Extending the Cppy support in Numba

Progress Update
RECAP

- **Cppy**: An automatic, run-time, Python-C++ bindings generator
- **Cling** is used in Cppy's backend since an interactive C++ interpreter provides a runtime exec approach to C++ code
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.
  
  ```python
  @numba.jit
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
      trace += cppyy.gbl.tanh(a[i, 1])
    return a + trace
  ```

  ```python
  x = np.arange(10000, dtype=np.float64).reshape(100, 100)
  run jit_test(x)
  ```
  
  Numba disabled = 0.10824203491210938 ms
  Numba type infer in dispatcher: array(float64, 2d, C)
  Numba njit enabled = 0.00786781311931562 ms

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

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]
• 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

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

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

Args:
CppClass(Box)   int64

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

x = 2856  
y = 1896

inc_b(d, k)  
inc_c(d, k)  
assert b.value == y + k  
assert c.value == z + k
The “pointer” like behavior is especially useful in cases like these

```cpp

copy.cpp

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)++;
            }
        };
    }

    void inc_b(d, k) {
        b = 7856;
        c = 6893;
        assert b.value == y + k
        assert c.value == z + k
    }

    void inc_c(d, k) {
        b = 7856;
        c = 6893;
        assert b.value == y + k
        assert c.value == z + k
    }

    Box x, b = 2856, y, c = 1893;
    Box 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

    void inc_ref(d.b, d.c) {
        d.b = 7856;
        d.c = 6893;
        assert b.value == z + k
        assert c.value == y + k
    }

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

    b = 6893, c = 7856

    assert b.value == z + k
    assert c.value == y + k
}
The fact that Numba lowers `cppyy` calls using C++ pointers to LLVM IR opens the avenue of significant speedups.
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.
Initial benchmarks with Numpy-C++ equivalent functions for the same operations:

```python
# Njitted Cppyy function

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
```

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

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

---

Njitted Cppyy function

Standard loop over numpy function

njit execution time: 20.38860321044922
numpy execution time: 77.96597480773926
njit execution time: 0.0581741330078125
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
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;
            }
    };
}"""
```
Exploring vectorization speedups with a dot product operation

```cpp
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
Exploring vectorization speedups with a dot product operation

```python
@numba.njit()
def dot_product_fast(d):
    res = 0
    for i in range(10000):
        res += d.self_dot_product()
    return res

def np_dot_product(x, y):
    res = 0
    for i in range(10000):
        res += np.dot(x, y)
    return res
```

```python
x = cppyylstdcvector['long'](a.flatten())
y = cppyylstdcvector['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()
np_res = np_dot_product(x, y)
time_np = time.time() - t0
```
Some benchmark trends

Performance Comparison: Fixed Array Size

- NumPy
- Cppy Njit

Number of Loops

Time (seconds)
SOME BENCHMARK TRENDS

Performance Comparison: Fixed Loop Size

- NumPy
- Cppy Njit
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>)

```cpp
# Define the templated function that takes Eigen objects
cpyy.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 0x7fd6d6d0>
VAL: <class cppyy.gbl.Eigen.Matrix<float,3,1,0,3,1> at 0x7fd6d6d8> type: <class 'Matrix<float,3,1,0,3,1>._meta'>
Thank You!

Aaron Jomy