Lecture 6: Efficiently Evaluating Deep Networks

Visual Computing Systems
Stanford CS348K, Spring 2023
Today

- We will discuss the workload of evaluating deep neural networks (performing “inference”)

  - This lecture will be heavily biased towards concerns of DNNs that process images (to be honest, because that is what your instructor knows best)
Efficiency challenge

Many DNN topologies
(Many variants on common backbones)

Many Target Devices
Mini-review / crash course:
Convolutional Neural Networks
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.

Responds to horizontal gradients

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1
\end{bmatrix}
\]

Responds to vertical gradients

\[
\begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{bmatrix}
\]
Applying many filters to an image at once

Input RGB image (W x H x 3)

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

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

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

3x3 spatial convolutions on image
\[ 3x3 \times \text{num}_f \text{ilters weights} \]

Output: filter responses
\[ W \times H \times \text{num}_f \text{ilters} \]

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

Filter response maps
(num_filters of them)
Adding additional layers

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

3x3 spatial convolutions
3x3 \( \times \) num\_filters weights

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

After ReLU \(W \times H \times \text{num\_filters}\)

After Pool \(W/2 \times H/2 \times \text{num\_filters}\)

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

Filter responses

ReLU

Pool (max response in 2x2 region)

Note data reduction as a result of “pooling”
Going deeper

Visualization: images that generate strongest response for filters at each layer
More recent image understanding networks

Inception (GoogleLeNet)

ResNet (34 layer version)

Fully Convolutional Network for image segmentation

Stanford CS348K, Spring 2023
Efficiently implementing convolution layers
Direct implementation of conv layer (batched)

```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;
            }
```

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

```c
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

Note: 0-pad matrix

O(N) storage overhead for filter with N elements
Must construct input data matrix
3x3 convolution as matrix-vector product ("explicit gemm")

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

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
x_{00} & x_{01} & x_{02} & x_{03} \\
x_{10} & x_{11} & x_{12} & x_{13} \\
x_{20} & x_{21} & x_{22} & x_{23} \\
x_{30} & x_{31} & x_{32} & x_{33} \\
\vdots & \vdots & \vdots & \vdots \\
x_{80} & x_{81} & x_{82} & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
w_{00} & w_{01} & w_{02} & \ldots \\
w_{10} & w_{11} & w_{12} & \ldots \\
w_{20} & w_{21} & w_{22} & \ldots \\
w_{30} & w_{31} & w_{32} & \ldots \\
\vdots & \vdots & \vdots & \vdots \\
w_{80} & w_{81} & w_{82} & \ldots \\
\end{array}
\]

\[
\text{num filters}
\]
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 a 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)

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

Get It Now →
Dense matrix multiplication

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

// compute C += A * B
#pragma omp parallel for
for (int j=0; j<M; j++)
    for (int i=0; i<N; i++)
        for (int k=0; k<K; k++)
            C[j][i] += A[j][k] * B[k][i];

What is the problem with this implementation?

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

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

// compute C += A * B
#pragma omp parallel for
for (int jblock=0; jblock<M; jblock+=BLOCKSIZE_J)
  for (int iblock=0; iblock<N; iblock+=BLOCKSIZE_I)
    for (int kblock=0; kblock<K; kblock+=BLOCKSIZE_K)
      for (int j=0; j<BLOCKSIZE_J; j++)
        for (int i=0; i<BLOCKSIZE_I; i++)
          for (int k=0; k<BLOCKSIZE_K; k++)
            C[jblock+j][iblock+i] += A[jblock+j][kblock+k] * B[kblock+k][iblock+i];

Idea: compute partial result for block of C while required blocks of A and B remain in cache
(Assumes BLOCKSIZE chosen to allow block of A, B, and C to remain resident)

Self check: do you want as big a BLOCKSIZE as possible? Why?
Hierarchical blocked matrix mult

Exploit multiple levels of memory hierarchy

