CPE810 – GPU And Multicore Programming

Lab Report # 2: Homework 2

Histogram

Instructor: Zhou Feng

TA: Ali Aghdaei

James Fong

I pledge my honor that I have abided by the Stevens Honor System

Table of Contents

Objective

Implement a histogram routine using atomic operations and shared memory in CUDA. Your code should be able handle arbitrary input vector sizes (vector values should be randomly generated using integers between 0~1023).

Required Functions and Implementation Details

1. Allocate Device Memory

```
//Sample Code
printf("Initializing data...\n");
printf("...allocating CPU memory.\n");
h_Data         = (uchar *)malloc(byteCount);
h_HistogramCPU = (uint *)malloc(HISTOGRAM256_BIN_COUNT * sizeof(uint));
h_HistogramGPU = (uint *)malloc(HISTOGRAM256_BIN_COUNT * sizeof(uint));

//Submission
//Allocating memory for Host
cudaMallocHost(&h_HistogramGPU_input, bytes);
cudaMallocHost(&h_HistogramGPU_bins, bytes_bins);

//Allocating memory for Device
cudaMalloc(&d_input, bytes);
cudaMalloc(&d_bins, bytes_bins);

//Allocating memory for CPU
cudaMallocHost(&h_HistogramCPU_input, bytes);
       cudaMallocHost(&h_HistogramCPU_bins, bytes_bins);
```

2. Copy Host Memory to Device

```
//Sample Code
printf("...allocating GPU memory and copying input data\n\n");
checkCudaErrors(cudaMalloc((void **)&d_Data, byteCount));
checkCudaErrors(cudaMalloc((void **)&d_Histogram, HISTOGRAM256_BIN_COUNT *
sizeof(uint)));

checkCudaErrors(cudaMemcpy(d_Data, h_Data, byteCount, cudaMemcpyHostToDevice));

//Submission
// CudaMemcpy from Host to Device
cudaMemcpy(d_input, h_HistogramGPU_input, bytes, cudaMemcpyHostToDevice);
       cudaMemcpy(d_bins, h_HistogramGPU_bins, bytes_bins,
cudaMemcpyHostToDevice);
```

3. Initialize Thread Block and Kernel Grid Dimensions

```
//Sample Code
printf("...generating input data\n");
srand(2009);
for (uint i = 0; i < byteCount; i++)
{
h_Data[i] = rand() % 256;
}

       //Submission
//Initialize an array with random numbers, 1023 being the maximum
void init_array(int *input, int N, int MAX) {
```

```
	for (int i = 0; i < N; i++) {
		input[i] = rand() % MAX;
	}
	}

	// Set the max value for data as 1023 as instructed in HW
	int MAX = 1023;

	// Initialize our input data, pulled from CoffeeBeforeArch Example Histogram
Shared Mem
	init_array(h_HistogramGPU_input, N, MAX); //init_array(input, N, MAX);

	// Set the divisor, pulled from CoffeeBeforeArch Example Histogram Shared Mem
	int DIV = (MAX + N_bins - 1) / N_bins;

	// Set bins = 0 for first run instance
	for (int i = 0; i < N_bins; i++) {
		h_HistogramGPU_bins[i] = 0;
	}

	// Set the Cooperative Thread Array(thread blocks) and Grid Dimensions, pulled
from CoffeeBeforeArch Example Histogram Shared Mem
	int THREADS = 512;
	int BLOCKS = (N + THREADS - 1) / THREADS;

	// Setting size of dynamically allocated shared memory, pulled from
CoffeeBeforeArch Example Histogram Shared Mem
		size_t SHMEM = N_bins * sizeof(int);
```

4. Invoke CUDA Kernel

```
	histogram <<<BLOCKS, THREADS, SHMEM>>>(d_input, d_bins, N, N_bins, DIV);
```

5. Copy Results from Device to Host

```
	// CudaMemcpy from Device to Host
	cudaMemcpy(h_HistogramGPU_input, d_input, bytes, cudaMemcpyDeviceToHost);
	cudaMemcpy(h_HistogramGPU_bins, d_bins, bytes_bins, cudaMemcpyDeviceToHost);
```

