Jessica Spokoyny
June 1, 2017
Project #6
OpenCL Array Multiply, Multiply-Add, and Multiply-Reduce

1. <u>My Machine:</u>

I ran this program on rabbit in linux from a windows 10 laptop. My main functions are contained in 2 files called first.cpp and second.cpp. first.cpp takes care of multiply and multiply-add while second.cpp takes care of multiply-reduce.

I wrote scripts to automate each of these 3 processes in file called runProj6M.py, runProj6MA.py, and runProj6MR.py for multiply, multiply-add, and multiply-reduce, respectively. They can be compiled and executed by typing:

python3 runProj6M.py

python3 runProj6MA.py

python3 runProj6MR.py

2. <u>My performance results:</u>

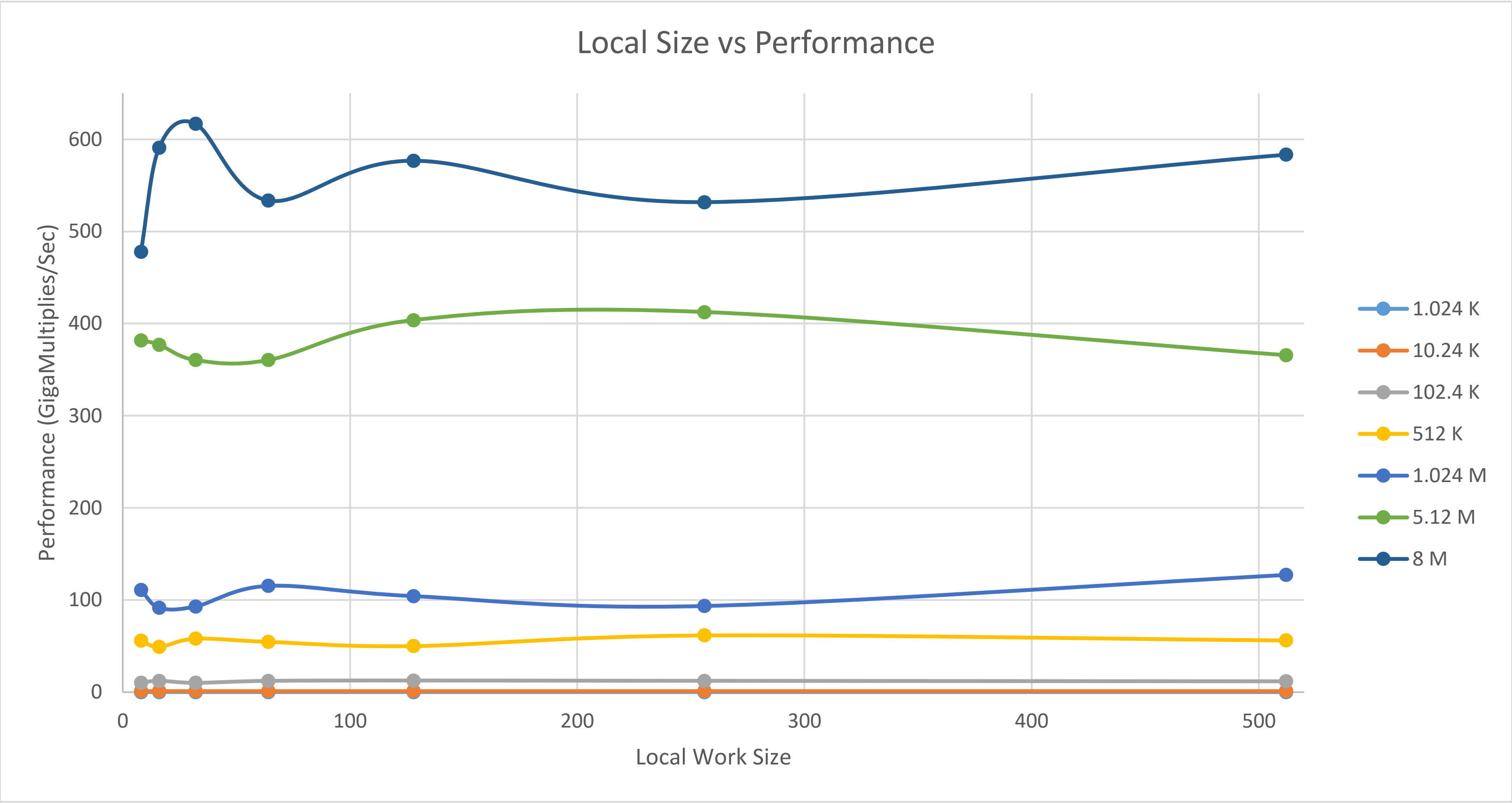
Array multiplication and Array multiplication + addition:

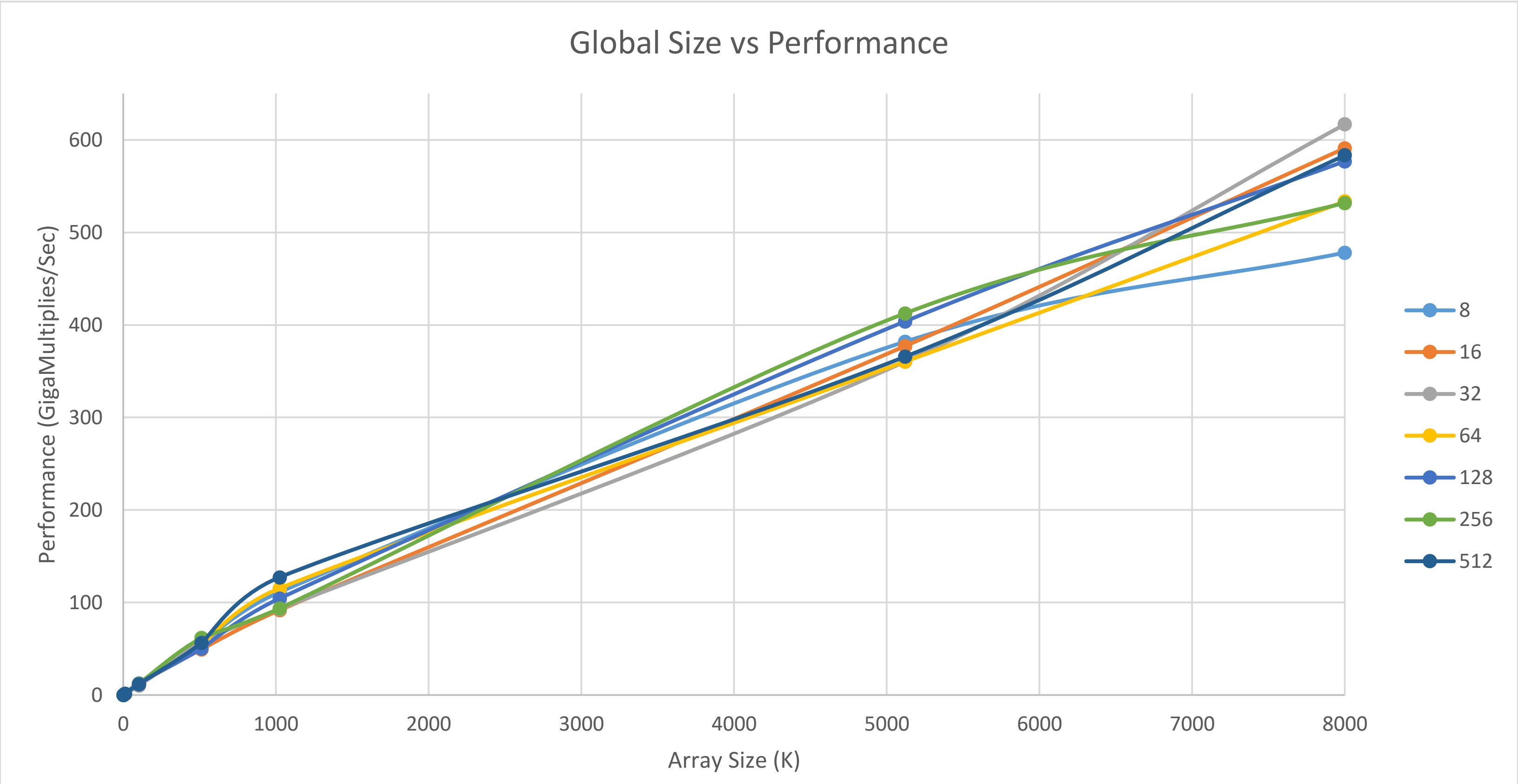
NUMBER OF ELEMENTS	LOCAL WORK SIZE	NUMBER OF WORK GROUPS	MULT PERFORMANCE (MegaMults)	MULT-ADD PERFORMANCE (MegaMult-Adds)
1024	8	128	136.880071	76.738605
1024	16	64	137.50502	128.208329
1024	32	32	139.300775	124.984726
1024	64	16	136.715622	133.559405
1024	128	8	118.628364	130.031777
1024	256	4	129.899787	128.352967
1024	512	2	127.315679	125.582506
10240	8	1280	1171.892955	1247.25978
10240	16	640	1244.984959	1051.118698
10240	32	320	1227.96518	1277.126499
10240	64	160	1259.532811	1255.517528
10240	128	80	1221.811366	1252.905426
10240	256	40	1187.108654	1238.509664
10240	512	20	1190.144125	1168.816512
102400	8	12800	10328.82818	11447.7356
102400	16	6400	12318.05628	10638.96045

