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.

Unlocking the Power of C++ as a Service: Uniting Python's Usability with C++'s Performance

Vassil Vassilev, compiler-research.org

12.10.2023

Motivation

Is there a way to combine the expressiveness of Python and the power of C++ without creating a new programming language?

Exploratory programming with C++

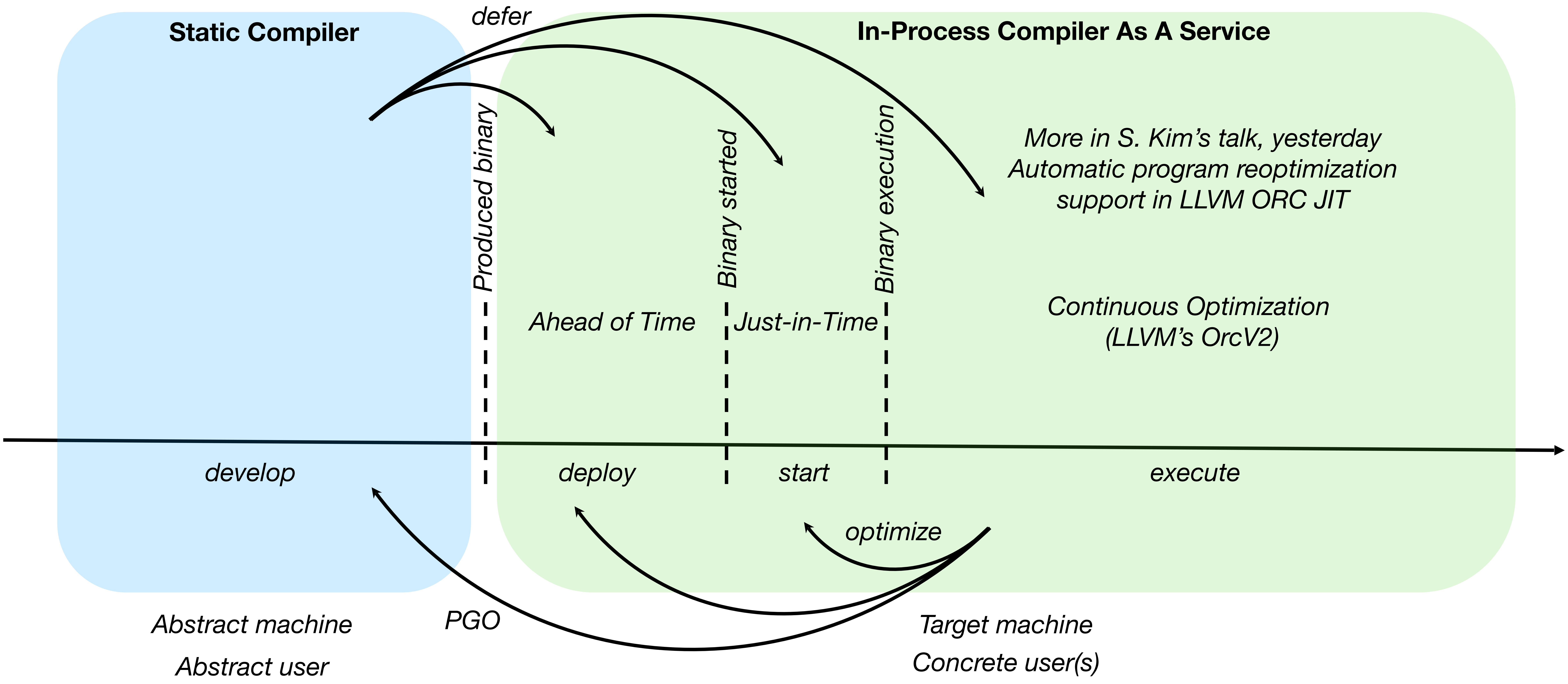
Interactive C++. Key Insights

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

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

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

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

> std::string v = "Hello World"
(std::string &) "Hello World"
```

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

Compiler (C++) As A Service



CaaS. Programming Model

