APyTypes

Flexible Fixed- and Floating-Point Types for Word Length Simulation in Python

Theodor Lindberg, Mikael Henriksson, and Oscar Gustafsson

LINKÖPING UNIVERSITY

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.)

Scalar Classes

Fixed-Point

- Characterized by the number of bits before and after the binary point

$$\underbrace{\underbrace{x_{n-1}\ x_{n-2}\ \cdots\ x_{k+1}\ x_k}_{\text{int\_bits}} \cdot \underbrace{x_{k-1}\ x_{k-2}\ \cdots\ x_1\ x_0}_{\text{frac\_bits}}}_{\text{bits}} \tag{1}$$

Scalar Classes

Fixed-Point

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

Scalar Classes

Fixed-Point

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

a = APyFixed.from_float(3.5, int_bits=4, frac_bits=1)
b = APyFixed(0b00111, 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(0b00111, bits=5, int_bits=2) # 7 / 2**(5-2) = 0.875

# Word lengths are increased to accomodate the results
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(0b00111, bits=5, int_bits=2) # 7 / 2**(5-2) = 0.875

# Word lengths are increased to accomodate the results
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 the number of exponent and mantissa bits, and bias

Scalar Classes

Floating-Point

- Format defined by the number of exponent and mantissa bits, and bias
- Based on the IEEE-754 standard, default $\texttt{bias} = 2^{\texttt{exp\_bits}-1} - 1$

Scalar Classes

Floating-Point

- Format defined by the number of exponent and mantissa bits, and bias
- Based on the IEEE-754 standard, default $\texttt{bias} = 2^{\texttt{exp\_bits}-1} - 1$
- A number x is represented as the triplet $(\texttt{sign}, \texttt{exp}, \texttt{man})$, where

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

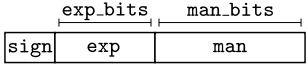


Figure: Binary format of a floating-point number.

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

```python
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Calculation with quantization towards negative infinity
with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    z = x + y
```

Context Handling

Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Calculation with quantization towards negative infinity
with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    z = x + y
```

- Contexts allow for fine-grained control

Context Handling

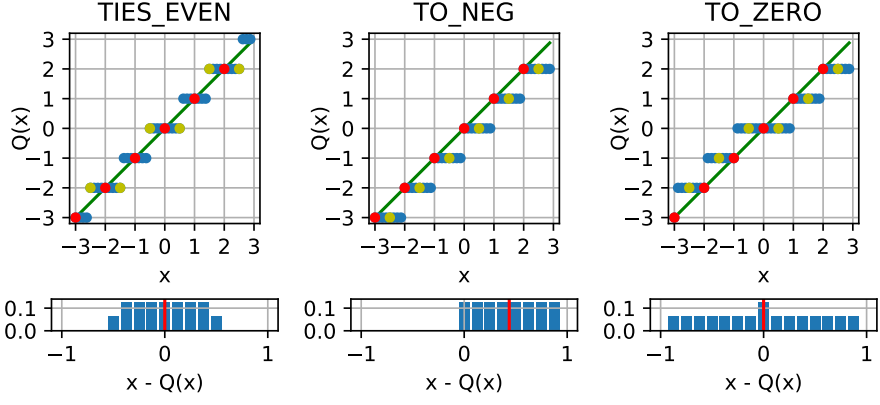
Quantizations

