矩阵乘法的优化

姓名：郁万祥　　学号：2013852

一、问题描述

分别实现四种矩阵乘法程序，并对比运行效率

1） 串行算法

2） Catch 优化

3） SSE 版本

4） 分片策略

二、算法设计与实现

1、串行算法：

1）算法概述：

A,B 为待乘矩阵，C 为结果矩阵，遍历 A 的第 i 个行向量与 B 的第 i 个列向量，乘积之和作为 C 的第 i 个分量，依次遍历，得到结果矩阵 C。

2）时间复杂度分析：

对矩阵 A 的每一行的元素，矩阵 B 的每一列的元素都要与之相乘并相加，所以需要先遍历矩阵 A 一行中的元素的同时遍历矩阵 B 一列中的元素，再遍历矩阵 B 中的每一列，再遍历矩阵 A 中的每一行，三层遍历嵌套，时间复杂度 O(n) = n^3。

2、Catch 优化：

1）算法概述：

考虑到 C++中的数组在内存中的物理地址是连续的，而按照穿行算法，每次都会进行跳跃寻址，又因为，在程序运行的过程中，会先访问速度更快的 cache，因此，我们通过先将 b 转置，从而实现增大程序访存过程中 cache 的命中率，从而提高程序运行的速度。

2）时间复杂度分析：

因为此算法只是通过提高了 cache 的命中率优化了运算过程中访存

的时间，而对于程序本省而言，依然通过了三层 for 循环，因此时间复杂度依旧是 O(n) = n^3。

3、SSE 版本：

1）算法概述：

基于 SIMD 编程，将多个元素同时进行寄存器加载和存储器存值等操作能够提高并行效率。使用 SSE 指令能够实现该功能。同样对矩阵 B 进行转置。

2）时间复杂度分析：

时间复杂度为 O(n) = (n^3)/4。

4、分片策略：

1）算法概述：

先针对 cpu 线程数设置分片数，将矩阵分为多个小矩阵，对每个小矩阵进行 SSE 指令计算，同样对矩阵 B 进行转置。

2）时间复杂度分析：

时间复杂度为 O(n) = n^3

三、实验及结果分析

1、测试数据：

1）、对于矩阵规模 n，从 10 到 1000 采取递增策略，100 以内每次递增 10，100~1000 每次递增 100。

2）、同时设定固定矩阵规模 500，采取 10 次取平均值的方法，观察每种算法的运行时间。

2、测试结果：

1）、

```
C:\Workspace\CodeBlocksProjects\demo\bin\Debug\demo.exe
现在的矩阵规模为：10
串行算法：0.0197ms    cache优化算法：0.0016ms    SSE版本：0.0014ms    分片策略：0.0007ms
现在的矩阵规模为：20
串行算法：0.074ms    cache优化算法：0.0208ms    SSE版本：0.0183ms    分片策略：0.0108ms
现在的矩阵规模为：30
串行算法：0.2103ms    cache优化算法：0.0631ms    SSE版本：0.0525ms    分片策略：0.0403ms
现在的矩阵规模为：40
串行算法：0.5791ms    cache优化算法：0.1357ms    SSE版本：0.103ms    分片策略：0.0733ms
现在的矩阵规模为：50
串行算法：1.1007ms    cache优化算法：0.2204ms    SSE版本：0.1522ms    分片策略：0.1113ms
现在的矩阵规模为：60
串行算法：1.8854ms    cache优化算法：0.3562ms    SSE版本：0.2271ms    分片策略：0.1485ms
现在的矩阵规模为：70
串行算法：3.0252ms    cache优化算法：0.5579ms    SSE版本：0.3075ms    分片策略：0.284ms
现在的矩阵规模为：80
串行算法：5.7589ms    cache优化算法：0.8144ms    SSE版本：0.4199ms    分片策略：0.3331ms
现在的矩阵规模为：90
串行算法：9.6116ms    cache优化算法：1.1163ms    SSE版本：0.5835ms    分片策略：0.3967ms
现在的矩阵规模为：100
串行算法：14.8396ms    cache优化算法：2.5025ms    SSE版本：0.7329ms    分片策略：0.4541ms
现在的矩阵规模为：200
串行算法：115.398ms    cache优化算法：10.8284ms    SSE版本：3.7322ms    分片策略：3.4144ms
现在的矩阵规模为：300
串行算法：387.783ms    cache优化算法：35.1766ms    SSE版本：10.8834ms    分片策略：7.6518ms
现在的矩阵规模为：400
串行算法：931.318ms    cache优化算法：81.7902ms    SSE版本：25.4173ms    分片策略：21.0565ms
现在的矩阵规模为：500
串行算法：1815.56ms    cache优化算法：160.442ms    SSE版本：50.181ms    分片策略：32.9446ms
现在的矩阵规模为：600
串行算法：3114.45ms    cache优化算法：297.67ms    SSE版本：122.795ms    分片策略：65.3449ms
现在的矩阵规模为：700
串行算法：3805.01ms    cache优化算法：309.435ms    SSE版本：149.561ms    分片策略：53.2271ms
现在的矩阵规模为：800
串行算法：4652.21ms    cache优化算法：433.828ms    SSE版本：201.669ms    分片策略：89.4406ms
现在的矩阵规模为：900
串行算法：6738.37ms    cache优化算法：571.199ms    SSE版本：266.058ms    分片策略：138.548ms
现在的矩阵规模为：1000
串行算法：9287.76ms    cache优化算法：772.483ms    SSE版本：351.122ms    分片策略：171.519ms

Process returned 1 (0x1)   execution time : 35.429 s
Press any key to continue.
```

