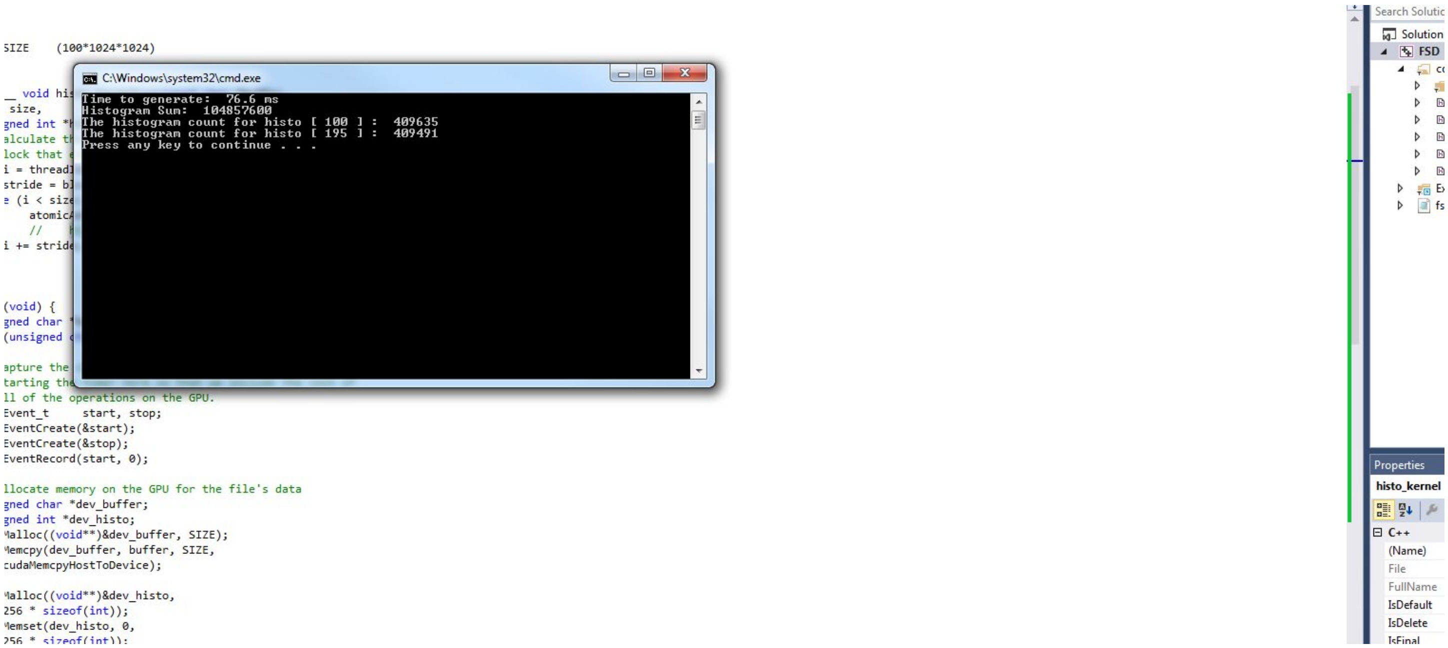
CSC8014 Topics: GPU Programming Spring 2017