6. Deallocate Device Memory

```
	// Closing CUDA and freeing memory
	printf("Shutting down...\n");
	sdkDeleteTimer(&hTimer);

	cudaFree(d_input);
	cudaFree(d_bins);
	cudaFreeHost(h_HistogramGPU_input);
	cudaFreeHost(h_HistogramGPU_bins);
	cudaFreeHost(h_HistogramCPU_input);
	cudaFreeHost(h_HistogramCPU_bins);
```

7. Implement the routine using atomic operations and shared memory

```
// Histogram calculation pulled from CoffeeBeforeArch located here:
https://www.youtube.com/watch?v=Bwv5J7dHYjU

// Git page located here:
https://github.com/CoffeeBeforeArch/cuda_programming/blob/master/04_histogram/shmem_atomic/histogram.cu

__global__ void histogram(int *d_input, int *d_bins, int N, int N_bins, int DIV) {
	// Allocate shared memory
	extern __shared__ int s_bins[];

	// Calculate a global thread ID
	int tid = blockIdx.x * blockDim.x + threadIdx.x;

	// Initialize our shared memory
	if (threadIdx.x < N_bins)
		s_bins[threadIdx.x] = 0; // every thread block will have its own version of s_bins

	// Synchronize; wait for threads to zero out shared memory
	__syncthreads(); //need to make sure no thread starts to accumulate in s_bins before it 0's out

	// Range check
	if (tid < N) {
		int bin = d_input[tid] / DIV;
		atomicAdd(&s_bins[bin], 1);
	}

	__syncthreads(); // make sure eveyrone has done their atomic adds, everyone in thread block has their shared memory, before finish binning their elements

	// Write back our partial results to main memory
	if (threadIdx.x < N_bins)
		atomicAdd(&d_bins[threadIdx.x], s_bins[threadIdx.x]);

}
```

8. Handle thread divergence when dealing with arbitrary input sizes

```
	// Compares Host data with CPU data if there is an error; pulled from example histogram main.cpp
	printf("Beginning Histogram data comparison with Host and Device: \n\n");

	for (unsigned int i = 0; i < Histogram_Bin_Count; i++)
		if (h_HistogramGPU_input[i] != h_HistogramCPU_input[i]) {
			printf("Failure; histogram comparison failed. Exiting program.");
			return 0;
		}

	printf("Histogram data comparison passed. \n\n");
```

```
cudaDeviceSynchronize();
```

Questions

1. Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance. Which optimizations gave the most benefit?

An alternative optimization that I had attempted to integrate into the assignment that could be a potential optimization is to attempt an aggregate histogram, which performs a histogram calculation given a table, column, and interval function. An input parameter is used to produce bins in the same size. The result of the computation would be over the records falling within each bin. The start value for each bin would be inclusive but end as exclusive. The returned value in each bin would be the number of records inside of it [1]. The advantage of this method would be able to handle cases with many similarities between elements, you can begin aggregating the number of elements that are the same without the need of an atomic operation.

However, the issue with this implementation is that the total number of bins that are requested cannot exceed 10,000; whereas the current implementation can support a greater amount with various types of optimizations. In addition, our histogram is set to receive data that is completely randomized; while in certain samples this would be a more effective method, it would be a more inconsistent criteria as aggregation relies on similar data types. Finally, the assignment requests that we use atomic adds as a key function of our program; thus, I did not fully commit to this optimization.

2. How many global memory reads (write) per input element are being performed by your kernel? Explain.

The global memory reads, per input element, are performed after every int bin for each element, and global memory writes per input element are being done after writing to d_bins once per element. This is apparent in our atomicAdd functions:

```
	// Initialize our shared memory
	if (threadIdx.x < N_bins)
		s_bins[threadIdx.x] = 0; // every thread block will have its own version of s_bins
```

```
	// Synchronize; wait for threads to zero out shared memory
	__syncthreads(); //need to make sure no thread starts to accumulate in s_bins
before it 0's out

	// Range check
	if (tid < N) {
		int bin = d_input[tid] / DIV;
		atomicAdd(&s_bins[bin], 1);
	}

	__syncthreads(); // make sure eveyrone has done their atomic adds, everyone in
thread block has their shared memory, before finish binning their elements

	// Write back our partial results to main memory
	if (threadIdx.x < N_bins)

		atomicAdd(&d_bins[threadIdx.x], s_bins[threadIdx.x]);
```

3. How many atomic operations are being performed? Explain.

   There are two separate atomic operations are being performed.

