Chapter 9

Parallel Patterns – Parallel Histogram Computation

An Introduction to atomic operations and privatization

Keywords: histogram, feature extraction, output interference, race condition, atomic operation, read-modify-write, memory bound, memory latency, memory throughput, memory coalescing, block partition, interleaved partition

CHAPTER OUTLINE

The parallel computation patterns that we have presented so far all allow the task of computing each output element to be assigned to a thread. Therefore, these patterns are amenable to the owner-computes rule, where every thread can write into their designated output element(s) without concern about interference from other threads. This chapter introduces the parallel histogram computation pattern, a frequently encountered application computing pattern where each output element can potentially be updated by all threads. As such, one must take care to coordinate among threads as

they update output elements and avoid any interference that corrupts the final results. In practice, there are many other important parallel computation patterns where output interference cannot be easily avoided. Therefore, the parallel histogram computation pattern provides an example with output interference in these patterns. We will first examine a baseline approach that uses *atomic operations* to serialize the updates to each element. This baseline approach is simple but inefficient, often resulting in disappointing execution speed. We will then present some widely-used optimization techniques, most notably privatization, to significantly enhance execution speed while preserving correctness. The cost and benefit of these techniques depend on the underlying hardware as well as the characteristics of the input data. It is therefore important for a developer to understand the key ideas of these techniques and reason about their applicability under different circumstances.

9.1. Background

A histogram is a display of the frequency of data items in successive numerical intervals. In the most common form of histogram, the value intervals are plotted along the horizontal axis and the frequency of data items in each interval is represented as the height of a rectangle, or bar, rising from the horizontal axis. For example, a histogram can be used to show the frequency of alphabets in the phrase "programming massively parallel processors." For simplicity, we assume that the input phrase is in all lower-case. By inspection, we see that there are four "a" letters, zero 'b" letters, one "c" letter, and so on. We define each value interval as a continuous range of four alphabets. Thus, the first value interval is "a" through "d", the second "e" through "h", and so on. Figure 9.1 shows the histogram that displays the frequency of letters in the phrase "programming massively parallel processors" according to our definition of value interval.

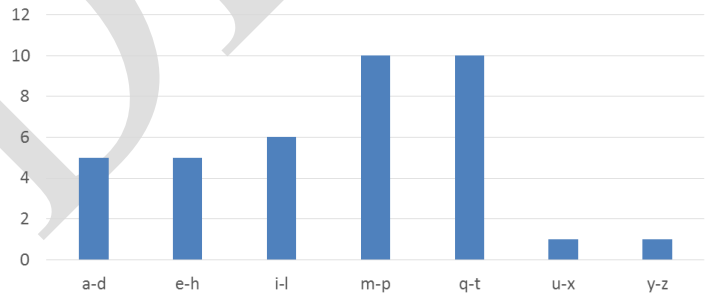

Figure 9.1 A histogram representation of "programming massively parallel processors."

Histograms provide useful summaries of data sets. In our example, we can see that the phrase being represented consists of letters that are heavily concentrated in the middle intervals of the alphabet and very light in the later intervals. Such shape of the histogram is sometimes referred to as a *feature* of the data set, and provides a quick way to determine if there are significant phenomena in the data set. For example, the shape of a histogram of the purchase categories and locations of a credit card account can be used to detect fraudulent usage. When the shape of the histogram deviates significantly from the norm, the system raises a flag of potential concern.

Many other application domains rely on histograms to summarize data sets for data analysis. One such area is computer vision. Histograms of different types of object images, such as faces vs. cars, tend to exhibit different shapes. By dividing an image into subareas and analyzing the histograms for these subareas, one can quickly identify the interesting subareas of an image that potentially contain the objects of interest. The process of computing histograms of image subareas is the basis of *feature extraction* in computer vision, where feature refers to patterns of interest in images. In practice, whenever there is a large volume of data that needs to be analyzed to distill interesting events (i.e., "Big Data"), histograms are likely used as a foundational computation. Credit card fraudulence detection and computer vision obviously meet this description. Other application domains with such needs include speech recognition, website purchase recommendations, and scientific data analysis such as correlating heavenly object movements in astrophysics.

```
1.  sequential_Histogram(char *data, int length, int *histo) {
2.      for (int i = 0; i < length; i++) {
3.          int alphabet_position = data[i] – 'a';
4.          if (alphabet_position >= 0 && alphabet_position < 26)
5.              histo[alphabet_position/4]++
6.          }
7.      }
8.  }
```

Figure 9.2 A simple C function for calculating histogram for an input text string.

