Efficient and Accurate Automatic Python Bindings with Cppyy & Cling

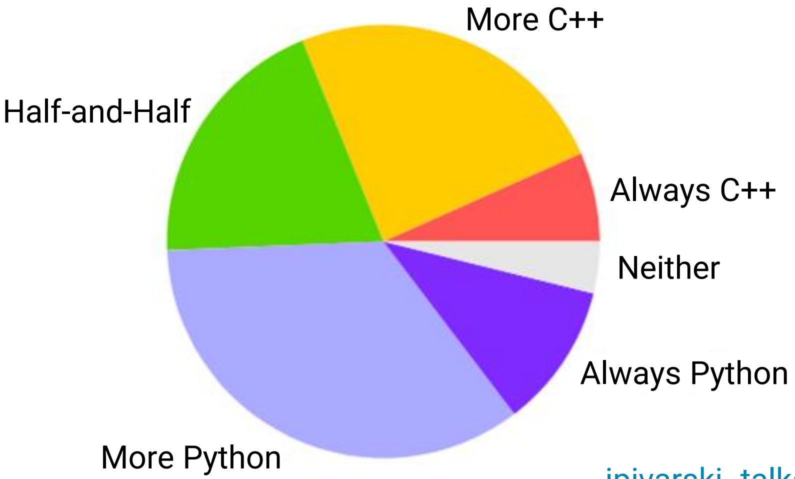
Authors: Baidyanath Kundu, Vassil Vassilev, Wim Lavrijsen

Compiler Research

"The current work is partially supported by National Science Foundation under Grant OAC-1931408. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation."

Introduction

PyHEP 2020 survey:
"How often do you use Python relative to C/C++"



jpivarski -talks

Goal: Tight language integration between Python and C++

Cppyy

Cppyy is an automatic C++ - Python runtime bindings generator and supports a wide range of C++ features.

C++ code (MyClass.h)

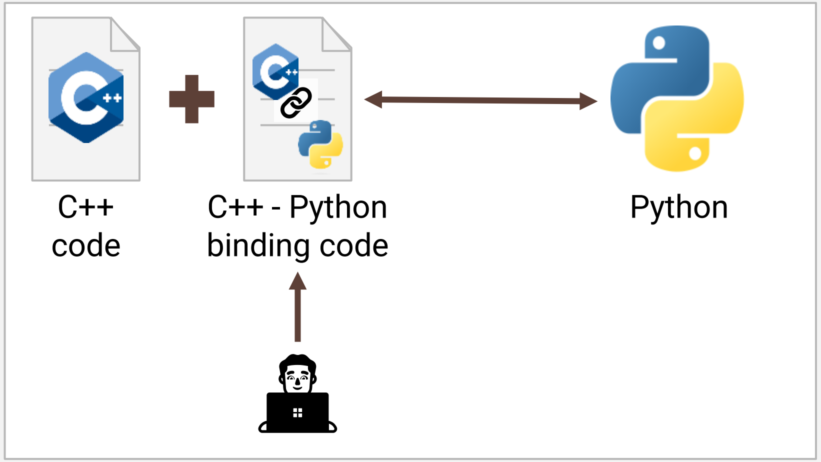
```
struct MyClass {
  MyClass(int i) : fData(i) {}
  virtual ~MyClass() {}
  virtual int add(int i) {
    return fData + i;
  }
  int fData;
};
```

