Lab 3 — Programming Assignment 3

Part 1. SSE extensions using C structs and union

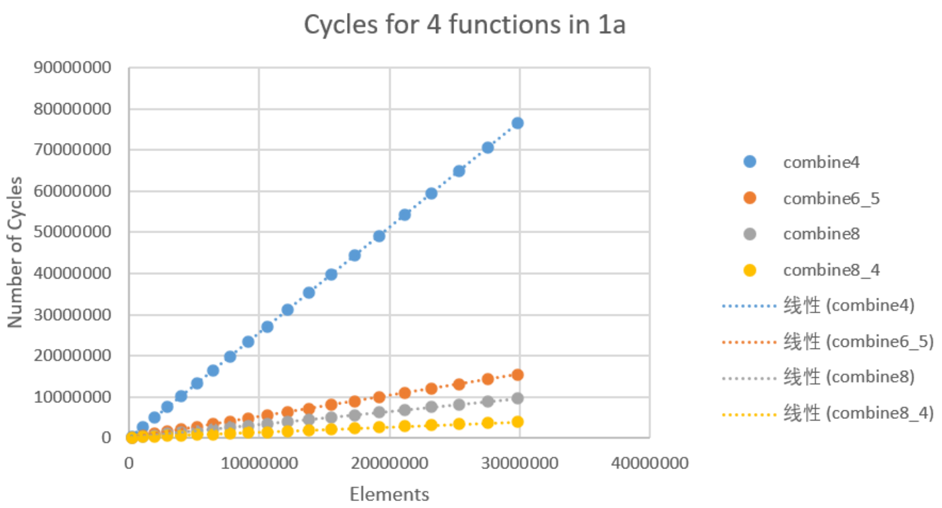
1a:

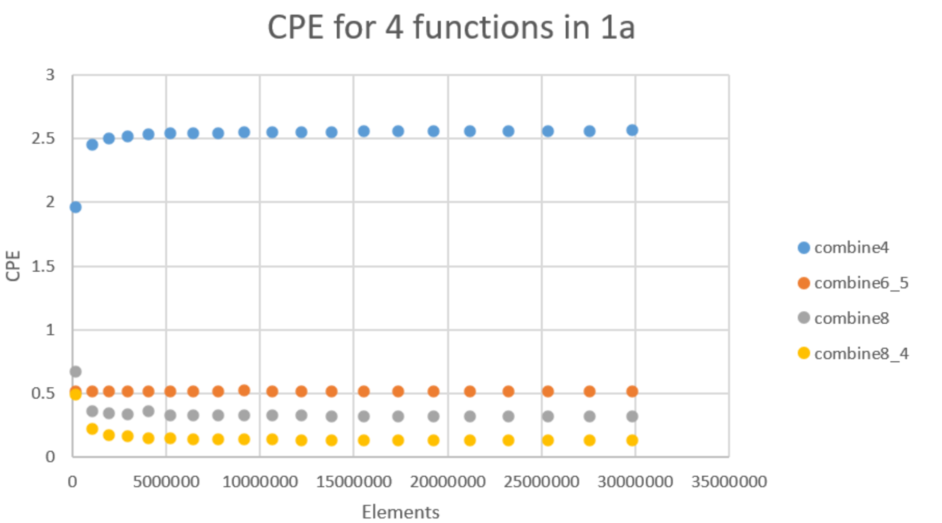
As the problem demands that is should do the addition, we set the OP to be “+” and IDENT to be 0. Also, the VSIZE can be calculated as (VBYTES/sizeof(float)), so that it should be 32/4=8. In order to meet the demands that Ax^2+Bx+C is always a multiple of VSIZE, we can set A, B and C to be 40, 800 and 200 so that whatever x is it always be a multiple of 8. And due to the final calculation of 40*400+800*20+200=29840 is larger than 10000, so it meets the standard.

And here is the result: we can see from the image given below that the combine8_4 performs the best since it not only uses the vector but also multiple accumulators.

And the CPE for the four functions is:

