CS194-15: Engineering Parallel Software Assignment 6

CS194 Course Staff

October 17, 2013

As your final CS194 homework, you'll be implementing radix sort on GPUs using OpenCL. While this might sound scary, you've already implemented the key kernel in data parallel radix sort, prefix scan, in last week's homework. After implementing radix sort, you can truly call yourself a GPU programmer.

1 Background on radix sort

You'll be implementing a least significant digit (LSD) version of radix sort. In this version of radix sort, we'll process keys starting with the least significant bit (bit 0) and proceed to the most significant bit (bit 31).

At each stage (bit) of the LSB radix sort, we place keys into two buckets. One bucket holds keys with the current bit set while the other bucket holds keys with bit unset. While we split the keys into two separate bucks, the key to the correctness of radix is that the original order of the keys (within a bucket) is preserved. The split computation is performed 32 times (for the case of 32-bit) integers to fully sort the array.

```
1  for(int32_t k = 0; k < 32; k++)
2    {
3      /* compute entries with set lsb */
4      for(int32_t i = 0; i < n; i++)
5        temp[i] = (in[i] >> k) & 0x1;
6
7      /* compute indices of elements with 0 in lsb */
8      scan(temp, zeros, n, 0);
9      /* compute indices of elements with 1 in lsb */
10     scan(temp, ones, n, 1);
11
12     /* scatter partially ordered output*/
13     for(int32_t i = 0; i < n; i++)
14       {
15         if(temp[i])
16           idx = zeros[n-1] + ones[i] - 1;
17         else
18           idx = zeros[i] - 1;
19         out[idx] = in[i];
20       }
21
22     /* Not an in-place sort,
23      * need a pointer swap */
24     tPtr = in;
25     in = out;
26     out = tPtr;
27   }
```

Listing 1: An approach for implementing data-parallel radix sort

Listing 1 presents the core of the data-parallel radix sort algorithm. Line 1 iterates over the 32 bits in an int32_t while lines 4 and 5 compute if each key has the current bit set. Lines 8 and 10 compute an additive scan on the set bit temporary array in order to find the output index. Finally, we need to merge the two buckets back into a partially sorted array. The address computation for keys with a zero in the current bit under inspection is straightforward and only requires accessing the zero scan array.

The set bit index requires using the last entry of the zero array as an offset (think about why). Finally, radix sort is an out-of-place algorithm, so lines 24 through 26 handle double buffering.

2 Your task

For this assignment, you'll need to implement a fully functioning GPU radix sort. We'll provide the core of the scan primitive, but you'll need to add a little code to the scan kernel though (from HW5). Your primary challenge is to implement the rest of the algorithm. This should require implementing just one new kernel then hooking everything appropriately. The starter code includes a correctness checking routine too, so you'll know when your code works.

2.1 Submit your code

We want your commented radix sort code. Make sure the comments explain how your code works and any design decisions you made when implementing radix sort.

2.2 How does it perform?

Our correctness testing code calls the C standard library implementation of quicksort. We also record the CPU runtime of quicksort and the GPU runtime of radix sort. If you use the included benchmarking script (run with the command **"bash ./bench.sh"**), you will be rewarded with a CSV (comma-separated values) file that contains performance data for both CPU and GPU implementations. Please plot these results and explain the performance behavior. In particular, what array length is required for better performance on the GPU than the CPU? We want both your graph and a short paragraph of analysis included in a PDF document submitted with your code.