CMPE 220 HA4 Report

Group 03:
Pranav C Yerabati Venkata
Ishaan Paranjape

Question 1. How do the results of the best time for the shared memory GPU code compared with block only code, and compared with the CPU code.

Answer:
Out obtained values are:
CPU Run: 205ms
Device Run, **Not Using** Shared Memory: 11ms
Device Run, **Using** Shared Memory: 8.87ms

Looking at the values, it's clear that using the Nvidia GPUs gives a much faster result (comparing the CPU runtime of 205ms with Device runtime of 11ms). However, between the two Device runs, our results are what we expected. Using shared memory results in a faster computation (comparing 11ms with 8.87ms). Using shared memory, each threads in a given block will use the low latency memory near the processor core (like the L1 cache) shared memory rather than go to main memory, which takes much longer.

Question 2. What is the purpose of using these "halos" in GPU programming? What kind of programs do would this apply to.

Answer:
The halos refer to the left and right segments in the shared memory, that includes elements from adjacent blocks. To compute a stencil, each thread computes the sum of all the elements within a certain radius. However, the beginning of each block, and the end, must include the end and beginning of the adjacent block segments respectively to compute properly. These end and beginning segments are known halos.
Halos are needed mostly for computation of stencils, which are widely used for running computer simulations on GPUs.

Question 3. - What do __syncthreads do?

Answer: (According to CUDA Toolkit Documentation Section B.6) - __synchthreads waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block. This function is used to coordinate communication between threads to prevent data hazards while accessing shared memory. Synchronizing threads in-between accesses can prevent this.

Question 4. Can you use __synchthreads to synchronize all the threads in a Grid?

Answer: No, _syncthreads can only apply to threads within a block. It does not apply to threads between blocks.

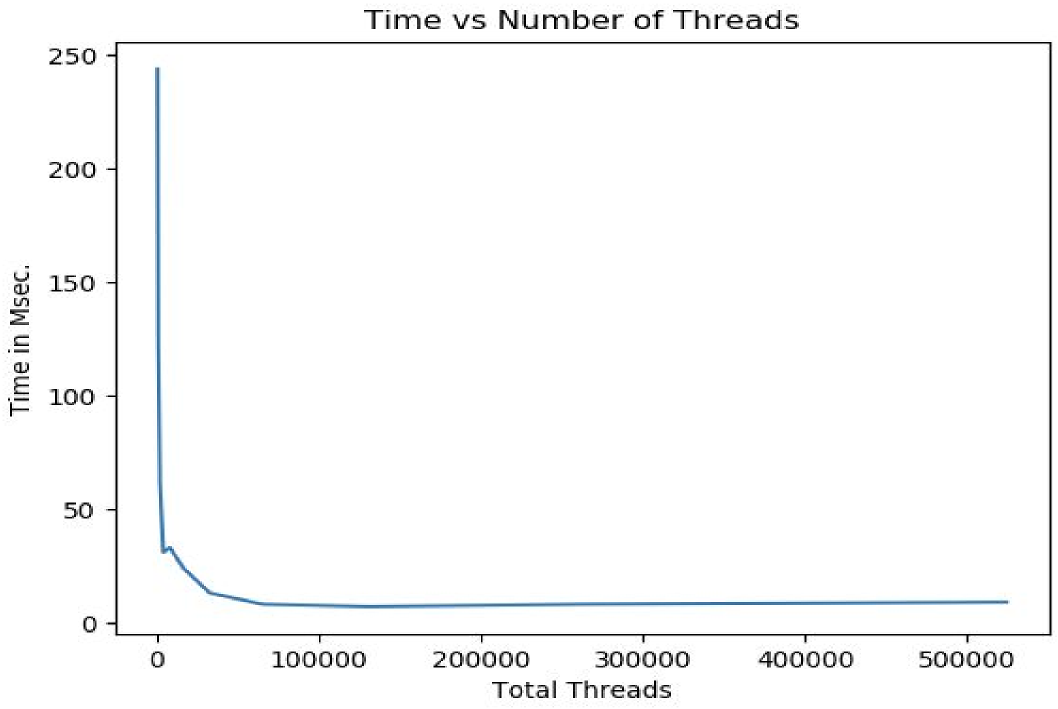
Question 5. What is the purpose of using __syncthreads in the shared memory version of the stencil program?

