Compiling Neural Networks

For inference
What is a neural network

• I won’t go into the theory

• From a compiler point of view
  • A set of kernels, typically dominated by convolutions and connected by non-linear components
  • A set of coefficients
    • Training: figure out the best set of coefficients to minimize an objective function that matches output to desired output
    • Inference: invoke the network on an input to generate an output
Targets

• GPU
  • Dominates training
• CPU
• DSP
• Specialized accelerators
  • 148 AI hardware companies
Compilers

• Compiler techniques Ad-Hoc
  • Many use library based approach
  • 148 –x (x is small) hardware companies don’t let you program at a low level

• This presentation
  • Basic functions in many neural networks
  • A set of optimizations that improve the efficiencies of these operations
Convolutions

1D example, 3-pt dot product applied to rolling window
2D Convolutions

Typically “channel” independent tensors
Typically “n” independent filters
Each filter sweeps over the image doing a dot product over the channels
CxHxW tensor convolved with N, CxHxW filters to create Nx(H-2)x(W-2) output
2x2 convolution in code

```c
for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = bias[i];
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y-1][x-1] * coeff[i][z][0][0] +
                                inp[z][y][x-1] * coeff[i][z][1][0] +
                                inp[z][y-1][x] * coeff[i][z][0][1] +
                                inp[z][y][x] * coeff[i][z][1][1];
            }
        }
    }
}
```
Batching

for (int n=0; n<Num_batches; n++) {
    for (int i=0; i<output_channels; i++) {
        for (int y=1; y<h-1; y++) {
            for (int x=1; x<w-1; x++) {
                out[n][i][y][x] = bias[i]
                for (int z=0; z<c; z++) {
                    out[n][i][y][x] += inp[n][z][y-1][x-1] * coeff[i][z][0][0] +
                                        inp[n][z][y][x-1] * coeff[i][z][1][0] +
                                        inp[n][z][y-1][x] * coeff[i][z][0][1] +
                                        inp[n][z][y][x] * coeff[i][z][1][1];
                }
            }
        }
    }
}

• Do ‘n’ independent convolutions
• Purely independent work
• Reuse coefficients
• Great for training
  • Make batches as big as memory allows
• Not so good for inference
  • I don’t want to wait 32 frames before engaging my brakes
StridedConvolution

1D example, 3-pt dot product applied to rolling window
Dilated Convolutions

<table>
<thead>
<tr>
<th>*</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
2x2 grouped convolution

for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = bias[i];
            out[i][y][x] += inp[i][y-1][x-1] * coeff[i][0][0] +
                            inp[i][y][x-1] * coeff[i][1][0] +
                            inp[i][y-1][x] * coeff[i][0][1] +
                            inp[i][y][x] * coeff[i][1][1];
        }
    }
}
Activation Functions

• Need a non-linearity
  • A set of convolutions in series is really just a single convolution
• ReLU
  • $\text{out}[c][x][y] = \text{inp}[c][x][y] > 0 \ ? \ \text{inp}[c][x][y] : 0$
• Tanh
  • $\text{out}[c][x][y] = \tanh(\text{inp}[c][x][y])$
• Sigmoid
  • $\text{out}[c][x][y] = \frac{1}{1+e^{-\text{inp}[c][x][y]}}$
Max Pooling

• 2x2 example

for (int i=0; i<n; i++) {
    for (int y=0; y<h; y+=2) {
        for (int x=0; x<w; x+=2) {
            out[i][y][x] = max(out[i][y][x],
                                out[i][y][x+1],
                                out[i][y+1][x],
                                out[i][y+1][x+1]);
        }
    }
}
Neural Networks

• Convolutional neural networks
  • Feed forward networks, data moves through the network in one direction

• Recurrent Neural Networks
  • Feedback: $h(t) = f(h(t-1), x(t))$

• LSTM (long short term memory)

$$
\begin{align*}
h_f &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\
h_i &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\
h_o &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\
h_c &= \tanh(W_c[h_{t-1}, x_t] + b_c) \\
c &= h_f * c_{t-1} + h_i * h_c \\
h &= h_o * \tanh(c_t)
\end{align*}
$$

t1 = tf.nn.conv2d(L1, coeff.core_network_inc_conv_x_weight, strides=[1, 1, 1, 1], padding='VALID', name='L1')
t2 = tf.math.tanh(t1)
Optimizing: Layer Fusion

