

# Vector\_DotProduct

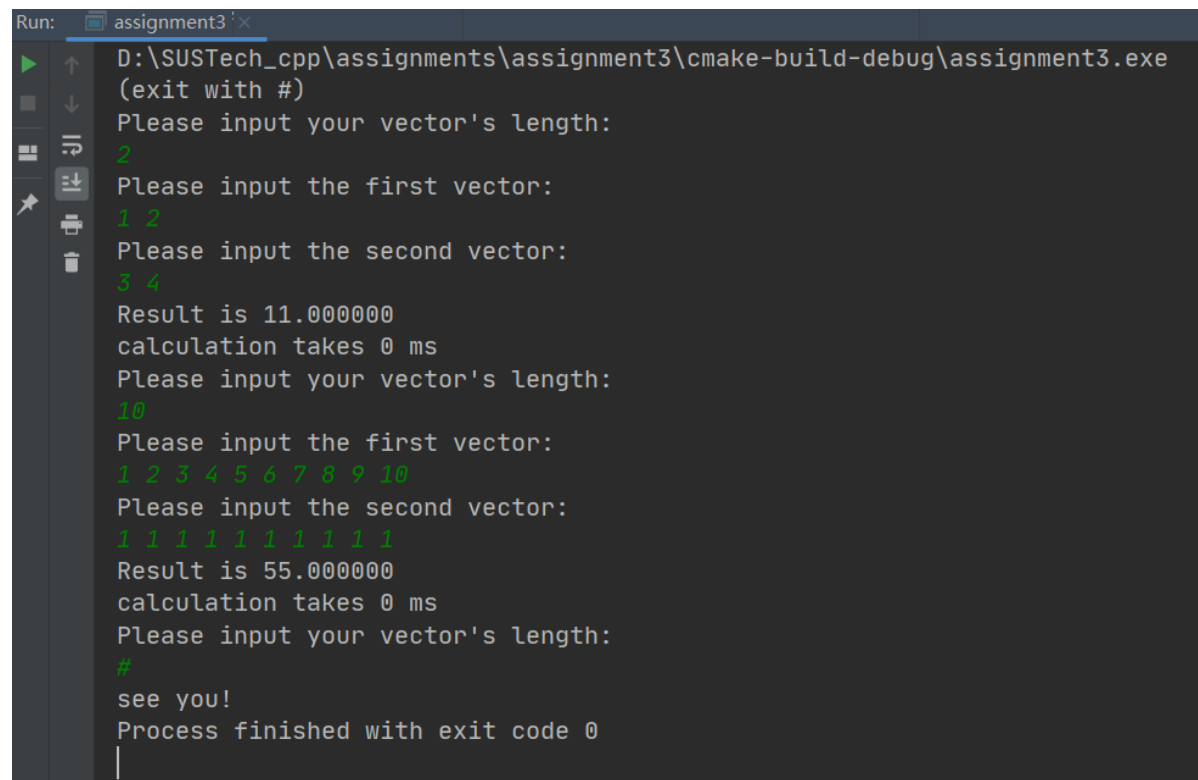
*Aimed to optimize the process of vector dot product calculation*

**Each test of time cost is under the same pair of vectors with length 200 million**

## Some Basic features

### One execution Multiple calculation!

In this program, you can try multiple dot product of different vectors just in one execution. If you want to exit, just type #.



```
Run: assignment3
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
2
Please input the first vector:
1 2
Please input the second vector:
3 4
Result is 11.000000
calculation takes 0 ms
Please input your vector's length:
10
Please input the first vector:
1 2 3 4 5 6 7 8 9 10
Please input the second vector:
1 1 1 1 1 1 1 1 1 1
Result is 55.000000
calculation takes 0 ms
Please input your vector's length:
#
see you!
Process finished with exit code 0
```

### Won't crash down when you input something wrong!

This program will detect whether your input is valid, if not, you are asked to input again.

```
Run: assignment3 | x
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
30h
wrong input, please input again!
Please input your vector's length:
2@
wrong input, please input again!
Please input your vector's length:
2
Please input the first vector:
11 3
wrong input, please input again!
Please input your vector's length:
2
Please input the first vector:
1 3
Please input the second vector:
2 4*
wrong input, please input again!
Please input your vector's length:
2
Please input the first vector:
1 3
Please input the second vector:
2 4
Result is 14.000000
calculation takes 0 ms
Please input your vector's length:
#
see you!
Process finished with exit code 0
```

## Version\_1.0

### Brutal Force

Simply use the definition of dot product. It's very easy to implement but definitely has low efficiency in time.

```
1 // begin time
2 auto start1 = std::chrono::steady_clock::now();
3 for (int i = 0; i < n; ++i) {
4     result += f1[i] * f2[i];
5 }
6 // end time
7 auto end1 = std::chrono::steady_clock::now();
```

Here we use float as the data type since float has a better behavior in efficiency. Also we use dynamic array to store what we input for its high efficiency when doing indexing, invoking and calculating.

