CppInterOp: Advancing Interactive C++ & Python for High Energy Physics

Aaron Jomy, CERN EP-SFT
Jonas Rembser, CERN EP-SFT
Vassil Vassilev, Princeton University

for the ROOT team

EP R&D Software Working Group Meeting
15.01.25
aaron.jomy@cern.ch

EP R&D

SFT

PRINCETON UNIVERSITY

Introduction

cppyy

cppyy
An automatic C++ - Python runtime bindings engine which powers ROOT's python interoperability

Cling
ROOT's interactive C++ interpreter, built on **LLVM** and **Clang**
Used in cppyy's(upstream) backend

Clang-REPL
A lightweight generalization of Cling in LLVM - supports interactive programming for C++ in a read-evaluate-print-loop (REPL) style

ROOT’s C++ Interpreter and Python: Motivation

To develop software solutions for HEP which are generalized for other sciences.

- *ROOT 6 developed SoA JIT compilation technology through Cling*
- *At the cost of in-house extensions to LLVM (~50 patches)*

The NSF funded certain developments in the area to generalize them for other sciences

- *In LLVM through the Compiler-Research initiative (Clang-REPL, CppInterOp)*

Ongoing R&D aims to implement similar solutions in ROOT to reduce the maintenance cost and improve the resilience of the HEP software ecosystem.

- *How?*

Challenges

ROOT and Python

- **Maintenance cost**
  - Keeping ROOT's fork of cppyy up to date, while supporting ROOT users Python code

- **The development of cppyy upstream and ROOT's fork diverged**
  - cppyy's fork of cling is patched for optimal python bindings
  - ROOT's cling does not necessarily ensure the same behavior

- **Compiler level API comes from ROOT's type system (ROOT meta)**
  - ROOT meta is essential for the reflection system that enables ROOT I/O
  - However, does not provide the best reflection API, as it was not designed with language bindings in mind (*Eg. Template instantiation*, *overload resolution*, *enums*)

ROOT meta

- ROOT uses LLVM API to drive its C++ reflection system (ROOT meta).
  This reflection system is used for I/O, as well as perform python bindings

- Over the years, ROOT's core/metacling system grew organically, and hinders the adoption of newer technologies like CUDA and advanced language interoperability (*Python*, *Julia*)

- **CppInterOp is a solution that leverages our experience into small, well-tested and versatile libraries that provides building blocks for *both* dictionaries and advanced language interoperability**

The solution we identified: CppInterOp

A finer-grained lightweight layer on top of LLVM/Clang that provides efficient, on-demand reflection, that drives language bindings generation.

The ability to be powered by multiple interpreters:

- ROOT's Cling
- LLVM's Clang-REPL

This opens the door to eventually upstream the interpreter into LLVM, bringing in efforts from the broader LLVM community, further reducing the maintenance cost.

Content

This presentation aims to showcase the latest developments in ROOT, and the potential of this technology through several use cases:

1. Reduction of in-house technical debt: Removing patches to LLVM by moving parts **upstream**
2. Providing better encapsulation of C++ reflection information in ROOT dictionaries via CppInterOp
3. Enabling cutting edge R&D in the domain of language bindings (Python, numba, Julia..)

Work Done and Impact

LLVM work trickling to ROOT

Reduction of patches by fixing existing LLVM issues.

Collaboration with Apple Engineers: *Exceptions on Apple Silicon*

Collaboration with Google Engineers: *Lazy Template Specialization Loading*

Ran ROOT use cases

Core Developments:

- Dynamic Library Manager upstreamed to LLVM: *S. Patildar*
  Upstream LLVM PR motivated by ROOT's use case
  https://github.com/llvm/llvm-project/pull/109913 ->
  Review suggested many design changes which are incorporated back into ROOT
  https://github.com/root-project/root/pull/17227

