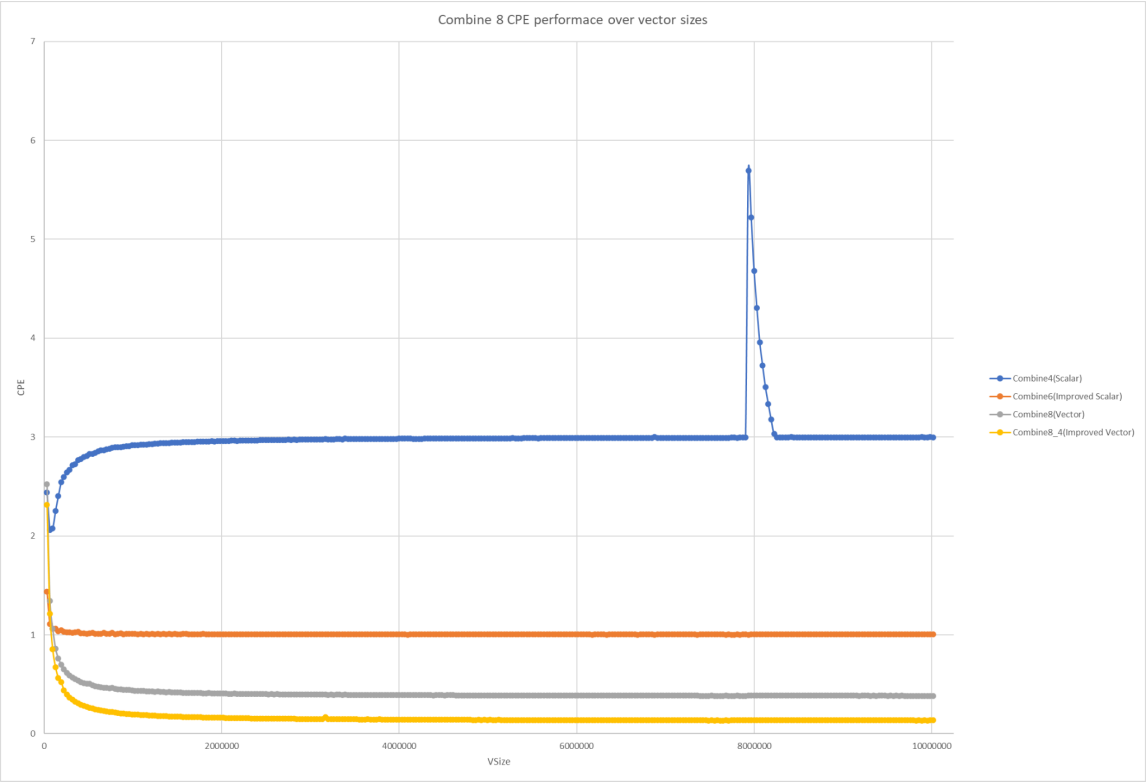
Evan Lang, Yang Lu
EC 527: Lab 3

1.

a.

