PyHEP 2022 - Using C++ From Numba, Fast and Automatic

Authors: **Baidyanath Kundu** (*Princeton University*); **Vassil Vasilev** (*Princeton University*); **Wim Lavrijsen** (*Lawrence Berkeley National Lab.*)

Introduction

PyROOT has enabled the use of ROOT-based data models of various HEP experiments in Python. While this has enabled users to benefit from Python functionalities and libraries, use of loops and other native Python features is slow.

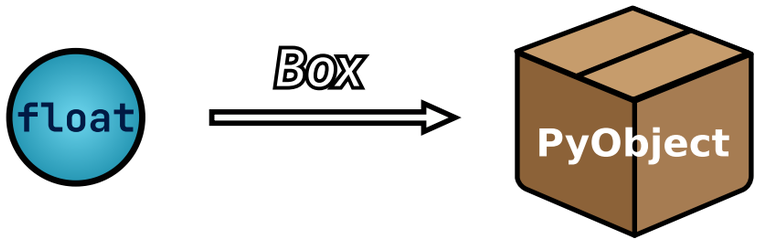
Numba is a package that allows numerically heavy programs to be written in Python without skimming on execution speeds. It translates Python functions into optimized machine code at runtime using the industry-standard LLVM compiler framework. However, PyROOT objects were not supported by Numba.

This led us to combine the two and create an extension for PyROOT that enables the use of PyROOT objects inside Numba JITted functions. This extension allows Numba to determine the type of PyROOT objects and efficiently JIT functions, converting them into machine code.

Numba: Tradeoff between flexibility and performance

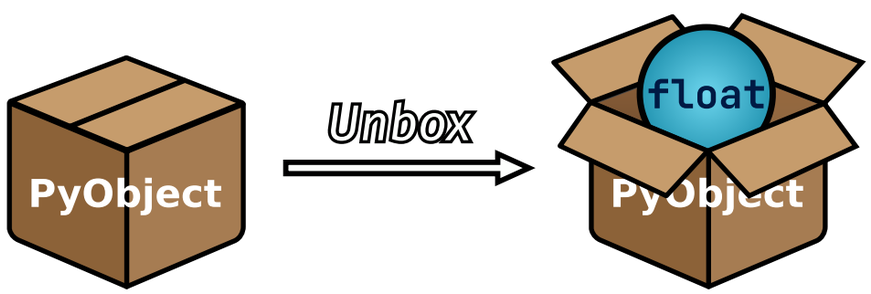
Python has the flexibility of converting easily between different data types. This is because each Python object is a PyObject that can represent any datatype that is used in Python.

```
f = 0.5
```



