Award #: OAC-1931408

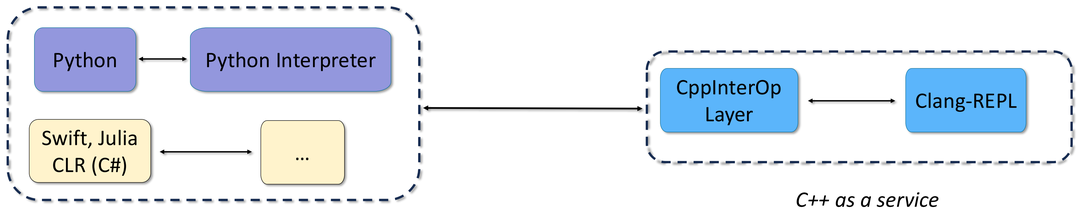
CppInterOp: Advancing Interactive C++ for High Energy Physics

Aaron Jomy[1,2]; Baidyanath Kundu[3]; Wim Lavrijsen[4]; Alexander Penev[5]; Vassil Vassilev[1,2]

[1]CERN, [2]Princeton University, [3]ETH Zurich, [4]Lawrence Berkeley National Lab, [5]University of Plovdiv

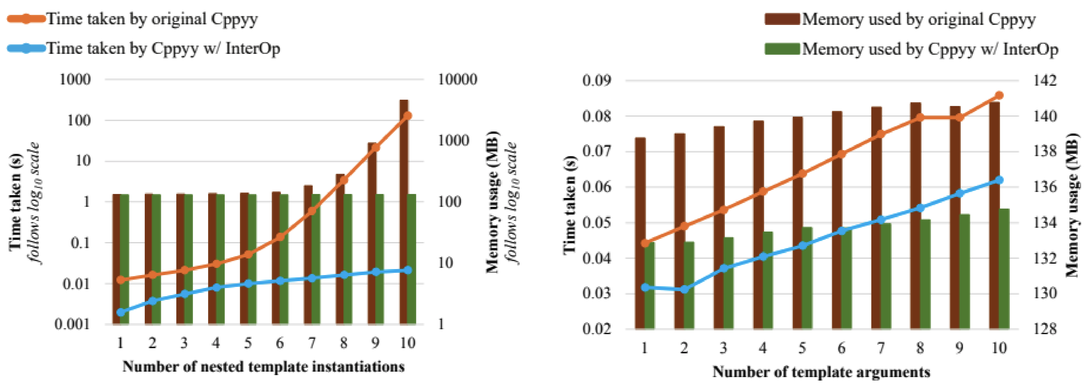
The Cling[3] C++ interpreter has transformed language bindings by enabling incremental compilation at runtime. This allows Python to interact with C++ on demand and lazily construct bindings between the two. The emergence of Clang-REPL as a potential alternative to Cling within the LLVM compiler framework highlights the need for a unified framework for interactive C++ technologies.

We present CppInterOp, a C++ Interoperability library, which leverages Cling and LLVM's Clang-REPL, to provide a minimalist and backward-compatible API facilitating seamless language interoperability. This provides downstream interactive C++ tools with the compiler as a service by embedding Clang and LLVM as libraries in their codebases. By enabling dynamic Python interactions with static C++ codebases, CppInterOp enhances computational efficiency and rapid development in high-energy physics. The library offers primitives enabling cppyy(PyROOT), an automatic, run-time, Python-C++ bindings generator. We also demonstrate CppInterOp's utility in diverse computing environments through its adoption as the runtime engine for xeus-cpp[4], a Jupyter kernel designed for C++.



CppInterOp focuses on enabling dynamic C++ interactions with multiple languages and diverse computing environments like Jupyter

It achieves this by providing other languages/environments with a performant JIT, to incrementally compile C++ code, while driving bindings generation using its reflection API.



On the left, we compare template instantiations with std::tuple, where more arguments increase instantiation times. On the right, we compare nested templates like std::vector<...<std::vector>>, where cppyy instantiates from the innermost to the outermost layer. These patterns are key to high-performance numerics libraries using template expressions.

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.

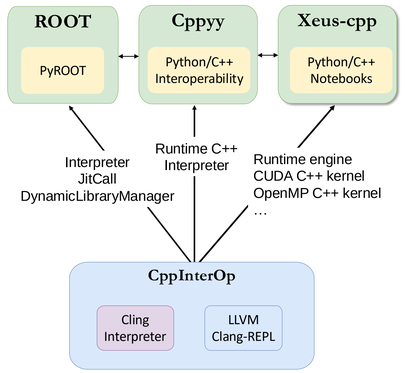
References

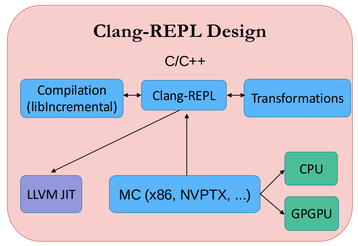
[1] The official repository for ROOT: analyzing, storing and visualizing big data, scientifically. https://github.com/rootproject/root

[2] ROOT: analyzing petabytes of data, scientifically. https://root.cern/

[3] V Vasilev, Ph Canal, A Naumann, and P Russo. Cling–the new interactive interpreter for ROOT 6. In Journal of Physics: Conference Series, volume 396, page 052071, 2012.

[4] Xeus is now a Jupyter subproject. https://blog.jupyter.org/xeus-is-now-ajupyter-subproject- c4ec5a1bf30b





The adoption of CppInterOp in ROOT[1, 2] is currently being tested and aims to abstract the interpreter infrastructure into LLVM. This also brings in efforts from the broader LLVM community upstream.

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

Cling enables data science in ROOT and is a core part of high energy physics analysis and discoveries. Generalized in LLVM as Clang-REPL, it allowed us to build a backward-compatible abstraction: CppInterOp

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

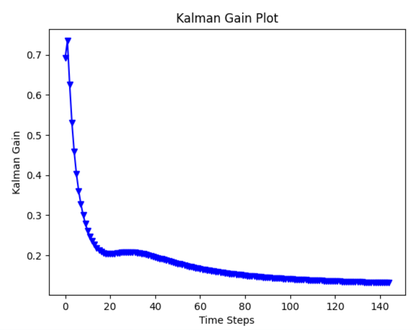
```
%%python

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)
```

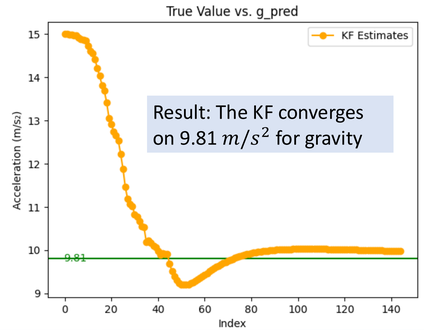
Run the CUDA accelerated C++ function on the same data

```
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
```

Plot in Matplotlib! This workflow is achieved in a notebook environment providing the best of all worlds





See more demos at our talk at the 2023 LLVM developers meeting:

Contact

GitHub: https://github.com/compiler-research/CppInterOp
Email: aaron.jomy@cern.ch, vassil.vassilev@cern.ch
Visit us at https://compiler-research.org