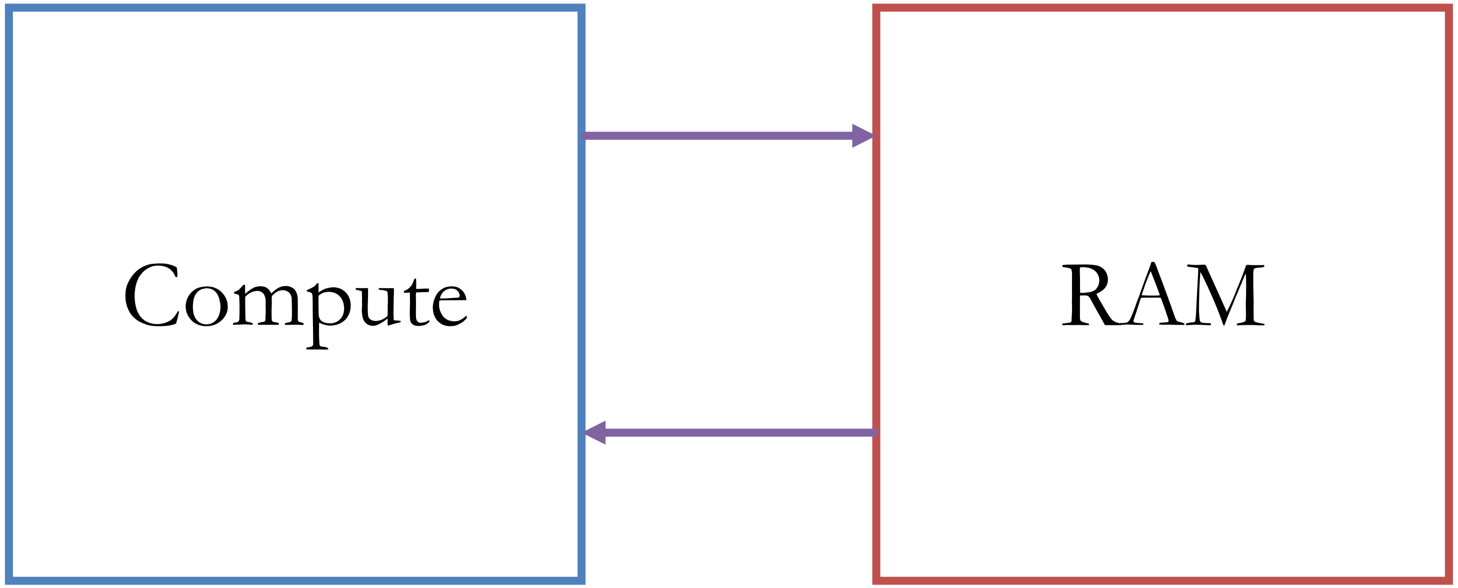
Memory Bandwidth and Low Precision Computation

