Written by Aaron
on September 25, 2023

Google Summer of Code 2023 @CERN-HSF

This is a blog post entailing the cumulative work I achieved during my 4 months as a GSoC student under the CERN-HSF organization.

Here is a link to the project proposal :

- **Extending the Cppyy support in Numba**

Presentations at the Compiler Research Group:

- **May 17, 2023**
- **Aug 16, 2023**

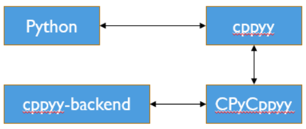
Experience

Participating in Google Summer of Code has been incredibly rewarding for me. I had the opportunity to immerse myself in the advanced computing environment at CERN and interact with technologies in the ROOT ecosystem like Cling. My project, "Extending the Cppyy support in Numba", allowed me to delve deep into the architecture of Cppyy, enabling me to understand its structure and significance. This experience has enriched my knowledge base and elevated my problem-solving and coding abilities, helping me see the broader implications of software development tools and their applications in real-world scenarios. Working on a mixed C++ and Python codebase on a daily basis subconsciously elevated my problem-solving skills in other aspects of my life, like university or other coding challenges

My experience was quite fantastic, and I am glad that I could succesfully contribute to the amazing work in high performance and interactive computing being done at CERN. My favourite part of Google Summer of Code was the exposure to software development on the highest level, specifically in compiler research. I believe CERN-HSF has given me a platform to systematically explore and work in high-performance computing. This also includes the wonderful conversations I had with my mentors and other group members that definitely contributed to improving my knowledge and ideas. I am very grateful for the footing I have obtained in performing open-source contributions in a streamlined fashion.

Extending the Cppyy support in Numba

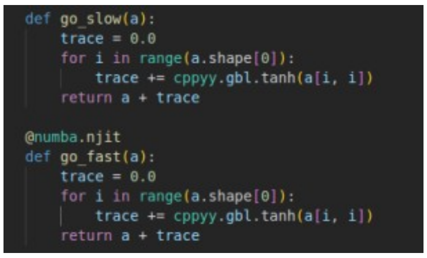
Cppyy : An automatic, run-time, Python-C++ bindings generator
Cling : is used in backend since an interactive C++ interpreter provides a runtime exec approach to C++ code
Numba : A JIT compiler that translates Python and NumPy code into fast machine code



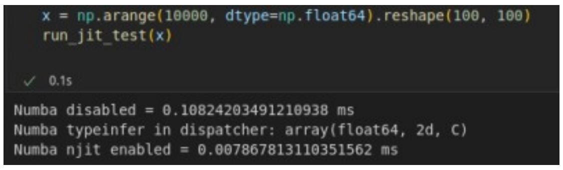
Why use Numba?

The compute time overhead while switching between languages accumulates in loops with cppyy objects.

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



```
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
```



```
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
```

Challenges

Typing is one of the largest problems posed: Template function utilization, reference types and correct function matching depend on the type resolution system

Type Inference solution: A mechanism to handle implicit casting based on propagated type info and the cppyy reflection layer.

Note: Typing does not backtrack since the numba extension will only ever obtain the numba type inference result.

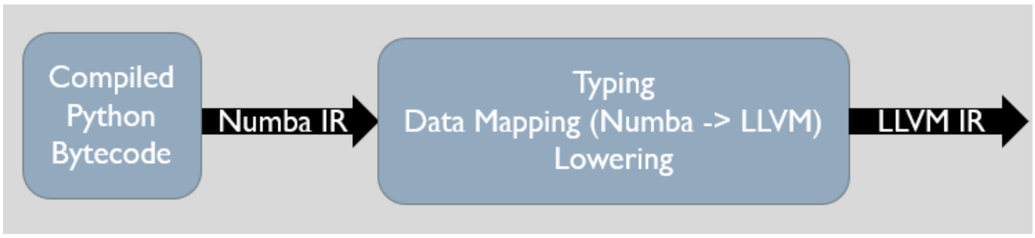
Python	Numba Type	LLVM Type used in Numba lowering
3 (int)	int64	i64
3.14 (float)	float64	double
(1, 2, 3)	UniTuple(int64, 3)	[3 x i64]
(1, 2.5)	Tuple(int64, float64)	{i64, double}
np.array([1, 2], dtype=np.int32)	array(int64, 1d, C)	{i8, *i8*, i64, i64, i32*, [1 x i64]}
“Hello”	unicode_type	{i8, *i64*, *i32*, *i32*, *i64*, *i8*, i8*}

This is an important step to addressing the challenges on the avenues of adding efficient reference type detection and handling, as well as extending the capabilities of the current template functions supported.