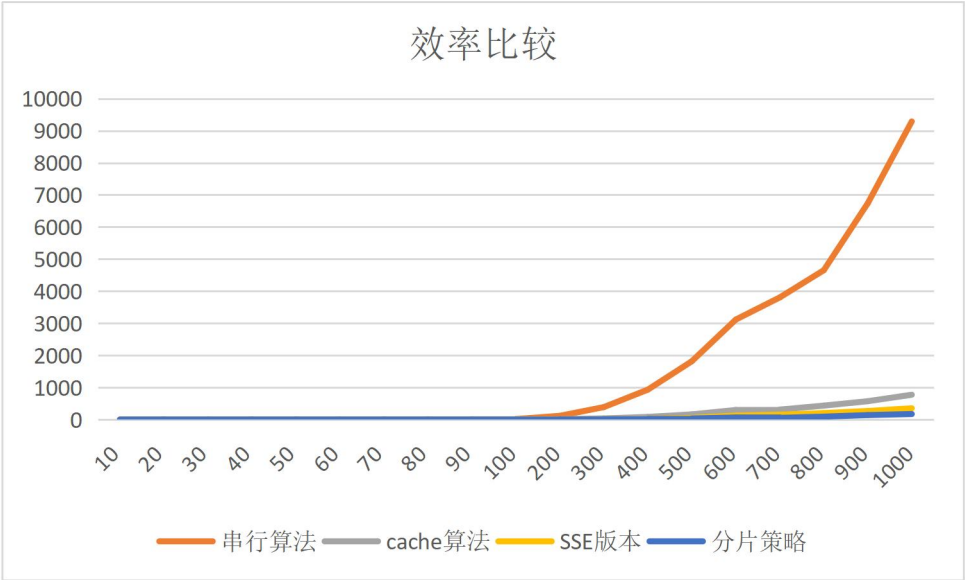
2）、

```
C:\Workspace\CodeBlocksProjects\demo\bin\Debug\demo.exe
矩阵规模n为：500
串行算法：1140.4ms    cache优化算法：167.48ms    SSE版本：57.0463ms    分片策略：37.8762ms
矩阵规模n为：500
串行算法：1121.7ms    cache优化算法：167.05ms    SSE版本：54.0792ms    分片策略：32.5717ms
矩阵规模n为：500
串行算法：1121.97ms    cache优化算法：158.988ms    SSE版本：49.2788ms    分片策略：32.3904ms
矩阵规模n为：500
串行算法：1119.33ms    cache优化算法：169.757ms    SSE版本：48.6213ms    分片策略：32.4446ms
矩阵规模n为：500
串行算法：1121.78ms    cache优化算法：158.75ms    SSE版本：47.9648ms    分片策略：32.2154ms
矩阵规模n为：500
串行算法：1118.91ms    cache优化算法：145.804ms    SSE版本：43.1776ms    分片策略：27.766ms
矩阵规模n为：500
串行算法：794.753ms    cache优化算法：94.6502ms    SSE版本：30.451ms    分片策略：19.2419ms
矩阵规模n为：500
串行算法：666.914ms    cache优化算法：94.547ms    SSE版本：27.9728ms    分片策略：19.2325ms
矩阵规模n为：500
串行算法：669.888ms    cache优化算法：96.5392ms    SSE版本：28.3049ms    分片策略：19.3628ms
矩阵规模n为：500
串行算法：666.943ms    cache优化算法：95.0481ms    SSE版本：29.3085ms    分片策略：19.2115ms
串行算法平均耗时：953ms
cache优化算法平均耗时：134ms
SSE算法平均耗时：41ms
分片算法平均耗时：26ms

Process returned 1 (0x1)   execution time : 11.664 s
Press any key to continue.
```

