编程作业3.1：矩阵乘法的优化

题目：实现课件中四种矩阵乘法程序，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

要求：提交实验报告（问题描述、算法设计（最好有复杂性分析）与实现、实验及结果分析）和源码（只将程序文件和工程文件提交，不要将编译出的目标文件和可执行文件也打包提交）。

1 问题描述

本实验分别使用了串行算法、Cache 优化、SSE 编程和分片策略四种算法实现了矩阵乘法，详细设计算法，进行复杂度的分析，并对比运行效率。

2 算法设计与复杂性分析

2.1 串行算法

矩阵 A 与 B 相乘，乘积的第 i 个分量为矩阵 A 的第 i 个行向量与 B 的列向量对应的乘积之和。串行计算时需要先遍历矩阵 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，其中A中的每一行元素只需要被读取一次，而B中的每一行元素需要被读取n次，时间复杂度为 $O(n) = n^3$。

```
// serial 串行
void serial(float a[maxN][maxN], float b[maxN][maxN]){
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

2.2 Catch优化

由于寄存器从内存中取值时，需要从内存中寻址，串行算法需要先对矩阵 B 的列进行遍历，又因为在 c++ 中，二维数组的行相邻元素地址相近，而二维数组列相邻的元素地址很远，在进行列读取时，cache命中率很低，因此对列进行遍历造成了地址跳跃，需要更长的寻址时间。将矩阵 B 转置，就能够实现对连续的地址进行读写操作，节省大量寻址时间。算法时间复杂度 $O(n) = n^3$。

```
// cache 缓存优化
void cache(float a[maxN][maxN], float b[maxN][maxN]){
    // transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                c[i][j] += a[i][k] * b[j][k];
            }
        }
    }

    // transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }
}
```

2.3 SSE版本

使用 SSE 指令集，将多个元素同时进行寄存器加载和存储器存值等操作，使用向量化计算矩阵乘法。因为 _mm_loadu_ps 同时加载四个单精度浮点数值，故循环每次跳跃四个元素，算法时间复杂度 $O(n) = (n^3) / 4$

```
// SSE SSE编程
void sse(float a[maxN][maxN], float b[maxN][maxN]){
    __m128 t1, t2, sum;
```

```
  // transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }

    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            c[i][j] = 0.0;
            sum = _mm_setzero_ps();
            for (int k = n - 4; k >= 0; k -= 4){
                // sum every 4 elements
                t1 = _mm_loadu_ps(a[i] + k);
                t2 = _mm_loadu_ps(b[j] + k);
                t1 = _mm_mul_ps(t1, t2);
                sum = _mm_add_ps(sum, t1);
            }
            sum = _mm_hadd_ps(sum, sum);
            sum = _mm_hadd_ps(sum, sum);
            _mm_store_ss(c[i] + j, sum);
            for (int k = (n % 4) - 1; k >= 0; --k){
                // handle the last n%4 elements
                c[i][j] += a[i][k] * b[j][k];
            }
        }
    }

// transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }
}
```

2.4 分片策略

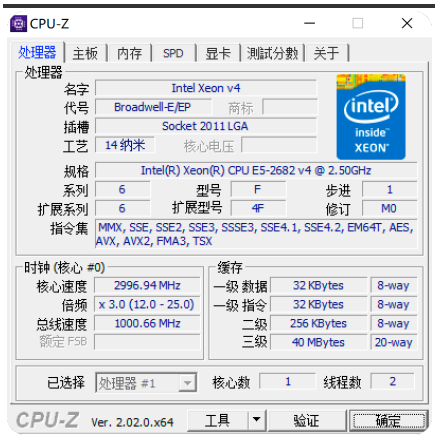
cache 不变，矩阵越大，越难装进 cache 中，高速缓存缺失更多。充分利用高速缓存的局部性原理将矩阵乘法分块，设计一个合适的块大小可以一次将小块进行缓存，实现分片策略算法：先设置分片数，将矩阵分为多个小矩阵，再对每个小矩阵进行 SSE 指令计算。算法的时间复杂度仍然为 $O(n) = (n^3)$ ，但因充分利用了高速缓存，故计算性能也会有较显著提升。

```
// tile 分片策略
void tile(float a[maxN][maxN], float b[maxN][maxN]){
    __m128 t1, t2, sum;
    float t;

  // transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }

    for (int r = 0; r < n / T; ++r)
        for (int q = 0; q < n / T; ++q){
            for (int i = 0; i < T; ++i)
                for (int j = 0; j < T; ++j){
                    c[r * T + i][q * T + j] = 0.0;
                }
            for (int p = 0; p < n / T; ++p){
                for (int i = 0; i < T; ++i)
                    for (int j = 0; j < T; ++j){
                        sum = _mm_setzero_ps();
                        for (int k = 0; k < T; k += 4){
                            //sum every 4th elements
                            t1 = _mm_loadu_ps(a[r * T + i] + p * T + k);
                            t2 = _mm_loadu_ps(b[q * T + j] + p * T + k);
                            t1 = _mm_mul_ps(t1, t2);
                            sum = _mm_add_ps(sum, t1);
                        }
                        sum = _mm_hadd_ps(sum, sum);
                        sum = _mm_hadd_ps(sum, sum);
                        _mm_store_ss(&t, sum);
                        c[r * T + i][q * T + j] += t;
                    }
            }
        }
```

