

Lecture 6:

Efficiently Evaluating Deep Networks

**Visual Computing Systems
Stanford CS348K, Spring 2023**

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)**

Efficiency challenge

Many DNN topologies
(Many variants on common backbones)

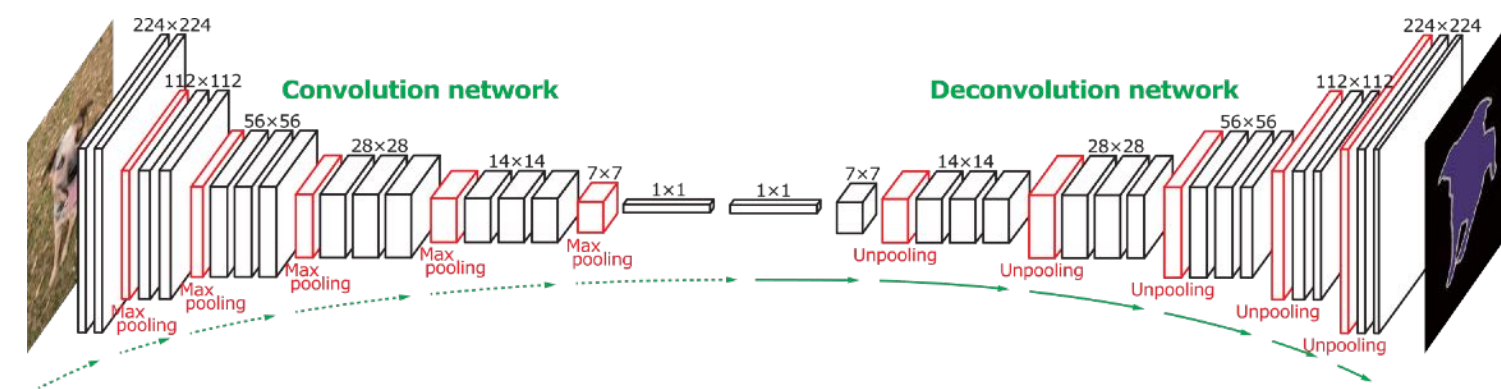
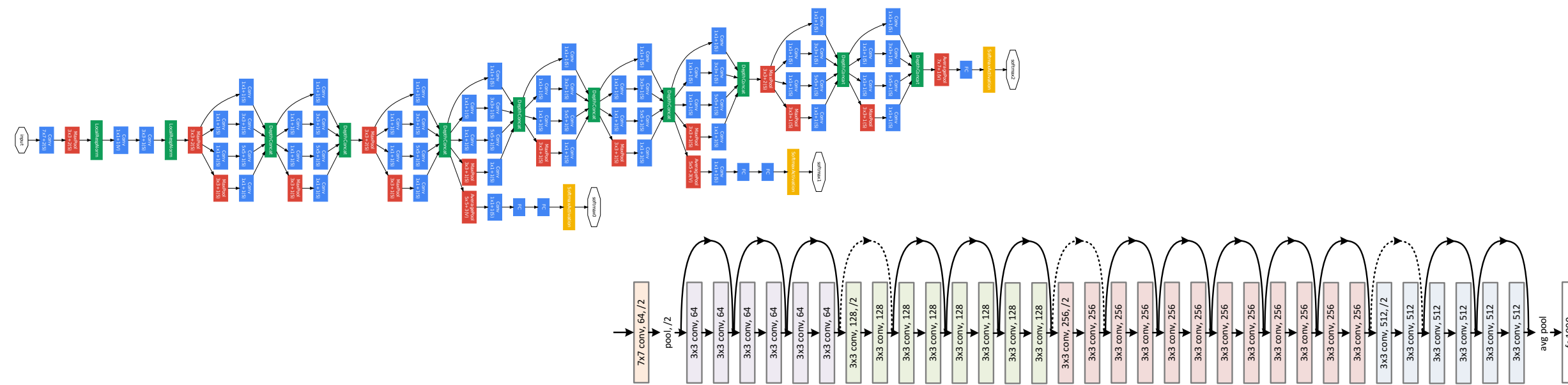


Table 1. MobileNet Body Architecture

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	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5x 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	$1 \times 1 \times 1000$

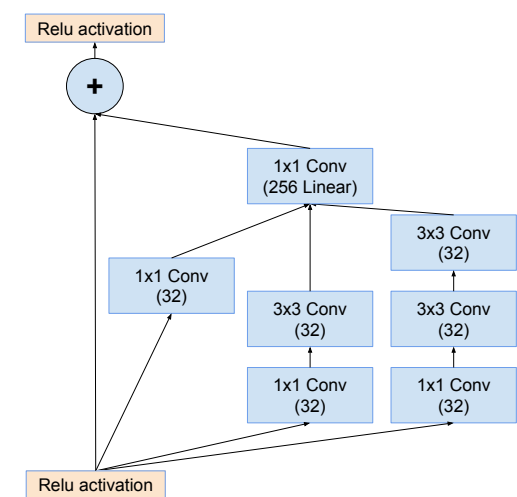
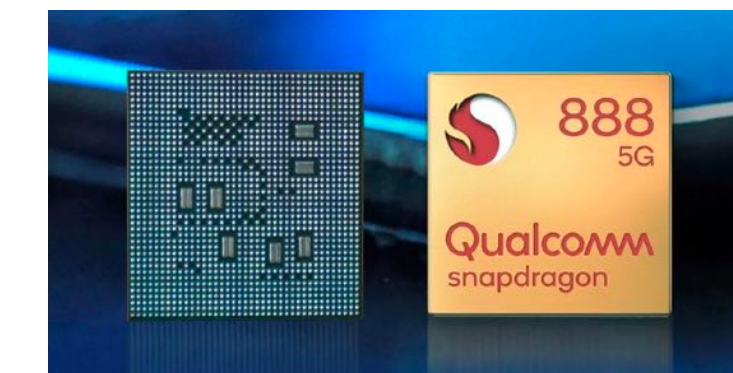


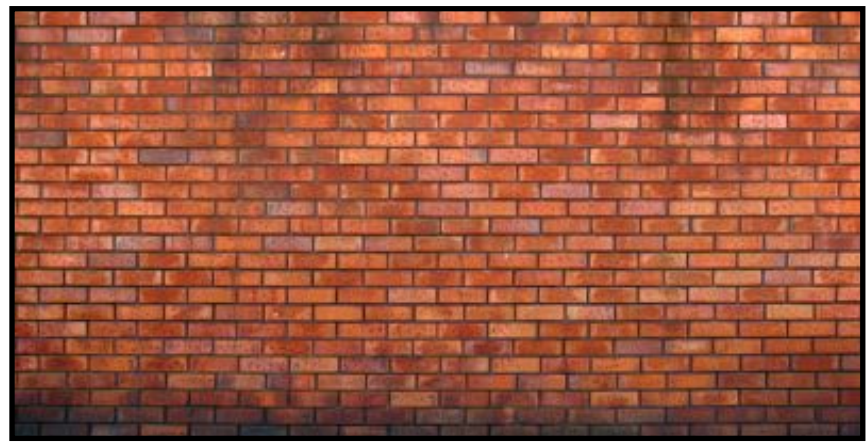
Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

Many Target Devices



**Mini-review / crash course:
Convolutional Neural Networks**

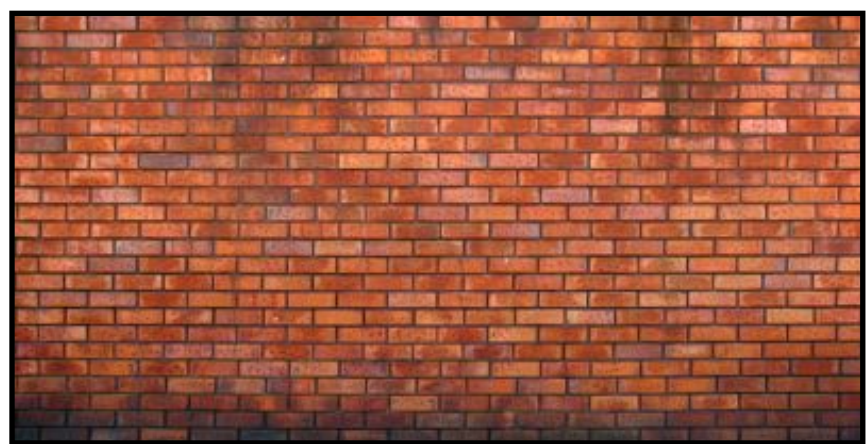
Gradient detection filters



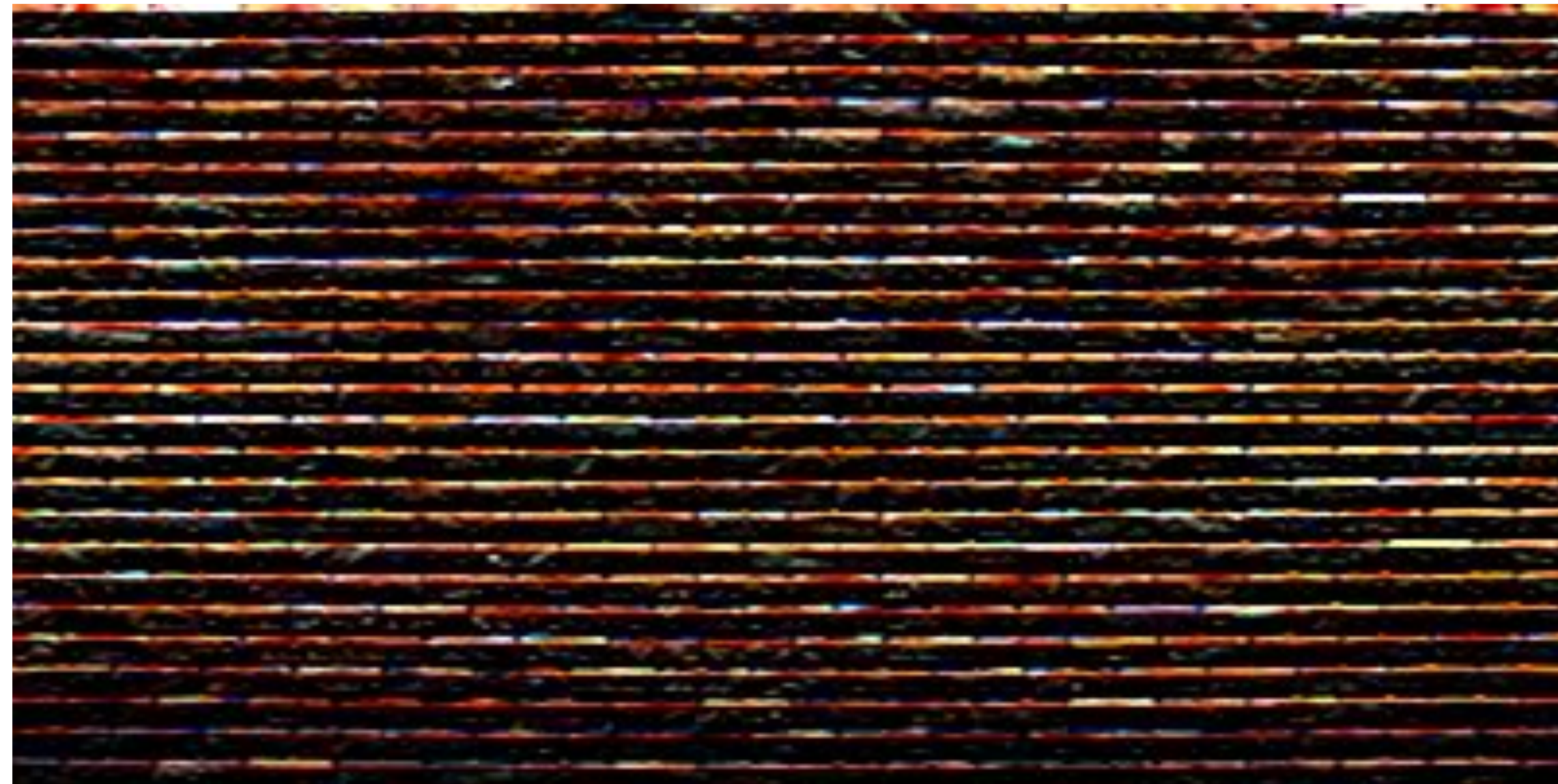
$$* \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} =$$



Responds to
horizontal
gradients



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



Responds to
vertical
gradients

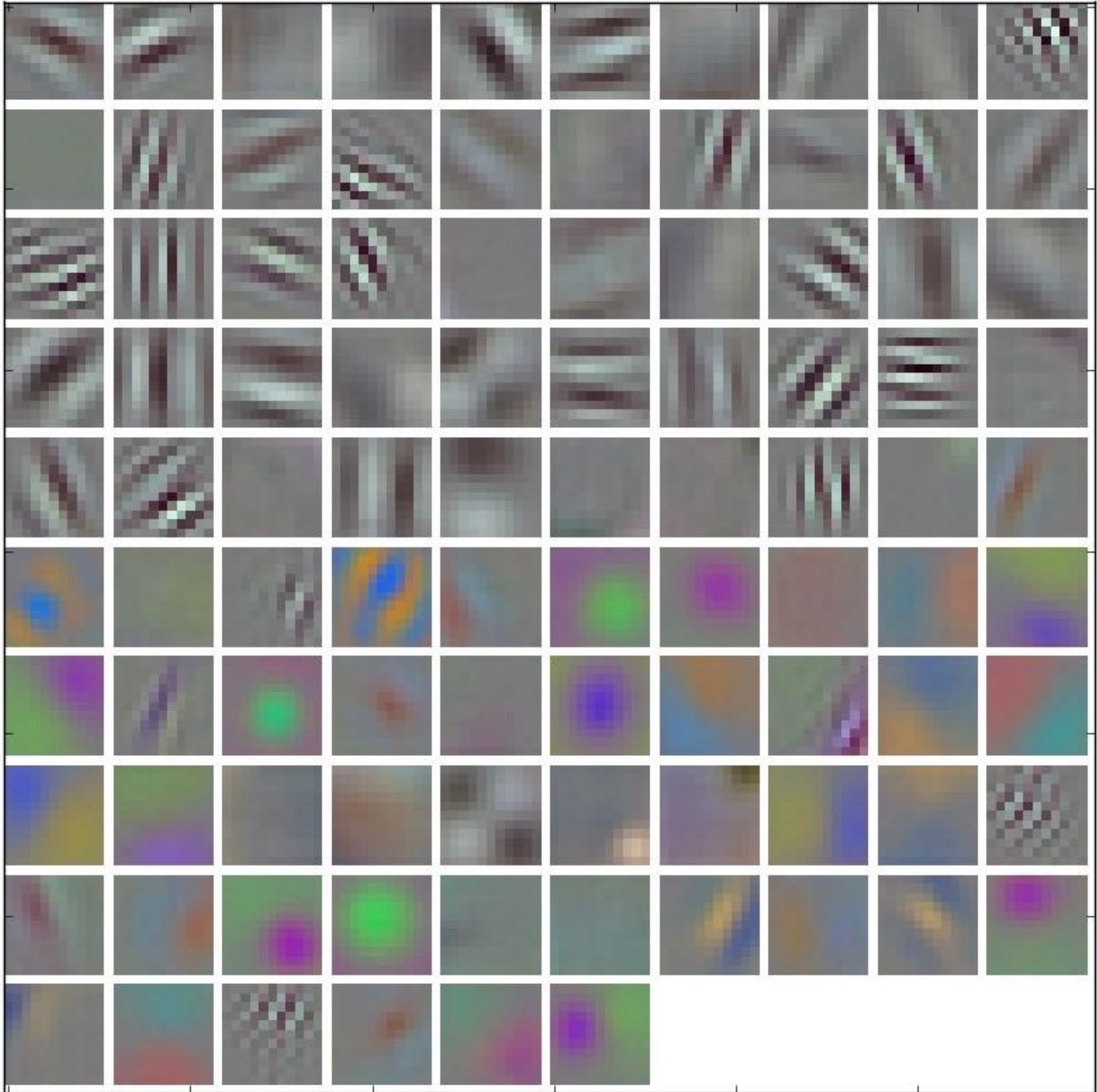
Note: you can think of a filter as a “detector” of a pattern, and the magnitude of a pixel in the output image as the “response” of the filter to the region surrounding each pixel in the input image