Primary Deliverables:

- Add general support for C++ templates in Numba through Cppyy
- Add support for C++ reference types in Numba through Cppyy

Numba pipeline



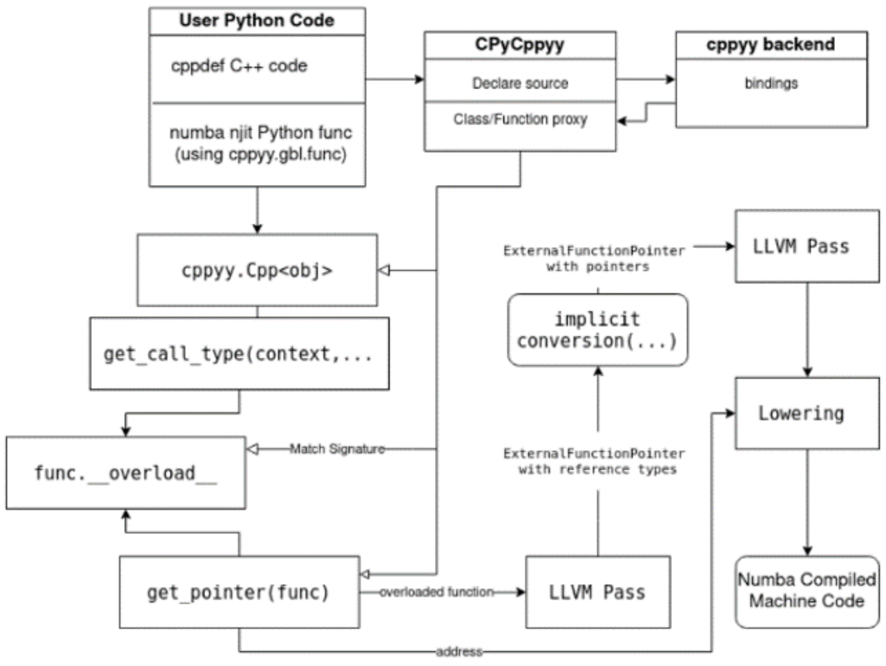
Typing : Numba core has a type inference algorithm which assigns a nb_type for a variable

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 in Cppyy:



Final project 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]
- nJIT(non-boxing calls) for Eigen templated classes passed-by-ref [Test 14]

Pull Requests during GSoC:

https://github.com/wlav/cppyy/pull/177

https://github.com/wlav/cppyy-backend/pull/11

https://github.com/wlav/cppyy-backend/pull/14

https://github.com/compiler-research/xeus-clang-repl/pull/56

https://github.com/compiler-research/xeus-clang-repl/pull/57

https://github.com/compiler-research/xeus-clang-repl/pull/58

https://github.com/compiler-research/xeus-clang-repl/pull/60

https://github.com/compiler-research/xeus-clang-repl/pull/61

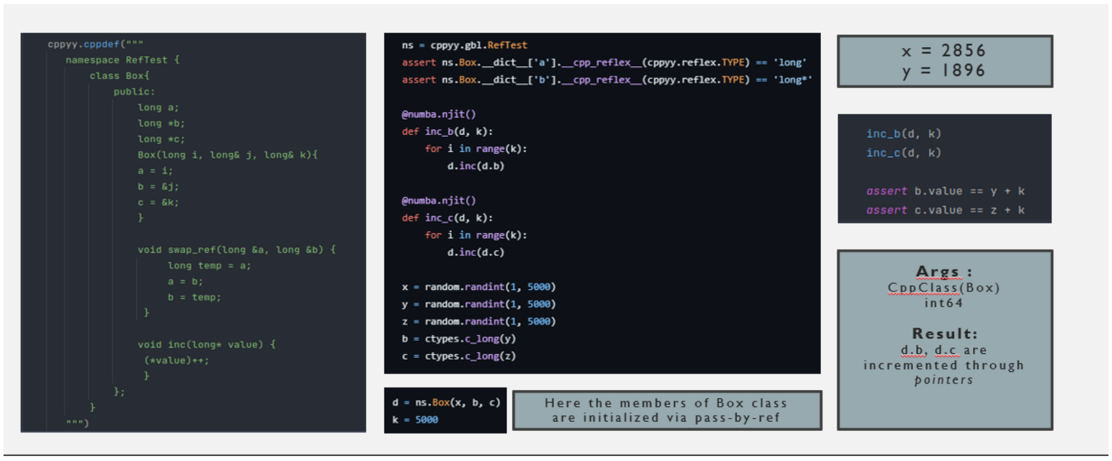
https://github.com/compiler-research/xeus-clang-repl/pull/62

https://github.com/compiler-research/xeus-clang-repl/pull/63

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)
```

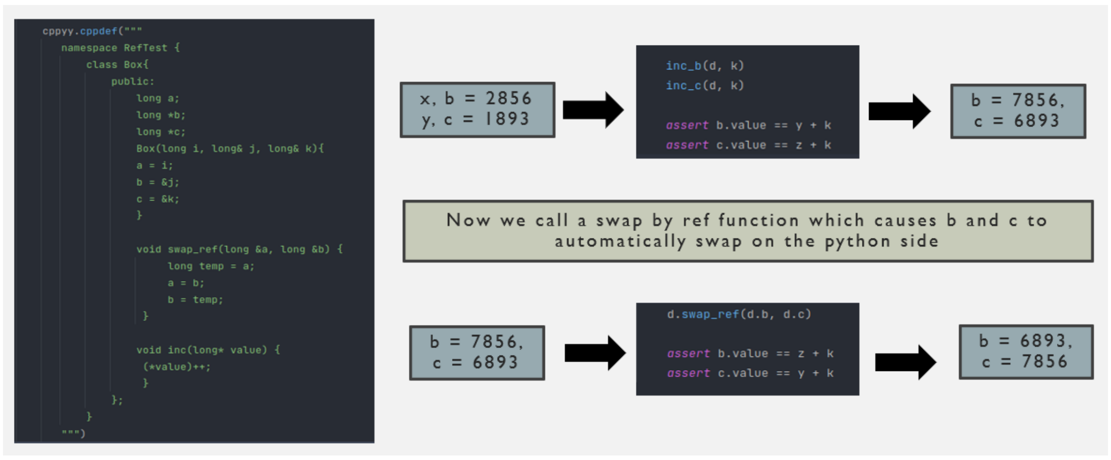
```
d = ns.Box(x, b, c)
k = 5000
```

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

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

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

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*

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
```

