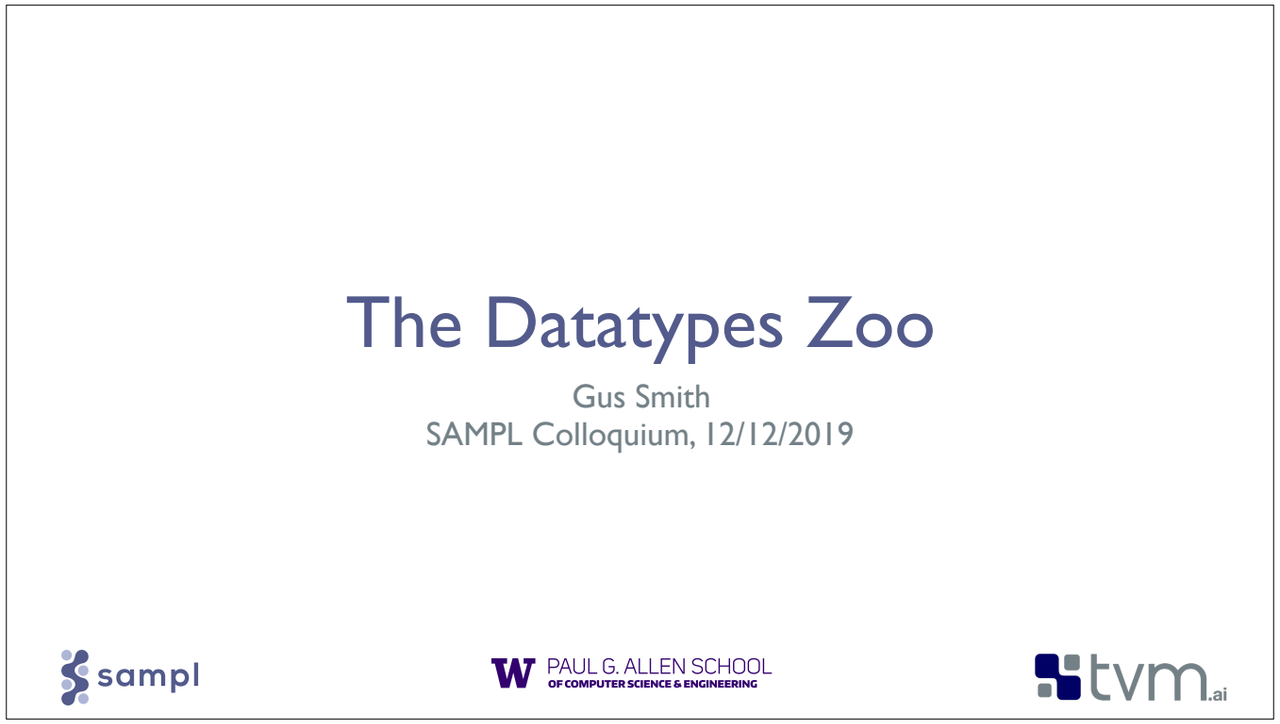
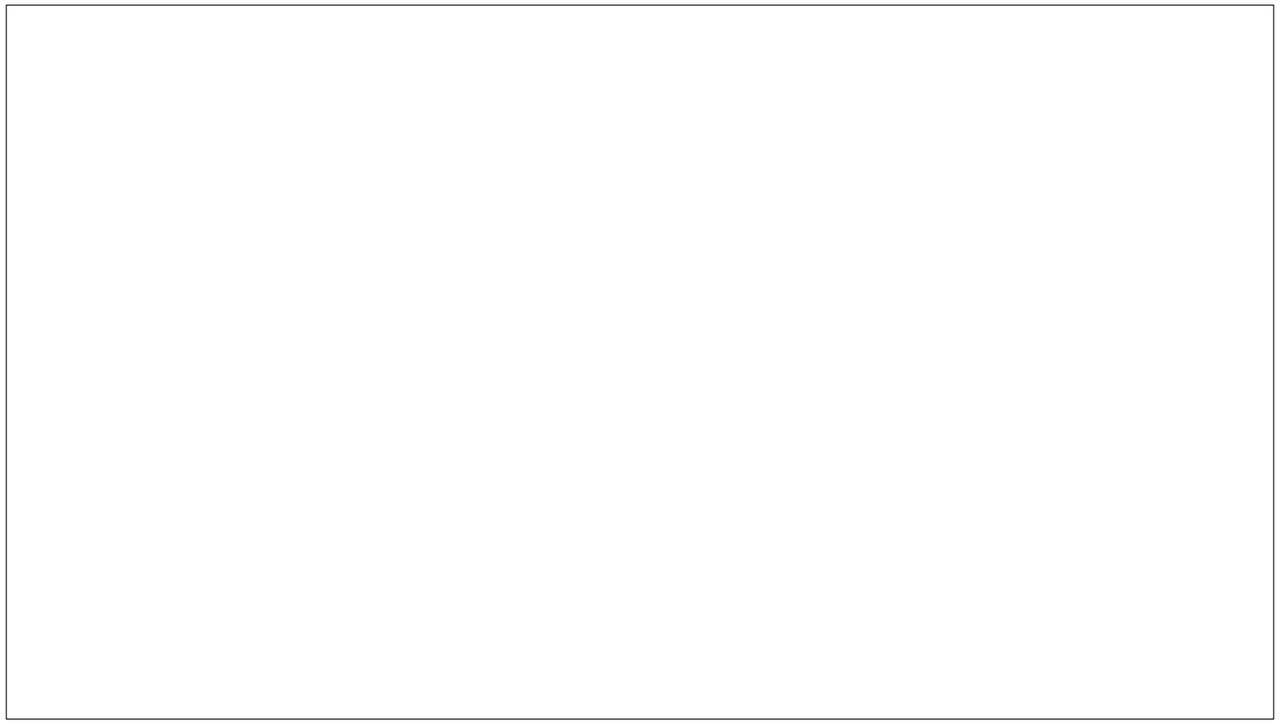