```
from apytypes import APyFloatQuantizationContext, QuantizationMode

# Calculation with quantization towards negative infinity
with APyFloatQuantizationContext(QuantizationMode.TO_NEG):
    z = 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(int_bits=14, frac_bits=9):
        d = A @ b.T
```

Integration with the 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:

$$\text{(fixed-point)} \quad \frac{35}{2^3} = 4.375 \tag{3}$$

$$\text{(floating-point)} \quad \left(1 + \frac{9}{2^4}\right) 2^{18-15} = 25 \times 2^{-1} = 12.5 \tag{4}$$

Implementation

- Cross-platform
- Backend written in C++
- Python-wrapper using nanobind
- Leverages native SIMD features

What About Other Libraries?

Features – Configurable fixed-point libraries

Library[1]	**APyTypes**	fpbinary	fxpmath	numfi[2]
Arrays	Yes	No	Yes	Yes
Matrix mul.	Yes	N/A	No	No
Floating-Point	Yes	No	No	No
Written in	C++/Python	C/Python	Python	Python

[1] `https://apytypes.github.io/apytypes/comparison.html`
[2] numfi is limited to 64 bits in total, including results from multiplication

What About Other Libraries?

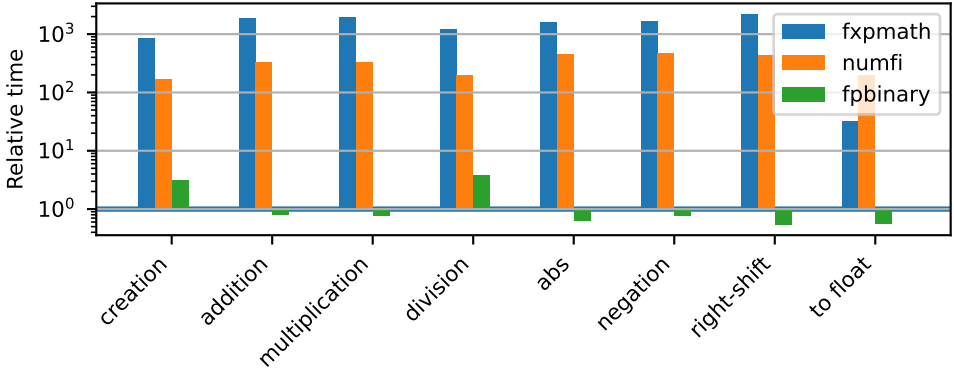
Features – Configurable floating-point libraries

