Extending the Cppyy support in Numba

Progress Update

Aaron Jomy

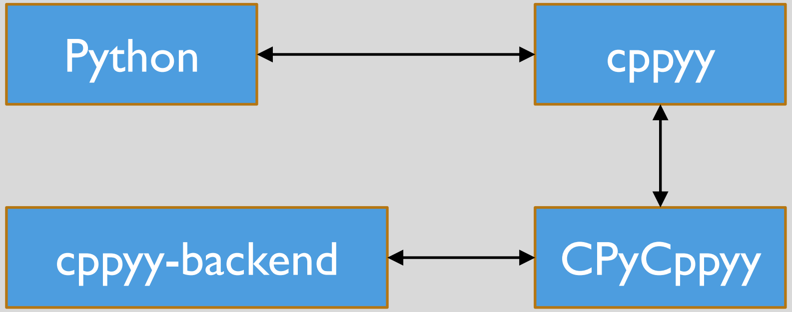
cppyy

Numba

RECAP

RECAP

- **Cppyy**:
  An automatic, run-time, Python-C++ bindings generator
- **Cling**
  is used in Cppyy's backend since an interactive C++ interpreter provides a runtime exec approach to C++ code

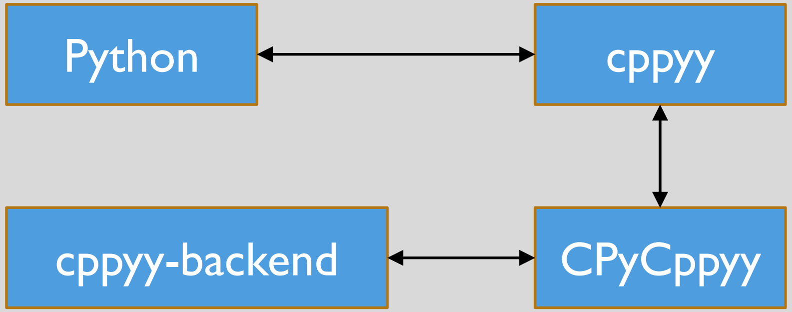



```
cppyy.cppdef(r"""\
struct MyNumbaData03 {
    MyNumbaData03(int64_t i1, int64_t i2) : fField1(i1), fField2(i2) {}
    int64_t fField1;
    int64_t fField2;
};""")
```

```
@numba.jit(nopython=True)
def go_fast(a, d):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += d.fField1 + d.fField2
    return a + trace
```

INTRODUCTION

- **Cppyy**:
  An automatic, run-time, Python-C++ bindings generator
- **Cling**
  is used in backend since an interactive C++ interpreter provides a runtime exec approach to C++ code



WHY USE NUMBA?

- **Numba**
  JIT compiler that translates Python and NumPy code into fast machine code.
- The compute time overhead while switching between languages accumulates in loops with cppyy objects.

```
def go_slow(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += cppyy.gbl.tanh(a[i, i])
    return a + trace

@numba.njit
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += cppyy.gbl.tanh(a[i, i])
    return a + trace
```

- Numba optimizes the loop and compiles it into machine code which crosses the language barrier only once

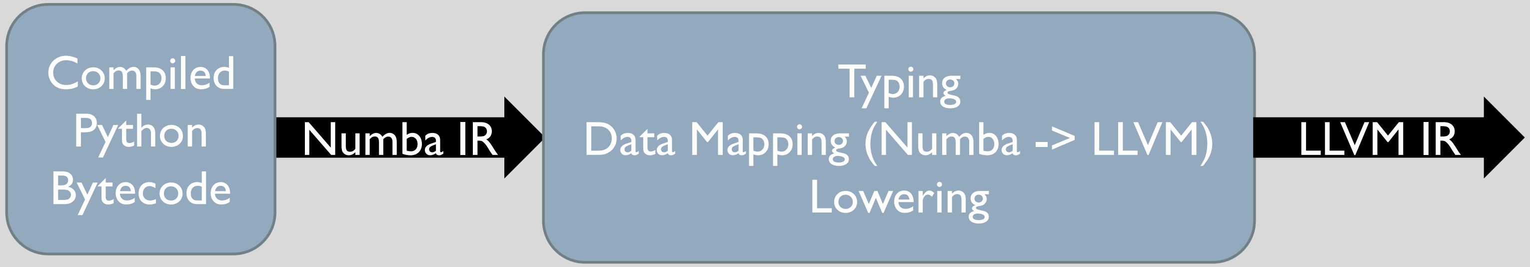
```
x = np.arange(10000, dtype=np.float64).reshape(100, 100)
run_jit_test(x)

✓ 0.1s

Numba disabled = 0.10824203491210938 ms
Numba typeinfer in dispatcher: array(float64, 2d, C)
Numba njit enabled = 0.007867813110351562 ms
```

