Bring Your Own Datatypes

Gus Smith
University of Washington
Why?
<table>
<thead>
<tr>
<th>Data type</th>
<th>fp32</th>
<th>fp16</th>
<th>int8</th>
</tr>
</thead>
</table>

Why?
Beating Floating Point at its Own Game: Posit Arithmetic

*John L. Gustafson*, *Isaac Yonemoto*

A new data type called a *posit* is designed as a direct drop-in replacement for IEEE Standard 754 floating-point numbers (floats). Unlike earlier forms of universal number (unum) arithmetic, posits do not require interval arithmetic or variable size operands; like floats, they round if an answer is inexact. However, they provide compelling advantages over floats, including larger dynamic range, higher accuracy, better closure, bitwise identical results across systems, simpler hardware, and simpler exception handling. Posits never overflow to infinity or underflow to zero, and “Not-a-Number” (NaN) indicates an action instead of a bit pattern. A posit processing unit takes less circuitry than an IEEE float FPU. With lower power use and smaller silicon footprint, the posit operations per second (POPS) supported by a chip can be significantly higher than the FLOPS using similar hardware resources. GPU accelerators and Deep Learning processors, in particular, can do more per watt and per dollar with posits, yet deliver superior answer quality.

A comprehensive series of benchmarks compares floats and posits for decimals of accuracy produced for a set precision. Low precision posits provide a better solution than “approximate
Why?

| Data type | fp32 | fp16 | int8 |

---

Beating Floating Point at its Own Game

*John L. Gustafson¹, Isaac Yonemoto²*

A new data type called a *posit* is designed as a replacement for 754 floating-point numbers (floats). Unlike earlier floating-point formats, posit numbers do not require interval arithmetic or variable precision, nor do they carry an error flag. Unlike earlier floating-point numbers, a posit's answer is exact. However, they provide compelling advantages over conventional floating-point numbers in terms of wide range, higher accuracy, better closure, bitwise identical comparison, and simpler exception handling. Posits never overflow, never underflow, never divide by zero, and never have a-Number” (NaN) indicates an action instead of a value. A posit's operation consumes less circuitry than an IEEE float FPU. With lower power and higher point operations per second (POPS) supported by a chip, a posit FPU is expected to outperform a floating-point FPU using similar hardware resources. GPU accelerators can do more per watt and per dollar with posits, yet the performance is not limited to GPUs.

A comprehensive series of benchmarks comparing posits to conventional floats have been produced for a set precision. Low precision posits have been found to achieve speedups similar to those found for high precision, but with lower overhead for intermediate floating-point representations. This demonstrates that posits have potential for high-performance computing, where hardware resources are limited.

---

Rethinking floating point for deep learning

*Jeff Johnson*

Facebook AI Research

New York, NY

jhj@fb.com

Abstract

Reducing hardware overhead of neural networks for faster or lower power inference and training is an active area of research. Uniform quantization using integer multiply-add has been thoroughly investigated, which requires learning many quantization parameters, fine-tuning training or other prerequisites. Little effort is made to improve floating point relative to this baseline; it remains energy ineff-
The Goal: BYO-Datatype

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

Edge FPGA

Cloud FPGA

ASIC
The Goal: BYO-Datatype

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal | VTA

Edge FPGA | Cloud FPGA | ASIC

+ your custom datatypes!
The Goal: BYO-Datatype

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

+ other datatype libraries

+ your custom datatypes!

NGA SoftPosit

Edge FPGA

Cloud FPGA

ASIC
The Goal: BYO-Datatype

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

Optimization

AutoTVM

AutoVTA

Edge FPGA

Cloud FPGA

ASIC

Hardware Fleet
The Goal: BYO-Datatype

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

Optimization

AutoTVM

AutoVTA

Hardware Fleet

Edge FPGA

Cloud FPGA

ASIC

+ other datatype libraries
Future Directions: Hardware

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

Edge FPGA, Cloud FPGA, ASIC
Future Directions: Hardware

High-Level Differentiable IR

Tensor Expression IR

LLVM, CUDA, Metal

VTA

facebookresearch / deepfloat

Edge FPGA Cloud FPGA ASIC

+ other hardware libraries
Future Directions: Hardware

High-Level Differentiable IR
Tensor Expression IR
LLVM, CUDA, Metal
VTA
Optimization
AutoTVM
AutoVTA
Tunable Datatype Software and Hardware
Hardware Fleet