Homework #5

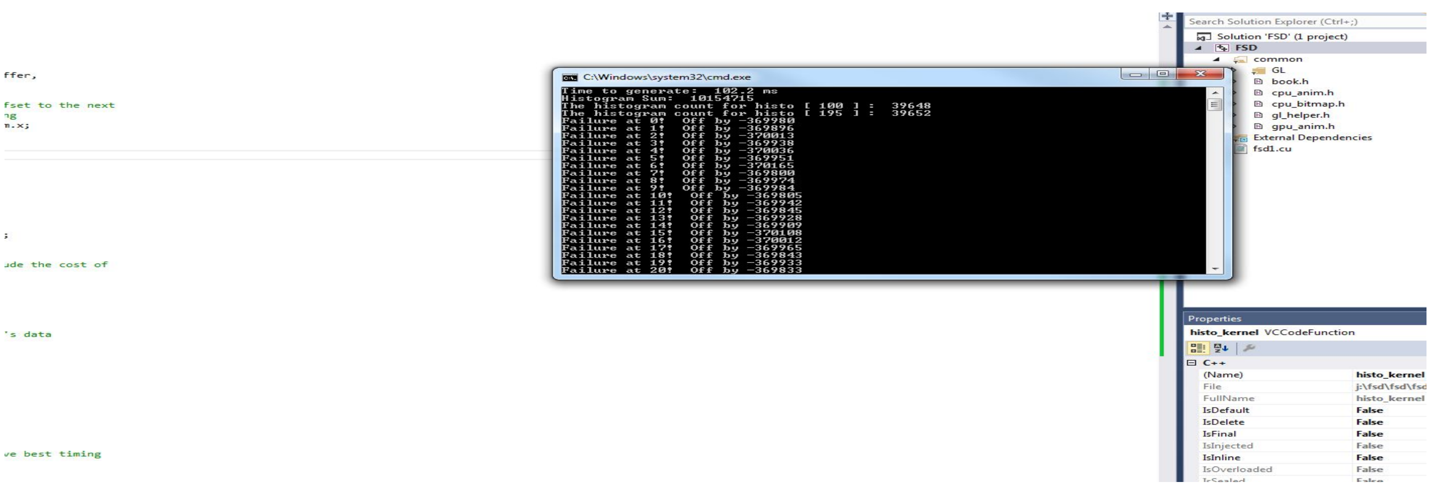
1. Compile and run the attached CUDA code hist_gpu_gmem_atomics.cu of histogram. Record the outcome. Update kernel function by replacing atomicAdd(&histo[buffer[i]], i) with histo[buffer[i]]++. And then re-compile and run the code. Answer the following questions:

    a. Are the outcomes (before and after the update) different? Why?

Answer : The outcome after using atomicAdd(&histo[buffer[i]], i) :



The outcome after using histo[buffer[i]]++ :



The sum differs a lot 104857600 to 10154715 when we change atomicAdd(&histo[buffer[i]], i) to histo[buffer[i]]++. This difference is created because X++ may not give the right value of X once its run if it is using multithreading. Since X may be being used by some other thread which can cause the result to differ from what we might expect. In this case we are using 100mb which comes to 104857600 bytes so we expect the sum to be 104857600 which we can get using atomicAdd(&histo[buffer[i]], i) but histo[buffer[i]]++ gives us an undesired result.

b. What does function atomicAdd(&histo[buffer[i]], i) do?

This line walks through the input array and incrementing the histogram bin. The call atomicAdd(addr, y); generates an atomic sequence of operations that read the value at address addr, adds y to that value, and stores the result back to the memory address addr. The hardware guarantees us that no other thread can read or write the value at address addr while we perform these operations, thus ensuring predictable results. In our example, the address in question is the location of the histogram bin that corresponds to the current byte. If the current byte is buffer[i], just like we saw in the CPU version, the corresponding histogram bin is histo[buffer[i]]. The atomic operation needs the address of this bin, so the rst argument is therefore &(histo[buffer[i]]). Since we simply want to increment the value in that bin by one, the second argument is 1.