102400	32	3200	10229.76967	11089.45462
102400	64	1600	12263.47318	10835.97976
102400	128	800	12615.501	10766.47998
102400	256	400	12298.82271	6771.590585
102400	512	200	11778.23798	10963.59594
512000	8	64000	55895.18824	52138.49376
512000	16	32000	49263.92914	55579.6848
512000	32	16000	58135.56741	52788.94374
512000	64	8000	54753.49821	31980.00733
512000	128	4000	50156.74292	49497.29433
512000	256	2000	61664.45218	51911.19846
512000	512	1000	56251.37802	53029.50223
1024000	8	128000	111123.1649	107157.818
1024000	16	64000	91690.55829	105404.0091
1024000	32	32000	93082.44257	107382.5799
1024000	64	16000	115354.2949	97878.05734
1024000	128	8000	104308.8486	104223.8968
1024000	256	4000	93644.25882	98433.13967
1024000	512	2000	127236.5745	102656.6462
5120000	8	640000	381804.5957	507986.9812
5120000	16	320000	377108.3796	510825.1507
5120000	32	160000	360461.8224	495356.0452
5120000	64	80000	360436.4354	528107.3328
5120000	128	40000	403753.6748	527617.5187
5120000	256	20000	412470.7918	502650.7617
5120000	512	10000	365688.1723	520430.9528
8000000	8	1000000	478040.0122	790279.5762
8000000	16	500000	591016.5244	790123.4022
8000000	32	250000	616950.6898	788177.2708
8000000	64	125000	533582.3331	713648.6061
8000000	128	62500	576826.0438	766136.7833
8000000	256	31250	531808.8603	809225.1633
8000000	512	15625	583388.0628	697228.5397

