1. FLIP and virtual image of Kali Linux 2018.1 on personal computer (with similar results)
   - Kali Linux 2018.1 (Bash Terminal)
   - Intel i7-7700K @ 4.20GHz (4 cores, 8 threads)

2. Tables and graphs attached to end. Descriptions for each are as follows:
   - SIMD SSE C++ performance (default)
   - SIMD via OpenMP rewritten in C (single-threaded)
   - SIMD via OpenMP C++ (single-threaded)

3. As evidenced in the submitted code (also attached below the tables), the code deployed is a straightforward implementation of what was provided in our Project #5 instructions, tested on FLIP and a virtual image of Kali Linux 2018.1 to similar results.

What I observed is clearly not what was intended by this project, which will be addressed more in the next questions. Here, assembly language SIMD SSE instructions present an unmistakable flat-line in performance notwithstanding the array size given, whereas our reference (the personal) code increased in performance in step with the increasing array size. Therefore, as array-size boundary increases, the MegaMults/Sec gulf between SIMD SSE and non-SIMD SSE structures amplifies.

For comparison—and in concerted effort to coax out my anticipated performance—the gulf in performance drastically lessened when the assembly language is replaced with an OpenMP SIMD pragma. While the C-language reimplementation never achieves the expected better performance than reference (personal) calculation functions, this method reflects the best SIMD results I could muster for the three.

4. In short, yes. Should they have been? No. As personal research uncovered, speed-up, theoretically, should close upon its maximum performance while the array size is below the L1 Cache limit, then worsen as the array size passes L1 size, dropping again as array size passes L2 cache size, and once more as array size passes L3 cache size boundary ["OpenMP and Vectorization Training Introduction," NeRSC (src), and see "Effective Vectorization with OpenMP 4.5," Oak Ridge National Laboratory (src)]. However, as described above, performance either (a) remains flat or (b) ascends indiscriminately, as if unaffected by cache line distribution.

5. The most I can make of this difference in performance between our _asm SIMD SSE structure and optimizing the same code with an OpenMP pragma simd (let alone an OpenMP pragma simd reduction (+:sum)), is a failure on the part of the GNU compiler to auto-recognize vector optimization. (Note: I describe this compiler as the GNU as I am referring to both the G++ and GCC implementations, as I rewrote the program for C language deployment to eerily similar testing results. So, that is C language along with the independent OpenMP SIMD C++ testing.)

(Reading Oak Ridge National Laboratory's (ORNL) PDF guide, I noted their difficulty vectorizing instructions for a benchmark that included reduction statements, allowing me to believe in the possibly fickle nature getting GCC or Clang compilers to comply (see chap. 6, "HACCmk"). However, that only explains where reductions are concerned, and would apply specifically to the OpenMP 4.5 SIMD standard.)

6./7. Because instructions for a single function loop can oversee four floating-point operations. As explained in ORNL's "Effective Vectorization with OpenMP 4.5" (src):

> SIMD instructions allow the same operation to be performed simultaneously on multiple pieces of data. This allows a large set of independent operations to be broken down into a smaller set of packed data that can be handled in parallel. This parallelism allows large loops to execute in a fraction of the time.
>
> The benefits of vectorization** can be seen if the loop is unrolled by a certain factor. If the processor’s architecture supports the SSE instruction set, the processor can make use of 128-bit XMM vector registers. . . . [A] XMM register is able to operate on four floats simultaneously. So, the loop can be unrolled by a factor of four.
>
> *** The term "vectorization" refers to when multiple pieces of data are packed into a single, larger data type. This can be thought of as a one-dimensional array of fixed size. The boundaries of the array are determined by the data type packed in the array. SIMD instructions work on this array of packed data by modifying the entire array with a single processor instruction.*

