Solution Construction

Implementing the MPi section proved to be very challenging, simply because of the recursive nature of quicksort. In the original solution I split up the unsorted array across all available processes as equally as possible, and had each process run a standard sequential quicksort implementation on their sub-arrays to produce a sorted array that will be merged back into one on the master node.

The merge step was easy and was completed using cumulative sums to determine the indexes of each element, but the post-merging stage proved to be the difficult part, and for a truly MPI-based quicksort solution, this needs to be done recursively by splitting up the left sub-array to send back across to each process and similarly with the right sub-array. The merge itself is easy enough to achieve using counters and cumulative sums, but this will not produce a sorted array, only an array where the pivot elements are in their final sorted positions. Any other implementation of performing the merge on the master node that results in a fully sorted array is then bound to the time complexity of the sorting algorithm/method that is chosen, making the first step of MPI quicksort essentially redundant and not providing any true speed up.

It was advised that I do this instead follow the recursive route. With N sorted sub-arrays I decided that implementing a min-selection based sorting algorithm from scratch was also redundant, and instead used the inbuilt priority queue data structure available within c++ to perform the final merge step on the master node. While this is then bound to the time complexity of the priority queue push and pop methods, it at the very least provides a benchmark for moving into the OpenCL quicksort implementation. As a side note, I did include efficient methods of calculating the cumulative sums in the MPI root by using MPI_Scan. This can alternatively be changed to MPI_Exscan to retrieve the exclusive prefix scan, but the exclusive scan can also be retrieved from the inclusive scan anyway. While these counts were not used in the end, I thought it was necessary to prove I was able to calculate them. The cumulative sums are also used in the OpenCL implementation below to partition the data, and the application can be seen in the partition kernel function of the solution.

OpenCL:

This part also took me several hours to both understand and implement. I ended up taking inspiration from the GPU-Quicksort paper (http://www.cse.chalmers.se/~tsigas/papers/GPU-Quicksort-jea.pdf) and started to understand the concepts required to implement quicksort in OpenCL. I split the array up into a number of sub-arrays depending on the OPENCL_MAX_GROUP_SIZE definition, and assigned a local size equal to the number of elements per group, and a global size equal to the number of elements per group multiplied by the number of groups.

This meant that each work group would have a number of threads that processed 1 element each, and any leftover elements as a result of the size not being divisible by the number of groups were allocated to an extra group (NOT the last group). The reason for this is one work group can not have more threads than another, which is defined by the local size, but it can use less threads than others (simply by calling return in the kernel function).

I then split the quicksort problem up into two important sections.

1. **Compute Counts kernel**
   The compute counts kernel works through each sub-section of the array and counts the elements that are less than and greater than the pivot directly into 2 global counter arrays (one for less one for greater) using atomic increment functions.
   The kernel then exits once this is complete to globally synchronize the data on the host and calculate the exclusive cumulative sums allowing the next section to determine where each element needs to be placed in the array. The global synchronization is performed because the program is developed with OpenCL 1.2, which doesn’t include useful functions such as work_group_scan functions and the ability to enqueue another kernel from within a kernel.

2. **Partition kernel**
   This kernel receives some parameters including the unsorted array, the cumulative sums, and other data allowing each thread to locate its elements in the unsorted array and locate the destinations for their elements in an auxillary buffer. Each global thread (1 thread represents a group) then iterates through their group of elements and inserts them into the array. Then the kernel exits and the host fills the gap with the pivot elements, resulting in the pivot elements being in their final sorted position.

After these 2 kernels are run, the quicksort function is called again, but on the smaller sized sub-array first. This limits the recursion call stack to a maximum of $\text{Log}_2 n$. Finally, a condition is set in each quicksort function where if the number of elements is less than a certain threshold, insertion sort will be used. This is done to ensure that there is enough work being done by the GPU to reasonably negate the overhead caused by initialising each kernel and executing them.

MPI + OpenCL:
Combining the two is simply a matter of replacing the sequential quicksort function with the new OpenCL based quicksort function, allowing the array to be split up across multiple hosts and run on each host’s GPU.