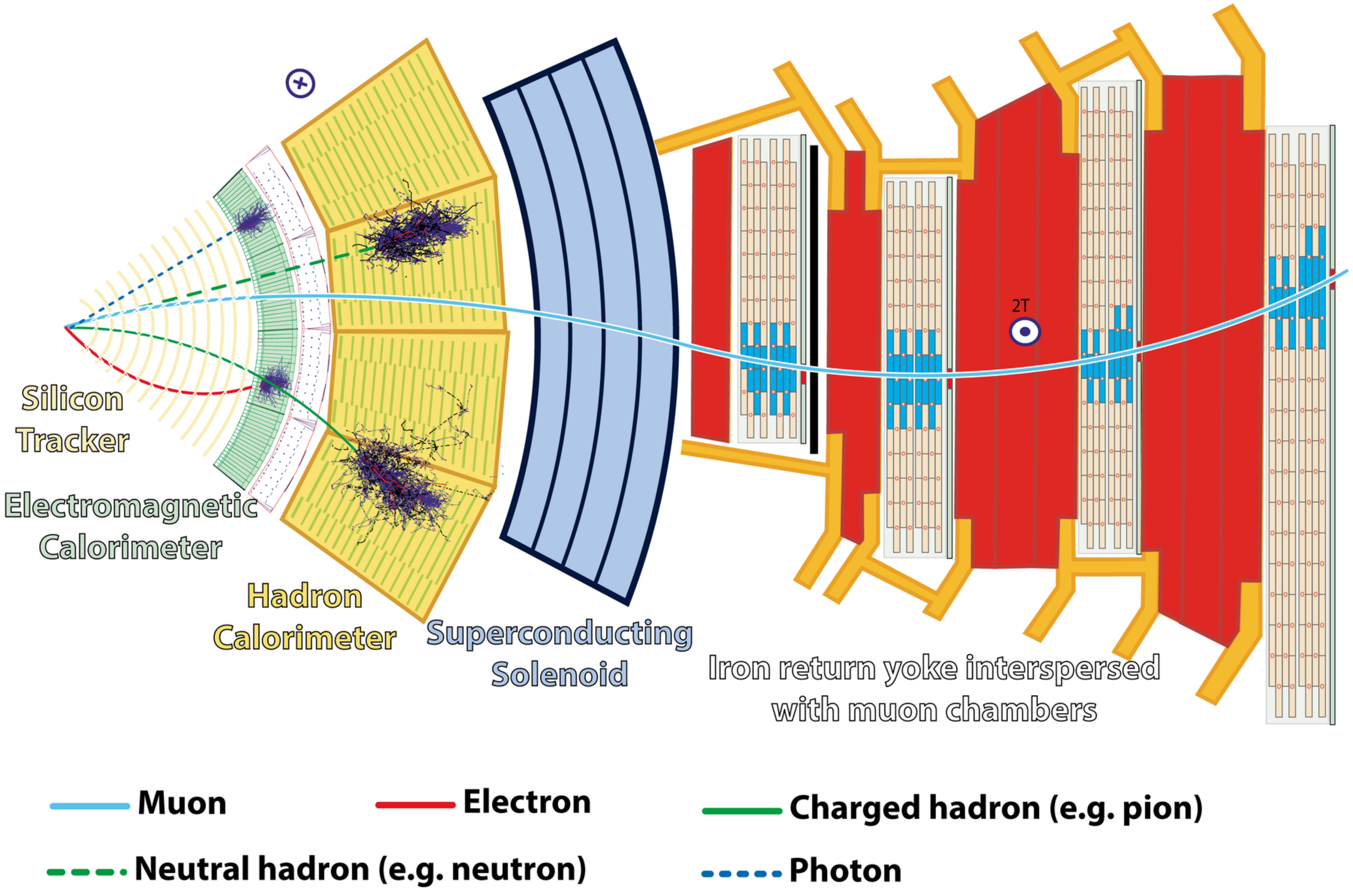
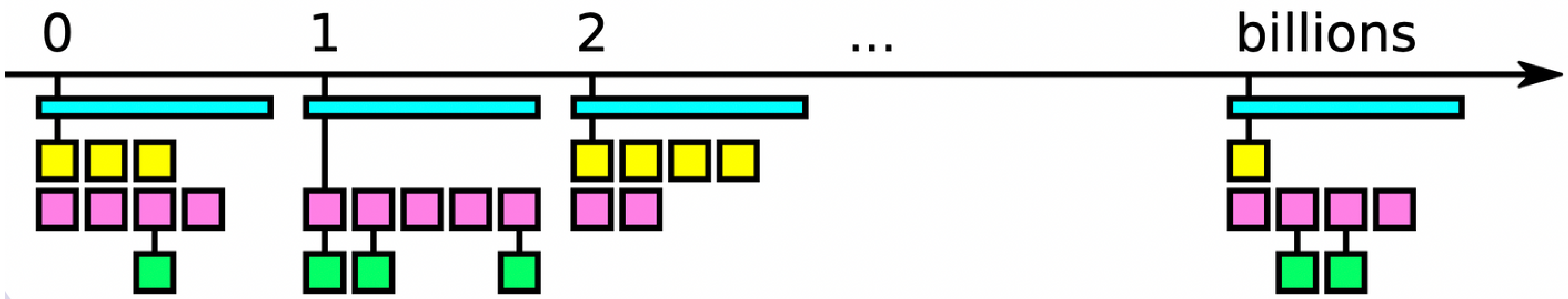
The New Awkward Ecosystem