Python Interpreter

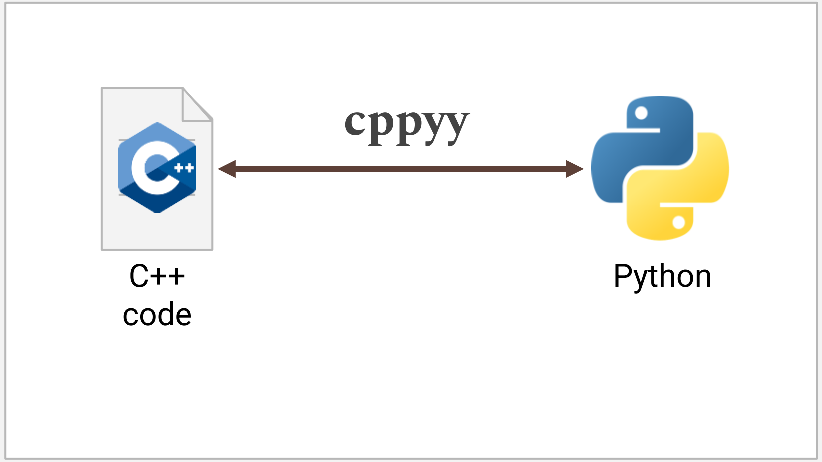
```
>>> import cppyy
>>> import cppyy.gbl as Cpp
>>> cppyy.include("MyClass.h")
>>> class PyMyClass(Cpp.MyClass):
...  def add(self, i):
...    return self.fData + 2*i
...
>>> m = Cpp.MyClass(1)
>>> m.add(2) # = 1 + 2
3
>>> m = PyMyClass(1)
>>> m.add(2) # = 1 + 2 * 2
5
```

Python-C++ Bindings Generators

Manual Bindings Generators



Automatic Bindings Generators



Motivation

Can we make cppyy faster and lighter?

Original design

Disadvantages of using ROOT/meta in Cppyy:

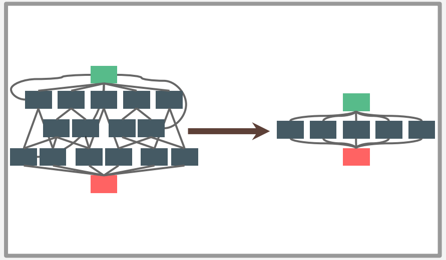
- Performance penalty from its abstraction
- Difficult to extend
- Hard to evolve reflection interfaces

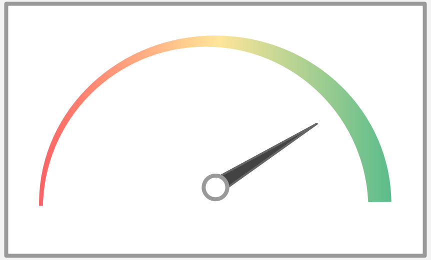
Goal

Current design

Our goal is rebase Cppyy on top of pure LLVM to address the disadvantages. Clang-REPL, a generalization of Cling in LLVM, will provide the necessary reflection information.

Benefits





Simpler codebase

Removal of string parsing logic leads to a simpler codebase

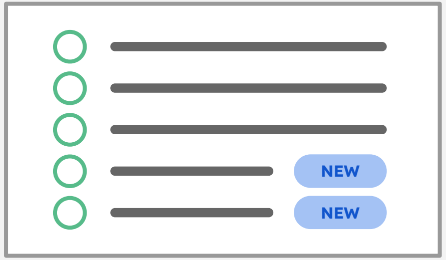
Better performance

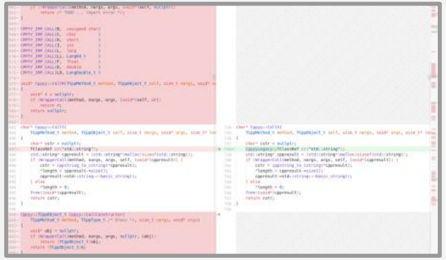
It also leads to better performance.

