APyTypes

Algorithmic Data Types in Python for Efficient Simulation of Finite Word-Length Effects

Mikael Henriksson, Theodor Lindberg, and Oscar Gustafsson

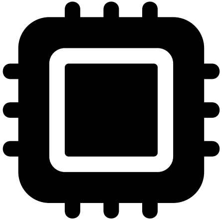
LINKÖPING UNIVERSITY

Why finite word-length effect simulations?

NumPy

MATLAB

Word lengths? →



Algorithm designed in software using 32-bit or 64-bit floating-point

Specialized hardware implementation

What are we trying to do?

What are we trying to do?

- Study the performance of algorithms under the effect of quantization.
  - Simulation of finite word-length effects in RTL simulators is error-prone and slow.
  - If word lengths can be determined during algorithm development, time can be saved in the hardware implementation process.

What are we trying to do?

- Study the performance of algorithms under the effect of quantization.
  - Simulation of finite word-length effects in RTL simulators is error-prone and slow.
  - If word lengths can be determined during algorithm development, time can be saved in the hardware implementation process.
- Custom word-length bit-exact simulations in software. These act as good reference models ("golden" references) and can be used for co-simulations.
  - A NumPy-like library with fully configurable number representations, of both scalar and array types.
  - Always produces bit-exact results that can be used for, e.g., design co-verification.

Previous works

C/C++ custom word-length libraries:

- Xilinx HLS Arbitrary Precision Types (AP Types)
- Siemens EDA Algorithmic C Datatypes (AC Types)

Previous works

C/C++ custom word-length libraries:

- Xilinx HLS Arbitrary Precision Types (AP Types)
- Siemens EDA Algorithmic C Datatypes (AC Types)

Python custom word-length libraries:

- ml-dtype (`bfloat16`, `float8`, `int4`, `int2`)
- MPtorch (quantization for ML training)
- fpbinary (custom word-length fixed-point library)
- fxpmath (custom word-length fixed-point library)

Previous works

C/C++ custom word-length libraries:

- Xilinx HLS Arbitrary Precision Types (AP Types)
- Siemens EDA Algorithmic C Datatypes (AC Types)

Python custom word-length libraries:

- ml-dtype (`bfloat16`, `float8`, `int4`, `int2`)
- MPtorch (quantization for ML training)
- fpbinary (custom word-length fixed-point library)
- fxpmath (custom word-length fixed-point library)

More in-depth comparisons:
`https://apytypes.github.io/apytypes/comparison.html`

What is APyTypes?

- Python library for bit-exact custom fixed- and floating-point formats.
- Implemented with a performant C++ backend.
- Tailored towards algorithm and digital hardware designers.
- Designed for exploration of finite word-length effects.
- Leverages and integrates the rich ecosystem of Python (NumPy, Matplotlib etc.).

Introductory example

Introductory example

```
# Double-precision floating-point (64-bit) FIR filter using NumPy
import numpy as np
h = np.array(np.fromfile("lpass.csv"))
x = np.array(np.fromfile("input.csv"))
result = np.convolve(h, x)
```

Introductory example

```
# Double-precision floating-point (64-bit) FIR filter using NumPy
import numpy as np
h = np.array(np.fromfile("lpass.csv"))
x = np.array(np.fromfile("input.csv"))
result = np.convolve(h, x)
```

```
# Fixed-point FIR filter using APyTpyes
from apytypes import APyFixedArray, convolve
import numpy as np
h = APyFixedArray.from_float(np.fromfile("lpass.csv"), bits=7, int_bits=1)
x = APyFixedArray.from_float(np.fromfile("input.csv"), bits=16, int_bits=2)
result = convolve(h, x)
```

Scalar classes

Fixed-Point

Two's complement binary fixed-point characterized by the number of bits before and after a binary point.

$$\overbrace{\underbrace{x_{n-1}\ x_{n-2}\ \dots\ x_{k+1}\ x_k}_{\texttt{int\_bits}}\ .\ \underbrace{x_{k-1}\ x_{k-2}\ \dots\ x_1\ x_0}_{\texttt{frac\_bits}}}^{\texttt{bits}}$$

Scalar classes

Fixed-Point

Two's complement binary fixed-point characterized by the number of bits before and after a binary point.

