# Assignment 4: Bitonic Sort using CUDA

Aniket Gattani

## Notations:

1. k : This is the user input. k varies from 0 to 26 and alters the array size.
2. N: Array length input to the bitonicSort function. N = pow(2, k) and N = batchSize * arrayLength
3. Nmax: maximum size that can be used to perform bitonic sorting in one iteration on the GPU.
4. SHARED_SIZE_LIMIT: the amount of shared size available for each multiprocessor.
5. arrayLength: current size of the chunk on which sorting or merging has to be performed
6. size: 2*stride
7. stride: length difference of the current iteration of bitonic sort chunk of size arrayLength

## Existing Code:

The existing code provides two ways to perform bitonic sorting:

1. The code performs multiple iterations of bitonic sorting using different arrayLengths and batchSizes. batchSize * arrayLength = N. All memories initialized in the host or the device are of size = N.
2. The array is initialized using random integers from 0 to 65535 and another array is used to store the position of every element in the array. When performing compare and swaps, the positions are also swapped.
3. **arrayLength <= SHARED_SIZE_LIMIT:**
    a. If the arrayLength is small enough to fit in the shared memory, then the sorting and merging is all done using the shared memory (***bitonicSortShared***). Explicit synchronization in the thread block ensures no data races within the shared memory.
    b. Number of threads in a block here are SHARED_SIZE_LIMIT/2. The different sized subarrays are sorted first and then merged. The shared memory is accessible to all threads so between different iterations sorting and merging, they can sort the array using different stride lengths.
    c. Since the arrayLength can be from 1 to SHARED_SIZE_LIMIT, if the arrayLength < 32, then some threads will remain idle since the warp size = 32.
    d. Since each warp is executed on a cuda core and there are 2496 cores (192 on each of 13 SMPs) if N < 79872 (2496*32), not all SMPs will be employed.
4. **arrayLength > SHARED_SIZE_LIMIT:**
    a. If the arrayLength is bigger to fit in the shared memory, then the subarrays of length = SHARED_SIZE_LIMIT are sorted (***bitonicSortShared1***) in odd-even fashion. However, part of the merging cannot be performed in shared memory. As a result, the function ***bitonicMergeGlobal*** uses global memory to sort when the stride length

   >= **SHARED_SIZE_LIMIT**.  When the stride is less than this, the subarrays can be merged in the shared memory (***bitonicMergeShared***). Again, code performing operations within a shared memory have explicit barriers using the thread_block cooperative_group.
   b. Again if N > 79872 (2496*32), not all SMPs will be employed.
5. Each thread performs a comparison between two elements at a time. Hence, the number of threads required for one computation should be equal to N/2. Every block performs computation on a shared memory. The code uses only 1-D grids and hence, the **blockCount = N/ SHARED_SIZE_LIMIT.** Now, **blockCount * threadCount = N/2.** Hence, **threadCount = SHARED_SIZE_LIMIT/2.**
6. This code also imposes a restriction on the array size to be a power of 2 and **N%SHARED_SIZE_LIMIT == 0**.
7. Finally there are two validations performed by the code, one checks whether the input and the output have the same histograms and checks the order of the values. The other is a validation on the positions of the sorted values and whether the array is stable i.e. similar values have decreasing order of positions.

