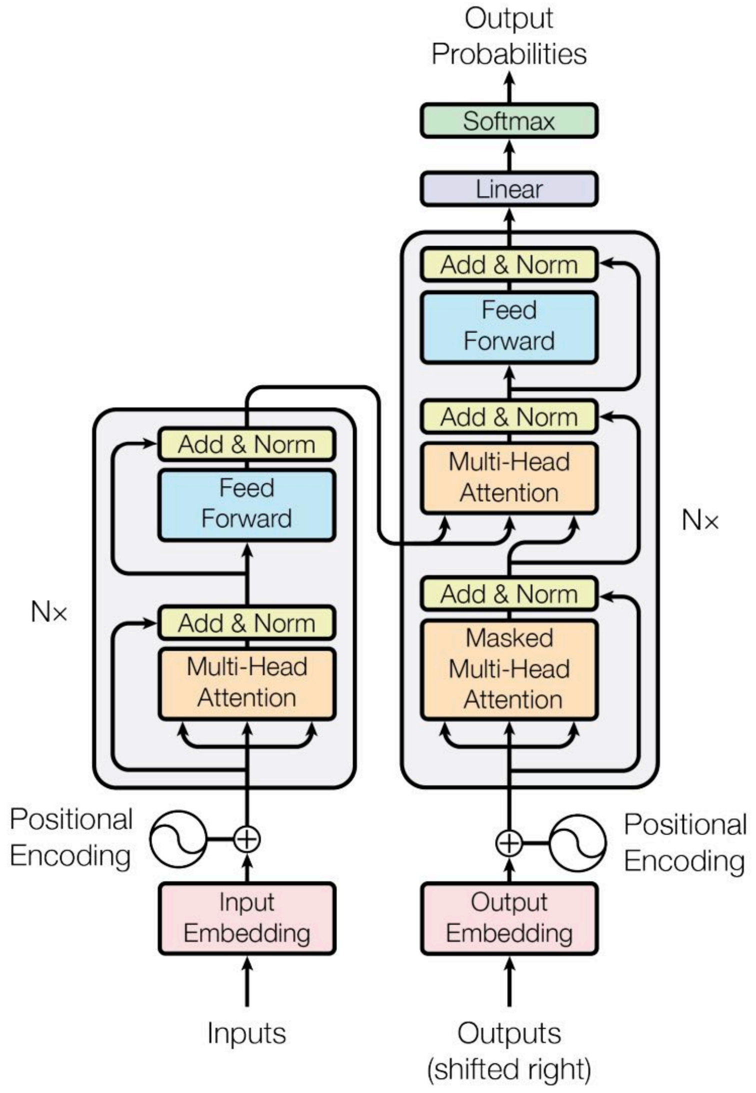
Lecture 9:

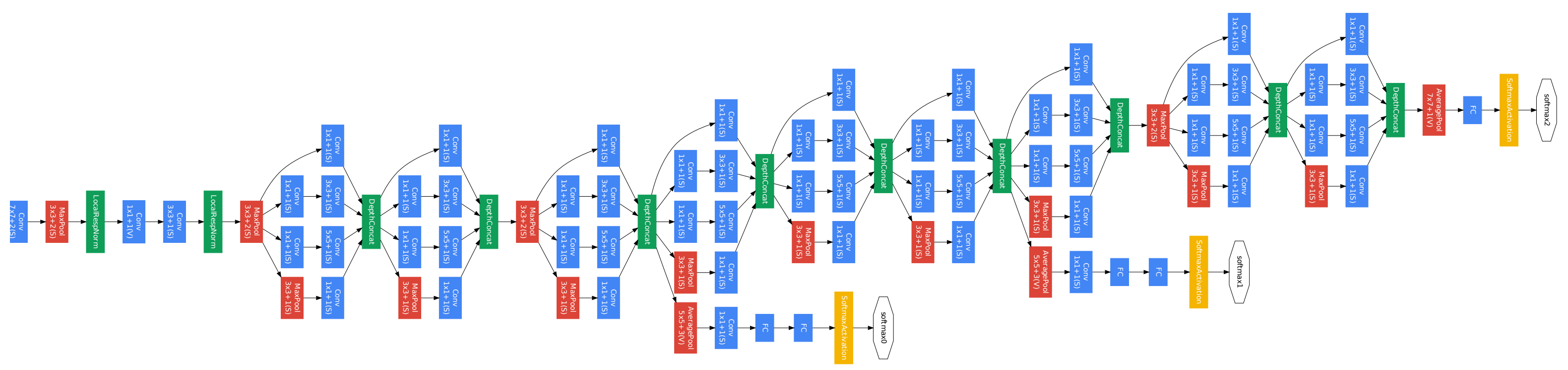
Efficiently Evaluating DNNs

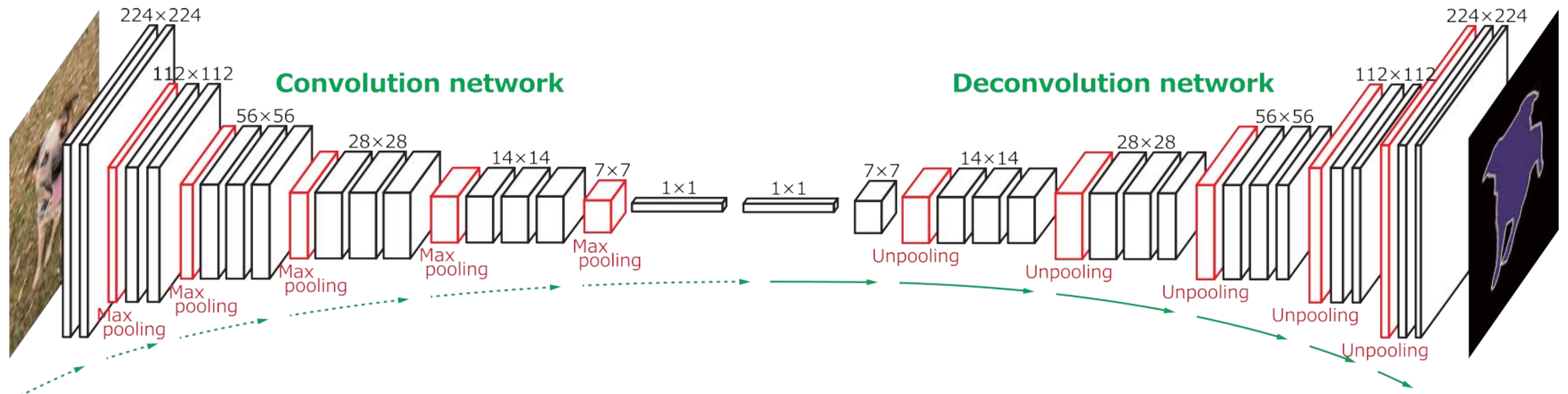
Parallel Computing
Stanford CS149, Fall 2025

Extreme efficiency challenge

Diverse collection of AI models (topology and size)







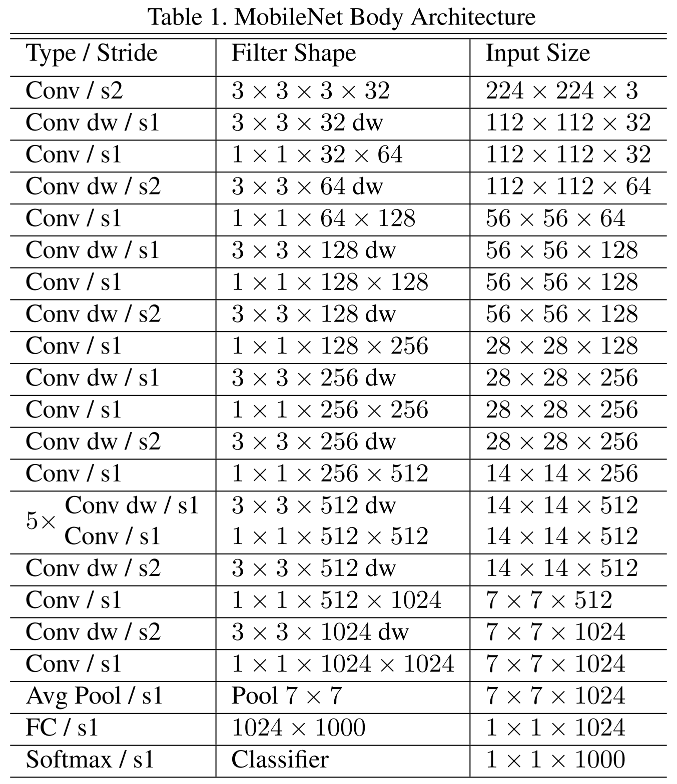


Table 1. MobileNet Body Architecture

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

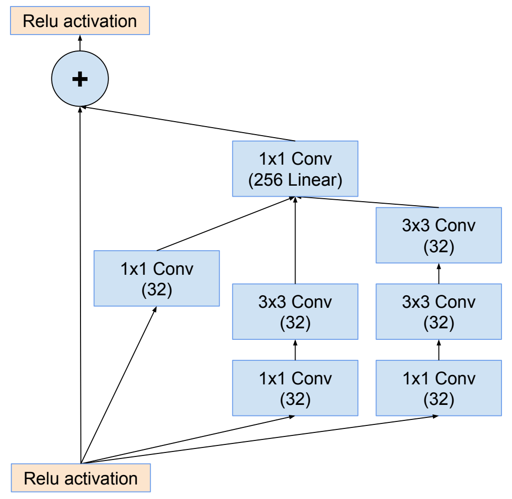
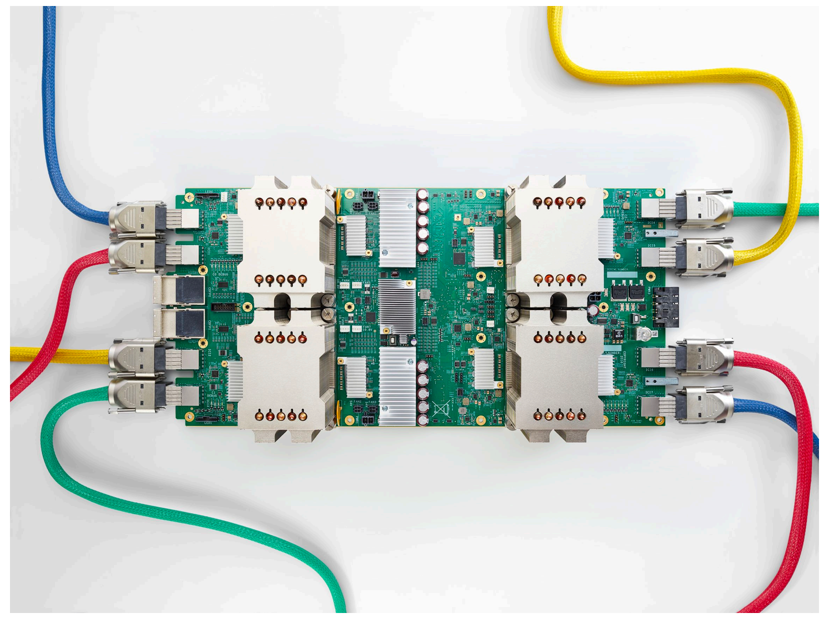
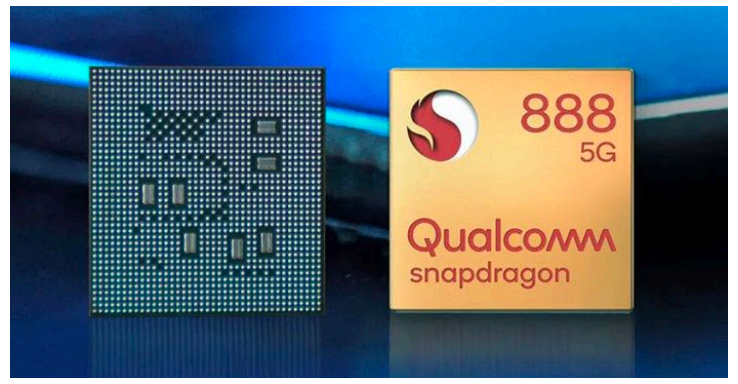


Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

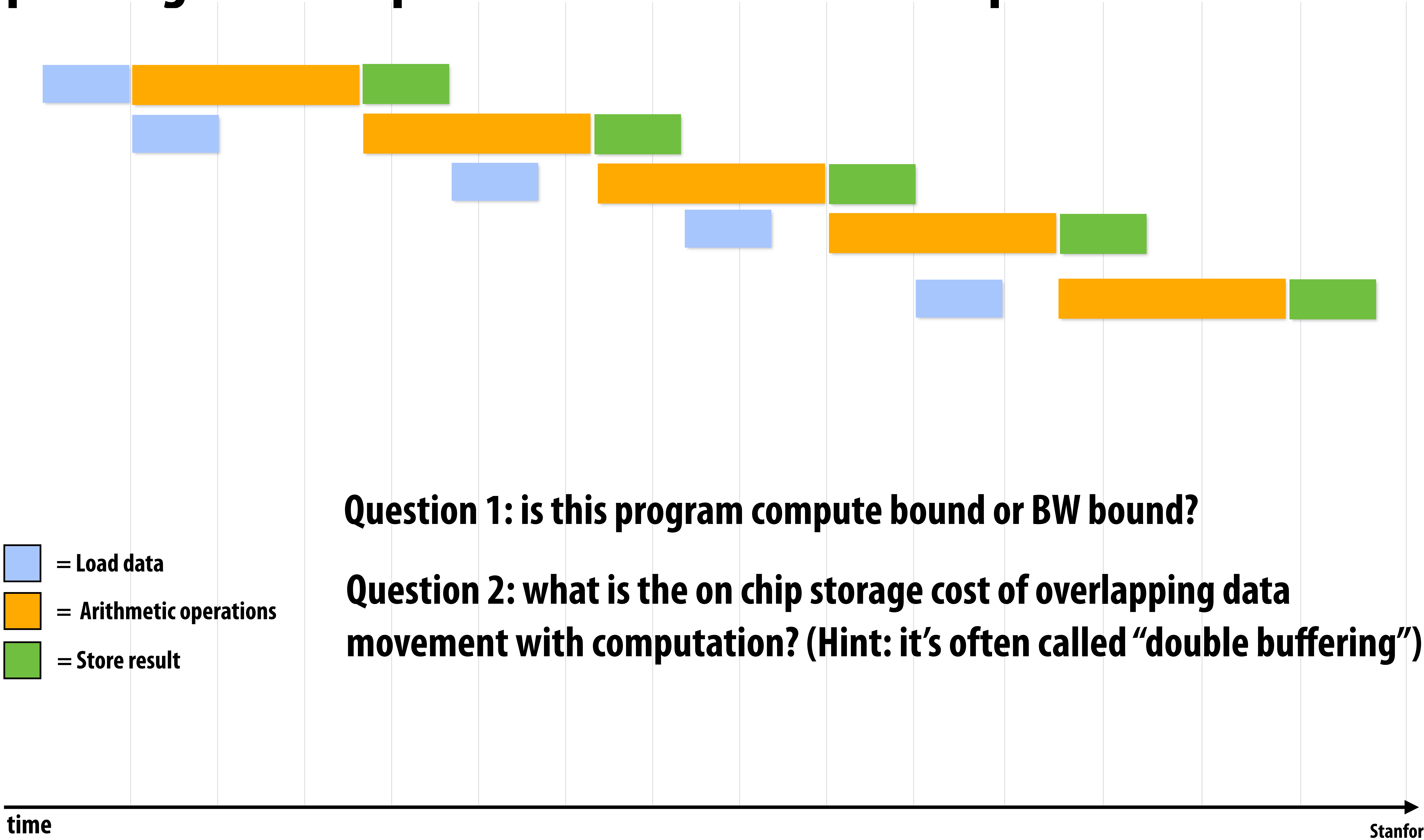
Many Target Devices





Things you already know — and should remember

Pipelining to overlap data movement with computation



🟦 = Load data

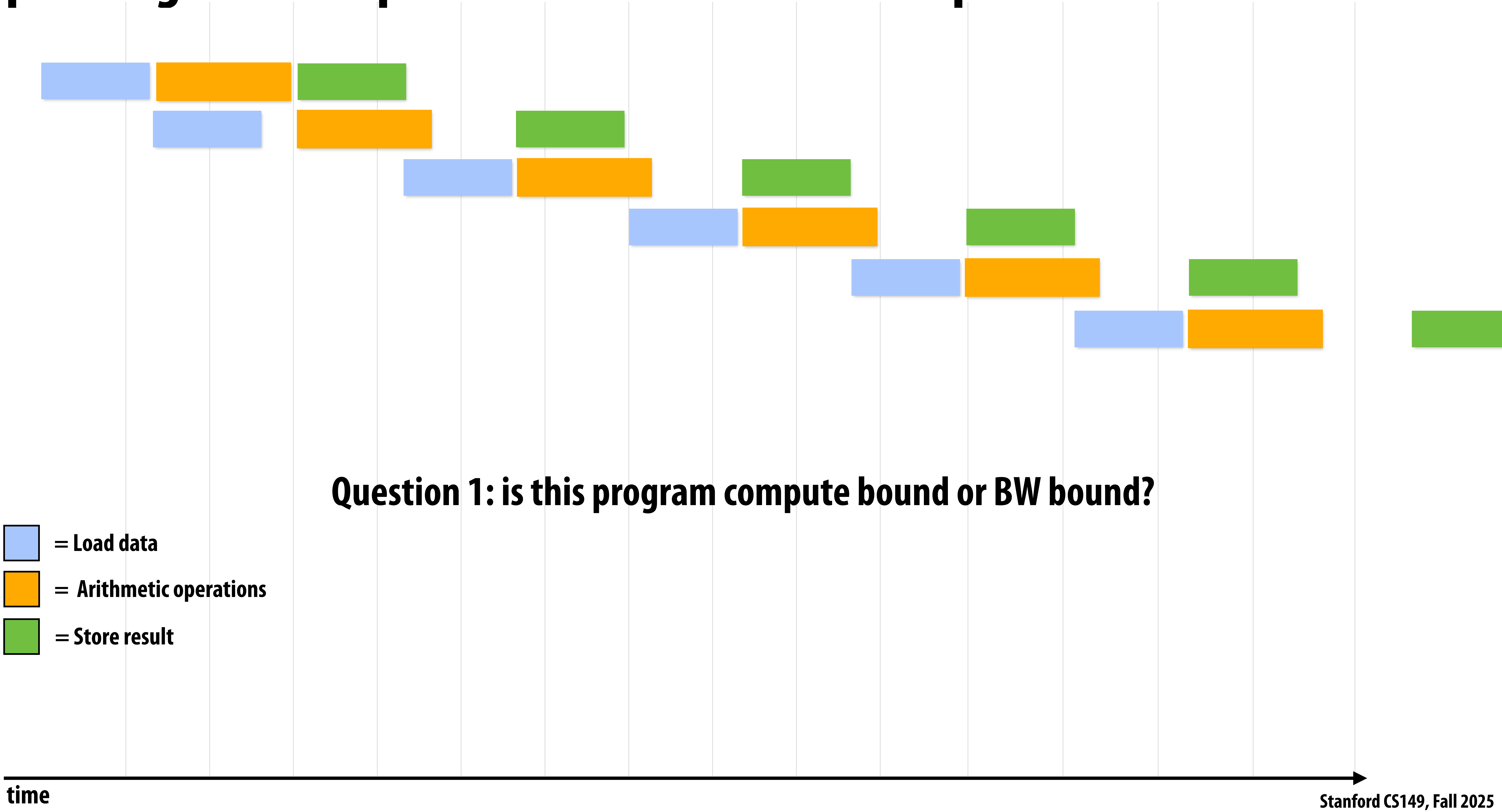
🟧 = Arithmetic operations

🟩 = Store result

Question 1: is this program compute bound or BW bound?

Question 2: what is the on chip storage cost of overlapping data movement with computation? (Hint: it's often called "double buffering")

Pipelining to overlap data movement with computation



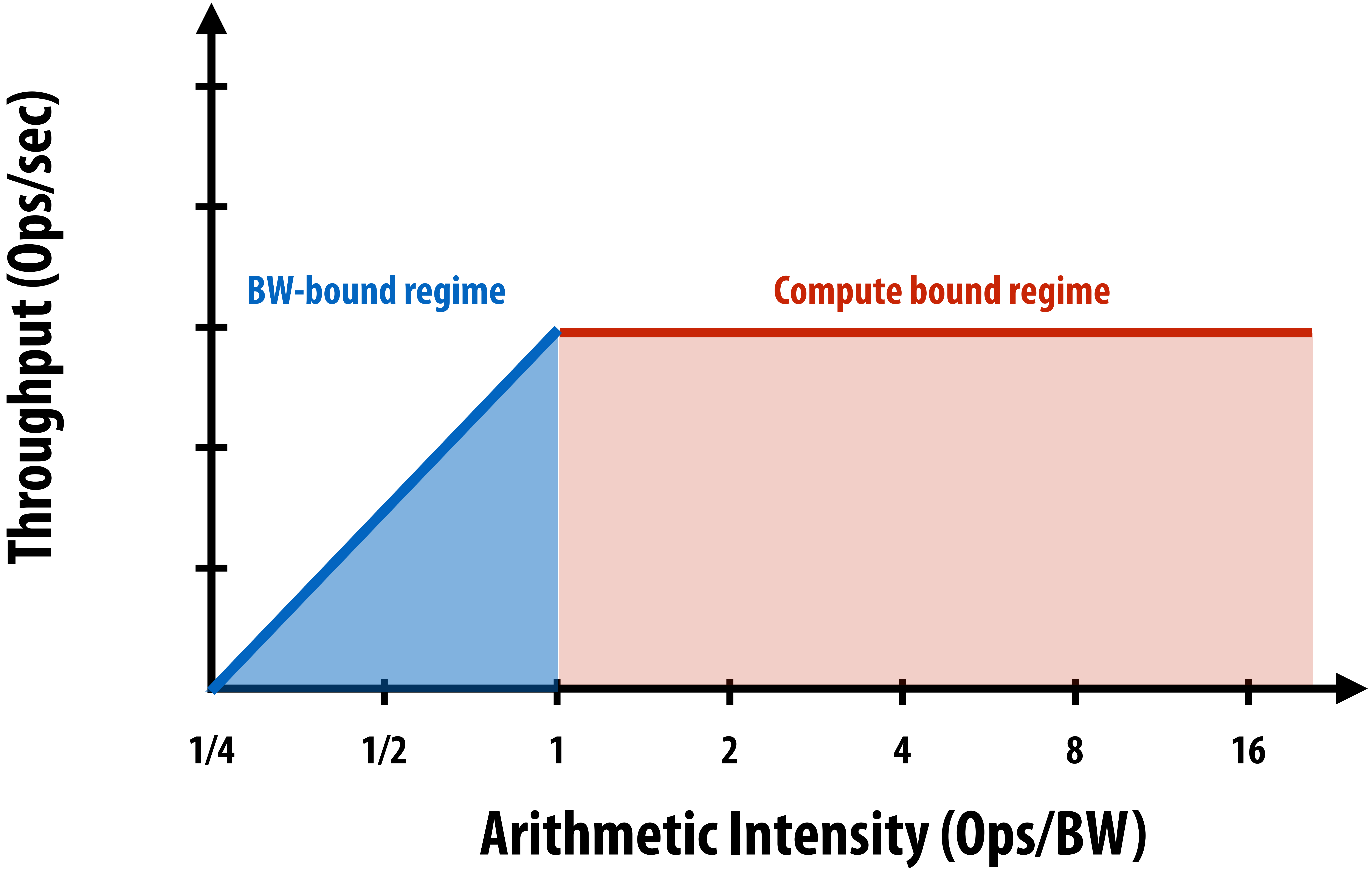
Question 1: is this program compute bound or BW bound?

🟦 = Load data

🟧 = Arithmetic operations

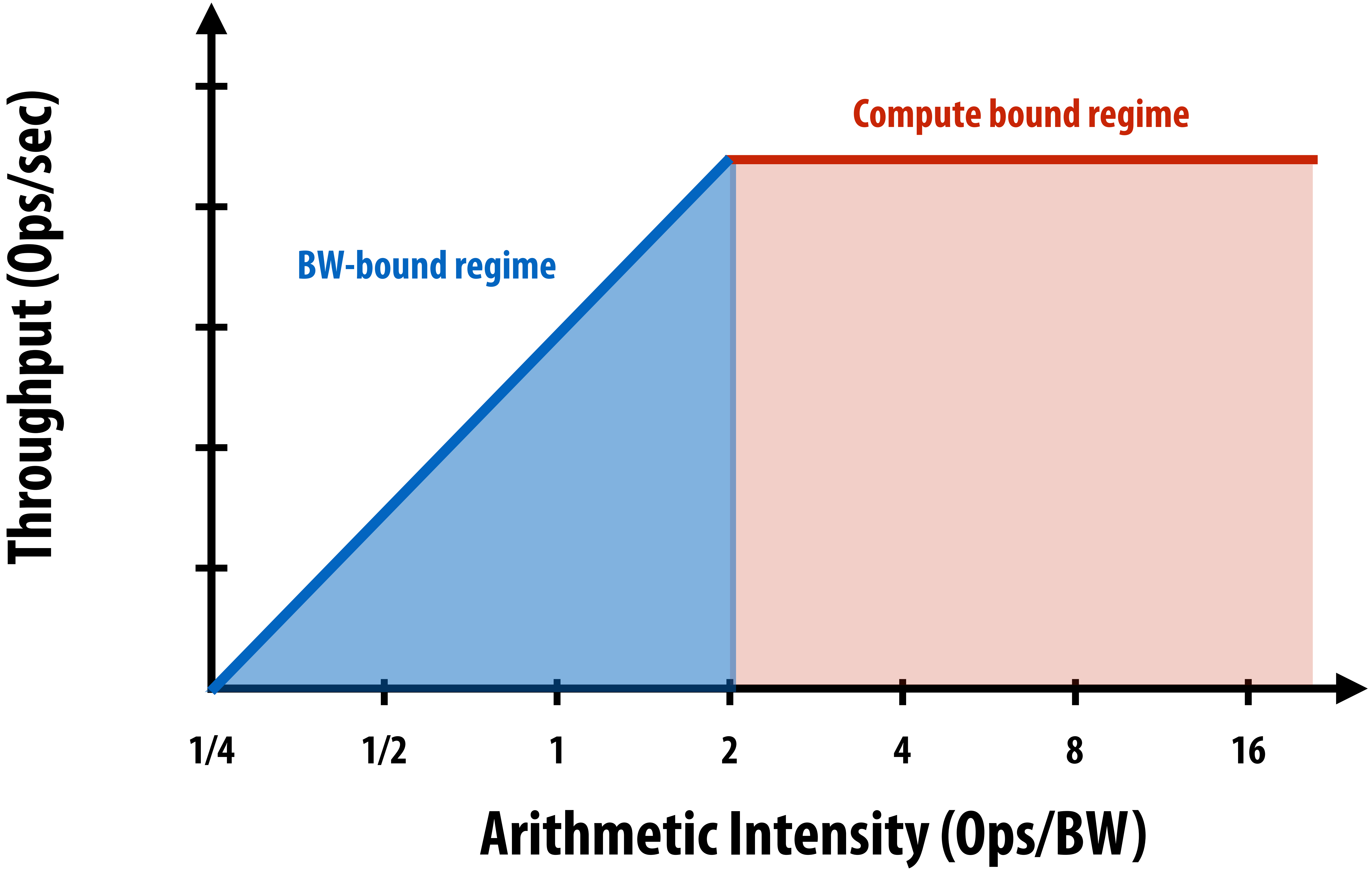
🟩 = Store result

A roofline curve



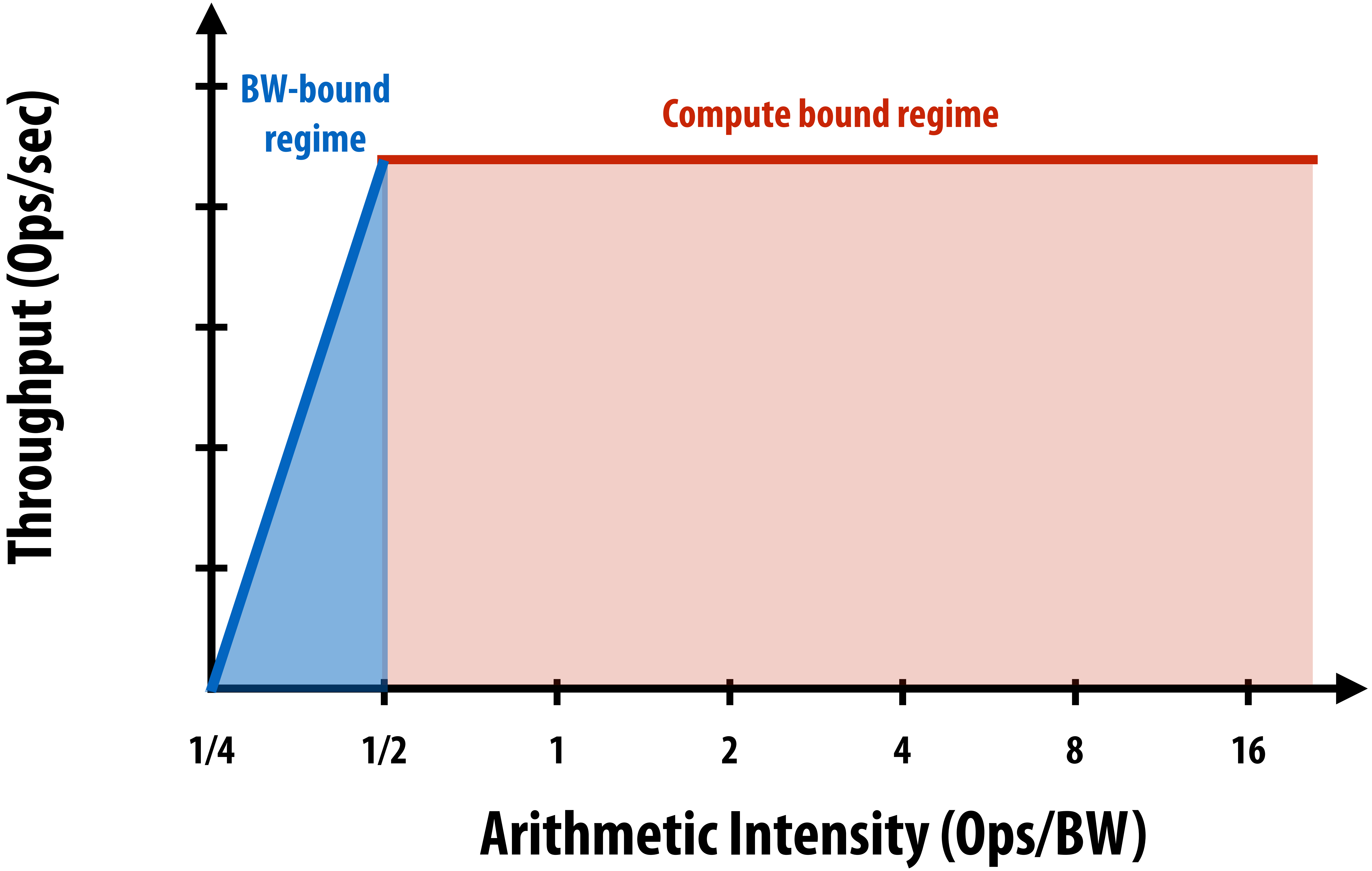
A roofline curve

Computer with the same memory system but higher peak compute capability



A roofline curve

Computer with the higher-throughput memory system and higher peak compute capability



Recall the loop fusion transformation: fuse multiple loops into one to increase a program’s arithmetic intensity

The transformation of the code in program 1 to the code in program 2 is called “loop fusion”

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

Two loads, one store per math op
(arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Program 2

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

Review

- When communication and computation are overlapped (hiding memory latency), the capabilities of the machine (ops throughput and communication bandwidth) AND the arithmetic intensity of the program determine if the program's overall instruction throughput is limited by available bandwidth ("bandwidth bound") or by the machine's instruction processing capability ("compute bound")

- Overlapping communication and computation costs footprint, since buffers for the data being processing AND the data being transferred need to be maintained on chip.

- Increasing arithmetic processing ability ("faster hardware") makes a program more likely to be bandwidth bound

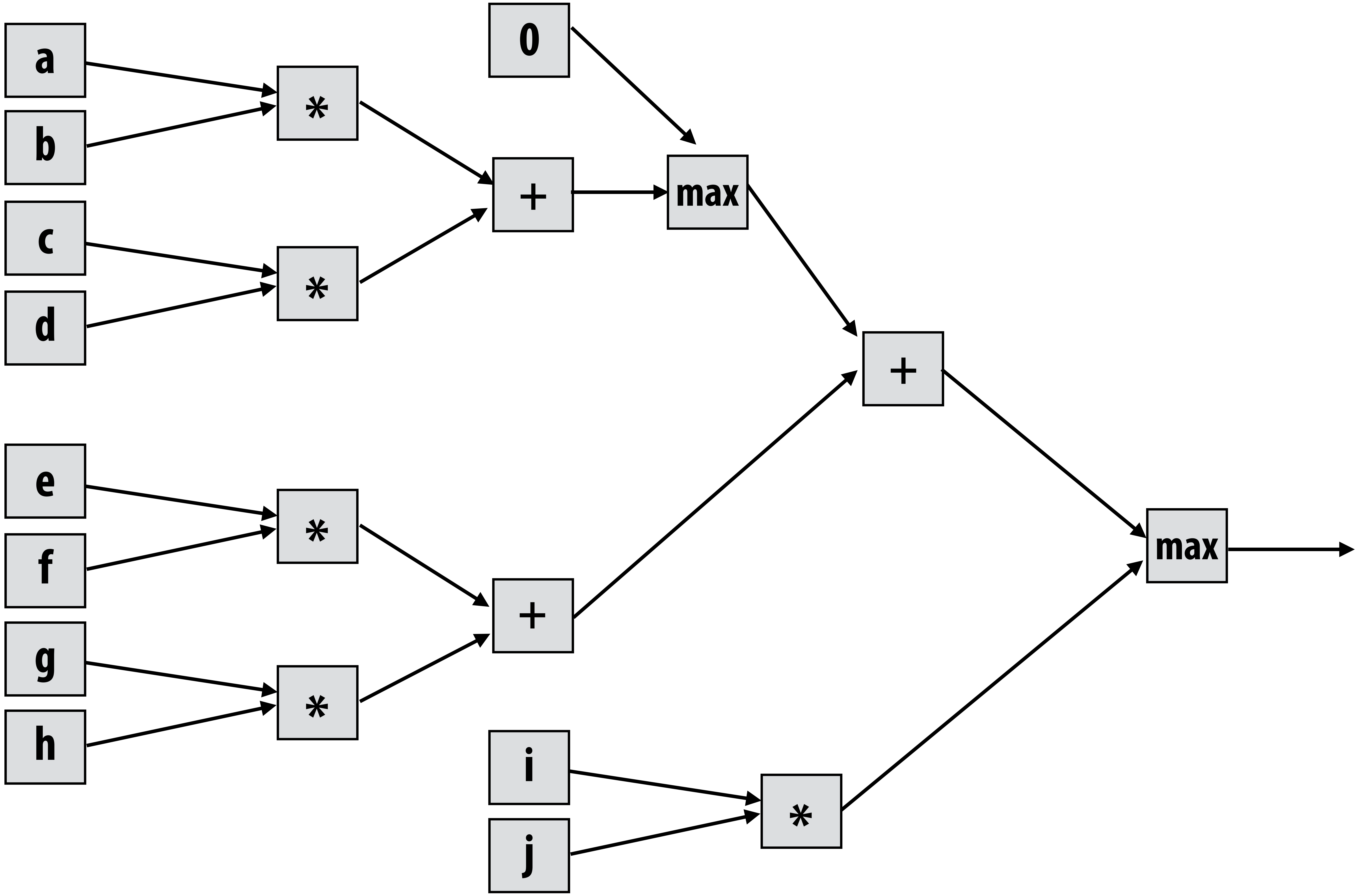
- Increasing a program's arithmetic intensity ("a program change") makes a program more likely to be compute bound

If you know the previous slide, you know almost everything you need to know about the software side* of performance optimization of modern AI.

* If you want to know the rest, wait for next class… and it basically amounts to (1) data movement costs energy, (2) chip resources used for on-chip storage are resources that cannot be used for compute, so minimize buffers as much as possible

Mini-intro:
Convolutional Neural Networks

Consider the following expression

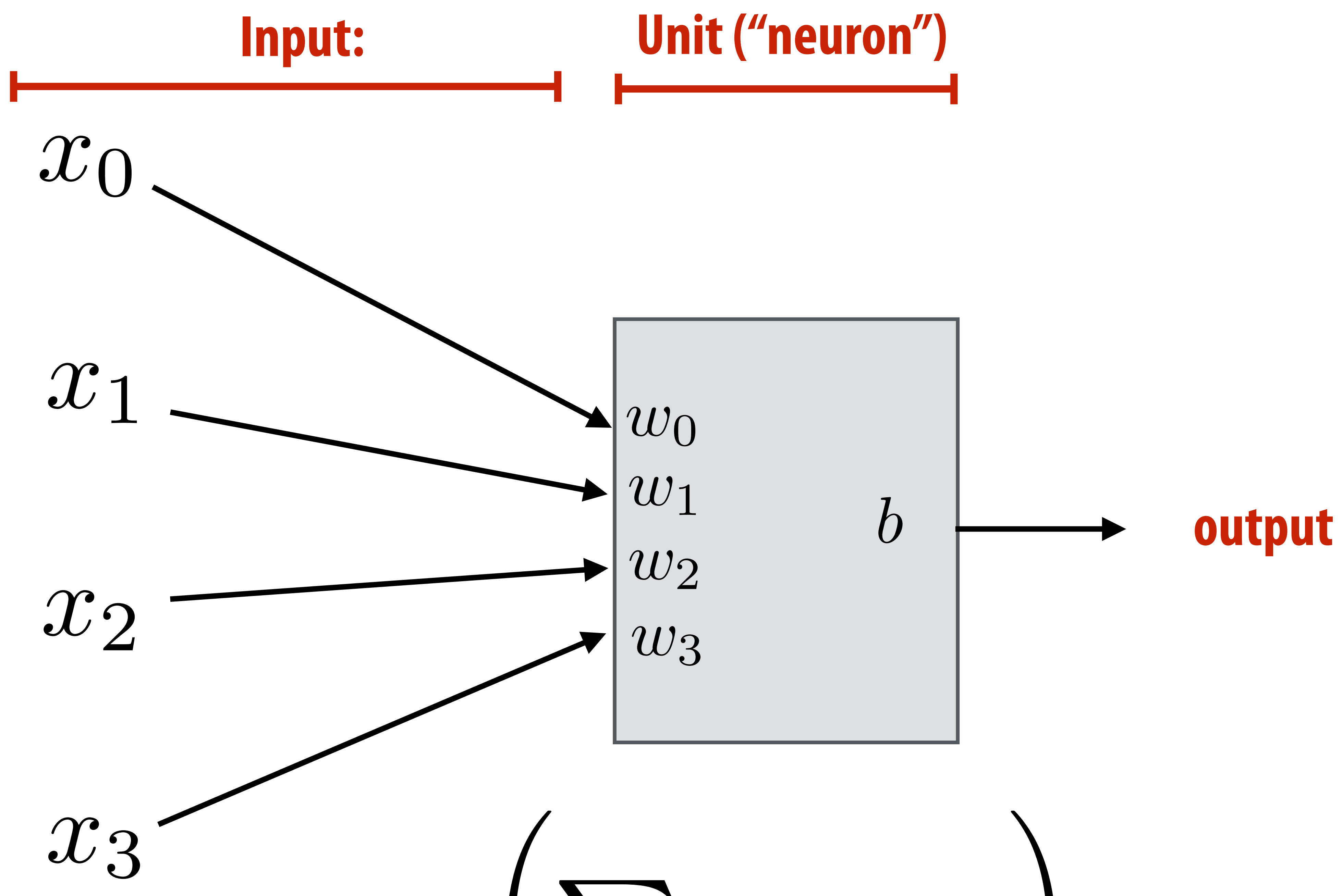


```
max( max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j)
```

What is a deep neural network?

