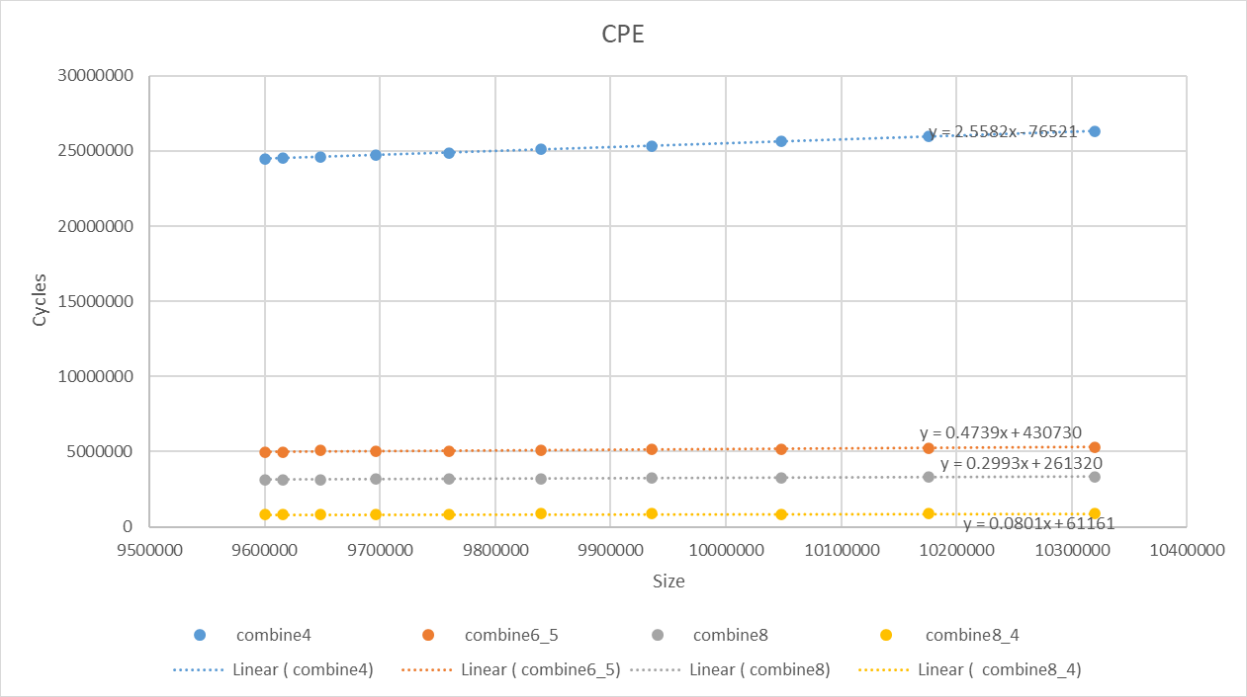
EC527
Lab3 report
Bin Xu
Seyed Reza Sajjadinasab

Part 1

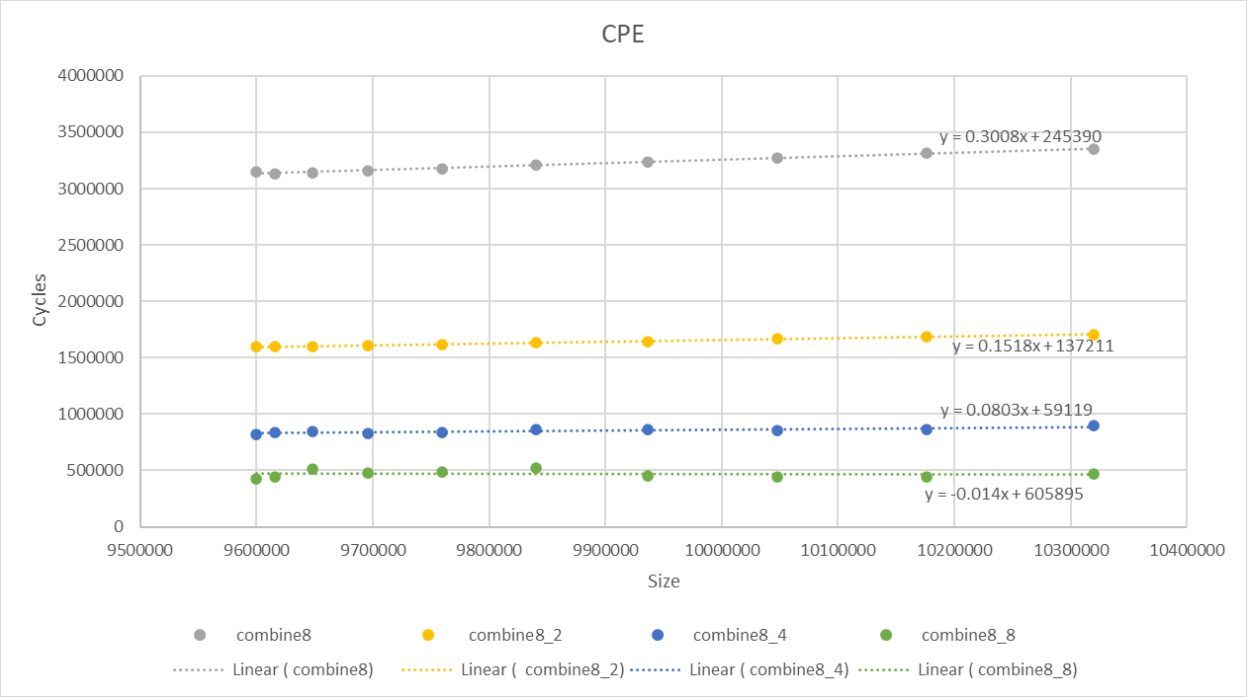
1a.

