

Enabling Interactive C++ in Clang

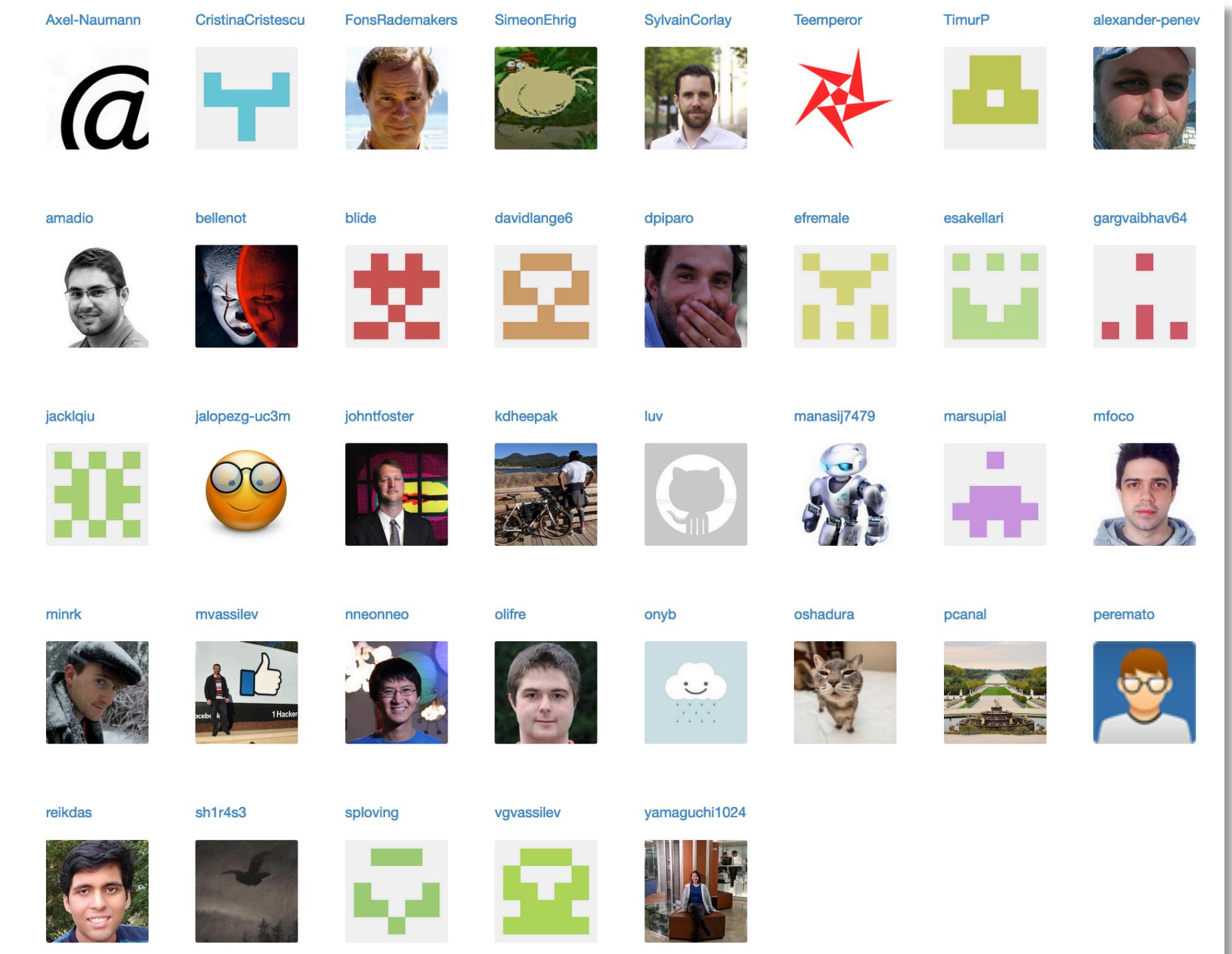
Vassil Vassilev, Princeton University
compiler-research.org

Outline

- Introduction
 - Key insights of Interactive C++
 - Interpreting C++. Tools and technology
- Applications of interactive C++
 - C++ in Jupyter notebooks; Interactive CUDA C++; Automatic language bindings; Eval-style programming
- Compiler As A Service
 - Crossing compile-time/runtime boundaries; Extensions; Automatic differentiation
- Evolving the technology towards Clang mainline via Clang-Repl
 - Showcase incremental compilation in Clang; Demonstrate template instantiation in C and Python
- Summary