Applying many filters to an image at once

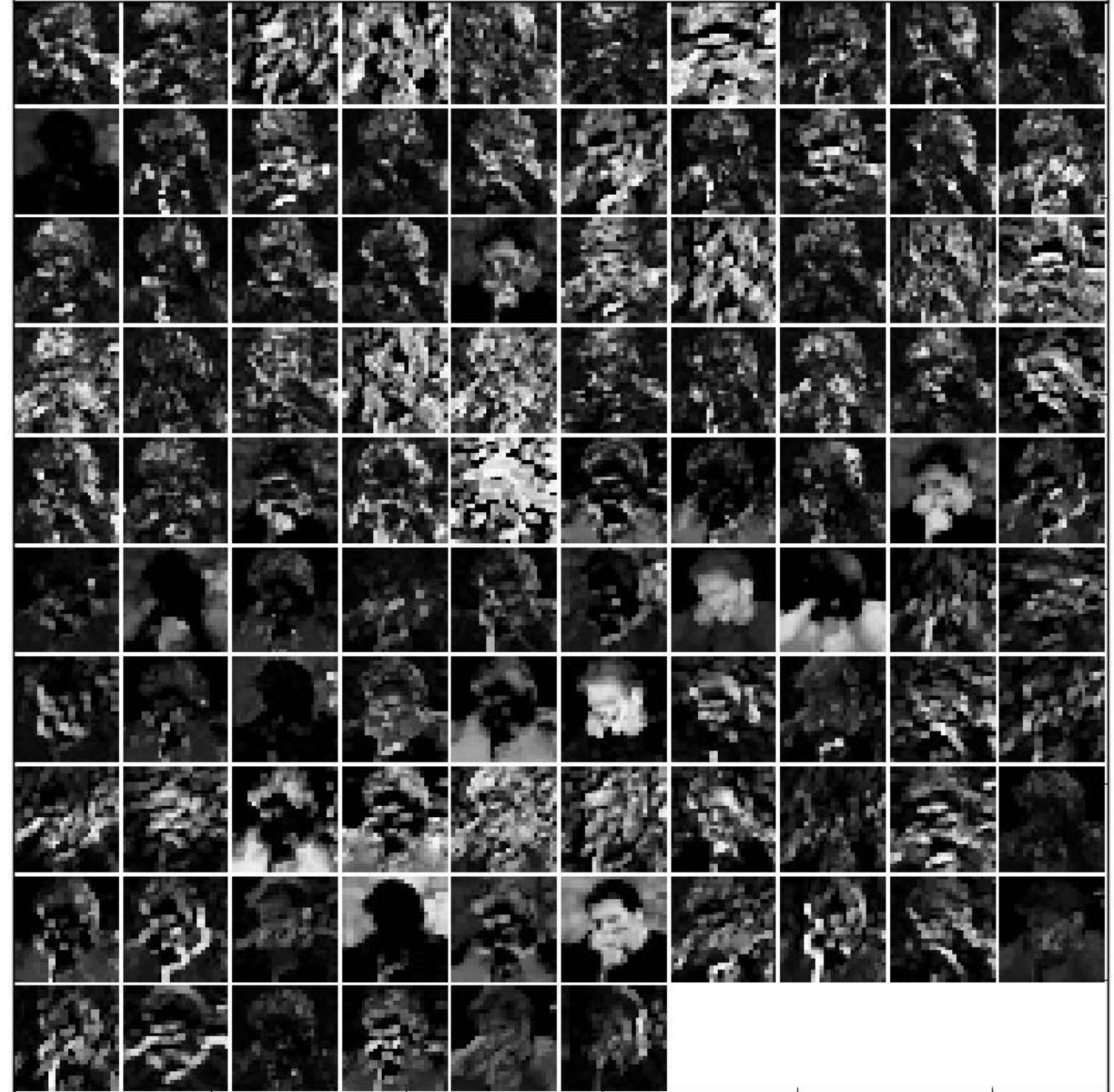
Input RGB image (W x H x 3)



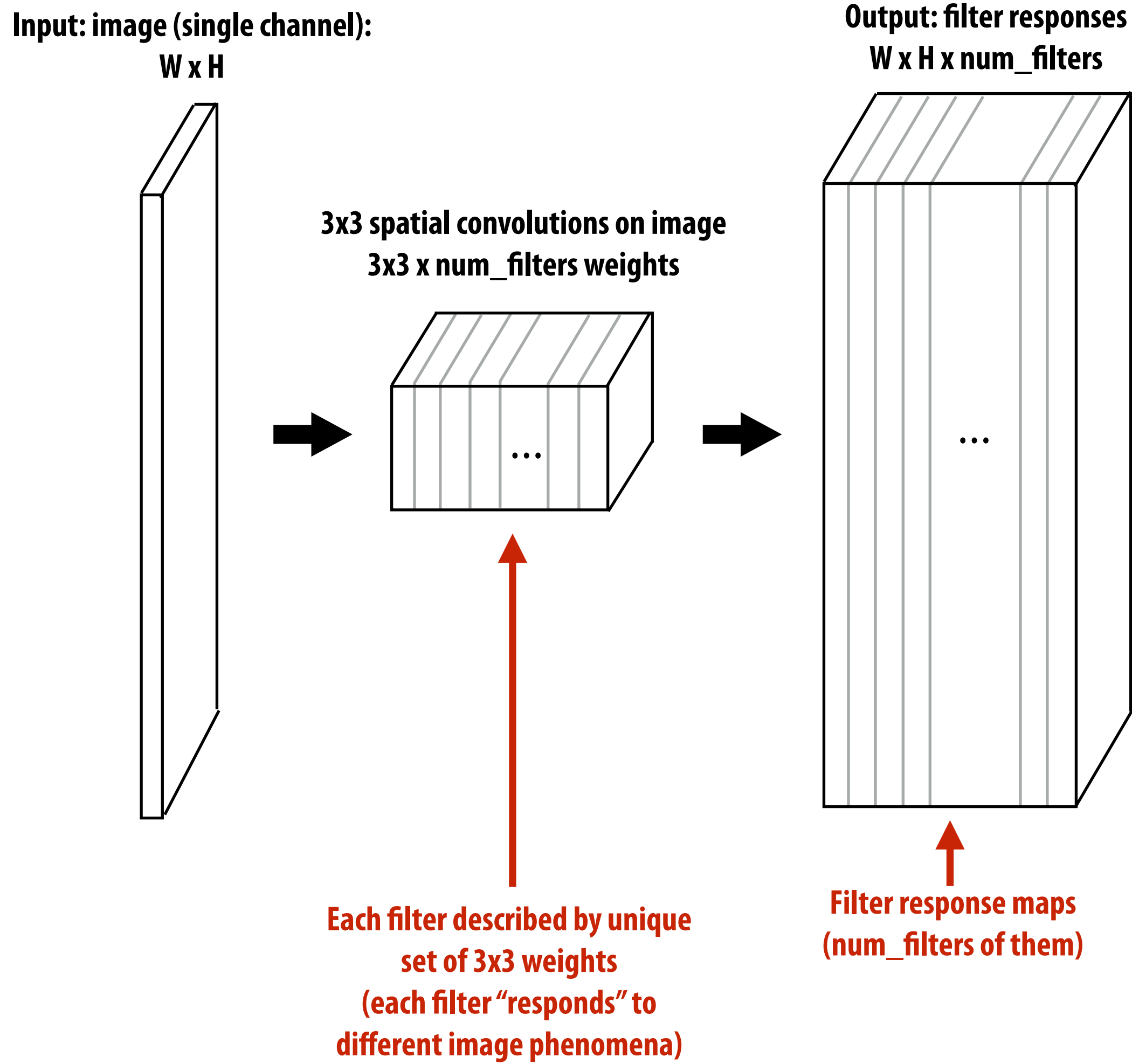
96 11x11x3 filters
(3D because they operate on RGB)



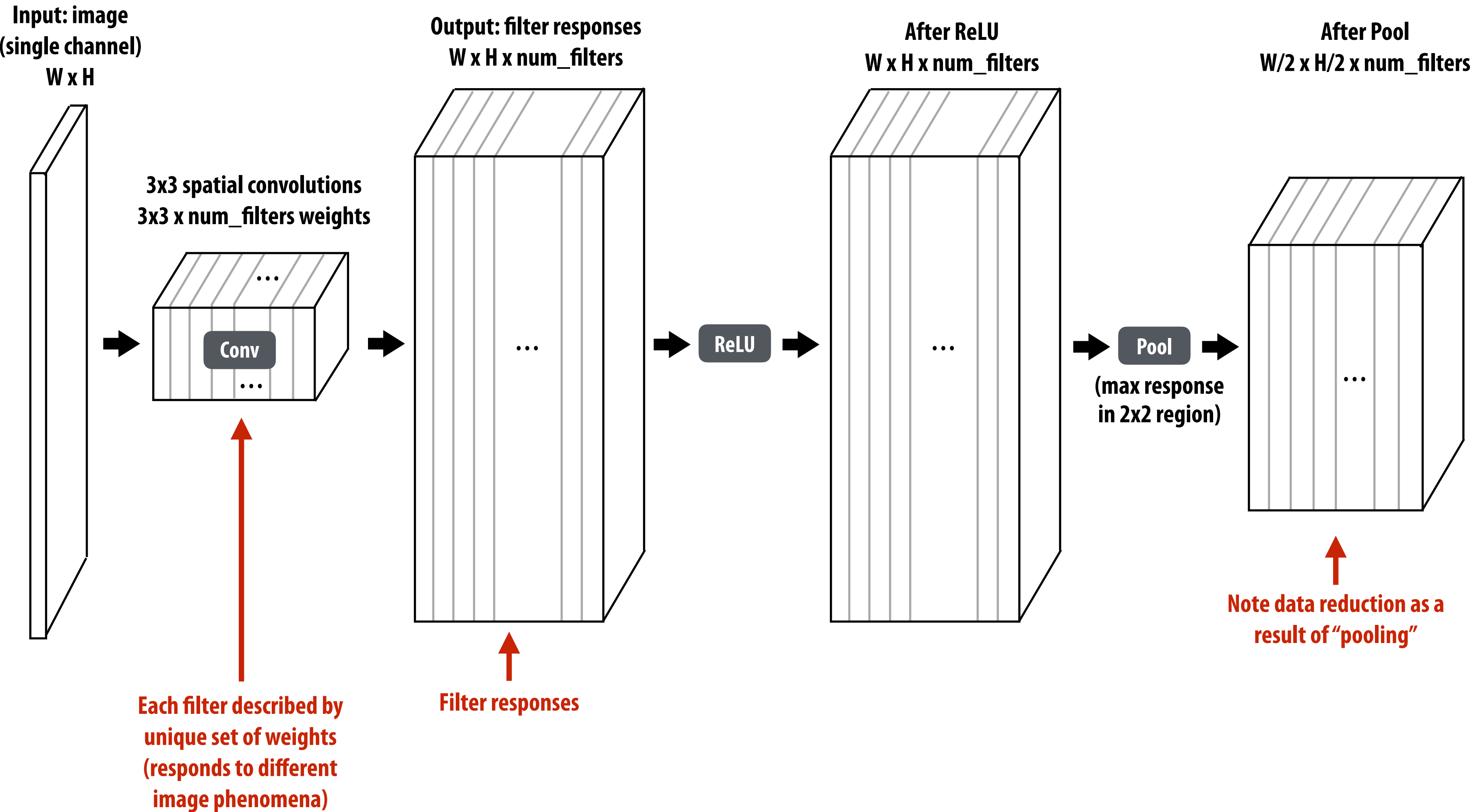
96 responses (normalized)



