Linge Ge - CS475
Project 5 - Vectorized Array Multiplication

Overview: Project 5 takes a break from using any openMP pragmas for performance, and instead looks at the speedup achieved from using SSE SIMD vectors that can hold up to 4 floating points per vector array. The program compares the speedup achieved from taking advantage of SIMD at the assembly language level in array multiplication and array multiplication plus summing compared to doing it normally.

1. The code ran on:
    - Linux flip3.engr.oregonstate.edu 3.10.0-693.11.1.el7.x86_64 #1 SMP Mon Dec 4 23:52:40 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
    - 96GiB RAM Memory
    - 12 (hyperthreading x2) CPUs @ Intel(R) Xeon(R) CPU X5650 @ 2.67GHz

2. The **RAW DATA** table generated from the program. Performance is measured as MegaMults/second. The average is generated after 256 runs for each array size.

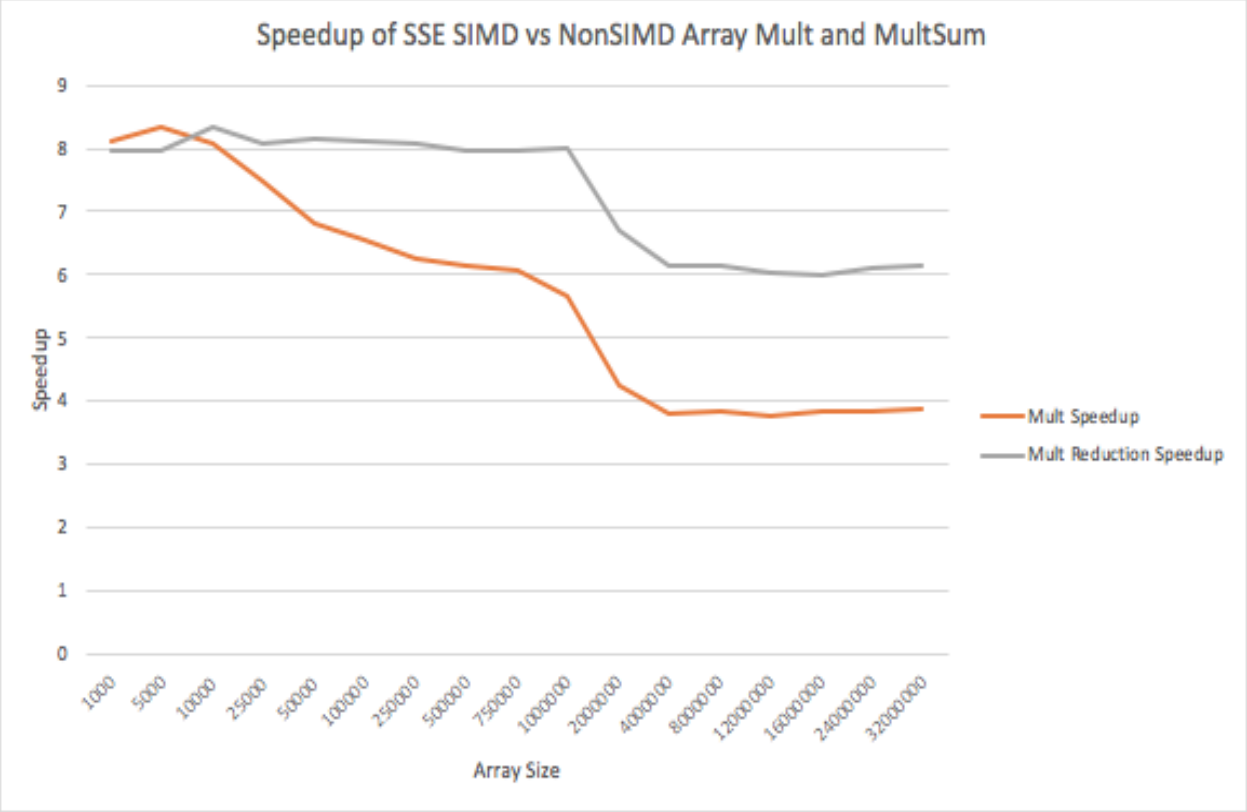
array_size	**nonSIMD Mult Peak**	**nonSIMD Mult Avg**	**SIMD Mult Peak**	**SIMD Mult Avg**	**nonSIMD Mult Reduct Peak**	**nonSIMD Reduct Mult Avg**	**SIMD Mult Reduct Peak**	**SIMD Mult Reduct Avg**
1000	119.961101	119.09349	971.052972	970.311429	121.715286	121.065559	967.117157	964.449793
5000	121.450955	120.569346	1010.675663	1002.867859	122.992368	122.579572	978.441611	972.648439
10000	121.088687	119.974158	979.334024	967.279645	123.213245	122.362562	1029.129078	1021.171346
25000	121.121742	120.044104	905.202222	886.648226	123.233467	121.52138	993.129502	980.001477
50000	120.727759	120.028505	820.480122	799.517285	123.205469	122.781786	1001.98469	994.951848
100000	120.58726	119.864321	792.173615	786.033142	123.229648	122.615268	1001.360951	990.281493
250000	224.034014	221.707221	1399.023086	1351.858276	235.208442	219.112963	1895.432787	1881.763146
500000	229.6	209.593	1411.0	1388.3	234.883032	234.0799	1868.64	1851.6

