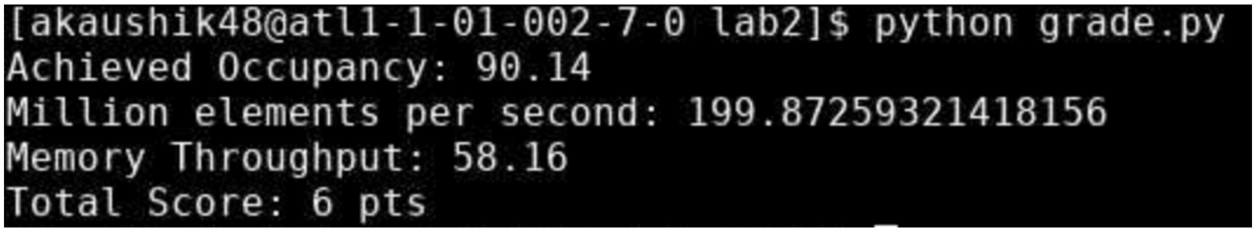
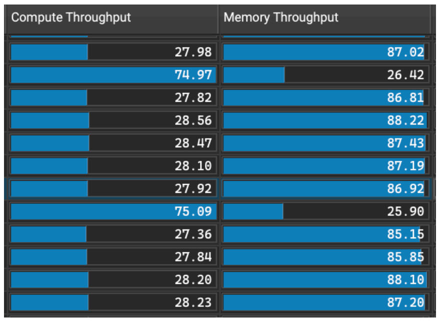
Introduction
The goal of this project was to write a program to run bitonic sort on an array in CUDA and optimize it to run quickly. The way I approached it was to first implement the bitonic sort algorithm correctly, and then add optimizations that would enhance the performance of the code on top of that. The primary optimizations that I made were in the theme of utilizing shared memory, reducing warp divergence, and reducing data transfer.

Implementation
My implementation of bitonic sort utilizes two kernels; one that uses shared memory and one that does not. The non-shared memory kernel is very simple; it swaps the two elements if they are out of order. The shared memory kernel first loads a subsequence of the global array into a shared memory buffer, and then performs bitonic sort on this buffer. Lastly it writes this back out to global memory.

I chose a buffer size of 2048 ints for the shared memory kernel because each thread loads two values into shared memory, and the maximum number of threads per block on an L40S GPU is 1024. Thus, with that maximum thread size, 2048 ints is the largest buffer I can accommodate.

Metrics

```
[akaushik48@atl1-1-01-002-7-0 lab2]$ python grade.py
Achieved Occupancy: 90.14
Million elements per second: 199.87259321418156
Memory Throughput: 58.16
Total Score: 6 pts
```
This was run from a V100. As you can see, the achieved occupancy is very high, which is good, but the memory throughput is low.

Another metric that I queried was smsp__thread_inst_executed_per_inst_executed.ratio. This was very high for my run, not being lower than 31.5 for any of the warps. This indicates to me that there was very little thread divergence within the warps, which shows that my optimizations to fix this aspect were good.

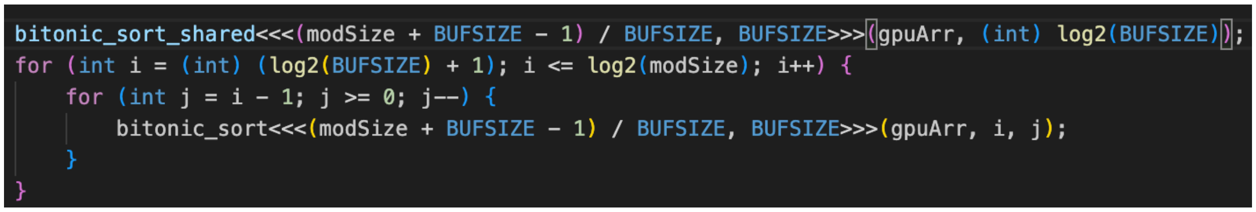


Compute Throughput	Memory Throughput
27.98	87.02
74.97	26.42
27.82	86.81
28.56	88.22
28.47	87.43
28.10	87.19
27.92	86.92
75.09	25.90
27.36	85.15
27.84	85.85
28.20	88.10
28.23	87.20

This is a screenshot from Nsight, and the kernels that have high memory throughput are the non-shared kernels and the kernels that have low memory throughput are the shared kernels. Also you see that the shared kernels have high compute throughput. It's possible that the shared kernels have too much computation to be done and finding a way to reduce that would be good for performance. It makes sense that memory throughput is high for the global kernel since shared memory is not being utilized as effectively, so all the reads have to come through the global data bus.

Optimizations
The first class of optimizations that I made was in the theme of shared memory.

