Lab 2 — Programming Assignment

Annotations: I have finished the lab2 before seeing the lab0 feedback so that I did not delete the lines from the plots. I will modify them in the next lab.

Part 1. Experiment with basic optimization methods as presented in B&O 5.4-5.10

1a:

Before running the codes, first we need to calculate the maximum of elements being used in the programs to make sure that (Ax2+Bx+C) × sizeof (data_t) is less than 32K bytes. And the 32K in the CPU is equals to 32768 bytes.

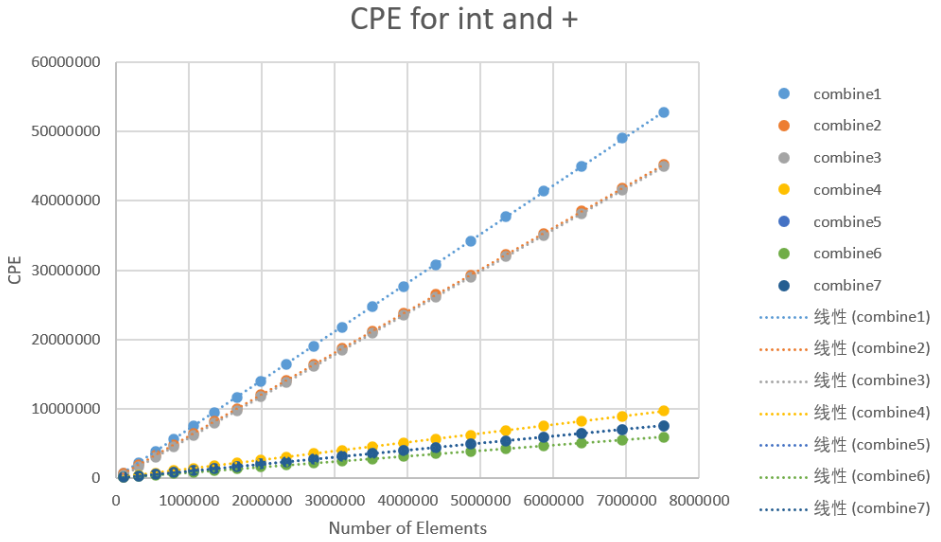
So, for each type of data, we can see that sizeof (int)=4, sizeof (long int)=8, sizeof (float)=4, sizeof (double)=8, sizeof (long double)=16.

So, for the largest size fit in the level1 cache, to make it easier to compute, I set the largest data type to be **double** and set A, B and C to be 5, 100 and 50, and the total number of tests to be 20, so that the Ax^2+Bx+C=3755 less than 4096 for the double.

As I tried three data types for int, float and double, and each tried operation for add and multiple. Finally, I randomly choose method of combine1 and fix all them together. Here is the images shown below.

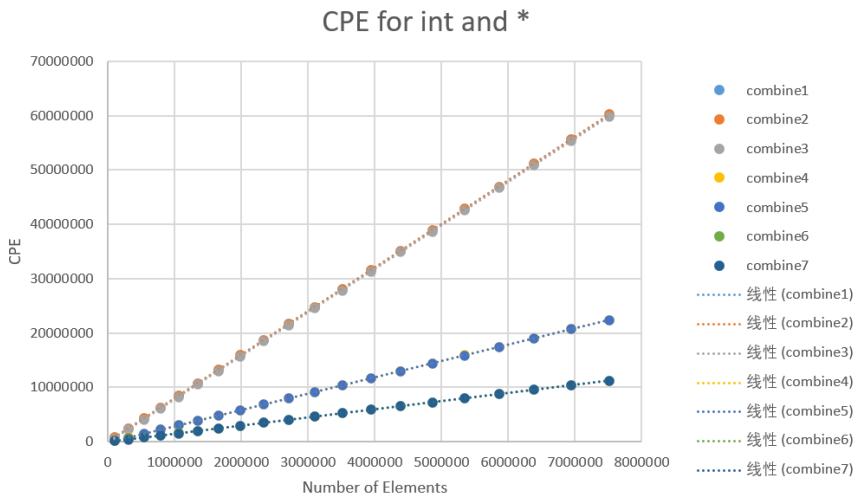
For the int and +

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	7.0396	6.0102	6.0105	1.274	1.0023	0.783	1.0022