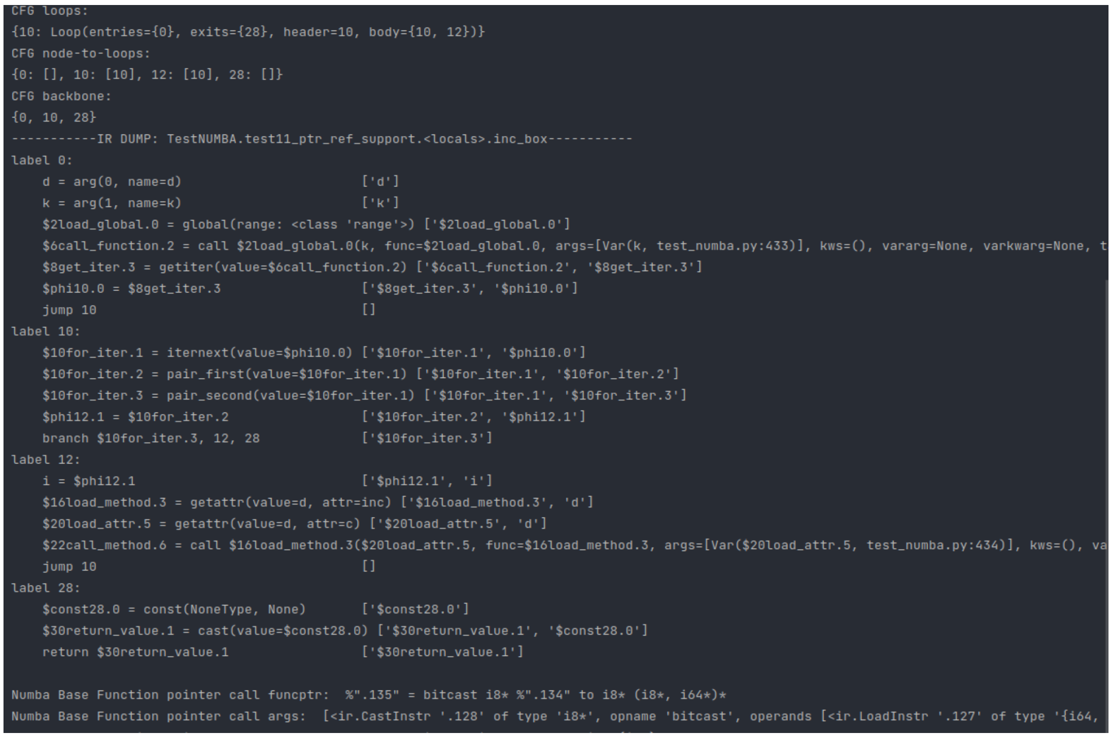
Now we call a swap by ref function which causes b and c to automatically swap on the python side

```
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
```

The fact that Numba lowers Cppyy calls that use C++ pointers, to LLVM IR, opens an avenue of significant speedup possibilities.

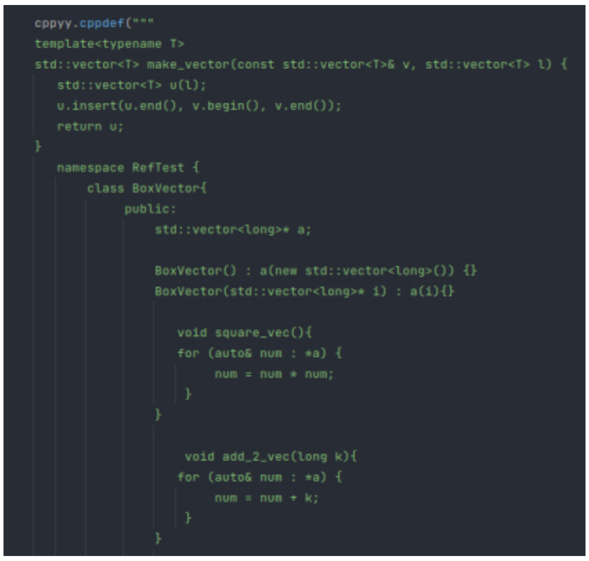


```
CFG loops:
{10: Loop(entries={0}, exits={28}, header=10, body={10, 12})}
CFG node-to-loops:
{0: [], 10: [10], 12: [10], 28: []}
CFG backbone:
{0, 10, 28}
-----------IR DUMP: TestNUMBA.test11_ptr_ref_support.<locals>.inc_box-----------
label 0:
    d = arg(0, name=d)                       ['d']
    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,
```