Library	**APyTypes**[3]	gmpy2
Arrays	Yes	No
Matrix mul.	Yes	N/A
Fixed-point	Yes	No
Written in	C++/Python	C/Python

[3] Limited to 32-bit exponent and 64-bit mantissa by design

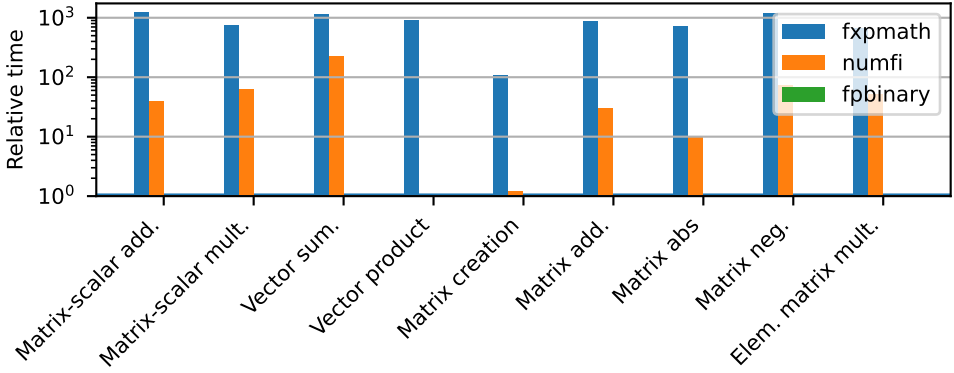
What About Other Python Libraries?

Performance relative to APyTypes – Fixed-point scalars

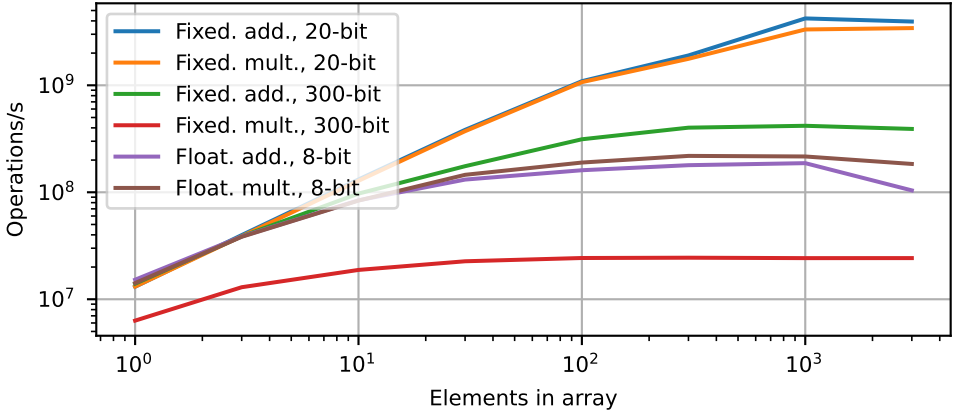


What About Other Python Libraries?

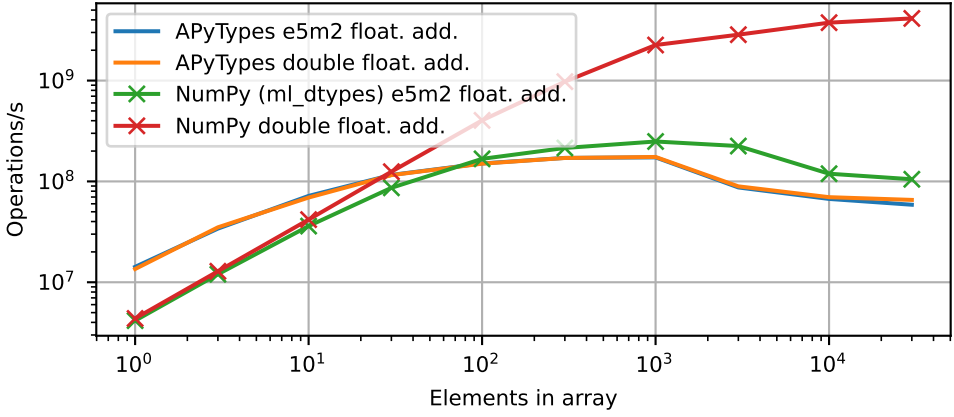
Performance – Fixed-point arrays



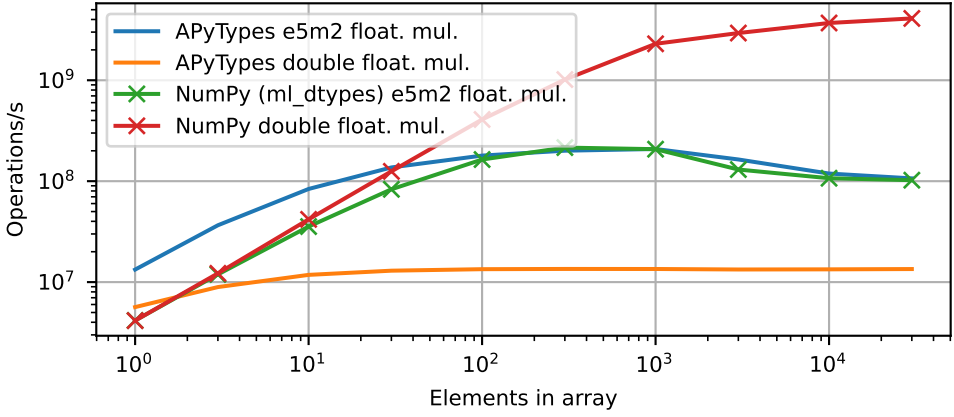
Performance – Scaling of Arrays



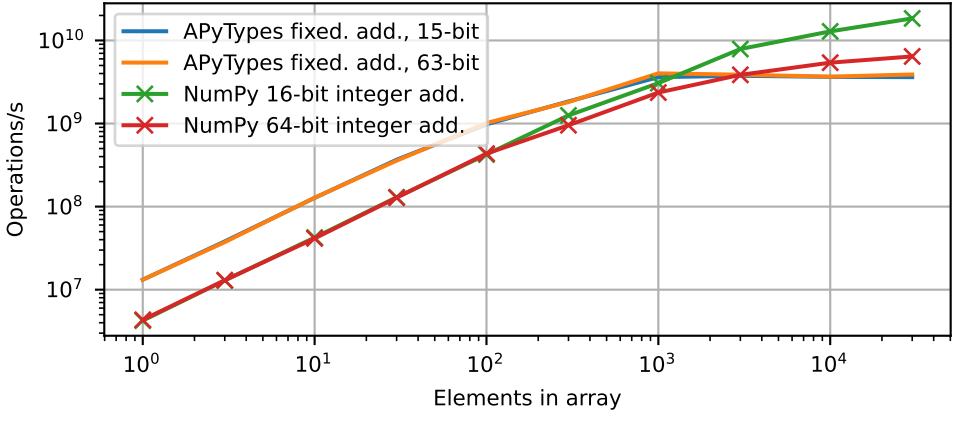
Comparing Performance with NumPy – Floating-Point Addition



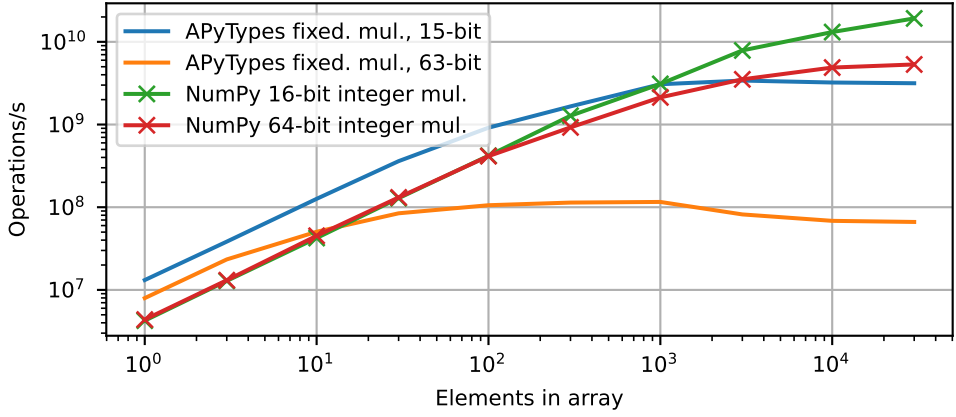
Comparing Performance with NumPy – Floating-Point Multiplication



Comparing Performance with NumPy – Fixed-Point Addition



Comparing Performance with NumPy – Fixed-Point Multiplication



Future Plans

- Test against other libraries
- Variants of floating-point formats
- Explicit unsigned fixed-point numbers
- Generation of test and verification data
- Support some of the NumPy mathematical functions

Theodor Lindberg, Mikael Henriksson, and Oscar Gustafsson

https://github.com/apytypes/apytypes

LINKÖPING UNIVERSITY