```
// transposition
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j){
            float t=b[i][j];
            b[i][j]=b[j][i];
            b[j][i]=t;
        }
}
```

3 实验设计

3.1 实验环境



运行实验的电脑CPU

操作系统 win10，编辑器 codeblocks，编译器 TDM-GCC

3.2 实验细节设计

1、 计时：

a) 计时方法：参考课件示例代码中的计时方法，使用QueryPerformance系列API；

b) 计时精度：即使精度最高的计时机制，也可能测量不出太快的计算过程。如何解决？将计算过程重复多次，足以匹配计时精度，再计算平均时间。——将每一个计算过程重复 50 次，累积时间后除以 50 计算平均时间。

```
QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        QueryPerformanceCounter((LARGE_INTEGER *)&head);
        for(int i=0;i<50;i++){
```

```
        serial(dataset[i].a,dataset[i].b);
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    time[0][n/50]=((tail-head)*1000.0/freq)/50;
    cout<<time[0][n/50]<<endl;

    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<50;i++){
        cache(dataset[i].a,dataset[i].b);
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    time[1][n/50]=((tail-head)*1000.0/freq)/50;
    cout<<time[1][n/50]<<endl;

    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<50;i++){
        sse(dataset[i].a,dataset[i].b);
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    time[2][n/50]=((tail-head)*1000.0/freq)/50;
    cout<<time[2][n/50]<<endl;

    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for(int i=0;i<50;i++){
        tile(dataset[i].a,dataset[i].b);
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    time[3][n/50]=((tail-head)*1000.0/freq)/50;
    cout<<time[3][n/50]<<endl;
```

2、 实验设计：

a) 实验数据：矩阵大小可自定，也可测试不同矩阵规模，由小至大，随机生成矩阵元素值——矩阵规模从 0 到 1000，间隔为 50；且每个规模下由随机数组成的 50 个矩阵样本。

```
for(n=0;n<=1000;n+=50){
        init(dataset);
  //实验内容
    }
```

```
// 随机生成矩阵
void init(Matrix* dataset) {
    srand(static_cast <unsigned> (time(0)));
    for (int i = 0; i < 50; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                dataset[i].a[j][k] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / 1000));
                dataset[i].b[j][k] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / 1000));
            }
        }
    }
}
```

b) 实验方法：为降低误差，每个测试应重复多次——累积计算这 50 个矩阵乘法后计算平均执行时间；执行 3 次，获得 3 组各规模下的运行时间，再计算平均时间以避免误差。

第一组

```
Process returned 0 (0x0)   execution time : 4474.938 s
Press any key to continue.
Process terminated with status 0 (88 minute(s), 25 second(s))
```

第二组

```
Process returned 0 (0x0)   execution time : 1447.084 s
Press any key to continue.
Process terminated with status 0 (600 minute(s), 42 second(s))
```

第三组

