Bring Your Own Datatypes: Enabling Custom Datatype Exploration in Deep Learning

Gus Smith, Qualifying Exam
February 24th, 2020

sampl

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

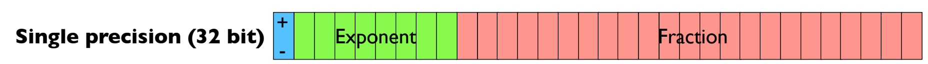
tvm.ai

Hi everybody.
Today I'll be talking about my work on enabling custom datatype exploration in deep learning through the Bring Your Own Datatypes framework.

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

By datatypes, 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.

IEEE 754 Floating Point



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

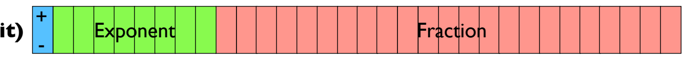
For more than thirty years, we've had a solution to this problem: the IEEE 754 standard for floating point arithmetic.

IEEE 754 was designed to be flexible for many applications, and also specifies floats of many sizes.
[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!

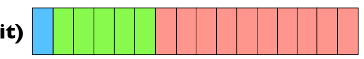
IEEE 754 Floating Point

Single precision (32 bit)



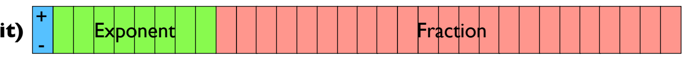
$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

Half precision (16 bit)



IEEE 754 Floating Point

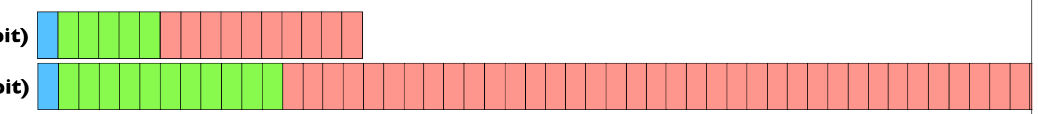
Single precision (32 bit)



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

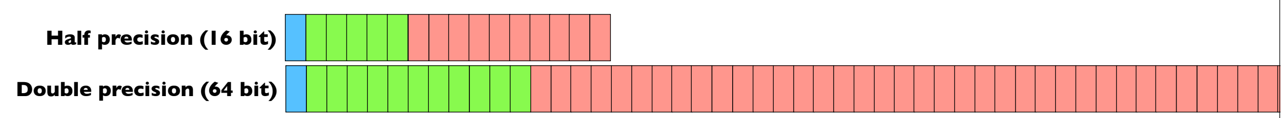
Half precision (16 bit)

Double precision (64 bit)



IEEE 754 Floating Point

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



Has remained an industry standard for more than thirty years!

Now, let’s imagine it’s 2020, and you’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, you'll build
[build] custom chips fully dedicated to accelerating machine learning algorithms, such as Google's TPU, pictured here.

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

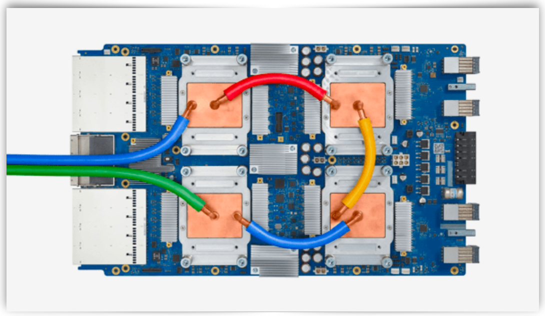
Well, there are a number of reasons you might not!

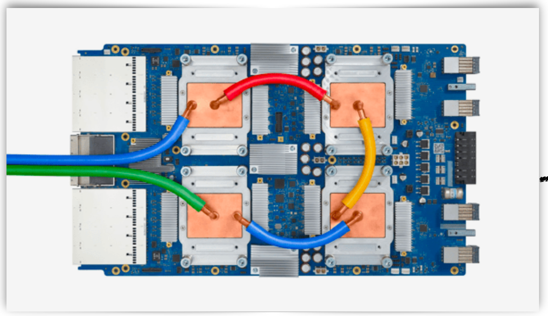
[build] You're going to want your datatype to be as fast and power-efficient as possible. IEEE floats were not designed for speed or power efficiency.

[build] You'll also want your 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, you may want to push even smaller.

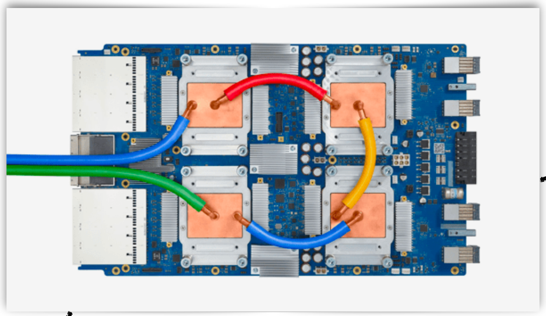
[build] Finally, and perhaps most interestingly, in machine learning, most weights and activations will lie within a very specific range of values, such as [-1, 1]. Floats do not take advantage of this fact, but you can utilize a datatype which does.

All of these are great reasons to explore alternative datatypes.





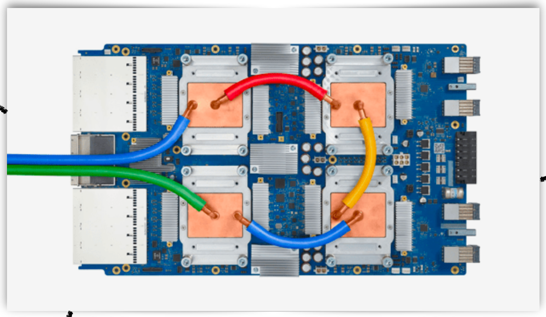
Should be fast and power-efficient



Should be fast and power-efficient

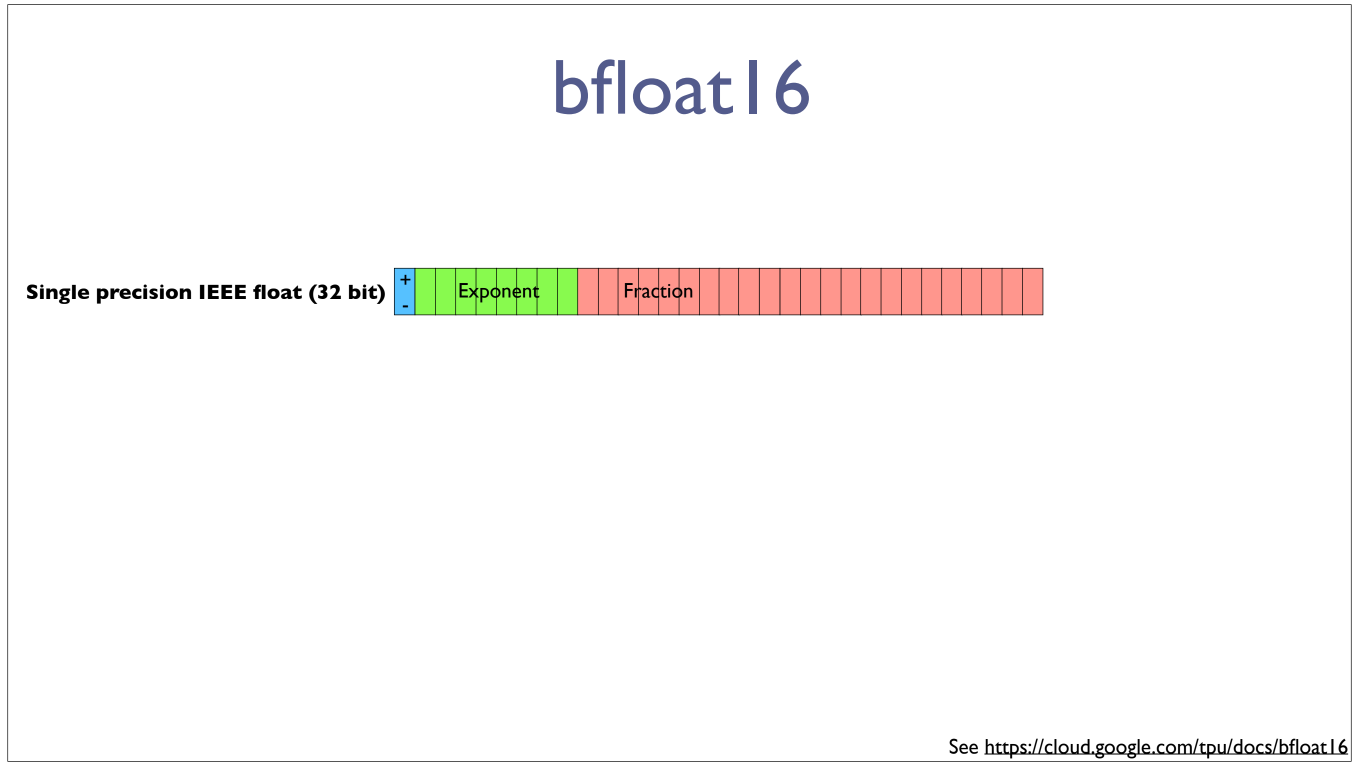
Needs small weights and activations
to maximize usage of chip area

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



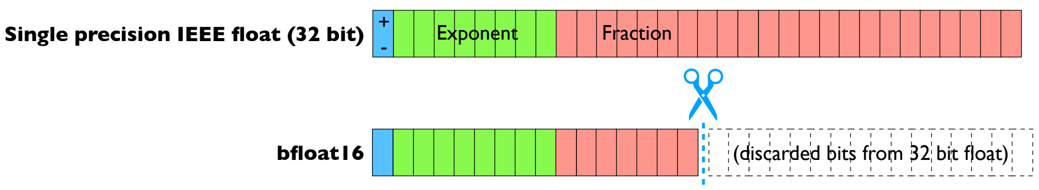
One example of a datatype that exploits the domain-specific properties of deep learning is the bfloat.

Bfloat is simply an IEEE single precision float —
[build] if you chopped off half of the bits!

As a result, bfloat has much less precision than a float, but the same dynamic range—which turns out to be more important for many workloads.

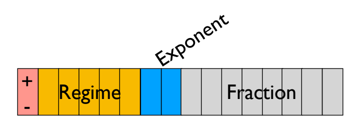
Google discovered this while building their TPU, and now the bfloat is used natively on the TPU and many other accelerators.

bfloat16



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

Posits



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

Another example datatype is the posit.

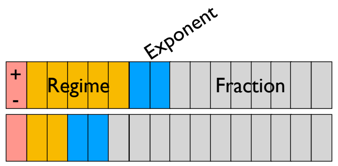
Posits introduce a new field—the regime—which can
[build] vary
[build] in size.

[build] When the regime is small, the number is closer to plus or minus 1; because it has more bits left over for the fraction, it can be much more precise.

[build] On the other hand, when the regime is large, the number is closer to the extremes, where it will have much less precision.

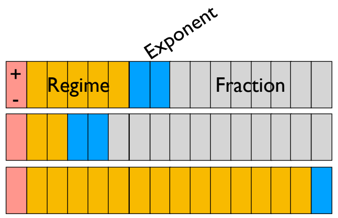
This gives the posit *tapered precision,* meaning that it delivers more precision closer to plus and minus one.
This, as we noted, is good for machine learning workloads, whose values often cluster in the range [-1,1]!

Posits



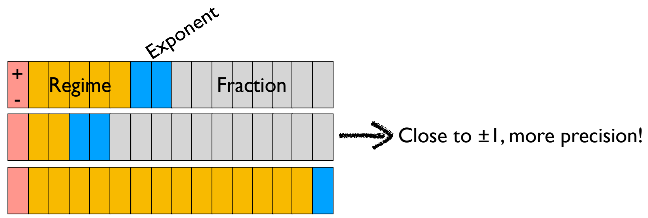
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



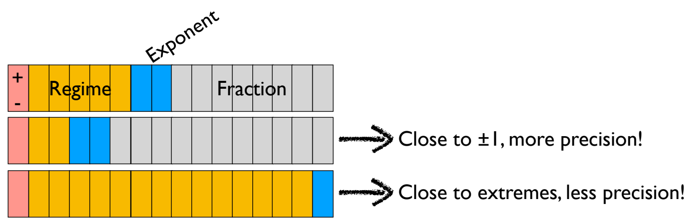
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



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

Posits



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

Seeing the popularity of deep learning accelerators nowadays, it’s clear that
[build] deep learning needs hardware specialization.

But, seeing how IEEE 754’s shortcomings can be fixed with specialized datatypes, it’s also clear that
[build] hardware specialization needs new datatypes!

Deep learning needs hardware specialization…

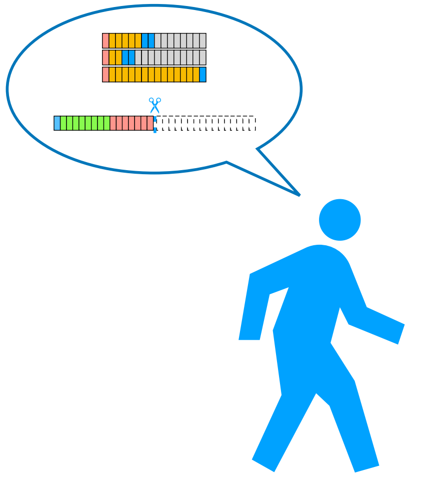
Deep learning needs hardware specialization…
…and hardware specialization needs new datatypes!

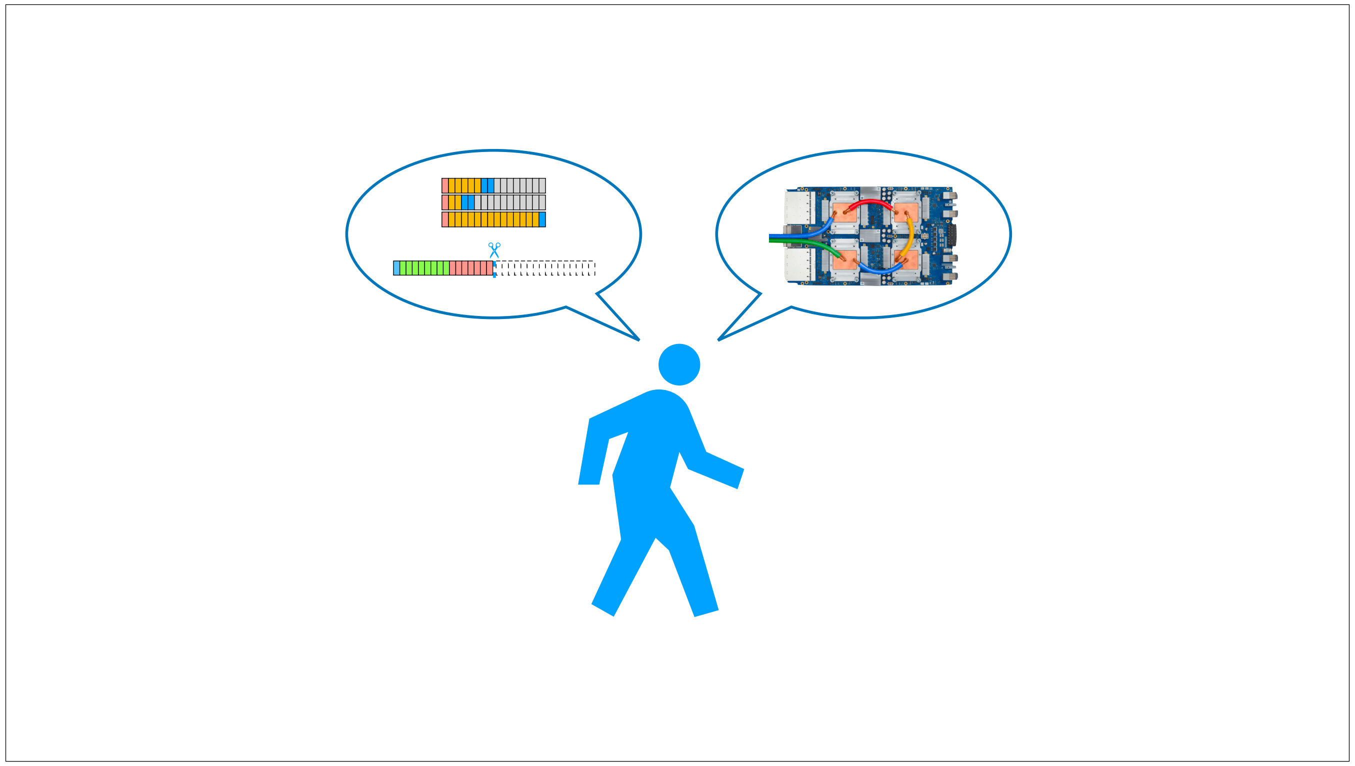
So now, let's put ourselves into the shoes of a
[build] researcher who actually wants to experiment with some of these new datatypes.

