Project #5
Vectorized Array Multiplication and Reduction

Haoxiang Wang; Student ID: 932359049

1. Machine Running the Test

I used school's server "flip" to run the test. At the time my test is running, the "uptime" is around 0.2.

2. Performance Results

I used Benchmarks to let the program running with different array sizes. The array sizes are set from 1000 to 32M, each size doubles the number of previouse one. The performance is set as mega-calculation per second and the problem size is the array size.

The tests are run with four different functions. Two functions are written with assembly language inside performing SIMD methods for multiplying and reduction sum. Another two functions are just simple functions do the same job as SIMD's functions using simple for loop. The following form is the result I got from the test.

The form is too long to show in one complete one, so I separate it to two parts:

	1000	2000	4000	8000	16000	32000	64000
SimdMul	134.31	238.41	385.47	557.07	724.22	842.11	967.92
NormMul	66.92	86.32	101.1	109.33	114.77	117.67	120.4
SimdMulSum	133.26	237.99	388.28	563.82	733.69	864.28	1047.78
NormSum	68.09	87.51	101.4	110.93	116.42	119.36	122.36

128000	256000	512000	1024000	2048000	4096000	8192000	16384000	32768000
937.1	932.2	954.93	1402.51	1298.66	1293.03	1280.32	1269.87	1219.56
120.55	120.23	123.41	201.43	203.38	221.65	225.51	227.58	225.12
1052.02	1026.82	973.19	1693.96	1643.77	1852.26	1861.45	1864.38	1855.98
122.39	122.39	125.71	204.97	206.32	226.09	229.31	231.04	229.14

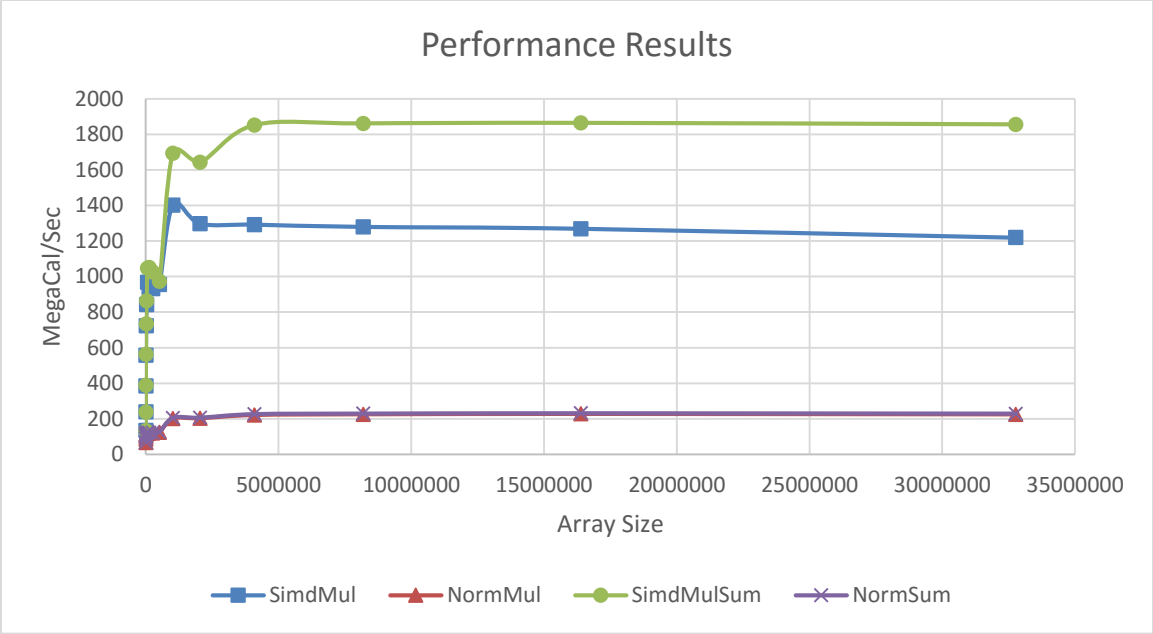
The graph based on the data is shown in the first figure in the next page.

