Lecture 18:

Efficiently Evaluating DNNs

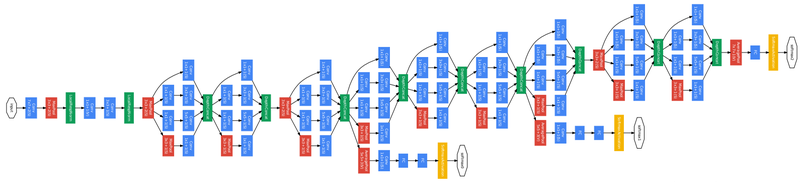
Parallel Computing
Stanford CS149, Fall 2022

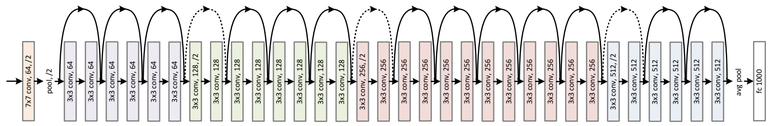
Today

- We will discuss the workload of evaluating deep neural networks (performing "inference")
  - This lecture will be heavily biased towards concerns of DNNs that process images (to be honest, because that is what your instructor knows best)
  - But, image processing is not the application driving the majority of DNN evaluation in the world right now (its text processing, speech, ads, etc.)

Efficiency challenge

Many DNN topologies
(Many variants on common backbones)





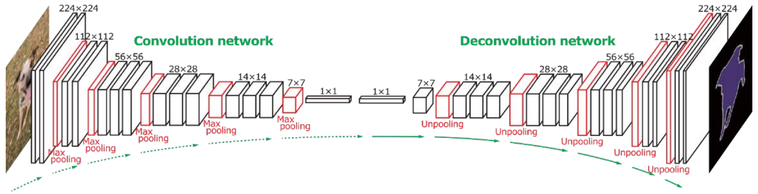


Table 1. MobileNet Body Architecture

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	$1 \times 1 \times 1000$

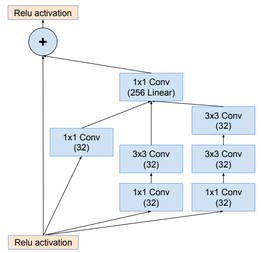
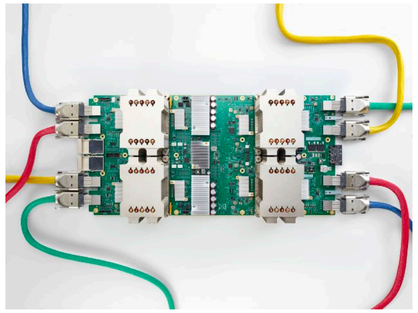
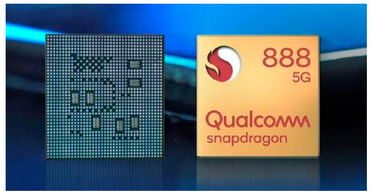


Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

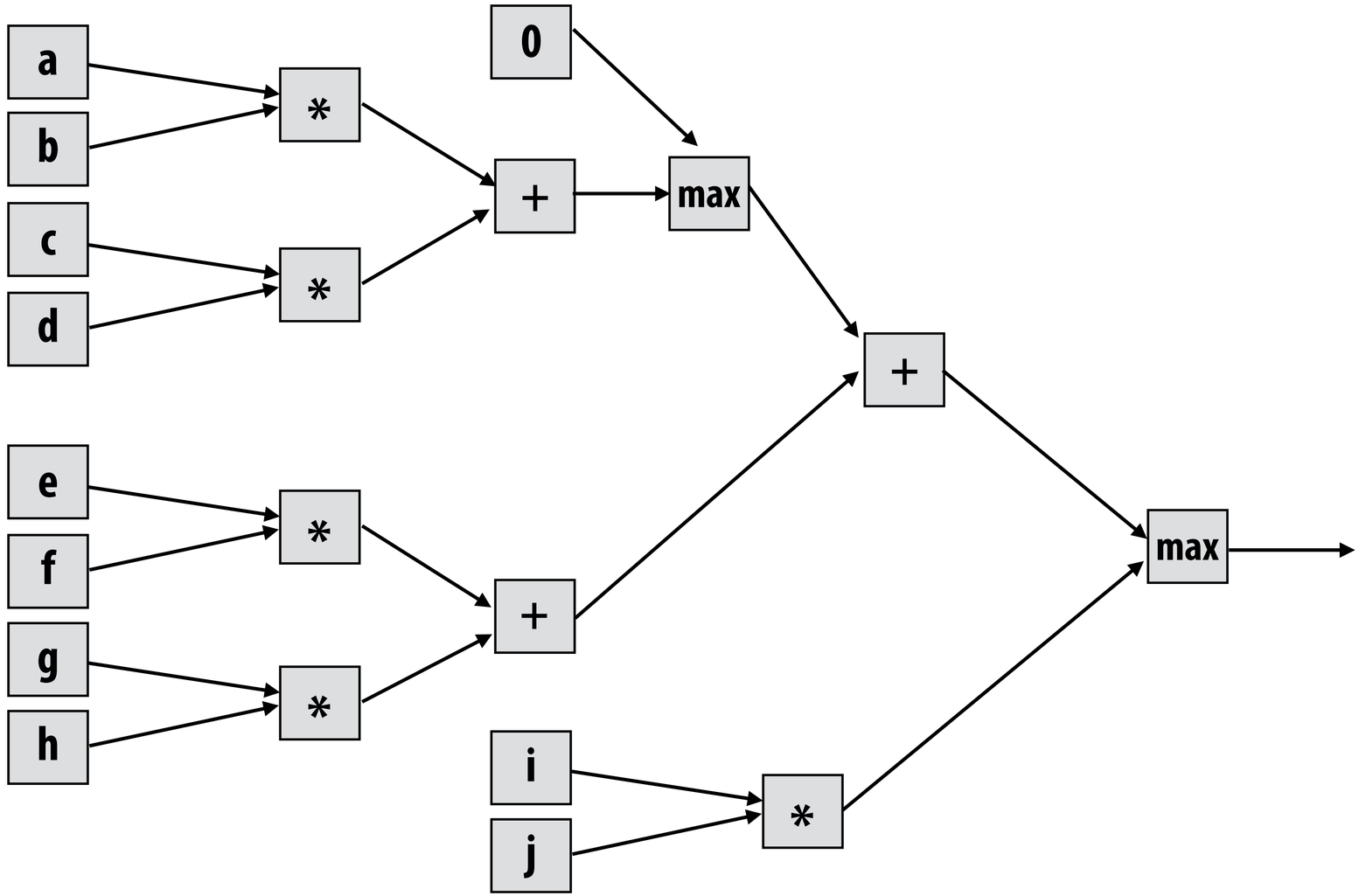
Many Target Devices





Mini-intro/review:
Convolutional Neural Networks

Consider the following expression

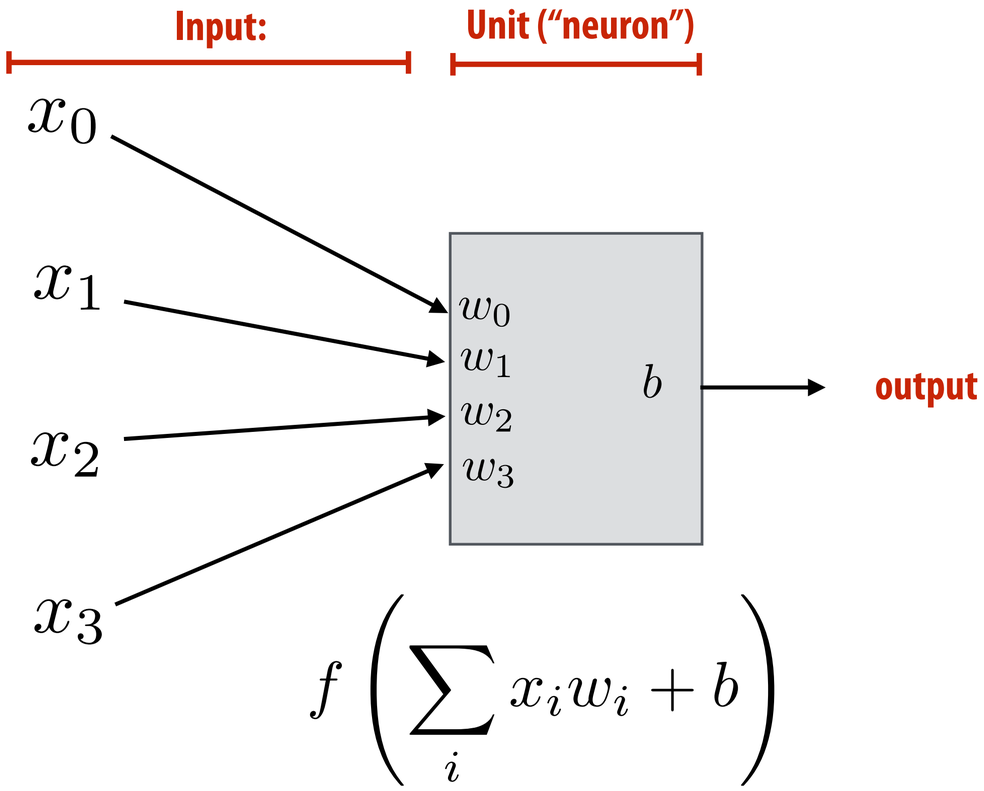



```
max( max(0, (a*b) + (c*d)) + (e*f) + (g*h), i*j)
```

What is a deep neural network?

A basic unit:

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



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

Example: rectified linear unit (ReLU)

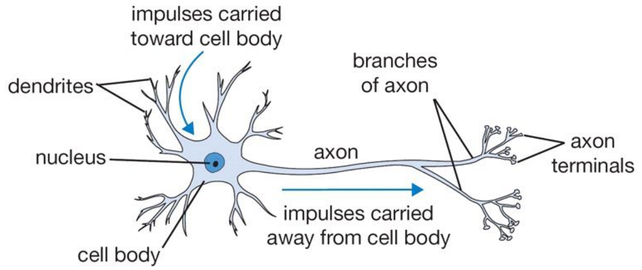
$f(x) = max(0, x)$

Basic computational interpretation:

It is just a circuit!

Biological inspiration:

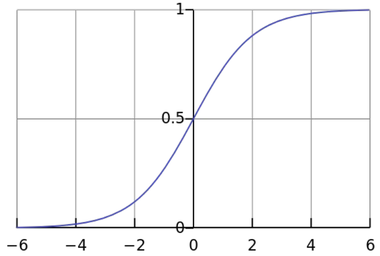
unit output corresponds loosely to activation of neuron



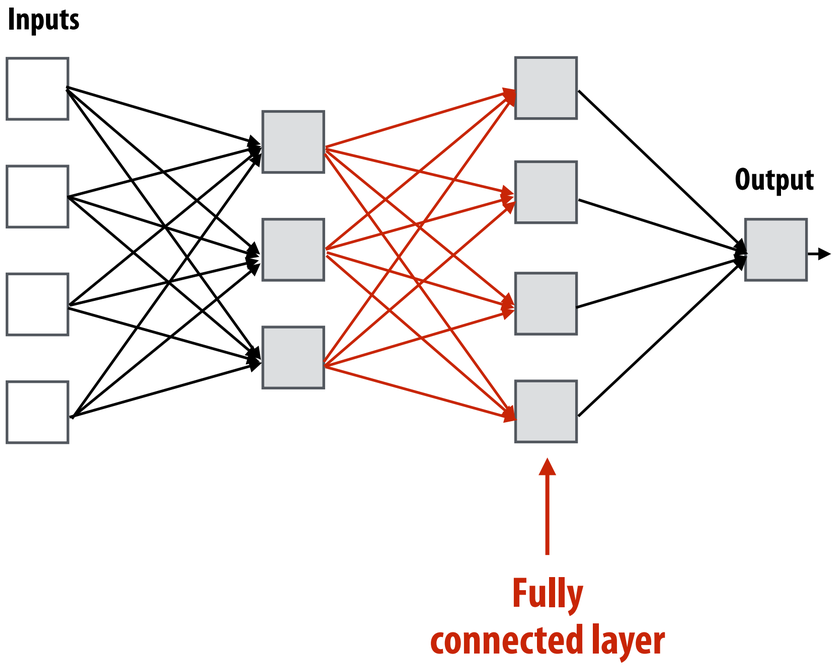
Machine learning interpretation:

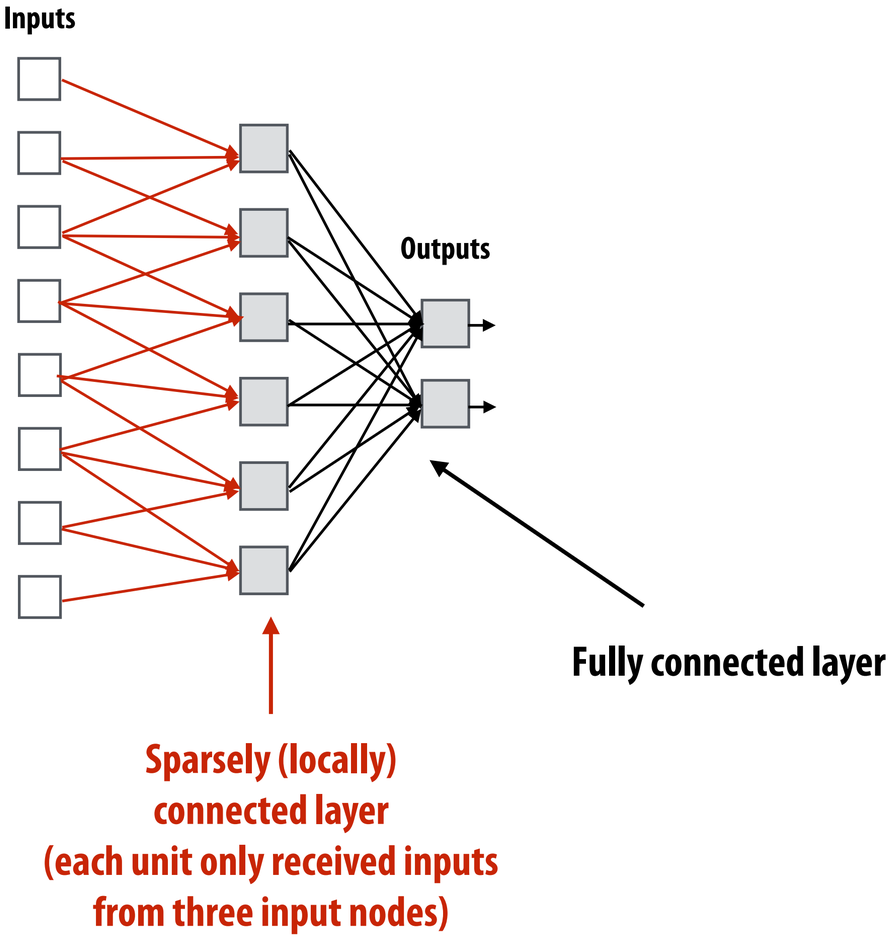
binary classifier: interpret output as the probability of one class

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

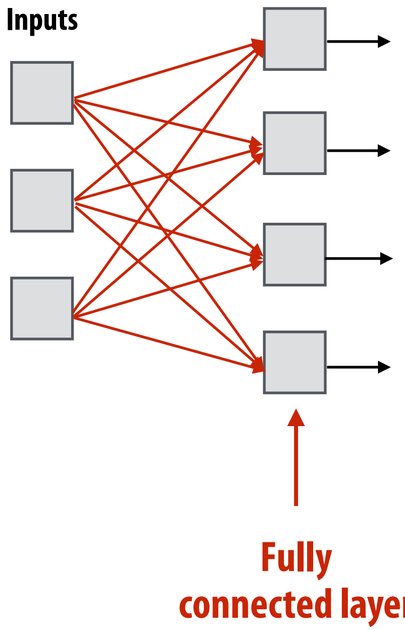


Deep neural network: topology





Fully connected layer as matrix-vector product

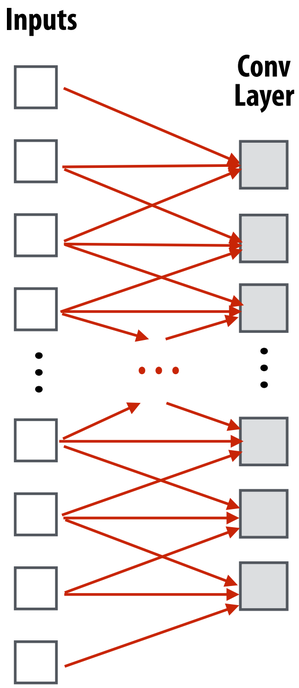


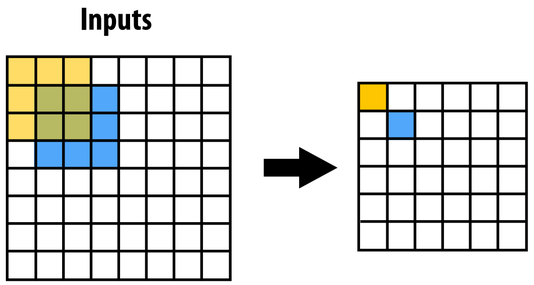
$$f\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{22} & w_{21} & w_{22} \\ w_{32} & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

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

Recall image convolution (3x3 conv)

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


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

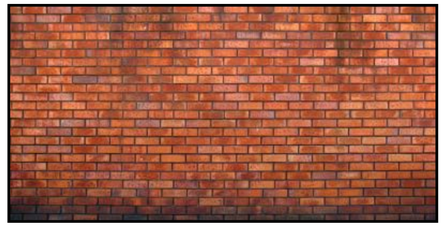





```
for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
    output[j*WIDTH + i] = tmp;
  }
}
```

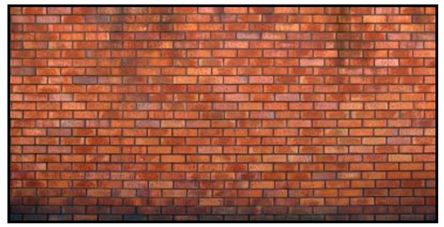
Convolutional layer: locally connected AND all units in layer share the same parameters (same weights + same bias):
(note: network illustration above only shows links for a 1D conv: a.k.a. one iteration of `ii` loop)

Gradient detection filters



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

Responds to horizontal gradients



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

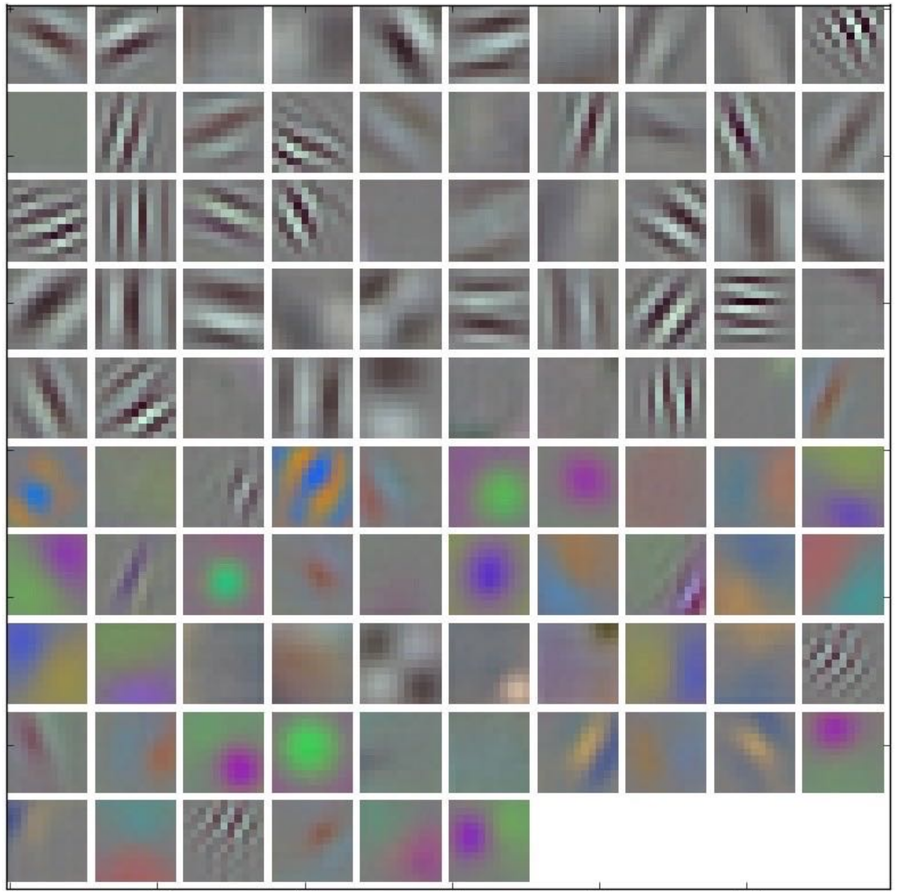
Responds to vertical gradients

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

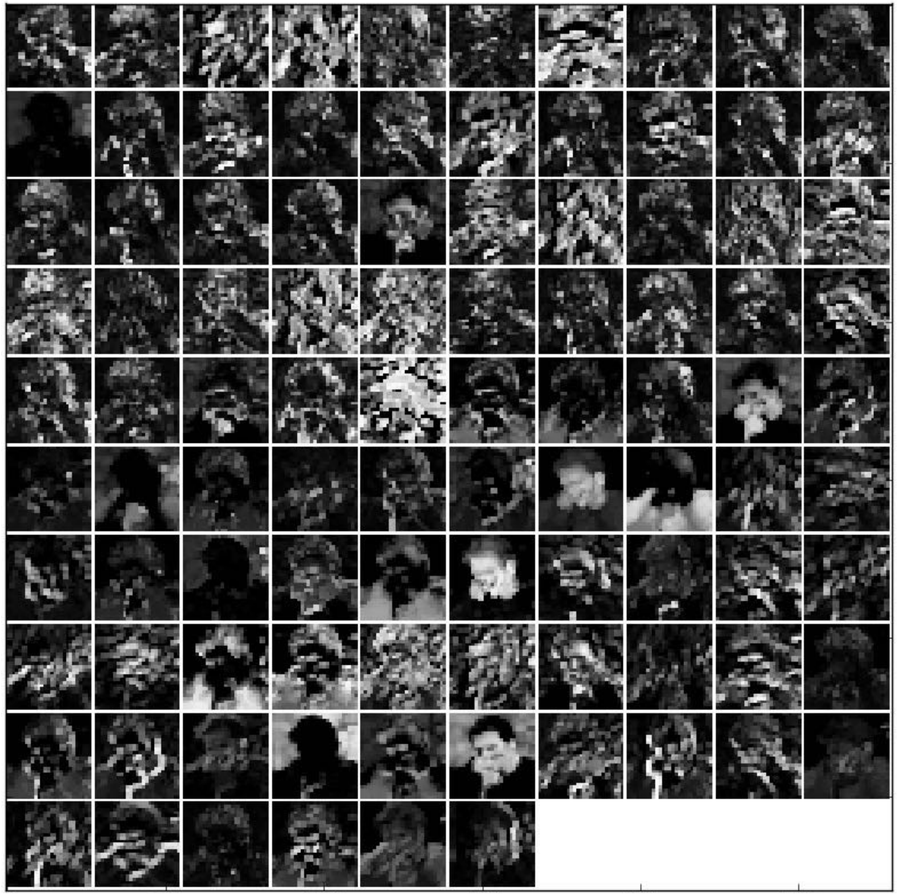
Applying many filters to an image at once