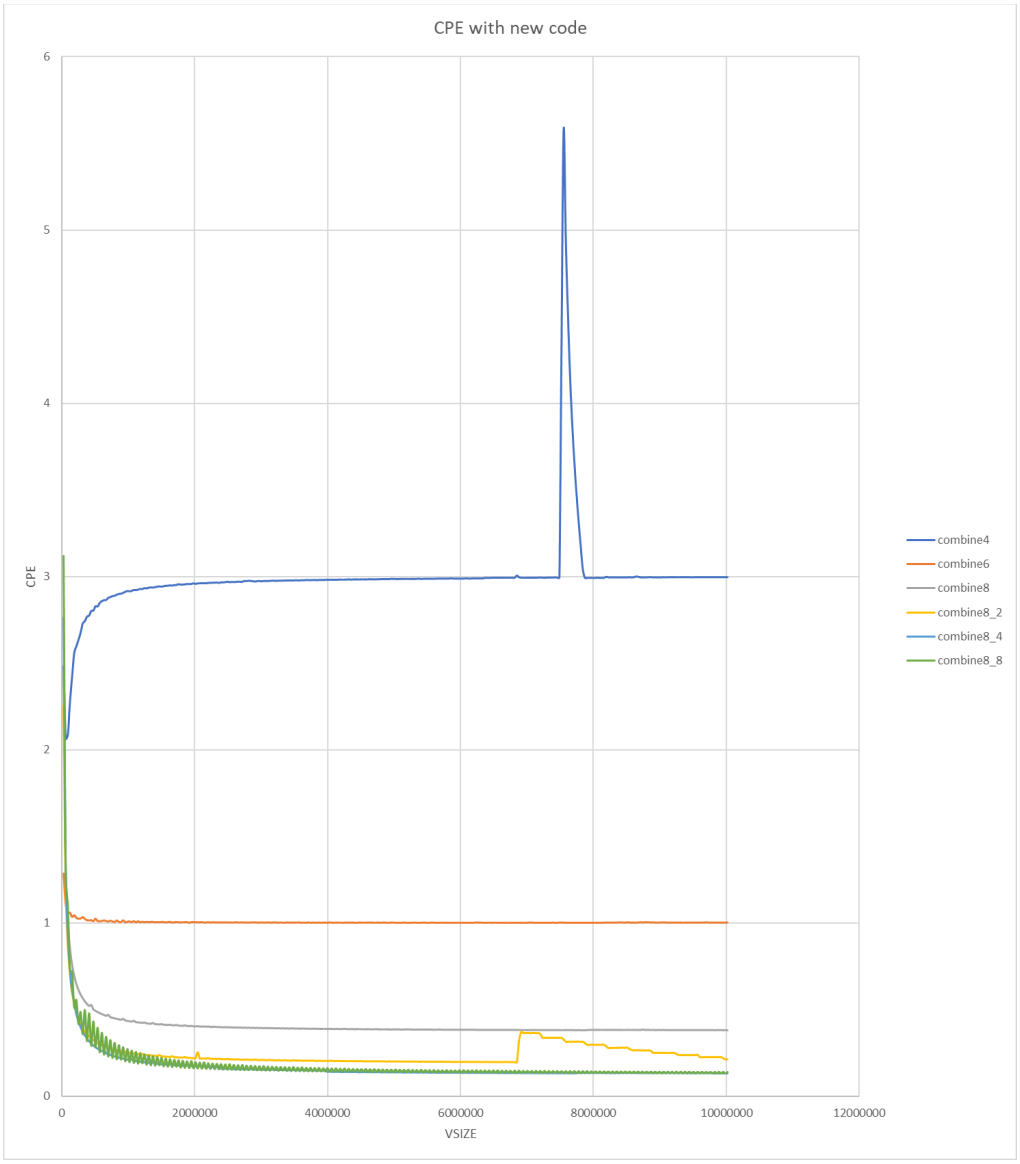


Compared to the scalar results, the vector results are seriously improved. Using the vectors over scalar adds a 6x improvement. Adding the improvements even more so. This can be attributed to the efficiency increases inherent to vector improved code and the hardware accelerated instructions that run them.
The data relevant to this can be found in the included file Lab3Q1a.csv

Average CPE:

combine4	combine6_5	combine8	combine8_4
2.990704	1.007317	0.415936	0.170367

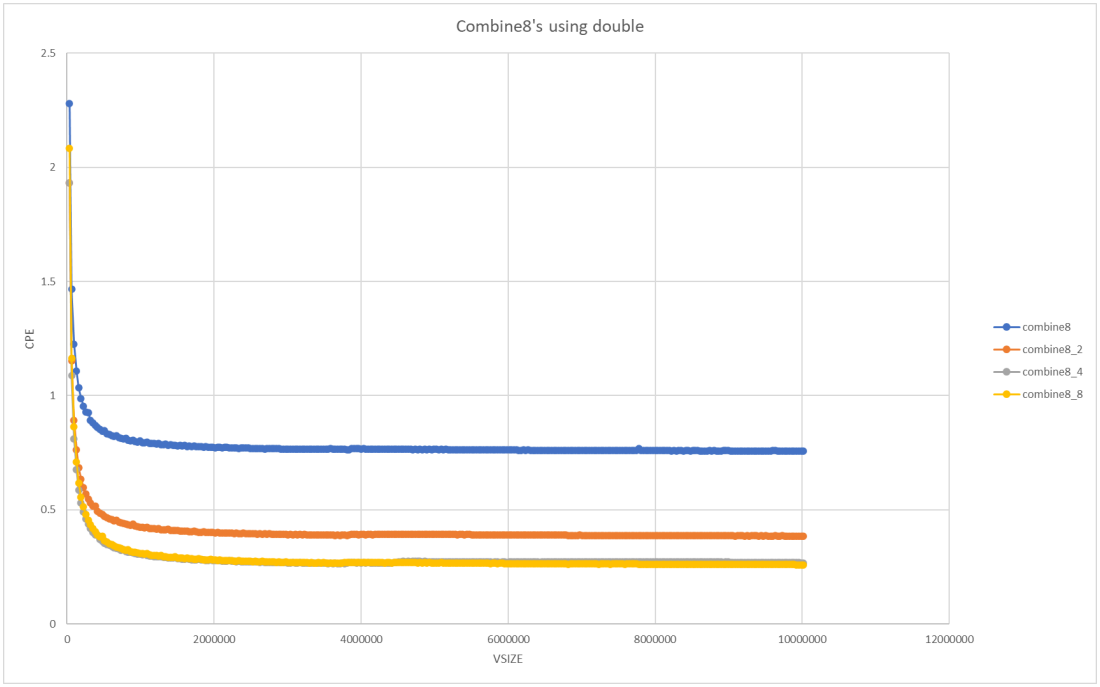
b.



Unsurprisingly the 2 accumulator method presents slightly degraded performance when compared to the 4 accumulator method. What is more surprising is the noise seen when using 8 accumulators. While the average CPE is very close to 4 accumulators the noise now seen shows it is less ideal. This may be due to memory scaling although all increments of size are multiples of 32. *Data for this can be seen in Lab3Q1b.csv*

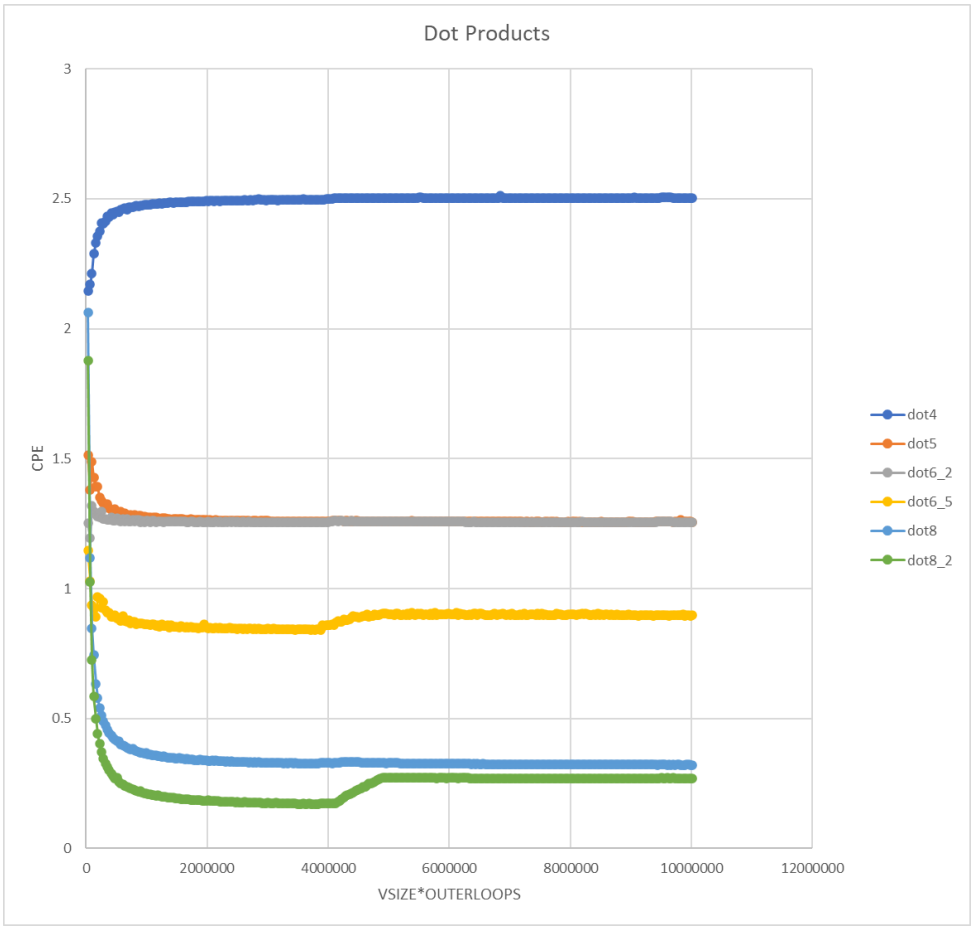
combine8_2	combine8_4	combine8_8
0.257238	0.177663	0.189288

