*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.*

# C++ as a service — rapid software development and dynamic interoperability with Python and beyond

# Interactive C++: Showcase

Vassil Vassilev

*20.09.2023*

# Outline

- Motivation
- Exploratory programming with C++. Compiler-As-A-Service
- Project Goals and Implementation
  - Support For Incremental Compilation
  - Language Interoperability
  - Heterogeneous Hardware Support
- Tutorials and Community Outreach
- Live Demo
- Future Work

# Motivation

- The C++ programming language is used for many numerically intensive scientific applications.
- C++ is often seen as difficult to learn and inconsistent with rapid application development
- The use of new programming languages has grown steadily in science and in fact Python is the language of choice for data science and application control but its computational performance is mediocre

Is there a way to combine the expressiveness of Python and the power of C++?

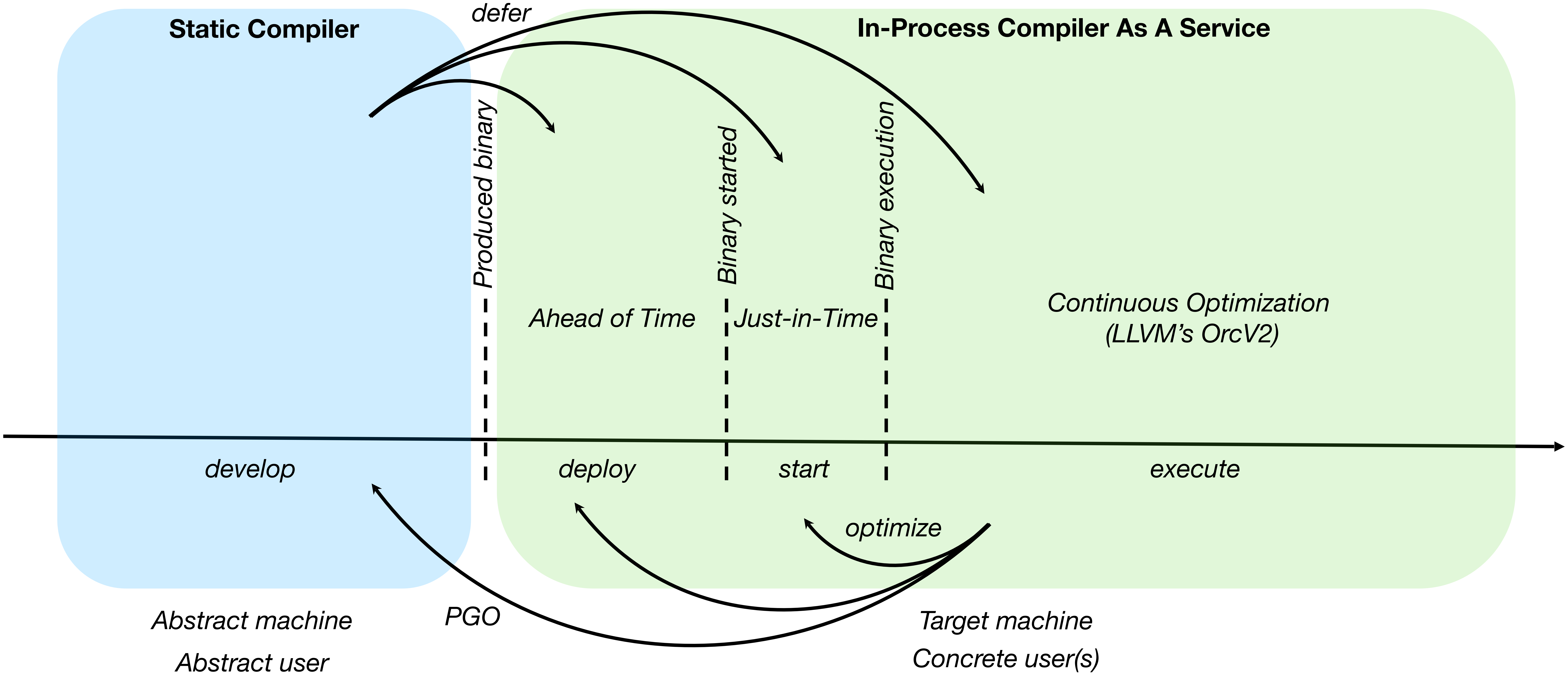
# Exploratory programming with C++

# Interactive C++. Key Insights

- Incremental Compilation
- Handling errors
  - Syntactic
  - Semantic
- Execution of statements
- Displaying execution results
- Entity redefinition

```
[cling] #include <vector>
[cling] std::vector<int> v = {1,2,3,4,5};

[cling] std.sort(v.begin(), v.end());
input_line_1:1:1: error: unexpected namespace
name 'std': expected expression
std.sort(v.begin(), v.end());
^

[cling] std::sort(v.begin(), v.end());
[cling] v // No semicolon
(std::vector<int> &) { 1, 2, 3, 4, 5 }

[cling] std::string v = "Hello World"
(std::string &) "Hello World"
```