Ioana Ifrim
Angus Hollands, Ianna Osborne, Jim Pivarski, Henry Schreiner
26th International Conference on Computing in High Energy & Nuclear Physics - May 2023, Norfolk, Virginia



In particle physics, data analysis frequently needs variable-length, nested data structures such as arbitrary numbers of particles per event and combinatorial operations to search for particle decay.



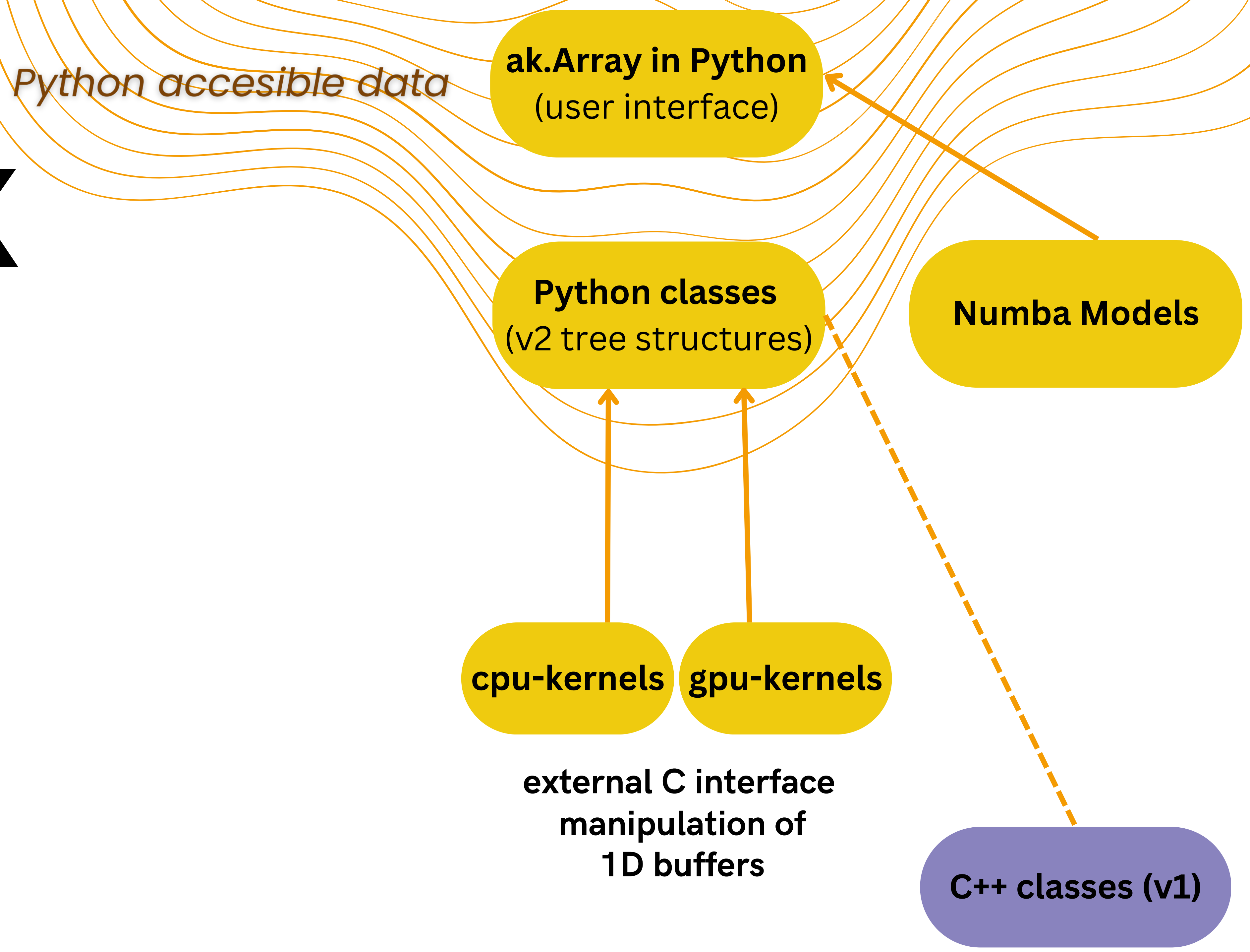
Awkward Array

Nested, variable-sized data, including arbitrary-length lists, records, mixed types, and missing data; Arrays are dynamically typed, but operations on them are compiled and fast. Their behaviour coincides with NumPy when array dimensions are regular and generalises when they are not.