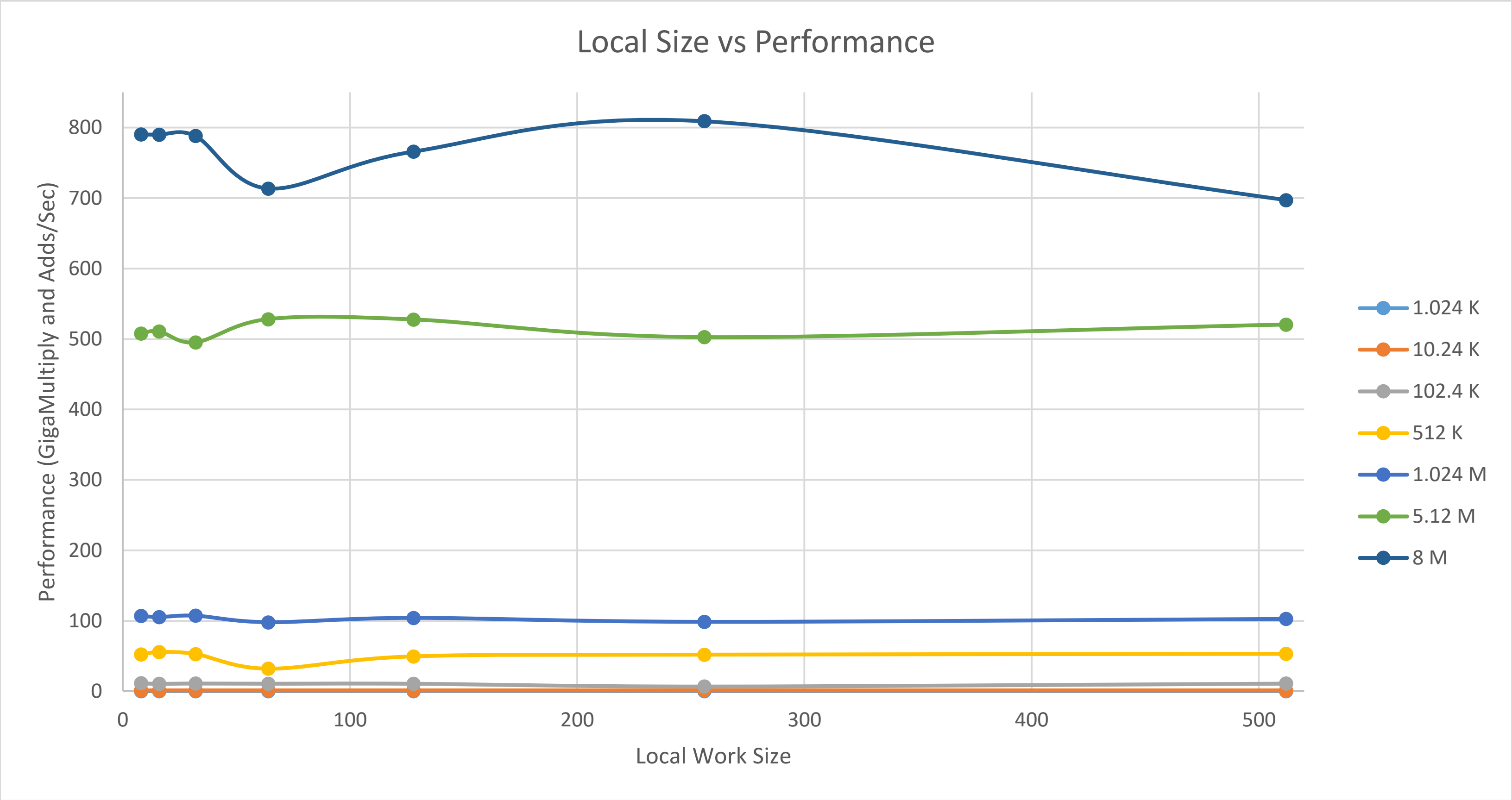
Graphs of Performance:

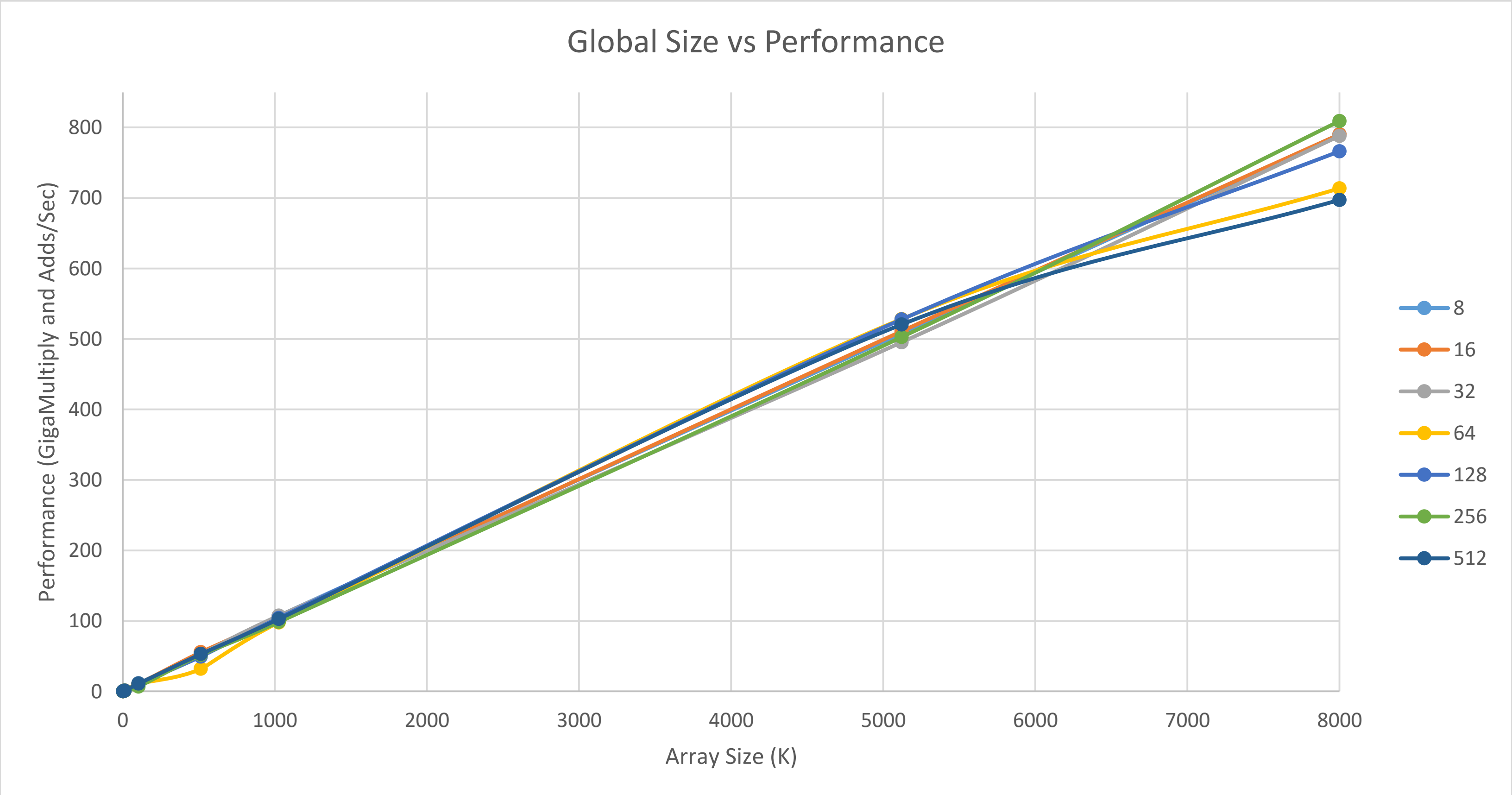
Array multiplication:



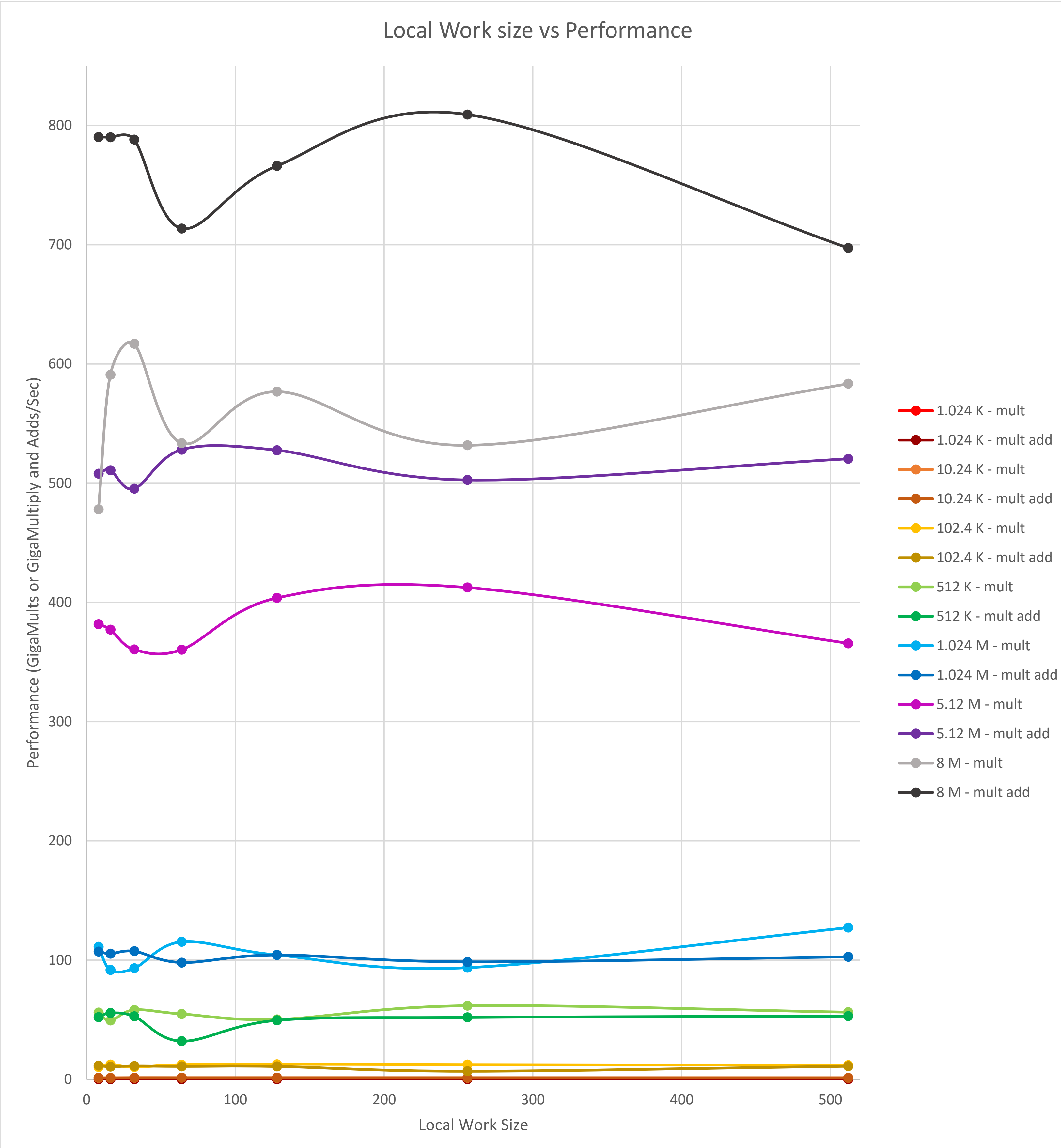


Array multiplication + addition:

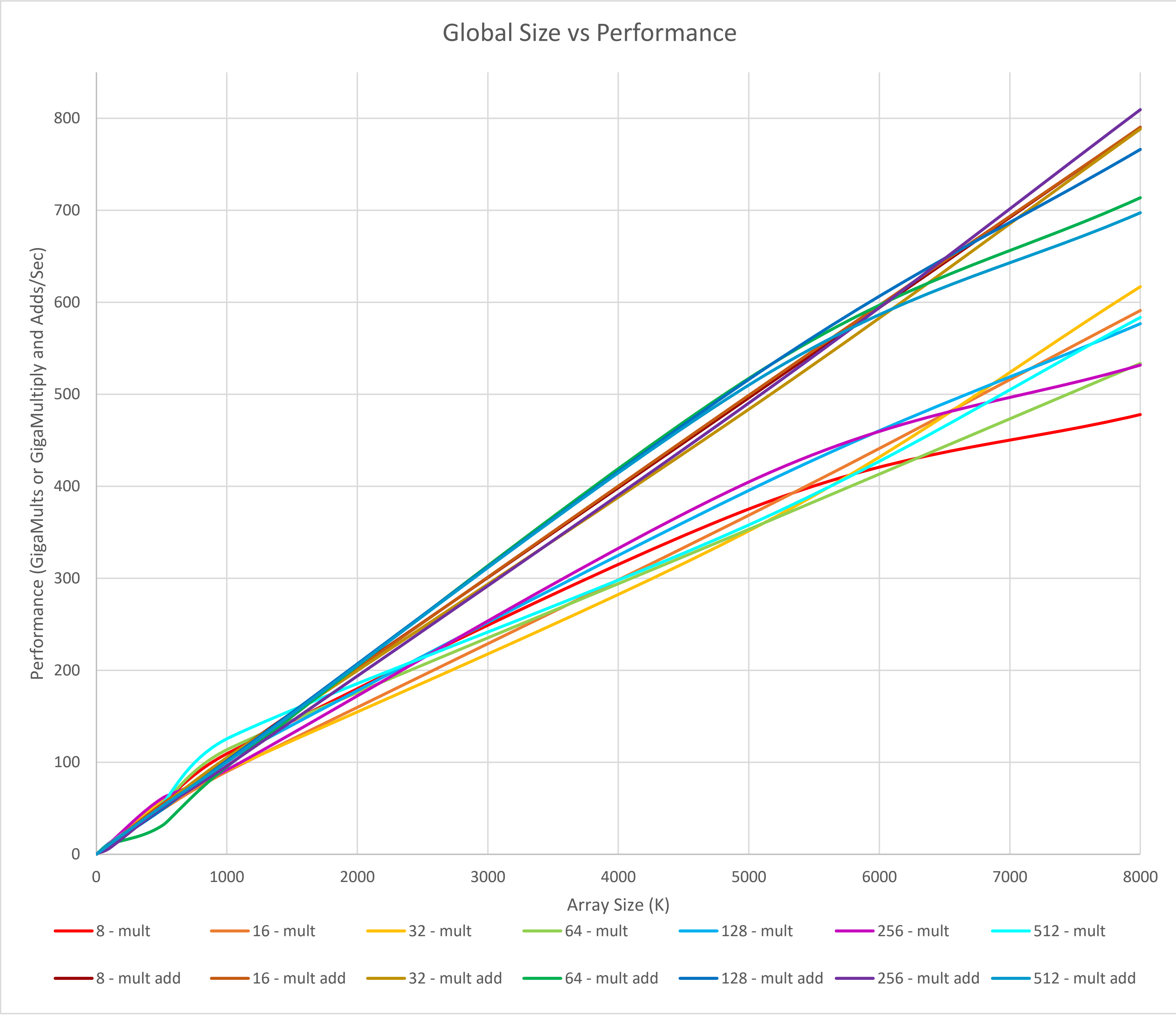




Multiplication vs Multiplication-Addition (local size vs performance):



Multiplication vs Multiplication-Addition (global size vs performance):



* local size vs performance graphs have colored curves to represent global work sizes

* global size vs performance graphs have colored curves to represent local work sizes

3. <u>Patterns:</u>

   Looking at the local work size vs performance graphs for both multiplication and multiplication-addition, we can see that performance stays pretty consistent across the work sizes for smaller array sizes. When the array size is 5.12M and 8M, we see a bit more fluctuation in performance, but still centered around a consistent performance figure.

   On the other hand, when we look at global work size vs performance, we see a clear upward trend in performance for both multiplication and multiplication-addition.

4. <u>Explanation of Patterns:</u>

   As the local work size increases, our performance stays stable because these problems are extremely parallel. If there were other components such as if-statements, we would see a difference in having a larger work group size so that there are more threads on each compute unit which could take over when other threads block.

   As the global work size increases, the smaller local work sizes start to top-out and go down a bit because there are more processing elements in each compute unit than the size of the local group. This is wasting a lot of compute time. The larger work sizes are better because we have more threads than processing elements so if some threads block, we have more to bring in. For smaller array sizes, it doesn't make sense because we don't get enough work done to overcome the overhead of setting it all up.

5. <u>Performance Difference for Multiply vs Multiply-Add:</u>
   For local work size vs performance, we see that the performances for multiply and multiply add were almost identical for array sizes <= 1.024M. For array sizes of 5.12M and 8M, we see a much larger gap in the performances with multiply-add performing better in both.

   For global work size vs performance, we see that the performances for multiply and multiply add started off almost identical for local work sizes of <= 1.024M. Then the multiply-add performances are consistently higher as the number of elements increases.