$$\overbrace{\underbrace{x_{n-1}\ x_{n-2}\ \dots\ x_{k+1}\ x_k}_{\texttt{int\_bits}}\ .\ \underbrace{x_{k-1}\ x_{k-2}\ \dots\ x_1\ x_0}_{\texttt{frac\_bits}}}^{\texttt{bits}}$$

$$110.01_2 = -1.75$$

Scalar classes

Fixed-Point

Two's complement binary fixed-point characterized by the number of bits before and after a binary point.

$$\overbrace{\underbrace{x_{n-1}\ x_{n-2}\ \cdots\ x_{k+1}\ x_k}_{\texttt{int\_bits}}\ .\ \underbrace{x_{k-1}\ x_{k-2}\ \cdots\ x_1\ x_0}_{\texttt{frac\_bits}}}^{\texttt{bits}}$$

$$\begin{aligned} 1\,1\,0\,.\,0\,1_2 &= -1.75 \\ &= \texttt{APyFixed(0b110\_01, int\_bits=3, frac\_bits=2)} \end{aligned}$$

Scalar classes

Fixed-Point

```
from apytypes import APyFixed, QuantizationMode, OverflowMode
```

Scalar classes

Fixed-Point

```
from apytypes import APyFixed, QuantizationMode, OverflowMode

a = APyFixed.from_float(3.5, int_bits=4, frac_bits=1)
b = APyFixed(0b00_111, bits=5, int_bits=2) # 7 / 2**(5-2) = 0.875
```

Scalar classes

Fixed-Point

```
from apytypes import APyFixed, QuantizationMode, OverflowMode

a = APyFixed.from_float(3.5, int_bits=4, frac_bits=1)
b = APyFixed(0b00_111, bits=5, int_bits=2) # 7 / 2**(5-2) = 0.875

# Word lengths are statically increased based on operation
c = a + b # APyFixed(35, bits=8, int_bits=5) = 4.375
d = a * b # APyFixed(49, bits=10, int_bits=6) = 3.0625
```

Scalar classes

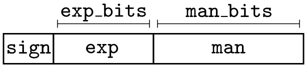
Fixed-Point

```
from apytypes import APyFixed, QuantizationMode, OverflowMode

a = APyFixed.from_float(3.5, int_bits=4, frac_bits=1)
b = APyFixed(0b00_111, bits=5, int_bits=2) # 7 / 2**(5-2) = 0.875

# Word lengths are statically increased based on operation
c = a + b # APyFixed(35, bits=8, int_bits=5) = 4.375
d = a * b # APyFixed(49, bits=10, int_bits=6) = 3.0625

# Quantization is done explicitly
e = d.cast(bits=7, int_bits=4,
        quantization=QuantizationMode.RND,
        overflow=OverflowMode.SAT)
```

Scalar classes

Floating-Point

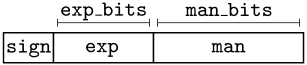
- Format defined by number of exponent bits and mantissa bits, and a bias.



Scalar classes

Floating-Point

- Format defined by number of exponent bits and mantissa bits, and a bias.



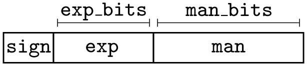
- A number x is represented as the triplet $(\texttt{sign}, \texttt{exp}, \texttt{man})$, where

$$x = (-1)^{\texttt{sign}} \times 2^{\texttt{exp}-\texttt{bias}} \times (1 + \texttt{man} \times 2^{\texttt{-man\_bits}}).$$

Scalar classes

Floating-Point

- Format defined by number of exponent bits and mantissa bits, and a bias.



- A number x is represented as the triplet $(\texttt{sign}, \texttt{exp}, \texttt{man})$, where

$$x = (-1)^{\texttt{sign}} \times 2^{\texttt{exp}-\texttt{bias}} \times (1 + \texttt{man} \times 2^{\texttt{-man\_bits}}).$$

- Generalization of the IEEE-754 standard, default $\texttt{bias} = 2^{\texttt{exp\_bits}-1} - 1$.