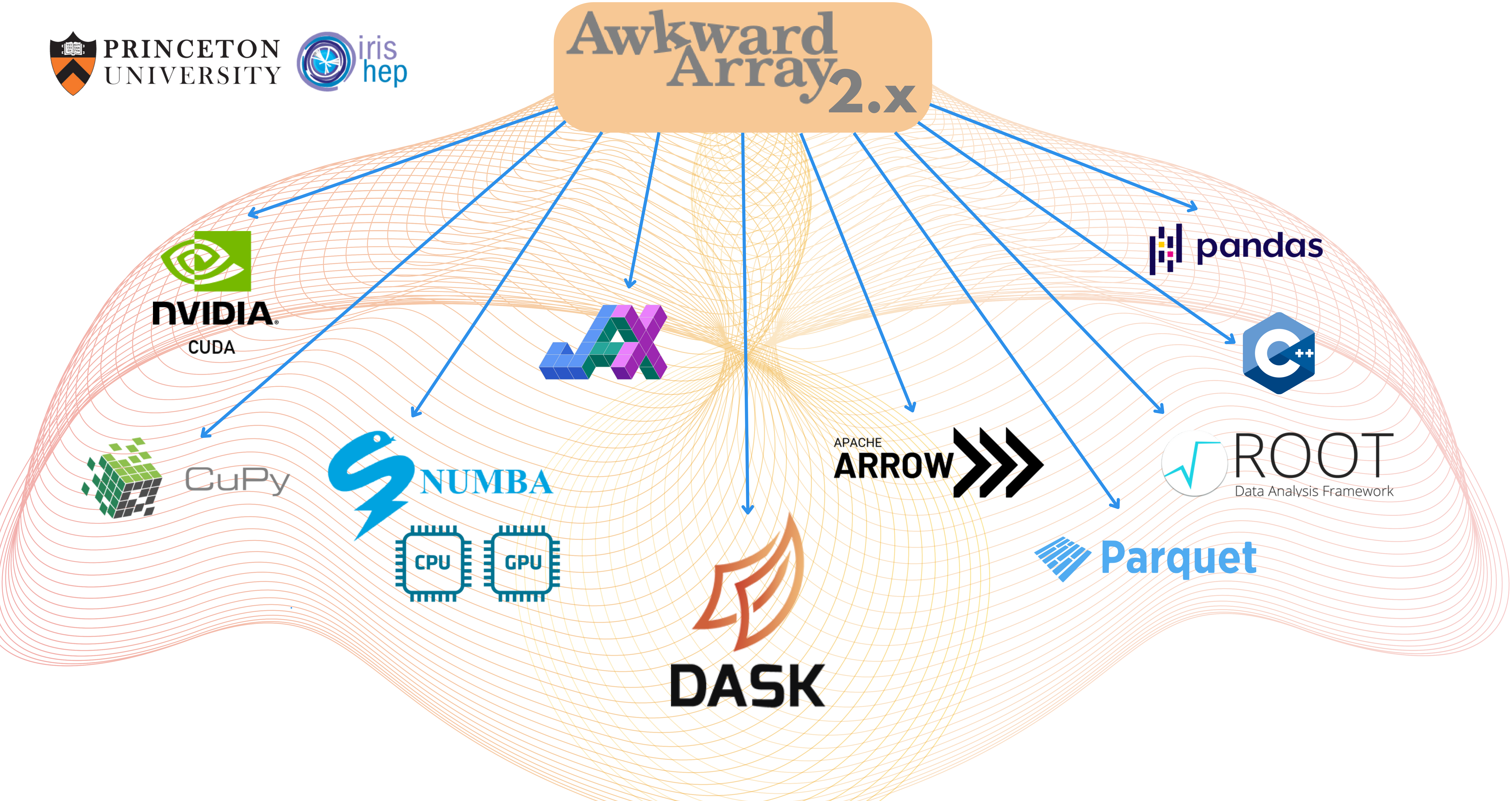
Awkward 2.x

Awkward Array has been deeply restructured to enable its integration with other libraries while preserving its existing high-level API and C++ performance-critical algorithms. In the latest 2.0 release, 50k LoC of C++ have been converted to 20 kLoC of Python.

Writing the “tree structures” in Python, rather than C++, allows us to use other Python libraries at this level., this translates into: INTEROPERABILITY capabilities = fluid data transfer between libraries



Fluid Data Transfer


PRINCETON UNIVERSITY
iris hep
Awkward Array 2.x
NVIDIA CUDA
CuPy
NUMBA
CPU
GPU
DASK
APACHE ARROW
Parquet
pandas
C++
ROOT
Data Analysis Framework

CUDA Integration

CUDA integration

Awkward Arrays can be copied to a GPU through Python functions compiled by Numba for GPUs or can be converted to/from CuPy arrays

```
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)
```

```
#to cupy
one = ak.Array([[1.1, 2.2, 3.3], [], [4.4, 5.5]], backend="cuda")
two = ak.Array([100, 200, 300], backend="cuda")
three = one + two
assert ak.to_list(three) == [[101.1, 102.2, 103.3], [], [304.4, 305.5]]
assert ak.backend(three) == "cuda"

#from cupy
cupy_array_2d = cp.array([[1.1, 2.2], [3.3, 4.4], [5.5, 6.6], [7.7, 8.8]])
ak_cupy_array_1d = ak.from_cupy(cupy_array_1d)
```