Applying many filters to an image at once



Adding additional layers

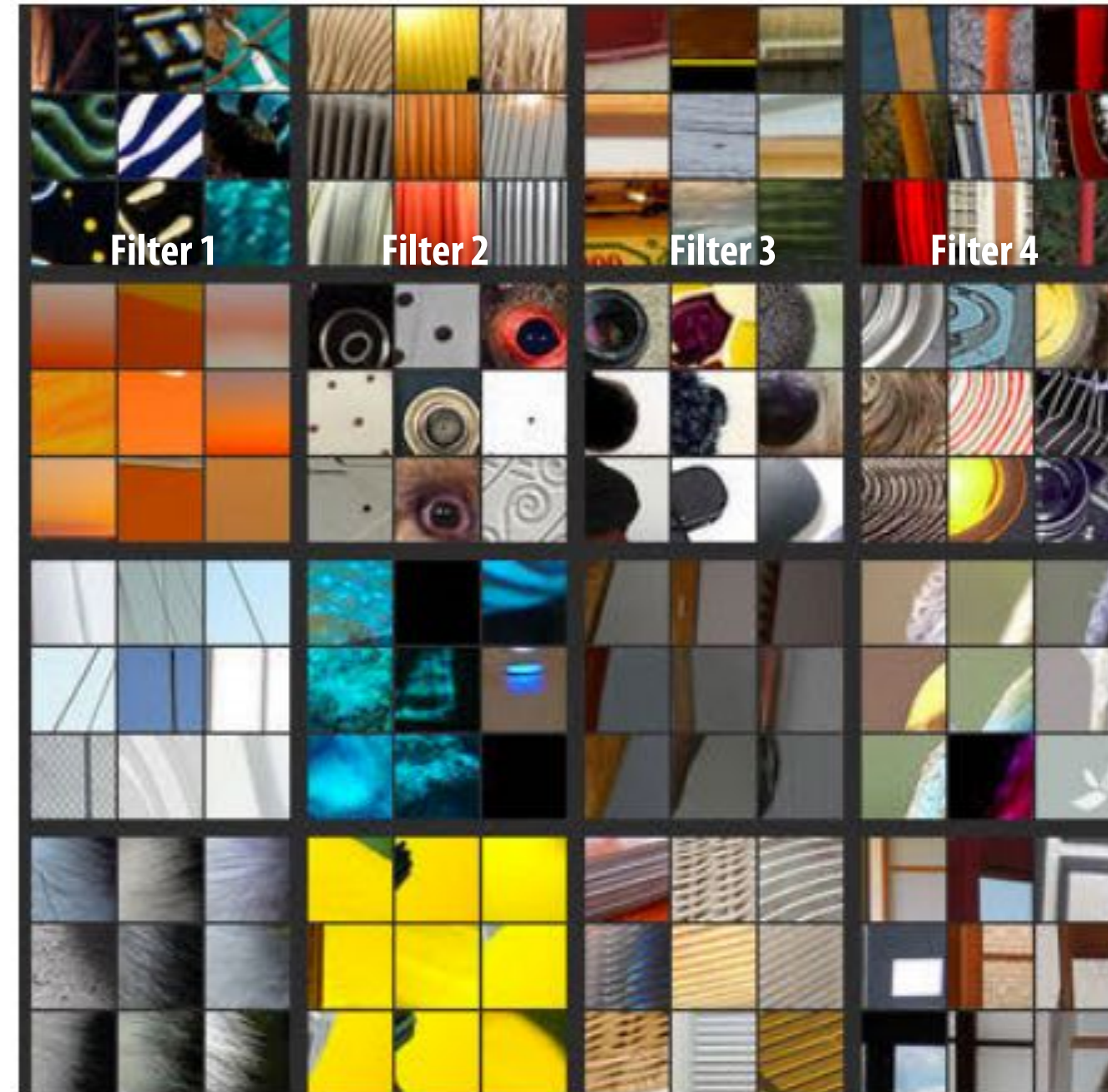


Going deeper

Layer 1



Layer 2

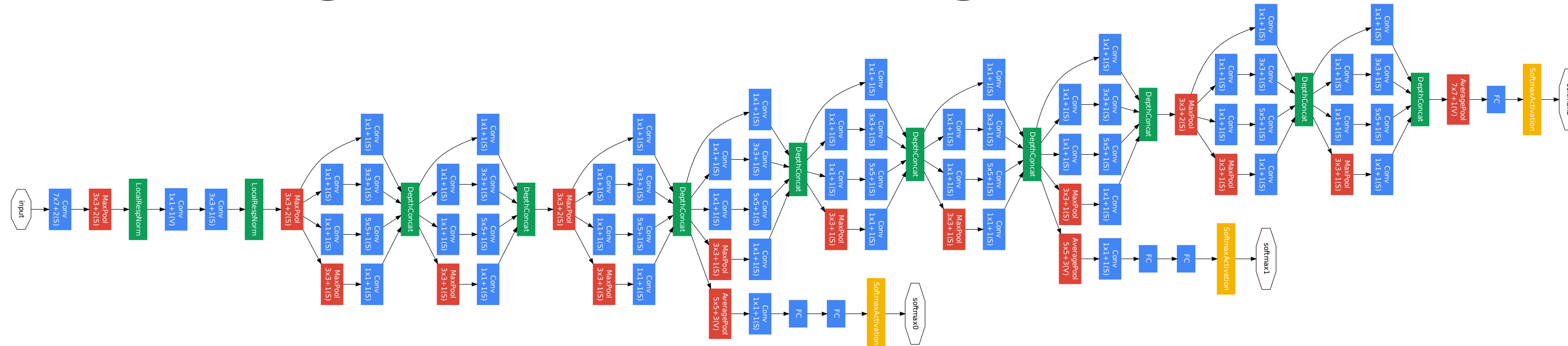


Layer 3

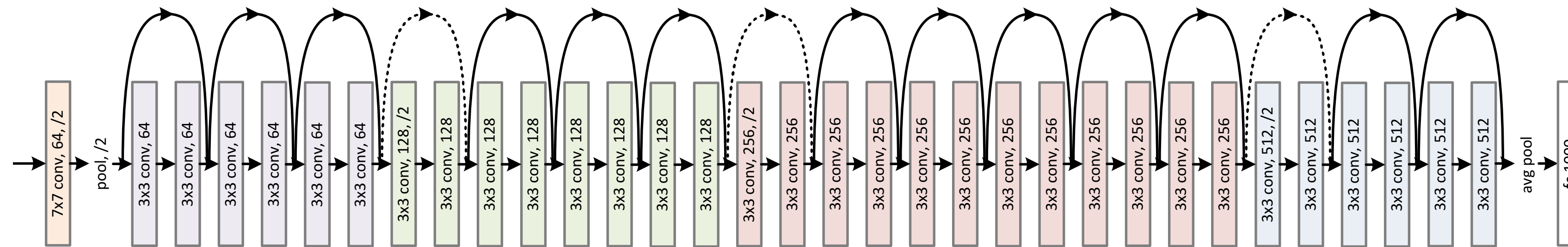


Visualization: images that generate strongest response for filters at each layer

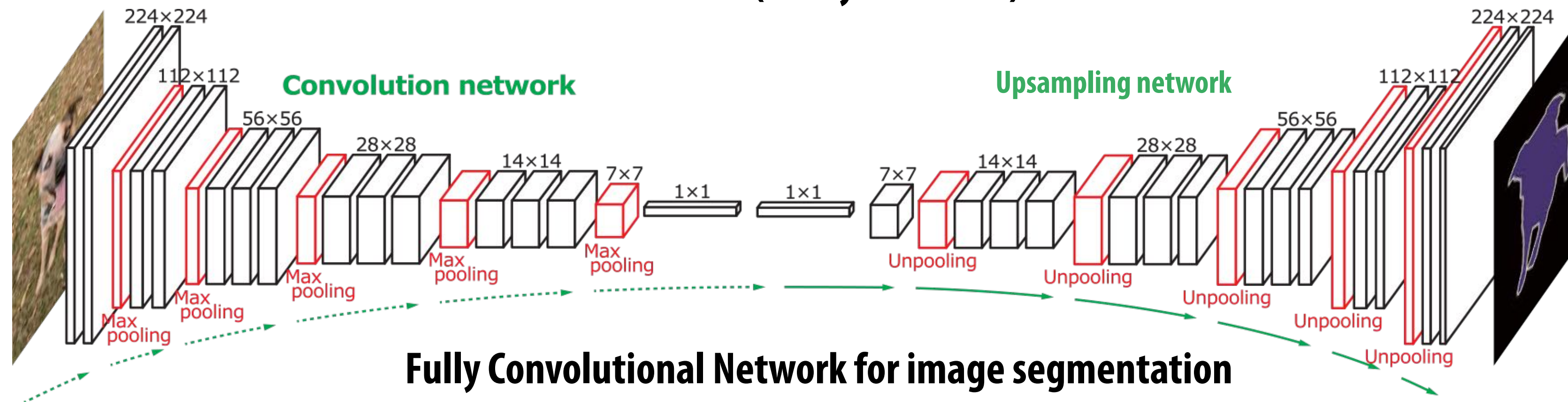
More recent image understanding networks



Inception (GoogleLeNet)



ResNet (34 layer version)



Fully Convolutional Network for image segmentation

Efficiently implementing convolution layers

Direct implementation of conv layer (batched)

```
float input[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH];           // input activations
float output[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS];     // output activations
float layer_weights[LAYER_NUM_FILTERS][LAYER_CONVY][LAYER_CONVX][INPUT_DEPTH];
float layer_biases[LAYER_NUM_FILTERS];

// 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 = layer_biases[LAYER_NUM_FILTERS];
                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;
            }
}
```

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

Convolutional layer in Halide

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

Func in_func;                    // assume input function (activations) 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;                  // z is num input channels, n is batch 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 of size f_w x f_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);
```

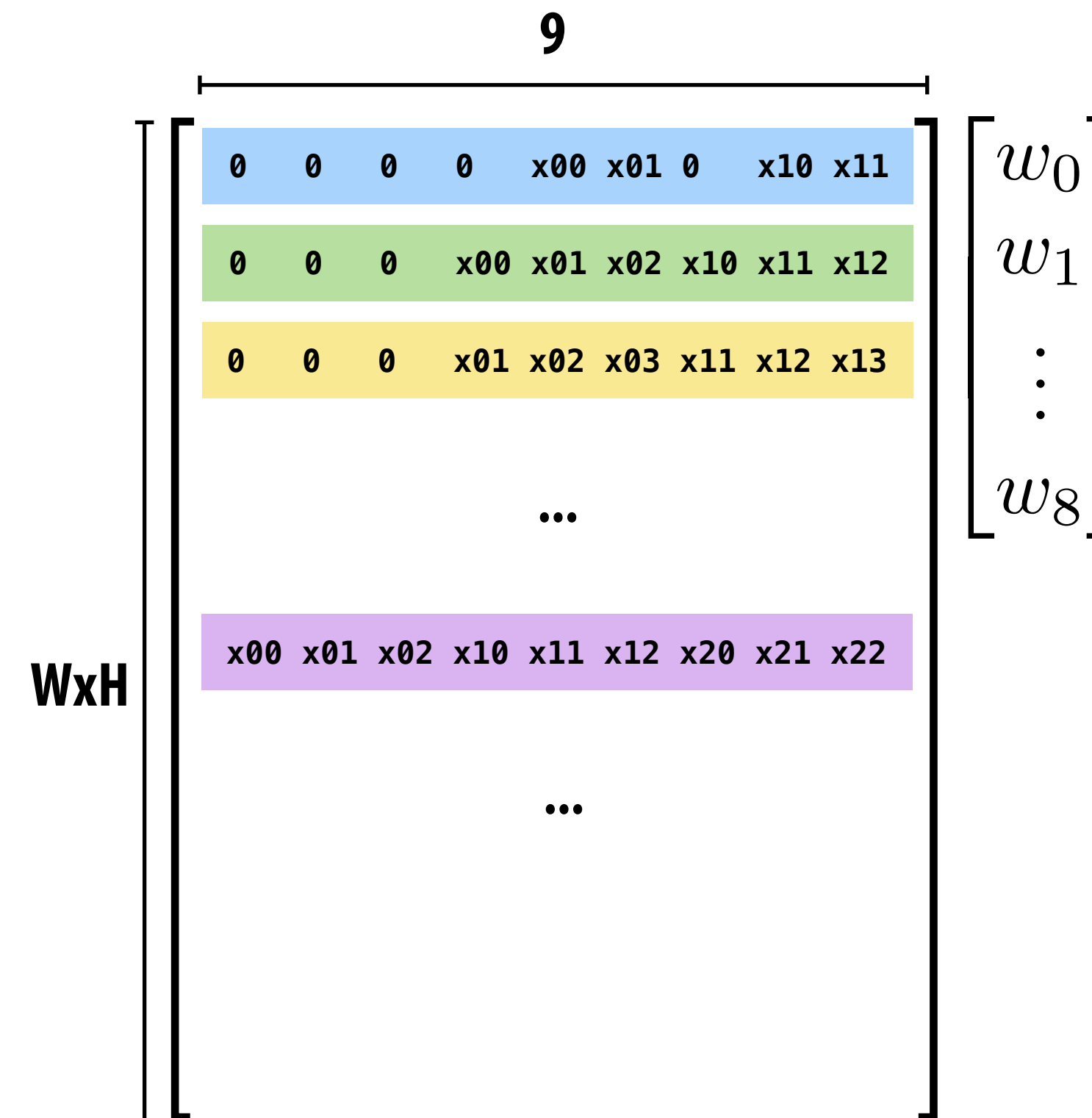
Consider scheduling this seven-dimensional loop nest!

3x3 convolution as matrix-vector product (“explicit gemm”)

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}	...			
				

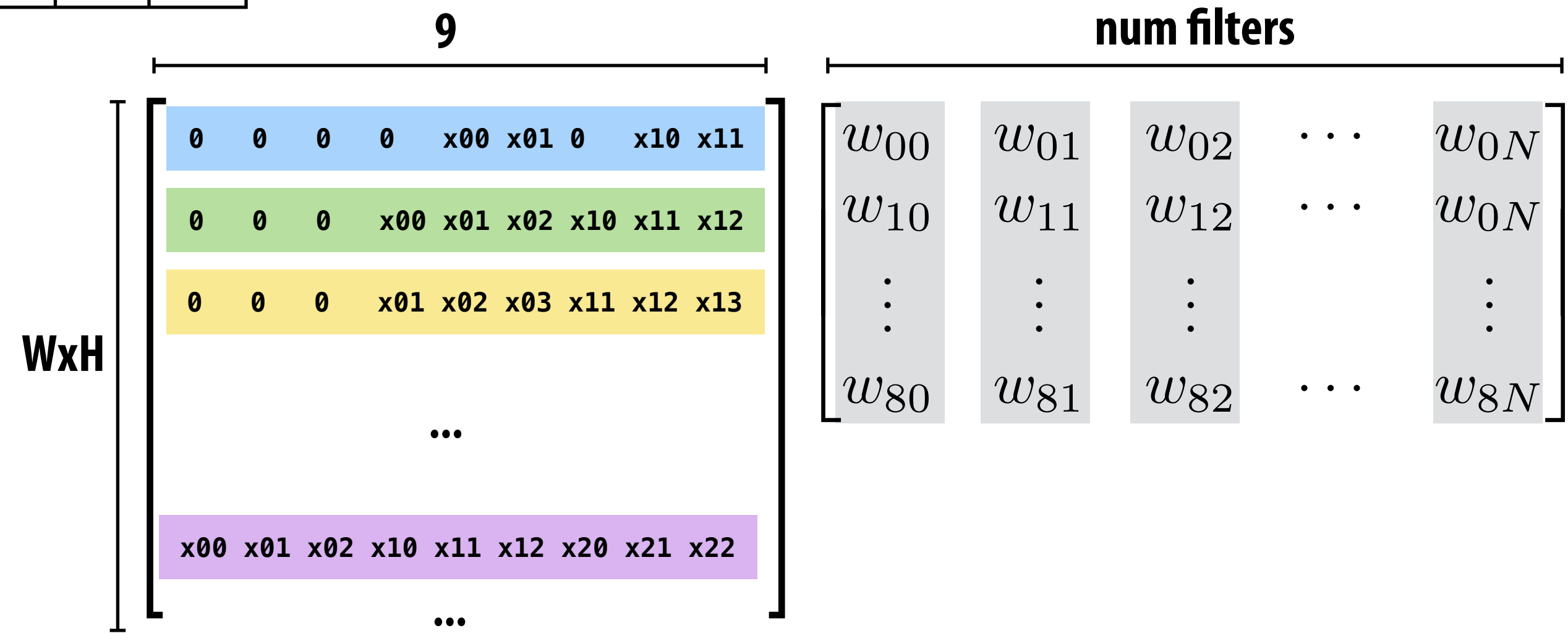
$O(N)$ storage overhead for filter with N elements
Must construct input data matrix



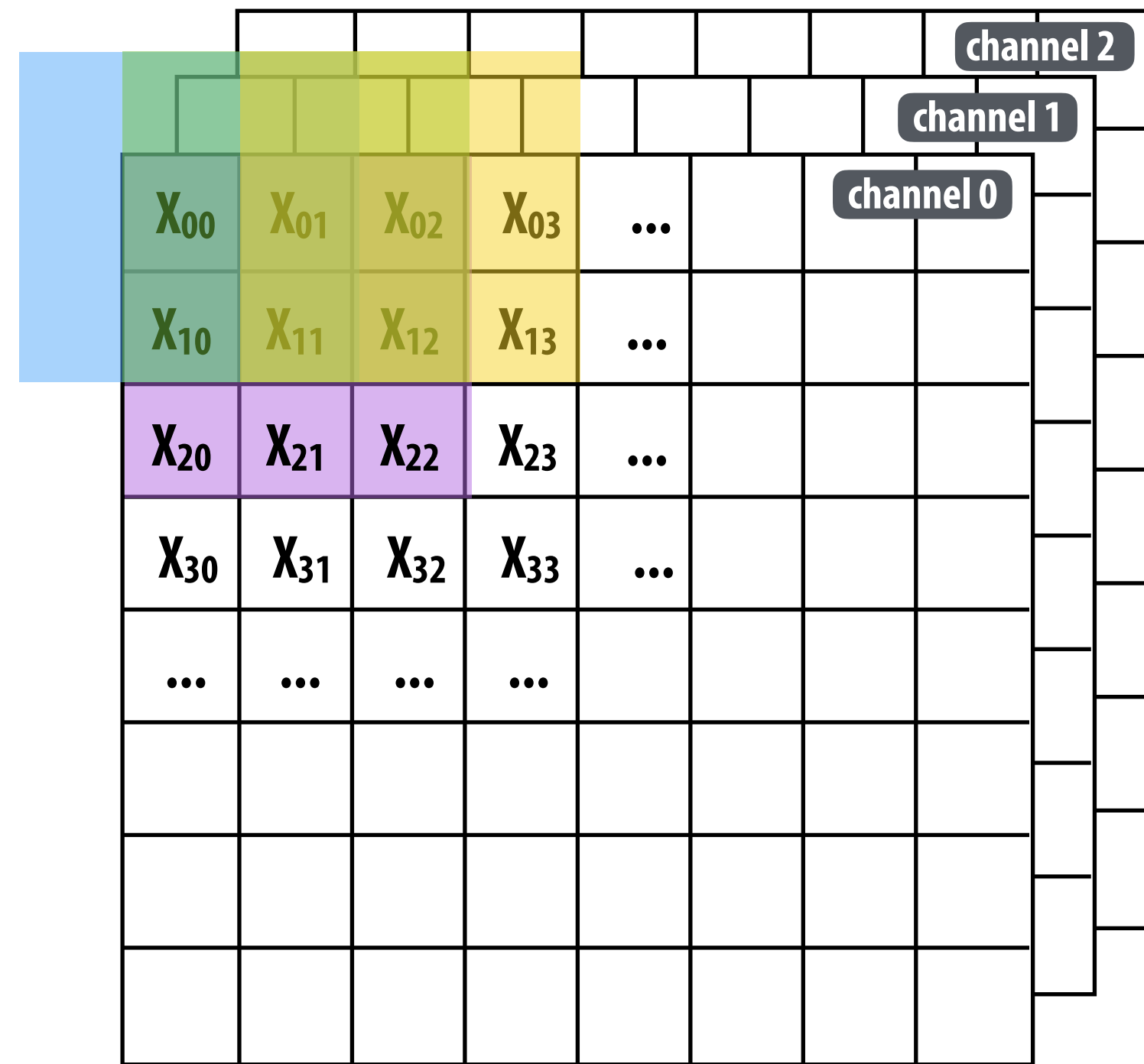
Note: 0-pad matrix

3x3 convolution as matrix-vector product (“explicit gemm”)

	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}	...			
				

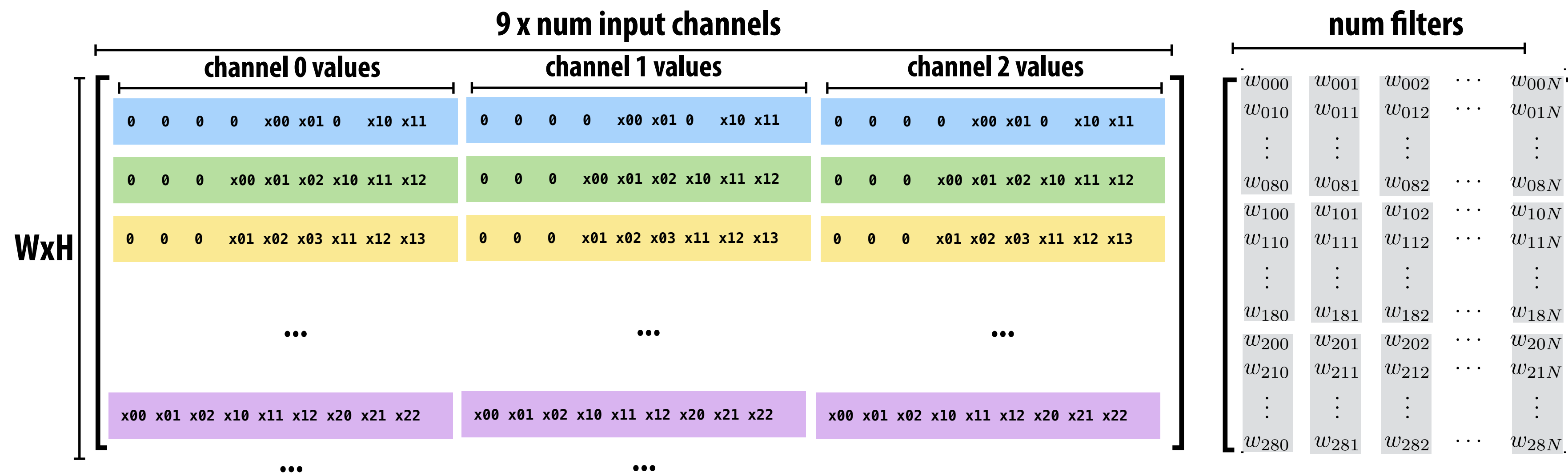


Multiple convolutions on multiple input channels



For each filter, sum responses over input channels

Equivalent to $(3 \times 3 \times \text{num_channels})$ convolution on $(W \times H \times \text{num_channels})$ input data



Conv layer to explicit GEMM mapping

The convolution operation on 4D tensors can be mapped as matrix-multiply operation on 2D matrices

Convolution		GEMM
$y = CONV(x, w)$		$C = GEMM(A, B)$
$x[N, H, W, C]$: 4D activation tensor	→	$A[NPQ, RSC]$: 2D convolution matrix
$w[K, R, S, C]$: 4D filter tensor	→	$B[RSC, K]$: 2D filter matrix
$y[N, P, Q, K]$: 4D output tensor	→	$C[NPQ, K]$: 2D output matrix

Symbol reference:
Spatial support of filters: $R \times S$
Input channels: C
Number of filters: K
Batch size: N

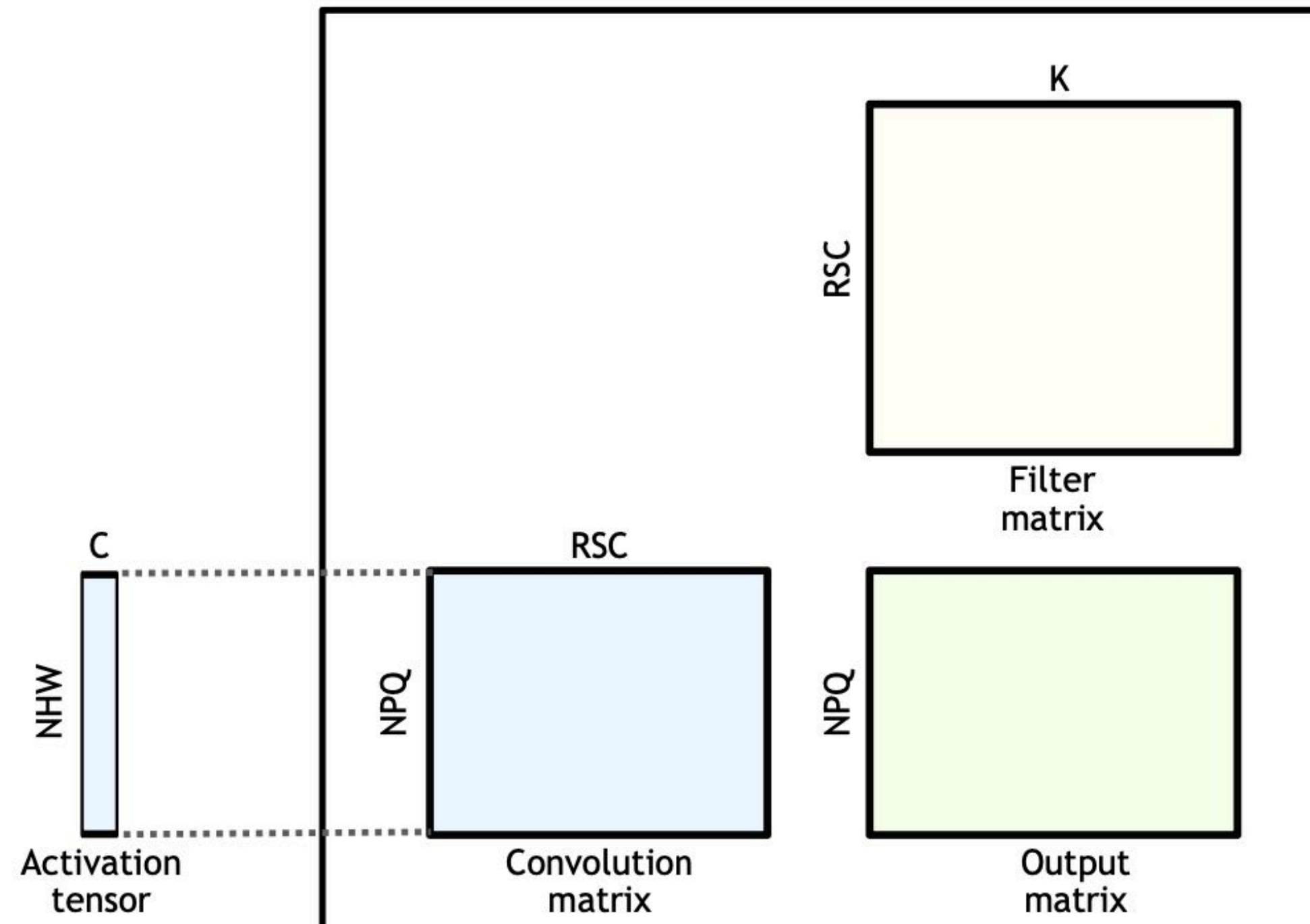


Image credit: NVIDIA

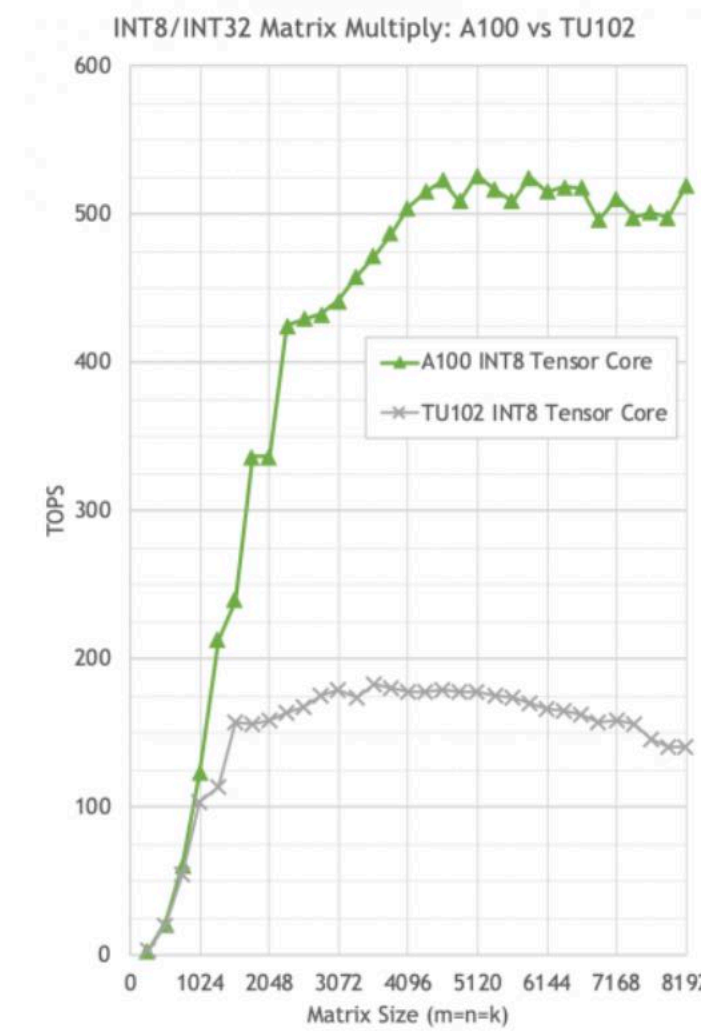
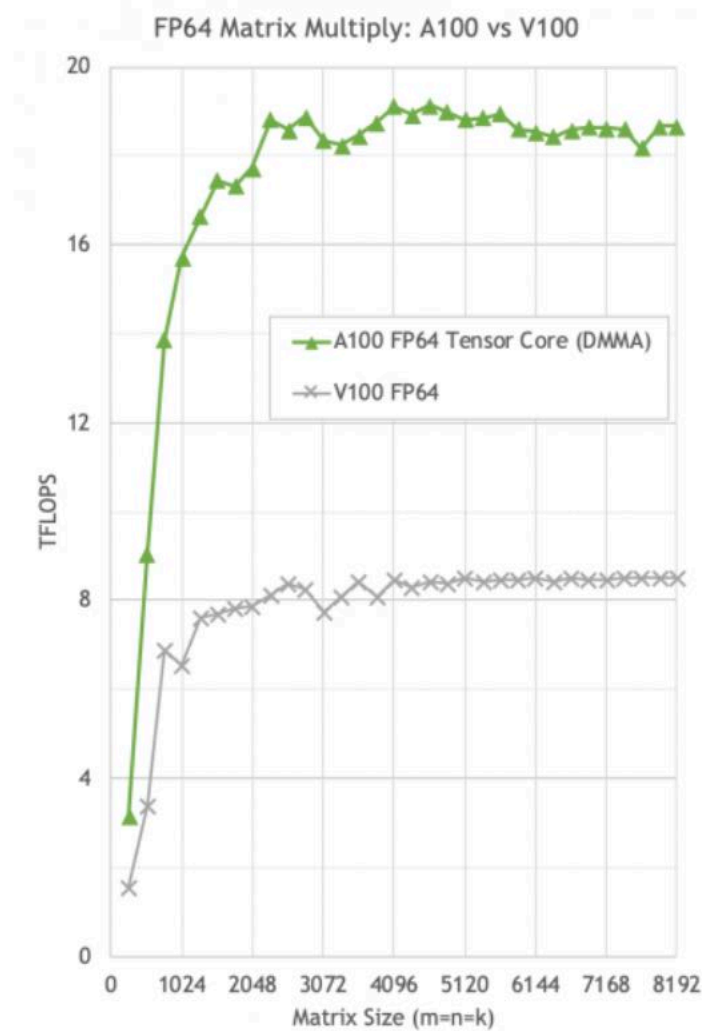
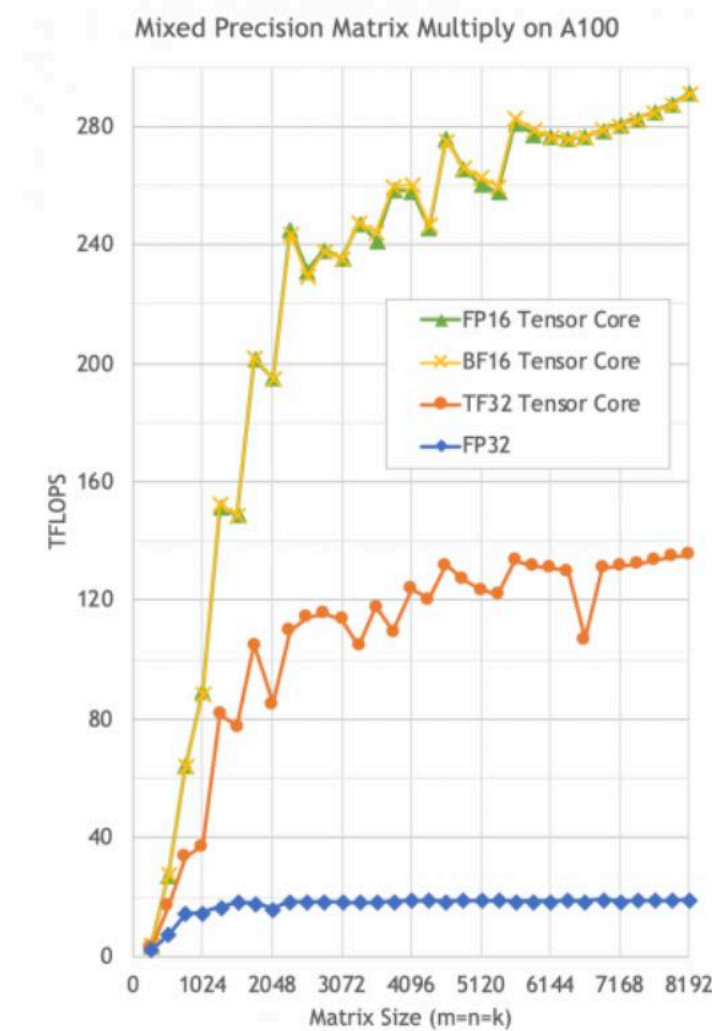
High performance implementations of GEMM exist

cuBLAS Performance

The cuBLAS library is highly optimized for performance on NVIDIA GPUs, and leverages tensor cores for acceleration of low and mixed precision matrix multiplication.

cuBLAS Key Features

- Complete support for all 152 standard BLAS routines
- Support for half-precision and integer matrix multiplication
- GEMM and GEMM extensions optimized for Volta and Turing Tensor Cores
- GEMM performance tuned for sizes used in various Deep Learning models
- Supports CUDA streams for concurrent operations



To use “off the shelf” libraries, must materialize input matrices.

Increases DRAM traffic by a factor of $R \times S$
(To read input data from activation tensor and constitute “convolution matrix”)

Also requires large amount of aux storage

Intel® oneAPI Math Kernel Library

Intel®-Optimized Math Library for Numerical Computing

Optimized Library for Scientific Computing

- Enhanced math routines enable developers and data scientists to create performant science, engineering, or financial applications
- Core functions include BLAS, LAPACK, sparse solvers, fast Fourier transforms (FFT), random number generator functions (RNG), summary statistics, data fitting, and vector math
- Optimizes applications for current and future generations of Intel® CPUs, GPUs, and other accelerators
- Is a seamless upgrade for previous users of the Intel® Math Kernel Library (Intel® MKL)

Download as Part of the Toolkit

oneMKL is included in the Intel oneAPI Base Toolkit, which is a core set of tools and libraries for developing high-performance, data-centric applications across diverse architectures.

[Get It Now →](#)

Dense matrix multiplication

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

