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).

Instructions

Make a new folder for your project. Copy and modify the code of histogram in CUDA sample code to include the following key functions:

1. Allocate device memory
2. Copy host memory to device
3. Initialize thread block and kernel grid dimensions
4. Invoke CUDA kernel
5. Copy results from device to host
6. Deallocate device memory
7. Implement the routine using atomic operations and shared memory
8. Handle thread divergence when dealing with arbitrary input sizes

Preface

My implementation was written inspired from the CUDA histogram64 sample, as well as from course-external sources of which will be cited explicitly for reference.

Code Execution Guide

My code can be ran via two methods. Command Line argument following:

./HW2-Histogram.exe <BinNum> <VecDim> <ThreadsPerBlock>

<BinNum>	Number of bins for the Histogram. Between 2 and 8, inclusive. Where that value is 'k' in: 2^k, is valid.
<VecDim>	Number of elements in the input array. Greater than 0 is valid
<ThreadsPerBlock>	Number of threads in a block. Between 256 and 1024, inclusive, is valid.

Or via console user prompt. To access this, exclude command line arguments when running the executable. The prompts will guide your operation.

I have included an instruction checklist which allows for quick indexing of my code to ensure I followed the instructions outlined on the previous page.

Instruction Checklist		
	Line of Occurrence	Function/Kernel/Definition
1	101	`cudaMalloc(&input_D, bytes);…`
2	122	`cudaMemcpy(input_D, input_H, bytes, cudaMemcpyHostToDevice);…`
3	115 & 116	`int THREADS = thread_count;…`
4	135	`histogramShared <<<BLOCKS, THREADS, bytes_sharedMem>>> (input_D, bins_D, numElements, numBins, DIV);`
5	136	`cudaMemcpy(bins_H, bins_D, bytes_bins, cudaMemcpyDeviceToHost);`
6	193	`cudaFree(input_D);…`
7	49	`atomicAdd(&shared_bins[bin], 1);`
8	139	`__syncthreads();`

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?

   Optimizations I propose:

   1. Depending on ‘nvcc’ compile aggression, variables may be stored in global device memory, explicitly pushing these variables to shared or constant memory could see an improvement. Particularly and of the input variables for the kernel call, or the internally declared variables.
   2. If the size of the input array is a known variable, I may be able to optimize the block thread count to ensure alignment and a cleaner execution.

   I was unable to commit the time necessary to implement these optimizations; aka I ran out of time in my week.

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

“

```
__global__ void histogramShared(int* input, int* bins, int numElements, int numBins, int
DIV)
{
	…

	if (tid < numElements)
	{
		int bin = input[tid] / DIV; // DIV is the mapping value to the bins.
		atomicAdd(&shared_bins[bin], 1);
	}

	…

}
```

“

Above is a snippet of my kernel implementation. It shows the relevant code for element wise calculation. The rest of the global memory accesses depend on number of bins in the end histogram rather than elements in the input array.

If it is to be assumed ‘input’, ‘bins’, ‘numElements’, ‘numBins’, and ‘DIV’ are stored in global device memory, as well as every variable declaration is in device global memory, then, there are:

Global Writes: 1 per element

- One for assigning a value to ‘int bin’

Global Reads: 5 per element

- One for accessing the value of ‘tid’
- One for accessing the value of ‘numElements’
- Two for accessing the value of ‘input[tid]’
- One for accessing the value of ‘DIV’

** If the cuda compiler is smart, some of these variables would be stored on register and reduce these global accesses.

---

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

This is the formula for the number of atomic operations given several variables.

```
AtomicOpsTotal = numElements + (((BLOCKS - 1) * THREADS) + overflowThreads) * numBins
```

Where:

Variable Names	Definition
`BLOCKS`	Number of Blocks called for in the kernel call
`numBins`	Number of Bins in the end histogram
`numElements`	Number of Elements in the input array
`overflowThreads`	Number of Threads in the last block (numElements % THREADS)
`THREADS`	Number of Threads per Block called for in the kernel call

‘numElements’ represents the first if statement in my kernel that includes an atomic operation, this if statement will occur ‘numElement’ times. The statements after that first plus sign represent the second if statement that includes an atomic operation, it relies on the number of block executing and how many of those block’s threads are computing that if statement. That is equal to “`(((BLOCKS - 1) * THREADS) + overflowThreads) * numBins`”.

