Lecture 10:

Efficiently Evaluating DNNs

Parallel Computing
Stanford CS149, Fall 2023
Efficiency challenge

Many DNN topologies
(Many variants on common backbones)

<table>
<thead>
<tr>
<th>Table 1: MobileNet Body Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Conv1</td>
</tr>
<tr>
<td>Conv2</td>
</tr>
<tr>
<td>Conv3</td>
</tr>
<tr>
<td>Conv4</td>
</tr>
<tr>
<td>Conv5</td>
</tr>
<tr>
<td>Conv6</td>
</tr>
<tr>
<td>Conv7</td>
</tr>
<tr>
<td>Conv8</td>
</tr>
<tr>
<td>Conv9</td>
</tr>
<tr>
<td>Conv10</td>
</tr>
<tr>
<td>Conv11</td>
</tr>
<tr>
<td>Conv12</td>
</tr>
</tbody>
</table>

Figure 3: Example network architectures for ImageNet.

Many Target Devices

Figure 10. The schema for the Inception-v4 network. For the non-linearities, we used ReLU [1] and GEMM but require an initial reordering in memory to stack the convolution into dense computation. 

Figure 9. The overall schema of the Inception-v4 network. For the convolution, we used an initial reordering in memory to stack the convolution into dense computation. 

Figure 11. The schema for the Inception-v4 network. For the convolution, we used an initial reordering in memory to stack the convolution into dense computation. 

Figure 12. The schema for the Inception-v4 network. For the convolution, we used an initial reordering in memory to stack the convolution into dense computation.
Mini-intro/review: Convolutional Neural Networks
Consider the following expression

\[
\max( \max(0, (a \times b) + (c \times d)) + (e \times f) + (g \times h), i \times j)
\]
What is a deep neural network?

A basic unit:
Unit with $n$ inputs described by $n+1$ parameters (weights + bias)

$$f \left( \sum_{i} x_i w_i + b \right)$$

Example: rectified linear unit (ReLU)
$$f(x) = \max(0, x)$$

Basic computational interpretation:
It is just a circuit!

Machine learning interpretation:
Binary classifier: interpret output as the probability of one class
$$f(x) = \frac{1}{1 + e^{-x}}$$
Fully connected layer as matrix-vector product

Assume $f()$ is the element-wise max function (ReLU)
2D convolution: what does this C code do?

