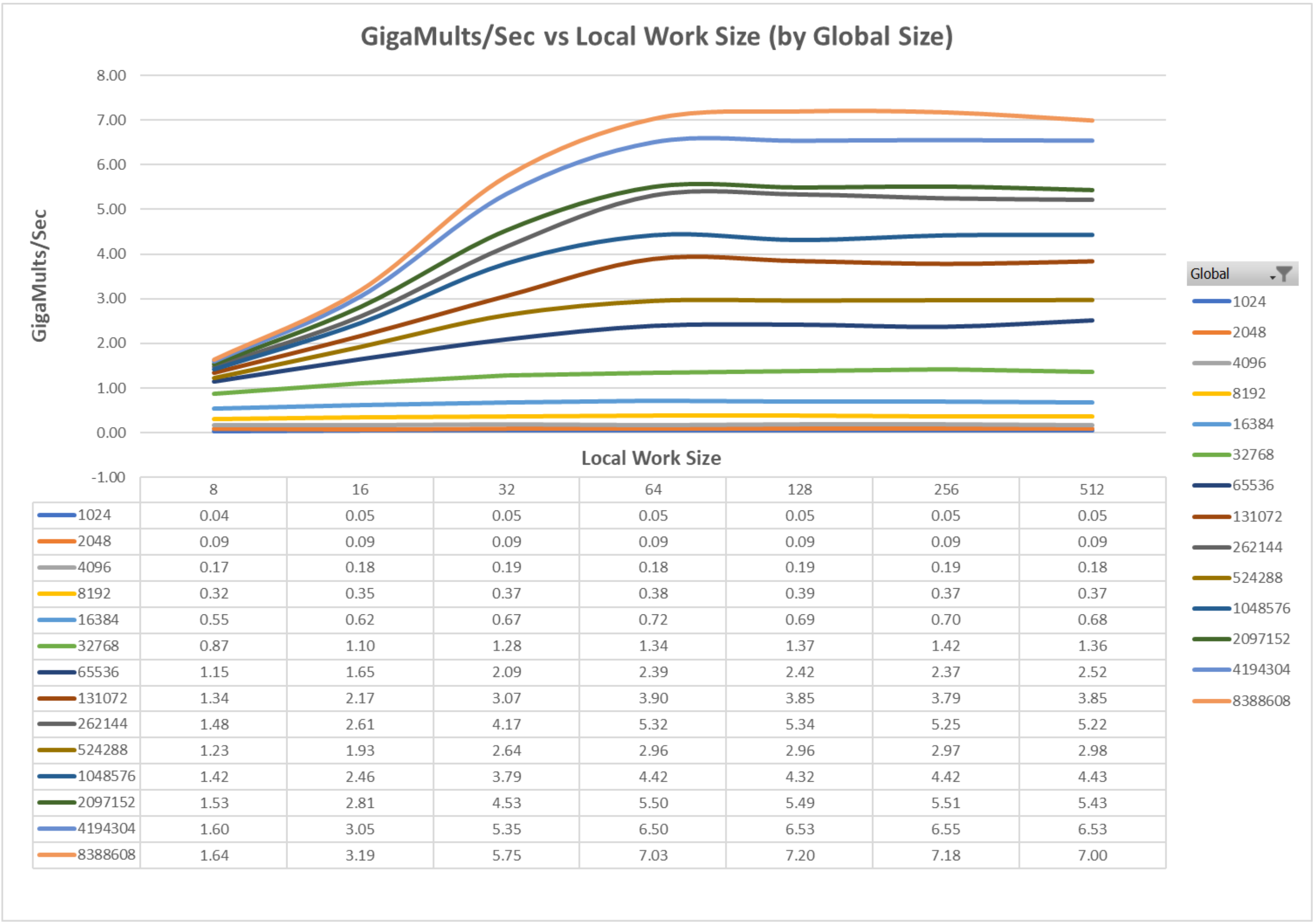
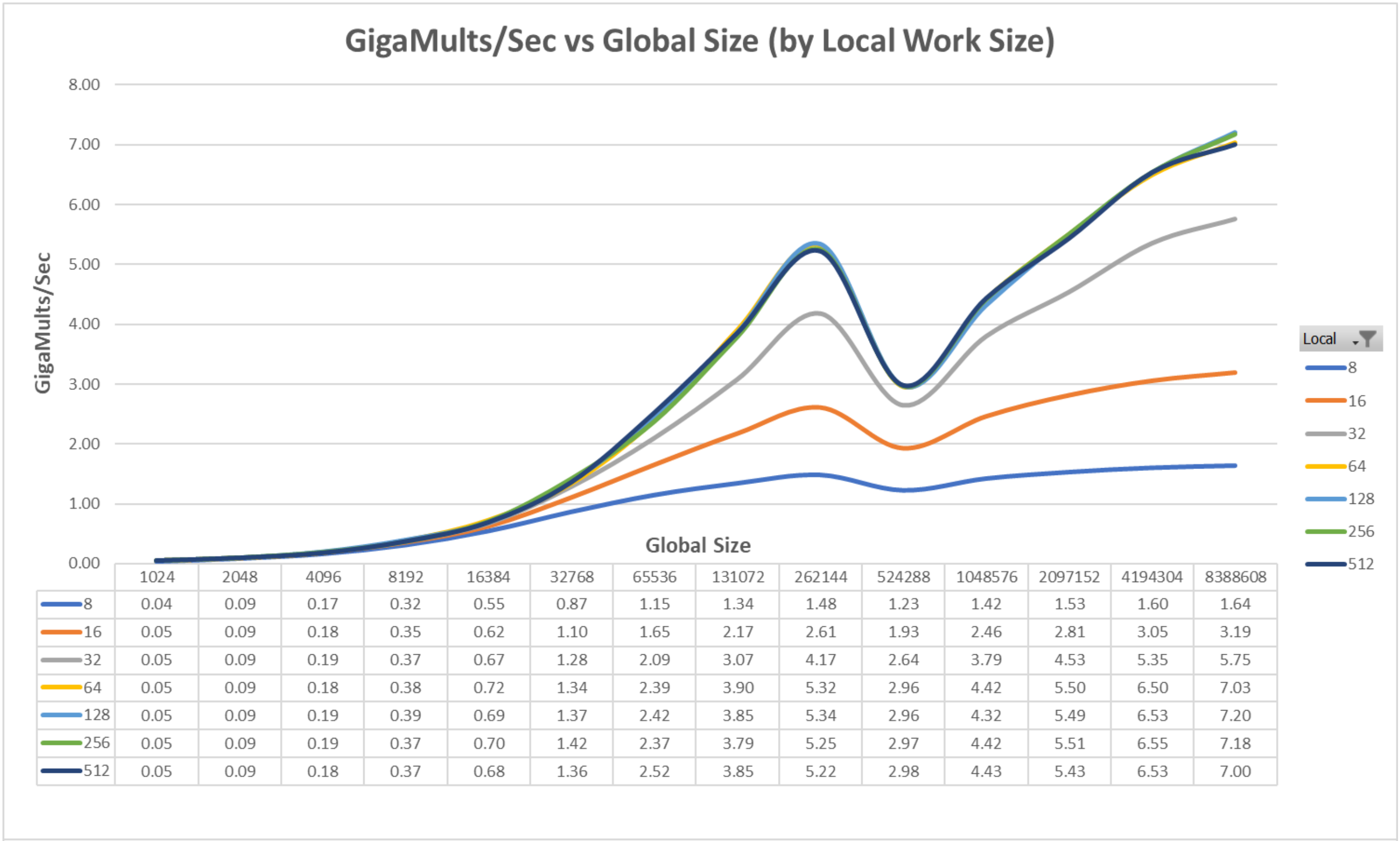
Clinton Hawkes
hawkesc@oregonstate.edu
CS475-400
Project #6