CS6787 Lecture 10 — Fall 2019

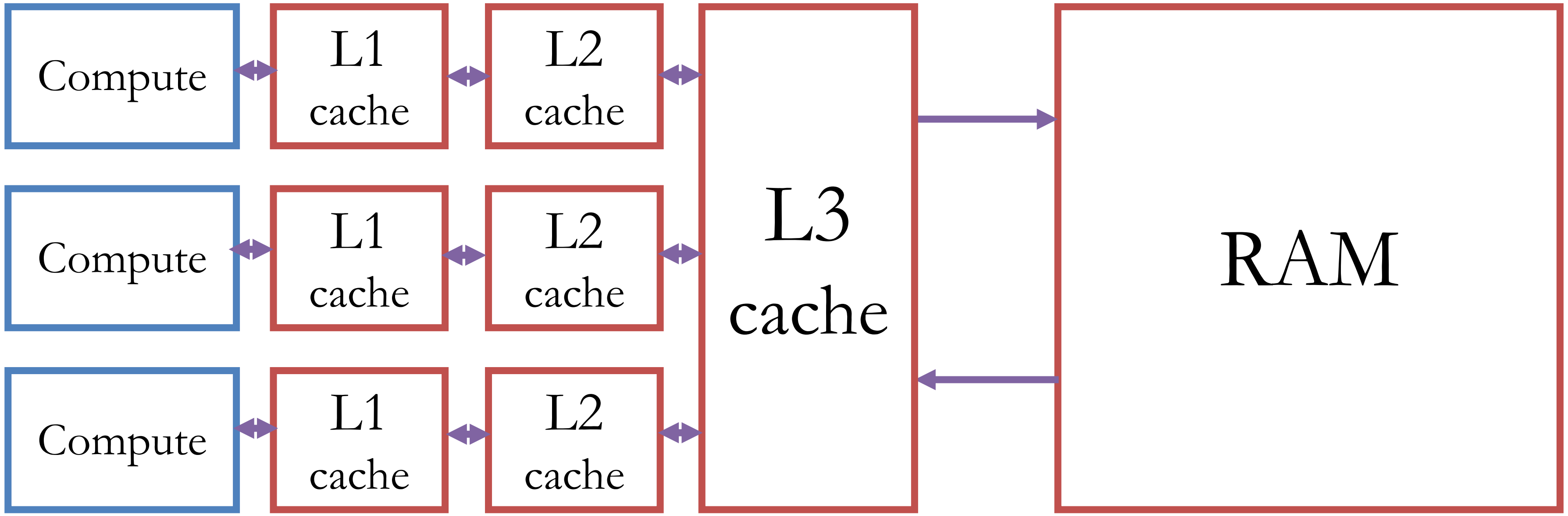
Memory as a Bottleneck

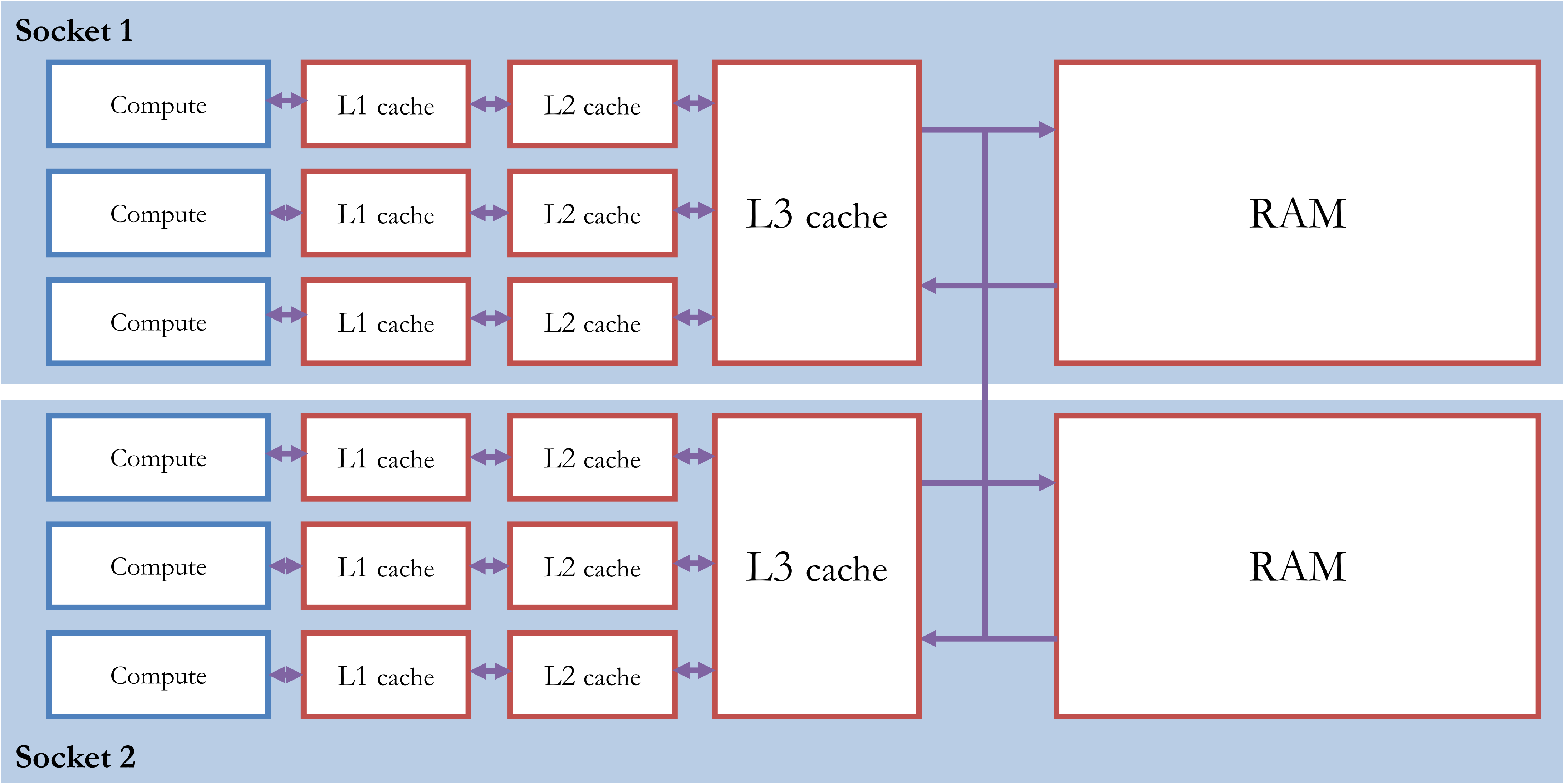
- So far, we've just been talking about **compute**
  - e.g. techniques to decrease the amount of compute by decreasing iterations
- But machine learning systems need to process **huge amounts of data**
- Need to **store, update, and transmit** this data
- As a result: **memory** is of critical importance
  - Many applications are memory-bound

Memory: The Simplified Picture



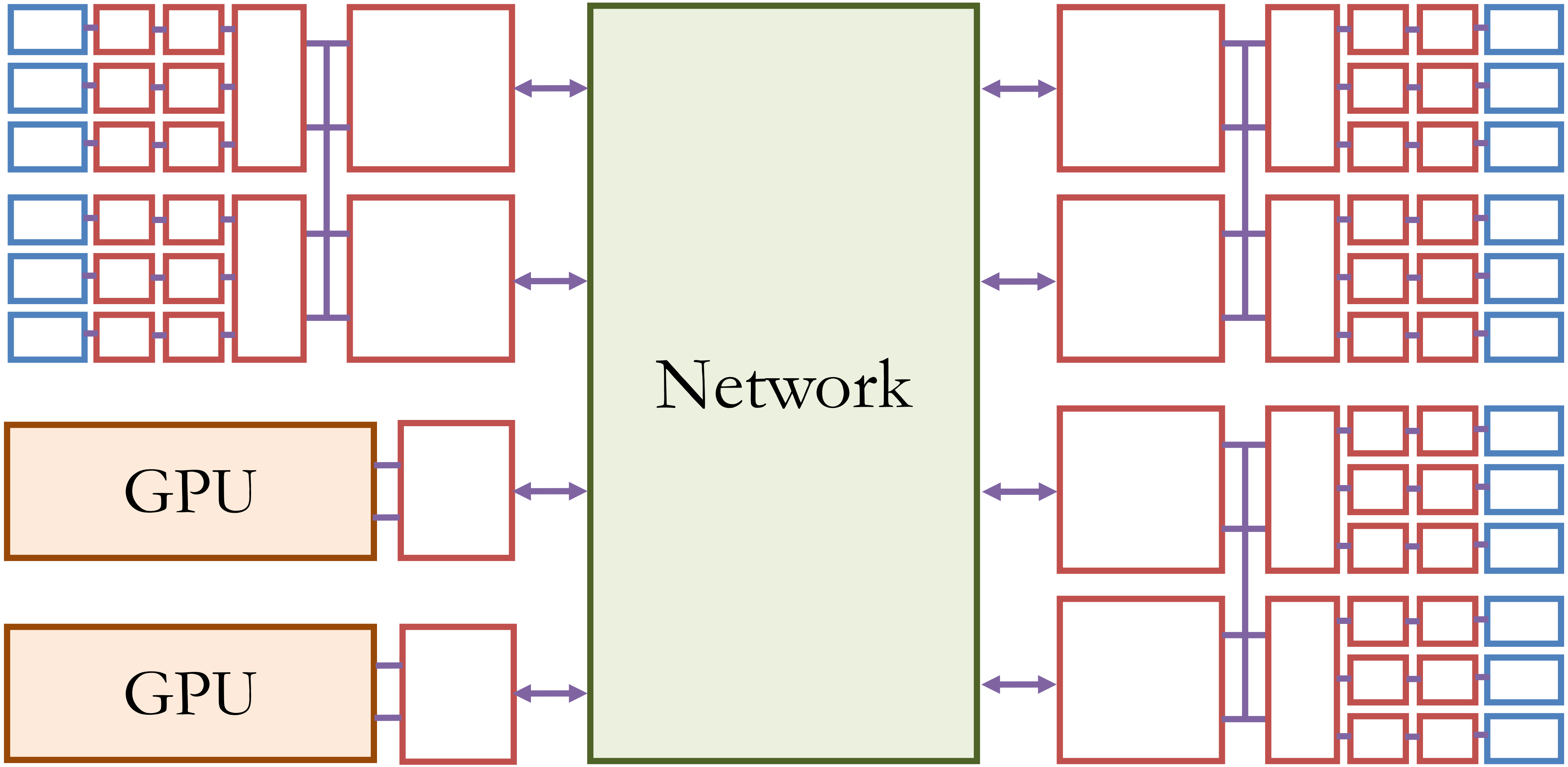
Memory: The Multicore Picture





Socket 1
Compute
L1 cache
L2 cache
Compute
L1 cache
L2 cache
Compute
L1 cache
L2 cache
L3 cache
RAM
Compute
L1 cache
L2 cache
Compute
L1 cache
L2 cache
Compute
L1 cache
L2 cache
L3 cache
RAM
Socket 2

