# Creating an infrastructure for a **CUDA backend for Awkward Arrays**

Anish Biswas
Google Summer Of Code Participant
Manipal Institute Of Technology

Jim Pivarski
Google Summer Of Code Mentor
Princeton University

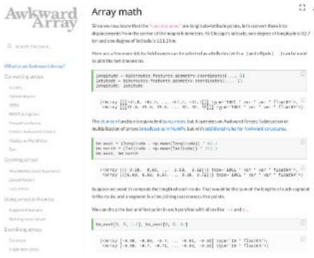
# What are Awkward Arrays?

Awkward Array

Scikit-HEP Project | NSF 1836650 | DOI 10.5281/zenodo.3952674 | python 2.7 3.5 3.6 3.7 3.8 | License BSD 3-Clause

Awkward Array is a library for **nested, variable-sized data**, including arbitrary-length lists, records, mixed types, and missing data, using **NumPy-like idioms**.

Arrays are **dynamically typed**, but operations on them are **compiled and fast**. Their behavior coincides with NumPy when array dimensions are regular and generalizes when they're not.




How-to tutorials

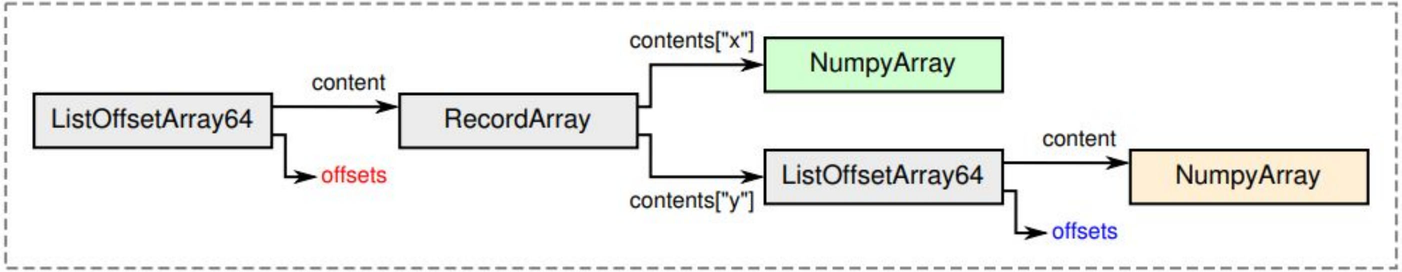

Python API reference

C++ API reference

# Transferring Buffers onto the GPU

Let's define an Awkward Array!

```
array = ak.Array([
        [{"x": 1,  "y": [11]},
         {"x": 4,  "y": [12, 22]},
         {"x": 9,  "y": [13, 23, 33]}],
        [],
        [{"x": 16, "y": [14, 24, 34, 44]}]
])
```




**CUDA is not good with complex Data Structures like this, but it is excellent for linear buffers!**

With Awkward Arrays, this transfer becomes very **simple and efficient!**

# Transferring Buffers onto the GPU

Here’s the internal representation of the Awkward Array, while it’s still in main memory!

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 3 3 4]" offset="0" length="4"/></offsets>
    <content><RecordArray>
        <field index="0" key="x">
            <NumpyArray format="l" shape="4" data="1 4 9 16"/>
        </field>
        <field index="1" key="y">
            <ListOffsetArray64>
                <offsets><Index64 i="[0 1 3 6 10]" offset="0" length="5"/></offsets>
                <content><NumpyArray format="l" shape="10" data="11 12 22 13 23 33 14 24 34 44"/></content>
            </ListOffsetArray64>
        </field>
    </RecordArray></content>
</ListOffsetArray64>
```

→ `ak.to_kernels(array, "cuda")` →

# Transferring Buffers onto the GPU

This is what you get after a transfer to GPU! **Notice the lib, under certain nodes!** That's what makes the entire transfer easy and efficient!

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 3 3 4]" offset="0" length="4">
        <Kernels lib="cuda" device="0" device_name="GeForce 940MX"/>
    </Index64></offsets>
    <content><RecordArray>
        <field index="0" key="x">
            <NumpyArray format="l" shape="4" data="1 4 9 16">
                <Kernels lib="cuda" device="0" device_name="GeForce 940MX"/>
            </NumpyArray>
        </field>
        <field index="1" key="y">
            <ListOffsetArray64>
                <offsets><Index64 i="[0 1 3 6 10]" offset="0" length="5">
                    <Kernels lib="cuda" device="0" device_name="GeForce 940MX"/>
                </Index64></offsets>
                <content><NumpyArray format="l" shape="10" data="11 12 22 13 23 33 14 24 34 44">
                    <Kernels lib="cuda" device="0" device_name="GeForce 940MX"/>
                </NumpyArray></content>
            </ListOffsetArray64>
        </field>
    </RecordArray></content>
</ListOffsetArray64>
```