Method	Combine4	Combine6_5	Combine8	Combine8_4
CPE	2.5652	0.5152	0.3197	0.1275

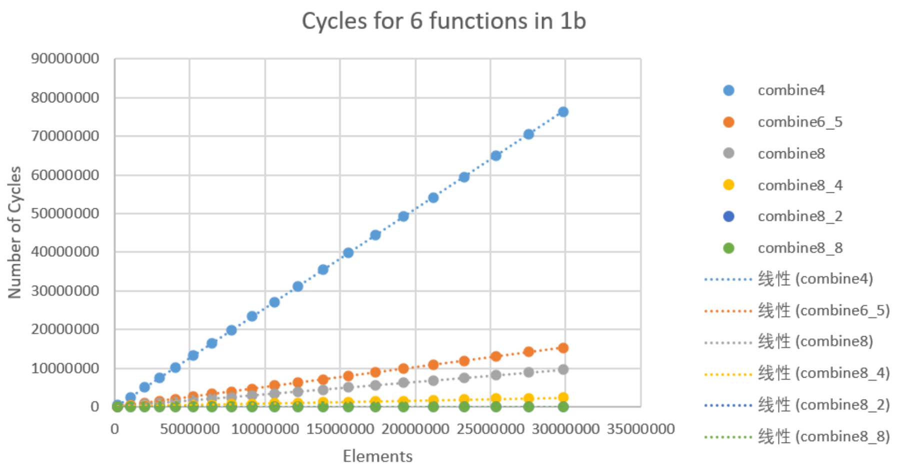


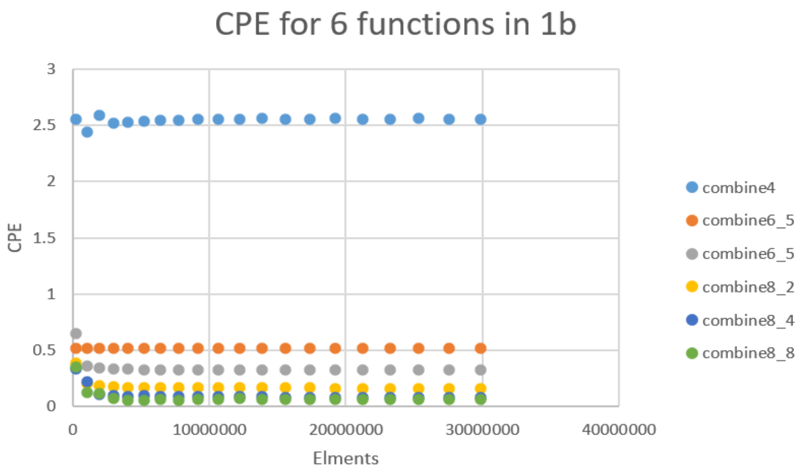


1b:

After modifying the codes to add the combine8_2 and combine8_8 function, we can see from the result that the more accumulators, the better the performance, and they are likely to multiply by 2. The reason behind that is that the greater parallelization is achieved through more elements being engaged in each iteration for the vectorized version.

Method	Combine4	Combine6_5	Combine8	Combine8_2	Combine8_4	Combine8_8
CPE	2.5611	0.5149	0.3294	0.1608	0.0799	0.0591





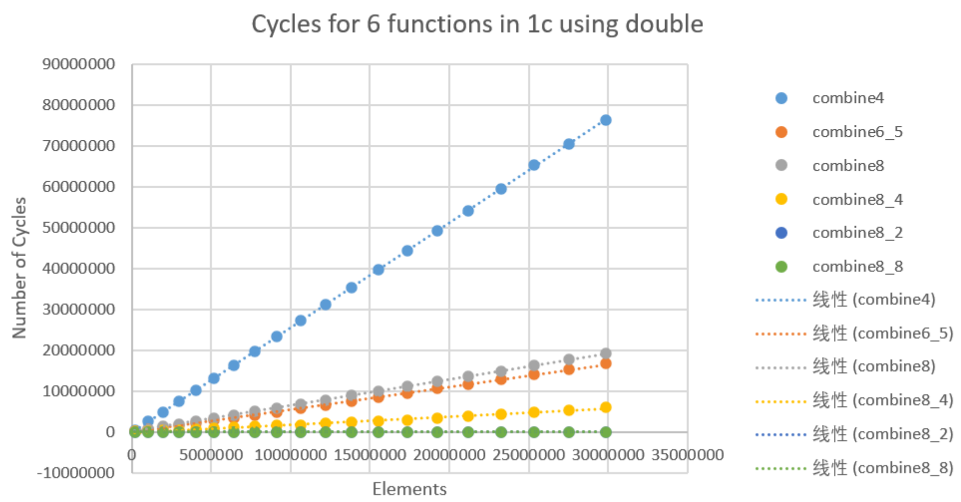
1c:

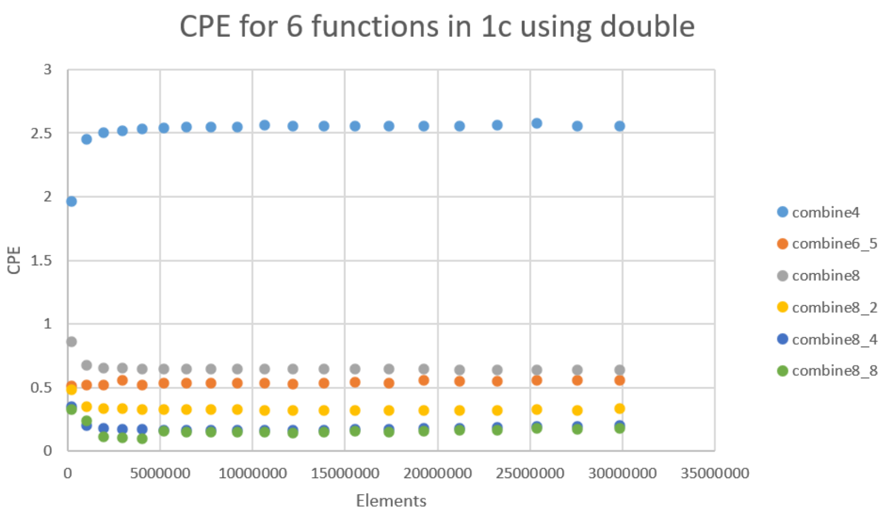
When changing the data type from float to double, we can see that the rank between the all 6 functions does not change much, but the overall CPE does increase for some bit. But for the Combine8_8, its performance does not increase much compared to the Combine8_4.

The reason behind the difference is probably that for the float, the VSIZE is calculated as 8, so that there are 8 processes being calculated as the same time; but for the double, each one has 8 bytes, so that there are only 4 processes at the same time. So compared to the CPE in float, it is slowed down.

Method	Combine4	Combine6_5	Combine8	Combine8_2	Combine8_4	Combine8_8

CPE	2.5665	0.5578	0.5403	0.3251	0.1927	0.1771

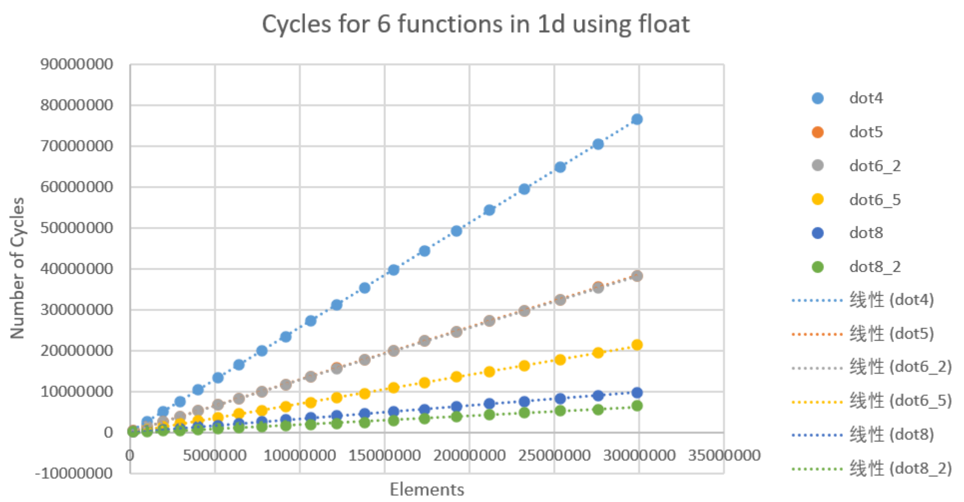


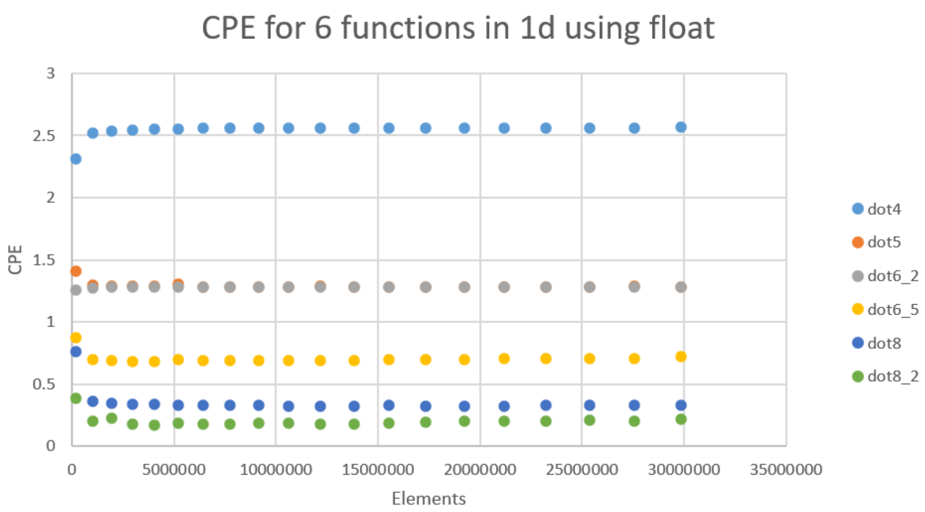


1d:

We can see from the result showing that the performance of dot5 and dot6_2 are almost the same, and the performance between dot8 and dot8_2 are almost 2 times. This is perhaps because we are traversing 4 elements in one cycle and operating on all of them instead of operating on individual elements in scalar dot product. As always, the dot4 which uses neither vector nor multiple accumulators are the worst one.