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

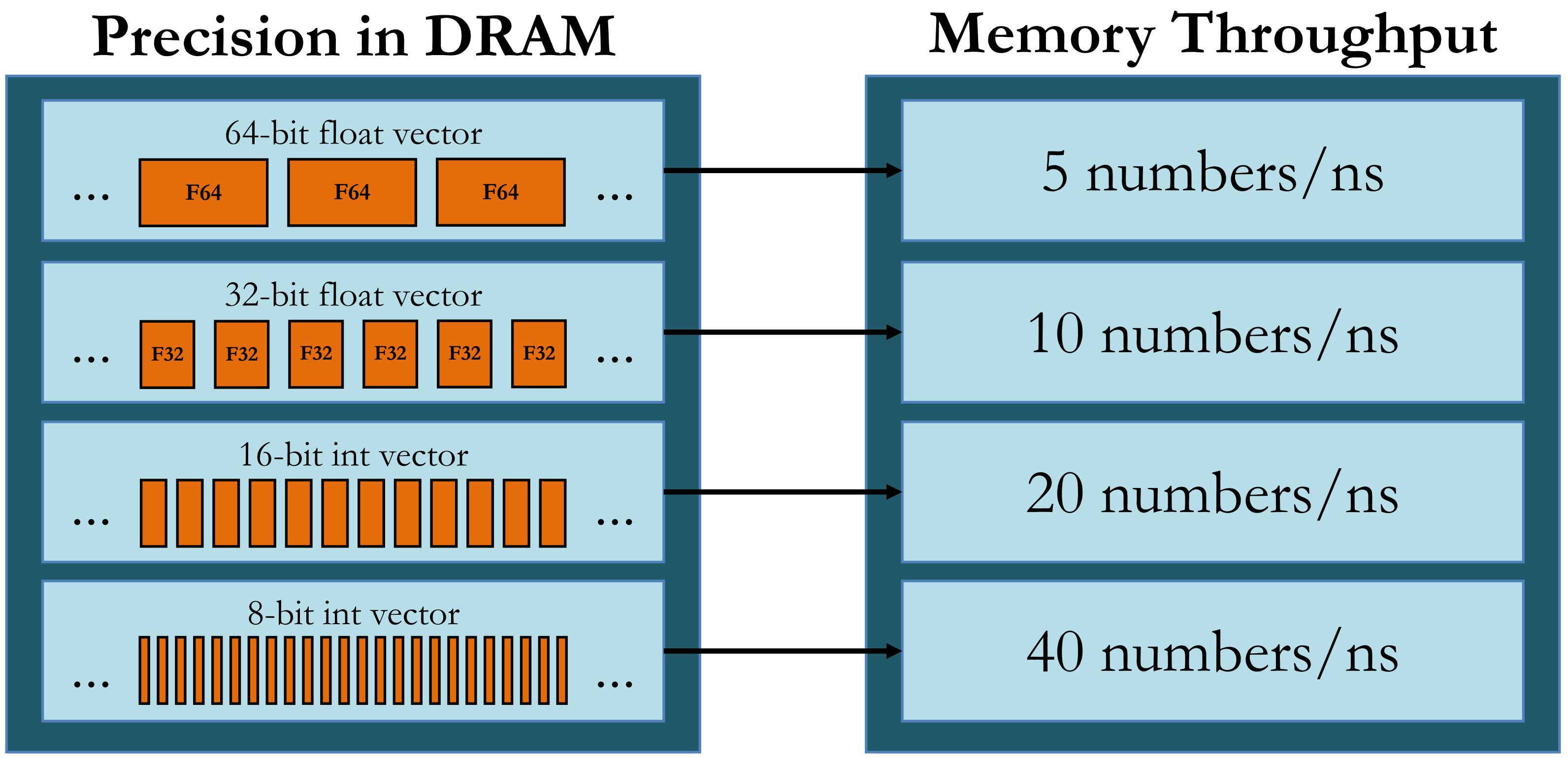
- Traditional ML systems use 32-bit or 64-bit **floating point numbers**

