Varsha Singh
EC527, Lab 3
Martin Herbordt

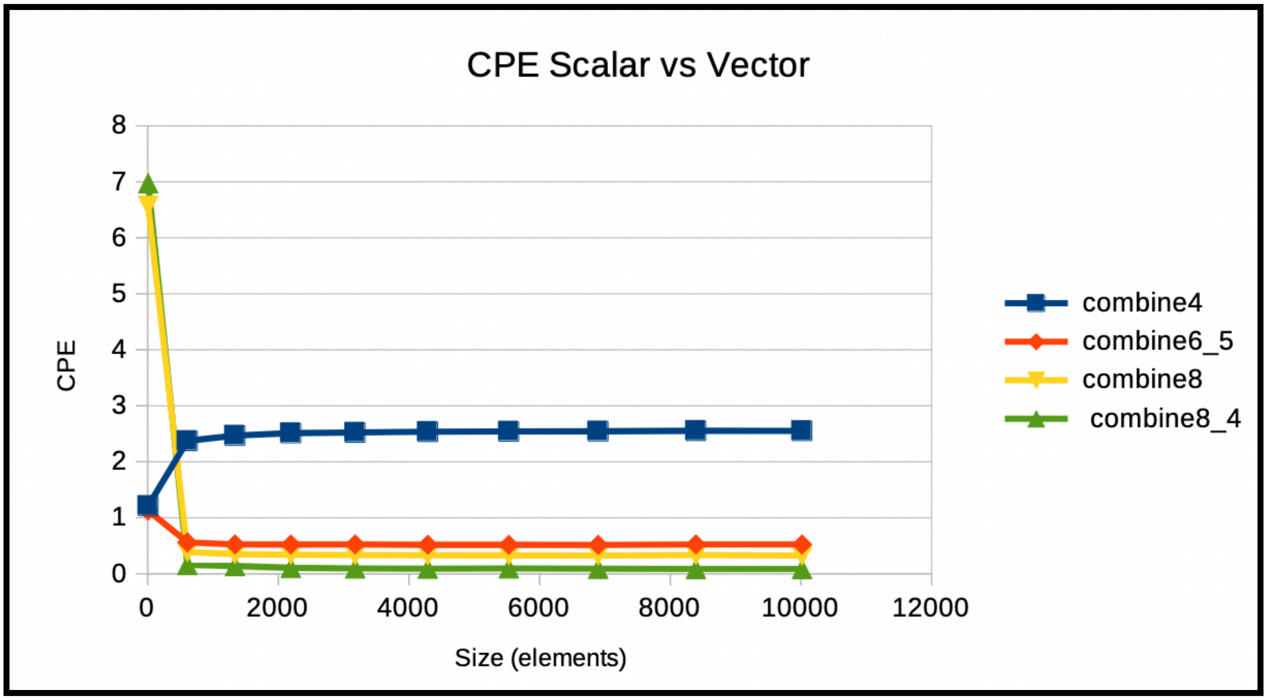
Lab 3

Varsha Singh

Part 1: SSE extensions using C structs and union

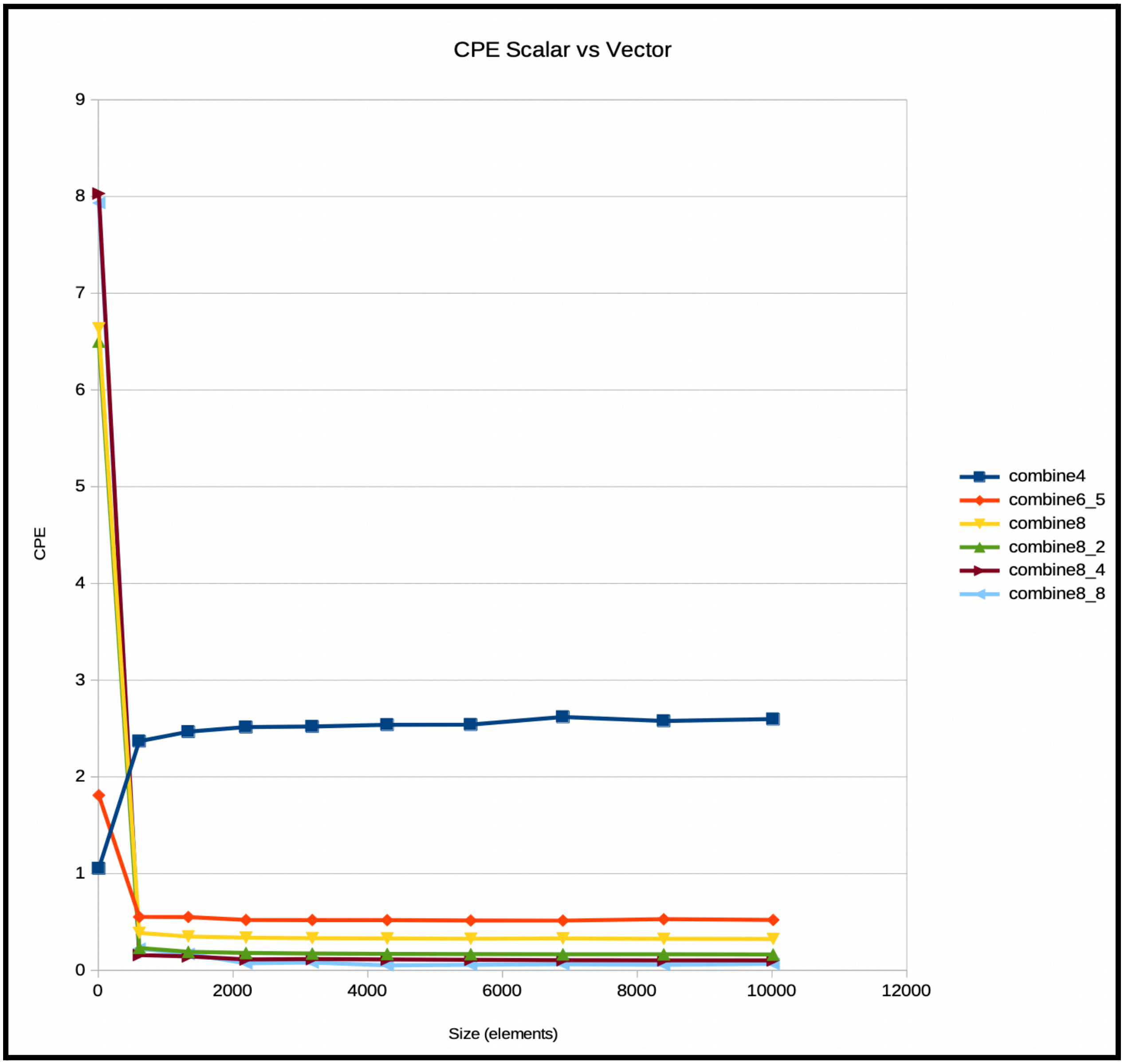
1a. Compile test_combine8.c and run. Plot the results and get the CPE. Justify the vector results (also comparing with the scalar results).