```
Process returned 0 (0x0)   execution time : 1411.596 s
Press any key to continue.
Process terminated with status 0 (29 minute(s), 16 second(s))
```

3、 报告撰写:

a) 有清晰的结果呈现，可以结合图表数据分析各算法性能。

3.3 实验数据分析

过程：进行了3组的实验，每组都跑了矩阵规模为0-1000，间隔50的实验，由系统生成随机数随机生成50个相应规模的矩阵进行乘法，累积时间后除以 50 计算该矩阵规模下，该算法的矩阵乘法的平均时间。

生成的3组数据打包在实验结果的txt文件中，以下是由实验结果生成的图表数据分析：

3.3.1 串行实现的3次实验

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.20E-05	2.60E-05	2.80E-05	2.53333E-05
50	0.392974	0.386722	0.40146	0.393718667
100	3.30861	3.2779	3.35332	3.313276667
150	10.8266	10.9151	10.9082	10.8833
200	25.4078	25.5228	26.0422	25.6576
250	49.8336	49.6616	50.5942	50.0298
300	88.4368	85.3413	88.0136	87.2639
350	137.505	242.609	138.143	172.7523333
400	203.484	539.99	207.54	317.0046667
450	290.537	362.733	295.666	316.312
500	401.591	408.173	408.221	405.995
550	537.176	539.867	545.065	540.7026667
600	693.712	693.708	699.589	695.6696667
650	887.388	886.006	886.978	886.7906667
700	1087.38	1129.78	1104.8	1107.32
750	1356.66	1351.28	1365.74	1357.893333
800	1643.91	1633.42	1654.78	1644.036667
850	1967.72	1997.91	1982.61	1982.746667
900	2329.64	2369.74	2365.04	2354.806667
950	2793.99	2771.15	2782.44	2782.526667
1000	3218.8	3244.43	3258.75	3240.66

3.3.2 缓存优化的3次实验

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.00E-05	3.20E-05	2.20E-05	2.46667E-05
50	0.411946	0.215856	0.198744	0.275515333
100	1.38824	1.4073	1.3918	1.39578
150	4.33584	4.32596	4.50097	4.38759
200	9.92556	9.73443	9.82846	9.829483333
250	18.9184	18.6677	19.0158	18.8673
300	31.9891	32.3832	32.2515	32.20793333
350	49.328	136.015	50.8681	78.73703333
400	73.7342	123.406	74.1358	90.42533333
450	103.829	113.413	106.42	107.8873333
500	142.721	143.642	143.787	143.3833333
550	188.579	190.468	190.999	190.0153333
600	251.374	244.454	245.692	247.1733333
650	308.36	312.417	313.296	311.3576667
700	383.274	387.355	391.077	387.2353333
750	478.838	470.911	480.193	476.6473333
800	574.333	632.951	577.601	594.9616667
850	674.363	684.2	681.296	679.953
900	806.528	808.754	808.301	807.861
950	961.438	947.275	947.017	951.91
1000	1098.27	1104.55	1108.8	1103.873333

3.3.3 SSE编程时间

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.20E-05	3.40E-05	2.40E-05	2.66667E-05
50	0.10706	0.096484	0.100258	0.101267333
100	0.669732	0.67519	0.634362	0.659761333
150	1.72495	1.76289	1.6915	1.726446667
200	4.1667	3.80115	3.84629	3.938046667
250	6.93668	6.79293	6.85964	6.863083333
300	12.7371	11.7974	12.3516	12.29536667
350	18.0653	57.2675	18.3031	31.21196667
400	28.6733	73.2315	27.5979	43.16756667
450	37.8396	40.6387	38.9275	39.13526667
500	44.299	45.2432	45.7749	45.1057
550	57.4652	59.5785	59.4104	58.81803333
600	76.2743	76.5073	80.7439	77.84183333
650	95.4493	100.409	98.9074	98.25523333
700	122.977	126.224	127.54	125.5803333
750	165.992	161.005	170.73	165.909
800	181.208	183.249	184.59	183.0156667
850	225.378	232.743	229.712	229.2776667
900	233.711	236.528	238.411	236.2166667
950	279.461	279.562	273.162	277.395
1000	318.548	319.472	327.065	321.695

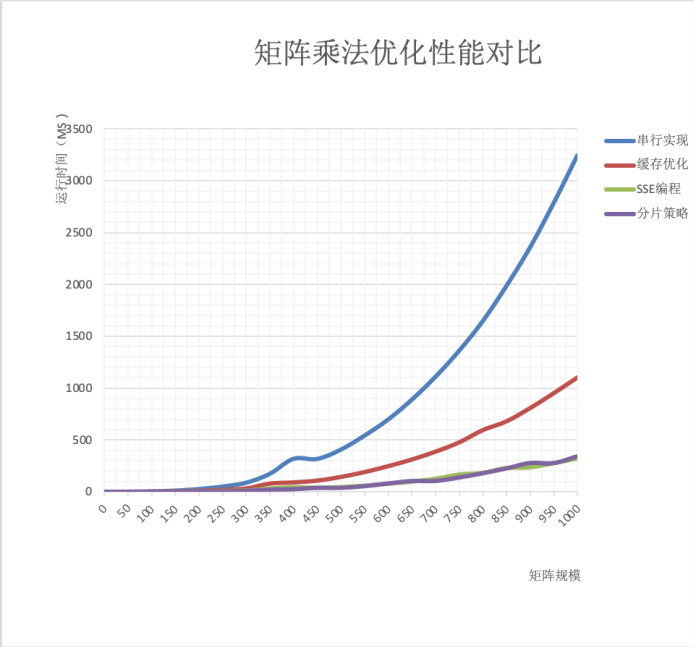
3.3.4 分片策略时间

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	3.00E-05	2.80E-05	3.00E-05	2.93333E-05
50	0.031604	0.036362	0.029264	0.03241
100	0.259786	0.272044	0.266896	0.266242
150	1.21908	1.27291	1.21254	1.234843333
200	3.40007	3.61324	3.58725	3.53352