A basic unit:

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



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

Example: rectified linear unit (ReLU)

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

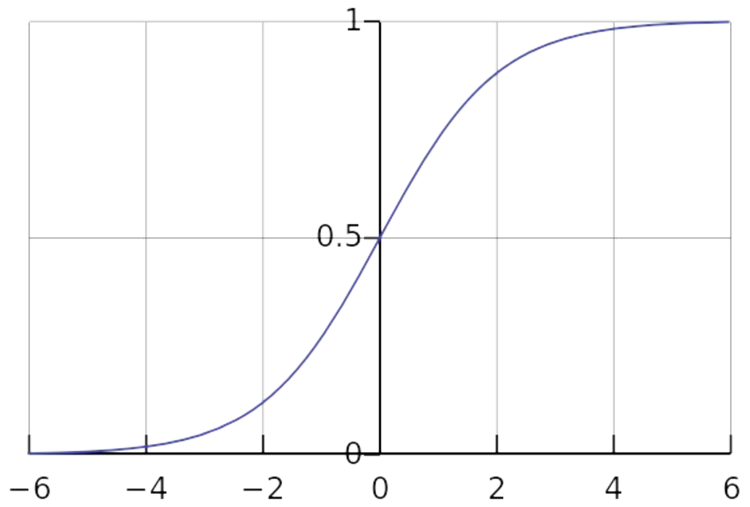
Basic computational interpretation:

It is just a circuit!

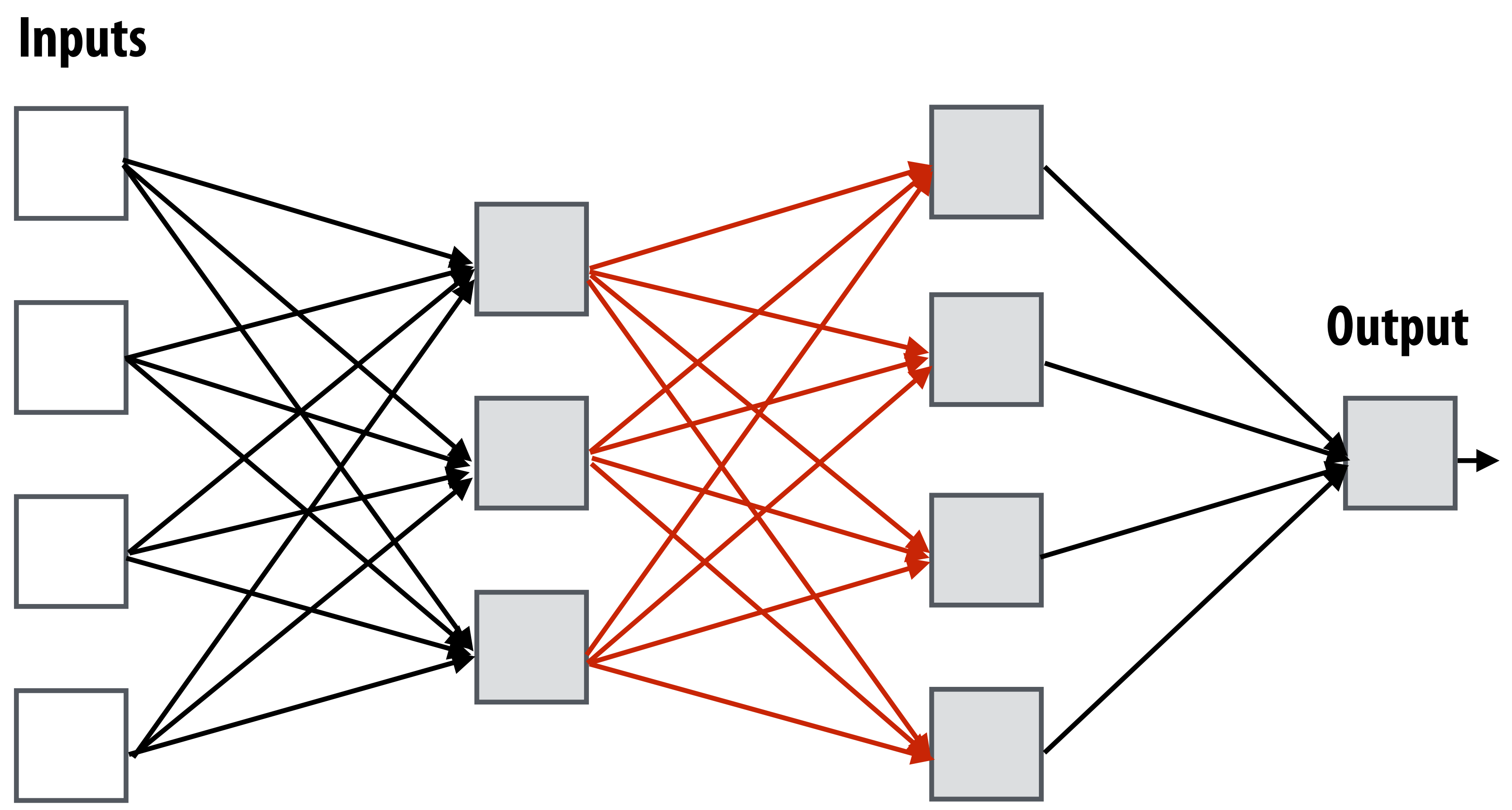
Machine learning interpretation:

Binary classifier: interpret output as the probability of one class

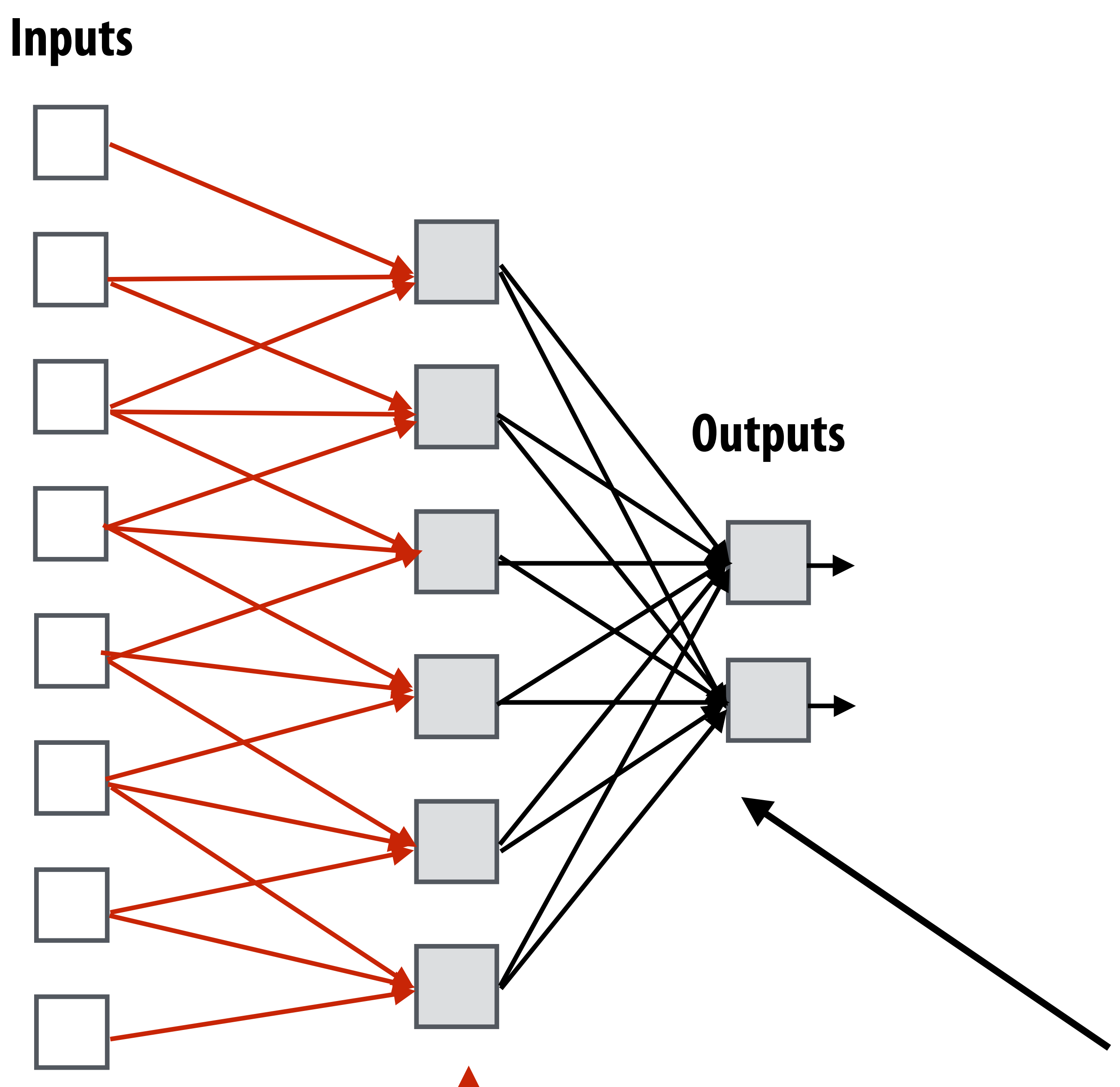
$$f(x) = \frac{1}{1+e^{-x}}$$



Deep neural network: topology



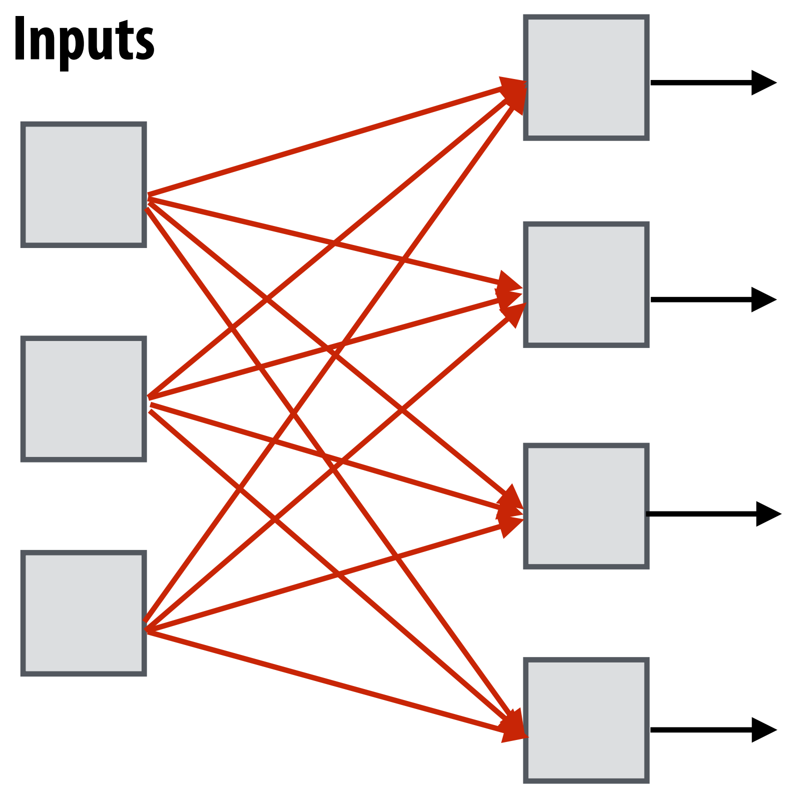
Fully connected layer



Sparsely (locally) connected layer (each unit only received inputs from three input nodes)

Fully connected layer

Fully connected layer as matrix-vector product



Fully connected layer

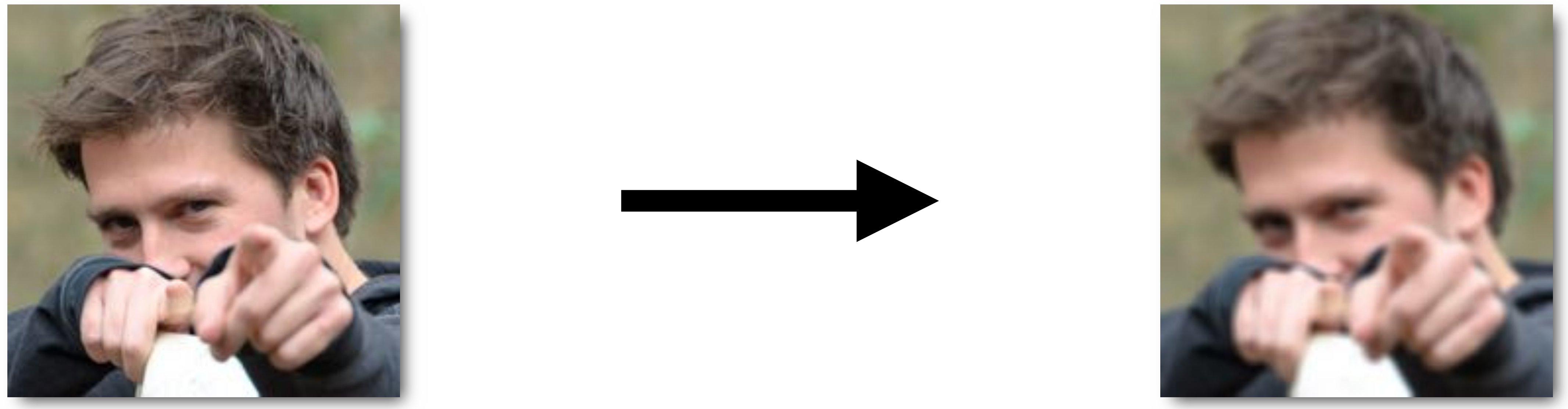
$$f\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{22} & w_{21} & w_{22} \\ w_{32} & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

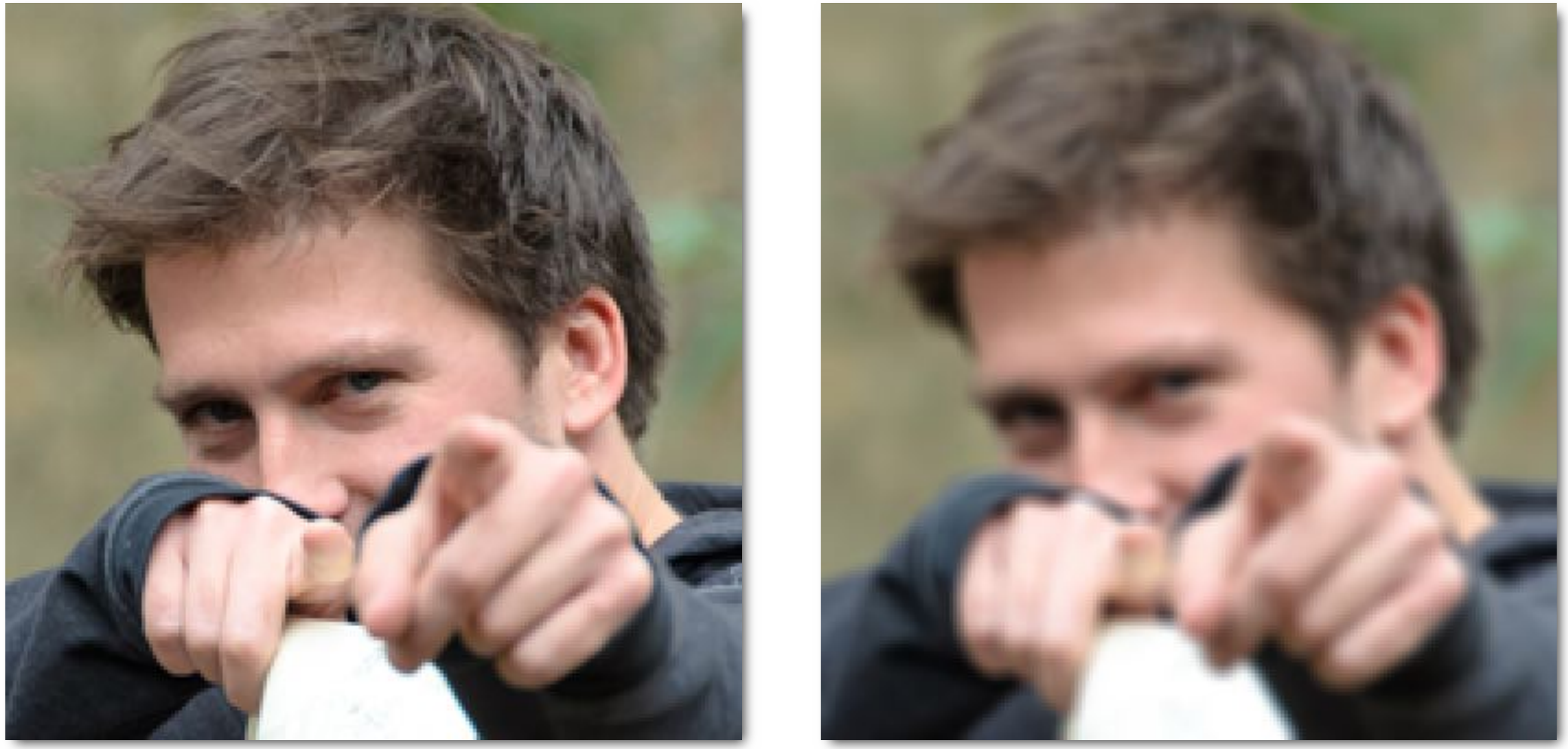
Assume f() is the element-wise max function (ReLU)

2D convolution: what does this C code do?

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

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/9,
                   1.f/9, 1.f/9, 1.f/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;
  }
}
```

The code on the previous slide performed a 3x3 blur

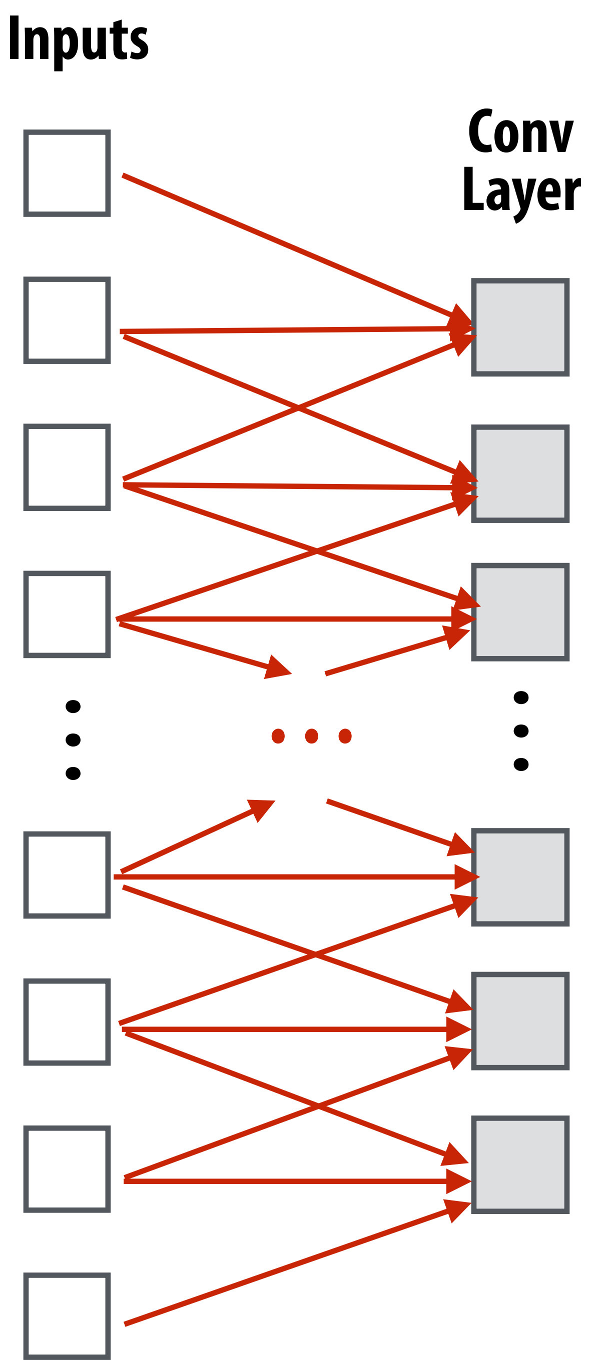


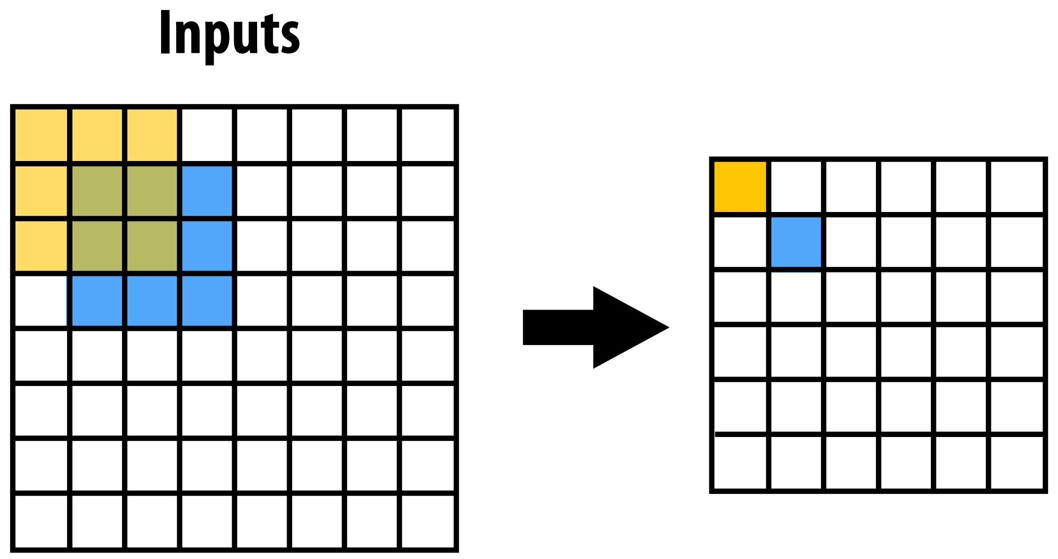


(Zoomed view)

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;
  }
}
```

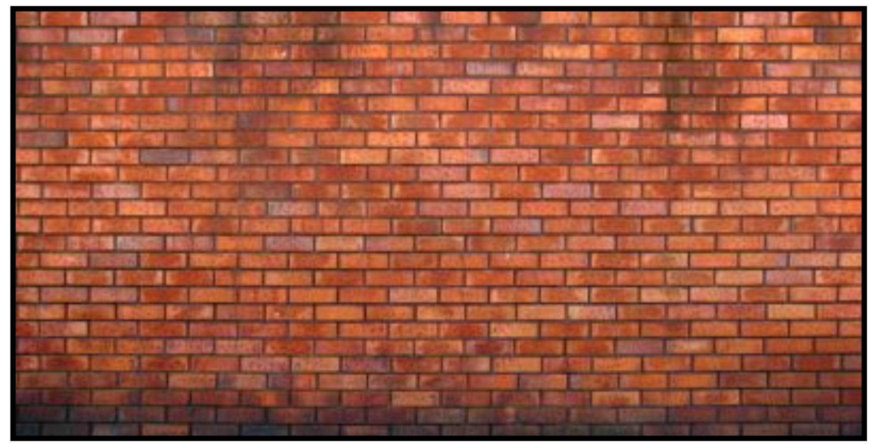




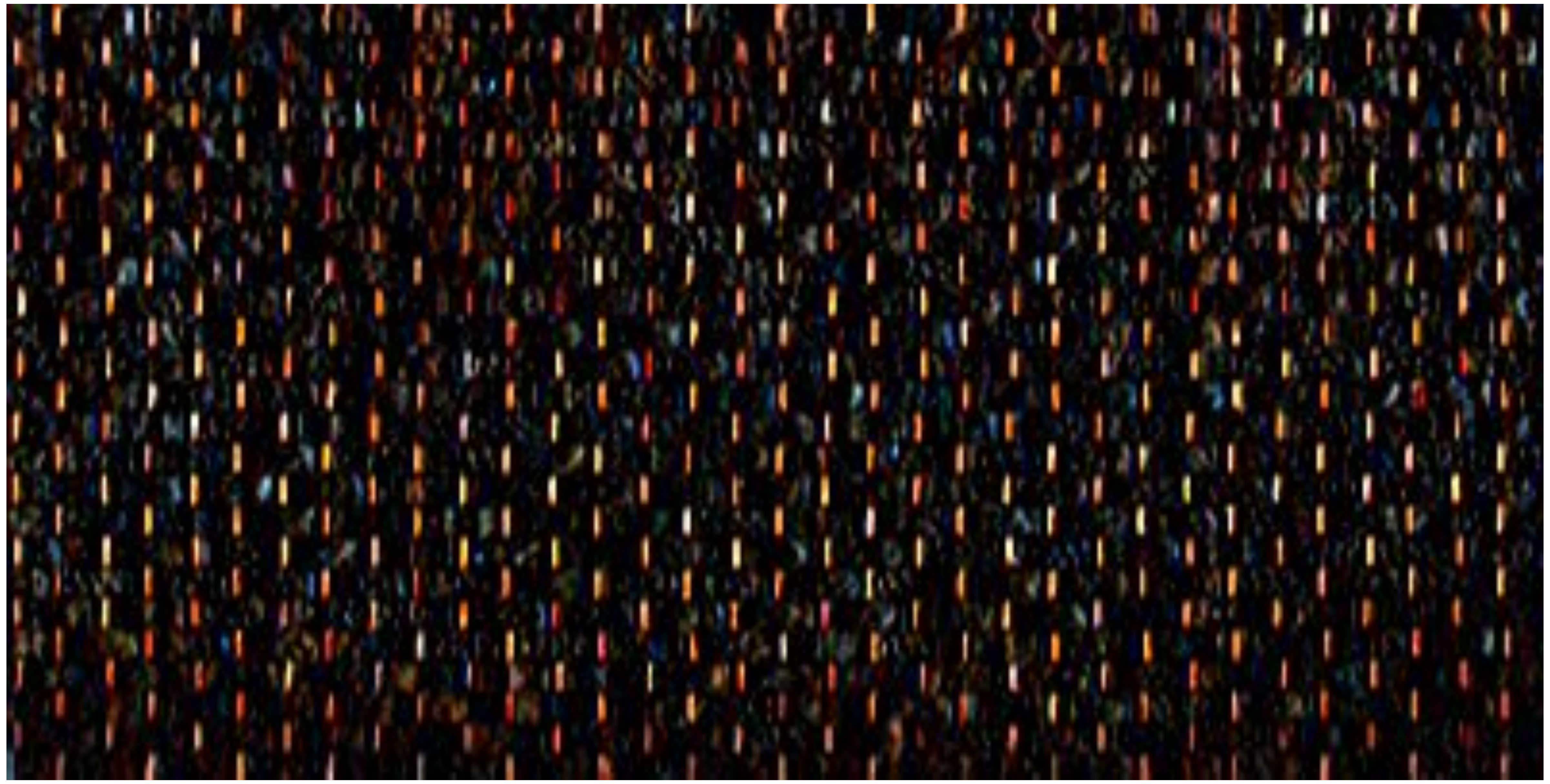
Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):

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

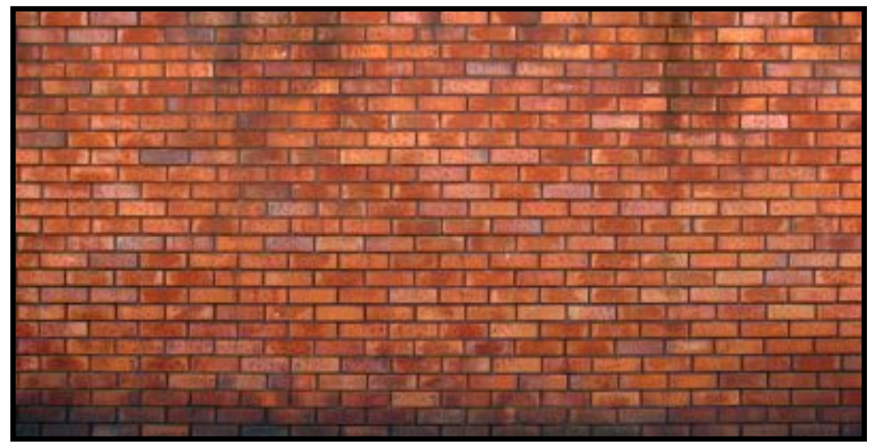
Gradient detection filters



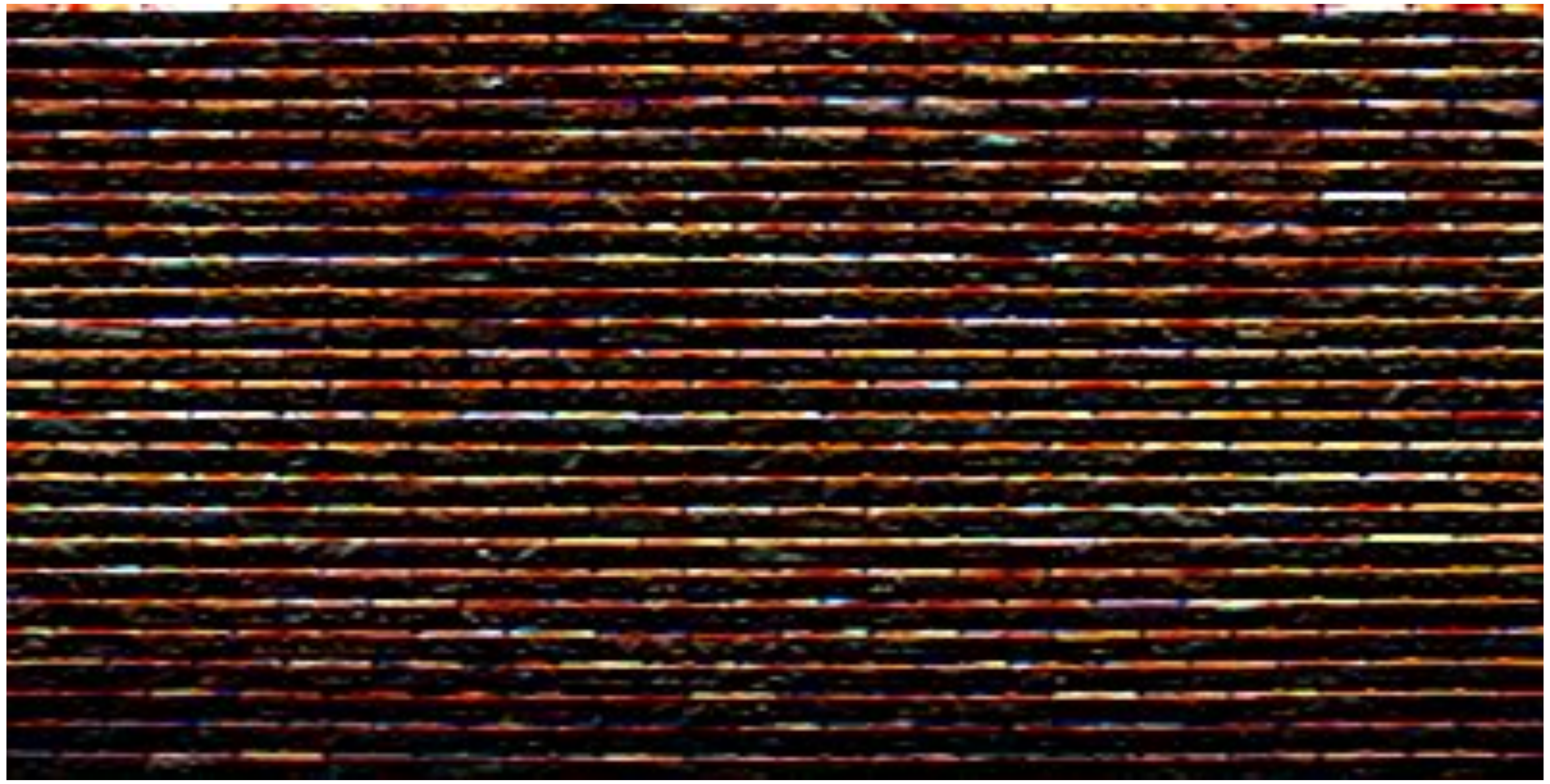
$$* \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} =$$



Responds to horizontal gradients



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



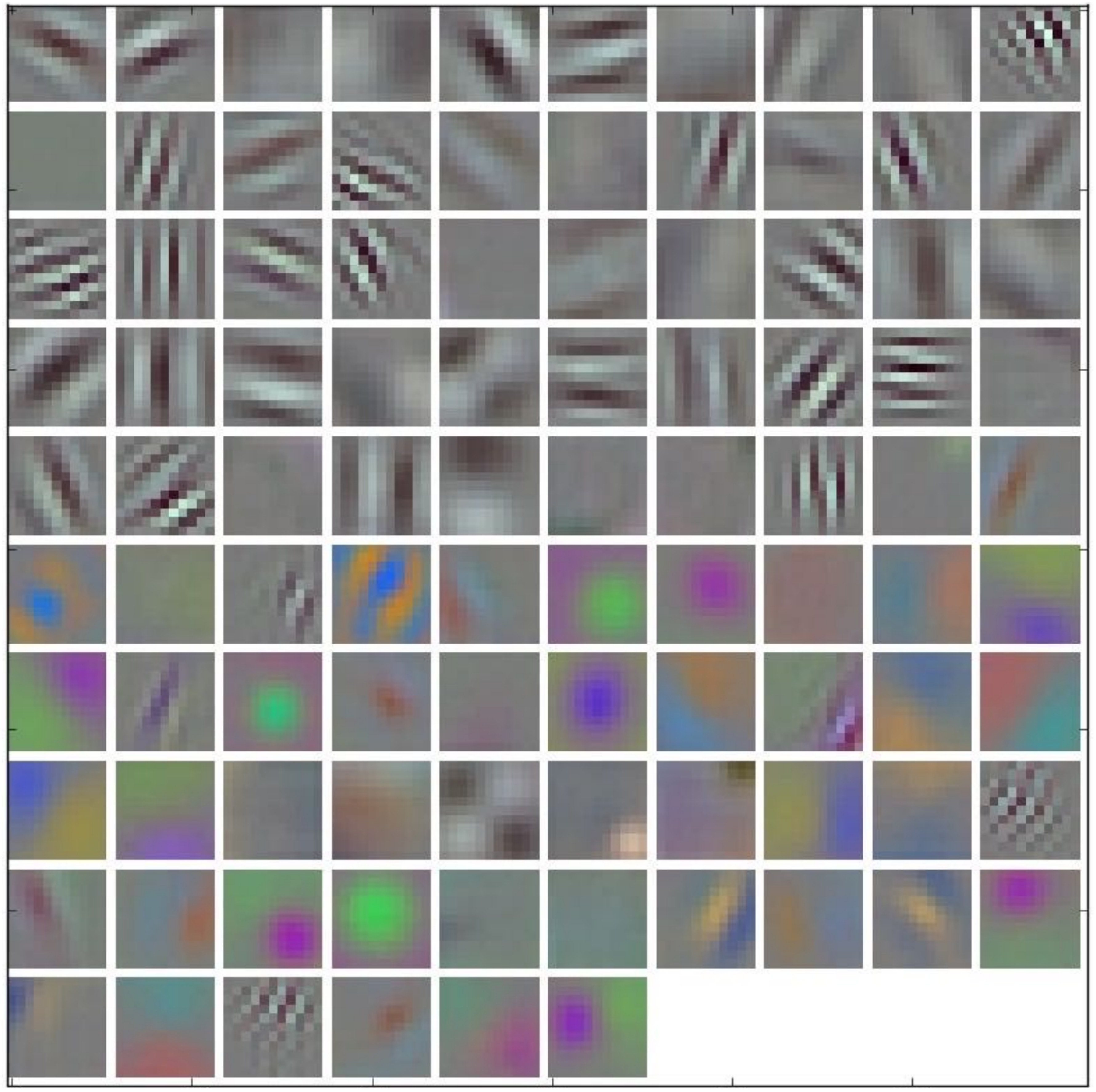
Responds to vertical gradients

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

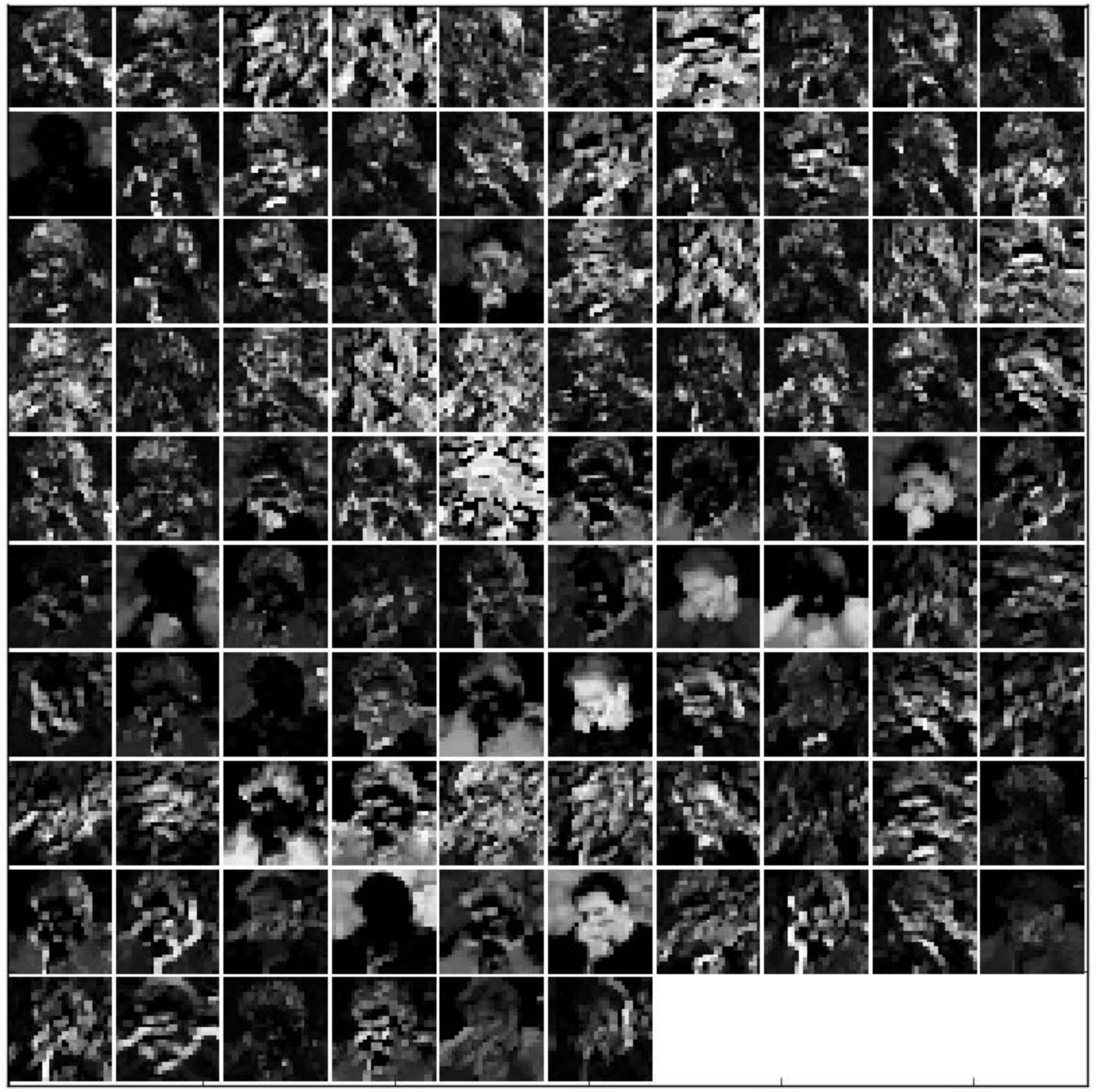
Applying many filters to an image at once

Input RGB image (W x H x 3)