VSIZE is 8, so I set A, B, C to be 8, 8, 9600 proponents to the VSIZE. The reason why I choose C to be 9600 is because doing this makes Ax2+Bx+C to be near 10000.

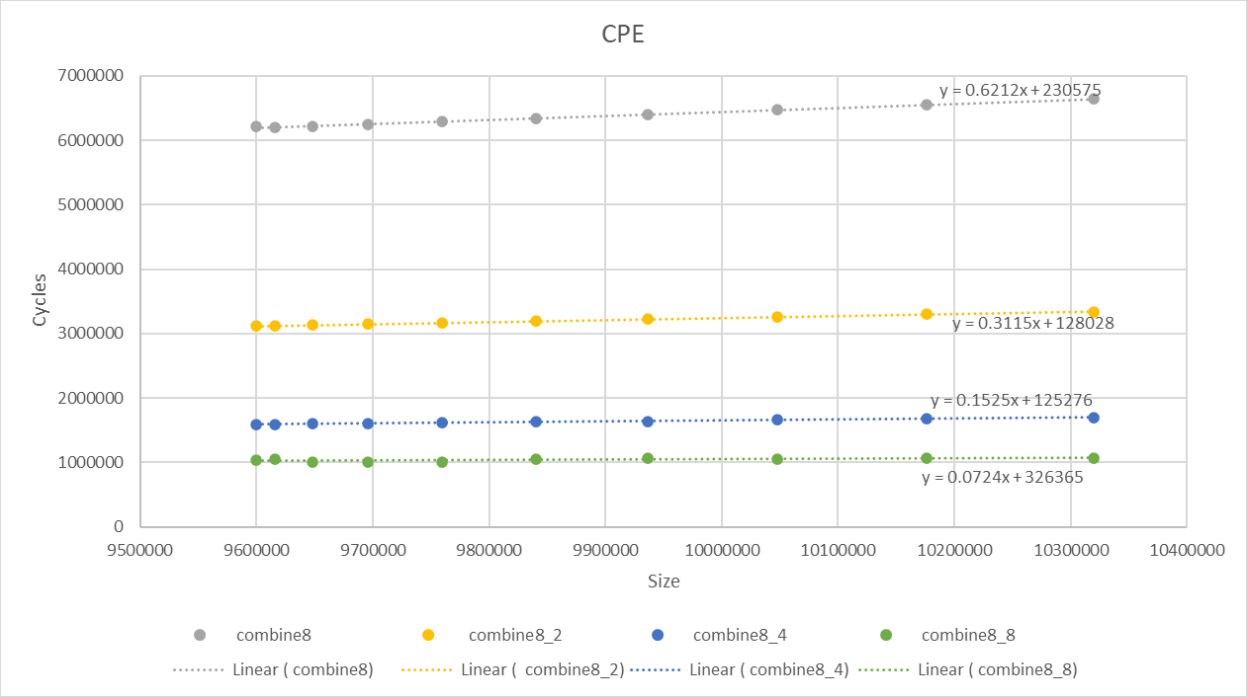


1b.

As we can see in the graph by increasing the number of accumulators the CPE is decreasing significantly. This indicates that more accumulators help parallelism in the code, and enhance the performance of the program.



1c.



As we can see, although the CPE of each corresponding combine is bigger than the one with floating points, having 8 accumulators still has benefits, and CPEs of those that have more accumulators are much better.

1d.
By changing if(0) to if(1) we have:
dot product examples
dot4(v0, v1, 10) == 285
dot8_2(v0, v1, 10) == 385
That shows the incorrectness of dot8_2.

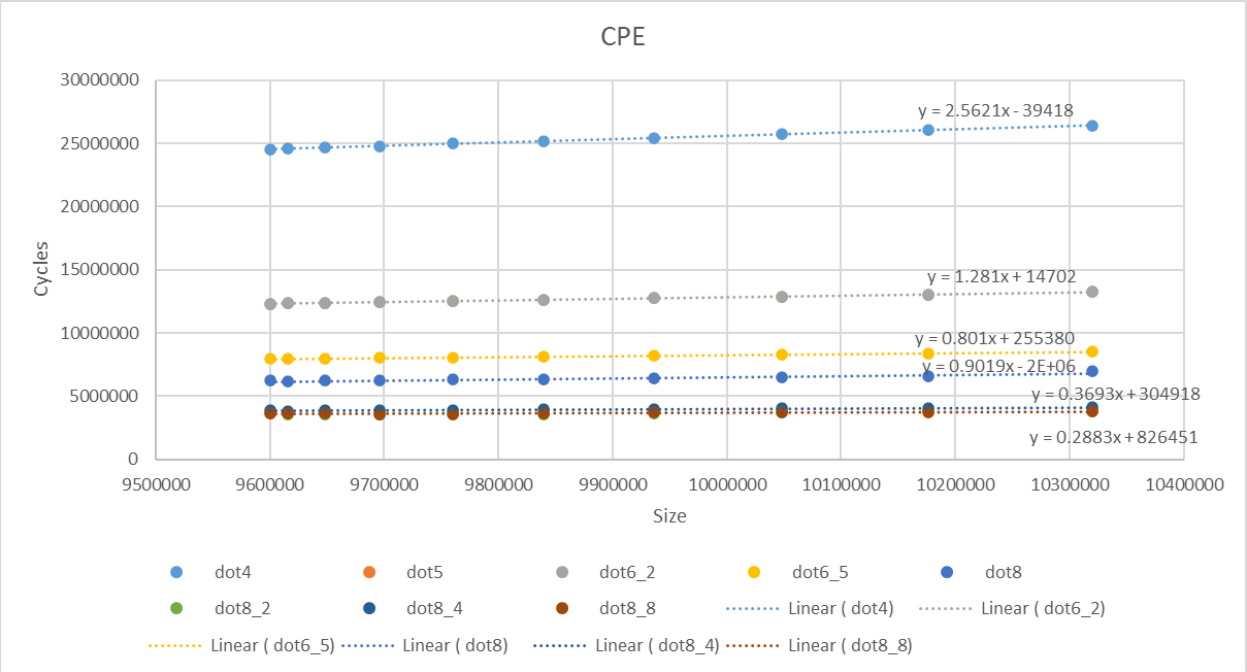
Also with if(1) we have:
Computed dot products:

size	dot4	dot5	dot6_2	dot6_5	dot8	dot8_2
10,	216.04,	216.04,	216.04,	216.04,	216.04,	318.22
12,	365.57,	365.57,	365.57,	365.57,	365.57,	446.03
16,	612.28,	612.28,	612.28,	612.28,	612.28,	781.44
22,	1477.2,	1477.2,	1477.2,	1477.2,	1477.2,	1749.7
30,	5316.5,	5316.5,	5316.5,	5316.5,	5316.5,	5607.7
40,	10790,	10790,	10790,	10790,	10790,	10918
52,	26025,	26025,	26025,	26025,	26025,	26292
66,	45278,	45278,	45278,	45278,	45278,	49331
82,	90031,	90031,	90031,	90031,	90031,	94070
100,	1.8933e+05,	1.8933e+05,	1.8933e+05,	1.8933e+05,	1.8933e+05,	1.8933e+05

As we can see in larger sizes the error is not obvious.

By debugging the code we find the problem. The problem in this 8_2 is that in the `Single step through remaining elements` the condition is cnt>=0. This is not valid because the initial value of cnt is the length of the array, and by doing this we are doing the math one more time than needed. So, by changing the condition to cnt>0 we'll have a valid dot function.
Note: Since in larger sizes the last operation becomes less effective compared to many other operations, we saw that in larger sizes the error was negligible.

1e.
As expected CPE is improving, but for dot8_8 vs dot8_4 there is no significant difference.



Part 2

2a.

Not all methods work. Unalign_local_alloc, align_heap_1,unalign_storeu_ps work.

Segmentation fault occurs when the program tries to access the memory location that it is not allowed to access.

AVX instructions require memory accesses to be aligned to 32-byte boundaries, and the code in unalign_heap_naive may not guarantee this alignment.

After we make new_method unalign_heap_naive, we get below results.

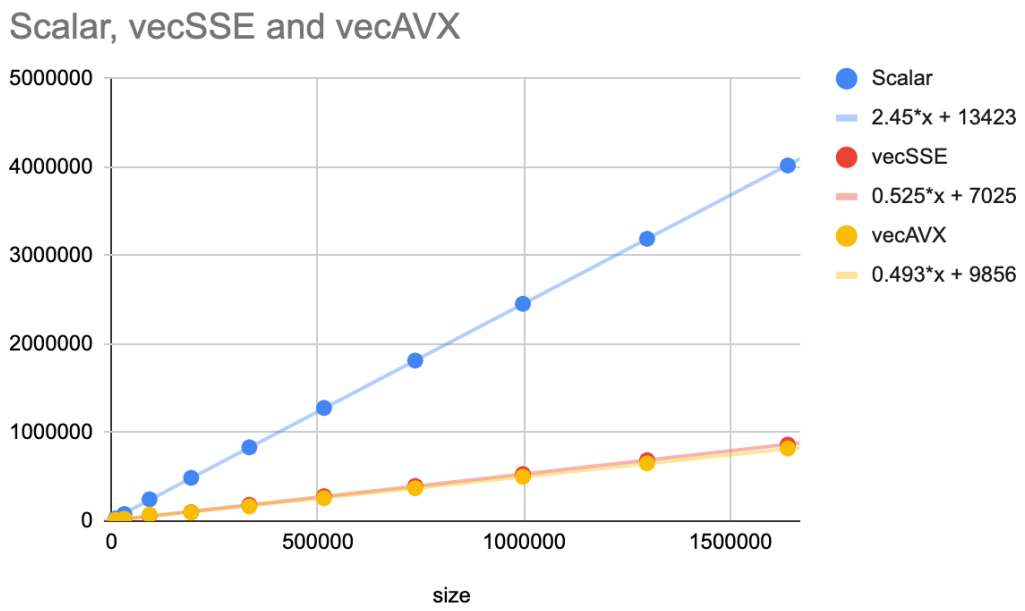
```
AVX load/store alignment tests
unalign_local_alloc:
  p1 == 0x7ffdc11d7f94   p2 == 0x7ffdc11d7f98
     2      3      4      5      6      7      8      9     10     11
     2      2      3      4      5      6      7      8      9     11
     2      2  1.414  1.732      2  2.236  2.449  2.646  2.828      3

new_unalign_heap_naive:
     2      3      4      5      6      7      8      9     10     11
     2      2      3      4      5      6      7      8      9     11
     2      2  1.414  1.732      2  2.236  2.449  2.646  2.828      3

align_heap_1:
  p1 == 0xdce040   p2 == 0xdce060
     2      3      4      5      6      7      8      9     10     11
     2      3      4      5      6      7      8      9     18     19
  1.414  1.732      2  2.236  2.449  2.646  2.828      3     26     27

unalign_storeu_ps:
  p1 == 0xdce024   p2 == 0xdce028
     2      3      4      5      6      7      8      9     10     11
     2      2      3      4      5      6      7      8      9     11
     2      2  1.414  1.732      2  2.236  2.449  2.646  2.828      3
```

2b.