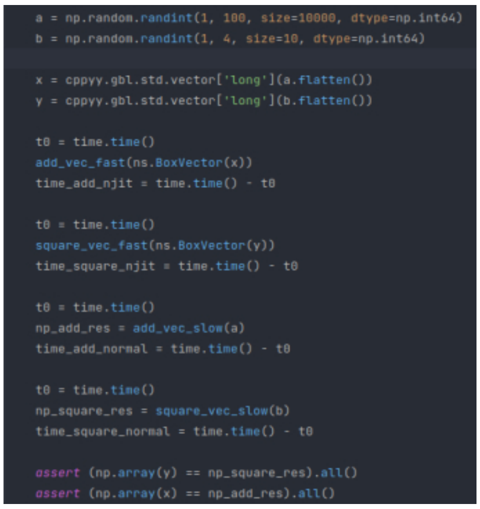
STD::VECTOR<>* and Numpy Arrays

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

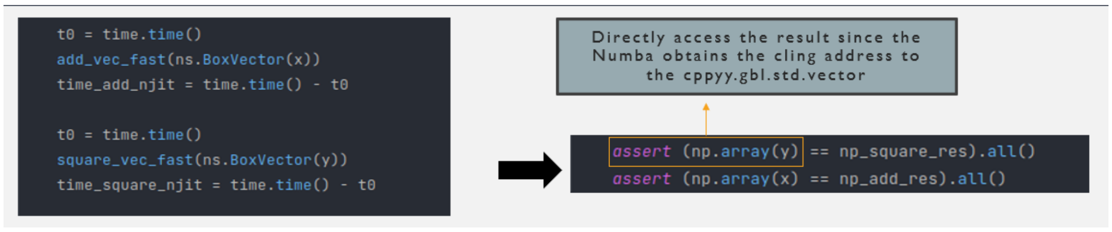


```
cppyy.cppdef("""
template<typename T>
std::vector<T> make_vector(const std::vector<T>& v, std::vector<T> l) {
   std::vector<T> u(l);
   u.insert(u.end(), v.begin(), v.end());
   return u;
}
   namespace RefTest {
       class BoxVector{
            public:
                std::vector<long>* a;

                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;
                    }
                }
```



```
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())

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

t0 = time.time()
np_add_res = add_vec_slow(a)
time_add_normal = time.time() - t0

t0 = time.time()
np_square_res = square_vec_slow(b)
time_square_normal = time.time() - t0

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

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



```
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

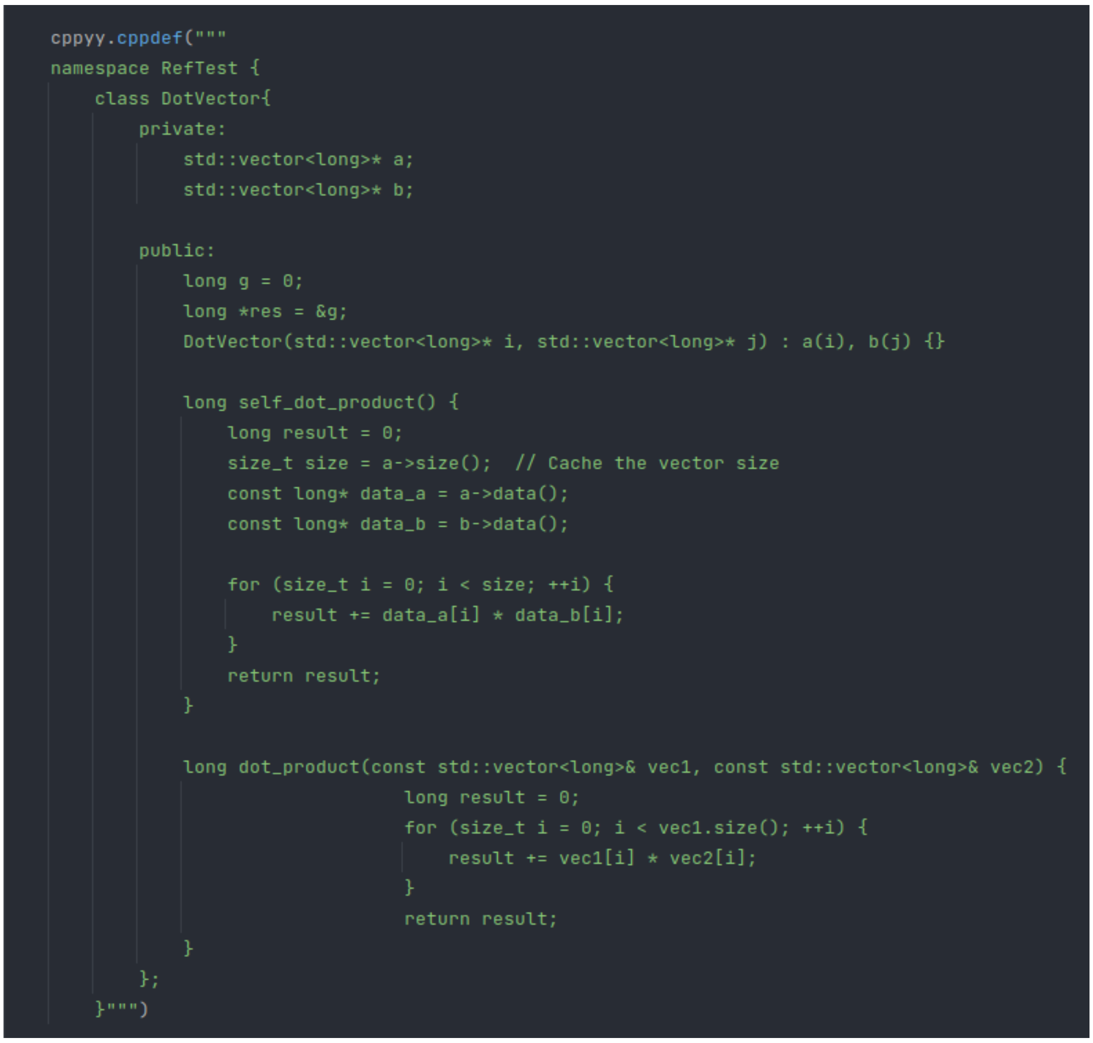
```
assert (np.array(y) == np_square_res).all()
assert (np.array(x) == np_add_res).all()
```

