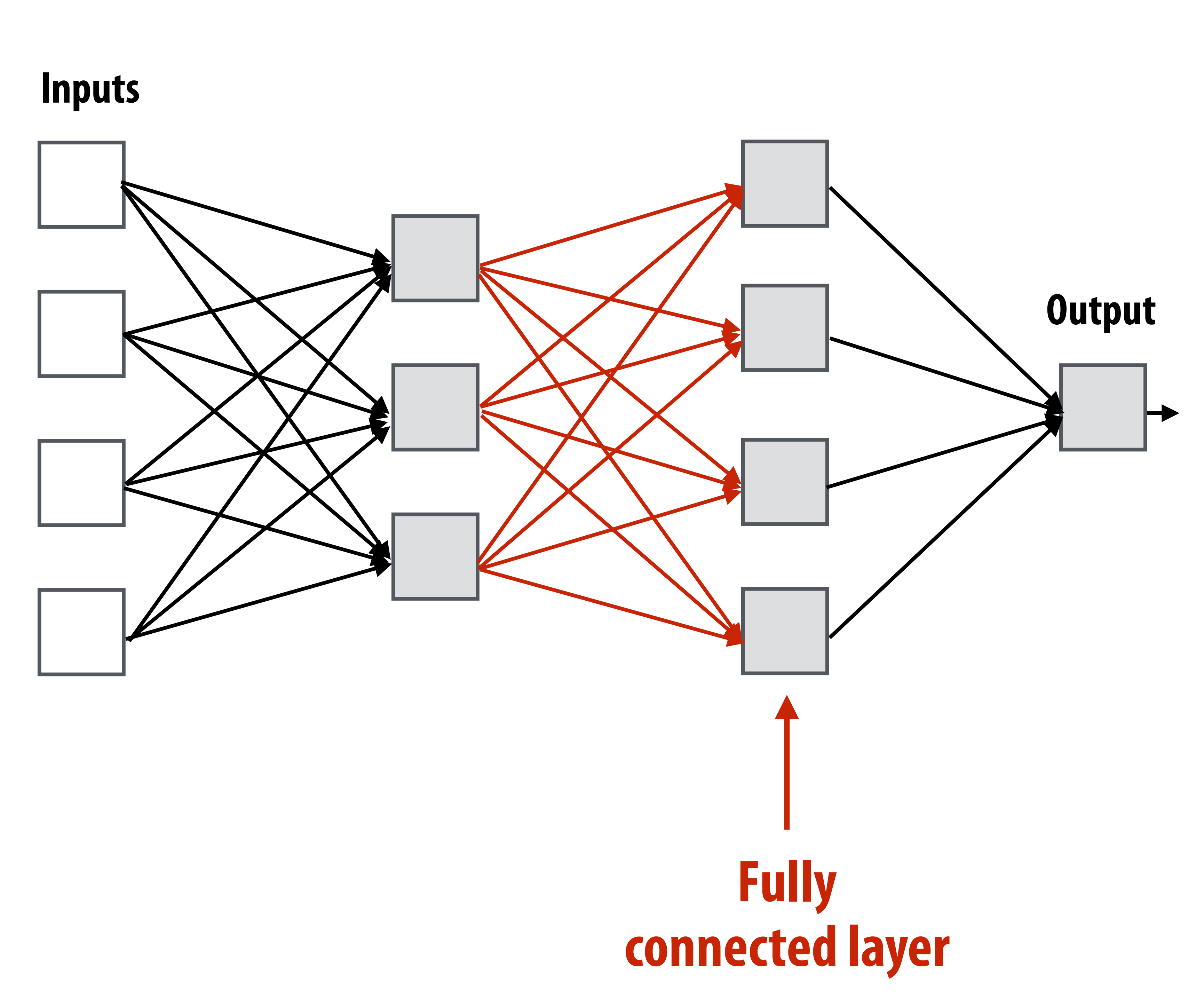


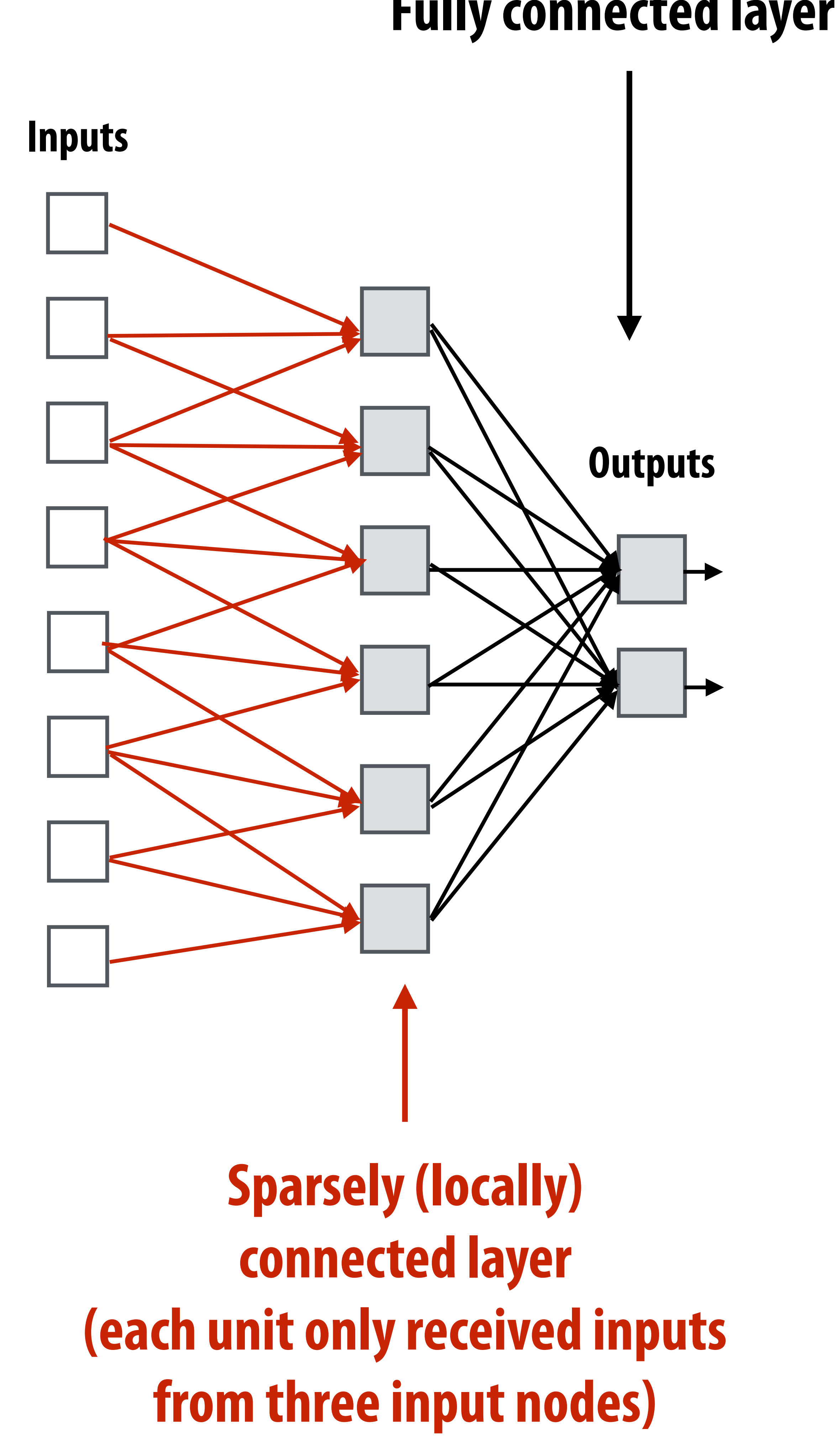
Machine learning interpretation:

binary classifier: interpret output as the probability of one class

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

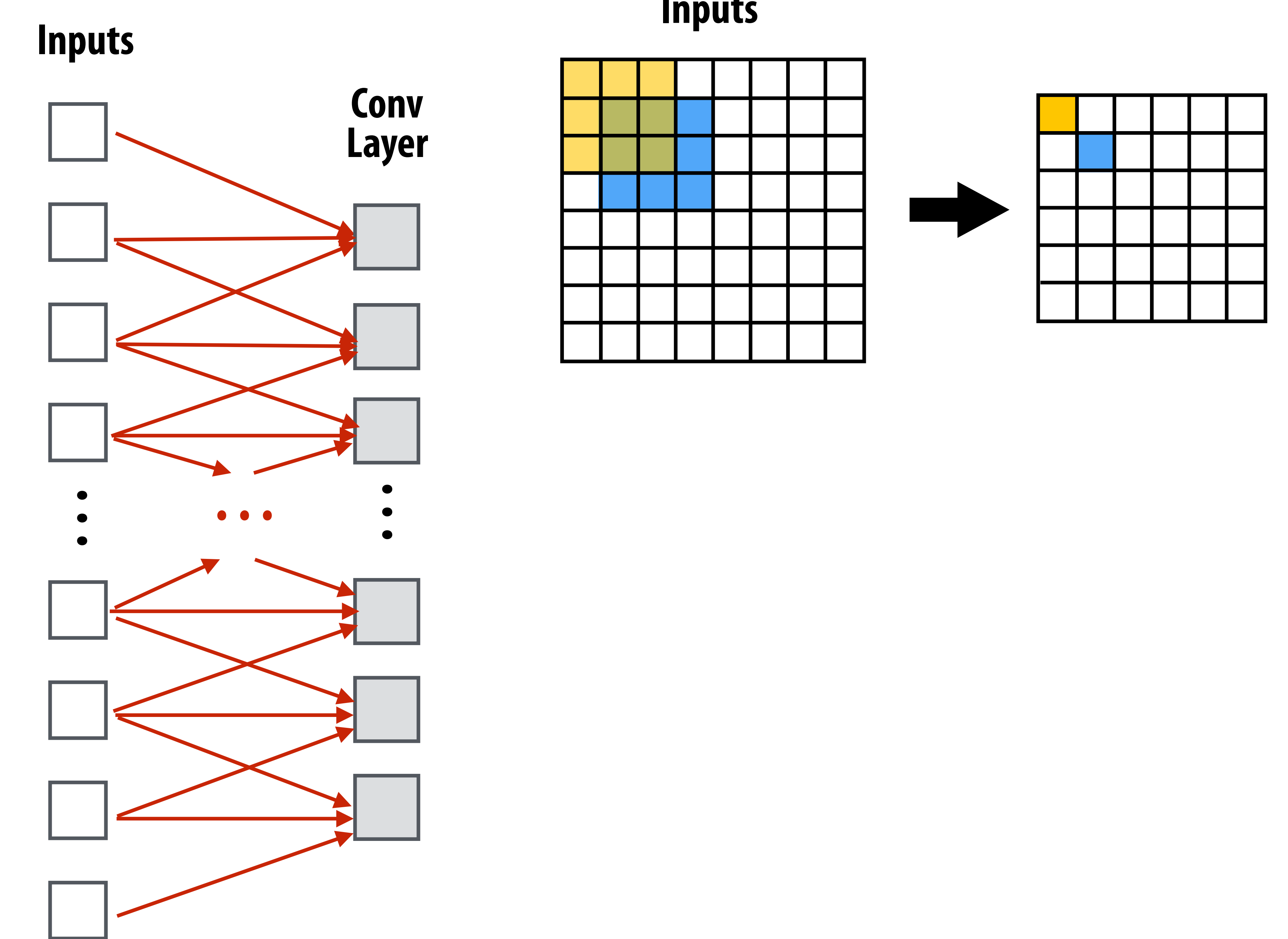
Deep neural network: topology





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};


