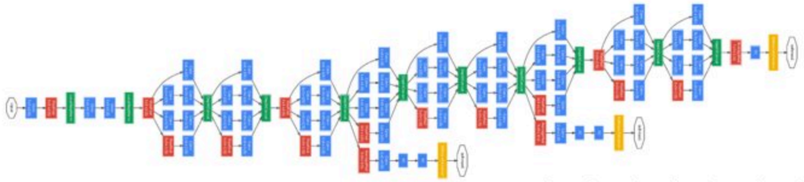
Lecture 10:

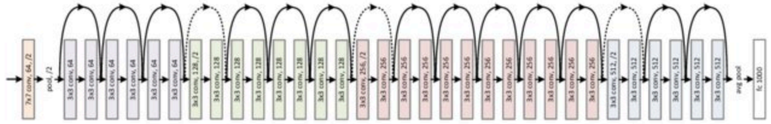
Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2024

Efficiency challenge

Many DNN topologies
(Many variants on common backbones)





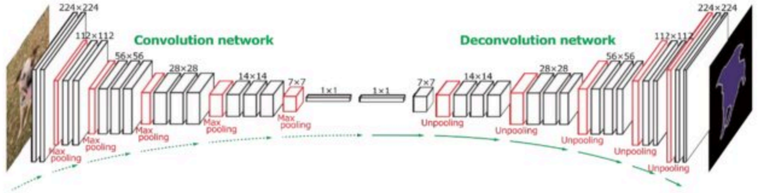


Table 1. MobileNet Body Architecture

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	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

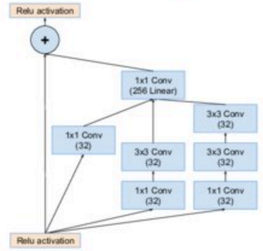
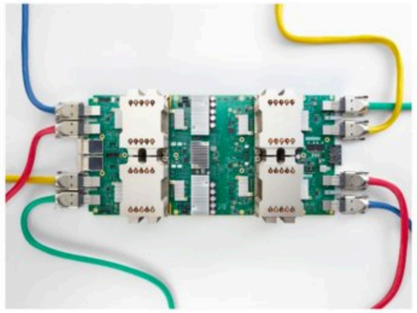
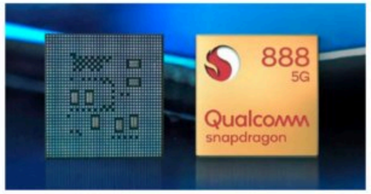


Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

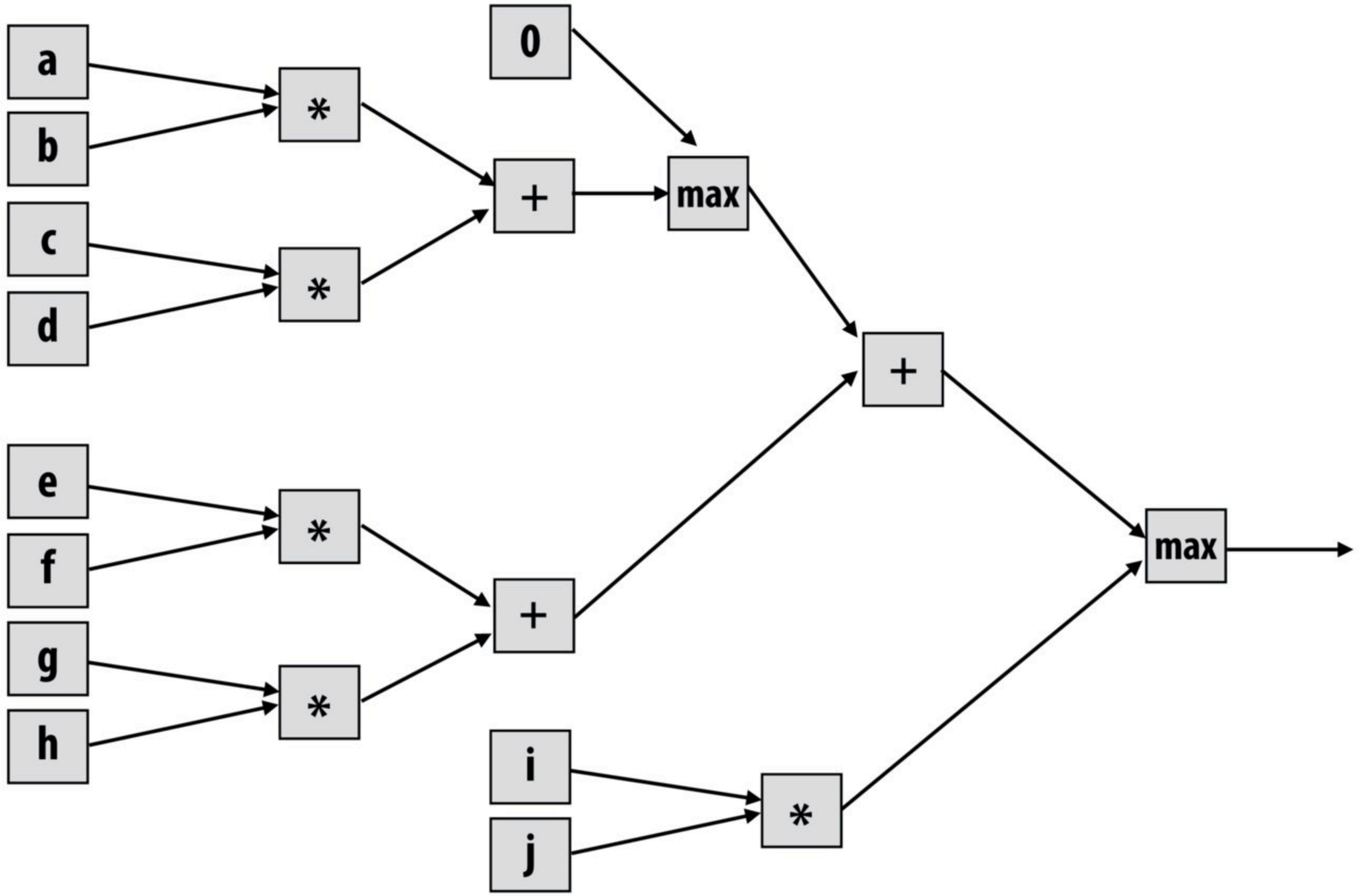
Many Target Devices





Mini-intro/review: Convolutional Neural Networks

Consider the following expression

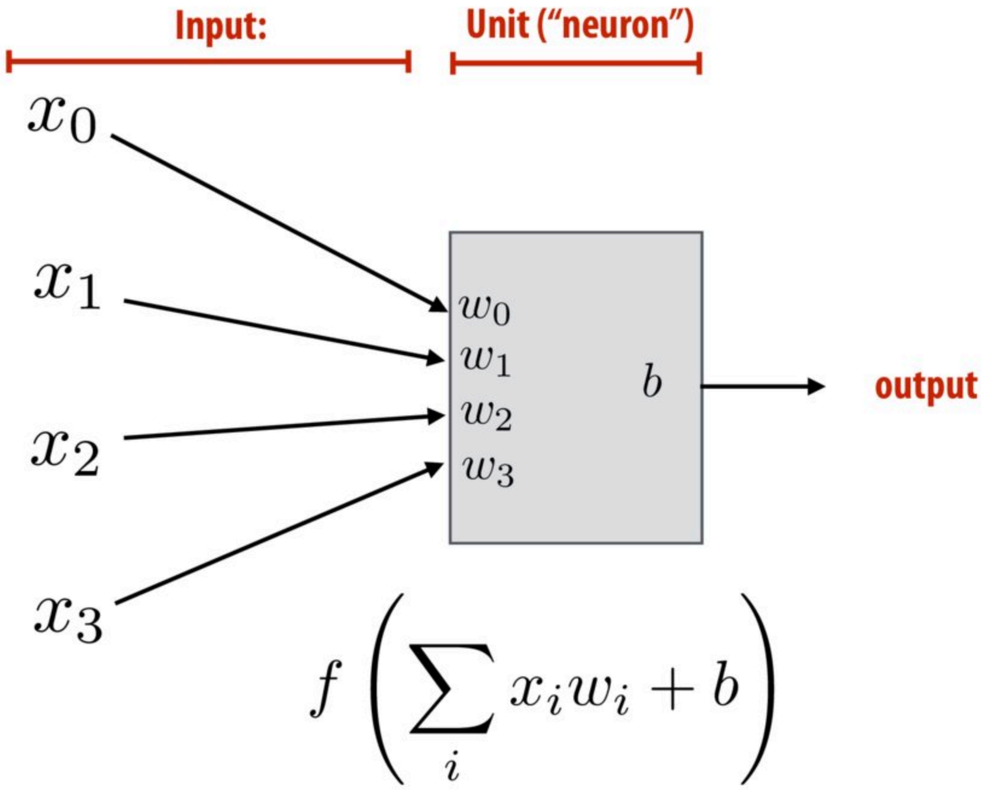



```
max( max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j)
```

What is a deep neural network?

A basic unit:

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



Example: rectified linear unit (ReLU)

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

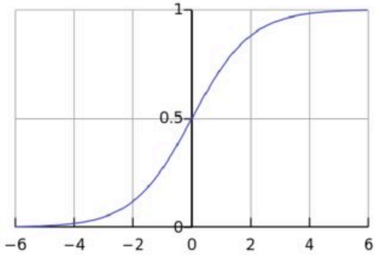
Basic computational interpretation:

It is just a circuit!

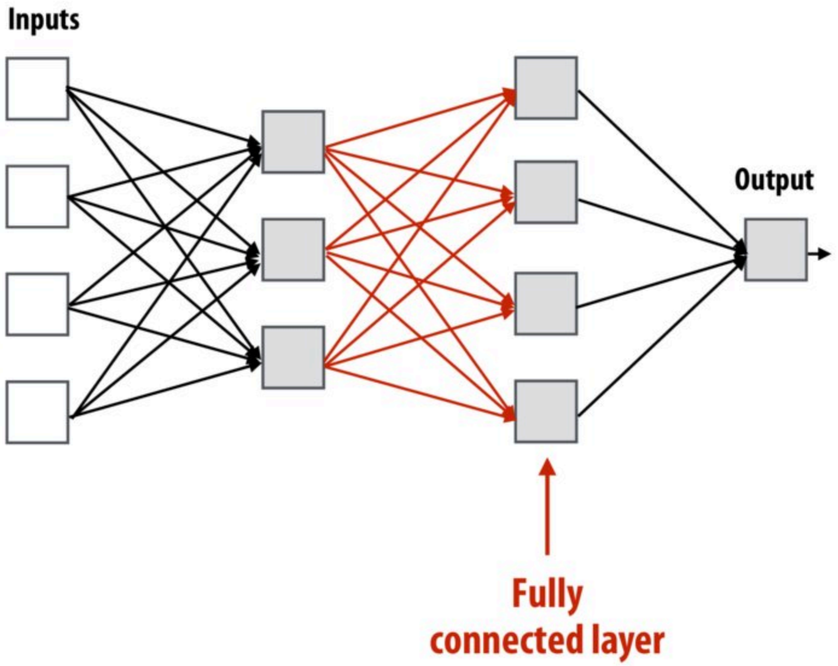
Machine learning interpretation:

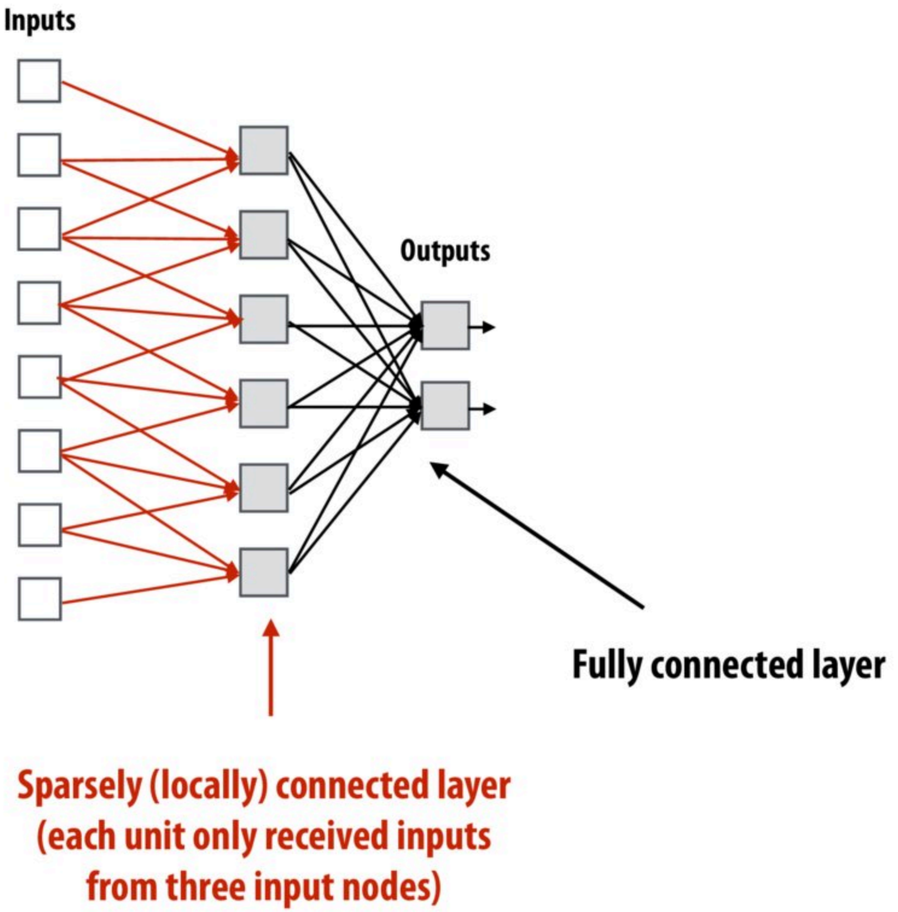
Binary classifier: interpret output as the probability of one class

