PRINCETON UNIVERSITY

iris hep

Awkward Just-In-Time (JIT) Compilation:

A Developer's Experience

Norfolk, Virginia, USA • **May 8-12, 2023**

CHEP 2023

Computing in High Energy & Nuclear Physics

Angus Hollands, Ioana Ifrim, Ianna Osborne, Jim Pivarski, Henry Schreiner

PRINCETON UNIVERSITY

iris hep

Awkward Array

array-oriented programming

- Awkward Array is a library for performing NumPy-like computations on nested, variable-sized data, enabling array-oriented programming on arbitrary data structures in Python

  - The New Awkward Ecosystem by Ioana Ifrim - next talk
  - Analysis of physics analysis by Jim Pivarski
  - Fine-Grained HEP Analysis Task Graph Optimization with Coffea and Dask, by Lindsey Gray


```
>>> import awkward as ak
>>> A = ak.Array([1, 2, 3])
>>> B = ak.Array([3, 2, 1])
>>> C = A + B
>>> C
<Array [4, 4, 4] type='3 * int64'>
```

```
>>> array = ak.Array(
...     [
...         [{"x": 1, "y": [1.1]}, {"x": 2, "y": [2.2, 0.2]}],
...         [],
...         [{"x": 3, "y": [3.0, 0.3, 3.3]}],
...     ]
... )
>>> array.x*array.x + array.y*array.y
<Array [[[2.21], [8.84, 4.04]], ..., [[…]]]
                                type='3 * var * var * float64'>
```

```
def jet_pt_resolution(pt):
    # normal distribution with 5% variations, shape matches jets
    counts = ak.num(pt)
    pt_flat = ak.flatten(pt)
    resolution_variation = np.random.normal(np.ones_like(pt_flat), 0.05)
    return ak.unflatten(resolution_variation, counts)

class TtbarAnalysis(processor.ProcessorABC):
    ...
    def process(self, events):
        ...
        events["pt_res_up"] = jet_pt_resolution(events.Jet.pt)
```

Awkward Array

imperative solutions

- Imperative (procedural) code can sometimes be easier to write or faster to run
- Performant imperative programming requires compilation
- JIT-compilation makes it convenient to compile in an interactive Python environment

```
>>> for i in range(10):
...   if array[i].nMuon == 2:
...     if array[i].Muon_charge[0] != array[i].Muon_charge[1]:
...       print(array[i].Muon_pt)
...
[10.5, 16.3]
[57.6, 53]
[11.3, 23.9]
[10.2, 14.2]
[11.5, 3.47]
```

```
df = ak.to_rdataframe({"Events": array})

rdf = (df.Filter('Events.nMuon() == 2')
    .Filter('Events.Muon_charge()[0] != Events.Muon_charge()[1]')
    .Define("dimuon_mass", """
return std::sqrt(2 * Events.Muon_pt()[0] * Events.Muon_pt()[1]
    * (std::cosh(Events.Muon_eta()[0] - Events.Muon_eta()[1])
    - std::cos(Events.Muon_phi()[0] - Events.Muon_phi()[1])));
"""))
```

```
>>> array.show(type=True)
type: 61540413 * {
    Muon_charge: var * int32,
    Muon_eta: var * float32,
    Muon_mass: var * float32,
    Muon_phi: var * float32,
    Muon_pt: var * float32,
    nMuon: uint32
}
```

slow

fast

```
>>> array[:10].Muon_pt
<Array [[10.8, 15.7], [10.5, ...], ..., [11.5, 3.47]] type='10 * var * float32'>
```