However, to make things more clear and convincing, we will also see what will happen if we use double and vector separately, compare their time cost and give a reasonable explanation.

## Time Cost

test this brutal force algorithm with two vectors each has 200 million elements and output its' calculation time.

```
Run: assignment3 x
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -365884255676600942592.000000
calculation takes 1165 ms
Process finished with exit code 0
```

We can see that the calculation of dot product takes 1165ms, which has a lot of space to do our optimization.

Also, let us see how do double and vector behave!

### double

```
Run: assignment3 x
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -364535047238995935232.000000
calculation takes 1206 ms
Process finished with exit code 0
```

As expected, double is lower in efficiency than float since double occupies 8 bytes while float just 4 bytes. This will bring about more waste on time when doing calculation for operating 64 bits other than 32 bits one time.

### vector

```
Run: assignment3 x
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -365884255676600942592.000000
calculation takes 2042 ms
Process finished with exit code 0
```

Additionally, the time cost of vector is also within our expectation. It's even lower than double.

Here we need to clarify the principle when allocating memory to vector:

when initiating a vector, its default size = capacity = 0. That's to say, CPU will not initiaively allocate memory for a vector, thus when using `vector.pushback` but there is not enough space for vector to store a new 4-byte float(capacity = size), this time vector will reapply a new memory space which is twice as before. Then it will copy the elements in the original space to the newly-applied memory, put the element you pushback in the tail of the new vector and then set the original memory space free. The whole process is very low in efficiency.

## Version\_2.0

## Block Vector Calculation

Use divide and conquer, partition the vector with many small block vectors with length 6, and calculate the result of dot product of the two corresponding block vectors. And then we combine each of the dot product. This way we can decrease the loop counts and arithmetic times.

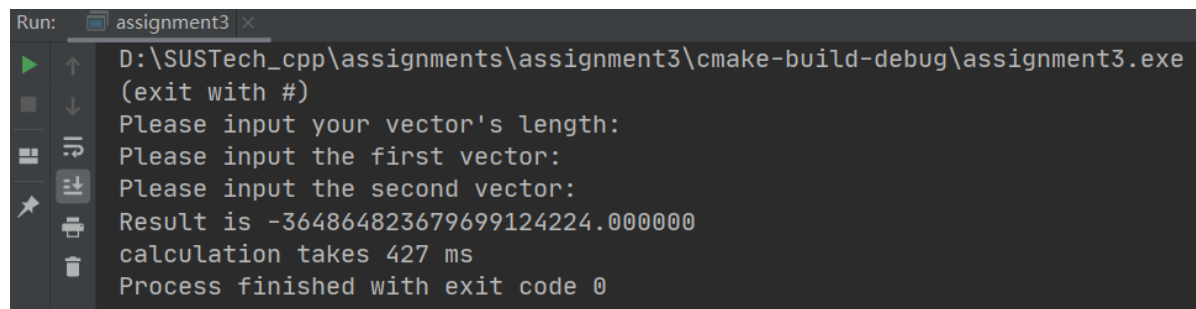
Here we can divide it into small unit vectors with length 2, 4, 6 and 8, depending on the data size.

```
1 float block(const float *f1, const float *f2, int i) {
2     return f1[i] * f2[i] + f1[i + 1] * f2[i + 1] + f1[i + 2] * f2[i + 2] +
3     f1[i + 3] * f2[i + 3] +
4     f1[i + 4] * f2[i + 4] + f1[i + 5] * f2[i + 5];
5 }
```

```
1 // begin time
2 auto start1 = std::chrono::steady_clock::now();
3 int m = n % 6;
4 for (int i = 0; i < m; ++i) {
5     result += f1[i] * f2[i];
6 }
7 for (int i = m; i < n; i += 6) {
8     result += block(f1, f2, i);
9 }
10 // end time
11 auto end1 = std::chrono::steady_clock::now();
```

## Time Cost

test this block vector algorithm with two vectors each has 200 million elements and output its' calculation time.



```
Run: assignment3
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -364864823679699124224.000000
calculation takes 427 ms
Process finished with exit code 0
```

We can see from the screenshot that after this divide step, the time cost is reduced by half.