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;
  }
}
```



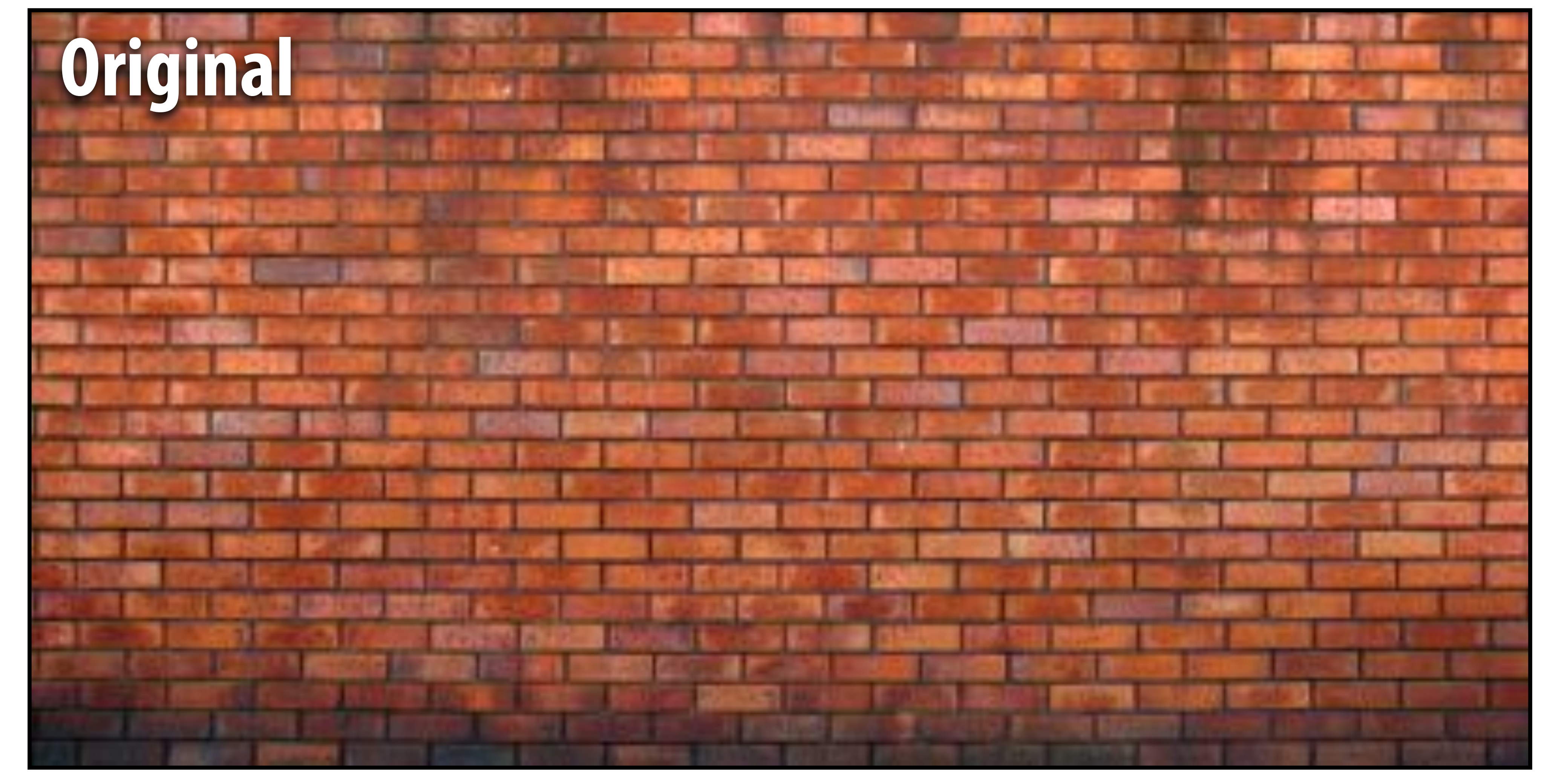
Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):

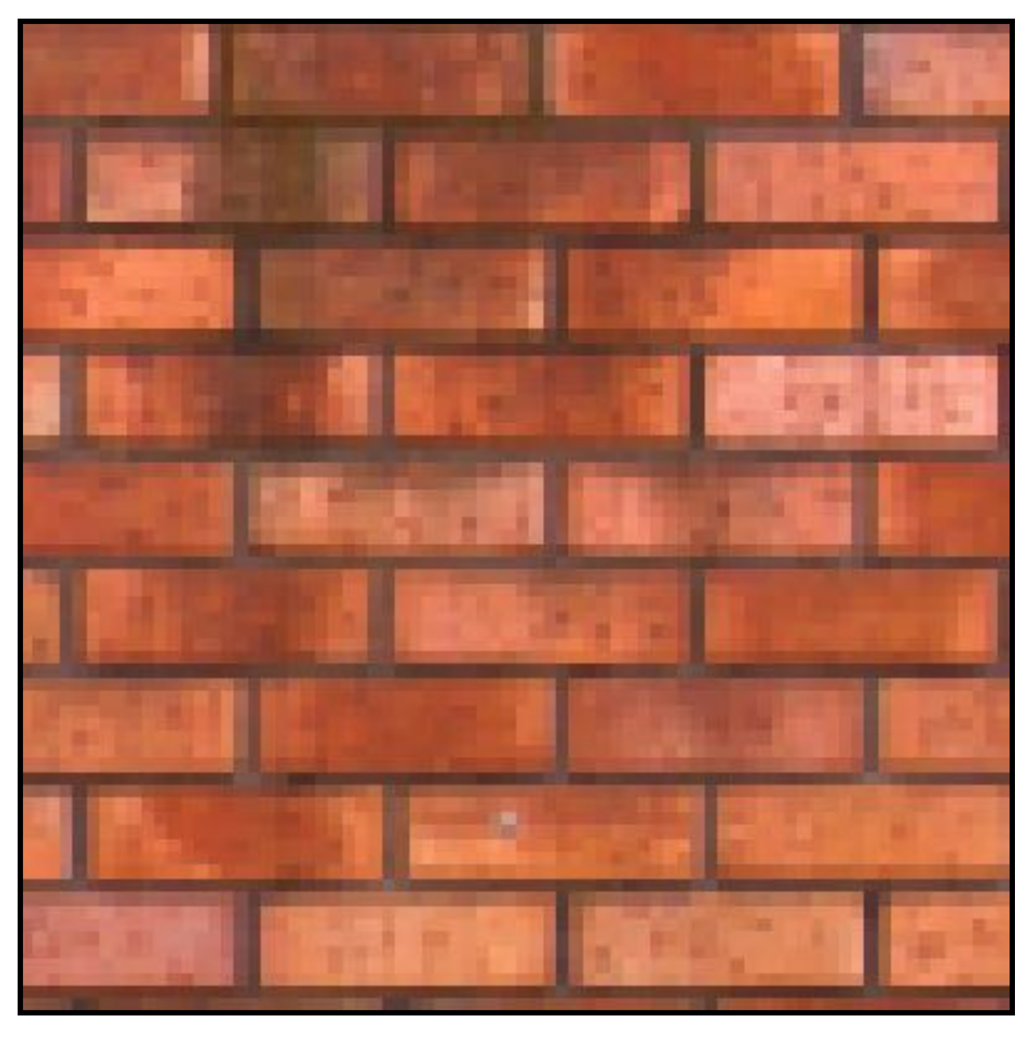
(note: network illustration above only shows links for a 1D conv: a.k.a. one iteration of `ii` loop)

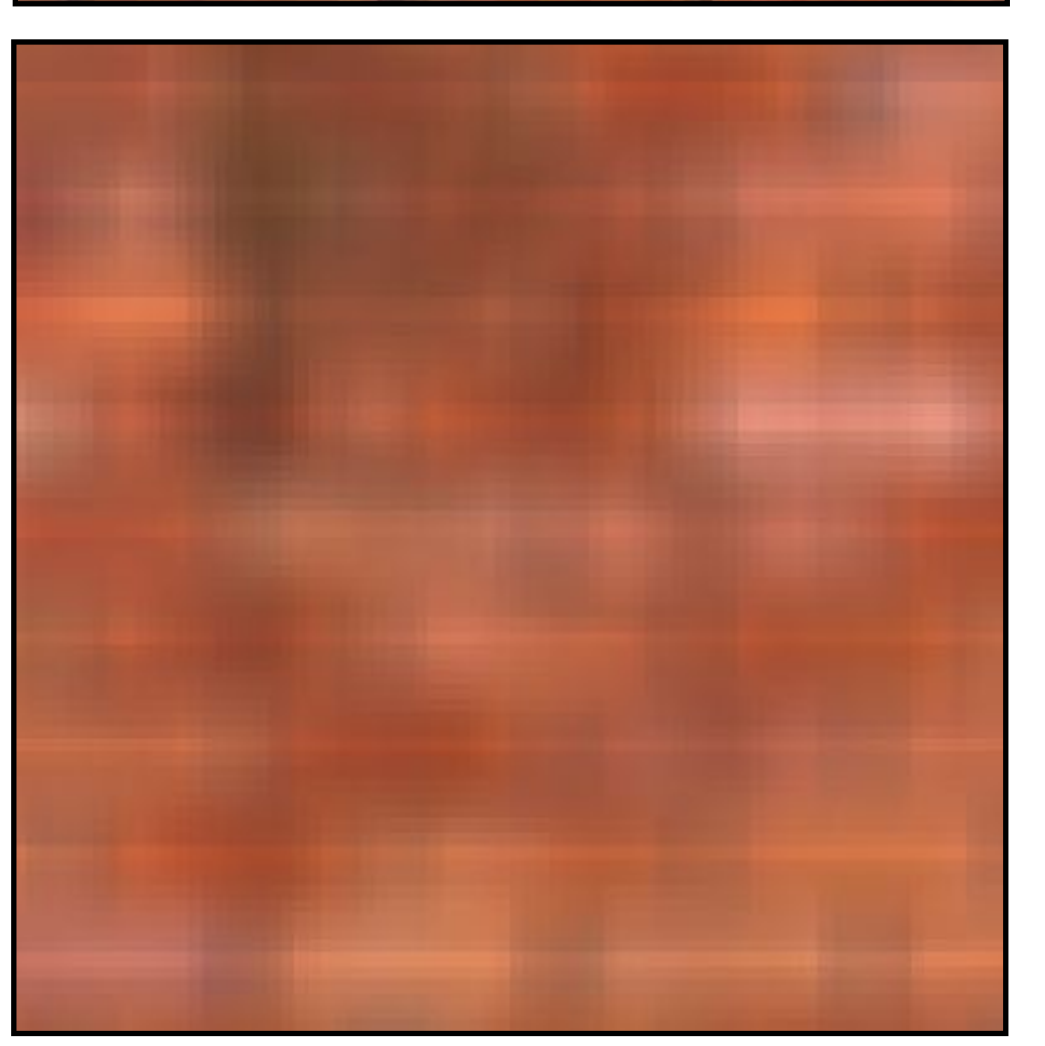
What does convolution using these filter weights do?

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

"Box blur"







What does convolution with these filters do?

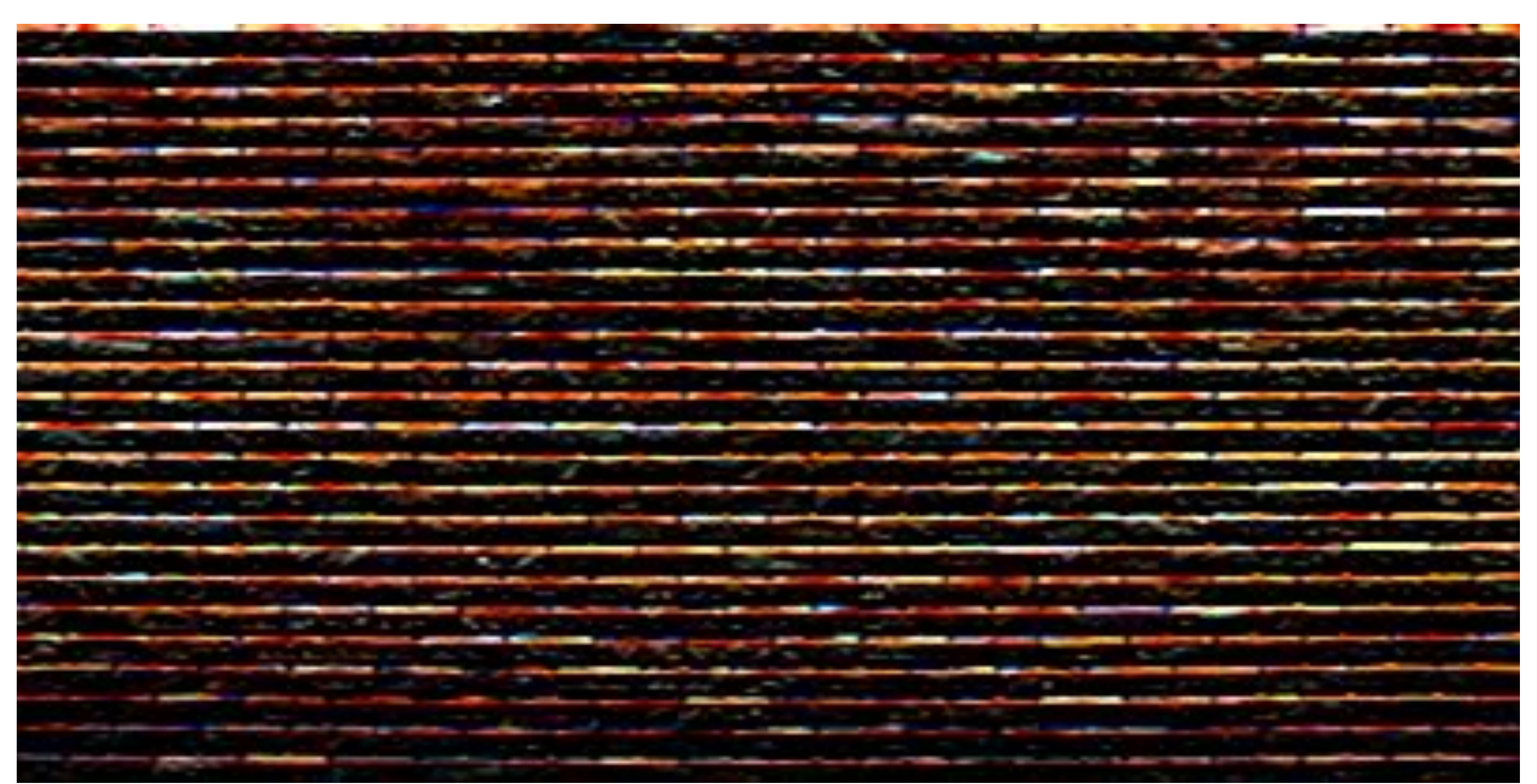
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Extracts horizontal gradients

Extracts vertical gradients

Gradient detection filters

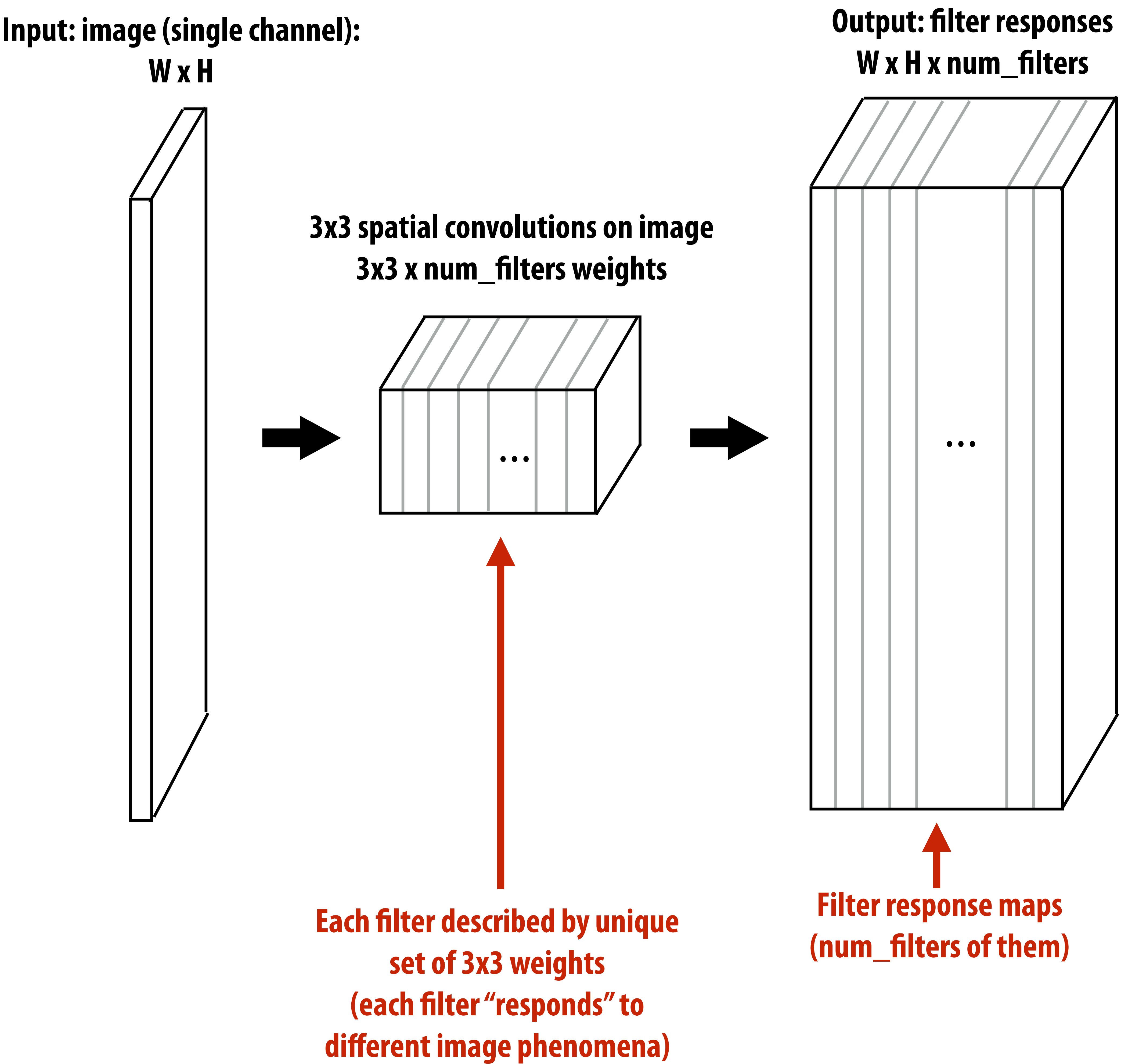
Horizontal gradients



Vertical gradients

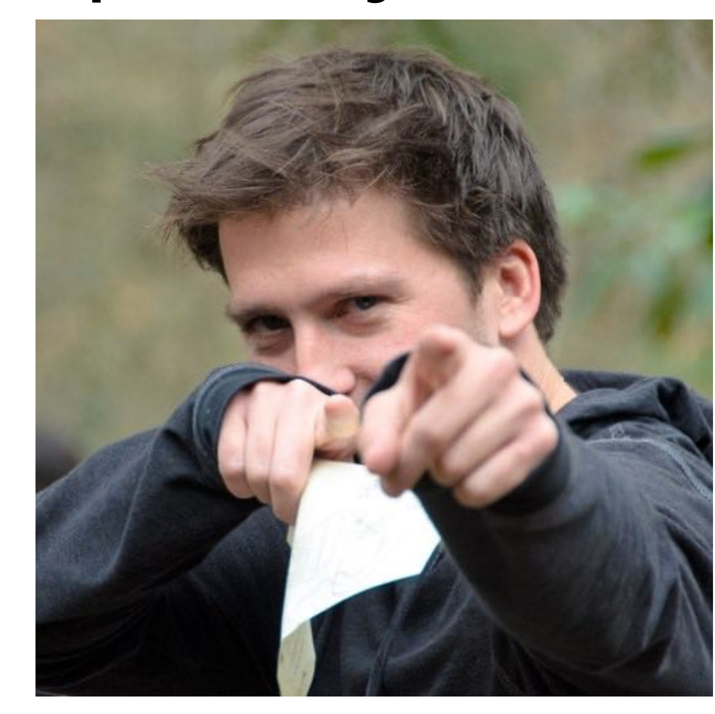
Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image

Applying many filters to an image at once

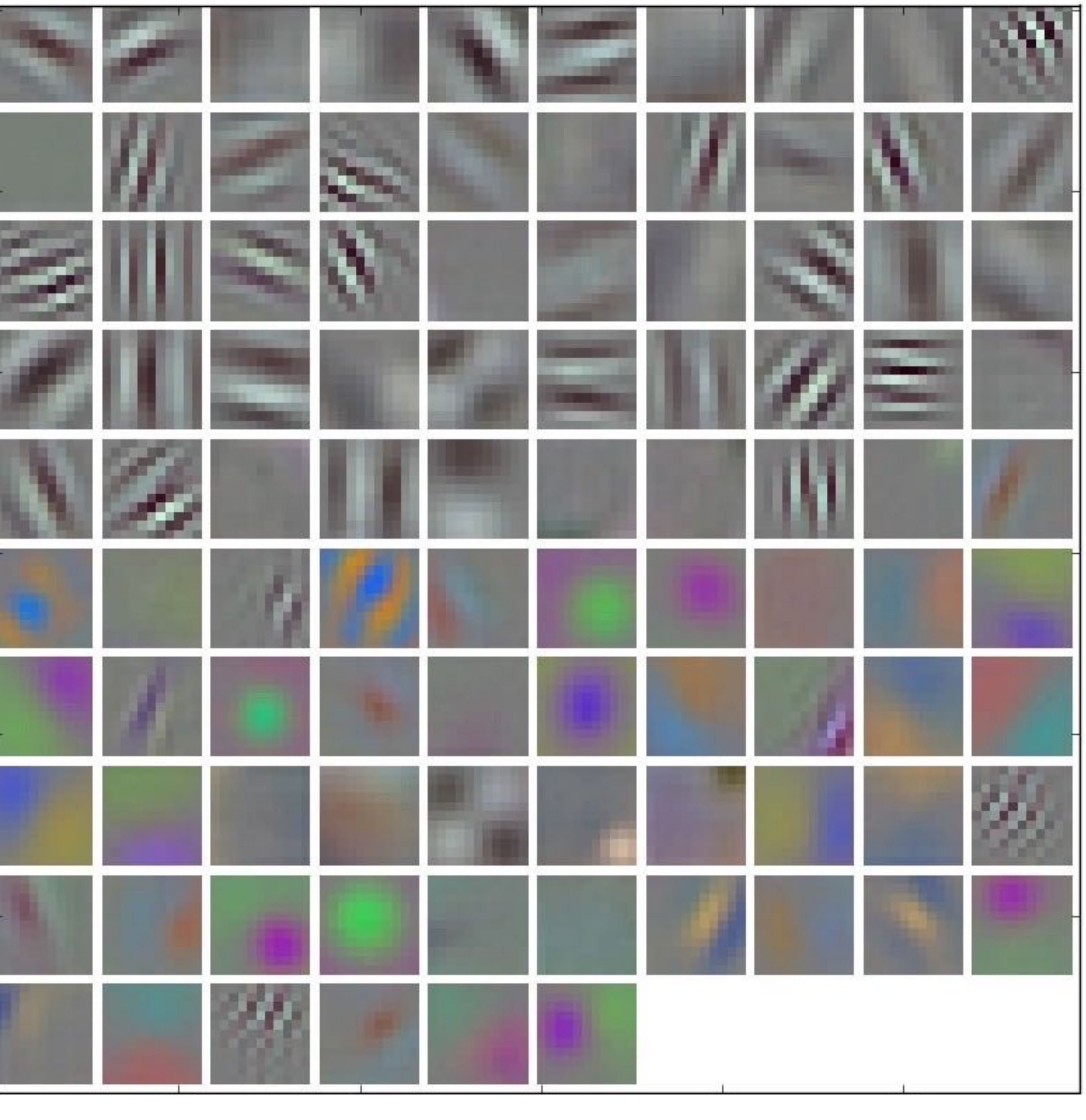


Applying many filters to an image at once

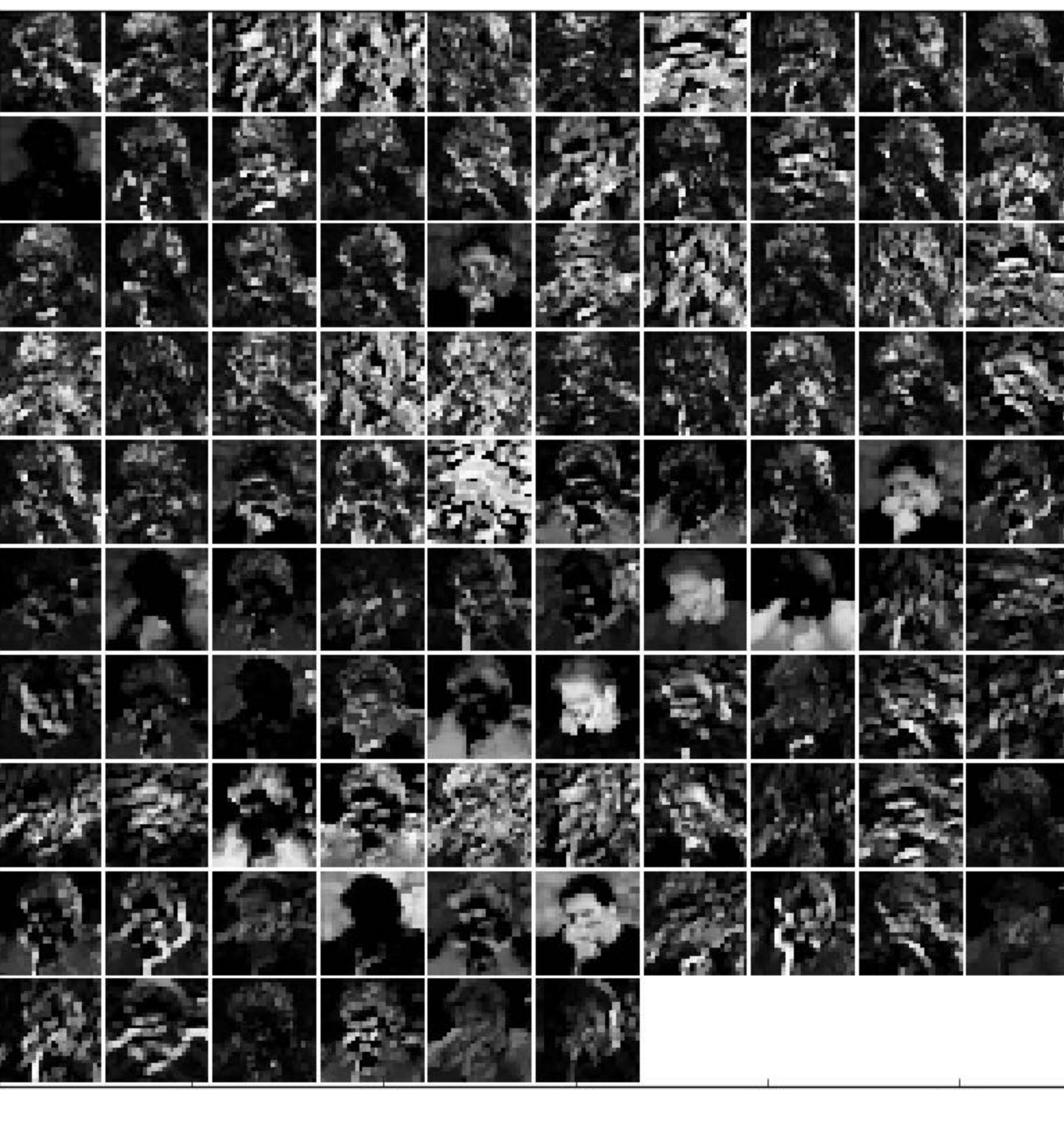
Input RGB image (W x H x 3)



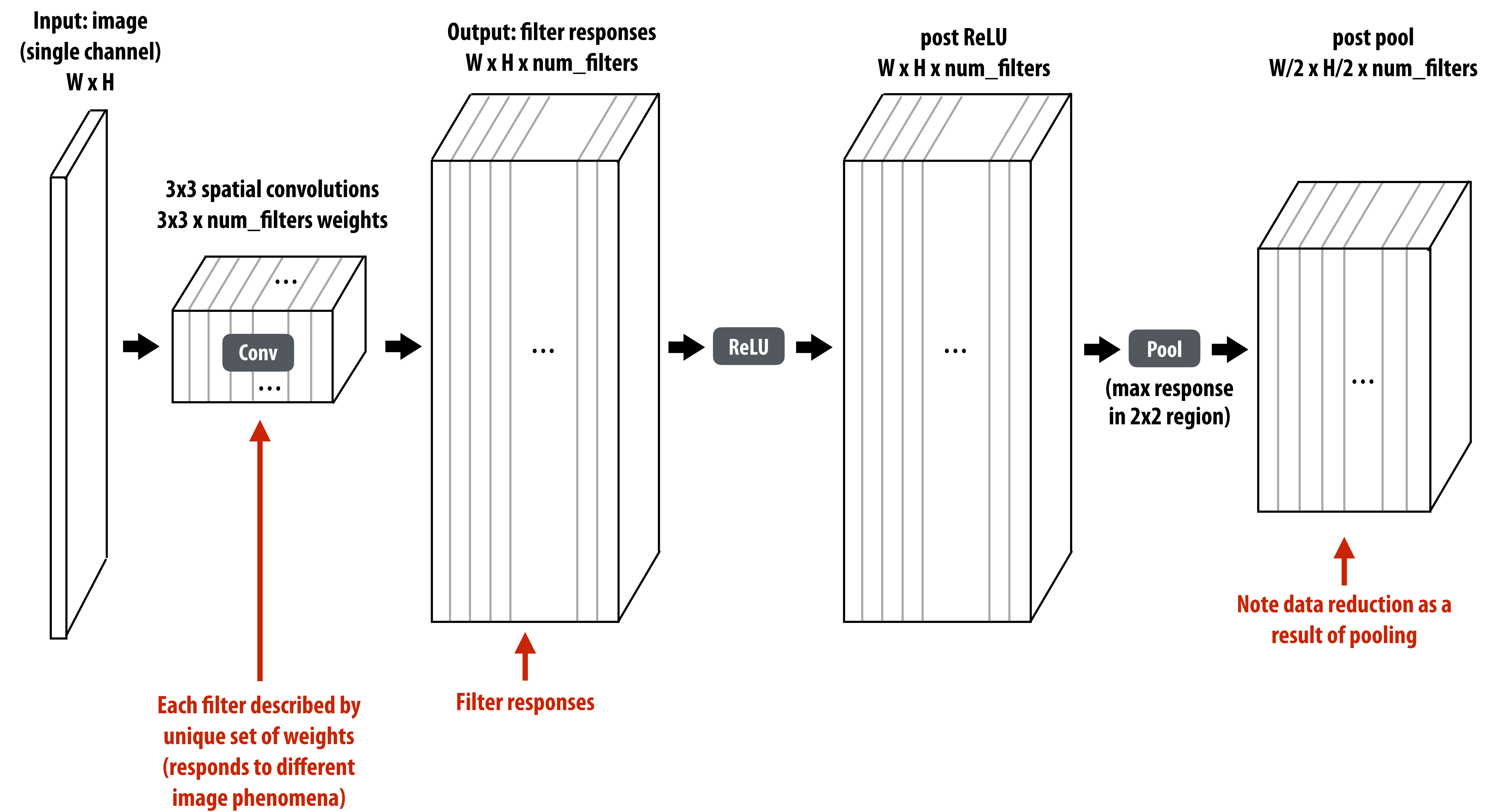
96 11x11x3 filters
(operate on RGB)



96 responses (normalized)



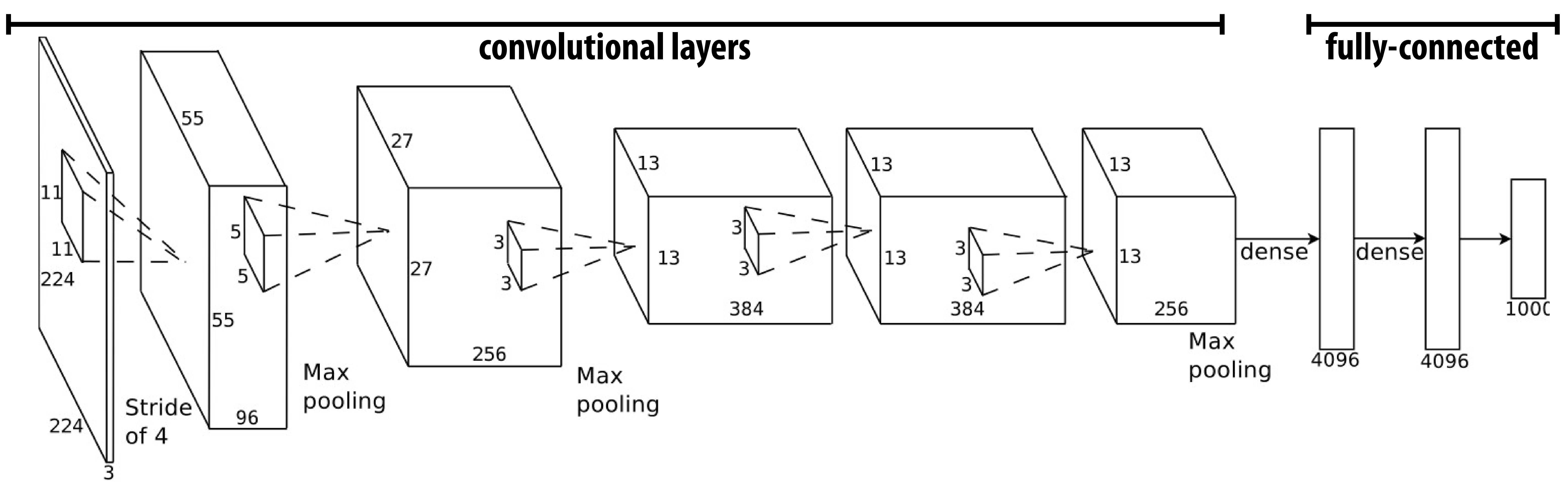
Adding additional layers



Example: "AlexNet" object detection network

Sequences of conv + reLU + pool (optional) layers

Example: AlexNet [Krizhevsky12]: 5 convolutional layers + 3 fully connected layers



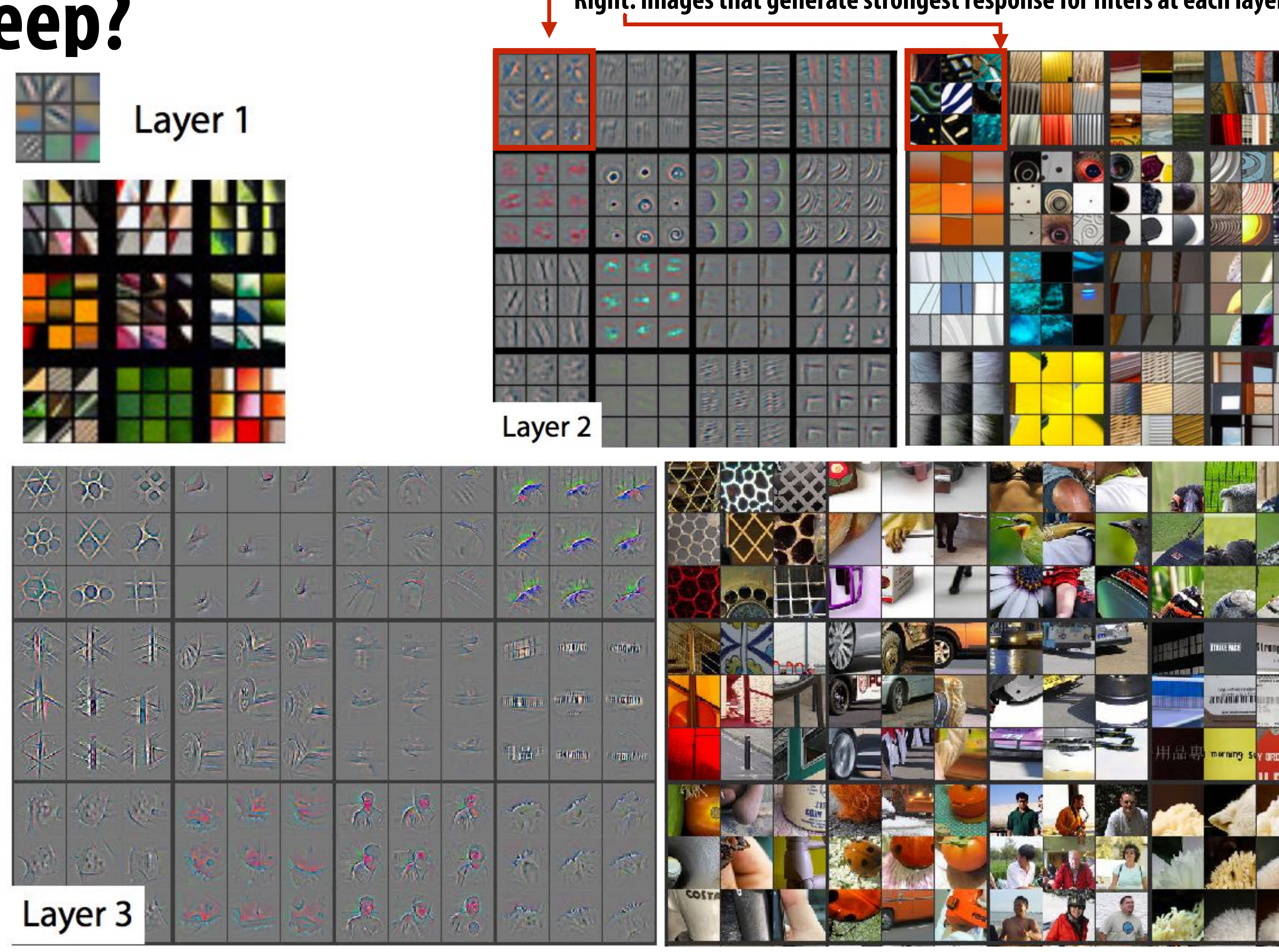
Another example: VGG-16 [Simonyan15]: 13 convolutional layers

input: 224 x 224 RGB
conv/reLU: 3x3x3x64
conv/reLU: 3x3x64x64
maxpool
conv/reLU: 3x3x64x128
conv/reLU: 3x3x128x128
maxpool

conv/reLU: 3x3x128x256
conv/reLU: 3x3x256x256
conv/reLU: 3x3x256x256
maxpool
conv/reLU: 3x3x256x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool

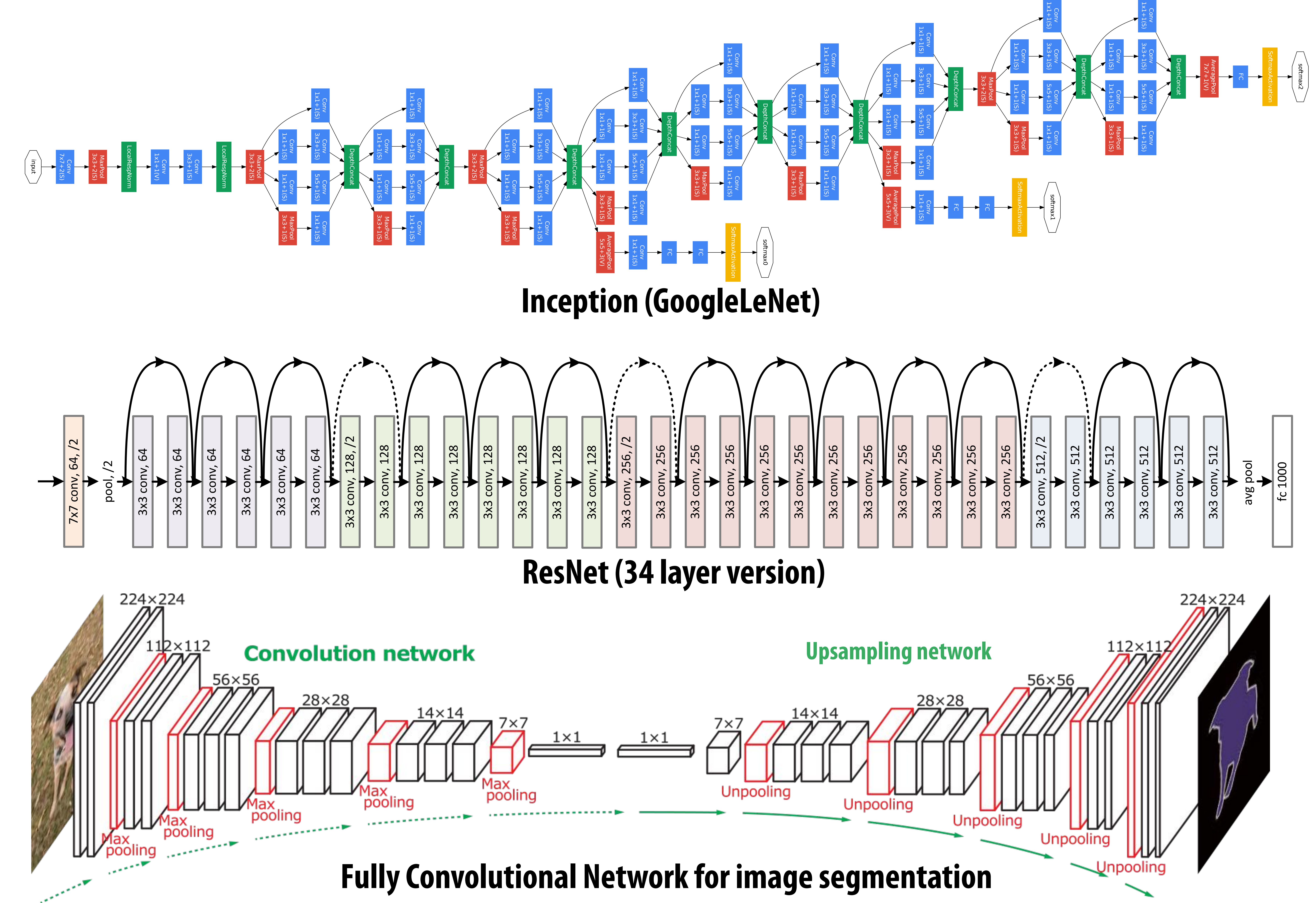
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
conv/reLU: 3x3x512x512
maxpool
fully-connected 4096
fully-connected 4096
fully-connected 1000
soft-max

Why deep?



[image credit: Zeiler 14]

More recent image understanding networks



Inception (GoogleLeNet)

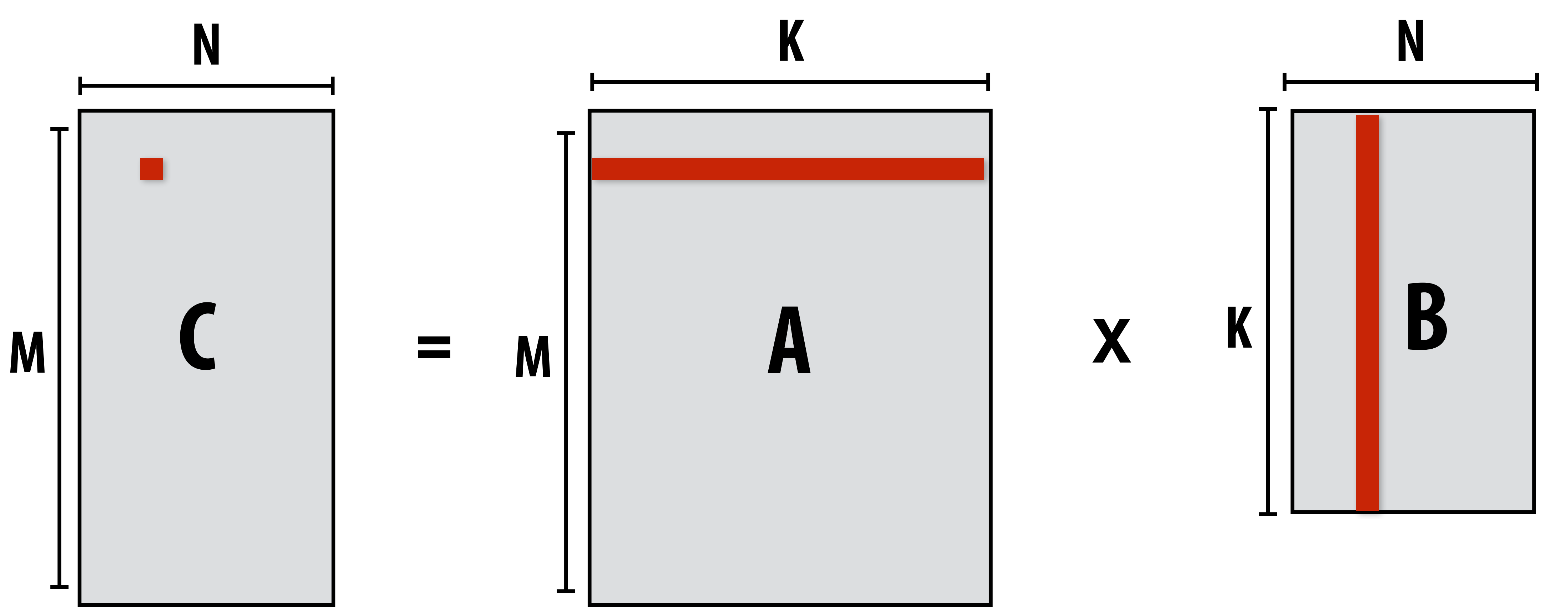
ResNet (34 layer version)

Fully Convolutional Network for image segmentation

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];
```

N K N
M C = M A x K B

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?