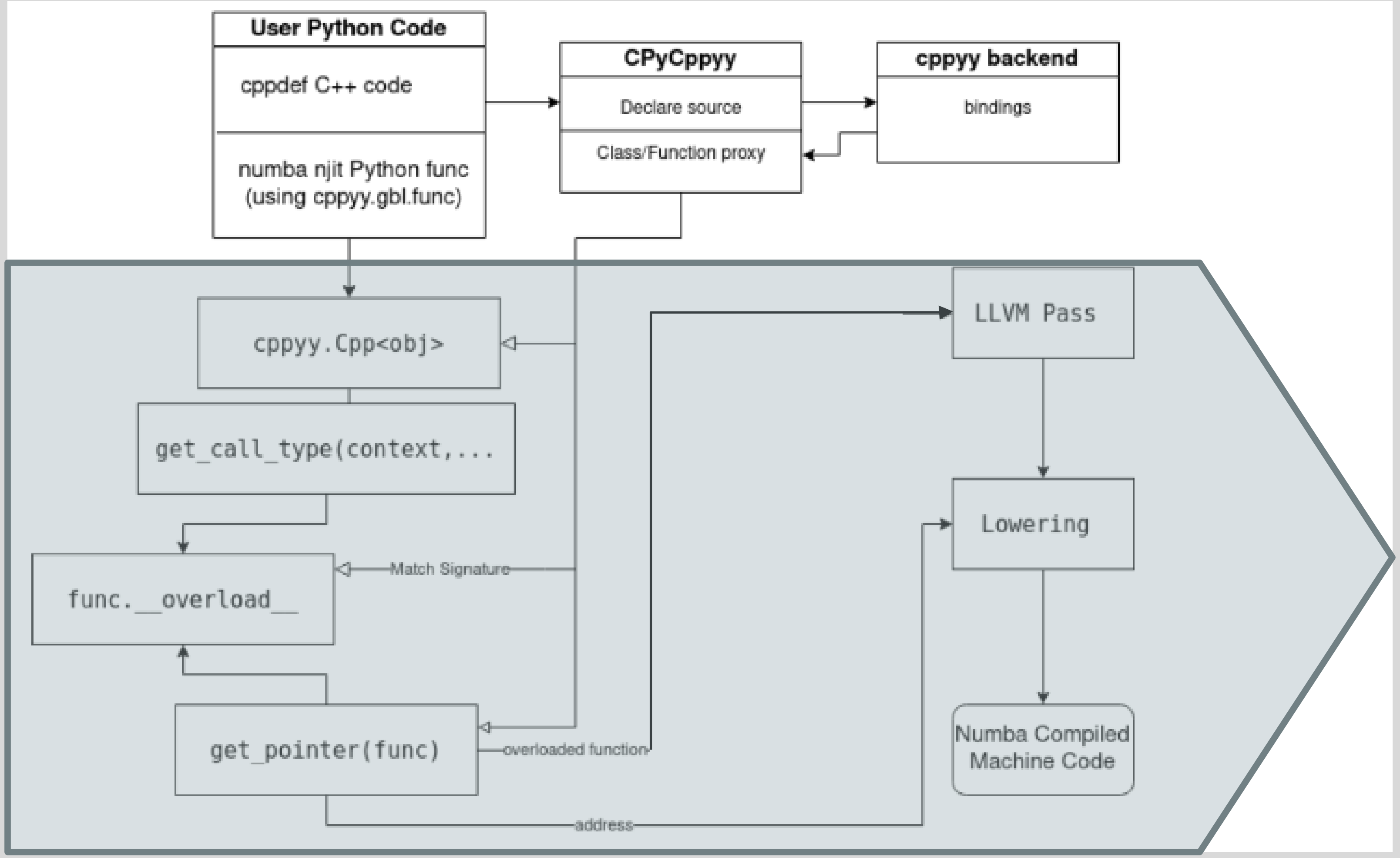
NUMBA PIPELINE

- **Typing**
  Numba core has a type inference algorithm which assigns a nb_type for a variable
- **Lowering**
  Numba lowers high-level Python operations into low-level LLVM code.
  Exploits typing to map to LLVM type
- **Boxing and unboxing**
  convert PyObject* 's into native values, and vice-versa.