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

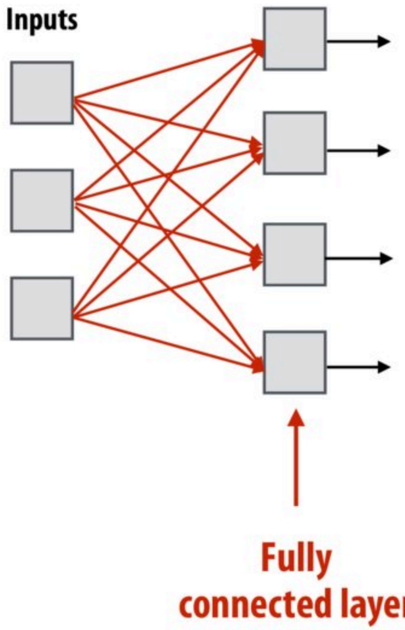


Deep neural network: topology





Fully connected layer as matrix-vector product



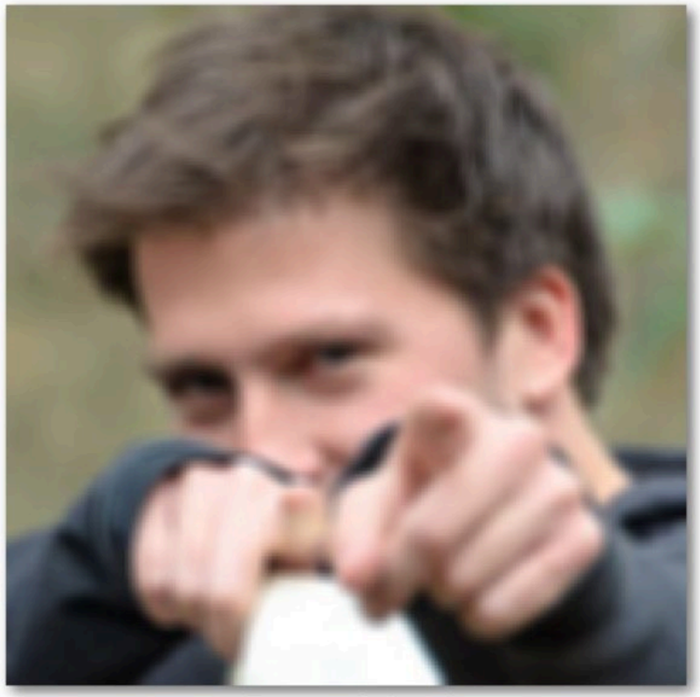
$$f\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{22} & w_{21} & w_{22} \\ w_{32} & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

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;
  }
}
```

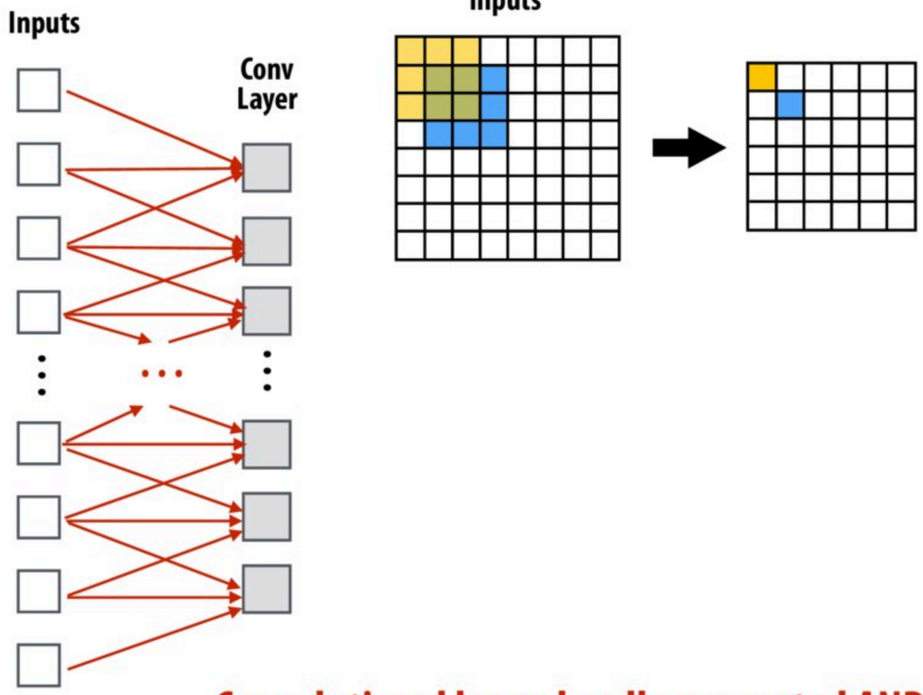
The code on the previous slide performed a 3x3 blur



(Zoomed view)

Image convolution (3x3 conv)

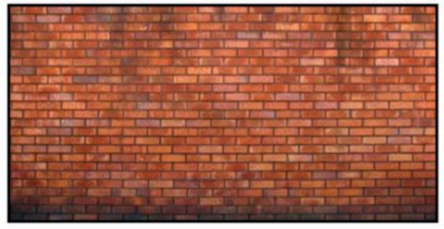
```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];