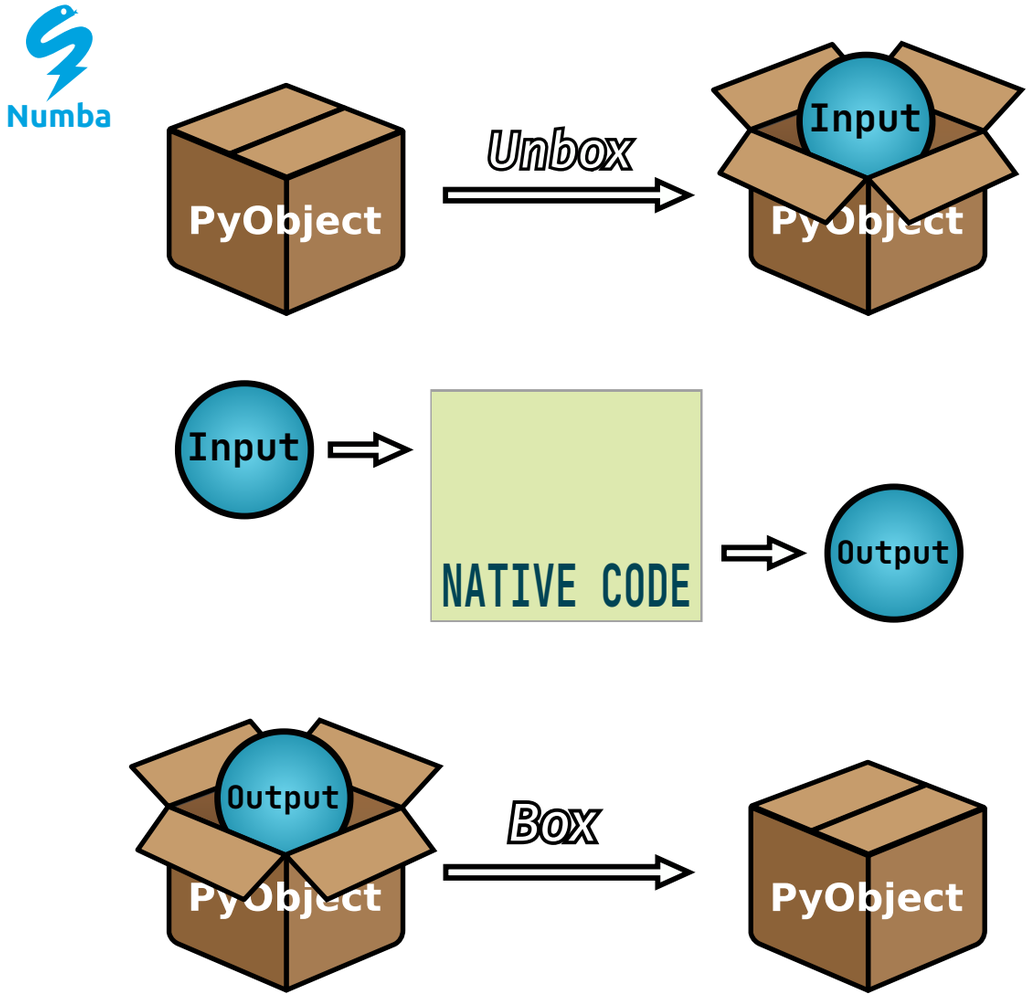
So when you store a floating point number in a variable in Python. Python first converts it to a Pyobject, which is called boxing, and then the pointer to this PyObject is what the variable stores. Whenever the native value, the floating point number in this case, is required for any calculations it needs to be unboxed from the PyObject and then used for calculations.

```
f = f ** 2
```



These boxing and unboxing operations are detrimental to performance but provide the necessary flexibility for Python duck typing.

Numba on the other hand gets rid of this flexibility for performance. It unboxes the inputs of the function and the whole function is run on native values and not PyObjects. At the end the output is boxed so that Python can use it. For this to work Numba needs to figure out the types of not only the input and output but the intermediate variables as well.



The drawback to this approach is that if the types in the program are not determinable the speed up will be minimal.

Performance benefits from Numba

To measure the performance benefits from Numba we use a math heavy function shown below. This function calculates the `tanh` of the trace of a matrix and adds it back to the whole matrix. **The only difference between the two is the Numba decorator on Line 15.** This decorator instructs Numba to compile the function into native code.

In [ ]:
```
from numba import jit
import numpy as np
import time

################ Pure Python ###############
# Function is not compiled and runs in byte code
def python_trace(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

################ Numba ###############
# Function is compiled and runs in machine code
@jit(nopython=True) # <--------------- Numba decorator
def numba_trace(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace
```

In [ ]:
```
def measure_execution(func, args):
    start = time.perf_counter()
    results = func(*args)
    end = time.perf_counter()
    elapsed = end - start
    return elapsed, results
```

```
x = np.random.rand(100,100)

numba_warmup, _ = measure_execution(numba_trace, (x,))
python_elapsed, _ = measure_execution(python_trace, (x,))
numba_elapsed, _ = measure_execution(numba_trace, (x,))
```

In [ ]:
```
print(f"Numba Function : Warmup  = {numba_warmup:.7f}s")
print(f"Python Function: Elapsed = {python_elapsed:.7f}s")
print(f"Numba Function : Elapsed = {numba_elapsed:.7f}s")
```

```
Numba Function : Warmup  = 0.4006952s
Python Function: Elapsed = 0.0001682s
Numba Function : Elapsed = 0.0000124s
```

These results show that during warmup Numba takes a lot more time than a conventional Python function. This is because it takes time to load necessary modules and compile the function into machine code whereas the conventional Python function execution does not have to go through these extra steps. After the compilation is done Numba can provide a speedup of one or two orders of magnitude depending on the program itself. Thus Numba is used when the same function has to be run a lot of times.

Cppyy

Cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++. It is in the core of PyROOT, which is why the extension was originally developed with Cppyy and later ported to PyROOT. Since the extension may be used with both Cppyy and PyROOT, we will discuss how to do so.