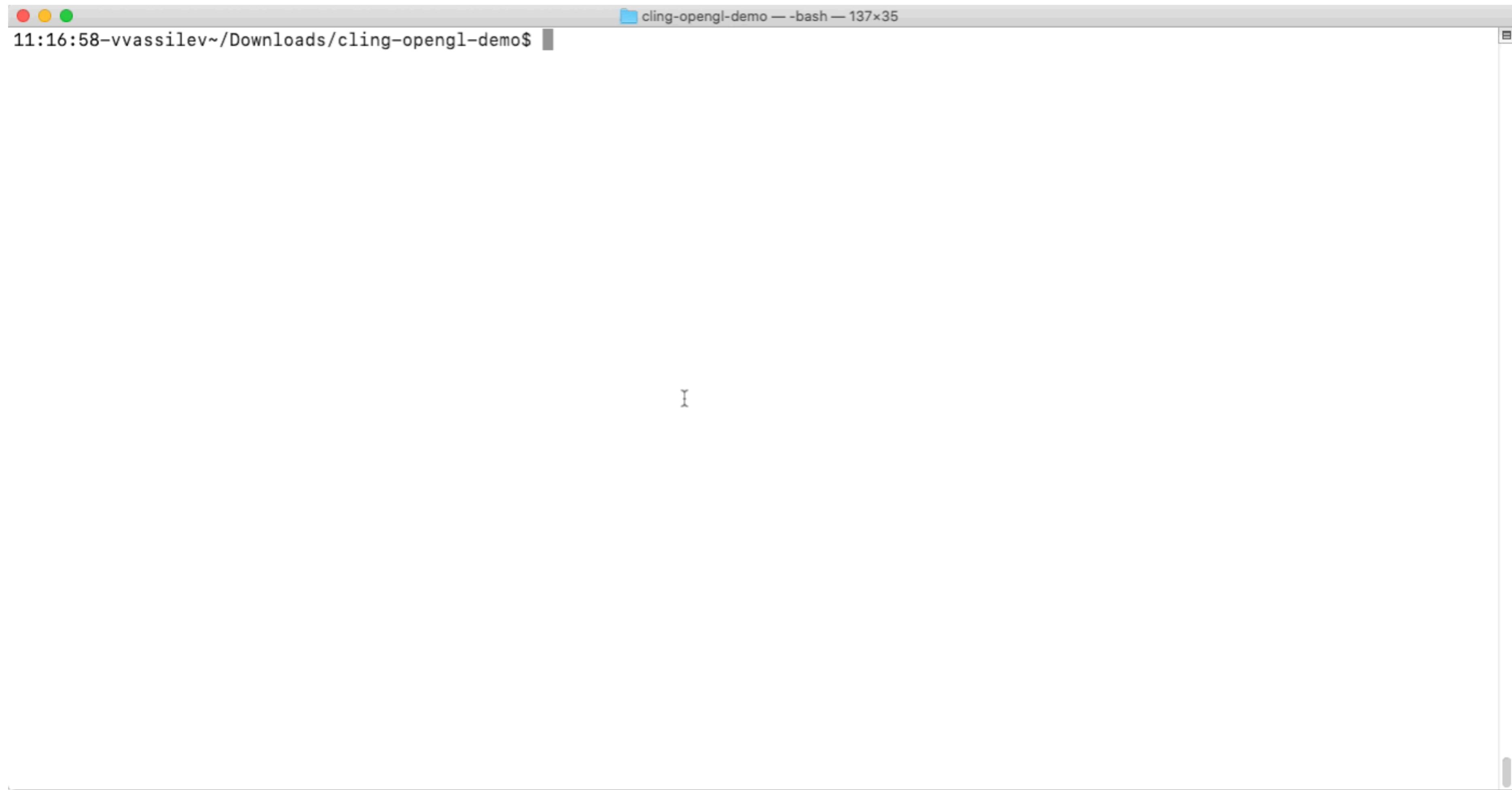
Acknowledgement & Disclaimer

- This talk includes technologies developed by various individuals and organizations in the area of interpretative C++ since 1998
- This talk is about work conducted by me but also the work of dozens colleagues and contributors from science and industry. In the slides I have tried to mention individuals and organizations where possible.
- Any characterizations, mischaracterizations, emphasis or errors are solely mine and do not necessarily represent the views of other individuals or organizations.



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.

Interactive C++



Video Credits: A. Penev

The invisible compile-run cycle aids interactive use and offers a different programming experience while enhancing productivity. It becomes trivial to orient a shape, choose size and color or compare to previous settings

Interactive C++. Key Insights

- Incremental Compilation

- Handling errors

- Syntactic
- Semantic

- Execution of statements

- Displaying execution results

- Entity redefinition

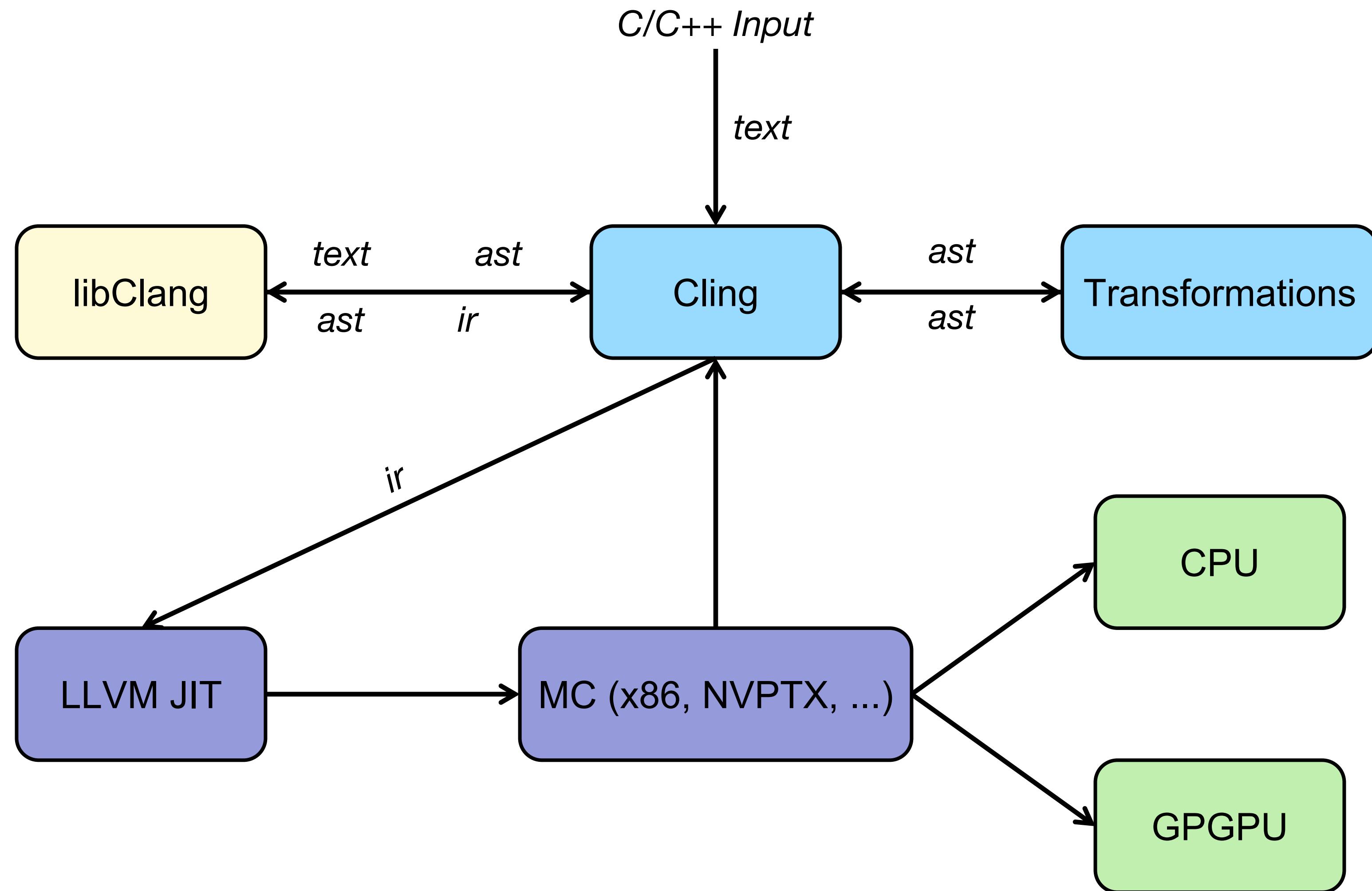
```
[cling] #include <vector>
[cling] std::vector<int> v = {1,2,3,4,5};
```

```
[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"
```

Interpreting C++. Cling



Applications of Interactive C++

Xeus-Cling. C++ in Notebooks

The screenshot displays three Xeus-Cling notebooks side-by-side:

- Left Notebook:** Shows code for handling image mime types and a visualization of a portrait of Marie Curie.
- Middle Notebook:** Shows symbolic computation with Symengine, including a cell output showing the expansion of a mathematical expression.
- Right Notebook:** Shows direct access to cppreference.com documentation for std::vector, including template code and notes about its usage.