```
// compute C += A * B
```

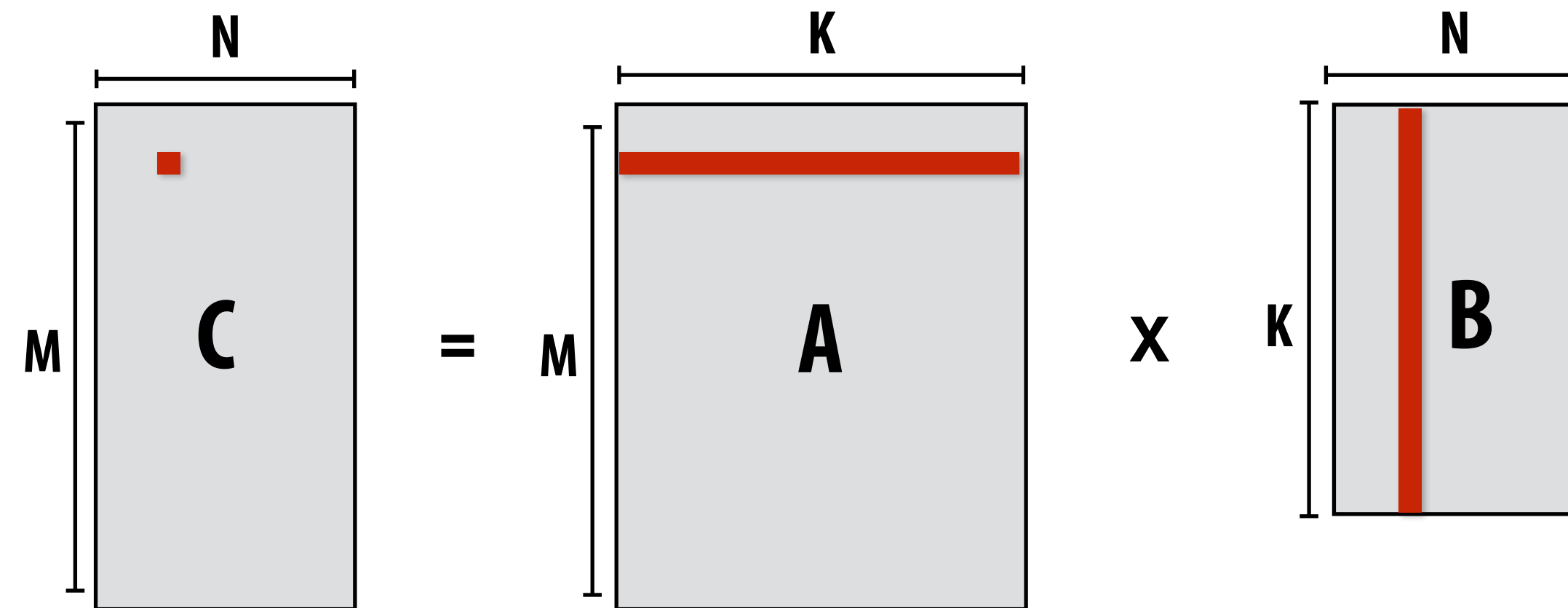
```
#pragma omp parallel for
```