Input RGB image (W x H x 3)

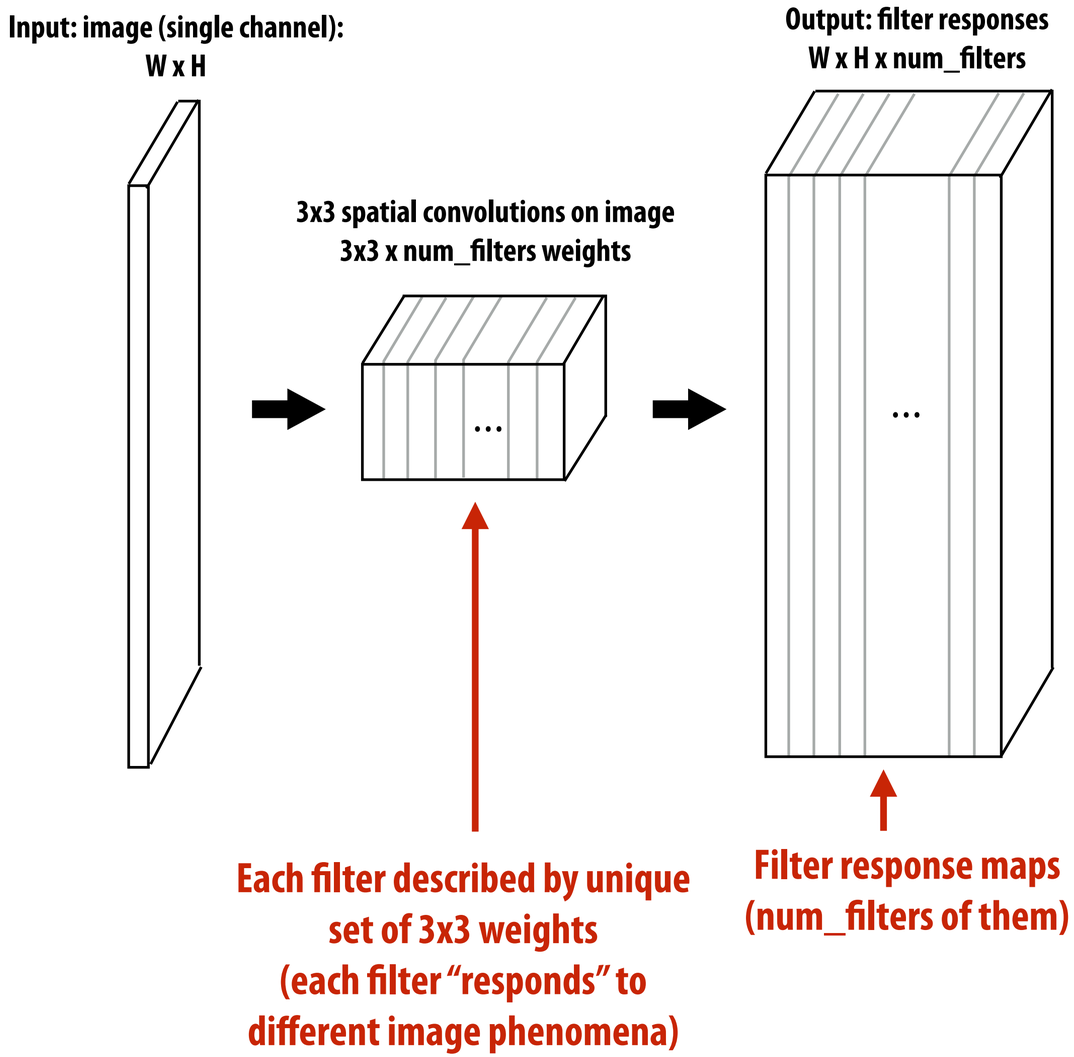
96 11x11x3 filters
(3D because they operate on RGB)



96 responses (normalized)



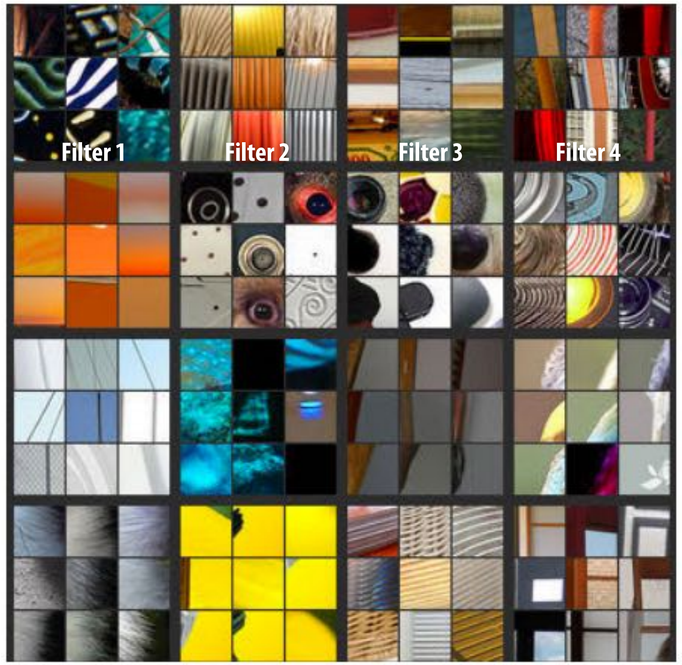
Applying many filters to an image at once



Going deeper

Layer 1

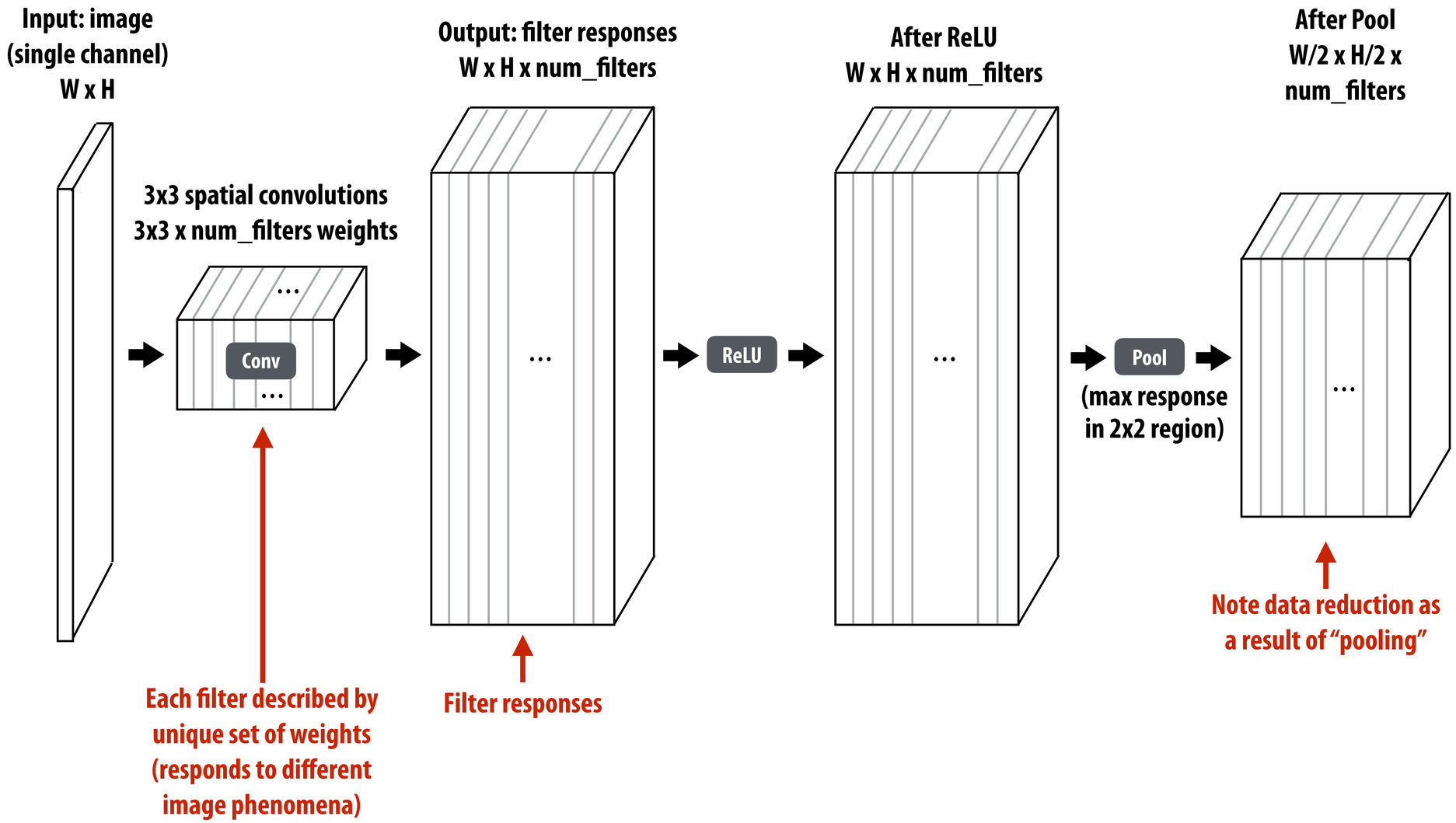
Layer 2



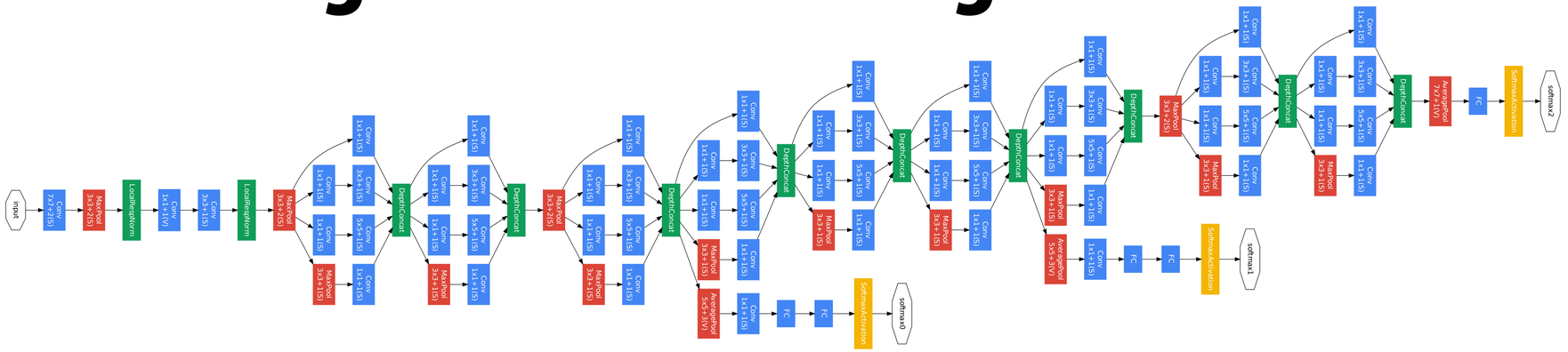
Layer 3

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

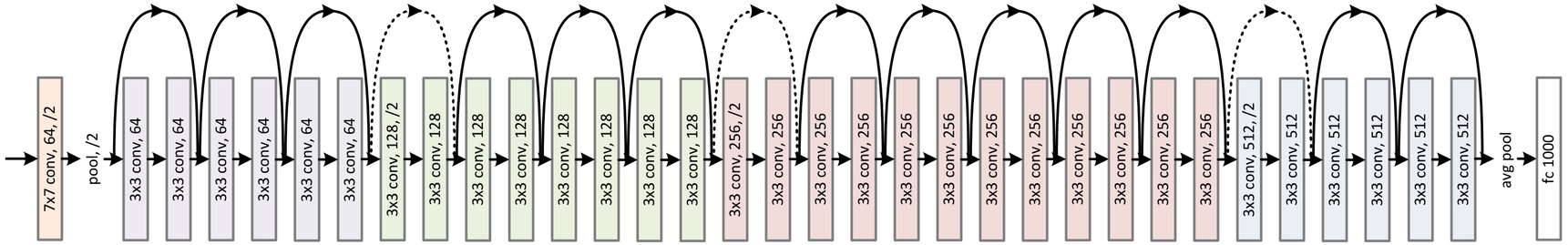
Adding additional layers



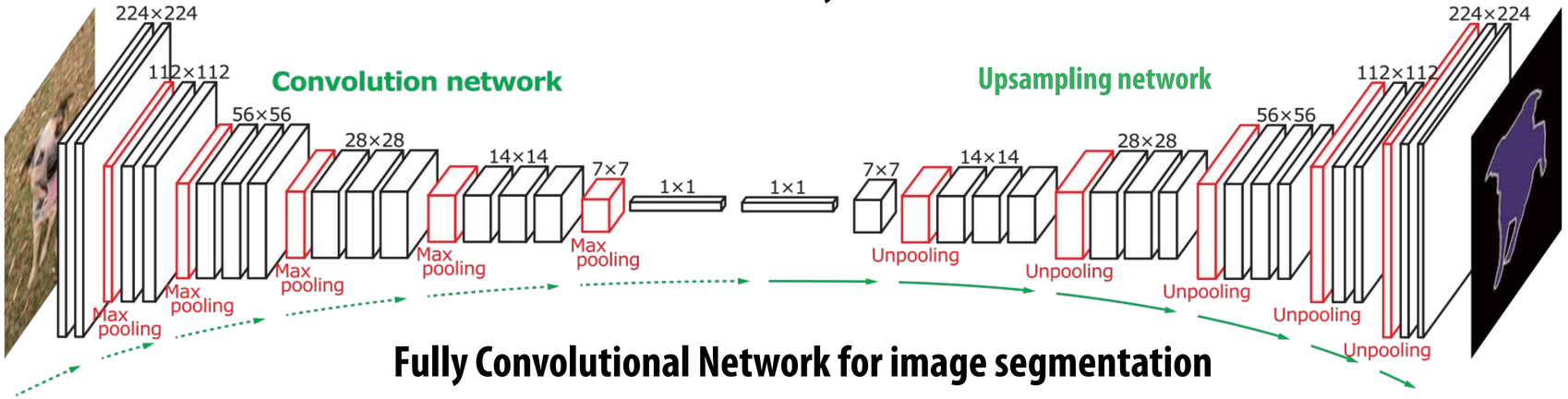
More recent image understanding networks



Inception (GoogleLeNet)



ResNet (34 layer version)

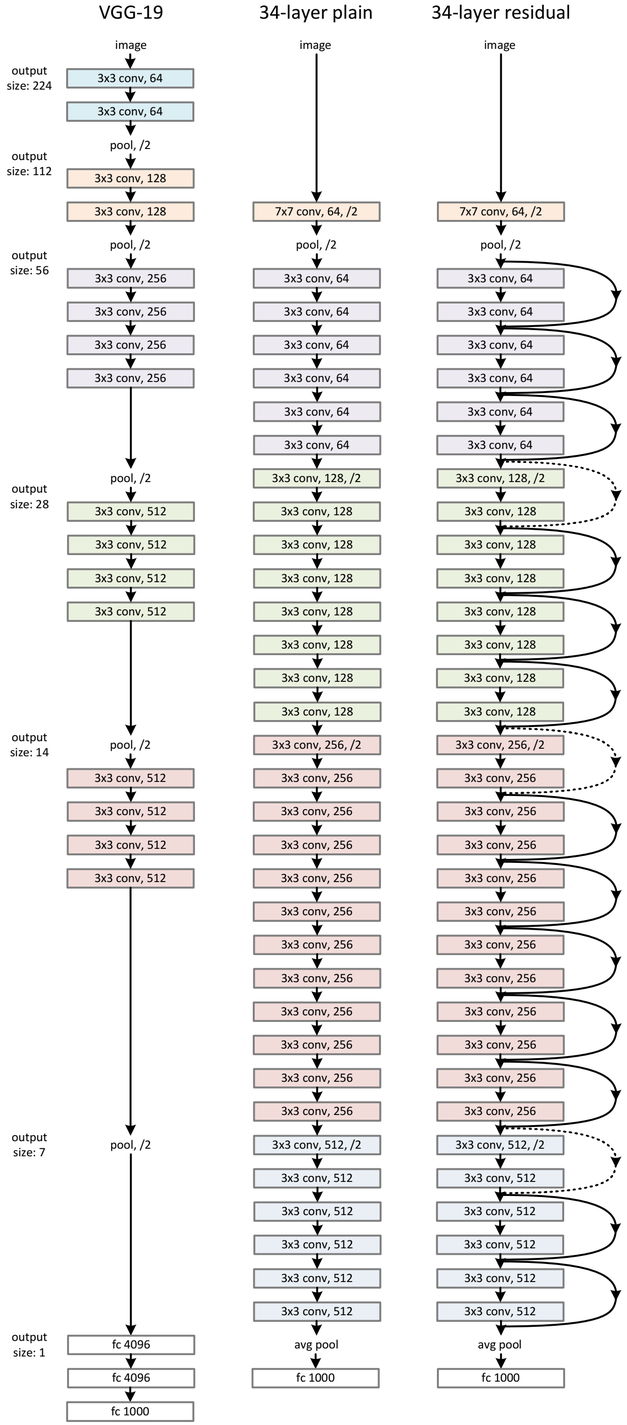


Fully Convolutional Network for image segmentation

Efficiently implementing convolution layers

Approach 1:
Algorithmic innovation: more efficient topologies

ResNet



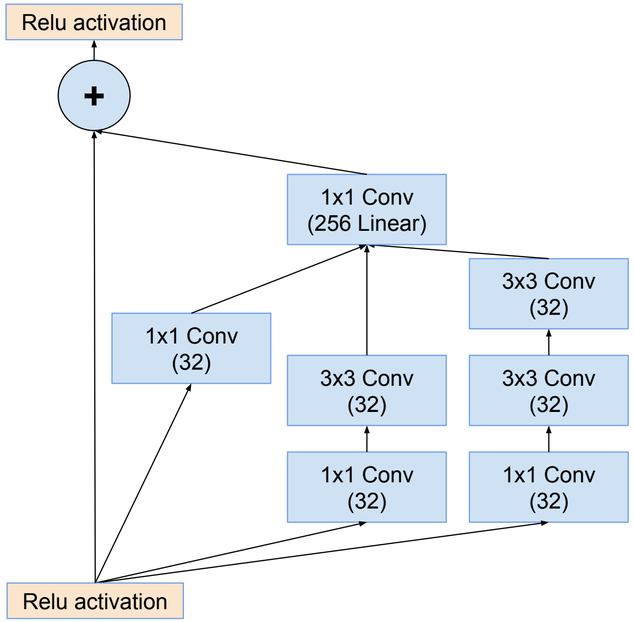


Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

MobileNet

[Howard et al. 2017]

Factor NUM_FILTERS 3x3xNUM_CHANNELS convolutions into:

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

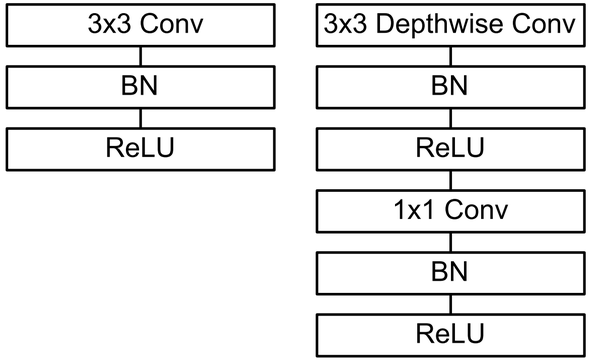


Table 1. MobileNet Body Architecture

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

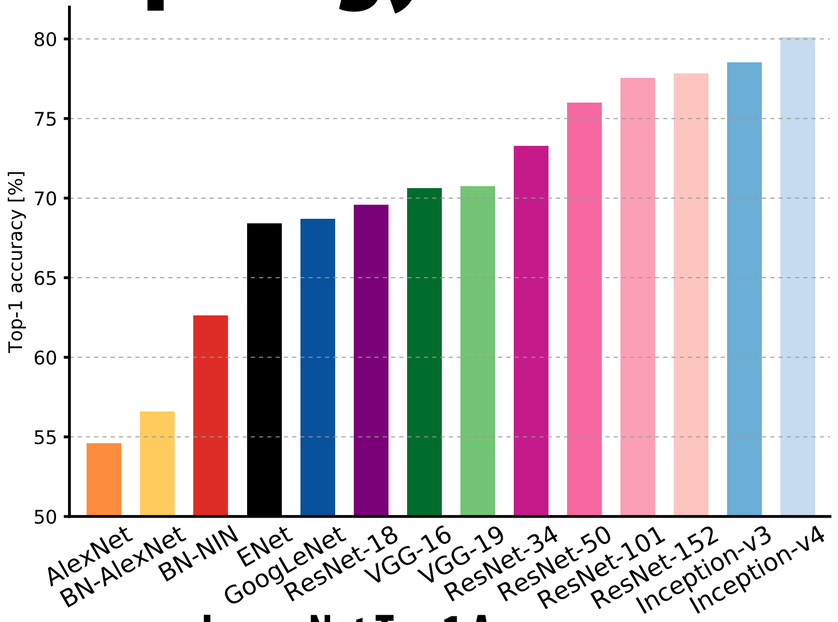
Image classification (ImageNet)
Comparison to Common DNNs