- Conv -> ReLU gets transformed to ConvReLU

```java
for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = bias[i];
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y-1][x-1] * coeff[i][z][0][0] +
                                inp[z][y][x-1] * coeff[i][z][1][0] +
                                inp[z][y-1][x] * coeff[i][z][0][1] +
                                inp[z][y][x] * coeff[i][z][1][1];
            }
            out[i][y][x] = out[i][y][x] > 0 ? out[i][y][x] : 0;
        }
    }
}
```

- Does it matter?
- Saving n*h*w memory accesses
- But compute is 4*n*h*w*c
- In our work not a huge deal but it makes a difference
Optimizing Layer Fusion

• Peephole graph optimization
• Anytime there is a node with one successor and the only predecessor of the successor is the node, can fuse as long as fused operator is available
• Can rely on C compiler to fuse compound nodes
  • We hand fused library components
Tensor Scheduling

- Execute the nodes in what order?
  - a->b->c->d->e
- How do you know
  - Ready list
  - Pick a node on the ready list
  - If successor has no unscheduled predecessor add it to ready list
Tensor Allocation

- Can a and b share memory? yes
- Can b and c? no
- Can c and d? yes
- Can d and e? yes
- Can a and inp?
  
  ```cpp
  for (i=0; i<n; i++) {
    for (x=0; x<w; x++) {
      for (z=0; z<c; z++) {
        a[x][i] += inp[x][c] * ...
      }
    }
  }
  ```
Tensor Allocation

• Schedule graph first
• Build interference graph
  • Edge from input to output variables in a node are node dependent
  • Across nodes, dataflow problem to build edges
• Allocate graph
  • Is it the same as standard register allocation?
    • We don’t have a fixed number of registers
    • Different tensors are different sizes
    • Temporal locality matters
  • We use a greedy algorithm, combine greedily in order
Node Interchange

- Convolve did twice the necessary multiplies
  - Change the source: could but not so easy
  - Pattern match this case
  - Can a general compiler algorithm figure this out, I don’t know
  - Would a general compiler do this, probably not
Optimization on 2x2 convolution

for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = 0;
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y-1][x-1] * coeff[i][z][0][0] +
                                inp[z][y][x-1] * coeff[i][z][1][0] +
                                inp[z][y-1][x] * coeff[i][z][0][1] +
                                inp[z][y][x] * coeff[i][z][1][1];
            }
        }
    }
}

- Loop optimizations
  - Which loop should be inner
  - Unroll an outer loop
    - Inp is independent of 'i'
    - Coeff is independent of 'x' and 'y'
  - Cache tiling/blocking
  - Vectorizing
Which loop to vectorize

```c
for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = 0;
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y-1][x-1] * coeff[i][z][0][0] +
                                inp[z][y][x-1] * coeff[i][z][1][0] +
                                inp[z][y-1][x] * coeff[i][z][0][1] +
                                inp[z][y][x] * coeff[i][z][1][1];
            }
        }
    }
}
```

- Vectorize x, keep z inner
  - One vector load of inp, one scalar load
- Vectorize i
  - But out and coeff are not stride-1 in ‘i’
  - So change their data layout
    - Coeff is easy, it’s just a constant array
    - Out is more complicated except
      - N*h*w*C convolutions, N*h*w elements, if C is large can transpose cheaply
      - Analyze globally to find best ordering for a tensor
- Choices are not obvious because sizes vary
- Default data layout
  - Tensorflow is NHWC
  - Nvidia and Pytorch are NCHW
Convolution -> Matrix Multiply

```c
for (int i=0; i<n; i++) {
    for (int x=0; x<w; x++) {
        for (int z=0; z<c; z++) {
            out[i][x] += coeff[i][z] * inp[z][x];
        }
    }
}
```