Benefits of using Cppyy/PyROOT with Numba?

1. **Numba makes loops fast:** When using Cppyy/PyROOT with Python, the loops in Python are slower as compared to languages such as C/C++. Numba alleviates this problem and can make it as fast as C without much code instrumentation.

2. **Code completely in python:** This makes debugging easier. To debug Numba instrumented code you can either comment out the instrumentation line and debug the code as you would do in Python or use gdb using `numba.gdb`. Numba also has a variety of flags that can be turned on to see tracebacks and the intermediate steps taken by Numba. *This is easier than to debug a code that is setup in Python and uses RDF for hotspots.*

3. **No conversions in the machine code:** Cppyy can be converted to machine code cleanly, that is no boxing and unboxing is required, so we do not spend any time in type conversions and gain the maximum amount of speedup possible.

4. **Two worlds closer together:** You can switch easily between C++ and Python as and when you want.

Performance

Similar to the tanh example used to compare Numba vs Python we use the `std::tanh` from C++ to compare the performance against Numba. We just replace the `np.tanh` function and no extra changes are done.

In [ ]:
```
import cppyy
import cppyy.numba_ext # <------ Imports the extension

################ Cppyy ###############
# Function is compiled and runs in machine code
@jit(nopython=True)
def cppyy_numba_trace(a):
    trace = 0.0
    for i in range(a.shape[0]):
```

```
        trace += cppyy.gbl.tanh(a[i, i]) # <---------------- Replaces np.tanh
    return a + trace
```

In [ ]:
```
cppyy_warmup, _ = measure_execution(cppyy_numba_trace, (x,))
cppyy_elapsed, _ = measure_execution(cppyy_numba_trace, (x,))

print(f"Numba Function : Warmup  = {numba_warmup:.7f}s")
print(f"Cppyy Function : Warmup  = {cppyy_warmup:.7f}s")
print()
print(f"Python Function: Elapsed = {python_elapsed:.7f}s")
print(f"Numba Function : Elapsed = {numba_elapsed:.7f}s")
print(f"Cppyy Function : Elapsed = {cppyy_elapsed:.7f}s")
```

```
Numba Function : Warmup  = 0.4006952s
Cppyy Function : Warmup  = 0.1550883s

Python Function: Elapsed = 0.0001682s
Numba Function : Elapsed = 0.0000124s
Cppyy Function : Elapsed = 0.0000142s
```

The result show that overhead for using Cppyy in a Numba function is minimal as the time elapsed is almost similar to the Numba only function.

Features provided by the extension

1) Plug and Play

To use the extension you just need to import `cppyy.numba_ext` and then you can use C++ functions in Numba directly.

In the example shown below `sqrt` is a C++ function that can be used directly inside the Numba jitted function with the help of the extension.

In [ ]:
```
import numba
import cppyy
import cppyy.numba_ext   # <------- Imports the necessary information for numba to work with cppyy
import math

@numba.jit(nopython=True)
def cpp_sqrt(x):
    return cppyy.gbl.sqrt(x) # <------------ Direct use, no extra setup required

print("Sqrt of 4: ", cpp_sqrt(4.0))
print("Sqrt of Pi: ", cpp_sqrt(math.pi))
```

```
Sqrt of 4:  2.0
Sqrt of Pi:  1.7724538509055159
```

2) Template instantiation

Cppyy supports template instantiation which gives you access to an important feature set in C++ that is used abundently in lot of codebases. This extension extends that support to Numba too so any templated C++ function can be used in Numba. Below we have a templated square function and depending on the type of the matrix the extension instantiates the required template argument.

In [ ]:
```
import cppyy
import cppyy.numba_ext
import numba
import numpy as np

cppyy.cppdef("""
template<typename T>
```

```
T square(T t) { return t*t; }
""")

@numba.jit(nopython=True)
def tsa(a):
    total = type(a[0])(0)
    for i in range(len(a)):
        total += cppyy.gbl.square(a[i])
    return total

a = np.array(range(10), dtype=np.float32)
print("Float array: ", a)
print("Sum of squares: ", tsa(a))
print()
a = np.array(range(10), dtype=np.int32)
print("Integer array: ", a)
print("Sum of squares: ", tsa(a))
```

```
Float array:  [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Sum of squares:  285.0

Integer array:  [0 1 2 3 4 5 6 7 8 9]
Sum of squares:  285
```

3) Overload selection

Similar to template instantiation the extension will select the appropriate overload based on the type of the input provided to the function.