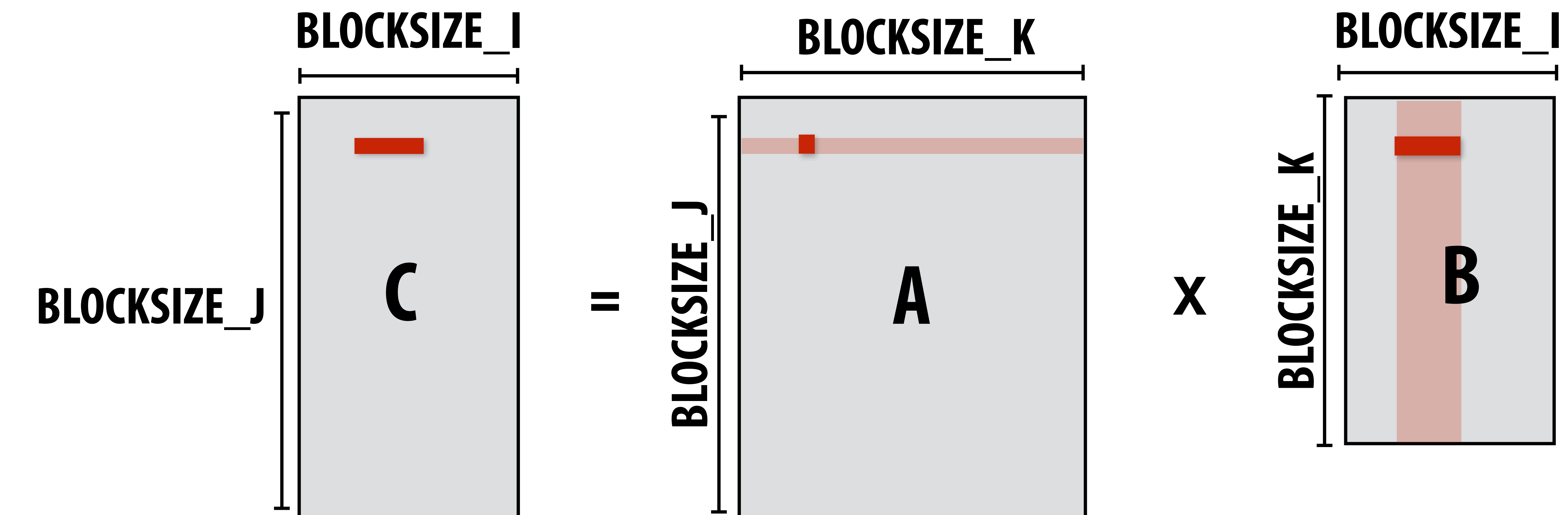
Hierarchical blocked matrix mult

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

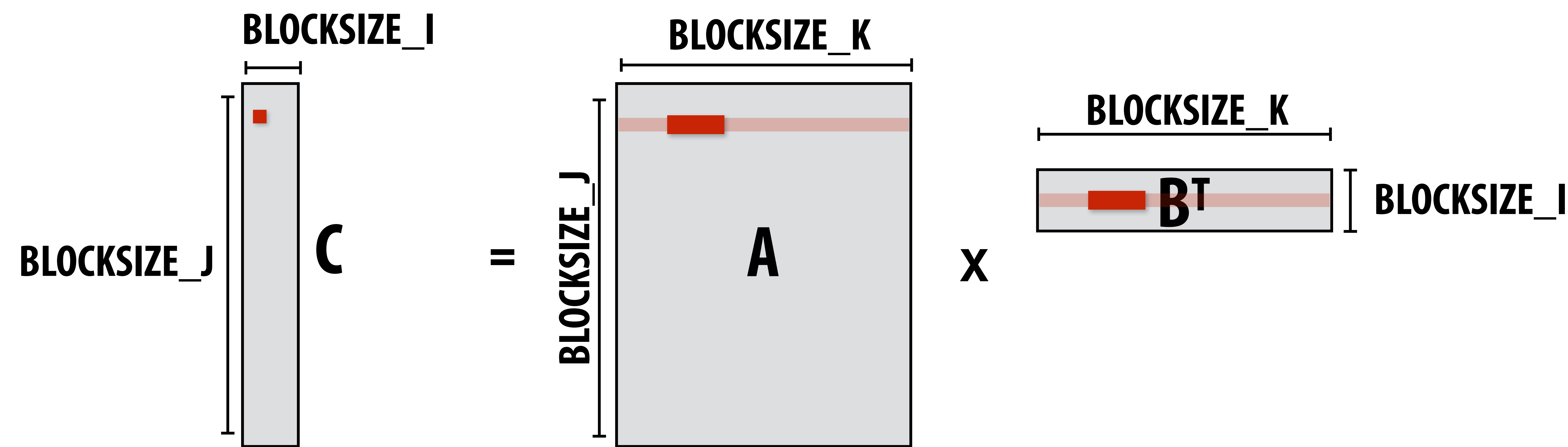
```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
      simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in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



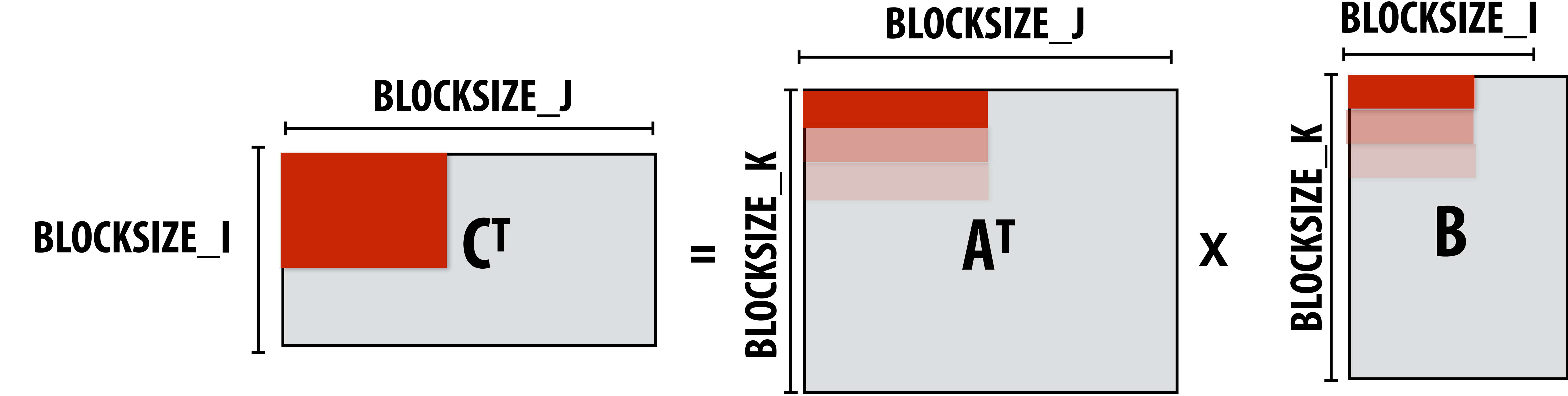

```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      // C_scalar += dot(row of A,row of B)
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
        C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][[kblock+k]);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

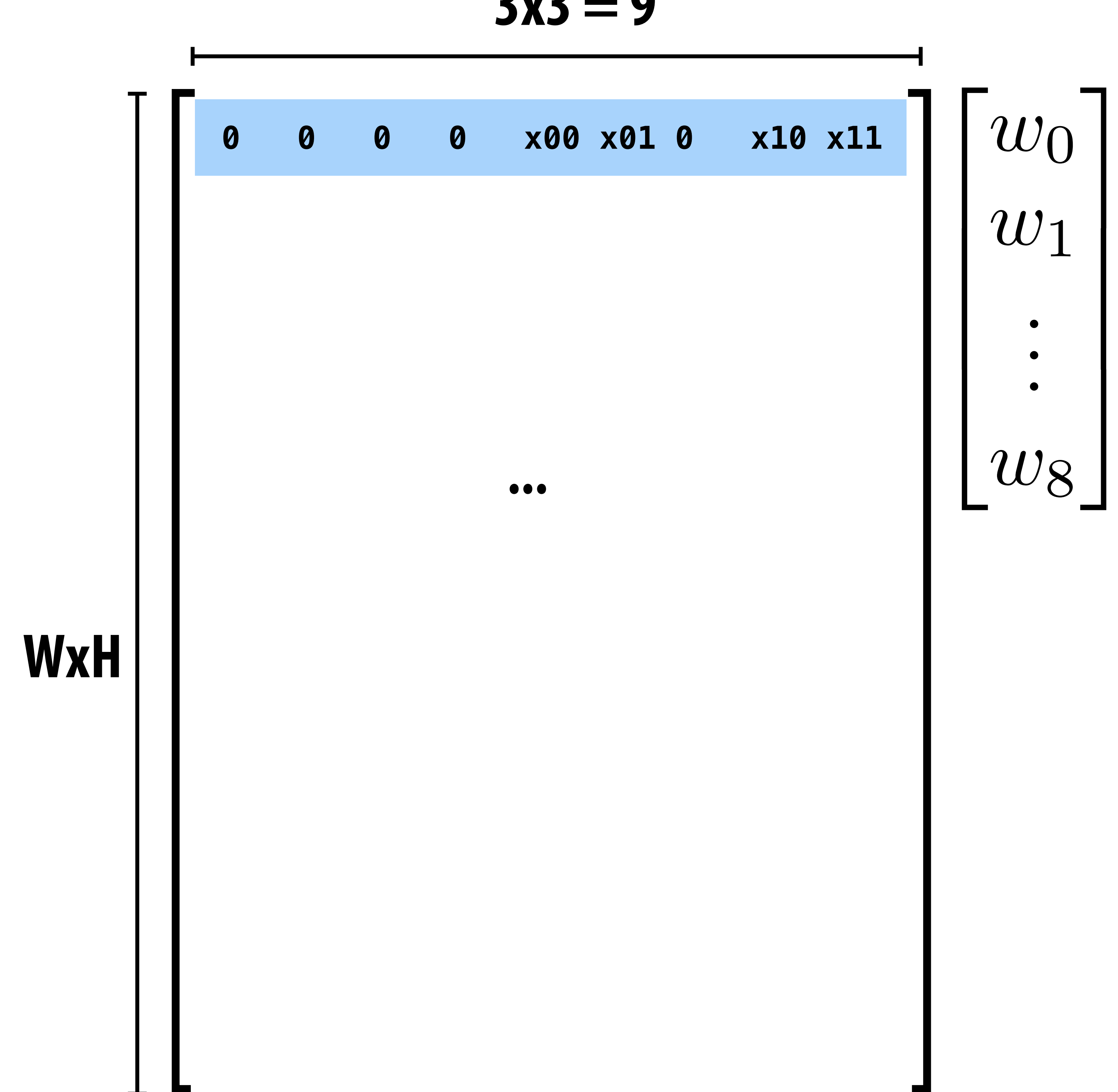
Convolution as matrix-vector product

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

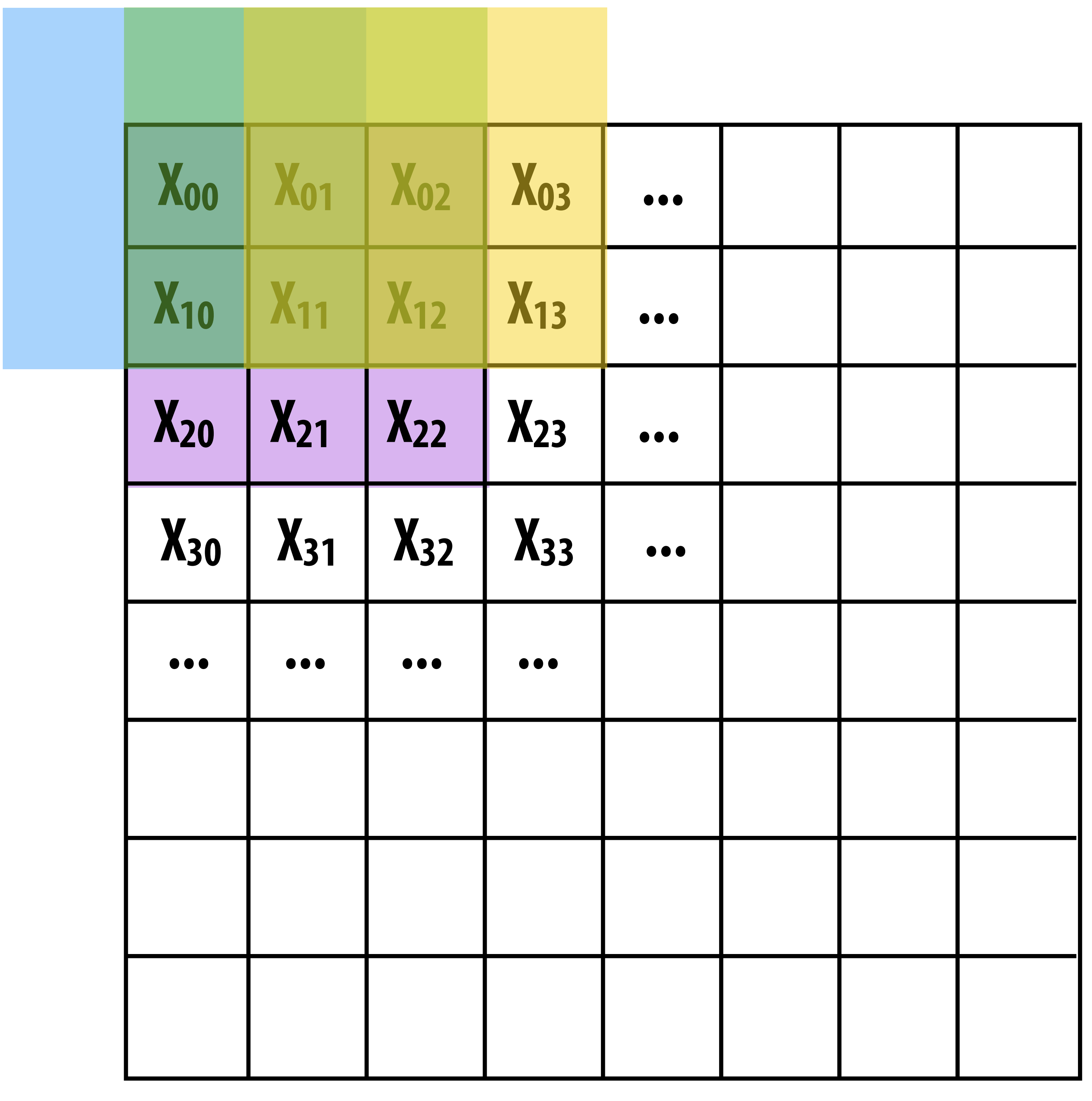
Must construct input data matrix



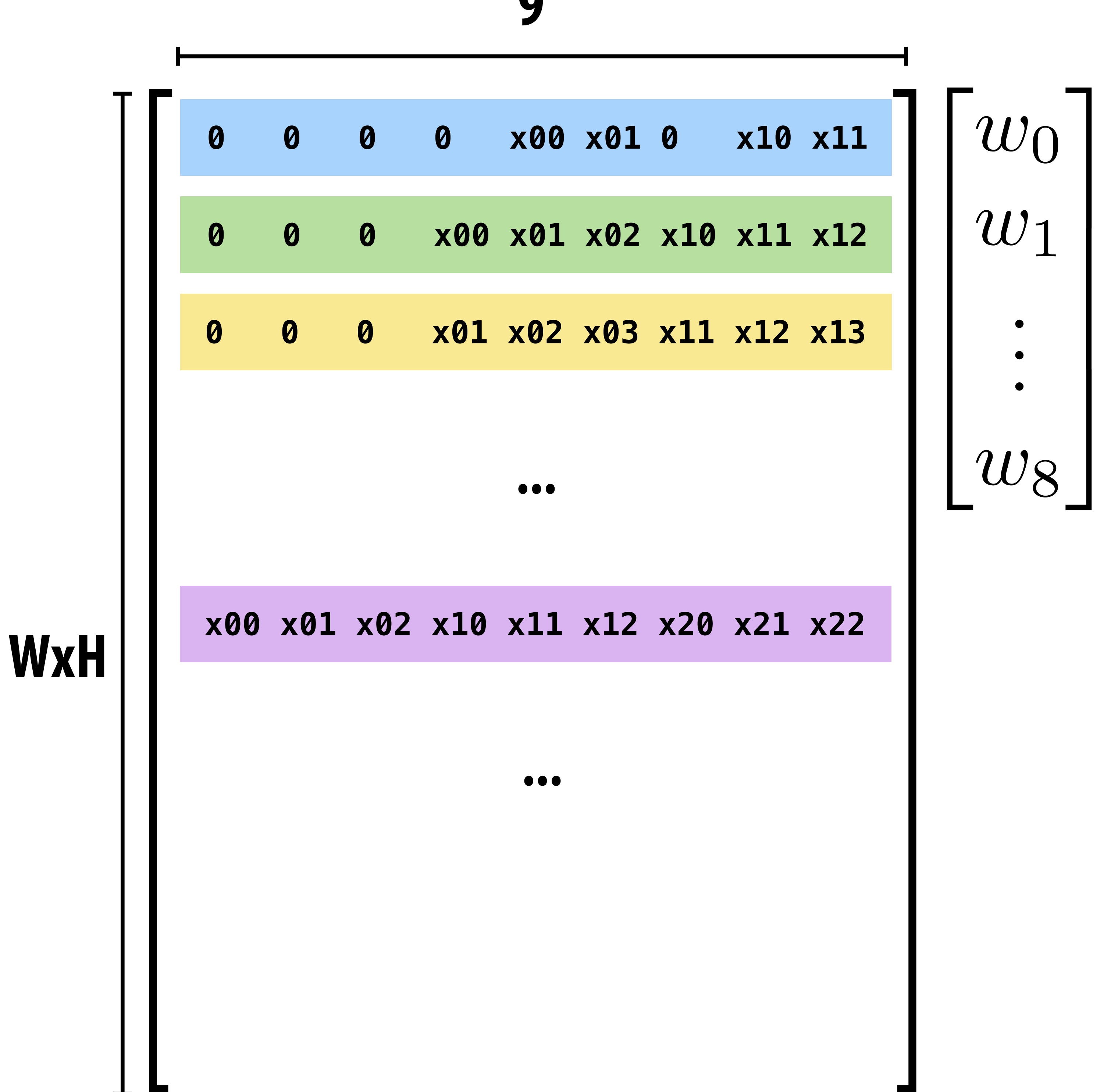
Note: 0-pad matrix

3x3 convolution as matrix-vector product

Construct matrix from elements of input image

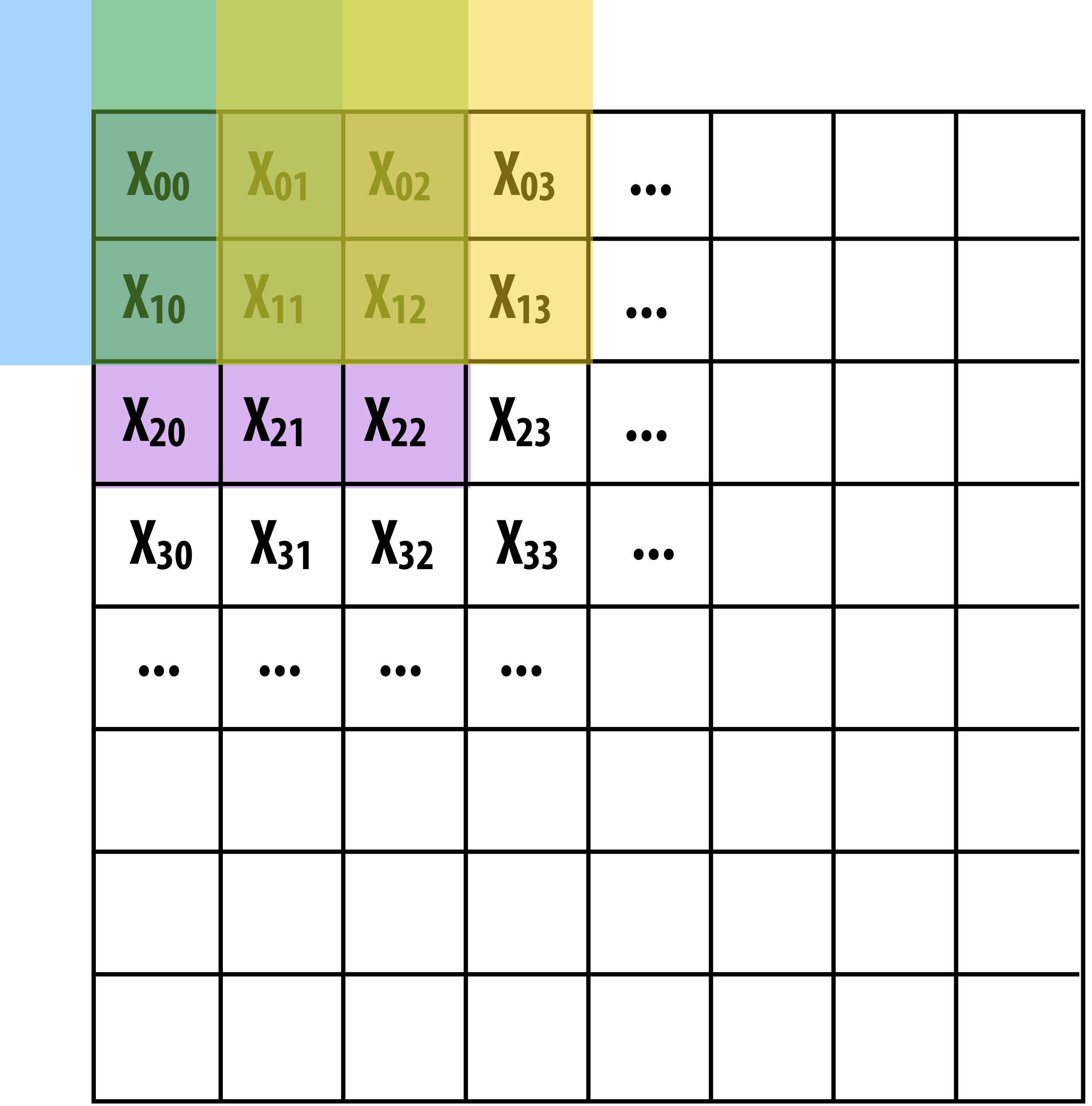


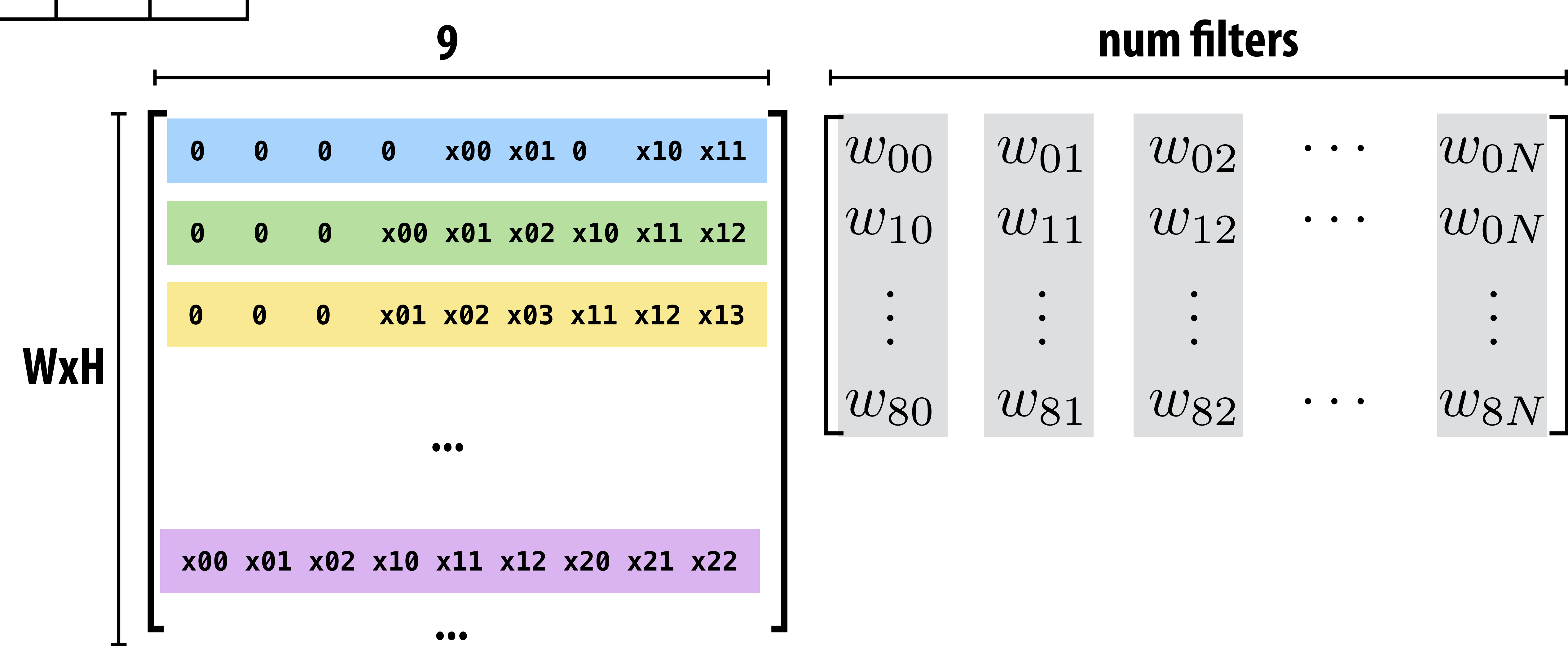
O(N) storage overhead for filter with N elements
Must construct input data matrix