- **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

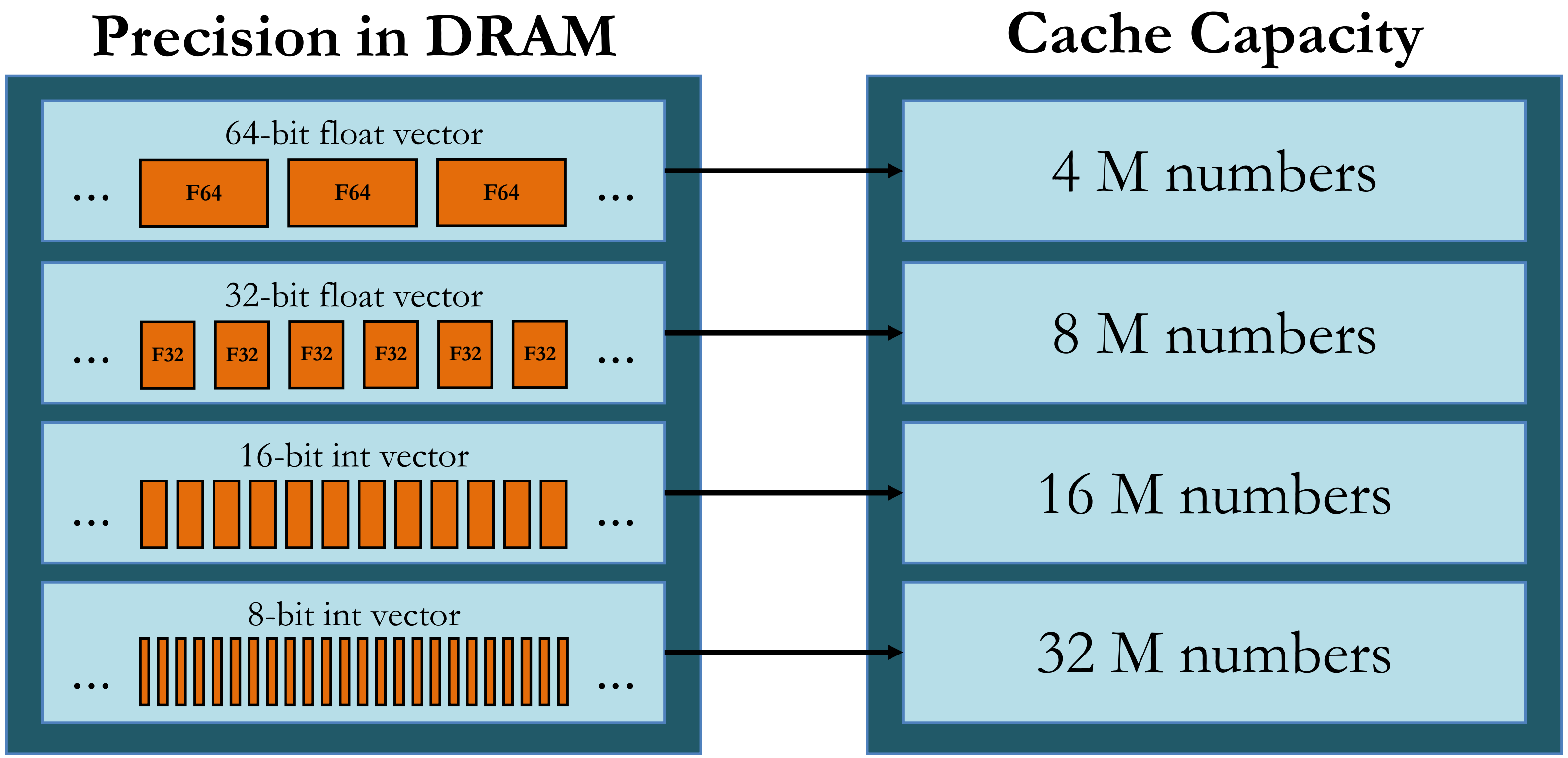
- Major benefit of low-precision: **uses less memory bandwidth**



(assuming ~40 GB/sec memory bandwidth)

Low Precision and Memory

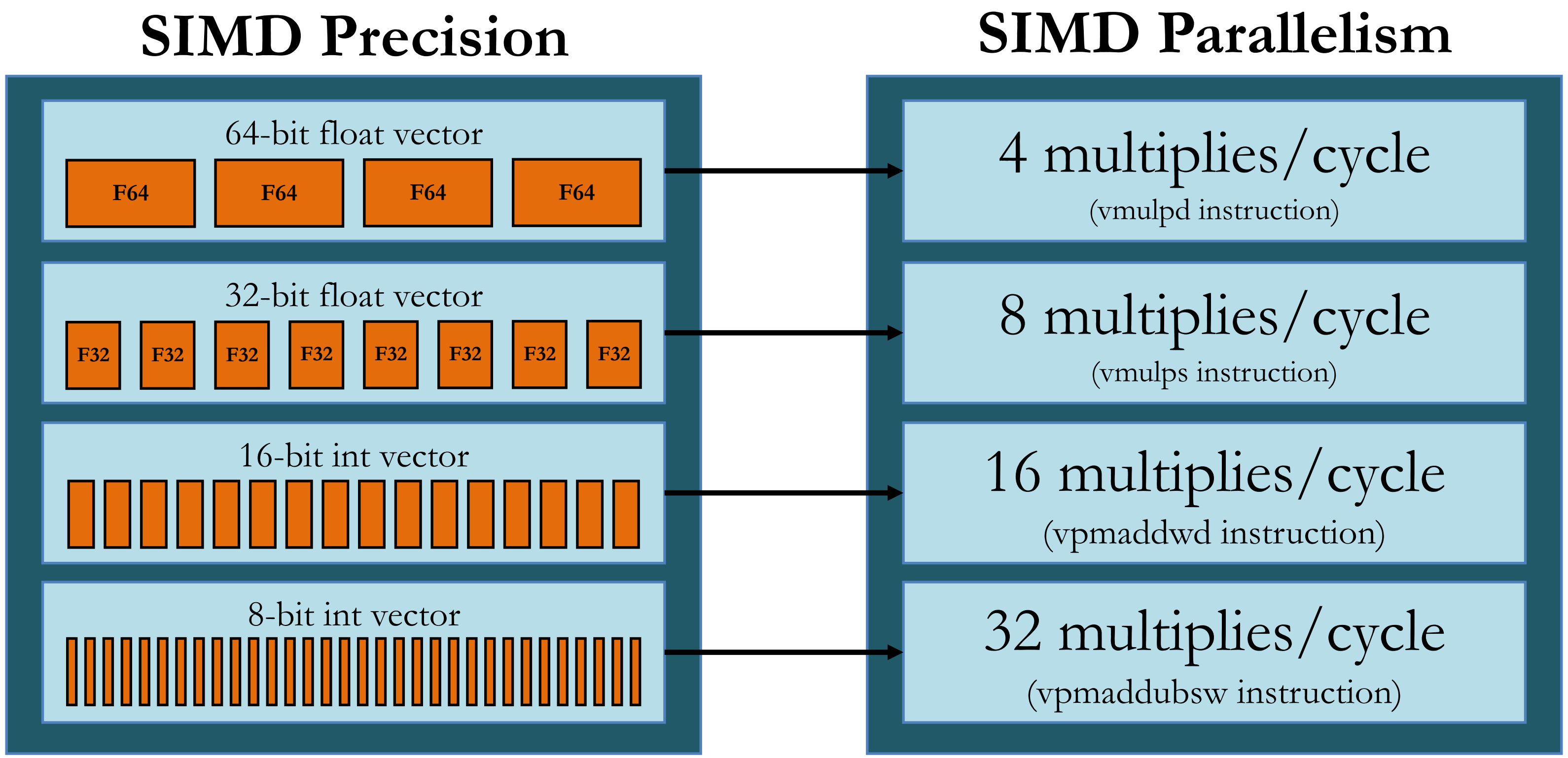
- Major benefit of low-precision: **takes up less space**



(assuming ~32 MB cache)