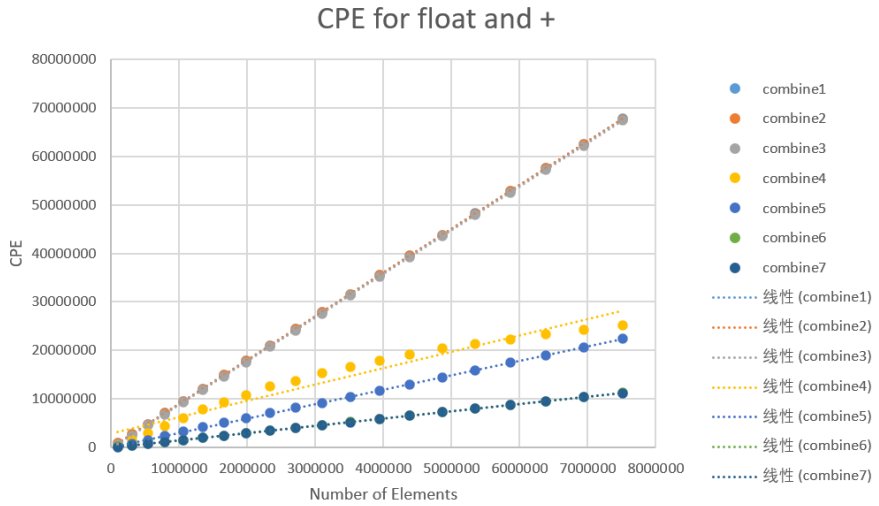
For the int and *

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	8.01545	8.014962	7.971711	2.980487	2.978845	1.4995	1.49659



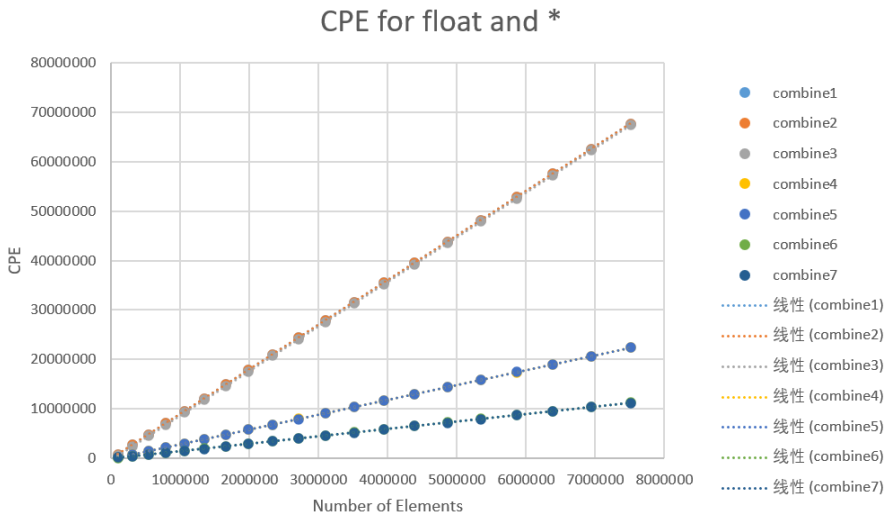
For the float and +

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	9.016785	9.013944	8.96973	3.34989	2.98134	1.500553	1.492301



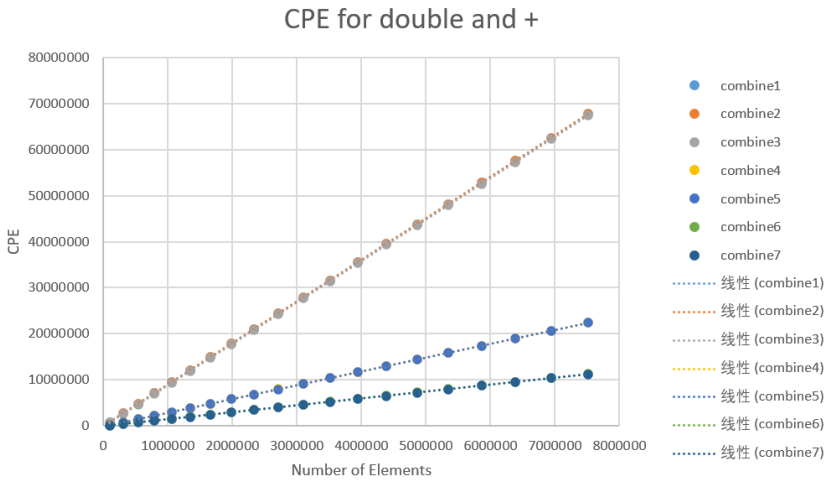
For the float and *

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	9.013056	9.013056	8.969019	2.982914	2.981665	1.498677	1.489383