```cpp
/// Call an interpreted function using its symbol address.
void callInterpretedFn(clang::Interpreter& interp) {
  // Declare a function to the interpreter. Make it extern "C"
  // to remove mangling from the game.
  clang::Value V;
  interp.ParseAndExecute("extern \"C\" int sq(int x) { return x * x; }"
                          "sq(12)", &V);
  printf("From JIT: square(12)=%d\n", V.getInt());
  auto SymAddr = ExitOnErr(Interp->getSymbolAddress("sq"));
  auto squarePtr = SymAddr.toPtr<int(*)(int)>();
  printf("From compiled code: sq(13)=%d\n", squarePtr(13));
}
```

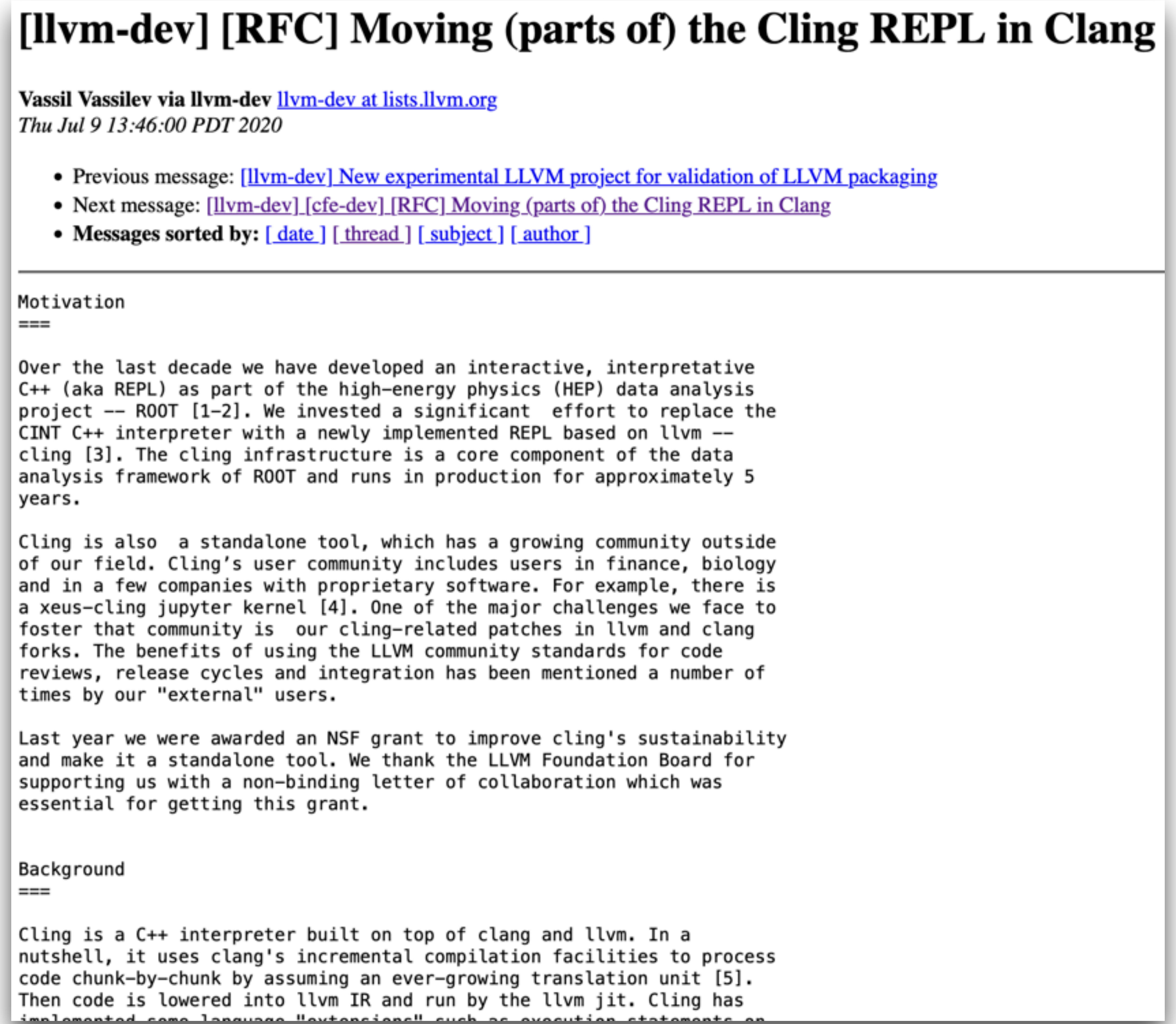
```cpp
// caas-demo.cpp
// g++ ... caas-demo.cpp; ./caas-demo
int main(int argc, const char* const* argv) {
  clang::IncrementalCompilerBuilder CB;
  CB.SetCompilerArgs({"-std=c++20"});
  auto I = Interpreter::create(std::move(CB.CreateCpp()));
  callInterpretedFn(I);
  return 0;
}
```

