Lachlan Sinclair

sinclala@oregonstate.edu

5/10/2020

CS 475 Spring

Project #4: Vectorized Array Multiplication/Reduction using SSE

System: For this project I used the flip3 server. The results used for both the non-parallel and parallel sections were taken early in the morning when the server load was reasonably low. Running this on my Windows machine was producing erroneous results. The project4script.py file is used to compile and run the two programs.

Results:

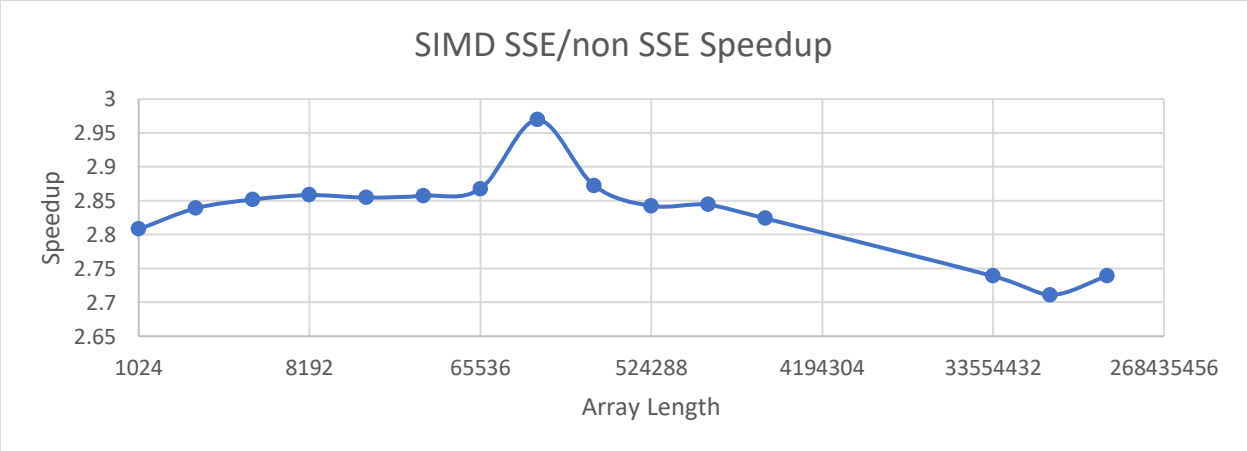
Array Length	Plain Perf. (Mega multiplies/reductions per second)	SSE Perf. (Mega multiplies/reductions per second)	SSE SIMD Speedup
1024	223.637065	627.93354	2.807824097
2048	224.275702	636.659889	2.838737693
4096	225.079146	641.86318	2.851722123
8192	225.234758	643.765728	2.858198858
16384	236.235025	674.353083	2.854585526
32768	225.984849	645.709212	2.857311961
65536	224.888885	644.874855	2.867526579
131072	226.663813	673.022839	2.969255789
262144	224.160678	643.815056	2.872114154
524288	229.860774	653.343639	2.842345075
1048576	226.187738	643.291423	2.844059668
2097152	222.081905	627.133282	2.823882846
33554432	226.615584	620.644205	2.738753417
67108864	226.241558	613.320142	2.710908409
134217728	219.4393	600.990559	2.73875536