PRINCETON UNIVERSITY

iris hep

Awkward Array Acceleration

Just-In-Time techniques

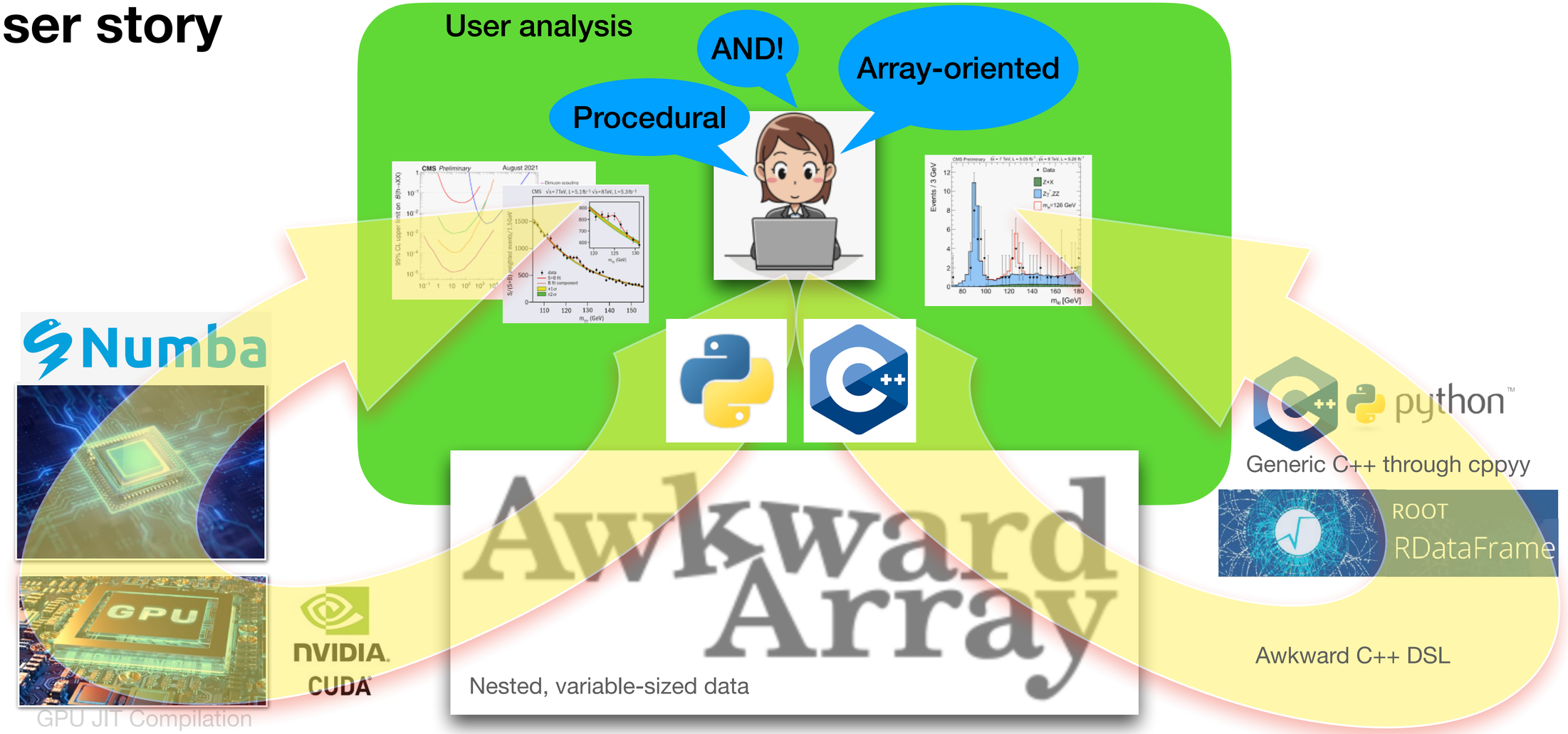
- Several functions in Awkward Array JIT-compile a user's code into executable machine code. They use different techniques, but reuse parts of each others' implementations.
- We will discuss the techniques used, focusing on RDataFrame, cppyy, and Numba, particularly Numba on GPUs:
  - Conversions of Awkward Arrays to and from RDataFrame
  - Standalone cppyy
  - Passing Awkward Arrays to and from Python functions compiled by Numba
  - Passing Awkward Arrays to Python functions compiled for GPUs by Numba
  - Populating Awkward Arrays from C++ without any Python dependencies (header-only)

