Lecture: Variable Precision

- Topics: approaches to support variable precision (Stripes, Bit-Pragmatic, Laconic, BitFusion), compression during training, pruning

The next slide talks about a few general strategies to improve DNN efficiency (clearly not comprehensive, e.g., doesn’t include analog approaches). Today’s discussion focuses on efficiency opportunities that exploit quantization and variable precision, in both inference and training. We’ll start with a series of papers from U. Toronto and one from UCSD that target inference. For much of this discussion, the variable precision is in the activations (the weights may be held at say 16b, or may be variable).
Wasted Work

Four types of unnecessary work in DNN execution:

• Data re-fetch: when the same value is fetched repeatedly from storage structures; can be addressed with reuse/tiling

• High data width: when the bits exceed the information content; addressed with quantization/approximation

• Sparsity: computations involving zeros; already seen pruning, Cnvlutin, SCNN

• Temporal/spatial redundancy: value similarity in time/space; can be addressed with early diffs
The first paper is focusing on “static” exploitation of low precision. This means that the DNN has been pre-analyzed and you’ve already figured out that you can achieve 100% of the baseline accuracy or 99% of the baseline accuracy by using fewer bits per layer (see Table above). So every time a layer starts execution, it knows it only needs (say) 8 bits of precision. If it can somehow configure the hardware to do 2 8b computations on a 16b unit, you can get 2x speedup. The approach in this work is to break up any N-bit computation into N cycles and do 1-bit of input at a time. If you have a 16b unit, you can do 16 neurons in parallel. So the per-cycle work done is the same as the baseline. But if you are dealing with 8b inputs, your work finishes 2x sooner.
Stripes

- Philosophy: spread the computation over several cycles; perform one bit at a time; compensate with parallel multiplications of other activations that contribute to the same neuron; computation ends early for low-res values
- Increases area by 32% (more weights, more shift registers, more accumulators)
- Reduces energy by 57%, improves performance by 1.92x on average (fewer bits moving through the memory hierarchy and through compute)

The major drawback is that the compute unit needs a lot more area. It needs large shifters and accumulators for all the 16 neurons being processed in parallel. But the energy and performance benefit can be substantial, especially since small operands will also consume less energy in the memory hierarchy.

Worth noting that an alternative approach is to just have flexible ALUs where a 16b unit has logic to manage carries and re-configure the ALU into say 4 4b units. That alternative is never discussed in the Toronto papers.
This paper is looking for dynamic opportunities. Even if the operand is only 9b, more than half those bits are likely 0 and don’t induce a shift+add. So the input operand is first transformed into a new format that has the positions of all the 1 bits. That then causes the appropriate shift+add. While there’s more complexity and more bits being moved, this takes fewer cycles.

- Starts with a Stripes-like design, then exploits the fact that 0’s in the activation are not doing anything effectual; by skipping those cycles, the multiplication finishes even sooner.
- Requires a pre-processing step where the input value is encoded to only keep the 1 bits (results in operand expansion → more energy, area, complexity).
- The position indicates the shift amount before adding.
Effectuals in Activations

One key difference here is that if the ALUs are getting faster, the buffers need to feed data at a higher rate (unlike the static Stripes technique where we were also storing less data in buffers).

<table>
<thead>
<tr>
<th></th>
<th>Alexnet</th>
<th>NiN</th>
<th>Google</th>
<th>VGGM</th>
<th>VGGS</th>
<th>VGG19</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>16-bit Fixed-Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>7.8%</td>
<td>10.4%</td>
<td>6.4%</td>
<td>5.1%</td>
<td>5.7%</td>
<td>12.7%</td>
</tr>
<tr>
<td>NZ</td>
<td>18.1%</td>
<td>22.1%</td>
<td>19.0%</td>
<td>16.5%</td>
<td>16.7%</td>
<td>24.2%</td>
</tr>
<tr>
<td><strong>8-bit Quantized</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>31.4%</td>
<td>27.1%</td>
<td>26.8%</td>
<td>38.4%</td>
<td>34.3%</td>
<td>16.5%</td>
</tr>
<tr>
<td>NZ</td>
<td>44.3%</td>
<td>37.4%</td>
<td>42.6%</td>
<td>47.4%</td>
<td>46.0%</td>
<td>29.1%</td>
</tr>
</tbody>
</table>

**TABLE I**

AVERAGE FRACTION OF NON-ZERO BITS PER NEURON FOR TWO FIXED-LENGTH REPRESENTATIONS: 16-BIT FIXED-POINT, AND 8-BIT QUANTIZED. **ALL:** OVER ALL NEURONS. **NZ:** OVER NON-ZERO NEURONS ONLY.