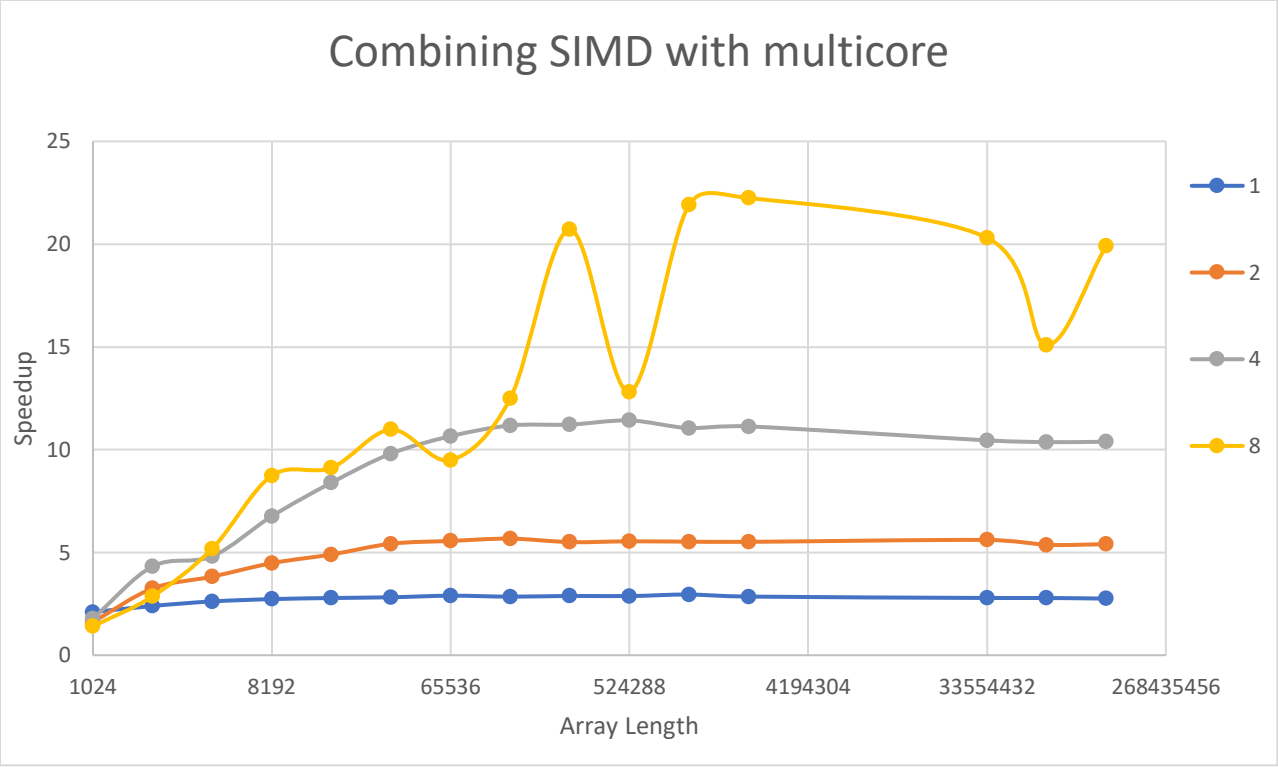
Array Length	Number of threads	Plain Perf. (Mega multiplies/reductions per second)	SSE+parallel Perf. (Mega multiplies/reductions per second)	Parallel+SIMD Speedup
1024	1	223.568855	469.074927	2.098122867
1024	2	223.591587	362.337	1.620530561
1024	4	223.500687	400.113402	1.790211061
1024	8	223.523405	314.235961	1.405830235
2048	1	234.512451	562.122509	2.396983642

2048	2	234.549971	761.698391	3.247488745
2048	4	224.321458	969.373267	4.321357732
2048	8	224.321458	642.613459	2.864699012
4096	1	234.932107	615.1114	2.618251749
4096	2	234.957209	898.477326	3.824004081
4096	4	234.957209	1130.748557	4.812572305
4096	8	234.944657	1218.295432	5.185457067
8192	1	235.211665	642.097454	2.729870791
8192	2	235.246263	1053.92919	4.480110232
8192	4	235.265139	1590.613566	6.760940328
8192	8	235.265139	2055.161921	8.735514024
16384	1	236.268338	657.787061	2.784067754
16384	2	225.930432	1107.680773	4.902751538
16384	4	236.227094	1982.776674	8.393519306
16384	8	225.92608	2060.095561	9.118449543
32768	1	226.024046	637.357633	2.819866489
32768	2	225.855017	1225.231908	5.424860268
32768	4	226.00372	2217.036679	9.80973534
32768	8	123.29776	1353.609514	10.9783788
65536	1	175.231449	507.496794	2.89615133
65536	2	122.689175	682.234576	5.560674575
65536	4	224.843615	2395.857961	10.65566376
65536	8	224.898948	2135.719204	9.496350352
131072	1	224.789028	640.595219	2.849761951
131072	2	122.206732	693.206165	5.672405715
131072	4	122.160059	1365.462027	11.17764708
131072	8	122.149032	1525.725403	12.49068763
262144	1	232.631638	671.142084	2.884999176
262144	2	232.373234	1279.24564	5.505133349
262144	4	224.292054	2516.843577	11.22127838
262144	8	234.160797	4849.755797	20.71122006
524288	1	231.934183	667.492653	2.877939959
524288	2	231.114019	1280.671635	5.54129793
524288	4	223.178428	2549.482488	11.42351665
524288	8	224.089562	2871.834661	12.8155664