Answer: __synthreads is needed in two places in the shared memory version on the program. The first place it is used is after a thread finishes assigning a value to its location in the shared array. It has to wait till all threads finish writing to the shared array to avoid reading an invalid value from the shared array in its stencil computation. For example, if we consider two adjacent threads 10 and 11, if 11 has finished writing to the shared array and continues to compute the stencil before 10 writes to its position in the shared array, thread 11's computation of the stencil will have an incorrect value.

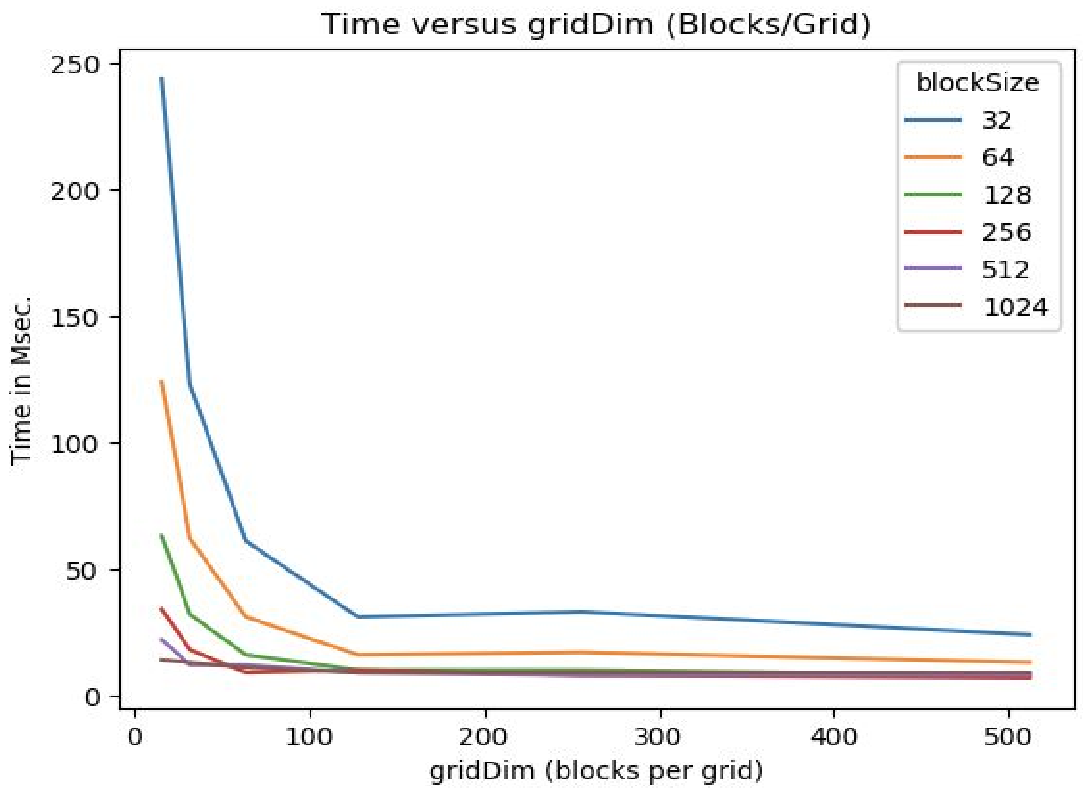
The second place __syncthreads is used is at the end of the computation for the loop iteration. All threads have to finish before starting another iteration to avoid data races for the output array.

Question 6. Make some observations based on your resulting graphs. Add comments as to what other optimizations can be done to optimize stencil.