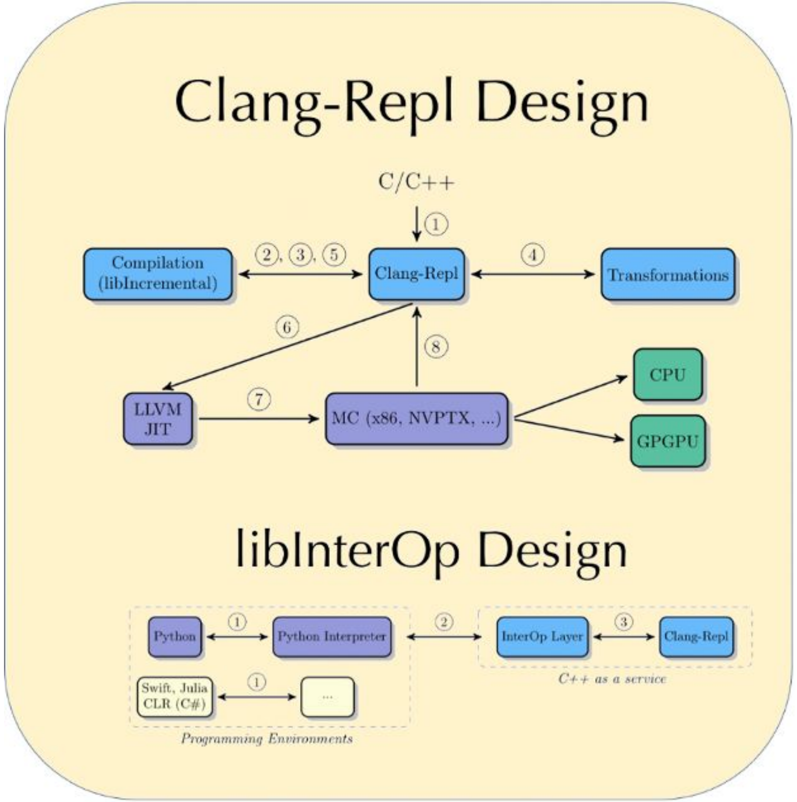
CppInterOp

CppInterOp exposes API from Clang and LLVM in a backward compatible way.

The API support downstream tools that utilize interactive C++ by using the compiler as a service.

This allows ROOT to embed Clang and LLVM as a libraries in their codebases.

The API are designed to be minimalistic and aid non-trivial tasks such as language interoperability on the fly.

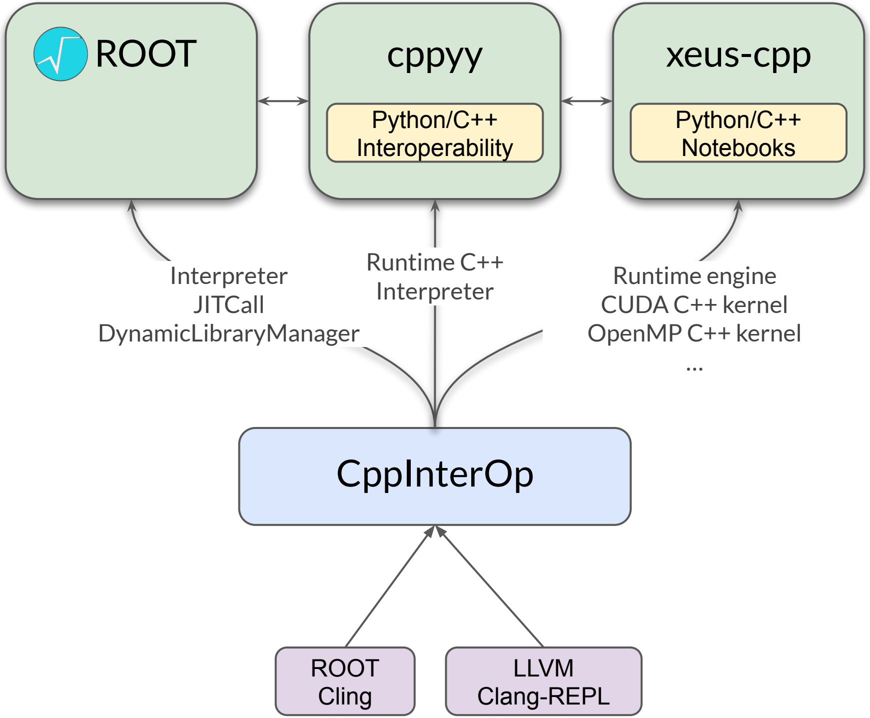


Integration with ROOT - Design

The adoption of CppInterOp in ROOT is underway and aims to abstract the interpreter infrastructure into LLVM.