Yes, the vectorized methods are faster than the non-vectorized one.

2c.

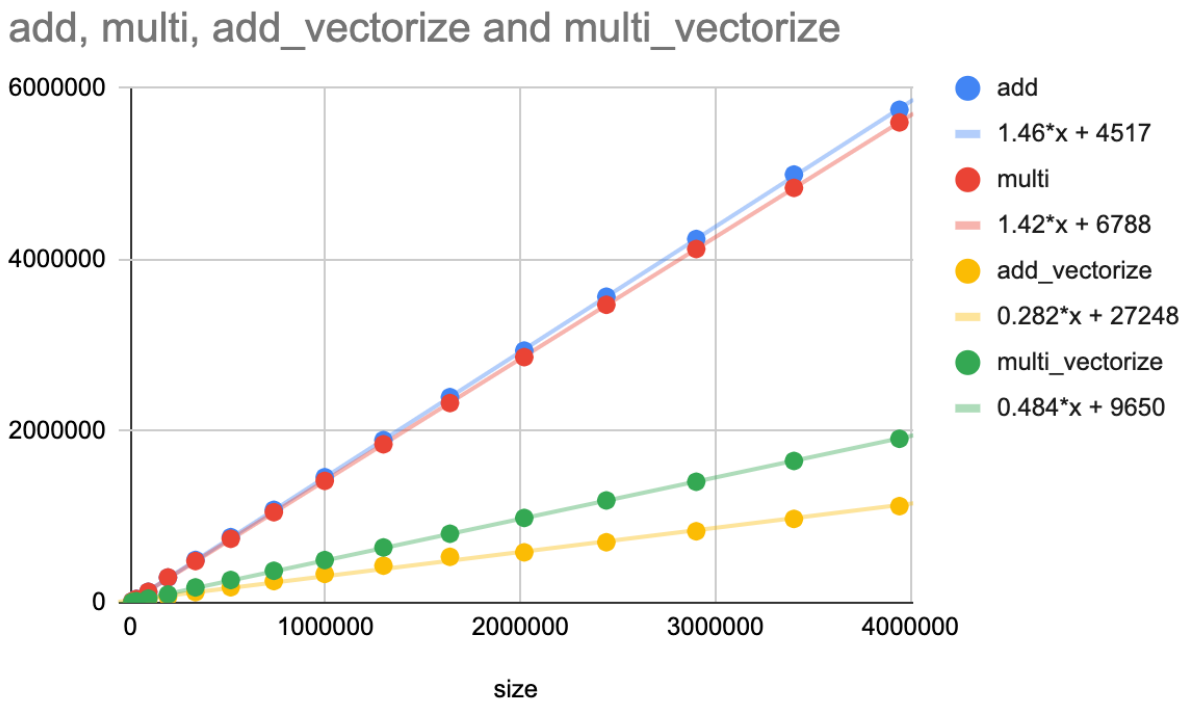
```
void add (data_t* pArray1,       // [in] 1st source array
        data_t* pArray2,       // [in] 2nd source array
        data_t* pResult,       // [out] result array
        long int nSize)     {
  data_t* pSource1 = pArray1;
  data_t* pSource2 = pArray2;
  data_t* pDest  = pResult;

  for (int i = 0; i < nSize; i++) {
     pDest[i] = pSource1[i] + pSource2[i];
  }
}

void add_vectorize(data_t* pArray1,       // [in] 1st source array
                data_t* pArray2,       // [in] 2nd source array
                data_t* pResult,       // [out] result array
                long int nSize) {
  long  nLoop = nSize / 4;
  __m128*  pSrc1 = (__m128*) pArray1;
  __m128*  pSrc2 = (__m128*) pArray2;
  __m128*  pDest = (__m128*) pResult;

  for (long i = 0; i < nLoop; i++){
     *pDest = _mm_add_ps(*pSrc1, *pSrc1);

     pSrc1++;
     pSrc2++;
     pDest++;
  }
}
```



From the above CPE chart, the vectorized methods are much faster than non-vectorized methods. The performance has improved a lot.

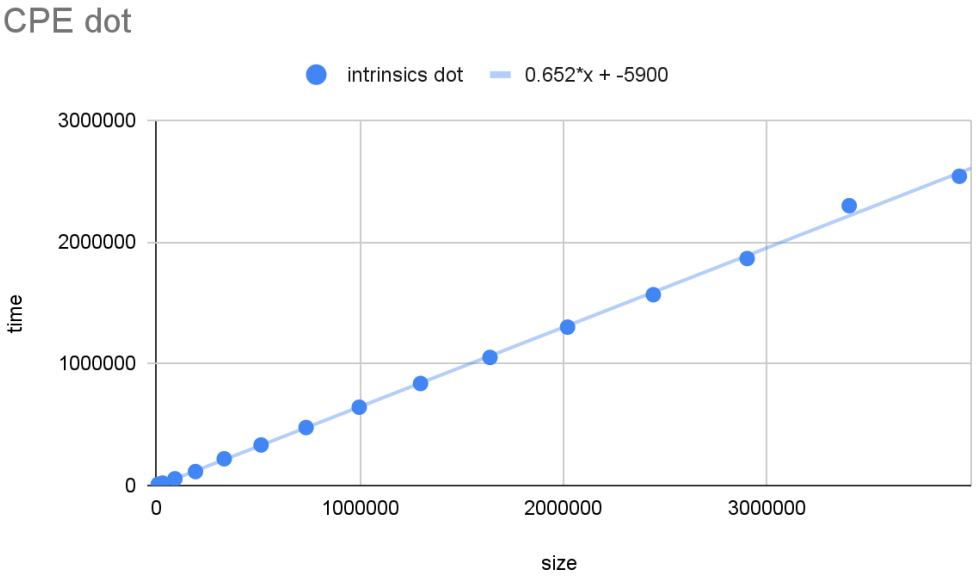
2d.

