Memory Bandwidth and Low Precision Computation

CS6787 Lecture 9 — Fall 2017
Memory as a Bottleneck

• So far, we’ve just been talking about compute
  • e.g. techniques to decrease the amount of compute by decreasing iterations

• But machine learning systems need to process huge amounts of data

• Need to store, update, and transmit this data

• As a result: memory is of critical importance
  • Many applications are memory-bound
Memory: The Simplified Picture

Compute \[ \rightarrow \] RAM
Memory: The Multicore Picture
Memory: The Multisocket Picture
Memory: The Distributed Picture
What can we learn from these pictures?

• Many more memory boxes than compute boxes
  • And even more as we zoom out

• Memory has a hierarchical structure

• Locality matters
  • Some memory is closer and easier to access than others
  • Also have standard concerns for CPU cache locality
What limits us?

• **Memory capacity**
  • How much data can we store locally in RAM and/or in cache?

• **Memory bandwidth**
  • How much data can we load from some source in a fixed amount of time?

• **Memory locality**
  • Roughly, how often is the data that we need stored nearby?

• **Power**
  • How much energy is required to operate all of this memory?
One way to help: Low-Precision Computation
Low-Precision Computation

• Traditional ML systems use 32-bit or 64-bit floating point numbers

• But do we actually need this much precision?
  • Especially when we have inputs that come from noisy measurements

• Idea: instead use 8-bit or 16-bit numbers to compute
  • Can be either floating point or fixed point
  • On an FPGA or ASIC can use arbitrary bit-widths
Low Precision and Memory

- Major benefit of low-precision: uses less memory bandwidth

Precision in DRAM

- 64-bit float vector
  - F64
  - F64
  - F64
- 32-bit float vector
  - F32
  - F32
  - F32
  - F32
- 16-bit int vector
  - ...
- 8-bit int vector
  - ...

Memory Throughput

- 5 numbers/ns
- 10 numbers/ns
- 20 numbers/ns
- 40 numbers/ns

(assuming ~40 GB/sec memory bandwidth)
Low Precision and Memory

- Major benefit of low-precision: takes up less space

### Precision in DRAM

- 64-bit float vector
  - F64 F64 F64 ...
- 32-bit float vector
  - F32 F32 F32 F32 F32 F32 F32 ...
- 16-bit int vector
  - ...
- 8-bit int vector
  - ...

### Cache Capacity

- 4 M numbers
- 8 M numbers
- 16 M numbers
- 32 M numbers

(assuming ~32 MB cache)
Low Precision and Parallelism

• Another benefit of low-precision: use **SIMD instructions** to get more parallelism on CPU

<table>
<thead>
<tr>
<th>SIMD Precision</th>
<th>SIMD Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit float vector</td>
<td>4 multiplies/cycle (vmulpd instruction)</td>
</tr>
<tr>
<td>32-bit float vector</td>
<td>8 multiplies/cycle (vmlps instruction)</td>
</tr>
<tr>
<td>16-bit int vector</td>
<td>16 multiplies/cycle (vpmaddwd instruction)</td>
</tr>
<tr>
<td>8-bit int vector</td>
<td>32 multiplies/cycle (vpmaddubsw instruction)</td>
</tr>
</tbody>
</table>
Low Precision and Power

- Low-precision computation can even have a super-linear effect on energy

- Memory energy can also have quadratic dependence on precision
Effects of Low-Precision Computation

• **Pros**
  • Fit more numbers (and therefore more training examples) in memory
  • Store more numbers (and therefore larger models) in the cache
  • Transmit more numbers per second
  • Compute faster by extracting more parallelism
  • Use less energy

• **Cons**
  • Limits the numbers we can represent
  • Introduces **quantization error** when we store a full-precision number in a low-precision representation
Ways to represent low-precision numbers
FP16/Half-precision floating point

• 16-bit floating point numbers

\[ x = (-1)^{\text{sign bit}} \cdot 2^{\text{exponent} - 15} \cdot 1.\text{significand}_2 \]
Arithmetic on half-precision floats

• **Complicated**
  • Has to handle adding numbers with different exponents and signs
  • To be efficient, needs to be **supported in hardware**

• **Inexact**
  • Operations can experience overflow/underflow just like with more common floating point numbers, but it happens more often

• Can represent a **wide range of numbers**
  • Because of the exponential scaling
Half-precision floating point support

• Supported on some **modern GPUs**
  • Including new efficient implementation on NVIDIA Pascal GPUs