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

Experiment [Time in milliseconds]	NumPy (Standard Python loop)	NumPy NJIT (Equivalent LLVM IR)	Cppyy NJIT
1	0.19	246.39	0.042
2	0.22	291.74	0.058
3	77.95	368.68	17.94
4	92.36	374.97	20.38

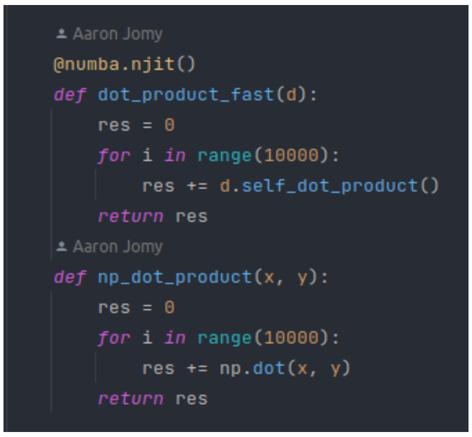
Exploring vectorization speedups with a dot product operation:

Initialising using pointers and running the dot product through pass-by-ref. We can achieve an even faster dot product by using the DotVector datamembers which are in turn `std::vector` pointers:

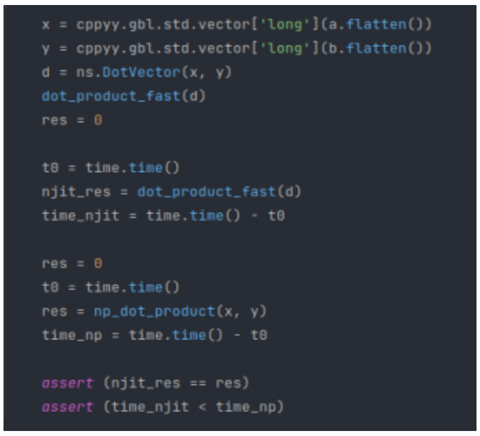


```
cppyy.cppdef("""
namespace RefTest {
    class DotVector{
        private:
            std::vector<long>* a;
            std::vector<long>* b;

        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;
            }

            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;
            }
        };
    }""")
```

Usage :