1-pt convolution is matrix multiplication
im2col

• Coeff is

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Input Data

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

A B C D
Data copying (dilated convolutions)

for (int i=0; i<n; i++) {
    for (int y=0; y<h; y++) {
        for (int x=0; x<w; x++) {
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y][4*x] *…
            }
        }
    }
}

for (int i=0; i<n; i++) {
    for (int y=0; y<h; y++) {
        for (int x=0; x<w; x++) {
            for (int z=0; z<c; z++) {
                tmp[z][y][x] = inp[z][y][4*x];
            }
        }
    }
}

for (int i=0; i<n; i++) {
    for (int y=0; y<h; y++) {
        for (int x=0; x<w; x++) {
            for (int z=0; z<c; z++) {
                out[i][y][x] += tmp[z][y][x]*…
            }
        }
    }
}
DMA

for (int i=0; i<n; i++) {
    for (int y=0; y<h; y++) {
        for (int x=0; x<w; x++) {
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y][x] *…
            }
        }
    }
}

for (int i=0; i<n; i+=bi) {
    for (int y=0; y<h; y+=by) {
        for (int x=0; x<w; x+=bx) {
            for (int z=0; z<c; z+=bz) {
                dma(t_inp2, inp, z+bz,y+by,x+bx);
                wait for dma into t_inp;
                for (int i2=i; i2<i2+bi; i2++) {
                    for (int y2=y; y2<y2+by; y++) {
                        for (int x2=x; x2<x2+bx; x2++) {
                            for (int z2=z; z2<z2+bz; z2++) {
                                t_out2[i2-i][y2-y][x2-x] +=
                                t_inp[z2-z][y2-y][x2-x] *…
                            }
                        }
                    }
                }
                dma(out, t_out2, z,y,x);
                wait for dma of t_out to finish
                swap t_inp and t_inp2, swap t_out and t_out2
Cooperative Hierarchical Memory

for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024): for i in range(4):
      AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
    for each i in 0..4:
      BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
    memory_barrier_among_threads()
    for yi in range(8):
      for xi in range(8):
        CL[yi][xi] += AS[yi] * BS[xi]
    for yi in range(8):
      for xi in range(8):
        C[yo*8+yi][xo*8+xi] = CL[yi][xi]

- Parallelize matrix multiply using 2x2 processors
  - Processors 1 to 4 partition result C matrix
  - Processors 1 and 2 use top two blocks of A
    - Let them jointly bring it into memory they share

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning
13th USENIX Symposium on Operating Systems Design and Implementation
Putting Things Together: TVM

• Neural Network compiler from University of Washington
• Industry support from ARM, Qualcomm, Amazon
• Support compiling for large variety of machines
  • CPU
  • GPU
  • Accelerators
• Key components
  • Tensor Expression Language
  • ML based search across optimization space
Tensor Expression Language

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
    t.sum(w[i, k] * x[j, k], axis=k))
```