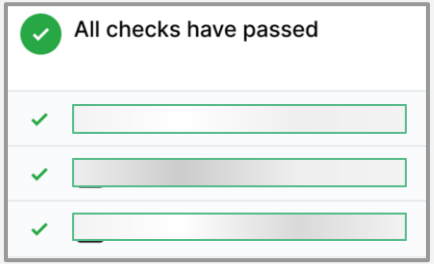
LLVM umbrella

The libInterOp interfaces will be a part of LLVM toolchain through Clang-REPL

Benefits







Better C++ feature set support

C++ features such as partial template specialisation is possible because of libInterOp

Huge reduction in lines of code

A lot of dependencies and workarounds are removed thus reducing the lines of code required to run Cppyy

Well tested interoperability layer

The libInterOp interfaces have full unit test coverage

Template Instantiation Example

C++ code (Tmpl.h)

```cpp
template <typename T>
struct Tmpl {
    T m_num;
    T add (T n) {
        return m_num + n;
    }
};
```

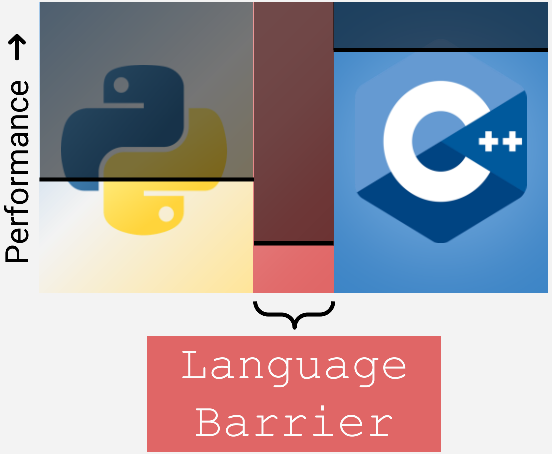
Currently, our developmental Cppyy version can run basic examples such as the one here. Features such as standalone functions and basic classes are also supported.

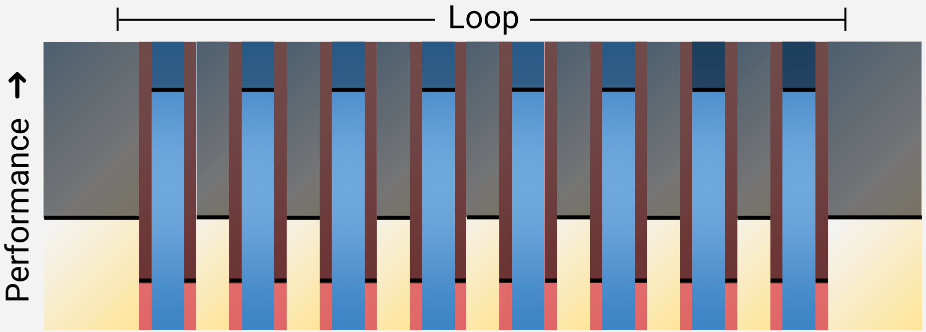
Python Interpreter

```python
>>> import cppyy
>>> import cppyy.gbl as Cpp
>>> cppyy.include("Tmpl.h")
>>> tmpl = Tmpl[int]()
>>> tmpl.m_num = 4
>>> print(tmpl.add(5))
9
>>> tmpl = Tmpl[float]()
>>> tmpl.m_num = 3.0
>>> print(tmpl.add(4.0))
7.0
```