Visualization of user-defined images

Direct access to documentation

Rich mime type rendering in Jupyter

Xeus-Cling. C++ in Notebooks

The screenshot shows a Jupyter notebook interface with the title "xwidgets.ipynb". The code cell contains the following C++ code:

```
#include "xwidgets/xslider.hpp"
xw::slider<double> slider;
slider.value() = 20;
// change some more properties
slider.max = 40;
slider.style().handle_color = "blue";
slider.orientation = "vertical";
slider.description = "A slider";
#include "xcpp/xdisplay.hpp"
using xcpp::display;
```

Xwidgets – User-defined controls

The screenshot shows a Jupyter notebook interface with the title "xleaflet_split_n.ipynb". The code cell contains the following C++ code:

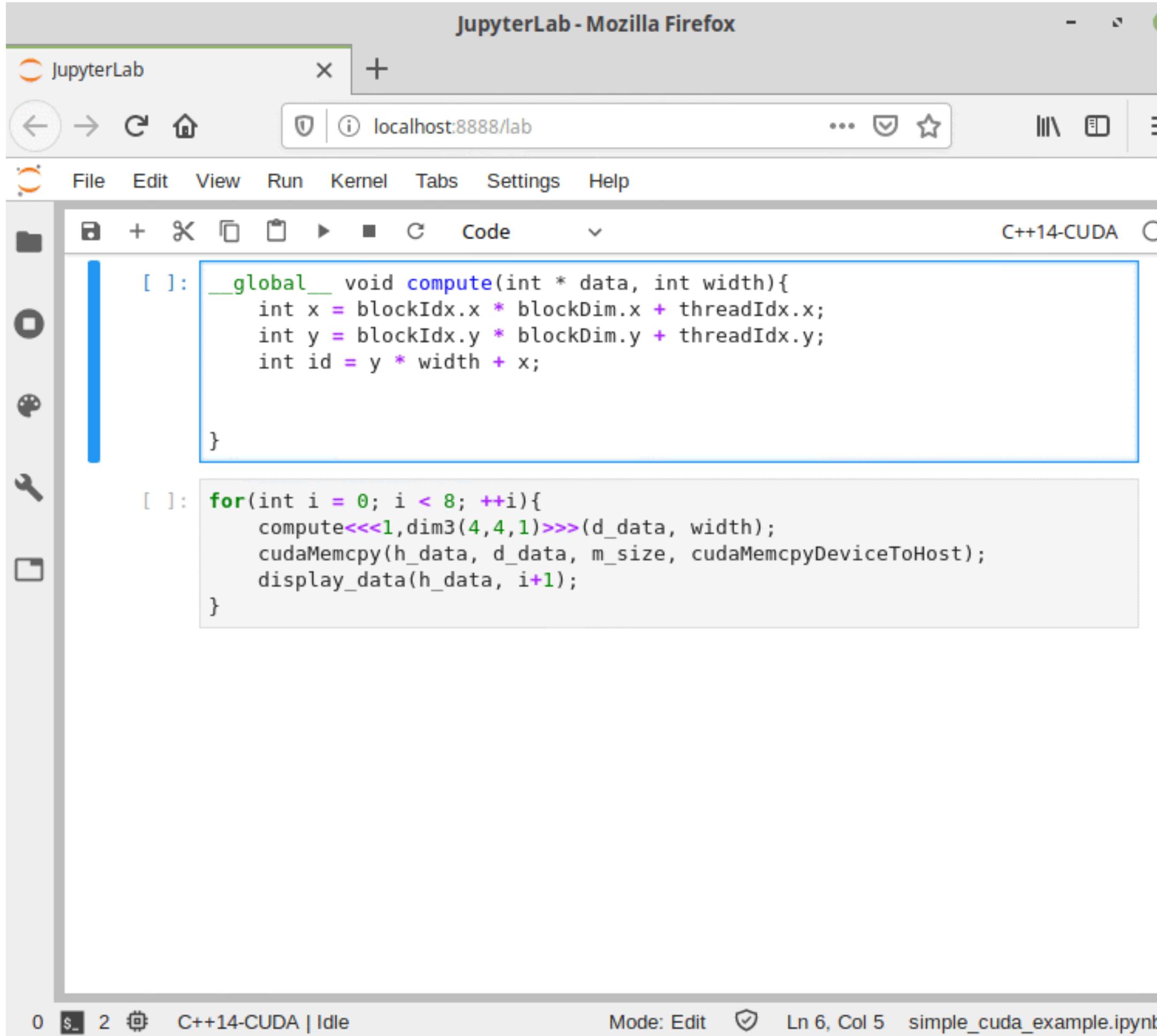
```
std::ifstream file("geo.json");
yajl::json geo_data;
```

The output cell displays an interactive map of Jacksonville, Florida, showing various geographical features and airports.