Hey all, for those that don’t know me, my name’s Gus. I’ve been working with Luis and Zach on exploring datatypes in deep learning.
Today, I want to talk about the world of datatypes that I’ve been introduced to over the past year. None of this is actually my work — it’s just a bunch of interesting stuff I’ve learned along the way!



So what do I actually mean by datatypes? Because each subfield of computer science has its own definition—if not multiple definitions—of this word.
I'm talking specifically about numerical datatypes, which specify how real numbers are stored in hardware—how a series of, say, 32 bits, can map to a real number.

By “datatypes”, we mean **numerical datatypes:** how the hardware represents and operates on real numbers.

IEEE 754 Floating Point

For more than thirty years, we've had a solution to this problem: the IEEE 754 standard for floating point arithmetic.
IEEE 754 splits a fixed number of bits—such as 32, for the so-called single-precision "floats" that we're used to—into three fields:
[build] a sign bit,
[build] a group of exponent bits,
[build] and a group of fraction bits.
The resulting number is calculated as
[build] the sign, times
[build] two to the exponent times
[build] one dot the fraction bits.

So then, we can see that the exponent field is in control of the magnitude of numbers.
In datatype terms, we call this the
[build] dynamic range — the dynamic range describes how large and how small of numbers a datatype can describe.

Furthermore, we can see that the fraction field determines the
[build] precision of numbers—that is, how many digits we get after the radix point.

These two terms—dynamic range and precision—will keep coming up throughout the talk!

So that's how IEEE 754 floats work.

IEEE 754 was designed to be flexible for many applications, and also specifies floats of other sizes, such as
[build] halves and
[build] doubles.
Because it was designed to be so flexible, it has
[build] remained an industry standard for more than thirty years!

IEEE 754 Floating Point

Single precision (32 bit) 

IEEE 754 Floating Point

Single precision (32 bit)

IEEE 754 Floating Point

IEEE 754 Floating Point

value ≈ sign

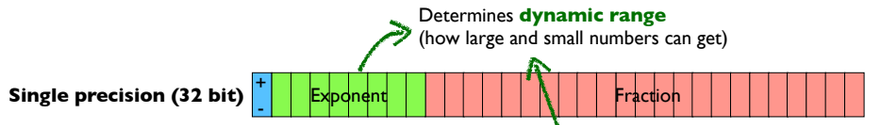
IEEE 754 Floating Point

$$\text{value} \approx \text{sign} * 2^{\text{exponent}}$$

IEEE 754 Floating Point

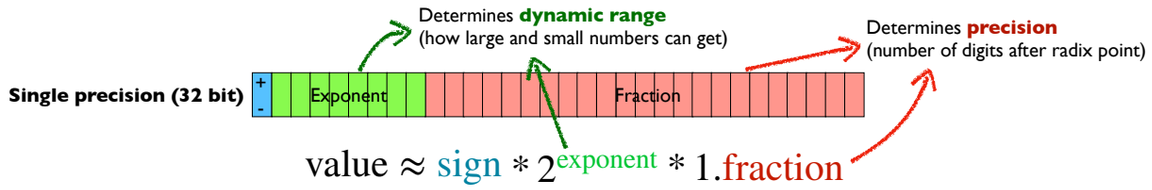
$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

