Linge Ge - CS475
Project 6 - OpenCL Array Multiply, Multiply-Add, and Multiply-Reduce

Overview: Project 6 explores parallelization using the GPU via OpenCL. We do the classic array multiply, multiply-add, and multiply-reduce using the GPU processing power. OpenCL takes advantage of the GPU processors and memory buffer, does all the calculations, then simply returns the final result(s) back to the CPU after all parallel calculations are completed.

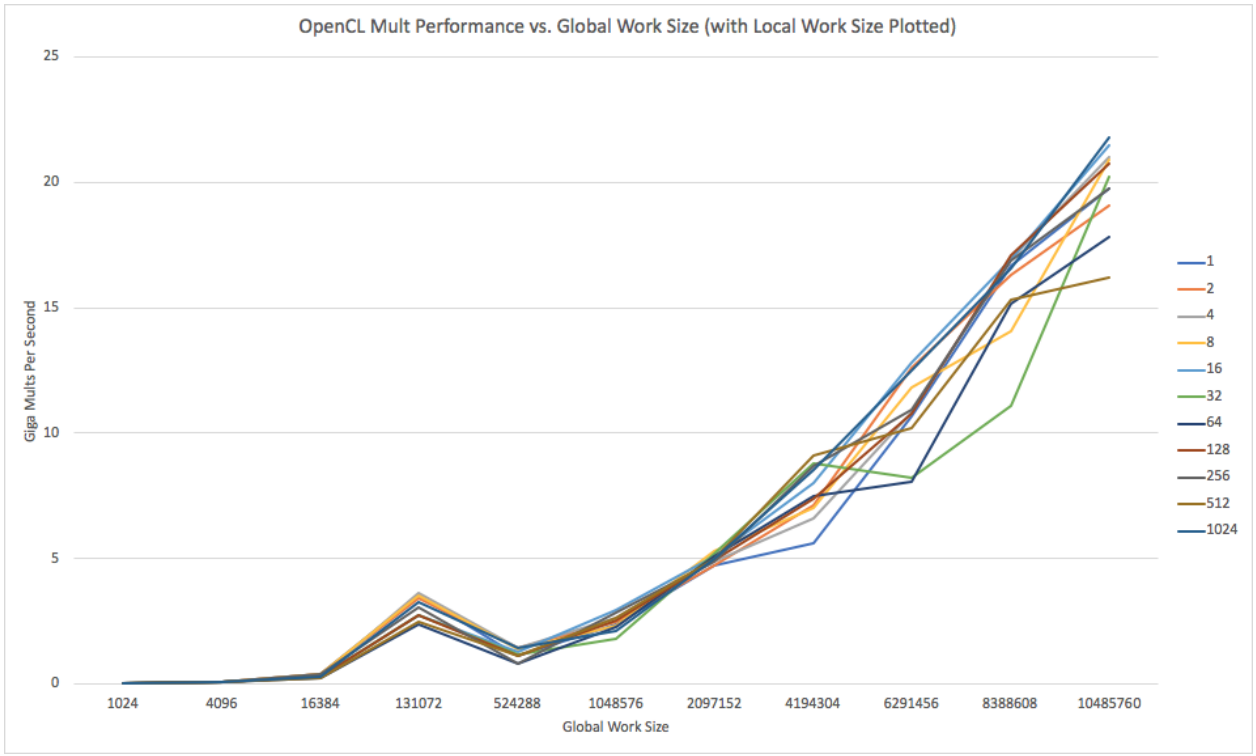
1. The code ran on:
    - Linux rabbit.engr.oregonstate.edu 2.6.32-696.3.1.el6.x86_64
    - 64GB RAM Memory
    - 16 CPUs @ Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
    - GPU: NVIDIA Titan Black - 15 SM, 2880 CUDA cores, 6GB memory