```
11:15:00-vvassilev~$ ./caas-demo
Hello Interpreter World!
From JIT: square(12)=144
From compiled code: square(13)=169
```

In-Tree Support for Incremental Compilation With Clang-Repl

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

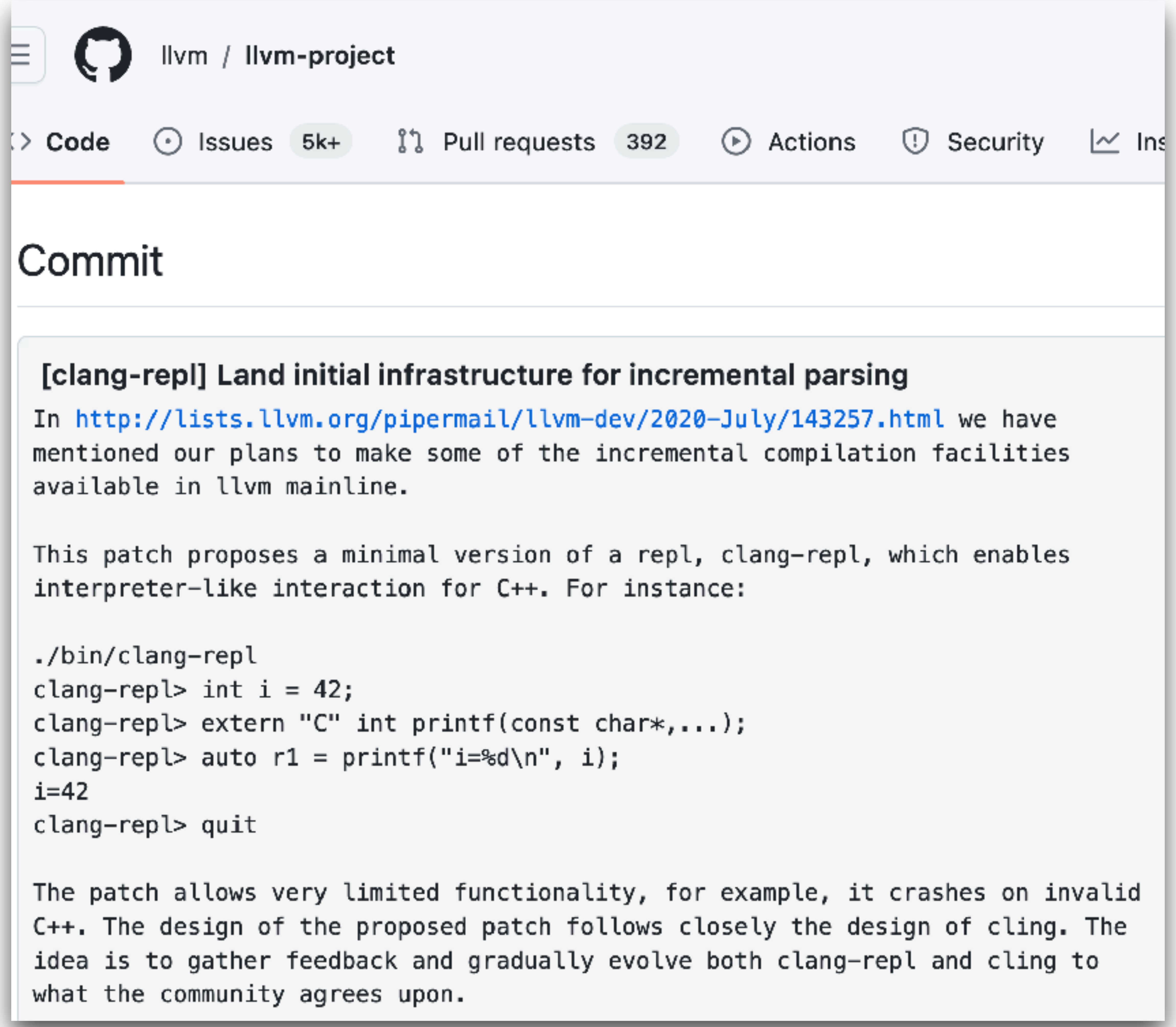
- Previous message: [llvm-dev] New experimental LLVM project for validation of LLVM packaging
- Next message: [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang
- **Messages sorted by:** [date] [thread] [subject] [author]