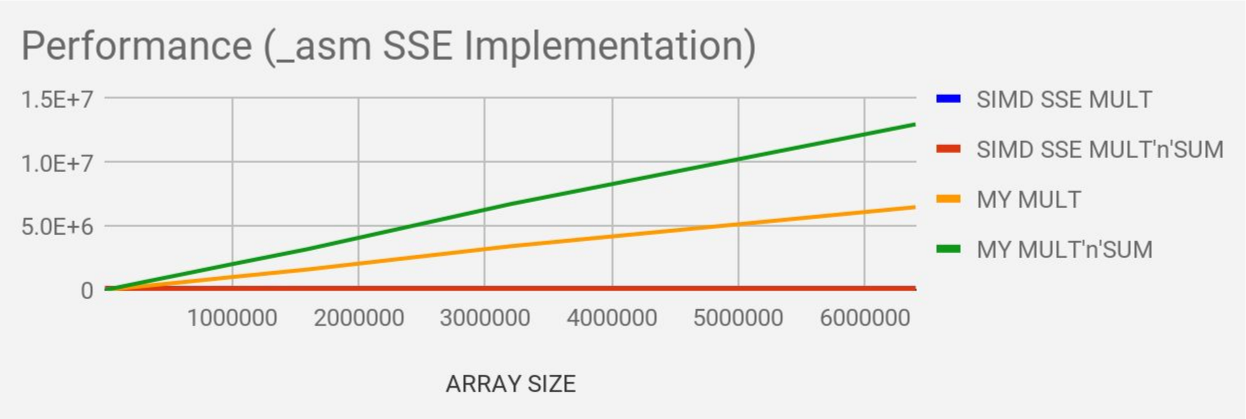
The reason for reduction's similar performance capability (and speedup capacity) is its functional relationship to linearly-structured arithmetic. Restated (again from ORNL):

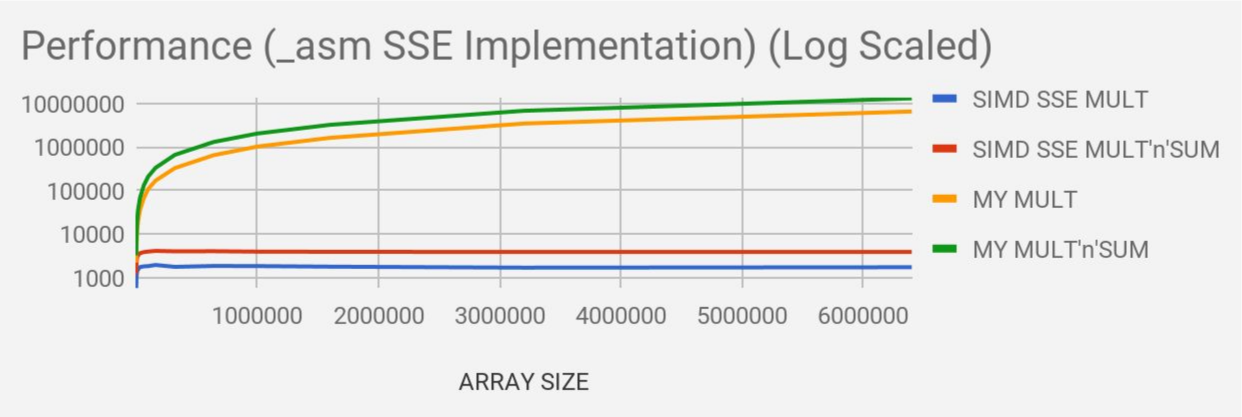
> The reduction clause [aggregates partial work] by creating a private vector copy of the variable inside [a parallel] loop[*sic*] which is used to store the partial values. When the SIMD [(parallel)] region ends, the vector copy of the original variable is horizontally aggregated. The final value is then moved from the vector copy to the original variable.