Histograms can be easily computed in a sequential manner, as shown in Figure 9.2. For simplicity, the function is only required to recognize lower-case letters. The C code assumes that the input data set comes in a char array *data[]* and the histogram will be generated into the int array *histo[]* (Line 1). The number of data items is specified in function parameter *length*. The for loop (Line 2 through Line 4) sequentially traverses

the array, identifies the particular alphabet index into the *index* variable, and increments the *histo[index/4]* element associated with that interval. The calculation of the alphabet index relies on the fact that the input string is based on the standard ASCII code representation where the alphabet characters "a" through "z" are encoded in consecutive values according to the alphabet order.

Although one may not know the exact encoded value of each letter, one can assume that the encoded value of a letter is the encoded value of "a" plus the alphabet position difference between that letter and "a". In the input, each character is stored in its encoded value. Thus, the expression *data[i] – "a"* (Line 3) derives the alphabet position of the letter with the position of "a" being 0. If the position value is greater than or equal to 0 and less than 26, the data character is indeed a lower-case alphabet letter (Line 4). Keep in mind that we defined the intervals such that each interval contains four alphabet letters. Therefore, the interval index for the letter is its alphabet position value divided by 4. We use the interval index to increment the appropriate *histo[]* array element (Line 4).

The C code in Figure 9.2 is quite simple and efficient. The data array elements are accessed sequentially in the *for* loop so the CPU cache lines are well used whenever they are fetched from the system DRAM. The *histo[]* array is so small that it fits well in the level-one (L1) data cache of the CPU, which ensures very fast updates to the *histo[]* elements. For most modern CPUs, one can expect that execution speed of this code to be memory bound, i.e., limited by the rate at which the *data[]* elements can be brought from DRAM into the CPU cache.

9.2. Use of Atomic Operations

A straightforward approach to parallel histogram computation is dividing the input array into sections and have each thread to process one of the sections. If we use P threads, each thread would be doing approximately 1/P of the original work. We will refer to this approach as "Strategy I" in our discussions. Using this strategy, we should be able to expect a speedup close to P. Figure 9.3 illustrates this approach using our text example. To make the example fit in the picture, we reduce the input to the first 24 characters in the phrase. We assume that P = 4 and each thread processes a section of 6 characters. We show the workload of the four threads in Figure 9.3.

Each thread iterates through its assigned section and increment the appropriate interval counter for each character. Figure 9.3 shows the actions taken by the four threads in the

first iteration. Note that threads 0, 1, and 2 all need to update the same counter (m-p), which is a conflict referred to as output interference. One must understand the concepts of race conditions and atomic operations in order to confidently handle such output interferences in his/her parallel code.

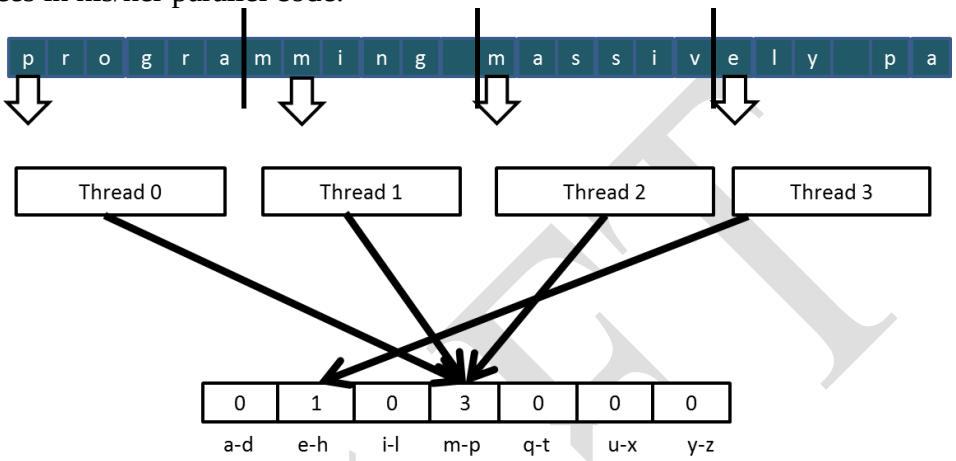

Figure 9.3 Strategy I for parallelizing histogram computation

An increment to an interval counter in the *histo[]* array is an update, or read-modify-write, operation on a memory location. The operation involves reading the memory location (read), adding one to the read content (modify), and writing the new value back to the memory location (write). Read-modify-write is a very common operation for coordinating collaborative activities.