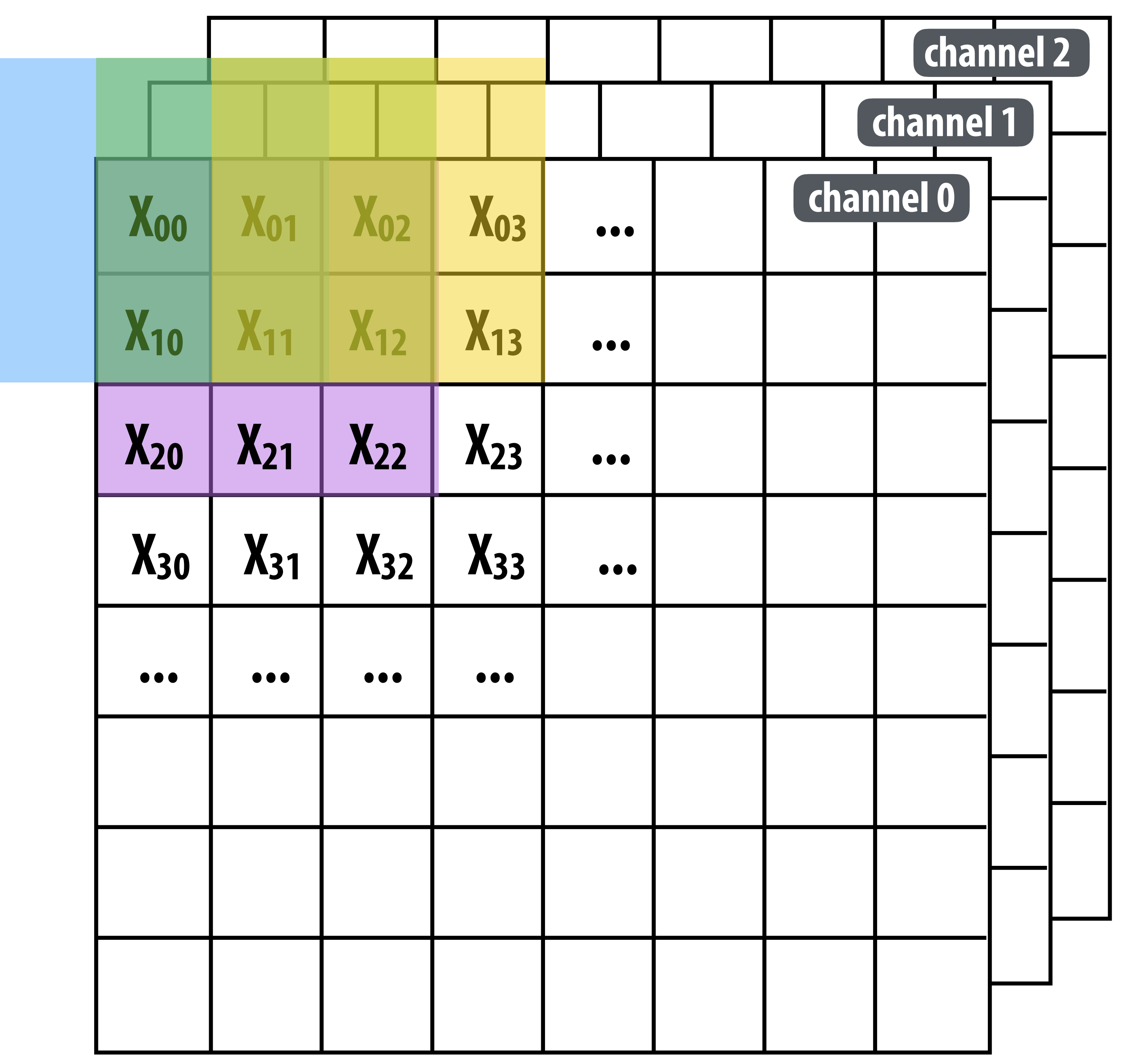
Note: 0-pad matrix

Multiple convolutions as matrix-matrix mult



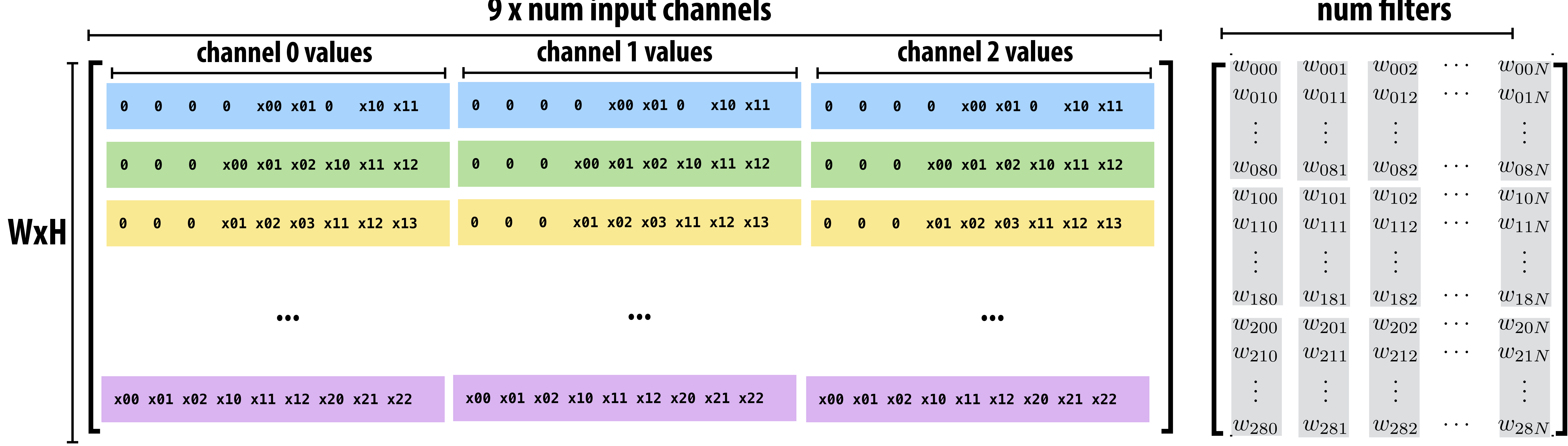


Multiple convolutions on multiple input channels



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)

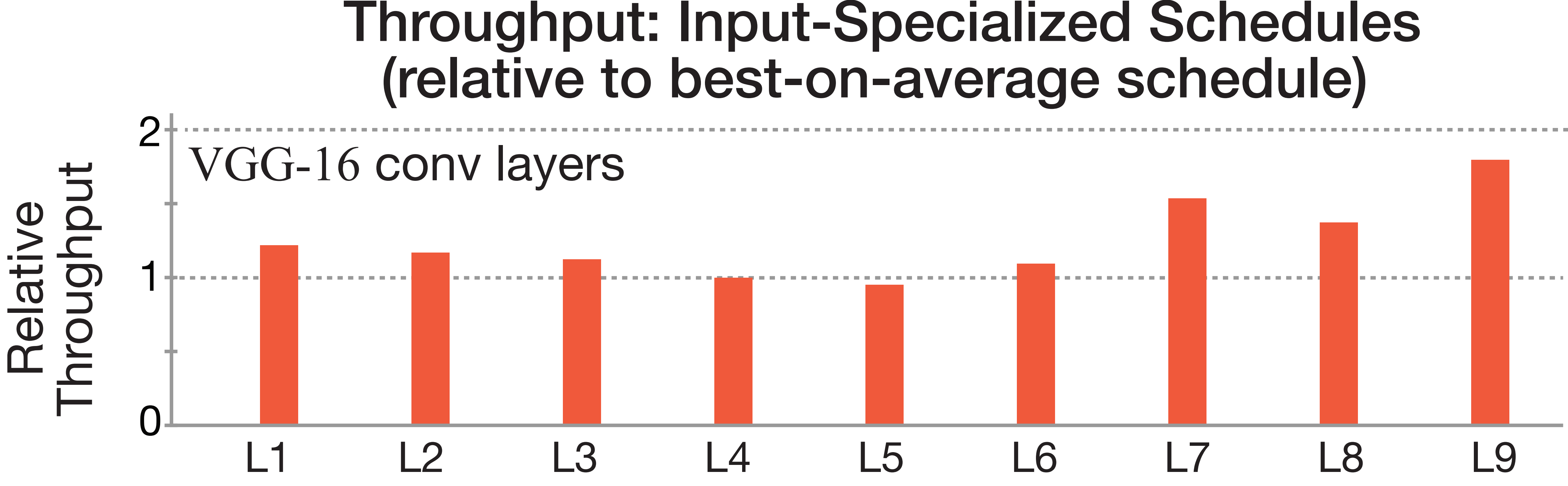
Avoids O(N) footprint increase by avoiding input matrix materialization
But must roll your own highly optimized implementation of complicated loop nest.

Convolutional layer in Halide

```
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



[Figure credit: Mullapudi et al. 2016]

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	$14 \times 14 \times 512$
5× 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	$1 \times 1 \times 1000$

Optimization of Manually Authored Schedules

LENSBLUR

MAXFILTER

NL MEANS

Throughput (1/ms)

Schedule development time (minutes)

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

Libraries offering high-performance implementations of key DNN layers

TensorFlow NN ops

tensorflow::ops::AvgPool	Performs average pooling on the input.
tensorflow::ops::AvgPool3D	Performs 3D average pooling on the input.
tensorflow::ops::AvgPool3DGrad	Computes gradients of average pooling function.
tensorflow::ops::BiasAdd	Adds `bias` to `value`.
tensorflow::ops::BiasAddGrad	The backward operation for "BiasAdd" on the "bias" te
tensorflow::ops::Conv2D	Computes a 2-D convolution given 4-D `input` and `fi`
tensorflow::ops::Conv2DBackpropFilter	Computes the gradients of convolution with respect t
tensorflow::ops::Conv2DBackpropInput	Computes the gradients of convolution with respect t
tensorflow::ops::Conv3D	Computes a 3-D convolution given 5-D `input` and `fi`
tensorflow::ops::Conv3DBackpropFilterV2	Computes the gradients of 3-D convolution with respe
tensorflow::ops::Conv3DBackpropInputV2	Computes the gradients of 3-D convolution with respe
tensorflow::ops::DataFormatDimMap	Returns the dimension index in the destination data f
tensorflow::ops::DataFormatVecPermute	Permute input tensor from `src_format` to `dst_for`
tensorflow::ops::DepthwiseConv2dNative	Computes a 2-D depthwise convolution given 4-D `in` tensors.
tensorflow::ops::DepthwiseConv2dNativeBackpropFilter	Computes the gradients of depthwise convolution wit
tensorflow::ops::DepthwiseConv2dNativeBackpropInput	Computes the gradients of depthwise convolution wit
tensorflow::ops::Dilation2D	Computes the grayscale dilation of 4-D `input` and 3-
tensorflow::ops::Dilation2DBackpropFilter	Computes the gradient of morphological 2-D dilation filter.
tensorflow::ops::Dilation2DBackpropInput	Computes the gradient of morphological 2-D dilation input.
tensorflow::ops::Elu	Computes exponential linear: `exp(features) - 1` otherwise.
tensorflow::ops::FractionalAvgPool	Performs fractional average pooling on the input.
tensorflow::ops::FractionalMaxPool	Performs fractional max pooling on the input.
tensorflow::ops::FusedBatchNorm	Batch normalization.

tensorflow::ops::FusedBatchNormGrad	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormGradV2	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormGradV3	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormV2	Batch normalization.
tensorflow::ops::FusedBatchNormV3	Batch normalization.
tensorflow::ops::FusedPadConv2D	Performs a padding as a preprocess during a convolution.
tensorflow::ops::FusedResizeAndPadConv2D	Performs a resize and padding as a preprocess during a convolution.
tensorflow::ops::InTopK	Says whether the targets are in the top K predictions.
tensorflow::ops::InTopKV2	Says whether the targets are in the top K predictions.
tensorflow::ops::L2Loss	L2 Loss.
tensorflow::ops::LRN	Local Response Normalization.
tensorflow::ops::LogSoftmax	Computes log softmax activations.
tensorflow::ops::MaxPool	Performs max pooling on the input.
tensorflow::ops::MaxPool3D	Performs 3D max pooling on the input.
tensorflow::ops::MaxPool3DGrad	Computes gradients of 3D max pooling function.
tensorflow::ops::MaxPool3DGradGrad	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGrad	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGradV2	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGradWithArgmax	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradV2	Computes gradients of the maxpooling function.
tensorflow::ops::MaxPoolV2	Performs max pooling on the input.
tensorflow::ops::MaxPoolWithArgmax	Performs max pooling on the input and outputs both max values and indices.
tensorflow::ops::NthElement	Finds values of the n-th order statistic for the last dimension.
tensorflow::ops::QuantizedAvgPool	Produces the average pool of the input tensor for quantized types.
tensorflow::ops:: QuantizedBatchNormWithGlobalNormalization	Quantized Batch normalization.
tensorflow::ops::QuantizedBiasAdd	Adds Tensor 'bias' to Tensor 'input' for Quantized types.
tensorflow::ops::QuantizedConv2D	Computes a 2D convolution given quantized 4D input and filter tensors.
tensorflow::ops::QuantizedMaxPool	Produces the max pool of the input tensor for quantized types.

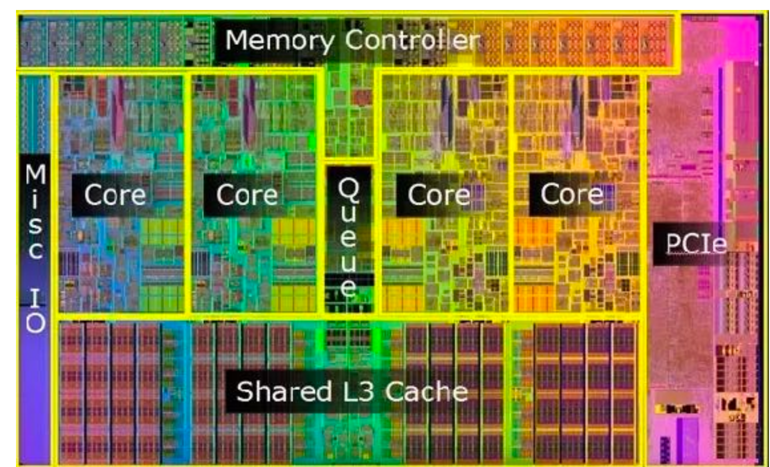
Libraries offering high-performance implementations of key DNN layers

NN ops

tensorflow::ops::AvgPool	Performs average pooling on the input.
tensorflow::ops::AvgPool3D	Performs 3D average pooling on the input.
tensorflow::ops::AvgPool3DGrad	Computes gradients of average pooling function.
tensorflow::ops::BiasAdd	Adds `bias` to `value`.
tensorflow::ops::BiasAddGrad	The backward operation for "BiasAdd" on the "bias" te
tensorflow::ops::Conv2D	Computes a 2-D convolution given 4-D `input` and `fi`
tensorflow::ops::Conv2DBackpropFilter	Computes the gradients of convolution with respect t
tensorflow::ops::Conv2DBackpropInput	Computes the gradients of convolution with respect t
tensorflow::ops::Conv3D	Computes a 3-D convolution given 5-D `input` and `fi`
tensorflow::ops::Conv3DBackpropFilterV2	Computes the gradients of 3-D convolution with resp
tensorflow::ops::Conv3DBackpropInputV2	Computes the gradients of 3-D convolution with resp
tensorflow::ops::DataFormatDimMap	Returns the dimension index in the destination data f
tensorflow::ops::DataFormatVecPermute	Permute input tensor from `src_format` to `dst_for`
tensorflow::ops::DepthwiseConv2dNative	Computes a 2-D depthwise convolution given 4-D `in` tensors.
tensorflow::ops::DepthwiseConv2dNativeBackpropFilter	Computes the gradients of depthwise convolution wi
tensorflow::ops::DepthwiseConv2dNativeBackpropInput	Computes the gradients of depthwise convolution wi
tensorflow::ops::Dilation2D	Computes the grayscale dilation of 4-D `input` and 3-
tensorflow::ops::Dilation2DBackpropFilter	Computes the gradient of morphological 2-D dilation filter.
tensorflow::ops::Dilation2DBackpropInput	Computes the gradient of morphological 2-D dilation input.
tensorflow::ops::Elu	Computes exponential linear: `exp(features) - 1` otherwise.
tensorflow::ops::FractionalAvgPool	Performs fractional average pooling on the input.
tensorflow::ops::FractionalMaxPool	Performs fractional max pooling on the input.
tensorflow::ops::FusedBatchNorm	Batch normalization.

NVIDIA cuDNN

Intel® oneAPI Deep Neural Network Library



Example: CUDNN convolution

```
cudnnStatus_t cudnnConvolutionForward(
    cudnnHandle_t                       handle,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *x,
    const cudnnFilterDescriptor_t       wDesc,
    const void                         *w,
    const cudnnConvolutionDescriptor_t  convDesc,
    cudnnConvolutionFwdAlgo_t           algo,
    void                               *workSpace,
    size_t                              workSpaceSizeInBytes,
    const void                         *beta,
    const cudnnTensorDescriptor_t       yDesc,
    void                               *y)
```

Possible algorithms:

`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`
This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`
This algorithm expresses convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_GEMM`
This algorithm expresses the convolution as an explicit matrix product. A significant memory workspace is needed to store the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_DIRECT`
This algorithm expresses the convolution as a direct convolution (for example, without implicitly or explicitly doing a matrix multiplication).

`CUDNN_CONVOLUTION_FWD_ALGO_FFT`
This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results.

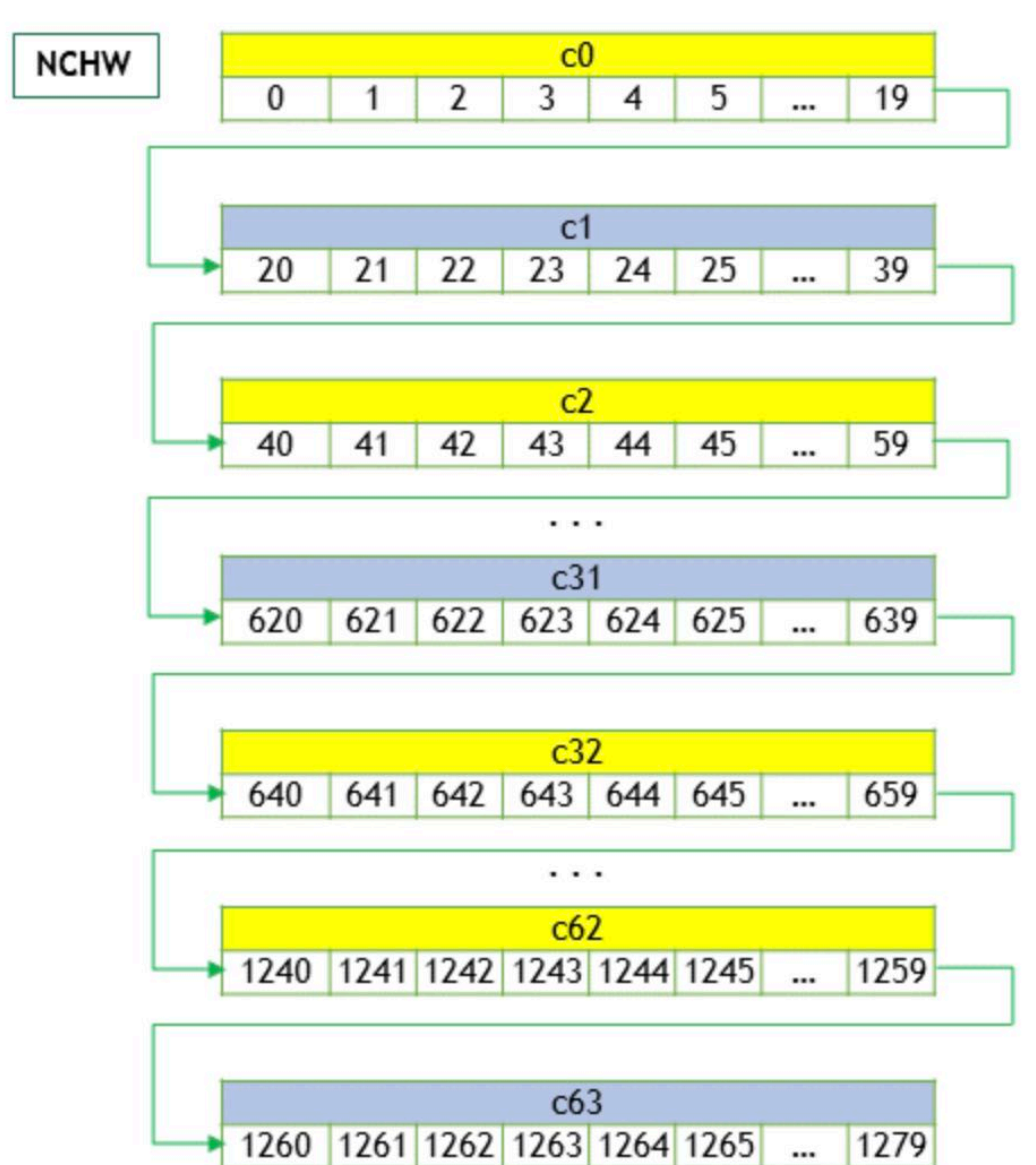
`CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING`
This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than `CUDNN_CONVOLUTION_FWD_ALGO_FFT` for large size images.

`CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD`
This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.

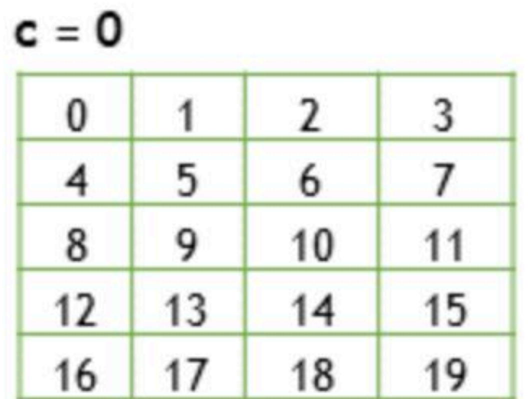
`CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED`
This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.

NCHW data layout

- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., number of channels); 64.
- **H** is the image height; 5.
- **W** is the image width; 4.



c = 0

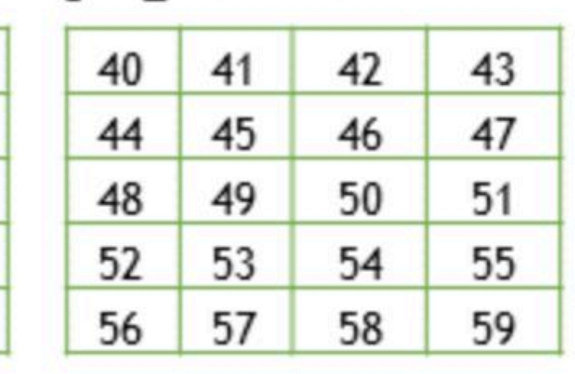


0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

c = 1

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

c = 2



40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

. . .

. . .

c = 62

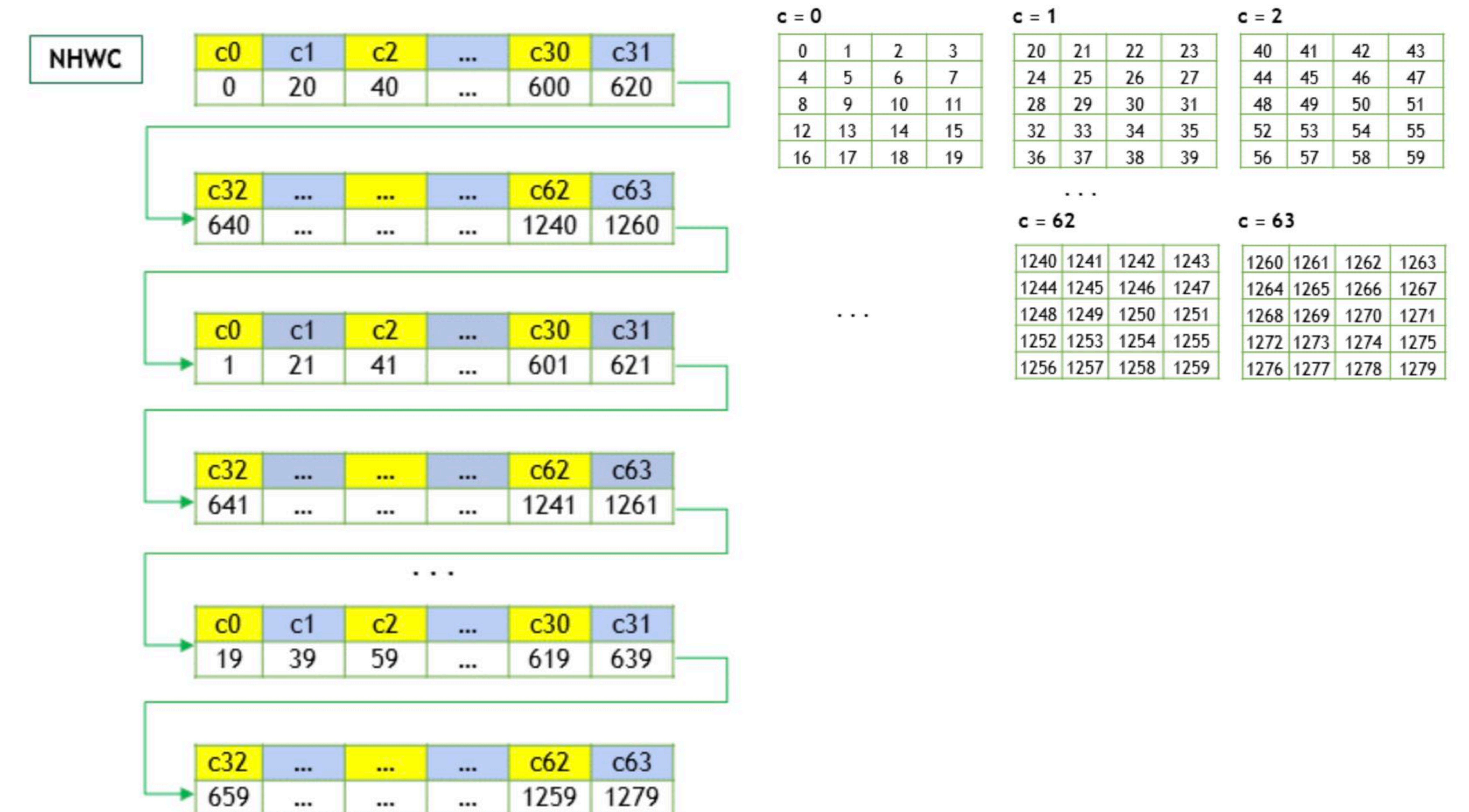
1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

c = 63

1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

NHWC data layout

- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., number of channels); 64.
- **H** is the image height; 5.
- **W** is the image width; 4.



Memory traffic between operations

- Consider this sequence:

- Imagine the bandwidth cost of dumping 1 GB of conv outputs to memory, and reading it back in between each op!
- But note that per-element [scale+bias] operation can easily be performed per-element right after each element is computed by conv!
- And max pool's output can be computed once every 2x2 region of output is computed.

Fusing operations with 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++) {
            float tmp = 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)
                     tmp += layer_weights[f][jj][ii][kk] * input[img][j+jj][i+ii][kk];
            output[img][j][i][f] = tmp*scale + bias;
         }
```

Exercise to class 1:
Is there a way to eliminate the scale/bias operation completely?