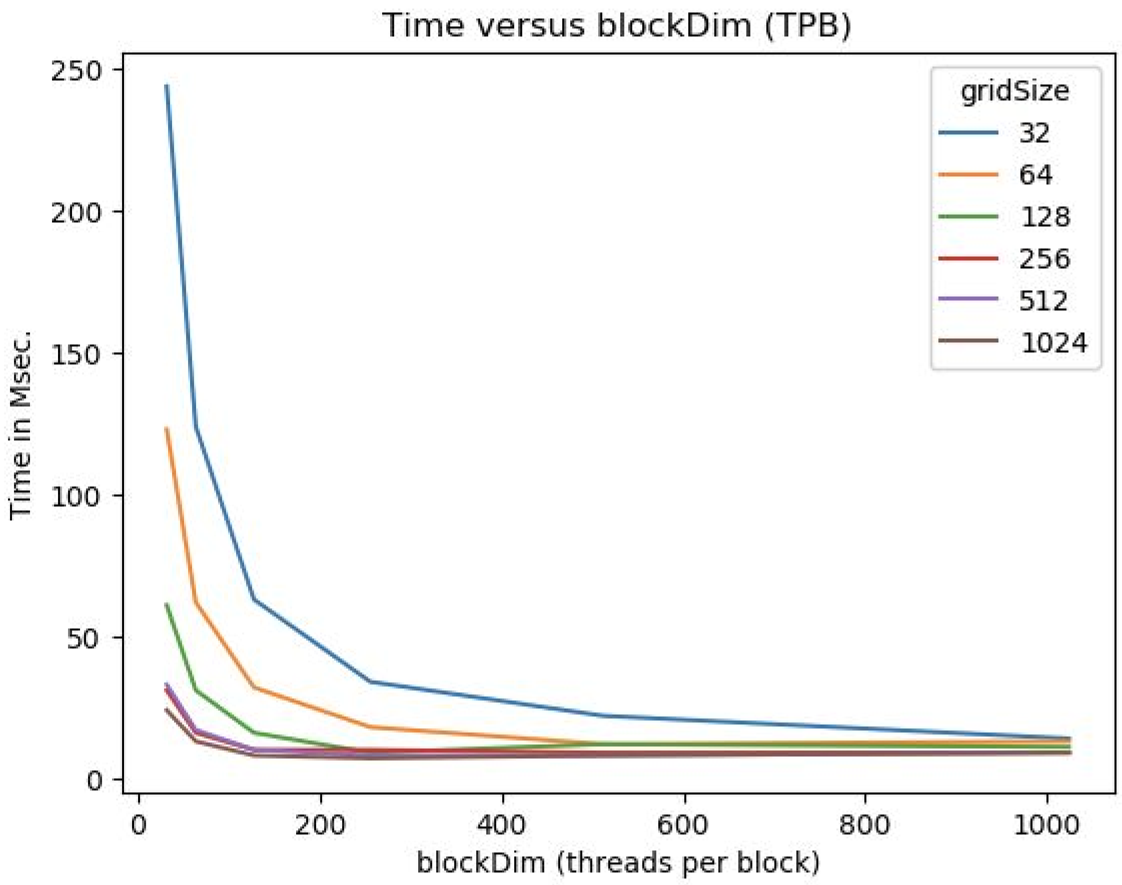
Answer: The graphs are shown below -



The observation from this graph is as the thread count increases, the total execution time reduces. Looking at the graph, we can see that the time reduces much faster as the number of threads are incremented upto a certain value, and begins to plateau, after which incrementing the number of threads will not yield a faster time. This is because each thread is still limited by the memory access times and other aspects, and increasing the number of threads does effect the time in such a situation.



In this graph we can make an observation that after a threshold thread count (512) within a block, increasing the number of blocks within a grid does not provide us with any improvement in execution times for a particular block size. For example, we can see that increasing the number of blocks after 100 for a block size of 32 threads per block will not yield much better times. However, increasing the block size to more threads per block does yield a faster time, which can be seen in the plateau times between block sizes 32 and 512 (blue and purple lines).



For the 32 blocks per grid setting, we see a sharp decrease in execution times as the number of threads per block are increased. This initial rate of decrease keeps reducing as the number of blocks per grid is increased. However, we can see that after a certain number of threads per block (around 1024), increasing the number of threads does not affect the latency as much. This can be due to the memory access times and other limitations of each thread. However, incrementing the grid size (number of blocks per grid) yields a faster rate of decrease in latency, as we have more cores to work with. For a high enough thread count per block the grid size doesn't matter as the execution times are mostly the same.