---

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.

The number of contentions for an array with all equivalent values is equal to the number of elements minus one. As each atomic operation would form a contention, and effectively serialize the process.

In the case of every element having a random value: if we assume and ideal case where the input array is of infinite size, with evenly distributed infinite integers, as well as infinite blocks, the histogram can be perfectly parallelized. In essence providing zero contention. (That or the contentions would be congruent to the “assumed” gaussian distribution of the number set).

5. How would the performance (GFLOPS) 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.

As the operations are not floating point operation in this kernel, I have forgone the GFLOPS performance metric for something seemingly more applicable. I have chosen to measure the performance of my kernel implementation by GAOPS or GigaAtomic Operations per Second.

```
int overflowThreads = numElements % THREADS;
int AtomicOps = numElements + ((((BLOCKS - 1) * THREADS) + overflowThreads) * numBins);
double GigaAtomicOpsPerSec = (double)AtomicOps / (deviceAvgSecs * 1000000000);
```

Using the above code snippet I was able to calculate the GAOPS for my kernel.

My predictions are that as we increase the <BinNum> performance will be better, following my logic for question 4.
Using conditions:

Input array size:	10,000
Threads per Block:	512

K = 2:
GAOPS: 15.8999
K = 3:
GAOPS: 28.8154
K = 4:
GAOPS: 49.5771
K = 5:
GAOPS: 92.1187
K = 6:
GAOPS: 184.747
K = 7:
GAOPS: 371.579
K = 8:
GAOPS: 751.975

My prediction match my performance examination.

---

6. Propose a scheme for handling extremely large data set that cannot 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?)

I propose distributing your input data across multiple CUDA devices. To do this you must split your input data into chunks that are appropriately sized to fill your CUDA devices maximum memory. Each device will compute a partial histogram much like how my current implementation does for shared to global device memory, however in this multi-device setup, you must transfer each partial histogram from the individual device memory to the host memory. After all the CUDA devices complete their operations, you can piece the partial histograms into one large one in the end either on CPU or a given GPU.

Experimentation

For experimentation, I gathered a set of 6 different execution configurations.

Execution Configurations	Number of Bins	Number of Elements	Threads per Block
1	16	10,000,000	256
2	256	10,000,000	256
3	16	10,000,000	512
4	256	10,000,000	512
5	16	10,000,000	1024
6	256	10,000,000	1024

Here are my default results table:

Number of Bins	Number of Elements	Threads per Block	GAOPS	Execution Time (ms)
16	10,000,000	256	244.772	0.2
256	10,000,000	256	493.579	0.18
16	10,000,000	512	239.204	0.21
256	10,000,000	512	482.599	0.19
16	10,000,000	1024	209.731	0.24
256	10,000,000	1024	422.148	0.21

Table sorted by most GAOPS:

Number of Bins	Number of Elements	Threads per Block	GAOPS	Execution Time (ms)
256	10,000,000	256	493.579	0.18
256	10,000,000	512	482.599	0.19
256	10,000,000	1024	422.148	0.21
16	10,000,000	256	244.772	0.2
16	10,000,000	512	239.204	0.21
16	10,000,000	1024	209.731	0.24

Here it is clear that the increased bin amount has large improvement in operations per second. Also note that when the number of threads per block match the number of bins in the final histogram, we find the most performant execution configuration. Notice that the execution time is also roughly sorted here, that is a coincidence, but it makes sense.

Conclusion

Finally, creating my own GPU histogram implementation which uses atomic operations and utilizes shared memory transfer has expanded my proficiency in CUDA. Avoiding data races with atomic operations comes easily to me after this exercise. As progress from the last assignment I believe my understand for parallel algorithms has increased. From my experimentation, it is rather clear that being able to match the size of your data to you thread per block count will commonly provide a optimal execution.