Low Precision Arithmetic and Quantization

CS6787 Lecture 9 — Spring 2024
Final Project Proposal Discussion

Split up into groups of 4–5
Do not be in a group with your project partners
Each of you presents a 2-minute pitch
Then discuss after everyone has pitched
Reminder: Final Project Requirements

• **Implement a machine learning system** to solve a problem

• Use one or more of the **techniques we discussed in class**
  • The mere use of a LLM in the project does not constitute a technique

• To achieve an **improvement over some baseline method**
  • Measuring both statistical performance and hardware performance
  • Or at least evaluate and attempt to achieve such a speedup

• Otherwise, very **open-ended**
  • **Groups of up to three**
Project proposals due **NEXT MONDAY**

- The main body should be about one page in length.
- It should describe the project you intend to do.
- It should contain at least one citation of a relevant paper that we did not cover in class.
- It should include some preliminary or exploratory work you've already done, that helps to support the idea that your project is feasible.
  - Don’t need a lot of work, just a nonzero amount of work supporting feasibility.
- In addition to the one-page text proposal, one short **experiment plan** per person
Experiment plan

• The hypothesis

• The proxy

• The protocol

• Expected results
Low Precision Arithmetic and Quantization

CS6787 Lecture 9 — Spring 2024
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

![Diagram showing memory hierarchy with compute units, L1, L2, L3 caches, and 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

**Precision in DRAM**
- 64-bit float vector
- 32-bit float vector
- 16-bit int vector
- 8-bit int vector

**Memory Throughput**
- 5 numbers/ns (assuming ~40 GB/sec memory bandwidth)
- 10 numbers/ns
- 20 numbers/ns
- 40 numbers/ns
Low Precision and Memory

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

Precision in DRAM

- 64-bit float vector
  - F64
- 32-bit float vector
  - F32
- 16-bit int vector
  - Int16
- 8-bit int vector
  - Int8

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
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:

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.

Weights

- $\text{activations}_{\text{prev}}$ → Weights → activations
- $\text{backward}$ → Weights → $\text{Weight gradient}$ → $\text{backward}_{\text{next}}$

Weight accumulator

$\text{Weight gradient}$

storage

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sign | 8-bit exponent | 23-bit mantissa |

• Usually, the represented value is

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 -6.5 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 101000000000000000000000000
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, \(\star\) is any primitive numerical operation (e.g. +, \(\times\), etc.), and \(\oplus\) is the floating-point “version” of that op, then

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

• 16-bit floating point numbers

\[ x = (-1)^{\text{sign bit}} \cdot 2^{\text{exponent} - 15} \cdot 1.\text{significand}_2 \]
What benefits can we expect to get from computing in lower precision?

In comparison to single-precision floating point numbers, half-precision floats have:

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

- 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

- Good empirical results for deep learning

---


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/
Another common option

Bfloat16 — “brain floating point”

• Another 16-bit floating point number

```
1-bit  8-bit  7-bit
sign   exponent  significand
```

\[ \varepsilon_{\text{machine}} = 3.9 \times 10^{-3} \]

• 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
A more recent option

fp8

• An 8-bit floating point number: e5m2

• Now supported in TensorCores on NVIDIA GPUs
A more recent option

**fp8**

- An 8-bit floating point number: e4m3

![Diagram of fp8 format]

1-bit sign
4-bit exponent
3-bit significand

- Now supported in TensorCores on NVIDIA GPUs
An alternative to low-precision floating point

Fixed point numbers

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

The represented number is

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

\[
= 2^{-q} \cdot \text{whole thing as signed integer}
\]
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

- Some support for **4-bit int on GPUs**
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.
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)
Vector quantization

• Group $K$ numbers together into a $K$-dimensional vector and quantize this to a $K$-dimensional codebook.