```
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)
```

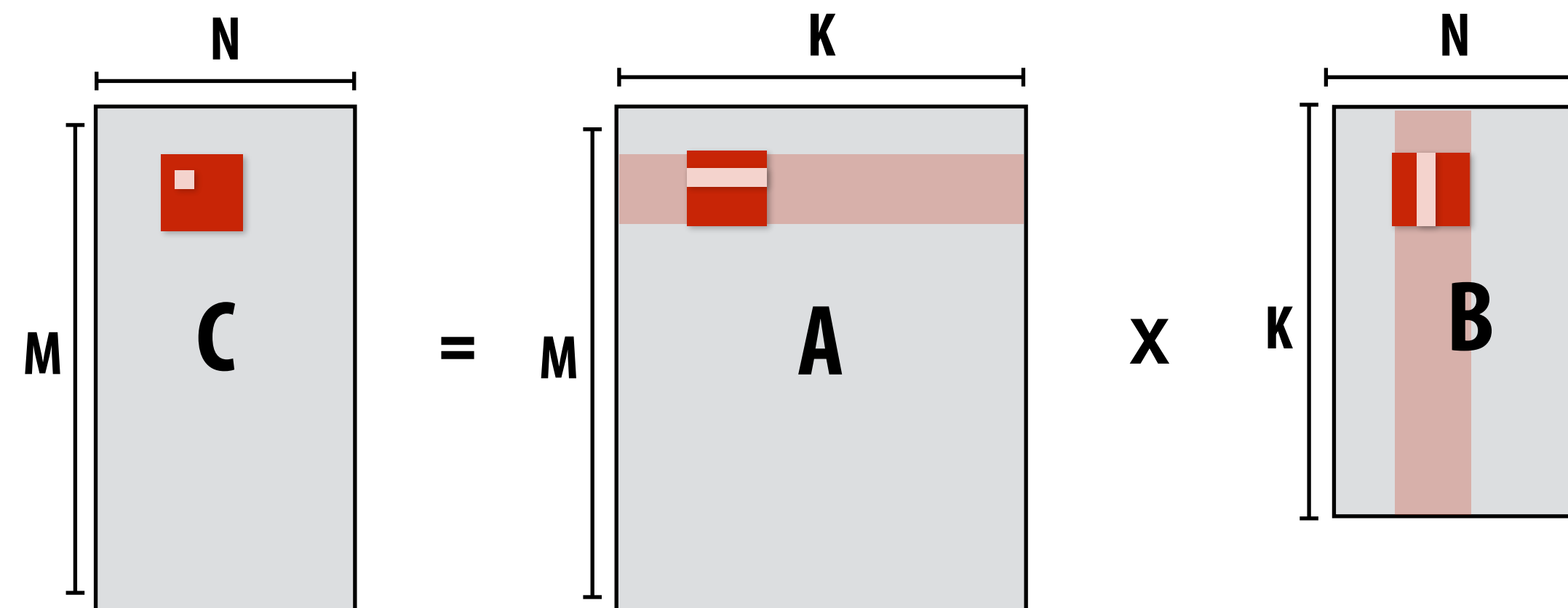
```
    for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
```

```
      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

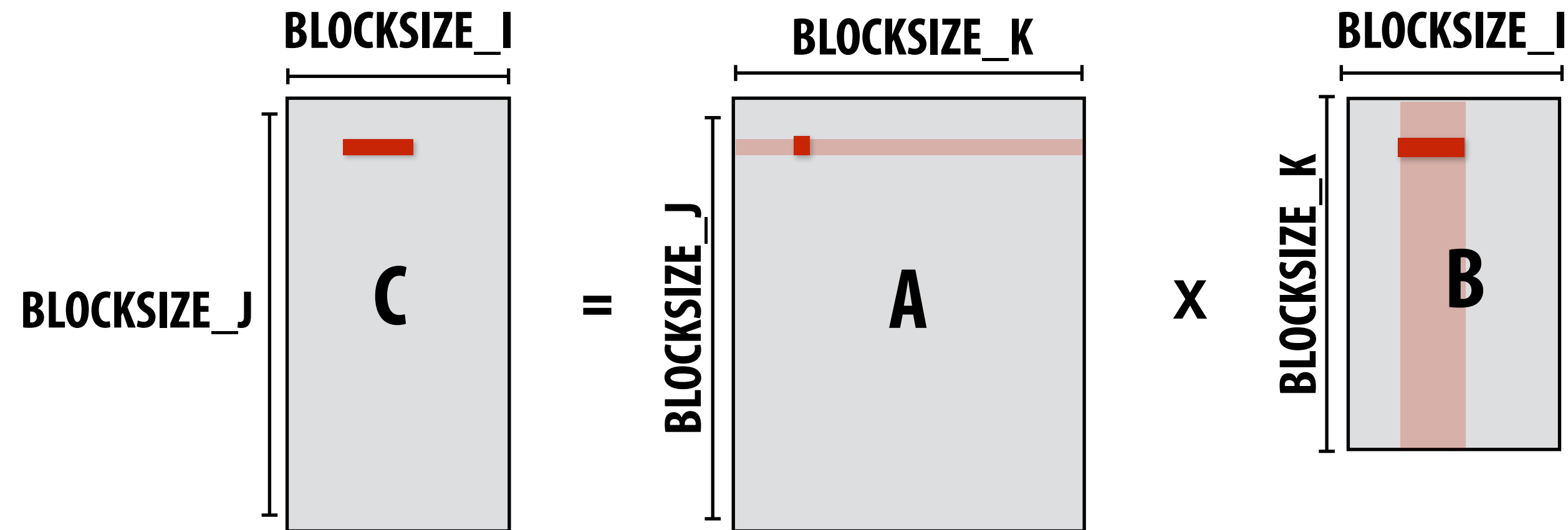
```
float A[M][K];
float B[K][N];
float C[M][N];

// 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



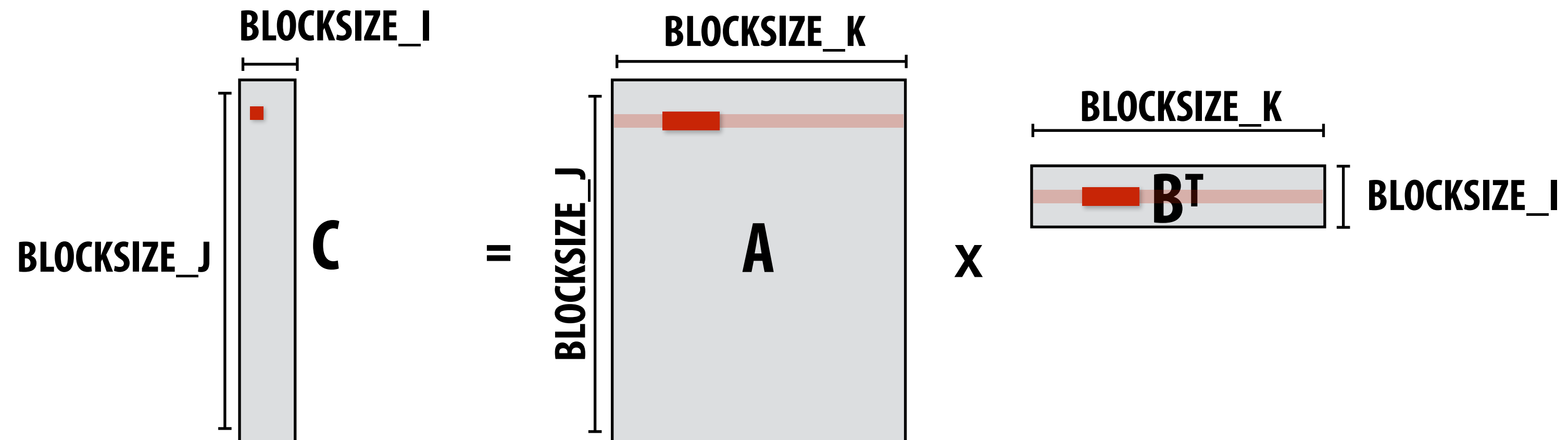
```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
    for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
        simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
        for (int k=0; k<BLOCKSIZE_K; k++) {
            // C = A*B + C
            simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single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

Different layers of a single DNN may benefit from unique scheduling strategies (different matrix dimensions)

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

Ug for library implementers!

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	$1 \times 1 \times 1000$	

Optimization: do not materialize full matrix ("implicit gemm")

This is a naive implementation that does not perform blocking, but indexes into input weight and activation tensors.

Symbol reference:

Spatial support of filters: $R \times S$

Input channels: C

Number of filters: K

Batch size: N

Image credit: NVIDIA

GEMM TRIPLE NEST LOOP

```
int GEMM_M = N * P * Q;
int GEMM_N = K;
int GEMM_K = R * S * C;

for (int gemm_m = 0; gemm_m < GEMM_M; ++gemm_m) {
  for (int gemm_n = 0; gemm_n < GEMM_N; ++gemm_n) {

    int n = gemm_m / (PQ);
    int npq_residual = gemm_m % (PQ);
    int p = npq_residual / Q;
    int q = npq_residual % Q;

    Accumulator accum = 0;
    for (int gemm_k = 0; gemm_k < GEMM_K; ++gemm_k) {

      int k = gemm_n;
      int crs_residual = gemm_k / C;
      int r = crs_residual / S;
      int s = crs_residual % S;
      int c = gemm_k % C;

      int h = h_bar(p, r);
      int w = w_bar(q, s);

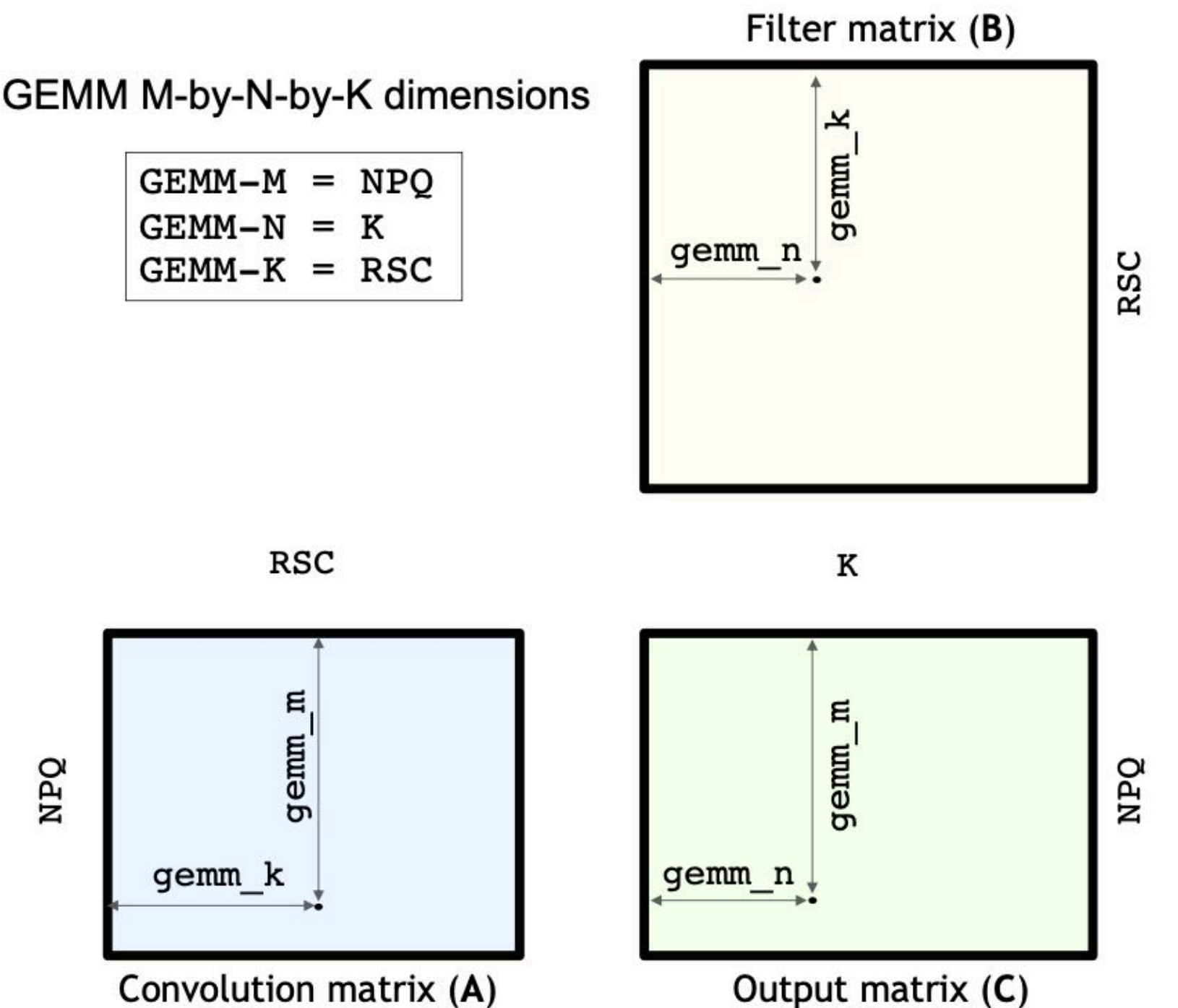
      ElementA a = activation_tensor.at({n, h, w, c});
      ElementB b = filter_tensor.at({k, r, s, c});
      accum += a * b;
    }

    C[gemm_m * K + gemm_n] = accum;
  }
}
```

CONVOLUTION MAPPED TO GEMM

GEMM M-by-N-by-K dimensions

GEMM-M = NPQ
GEMM-N = K
GEMM-K = RSC



$$C[\text{gemm}_m, \text{gemm}_n] = \sum_{\text{gemm}_k=0}^{RSC-1} (A[\text{gemm}_m, \text{gemm}_k] * B[\text{gemm}_k, \text{gemm}_n])$$

Optimization: do not materialize full matrix (“implicit gemm”)

**Better implementation:
materialize only a sub-block of the
convolution matrix at a time in
GPU on-chip “shared memory”**

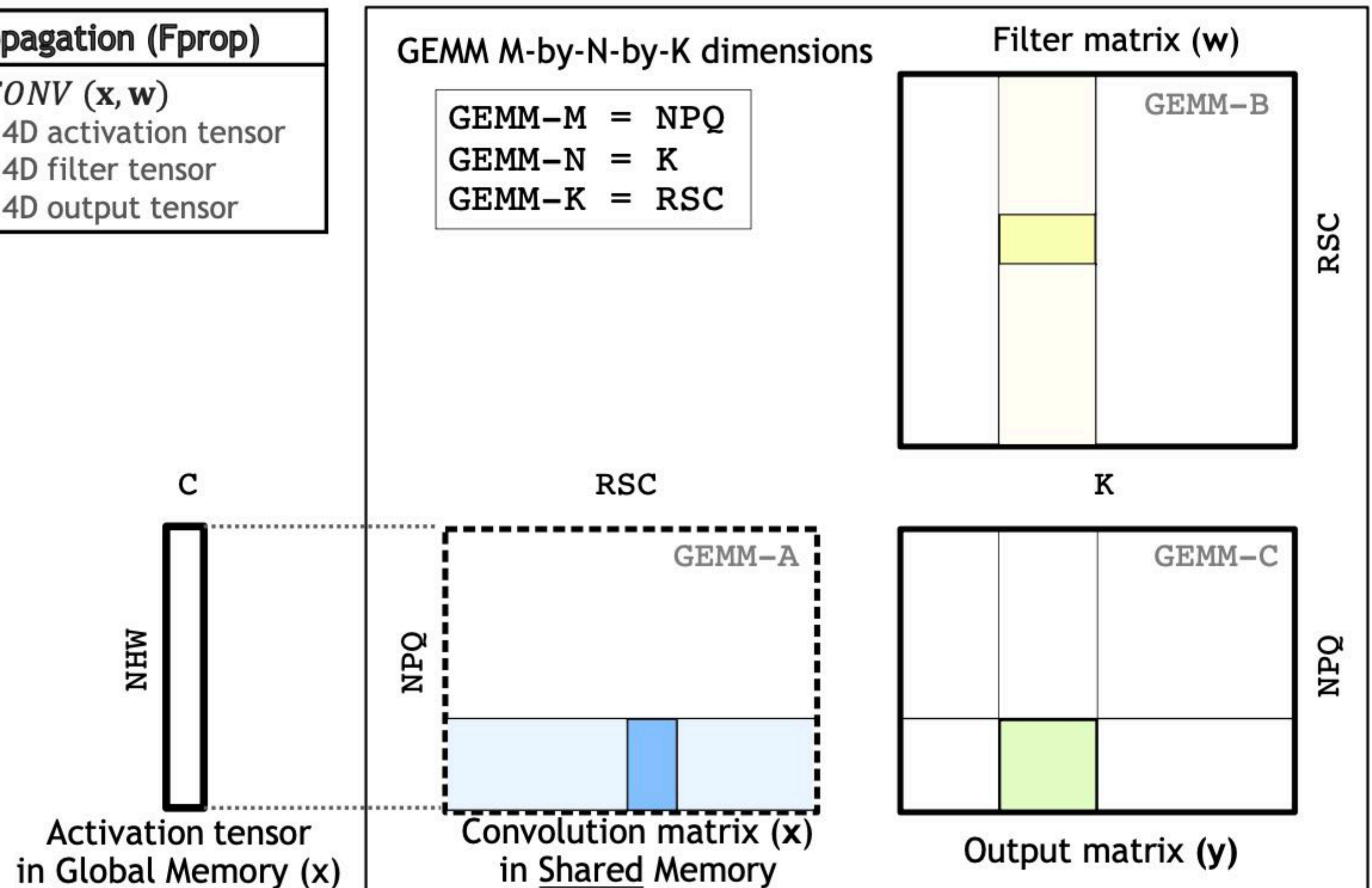
Forward Propagation (Fprop)	
$y = CONV(x, w)$	
$x[N, H, W, C]$: 4D activation tensor
$w[K, R, S, C]$: 4D filter tensor
$y[N, P, Q, K]$: 4D output tensor