```
def gemm_intrin_lower(inputs, outputs):
    ww_ptr = inputs[0].access_ptr("r")
    xx_ptr = inputs[1].access_ptr("r")
    zz_ptr = outputs[0].access_ptr("w")
    compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
    reset = t.hardware_intrin("fill_zero", zz_ptr)
    update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
    return compute, reset, update
```
ML Based Search in TVM

• Billions of configurations
  • Schedule Explorer
  • Machine learning cost evaluation

• Machine learning cost evaluation
  • Generate and run potential kernels
  • Datapoints used to train machine learning algorithm
  • Machine learning algorithm predicts new datapoints

• Schedule Explorer
  • Simulated annealing used to select adjacent configurations to explore
Quantization

• Common scenario
  • Training in floating point
  • Inference will be more efficient as integer

• Convolutions
  • Coefficients and data
  • Coefficients are known literals
    • Multiply each coefficient by a fixed quantization factor
      • Quantization factor is $2^{n\text{-head\_room}} / \text{max(coefficients)}$
      • Coefficients are 0.23, 1.45, -1.67
        • Quantization factor is 128/1.67
      • One quantization factor per what
        • Coefficient, that’s floating point
        • Layer, some filters might lose all precision
        • Per layer per filter
Quantizing Coefficients

for (int i=0; i<n; i++) {
    for (int y=1; y<h-1; y++) {
        for (int x=1; x<w-1; x++) {
            out[i][y][x] = bias[i];
            for (int z=0; z<c; z++) {
                out[i][y][x] += inp[z][y-1][x-1] * coeff[i][z][0][0] +
                                inp[z][y][x-1] * coeff[i][z][1][0] +
                                inp[z][y-1][x] * coeff[i][z][0][1] +
                                inp[z][y][x] * coeff[i][z][1][1];
            }
            out[i][y][x] = out[i][y][x] >>
                            lg(quantization_factor[i]);
        }
    }
}
Quantizing Data

• Don’t know the actual range of data, it’s not literal constants
• Instrument your code in floating point
• Run on representative data
• Find the maximum data per layer or per layer per output filter
  • Or find a sigma away from maximum data (somewhat less or maybe somewhat more)
  • Adjust quantization factor so that expected data stays in range
    • Maybe headroom, maybe allow slight overflow
    • Use saturating arithmetic so that overflow saturates rather than rolls over
• I’ve described symmetric quantization, int(x) = float(x) * a constant
  • Non-symmetric, int(x) = float(x) * c0 + c1
    • Makes convolutions more expensive to calculate
Streaming Audio

- Most important optimization for streaming in CNNs
- Not relevant for most networks as most networks aren’t invoked with streaming
  - Standard compilers don’t talk about this

That makes sense, Ok Webex, please note the issue and let’s continue

- When network hears “ebex, pleas”, can it recognize “Ok Webex”
  - RNN maybe
  - CNN no, it has no memory of “e, Ok We”
Streaming Audio on CNNs

- Feed overlapping data into network in succession
  - That makes sense
  - Makes sense, ok w
  - Sense, Ok Webex
  - Ok Webex, please
- By default will do a lot of extra work

That makes sense, Ok Webex, please note the issue and let’s continue
Streaming Audio on CNNs

for (int x=0; x<n; x++)
    out[x] = in[x] * coeff[0] + in[x+1] * coeff[1];

Next time

for (int x=n/4; x<n+n/4; x++)
    out[x] = in[x] * coeff[0] + in[x+1] * coeff[1];

First convolution in second instance is repeated from $n/4$’th in first
Streaming Audio in CNNs

```c
for (int x=0; x<n; x++)
    out[x] = in[x] * coeff[0] + in[x+1] * coeff[1];

Replace with
```Buffer Data Previous Invocation``` New data```

```c
memcpy(buffer, buffer+n/4, 3*n/4 elements);
for (int x=0; x<n/4; x++)
    out[x] = in[x+3*n/4]   * coeff[0] +
             in[x+3*n/4+1] * coeff[1];
memcpy(buffer+3*n/4, out, n/4 elements)
```
Streaming Audio CNNs

- If input to convolve is shifted 32 points, what happens to output?
  - Shifted 32 points

- If input to maxpool2 is shifted by 32 points, what happens to output?
  - Shifted 16 points
Computing Overlap

- Walk the tree in scheduled order
- If predecessors in a node all have the same overlap factor
  - Compute output overlap based on input overlap and type of node
- If predecessors have inconsistent overlap or bad overlap
  - Set overlap of node to bad

\[
a = \text{convolve}(\text{inp})
\]
\[
b = \text{tanh}(a)
\]
\[
c = \text{MaxPool2}(b)
\]
Add
bad
Streaming Audio CNNs

- Do I need to store 3*n/4 elements of output for both convolve and tanh?
  - After buffering, tanh is only reading the same elements that were computed by convolve
  - In general more complicated, a three point convolution reads two more input points than it computes
- Algorithm
  - Walk the tree in reverse scheduled order
  - Buffer for a node the maximum number of elements needed by its successors

```c
for (int i=0; i<n-2; i++)
    out[i] = c0 * inp[i] + c1 * inp[i+1] + c2 * inp[i+2];
```