The CaaS Project. Progress & Plans Q3, Q4

Vassil Vassilev

Project Goals

- Support for incremental compilation (clang::libInterpreter, Clang-Repl)
- Language interoperability layer (cppyy, libInterOp)
- Heterogeneous hardware support (offload execution, clad demonstrator)
- Use case development & community outreach (tutorial development, demonstrators)

Project Goals

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

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

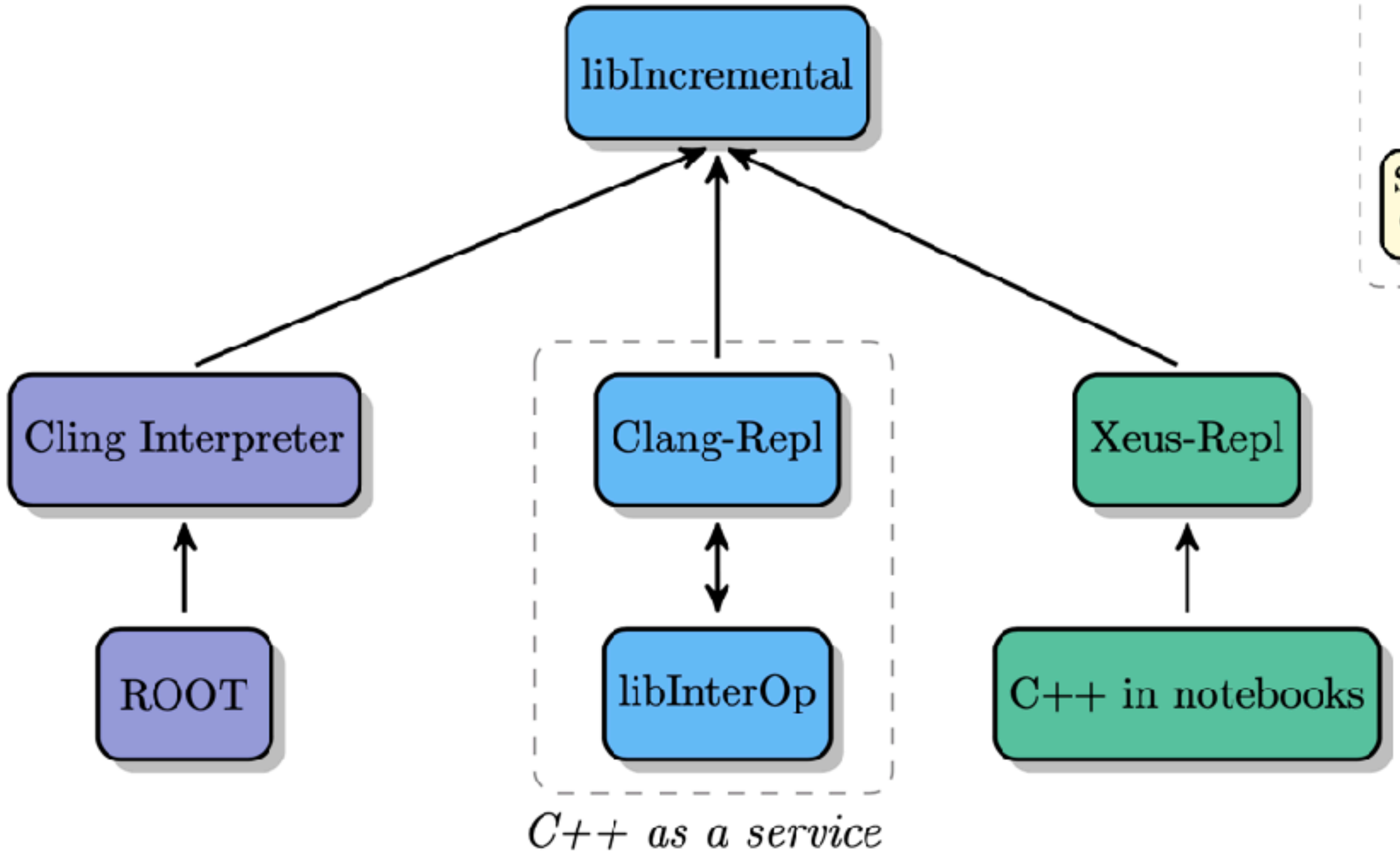
```
In [3]: print(python_vec[0].val)

        1
```