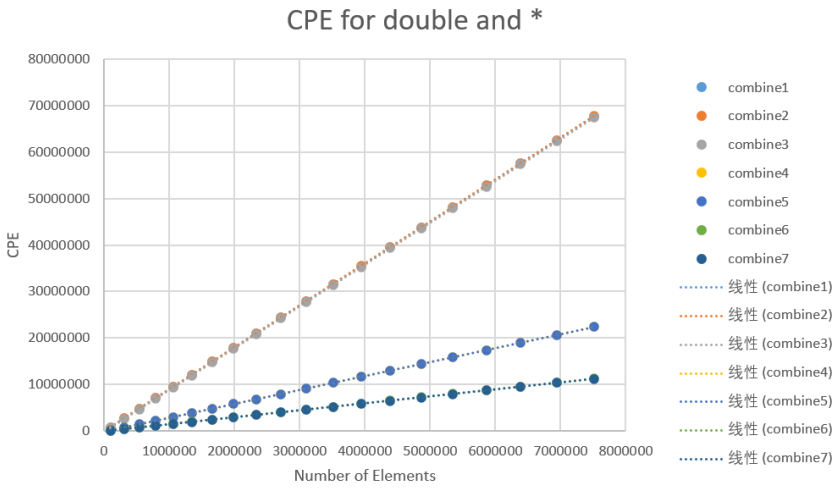
For the double and +

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	9.015462	9.015222	8.971676	2.982647	2.979907	1.500135	1.489353

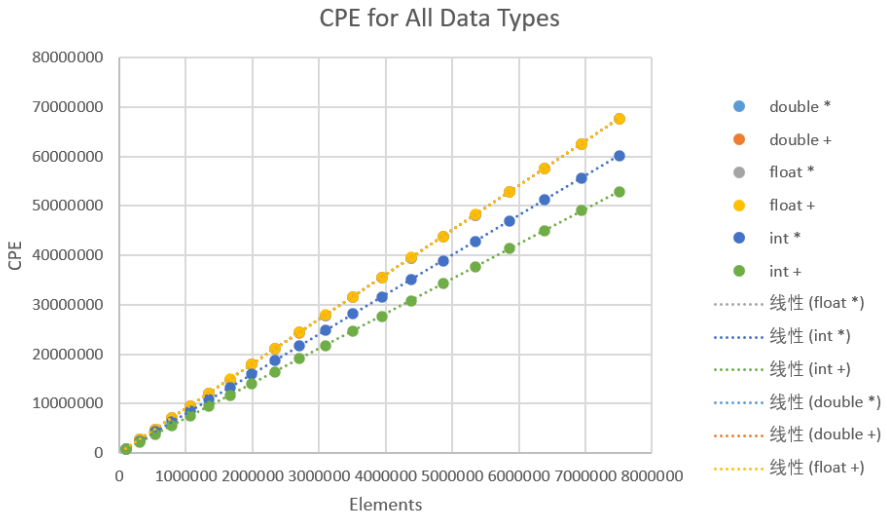


For the double and *:

Method	Combine1	Combine2	Combine3	Combine4	Combine5	Combine6	Combine7
CPE	9.015462	9.015222	8.971676	2.982647	2.979907	1.500135	1.489353



For all:



As we can see, the results obtained are almost in agreement with the results given in the book. The CPE is observed to go down with every optimization from combine1 to combine7.

