Program Descriptions:

In this assignment, we are required to implement Bitonic Sort with GPU by edifying the existing Bitonicsort program in cuda-samples. Note: to compile the program, we need the "helper_cuda.h" header file, which is provided in the sample file. So we need to put the file "sortingtry" into the"cudasamples"in order to compile it.

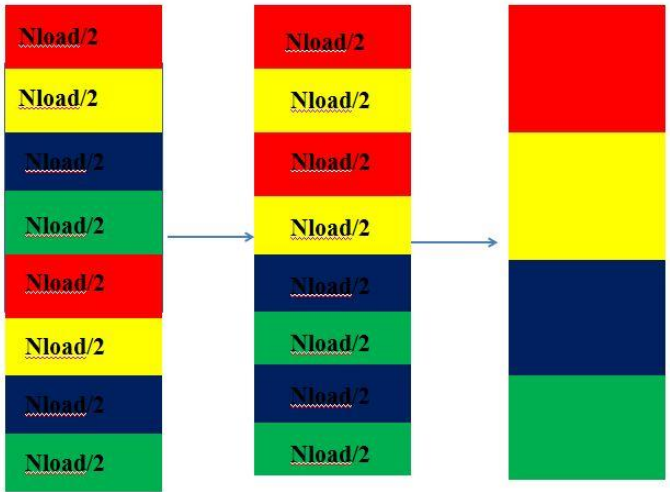
Note: The code initially has the "value" part and the odd-even sorting part. I intentionally deleted this part for my implementation.

I modified the program so that it can sort both the large arrays (k>20) and the small ones (k<10). The first part of the main.cpp Host function code will deal with the case for k<20. I use the existing example code for this implementation. It assigns the data into the GPU and get it back by cudaMemcpy() after going through the bitonic sorting network.

The second part of the Host function deals with the large array (k>20). It will assign the data into GPU 1M by 1M, since we assume the GPU global memory has only one megabyte of data. Firstly, it will still use the existing bitonicSort code to sort small size of array chunk by chunk. While sorting, the program is implemented so that each chunk has different order with respect to its neighbors, ensuring the "bitonicity". Later, for chunks that are larger than 1M, we separate them into smaller chunks and send them separately. Variable "Nload" is used to implement this. It will grab Nload/2 of data from the "upper" part of the data and Nload/2 from the "lower" part, and put them into the BitonicSort2.cpp device code.

We would like to use the graph below to show the idea of this implementation: The chunks with the same color are loaded into the GPU global memory once at a time. After the sorting, they will be loaded back into the host memory.

After the step, we change to load another pair of chunks (still, same color shown in the middle graph). In the final step, the chunk will send Nload of consecutive memory of data into the GPU, and we use the original function BitonicSort() to complete the final sorting network.



The BitonicSort2.cpp device code will only utilize the function bitonicMergeGlobal2() to do the“compare and swap” implementation. Later, if the stride of comparison is smaller than Nload/2, the Host code will call BitonicSort.cpp again to do a thorough sorting and finish the last part of this implementation.

The validateSortedKeys() function in the Host code will help validate the correctness of the sorted array. It consists of several testing steps. Firstly, It helps detect if the array length is too short (<2);secondly, it tests if the value of interger elements in the array are within the limit set earlier in the Host code (here we use numValues = 65536 as the limit); thirdly, it tests if the sorted array can match the original array (if the sorted array has grabbed different number from somewhere else, or happened to doubly copy part of itself, it cannot pass this test). Most importantly, it helps detect if the sorted array has the order assigned correctly. If the array passed all the test, the validateSortedKeys() will return and print “OK” on the screen.

For the Device Code bitonicSort.cpp, it is composed of the following parts:

The function BitonicSort() will decide whether or not to use the shared memory according to the size of the data. If the array unsorted can fit into the shared memory, then it will call the function bitonicSortShared(). This function will implement a complete bitonic merging network in one block of shared memory to do the sorting for this small array (in this way the implementation is very fast). On the other side, If the array size is not able to fit into the shared memory, the BitonicSort() will call the bitonicSortShared1() to ensure even/odd subarrays in the large array being sorted in different direction. Then bitonicMergeGlobal() is called for merging the stride greater or equal to the SHARED_SIZE_LIMIT, the idea is similar to what shown above for the colorful scheme. Later, the bitonicMergeShared() is called in order to complete the sort for stride <= SHARED_SIZE_LIMIT/2, using the shared memory in just one block.
There is other small functions like fatorRadix2(), which is for checking if the size of the input array is a power of 2. Here only the power-of-two array is supported in our implementation. BitonicSort2.cpp only uses the BitonicMergeGlobal2() function to deal with the compare-and-swap implementation for the stride >= 1M.

