APMA 2822b Homework 4

Ryan Greenblatt

April 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 clear. Because this algorithm can't be easily parallelized, a different algorithm was used on the GPU.

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 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 timed each run individually so that the start up memory transfer cost for each approach could be evaluated. Results are reported in terms of the time required for one iteration of the provided matrix. The exact time and GLOPS of the computation will depend on the characteristics of the sparse matrix.

On the GPU, the cuSPARSE implementation was the fastest. The ELLPACK and standard implementations were very similar and almost as fast as the cuSPARSE implementation. On the CPU, the CRS implementation was the fastest. The optimal value for ELLPACK row loop unrolling was found to be 4. Storing the number of elements in each row and only looping through the required elements improved CPU and GPU performance. Using a texture resulted in no improvement for CRS or ELLPACK. The timing results can be seen in tables 1 and 2.

Unified and device memory had similar performance after the first iteration. The first iteration using managed memory which switched from the CPU to GPU was substantially slower because the managed memory needed to be transfered from device to host.

Interestingly, the average kernel timings reported by nvprof were substantially different from the timing found by the code. I used the CUDA event API to time CUDA kernels in the code. I tested on both my laptop and the CCV and the disparity wasn't at all present on my laptop. The average kernels durations as reported by nvprof are given in table 3 below. I think that most likely the kernel timings reported by nvprof are incorrect and the issue may be related to the nvprof error which using CUDA 10. The issue may also be occurring in CUDA 9, but isn't properly reported.

Profiling the code using nvprof allowed for detailed inspection of the time required for each CUDA API call. The timings for each CUDA API call are given in table 4 below. `cudaFree` was found to use a large percentage of the overall time of the program (437.68 ms). It isn't clear to me why this would be the case. `cudaMalloc`, `cudaMallocManaged`, and `cudaMallocPitch` also took substantial program execution time. `cudaMemcpy` consumed most of the remaining time spent on CUDA API calls.

Method	Average time	1	2	3
CPU	1.600510e-04	7.989560e-04	1.593470e-04	1.586510e-04
GPU	4.646400e-05	5.334400e-05	4.668800e-05	4.742400e-05
GPU texture memory	4.848800e-05	1.091520e-04	5.129600e-05	4.912000e-05
CPU managed before GPU	1.610724e-04	2.447670e-04	2.344980e-04	1.753500e-04
GPU managed	4.674400e-05	4.469568e-03	4.918400e-05	4.796800e-05
CPU managed after GPU	1.610724e-04	2.447670e-04	2.344980e-04	1.753500e-04
cuSPARSE	3.886400e-05	1.065280e-04	4.323200e-05	3.977600e-05

Table 1: The recorded timings for each method utilizing the CRS data format. The average is the average over the 10 runs excluding the first two runs. Only the first 3 runs are shown to highlight transfer times associated with managed memory.

Method	Average time	1	2	3
CPU	2.352189e-04	7.101640e-04	3.498100e-04	2.545160e-04
GPU	4.766800e-05	5.331200e-05	4.723200e-05	4.729600e-05
GPU texture memory	4.765600e-05	4.881280e-04	4.928000e-05	4.761600e-05
CPU managed before GPU	2.949605e-04	1.024591e-03	8.099850e-04	3.533850e-04
GPU managed	4.756000e-05	8.975392e-03	5.948800e-05	4.678400e-05
CPU managed after GPU	2.949605e-04	1.024591e-03	8.099850e-04	3.533850e-04

Table 2: The recorded timings for each method utilizing the ELLPACK data format. The average is the average over the 10 runs excluding the first two runs. Only the first 3 runs are shown to highlight transfer times associated with managed memory.

Method	Average kernel time
ELLPACK	865.07us
CRS	260.40us
ELLPACK texture memory	82.038us
CRS texture memory	43.023us
cuSPARSE	38.012us

Table 3: The average running time of each kernel as reported by nvprof. I think these timings are likely wrong.

API call	Time
cudaFree	437.68ms
cudaMalloc	296.62ms
cudaMemcpy	30.429ms
cudaMallocManaged	20.848ms
cudaEventSynchroni	16.806ms
cudaDeviceSynchron	8.6555ms
cuDeviceGetAttribu	1.2526ms
cudaLaunchKernel	607.89us
cuDeviceTotalMem	551.48us
cudaEventRecord	321.05us

Table 4: The total time for all API calls which took longer than 200 us.