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

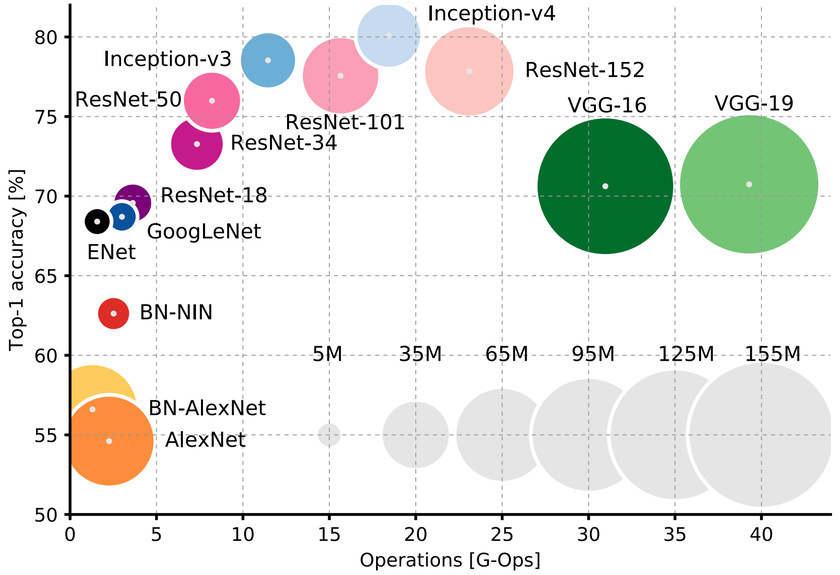
Image classification (ImageNet)
Comparison to Other Compressed DNNs

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

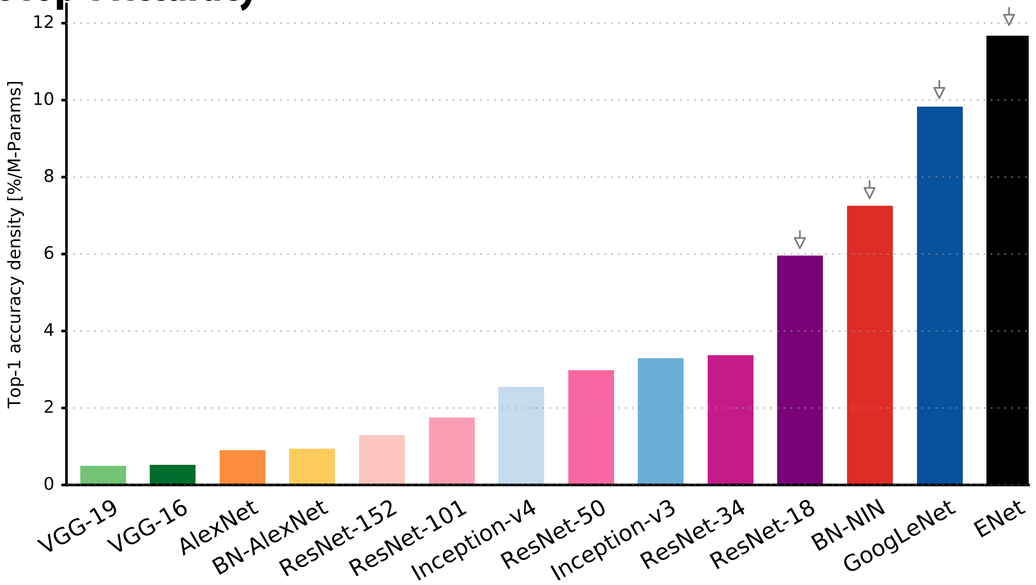
Effect of topology innovation



ImageNet Top 1 Accuracy



Flops cost (area of circle is # params)



Accuracy (points) per flop

Figure credit: Canziani et al 2017

Improving accuracy/cost (image classification)

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

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

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

Approach 2:
Code optimization: implement layers efficiently on modern hardware using many of the techniques discussed in CS149

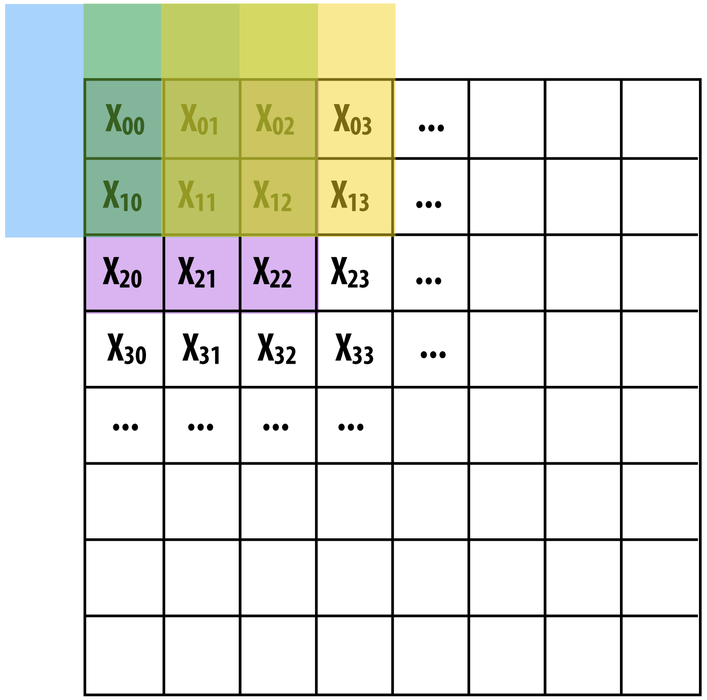
Direct implementation of conv layer (batched)

```
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)

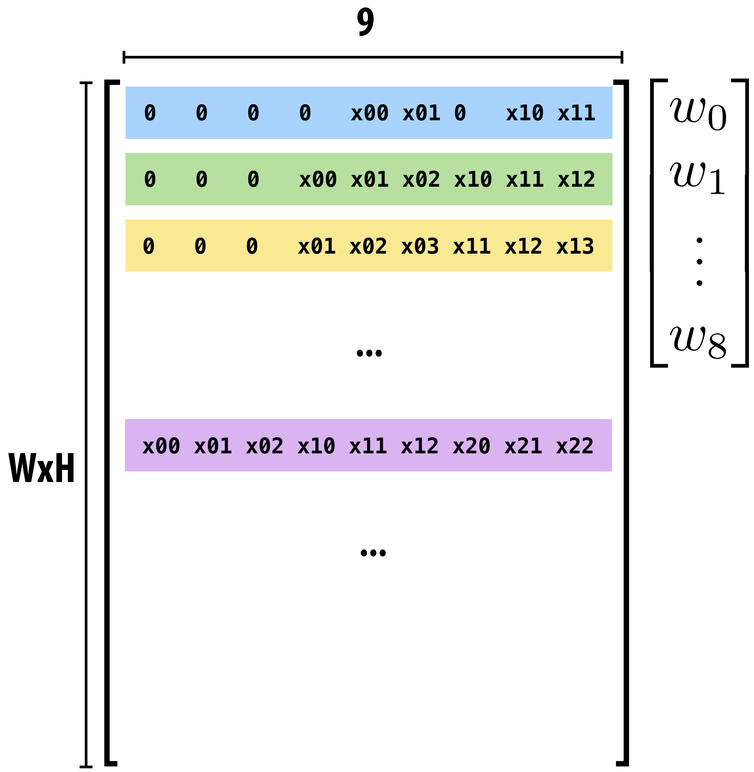
3x3 convolution as matrix-vector product ("explicit gemm")

Construct matrix from elements of input image



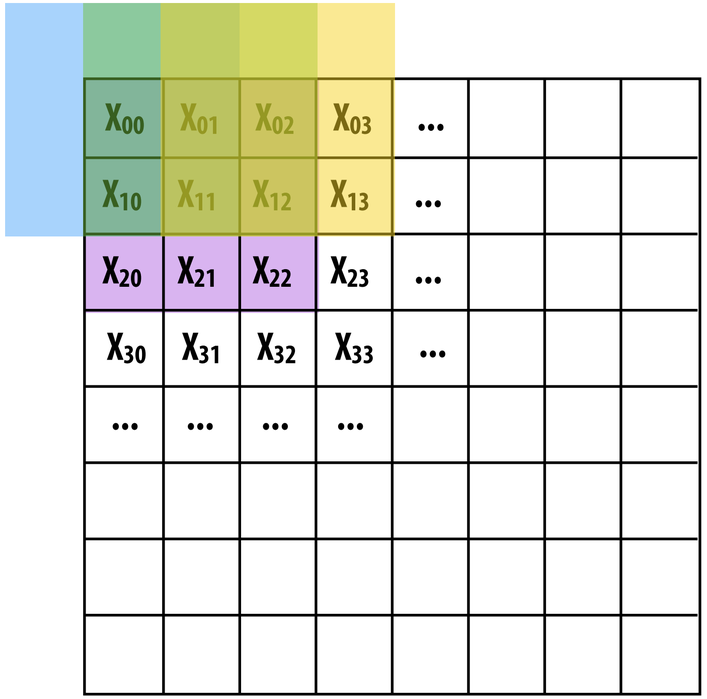
O(N) storage overhead for filter with N elements

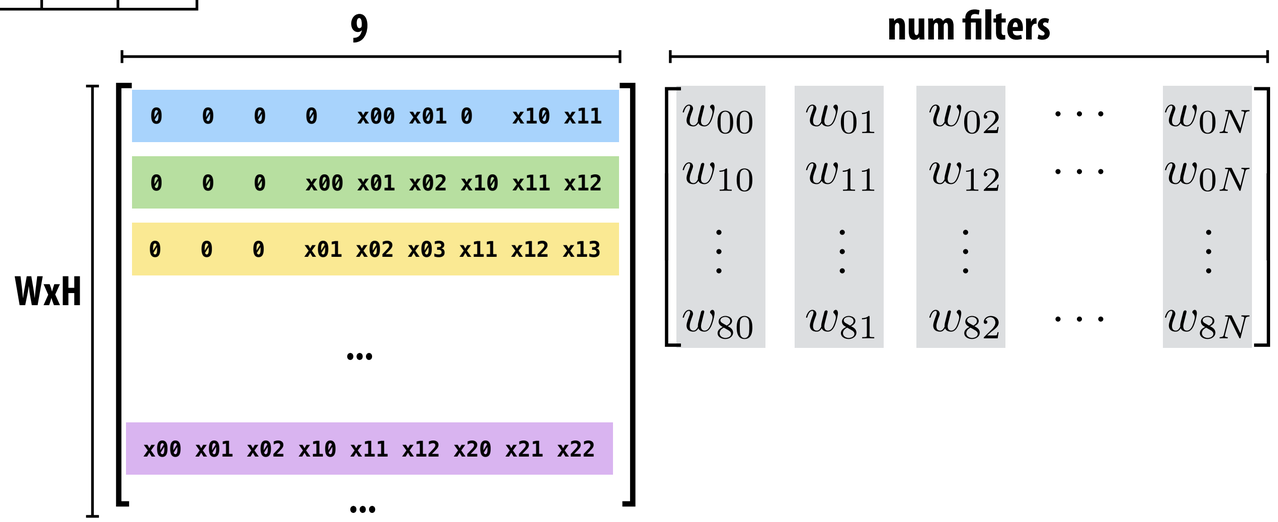
Must construct input data matrix



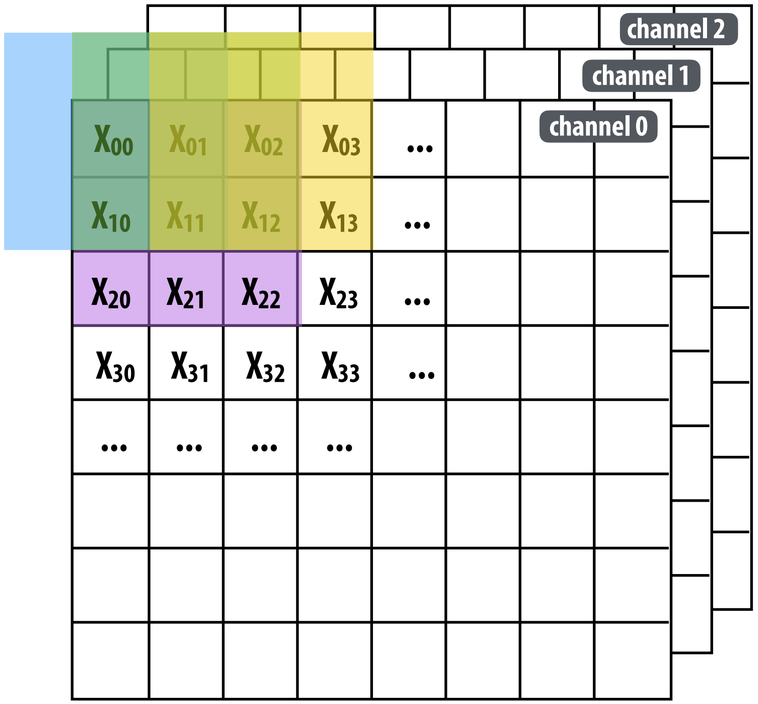
Note: 0-pad matrix

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



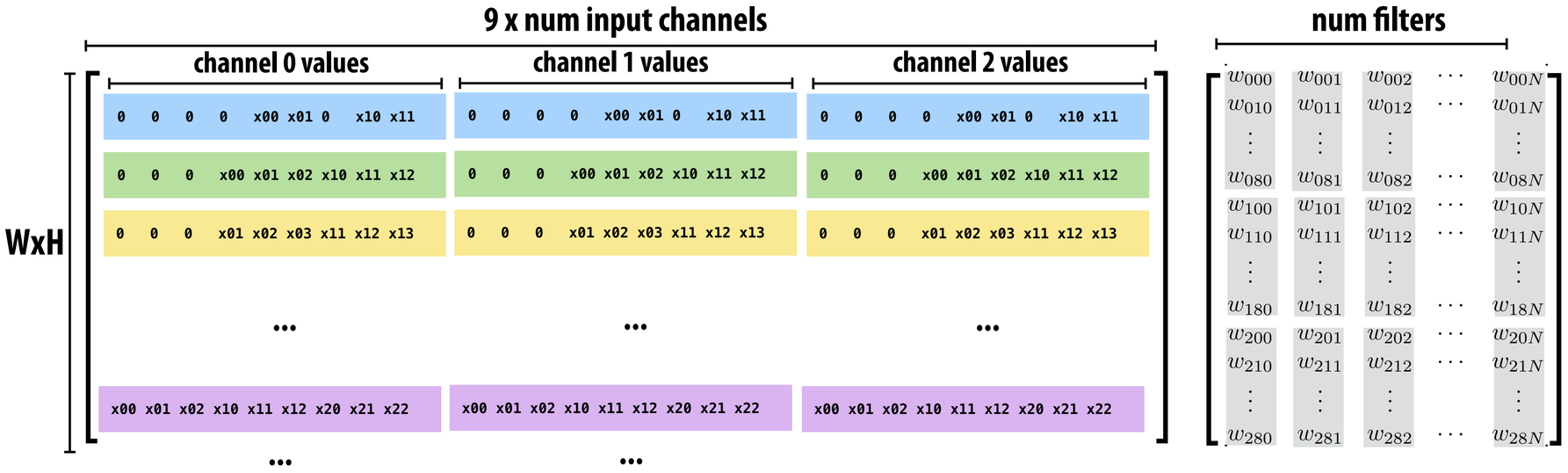


Multiple convolutions on multiple input channels



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

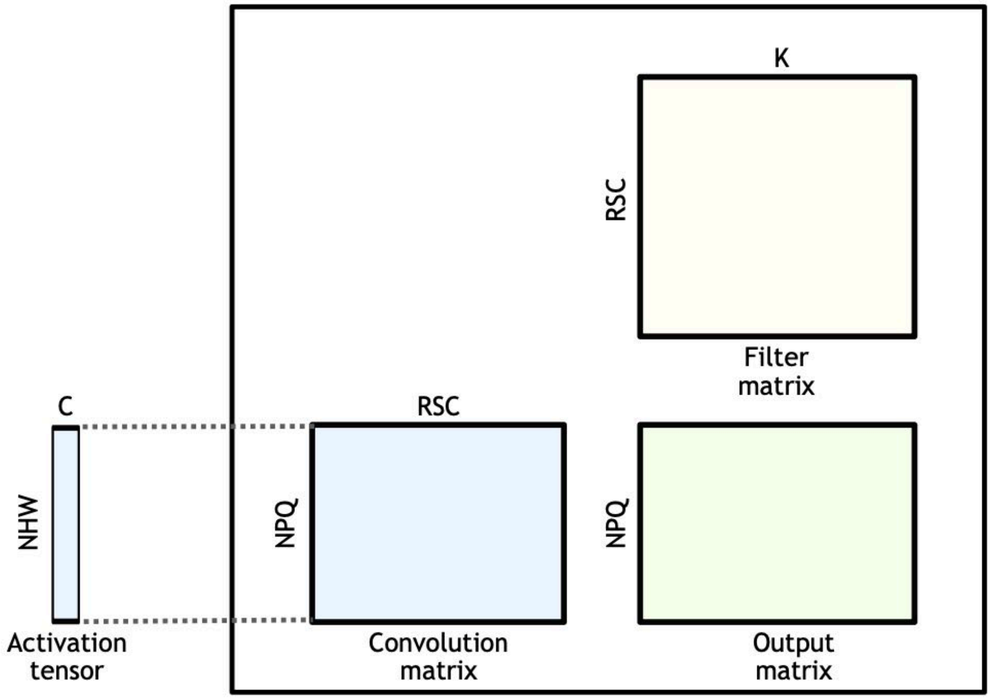


Image credit: NVIDIA

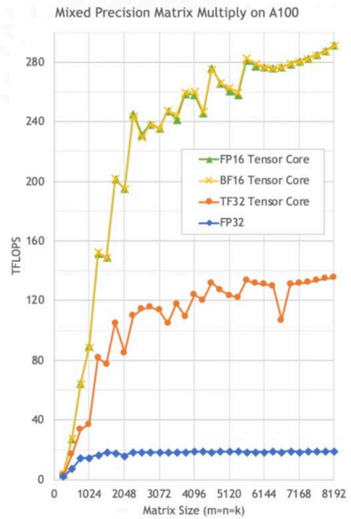
High performance implementations of GEMM exist

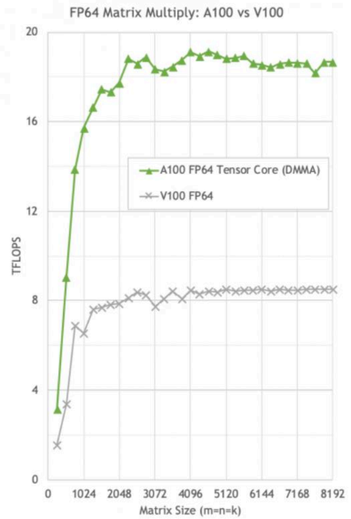
cuBLAS Performance

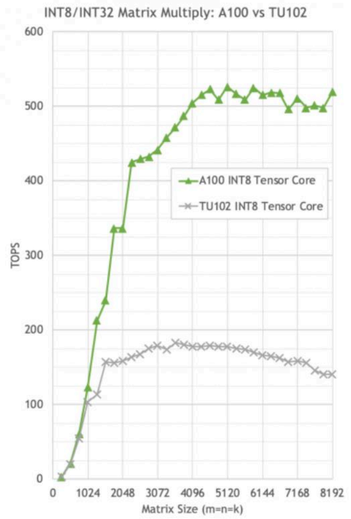
The cuBLAS library is highly optimized for performance on NVIDIA GPUs, and leverages tensor cores for acceleration of low and mixed precision matrix multiplication.

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







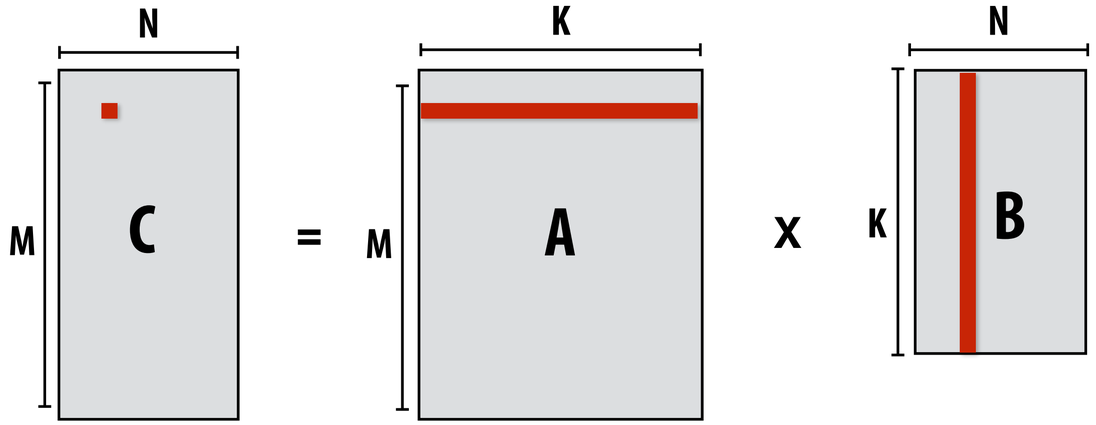
To use "off the shelf" libraries, must materialize input matrices.

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

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];
```

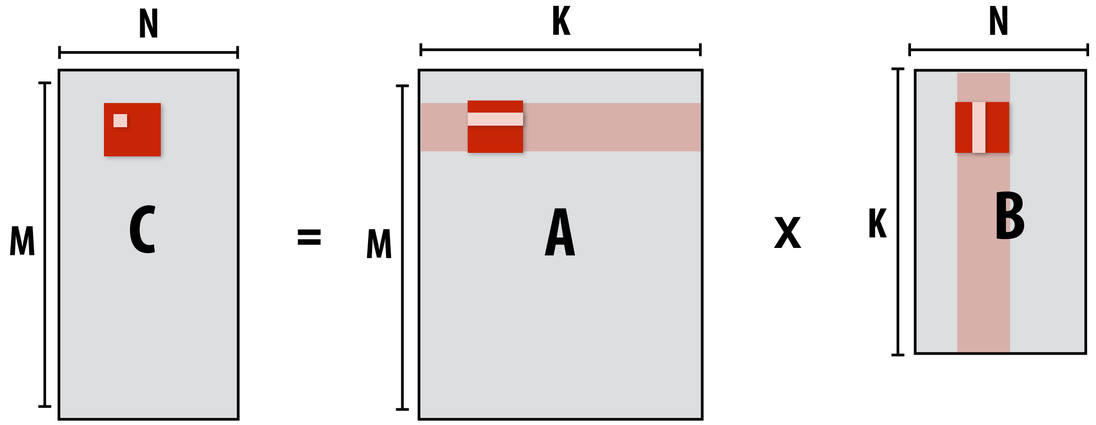


What is the problem with this implementation?

