

Lecture 6:

Efficient DNN Inference

**Visual Computing Systems
Stanford CS348K, Spring 2024**

Today

- **Key issues optimizing the efficiency in DNN inference**
- **Today: why matrix-matrix multiply is such a key building block**
 - **Implementation in software vs. accelerator hardware**
 - **Specialization trade-offs**
 - **[Next time] why low precision is so important**
- **Last 10 minutes of class:**
 - **Conversation about projects**

Efficiency challenge

Many DNN topologies
(Many variants on common backbones)

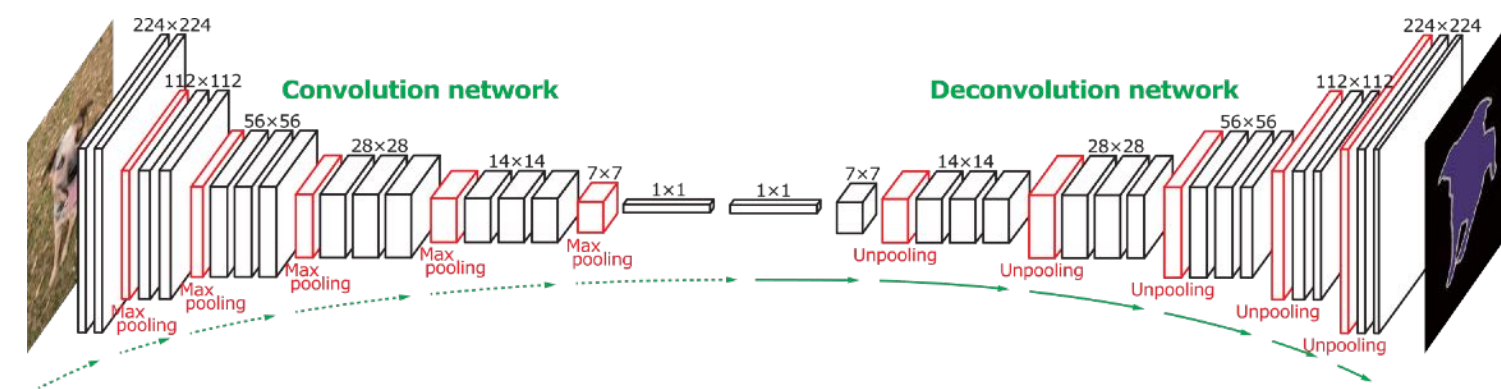
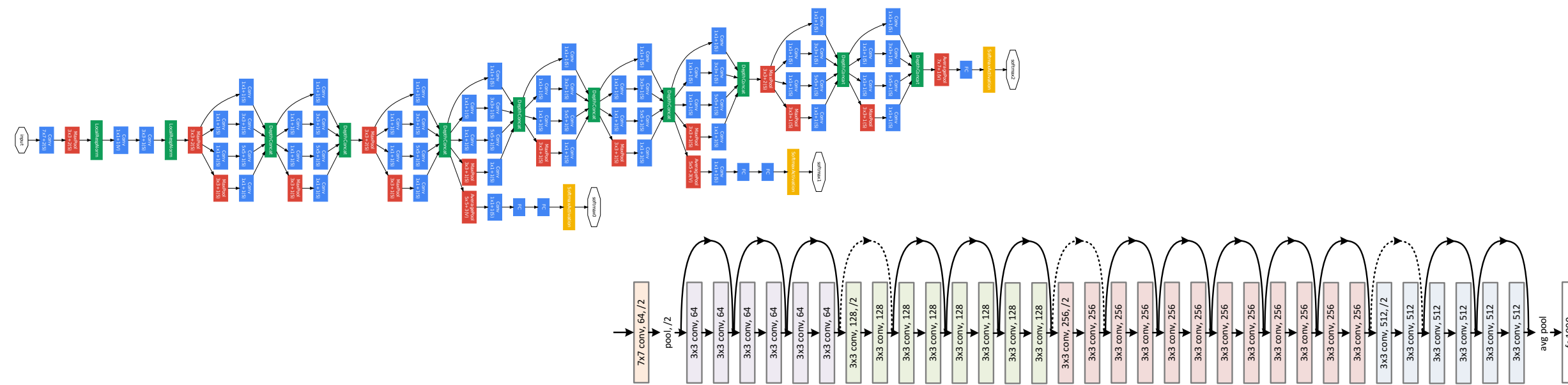


Table 1. MobileNet Body Architecture

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

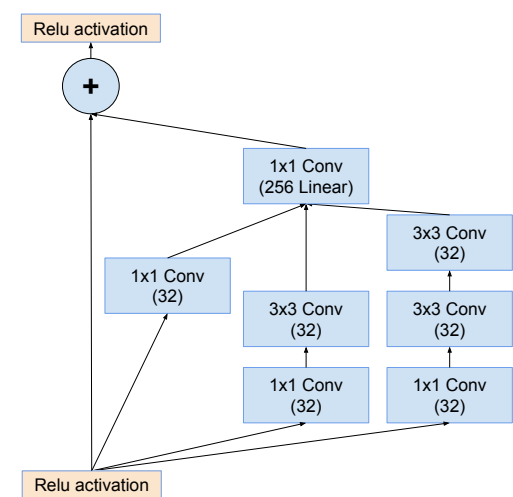
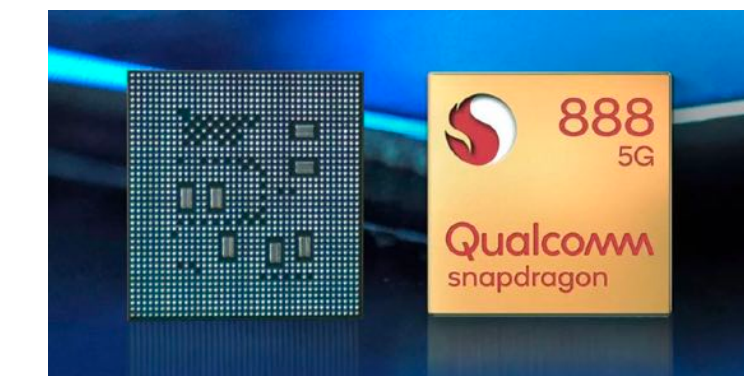


Figure 10. The schema for 35×35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

Many Target Devices



3x3 convolution (again)

```
float input[WIDTH+2][HEIGHT+2];
float output[WIDTH][HEIGHT];

float weights[3][3] = {{1./9, 1./9, 1./9},
                       {1./9, 1./9, 1./9},
                       {1./9, 1./9, 1./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][i+ii] * weights[jj][ii];
        output[j][i] = tmp;
    }
}
```

**Total work per output image =
9 x WIDTH x HEIGHT**

For NxN filter: N^2 x WIDTH x HEIGHT

Performing many 3x3 convolutions

```
float input[WIDTH+2][HEIGHT+2];
float output[NUM_FILTERS][WIDTH][HEIGHT];

float weights[NUM_FILTERS][3][3] = { {{1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}}, ...

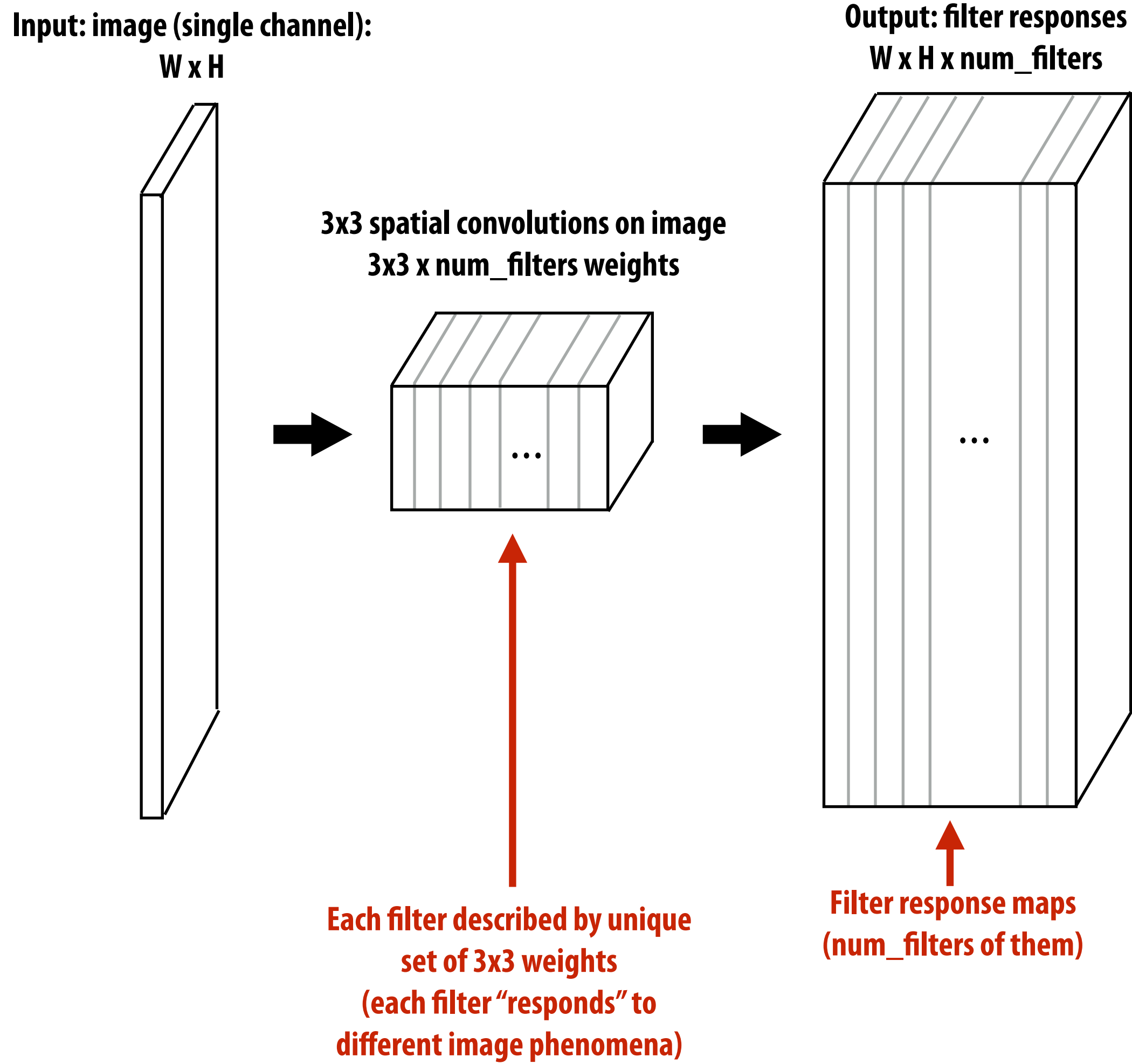
for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT; j++) {
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int jj=0; jj<3; jj++)
                for (int ii=0; ii<3; ii++)
                    tmp += input[j+jj][i+ii] * weights[k][jj][ii];
            output[k][j][i]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

**Total work per output image =
9 x WIDTH x HEIGHT**

For NxN filter: N² x WIDTH x HEIGHT

**Total work =
NUM_FILTERS x 9 x WIDTH x HEIGHT**

Applying many filters to an image at once



Performing many 3x3 convolutions, then a reLU

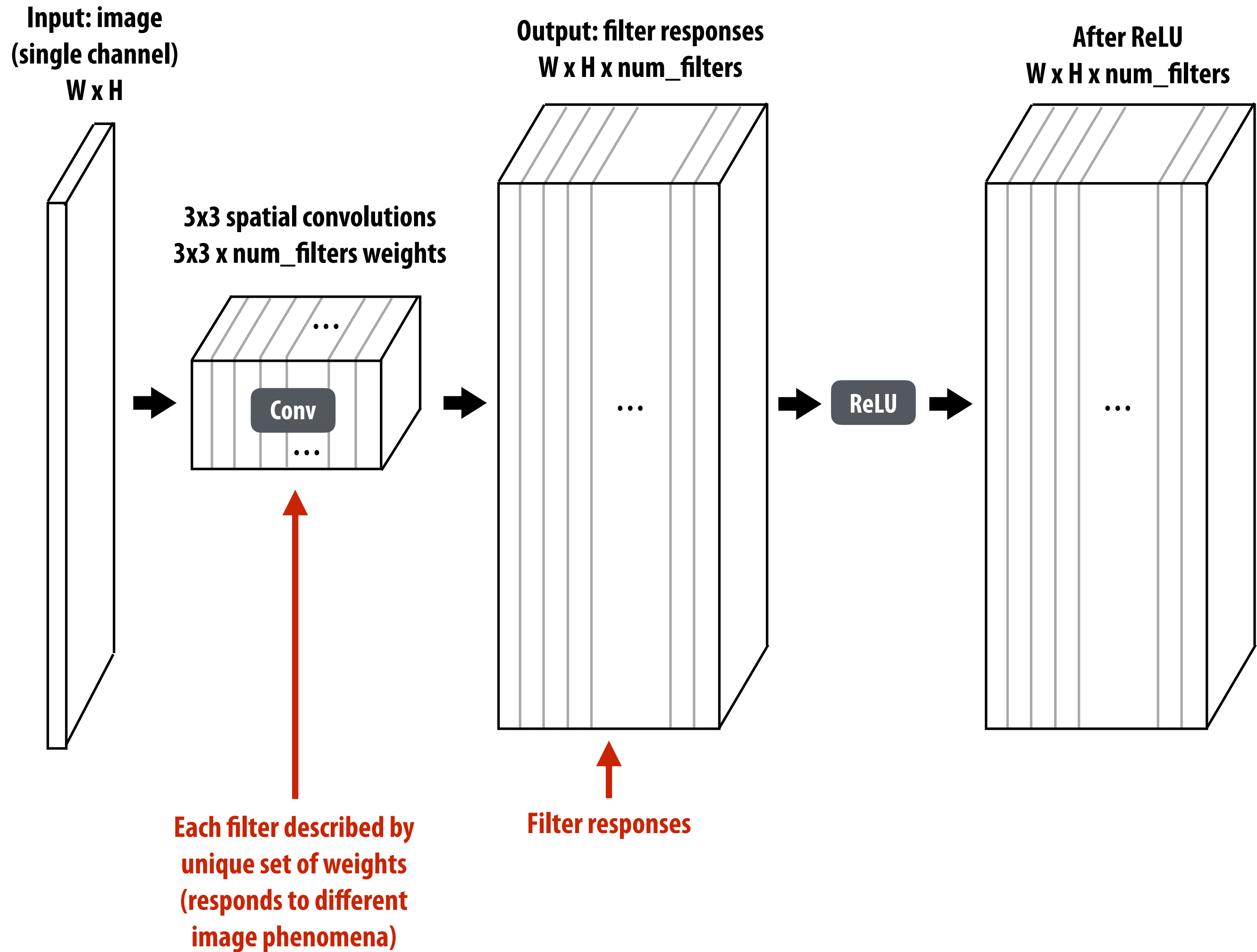
```
float input[WIDTH+2][HEIGHT+2];
float output[NUM_FILTERS][WIDTH][HEIGHT];
float output2[NUM_FILTERS][WIDTH][HEIGHT];

float weights[NUM_FILTERS][3][3] = { {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}}, ...

for (int k=0; k<NUM_FILTERS; k++) {
    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][i+ii] * weights[k][jj][ii];
            output[k][j*WIDTH + i] = tmp;
        }
    }
}

for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT; j++) {
        for (int i=0; i<WIDTH; i++) {
            output2[k][j][i] = output[k][j][i] < 0 ? 0 : output[k][j][i];
        }
    }
}
```

Adding additional layers



Performing many 3x3 convolutions, then a reLU, then a max pool

```
float input[WIDTH+2][HEIGHT+2];
float output[NUM_FILTERS][WIDTH][HEIGHT];
float output2[NUM_FILTERS][WIDTH][HEIGHT];
float output2[NUM_FILTERS][WIDTH/2][HEIGHT/2];
```

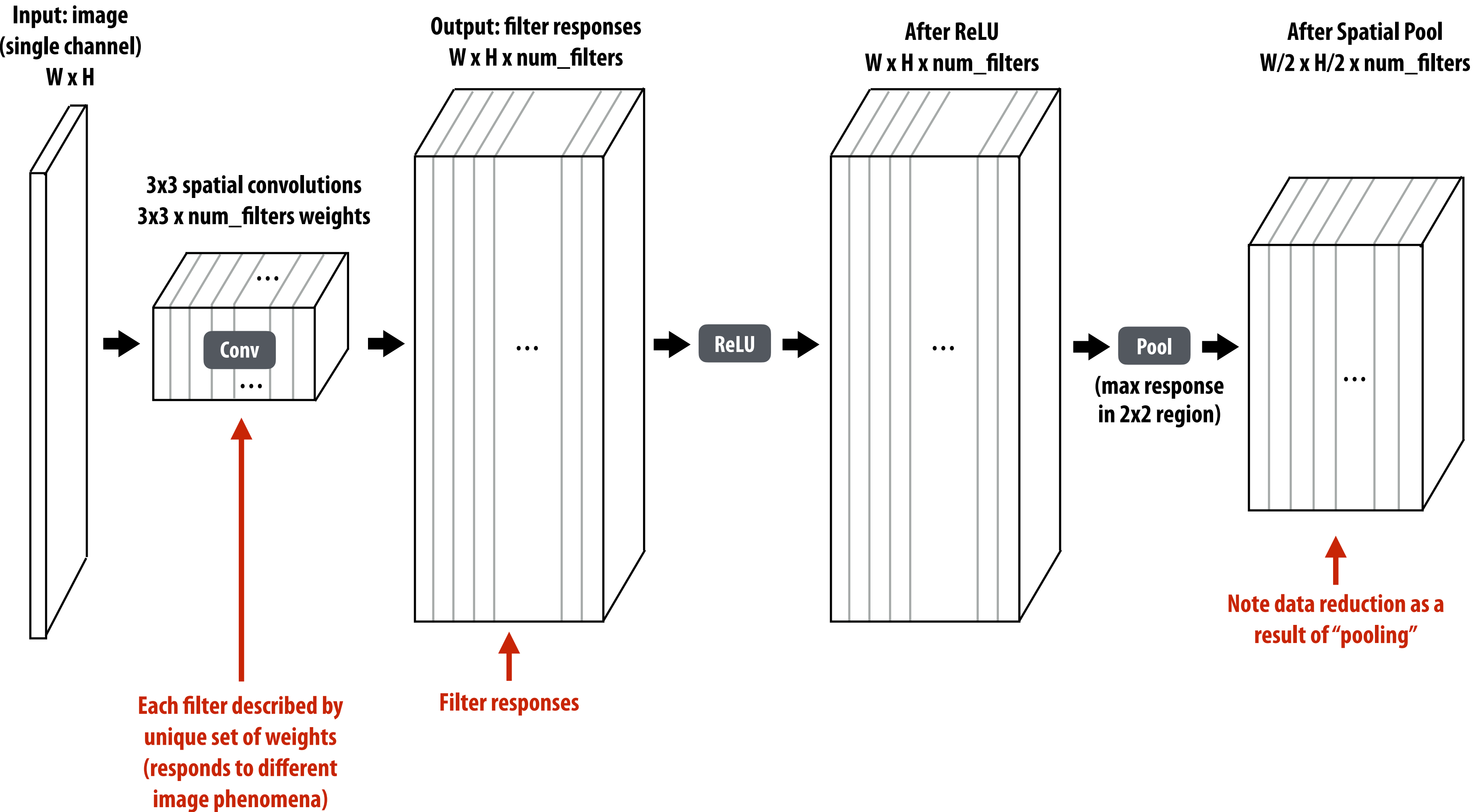
```
float weights[NUM_FILTERS][3][3] = { {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}}, ...
```

```
for (int k=0; k<NUM_FILTERS; k++) {
    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][i+ii] * weights[k][jj][ii];
            output[k][j*WIDTH + i] = tmp;
        }
    }
}
```

```
for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT; j++) {
        for (int i=0; i<WIDTH; i++) {
            output2[k][j][i] = output[k][j][i] < 0 ? 0 : output[k][j][i];
        }
    }
}
```

```
for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT/2; j++) {
        for (int i=0; I<WIDTH/2; i++) {
            output3[k][j][i] = max(output2[k][2*j][2*I],
                                     output2[k][2*j][2*i+1],
                                     output2[k][2*j+1][2*i],
                                     output2[k][2*j+1][2*I+1]);
        }
    }
}
```

Adding additional layers



Performing many 3x3 convolutions, then a reLU, on multi-channel input (not showing max pool)

```
float input[WIDTH+2][HEIGHT+2][NUM_CHANNELS];
float output[NUM_FILTERS][WIDTH][HEIGHT];
float output2[NUM_FILTERS][WIDTH][HEIGHT];

float weights[NUM_FILTERS][3][3][NUM_CHANNELS] = { {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}, {1./9, 1./9, 1./9}}, ...

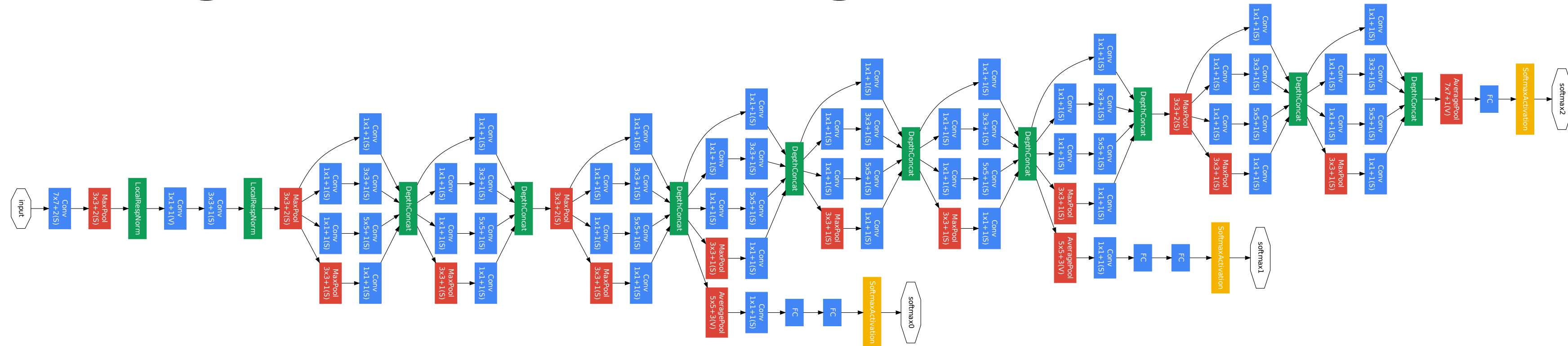
for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT; j++) {
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int jj=0; jj<3; jj++)
                for (int ii=0; ii<3; ii++)
                    for (int cc=0; cc<NUM_CHANNELS; cc++)
                      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