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

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

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```
The code on the previous slide performed a 3x3 blur
Image convolution (3x3 conv)

```c
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

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)
(3D because they operate on RGB)

96 11x11x3 filters

96 responses (normalized)
Applying many filters to an image at once

Input: image (single channel): \( W \times H \)

3x3 spatial convolutions on image
3x3 x num\_filters weights

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

Output: filter responses
\( W \times H \times \text{num\_filters} \)

Filter response maps
(num\_filters of them)
Adding additional layers

Input: image (single channel) $W \times H$

$3 \times 3$ spatial convolutions $3 \times 3 \times \text{num\_filters}$ weights

Output: filter responses $W \times H \times \text{num\_filters}$

Each filter described by unique set of weights (responds to different image phenomena)

Filter responses

After ReLU $W \times H \times \text{num\_filters}$

After Pool $W/2 \times H/2 \times \text{num\_filters}$ (max response in $2 \times 2$ region)

Note data reduction as a result of “pooling”
More recent image understanding networks

- Inception (GoogleLeNet)
- ResNet (34 layer version)
- Fully Convolutional Network for image segmentation

We evaluate both top-1 and top-5 error rates.
Efficiently implementing convolution layers
Approach 1:
Algorithmic innovation: more efficient topologies
ResNet

Figure 3. Example network architectures for ImageNet.

- A plain network with 34 parameter layers (3.6 billion FLOPs).
- A residual network with 34 parameter layers (3.6 billion FLOPs) as a reference.

The tailed architectures.

The 18-layer plain net is of a similar form. See Table 4. Experiments at multiple scales (images are resized such that the shorter dimension is 256).

We use a weight decay of 0.0001 and a momentum of 0.9. We use SGD with a mini-batch size of 256. The learning rate is reduced by a factor of 10 before activation, following the practice in 

\[\text{normalization (BN)}\]

\[\] and train all plain/residual nets from scratch.

Our implementation for ImageNet follows the practice of grid-reduction module.

**4. Experiments**

A detailed structure of the various components. Please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed modules. Please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components. Please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components. Please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

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

Stanford CS149, Fall 2023
MobileNet

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

<table>
<thead>
<tr>
<th>Type / Stride</th>
<th>Filter Shape</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv / s2</td>
<td>3 x 3 x 3 x 32</td>
<td>224 x 224 x 3</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 32 dw</td>
<td>112 x 112 x 32</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 32 x 64</td>
<td>112 x 112 x 32</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 64 dw</td>
<td>112 x 112 x 64</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 64 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 128 dw</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 128 x 128</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 128 dw</td>
<td>56 x 56 x 128</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 128 x 256</td>
<td>28 x 28 x 128</td>
</tr>
<tr>
<td>Conv dw / s1</td>
<td>3 x 3 x 256 dw</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 256 x 256</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 256 dw</td>
<td>28 x 28 x 256</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 256 x 512</td>
<td>14 x 14 x 256</td>
</tr>
<tr>
<td>5 x Conv dw / s1</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 512</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 512 dw</td>
<td>14 x 14 x 512</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 512 x 1024</td>
<td>7 x 7 x 512</td>
</tr>
<tr>
<td>Conv dw / s2</td>
<td>3 x 3 x 1024 dw</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Conv / s1</td>
<td>1 x 1 x 1024 x 1024</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>Avg Pool / s1</td>
<td>Pool 7 x 7</td>
<td>7 x 7 x 1024</td>
</tr>
<tr>
<td>FC / s1</td>
<td>1024 x 1000</td>
<td>1 x 1 x 1024</td>
</tr>
<tr>
<td>Softmax / s1</td>
<td>Classifier</td>
<td>1 x 1 x 1000</td>
</tr>
</tbody>
</table>

Image classification (ImageNet)
Comparison to Common DNNs

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 MobileNet-224</td>
<td>70.6%</td>
<td>569</td>