For example, when we make a flight reservation with an airline, we bring up the seat map and look for available seats (read), we pick a seat to reserve (modify), and change the seat status to unavailable in the seat map(write). A bad potential scenario can happen as follows:

- Two customers simultaneously bring up seat map of the same flight.
- Both customers pick the same seat, say 9C.
- Both customers change the status of seat 9C to unavailable.

After the sequence, both customers thought that they have seat 9C. We can imagine that they will have an unpleasant situation when they board the flight and find out that one of them cannot take the reserved seat! Believe it or not, such unpleasant situation indeed happen in real life due to flaws in airline reservation software.

For another example, some stores allow customers to wait for service without standing in line. They ask each customer to take a number from one of the kiosks. There is a display that shows the number that will be served next. When a service agent becomes available, he/she asks the customer to present the ticket that matches the number, verify the ticket, and update the display number to the next higher number. Ideally, all customers will be served in the order they enter the store. An undesirable outcome would be that two customers simultaneously sign in at two kiosks and both receive tickets with the same number. When a service agent call for that number, both customers will feel that they are the one who should receive service.

In both examples, undesirable outcomes are caused by a phenomenon called *race condition*, where the outcome of two or more simultaneous update operations varies depending on the relative timing of the operations involved. Some outcomes are correct and some are incorrect. Figure 9.4 illustrates a race condition when two threads attempt to update the same *histo[]* element in our text histogram example. Each row in Figure 9.4 shows the activity during a time period, with time progressing from top to bottom.

Time	Thread 1	Thread 2
1	(0) Old ← histo[x]	
2	(1) New ← Old + 1	
3	(1) histo[x] ← New	
4		(1) Old ← histo[x]
5		(2) New ← Old + 1
6		(2) histo[x] ← New

(a)

Time	Thread 1	Thread 2
1	(0) Old ← histo[x]	
2	(1) New ← Old + 1	
3		(0) Old ← histo[x]
4	(1) histo[x] ← New	
5		(1) New ← Old + 1
6		(1) histo[x] ← New

(b)

Figure 9.4 Race condition in updating a *histo[]* array element.

Figure 9.4(a) depicts a scenario where Thread 1 completes all three parts of its read-modify-write sequence during time periods 1 through 3 before Thread 2 starts its sequence at time period 4. The value in the parenthesis in front of each operation shows the value being written into the destination, assuming the value of *histo[x]* was initially 0. Under this scenario, the value of *histo[x]* afterwards is 2, exactly what one would expect. That is, both threads successfully incremented the histo[x] element. The element value starts with 0 and becomes 2 after the operations complete.

In Figure 9.4(b), the read-modify-write sequences of the two threads overlap. Note that Thread 1 writes the new value into *histo[x]* at time period 4. When Thread 2 reads

histo[x] at time period 3, it still has the value 0. As a result, the new value it calculates and eventually writes to *histo[x]* is 1 rather than 2. The problem is that Thread 2 read *histo[x]* too early, before Thread 1 completes its update. The net outcome is that the value of *histo[x]* afterwards is 1, which is incorrect. The update by Thread 1 is lost.

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3		(1) histo[x] ← New
4	(1) Old ← histo[x]	
5	(2) New ← Old + 1	
6	(2) histo[x] ← New	

(a)

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3	(0) Old ← histo[x]	
4		(1) histo[x] ← New
5	(1) New ← Old + 1	
6	(1) histo[x] ← New	

(b)

Figure 9.5 Race condition scenarios where Thread 2 runs ahead of Thread 1.

During parallel execution, threads can run in any order relative to each other. In our example, Thread 2 can easily start its update sequence ahead of Thread 1. Figure 9.5 shows two such scenarios. In Figure 9.5(a), Thread 2 completes its update before Thread 1 starts its. In Figure 9.5(b), Thread 1 starts its update before Thread 2 completes its. It should be obvious that the sequences in 9.5(a) result in correct outcome for histo[x] but those in 9.5(b) produce incorrect outcome.

The fact that the final value of *histo[x]* varies depending on the relative timing of the operations involved indicates that there is a race condition. We can eliminate such variation by eliminating the possible interleaving of operation sequences of Thread 1 and Thread 2. That is, we would like to allow the timings shown in Figures 9.4(a) and 9.5(a) while eliminating the possibilities shown in Figures 9.4(b) and 9.5(b). This can be accomplished through the use of *atomic operations*.