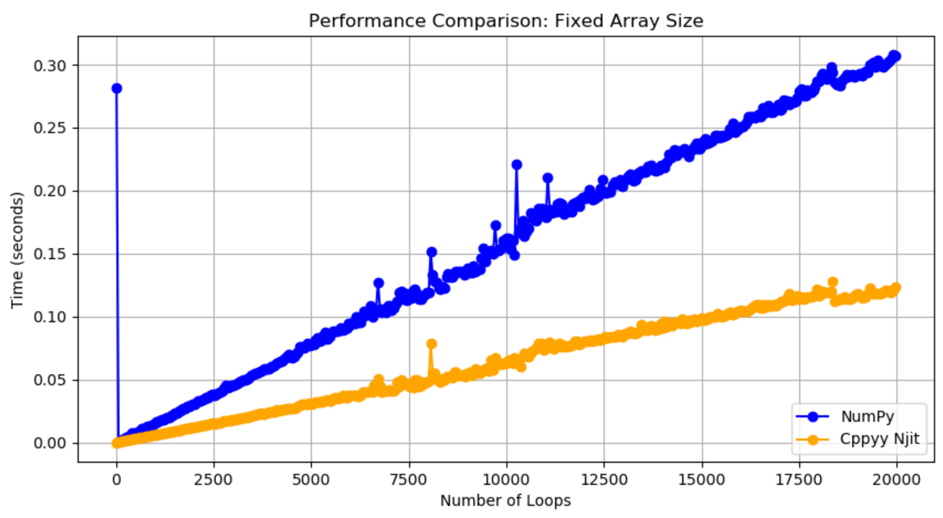


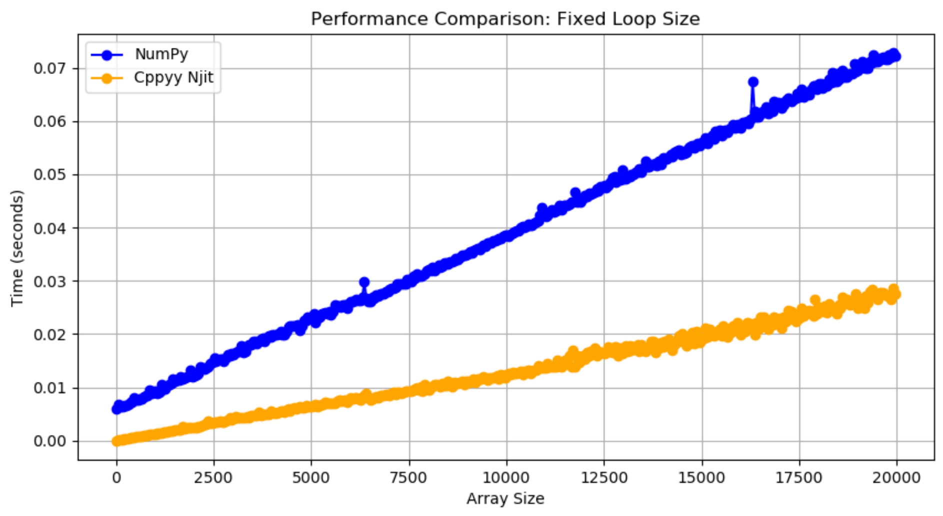
```
Aaron Jomy
@numba.njit()
def dot_product_fast(d):
    res = 0
    for i in range(10000):
        res += d.self_dot_product()
    return res
Aaron Jomy
def np_dot_product(x, y):
    res = 0
    for i in range(10000):
        res += np.dot(x, y)
    return res
```



```
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

assert (njit_res == res)
assert (time_njit < time_np)
```

Some Benchmark Trends





Added Eigen Support

Templated class args like `Eigen::Matrix<Scalar, Rows, Cols, Options, MaxRows, MaxCols>` are resolved to C++ types and successfully matches the CPPOverload. The Numba typeinfer of the Eigen metaclass is refactored into the C++ expression and handled in `numba2cpp` . This support currenbtly works only for `Eigen::Dense` .

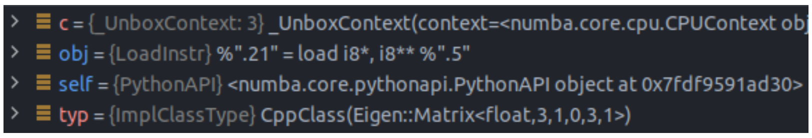
Dispatcher typeinfer :

