PHY-2323298

Advancing Awkward Arrays for High-Performance CPU and GPU Processing

Ianna Osborne, Princeton University, Manasvi Goyal, Harvard University

This work presents ongoing efforts to optimize Awkward Arrays for GPUs using CUDA, aiming to achieve performance parity with or surpass CPU kernel implementations. Key improvements focus on optimized memory management, leveraging CUDA-specific features, and maximizing parallelism to enhance throughput. These advancements enable faster and more scalable data processing, particularly for HL-LHC data analysis within the Python ecosystem. We will discuss the challenges, solutions, and performance benchmarks.

Lossless, bidirectional, zero-copy conversions between Awkward Array, Apache Arrow, Feather, and Parquet.

Manipulating ragged arrays as though they were **NumPy** or **CuPy** arrays, following the Array API specification.

dask-awkward is a high-level collection type for Dask, alongside Array and DataFrame.

JIT-compiled as a C++ iterator in ROOT's RDataFrame workflow.

Bidirectional conversion between Awkward Array and TensorFlow RaggedTensor.

Integrated into AnnData for single-cell genomics.

Auto-diffirentiate through expressions involving Awkward Arrays, using JAX

Awkward Arrays on GPUs, backed by CuPy.

Efficiently iterate over or build Awkward Arrays in code JIT-compiled with Numba, on CPUs and GPUs

Read any Kaitai Struct format into Awkward Arrays.

Store Awkward Arrays in the Tiled database and slice them in the cloud.

JIT-compile Awkward Arrays in C++ with cppyy. Header-only library to build Awkward Arrays in C++ and transfer them to Python.

Akimbo: work with Awkward Arrays as DataFrame columns

Exchange data with the Julia language through a reimplementation of the Awkward Array memory layouts.

PHY-2323298

Awkward Array

Example of an "awkward" array

```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

Record structures with differently typed fields

Array of variable-length lists ("ragged" or "jagged" arrays)

Nested variable-length lists

Missing data

Heterogenous data (union/variant types)

NumPy-like expression

```
output = np.square(array["y", ..., 1:])
```

Result

```
output.to_list()
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

Equivalent Python

```
output = []
for sublist in python_objects:
    tmp1 = []
    for record in sublist:
        tmp2 = []
        for number in record["y"][1:]:
            tmp2.append(np.square(number))
        tmp1.append(tmp2)
    output.append(tmp1)
```

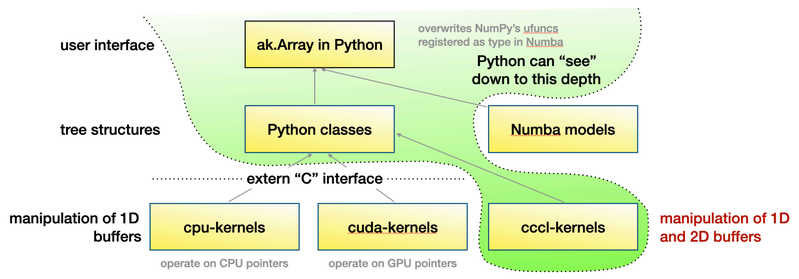
(for a similar calculation 10 million times larger, single threaded)

1.5 seconds
2.1 GB of memory

140 seconds
22 GB of memory

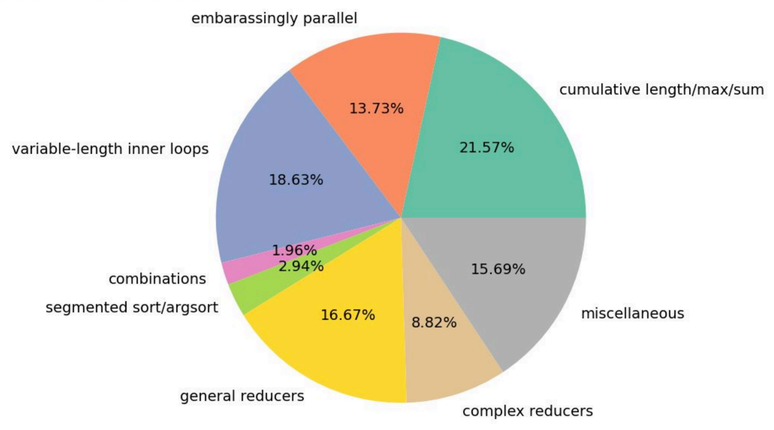
Awkward Array v2 architecture

- No direct dependency on CUDA
- Introduces an indirection as we move from the upper to the lower layers.
- Awkward functions can be used on GPU with just pip install awkward



Awkward CPU backend

Overall 144 CPU kernels:



CuPy is used to handle the higher level functions

Awkward CCCL kernels

```
import cuda.cccl.parallel.experimental as parallel

def f(grid, block, args):
    """
    Min reduction for sorted, present parents on device:
    (toptr, fromptr, offsets, outlength, identity, invocation_index, err_code)
    """
    (
        toptr,
        fromptr,
        offsets,
        outlength,
        identity,
        invocation_index,
        err_code,
    ) = args

    min_op = (
        min_op_complex
        if np.issubdtype(fromptr.dtype, np.complexfloating)
        else min_op_real
    )

    toptr[:outlength] = identity

    # Keep host copy of identity for segmented_reduce
    identity_host = np.asarray(identity, dtype=fromptr.dtype)

    # Segmented reduction (identity must be host scalar/array)
    parallel.segmented_reduce(
        fromptr, toptr, offsets[:-1], offsets[1:], min_op, identity_host, outlength
    )
```

```
def min_op_real(a, b):
    return a if a < b else b


def min_op_complex(a, b):
    if abs(a) < abs(b):
        return a
    elif abs(b) < abs(a):
        return b
    else:
        if a.real != b.real:
            return a if a.real < b.real else b
        else:
            return a if a.imag < b.imag else b
```

Awkward functions: CPU vs CUDA backend

```
array_cpu = ak.Array([[],[6.3]],backend="cpu")
ak.min(array_cpu, axis=1)

jets = ak.Array([[],[1.1,2.1,3.1]],backend="cpu")
trijet = ak.combinations(jets, 3, fields=["j1", "j2", "j3"])
result = trijet.j1 + trijet.j2 + trijet.j3
ak.min(result, axis=1)
```

```
array_gpu = ak.Array([[],[6.3]],backend="cuda")
ak.min(array_gpu, axis=1)

jets = ak.Array([[],[1.1,2.1,3.1]],backend="cuda")
trijet = ak.combinations(jets, 3, fields=["j1", "j2", "j3"])
result = trijet.j1 + trijet.j2 + trijet.j3
ak.min(result, axis=1)
```

Automated infrastructure connecting CUDA kernels to Awkward operations

Testing of kernels via both generated tests:
- Python test generation scripts produce custom unit tests in Python
  - they are generated from a JSON file of test cases.

and:
- integration tests for each Awkward function on the CUDA backend

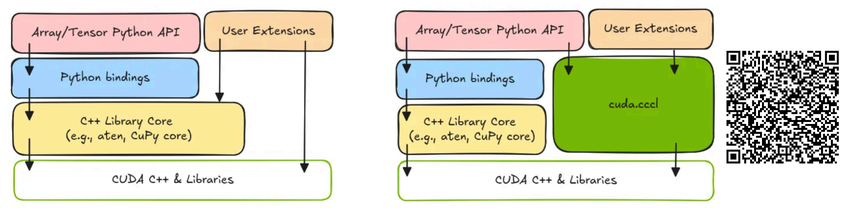
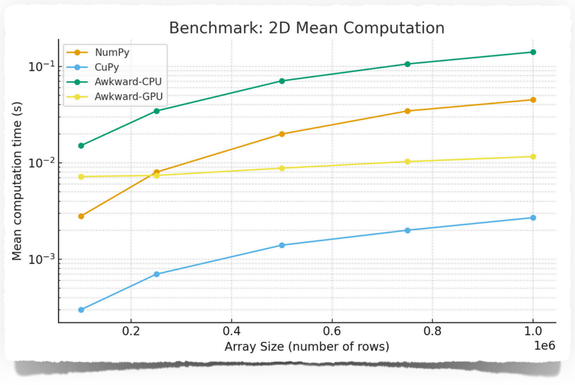


Figure 1. Architecture of CUDA-enabled Python packages like PyTorch and CuPy today (left), and the gap filled by cuda.cccl (right)

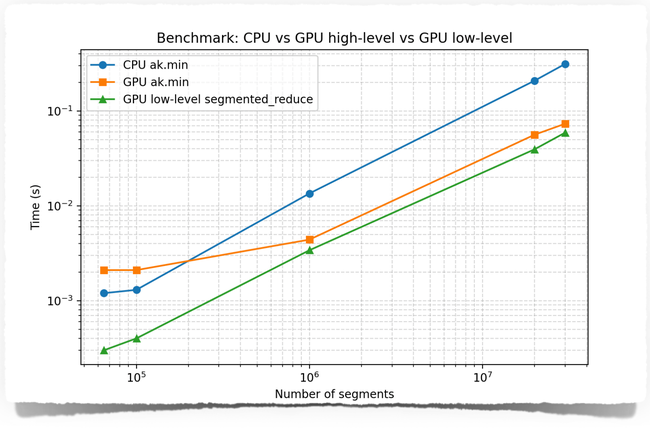
Optimizing Awkward Arrays for GPUs using the CUDA Core Compute Libraries in Python, improving memory use and parallelism to efficiently handle large, irregular data.

Performance benchmarks

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.65.06              Driver Version: 580.65.06      CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-PCIE-40GB          On  |   00000000:21:00.0 Off |                    0 |
| N/A   34C    P0             36W /  250W |       6MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
```



The logarithmic y-axis makes it clear that **CuPy and Awkward-GPU** are significantly faster than **NumPy and Awkward-CPU**, especially as array size grows.



CPU times increase sharply with segment count.
GPU high-level ak.min is slightly faster at mid-range, but slower than low-level for large arrays.
GPU low-level segmented reduction is the fastest overall, especially for tens of millions of segments.