An atomic operation on a memory location is an operation that performs a read-modify-write sequence on the memory location in such a way that no other read-modify-write sequence to the location can overlap with it. That is, the read, modify, and write parts of the operation form an undividable unit, thus the name atomic operation. In practice, atomic operations are realized with hardware support to lock out other operations to the same location until the current operation is complete. In our example, such support eliminates the possibility depicted in Figures 9.4(b) and 9.5(b) since the trailing thread cannot start its update sequence until the leading thread completes its.

It is important to remember that atomic operations do not enforce particular execution order between threads. In our example, both orders shown in Figure 9.4(a) and 9.5(b) are allowed by atomic operations. Thread 1 can run either ahead of or behind Thread 2. The rule being enforced is that if both threads perform atomic operations on the same memory location, the atomic operation performed by the trailing thread cannot be started until the atomic operation of the leading thread completes. This effectively serializes the atomic operations being performed on a memory location.

Atomic operations are usually named according to the modification performed on the memory location. In our text histogram example, we are adding a value to the memory location so the atomic operation is called atomic add. Other types of atomic operations include subtraction, increment, decrement, minimum, maximum, logical and, logical or, etc.

A CUDA program can perform an atomic add operation on a memory location through a function call:

```
int atomicAdd(int* address, int val);
```

The function is an intrinsic function that will be compiled into a hardware atomic operation instruction which reads the 32-bit word pointed to by the address argument in global or shared memory, adds val to the old content, and stores the result back to memory at the same address. The function returns the old value of the address.

Intrinsic Functions

Modern processors often offer special instructions that either perform critical functionality (such as the atomic operations) or substantial performance enhancement (such as vector instructions). These instructions are typically exposed to the programmers as intrinsic functions, or simply instrinsics. From the programmer's perspective, these are library functions. However, they are treated in a special way by compilers; each such call is translated into the corresponding special instruction. There is typically no function call in the final code, just the special instructions in line with the user code. All major modern compilers, such as Gnu C Compiler (gcc), Intel C Compiler and LLVM C Compiler support intrinsics.

Figure 9.6 shows a CUDA kernel that performs parallel histogram computation based on Strategy I. Line 1 calculates a global thread index for each thread. Line 2 divides the total amount of data in the buffer by the total number of threads to determine the number of characters to be processed by each thread. The ceiling formula, introduced in Chapter 2, is used to ensure that all contents of the input buffer are processed. Note that the last few threads will likely process a section that is only partially filled. For example, if we have 1000 characters in the input buffer and 256 threads, we would assign sections of (1000-1)/256 +1 = 4 elements to each of the first 250 threads. The last 6 threads will process empty sections.