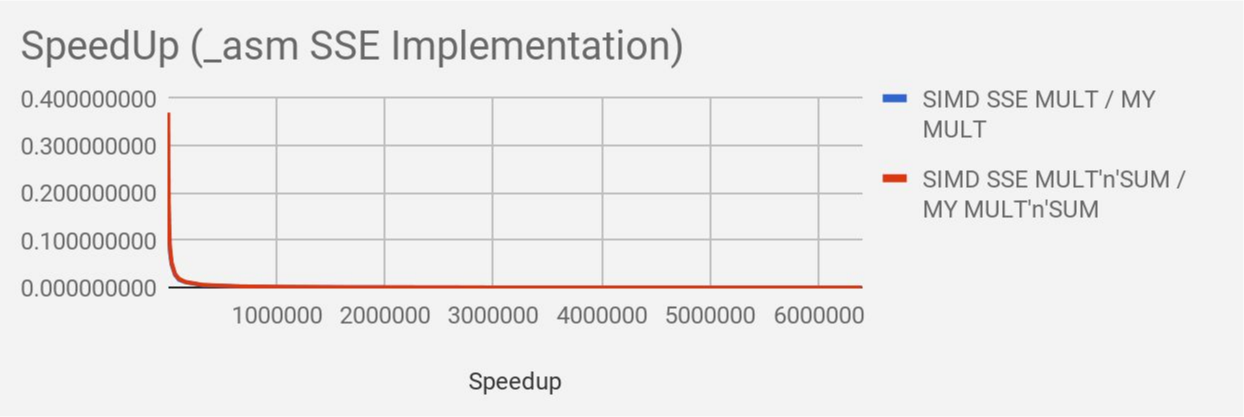
DEFAULT (_asm SIMD SSE)

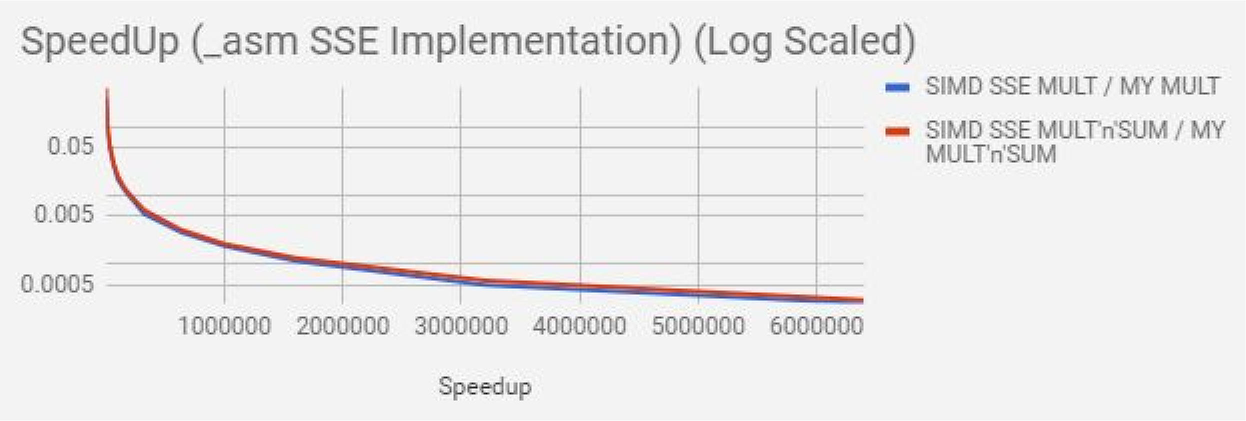
PERF (SIMD using _asm SIMD SSE)						SPEEDUP (SIMD using _asm SIMD SSE)		
ARRAY SIZE	SIMD SSE MULT	SIMD SSE MULT'n'SUM	MY MULT	MY MULT'n'SUM		SPEEDUP	SIMD SSE MULT / MY MULT	SIMD SSE MULT'n'SUM / MY MULT'n'SUM
1000	572.62	1194.20	2208.22	3230.52		1000	0.259312931	0.369661850
1600	708.99	1557.06	3144.56	4688.38		1600	0.225465566	0.332110452
3200	995.13	2108.92	5288.46	8440.94		3200	0.188170091	0.249844212
6400	1302.25	2649.84	8932.88	15579.26		6400	0.145781652	0.170087668
10000	1411.82	3043.46	13117.85	23176.40		10000	0.107625869	0.131317202
16000	1533.22	3202.99	19755.79	35883.08		16000	0.077608640	0.089261847
32000	1739.59	3619.39	35498.50	67454.21		32000	0.049004606	0.053656992
64000	1808.06	3845.94	67126.43	131740.48		64000	0.026935143	0.029193305
100000	1831.27	3956.70	108365.15	208800.93		100000	0.016899068	0.018949628
160000	1945.65	4093.66	167793.37	329652.32		160000	0.011595512	0.012418114
320000	1761.60	4001.07	326526.62	655877.84		320000	0.005394966	0.006100328
640000	1846.28	4023.52	643803.82	1295581.28		640000	0.002867768	0.003105571
1000000	1834.93	3934.16	1015031.72	2021993.53		1000000	0.001807756	0.001945684
1600000	1780.10	3905.30	1611953.21	3205602.36		1600000	0.001104312	0.001218273
3200000	1688.63	3865.16	3429648.23	6731769.05		3200000	0.000492362	0.000574167
6400000	1726.94	3884.35	6492672.75	12981042.12		6400000	0.000265983	0.000299233

