# Language Transformations for the Awkward Array Library

Pratyush Das
(Institute of Engineering and Management, Kolkata)

Jim Pivarski
(Princeton University)

Anish Biswas
(Manipal Institute of Technology)

# You *might* already know me

2017 – Worked with Jim Pivarski on root4j and spark-root

- Spark-ROOT: Viktor Khristenko - CERN OpenLab, Pratyush Das - Institute of Engineering and Management

2018 – DIANA-HEP Fellow, worked with Jim Pivarski on uproot

Current and past DIANA fellows

Pratyush Das, Institute of Engineering and Management (Kolkata) [Undergrad]

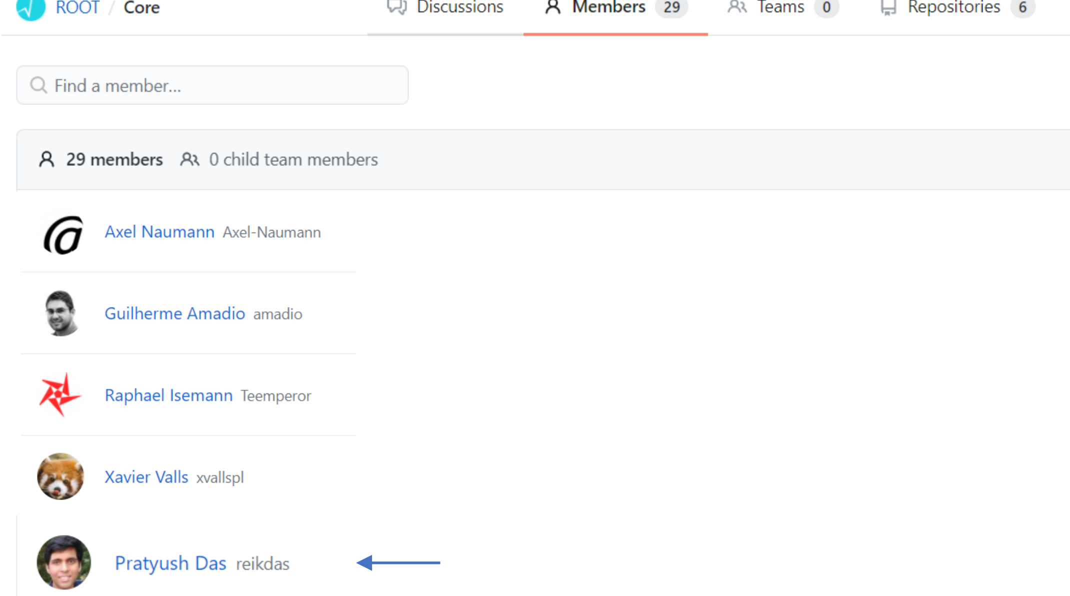
- Topic: Add write functionality to uproot - proposal
- Mentor: Jim Pivarski, Princeton University
- Dates/Location: *summer, 2018 (FNAL)*

2019 – IRIS-HEP Fellow, worked with Jim Pivarski on uproot
2020 – IRIS-HEP Fellow, worked with Jim Pivarski on Awkward Array