We might be a
[build] datatypes researcher, interested in testing out some types we've made.

Or we might be a
[build] hardware designer, looking for a new datatype for our next accelerator.

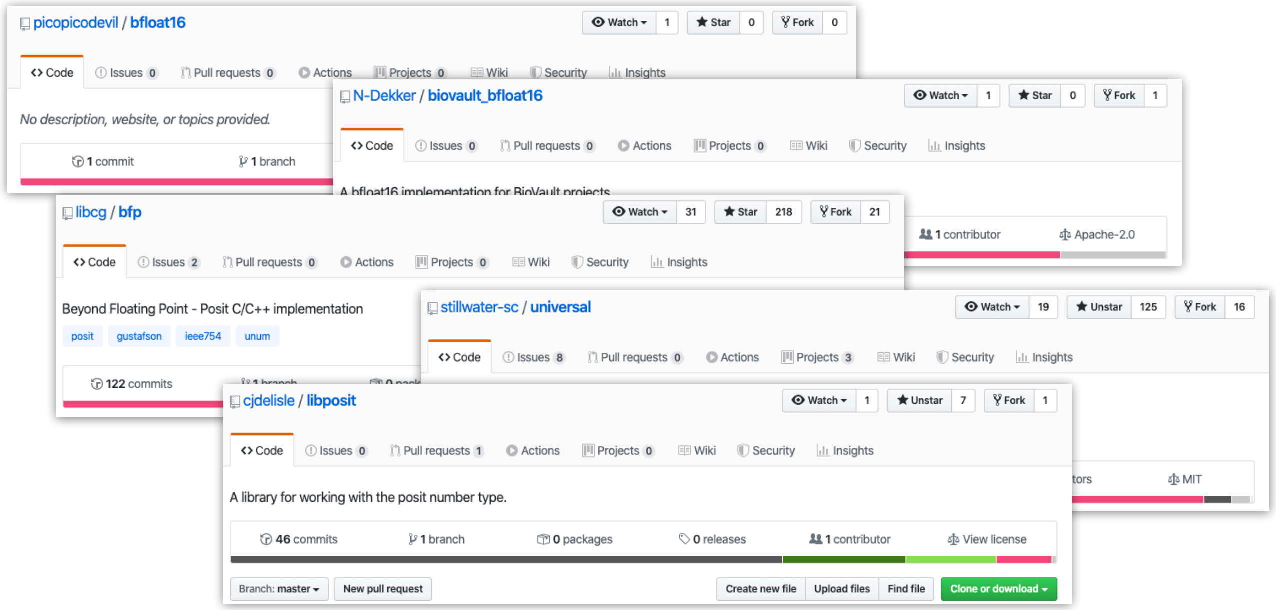
And let's assume we have a datatype we want to build, or we found a datatype we want to play with—where do we begin?





Well, before trying to design any actual hardware for a datatype, it’s the norm to first implement your datatype as a software library.

[build] Here are just a few examples from GitHub, of various datatype libraries.


picopicodevil / bfloat16
No description, website, or topics provided.
N-Dekker / biovault_bfloat16
A bfloat16 implementation for BioVault projects
libcg / bfp
Beyond Floating Point - Posit C/C++ implementation
posit
gustafson
ieee754
unum
stillwater-sc / universal
cjdelisle / libposit
A library for working with the posit number type.
46 commits
1 branch
0 packages
0 releases
1 contributor
View license

https://github.com/cjdelisle/libposit

These libraries look like you might expect:
They have
[build] types, representing each datatype, and
[build] operations over those types.

They might have
[build] interesting functions specific to the datatype, such as, in the posit's case, these functions for computing error-free adds and subtracts.
However, all of these libraries will always have a base set of functions needed to implement worthwhile numeric workloads, such as
[build] add,
[build] subtract,
[build] multiply, and
[build] divide.

Types

libposit defines the following types:

- `posit8_t` 8 bit posit
  - `posit8x2_t` pair of 8 bit posits, result of `*_exact` functions.
- `posit16_t` 16 bit posit
  - `posit16x2_t` pair of 16 bit posits, result of `*_exact` functions.
- `posit32_t` 32 bit posit
  - `posit32x2_t` pair of 32 bit posits, result of `*_exact` functions.
- `posit64_t` 64 bit posit
  - `posit64x2_t` pair of 64 bit posits, result of `*_exact` functions.
- `posit128_t` 128 bit posit, result of posit64 `*_promote` functions. Only functions which work on a `posit128_t` are `posit128_iexp()` and `posit128_fract()`.

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- posit8_t 8
  - posit8x
- posit16_t 1
  - posit16
- posit32_t 3
  - posit32
- posit64_t 6
  - posit64
- posit128_t
  posit128_ie

Operations

For each of these types, libposit provides the following functions:

- posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y) : Add two posits together, output a rounded result.
- posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y) : Subtract one posit from another, output a rounded result.
- posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y) : Output 2 results, sum and remainder which when added will result in the exact same value as the inputs x and y but where the absolute value of remainder is as small as possible.
- posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y) : Exactly the same as posit<X>_add_exact(x, y) but with the sign of y flipped.
- posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y) : Multiply two posits, round the result to nearest
- posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y) : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)`: Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)`: Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)`: Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)`: Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)`: Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)`: Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

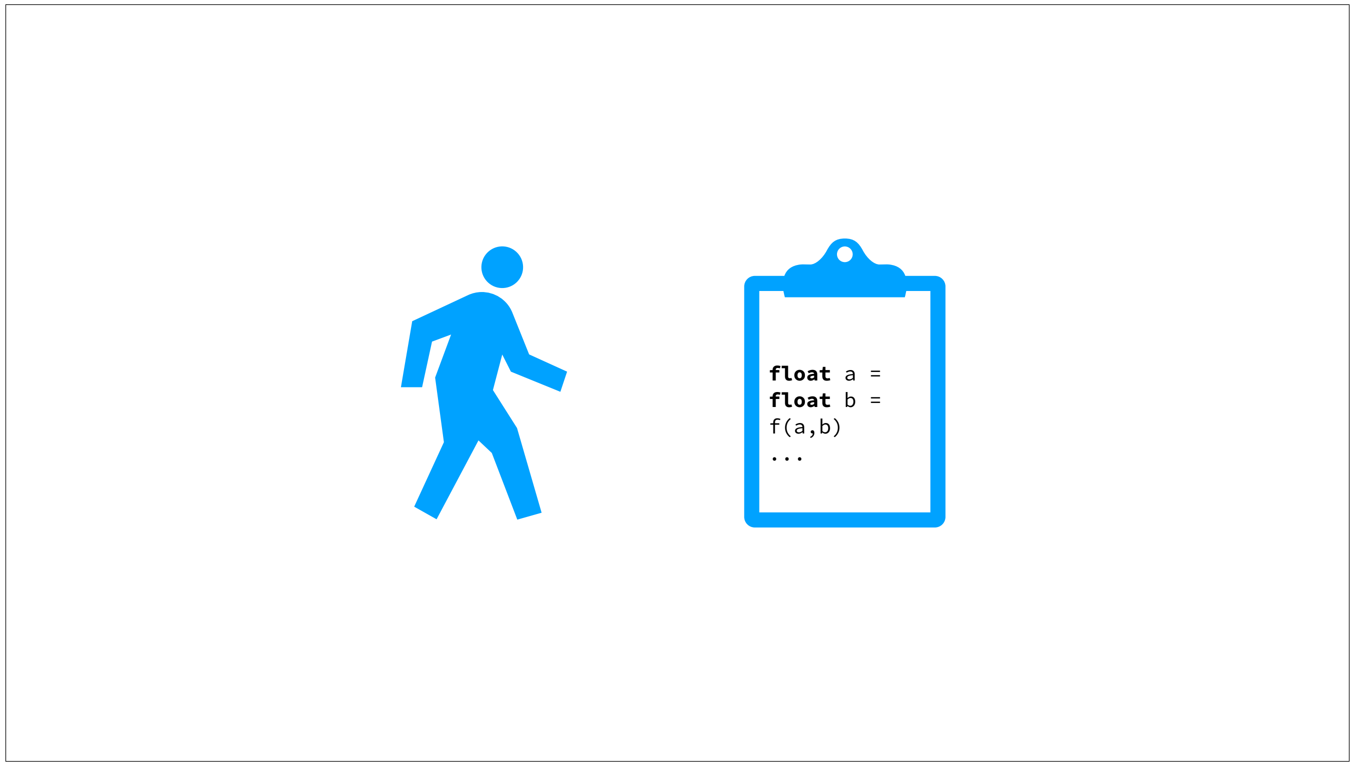
- posit8_t 8
  - posit8x
- posit16_t 1
  - posit16
- posit32_t 3
  - posit32
- posit64_t 6
  - posit64
- posit128_t
  posit128_ie

Operations

For each of these types, libposit provides the following functions:

- posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y): Add two posits together, output a rounded result.
- posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y): Subtract one posit from another, output a rounded result.
- posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y): Output 2 results, sum and remainder which when added will result in the exact same value as the inputs x and y but where the absolute value of remainder is as small as possible.
- posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y): Exactly the same as posit<X>_add_exact(x, y) but with the sign of y flipped.
- posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y): Multiply two posits, round the result to nearest
- posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y): Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit



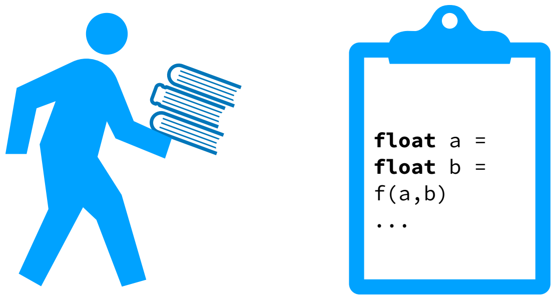
So we have our datatype in the form of a
[build] library, which we might have built, or might have found online.

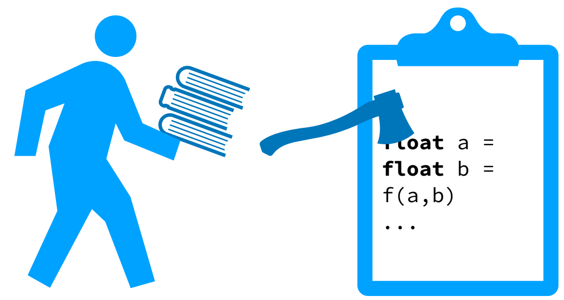
But we want to do something useful with it: we want to run a workload, such as the one depicted here.
Most workloads we want to run will not be using our type—they will likely be using IEEE floats!
So we need to figure out how to get our datatype into the workload.

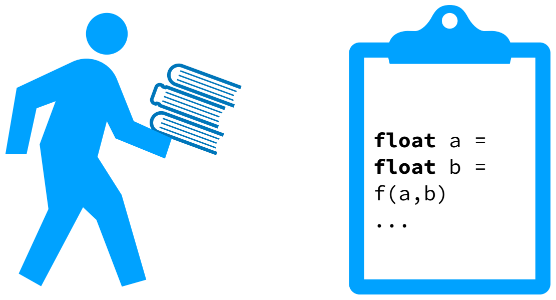
There are two ways about this.

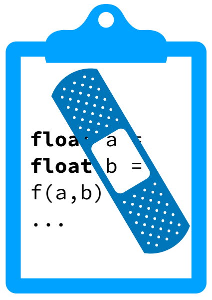
Method 1 is to
[build] hack
[build] the
[build] datatype into the workload; I.e., explicitly replace every float with your datatype, and replace every operation with a call to your library.

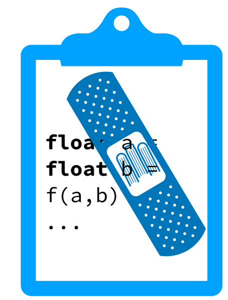
This is tedious and not very repeatable, as it needs to be done for each workload.