1. I ran this on my personal laptop:
   Lenovo X1 Extreme
   Intel Core i7 (6 cores/12 threads)
   32GB Ram
   Nvidia GTX 1650 Max Q
   Running Linux kernel 5.5.9

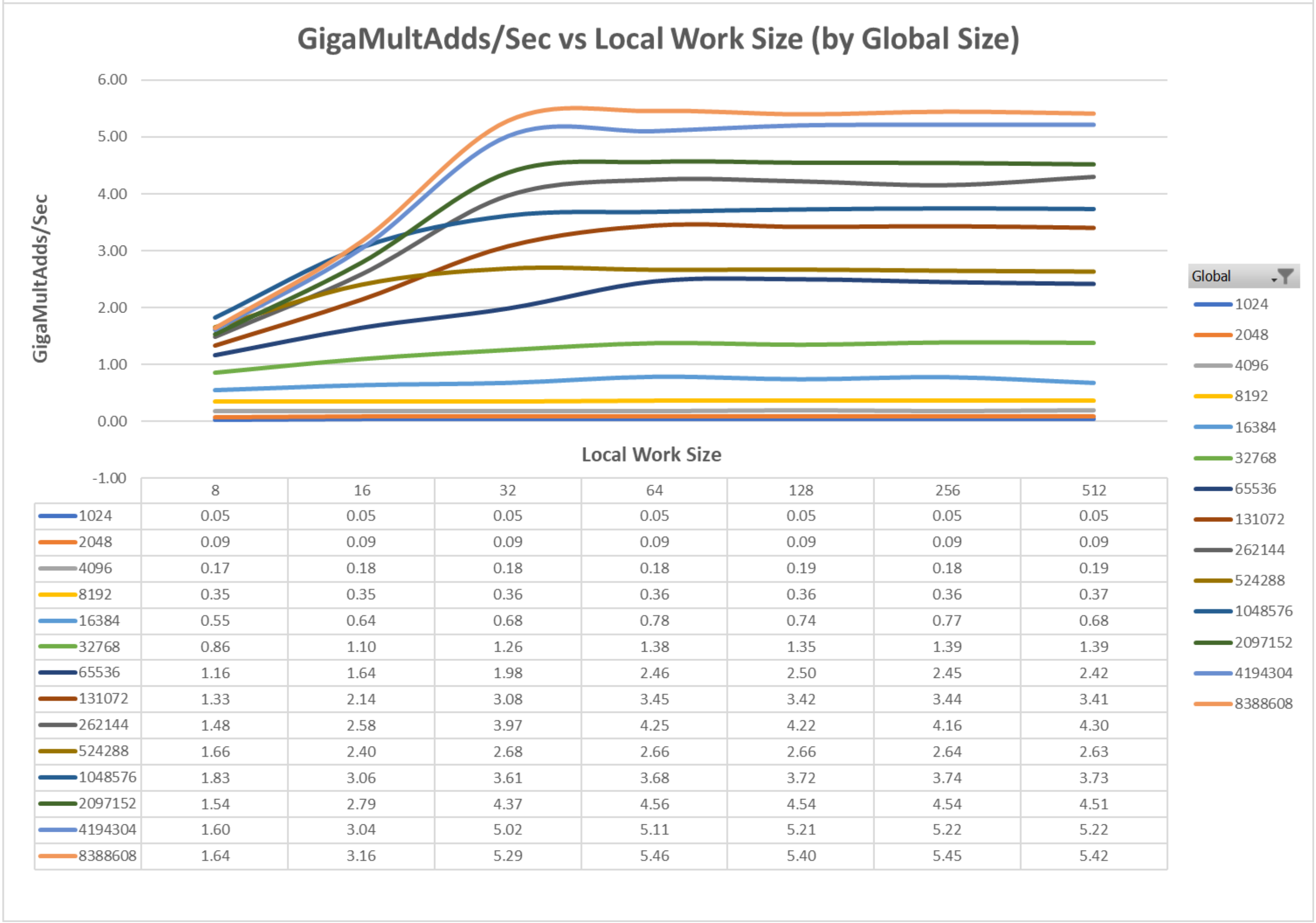
2. Here are the 4 graphs and charts for the multiply and multiply-add programs:



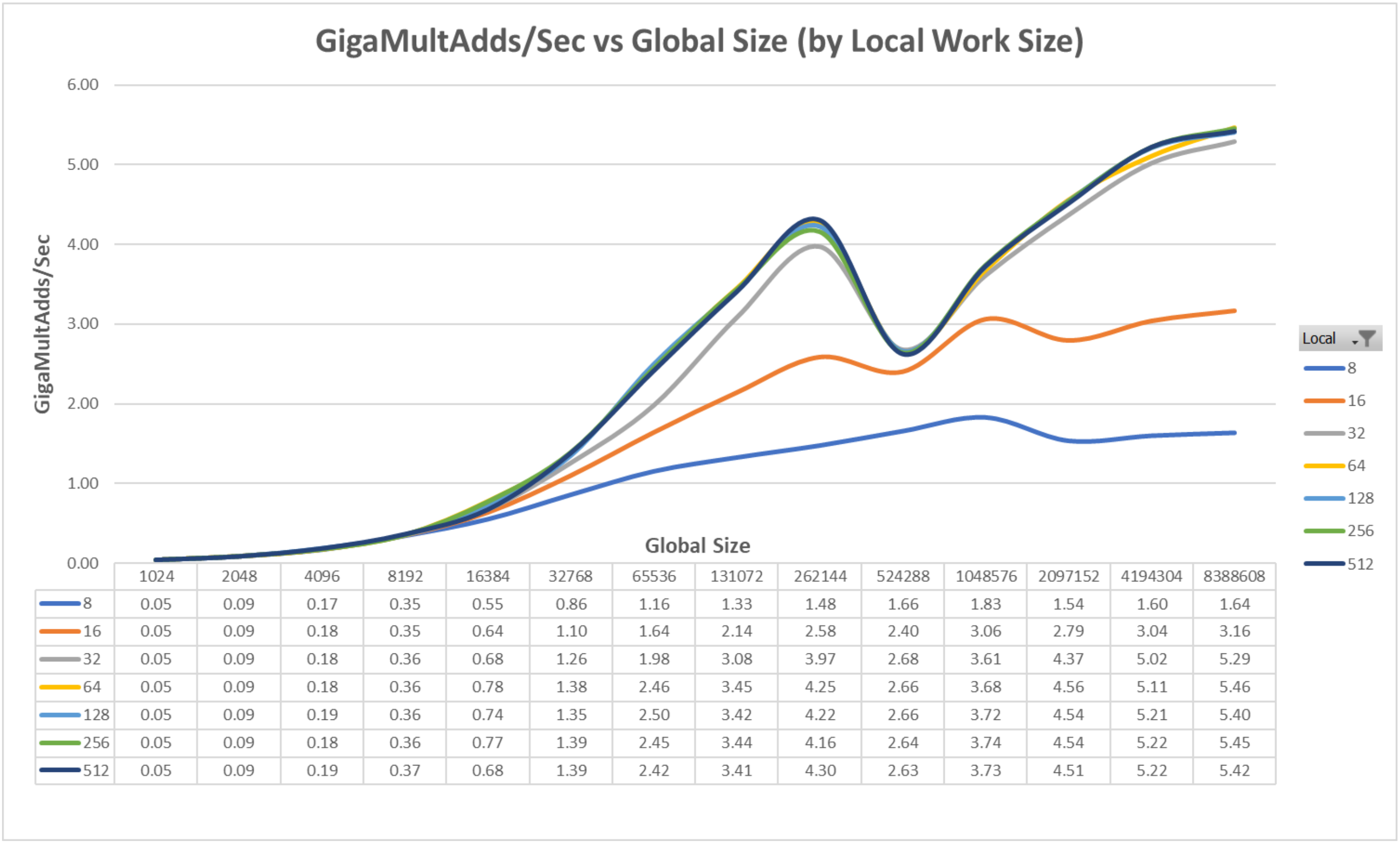
	8	16	32	64	128	256	512
1024	0.04	0.05	0.05	0.05	0.05	0.05	0.05
2048	0.09	0.09	0.09	0.09	0.09	0.09	0.09
4096	0.17	0.18	0.19	0.18	0.19	0.19	0.18
8192	0.32	0.35	0.37	0.38	0.39	0.37	0.37
16384	0.55	0.62	0.67	0.72	0.69	0.70	0.68
32768	0.87	1.10	1.28	1.34	1.37	1.42	1.36
65536	1.15	1.65	2.09	2.39	2.42	2.37	2.52
131072	1.34	2.17	3.07	3.90	3.85	3.79	3.85
262144	1.48	2.61	4.17	5.32	5.34	5.25	5.22
524288	1.23	1.93	2.64	2.96	2.96	2.97	2.98
1048576	1.42	2.46	3.79	4.42	4.32	4.42	4.43
2097152	1.53	2.81	4.53	5.50	5.49	5.51	5.43
4194304	1.60	3.05	5.35	6.50	6.53	6.55	6.53
8388608	1.64	3.19	5.75	7.03	7.20	7.18	7.00