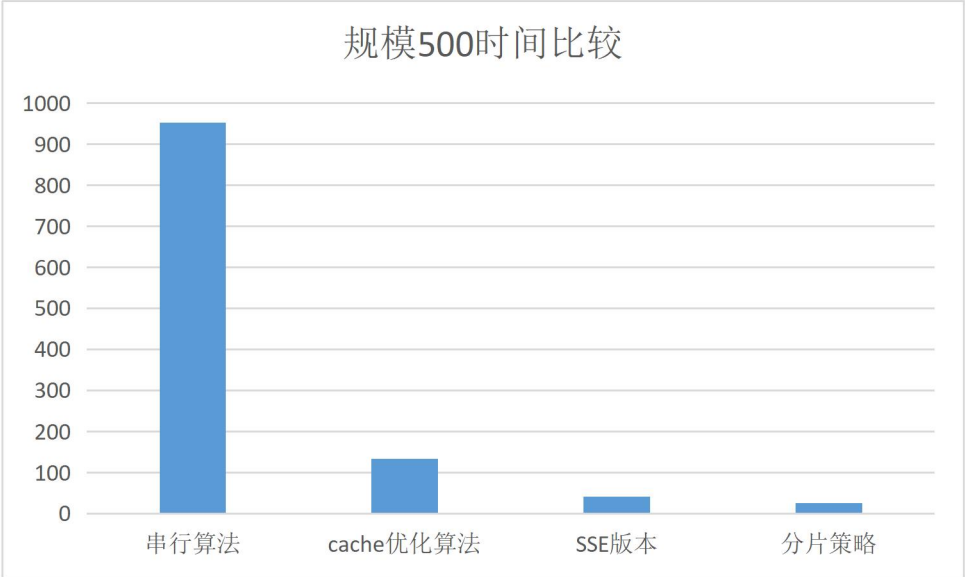
3、结果分析：

1）、



可以看到，随着规模逐渐增大，cache 算法，SSE 算法和分片策略相对于串行算法的优化效果越来越明显。

2）、



可以看到，四种算法优化程度比较：分片策略>SSE 版本>cache 优化算法>串行算法。

高斯消元法 SSE 并行化

一、算法分析设计：

算法伪码：

```
1. procedure LU (A)
2. begin
3.    for k := 1 to n do
4.        for j := k+1 to n do
5.            A[k, j] := A[k, j]/A[k, k];
6         A[k, k] := 1.0;
7.        endfor;
8.        for i := k + 1 to n do
9.            for j := k + 1 to n do
10.               A[i, j] := A[i, j] - A[i, k] × A[k, j ];
11.           endfor;
12.           A[i, k] := 0;
13.       endfor;
14. endfor;
15. end LU
```

通过分析高斯的程序可以发现，高斯消去法有两部分可以实现并行，分别是第一部分的除法和第二部分的减法。即:

1.第一个内嵌的 for 循环里的 A[k, j]:= A[k, j]/A[k, k]; 我们可以做除法并行。

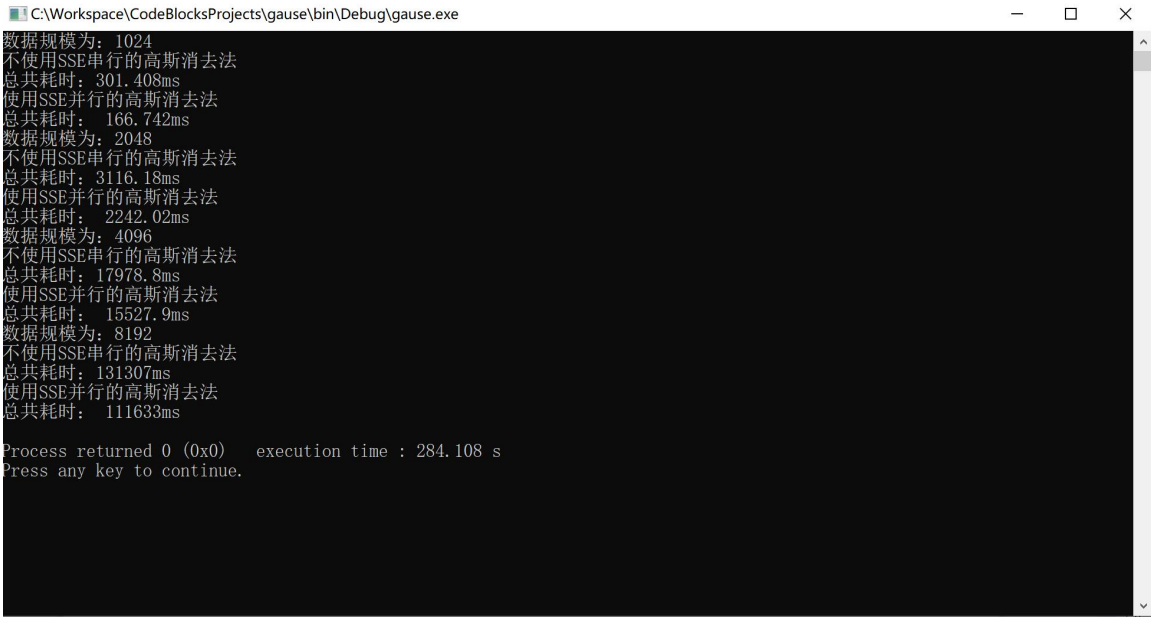
2.第二个双层 for 循环里的 A[i, j] := A[i, j] - A[i, k] × A[k, j];我们可以做减法并行。