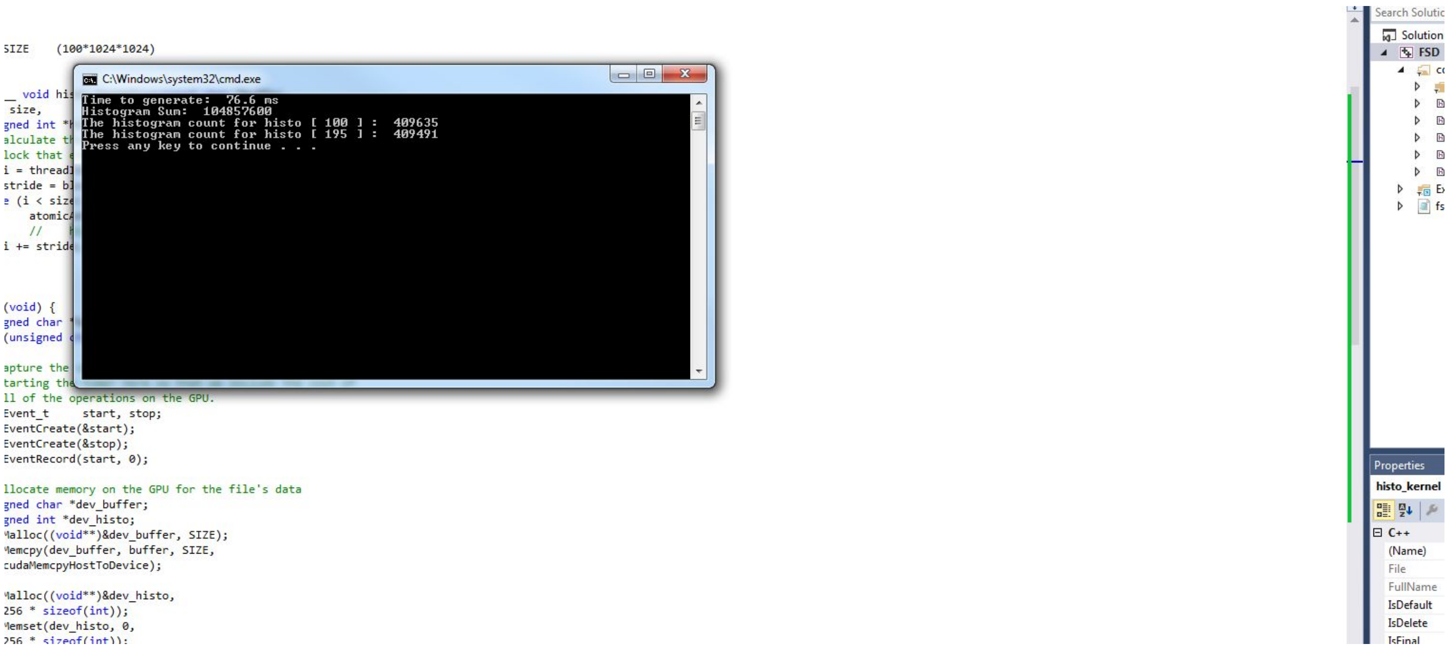
c. What's difference between atomicAdd(&histo[buffer[i]], i) and histo[buffer[i]]++?

When we use atomicAdd(&histo[buffer[i]], i) the hardware guarantees us that no other thread can read or write the value at the address we are using. While histo[buffer[i]]++ can result in botched up result of operations due to multiple threads using the same address to perform calculations. Thus giving different value as result than what would have been required.