<td>4.2</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>69.8%</td>
<td>1550</td>
<td>6.8</td>
</tr>
<tr>
<td>VGG 16</td>
<td>71.5%</td>
<td>15300</td>
<td>138</td>
</tr>
</tbody>
</table>

Image classification (ImageNet)
Comparison to Other Compressed DNNs

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50 MobileNet-160</td>
<td>60.2%</td>
<td>76</td>
<td>1.32</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>57.5%</td>
<td>1700</td>
<td>1.25</td>
</tr>
<tr>
<td>AlexNet</td>
<td>57.2%</td>
<td>720</td>
<td>60</td>
</tr>
</tbody>
</table>
In this section we report our results and comparisons. We analysed the following DDNs: AlexNet, VGG, ResNet, BN-NIN, Inception-v3, ResNet-18, and ENet. For inference time and memory usage measurements we have used Torch7 (Collobert et al., 2011) and CUDA (Chetlur et al., 2014) respectively. We use a 64-bit ARM A57 CPU, a 1 T-Flop/s 256-core NVIDIA Maxwell GPU and 4 GB LPDDR4 to run our experiments.

As the spoiler in section 3.1 gave already away, the linear nature of the accuracy vs. the number of operations is well stated. We show that accuracy and inference time are in a hyperbolic relationship: a little increment in the number of parameters leads to a greater increment in inference time. As the number of parameters increases, the accuracy starts to level off. As a result, we can define an upper bound on the maximum achievable accuracy and model complexity, in terms of operations counts. Therefore, we need to move to more efficient architectures for practical application, and optimisation of the often-limited resources in actual deployments, which lead us to the creation of ENet — or Efficient-Network — for ImageNet.

ENet (Paszke et al., 2017) is the best architecture in terms of parameters space utilisation, squeezing all their neurons to learn the given task, and are the winners of this section. ENet (Paszke et al., 2017) is an efficient neural network that has a low complexity, with an upper bound on the maximum achievable accuracy and model complexity. ENet is a key architecture for efficient neural networks, and can be used in practical applications, with an upper bound on the maximum achievable accuracy and model complexity.

Moreover, ENet (Paszke et al., 2017) is the best architecture in terms of parameters space utilisation, squeezing all their neurons to learn the given task, and are the winners of this section. ENet (Paszke et al., 2017) is an efficient neural network that has a low complexity, with an upper bound on the maximum achievable accuracy and model complexity. ENet is a key architecture for efficient neural networks, and can be used in practical applications, with an upper bound on the maximum achievable accuracy and model complexity.

Finally, we show that ENet is the best architecture in terms of parameters space utilisation, squeezing all their neurons to learn the given task, and are the winners of this section. ENet (Paszke et al., 2017) is an efficient neural network that has a low complexity, with an upper bound on the maximum achievable accuracy and model complexity. ENet is a key architecture for efficient neural networks, and can be used in practical applications, with an upper bound on the maximum achievable accuracy and model complexity.
## Improving accuracy/cost (image classification)

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

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

* 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

\[
\begin{bmatrix}
X_{00} & X_{01} & X_{02} & X_{03} & \ldots \\
X_{10} & X_{11} & X_{12} & X_{13} & \ldots \\
X_{20} & X_{21} & X_{22} & X_{23} & \ldots \\
X_{30} & X_{31} & X_{32} & X_{33} & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

Note: 0-pad matrix

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

\[
\begin{bmatrix}
0 & 0 & 0 & x00 & x01 & 0 & x10 & x11 \\
0 & 0 & 0 & x00 & x01 & x02 & x10 & x11 & x12 \\
0 & 0 & 0 & x01 & x02 & x03 & x11 & x12 & x13 \\
\vdots \\
x00 & x01 & x02 & x10 & x11 & x12 & x20 & x21 & x22 \\
\vdots
\end{bmatrix}
\]
3x3 convolution as matrix-vector product (“explicit gemm”)
Multiple convolutions on multiple input channels

For each filter, sum responses over input channels

Equivalent to \((3 \times 3 \times \text{num\_channels})\) 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.

<table>
<thead>
<tr>
<th>Convolution</th>
<th>GE-MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = CONV (x, w)$</td>
<td>$C = GEMM (A, B)$</td>
</tr>
</tbody>
</table>

- $x[N, H, W, C]$: 4D activation tensor
- $y[N, P, Q, K]$: 4D output tensor
- $A[NPQ, RSC]$: 2D convolution matrix
- $B[RSC, K]$: 2D filter matrix
- $C[NPQ, K]$: 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)
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**

```c
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

```c
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

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
for (int j=0; j<\text{BLOCKSIZE}_J; j++)
  for (int i=0; i<\text{BLOCKSIZE}_I; i++) {
    float C\_scalar = C[j\_block+j][i\_block+i];
    // C\_scalar += \text{dot}(\text{row of } A, \text{row of } B)
    for (int k=0; k<\text{BLOCKSIZE}_K; k+=\text{SIMD\_WIDTH}) {
      C\_scalar += \text{simd\_dot}(&A[j\_block+j][k\_block+k], \text{vec\_load}(&B\_trans[i\_block+i][k\_block+k]));
    }
    C[j\_block+j][i\_block+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**
// 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]);

        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

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

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
NVIDIA CUTLASS

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

Fast (in-shared memory) GEMM

Fast WARP level GEMMs

Iterators for fast block loading/tensor indexing

Tensor reductions

Etc.
Recall: NVIDIA V100 GPU (80 SMs)

Many processing units and many tensor cores.

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

<table>
<thead>
<tr>
<th>L2 Cache (6 MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>900 GB/sec</td>
</tr>
<tr>
<td>(4096 bit interface)</td>
</tr>
<tr>
<td>GPU memory (HBM)</td>
</tr>
<tr>
<td>(16 GB)</td>
</tr>
</tbody>
</table>
Higher performance with “more work”

N=1, P=Q=64 case:
64 x 64 x 128 x 1 = 524K outputs = 2 MB of output data (float32)

N=32, P=Q=256 case:
256 x 256 x 128 x 32 = 256M outputs = 1 GB of output data (float32)
Direct implementation

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

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

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

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

- Direct convolution using work-efficient Winograd convolutions

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

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

\[
\begin{align*}
m_1 &= (x_0 - x_1)w_0 \\
m_2 &= (x_1 + x_2)\left(\frac{w_0 + w_1 + w_2}{2}\right) \\
m_3 &= (x_2 - x_1)\left(\frac{w_0 - w_1 + w_2}{2}\right) \\
m_4 &= (x_1 - x_3)w_2
\end{align*}
\]

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)
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

<table>
<thead>
<tr>
<th>TensorFlow op</th>
<th>NN ops</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf.nn.avg_pool</td>
<td>AvgPool</td>
<td>Performs average pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.avg_pool_3d</td>
<td>AvgPool3D</td>
<td>Performs 3D average pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.bias_add</td>
<td>BiasAdd</td>
<td>Computes gradients of average pooling function.</td>
</tr>
<tr>
<td>tf.nn.bias_add_grad</td>
<td>BiasAddGrad</td>
<td>The backward operation for &quot;BiasAdd&quot; on the &quot;bias&quot; tensor.</td>
</tr>
<tr>
<td>tf.nn.conv2d</td>
<td>Conv2D</td>
<td>Computes a 2-D convolution given 4-D input and filter tensors.</td>
</tr>
<tr>
<td>tf.nn.conv3d_backprop_filter</td>
<td>Conv3D</td>
<td>Computes a 3-D convolution given 5-D input and filter tensors.</td>
</tr>
<tr>
<td>tf.nn.conv3d_backprop_input</td>
<td>Conv3D</td>
<td>Computes the gradients of a 3-D convolution with respect to the input.</td>
</tr>
<tr>
<td>tf.nn.conv2d_backprop_filter</td>
<td>Conv2DBackpropFilter</td>
<td>Computes the gradients of convolution with respect to the filter.</td>
</tr>
<tr>
<td>tf.nn.conv2d_backprop_input</td>
<td>Conv2DBackpropInput</td>
<td>Computes the gradients of convolution with respect to the input.</td>
</tr>
<tr>
<td>tf.nn.depthwise_conv2d</td>
<td>DepthwiseConv2D</td>
<td>Computes a 2-D depthwise convolution given 4-D input and 3-D filter tensors.</td>
</tr>
<tr>
<td>tf.nn.depthwise_conv2d_backprop_filter</td>
<td>DepthwiseConv2DBackpropFilter</td>
<td>Computes the gradients of depthwise convolution with respect to the filter.</td>
</tr>
<tr>
<td>tf.nn.depthwise_conv2d_backprop_input</td>
<td>DepthwiseConv2DBackpropInput</td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.dilation_2d</td>
<td>Dilate2D</td>
<td>Computes the grayscale dilation of 4-D input and 3-D filter.</td>
</tr>
<tr>
<td>tf.nn.dilation_2d_backprop_filter</td>
<td>Dilate2DBackpropFilter</td>
<td>Computes the gradient of morphological 2-D dilation filter.</td>
</tr>
<tr>
<td>tf.nn.dilation_2d_backprop_input</td>
<td>Dilate2DBackpropInput</td>
<td>Computes the gradient of morphological 2-D dilation input.</td>
</tr>
<tr>
<td>tf.nn.elu</td>
<td>Elu</td>
<td>Computes exponential linear $\exp(\text{features}) - 1$ otherwise.</td>
</tr>
<tr>
<td>tf.nn.fractional_avg_pool</td>
<td>FractionalAvgPool</td>
<td>Performs fractional average pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.fractional_max_pool</td>
<td>FractionalMaxPool</td>
<td>Performs fractional max pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.fused_batch_norm</td>
<td>FusedBatchNorm</td>
<td>Batch normalization.</td>
</tr>
<tr>
<td>tf.nn.fused_batch_norm_grad</td>
<td>FusedBatchNormGrad</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tf.nn.fused_batch_norm_grad_v2</td>
<td>FusedBatchNormGradV2</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tf.nn.fused_batch_norm_grad_v3</td>
<td>FusedBatchNormGradV3</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tf.nn.fused_batch_norm_v2</td>
<td>FusedBatchNormV2</td>
<td>Batch normalization.</td>
</tr>
<tr>
<td>tf.nn.fused_conv2d</td>
<td>FusedConv2D</td>
<td>Performs a padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td>tf.nn.fused_resize_and_padding_conv2d</td>
<td>FusedResizeAndPaddingConv2D</td>
<td>Performs a resize and padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td>tf.nn.in_top_k</td>
<td>InTopK</td>
<td>Says whether the targets are in the top K predictions.</td>
</tr>
<tr>
<td>tf.nn.l2_loss</td>
<td>L2Loss</td>
<td>Computes the variance of the input.</td>
</tr>
<tr>
<td>tf.nn.lrn</td>
<td>LRN</td>
<td>Local Response Normalization.</td>
</tr>
<tr>
<td>tf.nn.log_softmax</td>
<td>LogSoftmax</td>
<td>Computes log softmax activations.</td>
</tr>
<tr>
<td>tf.nn.max_pool</td>
<td>MaxPool</td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d</td>
<td>MaxPool3D</td>
<td>Performs 3D max pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_grad</td>
<td>MaxPool3DGrad</td>
<td>Computes gradients of 3D max pooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_grad_v2</td>
<td>MaxPool3DGradV2</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_grad_v3</td>
<td>MaxPool3DGradV3</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_grad_v4</td>
<td>MaxPool3DGradV4</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_grad_with_argmax</td>
<td>MaxPool3DGradWithArgmax</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_3d_with_argmax</td>
<td>MaxPool3DWithArgmax</td>
<td>Computes gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tf.nn.max_pool_v2</td>
<td>MaxPoolV2</td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td>tf.nn.nth_element</td>
<td>NthElement</td>
<td>Finds values of the n-th order statistic for the last dimension.</td>
</tr>
<tr>
<td>tf.nn.quantized_avg_pool</td>
<td>QuantizedAvgPool</td>
<td>Produces the average pool of the input tensor for quantized types.</td>
</tr>
<tr>
<td>tf.nn.quantized_batch_norm</td>
<td>QuantizedBatchNorm</td>
<td>Quantized Batch normalization.</td>
</tr>
<tr>
<td>tf.nn.quantized_bias_add</td>
<td>QuantizedBiasAdd</td>
<td>Adds Tensor <code>bias</code> to Tensor <code>input</code> for Quantized types.</td>
</tr>
<tr>
<td>tf.nn.quantized_conv2d</td>
<td>QuantizedConv2D</td>
<td>Computes a 2D convolution given quantized 4D input and filter tensors.</td>
</tr>
<tr>
<td>tf.nn.quantized_max_pool</td>
<td>QuantizedMaxPool</td>
<td>Produces the max pool of the input tensor for quantized types.</td>
</tr>
</tbody>
</table>
Libraries offering high-performance implementations of key DNN layers

**TensorFlow**

- `tf.nn.avg_pool`: Performs average pooling on the input.
- `tf.nn.max_pool`: Performs max pooling on the input.
- `tf.nn.avg_pool3d`: Computes gradients of average pooling function.
- `tf.nn.bias_add`: Adds bias to values.
- `tf.nn.max_pool3d`: The backward operation for `bias_add` on the `bias`.
- `tf.nn.conv2d`: Computes a 2-D convolution given 4D input and filter.
- `tf.nn.conv2d_backprop_filter`: Computes the gradients of convolution with respect to the filter.
- `tf.nn.conv2d_backprop_input`: Computes the gradients of convolution with respect to the input.
- `tf.nn.conv3d`: Computes a 3-D convolution given 5D input and 5D filter.
- `tf.nn.conv3d_backprop_filter`: Computes the gradients of 3-D convolution with respect to the filter.
- `tf.nn.conv3d_backprop_input`: Computes the gradients of 3-D convolution with respect to the input.
- `tf.nn.data_format_dim_map`: Returns the dimension index in the destination data format.
- `tf.nn.data_format_image_permute`: Permutes input tensor from src_format to dst_format.
- `tf.nn.depthwise_conv2d`: Performs a 2-D depthwise convolution given 4D input and 4D filter.
- `tf.nn.depthwise_conv2d_native`: Performs a 2-D depthwise convolution given 4D input and 4D filter.
- `tf.nn.dilation2d`: Performs a 2-D dilation of 4D input and 4D filter.
- `tf.nn.dilation2d_native`: Performs a 2-D dilation of 4D input and 4D filter.
- `tf.nn.max_pool`: Performs 2-D max pooling.
- `tf.nn.relu`: Computes the unreduced ReLU of `input`.
- `tf.nn.relu6`: Computes the unreduced ReLU6 of `input`.
- `tf.nn.softmax`: Computes the softmax of `input`.
- `tf.nn.softmax_cross_entropy_with_logits`: Computes softmax cross-entropy cost.
- `tf.nn.softmax_with_logits`: Computes softmax over all dimensions.
- `tf.nn.sigmoid`: Computes sigmoid activation of `input`.
- `tf.nn.softmax`: Computes softmax over all dimensions.

**NVIDIA cuDNN**

**Intel® oneAPI Deep Neural Network Library**
Example: CUDNN convolution

```c
void 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,
  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:

  ![Diagram](image)

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

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

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

Exercise to class:
How would you “fuse” a max pool operation following this layer (max of 2x2 blocks of output matrix)?
Fusion trick for computing “attention”
Transformer architecture for sequence models

Sequence of tokens in, sequence of tokens out

Example: next word prediction

Let's talk about optimizing the computation of attention in a modern DNN. Let's start with a DNN, which has a simple model, say a some simple examples of a conv
Attention module

Let $S = QK^T \in \mathbb{R}^{N \times N}$

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

Let $O = PV \in \mathbb{R}^{N \times d}$

softmax$(S)$

Is computing softmax over the rows of $S$

For a row $x$:

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

Where:

$$f(x) = \left[ e^{x_1 - m(x)} \quad e^{x_1 - m(x)} \quad \ldots \quad e^{x_B - m(x)} \right]$$

$$m(x) = \max_i (x_i)$$

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

Notes:

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

Naive implementation uses $N^2$ space! Trouble!!!
Computing attention

Let $x = S^i = \text{i}^{\text{th}} \text{row of } S$.
Then define $\text{softmax}(x)$ as:

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

where

- $m(x) = \max_i (x_i)$
- $f(x) = [e^{x_1 - m(x)} \ldots e^{x_B - m(x)}]$
Let’s look into softmax more closely…

\[
\text{softmax}(x) = \frac{f(x)}{l(x)}
\]

Where:
\[
f(x) = \begin{bmatrix} e^{x_1 - m(x)} & e^{x_1 - m(x)} & \ldots & e^{x_B - m(x)} \end{bmatrix}
\]
\[
m(x) = \max_i (x_i)
\]
\[
l(x) = \sum_i f(x)_i = \sum_i e^{x_i - m(x)}
\]

Let’s break vector \( x \) into chunks:
\[
x = \begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}
\]

Now:
\[
m(x) = \max \left( m(x^{(1)}), m(x^{(2)}) \right)
\]
\[
f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix}
\]
\[
l(x) = e^{m(x^{(1)}) - m(x)} l(x^{(1)}) + e^{m(x^{(2)}) - m(x)} l(x^{(2)})
\]

So softmax can be computed in chunks!
Fused attention

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

Save memory footprint:
Never materialize $N^2$ matrix

Save memory bandwidth:
(high arithmetic intensity)
- Read 3 blocks (from $Q$, $K$, $V$)
- Do two matrix multiplies + a few row summations
- Accumulate into $O$ block (which is resident in cache)

Note there is additional computation vs. the original version (must re-scale prior values of $O$ each step of i-loop)
Fusion in modern DNN frameworks
Old style: library writers hardcoded a few “fused” ops

```c
void 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 \( y = \text{act} \left( \alpha_1 \ast \text{conv}(x) + \alpha_2 \ast z + \text{bias} \right) \).

**Tensorflow:**

- `tensorflow::ops::FusedBatchNorm`: Batch normalization.
- `tensorflow::ops::FusedResizeAndPadConv2D`: Performs a resize and padding as a preprocess during a convolution.
More flexible fusion example: CUDNN “backend”

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

Note: this is Halide “compute at”
Many compiler-based efforts to automatically schedule key DNN operations.
Another trick: use of low precision values

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

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

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

- Forms of approximation: compressing models
  - Lower bit precision
  - Automatic sparsification/pruning (not discussed today)
Why might a GPU be a good platform for DNN evaluation?

consider: arithmetic intensity, SIMD, data-parallelism, memory bandwidth requirements
Deep neural networks on GPUs

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

NVIDIA A100
Why might a GPU be a sub-optimal platform for DNN evaluation?

(Hint: is a general purpose processor needed?)
Special instruction support
Amortize overhead of instruction stream control using more complex instructions

- Fused multiply add \( ax + b \)
- 4-component dot product \( x = A \cdot B \)
- 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
Energy 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%

* 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
Maximizing efficiency: hardware acceleration of DNN inference/training

- Google TPU3
- Intel Deep Learning Inference Accelerator
- Apple Neural Engine
- GraphCore IPU
- SambaNova Cardinal SN10
- Ampere GPU with Tensor Cores
- Cerebras Wafer Scale Engine