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;
  }
}
```

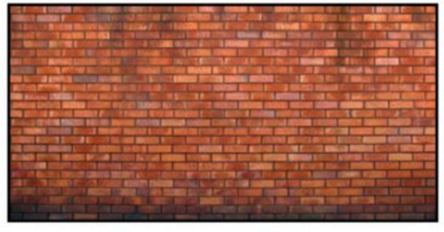
Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):
(note: network illustration above only shows links for a 1D conv: a.k.a. one iteration of `ii` loop)

Gradient detection filters



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

Responds to horizontal gradients



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

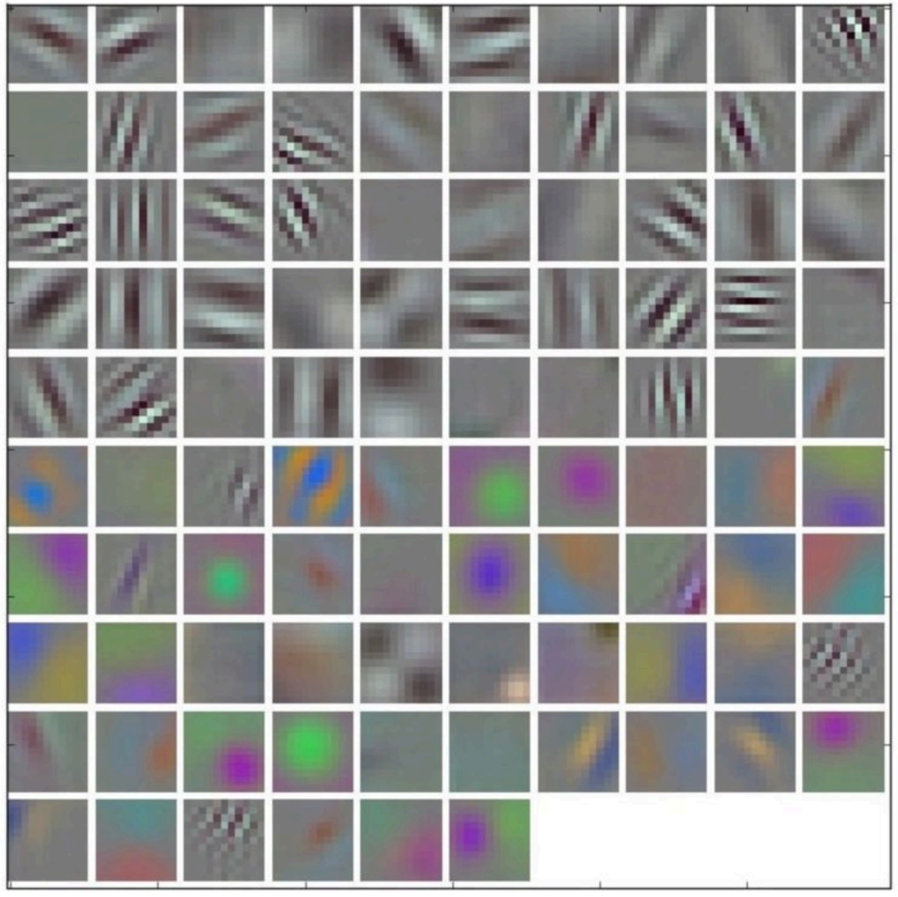
Responds to vertical gradients

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

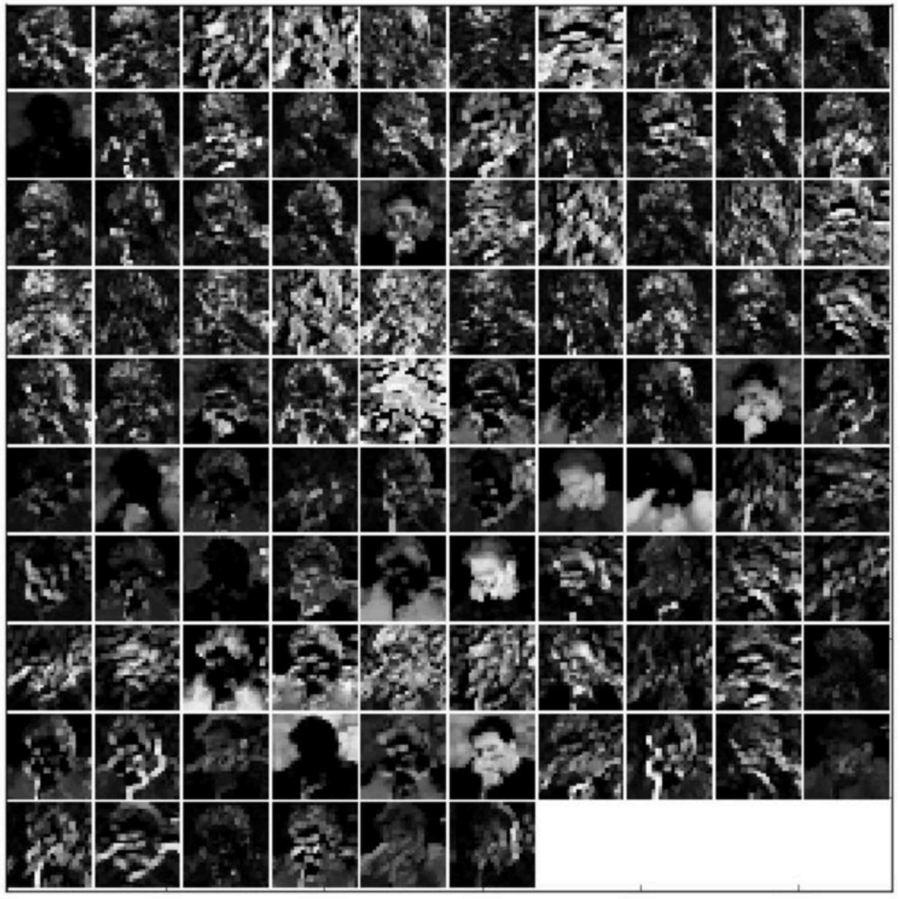
Applying many filters to an image at once

Input RGB image (W x H x 3)

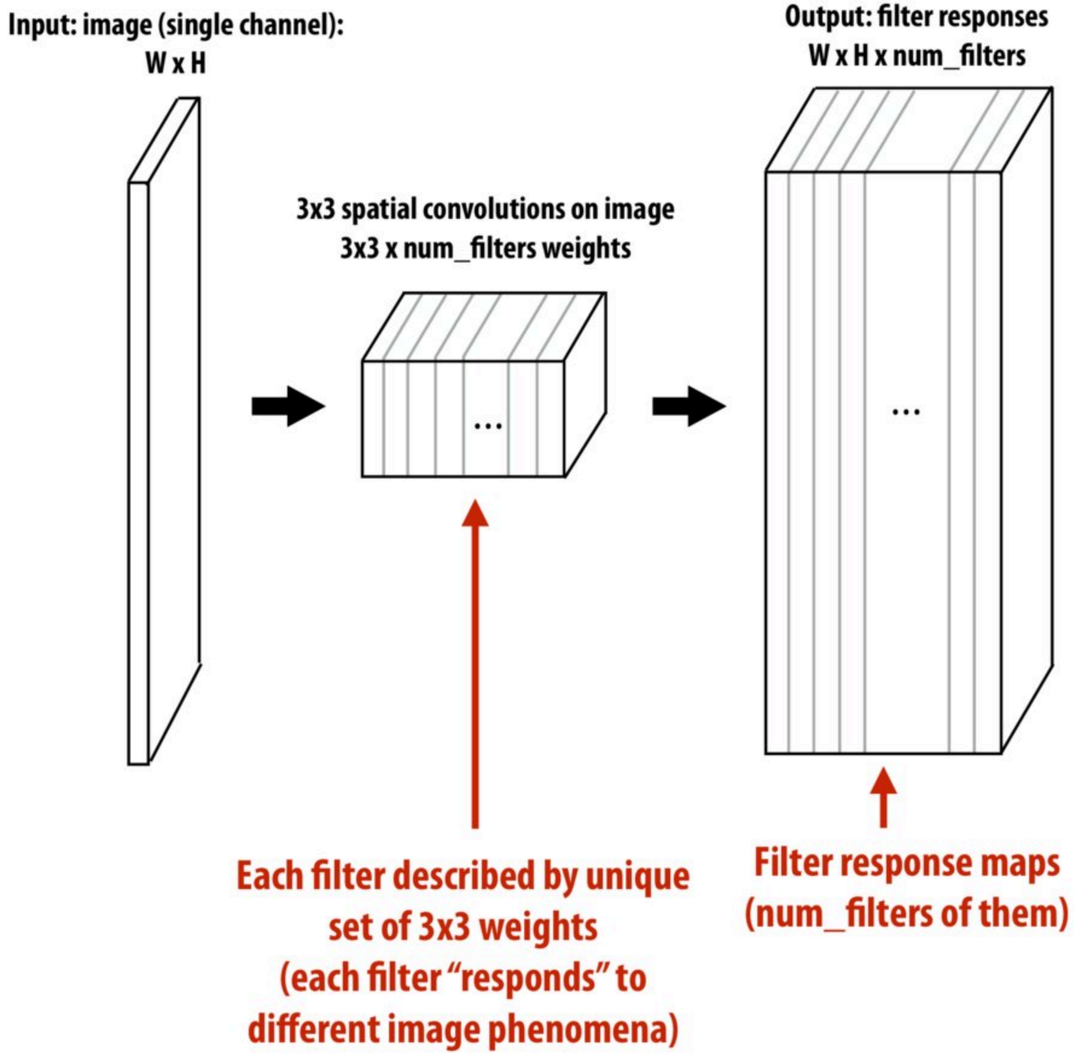
96 11x11x3 filters
(3D because they operate on RGB)



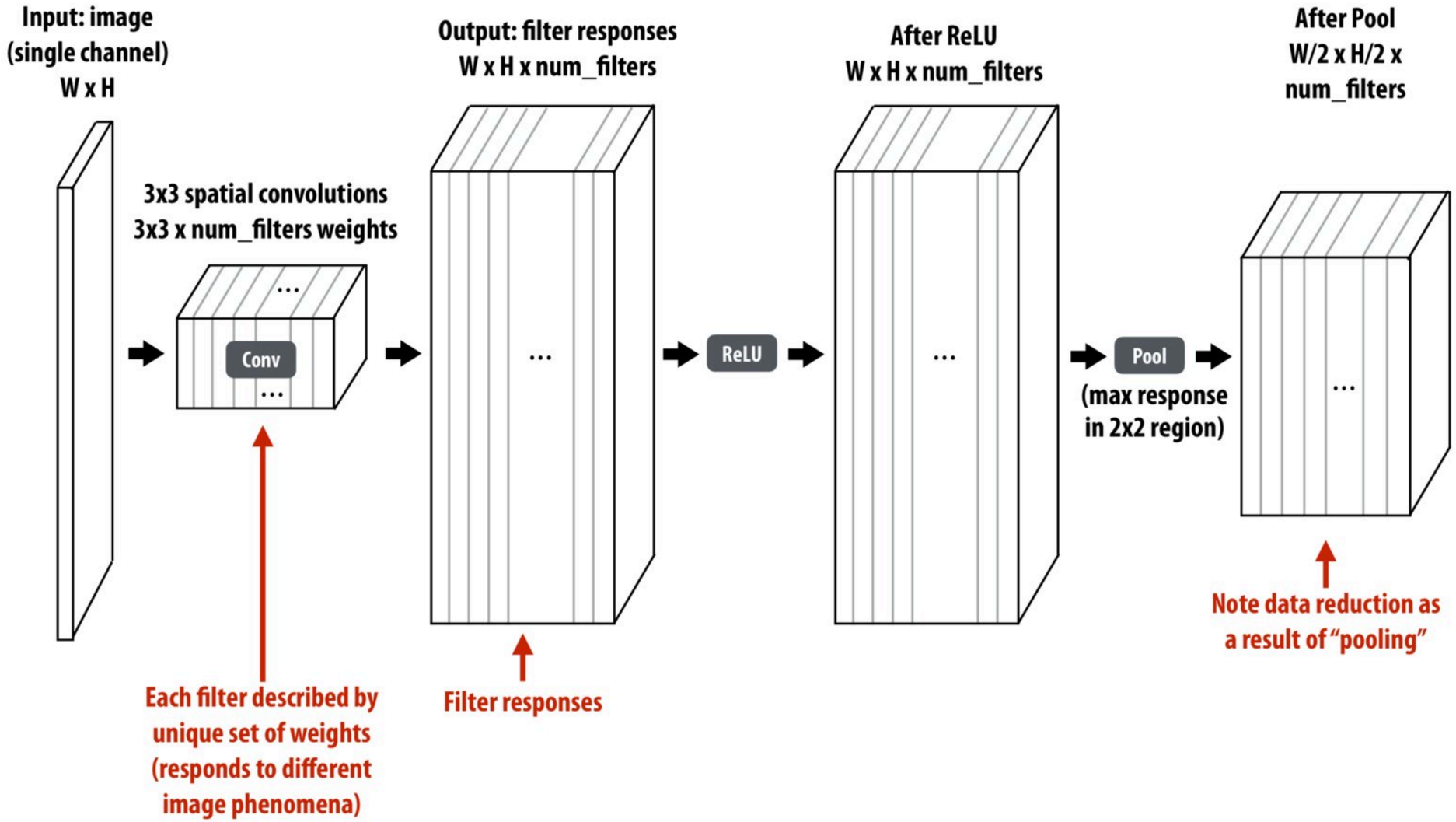
96 responses (normalized)



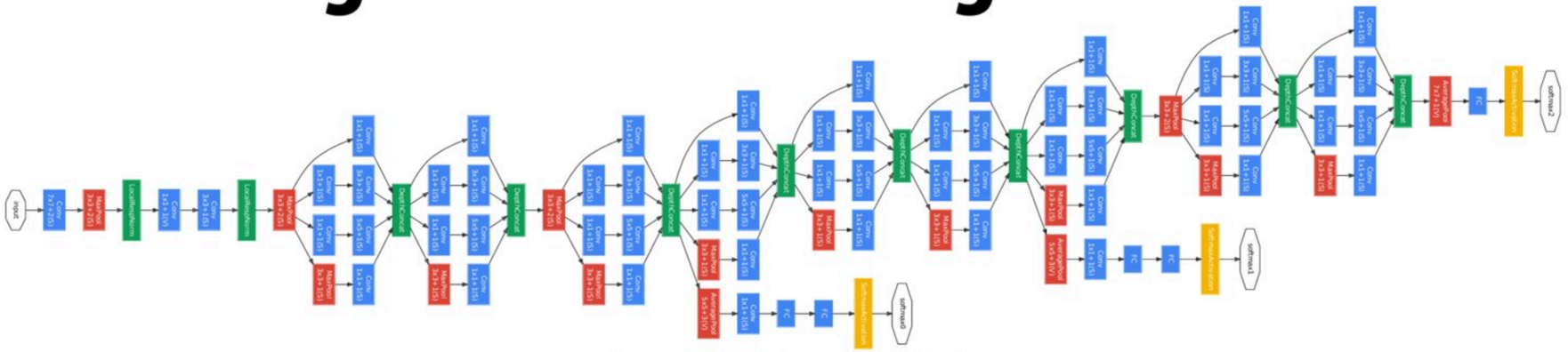
Applying many filters to an image at once



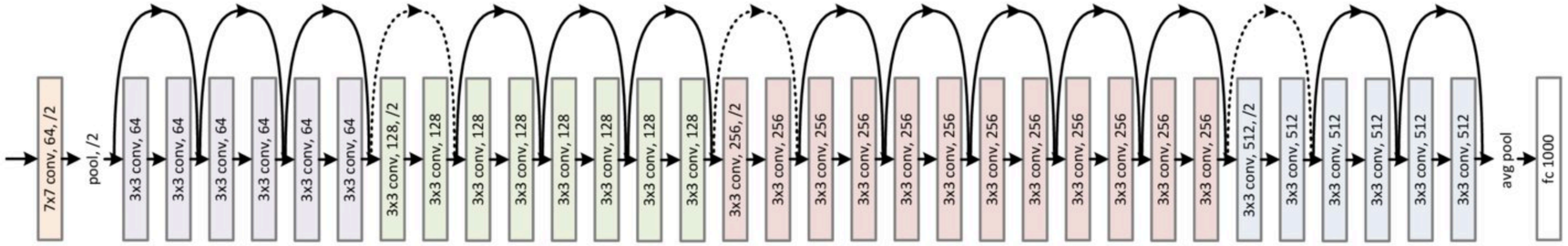
Adding additional layers



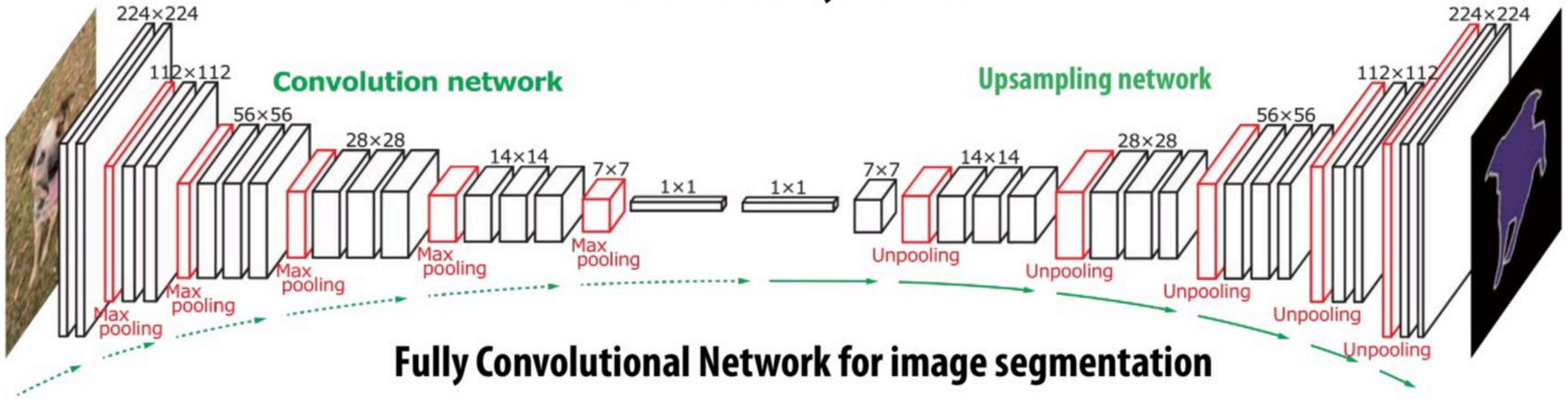
More recent image understanding networks



Inception (GoogleLeNet)



ResNet (34 layer version)



Fully Convolutional Network for image segmentation

Efficiently implementing convolution layers

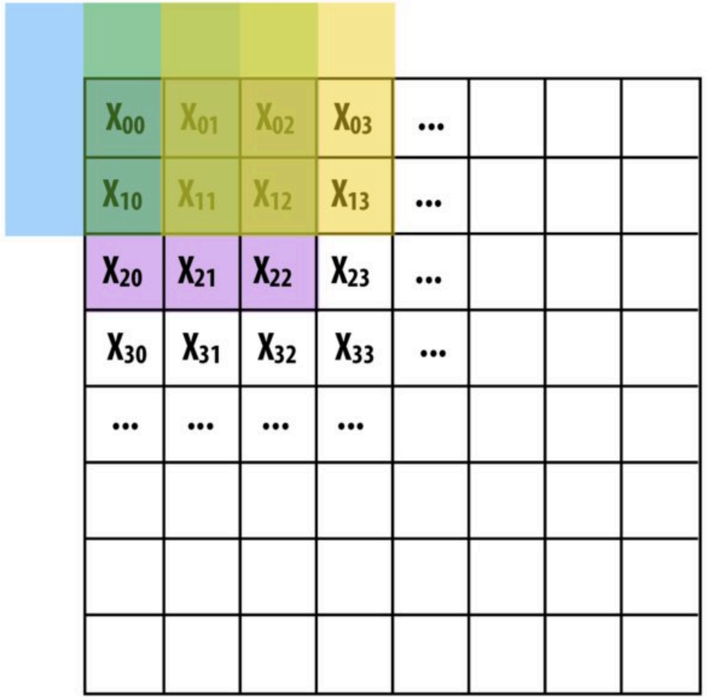
Direct implementation of conv layer (batched)

```
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)

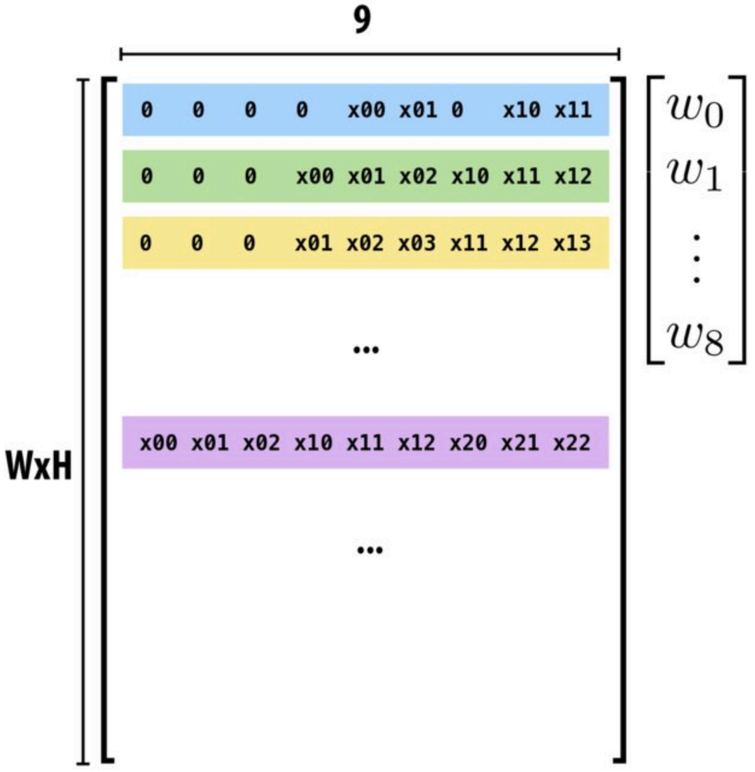
3x3 convolution as matrix-vector product ("explicit gemm")

Construct matrix from elements of input image



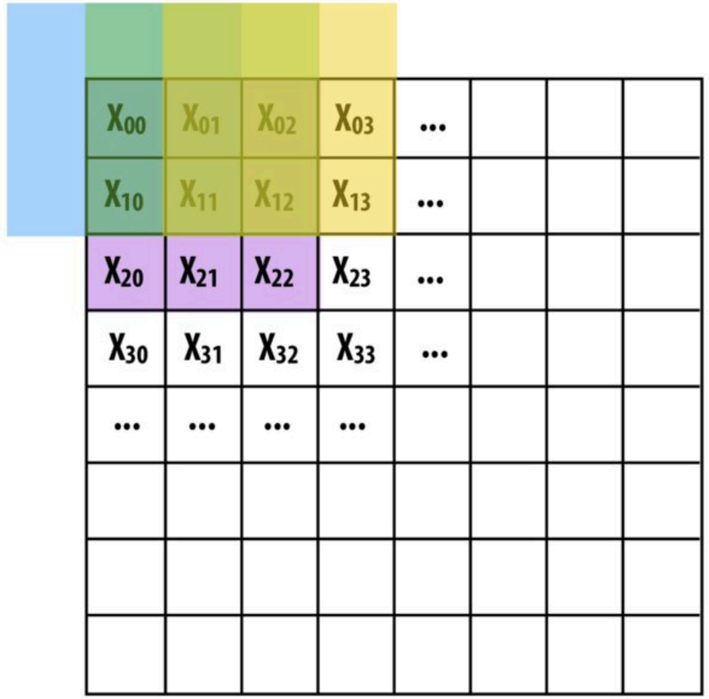
O(N) storage overhead for filter with N elements