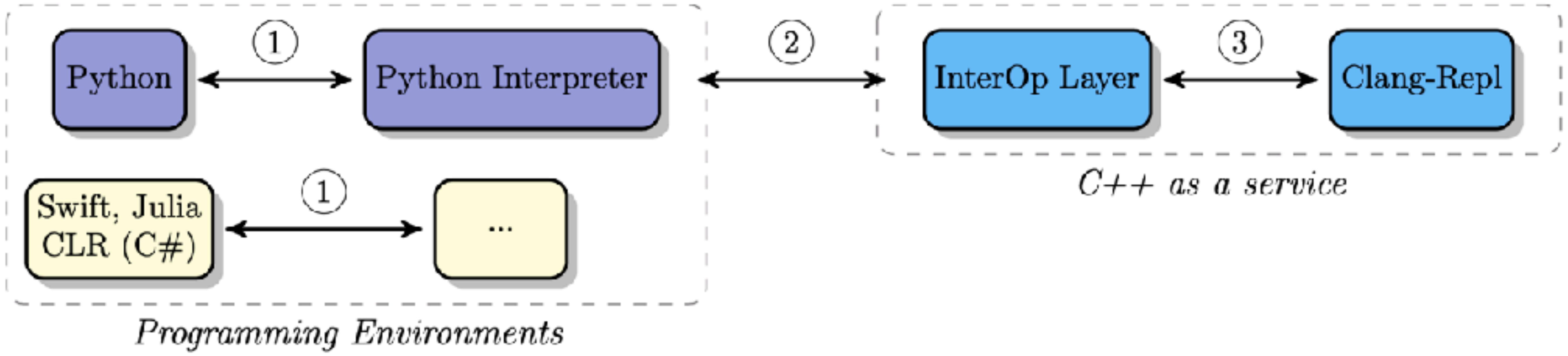
```
In [4]: class Derived(S)
            def __init__(self):
                self.val = 0
        res = Derived()
```

```
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);
```

Enable bi-directional language communication capable of controlling accelerator hardware

Project Goals





Reroute the cling-based ecosystem more to llvm upstream

Q3 Progress

1. [Q1/VV] Upgrade to LLVM 13 — **95**% complete

2. [Q1/VV] Update Cling to use more of LLVM13 — 60% complete (depends on 1.)

3. [Q1/DL] Construct simple patches to upstream dashboard to track — 100% complete

4. [Q1-Q4] Upstream Cling-specific patches — **23/87** complete

5. [Q1-Q4/DL] Keep track of Cling SLoC — Q3
 41 files changed, 847 insertions(+), 1242 deletions(-)

6. [Q2/II] Connect Clang-Repl to the Python Interpreter —100% complete, needs to land in llvm

7. [Q2/PA] Differentiate CUDA kernels — complete for forward mode

8. [Q2/VV] Implement in clang an extension to allow statements on the global scope — D127284

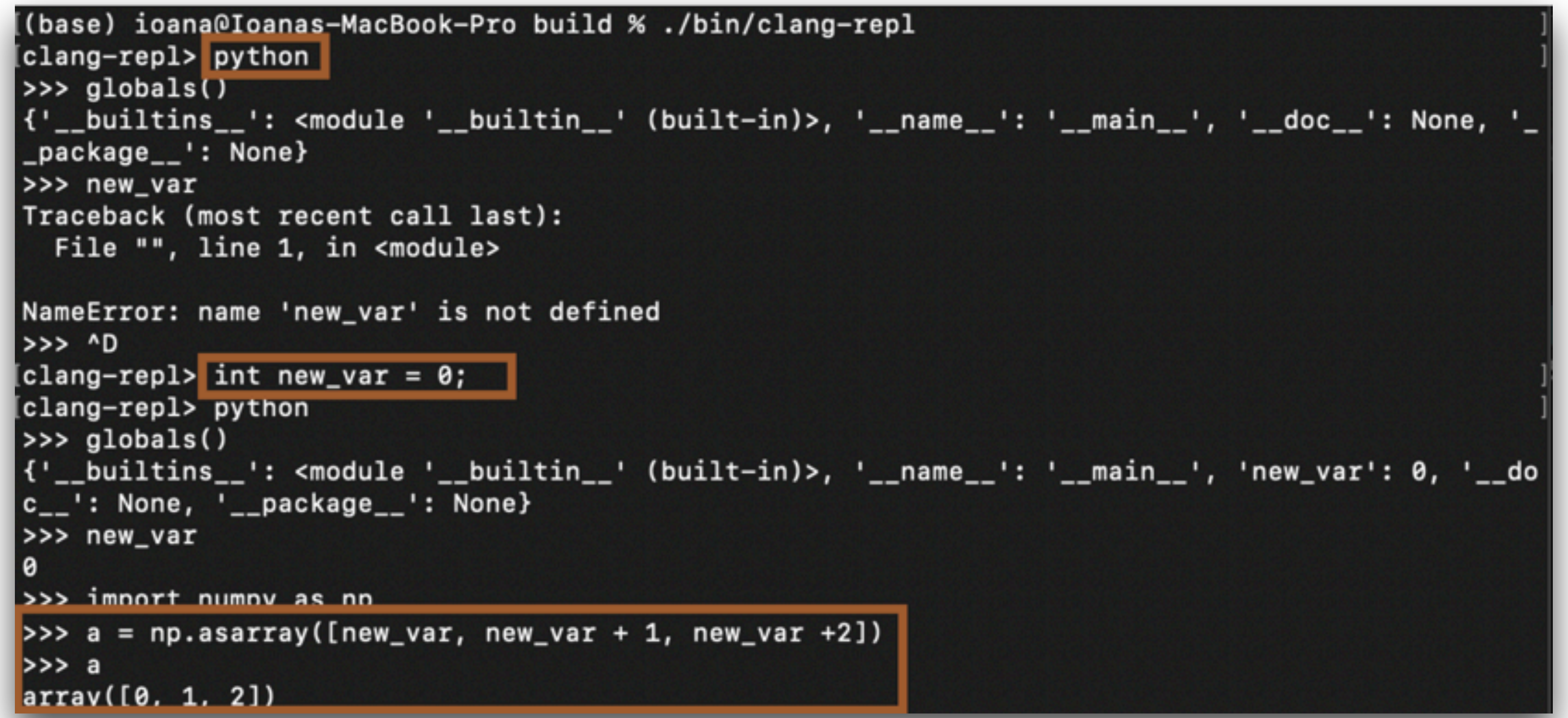
9. [Q2/PC] Advance error recovery and code unloading — D126682

