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 -

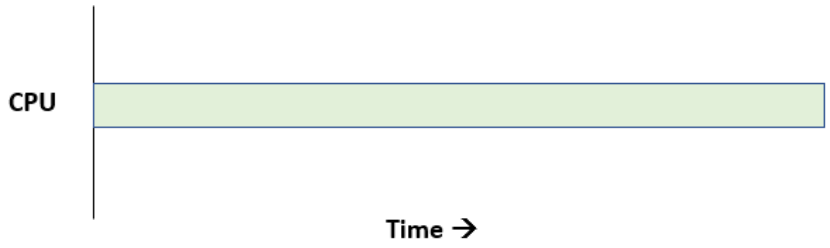

Figure 1: Visualizing sequential CPU algorithm

2 Sequential algorithm on an Nvidia GPU

2.1 Sequential algorithm on a single thread

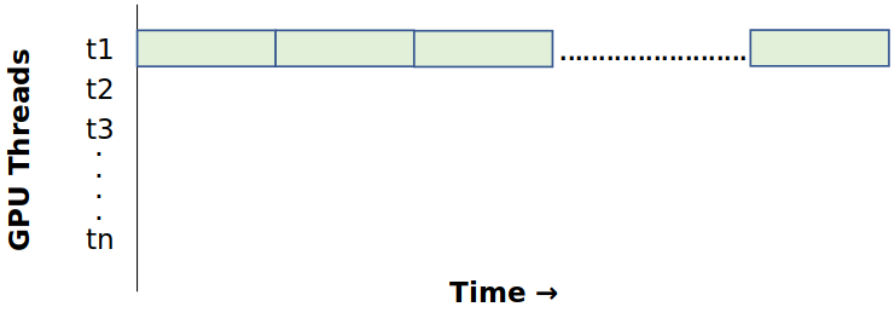

Figure 2: Visualizing GPU threads at work

```
__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

2.2 Sequential algorithm on multiple threads

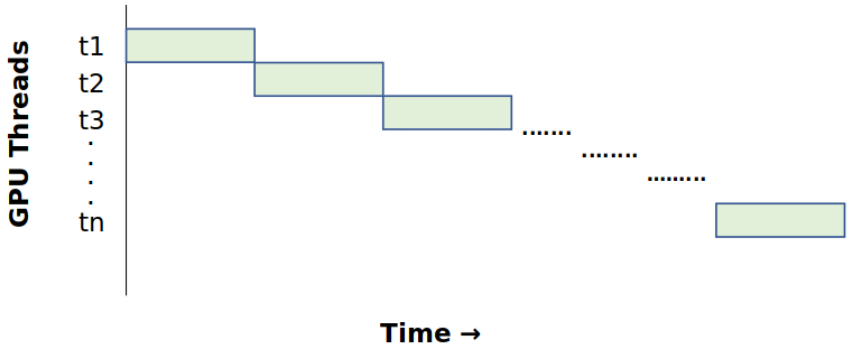

Figure 3: Visualizing GPU threads at work

```
__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;
}
```

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

```
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);
  }
}
```

3 Parallel algorithm on an Nvidia GPU

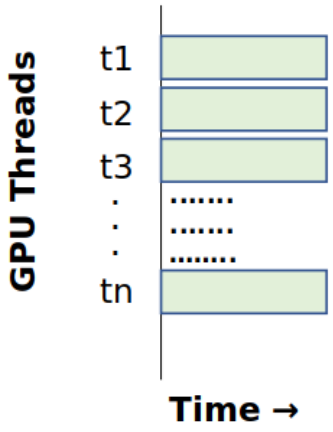

Figure 4: Visualizing GPU threads at work

```
#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));
}
```

4 Benchmarks

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.