Evaluation task for Awkward Array GSoC project

Pratyush Das[1]

[1]Institute of Engineering & Management, Kolkata

Abstract

In this report, we demonstrate a sample GPU kernel designed to be an alternative to the CPU backend for array operations in the Awkward Array project. In particular, we implement a CUDA translation of the *awkward_listarray_compact_offsets* CPU kernel, and show how parallelizing the code on a GPU could significantly increase the speed of computation. The motivation of this report is to serve as an evaluation task to qualify to work on the project titled "Awkward Array GPU kernels" under the mentorship of Jim Pivarski and David Lange.

1 Introduction

The sample CPU kernel (directly taken from the Awkward Array codebase) to be translated is defined below -

```
template <typename C, typename T>
ERROR awkward_listarray_compact_offsets(T* tooffsets, const C* fromstarts, const C
    * fromstops, int64_t startsoffset, int64_t stopsoffset, int64_t length) {
  tooffsets[0] = 0;
  for (int64_t i = 0;  i < length;  i++) {
    C start = fromstarts[startsoffset + i];
    C stop = fromstops[stopsoffset + i];
    if (stop < start) {
      return failure("stops[i] < starts[i]", i, kSliceNone);
    }
    tooffsets[i + 1] = tooffsets[i] + (stop - start);
  }
  return success();
}
```

Translating this particular CPU kernel serves as a good test because parallelizing this CPU kernel involves overcoming the loop carried dependency in the above code -

```
tooffsets[i + 1] = tooffsets[i] + (stop - start);
```

where `tooffsets[i + 1]` depends on `toffsets[i]`.

The above algorithm can be visualized to illustrate its linear nature -

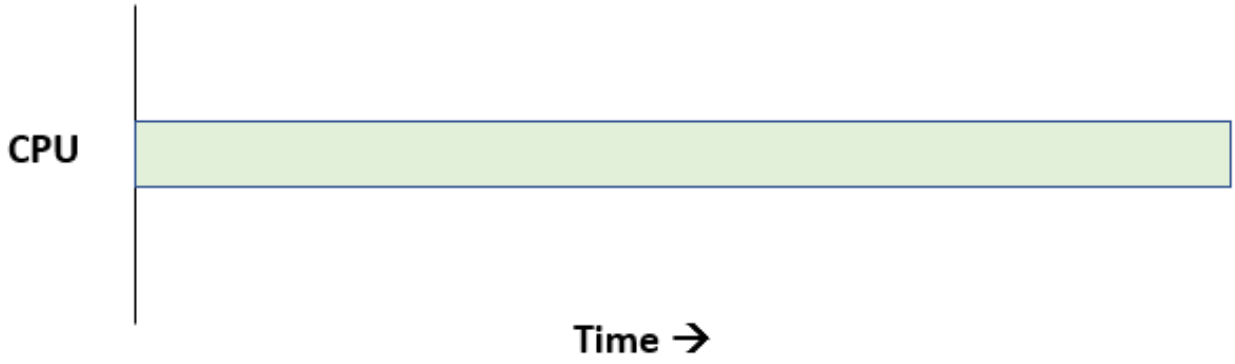

Figure 1: Visualizing sequential CPU algorithm

1.1 Implementation of GPU algorithm

The evaluation task required creating a parallel algorithm as an alternative for the sequential CPU algorithm. There are several ways to do this -

1. OpenCL, which is a more uniform interface to write parallel algorithms to be executed on multiple backends, but is slower than CUDA and losing popularity.
2. CUDA, which is an easier and more performant interface for executing parallel algorithms only on an NVIDIA GPU.

Due to the availability of an NVIDIA GPU and performance requirements, we have decided to use CUDA to write and execute our parallel algorithms.

We use some CUDA intrinsics like `thrust` in our sample code for the evaluation task which we might not use in the production code due to avoid locking the implementation to CUDA and allow expansion to newer interfaces in the future.

2 Sequential algorithm on an Nvidia GPU

2.1 Sequential algorithm on a single thread

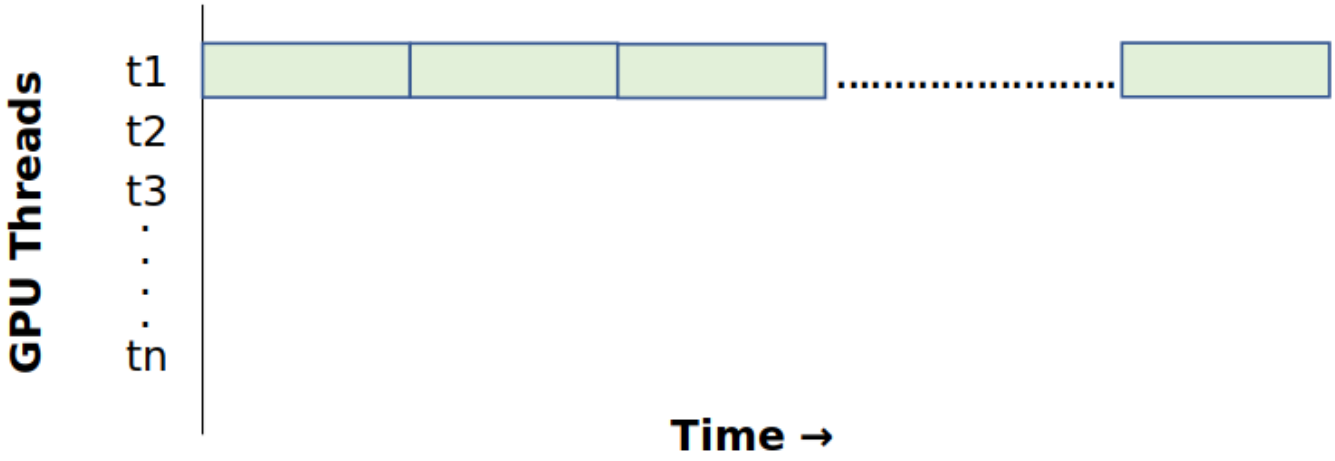

Figure 2: Visualizing GPU threads at work

```
__device__
void success_cuda(Error* err) {
  err.str = nullptr;
  err.identity = kSliceNone;
  err.attempt = kSliceNone;
  err.extra = 0;
}

__device__
void failure_cuda(const char* str, int64_t identity, int64_t attempt, Error* err)
    {
  err.str = str;
  err.identity = identity;
  err.attempt = attempt;
  err.extra = 0;
}

template <typename C, typename T>
__global__
void awkward_listarray_compact_offsets(T* tooffsets, const C* fromstarts, const C*
    fromstops, int64_t startsoffset, int64_t stopsoffset, int64_t length, Error*
```

```
    err) {
  __shared__ int flag[1];
  int idx = threadIdx.x + (blockIdx.x * blockDim.x);
  if (idx == 0) {
    tooffsets[0] = 0;
    flag[0] = 0;
  }
  if (idx < length) {
    if (idx == 0) {
      for (int i = 0; i < length; i++) {
        C start = fromstarts[startsoffset + i];
        C stop = fromstops[stopsoffset + i];
        if (stop < start) {
          failure_cuda("stops[i] < starts[i]", i, kSliceNone, err);
          flag[0] = 1;
        }
        tooffsets[i + 1] = tooffsets[i] + (stop - start);
      }
    }
  }
  __syncthreads();
  if (flag[0] != 1) {
    success_cuda(err);
  }
}
```

In the above code block[1] we execute the algorithm only for a single GPU thread while the other threads sit idle.

```
    if (idx == 0) {
        ...
    }
```

where the variable `idx` is the thread index.

Since CUDA GPU kernels have to be of the return type void, we use additional device functions, `success_cuda` and `failure_cuda` to record whether the kernel behaved as expected.

```
__device__
void success_cuda(Error* err) {
  err.str = nullptr;
  err.identity = kSliceNone;
  err.attempt = kSliceNone;
```

[1]Full benchmarking code: `https://github.com/reikdas/GSoC-Proposal-2020/blob/master/testgsoc/naivesingle.cu`

```
  err.extra = 0;
}

__device__
void failure_cuda(const char* str, int64_t identity,
                  int64_t attempt, Error* err) {
  err.str = str;
  err.identity = identity;
  err.attempt = attempt;
  err.extra = 0;
}
```

We use a flag implemented using shared memory and the CUDA intrinsic thread synchronization

```
__shared__ int flag[1];
...
__syncthreads();
if (flag[0] != 1) {
    ...
}
```

to determine whether the `success_cuda()` call should be executed once the `failure_cuda()` call has already been reached by one of the GPU threads.

2.2 Sequential algorithm on multiple threads

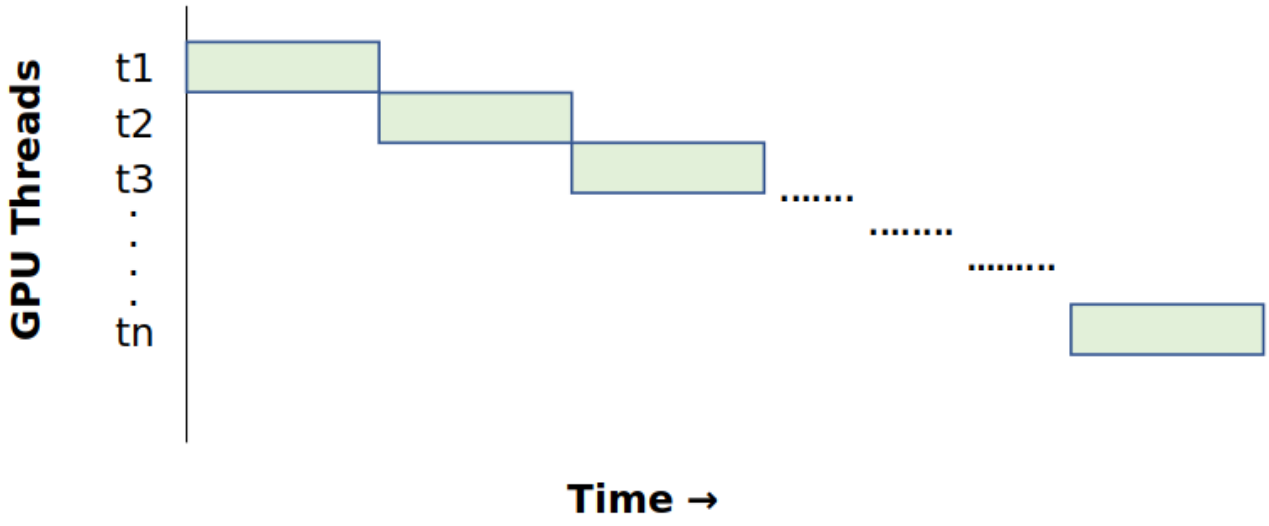

Figure 3: Visualizing GPU threads at work

```
__device__
void success_cuda(Error* err) {
  err.str = nullptr;
  err.identity = kSliceNone;
  err.attempt = kSliceNone;
  err.extra = 0;
}

__device__
void failure_cuda(const char* str, int64_t identity, int64_t attempt, Error* err)
    {
  err.str = str;
  err.identity = identity;
  err.attempt = attempt;
  err.extra = 0;
}

template <typename C, typename T>
__global__
```

```
void awkward_listarray_compact_offsets(T* tooffsets, const C* fromstarts, const C*
    fromstops, int64_t startsoffset, int64_t stopsoffset, int64_t length, Error*
    err) {
  __shared__ int flag[1];
  int idx = threadIdx.x + (blockIdx.x * blockDim.x);
  if (idx < length) {
    if (idx == 0) {
      tooffsets[0] = 0;
      flag[0] = 0;
    }
    for (int i = 0; i < length; i++) {
      __syncthreads();
      if (i == idx) {
        C start = fromstarts[startsoffset + i];
        C stop = fromstops[stopsoffset + i];
        if (stop < start) {
          failure_cuda("stops[i] < starts[i]", i, kSliceNone, err);
          flag[0] = 1;
        }
        tooffsets[i + 1] = tooffsets[i] + (stop - start);
      }
    }
  }
  __syncthreads();
  if (flag[0] != 1) {
    success_cuda(err);
  }
}
```

The sequential algorithm on multiple GPU threads implementation above[2] differs only slightly from the sequential algorithm on single GPU threads implementation.

The main difference is that the multiple GPU threads implementation has all the threads execute the algorithm one after another, instead of all at the same time. When one thread is being executed, the other threads sit idly, then the next thread is executed while the previous and all other threads sit idle.

```
    _for (int i = 0; i < length; i++) {
      ...
      if (i == idx) {
        ...
        }
    }
```

We use the CUDA intrinsic `__syncthreads()` before each iteration of the loop to ensure that all the threads are at the same position and ready to be executed before each thread is executed.

[2]Full benchmarking code: `https://github.com/reikdas/GSoC-Proposal-2020/blob/master/testgsoc/naivemultiblocks.cu`

3 Parallel algorithm on an Nvidia GPU

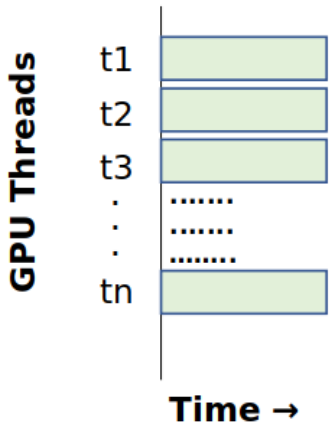

Figure 4: Visualizing GPU threads at work

```
#include <thrust/device_vector.h>
#include <thrust/scan.h>
#include "helper_cuda.h"

__device__
void success_cuda(Error* err) {
  err.str = nullptr;
  err.identity = kSliceNone;
  err.attempt = kSliceNone;
  err.extra = 0;
}

__device__
void failure_cuda(const char* str, int64_t identity, int64_t attempt, Error* err)
    {
  err.str = str;
  err.identity = identity;
  err.attempt = attempt;
  err.extra = 0;
}

template <typename T, typename C>
__global__
```

```
void sub(T* output, const C* starter, const C* stopper, int64_t startsoffset,
    int64_t stopsoffset, int64_t n, Error* err) {
  __shared__ int flag[1];
  int thid = threadIdx.x + blockIdx.x * blockDim.x;
  if (thid == 0) flag[0] = 0;
  if (thid < n) {
    C start = starter[thid + startsoffset];
    C stop = stopper[thid + stopsoffset];
    if (stop < start) {
      failure_cuda("stops[i] < starts[i]", thid , kSliceNone, err);
      flag[0] = 1;
    }
    output[thid] = stop - start;
  }
  __syncthreads();
  if (flag[0] != 1) {
    success_cuda(err);
  }
}

template <typename T, typename C>
void prefix_sum(T* output, const C* arr, const C* arr2, int64_t startsoffset,
    int64_t stopsoffset, int64_t length, Error* err) {
  int block, thread;
  if (length > 1024) {
    block = (length / 1024) + 1;
    thread = 1024;
  }
  else {
    thread = length;
    block = 1;
  }
  T* d_output;
  C* d_arr, * d_arr2;
  checkCudaErrors(cudaMalloc((void**)&d_output, length * sizeof(T)));
  checkCudaErrors(cudaMalloc((void**)&d_arr, length * sizeof(C)));
  checkCudaErrors(cudaMemcpy(d_arr, arr, length * sizeof(C),
      cudaMemcpyHostToDevice));
  checkCudaErrors(cudaMalloc((void**)&d_arr2, length * sizeof(C)));
  checkCudaErrors(cudaMemcpy(d_arr2, arr2, length * sizeof(C),
      cudaMemcpyHostToDevice));
  sub<T, C><<<block, thread>>>(d_output, d_arr, d_arr2, startsoffset, stopsoffset,
      length, err);
  checkCudaErrors(cudaDeviceSynchronize());
  thrust::device_vector<T> data(d_output, d_output+length);
```

```
    thrust::device_vector<T> temp(data.size() + 1);
    thrust::exclusive_scan(data.begin(), data.end(), temp.begin());
    temp[data.size()] = data.back() + temp[data.size() - 1];
    thrust::copy(temp.begin(), temp.end(), output);
    checkCudaErrors(cudaFree(d_output));
    checkCudaErrors(cudaFree(d_arr));
    checkCudaErrors(cudaFree(d_arr2));
  }
```

The above code[3]

4 Benchmarks

We have compared the performance of the various algorithms relative to each other[4].

[3] Full benchmarking code: `https://github.com/reikdas/GSoC-Proposal-2020/blob/master/testgsoc/thrust.cu`

[4] Benchmarking numbers: `https://github.com/reikdas/GSoC-Proposal-2020/blob/master/testgsoc/data.txt` generated using `https://github.com/reikdas/GSoC-Proposal-2020/blob/master/testgsoc/plotter.py`

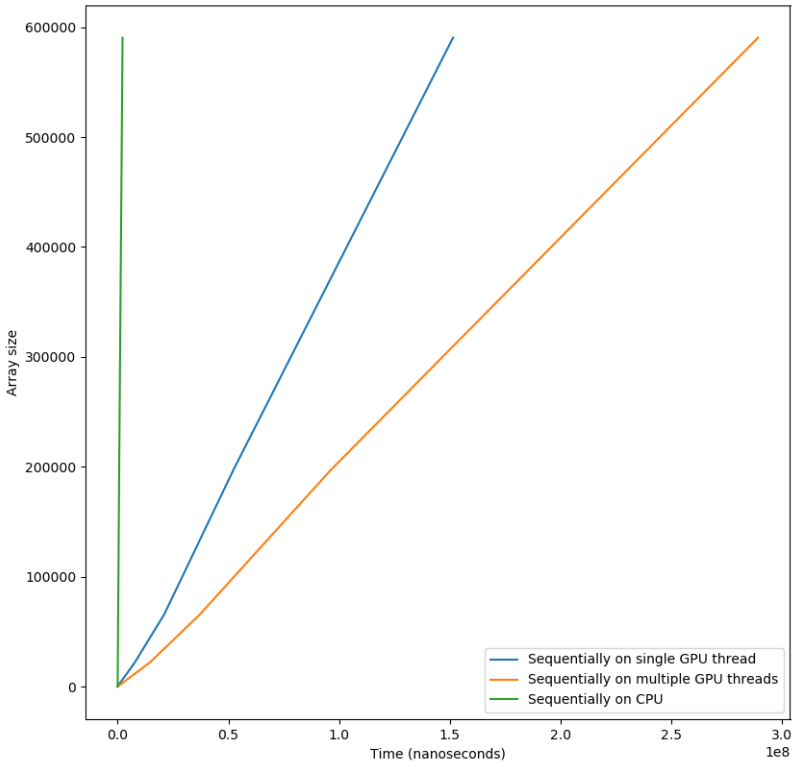

Figure 5: Comparing performance of sequential GPU algorithms with CPU algorithm

We can see that the sequential algorithms on the GPU run much slower than the sequential algorithm on the CPU. This can be attributed to the inherent difference in the design of GPUs and CPUs. GPUs are designed to operate on independent data in parallel, whereas CPUs are optimized to execute a single stream of instructions as quickly as possible. So when we are executing a single instruction stream on the GPU, it is expected that the time for completion will be more than the time for completion of the same instructions on a CPU.

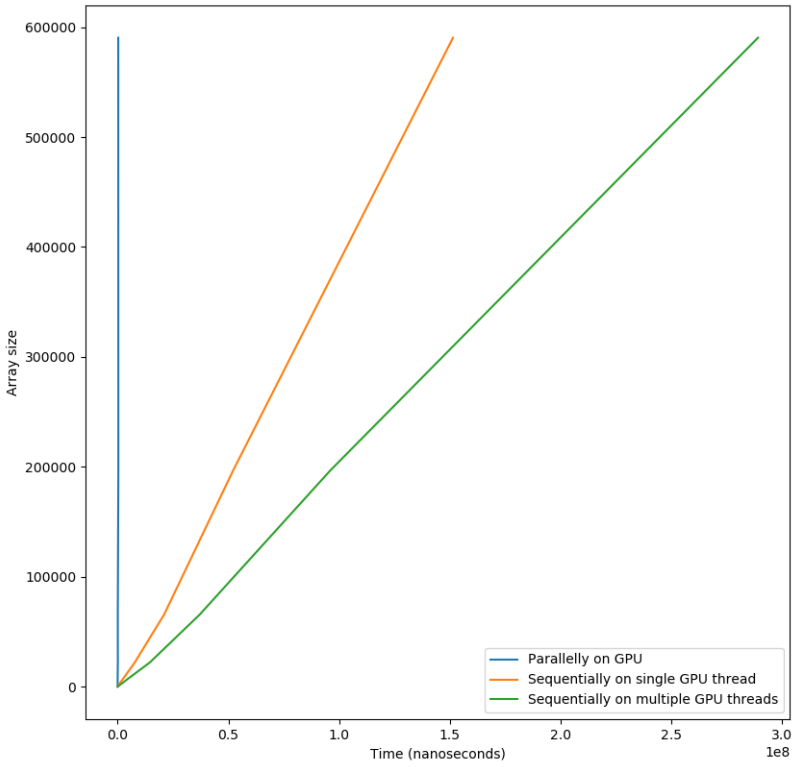

Figure 6: Comparing performance of sequential and parallel GPU algorithms

Similar to the previous plot, we can see that our implementation of the given CPU algorithm to run parallely on the GPU is much faster than the sequential algorithms on the GPU. This gives us an inclination that our implementation is correct and gives a speedup over the sequential algorithm when executed on the same hardware (GPU in our test machine in this case)[5].

[5]We used an Nvidia GeForce 1070 Max Q for our tests

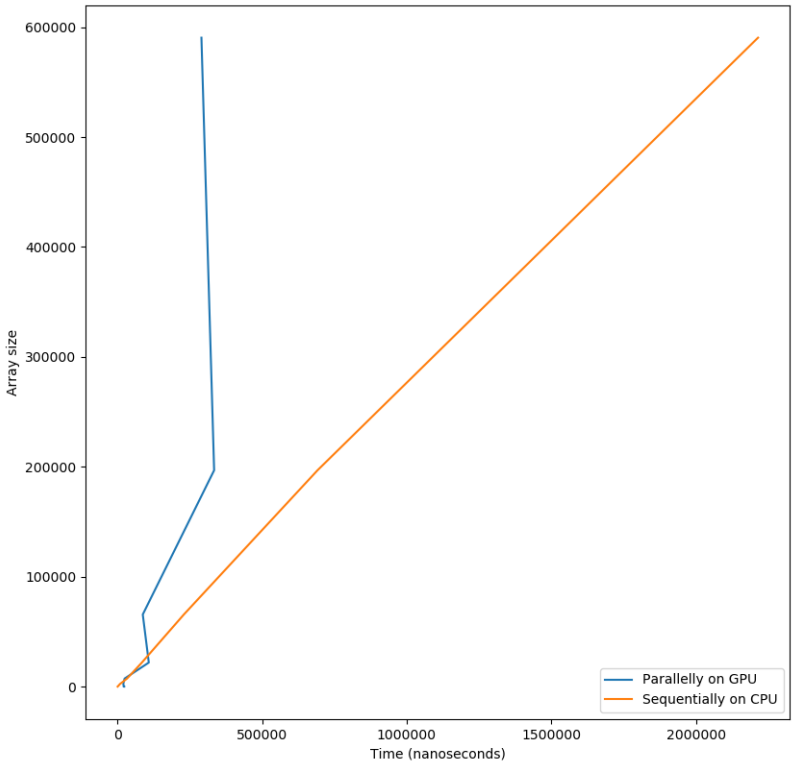

Figure 7: Comparing parallelized GPU performance vs CPU performance

This is the most important result of the evaluation task. Here, we can clearly visualize that the parallel algorithm we designed to be executed on the GPU is much faster than the given sequential algorithm on the CPU.

5 Appendix

We have only demonstrated the relevant CUDA kernels and not the surrounding code or main functions which are required to execute the kernels. The entire collection of complete code samples can be found in the GitHub repository.

In our full sample code which we have used for benchmarking, we have used 2 header files which includes helper functions to facilitate CUDA error checking in our code - *helper_cuda.h* and *helper_string.h*. Both of these helper header files are included in the NVIDIA CUDA SDK under the *samples/inc/* folder. In the Awkward Array production code we might use the same CUDA error checking header files or we might decide to write our own CUDA error checking code.