Conversion Facilities

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": None, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": None, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

Conversion facilities

Conversion facilities are now available for Awkward Arrays and:

- **Arrow**
- Parquet
- ROOT RDataFrame
- cppyy
- Pandas

```
>>> ak.to_arrow(array)
<awkward._connect.pyarrow.AwkwardArrowArray object at 0x7fbd7a6a1e80>
[
  -- is_valid: all not null
  -- child 0 type: extension<awkward<AwkwardArrowType>>
    [
      1.1,
      null,
      3.3
    ]
>>> ak.from_arrow(ar_arrow)
[[{x: 1.1, y: [1]}, {x: None, y: [...]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: None, y: [1, 2, 3, 4]}, {x: 5.5, y: [1, ..., 5]}]]
----------------------------------------------------------------
type: 3 * var * {
    x: ?float64,
    y: var * int64
}
```

Conversion facilities

Conversion facilities are now available for Awkward Arrays and:

- Arrow
- **Parquet**
- ROOT RDataFrame
- cppyy
- Pandas

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": None, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": None, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

```
>>> ak.to_parquet(array, "/tmp/example.parquet")
<pyarrow._parquet.FileMetaData object at 0x7fbd7a6b0270>
  created_by: parquet-cpp-arrow version 10.0.1
  num_columns: 2
  num_rows: 3
  num_row_groups: 1
  format_version: 2.6
  serialized_size: 0
```

```
>>> ak.from_parquet("/tmp/example.parquet")
[[{x: 1.1, y: [1]}, {x: None, y: [...]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: None, y: [1, 2, 3, 4]}, {x: 5.5, y: [1, ..., 5]}]]
-----------------------------------------------------------------
type: 3 * var * {
    x: ?float64,
    y: var * int64
}
```

PRINCETON UNIVERSITY

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": None, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": None, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

Conversion facilities

Conversion facilities are now available for Awkward Arrays and:

- Arrow
- Parquet
- **ROOT RDataFrame**
- cppyy
- Pandas

```
# The arrays given for each column have to be equal length:
assert len(array['x']) == len(array['y'][:][0])

#The dictionary key defines a column name in RDataFrame.
df = ak.to_rdataframe({'x':array['x'], ‘y’:array['y'][:][0]})

ak.from_rdataframe(df, columns=('x', 'y'))
[{y: [1], x: [1.1, None, 3.3]},
 {y: [1, 2], x: []},
 {y: [1, 2, 3], x: [None, 5.5]}]
--------------------------------
type: 3 * {
    y: var * int64,
    x: var * ?float64
}
```

ROOT
Data Analysis Framework

Conversion facilities

Conversion facilities are now available for Awkward Arrays and:

- Arrow
- Parquet
- ROOT RDataFrame
- **cppyy**
- Pandas

```
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"])
```

Conversion facilities

Conversion facilities are now available for Awkward Arrays and:

- Arrow
- Parquet
- ROOT RDataFrame
- cppyy
- **Pandas**

```
pandarray = akpd.from_awkward(array, name="awkward-pandas")

df = pd.DataFrame({"integers": np.arange(0, len(pandarray)),
                   "awkward": pandarray})
   integers    awkward
0         0    [{'x': 1.1, 'y': [1]}, {'x': None, 'y': [1, 2]...
1         1    []
2         2    [{'x': None, 'y': [1, 2, 3, 4]}, {'x': 5.5, 'y...
3         3    [{'x': 1.1, 'y': [1]}, {'x': None, 'y': [1, 2]...
4         4    []
5         5    [{'x': None, 'y': [1, 2, 3, 4]}, {'x': 5.5, 'y...

df.query("integers%2 == 0")
    integers    awkward
0         0    [{'x': 1.1, 'y': [1]}, {'x': None, 'y': [1, 2]...
2         2    [{'x': None, 'y': [1, 2, 3, 4]}, {'x': 5.5, 'y...
4         4    []

pandarray.ak.array.fields
['x', 'y']
pandarray.ak.array
[[{x: 1.1, y: [1]}, {x: None, y: [...]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: None, y: [1, 2, 3, 4]}, {x: 5.5, y: [1, ..., 5]}],
 [{x: 1.1, y: [1]}, {x: None, y: [...]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: None, y: [1, 2, 3, 4]}, {x: 5.5, y: [1, ..., 5]}]]
-----------------------------------------------------------------
type: 6 * var * {
    x: ?float64,
    y: var * int64
}
```

pandas

Task graphs with Dask

Eager Awkward

In the eager manner, the from_json call will immediately begin to read data from disk and decode the JSON. Sequentially, the selection step will execute

Awkward Array

```
from pathlib import Path
import awkward as ak
file = Path("data.00.json")
x = ak.from_json(file, line_delimited=True)
x = x[ak.num(x.foo) > 2]
```