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

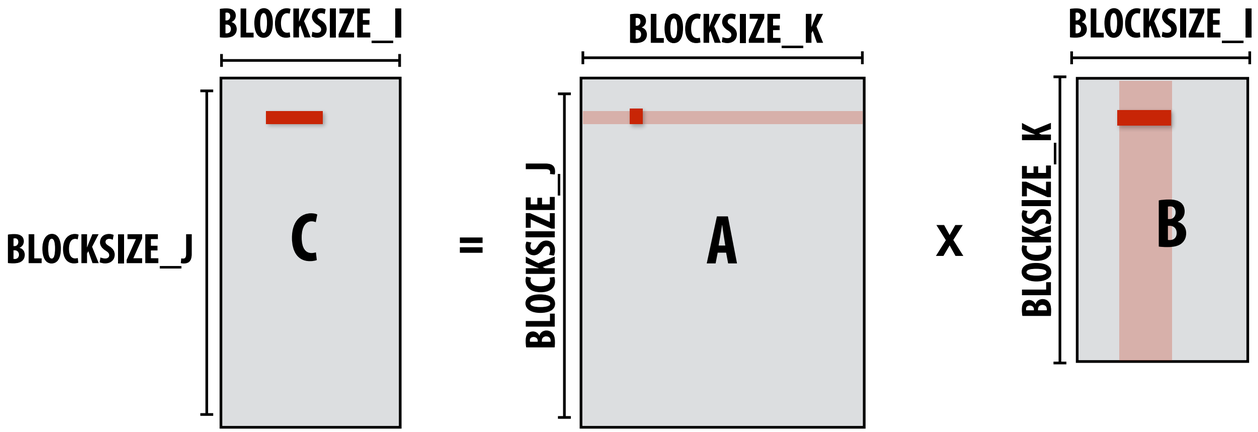
Exploit multiple levels of memory hierarchy

```
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



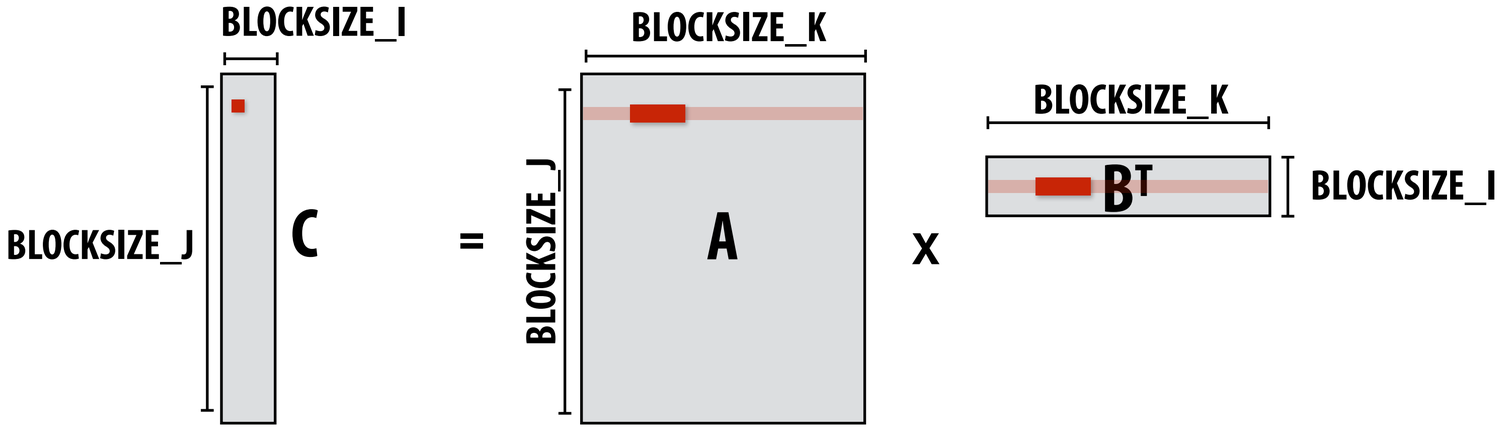

```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
   for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
      simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
      for (int k=0; k<BLOCKSIZE_K; k++) {
         // C = A*B + C
         simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single element in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



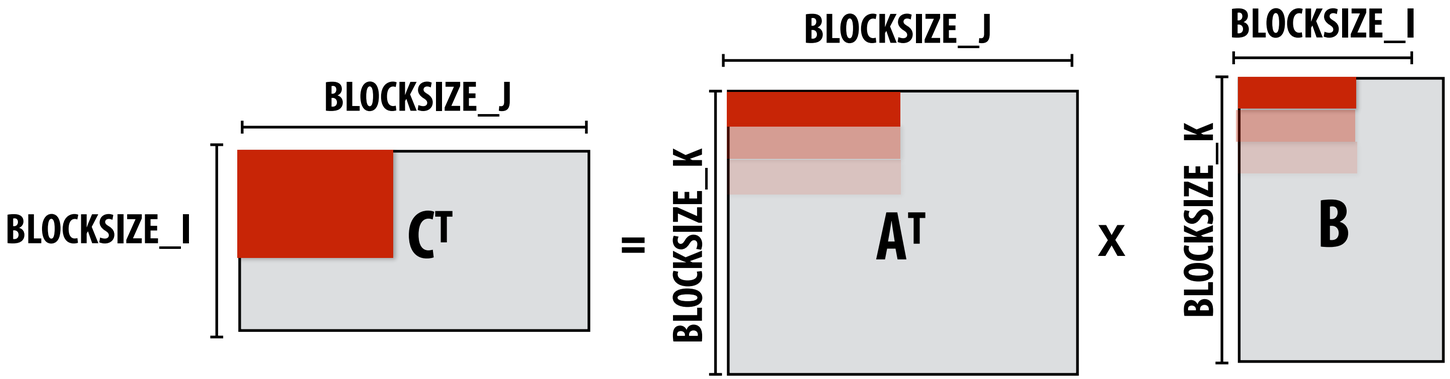

```
...
for (int j=0; j<BLOCKSIZE_J; j++)
   for (int i=0; i<BLOCKSIZE_I; i++) {
      float C_scalar = C[jblock+j][iblock+i];
      // C_scalar += dot(row of A,row of B)
      for (int k=0; k<BLOCKSIZE_K; k+=SIMD_WIDTH) {
        C_scalar += simd_dot(vec_load(&A[jblock+j][kblock+k]), vec_load(&Btrans[iblock+i][[kblock+k]);
      }
      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	$1 \times 1 \times 1000$

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

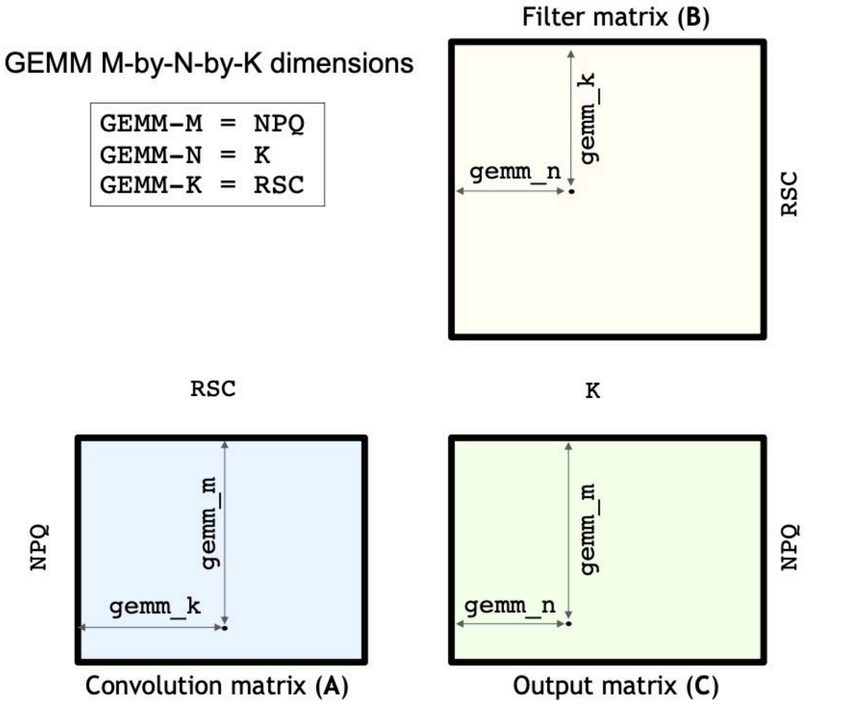
GEMM TRIPLE NEST LOOP