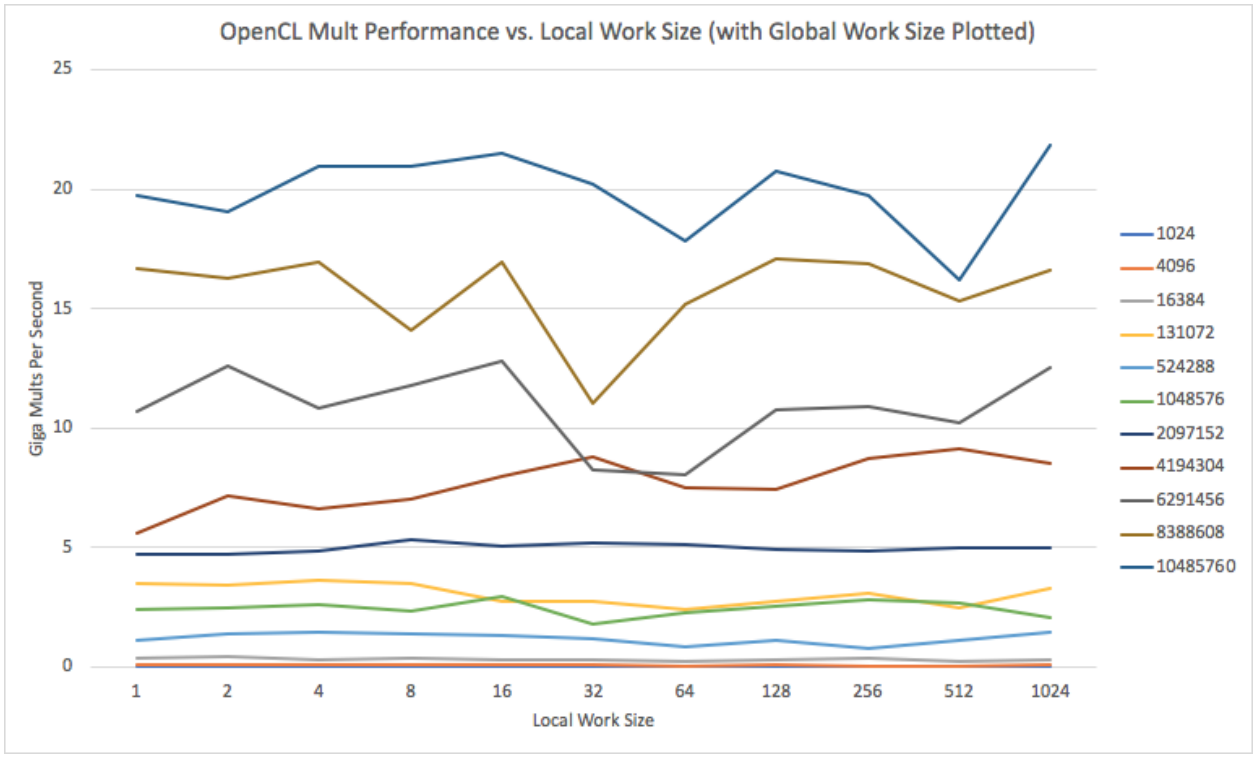
2.

Multiply:

- Performance Unit is **Giga Mults Per Second**
- Each data point is taken as the max GigaMults/s after 5 runs to minimize server noise that would reduce precision of the data