IEEE 754 Floating Point

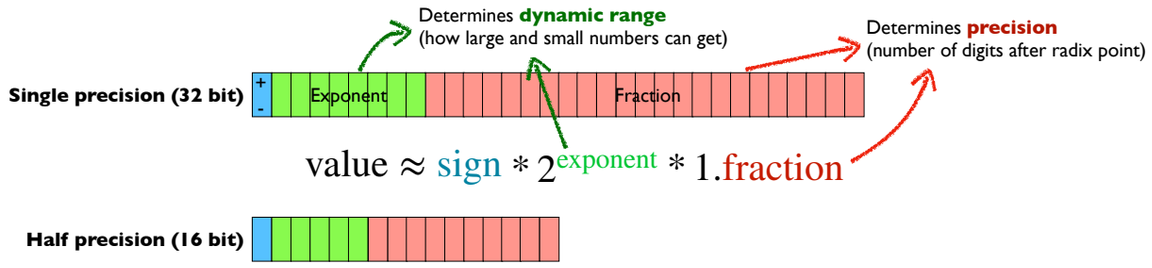


$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

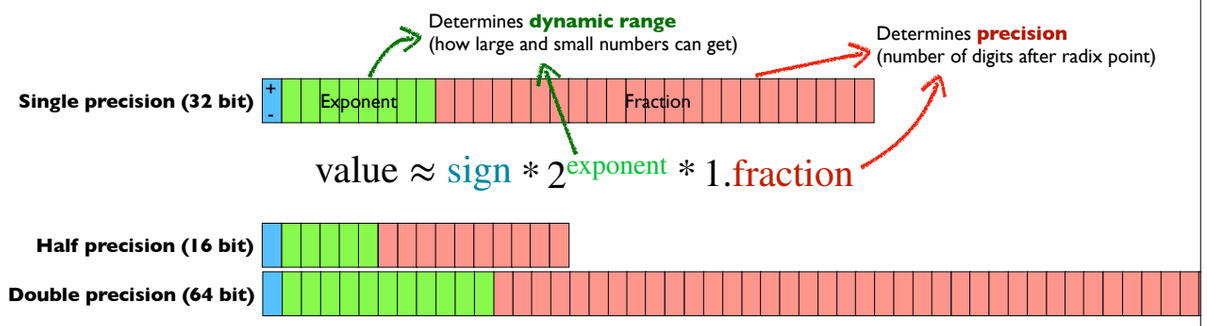
IEEE 754 Floating Point



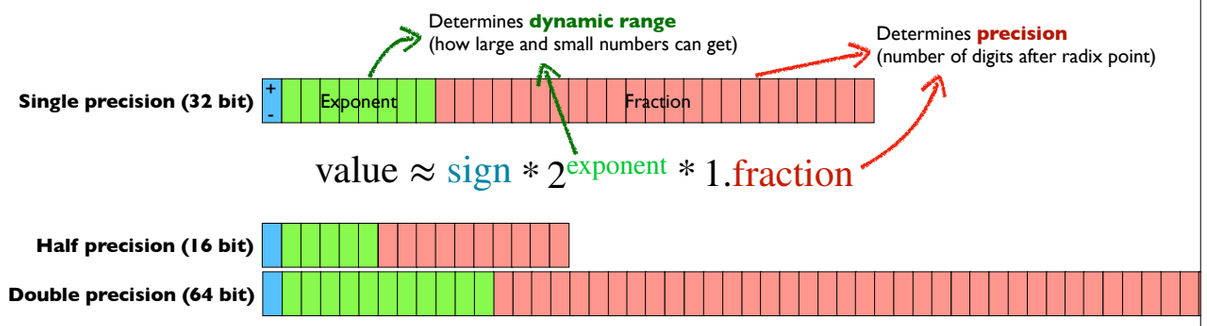
IEEE 754 Floating Point



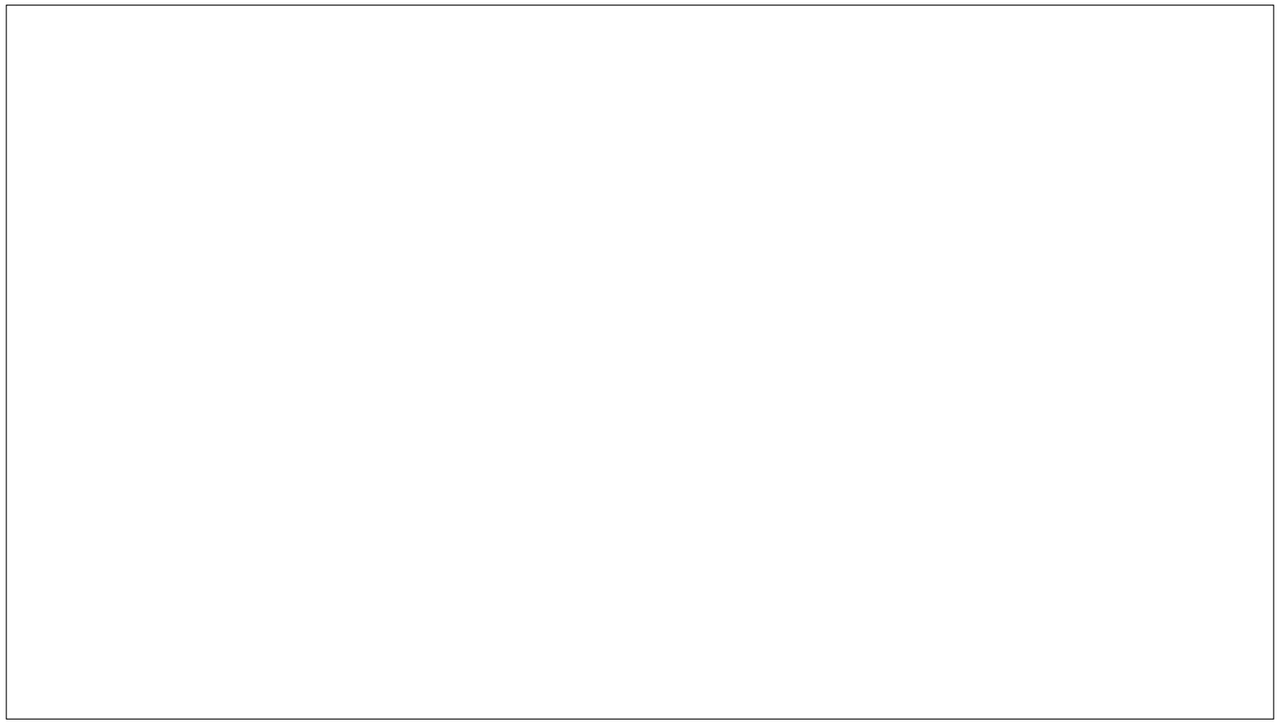
IEEE 754 Floating Point



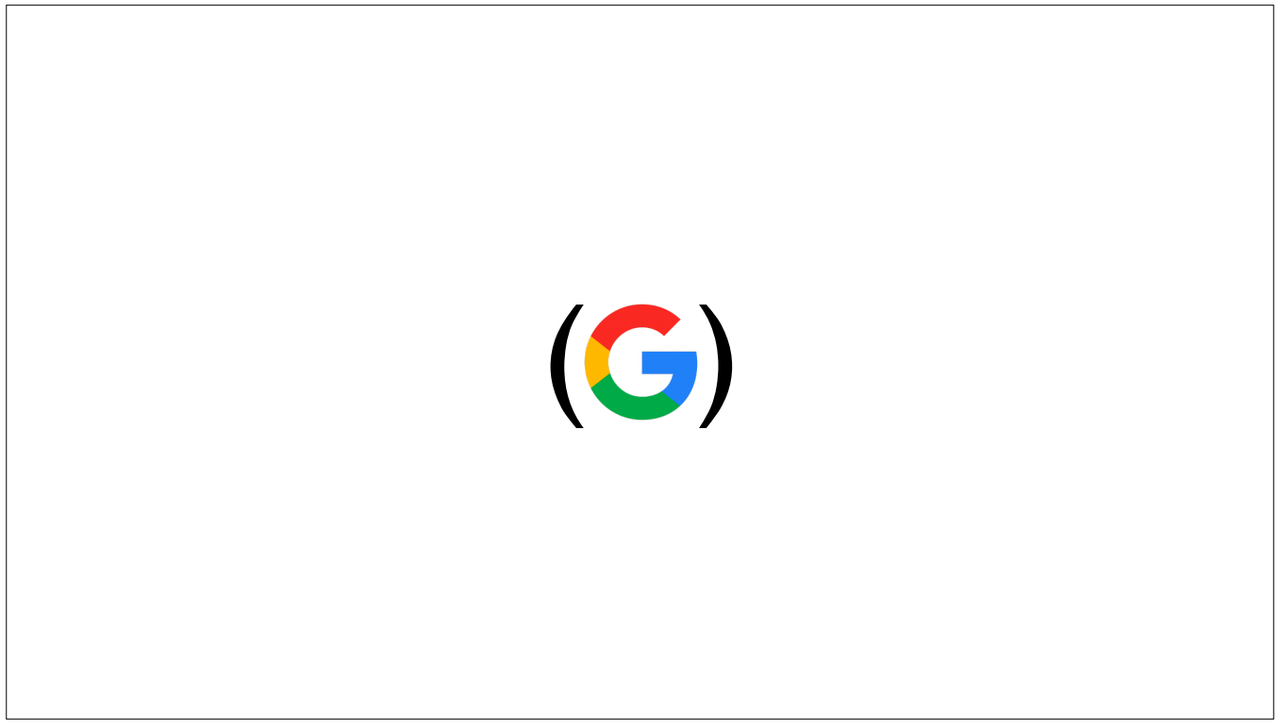
IEEE 754 Floating Point

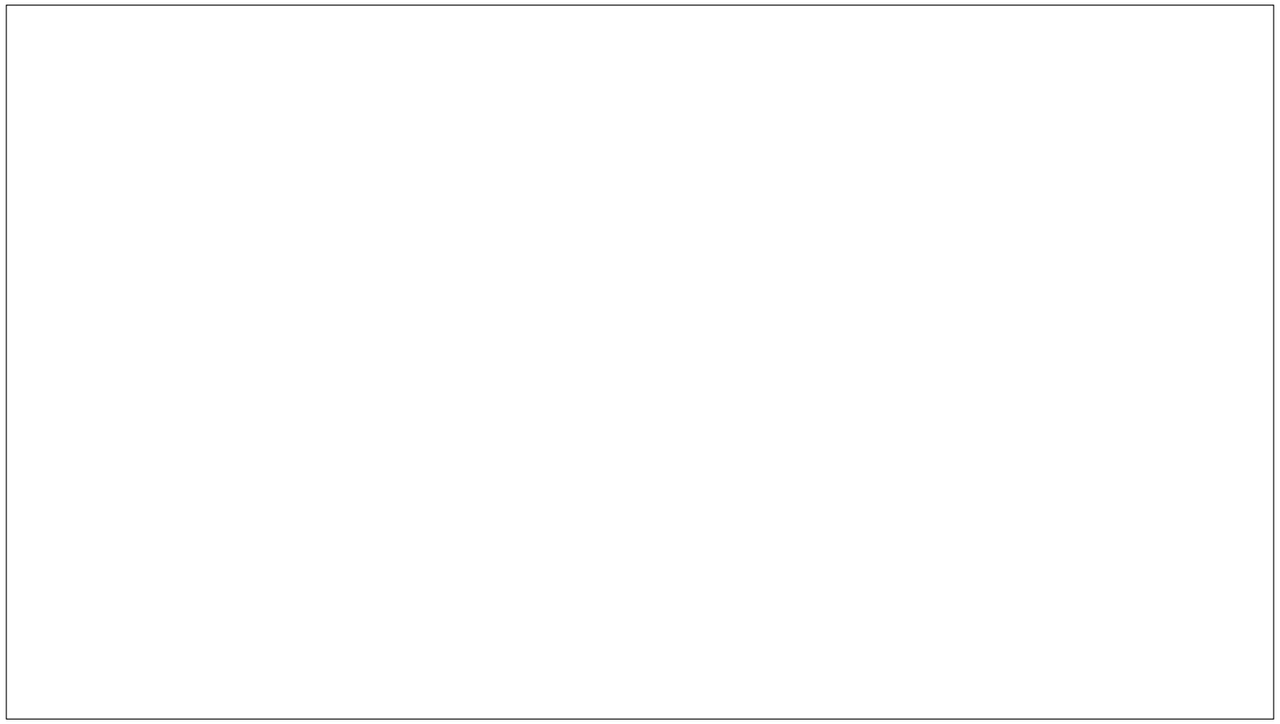


Has remained an industry standard for more than thirty years!



Now, let’s imagine it’s 2019, and we’re some