We utilise the runtime numba compilation process to lower C++ code cppdef'ed in Python
How? →

NUMBA LOW LEVEL EXTENSION API



CPPYY NUMBA SUPPORT
cppyy/numba_ext.py

STATUS

Currently, the following functionality has been added to Cppyy's numba extension:

- Extended typing and non type template definition support [Test 9]
- nJIT function pointers to C++ functions that return a reference type [Test 10]
- nJIT support for pointers and reference types to builtins and std::vectors [Test 11- 13]
- Successful typing and overload matching for Eigen templated classes [Test 14]

POINTER AND REFERENCE TYPE SUPPORT

- The Numba extension now supports njitting ref types, const refs and pointers to C++ methods/functions
- The results are reflected directly on the python side using the ctypes interface that provides a "pointer-like" behaviour that can be emulated in Python

```
cppyy.cppdef("""
  namespace RefTest {
    class Box{
      public:
        long a;
        long *b;
        long *c;
        Box(long i, long& j, long& k){
        a = i;
        b = &j;
        c = &k;
        }

        void swap_ref(long &a, long &b) {
            long temp = a;
            a = b;
            b = temp;
         }

        void inc(long* value) {
         (*value)++;
         }
      };
    }
  """)
```

```
ns = cppyy.gbl.RefTest
assert ns.Box.__dict__['a'].__cpp_reflex__(cppyy.reflex.TYPE) == 'long'
assert ns.Box.__dict__['b'].__cpp_reflex__(cppyy.reflex.TYPE) == 'long*'

@numba.njit()
def inc_b(d, k):
    for i in range(k):
        d.inc(d.b)

@numba.njit()
def inc_c(d, k):
    for i in range(k):
        d.inc(d.c)

x = random.randint(1, 5000)
y = random.randint(1, 5000)
z = random.randint(1, 5000)
b = ctypes.c_long(y)
c = ctypes.c_long(z)
```

```
d = ns.Box(x, b, c)
k = 5000
```

Here the members of Box class are initialized via pass-by-ref

```
x = 2856
y = 1896
```

```
inc_b(d, k)
inc_c(d, k)

assert b.value == y + k
assert c.value == z + k
```

Args :
CppClass(Box)
int64

Result:
d.b, d.c are incremented through *pointers*

POINTER AND REFERENCE TYPE SUPPORT

The "pointer" like behavior is especially useful in cases like these

```
cppyy.cppdef("""
  namespace RefTest {
      class Box{
          public:
              long a;
              long *b;
              long *c;
              Box(long i, long& j, long& k){
              a = i;
              b = &j;
              c = &k;
              }

              void swap_ref(long &a, long &b) {
                  long temp = a;
                  a = b;
                  b = temp;
               }

              void inc(long* value) {
               (*value)++;
               }
          };
      }
  """)
```

x, b = 2856
y, c = 1893

```
inc_b(d, k)
inc_c(d, k)

assert b.value == y + k
assert c.value == z + k
```

b = 7856,
c = 6893

Now we call a swap by ref function which causes b and c to automatically swap on the python side

b = 7856,
c = 6893

```
d.swap_ref(d.b, d.c)

assert b.value == z + k
assert c.value == y + k
```