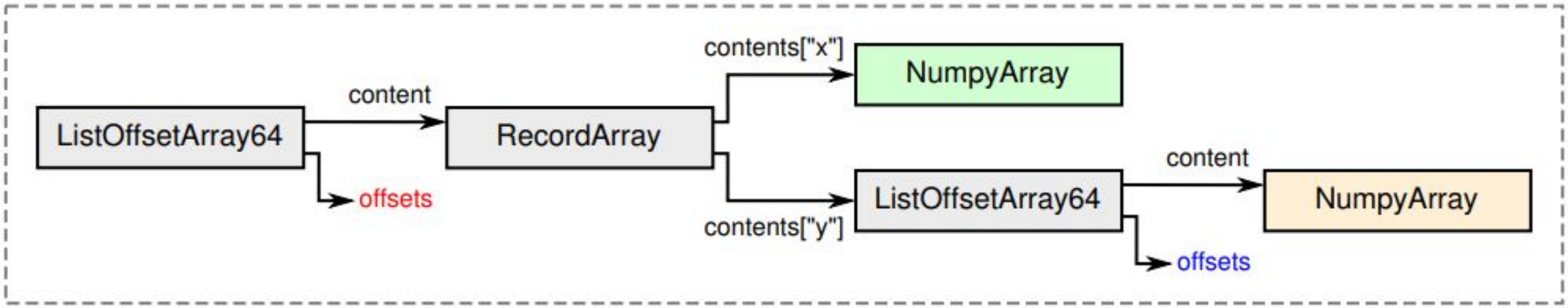
The **leaf nodes here, Index Class and NumpyArray Class** are the only linear buffers, we take care of.

**This turns the transfer to GPU problem, into a simple recursive walk down the complex Data Structure where the `base` case is transferring the leaf nodes, onto the GPU!**

# How do we know where the Array Buffers exist?

- We keep track of the leaf nodes of Awkward Arrays by giving them **an enum class type** which signifies which kernel, should that Array use when we are doing operations on them.
- This enum can later be expanded to include other kernel library like opencl and so on.

```
enum lib {
    cpu,
    cuda
}
```



# Awkward Arrays and CUDA Dependency

**`pip install awkward1[cuda]`**

- That's it. Awkward Arrays has no direct dependency on CUDA. The **`awkward1-cuda-kernels`** are just an extension to Awkward Arrays.

- The pip package consists of:
  - **`__init__.py`**
  - **`libawkward-cuda-kernels.so`**

- The **`shared library`**, helps the **`awkward1-cuda-kernels pip package`** `to` be accessible across all Linux systems and makes the package itself extremely portable.

# What about the CUDA dependency?

How is Awkward Array able to access the shared library?

- **`dlopen`** - To open the library
- **`dlsym`** - To access all the symbols / functions in it

One potential disadvantage of having such system calls!

- The function calls are largely similar across all kernels, it would be very difficult to write and maintain more than 100 such calls for the 100+ kernels!

Let the **`preprocessor`** do the work for us! We define a **`Macro`** to automate the process of writing the system calls!

```
#define CREATE_KERNEL(libFnName, ptr_lib)    \
  auto handle = acquire_handle(ptr_lib);     \
  typedef decltype(libFnName) functor_type; \
  auto* libFnName##_fcn =                    \
      reinterpret_cast<functor_type*>(acquire_symbol(handle, #libFnName));
```

# Finally, we can introduce the Indirection!

- We can finally distinguish between Arrays on main memory and arrays on GPU!
- The next step would be to introduce a dispatch mechanism that actually calls the right library according to where the buffer resides!
- Here's an generalized example of how every function in the kernel-dispatch file looks like!

```
Error Struct <Kernel Name>(
  kernel::lib ptr_lib,
  <more arguments>) {

  if (ptr_lib == kernel::lib::cpu) {
     return awkward_<Kernel Name>(<more arguments>);
  }

  else if (ptr_lib == kernel::lib::cuda) {

     CREATE_KERNEL(awkward_<Kernel Name>, ptr_lib);
     return (*awkward_<Kernel Name>_fcn)(<more arguments>);

  }
}
```

# Time for some examples!

- Let’s consider a Record Array!

```
array = ak.Array([
            [{"x": 1, "y": [11]},
            {"x": 4, "y": [12, 22]},
            {"x": 9, "y": [13, 23, 33]}],
            [],
            [{"x": 16, "y": [14, 24, 34, 44]}]], kernels = "cuda")
```

- We can now perform non-trivial things with this array!