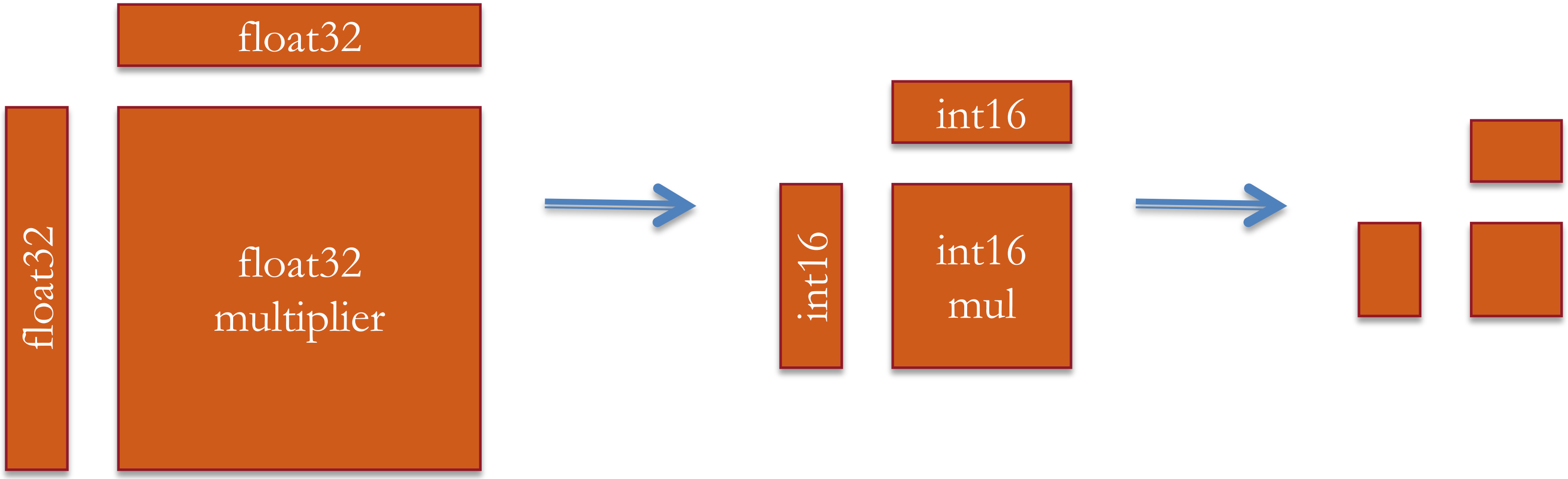
Low Precision and Parallelism

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

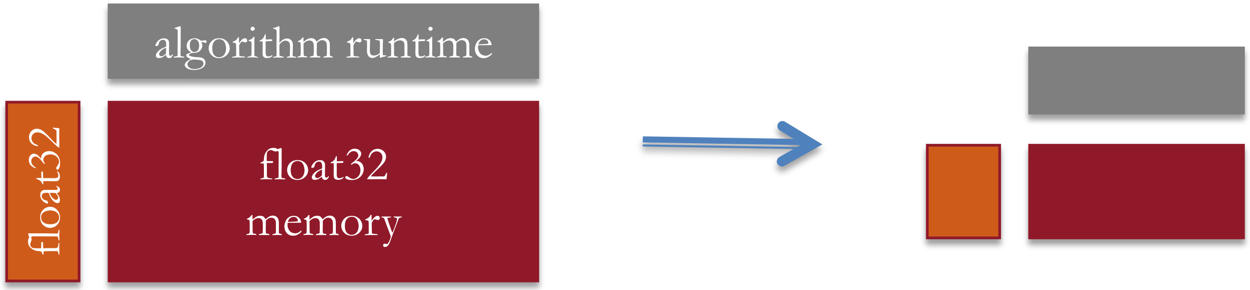


Low Precision and Power

- 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

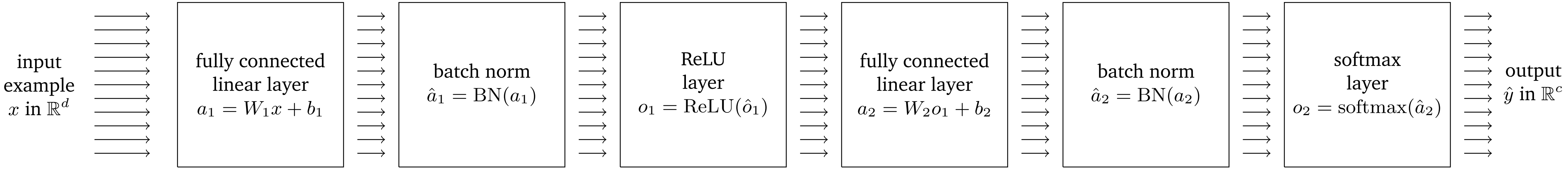
Numeric Formats in Machine Learning

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:



Many layers connected to each other in series.

To train, we compute the loss gradient and run stochastic gradient descent:

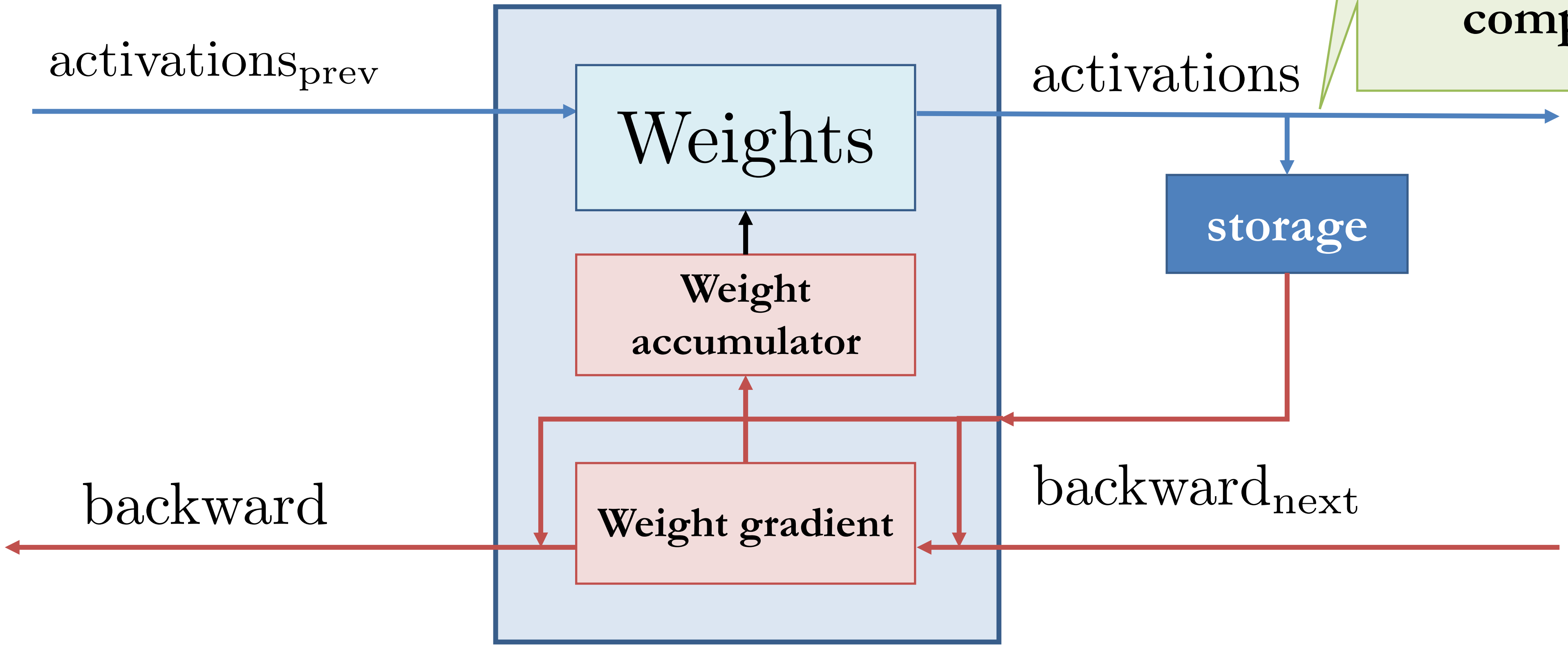
$$w_{t+1} = w_t - \alpha_t \nabla f(w_t; x_t)$$

