CSSI Element: C++ as a service - rapid software development and dynamic interoperability with Python and beyond

Princeton University: David Lange (PI), Ioana Ifrim, and Vassil Vassilev . Open-source contributors, students, interns: Parth Arora, Sara Bellei, Purva Chaudhari, Anubhab Ghosh, Matheus Izvekov, Manish Kausik, Sunho Kim, Baidyanath Kundu, Tapasweni Pathak, Rohit Rathaur, Garima Singh, Roman Shakhov, Surya Somayyajula, Jun Zhang

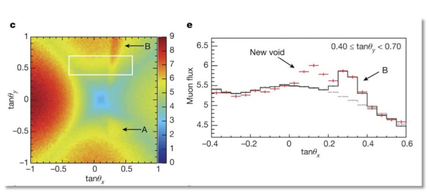
Award #: OAC-1931408

Project Goals

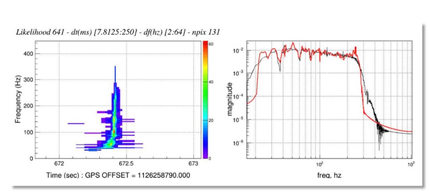
CaaS aims to provide programmers and data scientists a simple and general solution to language interoperability:

- Advance interpretative technology to provide scientists a state-of-the-art C++ execution environment
- Enable functionality to provide dynamic, native-like, runtime interoperability between C++ and Python
- Allow seamless utilization of heterogeneous hardware (e.g., hardware accelerators)
- To enable rapid application development even with a complex codebase

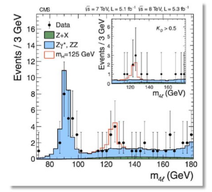
Our approach is to generalize a high-energy physics analysis tool ("Cling") to a generally accessible and fully functional tool that is part of LLVM/Clang.



[1]



[2]



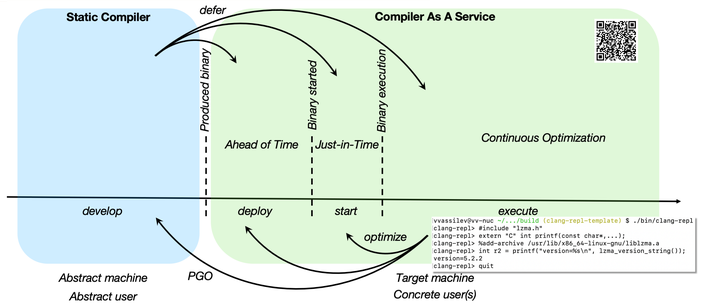
[3]

Scientific breakthroughs such as the discovery of the large void in the Khufu's Pyramid, gravitational waves and the Higgs boson heavily rely on the ROOT software package

[1] K. Morishima et al, **Discovery of a big void in Khufu's Pyramid by observation of cosmic-ray muons**, *Nature, 2017*
[2] Abbott et al, **Observation of gravitational waves from a binary black hole merger**. *Physical review letters, 2016*
[3] CMS Collab, **Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC**. *Physics Letters B, 2012*

Project Accomplishments

- LLVM community engagement / acceptance of CaaS concept and approach
- Initial release of Clang-Repl achieved in LLVM13
- Clang-Repl based plugin (Clad) implemented and demonstrated including offload of calculations to GPU
- LibInterop design completed after extensive community discussion. Now co-developing with application developers including
  - CPPYY package enabling run-time python <-> C++ bindings
  - Xeus based Jupyter plugin supporting interoperability and data exchange between C++ and python
- Science applications include automatic differentiation, uncertainty quantification, and embedded device control



Project Results and Applications

Proposed CaaS programing model now realized in prototype

jupyter Clang-Repl C++ Python Integration Demo (unsaved changes) Logout

File Edit View Insert Cell Kernel Help Trusted clang-repl

Run Markdown

Setup Computation in Python

```
In [1]: python
nThreads=[1,2,4,8,16,32]
numFlips=1000000000
nTrials = len(nThreads)
```

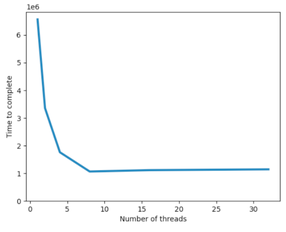
Computationally intensive C++ code & use OpenMP to speed it up

```
In [2]: #include "coinflip.cc"

std::vector<int> computed = timeIt(numFlips, nThreads, nTrials);
```

Plot the results in Python

```
In [3]: python
import matplotlib.pyplot as plt
plt.plot(nThreads,computed,linewidth=3)
plt.xlabel('Number of threads')
plt.ylabel('Time to complete')
plt.ylim(ymin=0)
plt.savefig('line_plot22.png')
```



CPPYY run-time bindings are first libInterop use case

Basic Performance Test: empty call

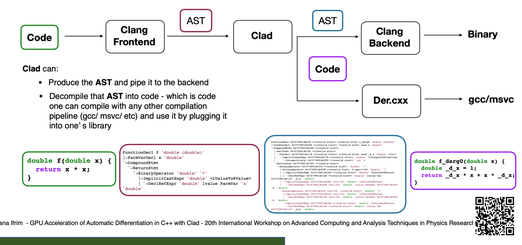
Tool	Execution time (ns/call)*
C++ (Cling w/ -O2; out-of-line)	1.5
cppyy / pypy-c	16
swig (builtin)	27
cppyy / CPython	68
pybind11	68
swig (default)	104

⇒ Empty global function call is a pure overhead measure (zero work)
⇒ pypy-c slower than C++ b/c of global interpreter lock (GIL) release
⇒ "Builtin" swig trades functionality for speed
⇒ There is no obvious benefit to "static" over runtime bindings

(*) lower is better

CLAD: Source transformation Automatic Differentiation

Clad is a Clang Plugin transforming the AST of the supported languages : C++, CUDA, C, ObjC



Ioana Ifrim - GPU Acceleration of Automatic Differentiation in C++ with Clad - 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research

Clad AD applied to statistical analysis problems

ROOT is a data analysis software package used to process data in the field of high-energy physics.

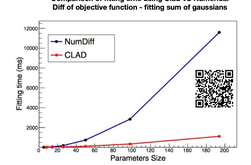
Clad has replaced numerical gradient calculations for formula based functions.

The Clad gradient is then used to compute the gradient of the objective function (χ^2 or negative log-likelihood function) when fitting

$$\chi^2 = \sum_{i=1}^{N} \frac{(Y_i - f(x_i, \mathbf{p}))^2}{\sigma_i^2}$$

Thus, ROOT fitting class computes $\nabla_{\mathbf{p}}(\chi^2)$ from $\nabla_{\mathbf{p}}(f(x, \mathbf{p}))$ obtained using Clad

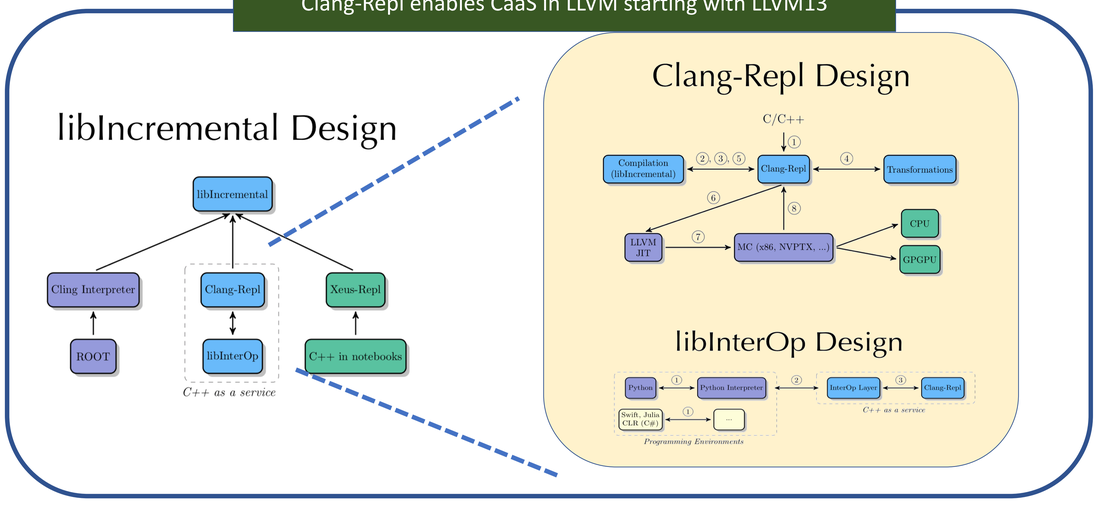
Clad Hessian Mode in ROOT (GSoC 2021- Baidyanath Kundu)



Lorenzo Moneta 94th ROOT PPP Meeting

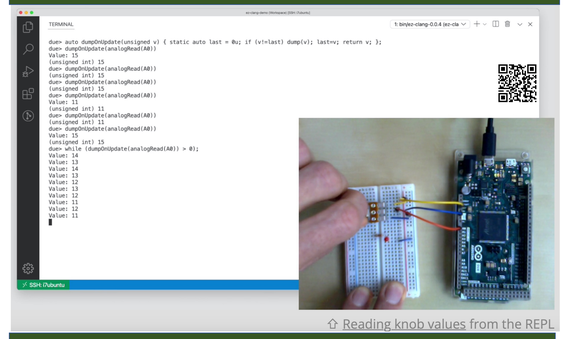
* current implementation still requires one numerical gradient call for second derivatives (when seeding) - higher speedups will be possible when introducing second derivatives computation using Clad

Clang-Repl enables CaaS in LLVM starting with LLVM13



Thanks to this project, we have grown a diverse user community around our technology including contributors from data science and industry. We established a monthly community meeting series to discuss results and applications. Visit us at https://compiler-research.org

EZ-Clang: Extremely small JIT for embedded devices



Compiler driven Uncertainty Quantification

Case Study: Simpson's Rule

Results

Precision configurations	Absolute Error	Clad's Estimated Upperbound	Variables in lower precision (out of 11)
10-byte extended precision (*long double*)	4.07e-14	3.1e-12	0
Clad's mixed precision	4.08e-14	3.0e-12	6
IEEE double-precision (*double*)	6.8e-11	6.2e-9	-
IEEE single-precision (*float*)	0.038	3.31	-

"Demoting" low-sensitivity variables to lower precision improves performance by ~10% in this example.

Clad's estimate also agrees that there is no significant change in the final error. This can be useful in the cases where an accurate ground-truth comparison is not available.

```
long double f(long double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}

long double simpsons(double a, double b) {
int n = 1000000;

double h = (b - a) / (2.0 * n);
long double x = a;
double tmp;
double fa = f(a), fb = f(b);
long double s1 = fa;

for(int i = 0; i < n; i++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

V. Vassilev, G. Singh, *Floating-Point Error Estimation Using AD*, SIAM UQ22