	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608
8	0.04	0.09	0.17	0.32	0.55	0.87	1.15	1.34	1.48	1.23	1.42	1.53	1.60	1.64
16	0.05	0.09	0.18	0.35	0.62	1.10	1.65	2.17	2.61	1.93	2.46	2.81	3.05	3.19
32	0.05	0.09	0.19	0.37	0.67	1.28	2.09	3.07	4.17	2.64	3.79	4.53	5.35	5.75
64	0.05	0.09	0.18	0.38	0.72	1.34	2.39	3.90	5.32	2.96	4.42	5.50	6.50	7.03
128	0.05	0.09	0.19	0.39	0.69	1.37	2.42	3.85	5.34	2.96	4.32	5.49	6.53	7.20
256	0.05	0.09	0.19	0.37	0.70	1.42	2.37	3.79	5.25	2.97	4.42	5.51	6.55	7.18
512	0.05	0.09	0.18	0.37	0.68	1.36	2.52	3.85	5.22	2.98	4.43	5.43	6.53	7.00



	8	16	32	64	128	256	512
1024	0.05	0.05	0.05	0.05	0.05	0.05	0.05
2048	0.09	0.09	0.09	0.09	0.09	0.09	0.09
4096	0.17	0.18	0.18	0.18	0.19	0.18	0.19
8192	0.35	0.35	0.36	0.36	0.36	0.36	0.37
16384	0.55	0.64	0.68	0.78	0.74	0.77	0.68
32768	0.86	1.10	1.26	1.38	1.35	1.39	1.39
65536	1.16	1.64	1.98	2.46	2.50	2.45	2.42
131072	1.33	2.14	3.08	3.45	3.42	3.44	3.41
262144	1.48	2.58	3.97	4.25	4.22	4.16	4.30
524288	1.66	2.40	2.68	2.66	2.66	2.64	2.63
1048576	1.83	3.06	3.61	3.68	3.72	3.74	3.73
2097152	1.54	2.79	4.37	4.56	4.54	4.54	4.51
4194304	1.60	3.04	5.02	5.11	5.21	5.22	5.22
8388608	1.64	3.16	5.29	5.46	5.40	5.45	5.42



	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608
8	0.05	0.09	0.17	0.35	0.55	0.86	1.16	1.33	1.48	1.66	1.83	1.54	1.60	1.64
16	0.05	0.09	0.18	0.35	0.64	1.10	1.64	2.14	2.58	2.40	3.06	2.79	3.04	3.16
32	0.05	0.09	0.18	0.36	0.68	1.26	1.98	3.08	3.97	2.68	3.61	4.37	5.02	5.29
64	0.05	0.09	0.18	0.36	0.78	1.38	2.46	3.45	4.25	2.66	3.68	4.56	5.11	5.46
128	0.05	0.09	0.19	0.36	0.74	1.35	2.50	3.42	4.22	2.66	3.72	4.54	5.21	5.40
256	0.05	0.09	0.18	0.36	0.77	1.39	2.45	3.44	4.16	2.64	3.74	4.54	5.22	5.45
512	0.05	0.09	0.19	0.37	0.68	1.39	2.42	3.41	4.30	2.63	3.73	4.51	5.22	5.42

3. The patterns seen in the multiply and multiply-add programs are more similar than I thought they would be. Let's first take a look at the pattern when comparing performance to local work size.
   Both programs had low performance when the local work size was below 32. This was expected, due to the thread idle time that occurs when the local size is not a multiple of 32. When we look at the local sizes that are 32 and larger, we see the performance of both programs level off. This happens at different sizes for each program. The multiply program hits a performance ceiling at around a local size of 64, and then the performance remains approximately the same as the local work size increases.
   The multiply-add program hits a performance ceiling at around a local size of 32. The multiply-add program had a little more variation. Some of the curves didn't hit max performance until the local work size was 64. It did appear that the multiply-add program performed better than the multiply program at lower local work sizes.

   Moving on to performance vs global size, I see some pretty odd behavior. Overall, the performance of the curves increases as the global size increases, but there is a weird "kink" in the middle of the curves. The kink indicates a sharp drop in performance, which doesn't last long. This behavior appears for both multiply and multiply-add when the global size is 512K. After the "kink" ceases, there is a steady gain in performance as the global size increases.