- I set my A B and C constants to have the size of the array be around 10000 elements with X = NUM_TESTS-1. I reached a size of 10016 by setting A B and C to 64, 536, and 8. It seems like the results are expected as the number of elements increase – the base scalar code appears to even out between 2 and 3 CPE, whereas the 'best' scalar code which unrolls 5 times and 5 accumulators performs between 0 and 1 CPE. The best scalar is around ~0.50 CPE, whereas the base vector code is around ~0.3 CPE and the vector unrolled 4 times with 4 accumulators is around ~0.1 CPE.
  Compared to the scalar results, the larger-scaled behavior of the combine8_4 function performs 5x better compared to the best scalar implementation. I believe that this is due to vector programming being highly parallelized as compared to scalar programming. Vector programming processes chunks of data as units, and the vector instructions access memory that has a known configuration, meaning that no data caches are required.



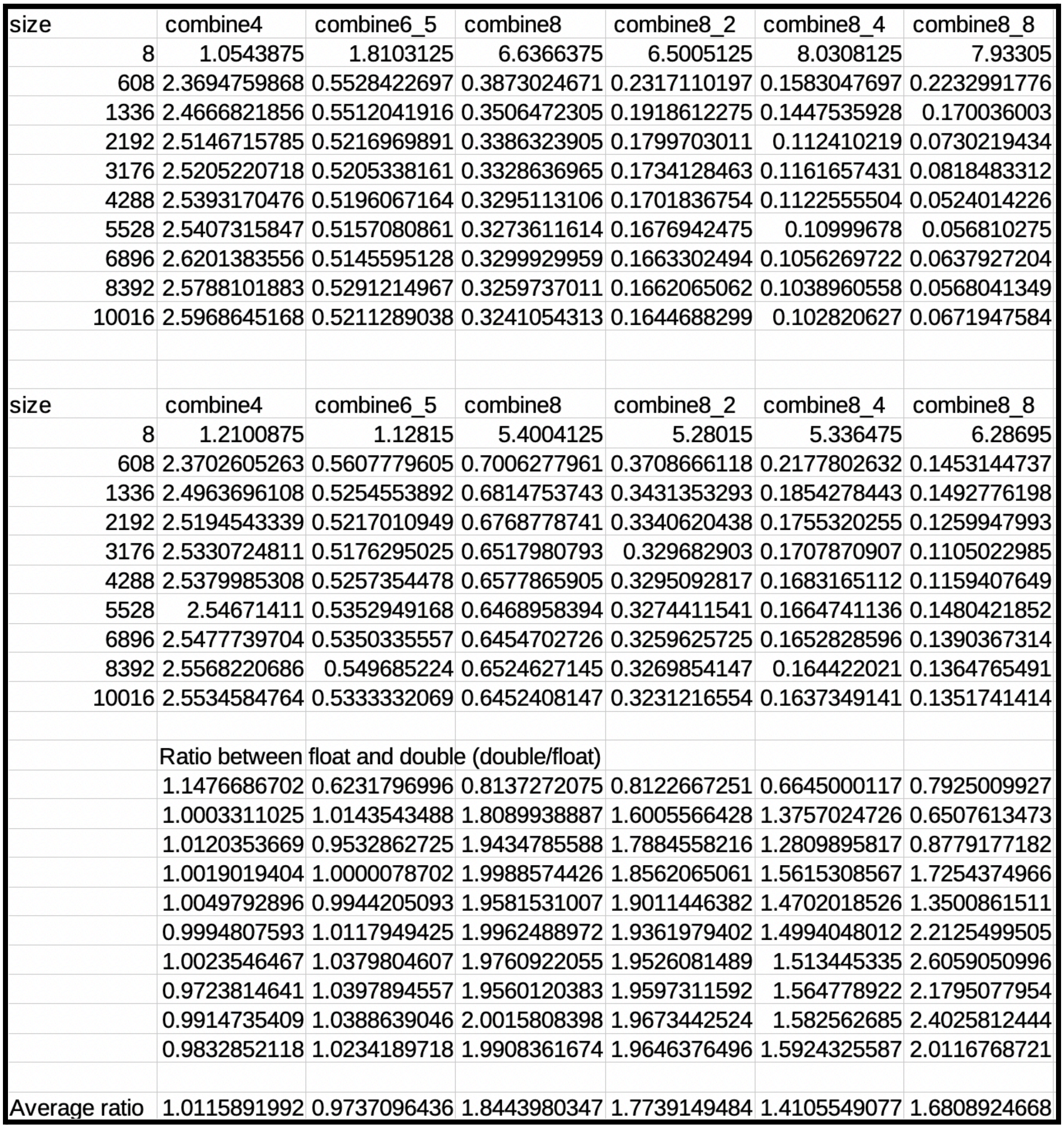
1b. Write code for two more functions, with 2 and 8 accumulators, respectively.