# Compiler (C++) As A Service



# CaaS. Programming Model

```
/// Call an interpreted function using its symbol address.
void callInterpretedFn(cling::Interpreter& interp) {
  // Declare a function to the interpreter. Make it extern "C"
  // to remove mangling from the game.
  interp.declare("#pragma cling optimize(1)"
      extern \"C\" int cube(int x) { return x * x * x; }");
  void* addr = interp.getAddressOfGlobal("cube");
  using func_t = int(int);
  func_t* pFunc = cling::utils::VoidToFunctionPtr<func_t*>(addr);
  std::cout << "7 * 7 * 7 = " << pFunc(7) << '\n';
}
```

```
// caas-demo.cpp
// g++ ... caas-demo.cpp; ./caas-demo
int main(int argc, const char* const* argv) {
  cling::Interpreter interp(argc, argv, LLVMDIR);

  callInterpretedFn(interp);
  return 0;
}
```

```
vvassilev@vv-nuc ~/.../builddir $ ./caas-demo
7 * 7 * 7 = 343
vvassilev@vv-nuc ~/.../builddir $
```

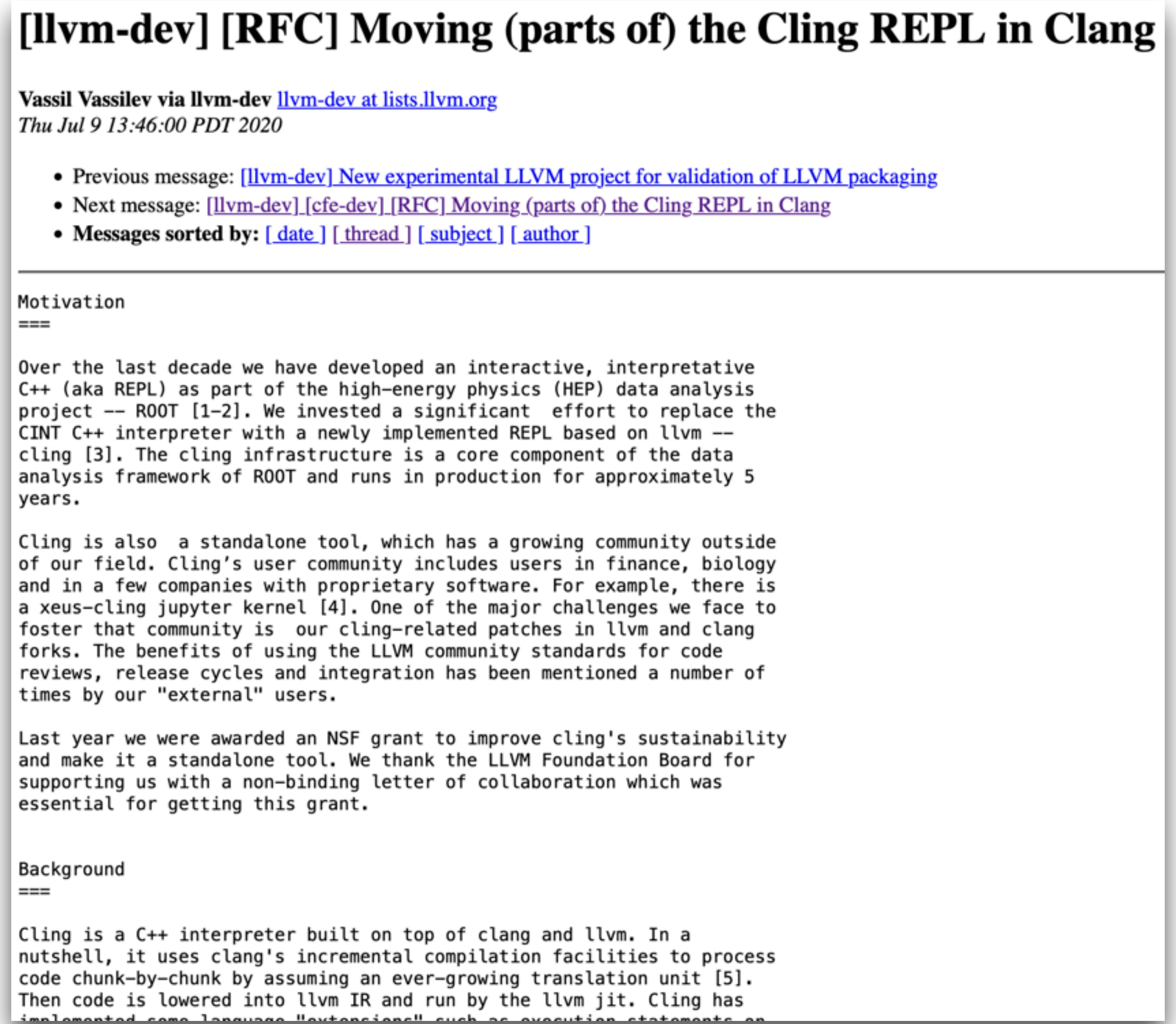
# Project Goals and Implementation

Leverage the infrastructure developed in the field of high energy physics and make it available to other scientific domains via LLVM and open source.

# Support For Incremental Compilation

# Support For Incremental Compilation

Positive outcome for our LLVM community reachout. Adapting mainline LLVM infrastructure started shortly after.