• Ways to make the codebook:
  • Choose via k-Means
  • Choose it to be a lattice with nice packing properties
    • E.g. E8 lattice
  • Learn it via SGD

• What is this most useful for? Weights? Activations?
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
Quantization-Aware Training & The Straight-Through Estimator

• A “function” where

\[ Q_{\text{straight-thru}}(x) = \text{round}(x) \]

\[ Q'_{\text{straight-thru}}(x) = 1 \]

• Just round in the forward pass, and pretend the round is not there in the backward pass.
A modern recipe for training in low-precision: Mixed Precision Training

- Use fp16 to store model and activations wherever this is possible without significant loss of precision

- Use **loss scaling** to stop small gradient values & backward signals from underflowing

- Keep optimizer state in fp32

Published as a conference paper at ICLR 2018

**Mixed Precision Training**

Sharan Narang*, Gregory Diamos, Erich Elsen
Baidu Research
{sharan, gdiamos}@baidu.com

Paulius Micikevicius*, Jonah Alben, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, Hao Wu
NVIDIA
{pauliusm, alben, dagarcia, bginsburg, mhouston, okuchaiev, gavenkatesh, skyw}@nvidia.com
Theoretical Guarantees for Low Precision

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

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

Using this, we can prove guarantees that SGD converges with a low precision model.
Why stochastic 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

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

• To implement an unbiased to-integer quantizer:

\[ u \sim \text{Unif}[0, 1], \text{ then set } Q(x) = \lfloor x + u \rfloor \]

• Why is this unbiased?

\[
\mathbb{E}[Q(x)] = \lfloor x \rfloor \cdot P(Q(x) = \lfloor x \rfloor) + (\lfloor x \rfloor + 1) \cdot P(Q(x) = \lfloor x \rfloor + 1) \\
= \lfloor x \rfloor + P(Q(x) = \lfloor x \rfloor + 1) = \lfloor x \rfloor + P(\lfloor x + u \rfloor = \lfloor x \rfloor + 1) \\
= \lfloor x \rfloor + P(x + u \geq \lfloor x \rfloor + 1) = \lfloor x \rfloor + P(u \geq \lfloor x \rfloor + 1 - x) \\
= \lfloor x \rfloor + 1 + (\lfloor x \rfloor + 1 - x) = x.
\]
Doing stochastic 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
Limitations of stochastic rounding

• Technique only makes sense when we’re summing up a bunch of independently rounded values

• Works best for the accumulators in the optimizer!

• But in the Mixed Precision recipe, we store those accumulators in full-precision anyway
  • ...so there’s not much point in the stochastic rounding

• Also it introduces a lot of noise to the training process.
Benefits of low-precision
On a real device...

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>H100 SXM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP64</td>
<td>34 teraFLOPS</td>
</tr>
<tr>
<td>FP64 Tensor Core</td>
<td>67 teraFLOPS</td>
</tr>
<tr>
<td>FP32</td>
<td>67 teraFLOPS</td>
</tr>
<tr>
<td>TF32 Tensor Core</td>
<td>989 teraFLOPS²</td>
</tr>
<tr>
<td>BFLOAT16 Tensor Core</td>
<td>1,979 teraFLOPS²</td>
</tr>
<tr>
<td>FP16 Tensor Core</td>
<td>1,979 teraFLOPS²</td>
</tr>
<tr>
<td>FP8 Tensor Core</td>
<td>3,958 teraFLOPS²</td>
</tr>
<tr>
<td>INT8 Tensor Core</td>
<td>3,958 TOPS²</td>
</tr>
</tbody>
</table>

Drawbacks of low-precision

• The drawback 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**
Classical Example: Low-Precision Neural Net

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.
A modern LLM example

Figure 5: Quantizing OPT models up to 66B parameters. Our method QuIP is the first PTQ procedure to achieve good quantization at 2 bits per weight, across a variety of model sizes and evaluation tasks.
Next time...

• **Post-training quantization** & **compression**!

• **How can we leverage low-precision arithmetic to make large models work on small devices?**
Questions?

• Upcoming things
  • Project proposal due NEXT MONDAY!