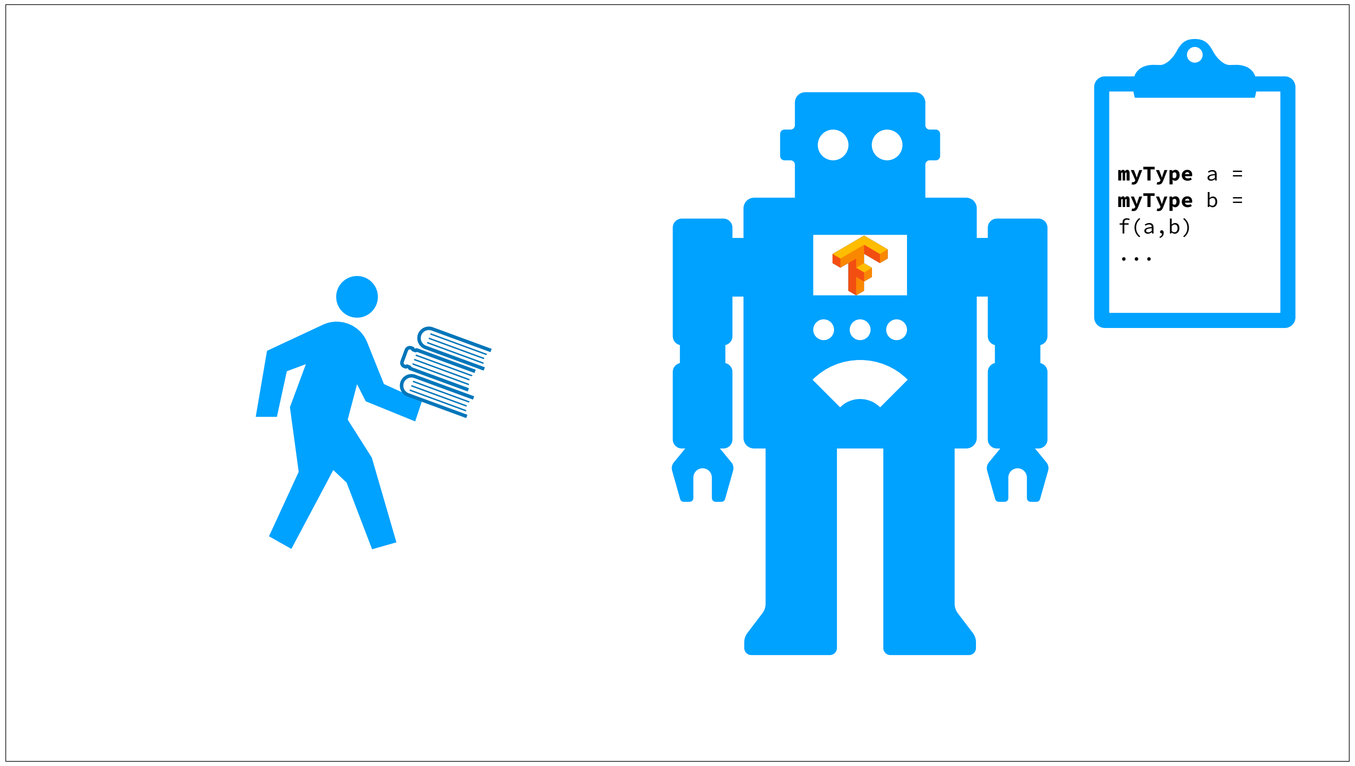

float a =
float b =
f(a,b)
...


loat a =
float b =
f(a,b)
...


float a =
float b =
f(a,b)
...


float b =
f(a,b)
...

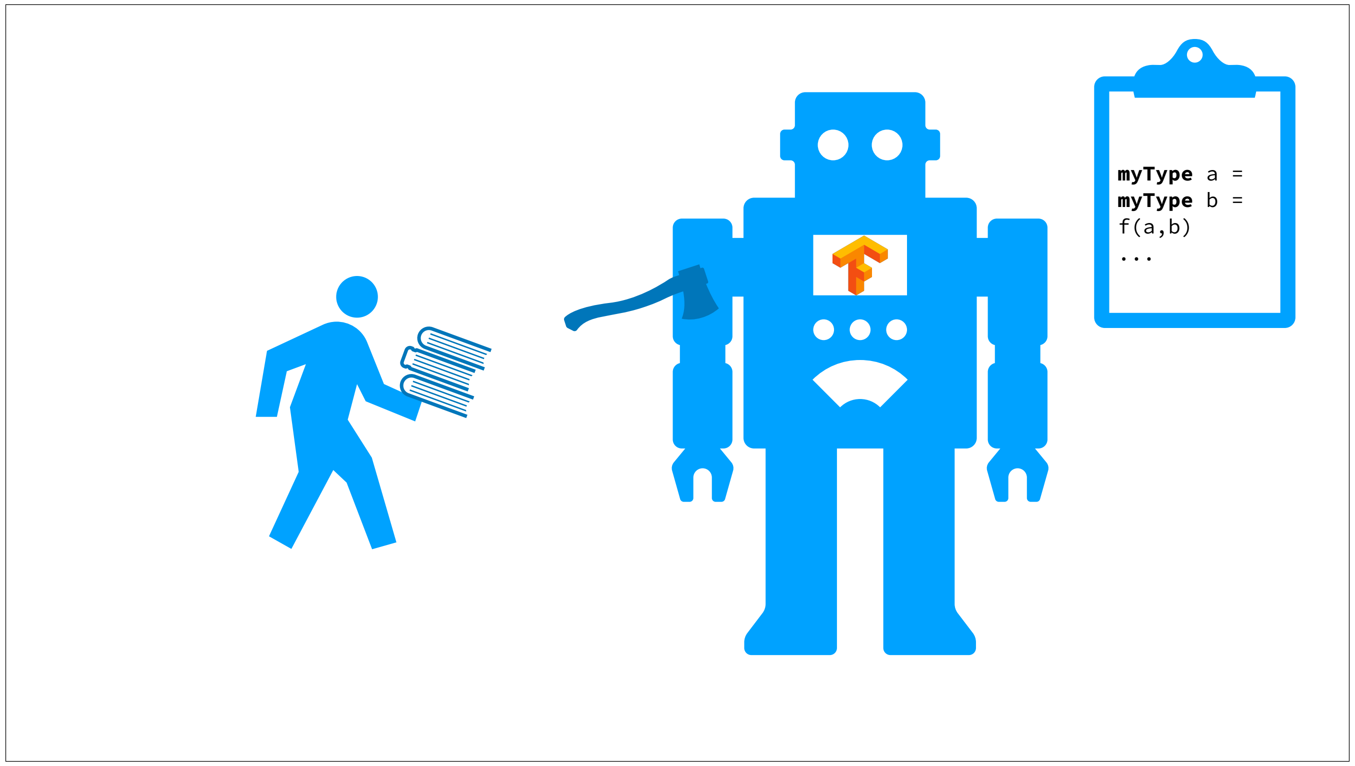

float b
f(a,b)
...

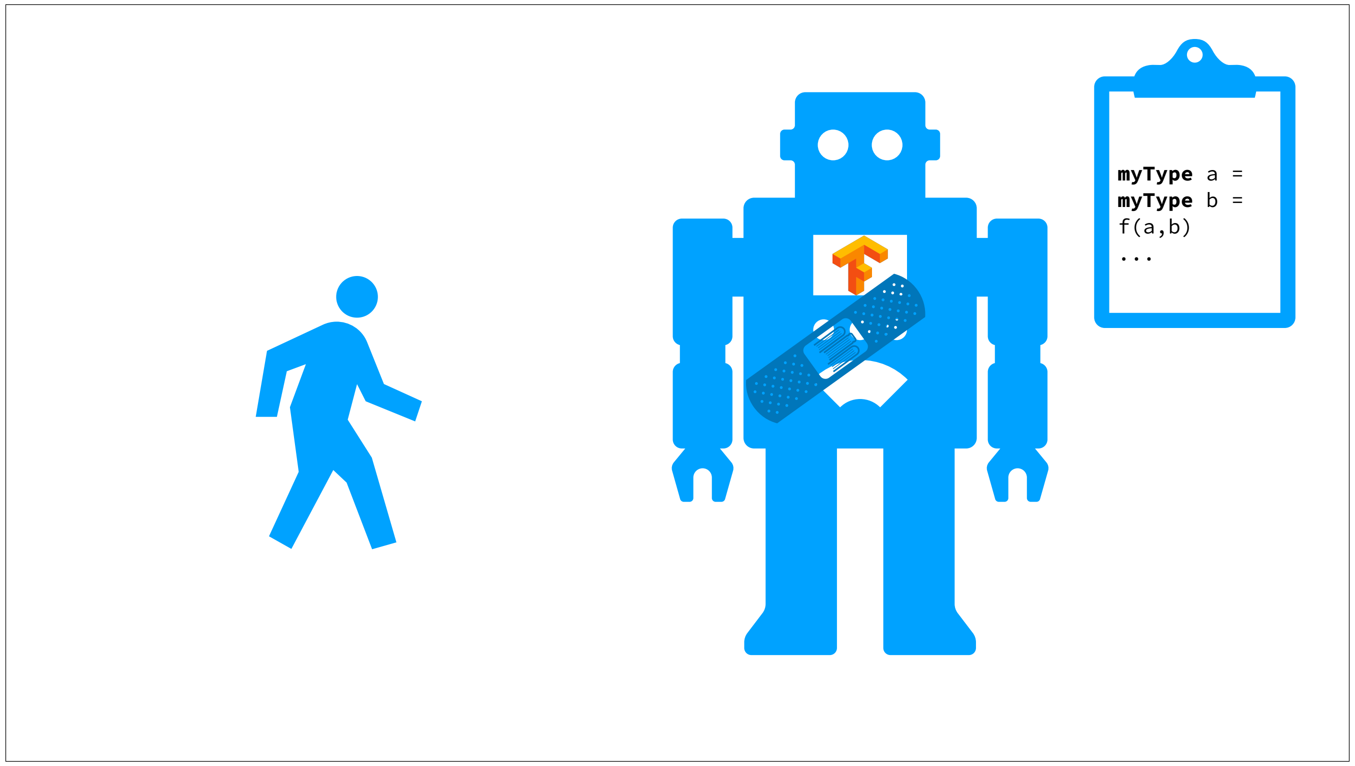


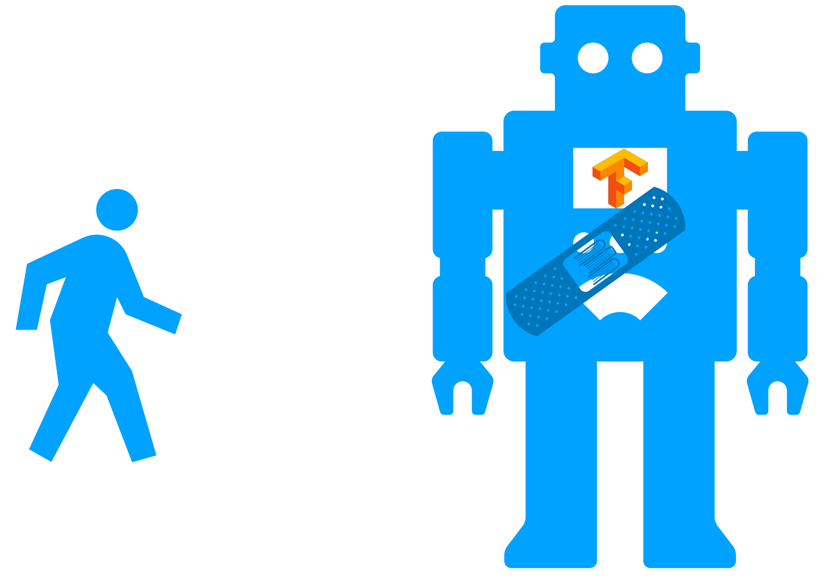
A more sustainable solution is to modify an existing framework or compiler to support your datatype.

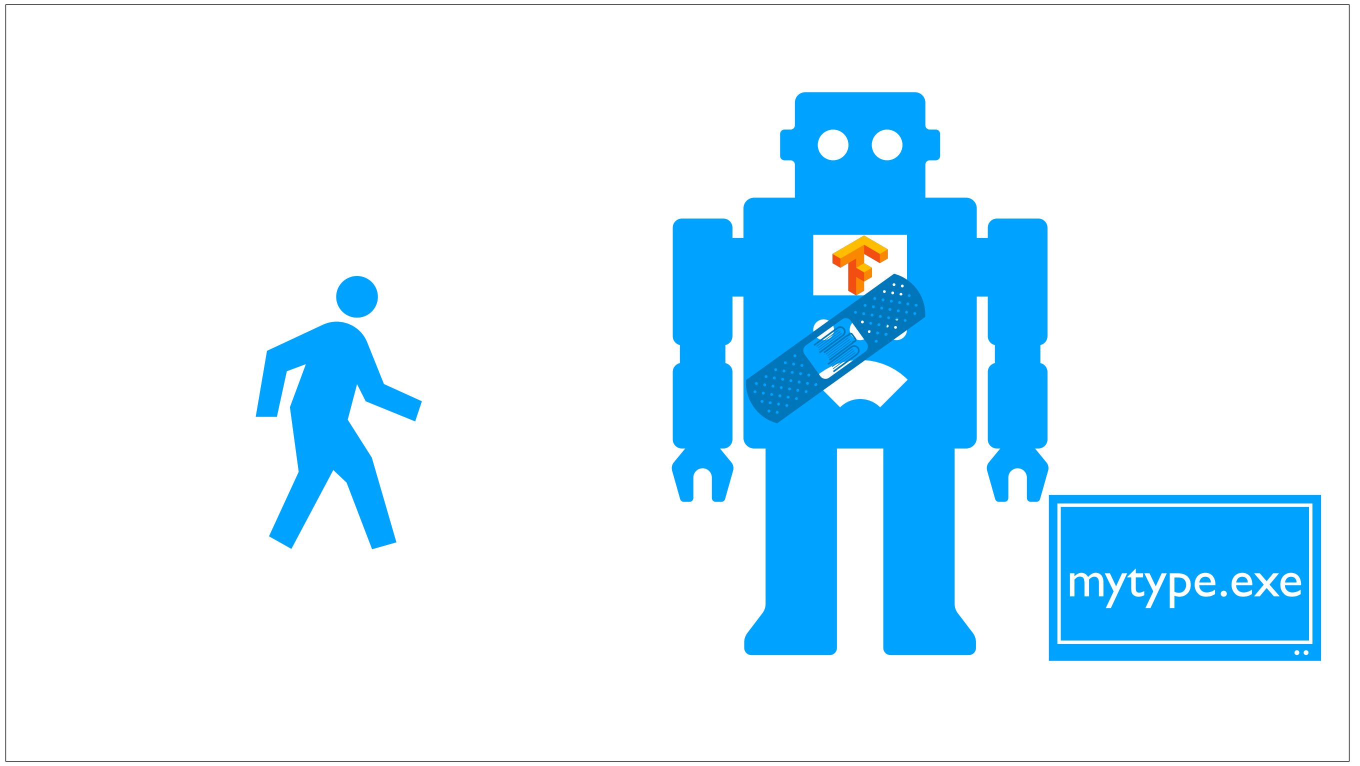
Make sure to cast this as the "intelligent" choice…but then, turn it around and be like, "…but it's still hard!"

However, these frameworks are large and unwieldy, and modifying them…


myType a =
myType b =
f(a,b)
...


myType a =
myType b =
f(a,b)
...



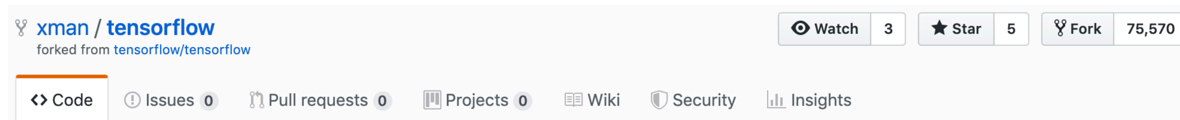

mytype.exe

An Example in the Wild!

Let's take a look at an example in the wild, which will show us just how much work it can be to hack a datatype into existing code.
[build] This is an HPC researcher's fork of tensorflow. To experiment with posits, this researcher built posits directly into tensor flow.
[build] The result was 237 commits worth of work—
[build] nearly 6000 lines of code added to tensor flow,
[build] touching a huge number of files across the codebase.