Provides out-of-the box compatibility with **CUDA**, **OpenMP** and other parallel computing platforms

CppInterOp enables seamless utilization of *hardware accelerators* and other heterogeneous hardware



An illustration of a scientific workflow powered by CppInterOp

Define a function that updates a discrete Kalman filter cycle, using CUDA kernels for all matrix computations

```
std::vector<double> KalmanFilter::update(const std::vector<double>& y) {
    if (!initialized)
        throw std::runtime_error("Filter is not initialized!");

    // Discrete Kalman filter time update
    x_hat_new = matvecmulCUDA(A, x_hat);
    P = mataddCUDA(matmulCUDA(matmulCUDA(A, P), mattransposeCUDA(A)), Q);

    // Discrete Kalman filter measurement update
    std::vector<std::vector<double>> inv = matinverse(mataddCUDA(matmulCUDA(
    K = matmulCUDA(matmulCUDA(P, mattransposeCUDA(C)), inv);
    std::vector<double> temp = matvecmulCUDA(C, x_hat_new);
    std::vector<double> difference = vecsubCUDA(y, temp);
    std::vector<double> gain = K[0];
    for (size_t i = 0; i < x_hat_new.size(); i++) {
        x_hat_new[i] += matvecmulCUDA(K, difference)[i];
    }

    P = matmulCUDA(matsubCUDA(I, matmulCUDA(K, C)), P);

    x_hat = x_hat_new;
    t += dt;
```

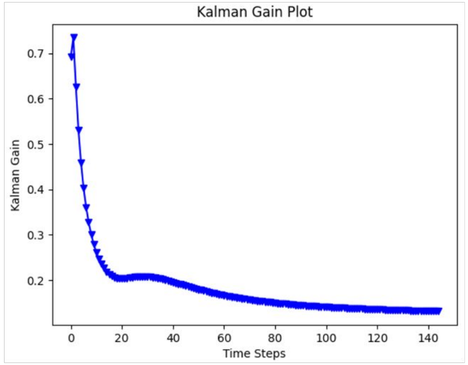
Load 1D projectile motion dataset in Python with pyyaml

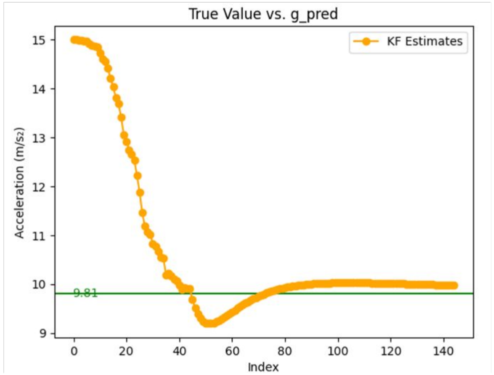
```
%%python

import yaml
import cppyy

with open('data/measurements.yml', 'r') as file:
    data_dict = yaml.safe_load(file)
    data_list = list(float(x) for x in data_dict['data'])

measurements_vector = cppyy.gbl.std.vector['double'](data_list)
```

Run the CUDA accelerated C++ function on the same data

```
std::vector<std::vector<double>> g_res = run_kf(true);
```

```
t = 0, x_hat[0]: 1.04203 0 -15
t = 0.0333333, y[0] = 1.04203, x_hat[0] = 1.04203 -0.5 -15
t = 0.0666667, y[1] = 1.10727, x_hat[1] = 1.08556 -0.0966619 -14.9988
t = 0.1, y[2] = 1.29135, x_hat[2] = 1.21317 0.720024 -14.9952
t = 0.133333, y[3] = 1.48485, x_hat[3] = 1.36865 1.21707 -14.9881
t = 0.166667, y[4] = 1.72826, x_hat[4] = 1.55548 1.60875 -14.9732
t = 0.2, y[5] = 1.74216, x_hat[5] = 1.66278 1.38374 -14.9637
```





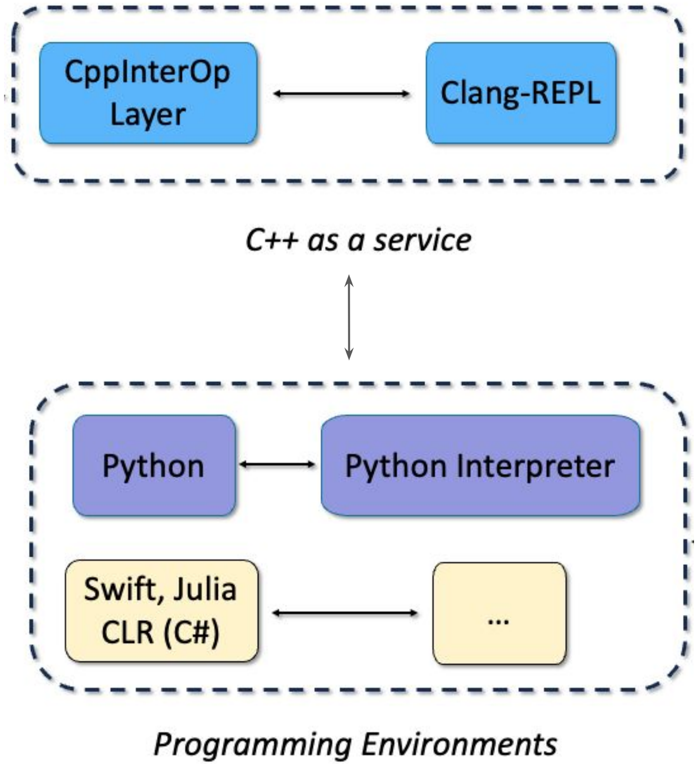
This ties in with on going work by L. Breitwieser, EP-SFT, enabling the capability to JIT CUDA code, another EP R&D initiative

Integration with ROOT - Language Bindings

CppInterOp enables dynamic C++ interactions with *multiple* languages and diverse computing environments like Jupyter

This is achieved by providing ROOT with:

- *a performant JIT, to incrementally compile C++ code*
- *a reflection API to drive bindings generation.*



The model we want

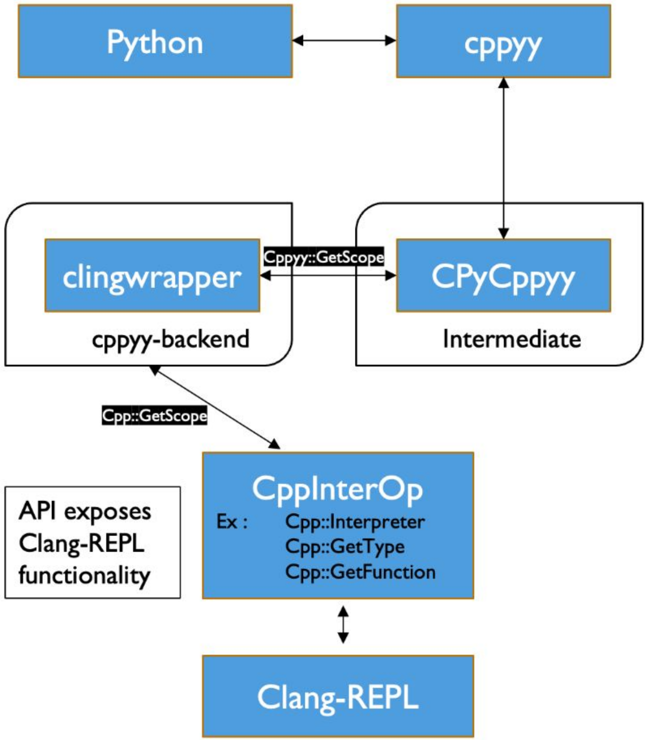
Ongoing development of a new cppyy based on CppInterOp, setting a standard for improved language bindings that ROOT can adopt.

Current progress: *335/504* passing test cases on Linux

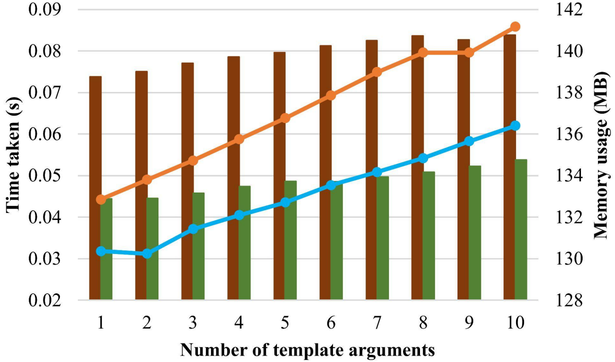
CppInterOp allows Cppyy to use LLVM's Clang-REPL as a runtime compiler inviting longer term sustainability

Opens up more C++ features that can be used by Cppyy users - *Eg. Partially specialized templates*

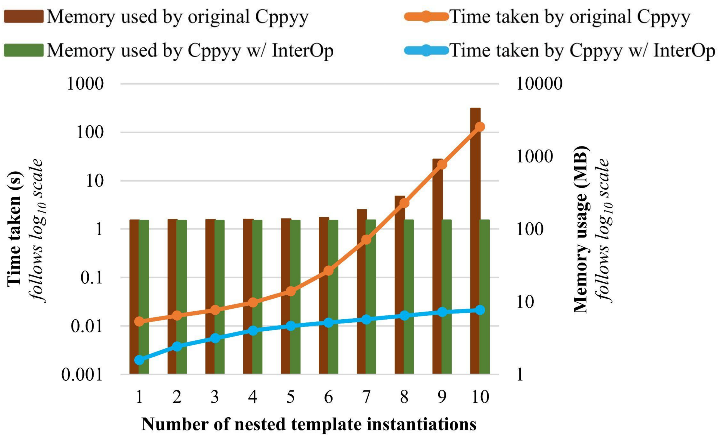
Lower dependencies leads to a performance improvement



