Histogram Computation
Assignment 3
CS6023

Anant Shah
EE16B105
Department of Electrical Engineering

September 23, 2018

Abstract

This report looks at computing the "N-count-grams" of a text, which is later represented as a histogram. Histogram computation is inherently a serial computation, where an array of elements is processed and each element is put into a bin based on the required property of that element. Efficient methods and algorithms to find the histogram using parallel processors are shown in this report. The histogram for "N-count-grams" is copmuted on the Tesla K40C using dynamic shared memory and privatization.

1 Histogram Computation

The "N-count-grams" are computed for a text file with N taken as a parameter to the program. Since "N-count-grams" include counting the length of each word in a set of N words, this method stores the length of the words instead of the whole word. The reason for storing the word lengths instead of the word will be given in the section where shared memory is used to store the word lengths. The most primitive algorithm would include storing the word lengths and the histogram in the global memory. Global memory access is a high latency operation and hence when each thread tries to access its respective set of words to calculate which bin it would go in, it will generally be hit by a stall hence reducing the efficiency of the computation. Since all the threads are working in parallel, two threads competing to write to the same bin will lead to read-write race conditions. Hence atomics need to be introduced so that the actual frequency of that bin is captured. Atomics essentially puts a lock on that data-element and tells other threads trying to access it that they need to wait as another thread is working on the element. Once that thread is done, then access to that data-element is up for grabs. As is evident, atomics in the global memory will lead to further stalls. This could cause a lot of harm as potentially thousand of threads could want access to a bin, but all of them would be stalled because of the atomic command. Techniques such as storing the word length in the shared memory and privatization of the histogram increase the performance of the histogram computation.

2 Performance Improvement Techniques

2.1 Use of Shared Memory

Consider the sentence "The quick brown fox jumps over the lazy dog". Say we are computing the "2-count-grams". Hence the words "The quick", "quick brown", "brown fox" form different sets which need to be added to their respective bins. Now if we use the fact that one thread will work on one set we can clearly see that data is being shared between threads. Hence the shared memory is utilized to increase the performance. Now the layout of the shared memory is such that it is divided into equally sized memory modules(banks) that can be accessed simultaneously[1]. Hence memory load or store operations to different banks simultaneously will give a higher bandwidth than that to a single bank. Memory access to the same bank by threads in a warp will be serialized hence decreasing the performance. For the Tesla K40C, the width of one bank is 32 bits, with 32 such banks. The data in the shared memory is stored such that successive elements go to successive banks. For example : logical address i goes to bank i%N where N is the number of banks. Therefore, storing word lengths which are 4 bytes long is better than storing the actual words which have a maximum length

of 20 bytes. The word lengths will exactly fit into each bank while the words themselves will be stored in different banks and hence their access by threads in a warp won't give the bandwidth that we expect with the word lengths. Now, suppose that the number of tasks(number of N-sets it acts on) per thread is NUM_TASKS and the block dimension is NUM_THREADS. The number of word lengths required in the shared memory would be

$$SHARED - WORDS = (NUM - THREADS * NUM - TASKS) + N - 1$$

These words are loaded synchronously by the threads in a warp to the shared memory in such a manner that the accesses lie in one burst section in the global memory. This "burst" provided is done by the hardware which realizes that the elements being accessed are contiguous and so instead of separate reads, provide one coalesced "burst". Once the loads are done to the shared memory, then they are accessed in such a manner that the 32 threads of a warp access different banks in the shared memory, hence getting a higher bandwidth. The parameters - number of tasks per thread and the number of threads per block are varied so as to get the optimal performance. Since the amount of shared memory required depends on the value of N, the allocation needs to be done dynamically. CUDA allows this by passing the amount of shared memory required as the third parameter in the kernel launch configuration.

2.2 Privatization

As mentioned earlier, atomics are required to update the bins in the global memory which lead to a decrease in the performance. One other optimization technique is to keep a private histogram in the shared memory of each block. The threads keep updating the bins in the shared memory atomically. Once all of them are done, they atomically update the hisotgram in the global memory. Atomic operations in the shared memory take much less time than in the global memory. When the number of bins are small(which can fit in the shared memory), the solution is efficient as each block has its private histogram, so once the threads finish computing, the private histograms just need to be atomically added in the global memory. However, the case for "N-count-grams" is a bit more complicated as the number of bins are much greater than the size of the shared memory. From the previous section, the optimal number of tasks per thread was 8 and the number of threads per block was 256. This accounts to a shared memory usage of

$$SHARED - WORDS(bytes) = 256 * 8 * 4 = 8192 = 8kB$$

Since we are using 256 threads per block, a maximum of 4 blocks will run on a Streaming Multiprocessor becuase of the constraint that the maximum number of threads on a SM is 1024. Since the amount of shared memory on a Tesla K40C is 48kB, each block will get a shared memory of 12kB. It is imperative that this limit is not crossed as this will lead to a reduction of the number of blocks on a SM hence decreasing the performance. Hence the partial histogram needs to be chosen such that the shared memory space occupied by it will be less than 4kB. Since a block requires other shared variables to facilitate dynamic movement of the privatized histogram, we choose the number of bins in the shared memory to be 512. This corresponds to around 2kB of shared memory.

The concept used to dynamically change the indices of the privatized bins is similar to that of caching. What happens in caching is that if we access an element in the global memory, then a contiguous set of element around it are loaded into the cache. Similarly, a counter is used to check which section of global memory is being visited frequently. After $1/4^{th}$ of the number of iterations for a thread, one of the thread checks if there are a large number of accesses to the global memory(greater than the one already present in the shared memory). If so, then all the threads combine and store the previous histogram value in the global memory and chnage the indices of the privatized histogram to the most frequently visited section in the global memory. Once the threads finish their operations, they atomically update the histogram in the global memory with the values of the privatized histogram.

3 Timings

The timings of the "N-count-gram" for N=1,2,3,4 and 5 on a file consisting of Shakespeare's text is given below. The time is taken using the linux time command. The parameters used are :

1. Number of threads per block(1-D) - 256

2. Number of operations per thread - 8

3. Number of bins in shared memory - 512

Sr. No.	N=1(seconds)	N=2(seconds)	N=3(seconds)	N=4(seconds)	N=5(seconds)
1	0.330	0.372	0.361	0.356	0.473
2	0.335	0.318	0.328	0.352	0.471
3	0.342	0.347	0.332	0.350	0.452
4	0.343	0.356	0.341	0.363	0.464
5	0.342	0.358	0.348	0.367	0.462
Average Time(s)	0.338	0.350	0.342	0.358	0.464

4 References

[1] `https://devblogs.nvidia.com/using-shared-memory-cuda-cc/`