A representative setup: DNN training

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

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

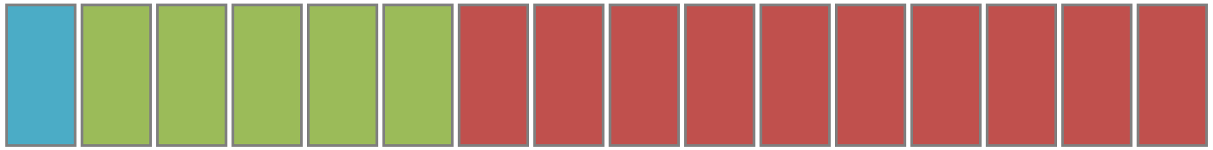
- Usually, the represented value is

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

- Has a machine epsilon (measures relative error) of $\epsilon_{\text{machine}} \approx 1.2 \times 10^{-7}$.

A low-precision alternative
FP16/Half-precision floating point

- 16-bit floating point numbers



1-bit sign | 5-bit exponent | 10-bit 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 $\epsilon_{\text{machine}} \approx 9.8 \times 10^{4}$
  - Compare 32-bit floats which had $\epsilon_{\text{machine}} \approx 1.2 \times 10^{-7}$.
- A smaller overflow threshold (**easier to overflow**) at about 6.5×10^{4}
  - Compare 32-bit floats where it's 3.4×10^{38}
- A larger underflow threshold (**easier to underflow**) at about 6.0×10^{-8}.
  - Compare 32-bit floats where it's 1.4×10^{-45}

With all these drawbacks, does anyone use this?

Half-precision floating point support

- Supported on most **modern machine-learning-targeted GPUs**
  - Including efficient implementation on NVIDIA Pascal GPUs

Pascal Hardware Numerical Throughput

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

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**

Micikevicius et al. "Mixed Precision Training." on arxiv, 2017.

Another common option
Bfloat16 — “brain floating point”

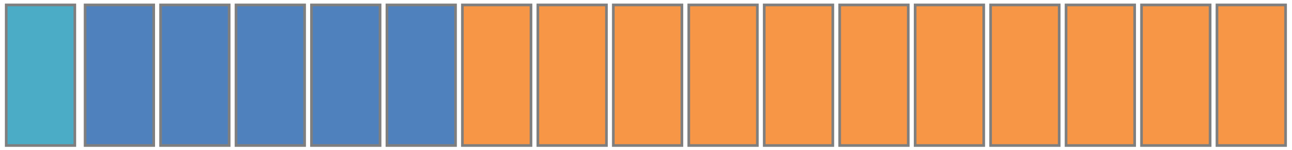
- Another 16-bit floating point number

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

- 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

$$x = (-1)^{\text{sign bit}} \left(\text{integer part} + 2^{-q} \cdot \text{fractional part} \right)$$
$$= 2^{-q} \cdot \text{whole thing as signed integer}$$

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**
  - 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**

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

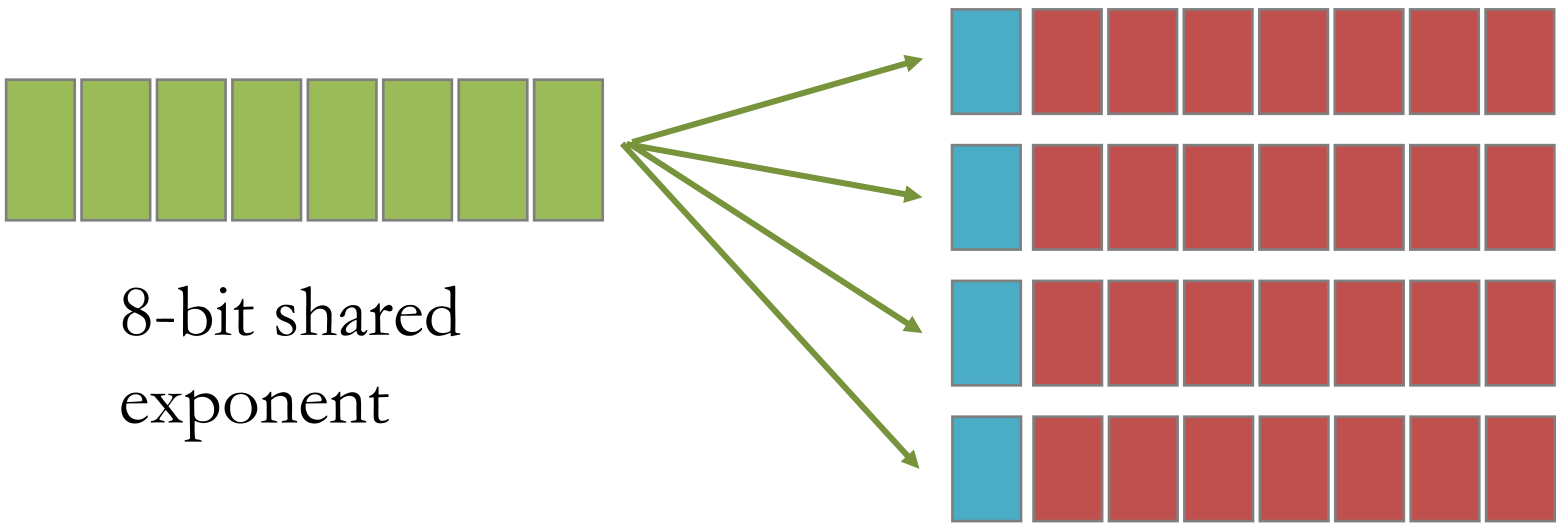
- Particularly effective on **FPGAs and ASICs**
  - Where floating point units are costly

- 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.



