APMA 2822b Homework 4

Ryan Greenblatt

May 2019

1

Code is attached in the email. I have included a CMakeLists.txt file which can be used to compile the code on the CCV. The `cuda/10.0.130` module must be loaded. I also added the SLURM scripts I used. Note that gencode is specified to be "arch=compute_70,code=sm_70" which may cause issues on older GPUs.

Algorithm and Implementation

I found the maximum on the CPU and GPU by traversing the array and comparing each element to the maximum element found thus far. On the CPU, a single thread was dedicated to each array while parallel reductions were used on the GPU. Two GPU implementations were tested for this task. One used one warp per row and assigned multiple elements to each thread initially. Shuffle operations were used to determine the maximum among the thread maximums found. The other implementation another has one thread per every element and uses shuffle operations for parallel reduction and shared memory to collect data from each warp. The reductions performed after syncing with shared memory are also performed using shuffle operations.

I found the nth maximum on the CPU by using the quickselect algorithm which has expected $O(n)$ runtime. The algorithm involves selecting some element in the array and using that element to partition the array into two parts. The first part is less than the element while the second part is greater than or equal to the element. This can be done entirely in place. Then, the nth element must be in the sub array of smaller elements if nth is less than the number of elements in the smaller array. In the other case, it must be in the greater or equal sub array. This can be recursively done until the nth element is found. Because this algorithm can't be easily parallelized and involves branching, a different algorithm was used on the GPU. The quickselect algorithm was also tested on the GPU with only parallelism over the each different array for the purposes of comparison.

Least significant digit radix sort was used on the GPU. This has $O(n)$ runtime. This involves sorting an array of elements by some number of the least significant bits. I used 2 bits. The values can then be binned and sorted based on those bits by computing how many values occur before each value in the array. This can be done by counting the number of values of each bin per thread and using prefix sum parallel reductions. The number of threads used was the size of the array divided by 4 to have 4 values per thread for most threads. Because the number of values per warp is less than 256, the counts per each warp may be stored

in 8 bit unsigned integers. Because there are 4 bins, these values can be packed into a 32 bit unsigned integer. Instead of looping over each bin in the warp local reductions, packed 32 bit unsigned integers can be operated on to compute prefix sums. After the warp local operations 32 bit unsigned integers must be used to ensure no overflow occurs.

2 Analysis

A V100 GPU was used which has an approximate memory bandwidth of 800 GB/s. A reasonable estimate of CPU memory bandwidth is 55 GB/s. The code was tested with $N = 16384$ and $M = 1024$, so this is used in analysis.

Finding the maximum element in an array on the CPU just requires using each element in the array once, so the expected memory access is just the size of the array. A total of $N * M * 4 = 67108864$ bytes must be accessed which is 0.0625 GB. The expected time is $\frac{0.0625}{55} = 0.00114$. The GPU will require additional reduction operations. In the case of the approach which uses one warp per every array, there are an insignificant amount of additional reductions. The memory access is well approximated by the same analysis done for the CPU. The expected time is $\frac{0.0625}{800} = 0.0000781$ seconds.

The number of iterations required by the quickselect algorithm will vary and the number of elements which must be accessed in each iteration will also vary. As such, it is difficult to estimate without making assumptions. This estimate will be done assuming a constant reduction factor per iteration. Further, it isn't unreasonable to estimate using an infinite sum even though the algorithm doesn't require infinite iterations because the additional memory accesses of the last iterations are small. In addition to reading the memory, the memory will need to be copied to new arrays which will double the total memory bandwidth. The index and value must both be stored which is 8 bytes. Assuming that the array is reduced by half each iteration, the total amount of memory access is $2\frac{N*M*8}{1-\frac{1}{2}} = N * M * 32 = 536870912$ in bytes or 0.5 GB. The expected time is $\frac{0.5}{55} = 0.00909$ seconds.

Radix sort will require iterating over all elements in the array for each set of significant digits which must be binned. In this implementation, 16 iterations are required. Each element iterated over also required updating a single byte counter. In addition to iterating over the array, reductions must be computed. Finally, indices and elements must be copied to the appropriate place in the other array. The last few reductions will be ignored because the total memory bandwidth is small. The total bytes required is:

$$16 * (N * M * (4 + 1 + \log_2(32) * (4 + 4) + 4 + 4) + \frac{N * M}{32} * \log_2(32) * (4 + 4) = 14562623488$$

This is 13.5625 GB. The expected time is $\frac{13.5625}{900} = 0.0151$ seconds.

3 Results

All testing was done on the CCV with a GPU (V100) and 8 CPU cores. Time required to copy data and convert between formats wasn't counted. I ran each test 10 times and averaged the last 8 runs to not count time required for shared memory migration and copying

between formats.

The GPU implementation for finding the maximum element was significantly faster than the CPU implementation. The version which uses one warp for every array rather than a thread for every value was slightly faster.

The GPU radix sort was significantly slower than the CPU algorithm. However, the GPU radix sort is far more scalable.

As an additional test, I also evaluated running the quickselect algorithm on the GPU with one block per each array. This massively underutilizes the width of the GPU, but turned out to be faster than the GPU radix sort. This suggests that either the implementation of radix sort I wrote is very poor or radix sort is ill suited to the problem.

variant	time (s)
CPU time	0.00161193
GPU time warp per array	0.000357507
GPU time	0.0004617

Table 1: Time to find the maximum element.

variant	time (s)
CPU time quick select	0.0147669
GPU time radix sort	0.997017
GPU time quick select	0.054615

Table 2: Time to find the nth element. Note that the radix sort also finds the full ordering including the max element

4 Conclusions

I learned that `cuda-gdb` is very useful when debugging GPU code. I also learned that radix sort can be very tricky to implement massively in parallel.