Google
Summer of Code

Extend the Cppyy support in Numba

Aaron Jomy (maximusron)
Manipal Institute of Technology
Computer Science Undergraduate
aaronjomyjoseph@gmail.com

Mentors:

Baidyanath Kundu
baidyanath.kundu@cern.ch

Wim Lavrijsen
WLavrijsen@lbl.gov

Vassil Vassilev
Vassil.Vassilev@cern.ch

Abstract

Numba is a JIT compiler that translates a subset of Python and NumPy code into fast machine code. Cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++.

Cppyy has to pay a time penalty each time it needs to switch between languages which can multiply into large slowdowns when using loops with cppyy objects. This is where Numba can help. Since Numba compiles the code in loops into machine code it only has to cross the language barrier once and the loops thus run faster.

Initial support for Cppyy objects in Numba enabled the use of builtin types and classes, but some essential C++ features, such as references and STL classes, are not yet supported.

The project aims to add support for C++ reference types in Numba through Cppyy and improve the existing numba extension implementation to provide general support for C++ templates.

This added support will allow cppyy users to define a wider array of standard and templated functions that can leverage reference types to the C++ code defined in python.

Basic Information

Personal Information:

Name:	Aaron Jomy
Email:	aaronjomyjoseph@gmail.com
Mobile Number:	+91 9901299595
Time Zone:	IST (UTC +5:30)
Github:	maximusron
University:	Manipal Institute of Technology
Major:	Computer Science Engineering
Degree:	Bachelor of Technology
Current Year:	3rd year (Expected graduation 2024)
Availability:	I am available for the entire project duration

Programming Experience:

As a part of Project MANAS, my university student team that focuses on intelligent robotics, I have actively contributed towards the development of autonomous software on a driverless car, aerial and ground vehicles. This provided me with most of my experience in C++ and Python, and working with configuring, modifying, and debugging various open-source toolkits.

I am currently working at the Artificial Intelligence and Robotics lab at the Indian Institute of Science where I work on researching and developing tracking and navigation algorithms in both Python and C++. As such, I have experience working with large codebases deployed on robot platforms.

My background in python stems from my past projects in deep learning that familiarized me with a working understanding of python and its interpreter. I have frequently used numba in python code to provide speedups through vectorization

I have previously taken Stanford's CS106B online course, which advanced my understanding of C++ and efficient software design. I developed multiple modules in C++ for Point Cloud processing that required me to make efficient design and control flow choices to ensure a high execution rate. I am also comfortable with version control systems.

After exploring how numba works, I am very excited at the prospect of working on cppyy.

Development Environment

The choice of my development environment was an initial dilemma while working on cppyy due to the large amount of C++ code present in CPyCppyy that acts as the cpp-python intermediate.

Hence, I chose to use CLion, with the Python extensions that provide me with the necessary support tools and the freedom to configure a local interpreter.

A workaround I used through CLion was adding a blank CMakeList file so I can mark the python directory in cppyy as a source root folder. This lets me emulate all the functionality I have with PyCharm along with the flexibility to examine the C++ side of things.

For debugging, since the python traces can end up in the CPyCppyy intermediate, I chose to attach my python interpreter process to my native debugger(GDB) that allows me to trace it out.

For quick working/debugging with just the cppyy frontend, I use PyCharm.

My primary OS is Ubuntu 20.04 LTS with the xmonad wm and zsh.

Evaluation Task

I had submitted my solution for my evaluation task, which was to add numba support for a const char* type argument in the cppyy defined function:

```
import cppyy, numba
cppyy.cppdef("""
void print_str(const char *str) {
    printf(\"Received: %s \\n\", str);
}
""")
@numba.njit
def print_cpp_str(msg):
    cppyy.gbl.print_str(msg)
print_cpp_str("message")
```

Though my initial solution was a workaround to add a unicode_type mapping to const char*, my approach led me to realize that the information about the overload cannot be known before the mapping is done. This type-resolution system followed by cppyy poses the largest challenge. I tried to solve this shortcoming with a preliminary approach, modifying the type mapping and the resolution mechanism to use a many-to-one mapping from numba to cpp instead of the earlier used inversion of the cpp2numba mapping. I have outlined the same in detail in this issue ticket: https://github.com/wlav/cppyy/issues/147