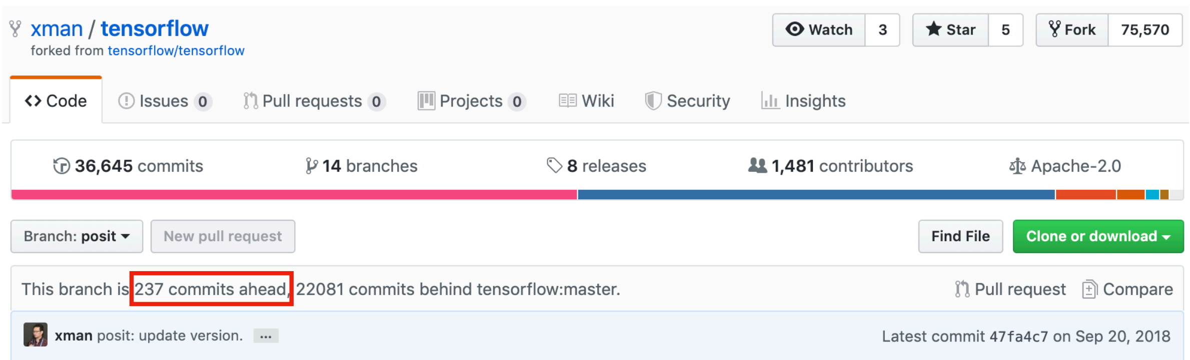
This amount of work is prohibitive for many researchers! Neither datatype researchers nor users interested in employing new datatypes will want to spend the time to hack their datatype into tensorflow. This high bar will put a damper on useful datatypes research.

An Example in the Wild!



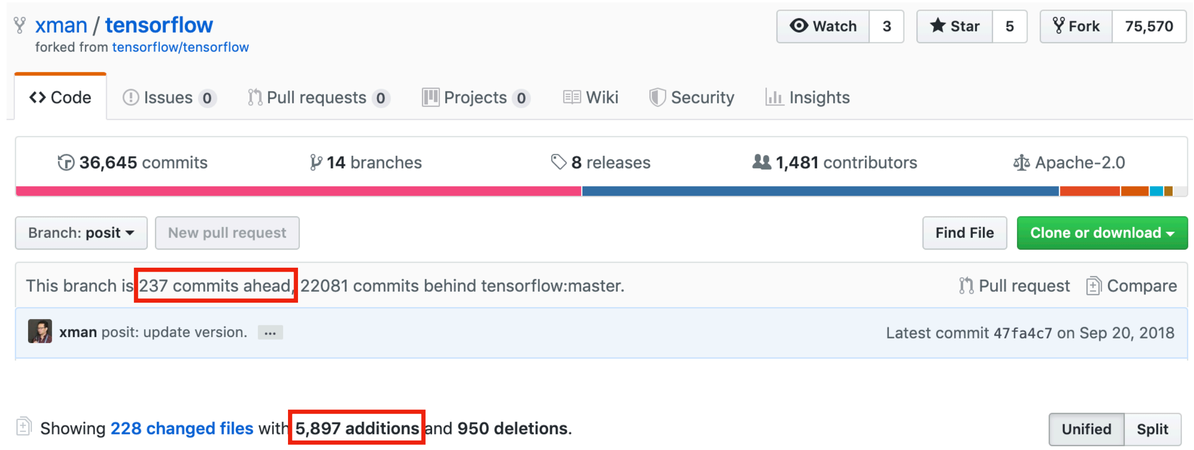
https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



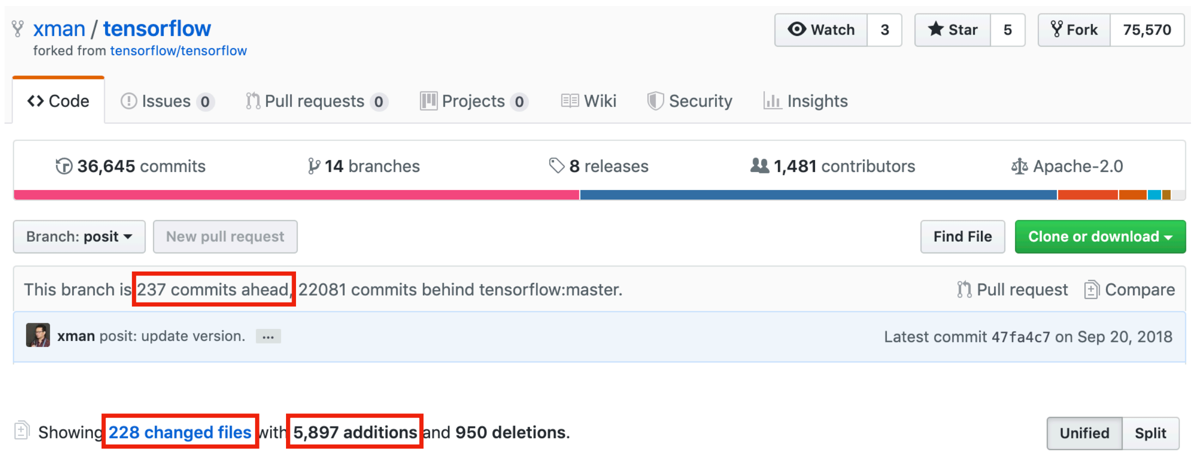
https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



https://github.com/xman/tensorflow/tree/posit

The computational demands of deep learning require new datatypes…

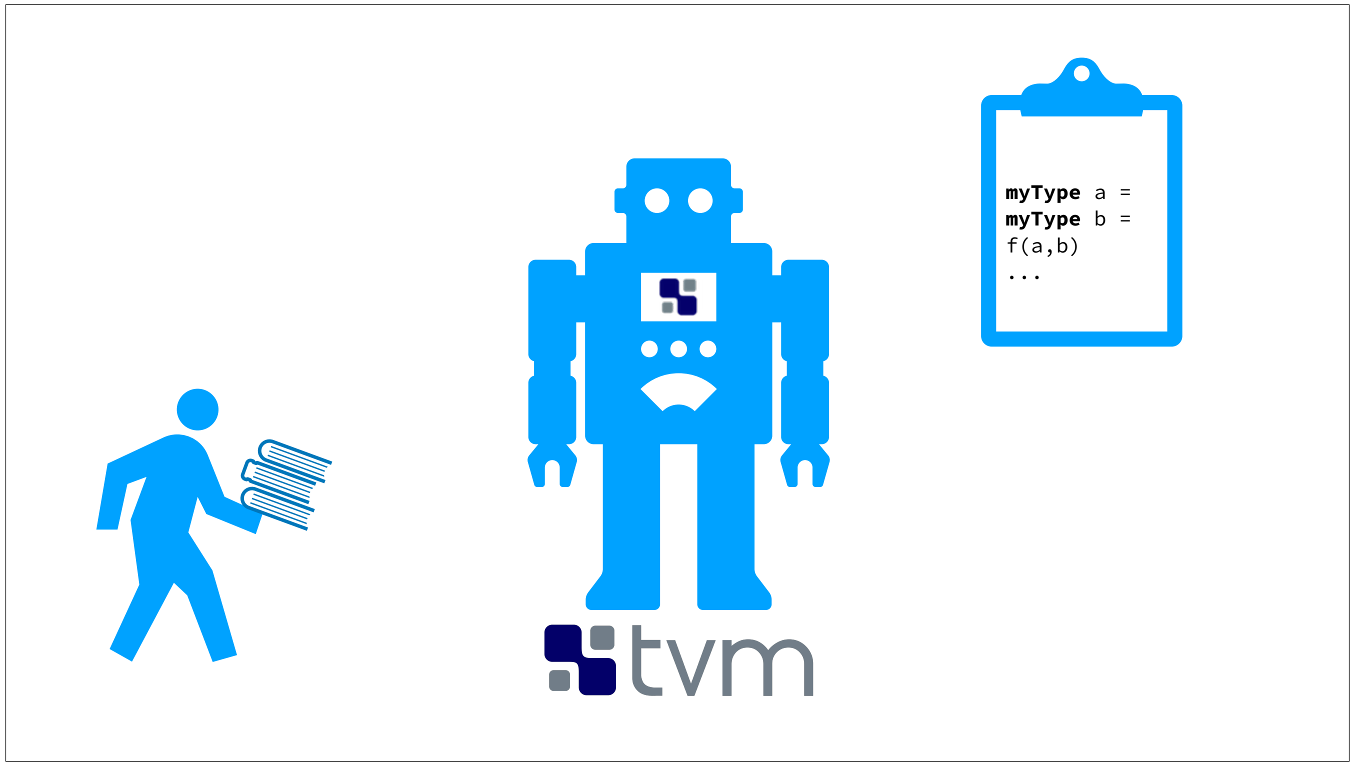
The computational demands of deep learning require new datatypes…

…but datatype research is difficult!

The computational demands of deep learning require new datatypes…

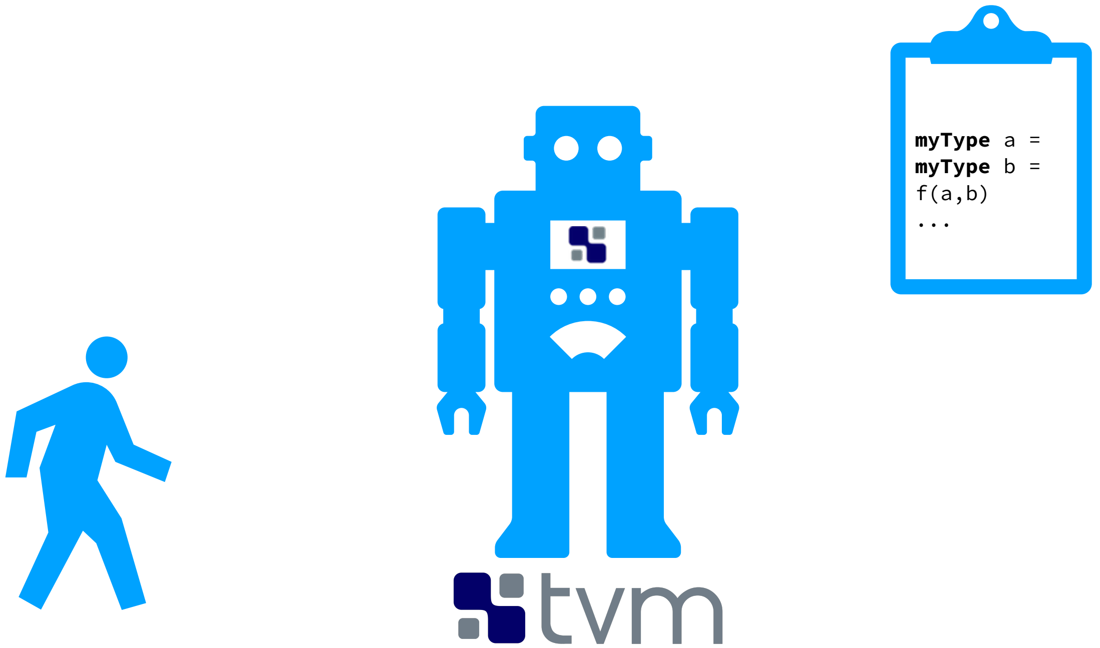
…but datatype research is difficult!

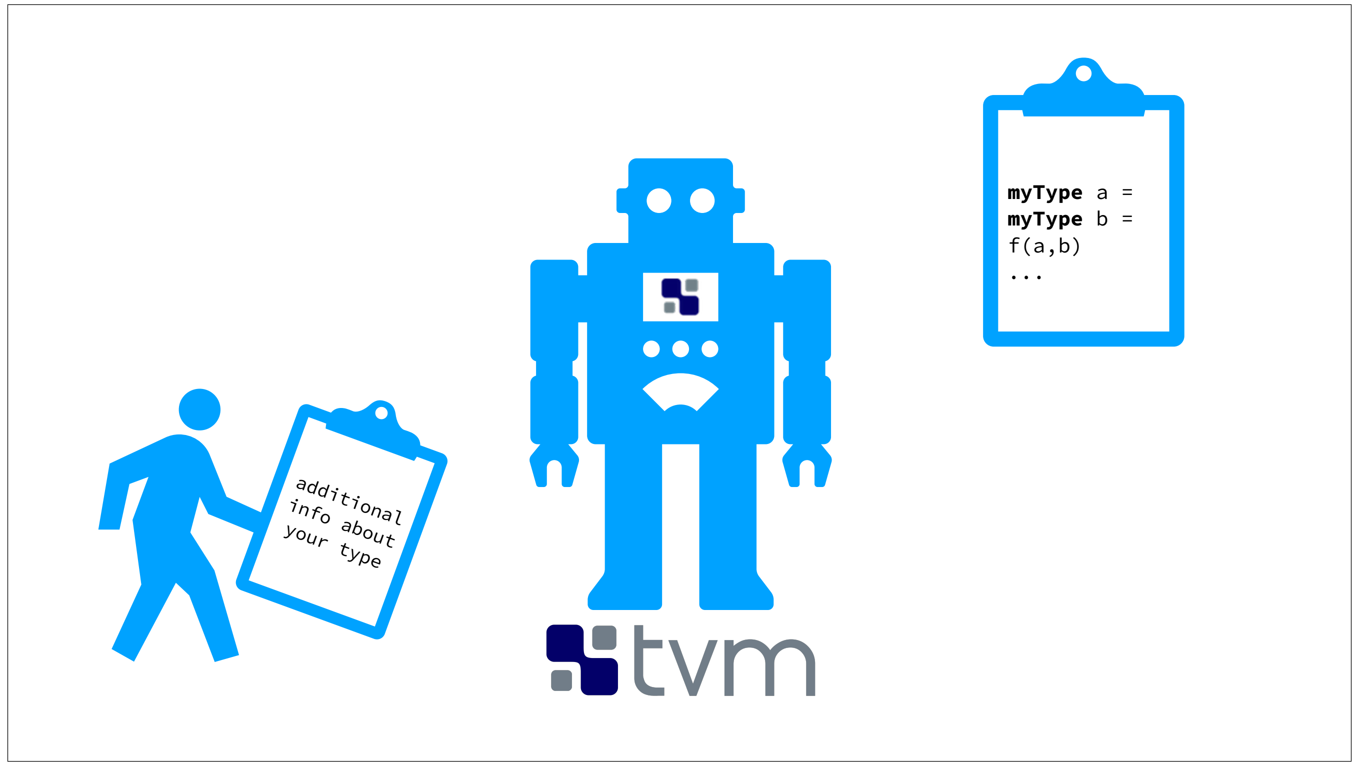
Our solution: the Bring Your Own Datatypes framework.