for (int k=0; k<NUM_FILTERS; k++) {
    for (int j=0; j<HEIGHT; j++) {
        for (int i=0; i<WIDTH; i++) {
            output2[k][j][i] = output[k][j][i] < 0 ? 0 : output[k][j][i];
        }
    }
}
```

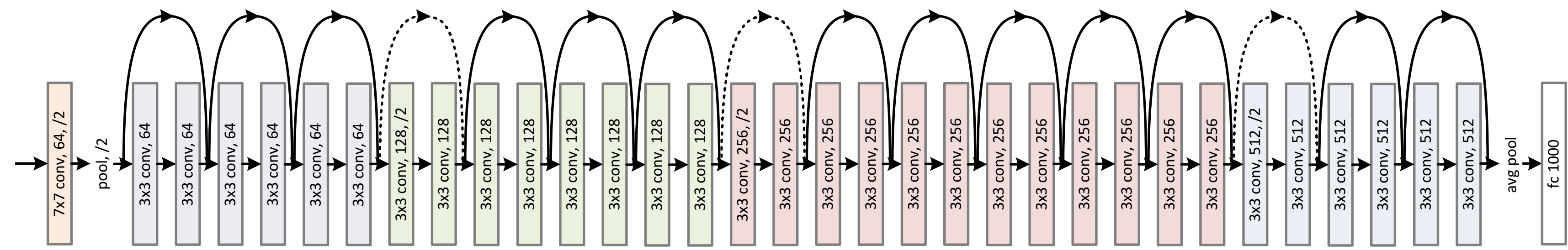
**Total work per output image =
9 x NUM_CHANNELS x WIDTH x HEIGHT**

**Total work =
NUM_FILTERS x 9 x NUM_CHANNELS x WIDTH x HEIGHT**

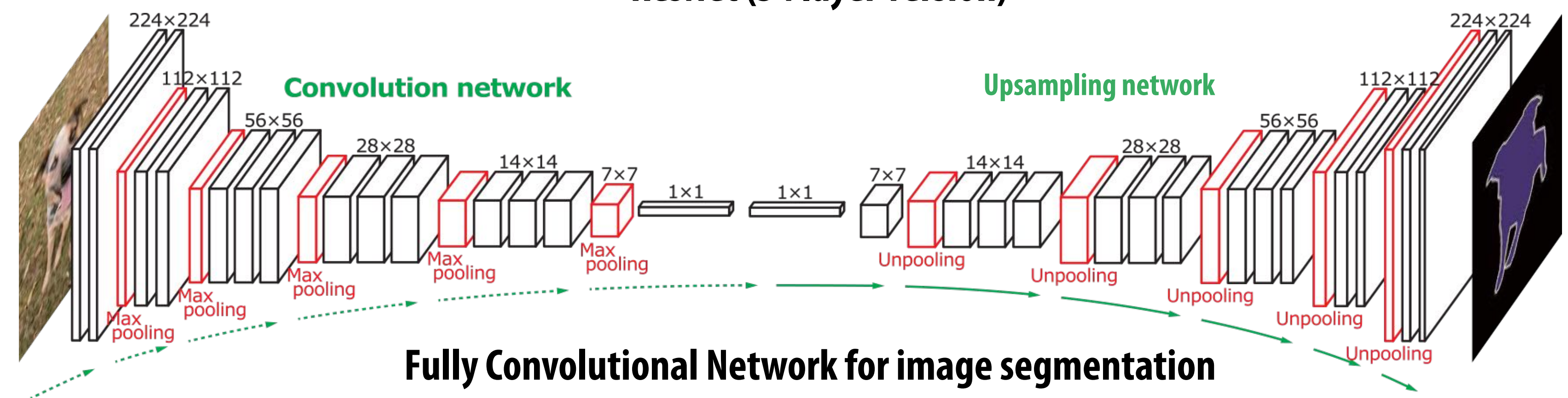
Common image understanding networks



Inception (GoogleLeNet)



ResNet (34 layer version)



Fully Convolutional Network for image segmentation

Efficiently implementing convolution layers

Direct implementation of conv layer (batched)

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

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

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

Convolutional layer in Halide

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

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

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

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

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

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

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

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

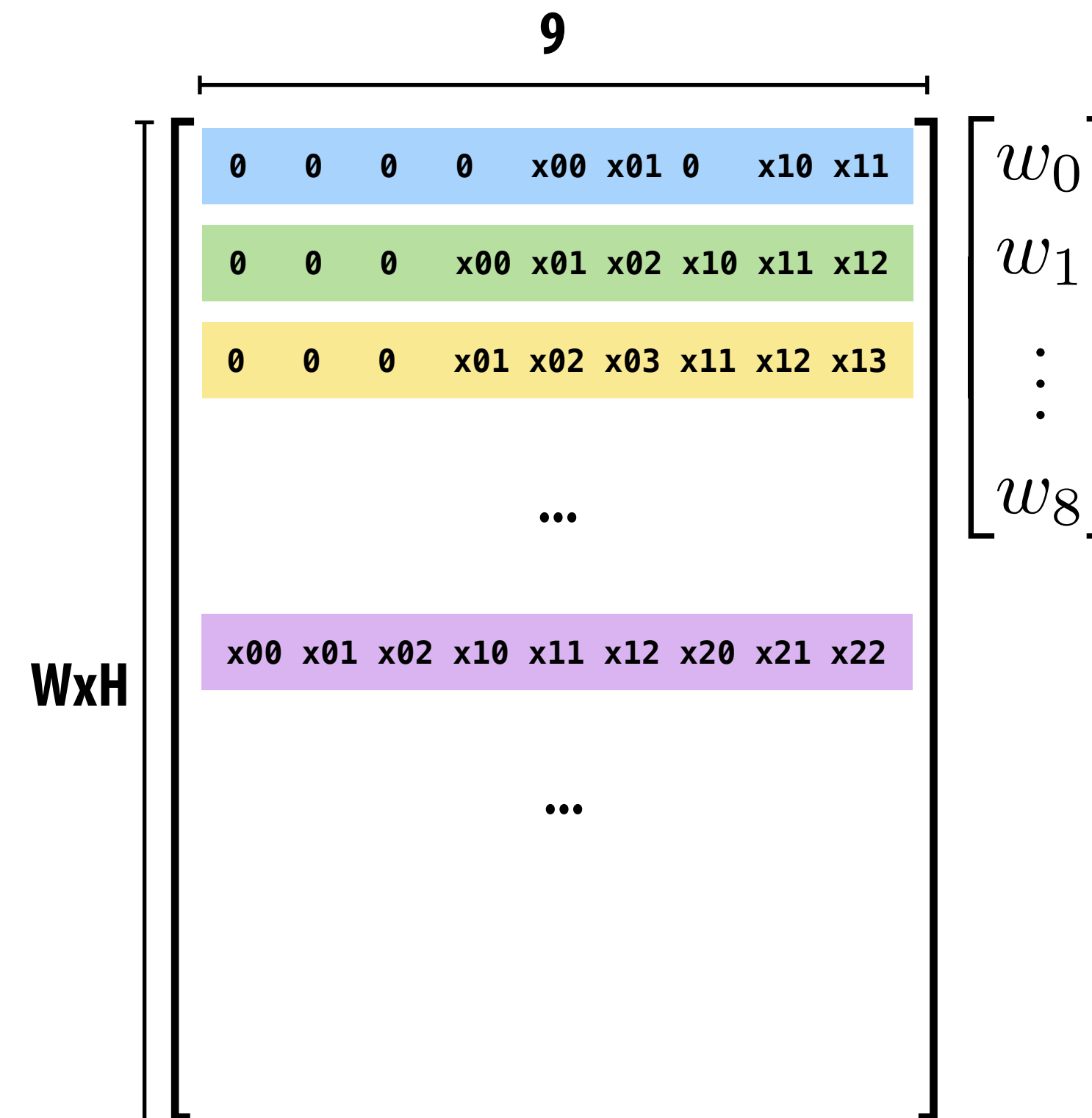
Consider scheduling this seven-dimensional loop nest!

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

Construct matrix from elements of input image

	x_{00}	x_{01}	x_{02}	x_{03}	...			
	x_{10}	x_{11}	x_{12}	x_{13}	...			
	x_{20}	x_{21}	x_{22}	x_{23}	...			
	x_{30}	x_{31}	x_{32}	x_{33}	...			
				

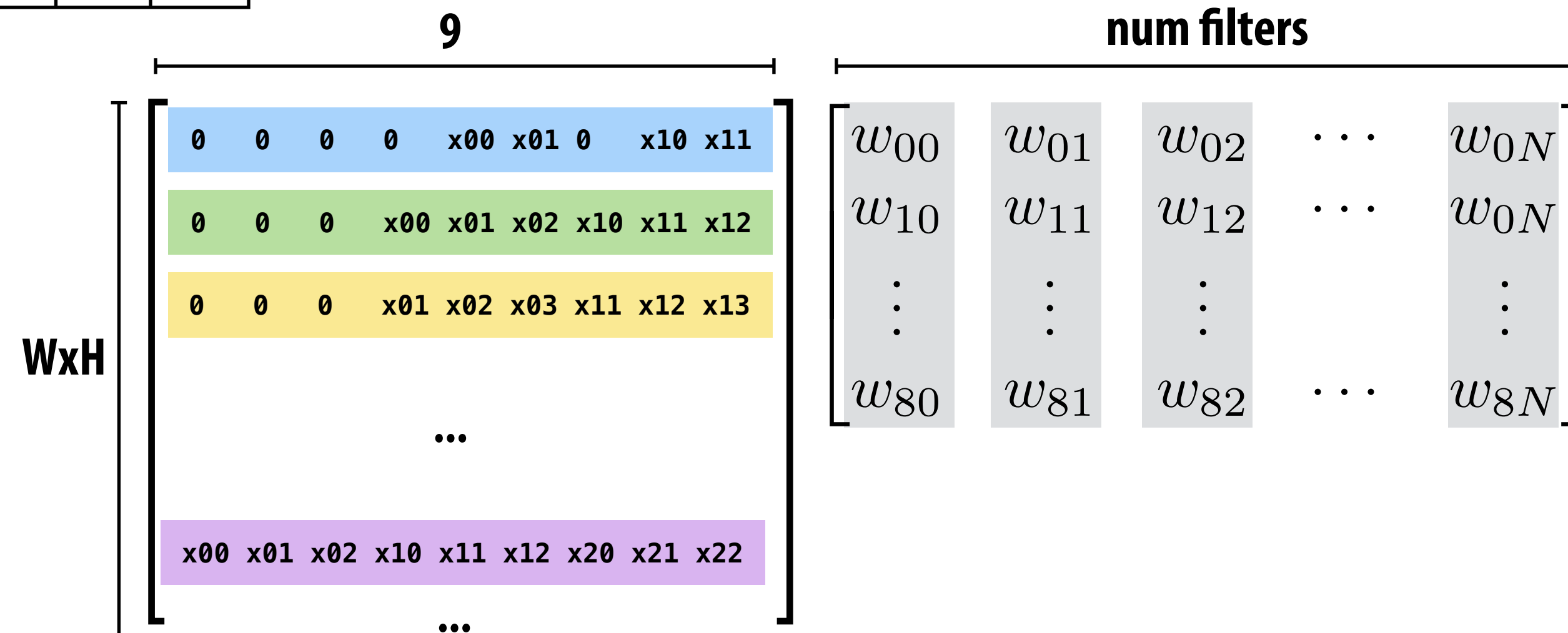
$O(N)$ storage overhead for filter with N elements
Must construct input data matrix



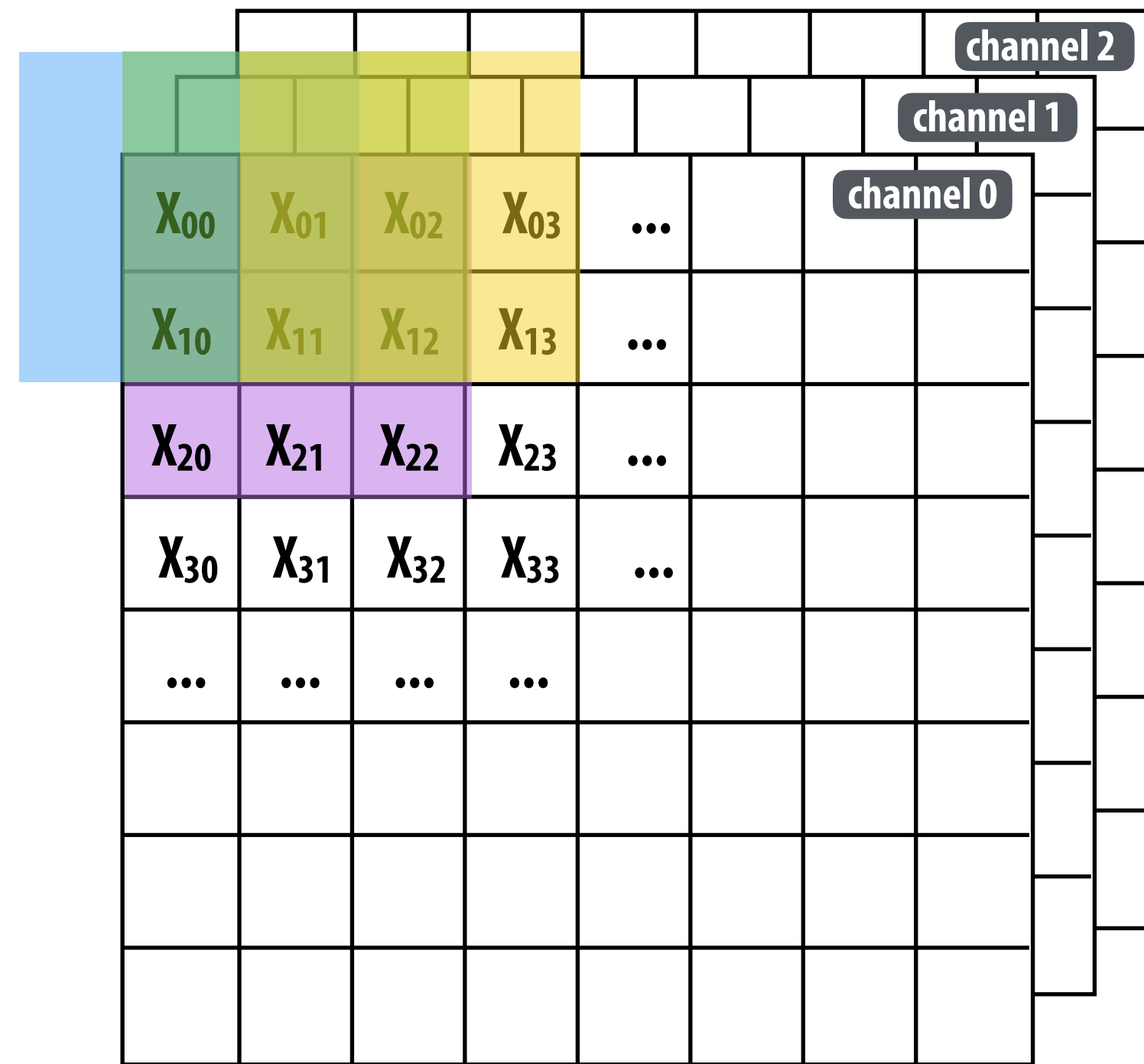
Note: 0-pad matrix

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

	x_{00}	x_{01}	x_{02}	x_{03}	...			
	x_{10}	x_{11}	x_{12}	x_{13}	...			
	x_{20}	x_{21}	x_{22}	x_{23}	...			
	x_{30}	x_{31}	x_{32}	x_{33}	...			
				

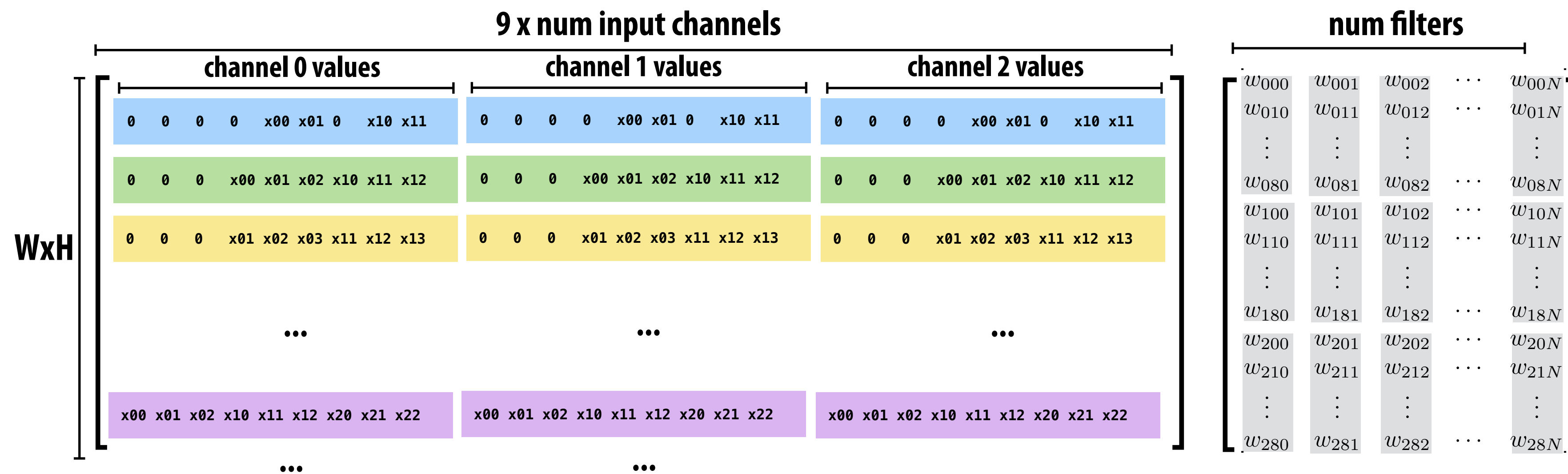


Multiple convolutions on multiple input channels



For each filter, sum responses over input channels

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



Conv layer to explicit GEMM mapping

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

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

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

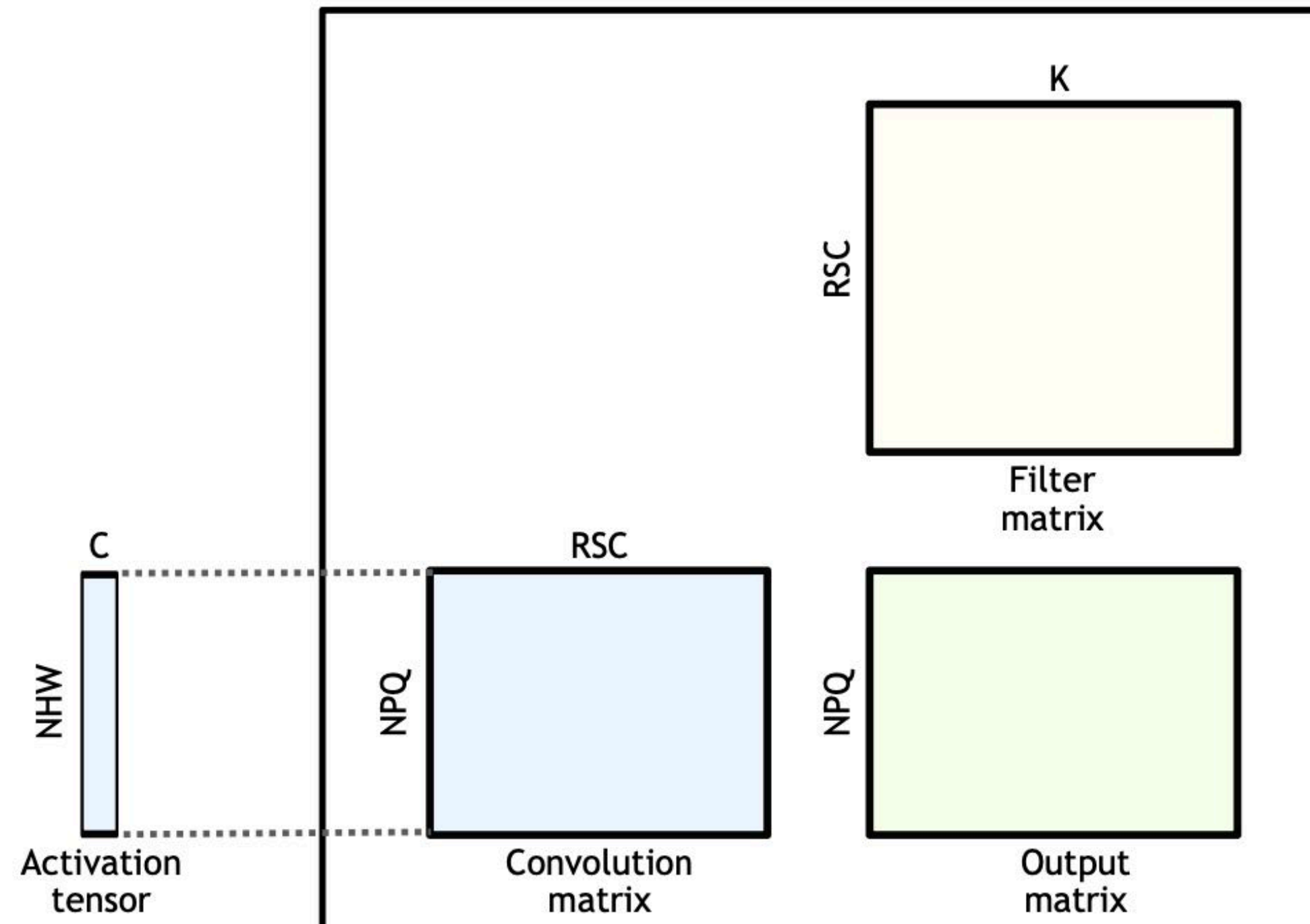


Image credit: NVIDIA

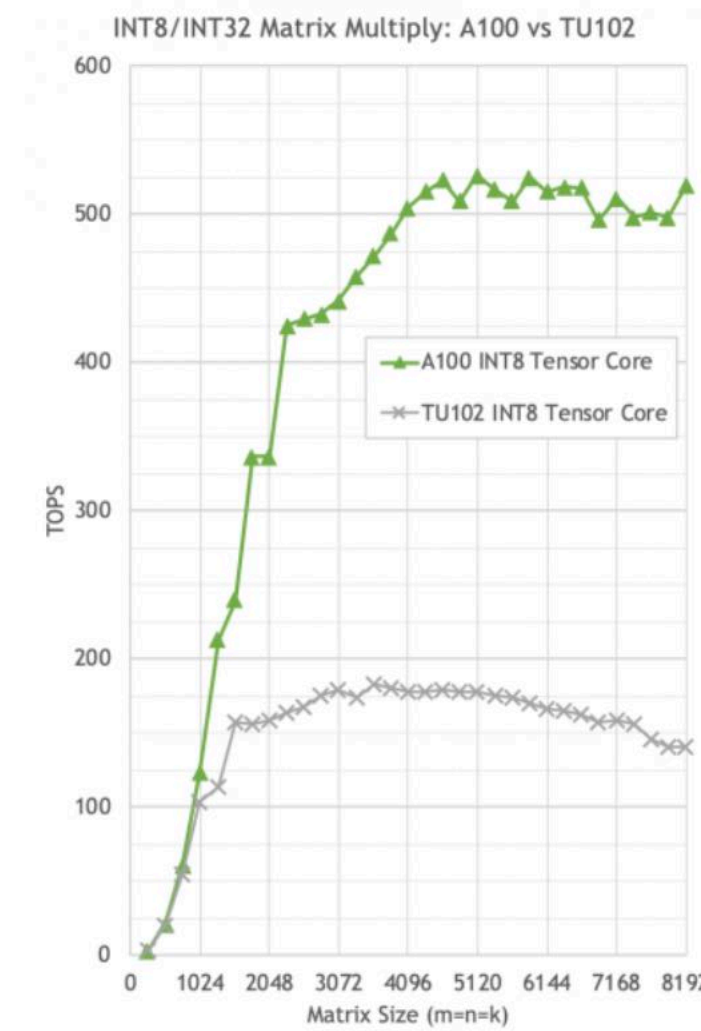
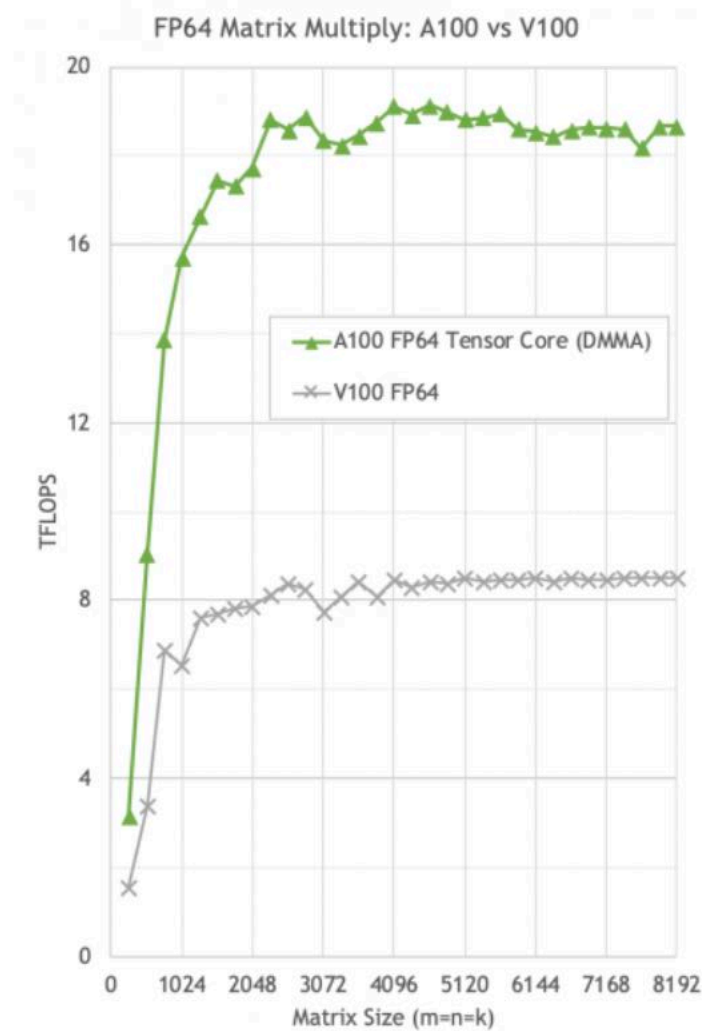
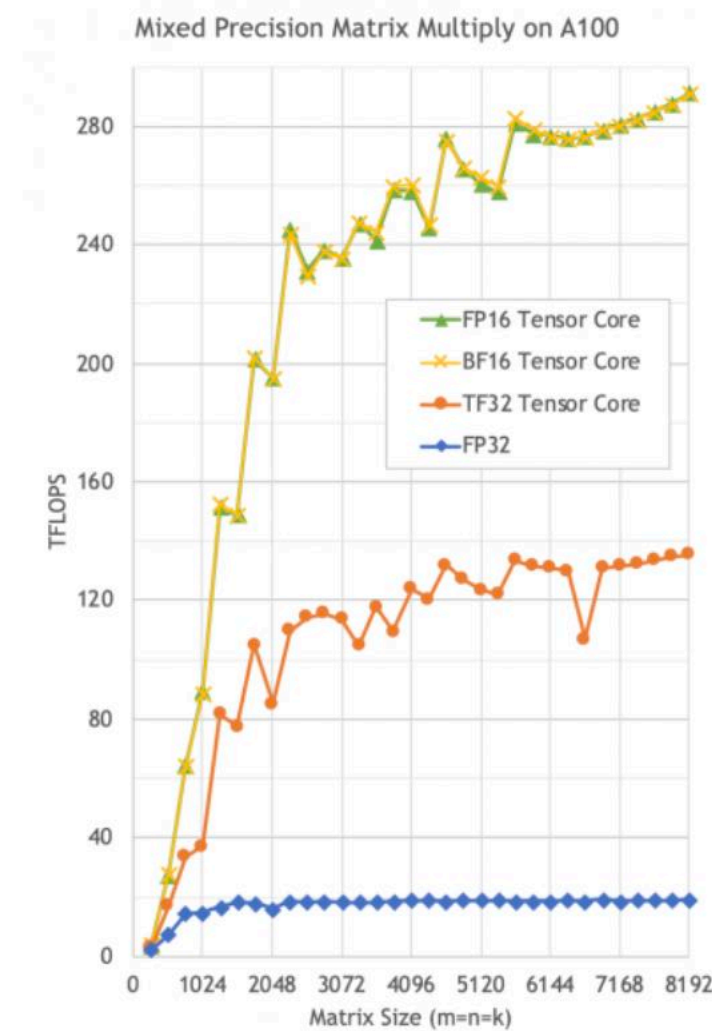
High performance implementations of GEMM exist

cuBLAS Performance

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

cuBLAS Key Features

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



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

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

Also requires large amount of aux storage

Intel® oneAPI Math Kernel Library

Intel®-Optimized Math Library for Numerical Computing

Optimized Library for Scientific Computing

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

Download as Part of the Toolkit

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

[Get It Now →](#)

Dense matrix multiplication

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

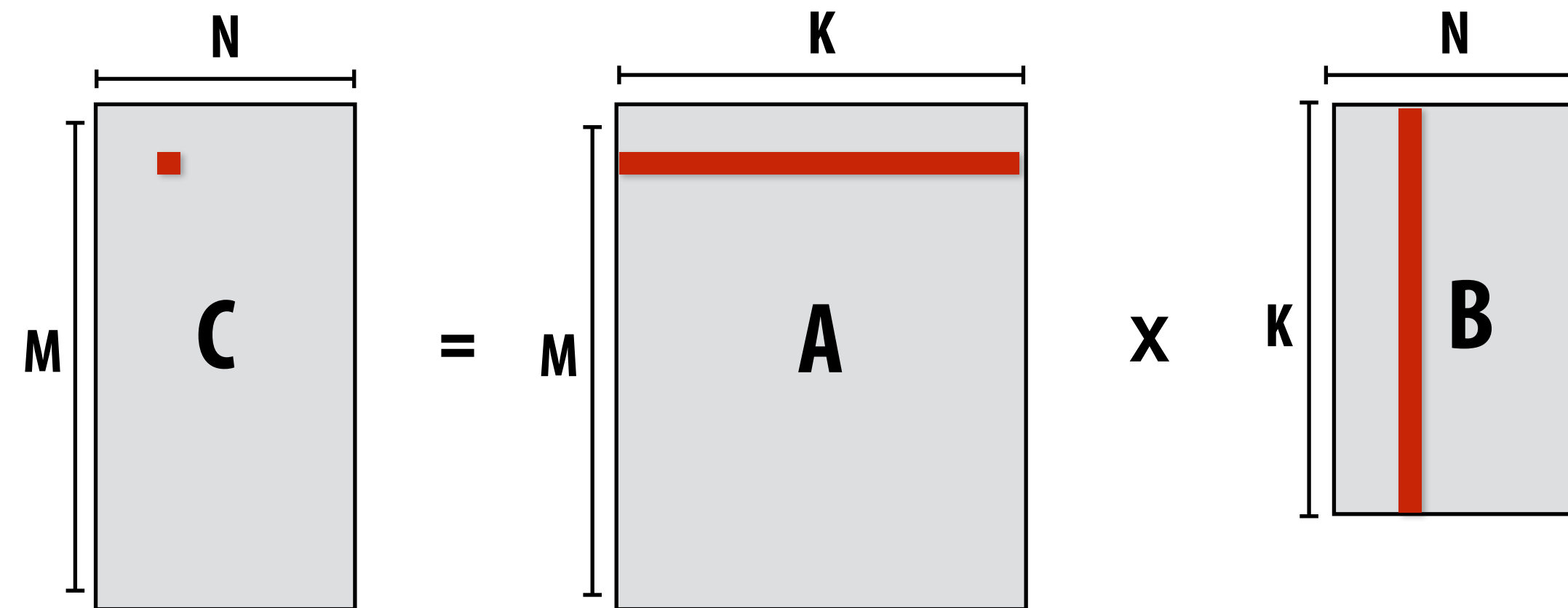
```
// compute C += A * B
```

```
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?

