Implementation:

For Cuda Merge Sort, I used the cooperative groups which allows for blocks in the grid to be synchronized. One tradeoff for cooperative groups is that the number of concurrent blocks allowed to be cooperative is equal to the max number of multiprocessors in the Cuda Device; thus, in the COC ICE computing device, there are 80 multiprocessors which is a maximum of 80 blocks. Thus, in order to fit 1 million numbers into 80 blocks of 1024 threads each thread will need to manipulate 16 numbers to have a maximum of 1.3 million elements possible. We call the number of elements each thread in the block needs to manipulate as defined as tile and the number of max threads specified as THREADS. In order to increase the number of elements, we will have to increase the number of elements in a tile.

In the host main function, CudaMalloc was used to malloc the input device array and the output device array of the desired size. The program takes in an input as a number of elements which will be the size of the input array and output array. The input array was generated with random numbers by the rand() function. CudaMemcpy is used to copy the input array and the output array from host to device. Since we are using cooperative groups, the kernel function has to be launched with `cudaLaunchCooperativeKernel` with the specified THREADS and tile numbers. In order to test the output of the kernel function, the C++ std::sort method is used , and a for loop is used to check every element from the output of the kernel function. Lastly, cudaFree is called on the device allocation input and output array.

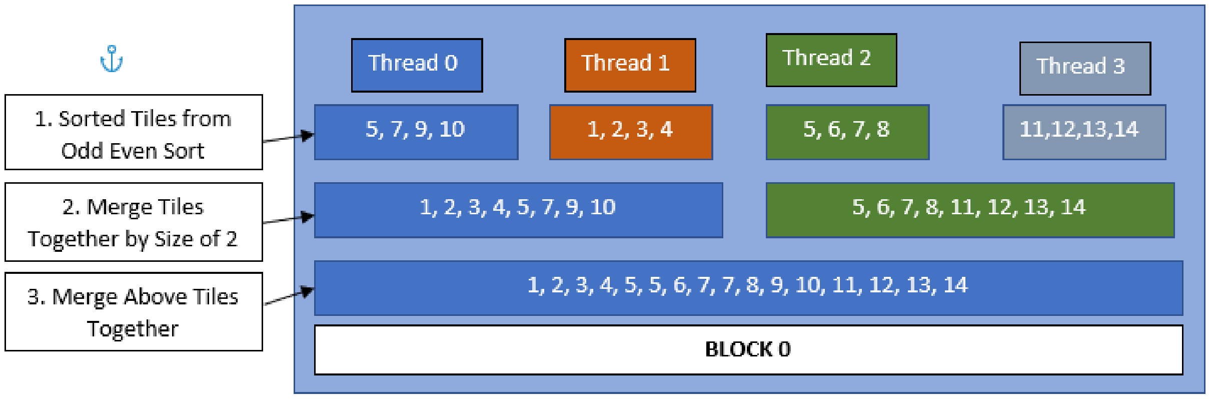
For the kernel implementation of the merge sort, because of the possibility of a large number of elements and the use of a cooperative group, each thread in the block will be sorting a tile amount of elements with an odd even sort which has a runtime of O(tile) runtime in parallel per block. In Figure 1, Odd and even sort is basically sorting the even pairs of numbers and sorting the odd pairs of numbers , for example index 0 is compared with index 1 and swap if index 1 is smaller than index 0. After the tile amount is sorted per thread, the thread tiles are merged together in the block by iteratively merging two pointer method which runtime is O(n) which n is the number of elements by the even thread index merging by groups of 2 and collapsing into groups of 4 and groups of 8 with each successive merge size multiple by 2 like how merge sort collapses but from the standpoint of threads tiles into a single block as in Figure 2. Next, the blocks are merged together in a parallel manner in which each thread index will binary search for its tile size of elements as mentioned in [1]. If the tile is 16, then each thread will search for the global position of 16 elements in the output array. When the global index is found, the output array at that index will equal that specific element. The array manipulations were used with indexing input array and the output array. Shared memory wasn't used because of the limited amount of shared memory for large numbers. Before the merging blocks part, the cooperative group had to sync.

Without cooperative groups, the merge sort will require 2 kernel calls, 1 kernel call for splitting the data and merging within blocks and the next kernel call will be merging the blocks together.

Figure 1. Odd and Even Sort: The Green color numbers are compared with each other and the blue color numbers are compared with each other, in the first row, index 0 is compared with index 1, and index 2 is compared with index 3. In the second row, index 1 is compared with index 2. After n - 1 rows of alternating comparisons , the array is sorted.

5	3	7	1
3	5	1	7
3	1	5	7
1	3	5	7

Figure 2. Merging Tiles Together From their threads to a sorted block. The colors represent which thread is going to do the merging in the block.



Performance and Discussions:

With random numbers and different random number generator seeds, the program was executed 5 times, the average of the statistics of number of global memory accesses, number of local memory accesses, number of divergent branches , and achieved occupancy for different array sizes are shown in Figure 3. From the averages, the number of local memory access was 0 because shared memory was not used. Number of global memory accesses increases as the number of elements in the arrays increase. Our theoretical occupancy is at 50% and our achieved occupancy is around 45 to 49 percent for all the cases which means that about half of the possible maximum warps was not used which can mean that more blocks can be launched. In addition, our thread per instruction executed is less than 50% at ~25% for 10 elements to 19.80% for 1M elements which means that large amounts of numbers were packed together which is the case as our tile size is constant at 16 for each thread. We can not use cooperative threads and split the kernels which will maybe increase the occupancy and the active thread per instruction executed. Our divergence efficiency varies by input size, but for all of them it is less than 30% which means that less than 30% of possible threads in a warp is executed at a time, this can be cause by the threads in each block merging the sequence together by alternate threads and in the final step only one thread merges the 2 half sorted sequences in a thread. In addition, since thread count is constant at 1024 and tile size is 16 for each thread, an entire block can support over 16,000 elements and one 32 thread warp can support over 512 elements definitely is a reason for the low warp efficiency.

Figure 3. Ncu Metric Measurements

Array Size	10	100	1000	10000	100000	1000000
Metric						
L1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum (global memory access)	191	3,882	48,732	597,671	11,774,780	697,671,236
L1tex__t_sectors_pipe_lsu_mem_local_op_ld.sum (local memory access)	0	0	0	0	0	0
sm__maximum_warps_per_active_cycle_pct (theoretical occupancy)	50	50	50	50	50	50
sm__warps_active.avg.pct_of_peak_sustained_active (achieved_occupancy)	45.63	47.88	49.57	49.91	49.18	49.53
smsp__thread_inst_executed_per_inst_executed.ratio (divergence efficiency) (warp_execution_efficiency	29.69	23.79	10.35	3.05	12.36	23.93
smsp__thread_inst_executed_pred_on_per_inst_executed.ratio Active Threads per Inst Executed%: 25% << 50% Indicates packing issue and/or high divergence	25.63	20.48	8.77	2.55	10.24	19.80

Using nvprof to analyze the total execution time, we see that total execution time and the data transfer time increase as the number of elements increase in Figure 4. Execution time between 10000 and 100000 elements only doubled because of an increased number of blocks at around 16,000. Because each block has 1024 threads and each thread manipulates 16 elements, that's 16384 elements until you get a new block which explains 2x in time while 10x the number of elements. Data transfer time seems to increase by 10x from 10k elements to 100k to 1M elements.

Figure 4. Nvprof Time Measurements

	10	100	1000	10000	100000	1000000
Total execution time including kernel launch time.	156.5 us	221.3 us	396.19 us	3.27 ms	6.27 ms	19.95 ms
Data Transfer Time	4.77 us	6.7 us	7.14 us	33.32 us	497.7 us	5.26 ms
Execution time excluding data transfer time	151.7 us	214.6 us	389.05 us	3.40 ms	5.77 ms	14.7 ms

Performance optimization techniques and discussions

If we want to dynamically define the number of threads instead of a macro definition, the merge sort device function will need 2 extra parameters for the dynamically determined thread count and tile number. This will increase the occupancy for the GPU. Another optimization that can be tested is instead of using 2 pointers, each thread tile can binary search its element's position in the sorted block sequence which will allow more parallelism between threads in blocks. Instead of odd & even sort for the tile in each thread, the paper [1], Satish mentions Batcher's bitonic sort and Batcher's odd even merge sort which can make the tile sorting parallel among threads without the use of cooperative groups which limits total thread count. So each number will occupy a thread and parallel sort.

To run the program: ./a.out <number of elements> <random seed>

```
$ nvcc mergesort.cu
$ ./a.out 10 1
$ ./a.out 1000000 1
```

References:

[1] "Designing efficient sorting algorithms for manycore GPUs"
 17 * by Nadathur Satish, Mark Harris, and Michael Garland,
http://mgarland.org/files/papers/gpusort-ipdps09.pdf
[2] https://ams148-spring18-01.courses.soe.ucsc.edu/system/files/attachments/note8_0.pdf