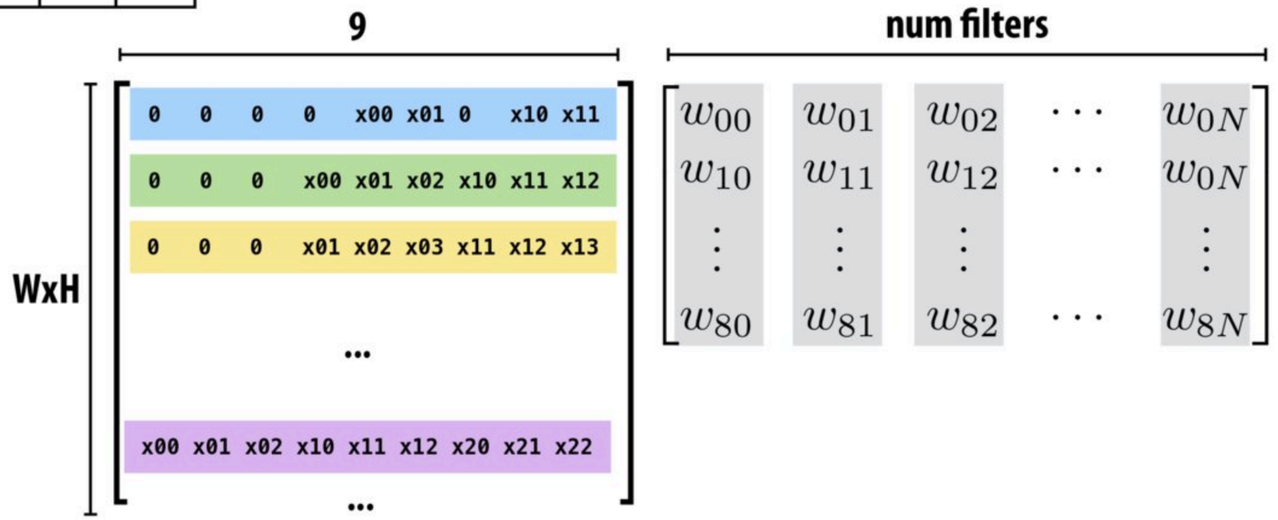
Must construct input data matrix



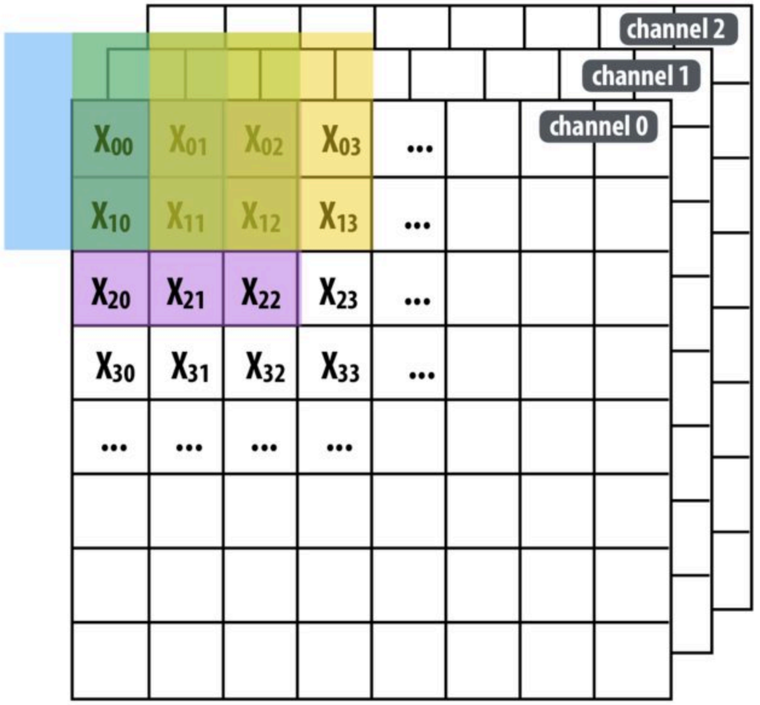
Note: 0-pad matrix

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



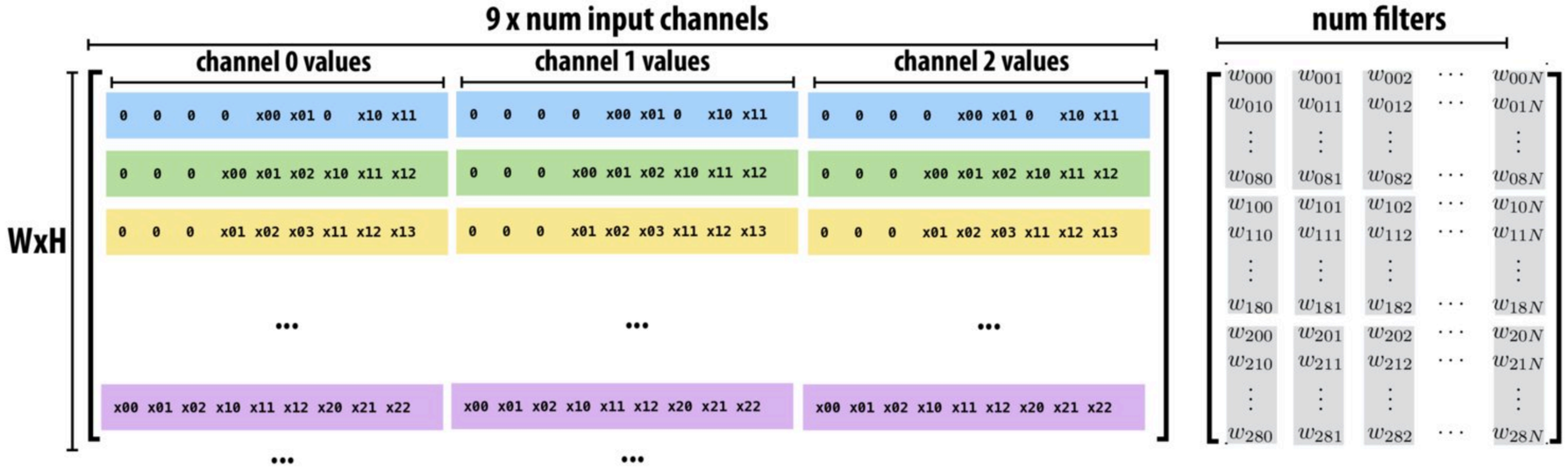


Multiple convolutions on multiple input channels



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



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

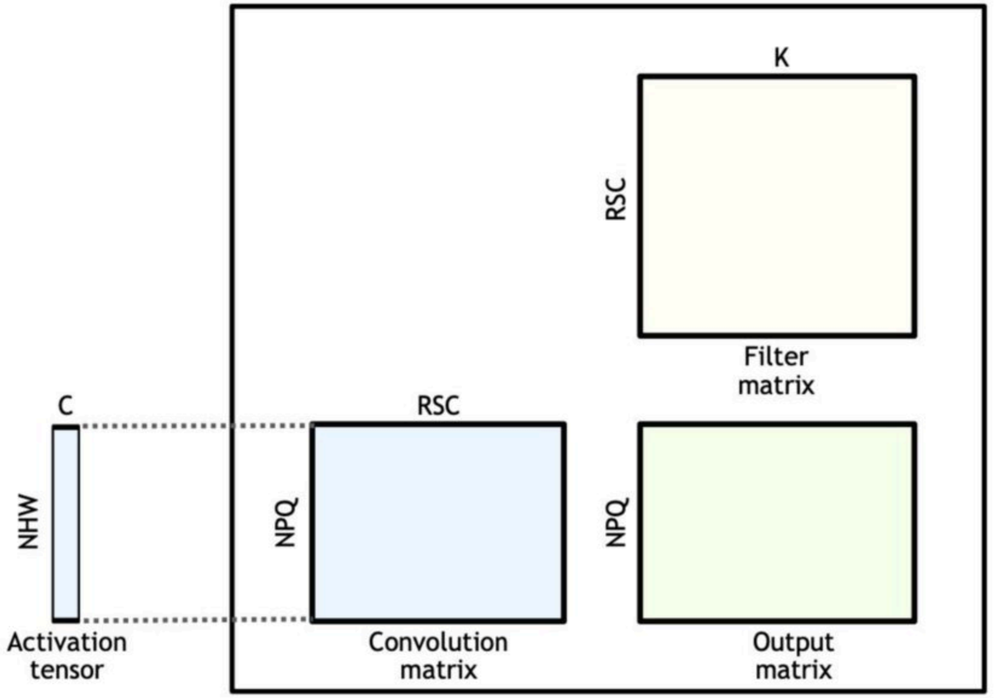


Image credit: NVIDIA

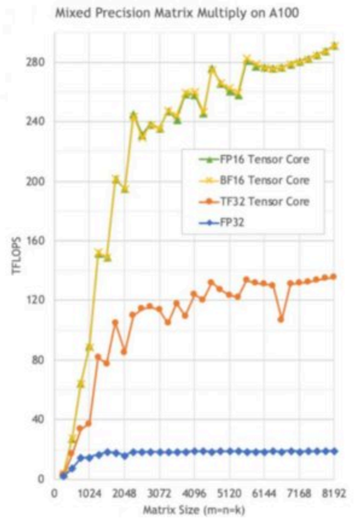
High performance implementations of GEMM exist

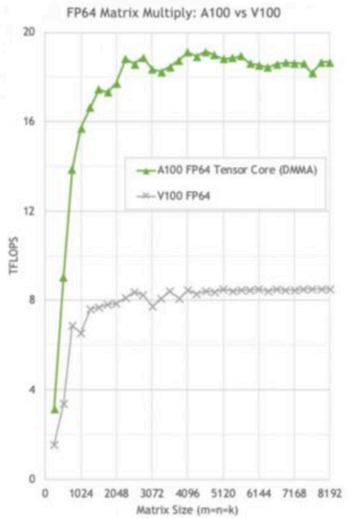
cuBLAS Performance

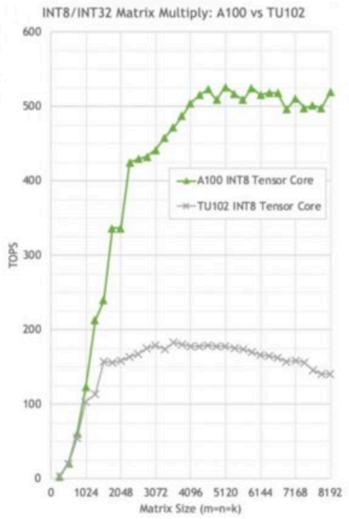
The cuBLAS library is highly optimized for performance on NVIDIA GPUs, and leverages tensor cores for acceleration of low and mixed precision matrix multiplication.

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 x 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)

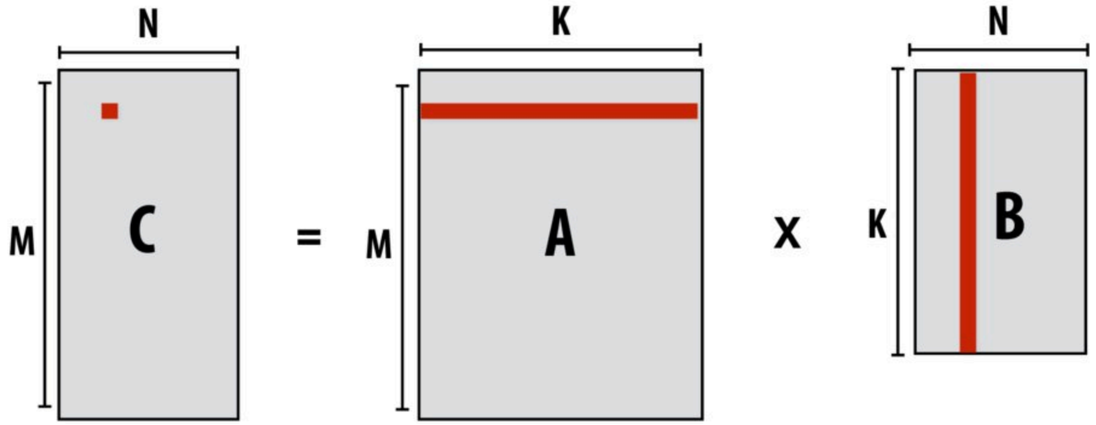
Download as Part of the Toolkit

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];
```

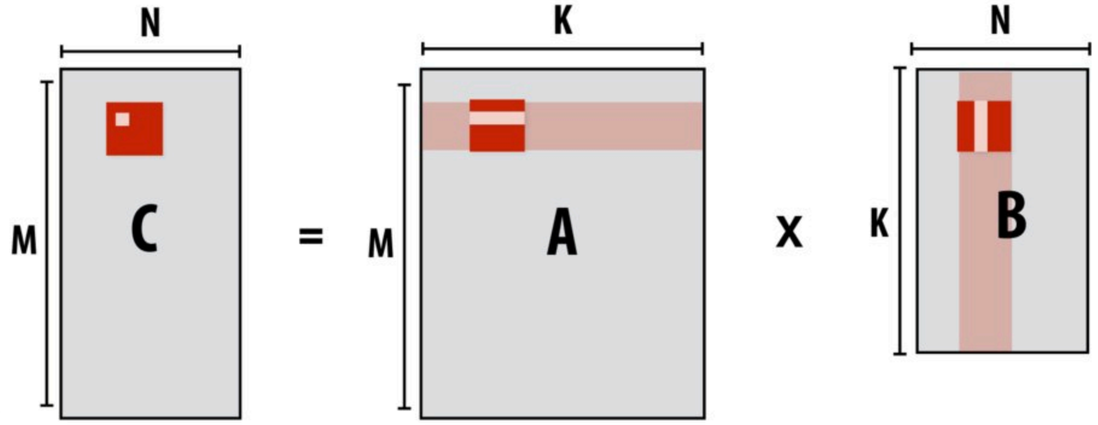


What is the problem with this implementation?

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

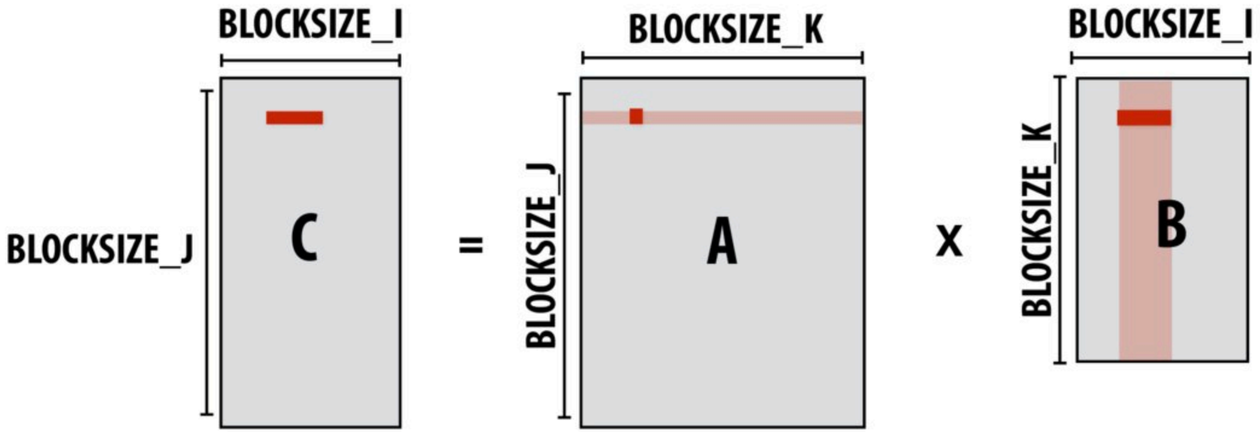
Exploit multiple levels of memory hierarchy