二、实验结果：

1、数据选取：

为了尽可能地贴合并行算法的实际使用情况，此次实验没有选取规模较小的数据进行测试，而是从比较大的数据开始测试，因此，此次实验所选区的数据规模从小到大依次取：1024，2048，4096，8192。

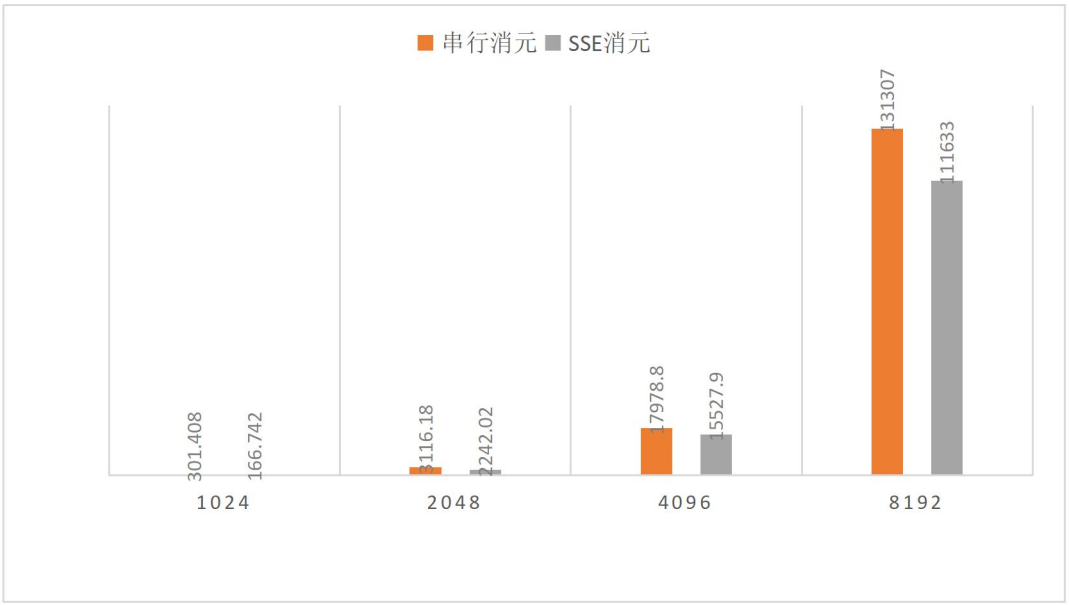
2、实验运行结果：



三、讨论：

1、相同算法对于不同问题规模的性能提升是否有影响，影响情况如何：

我们根据实验结果将串行消元和 SSE 消元作比较：



我们可以看到，相同算法对于不同问题规模的性能提升有一定的影响，但是影响情况不大。

2、消元过程中采用向量编程的性能提升情况如何：

消元过程中，如果数据比较多，比较普遍，经过计算，向量编程的性能提升大约在 14%左右，因此在消元过程中,向量编程的性能提升情况一般。

3、回代过程可否向量化，有的话性能提升情况如何；

回代过程可以向量化，但是对性能提升效果不佳。

4、数据对齐与不对齐对计算性能有怎样的影响

对于串行算法，不需要数据对齐，因此数据对齐与否对时间复杂度与性能影响不大，对于 SSE 并行算法，因为在并行计算前，数据对齐是固定的步骤，如果数据本身对对齐性较差，则需要额外的时间开销进行数据对齐，因此对性能影响比较大。

正是可能因为 SSE 有一些固定的步骤，才导致了 SSE 对于高斯消元优化的效果并不是特别明显。