Xleaflet – Interactive Geo Information System

S. Corlay, Quantstack, [Deep dive into the Xeus-based Cling kernel for Jupyter](#), May 2021, compiler-research.org

Interactive CUDA C++



JupyterLab - Mozilla Firefox

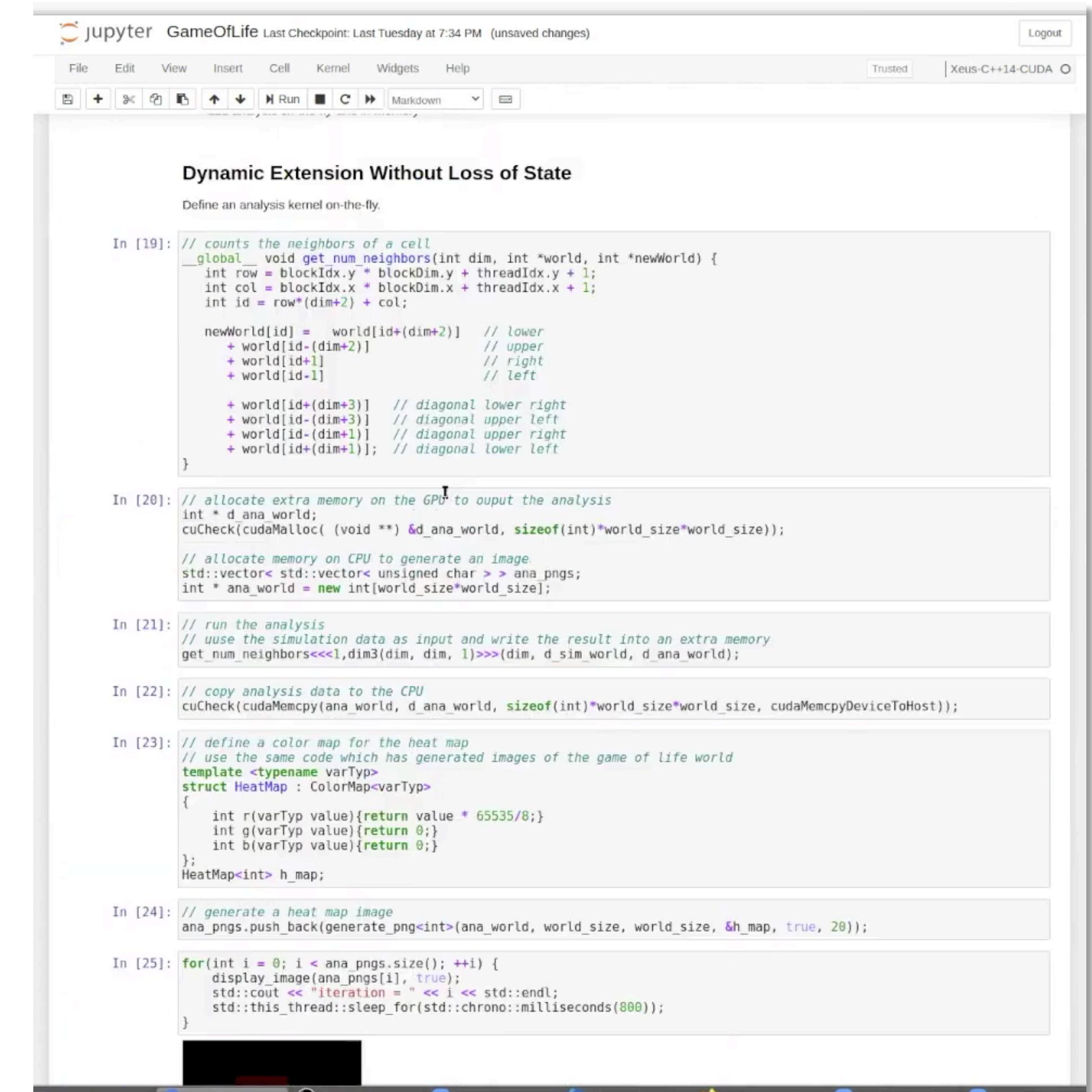
localhost:8888/lab

File Edit View Run Kernel Tabs Settings Help

C++14-CUDA

```
[ ]: __global__ void compute(int * data, int width){  
    int x = blockIdx.x * blockDim.x + threadIdx.x;  
    int y = blockIdx.y * blockDim.y + threadIdx.y;  
    int id = y * width + x;  
  
}  
  
[ ]: for(int i = 0; i < 8; ++i){  
    compute<<<1,dim3(4,4,1)>>>(d_data, width);  
    cudaMemcpy(h_data, d_data, m_size, cudaMemcpyDeviceToHost);  
    display_data(h_data, i+1);  
}
```

Mode: Edit Ln 6, Col 5 simple_cuda_example.ipynb



jupyter GameOfLife Last Checkpoint: Last Tuesday at 7:34 PM (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Logout Trusted Xeus-C++14-CUDA

Dynamic Extension Without Loss of State

Define an analysis kernel on-the-fly.

```
In [19]: // counts the neighbors of a cell  
__global__ void get_num_neighbors(int dim, int *world, int *newWorld) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y + 1;  
    int col = blockIdx.x * blockDim.x + threadIdx.x + 1;  
    int id = row*(dim+2) + col;  
  
    newWorld[id] = world[id+(dim+2)] // lower  
    + world[id-(dim+2)] // upper  
    + world[id+1] // right  
    + world[id-1] // left  
  
    + world[id+(dim+3)] // diagonal lower right  
    + world[id-(dim+3)] // diagonal upper left  
    + world[id-(dim+1)] // diagonal upper right  
    + world[id+(dim+1)]; // diagonal lower left  
}  
  
In [20]: // allocate extra memory on the GPU to output the analysis  
int * d_ana_world;  
cuCheck(cudaMalloc( (void **) &d_ana_world, sizeof(int)*world_size*world_size));  
  
// allocate memory on CPU to generate an image  
std::vector< std::vector< unsigned char > > ana_pngs;  
int * ana_world = new int[world_size*world_size];  
  
In [21]: // run the analysis  
// use the simulation data as input and write the result into an extra memory  
get_num_neighbors<<<1,dim3(dim, dim, 1)>>>(dim, d_sim_world, d_ana_world);  
  
In [22]: // copy analysis data to the CPU  
cuCheck(cudaMemcpy(ana_world, d_ana_world, sizeof(int)*world_size*world_size, cudaMemcpyDeviceToHost));  
  
In [23]: // define a color map for the heat map  
// use the same code which has generated images of the game of life world  
template <typename varTyp>  
struct HeatMap : ColorMap<varTyp>  
{  
    int r(varTyp value){return value * 65535/8;}  
    int g(varTyp value){return 0;}  
    int b(varTyp value){return 0;}  
};  
HeatMap<int> h_map;  
  
In [24]: // generate a heat map image  
ana_pngs.push_back(generate_png<int>(ana_world, world_size, world_size, &h_map, true, 20));  
  
In [25]: for(int i = 0; i < ana_pngs.size(); ++i){  
    display_image(ana_pngs[i], true);  
    std::cout << "iteration = " << i << std::endl;  
    std::this_thread::sleep_for(std::chrono::milliseconds(800));  
}
```

S. Ehrig, HZDR, [Cling's CUDA Backend: Interactive GPU development with CUDA C++](#), Mar 2021, compiler-research.org

Automatic Language Bindings

cppyy: Yet another Python – C++ binder?!

- Yes, but it has its niche: *bindings are runtime*
 - Python is all runtime, so runtime is more natural
 - C++-side runtime-ness is provided by Cling
- Very complete feature-set (not just “C with classes”)
- Good performance on CPython; great with PyPy*

pip: <https://pypi.org/project/cppyy/>
conda: <https://anaconda.org/conda-forge/cppyy>
git: <https://github.com/wlav/cppyy>
docs: <https://cppyy.readthedocs.io/en/latest/>

For HEP users: *cppyy in ROOT is an old fork. It won't run all the examples here, doesn't work with PyPy, and has worse performance.*

(* PyPy support lags CPython

- 2 -



[1]

sil-cling: Interface with cppyy

```
1 //cling.dpp
2
3 #include "capi.h" // cppyy's C header
4
5 // D code ↓
6 import std.string : fromStringz, toStringz;
7
8 string resolveName(string cppItemName)
9 {
10    import core.stdc.stdlib : free;
11    // Calling cppyy_resolve_name ↓
12    char* chars = cppyy_resolve_name(cppItemName.toStringz());
13    string result = chars.fromStringz.idup;
14    free(chars);
15    return result;
16 }
```

→ ~ dub run dpp -- cling.dpp --keep-d-files -c

[2]

How does it work? - Runtime

The screenshot shows a Julia REPL session. The user defines a C++ struct named 'nontrivial' with a destructor that prints "I got deleted". They then create an instance of this struct and assign it to a variable 'a'. Later, they set 'a' to nothing and perform a garbage collection. The output shows the destructor being called at each step, with annotations explaining the process:

- Nesting works also at global scope
- Last reference dropped here
- Julia GC deletes object => C++ destructor called

[3]

[1] W. Lavrijsen, LBL, [cppyy](#), Sep 2021, compiler-research.org

[2] A. Militaru, Symmetry Investments, [Calling C++ libraries from a D-written DSL: A cling/cppyy-based approach](#), Feb 2021, compiler-research.org

[3] K. Fischer, Julia Computing, [A brief history of Cxx.jl](#), Aug 2021, compiler-research.org

Eval-Style Programming

```
[cling]$ #include <cling/Interpreter/Value.h>
[cling]$ #include <cling/Interpreter/Interpreter.h>
[cling]$ int i = 1;
[cling]$ cling::Value v;
[cling]$ gCling->evaluate("++i", v);
[cling]$ i
(int) 2
[cling]$ v
(cling::Value &) boxes [(int) 2]
[cling]$ ++i
(int) 3
[cling]$ v
(cling::Value &) boxes [(int) 2]
```

Eval-style programming enables Cling to be embedded in frameworks.

Key Insights

- Cling is not just a Repl, it is an embeddable and extensible execution system for efficient incremental execution of C++
- Cling is used in several high-performance systems to provide reflection and introspection information
- Cling can produce efficient code for performance-critical tasks where hot-spot regions can be annotated with specific optimization levels
- Cling allows us to decide how much we want to compile statically and how much to defer for the target platform

Compiler As A Service

Compiler As A Service (CaaS)

Cling can be used on-demand, as a service, to compile, modify, describe or extend C++.