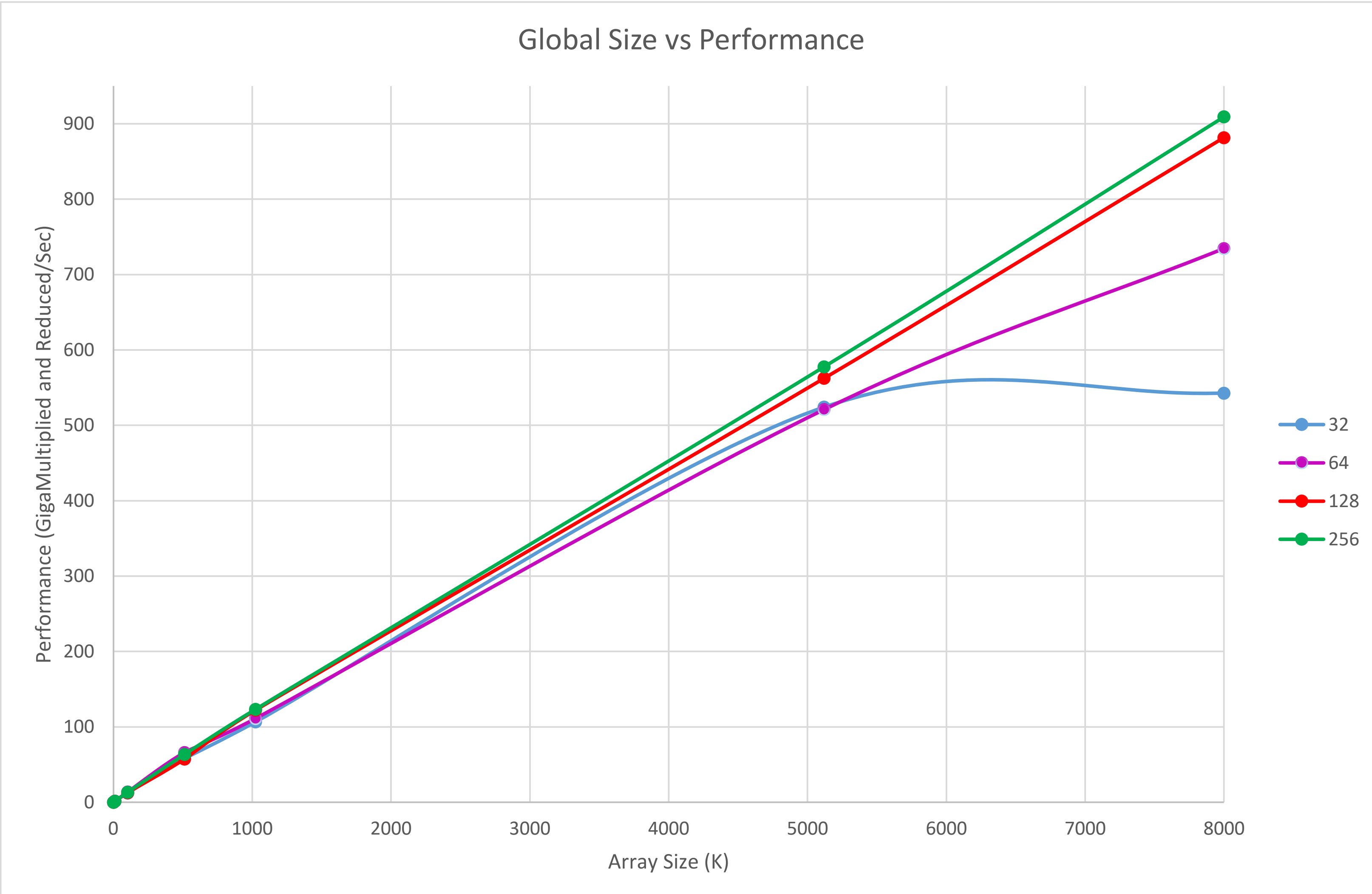
6. <u>Meaning for GPU Parallel Computing:</u>
   We can conclude from this comparison that the larger the global work size, the more valuable OpenCL is to boost performance (up to the maximum global work size the system can handle). For local work size, if the problem is very parallel, a small local size is pointless because the benefits don't compensate for the amount of overhead and a large local size would leave many threads idle without boosting performance.

7. Table and Graph

Array multiplication + reduction:

NUMBER OF ELEMENTS	LOCAL WORK SIZE	NUMBER OF WORK GROUPS	MULT-REDUCE PERFORMANCE (MegaMult-Reds)
1024	32	32	125.73672
1024	64	16	125.244605
1024	128	8	121.384544
1024	256	4	135.25298
10240	32	320	1343.127072
10240	64	160	1346.305136
10240	128	80	1321.290185
10240	256	40	1305.789065
102400	32	3200	12291.44076
102400	64	1600	13597.12868
102400	128	800	12722.07885
102400	256	400	13290.06682
512000	32	16000	57912.01461
512000	64	8000	65996.3847
512000	128	4000	57245.08049
512000	256	2000	63382.01761
1024000	32	32000	106600.0487
1024000	64	16000	111123.1649
1024000	128	8000	122224.8802
1024000	256	4000	123343.7933
5120000	32	160000	524160.4061
5120000	64	80000	520907.5084
5120000	128	40000	562451.8461
5120000	256	20000	577812.9196
8000000	32	250000	542777.7277
8000000	64	125000	734551.5343
8000000	128	62500	881736.7567
8000000	256	31250	909297.5019

Array multiplication and reduction:



8. Pattern:
   We can see in the graph above that there isn't much difference in the performances until the array size reaches around 5M. After this, the smaller local work sizes start to slow down and top-out but the larger ones continue to see performance increases.

9. Pattern Explanation:
   As the global work size increases, the smaller local work sizes start to top-out and go down a bit because there are more processing elements in each compute unit than the size of the local group. This means that we must access global memory which is wasting a lot of compute time. The larger work sizes are better because we have more threads than processing elements so if some threads block, we have more to bring in. For smaller array sizes, it doesn't make sense because we don't get enough work done to overcome the overhead of setting it all up.

10. <u>Meaning for Proper Use of GPU Parallel Computing:</u>
    As mentioned above, increasing global work size will have a positive effect on performance but will be much more valuable for larger local work sizes. Smaller local work sizes will top out earlier at a certain global size and increasing it further will not increase performance any more. It’s important to look at the local shared memory size and the global size.