250	3.71091	3.65636	3.94321	3.77016
300	8.52167	7.85564	8.30631	8.227873333
350	14.192	31.2588	15.7929	20.41456667
400	23.4712	26.913	24.2578	24.88066667
450	36.3163	37.8041	37.9938	37.3714
500	37.4502	37.1684	38.8448	37.82113333
550	52.6138	54.039	55.2284	53.9604
600	81.6385	75.8482	80.358	79.28156667
650	102.271	104.332	102.145	102.916
700	102.566	102.45	106.818	103.9446667
750	132.906	132.368	144.842	136.7053333
800	172.072	177.949	182.47	177.497
850	220.944	221.126	230.013	224.0276667
900	277.399	266.371	282.291	275.3536667
950	272.514	274.351	277.855	274.9066667
1000	339.746	329.808	349.302	339.6186667

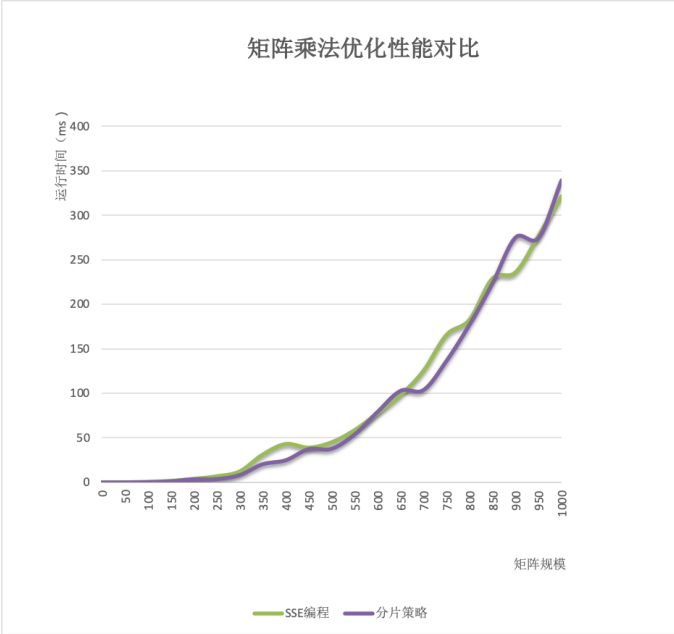
3.3.5 对比分析

矩阵乘法实现方法	串行实现（ms）	缓存优化（ms）	SSE编程（ms）	分片策略（ms）
规模（矩阵大小）	平均时间	平均时间	平均时间	平均时间
0	2.53333E-05	2.46667E-05	2.66667E-05	2.93333E-05
50	0.393718667	0.275515333	0.101267333	0.03241
100	3.313276667	1.39578	0.659761333	0.266242
150	10.8833	4.38759	1.726446667	1.234843333
200	25.6576	9.829483333	3.938046667	3.53352
250	50.0298	18.8673	6.863083333	3.77016
300	87.2639	32.20793333	12.29536667	8.227873333
350	172.7523333	78.73703333	31.21196667	20.41456667
400	317.0046667	90.42533333	43.16756667	24.88066667
450	316.312	107.8873333	39.13526667	37.3714
500	405.995	143.3833333	45.1057	37.82113333
550	540.7026667	190.0153333	58.81803333	53.9604
600	695.6696667	247.1733333	77.84183333	79.28156667
650	886.7906667	311.3576667	98.25523333	102.916
700	1107.32	387.2353333	125.5803333	103.9446667
750	1357.893333	476.6473333	165.909	136.7053333
800	1644.036667	594.9616667	183.0156667	177.497
850	1982.746667	679.953	229.2776667	224.0276667
900	2354.806667	807.861	236.2166667	275.3536667
950	2782.526667	951.91	277.395	274.9066667
1000	3240.66	1103.873333	321.695	339.6186667

使用上述的平均时间绘制折线图，如下：



四种算法对比



SSE与分片对比

可以看出，矩阵乘法的串行算法时间比起其他算法的来说，时间太长，进行缓存优化后，算法时间大幅度减小，进行sse编程后，算法时间再次大幅减小，但是在sse编程基础上再进行分片策略的话，有效果但时间几乎一致，没有什么大幅度的提高。

由于sse与分片的时间折叠，因此分开进行对比，可以观察到，当数据量小的时候或者大部分时候，分片算法的效果确实优于没有分片的，

但是事实上由于数据的波动与随机性，我们也可以看到有一些地方有所波动，导致没有分片的效果更好。

3.3.6 分析总结

3.3.6.1 不同算法性能对比：

耗时最多的算法是串行算法，也是最原始的算法。

Cache 优化对串行算法的取址时间进行了压缩，有很明显的优化效果，时间大约缩短了一半左右。

SSE 编程引入SSE的办法，一次处理四个数据，使得复杂度缩小4倍，算法时间缩短。

分片策略算法消耗的时间最少，但与 SSE 差别不算太明显。

3.3.6.2 矩阵规模对算法性能的影响：

一开始，各个算法之间的时间差距并不明显。随着矩阵规模的增加，串行算法和缓存优化的时间增加越来越快，SSE 编程和分片策略增加缓慢；且各个算法之间的时间差距在逐渐扩大。

3.3.6.3 从加速比的角度分析：

SSE 编程引入SSE的办法，一次处理四个数据，SSE 编程和 Cache 缓存优化加速比理论上是4倍，实验证明加速度比接近 4 但未超过 4，说明 SSE 编程一次计算处理四个单精度浮点数，同时并行处理需要消耗时间，故加速比不能为 4，只能逼近。

3.3.6.4 SSE与分片策略的波动问题

分片策略与 SSE 版本用时差距不大，某些节点出现时间数据的波动，可能是因为分片数量不理想，并行成本消耗过高，导致优化的效果不够明显。

编程作业3.2：高斯消元法SSE并行化

题目：首先熟悉高斯消元法解线性方程组的过程（见**附录2**），然后实现SSE算法编程。过程中，请自行构造合适的线性方程组，并选取至少2个角度，讨论不同算法策略对性能的影响。

可选角度**包括但不限于**以下几种选项：

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；

② 消元过程中采用向量编程的的性能提升情况如何；

③ 回代过程可否向量化，有的话性能提升情况如何；

④ 数据对齐与不对齐对计算性能有怎样的影响；

⑤ SSE编程和AVX编程性能对比。

要求：

1. 提交实验报告（问题描述、算法设计（最好有复杂性分析）与实现、实验及结果分析）和源码（只将程序文件和工程文件提交，不要将编译出的目标文件和可执行文件也打包提交）。
2. 与编程作业3.1一起，写成一份报告。作业的命名格式，“姓名-学号-并行第2次作业”，如果是多个文件，需要打包为“.zip”格式并按如上方式命名。

1 问题描述

本实验分别使用了串行算法、SSE 编程分别实现了高斯消元法和回代求解的过程，并就各算法的执行性能进行比较与分析。

2 算法设计与复杂度分析

2.1 高斯消元法

2.1.1 串行算法消元

矩阵 A 和 x 相乘等于 b，将矩阵 A 化为上三角矩阵。将第 2 行至第 n 行，每行分别与第一行做运算，消掉每行第一个参数。从新矩阵的 a22 开始（a22 不能为 0），以第二行为基准，将第三行至第 n 行分别与第二行做运算，消掉每行第二个参数。经过 n-1 步，方程组也就转化为了我们希望得到的上三角方程组。算法 的时间复杂度为 O(n) = n^3。