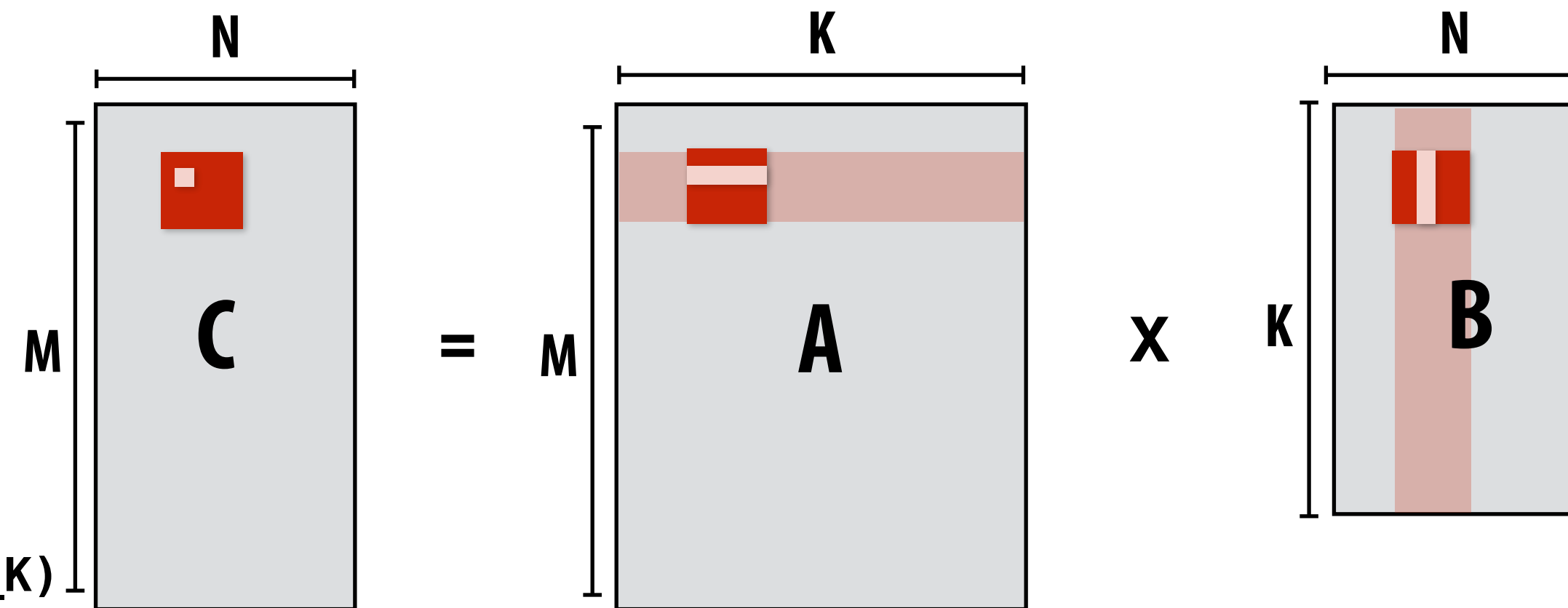
Low arithmetic intensity (does not exploit temporal locality in access to A and B)

Blocked dense matrix multiplication

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

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

```
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

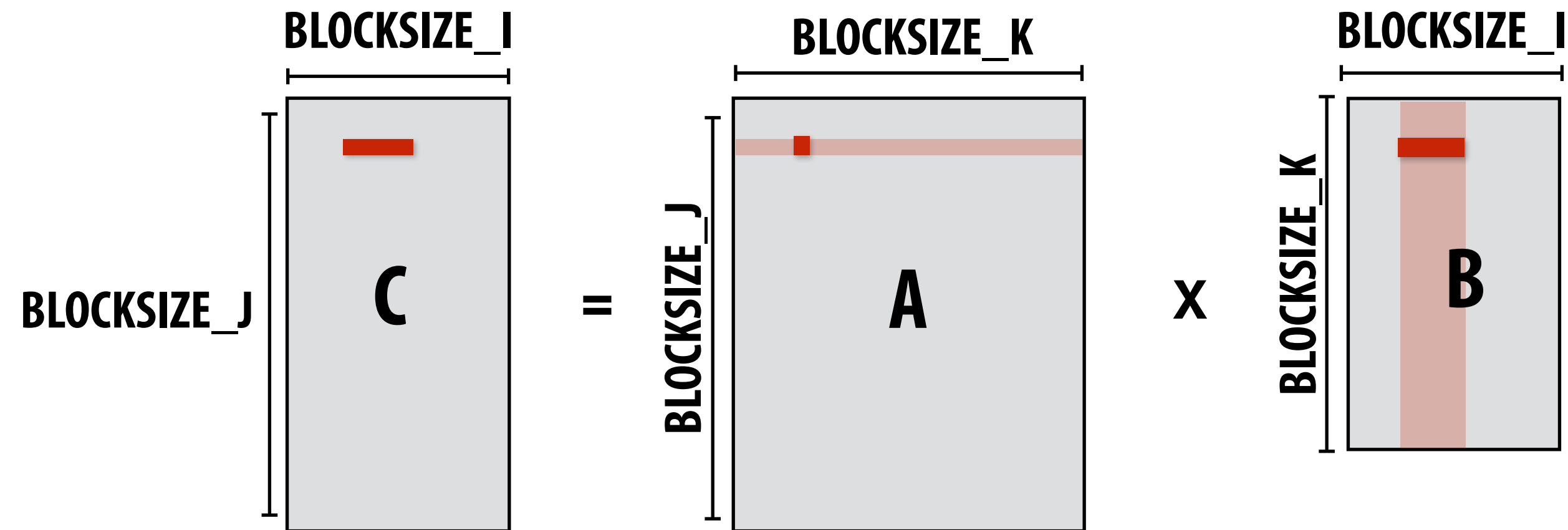
```
float A[M][K];
float B[K][N];
float C[M][N];

// compute C += A * B
for (int jblock2=0; jblock2<M; jblock2+=L2_BLOCKSIZE_J)
  for (int iblock2=0; iblock2<N; iblock2+=L2_BLOCKSIZE_I)
    for (int kblock2=0; kblock2<K; kblock2+=L2_BLOCKSIZE_K)
      for (int jblock1=0; jblock1<L1_BLOCKSIZE_J; jblock1+=L1_BLOCKSIZE_J)
        for (int iblock1=0; iblock1<L1_BLOCKSIZE_I; iblock1+=L1_BLOCKSIZE_I)
          for (int kblock1=0; kblock1<L1_BLOCKSIZE_K; kblock1+=L1_BLOCKSIZE_K)
            for (int j=0; j<BLOCKSIZE_J; j++)
              for (int i=0; i<BLOCKSIZE_I; i++)
                for (int k=0; k<BLOCKSIZE_K; k++)
                  ...
```

Not shown: final level of “blocking” for register locality...

Blocked dense matrix multiplication (1)

Consider SIMD parallelism within a block



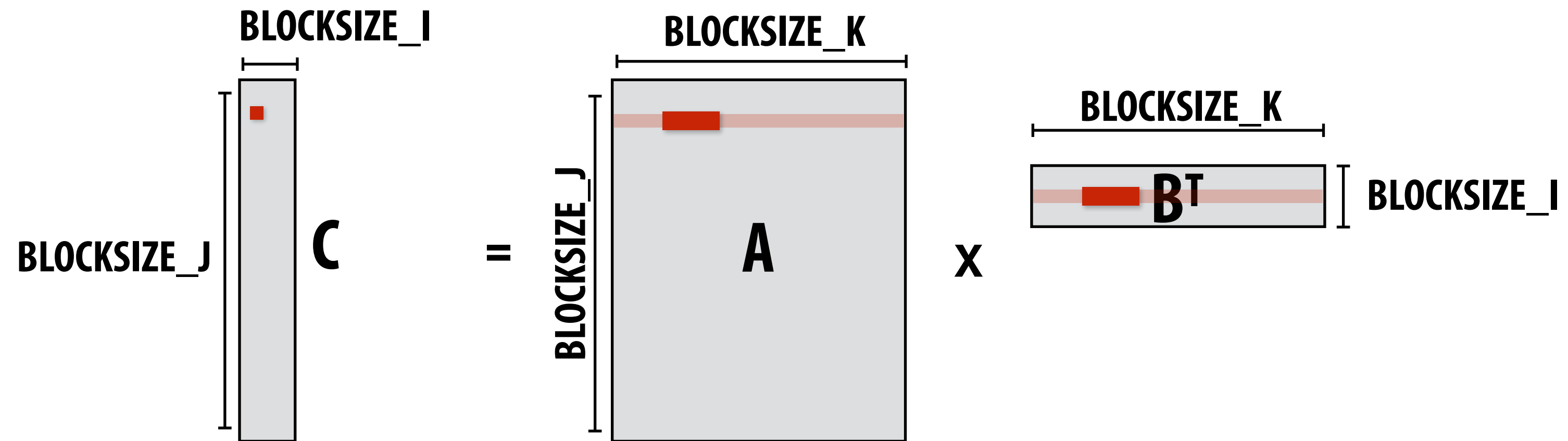
```
...
for (int j=0; j<BLOCKSIZE_J; j++) {
    for (int i=0; i<BLOCKSIZE_I; i+=SIMD_WIDTH) {
        simd_vec C_accum = vec_load(&C[jblock+j][iblock+i]);
        for (int k=0; k<BLOCKSIZE_K; k++) {
            // C = A*B + C
            simd_vec A_val = splat(&A[jblock+j][kblock+k]); // load a single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

Vectorize i loop

Good: also improves spatial locality in access to B

Bad: working set increased by SIMD_WIDTH, still walking over B in large steps

Blocked dense matrix multiplication (2)



...

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

Assume i dimension is small. Previous vectorization scheme (1) would not work well.

Pre-transpose block of B (copy block of B to temp buffer in transposed form)

Vectorize innermost loop

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

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

Ug for library implementers!

Table 1. MobileNet Body Architecture

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

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: $R \times S$

Input channels: C

Number of filters: K

Batch size: N

Image credit: NVIDIA

GEMM TRIPLE NEST LOOP

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

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

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

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

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

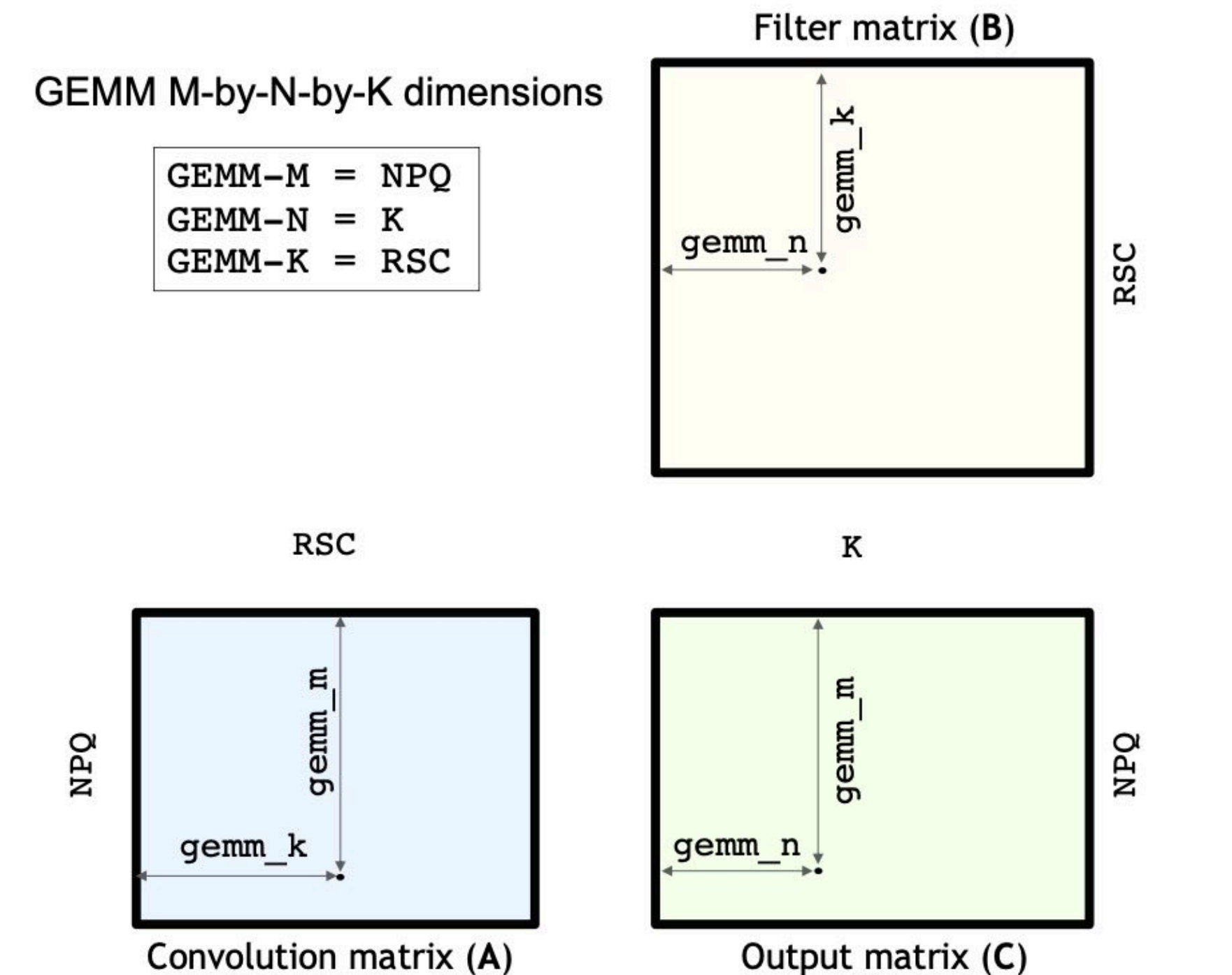
      int h = h_bar(p, r);
      int w = w_bar(q, s);

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

    }

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

CONVOLUTION MAPPED TO GEMM



$$C[gemm_m, gemm_n] = \sum_{gemm_k=0}^{RSC-1} (A[gemm_m, gemm_k] * B[gemm_k, gemm_n])$$

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"**

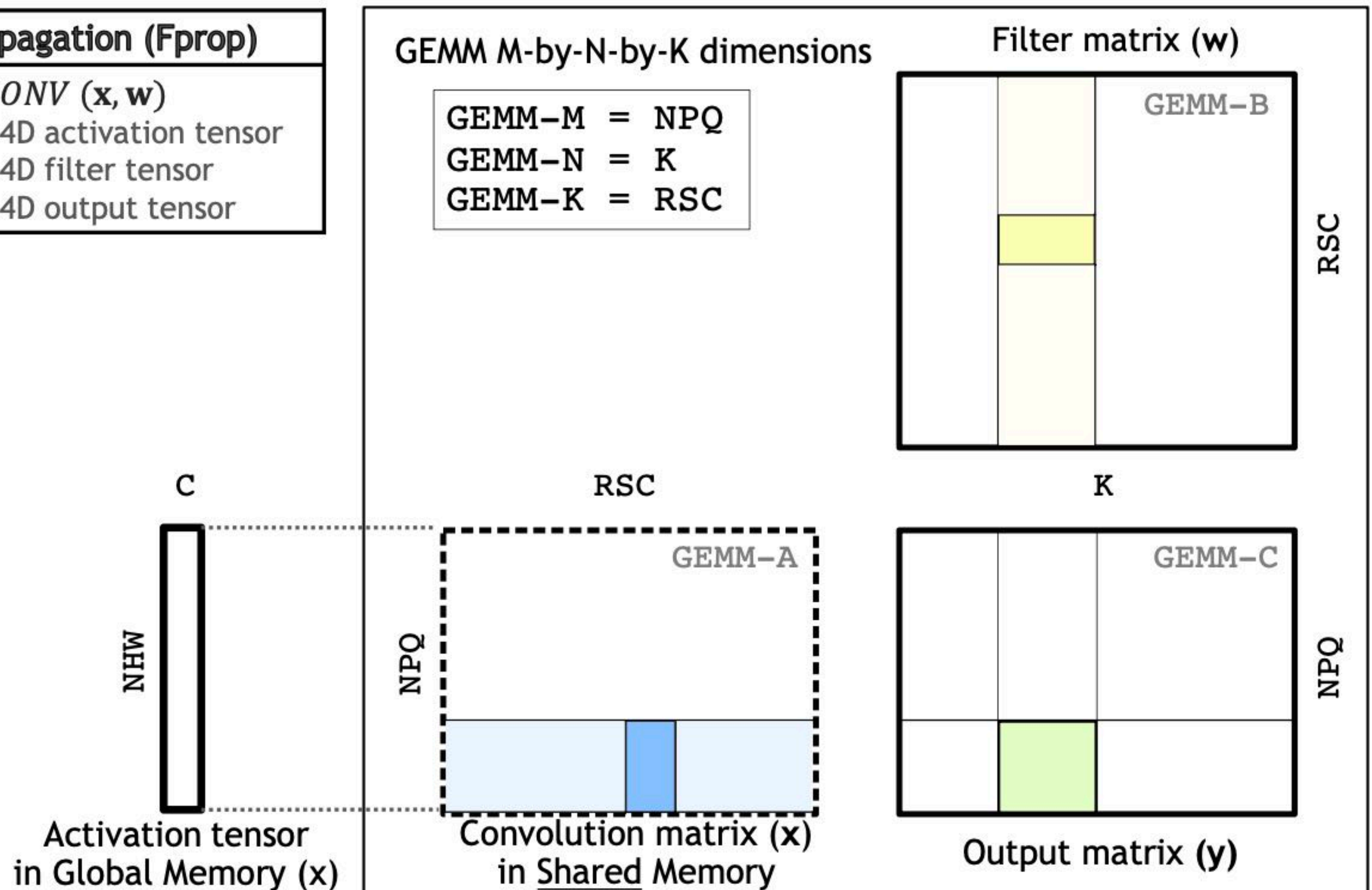
Forward Propagation (Fprop)	
$y = CONV(x, w)$	
$x[N, H, W, C]$: 4D activation tensor
$w[K, R, S, C]$: 4D filter tensor
$y[N, P, Q, K]$: 4D output tensor

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

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

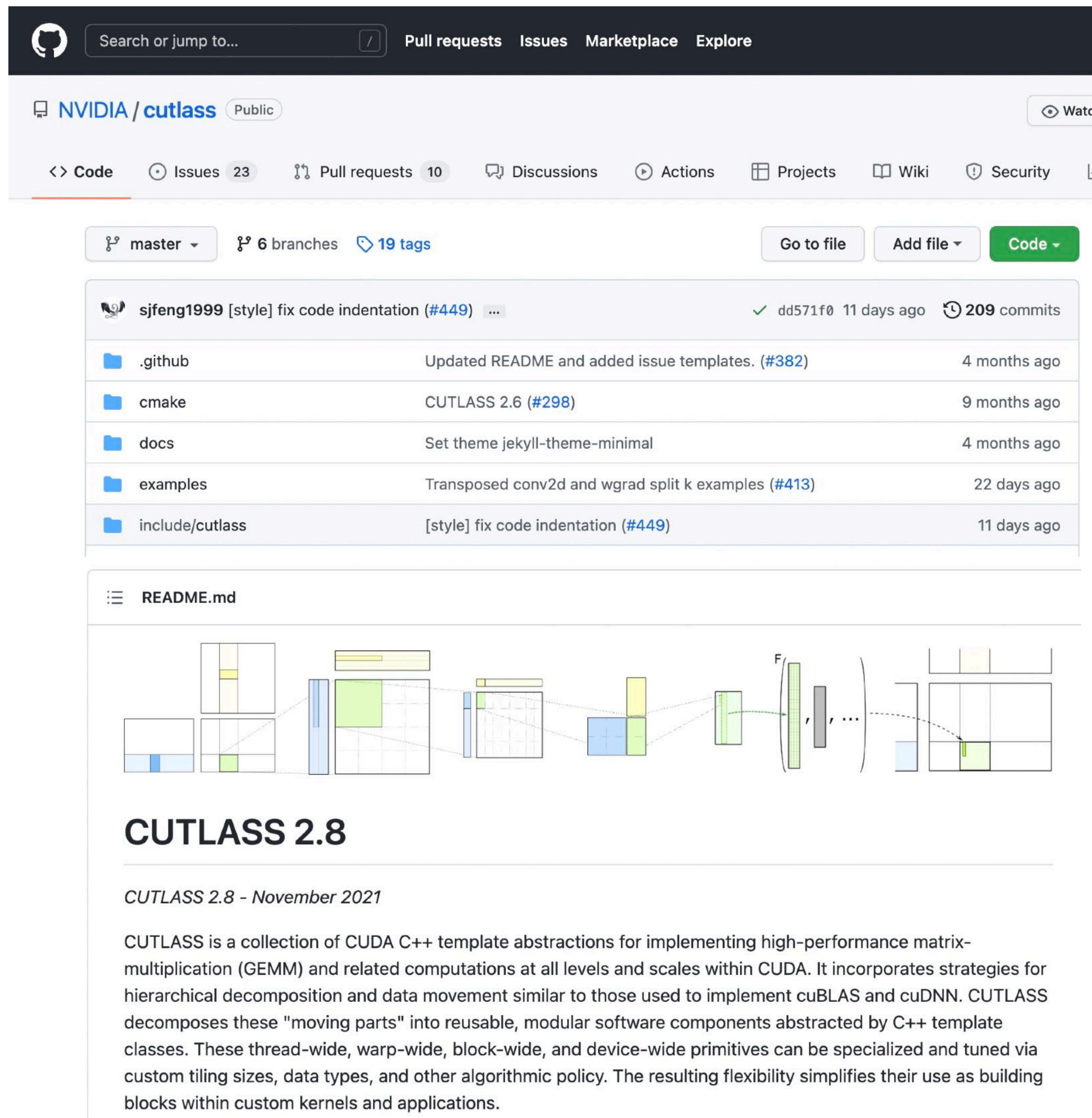
**Symbol reference:
Spatial support of filters: $R \times S$
Input channels: C
Number of filters: K
Batch size: N**

Image credit: NVIDIA



NVIDIA CUTLASS

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



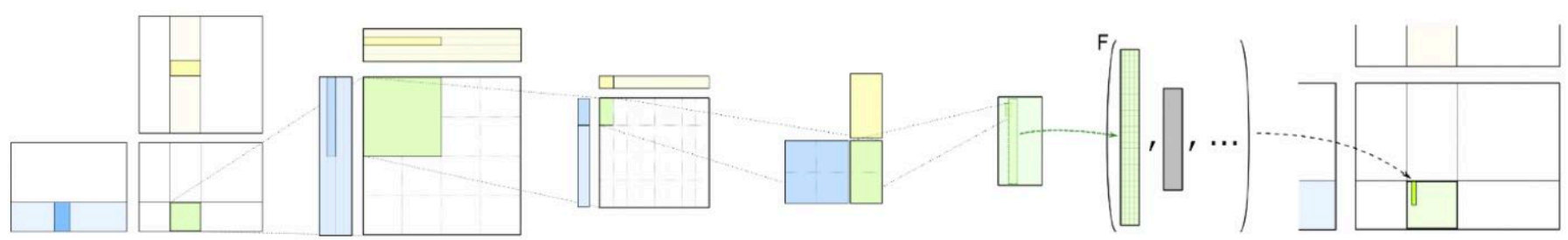
master 6 branches 19 tags

Go to file Add file Code

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

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

README.md



CUTLASS 2.8

CUTLASS 2.8 - November 2021

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

Fast (in-shared memory) GEMM

Fast WARP level GEMMs

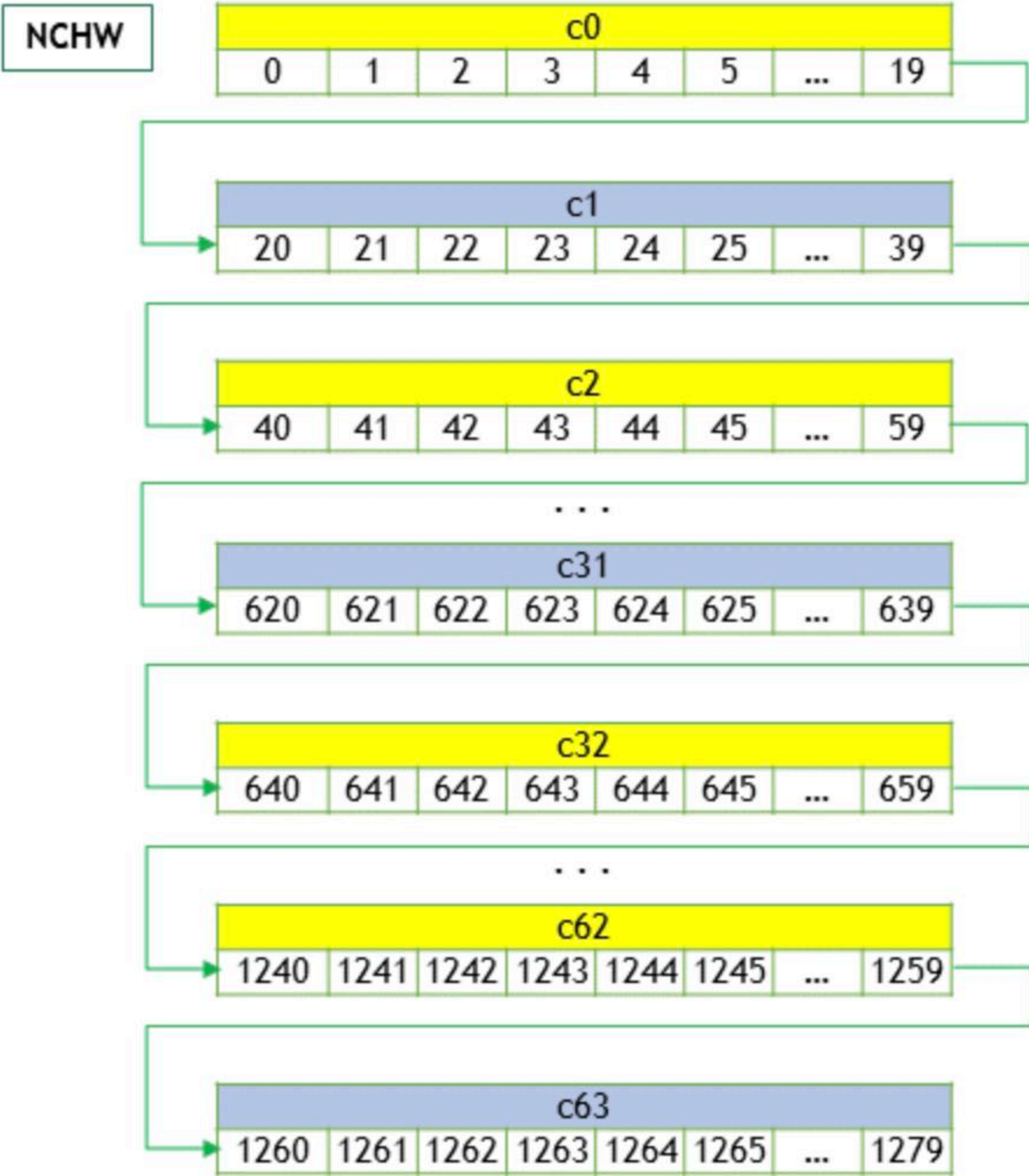
Iterators for fast block loading/tensor indexing

Tensor reductions

Etc.

NCHW tensor data layout

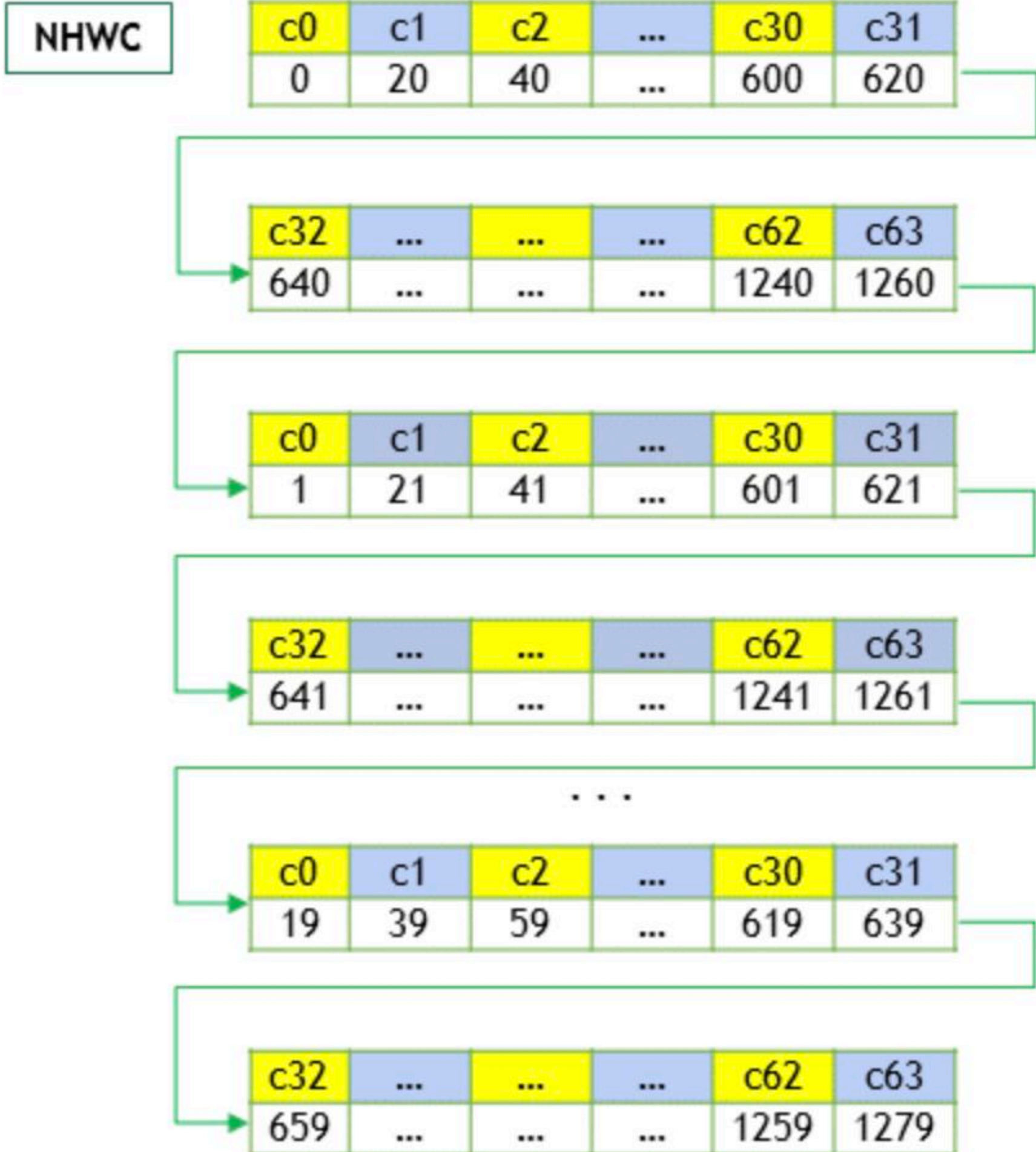
- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., number of channels); 64.
- **H** is the image height; 5.
- **W** is the image width; 4.