```
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



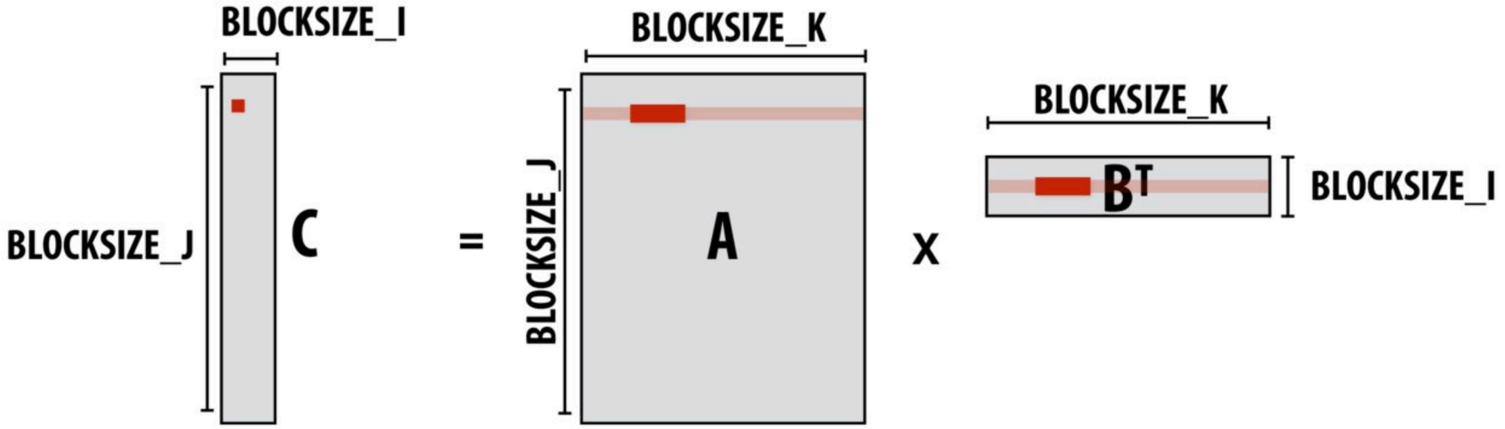

```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
      simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in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



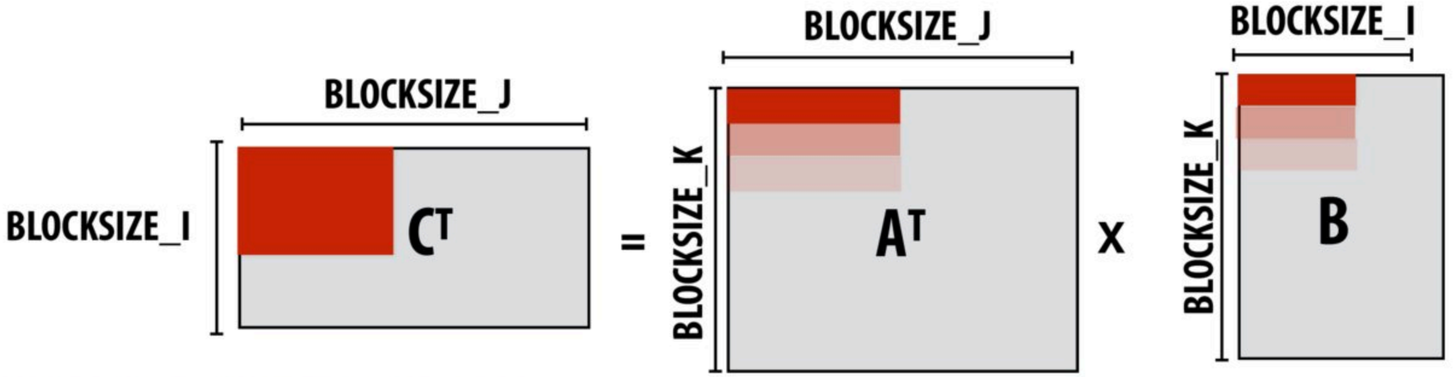

```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      // C_scalar += dot(row of A,row of B)
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
        C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][[kblock+k]);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

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

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

Image credit: NVIDIA

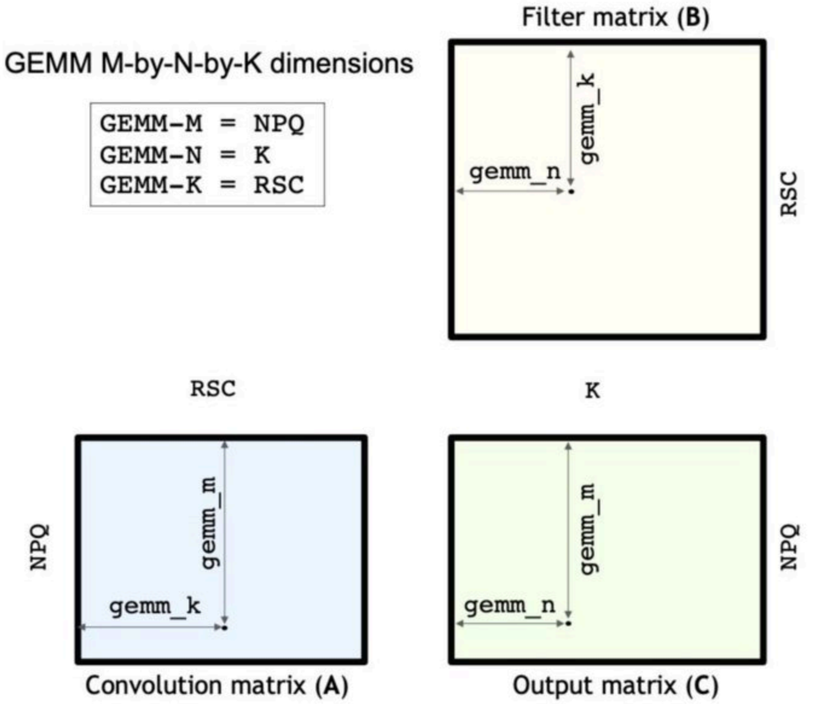
GEMM TRIPLE NEST LOOP

```
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
```



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

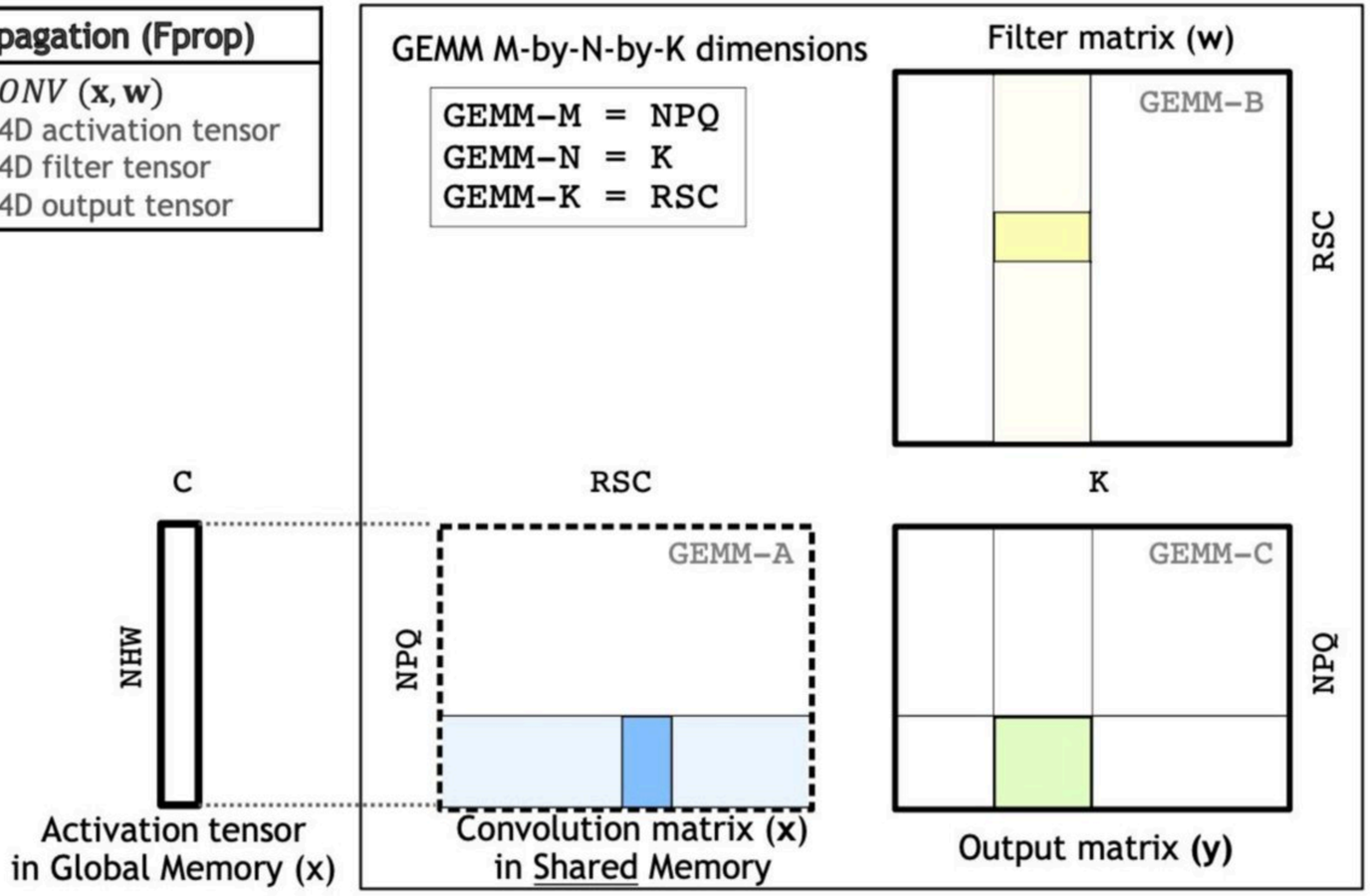
Optimization: do not materialize full matrix ("implicit gemm")

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

Forward Propagation (Fprop)

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

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



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)

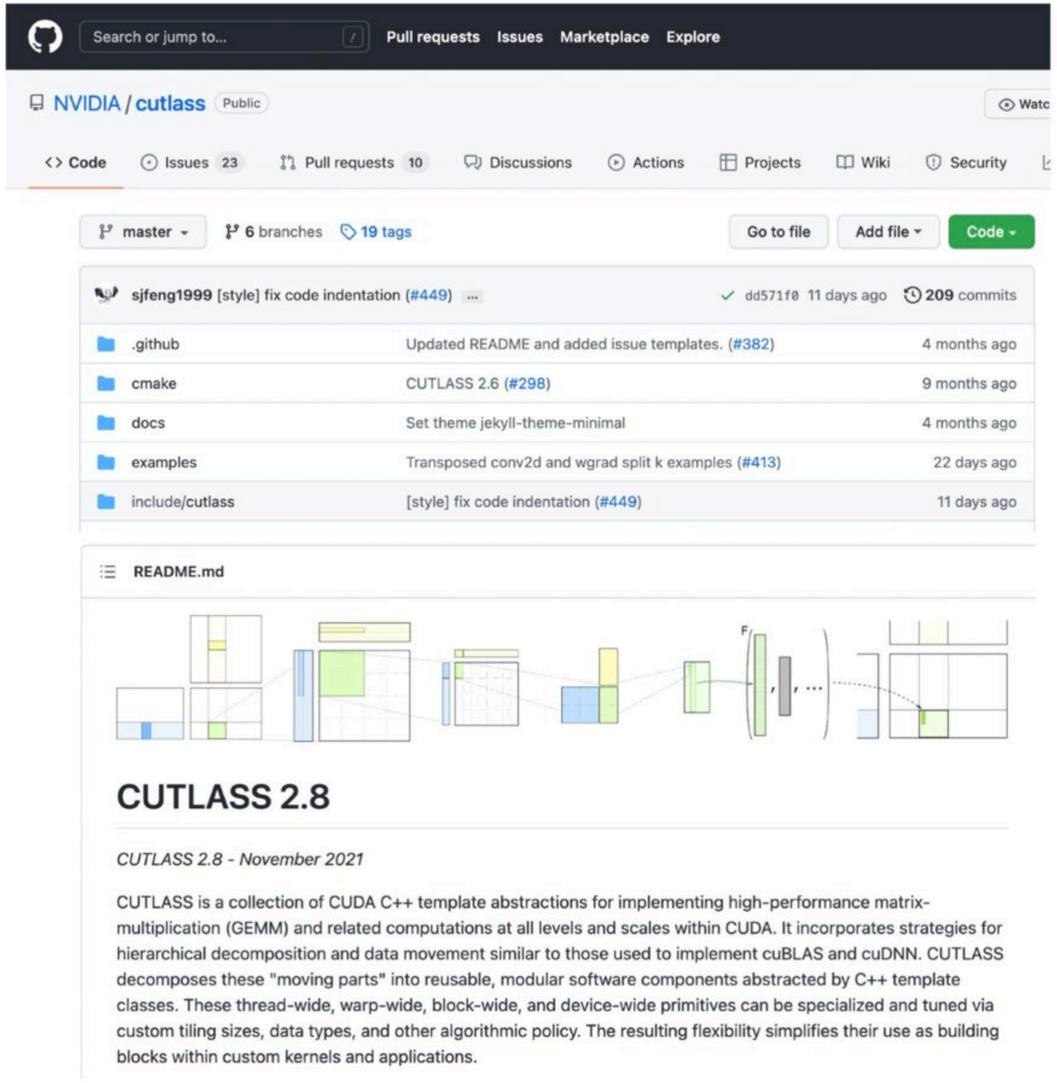
Symbol reference:

- **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)



Fast (in-shared memory) GEMM

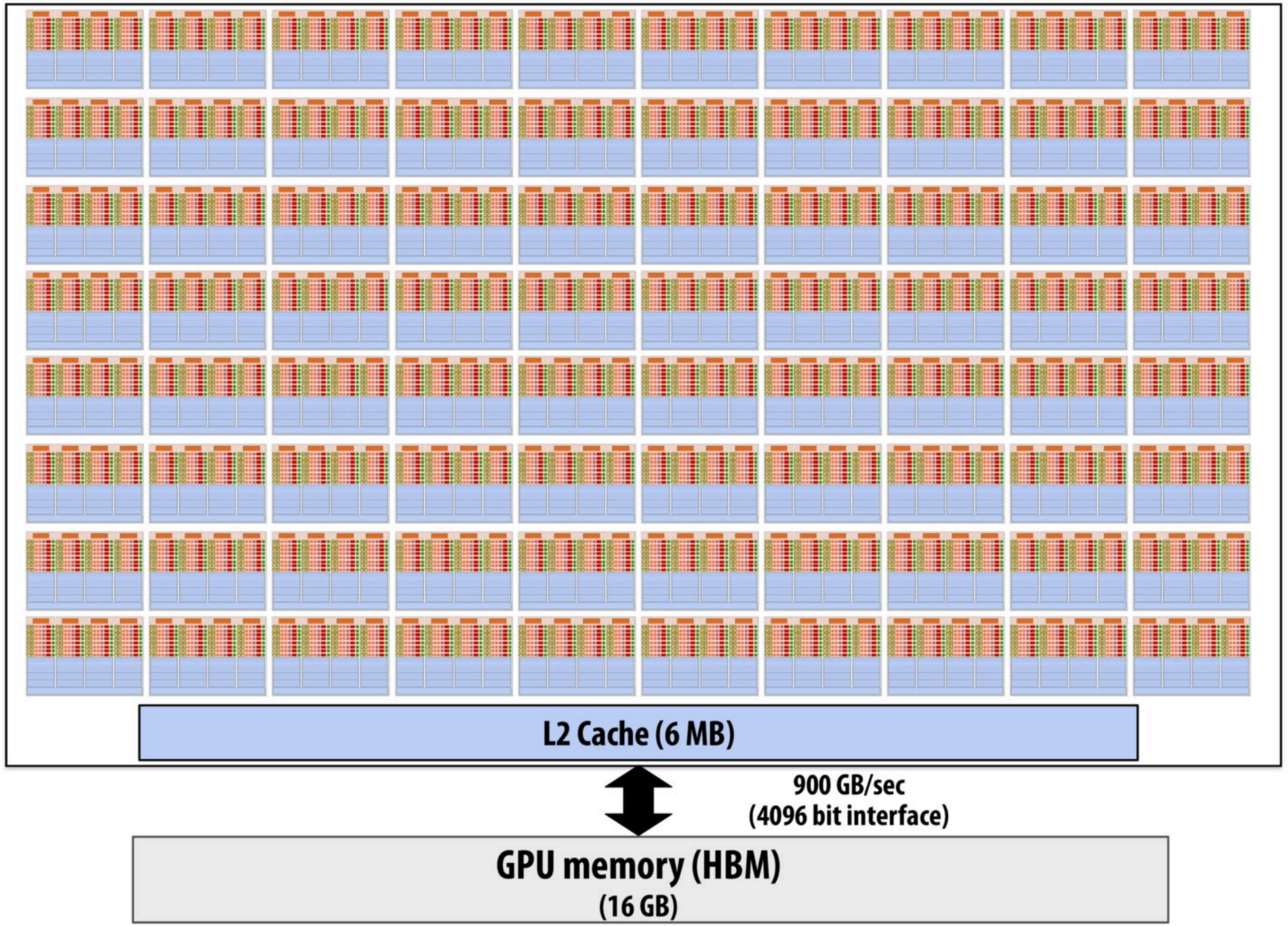
Fast WARP level GEMMs

Iterators for fast block loading/tensor indexing

Tensor reductions

Etc.

Recall: NVIDIA V100 GPU (80 SMs)



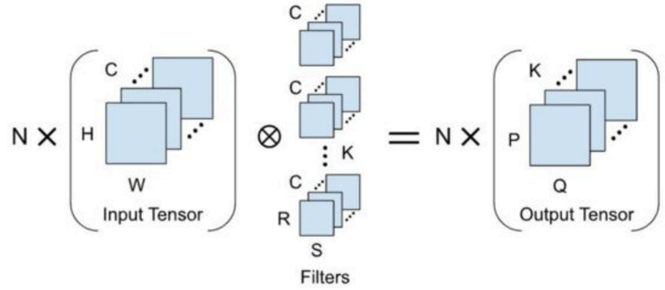
Many processing units and many tensor cores.

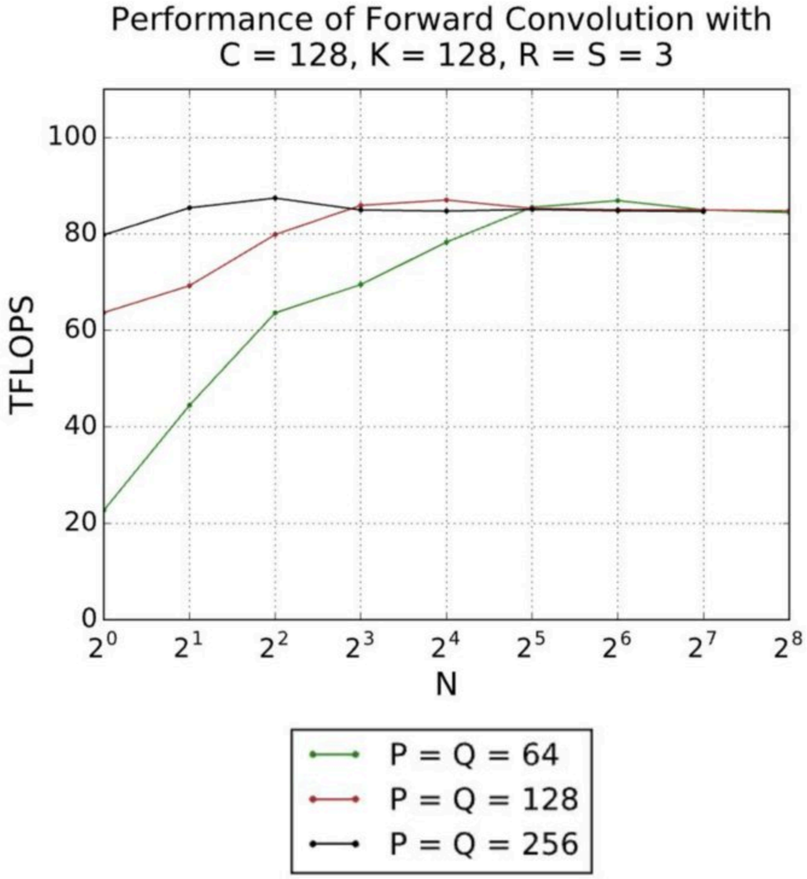
Need "a lot of parallel work" to fill the machine.

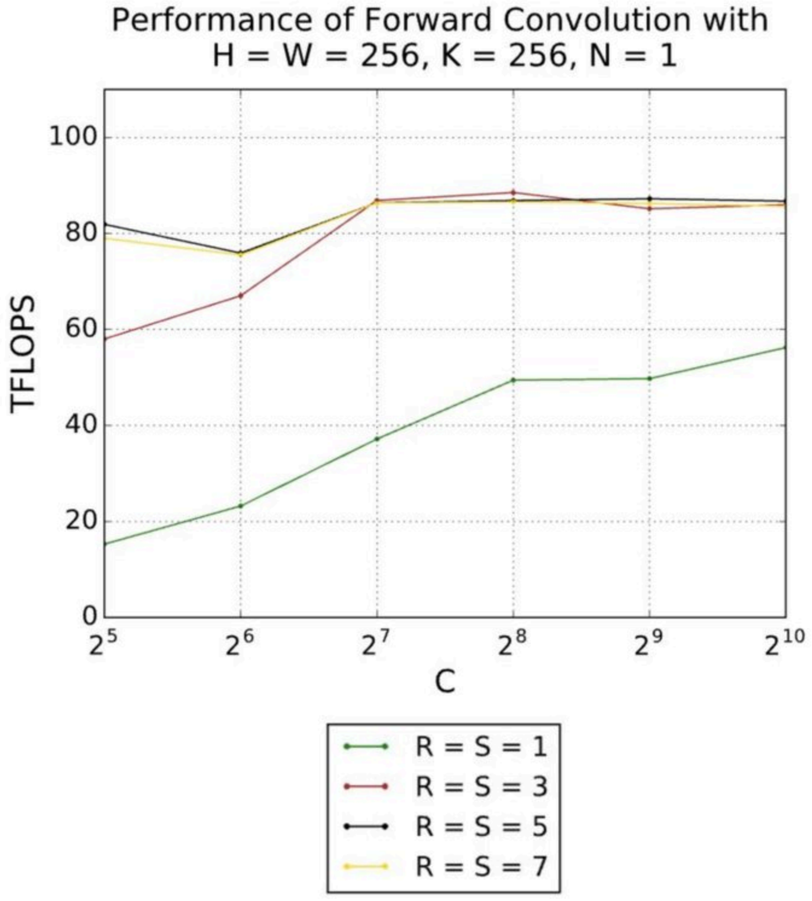
Higher performance with "more work"

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)