+ other hardware libraries

facebookresearch/deepfloat
What it Looks Like Today
What it Looks Like Today

Adding a custom datatype in Python:

```python
tvm.register_datatype("myfloat", 24)
X = tvm.placeholder((3,), name="X")
Y = topi.cast(X, dtype="custom[myfloat]32")
```
What it Looks Like Today

Adding a custom datatype in Python:

tvm.register_datatype("myfloat", 24)

X = tvm.placeholder((3,), name="X")
Y = topi.cast(X, dtype="custom[myfloat]32")

cast = tvm.build(flist[0], target=tgt)
What it Looks Like Today

Adding a custom datatype in Python:

```python
tvm.register_datatype("myfloat", 24)

X = tvm.placeholder((3,), name="X")
Y = topi.cast(X, dtype="custom[myfloat]32")

cast = tvm.build(flist[0], target=tgt)

x = tvm.nd.array(np.random.uniform(size=3).astype(X.dtype), ctx)
y = tvm.nd.empty(Y.shape, dtype=Y.dtype, ctx=ctx)
```
What it Looks Like Today

Adding a custom datatype in Python:

```python
import tvm
import numpy as np

tvm.register_datatype("myfloat", 24)

X = tvm.placeholder((3,), name="X")
Y = tvm.topi.cast(X, dtype="custom[myfloat]32")

cast = tvm.build(flist[0], target=tgt)

x = tvm.nd.array(np.random.uniform(size=3).astype(X.dtype), ctx)
y = tvm.nd.empty(Y.shape, dtype=Y.dtype, ctx=ctx)

cast(x,y)
```
What it Looks Like Today

Adding a custom datatype in Python:

tvm.register_datatype("myfloat", 24)

X = tvm.placeholder((3,), name="X")
Y = topi.cast(X, dtype="custom[myfloat]32")

cast = tvm.build(flist[0], target=tgt)

x = tvm.nd.array(np.random.uniform(size=3).astype(X.dtype), ctx)
y = tvm.nd.empty(Y.shape, dtype=Y.dtype, ctx=ctx)

cast(x,y)

Defining lowering from our custom datatype in C++:

TVM_REGISTER_GLOBAL("tvm.datatypes.lower.llvm.cast.myfloat.float")
 .set_body([](runtime::TVMArgs args, runtime::TVMRetValue *rv) {
   Expr e = args[0];
   const ir::Cast* cast = e.as<ir::Cast>();
   internal_assert(cast);
   auto type = cast->type;
   *rv = reinterpret(tvm::UInt(type.bits(), type.lanes()), cast->value);
 });
What it Looks Like Today

Adding a custom datatype in Python:

```python
tvm.register_datatype("myfloat", 24)
X = tvm.placeholder((3,), name="X")
Y = topi.cast(X, dtype="custom[myfloat]32")

cast = tvm.build(flist[0], target=tgt)

x = tvm.nd.array(np.random.uniform(size=3).astype(X.dtype), ctx)
y = tvm.nd.empty(Y.shape, dtype=Y.dtype, ctx=ctx)

cast(x, y)
```

Defining lowering from our custom datatype in C++:

```c++
TVM_REGISTER_GLOBAL("tvm.datatypes.lower.llvm.cast.myfloat.float")
.set_body([](tvm::TVMArgs args, tvm::TVMRetValue *rv) {
  Expr e = args[0];
  const ir::Cast* cast = e.as<ir::Cast>();
  internal_assert(cast);
  auto type = cast->type;
  *rv = reinterpret(tvm::UInt(type.bits(), type.lanes()), cast->value);
});
```

```
x: [0.25599805 0.5752605  0.0941305 ]
y: [1048777261 1058227270 1036044158]
```
Try it out and get involved!

My TVM fork: 
https://github.com/gussmith23/tvm
Or reach out to me at 
gussmith@cs.washington.edu