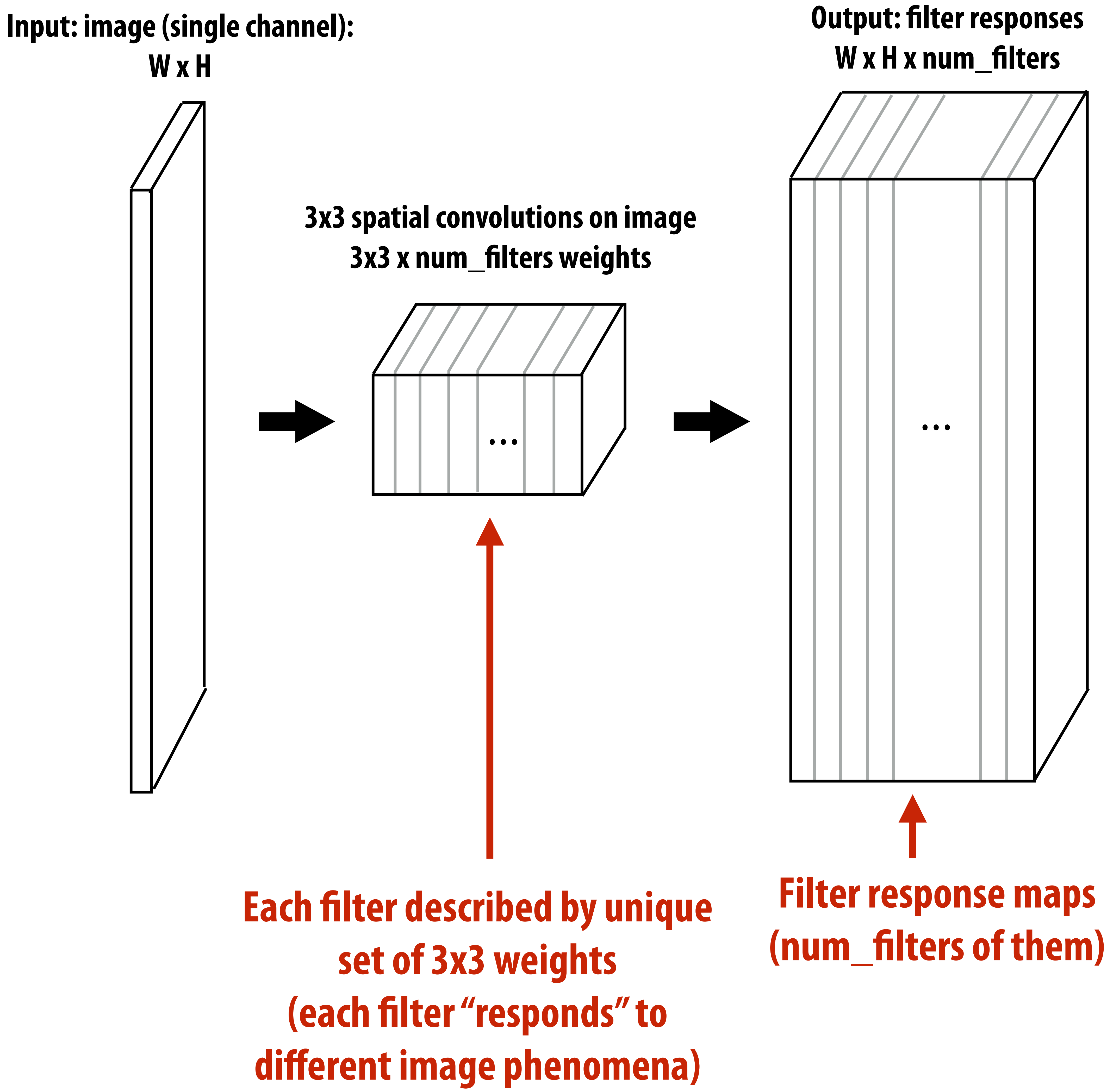
96 11x11x3 filters
(3D because they operate on RGB)



96 responses (normalized)



Applying many filters to an image at once



Input: image (single channel): W x H

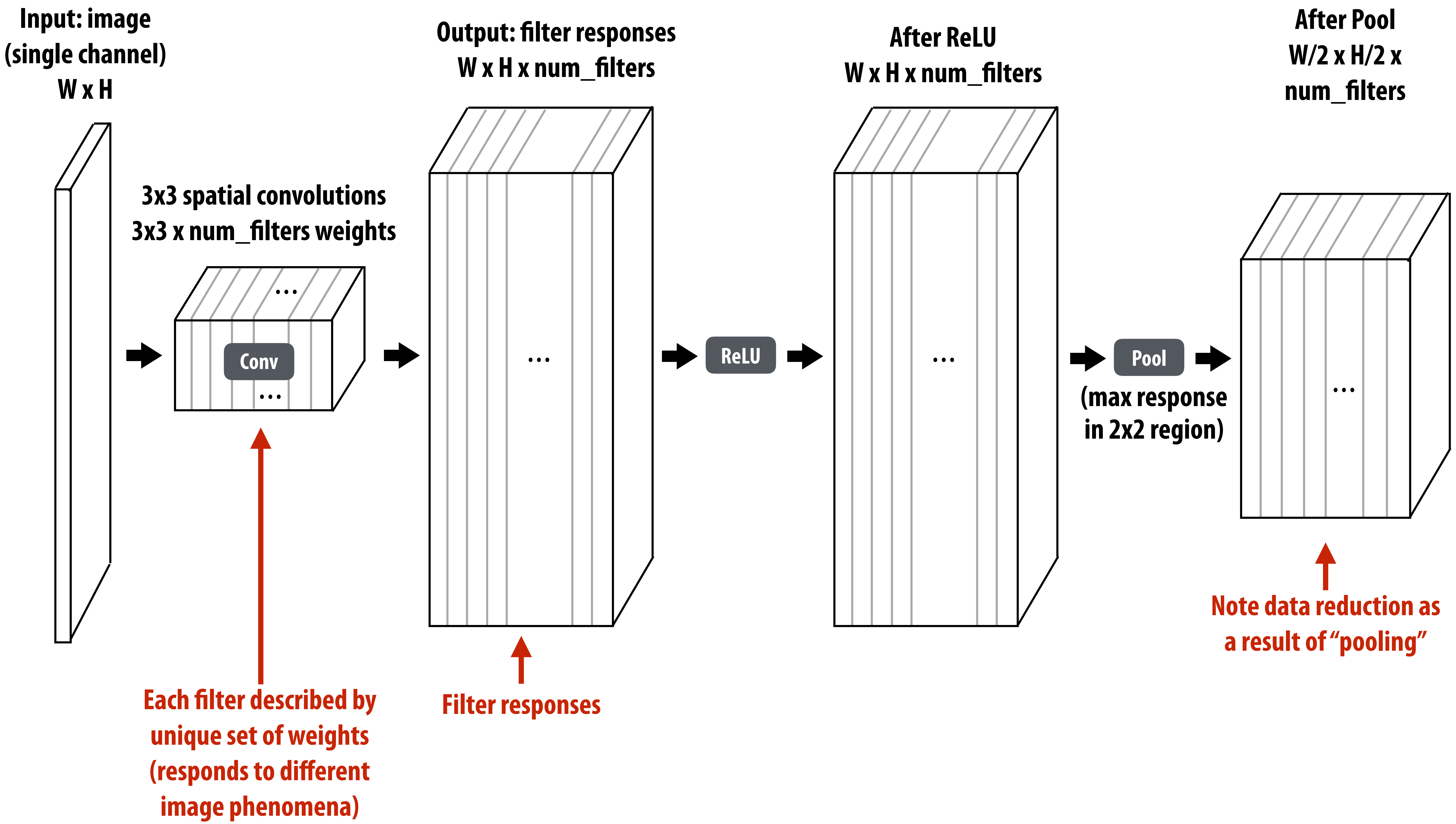
3x3 spatial convolutions on image 3x3 x num_filters weights

Output: filter responses W x H x num_filters

Each filter described by unique set of 3x3 weights (each filter "responds" to different image phenomena)

Filter response maps (num_filters of them)

Adding additional layers



Efficiently implementing convolution layers

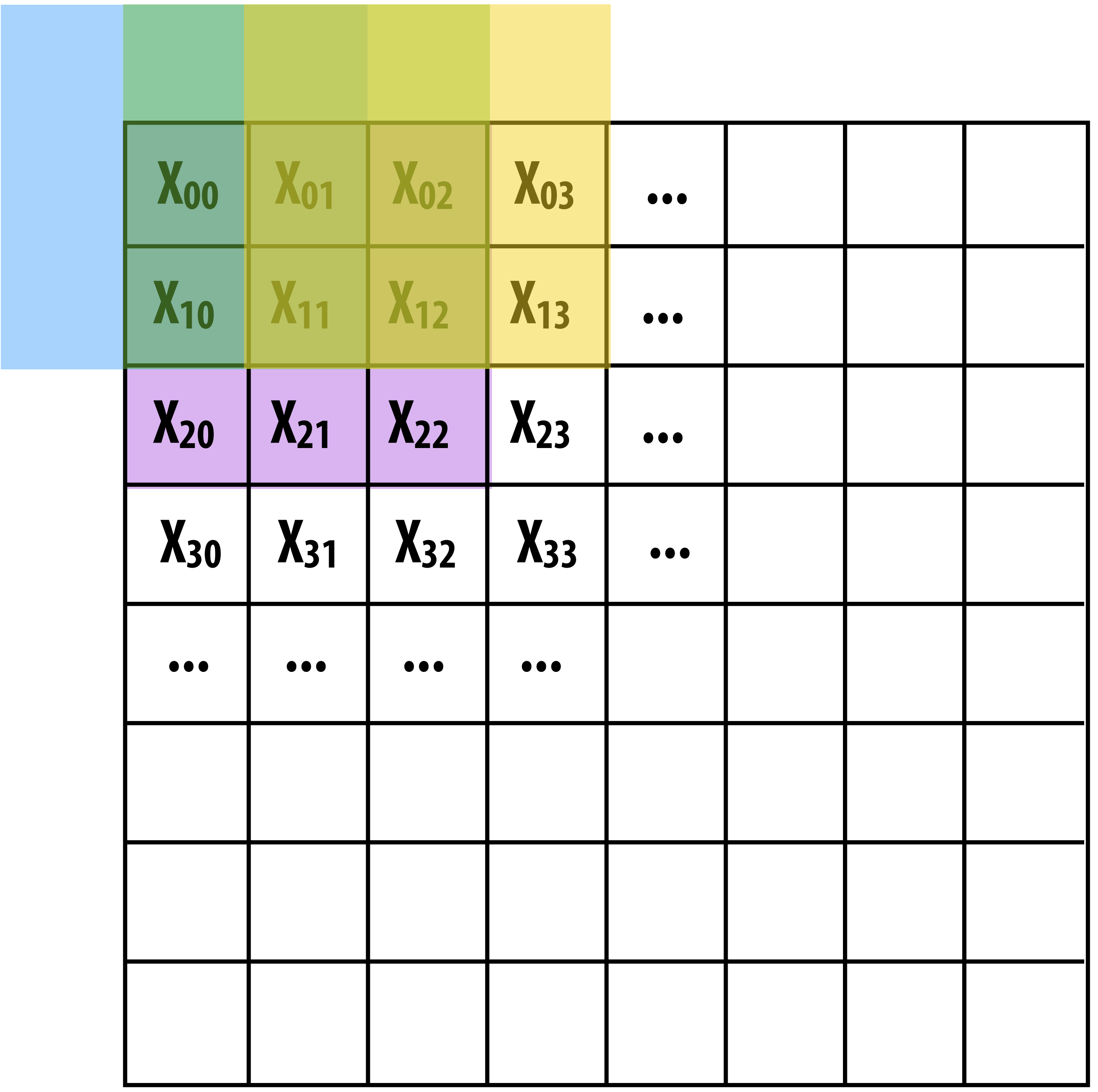
Direct implementation of conv layer (batched)

```
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)

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}	...			
...	...	...	...				

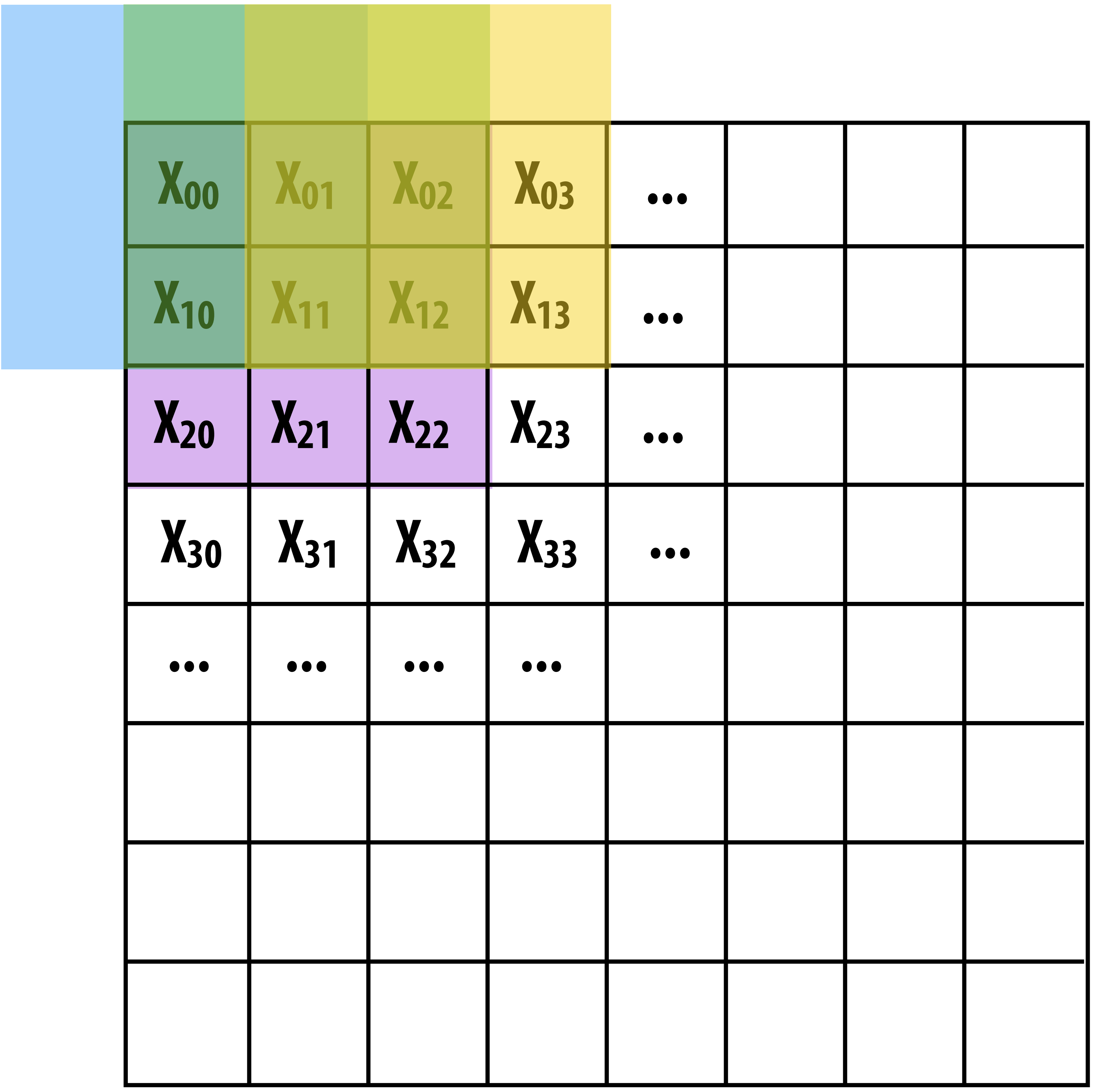
O(N) storage overhead for filter with N elements

Must construct input data matrix

$$\begin{array}{cc} & \overbrace{\hphantom{0\ 0\ 0\ 0\ x00\ x01\ 0\ x10\ x11}}^{9} \\ \text{WxH} & \begin{bmatrix} 0 & 0 & 0 & 0 & x00 & x01 & 0 & x10 & x11 \\ 0 & 0 & 0 & x00 & x01 & x02 & x10 & x11 & x12 \\ 0 & 0 & 0 & x01 & x02 & x03 & x11 & x12 & x13 \\ & & & & \dots & & & & \\ x00 & x01 & x02 & x10 & x11 & x12 & x20 & x21 & x22 \\ & & & & \dots & & & & \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_8 \end{bmatrix} \end{array}$$

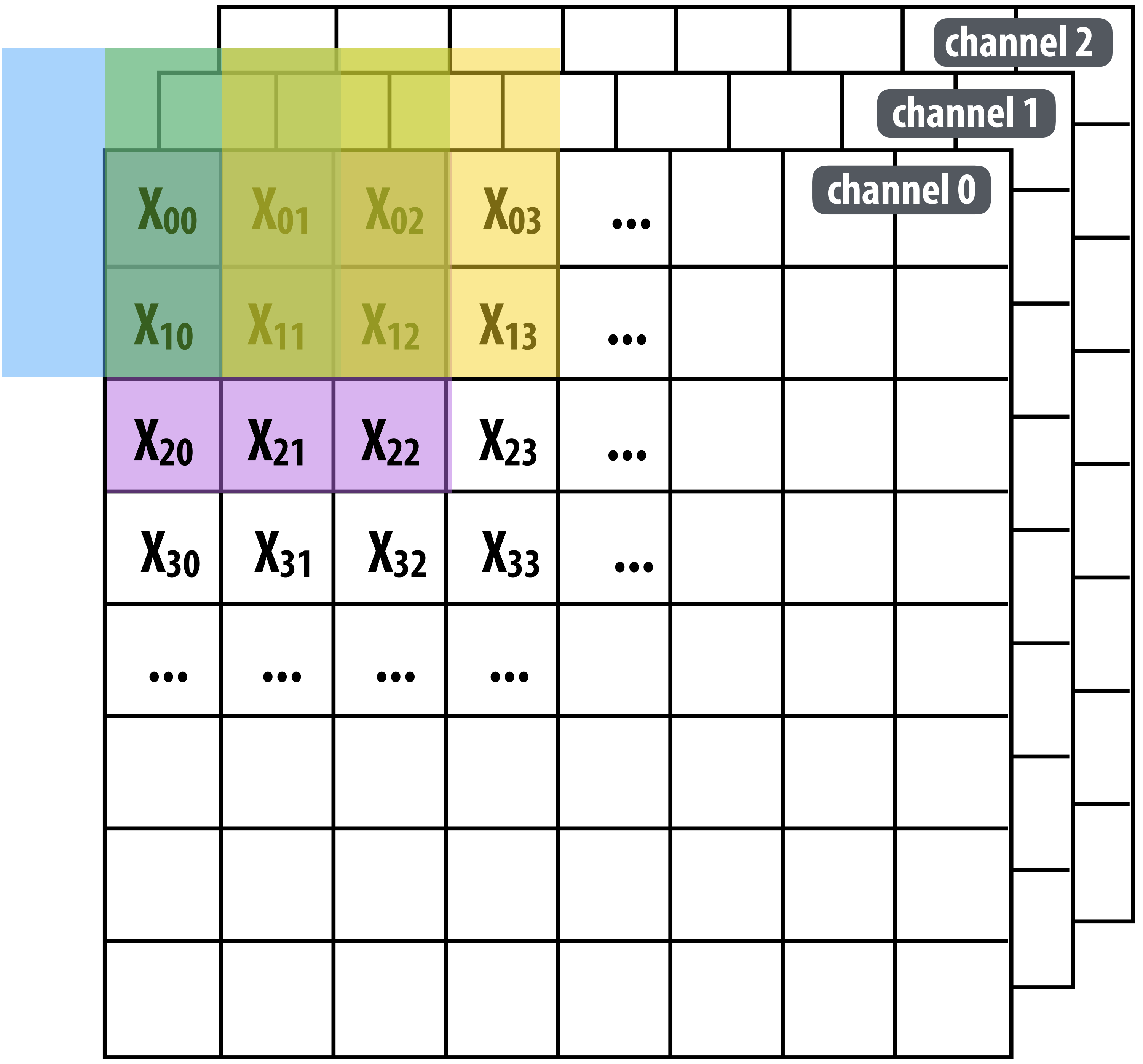
Note: 0-pad matrix

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



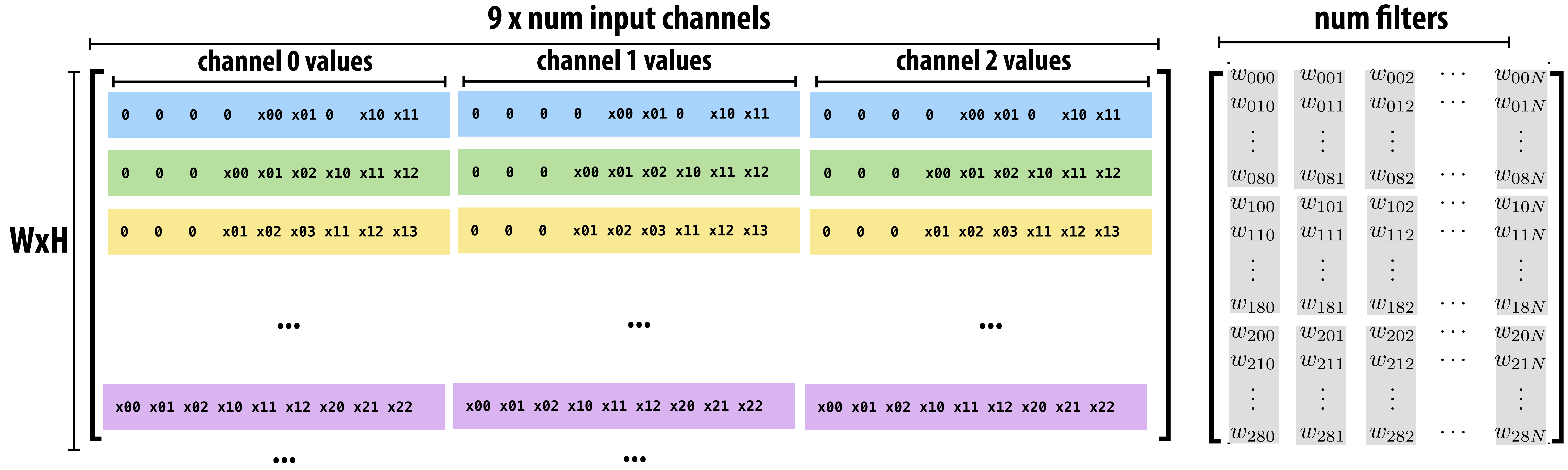
$$
\begin{array}{c|c|c}
 & \overbrace{\hspace{6em}}^{9} & \overbrace{\hspace{6em}}^{\text{num filters}} \\
\text{WxH} &
\left[\begin{array}{ccccccccc}
0 & 0 & 0 & 0 & x00 & x01 & 0 & x10 & x11 \\
0 & 0 & 0 & x00 & x01 & x02 & x10 & x11 & x12 \\
0 & 0 & 0 & x01 & x02 & x03 & x11 & x12 & x13 \\
 & & & & \ldots & & & & \\
x00 & x01 & x02 & x10 & x11 & x12 & x20 & x21 & x22 \\
 & & & & \ldots & & & &
\end{array}\right]
&
\left[\begin{array}{ccccc}
w_{00} & w_{01} & w_{02} & \cdots & w_{0N} \\
w_{10} & w_{11} & w_{12} & \cdots & w_{0N} \\
\vdots & \vdots & \vdots & & \vdots \\
w_{80} & w_{81} & w_{82} & \cdots & w_{8N}
\end{array}\right]
\end{array}
$$

Multiple convolutions on multiple input channels



For each filter, sum responses over input channels

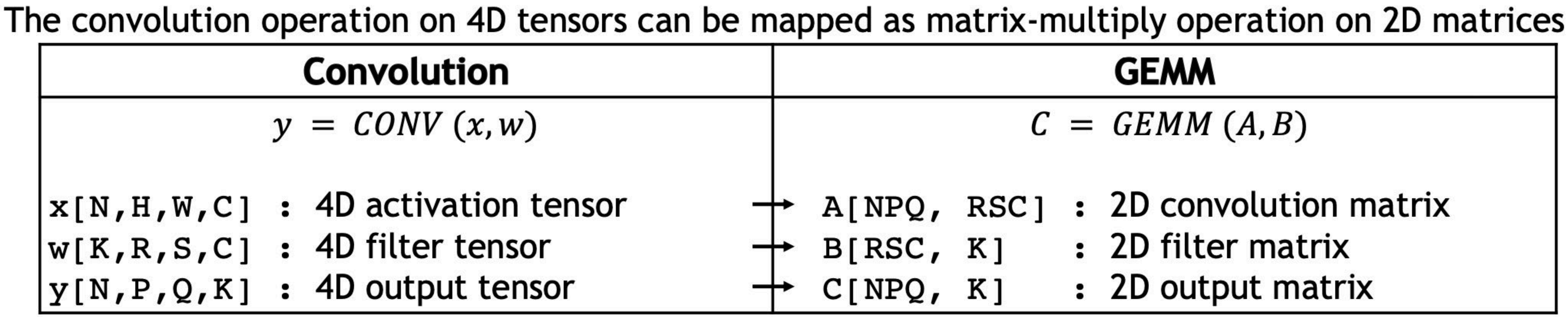
Equivalent to (3 x 3 x num_channels) convolution on (W x H x num_channels) input data



channel 0 values	channel 1 values	channel 2 values
0 0 0 0 x00 x01 0 x10 x11	0 0 0 0 x00 x01 0 x10 x11	0 0 0 0 x00 x01 0 x10 x11
0 0 0 x00 x01 x02 x10 x11 x12	0 0 0 x00 x01 x02 x10 x11 x12	0 0 0 x00 x01 x02 x10 x11 x12
0 0 0 x01 x02 x03 x11 x12 x13	0 0 0 x01 x02 x03 x11 x12 x13	0 0 0 x01 x02 x03 x11 x12 x13
...	...	...
x00 x01 x02 x10 x11 x12 x20 x21 x22	x00 x01 x02 x10 x11 x12 x20 x21 x22	x00 x01 x02 x10 x11 x12 x20 x21 x22
...	...	

$$\begin{bmatrix} w_{000} & w_{001} & w_{002} & \cdots & w_{00N} \\ w_{010} & w_{011} & w_{012} & \cdots & w_{01N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{080} & w_{081} & w_{082} & \cdots & w_{08N} \\ w_{100} & w_{101} & w_{102} & \cdots & w_{10N} \\ w_{110} & w_{111} & w_{112} & \cdots & w_{11N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{180} & w_{181} & w_{182} & \cdots & w_{18N} \\ w_{200} & w_{201} & w_{202} & \cdots & w_{20N} \\ w_{210} & w_{211} & w_{212} & \cdots & w_{21N} \\ \vdots & \vdots & \vdots & & \vdots \\ w_{280} & w_{281} & w_{282} & \cdots & w_{28N} \end{bmatrix}$$

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 x S
Input channels: C
Number of filters: K
Batch size: N

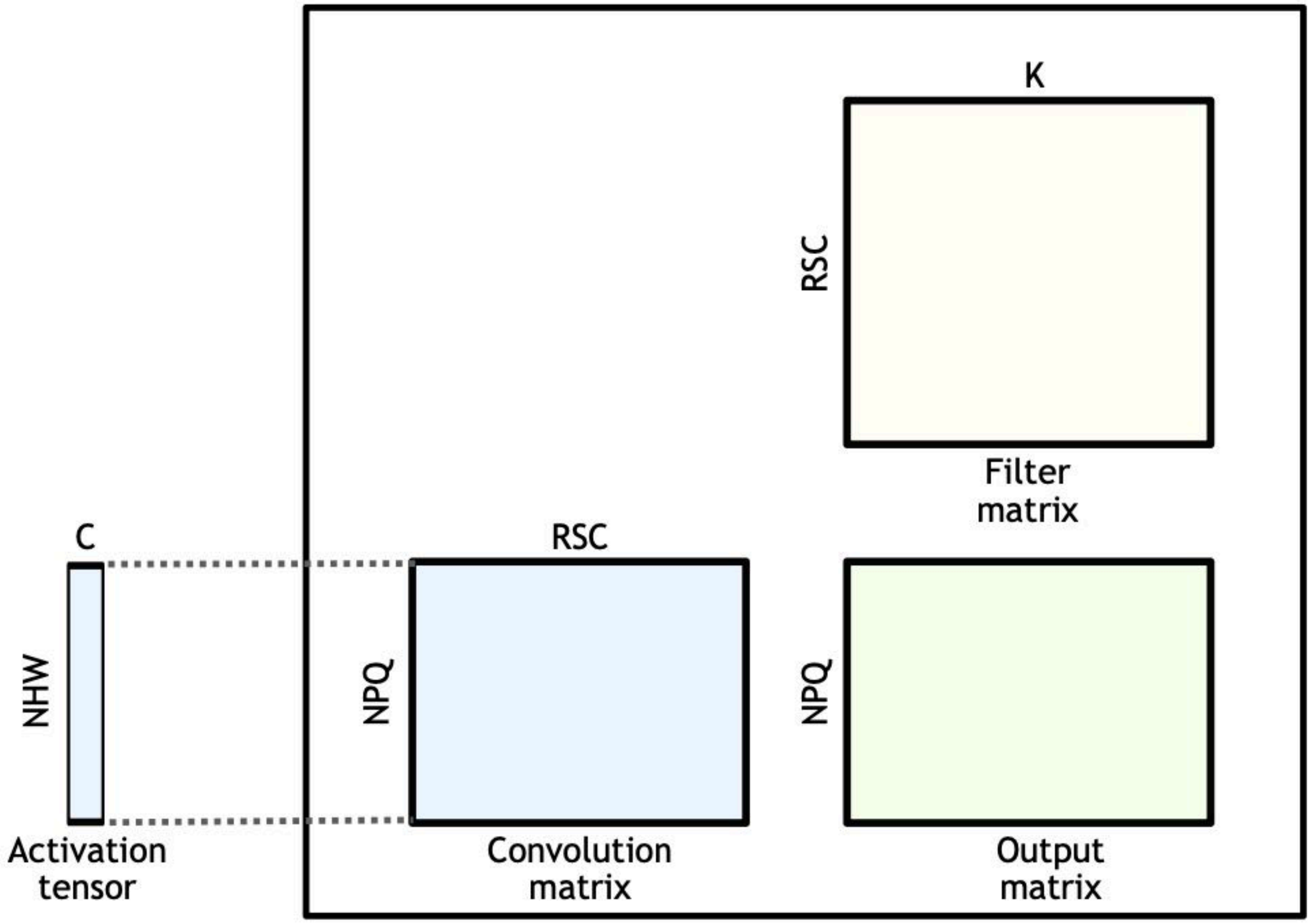
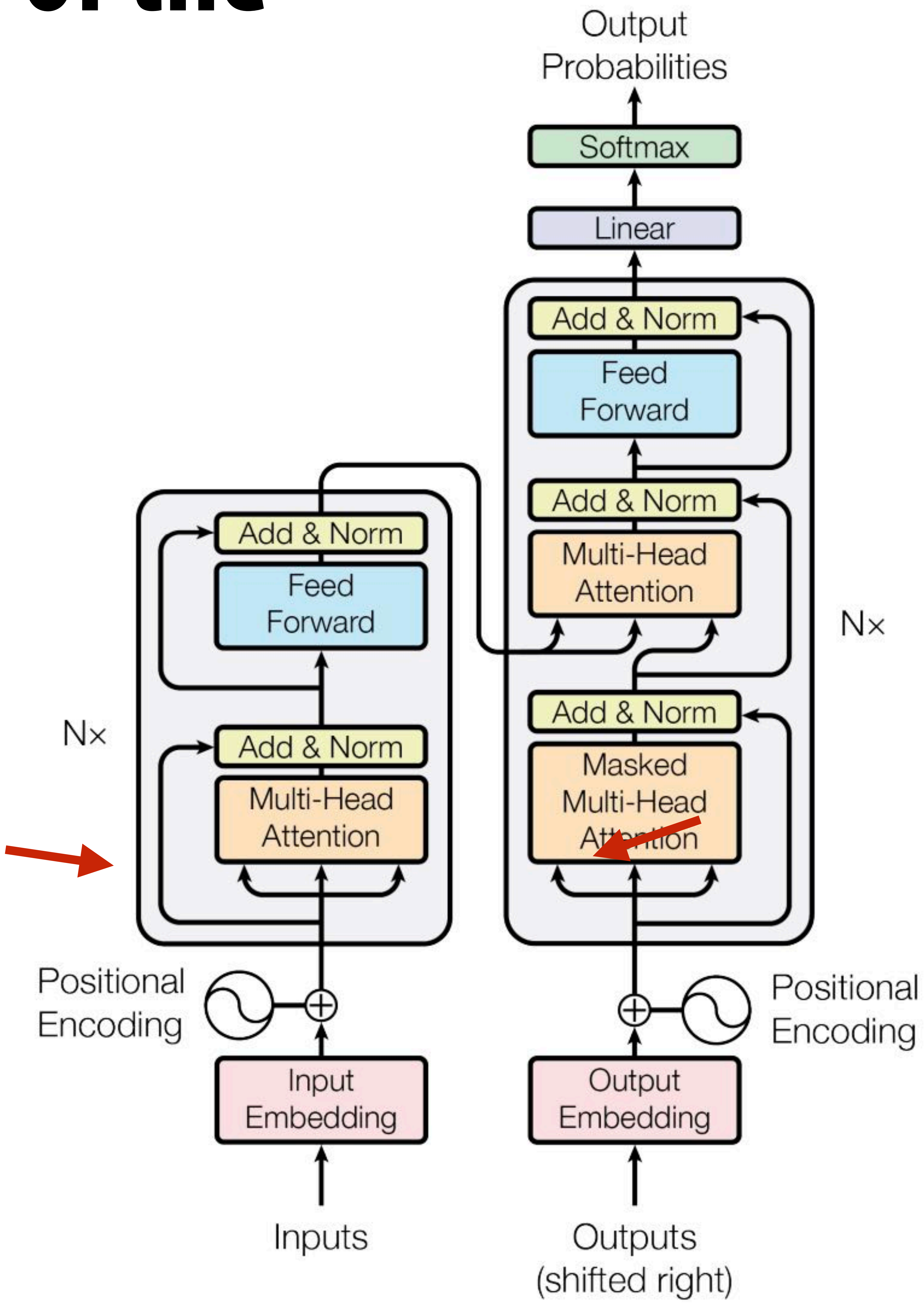


Image credit: NVIDIA

Matrix multiplication is also at the heart of the “attention” blocks of a transformer architecture

Matrix multiplication is at the heart of the "attention" blocks of a transformer architecture

Sequence of tokens in, sequence of tokens out

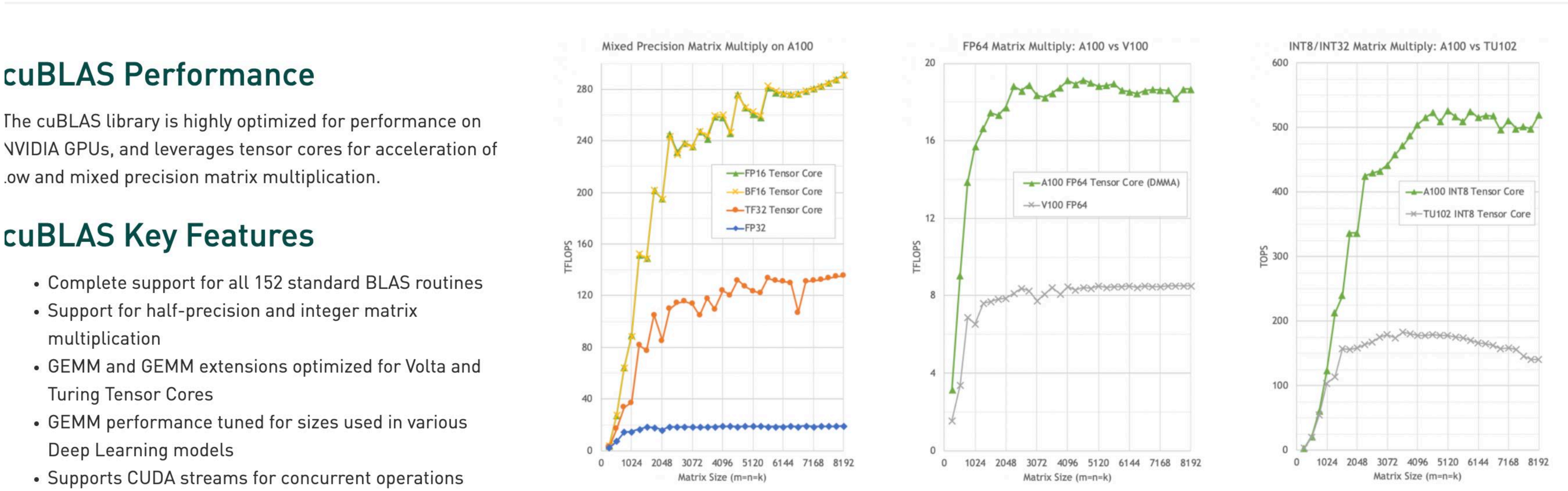


The importance of dense matrix-matrix multiplication (GEMM) to modern AI

The kernel for…

- **Fully-connected layers**
- **Convolutional layers**
- **The attention block of a transformer**

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.

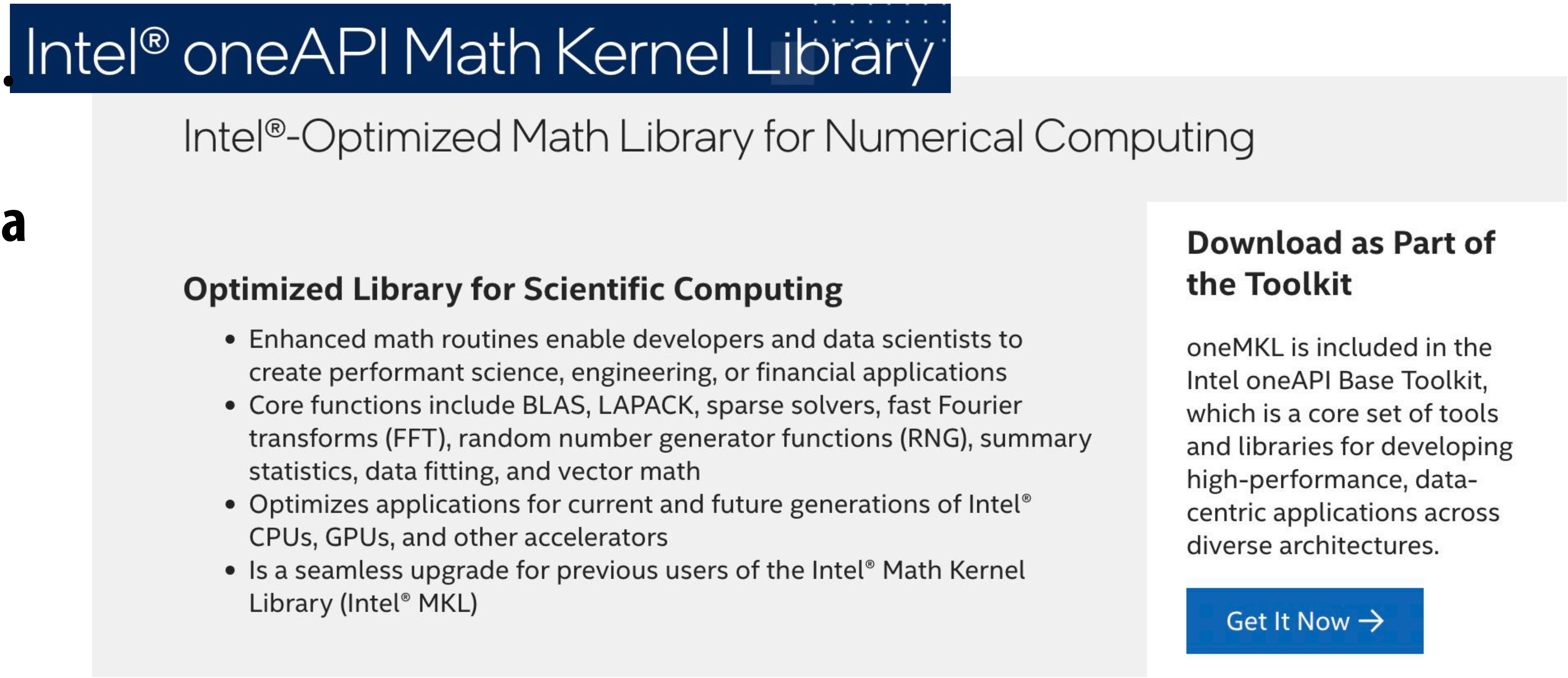
cuBLAS Key Features

- 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.

For convolutional layer implications, Increases DRAM traffic by a factor of R x S
(To read input data from activation tensor and constitute "convolution matrix")

Also requires large amount of additional storage



Intel® oneAPI Math Kernel Library

Intel®-Optimized Math Library for Numerical Computing

Optimized Library for Scientific Computing

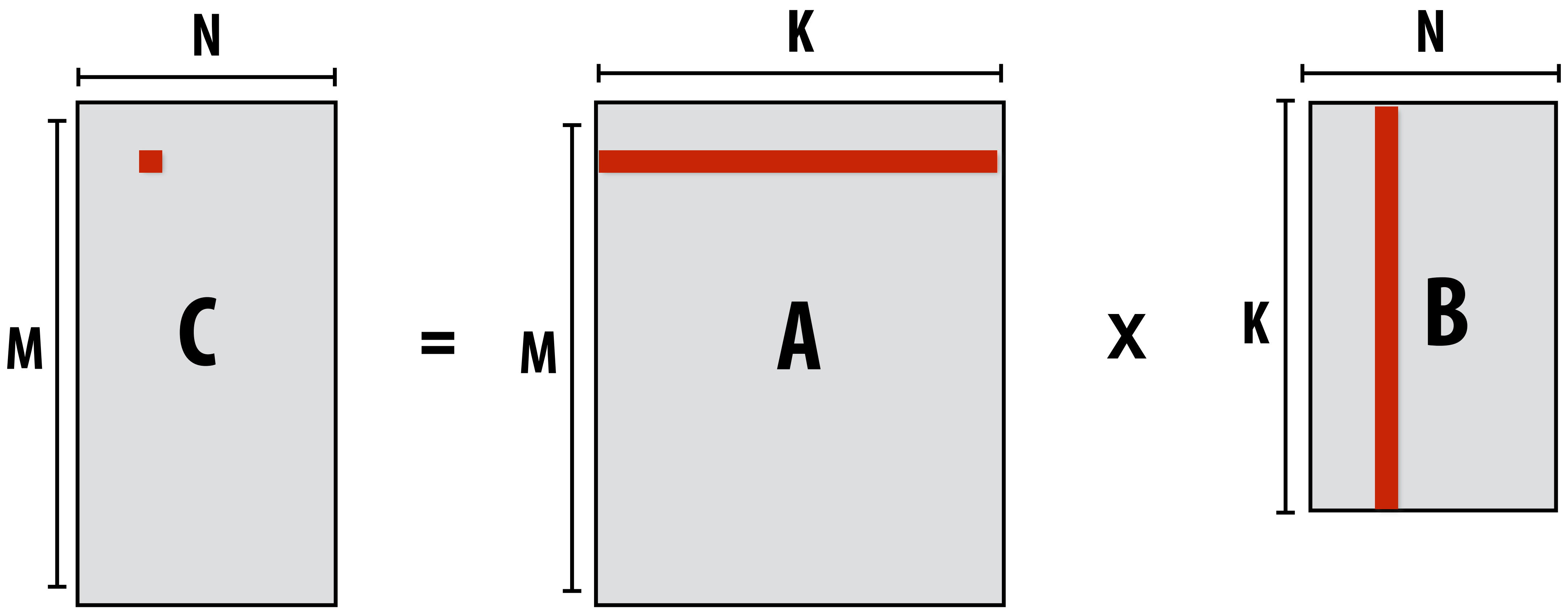
- Enhanced math routines enable developers and data scientists to create performant science, engineering, or financial applications
- Core functions include BLAS, LAPACK, sparse solvers, fast Fourier transforms (FFT), random number generator functions (RNG), summary statistics, data fitting, and vector 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.

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];
```

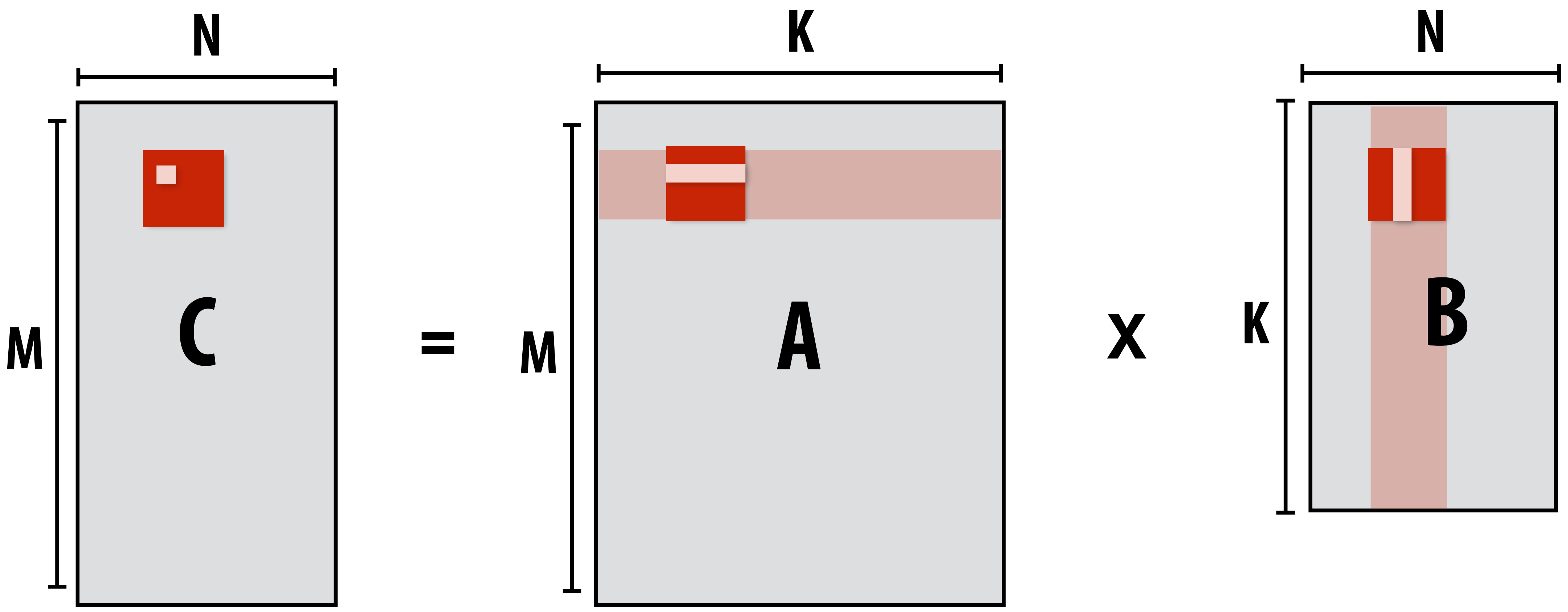


What is the problem with this implementation?