   The first atomic add is done once per element;

```
	// Range check
	if (tid < N) {
		int bin = d_input[tid] / DIV;
		atomicAdd(&s_bins[bin], 1);

	}
```

   The second atomic add is done once per bin;

```
	// Write back our partial results to main memory
	if (threadIdx.x < N_bins)

		atomicAdd(&d_bins[threadIdx.x], s_bins[threadIdx.x]);
```

4. How many contentions would occur if every element in the array has the same value? What if every element has a random value? Explain.

Contention is simply when two threads try to access either the same resource or related resources in such a way that at least one of the contending threads runs more slowly than it would if the other thread(s) were not running [2].

If every element in the array had the same value, contentions would occur depending on the array size, and therefore the number of blocks you have, and how they're divided. When every element in the array has the same value, they would all be trying to access the same resource and would require either waiting for the resource to be released from another thread, or multiply and iterate to the number of bits taken per bin. [3] Thus, performance degrades due to contention and will loop while the thread that acquired the lock that is trying to be accessed by all other threads [4].

The current function is set up to utilize shared memory by duplicating the data structure and having a local copy of bins, to locally accumulate the values for a particular set of threads and updates each thread locally in shared memory. Once finished, they can perform a single write out to main memory; thus competing for memory only within the accessed thread block instead of all threads within the system.

If every element has a random value, it would depend on the distribution of random values, but would face less contention as they are not all trying to access the same lock (unless some of the random values are the same). Ideally, assuming infinite blocks with an infinite array size, every element will be perfectly parallelized and run without contention.

5. How would the performance (GLFOPS) change when sweeping <BinNum> from 4 to 256 (k=2 to 8)? Compare your predicted results with the realistic measurements when using different thread block sizes.

A parameter sweep is an iterative process in which simulations are run repeatedly using different values of the parameter(s) of choice. Thus, when sweeping the bin number from 4 to 256, the smaller bin size would result in a larger number of contentions and therefore result in a worse performance and take longer to compute. Having too few bins would not display an accurate picture of data, and too many bins would not represent the data distribution correctly.

I have compared my predicted results while using different thread sizes. I find that a smaller thread size leads to a higher need of data transfer rate, while transferring the same number of bytes and elements. The number of Atomic Operations remain the same. Comparing Table 1 with Table 2, Table 1 uses a lower amount of data transfer on average, while Table 2

uses a higher amount of data transfer for the same amount of time. This can be directly compared on graph 1 and graph 2.

According to my recordings, the smaller bin sizes lead to a larger amount of contentions, and resulted in a worse performance.

Larger blocks ended up with more serialization, thus needing less data to be transferred in the same amount of time, while a smaller block size resulted in a higher memory usage.

6. Propose a scheme for handling extremely large data set that can not be fully stored in a single GPU's device memory. (Hint: how to design an implementation for efficiently leveraging multiple GPUs for parallel histogram computation?)

A scheme for handling extremely large data that cannot be fully stored into a single GPU's device memory would likely require larger amounts of parallelization for testing out various schemas.

You would more than likely first create an array that points into several other arrays that can be computed. After separating the alternative arrays, you would compute each one separately utilizing atomic add functions into the kernel and pass them into separate GPUs. Each separate GPU would calculate each GPU through the same atomic add functions. Each would be parallelized on the block level. Finally, after multiple computations, return each computation together from Device memory back into the host memory, and deliver the expected result.

It would need a direct comparison with a single GPU calculation as to see which method is more efficient, as to analyze whether this scheme or a more optimized method utilizing large data handing would be more efficient.

Experimentation

Experimentation can be found on tables 1 and 2, with explicit comparison highlights on tables 3 and 4, which are condensed versions of the same tables.

Varying the number of elements from 2^10 resulted in altering the amount of atomic operations and had dramatically decreased the amount of data being passed, and amount of time for the operation to complete, making it difficult to compare data. Thus, for the lab report submission, I utilized the data for 2^20 for more usable data.

Ultimately, comparing data from the 2^10 total element data with its 1024 thread size with the 512 thread size led to the same conclusion I had arrived at from question 5. It is amount the same result, except it is much more difficult to compare due to the significantly smaller amount of elements. Thus, having a high enough threshold for experimental data with element data is important for understanding how different altercations affect the end result. However, comparing Tables 3 and table 4 better identifies that a smaller thread size lead to a smaller amount of Atomic Operations being performed, thus a higher thread count resulting in a faster computation time.

