doc.md

Index

- Index
- Problem Statement
- Naive CPU Implementation
- SIMD CPU Implementation
- Belloch Scan
- Optimizing Belloch
- Warp level primitives

Problem Statement

The evaluation task is to write an appropriate kernel for awkward_listarray_compact_offsets.

```
    template <typename C, typename T>
    ERROR awkward_listarray_compact_offsets(T* tooffsets, const C* fromstarts, const C* fromstops, int64_t startsoffse
      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();
    }
```

As stated in the email this is essentially a prefix-sum problem which can be parallelized and computed in O(lg n) time with O(n lg n) work. Compared to the sequential which is O(n) time and work. The benefit of using a GPU for the scan primitive was realised as early as 2006. Sengupta, Harris et al published a paper showing promising benchmarks in 2007.

An important question is how to deal with errors on the GPU. Not only we must avoid any race conditions, it is probably better to give the lowest index where `stop < start` to repicate behaviour of the CPU function. We will discuss this later when we show the GPU implementations.

Another thing to note is the type specifications for this function which are:

```
ERROR awkward_listarray32_compact_offsets64(int64_t* tooffsets, const int32_t* fromstarts, const int32_t* fromstops, i
  return awkward_listarray_compact_offsets<int32_t, int64_t>(tooffsets, fromstarts, fromstops, startsoffset, stopsoffs
}
ERROR awkward_listarrayU32_compact_offsets64(int64_t* tooffsets, const uint32_t* fromstarts, const uint32_t* fromstops
  return awkward_listarray_compact_offsets<uint32_t, int64_t>(tooffsets, fromstarts, fromstops, startsoffset, stopsoff
}
ERROR awkward_listarray64_compact_offsets64(int64_t* tooffsets, const int64_t* fromstarts, const int64_t* fromstops, i
  return awkward_listarray_compact_offsets<int64_t, int64_t>(tooffsets, fromstarts, fromstops, startsoffset, stopsoffs
}
```

```
<typename C, typename T>

<int32_t, int64_t>
<uint32_t, int64_t>
<int64_t, int64_t>
```

The system I will be using to benchmarks is as follows:

- i5 8600K 3.6 GHz (Coffee Lake)
- 8 GB RAM 2666MHz
- GTX 1060 6 GB (1280 CUDA cores, 10 SM's)

Naive CPU Implementation

The naive CPU implementation can be found in `naive_cpu.cpp` this is simply a copy-paste of the original code, I however wanted to see the difference between different optimization levels and template signatures. Time was measured using `std::chrono`.

The difference in the `-O2` and `-O3` versions of the code is that `-O3` realises that while computing `tooffsets[i + 1] = tooffsets[i] + (stop - start)` there is never any need to fetch `tooffsets[i]` as it is already present in a register. The x86 ASM was generated to realize the difference.

The template type `<int64_t, int64_t>` is a bit slower than the other templates as expected due to more memory bandwidth and larger datatype being used. In total it uses 1.5x the memory compared to other templates.

Note that to have different sections of code compile to different optimization levels we used function attributes. While GCC has support for them other compilers don't necessarily do (for example Clang).

The results for -O3 can be found in `simple_bench2.txt` and the complete results can be found in `simple_bench.txt`.

Length	Opt	Type	Avg time	Std deviation
2^{20} = 1048576	3	`<int64_t, int64_t>`	2.4774ms	0.151838
2^{21} = 2097152	3	`<int64_t, int64_t>`	4.7833ms	0.0571438
2^{22} = 4194304	3	`<int64_t, int64_t>`	9.1731ms	0.162497
2^{23} = 8388608	3	`<int64_t, int64_t>`	17.9429ms	0.141502
2^{24} = 16777216	3	`<int64_t, int64_t>`	35.7489ms	0.354522
2^{25} = 33554432	3	`<int64_t, int64_t>`	71.0009ms	0.577765

Length	Opt	Type	Avg time	Std deviation
2^{26} = 67108864	3	`<int64_t, int64_t>`	141.316ms	0.99137
2^{27} = 134217728	3	`<int64_t, int64_t>`	396.648ms	0.425821

[simple_bench2.txt]

SIMD CPU Implementation

It feels extremely unfair to compare a naive CPU implementation to any other solution as it basically ignores majority of developments since 2004. With multiple cores and SIMD vector extensions our program should become much faster.