Scalar classes

Floating-Point

```
from apytypes import APyFloat
# x = 2.5
x = APyFloat.from_float(2.5, exp_bits=3, man_bits=4)
```

Scalar classes

Floating-Point

```
from apytypes import APyFloat
# x = 2.5
x = APyFloat.from_float(2.5, exp_bits=3, man_bits=4)

# y = 2.125, z = -1.75
y = APyFloat.from_bits(0b0_100_0001, exp_bits=3, man_bits=4)
z = APyFloat(sign=1, exp=15, man=3, exp_bits=5, man_bits=2)
```

Scalar classes

Floating-Point

```
from apytypes import APyFloat
# x = 2.5
x = APyFloat.from_float(2.5, exp_bits=3, man_bits=4)

# y = 2.125, z = -1.75
y = APyFloat.from_bits(0b0_100_0001, exp_bits=3, man_bits=4)
z = APyFloat(sign=1, exp=15, man=3, exp_bits=5, man_bits=2)

# APyFloat(sign=0, exp=5, man=2, exp_bits=3, man_bits=4)
v = x + y # 4.5
```

Scalar classes

Floating-Point

```
from apytypes import APyFloat
# x = 2.5
x = APyFloat.from_float(2.5, exp_bits=3, man_bits=4)

# y = 2.125, z = -1.75
y = APyFloat.from_bits(0b0_100_0001, exp_bits=3, man_bits=4)
z = APyFloat(sign=1, exp=15, man=3, exp_bits=5, man_bits=2)

# APyFloat(sign=0, exp=5, man=2, exp_bits=3, man_bits=4)
v = x + y # 4.5

# APyFloat(sign=1, exp=17, man=2, exp_bits=5, man_bits=4)
w = x * z # -4.5
```

Array types

Array types

```
from apytypes import APyFloatArray

# Array definition and operations
A = APyFloatArray.from_float( # (2, 2)-array
        [[1., 1.25],[4.5, 9.]], exp_bits=5, man_bits=7)
b = APyFloatArray.from_float( # From NumPy (2,)-array
        np.asarray([3.5, 7.]), exp_bits=5, man_bits=7)
```

Array types

```
from apytypes import APyFloatArray

# Array definition and operations
A = APyFloatArray.from_float( # (2, 2)-array
        [[1., 1.25],[4.5, 9.]], exp_bits=5, man_bits=7)
b = APyFloatArray.from_float( # From NumPy (2,)-array
        np.asarray([3.5, 7.]), exp_bits=5, man_bits=7)

# Matrix multiplication
C = A @ b.T # (2,)-array
# Mixed array and scalar operations
D = C * w
# Conversion to NumPy array
E = D.to_numpy()
```

Context handling

Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode
```

Context handling

Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Addition rounds to nearest, ties to even
a = x + y
```

Context handling

Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Addition rounds to nearest, ties to even
a = x + y

with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    # Calculations with quantization towards negative infinity
    b = x + y
    C = X @ Y
```

Context handling

Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Addition rounds to nearest, ties to even
a = x + y

with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    # Calculations with quantization towards negative infinity
    b = x + y
    C = X @ Y
```

- Contexts allow for fine-grained control

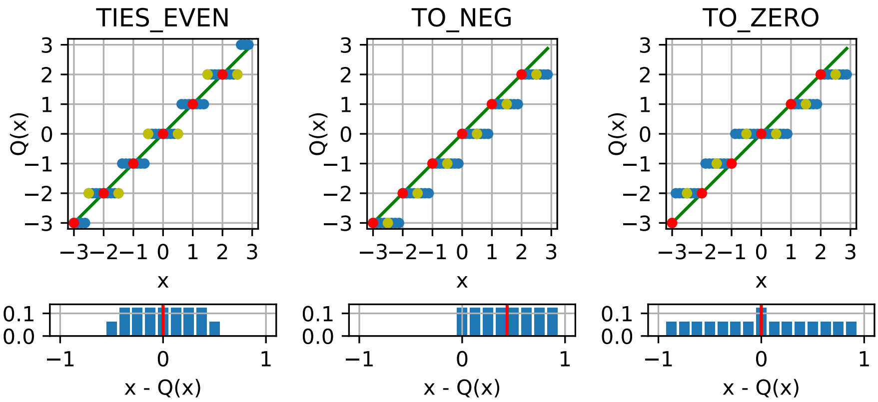
Context handling

Quantizations

```python
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Addition rounds to nearest, ties to even
a = x + y

with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    # Calculations with quantization towards negative infinity
    b = x + y
    C = X @ Y
```

- Contexts allow for fine-grained control
- Currently 15 different quantization modes

Quantization modes



Context handling

Accumulators

```
from apytypes import APyFixedArray, APyFixedAccumulatorContext
import numpy as np
```

Context handling

Accumulators

```
from apytypes import APyFixedArray, APyFixedAccumulatorContext
import numpy as np

# Fixed-point matrix (100, 100) of random data
A = APyFixedArray.from_float(
        np.random.normal(1, 2, size=(100, 100)), bits=10, int_bits=3)
# Fixed-point vector of random data
b = APyFixedArray.from_float(
        np.random.uniform(0, 1, size=100), int_bits=4, frac_bits=5)
```