0	86411	845	21921	63834		58	9392	74221
750000	228.794697	205.586503	1389.058586	1304.369457	235.123132	233.935838	1876.191378	1860.175325
1000000	228.06322	211.268548	1289.843105	1171.175632	235.093214	233.939814	1881.498962	1859.8852
2000000	225.757408	216.630177	959.66715	921.406256	231.880696	230.085551	1553.638389	1464.967622
4000000	225.712093	214.171558	861.484701	812.702464	231.74959	231.189435	1424.392905	1343.032774
8000000	225.490285	221.533296	870.569884	803.788619	231.721887	231.097431	1425.221602	1216.114117
12000000	225.476521	223.918032	852.13243	809.014786	231.74719	231.272708	1396.094448	1311.157884
16000000	225.416315	223.259269	863.283189	825.401672	231.91295	231.45667	1392.779827	1259.577656
24000000	227.230148	226.126981	873.335432	821.602122	231.807895	231.242016	1418.268454	1257.412939
32000000	225.88442	224.770722	876.626276	840.431791	231.876878	231.178676	1426.093984	1340.140464

The **REPORT** table with **Speedup** calculated using peak performance:

array_size	**Mult Speedup**	**Mult Reduction Speedup**
1000	8.094732075	7.945732938
5000	8.321677364	7.955303462
10000	8.087741706	8.352422485
25000	7.473490779	8.058926898
50000	6.796118215	8.132631596
100000	6.569297743	8.125974287
250000	6.244690532	8.058523626

500000	6.143253817	7.95565936
750000	6.071200968	7.979612053
1000000	5.655638401	8.003204048
2000000	4.250877783	6.700162695
4000000	3.816741449	6.146258576
8000000	3.860786659	6.150569635
12000000	3.779251277	6.024213057
16000000	3.82972807	6.005614723
24000000	3.843395956	6.118292278
32000000	3.880862062	6.150220739



3. Patterns:
    - As the array size grows, speedup slightly decreases for both functions, with a dramatic decrease when the array size is between 1M and 4M, and afterwards, the speedup is more or less consistent.
    - Speedup for SIMD Multiplication Reduction is consistently greater than SIMD Multiplication at larger arrays (>50000)
    - With the exception of SIMD Multiplication speedup at very large array sizes (>4M), all other Speedup is >4.

4. The speedup is not necessarily consistent across the variety of array sizes:
    - With array size less than 1M, speedup is remarkably high, although we see SIMD Mult gradually decreasing as the array size gets bigger.
    - Between size 1M and 4M, the speedup drastically reduces for both functions.
    - For size 4M and onwards, the speedup is relatively consistent for both functions.

5. After some research, the main reason why we see a decrease in Speedup as array size grows bigger is due to the limited cache size. On the flip server, there are three levels of CPU cache:
    - L1d cache: 32K
    - L1i cache: 32K
    - L2 cache: 256K
    - L3 cache: 12288K

    Since there is no pre-fetching involved (although my understanding is that L3 cache utilizes some pre-fetching at the hardware level), utilizing spatial coherence and having data already in cache to be processed is important to performance. Having a giant L3 cache line size reduces the frequency to fetch data from memory. The L3 cache can feed to the L2 and L1 cache directly to the processor. For smaller array sizes, the data can be stored directly on the cache ready to be fed to the processor, reducing the number of cache misses. However, around 1M-4M array sizes, where we see the dramatic reduction in Speedup, the L3 cache no longer has enough space to hold array data, resulting in more cache misses and the need to grab data from main memory.

6.
    - The biggest performance hit that explains why SIMD array multiplication is <4.0 is the cache misses that are incurred, especially at larger array sizes, especially when we consider that there's no pre-fetching involved, and for SIMD, it's calculating four values at once. At this rate of processing, it's very likely the CPU has a much higher chance waiting for data to come from memory compared to SISD.
    - It's much harder to explain why SIMD array multiplication may be >4.0, especially in this project, where we see the Speedup peak at 8.5 for smaller array sizes! One weak explanation is just bad data: we capture SIMD values at better peaks and non-SIMD values at worse peaks, and hence got a Speedup of >4.0. A much

better reason would be focused on the fact that the compiling non-SIMD instructions may have some overhead for each multiplication we do in the loop. In this project, the SIMD multiplication function is *very* efficient, as it's directly written in assembly. The non-SIMD function is not. There may be very well be additional assembly execution steps involved for each calculation that the compiler adds. This would reduce the performance of the non-SIMD function from it's theoretical best, which leads to a much greater Speedup > 4.0 when it's compared to its SIMD assembly-wrapped equivalent.

7.

- The reason for the Speedup being <4.0 for SIMD multiplication add would be the same as for SIMD multiplication: the CPU has a much higher chance waiting for data to come from memory compared to SISD, resulting in higher cache misses and reduced performance.
- I believe the reason why the Speedup can be >4.0 for SIMD multiplication add (henceforth abbreviated as *SIMDMA*) is the same as that of SIMD multiplication. From the data, we see that the Speedup is consistently better than the Speedup of SIMD multiplication. Given the function, I observed that there is one additional step for each calculation: adding the calculated value to a 3rd register (there's also the additional step at the end of summing all the values in the 3rd vector register, but this is negligible given it's done only once). Again, this is incredibly efficient at the assembly level, and requires only one extra execution step. However, this may not be the case when written in C++ for the non-SIMD function. Although it's a single extra C++ step to add the value to a cumulator after the multiplication step, the compiler might not be entirely efficient and break this down as several assembly steps. Hence, we see why when comparing the Speedup between the 2 functions, SIMDMA consistently is better. This fact points us to the fact that the C++ code is simply not as efficient, and hence, would reduce the performance of the non-SIMD function from it's theoretical best, which leads to a much greater Speedup > 4.0 when it's compared to its SIMD assembly-wrapped equivalent.