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

Increasing arithmetic intensity by “blocking”

```
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

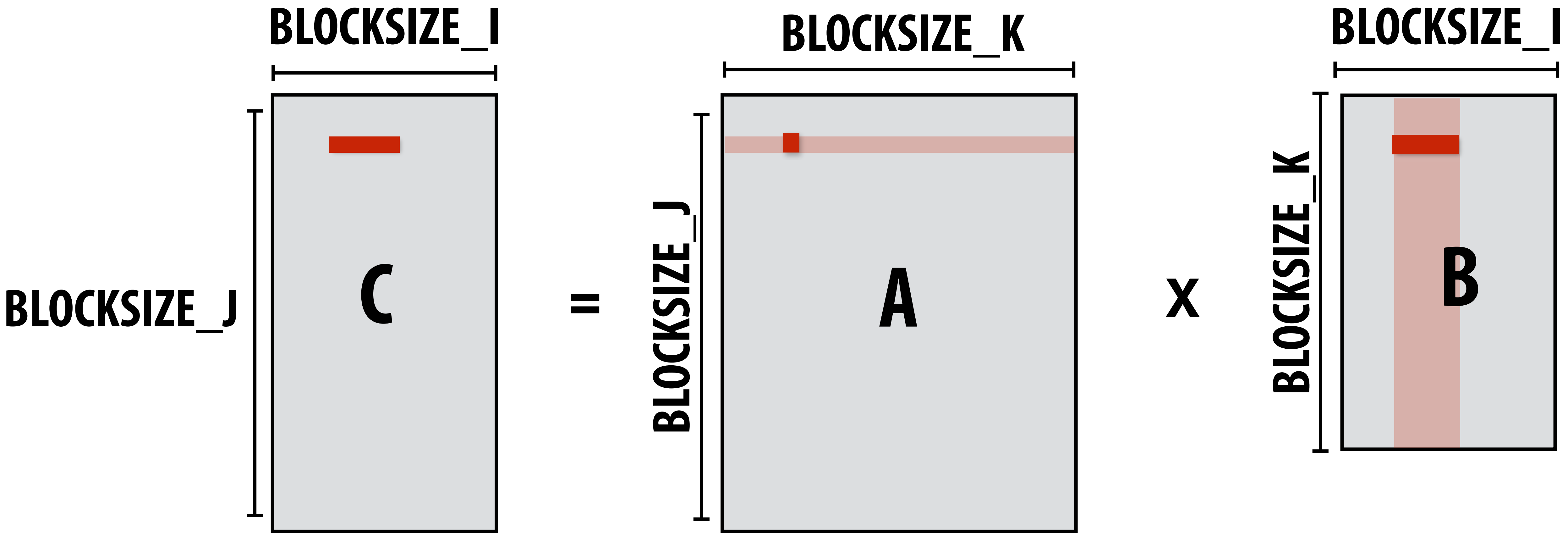
Exploit multiple levels of memory hierarchy (increase arithmetic intensity when considering multiple levels of memory hierarchy)

```
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…

Vectorized, blocked dense matrix multiplication (1)

Consider SIMD parallelism within a block



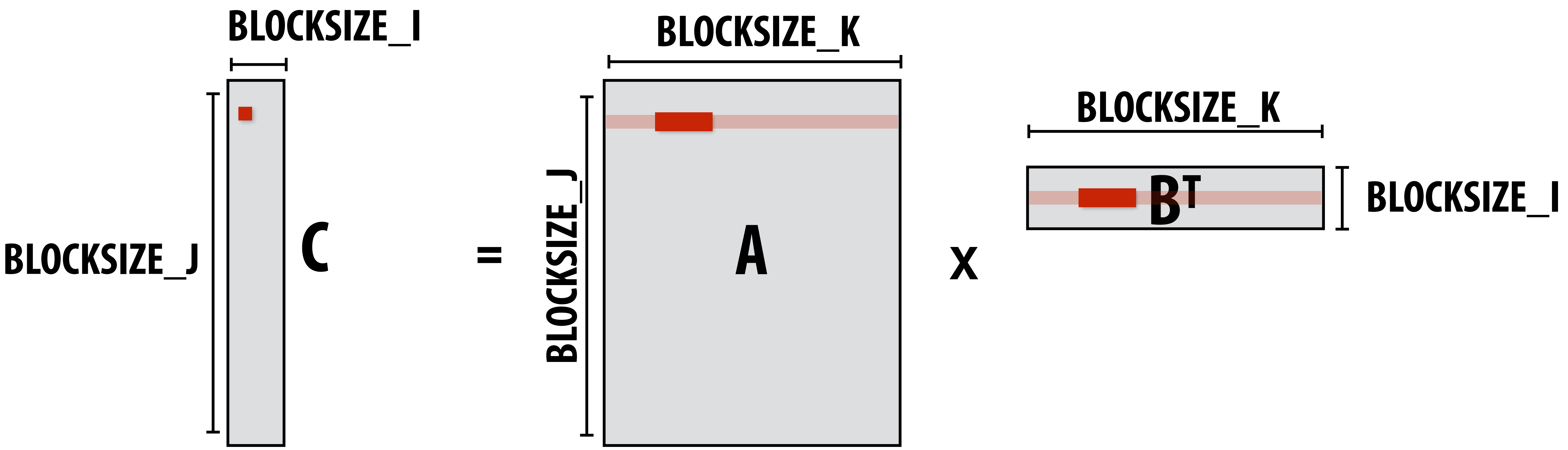
```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
      simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in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

Vectorized, blocked dense matrix multiplication (2)



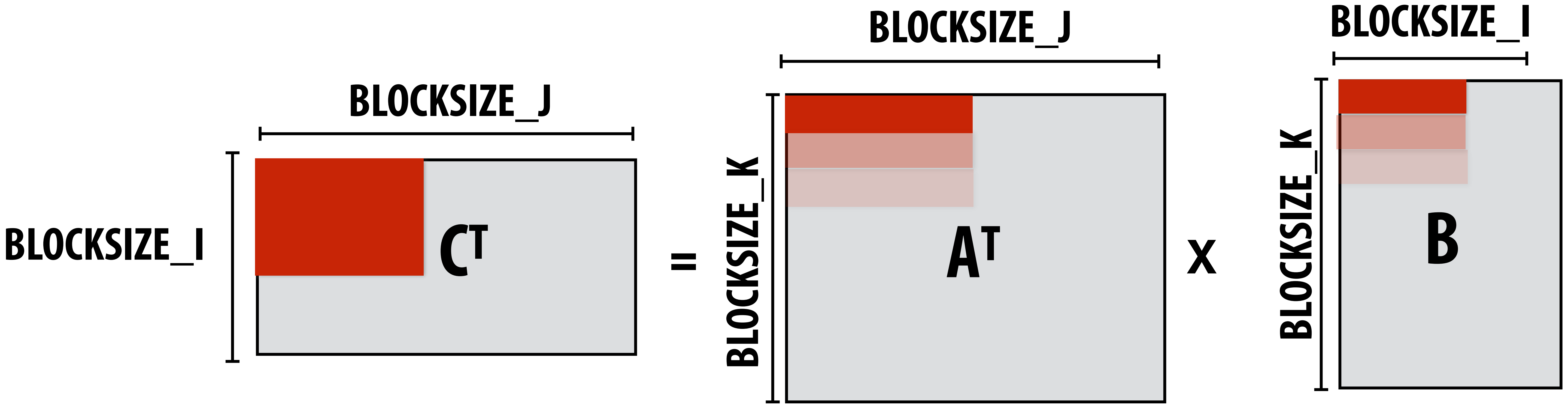
```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      // C_scalar += dot(row of A,row of B)
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
        C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][[kblock+k]);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

Vectorized, blocked dense matrix multiplication (3)



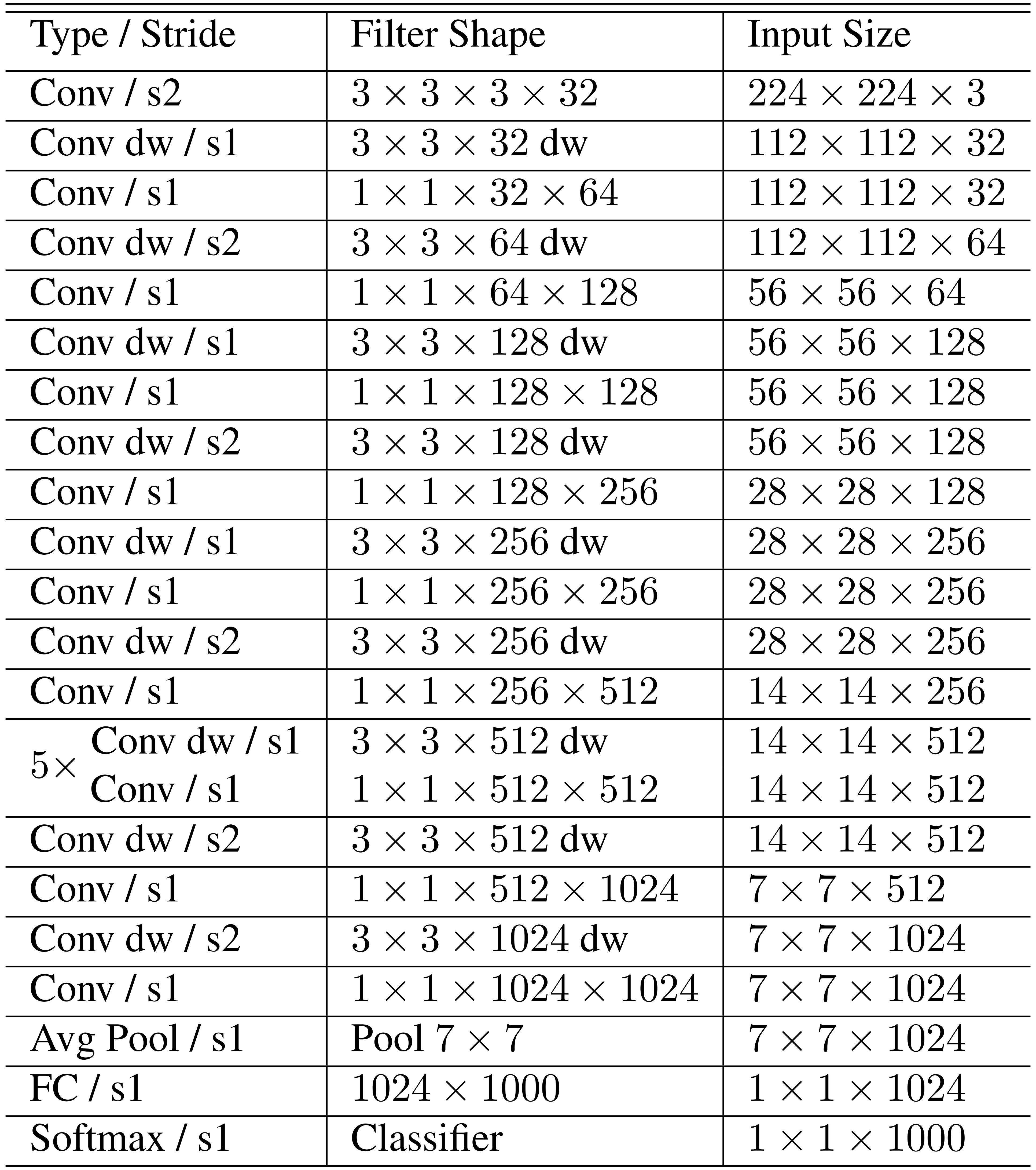
```
// assume blocks of A and C are pre-transposed as Atrans and Ctrans
for (int j=0; j<BLOCKSIZE_J; j+=SIMD_WIDTH) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {

      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]);
   }
}
```

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



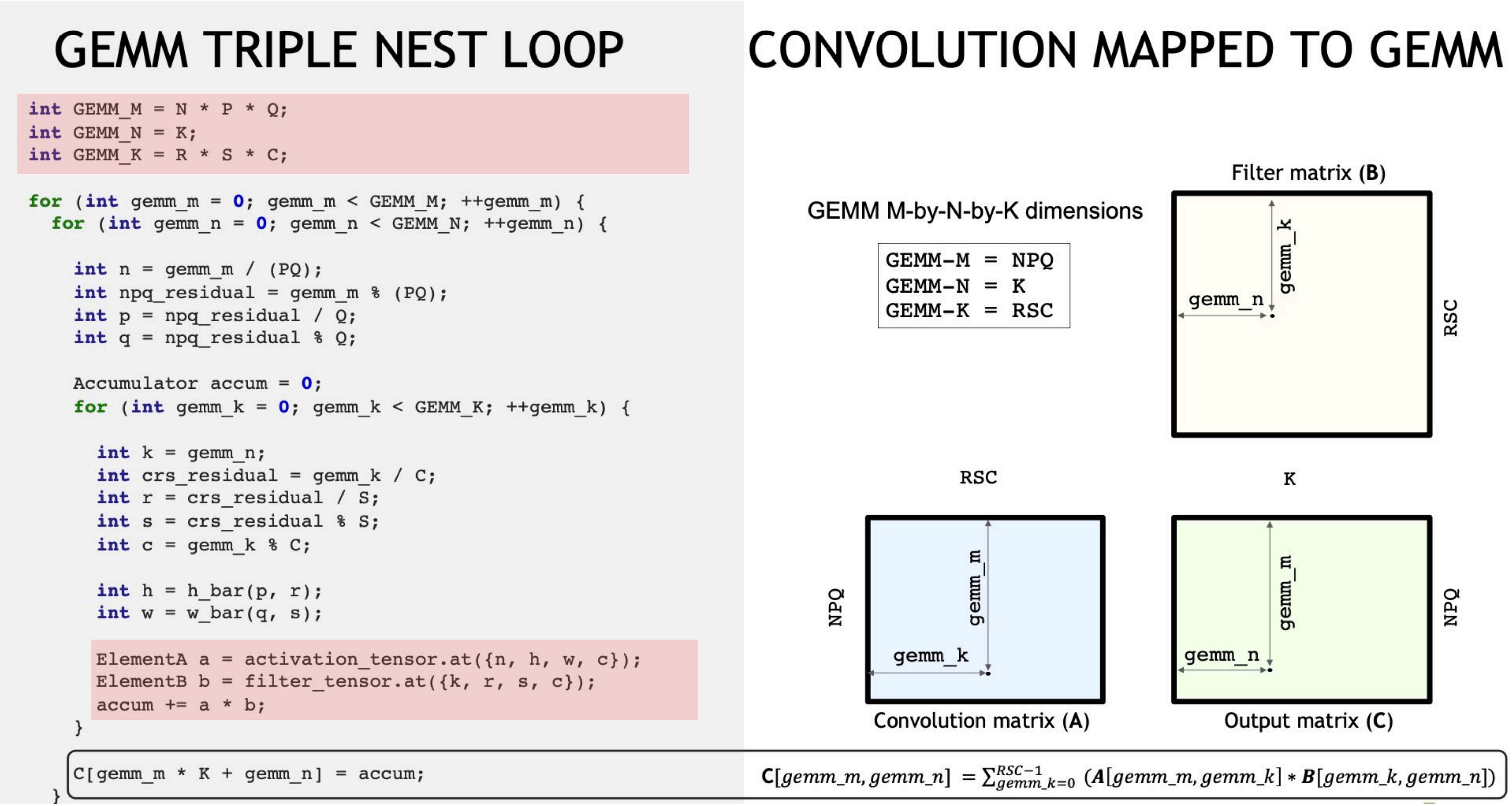
Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 ×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

Matrix multiplication implementations

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 x S
Input channels: C
Number of filters: K
Batch size: N



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

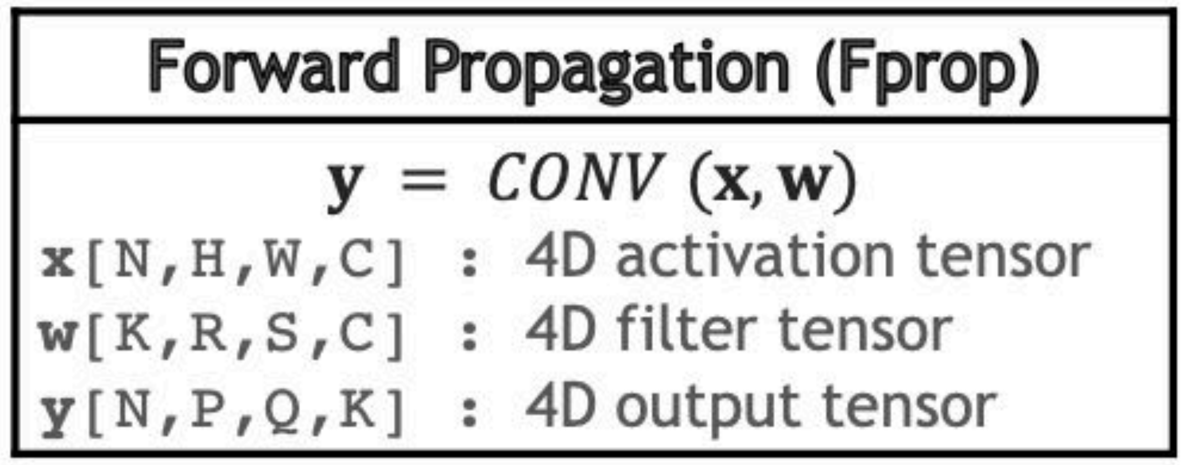
```
GEMM-M = NPQ
GEMM-N = K
GEMM-K = RSC
```

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

Image credit: NVIDIA

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

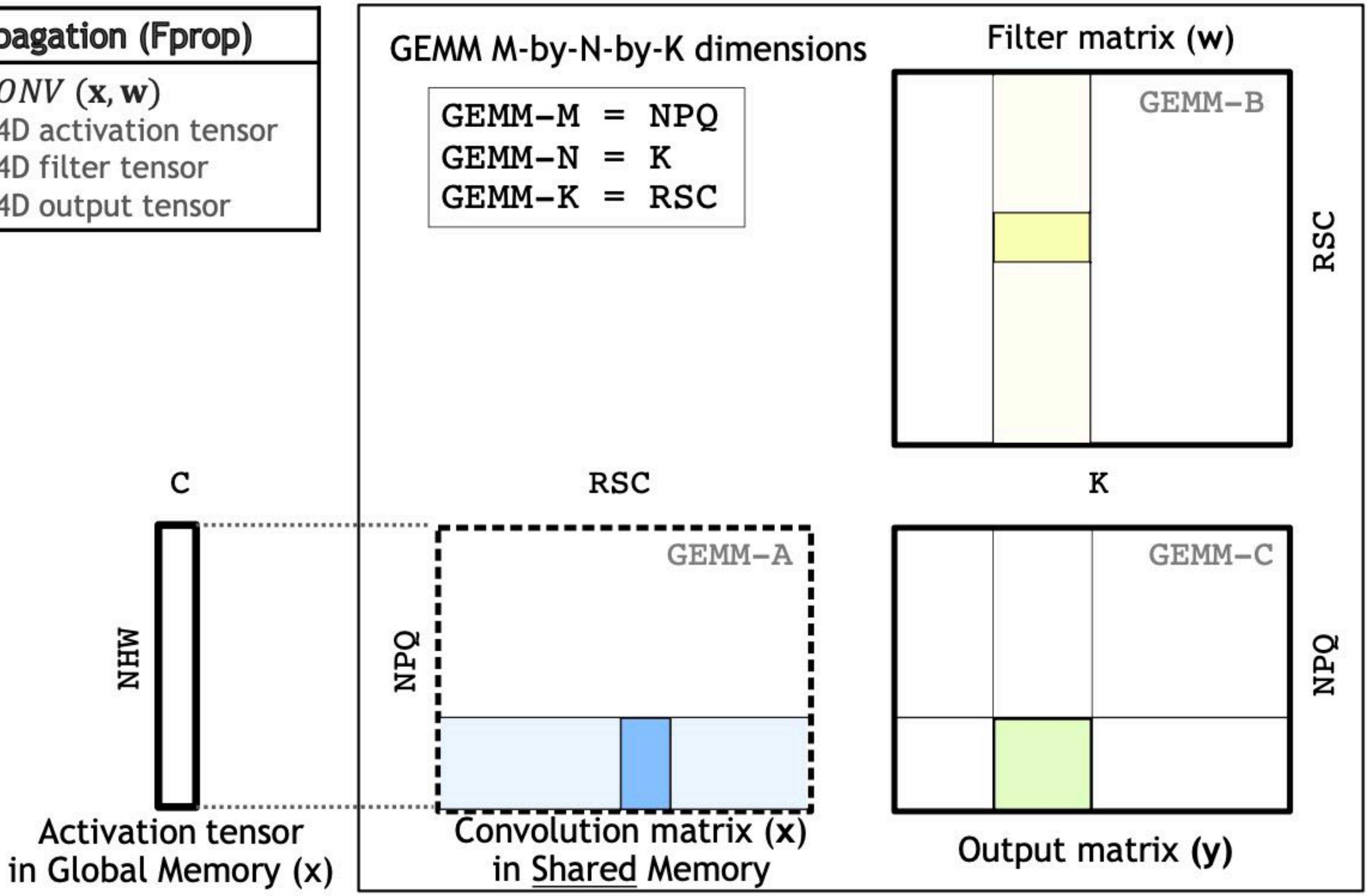
Better implementation: materialize only a sub-block of the convolution matrix at a time in GPU on-chip "shared memory"



Forward Propagation (Fprop)

$y = CONV(\mathbf{x}, \mathbf{w})$

`x[N,H,W,C]` : 4D activation tensor
`w[K,R,S,C]` : 4D filter tensor
`y[N,P,Q,K]` : 4D output tensor



GEMM M-by-N-by-K dimensions

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

Filter matrix (w)
GEMM-B
RSC
K
RSC
C
NHW
GEMM-A
NPQ
GEMM-C
NPQ
Activation tensor in Global Memory (x)
Convolution matrix (x) in Shared Memory
Output matrix (y)

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

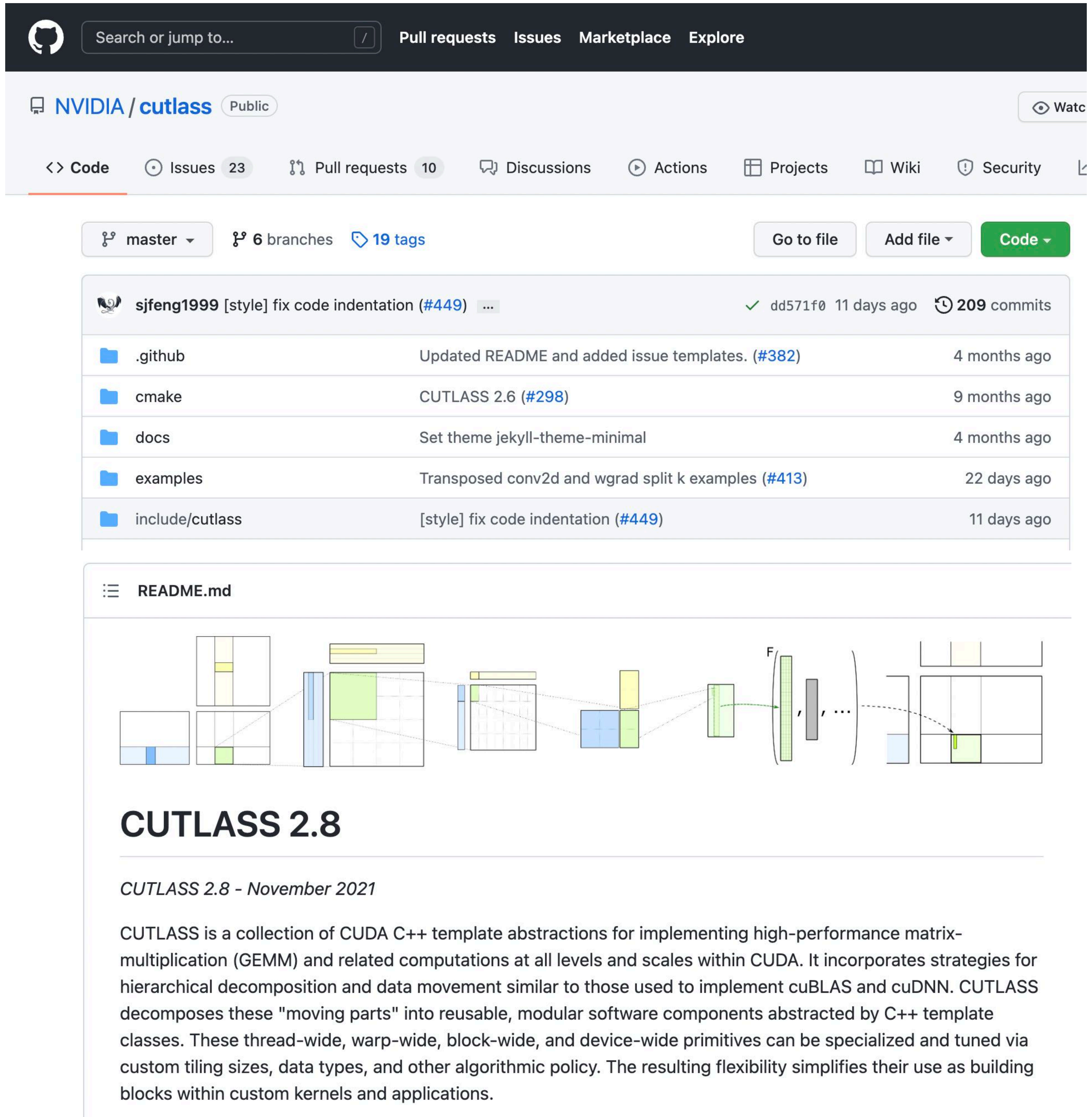
Use well-tuned shared-memory based GEMM routines to perform sub-block GEMM (see CUTLASS)

Symbol reference:
- **Output size: PxQ**
- **Spatial support of filters: R x S**
- **Input channels: C**
- **Number of filters (output channels): 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)



NVIDIA / cutlass Public

6 branches 19 tags

sjfeng1999 [style] fix code indentation (#449) ... ✓ dd571f0 11 days ago 209 commits

.github	Updated README and added issue templates. (#382)	4 months ago
cmake	CUTLASS 2.6 (#298)	9 months ago
docs	Set theme jekyll-theme-minimal	4 months ago
examples	Transposed conv2d and wgrad split k examples (#413)	22 days ago
include/cutlass	[style] fix code indentation (#449)	11 days ago

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

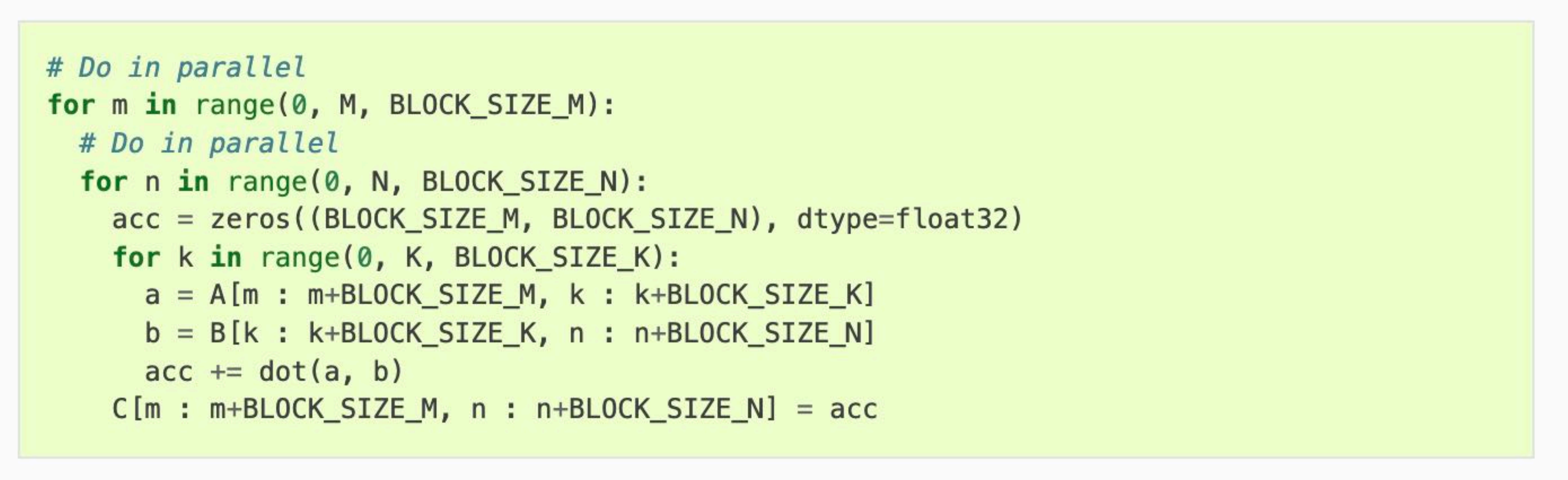
Tensor reductions

Etc.

Triton

- Language support for operations that load/store tensors
- Load “blocks” of data into GPU shared memory
- Perform data-parallel operations on those blocks

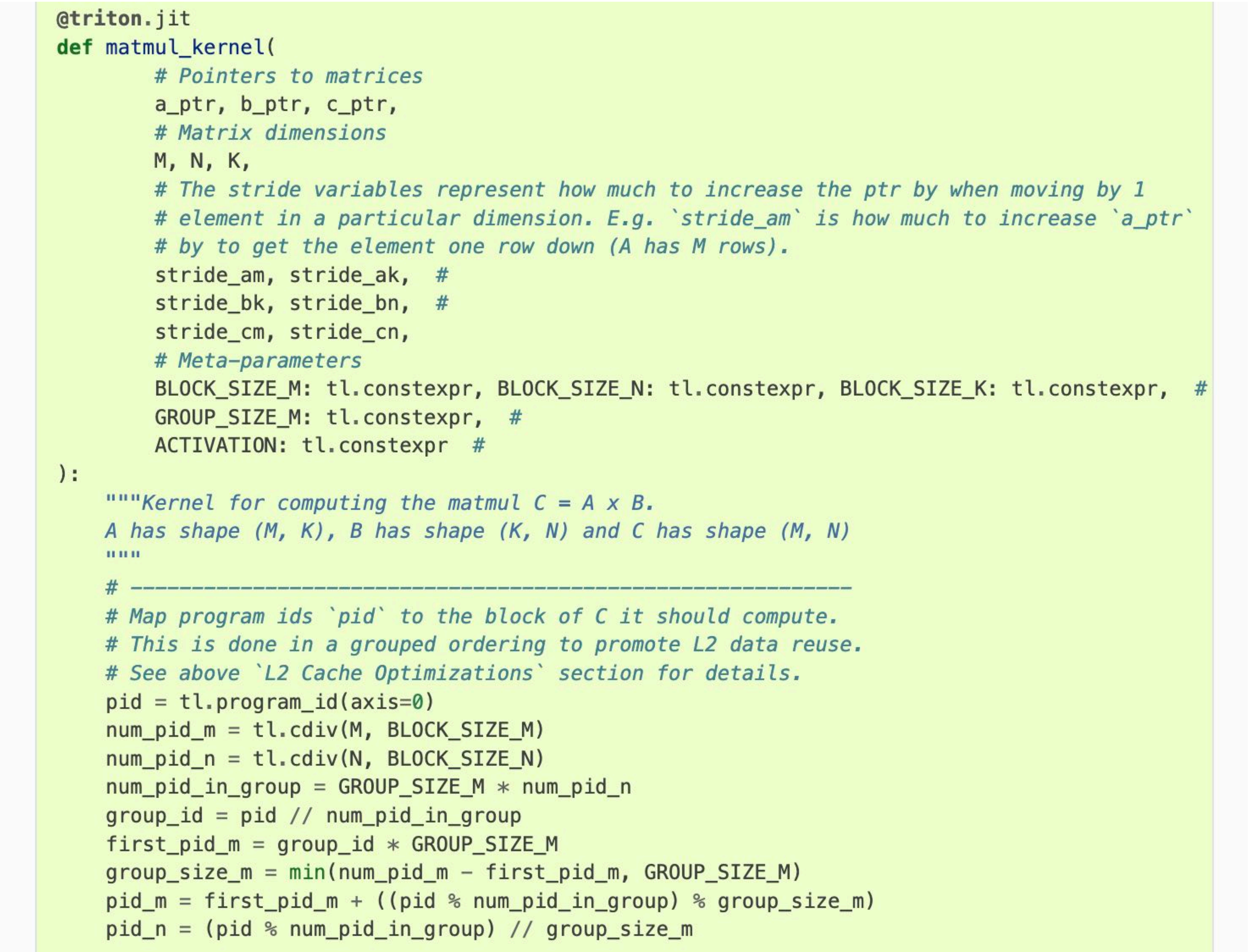
A simple blocked matrix multiplication



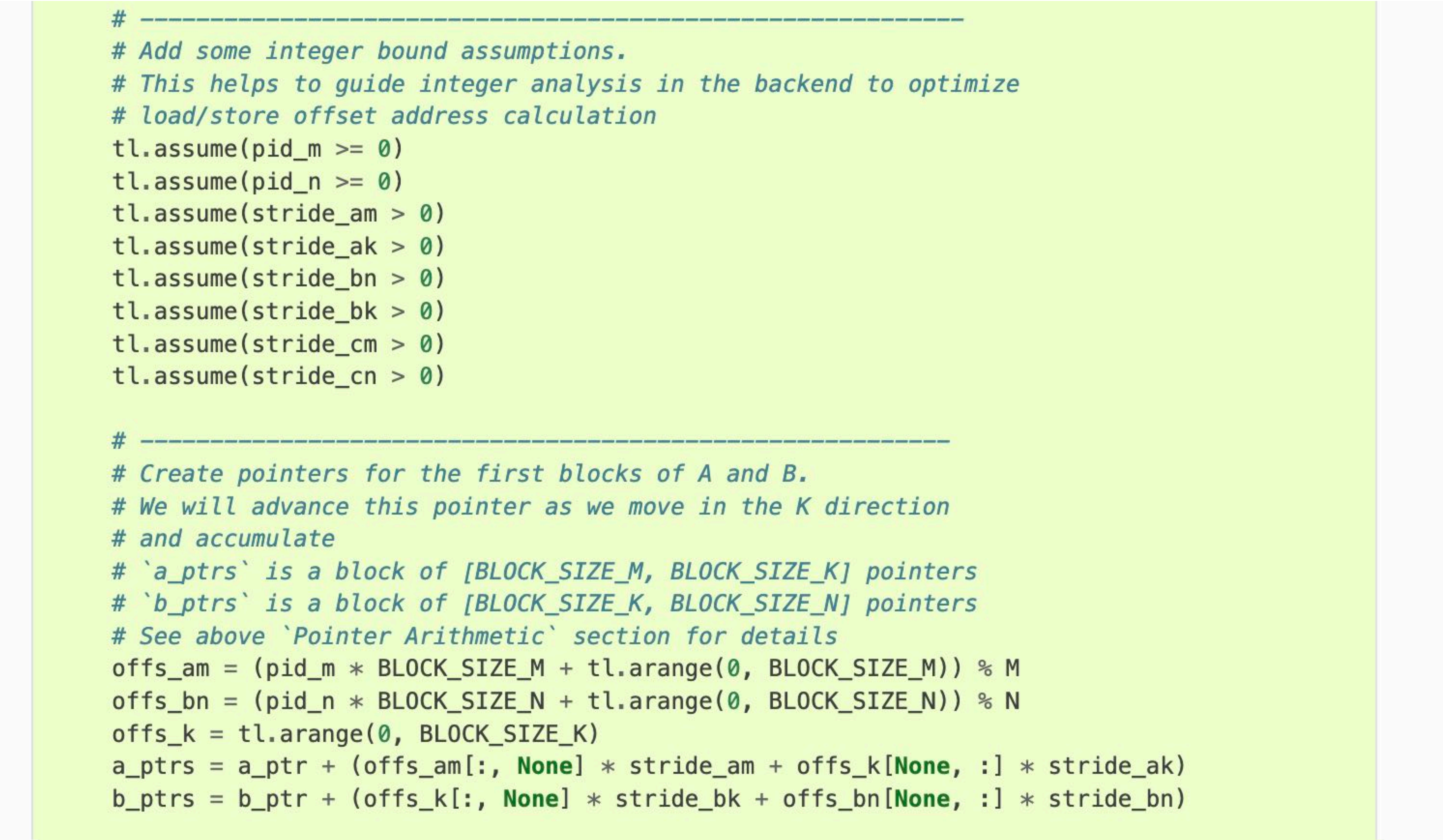
```
# Do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

Triton

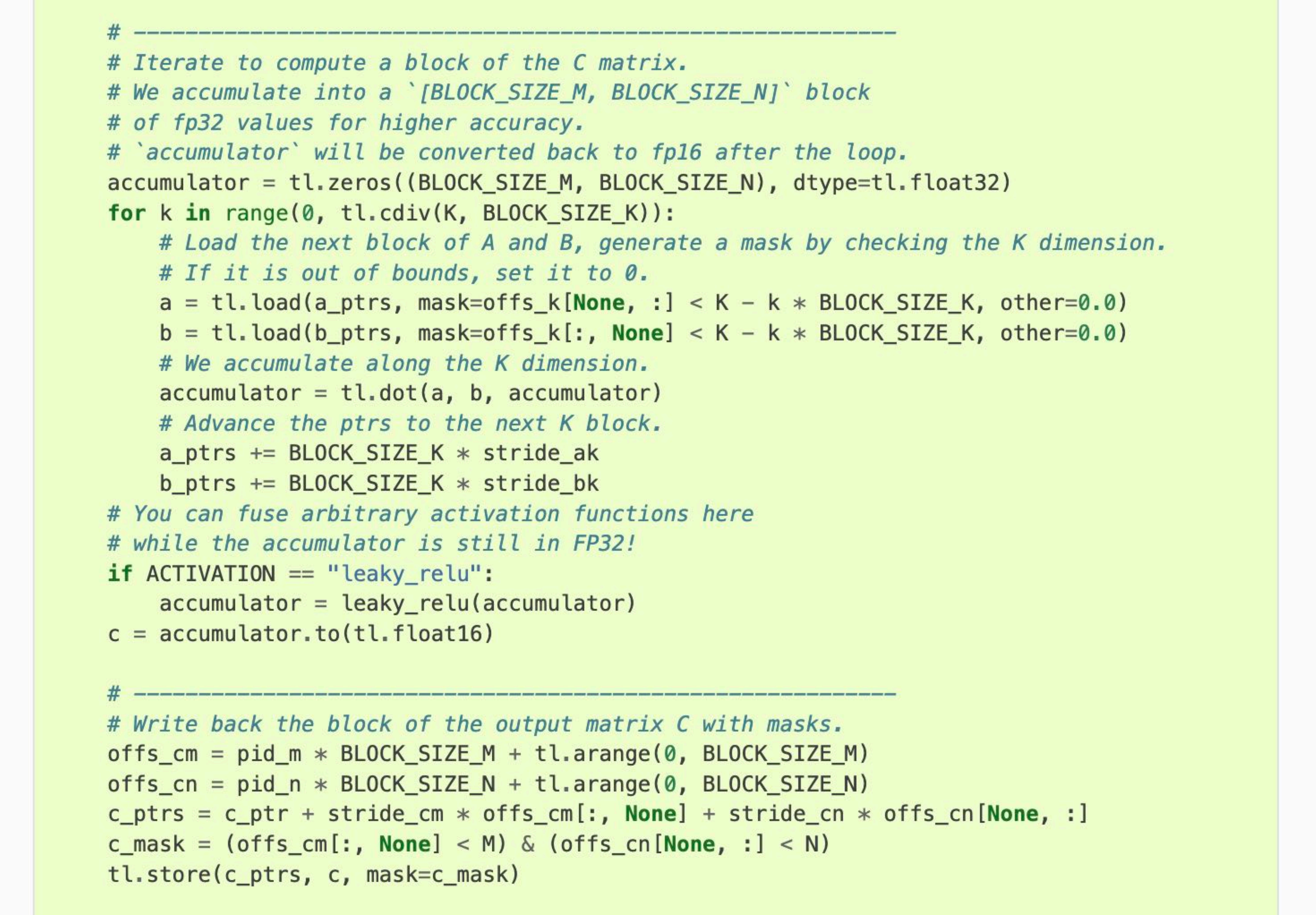
Full Triton reference implementation: two levels of blocking