Line 3 calculates the starting point of the section to be processed by each thread using the global thread index calculated in Line 1. In the example above, the starting point of the section to be processed by thread i would be i * 4 since each section consists of 4 elements. That is, the starting point of thread 0 is 0, thread 8 is 32, and so on.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.   int i = threadIdx.x + blockIdx.x * blockDim.x;
2.   int section_size = (size-1)  / (blockDim.x * gridDim.x) +1;
3.   int start = i*section_size;
     // All threads handle blockDim.x * gridDim.x  consecutive elements
4.   for (k = 0; k < section_size; k++) {
5.    if (start+k < size) {
6.      int alphabet_position = buffer[start+k] – ‘a’;
7.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
       }
     }
}
```

Figure 9.6 A CUDA kernel for calculation histogram based on Strategy I

The for-loop starting in line 4 is very similar to the one we have in Figure 9.2. This is because each thread essentially executes the sequential histogram computation on its assigned section. There are two noteworthy differences. First, the calculation of the alphabet position is guarded by an if-condition. This test ensures that only the threads whose index into the buffer is within bounds will access the buffer. It is to prevent the threads that receive partially filled or empty sections from making out-of-bound memory accesses.

Finally, the increment expression (histo[alphabet_position/4]++) in Figure 9.2 becomes an atomicAdd() function call in Line 6 of Figure 9.6. The address of the location to be updated, &(histo[alphabet_position/4]), is the first argument. The value to be added to the location, 1, is the second argument. This ensures that any simultaneous updates to any histo[] array element by different threads are properly serialized.

9.3. Block vs. Interleaved Partitioning

In Strategy I, we partition the elements of buffer[] into sections of continuous elements, or blocks, and assign each block to a thread. This partitioning strategy is often referred to as *block partitioning*. Partitioning data into continuous blocks is conceptually simple and intuitive. On a CPU, where parallel execution typically involve a small number of threads, block partitioning is often the best performing strategy since the sequential access pattern by each thread makes good use of cache lines. Since each CPU cache typically supports only a small number of threads, there is little interference in cache usage by different threads. The data in cache lines, once brought in for a thread, can be expected to remain for the subsequent accesses.

As we learned in Chapter 5, the large number of simultaneously active threads in an SM typically cause too much interference in the caches that one cannot expect a data in a cache line to remain available for all the sequential accesses by a thread under Strategy I. Rather, we need to make sure that threads in a warp access consecutive locations to enable memory coalescing. This means that we need to adjust our strategy for partitioning buffer[].

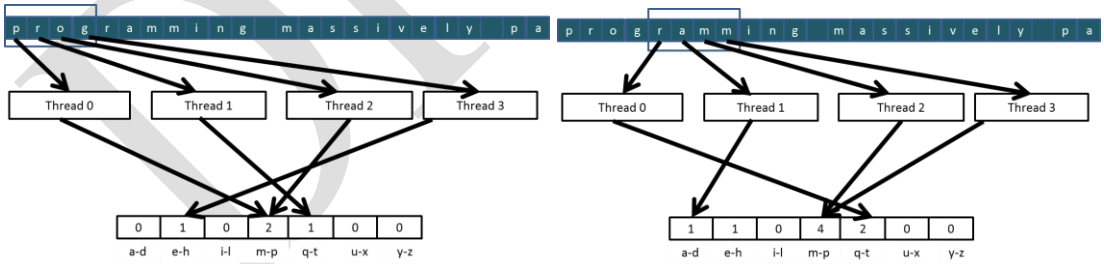

Figure 9.7 Desirable access pattern to the input buffer for memory coalescing – Strategy II.

Figure 9.7 shows the desirable access pattern for memory coalescing for our text histogram example. During the first iteration, the four threads access characters 0 through 3 ("prog"), as shown in Figure 11.7(a). With memory coalescing, all the

elements will be fetched with only one DRAM access. During the second iteration, the four threads access characters "ramm" in one coalesced memory access. Obviously, this is a toy example. In reality, there will be many more threads. There are also subtle issues such as each thread should process four characters in each iteration to fully utilize the bandwidth between the caches and the SMs.

Now that we understand the desired access pattern, we can derive the partitioning strategy to solve this problem. Instead of the block partitioning strategy, we will use an interleaved partitioning strategy where each thread will process elements that are separated by the elements processed by all threads during one iteration. In Figure 9.7, the partition to be processed by thread 0 would be elements 0 ("p"), 4 ("r"), 8 ("i"), 12 ("m"), 16 ("i"), and 20 ("y"). Thread 1 would process elements 1 ("r"), 5 ("a"), 9 ("n"), and 13 ("a"), 17 ("v"), and 21 ("_"). It should be clear why this is called interleaved partitioning: the partition to be processed by different threads are interleaved with each other.

Figure 9.8 shows a revised kernel based on Strategy II. It implements interleaved portioning in Line 2 by calculating a stride value, which is the total number threads launched during kernel invocation (blockDim.x*gridDim.x). In the first iteration of the while loop, each thread index the input buffer using its global thread index: Thread 0 accesses element 0, Thread 1 accesses element 1, Thread 2 accesses element 2, etc. Thus, all threads jointly process the first blockDim.x*gridDim.x elements of the input buffer. In the second iteration, all threads add blockDim.x*gridDim.x to their indices and jointly process the next section of blockDim.x*gridDim.x elements.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.  unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
// All threads handle blockDim.x * gridDim.x consecutive elements in each iteration
2.   for (unsigned int i = tid;  i < size; i += blockDim.x*gridDim.x ) {
3.      int alphabet_position = buffer[i] – 'a';
4.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1)
     }
}
```

Figure 9.8 A CUDA kernel for calculating histogram based on Strategy II.

The for-loop controls the iterations for each thread. When the index of a thread exceeds the valid range of the input buffer (i is greater than or equal to size), the thread has

completed processing its partition and will exit the loop. Since the size of the buffer may not be a multiple of the total number of threads, some of the threads may not participate in the processing of the last section. So some threads will execute one fewer for-loop iteration than others.

Thanks to the coalesced memory accesses, the version in Figure 9.8 will likely execute several times faster than that in Figure 9.6. However, there is still plenty of room for improvement, as we will show in the rest of this chapter. It is interesting that the code in Figure 9.8 is actually simpler even though interleaved partitioning is conceptually more complicated than block partitioning. This is often true in performance optimization. While an optimization may be conceptually complicated, its implementation can be quite simple.