My modifications led me to perform the following:

- Add support for functions with int64_t args(and other types whose other numba signatures were being overwritten)
- Modify the args passed to `__overload__` to follow the format in CPyCppyy to allow multiple arguments(which earlier gave an argcount exceeded error since it was being passed as a packed generator expression)
- This also allowed me to be able to successfully run a function of the type:

```
cppyy.cppdef("""
            namespace NumbaSupportExample {
            template<typename T1>
            T1 add(T1 a, T1 b) { return a + b; }
            }""")
```

which was a good first step towards improving the template support in the numba extension.

Task Implementation Approach

The project proposal focuses on two primary deliverables:

1. Add general support for C++ templates in Numba through Cppyy:

Some directions on extended support can be observed in the following functionality:

Functionality goal #1: Multiple template parameters:

```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <typename T, typename U>
   T multiply(T t, U u) { return t * u; }
}""")
```

Functionality goal #2: Non-type template parameters:

```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <typename T, int N>
   T power(T t)
   {
       T result = 1;
       for (int i = 0; i < N; ++i)
           result *= t;
       return result;
   }
}""")
```

Functionality goal #3: Template template parameters:

```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <template <typename> class Container, typename T>
   T sum(const Container<T> &container)
   {
       T total = T(0);
       for (const T &value : container)
       {
           total += value;
       }
       return total;
   }
}""")
```

With the objective of improving the handling of C++ templates in the `numba_ext.py` script, I propose to leverage the features of the cppyy.TemplateProxy class, focusing on the modifications of the `CppFunctionNumbaType` class and the `typeof_template` function.

- Enhancing the type resolution system to handle multiple template parameters, non-type template parameters, and template-template parameters.
  This requires updating how the type information is extracted to cover these additional cases. (I explored the same with some preliminary ideas I tried, which are referred to under the evaluation task section)
- The handling of template classes and functions should be improved to support template specializations and partial template specializations
- Ensure the boxing/unboxing mechanisms handles the new template constructs(correct template resolution as a class/function ensures the mechanism automatically just works)

Task 1:

- Modify the `typeof_template` function:
  a. Update the function to leverage the `TemplateProxy` object to handle a wider range of template parameter types.

  b. Implement custom logic using the `TemplateProxy` object for parameter handling and Numba type deduction and compatibility.

Task 2:

- Enhance the `CppFunctionNumbaType` class:
  a. Add a handler for when a `CPPOverload` object derived from the template function is passed as `_func` in CppFunctionNumbaType for better signature matching.

  b. Update existing methods or add new ones to handle argument deduction, specialization, and partial specialization using the `TemplateProxy` object.

  c. Implement helper functions that utilize the `TemplateProxy` object for more generic template construct management. This builds on this existing PR which improves `__overload__` that can be accessed here. The implicit casting method can be utilized to better handle template matches on the numba_ext.py side.

- Update the lowering process:
  a. Adjust the lower_external_call function to utilize the `TemplateProxy` object for better handling of template-related constructs. This should ensure compatibility with Numba's LLVM code generation by integrating the object into the updated lowering process.

- Develop comprehensive tests:
  Design test cases that cover different use cases and edge cases related to the `TemplateProxy` object, `CppFunctionNumbaType` class, and `typeof_template` function modifications.

2. Add support for C++ reference types in Numba through Cppyy

On the `numba.ext` side for reference detection:

Unfortunately, numba_ext.py doesn't have a built-in mechanism to automatically detect whether a cppyy function argument is a reference or not since Python doesn't support pointers or references, and thus the information about whether an argument is a reference or not is lost.