Dask-Awkward

In dask, the reading (task graph creation) followed by the selection (extending the task graph) will be staged. Dask will execute the JSON reading and decoding of each file in parallel and upon completion the selection tasks will follow. Dask will schedule the tasks itself (and it will attempt to optimize its work)

Dask

```
import dask_awkward as dak
# dask-awkward only supports line-delimit
ed=True
x = dak.from_json("data.*.json")
x = x[dak.num(x.foo) > 2]

# With Dask we have to ask for the result
 with compute
x = x.compute()
```

Dask and TypeTracers

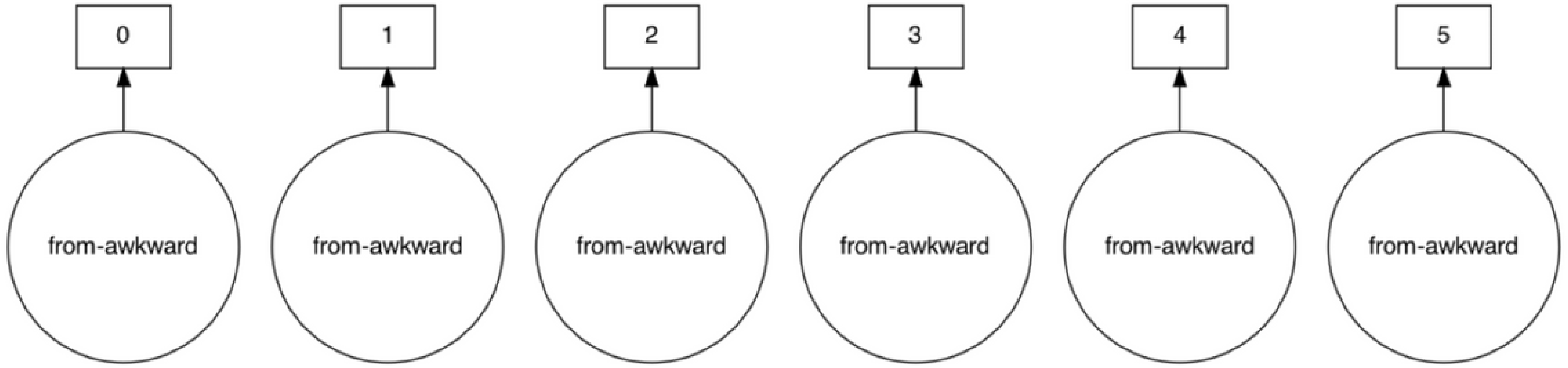
We can run the entire task graph on data-less typetracer arrays (Awkward Arrays without any data buffers, but with the same structure and types). Since there is no real data, the execution will be negligible compared to computing on real data from disk.

Necessary Columns Optimisation

The data-less execution of the graph helps determine which parts of a dataset sitting on disk are actually required to be read in order to successfully complete the compute. To know which fields get used in the graph, a mutable typetracer report object is attached to the first layer of the typetracer based graph. After executing the typetracer based graph, the report object tells us which exact fields were touched along the lifetime of the computation.

```
import awkward
import dask_awkward as dak
import dask
arr = awkward.Array([
    {"foo": 5, "bar": {"x": [-1, -2], "y": -2.2}},
    {"foo": 6, "bar": {"x": [-3], "y":  3.3}},
    {"foo": 7, "bar": {"x": [-5, -6, -7], "y": -4.4}},
    {"foo": 8, "bar": {"x": [8, 9, 10, 11, 12], "y":  5.5}},
]*10)
```

```
ds = dak.from_awkward(arr,6)
ds.visualize()
```




```
result = ds.bar.x / ds.foo
```

```
result.dask
```

HighLevelGraph

HighLevelGraph with 5 layers and 50 keys from all layers.

- ▸ **Layer1: from-awkward**
- ▸ **Layer2: foo**
- ▸ **Layer3: bar**
- ▸ **Layer4: x**
- ▸ **Layer5: divide**