10. [Q4/II/VV] Connect to xeus-cling (scope out missing functionality for xeus-repl) — working Jupyter Xeus-ClangRepl kernel

11. [Q3/II/VV] Develop demonstrators (eg the one we used for the cssi proposal) — simple example based on builtin types.

Q3 Progress

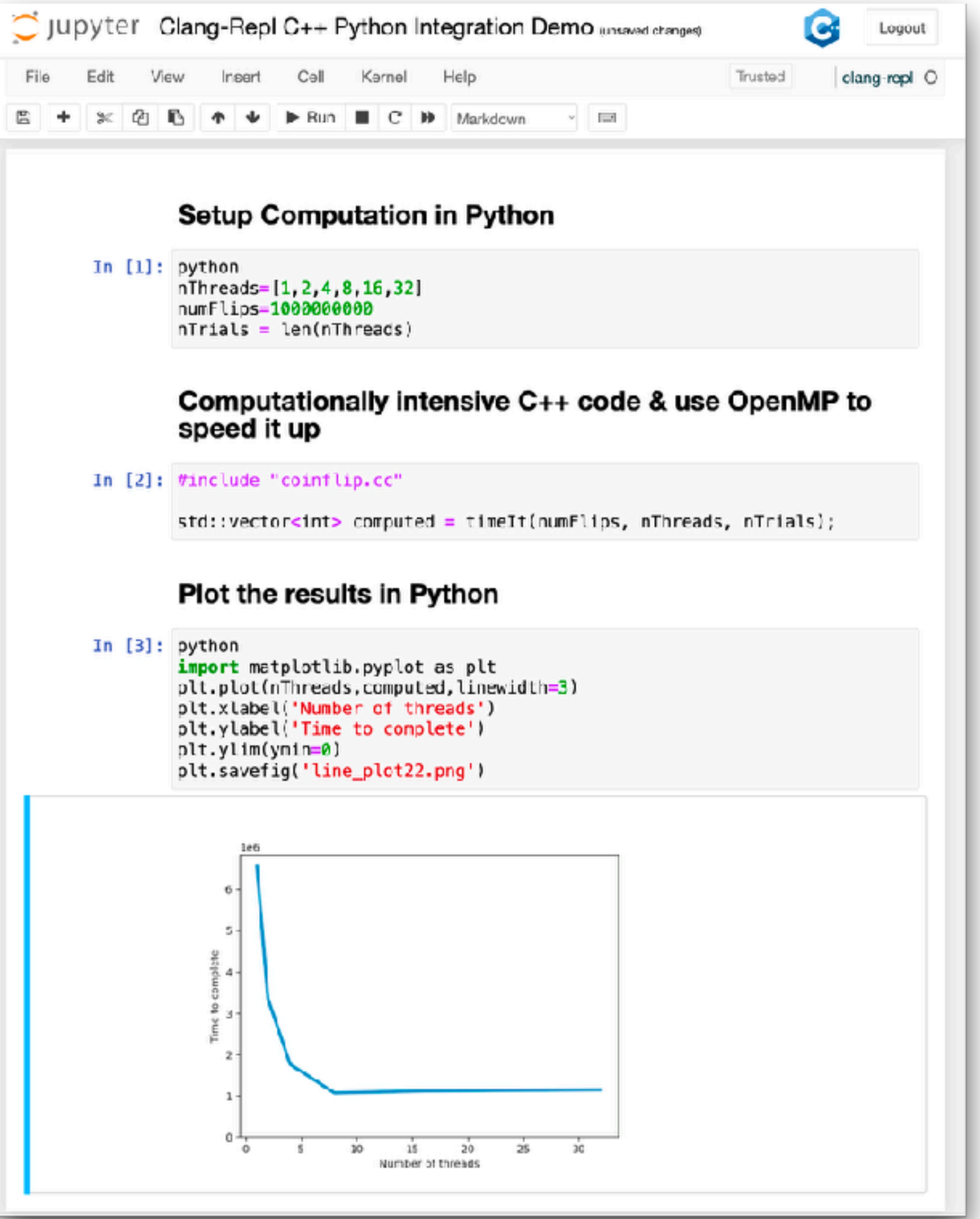
12. [Q4/II/VV] Basic Connect to xeus-cling (scope out missing functionality for xeus-repl) — working Jupyter Xeus-ClangRepl kernel



Q3 Progress

14. [Q3/II/VV] Develop demonstrators (eg the one we used for the cssi proposal) — simple example based on builtin types.