Alternatively, other methods of experimentation is to continue to decrease the thread size even further (from 512 to 256), which would allow better observation to view contention take place during calculation. Further increasing the bin size would result in higher amounts of Atomic Operations

GPU time and Data Transfer are directly dependent on the amount of elements being distributed in each bin, and thus are calculated around the same amount of time but with varying Data Transfer rates (which also depend on GPU boosting).

Conclusion

I used a large amount of my time identifying why the requested Histogram method of utilizing shared memory and dynamic allocation is more efficient than using the CPU for calculation. Using Shared memory and dynamic allocation allowed better parallelization and performance metrics; more importantly, it accounted for several different race conditions and accounted for a significant more amount of data loss.

Larger Histogram Bin Sizes is directly correlated with the number of elements when accounting for the total amount of Atomic Operations. A larger element size affects the total amount of Data Transfer, and is more effective through GPU boosting, and the GPU capability by CUDA compute capability. A larger amount of Data Transfer affects the amount of GPU time, and how much is needed to be calculated.

Further conclusions for Thread operation, amount of Atomic Operations, and efficiency optimizations can be found on in their respecting questions located in the Questions section of this report.

Thanks to Kerim Karabacak and Matthew Lepis for further help, as many of the questions and conclusions were found with them.

Tables

Table 1:

Histogram Bin Size	Number of Elements	GPU Time (avg) (sec)	Data Transfer (MB/sec)	Size (Bytes)	Atomic Operations	Threads
4	1048576	0.00025	17051.749	4194304	5238784	1024
8	1048576	0.00024	17470.3522	4194304	9428992	1024
16	1048576	0.00028	14911.7552	4194304	17809408	1024
32	1048576	0.00028	15227.098	4194304	34570240	1024
64	1048576	0.00023	18200.4943	4194304	68091904	1024
128	1048576	0.00023	18301.7218	4194304	135135232	1024
256	1048576	0.00025	16543.9471	4194304	269221888	1024

Table 2:

Histogram Bin Size	Number of Elements	GPU Time (avg) (sec)	Data Transfer (MB/sec)	Size (Bytes)	Atomic Operations	Threads
4	1048576	0.00025	17021.9054	4194304	5240832	512
8	1048576	0.00023	18051.1779	4194304	9433088	512
16	1048576	0.00023	18404.6465	4194304	17817600	512
32	1048576	0.00022	19156.994	4194304	34586624	512
64	1048576	0.00022	19252.0694	4194304	68124672	512
128	1048576	0.00027	15650.3877	4194304	135200768	512
256	1048576	0.00022	18756.4955	4194304	269352960	512

Table 3:

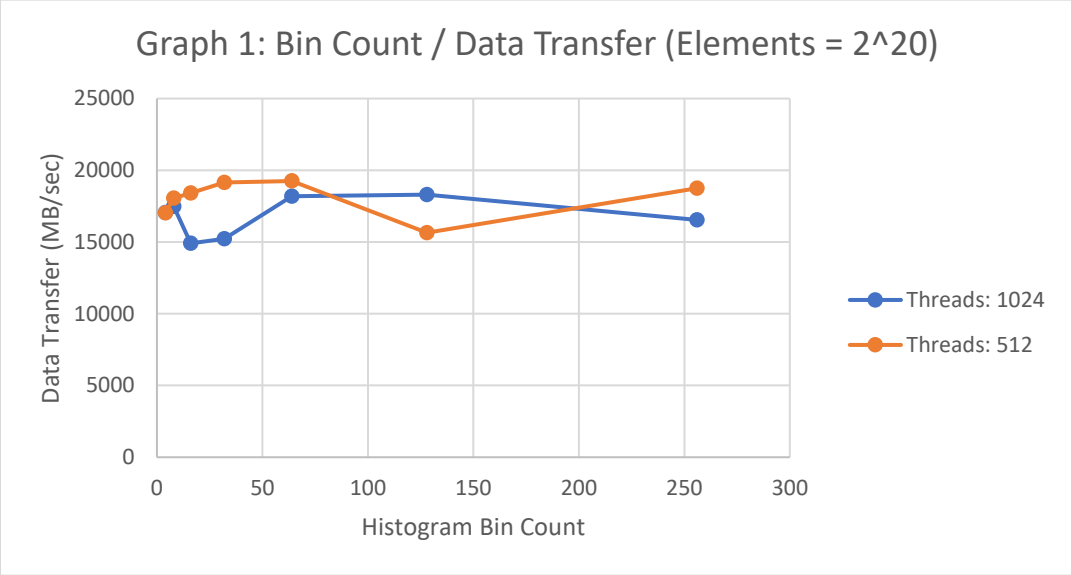
Histogram Bin Size	Number of Elements	GPU Time (avg) (sec)	Data Transfer (MB/sec)	Size (Bytes)	Atomic Operations	Threads
4	1024	0.00002	193.5499	4096	1024	1024
8	1024	0.00001	284.1977	4096	1024	1024
16	1024	0.00002	191.5137	4096	1024	1024
32	1024	0.00001	290.1107	4096	1024	1024
64	1024	0.00002	253.1325	4096	1024	1024
128	1024	0.00002	254.5087	4096	1024	1024
256	1024	0.00001	286.5588	4096	1024	1024