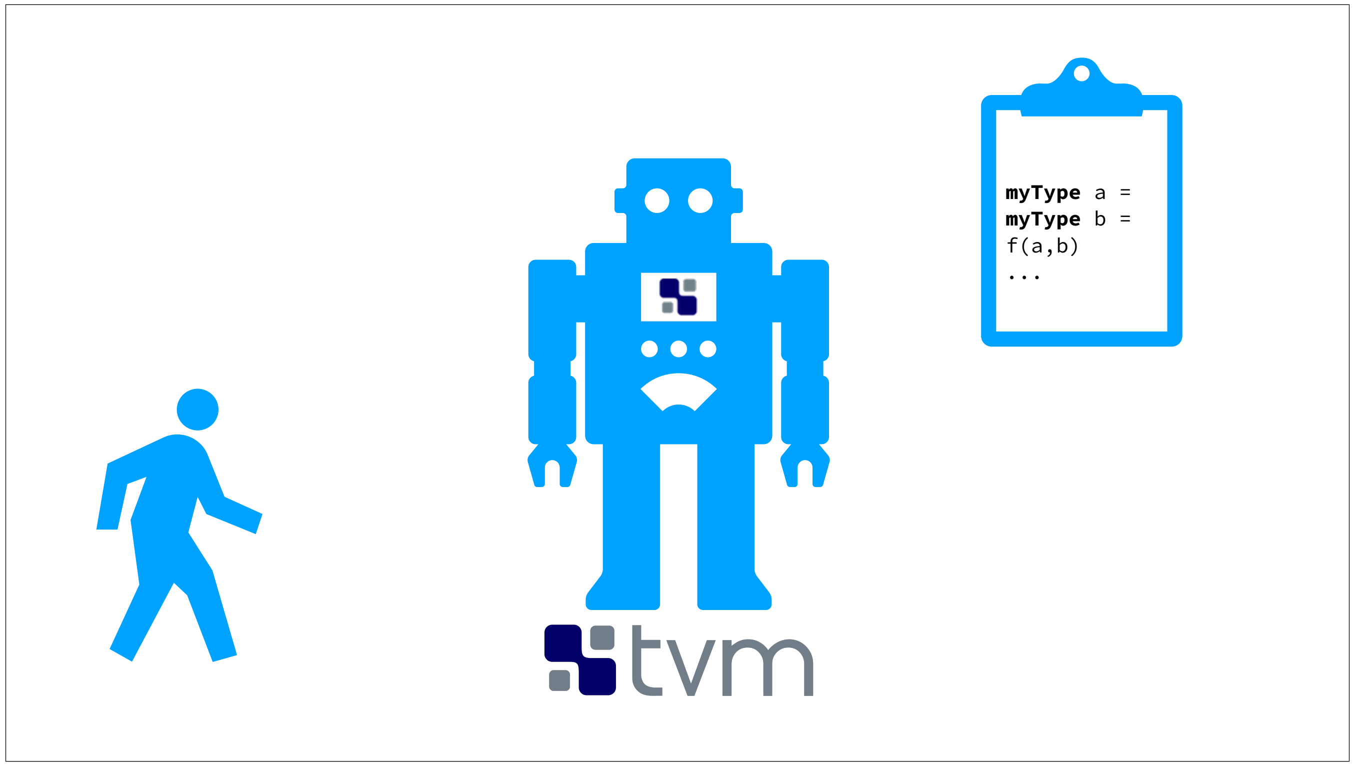


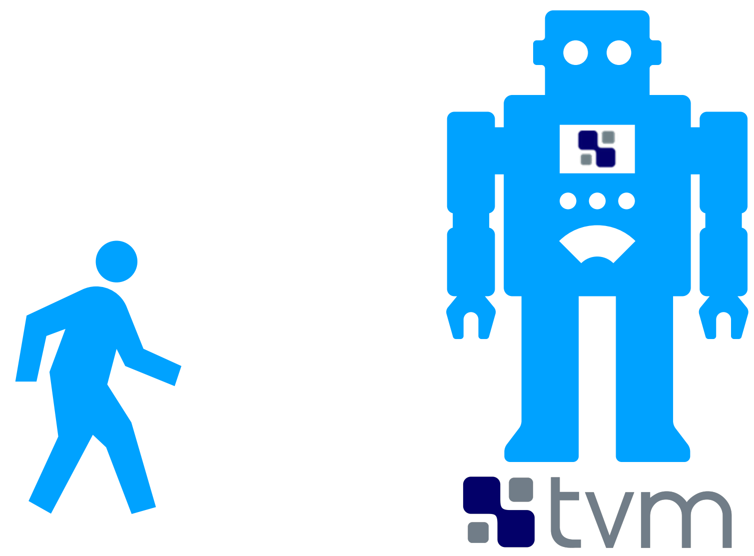
Using TVM, the Bring Your Own Datatypes framework enables a similar workflow to what we saw in the Tensorflow case, but without any need for compiler hacking.

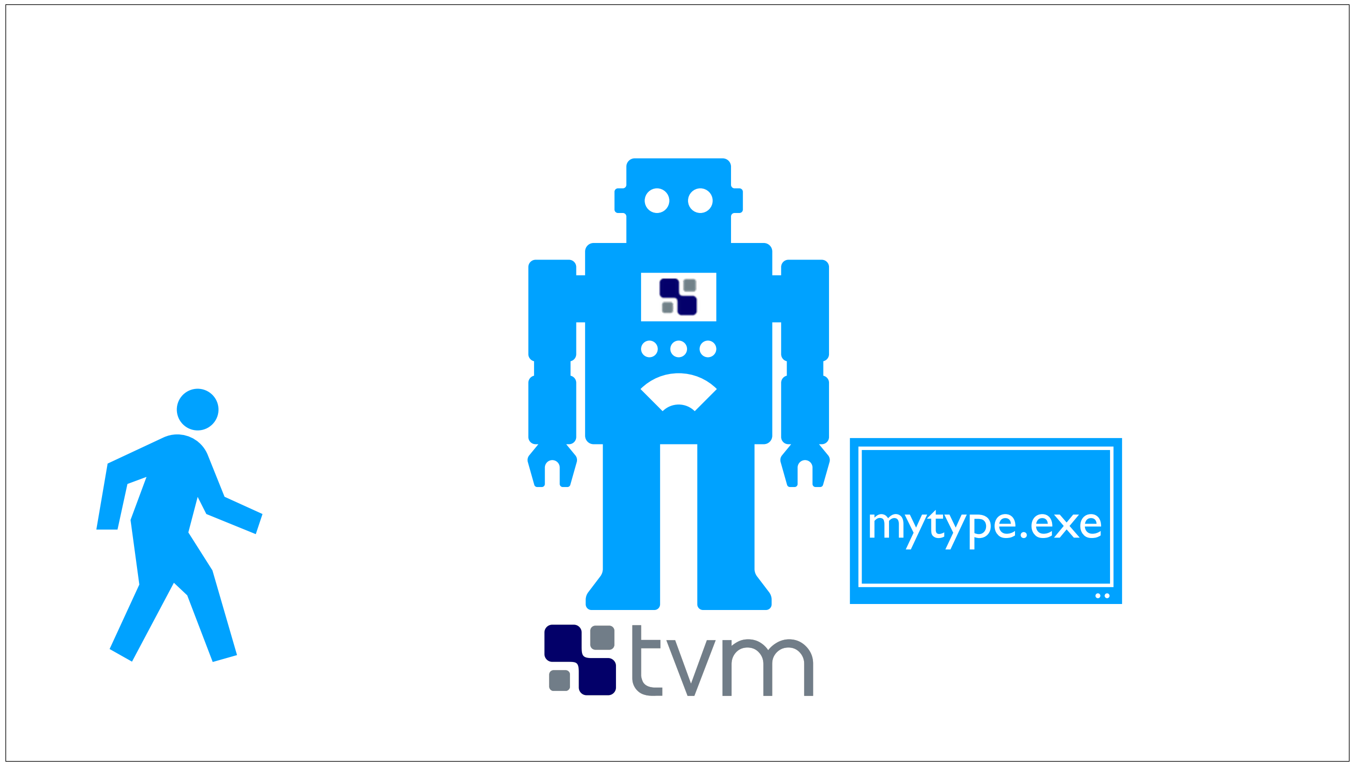
If you just
[build] give TVM your datatype library, in addition to some
[build] additional information about your type, you are easily able to
[build] compile and run programs which use your datatype! No compiler hacking required!


myType a =
myType b =
f(a,b)
...
tvm


additional info about your type
myType a =
myType b =
f(a,b)
...
tvm


myType a =
myType b =
f(a,b)
...
tvm


tvm


mytype.exe
tvm

Bring Your Own Datatypes

Now, I'll talk about the framework in detail.

What do we want?

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads
2. User **gives TVM some information** about the library

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads
2. User **gives TVM some information** about the library
3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

But first…what is TVM?

tvm

Before we go further, let’s actually talk about what TVM is.

Well, put simply, it’s an extensible, optimizing compiler for deep learning.
It aggregates a number of common optimizations for graph-based programs.
It is also incredibly extensible, allowing for work like this, and a lot of other interesting research!

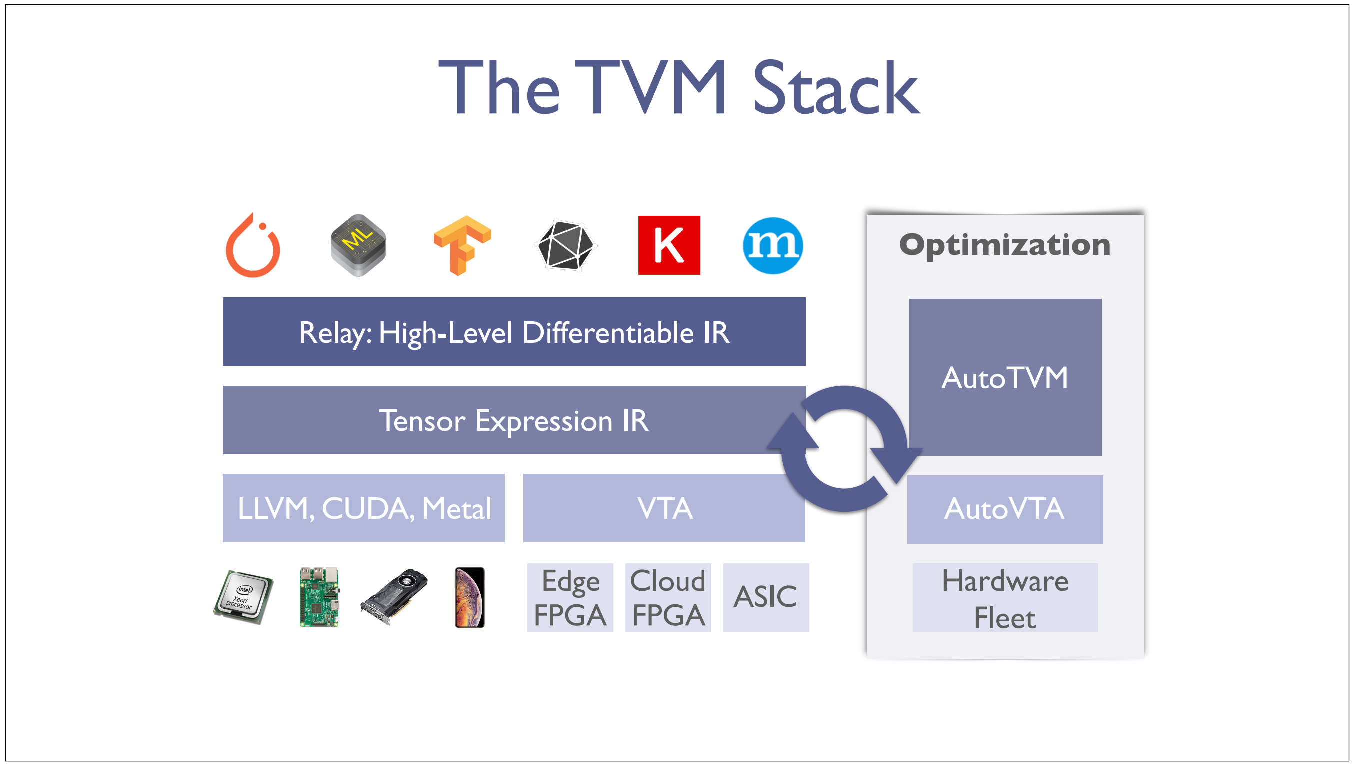
But first…what is TVM?

An **extensible, optimizing compiler** for deep learning.

But first…what is TVM?

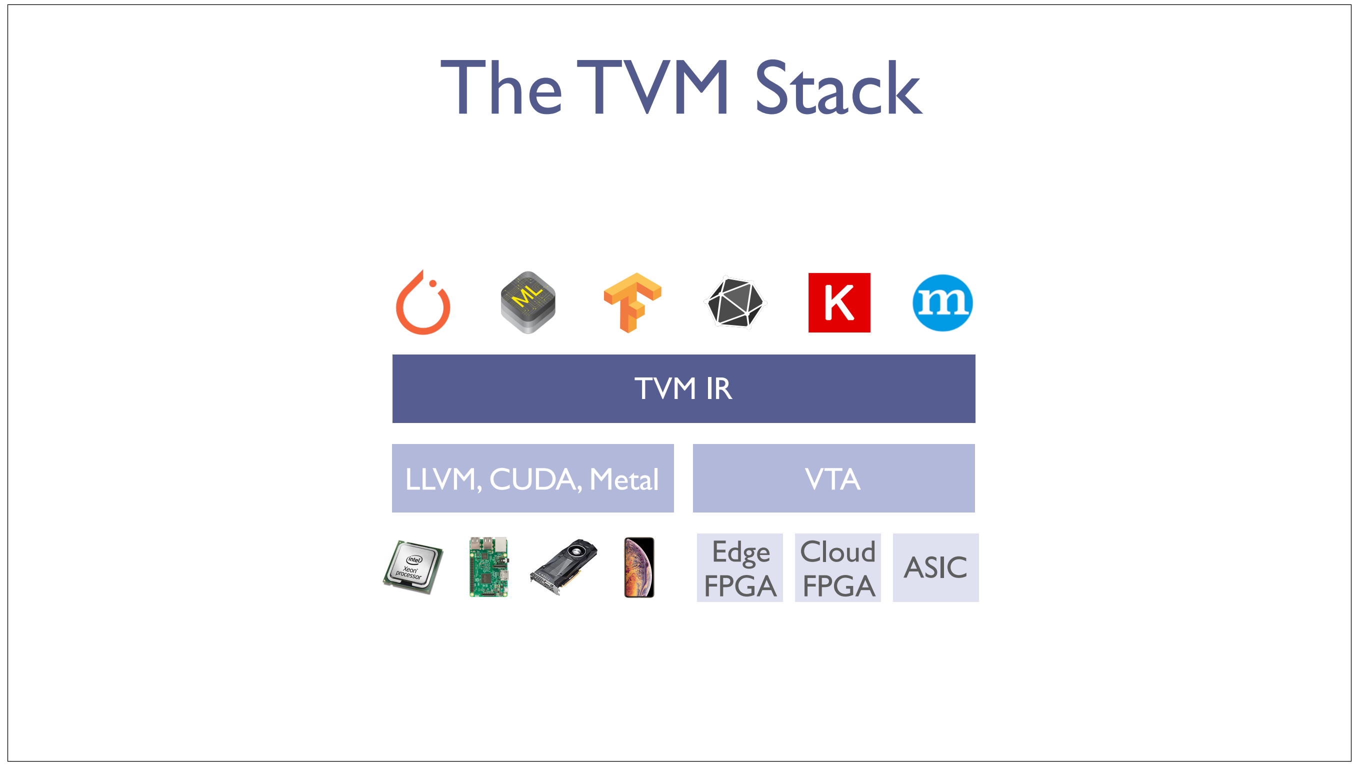
An **extensible, optimizing compiler** for deep learning.

See tvm.ai for more info!

