**Part 1**

As it can be seen from the graphs, the most efficient functions are the CPE 8\_4 (the Vector 4x unrolled version) & the CPE 8\_8 (the Vector 8x unrolled version). However the CPE for 8\_4 is minutely better than the CPE for the 8\_8 version. The vector parallelization is definitely better than the scalar, since greater parallelization is achieved through more elements being engaged in each iteration for the vectorized version. The 8\_4 version might be slightly better than the8\_8 version as a CPU pipeline has only 5 stages, so having an unroll factor of 8 might not be helping.

b) Using Double

Again similar to the results with the floats, the CPE for the double version, can be seen to be lower for the vectorized version compared to the scalar version. The reasoning would be similar to the one above- greater parallelization is achieved through more elements being engaged in each iteration for the vectorized version. In this case however, the CPE 8\_8 is slightly better than both the CPE8\_4 & CPE8\_2, which suggest that having an unroll factor of 8 does help.

c) Test\_dot.c

In this case it can be seen that having an unroll factor of 5 in the scalar version of the dot product is more efficient than all the vectorized version and scalar versions of the function. In the case of the vectorized versions, it can be seen that having an unroll factor of 2 compared to no unroll factor is more efficient as one would expect, since it makes use of parallelism.

d) All vectorized versions of dot product

In this particular case for vectorized dot products, it can be seen that having no unroll factor has a lower CPE than having unroll factors of 2,4 & 8. It can also be seen that although increasing the unroll factor from 2 to 4 leads, to a small dip in CPE, increase from 4 to 8 causes an increase in CPE. This might again be owing to the fact that CPU has only five pipeline stages, so no advantage of increasing the unroll factor.

**Part 2**

This is somewhat what I expected since, CPE\_Test2 has the lowest CPE, and this uses intrinsics to calculate the distance, as opposed to just vectorized code. However CPE\_Test3 (uses 256 bit intrinsics) is slower than CPE\_Test1 which just uses vectorized code. This is because the 256 bit data are split into two 128 bit data and then the operations are performed, so theoretically the CPE\_Test2 should be 2 times faster than CPE\_Test3, since it is performing half the operations.

b)

c)

It can be seen that the intrinsic dot product is a lot faster than the vectorized dot product. This is expected.

e) Using intrinsics provides definite benefits in performance when compared to regular vectorizable code, as can be seen with the examples of the dot product & the distance calculation. It can also be seen that there is much greater ease of programmability with intrinsics compared to the vectorized code for the dot product. This matters especially as the unroll factor starts to increase and the code gets more complicated. The dot product using the intrinsic version provides a CPE that is 3.1 times lower than the vectorized version where no unroll factor is used. It can also be seen from the above graph that the CPE for the intrinsic dot product is lower than the CPE for the vectorized dot product for all vector sizes. In the distance calculation, using the 128 bit Intrinsic code provides a CPE that is 3.8 times lower than that using vectorized code where no unroll factor is used. The CPE for the 128 bit intrinsics version is lower than the CPE for the vectorized version for all vector lengths.

**Part 3**

It can be seen from the chart that CPE\_transpose\_intrinsic is faster than the transpose rev and regular transpose.

From the chart it can be seen that there is not much difference between O1, O2 & O3 optimizations in the transpose function.

**Extra Credit**