```
// serial串行消元
void serialGauss(Matrix* dataset) {
    for (int count = 0; count < 50; count++) {
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                float temp = dataset[count].a[j][i] / dataset[count].a[i][i];
                for (int k = i; k < n; k++) {
                    dataset[count].a[j][k] = dataset[count].a[j][k] - temp *
dataset[count].a[i][k];
                }
            }
        }
    }
}
```

2.1.2 SSE 编程消元

使用 SSE 指令集， 将多个元素同时进行寄存器加载和存储器存值等操作， 使用向量化计算。 因为 _mm_loadu_ps 同时加载四个单精度浮点数值，故循环每次跳跃四个元素，算法时间复杂度 $O(n) = (n^3) / 4$。

```
// sse消元
void sseGauss(Matrix* dataset) {
    for (int count = 0; count < 50; count++) {
        __m128 t1, t2, t3, t4;
        for (int k = 0; k < n; k++) {
            float tmp[4] = { dataset[count].a[k][k], dataset[count].a[k][k],
dataset[count].a[k][k], dataset[count].a[k][k] };
            t1 = _mm_loadu_ps(tmp);
            for (int j = n - 4; j >= k; j -= 4) {
                t2 = _mm_loadu_ps(dataset[count].a[k] + j);
                t3 = _mm_div_ps(t2, t1);
                _mm_storeu_ps(dataset[count].atemp[k] + j, t3);
            }
            if (k % 4 != (n % 4)) {
                for (int j = k; j % 4 != (n % 4); j++) {
                    dataset[count].atemp[k][j] = dataset[count].a[k][j] / tmp[0];
                }
            }
            for (int j = (n % 4) - 1; j >= 0; j--) {
                dataset[count].atemp[k][j] = dataset[count].a[k][j] / tmp[0];
            }
            for (int i = k + 1; i < n; i++) {
                float tmp[4] = { dataset[count].a[i][k], dataset[count].a[i][k],
dataset[count].a[i][k], dataset[count].a[i][k] };
```

```
                t1 = _mm_loadu_ps(tmp);
                for (int j = n - 4; j > k; j -= 4) {
                    t2 = _mm_loadu_ps(dataset[count].a[i] + j);
                    t3 = _mm_loadu_ps(dataset[count].atemp[k] + j);
                    t4 = _mm_sub_ps(t2, _mm_mul_ps(t1, t3));
                    _mm_storeu_ps(dataset[count].a[i] + j, t4);
                }
                for (int j = k + 1; j % 4 != (n % 4); j++) {
                    dataset[count].a[i][j] = dataset[count].a[i][j] - dataset[count].a[i]
[k] * dataset[count].atemp[k][j];
                }
                dataset[count].a[i][k] = 0;
            }
        }
    }
}
```

2.2 回代求解

2.2.1 串行实现回代求解

将得到的上三角矩阵，从第 n 行开始，倒序回代到前面的每一行，即可求解 Xn 到 X1 的值。算法的时间复杂度为 $O(n) = n^2$。

```
// serial串行回代
void serialBack(Matrix* dataset) {
    for (int count = 0; count < 50; count++) {
        for (int i = n - 1; i >= 0; i--) {
            float sum = 0;
            for (int j = n - 1; j > i; j--) {
                sum += dataset[count].a[i][j] * dataset[count].x[j];
            }
            dataset[count].x[i] = (dataset[count].b[i] - sum) / dataset[count].a[i][i];
        }
    }
}
```

2.2.2 SSE 编程回代求解

使用 SSE 指令集， 将多个元素同时进行寄存器加载和存储器存值等操作， 使用向量化计算。 因为 _mm_loadu_ps 同时加载四个单精度浮点数值，故循环每次跳跃四个元素，算法时间复杂度 $O(n) = (n^2) / 4$。

```
// sse回代
void sseBack(Matrix* dataset) {
    __m128 t1, t2, sumSSE;
    float sum;
    for (int count = 0; count < 50; count++) {
        //先处理后四个解，以使sse并行计算能够启动
        for (int i = n - 1; i >= 0; i--) {
            if ((n - 1 - i) == 4)break;
            float sum = 0;
            for (int j = n - 1; j > i; j--) {
                sum += dataset[count].a[i][j] * dataset[count].x[j];
            }
            dataset[count].x[i] = (dataset[count].b[i] - sum) / dataset[count].a[i][i];
        }
        //向量化计算
        for (int i = n - 5; i >= 0; i--) {
            sumSSE = _mm_setzero_ps();
            sum = 0;
            int forSerial = (n - i - 1) % 4;
            for (int j = i + forSerial + 1; j <= n - 4; j += 4) {
                t1 = _mm_loadu_ps(dataset[count].a[i] + j);
                t2 = _mm_loadu_ps(dataset[count].x + j);
                t1 = _mm_mul_ps(t1, t2);
                sumSSE = _mm_add_ps(sumSSE, t1);
            }
            sumSSE = _mm_hadd_ps(sumSSE, sumSSE);
            sumSSE = _mm_hadd_ps(sumSSE, sumSSE);
            _mm_store_ss(&sum, sumSSE);

            for (int j = i + 1; j <= i + forSerial; j++) {
                sum += dataset[count].a[i][j] * dataset[count].x[j];
            }

            dataset[count].x[i] = (dataset[count].b[i] - sum) / dataset[count].a[i][i];
        }
    }
}
```

2.3 实验细节设计

1、 计时：

a) 计时方法：参考课件示例代码中的计时方法，使用QueryPerformance系列API；

b) 计时精度：即使精度最高的计时机制，也可能测量不出太快的计算过程。如何解决？将计算过程重复多次，足以匹配计时精度，再计算平均时间。——将每一个计算过程重复 50 次，累积时间后除以 50 计算平均时间。

```
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

        cout <<"两种高斯消元方法运行时间:"<< endl;
        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //普通的高斯方法消元
        serialGauss(dataset);
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        time[0][n/50] = ((tail - head) * 1000.0 / freq)/50;
        cout << time[0][n/50] << " ";

        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //sse的高斯方法消元
        sseGauss(dataset);
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        time[1][n/50] = ((tail - head) * 1000.0 / freq)/50;
        cout << time[1][n/50] << " "<< endl;

        cout <<"两种回代求解方法运行时间:"<< endl;
        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //普通的回代求解
        serialBack(dataset);
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        time[2][n/50] = ((tail - head) * 1000.0 / freq)/50;
        cout << time[2][n/50] << " ";

        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        //sse的回代求解
        sseBack(dataset);
        QueryPerformanceCounter((LARGE_INTEGER*)&tail);
        time[3][n/50] = ((tail - head) * 1000.0 / freq)/50;
        cout << time[3][n/50] << " "<< endl;
```

2、 实验设计：

a) 实验数据：矩阵大小可自定，也可测试不同矩阵规模，由小至大，随机生成矩阵元素值——矩阵规模从 0 到 1000，间隔为 50；且每个规模下由随机数组成的 50 个矩阵样本。

```
for(n=0;n<=1000;n+=50){
        init(dataset);
  //实验内容
    }
```

```
// 随机生成矩阵
void init(Matrix* dataset) {
    srand(static_cast <unsigned> (time(0)));
    for (int i = 0; i < 50; i++) {
        for (int j = 0; j < n; j++) {
            dataset[i].x[j] = static_cast <float> (rand()) / (static_cast <float>(RAND_MAX
/ 1000));
            for (int k = 0; k < n; k++) {
                dataset[i].a[j][k] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (1000 - 1)));
                dataset[i].atemp[j][k] = dataset[i].a[j][k];
            }
        }
        for (int j = 0; j < n; j++) {
            dataset[i].b[j] = 0;
            for (int k = 0; k < n; k++) {
                dataset[i].b[j] += dataset[i].a[j][k] * dataset[i].x[k];
            }
        }
    }

}
```

b) 实验方法：为降低误差，每个测试应重复多次——累积计算这 50 个矩阵乘法后计算平均执行时间；执行 3 次，获得 3 组各规模下的运行时间，再计算平均时间以避免误差。

```
int times=3;

    while(times>0){
            cout<<"第"<<4-times<<"次测试"<<endl;
      //实验内容
      times--;
    }
```

3、 报告撰写:

a) 有清晰的结果呈现，可以结合图表数据分析各算法性能。

2.4 实验数据分析

过程: 进行了3组的实验，每组都跑了矩阵规模为0-1000，间隔50的实验，由系统生成随机数随机生成50个相应规模的矩阵进行乘法，累积时间后除以 50 计算该矩阵规模下，该算法的矩阵乘法的平均时间。

2.4.1 串行算法消元

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.80E-05	2.00E-05	2.20E-05	2.33333E-05
50	0.044986	0.054272	0.051202	0.050153333
100	0.370964	0.44176	0.364534	0.392419333
150	1.23583	1.15647	1.1747	1.189
200	2.61089	2.79301	2.56762	2.657173333
250	4.70531	5.04194	4.82301	4.856753333
300	8.7838	8.08104	8.18566	8.350166667
350	13.4449	13.6555	13.2987	13.46636667
400	20.1758	18.3712	18.2119	18.91963333
450	25.5433	26.4326	27.3221	26.43266667
500	34.2218	35.4526	34.6036	34.75933333
550	45.6752	46.6278	46.1508	46.15126667
600	58.6405	59.3669	59.9186	59.30866667
650	74.9429	74.6475	74.9639	74.85143333
700	93.3381	94.683	93.0036	93.6749
750	111.059	113.825	112.328	112.404
800	134.408	138.274	133.564	135.4153333
850	157.547	166.96	162	162.169
900	193.341	187.718	192.238	191.099
950	225.494	228.524	227.588	227.202
1000	267.511	263.451	257.224	262.7286667

2.4.2 SSE编程消元

规模（矩阵大小）	第一次	第二次	第三次	平均
0	2.60E-05	2.60E-05	2.20E-05	2.46667E-05
50	0.038672	0.036918	0.043424	0.039671333
100	0.2368	0.302468	0.243676	0.260981333
150	0.797782	0.736906	0.764216	0.766301333
200	1.67078	1.6447	1.64437	1.653283333
250	3.11083	3.08422	3.208	3.13435
300	6.02104	4.95044	5.01425	5.328576667
350	9.26124	8.57034	7.91154	8.58104
400	11.3803	11.4402	11.9058	11.57543333
450	16.2325	16.3302	16.6409	16.4012
500	21.2615	21.1885	21.0956	21.18186667
550	27.0216	28.3916	28.1422	27.8518
600	32.1536	33.519	33.2873	32.98663333
650	40.7815	41.3972	41.6979	41.2922
700	48.4493	49.9913	48.265	48.90186667
750	58.1581	60.4484	59.8194	59.4753
800	69.8736	70.9084	69.786	70.18933333
850	85.321	83.9452	85.9964	85.08753333
900	98.0352	98.3875	97.0695	97.83073333
950	113.601	116.047	113.271	114.3063333
1000	130.615	162.957	133.617	142.3963333

2.4.3 串行实现回代

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.00E-05	2.60E-05	2.20E-05	2.26667E-05
50	0.003098	0.005836	0.00597	0.004968
100	0.014454	0.016014	0.015974	0.015480667
150	0.0327	0.036728	0.040076	0.036501333
200	0.060394	0.061598	0.06114	0.061044
250	0.11165	0.108678	0.0878	0.102709333
300	0.12476	0.118918	0.11883	0.120836
350	0.24746	0.15767	0.153192	0.186107333
400	0.198218	0.205	0.204626	0.202614667
450	0.298176	0.2235	0.214262	0.245312667
500	0.292834	0.293954	0.271248	0.286012
550	0.305914	0.323474	0.327484	0.318957333
600	0.387614	0.398194	0.40061	0.395472667
650	0.536058	0.390486	0.380822	0.435788667
700	0.506004	0.46404	0.474194	0.481412667
750	0.558378	0.638718	0.550968	0.582688
800	0.661564	0.695046	0.740282	0.698964