In order to work around the limitation, we can rely on both:

- The numba type inference on explicit CPointer objects
- Numpy arrays that can be used through `numba.carray(ptr)`

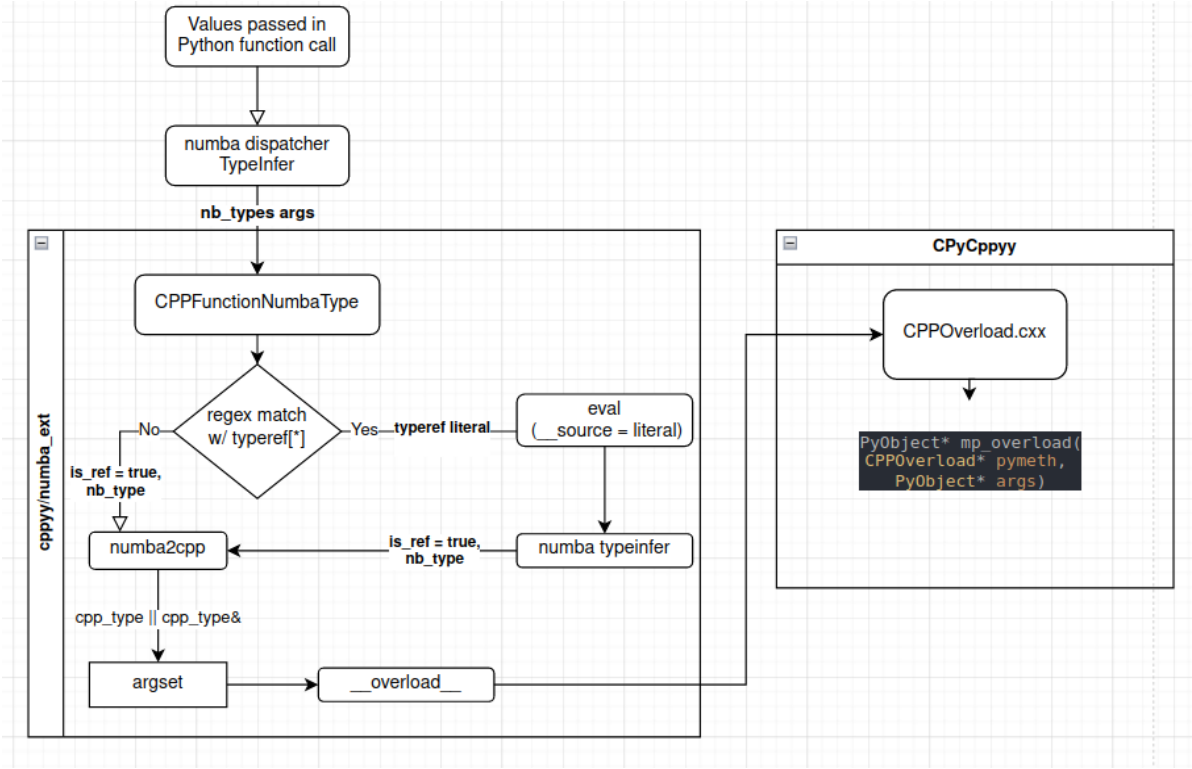
That can be unified through a class that subclasses the `numba.Types.RawPointer` class(`NumbaCppyyPointer` explained in task 2)

The following tasks would require modifications in the `__overload__` method of the cppyy.CPPOverload object, as well as in the corresponding numba function signature. This signatures modifications ensure that the reference information is consistent between the two.

Task 3: For reference type arguments explicitly passed as `numba.types.CPointer` objects

- We can build on this case by integrating the following regex match into the type resolution system.

```
Numba typeinfer in dispatcher: typeref[15*], typeref[20.0*]
Resolved literals via second numba typeinfer pass
Numba possible args: (['long&', 'long long&', 'int64_t&'], ['double&'])
ARG COMBO ('long&', 'double&')
```



Task 4: Utilizing the numba `RawPointer` class:

An effective arg scheme can utilize the `RawPointer` class in numba to pass over reference information.