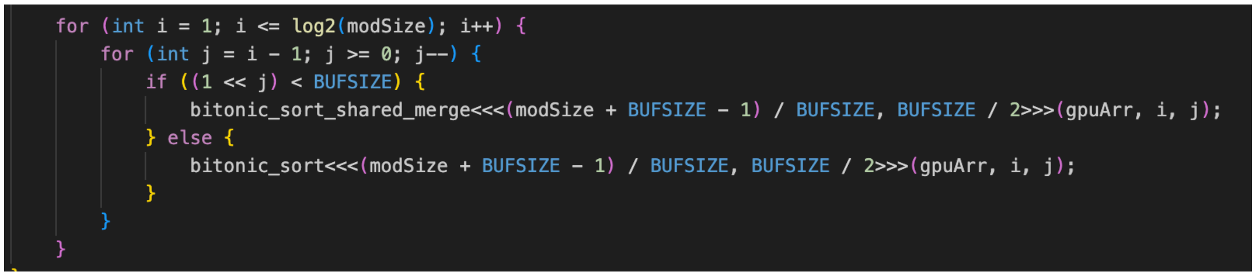
The first thing I did in this theme was to make a kernel that ran before everything else and ran bitonic sort on each subsection of the array that could fit in a shared memory buffer. After this, I ran the normal "naive" bitonic sort as normal.



```
bitonic_sort_shared<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE>>>(gpuArr, (int) log2(BUFSIZE));
for (int i = (int) (log2(BUFSIZE) + 1); i <= log2(modSize); i++) {
    for (int j = i - 1; j >= 0; j--) {
        bitonic_sort<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE>>>(gpuArr, i, j);
    }
}
```

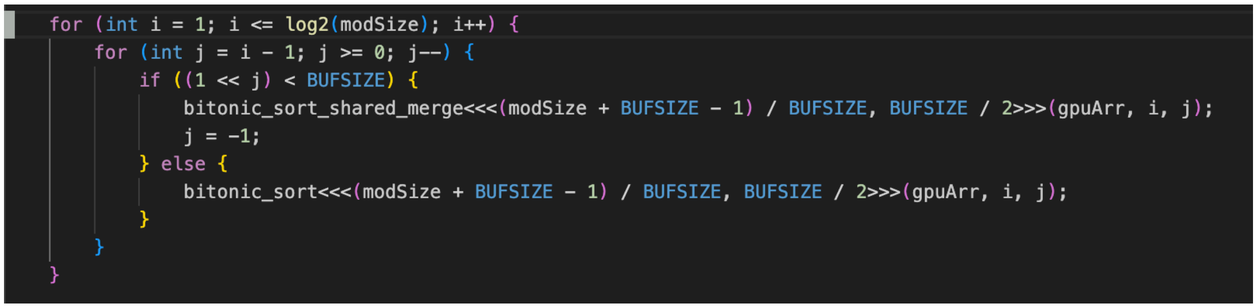
However, I realized that this is not optimal, because in the inner for loop, there are still some steps of the sort that can be done with shared memory rather than global memory.

The constraint on whether a bitonic sort step can be done with shared memory rests on how far away two elements that need to be switched are. In this for loop, an element k would be switched with k ^ (1 << j), which is the same as either k + 2^j or k - 2^j depending on the bit in the jth slot of k. Thus, if for all elements k within the same kernel, k ^ (1 << j) is within its own buffer, then we can run that kernel fully within shared memory. This holds true for small enough values of j in the inner for loop. Thus, we are leaving optimization on the table by only running the shared kernel once in the entire program. Ideally, we should be running the shared kernel whenever the rest of the sort can be finished in shared memory.



```
    for (int i = 1; i <= log2(modSize); i++) {
        for (int j = i - 1; j >= 0; j--) {
            if ((1 << j) < BUFSIZE) {
                bitonic_sort_shared_merge<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE / 2>>>(gpuArr, i, j);
            } else {
                bitonic_sort<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE / 2>>>(gpuArr, i, j);
            }
        }
    }
```

This for loop shows the next iteration of my optimization. In this loop, when the inner for loop is low enough for the sort to fit in shared memory, then the shared merge kernel is run. This kernel runs one step of the bitonic merge where elements loaded into shared memory are swapped. However, this is still not as good because it only runs one step per kernel, even though it is perfectly possible to run all the steps to completely sort the shared buffer in a single kernel. The problem with this current implementation is that there are multiple kernel launches for the shared memory version and more importantly, each shared memory kernel has to reload the shared buffer from global memory for each iteration of the inner loop (that is below log(BUFSIZE)).