## Version\_3.0

### Parallel Computing

We take notice of that in version\_2.0 when we do add operation to add the block sum to result, we just simply let `result += block(f1, f2, i)`. But according to Eric said in "Optimizing Dot Product"(Dec 10, 2012), "the problem is, this code must still run basically sequentially because each statement depends on the previous statement." Therefore, generated by this idea. We can do the optimization as followed codes.

```

1  float dot1 = 0.0, dot2 = 0.0, dot3 = 0.0, dot4 = 0.0, dot5 = 0.0, dot6 =
   0.0;
2
3  void block(const float *f1, const float *f2, int i) {
4      dot1 += f1[i] * f2[i];
5      dot2 += f1[i + 1] * f2[i + 1];
6      dot3 += f1[i + 2] * f2[i + 2];
7      dot4 += f1[i + 3] * f2[i + 3];
8      dot5 += f1[i + 4] * f2[i + 4];
9      dot6 += f1[i + 5] * f2[i + 5];
10 }

```

```

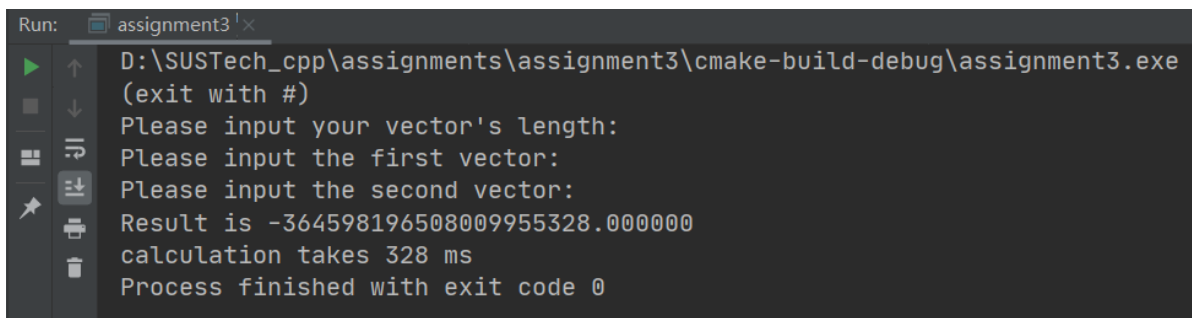
1  for (int i = m; i < n; i += 6) {
2      block(f1, f2, i);
3  }
4  result += dot1 + dot2 + dot3 + dot4 + dot5 + dot6;

```

This meant there was no longer any dependency between statements, and the CPU's out of order execution engine could run all four of these as soon as the data was available. In expectation, it will run faster than just the block calculation.

## Time Cost

Let's test this parallel computing algorithm with two vectors each has 200 million elements and output its' calculation time.



```

Run: assignment3
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -364598196508009955328.000000
calculation takes 328 ms
Process finished with exit code 0

```

We can see from the pic that this parallel design based on block actually has some progress in time although this optimization is not so obvious.

## version\_4.0

### Using SSE(Streaming SIMD Extensions) Optimized Dot Product

SSE is short for Internet Streaming SIMD Extensions. In addition to maintaining the original MMX instructions, SSE added 70 new instructions, which improved the efficiency of memory usage and made the memory operations faster while speeding up floating point operation. Here we use SSE to deal with the vector dot product and see how fast it works!

The codes below is partially learnt from what I searched in the Internet. And here showing these codes is just a summary a new optimization of dot product.

```

1  float sse_inner(const float *a, const float *b, unsigned int size) {
2      float z = 0.0f, fres = 0.0f;
3      float ftmp[4] = {0.0f, 0.0f, 0.0f, 0.0f};

```

```

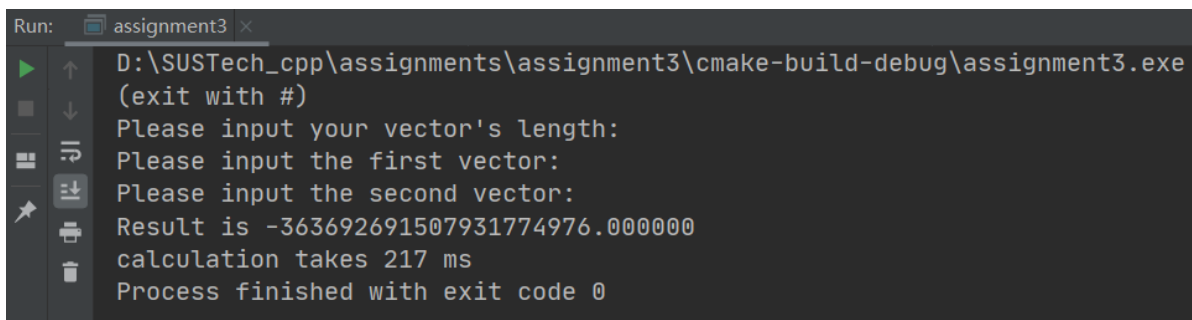
4     __m128 mres;
5     if ((size / 4) != 0) {
6         mres = _mm_load_ss(&z);
7         for (unsigned int i = 0; i < size / 4; i++) {
8             mres = _mm_add_ps(mres, _mm_mul_ps(_mm_loadu_ps(&a[4 * i]),
9             _mm_loadu_ps(&b[4 * i])));
10        }
11        __m128 mv1 = _mm_movehl_ps(mres, mres);
12        __m128 mv2 = _mm_movehl_ps(mres, mres);
13        mres = _mm_add_ps(mv1, mv2);
14        _mm_store_ps(ftmp, mres);
15        fres = ftmp[0] + ftmp[1];
16    }
17    if ((size % 4) != 0) {
18        for (unsigned int i = size - size % 4; i < size; i++) {
19            fres += a[i] * b[i];
20        }
21    }
22
23    return fres;
24 }