## Limitations of existing code:

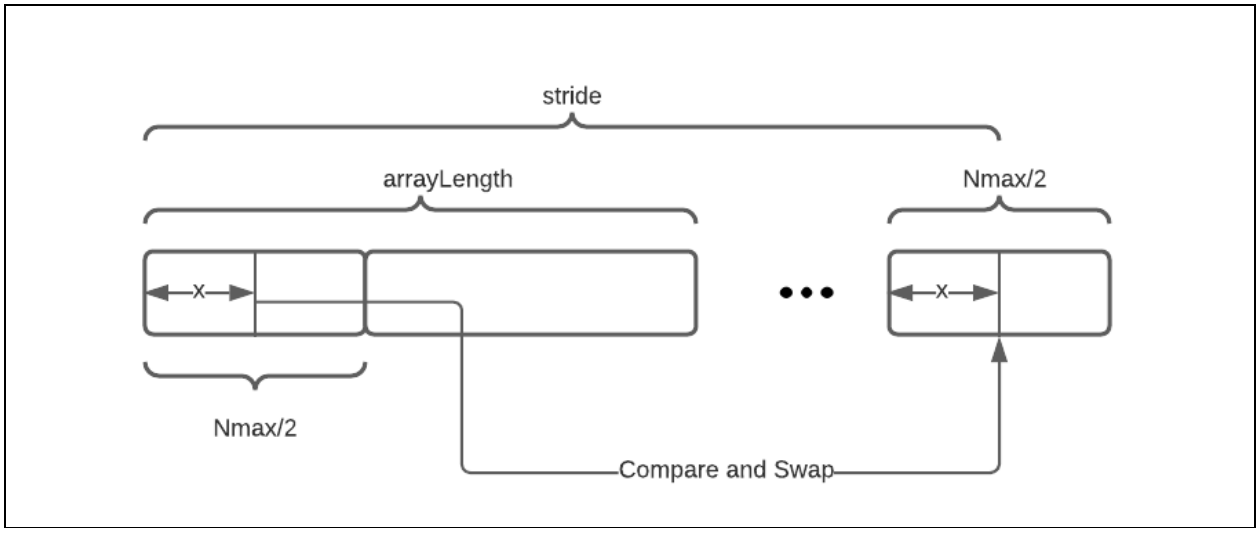
1. The existing code assumes that the entire array fits into the device's global memory. However, this cannot be true when N>2^20.
2. It also assumes that N is a multiple of **SHARED_SIZE_LIMIT.** However, this is not true for N < **SHARED_SIZE_LIMIT.**
3. Lastly, the existing code unnecessarily uses a key-value pair model and sorts both of them. This is redundant since we can validate the entire ordering using either only the keys or values.

## Program Overview:

The program implements the bitonic sort algorithm in 3 different steps:

1. arrayLength <= Nmax or N < Nmax:
   a. Perform bitonic sort the regular way. First sort all chunks in shared memory. Then recursively merge different sized chunks and strides.
2. arrayLength > Nmax and stride > Nmax:
   a. Since all of the data cannot fit inside memory, we need to load Nmax chunk data into GPU and perform a global merge using ***bitonicMergeGlobal(arrayLength=Nmax, stride=stride, size=Nmax)***
3. arrayLength > Nmax and stride = Nmax/2:
   a. Since all of the subsequent merging use stride < Nmax which means all data can fit inside device memory, we can perform the rest of the merging in one iteration.  Half of the merge using ***bitonicMergeGlobal(arrayLength=Nmax, stride=stride, size=Nmax)*** when stride > SHARED_SIZE_LIMIT.  If stride < SHARED_SIZE_LIMIT use ***bitonicMergeShared(arrayLength=Nmax, size=Nmax)***

4. The above branching is denoted by the variable onlyMerge. onlyMerge = 0 when arrayLength == Nmax since we need to first sort the array. onlyMerge=1 when arrayLength > Nmax and stride > Nmax since we only need to merge the chunks. onlyMerge=2 when when arrayLength > Nmax and stride = Nmax/2. This special case handles the recursive merging when stride <= Nmax/2 in one single transaction avoiding multiple transfers to and from device memory to host.
5. Bitonic sort works by first sorting subarrays into an odd and even sequence . Merging them to double the size and then recursively merging smaller subarrays again. Hence, the arrayLengths iterate over (Nmax, N) and strides over (arrayLength/2 ,  Nmax/2).
6. Since we cannot sort the entire array in a bunch, we have to iterate over the entire length N (N/arrayLength) times and for every stride > Nmax, we have to also iterate (stride/ (Nmax/2)) times.  Once we reach stride = Nmax/2 i.e. size = Nmax, we can perform the bitonic merge as shown above.
7. Every iteration when stride > Nmax, first copies Nmax/2 of the array into the device memory. Then finds the part separated by the stride length and copies Nmax/2. Then we use the function ***bitonicMergeGlobal,*** to merge the chunk with size = 2*stride.
8. One important caveat here is that the direction of sorting has to change with every subarray of size = arrayLength.
9. Once again, all SMPs will be employed when N>Nmax since Nmax >  79872. There is no branching for any thread inside a block and hence there is no stalling.