DEFAULT (_asm SIMD SSE)





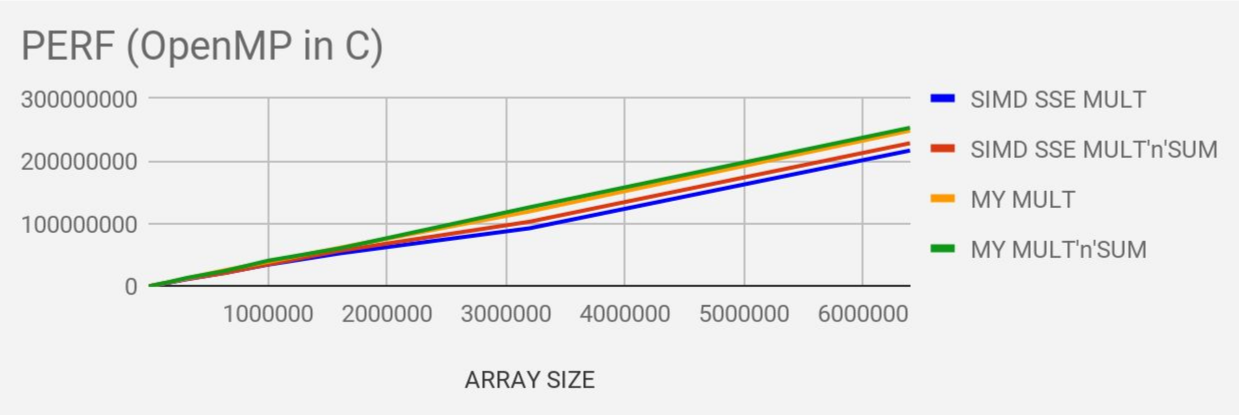


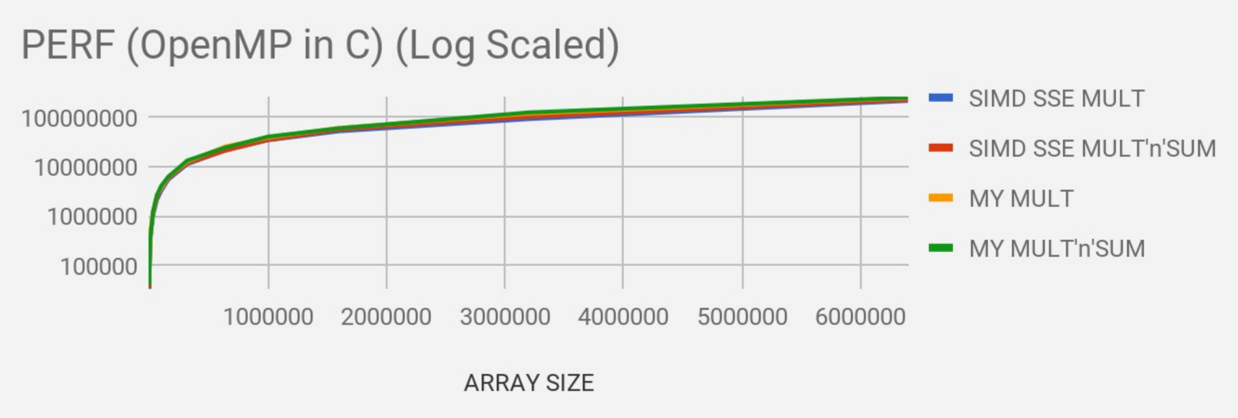


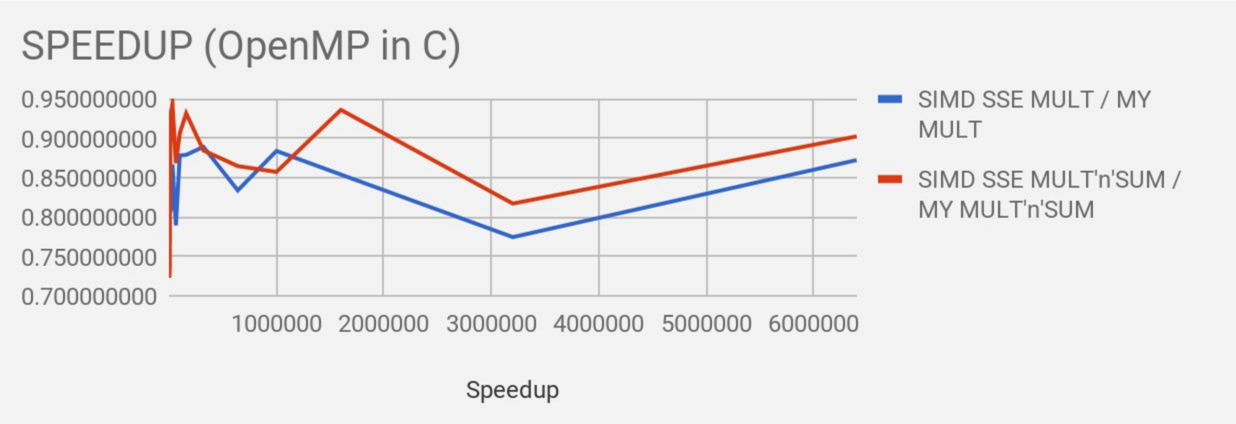
OPENMP SIMD (C LANG)