15. [VV] Added OpenMP support



Carry-over for Q4

1. Rebase cppyy to use cling-only interfaces (making cppyy ROOT-independent) — Q1/BK → Q4
   The task is about transforming the various ROOT Meta layer calls to their underlying clang/cling analogs

2. Define a set of new classes which handle what's needed (eg TClingCallFunc, etc) — Q1/BK → VV/Q4
   The task is about extracting the common cases where we need a lot of boilerplate code and provide abstractions for it. For example, the mechanism to call functions in a uniform way (currently done with TClingCallFunc) needs to modernized into its own ROOT-independent entity in libInterOp

3. Connect libInterOp with clang-repl — Q2/BK → Q4
   The python interpreter provides C API which allows to expose itself and switch to writing python code on the prompt. In ROOT this happens via TPython::Prompt and we want the modern version of this for clang-repl.

4. Improve test cases and demonstrators — Q2/II → Q4
   The task is about updating the existing demonstrators and developing new ones given the advances in Clad.

Carry-over for Q4

1. Add extensible value printing facility — Q2/VV → Q4
   The task is to improve and generalize the implementation of the PTX support in cling and demonstrate it in clang-repl.

2. Rebase cppyy to use clang-repl/libInterpreter interfaces — Q2/BK → Q4

3. Develop demonstrators (eg the one from the Jupyter mockup) — Q2/BK → N/A

4. Design and Develop a CUDA engine working along with C++ mode — Q2/II → N/A
   The task is to improve and generalize the implementation of the PTX support in cling and demonstrate it in clang-repl.

5. Design and implement a backend capable of offloading computations to a GPGPU. Assess technical performance of gradient produced by Clad on GPGPU — Q2/II,VV → N/A

6. Support Tensors and showcase differentiation of Eigen entities — Q1/PA → N/A

Carry-over for Q4

7. Add more clad benchmarks — Q2/DL → Q4
8. Add extensible value printing facility — Q2/VV → Q4
9. Write a paper on incremental C++ — Q2/VV → Q4
10. Write a paper on AD for the aggregate types — Q2/PA → N/A
11. Write an Error Estimation paper — Q2/GS → Q4

Plans for Q4

1. Upstream the type sugaring patch — GSoC Matheus
   The task includes re-engineering the solution we have in ROOT and making it acceptable for Clang.

2. Upstream the lazy template specializations patch —> N/A
   The task includes re-engineering the solution we have in ROOT and making it acceptable for Clang.

3. Develop documentation, examples and tutorials (in llvm documentation as well) — Sara and Rohit
   The task writing technical documentation and blog posts about the developed technologies.

4. Initiate tutorial development within the Clang-Repl community and integrate Clang-Repl into Xeus. Blog post on working notebook demonstrating tutorial — Sara and Rohit?
   The task writing technical documentation and blog posts about the developed technologies.

Plans for Q4

5. Implement an API to offload computations on GPGPUs in CaaS allowing to mix C/C++/CUDA and demonstrate Clad gradient in CUDA — ?

6. Optimize ROOT use of modules for large codebases (eg, CMSSW) — GSoC Jun
   One source of performance loss is the need for symbol lookups across the very large set of CMSSW modules. ROOT needs to be improved to optimize this lookup so that it does not pull all modules defining namespace `edm` on `edm::X` lookups. The task includes implementing a global module index extension which keeps information if an identifier name was a namespace and then integrating it in CMSSW builds.

7. Develop and document interoperability demonstrators based on MolSSI software packages — [II]

8. Write a paper on C++ Compiler As A Service. Dynamic Language Interop with C++ → [BK/VV]

9. Implement the LLVM extension of binding C++ memory management model more accurately and implement prototype using cppyy based on LLVM IR and the type resugaring

GSoC 2022

Contributors

Surya Somayyajula

IRIS-HEP Fellow, University of Wisconsin-Madison, USA

Improve Cling's packaging system: Cling Packaging Tool
(May 2022-Sep 2022)

Manish Kausik H

GSoC22, Computer Science and Engineering(Dual Degree), Indian Institute of Technology Bhubaneswar

Add Initial Integration of Clad with Enzyme
(May 2022-Sep 2022)

Matheus Izvekov

GSoC22, Computer Science

Preserve type sugar for member access on template specializations
(May 2022-~~Sep~~Nov 2022)

People

Sunho Kim

GSoC22, De Anza College, Cupertino, USA

Write JITLink support for a new format/architecture (ELF/AARCH64) (May 2022-Sep 2022)

Jun Zhang

GSoC22, Anhui Normal University, WuHu, China

Optimize ROOT use of modules for large codebases (May 2022-Sep 2022)