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 (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;
         }
```

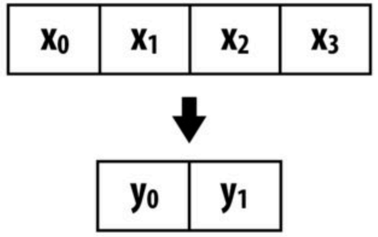
Or you can just directly implement this loop nest directly yourself.

Algorithmic improvements

- **Direct convolution can be implemented efficiently in Fourier domain (convolution → element-wise multiplication)**
  - Overhead: FFT to transform inputs into Fourier domain, inverse FFT to get responses back to spatial domain (NlgN)
  - Inverse transform amortized over all input channels (due to summation over inputs)

- **Direct convolution using work-efficient Winograd convolutions**

1D example: consider producing two outputs of a 3-tap 1D convolution with weights: w_0 w_1 w_2



$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (x_0 - x_1)w_0$$

$$m_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2}$$

$$m_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2}$$

$$m_4 = (x_1 - x_3)w_2$$

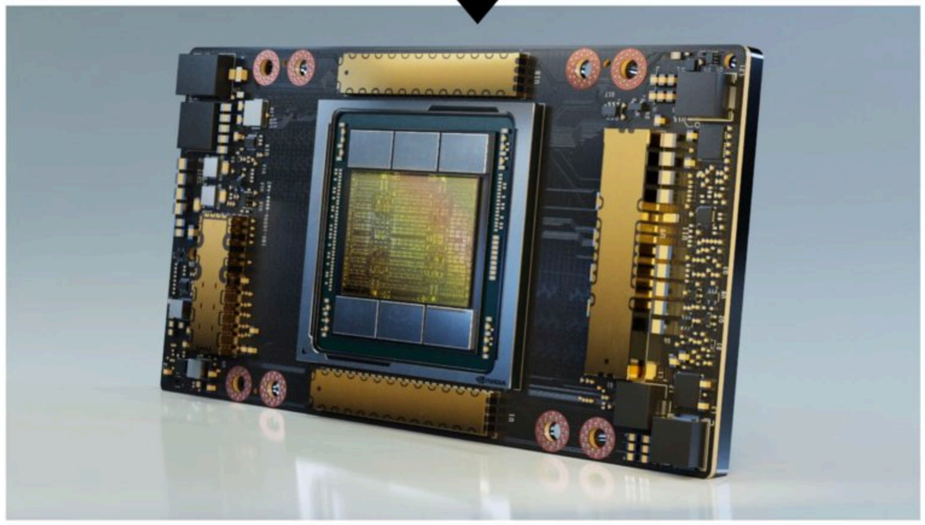
Filter dependent (can be precomputed)

Winograd 1D 3-element filter:
4 multiplies
8 additions
(4 to compute m's + 4 to reduce final result)

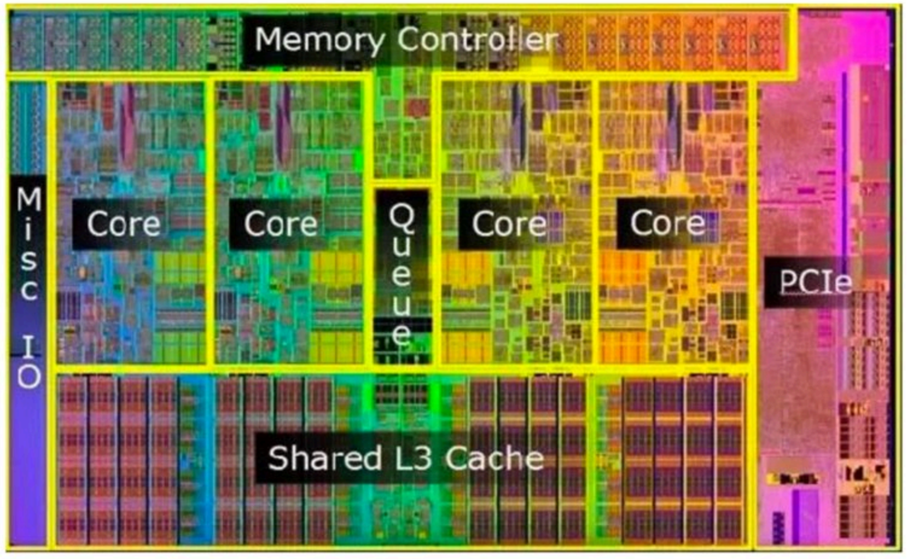
Direct convolution: 6 multiplies, 4 adds
In 2D can notably reduce multiplications
(3x3 filter: 2.25x fewer multiples for 2x2 block of output)

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

NVIDIA cuDNN



Intel® oneAPI Deep Neural Network Library



Libraries offering high-performance implementations of key DNN layers

TensorFlow NN ops

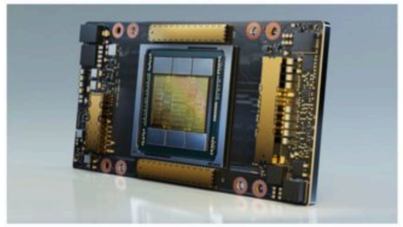
tensorflow::ops::AvgPool	Performs average pooling on the input.
tensorflow::ops::AvgPool3D	Performs 3D average pooling on the input.
tensorflow::ops::AvgPool3DGrad	Computes gradients of average pooling function.
tensorflow::ops::BiasAdd	Adds `bias` to `value`.
tensorflow::ops::BiasAddGrad	The backward operation for "BiasAdd" on the "bias" te
tensorflow::ops::Conv2D	Computes a 2-D convolution given 4-D `input` and `fi`
tensorflow::ops::Conv2DBackpropFilter	Computes the gradients of convolution with respect t
tensorflow::ops::Conv2DBackpropInput	Computes the gradients of convolution with respect t
tensorflow::ops::Conv3D	Computes a 3-D convolution given 5-D `input` and `fi`
tensorflow::ops::Conv3DBackpropFilterV2	Computes the gradients of 3-D convolution with respe
tensorflow::ops::Conv3DBackpropInputV2	Computes the gradients of 3-D convolution with respe
tensorflow::ops::DataFormatDimMap	Returns the dimension index in the destination data f
tensorflow::ops::DataFormatVecPermute	Permute input tensor from `src_format` to `dst_for`
tensorflow::ops::DepthwiseConv2dNative	Computes a 2-D depthwise convolution given 4-D `inp` tensors.
tensorflow::ops::DepthwiseConv2dNativeBackpropFilter	Computes the gradients of depthwise convolution wit
tensorflow::ops::DepthwiseConv2dNativeBackpropInput	Computes the gradients of depthwise convolution wit
tensorflow::ops::Dilation2D	Computes the grayscale dilation of 4-D `input` and 3-
tensorflow::ops::Dilation2DBackpropFilter	Computes the gradient of morphological 2-D dilation filter.
tensorflow::ops::Dilation2DBackpropInput	Computes the gradient of morphological 2-D dilation input.
tensorflow::ops::Elu	Computes exponential linear: `exp(features) - 1` otherwise.
tensorflow::ops::FractionalAvgPool	Performs fractional average pooling on the input.
tensorflow::ops::FractionalMaxPool	Performs fractional max pooling on the input.
tensorflow::ops::FusedBatchNorm	Batch normalization.
tensorflow::ops::FusedBatchNormGrad	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormGradV2	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormGradV3	Gradient for batch normalization.
tensorflow::ops::FusedBatchNormV2	Batch normalization.
tensorflow::ops::FusedBatchNormV3	Batch normalization.
tensorflow::ops::FusedPadConv2D	Performs a padding as a preprocess during a convolution.
tensorflow::ops::FusedResizeAndPadConv2D	Performs a resize and padding as a preprocess during a convolution.
tensorflow::ops::InTopK	Says whether the targets are in the top **K** predictions.
tensorflow::ops::InTopKV2	Says whether the targets are in the top **K** predictions.
tensorflow::ops::L2Loss	L2 Loss.
tensorflow::ops::LRN	Local Response Normalization.
tensorflow::ops::LogSoftmax	Computes log softmax activations.
tensorflow::ops::MaxPool	Performs max pooling on the input.
tensorflow::ops::MaxPool3D	Performs 3D max pooling on the input.
tensorflow::ops::MaxPool3DGrad	Computes gradients of 3D max pooling function.
tensorflow::ops::MaxPool3DGradGrad	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGrad	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGradV2	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradGradWithArgmax	Computes second-order gradients of the maxpooling function.
tensorflow::ops::MaxPoolGradV2	Computes gradients of the maxpooling function.
tensorflow::ops::MaxPoolV2	Performs max pooling on the input.
tensorflow::ops::MaxPoolWithArgmax	Performs max pooling on the input and outputs both max values and indices.
tensorflow::ops::NthElement	Finds values of the n-th order statistic for the last dimension.
tensorflow::ops::QuantizedAvgPool	Produces the average pool of the input tensor for quantized types.
tensorflow::ops::QuantizedBatchNormWithGlobalNormalization	Quantized Batch normalization.
tensorflow::ops::QuantizedBiasAdd	Adds Tensor 'bias' to Tensor 'input' for Quantized types.
tensorflow::ops::QuantizedConv2D	Computes a 2D convolution given quantized 4D input and filter tensors.
tensorflow::ops::QuantizedMaxPool	Produces the max pool of the input tensor for quantized types.

Libraries offering high-performance implementations of key DNN layers

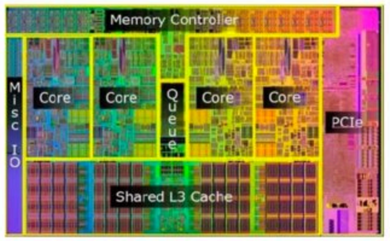
TensorFlow **NN ops**

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

NVIDIA cuDNN



Intel® oneAPI Deep Neural Network Library



Example: CUDNN convolution

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

Possible algorithms:

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

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

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

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

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

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

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

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

Memory traffic between operations

- Consider this sequence:

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

Fusing operations with conv layer

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

// assumes convolution stride is 1
for (int img=0; img<IMAGE_BATCH_SIZE; img++)
   for (int j=0; j<INPUT_HEIGHT; j++)
      for (int i=0; i<INPUT_WIDTH; i++)
         for (int f=0; f<LAYER_NUM_FILTERS; f++) {
            float tmp = 0.0f;
            for (int kk=0; kk<INPUT_DEPTH; kk++)          // sum over filter responses of input channels
               for (int jj=0; jj<LAYER_FILTER_Y; jj++)    // spatial convolution (Y)
                  for (int ii=0; ii<LAYER_FILTER_X; ii+)  // spatial convolution (X)
                     tmp += layer_weights[f][jj][ii][kk] * input[img][j+jj][i+ii][kk];
           output[img][j][i][f] = tmp*scale + bias;
         }
```

Exercise to class:

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

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

A good idea: fusion trick for computing "attention"

Transformer architecture for sequence models

Sequence of tokens in, sequence of tokens out

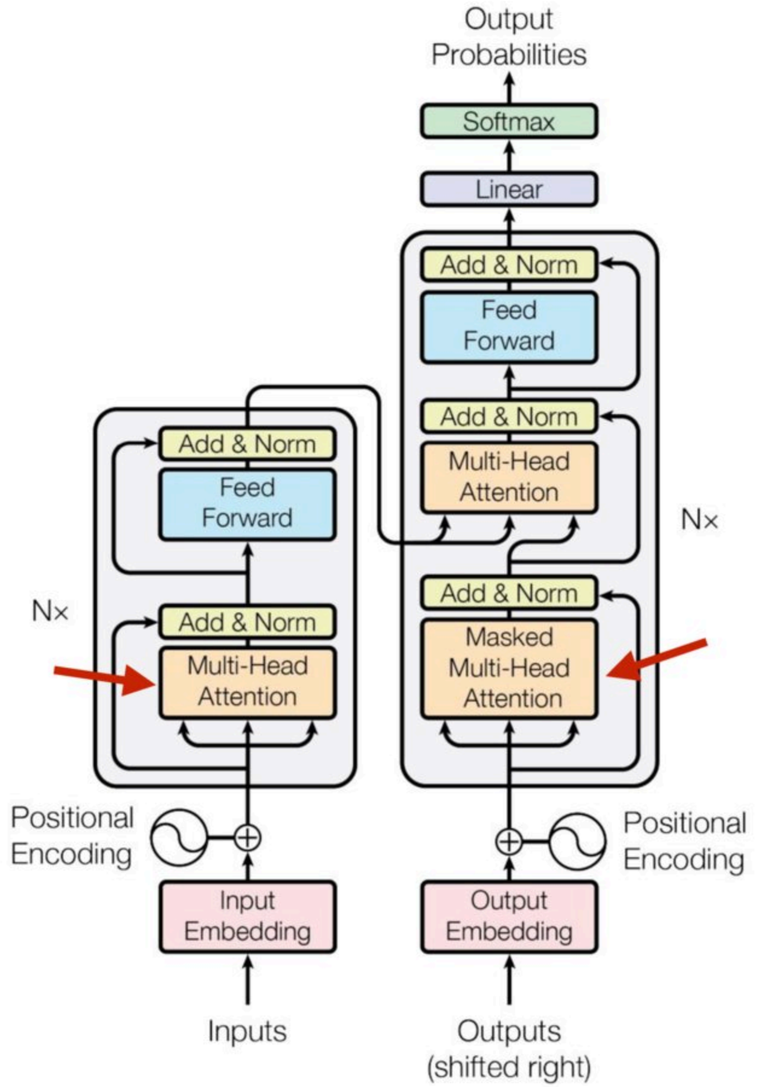
Example: next word prediction

Let's talk about optimizing the computation of attention in a modern DNN. Let's start with

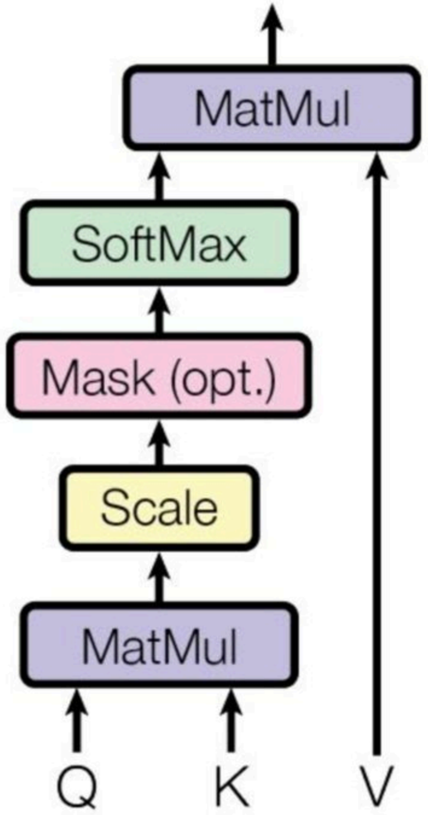
a DNN, which has

a simple model, say a

some simple examples of a conv



Attention module



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

Let $\mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}$

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

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

For a row x:

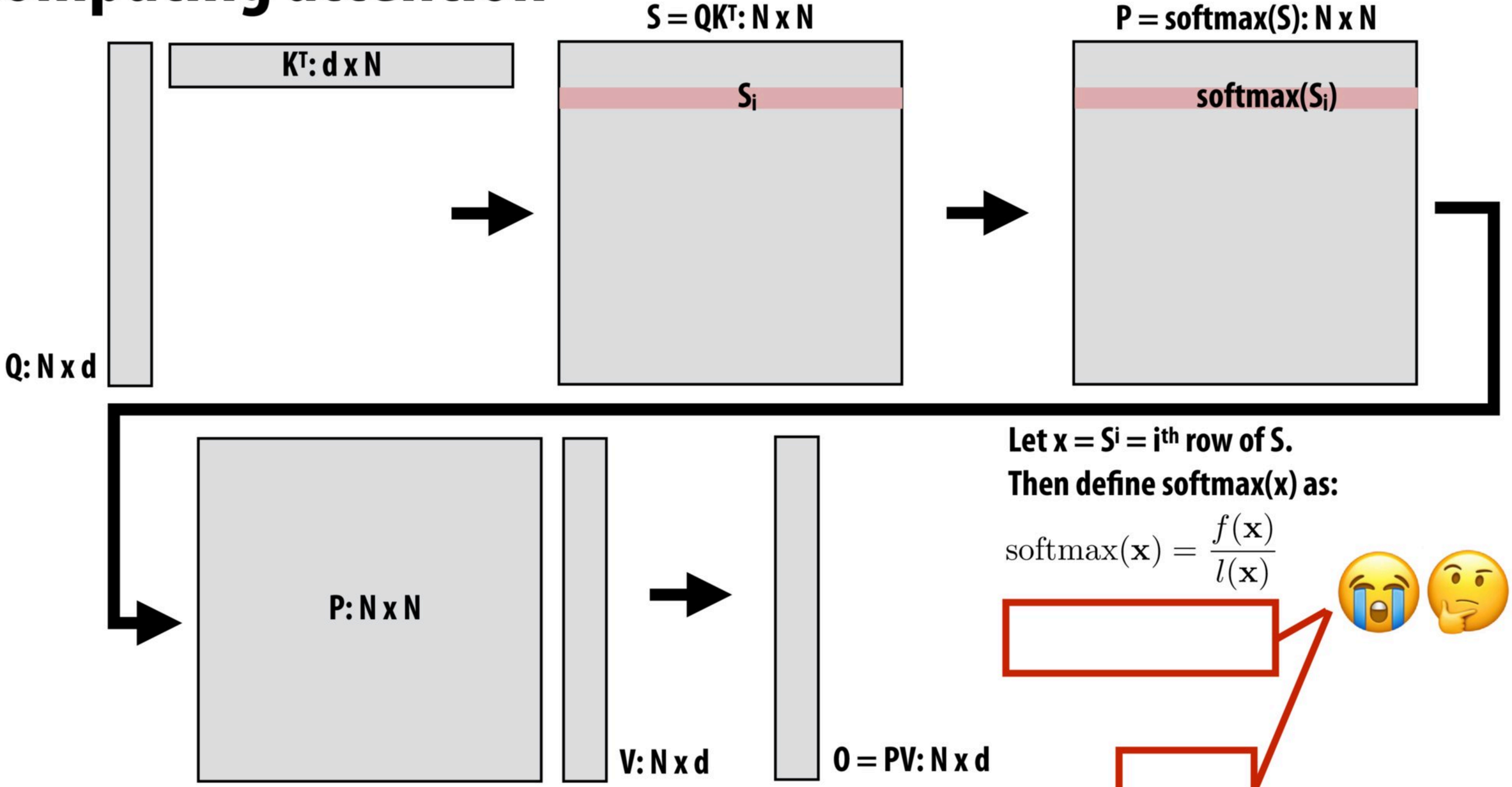
$$\mathrm{softmax}(\mathbf{x}) = \frac{f(\mathbf{x})}{l(\mathbf{x})}$$