In [ ]:
```
cppyy.cppdef("""
int mul(int x) { return x * 2; }
float mul(float x) { return x * 3; }
```

```
""")

@numba.jit(nopython=True)
def oversel(a):
    total = type(a[0])(0)
    for i in range(len(a)):
        total += cppyy.gbl.mul(a[i])
    return total

a = np.array(range(10), dtype=np.float32)
print("Array: ", a)
print("Overload selection output: ", oversel(a))

a = np.array(range(10), dtype=np.int32)
print("Array: ", a)
print("Overload selection output: ", oversel(a))
```

```
Array:  [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Overload selection output:  135.0
Array:  [0 1 2 3 4 5 6 7 8 9]
Overload selection output:  90
```

Demos

1) Numba physics example

Taken from: https://github.com/numba/numba-examples/blob/master/examples/physics/lennard_jones/numba_scalar_impl.py

$$V_{LJ}(r) = 4\varepsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$$

$$\varepsilon = 1, \sigma = 1$$

In [ ]:

```
import numba
import cppyy
import cppyy.numba_ext

cppyy.cppdef("""
#include <vector>

struct Atom {
    float x;
    float y;
    float z;
};

std::vector<Atom> atoms = {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}, {5, 6, 7}};
""")

@numba.njit
def lj_numba_scalar(r):
    sr6 = (1./r)**6
    pot = 4.*(sr6*sr6 - sr6)
    return pot

@numba.njit
def distance_numba_scalar(atom1, atom2):
    dx = atom2.x - atom1.x
    dy = atom2.y - atom1.y
    dz = atom2.z - atom1.z

    r = (dx * dx + dy * dy + dz * dz) ** 0.5

    return r
```

```
def potential_numba_scalar(cluster):
    energy = 0.0
    for i in range(cluster.size() - 1):
        for j in range(i + 1, cluster.size()):
            r = distance_numba_scalar(cluster[i], cluster[j])
            e = lj_numba_scalar(r)
            energy += e

    return energy

print("Total lennard jones potential =", potential_numba_scalar(cppyy.gbl.atoms))
```

```
Total lennard jones potential = -0.5780277345740283
```

2) Using the extension with PyROOT

To use the extension with PyROOT, just as we do with Cppyy, we need to **import `cppyy.numba_ext`**.

In the example we use the TLorentzVector class from ROOT. It has with four properties: `Px`, `Py`, `Pz` and `E`

It also provides the transverse momentum `Pt` to the user which can be calculated by:

$$Pt = \sqrt{Px^2 + Py^2}$$

In [ ]:
```
######################################## Setup Code ################################
import numba
import math
import ROOT
import cppyy.numba_ext # <---------- Import the Numba extension
import time
```

```
ROOT.gInterpreter.Declare("""
std::vector<TLorentzVector> vec_lv;

const int no_of_samples = 100;

void fill() {
  vec_lv.reserve(no_of_samples);
  TRandom3 R(111);

  for (int i = 0; i < no_of_samples; ++i) {
    double Px = R.Gaus(0,10);
    double Py = R.Gaus(0,10);
    double Pz = R.Gaus(0,10);
    double E  = R.Gaus(100,10);
    vec_lv.push_back(TLorentzVector(Px, Py, Pz, E));
  }
}
""")
ROOT.gInterpreter.ProcessLine("""
fill();
""")

print()
```

```
Welcome to JupyROOT 6.27/01
```

In this example we calculate the same using Python and show how we can speed up the calculation using Numba. The `calc_pt` function uses pure Python to calculate `Pt` whereas the `numba_calc_pt` uses Numba to do the same. As before the only **difference between the two is the `numba.njit` decorator** so you do not need to change anything.

```
```

In [ ]:
```
def calc_pt(lv):
    return math.sqrt(lv.Px() ** 2 + lv.Py() ** 2)

def calc_pt_vec(vec_lv):
    pt = []
    for i in range(vec_lv.size()):
        pt.append((calc_pt(vec_lv[i]), vec_lv[i].Pt()))
    return pt


@numba.njit
def numba_calc_pt(lv):
    return math.sqrt(lv.Px() ** 2 + lv.Py() ** 2)

def numba_calc_pt_vec(vec_lv):
    pt = []
    for i in range(vec_lv.size()):
        pt.append((numba_calc_pt(vec_lv[i]), vec_lv[i].Pt()))
    return pt
```