Context handling

Accumulators

```
from apytypes import APyFixedArray, APyFixedAccumulatorContext
import numpy as np

# Fixed-point matrix (100, 100) of random data
A = APyFixedArray.from_float(
        np.random.normal(1, 2, size=(100, 100)), bits=10, int_bits=3)
# Fixed-point vector of random data
b = APyFixedArray.from_float(
        np.random.uniform(0, 1, size=100), int_bits=4, frac_bits=5)

# Multiplication using using narrow accumulator
with APyFixedAccumulatorContext(frac_bits=9):
        d = A @ b.T
```

Integration with Python ecosystem

- Conversion to and from NumPy-arrays
- Direct plotting using Matplotlib 3.6 and later
- Extended LaTeX-based representations, for e.g. Jupyter Notebook and Spyder:

APyFixed: $\frac{35}{2^3} = 4.375$

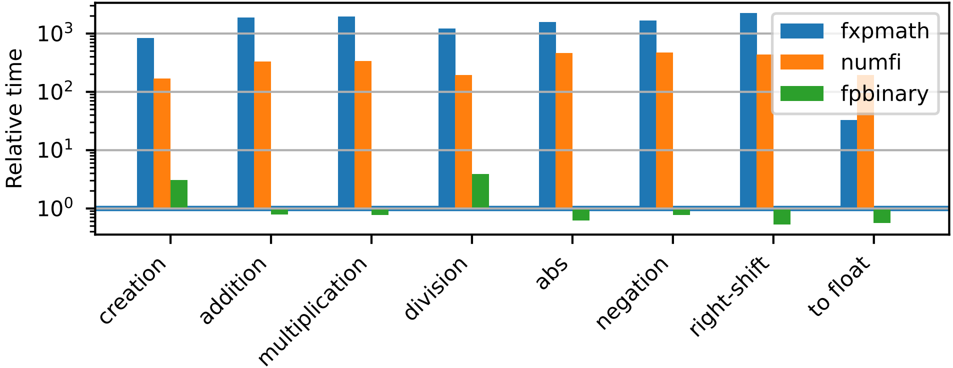
APyFloat: $\left(1 + \frac{9}{2^4}\right) 2^{18-15} = 25 \times 2^{-1} = 12.5$

APyTypes implementation

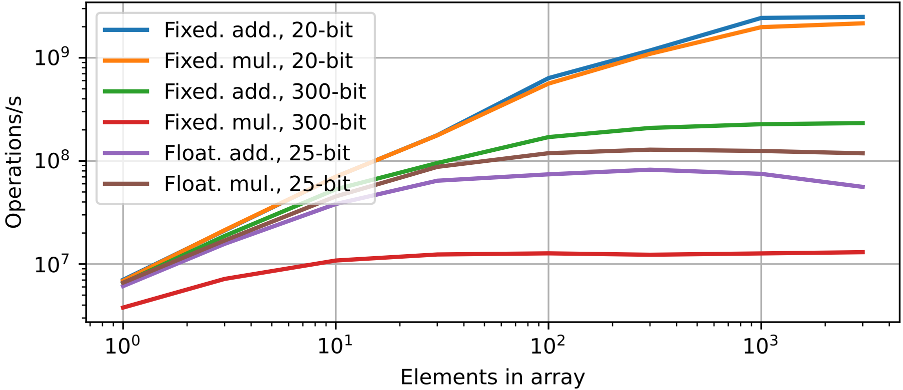
- Backend written in performance aware C++.
- Cross-platform, continuously tested on Linux, MacOS, and Windows.
- Python bindings using nanobind.
- Leverages native SIMD features using Google Highway.

Performance compared to other libraries

Performance relative to APyTypes – Fixed-point scalars



Performance – scaling of arrays



Future plans

- More floating-point formats.
- Explicit unsigned fixed-point numbers.
- Generation of test and verification data.
- Support for more of the NumPy mathematical functions.

Want to help out?

If you are interested in helping with the apytypes project:

- The absolute best way to help out is to use the library:
  `pip install apytypes`
- Submit new issues:
  `https://github.com/apytypes/apytypes`
- Contribute by making pull-requests.

Mikael Henriksson, Theodor Lindberg, and Oscar Gustafsson

https://github.com/apytypes/apytypes