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

NHWC tensor data layout

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



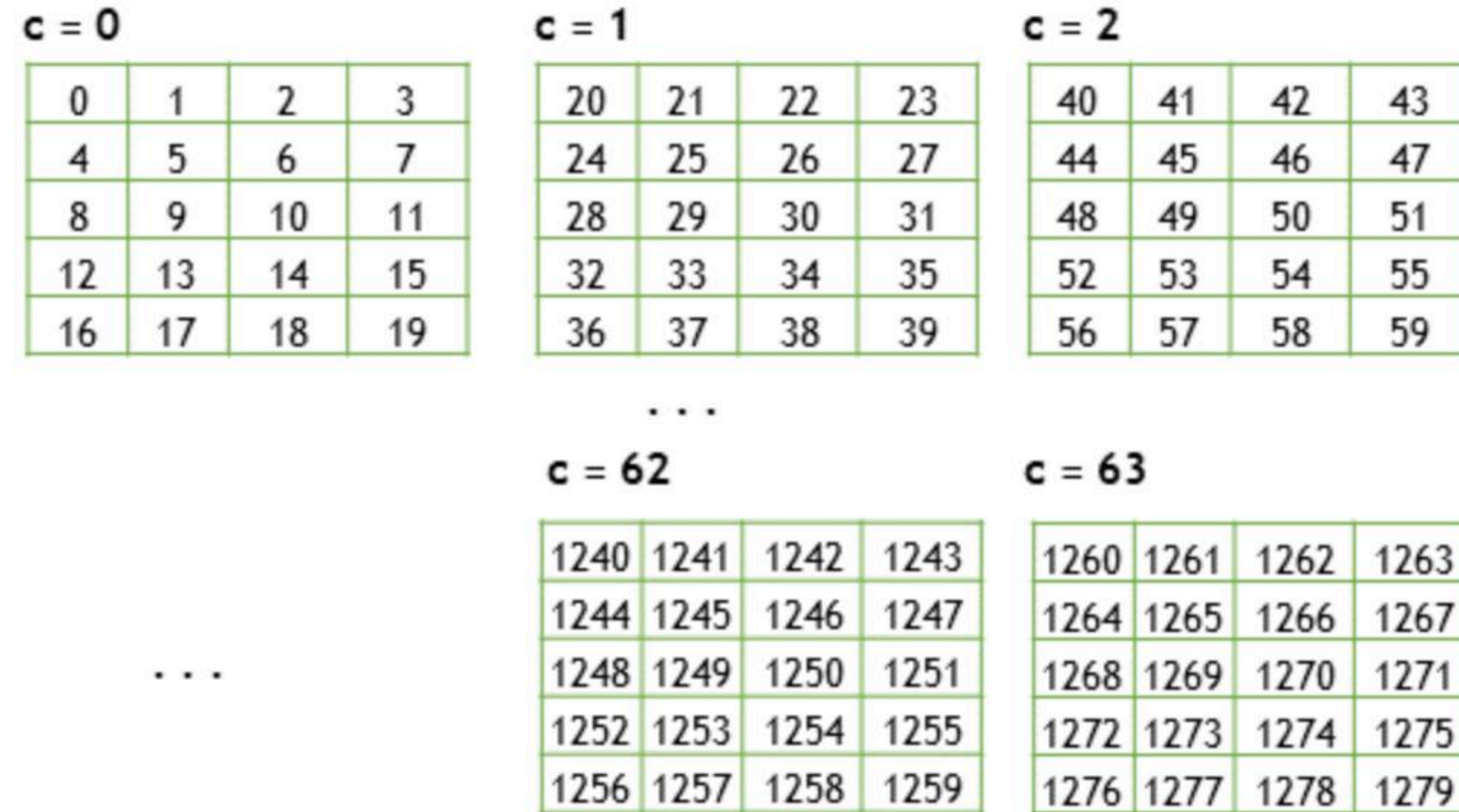
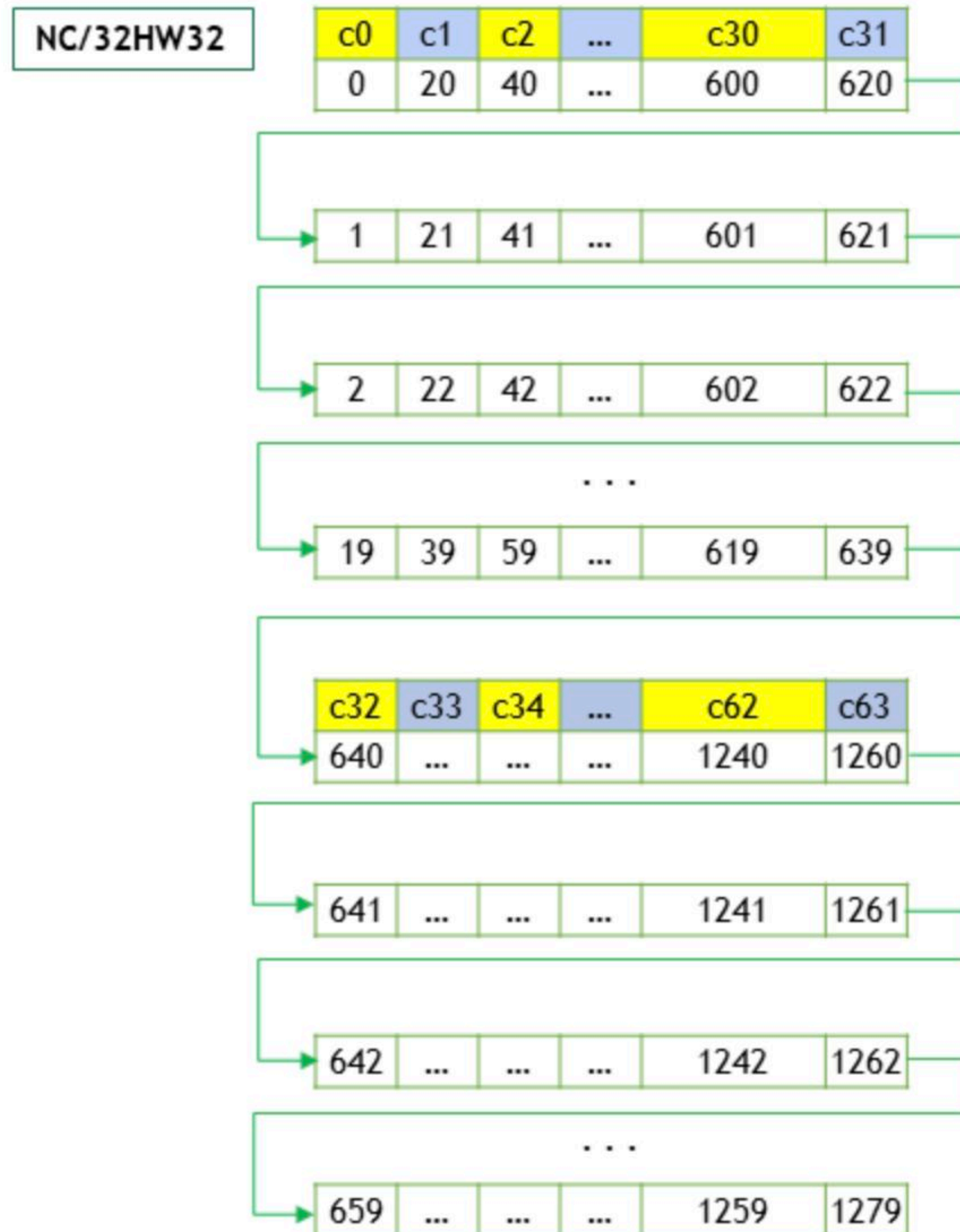
c = 0				c = 1				c = 2			
0	1	2	3	20	21	22	23	40	41	42	43
4	5	6	7	24	25	26	27	44	45	46	47
8	9	10	11	28	29	30	31	48	49	50	51
12	13	14	15	32	33	34	35	52	53	54	55
16	17	18	19	36	37	38	39	56	57	58	59

...

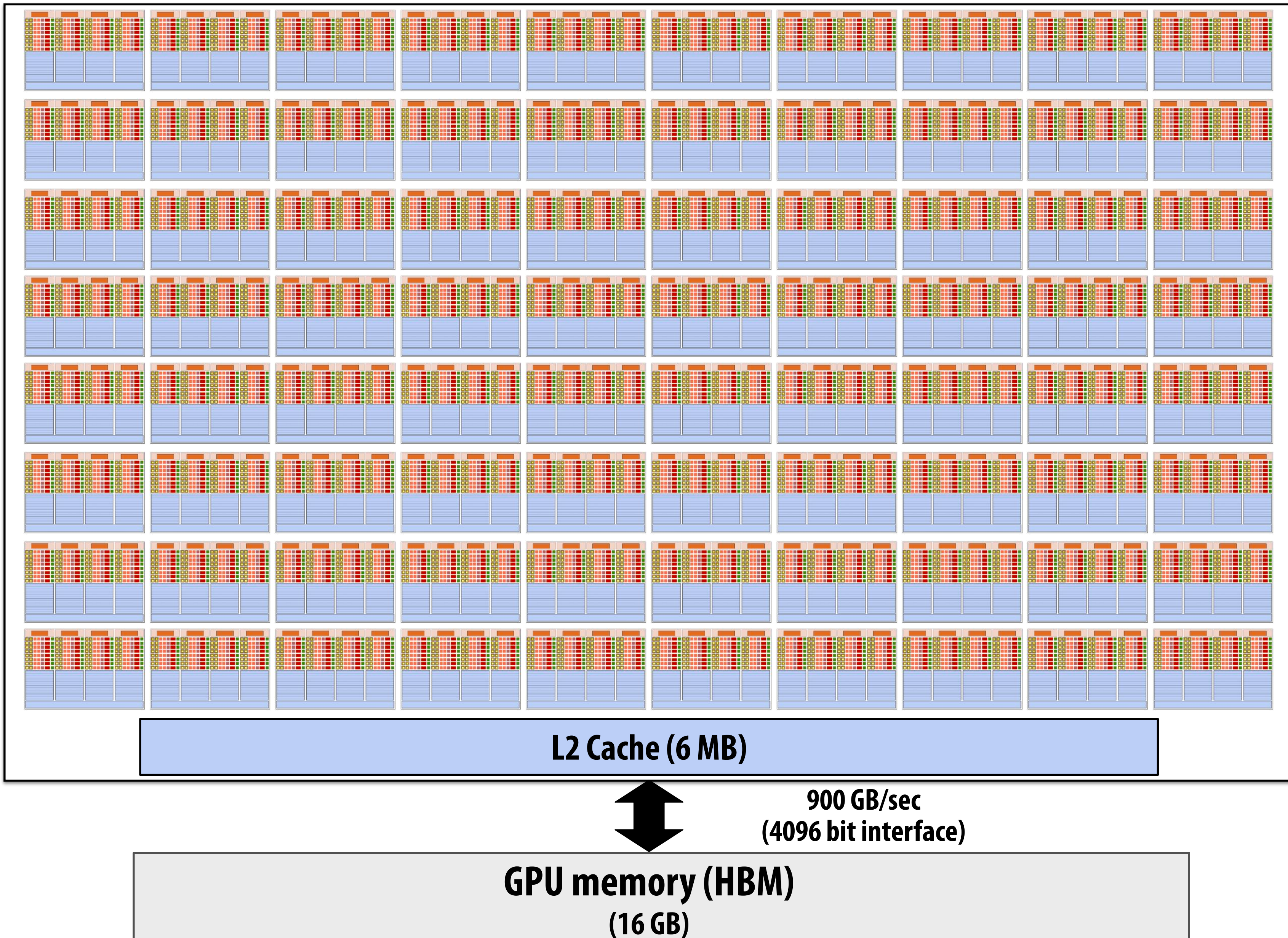
c = 62				c = 63			
1240	1241	1242	1243	1260	1261	1262	1263
1244	1245	1246	1247	1264	1265	1266	1267
1248	1249	1250	1251	1268	1269	1270	1271
1252	1253	1254	1255	1272	1273	1274	1275
1256	1257	1258	1259	1276	1277	1278	1279

Another tensor layout (blocked C)

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



Recall: NVIDIA V100 GPU (80 SMs)



Many processing units and many tensor cores.

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

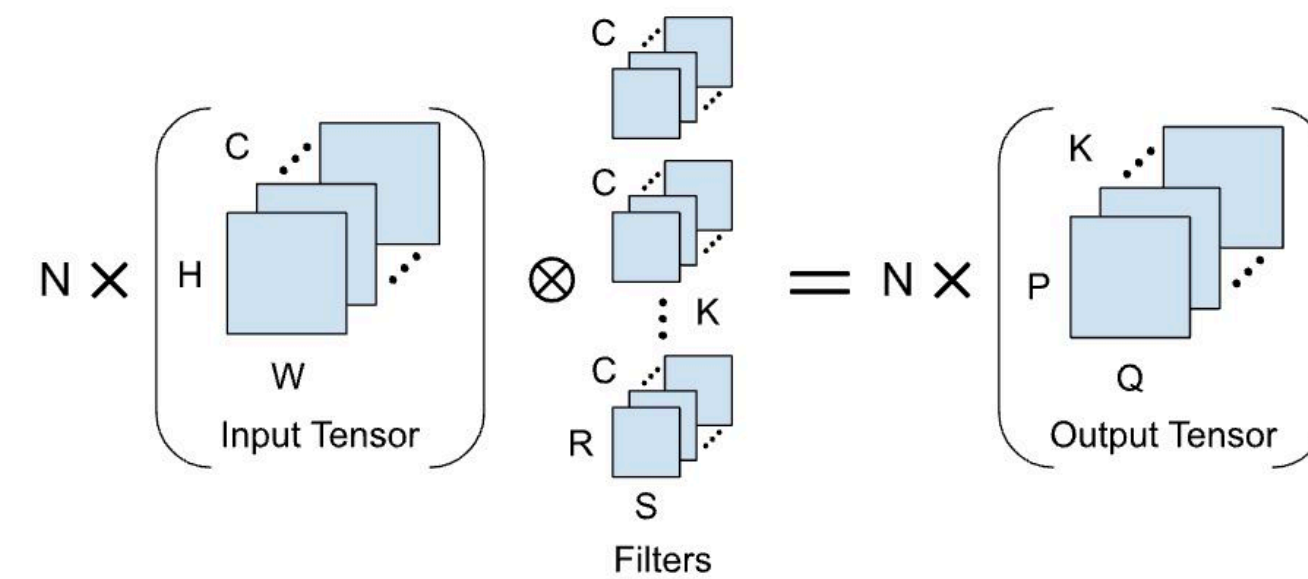
Higher performance with “more work”

N=1, P=Q=64 case:

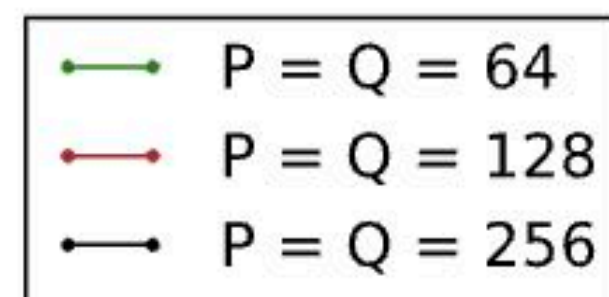
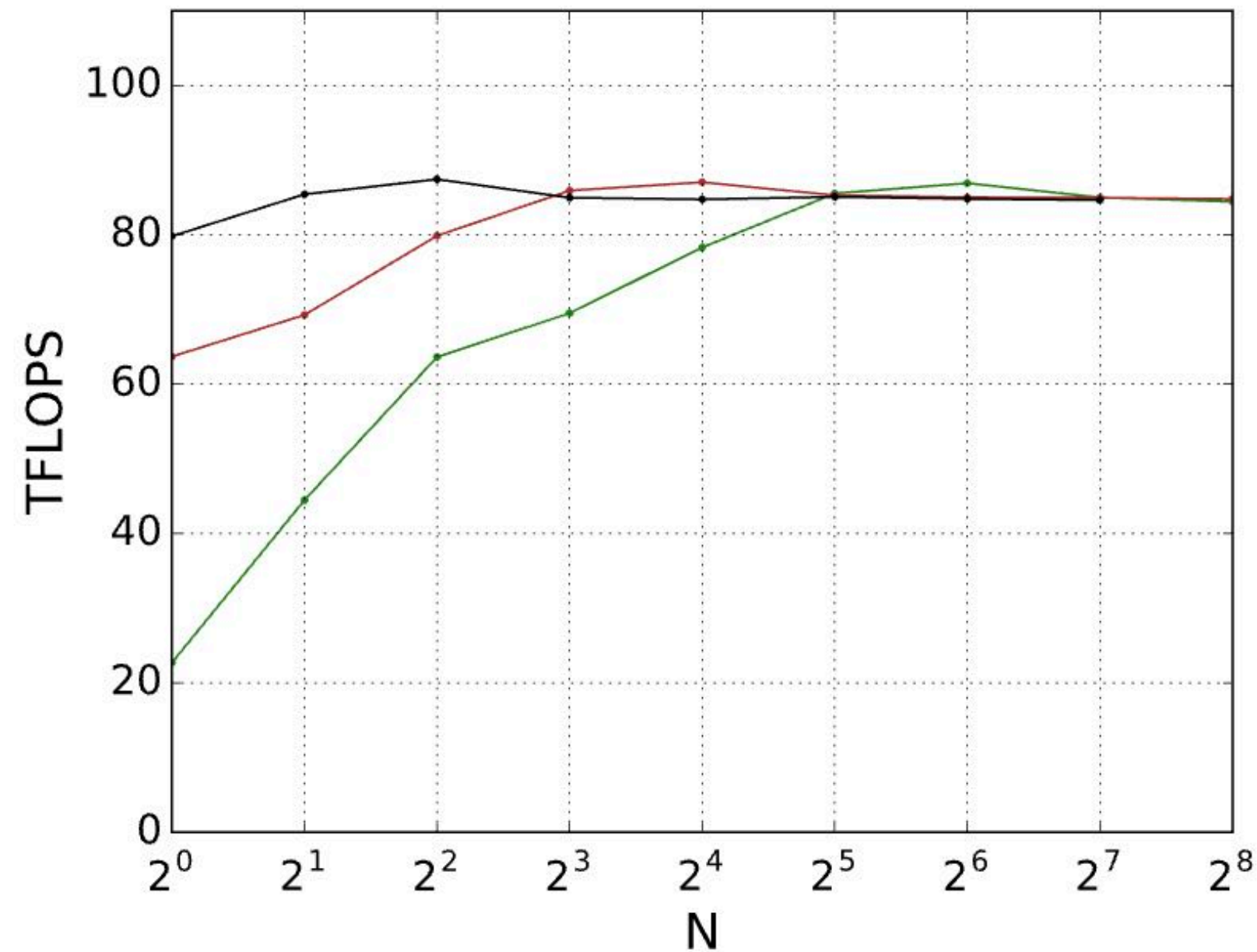
64 x 64 x 128 x 1 = 524K outputs = 2 MB of output data (float32)

N=32, P=Q=256 case:

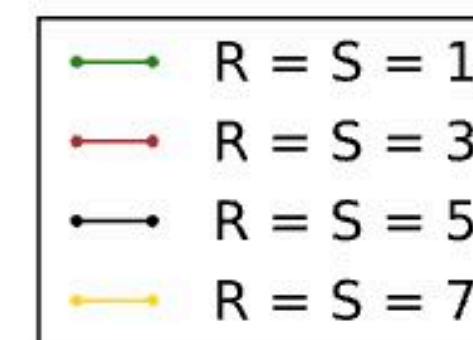
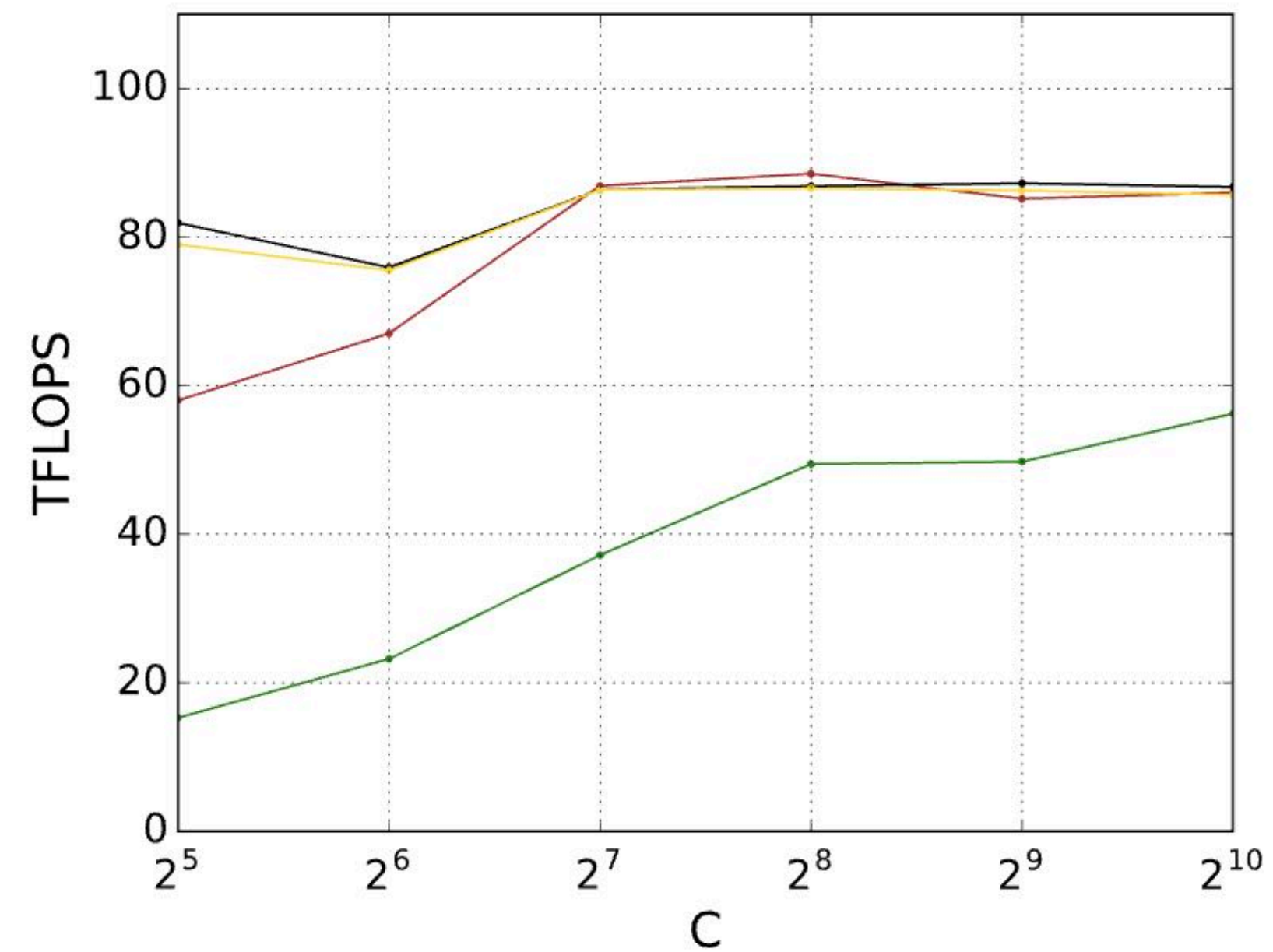
256 x 256 x 128 x 32 = 256M outputs = 1 GB of output data (float32)



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



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



Algorithmic improvements

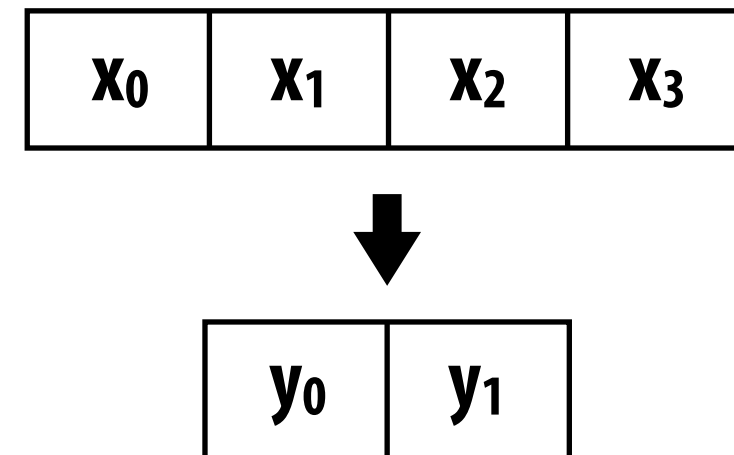
■ Direct convolution can be implemented efficiently in Fourier domain

(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: $w_0 w_1 w_2$



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 multiplies for 2x2 block of output)

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

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

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

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

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

Filter dependent
(can be precomputed)

