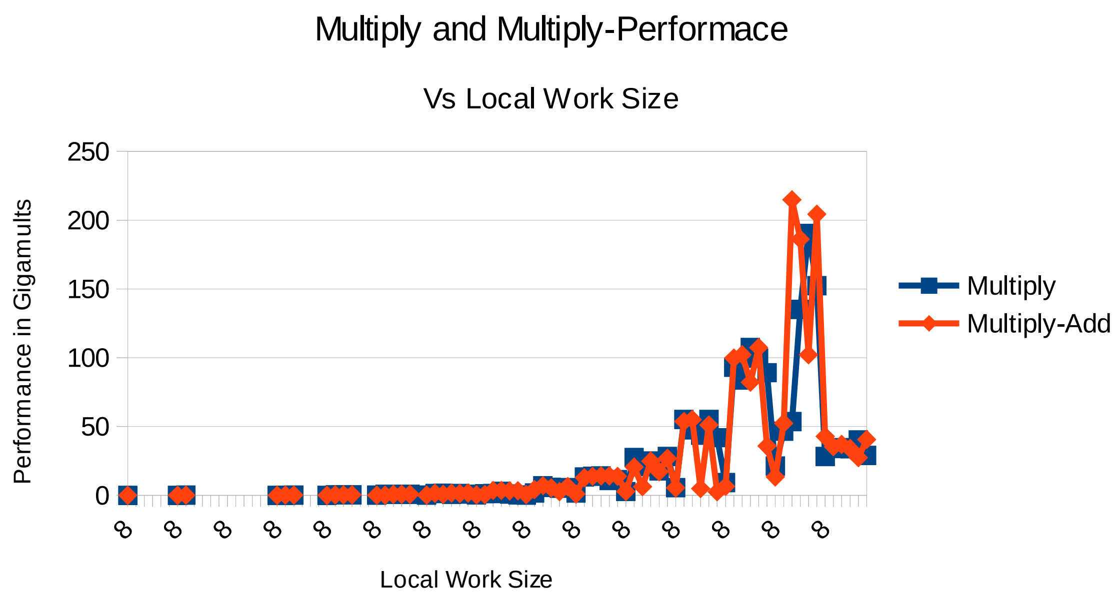
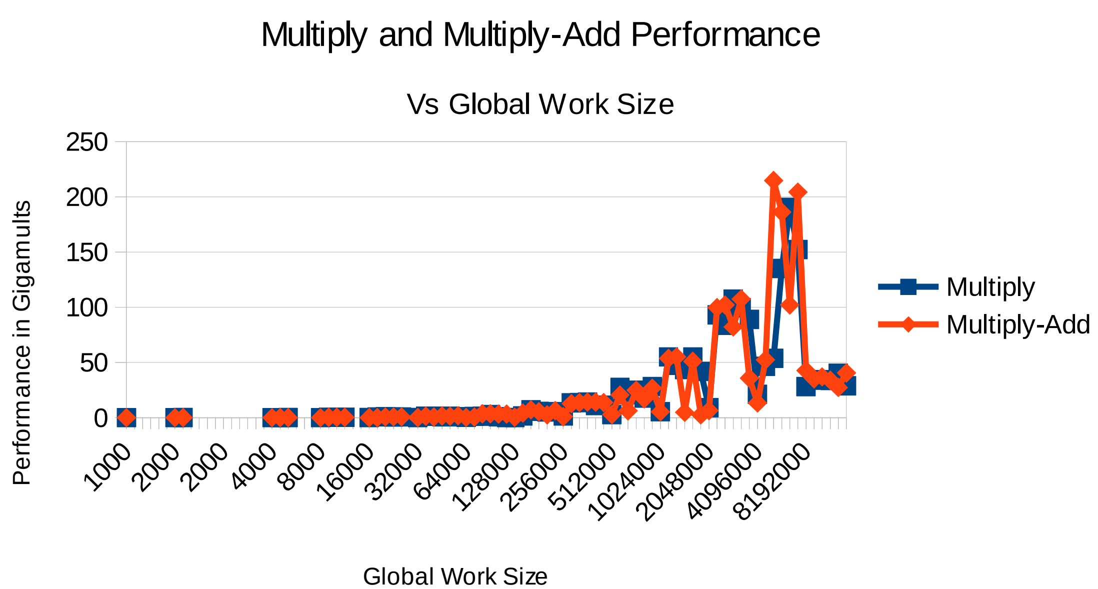
Travis Robinson
CS475
Spring 2016
Project 6

OpenCL Array Multiply, Multiply-Add, and Multiply-Reduce

This project was run on my home laptop, an HP Pavilion dv7 with an AMD A8-3520M processor and an AMD Radeon HD 6620G GPU. Due to machine limits, I wasn't able to perform benchmarks at a local size of 512, but was able to at 256.