Method	Dot4	Dot5	Dot6_2	Dot6_5	Dot8	Dot8_2
CPE	2.5646	1.2835	1.2821	0.7090	0.3243	0.2101





1e:

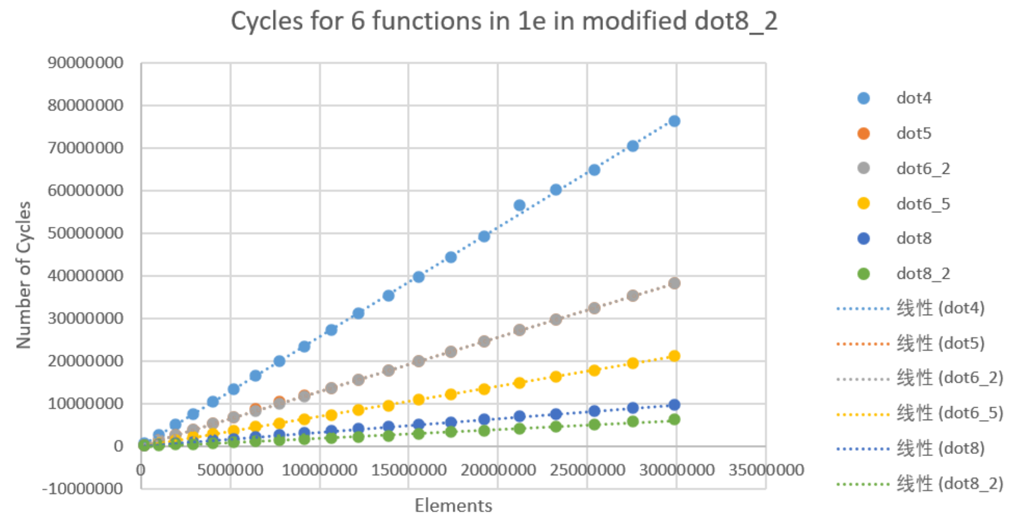
For the modified dot8_2 function, we can see the new performance from below.

And the error in the codes is at

```
while (cnt) {
    result += *data0++ * *data1++;
    cnt--;
  }
```

That it uses the wrong while (cnt >=0).

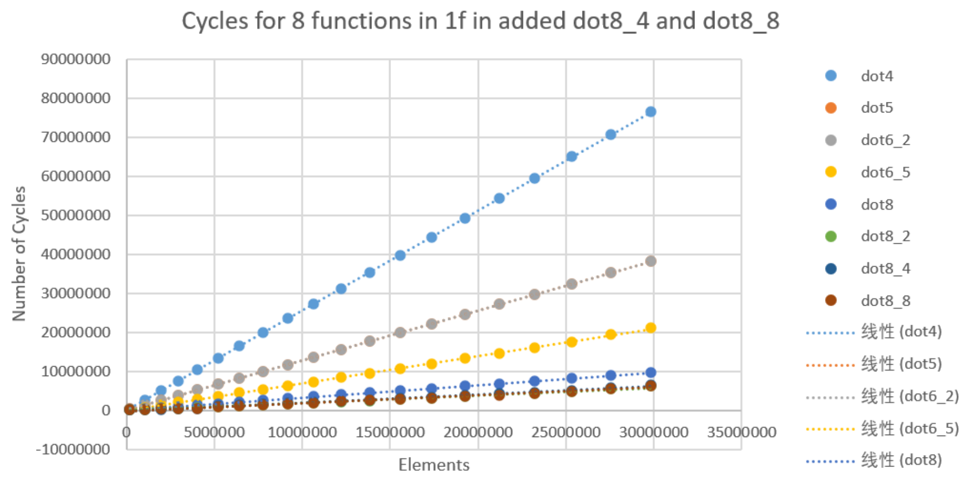
Method	Dot4	Dot5	Dot6_2	Dot6_5	Dot8	Dot8_2
CPE	2.5781	1.2786	1.2815	0.7059	0.3229	0.2051