```python
@triton.jit
def matmul_kernel(
        # Pointers to matrices
        a_ptr, b_ptr, c_ptr,
        # Matrix dimensions
        M, N, K,
        # The stride variables represent how much to increase the ptr by when moving by 1
        # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
        # by to get the element one row down (A has M rows).
        stride_am, stride_ak,  #
        stride_bk, stride_bn,  #
        stride_cm, stride_cn,
        # Meta-parameters
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,  #
        GROUP_SIZE_M: tl.constexpr,  #
        ACTIVATION: tl.constexpr  #
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    # -----------------------------------------------------------
    # Map program ids `pid` to the block of C it should compute.
    # This is done in a grouped ordering to promote L2 data reuse.
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m
```



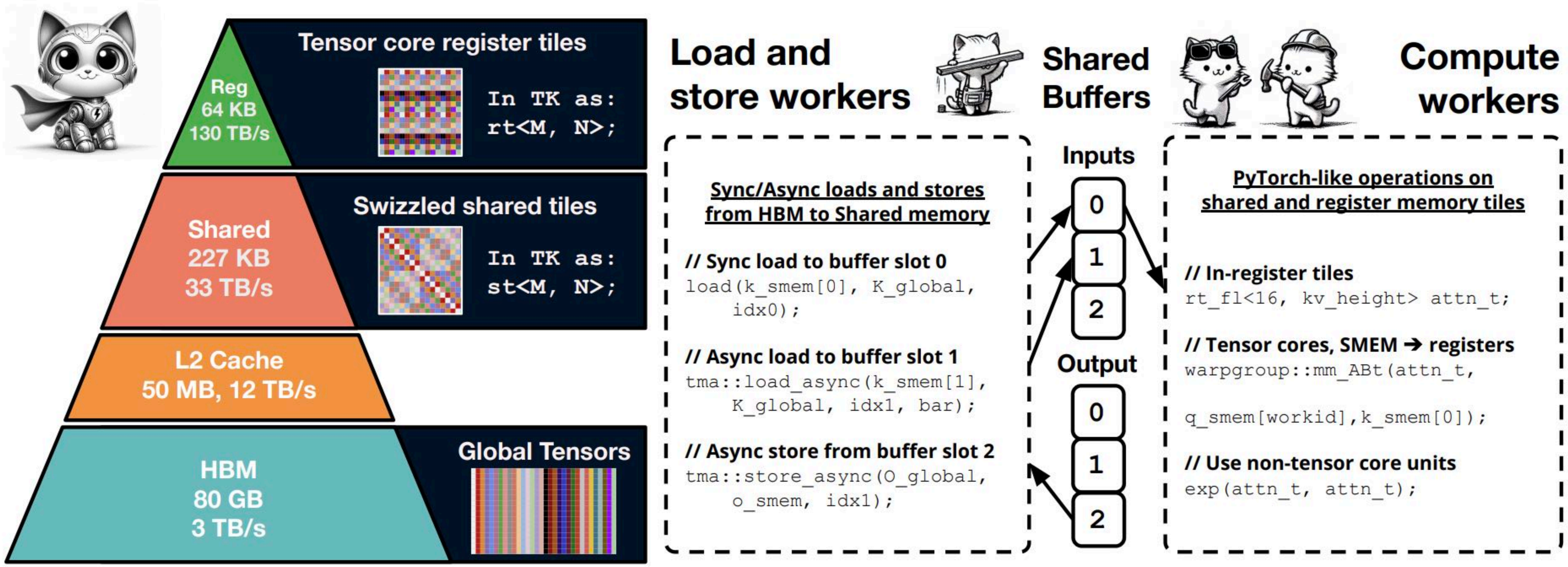
```python
    # -----------------------------------------------------------
    # Add some integer bound assumptions.
    # This helps to guide integer analysis in the backend to optimize
    # load/store offset address calculation
    tl.assume(pid_m >= 0)
    tl.assume(pid_n >= 0)
    tl.assume(stride_am > 0)
    tl.assume(stride_ak > 0)
    tl.assume(stride_bn > 0)
    tl.assume(stride_bk > 0)
    tl.assume(stride_cm > 0)
    tl.assume(stride_cn > 0)

    # ----------------------------------------------------------
    # Create pointers for the first blocks of A and B.
    # We will advance this pointer as we move in the K direction
    # and accumulate
    # `a_ptrs` is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
    # `b_ptrs` is a block of [BLOCK_SIZE_K, BLOCK_SIZE_N] pointers
    # See above `Pointer Arithmetic` section for details
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
```



```python
    # -----------------------------------------------------------
    # Iterate to compute a block of the C matrix.
    # We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
    # `accumulator` will be converted back to fp16 after the loop.
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking the K dimension.
        # If it is out of bounds, set it to 0.
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        # We accumulate along the K dimension.
        accumulator = tl.dot(a, b, accumulator)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # You can fuse arbitrary activation functions here
    # while the accumulator is still in FP32!
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    # -----------------------------------------------------------
    # Write back the block of the output matrix C with masks.
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

Thunderkittens

- CUDA library of useful tile-based programming primitives
- Intended to make advanced developers (CS149-level folks) more productive writing blocked code
  - Async load/store of tiles
  - Support for advanced memory layouts (blocked tiles, interleaved elements, etc.)



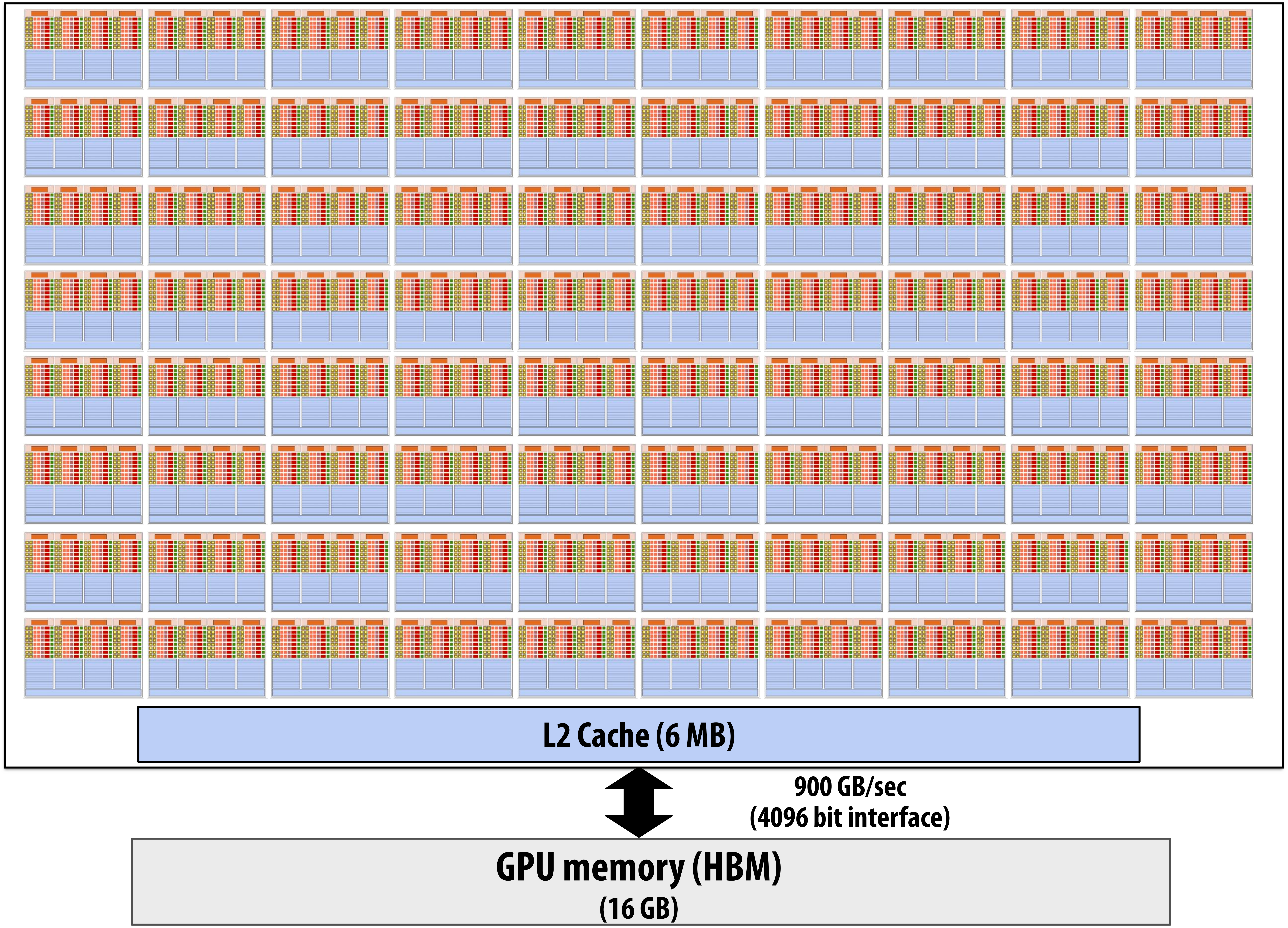
Sync/Async loads and stores from HBM to Shared memory

```
// Sync load to buffer slot 0
load(k_smem[0], K_global,
     idx0);

// Async load to buffer slot 1
tma::load_async(k_smem[1],
     K_global, idx1, bar);

// Async store from buffer slot 2
tma::store_async(O_global,
     o_smem, idx1);
```

PyTorch-like operations on shared and register memory tiles

```
// In-register tiles
rt_fl<16, kv_height> attn_t;

// Tensor cores, SMEM ➔ registers
warpgroup::mm_ABt(attn_t,

q_smem[workid],k_smem[0]);

// Use non-tensor core units
exp(attn_t, attn_t);
```

Recall: NVIDIA V100 GPU (80 SMs)



L2 Cache (6 MB)

900 GB/sec
(4096 bit interface)

GPU memory (HBM)
(16 GB)

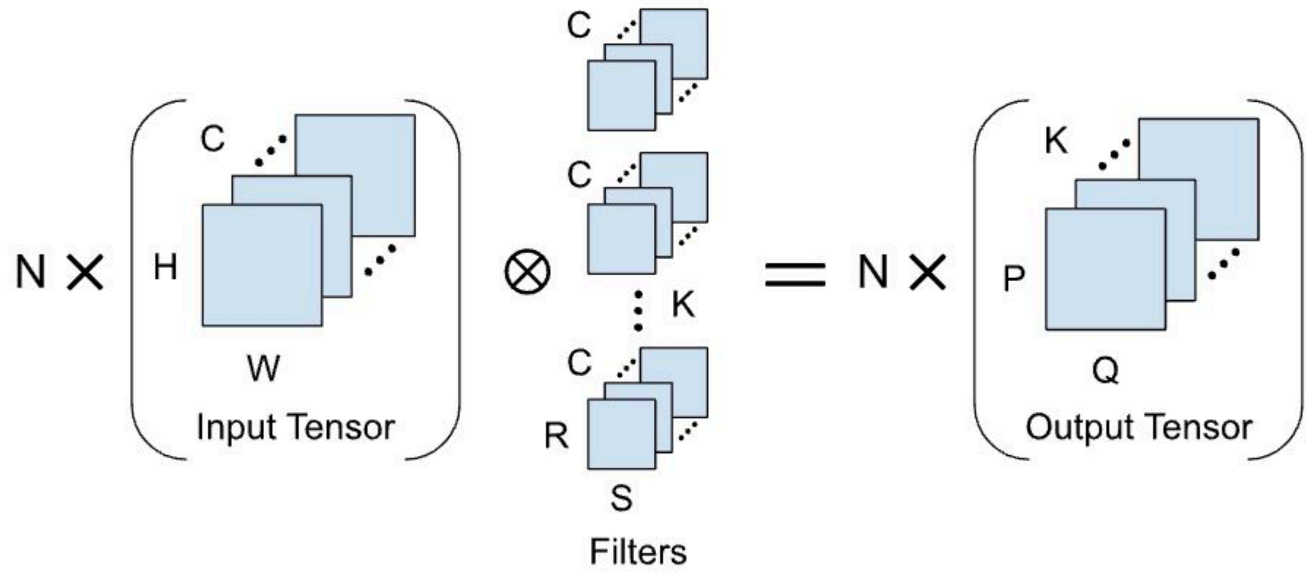
Many processing units and many tensor cores.

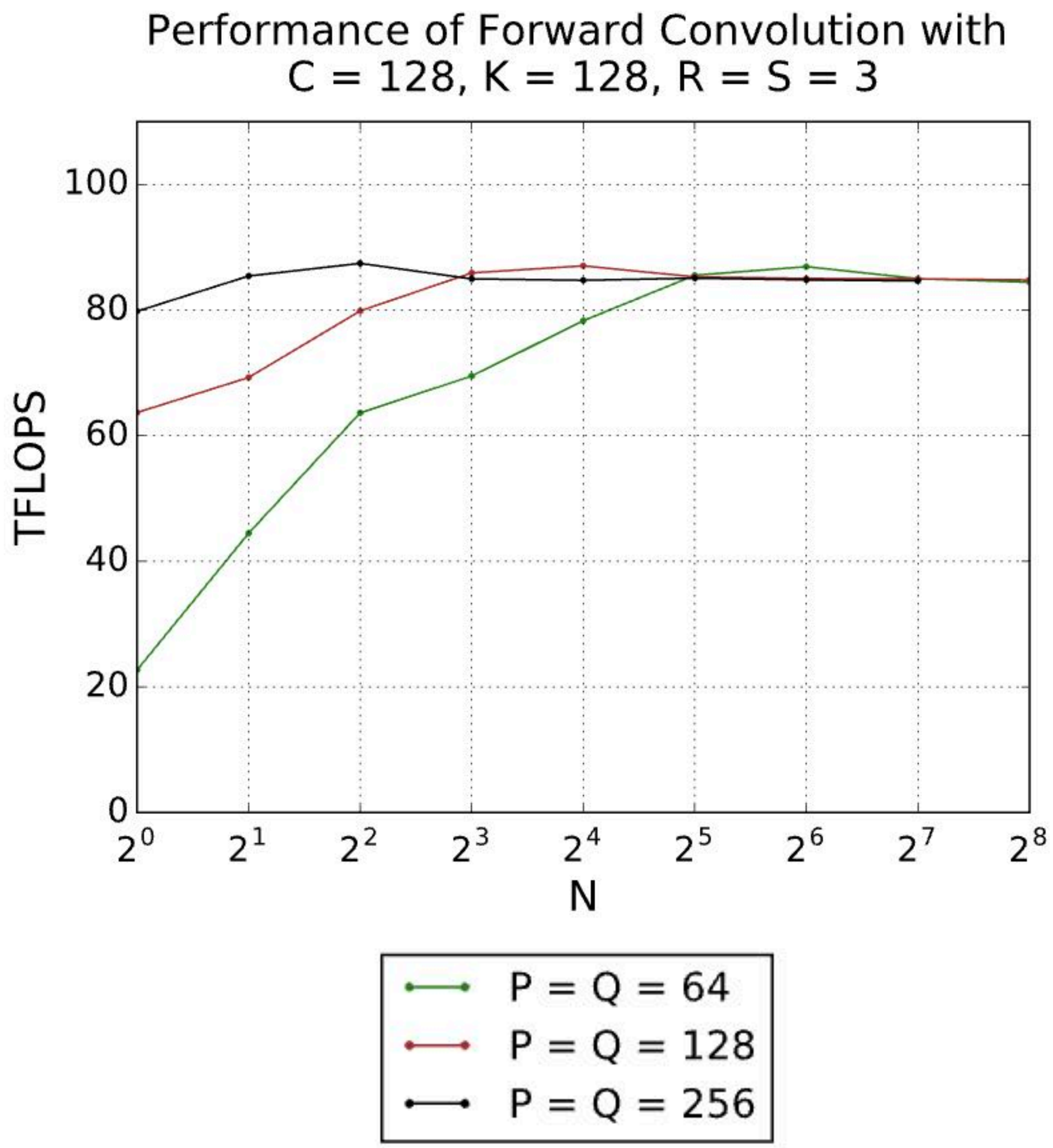
Need "a lot of parallel work" to fill the machine.

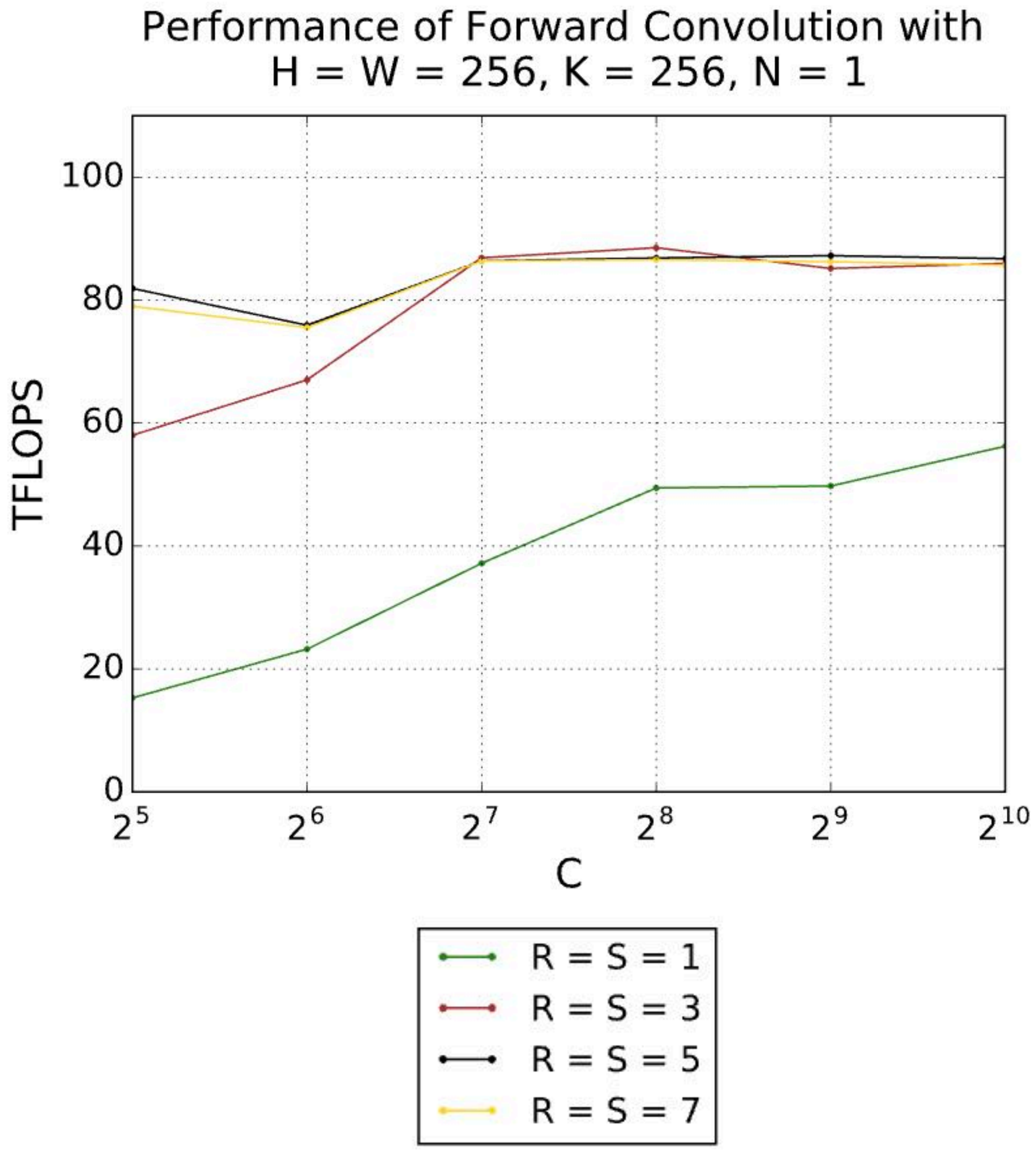
Higher performance with "more work"

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)







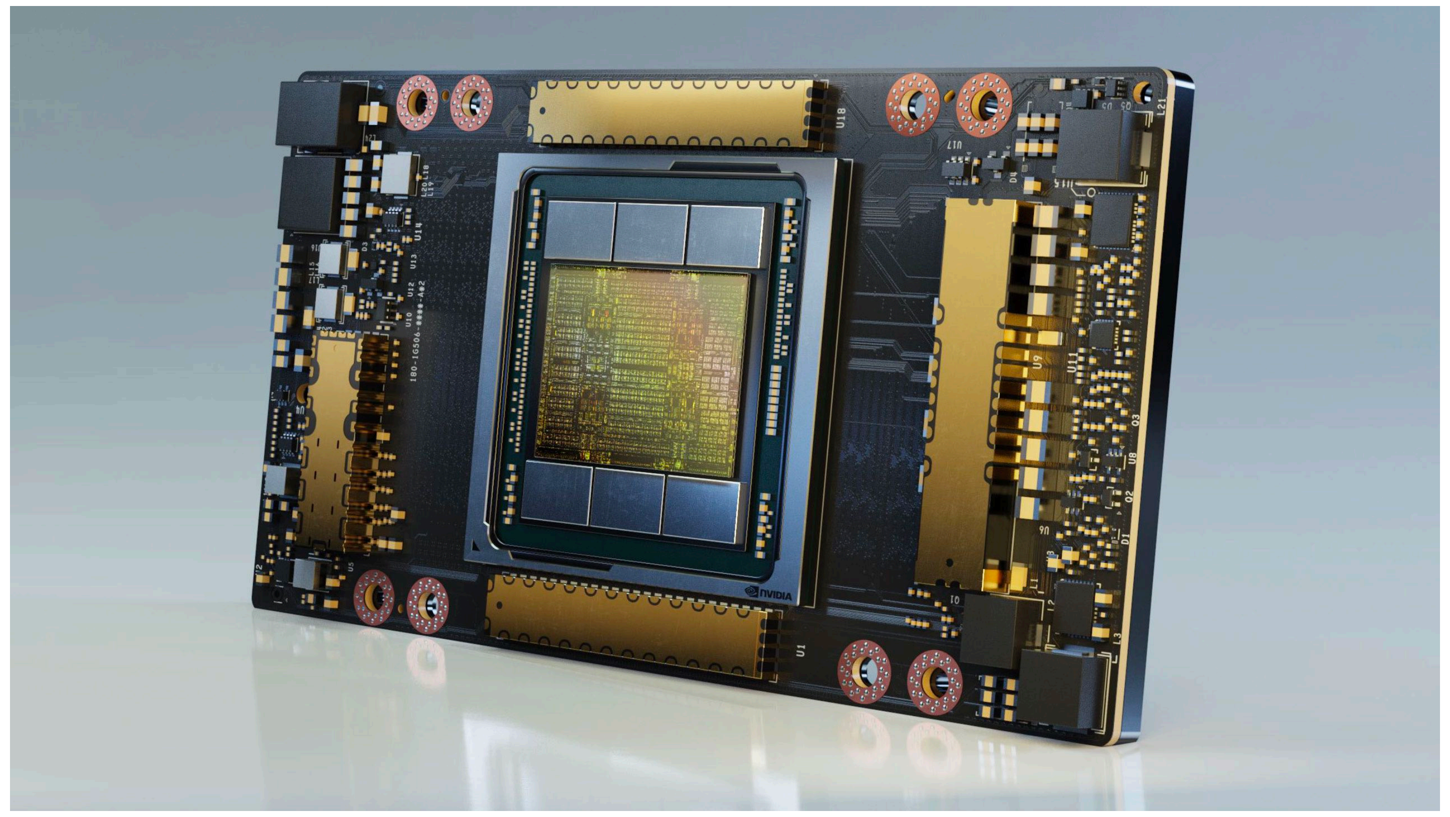
Direct implementation

```
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 all images in batch
   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++) {        // for all output channels
            float tmp = layer_biases[LAYER_NUM_FILTERS];
            for (int kk=0; kk<INPUT_DEPTH; kk++)          // combine filter responses from all 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;
         }
```

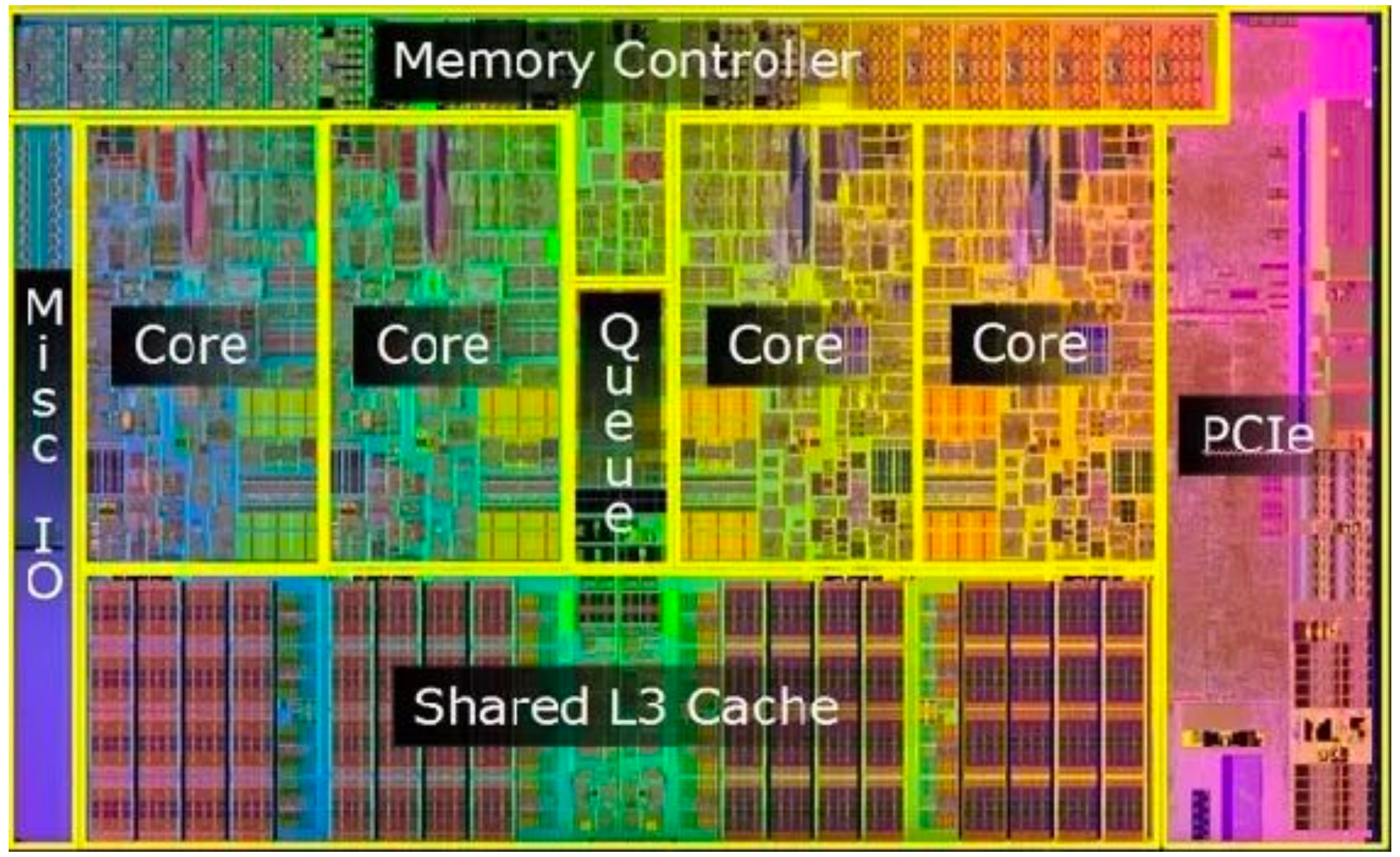
Or you can just directly implement this loop nest directly yourself.

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

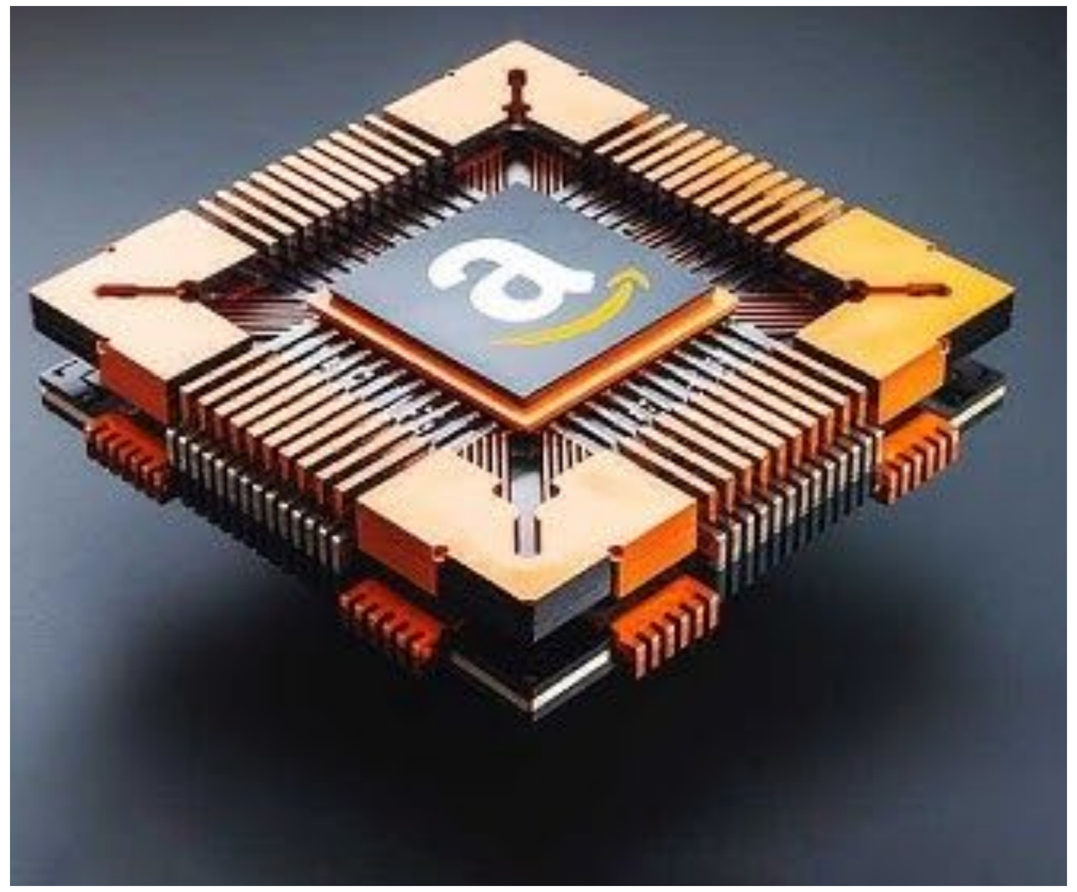
NVIDIA cuDNN



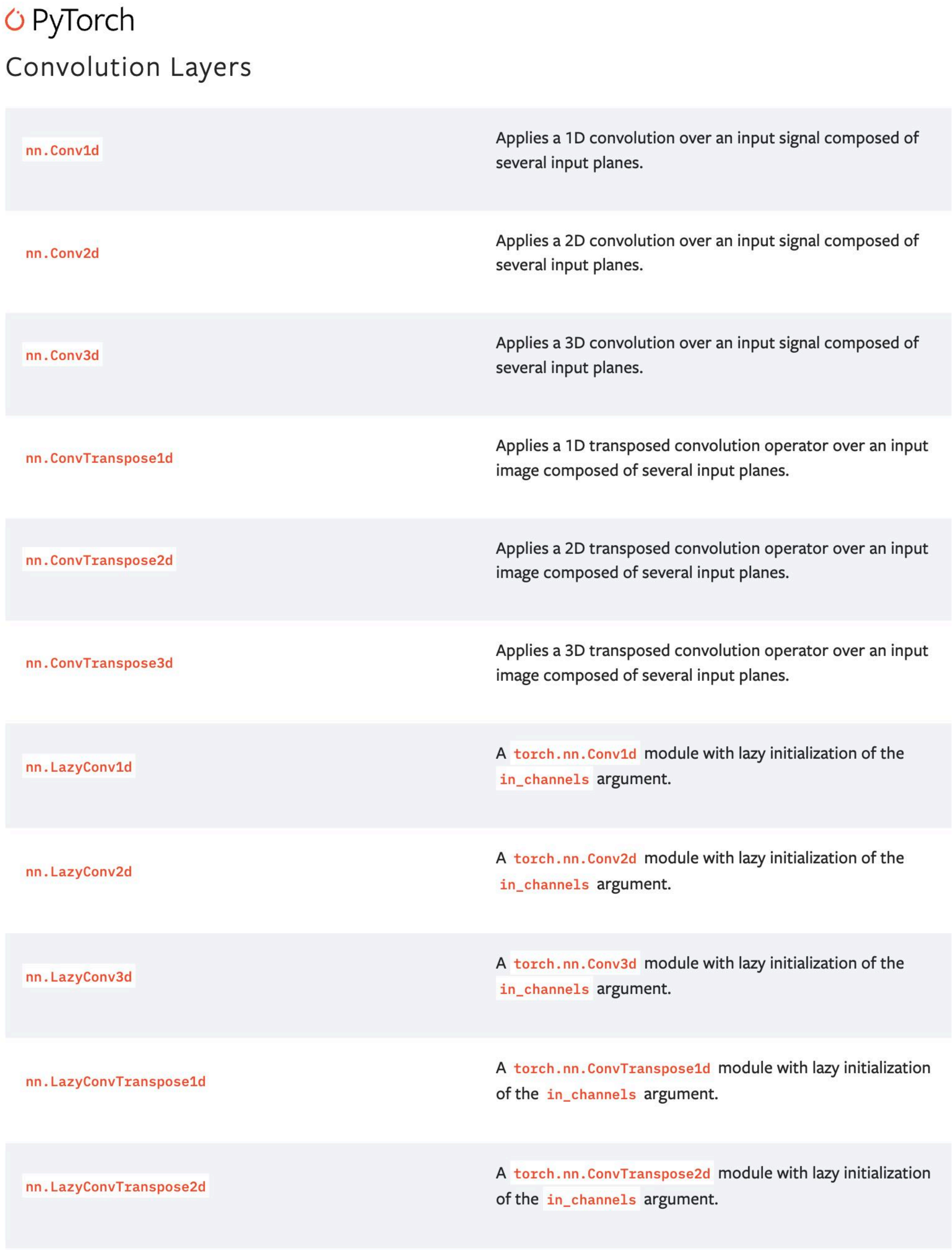
Intel® oneAPI Deep Neural Network Library



AWS NKI



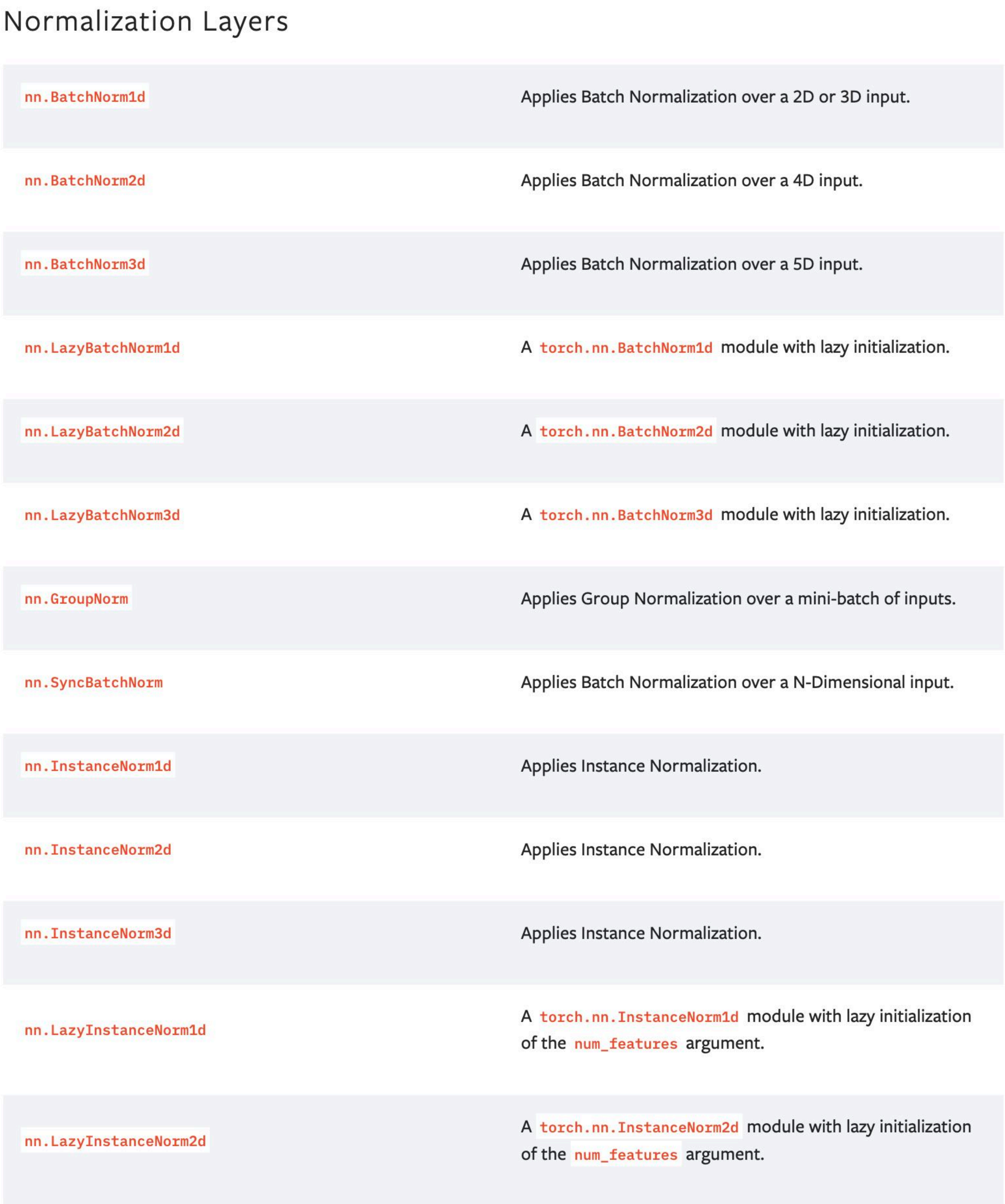
Libraries offering high-performance implementations of key DNN layers



PyTorch

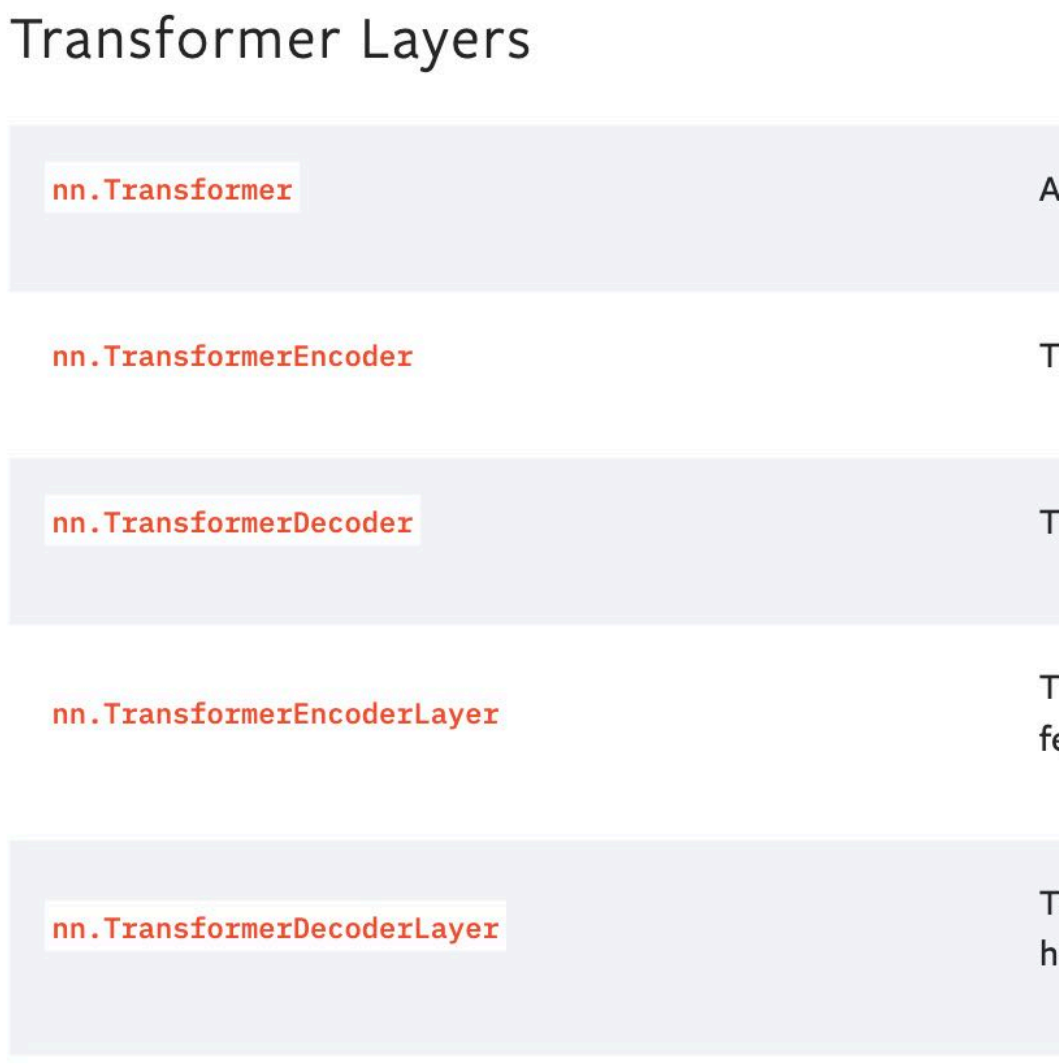
Convolution Layers

nn.Conv1d	Applies a 1D convolution over an input signal composed of several input planes.
nn.Conv2d	Applies a 2D convolution over an input signal composed of several input planes.
nn.Conv3d	Applies a 3D convolution over an input signal composed of several input planes.
nn.ConvTranspose1d	Applies a 1D transposed convolution operator over an input image composed of several input planes.
nn.ConvTranspose2d	Applies a 2D transposed convolution operator over an input image composed of several input planes.
nn.ConvTranspose3d	Applies a 3D transposed convolution operator over an input image composed of several input planes.
nn.LazyConv1d	A torch.nn.Conv1d module with lazy initialization of the in_channels argument.
nn.LazyConv2d	A torch.nn.Conv2d module with lazy initialization of the in_channels argument.
nn.LazyConv3d	A torch.nn.Conv3d module with lazy initialization of the in_channels argument.
nn.LazyConvTranspose1d	A torch.nn.ConvTranspose1d module with lazy initialization of the in_channels argument.
nn.LazyConvTranspose2d	A torch.nn.ConvTranspose2d module with lazy initialization of the in_channels argument.