```
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
```



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

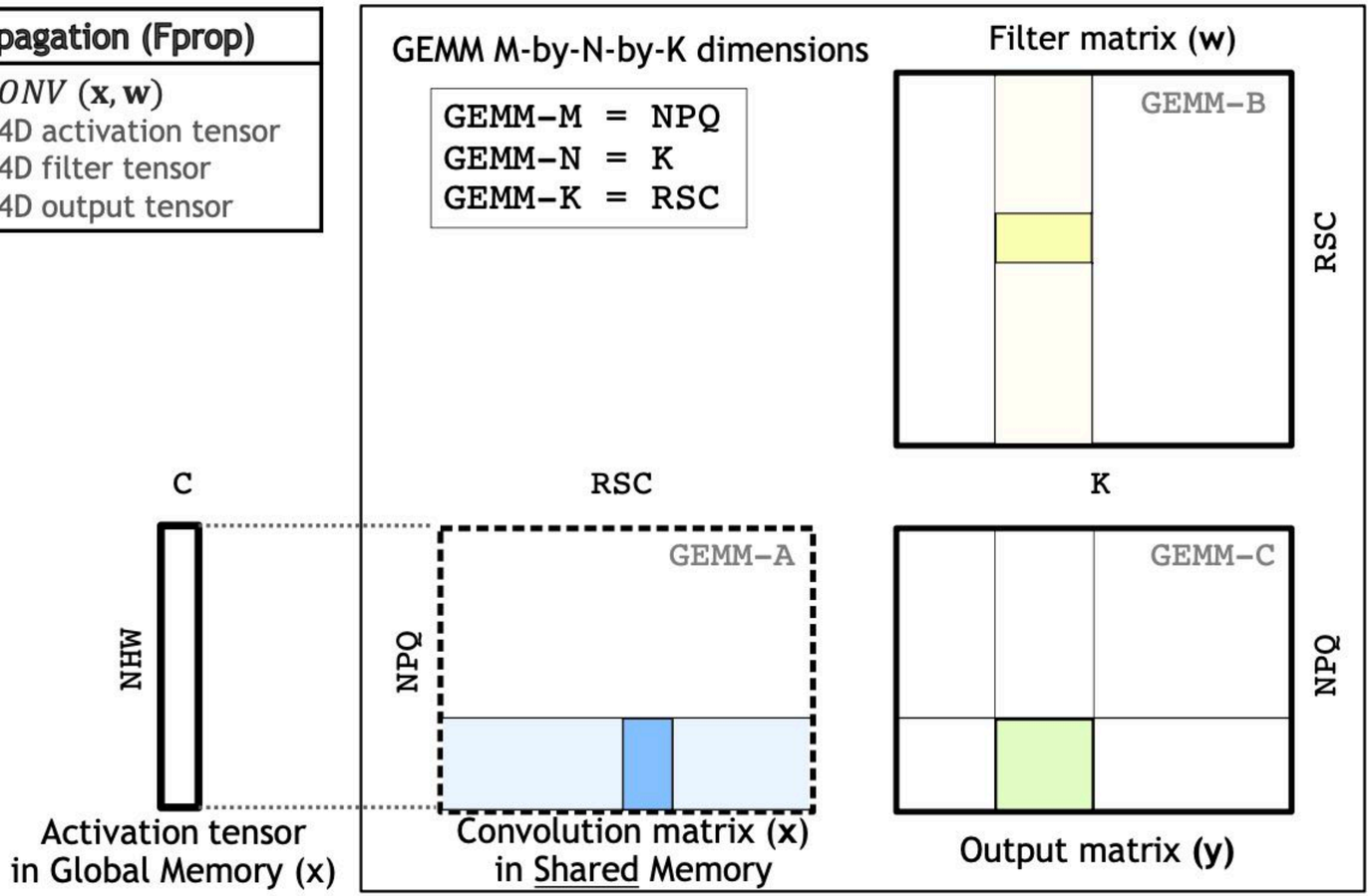
Optimization: do not materialize full matrix ("implicit gemm")

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

Image credit: NVIDIA

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.

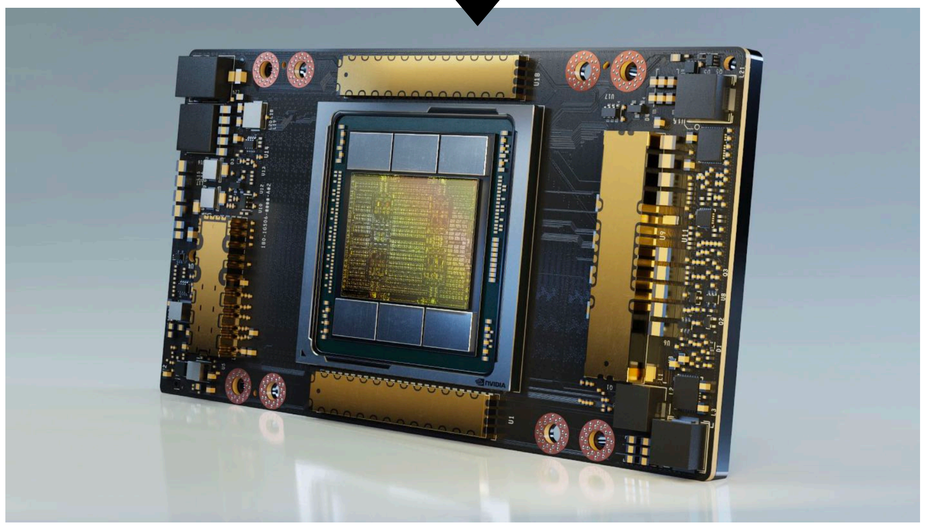
Convolutional layer in Halide

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

Func in_func;                           // assume input function (activations) is initialized

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

Func forward = Func("conv");
Var x, y, z, n;                         // z is num input channels, n is batch dimension

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

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

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

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

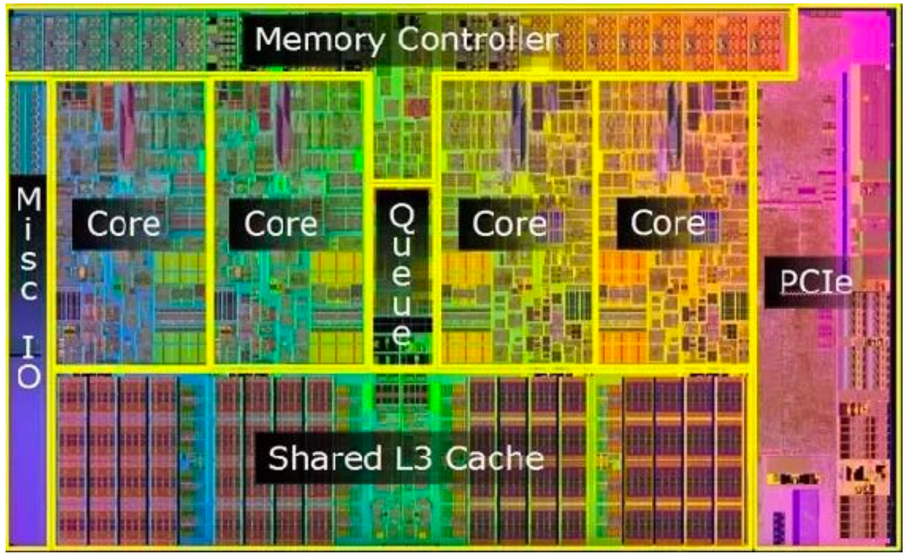
Consider scheduling this seven-dimensional loop nest!

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

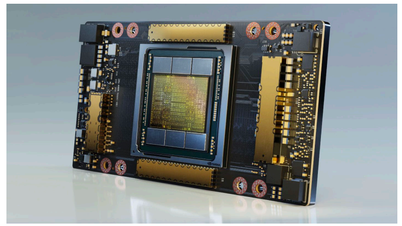
tensorflow::ops::AvgPool	Performs average pooling on the input.
tensorflow::ops::AvgPool3D	Performs 3D average pooling on the input.
tensorflow::ops::AvgPool3DGrad	Computes gradients of average pooling function.
tensorflow::ops::BiasAdd	Adds `bias` to `value`.
tensorflow::ops::BiasAddGrad	The backward 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

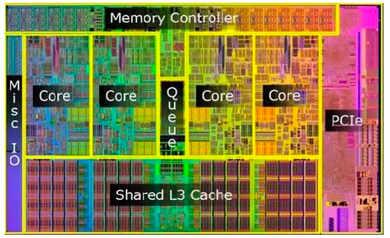
TensorFlow **NN ops**

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 wi
tensorflow::ops::DepthwiseConv2dNativeBackpropInput	Computes the gradients of depthwise convolution wi
tensorflow::ops::Dilation2D	Computes the grayscale dilation of 4-D `input` and 3-
tensorflow::ops::Dilation2DBackpropFilter	Computes the gradient of morphological 2-D dilation filter.
tensorflow::ops::Dilation2DBackpropInput	Computes the gradient of morphological 2-D dilation input.
tensorflow::ops::Elu	Computes exponential linear: `exp(features) - 1` otherwise.
tensorflow::ops::FractionalAvgPool	Performs fractional average pooling on the input.
tensorflow::ops::FractionalMaxPool	Performs fractional max pooling on the input.
tensorflow::ops::FusedBatchNorm	Batch normalization.

NVIDIA cuDNN



Intel® oneAPI Deep Neural Network Library



Example: CUDNN convolution

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

Possible algorithms:

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

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

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

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

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

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

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

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

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 1:

Is there a way to eliminate the scale/bias operation completely?

Exercise to class 2:

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

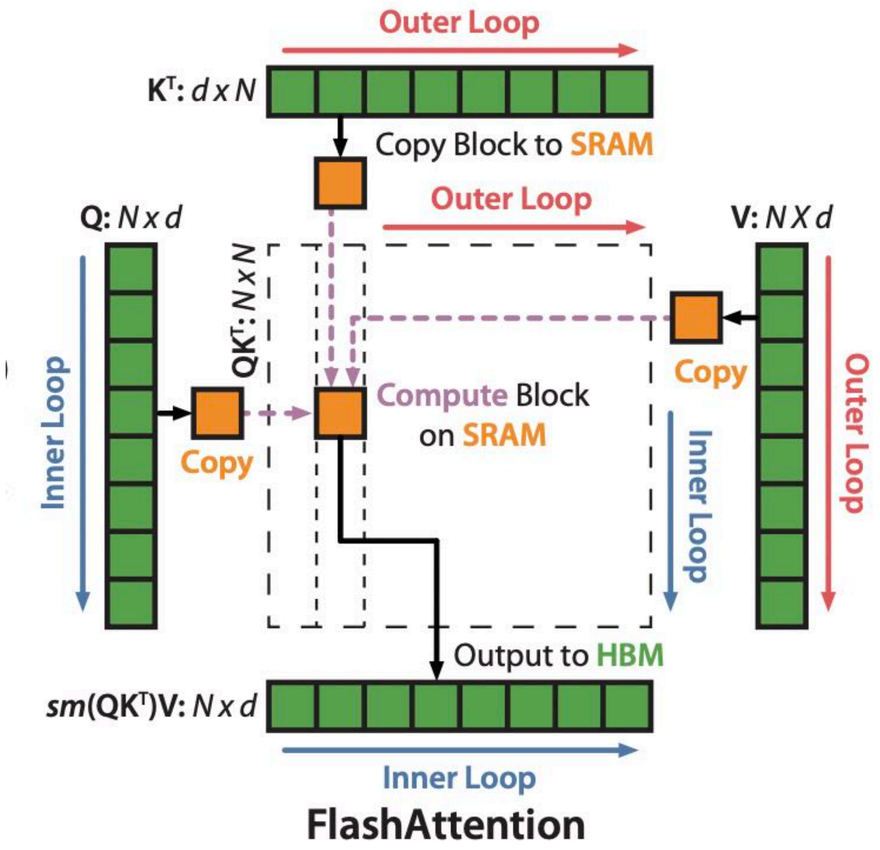
A good recent (2022) example: FlashAttention [Dao et al. 2022]

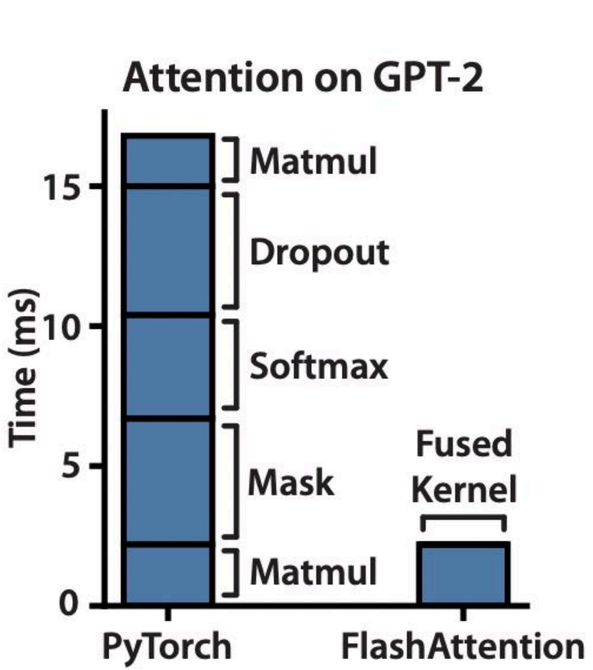
- Improves performance of attention layers through clever fusion/blocking of matrix multiplication and softmax
- Significantly reduces memory accesses to off-chip memory

$$sm(QK^T)V$$

***sm*() is the softmax function on matrix**

Q: N x d
K^T: d x N
V: N x d





Old style: hardcoded “fused” ops

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

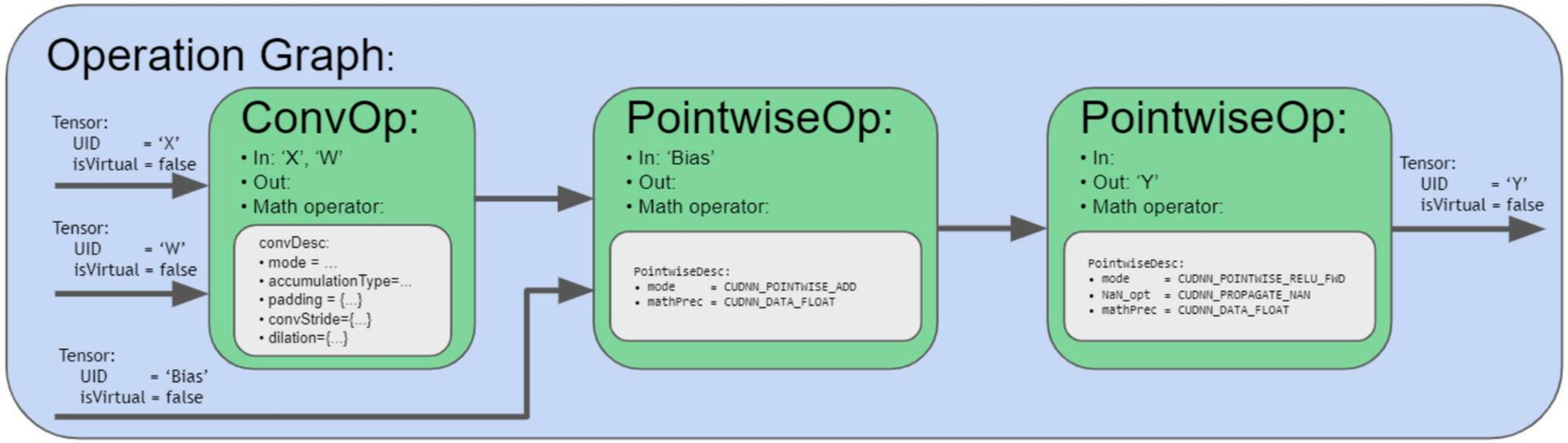
This function applies a bias and then an activation to the convolutions or cross-correlations of cudnnConvolutionForward(), returning results in `y`. The full computation follows the equation `y = act ( alpha1 * conv(x) + alpha2 * z + bias )`.

Tensorflow:

tensorflow::ops::FusedBatchNorm	Batch normalization.

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

Fusion example: CUDNN “backend"

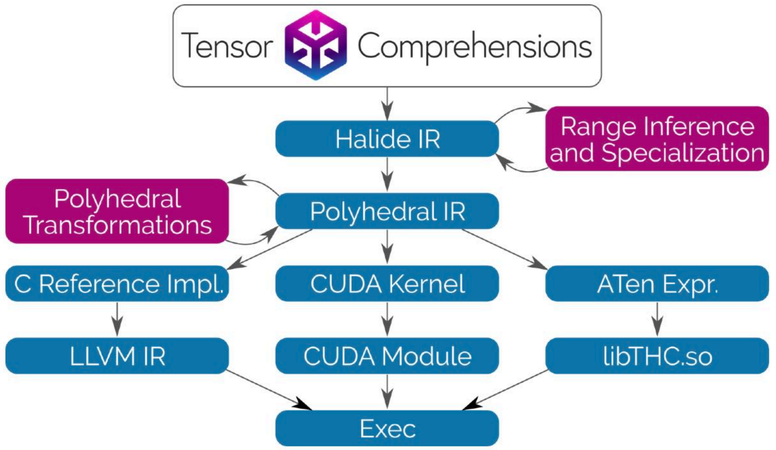


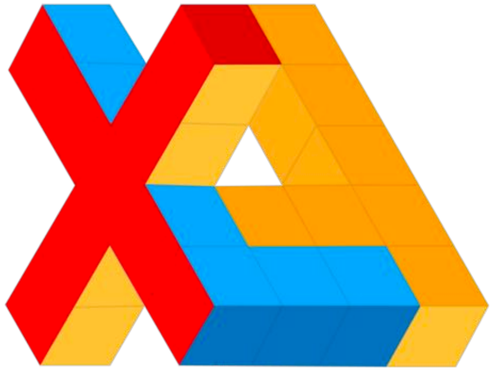
Note for operation fusion use cases, there are two different mechanisms in cuDNN to support them. First, there are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If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

Compiler 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 efforts to automatically schedule key DNN operations



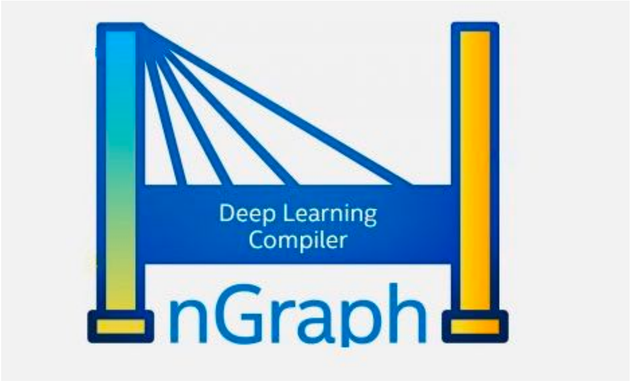


tvm Open Deep Learning Compiler Stack

license Apache 2.0 build passing

Documentation | Contributors | Community | Release Notes

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



NVIDIA TensorRT