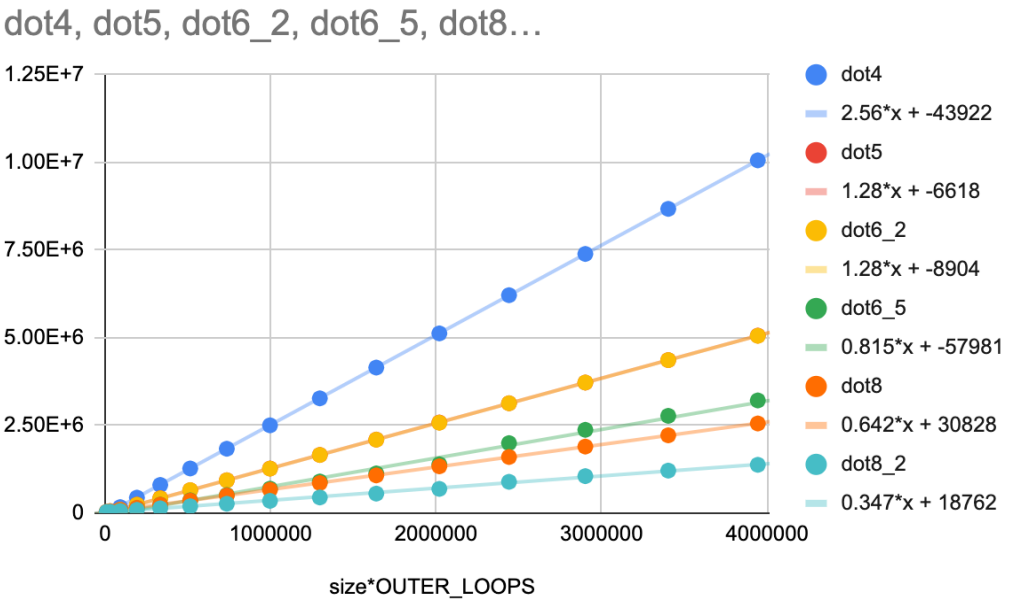
```
void dot(data_t* pArray1,       // [in] 1st source array
      data_t* pArray2,       // [in] 2nd source array
      data_t* pResult,       // [out] result array
      long int nSize) {
      long  nLoop = nSize / 4;

  __m128*  pSrc1 = (__m128*) pArray1;
  __m128*  pSrc2 = (__m128*) pArray2;
  __m128*  pDest = (__m128*) pResult;
  __m128 a,b,c;

  c = _mm_setzero_ps();

  for (int i = 0; i < nLoop; i++) {
     a = _mm_loadu_ps(pSrc1 + i);
     b = _mm_loadu_ps(pSrc2 + i);
     c = _mm_add_ps(c, _mm_dp_ps(a, b, 0xff));
  }
  _mm_store_ss(pDest, c);
}
```





The performance of intrinsics is close to the performance of dot8 which uses vector attributes. CPE of intrinsics is 0.652, and CPE of dot8 is 0.642. Dot8_2 which uses 2 accumulators is faster than dot_intrinsics.

2e.

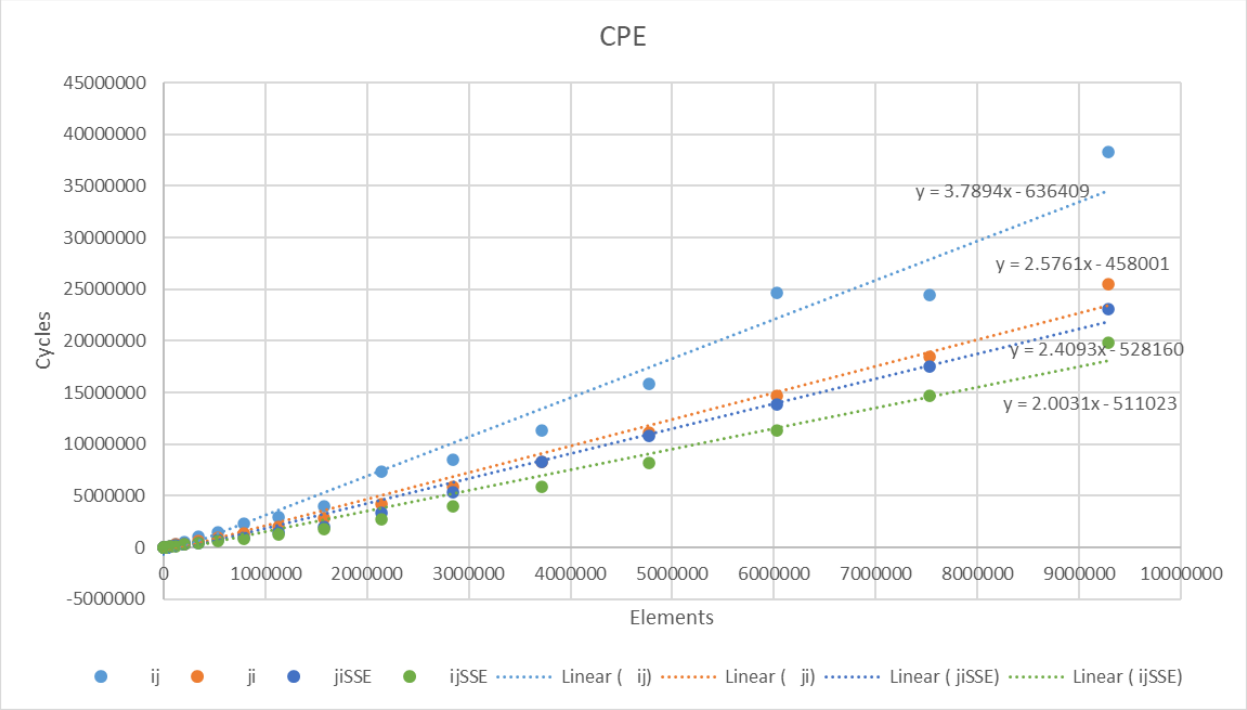
Intrinsics are usually defined in a header file provided by the compiler, such as immintrin.h.