Tables and Graphs





Global Data Size	Local Work Size	Number Work Groups	Multiply	Multiply-Add
1000	8	125	0.005	0.006
1000	16	62.5		
1000	32	31.25		
1000	64	15.625		
1000	128	7.8125		
1000	256	3.90625		
2000	8	250	0.01	0.012
2000	16	125	0.091	0.085
2000	32	62.5		
2000	64	31.25		
2000	128	15.625		
2000	256	7.8125		
2000	8	250		
2000	16	125		
2000	32	62.5		
2000	64	31.25		
2000	128	15.625		
2000	256	7.8125		
4000	8	500	0.02	0.024
4000	16	250	0.0215	0.178
4000	32	125	0.164	0.17
4000	64	62.5		
4000	128	31.25		
4000	256	15.625		
8000	8	1000	0.05	0.047
8000	16	500	0.39	0.409
8000	32	250	0.442	0.399
8000	64	125	0.431	0.409
8000	128	62.5		
8000	256	31.25		
16000	8	2000	0.092	0.088
16000	16	1000	0.818	0.106
16000	32	500	0.839	0.761
16000	64	250	0.617	0.798
16000	128	125	0.818	0.798
16000	256	62.5		
32000	8	4000	0.188	0.14
32000	16	2000	1.423	1.488
32000	32	1000	1.455	0.727
32000	64	500	0.761	1.596
32000	128	250	1.393	1.364
32000	256	125	0.829	1.769
64000	8	8000	0.306	0.296
64000	16	4000	1.247	0.601
64000	32	2000	1.322	3.538
64000	64	1000	2.846	3.538
64000	128	500	0.909	3.273
64000	256	250	0.2618	3.273
128000	8	16000	0.0663	0.542
128000	16	8000	1.657	4.223
128000	32	4000	7.076	7.076
128000	64	2000	5.818	5.455
128000	128	1000	5.134	2.645
128000	256	500	5.571	6.386
256000	8	32000	1.64	0.938
256000	16	16000	13.427	12.772
256000	32	8000	13.78	13.78
256000	64	4000	14.152	14.152
256000	128	2000	10.909	14.152
256000	256	1000	11.383	13.78
512000	8	64000	2.793	2.83
512000	16	32000	27.56	20.535
512000	32	16000	20.945	6.271
512000	64	8000	24.935	24.935
512000	128	4000	17.75	17.168
512000	256	2000	28.305	26.853
1024000	8	128000	5.6	5.395
1024000	16	64000	55.12	53.706
1024000	32	32000	47.603	55.12
1024000	64	16000	43.636	4.894
1024000	128	8000	55.12	51.086
1024000	256	4000	41.891	2.599
2048000	8	256000	9.227	6.485
2048000	16	128000	93.091	99.74
2048000	32	64000	83.782	102.173
2048000	64	32000	107.413	82.139
2048000	128	16000	99.74	107.413
2048000	256	8000	89.13	35.804
4096000	8	512000	21.264	13.362
4096000	16	256000	46.545	52.364
4096000	32	128000	53.706	214.825
4096000	64	64000	135.132	186.182
4096000	128	32000	190.413	102.173
4096000	256	16000	152.331	204.346
8192000	8	1024000	28.209	42.746
8192000	16	512000	33.989	35.277
8192000	32	256000	34.621	36.99
8192000	64	128000	33.85	33.989
8192000	128	64000	40.183	27.38
8192000	256	32000	28.99	40.572

Patterns

What we see in the performance curves is that we get optimal performance with large local and global work sizes. The reason we see optimal performance with large local sizes is that there are more parallel processes going on, making it possible for the GPU to perform more calculations at the same time.

The reason we see optimal performance with large global work sizes is that the sequential part of the program takes up less relative time. There is a time cost with enqueueing and then reading the results back; this will be relatively constant across data sizes, meaning it takes a smaller proportion of overall time for larger data sizes. So as the global size gets larger, the parallel portion of the program gets larger and the sequential portion gets smaller, giving us more computations per second.