	Local Work Size										
Global Work Size	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
1024	0.025	0.025	0.019	0.021	0.02	0.025	0.018	0.028	0.019	0.014	0.021
4096	0.102	0.105	0.075	0.1	0.086	0.105	0.064	0.103	0.071	0.07	0.075
16384	0.411	0.416	0.285	0.37	0.289	0.291	0.247	0.283	0.392	0.251	0.309
131072	3.511	3.432	3.609	3.517	2.768	2.73	2.399	2.732	3.079	2.505	3.284
524288	1.134	1.431	1.458	1.373	1.298	1.16	0.828	1.105	0.809	1.134	1.456
1048576	2.389	2.49	2.621	2.32	2.927	1.826	2.275	2.555	2.834	2.664	2.106
2097152	4.724	4.731	4.862	5.307	5.071	5.207	5.117	4.911	4.888	4.984	4.962
4194304	5.606	7.141	6.618	7	8.002	8.82	7.506	7.409	8.718	9.127	8.519
6291456	10.682	12.589	10.838	11.803	12.82	8.228	8.059	10.785	10.913	10.223	12.507
8388608	16.679	16.296	16.922	14.089	16.951	11.07	15.152	17.072	16.875	15.295	16.572
10485760	19.761	19.083	20.984	20.921	21.487	20.229	17.816	20.764	19.729	16.203	21.814

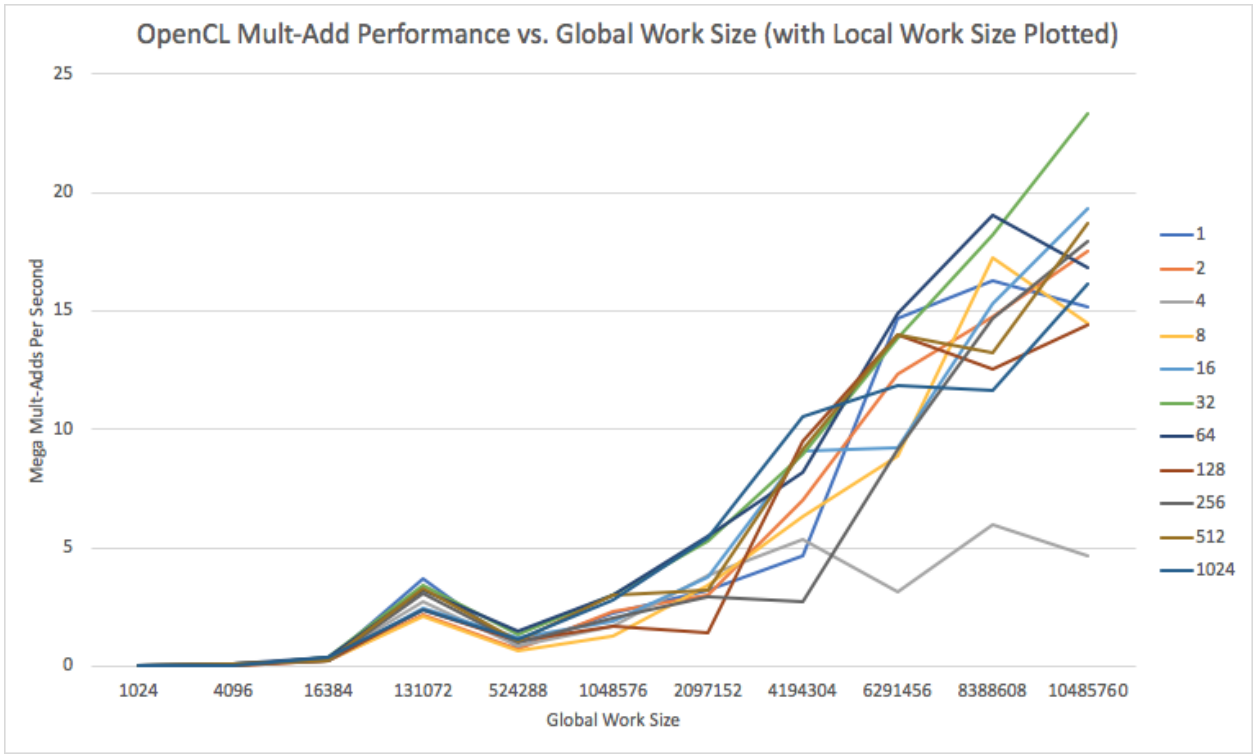

OpenCL Mult Performance vs. Global Work Size (with Local Work Size Plotted)
Giga Mults Per Second
25
20
15
10
5
0
1024
4096
16384
131072
524288
1048576
2097152
4194304
6291456
8388608
10485760
Global Work Size
1
2
4
8
16
32
64
128
256
512
1024