Exercise to class 2:
How would you also "fuse" in the max pool?

Old style: hardcoded "fused" ops

```
cudnnStatus_t cudnnConvolutionBiasActivationForward(
    cudnnHandle_t                      handle,
    const void                        *alpha1,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const cudnnFilterDescriptor_t      wDesc,
    const void                        *w,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionFwdAlgo_t          algo,
    void                              *workSpace,
    size_t                             workSpaceSizeInBytes,
    const void                        *alpha2,
    const cudnnTensorDescriptor_t      zDesc,
    const void                        *z,
    const cudnnTensorDescriptor_t      biasDesc,
    const void                        *bias,
    const cudnnActivationDescriptor_t  activationDesc,
    const cudnnTensorDescriptor_t      yDesc,
    void                              *y)
```

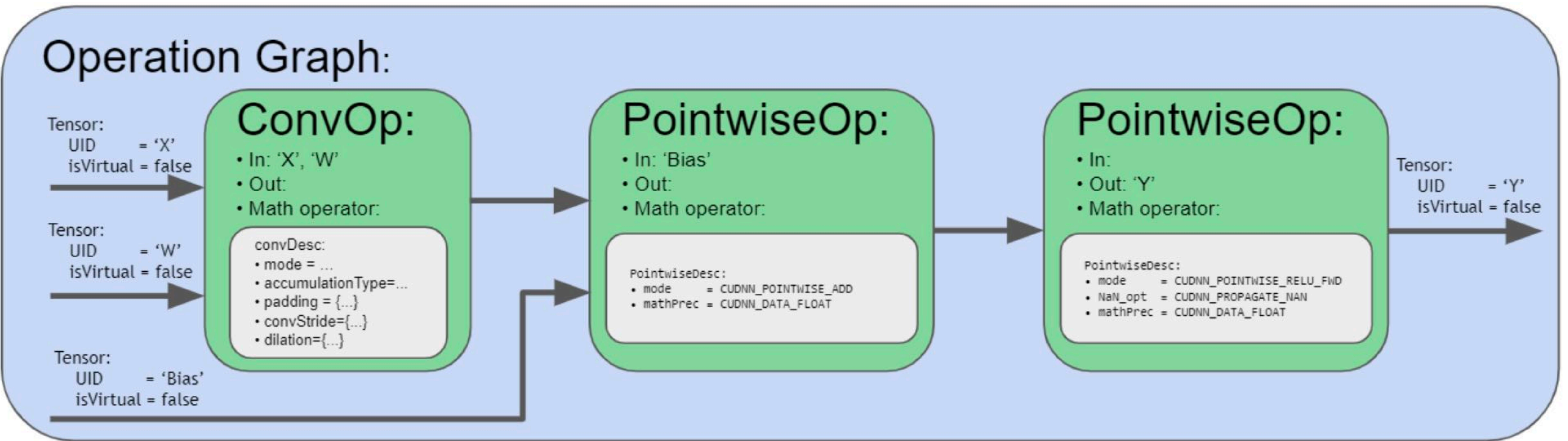
This function applies a bias and then an activation to the convolutions or cross-correlations of cudnnConvolutionForward(), returning results in `y`. The full computation follows the equation `y = act ( alpha1 * conv(x) + alpha2 * z + bias )`.

Tensorflow:

tensorflow::ops::FusedBatchNorm	Batch normalization.

tensorflow::ops::FusedResizeAndPadConv2D	Performs a resize and padding as a preprocess during a convolution.

Fusion example: CUDNN “backend"

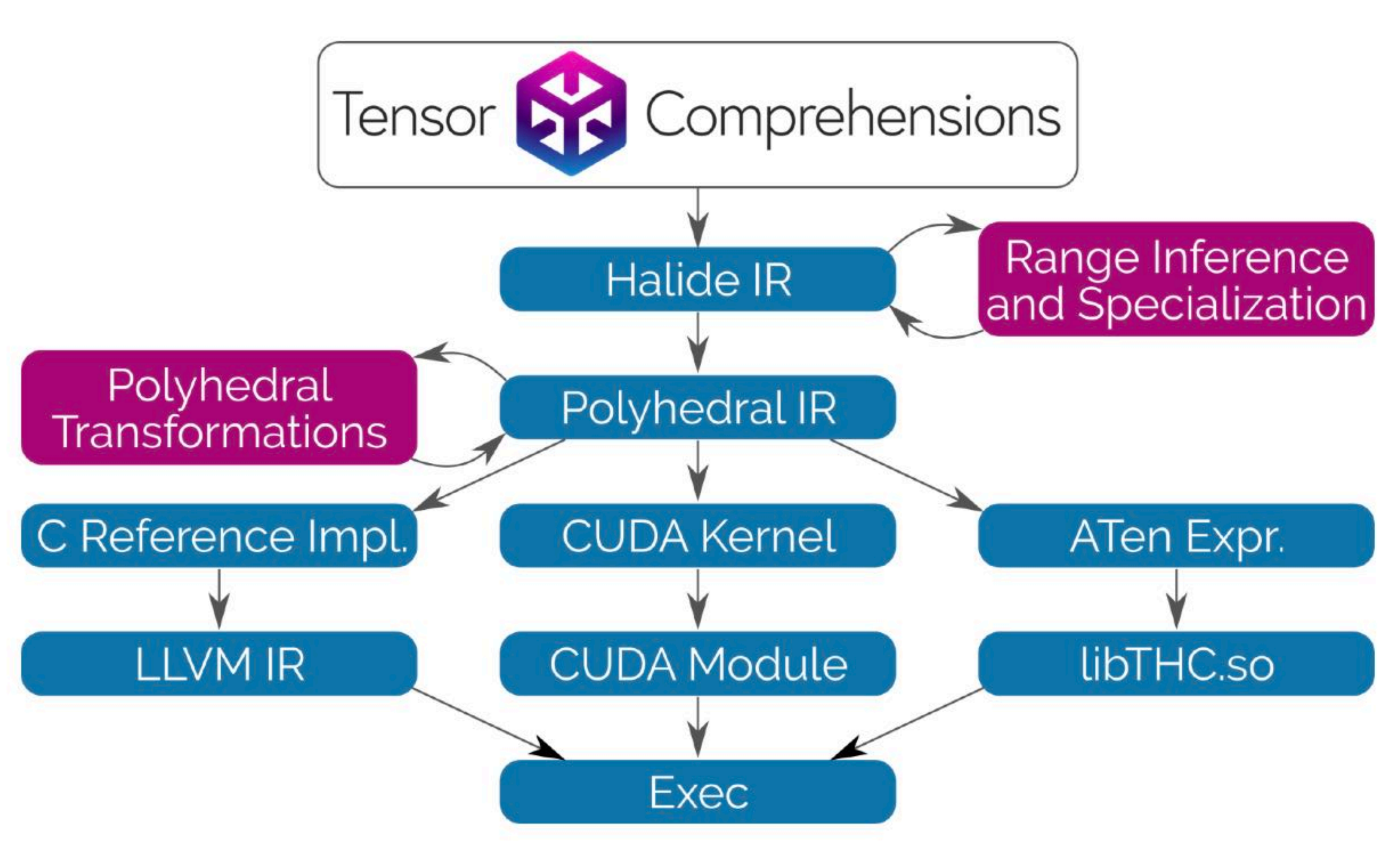


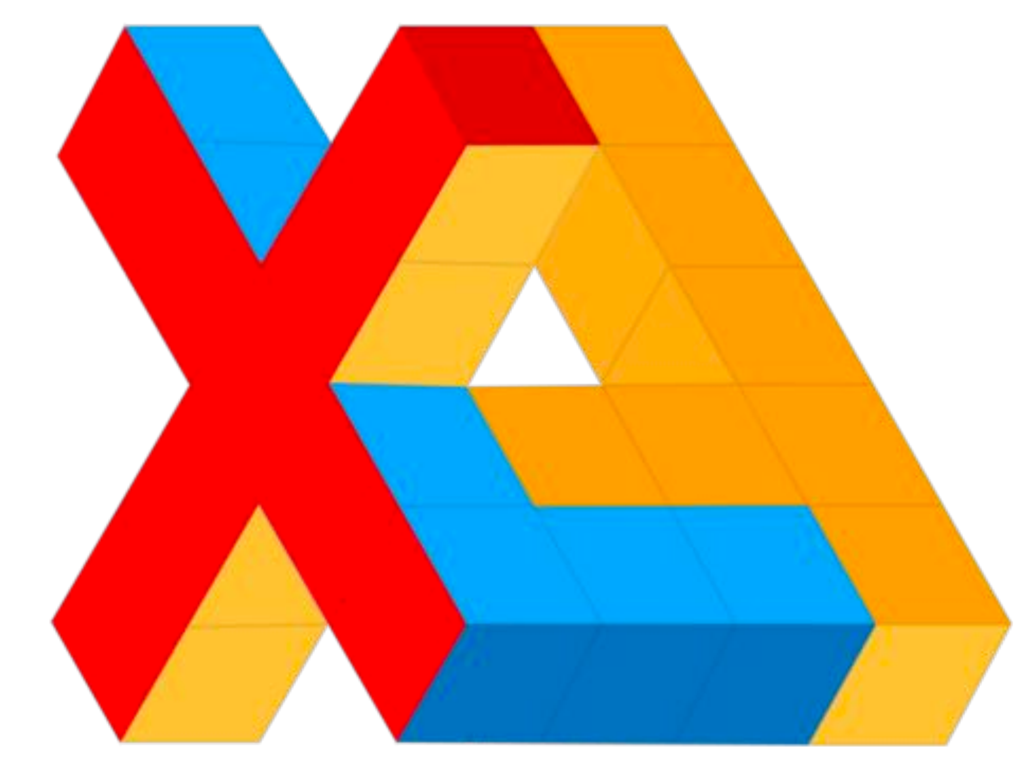
Note for operation fusion use cases, there are two different mechanisms in cuDNN to support them. First, there are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If there is a match, then that particular engine is deemed suitable for this use case. In addition, there are also runtime fusion engines to be made available in the upcoming releases. Instead of passively matching the user graph, such engines actively walk the graph and assemble code blocks to form a CUDA kernel and compile on the fly. Such runtime fusion engines are much more flexible in its range of support. However, because the construction of the execution plans requires runtime compilation, the one-time CPU overhead is higher than the other engines.

Compiler generate new implementations that “fuse” multiple operations into a single node that executes efficiently (without runtime overhead or communicating intermediate results through memory)

Note: this is Halide “compute at”

Many efforts to automatically schedule key DNN operations



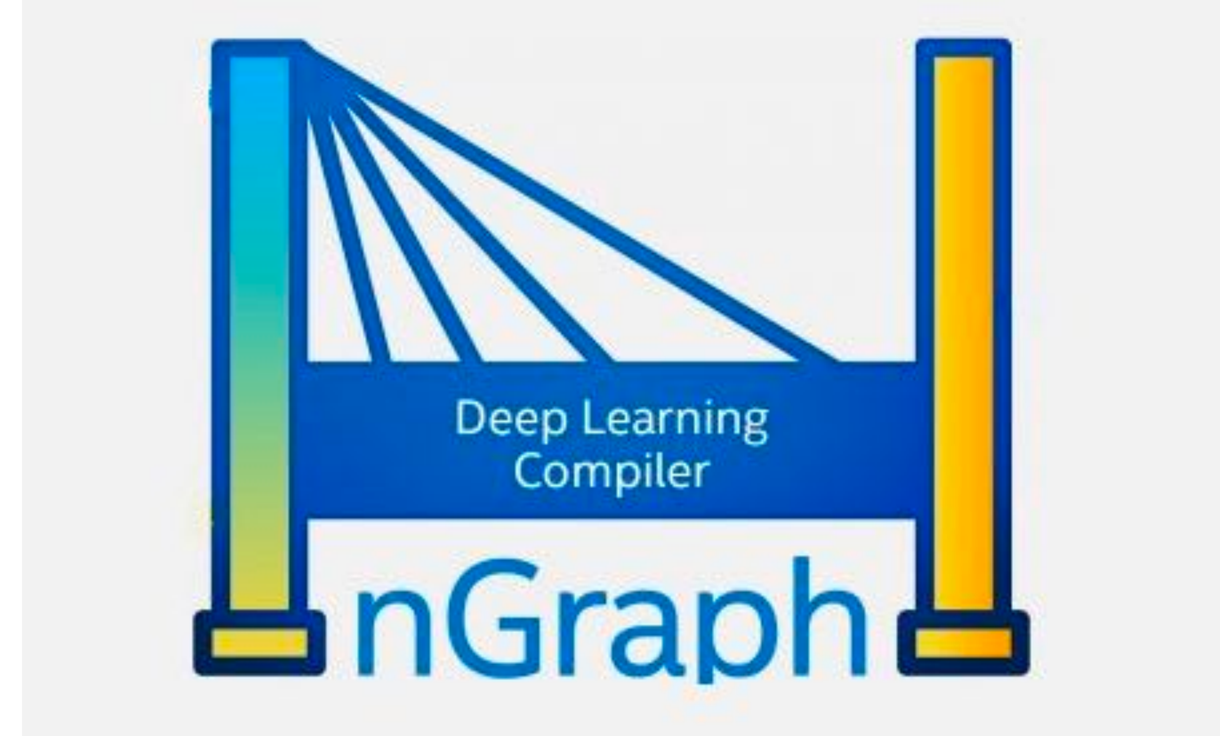


tvm Open Deep Learning Compiler Stack

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

Use of low precision values

- Many efforts to use low precision values for DNN weights and intermediate activations
- Eight and 16 bit values are common
- 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× 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× faster convolutional operations (in terms of number of the high precision operations) and 32× 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.

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**

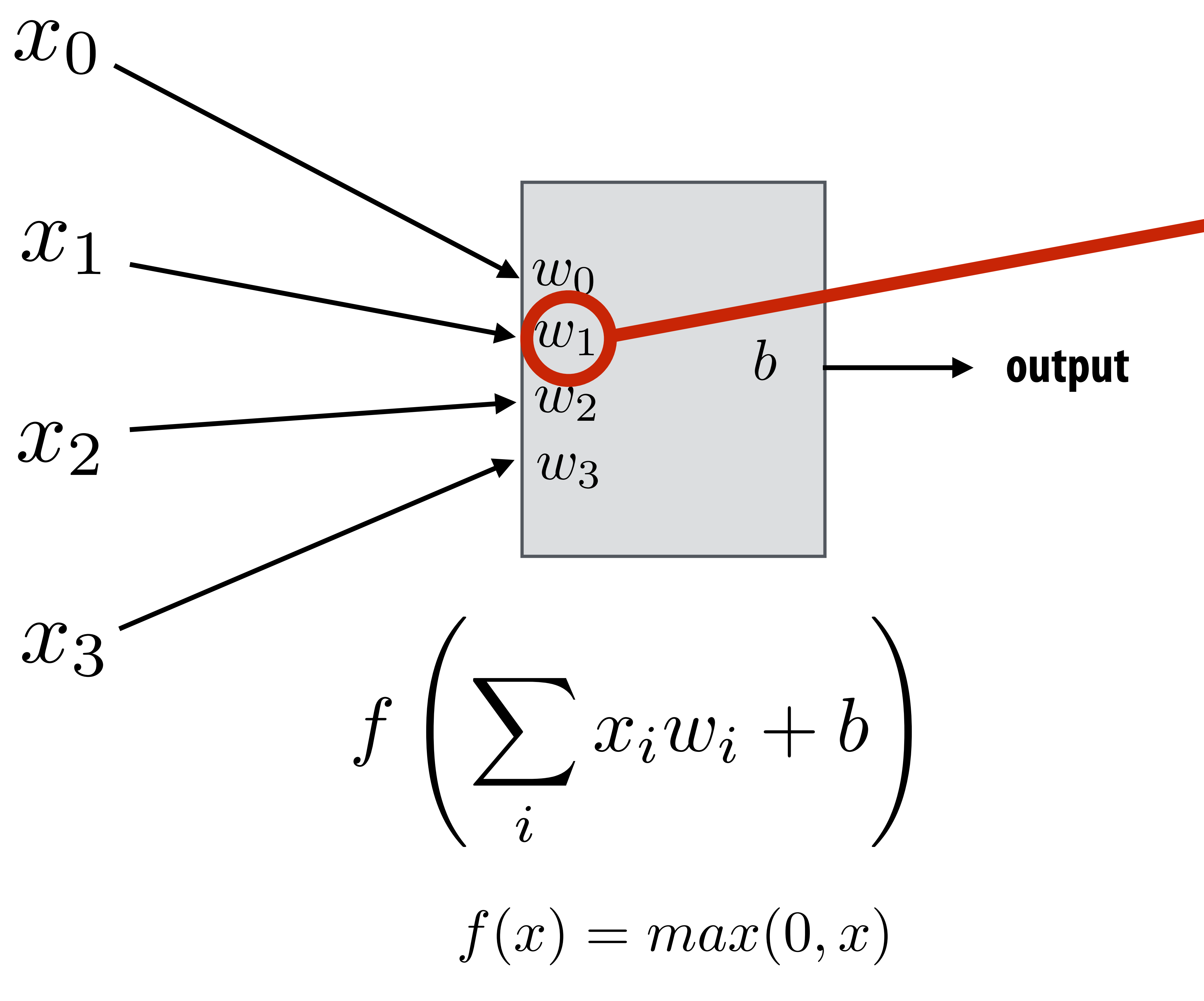
Relative Energy Co

1 10 100 10

Estimates for 45nm process
[Source: Mark Horowitz]

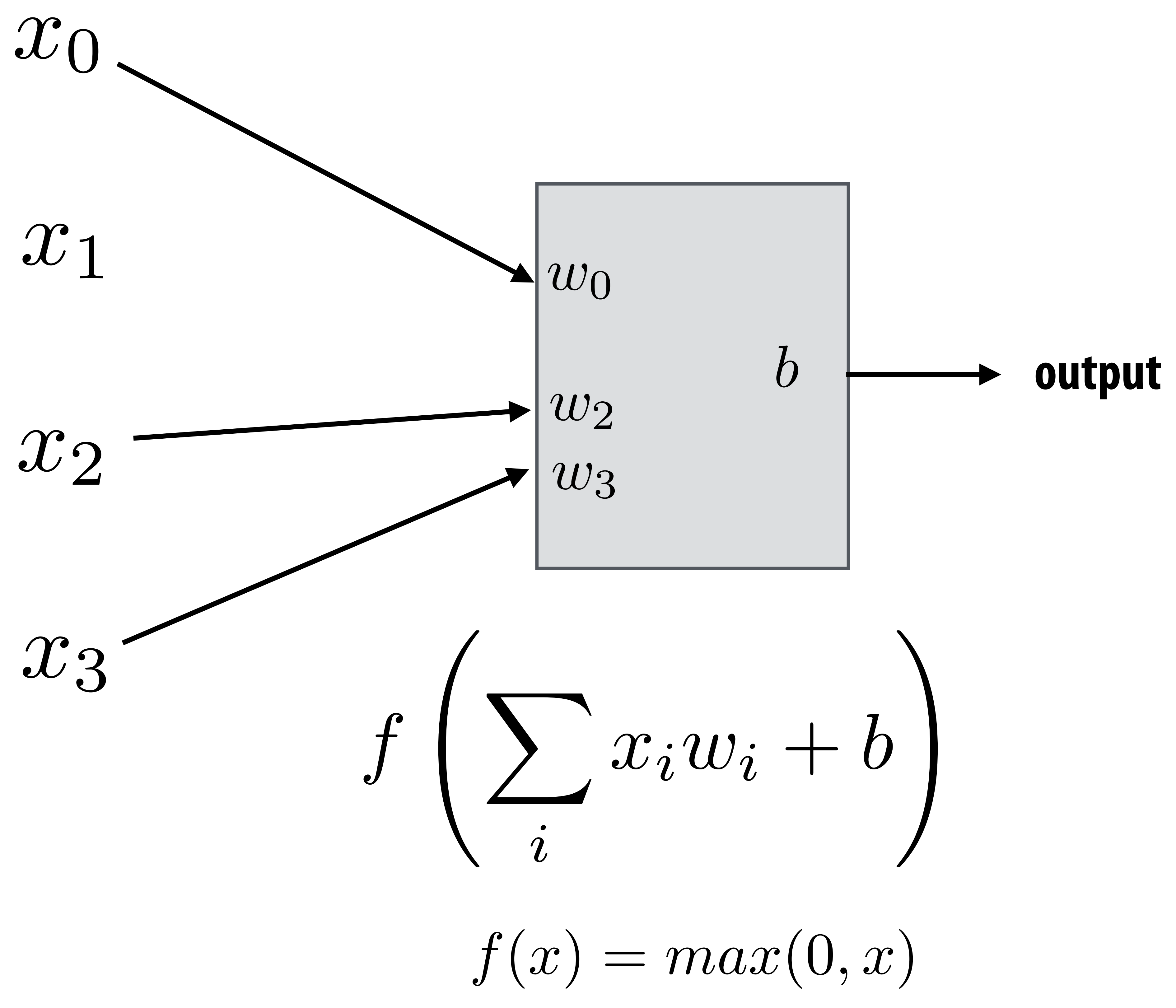
Is there an opportunity for compression?

"Pruning" (sparsifying) a network



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

“Pruning” (sparsifying) a network



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**

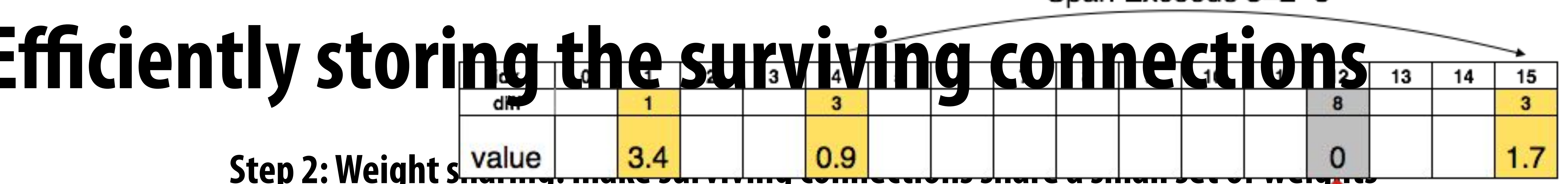
```
Indices    1    4   9  ...
Value     1.8  0.5  2.1
```

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

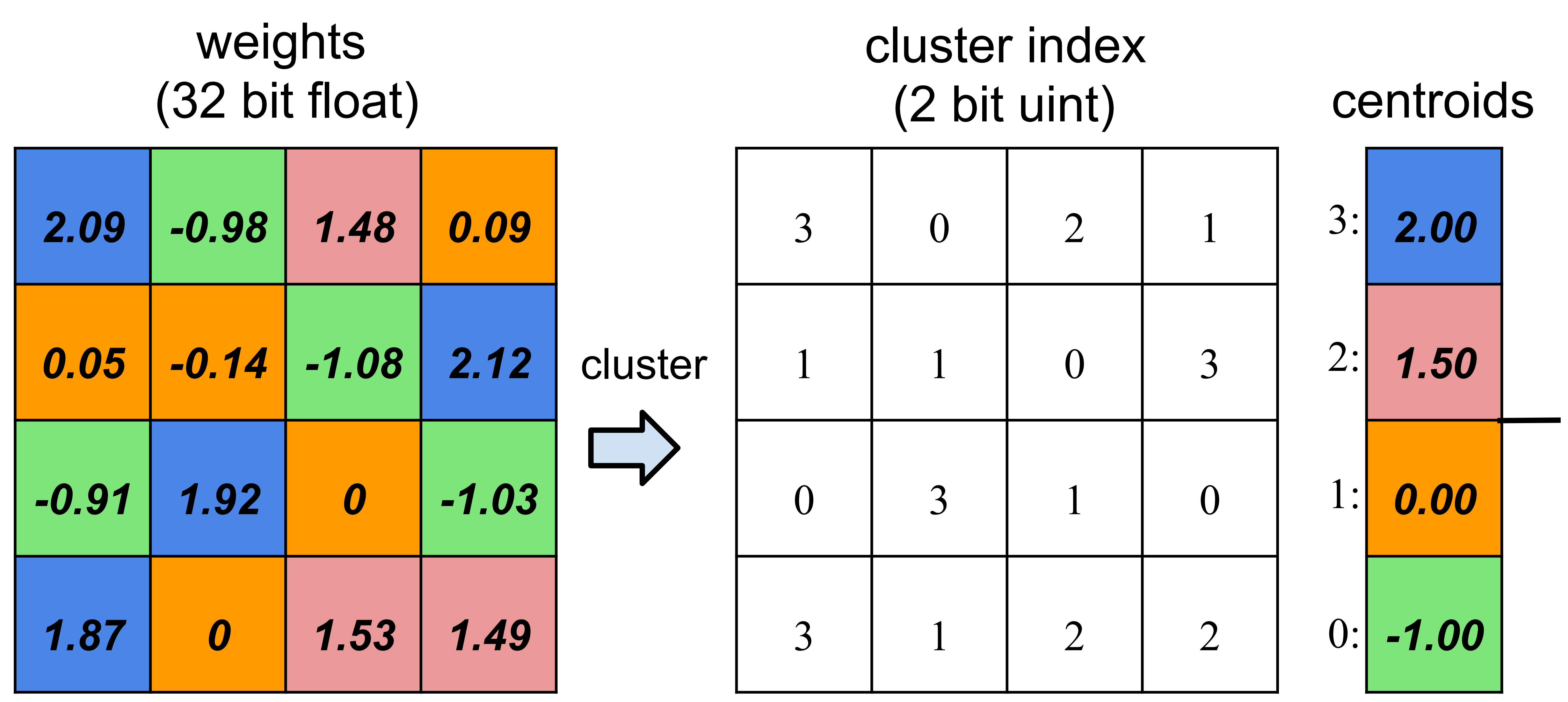
```
Indices   1    3    5  ...
Value     1.8  0.5  2.1
```

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**



Step 3: Huffman encode quantized weights and CSR indices (lossless compression)

[Figure credit: Han ICLR16]

VGG-16 sparsification

[Han ICLR16]