On the programmability side, the main difference between the two approaches is that intrinsics provide more control over the use of SIMD instructions, while __attribute__((vector_size(VBYTES))) provides a more concise syntax for declaring and

manipulating vectors. intrinsics are more suitable for complex vectorization tasks, while __attribute__((vector_size(VBYTES))) is more suitable for simple vectorization tasks.
On the performance side, SSE intrinsic programming takes care of data alignment. So it provides better performance. Vector version's data alignment has to be taken care manually.

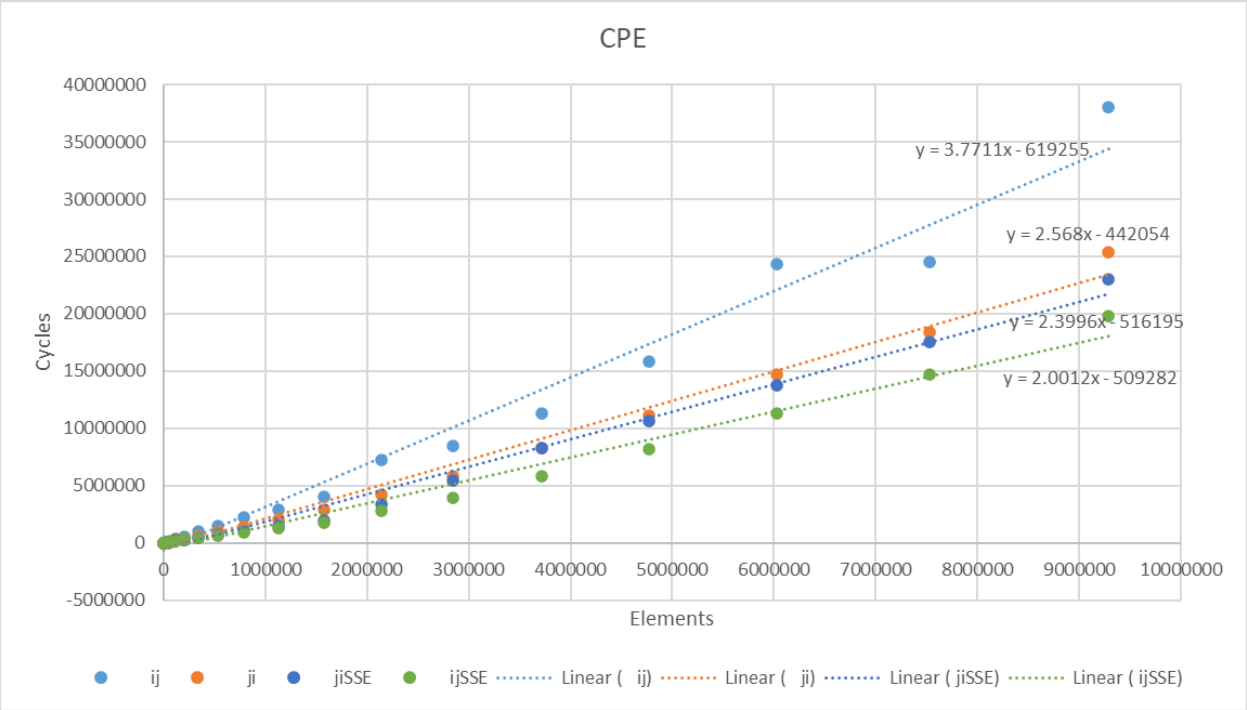
Part 3

3a.
Compared to the original versions, we obtain these results by rewriting the code with SSE.