A blueprint subclass of `RawPointer` could hold a reference to the type of the pointee, plus the raw pointer value:

```
class NumbaCppyyPointer(nb.types.RawPointer):
    def __init__(self, pointee):
        self._pointee = pointee
        super().__init__(pointee)

    def get_pointee_type(self):
        return self._pointee
```

And a function to obtain the number of indirections:

```
def count_indirections(ptr):
    indirections = 0
    tp = ptr
    while isinstance(tp, NumbaCppyyPointer):
        indirections += 1
        tp = tp.get_pointee_type()
    return indirections
```

Hence a unified design of this class can be implemented that handles derived classes too:

```
class NumbaCppyyPointer(nb.types.RawPointer):

    def __init__(self, pointee):
            self._pointee = pointee
            self._is_derived = False

            if isinstance(pointee, nb.types.Type):
                # Handle normal classes and ctypes
                super().__init__(pointee)
            elif isinstance(pointee, tuple) and len(pointee) == 2 and
    isinstance(pointee[0], nb.types.Type):
                # Handle derived classes passed as base class arguments
                self._is_derived = True
                self._base_type = pointee[0]
                self._derived_type = pointee[1]
                super().__init__(nb.types.RawPointer(pointee[1]))
            else:
                # Handle primitive types
                super().__init__(pointee)
```

Task 5: The `__overload__` modifications:

An automated solution to perform reference detection using cppyy's reflection layer can utilize this approach:

- Perform standard signature matching using the numba2cpp pipeline that performs implicit casts on the CPyCppyy side.
- For all methods in `fMethods`, the matching will be done based on the underlying type and not a string match. For a signature match, the presence of reference indications in the C++ code can be used to set the arg types in the returned `sig_matched`.

```
CPPOverload::Methods_t &methods = fMethodInfo->fMethods;
   for (auto &meth : methods)
   {
     bool found = accept_any;
     if (!found){
       PyObject *pysig2 = meth->GetSignature(false);
       std::string sig2(CPyCppyy_PyText_AsString(pysig2));
       sig2.erase(std::remove(sig2.begin(), sig2.end(), ' '),
std::end(sig2));
           Py_DECREF(pysig2);
//Strip indicators of reference & to ensure only underlying types are used
to match
           if (//signature match)
           {
               found = true;
               std::cout << "Matched CPyCppyy Signature:" << sig2 << "\n";
           }
        }

        return CPPOverload *newmeth, sig_matched
   }
```

After the successful signature match occurs, an iteration over the arg list is performed, and a representation of which args are reference types is passed back along with the overload

Change the `mp_overload` function in `CPPOverload.cxx` to handle the new `RawPointer` objects created in step 1.

- Modify the signature of the `FindOverload` to accept `PyObject*` arguments
- Iteration over the input argument tuple should now perform reference type checking.

A prototype of the function:

```
PyObject* FindOverload(const std::string& signature, int want_const = -1);
```

Will be modified to this overload:

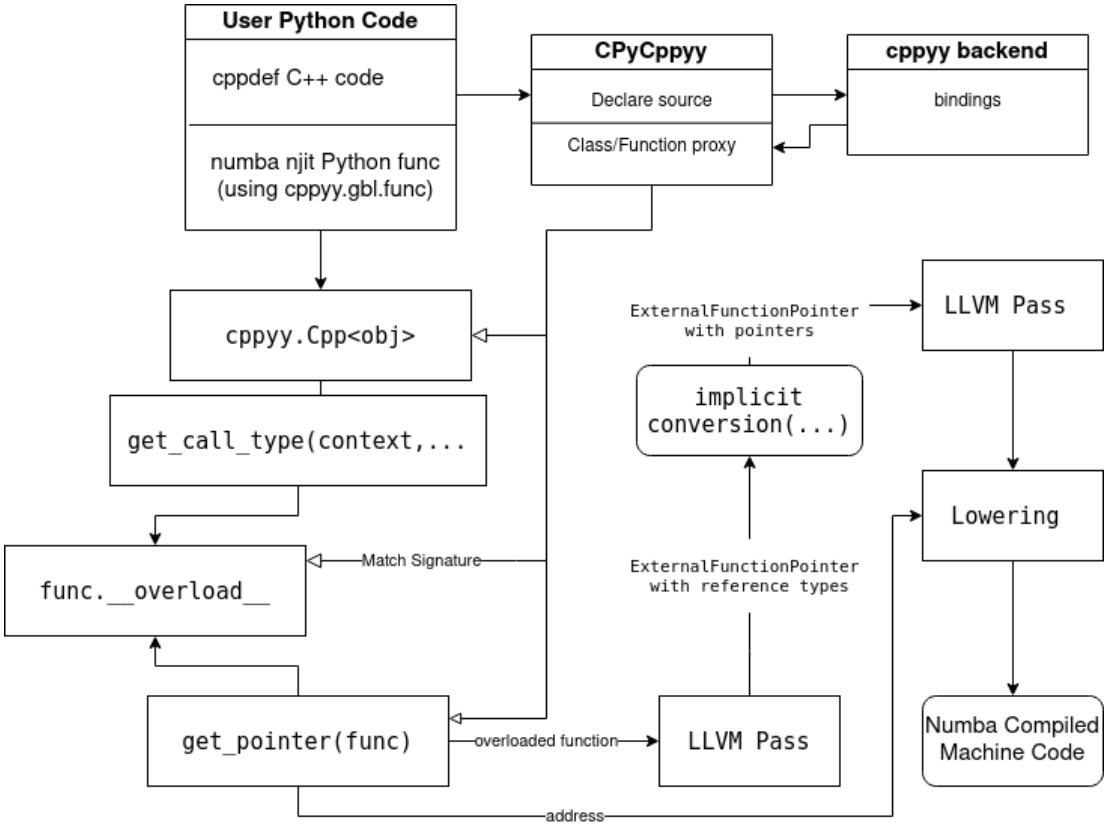
```
PyObject* FindOverloadRef(const std::string& signature, PyObject* args, int
want_const);
```

With these modifications, functions can accept pointers for numpy arrays, standard primitive arguments, and specified classes

CPPOverload now performs reference detection, and the `numba_ext` module can handle these arguments by passing the relevant `RawPointers` to the C++ function in the lowering stage.

Task 6: Handling the reference types for lowering:

Now that the reference addresses to the relevant data are obtained, we are required to update the LLVM lowering pipeline since reference types in LLVM IR need implicit conversion from the pointer to its value when it is required.



A solution to this defines an updated lowering mechanism that accounts for the possibilities with reference types:

Define a new `LLVMFunctionPass` that identifies function arguments that are reference types and converts them to pointers. This pass modifies the input `llvmlite.ir.Function` signature by replacing reference types with pointer variables initialized with the respective reference addresses.

After this pass LLVM IR function will have pointer args which allows the function to be called through Numba with the `NumbaCppyyPointer(nb.types.RawPointer)`

1. Add a new LLVMFunctionModel to the `nb_dm` module representing an LLVM Function type. This model will have a corresponding numba type to `llvmlite.ir.Function`.

2. In `lower_external_call`, create a `llvmlite.ir.Function` with the same signature as the ExternalFunctionPointer(built using `ext_sig` and the pointer from `ol.get_pointer`)

3. Run it through `LLVMFunctionPass` for the reference types to be converted to pointers

4. The `lower_external_call` will now lower the CppFunctionNumbaType to an `LLVMFunctionModel`. This pass uses the CPPOverload object to obtain the `llvmlite.ir.Function`, then create a `LLVMFunctionModel` with the appropriate signature.

Task 7: The return type is a reference:

When cpp2numba is run on the overloads cppyy reflection RETURN_TYPE, it fails because no match is present for reference types. We add a handler that detects a returned reference type here and add changes in `get_call_type` that will modify the `ol.sig` return_type. A preliminary implementation of a fix successfully demonstrates a working model of a reference return type:

```
def ref_test():
    cppyy.cppdef("""
    int64_t& ref_add_8(int64_t x) {
        static int64_t result = x+8;
        return result;
    }
    """)
    @numba.njit()
    def run_add(a):
        k = cppyy.gbl.ref_add_8(a)
        result = k[0]
        return result

    x = 17
        print("Result of ref_add_8",
run_add(x))
```

```
Matched CPyCppyy Signature
2:(int64_t)
Reference return type detected
Performing lowering

Obtaining the function __overload__
in get_pointer:
Matched CPyCppyy Signature
2:(int64_t)
Succesful arg combo match in
get_pointer= ('int64_t',)

Result of ref_add_8: 25
```