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**×

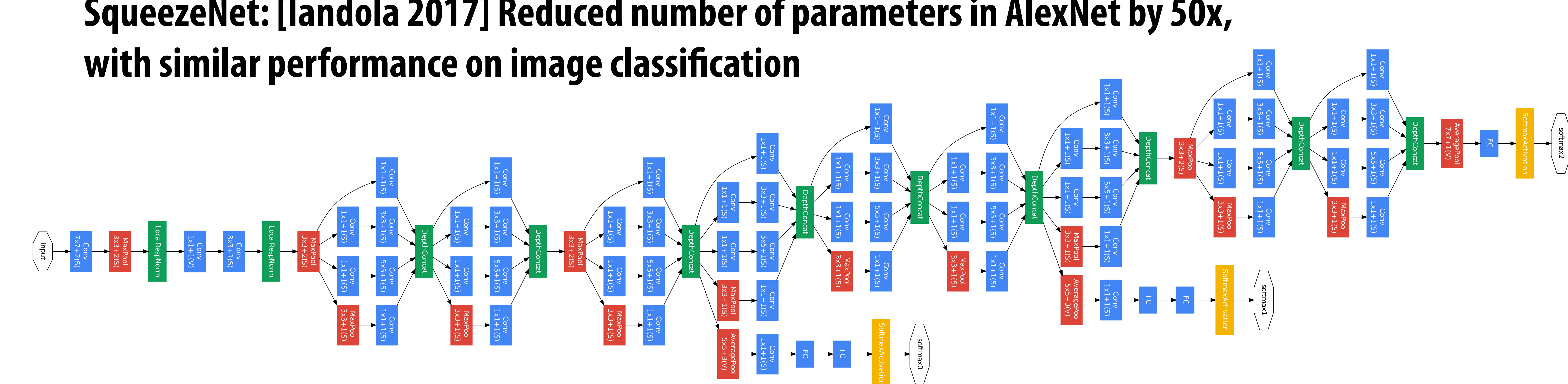
*** Benefits of automatic pruning apply mainly to fully connected layers, but unfortunately many modern networks are dominated by costs of convolutional layers**

DNN optimization is a great example of non-domain-specific vs. domain-specific approach to innovation

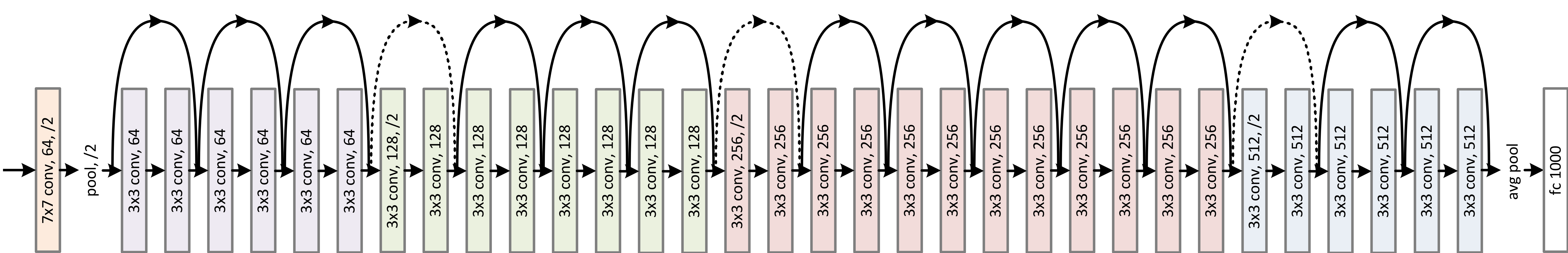
Leveraging domain-knowledge: more efficient topologies (aka better algorithm design)

- 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

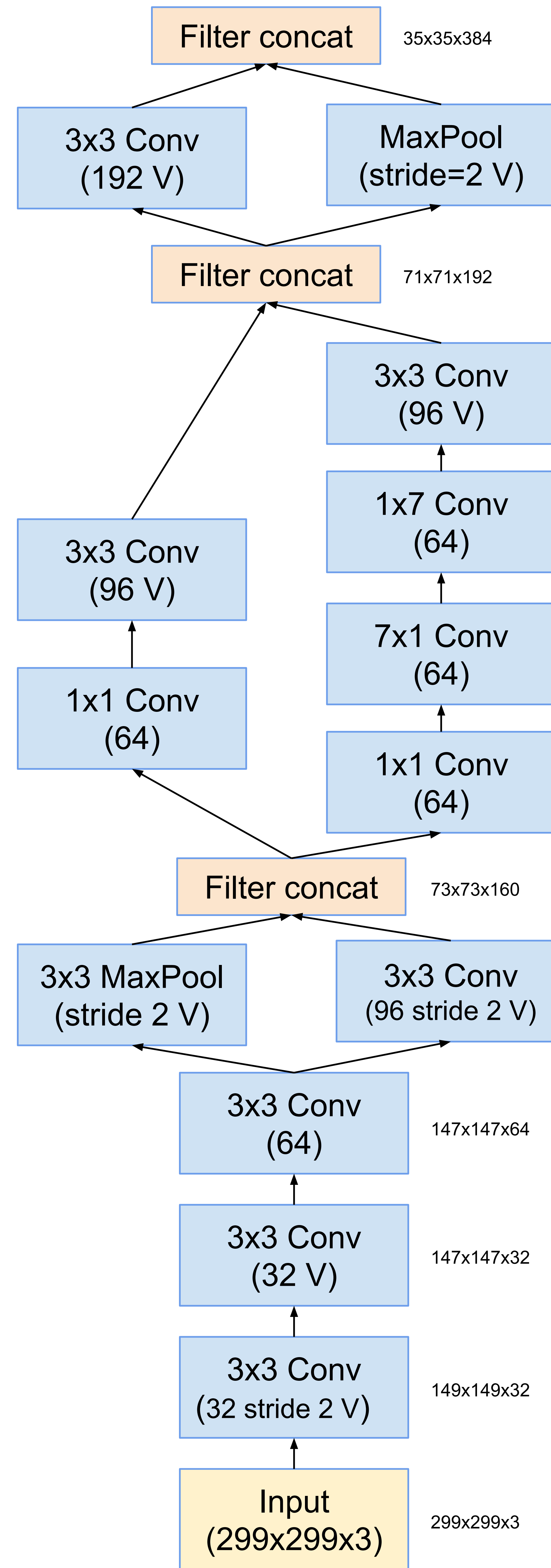


Inception v1 (GoogleLeNet) — 27 total layers, 7M parameters

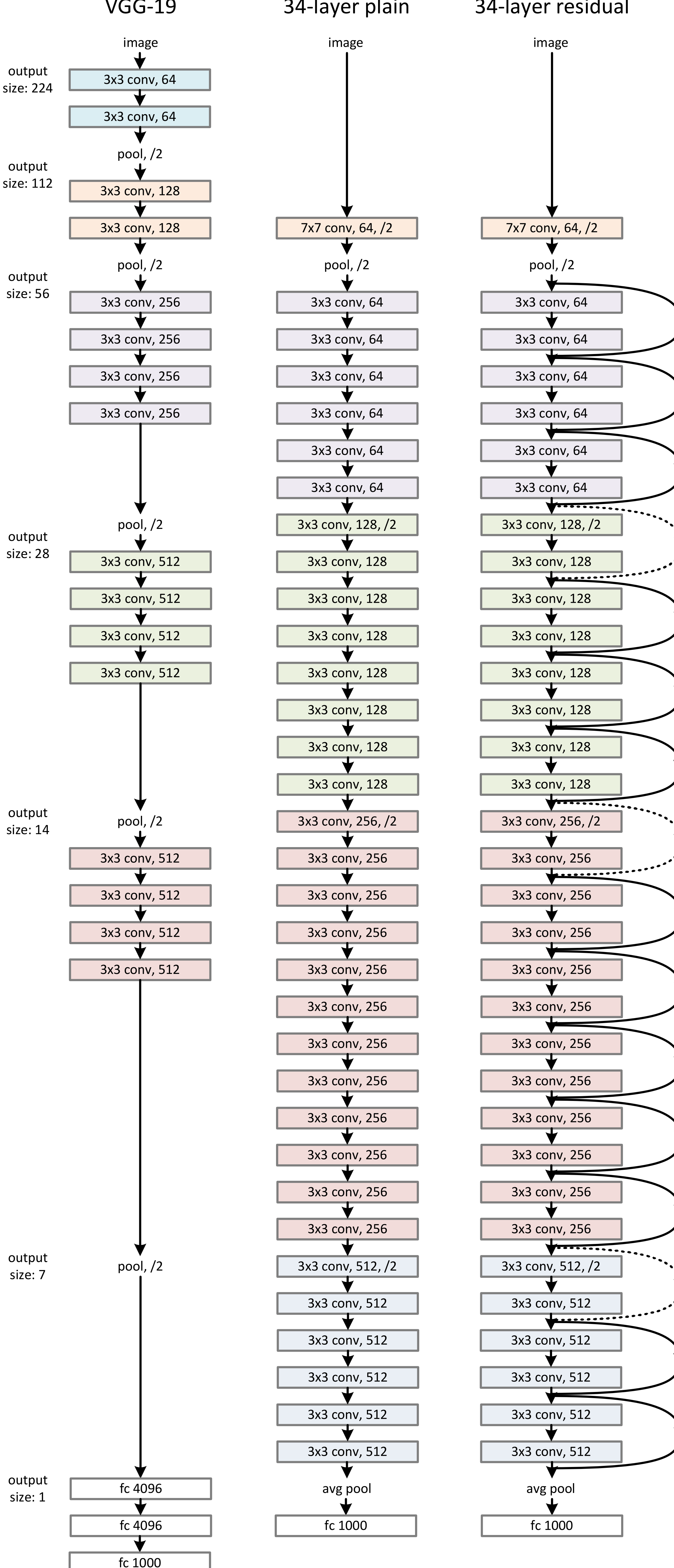


ResNet (34 layer version)

Inception stem



ResNet



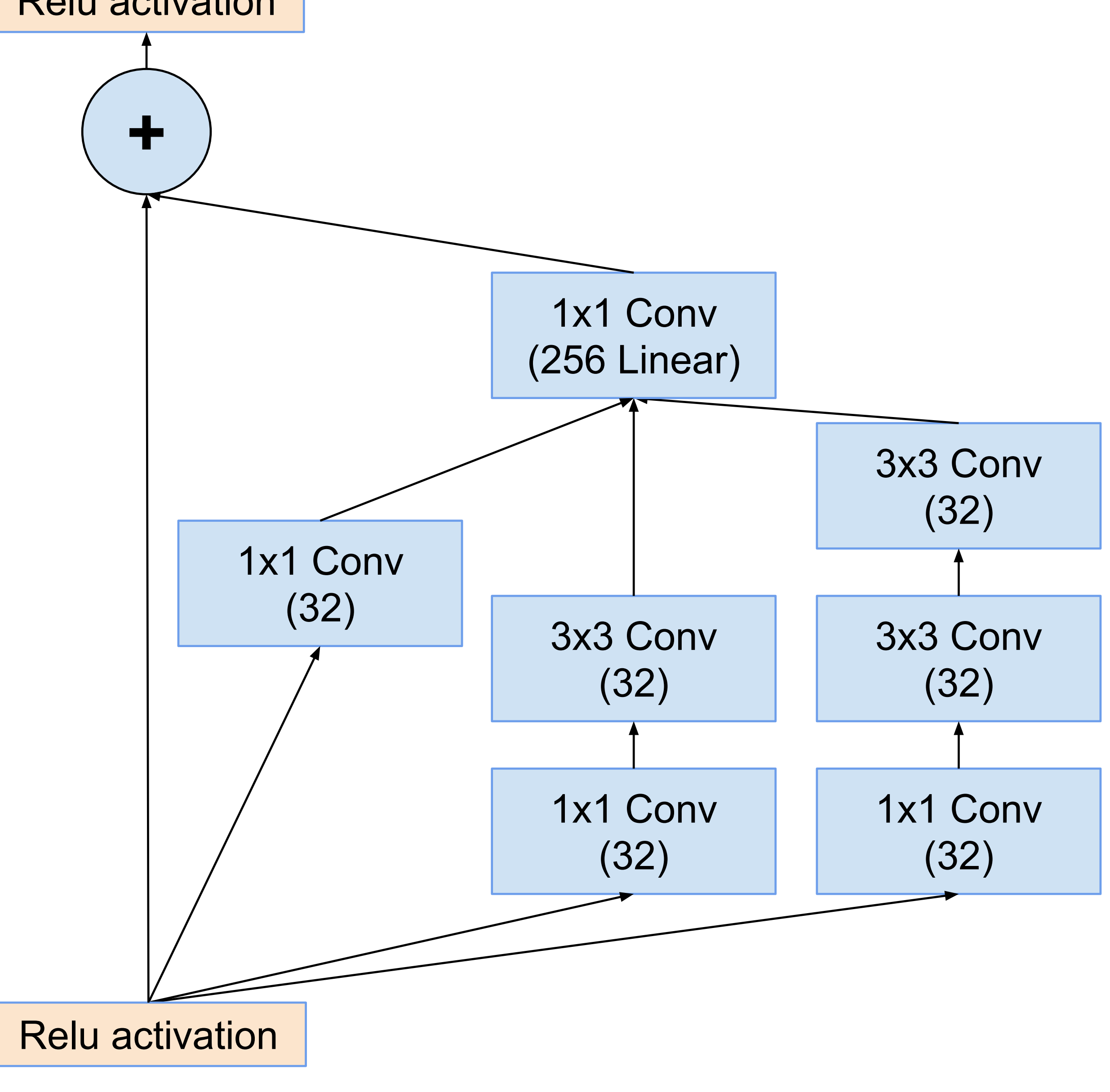


Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

Effect of topology innovation

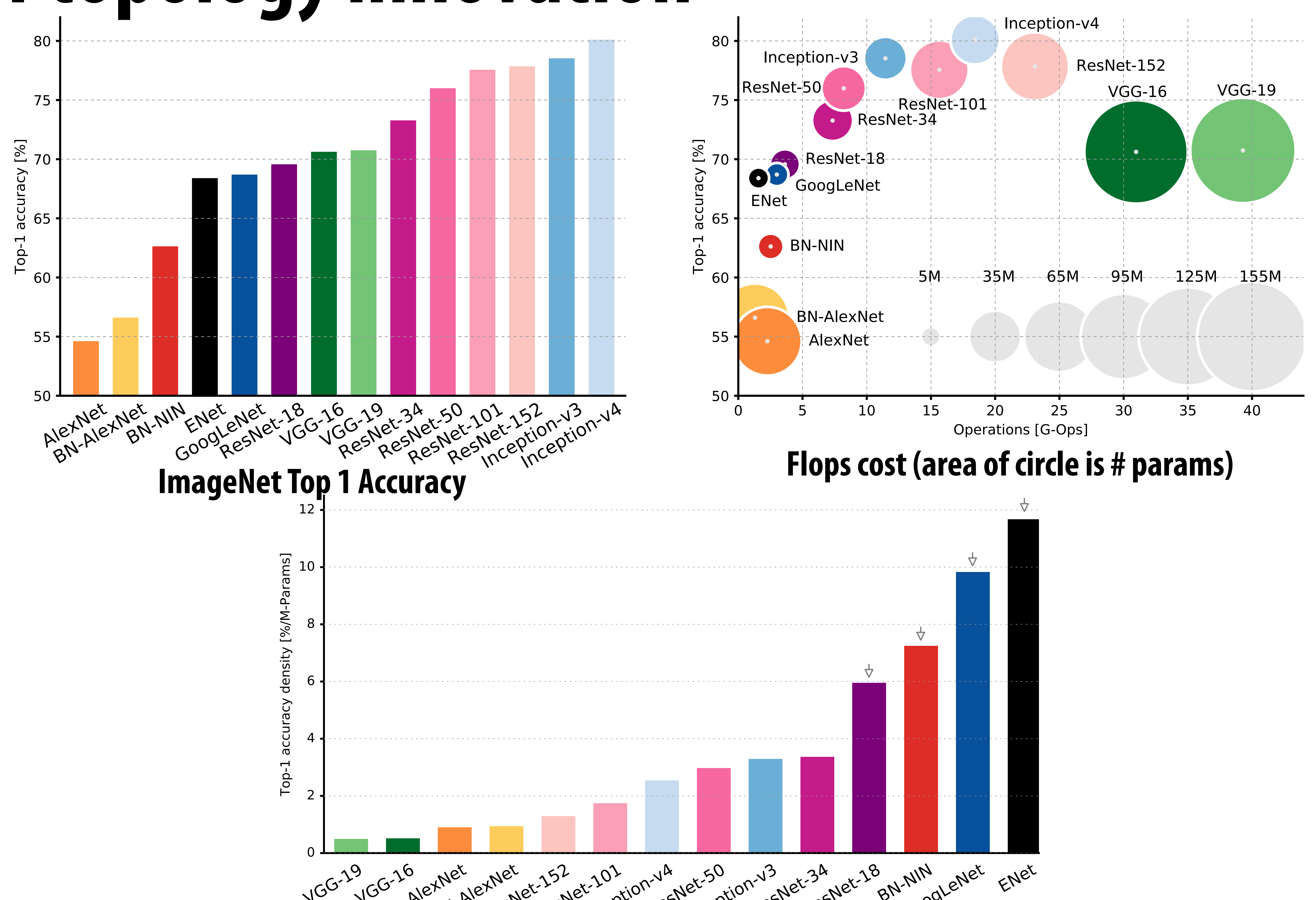


Figure credit: Canziani et al 2017

Improving accuracy/cost (image classification)

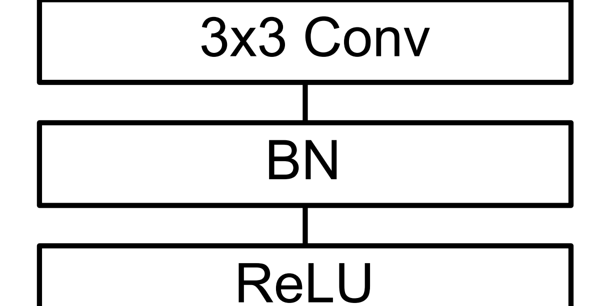
2014 → 2017 ~ 25x improvement in cost at similar accuracy

	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)

MobileNet

[Howard et al. 2017]



3x3 Depthwise Conv
BN
ReLU
1x1 Conv
BN
ReLU

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**

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	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ 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

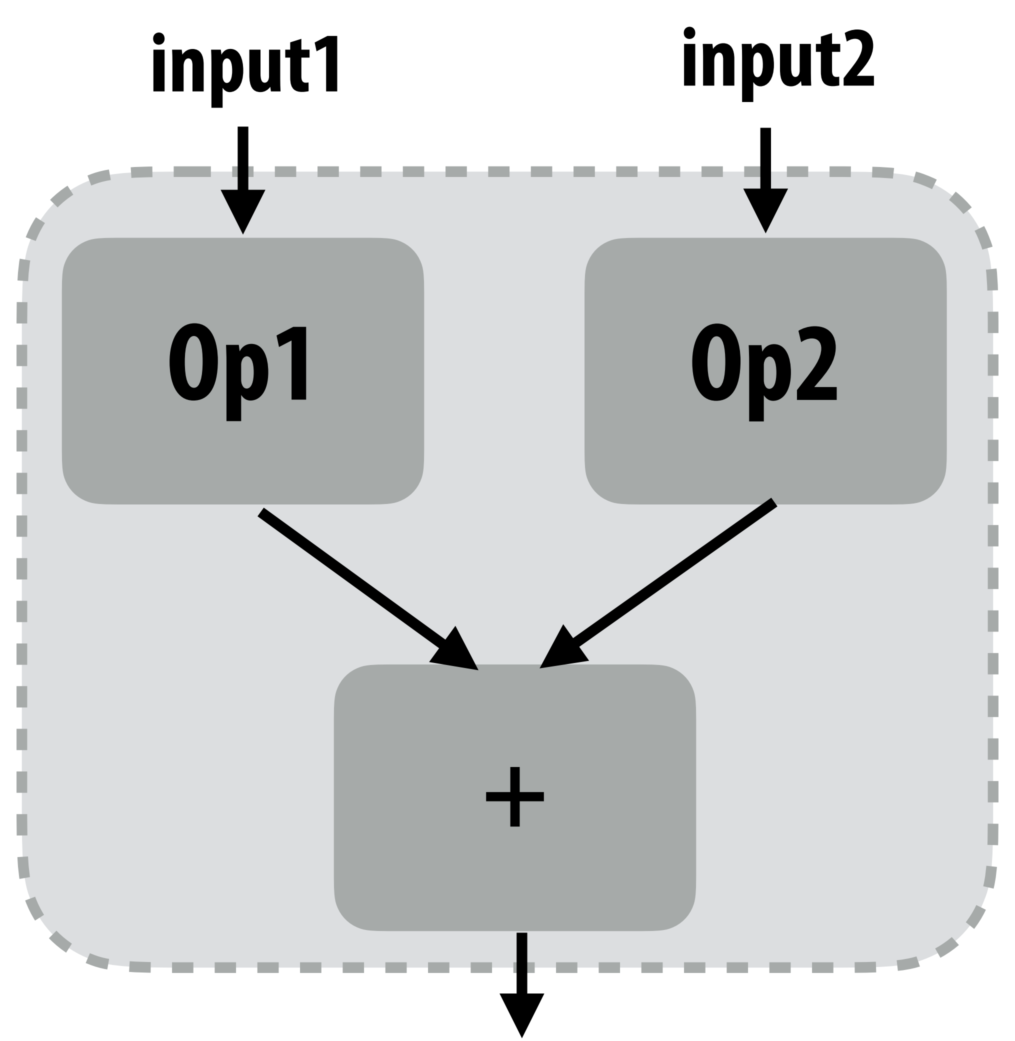
Model optimization techniques

- 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)



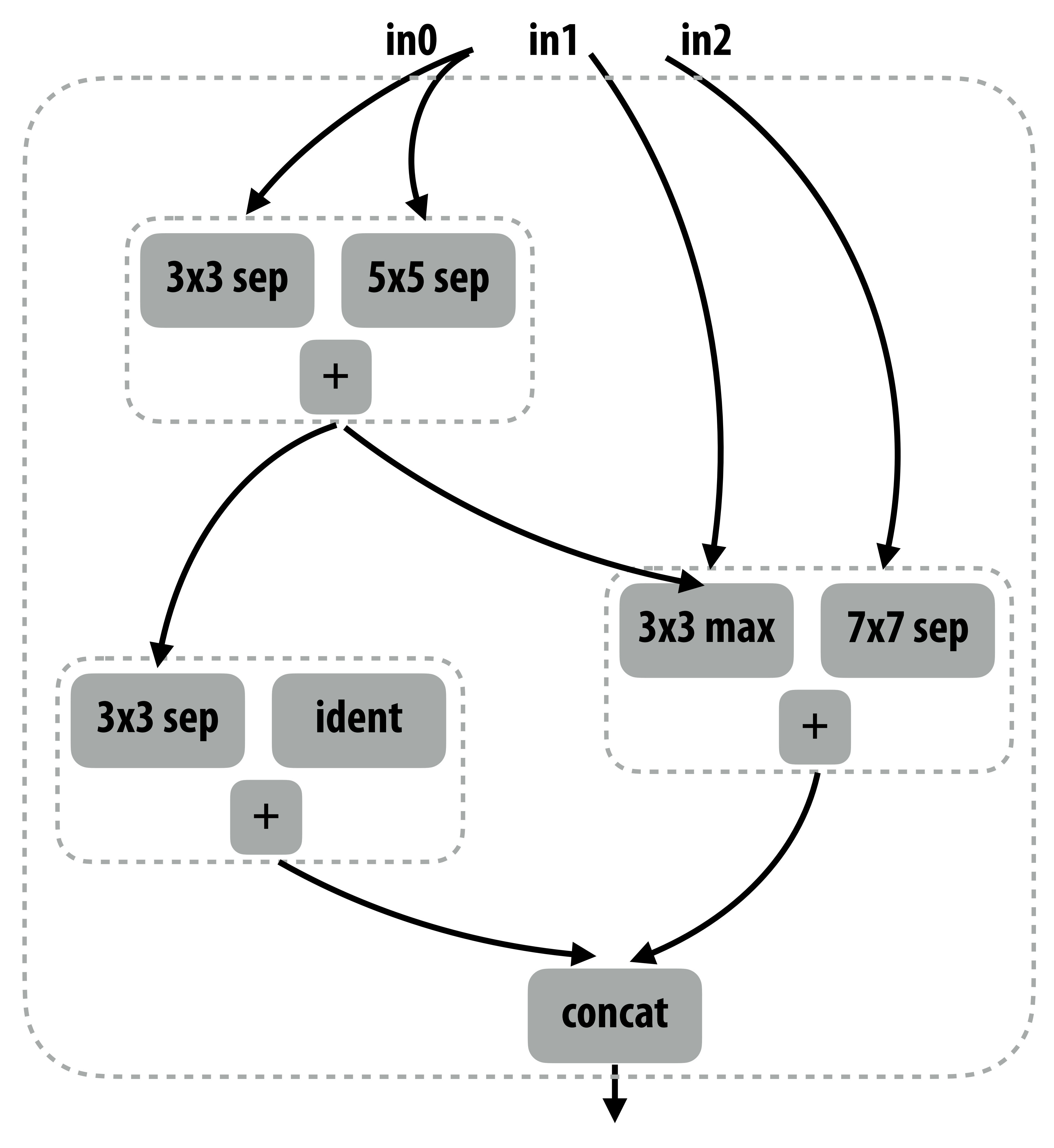
Eight possible operations:

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

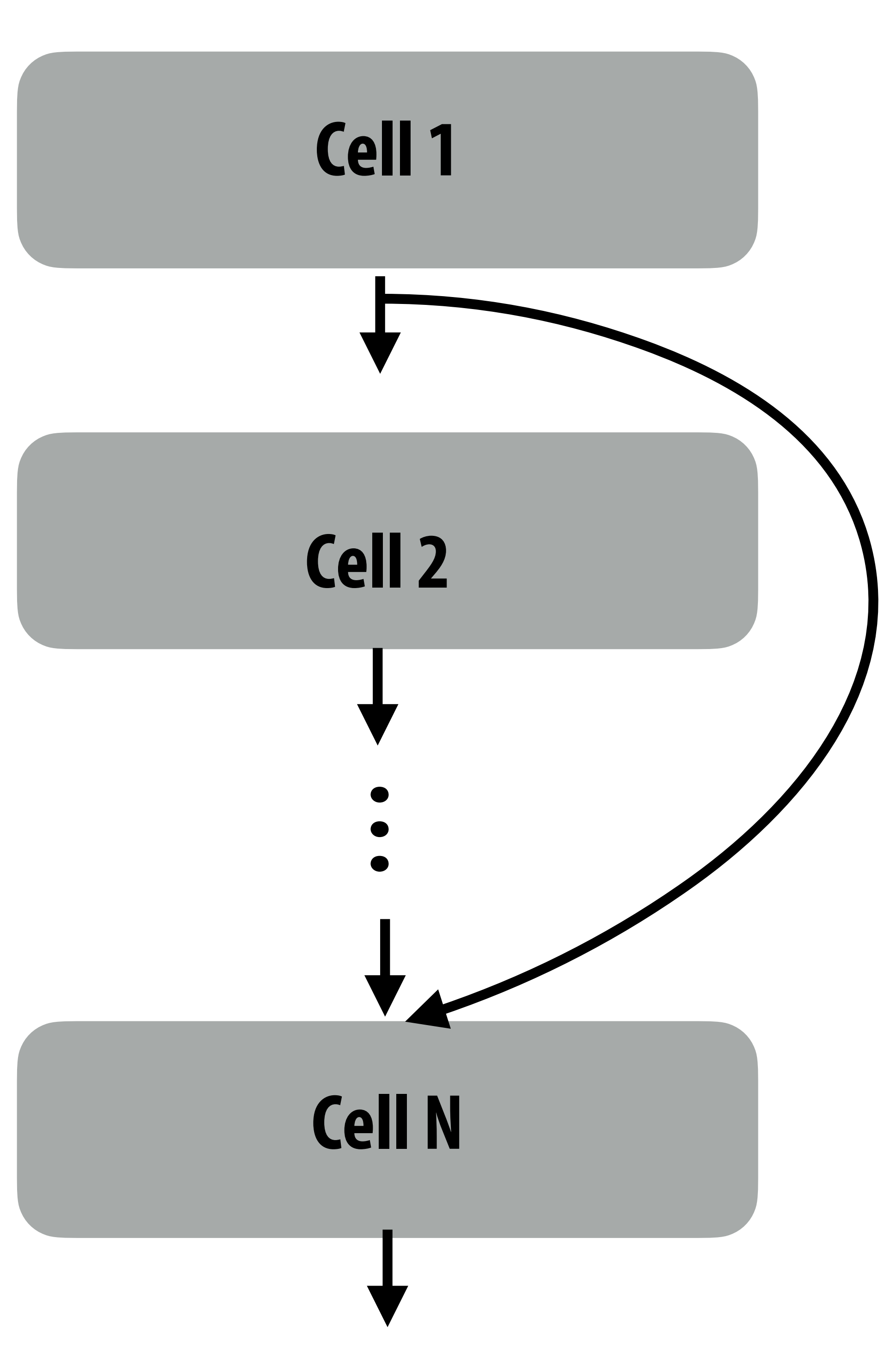
[Liu et al. 18]

Architecture search space

Cells are DAGs of *B* blocks



DNNs are sequences of *N* cells



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

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

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

consider:
arithmetic intensity, SIMD, data-parallelism, memory bandwidth requirements

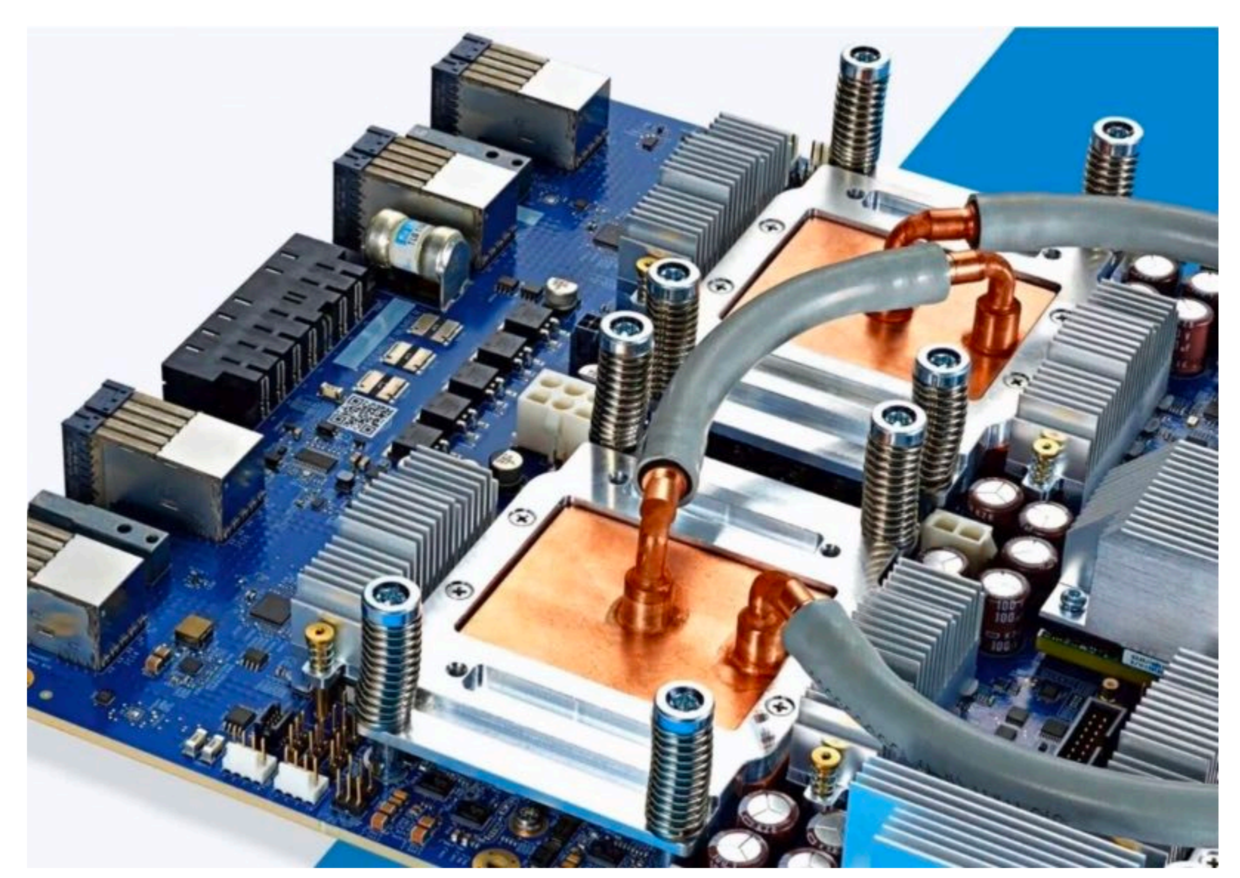
Deep neural networks on GPUs

- **Many high-performance DNN implementations target GPUs**
  - High arithmetic intensity computations (computational characteristics similar to dense matrix-matrix multiplication)
  - Benefit from flop-rich GPU architectures
  - Highly-optimized library of kernels exist for GPUs (cuDNN)

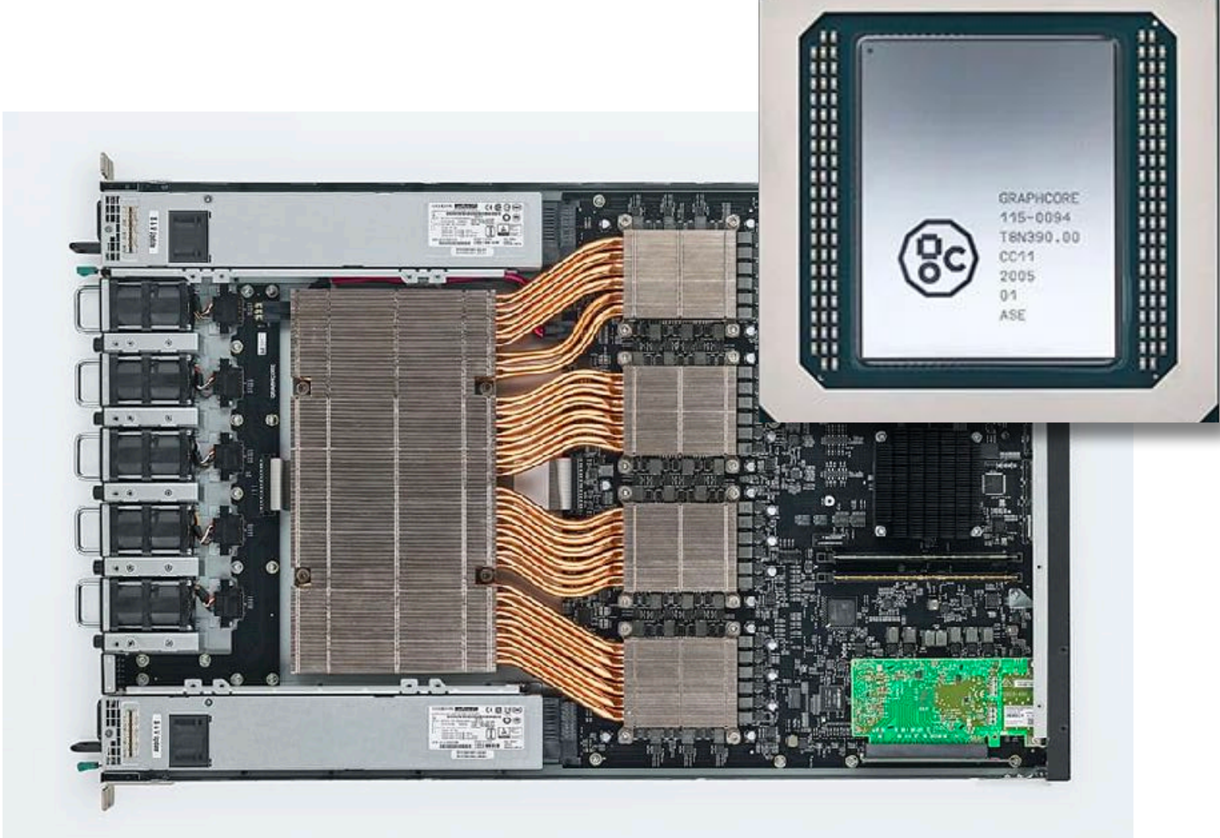
Why might a GPU be a sub-optimal platform for DNN evaluation?

consider: is a general purpose processor needed?

Hardware acceleration of DNN inference/training



Google TPU3

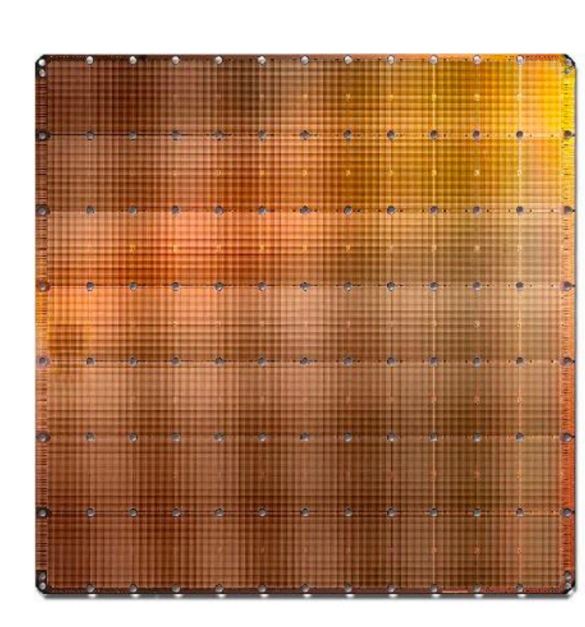


GraphCore IPU

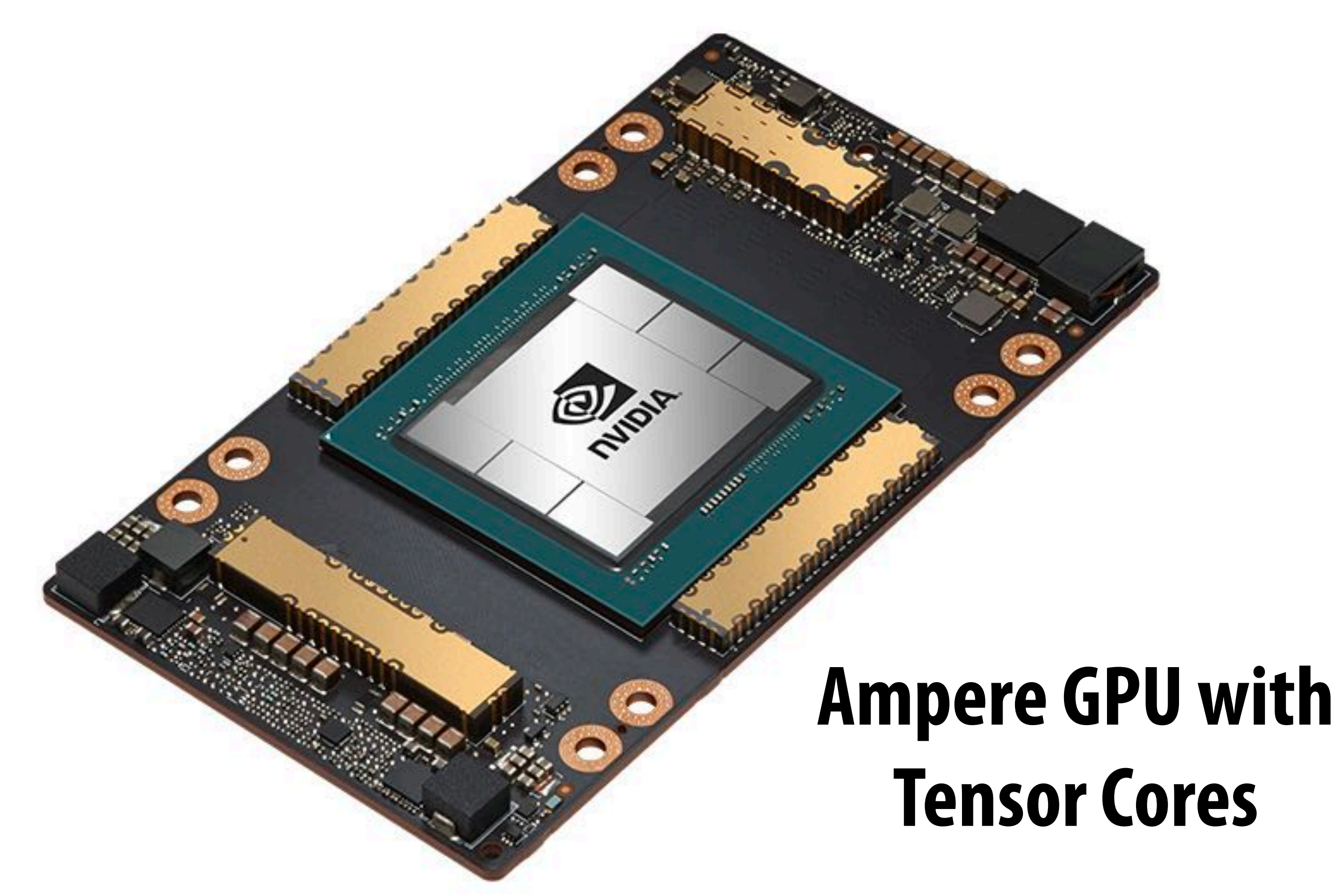
Apple Neural Engine

Intel Deep Learning
Inference Accelerator

SambaNova
Cardinal SN10

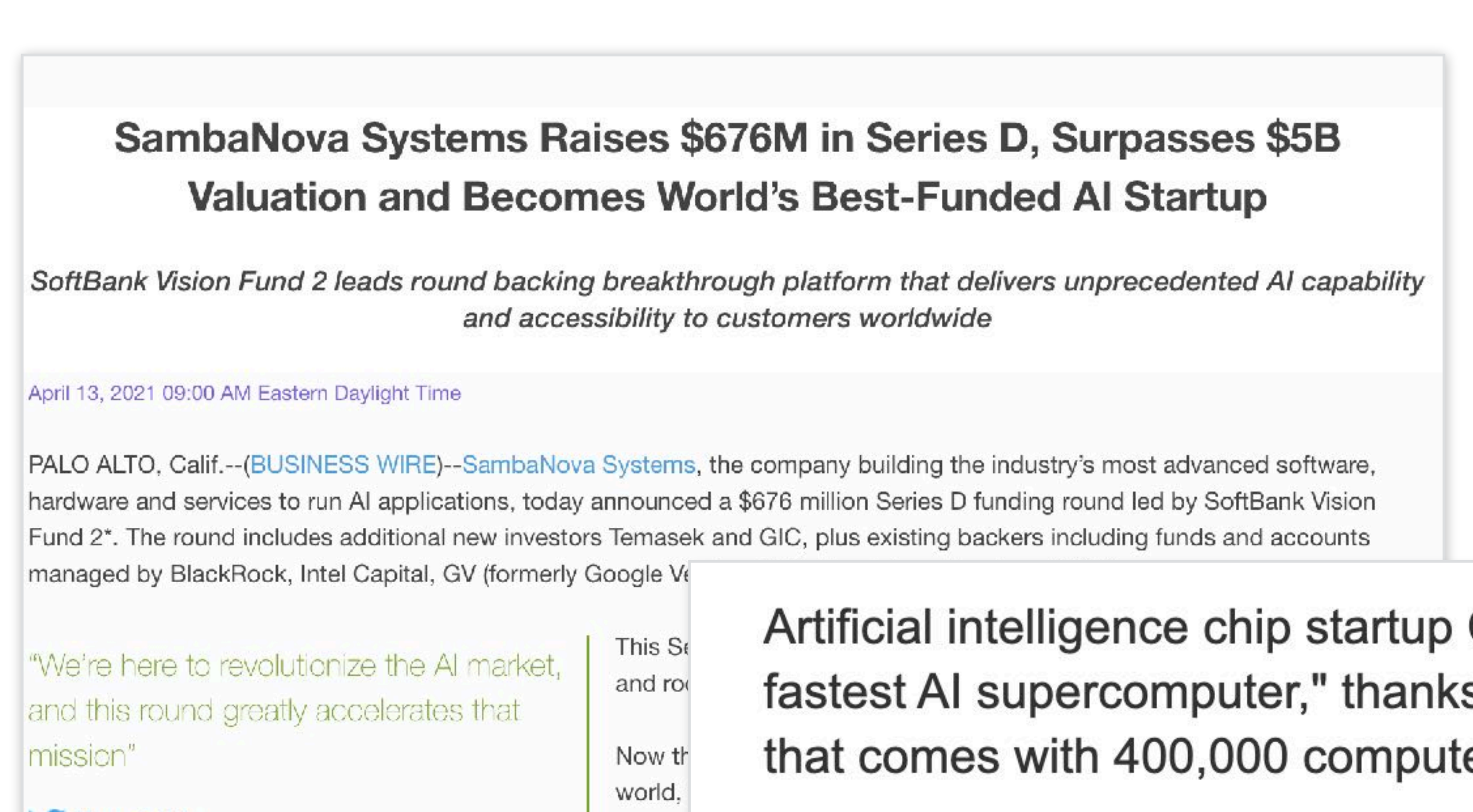


Cerebras Wafer Scale Engine



Ampere GPU with
Tensor Cores

Investment in AI hardware



SambaNova Systems Raises $676M in Series D, Surpasses $5B Valuation and Becomes World's Best-Funded AI Startup

SoftBank Vision Fund 2 leads round backing breakthrough platform that delivers unprecedented AI capability and accessibility to customers worldwide

April 13, 2021 09:00 AM Eastern Daylight Time

PALO ALTO, Calif.--(BUSINESS WIRE)--SambaNova Systems, the company building the industry's most advanced software, hardware and services to run AI applications, today announced a $676 million Series D funding round led by SoftBank Vision Fund 2*. The round includes additional new investors Temasek and GIC, plus existing backers including funds and accounts managed by BlackRock, Intel Capital, GV (formerly Google Ve

"We're here to revolutionize the AI market, and this round greatly accelerates that mission"

Tweet this

This S and ro

Now th world, legacy hardwa solutio

"We're here to revolutionize the AI market, and this round gre founder and CEO. "Traditional CPU and GPU architectures ha to solve humanity's greatest technology challenges, a new ap to see a wealth of prudent investors validate that."

SambaNova's flagship offering is Dataflow-as-a-Service (Daa to jump-start enterprise-level AI initiatives, augmenting organ centers, allowing the organization to focus on its business objectives instead of infrastructure.

Artificial intelligence chip startup Cerebras Systems claims it has the "world's fastest AI supercomputer," thanks to its large Wafer Scale Engine processor that comes with 400,000 compute cores.

The Los Altos, Calif.-based startup introduced its CS-1 system at the Supercomputing conference in Denver last week after raising more than $200 million in funding from investors, most recently with an $88 million Series D round that was raised in November 2018, according to Andrew Feldman, the founder and CEO of Cerebras who was previously an executive at AMD.

AI chipmaker Graphcore raises $222M at a $2.77B valuation and puts an IPO in its sights

Ingrid Lunden @ingridlunden / 10:59 PM PST • December 28, 2020

Comment

Applications based on artificial intelligence — whether they are systems running autonomous services, platforms being used in drug development or to predict the spread of a virus, traffic management for 5G networks or something else altogether — require an unprecedented amount of computing power to run. And today, one of the big names in the world of designing and

Groq Closes $300 Million Fundraise

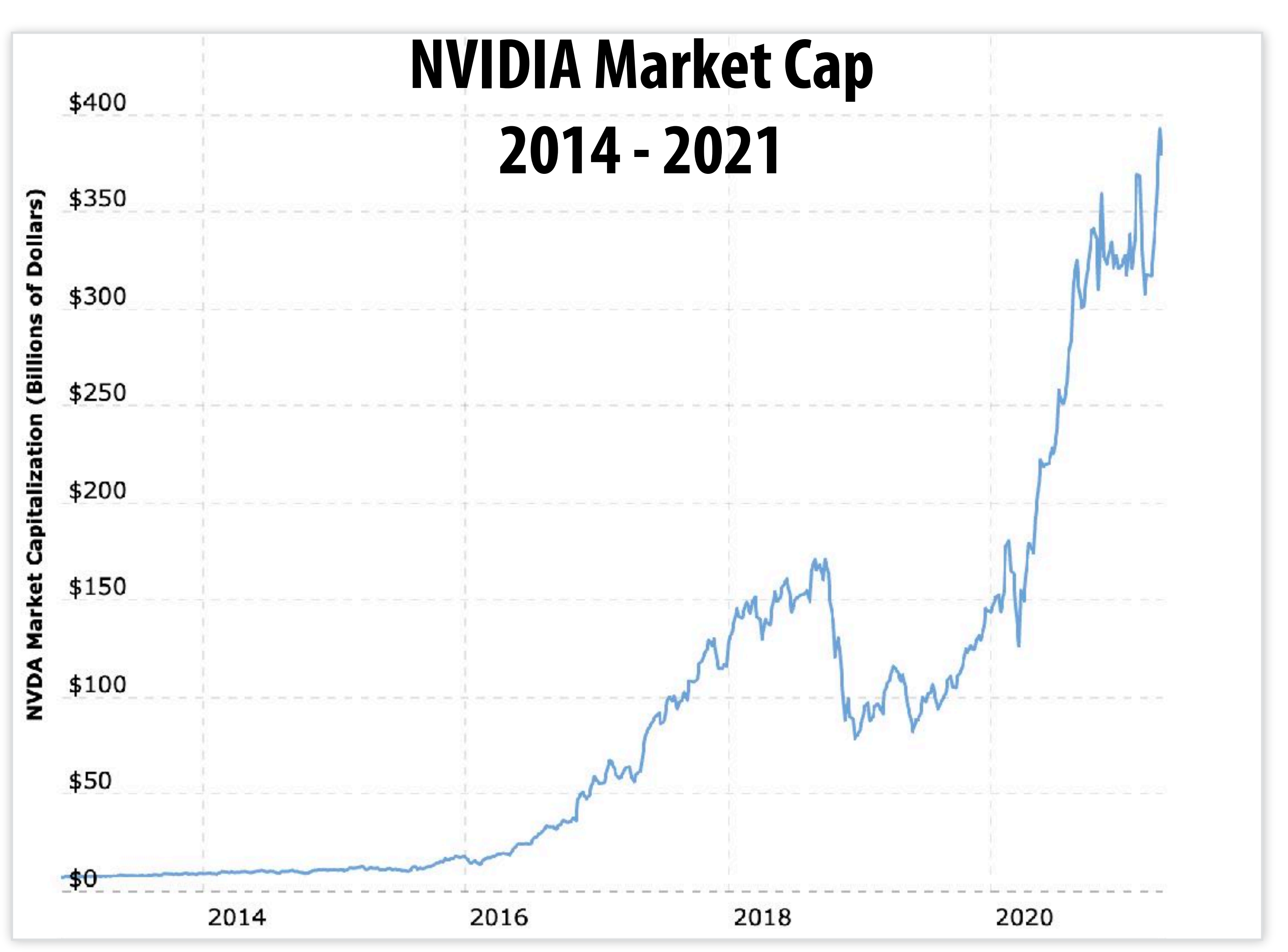
Wed, April 14, 2021, 6:00 AM · 4 min read

With Investment Co-Led by Tiger Global Management and D1 Capital, Groq Is Well Capitalized for Accelerated Growth

MOUNTAIN VIEW, Calif., April 14, 2021 /PRNewswire/ -- Groq Inc., a leading innovator in compute accelerators for artificial intelligence (AI), machine learning (ML) and high performance computing, today announced that it has closed its Series C fundraising. Groq closed $300 million in new funding, co-led by Tiger Global Management and D1 Capital, with participation from The Spruce House Partnership and Addition, the venture firm founded by Lee Fixel. This round brings Groq's total funding to $367 million, of which $300 million has been raised since the second-half of 2020, a direct result of strong customer endorsement since the company launched its first product.

groq™

Groq logo



Intel Acquires Artificial Intelligence Chipmaker Habana Labs

Combination Advances Intel's AI Strategy, Strengthens Portfolio of AI Accelerators for the Data Center

SANTA CLARA Calif., Dec. 16, 2019 – Intel Corporation today announced that it has acquired Habana Labs, an Israel-based developer of programmable deep learning accelerators for the data center for approximately $2 billion. The combination strengthens Intel's artificial intelligence (AI) portfolio and accelerates its efforts in the nascent, fast-growing AI silicon market, which Intel expects to be greater than $25 billion by 2024[1].

"This acquisition advances our AI strategy, which is to provide customers with solutions to fit every performance need – from the intelligent edge to the data center," said Navin Shenoy, executive vice president and general manager of the Data Platforms Group at Intel. "More specifically, Habana turbo-charges our AI offerings for the data center with a high-performance training processor family and a standards-based programming environment to address evolving AI workloads."