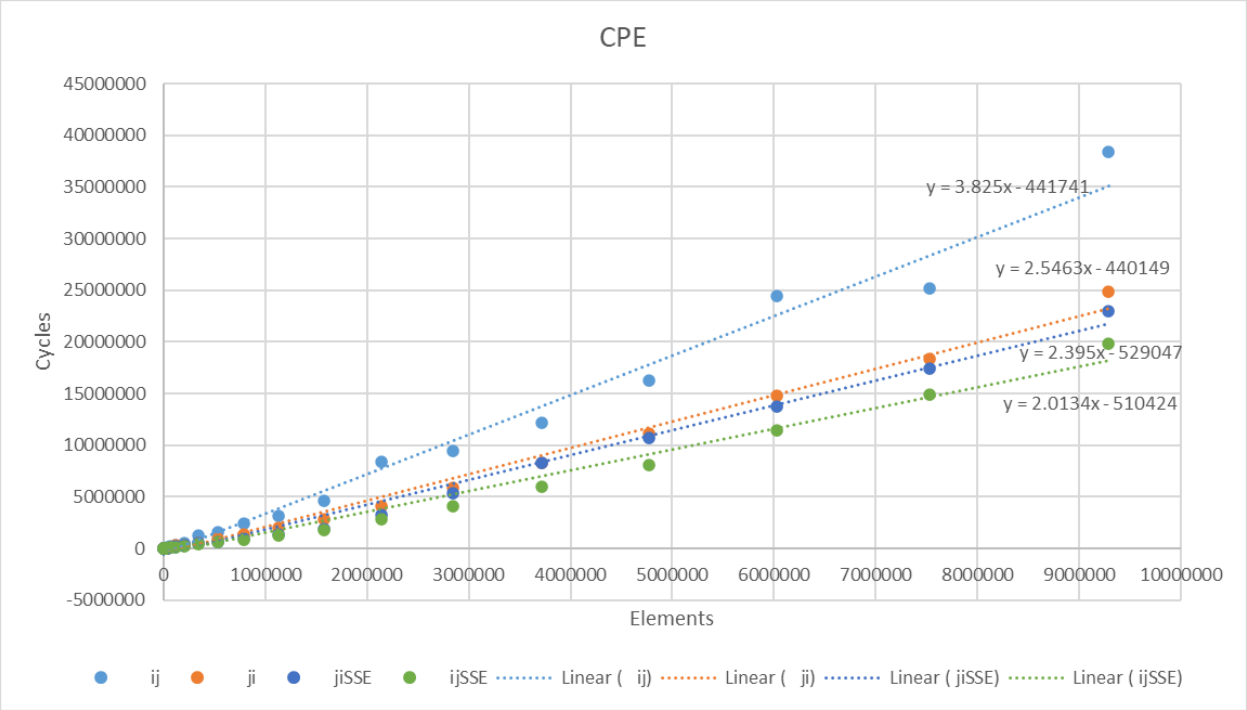


Compiling all the results with -O1 result as above which indicates the ijSSE version is the best version we can have.
Note: In the original version ji was better than ij!

3b.
Compile using O2:



Compile using O3:

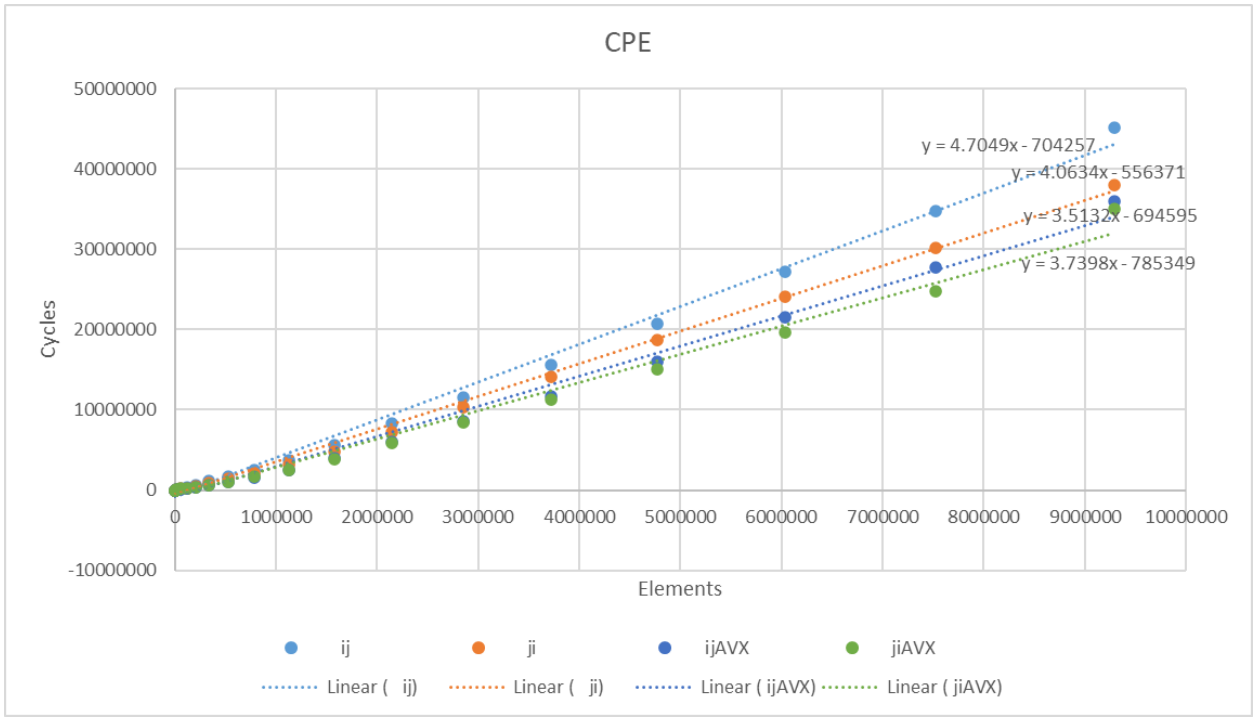


It doesn't make a significant difference. We used `__attribute__((optimize("O3")))` to identify the level of optimization for each function. This is why the performance of SSE versions have drifted a little bit.