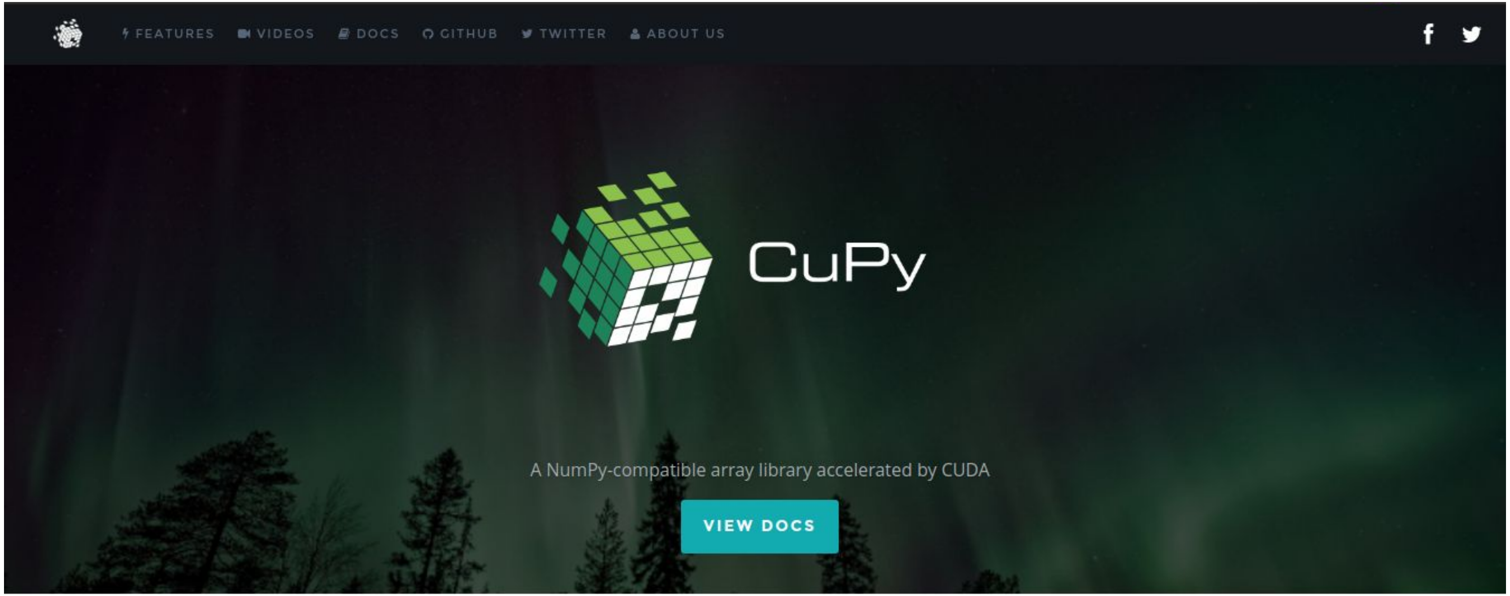
```
Let’s do a ak.num(array), by default the axis is 1, so you’ll get:

<Array:cuda [3, 0, 1] type='3 * int64'>


What if we want to find the number of elements in the list corresponding to a list,
ak.num(array[“y”], axis = 2), should give us:

<Array:cuda [[1, 2, 3], [], [4]] type='3 * var * int64'>
```

# CuPy Integration



# CuPy Integration

- Awkward Arrays already had a strong integration with NumPy, now it can support CuPy operations too!

From CuPy To Awkward Array

```
ak.Array(cp.array([[1, 2], [3, 4],[5, 6]]))
<Array:cuda [[1, 2], [3, 4], [5, 6]] type='3 * 2 * int64'>
```

From Awkward Array to Cupy

```
array = ak.Array([[1, 2], [3, 4],[5, 6]], kernels=”cuda”)
cp.asarray(array)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

# Concluding my Summer of Code!

- Nearly met all the deliverables
  - Track “memory location” through Awkward Array classes(#262, #276)
  - Operations involving a CPU array and a GPU array should be handled intelligently(#293, #299)
  - Develop a deployment strategy for users with GPUs and users without GPUs(#345, #357)
  - Integrate CuPy with Awkward Arrays(#362, #372)

# THANK YOU!

trickarcher

anishbiswas271@gmail.com