CaaS. Crossing Boundaries

```
// 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, LLVMMDIR);  
  
    callInterpretedFn(interp);  
    return 0;  
}
```

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

CaaS. Extensions

```
int main(int argc, const char* const* argv) {
    std::vector<const char*> argvExt(argv, argv+argc);
    argvExt.push_back("-fplugin=etc/cling/plugins/lib/clad.so");
    cling::Interpreter interp(argvExt.size(), &argvExt[0], LLVMDIR);
    gimme_pow2dx(interp);
    return 0;
}
```

CaaS. Clad Extension for AutoDiff

```
#include <...>
// Derivatives as a service.
void gimme_pow2dx(cling::Interpreter &interp) {
    // Definitions of declarations injected also into cling.
    interp.declare("double pow2(double x) { return x*x; }");
    interp.declare("#include <clad/Differentiator/Differentiator.h>");
    interp.declare("#pragma cling optimize(2)");
    interp.declare("auto dfdx = clad::differentiate(pow2, 0);");

    cling::Value res; // Will hold the evaluation result.
    interp.process("dfdx.getFunctionPtr()", &res);

    using func_t = double(double);
    func_t* pFunc = res.getAs<func_t*>();
    printf("dfdx at 1 = %f\n", pFunc(1));

    interp.process("dfdx.getCode()", &res);
    printf("dfdx code: %s\n %s\n", res.getAs<const char*>());
}
```

```
vvassilev@vv-nuc ~/.../builddir $ ./caas-demo
dfdx at 1 = 2.000000
dfdx code: double pow2_darg0(double x) {
            double _d_x = 1;
            return _d_x * x + x * _d_x;
}

vvassilev@vv-nuc ~/.../builddir $
```

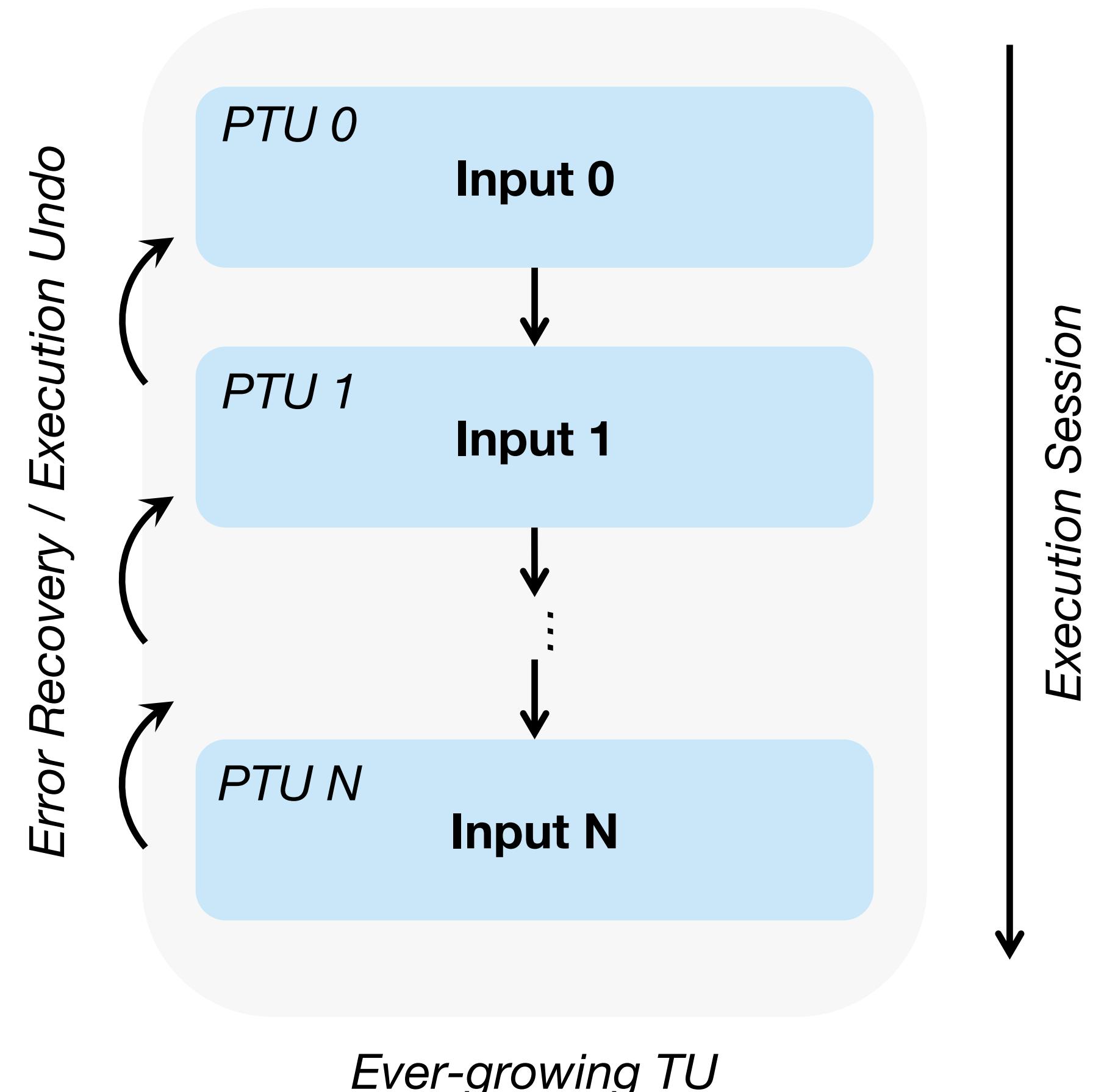
Evolving Cling Into CaaS and Clang-Repl in LLVM Mainline

Evolving Cling Into CaaS and Clang-Repl

- Generalize Cling in a tool available in LLVM mainline (`clang/tools/clang-repl`)
- Consolidate various incremental compilation APIs in Clang (`clang/lib/Interpreter`)
- Advance the incremental compilation support in Clang
- `libclangInterpreter` and `clang-repl` are available in LLVM13

Ever-growing TU in Clang

- We can split the translation unit into a sequence of partial translation units (PTU)
- Processing a PTU might extend an earlier PTU (template instantiation)
- Each PTU can have its own allocator



Incremental Compilation in Clang

```
#include "clang/Interpreter/Interpreter.h"
// ...
int main() {
    std::vector<const char*> Args;
    auto CI = clang::IncrementalCompilerBuilder::create(Args);
    auto Interp = clang::Interpreter::create(std::move(CI));
    auto PTU = Interp->Parse("extern \"C\" int printf(const char*,...);");
    Interp->ParseAndExecute("auto r = printf(\"Hello interpreted world\");");
    // prints 'Hello interpreted world'
}
```