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

Transformer architecture for sequence models

Sequence of tokens in, sequence of tokens out

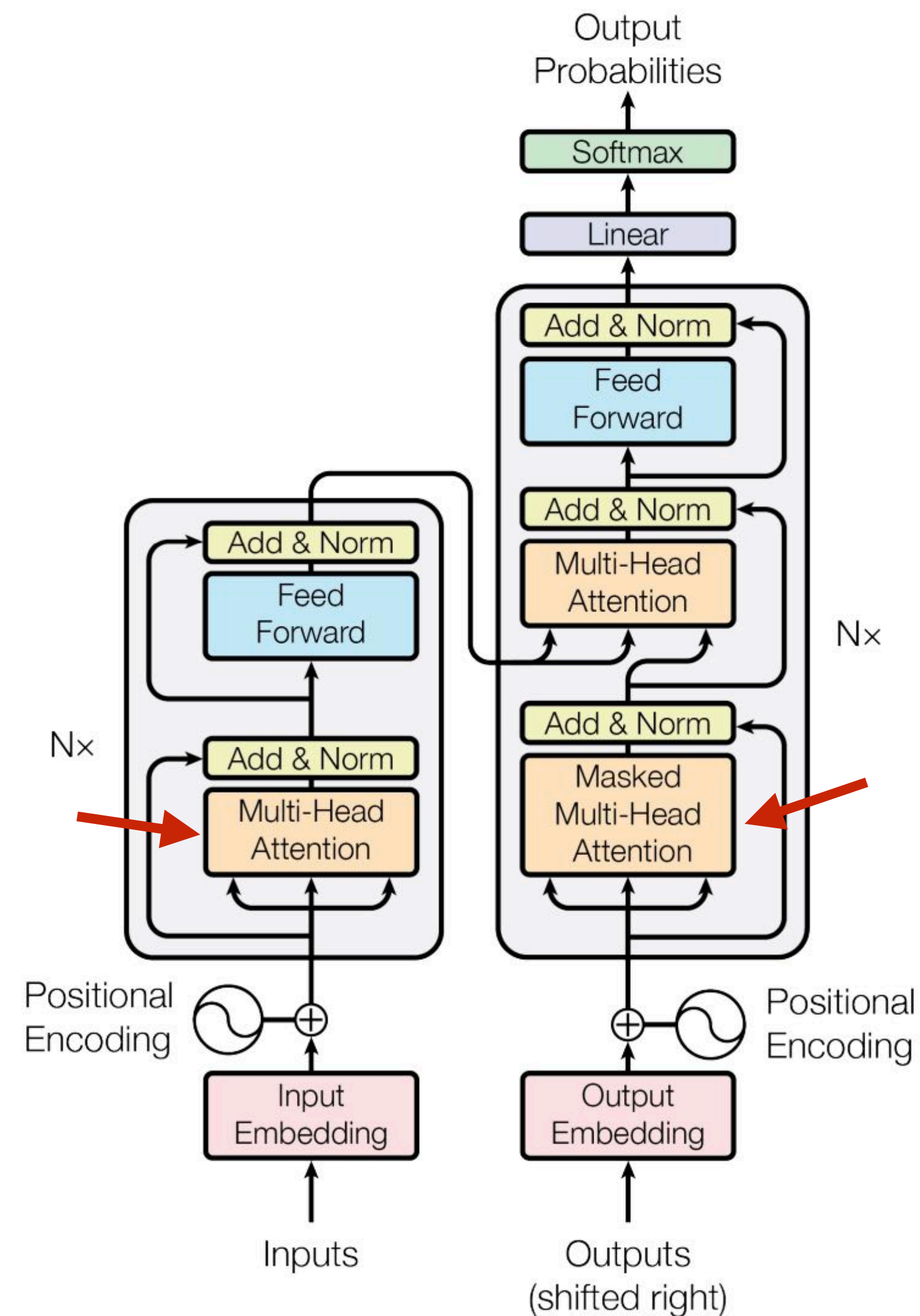
Example: next word prediction

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

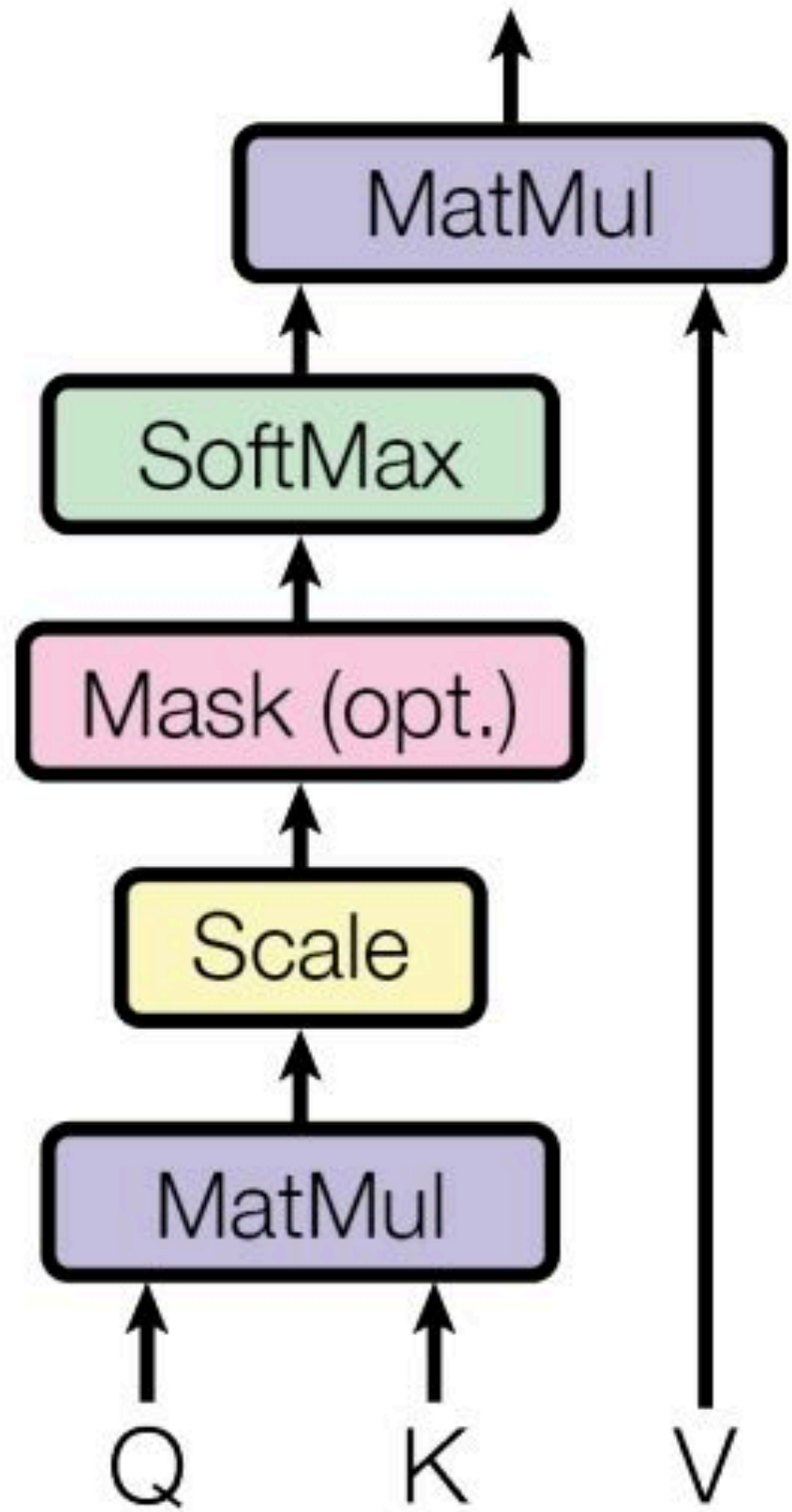
a DNN, which has

a simple model, say a

some simple examples of a conv



Attention module

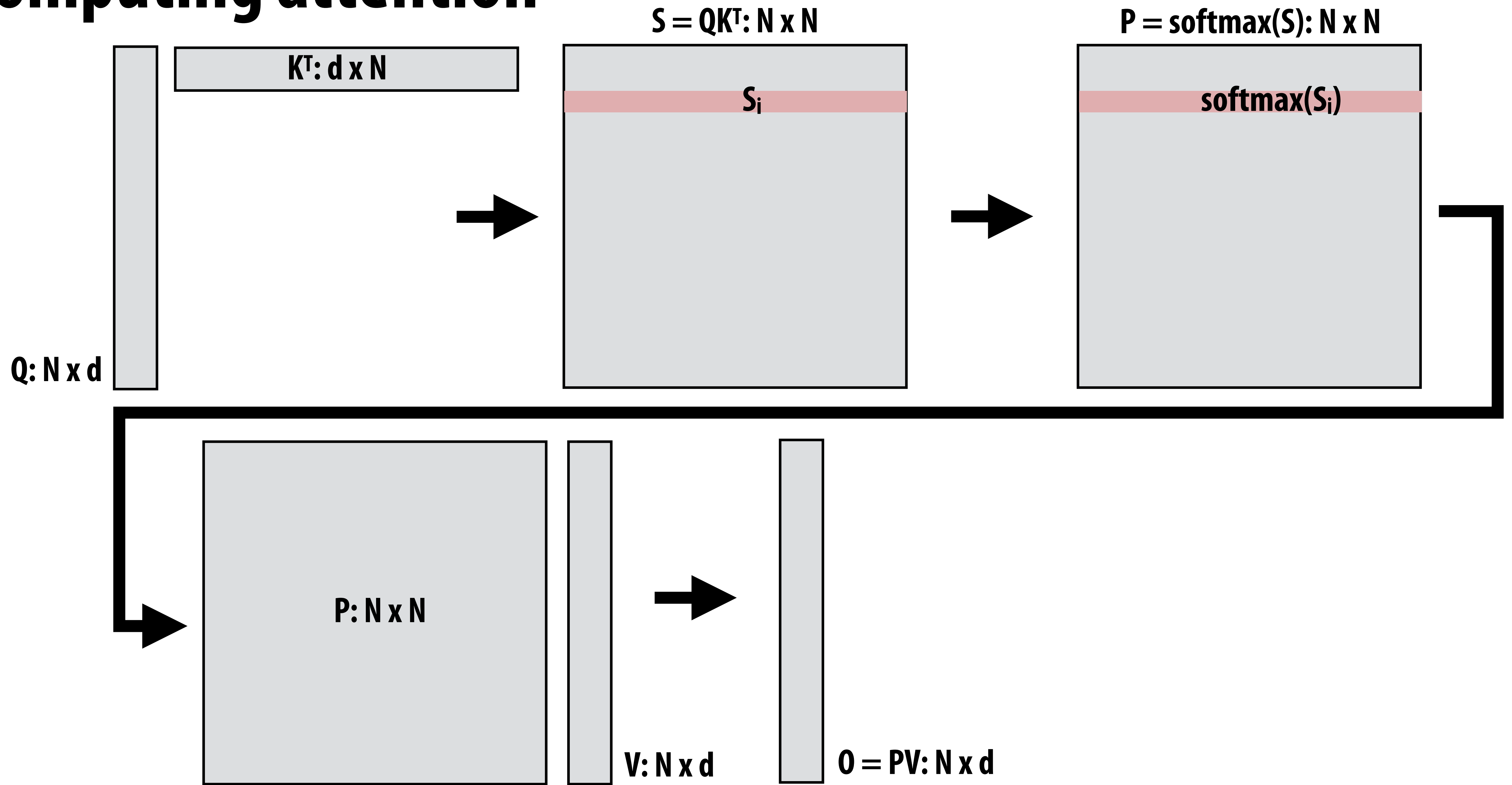


Let $\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$

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

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

Computing attention

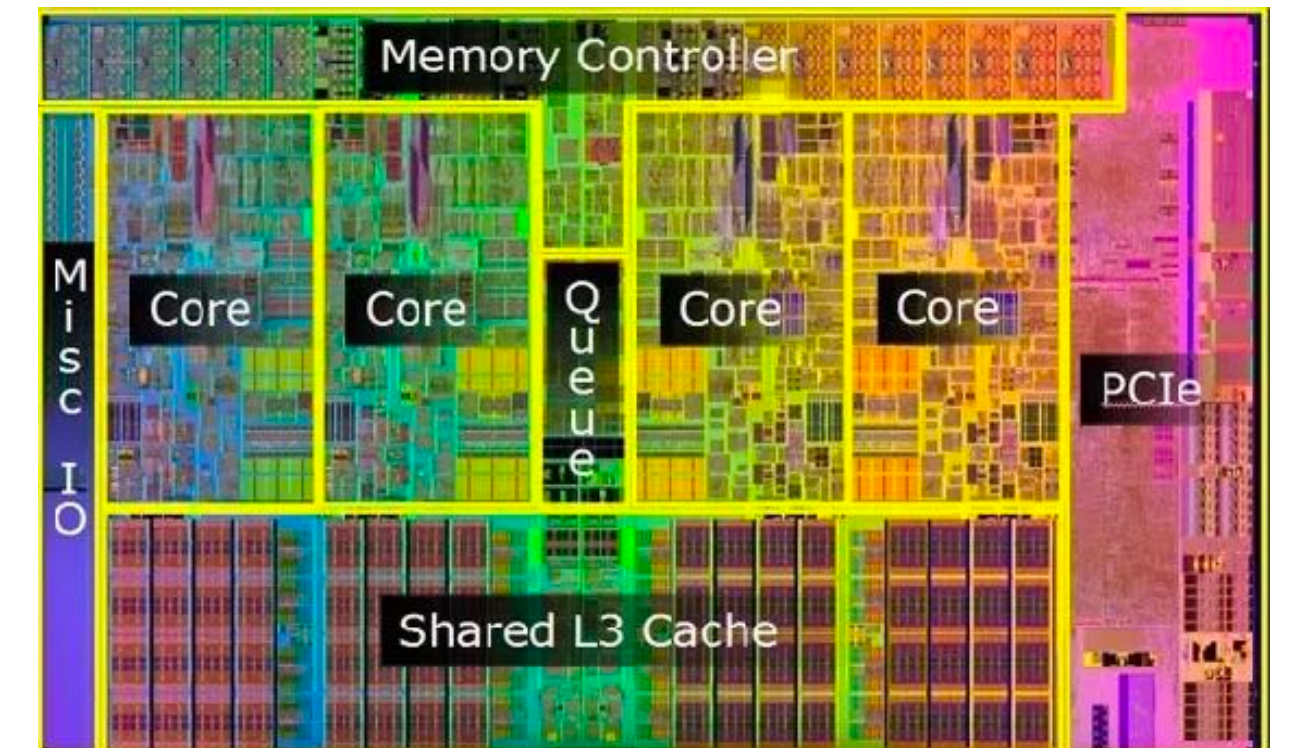


Low-level vendor libraries offer high-performance implementations of key DNN layers

NVIDIA cuDNN



Intel® oneAPI Deep Neural Network Library



Example: CUDNN convolution

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

Possible algorithms:

CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_DIRECT

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

CUDNN_CONVOLUTION_FWD_ALGO_FFT

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

CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING

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

CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD

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

CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED

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

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



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

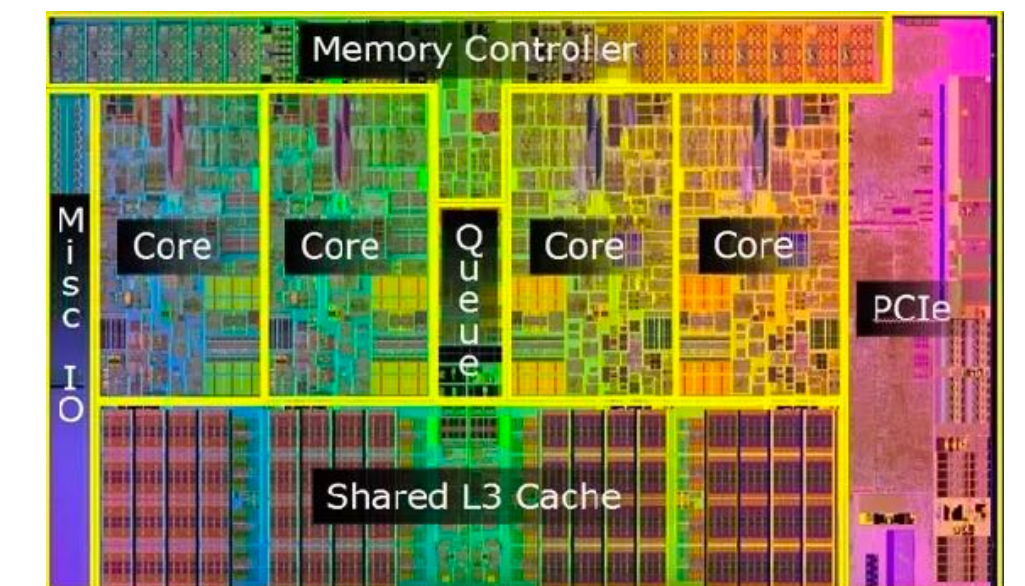
NVIDIA cuDNN



Implemented via lower level libraries



Intel® oneAPI Deep Neural Network Library



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!



Fusing operations with conv layer

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

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

How would you also “fuse” in the max pool? (output = max of every 2x2 block of input)