The vectorized version of the code seems to perform much better than either of the scalar versions. However, the first test, with 8 elements, has a much worse performance for the vectorized code as compared to the scalar code. To get my number of elements to around 10000 with x = NUM_TESTS - 1, I had to increase the linear and quadratic constants to a larger value. This made the number of smaller element arrays unaccounted for. There could be values in that range that can help bridge the gap between the CPE for an array size of 8 and 608, but my best explanation for this would be that VSIZE is equal to 8, and that the code will be done with one pass. There is a greater overhead to set up the vector registers, align in memory, and additional instructions that would outweigh the benefit of parallel execution. This can show that the vectorization is most effective when the cost for vectorizing will be outweighed by the amount of work that there is to do.



1c. Recompile using double rather than float. Does having 8 accumulators still help?

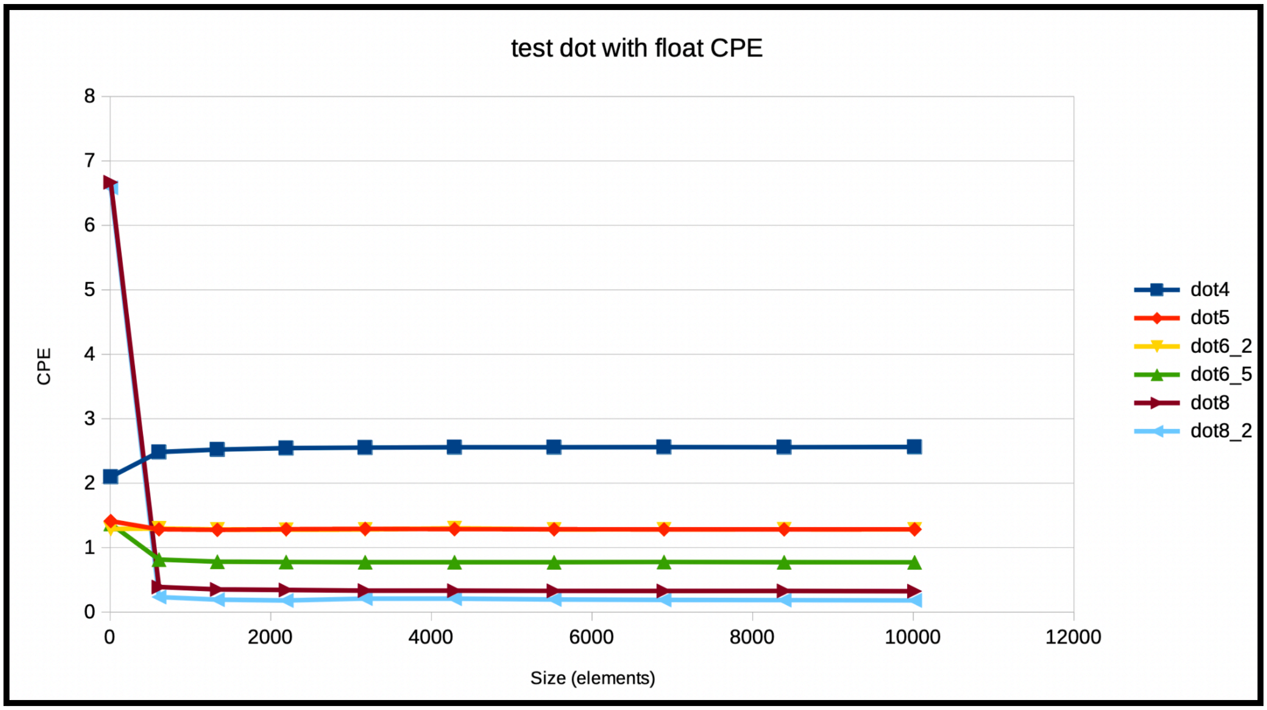
It depends on how you are determining what is helping. As compared to the other combine functions, it still generally performs better, but the average CPE for the double data type has an increase in CPE of about 1.68x greater than the average CPE for the float data type. There are some measurements where the double data type performs better than the float data type, but usually the double performs worse than its float counterpart.



size	combine4	combine6_5	combine8	combine8_2	combine8_4	combine8_8
8	1.0543875	1.8103125	6.6366375	6.5005125	8.0308125	7.93305
608	2.3694759868	0.5528422697	0.3873024671	0.2317110197	0.1583047697	0.2232991776
1336	2.4666821856	0.5512041916	0.3506472305	0.1918612275	0.1447535928	0.170036003
2192	2.5146715785	0.5216969891	0.3386323905	0.1799703011	0.112410219	0.0730219434
3176	2.5205220718	0.5205338161	0.3328636965	0.1734128463	0.1161657431	0.0818483312
4288	2.5393170476	0.5196067164	0.3295113106	0.1701836754	0.1122555504	0.0524014226
5528	2.5407315847	0.5157080861	0.3273611614	0.1676942475	0.10999678	0.056810275
6896	2.6201383556	0.5145595128	0.3299929959	0.1663302494	0.1056269722	0.0637927204
8392	2.5788101883	0.5291214967	0.3259737011	0.1662065062	0.1038960558	0.0568041349
10016	2.5968645168	0.5211289038	0.3241054313	0.1644688299	0.102820627	0.0671947584
size	combine4	combine6_5	combine8	combine8_2	combine8_4	combine8_8
8	1.2100875	1.12815	5.4004125	5.28015	5.336475	6.28695
608	2.3702605263	0.5607779605	0.7006277961	0.3708666118	0.2177802632	0.1453144737
1336	2.4963696108	0.5254553892	0.6814753743	0.3431353293	0.1854278443	0.1492776198
2192	2.5194543339	0.5217010949	0.6768778741	0.3340620438	0.1755320255	0.1259947993
3176	2.5330724811	0.5176295025	0.6517980793	0.329682903	0.1707870907	0.1105022985
4288	2.5379985308	0.5257354478	0.6577865905	0.3295092817	0.1683165112	0.1159407649
5528	2.54671411	0.5352949168	0.6468958394	0.3274411541	0.1664741136	0.1480421852
6896	2.5477739704	0.5350335557	0.6454702726	0.3259625725	0.1652828596	0.1390367314
8392	2.5568220686	0.549685224	0.6524627145	0.3269854147	0.164422021	0.1364765491
10016	2.5534584764	0.5333332069	0.6452408147	0.3231216554	0.1637349141	0.1351741414
	Ratio between float and double (double/float)					
	1.1476686702	0.6231796996	0.8137272075	0.8122667251	0.6645000117	0.7925009927
	1.0003311025	1.0143543488	1.8089938887	1.6005566428	1.3757024726	0.6507613473
	1.0120353669	0.9532862725	1.9434785588	1.7884558216	1.2809895817	0.8779177182
	1.0019019404	1.0000078702	1.9988574426	1.8562065061	1.5615308567	1.7254374966
	1.0049792896	0.9944205093	1.9581531007	1.9011446382	1.4702018526	1.3500861511
	0.9994807593	1.0117949425	1.9962488972	1.9361979402	1.4994048012	2.2125499505
	1.0023546467	1.0379804607	1.9760922055	1.9526081489	1.513445335	2.6059050996
	0.9723814641	1.0397894557	1.9560120383	1.9597311592	1.564778922	2.1795077954
	0.9914735409	1.0388639046	2.0015808398	1.9673442524	1.582562685	2.4025812444
	0.9832852118	1.0234189718	1.9908361674	1.9646376496	1.5924325587	2.0116768721
Average ratio	1.0115891992	0.9737096436	1.8443980347	1.7739149484	1.4105549077	1.6808924668