Instantiating a C++ template in C

```
// gcc ... template_instantiate_demo.c
#include "InterpreterUtils.h" // libInterOp.so

int main(int argc, char **argv) {
    Clang_Parse("void* operator new(__SIZE_TYPE__,
void* __p);"
    "extern \"C\" int printf(const char*,...);"
    "class A {};" 
    "\n #include <typeinfo> \n"
    "struct B {"
    "    template<typename T>"
    "    void callme(T) {"
    "        printf(\" Instantiated with [%s] \\n \",
typeid(T).name());"
    "    }"
    "};"
    const char * InstArgs = "A*";
    Decl_t T = Clang_LookupName("A");
    Decl_t TemplatdClass = Clang_LookupName("B");
```

```
// ...
// Instantiate B::callme with the given types
Decl_t Inst
    = Clang_InstantiateTemplate(TemplatedClass,
"callme", InstArgs);

// Get the symbol to call
typedef void (*fn_def)(void*);
fn_def callme_fn_ptr
    = (fn_def) Clang_GetFunctionAddress(Inst);

// Create object of type T
void* NewT = Clang_CreateObject(T);

callme_fn_ptr(NewT);

return 0;
}
```

```
vvassilev@vv-nuc ~/.../cpptemplate $ LD_LIBRARY_PATH=." ./template_instantiate_demo.out
Instantiated with [P1A]
vvassilev@vv-nuc ~/.../cpptemplate $ LD_LIBRARY_PATH=." ./template_instantiate_demo.out "class MyClass1{};" "MyClass1" "MyClass1*"
Instantiated with [P8MyClass1]
vvassilev@vv-nuc ~/.../cpptemplate $
```

Instantiating a C++ template in Python

```
# template_instantiate_demo.py
import ctypes

libInterop = ctypes.CDLL("./libInterop.so")
# tell ctypes which function to call and what are the
# expected in/out types.
_cpp_compile = libInterop.Clang_Parse
_cpp_compile.argtypes = [ctypes.c_char_p]

def cpp_compile(arg):
    return _cpp_compile(arg.encode("ascii"))

# define some classes to play with
cpp_compile(r"""
void* operator new(__SIZE_TYPE__, void* __p);
extern "C" int printf(const char*, ...);
class A {};
class B {
public:
    template<typename T, typename S>
    void callme(T, S) { printf(" callme in B! \n"); }
};
""")
```

```
# initialize our C++ interoperability layer wrapper
gIL = InterOpLayerWrapper()

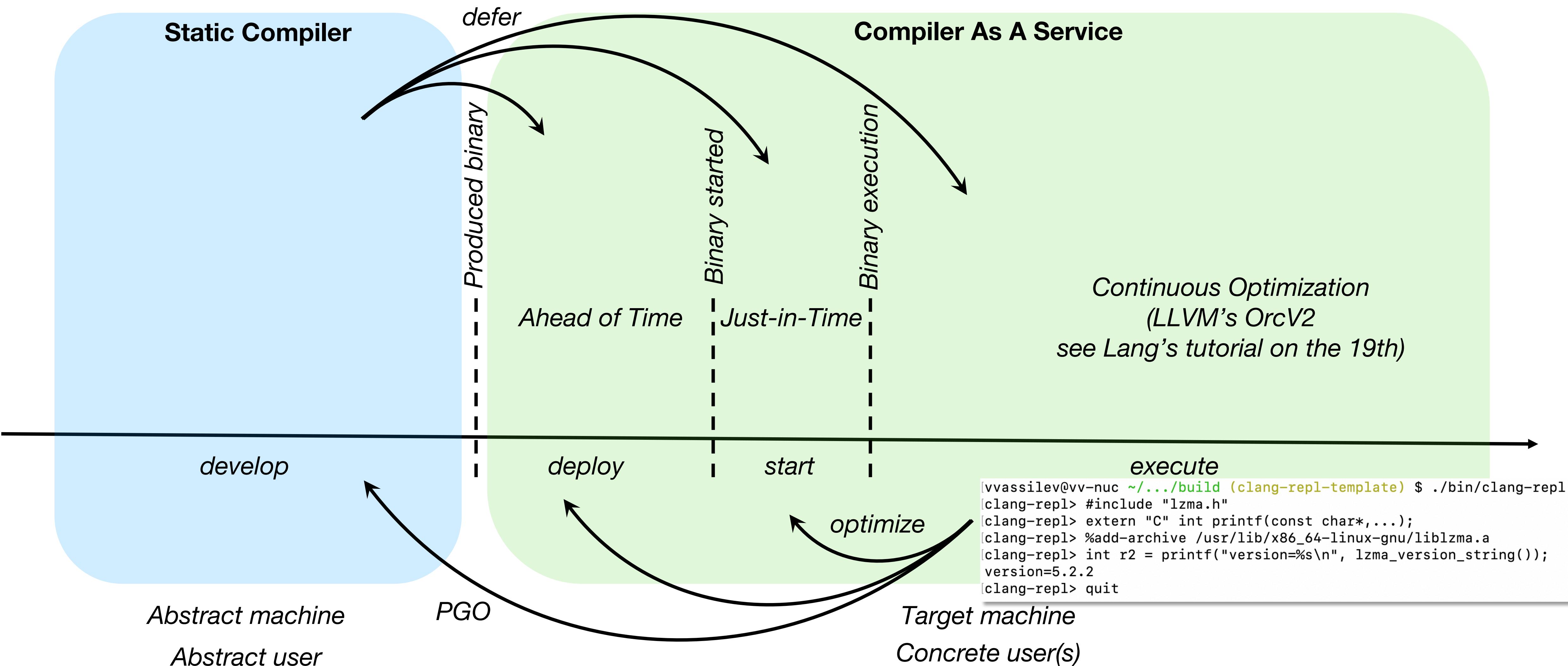
if __name__ == '__main__':
    # create a couple of types to play with
    A = type('A', (), {
        'handle' : gIL.get_scope('A'),
        '__new__' : cpp_allocate
    })
    h = gIL.get_scope('B')
    B = type('B', (A,), {
        'handle' : h,
        '__new__' : cpp_allocate,
        'callme' : TemplateWrapper(h, 'callme')
    })
    # call templates
    a = A()
    b = B()

    # explicit template instantiation
    b.callme['A, int'](a, 42)

    # implicit template instantiation
    b.callme(a, 42)
```

```
[vvassilev@vv-nuc ~/.../cpptemplate $ python3 template_instantiate_demo.py
callme in B!
callme in B!
vvassilev@vv-nuc ~/.../cpptemplate $
```

Lifelong Optimization



Summary

- Interactive C++ is more than just a REPL
- CaaS allows to defer computations until runtime, possibly improving performance and reducing binary sizes (template instantiations)
- CaaS offers ways to extend the language for a particular use or domain

Thank You!