[build] big tech company which has millions and millions of dollars of incentive to run machine learning as fast and as cheaply as possible.





To achieve this, we'll build
[build] custom chips fully dedicated to accelerating machine learning algorithms.

When it comes time to choose how we will represent real numbers in hardware, should we necessarily choose IEEE 754 floats?

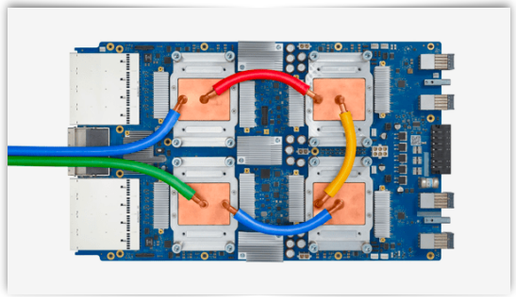
Well, there are a number of reasons we might not!

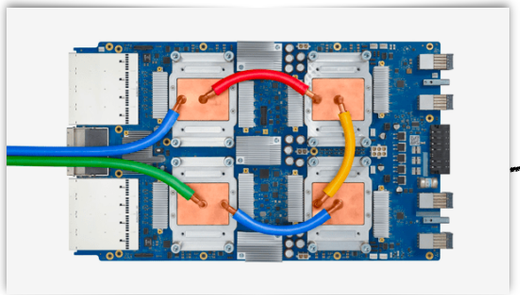
[build] We're going to want our datatype to be as fast and power-efficient as possible. IEEE floats were not designed for speed or power efficiency.

[build] We also want our datatypes to be as small as possible, to maximize the number of weights and activations you can store and operate over on-chip. Though the standard defines 16 bit floats, we may want to push even smaller.

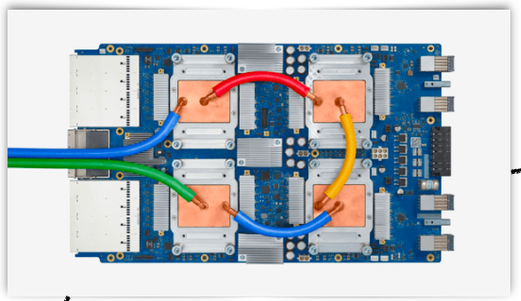
[build] Finally, and perhaps most interestingly, in machine learning, most weights and activations will lie within a very specific range of values, such as [-1, 1]. Floats do not take advantage of this fact, but we can build a datatype which does.

All of these are great reasons to explore alternative datatypes. So with that, let's jump right into…[slide transition]





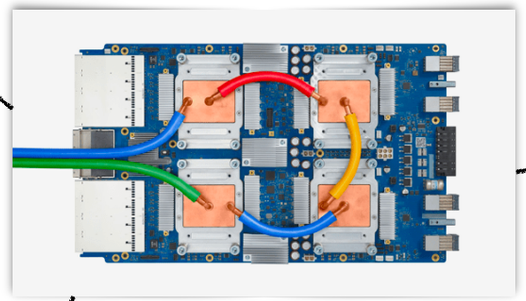
Should be fast and power-efficient



Should be fast and power-efficient

Needs small weights and activations
to maximize usage of chip area

Only needs to represent a specific range of values:
Weights and activations cluster (e.g. around [-1, 1])



Should be fast and power-efficient

Needs small weights and activations
to maximize usage of chip area

The Datatypes Zoo

…the datatypes zoo.

What do new datatypes look like?

Before we begin our journey through the datatypes zoo, I want to give you a general flavor for the ways in which we can create new datatypes.

If our starting point is this IEEE single precision float, there are many ways to branch off from here to create new datatypes.

[build] Some datatypes keep the same fields as the IEEE float, but simply change the sizes of the fields to explore different ranges and precisions of numbers.

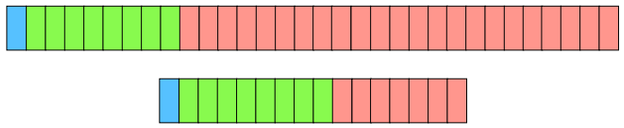
[build] Some datatypes will use entirely new fields with different sizes and meanings,
[build] or even use fields whose sizes can change!

[build] and other datatypes will look entirely unfamiliar, combining a host of other tricks!

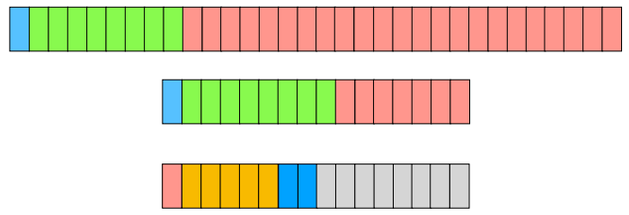
The design space of numerical datatypes is wide open, and the search for new and interesting types is just beginning. Now, I'll show you some example points within this design space.

So with that, let's jump right into the datatypes.

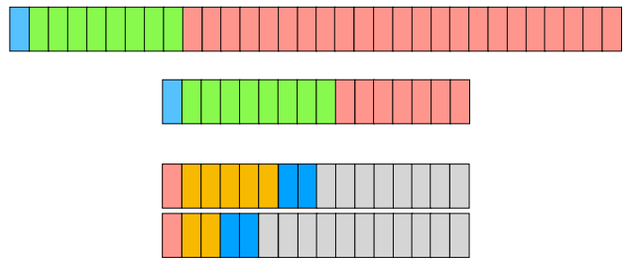
What do new datatypes look like?



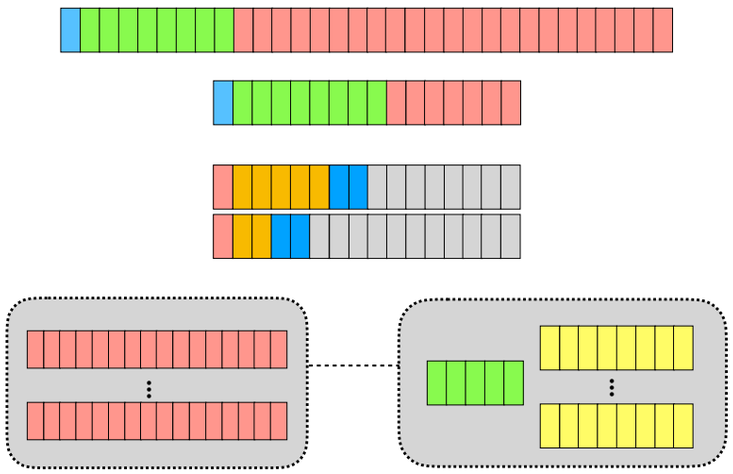
What do new datatypes look like?



What do new datatypes look like?



What do new datatypes look like?



bfloat16

See https://cloud.google.com/tpu/docs/bfloat16

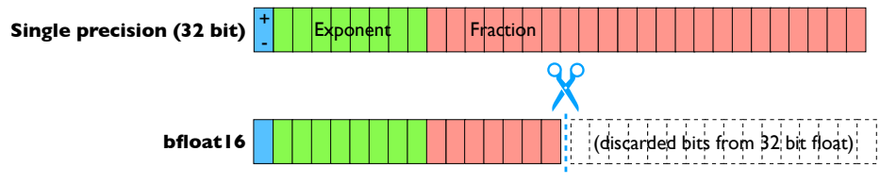
As I stated before, a core objective when building a machine learning accelerator is to maximize the amount of weights and activations you can have on-chip.
An easy way to do this is to shrink your datatype, which shrinks the memories needed to store your data, and also shrinks the hardware needed to perform computations on your data.
When Google began building their machine learning accelerator, the Tensor Processing Unit (which is the chip I showed on a previous slide), they took the simplest possible approach to shrinking datatypes: they took the IEEE single-precision float that we know and love, and just
[build] chopped off the last 16 bits!

And surprisingly, this actually worked!
They called the result the “brain float”, or bfloat16 for short.

[build] Because the bfloat has the same exponent size as the single precision float, this means the two types have the same dynamic range.
What google discovered was that this dynamic range seemed to be far more important than the type’s precision.
In fact, training machine learning models with the less-precise bfloat often has no impact on training, and in some cases can actually make the final trained model more accurate!

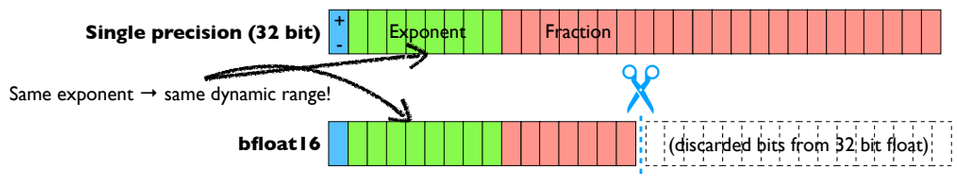
Bfloat is a great example of taking a simple approach to producing a new datatype. But what if we wanted to build a new datatype from the ground up?

bfloat16



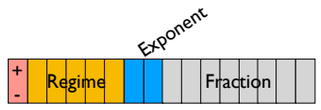
See https://cloud.google.com/tpu/docs/bfloat16

bfloat16



See https://cloud.google.com/tpu/docs/bfloat16

Posits



See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits are a great example of a datatype that has been built from the ground up.
Unlike IEEE floating point, which delivers roughly the same amount of accuracy to all numbers, posits are designed to deliver more accuracy to numbers closer to plus or minus one.
This is potentially great for machine learning, because, as we said, weights and activations often cluster in the range from -1 to 1.

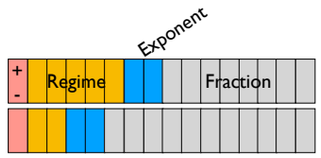
To achieve this variable amount of precision, the posit introduces the regime field.
The regime field is encoded in unary, which actually means it can
[build] shrink
[build] and grow in size.

When the regime is small, the value of the number is
[build] close to plus or minus 1. And because the regime is small, we have more bits left over for the fraction, meaning our number is more precise and can deliver more accuracy.
When the regime is large, on the other hand, the value of the number is
[build] closer to the extremes. In these cases, there can be little to no bits left for the fraction, leading to less precision and accuracy in general.

This behavior—delivering more accuracy near plus or minus 1, and less at the extremes—makes sense for machine learning, as values often cluster in the 1 to -1 range.

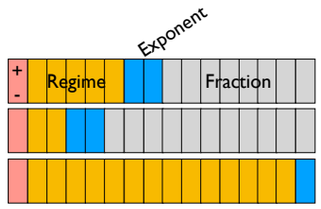
As a result, there has recently been a lot of buzz about using posits for machine learning.

Posits



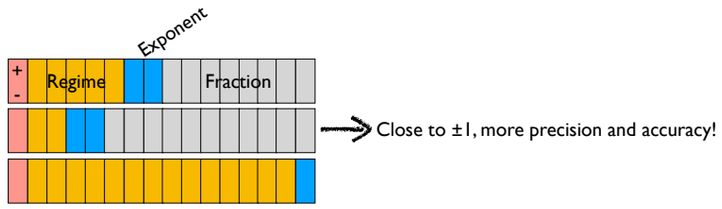
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



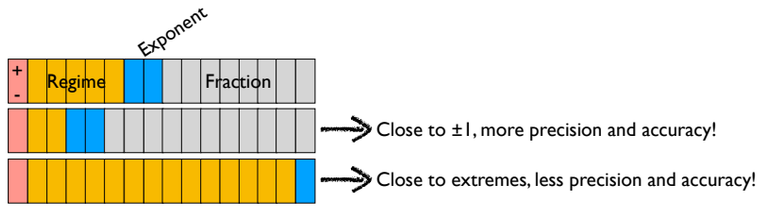
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



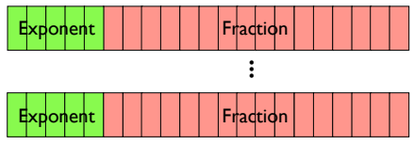
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Flexpoint



See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

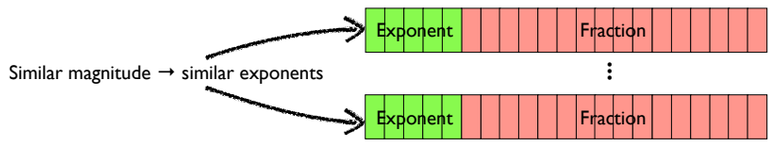
The last datatype I want to talk about is perhaps the most interesting, from a systems design perspective, and begins to show how far we can push datatype design.

As we've been talking about, values in machine learning often lie within specific ranges.

Unsurprisingly, we also see this pattern arise at the tensor level: values within tensors will often have similar magnitudes,
[build] meaning that their exponents will be similar.

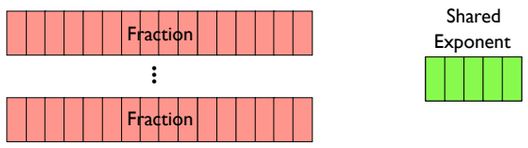
If this is the case, then we can often have…[slide transition]

Flexpoint



See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

Flexpoint



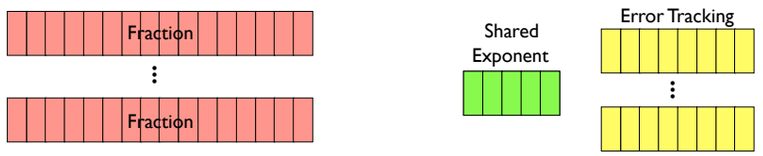
See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

…whole blocks of numbers share their exponents.
This is called "block floating point", and is a common space-saving technique in datatype design.

The larger our block, the more numbers we have sharing an exponent, and thus, the more potential space saving.
However, this sharing will also introduce error, as the exponent will not accurately represent the full range of values in the block.

[slide transition]

Flexpoint

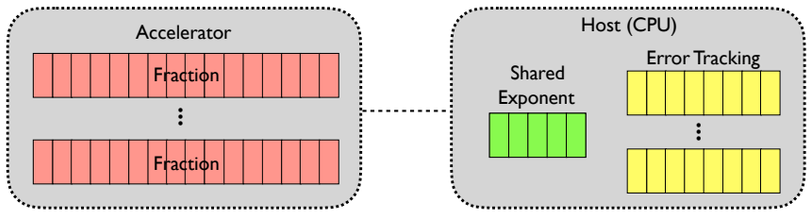


See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

To handle this, we can introduce data structures for tracking errors and making predictions on how to change the exponent so as to minimize error in the future.

Finally, to take this idea to its extreme point, we can [slide transition]

Flexpoint



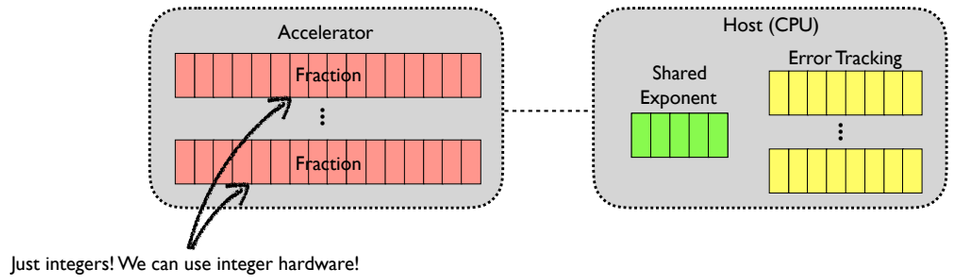
See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

…physically separate the fractional bits from the exponent and error tracking bits. The fractional bits are all that's really needed to do computation on the accelerator, and so we'll keep everything else on the host device, for example, the CPU that's interacting with the accelerator.

Furthermore, the fractional bits are
[build] essentially just integers, and so we can operate on them using integer hardware. This greatly simplifies the accelerator, as integer hardware is faster, power efficient, and smaller than floating point hardware.

This datatype I've just described is Intel's Flexpoint, a datatype native to their Nervana accelerator, and shows an interesting extreme in the datatype design space

Flexpoint



See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

In conclusion,

So, hopefully now you see that

[build] The basic problem of representing real numbers in hardware is an ongoing challenge, and

[build] there are many interesting solutions out there, beyond IEEE floats!

In conclusion,

- Representing real numbers in hardware is an ongoing challenge

In conclusion,

- Representing real numbers in hardware is an ongoing challenge
- There are many interesting solutions out there, beyond IEEE floats!

Thank you!