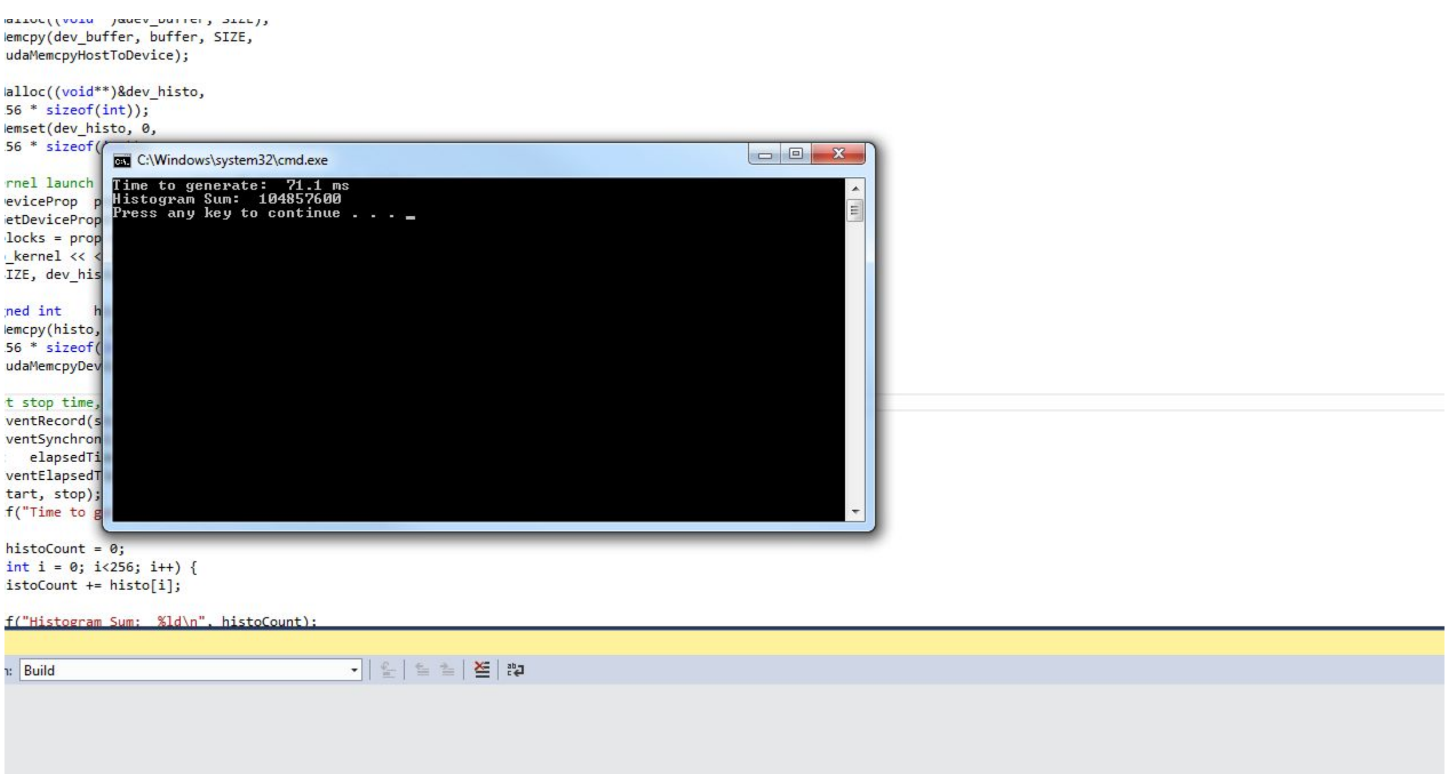
2. Compile and run the attached CUDA code hist_gpu_shmem_atomics.cu of histogram. Record the outcome. Please answer the following questions:

    a. Compare the outcomes of hist_gpu_gmem_atomics.cu (in problem 1)and hist_gpu_shmem_atomics.cu. What's the difference in outcomes?

when we run hist_gpu_gmem_atomics.cu :



when we run hist_gpu_shmem_atomics.cu :



The time taken when we use shmem reduces considerably.

b. Which program is fast? Why?

Shared memory is magnitudes faster to access than global memory. Its like a local cache shared among the threads of a block. The use of shared memory is when you need to within a block of threads, reuse data already pulled or evaluated from global memory. So instead of pulling from global memory again, you put it in the shared memory for other threads within the same block to see and reuse.

3. What's the difference between malloc() and cudaHostAlloc(). What's the purpose of using cudaHostAlloc(). What is the trade-off of using cudaHostAlloc().

The C library function malloc() allocates standard, pageable host memory, while cudaHostAlloc() allocates a buffer of page-locked host memory. Sometimes called pinned memory, page-locked buffers have an important property: The operating system guarantees us that it will never page this memory out to disk, which ensures its residency in physical memory. Using pinned memory is a double-edged sword. By doing so, you have effectively opted out of all the nice features of virtual memory. Speci cally, the computer running the application needs to have available physical memory for every page-locked buffer, since these buffers can never be swapped out to disk. This means that your system will run out of memory much faster than it would if you stuck to standard malloc() calls. Not only does this mean that your application might start to fail on machines with smaller amounts of physical memory, but it means that your application can affect the performance of other applications running on the system.