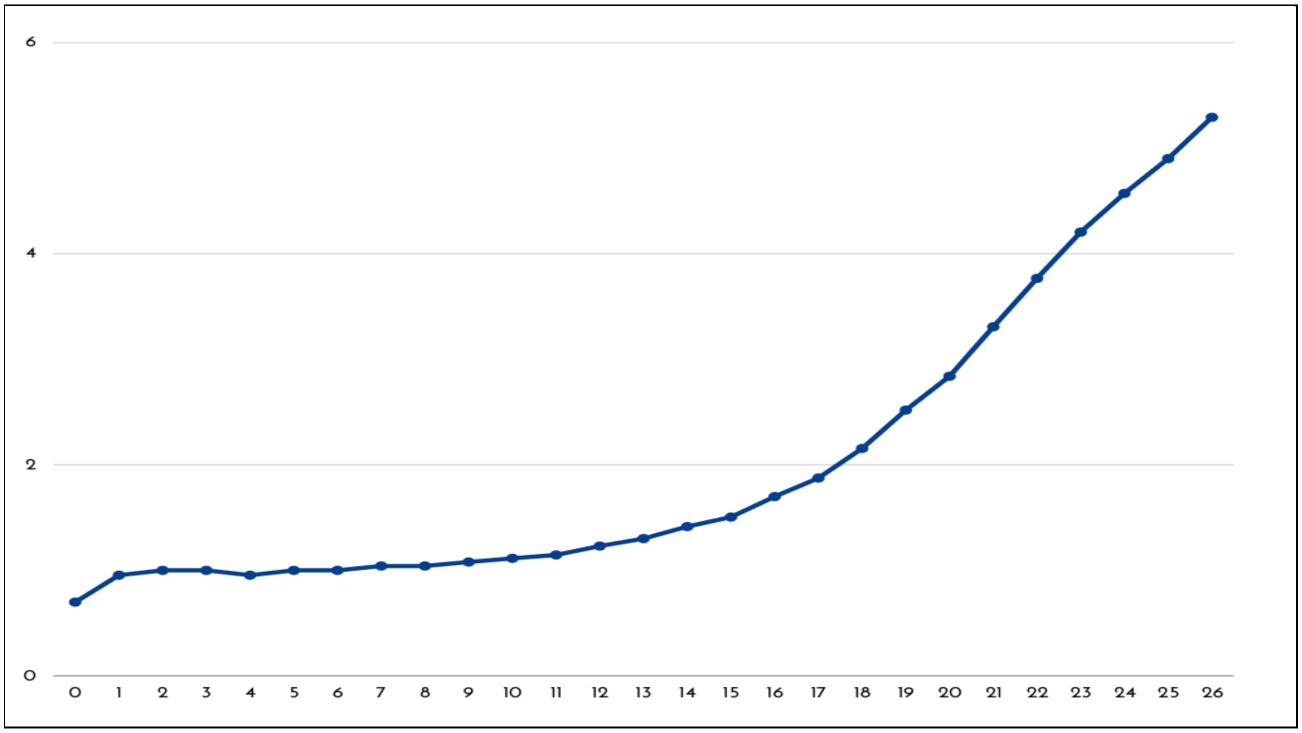


*Visualization of Host Memory copied to device when arrayLength > Nmax and stride > Nmax/2*