9.4. Latency vs. Throughput of Atomic Operations

The atomic operation used in the kernels of Figures 9.6 and 9.8 ensures the correctness of updates by serializing any simultaneous updates to a location. As we all know, serializing any portion of a massively parallel program can drastically increase the execution time and reduce the execution speed of the program. Therefore, it is important that such serialized operations account for as little execution time as possible.

As we learned in Chapter 5, the access latency to data in DRAMs can take hundreds of clock cycles. In Chapter 3, we learned that GPUs use zero-cycle context switching to tolerate such latency. As long as we have many threads whose memory access latencies can overlap with each other, the execution speed is limited by the throughput of the memory system. Thus it is important that GPUs make full use DRAM bursts, banks, and channels to achieve very high memory access throughput.

It should be clear to the reader at this point that the key to high memory access throughput is the assumption that many DRAM accesses can be simultaneously in progress. Unfortunately, this assumption breaks down when many atomic operations update the same memory location. In this case, the read-modify-write sequence of a trailing thread cannot start until the read-modify-write sequence of a leading thread is complete. As shown in Figure 9.9, the execution of atomic operations to the same memory location is such that each one is the only one in progress. The duration of each atomic operation is approximately the latency of a memory read (left section of the atomic operation time) plus the latency of a memory write (right section of the atomic operation time). The length of these time sections of each read-modify-write operation,

usually hundreds of clock cycles, defines the minimal amount of time that must be dedicated to servicing each atomic operation and limits the throughput, or the rate at which atomic operations can be performed.

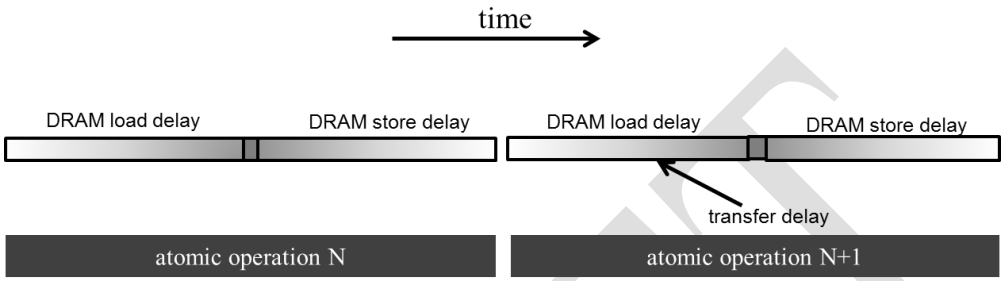

Figure 9.9 Throughput of atomic operation is determined by the memory access latency.

For example, assume a memory system with 64-bit Double Data Rate DRAM interface, 8 channels, 1 GHz clock frequency, and typical access latency of 200 cycles. The peak access throughput of the memory system is 8 (bytes/transfer)*2 (transfers per clock per channel)* 1G (clocks per second) * 8 (Channels) = 128 GB/sec. Assuming each data accessed is 4 bytes, the system has a peak access throughput of 32G data elements per second.

However, when performing atomic operations on a particular memory location, the highest throughput one can achieve is one atomic operation every 400 cycles (200 cycles for the read and 200 cycles for the write). This translates into a time-based throughput of 1/400 atomics/clock * 1G (clocks/second) = 2.5M atomics/second. This is dramatically lower than most users expect from a GPU memory system.

In practice, not all atomic operations will be performed on a single memory location. In our text histogram example, the histogram has 7 intervals. If the input characters are uniformly distributed in the alphabet, the atomic operations evenly distributed among the histo[] elements. This would boost the throughput to 7*2.5M = 17.5M atomic operations per second. In reality, the boost factor tends to be much lower than the number of intervals in the histogram because the characters tend to have biased distribution in the alphabet. For example, in Figure 9.1, we see that characters in the example phrase are heavily biased towards the m-p and q-t intervals. The heavy contention traffic to update these intervals will likely reduce the achievable throughput to much less than 17.5M atomic operations per second.

For the kernels of Figure 9.6 and Figure 9.8, low throughput of atomic operations will have significant negative impact on the execution speed. To put things into perspective, assume for simplicity that the achieved throughput of the atomic operations is 17.5M atomic operations per second. We see that the kernel in Figure 9.8 performs approximately six arithmetic operations (-, >=, <. /, +, +) with each atomic operation. Thus the maximal arithmetic execution throughput of the kernel will be 6*17.5M = 105M arithmetic operations per second. This is only a tiny fraction of the typical peak throughput of 1,000,000M or more arithmetic operations per second on modern GPUs! This type of insight has motivated several categories of optimizations to improve the speed of parallel histogram computation as well as other types of computation using atomic operations.