PRINCETON UNIVERSITY

iris hep

Awkward Array Acceleration

user story



PRINCETON UNIVERSITY

iris hep

Awkward Array to and from RDataFrame

faster execution using ROOT C++ functions

- ak.to_rdataframe function presents a view of an Awkward Array as an RDataFrame source
- ak.from_rdataframe function converts the selected columns as native Awkward Arrays
- *Why is it fast?* A zero-copy Awkward Array view and all for-loops on data are implemented in C++

```
df = ak.to_rdataframe({"Events": array})

rdf = df.Filter('Events.nMuon() == 2')\
    .Filter('Events.Muon_charge()[0] != Events.Muon_charge()[1]')\
    .Define("dimuon_mass", """
return std::sqrt(2 * Events.Muon_pt()[0] * Events.Muon_pt()[1]
    * (std::cosh(Events.Muon_eta()[0] - Events.Muon_eta()[1])
    - std::cos(Events.Muon_phi()[0] - Events.Muon_phi()[1])));
""")
```

```
df = ROOT.RDataFrame('Events', 'root://eospublic.cern.ch//eos/opendata/
cms/derived-data/AOD2NanoAODOutreachTool/
Run2012BC_DoubleMuParked_Muons.root')

>>> array = ak.from_rdataframe(
...     df,
...
columns=("Muon_charge","Muon_eta","Muon_mass","Muon_phi","Muon_pt","nMu
on",),)

>>> array.show(type=True)
type: 61540413 * {
    Muon_charge: var * int32,
    Muon_eta: var * float32,
    Muon_mass: var * float32,
    Muon_phi: var * float32,
    Muon_pt: var * float32,
    nMuon: uint32
}
[{Muon_charge: [-1, -1], Muon_eta: [1.07, -0.564], Muon_mass:
[...], ...},
 {Muon_charge: [1, -1], Muon_eta: [-0.428, 0.349], Muon_mass:
[...], ...},
 {Muon_charge: [1], Muon_eta: [2.21], Muon_mass: [0.106],
Muon_phi: ..., ...},
 {Muon_charge: [1, 1, 1, 1], Muon_eta: [-1.59, ...], Muon_mass:
[...], ...},
 {Muon_charge: [-1, -1, 1, 1], Muon_eta: [-2.17, ...], Muon_mass:
[...], ...},
...,
{Muon_charge: [-1, 1], Muon_eta: [-2.15, 0.291], Muon_mass:
[...], ...}]
```

PRINCETON UNIVERSITY

iris hep

Awkward Array and cppyy

faster execution writing C++ functions

- Awkward Arrays can be passed to a C++ (possibly templated) function defined by cppyy compiler
- A user does not need to know what cpp_type is
  - cpp_type is generated on demand when the Array needs to be passed to the function
- Based on cppyy 3.1.0

```
array = ak.Array(
  [
    [{"x": 1, "y": [1.1]}, {"x": 2, "y": [2.2, 0.2]}],
    [],
    [{"x": 3, "y": [3.0, 0.3, 3.3]}],
  ]
)

source_code_cpp = """
template<typename T>
double go_fast_cpp(T& awkward_array) {
 double out = 0.0;

 for (auto list : awkward_array) {
   for (auto record : list) {
     for (auto item : record.y()) {
       out += item;
     }
   }
 }

 return out;
}
"""
cppyy.cppdef(source_code_cpp)

out = cppyy.gbl.go_fast_cpp[array.cpp_type](array)
assert out == ak.sum(array["y"])
```

PRINCETON UNIVERSITY

iris hep

Awkward Array and Numba

speed up array-oriented & math-heavy functions written in Python

- Numba infers the argument types at call time, and generates optimized code based on this information
- Numba also compiles separate specializations depending on the input types
- Awkward Arrays can be passed to and from Python functions compiled by Numba

```
@numba.njit
def path_length(array):
  result = np.zeros(len(array), dtype=np.float32)
  for i, row in enumerate(array):
    result[i] = 0
    for j, val in enumerate(row):
      result[i] += val
  return result
```

Numba

PRINCETON UNIVERSITY

iris hep

Awkward Array and Numba CUDA

speed up Python functions on GPU

NVIDIA CUDA

- Passing Awkward Arrays to Python functions compiled for GPUs by Numba

```
N = 2**20
counts = ak.Array(cp.random.poisson(1.5, N).astype(np.int32))
content = ak.Array(cp.random.normal(0, 45.0, int(ak.sum(counts))).astype(np.float32))
array = ak.unflatten(content, counts)

@numba.cuda.jit(extensions=[ak.numba.cuda])
def path_length(out, array):
    tid = numba.cuda.grid(1)
    if tid < len(array):
        out[tid] = 0
        for i, x in enumerate(array[tid]):
            out[tid] += x

blocksize = 256
numblocks = (N + blocksize - 1) // blocksize

result = cp.empty(len(array), dtype=np.float32)

path_length[numblocks, blocksize](result, array)
```