Further Optimization of Python/C++

Problem 1



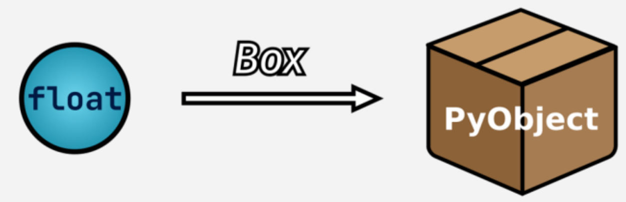


Usual usage with Cppyy

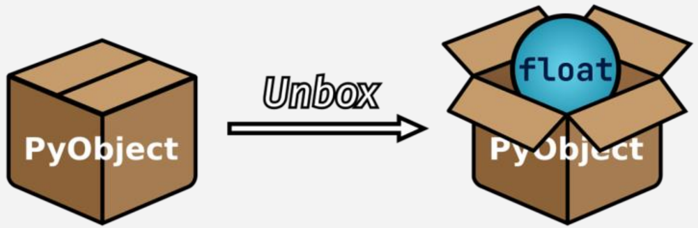
Further Optimization of Python/C++

Problem 2

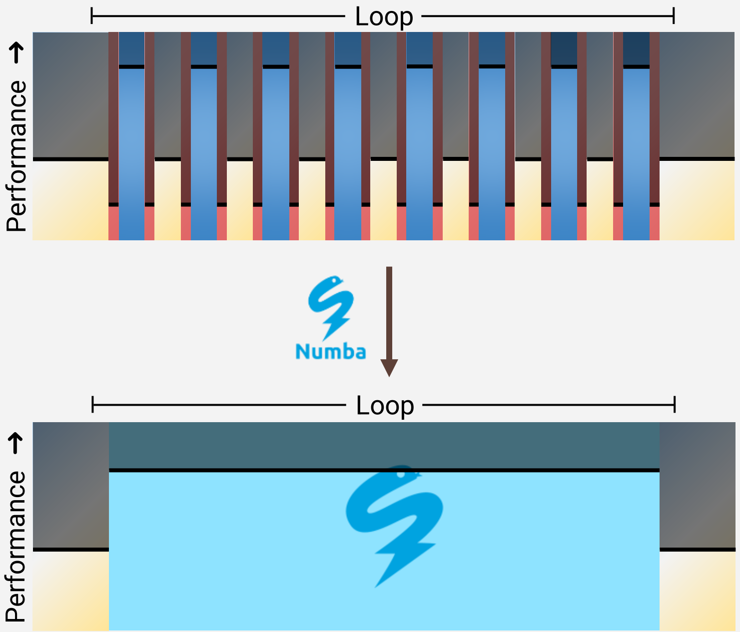
```
num = 6.0
```




```
num = num ** 3
```



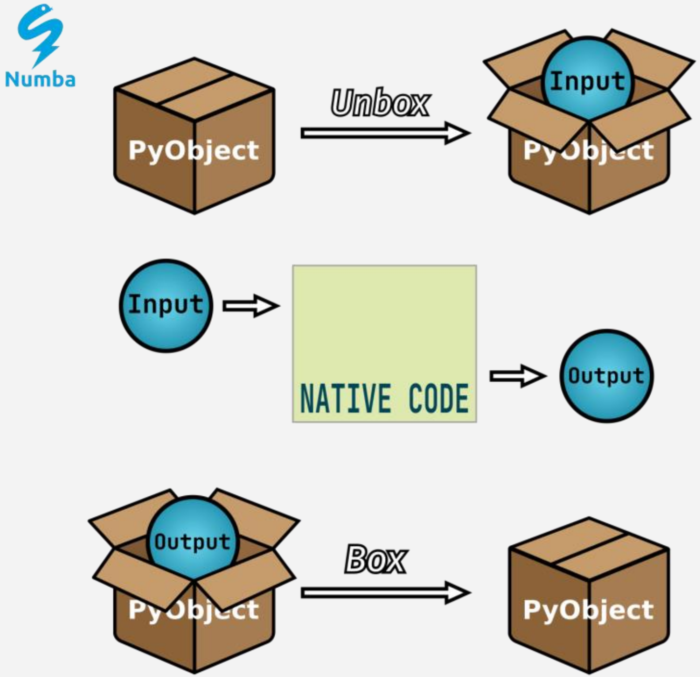
Extending Cppyy using Numba is the solution