c.



Yes, in both float and double 8 accumulators provide almost the exact same performance as 4. However when using double noise of combine 8 is significantly improved. It also maintains a slightly better average CPE of .25 for 8 accumulators versus .26 for 4 accumulators. *Data for this can be seen in Lab3Q1c.csv*

d.



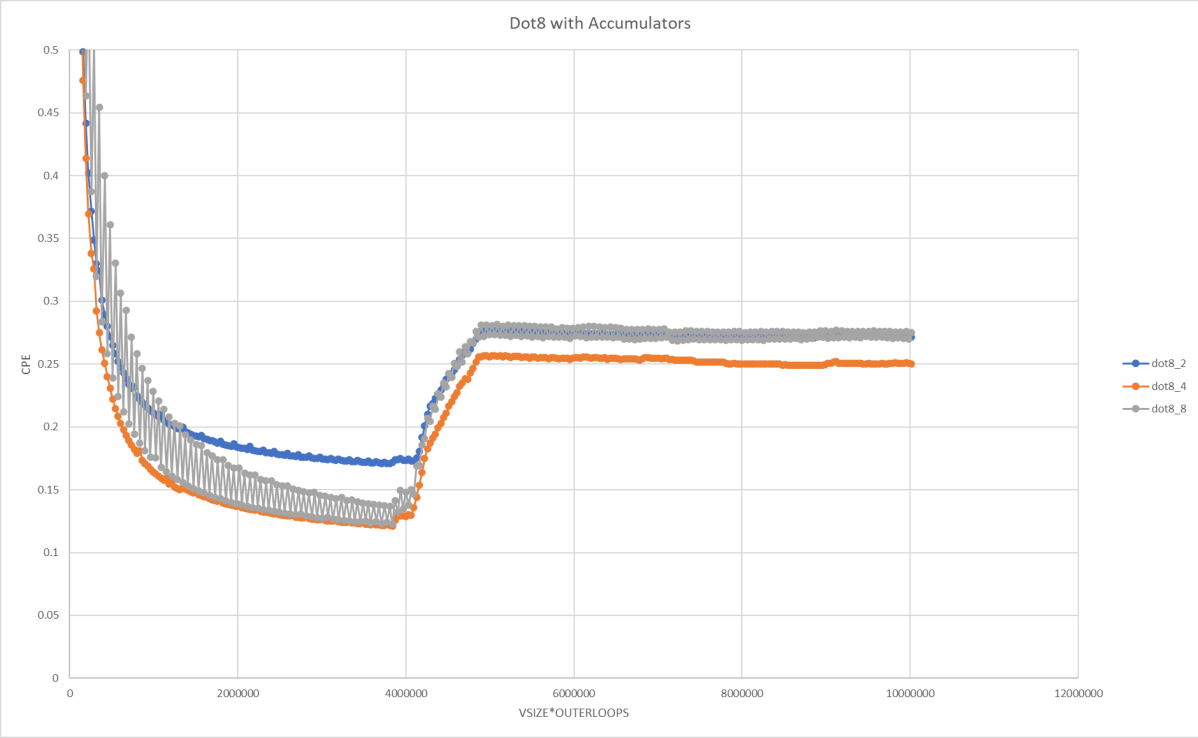
Once again the vector dots (dot8 and dot8_2) had significantly elevated performance when compared to the multiple scalar strategies. Even using the optimization strategies like loop unrolling and accumulators on scalar did not bring the performance even close to the vector methods. It should be noted that the large benefit of the 2 accumulators used in 8_2 began to degrade around 4200000 VSIZE*Outer Loops. While the accumulators stilled provided somewhat of a benefit, after 4500000 it was significantly reduced. *Data for this part of the lab can be found in Lab3Q1d.csv*

e.

```
620   /* Single step through remaining elements */
621   while (cnt>0) {
622     result += *data0++ * *data1++;
623     cnt--;
624   }
```