Notes:

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

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

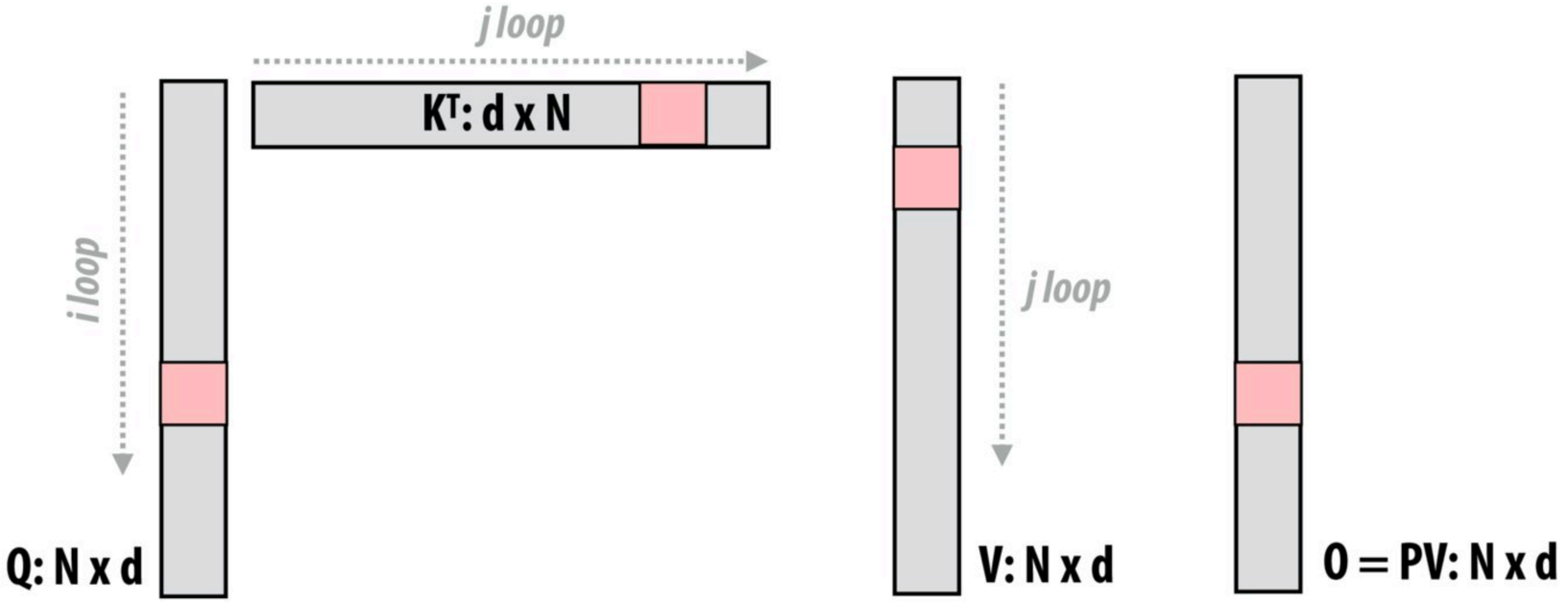
Computing attention



Let's look into softmax more closely…

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

Fused attention



Save memory footprint:
Never materialize N^2 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)

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)

Fusion in modern DNN frameworks

Old style: library writers hardcoded a few "fused" ops

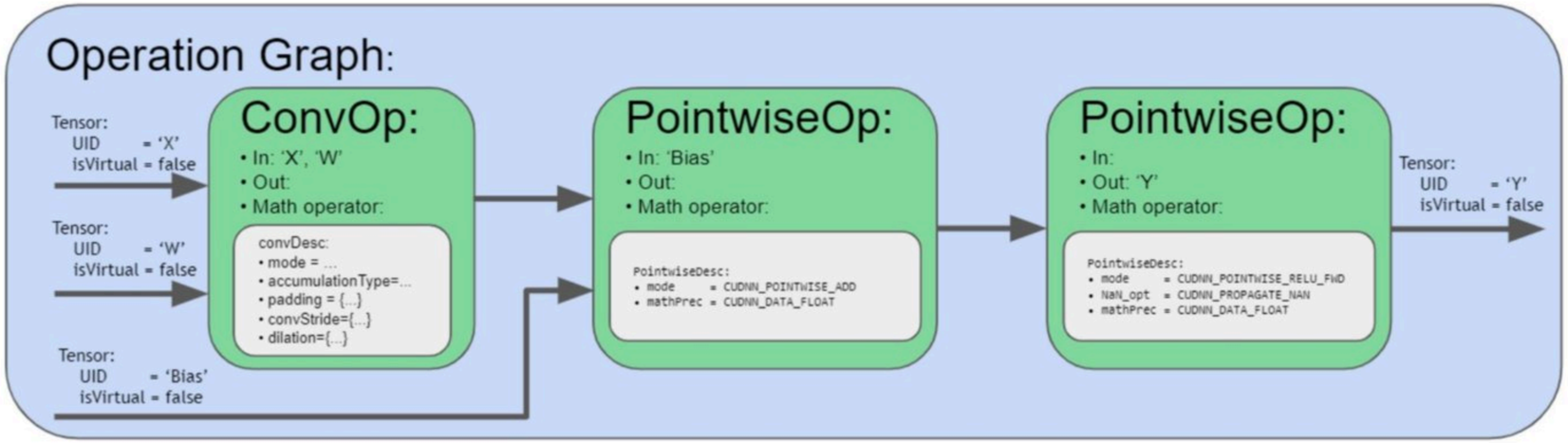
```
cudnnStatus_t cudnnConvolutionBiasActivationForward(
    cudnnHandle_t                       handle,
    const void                         *alpha1,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *x,
    const cudnnFilterDescriptor_t       wDesc,
    const void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

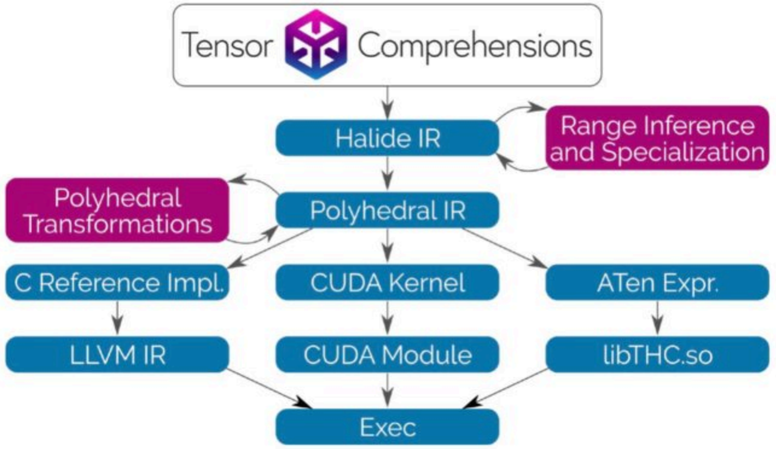


Note for operation fusion use cases, there are two different mechanisms in cuDNN to support them. First, there are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If 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)

Note: this is Halide “compute at”

Many compiler-based efforts to automatically schedule key DNN operations



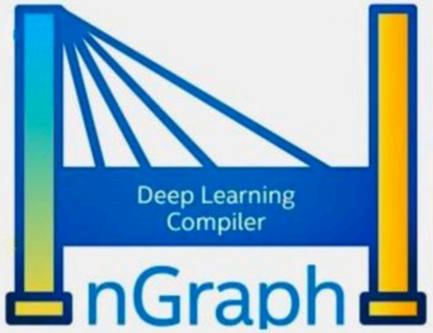
JAX: Autograd and XLA

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 the productivity-focused deep learning frameworks, and the performance- and efficiency-focused hardware backends. TVM works with deep learning frameworks to provide end to end compilation to different backends. Checkout the tvm stack homepage for more information.



NVIDIA TensorRT

Programmable Inference Accelerator

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
- In the extreme case: 1-bit ;-)

XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

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

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

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

Optimization techniques

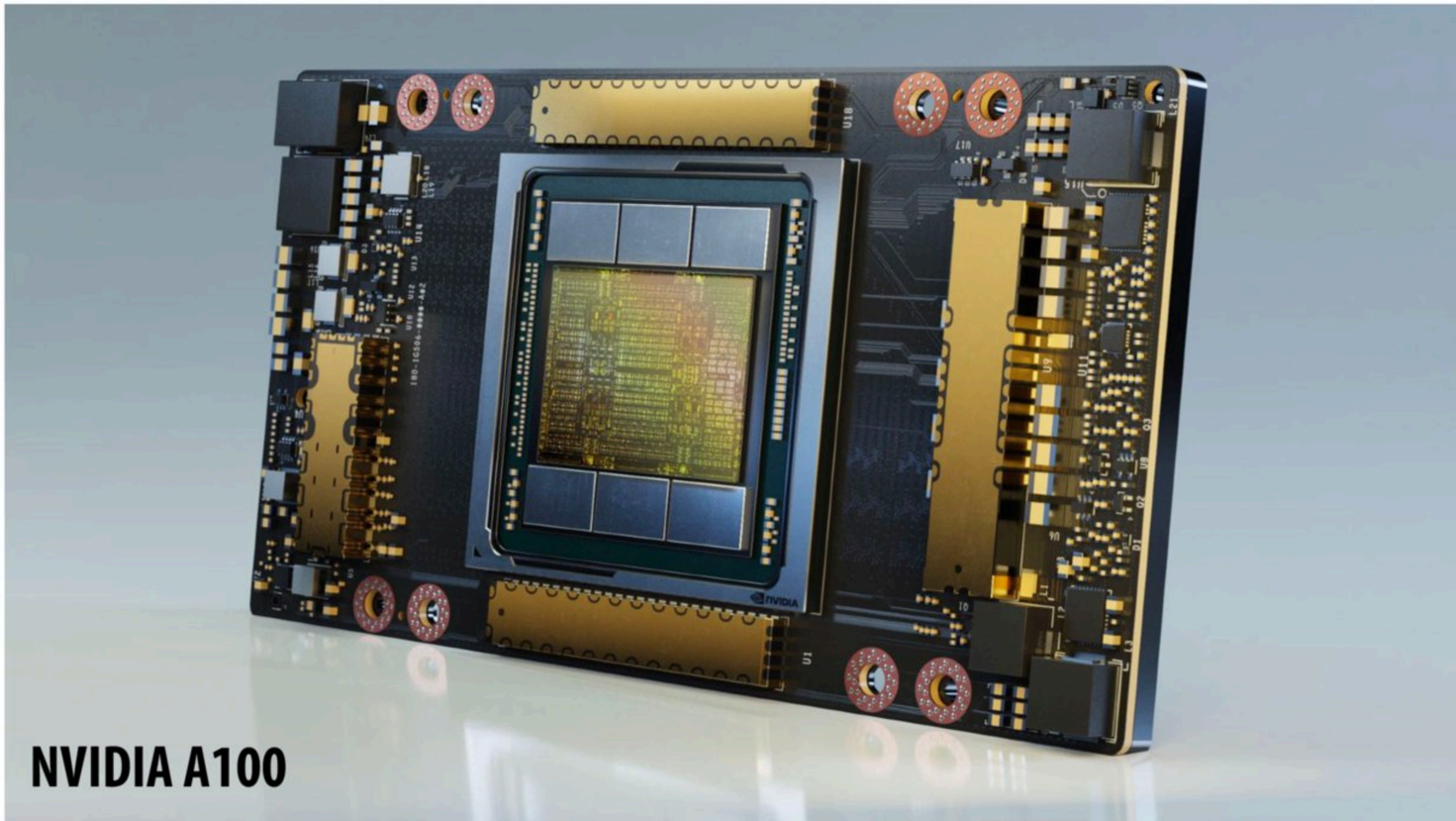
- Better algorithms: manually designing better ML models
  - Common parameters: depth of network, width of filters, number of filters per layer, convolutional stride, etc.
  - Common to perform automatic search for efficient topologies
- Software optimization: Good scheduling of performance-critical 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
  - Automatic sparsification/pruning (not discussed today)

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

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

Deep neural networks on GPUs

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

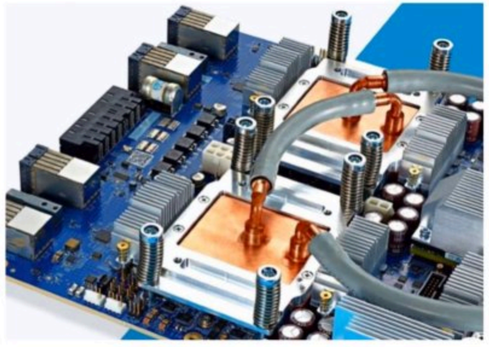


NVIDIA A100

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

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

Next time: the benefits of hardware specialization.
(For DNN evaluation, but also for many other domains!)



Google TPU3

Ampere GPU with Tensor Cores

Apple Neural Engine

Special instruction support