1d. Compile and run test_dot8.c using float. Plot the results and get the CPE. Justify the vector results (also comparing with the scalar results).

Like before, it appears that vectorizing the dot product function significantly improves performance as the size of the array increases. The most optimal scalar operation performs around 0.75 CPE, whereas the vectorized operation performs around 0.33 CPE and 0.2 CPE respectively, not including the performance with an array size of 8 elements. The vectorized performance will do better than the scalar performance because the operations are not dependent on one another and can be executed in parallel with SIMD. This will be more efficient because it can process more elements than with a scalar, and it has a higher throughput with SIMD registers.

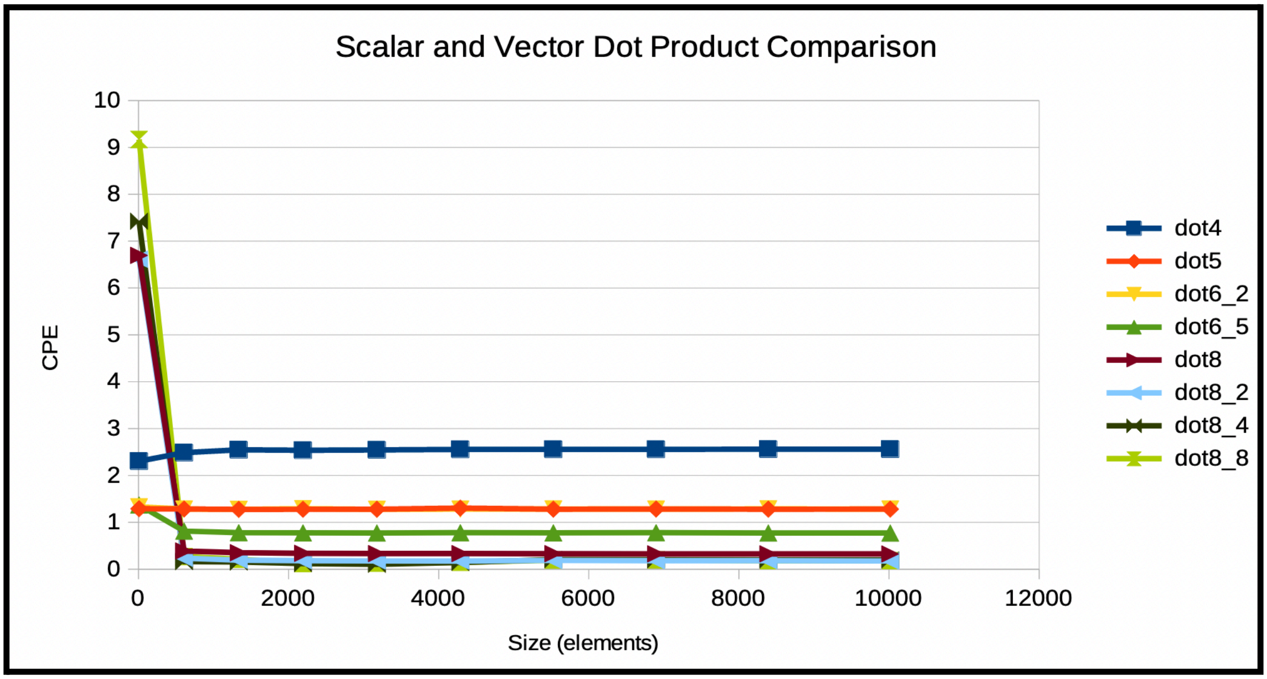


1e. Debug and fix dot8_2.

I debugged dot8_2 by looking at the final while loop for single stepping. The giveaway was that our dot product seemed to have a factor of n more than it should have. I changed the while loop logic to be *while(cnt)* rather than *while(cnt>=0)*, which seemed to fix this issue.

1f. Now that you have dot8_2 working, write code for new functions with 4 and 8 accumulators. As before you have to add to main() in a few places. Plot the results, get the CPEs, and justify.

I had further issues in getting the dot products to be the same value with these accumulator numbers. I found this to be due to inconsistent pointer arithmetic in the loop which assigns the data chunks. After I properly did this, the dot product values became the same. The speedup for the dot product values is less than the speedup for the combine values, but it is definitely evident that switching to a vectorized version helps the performance. The speedup comes from the idea of parallelizing the functions with SIMD, but it might take more cycles to do a multiplication than it does take for an addition. The number of operations between combine and dot product are different, so there are going to be more cycles required for dot products. However The speedup is better for dot8_2 as compared to dot8_4 and dot8_8. SIMD instructions aim to keep the ALU busy, but I think that unrolling more than 2 might cause it such that it will have to wait to run 4 or 8 in parallel.