b = 6893,
c = 7856

POINTER AND REFERENCE TYPE SUPPORT

The fact that Numba lowers cppyy calls using C++ pointers to LLVM IR opens the avenue of significant speedups

```
label 0:
    d = arg(0, name=d)                          ['d']
    k = arg(1, name=k)                          ['k']
    $2load_global.0 = global(range: <class 'range'>) ['$2load_global.0']
    $6call_function.2 = call $2load_global.0(k, func=$2load_global.0, args=[Var(k, test_numba.py:433)], kws=(), vararg=None, varkwarg=None, t
    $8get_iter.3 = getiter(value=$6call_function.2) ['$6call_function.2', '$8get_iter.3']
    $phi10.0 = $8get_iter.3                     ['$8get_iter.3', '$phi10.0']
    jump 10                                     []
label 10:
    $10for_iter.1 = iternext(value=$phi10.0) ['$10for_iter.1', '$phi10.0']
    $10for_iter.2 = pair_first(value=$10for_iter.1) ['$10for_iter.1', '$10for_iter.2']
    $10for_iter.3 = pair_second(value=$10for_iter.1) ['$10for_iter.1', '$10for_iter.3']
    $phi12.1 = $10for_iter.2                    ['$10for_iter.2', '$phi12.1']
    branch $10for_iter.3, 12, 28                ['$10for_iter.3']
label 12:
    i = $phi12.1                                ['$phi12.1', 'i']
    $16load_method.3 = getattr(value=d, attr=inc) ['$16load_method.3', 'd']
    $20load_attr.5 = getattr(value=d, attr=c) ['$20load_attr.5', 'd']
    $22call_method.6 = call $16load_method.3($20load_attr.5, func=$16load_method.3, args=[Var($20load_attr.5, test_numba.py:434)], kws=(), va
    jump 10                                     []
label 28:
    $const28.0 = const(NoneType, None)          ['$const28.0']
    $30return_value.1 = cast(value=$const28.0) ['$30return_value.1', '$const28.0']
    return $30return_value.1                    ['$30return_value.1']

Numba Base Function pointer call funcptr:  %".135" = bitcast i8* %".134" to i8* (i8*, i64*)*
Numba Base Function pointer call args:  [<ir.CastInstr '.128' of type 'i8*', opname 'bitcast', operands [<ir.LoadInstr '.127' of type '{i64,
```

STD::VECTOR<>* AND NUMPY ARRAYS

We can explore those speedups by also adding pointer and reference support to std::vector objects

This is achieved by constructing IR Pointer Types to Array and Vector Types, that point to cppyy.gbl.std.vector() objects linked to numpy arrays for initialization

```
cppyy.cppdef("""
template<typename T>
std::vector<T> make_vector(const std::vector<T>& v, std::vector<T> l) {
    std::vector<T> u(l);
    u.insert(u.end(), v.begin(), v.end());
    return u;
}
    namespace RefTest {
        class BoxVector{
            public:
                std::vector<long>* a;

                BoxVector() : a(new std::vector<long>()) {}
                BoxVector(std::vector<long>* i) : a(i){}

                void square_vec(){
                for (auto& num : *a) {
                    num = num * num;
                 }
                }

                 void add_2_vec(long k){
                for (auto& num : *a) {
                    num = num + k;
                 }
                }

                 void append_vector(const std::vector<long>& value) {
                    *a = make_vector(value, *a);
                 }
               };
            }
    """)
```

```
a = np.random.randint(1, 100, size=10000, dtype=np.int64)
b = np.random.randint(1, 4, size=10, dtype=np.int64)

x = cppyy.gbl.std.vector['long'](a.flatten())
y = cppyy.gbl.std.vector['long'](b.flatten())
```

```
Aaron Jomy
@numba.njit()
def square_vec_fast(d):
    for i in range(5):
        d.square_vec()

Aaron Jomy
@numba.njit()
def square_vec_slow(x):
    for i in range(5):
        x = np.square(x)
    return x
```

```
Aaron Jomy
@numba.njit()
def add_vec_fast(d):
    for i in range(10000):
        d.add_2_vec(i)

Aaron Jomy
@numba.njit()
def add_vec_slow(x):
    for i in range(10000):
        x = x + i
    return x
```

```
t0 = time.time()
square_vec_fast(ns.BoxVector(y))
time_square_njit = time.time() - t0
```

Here the members of the BoxVector class are initialized via pass-by-ref within the python function call

STD::VECTOR<>* AND NUMPY ARRAYS

Initial benchmarks with Numpy-C++ equivalent functions for the same operations:

```
t0 = time.time()
add_vec_fast(ns.BoxVector(x))
time_add_njit = time.time() - t0

t0 = time.time()
square_vec_fast(ns.BoxVector(y))
time_square_njit = time.time() - t0
```

```
assert (np.array(y) == np_square_res).all()
assert (np.array(x) == np_add_res).all()
```