Anubhab Ghosh

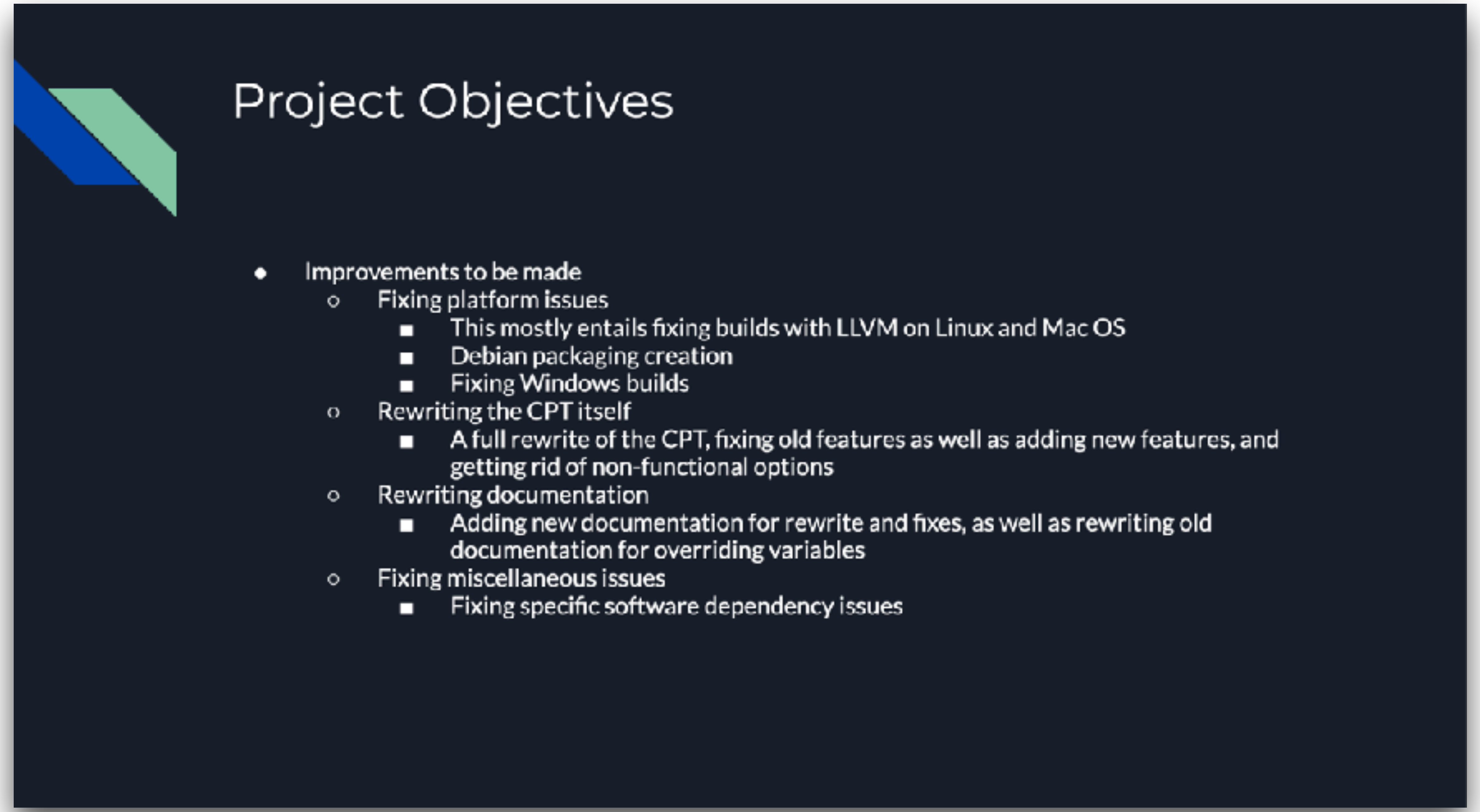
GSoC22, Indian Institute of Information Technology, Kalyani, India