```

```
1 result += sse_inner(f1, f2, n);
```

## Time Cost

Let's test this parallel computing algorithm with two vectors each has 200 million elements and output its' calculation time.



```

Run: assignment3 x
D:\SUSTech_cpp\assignments\assignment3\cmake-build-debug\assignment3.exe
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -363692691507931774976.000000
calculation takes 217 ms
Process finished with exit code 0

```

We notice that the progress is obvious since we use SIMD floating point register here to do the task. The time reduces by 100ms or so.

## Test With OpenBLAS

### OpenBLAS

OpenBLAS is an optimized Basic Linear Algebra Subprograms (BLAS) library based on GotoBLAS2 1.13 BSD version. It's one of the fastest matrix operation library in the world. Let's see how it behaves under our 400 million vector elements!


```
1 result += cblas_sdot(n, f1, 1, f2, 1);
```

Before running the code, we have to compile the project cloned from github website, link it into the project cmakeFile and include the head file.

```
1 | #include <cblas.h>
```

## Time Cost

Let's test this parallel computing algorithm with two vectors each has 200 million elements and output its' calculation time.

 Microsoft Visual Studio 调试控制台

```
(exit with #)
Please input your vector's length:
Please input the first vector:
Please input the second vector:
Result is -364535040560110501888.000000
calculation takes 166(null)
```

## Summary

In this assignment we are working on how to optimizing vector dot product and how different ideas works and why. Up to now, we have worked out 4 versions and seven ways to do the dot product, from brutal\_double to OpenBLAS. And each of them differs in efficiency, let's summarize from the data below.

method	result	time cost
brutal_vector	-365884255676600942592.000000	2042 ms
brutal_double	-364535047238995935232.000000	1206 ms
brutal_float	-365884255676600942592.000000	1165 ms
block	-364864823679699124224.000000	427 ms
parallel	-364598196508009955328.000000	328 ms
SSE	-363692691507931774976.000000	217 ms
OpenBLAS	-364535040560110501888.000000	116 ms

We note that the results are different in some method, that's because in different method, we deal with those data in different ways. For example, between block and brutal\_float, in brutal\_float we simply add each product together to the final result(every element can cause a precision lost to the final result), but in block we first let a unit block(length 6) multiple correspondingly and add them together, then add this block's result to the final result(every six elements may cause a precision lost). See from the chart, the result of block is -364864823679699124224.000000 which is loser to -364535040560110501888.000000(OpenBLAS) than -365884255676600942592.000000(brutal\_float), This also verify our explanation!

## Others

## About the test case

All the test cases used above to test each method's efficiency is from the same pair of vectors, they are stored in two separated binary file(200M float data occupies almost 800MB disk memory) and tested by the same computer(the same CPU). Therefore, using them to do the comparison is convincing. Which needs mentioning is that here we only record the calculation time cost but ignore the IO time cost.

 random1.dat	2020/10/13 12:04	DAT 文件	781,250 KB
 random2.dat	2020/10/13 12:05	DAT 文件	781,250 KB

Below shows how we utilize these 200M data:

```
1  int n = 200000000;
2      ifstream file1(R"(D:\SUSTech_cpp\assignments\assignment3\random1.dat)",
ios::binary);
3      ifstream file2(R"(D:\SUSTech_cpp\assignments\assignment3\random2.dat)",
ios::binary);
4      auto* f1 = new float[n];
5      auto* f2 = new float[n];
```

```
1  if (file1.is_open()) {
2      for (int i = 0; i < n; ++i) {
3          file1.read(reinterpret_cast<char*>(&f1[i]), sizeof(f1[i]));
4      }
5  }
```

```
1  if (file2.is_open()) {
2      for (int i = 0; i < n; ++i) {
3          file2.read(reinterpret_cast<char*>(&f2[i]), sizeof(f2[i]));
4      }
5  }
```

## Source Code

See and clone the whole project from this github repository!