Multiply vs Multiply-Add

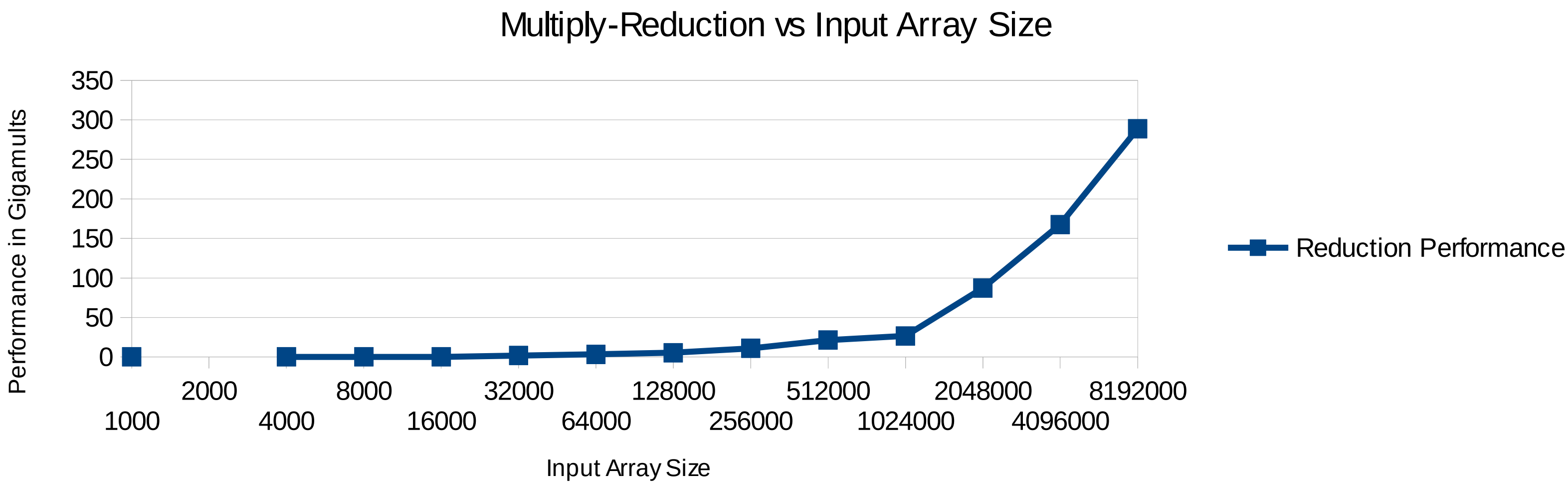
There was little difference between mltiply and multiply-add in performance. There was of course some variation, but not enough to say that one calculation was better than the other. The mean calculations per second for multiply was 25.52, and for multiply-add it was 26.34, with medians of 5.818 and 5.95, respectively.

What this means for proper use of GPU computing is that when multiplication and addition both need to be done, our best performance will come from doing them at the same time, rather than one then the other. Also, due to the way that multiply-add works, it also gives more accurate results, due to fewer rounding operations vs its sequential multiply then add counterpart.

In addition to that, we also see that GPU computing is most efficient with larger data sizes. Due to the large sequential portion of smaller data sizes, if the number of computations is too small, it's not worth it to use GPU parallel computing.

Table and Chart for Reduction Performance

Array Size	Local Work Size	Reduction Performance
1000	31.25	0.001
2000	62.5	
4000	125	0.022
8000	250	0.049
16000	500	0.066
32000	1000	1.769
64000	2000	3.273
128000	4000	5.343
256000	8000	10.909
512000	16000	21.373
1024000	32000	26.513
2048000	64000	87.273
4096000	128000	167.564
8192000	256000	288.903



Patterns

What we see in this curve is that as input array size increases, our performance also increases. The reason for this is that with reductions working in O(N) time, as N gets larger, the time increase gets smaller. In addition to that, like with multiplication and the fused multiplication-addition, with larger array sizes the sequential part (such as reading data to the GPU, getting results back, etc) becomes smaller and smaller relatively speaking, so that we end up with more calculations per second.

What this means for GPU parallel computing is that we get our best results with larger amounts of data, but with smaller amounts the sequential portion of processing makes it not worth while to do GPU computing.