PRINCETON UNIVERSITY

iris hep

Awkward Array Layout Builders

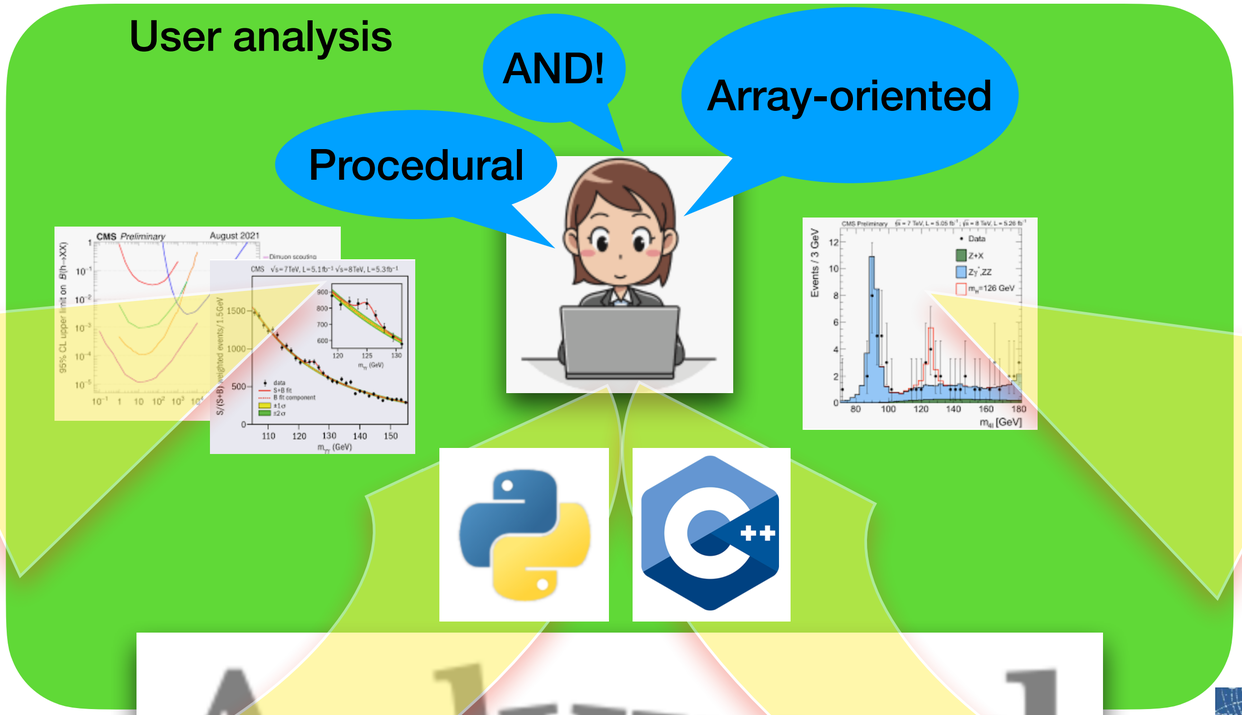
build arrays fast in C++

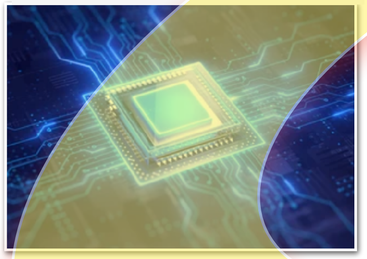
- Header-only libraries for populating Awkward Arrays from C++ without any Python dependencies
- And pass them to Python

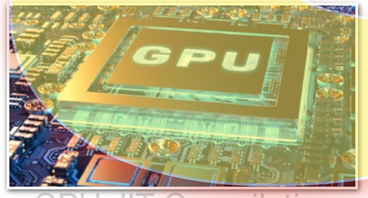
```cpp
#include "awkward/LayoutBuilder.h"

enum Field : std::size_t {one, two};

using UserDefinedMap = std::map<std::size_t, std::string>;

UserDefinedMap fields_map({
    {Field::one, "one"},
    {Field::two, "two"}
});

… // Type aliases omitted for brevity

RecordBuilder<
  RecordField<Field::one, NumpyBuilder<double>>,
  RecordField<Field::two, ListOffsetBuilder<int64_t,
      NumpyBuilder<int32_t>>>
> builder(fields_map);

auto& one_builder = builder.field<Field::one>();
auto& two_builder = builder.field<Field::two>();

one_builder.append(1.1);
auto& two_subbuilder = two_builder.begin_list();
two_subbuilder.append(1);
two_builder.end_list();

one_builder.append(2.2);
two_builder.begin_list();
two_subbuilder.append(1);
two_subbuilder.append(2);
two_builder.end_list();

one_builder.append(3.3);
```