## [llvm-dev] [RFC] Moving (parts of) the Cling REPL in Clang

**Vassil Vassilev via llvm-dev** llvm-dev at lists.llvm.org
*Thu Jul 9 13:46:00 PDT 2020*

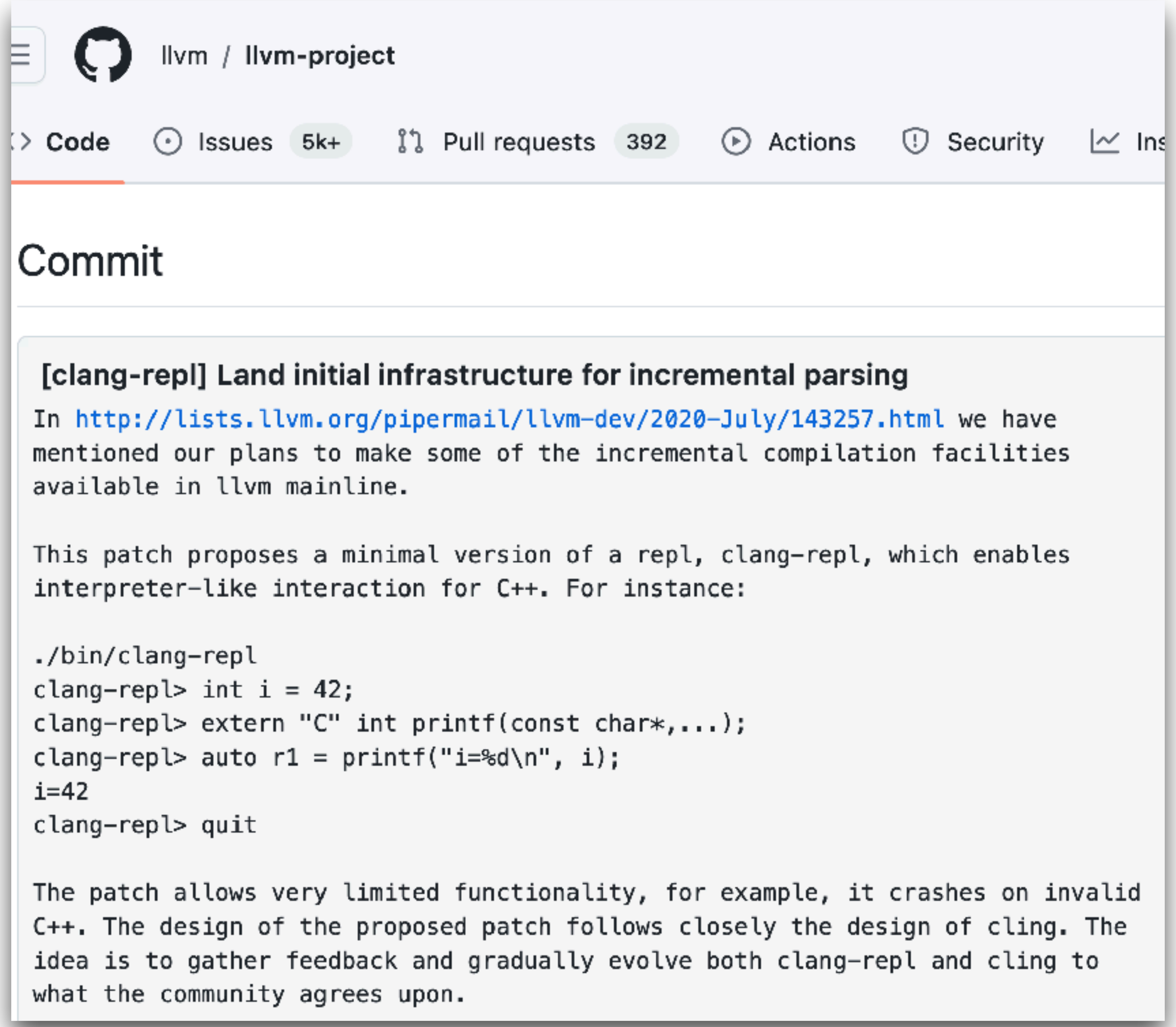
- Previous message: [llvm-dev] New experimental LLVM project for validation of LLVM packaging
- Next message: [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang
- **Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

```
Motivation
===

Over the last decade we have developed an interactive, interpretative
C++ (aka REPL) as part of the high-energy physics (HEP) data analysis
project -- ROOT [1-2]. We invested a significant  effort to replace the
CINT C++ interpreter with a newly implemented REPL based on llvm --
cling [3]. The cling infrastructure is a core component of the data
analysis framework of ROOT and runs in production for approximately 5
years.

Cling is also  a standalone tool, which has a growing community outside
of our field. Cling's user community includes users in finance, biology
and in a few companies with proprietary software. For example, there is
a xeus-cling jupyter kernel [4]. One of the major challenges we face to
foster that community is  our cling-related patches in llvm and clang
forks. The benefits of using the LLVM community standards for code
reviews, release cycles and integration has been mentioned a number of
times by our "external" users.

Last year we were awarded an NSF grant to improve cling's sustainability
and make it a standalone tool. We thank the LLVM Foundation Board for
supporting us with a non-binding letter of collaboration which was
essential for getting this grant.


Background
===

Cling is a C++ interpreter built on top of clang and llvm. In a
nutshell, it uses clang's incremental compilation facilities to process
code chunk-by-chunk by assuming an ever-growing translation unit [5].
Then code is lowered into llvm IR and run by the llvm jit. Cling has
```

# Support For Incremental Compilation. Clang-Repl

Initial version of the incremental compilation infrastructure landed in LLVM and was released in LLVM 13. Gradual improvements in each release. Currently LLVM 17.