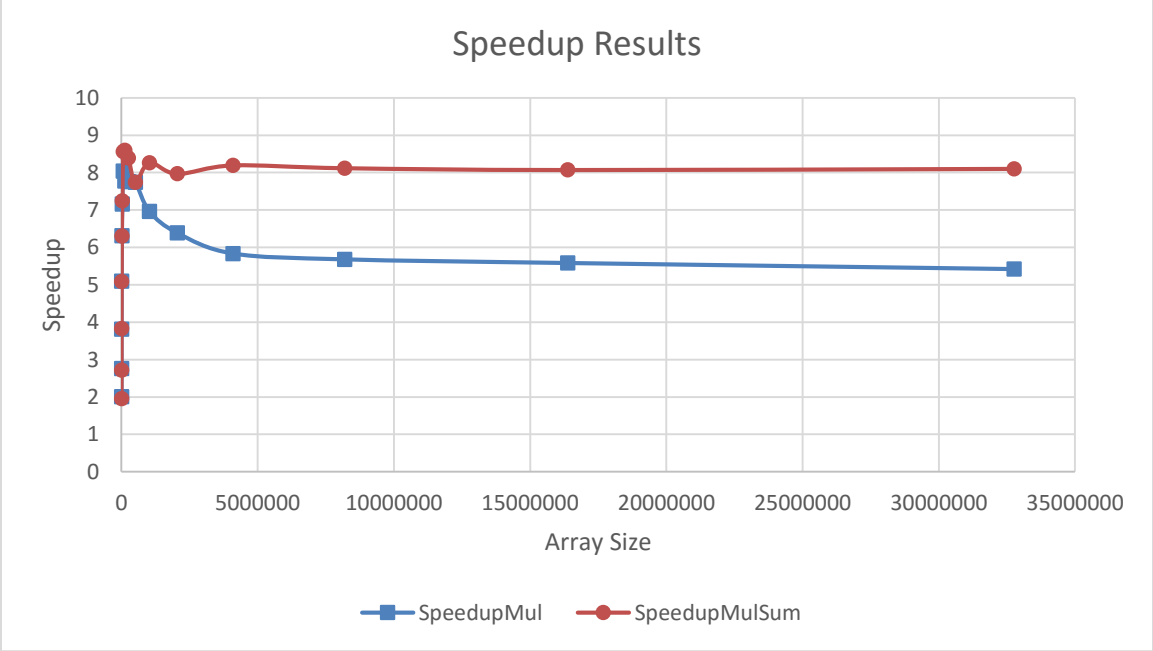
The speedup in between SIMD method and normal method doing same operation is calculated and shown in the following form:

	1000	2000	4000	8000	16000	32000	64000
SpeedupMul	2.007023	2.761932	3.81276	5.095308	6.310186	7.156539	8.039203
SpeedupMulS	1.957116	2.719575	3.829191	5.082665	6.302096	7.240952	8.563093

128000	256000	512000	1024000	2048000	4096000	8192000	16384000	32768000
7.773538	7.753473	7.737866	6.962766	6.385387	5.833657	5.677442	5.579884	5.417377
8.595637	8.389738	7.741548	8.264429	7.96709	8.192578	8.117614	8.069512	8.099764

The graph based on the data is shown in the second figure in the next page.





3. Behavior Explanation

3.1 Patterns

There are two curves in the speed-up graph. One colored blue is the speed up for comparing SIMD and normal methods on multiplying. Another one colored red is the speed up for comparing SIMD and normal methods on multiplying and reduction. Both of these two curves increase at the beginning, then once reach the limit, they stay at the similar value. The

multiply one has the speedup at around 5.5 and the multiply reduction has the speedup at around 8.

3.2 Analysis

As it mentioned in the previous section, both of these two speedup curves increase at the beginning since not reaching the performance limit. Once they meet the performance limitation, the speed up curves also consistent across a variety of array sizes.

3.3 Explanation

At the beginning of the graph, the performance and the speed-up increases since the SIMD performance doesn't meet the limitation within small array sizes. As the array sizes increasing, the SIMD performance keep increasing while the normal method keep performing around low number. Therefore, the speed-up keep increasing. Once the performance hits the limitation, the performance keeps in a stable range of number, so that the speed-up also keeps in a stable range.

The decreasing speed-up in the multiply case fits the "popped out" performance in the performance graph. This might relate to the little mistake occurs during the time catching. The big tendency of the speed-up is increasing first and then stay consistent.

3.4 Speed-up Explanation

The speed-up I got are both larger than 4.0. As it mentioned in the previous report, the multiply gets the speed-up around 5.5 and the multiply reduction gets the speed-up around 8.

I think the reason for these two larger-than-4 numbers is because the overhead for my normal method. The assembly language written in the SIMD method is so tight that it hardly has overhead. However, the for-loop I used in the normal method needs a lot "set up" work which creates the overhead. Also, the assembly language uses register to do the calculating, it does the work locally, which is performing much better than the normal methods I wrote. The normal methods I used in the code are not efficiency at all, this may also result in a high speed-up.