Old style: APIs include hardcoded versions of “fused” ops

```
cudaStatus_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 $y = \text{act}(\alpha_1 * \text{conv}(x) + \alpha_2 * z + \text{bias})$.

Tensorflow:

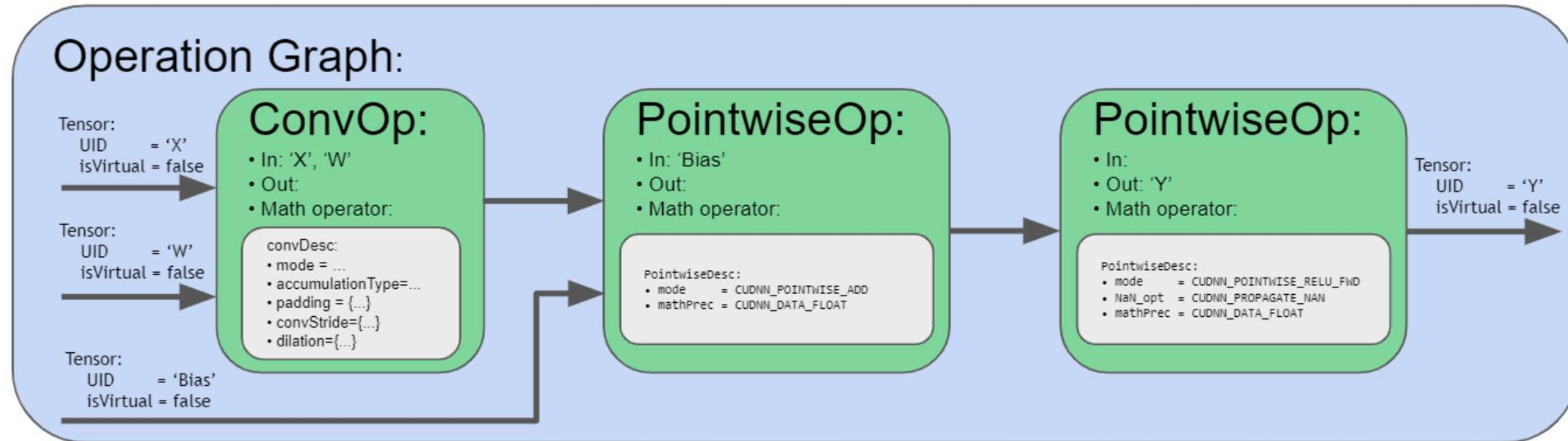
`tensorflow::ops::FusedBatchNorm`

Batch normalization.

`tensorflow::ops::FusedResizeAndPadConv2D`

Performs a resize and padding as a preprocess during a convolution.

Limited fusion example: CUDNN “backend”

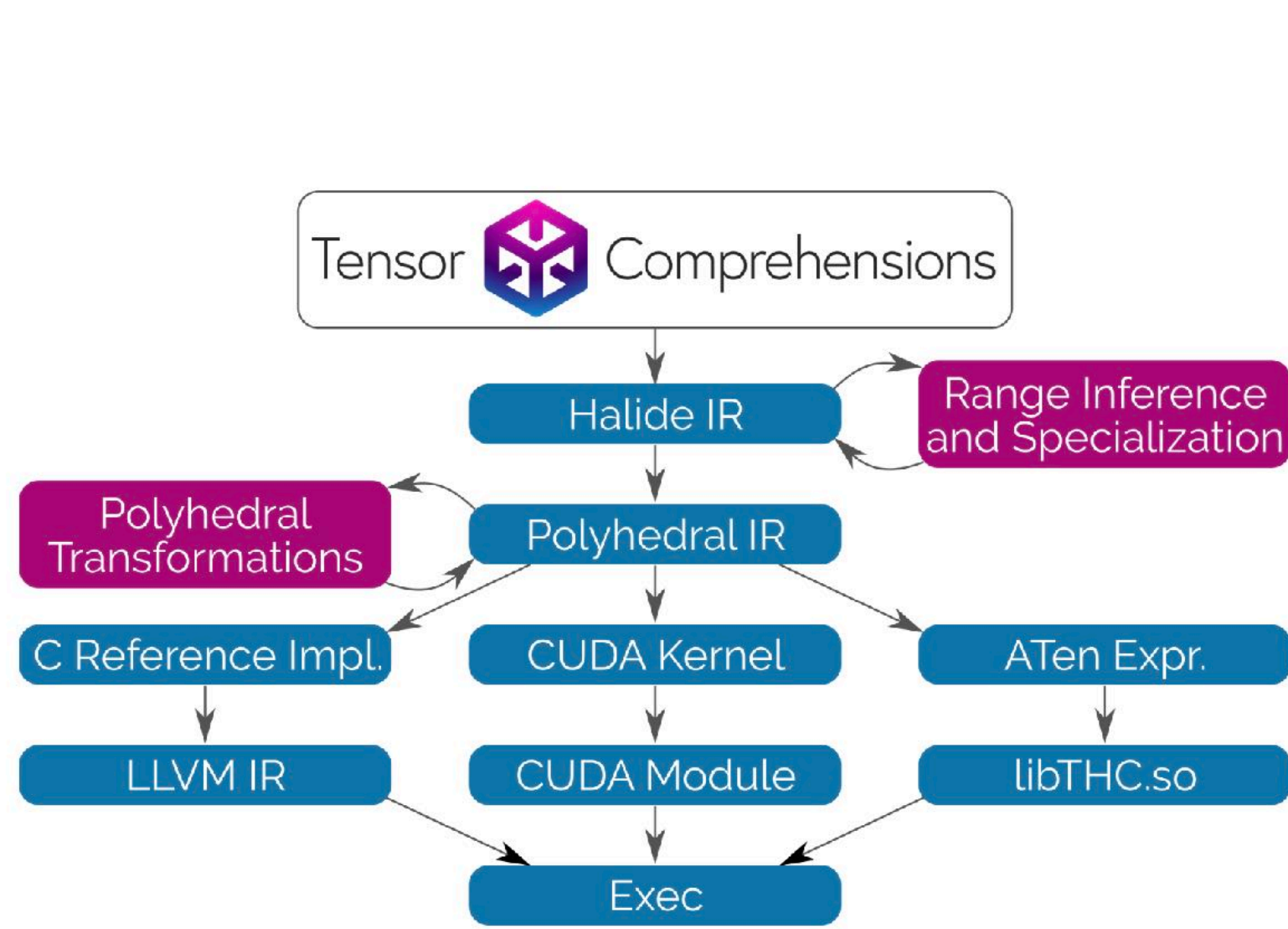


Note for operation fusion use cases, there are two different mechanisms in cuDNN to support them. First, there are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If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

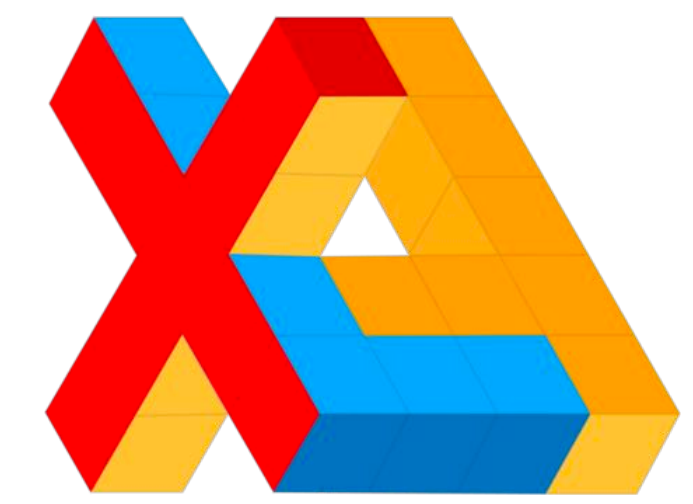
Compiler generates new implementations that “fuse” multiple operations into a single node that executes efficiently (without runtime overhead or communicating intermediate results through memory)

Note: this is Halide “compute at”

Many efforts to automatically schedule key DNN operations



MLIR
Multi-Level IR Compiler Framework



tvm Open Deep Learning Compiler Stack

license [Apache 2.0](#) build [passing](#)

[Documentation](#) | [Contributors](#) | [Community](#) | [Release Notes](#)

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

NVIDIA TensorRT
Programmable Inference Accelerator

COMMUNITY | DOWNLOAD | VTA | BLOG | DOCS | CONFERENCE | GITHUB

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

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

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

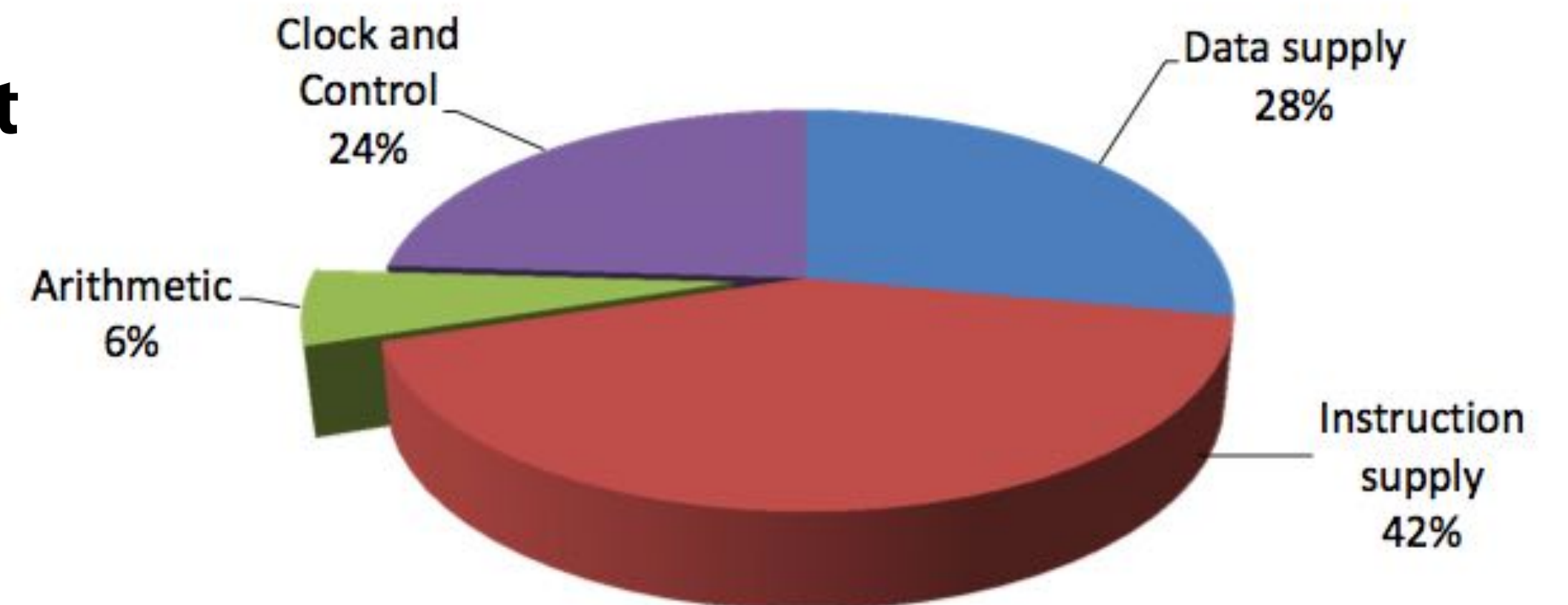
Our life will be much easier if we can just write mathematical expressions and have something magically turn them into efficient code implementations. Three years ago, we started deep learning compiler TVM and its search module AutoTVM were built as the first step towards this goal. AutoTVM employs a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

Deep Learning Compiler
nGraph

Two computer architecture reminders

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]

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
- **Implications**
 - Reading 10 GB/sec from memory: ~1.6 watts
 - Entire power budget for mobile GPU: ~1 watt
(remember phone is also running CPU, display, radios, etc.)
 - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
 - Exploiting locality matters!!!

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

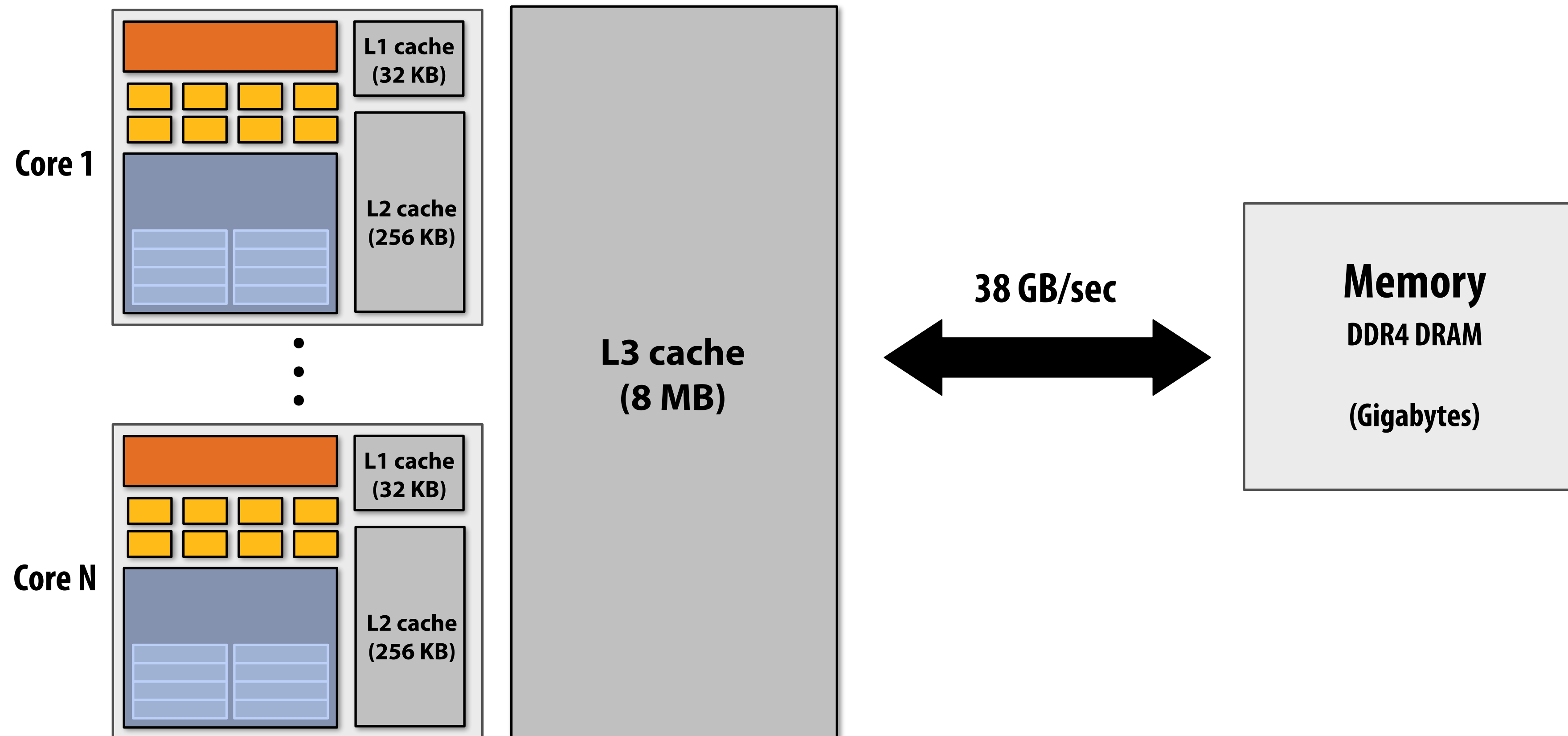
* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

On-chip caches locate data near processing

Processors run efficiently when data is resident in caches

Caches reduce memory access latency*

Caches reduce the energy cost of data access



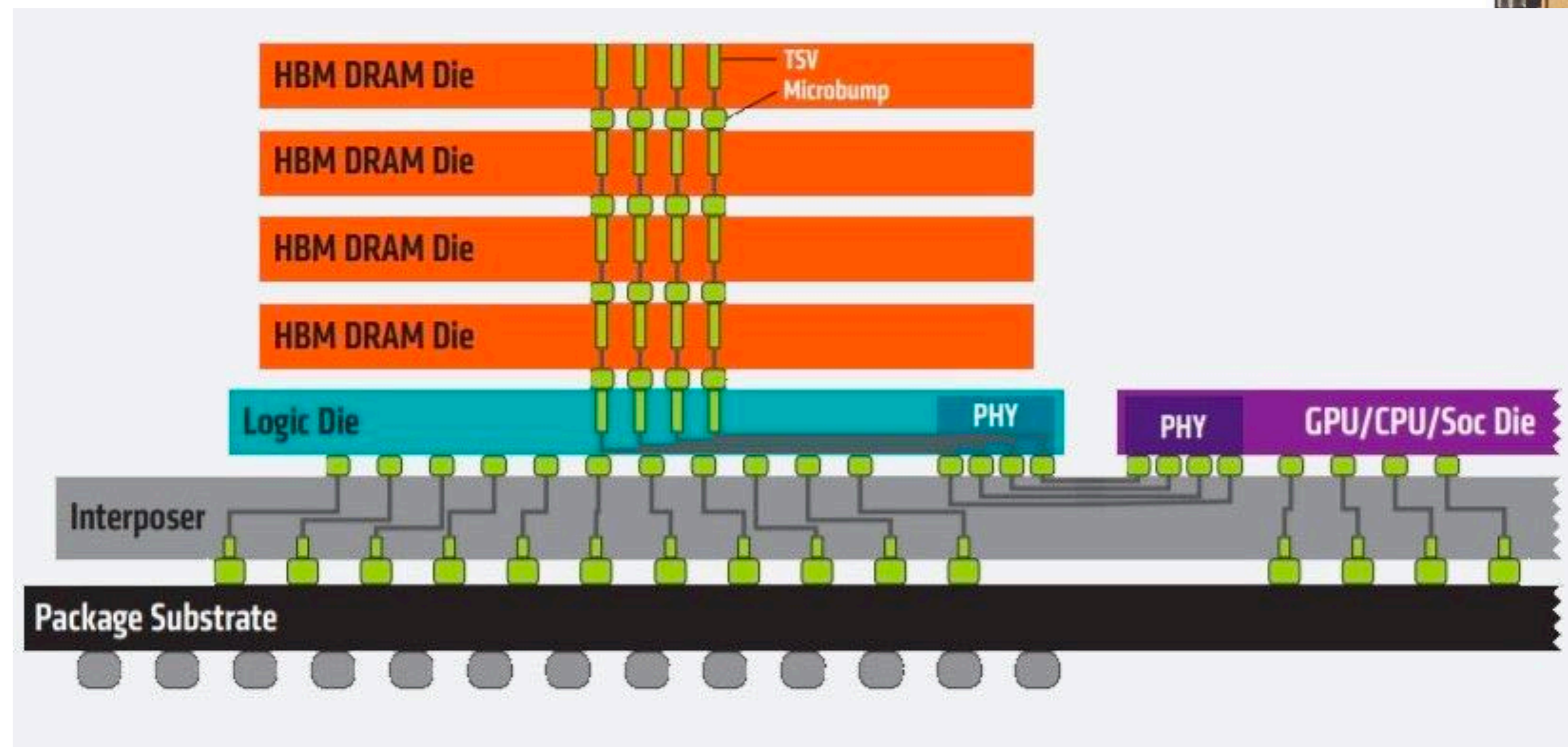
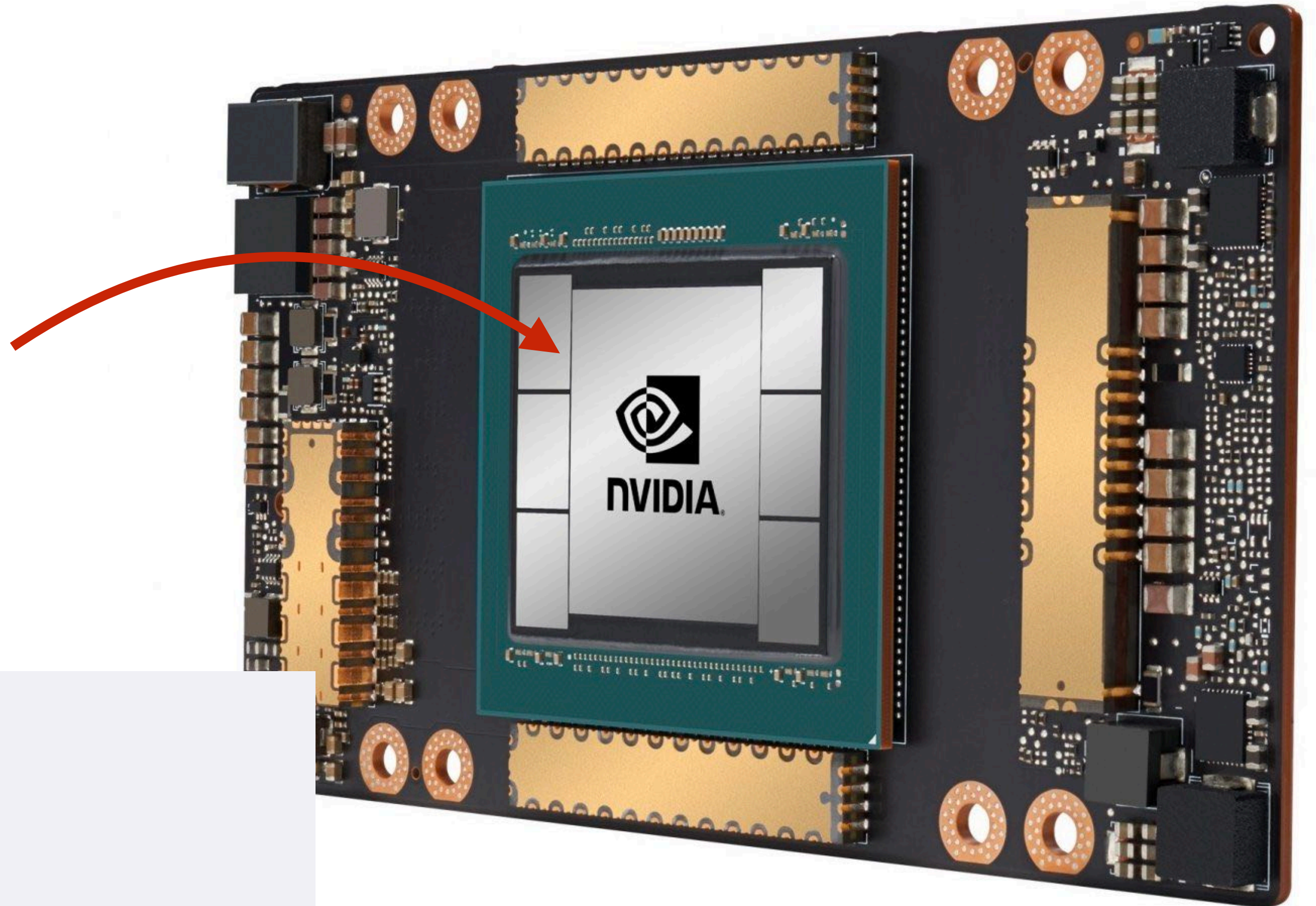
* Caches also provide high bandwidth data transfer to CPU

Memory stacking locates memory near chip

**Example:
NVIDIA A100 GPU**

**Up to 80 GB HBM2 stacked memory
2 TB/sec memory bandwidth**

**Also note: A100 has 40 MB L2 cache
(increased from 6.1 MB on V100)**



Improving hardware efficiency for DNN operations

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 hardware and services to run AI applications, today announced a \$676 million Series D funding round I Fund 2*. The round includes additional new investors Temasek and GIC, plus existing backers including managed by BlackRock, Intel Capital, GV (formerly Google Ventures), Walden International and WRVI.