This forms the basis of the tasks that build up on this for general reference type returns:

Detection:

We can leverage the cppyy reflection layer to perform a string check for reference types:

```
return_type = self._func.__cpp_reflex__(cpp_refl.RETURN_TYPE)
if return_type.endswith("&"):
   is_reference_return = True
   return_type = return_type[:-1]
else:
   is_reference_return = False
```

This could be implemented in the cpp2numba function, but `get_call_type` must know when the return type is a reference.

Handling:

We add a handler in `get_call_type` that decides the external signature return type based on the reflection layer information.

```
return_type = self._func.__cpp_reflex__(cpp_refl.RETURN_TYPE)
if return_type.endswith("&"):
   is_reference_return = True
   return_type = return_type[:-1]
else:
   is_reference_return = False

sig_return_type = cpp2numba(return_type
```

- Implement the required additions to handle a reference return type and perform implicit conversions for the best match numba pointer type to be used in the external signature return value.

```
if is_reference_return:
   print("Reference return type detected")
   sig_return_type = #implicit_conversion#(sig_return_type)
```

Hence, the return value of the external signature initializing the `ExternalFunctionPointer` object in the lowering call will be a numba C type Pointer that is further used in the lowering.

The rest remains the same, creating the `ExternalFunctionPointer` and calling the C++ function using `context.call_function_pointer`.

Timeline

Community Bonding Period: May 4 - 28

Discussions with mentors in finalizing the design of the type resolution and casting system that accounts for improved numba2cpp mapping, multi-arg template handling, and implicit casts for numba-to-cppref types.

Phase 1: June 1 - July 10

Week 1: **Begin work on type resolution system**

- Implement the modified numba2cpp pipeline for type mapping
- Add tests for implicit type mapping from nb_type to cpp type.

Week 2: **Additions to type resolution mechanism for templates**

- Implement the parameter handling pass on the TemplateProxy object in numba_ext.py in the `typeof_template` (Task 1)
- Implement additions in `CppFunctionNumbaType` class (Task 2)

Week 3: **Build up on week 1 onto the reference conversions**

- Implement reference type detection on user explicit pointer args(Task 3)
- Add tests for detection and running the type resolution implemented in Week 1.

Week 4: **Begin work on CPyCppyy continuing on the reference types**

- Add changes to `__overload__` in CPyCppyy for unified arg handling (Task 5)
- Implement the `FindOverloadRef` function overload and run tests.

Week 5: **Testing and analysis of Week 1-4**

- Add tests for preliminary type resolution in template functions and reference detection.
- Document the type mapping, reference detection, and arg handling mechanisms developed in weeks 1 - 4.

Phase 2: July 10 - August 28

Week 6: **Modifications in numba_ext to provide references to CPPOverload**

- Implement the restructuring required in `get_call_type` to handle both primitive and numpy array args passed as references.

Week 7: **Reference return types**

- Implement additions to handle a reference return type and implicit conversions for the numba pointer type to be used in the external signature return value. (Task 7)

- Add tests for implicit conversions of general case reference return types

Week 8: **Reference type handling in numba lowering (Task 4)**

- Implementing the `NumbaCppyyPointer` to unify pointers of detected reference args.

- Modify the control flow of `get_call_type` for reference type arguments based on the arg handling mechanism done in week 4.

Week 9: **Begin work on LLVM IR for reference types (Task 6)**

- Implement `LLVMFunctionPass` for handling reference types in LLVM IR.

- Implement the implicit reference type to pointer conversion required.

- Add tests for the `LLVMFunctionPass` and conversions.

Week 10: **Integration and testing**

- Integrate the cumulative changes to streamline the pipeline w.r.t both template and reference types

- Add tests during integration to validate the performance of added functionality.

Week 11: **Final week**

- Buffer period for any unresolved tasks.

- Add documentation and results of the added tests.

- Compile progress report