Numba removes the language barriers in the loop

Solution 1

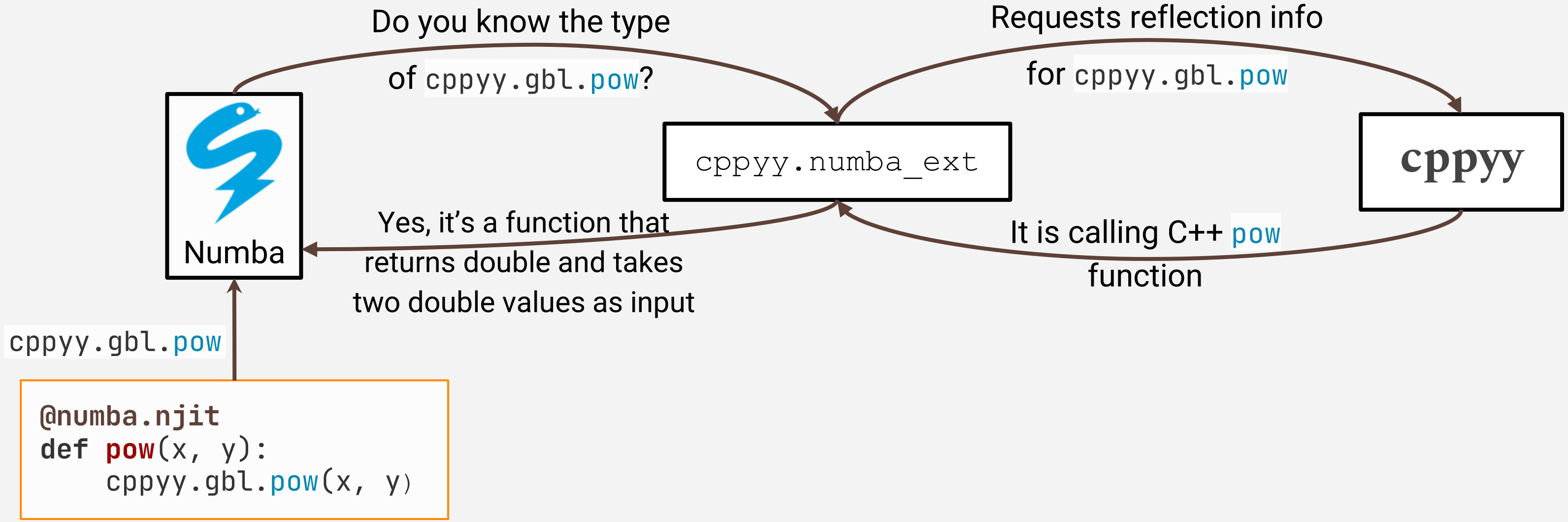
Solution 2



Cppyy-Numba Extension

Requirements of the Numba compilation step:

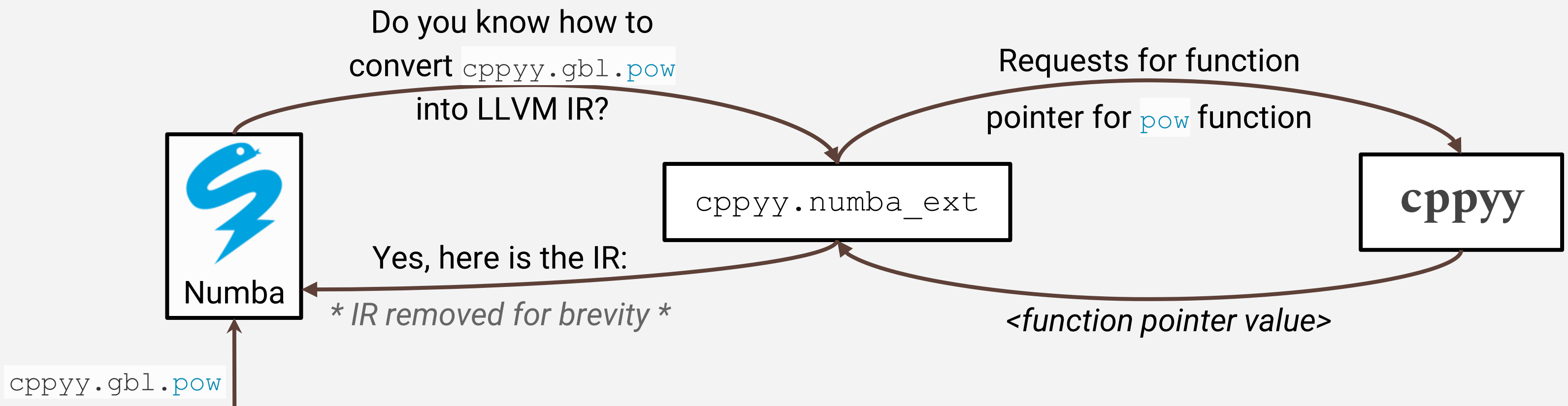
- ➢ Typing Information
- ➢ Conversion to LLVM IR