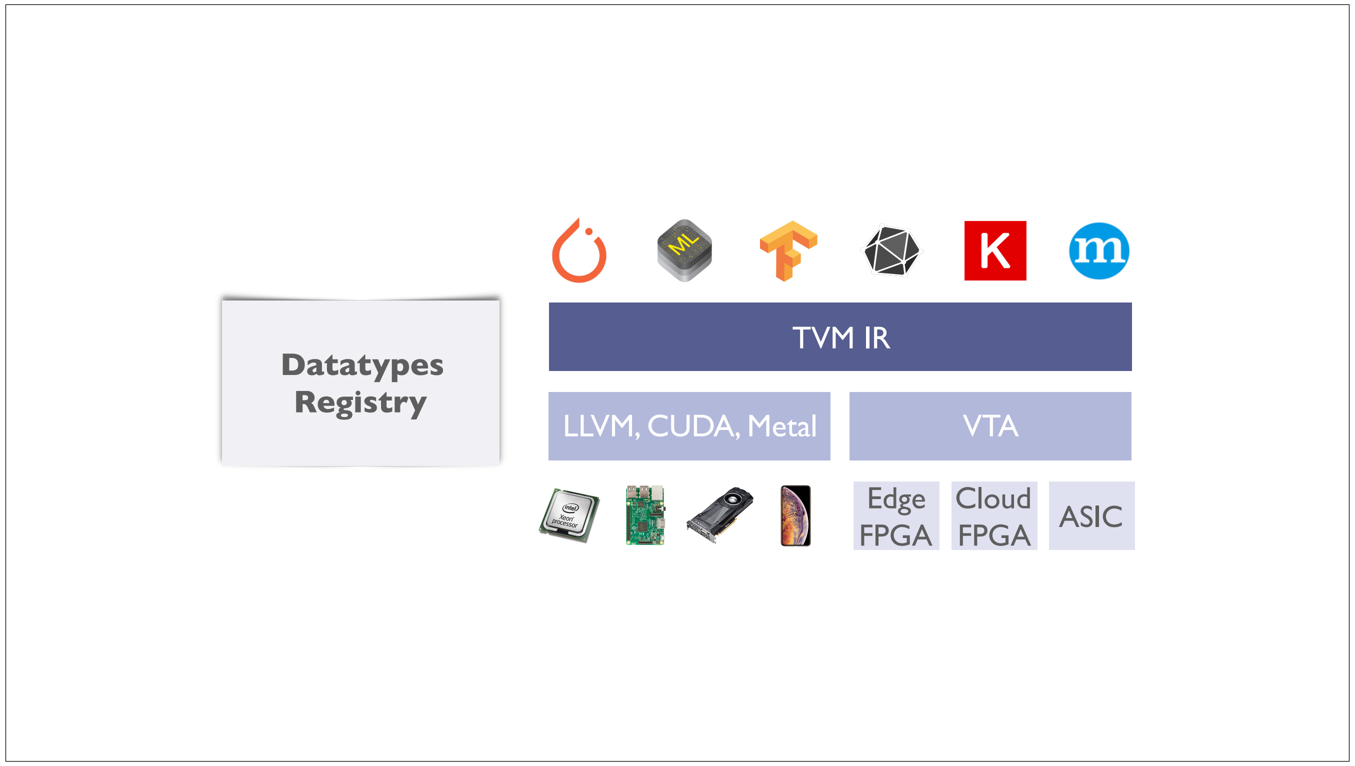


You start with a program written in TVM's DSL, or in a higher-level machine learning framework; for example, Tensorflow.
TVM then compiles through multiple intermediate representations (in the future, these will be merged)
TVM then compiles the IR to a number of supported backends, including LLVM, CUDA, and Metal. We also compile to VTA, which is an open-source deep learning accelerator being developed at UW.
And finally, TVM deploys do your device, whether it be a CPU, or a small embedded device, or an FPGA.

Now we'll talk about how the Bring Your Own Datatypes framework fits into this stack.



For our purposes, we’re going to use a simplified version of the stack

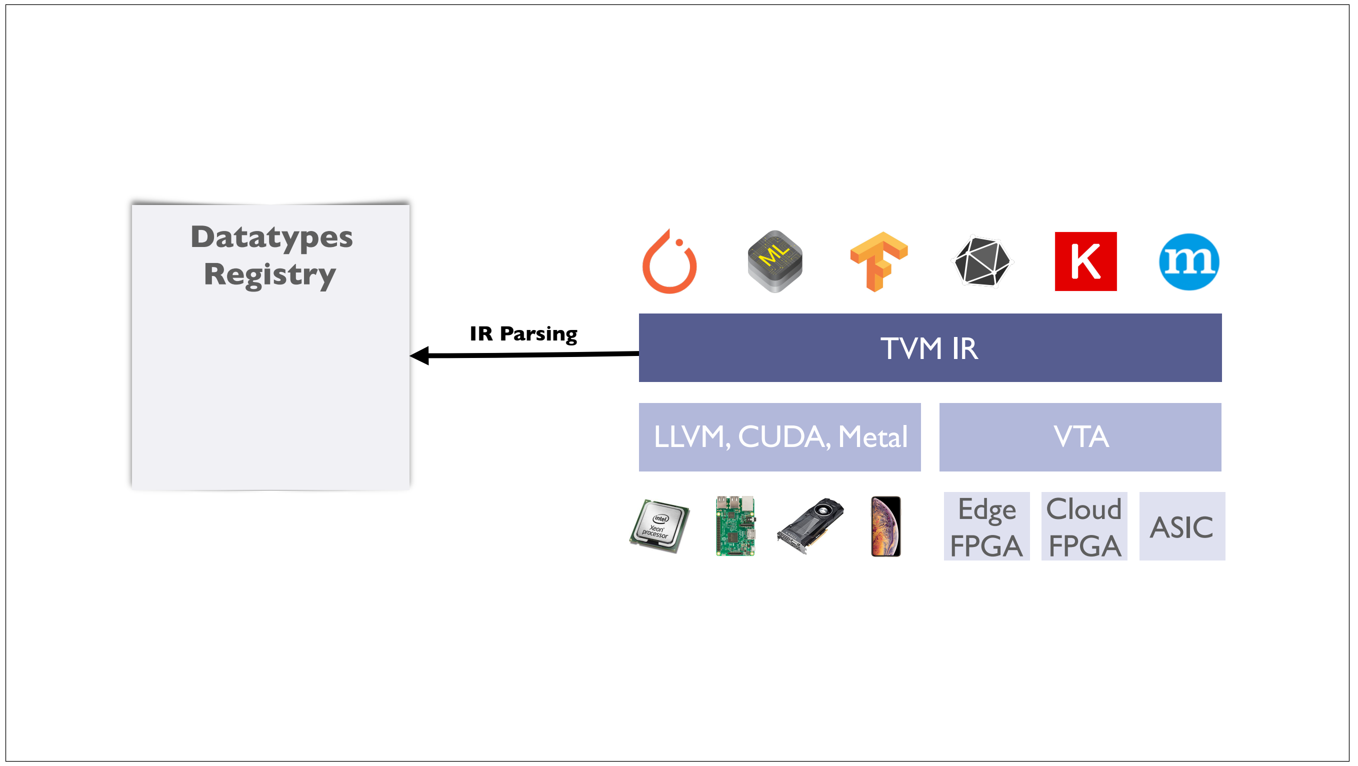


The Bring Your Own Datatypes framework is implemented as a registry of datatypes.
The registry sits alongside the normal TVM stack, and for the most part, the operation of the normal TVM stack is not affected by the registry.

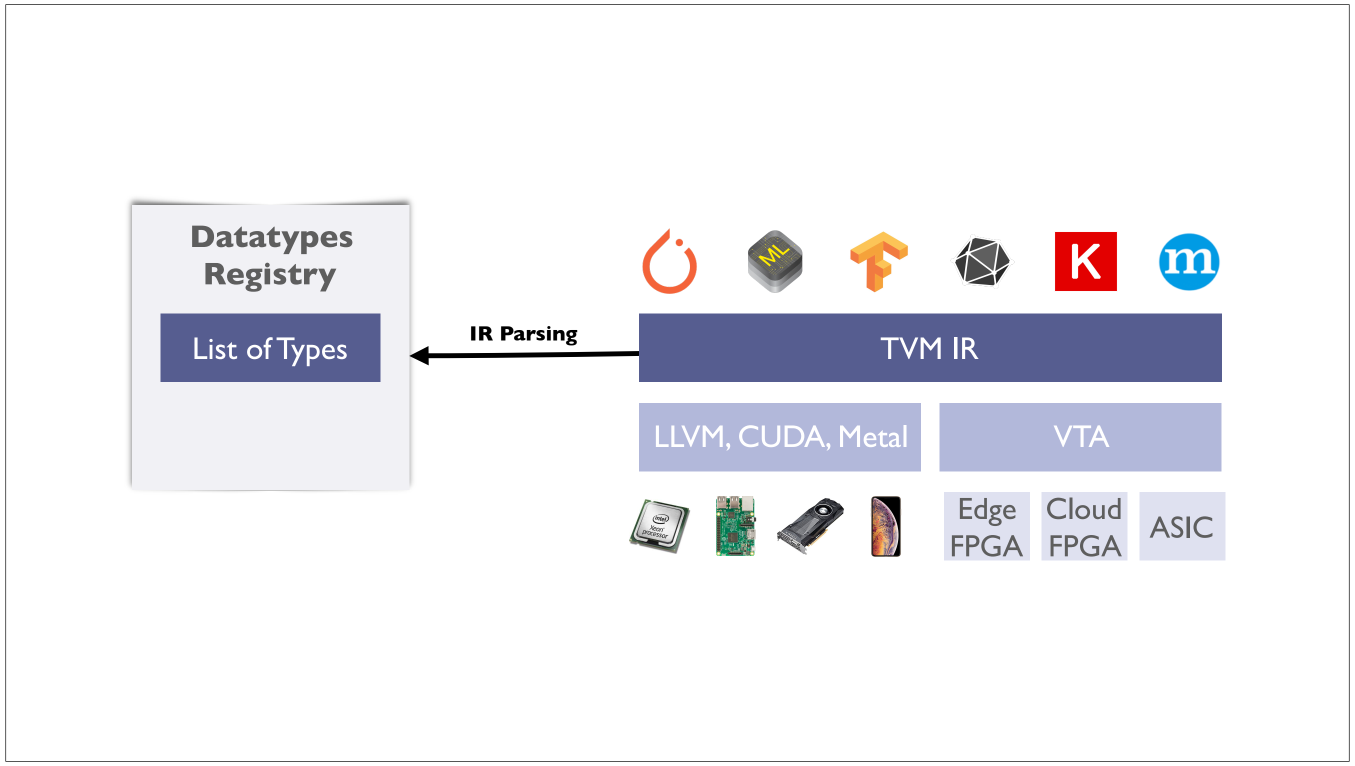
There are two key places where the registry coms into play.

[build] First is in IR parsing.
If TVM is parsing a program and encounters a datatype it doesn't recognize, it would normally just error out.
With the framework, instead of failing, it falls back to
[build] checking the registry for the type.
If the registry reports that the datatype has been registered, TVM proceeds without error.

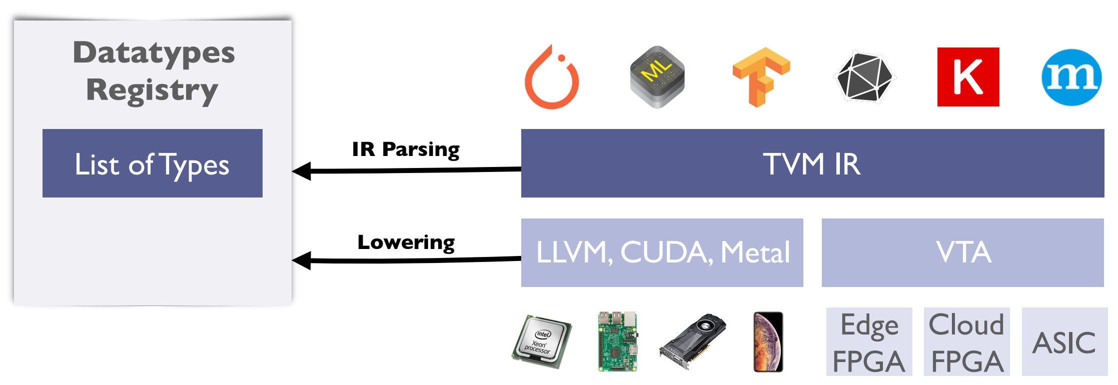
[build] Second is in lowering, when TVM compiles its IR to LLVM, CUDA, or some other language which can be compiled and run.
TVM understands how to lower its native datatypes, but not custom datatypes.
When TVM needs to lower an operation over a custom datatype,
[build] it simply defers to the registry.
For each custom type, the registry stores a set of lowering functions which TVM can call.

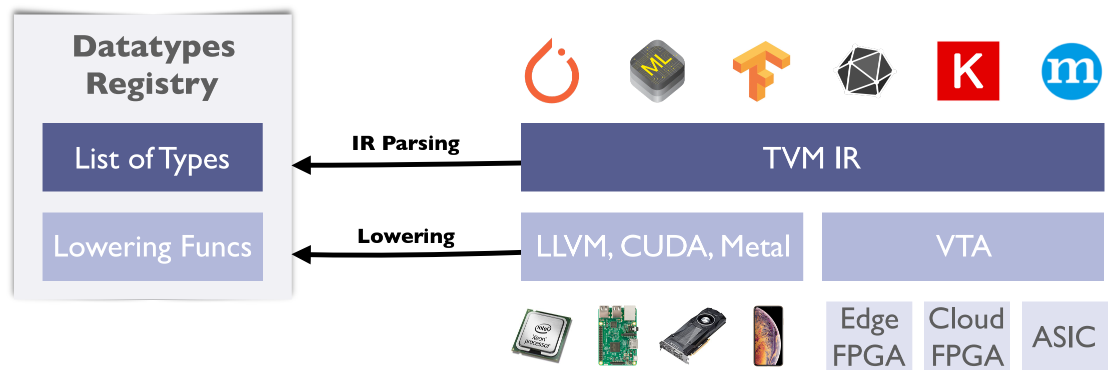


Datatypes Registry
IR Parsing
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC

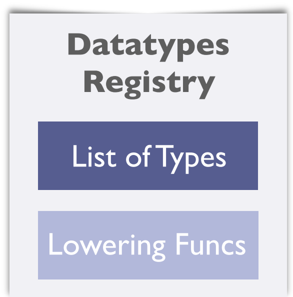


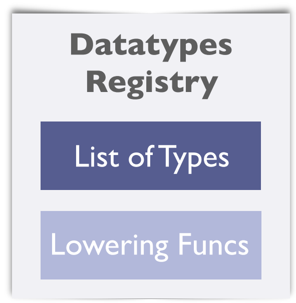
Datatypes Registry
List of Types
IR Parsing
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC


Datatypes Registry
List of Types
IR Parsing
Lowering
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC

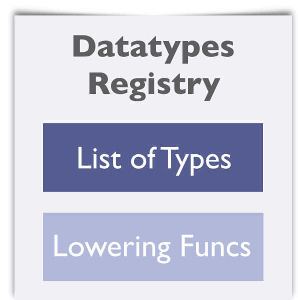


Datatypes Registry
List of Types
Lowering Funcs
IR Parsing
Lowering
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC


Datatypes Registry
List of Types
Lowering Funcs

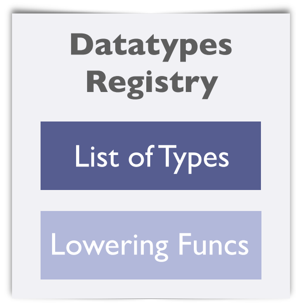


To register their datatype, the user provides a name and type code:



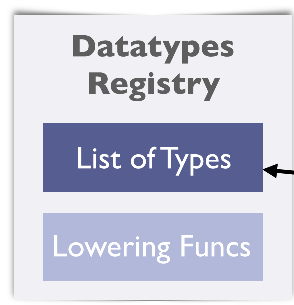
To register their datatype, the user provides a name and type code:

```
tvm.datatype.register("bfloat", 129)
```



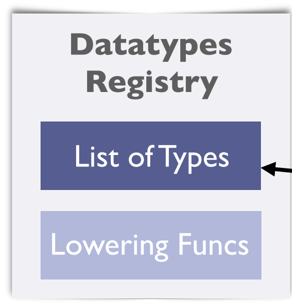
To register their datatype, the user provides a name and type code:

```
tvm.datatype.register("bfloat", 129)
```



To register their datatype, the user provides a name and type code:

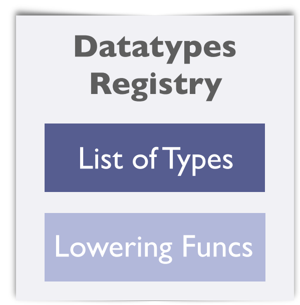
```
tvm.datatype.register("bfloat", 129)
```

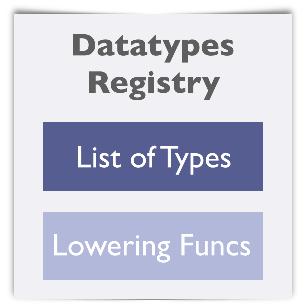


To register their datatype, the user provides a name and type code:

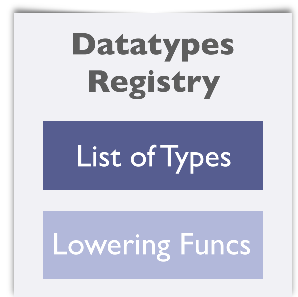
```
tvm.datatype.register("bfloat", 129)
```

This allows TVM to parse programs which use the datatype!

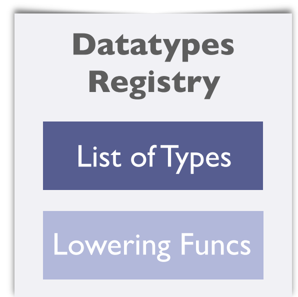

Datatypes Registry
List of Types
Lowering Funcs



Users register lowering functions with a similar API:

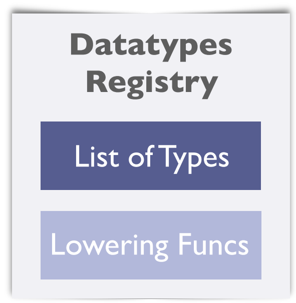


Users register lowering functions with a similar API:



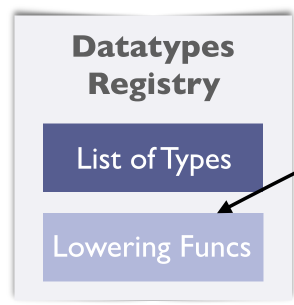
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
```



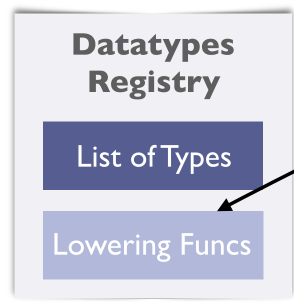
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
        lower_func, "Add", “llvm”, “bfloat”)
```



Users register lowering functions with a similar API:

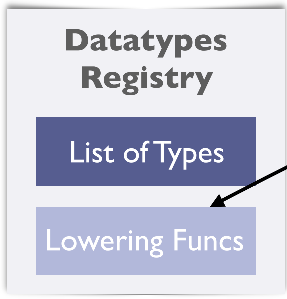
```
tvm.datatype.register_op(
        lower_func, "Add", “llvm", “bfloat”)
```



Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
         lower_func, "Add", “llvm", “bfloat”)
```

This registers a lowering function which lowers bfloat Adds when compiling to LLVM.



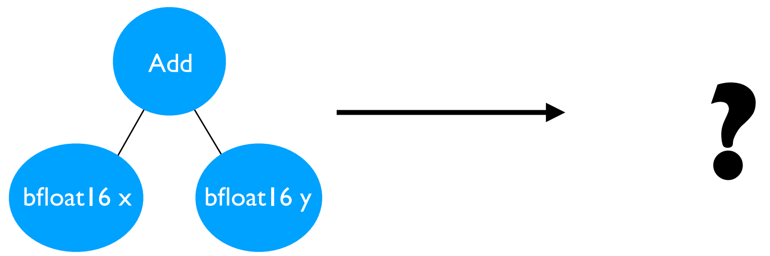
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
          lower_func, "Add", “llvm”, “bfloat”)
```

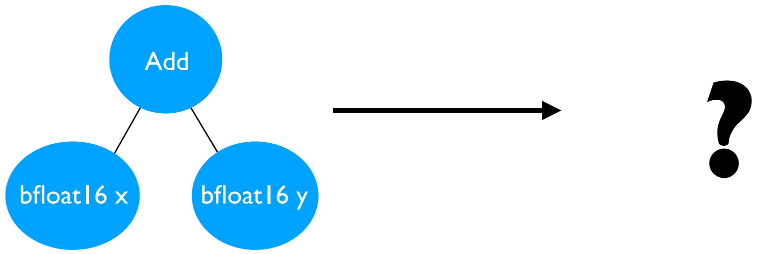
This registers a lowering function which lowers bfloat Adds when compiling to LLVM.

TVM will later use this lowering function when it spits out code!

What do lowering functions look like?

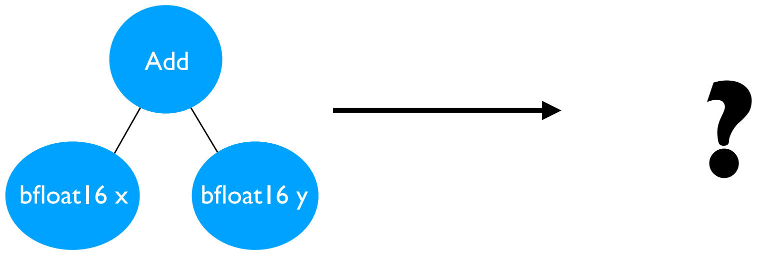


What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

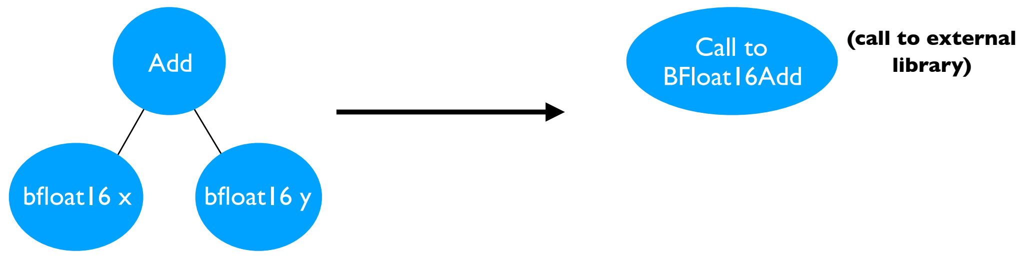
What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

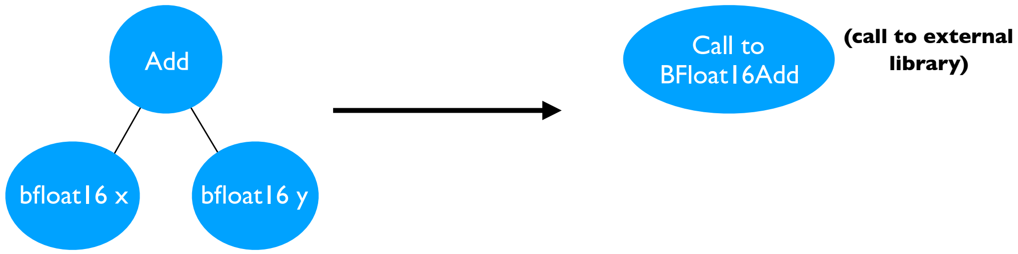
What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

What do lowering functions look like?

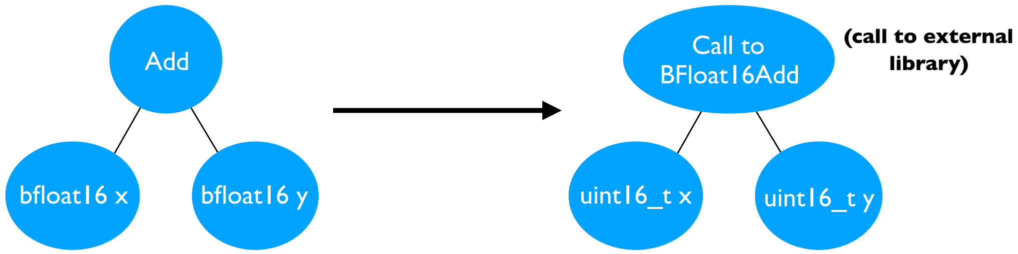


Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

What do lowering functions look like?

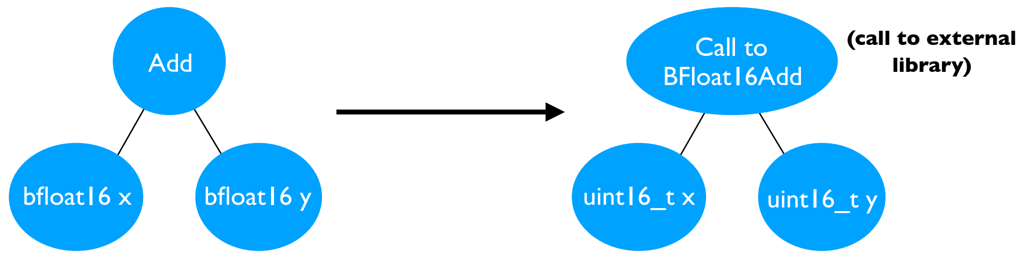


Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

What do lowering functions look like?



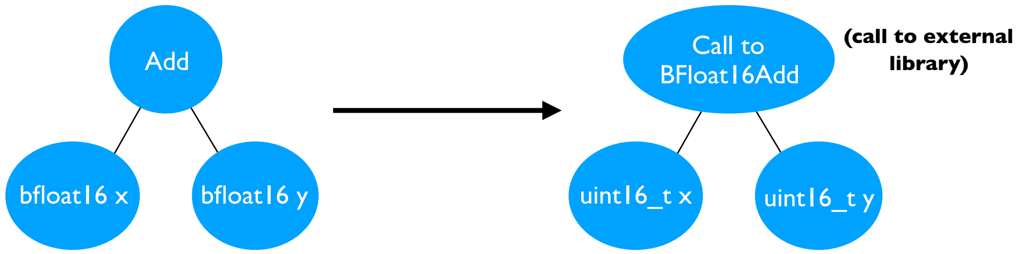
Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

We provide a helper function for creating this type of lowering function:

What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

We provide a helper function for creating this type of lowering function:
`tvm.datatype.create_lower_func("BFloat16Add")`

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads
2. User **gives TVM some information** about the library
3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

To give the lowering functions, the user essentially just needs to provide the names of library functions implementing the various operators over the datatype.

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

    - Datatype name

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

   - Datatype name

   - Lowering functions—user just provides names of library functions!

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

Evaluation

To exercise the framework, I decided to conduct a preliminary evaluation of how a model's trained accuracy changes as we change the datatype.

To exercise the framework, I decided to conduct a preliminary evaluation of how a model's trained accuracy changes as we change the datatype.

We will first discuss the experiment itself and its results—then, we will reflect on the utility of the framework.

Experiment Design

Experiment Design

1. Gathered a list of datatypes

Experiment Design

1. Gathered a list of datatypes
   - TVM-native

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made

Experiment Design

1. Gathered a list of datatypes
    - TVM-native
    - Hand-made
    - From GitHub

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

3. Converted the pretrained weights to the custom datatypes.

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub
2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch
3. Converted the pretrained weights to the custom datatypes.
   - This was done without retraining

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub
2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch
3. Converted the pretrained weights to the custom datatypes.
   - This was done without retraining
4. Ran the models with the converted datatypes over a sample of the CIFAR-10 test set (100 images each) using TVM

Datatypes:

Datatypes:

- **TVM-native float32** (not using the framework)

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit
- Two implementations of **bfloat16**:
  - My own "naive" implementation
  - biovault-bfloat16: another implementation from GitHub

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit
- Two implementations of **bfloat16**:
  - My own "naive" implementation
  - biovault-bfloat16: another implementation from GitHub
- "**noptype**": always returns 0

Models:

- MobilenetV1
- Resnet50

Experiment Results and Evaluation

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

This table shows the accuracy results for our custom datatypes, with float32 as the baseline to compare against.

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

The first thing to notice is that there is a very clear bimodal distribution of model accuracy.
It seems that datatypes either work or they don't.

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

Furthermore, within those two modes, there is very little variation.
In the cases where the model loses all of its accuracy, it drops down to just about random chance, which would be 10% for the 10-class CIFAR-10 dataset.

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

…and within the datatypes that worked, there was no deviation from float32 accuracy.

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

Unsurprisingly, which datatypes lose accuracy seem to be correlated with the size of the type.

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
✅ **Universal posit16**	0.77	0.71
✅ **Universal posit32**	0.77	0.71
libposit posit8	0.08	0.1
✅ **libposit posit16**	0.77	0.71
✅ **libposit posit32**	0.77	0.71

For example, the larger posits retain their accuracy,

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
❌ **Universal posit8**	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
❌ **libposit posit8**	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

while the 8-bit does not.

Same size!

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
❌ our bfloat16	0.08	0.11
❌ biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
✅ Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
✅ libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

What is interesting, though, is that the 16-bit posits do not lose accuracy, while the 16-bit bfloats do!

This is one of the more illuminating results of the experiment, and it begins to hint at the underlying implementations of each datatype.

It is hard to say without doing some deeper debugging, but we can hypothesize that the implementation of the posit16 causes it to accumulate less error than the bfloat16.
This is not hard to believe—our configuration of posit16 can represent many more numbers in the range -1,1 than bfloat16, which is a range where weights and activations often lie.

Framework Evaluation

Now, we will use this experiment to evaluate the framework as a whole.

To evaluate my work, I wanted to evaluate three aspects of the framework: [read them]

I evaluate overhead quantitatively, using the results of the experiment.

Ease of use and breadth of datatypes I evaluate qualitatively, relating experiences I had in building this experiment.