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

Tweet this

This Series D brings SambaNova's total funding and rockets its valuation to more than \$5 billion.

Now the best-funded AI systems and services platform, SambaNova will use its latest injection to legacy competitors as it continues to shatter the hardware and software currently on the market - solutions for private and public sectors more acc

"We're here to revolutionize the AI market, and this round greatly accelerates that mission," said Rodrigo founder and CEO. "Traditional CPU and GPU architectures have reached their computational limits. To to solve humanity's greatest technology challenges, a new approach is needed. We've figured out that to see a wealth of prudent investors validate that."

SambaNova's flagship offering is Dataflow-as-a-Service (DaaS), a subscription-based, extensible AI services platform designed to jump-start enterprise-level AI initiatives, augmenting organizations' AI capabilities and accelerating the work of existing data 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

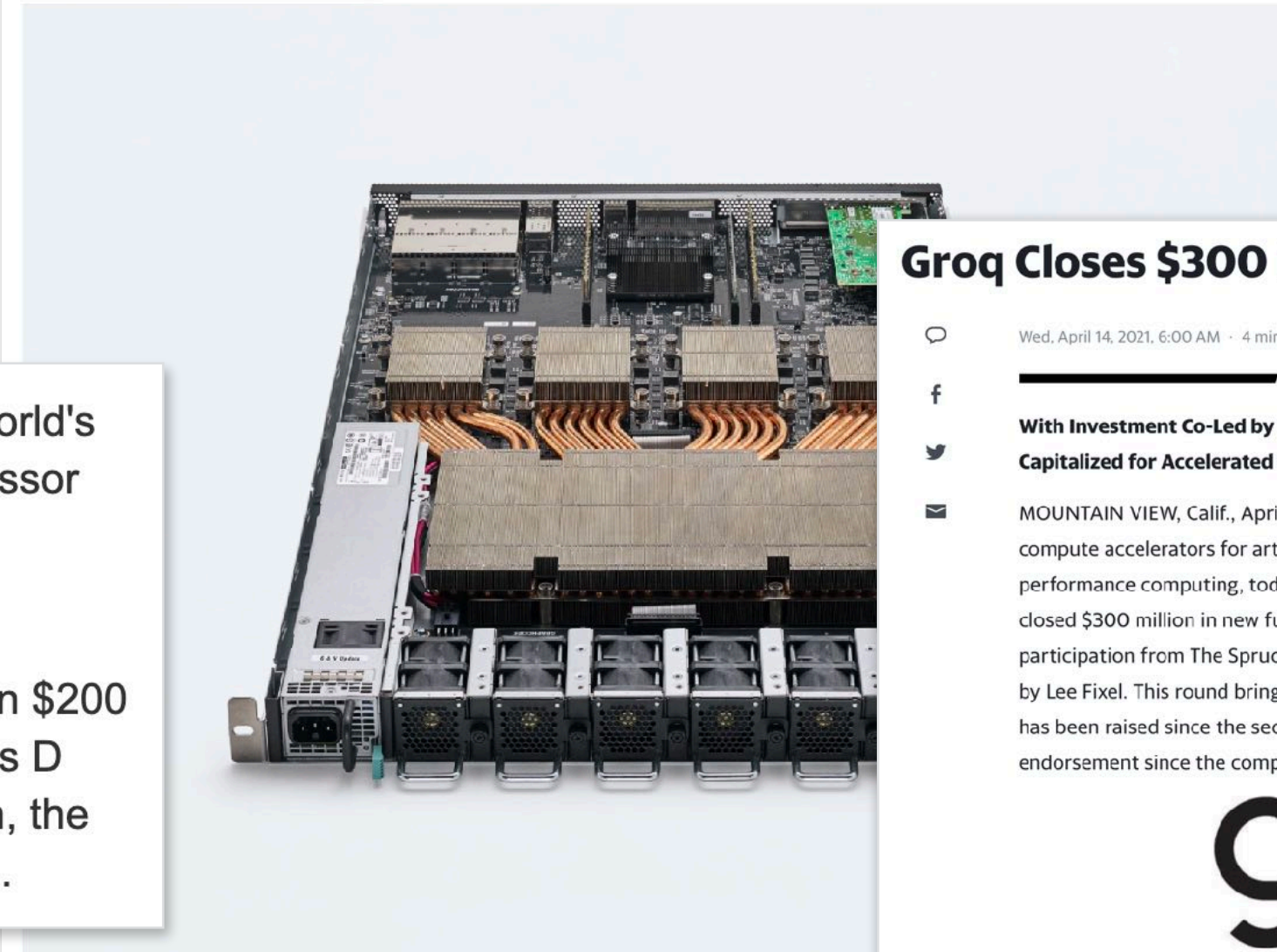


Image Credits: Graphcore

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

Groq Closes \$300 Million Fundraise

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

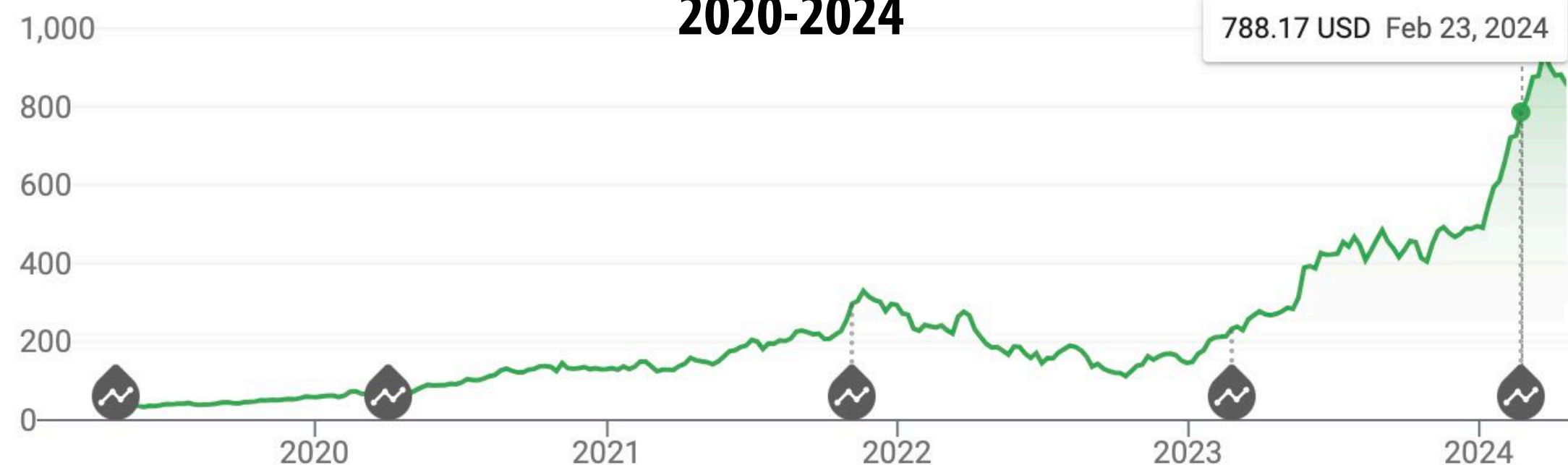
With Investment Co-Led by Tiger Global Management and D1 Capital, Groq Is Well Capitalized for Accelerated Growth

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



Groq logo

NVIDIA Market Cap 2020-2024



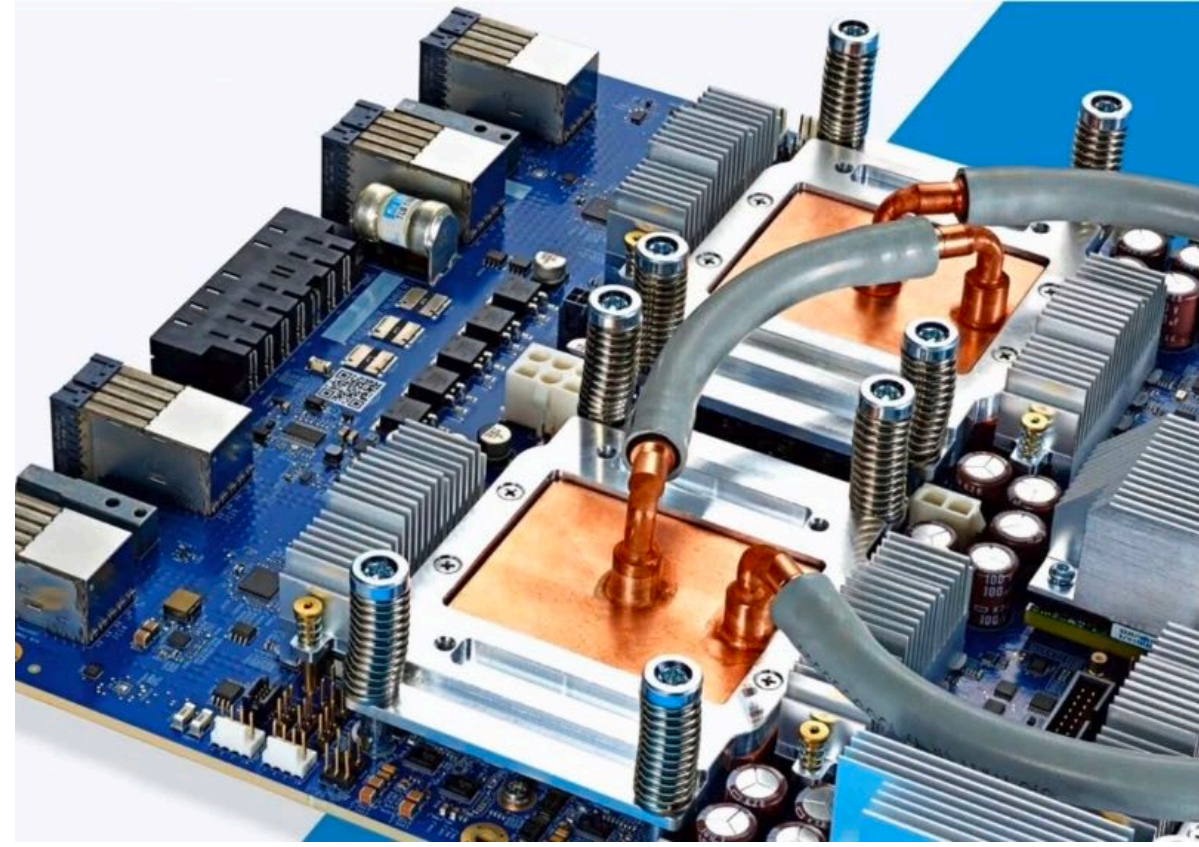
Intel Acquires Artificial Intelligence Chipmaker Habana Labs

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

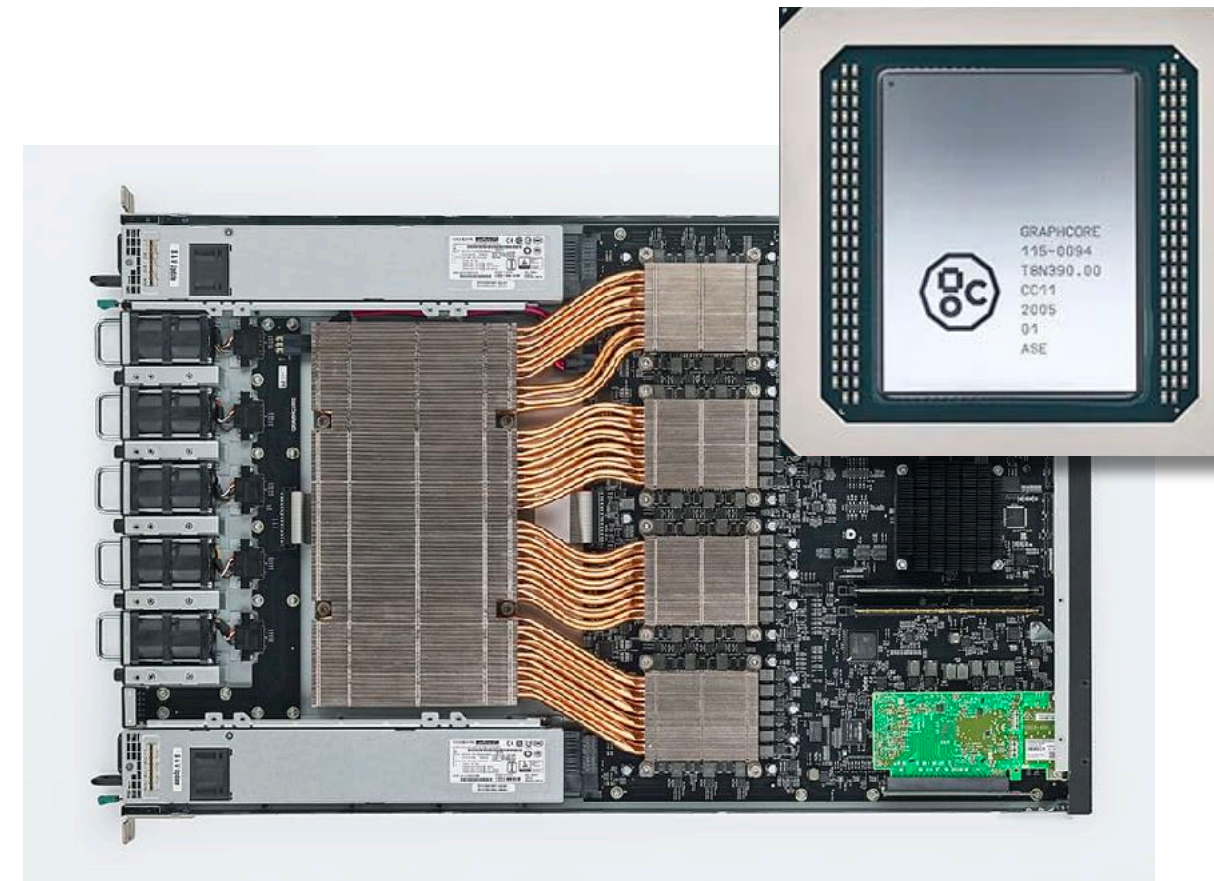
SANTA CLARA Calif., Dec. 16, 2019 – Intel Corporation today announced that it has acquired Habana Labs, an Israel-based developer of programmable deep learning accelerators for the data center for approximately \$2 billion. The combination strengthens Intel's artificial intelligence (AI) portfolio and accelerates its efforts in the nascent, fast-growing AI silicon market, which Intel expects to be greater than \$25 billion by 2024¹.

"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."

Hardware acceleration of DNN inference/training



Google TPU3



GraphCore IPU



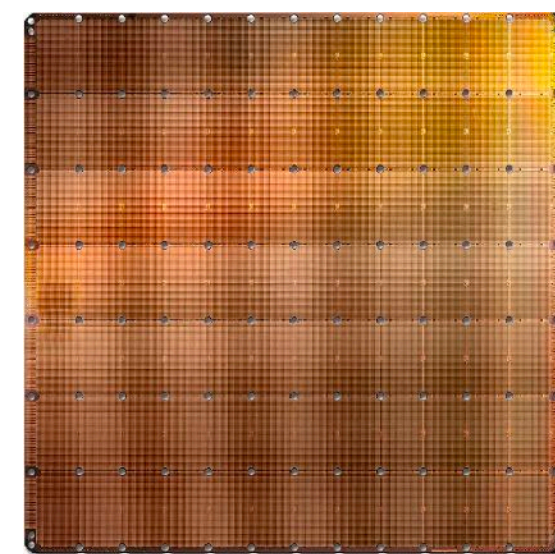
Apple Neural Engine



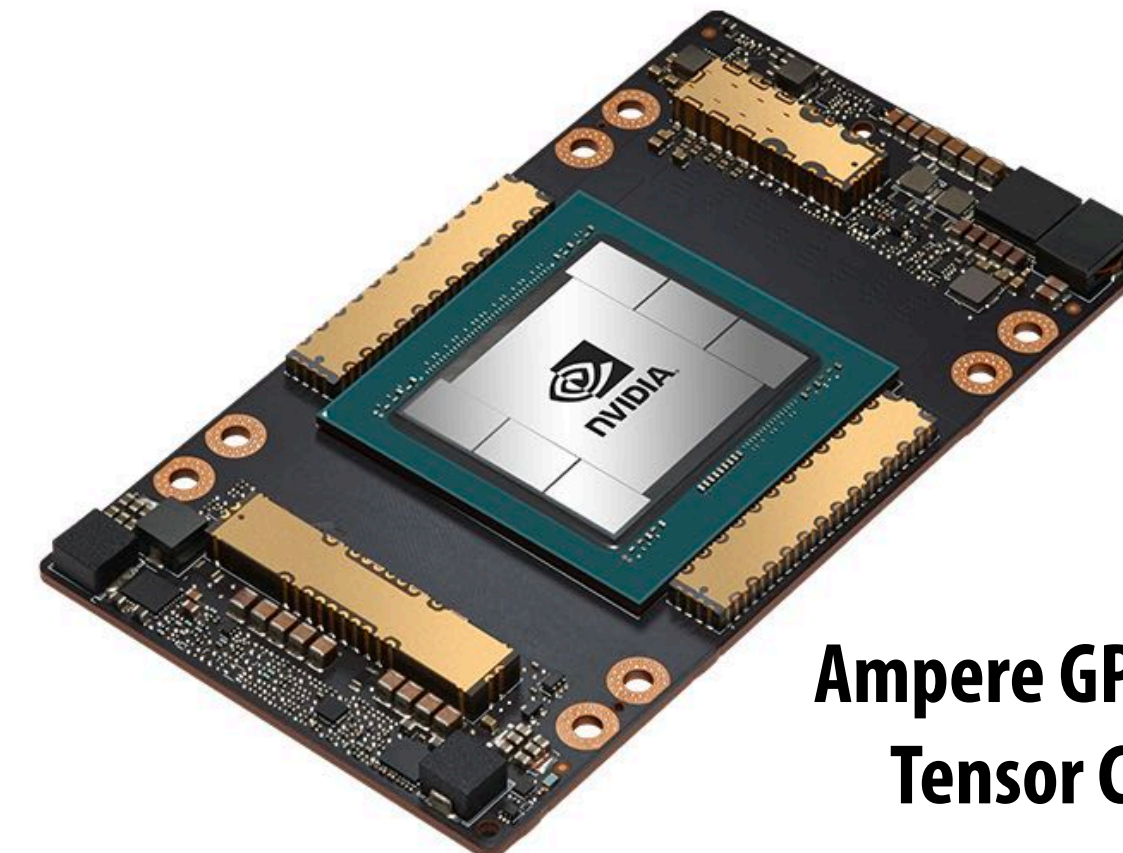
Intel Deep Learning Inference Accelerator



SambaNova Cardinal SN10



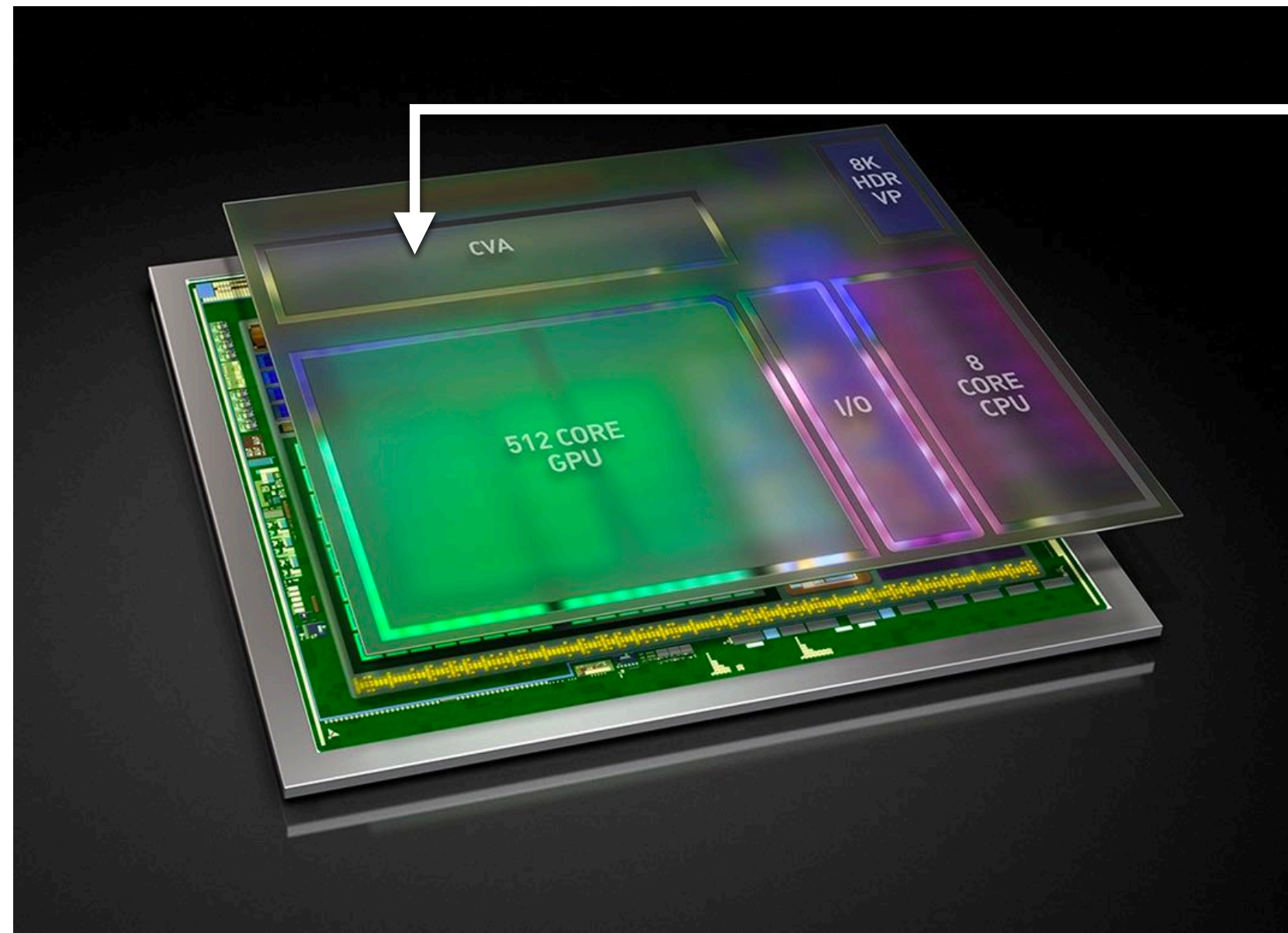
Cerebras Wafer Scale Engine



Ampere GPU with Tensor Cores

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 $8 \times 4 \times 4 \times 8$ matrix mul-add instr

$A \times B + C$ 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



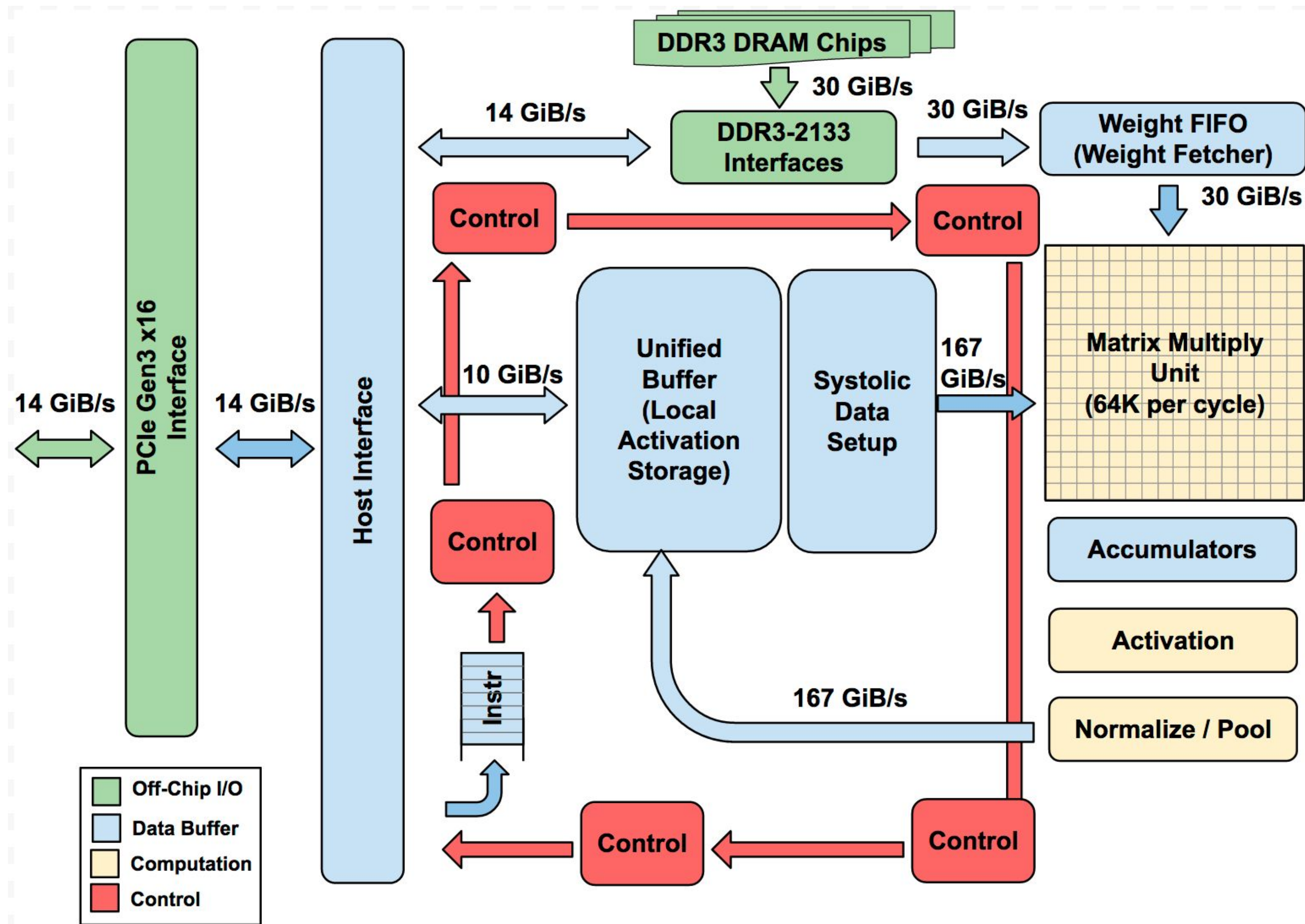
Single instruction to perform
 $2 \times 8 \times 4 \times 8$ FP16 + 8×8 TF32 ops



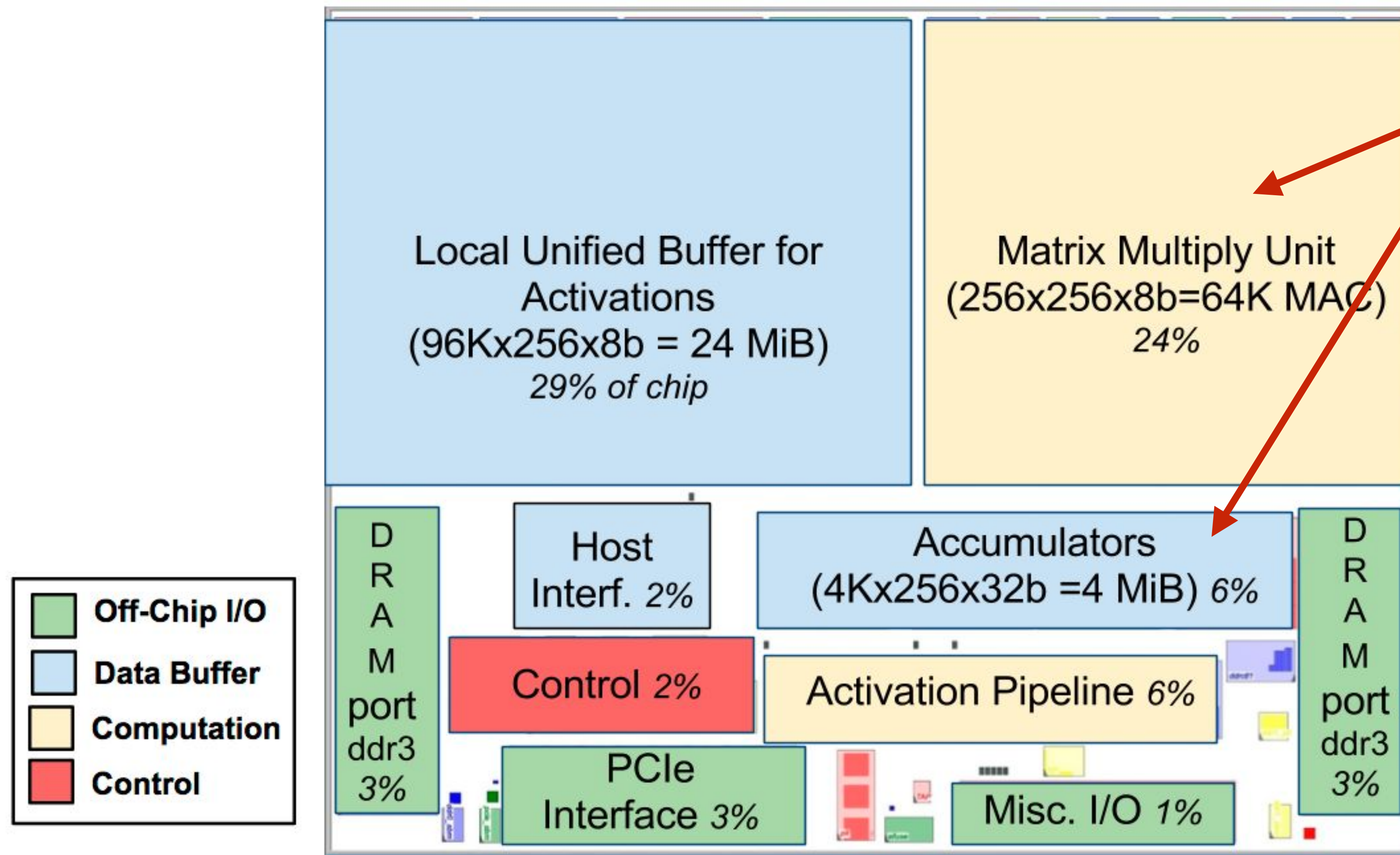
The NVIDIA tensor core approach is an evolutionary design: add DNN-specific instructions to a traditional programmable processor (“evolve, don’t replace”)

Google TPU (version 1)

Google's TPU (v1)



TPU area proportionality

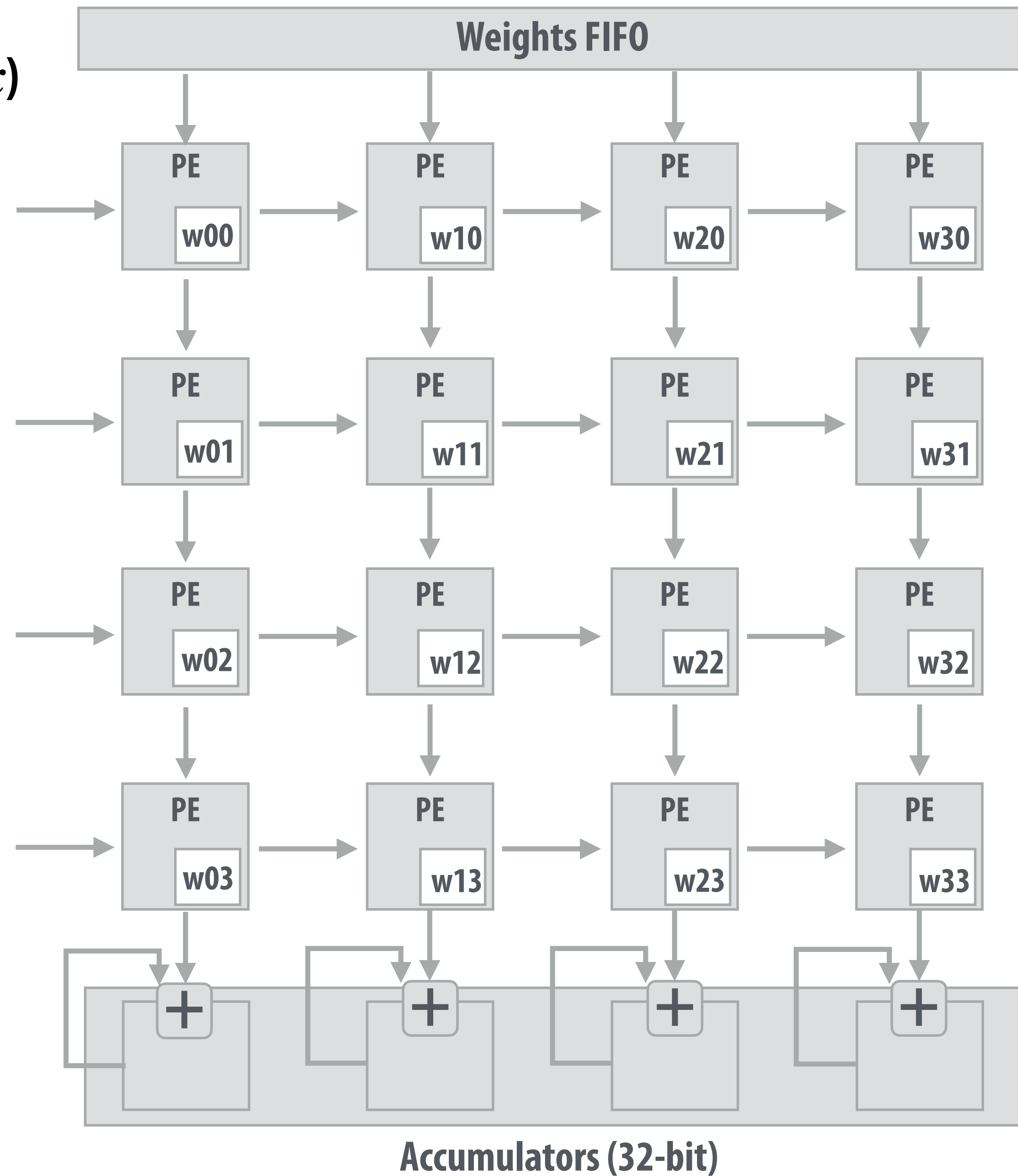


Arithmetic units ~ 30% of chip
Note low area footprint of control