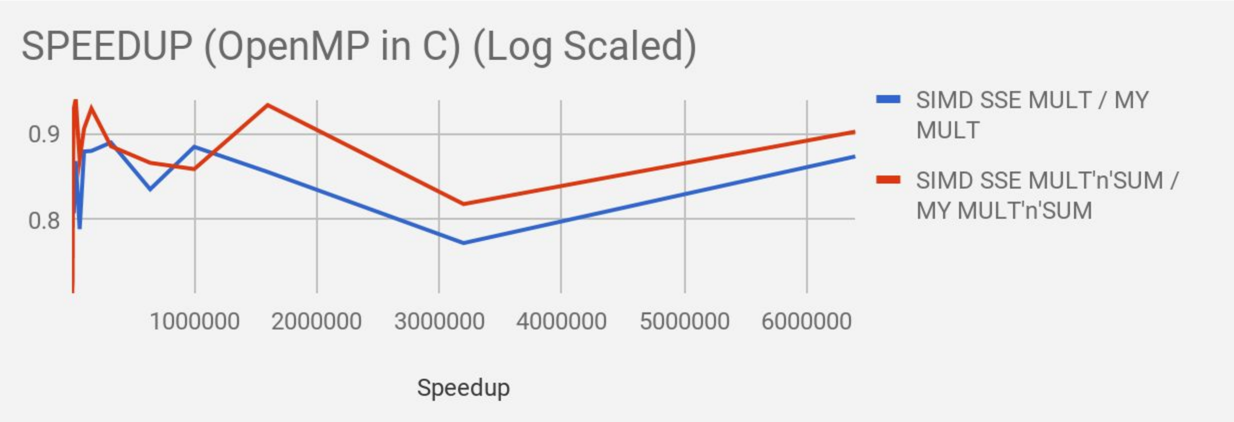
PERF (C-lang, Pragma OMP SIMD)						SPEEDUP (C-lang, Pragma OMP SIMD)		
ARRAY SIZE	SIMD SSE MULT	SIMD SSE MULT'n'SUM	MY MULT	MY MULT'n'SUM		SPEEDUP	SIMD SSE MULT / MY MULT	SIMD SSE MULT'n'SUM / MY MULT'n'SUM
1000	33329.39	35584.08	40822.56	40176.05		1000	0.816445468	0.885703736
1600	54968.41	56885.28	62129.94	65463.25		1600	0.884732998	0.868965175
3200	110278.78	93910.79	126118.36	129920.07		3200	0.874407054	0.722835112
6400	218332.55	195362.39	262134.87	265411.74		6400	0.832901614	0.736072911
10000	342489.84	349758.97	400115.43	410090.59		10000	0.855977574	0.852882220
16000	496547.66	518999.81	615456.37	557276.94		16000	0.806795870	0.931313982
32000	1024330.19	1194235.11	1181661.15	1265346.51		32000	0.866856110	0.943800849
64000	2112582.68	2287050.33	2676123.37	2633289.37		64000	0.789419015	0.868514625
100000	3215335.23	3687987.76	3660736.92	4069067.10		100000	0.878330047	0.906347247
160000	5476848.26	5848147.20	6230800.42	6276020.65		160000	0.878995938	0.931824086
320000	11258866.02	11818449.64	12661633.42	13359750.72		320000	0.889211183	0.884631000
640000	21394041.72	21119160.15	25656304.00	24422261.28		640000	0.833870760	0.864750397
1000000	34370829.48	34902342.35	38882154.63	40711199.92		1000000	0.883974404	0.857315491
1600000	52309804.68	56417705.24	61254902.72	60263656.97		1600000	0.853969272	0.936181242
3200000	92442324.49	102924799.26	119361600.89	125965085.44		3200000	0.774472894	0.817089902
6400000	216521945.50	228204400.47	248201829.36	252850292.54		6400000	0.872362408	0.902527730

OPENMP SIMD (C LANG)