I had to change line 621 from cnt >= 0 to cnt > 0. This prevented it from over looping one too many times.

f.

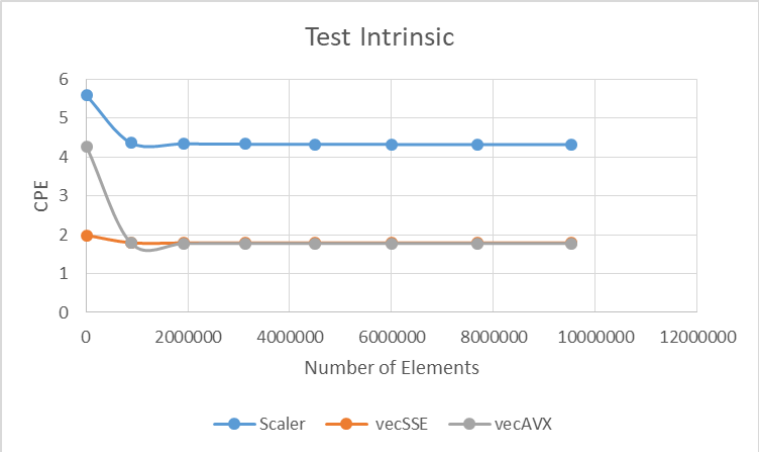


Once again all 3 accumulators perform better than no accumulators and scalar. Similarly 4 Accumulators perform the best for dot product as it avoids any memory issues seen in the noise for 8 accumulators and better performance than with only 2 accumulators. Once again this noise in the 8 accumulator method can be attributed to memory misses due caused by the size of the unroll. *Data for this portion of the lab can be found in Lab3Q1f.csv*

2.

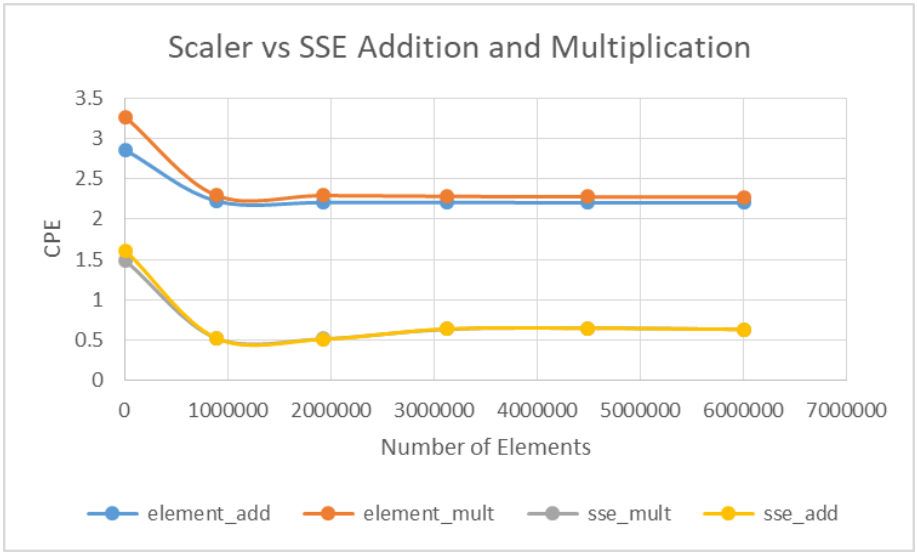
a. unalign_heap_naive is the function that does not work. This is due to the fact that __m256 accepts a group of numbers containing a total of 256 bits to allow for vector arithmetic. The loops in the unalign_heap_naive function are attempting to input one number at a time into the _m256 datatype which will ultimately lead to a compiler error.

b.