```
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). Most patches are released via LLVM17.

- 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 (1)

Y. Fu

Clang-Repl drove several new developments:

- Automatic completion at the prompt improving the overall user user experience (will be released in LLVM18). See Fred's student talk later today.
- Implement shared memory manager for JITLink enabling efficient out-of-process execution to improve system stability (LLVM15)
- Program reoptimization. See Sunho's talk from yesterday.

A. Ghosh

Developments Related to Clang-Repl (2)

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.

- Develop Windows Support (COFF in LLVM16)
- Develop ARM64 Unix Support (Aarch64 in LLVM16)
- Develop ARM32 Unix Support based on our ARM64 infrastructure — external contribution
- Develop RISCV JIT Support (LLVM16)
- Develop PowerPC Support (ppc64 in LLVM18) — contributed by IBM

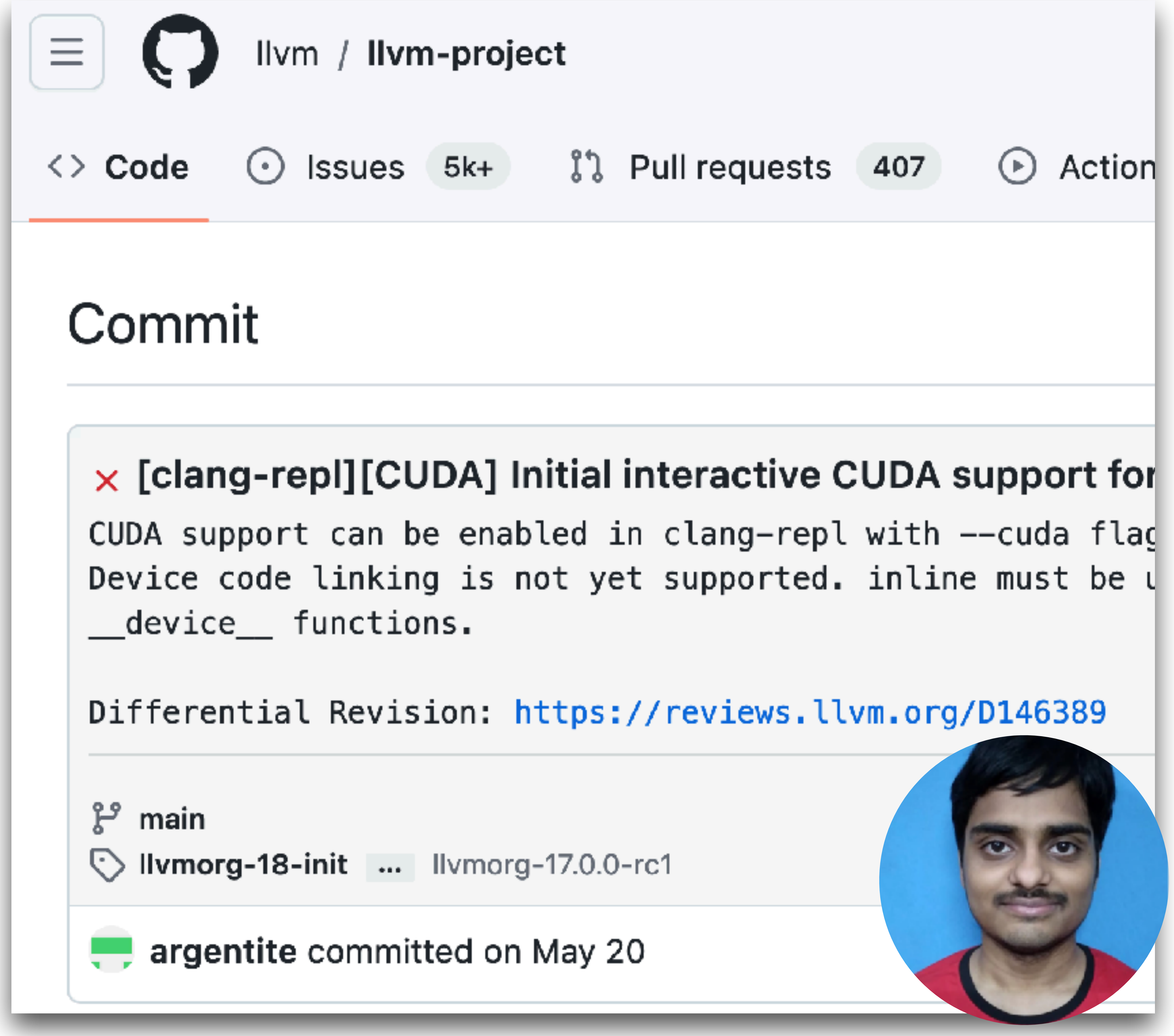
S. Kim

J. Hahnfeld

Interactive CUDA Support

Implemented a novel approach in interpreting CUDA codes where the PTX is passed through the virtual file system (LLVM17)

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



A. Ghosh

Clang-Repl in Data Science With Xeus

A. Ghosh

A. Penev

S. Corlay

I. Ifrim

Xeus is a protocol that enables executors to connect to the Jupyter infrastructure.

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

Automatic Language Interoperability

Interoperable, Interactive C++ 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 With Python

W. Lavrijsen

CPPYY is A CPython/PyPy Extension using their C API. It:

- offers automatic, on-demand mapping of Python to C++ concepts
- is an often neglected engineering marvel in language interoperability
- relies on on-demand reflection provided dynamically

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

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

While parsing we can associate each construct with a C++ entity. The approach does not require the project maintainer to bother providing static bindings