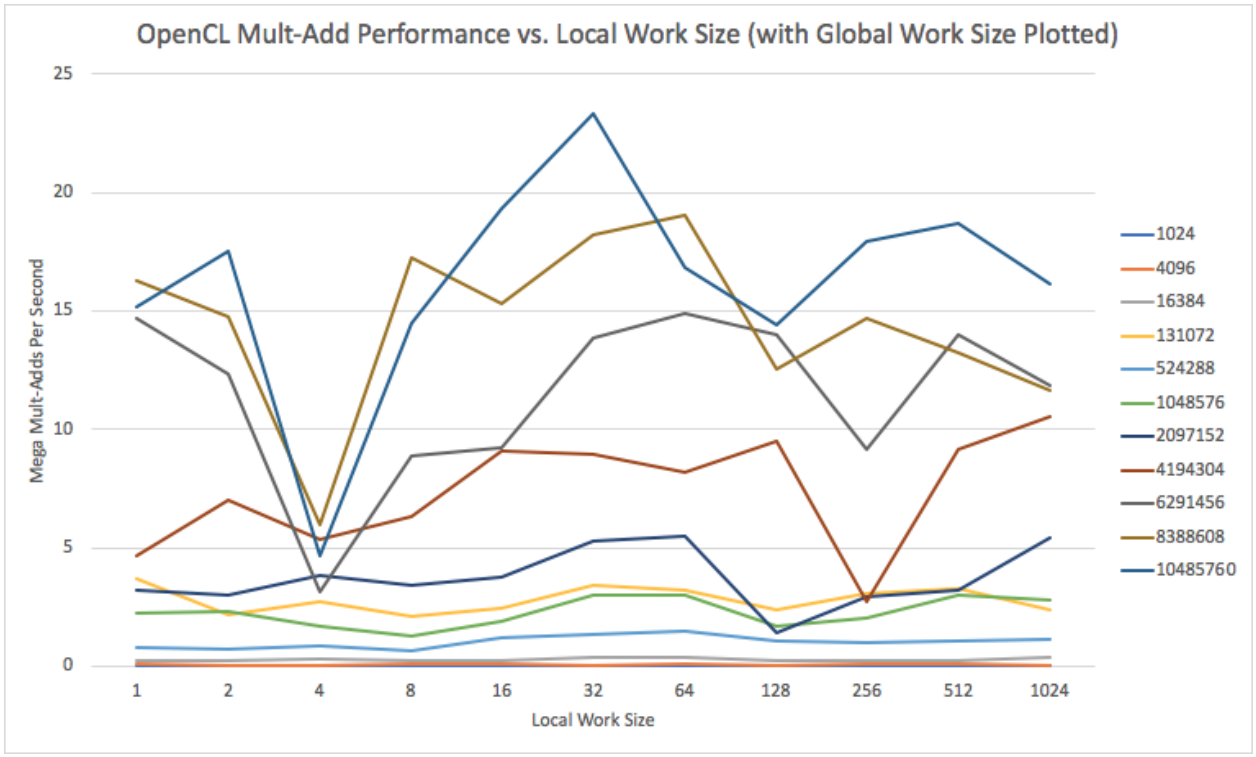

OpenCL Mult Performance vs. Local Work Size (with Global Work Size Plotted)
Giga Mults Per Second
25
20
15
10
5
0
1
2
4
8
16
32
64
128
256
512
1024
Local Work Size
1024
4096
16384
131072
524288
1048576
2097152
4194304
6291456
8388608
10485760

Multiply-Add:

- Performance Unit is **Giga Mult-Adds Per Second**
- Each data point is taken as the max GigaMults/s after 5 runs to minimize server noise that would reduce precision of the data

	Local Work Size										
Global Work Size	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
1024	0.019	0.02	0.018	0.02	0.019	0.015	0.024	0.017	0.026	0.015	0.022
4096	0.077	0.062	0.066	0.078	0.077	0.069	0.101	0.069	0.089	0.095	0.068
16384	0.269	0.227	0.29	0.259	0.242	0.354	0.398	0.264	0.28	0.255	0.376
131072	3.724	2.15	2.749	2.087	2.445	3.402	3.189	2.379	3.11	3.285	2.424
524288	0.825	0.744	0.892	0.682	1.198	1.341	1.471	1.058	0.992	1.102	1.118
1048576	2.255	2.314	1.675	1.284	1.88	3.002	3.046	1.709	2.062	3.002	2.771
2097152	3.244	3.045	3.808	3.415	3.791	5.284	5.487	1.388	2.94	3.252	5.429
4194304	4.679	7.007	5.387	6.326	9.095	8.954	8.198	9.512	2.746	9.172	10.551
6291456	14.704	12.361	3.118	8.883	9.249	13.838	14.914	13.98	9.171	13.976	11.877
8388608	16.25	14.792	5.991	17.249	15.287	18.187	19.075	12.551	14.675	13.25	11.66
10485760	15.152	17.539	4.697	14.475	19.302	23.327	16.829	14.416	17.913	18.694	16.108


OpenCL Mult-Add Performance vs. Global Work Size (with Local Work Size Plotted)
Mega Mult-Adds Per Second
25
20
15
10
5
0
1024
4096
16384
131072
524288
1048576
2097152
4194304
6291456
8388608
10485760
Global Work Size
1
2
4
8
16
32
64
128
256
512
1024


OpenCL Mult-Add Performance vs. Local Work Size (with Global Work Size Plotted)
Mega Mult-Adds Per Second
25
20
15
10
5
0
1
2
4
8
16
32
64
128
256
512
1024
Local Work Size
1024
4096
16384
131072
524288
1048576
2097152
4194304
6291456
8388608
10485760

3.

- For both Mult and Mult-Add, the performance is not very high before 1 million array size.
- For both calculation types, there is a spike of performance at 131,072 arrays.
- After 1 million array size, the performance greatly increased linearly for all local work sizes as global work size became bigger (more arrays).
    i. For Mult, all work sizes increased pretty consistently and at the same relative performance
    ii. For Mult-Add, the increase is consistent, but the rate is different. For example, local work size 32 had the highest performance at 10M array size, while local work size 4 did not have a significant linear increase.
- Different local work sizes does not seem to affect performance very much for smaller array sizes (for Mult, this would be <2M; for Mult-Add, this would be <131K)
- At large global array sizes, a run's local work size does affect performance
    i. For Mult, it seems performance is highest around local work size of 16 floats, with decrease in the 32-64 range, then a linear increase afterwards
    ii. For Mult-Add, there is a sharp decrease at 4, with strong performance peaks in the 32-64 range, followed by performance drops around 128-256.

4.

Data consistency may be a concern. The tests were run on rabbit and during seemingly high uptimes (~3-8). Despite the awesome 16 CPU cores on rabbit, unfortunately there's only so much GPU cores to go around, especially on larger array sizes, which would lead to execution queues on the GPU and inconsistent data. A good portion of inconsistent data was mitigated by re-running tasks 5 times and taking the peak value. Before doing this (just running once for each data value), I couldn't even get any sensible patterns at all.

For both calculation types, there was not much performance increase until the task was done on much larger array sizes (>1M). This totally makes sense, as the overhead for smaller array would eat the benefit of parallelism on the GPU using OpenCL. Overhead would include mapping addresses to appropriate GPU compute units for work-groups, assigning threads to work-items which must correlate to the correct processing element.