**IRIS-HEP Fellow: Pratyush (Reik) Das**

**Fellowship dates:** Jun-Sep 2019, Jun-Aug 2020
**Home Institution:** Institute of Engineering & Management (Kolkata)

Also, (infrequent) contributor to ROOT -



# The Awkward Array library

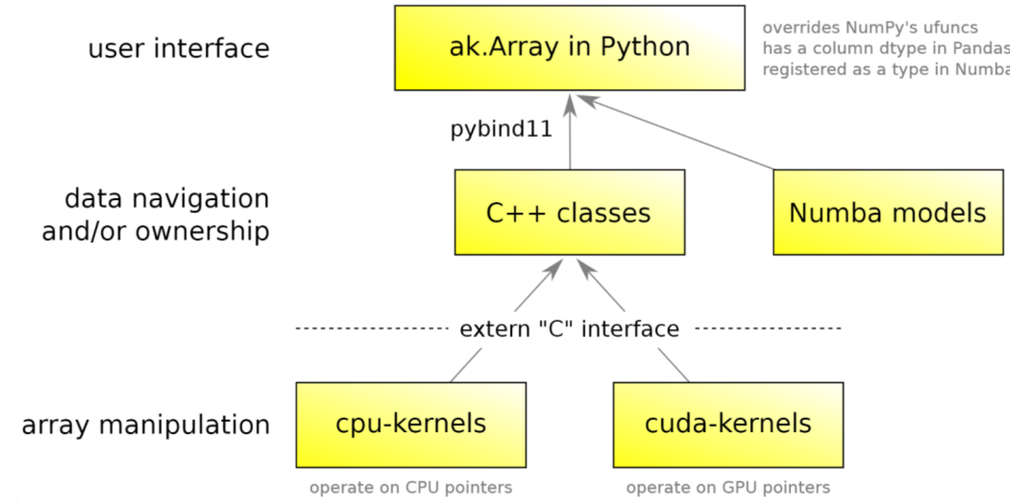
NSF 1836650 | DOI 10.5281/zenodo.3952674 | python 2.7 3.5 3.6 3.7 3.8 | License BSD 3-Clause

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.

# CUDA backend

- Particularly suited for a GPU backend due to its array-at-a-time approach that is common in the design of parallel algorithms which run on GPUs.
- Faster operations in a lot of cases.
- Why CUDA? – First step to full GPU support and CUDA is the most widely supported right now.

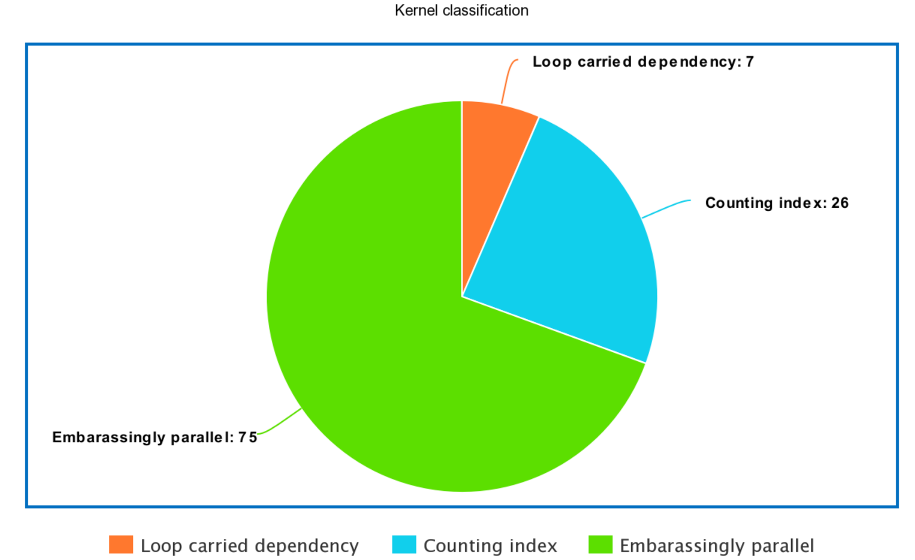



## Auto generating the CUDA kernels

CPU kernels follow a regular code structure

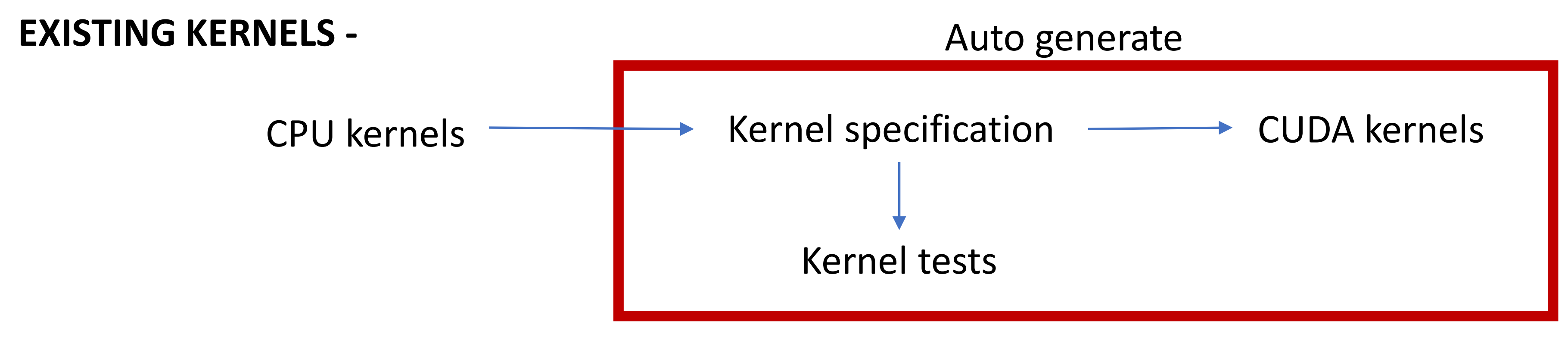
- Subset of C++ -> C with template types.
- Similar pattern repeated across all functions.
- Minimal use of header files.