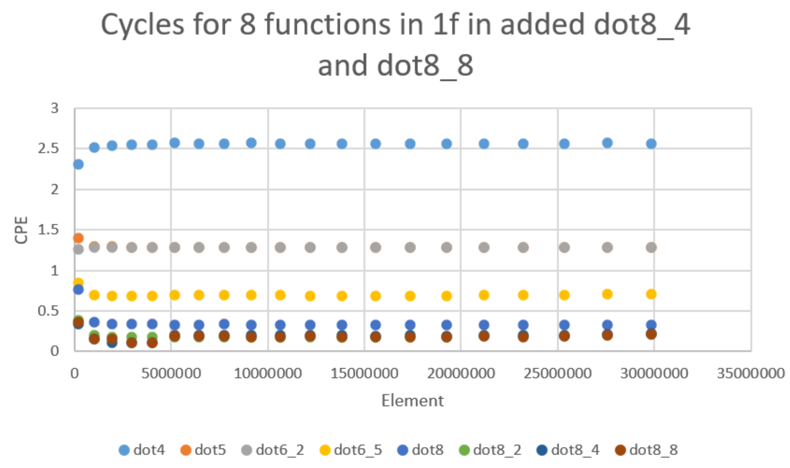


1f:

After adding the dot8_4 and dot8_8 function, we can see that the performance between them and dot8 and dot8_2 makes no big difference. This is probably because that the CPU has only five pipeline stages, so no advantage of increasing the unroll factor to a larger number.

Method	Dot4	Dot5	Dot6_2	Dot6_5	Dot8	Dot8_2	Dot8_4	Dot8_8
CPE	2.5665	1.2812	1.2816	0.7001	0.3205	0.1972	0.2129	0.1998





Part 2 -- SSE extensions using intrinsics

2a:

According to the compile result, the function unalign_heap_naive(a) doesn't work. The difference beween unalign_heap_naive(a) and align_heap_1(a) is that for the definition p1 and p2, they are not set to be aligned; and for the function unalign_storeu_ps(a), it only defines p0 to be type of __m256*.

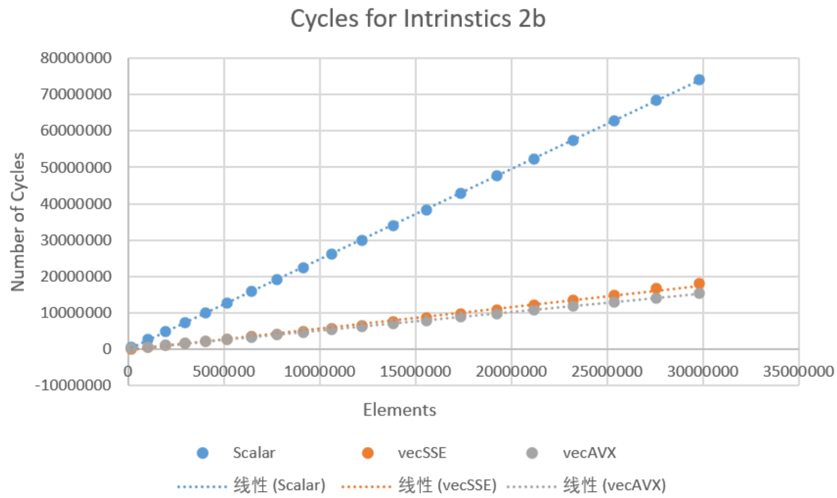
```
cloris@vlsi27$ ./avx_align
AVX load/store alignment tests
unalign_local_alloc:
  p1 == 0x7ffdc7f29354   p2 == 0x7ffdc7f29358
      2      3      4      5      6      7      8      9     10     11
      2      2      3      4      5      6      7      8      9     11
      2      2  1.414  1.732      2  2.236  2.449  2.646  2.828      3

align_heap_1:
  p1 == 0x852040   p2 == 0x852060
      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 == 0x852024   p2 == 0x852028
      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
```

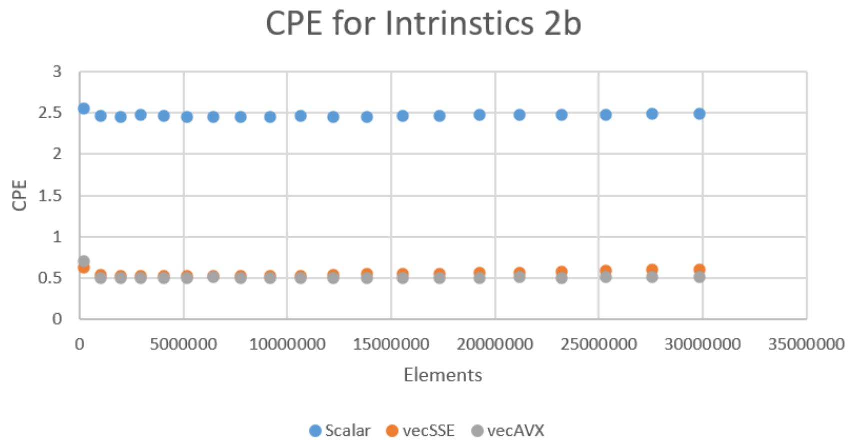
And the reason behind that is probably that _mm256_load_ps requires 256-bit (32-bytes) aligned memory, and the function unalign_heap_naive(a) uses the default allocator for std::vector doesn't meet that requirement. So it needs to turn to some instruction with less stringent alignment requirement such as _mm256_storeu_ps() in the unalign_storeu_ps().