<table>
<thead>
<tr>
<th>GPU</th>
<th>DFMA (FP64 TFLOP/s)</th>
<th>FFMA (FP32 TFLOP/s)</th>
<th>HFMA2 (FP16 TFLOP/s)</th>
<th>DP4A (INT8 TIOP/s)</th>
<th>DP2A (INT16/8 TIOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP100 (Tesla P100 NVLink)</td>
<td>5.3</td>
<td>10.6</td>
<td>21.2</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>GP102 (Tesla P40)</td>
<td>0.37</td>
<td>11.8</td>
<td>0.19</td>
<td>43.9</td>
<td>23.5</td>
</tr>
<tr>
<td>GP104 (Tesla P4)</td>
<td>0.17</td>
<td>8.9</td>
<td>0.09</td>
<td>21.8</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Table 1: Pascal-based Tesla GPU peak arithmetic throughput for half-, single-, and double-precision fused multiply-add instructions, and for 8- and 16-bit vector dot product instructions. (Boost clock rates are used in calculating peak throughputs. TFLOP/s: Tera Floating-point Operations per Second. TIOP/s: Tera Integer Operations per Second. https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/}

• Good empirical results for **deep learning**
Fixed point numbers

- \( p + q + 1 \)-bit fixed point number

1-bit sign  \( p \)-bit integer part  \( q \)-bit fractional part

- The represented number is

\[
x = (-1)^{\text{sign bit}} \left( \text{integer part} + 2^{-q} \cdot \text{fractional part} \right)
\]

\[
= 2^{-q} \cdot \text{whole thing as signed integer}
\]
Example: 8-bit fixed point number

• It’s common to want to represent numbers between -1 and 1
  • To do this, we can use a fixed point number with all fractional bits

  \[ x = 2^{-7} \cdot k \in \left\{ -1, -\frac{127}{128}, \ldots, -\frac{1}{128}, 0, \frac{1}{128}, \ldots, \frac{126}{128}, \frac{127}{128} \right\} \]
More generally: scaled fixed point numbers

- Sometimes we don’t want the decimal point to lie between two bits that we are actually storing
  - We might want more tight control over what our bits mean

- Idea: pick a real-number scale factor $s$, then let integer $k$ represent
  \[ x = s \cdot k \]

- This is a generalization of traditional fixed point, where
  \[ s = 2^{-\# \text{ of fractional bits}} \]
Arithmetic on fixed point numbers

• **Simple**
  • Can just use preexisting integer processing units

• **Mostly exact**
  • Underflow impossible
  • Overflow can happen, but is easy to understand
  • Can always convert to a higher-precision representation to avoid overflow

• Can represent a much narrower range of numbers than float
Example: Exact Fixed Point Multiply

• When we multiply two integers, if we want the result to be exact, we need to convert to a representation with more bits

• For example, if we take the product of two 8-bit numbers, the result should be a 16-bit number to be exact.
  • Why? $100 \times 100 = 10000$ which can’t be stored as an 8-bit number

• To have exact fixed point multiply, we can do the same thing
  • Since fixed-point operations are just integer operations behind the scenes
Support for fixed-point arithmetic

- **Anywhere integer arithmetic is supported**
  - CPUs, GPUs
  - Although not all GPUs support 8-bit integer arithmetic
  - And AVX2 does not have all the 8-bit arithmetic instructions we’d like

- Particularly effective on **FPGAs and ASICs**
  - Where floating point units are costly

- Sadly, very little support for other precisions
  - 4-bit operations would be particularly useful
Custom Quantization Points

• Even more generally, we can just have a list of $2^b$ numbers and say that these are the numbers a particular low-precision string represents
  • We can think of the bit string as indexing a number in a dictionary

• Gives us total freedom as to range and scaling
  • But computation can be tricky

• Some recent research into using this with hardware support
  • “The ZipML Framework for Training Models with End-to-End Low Precision: The Cans, the Cannots, and a Little Bit of Deep Learning” (Zhang et al 2017)
Recap of low-precision representations

- **Half-precision floating-point**
  - Complicated arithmetic, but good with hardware support
  - Difficult to reason about overflow and underflow
  - Better range
  - No 8-bit support as of yet

- **Fixed-point**
  - Simple arithmetic, supported wherever integers are
  - Easy to reason about overflow, but has worse range
  - Supports 8-bit and 16-bit arithmetic, but little to no 4-bit support
Low-Precision SGD
Recall: SGD update rule

\[ \omega_{t+1} = \omega_t - \alpha_t \nabla f(\omega_t; x_t, y_t) \]

