Low Precision Arithmetic
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
- RAM
Memory: The Multicore Picture
Socket 1

- Compute
- L1 cache
- L2 cache
- L3 cache
- RAM

Socket 2

- Compute
- L1 cache
- L2 cache
- L3 cache
- RAM
Memory: The Distributed Picture

Network

GPU

GPU
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 Arithmetic
Low-Precision Arithmetic

• 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
Low Precision and Memory

• Major benefit of low-precision: takes up less space
Low Precision and Parallelism

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

**SIMD Precision**
- 64-bit float vector
  - F64
  - F64
  - F64
  - F64
- 32-bit float vector
  - F32
  - F32
  - F32
  - F32
  - F32
  - F32
- 16-bit int vector
- 8-bit int vector

**SIMD Parallelism**
- 4 multiplies/cycle (vmulpd instruction)
- 8 multiplies/cycle (vmulps instruction)
- 16 multiplies/cycle (vpmaddwd instruction)
- 32 multiplies/cycle (vpmaddubsw instruction)
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
Numeric Formats in Machine Learning

How do we represent numbers as bit patterns on a computer?
A representative setup: DNN training

Many of the large-scale learning tasks we want to accelerate are deep learning tasks.

A deep neural network (DNN) looks like this:

Many layers connected to each other in series.

To train, we compute the loss gradient and run stochastic gradient descent: \[ w_{t+1} = w_t - \alpha_t \nabla f(w_t; x_t) \]
A representative setup: DNN training

- Standard method of computing gradient for SGD uses backpropagation.
- Computationally, it looks like this on the level of a single layer:

\[
\text{Weights} \rightarrow \text{activations} \rightarrow \text{backward} \rightarrow \text{Weight gradient} \rightarrow \text{storage} \rightarrow \text{activations}_{\text{prev}} \rightarrow \text{Weights}
\]

All of the signals here are vectors of real numbers.

But how are they stored on a computer?
The standard approach
Single-precision floating point (FP32)

• 32-bit floating point numbers

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

sign 8-bit exponent 23-bit mantissa

• Usually, the represented value is

\[
\text{represented number} = (-1)^{\text{sign}} \cdot 2^{\text{exponent}-127} \cdot 1.b_{22}b_{21}b_{20} \ldots b_0
\]

• Has a machine epsilon (measures relative error) of \( \varepsilon_{\text{machine}} \approx 6.0 \times 10^{-8} \)
An example

- Let’s convert the number 7.5 to floating point.

\[
6.5 = 13 \times 2^{-1} = (8 + 4 + 1) \times 2^{-1} \\
= 1101_b \times 2^{-1} = 1.101_b \times 2^2 \\
= 1.101_b \times 2^{129-127} \\
= 1.101_b \times 2^{10000001_b-127}
\]

1 10000001 1010000000000000000000000
What is the machine epsilon?

• Represents the relative error of the floating-point format
  • One half the distance between 1 and the next-largest floating point number
  • If there are \( m \) mantissa bits, \( \varepsilon_{\text{machine}} \approx 2^{-m-1} \)
  • Because the smallest representable number \( > 1 \) is \( 1 + 2^{-m} \)

**Relative error bound.** If \( x \in \mathbb{R} \) is any number in range of the format, and \( \hat{x} \) is the nearest number representable in the format, then

\[
|\hat{x} - x| \leq \varepsilon_{\text{machine}} \cdot |x|.
\]

Similarly, if \( x, y \in \mathbb{R} \) are two floating-point numbers, \( \ast \) is any primitive numerical operation (e.g. \(+, \times, \) etc.), and \( \oplus \) is the floating-point “version” of that op, then

\[
|(x \oplus y) - (x \ast y)| \leq \varepsilon_{\text{machine}} \cdot |x \ast y|.
\]
A low-precision alternative
FP16/Half-precision floating point

• 16-bit floating point numbers

```
[ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]  [ ]
```

1-bit  5-bit  10-bit
sign    exponent    significand

• Usually, the represented value is

\[ x = (-1)^{\text{sign bit}} \cdot 2^{\text{exponent} - 15} \cdot 1.\text{significand}_2 \]
Numeric properties of 16-bit floats

• A larger machine epsilon (larger rounding errors) of $\epsilon_{\text{machine}} = 4.9 \times 10^{-4}$
  • Compare 32-bit floats which had $\epsilon_{\text{machine}} \approx 6.0 \times 10^{-8}$

• A smaller overflow threshold (easier to overflow) at about $6.5 \times 10^4$
  • Compare 32-bit floats where it’s $3.4 \times 10^{38}$

• A larger underflow threshold (easier to underflow) at about $6.0 \times 10^{-8}$.
  • Compare 32-bit floats where it’s $1.4 \times 10^{-45}$

With all these drawbacks, does anyone use this?
Half-precision floating point support

- Supported on most **modern machine-learning-targeted GPUs**
  - E.g. efficient implementation as far back as NVIDIA Pascal GPUs

---

**Pascal Hardware Numerical Throughput**

<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**

Another common option

**Bfloat16** — “brain floating point”

- Another 16-bit floating point number

![Bfloat16 format diagram]

\[
\begin{array}{c|c|c|c}
\text{1-bit} & \text{8-bit} & \text{7-bit} & \varepsilon_{\text{machine}} = 3.9 \times 10^{-3} \\
\text{sign} & \text{exponent} & \text{significand}
\end{array}
\]

- Main benefit: numeric range is **now the same as single-precision float**
  - Since it looks like a truncated 32-bit float
  - This is useful because ML applications are **more tolerant to quantization error** than they are to overflow
An alternative to low-precision floating point
Fixed point numbers

• $p + q + 1$-bit fixed point number

The represented number is $p$-bit integer part and $q$-bit fractional part.