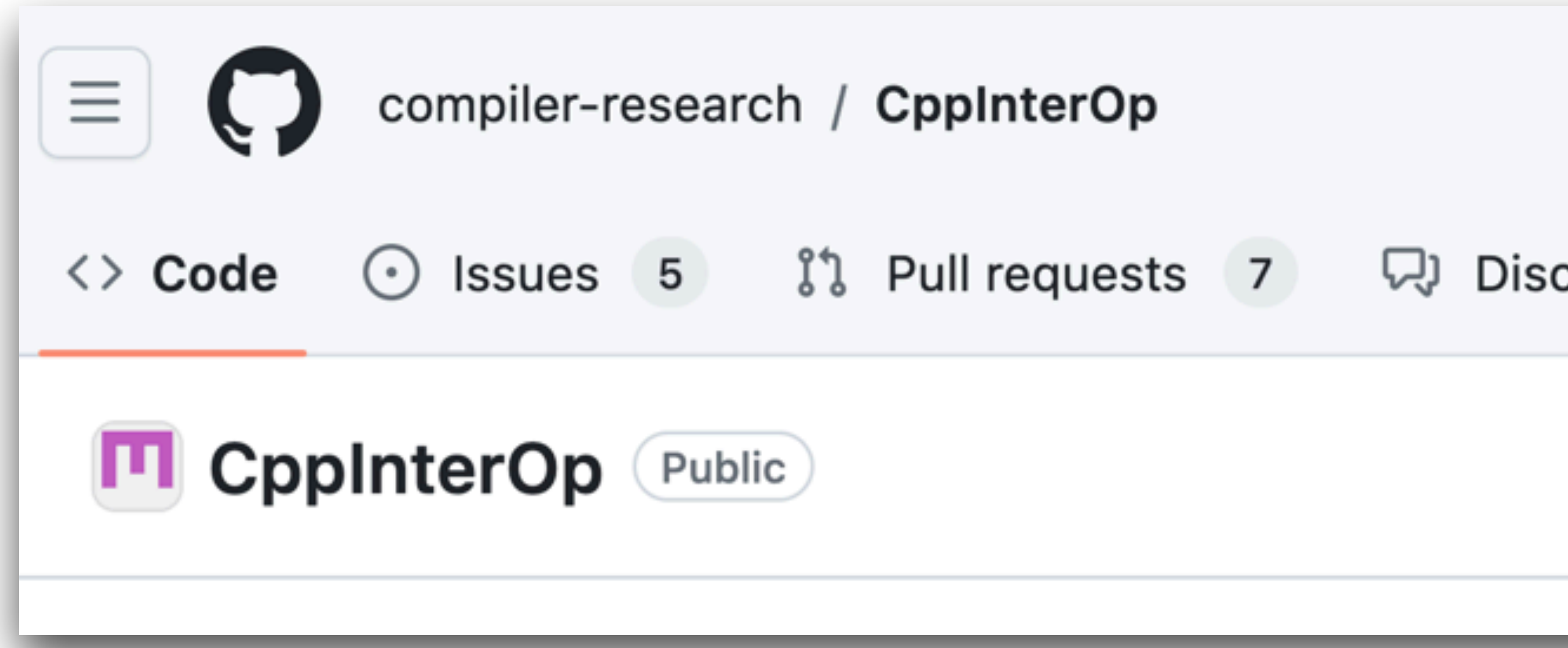
Moving CPPYY to LLVM Orbit With CppInterOp

Provides interoperability primitives for C++ to enable crosstalk in automatic way with Python but also for D, Julia, etc.

B. Kundu

The library allows replacing the cppyy backend with a specialized and more robust InterOp, moving it closer to LLVM orbit to allow:

- easier adoption of newer LLVM versions (CUDA, C++ standards)
- easer implementation of new features
- better release cycle, wider adoption