Directly access the result since the Numba obtains the cling address to the cppyy.gbl.std.vector

Njitted Cppyy function
Standard loop over numpy function

```
njit execution time: 20.38860321044922
numpy execution time: 77.96597480773926
njit execution time: 0.058174133300078125
numpy execution time: 0.2288818359375
```

Njitted Cppyy function
Njitted Numpy function

```
cppyy execution time: 17.93956756591797
numpy execution time: 368.68953704833984
cppyy execution time: 0.042438507080078125
numpy execution time: 291.74327850341797
```

STD::VECTOR<>* AND NUMPY ARRAYS

Exploring vectorization speedups with a dot product operation

```
cppyy.cppdef("""
namespace RefTest {
    class DotVector{
        private:
            std::vector<long>* a;
            std::vector<long>* b;

        public:
            long g = 0;
            long *res = &g;
            DotVector(std::vector<long>* i, std::vector<long>* j) : a(i), b(j) {}

            long dot_product(const std::vector<long>& vec1, const std::vector<long>& vec2) {
                                long result = 0;
                                for (size_t i = 0; i < vec1.size(); ++i) {
                                    result += vec1[i] * vec2[i];
                                }
                                return result;
            }
        };
    }""")
```

Initialising using pointers
Running the dot product through pass-by-ref

STD::VECTOR<>* AND NUMPY ARRAYS

Exploring vectorization speedups with a dot product operation

```
cppyy.cppdef("""
namespace RefTest {
    class DotVector{
        private:
            std::vector<long>* a;
            std::vector<long>* b;

        public:
            long g = 0;
            long *res = &g;
            DotVector(std::vector<long>* i, std::vector<long>* j) : a(i), b(j) {}

            long self_dot_product() {
                long result = 0;
                size_t size = a->size();  // Cache the vector size
                const long* data_a = a->data();
                const long* data_b = b->data();

                for (size_t i = 0; i < size; ++i) {
                    result += data_a[i] * data_b[i];
                }
                return result;
            }
```

Even faster dot product by using the DotVector datamembers which are in turn std::vector pointers

STD::VECTOR<>* AND NUMPY ARRAYS

Exploring vectorization speedups with a dot product operation

```python
# Aaron Jomy
@numba.njit()
def dot_product_fast(d):
    res = 0
    for i in range(10000):
        res += d.self_dot_product()
    return res
# Aaron Jomy
def np_dot_product(x, y):
    res = 0
    for i in range(10000):
        res += np.dot(x, y)
    return res
```