Since LLVM 13, approximately 30 developers have contributed in that area.



# Clang-Repl Helped Upstreaming Tech. Debt

Clang-Repl provided an environment which helps explain and test the custom patches developed in the domain of High-Energy Physics (HEP).

- During the project we have upstreamed the essential patches relevant for incremental compilation
- That lead to faster llvm upgrade cycles in HEP. Time for upgrades went down from approximately 1 year (llvm5->llvm9) to several months from (llvm9->llvm13) to several weeks (llvm13->llvm16).

T. Pathak

J. Zhang

J. Hahnfeld

# Developments Related to Clang-Repl

Clang-Repl drove several new developments:

- Automatic completion at the prompt improving the overall user user experience
- Implement shared memory manager for JITLink enabling efficient out-of-process execution to improve system stability
- Implement JITLink backends for aarch64, ppc, windows to merge the linking layers of the static linker and the JIT for improved performance and reliability.

Y. Fu

A. Ghosh

S. Kim

# Clang-Repl in Jupyter

A. Ghosh

A. Penev

Clang-Repl regular release schedule and packaging together with standard LLVM enabled easier adoption in the Jupyter system:

- Xeus-Clang-Repl enables incremental C++ with interoperability extensions in Jupyter
- Xeus-Cpp enables Clang-Repl in JupyterLite
- WebAssembly-based Clang-Repl in JupyterLite

S. Corlay

I. Ifrim

# Automatic Language Interoperability

# Clang-Repl in Jupyter

Crossing the language barrier is expensive

Our Compiler-As-A-Service Approach solves that

```
In [1]: struct S { double val = 1.; };
```

```
In [2]: from libInterop import std
        python_vec = std.vector(S)(1)
```

```
In [3]: print(python_vec[0].val)

        1
```

```
In [4]: class Derived(S)
            def __init__(self):
                self.val = 0
        res = Derived()
```

```
In [5]: __global__ void sum_array(int n, double *x, double *sum) {
          for (int i = 0; i < n; i++) *sum += x[i];
        }
        // Init N=1M and x[i] = 1.f. Run kernel on 1M elements on the GPU.
        sum_array<<<1, 1>>>(N, x, &res.val);
```

*compiler-research.org's Compiler-As-A-Service Project Final Goal. Shown in the live demo.*

# Automatic Language InterOp. Python

## Performance Compared to Static Approaches

- No fundamental CPU performance difference

Note carefully that *everything* in Python is runtime: compile-time just means that the bindings *recipe* is compiled, not the actual bindings themselves!

- But heavy Cling/LLVM dependency:
  - ~25MB download cost; ~100MB memory overhead
  - Complex installation (and worse build)

- 24 -

## Basic Performance Test: overload

| Tool | Execution time (ms/call)* |
|---|---|
| C++ (Cling w/ -O2; out-of-line) | 1.8E-6 |
| cppyy / pypy-c | 0.50 |
| cppyy / CPython | 1.25 |
| swig (builtin) | 1.29 |
| swig (default) | 4.23 |
| pybind11 | 6.97 |

⇒ C++ overload is resolved at compile time, not based on dynamic type
⇒ Largest overhead: Python instance type checking (avoidable, but clumsy)
⇒ There is no obvious benefit to "static" over runtime bindings

(*) lower is better

- 27 -

W. Lavrijsen, LBL, cppyy, Sep 2021, compiler-research.org

The approach does not require the project maintainer to bother providing static bindings

# CPPY In Brief

W. Lavrijsen

- A CPython/PyPy Extension using their C API
- Automatic, on-demand mapping of Python to C++ concepts
- Incredible piece of art and engineering, often neglected
- Relies on on-demand reflection information provided by the heavy ROOT framework.

# Moving CPPYY Closer To The LLVM Orbit

Replacing the cppyy backend with a specialized and more robust InterOp layer yields:

- Easier adoption of newer LLVM versions (CUDA, C++ standards)
- Easer implementation of new features
- Better release cycle
- Wider adoption

# CPPYY Backend Redesign With CppInterOp

Every unsuccessful lookup can be completed by a C++ entity connected to a python class wrapper.

```
val = std.vector[int]((1,2,3))
```