Programmable Inference Accelerator

Use of low precision values

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

XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks

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

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

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

Optimization techniques

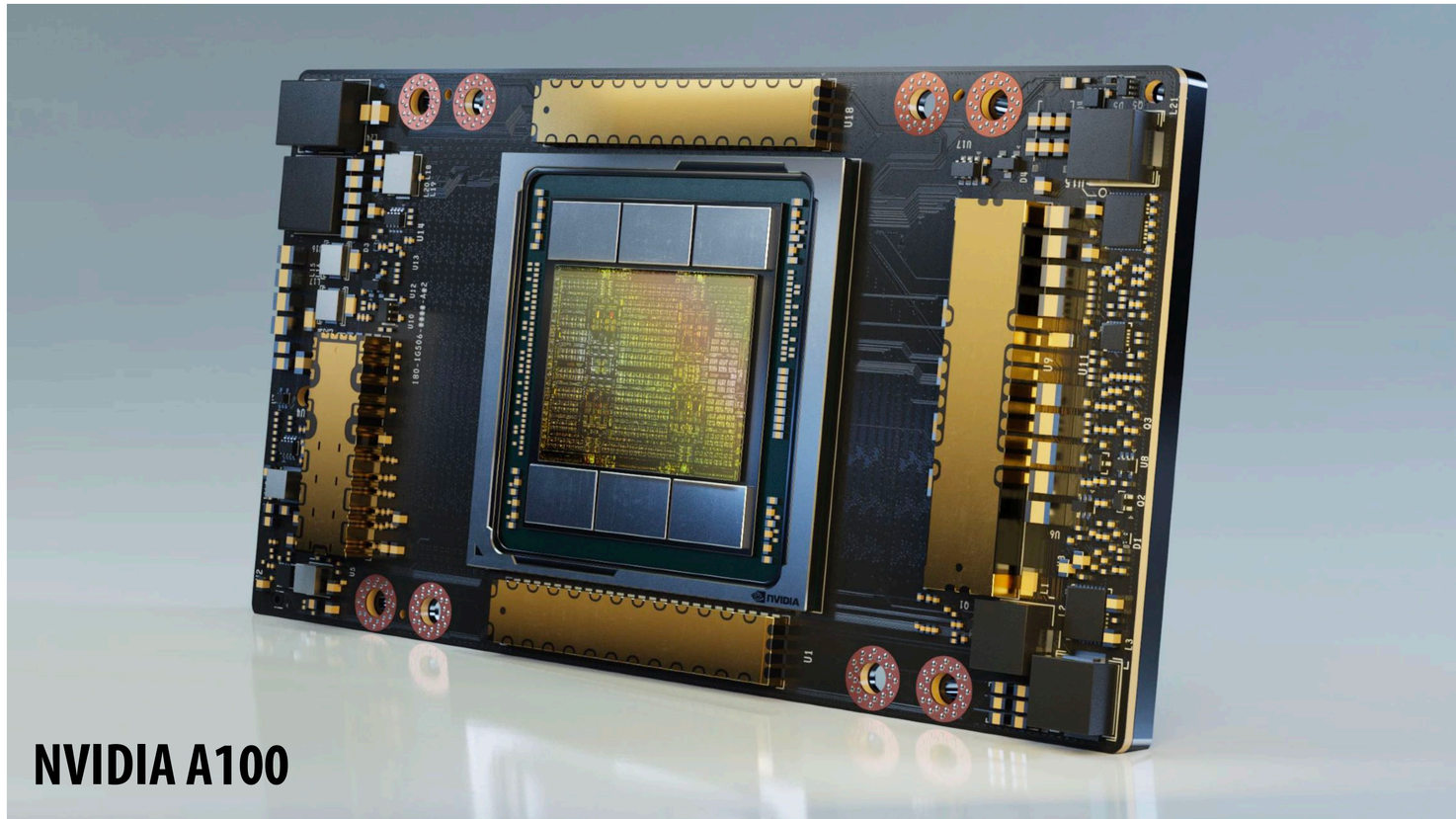
- Better algorithms: manually designing better models
  - Common parameters: depth of network, width of filters, number of filters per layer, convolutional stride, etc.
  - Common to perform automatic search for efficient topologies
- Software optimization: Good scheduling of performance-critical operations (layers)
  - Loop blocking/tiling, fusion
  - Typically optimized manually by humans (but significant research efforts to automate scheduling)
- 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 needed?)

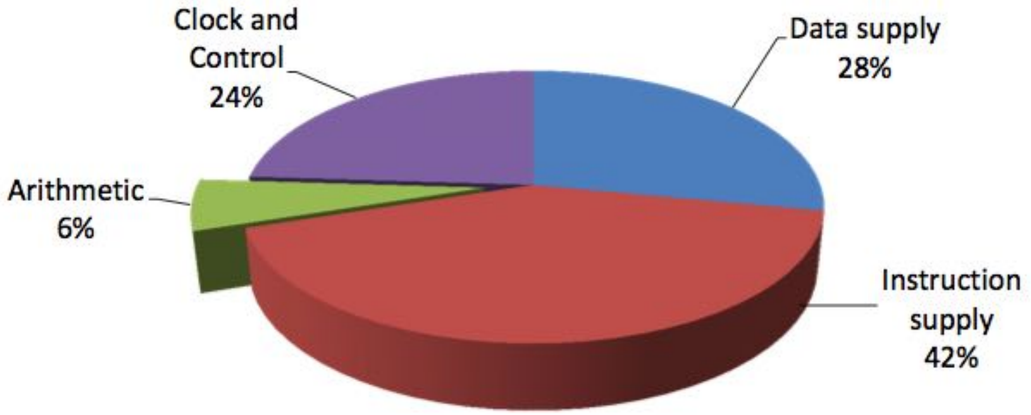
Special instruction support

Recall: compute specialization = energy efficiency

- Rules of thumb: compared to high-quality C code on CPU...

- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC ("application-specific integrated circuit")
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and and is not floating-point math



Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

[Source: Chung et al. 2010 , Dally 08]

Recall: data movement has high energy cost

- Rule of thumb in modern system design: always seek to reduce amount of data movement in a computer

- "Ballpark" numbers
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

[Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

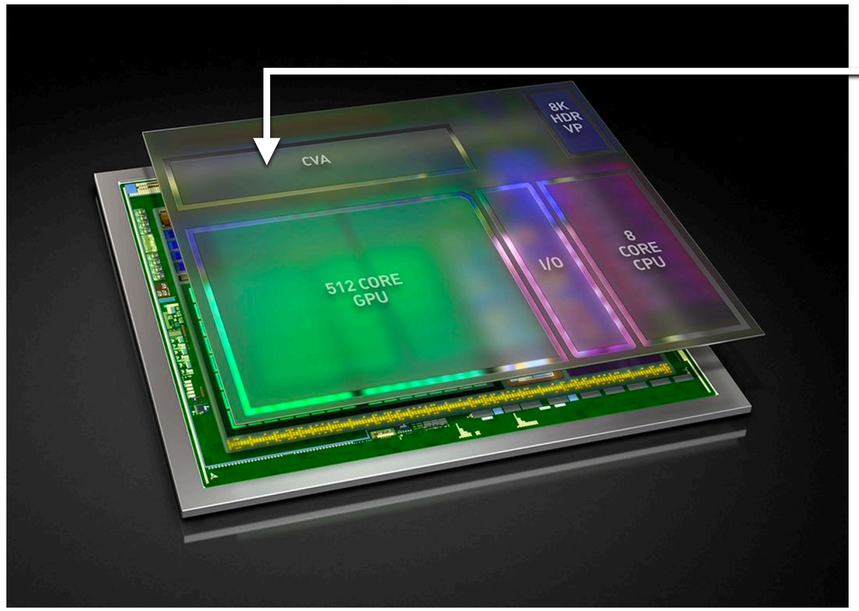
Amortize overhead of instruction stream control using more complex instructions

- Fused multiply add (ax + b)
- 4-component dot product x = A dot B
- 4x4 matrix multiply
  - AB + C for 4x4 matrices A, B, C

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

Efficiency estimates *

- Estimated overhead of programmability (instruction stream, control, etc.)
  - Half-precision FMA (fused multiply-add) 2000%
  - Half-precision DP4 (vec4 dot product) 500%
  - Half-precision 4x4 MMA (matrix-matrix multiply + accumulate) 27%



NVIDIA Xavier (SoC for automotive domain)

Features a Computer Vision Accelerator (CVA), a custom module for deep learning acceleration (large matrix multiply unit)

~ 2x more efficient than NVIDIA V100 MMA instruction despite being highly specialized component. (includes optimization of gating multipliers if either operand is zero)

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

Ampere GPU SM (A100)

Each SM core has:

64 fp32 ALUs (mul-add)

32 int32 ALUs

4 "tensor cores"

Execute 8x4 x 4x8 matrix mul-add instr

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

A, B stored as fp16, accumulation with fp32 C

There are 108 SM cores in the GA100 GPU:

6,912 fp32 mul-add ALUs

432 tensor cores

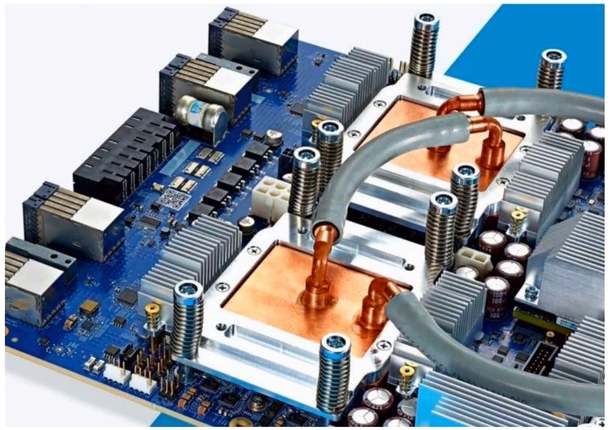
1.4 GHz max clock

= 19.5 TFLOPs fp32

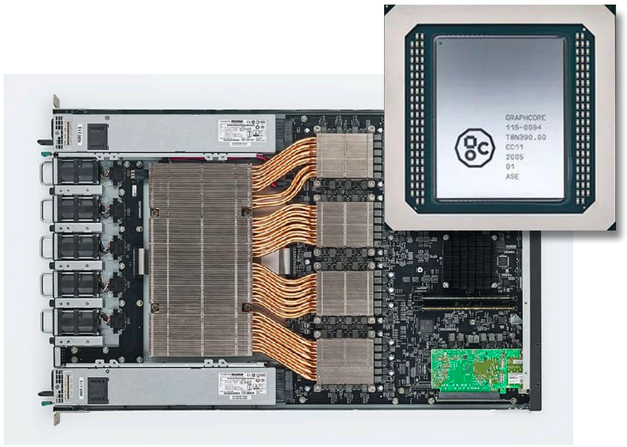
+ 312 TFLOPs (fp16/32 mixed) in tensor cores

Single instruction to perform
2x8x4x8 FP16 + 8x8 TF32 ops

Hardware acceleration of DNN inference/training



Google TPU3

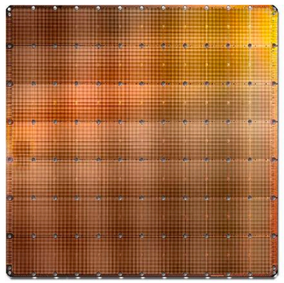


GraphCore IPU

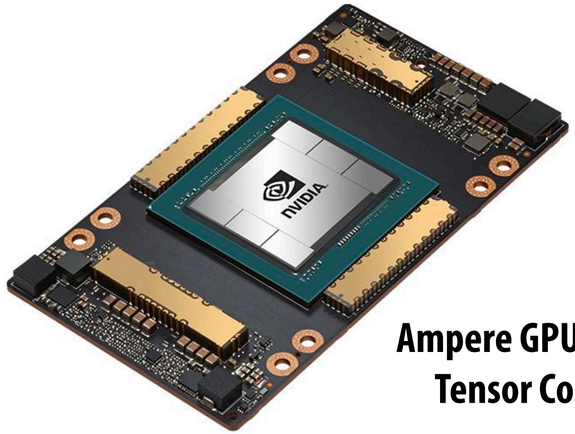
Apple Neural Engine

Intel Deep Learning
Inference Accelerator

SambaNova
Cardinal SN10



Cerebras Wafer Scale Engine



Ampere GPU with
Tensor Cores

Investment in AI hardware

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

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

April 13, 2021 09:00 AM Eastern Daylight Time

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

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

Tweet this

This S
and ro

Now th
world,
legacy
hardwa
solutio

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

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

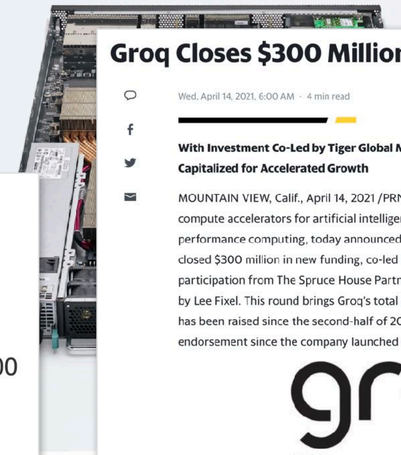
Artificial intelligence chip startup Cerebras Systems claims it has the "world's fastest AI supercomputer," thanks to its large Wafer Scale Engine processor that comes with 400,000 compute cores.

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

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

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

Comment



Groq Closes $300 Million Fundraise

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

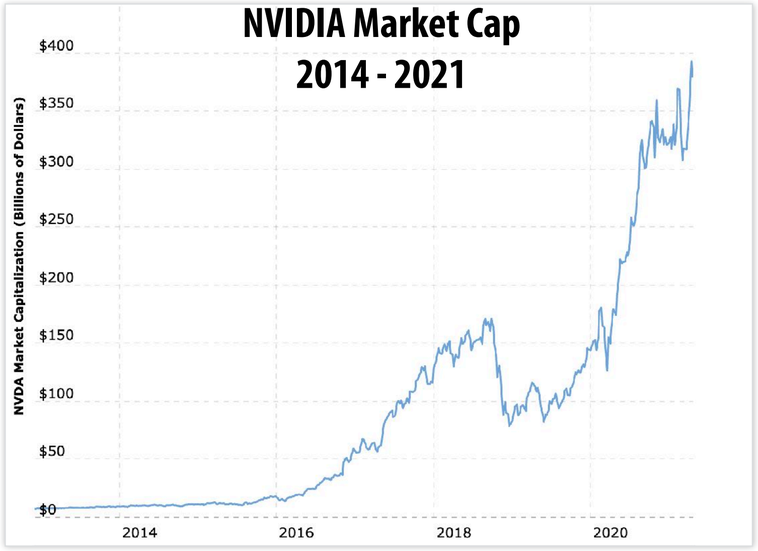
With Investment Co-Led by Tiger Global Management and D1 Capital, Groq Is Well Capitalized for Accelerated Growth

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

groq™

Groq logo

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