In [ ]:
```
numba_warmup, _ = measure_execution(numba_calc_pt, (ROOT.vec_lv[0], ))
python_elapsed, _ = measure_execution(calc_pt_vec, (ROOT.vec_lv, ))
numba_elapsed, pt = measure_execution(numba_calc_pt_vec, (ROOT.vec_lv, ))

print(f"Numba'd    : Warmup  = {numba_warmup  :.5f}s")
print(f"Pure Python: Elapsed = {python_elapsed:.5f}s")
print(f"Numba'd    : Elapsed = {numba_elapsed :.5f}s")

print(f"Speedup              = {python_elapsed / numba_elapsed:.5f}x")

no_of_samples = 3
print("\nCalc pT \tActual pT")
print("---------------------------")
```

```
print(*(f"{x:2.5f} \t{y:2.5f}" for x,y in pt[:no_of_samples]), sep="\n")

if False in tuple(x==y for x, y in pt):
    print("\nSome values do not match")
else:
    print("\nAll values for pT match")
```

```
Numba'd    : Warmup  = 0.04820s
Pure Python: Elapsed = 0.00813s
Numba'd    : Elapsed = 0.00037s
Speedup              = 22.21831x

Calc pT         Actual pT
---------------------------
8.95222         8.95222
4.11973         4.11973
25.97929        25.97929

All values for pT match
```

3) RDF

You can also use it inside RDF through `ROOT.Numba.Declare`. Underneath is a simple example where it is used to calculate the power function.

In [ ]:
```
import numba
import ROOT
import cppyy.numba_ext # <----- Import extension

ROOT.gInterpreter.Declare("""
double cpppow(double x, int y) { return pow(x, y); }
""")
```

```
@ROOT.Numba.Declare(['double', 'int'], 'double')
def pypownd(x, y):
    return ROOT.cpppow(x, y) # <--------- Numba.Declare supports PyROOT due to the Numba extension


ROOT.gInterpreter.ProcessLine("""
cout << "2^3 = " << Numba::pypownd(2, 3) << endl
     << "4^5 = " << Numba::pypownd(4, 5) << endl;""")
print()

# Or we can use the callable as well within a RDataFrame workflow.
data = ROOT.RDataFrame(4).Define('x', '(float)rdfentry_')\
                         .Define('x_pow3', 'Numba::pypownd(x, 3)')\
                         .AsNumpy()

print('pypownd({}, 3) = {}'.format(data['x'], data['x_pow3']))
```

```
pypownd([0. 1. 2. 3.], 3) = [ 0.  1.  8. 27.]
2^3 = 8
4^5 = 1024
```

Future work

1. **Complete C++ feature support**:

Currently the extension allows Numba to determine the type of PyROOT functions that return primitive values. We are actively working towards supporting returning of complex objects and reference types. Along with this we also want to support:

- Memory management
- Constructor support

- Virtual inheritance

2. **Inlining**:

For the code:

```
def numba_calc_pt(lv):
    return math.sqrt(lv.Px() ** 2 + lv.Py() ** 2)
```

The machine code generated involves calls to functions `Px()`, `Py()`. This can be optimized away by replacing it with memory accesses thus making the calculations faster.

3. **GPU support**: Numba supports running Python code on CUDA but the extension currently doesn't allow that with PyROOT objects. We want to allow Numba to lower PyROOT objects into GPU in the future.

4. **Automatic parallelization**: We want to support automatic parallelization using OpenMP instructions at the machine code level.

Conclusion

1) The extension enables you to use PyROOT with Numba.

This gives physicists a different way to carry out their analysis.

2) It is easy to use

Just importing `cppyy.numba_ext` allows you to use C++ in Numba.

3) Makes Python code faster while keeping debugging easy.

All debugging features available to Numba users are still available with the extension enabled and it makes it easy for the user to debug Numba code.

How to use?

The PR for the PyROOT entension is open, and will soon be merged in ROOT master. Until then, to **try out this notebook or the extension checkout the repo: https://github.com/sudo-panda/PyHEP-2022**

Thankyou