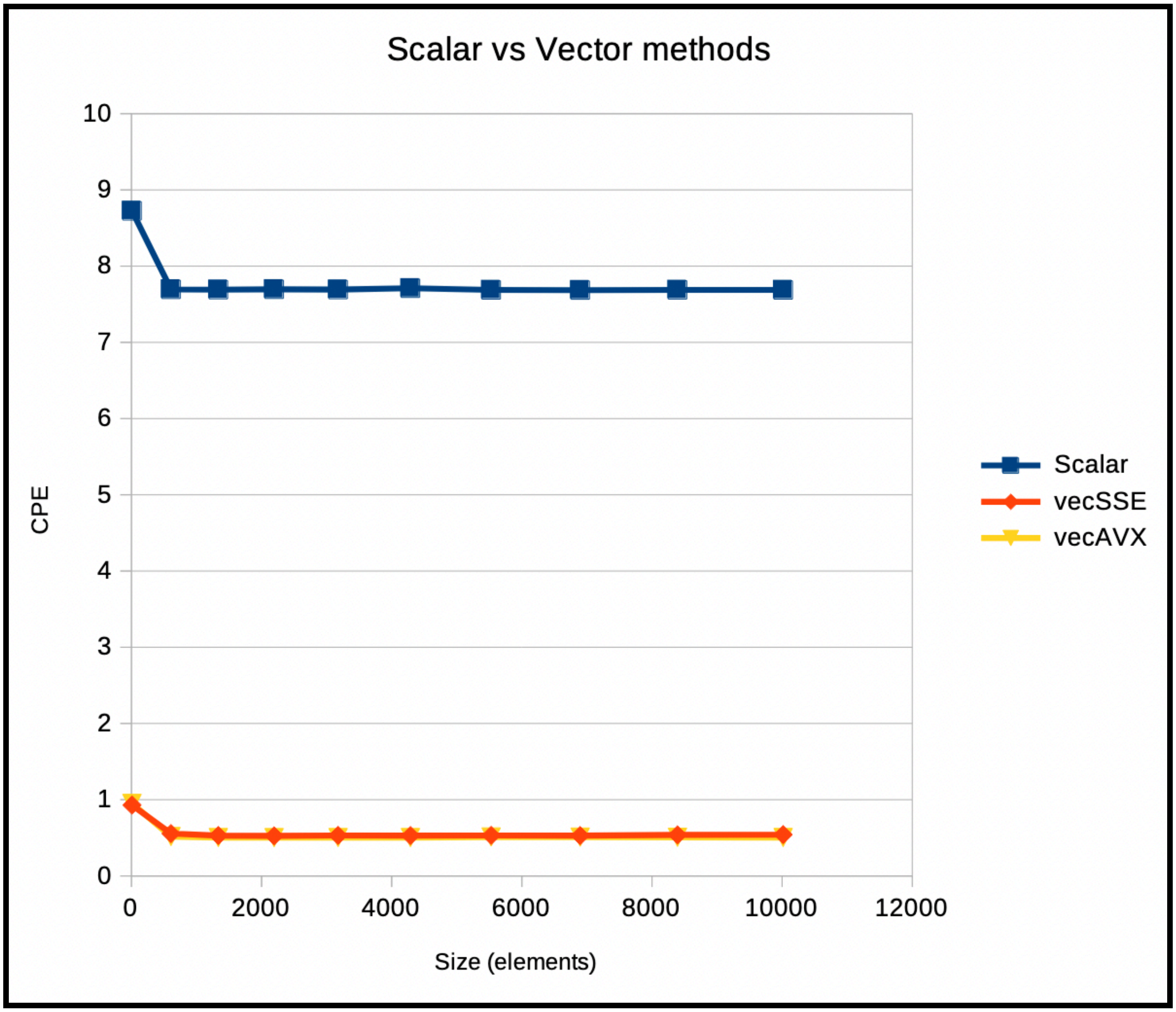
Part 2: SSE extensions using intrinsics

2a. Do all four methods work? Which ones do not work?

For the unedited avx_align.c file, the unalign_local_alloc worked and moved onto unalign_heap_native. This one only printed the first line but a segmentation fault occurred afterwards. It seems like that is the issue because there is incorrect memory alignment, as per my research online. We don’t seem to properly initialize to be 32-byte aligned, which if loaded to be 32-byte aligned will through a segmentation fault. So, a possible fix would be to use unaligned AVX instructions rather than directly assigning. This was the only broken function, everything else seems to have run with no errors.

2b. In test_intrinsics.c, set A, B and C as before. Compile and run. Plot the results and get the CPEs. Is this what you expected?

I expected that AVX would be faster than SSE, but did not have an estimation as to how fast it would be in comparison. I thought this because it is an extension to the x86 instruction set architecture and compared to SSE, it allows for a wider data processing with 356 bits instead of SSE’s 128 bits. This means that larger data sets would be able to offer more parallel power. However, I didn’t expect the two to have such similar results. They are relatively the same in performance, but AVX had abo



ut 1.04x lower CPE than SSE.

2c. Create two new (and very simple) non-vectorized functions to get execution time baselines: element-wise add and multiply (float only). What is the CPE? Can you vectorize these functions to make the throughput optimal?