JAX AD and Awkward

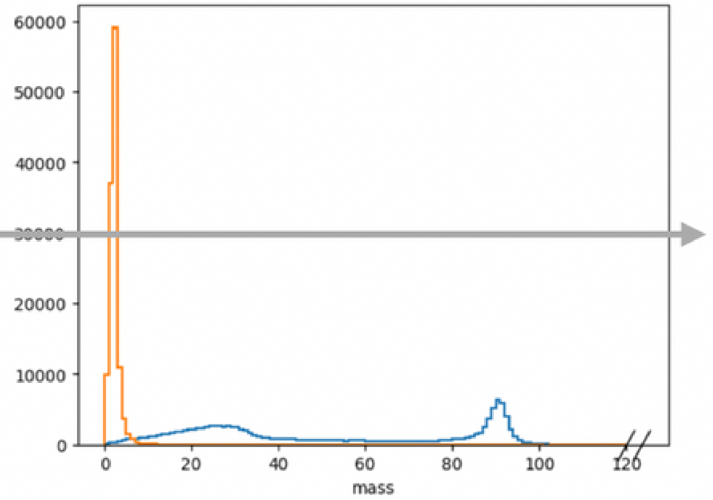
JAX

```python
import awkward as ak
import hist, jax, numpy, uproot

jax.config.update("jax_platform_name", "cpu")
jax.config.update("jax_enable_x64", True)
events = uproot.open("HiggsZZ4mu.root:Events")
events.show()
ak.jax.register_and_check()
# read data
muons = events.arrays(
    ["pt", "eta", "phi", "charge"],
    aliases={"pt": "Muon_pt", "eta": "Muon_eta", "phi": "Muon_phi", "charge": "Muon_charge"},
    array_cache=None,   # no cheating!
)
muons = ak.values_astype(ak.Array(muons.layout, backend="jax"), numpy.float64)

def f(muons):
    if (ak.sum(muons.charge[:, :2], axis=1) == 0).layout.backend.name == "jax":
        a = ak.Array((ak.num(muons.charge) >= 2).layout, backend="jax")
    else:
        a = (ak.num(muons.charge) >= 2)
    cut = a & (ak.sum(muons.charge[:, :2], axis=1) == 0)
    mu1, mu2 = muons[cut][:, 0], muons[cut][:, 1]
    return numpy.sqrt(2*mu1.pt*mu2.pt*(numpy.cosh(mu1.eta - mu2.eta) - numpy.cos(mu1.phi - 
mu2.phi)))

h = hist.Hist.new.Reg(120, 0, 120, name="mass").Double()
h1 = hist.Hist.new.Reg(120, 0, 120, name="mass").Double()

tan = ak.ones_like(muons)
primals, tangent = jax.jvp(f, (muons,), (tan,))
grad = ak.to_backend(tangent, backend="cpu")
original = ak.to_backend(muons, backend="cpu")

h.fill(f(original)).plot()
h1.fill(grad).plot()
```

Awkward Array implements support for the jax.jvp() and jax.vjp() JAX functions for computing forward/reverse-mode Jacobian-vector / vector-Jacobian products of functions that operate upon Awkward Arrays

The `backend` argument ensures that we build an Awkward Array which is backed by buffers of type jaxlib.xla_extension.DeviceArray, which power JAX's automatic differentiation and JIT compiling features.

Define a function that takes 2 muons and computes a Z peak (input is of type Awkward Array)



Compute the derivative of a Z peak. In the resulted plot, blue is the mass, orange is the derivative of the mass, with derivatives of the input parameters (pt1, pt2, eta1, ...) all being 1.