With multiple cores the program should scale very well with the number of cores provided significant memory bottlenecks don't occur and size of data amortizes the cost of thread creation and broadcast.

To investigate SIMD I wrote a program for the template `<int32_t, int32_t>` although that template is not used. This is just to give us a favourable advantage for a primitive implementation. My computer AVX2 (256 bit width) but no AVX512 (512 bit width) so using `int64_t` would have been relatively pointless as it can operate on 4 elements at a time on my PC (4 * 64 = 256), whereas on 8 elements with `int32_t`.

To implement SIMD the main options we have are:-

- Compiler pragmas
- SIMD Intrinsics
- A templated C++ SIMD library like Vc, xsimd, etc.

I found the Intel SIMD Intrinsics most easy to use and there are very deterministic guarantees as to what assembly is generated. Probably only minor rearrangements will be done to improve pipelining.

I found a thread on stack-overflow where an implementation of scan for intel CPU can be found. I copied the code making minor modifications as we operate on integers rather than floats. For a pure prefix sum we get a 2x improvement over a sequential version (I have reproduced this improvement as have many people on the stack-overflow thread). However, when I modify the program for our problem there is barely an improvement(7% ish). I failed at the task and even failed to understand why.

One reason can be that if we use the roofline model to visualize our kernel the arithmetic intensity is extremely low with memory being the bottleneck rather than compute.

I used the LLVM Machine Code Analyser (LLVM-MCA) to analyse my kernel. And I have "potentially" identified the problem. The text output llvm-mca can be found in `mca.txt`.

```
vpbroadcastd    .LCPI0_0(%rip), %ymm1
vmovdqu -4(%r14,%rax,4), %ymm2
vmovdqu -4(%rbx,%rax,4), %ymm3
```

[Lines 77 - 79]

All the instructions in lines 77-79 use ports 2 and 3. Three instructions using the same ports should mean that there are significant pipeline stalls. I don't really have experience with llvm-mca or analysing code at the hardware specific level. At this point I gave up as this might be hardware specific and can may be resolved through better instruction scheduling or better compilers.

The code for this program can be found in `simd_cpu.cpp`. The benchmarks for -O2 and -O3 can be found in simd_bench_O2.txt and simd_bench_O3.txt

Note: Don't forget the `-mavx2` and `-march=native` compilation flags.

Belloch Scan

The NVIDIA blog gives a very simple implementation of belloch scan which can essentially be copy pasted for our purpose.

Assuming each thread works on `2` elements and each block has `512` threads. Each block will work on `1024` elements. So for arrays of length > 1024 we must store the partial sum of each grid block into a different array, then call scan on that array (recursively so that length can be of any arbitrary size) and then add the partial sums back to the original array.

An important question is how to report failure. Using the same strategy as suggested in the email for `awkward_numpyarray_getitem_next_range` will fail as there is a race condition to overwrite `ERROR err`. It is also not possible to prematurely terminate the kernel when an error is detected. There is an option to prematurely crash which is to use `assert` but that will crash the entire cuda context so we will need to do a `cudaDeviceReset()` which has significant overhead. Not to mention all the memory of our process will get reset. There is a better way to prematurely quit but it requires two extra `__syncthreads()` in the `scan_diff` kernel. There are some other alternate ways too whose performance needs to be evaluated before making a serious production ready decision. At this point in time I feel it's best to delay this task.

For errors I have chosen to create an array `bool *error_block` which is initialised to `false`. Whenever a thread encounters an error it writes to the shared memory location within that block and finally thread 0 writes the value of that to `error_block`.

We have three kernels

- scan_diff: Write the difference of `stop - start` to `tooffsets`.
- scan_kernel: Generic scan kernel for partial sums
- broadcast: Write partial sums to an array.

The benchmarking results are very promising and can be found in `belloch_bench.txt` and in the table below.