Integration with cppyy



we compare template instantiations with std::tuple, where more arguments increase instantiation times

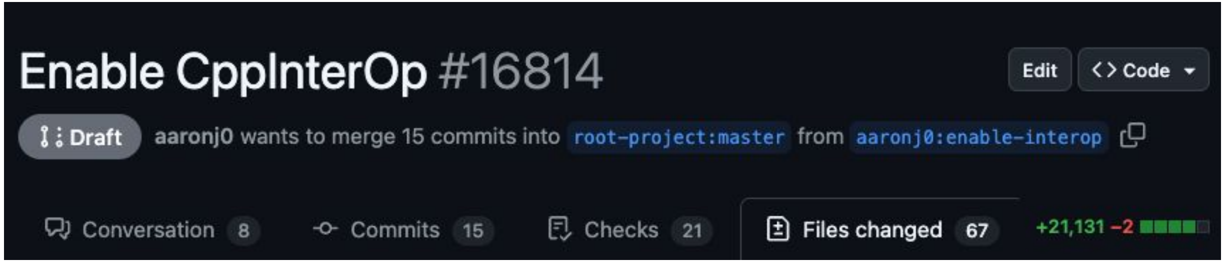


we compare nested templates like std::vector<...<std::vector>>, where cppyy instantiates from the innermost to the outermost layer

- Results without explicit optimisations show significant performance gains
- CppInterOp significantly improves cppyy in both time and memory for template instantiations.
- For `std::tuple` based multitype arrays:
  - CppInterOp is ***40% faster*** and ***4.5%* more *memory-efficient***.
  - Deeply nested templates show an **initial speedup of 6.2x**, tapering to 3.8x at 4 levels, with further scaling and memory gains.

Current Plan of Action

- PR enabling CppInterOp on ROOT is ready: https://github.com/root-project/root/pull/16814 and is stable on all platforms



- Next step is to incrementally tackle parts of ROOT's meta infrastructure with InterOp API, and propose the changes to LLVM for upstreaming
  - Eg. JITCall functionality- A function calling mechanism *(requires ~4-8 weeks)*

Current Plan of Action

- Short term benefits for ROOT:
  - Instantiating templates that ROOT currently cannot instantiate: https://github.com/root-project/root/issues/6481

```
class InheritTemplateFun: TemplateFun {
public:
    using TemplateFun::TemplateFun;
};
//Test
TClass *clInhTemplateFun =
TClass::GetClass("InheritTemplateFun");
   ASSERT_NE(clInhTemplateFun, nullptr);
```

Fails because `LookupHelper` *doesn't know how to instantiate function templates, even though at least the function template is made available to the derived class*

Current Plan of Action

- Short term benefits in ROOT:
  - Replace certain interfaces for more precise reflection, and therefore bindings.
  - Eg: `IsAggregate` sometimes returns true for non-aggregate classes like `std::tuple` *https://github.com/root-project/root/issues/16469*
    We can fix this bug today with the new clang-based API that CppInterOp provides

Current Plan of Action

- 100% test coverage of InterOp-based cppyy, and setting both upstream and ROOT to use this new and improved standard for python bindings *(ETA ~3 months)*

Future Opportunities: ROOT + Julia

This can allow ROOT to bind to the Julia runtime **non-invasively**, opening extensive avenues for R&D

- *we can prototype "ROOT.jl" by developing an automatic engine like cppyy for Julia*

Initial interest from the Julia community has led to several contributions to CppInterOp by a former developer of *Clang.jl*

- *https://github.com/Gnimuc/CppInterOp.jl*

Currently being used in JuliaPackaging/BinaryBuilder

- *https://github.com/JuliaPackaging/Yggdrasil/commits/master/L/libCppInterOp*

Summary

- *Puts in place a mechanism by which external expert effort is attracted to ROOT (LLVM compiler engineer community)*
- *The development of a new lightweight library on top of LLVM, driving improved reflection and language bindings*
- *The adoption of the CppInterOp library with ROOT, improving performance and stability of core libraries*
- *Redesign of cppyy based on InterOp, removing ROOT's forks of cppyy in the process*
- *Moving towards a patch-free LLVM for ROOT, and offloading the maintenance costs of the interpreter to LLVM*