Intel Acquires Artificial Intelligence Chipmaker Habana Labs

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

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

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

Google's TPU (v1)

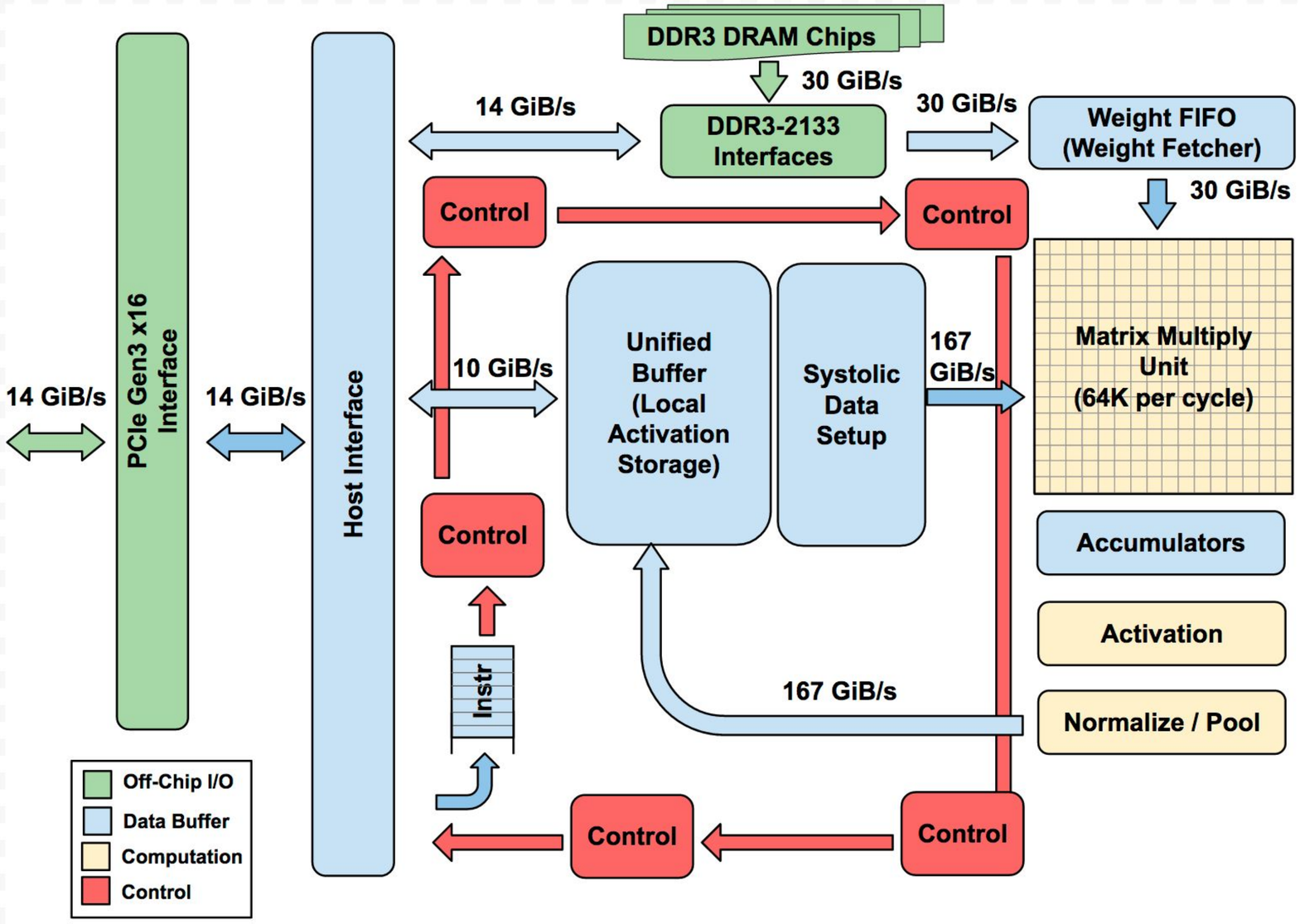
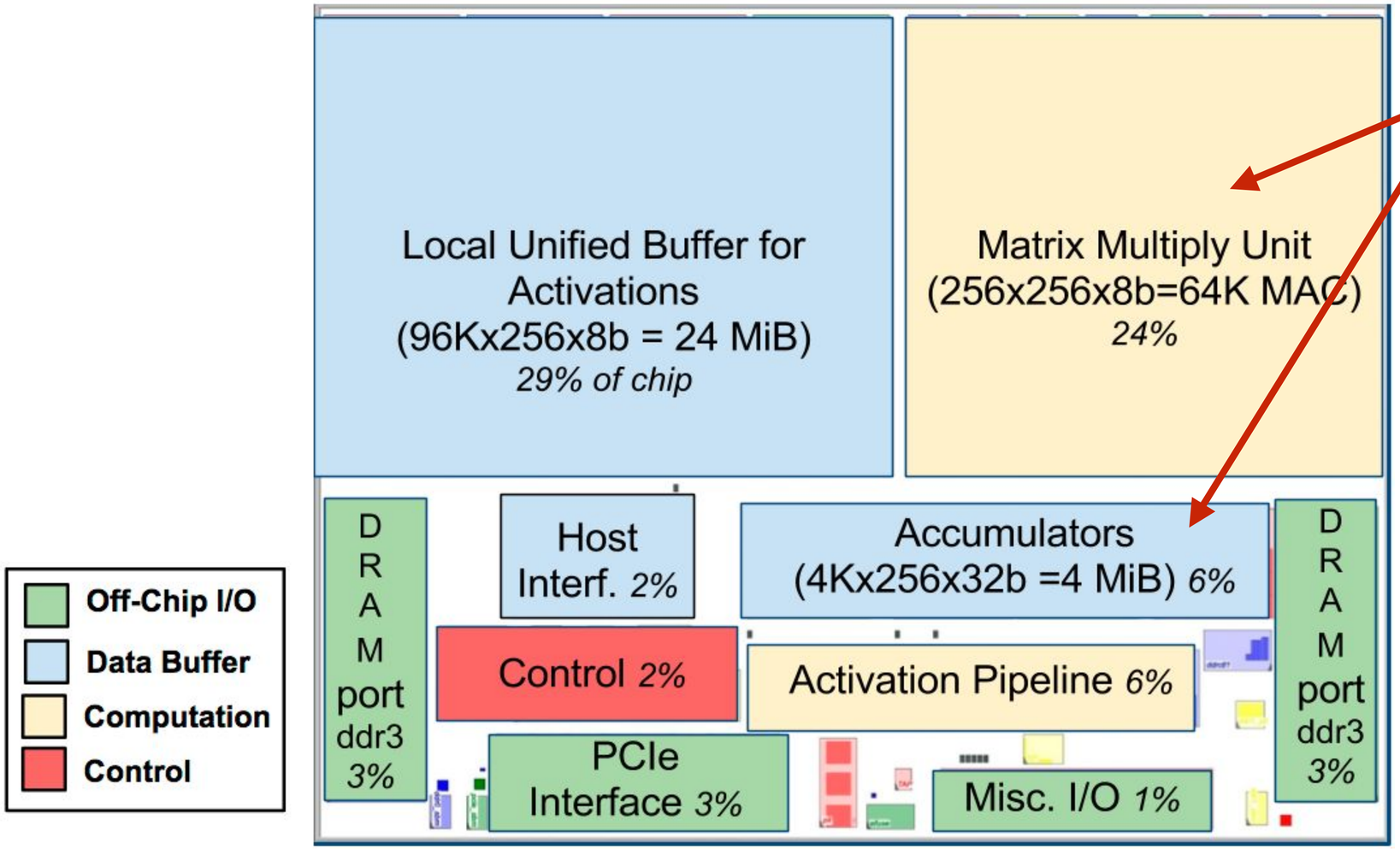


Figure credit: Jouppi et al. 2017

TPU area proportionality



Arithmetic units ~ 30% of chip

Note low area footprint of control

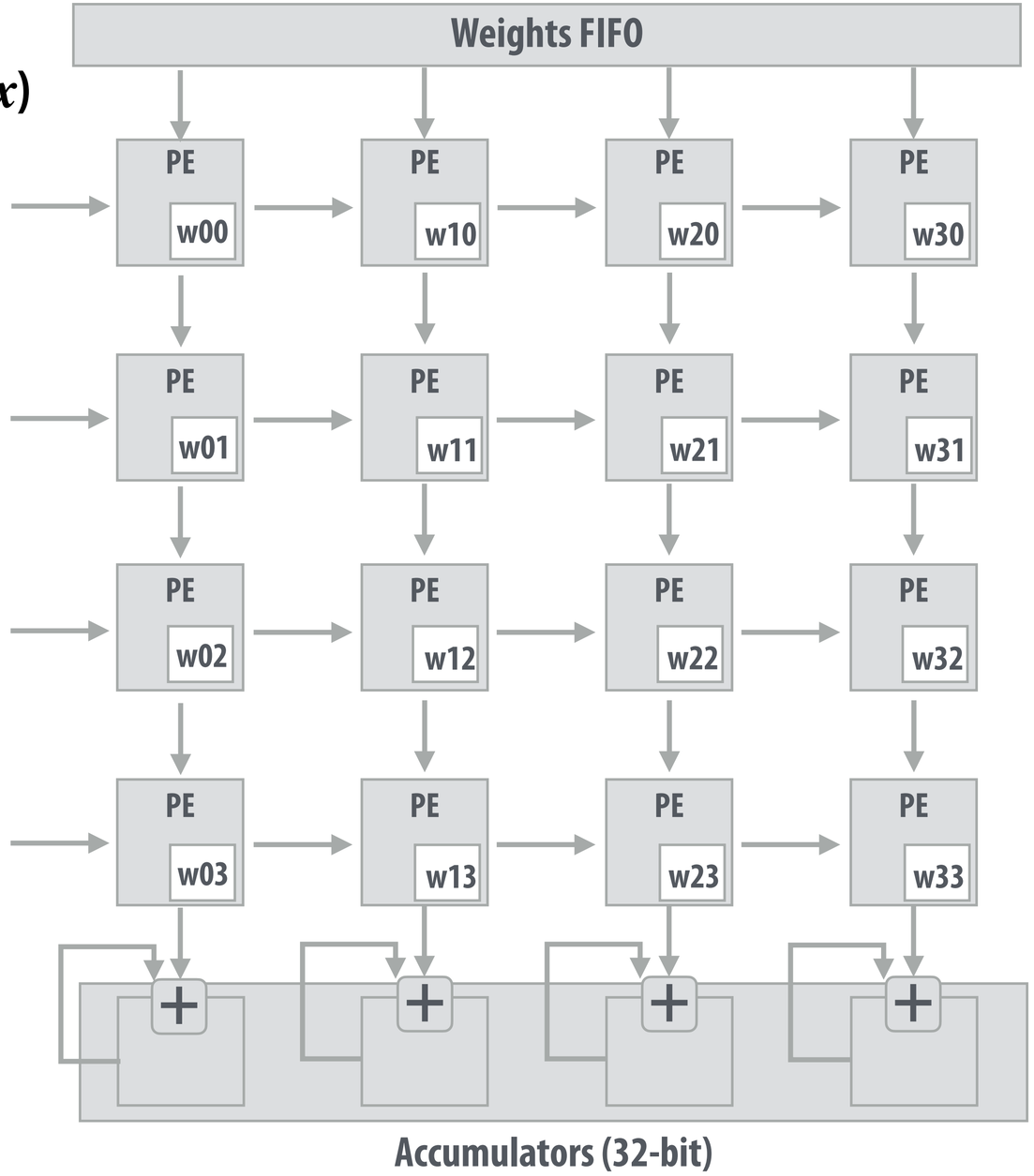
Key instructions:

- read host memory
- write host memory
- read weights
- matrix_multiply / convolve
- activate

Figure credit: Jouppi et al. 2017

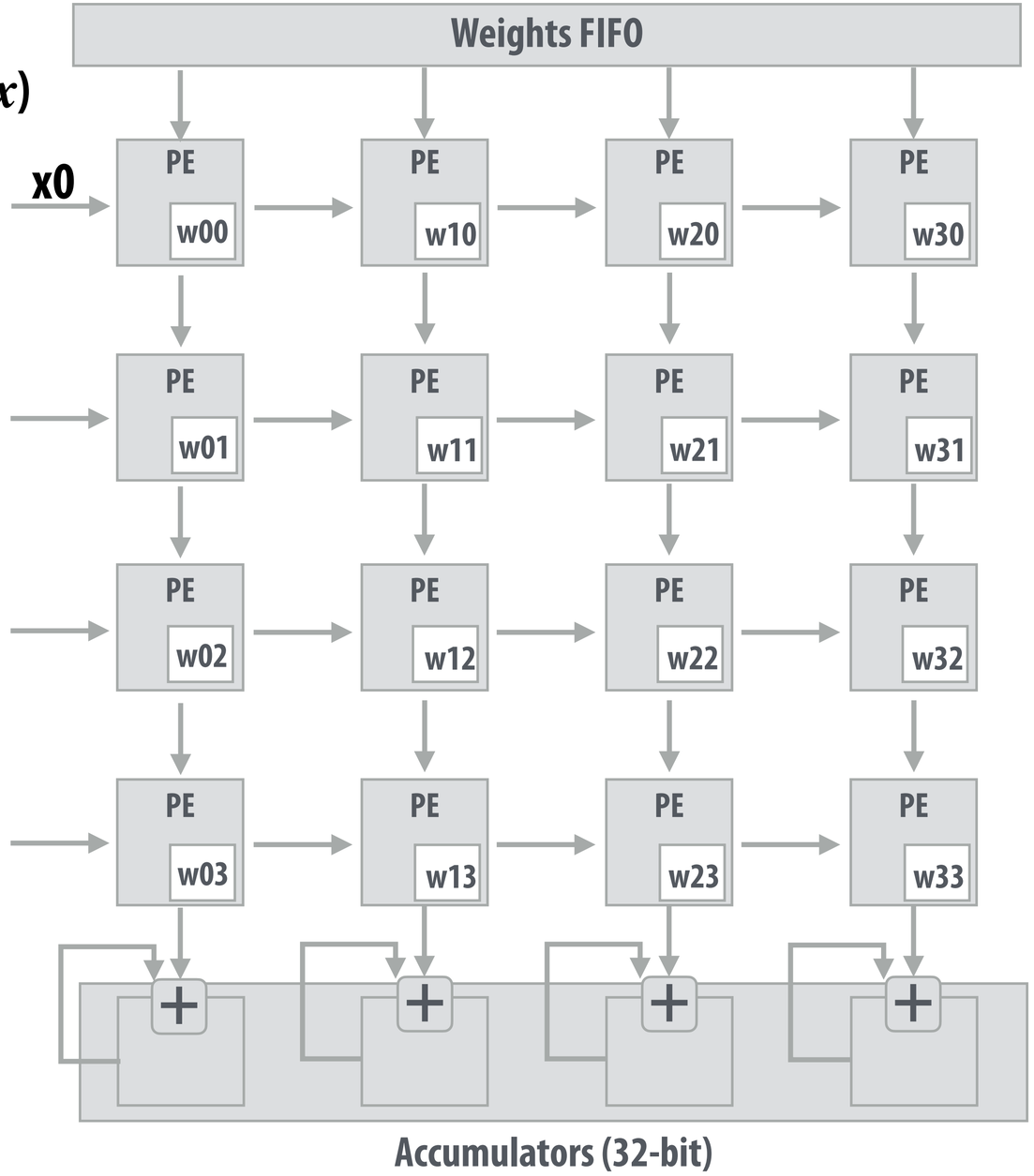
Systolic array

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



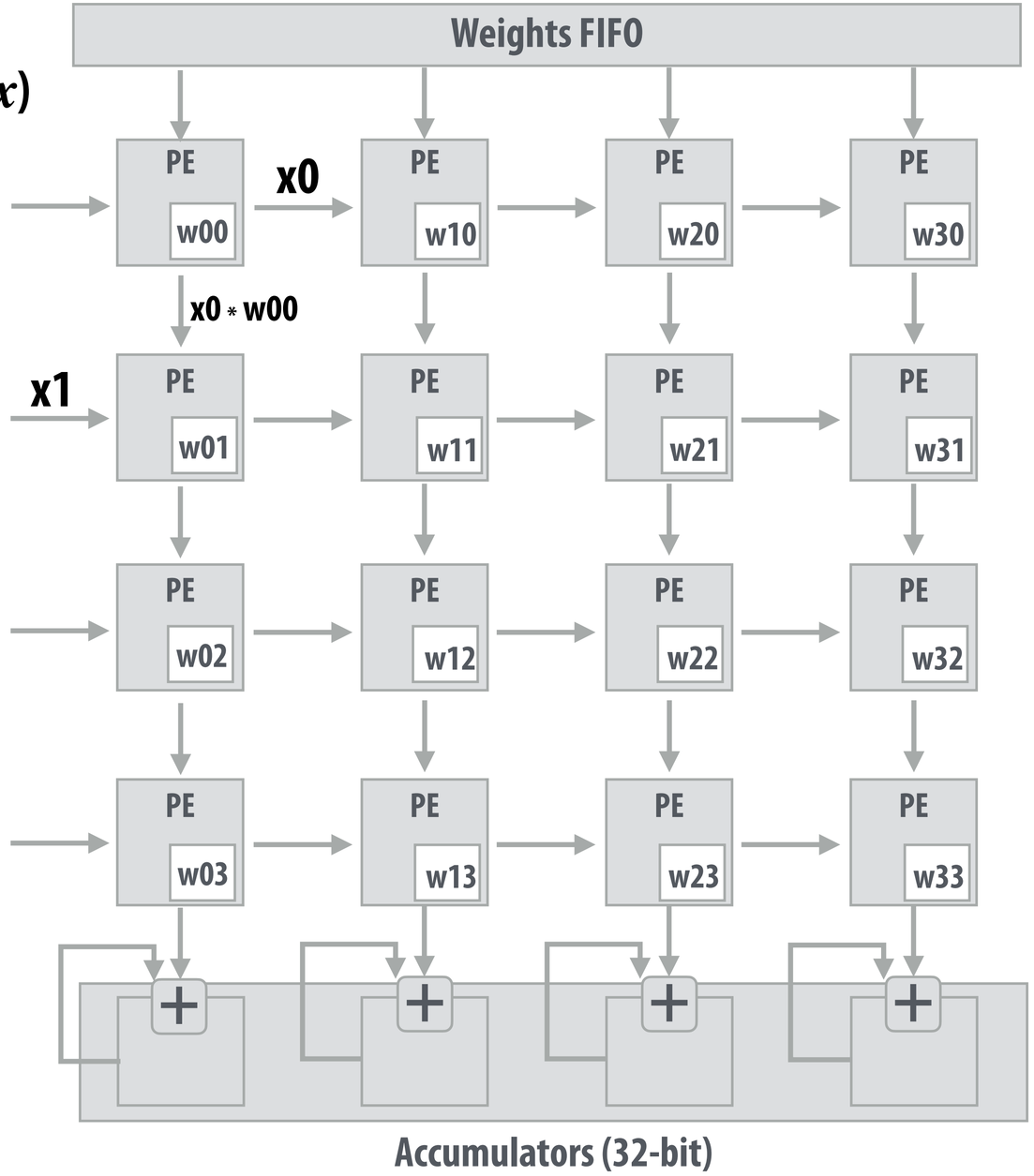
Systolic array

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



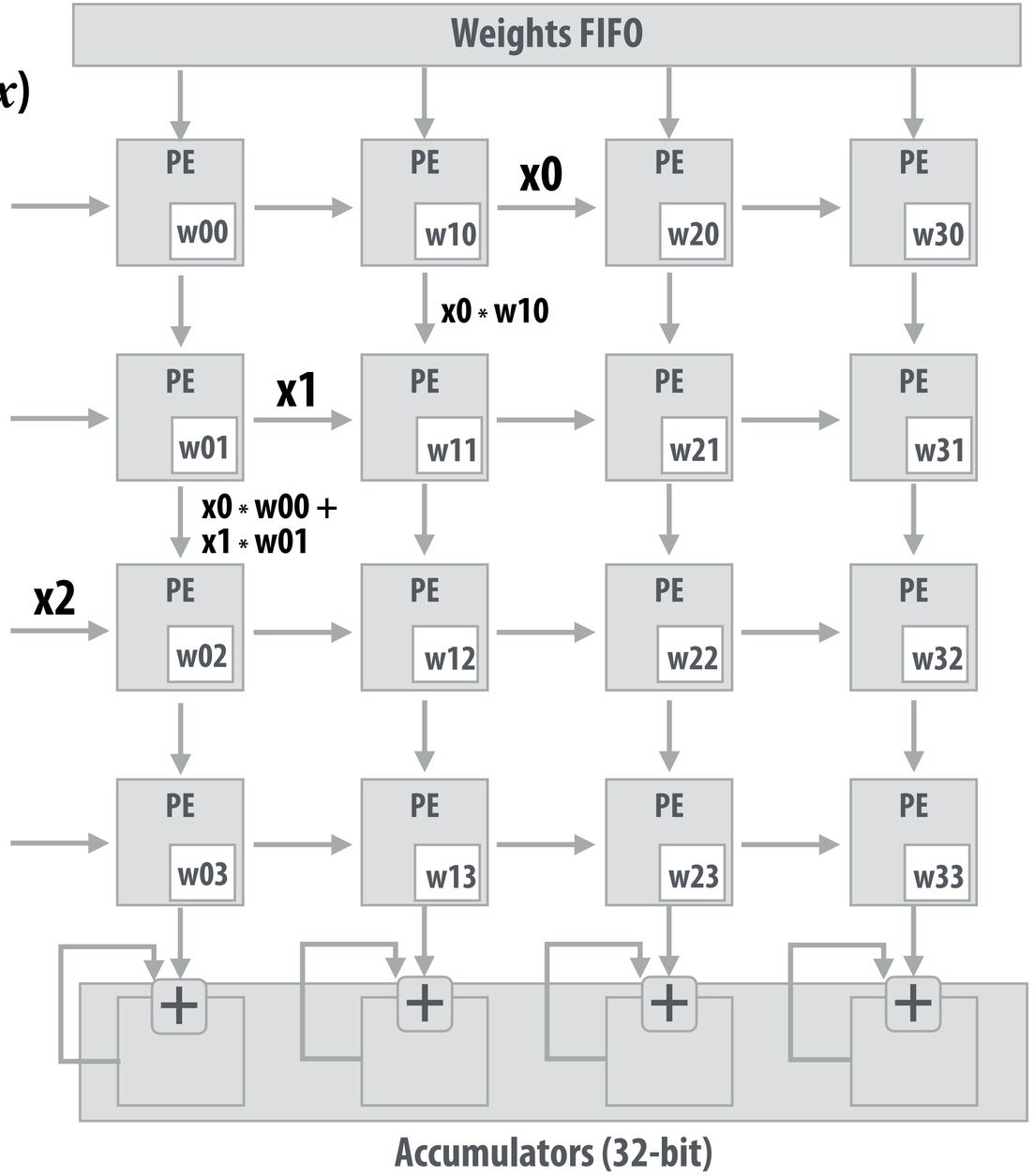
Systolic array

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



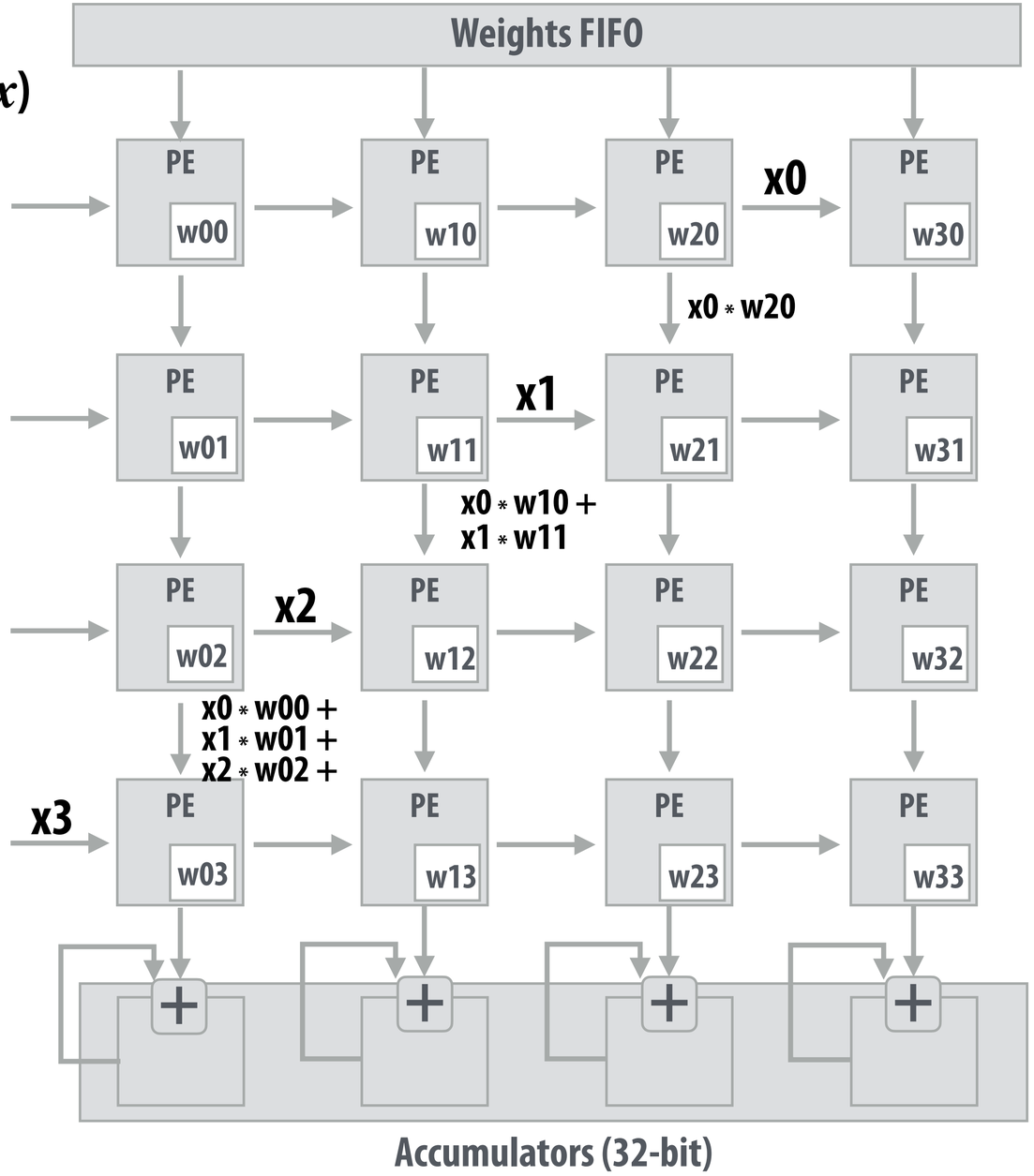
Systolic array

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



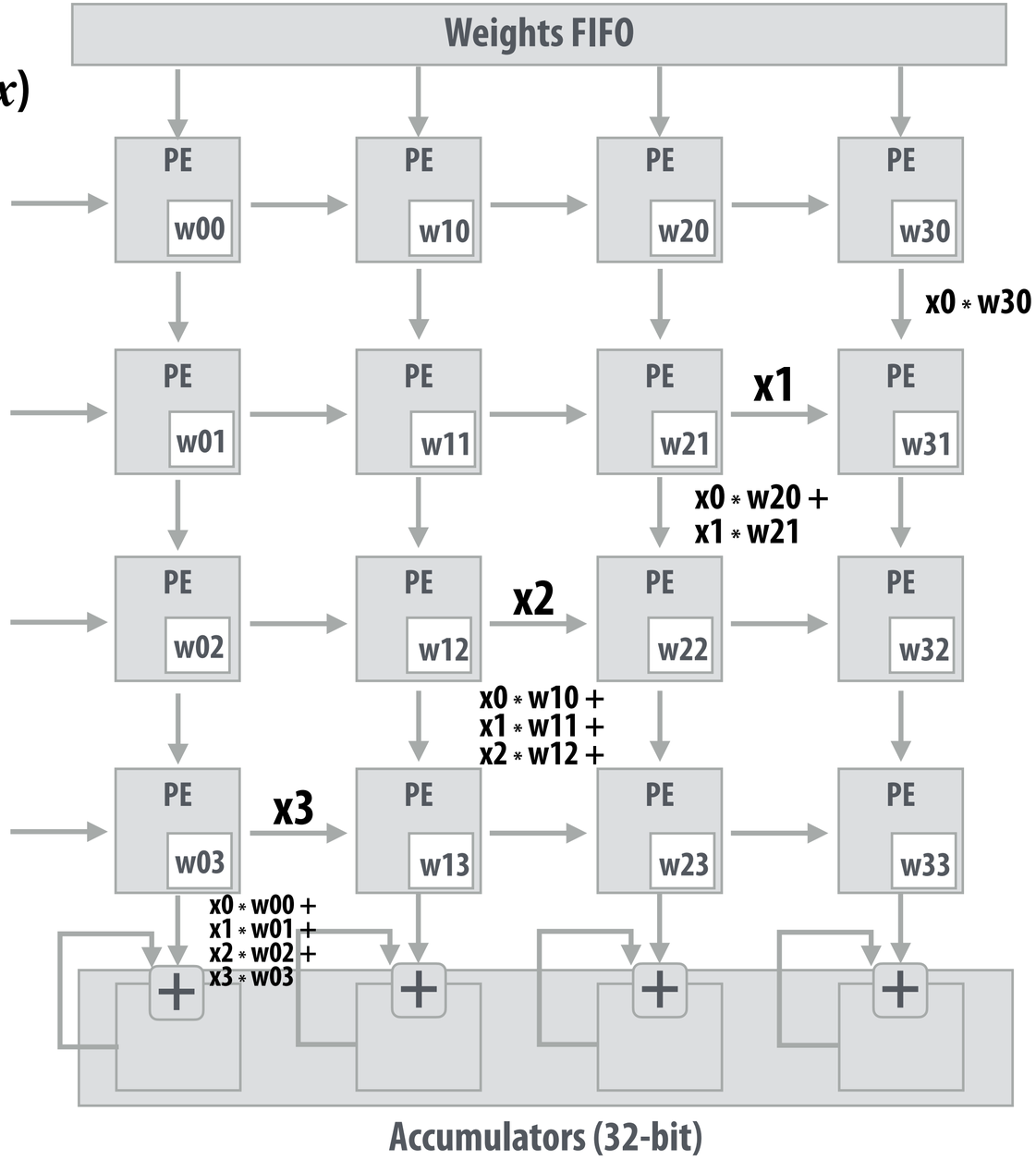
Systolic array

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



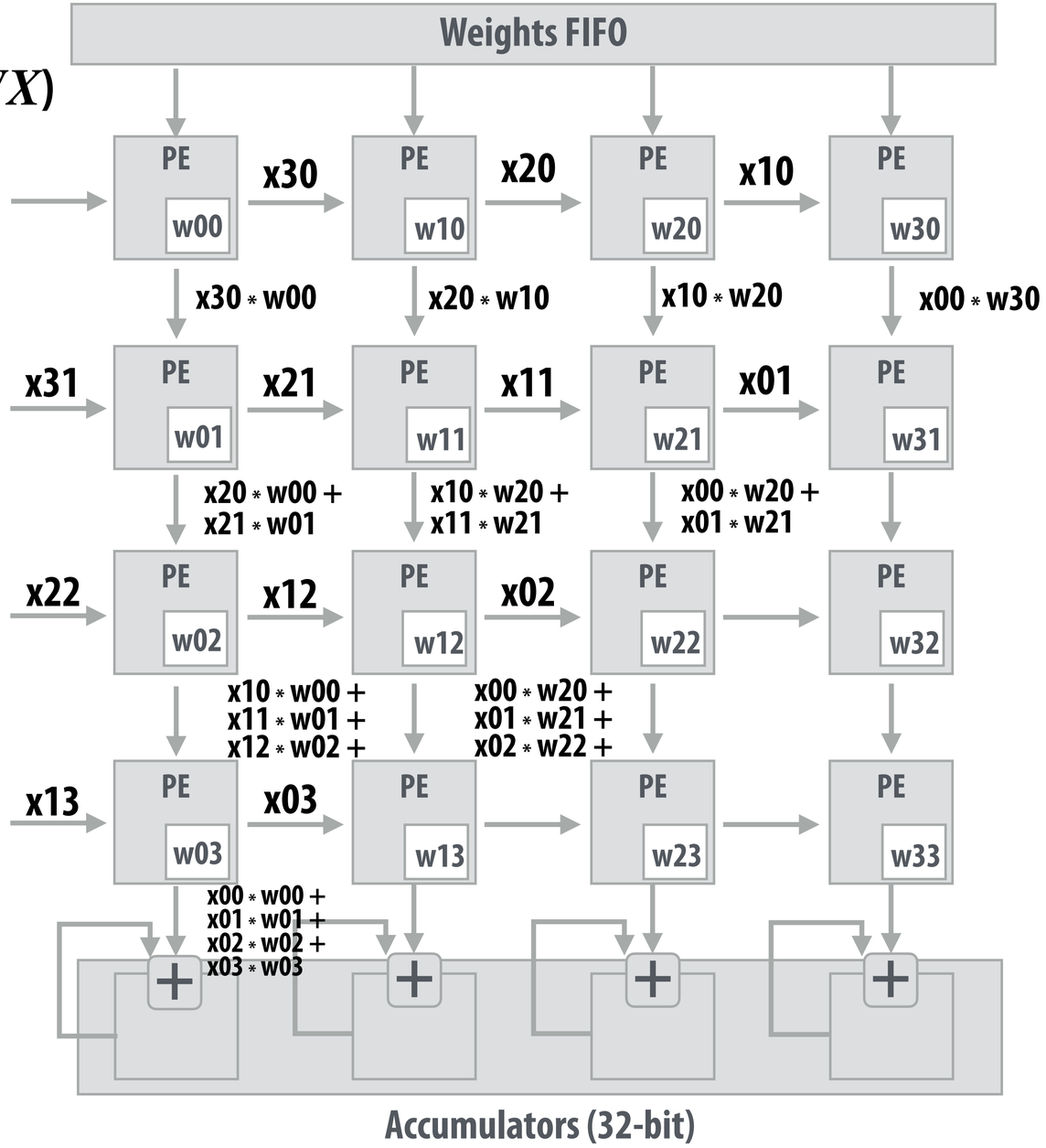
Systolic array

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



Systolic array

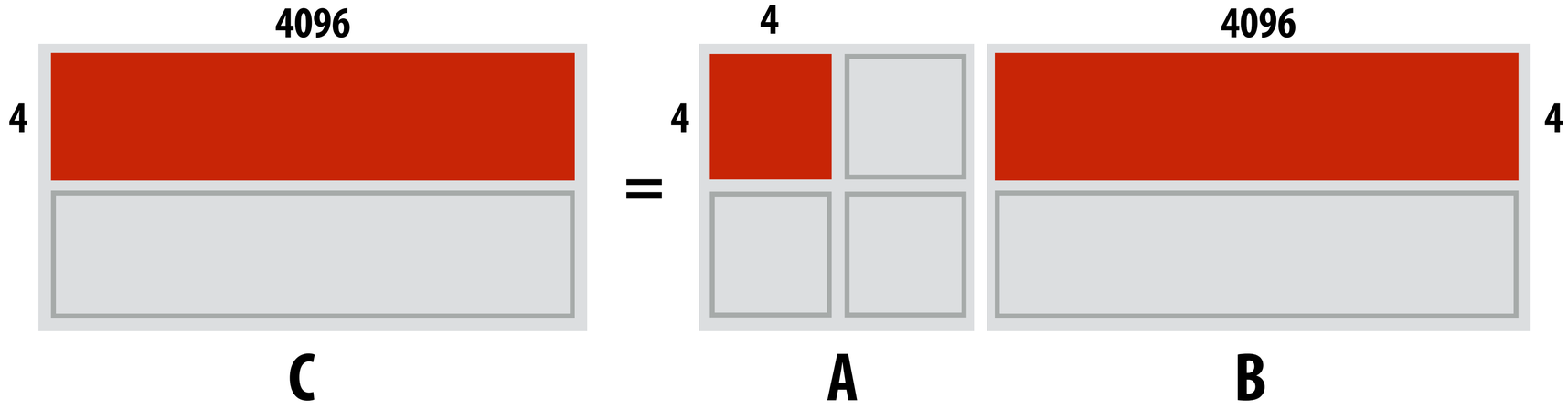
(matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

Building larger matrix-matrix multiplies

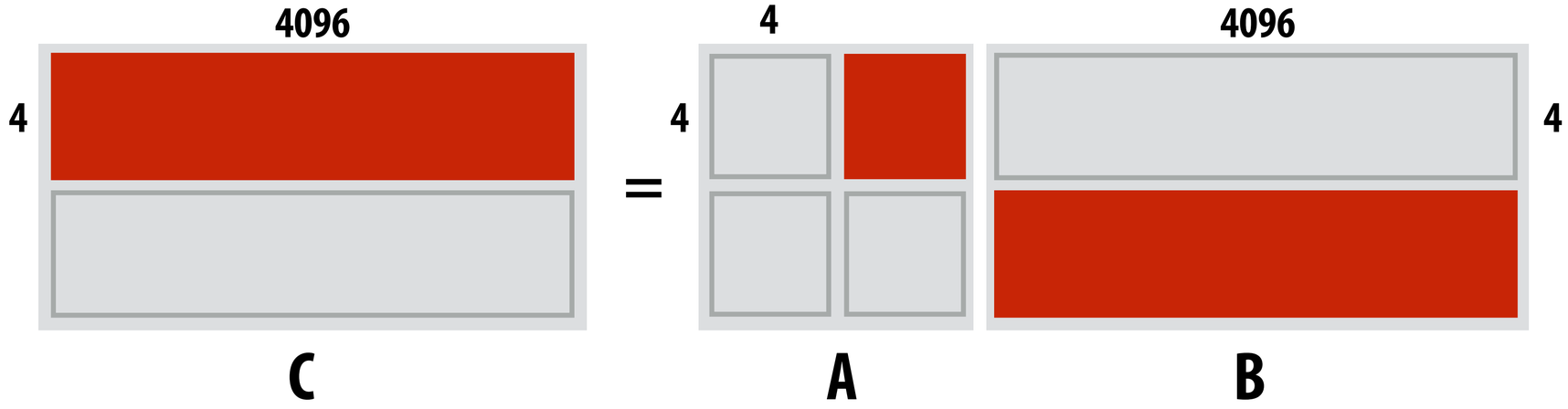
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

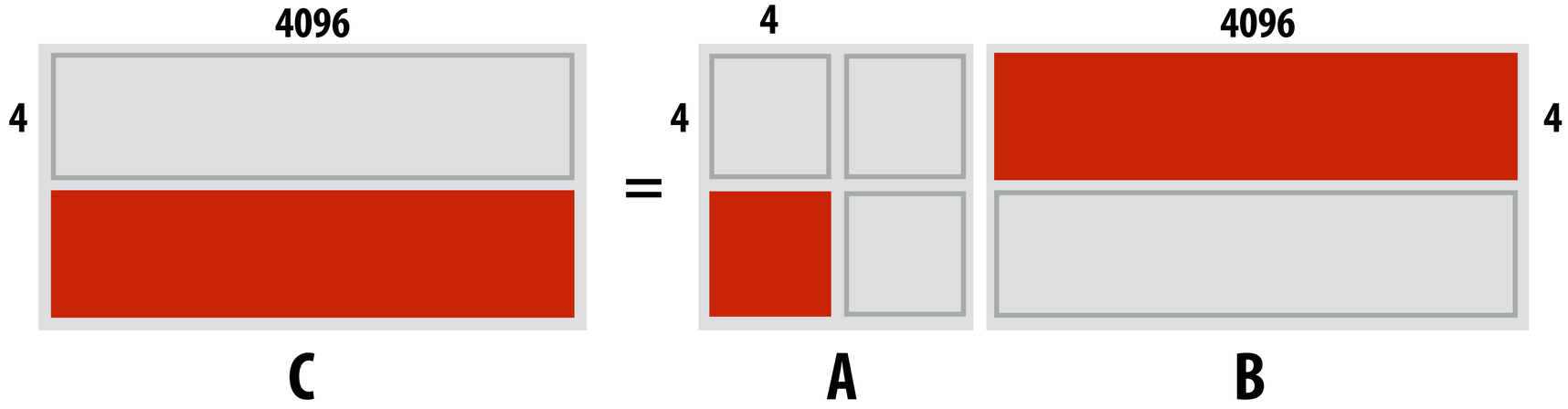
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

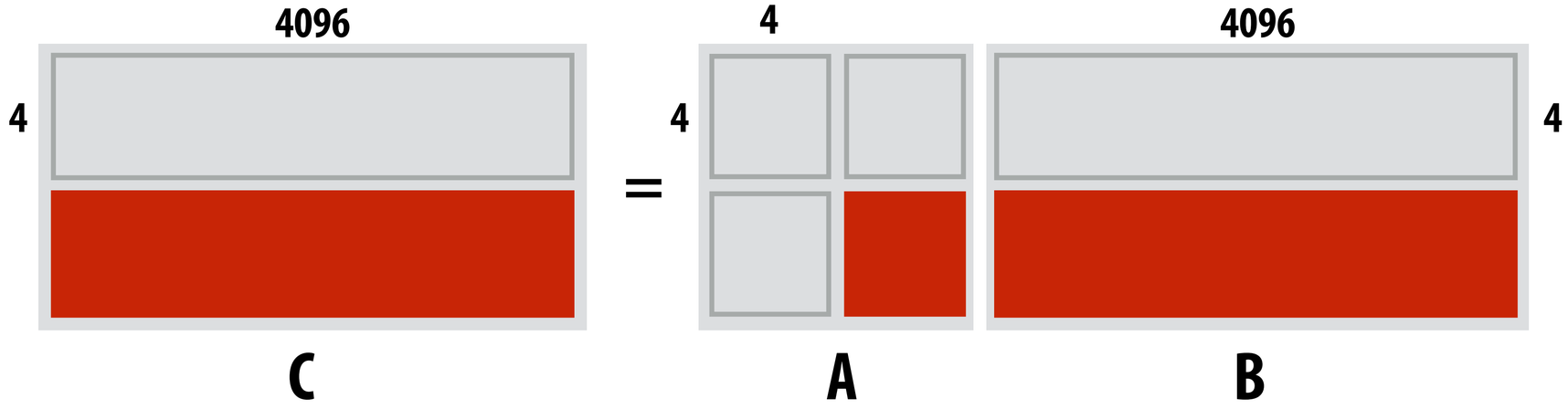
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

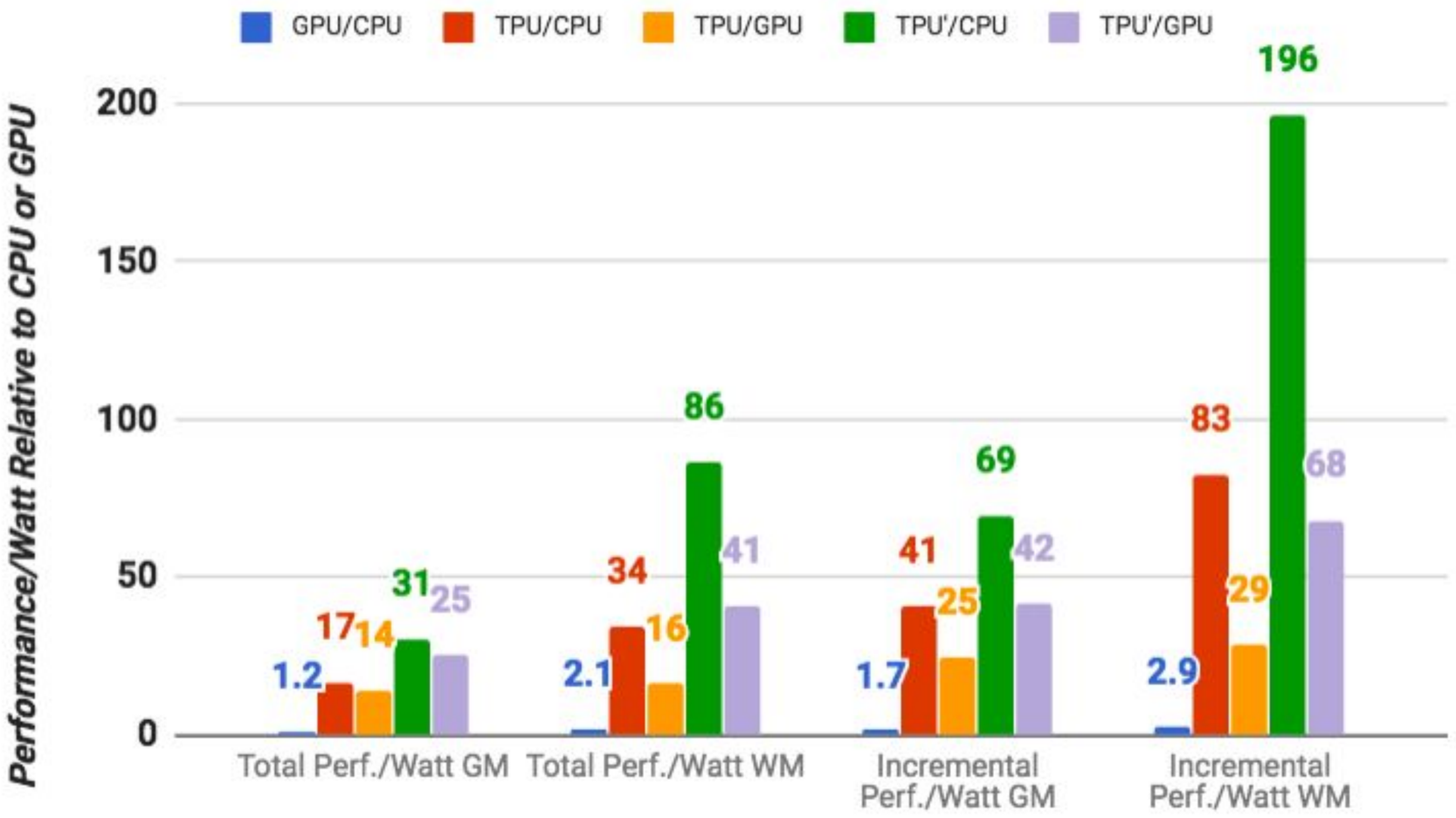
Building larger matrix-matrix multiplies

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



Assume 4096 accumulators

TPU Performance/Watt



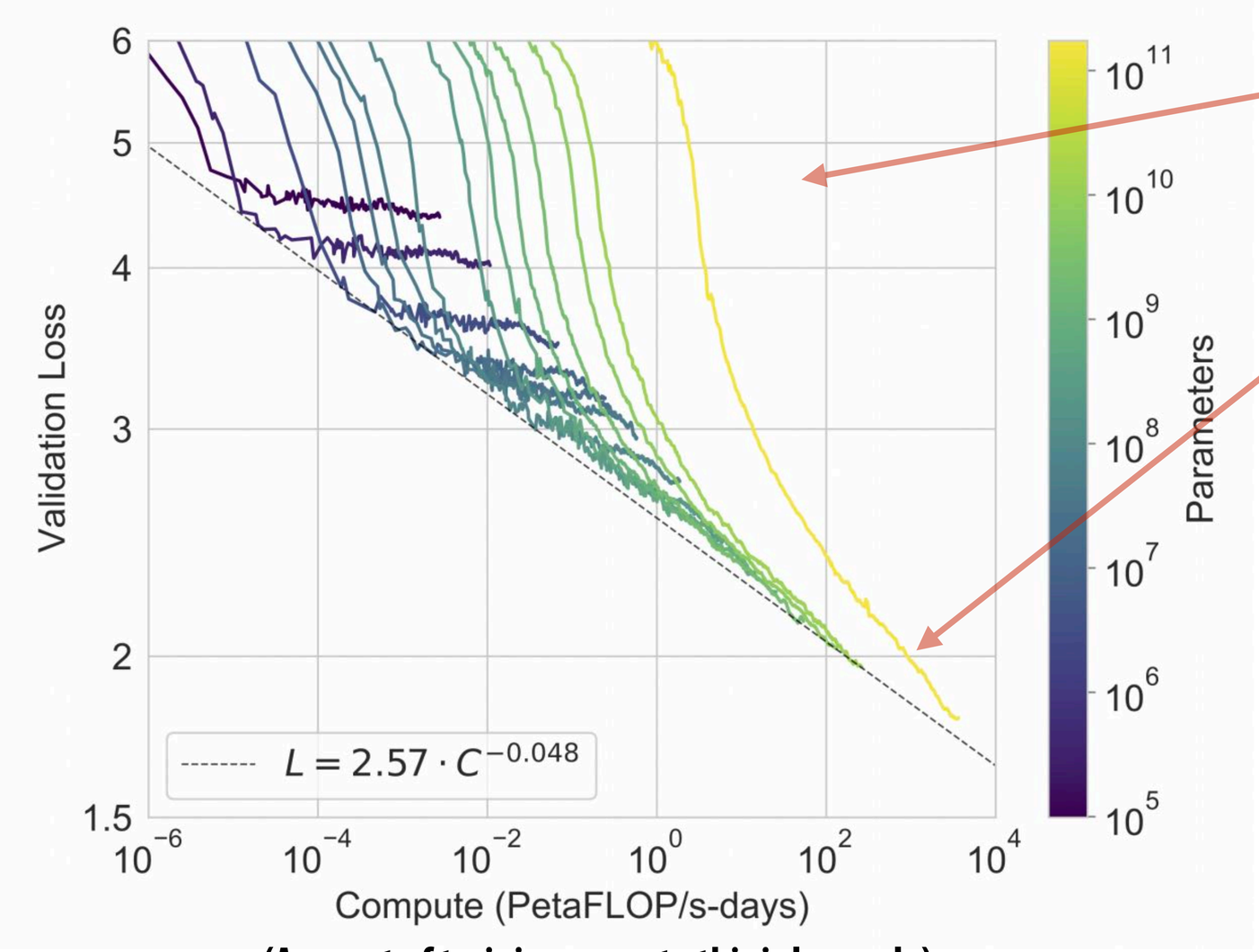
GM = geometric mean over all apps
WM = weighted mean over all apps

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

Figure credit: Jouppi et al. 2017

Scaling up (for training big models)

Example: GPT-3 language model



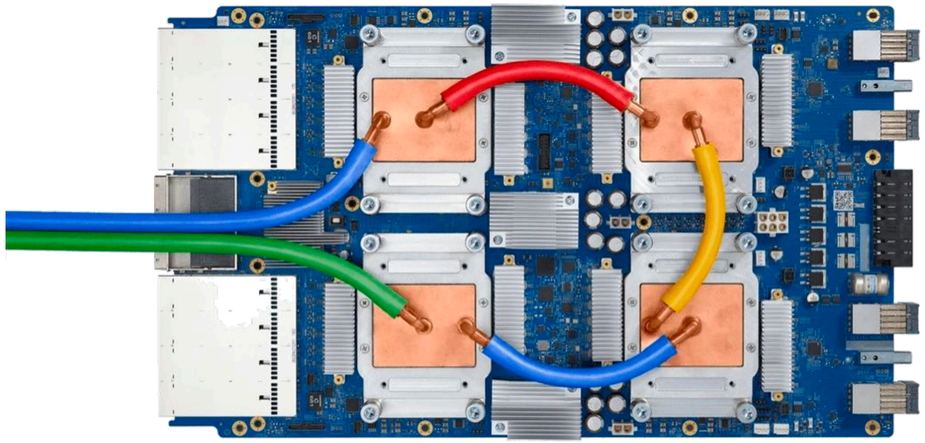
(Amount of training — note this is log scale)

Very big models +
More training
=
Better accuracy

Power law effect:
exponentially more compute to take constant step in accuracy

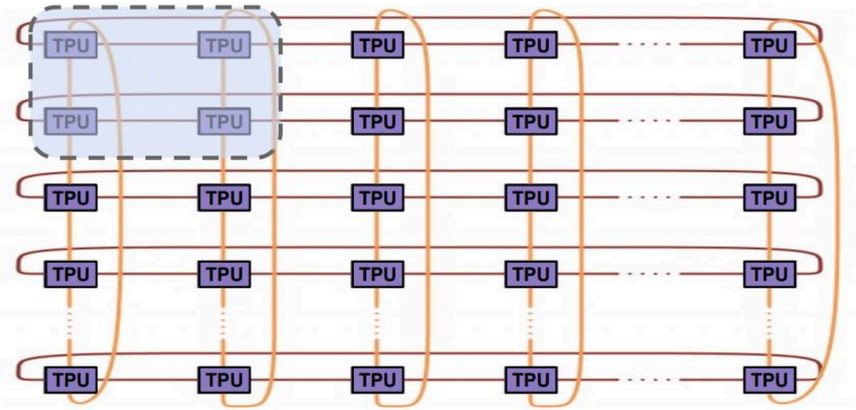
TPU v3 supercomputer

TPU v3 board
4 TPU3 chips

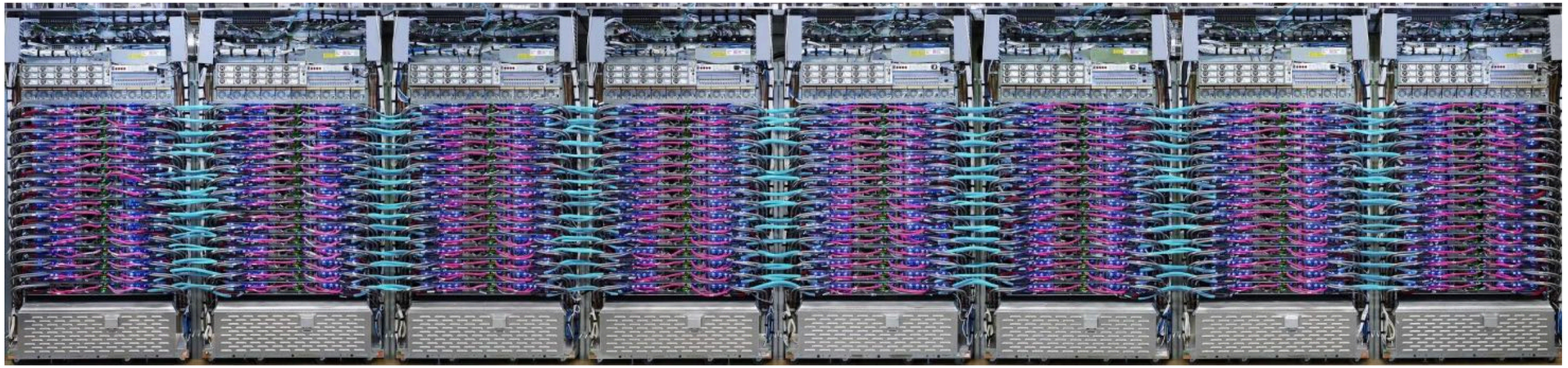


One TPU v3 board

TPUs connected by
2D Torus interconnect

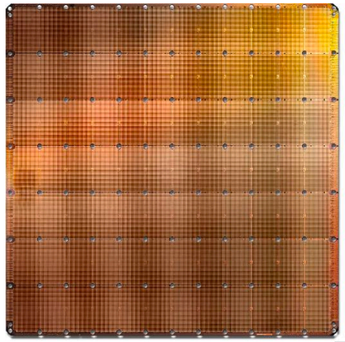


TPU supercomputer (1024 TPU v3 chips)

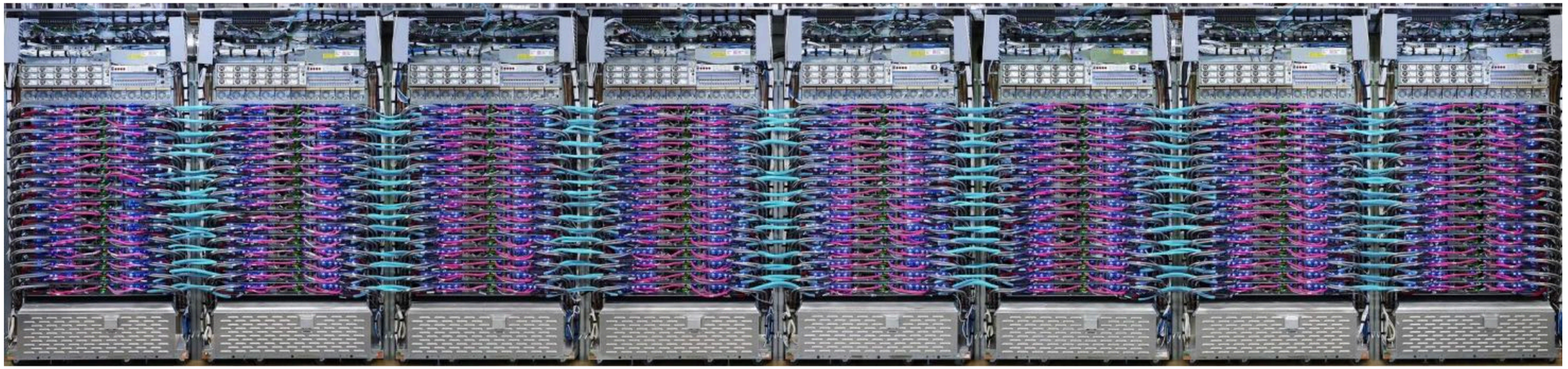


Summary: specialized hardware for DNN processing

- Specialized hardware for executing key DNN computations efficiently
- Feature many arithmetic units
- Customized/configurable datapaths to directly move intermediate data values between processing units (schedule computation by laying it out spatially on the chip)
- Large amounts of on-chip storage for fast access to intermediates



	Cerebras WSE
Chip size	46,225 mm^2
Cores	400,000
On chip memory	18 Gigabytes
Memory bandwidth	9 Petabytes/S
Fabric bandwidth	100 Petabits/S

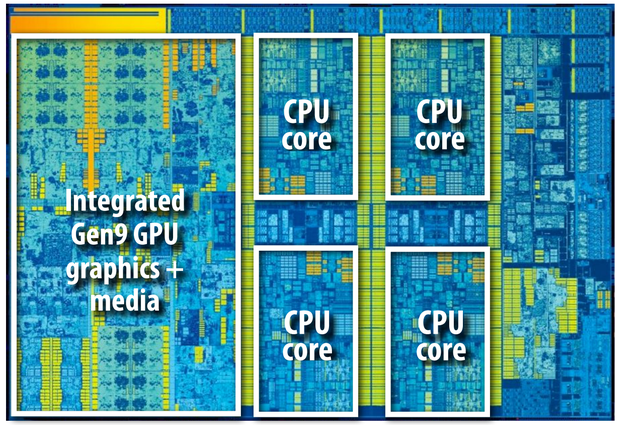


TPU supercomputer (1024 TPU v3 chips)

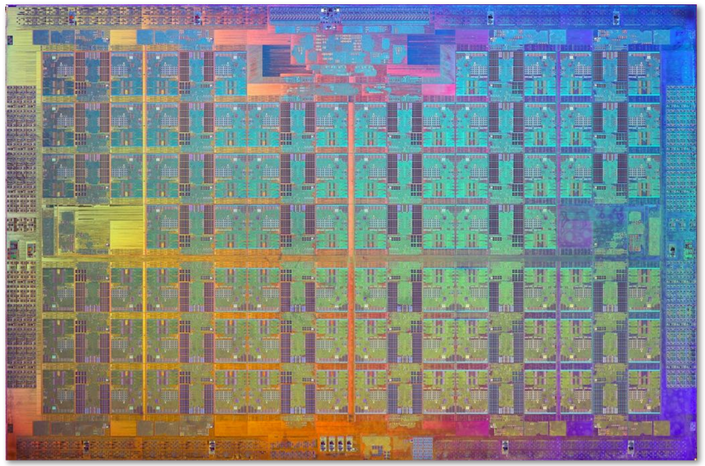
Course Wrap Up

(Students)

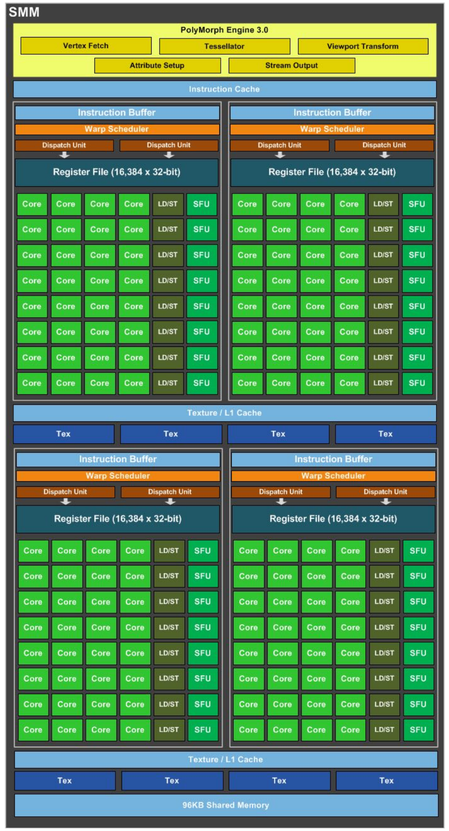
For the foreseeable future, the primary way to obtain higher performance computing hardware is through a combination of increased parallelism and hardware specialization.



Intel Core i7 CPU + integrated GPU and media

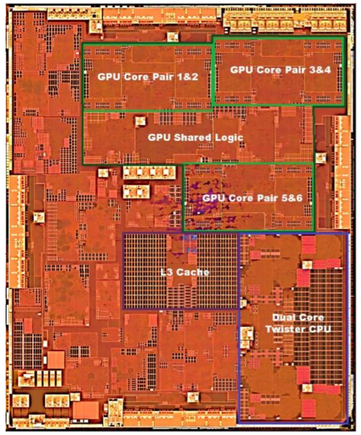


Intel Xeon Phi
72 cores, 16-wide SIMD, 4-way multi-threading



NVIDIA Maxwell GPU
(single SMM core)
32 wide SIMD
2048 CUDA/core threads per SMM

FPGA
(reconfigurable logic)



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

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

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

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

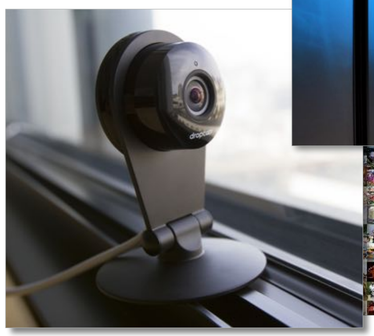
Given current software programming systems and tools, understanding how a parallel machine works is important to achieving high performance.

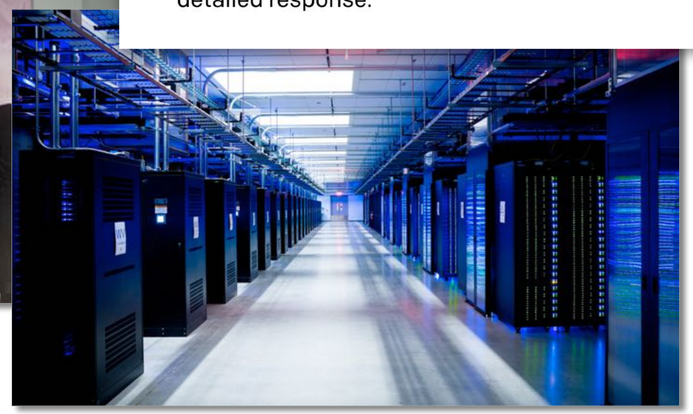
A major challenge going forward is making it simpler for programmers to extract performance on these complex machines.

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

DALL·E 2

DALL·E 2 is a new AI system that can create realistic images and art from a description in natural language.





ChatGPT: Optimizing Language Models for Dialogue