I measured 6 types of functions. I measured element-wise multiplication as a scalar, SSE, and AVX; I also measured element-wise addition in the same 3 ways. I measured the % speedup as compared to the scalar version for each of the different functions that were vectorized, and got the following results:

- ADD:

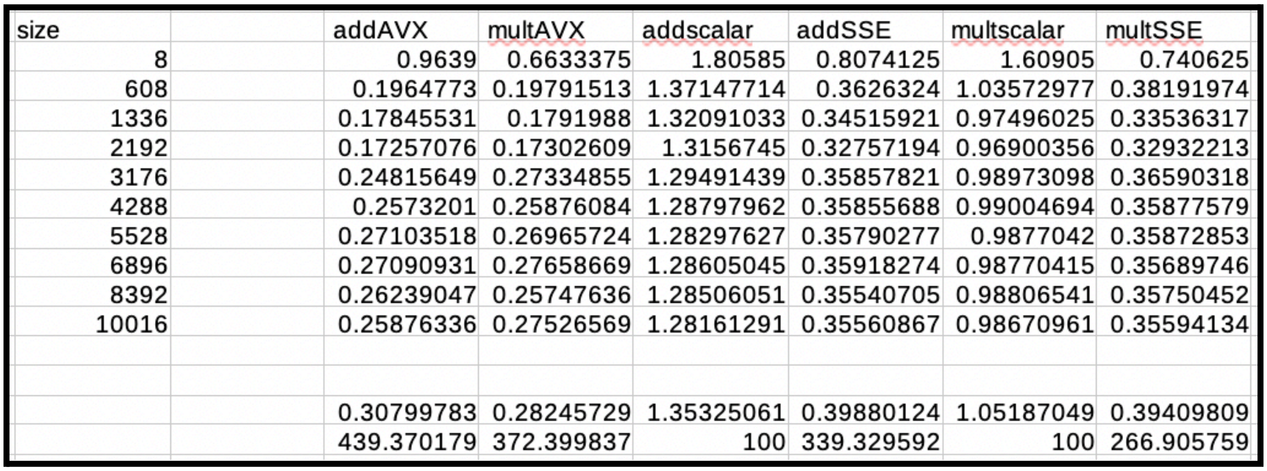
SSE - 339.33%

AVX: 439.37%

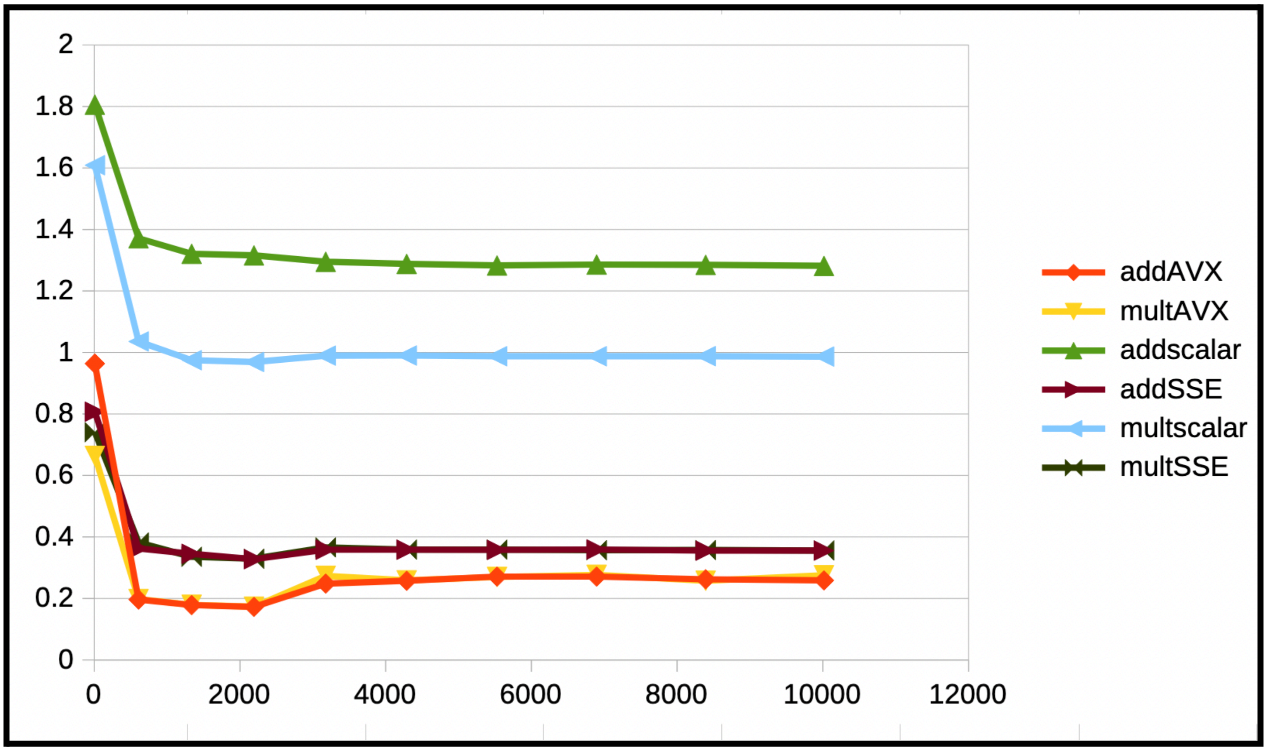
- MULT:

SSE - 266.91%

AVX: 372.40%

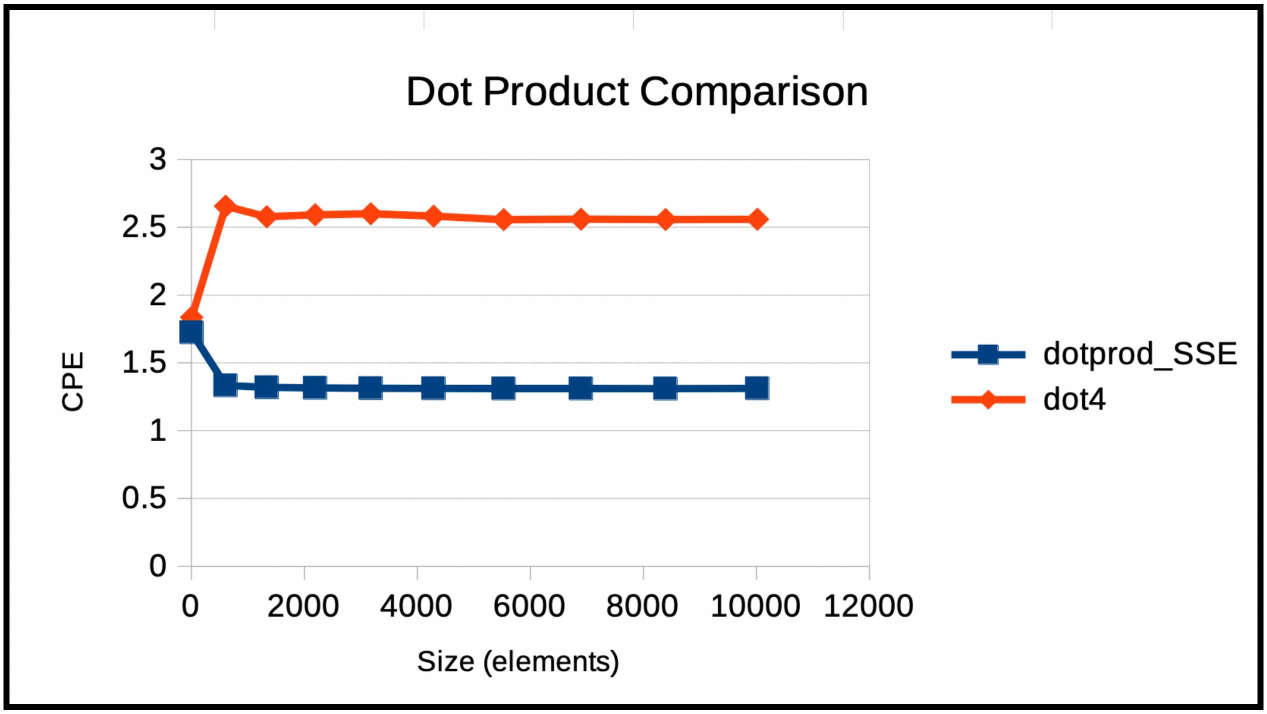


size		addAVX	multAVX	addscalar	addSSE	multscalar	multSSE
8		0.9639	0.6633375	1.80585	0.8074125	1.60905	0.740625
608		0.1964773	0.19791513	1.37147714	0.3626324	1.03572977	0.38191974
1336		0.17845531	0.1791988	1.32091033	0.34515921	0.97496025	0.33536317
2192		0.17257076	0.17302609	1.3156745	0.32757194	0.96900356	0.32932213
3176		0.24815649	0.27334855	1.29491439	0.35857821	0.98973098	0.36590318
4288		0.2573201	0.25876084	1.28797962	0.35855688	0.99004694	0.35877579
5528		0.27103518	0.26965724	1.28297627	0.35790277	0.9877042	0.35872853
6896		0.27090931	0.27658669	1.28605045	0.35918274	0.98770415	0.35689746
8392		0.26239047	0.25747636	1.28506051	0.35540705	0.98806541	0.35750452
10016		0.25876336	0.27526569	1.28161291	0.35560867	0.98670961	0.35594134
		0.30799783	0.28245729	1.35325061	0.39880124	1.05187049	0.39409809
		439.370179	372.399837	100	339.329592	100	266.905759



2d. Create a vectorized dot product function using intrinsics. Plot time measurement results and get the CPE. Compare the results you got this time(using intrinsics) to the results from test_dot8.c, which used __attribute__ ((vector_size(VBYTES))) declaration.

I made sure that the results agreed by adding the *if (1)* logic used for testing our test_dot.c main function. I added the code to the test_intrinsic.c file, but ensured they yielded the same result by adding code to the test_intrinsic.c main function that initializes the vector values to its index, like how it is done in test_dot.c. I also used an n size of 800 to test, and received the same result. This almost improves the speed by a factor of 2. I calculated the average speedup minus the initial calculation for a size of 8, and had an average speedup of 1.96x from dot4 to dotprod_SSE.



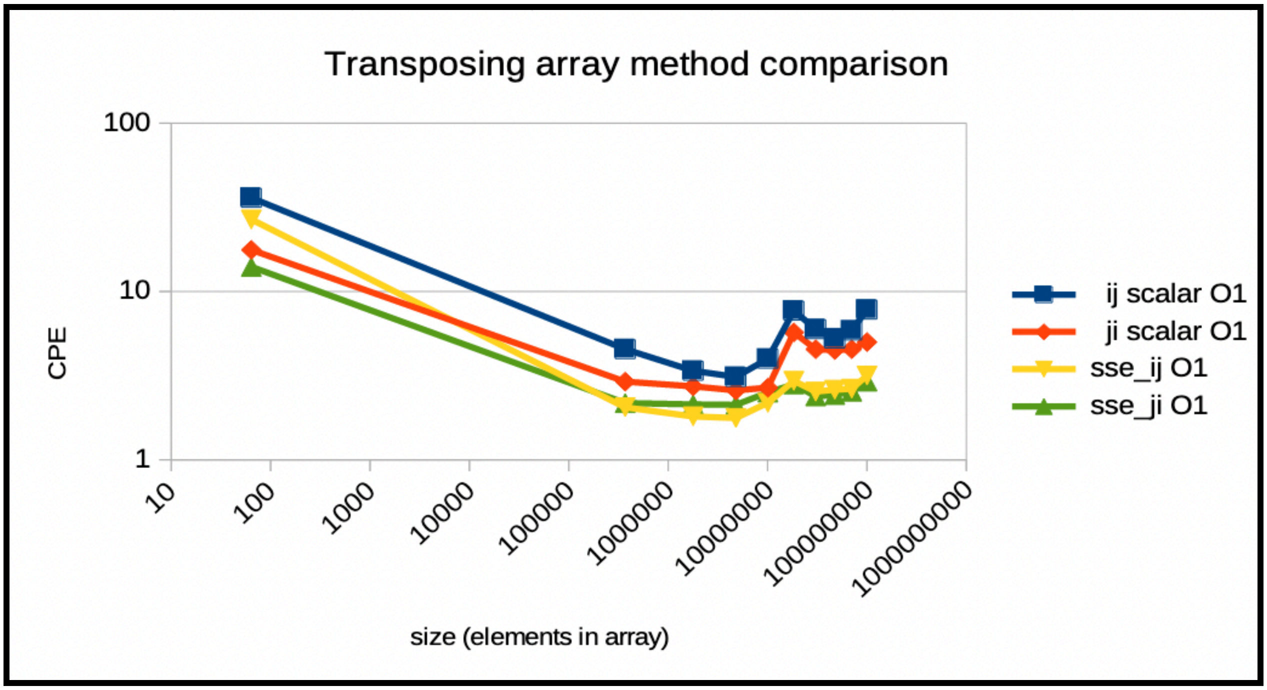
2e. How does this approach compare with test_dot8.c? What are the specific differences (performance, programmability, etc.) and when do they matter?