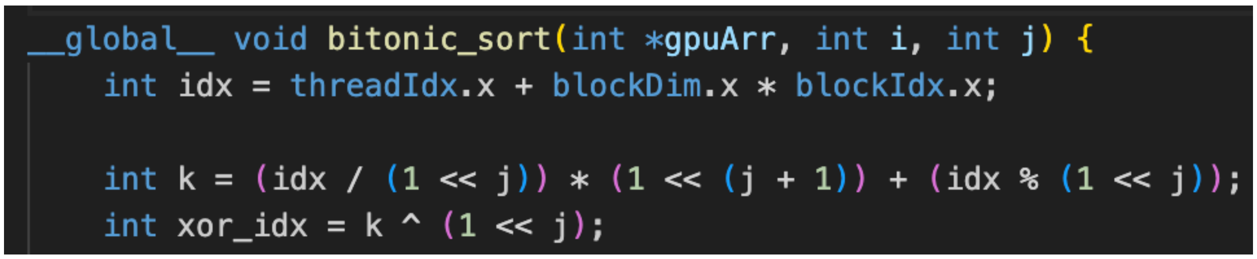
```
    for (int i = 1; i <= log2(modSize); i++) {
        for (int j = i - 1; j >= 0; j--) {
            if ((1 << j) < BUFSIZE) {
                bitonic_sort_shared_merge<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE / 2>>>(gpuArr, i, j);
                j = -1;
            } else {
                bitonic_sort<<<(modSize + BUFSIZE - 1) / BUFSIZE, BUFSIZE / 2>>>(gpuArr, i, j);
            }
        }
    }
```

This for loop shows the next iteration of my optimization. In this loop, the shared merge kernel only runs once per outer loop iteration. This is because once the kernel is run, it sorts the entire shared memory buffer, so it does not need to be run again until the next overall bitonic merge stage. This is much better than the previous iteration because it cuts down on all the kernel launches and shared memory reloads. The previous iteration had 300 kernel launches, while this version had 115 kernel launches. Unfortunately, this version had a lower memory throughput percentage, I think because there were far fewer memory loads. However, since it was still much more performant, I went with this optimization.

The second class of optimizations I did was with mitigating thread divergence.

In the bitonic sort algorithm, within a sorting step, for each index X in the array, there is an element X + 2^j or X - 2^j that it may be swapped with. In the naive algorithm, we execute the swap step for every index in the array, but we don't want to run this for both indices in the pair, since that would negate the swap, so instead there is an if statement that prevents the higher index in the pair from running. This is fine in a correctness sense, but what it means is that the higher index in the pair essentially does no work in the kernel. Thus, if you go the naive route and allocate one thread for each index in the array, then half the threads allocated drop out almost immediately and do no useful work. Thus, finding a way for all the threads to do something, even if you allocate fewer threads, is important for performance.

The optimization I made was to only allocate half the threads in each block as I was otherwise. Before I allocated one thread index in the array, but in the optimization, I allocated one thread per index pair in the array. This sounds simple, but I had to figure out how to make sure that I can figure out how to construct a unique pair in each thread so that none of them overlap. I was able to do this by enumerating each thread and using a formula to determine the smallest number that hasn't yet been "assigned" to a pair yet. This is based on the specific internal step of the bitonic sort step you are on, as it is determined by how far apart the pairs are (determined by 2^j).


```
__global__ void bitonic_sort(int *gpuArr, int i, int j) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;

    int k = (idx / (1 << j)) * (1 << (j + 1)) + (idx % (1 << j));
    int xor_idx = k ^ (1 << j);
```

The third class of optimizations I did was with data movement.

Bitonic sort requires the array to be a power of two, however the program does not require the input size to be a power of two. Thus, the program must in some way pad the array to make the bitonic sort work as if it has a power of two number of elements. When transferring the data from the CPU to the GPU, one could naively pad the array on the CPU side and then transfer that to the GPU, and then when sorting is done, transfer the full array back and then remove the padding. However, transferring the array with the padding is wasteful because that data is not necessary to be transferred, as it could just be constructed on the GPU side. Thus, the optimization I made was to allocate a power of two array on the GPU side, memcpy the contents of the CPU array to the end of the GPU array, and then memset the start of the GPU array to 0's. Thus, when the sorting finishes, the 0's will still be at the start of the array. Then, we can memcpy the end of the array from the GPU side to the CPU side. This prevents unnecessary data transfer. One optimization that I played with was allocating pinned memory on the host side to speed up the data transfer on both ends. Pinned memory is memory that is preventing from being paged out. This seemed promising, but I was accidentally doing the pinned memory allocation outside of the timing window. When I transferred it to inside the timing window, it exposed that allocating pinned memory is a big time sink, so I discarded that optimization idea.

Possible next steps

Based on the Nsight analysis, I see that the shared kernels are much more compute bound than they are memory bandwidth bound. Thus, I should look into seeing what computation is actually happening and see whether any of it can be reduced in some way. I also see that the global kernel is very memory bound, which makes sense since the global kernel is not using shared memory so it could be saturating the global data bus. However, this could indicate that I am not coalescing memory correctly, and it could be that I could find techniques to access memory in a more efficient manner.