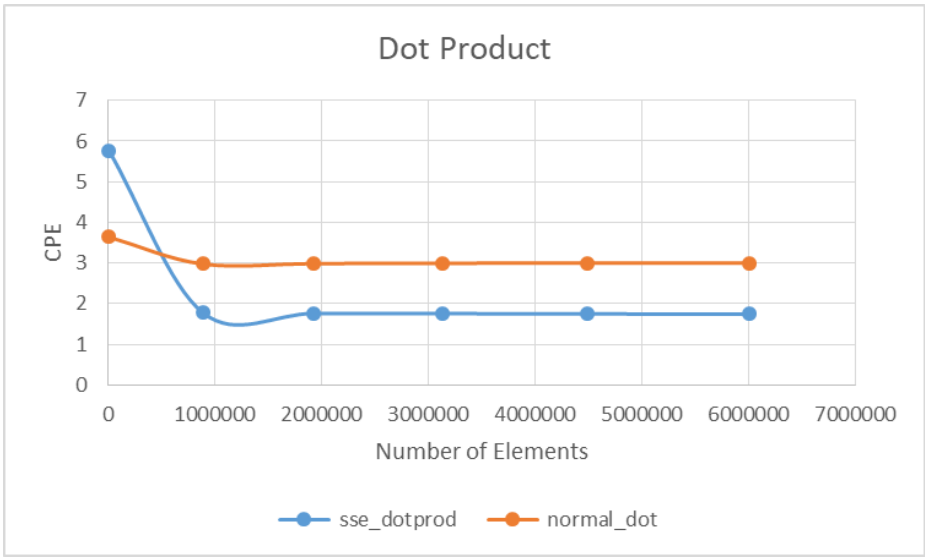
Above is the results from running the test_intrinsic.c file. As expected the SSE and AVX instruction set are much much more efficient than the scalar equivalent. The AVX and SSE instructions on the other hand are relatively similar in terms of CPE. AVX starts out slightly worse than the SSE equivalent, however, over time the AVX method will yield a imperceptible improvement over the SSE equivalent.

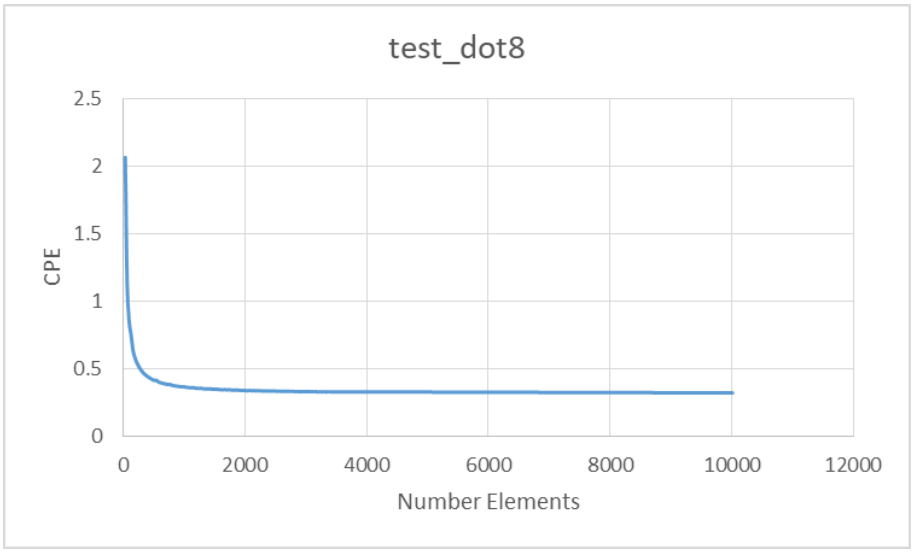
c.



Similar to part 2b. Utilizing SSE vectorization yields a much more efficient CPE than that of using the scaler equivalents. Note: this code is contained within test_intrinsics.c.

d.