PRINCETON UNIVERSITY

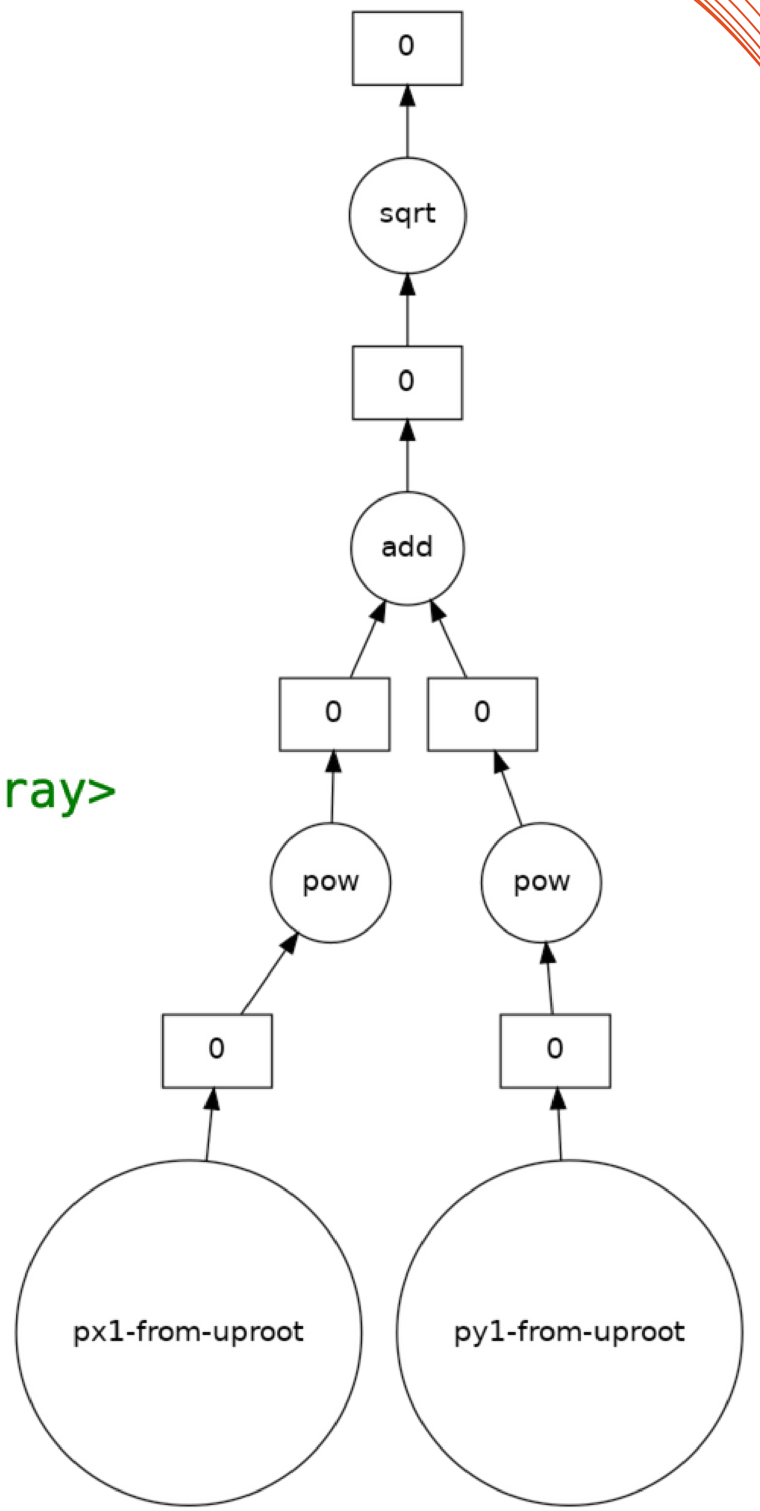
iris hep

Interop beyond Awkward

Uproot

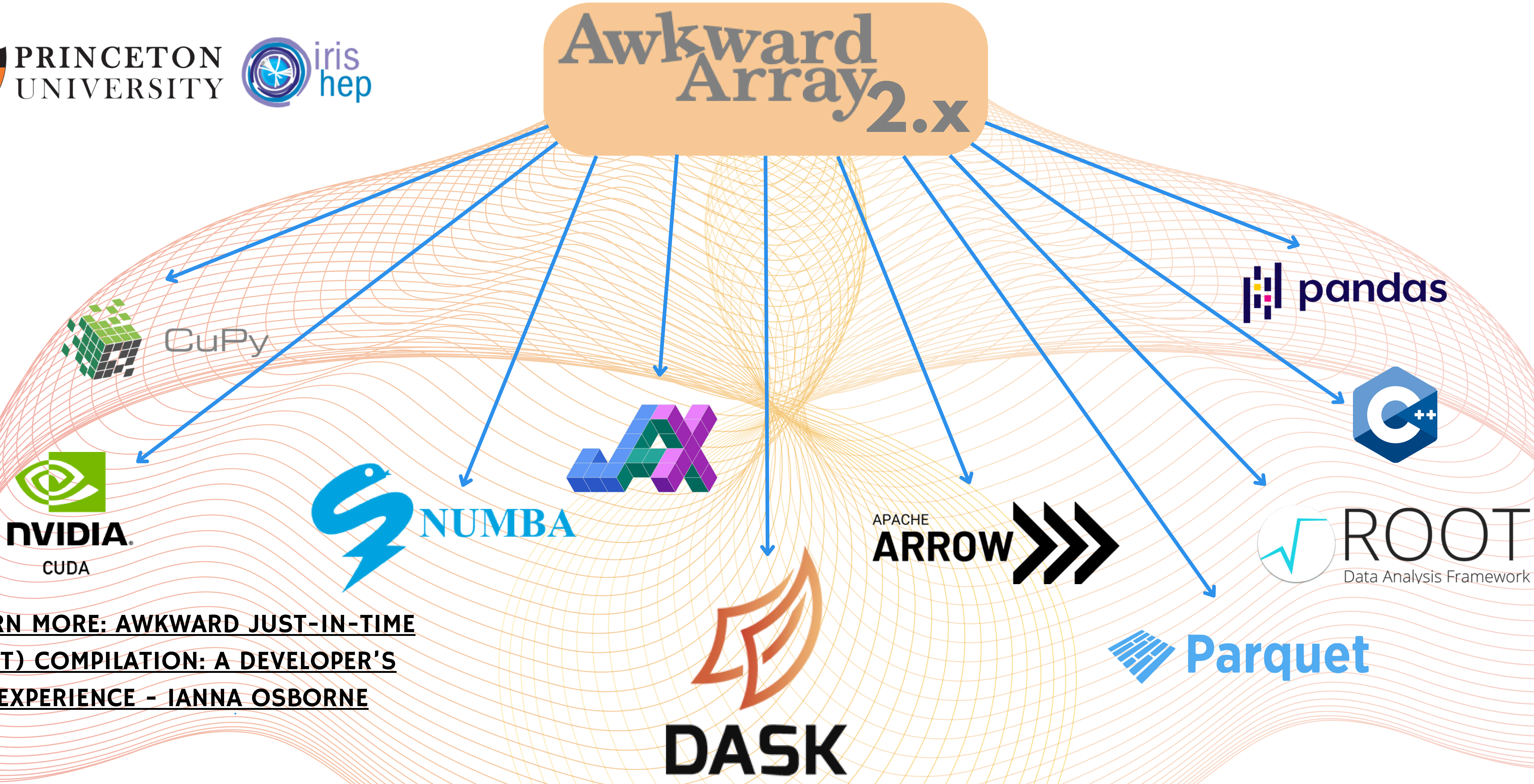
All the interoperability development from awkward 2.x is being passed on to uproot as well. Uproot supports reading TBranches into Dask collections with the uproot.dask function. If library='np', the array will be a dask.array, and if library='ak', the array will be a dak.Array. (library='pd' is in development, but the target would be dask.dataframe.)

```
dask_dict = uproot.dask(root_file, library='np')
px = dask_dict['px1']
py = dask_dict['py1']
pt = numpy.sqrt(px**2 + py**2)

# no data has been read yet
print(pt)
dask.array<sqrt, shape=(2304,), dtype=float64,
           chunksize=(2304,), chunktype=numpy.ndarray>

# Only after compute is called, the TBranch data
#is read and further computations are executed.
pt.compute()
array([44.7322, 38.8311, 38.8311, …, 32.5076])
```