Selected References

- <https://blog.llvm.org/posts/2020-11-30-interactive-cpp-with-cling/>
- <https://blog.llvm.org/posts/2020-12-21-interactive-cpp-for-data-science/>
- <https://blog.llvm.org/posts/2021-03-25-cling-beyond-just-interpreting-cpp/>
- <https://Compiler-Research.org>
- <https://root.cern>

Q&A

Backup

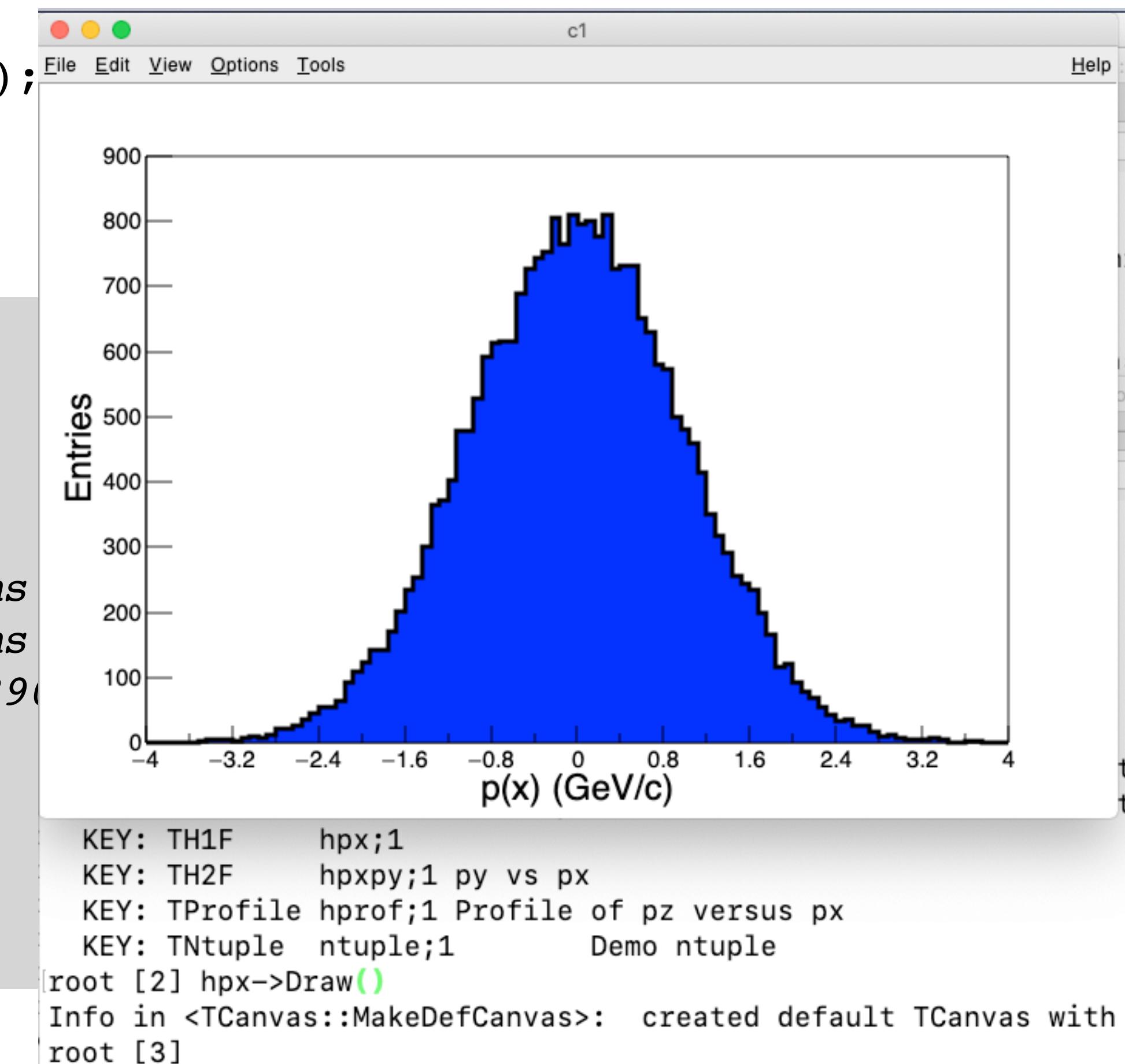
ROOT – Scientific Data Analysis



- The ROOT data analysis package embeds Cling to enable interactive C++ but also to use it as a reflection information service for data serialization
- The ROOT and Cling technology are used to store around 1EB physics data facilitating more than 1000 scientific publications last 7 years
- The ROOT package is developed and maintained by the field of high-energy physics and organizations such as CERN, FNAL, GSI, University of Nebraska, UC San Diego, Princeton

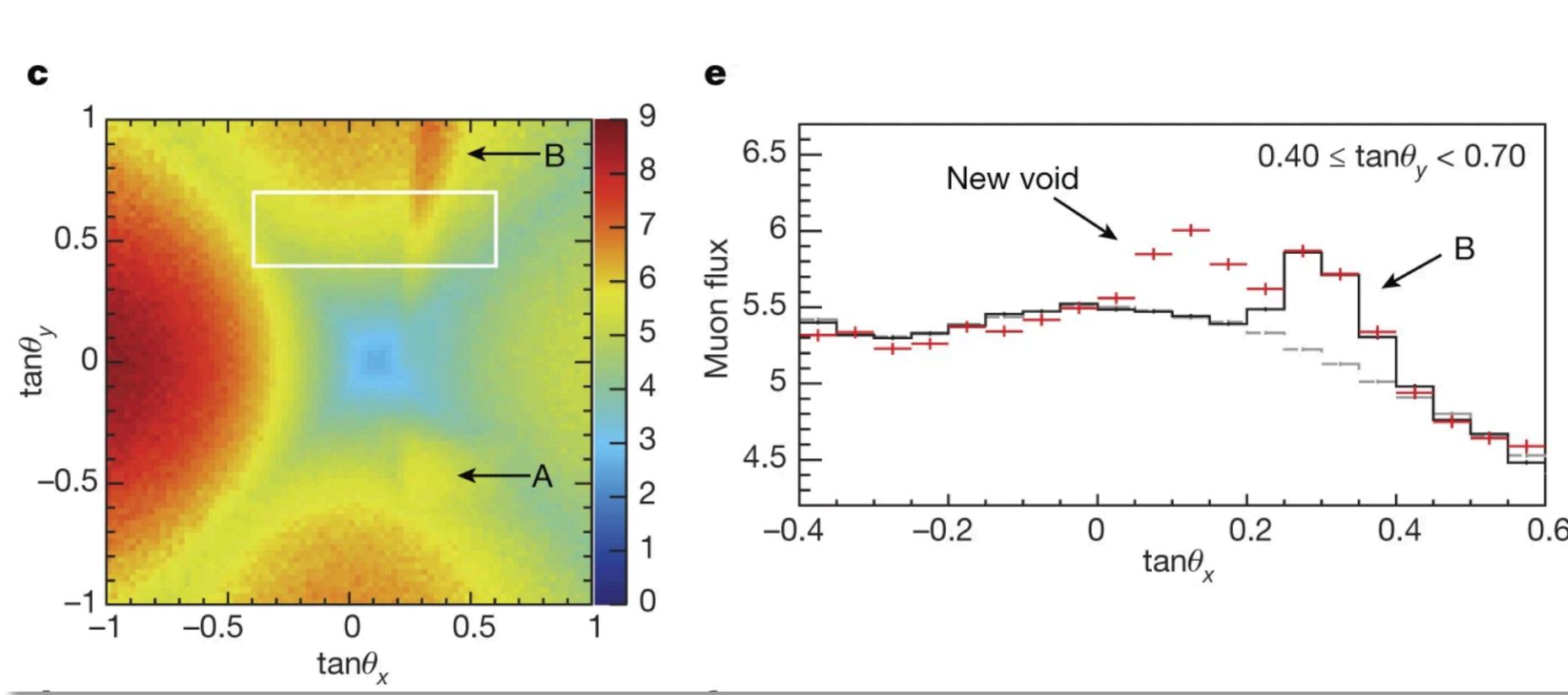
Dynamic Scopes. Runtime Lookup

```
gCling->EvaluateT</*ret type*/void>( "ntuple->GetTitle()", /*context*/);  
  
[root] ntuple->GetTitle()  
error: use of undeclared identifier 'ntuple'  
[root] TFile::Open("tutorials/hsimple.root"); ntuple->GetTitle()  
(const char *) "Demo ntuple"  
[root] gFile->ls();  
TFile** tutorials/hsimple.root Demo ROOT file with histograms  
TFile* tutorials/hsimple.root Demo ROOT file with histograms  
OBJ: TH1F hpx This is the px distribution : 0 at: 0x7fadbb84e390  
OBJ: TNtuple ntuple Demo ntuple : 0 at: 0x7fadbb93a890  
KEY: TH1F hpx;1 This is the px distribution  
[...]  
KEY: TNtuple ntuple;1 Demo ntuple  
[root] hpx->Draw()
```

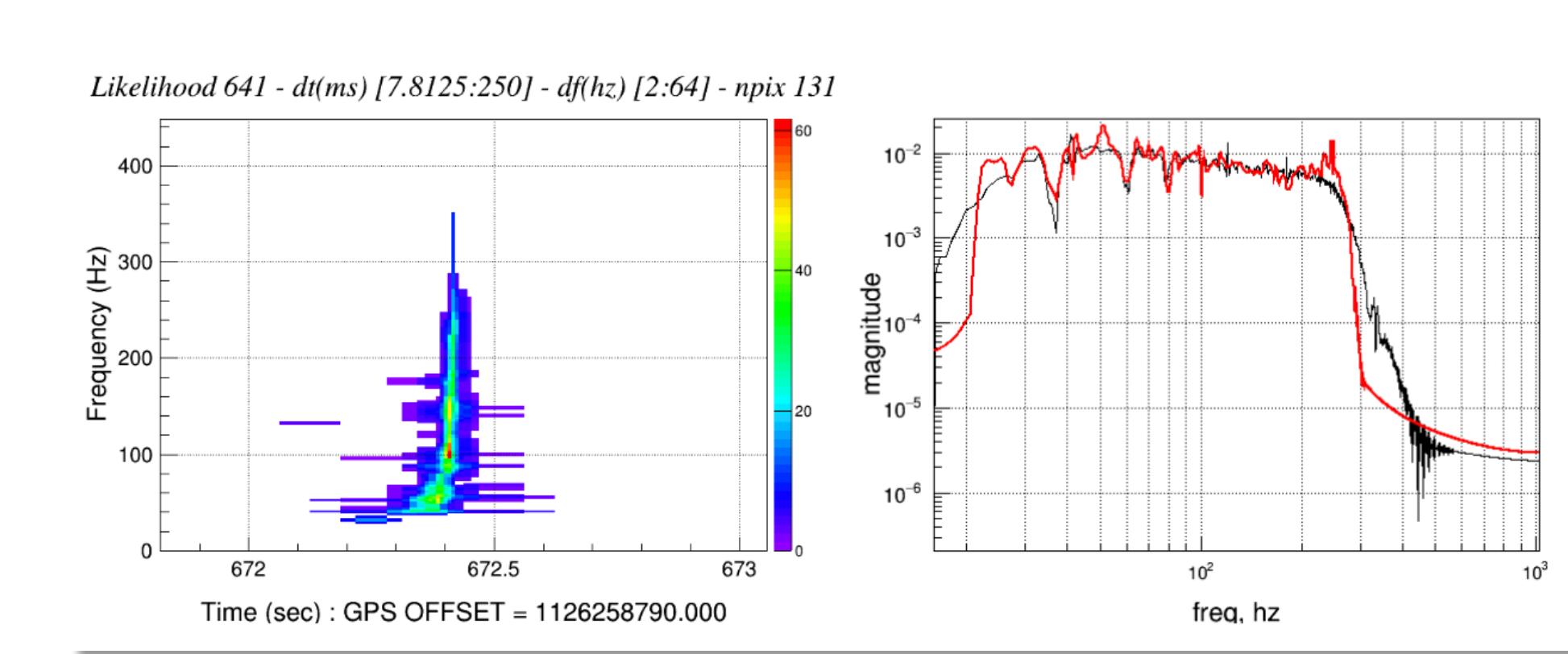


Eval-style programming enables Cling to be embedded in frameworks.

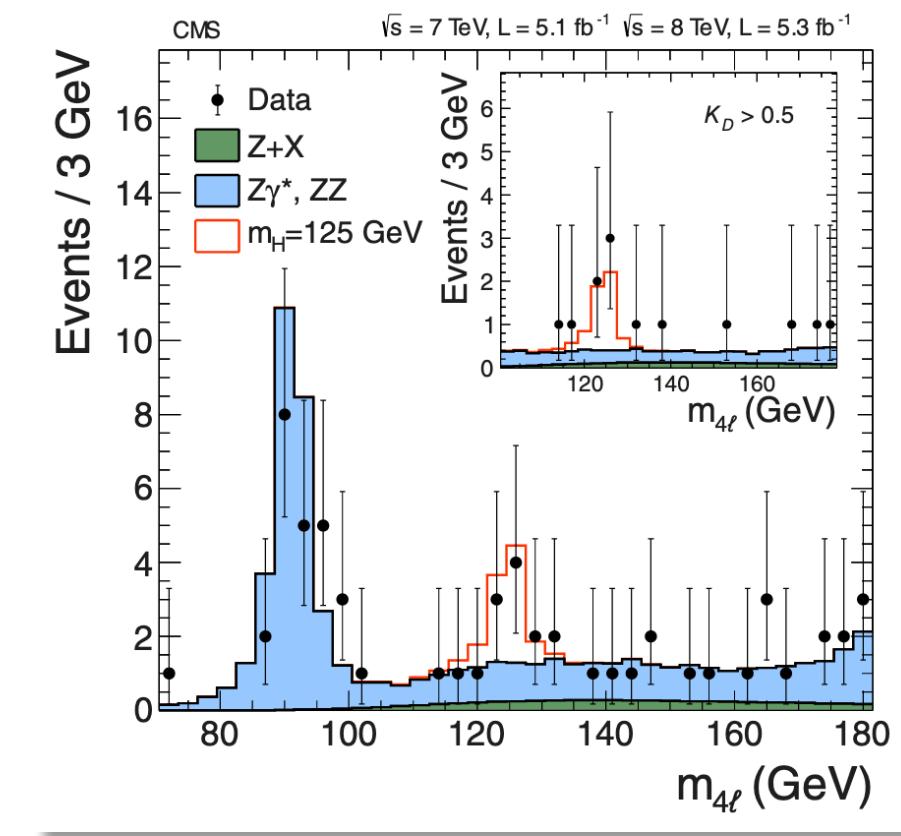
Impact of Interactive C++ in Physics



[1]



[2]



[3]

Scientific breakthroughs such as the discovery of the big void in the Khufu's Pyramid, the 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

Interpreting C++. Cling

- Cling was originally developed in the field of high energy physics to enable interactivity, dynamic interoperability and rapid prototyping capabilities to C++ developers.
- Cling supports the full C++ feature set including the use of templates, lambdas, and virtual inheritance.
- Cling adds a small set of extensions in C++ to allow interactive exploration and makes the language more welcoming for use.
- Cling compiles C++ code incrementally and relies on JIT compilation.
- Cling enables exploratory programming for C++.