4. I believe that performance increases as the global size increases, because there is more work to be done, and the GPU will be able to stay busy. When the global size is smaller, there is a larger portion of time spent on setup, which will reduce the performance. There is one exception though. There was a

weird kind in the curves when the global size is 512K. I honestly have no clue why there would be a kink right at this array size.

When looking at the local size for the multiply program, it looks like anything over and including 64 would be a good local size to have. 128 is probably the best size for the runs that I did.

When looking at the multiply-add program, it looks like anything over and including 32 would be a good choice. 32 was the best for some global sizes, while 64 was best for others. You would just need to look at the size of your global data to chose the best local size to maximize performance.
We can say with certainty that the local size should not be less than 32, and it should be a multiple of 32.

I have no explanation for the kink in the curves. I looked through the OpenCL output on my system, and the only thing that stood out was the global memory cache size is 512KiB. I don't know if that has anything to do with the behavior, but I would think that the GPU would use the cache efficiently so there wasn't an issue like we see here.

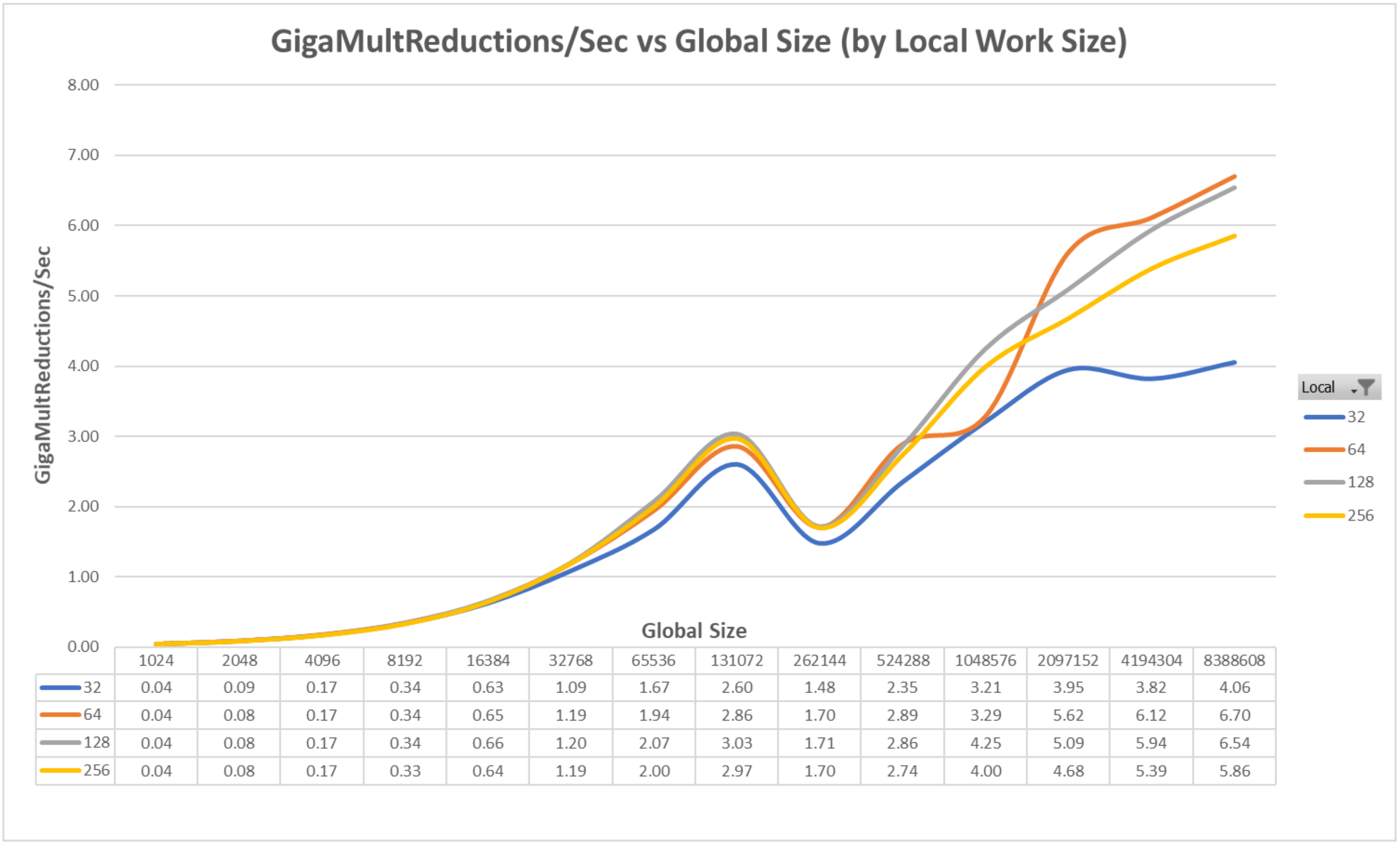
5. When looking at the max performance figures for each of the programs, I found that there is a 24% decrease in performance when going from the multiply program to the multiply-add. If you wanted to represent this difference of performance in a different way, you could say that there was almost a 31.9% increase in performance when moving from multiply-adds to the multiply program. Either way, you can clearly see that the GPU is able to perform multiplies faster than it is able to perform multiply-adds. This was the behavior I expected. When the GPU has to perform more instructions, there is most likely going to be a performance hit.