Key instructions:
read host memory
write host memory
read weights
matrix_multiply / convolve
activate

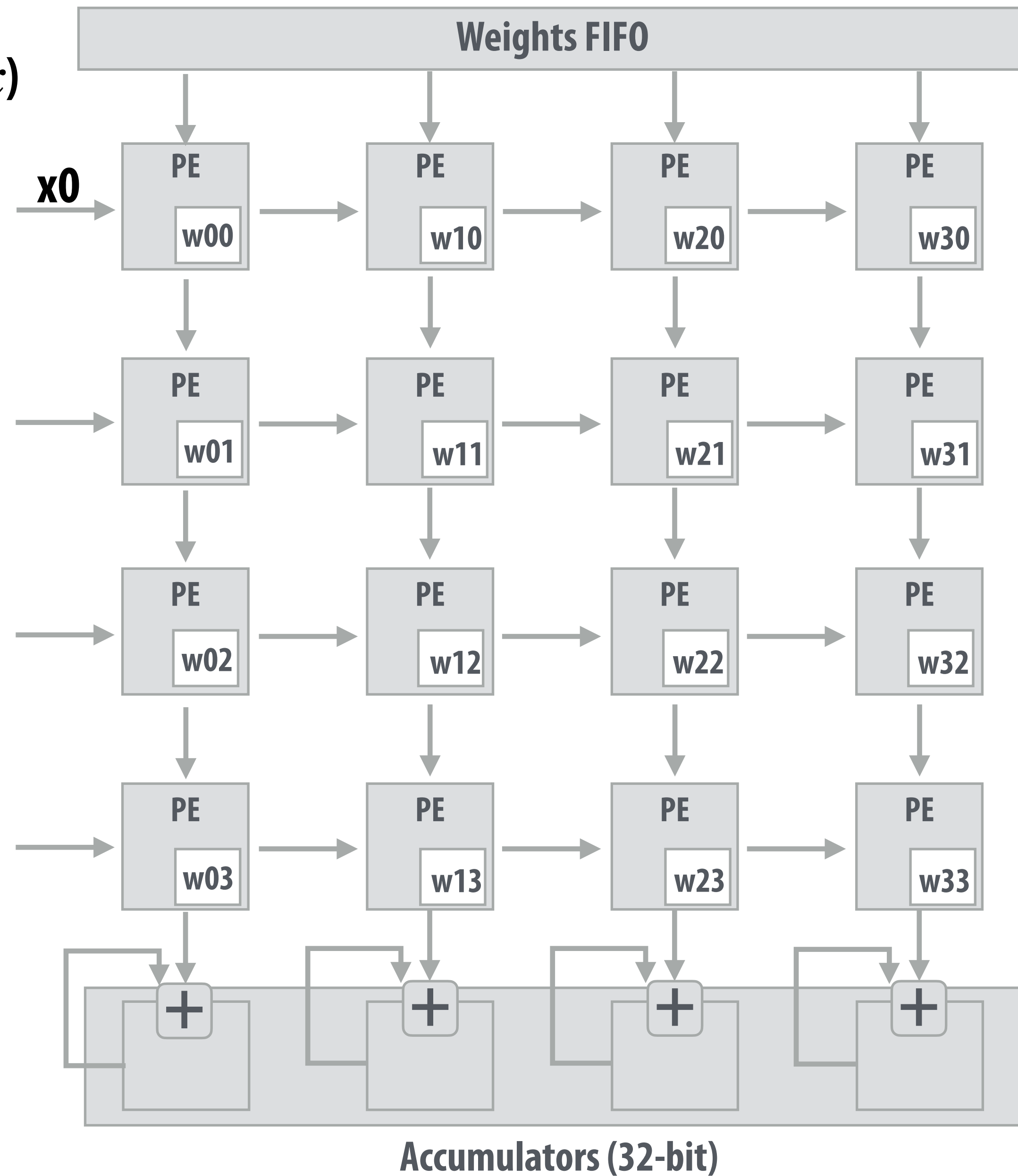
Systemic array

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



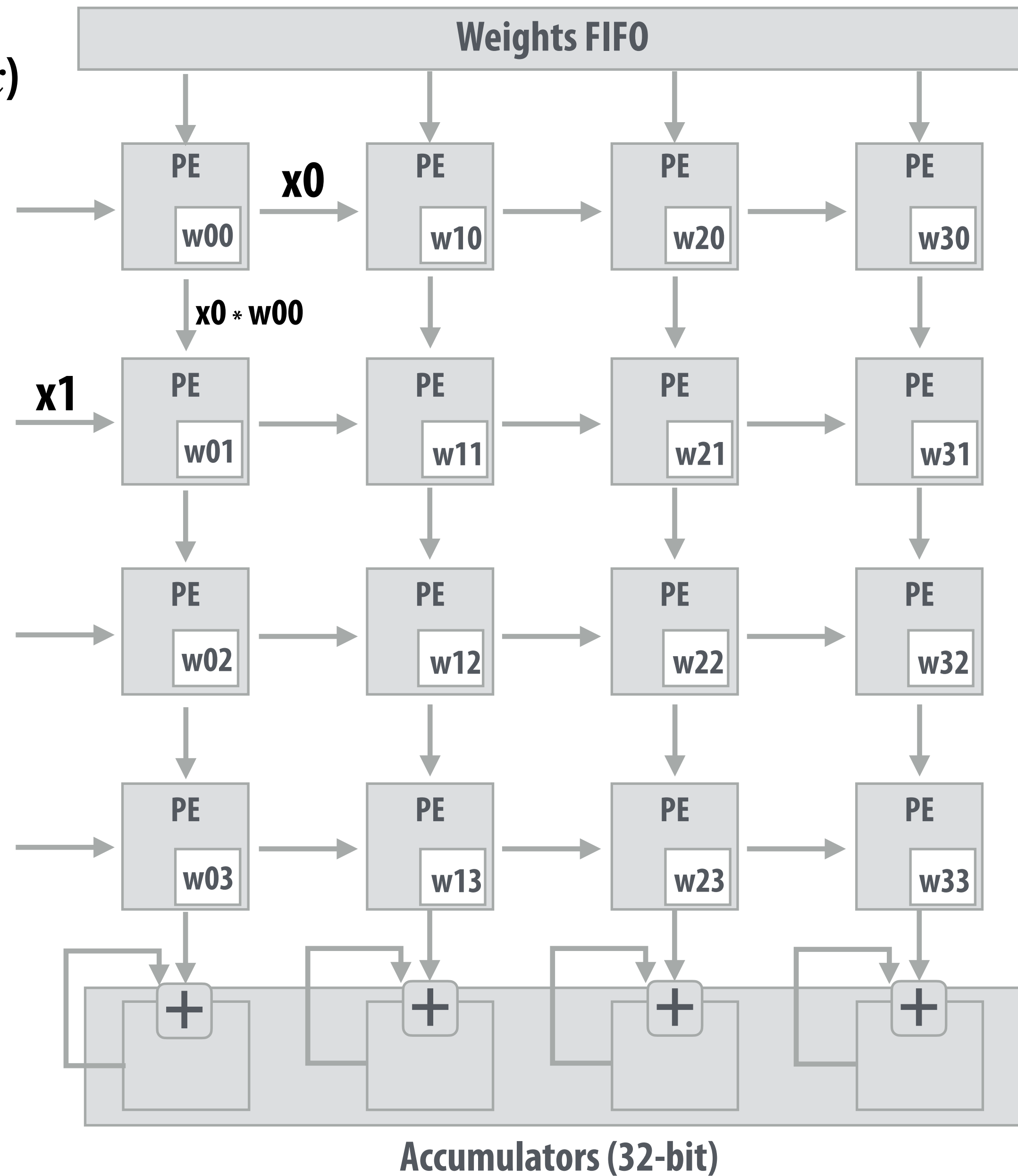
Systemic array

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



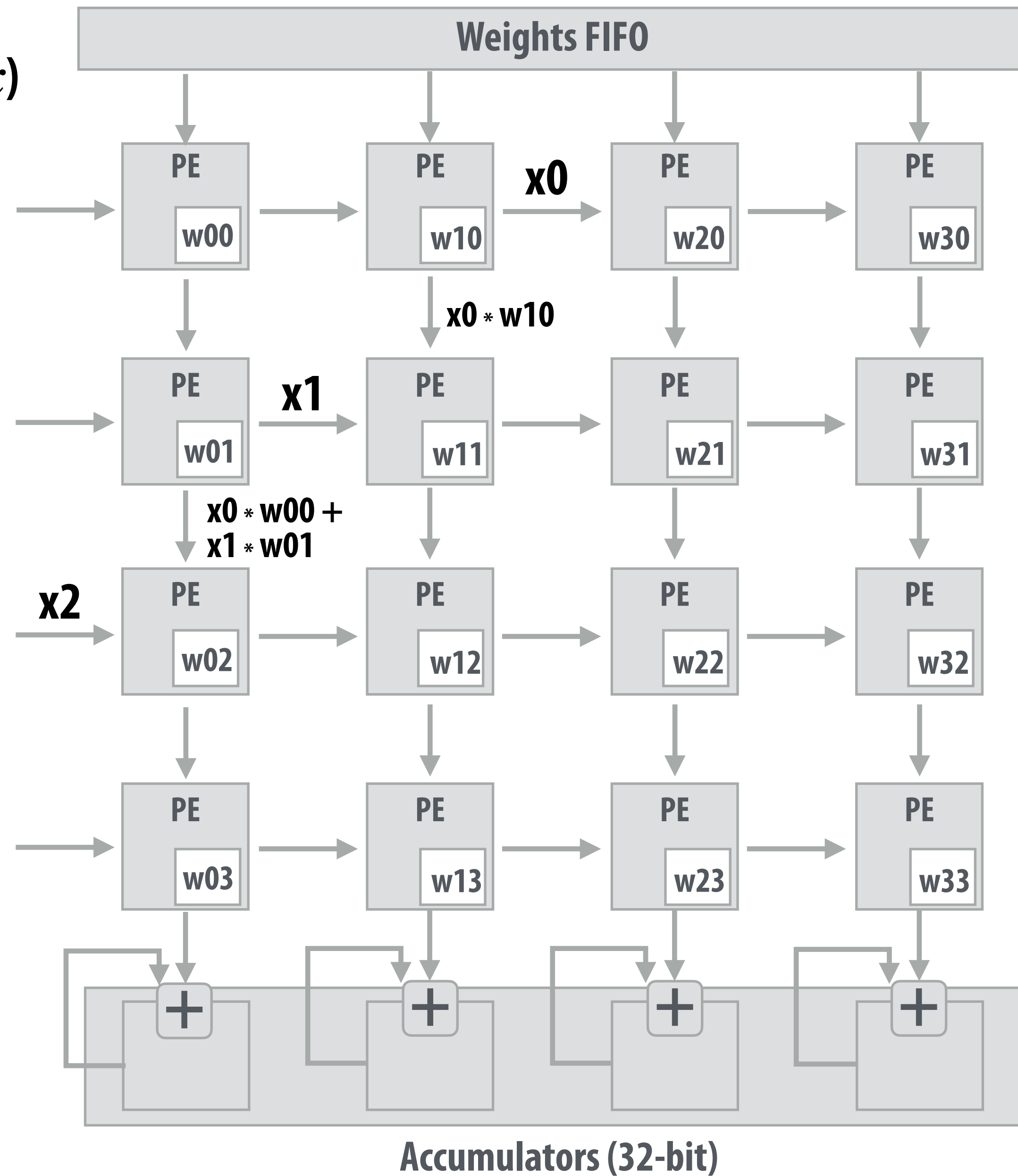
Systemic array

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



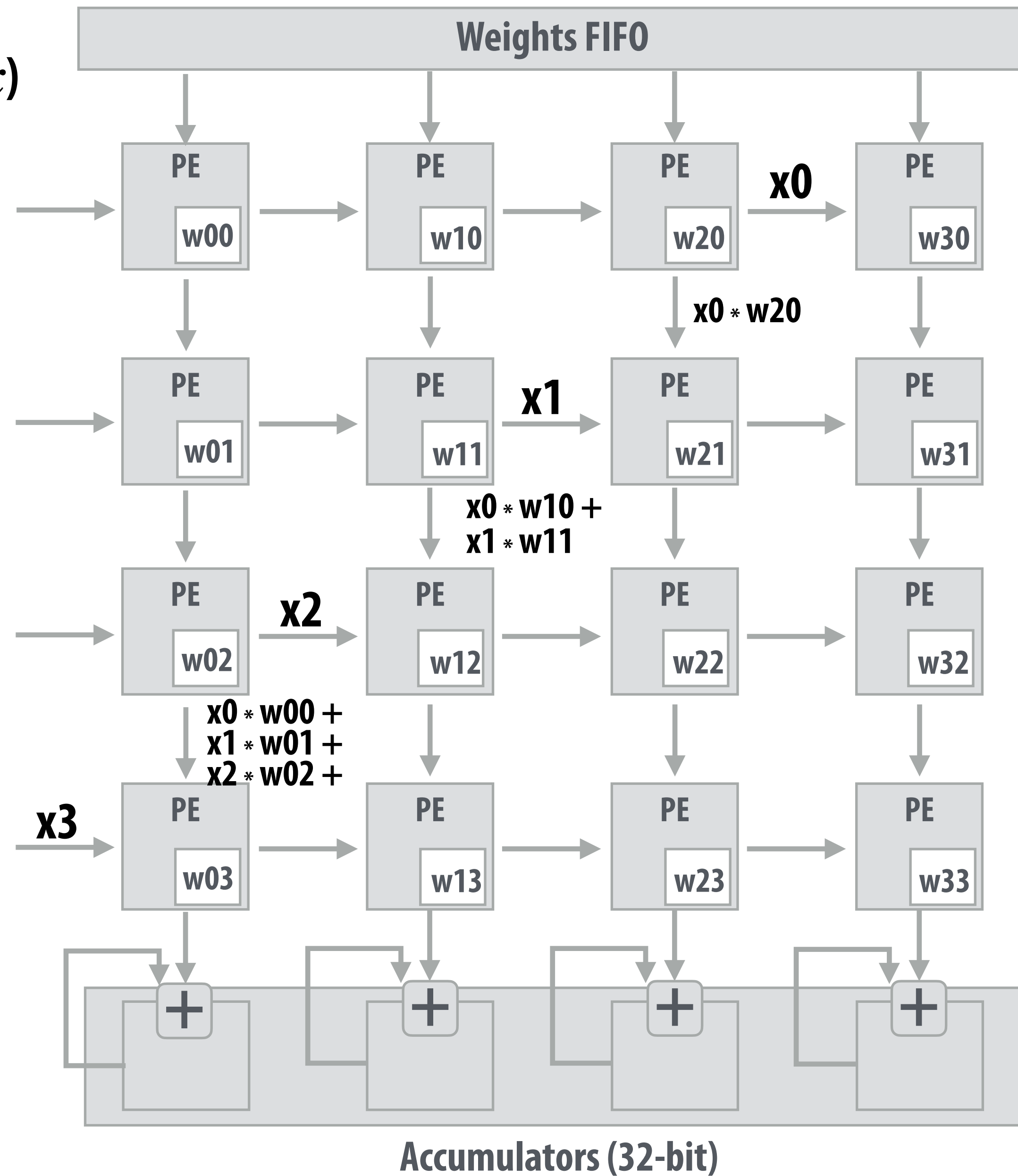
Systemic array

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



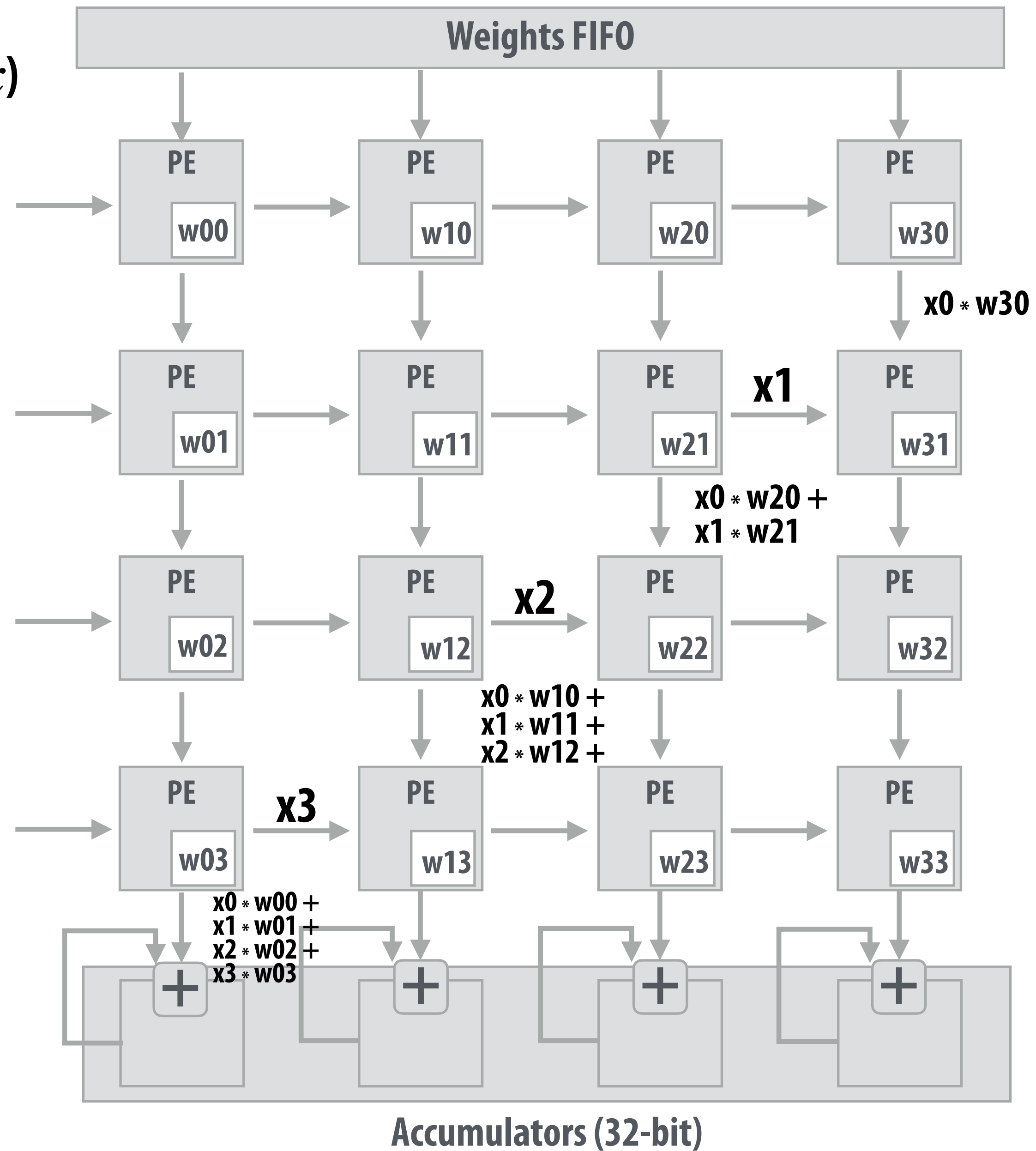
Systolic array

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



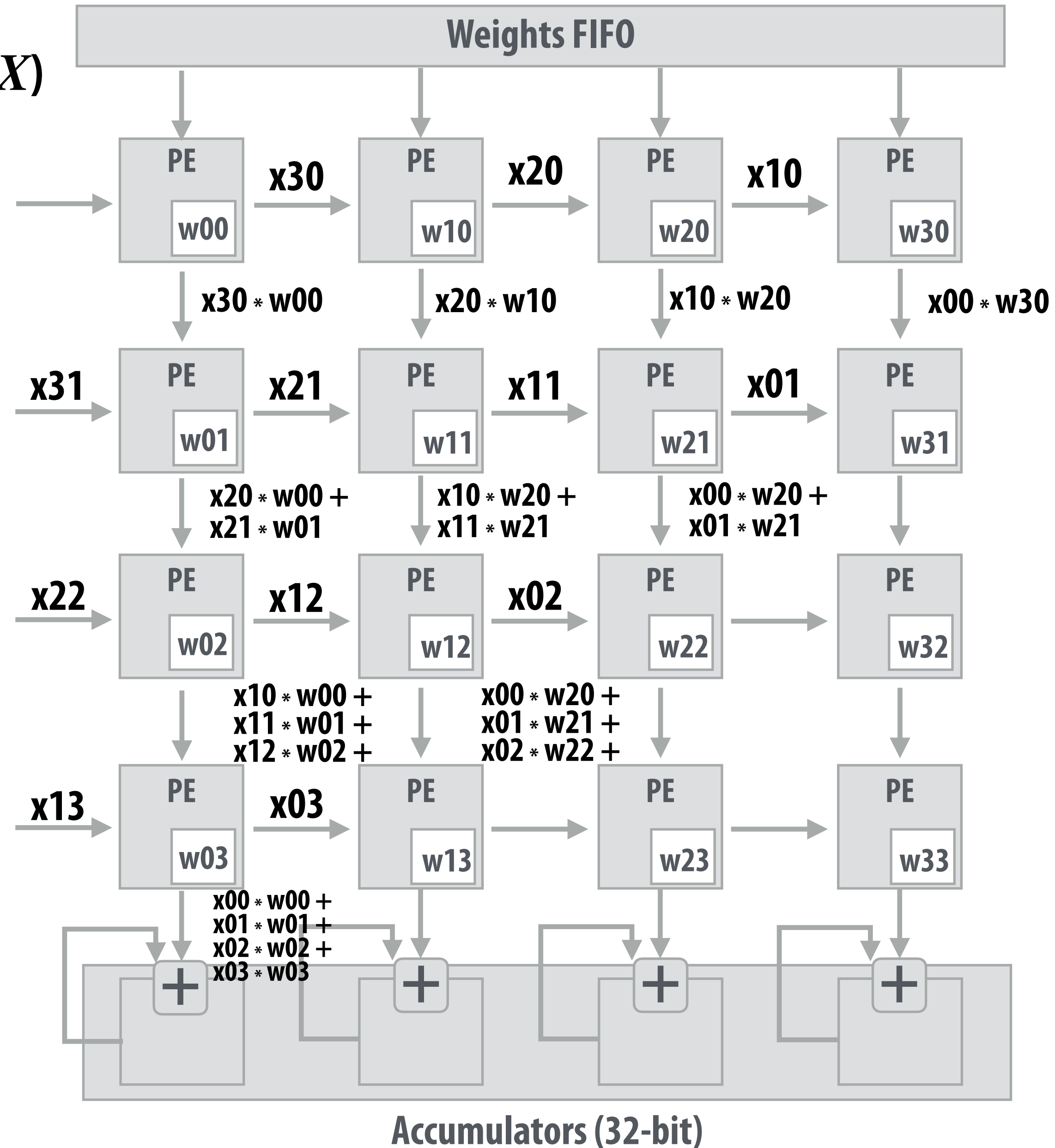
Systemic array

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



Systemic array

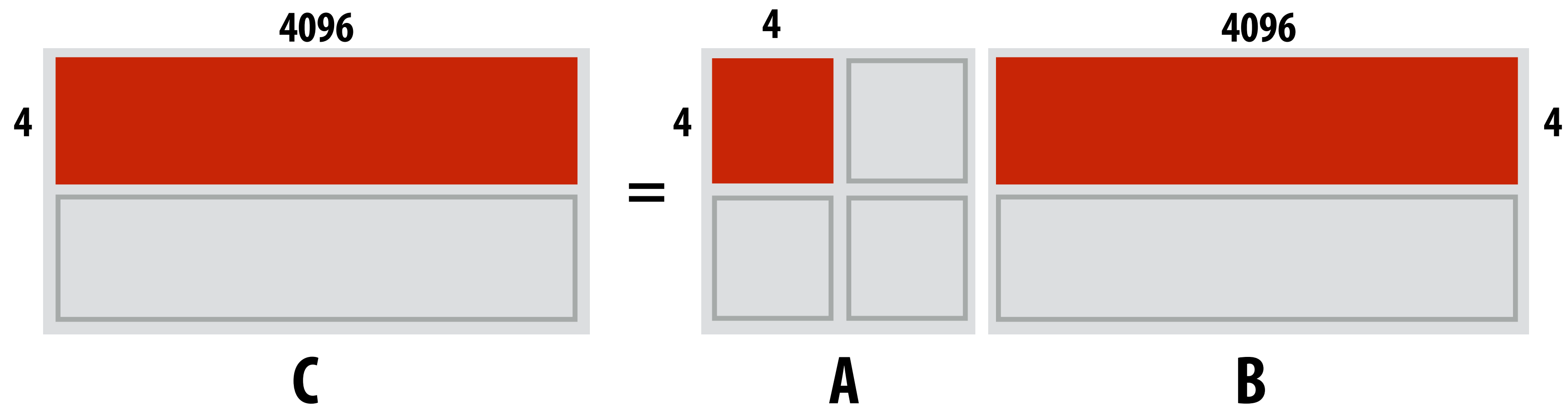
(matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

Building larger matrix-matrix multiplies

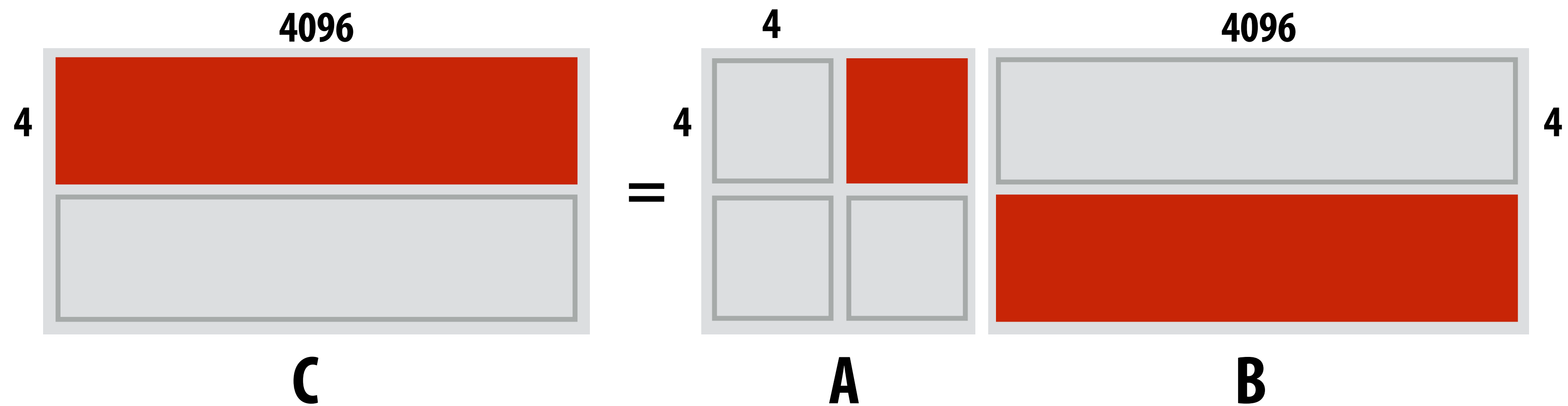
Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



Assume 4096 accumulators

Building larger matrix-matrix multiplies

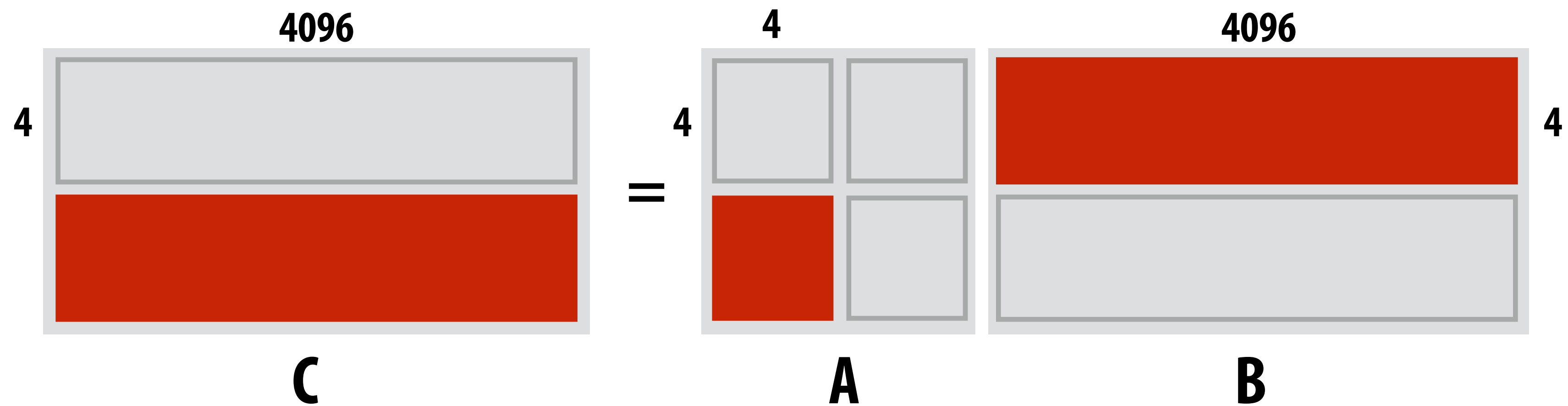
Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



Assume 4096 accumulators

Building larger matrix-matrix multiplies

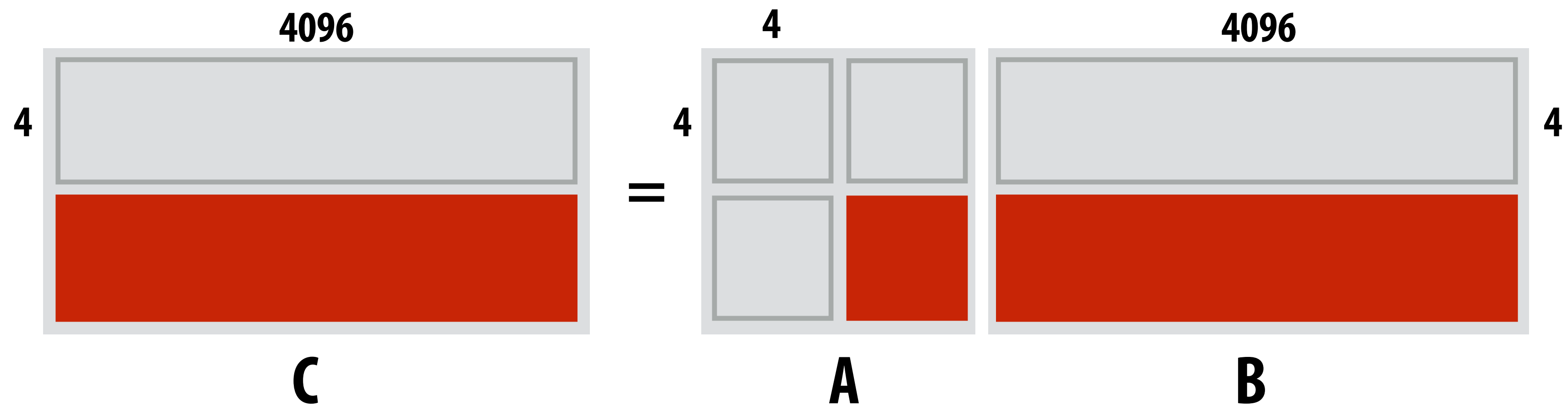
Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



Assume 4096 accumulators

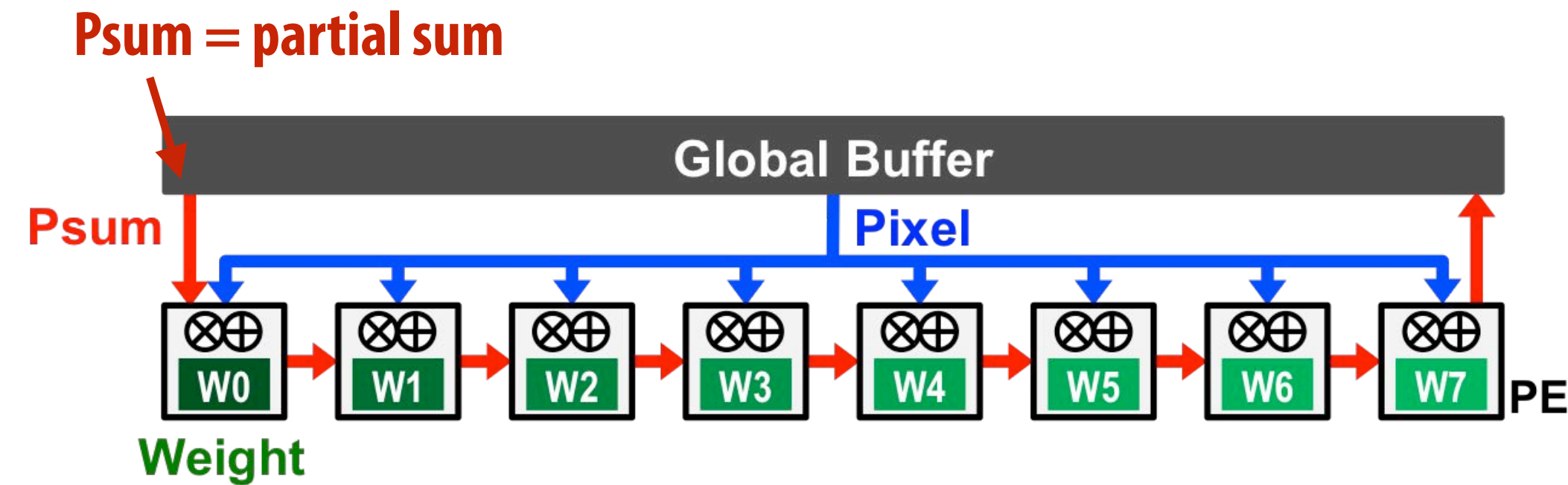
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



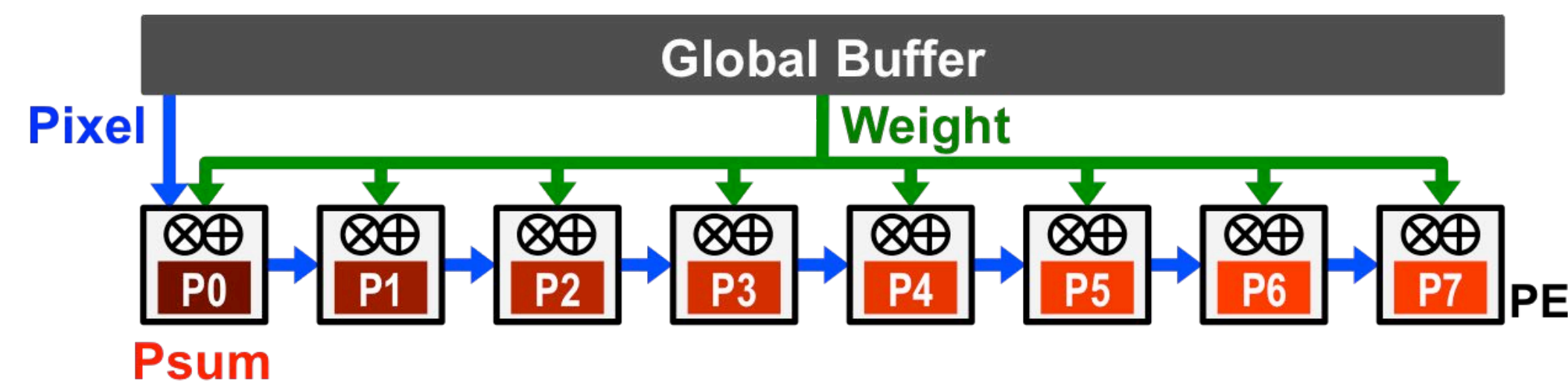
Assume 4096 accumulators

Alternative scheduling strategies



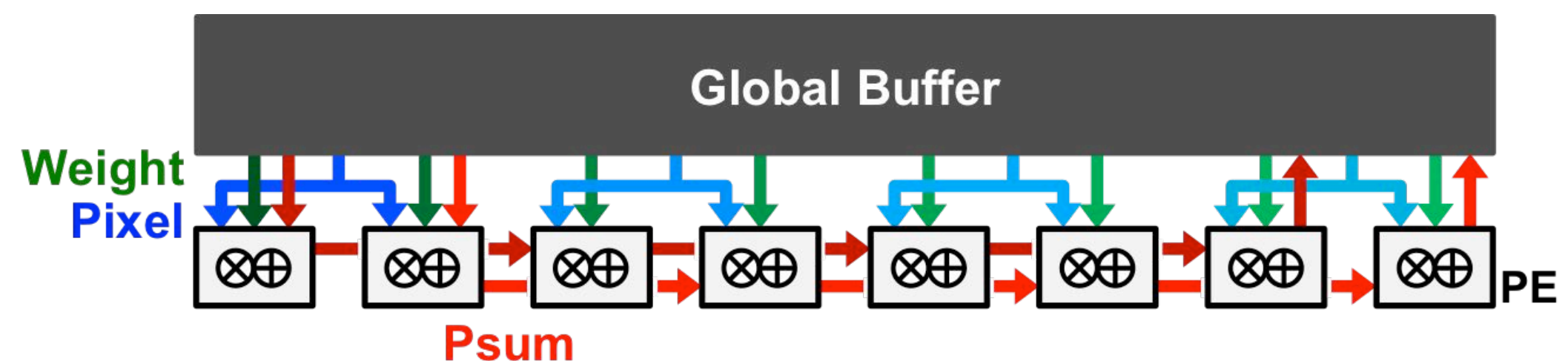
(a) Weight Stationary

TPU (v1) was “weight stationary”:
 weights kept in register at PE
 each PE gets different pixel
 partial sum pushed through array (array has one output)



(b) Output Stationary

“Output stationary”:
 each PE computes one output
 push input pixel through array
 each PE gets different weight
 each PE accumulates locally into output



(c) No Local Reuse

Takeaway: many DNN accelerators can be characterized by the data flow of input activations, weights, and outputs through the machine. (Just different “schedules”!)

Input stationary design (dense 1D conv example)

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

Assume:

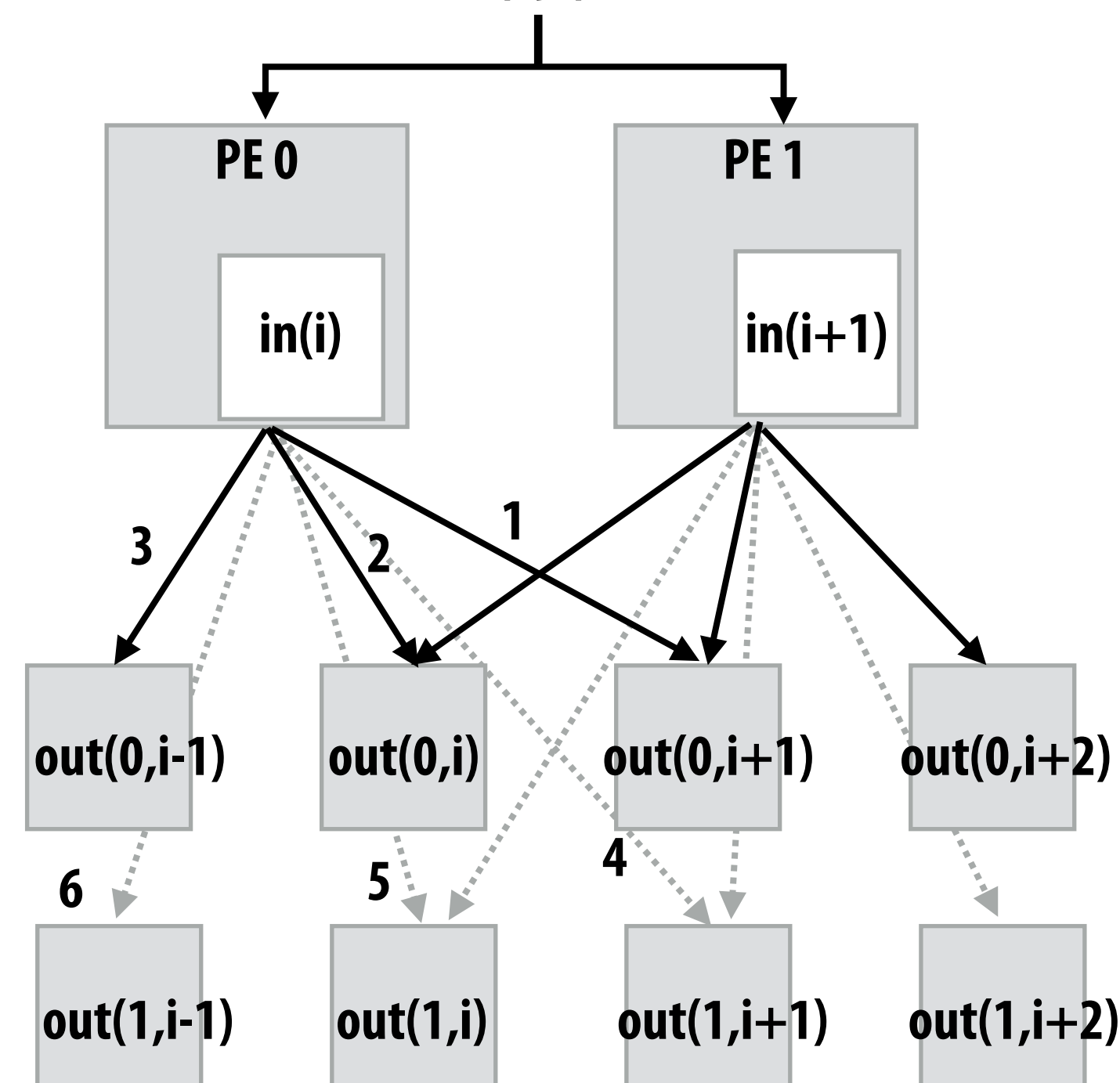
1D input/output

3-wide filters

2 output channels (K=2)

Stream Order	Weight
6	$w(1,2)$
5	$w(1,1)$
4	$w(1,0)$
3	$w(0,2)$
2	$w(0,1)$
1	$w(0,0)$

**Stream of weights
(2 1D filters of size 3)**



**Processing elements
(implement multiply)**

**Accumulators
(implement +=)**

Summary

- **Today's modern DNNs boil down to sequences of (ideally small) matrix multiplies**
- **Blocking/fusion critical to modern performance**
- **Accelerator hardware improves energy**
 - **in the form of special instructions (e.g., NVIDIA tensor core)**
 - **Or novel chips... (e.g., Google TPU)**
- **Next time we'll talk a bit about the impacts of low-precision on efficiency**

A few comments on projects

Projects

- **Team size**
 - **Teams of 2 (no questions asked)**
 - **Teams of 3-4 (possible, but need permission and a clear description of roles)**
- **Topics: must meet two requirements**
 - **You are really excited to work a lot on it**
 - **Can relate the key challenge to intellectual themes of the course**
- **Presentations on Tues/Wed of the last day of the quarter — times TBD**