9.5. Atomic Operation in Cache Memory

A key insight from the previous section is that long latency of memory access translates into low throughput in executing atomic operations on heavily contended locations. With this insight, an obvious approach to improving the throughput of atomic operations is to reduce the access latency to the heavily contended locations. Cache memories are the primary tool for reducing memory access latency.

Recent GPUs allow atomic operation to be performed in the last level cache, which is shared among all SMs. During an atomic operation, if the updated variable is found in the last level cache, it is updated in the cache. If it cannot be found in the last level cache, it triggers a cache miss and is brought into the cache where it is updated. Since the variables updated by atomic operations tend to be heavily accessed by many threads, these variables tend to remain in the cache once they are brought in from DRAM. Since the access time to the last level cache is in tens of cycles rather than hundreds of cycles, the throughput of atomic operations are improved by at least an order of magnitude by just allowing them to be performed in the last level cache. This was evident in the big throughput improvement of atomic operations from the Tesla generation to the Fermi generation, where the atomic operations are first supported in the last level (L2) cache. However, the improved throughput is still insufficient for many applications.

9.6. Privatization

The latency for accessing memory can be dramatically reduced by placing data in the shared memory. Shared memory is private to each SM and has very short access latency (a few cycles). Recall that this reduced latency directly translates into increase throughput of atomic operations. The problem is that due to the private nature of shared

memory, the updates by threads in one thread block are no longer visible to threads in other blocks. The programmer must explicitly deal with this lack of visibility of histogram updates across thread blocks.

In general, a technique referred to as *privatization*, is commonly used to address the output interference problem in parallel computing. The idea is to replicate highly contended output data structures into private copies so that each thread (or each subset of threads) can update its private copy. The benefit is that the private copies can be accessed with much less contention and often at much lower latency. These private copies can dramatically increase the throughput for updating the data structures. The down side is that the private copies need to be merged into the original data structure after the computation completes. One must carefully balance between the level of contention and the merging cost. Therefore, in massively parallel systems, privatization is typically done for subsets of threads rather than individual threads.

In our text histogram example, we can create a private histogram for each thread block. Under this scheme, a few hundred threads would work on a copy of the histogram stored in short-latency shared memory, as opposed to tens of thousands of threads pounding on a histogram stored in medium latency second level cache or long latency DRAM. The combined effect of fewer contending threads and shorter access latency can result in orders of magnitude of increase in update throughput.

Figure 9.10 shows a privatized histogram kernel. Line 2 allocates a shared memory array bins_s[] whose dimension is set during kernel launch. In the for loop at Line 3, all threads in the thread block cooperatively initialize all the bins of their private copy of the histogram. The barrier synchronization in Line 5 ensures that all bins of the private histogram have been properly initialized before any thread starts to update them.

The for loop at Lines 6-7 is identical to that in Figure 9.8, except that the atomic operation is performed on the shared memory histo_s[]. The barrier synchronization in Line 8 ensures that all threads in the thread block complete their updates before merging the private copy into the original histogram.

Finally, the for loop at Lines 9-10 cooperatively merge the private histogram values into the original version. Note that atomic add operation is used to update the original histogram elements. This is because multiple thread blocks can simultaneously update the same histogram elements and must be properly serialized with atomic operations. Note that both for loops in Figure 9.10 are written so that the kernel can handle histograms of arbitrary number of bins.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
                    unsigned int num_elements, unsigned int num_bins) {
1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
        // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.      histo_s[binIdx] = 0u;
    }
5.  __syncthreads();
        // Histogram
6.  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
      int alphabet_position = buffer[i] – “a”;
7.    if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo_s[alphabet_position/4]), 1);
    }
8.  __syncthreads();
  // Commit to global memory
9.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
10.     atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

Figure 9.10 A privatized text histogram kernel.

9.7. Aggregation

Some data sets have a large concentration of identical data values in localized areas. For example, in pictures of the sky, there can be large patches of pixels of identical value. Such high concentration of identical values causes heavy contention and reduced throughput of parallel histogram computation.