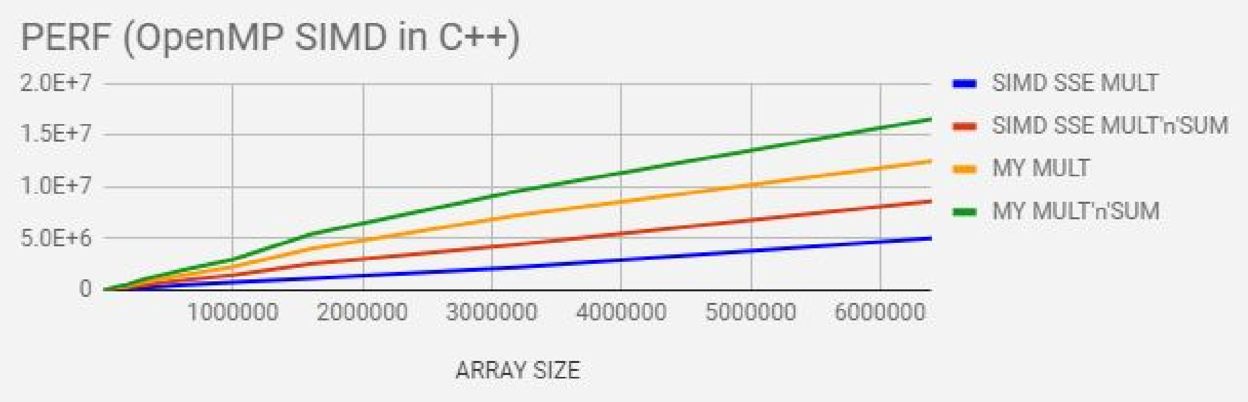


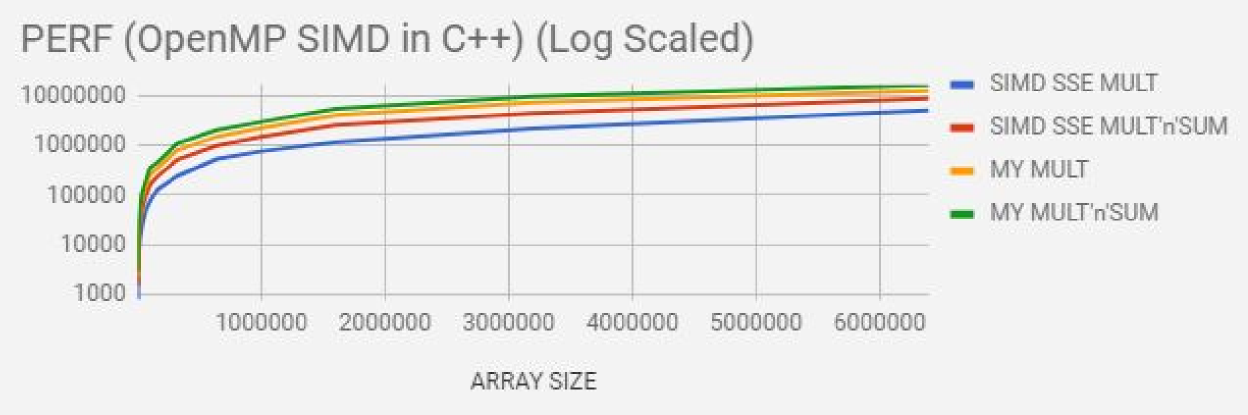


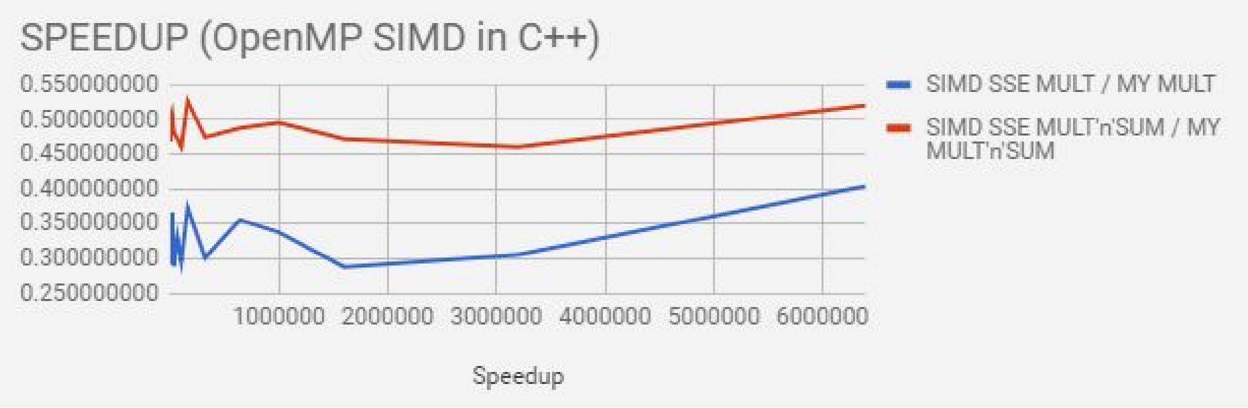
OPENMP SIMD SSE (C++ LANG)