• There are a lot of numbers we can make low-precision here
  • We can quantize the input dataset \( x, y \)
  • We can quantize the model \( w \)
  • We can try to quantize within the gradient computation itself
  • We can try to quantize the communication among the parallel workers
Four Broad Classes of Numbers

- **Dataset numbers**
  - used to store the immutable input data

- **Model numbers**
  - used to represent the vector we are updating

- **Gradient numbers**
  - used as intermediates in gradient computations

- **Communication numbers**
  - used to communicate among parallel workers
Quantize classes independently

- Using low-precision for different number classes has different effects on throughput.
  - Quantizing the **dataset numbers** improves memory capacity and overall training example throughput
  - Quantizing the **model numbers** improves cache capacity and saves on compute
  - Quantizing the **gradient numbers** saves compute
  - Quantizing the **communication numbers** saves on expensive inter-worker memory bandwidth
Quantize classes independently

• Using low-precision for different number classes has different effects on statistical efficiency and accuracy.
  • Quantizing the dataset numbers means you’re solving a different problem

• Quantizing the model numbers adds noise to each gradient step, and often means you can’t exactly represent the solution

• Quantizing the gradient numbers can add errors to each gradient step

• Quantizing the communication numbers can add errors which cause workers’ local models to diverge, which slows down convergence
Theoretical Guarantees for Low Precision

• Reducing precision adds noise in the form of round-off error.

• Two approaches to rounding:
  • biased rounding – round to nearest number
  • unbiased rounding – round randomly: $E[Q(x)] = x$

• I also proved we can combine low-precision computation with asynchronous execution, which we call BUCKWILD!

Using this, we can prove guarantees that SGD works with a low precision model.

Taming the Wild [NIPS 2015]
Why unbiased rounding?

- Imagine running SGD with a low-precision model with update rule

\[ w_{t+1} = \tilde{Q} \left( w_t - \alpha_t \nabla f(w_t; x_t, y_t) \right) \]

- Here, \( Q \) is an unbiased quantization function

- In expectation, this is just gradient descent

\[
\mathbb{E}[w_{t+1}|w_t] = \mathbb{E} \left[ \tilde{Q} \left( w_t - \alpha_t \nabla f(w_t; x_t, y_t) \right) | w_t \right] \\
= \mathbb{E} \left[ w_t - \alpha_t \nabla f(w_t; x_t, y_t) | w_t \right] \\
= w_t - \alpha_t \nabla f(w_t)
\]
Doing unbiased rounding efficiently

• We still need an efficient way to do unbiased rounding

• **Pseudorandom number generation can be expensive**
  • E.G. doing C++ rand or using Mersenne twister takes many clock cycles

• Empirically, we can use **very cheap** pseudorandom number generators
  • And still get good statistical results
  • For example, we can use XORSHIFT which is just a cyclic permutation
Memory Locality and Scan Order
Memory Locality: Two Kinds

- Memory locality is needed for **good cache performance**

- **Temporal locality**
  - Frequency of reuse of the same data within a short time window

- **Spatial locality**
  - Frequency of use of data nearby data that has recently been used

- **Where is there locality in stochastic gradient descent?**
Problem: no dataset locality across iterations

• The training example at each iteration is chosen randomly
  • Called a random scan order
  • Impossible for the cache to predict what data will be needed

$$w_{t+1} = w_t - \alpha_t \nabla f(w_t; x_t, y_t)$$

• Idea: process examples in the order in which they are stored in memory
  • Called a systematic scan order or sequential scan order
  • Does this improve the memory locality?
Random scan order vs. sequential scan order

• Much easier to prove theoretical results for random scan

• But sequential scan has better systems performance

• In practice, almost everyone uses sequential scan
  • There’s no empirical evidence that it’s statistically worse in most cases
  • Even though we can construct cases where using sequential scan does harm the convergence rate
Other scan orders

• **Shuffle-once**, then sequential scan
  • Shuffle the data once, then systematically scan for the rest of execution
  • Statistically very similar to random scan at the state

• **Random reshuffling**
  • Randomly shuffle on every pass through the data
  • Believed to be always at least as good as both random scan and sequential scan
  • But no proof that it is better
Questions?

• Upcoming things
  • Paper Review #8 — due today
  • Paper Presentation #9 on Wednesday