**Does not require additional off-chip storage and
does not increase required DRAM traffic.**

**Use well-tuned shared-memory based GEMM
routines to perform sub-block GEMM (see CUTLASS)**

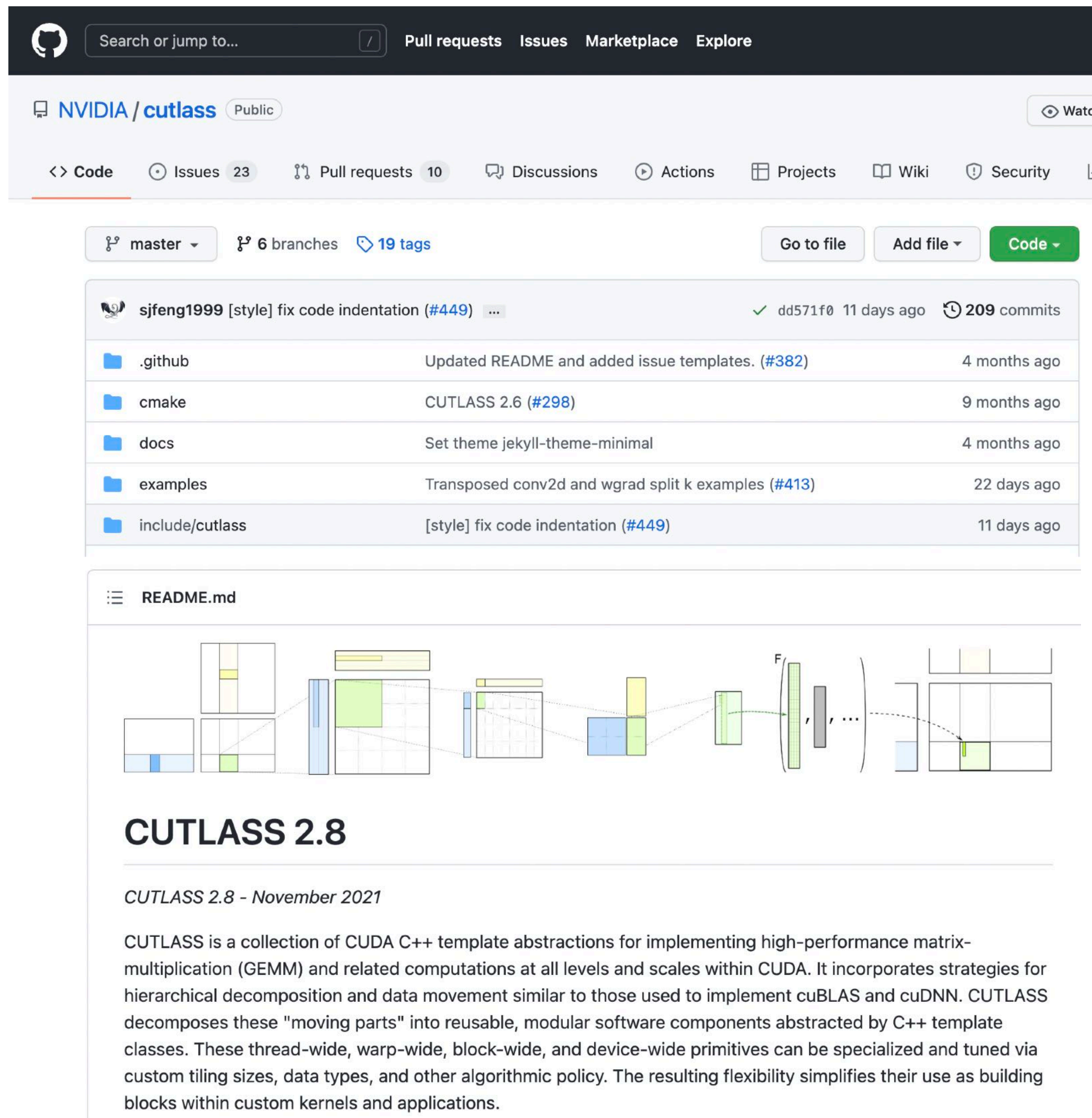
**Symbol reference:
Spatial support of filters: $R \times S$
Input channels: C
Number of filters: K
Batch size: N**

Image credit: NVIDIA



NVIDIA CUTLASS

Basic primitives/building block for implementing your custom high performance DNN layers. (e.g, unusual sizes that haven't been heavily tuned by cuDNN)



The screenshot shows the GitHub repository for NVIDIA CUTLASS. The repository is public and has 23 issues, 10 pull requests, and 209 commits. The README.md file is open, showing a diagram of the CUTLASS architecture and the title "CUTLASS 2.8".

CUTLASS 2.8

CUTLASS 2.8 - November 2021

CUTLASS is a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) and related computations at all levels and scales within CUDA. It incorporates strategies for hierarchical decomposition and data movement similar to those used to implement cuBLAS and cuDNN. CUTLASS decomposes these "moving parts" into reusable, modular software components abstracted by C++ template classes. These thread-wide, warp-wide, block-wide, and device-wide primitives can be specialized and tuned via custom tiling sizes, data types, and other algorithmic policy. The resulting flexibility simplifies their use as building blocks within custom kernels and applications.

Fast (in-shared memory) GEMM

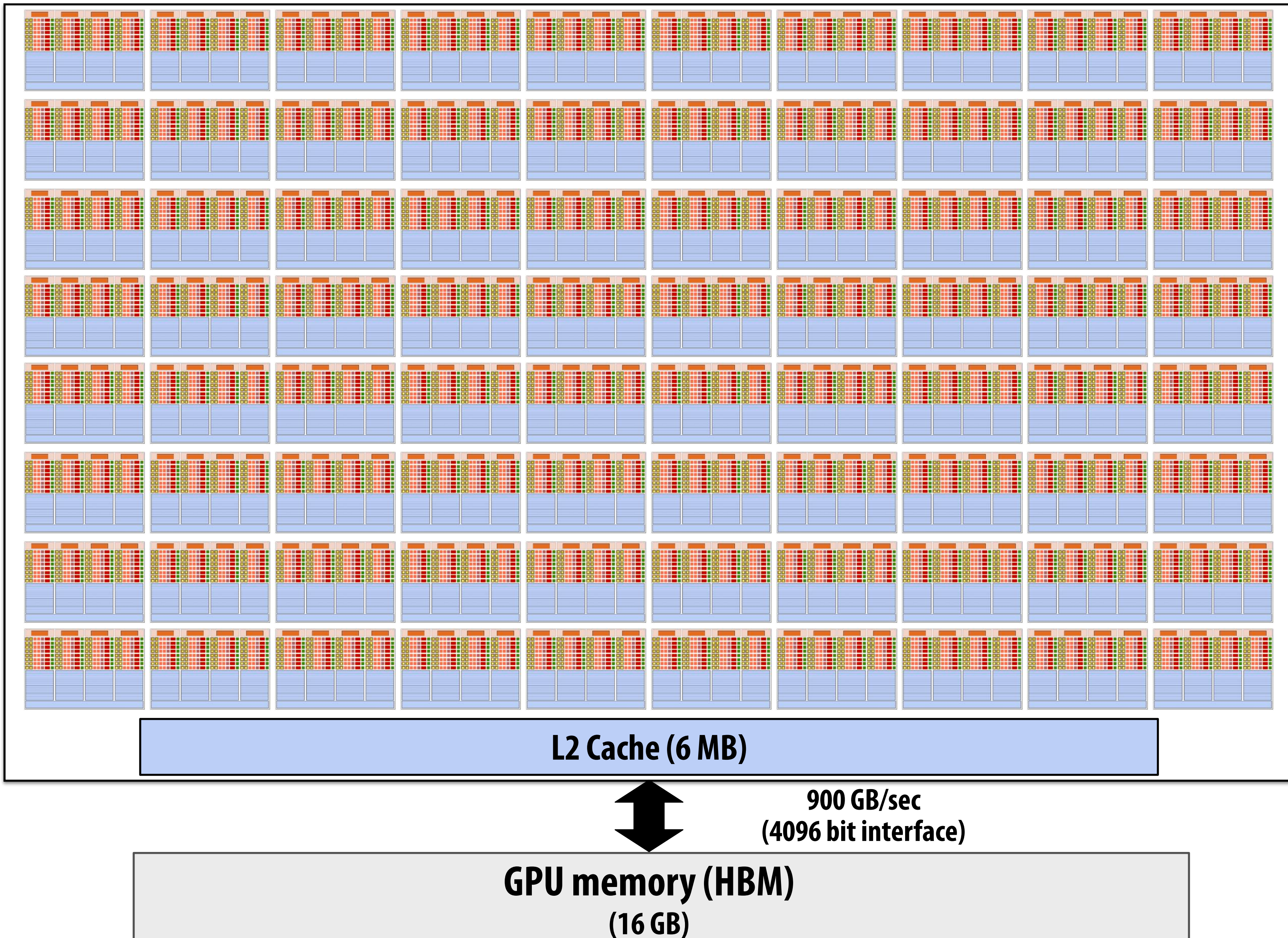
Fast WARP level GEMMs

Iterators for fast block loading/tensor indexing

Tensor reductions

Etc.

Recall: NVIDIA V100 GPU (80 SMs)



Many processing units and many tensor cores.

Need “a lot of parallel work” to fill the machine.

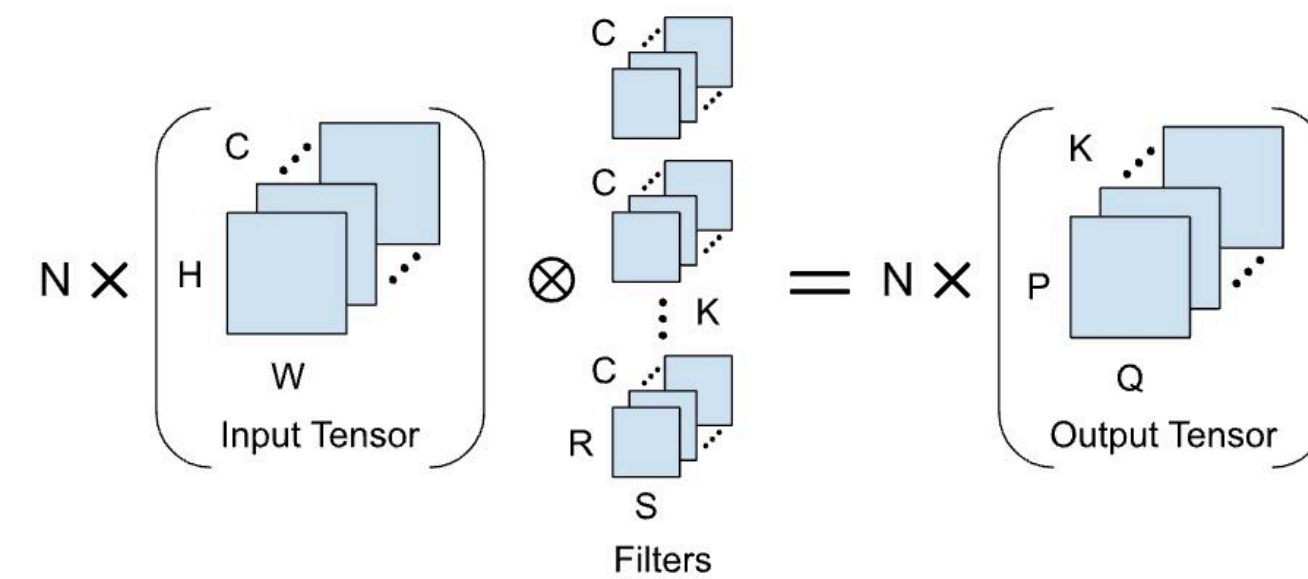
Higher performance with “more work”

N=1, P=Q=64 case:

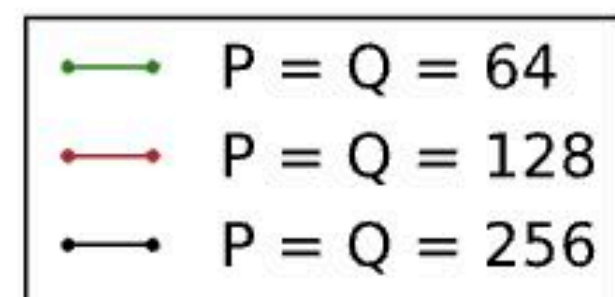
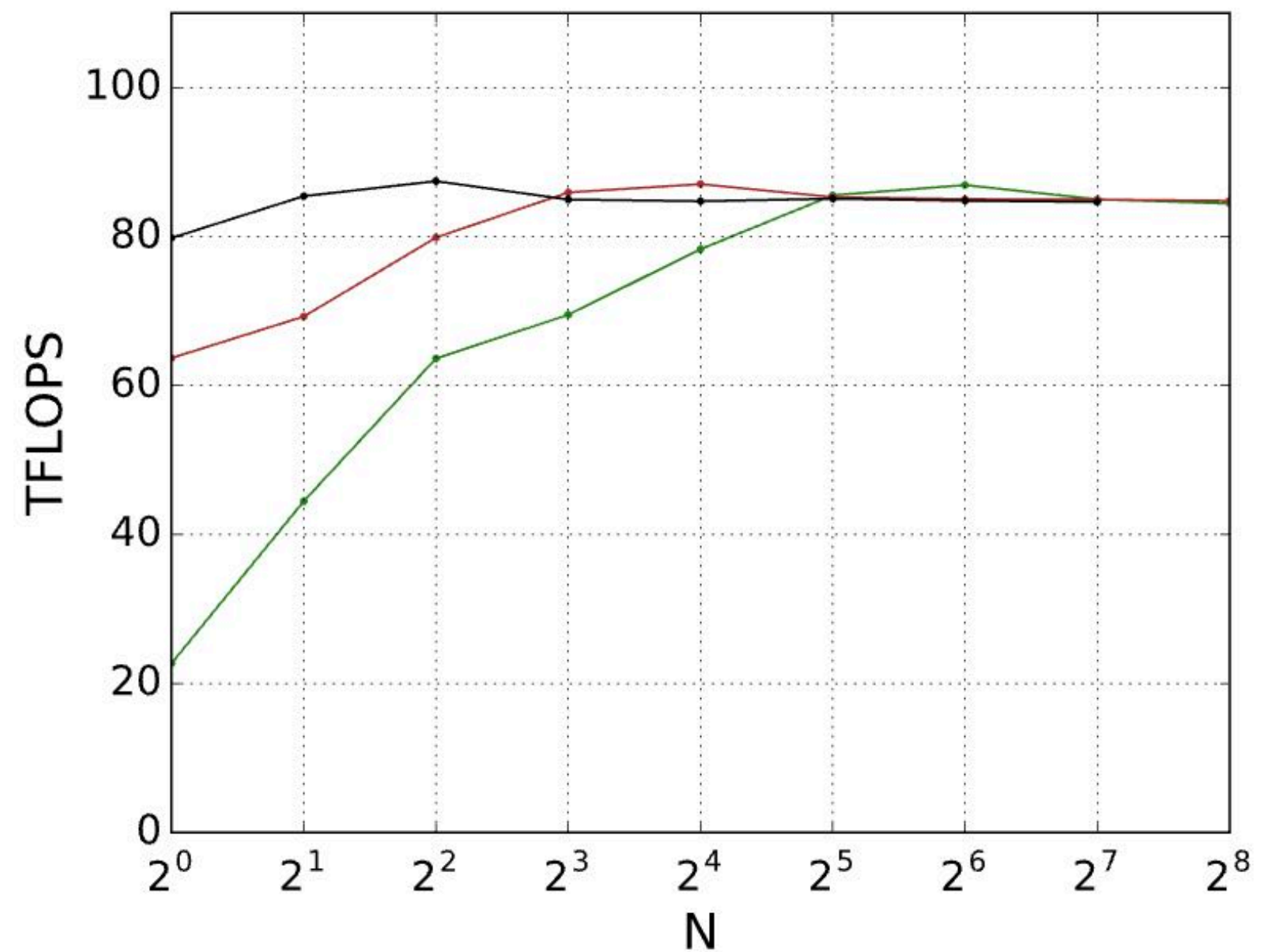
64 x 64 x 128 x 1 = 524K outputs = 2 MB of output data (float32)

N=32, P=Q=256 case:

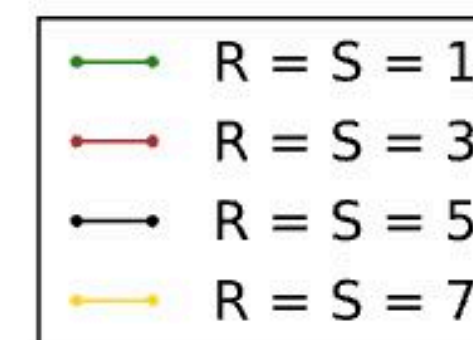
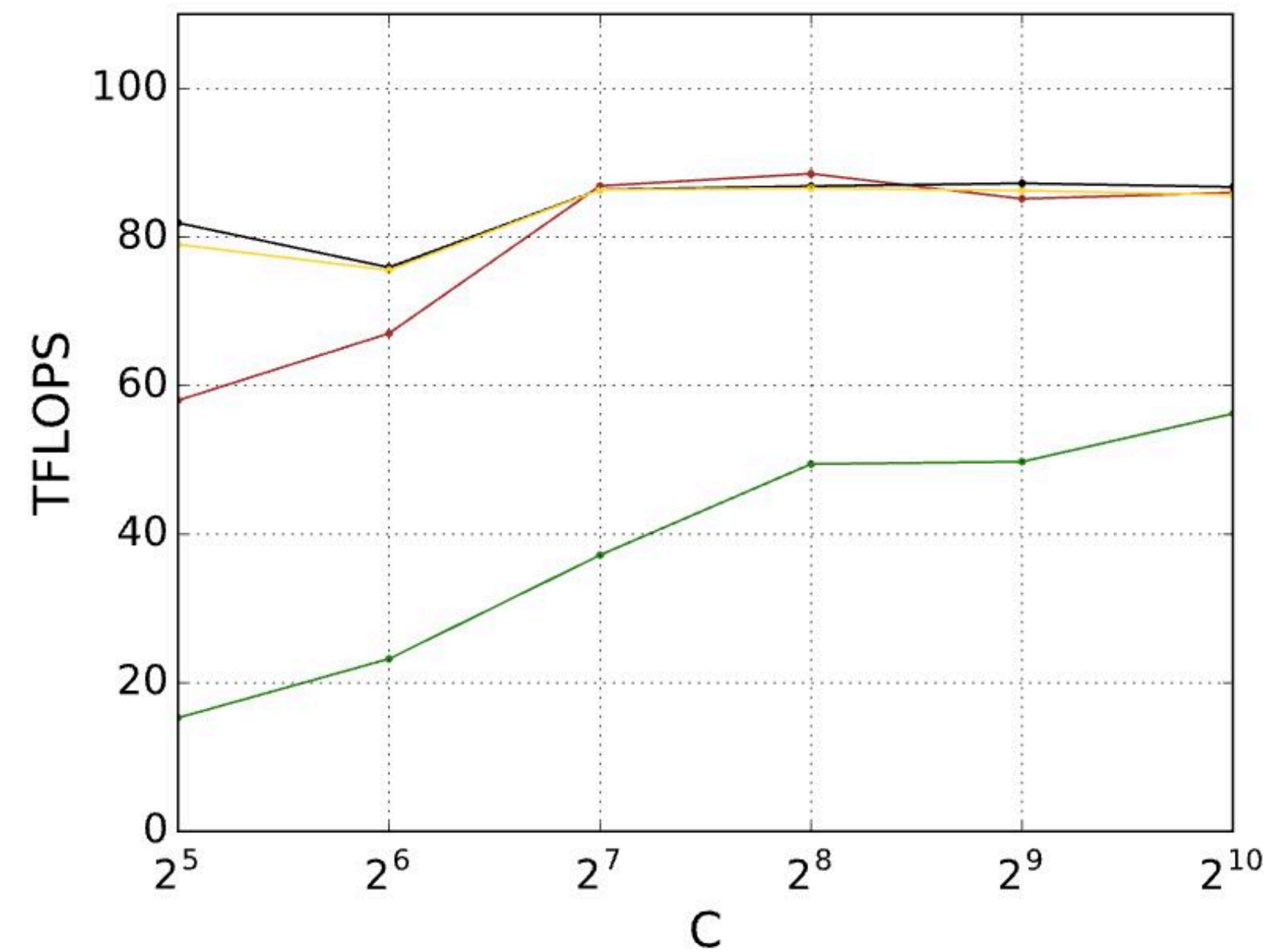
256 x 256 x 128 x 32 = 256M outputs = 1 GB of output data (float32)



Performance of Forward Convolution with
C = 128, K = 128, R = S = 3



Performance of Forward Convolution with
H = W = 256, K = 256, N = 1



Direct implementation of conv layer

```
float input[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][INPUT_DEPTH]; // input activations
float output[IMAGE_BATCH_SIZE][INPUT_HEIGHT][INPUT_WIDTH][LAYER_NUM_FILTERS]; // output activations
float layer_weights[LAYER_NUM_FILTERS][LAYER_CONVY][LAYER_CONVX][INPUT_DEPTH];
float layer_biases[LAYER_NUM_FILTERS];

// 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 = layer_biases[LAYER_NUM_FILTERS];
                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;
            }
```

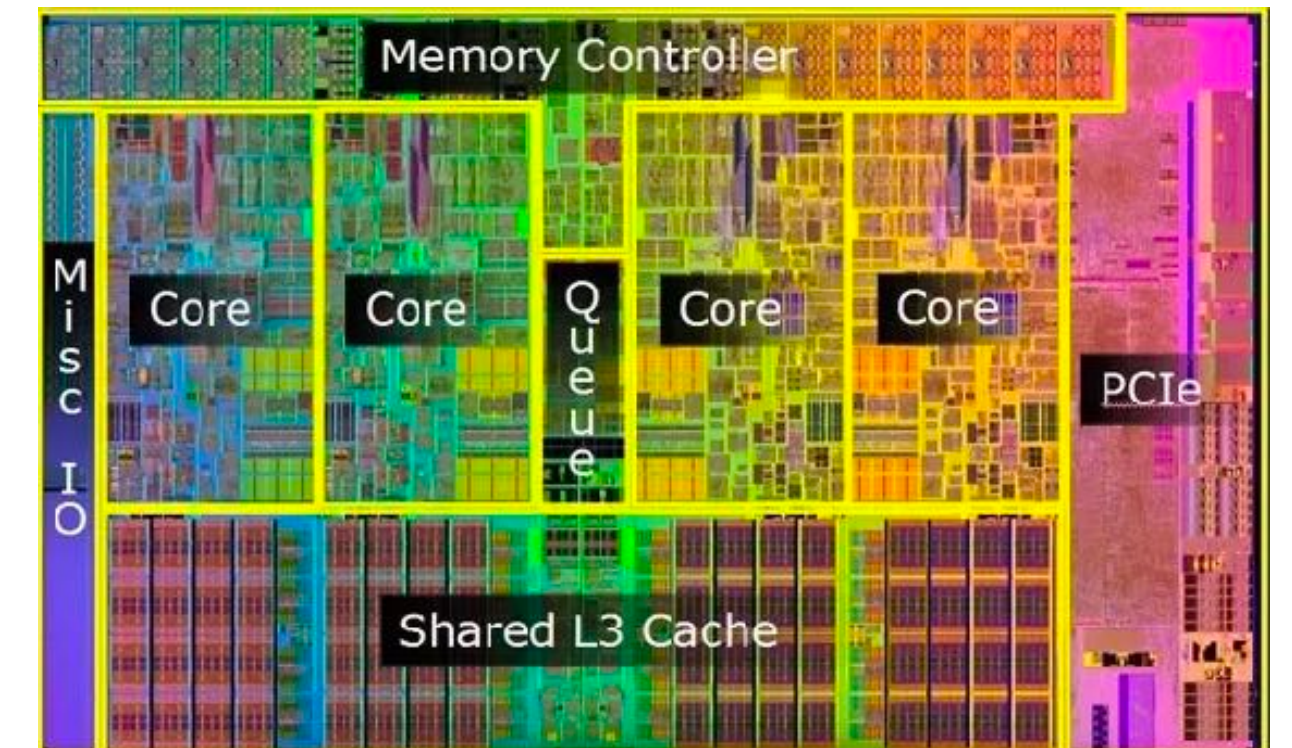
Or you can just directly implement this loop nest directly yourself.

Low-level vendor libraries offer high-performance implementations of key DNN layers