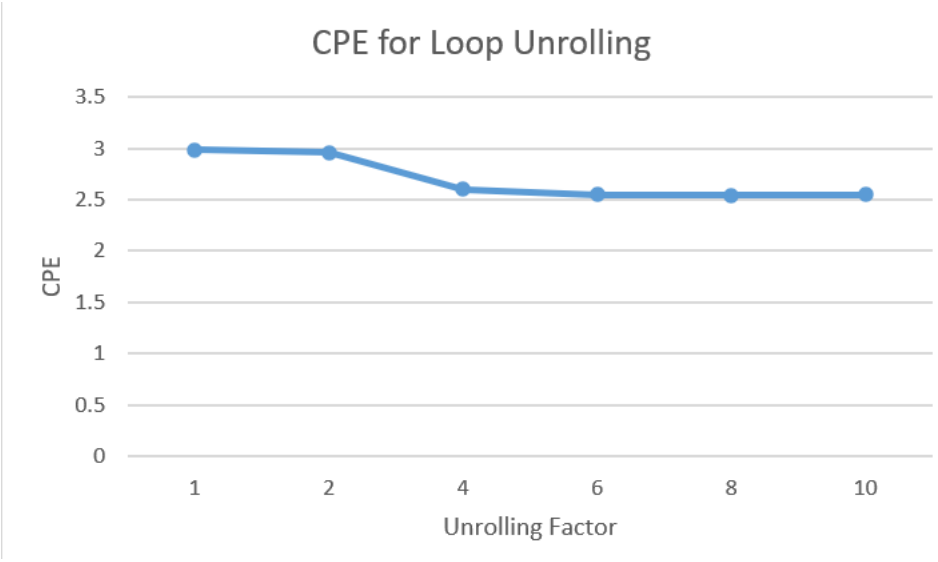
For the extreme situations, I also tried the data type such as “long double” which has 2 times the space of the double. And as the result shows, after the number of elements getting larger than 3000000, it increases greatly.

The CPE doesn't seem to vary a lot with respect to data types since the vector size is very small. If run for a larger data size, float and double take significantly more time than int. Also, multiplication of larger vectors is closed to the addition. I cannot see much difference between them.

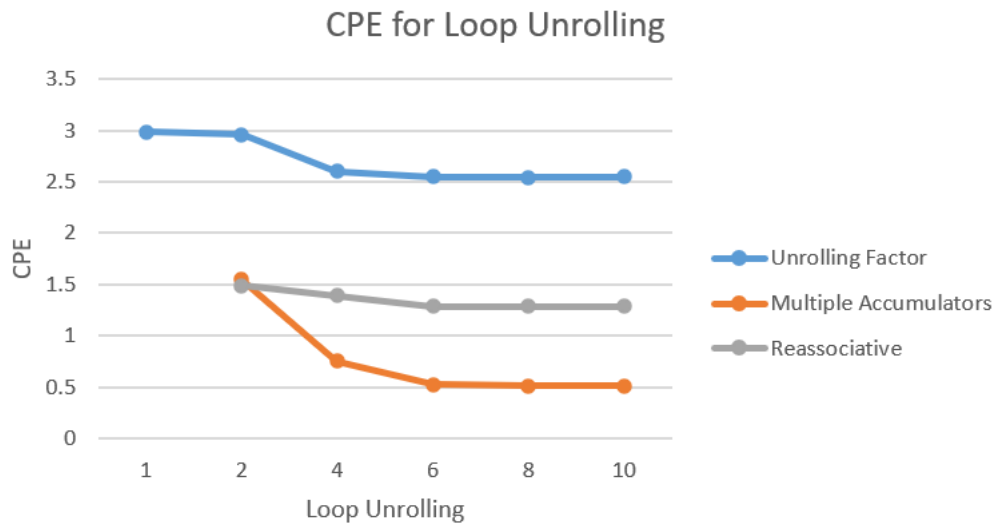
1b:

As shown in the image, the performance using different unrolling factor does not matter. One possible explanation for this could be the increase in number of operations carried out in one single iteration. Since multiply operation takes a significant number of cycles, increasing the unrolling factor beyond 4 would mean to perform more than 4

multiplications in one iteration.
This will also increase the number of registers required to store the intermediate results. All these reasons lead to the performance becoming stagnant beyond a certain unrolling factor. However, for small unrolling factor, the performance does show a significant improvement and hence proves the benefit of loop unrolling.



1c:
In the images showing below, we can see that the multiple accumulators performance is better than the reassociative transformation, but all these two performs much better than the unrolling loop.
However, for unrolling factors greater than 4, the number of operations per iteration increases and hence we observe an increase in CPE for associativity. Parallelization on the other hand does not take a major hit in performance but is stagnant which is because of the two accumulators to store the result.