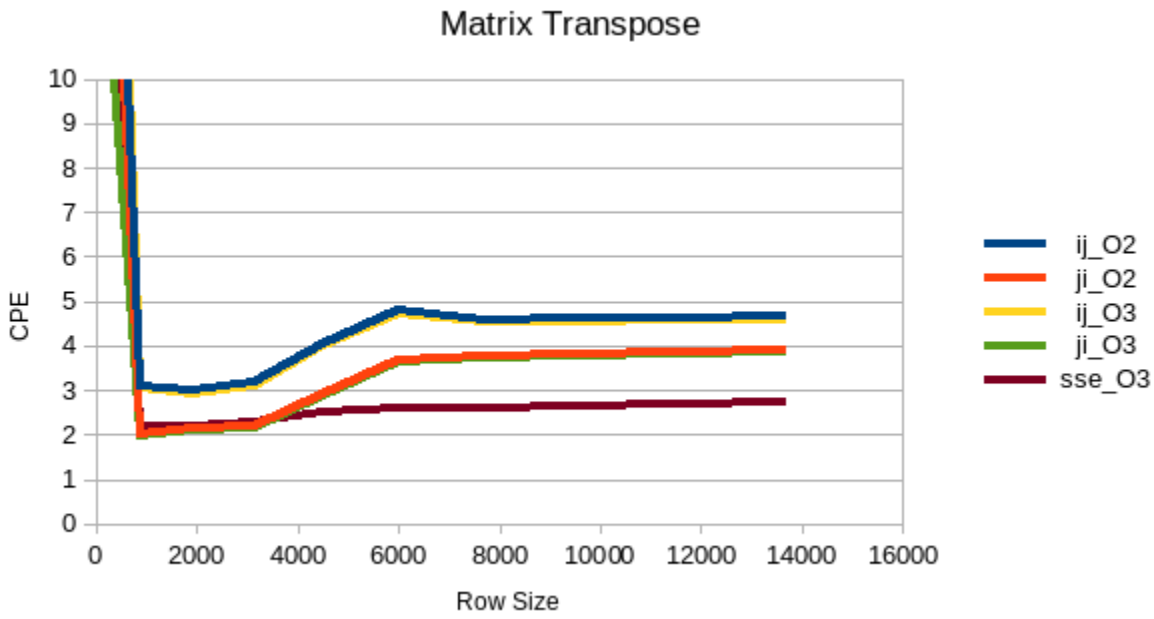


As seen in the 2 graphs above. Test_dot8 will have a CPE of less than 0.5. While the SSE dot product will have a CPE of slightly less than 2 after it reaches equilibrium. Based on this, it can be seen that test_dot8 is much more efficient as it reaches equilibrium sooner and has a lower CPE. Note: this code is contained within test_intrinsics.c.

e. Test_dot8 is performing the dot product by computing a groupwise single loop unroll partial dot product, with size vsize, and storing the result in 2 accumulators. Then it computes the partial dot products of the elements that didn't fit within vsize at the end and performs scalar dot products. Finally it sums all the accumulators to get the final dot product. Although this method is efficient, it is very difficult from a programmability standpoint, and is reliant on the data having very few elements that don't fit within the vsize. The more elements that don't get

fitted into vsize the more inefficient this algorithm will be. The method used for sse dot product on the other hand, although inefficient, is much easier to program. It utilizes sse intrinsics to perform the dot product on 4 numbers and then sums it into a accumulator. This method scales much more optimally in comparison to the test_dot8 method, although requiring much more CPE.

3.



As seen above, using SSE intrinsics creates a much more optimized variation of the matrix transpose code. Even with the most optimized compiler, scalar arithmetic is significantly worse than using sse vector intrinsics. Specifically, SSE intrinsics allows for a 33% improvement in CPE when compared to an O3 scaler optimization, and a 60% improvement when compared to an O2 scalar optimization upon reaching equilibrium.

4.

a. This lab to approximately 12 hours
b. Part 3 took significantly longer than expected as the results on our local machine did not match the results on eng-grid. While we were seeing a marked speedup on our local machines when compared to the original test_transpose, we encountered a perceived slow down when running the code on eng-grid. This was eventually resolved when discussed with the TA's in office hours, who gave us permission to use our local runs for data.
c. We were not missing any specific skills for this lab.
d. Outside of the previously discussed difficulties in part 3, there were no problems with this lab.