Length	Avg time	Std deviation
2^{20} = 1048576	0.845955ms	0.0219021
2^{21} = 2097152	1.57381ms	0.219242
2^{22} = 4194304	2.84332ms	0.480012
2^{23} = 8388608	5.08526ms	0.458507
2^{24} = 16777216	8.3371ms	0.832314
2^{25} = 33554432	16.5165ms	1.71707
2^{26} = 67108864	27.8151ms	1.81559

Length	Avg time	Std deviation
2^{27} = 134217728	73.2768ms	9.56866

A head-to-head comparison with the naive CPU implementation yields us:

Length	GPU time	CPU time (-O3)
2^{20} = 1048576	0.845955ms	2.4774ms
2^{21} = 2097152	1.57381ms	4.7833ms
2^{22} = 4194304	2.84332ms	9.1731ms
2^{23} = 8388608	5.08526ms	17.9429ms
2^{24} = 16777216	8.3371ms	35.7489ms
2^{25} = 33554432	16.5165ms	71.0009ms
2^{26} = 67108864	27.8151ms	141.316ms
2^{27} = 134217728	73.2768ms	396.648ms

Optimizing Belloch

The NVIDIA blog suggests a number of things to speed up belloch. Namely:

- Avoiding bank conflicts: This is done by changing the mapping of an element index in the shared memory. This however requires more memory per block decreasing occupancy.
- `elements_per_thread` : can be 2, 4, 8, ... The larger the number the more work an individual thread will do. Larger the number, smaller the occupancy(concurrent threads executing on a single SM) as the amount of shared memory required per thread will increase.
- `threads_per_block` : While we started out with 512 modifying this number will result in different occupancy figures.
- CUDA Vector types: By using inbuilt vector types like `longlong4` and `uint4` we can make loads and stores a tad more efficient. However, memory locations have an alignment requirement which cannot be met in the general case as `startsoffset` and `stopsoffset` is arbitrary. Along using vector types will mean that we can no longer write templated functions.
- Loop unrolling: using the compiler pragma "#pragma unroll". Results in more code but less work for the SP.

Tuning the parameters to find the best implementation requires a deep analysis and it is likely to change with the hardware. I have avoided that for now as this is a skill which must be acquired and using the correct tools to do such an analysis is required.

Instead here I have an implementation where we do 4 elements per thread, 256 threads per block, bank-conflict avoidance and loop unrolling. The results are somewhat better than the original implementation.

Length	Avg time	Std deviation
2^{20} = 1048576	0.650486ms	0.0205613
2^{21} = 2097152	1.16748ms	0.168066
2^{22} = 4194304	2.03023ms	0.183592
2^{23} = 8388608	3.86676ms	0.403391
2^{24} = 16777216	7.43524ms	0.79665
2^{25} = 33554432	14.611ms	1.71404
2^{26} = 67108864	25.1895ms	2.20119
2^{27} = 134217728	64.5166ms	9.2891

File: belloch4_gpu.cu Benchmark: belloch4_bench.txt

Warp level primitives

An alternate method of implementing scan is to use warp level primitives (link). The idea is that using these primitives it is possible to communicate within a warp(32 threads in NVIDIA) without using shared memory. These primitives are faster than shared memory hence there may be an advantage.

The cuda-samples have implemented a version scan using these primitives which can be found here. I essentially took that code and modified for our purpose. The implementation can be found in `shfl_scan.cu`.

The resulting benchmarks are as follows:

Length	Avg time	Std deviation
2^{20} = 1048576	0.449293ms	0.0301127
2^{21} = 2097152	0.814285ms	0.15225
2^{22} = 4194304	1.32608ms	0.0287871
2^{23} = 8388608	2.50158ms	0.0377251
2^{24} = 16777216	4.72269ms	0.0348666
2^{25} = 33554432	9.15111ms	0.0504847
2^{26} = 67108864	18.1809ms	0.0344593
2^{27} = 134217728	36.2698ms	0.33382

This implementation is easier to understand and almost twice as fast as the original GPU implementation and 5-10x faster than the naive CPU -O3 code.

The only potential issue is that it requires a compute capability of over 3.0 which is extremely reasonable. The last graphics card released which is under 3.0 was the GTX 560 Ti released in 2011.

File: shfl_scan.cu scan_bench.txt