Part 2. Apply basic methods to dot product

There are 2 optimization methods:

1. Unrolling Dot Product:
   This method increases the performance by reducing the number of iterations compared to the normal dot product method. It increases the number of computing operations of elements each iteration, also, it finally adds the last element at the end of the whole vector.
   So, it reduces the number of operations like loop indexing and conditional branch which do not directly contribute to the program result.

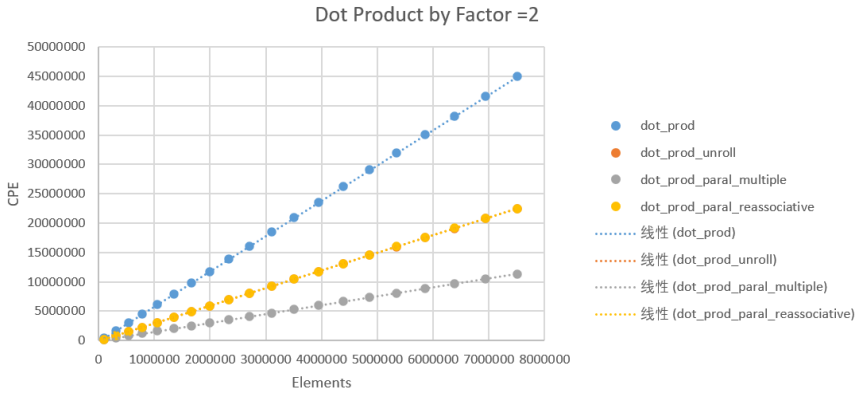
2. Parallelized Dot Product for multiple accumulators:
   This method separates the operations so that it increases the performance. In the codes, we combine the whole set of operations into two parts and combined the results in the end. So, at a particular index value of the loop dot product of two set of values are computed and stored in acc0 and acc1. Then another loop is used to compute results of remaining elements. Finally, both acc0 and acc1 are combined.

3. Parallelized Dot Product using reassociative transformation:
   The function for reassociative transformation is similar to the function of using unrolling factors, and the difference between them is that the unrolling factor add one dot result at each time and finally add them up but the reassociative function add two at each time. In the result from the 1.c we can see that the reassociative is nearly half the CPE of the unrolling factor, but for problem2 they are almost the same.

The results using different loop indexing are showing below.

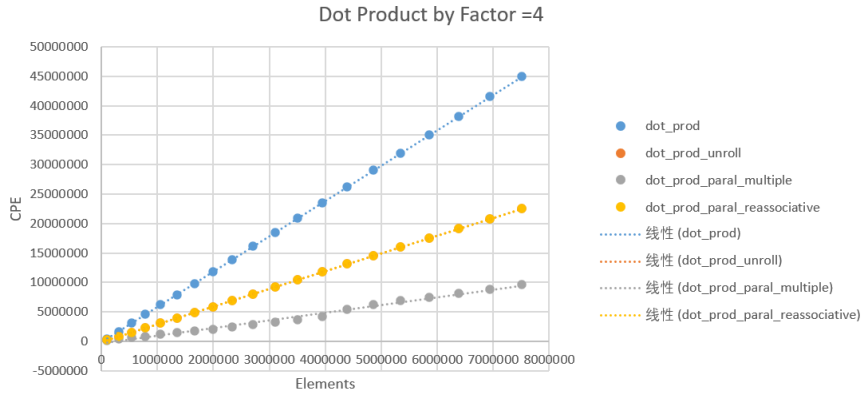
For factor=2:

Method	Dot_prod	Dot_unroll	Dot_paral_multiple	Dot_paral_reassociative
CPE	5.990714	2.992921	1.508513981	2.994884154