We've trained a model called ChatGPT which interacts in a conversational way. The dialogue format makes it possible for ChatGPT to answer followup questions, admit its mistakes, challenge incorrect premises, and reject inappropriate requests. ChatGPT is a sibling model to InstructGPT, which is trained to follow an instruction in a prompt and provide a detailed response.

Key issues we have addressed in this course

Identifying parallelism

(or conversely, identifying dependencies)

Efficiently scheduling parallelism

1. Achieving good workload balance

2. Overcoming communication constraints:

Bandwidth limits, dealing with latency, synchronization

Exploiting data/computation locality = efficiently managing state!

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

We discussed these issues at many scales and in many contexts

Heterogeneous mobile SoC

Single chip, multi-core CPU

Multi-core GPU

CPU+GPU connected via bus

Clusters of machines

Large scale, multi-node supercomputers

Key issues we have addressed in this course

Abstractions for thinking about efficient code

Data parallel thinking

Functional parallelism

Transactions

Tasks

How throughput-oriented hardware works

Multiple cores, hardware-threads, SIMD

Specialization to key domains

Two cool classes

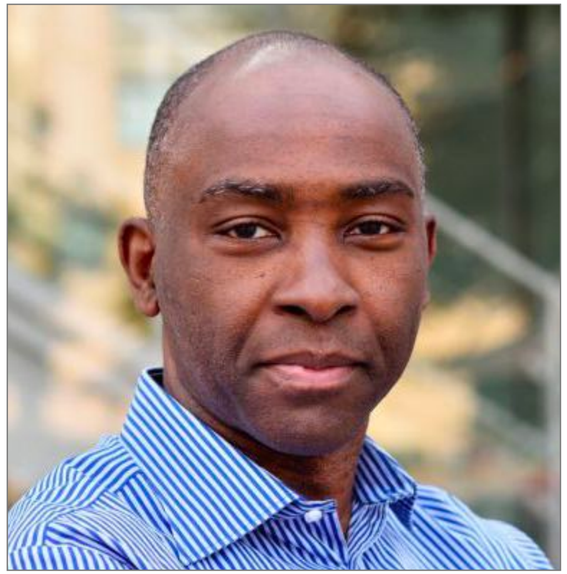
CS 217: Hardware Accelerators for Machine Learning (Winter, Kunle)

Focuses on design of specialized hardware architectures for ML (understanding the workload and building efficient hardware for that workload)

CS 348K: Visual Computing Systems (Spring, Kayvon)

Design of high-performance hardware/software systems for processing images and video (ray tracing, video analysis, smartphone camera processing, NeRF/ AI-based graphics, fast data labeling, etc)

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



Come talk to us!

Why research (or independent study)?

- Depth can be fun. You will learn way more about a topic than in any class.
- You think your undergrad/MS peers are amazingly smart? Come see our Ph.D. students! (you get to work side-by-side with them and with faculty). Imagine what level you might rise to.
- It's fun to be on the cutting edge. Industry might not even know about what you are working on. (imagine how much more valuable you are if you can teach them)
- It widens your mind as to what might be possible with tech.

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

- Designing a game engine to render frames at 10,000 fps per GPU to rapidly create training data for reinforcement learning
- Parallel ray tracing using 1000's of CPU cores in the cloud
- Designing more efficient DNN architectures for image and video processing
- New applications of analyzing video data at scale
  - Learning video game characters that move and play like real athletes (virtual Roger Federer) from TV broadcast video

HYPOTHESIS:

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

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

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

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

A conventional path…

CS student

DOES NOT SLEEP in order to do well in MANY CS classes

Even more impressive resume handed out at CS job fair → Resume gets student first-round interview → Student knows their stuff in interview (aces fine-grained linked list locking question) → **GOOD JOB** Woot!

An alternative path…

Amazing CS student

Takes fewer classes, but does some crazy extra credits in CS149. (really interested in parallel programming)

Student: "Hey Kayvon, I liked your class, is there anything I can help with in your research group next semester?"

Kayvon: "Yo! You did great in the class. I loved that extra credit you did. You should totally come help with this project in my group."

Student gets awesome experience working side-by-side with Stanford Ph.D. students and professors. Learns way more than in class.

Kayvon, to friend in industry: "Hey, you've got to hire this kid, they know more about parallel architecture than any undergrad in the country. They've been doing publishable research on it."

WICKED GOOD JOB

Woot!

Think bigger + broader

You are fortunate.

You are smart, talented, and hard-working.

You are in an amazing environment at Stanford.

How can you maximize that opportunity while you are here?
The mechanisms are in place (or we'll help you create them):

Course projects
Research
Independent study
Entrepreneurship

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

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

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

* taken from colleague Dave Eckhardt

Thanks for being a great class!
Thanks for putting in the work. (in the face of stressful times)
Stay healthy!

Have a great break!