A robust single-pass scan implementaion on GPUs

Troels Henriksen

September 21st, 2017

The problem

Scans (or “generalised prefix sums”)

A scan computes the reduction of prefixes of an array:

```
scan (+) 0 [1,2,3,4,5]
  == [reduce (+) 0 [1],
      reduce (+) 0 [1,2]
      reduce (+) 0 [1,2,3],
      reduce (+) 0 [1,2,3,4],
      reduce (+) 0 [1,2,3,4,5]]
  == [1, 3, 6, 10, 15]
```

Scans are useful for solving problems that do not look parallel at first glance.

Making a better scan

Scans are useful for solving problems that do not look parallel at first glance. Scans are memory-bottlenecked, and so we want an implementation that minimises memory traffic.

- *Common implementation*: $O(4n)$ memory ops.
- *Single-pass implementation*[0]: $O(2n)$ memory ops, but not robust.

The project is about prototyping a CUDA single-pass scan that is robust, such that it can be used by the Futhark compiler.
[0] "Higher-Order and Tuple-Based Massively-Parallel Prefix Sums", PLDI 2016

The common solution

An outline of the common scan algorithm

Assume we want to compute `scan (+) 0 [0...11]` using three CUDA thread blocks. We use three kernels:

1. Block `i` is assigned chunk `[i*4...(i+1)*4-1]` of the input, which it scans internally. This gives an intermediate result:

   ```
   [0, 1, 3, 6, 4, 9, 15, 22, 8, 17, 27, 38]


      block 0      block 1        block 2
   ```

2. The last element ("carry out") of each chunk is scanned in a single block, giving an array of *carry ins*:

   ```
   scan (+) 0 [6,22,38] == [6,28,66]
   ```

3. Block `i` reads carry in `i-1` and adds it to its partial result:

   ```
                [0, 1,   3,  6]
   ++ map (+6)  [4, 9,  15, 22]
   ++ map (+28) [8, 17, 27, 38]
   == [0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66]
   ```

Too slow

The problem is that we have to write the entire intermediate result to global memory after the first kernel, only to read it back in the third kernel. In total, we read and write the array four times.

1. <= read initial array
2. => write intermediate
3. <= read intermediate
4. => write final result

A faster solution is based on a limited form of *communication* between blocks, via atomic operations on global memory, where the intermediate result is never written to global memory.
We will still write the *carries* to global memory, but these are *much fewer in number* (one per thread block).

The faster solution

The faster solution

We use an extra auxiliary block that is not given a chunk of the original input.

1. Each block computes a partial result as before, but keeps it in registers (one element per thread), or shared memory. Each block writes the carry out to an auxiliary global memory array, but *the kernel keeps running*.
2. The auxiliary block waits for all carries to be available, scans them, and writes the result to global memory.
3. The blocks are notified that the carry ins are available, they read them, apply them to their partial results, and write the final result to global memory

Fast but *not robust*.

Making the faster solution robust

Problems

The fast solution assumes all thread blocks fit on the GPU, or else the auxiliary block will never receive *all* carry-outs. This is *not valid* in general, in particular on shared systems.
Other (minor) assumptions that are true in practise:

- Progress: once a CUDA block is started, it is not descheduled again.
- Consistency: atomic global memory operations are (eventually) visible to all blocks of a kernel.

Idea for fixing the robustness issue

- Block `i` does not need the scan of *all* carry outs, only those of preceding blocks `j < i`.
- Two candidate solutions:
  1. Duplicate the (partial) carry scan for each block.
  2. Do a (sequential?) progressive scan in the auxiliary block where partial results become available as soon as possible.

Significant wrinkle:

- CUDA (and OpenCL) does not guarantee that block `i` is launched before block `i+1`.
- Solution: instead of using the "physical" block IDs, perform "dynamic numbering" using a counter in global memory via atomic operations.

Concrete project goals

- Implement the $O(2n)$ scan *robustly* in CUDA (we already have a hacky prototype for inspiration).
- Experiment with different ways of computing the carry scans.
- Ensure the scan can handle any input size.
- Have fun getting code to run fast!
- If this turns out to be a good idea, we'll teach the Futhark compiler to use this technique (outside PMPH scope, but fun master's project)!