1048576	1	223.422584	660.042541	2.954233763
1048576	2	232.275678	1283.206234	5.524496775
1048576	4	231.959461	2562.914577	11.04897626
1048576	8	231.670509	5074.970225	21.90598297
2097152	1	229.243343	654.246542	2.853939109
2097152	2	229.409782	1265.603555	5.516781124
2097152	4	230.285162	2563.614852	11.13234926
2097152	8	229.742343	5110.42127	22.24414186
33554432	1	226.200126	630.376342	2.786808094
33554432	2	214.247753	1201.497931	5.607983814
33554432	4	227.129812	2374.307867	10.45352808
33554432	8	227.538717	4621.310174	20.30999487
67108864	1	226.58027	631.214444	2.785831458
67108864	2	227.387409	1221.257506	5.370822911
67108864	4	227.254789	2356.099748	10.36765719
67108864	8	228.227401	3445.538333	15.09695294
134217728	1	226.627832	624.269144	2.754600521
134217728	2	226.625545	1224.05466	5.401221032
134217728	4	227.345813	2361.812176	10.38863283
134217728	8	227.61432	4533.652145	19.91813233

	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152	33554432	67108864	1.34E+08
1	2.098122867	2.396983642	2.618251749	2.729870791	2.784067754	2.819866489	2.89615133	2.849762	2.884999	2.87794	2.954234	2.853939	2.786808	2.785831	2.754601
2	1.620530561	3.247488745	3.824004081	4.480110232	4.902751538	5.424860268	5.560674575	5.672406	5.505133	5.541298	5.524497	5.516781	5.607984	5.370823	5.401221
4	1.790211061	4.321357732	4.812572305	6.760940328	8.393519306	9.80973534	10.65566376	11.17765	11.22128	11.42352	11.04898	11.13235	10.45353	10.36766	10.38863
8	1.405830235	2.864699012	5.185457067	8.735514024	9.118449543	10.9783788	9.496350352	12.49069	20.71122	12.81557	21.90598	22.24414	20.30999	15.09695	19.91813





Explanation:

This program works by calling two functions that multiply pairwise values in two arrays then sums all those products into a single value that they return. One function uses the intrinsics code provided and the other does it using simple C++ code. The amount of time required for each function to execute is recorded and used to calculate its performance, this process is repeated ten times and the best result is recorded. The multithreading extra credit version of the program works in the same way, but it also adds the additional step of using multiple threads to execute the intrinsics function. All SIMD runs use SSE with a width of 4. Unfortunately, I couldn't achieve a speedup of 4 when using the SSE code, most runs had a speedup of around 2.8-3, this range appears to be the expected speedup for the given intrinsics code. To test this, I created a program to run multiplying pairwise values in arrays and store those values in another array (no reduction), this achieved speedups of roughly 3.8. This made me confident that my code was correct and that the reduction step is responsible for the speedup being around 2.85 because the non-reduction program matched the lectures examples extremely well. I did run some additional tests using the assembly code rather than the intrinsics code and got a speed up of around 8. The speed up we are seeing is because the SIMD code is performing the multiplication and addition instructions on 4 xmm registers at the same time which greatly reduces the total amount of time required to complete the computations.

Analysis:

The SIMD SEE/non SSE Speedup graph show that the speedup did not stay consistent over the various arrays sizes. For formatting proposes the x axis in both graphs is on a logarithmic base 2 scale. At the start of the SIMD SSE/non SSE Speedup graph you can see the speedup quickly ramps up then plateaus for a while. I think this ramp up is due to the small nature of the array at the start of the graph and the

performance was potentially reduced due to the function call requiring a more significant portion of the time to execute the code in between the two-time stamps (the benefit of fast computation is lessened). There is a small jump from a speedup of 2.85 to a speedup of almost 3 at a size of 131072. In most of the separate runs I preformed, one or two array lengths would see a slight jump in speedup. I think these jumps have to do with the server the program was being run on, it was hard to find a good time when the load was low on the server. In theory this shouldn't have mattered too much, but there might be a chance that the cache lines where being cycled out by another program running on the same CPU. Toward the larger array sizes, you can see a steady decline in the speedup, this is likely due to violating the temporal cache coherence. Specifically, the SSE performance as the array size got larger began to fall off and the non-SSE performance maintained at the same value causing the speedup to gradually fall.

Combining SIMD with multicore had the expected results. As the number of threads executing the SIMD code increased, the speedup also increased. However, like the SIMD only graph, every thread count used experienced a drop in the speedup as the array sizes grew large. This again is due to violating the temporal cache coherence. The larger the thread count the more it was affected by this violation. Overall the speedups were slightly more erratic because of the inherent nature of multithreading on a shared server. For the most part, doubling the number of threads being used doubled the speedup achieved once sufficiently large arrays were being processed. This proves that combining SIMD with multicore produces a multiplicatively faster program in certain situations.