```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…
Consider SIMD parallelism within a block

\[
\begin{array}{c}
\text{BLOCKSIZE}_J \hspace{1cm} \text{BLOCKSIZE}_I \\
\hline
C \\
\hline
\end{array}
\quad =
\begin{array}{c}
\text{BLOCKSIZE}_J \\
\hline
A \\
\hline
\end{array} \times
\begin{array}{c}
\text{BLOCKSIZE}_K \\
\hline
B \\
\hline
\end{array}
\]

\ldots

for (int j=0; j<\text{BLOCKSIZE}_J; j++) {
    for (int i=0; i<\text{BLOCKSIZE}_I; i+=\text{SIMD_WIDTH}) {
        \text{simd_vec} \ C\_\text{accum} = \text{vec\_load}(&C[j\_\text{block}+j][i\_\text{block}+i]);
        for (int k=0; k<\text{BLOCKSIZE}_K; k++) {
            // \text{C} = \text{A} \ast \text{B} + \text{C}
            \text{simd\_vec} \ A\_\text{val} = \text{splat}(&A[j\_\text{block}+j][k\_\text{block}+k]);
            \text{simd\_muladd} (A\_\text{val}, \text{vec\_load} (&B[k\_\text{block}+k][i\_\text{block}+i]), C\_\text{accum});
        }
        \text{vec\_store} (&C[j\_\text{block}+j][i\_\text{block}+i], C\_\text{accum});
    }
}

\textbf{Vectorize i loop}

\textbf{Good: also improves spatial locality in access to B}

\textbf{Bad: working set increased by }\text{SIMD\_WIDTH}, \text{ still walking over } B \text{ in large steps}
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>
<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

```
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;
    }
}
```
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: $R \times S$
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)

- Fast (in-shared memory) GEMM
- Fast WARP level GEMMs
- Iterators for fast block loading/tensor indexing
- Tensor reductions
- Etc.
Recall: NVIDIA V100 GPU (80 SMs)

Many processing units and many tensor cores.

Need “a lot of parallel work” to fill the machine.
Higher performance with “more work”

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

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

```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[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;
            }

Or you can just directly implement this loop nest directly yourself.
```
Low-level vendor libraries offer high-performance implementations of key DNN layers

NVIDIA cuDNN

Intel® oneAPI Deep Neural Network Library
Example: CUDNN convolution

```c

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

Possible algorithms:

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

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

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

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

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

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

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

- **CUDNN_CONVOLUTION_FWD_ALGO_WINograd_NONFUSED**
  - This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.
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.
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.
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.
High level DNN libraries offer high-performance implementations of key DNN layers

TensorFlow NN ops

<table>
<thead>
<tr>
<th>Operation (TensorFlow op)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tensorflow::ops:AvgPool</td>
<td>Performs average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:AvgPool3D</td>
<td>Performs 3D average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:AvgPool3DDgrad</td>
<td>Computes gradients of average pooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:BiasAdd</td>
<td>Adds bias to value.</td>
</tr>
<tr>
<td>tensorflow::ops:BiasAddGrad</td>
<td>The backward operation for &quot;BiasAdd&quot; on the &quot;bias&quot; tensor.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv2D</td>
<td>Computes a 2-D convolution given 4-D input and 5-D filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv2DBackpropFilter</td>
<td>Computes the gradients of convolution with respect to the filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv2DBackpropInput</td>
<td>Computes the gradients of convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv3D</td>
<td>Computes a 3-D convolution given 5-D input and 6-D filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv3DBackpropFilterV2</td>
<td>Computes the gradients of 3-D convolution with respect to the filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Conv3DBackpropInputV2</td>
<td>Computes the gradients of 3-D convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow::ops:DataFormatDimMap</td>
<td>Returns the dimension index in the destination data format.</td>
</tr>
<tr>
<td>tensorflow::ops:DataFormatVecPermute</td>
<td>Permute input tensor from src_format to dest_format.</td>
</tr>
<tr>
<td>tensorflow::ops:DepthwiseConv2dNative</td>
<td>Computes a 2-D depthwise convolution given 4-D input and 5-D kernel tensors.</td>
</tr>
<tr>
<td>tensorflow::ops:DepthwiseConv2dNativeBackpropFilter</td>
<td>Computes the gradients of depthwise convolution with respect to the filter.</td>
</tr>
<tr>
<td>tensorflow::ops:DepthwiseConv2dNativeBackpropInput</td>
<td>Computes the gradients of depthwise convolution with respect to the input.</td>
</tr>
<tr>
<td>tensorflow::ops:Dilation2D</td>
<td>Computes the grayscale dilation of 4-D input and 3-D filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Dilation2DBackpropFilter</td>
<td>Computes the gradient of morphological 2-D dilation filter.</td>
</tr>
<tr>
<td>tensorflow::ops:Dilation2DBackpropInput</td>
<td>Computes the gradient of morphological 2-D dilation input.</td>
</tr>
<tr>
<td>tensorflow::ops:Elu</td>
<td>Computes exponential linear <code>exp(features) - 1</code> otherwise.</td>
</tr>
<tr>
<td>tensorflow::ops:FractionalAvgPool</td>
<td>Performs fractional average pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:FractionalMaxPool</td>
<td>Performs fractional max pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedBatchNorm</td>
<td>Batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedBatchNormGrad</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedBatchNormGradV2</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedBatchNormGradV3</td>
<td>Gradient for batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedBatchNormV2</td>
<td>Batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedPadConv2D</td>
<td>Performs a padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td>tensorflow::ops:FusedResizeAndPadConv2D</td>
<td>Performs a resize and padding as a preprocess during a convolution.</td>
</tr>
<tr>
<td>tensorflow::ops:InTopK</td>
<td>Says whether the targets are in the top K predictions.</td>
</tr>
<tr>
<td>tensorflow::ops:InTopKV2</td>
<td>Says whether the targets are in the top K predictions.</td>
</tr>
<tr>
<td>tensorflow::ops:L2Loss</td>
<td>L2 Loss.</td>
</tr>
<tr>
<td>tensorflow::ops:LRN</td>
<td>Local Response Normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:LogSoftmax</td>
<td>Computes log softmax activations.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPool</td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPool3D</td>
<td>Performs 3D max pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPool3DDgrad</td>
<td>Computes gradients of 3-D max pooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPool3DDgradGrad</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolGradGrad</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolGradGradV2</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolGradGradWithArgmax</td>
<td>Computes second-order gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolGradV2</td>
<td>Computes gradients of the maxpooling function.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolV2</td>
<td>Performs max pooling on the input.</td>
</tr>
<tr>
<td>tensorflow::ops:MaxPoolWithArgmax</td>
<td>Performs max pooling on the input and outputs both max values and indices.</td>
</tr>
<tr>
<td>tensorflow::ops:NthElement</td>
<td>Finds values of the n-th order statistic for the last dimension.</td>
</tr>
<tr>
<td>tensorflow::ops:QuantizedAvgPool</td>
<td>Produces the average pool of the input tensor for quantized types.</td>
</tr>
<tr>
<td>tensorflow::ops:QuantizedBatchNorm</td>
<td>Quantized Batch normalization.</td>
</tr>
<tr>
<td>tensorflow::ops:QuantizedBiasAdd</td>
<td>Adds Tensor <code>bias</code> to <code>Tensor</code> <code>input</code> for Quantized types.</td>
</tr>
<tr>
<td>tensorflow::ops:QuantizedConv2D</td>
<td>Computes a 2D convolution given quantized 4D input and filter tensors.</td>
</tr>
<tr>
<td>tensorflow::ops:QuantizedMaxPool</td>
<td>Computes the max pool of the input tensor for quantized types.</td>
</tr>
</tbody>
</table>
High level DNN libraries offer high-performance implementations of key DNN layers

TensorFlow NN ops

Implemented via lower level libraries

NVIDIA cuDNN

Intel® oneAPI Deep Neural Network Library
Many efforts to automatically schedule key DNN operations