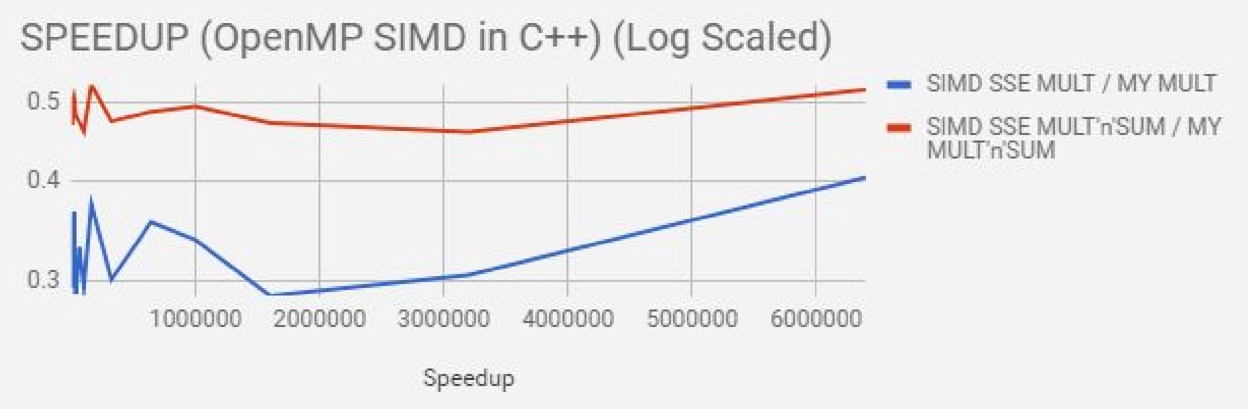
PERF (OpenMP SIMD in C++)						SPEEDUP (OpenMP SIMD in C++)		
ARRAY SIZE	SIMD SSE MULT	SIMD SSE MULT'n'SUM	MY MULT	MY MULT'n'SUM		SPEEDUP	SIMD SSE MULT / MY MULT	SIMD SSE MULT'n'SUM / MY MULT'n'SUM
1000	777.46	1489.81	2274.37	3039.82		1000	0.341835321	0.490098098
1600	1407.02	2738.01	3984.26	5424.69		1600	0.353144624	0.504731146
3200	2417.00	5223.46	8210.10	11120.23		3200	0.294393491	0.469725896
6400	5463.81	11020.81	16774.25	22764.74		6400	0.325726038	0.484117543
10000	8498.12	16124.33	24315.08	31886.15		10000	0.349499981	0.505684443
16000	13611.37	25310.77	37193.94	49711.65		16000	0.365956659	0.509151678
32000	23871.09	51437.98	82478.63	106869.91		32000	0.289421514	0.481313964
64000	47617.45	93354.03	144100.02	197365.28		64000	0.330447213	0.473001280
100000	76209.77	162298.98	257613.23	351797.49		100000	0.295830187	0.461342064
160000	130137.79	239491.56	347954.85	454274.00		160000	0.374007691	0.527196274
320000	242129.87	517148.78	803684.80	1089659.76		320000	0.301274666	0.474596566
640000	533192.55	986699.74	1499183.17	2023760.59		640000	0.355655373	0.487557543
1000000	760227.57	1471244.46	2250625.12	2970264.44		1000000	0.337785073	0.495324403
1600000	1152073.06	2560764.33	4001458.05	5421618.10		1600000	0.287913317	0.472324735
3200000	2209107.84	4426932.44	7231021.78	9609206.37		3200000	0.305504244	0.460696989
6400000	5041211.97	8627791.52	12482127.33	16590449.11		6400000	0.403874423	0.520045688

OPENMP SIMD SSE (C++ LANG)


PERF (OpenMP SIMD in C++)
2.0E+7
1.5E+7
1.0E+7
5.0E+6
ARRAY SIZE
SIMD SSE MULT
SIMD SSE MULT'n'SUM
MY MULT
MY MULT'n'SUM


PERF (OpenMP SIMD in C++) (Log Scaled)
ARRAY SIZE
SIMD SSE MULT
SIMD SSE MULT'n'SUM
MY MULT
MY MULT'n'SUM


SPEEDUP (OpenMP SIMD in C++)
Speedup
SIMD SSE MULT / MY MULT
SIMD SSE MULT'n'SUM / MY MULT'n'SUM


SPEEDUP (OpenMP SIMD in C++) (Log Scaled)
Speedup
SIMD SSE MULT / MY MULT
SIMD SSE MULT'n'SUM / MY MULT'n'SUM