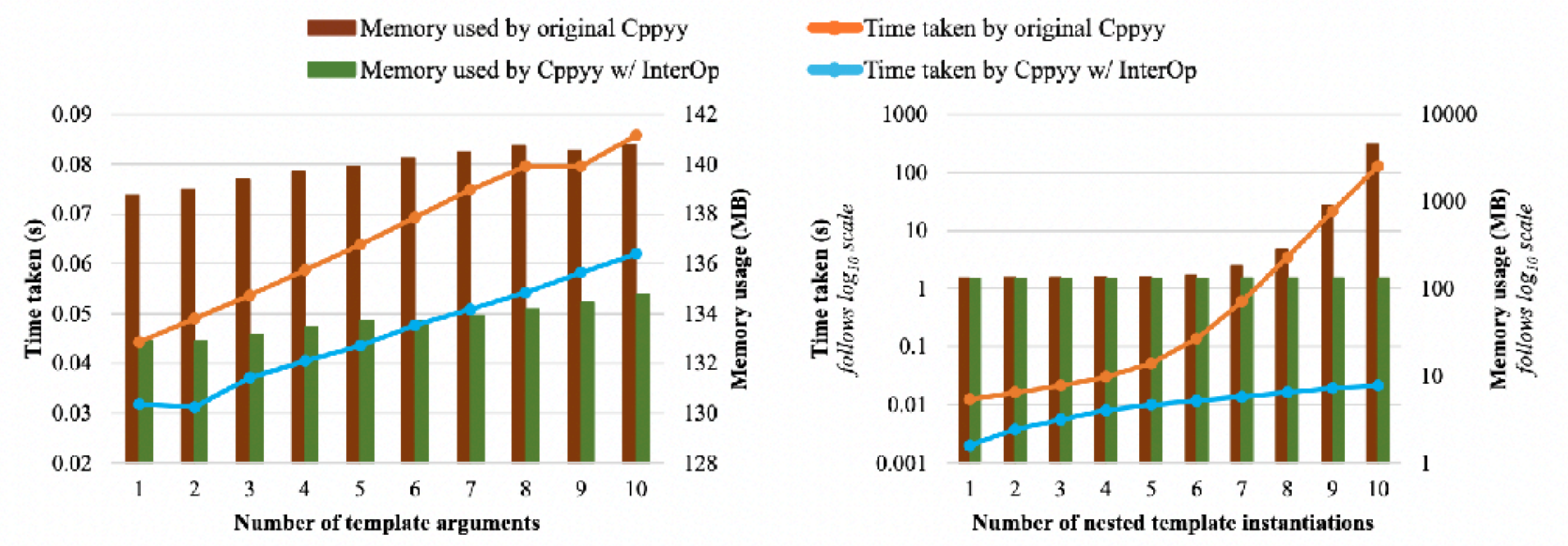



**Figure 3. Time taken and memory used during class template instantiation.** On the left, we compare template instantiations with `std::tuple<double, double, ...>` where the number of template instantiations done by the C++ interpreter increases with the number of template arguments. On the right, we compare instantiating nested templates, for example, `std::vector<...<std::vector<double> >`, where cppyy has to instantiate each nesting individually from the innermost to the outermost class template. These are common features of high-performance, templated numerics libraries that utilize template expressions.

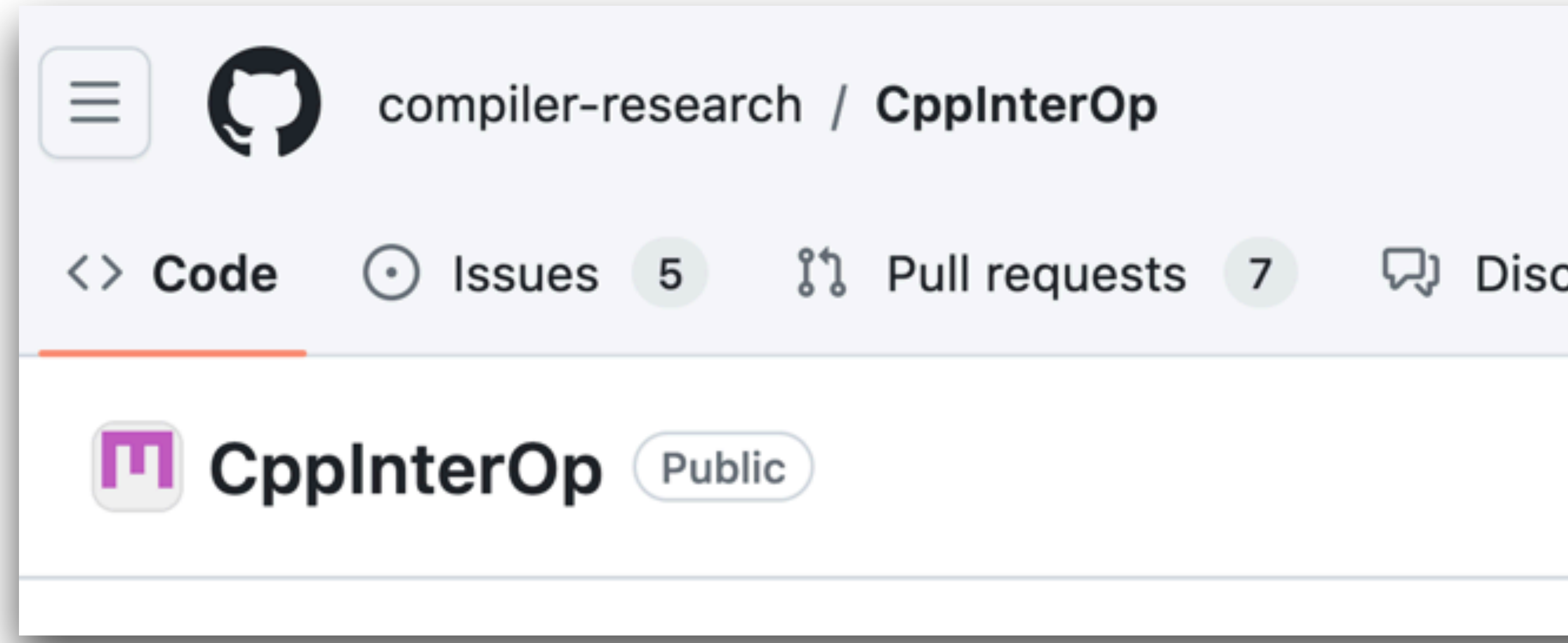
While parsing we can associate each construct with a C++ entity