`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'>
```

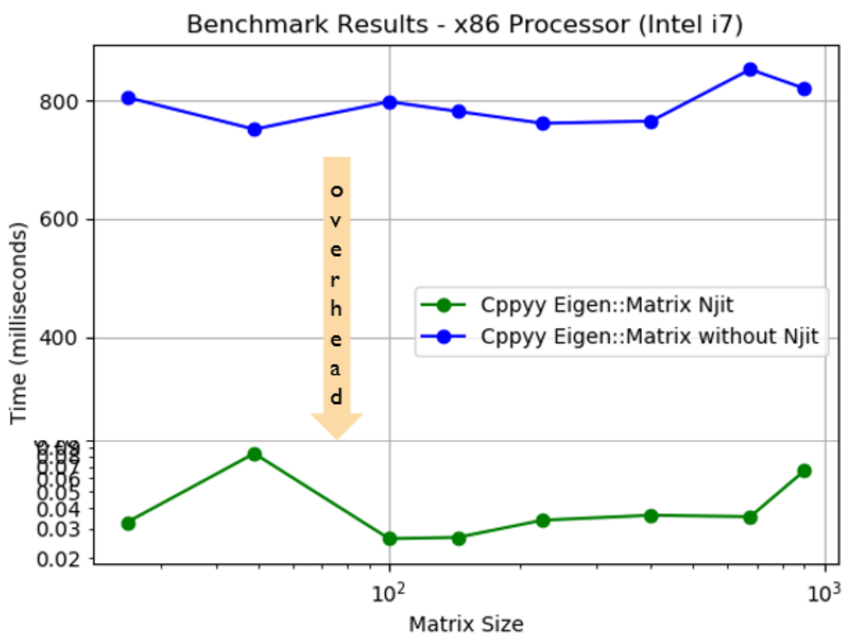
```
Return type: <class cppyy.gbl.Eigen.Matrix<float,3,1,0,3,1> at 0x73d66d0>
VAL: <class cppyy.gbl.Eigen.Matrix<float,3,1,0,3,1> at 0x73d66d0> type: <class 'Matrix<float,3,1,0,3,1>_meta'>
```



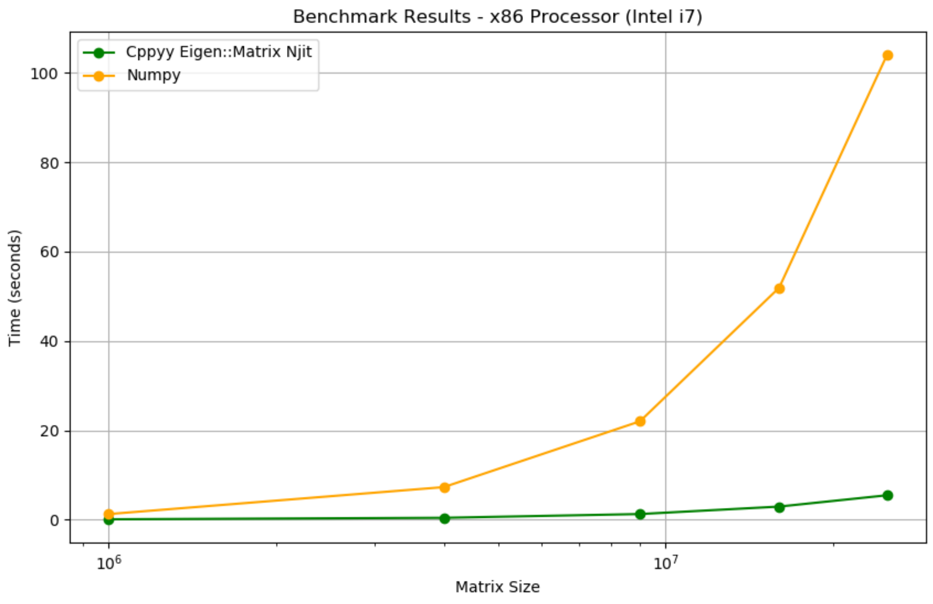
```
> ≣ c = {_UnboxContext: 3} _UnboxContext(context=<numba.core.cpu.CPUContext obj
> ≣ obj = {LoadInstr} %".21" = load i8*, i8** %".5"
> ≣ self = {PythonAPI} <numba.core.pythonapi.PythonAPI object at 0x7fdf9591ad30>
> ≣ typ = {ImplClassType} CppClass(Eigen::Matrix<float,3,1,0,3,1>)
```

Exciting Results with Eigen:

- The overhead of crossing the language barrier is eliminated:



- Comparison with NumPy `np.dot` for the same matrix dimensions:



←

Top