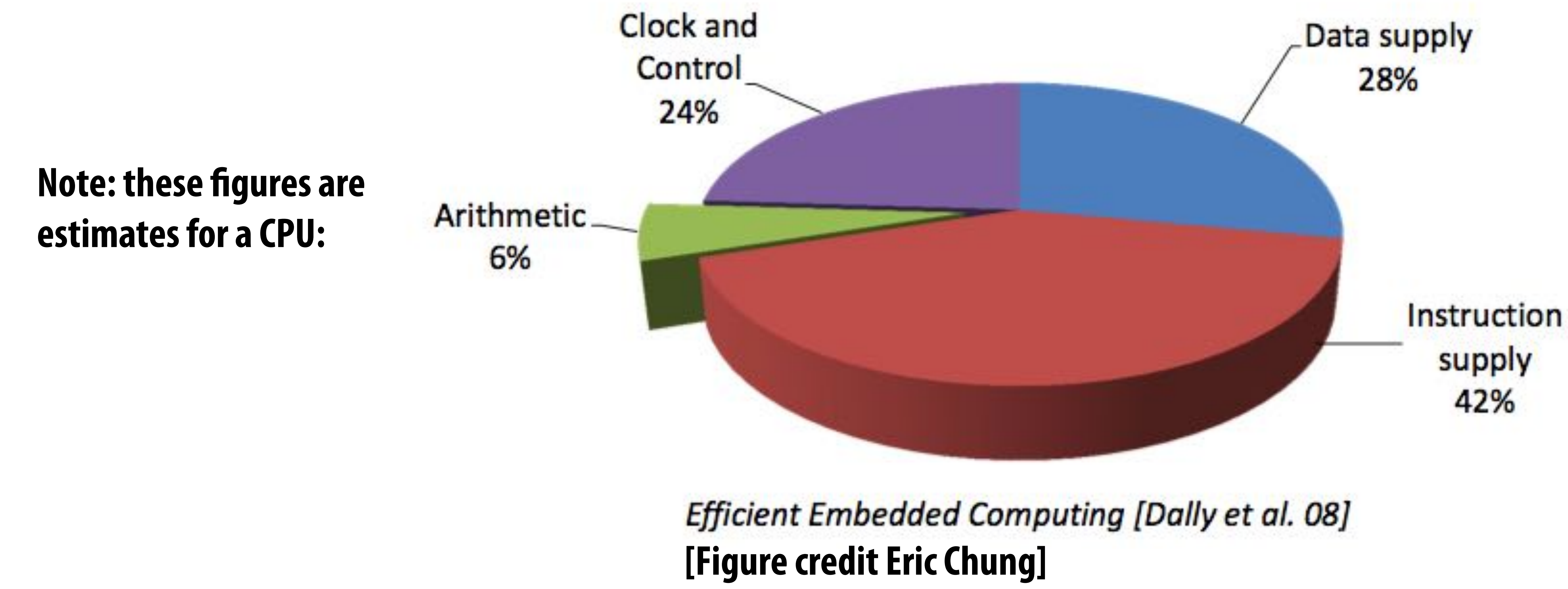
Recall: properties of GPUs

- "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:



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

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

There are 80 SM cores in the GV100 GPU:

5,120 fp32 mul-add ALUs

640 tensor cores

6 MB of L2 cache

1.5 GHz max clock

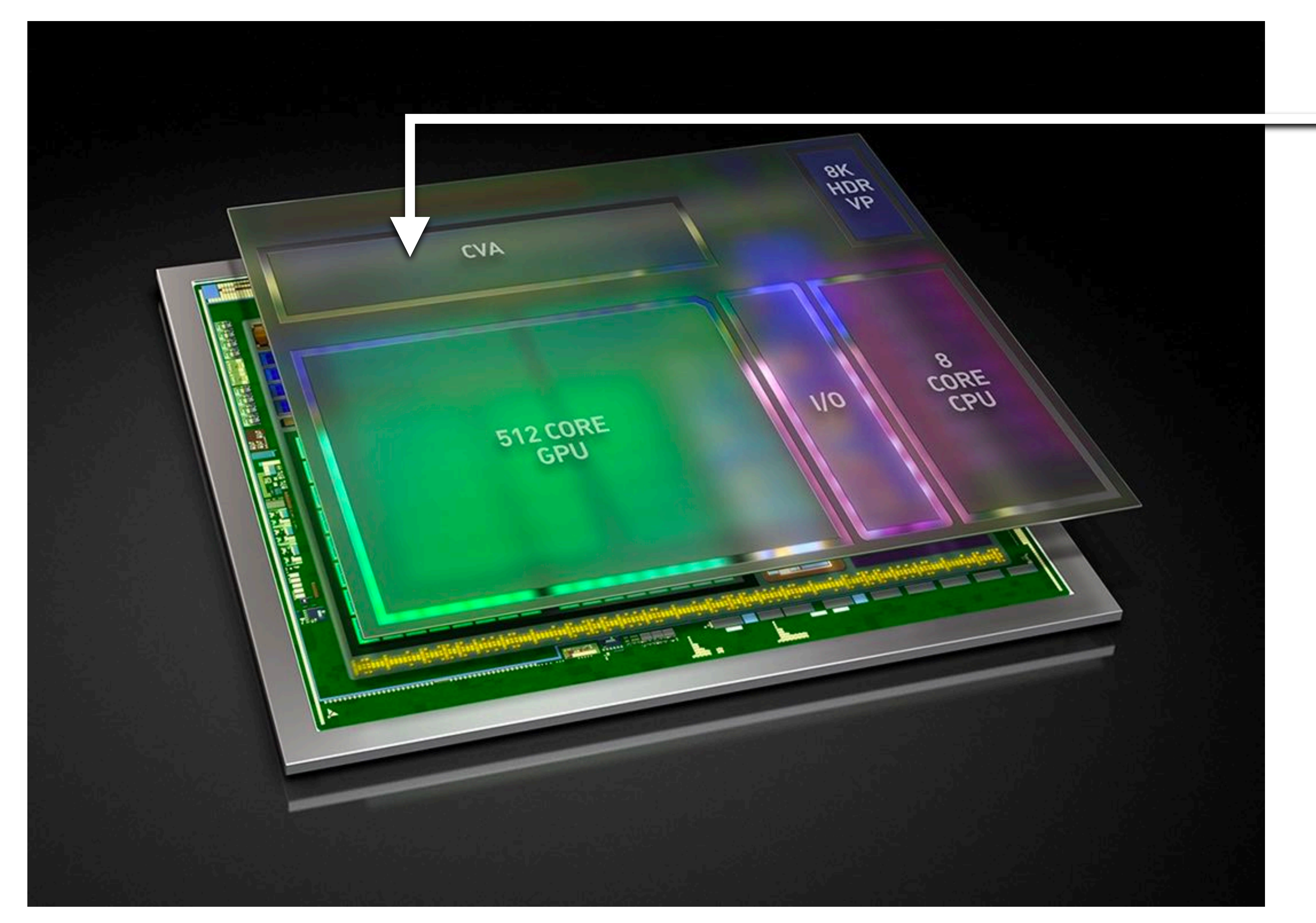
= 15.7 TFLOPs fp32

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

Single instruction to perform 2x4x4x4 + 4x4 ops

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

Summary: efficiently evaluating deep nets

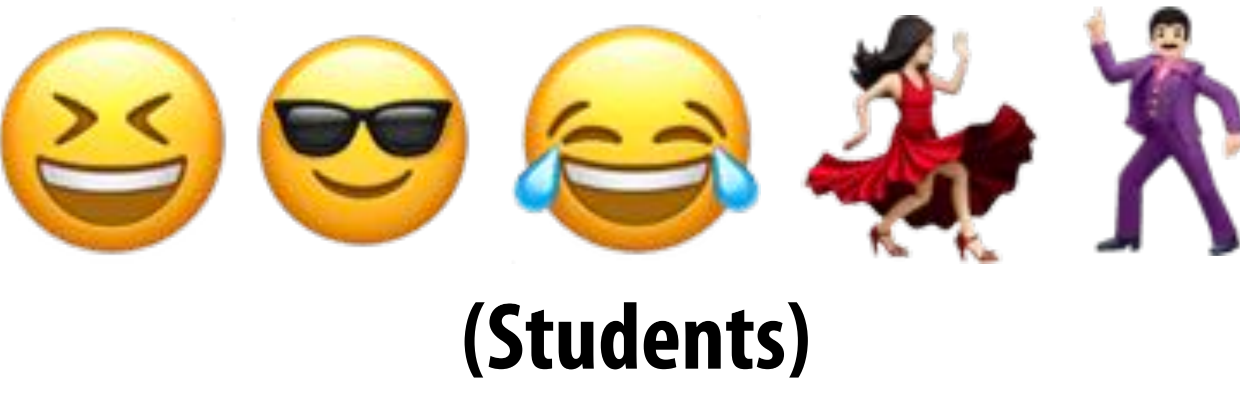
- Workload characteristics for image processing DNNs:
  - Convlayers: high arithmetic intensity, significant portion of cost when evaluating DNNs for computer vision

- Significant interest in reducing size of DNNs for more efficient evaluation

- Algorithmic techniques (better DNN model architectures) are responsible for significant speedups in recent years
  - Expect increasing use of automated model search techniques

- Huge innovation in specialized hardware accelerators

Course Wrap Up

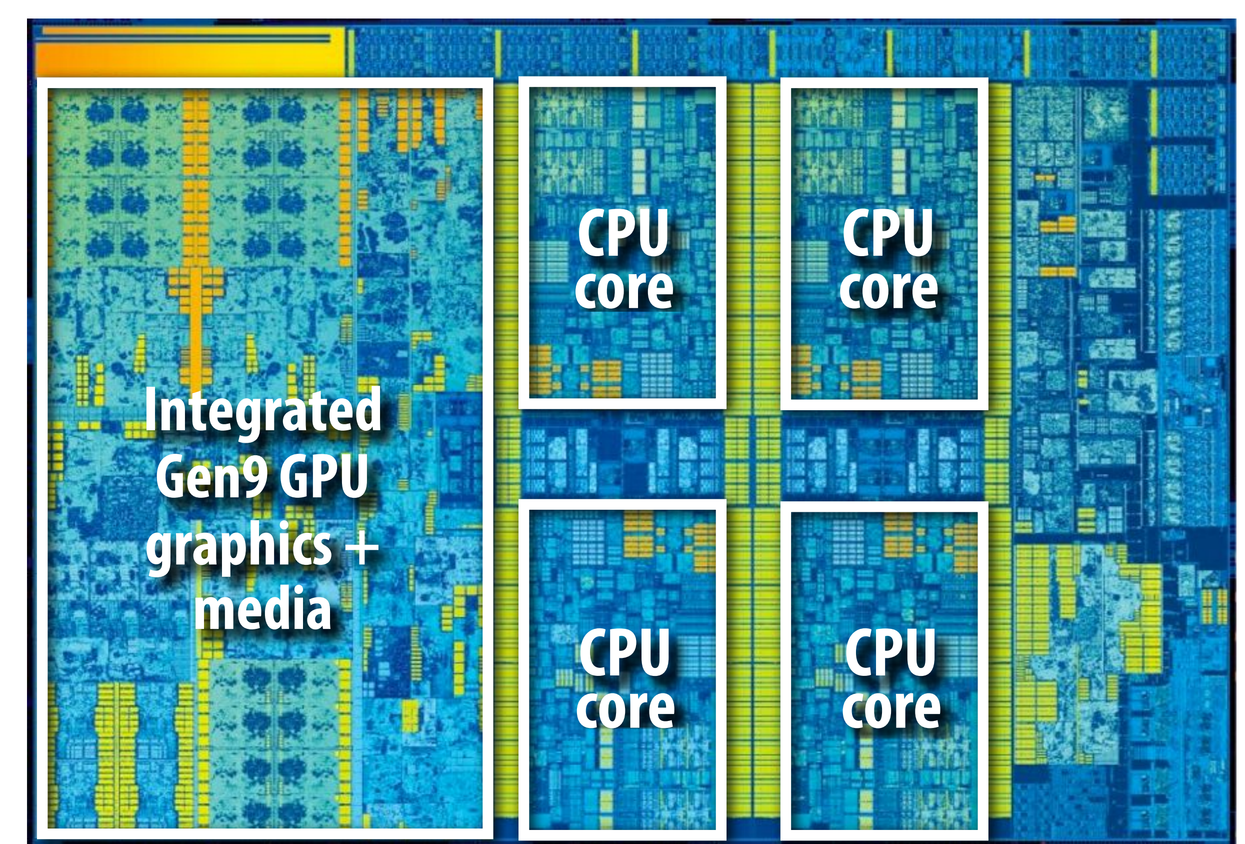


(Students)

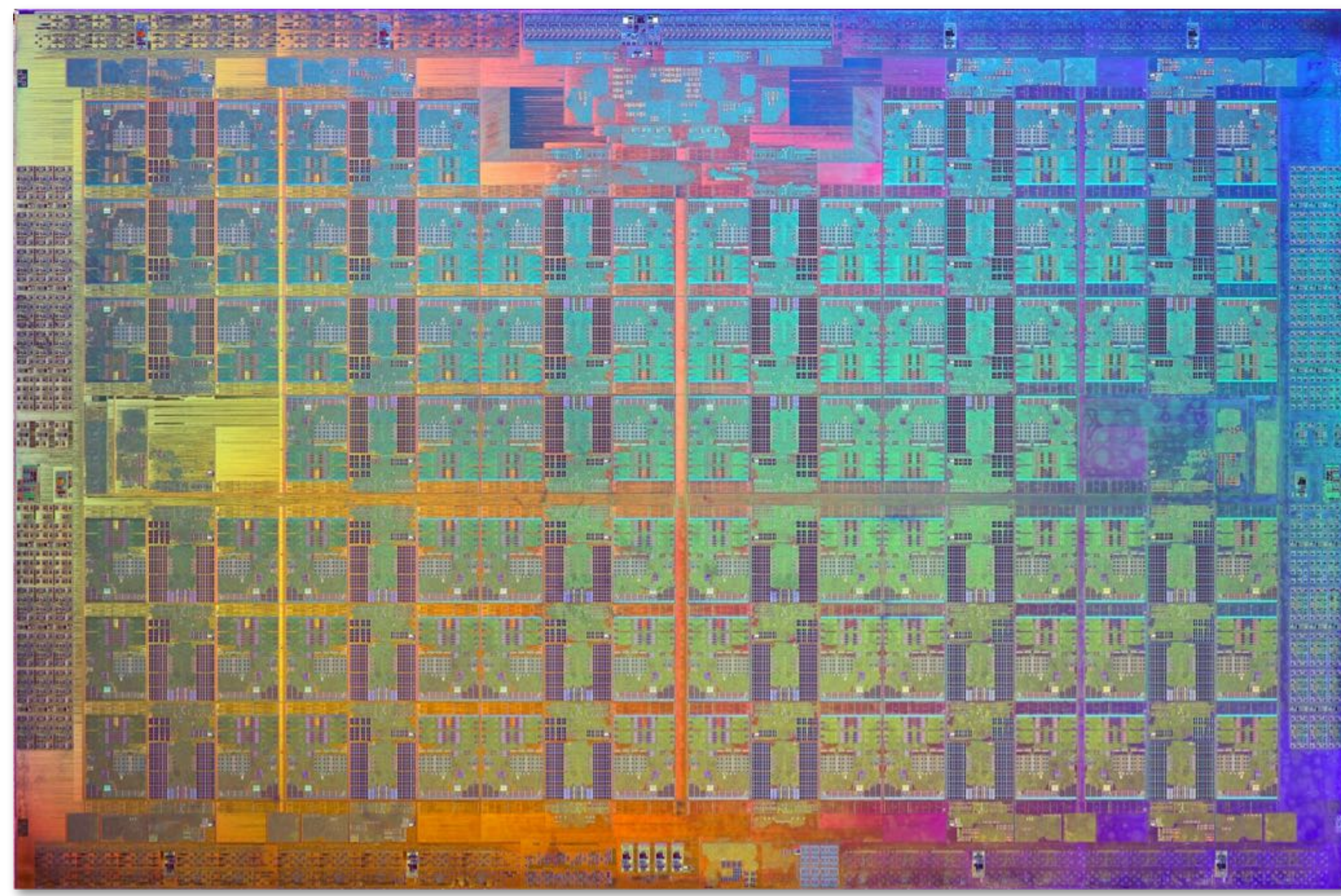
GeForce GTX 980 Whitepaper

GM204 HARDWARE ARCHITECTURE IN-DEPTH

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

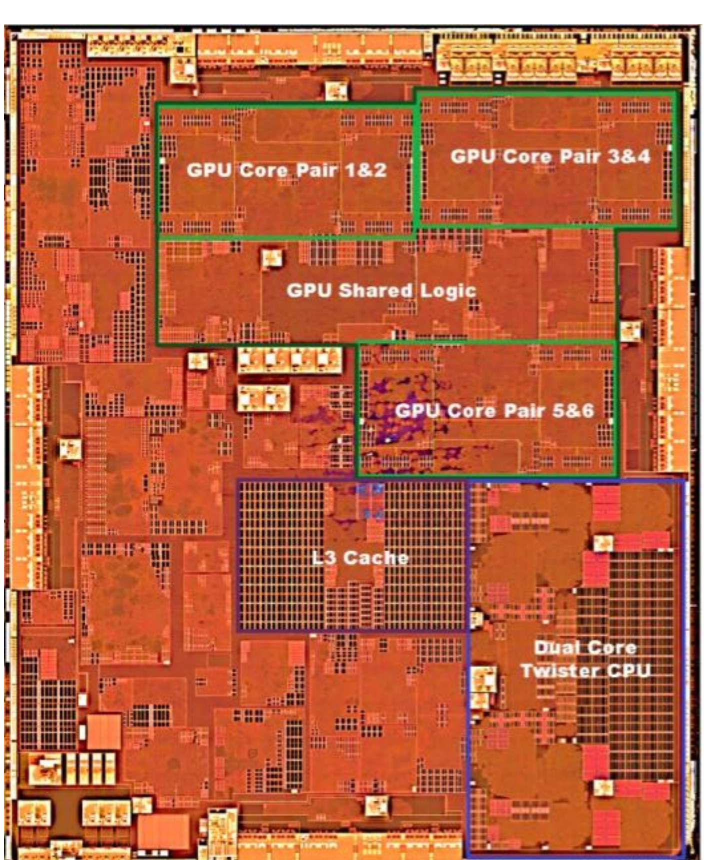


Intel Xeon Phi
72 cores, 16-wide SIMD, 4-way multi-threading

Figure 3: GM204 SMM Diagram (GM204 also features 4 DP units per SMM, which are not depicted on 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 to key domains

After taking this course, you can play a role in ongoing Stanford research in parallel computing!

Try CURIS/independent study/research!

Why research (or independent study)?

- Depth can be fun. You will learn way more about a topic than in any class.
- You think your undergrad/MS peers are amazingly smart? Come see our Ph.D. students! (you get to work side-by-side with them and with faculty). Imagine what level you might rise to.
- It's 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 might be possible with tech.

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

- Generating efficient code from DSLs for image processing or deep learning
- Designing a platform to render frames at 10,000 fps per GPU to rapidly create training data for reinforcement learning
- Human-in-the-loop systems for mining large image databases for training data (can a human, a big monitor, and a supercomputer create accurate DNN models in an afternoon?)
- Parallel rendering using 1000's of CPU cores in the cloud
- Designing more efficient DNNs
- New applications of analyzing video data at scale
  - Analyzing broadcast sports video to make virtual characters that move and play like real athletes (virtual Roger Federer)
  - Analyzing 230,000 hours of TV news video to understand representation and bias in the news.

Maybe you might like research and decide you want to go to grad school

Pragmatic comment: Without question, the number one way to get into a top grad school is to receive a strong letter of recommendation from faculty members. You get that letter only from being part of a research team for an extended period of time.

DWIC letter: ("did well in class" letter) What you get when you ask for a letter from a faculty member who you didn't do research with, but got an 'A' in their class. This letter is essentially thrown out by the Ph.D. admissions committee at good schools.

A very good reference

CMU Professor Mor Harchol-Balter's writeup:

"Applying to Ph.D. Programs in Computer Science"

http://www.cs.cmu.edu/~harchol/gradschooltalk.pdf

HYPOTHESIS:

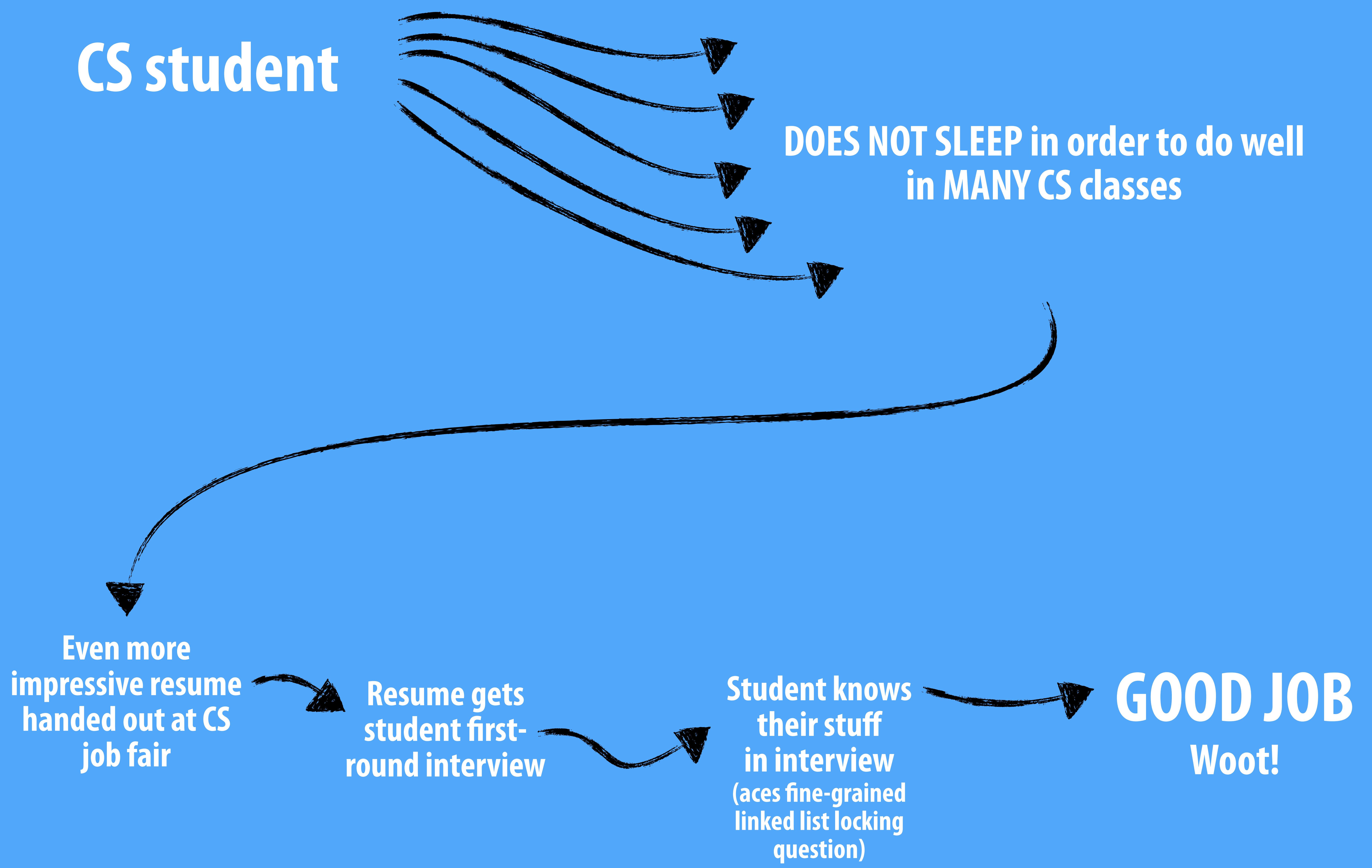
CS classes alone may not be the most effective way to maximize your experience at Stanford and opportunities afterward.

It may not be the best way to get a competitive job.

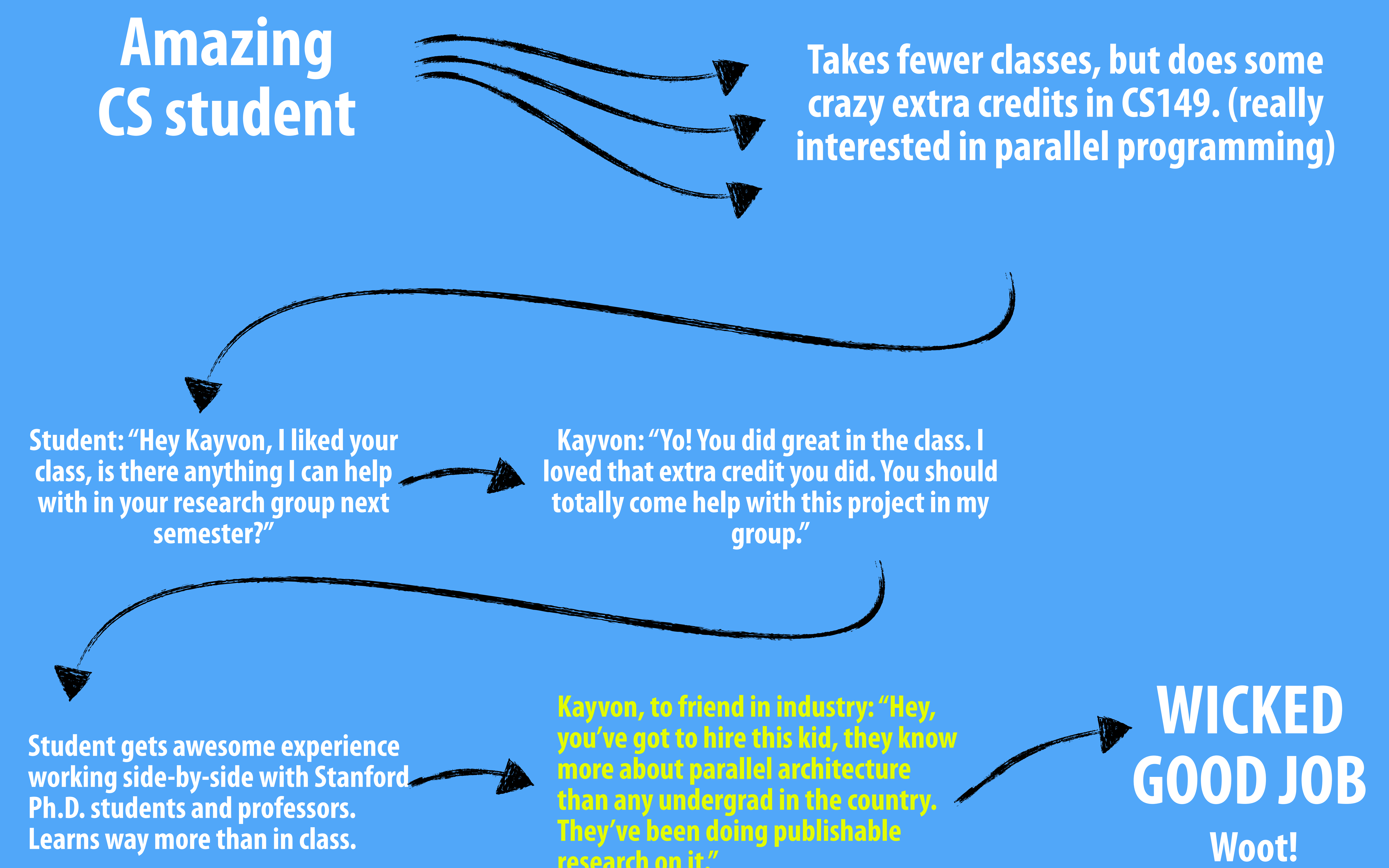
It may not be the best way to get the coolest jobs.

It may not be the best way to prepare yourself have the most impact in a future job or in the world at large.

A conventional path…



An alternative path…



Think bigger, think broader

You are fortunate.

You are smart, talented, and hard-working.

You are in an amazing environment at Stanford.

How can you maximize that opportunity while you are here?
The mechanisms are in place, if they aren't, we'll help you create them:

Course projects
Research
Independent study
Entrepreneurship

The biggest sign you are in the "real-world" isn't when you are paying your own bills, showing up to work on time, or ensuring your code passes regressions… it is asking your own questions and making your own decisions.

And there's a lot more to decide on than classes.

Or in other words*…
there are “grades” you can get at Stanford that are much higher than A/A+’s.

* taken from a colleague Dave Eckhardt

Thanks for being a great class!

Thanks for putting in the work. (in the face of stressful times)

Stay safe. Wear a mask.

Have a great break!