Awkward Array Acceleration

developer view







GPU JIT Compilation

Generic C++ through cppyy

ROOT RDataFrame

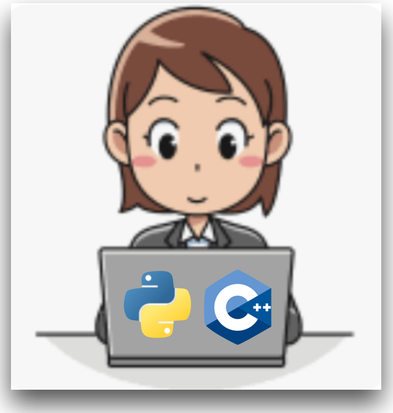
Awkward C++ DSL

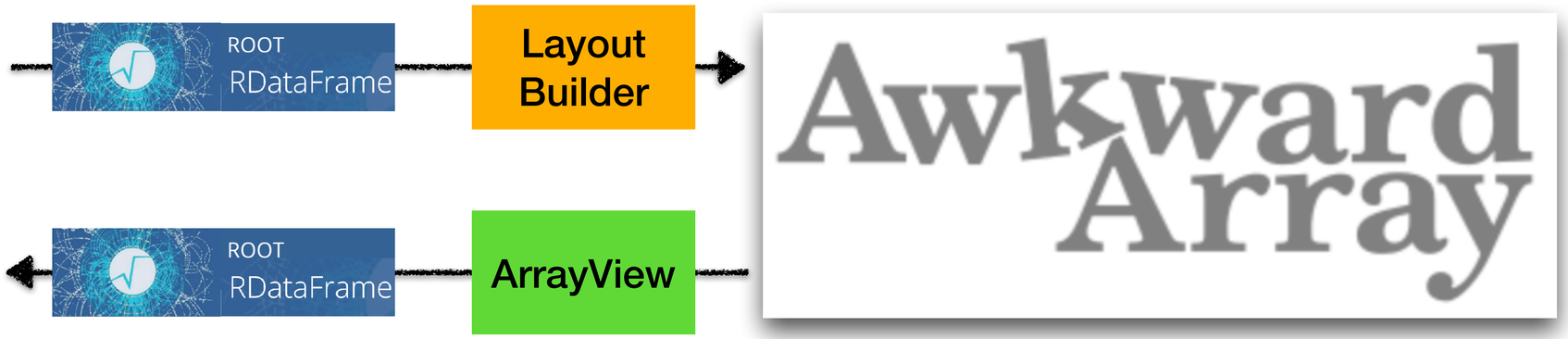
PRINCETON UNIVERSITY

iris hep

Awkward Array to and from RDataFrame

faster execution using ROOT C++ functions

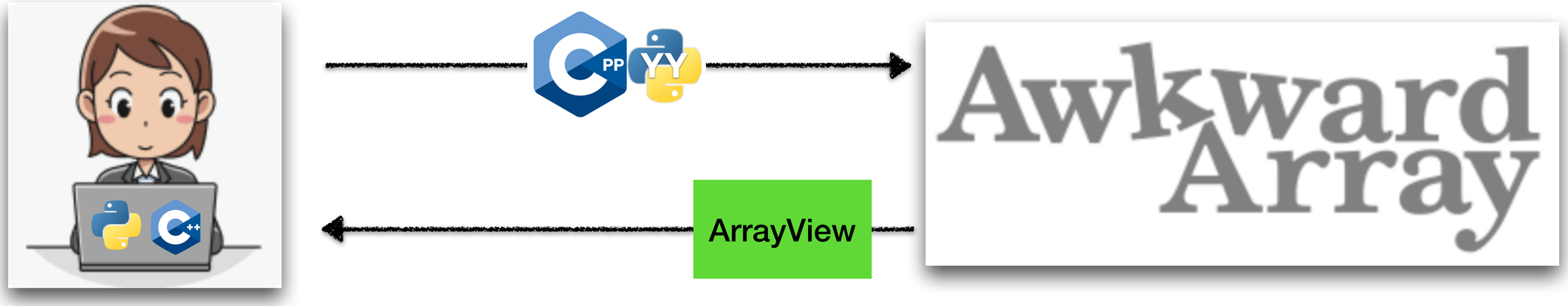




- The ArrayView is a lightweight 40-byte C++ object dynamically allocated on the stack
- The generated RDataSource takes pointers into the original array data via this view
- The C++ templated header-only implementation and the dynamically generated C++ code are used to extract the columns' types and data

