Lecture 12:

Hardware Acceleration of DNNs

Visual Computing Systems
Stanford CS348V, Winter 2018
Hardware acceleration for DNNs

Google TPU:

Huawei Kirin NPU

Apple Neural Engine

Intel Lake Crest
Deep Learning Accelerator

MIT Eyeriss

Volta GPU with Tensor Cores

Slide credit: Xuan Yang
And many more…

<table>
<thead>
<tr>
<th>Category</th>
<th>Companies</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC Giants</td>
<td>Intel, Qualcomm, Nvidia, Samsung, AMD, Apple, Xilinx, IBM, STMicroelectronics, NXP, MediaTek, HiSilicon</td>
<td>12</td>
</tr>
<tr>
<td>Cloud/HPC</td>
<td>Google, Amazon_AWS, Microsoft, Aliyun, Tencent Cloud, Baidu, Baidu Cloud, HUAWEI Cloud, Fujitsu</td>
<td>9</td>
</tr>
<tr>
<td>IP Vendors</td>
<td>ARM, Synopsys, Imagination, CEVA, Cadence, VeriSilicon</td>
<td>6</td>
</tr>
<tr>
<td>Startups in China</td>
<td>Cambricon, Horizon Robotics, DeePhi, Bitmain, Chipintelli, Thinkforce</td>
<td>6</td>
</tr>
</tbody>
</table>

Image credit: Shan Tang [https://medium.com/@shan.tang.g/a-list-of-chip-ip-for-deep-learning-48d05f1759ae]
Modern NVIDIA GPU
(Volta)
Recall properties of GPUs

- “Compute rich”: packed densely with processing elements
  - Good for compute-bound applications

- Good, because dense-matrix multiplication and DNN convolutional layers (when implemented properly) is compute bound

- But also remember cost of instruction stream processing and control in a programmable processor:

Note: these figures are estimates for a CPU:

![Pie chart showing instruction supply at 42%, data supply at 28%, clock and control at 24%, and arithmetic at 6%](efficient_embedded_computing.png)

*Efficient Embedded Computing [Daily et al. 08]*
*Figure credit Eric Chung*
Volta GPU

Each SM core has:
- 64 fp32 ALUs (mul-add)
- 32 fp64 ALUs
- 8 “tensor cores”

Execute 4x4 matrix mul-add instr
A x B + C for 4x4 matrices A,B,C
A, B stored as fp16, accumulation with fp32 C

There are 80 SM cores in the GV100 GPU:
- 5,120 fp32 mul-add ALUs
- 640 tensor cores
- 6 MB of L2 cache
- 1.5 GHz max clock

= 15.7 TFLOPs fp32
= 125 TFLOPs (fp16/32 mixed) in tensor cores
Google TPU
(version 1)
Discussion: workloads

- What did TPU paper state about characteristics of modern DNN workloads at Google?
Google’s TPU

Figure credit: Jouppi et al. 2017
TPU area proportionality

Compute ~ 30% of chip
Note low area footprint of control

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

Figure credit: Jouppi et al. 2017
Systolic array
(matrix vector multiplication example: \( y = Wx \))
Systolic array

(matrix vector multiplication example: \( y = Wx \))
Systolic array
(matrix vector multiplication example: $y = Wx$)
Systolic array
(matrix vector multiplication example: $y = Wx$)

Weights FIFO

Accumulators (32-bit)
Systolic array

(matrix vector multiplication example: \( y = Wx \))
Systolic array
(matrix vector multiplication example: $y = Wx$)
Systolic array
(matrix matrix multiplication example: $Y = WX$)

Notice: need multiple 4x32bit accumulators to hold output columns
Building larger matrix-matrix multiplies

Example: \( A = 8 \times 8, B = 8 \times 4096, C = 8 \times 4096 \)

Assume 4096 accumulators
Building larger matrix-matrix multiplies

Example: \( A = 8 \times 8, B = 8 \times 4096, C = 8 \times 4096 \)

Assume 4096 accumulators
Building larger matrix-matrix multiplies

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

Assume 4096 accumulators
Building larger matrix-matrix multiplies

Example: \( A = 8 \times 8, B = 8 \times 4096, C = 8 \times 4096 \)