Normalization Layers

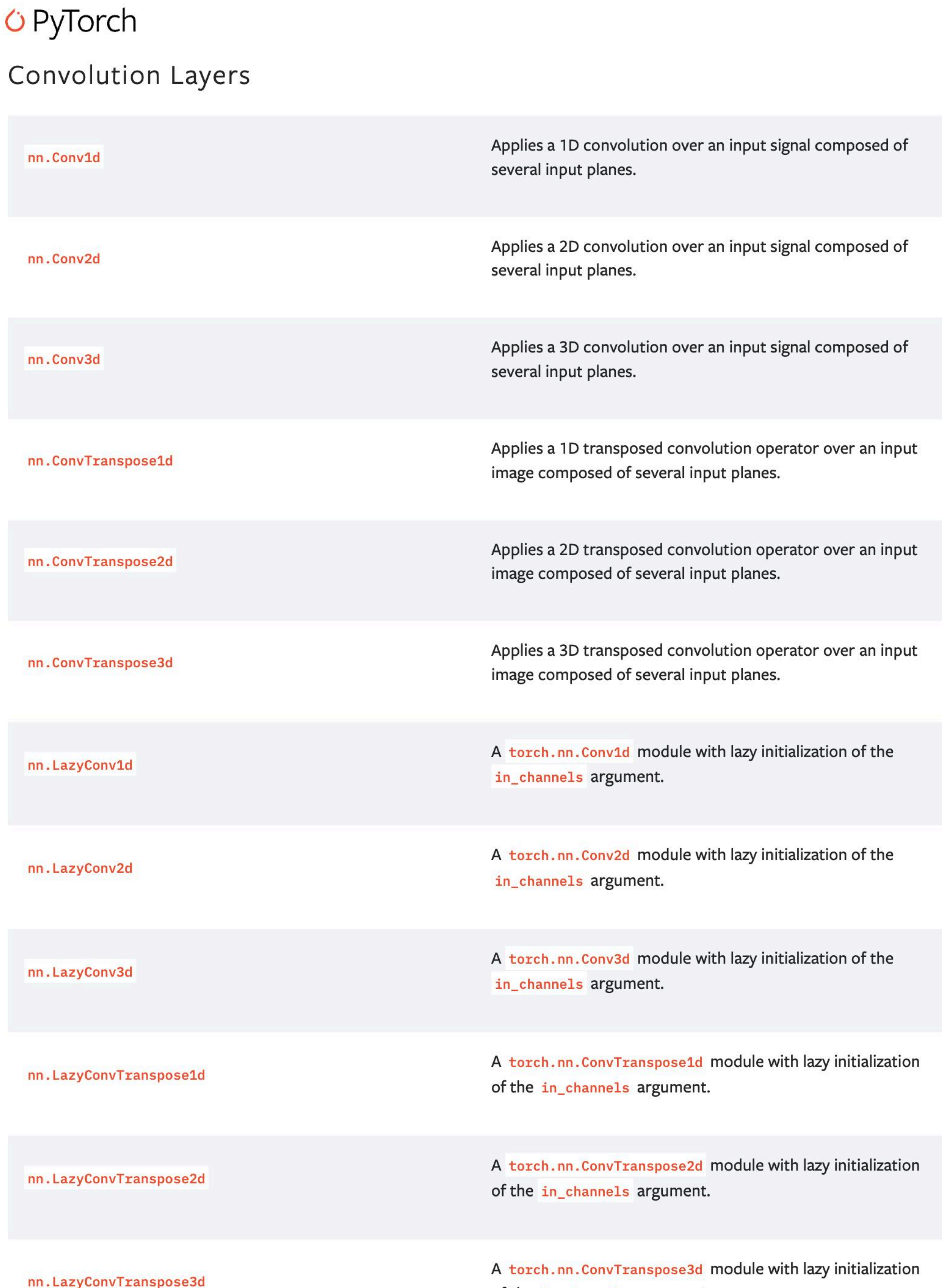
nn.BatchNorm1d	Applies Batch Normalization over a 2D or 3D input.
nn.BatchNorm2d	Applies Batch Normalization over a 4D input.
nn.BatchNorm3d	Applies Batch Normalization over a 5D input.
nn.LazyBatchNorm1d	A torch.nn.BatchNorm1d module with lazy initialization.
nn.LazyBatchNorm2d	A torch.nn.BatchNorm2d module with lazy initialization.
nn.LazyBatchNorm3d	A torch.nn.BatchNorm3d module with lazy initialization.
nn.GroupNorm	Applies Group Normalization over a mini-batch of inputs.
nn.SyncBatchNorm	Applies Batch Normalization over a N-Dimensional input.
nn.InstanceNorm1d	Applies Instance Normalization.
nn.InstanceNorm2d	Applies Instance Normalization.
nn.InstanceNorm3d	Applies Instance Normalization.
nn.LazyInstanceNorm1d	A torch.nn.InstanceNorm1d module with lazy initialization of the num_features argument.
nn.LazyInstanceNorm2d	A torch.nn.InstanceNorm2d module with lazy initialization of the num_features argument.



Transformer Layers

nn.Transformer	A
nn.TransformerEncoder	T
nn.TransformerDecoder	T
nn.TransformerEncoderLayer	T fe
nn.TransformerDecoderLayer	T h

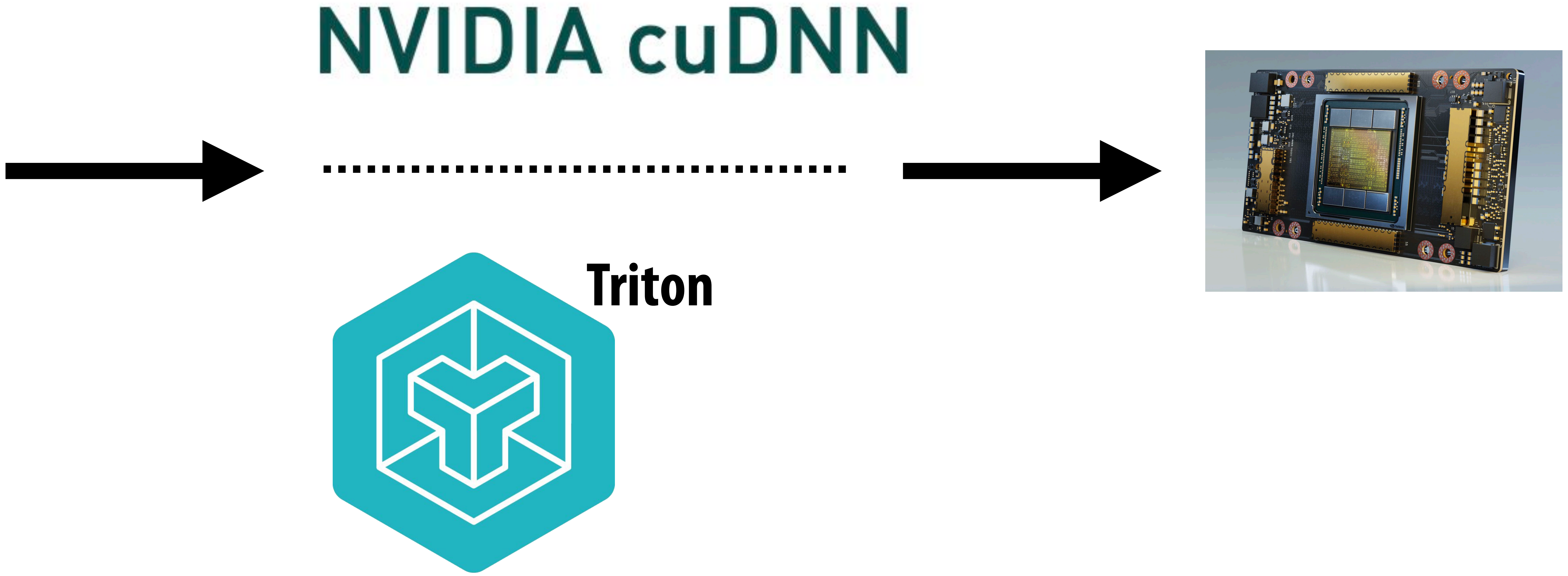
Libraries offering high-performance implementations of key DNN layers



PyTorch

Convolution Layers

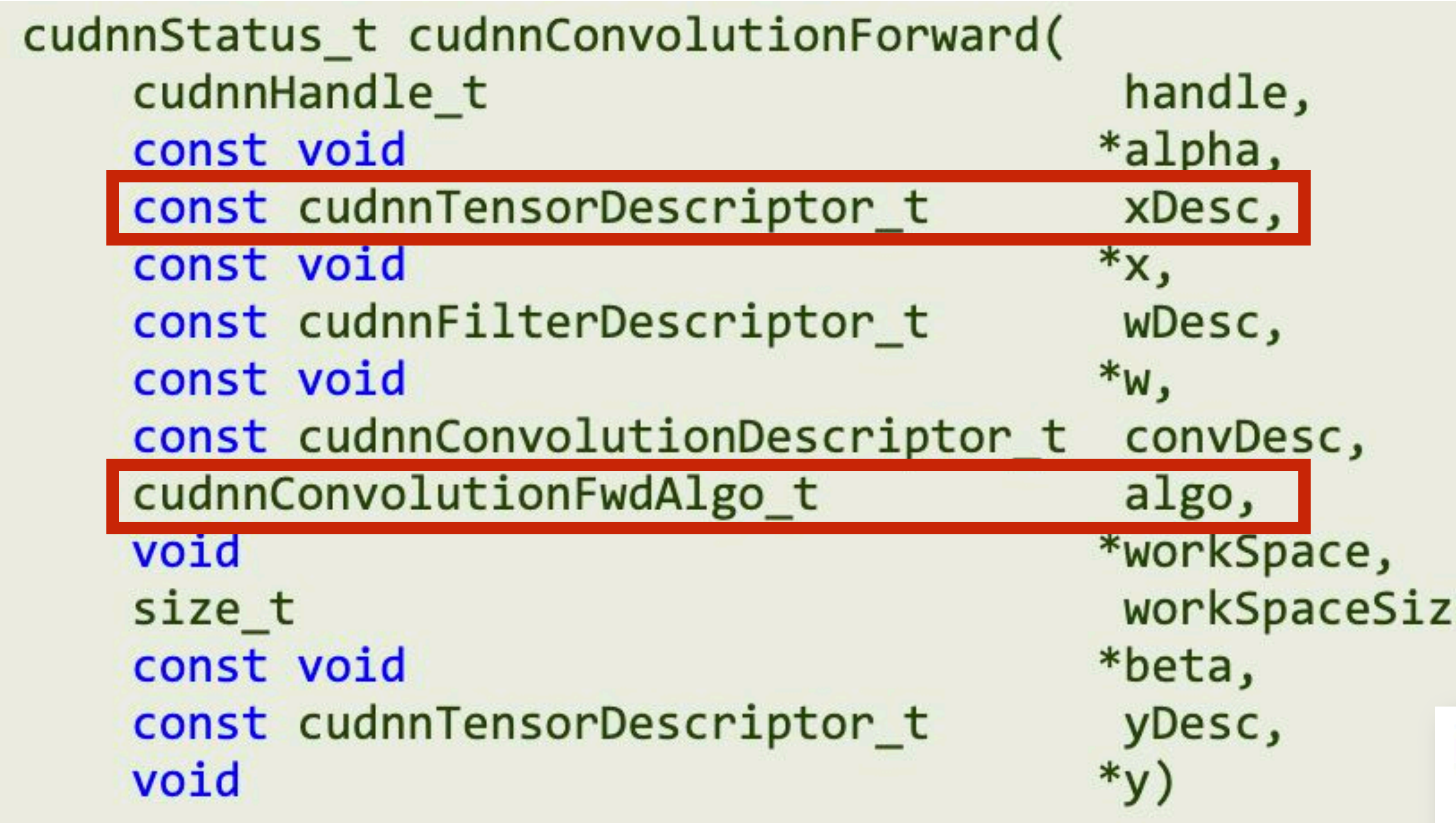
`nn.Conv1d`	Applies a 1D convolution over an input signal composed of several input planes.
`nn.Conv2d`	Applies a 2D convolution over an input signal composed of several input planes.
`nn.Conv3d`	Applies a 3D convolution over an input signal composed of several input planes.
`nn.ConvTranspose1d`	Applies a 1D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose2d`	Applies a 2D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose3d`	Applies a 3D transposed convolution operator over an input image composed of several input planes.
`nn.LazyConv1d`	A `torch.nn.Conv1d` module with lazy initialization of the `in_channels` argument.
`nn.LazyConv2d`	A `torch.nn.Conv2d` module with lazy initialization of the `in_channels` argument.
`nn.LazyConv3d`	A `torch.nn.Conv3d` module with lazy initialization of the `in_channels` argument.
`nn.LazyConvTranspose1d`	A `torch.nn.ConvTranspose1d` module with lazy initialization of the `in_channels` argument.
`nn.LazyConvTranspose2d`	A `torch.nn.ConvTranspose2d` module with lazy initialization of the `in_channels` argument.
`nn.LazyConvTranspose3d`	A `torch.nn.ConvTranspose3d` module with lazy initialization



NVIDIA cuDNN

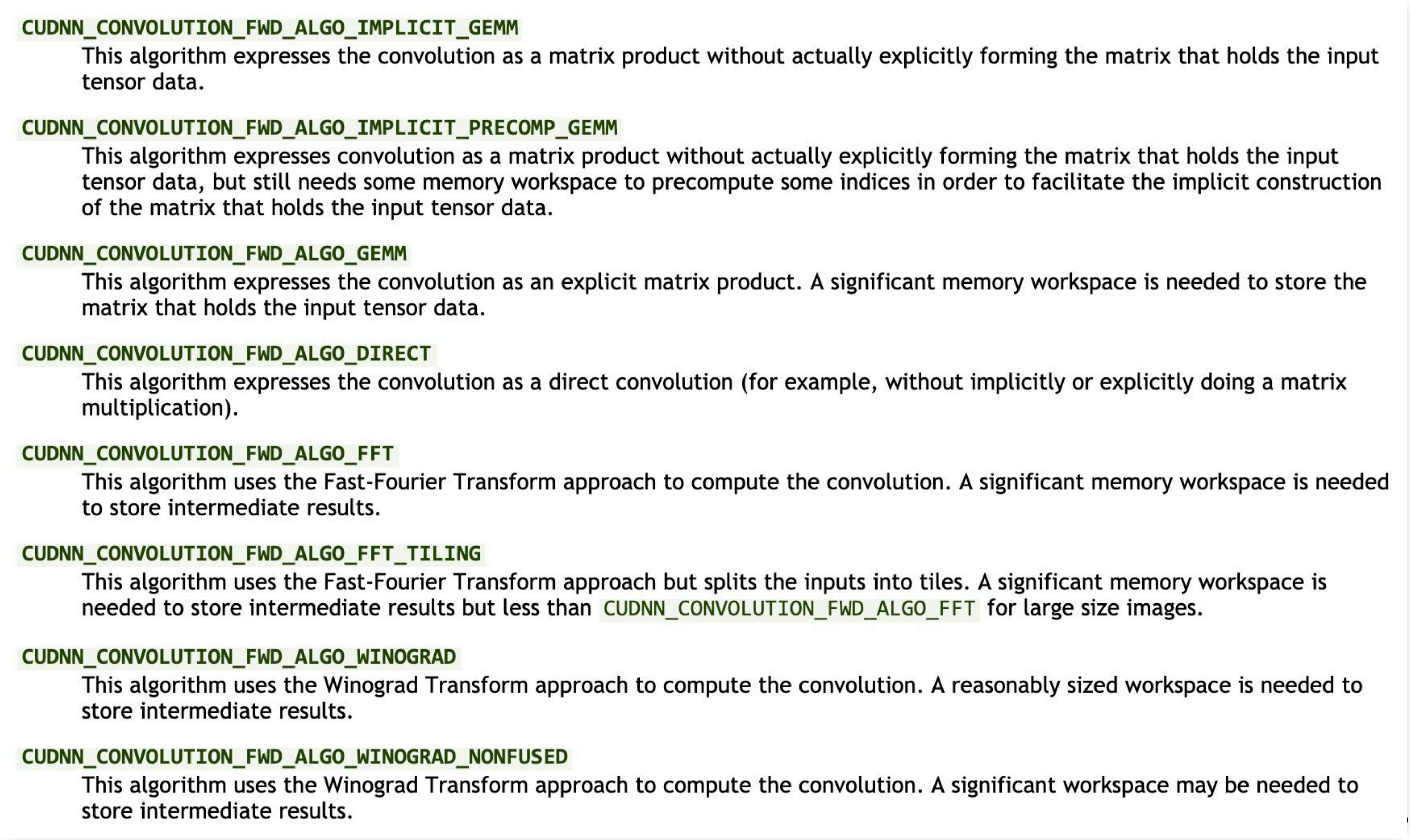
Triton

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.

Recall the loop fusion transformation: fuse multiple loops into one to increase a program’s arithmetic intensity

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

Two loads, one store per math op
(arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

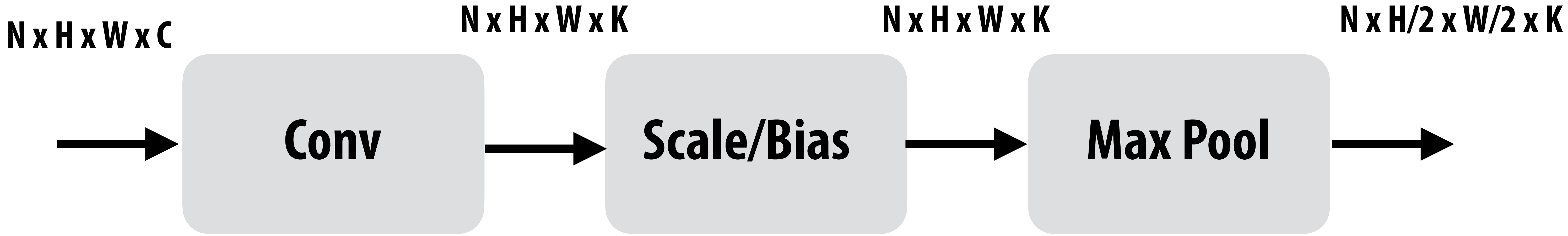
Program 2

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

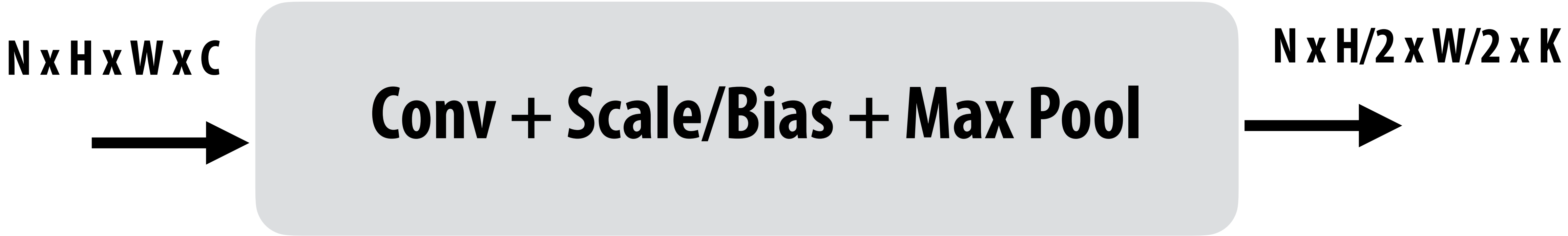
Memory traffic between operations

- Consider this sequence:



- Imagine the bandwidth cost of dumping 1 GB of conv outputs to memory, and then reading it back to just scale all the values, and then rereading to perform the pool! 😢

- Better solution:
  - 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 scale/bias 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 all images in batch
   for (int j=0; j<INPUT_HEIGHT; j++)
      for (int i=0; i<INPUT_WIDTH; i++)                   // for all output pixels
         for (int f=0; f<LAYER_NUM_FILTERS; f++) {        // for all output channels
            float tmp = 0.0f;
            for (int kk=0; kk<INPUT_DEPTH; kk++)          // filter combines responses from all 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:

How would you "fuse" a max pool operation following this layer (max of 2x2 blocks of output matrix)?

Hint: how would you "block" the yellow loops?

Another example: softmax on rows of a matrix

$\mathrm{softmax}(\mathbf{S})$ **is computing softmax over the rows of S**

For a row x:

$$\mathrm{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})}$$

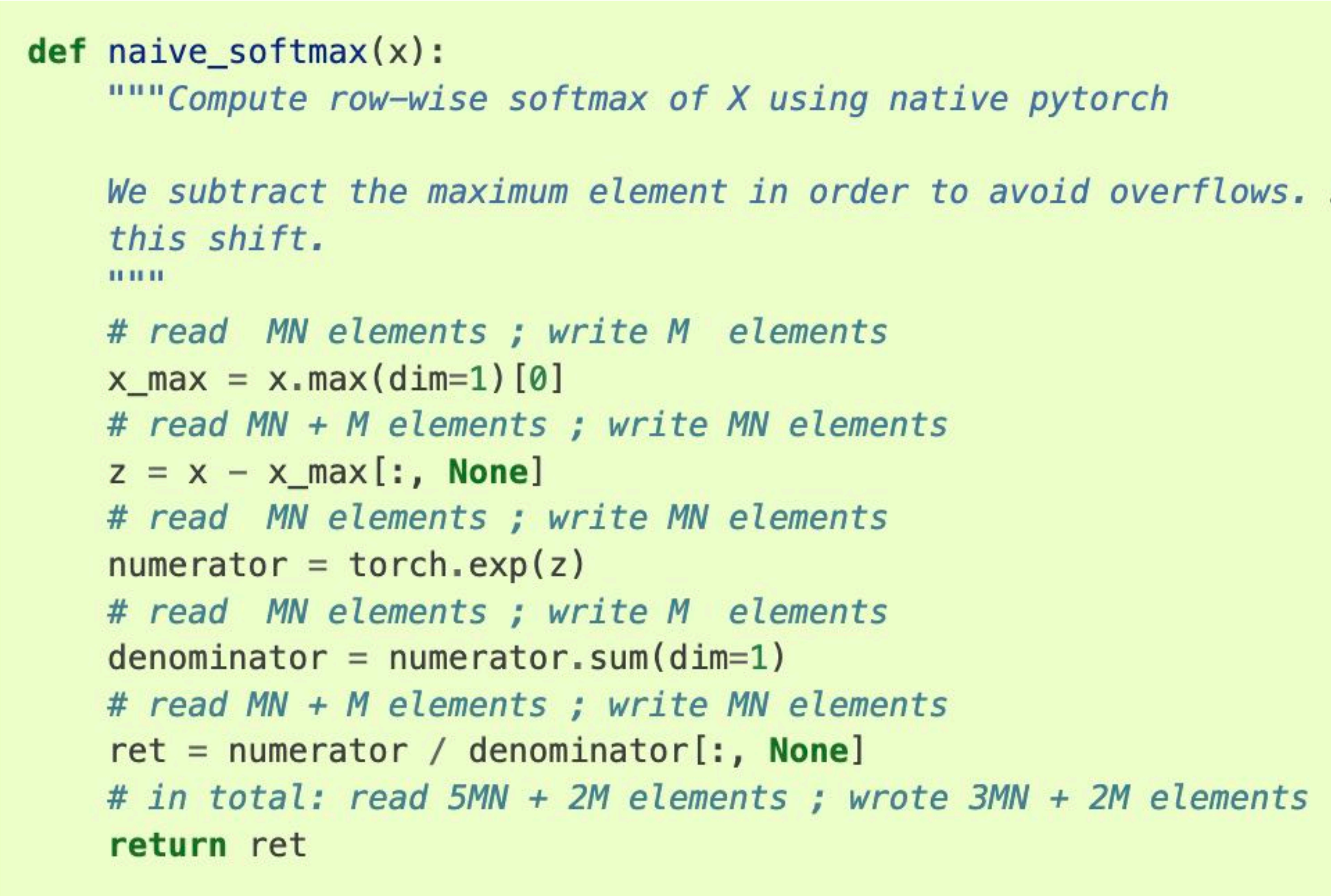
Where:

$$f(\mathbf{x}) = \begin{bmatrix} e^{\mathbf{x}_1 - m(\mathbf{x})} & e^{\mathbf{x}_1 - m(\mathbf{x})} & \ldots & e^{\mathbf{x}_B - m(\mathbf{x})} \end{bmatrix}$$

$$m(\mathbf{x}) = \max_i(\mathbf{x}_i)$$

$$l(\mathbf{x}) = \sum_i f(\mathbf{x})_i = \sum_i e^{\mathbf{x}_i - m(\mathbf{x})}$$

Naive code:

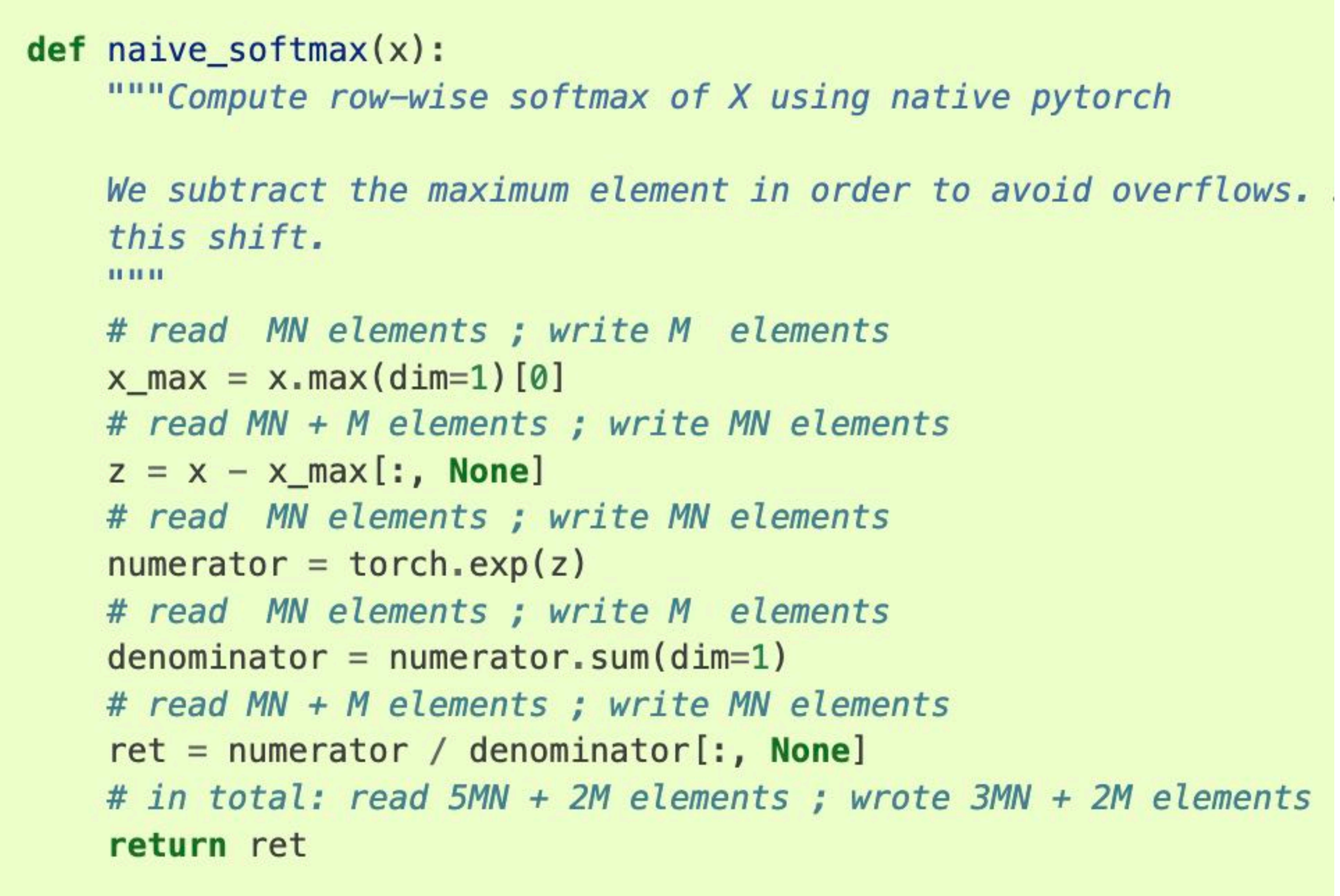


```python
def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch

    We subtract the maximum element in order to avoid overflows.
    this shift.
    """
    # read  MN elements ; write M  elements
    x_max = x.max(dim=1)[0]
    # read MN + M elements ; write MN elements
    z = x - x_max[:, None]
    # read  MN elements ; write MN elements
    numerator = torch.exp(z)
    # read  MN elements ; write M  elements
    denominator = numerator.sum(dim=1)
    # read MN + M elements ; write MN elements
    ret = numerator / denominator[:, None]
    # in total: read 5MN + 2M elements ; wrote 3MN + 2M elements
    return ret
```

The problem is that an entire M x N "matrix" is read/written from memory each step. So the problem has low arithmetic intensity.

Another example: softmax on rows of a matrix

Naive code:

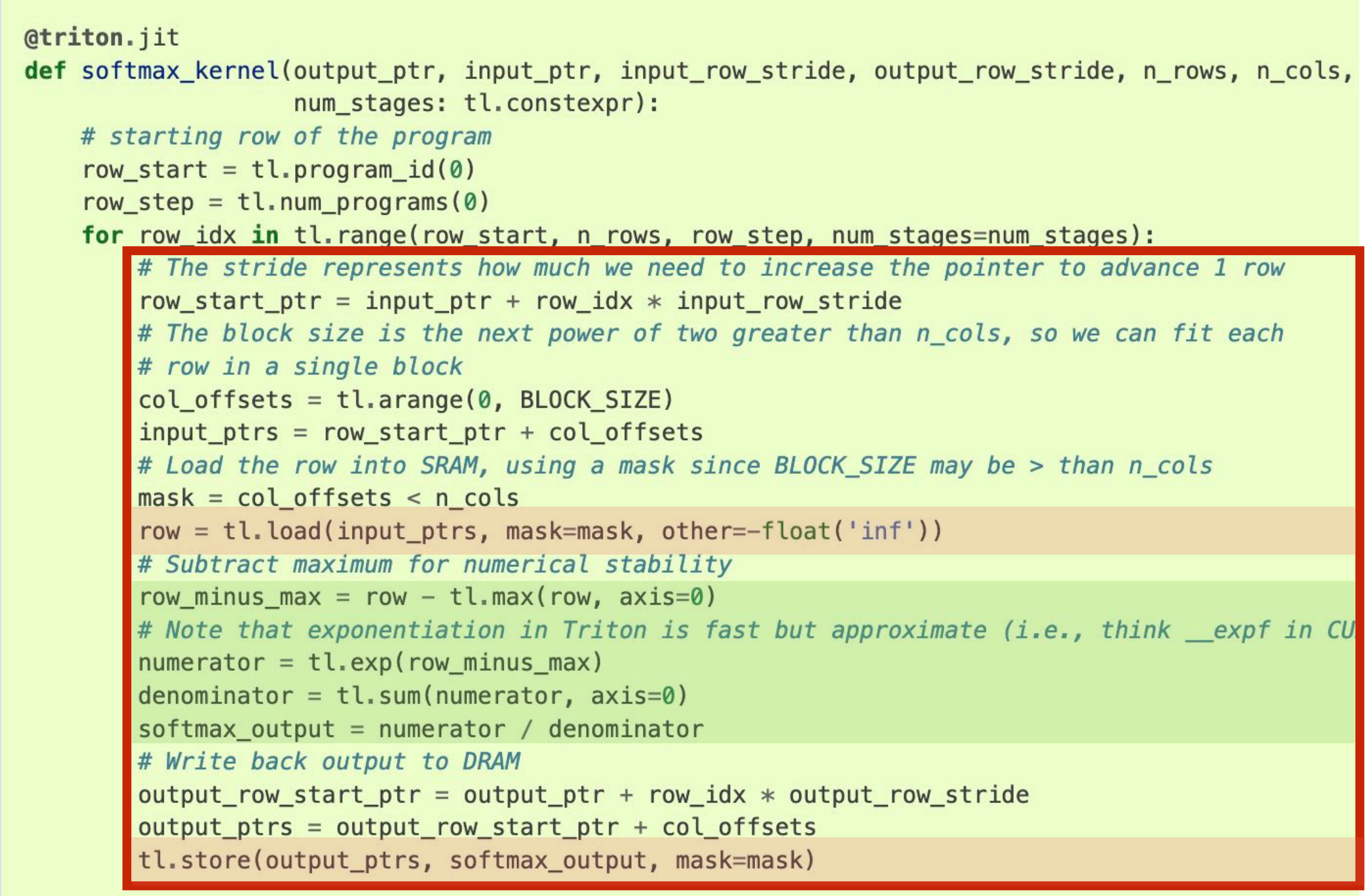


```
def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch

    We subtract the maximum element in order to avoid overflows.
    this shift.
    """
    # read  MN elements ; write M  elements
    x_max = x.max(dim=1)[0]
    # read MN + M elements ; write MN elements
    z = x - x_max[:, None]
    # read  MN elements ; write MN elements
    numerator = torch.exp(z)
    # read  MN elements ; write M  elements
    denominator = numerator.sum(dim=1)
    # read MN + M elements ; write MN elements
    ret = numerator / denominator[:, None]
    # in total: read 5MN + 2M elements ; wrote 3MN + 2M elements
    return ret
```

The problem is that an entire M x N "matrix" is read/written from memory each step. So the problem has low arithmetic intensity.