Awkward Array and cppyy

faster execution writing C++ functions



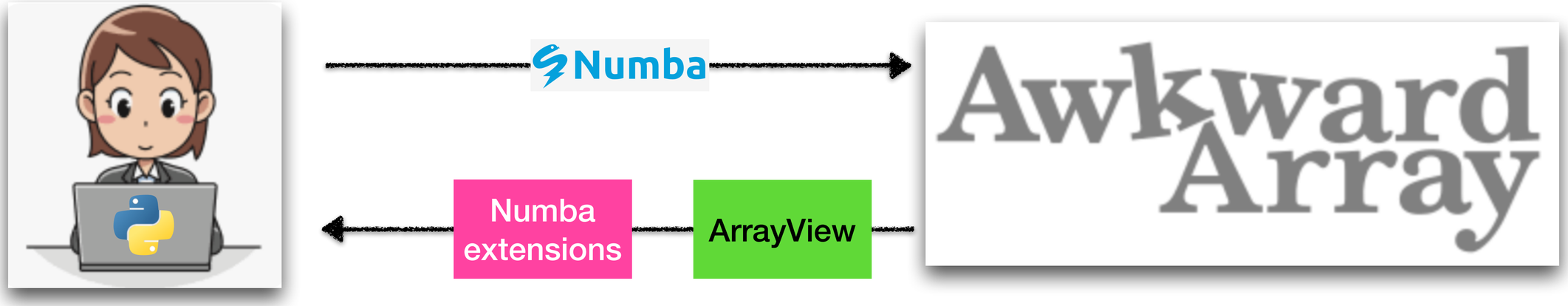
- The __cast_cpp__ method is called by cppyy to determine a C++ type of an ak.Array
- The ArrayView - the C++ type of an Awkward Array - is generated on demand when the array needs to be passed to a C++ (possibly templated) function defined by a `cppyy` compiler

PRINCETON UNIVERSITY

iris hep

Awkward Array and Numba

speed up array-oriented & math-heavy functions written in Python



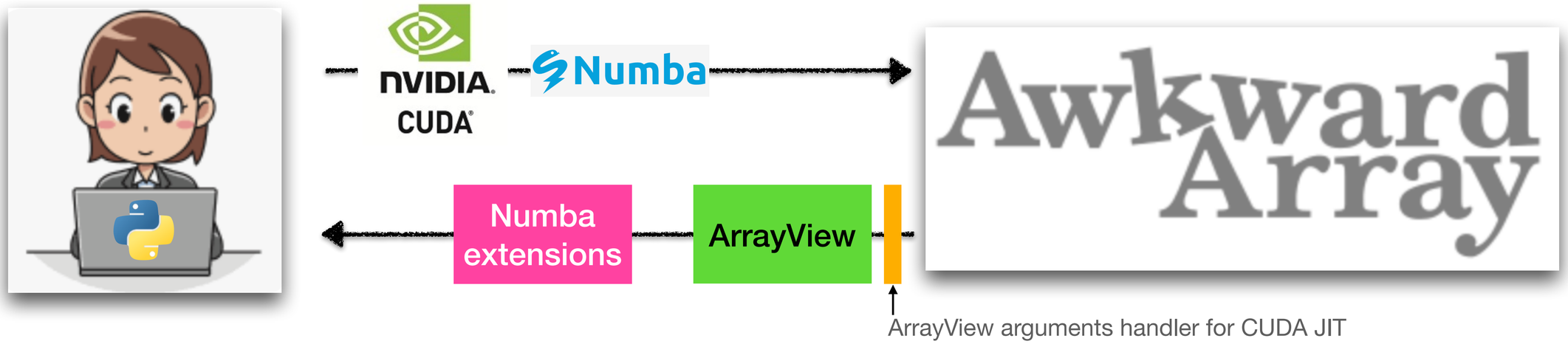
- Passing Awkward Arrays to and from Python functions compiled by Numba:
  - numba_type property

PRINCETON UNIVERSITY

iris hep

Awkward Array and Numba CUDA

speed up Python functions on GPU



- Passing Awkward Arrays to Python functions compiled for GPUs by Numba
- Awkward Numba CUDA extension prepares the ArrayView arguments before its lowering

PRINCETON UNIVERSITY

iris hep

Conclusions and Summary

Awkward Arrays

- Awkward Arrays - with its Awkward C++ dialect - are easy to use without compromising performance:
  - User can choose most suitable JIT-ed accelerator for the task at hand
  - Modular components are reused across the implementations
- The Awkward C++ implementations facilitate, and also highlight a clear roadmap for future developments, for example, a Layout builder in Numba, Kaitai - Awkward

Support for this work was provided by NSF cooperative agreement OAC-1836650 (IRIS-HEP)