   One weird occurrence was the "kink" showing up in both programs when the global size was 512K, so both programs took a performance hit because of it.

6. If you have a dataset that can utilize the GPU for calculations, you will see huge performance gains. While the multiply program had higher performance than the multiply-add program, the performance seen using the GPU was much higher than what we saw when using the CPU. That is, if you setup your program correctly. We see the performance hit when the local work size is less than 32, and it may not be worth the time to setup the program if you have fewer than 16K elements. If you do it right, though, you will see much better performance figures.

Multiply-Reduction program

1.



	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608
32	0.04	0.09	0.17	0.34	0.63	1.09	1.67	2.60	1.48	2.35	3.21	3.95	3.82	4.06
64	0.04	0.08	0.17	0.34	0.65	1.19	1.94	2.86	1.70	2.89	3.29	5.62	6.12	6.70
128	0.04	0.08	0.17	0.34	0.66	1.20	2.07	3.03	1.71	2.86	4.25	5.09	5.94	6.54
256	0.04	0.08	0.17	0.33	0.64	1.19	2.00	2.97	1.70	2.74	4.00	4.68	5.39	5.86

2. This graph looks very similar to the other Performance vs Global Size graphs above.

   First, we see that the performance increases as the global size increases. Most of the time anyway. You can see that I experienced the “kink” again when running the program. The other two programs experienced the kink at a global size of 512K, but in this program it happened at 256K.

   Second, we see that performance was lowest when the local size was 32 and peaked at size 64. Performance slowly declined as the local size increased. This is not what we saw in the other two programs. The performance of the curves representing local sizes of 64 and up were much more tightly grouped. This shows that a local size of 64 is probably the best local size to choose for multiply reductions.

   I noticed that the performance of this multiply reduction program is just slightly below that of the multiply program. I thought there would be a bigger difference, similar to the difference I saw between the first two programs.

3. I think the performance increases as the global size increases, because there are more calculations to keep the GPU busy. This, in turn, increases the performance.

I don't have an answer for the "kink" that is present in the graph. I mentioned earlier that the global cache memory was 512KiB, but that doesn't really make any sense given that the kink appeared at 256K. It is just really odd that there is a sharp decrease for one global size. All other global sizes saw an increase in performance as the size increased.

I think this multiply-reduction program performed better than I expected, because the reduction harnesses the log(n) properties when summing the multiplied elements. This allows it to execute fewer instructions than it would take other methods of summing all elements in the multiplied array. I still expected multiply-reduction to be closer in performance to multiply-add.

4. Again, as explained in question 6 above, there are huge performance gains to be realized if you have a problem that can use GPUs to perform calculations. Not all problems can use GPUs, so this is something you need to determine. If your problem can use GPUs, you need to be sure to setup the program correctly to gain the most performance possible. This usually means having enough data to make it worth your while and choosing a local work sized that is larger than 32 and is a multiple of 32.

   It will take longer to create the program if you decide to use GPU processing, but if you have a lot of data and you need decrease the time spent calculating, it may be a beneficial tradeoff.