Most of the CPU kernels are embarrassingly parallel – makes it easier to compile to parallelized CUDA.



# Workflow

**EXISTING KERNELS -**




---

**SOON -**

Tune specification → Check generated specification and CUDA kernels into git

---

**NEW KERNELS IN THE FUTURE -**

Kernel specification → Kernel tests

Kernel specification → Header generator (?)

Kernel specification → Kernel dispatch generator (?)

Maintained by hand:

- CPU kernels
- CUDA kernels

Helper tools:

- Specification generator
- CUDA kernel generator

# Kernel specification

```yaml
- name: awkward_new_Identities
  specializations:
    - name: awkward_new_Identities32
      args:
        - toptr: List[int32_t]
        - length: int64_t
    - name: awkward_new_Identities64
      args:
        - toptr: List[int64_t]
        - length: int64_t
  inparams: ['length']
  outparams: ['toptr']
  definition: |
    def awkward_new_Identities(toptr, length):
        for i in range(length):
            toptr[i] = i
  tests:
    - args:
        toptr: [123, 123, 123]
        length: 3
      successful: True
      results:
        toptr: [0, 1, 2]
```

All kernels are represented as a specification (YAML file).

Wait, is that Python in there?

How did we go from C++ code for the CPU kernels to a YAML specification?

# C++ ---> C

- CPU kernels are written in code that resembles C with template types (C++11).
- Usually have specializations for arguments with different types.
- Easier to parse C, than C++.

> **Parsing C++** is known to be **difficult** due to the complexity of the language and the numerous ambiguities and LALR(1) con- flicts in its official grammar.
>
> core.ac.uk › download › pdf ▾ PDF
>
> Parsing C/C++ code without preprocessing - Core

- Use regex matching to replace C++ headers, store template information and remove templates -> making the code C99 compliant.

*(Plug stored template info into specification)*

```
template <typename T>
ERROR awkward_new_Identities(
  T* toptr,
  int64_t length) {
  for (T i = 0;  i < length;  i++) {
    toptr[i] = i;
  }
  return success();
}
```

```
ERROR awkward_new_Identities32(
  int32_t* toptr,
  int64_t length) {
  return awkward_new_Identities<int32_t>(
    toptr,
    length);
}
```

# C ---> Python

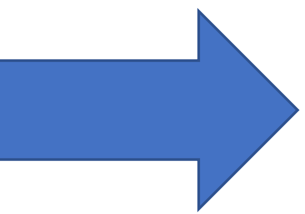
- Create C AST in Python using pycparser.

### 1.1 What is pycparser?

**pycparser** is a parser for the C language, written in pure Python. It is a module designed to be easily integrated into applications that need to parse C source code.

- Generate Python from C AST

```
template <typename T>
ERROR awkward_new_Identities(
  T* toptr,
  int64_t length) {
  for (T i = 0;  i < length;  i++) {
    toptr[i] = i;
  }
  return success();
}
```



```
def awkward_new_Identities(toptr, length):
    for i in range(length):
        toptr[i] = i
```

- Store types in our own representation

```
args:
  - toptr: List[int32_t]
  - length: int64_t
```

# Test generator

- The specification also contains test cases for the kernel.
- Annotate the parameters of the kernels with the Array node property it denotes.

```
/// @param toptr outparam
/// @param length inparam
EXPORT_SYMBOL struct Error
  awkward_new_Identities32(
    int32_t* toptr,
    int64_t length);
```

← Doxygen style annotation, so it is picked up in Doxygen C++ documentation.