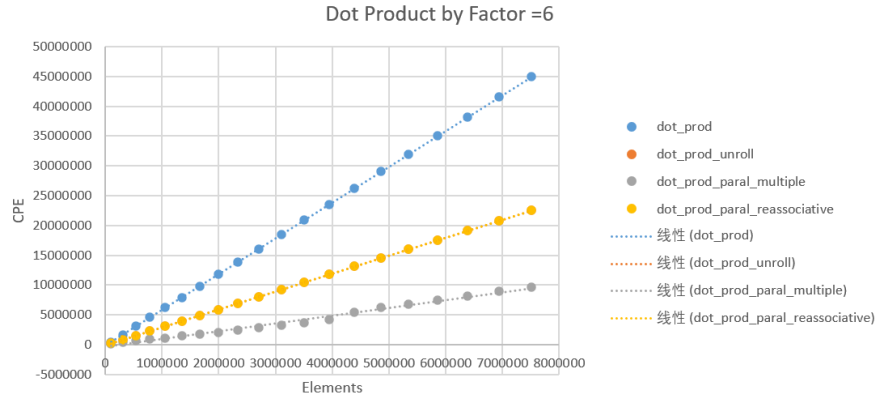
For factor=4:

Method	Dot_prod	Dot_unroll	Dot_paral_multiple	Dot_paral_reassociative
CPE	5.987337	2.997573	1.276194407	2.995416644



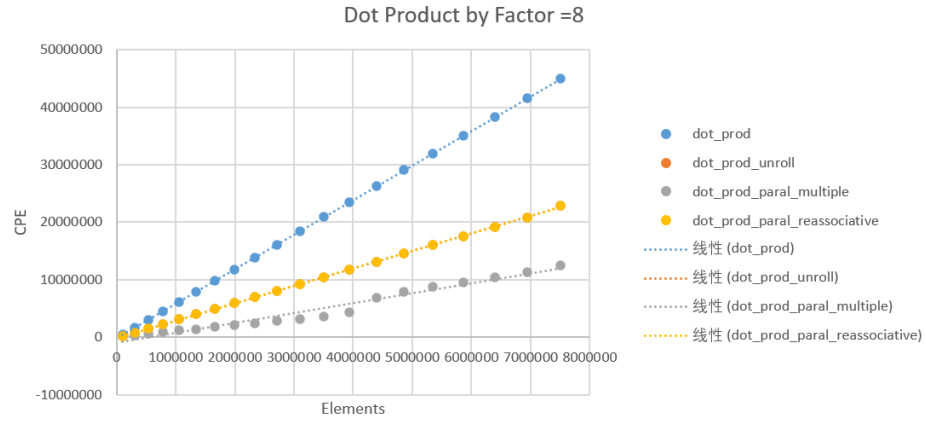
For factor=6:

Method	Dot_prod	Dot_unroll	Dot_paral_multiple	Dot_paral_reassociative
CPE	5.987773	2.995034	1.287722636	2.994247936



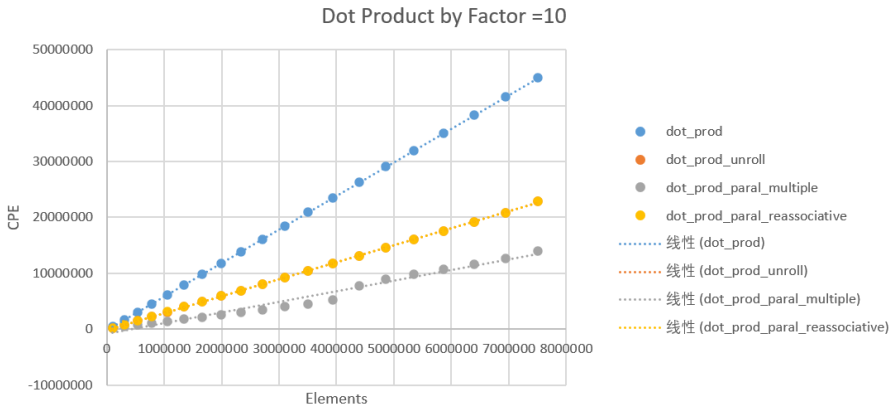
For factor=8:

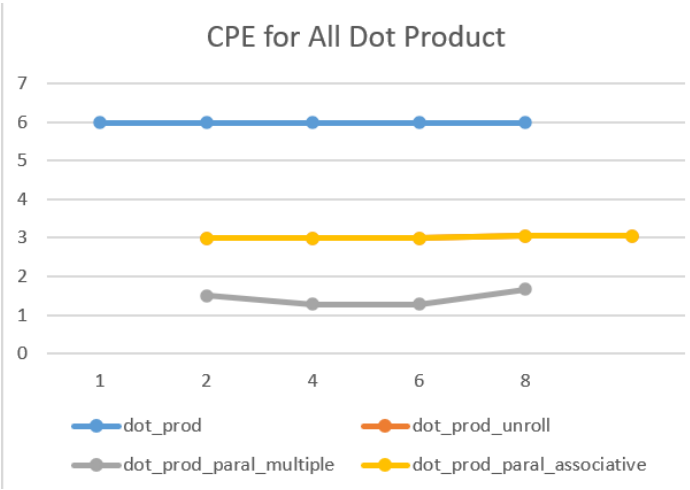
Method	Dot_prod	Dot_unroll	Dot_paral_multiple	Dot_paral_reassociative
CPE	5.987015	3.052275	1.666898003	3.052513316



For factor=10:

Method	Dot_prod	Dot_unroll	Dot_paral_multiple	Dot_paral_reassociative
CPE	5.987015	3.052275	1.666898003	3.052513316



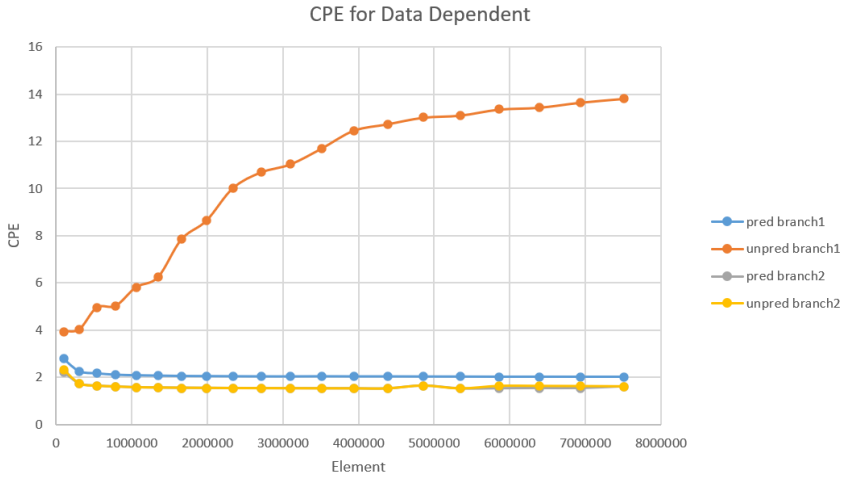


Overall, we can see that the two optimized methods all decreases the CPE of the dot product compared to the simplest way of dot product. And to be briefly, the parallelization function using reassociative transformation preforms the same as unrolling, and for all the factor that have been tested, the dot product using 4 multiple accumulators are the best.

Part 3 Force and evaluate conditional moves

3a:

The result is shown below:



For the initiation of the array, I set the predictable to be a constant, and for the unpredictable, I set them to be a random number.

We can see from the image that for the unpredictable for the branch1, as it needs more time to calculate, it is much more slowly than the predictable 2.

Also, if both using the predictable numbers, the branch2 preforms better than the branch1 since it does not need to use the branch.

3b:

As shown in the webpage choosing the “x86-64 gcc 4.8.5” compiler and “-O1” optimization, we can see that the first function uses a branch instruction (“jbe”, “highlighted”) and the second function uses the "maxsd", and the different data type such as “float” and “double” successfully confuse the compiler's optimizer so that it generated different codes.

Also, the reason that both the pred and the unpred initiation looks nearly the same is because that the modern compiler is smart enough to convert the branch for the comparation into the (A>B)?A:B, and that’s why they are making no difference. And in the case we choose the old-time compiler, they make a difference.

```
max_if(double, double):
        ucomisd xmm0, xmm1
        jbe     .L6
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
.L6:
        unpcklpd        xmm1, xmm1
        cvtpd2ps        xmm0, xmm1
        ret
max_ce(double, double):
        maxsd   xmm0, xmm1
```

```
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
```

On the other hand, if we change the data type all into double, then the compiler does not need to transform one into another, then the codes will become much more easier.

```
max_if(double, double):
        maxsd   xmm0, xmm1
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
max_ce(double, double):
        maxsd   xmm0, xmm1
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
```

Part 6: Quality Control

6a:

About 12 hours.

6b:

No, only making the graph in excel takes much time.

6c:

Not for this one.

6d:

Not for this one. The TA really helps.