NVIDIA cuDNN



Intel® oneAPI Deep Neural Network Library



Example: CUDNN convolution

```
cudaStatus_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

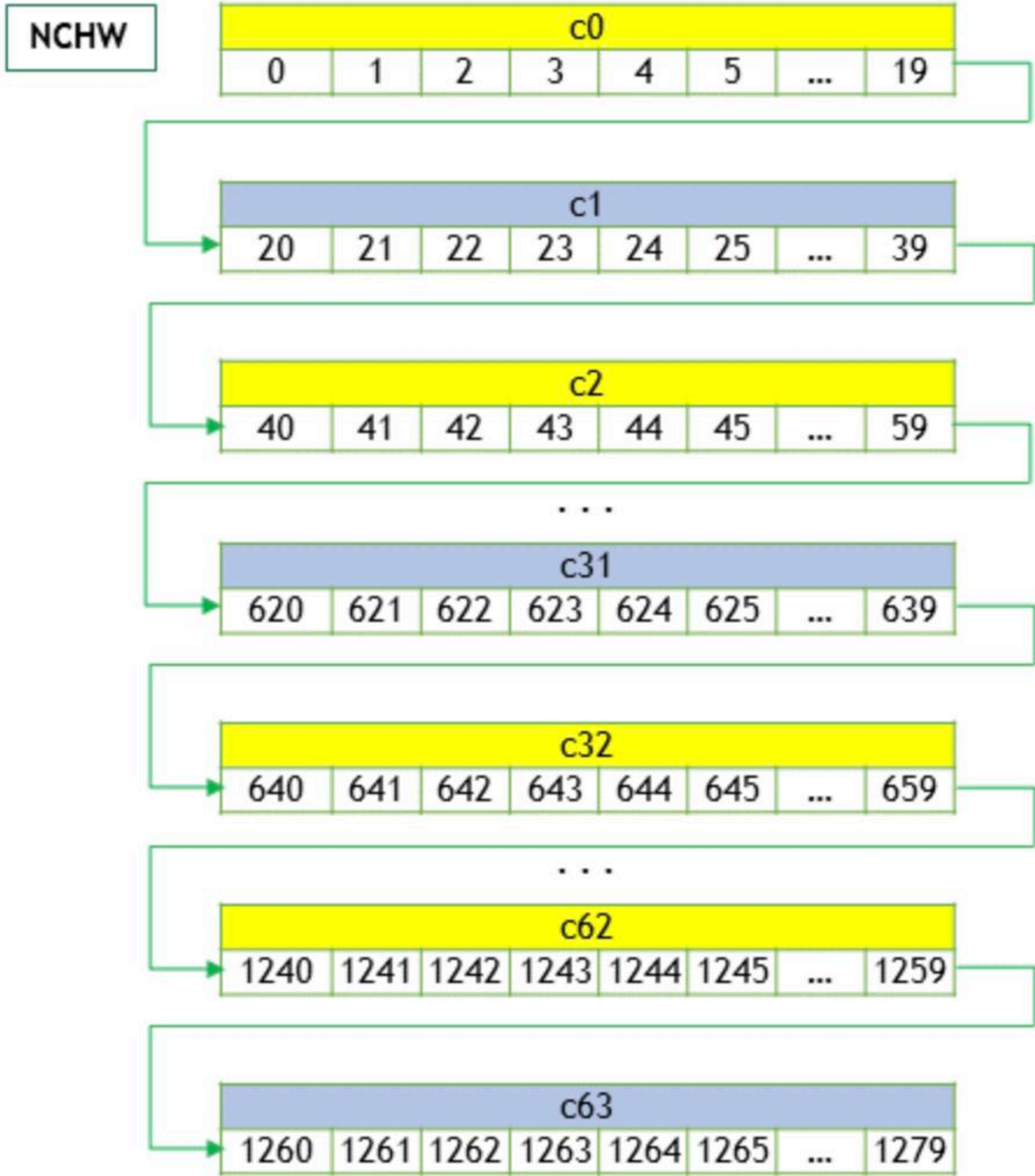
This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.

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 tensor data layout

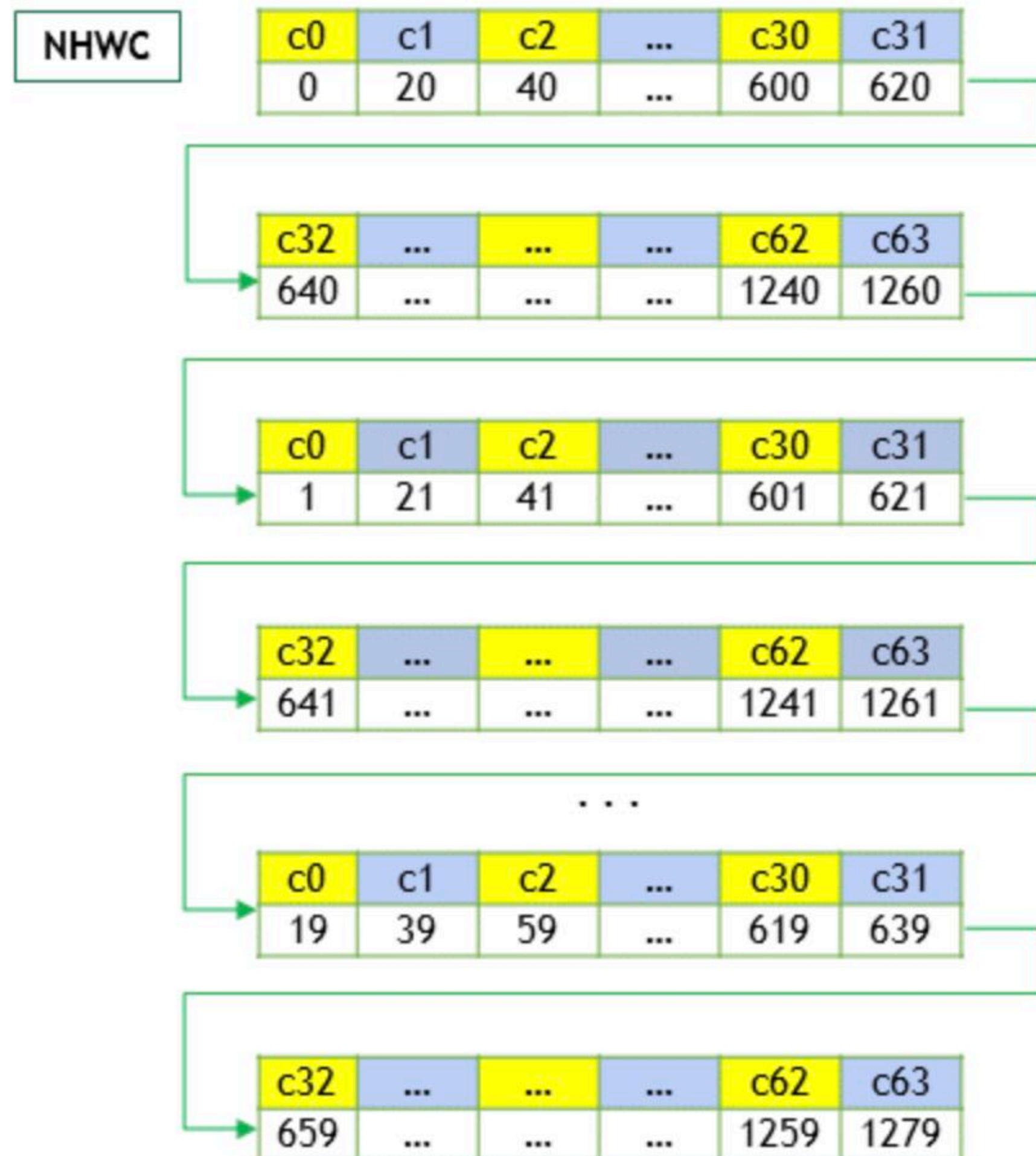
- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., number of channels); 64.
- **H** is the image height; 5.
- **W** is the image width; 4.



c = 0				c = 1				c = 2			
0	1	2	3	20	21	22	23	40	41	42	43
4	5	6	7	24	25	26	27	44	45	46	47
8	9	10	11	28	29	30	31	48	49	50	51
12	13	14	15	32	33	34	35	52	53	54	55
16	17	18	19	36	37	38	39	56	57	58	59
...											
				c = 62				c = 63			
				1240	1241	1242	1243	1260	1261	1262	1263
				1244	1245	1246	1247	1264	1265	1266	1267
				1248	1249	1250	1251	1268	1269	1270	1271
				1252	1253	1254	1255	1272	1273	1274	1275
				1256	1257	1258	1259	1276	1277	1278	1279

NHWC tensor 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

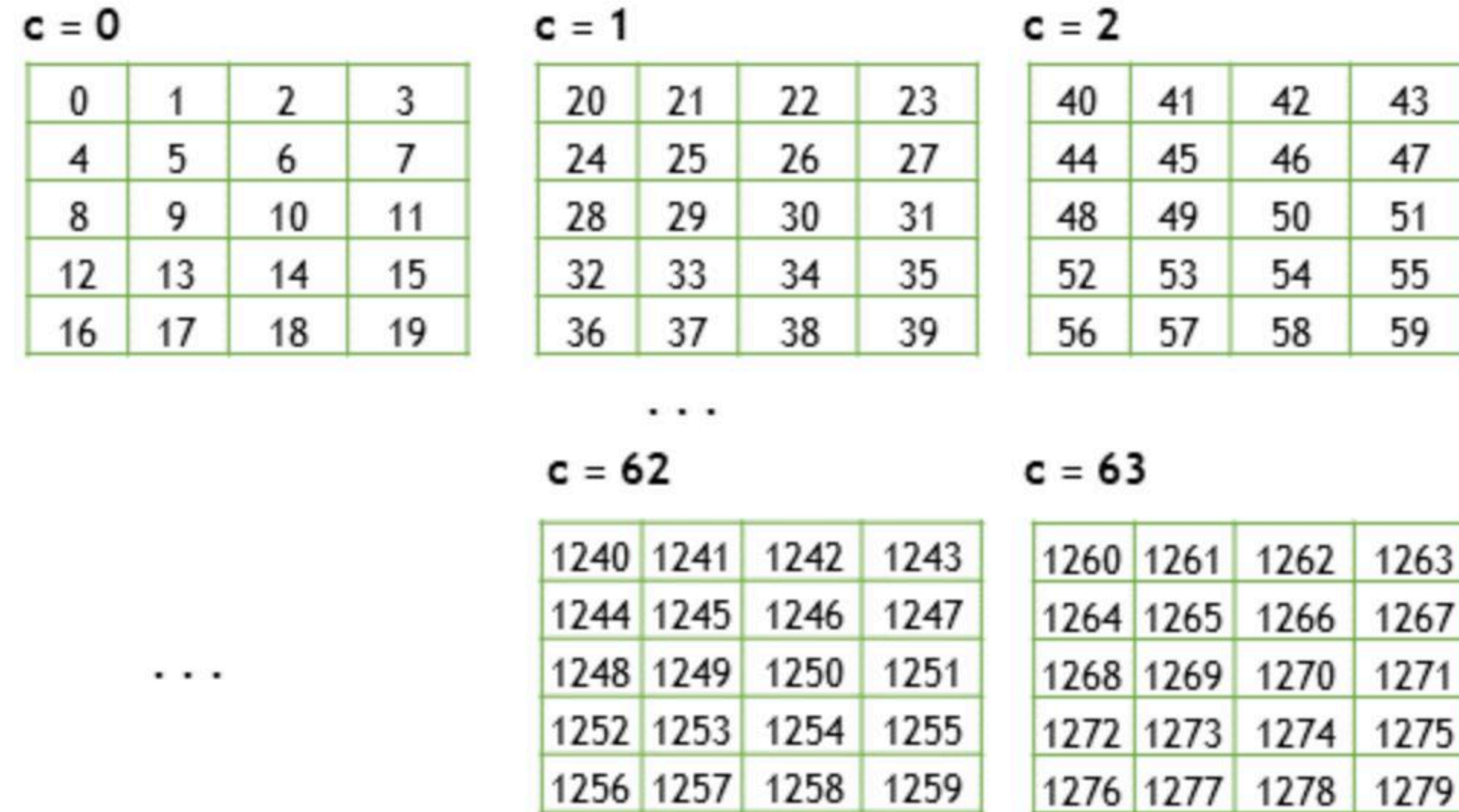
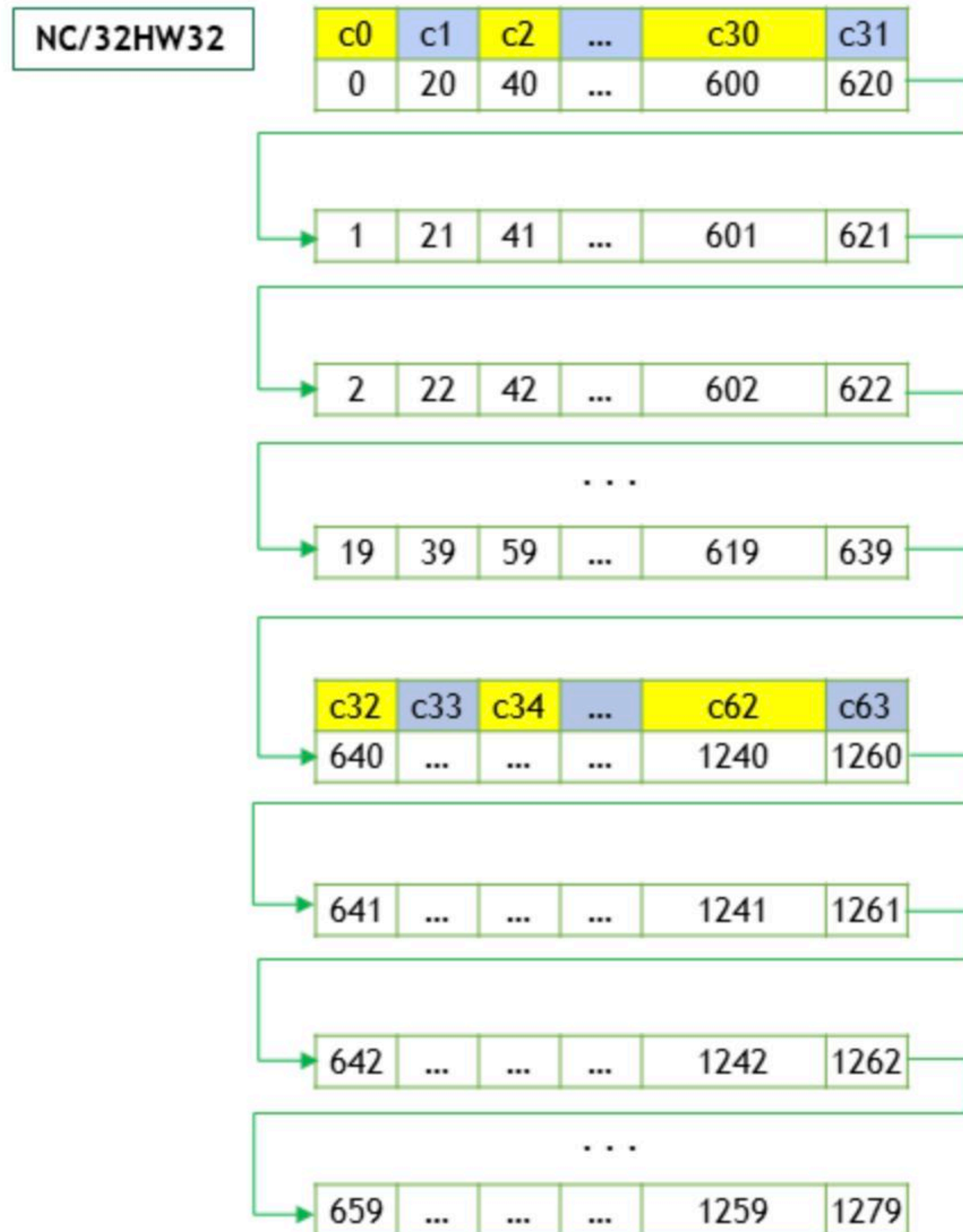
c = 63

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

...

Another tensor layout (blocked C)

- 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.



High level DNN libraries offer high-performance implementations of key DNN layers



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 respo
tensorflow::ops::Conv3DBackpropInputV2	Computes the gradients of 3-D convolution with respo
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 inp 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: $\exp(\mathbf{features}) - 1$ 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.

High level DNN libraries offer high-performance implementations of key DNN layers



<code>tensorflow::ops::AvgPool</code>	Performs average pooling on the input.
<code>tensorflow::ops::AvgPool3D</code>	Performs 3D average pooling on the input.
<code>tensorflow::ops::AvgPool3DGrad</code>	Computes gradients of average pooling function.
<code>tensorflow::ops::BiasAdd</code>	Adds <code>bias</code> to <code>value</code> .
<code>tensorflow::ops::BiasAddGrad</code>	The backward operation for "BiasAdd" on the "bias" tensor.
<code>tensorflow::ops::Conv2D</code>	Computes a 2-D convolution given 4-D <code>input</code> and filter.
<code>tensorflow::ops::Conv2DBackpropFilter</code>	Computes the gradients of convolution with respect to filter.
<code>tensorflow::ops::Conv2DBackpropInput</code>	Computes the gradients of convolution with respect to input.
<code>tensorflow::ops::Conv3D</code>	Computes a 3-D convolution given 5-D <code>input</code> and filter.
<code>tensorflow::ops::Conv3DBackpropFilterV2</code>	Computes the gradients of 3-D convolution with respect to filter.
<code>tensorflow::ops::Conv3DBackpropInputV2</code>	Computes the gradients of 3-D convolution with respect to input.
<code>tensorflow::ops::DataFormatDimMap</code>	Returns the dimension index in the destination data format.
<code>tensorflow::ops::DataFormatVecPermute</code>	Permute input tensor from <code>src_format</code> to <code>dst_format</code> .
<code>tensorflow::ops::DepthwiseConv2dNative</code>	Computes a 2-D depthwise convolution given 4-D <code>input</code> tensors.
<code>tensorflow::ops::DepthwiseConv2dNativeBackpropFilter</code>	Computes the gradients of depthwise convolution with respect to filter.
<code>tensorflow::ops::DepthwiseConv2dNativeBackpropInput</code>	Computes the gradients of depthwise convolution with respect to input.
<code>tensorflow::ops::Dilation2D</code>	Computes the grayscale dilation of 4-D <code>input</code> and 3-D <code>kernel</code> .
<code>tensorflow::ops::Dilation2DBackpropFilter</code>	Computes the gradient of morphological 2-D dilation filter.
<code>tensorflow::ops::Dilation2DBackpropInput</code>	Computes the gradient of morphological 2-D dilation input.

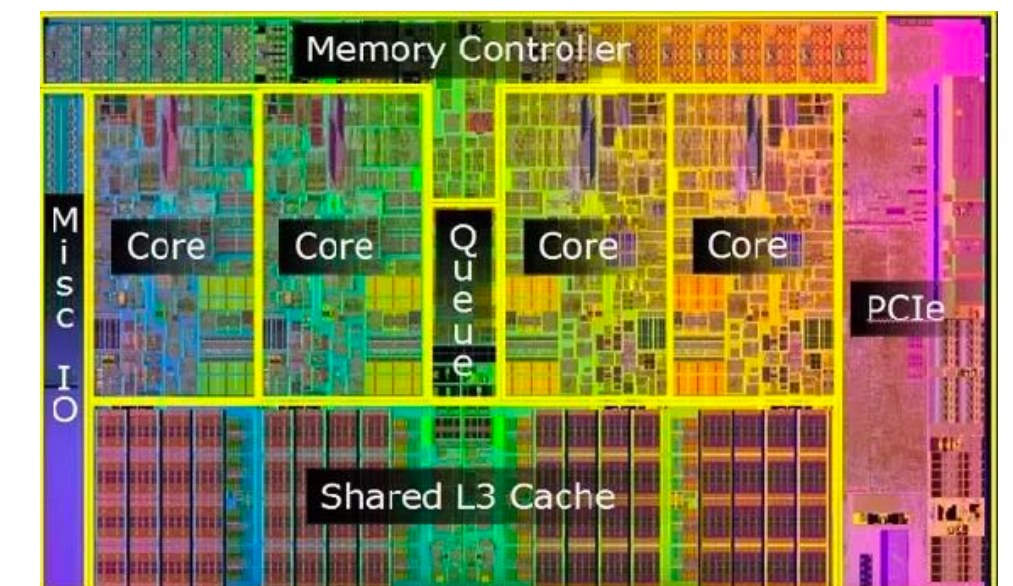
NVIDIA cuDNN



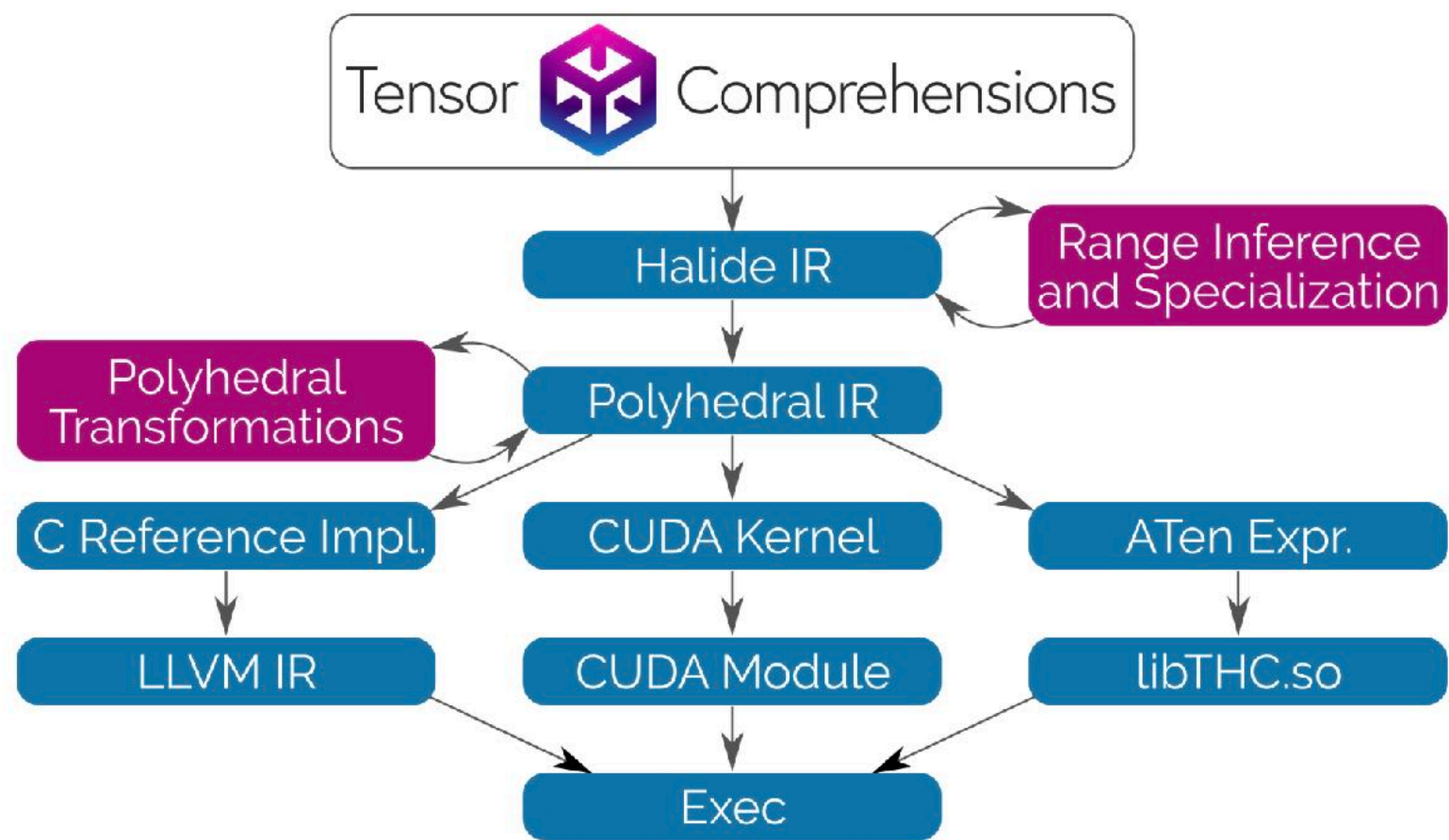
Implemented via
lower level libraries



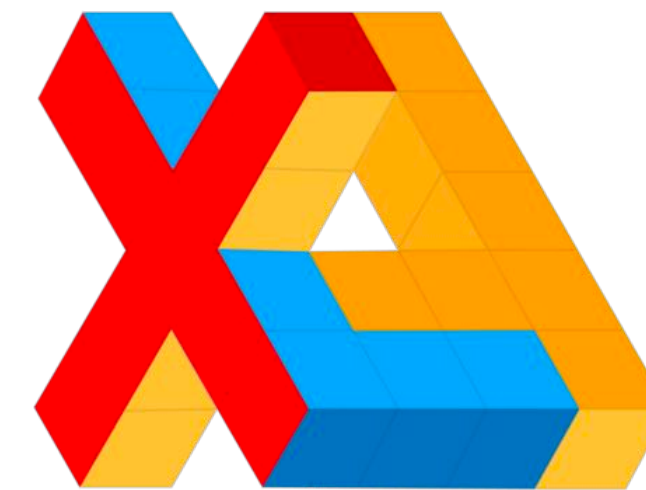
Intel® oneAPI Deep Neural Network Library



Many efforts to automatically schedule key DNN operations



MLIR
Multi-Level IR Compiler Framework



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 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. Check out the [tvm stack home](#) for more information.

NVIDIA TensorRT

Programmable Inference Accelerator



[COMMUNITY](#) [DOWNLOAD](#) [VTA](#) [BLOG](#) [DOCS](#) [CONFERENCE](#) [GITHUB](#)

Introducing TVM Auto-scheduler (a.k.a. Ansor)

Mar 3, 2021 • Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu

Optimizing the execution speed of deep neural networks is extremely hard with the growing model size, operator diversity, and hardware heterogeneity. From a computational perspective, deep neural networks are just layers and layers of tensor computations. These tensor computations, such as matmul and conv2d, can be described by mathematical expressions. However, providing high-performance implementations for them on modern hardware can be very challenging. We have to do various low-level optimizations and utilize special hardware intrinsics to achieve high performance. It takes huge engineering effort to build linear algebra and neural network acceleration libraries like CuBLAS, CuDNN, oneMKL, and oneDNN.

Our life will be much easier if we can just write mathematical expressions and have something magically turn them into efficient code implementations. Three years ago, we started deep learning compiler TVM and its search module AutoTVM were built as the first step towards this goal. AutoTVM employs a template-based search algorithm to find efficient implementations for a given tensor computation. However, it is a template-based approach, so it still requires domain experts to implement a non-trivial template for every operator on every platform. Today, there are more than 15k lines of code for these templates in the TVM code repository. Besides being very hard to develop, these templates often have inefficient and limited search spaces, making them unable to achieve optimal performance.