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 **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)

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
  - Block floating point numbers with different block sizes/layouts
  - Fixed point numbers with nonstandard widths

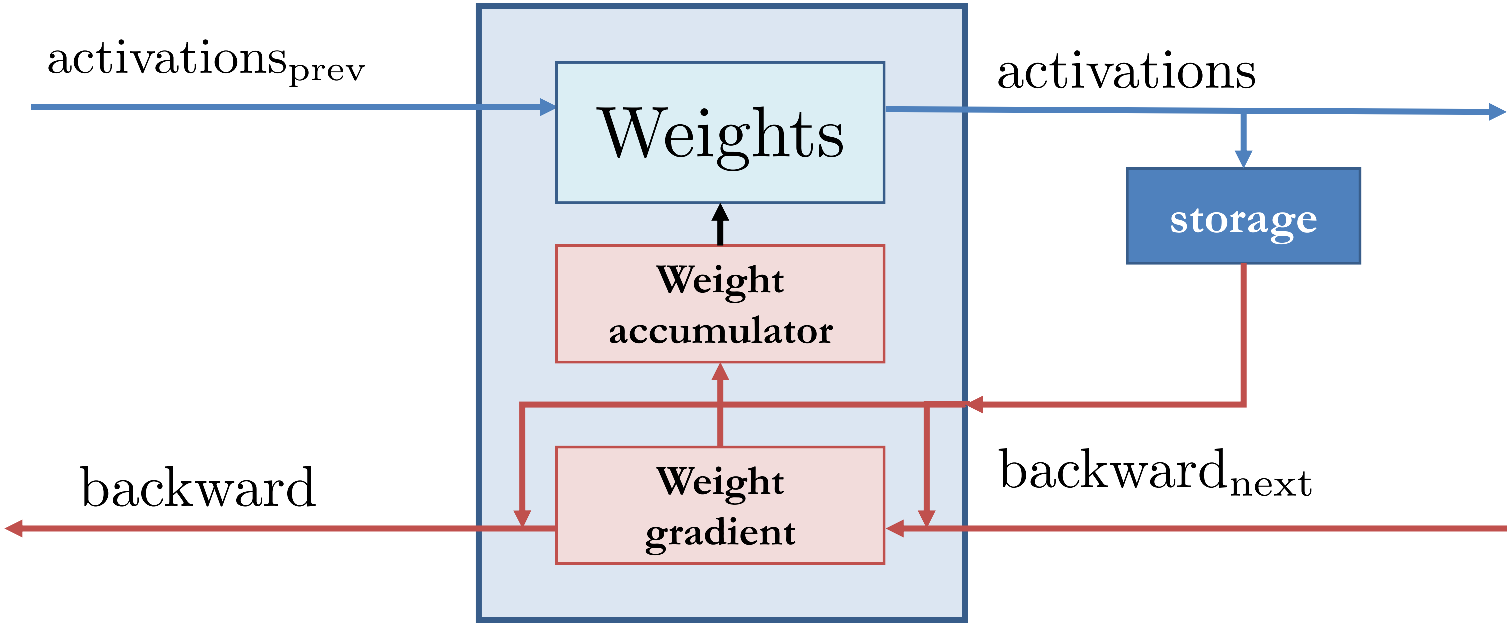
- Problem: there's **no hardware support** for these other things yet, so it's hard to get a sense of how they would perform.

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:

- **D**ataset numbers
- **M**odel/weight numbers
- **G**radient numbers
- **C**ommunication numbers
- Activation numbers
- Backward pass numbers
- Weight 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

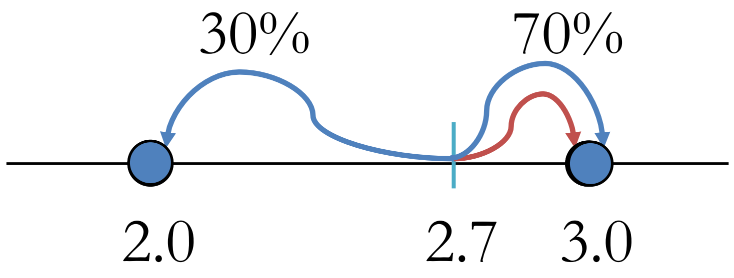
Theoretical Guarantees for Low Precision

- Reducing precision adds noise in the for

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

Taming the Wild [NIPS 2015]

- Two approaches to rounding:
  - **biased rounding** – round to nearest number
  - **unbiased rounding** – round randomly: $\boldsymbol{E}[Q(x)] = x$



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**

$$\begin{aligned}
\mathbf{E}[w_{t+1} | w_t] &= \mathbf{E}\left[\tilde{Q}\left(w_t - \alpha_t \nabla f(w_t; x_t, y_t)\right) \middle| w_t\right] \\
&= \mathbf{E}\left[w_t - \alpha_t \nabla f(w_t; x_t, y_t) | w_t\right] \\
&= w_t - \alpha_t \nabla f(w_t)
\end{aligned}$$

Implementing unbiased rounding

- To implement an unbiased to-integer quantizer:

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

- Why is this unbiased?

$$\begin{aligned}
\mathbf{E}[Q(x)] &= \lfloor x \rfloor \cdot \mathbf{P}(Q(x) = \lfloor x \rfloor) + (\lfloor x \rfloor + 1) \cdot \mathbf{P}(Q(x) = \lfloor x \rfloor + 1) \\
&= \lfloor x \rfloor + \mathbf{P}(Q(x) = \lfloor x \rfloor + 1) = \lfloor x \rfloor + \mathbf{P}(\lfloor x + u \rfloor = \lfloor x \rfloor + 1) \\
&= \lfloor x \rfloor + \mathbf{P}(x + u \geq \lfloor x \rfloor + 1) = \lfloor x \rfloor + \mathbf{P}(u \geq \lfloor x \rfloor + 1 - x) \\
&= \lfloor x \rfloor + 1 + (\lfloor x \rfloor + 1 - x) = x.
\end{aligned}$$