Summary and Outlook

PRINCETON UNIVERSITY iris hep



LEARN MORE: AWKWARD JUST-IN-TIME (JIT) COMPILATION: A DEVELOPER'S EXPERIENCE – IANNA OSBORNE

LEARN MORE: FINE-GRAINED HEP ANALYSIS TASK GRAPH OPTIMIZATION WITH COFFEA AND DASK – LINDSEY GRAY

Outlook

The first libraries shown in this list have now been integrated within the ecosystem. While, TensorFlow's RaggedTensor and PyTorch's NestedTensor are the next conversion targets for Awkward Arrays.

Awkward Array

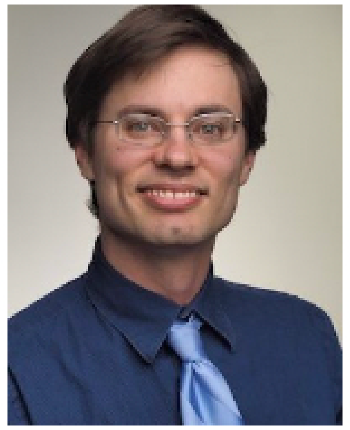
90.5%	numpy	4.2%	torch
56.9%	uproot	3.7%	seaborn
49.8%	matplotlib	3.6%	yahist
35.6%	coffea	3.2%	xgboost
31.2%	pandas	2.9%	sklearn
20.4%	mplhep	2.9%	h5py
11.9%	ROOT	2.6%	memory_profiler
11.8%	numba	2.3%	pympler
8.8%	hist	2.1%	psutil
8.4%	uproot_methods	1.9%	correctionlib
8.2%	yaml	1.8%	sortedcontainers
7.4%	utils	1.7%	cycler
6.7%	tqdm	1.7%	networkx
5.8%	boost_histogram	1.5%	pylab
5.0%	tensorflow	1.5%	PIL
4.8%	scipy	1.4%	helpers
4.3%	vector	1.4%	tabulate

Analysis of physics analysis - Jim Pivarski

Team

(research scientists)

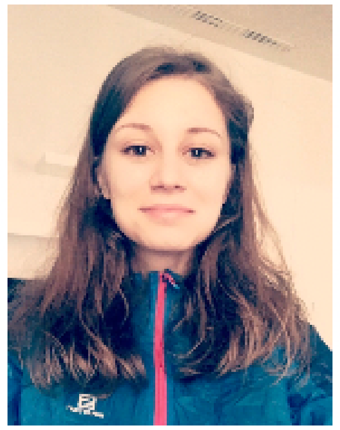
Jim Pivarski
Princeton University



Henry Schreiner
Princeton University

(research software engineers)

Ianna Osborne
Princeton University



Ioana Ifrim
Princeton University

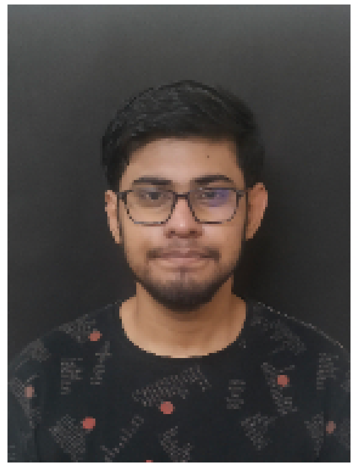
(postdoc)

Angus Hollands
University of Birmingham
→ Princeton University

(undergraduates)

Anish Biswas
Manipal Institute
of Technology

Manasvi Goyal
Delhi Technological
University

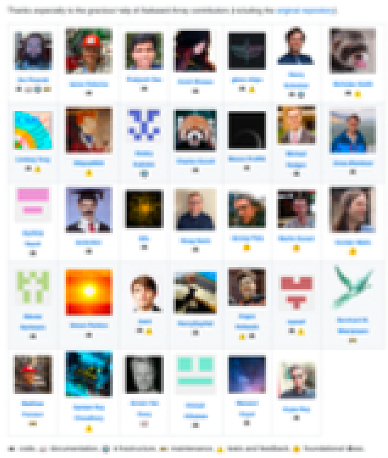


Aryan Roy
Manipal Institute
of Technology

(contractors)

Douglas Davis
Anaconda, Inc.

Martin Durant
Anaconda, Inc.



24 more contributors
on GitHub