# CppInterOp — Clang-based Language InterOp Library

B. Kundu

The goal was to create a C++ language interoperability layer allowing efficient automatic bindings with Python but also for D, Julia, etc…

- Created a document which describes prior art (cppyy and cxx.jl) and enumerates key features
- Implemented a proof of concept which is able to instantiate a C++ template on the fly from within Python
- Connected the Cppyy backend to CppInterOp

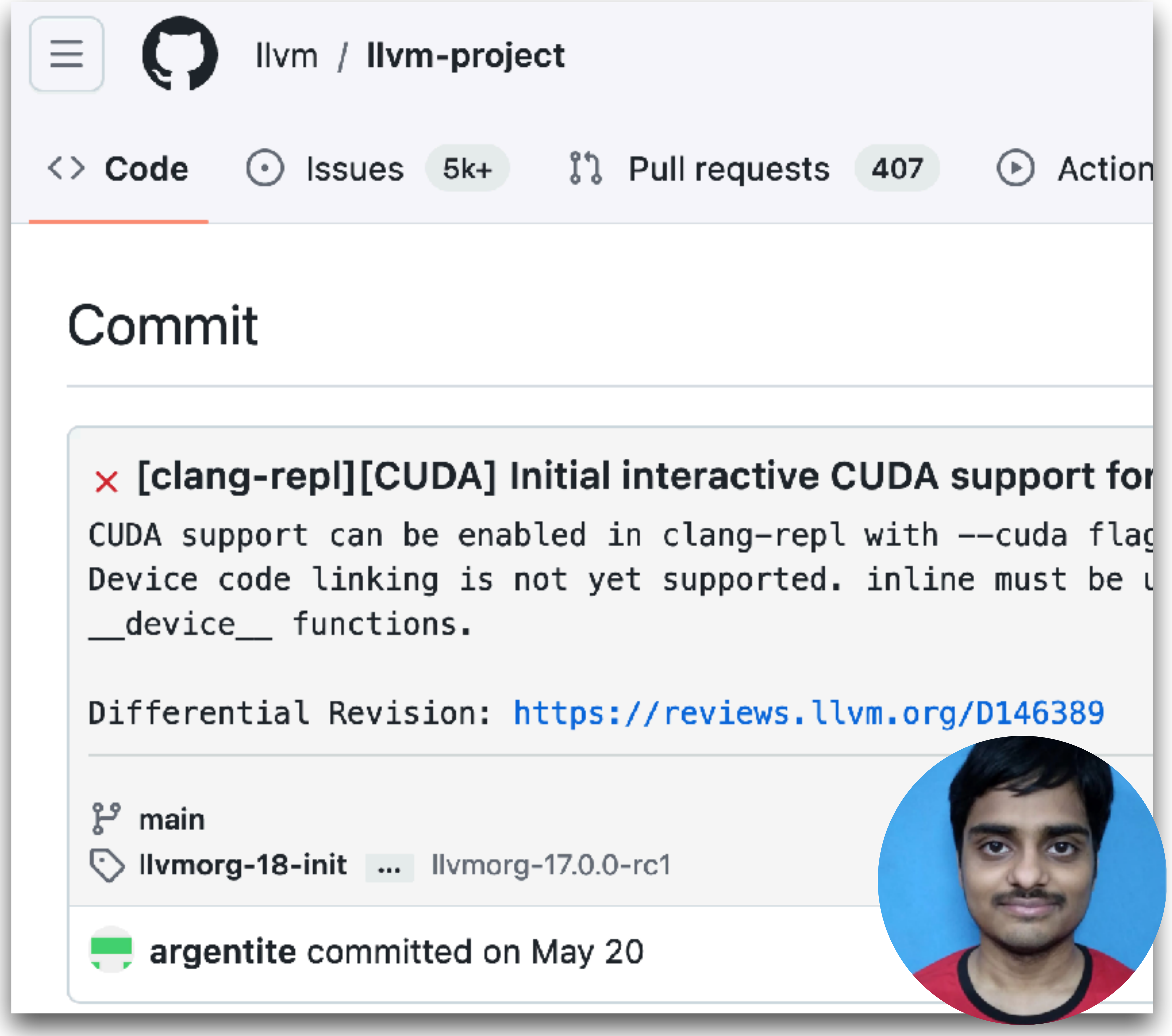


# Heterogeneous Hardware Support

# Design And Develop Interactive CUDA Support

Implemented a novel approach in interpreting CUDA codes where the PTX is passed through the virtual file system