2b:

We can see from the shown graph that the performance of the normal vector is the worst, while the vecAVX is a little bit better than the vecSSE.

Method	Scalar	vecSSE	vecAVX
CPE	2.4821	0.5978	0.5113

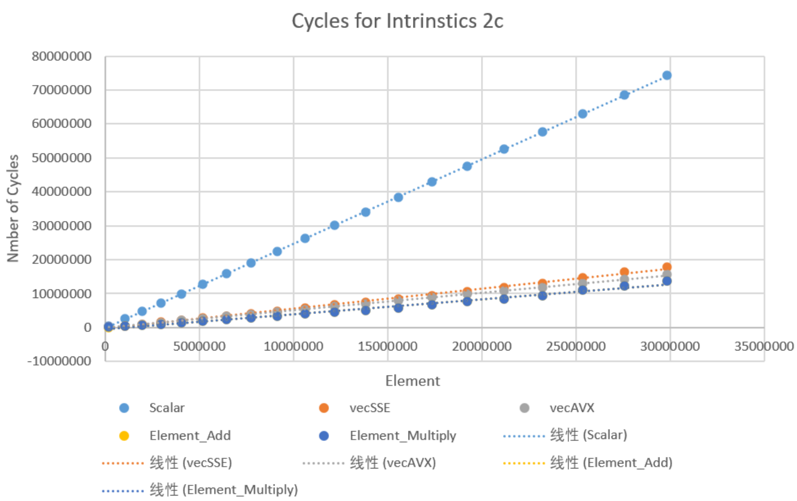


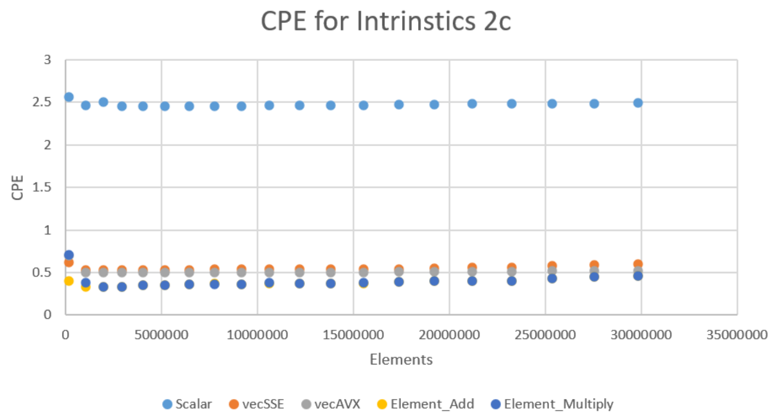


2c:

We can see from the given result that the regular result of Element_Add and Element_Multiply performs better than the scalar but not as good as vecSSE and vecAVX since they do not use the vector to do the calculation.

Method	Scalar	vecSSE	vecAVX	Element_Add	Element_Multi
CPE	2.4866	0.5869	0.5177	0.4450	0.4447