Table 4:

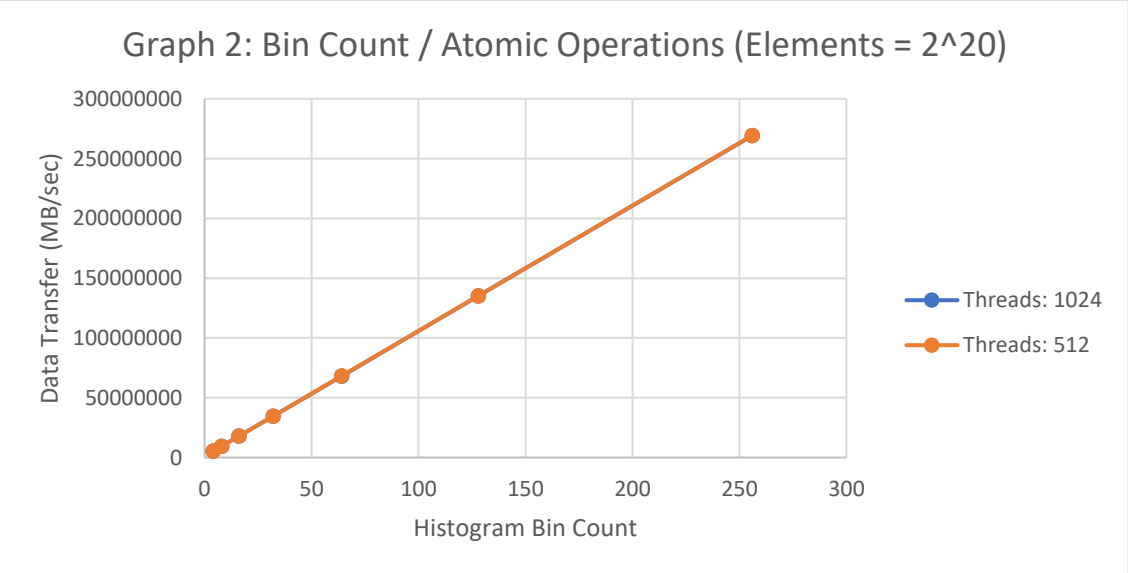
Histogram Bin Size	Number of Elements	GPU Time (avg) (sec)	Data Transfer (MB/sec)	Size (Bytes)	Atomic Operations	Threads
4	1024	0.00002	211.134	4096	3072	512
8	1024	0.00002	255.2025	4096	5120	512
16	1024	0.00001	314.4722	4096	9216	512
32	1024	0.00001	282.8485	4096	17408	512
64	1024	0.00001	319.0652	4096	33792	512
128	1024	0.00001	314.4722	4096	66560	512
256	1024	0.00002	264.7919	4096	132096	512