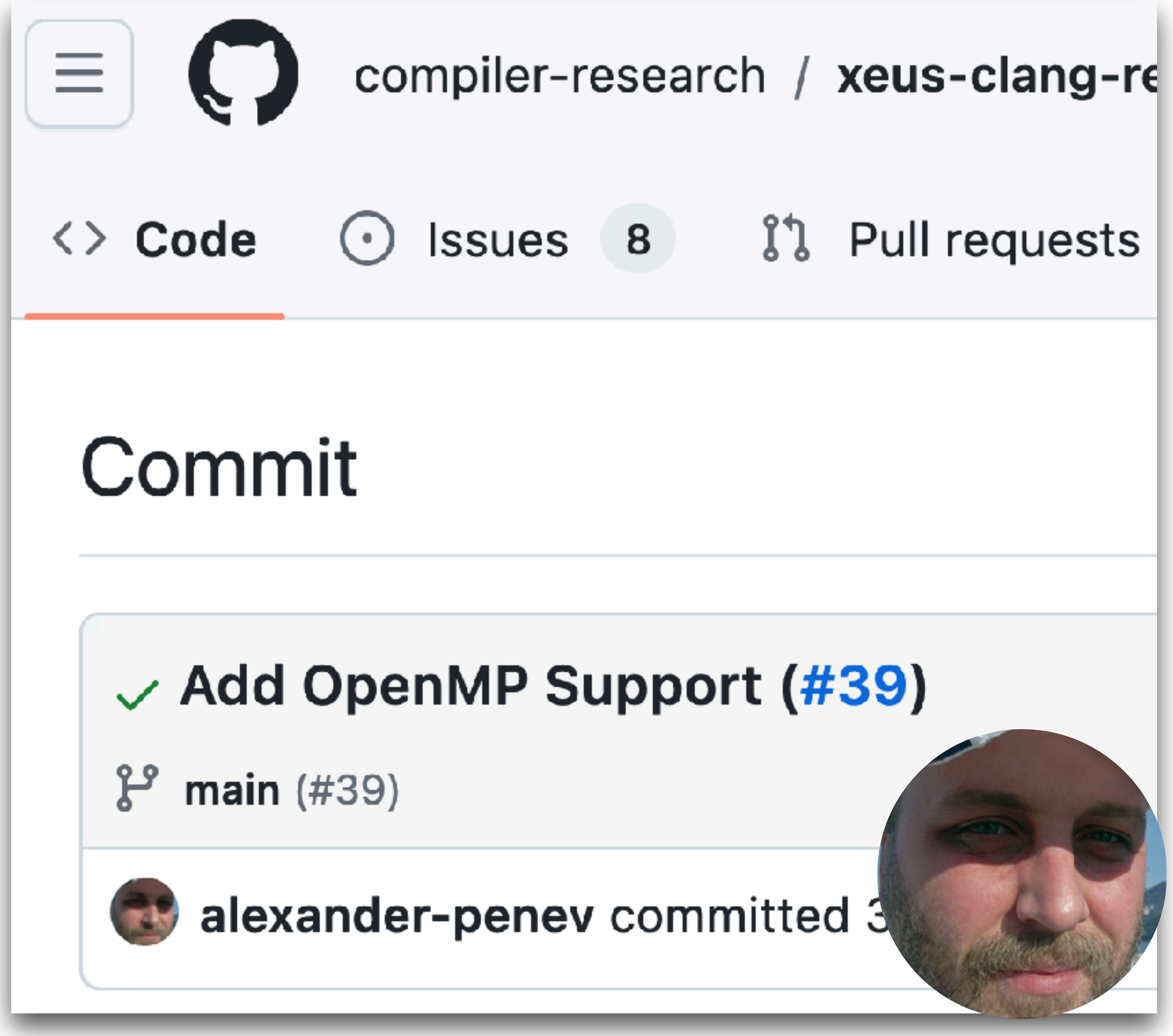
The CUDA engine in Clang-Repl helped discover issues in the mainstream CUDA support in Clang.




A. Ghosh

# Design And Develop Interactive OpenMP Support

Implemented OpenMP support in Clang-Repl and Jupyter




A. Penev

# Support New Architectures In JITLink

JITLink is a library for JIT Linking. That is a component enabling re-use of LLVM as an in-memory compiler by adding an in-memory link step to the end of the usual compiler pipeline.


S. Kim

- Develop Windows Support (COFF)
- Develop ARM64 Unix Support (Aarch64)
- Develop ARM32 Unix Support based on our ARM64 infrastructure — external contribution
- Develop PowerPC Support (ppc64) — contributed by IBM

# Tutorials & Community Outreach

# Community Outreach

https://compiler-research.org/vacancies/
https://compiler-research.org/team/
https://compiler-research.org/meetings/

- Open, Virtual Weekly Team Meetings
- Open, Virtual Monthly Meetings
  - 13 invited talks by speakers from institutions such as Apple, HZDR, QuantStack, Max-Planck, LBL, CERN and EA
- Student mentoring
  - 2 Unpaid Contributors
  - 2 CERN Interns
  - 4 IRIS-HEP Fellows
  - 15 Google Summer of Code
- 3 Technical Documentation Writers via Google Season of Docs