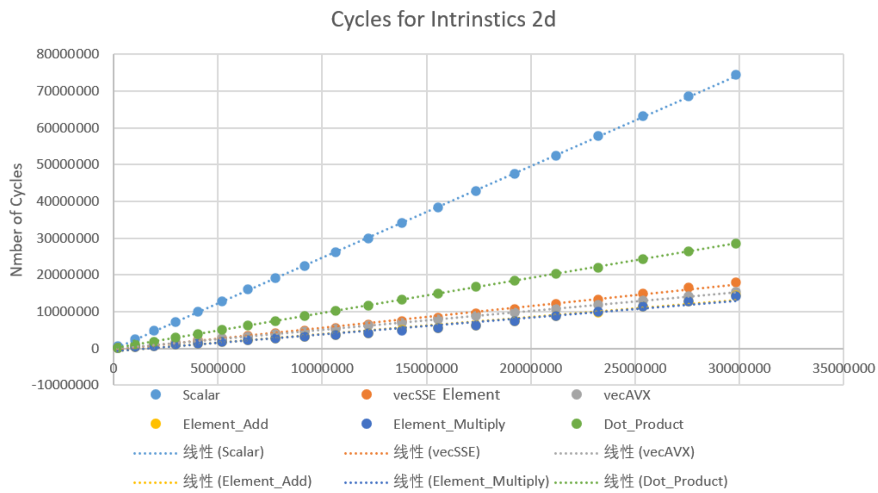


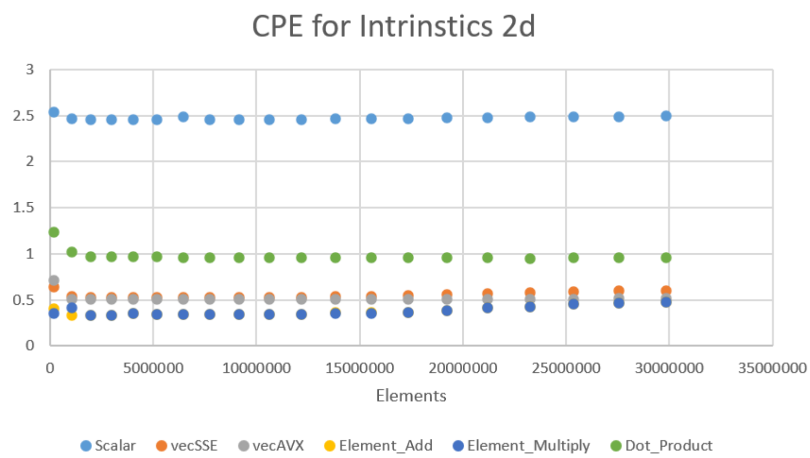
2d:

After implement the dot product function, we can see the total result that:

This is the CPE result using vectorized dot product.

Method	Scalar	vecSSE	vecAVX	Element_Add	Element_Multi	Dot Product
CPE	2.4821	0.5943	0.5125	0.4619	0.4612	0.9541

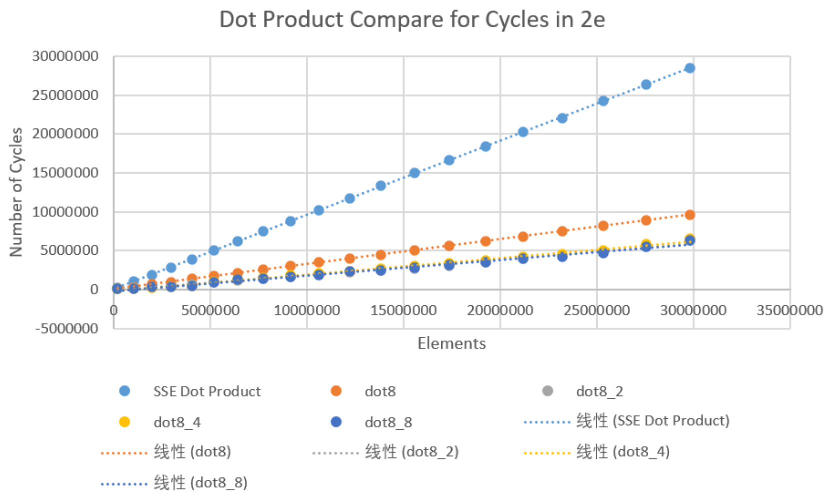


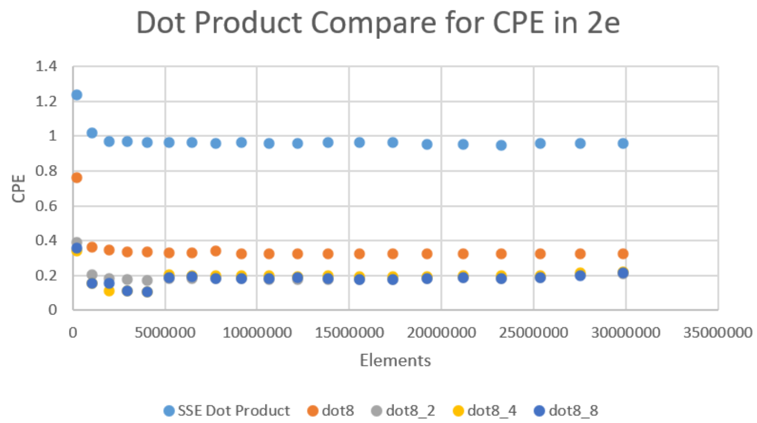


2e:

And this is the CPE using __attribute__ ((vector_size(VBYTES))).

Method	Dot8	Dot8_2	Dot8_4	Dot8_8	Dot Product
CPE	0.3205	0.1972	0.2129	0.1998	0.9541