- Values of attributes picked from pre-existing list of values.


```
"Identities": [
    {
        "Identities-array": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
        "Identities-array-offset": 0
    },
    {
        "Identities-array": [0, 2, 1, 1, 1, 1, 2, 0, 1, 2, 0, 1, 0, 2, 0, 1, 1, 1, 0, 2, 2, 2, 1, 2, 0, 1],
        "Identities-array-offset": 1
    }
],
```


- Perform some combinatorics on these lists to generate multiple tests.


```
tests:
  - args:
      toptr: [123, 123, 123]
      length: 3
    successful: True
    results:
      toptr: [0, 1, 2]
```

# Kernel specification

```yaml
- name: awkward_new_Identities
  specializations:
    - name: awkward_new_Identities32
      args:
        - toptr: List[int32_t]
        - length: int64_t
    - name: awkward_new_Identities64
      args:
        - toptr: List[int64_t]
        - length: int64_t
  inparams: ['length']
  outparams: ['toptr']
  definition: |
    def awkward_new_Identities(toptr, length):
        for i in range(length):
            toptr[i] = i
  tests:
    - args:
        toptr: [123, 123, 123]
        length: 3
      successful: True
      results:
        toptr: [0, 1, 2]
```

Here is another look at the kernel specification from earlier.

We use the specification to –

1. Provide a “correct” definition for the kernels.
2. Generate CUDA (for now).
3. Test that the CUDA and CPU kernels are correct.
    - Test the specification against itself.
    - Check if CPU kernels match specification.
    - Check if CUDA kernels match specification.

# Kernel documentation

We also have documentation of the kernels at - https://awkward-array.readthedocs.io/en/latest/_auto/kernels.html

The documentation is generated from the specification.

## awkward_new_Identities

`awkward_new_Identities64`*(toptr: List[int64_t], length: int64_t)*

`awkward_new_Identities32`*(toptr: List[int32_t], length: int64_t)*

```python
def awkward_new_Identities(toptr, length):
    for i in range(length):
        toptr[i] = i
```

# Generating CUDA kernels

- We use the Python definition and the type information from the specification to generate parallel CUDA kernels.
- We use Python's ast module to parse the Python definition and create an AST, from which we generate the CUDA.

```
template <typename T>
ERROR awkward_new_Identities(
  T* toptr,
  int64_t length) {
  for (T i = 0;  i < length;  i++) {
    toptr[i] = i;
  }
  return success();
}
```

```
template <typename A>
__global__ void
cuda_new_Identities(A* toptr, int64_t length) {
  int64_t block_id =
      blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
  int64_t thread_id = block_id * blockDim.x + threadIdx.x;
  if (thread_id < length) {
    toptr[thread_id] = thread_id;
  }
}
```

eg. A trivial embarrassingly parallel kernel

- Important: The generate CUDA kernels are parallel and do not execute sequentially on a single GPU thread.
- We can currently generate "accurate" CUDA kernels for 36 out of around 100 kernels.

# End of the summer

- I still have a month left! (Fellowship extended)
- Gave a talk to Princeton University's Liberty Research Group about my work.

THE LIBERTY RESEARCH GROUP

The Liberty Computer Architecture Research Group exploits unique opportunities exposed by considering the interaction of compilers and architectures to increase performance, to improve reliability, to reduce cost, to lower power, and to shorten the time to market of microprocessor systems. This objective is accomplished by providing critical computer architecture and compiler research, expertise, and prototypes to the community.

- Met the deliverables –
  1. Create a kernel test generation framework.
  2. Create a specification generator.
  3. Create a specification -> documentation generator.
  4. Create a CUDA kernel generator.

- What is left –
  1. Complete CUDA kernel test generator.
  2. Tune the generated specification.
  3. Manually write remaining CUDA kernels.
  4. Generate header files from specification (?)
  5. Generate kernel dispatch (?)

- Merged 26 Pull Requests to master.

# THANK YOU

(Anish will talk more about how the generated CUDA kernels are used, in the next talk)

@ReikDas

reikdas

reikdas@gmail.com