Graphs

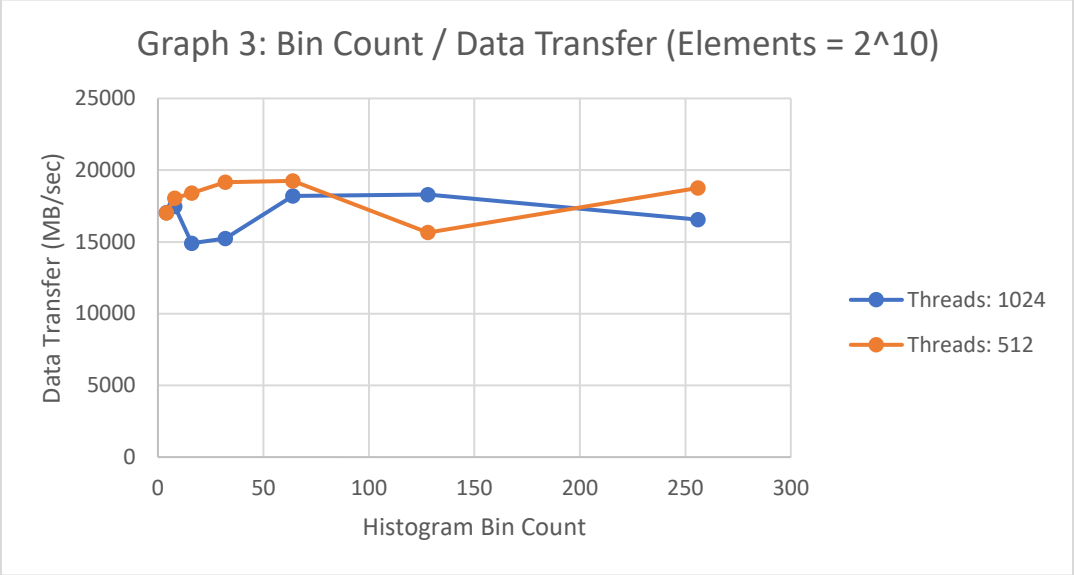
Graph 1:



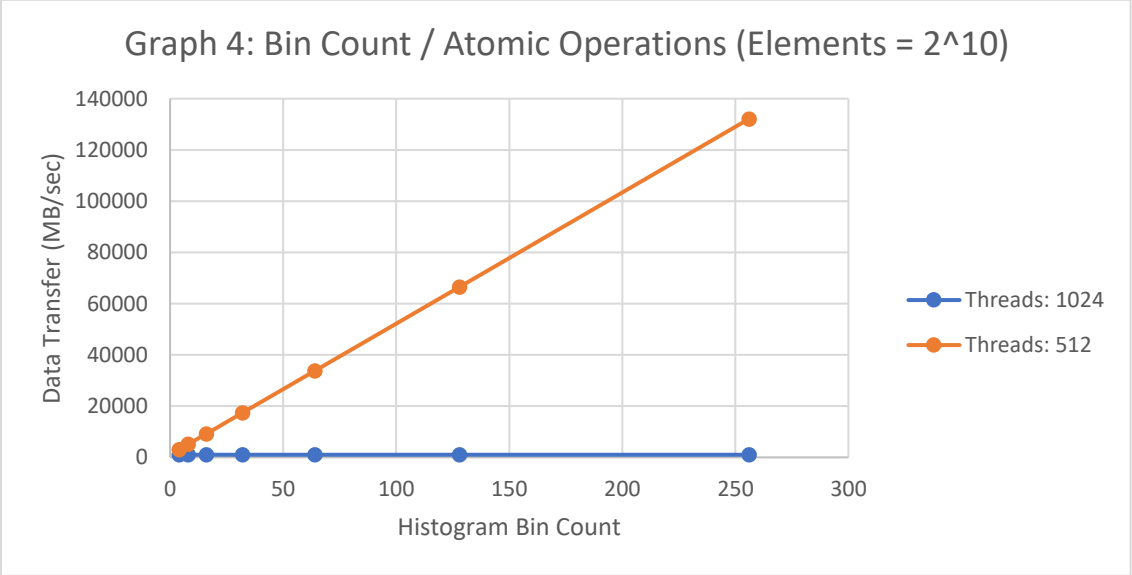
Graph 2:



Graph 3:



Graph 4:



References

[1] https://docs.kinetica.com/7.1/feature_overview/aggregate_histogram_feature_overview/

[2] https://stackoverflow.com/questions/1970345/what-is-thread-contention

[3] https://web.archive.org/web/20141127084626/http://www.cudahandbook.com/uploads/Chapter_8._Streaming_Multiprocessors.pdf

[4] https://www.informit.com/articles/article.aspx?p=2143393&seqNum=2