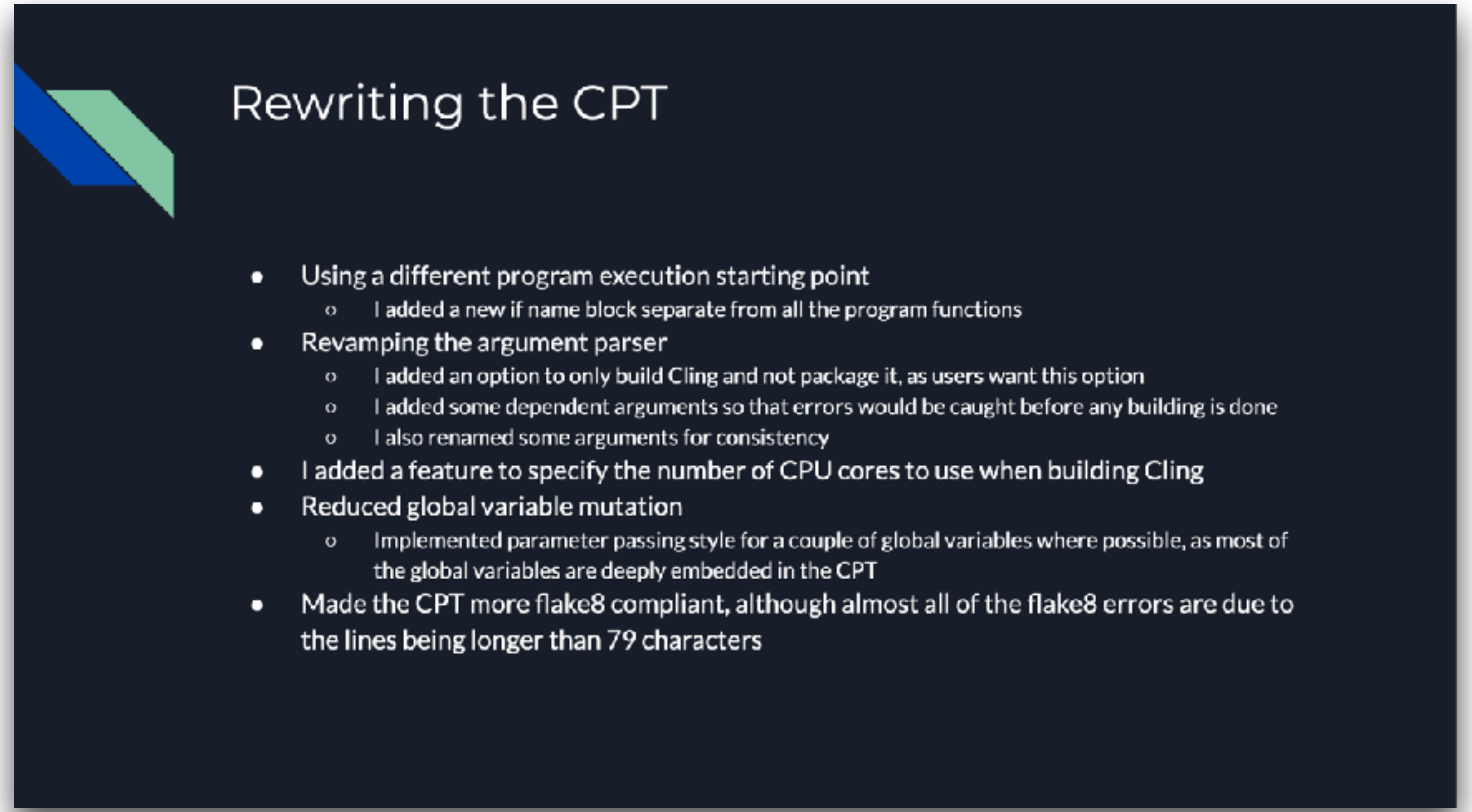
Shared Memory Based JITLink Memory Manager (May 2022-Sep 2022)

Surya Somayyajula

IRIS-HEP Fellow, University of Wisconsin-Madison.

Improve Cling's packaging system: Cling Packaging Tool (May 2022-Sep 2022). Slides: here.





Manish Kausik H

GSoC22, Indian Institute of Technology Bhubaneswar

Add Initial Integration of Clad with Enzyme
(May 2022-Sep 2022). Slides: here and here. Final report. Blog Post.

Implementation Ideas

2. Reverse Mode Differentiation Code Generation

- **DiffCollector::VisitCallExpr** must set a variable in the DiffRequest Object, that states whether the user wants to use enzyme or not.
- **ReverseModeVisitior::Derive** must create a new branch for Enzyme DiffRequests, with a constant template code
- Must link the Code generated by **ReverseModeVisitor::Derive** with the **CladFunction** class (Need to explore this)
- How can **DiffCollector::VisitCallExpr** recognise the request for use of enzyme based on a template parameter? (Need to explore this)

Integrating Enzyme Reverse Mode with Clad

1. Identifying a request for using Enzyme with Clad (PR #460)
2. Integrating Enzyme as a static library in Clad (PR #466)
3. Generating code for Enzyme Reverse mode with clad (PR #486)
4. Verifying Enzyme generated derivatives with clad(PR #488)

```
clad::gradient(f) //Normal Calling convention
clad::gradient<clad::opts::use_enzyme>(f) //Calling Convention for using Enzyme within Clad
```

Matheus Izvekov

Preserve type sugar for member access on template specializations (May 2022-~~Sep~~Nov 2022). Slides: here and here.

GSoC22

In simplest terms, with an example, we want this to work:

```
template<class T> struct foo { using type = T; };

struct Baz {};

using Bar [[gnu::aligned(64)]] = Baz;

using type = typename foo<Bar>::type;

// Clang as it stands will fail below assert
// as the foo template will only be instantiated
// with the structural part of the argument,
// which the Bar alias is not.
// So it only sees Baz, the aligned attribute is never seen.
static_assert(alignof(type) == 64);
```