I wanted to evaluate three aspects of the framework:

- Overhead

I wanted to evaluate three aspects of the framework:

- Overhead
- Ease of use

I wanted to evaluate three aspects of the framework:

- Overhead
- Ease of use
- Breadth of datatypes which can be used

Overhead

To measure overhead, I compared the inference time of three types:

To measure overhead, I compared the inference time of three types:

- TVM-native float32

To measure overhead, I compared the inference time of three types:

- TVM-native float32
- float32 implemented in the framework

To measure overhead, I compared the inference time of three types:

- TVM-native float32
- float32 implemented in the framework
- “noptype”: a custom type that does no work

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Here are the results.

We use native float32 as a baseline, and measure the framework overhead for the other two types.

We can see that the framework overhead ranges from 1.1x in the best case to 2.5x in the worst.

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

The first thing we'll notice is that the overhead seems to depend heavily on the model.

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

The other thing that we might notice is the large gap in overhead between the two types in Resnet.

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

One of the main projects in the Bring Your Own Datatypes framework is to enable the inlining of LLVM byte code. I suspect once we do this, we'll see very different overhead numbers!

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much
- float32 is low-overhead in Mobilenet, which is optimized for low total float ops

mobilenet v1	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much
- float32 is low-overhead in Mobilenet, which is optimized for low total float ops
- …but float32 is high-overhead in ResNet → native float32 ResNet much more optimized!

Ease of Use

We will evaluate ease of use qualitatively, by looking at the code needed to use a real custom datatype in a real workload. Specifically, we'll look at the code needed to use our libposit-posit32 code.

By the Numbers…

Here, we look at one example: implementing libposit posit32.

Because we need to make sure the calling convention of the Bring Your Own Datatype library matches the calling convention of the datatype library, we often need to build a small wrapper over the library. For libposit, to wrap over the 12 operators needed for Mobilenet and Resnet, our wrapper was about 70 lines of code.

From a programming perspective, converting the model is very low overhead—the most overhead

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
  - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
    - **57 lines** register the datatype and define lowering functions for the 12 operators,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
  - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
  - **57 lines** register the datatype and define lowering functions for the 12 operators,
  - **3 lines** convert the model to posit32,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
    - **57 lines** register the datatype and define lowering functions for the 12 operators,
    - **3 lines** convert the model to posit32,
    - **3 lines** convert the input, run the model, and convert the output.

How could we improve?

How could we improve?

- Allow the user to specify their own calling convention, removing the need for a wrapper over the library

How could we improve?

- Allow the user to specify their own calling convention, removing the need for a wrapper over the library
- Implement cleaner registration functions in the TVM Python frontend

Breadth of Datatypes

Breadth of Datatypes

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with "external" state

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with "external" state
- Datatypes with elements larger than 64 bits

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with "external" state
- Datatypes with elements larger than 64 bits

Could potentially be implemented by allowing datatypes to attach metadata to each scalar

Future Work

Future Work

- Training in TVM—ramifications for custom datatypes?

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM
  - Allow users to supply optimized kernels for higher-level operators, e.g. posit conv2d

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM
  - Allow users to supply optimized kernels for higher-level operators, e.g. posit conv2d
- Tackle complex datatypes like block floating point

In conclusion, I have presented the Bring Your Own Datatypes framework.

In conclusion, I have presented the Bring Your Own Datatypes framework.

In conclusion, I have presented the Bring Your Own Datatypes framework.

I have shown how the framework can enable useful datatype research.

Thank You!

Extra Slides

Registering the Datatype

```python
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

The first thing we do, before we write our Relay program, is register the datatype.

Registering the Datatype

```python
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

✅ **Any loaded C-linkage functions will work!**

Registering the Datatype

```
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

✅ Any loaded C-linkage functions will work!

✅ Helper functions for common-case

Implementing the Datatype

```
#include <cstdint>
#include <posit.h>

posit32_t Uint32ToPosit32(uint32_t in) {
  return posit32_reinterpret(in);
}

uint32_t Posit32ToUint32(posit32_t in) {
  return posit32_bits(in);
}

extern "C" uint32_t Posit32Add(uint32_t a, uint32_t b) {
  auto a_p = Uint32ToPosit32(a);
  auto b_p = Uint32ToPosit32(b);
  auto sum_p = posit32_add(a_p, b_p);
  return Posit32ToUint32(sum_p);
}
```

Implementing the Datatype

```
#include <cstdint>
#include <posit.h>

posit32_t Uint32ToPosit32(uint32_t in) {
  return posit32_reinterpret(in);
}

uint32_t Posit32ToUint32(posit32_t in) {
  return posit32_bits(in);
}

extern "C" uint32_t Posit32Add(uint32_t a, uint32_t b) {
  auto a_p = Uint32ToPosit32(a);
  auto b_p = Uint32ToPosit32(b);
  auto sum_p = posit32_add(a_p, b_p);
  return Posit32ToUint32(sum_p);
}
```

✗ Inflexible calling convention

Converting the Program

```
module, params = load_resnet50()
module, params = change_dtype('float32', 'custom[posit32]32',
                              module['main'], params)

ex = relay.create_executor(mod=module)
model = ex.evaluate()
```

Converting the Program

```
module, params = load_resnet50()
module, params = change_dtype('float32', 'custom[posit32]32',
                              module['main'], params)

ex = relay.create_executor(mod=module)
model = ex.evaluate()
```

✅ **Changing datatype of a program is easy!**

Running the Program

```
image = ... # load image

image = convert_ndarray('custom[posit32]32', image)

with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

Running the Program

```python
image = ... # load image

image = convert_ndarray('custom[posit32]32', image)

with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

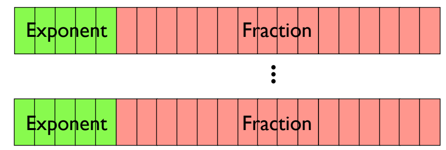
✅ **Converting data is also easy!**

Running the Program

```
image = ... # load image
                    ✅ Converting data is also easy!
image = convert_ndarray('custom[posit32]32', image)
                    ❌ Have to manually disable vectorization
with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

Flexpoint



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

TODO: tie back to TPU slide: how does this hit each of the three points?

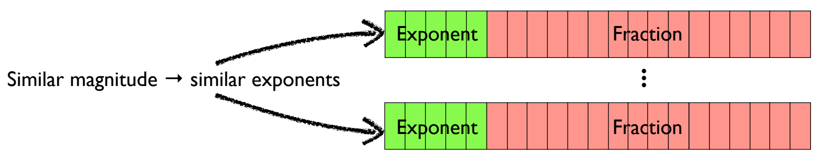
The last datatype I want to talk about is perhaps the most interesting, from a systems design perspective, and begins to show how far we can push datatype design.

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.

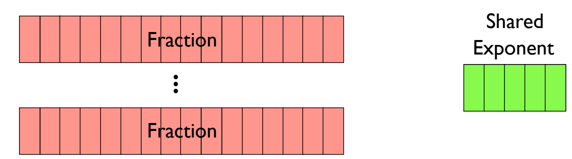
If this is the case, then we can often have…[slide transition]

Flexpoint



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

Flexpoint

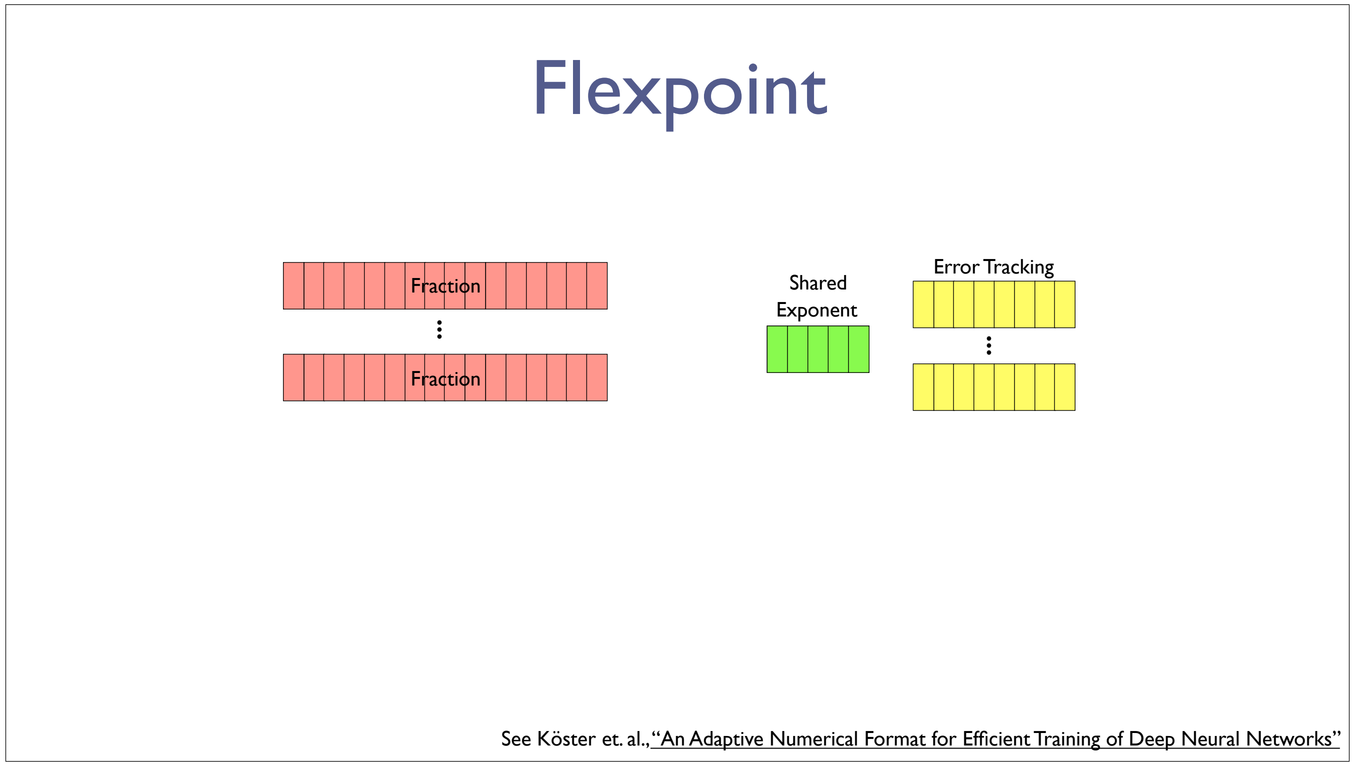


See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

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

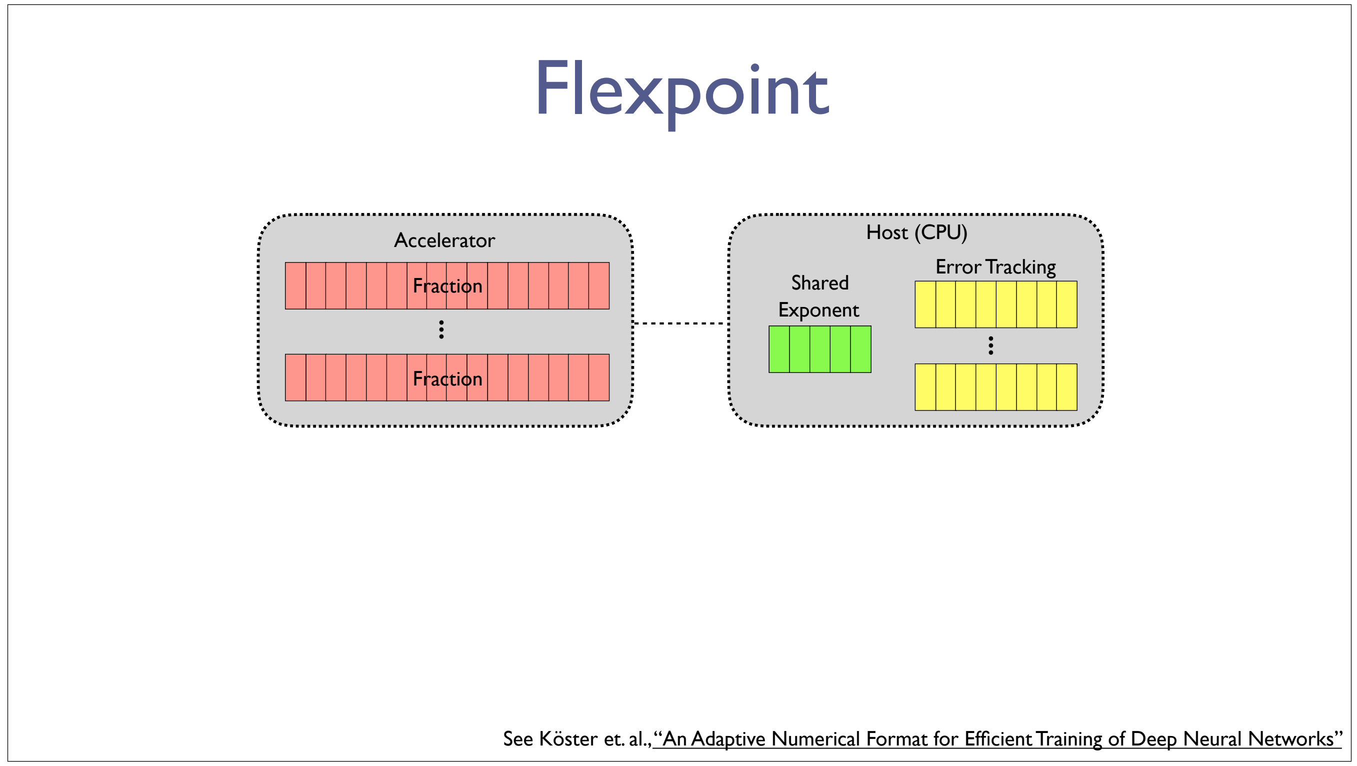
The larger our block, the more numbers we have sharing an exponent, and thus, the more potential space saving.
However, this sharing will also introduce error, as the exponent will not accurately represent the full range of values in the block.

[slide transition]



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]

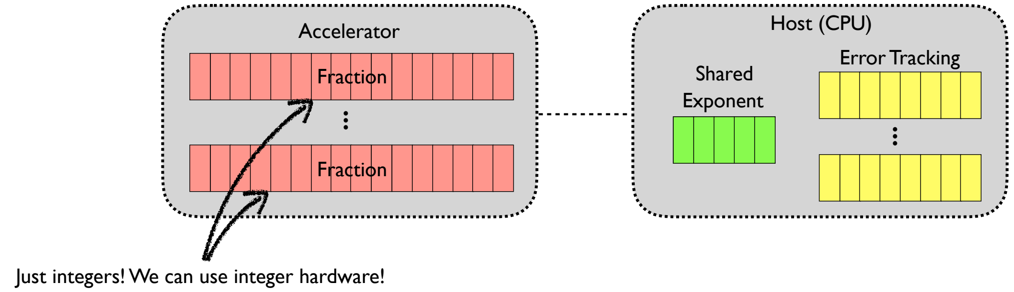


…physically separate the fractional bits from the exponent and error tracking bits. The fractional bits are all that's really needed to do computation on the accelerator, and so we'll keep everything else on the host device, for example, the CPU that's interacting with the accelerator.

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"