Reads 5MN + 2M elements, writes 3MN + 2M elements

"Fused" implementation:



```
@triton.jit
def softmax_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride, n_rows, n_cols,
                   num_stages: tl.constexpr):
    # starting row of the program
    row_start = tl.program_id(0)
    row_step = tl.num_programs(0)
    for row_idx in tl.range(row_start, n_rows, row_step, num_stages=num_stages):
        # The stride represents how much we need to increase the pointer to advance 1 row
        row_start_ptr = input_ptr + row_idx * input_row_stride
        # The block size is the next power of two greater than n_cols, so we can fit each
        # row in a single block
        col_offsets = tl.arange(0, BLOCK_SIZE)
        input_ptrs = row_start_ptr + col_offsets
        # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols
        mask = col_offsets < n_cols
        row = tl.load(input_ptrs, mask=mask, other=-float('inf'))
        # Subtract maximum for numerical stability
        row_minus_max = row - tl.max(row, axis=0)
        # Note that exponentiation in Triton is fast but approximate (i.e., think __expf in CU
        numerator = tl.exp(row_minus_max)
        denominator = tl.sum(numerator, axis=0)
        softmax_output = numerator / denominator
        # Write back output to DRAM
        output_row_start_ptr = output_ptr + row_idx * output_row_stride
        output_ptrs = output_row_start_ptr + col_offsets
        tl.store(output_ptrs, softmax_output, mask=mask)
```

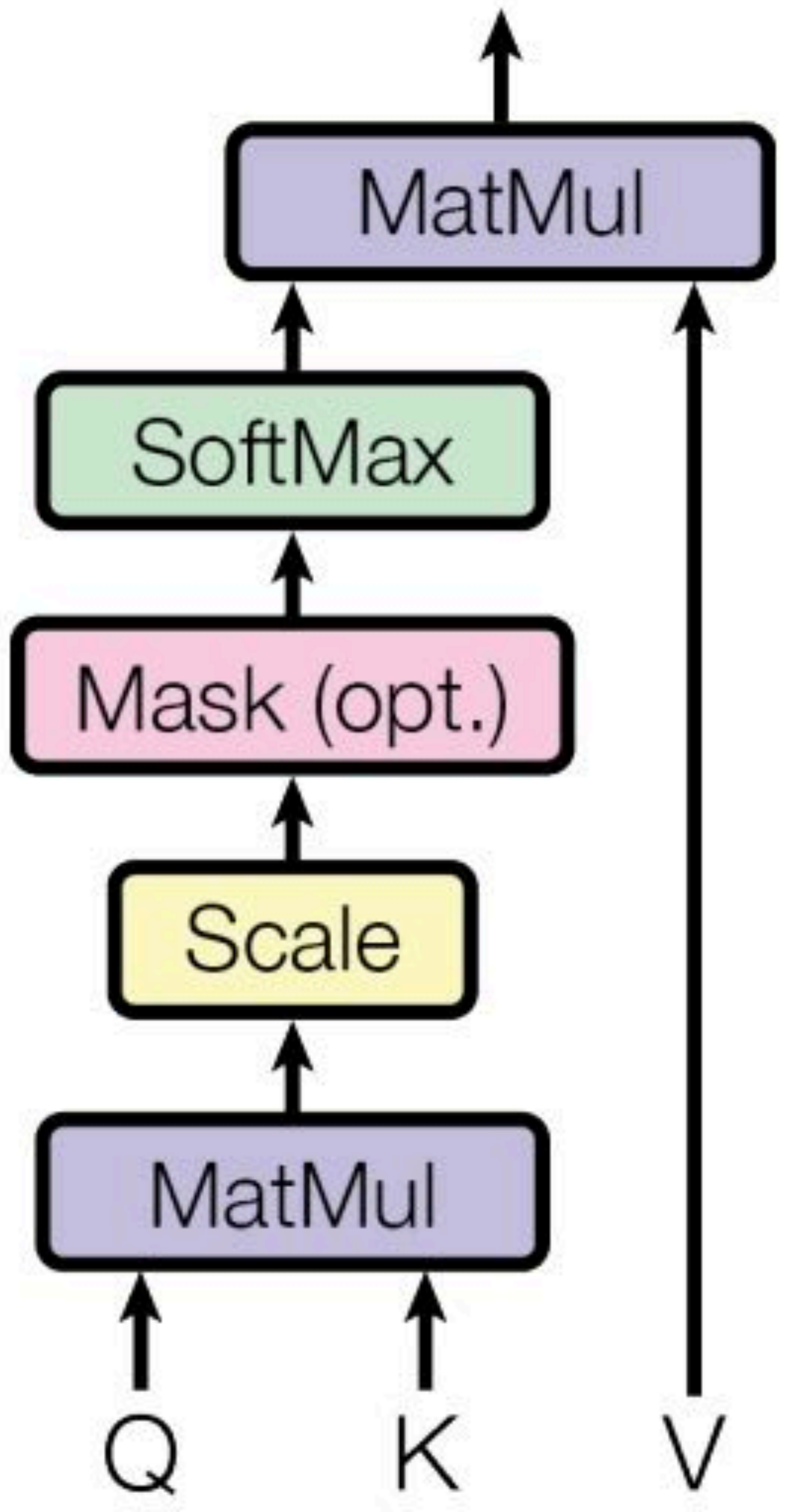
For each row:

Load row → compute entire softmax for a row → store row

Reads MN elements, writes MN elements,
assuming that working set for a single row fits in on-chip storage

A good idea:
fusion trick for computing “attention” in a modern transformer

Attention module in a modern transformer



Let N be the length of the input sequence

Let d be the size of a feature embedding

Where

Let Q be a N x d matrix

Let K be a N x d matrix

Let V be a N x d matrix

Let $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}$

Let $\mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}$ $\mathrm{softmax}(\mathbf{S})$ is computing softmax over the rows of S

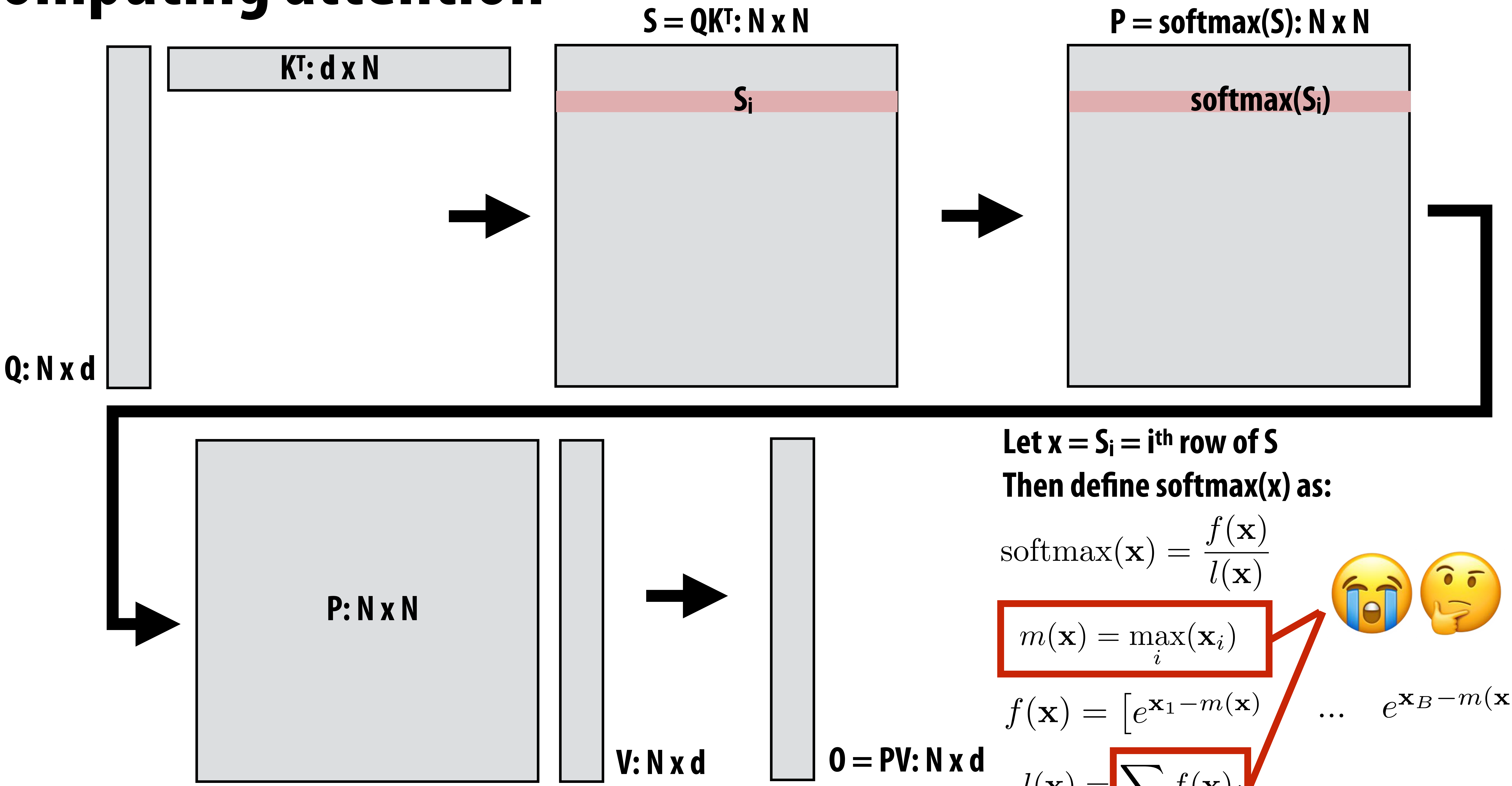
Let $\mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}$

Notes:

N can be long for long sequences (e.g., thousands)

Naive implementation uses N^2 space! Trouble!!!

Computing attention



Let x = S_i = ith row of S

Then define softmax(x) as:

$$\mathrm{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})}$$

$$\boxed{m(\mathbf{x}) = \max_i(\mathbf{x}_i)}$$

😭 🤔

$$f(\mathbf{x}) = \left[e^{\mathbf{x}_1 - m(\mathbf{x})} \quad \ldots \quad e^{\mathbf{x}_B - m(\mathbf{x})}\right]$$

$$l(\mathbf{x}) = \boxed{\sum_i f(\mathbf{x})_i}$$

Let's look into softmax more closely...

$$\text{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})}$$

Where:

$$f(\mathbf{x}) = \begin{bmatrix} e^{\mathbf{x}_1 - m(\mathbf{x})} & e^{\mathbf{x}_1 - m(\mathbf{x})} & \ldots & e^{\mathbf{x}_B - m(\mathbf{x})} \end{bmatrix}$$

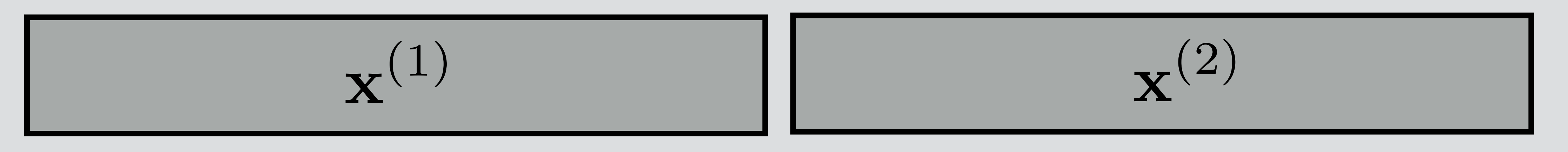
$$m(\mathbf{x}) = \max_i(\mathbf{x}_i)$$

$$l(\mathbf{x}) = \sum_i f(\mathbf{x})_i = \sum_i e^{\mathbf{x}_i - m(\mathbf{x})}$$

Let's break vector x into chunks:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} \end{bmatrix}$$

$\mathbf{x} =$



$\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)}$

Now:

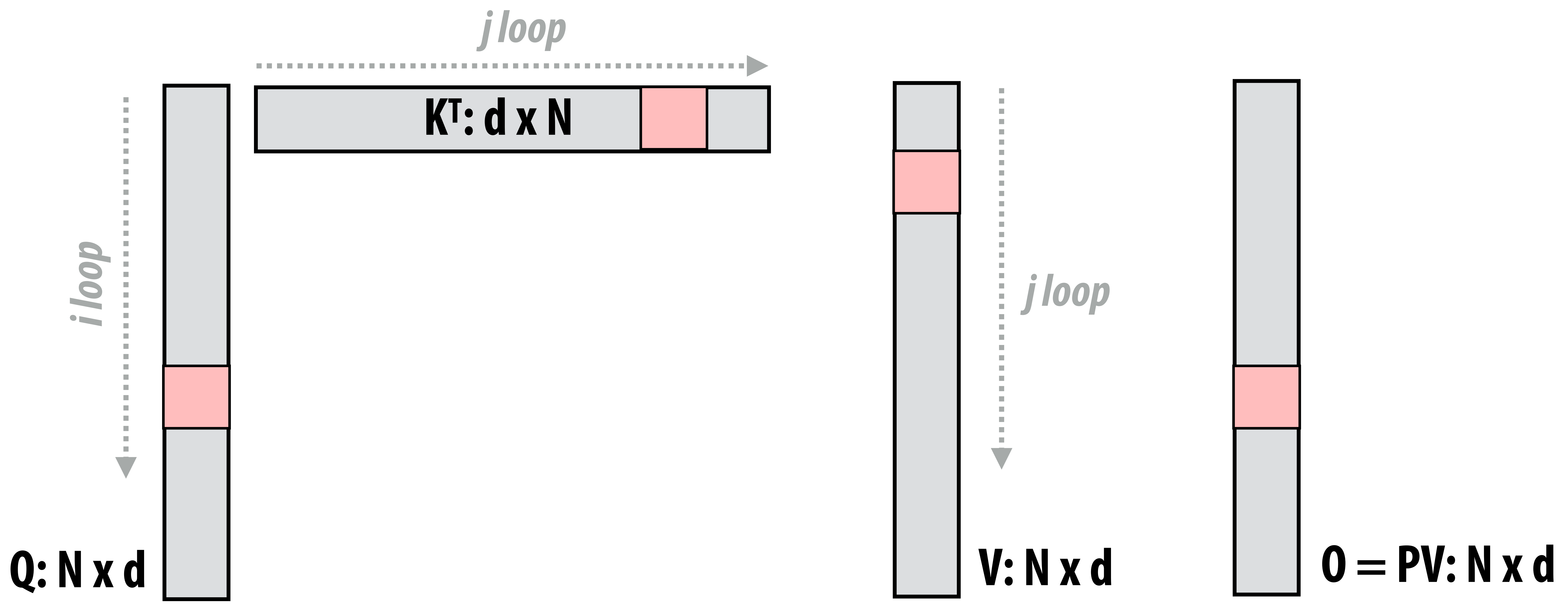
$$m(\mathbf{x}) = \max\left(m(\mathbf{x}^{(1)}), m(\mathbf{x}^{(2)})\right)$$

$$f(\mathbf{x}) = \begin{bmatrix} e^{m(\mathbf{x}^1) - m(\mathbf{x})} f(\mathbf{x}^{(1)}) & e^{m(\mathbf{x}^2) - m(\mathbf{x})} f(\mathbf{x}^{(2)}) \end{bmatrix}$$

$$l(\mathbf{x}) = e^{m(\mathbf{x}^{(1)}) - m(\mathbf{x})} l(\mathbf{x}^{(1)}) + e^{m(\mathbf{x}^{(2)}) - m(\mathbf{x})} l(\mathbf{x}^{(2)})$$

So softmax can be computed in chunks!

Fused attention



```
for each j:
   for each i:
         Load block $Q_i$, $K^T_j$, $V_j$, $O_i$
         Compute $S_{ij} = Q_i K^T_j$
         Compute $M_{ij} = m(S_{ij})$, $P_{ij} = f(S_{ij})$, and $l_{ij} = l(S_{ij})$      (all functions operate row-wise on row-vectors)
         Multiply $P_{ij}V_j$ and accumulate into $O_i$ with appropriate scalings (see previous slide for math)
```

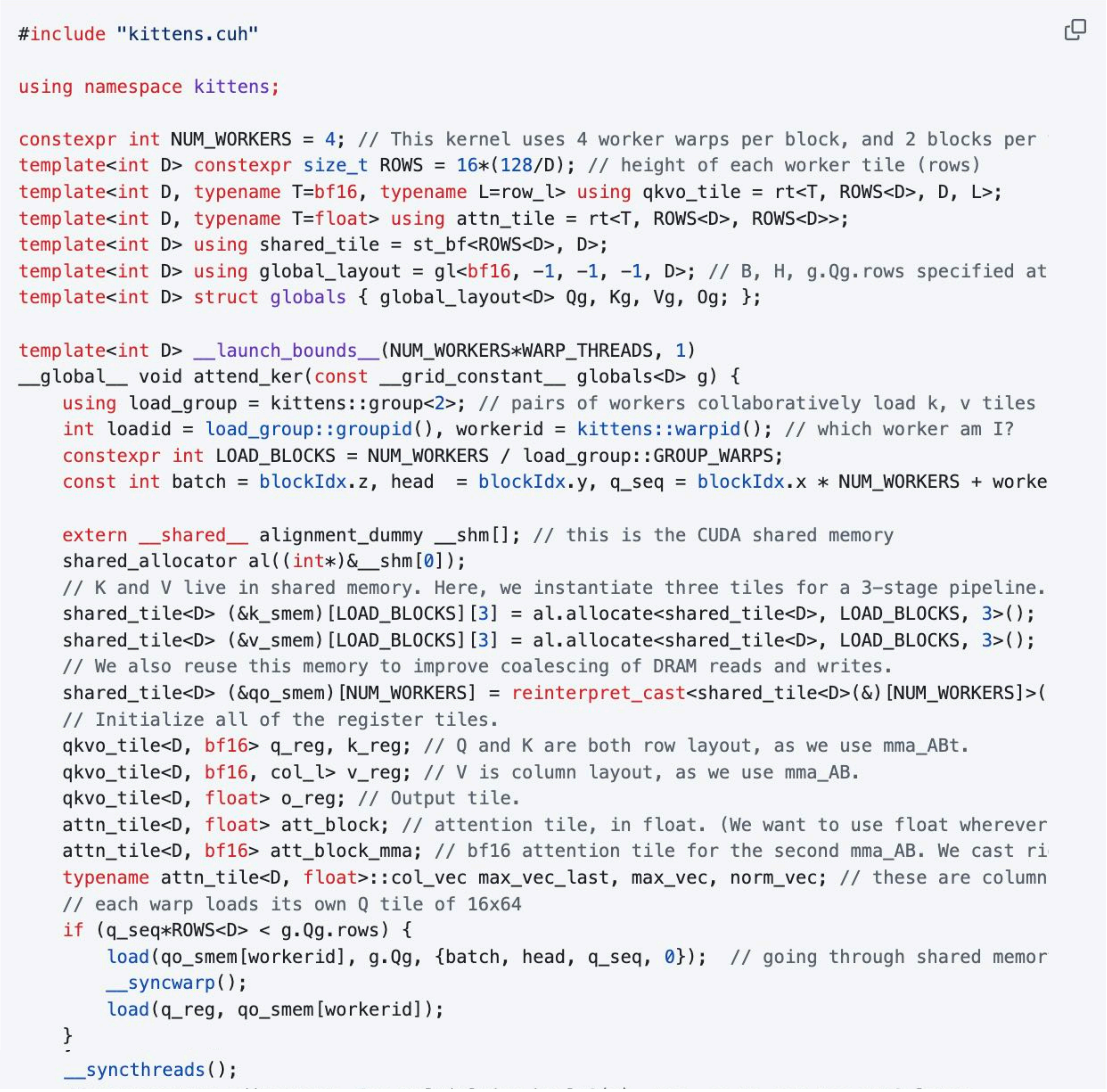
Save memory footprint:
Never materialize N² matrix

Save memory bandwidth:
(high arithmetic intensity)

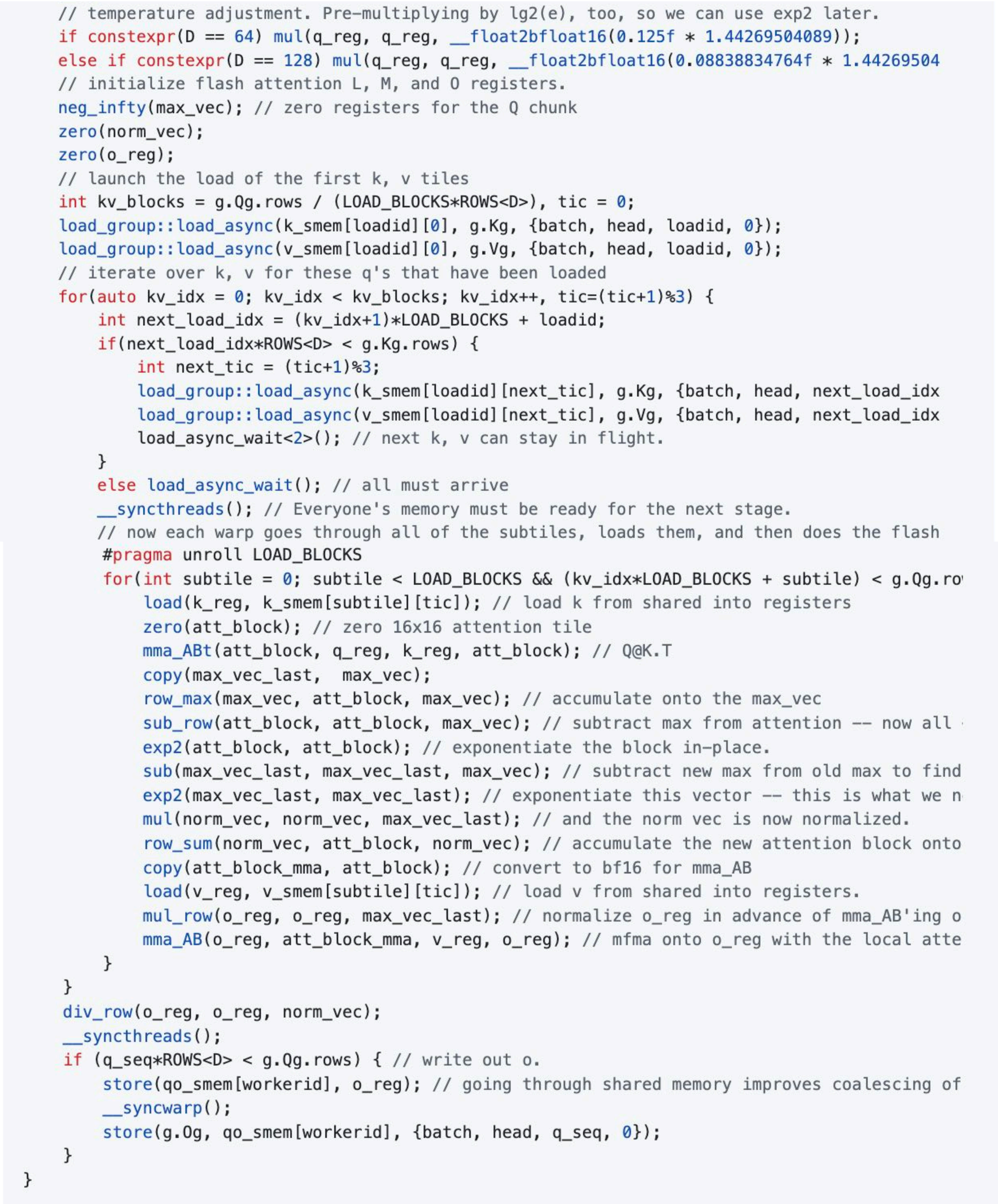
- Read 3 blocks (from Q, K, V)
- Do two matrix multiplies + a few row summations
- Accumulate into O block (which is resident in cache)

Note there is additional computation vs. the original version (must re-scale prior values of O each step of i-loop)

“Flash-Attention” in Thunderkittens



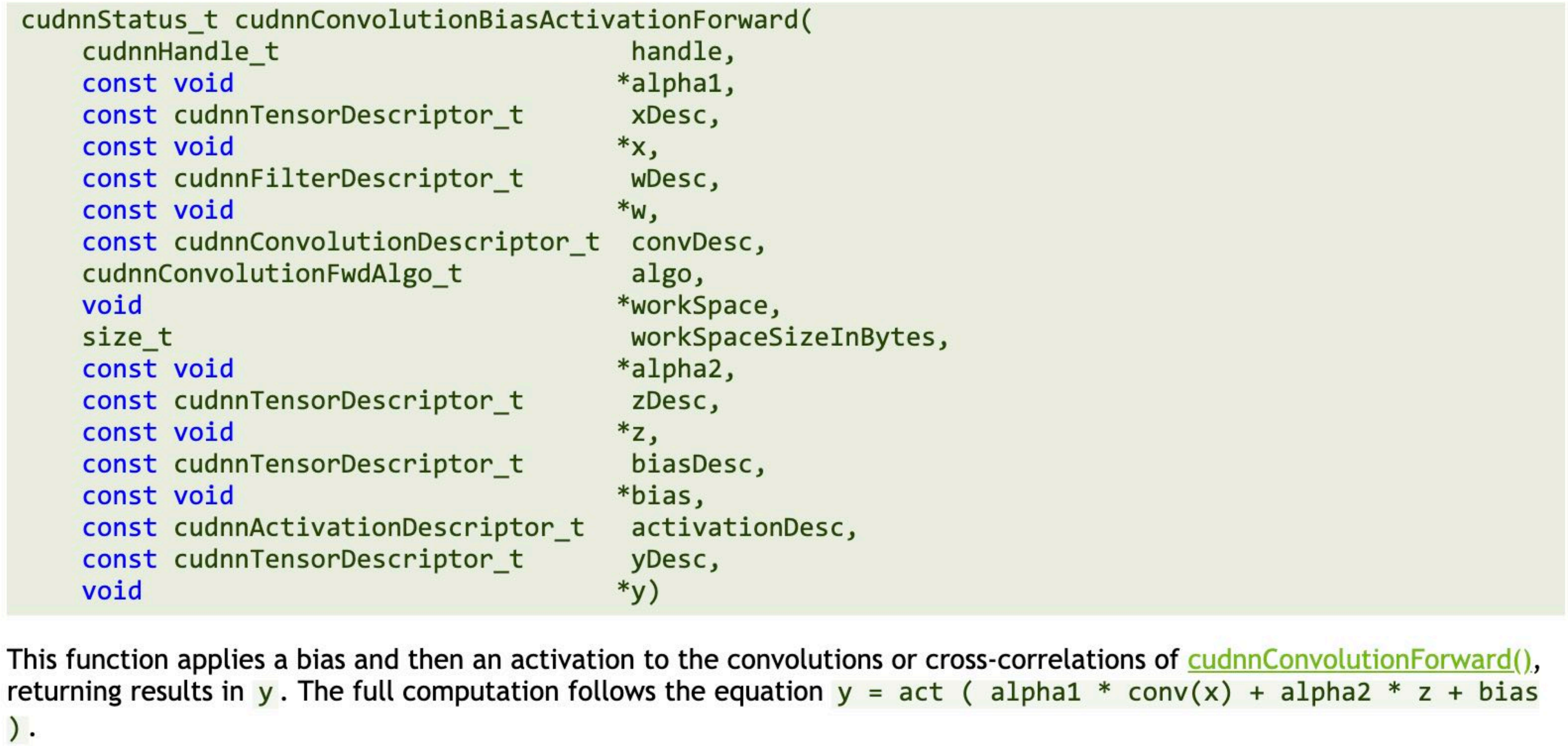
```
#include "kittens.cuh"

using namespace kittens;

constexpr int NUM_WORKERS = 4; // This kernel uses 4 worker warps per block, and 2 blocks per
template<int D> constexpr size_t ROWS = 16*(128/D); // height of each worker tile (rows)
template<int D, typename T=bf16, typename L=row_l> using qkvo_tile = rt<T, ROWS<D>, D, L>;
template<int D, typename T=float> using attn_tile = rt<T, ROWS<D>, ROWS<D>>;
template<int D> using shared_tile = st_bf<ROWS<D>, D>;
template<int D> using global_layout = gl<bf16, -1, -1, -1, D>; // B, H, g.Qg.rows specified at
template<int D> struct globals { global_layout<D> Qg, Kg, Vg, Og; };

template<int D> __launch_bounds__(NUM_WORKERS*WARP_THREADS, 1)
__global__ void attend_ker(const __grid_constant__ globals<D> g) {
    using load_group = kittens::group<2>; // pairs of workers collaboratively load k, v tiles
    int loadid = load_group::groupid(), workerid = kittens::warpid(); // which worker am I?
    constexpr int LOAD_BLOCKS = NUM_WORKERS / load_group::GROUP_WARPS;
    const int batch = blockIdx.z, head  = blockIdx.y, q_seq = blockIdx.x * NUM_WORKERS + worke

    extern __shared__ alignment_dummy __shm[]; // this is the CUDA shared memory
    shared_allocator al((int*)&__shm[0]);
    // K and V live in shared memory. Here, we instantiate three tiles for a 3-stage pipeline.
    shared_tile<D> (&k_smem)[LOAD_BLOCKS][3] = al.allocate<shared_tile<D>, LOAD_BLOCKS, 3>();
    shared_tile<D> (&v_smem)[LOAD_BLOCKS][3] = al.allocate<shared_tile<D>, LOAD_BLOCKS, 3>();
    // We also reuse this memory to improve coalescing of DRAM reads and writes.
    shared_tile<D> (&qo_smem)[NUM_WORKERS] = reinterpret_cast<shared_tile<D>(&)[NUM_WORKERS]>(
    // Initialize all of the register tiles.
    qkvo_tile<D, bf16> q_reg, k_reg; // Q and K are both row layout, as we use mma_ABt.
    qkvo_tile<D, bf16, col_l> v_reg; // V is column layout, as we use mma_AB.
    qkvo_tile<D, float> o_reg; // Output tile.
    attn_tile<D, float> att_block; // attention tile, in float. (We want to use float wherever
    attn_tile<D, bf16> att_block_mma; // bf16 attention tile for the second mma_AB. We cast ri
    typename attn_tile<D, float>::col_vec max_vec_last, max_vec, norm_vec; // these are column
    // each warp loads its own Q tile of 16x64
    if (q_seq*ROWS<D> < g.Qg.rows) {
        load(qo_smem[workerid], g.Qg, {batch, head, q_seq, 0});  // going through shared memor
        __syncwarp();
        load(q_reg, qo_smem[workerid]);
    }
    __syncthreads();
```



```
    // temperature adjustment. Pre-multiplying by lg2(e), too, so we can use exp2 later.
    if constexpr(D == 64) mul(q_reg, q_reg, __float2bfloat16(0.125f * 1.44269504089));
    else if constexpr(D == 128) mul(q_reg, q_reg, __float2bfloat16(0.08838834764f * 1.44269504
    // initialize flash attention L, M, and O registers.
    neg_infty(max_vec); // zero registers for the Q chunk
    zero(norm_vec);
    zero(o_reg);
    // launch the load of the first k, v tiles
    int kv_blocks = g.Qg.rows / (LOAD_BLOCKS*ROWS<D>), tic = 0;
    load_group::load_async(k_smem[loadid][0], g.Kg, {batch, head, loadid, 0});
    load_group::load_async(v_smem[loadid][0], g.Vg, {batch, head, loadid, 0});
    // iterate over k, v for these q's that have been loaded
    for(auto kv_idx = 0; kv_idx < kv_blocks; kv_idx++, tic=(tic+1)%3) {
        int next_load_idx = (kv_idx+1)*LOAD_BLOCKS + loadid;
        if(next_load_idx*ROWS<D> < g.Kg.rows) {
            int next_tic = (tic+1)%3;
            load_group::load_async(k_smem[loadid][next_tic], g.Kg, {batch, head, next_load_idx
            load_group::load_async(v_smem[loadid][next_tic], g.Vg, {batch, head, next_load_idx
            load_async_wait<2>(); // next k, v can stay in flight.
        }
        else load_async_wait(); // all must arrive
        __syncthreads(); // Everyone's memory must be ready for the next stage.
        // now each warp goes through all of the subtiles, loads them, and then does the flash
        #pragma unroll LOAD_BLOCKS
        for(int subtile = 0; subtile < LOAD_BLOCKS && (kv_idx*LOAD_BLOCKS + subtile) < g.Qg.ro
            load(k_reg, k_smem[subtile][tic]); // load k from shared into registers
            zero(att_block); // zero 16x16 attention tile
            mma_ABt(att_block, q_reg, k_reg, att_block); // Q@K.T
            copy(max_vec_last,  max_vec);
            row_max(max_vec, att_block, max_vec); // accumulate onto the max_vec
            sub_row(att_block, att_block, max_vec); // subtract max from attention -- now all
            exp2(att_block, att_block); // exponentiate the block in-place.
            sub(max_vec_last, max_vec_last, max_vec); // subtract new max from old max to find
            exp2(max_vec_last, max_vec_last); // exponentiate this vector -- this is what we n
            mul(norm_vec, norm_vec, max_vec_last); // and the norm vec is now normalized.
            row_sum(norm_vec, att_block, norm_vec); // accumulate the new attention block onto
            copy(att_block_mma, att_block); // convert to bf16 for mma_AB
            load(v_reg, v_smem[subtile][tic]); // load v from shared into registers.
            mul_row(o_reg, o_reg, max_vec_last); // normalize o_reg in advance of mma_AB'ing o
            mma_AB(o_reg, att_block_mma, v_reg, o_reg); // mfma onto o_reg with the local atte
        }
    }
    div_row(o_reg, o_reg, norm_vec);
    __syncthreads();
    if (q_seq*ROWS<D> < g.Qg.rows) { // write out o.
        store(qo_smem[workerid], o_reg); // going through shared memory improves coalescing of
        __syncwarp();
        store(g.Og, qo_smem[workerid], {batch, head, q_seq, 0});
    }
}
```

Fusion in modern DNN frameworks

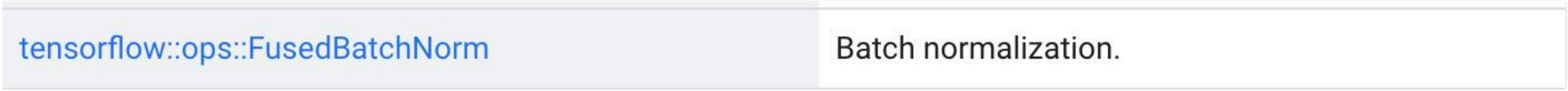
Old style: library writers hardcoded a few “fused” ops



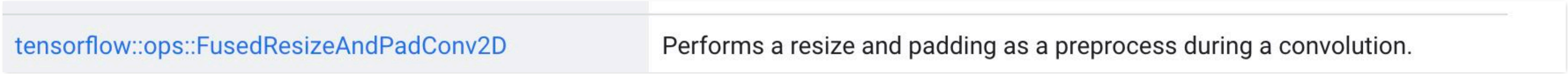
```
cudnnStatus_t cudnnConvolutionBiasActivationForward(
    cudnnHandle_t                          handle,
    const void                            *alpha1,
    const cudnnTensorDescriptor_t          xDesc,
    const void                            *x,
    const cudnnFilterDescriptor_t          wDesc,
    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:

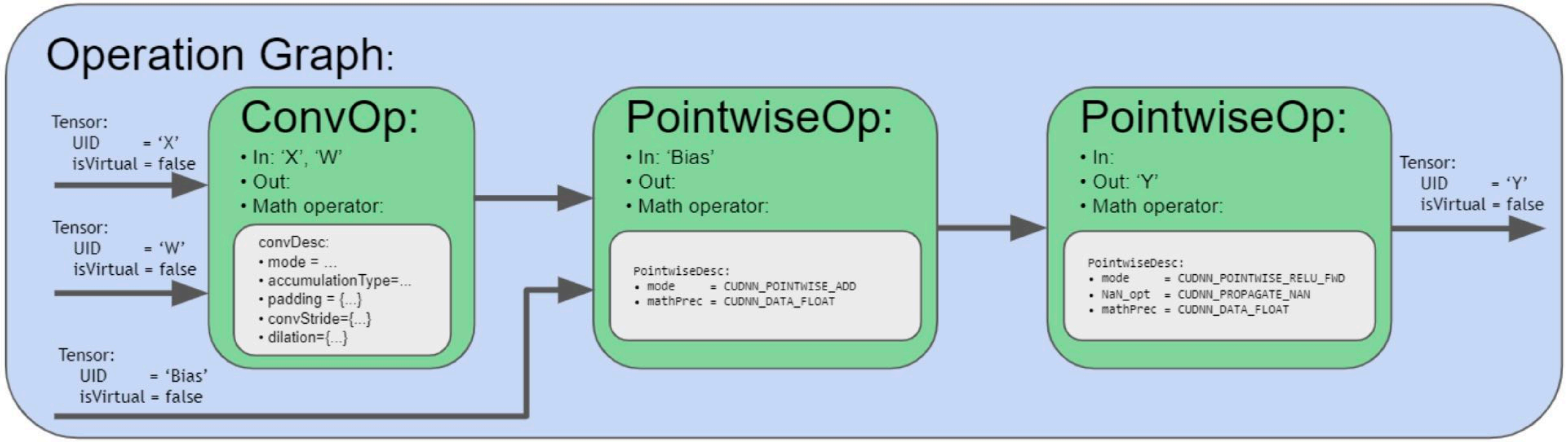


tensorflow::ops::FusedBatchNorm	Batch normalization.



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

More flexible fusion example: CUDNN “backend"



Operation Graph:

Tensor:
UID = ‘X’
isVirtual = false

Tensor:
UID = ‘W’
isVirtual = false

Tensor:
UID = ‘Bias’
isVirtual = false

ConvOp:
- In: ‘X’, ‘W’
- Out:
- Math operator:

convDesc:
- mode = ...
- accumulationType=...
- padding = {...}
- convStride={...}
- dilation={...}

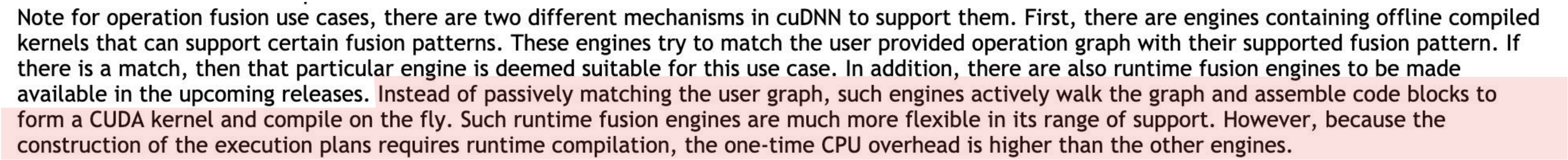
PointwiseOp:
- In: ‘Bias’
- Out:
- Math operator:

```
PointwiseDesc:
• mode     = CUDNN_POINTWISE_ADD
• mathPrec = CUDNN_DATA_FLOAT
```

PointwiseOp:
- In:
- Out: ‘Y’
- Math operator:

```
PointwiseDesc:
• mode     = CUDNN_POINTWISE_RELU_FWD
• NaN_opt  = CUDNN_PROPAGATE_NAN
• mathPrec = CUDNN_DATA_FLOAT
```

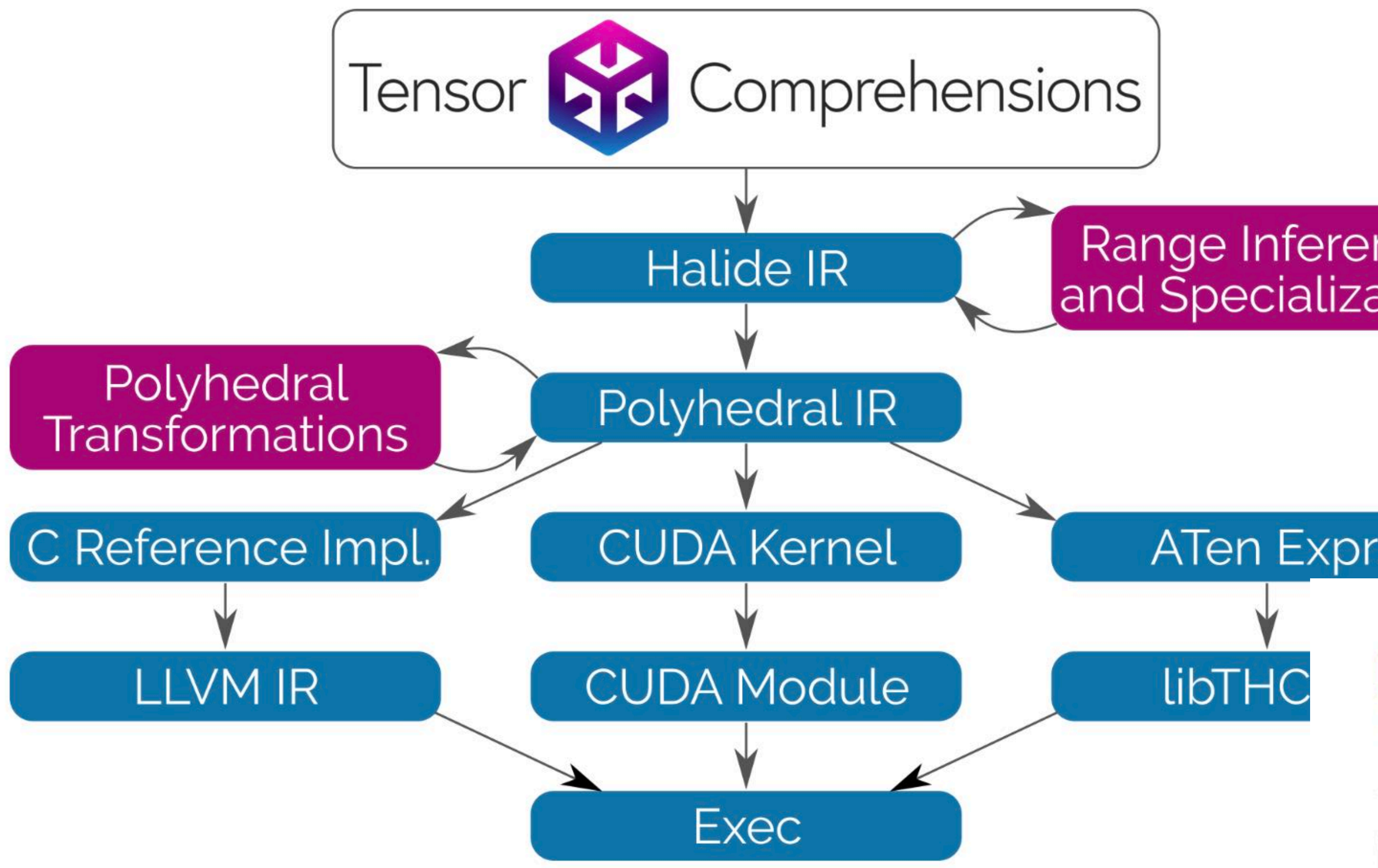
Tensor:
UID = ‘Y’
isVirtual = false



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

Many compiler-based efforts to automatically schedule key DNN operations



Tensor Comprehensions

Halide IR

Range Inference and Specialization

Polyhedral Transformations

Polyhedral IR

C Reference Impl.

CUDA Kernel

ATen Expr.

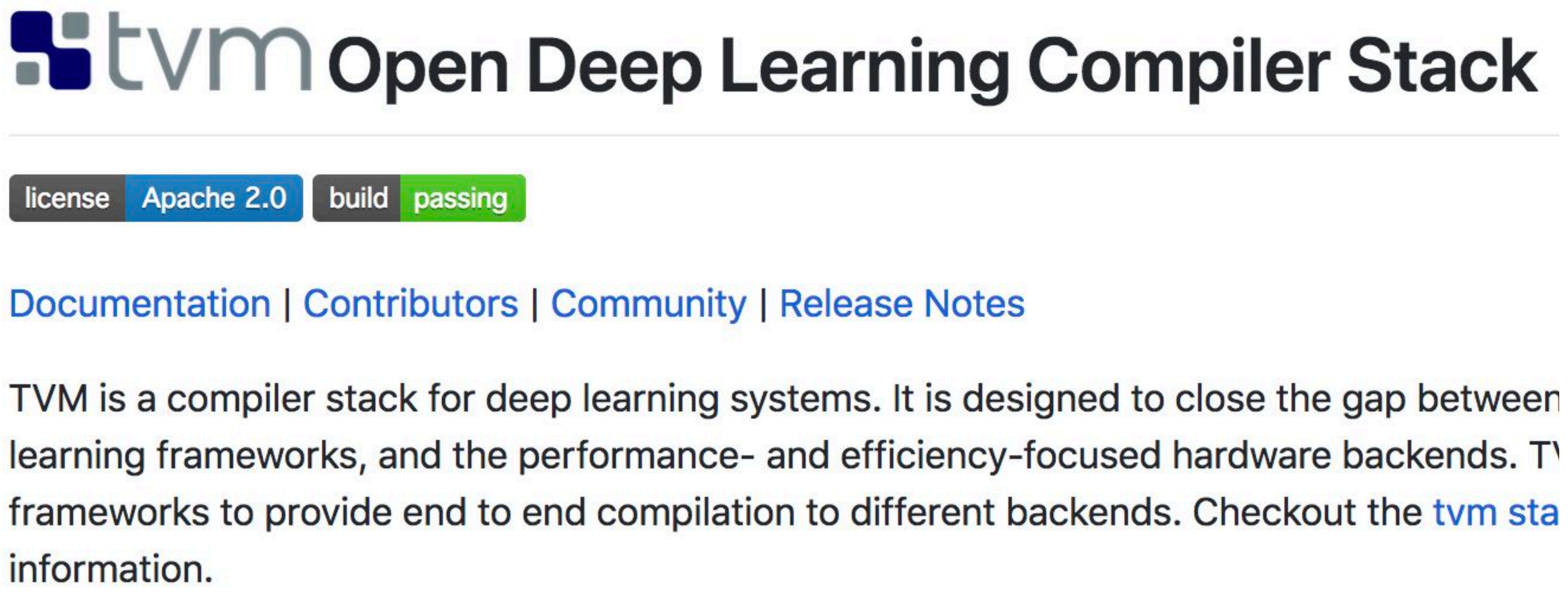
LLVM IR

CUDA Module

libTHC

Exec

JAX: Autograd and XLA

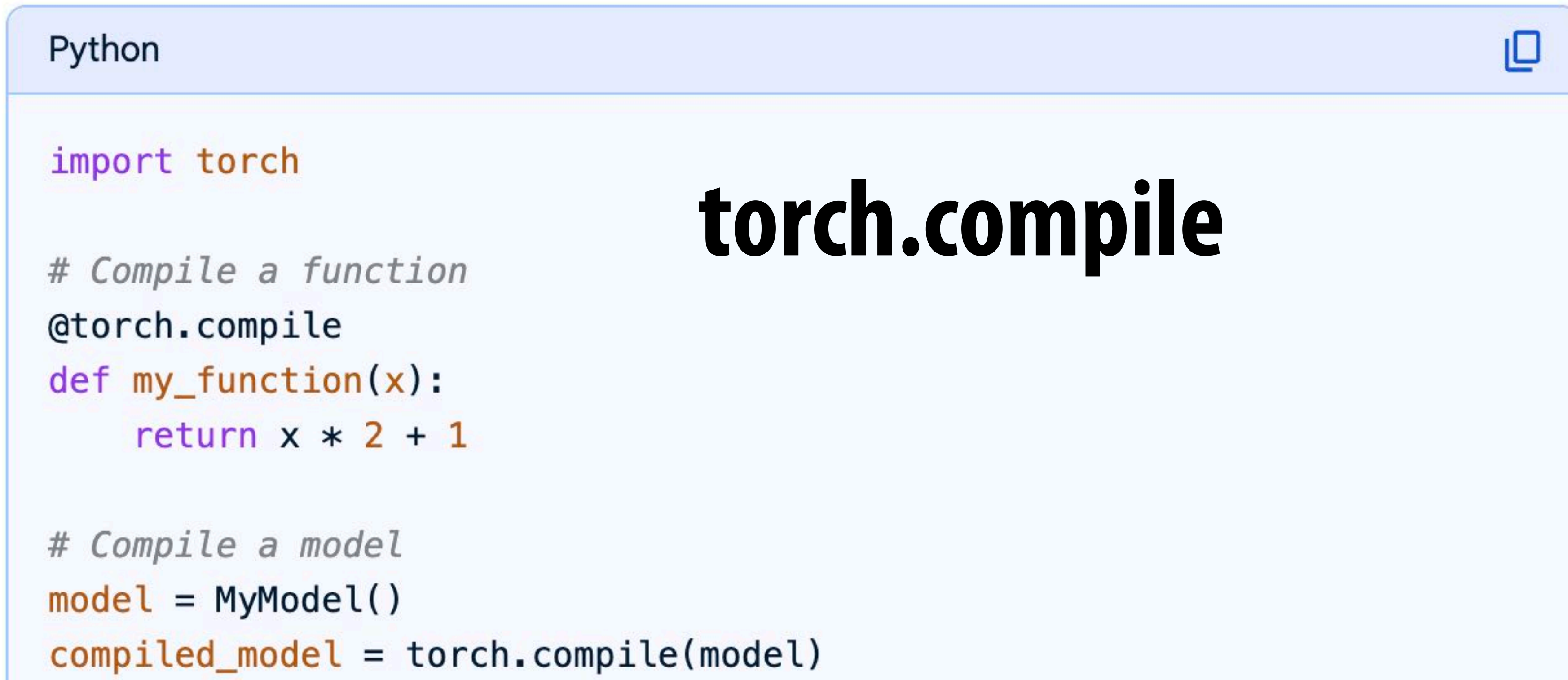


tvm Open Deep Learning Compiler Stack

license Apache 2.0 | build passing

Documentation | Contributors | Community | Release Notes

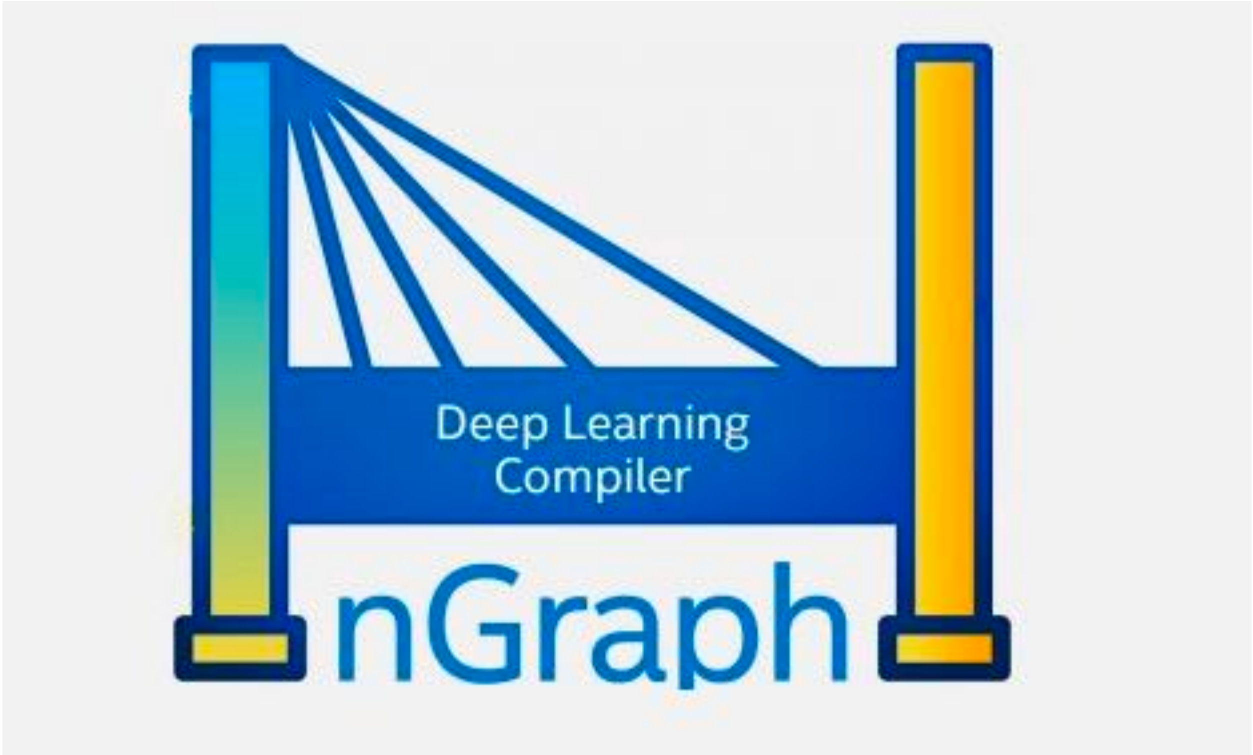
TVM is a compiler stack for deep learning systems. It is designed to close the gap between learning frameworks, and the performance- and efficiency-focused hardware backends. T frameworks to provide end to end compilation to different backends. Checkout the tvm sta information.

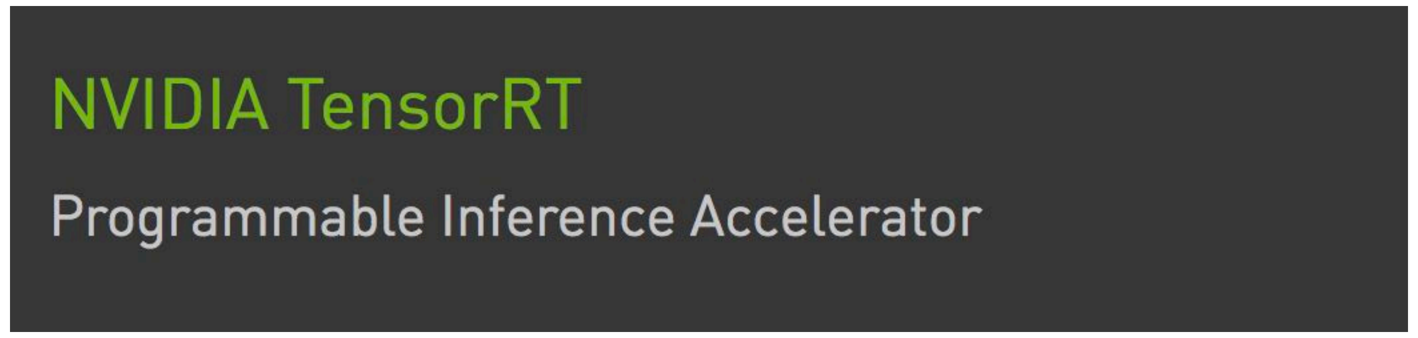


Python

```python
import torch

# Compile a function
@torch.compile
def my_function(x):
    return x * 2 + 1

# Compile a model
model = MyModel()
compiled_model = torch.compile(model)
```

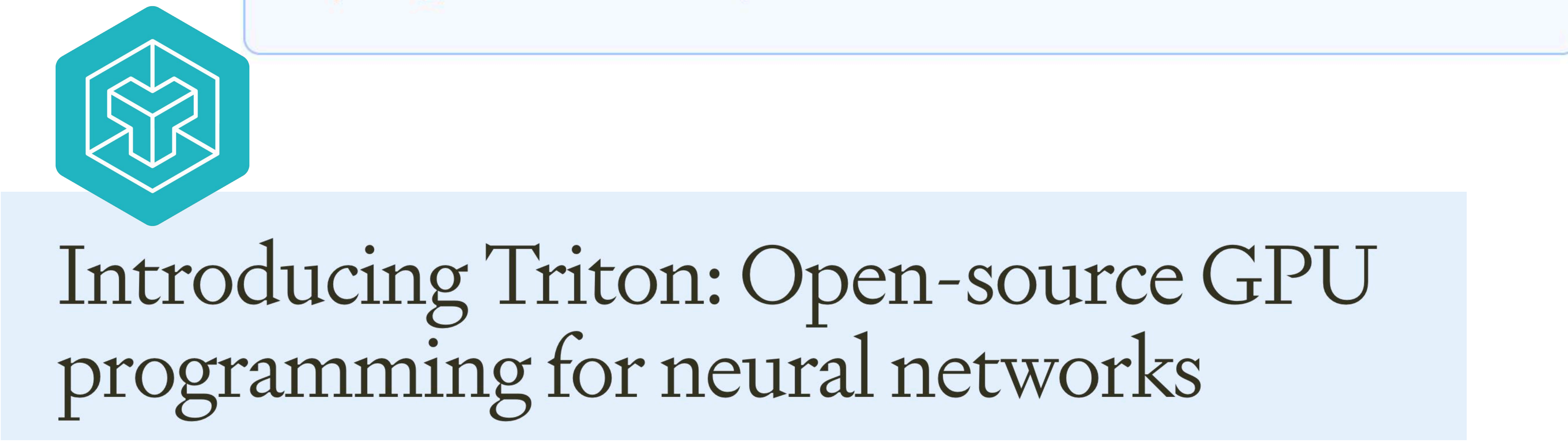
torch.compile





NVIDIA TensorRT

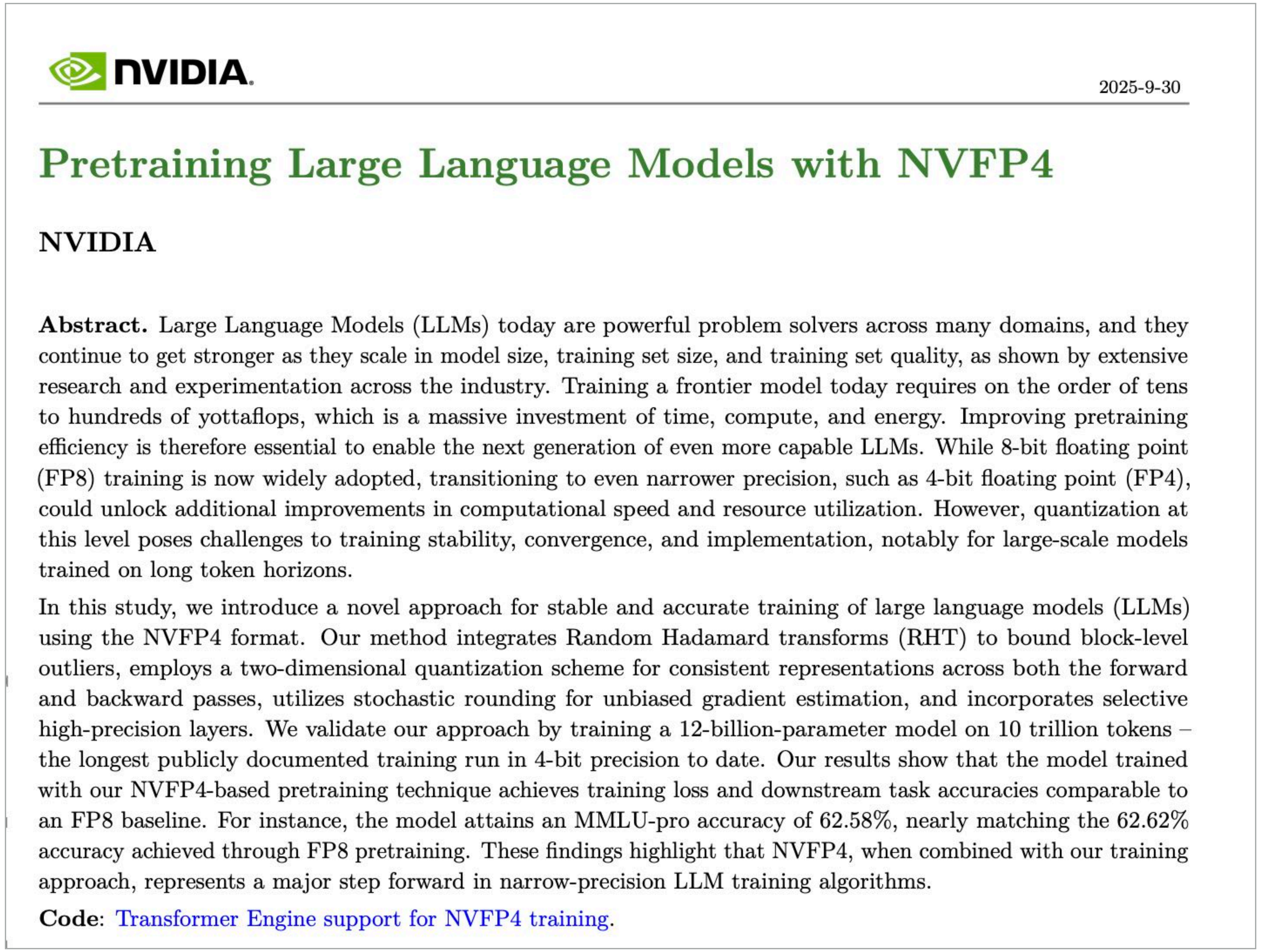
Programmable Inference Accelerator



Introducing Triton: Open-source GPU programming for neural networks

Another trick: use of low precision values

- Many efforts to use low precision values for DNN weights and intermediate activations
- 16 bit and 8-bit values are common
- Now moving into 4 bit values
- In the extreme case: 1-bit ;-)



NVIDIA. 2025-9-30

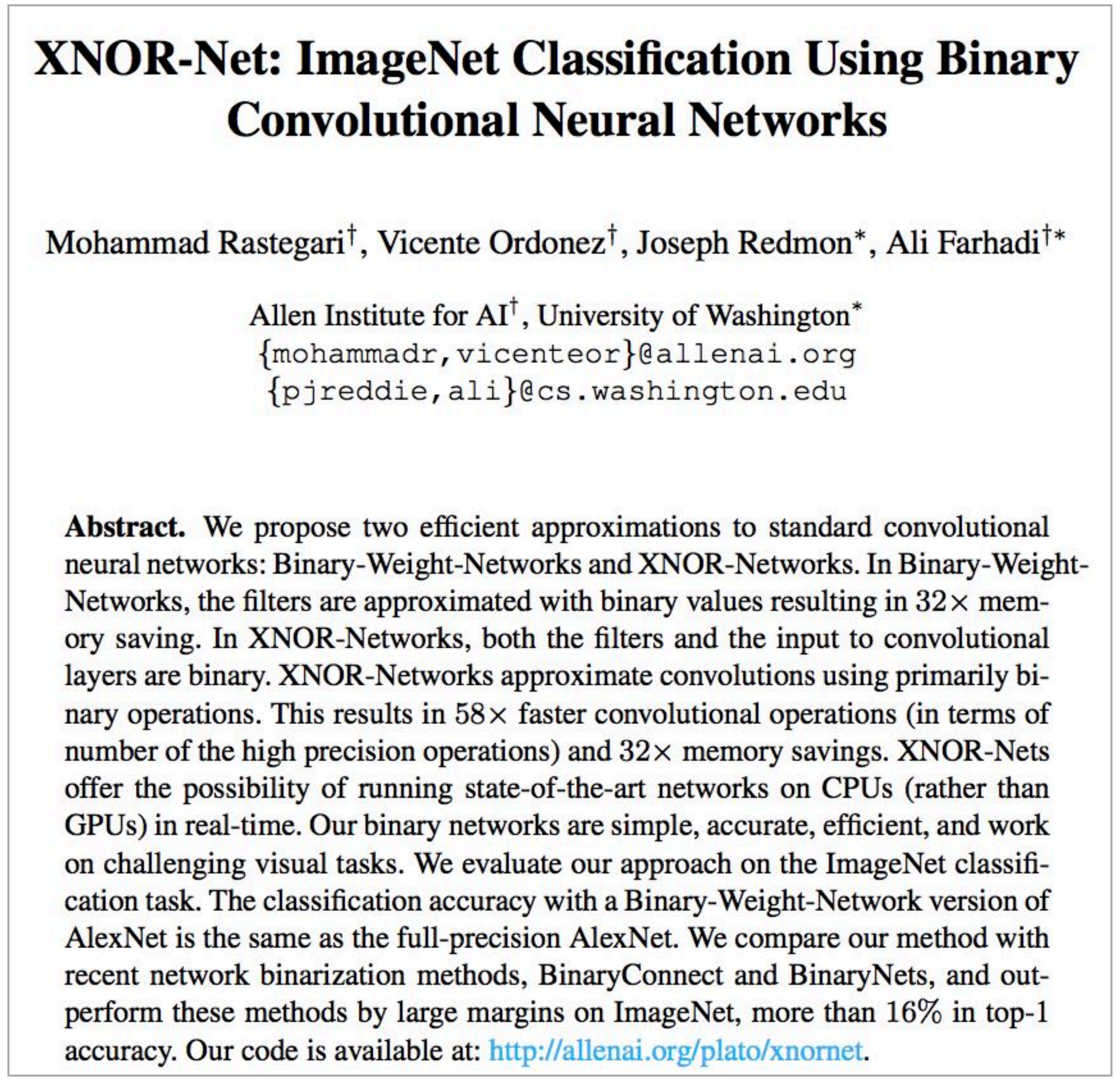
Pretraining Large Language Models with NVFP4

NVIDIA

Abstract. Large Language Models (LLMs) today are powerful problem solvers across many domains, and they continue to get stronger as they scale in model size, training set size, and training set quality, as shown by extensive research and experimentation across the industry. Training a frontier model today requires on the order of tens to hundreds of yottaflops, which is a massive investment of time, compute, and energy. Improving pretraining efficiency is therefore essential to enable the next generation of even more capable LLMs. While 8-bit floating point (FP8) training is now widely adopted, transitioning to even narrower precision, such as 4-bit floating point (FP4), could unlock additional improvements in computational speed and resource utilization. However, quantization at this level poses challenges to training stability, convergence, and implementation, notably for large-scale models trained on long token horizons.

In this study, we introduce a novel approach for stable and accurate training of large language models (LLMs) using the NVFP4 format. Our method integrates Random Hadamard transforms (RHT) to bound block-level outliers, employs a two-dimensional quantization scheme for consistent representations across both the forward and backward passes, utilizes stochastic rounding for unbiased gradient estimation, and incorporates selective high-precision layers. We validate our approach by training a 12-billion-parameter model on 10 trillion tokens – the longest publicly documented training run in 4-bit precision to date. Our results show that the model trained with our NVFP4-based pretraining technique achieves training loss and downstream task accuracies comparable to an FP8 baseline. For instance, the model attains an MMLU-pro accuracy of 62.58%, nearly matching the 62.62% accuracy achieved through FP8 pretraining. These findings highlight that NVFP4, when combined with our training approach, represents a major step forward in narrow-precision LLM training algorithms.

Code: Transformer Engine support for NVFP4 training.



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.

Optimization techniques

- Better algorithms: manually designing better ML models
  - Common parameters: depth of network, width of filters, number of filters per layer, convolutional stride, etc.
  - Common to perform automatic search for efficient topologies

- Software optimization: Good scheduling of performance-critical operations
  - Loop blocking/tiling, fusion
  - Typically optimized manually by humans (but significant research efforts to automate scheduling)

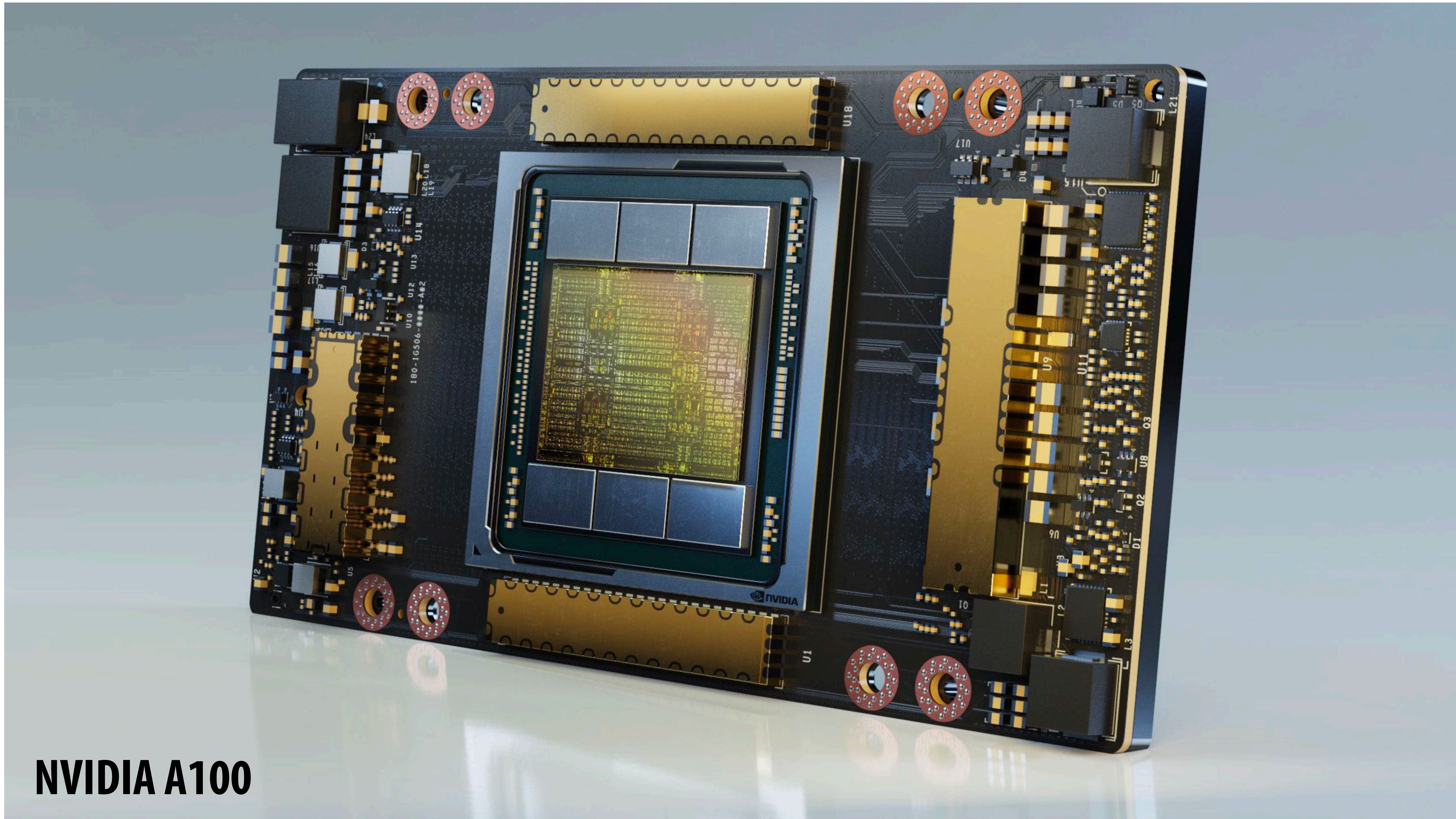
- Forms of approximation: compressing models
  - Lower bit precision

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 matrix-matrix computations benefit from flop-rich GPU architectures
  - Highly-optimized library of kernels exist for GPUs (cuDNN)

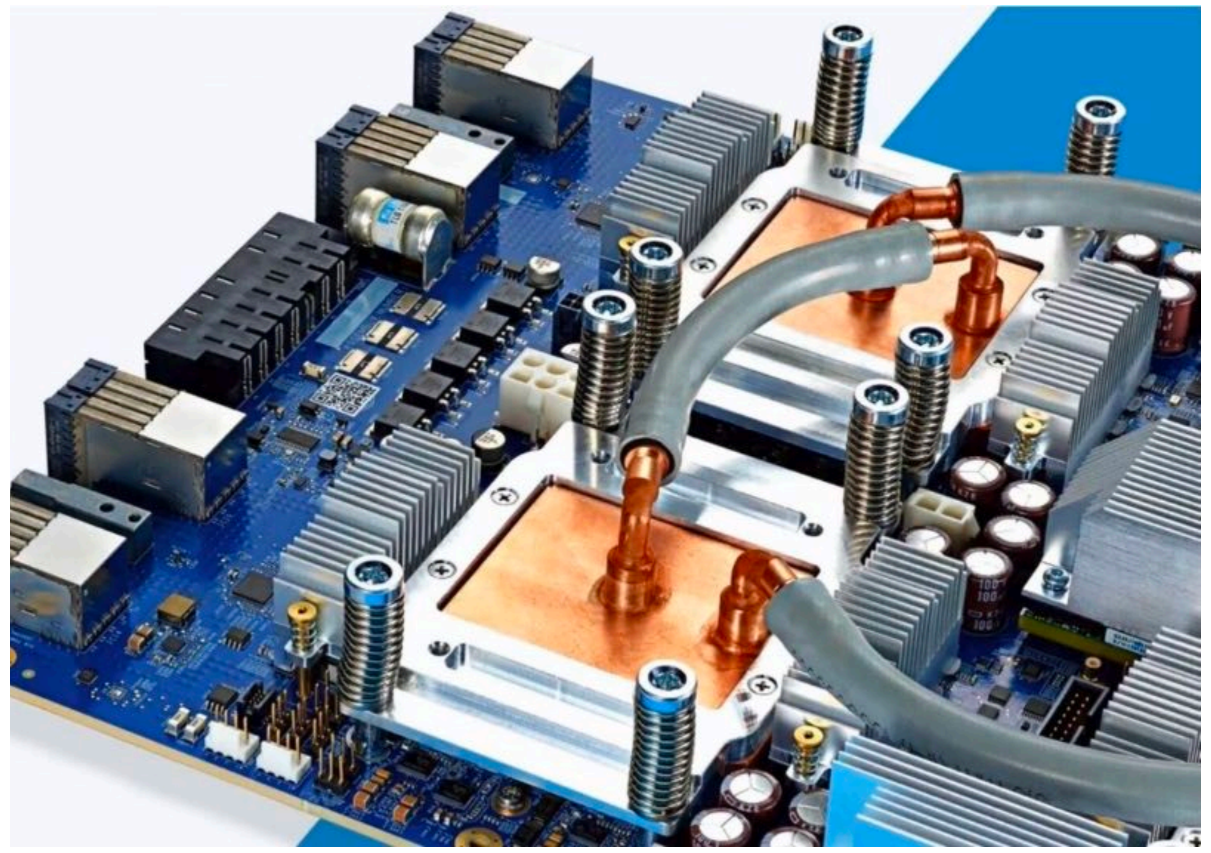


NVIDIA A100

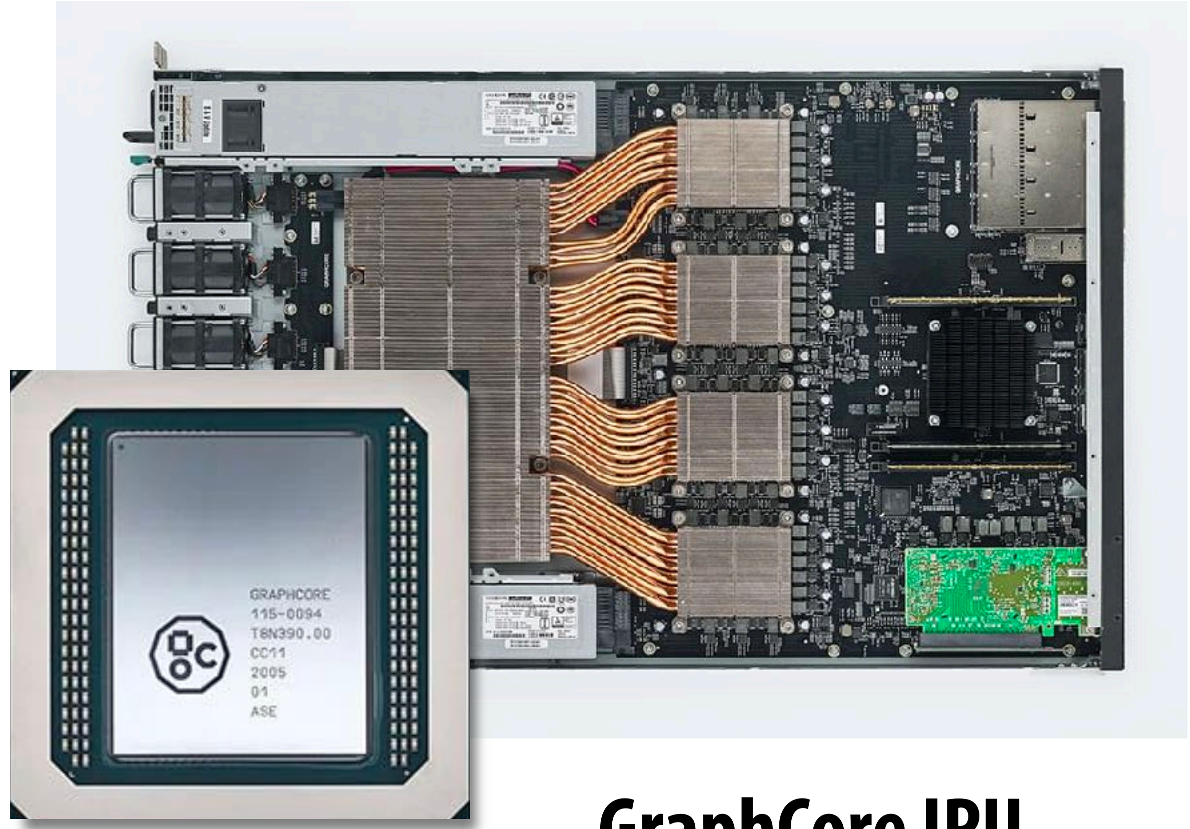
Why might a GPU be a sub-optimal platform for DNN evaluation?

(Hint: is a general purpose processor really needed?)

Next time: maximizing efficiency via specialized hardware acceleration for DNN inference/training



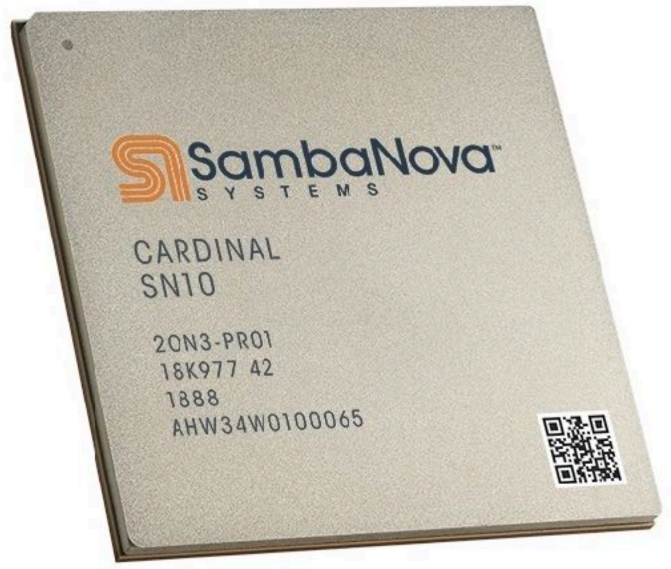
Google TPU3



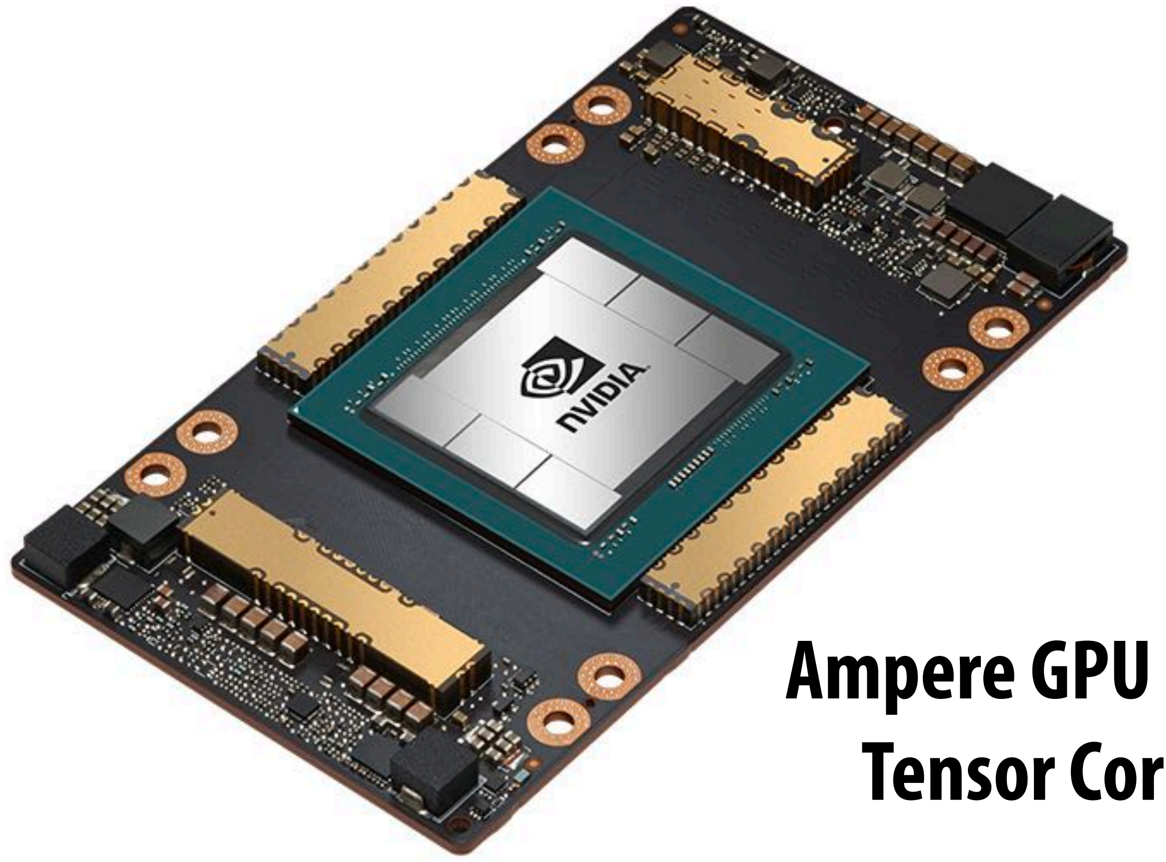
GraphCore IPU

Apple Neural Engine

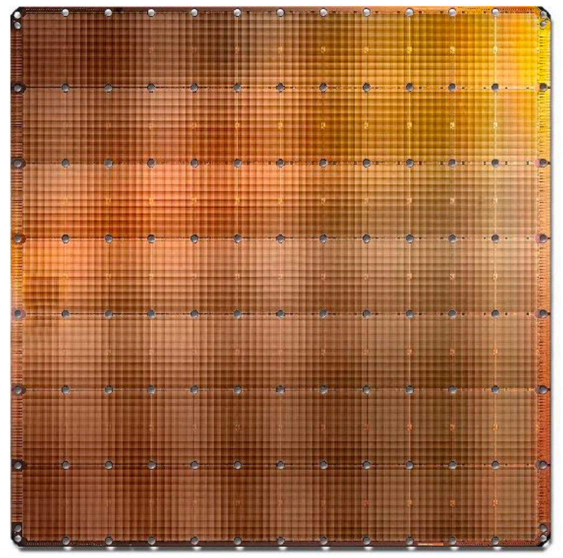
Intel Deep Learning Inference Accelerator



SambaNova Cardinal SN10



Ampere GPU with Tensor Cores



Cerebras Wafer Scale Engine