Assume 4096 accumulators
TPU Perf/Watt

GM = geometric mean over all apps
WM = weighted mean over all apps

Figure credit: Jouppi et al. 2017
Alternative scheduling strategies

(a) Weight Stationary

(b) Output Stationary

(c) No Local Reuse

TPU was weight stationary (weights kept in register at PE)

Figure credit: Sze et al. 2017
EIE: targeting sparsified networks
Sparse, weight-sharing fully-connected layer

\[ b_i = \text{ReLU} \left( \sum_{j=0}^{n-1} W_{ij} a_j \right) \]

Fully-connected layer:
Matrix-vector multiplication of activation vector \( a \) against weight matrix \( W \)

\[ b_i = \text{ReLU} \left( \sum_{j \in X_i \cap Y} S[I_{ij}] a_j \right) \]

Sparse, weight-sharing representation:
\( I_{ij} = \text{index for weight } W_{ij} \)
\( S[] = \text{table of shared weight values} \)
\( X_i = \text{list of non-zero indices in row } i \)
\( Y = \text{list of non-zero indices in } a \)

Note: activations are sparse due to ReLU
Efficient inference engine (EIE) ASIC

Custom hardware for decode and evaluate sparse, compressed DNNs

Hardware represents weight matrix in compressed sparse column (CSC) format to exploit sparsity in activations:

```plaintext
for each nonzero a_j in a:
    for each nonzero M_ij in column M_j:
        b_i += M_ij * a_j
```

More detailed version:

```plaintext
for j=0 to length(a):
    if (a[j] == 0) continue;   // scan to nonzero
    col_values = M_j_values[M_j_start[j]];
    col_indices = M_j_indices[M_j_start[j]];
    col_nonzeros = M_j_start[j+1]-M_j_start[j];
    for i=0, i_count=0 to col_nonzeros:
        i += col_indices[i_count]
        b[i] += lookup[M_j_values[i]] * a_values[j_count]
```

* Keep in mind there’s a unique lookup table for each chunk of matrix values
Parallelization of sparse-matrix-vector product

Stride rows of matrix across processing elements
Output activations strided across processing elements

\[ \vec{a} \begin{bmatrix} 0 & 0 & a_2 & 0 & a_4 & a_5 & 0 & a_7 \end{bmatrix} \times \begin{bmatrix} \vec{b} \end{bmatrix} = \begin{bmatrix} \vec{b} \end{bmatrix} \]

Weights stored local to PEs. Must broadcast non-zero \( a_j \)'s to all PEs
Accumulation of each output \( b_i \) is local to PE
EIE unit for quantized sparse/matrix vector product

 Tuple representing non-zero activation \((a_j, j)\) arrives and is enqueued
The compressed DNN model is produced as described from Caffe model zoo [28] and NeuralTalk model zoo [7]; of models: uncompressed DNN model and the compressed Tegra K1.

Components [26], [27] to report the AP+DRAM power for regulator efficiency and meter, then assumed doesn’t have software interface to report power consumption, PARSE CSRMV for the compressed sparse model. Tegra K1 cuBLAS GEMV for the original dense model and cuS-192 CUDA cores as our mobile GPU baseline. We used matrix-vector multiplication on GPUs.

cuSPARSE CSRMV kernel, which is optimized for sparse we stored the sparse matrix in in CSR format, and used the benchmark, we used cuBLAS GEMV to implement a state-of-the-art GPU for deep learning as our baseline CPU socket and DRAM power original dense model and MKL SPBLAS CSRMV for the on CPU, we used MKL CBLAS GEMV to implement.

Sources of energy savings:
- Compression allows all weights to be stored in SRAM (few DRAM loads)
- Low-precision 16-bit fixed-point math (5x more efficient than 32-bit fixed math)
- Skip math on inputs activations that are zero (65% less math)

Warning: these are not end-to-end: just fully connected layers!
Thoughts

- EIE paper highlights performance on fully connected layers (see graph above)
  - Final layers of networks like AlexNet, VGG ...
  - Common in recurrent network topologies like LSTMs

- But many state-of-the-art image processing networks have moved to fully convolutional solutions
  - Recall Inception, SqueezeNet, etc..
Summary of hardware techniques

- Specialized datapaths for dense linear algebra computations
  - Reduce overhead of control (compared to CPUs/GPUs)

- Reduced precision (computation and storage)

- Exploit sparsity

- Accelerate decompression