Accomplishments

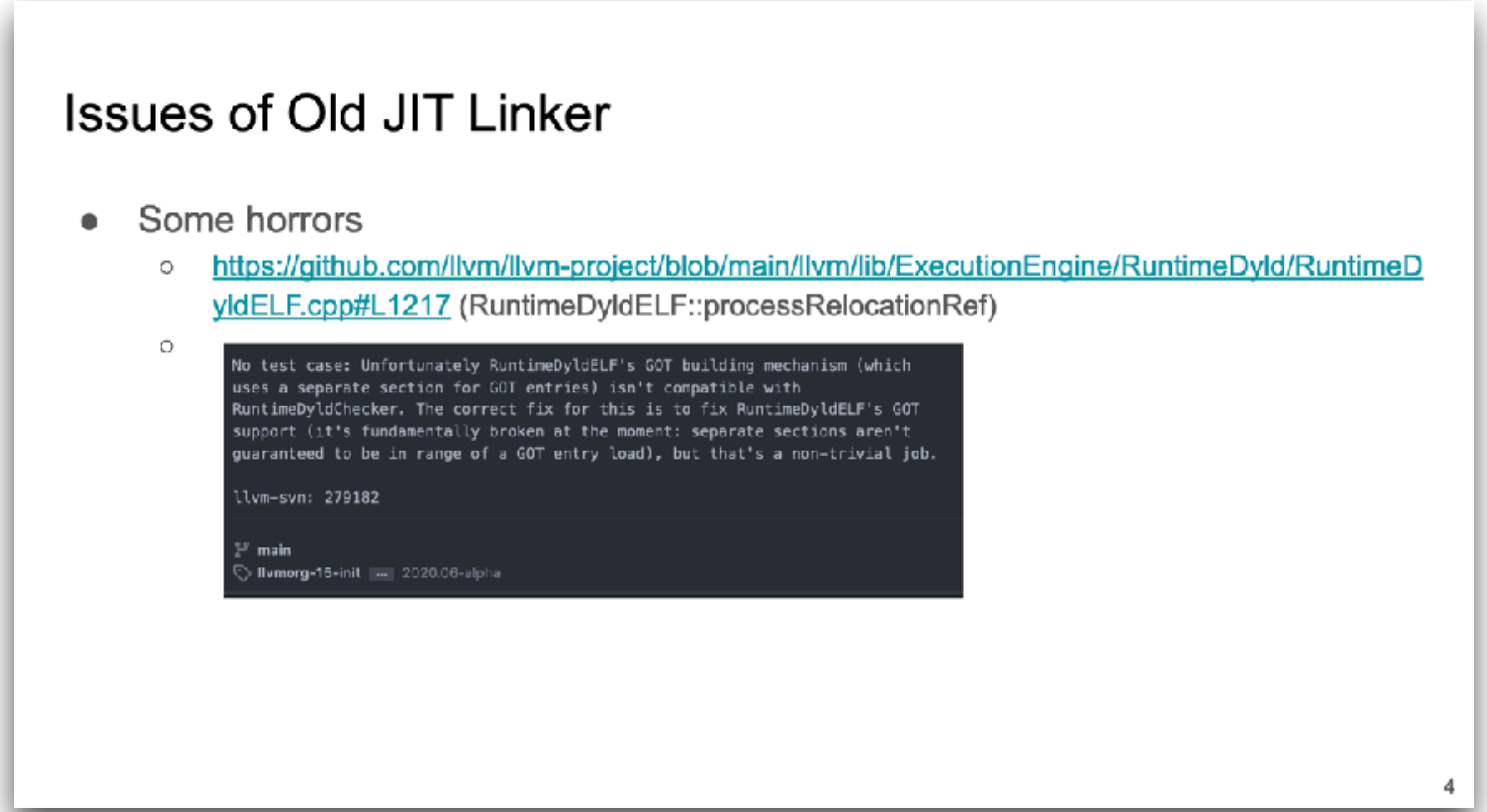
We submitted the RFC at https://discourse.llvm.org/t/rfc-improving diagnostics-with-template-specialization-resugaring/64294.

- We have positive feedback, people want to see this implemented
- We got one extra volunteer for reviewing.
- We got feedback that this work might influence debug info.
- We linked to the WIP patch in phabricator.
  However, the patch is too big and we must work on splitting it up

Sunho Kim

GSoC22, De Anza College, Cupertino, USA

Write JITLink support for a new format / architecture (ELF / AARCH64).
Slides: here and here. Final report.