I found that the dot product scalar version was much more intuitive to the way I learned how to program. This approach is the most heuristic to how dot product is taught, but it isn't that efficient in terms of coding. The differences are that performance becomes twofold, programmability is harder in terms of heuristic thinking, and they will matter when you aim to lower your runtime for more complex situations. For an average-day programmer, a difference in 2.5 cycles might not mean much if it is per-element, but if you are trying to get results for a very complex structure, for example, a molecular charge program for a chain of molecules, you would want to have as efficient of a program as possible.

Part 3: A simple SSE application from scratch: Transpose

3a. Create the fastest transpose you can.

Using the SSE transpose intrinsic and blocking for a 4x4 matrix, I achieved a speedup of 1.72x and 1.47x for ij and ji transpose methods. I did this with SSE transpose intrinsic and combined blocking with code as reference from Lab 1.



3b. Compile test_transpose.c with -O2 and -O3 options and compare with your version.

I plotted the scalars from each type of optimization, as well as both versions of sse for each type of optimization. I found the speedup for *ij* and *ji* for each level of optimization for the compiler and found the following:

O1:

- *ij*: 172.460131772485
- *ji*: 146.722938139198

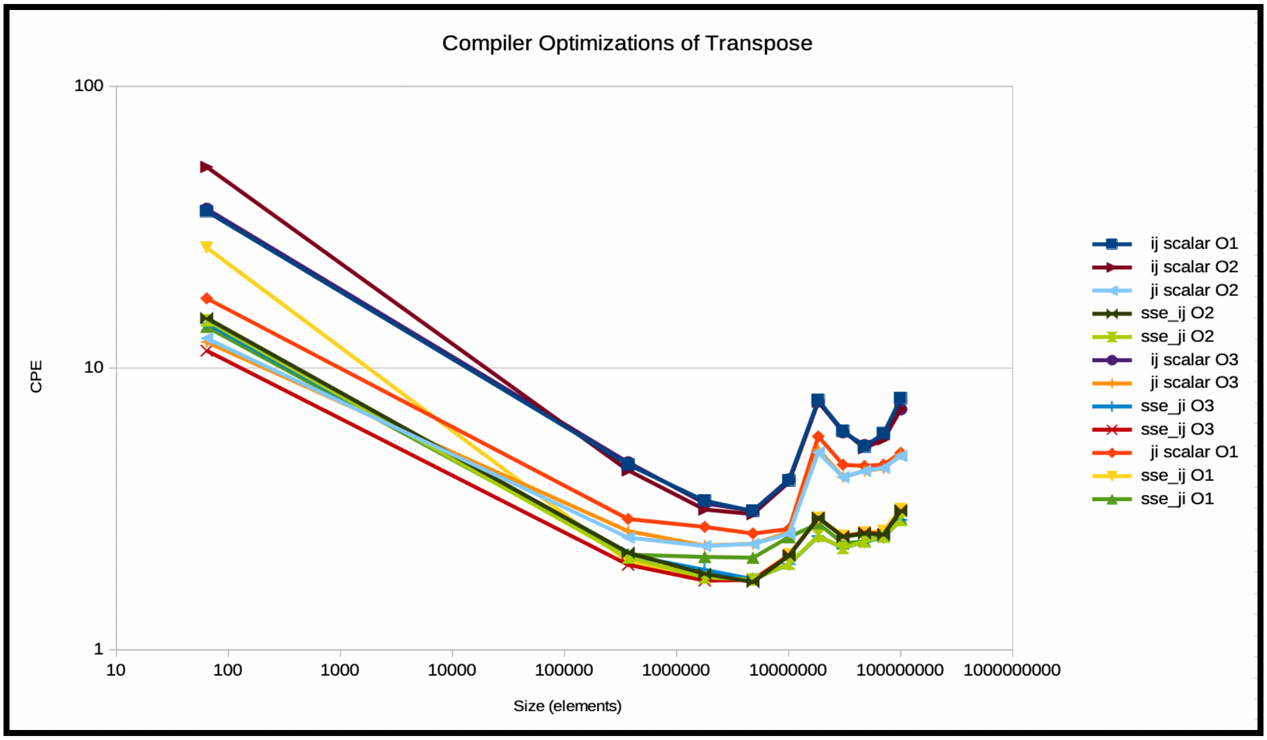
O2:

- *ij*: 265.927452863931
- *ji*: 128.748058123099

O3:

- *ij*:253.41010550179
- *ji*: 129.465741318054

Unexpectedly, it appears that sse_ji 02 and sse_ji 03 perform about the same as the size of the array increases. *ij* and *ji* do have a bit of a difference in all three optimizations, with *ji* being faster than *ij*.



Part 4: QC

4a. How long did this take?

This took longer than expected, I think it took about 12 hours.

4b. Did any part take and “unreasonable” amount of time for what it is trying to accomplish?

Nothing felt unreasonable, but it did take a long time to figure out the transposing with SSE intrinsic for part 3.

4c. Are you missing skills needed to carry out this assignment?

I did not think so

4d. Are there problems with the lab?

I could not find the L03 PDF that part 2 question 2d referred to. This was a little confusing, but I used Intel's website instead.