HW 3: SPMV

Ming Tai Ha

March 27, 2017

Problem

Implement 3 SPMV algorithms: Simple Atomics, Segmented Scan, and one of our own design in CUDA.

Implementation

Simple Atomics Based SPMV

In this implementation, each thread calculates the dot product between the vector and a row in the matrix. For each index in the vector (and row), the thread calculates the corresponding product and adds it to the corresponding entry of the output array. The addition of each partial product to the output entry is atomic. This ensures that additions to the partial product are free of any race conditions. Intuitively, it may be computationally expensive in order to guarantee atomicity for an instruction. In later implementations, however, we shall see that the cost of atomic operations may not necessarily be very high.

Segment Scan

The segment scan approach perform a prefix sums while taking into account the rows to which each nonzero element belongs. The standard way of tracking the rows to which each nonzero element belongs is to use flag array. Since I am given a sparse matrix in matrix market format, I use the row indexes directly for comparison. I noticed that positions of the nonzero elements in the sparse matrix was not sorted ascending order with respect to their rows. Since the segmented scan algorithm SPMV algorithm requires elements be ordered by their rows, I needed to first sort the elements. My implementation has each thread in a thread block computing a partial product and storing the result in shared memory. Each thread also stores the row to which the partial product belongs in shared memory as well. Then, the algorithm perform a segmented parallel scan on each warp. While the algorithm can be extended so that results from warps of the same block can be reduced, I chose to implement final segment scan with atomicAdd because the performance cost of atomicAdd was low. In my own algorithm, however, I will not be using atomics to implement my optimization on Segment Scan.

Own Design

The algorithm which I implemented is an optimization of the segmented scan. One of the difficulties in the implementation of segment scan is that it is not possible to coordinate threads of execution in CUDA. In order to take maximal advantage of the number of processors on the GPU, we compute partial sums from each thread block. One difficulty in aggregating the partial sums from a segmented scan is identifying a quick way of mapping partial results stored in shared memory into an array stored in global memory. After performing a segmented scan on a subarray, we want to store only the largest partial sums associated with each row whose elements are in the subarray. In general, it is not possible to know beforehand the the number of rows with elements in the given subarray. To perform SPMV, we only need a 'weaker' mapping.

Let A be the sparse matrix matrix represented in matrix market format be of length L, row, col be arrays each take values from $[1...M]$ and $[1...N], M, N \in \mathbb{N}$, where $row[i]$ and $col[i]$ are the row and column indexes of $A[i]$, respectively. Let x be of length N, $y = A * x$ be the output of SPMV between A and x. Let K be the number of thread blocks used, t be the number of threads per thread block, and

Figure 1: Thread Block Size: 1024; Number of Threads: 8

1024/8	(in ms)		
	Atomics	Segmented Scan	Own Design
cant	0.4	0.844	1.554
circuit5M_dc	3.484	7.407	14.002
consph	0.553	1.249	
FullChip	14.573	10.61	20.074
mac_econ_fwd500	0.315	0.597	1.068
mc2depi	0.361	0.857	1.505
pdb1HYS	0.395	0.875	
pwtk	1.042	2.378	
rail4284	4.525	4.604	9.363
rma10	0.43	0.972	1.791
scircuit	0.273	0.449	0.785
shipsec1	0.745	1.642	2.986
turon_m	0.234	0.421	0.693
watson_2	0.503	0.755	1.471
webbase-1M	0.601	1.243	

$K*t \geq L$. Since row is sorted based on their row indexes, equal subdivisions of sparse matrix A, denoted $A_i, i \in [1...K]$, will guarantee that we get a nondecreasing sequence of row indexes row_i corresponding to A_i. While it is possible for the elements of a row of the sparse matrix to span multiple thread blocks, it is also possible for elements of a row of the sparse matrix to be completely contained in one thread block. If the latter case occurs, then performing a segmented scan on the thread block containing all partial products the row will compute an entry which can be stored in the output array. So, we see that the only partial sums which cannot be stored in the output array are the ones which we are not sure span more than one thread block. We need to consider at most two such partial sums: the partial sum of the first row encountered in the thread block, and the sum of the last row encountered in the thread block.

By writing all partial sums where its row elements are completely in one thread block, we now have a more predictable mapping of elements when aggregating partial sums and row indexes from shared memory to global memory. A global array storing partial segmented scan results only needs to be twice the size of the number of thread blocks. To aggregate partial sums, we assign each thread block with two consecutive entries. This means that the global array storing the partial sums (aka global sums array) has length $2 * K$. We also keep track of the rows to which the partial sums belong in a global array (length $2*K$). A thread block is given two consecutive entries in the global row index array. Each thread block writes the first and last row encountered in the first and second entries associated with its block ID in the global row index array, and writes the first and last partial sum encountered in the first and second entries associated with its block ID in the global sums array. When a thread block is composed of elements belonging to only one row, then we write the last partial sum as the first element and 0 as the second element to the global sums array. We also write row index for both associated entries in the global row index array. Thus, we have a bijection between the partial sums (and row indexes) of a thread block and the memory location of the global arrays. This mapping also implies that the aggregation step can be done without atomics.

By doing so, we have two global arrays, one which stores the partial sums, and one which stores the associated row indexes. This reduces the problem to a smaller version of the same problem. We can use this method to repeated aggregate partial scans until all partial scans fit in one thread block, at which point we perform a final segmented scan and write all of the partial sums calculated to memory. For each element in the output vector y, the algorithm only writes to $y_j, j \in [1...M]$ once, there is no need to use atomicAdd for writing to output.

Results

The results show that the atomics implementation is the fastest, which is quite surprising. I expected that the segmented scan would perform slightly better since it aggregates the partial sums. It should be noted that the implementation of my algorithm currently has bugs involving memory accesses. Missing entries mean that the code does not run correctly for those tests

My own implementation in general performs worse than both implementations. My implementation

requires a fair amount of *syncthreads*(). Moreover, the step of writing partial sums to the output vector has a large degree of thread divergence. Also, act of writing partial sums to the output vector does not take advantage of techniques like memory coalescing.