Cppyy-Numba Extension

Requirements of the Numba compilation step:

- Typing Information
- Conversion to LLVM IR




```
@numba.njit
def pow(x, y):
    cppyy.gbl.pow(x, y)
```

Numba - PyROOT Example

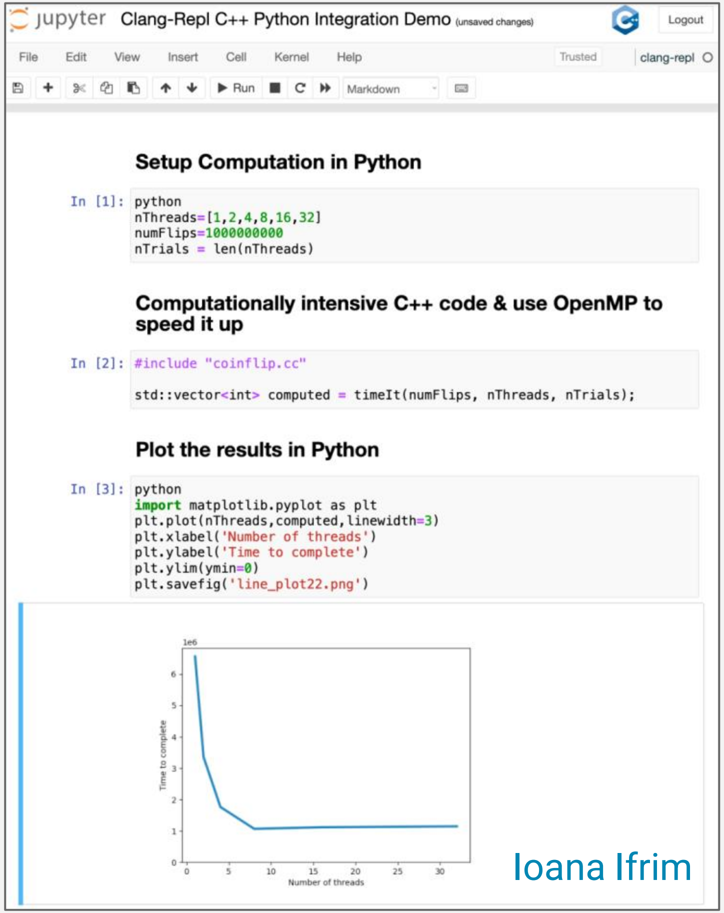
```
import numba
import math
import ROOT
import cppyy.numba_ext
# ⮝ Import the Numba extension
myfile=ROOT.TTree("vec_lv.root")
vector_of_lv=myfile.Get("vec_lv")
# ⮝ Vector of TLorentzVector

# ⮟ PyROOT pipeline
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 # ⮜ Numba decorator
def numba_calc_pt(lv):
    return math.sqrt(lv.Px()**2 +lv.Py()**2)

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

Pts = calc_pt_vec(vector_of_lv)
Pts = numba_calc_pt_vec(vector_of_lv)
```

When the traditional **PyROOT pipeline** is compared against the **Numba pipeline** in the above example we get a **17x** speedup. link

Ongoing Work

1. Maximize the C++ feature set supported in Numba.
2. Upstream libInterOp into LLVM master
3. Leverage Python-C++ interop in Jupyter using Cppyy. link



Conclusion

Tighter integration between Python and C++ can enable more efficient data analyses and is possible due to:

- Improved interoperability
- Optimizations in Cppyy/PyROOT via Numba
- Crosstalk between kernels in Notebook environments

Thank you