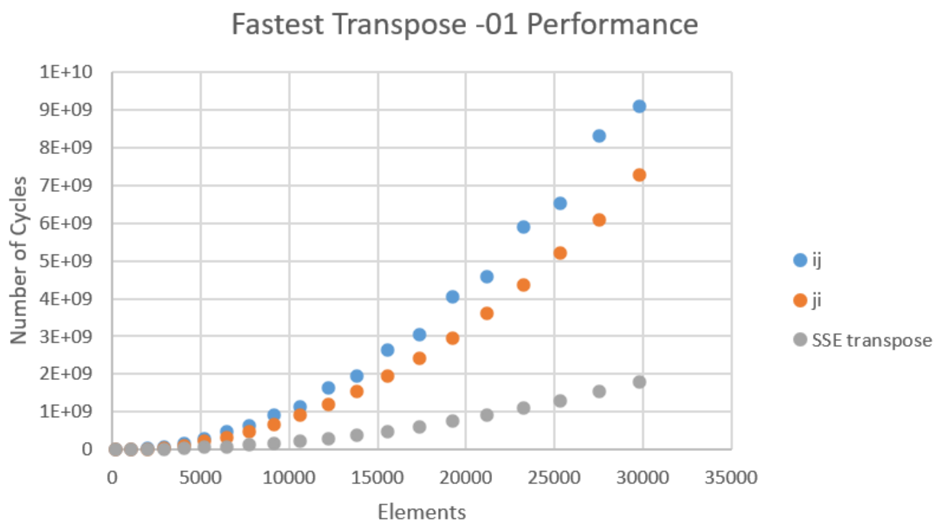
We can see that when using the intrinsic functions, it does not perform as good as the one using the vector unrolling, the Dot8 function has a CPE of 0.3, while using the intrinsic functions it is almost 3 times.

And for the programmability, for the vector unrolling, one needs to do several preparations such as go into steps to do memory alignment and deal with the remaining elements, but for the intrinsic functions, it is much more easier to write codes. Also, if one wants to accelerate the process, he can simply add more accumulators in the vectorize functions to speed up, but for the intrinsic function it cannot do much improvements.

Part 3 -- A simple SSE application from scratch: Transpose

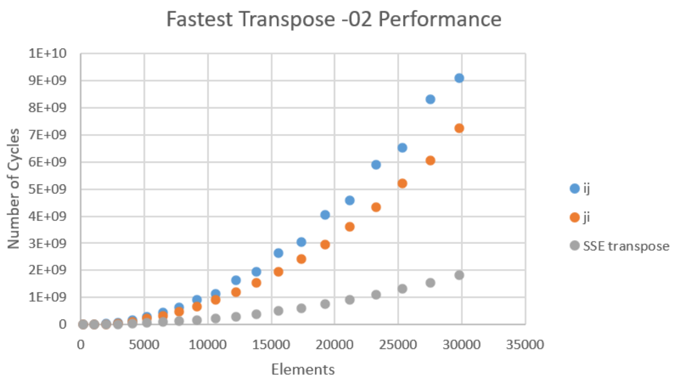
3a:

After implementing the function using SSE transpose intrinsic _MM_TRANSPOSE4_PS, we can see from the result graph that it does perform better than the original transpose ones.



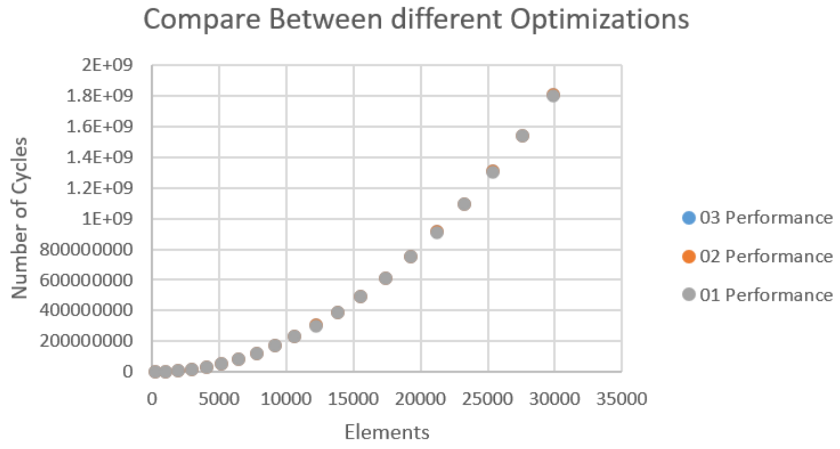
3b:

After testing with different kinds of optimizations, the result does not change much.



To make it easier to see, I make a graph combining the results of all the three optimization ways, and it shows that there is almost no difference.

According to the optimization method given in the compiler, if there are something in the codes that can be improved, trying different optimization types can have a change in the performance. But in this method, as I used the SSE transpose intrinsic method _MM_TRANSPOSE4_PS, there is not much to be optimized.



Extra Credits:
This is the result shown for the extra credit. According to the guide online, I used function _mm256_shuffle_pd and _mm256_permute2f128_pd to implement the transpose for the _mm256 data type, and it takes 2 times of the one using _MM_TRANSPOSE4_PS.
There are 2 reasons behind it. The first is that for the intrinsic function, it only uses one function; but for the transpose I implemented on my own, it uses two functions; also, the data type for the _mm128 and _mm256 has different length, so that it is different.