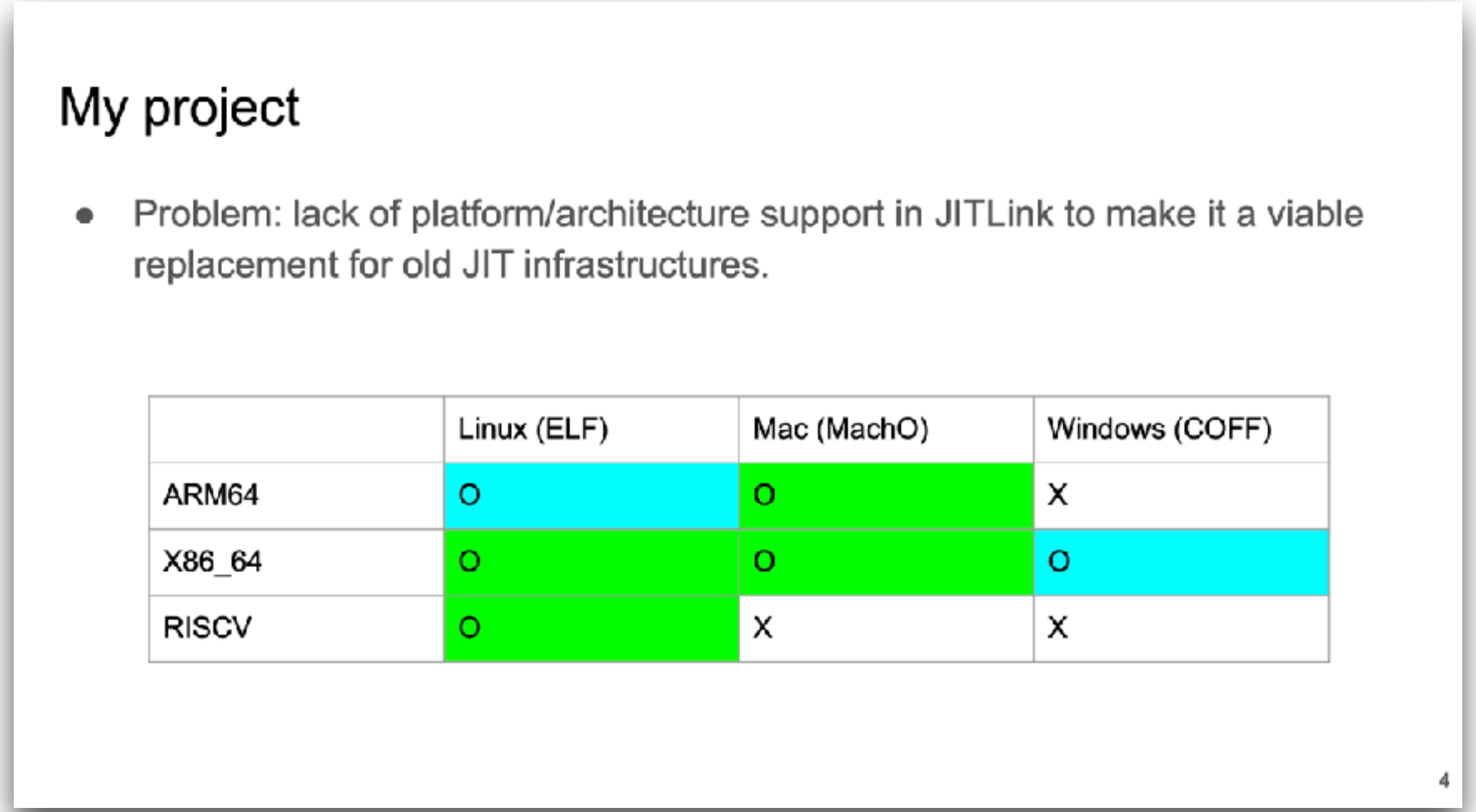


Issues of Old JIT Linker

- Some horrors
  - https://github.com/llvm/llvm-project/blob/main/llvm/lib/ExecutionEngine/RuntimeDyld/RuntimeDyldELF.cpp#L1217 (RuntimeDyldELF::processRelocationRef)
  -

```
No test case: Unfortunately RuntimeDyldELF's GOT building mechanism (which
uses a separate section for GOT entries) isn't compatible with
RuntimeDyldChecker. The correct fix for this is to fix RuntimeDyldELF's GOT
support (it's fundamentally broken at the moment: separate sections aren't
guaranteed to be in range of a GOT entry load), but that's a non-trivial job.

llvm-svn: 279182

main
llvmorg-15-init ... 2020.06-alpha
```

4



My project

- Problem: lack of platform/architecture support in JITLink to make it a viable replacement for old JIT infrastructures.

	Linux (ELF)	Mac (MachO)	Windows (COFF)
ARM64	O	O	X
X86_64	O	O	O
RISCV	O	X	X

4

Anubhab Ghosh

GSoC22, Indian Institute of Information Technology, Kalyani,

Shared Memory Based JITLink Memory Manager.
Slides: here. Final report.

The plan

- A MemoryMapper interface with implementations based on
  - Shared memory
    - When both executor and controller process share same physical memory
  - Regular memory allocation APIs
    - When the resultant code is executed in the same process
    - Useful for unit tests
  - EPC
    - Required when the executor and controller process run with different physical memory
    - Resultant code is transferred to the executor process over the EPC channel
- A JITLinkMemoryManager implementation that can use any MemoryMapper
  - It will allocate large chunks of memory using MemoryMapper and divide into smaller chunks
  - Better support for small code model by keeping everything close in memory

Design and Implementation

- `orc::MemoryMapper` interface: This is an interface to perform memory allocation, deallocation, setting memory protections etc. that handles most platform-specific operations. This abstraction allows us to decouple the transport for generated code from heap management making it simple for clients to use different transport mechanisms. (D127491)
  - `orc::InProcessMemoryMapper` : This implementation is used when running code in the same process where the JIT is running and uses `sys::Memory` API. (D127491)
  - `orc::SharedMemoryMapper` : This implementation is used when transferring code to a different executor process and uses POSIX or Win32 shared memory APIs. (D128544)
- `orc::MapperJITLinkMemoryManager` : This class implements the `jitlink::JITLinkMemoryManager` interface and handles all allocations within a slab. (D130392)
- Memory coalescing to join two consecutive free ranges and reuse them. (D131831)
- `llvm-jitlink` tool integration:
  - `MapperJITLinkMemoryManager` with an `InProcessMemoryMapper` is used by default when executing the code in the same process as the JIT. (D132315)
  - `MapperJITLinkMemoryManager` with a `SharedMemoryMapper` can be optionally used when `--use-shared-memory` is passed. (D132369)