# Community Outreach. Presentations

https://compiler-research.org/presentations

- https://compiler-research.org/presentations/#VVACAT2022ACAT 2022, V Vassilev, Invited talk
- Using C++ From Numba, Fast and Automatic, PyHEP 2022, B Kundu
- Enabling Interactive C++ with Clang, LLVM Developers' Meeting 2021, V Vassilev
- Estimating Floating-Point Errors Using Automatic Differentiation, SIAM UQ 2022, V Vassilev, G Singh
- Interactive C++ for Data Science, CppCon21, V Vassilev
- Differentiable Programming in C++, CppCon21, W Moses, V Vassilev

# Community Outreach. Publications

https://compiler-research.org/publications

- B Kundu, V Vassilev, W Lavrijsen, Efficient and Accurate Automatic Python Bindings with cppyy & Cling (2023)
- G Singh, B Kundu, H Menon, A Penev, et. al., Fast And Automatic Floating Point Error Analysis With CHEF-FP (2023)
- G Singh, J Rembser, L Moneta, D Lange, et. al., Automatic Differentiation of Binned Likelihoods With Roofit and Clad (2023)
- I Ifrim, V Vassilev, D Lange, GPU Accelerated Automatic Differentiation With Clad arXiv preprint arXiv:2203.06139 (2022)
- M Foco, M Rietmann, V Vassilev, M Wong, et. al., P2072R0: Differentiable programming for C++ (2020)

# Community Outreach. Tutorials

https://compiler-research.org/tutorials

- S Kim, Lang Hames, V Vassilev (Princeton/CERN), Building Programming Language Infrastructure With LLVM Components (2023-07-17)
- Simeon Ehrig, Game of Life on GPU Using Cling-CUDA (2021-11-09)
- Garima Singh, Floating-Point Error Estimation Using Automatic Differentiation with Clad (2021-08-21)
- Ioana Ifrim, Interactive Automatic Differentiation With Clad and Jupyter Notebooks

# Live Demo

# Demo1. Project Motivation Mockup

- C++: Create a C++ Struct `S`
- Python: Create a wrapper class over std::vector instantiated with `S`
- Python: Print the value of `S`
- Python: Derive from `S`
- CUDA: Perform a sum over array and record the result into res.

In [1]:
```
struct S { double val = 1.; };
```

In [2]:
```
from libInterop import std
python_vec = std.vector(S)(1)
```

In [3]:
```
print(python_vec[0].val)
```

```
1
```

In [4]:
```
class Derived(S)
    def __init__(self):
        self.val = 0
res = Derived()
```

In [5]:
```
__global__ void sum_array(int n, double *x, double *sum) {
  for (int i = 0; i < n; i++) *sum += x[i];
}
// Init N=1M and x[i] = 1.f. Run kernel on 1M elements on the GPU.
sum_array<<<1, 1>>>(N, x, &res.val);
```

# Demo2. OpenMP Hello World

M. Vassilev

- Run OpenMP codes in Jupyter

A. Penev

# Demo3. Python/C++ InterOp: Eigen

- C++: Use the Eigen template math library to define operations
- Python: Instantiate an eigen matrix class with python type

A. Jomy

# Demo4. CUDA Vector Addition Demo

- Run vector add in CUDA

A. Penev

# Demo5. Python/C++/CUDA InterOp: Kalman Filter

- C++: Use the Eigen template math library to define operations
- Python: Instantiate an eigen matrix class with python type

A. Jomy

# Demo6. JupyterLite

- Demonstrate Clang-Repl in browser

A. Ghosh

# Impact on Science & Education

The project developed compiler-based components for data science which helped:

- Connect domain experts with compiler engineers
- Simplify data science infrastructure in the field of High-Energy Physics
- Improve Julia-based workflows via the JitLink developments
- Improve stability in the ppc area useful for Numba/Numpy
- Offer Jupyter-based education environment to study parallel technologies such as OpenMP and CUDA
- Build an open, multicultural environment for advancing students' skills in engineering in LLVM and related software

# Broader Impact

The project developed technical and human capital in the intersection of compiler and data science. It connected domain scientists to the LLVM community via core technologies fostering synergies and collaborations with industry.

The project helped develop 27 young professionals from 11 different countries some of who went to prestigious academic and industrial companies such as UCSD, ETH Zurich, CERN, Pittsburgh U, IIT, QualComm and Bloomberg.

# Future Work

# Plans Next Year

The funding period is finished but we have plenty of interesting things to pursue in this area:

- Continue the open meetings policy
- Continue bug fixing and stabilizing Clang-Repl
- Merge Xeus-Cpp and Xeus-Clang-Repl
- Publish the results in the area of WebAssembly and on-line reoptimization
- Continue developing tutorials
- Reach out to other scientific domains to inform their communities for the new possibilities offered by our innovative software stack!

# A Note Of Gratitude

This multiyear, multi person effort would not have been possible without YOU!

The compiler-research team would like to express its deepest gratitude to the various people who contributed intellectual work in the area over the years!

# Conclusion

- C++ tools can bring us bare metal performance
- Existing tools can be reorganized and/or generalized with minimal efforts to enable new opportunities
- We should maintain them and grow them focusing on what they are good for
- Many community has multi-language expertise that can allow doing more science with the same budget

Thank you!