$1$-bit sign

$2^{q} \cdot \text{whole thing as signed integer}$

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

$= 2^{-q} \cdot \text{whole thing as signed integer}$
Arithmetic on fixed point numbers

• **Simple and efficient**
  • Can just use preexisting integer processing units
  • **Lower power** than floating point operations with the same number of bits

• **Mostly exact**
  • Can always convert to a higher-precision representation to avoid overflow

• Can represent a **much narrower range of numbers than float**

• **Has an absolute error bound, not relative error bound**
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
A powerful hybrid approach

**Block Floating Point**

• Motivation: when storing a vector of numbers, often these numbers all lie in the same range.
  • So they will have the same or similar exponent, if stored as floating point.

• **Block floating point** shares a single exponent among multiple numbers.

![Diagram showing block floating point with a 8-bit shared exponent]
A more specialized approach

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 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)
Low-precision formats in general

• These are some of the most common formats used in ML
  • …but we’re not limited to using only these formats!

• There are many other things we could try
  • For example, floating point numbers with different exponent/mantissa sizes
  • Fixed point numbers with nonstandard widths

• Problem: there’s no hardware support for these other things yet, so it’s hard to get a sense of how they would perform.
  • Need to simulate
Other Numerical Formats Used Rarely

• **BigFloats**
  • Higher-precision floating-point numbers that are implemented in software
  • Are sometimes necessary when you need very high precision, such as for very poorly conditioned problems

• **Exact arithmetic with rational numbers**
  • Lets you do arithmetic with no error
  • Numbers have variable length, because they require arbitrarily large integers
  • Can also support countable *field extensions of the rational numbers*
  • But these are very rarely used because of performance implications
Low-Precision SGD

Using low-precision arithmetic for training
How is precision used for training

- Recall our training diagram
  - Each of these signals forms a class of numbers
  - Generally, we assign a precision to each of the classes, and different classes can have different precisions

Number classes extended from “Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent,” ISCA 2017:

- Dataset numbers
- Model/weight numbers
- Gradient numbers
- Communication numbers
- Activation numbers
- Backward pass numbers
- Weight accumulator
- Linear layer accumulator
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: $E[Q(x)] = x$

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
  \[ w_{t+1} = \tilde{Q}(w_t - \alpha_t \nabla f(w_t; x_t, y_t)) \]
  • Here, \( Q \) is an unbiased quantization function

• In expectation, this is just gradient descent
  \[
  \mathbb{E}[w_{t+1} | w_t] = \mathbb{E} \left[ \tilde{Q}(w_t - \alpha_t \nabla f(w_t; x_t, y_t)) \right] | w_t
  \]
  \[
  = \mathbb{E} \left[ w_t - \alpha_t \nabla f(w_t; x_t, y_t) \right] | w_t
  \]
  \[
  = w_t - \alpha_t \nabla f(w_t)
  \]
Implementing unbiased rounding

• To implement an unbiased to-integer quantizer:
  
  sample $u \sim \text{Unif}[0, 1]$, then set $Q(x) = \lfloor x + u \rfloor$

• Why is this unbiased?

\[
\mathbb{E}[Q(x)] = \lfloor x \rfloor \cdot \Pr(Q(x) = \lfloor x \rfloor) + (\lfloor x \rfloor + 1) \cdot \Pr(Q(x) = \lfloor x \rfloor + 1)
\]
\[
= \lfloor x \rfloor + \Pr(Q(x) = \lfloor x \rfloor + 1) = \lfloor x \rfloor + \Pr([x + u] = \lfloor x \rfloor + 1)
\]
\[
= \lfloor x \rfloor + \Pr(x + u \geq \lfloor x \rfloor + 1) = \lfloor x \rfloor + \Pr(u \geq \lfloor x \rfloor + 1 - x)
\]
\[
= \lfloor x \rfloor + 1 + (\lfloor x \rfloor + 1 - x) = x.
\]
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
Benefits of Low-Precision Computation

Drawbacks of low-precision

• The draw back of low-precision arithmetic is the low precision!

• Low-precision computation means we accumulate more rounding error in our computations

• These rounding errors can add up throughout the learning process, resulting in less accurate learned systems

• The trade-off of low-precision: throughput/memory vs. accuracy
**Example: Low-Precision Neural Net**

![Graph](image)

**Performance eventually breaks down as precision is lowered**

(b) Test accuracies of low-precision SGD on LeNet neural network after 5000 passes, for various datasets.
Demo
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
  - Gets better upper bounds for SGD than with-replacement sampling
  - Very commonly used with deep learning
Demo
Questions?

• Upcoming things
  • Project feedback coming soon!