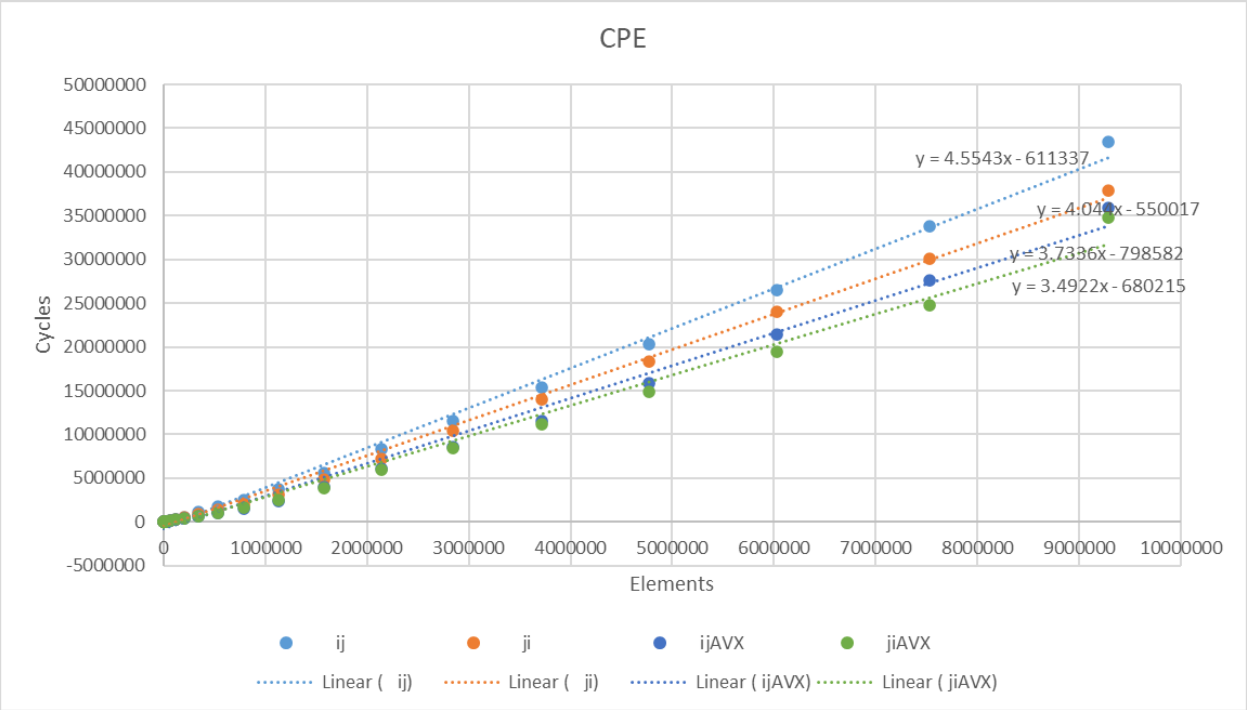
Extra.

In this part since we have to change the data_t from float to double we cannot report the result for SSE, and they aren't valid.
For every level of optimization for simple transpose functions we obtained the below graphs.

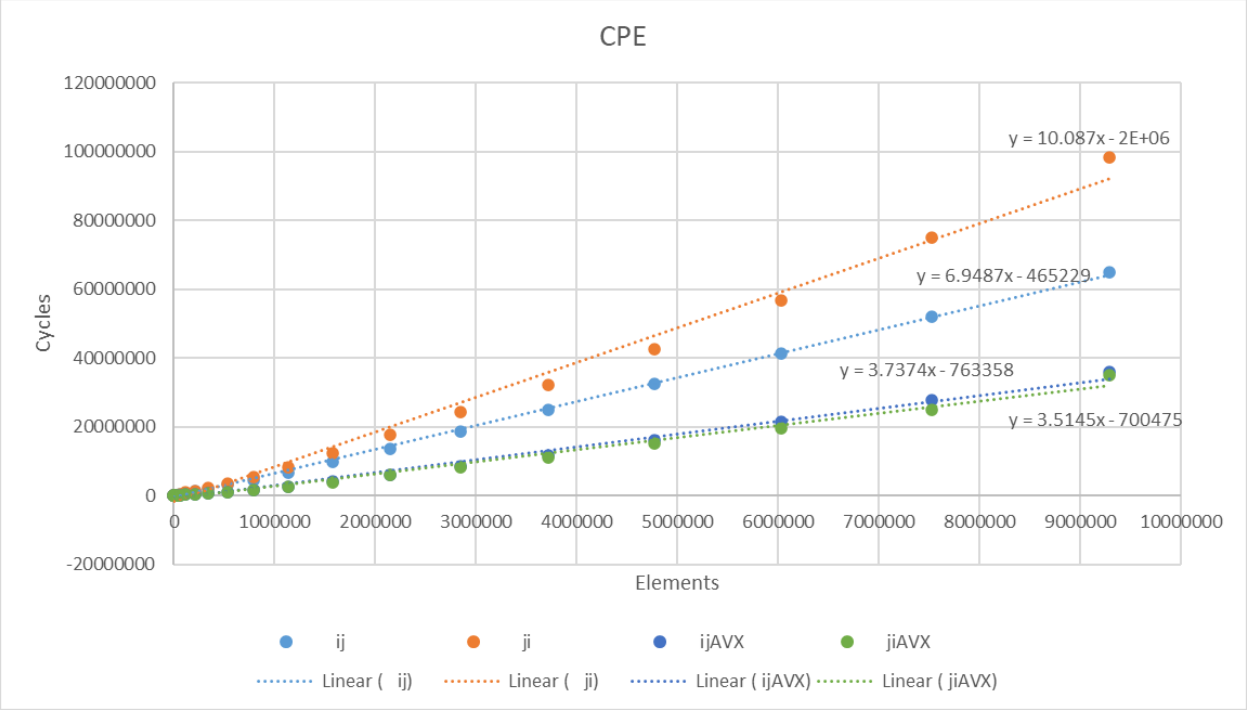
O3:



O2:



O1:



As opposed to the SSE when dealing with floating points, in the AVX, O2 and O3 optimizations make a huge difference. Although the AVX version is still doing better, their differences are much smaller.