They employ a few techniques that lower performance, but also reduce area (the size of the shifters/accum). Note that the buffers have to supply data at a higher rate.
Laconic

An 8b multiplication is like adding 64 single bits, most are 0
Not very area-effective for 8b baseline, but effective for 16b
BitFusion

- Not trying to exploit dynamic opportunities (like Bit-Pragmatic and Laconic), but exploiting narrow ops (like Stripes)

- Designs hardware that is more compact than bit-serial architectures like Stripes

- Organizes an 8bx8b multiplier (a Fusion Unit) as 16 2bx2b multipliers (Bit Bricks); smaller precision computations are then mapped to a set of Bit Bricks

This is another attempt to statically re-configure a MAC with variable precision. Uses a tree of aggregators and control within the tree nodes can be used for reconfiguration. A tree-like org is perhaps more in line with typical generate-propagate circuits.
This figure shows an example of a 4bx4b multiplication. You can see that the result is as if we put together the results of 4 2bx2b multiplications (with appropriate shifts).

(a) A 4-bit multiplication
\((6_{10} \times 11_{10} = 66_{10})\)

(b) Decomposing the 4-bit multiplication to four 2-bit multiplications.
This slide shows how the 2bx2b multipliers (BitBricks) are organized as an H-Tree. The figure also shows how the BitBricks are clubbed to create 4 8bx2b multipliers.
Comparison to Stripes

A key difference is that temporal spreading needs wide shifters and accumulators for all the computations happening in parallel. But spatial spreading is more efficient because different parts of the H-tree operate with lower bit-width.

Key difference is that Stripes needs large shifters and accumulators for each “bit brick”, while BitFusion does not.

Fig. 8: Temporal design. Operands $a$–$h$ are 2-bit.

Fig. 9: Spatial fusion. Operands $a$–$h$ are 2-bit.
The figure shows the nature of various DNN layers. It observes that if we’re storing the outputs of a pooling layer, we don’t really need to save the outputs of the preceding ReLU layer (just storing binary values is a lossless approach). With these 3 approaches, we can shrink the feature maps enough to fit on GPU memory and not need a vDNN-like offload.

- Competing approach to vDNN
- While activations are consumed at full precision in forward pass, they are stored in compressed format for the backward pass; compression can be lossy or lossless
- ReLu followed by Conv: Sparse encoding (lossless)
- ReLu followed by pooling: Binarize (lossless)
- Others: Low (8b) precision storage (lossy)

**Figure 3:** Breakdown of memory within stashed feature maps. ReLU layer consumes a major portion of the footprint.
Compressed vDNN

- Observes high sparsity in activations
- Data is compressed before the offload
- Average compression ratio of 2.6x and improves vDNN performance by 32%

There is a version of vDNN that also performs activation compression before doing the GPU ↔ CPU memory transfers. It seems to operate on a baseline where vDNN is more penal than the original vDNN paper.
Pruning

• Outlier aware pruning: 97% of weights can use 4-bit precision, while 3% of weights need full precision

• Dark side of pruning: prior work only focuses on output, but many DNNs also produce a confidence that is fed to other downstream algorithms (Viterbi); pruning can mess up the confidence even if it doesn’t affect the top-1 or top-5 classification

While pruning may produce the same winning neuron as the baseline, the raw neuron values (representing the confidence of the prediction) are also sometimes passed to downstream algorithms. Pruning may cause more variation in those algorithms.
References

• “Stripes: Bit Serial Deep Neural Network Computing”, P. Judd et al., MICRO 2016
• “Bit Pragmatic Deep Neural Network Computing”, J. Albericio et al., MICRO 2017
• “Laconic Deep Learning Inference Acceleration”, S. Sharify et al., ISCA 2019
• “Bit Fusion: Bit-level Dynamically Composable Architecture for Accelerating Deep Neural Networks”, H. Sharma et al., ISCA 2018
• “GIST: Efficient Data Encoding for Deep Neural Network Training”, A. Jain et al., ISCA 2018
• “Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks”, M. Rhu et al., HPCA 2018
• “The Dark Side of DNN Pruning”, R. Yazdani et al., ISCA 2018
• “Energy Efficient Neural Network Accelerator based on Outlier-Aware Low-Precision Computation”, E. Park et al., ISCA 2018