For such data sets, a simple and yet effective optimization is for each thread to aggregate consecutive updates into a single update if they are updating the same element of the histogram [Merrill2015]. Such aggregation reduces the number of atomic operations to the highly contended histogram elements, thus improving the effective throughput of the computation.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
   unsigned int num_elements, unsigned int num_bins) {
1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
          // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.      histo_s[binIdx] = 0u;
    }
5.  __syncthreads();
6.  unsigned int prev_index = -1;
7.  unsigned int accumulator = 0;

8.  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
9.      int alphabet_position = buffer[i] – “a”;
10.     if (alphabet_position >= 0 && alpha_position < 26) {
11.        unsigned int curr_index = alphabet_position/4;
12.        if (curr_index != prev_index) {
13.           if (accumulator >= 0) atomicAdd(&(histo_s[alphabet_position/4]), accumulator);
14.           accumulator = 1;
15.           prev_index = curr_index;
           }
16.        else {
17.           accumulator++;
           }
        }
     }
18.  __syncthreads();
          // Commit to global memory
19.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
20.     atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

Figure 9.11 An aggregated text histogram kernel.

Figure 9.11 shows an aggregated text histogram kernel. Each thread declares three additional register variables curr_index, prev_index and accumulator. The accumulator keeps track of the number of updates aggregated thus far and prev_index tracks the index of the histogram element whose updates has been aggregated. Each thread initializes the prev_index to -1 (Line 6) so that no alphabet input will match it. The accumulator is initialized to zero (Line 7), indicating that no updates has been aggregated.

When an alphabet data is found, the thread compares the index of the histogram element to be updated with that being aggregated. If the index is different, the streak of aggregated updates to the histogram element has ended (Line 12). The thread uses atomic operation to add the accumulator value to the histogram element whose index is tracked by prev_index. This effectively flushes out the total contribution of the previous streak of aggregated updates. If the curr_index matches the prev_index, the thread simply adds one to the accumulator (Line 17), extending the streak of aggregated updates by one.

An observation is that the aggregated kernel requires more statements and variables. Thus, if the contention rate is low, an aggregated kernel may execute at lower speed than the simple kernel. However, if the data distribution leads to heavy contention in atomic operation execution, aggregation result in significant higher speed.

9.8. Summary

Histogramming is a very important computation for analyzing large data sets. It also represents an important class of parallel computation patterns where the output location of each thread is data-dependent, which makes it infeasible to apply owner-computes rule. It is therefore a natural vehicle for introducing the practical use of atomic operations that ensure the integrity of read-modify-write operations to the same memory location by multiple threads. Unfortunately, as we explained in this chapter, atomic operations have much lower throughput than simpler memory read or write operations because their throughput is approximately the inverse of two times the memory latency. Thus, in the presence of heavy contention, histogram computation can have surprisingly low computation throughput. Privatization is introduced as an important optimization technique that systematically reduces contention and enabled the use of local memory such as the shared memory which supports low latency and thus high throughput. In fact, supporting very fast atomic operations among threads in a block is a very important use case of the shared memory. For data sets that cause heavy contention, aggregation can also lead to significantly higher execution speed.

9.9. Exercises

1. Assume that each atomic operation in a DRAM system has a total latency of 100ns. What is the maximal throughput we can get for atomic operations on the same global memory variable?
   (A) 100G atomic operations per second

(B) 1G atomic operations per second
(C) 0.01G atomic operations per second
(D) 0.0001G atomic operations per second

2. For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4ns to complete in L2 cache and 100ns to complete in DRAM. Assume that 90% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?
   (A) 0.225G atomic operations per second
   (B) 2.75G atomic operations per second
   (C) 0.0735 atomic operations per second
   (D) 100G atomic operations per second

3. In question 1, assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 500 GFLOPS
   (B) 5 GFLOPS
   (C) 0.05 GFLOPS
   (D) 0.0005 GFLOPS

4. In Question 1, assume that we privatize the global memory variable into shared memory variables in the kernel and the shared memory access latency is 1ns. All original global memory atomic operations are converted into shared memory atomic operation. For simplicity, assume that the additional global memory atomic operations for accumulating privatized variable into the global variable adds 10% to the total execution time. Assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 4500 GFLOPS
   (B) 45 GFLOPS
   (C) 4.5 GFLOPS
   (D) 0.45 GFLOPS

5. To perform an atomic add operation to add the value of an integer variable Partial to a global memory integer variable Total. Which one of the following statement should be used?
   (A) atomicAdd(Total, 1);
   (B) atomicAdd(&Total, &Partial);
   (C) atomicAdd(Total, &Partial);
   (D) atomicAdd(&Total, Partial);

References

Duane Merrill, , "***Using compression to improve the performance response of parallel histogram computation,"*** NVIDIA Research Technical Report, 2015.