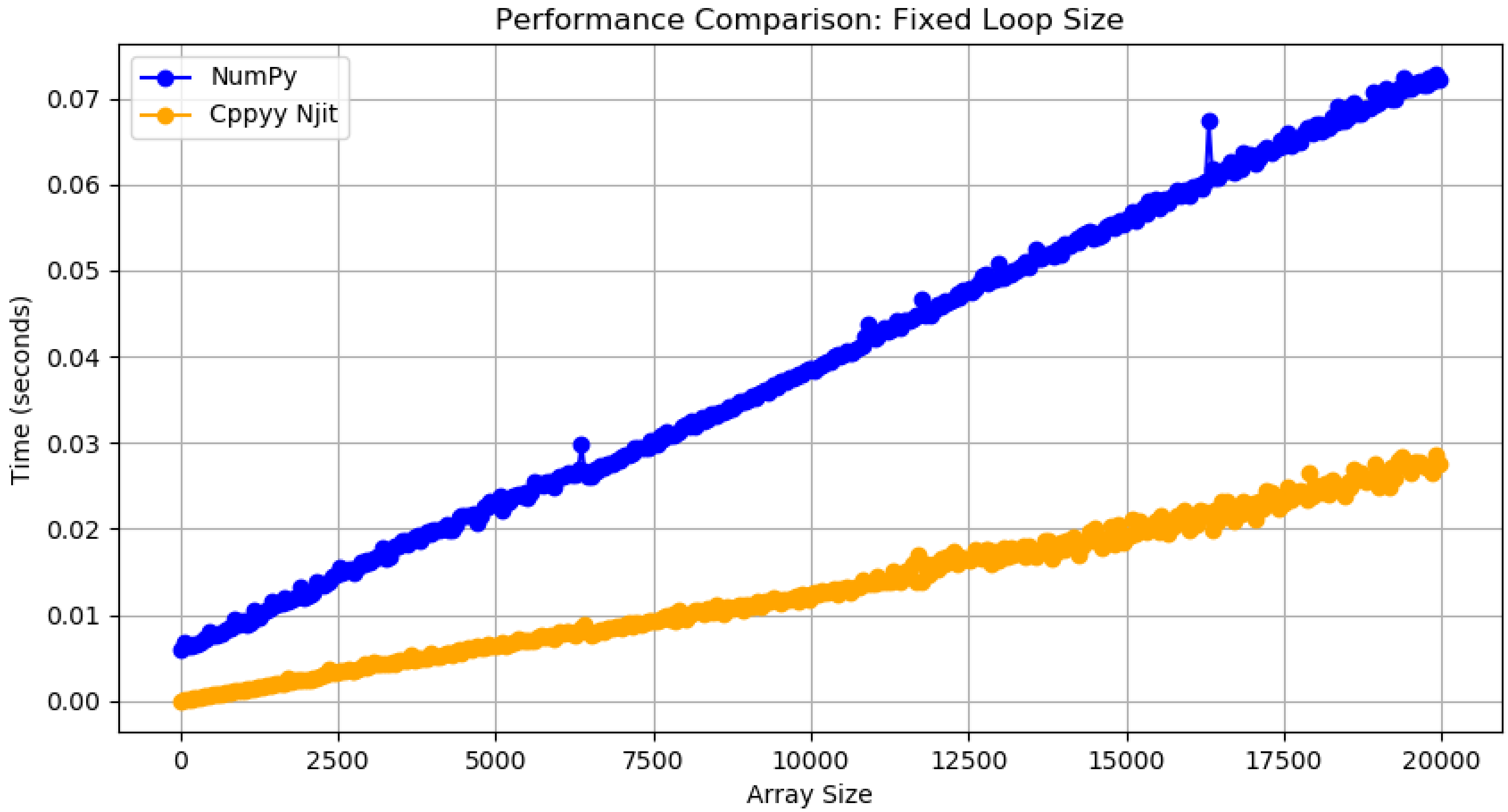
```python
x = cppyy.gbl.std.vector['long'](a.flatten())
y = cppyy.gbl.std.vector['long'](b.flatten())
d = ns.DotVector(x, y)
dot_product_fast(d)
res = 0

t0 = time.time()
njit_res = dot_product_fast(d)
time_njit = time.time() - t0

res = 0
t0 = time.time()
res = np_dot_product(x, y)
time_np = time.time() - t0
```

SOME BENCHMARK TRENDS

SOME BENCHMARK TRENDS



CURRENT STATUS WITH EIGEN

Templated class args like Eigen are resolved to cpp types and successfully matches the CPPOverloads in those cases

The Eigen numba typeinfer is refactored into the C++ expression and handled in numba2cpp

Able to handle the templated class Eigen::Matrix< Scalar_, Rows_, Cols_, Options_, MaxRows_, MaxCols_ > using Eigen::Dynamic
so the dispatcher typeinfer is CppClass(Eigen::Matrix<double,-1,-1,0,-1,-1>)

```
# Define the templated function that takes Eigen objects
cppyy.cppdef('''
template<typename T>
T multiply_scalar(T value, int64_t scalar) {
    return value * scalar;
}
''')
```

```
Return type: <class cppyy.gbl.Eigen.Matrix<float,3,1,0,3,1> at 0x73d66d0>
VAL: <class cppyy.gbl.Eigen.Matrix<float,3,1,0,3,1> at 0x73d66d0> type: <class 'Matrix<float,3,1,0,3,1>_meta'>
```

```
> c = {_UnboxContext: 3} _UnboxContext(context=<numba.core.cpu.CPUContext obj
> obj = {LoadInstr} %".21" = load i8*, i8** %".5"
> self = {PythonAPI} <numba.core.pythonapi.PythonAPI object at 0x7fdf9591ad30>
> typ = {ImplClassType} CppClass(Eigen::Matrix<float,3,1,0,3,1>)
```

Thank You!

Aaron Jomy