For both calculation types, there was a relative but sharp peak in performance at 131,072 array size, regardless of local work size. This was consistent among over dozens of runs. Around this global work size, the amount of overhead must be significantly reduced, hence taking the program a shorter amount of time to finish. The reason for reduced overhead is unknown, but likely revolves around the hardware specifications. The Titan Black has 30 CU with 128 PE each for a total of 3,840 PE. Likely a global work size of 100K allows the overhead algorithms to map and "slot" the workload more efficiently.

Overall, different local work group sizes do not affect performance too much. Different local work group sizes didn't affect smaller array sizes, and this makes sense, because given the high overhead and overshadowed parallelism, the effect of different local work group sizes just isn't significant. At larger global work group sizes, Mult and Mult-Add saw peak performance at around 16 and 32 local work group sizes, respectively. At these values, including 64, peaks are seen because of how threads are grouped. Nvidia groups 32 threads into a single group (warp), and optimization occurs when the number of work-items per work-group is a multiple of 32. The GPU wants as many warps to run simultaneously to hide any potential latencies in scheduling[(1)]. When we set local work group size to 32, that means a single warp can be mapped to exactly one work-group, with each thread running a work-item (one float). This leads to less overhead and scheduling latency, and we see this as a minor performance gain.

5.

Typically, a multiply-add should take longer than just a multiply, because of two reasons: there is an additional step of adding (1) and the added value must be retrieved from an additional place (whether it'd be on CPU or GPU, in cache, local memory or wherever). The retrieval, especially if it's local, and extra execution step may be insignificant, but over large data volumes there should be a noticeable difference in performance.

In my runs, there isn't a noticeable difference in performance between the two calculation types. This could be just inconsistent relative performance given external noise and having these two tests run at very different times of the day. However, another fact could be that the GPU performs a fused multiply-add instead of a typical two-step multiply then add. A fused multiply-add allows an express like a*b+c to be done in one single step, and would be equally fast as a multiply step.

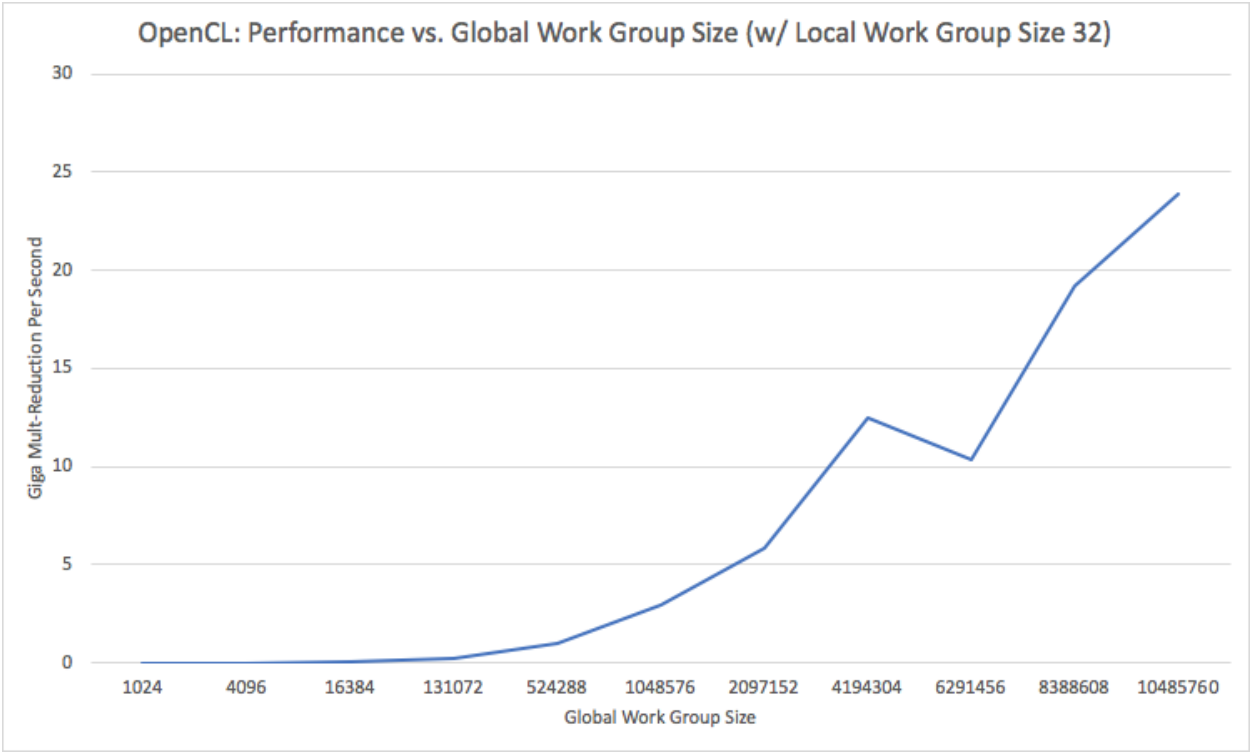
6.