850	0.634956	0.58996	0.556838	0.593918
900	0.60959	0.621592	0.588348	0.60651
950	0.72817	0.751534	0.66977	0.716491333
1000	0.750552	0.708058	0.682622	0.713744

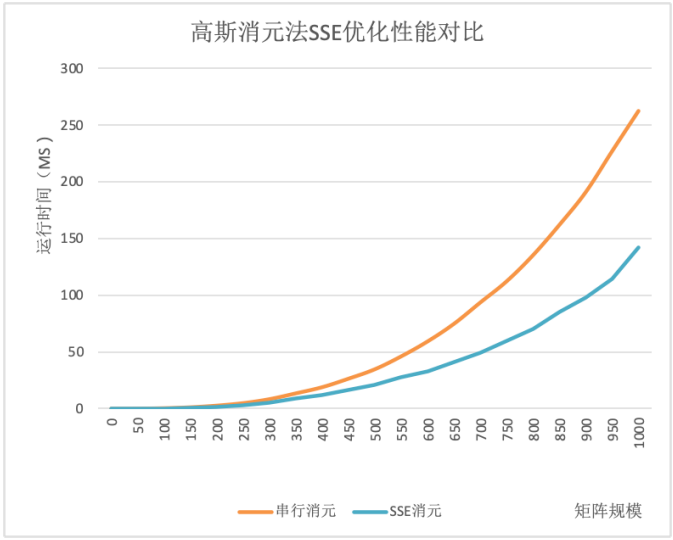
2.4.4 SSE编程回代

规模（矩阵大小）	第一次时间	第二次时间	第三次时间	平均时间
0	2.40E-05	2.20E-05	2.20E-05	2.26667E-05
50	0.001406	0.003576	0.002304	0.002428667
100	0.008352	0.008626	0.00701	0.007996
150	0.018166	0.023374	0.020396	0.020645333
200	0.034096	0.048662	0.035684	0.039480667
250	0.052964	0.054706	0.049814	0.052494667
300	0.089168	0.070566	0.076636	0.07879
350	0.092936	0.100706	0.085874	0.093172
400	0.118176	0.112304	0.112078	0.114186
450	0.122914	0.12593	0.12588	0.124908
500	0.168084	0.166776	0.146128	0.160329333
550	0.171374	0.168728	0.184956	0.175019333
600	0.214382	0.210796	0.21166	0.212279333
650	0.239592	0.229472	0.216618	0.228560667
700	0.254332	0.257348	0.263084	0.258254667
750	0.27319	0.315144	0.275118	0.287817333
800	0.31794	0.325818	0.294158	0.312638667
850	0.363094	0.323826	0.353248	0.346722667
900	0.419732	0.396706	0.396334	0.404257333
950	0.42216	0.405334	0.380126	0.40254
1000	0.450856	0.433334	0.54037	0.474853333

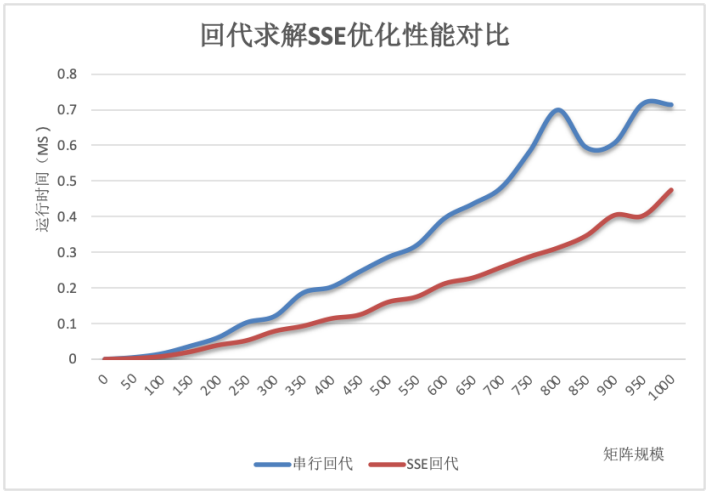
2.5 对比分析

实现方法	高斯串行（ms）	高斯SSE（ms）	串行回代（ms）	SSE回代（ms）
规模（矩阵大小）	平均时间	平均时间	平均时间	平均时间
0	2.33333E-05	2.46667E-05	2.27E-05	2.27E-05
50	0.050153333	0.039671333	0.004968	0.002429
100	0.392419333	0.260981333	0.015481	0.007996
150	1.189	0.766301333	0.036501	0.020645
200	2.657173333	1.653283333	0.061044	0.039481
250	4.856753333	3.13435	0.102709	0.052495
300	8.350166667	5.328576667	0.120836	0.07879
350	13.46636667	8.58104	0.186107	0.093172
400	18.91963333	11.57543333	0.202615	0.114186
450	26.43266667	16.4012	0.245313	0.124908
500	34.75933333	21.18186667	0.286012	0.160329
550	46.15126667	27.8518	0.318957	0.175019
600	59.30866667	32.98663333	0.395473	0.212279
650	74.85143333	41.2922	0.435789	0.228561
700	93.6749	48.90186667	0.481413	0.258255
750	112.404	59.4753	0.582688	0.287817
800	135.4153333	70.18933333	0.698964	0.312639
850	162.169	85.08753333	0.593918	0.346723
900	191.099	97.83073333	0.60651	0.404257
950	227.202	114.3063333	0.716491	0.40254
1000	262.7286667	142.3963333	0.713744	0.474853

在相同规模的矩阵下，高斯求解与回代求解算法的图表表现如下：



高斯算法



回代求解

大体上可以看到sse的算法比串行的普通算法好。

2.6 问题回答

a）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：无论是对于回代求解还是高斯消元，在一开始，各个算法之间的时间差距并不明显。随着矩阵规模的增加，串行算法时间增加越来越快，SSE 编程增加相对较为缓慢；且各个算法之间的时间差距在逐渐扩大。

b）消元过程中采用向量编程的的性能提升情况如何：消元过程中，在规模较小的情况下 SSE 编程的加速比接近 2，之后随着矩阵规模的增大，加速比也在增大。产生这种情况的原因可能是一开始，在规模不大时 SSE 向量化和从向量重新化成浮点数等准备过程消耗了一些额外时间，导致性能提升收到限制。随着矩阵规模的增大，时间消耗主要由计算产生，故加速比增大。但从理论来讲，加速比不会等于 4，只会接近 4。

c）回代过程向量化编程性能提升情况：回代求解的过程也能进行向量化计算。与高斯消元类似，我们也可以看出 SSE 编程对计算性能的提升，以及随着矩阵规模的增大，这种性能提升越来越显著。

源码和实验数据、根据数据生成的图表结果见压缩包附件（表格中有两个 sheet，分别对应两个实验的原始数据图表）