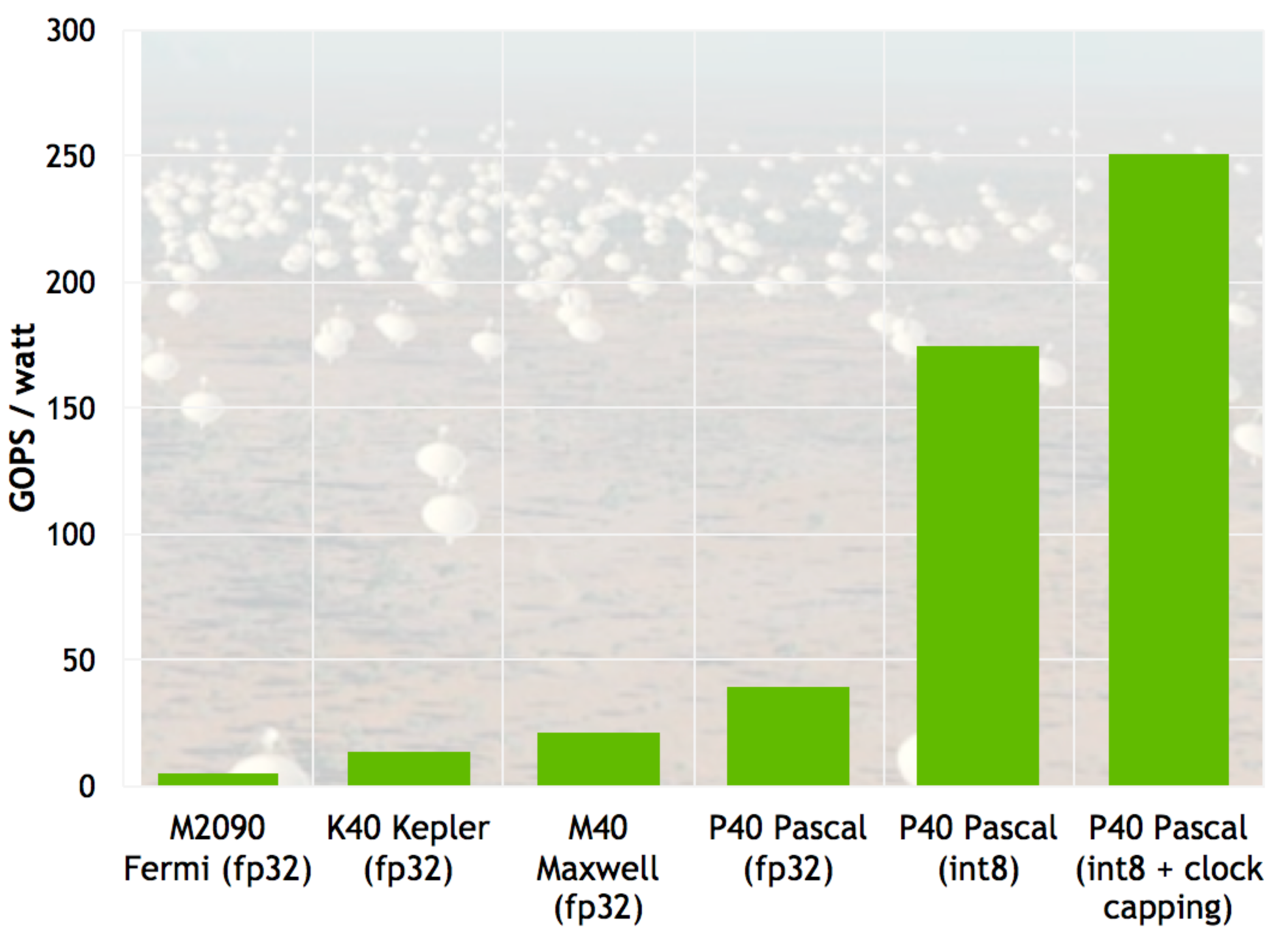
Doing unbiased rounding efficiently

- 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



From https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/

Drawbacks of low-precision

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

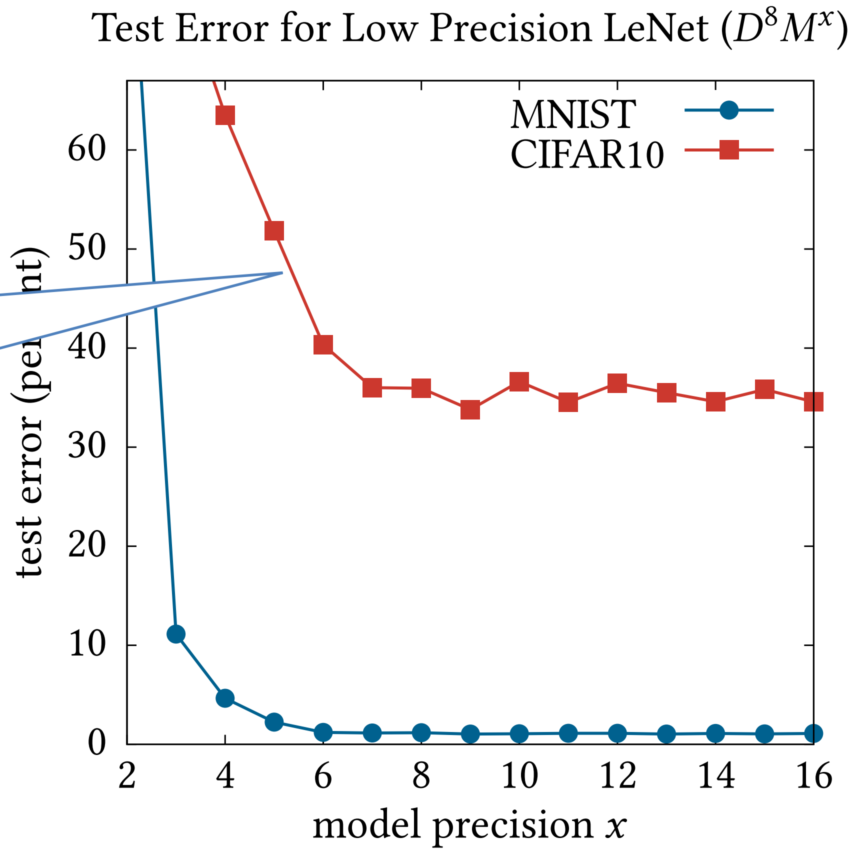
- Low-precision computation means we accumulate **more rounding error** in our computations

- 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

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
  - Believed to be always at least as good as both random scan and sequential scan
  - But no proof that it is better

Demo

Questions?

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