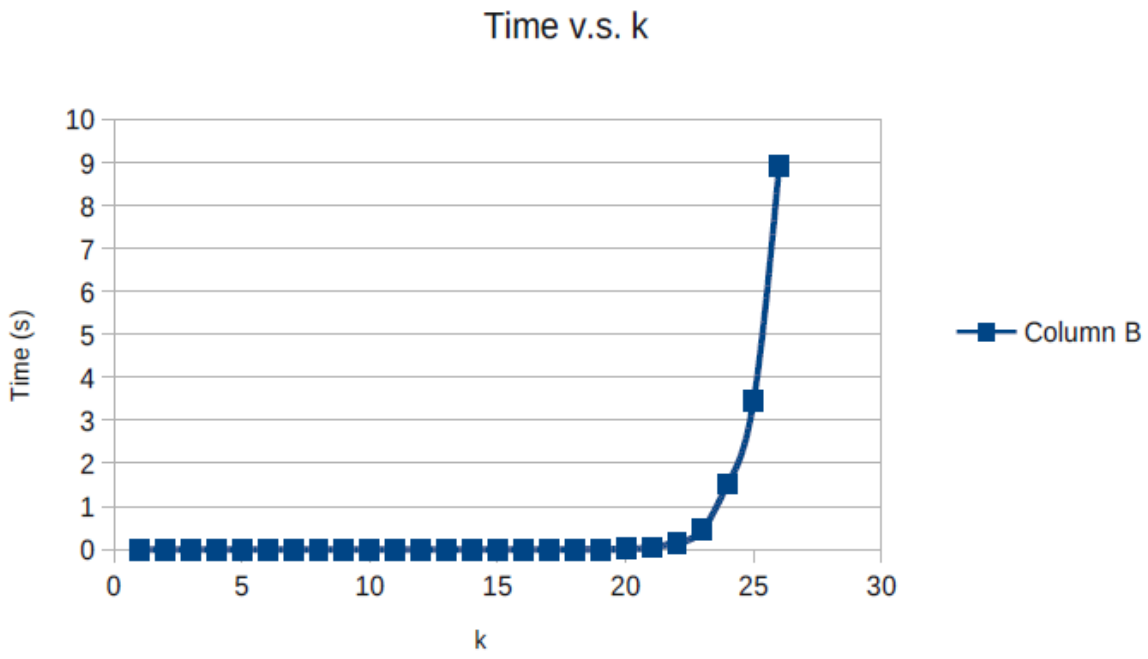
Utilization of the processors:
If the array size is larger than 1024U, then it will use the whole shared memory of one block with 512 threads. GPU is designed for stream computing, where cache memories are not effective . If some threads stall at some places of memory operation, other threads will switch to it. (ref. Understanding the CUDA Data Parallel Threading Model – A Primer by Michael Wolfe). In this way, we can keep a specific number of processors busy. Depending on the size of the data, it will keep different number of blocks busy. The number of blocks run on GPU is calculated as:

Number of blocks = (2^k) / 1024 = 2^ (k-10).
Since we are using only one dimension here, and the maximum number of thread blocks in one dimensional grid is 65535 = 2^16-1 (ref. CUDA_C_Programming), if we implement k=26, we will fully employ all of the thread blocks. Whether the program employ all of the multiprocessors will depend on how many processors are in charge of the blocks.

The shared memory is “on-chip”, meaning it has higher band-width and lower latency than global and local memory, on the condition that no bank conflicts happen between the threads (ref. CUDA_C_Best_Practices_guide.pdf). The reason for using the shared memory is because the bandwidth in shared memory is higher compared with global memory. So we use as much shared “on chip” memory as possible (here one block of 1024U) to minimize the data transfers between global memory and the device, thus speeding up the data processing.

This program uses the coherent loads in the global memory. Each time it will exchange the chunk of data in the global memory that are adjacent, unless the stride of the data is bigger or equal to the SHARED_SIZE_LIMIT. Even in that case, it still deals with two chunks of data that are continuous in the memory location, as we show earlier in schematic graph. With this kind of loading implementation, we could ensure a faster implementation of sorting.

Performance:



We can see that the running time would grow nearly exponentially with k