Tutorials & Community Outreach

https://github.com/compiler-research/pldi-tutorials-2023

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

Live Demo

Demo: 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);
```

Demo: OpenMP Hello World

M. Vassilev

- Run OpenMP codes in Jupyter

A. Penev

Demo: Image Processing. Mixing Python/C++/CUDA

Use Pillow and NumPy interactively with C++ and CUDA execution

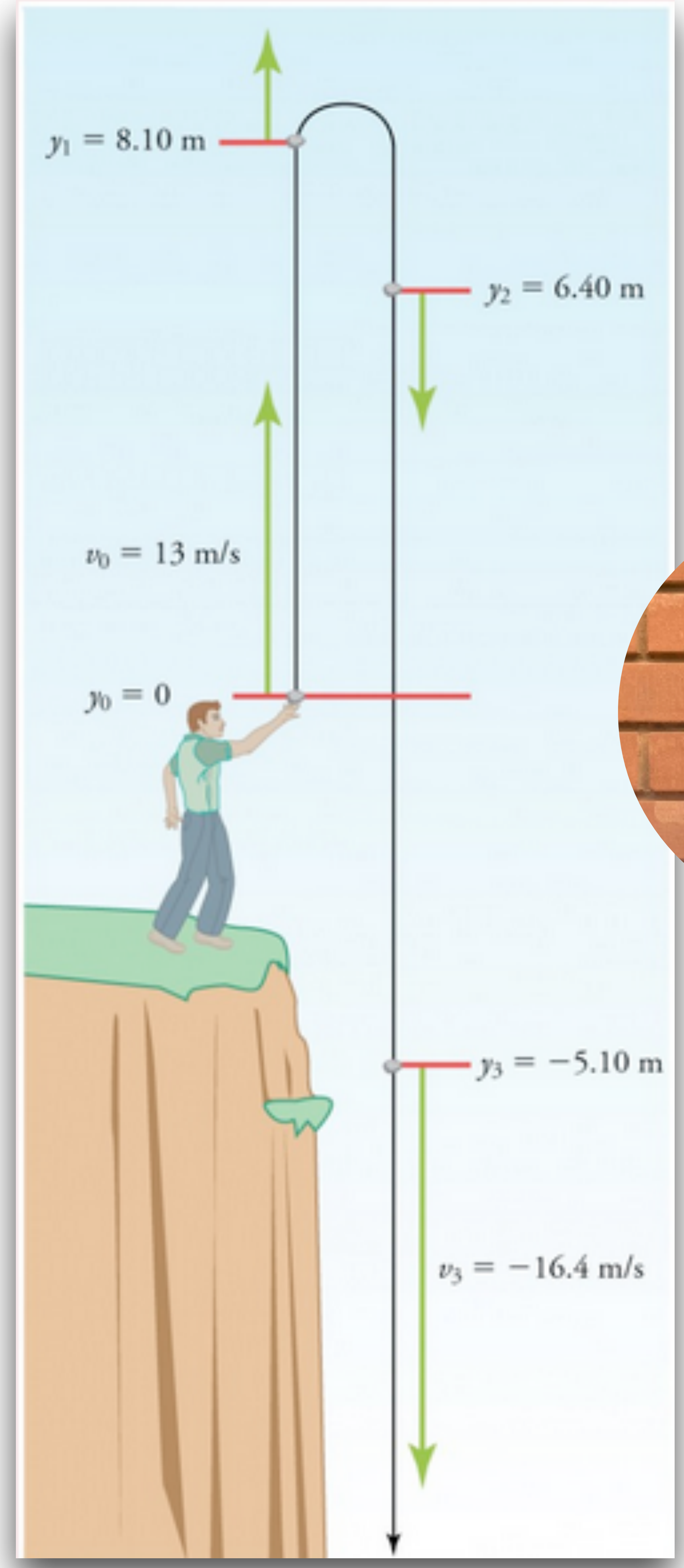
- CUDA: Apply underexposure to pixels based on a threshold value
- C++: Data conversion
- Python: Plotting

A. Jomy

Demo: Kalman Filter. Mixing Python/C++/CUDA

Use PyYaml and Matplotlib interactively with C++ and CUDA execution

- CUDA: Compute fast matrix and vector operations
- C++: Set of efficient CUDA function abstractions
- Python: Data processing and plotting



A. Jomy

Demo: JupyterLite

- Demonstrate Clang-Repl in browser

A. Ghosh

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 institutions such as UCSD, ETH Zurich, CERN, Pittsburgh U, IIT, QualComm and Bloomberg.

Future Work

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
- 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!

Thank you!