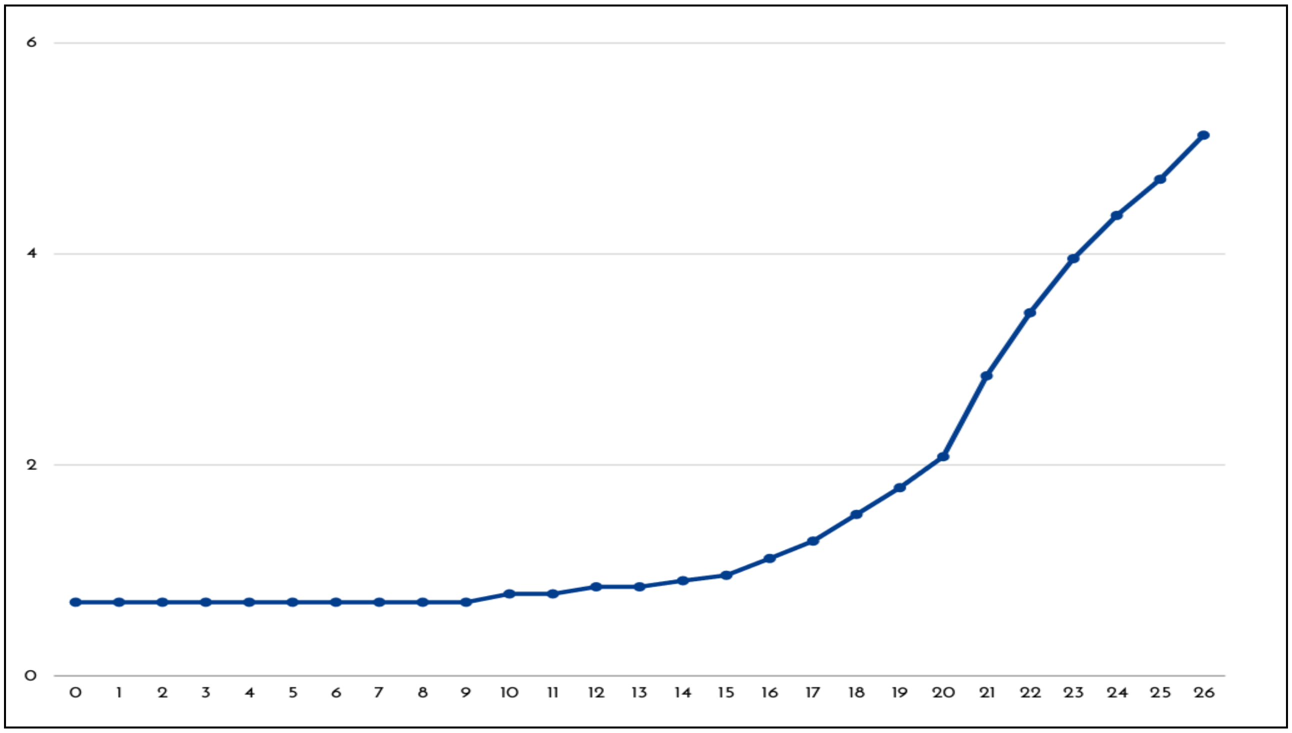
## Performance:

1. The performance measurements are done from k=0 to k-26. The time is graphed as $\log_{10} + 5$ to prevent negative numbers.
2. The program takes 1.96 secs and takes about 1.35 secs for copying to and from the device and is output by the code.

3. As we can see from the graph, the runtime increases exponentially with N. The growth is steep once k > 20, this is expected because we use multiple rounds of copying data back and forth from the device. The global memory is the slowest in the memory hierarchy and uses the PCIe bus and is an off-chip memory. Hence, the time spent in copying to and from the global memory is critical.
4. When k<=10, the time almost stays the same because we use shared memory which is on chip memory. The global memory is used only twice by the host. Once to push the input and once to get the output. This is also evident from copy time which is constant.
5. When k>10 and k<=20, we can start seeing the rise in the runtime. This is because not all the input can fit into the shared memory and part of the merging has to be performed using global. Since this memory is further away from threads, the threads spend a larger time to access this. The operations to global memory also increase with increasing k and that explains the added runtime.
6. While reading memory to and from the device, the cuda runtime first copies the pages allocated on pinned memory in the host so that the host OS does not move the pages in memory around. Using cudaMallocHost improves the copying time by almost a factor of 2. The cudaMallocHost directly pins the memory on the host and saves the time in each data transfer. Although cudaMallocHost is slower than malloc but it is a 1 time cost rather than incurring a cost for every transfer



*Execution Time graph (k vs. $\log_{10}$ (time) +5) k ∈ [0, 26]*



*Copy Time graph (k vs. $log_{10}$ (time) +5) k ∈ [0, 26]*