GPU should be used when there are tons of simple, straightforward data processing to be done. The higher number of cores and threads allow GPUs to easily parallelize tasks. However, a GPU should not be used when there is irregular data structures or flow control involved, as it lacks the ability to do branch prediction or out-of-order logic. Essentially, a GPU should be used to do grunt work processing for large streaming data, that's worth the cost of a good amount of initial setup (setting up buffers, loading data over from CPU, etc.) and overhead.

Multiply-Reduction

1.

Num Work Groups	Local Work Size	Global Work Size	Giga Mult-Reductions Per Second
32	32	1024	0.003
128	32	4096	0.008
512	32	16384	0.029
4096	32	131072	0.24
16384	32	524288	1.027
32768	32	1048576	2.968
65536	32	2097152	5.872
131072	32	4194304	12.51
196608	32	6291456	10.381
262144	32	8388608	19.229
327680	32	10485760	23.857



2.

- Slight performance increases are seen before 1M array size. Afterwards, there is a very significant increase from 1M to 10M.
    - There is a small, relative performance dip at around 6M array size.

3.

There was not much performance increase until the task was done on much larger array sizes (>1M). This totally makes sense, as the overhead for smaller array would eat the benefit of parallelism on the GPU using OpenCL. Overhead would include mapping addresses to appropriate GPU compute units for work-groups, assigning threads to work-items which must correlate to the correct processing element.

There is a slight dip around the 6M array size mark in performance. After a few dozen runs, I am almost absolutely certain this is simply just an inconsistency, as other data runs do not have this dip (table + graphs don't show an alternative, better run because of limited rabbit usage enforcement).

4.

GPU should be used when there are tons of simple, straightforward data processing to be done. The higher number of cores and threads allow GPUs to easily parallelize tasks. This was the case for the multiplication-reduction, as it simply just involves straightforward data stream processing of endless numbers of multiplication. The thousands of threads on the GPU can divide the work evenly and use the same instruction (ArrayMultReduction) to process all the data.

If possible, if there's further but different work to be done, it can also be done on the same kernel with additional instructions, as was done with this project's mult-reduction. After multiplying, instead of sending the data back to the CPU, letting the CPU setup another kernel to do the reduction, the GPU went ahead and just continued to do the reduction straightaway after the multiplication. Of course, there was some "waste" in the sense that each thread went through the if-condition in the reduction loop even though at the end, only one thread's reduction mattered. However, this is still done simultaneously so there's no loss of speed (still O(log(n)), and this is *much* preferable and performant than starting another kernel or having the CPU do the reduction.

Thus, as long as there is no branching logic or irregular branching structures, and a large data stream, the GPU should be used as much as possible for maximum parallel computing and performance of a program.

1: http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf