《并行程序与设计》实验报告

(2023~2024 学年第一学期)

实验名称：SSE 编程练习

学　　院：软件学院

姓　　名：陈高楠

学　　号：2112966

指导老师：孙永谦

2023 年 11 月 6 日

目录

实验一 矩阵乘法的优化

1 问题描述

本次实验分别使用串行算法、Cache 优化算法、SSE 编程和分片策略算法实现了矩阵乘法运算。比较在不同策略算法下矩阵乘法运算的时间差异。本次实验的矩阵的元素值为生成的随机数。每次随机生成的两个矩阵都需要经过四种策略算法的计算，共生成二十次随机矩阵，并记录每次运行时间和二十次运算的平均运行时间。计时采用 QueryPerformance。

2 算法设计与分析

2.1 串行算法

串行算法即不采取任何优化的操作，而直接用矩阵乘法的定义进行运算。矩阵乘法即对矩阵 A 的每一行的元素，矩阵 B 的每一列的元素都要与之相乘，然后相加从而得到结果。所以需要先遍历矩阵 A 一行中的元素，再遍历矩阵 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 $O(n) = n^3$

具体代码如下：

```
double mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) {
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    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];
```

```
                }
            }
        }
        QueryPerformanceCounter(&endTime);

        double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)freq.QuadPart;
        cout << "串行算法耗时： " << time  << "s" << endl;
        return time;

}
```

2.2 Cache 优化算法

矩阵在进行运算时，当读取的时候按行读取时，该行会进入 cache，缩短数据读取的时间（C 语言为行主序）。而当读取的时候按列读取时，由于同一列的数据并未同时进入 cache 中，这也导致在读取同一列的数据时，需要从内存而不是从缓存中读取数据，从而需要更长的寻址时间。在矩阵乘法时，矩阵 A 是按行读取的，读取速度快。但是矩阵 B 是按列读取的，读取速度慢 (因为 cache 命中率不高)。所以我们可以优化矩阵 B 的读取，将 B 转置，使得矩阵 B 在进行乘法时按行读取，减少缓存未命中的次数，利用 cache 的高速读取特性来加快运行速度，节省寻址时间。因为算法仍然为三层嵌套，所以算法时间复杂度仍为 $O(n) = n^3$

具体代码如下：

```
double trans_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) { // Cache优化 O(n) = n^3
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
```

```
    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)
            swap(b[i][j], b[j][i]);

    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
   freq.QuadPart;
    cout << "Cache优化耗时: " << time  << "s" << endl;
    return time ;
}
```

2.3 SSE 编程

SSE 编程能够多个元素同时加载进行并行运算，提高了运行的效率，使得运行时间缩短。由于四个元素同时进行并行运算，时间复杂度为 $O(n) = (n^3)/4$

具体代码如下：

```
double sse_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) {
 //SSE版本 O(n) = (n^3)/4
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    __m128 t1, t2, sum;  //__m128 == float
```

```
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            sum = _mm_setzero_ps(); //Initialize
            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];
            }
        }
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);

    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)freq.QuadPart;
    cout << "SSE版本耗时: " << time << "s" << endl;
    return time ;
}
```

2.4 分片策略

分片策略利用分块的思想，将矩阵每 T 个分为一个小矩阵块，每一个小矩阵块并行执行矩阵的乘法，时间复杂度 $O(n) = (n^3)$

具体代码如下：

```
double sse_tile(int n, float a[][maxN], float b[][maxN], float c[][maxN], int T) {
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    __m128 t1, t2, sum;
    float t;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
    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 = T-4; k >= 0; k -= 4) {
                            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;
                                for (int k = (T % 4) - 1; k ≥ 0; --k) { // handle the last n%4elements
                                    c[r * T + i][q * T + j] += a[r * T + i][p * T + k] * b[q * T + j][p * T + k];
                                }

                            }
                    }
                }
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < i; ++j)
                swap(b[i][j], b[j][i]);

        QueryPerformanceCounter(&endTime);

        double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)freq.QuadPart;
        cout << "分片策略耗时: " << time << "s" << endl;
        return time;
}
```

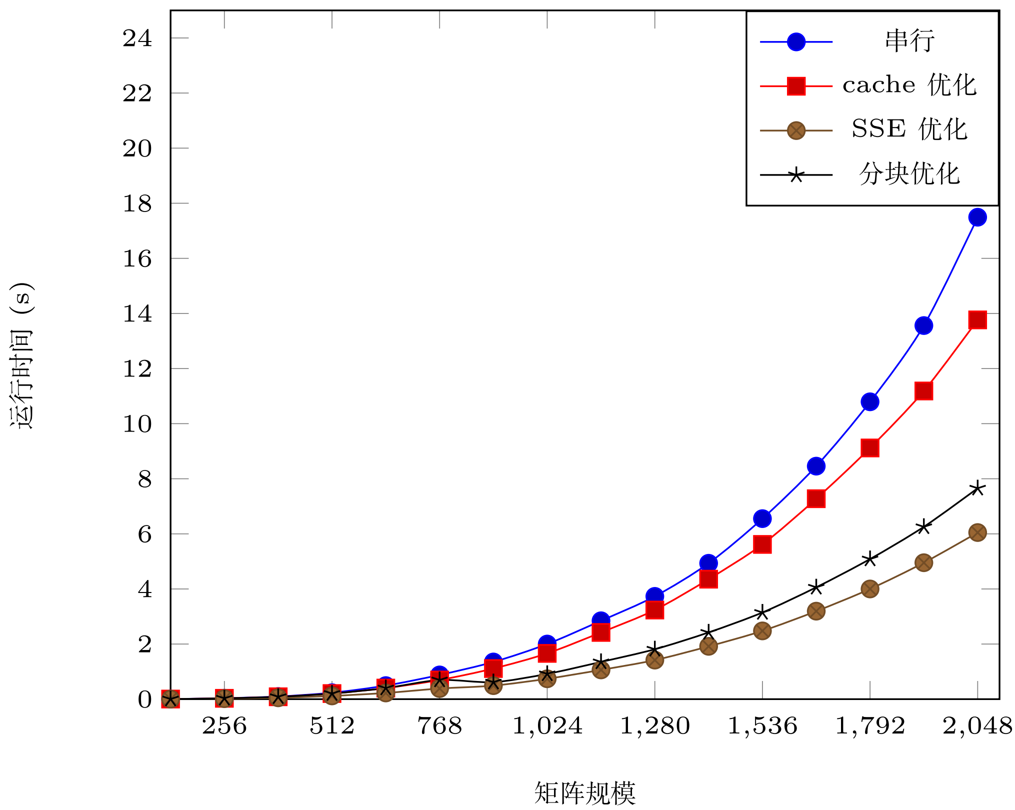
3 结果统计与分析

3.1 不同策略下矩阵运算运行时间的统计

以下为不同策略下矩阵运算运行时间的统计，每种矩阵规模下，每种策略均运行二十次取平均值，减少误差。

矩阵规模	串行	Cache 优化	SSE 并行	矩阵分块并行
128*128	0.0042058s	0.00387835s	0.00162774s	0.00206567s
256*256	0.0342032s	0.0307152s	0.0126819s	0.0175909s
384*384	0.0963745s	0.0866515s	0.037384s	0.0489838s
512*512	0.23331s	0.20333s	0.0897747s	0.117839s
640*640	0.491899s	0.402294s	0.177279s	0.224926s
768*768	0.879674s	0.695461s	0.303479s	0.385212s
896*896	1.3495s	1.11125s	0.490818s	0.617381s
1024*1024	2.00046s	1.66126s	0.739852s	0.916984s
1152*1152	2.84547s	2.42189s	1.06165s	1.35163s
1280*1280	3.73081s	3.23532s	1.4171s	1.81112s
1408*1408	4.93209s	4.35484s	1.91689s	2.42228s
1536*1536	6.55257s	5.61627s	2.47748s	3.14573s
1664*1664	8.4577s	7.27544s	3.1937s	4.06025s
1792*1792	10.793s	9.11793s	3.9997s	5.08599s
1920*1920	13.5604s	11.186s	4.95235s	6.25561s
2048*2048	17.4928s	13.7664s	6.04739s	7.65593

表 1.1: 不同优化方法下的运行时间



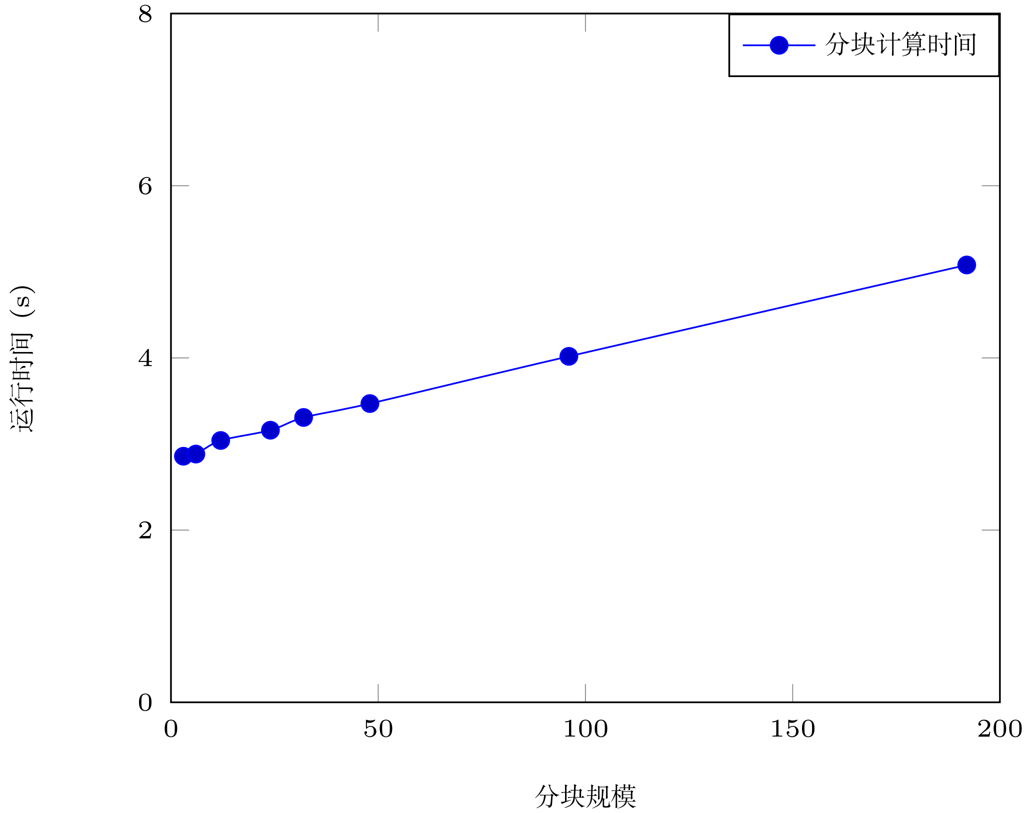
通过图表可知，串行算法所耗费的时间是最多的，因为他并没有做出什么优化。而 Cache 优化算法因为采取了 Cache 优化，因此耗费时间比串行时间略少，但提升幅度在矩阵规模较小时并不是很明显，一直到矩阵规模为 768*768 时才开始有差距。SSE 编程在 Cache 优化的

基础上采用了并行计算，极大减少了运行时间，可以从图中看到比串行和 Cache 优化的运行时间少很多，而且在矩阵规模并不大的时候就能体现。而分片策略尽管理论上能提高运行速度，但实际上由于缓存大小的影响，分片的片数，以及分片时多出的分片操作，导致分片策略运行时间甚至比 SSE 编程的还高。

因此，在四种矩阵乘法方法中，性能比较：SSE 编程 > 分片策略 >Cache 优化 > 串行算法。

3.2 分片大小的影响

矩阵大小固定时，测出矩阵乘法在不同分块下运行时间的比较。(测试的分块均能存进一级缓存中) 测得不同分块下运行时间变化曲线如下：



可以看出当分块的矩阵越大时，运行时间也会越来越大。分析：图中的分块都是可以装进 L1 缓存的，但是分块矩阵越大，cache 命中所需的时间越长，这也导致了运行时间的差异。

4 实验结论及心得体会

实验结论: 串行算法的耗费时间最多；Cache 优化对串行算法进行了优化，提高了寻址的速率，缩短了运行时间；SSE 编程使用了并行算法，极大缩短了运行时间；而分片策略因为分块多出来的时间损耗，总运行时间并不如 SSE 编程。分片规模对分片的性能提升也有影响，分片时的分片矩阵不宜过大。

心得体会：本次实验学习了四种矩阵乘法运算的方法，并初步学习了 SSE 的基本语法以及使用方法，对并行程序设计有了更深的认识。

实验二　高斯消元法 SSE/AVX 并行化

1　问题描述

首先熟悉高斯消元法解线性方程组的过程，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。

本次实验将采用并行的方法来解决高斯消元的问题，并通过三个方面来对高斯消元进行探讨。计时采用 QueryPerformance。

2　算法设计与分析

2.1　高斯消元法介绍

消元过程：将 Ax=b 按照从上至下、从左至右的顺序化为上三角方程组，中间过程不对矩阵进行交换，主要步骤如下：

Step1: 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。公式如下：

$$a_{11}^{(1)} \neq 0, l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \quad (i = 2 : n), \text{第}i\text{行} + (-l_{i1}) \times \text{第 1 行} \quad (i = 2 : n)$$

Step2: 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。公式如下：

$$l_{i2}=\frac{a_{i2}^{(2)}}{a_{22}^{(2)}}\quad(i=3:n),\text{第}i\text{行加上}(-l_{i2})\times\text{第 2 行}\quad(i=3:n)$$

Step K: 按照上述方法，当第 k 步运算时，公式为：

$$a_{kk}^{(k)}\neq 0,l_{ik}=\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}\quad(i=k+1\to n),\text{第}i\text{行加上}(-l_{ik})\times\text{第}k\text{行}\quad(i=k+1\to n)$$

Step n-1: 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，如下：

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ a_{22}^{(2)} & & \cdots & a_{2n}^{(2)} \\ \vdots & & \ddots & \vdots \\ a_{nn}^{(n)} & & & x_n \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}=\begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{bmatrix}$$

回代过程：从第 n 行开始，倒序回代前面的行中，即可求解 x_1 到 x_n 的值。

2.2 串行算法

串行高斯消元的方法主要分成两步，第一步就是把要当被减行的行进行放大或缩小使得该行第一个不为 0 的数为 1（该步骤可以省去，只是后面乘的系数不一样），第二步便是遍历被减行（第 k 行）下面的所有行（k+1 行到 n 行），每一行都减去被减行与一个系数的乘积，即假设现在是 k+1 行 k+1 列，该数减去被减行对应列的值乘以一个系数的乘积，即第 k 行 k+1 列的值乘以 k+1 行 k 列的值，即为更新的值，以此类推。最后再把 k+1 行 k 列设置为 0。

如果省略第一步，那么系数则不为 k+1 行 k 列的值，而是 k+1 行 k 列的值除以 k 行 k 列的值。注意在矩阵遍历的时候，矩阵为 n*（n+1）的规模，因为构成的矩阵为增广矩阵。由于循环最多嵌套三层，因此时间复杂度为 O(n) = n^3

具体代码如下：

```
double LU(int n, float a[][maxN]){
      LARGE_INTEGER freq;
```

```
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    for (int i = 0; i < n - 1; i++) //对每一行
        {
            for(int j=i+1;j<=n;j++)//对这一行的每一个数
            {
                a[i][j]=a[i][j]/a[i][i];
            }
            a[i][i]=1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
         {
            float tem = a[j][i] / a[i][i];
            for (int k = i + 1; k <= n; k++) //这个是列, 列是n+1个 (有b)
            {
                a[j][k] -= a[i][k] * tem;
            }
            a[j][i] = 0.00;
        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
    cout << "普通方法耗时: " << time << "s" << endl;
    return time;


}
```

2.3 串行回代求解

这里采用了最普通的求解方法，即从下到上进行回代求解。时间复杂度 $O(n) = n^2$

具体代码如下:

```
double generation(int n,float a[][maxN],float x[])
{
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    for (int i = n - 1; i >= 0; i--) {
        x[i] = a[i][n] / a[i][i];
        for (int j = i - 1; j >= 0; j--) {
            a[j][n] -= x[i] * a[j][i];
        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
    cout << "普通方法回代: " << time << "s" << endl;
    return time;

}
```

2.4 并行高斯消元 SSE 编程

在高斯消元的过程中，对 4 个计算进行并行处理，提高运行速率，时间复杂度 $O(n) = (n^3)$。这里采用 SSE 编程。

具体代码如下:

```
double LU_SSE(int n, float a[][maxN]){
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
        __m128 t1, t2, sub,tem2;
```

```
for (int i = 0; i ≤ n - 1; i++) //对每一行
        {
            for(int j=i+1;j≤n;j++)//对这一行的每一个数
            {
                a[i][j]=a[i][j]/a[i][i];
            }
            a[i][i]=1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
         {
            float tem = a[j][i] / a[i][i];
            tem2=_mm_set1_ps(tem);
            for (int k = i+1; k ≤n; k+=4) //这个是列, 列是n+1个 (有b)

            {     if(k+3>n)break;
                t1 = _mm_loadu_ps(a[i] + k);
                t2 = _mm_loadu_ps(a[j] + k);
                sub = _mm_sub_ps(t2, _mm_mul_ps(t1, tem2));
                _mm_storeu_ps(a[j] + k, sub);
            }
            for (int k =n-(n-i)%4+1; k ≤n; k+=1) {
                a[j][k] -= a[i][k] * tem;
            }
            a[j][i] = 0.00;
        }
    }
    QueryPerformanceCounter(&endTime);
    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "SSE耗时: " << time << "s" << endl;
    return time;
}
```

2.5 并行高斯消元 AVX 编程

在高斯消元的过程中，对 8 个计算进行并行处理，提高运行速率，时间复杂度 $O(n) = (n^3)$。这里采用 AVX。具体代码如下：

```
double LU_AVX(int n, float a[][maxN]){
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
        __m256 t1, t2, sub,tem2;
for (int i = 0; i ≤ n - 1; i++) //对每一行
        {
            for(int j=i+1;j≤n;j++)//对这一行的每一个数
            {
                a[i][j]=a[i][j]/a[i][i];
            }
            a[i][i]=1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
         {
            float tem = a[j][i] / a[i][i];
            tem2=_mm256_set1_ps(tem);
            for (int k = i+1; k ≤n; k+=8) //这个是列, 列是n+1个 (有b)

            {     if(k+7>n)break;
                t1 = _mm256_loadu_ps(a[i] + k);
                t2 = _mm256_loadu_ps(a[j] + k);
                sub = _mm256_sub_ps(t2, _mm256_mul_ps(t1, tem2));
                _mm256_storeu_ps(a[j] + k, sub);
            }
            for (int k =n-(n-i)%8+1; k ≤n; k+=1) {
                a[j][k] -= a[i][k] * tem;
            }
            a[j][i] = 0.00;
        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
```

```
    cout << "AVX高斯消元耗时： " << time << "s" << endl;
    return time;
}
```

2.6 并行回代优化 SSE 编程

这里采用并行的方法对回代进行优化，采取 SSE 编程。时间复杂度 $O(n) = n^2$

具体代码如下：

```
double generation_SSE(int n, float a[][maxN], float x[]) {

    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m128 t1, t2, t3, sub, tem;
    //转置


    for (int i = n - 1; i ≥ 0; i--) {
        x[i] = a[i][n];
        for (int j = i + 1; j < n; j += 4) {
            if (j + 3 ≥ n)break;
            t1 = _mm_load_ps(a[i] + j);
            t2 = _mm_load_ps(x + j);
            t3 = _mm_mul_ps(t1, t2);
            t3 = _mm_hadd_ps(t3, t3);
            t3 = _mm_hadd_ps(t3, t3);
            x[i] -= _mm_cvtss_f32(t3);

        }
        for (int j = n - (n - i-1) % 4 ; j < n; j++) {
            x[i] -= a[i][j] * x[j];
        }
```

```
    }
    QueryPerformanceCounter(&endTime);
    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "SSE方法优化回代：" << time << "s" << endl;
    return time;
}
```

2.7 并行回代优化 AVX 编程

这里采用并行的方法对回代进行优化，采取 AVX 编程。时间复杂度 $O(n) = n^2$

具体代码如下：

```
double generation_AVX(int n, float a[][maxN], float x[]) {

    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m256 t1, t2, t3,tem;


    for (int i = n - 1; i ≥ 0; i--) {
        x[i] = a[i][n];
        for (int j = i + 1; j < n; j += 8) {
            if (j + 7 ≥ n)break;
            t1 = _mm256_load_ps(a[i] + j);
            t2 = _mm256_load_ps(x + j);
            t3 = _mm256_mul_ps(t1, t2);
            t3 = _mm256_hadd_ps(t3, t3);
            t3 = _mm256_hadd_ps(t3, t3);
            x[i] -= (_mm256_cvtss_f32(t3) + _mm256_cvtss_f32(...
                _mm256_permute2f128_ps(t3, t3, 1)));
        }
```

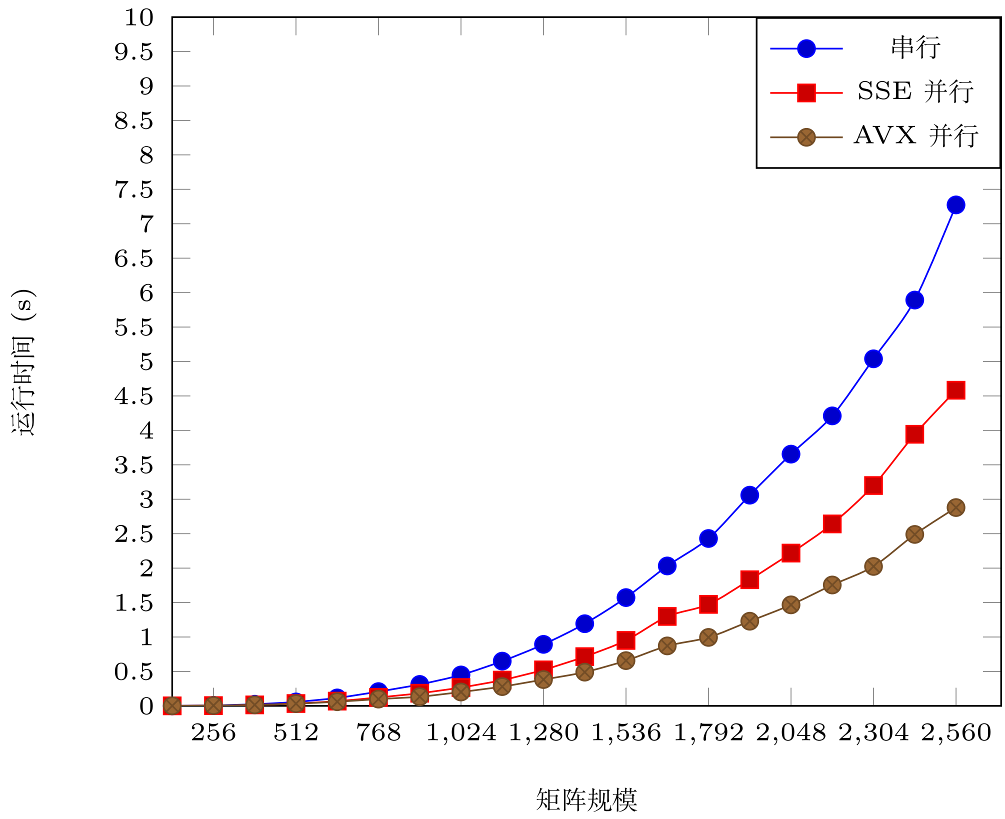
```
            for (int j = n - (n - i - 1) % 8; j < n; j++) {
                x[i] -= a[i][j] * x[j];
            }
        }
        QueryPerformanceCounter(&endTime);
        double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
        cout << "AVX方法优化回代: " << time << "s" << endl;
        return time;
}
```

3 问题讨论与分析

3.1 消元过程中采用向量编程的的性能提升情况如何

这里采用了 QueryPerformance 进行计时，通过生成随机矩阵来对比采用向量编程的的性能提升情况。矩阵的规模从从 128*128 到 2560*2560。

以下是统计的结果。



矩阵规模	串行	SSE 编程	AVX 编程
128*128	0.000909165	0.000556225	0.0010975
256*256	0.0070659	0.00414377	0.0054567
384*384	0.024183	0.0143697	0.0151507
512*512	0.0574497	0.0340906	0.0328093
640*640	0.115604	0.0688632	0.0600911
768*768	0.207282	0.122675	0.098506
896*896	0.311542	0.183689	0.133205
1024*1024	0.449007	0.266695	0.201336
1152*1152	0.649681	0.374257	0.279803
1280*1280	0.893172	0.522936	0.382136
1408*1408	1.19295	0.715037	0.489673
1536*1536	1.57222	0.950008	0.657101
1664*1664	2.03174	1.29849	0.870898
1792*1792	2.43019	1.47337	0.994015
1920*1920	3.05917	1.83142	1.22919
2048*2048	3.65508	2.21836	1.46685
2176*2176	4.21054	2.64181	1.75525
2304*2304	5.03897	3.19913	2.02437
2432*2432	5.89198	3.94339	2.48919
2560*2560	7.27353	4.58284	2.87923

表 2.1: 高斯消元不同方法下的运行时间

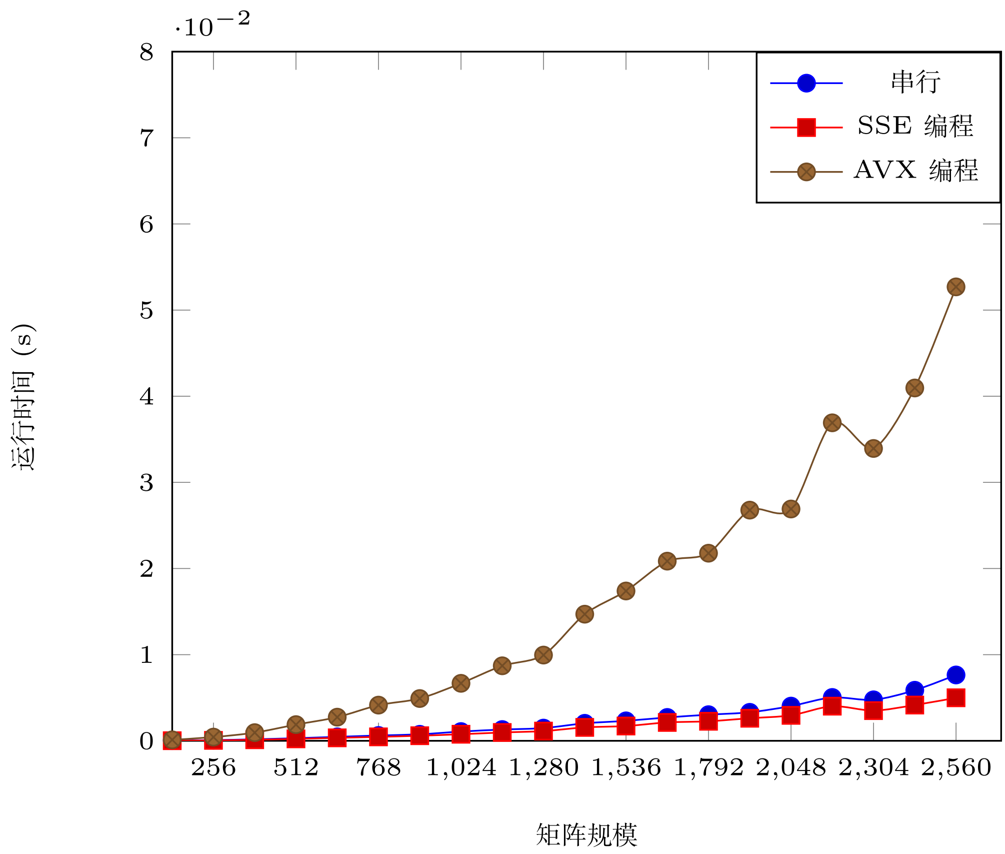
从表中可以分析得到，消元过程中采用向量编程在矩阵规模较小时性能提升并不是很明显，当矩阵规模越来越大时提升性能越来越明显。而在向量编程中，一开始矩阵规模比较小的时候，AVX 编程提升的性能是不如 SSE 编程的，推测是因为矩阵规模较小时，AVX 编程有较多元素无法并行计算，因此耗费时间比 SSE 编程长，而在矩阵规模变得很大时，AVX 编程优于 SSE。

3.2 回代过程可否向量化，有的话性能提升情况如何

回代过程中可以向量化。在回代过程中，求 x[i] 时，需要用 n+1 列对应的数减去已经求出的 x 的解与对应 i 行的值的乘积的和，而 x 的解与对应 i 行的值的乘积的和就可以向量化加快计算。这里展示了普通回代，SSE 回代，AVX 回代的数据。

矩阵规模	串行	SSE 编程	AVX 编程
128*128	2.3485e-05	1.3625e-05	0.00011446
256*256	7.786e-05	5.1745e-05	0.000431025
384*384	0.00017528	0.0001238	0.00094853
512*512	0.000304175	0.00022683	0.0018801
640*640	0.00046013	0.00034479	0.0027453
768*768	0.000619155	0.000458805	0.00414327
896*896	0.0007541	0.000592275	0.00490773
1024*1024	0.00107413	0.00076283	0.00667363
1152*1152	0.00130806	0.00096692	0.00870856
1280*1280	0.00147634	0.00112877	0.00995333
1408*1408	0.0020329	0.00155695	0.0146994
1536*1536	0.00231796	0.00171767	0.0173922
1664*1664	0.0027132	0.00213256	0.0208562
1792*1792	0.00303366	0.00226743	0.0217783
1920*1920	0.00330798	0.00262026	0.0267779
2048*2048	0.00402952	0.00295492	0.0269092
2176*2176	0.00503331	0.00402144	0.0369137
2304*2304	0.0047589	0.00350493	0.0339359
2432*2432	0.00586956	0.00415047	0.0409575
2560*2560	0.00764471	0.00499533	0.052693

表 2.2: 回代不同方法下的运行时间



从数据中可以发现，SSE 编程并行计算所耗费的时间比串行减少了一点，但 AVX 编程进行运算的时间却比串行和 SSE 编程多非常多。经过分析，有以下两种可能：

1. 代码错误。但是代码经过检验后正确性是比较有保证的，在测试数据的时候答案也是正确的。

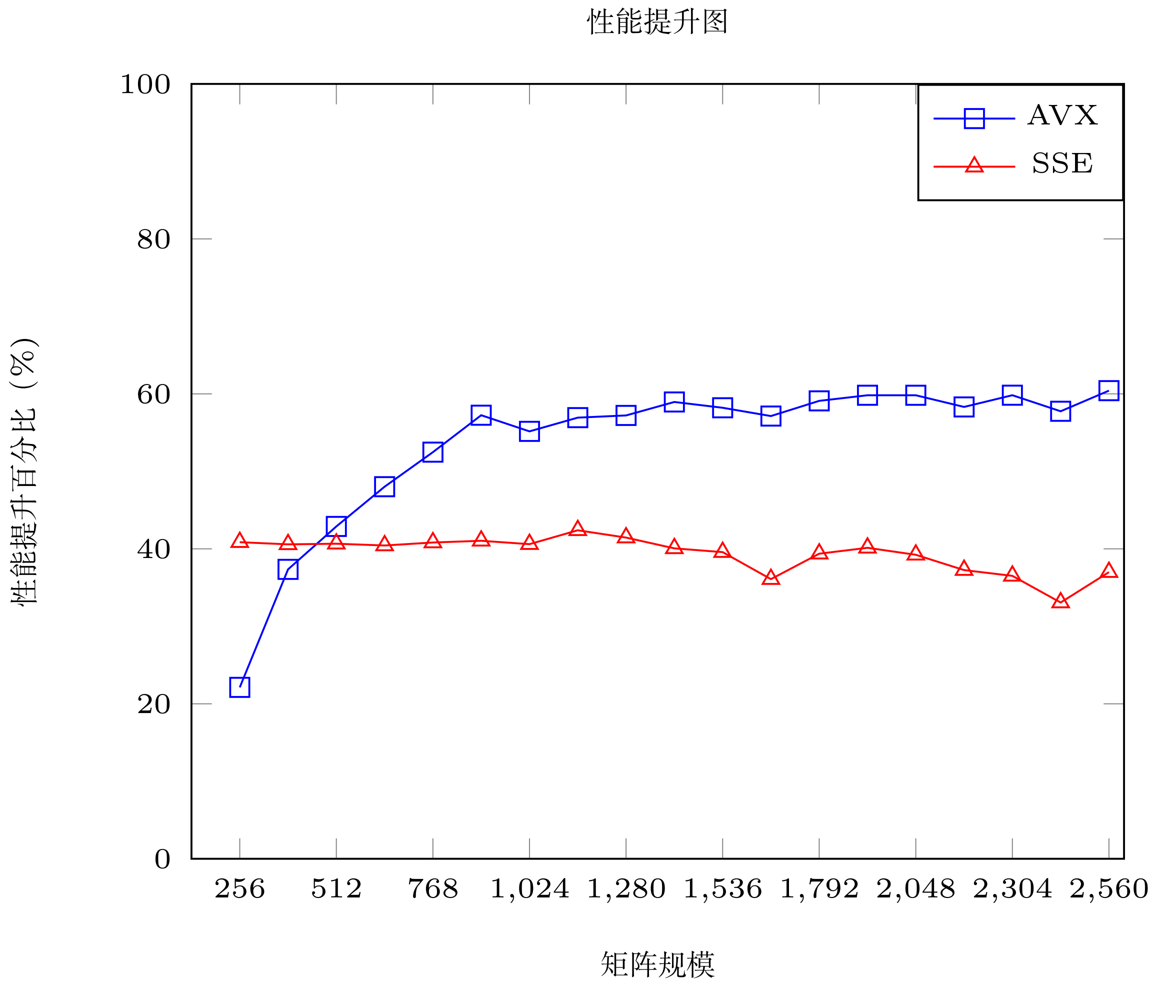
2.AVX 编程在并行的时合并所需操作比 SSE 编程和串行多，且末尾出现冗余的概率也比 SSE 编程和串行概率大。

3.3 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

相同算法对于不同问题规模的性能提升有影响。当矩阵的规模并不大时，SSE,AVX 编程的性能提升影响并不大，甚至出现性能不如串行的情况。(当矩阵规模 128*128 时，串行比 SSE 和 AVX 都快)。而当矩阵规模越来越大时，并行与串行之间时间差越来越大，但 SSE 和 AVX 性能提升比例并不是越来越大，而是在到某一值后变开始波动。性能提升比例变化曲线见下图。

矩阵规模	SSE 编程	AVX 编程
128*128	40.85%	22.12%
256*256	40.57%	22.12%
384*384	40.66%	37.34%
512*512	40.66%	42.89%
640*640	40.43%	48.01%
768*768	40.81%	52.47%
896*896	41.03%	57.24%
1024*1024	40.60%	55.15%
1152*1152	42.39%	56.93%
1280*1280	41.45%	57.21%
1408*1408	40.06%	58.95%
1536*1536	39.57%	58.20%
1664*1664	36.08%	57.13%
1792*1792	39.37%	59.09%
1920*1920	40.13%	59.81%
2048*2048	39.23%	59.81%
2176*2176	37.25%	58.31%
2304*2304	36.51%	59.82%
2432*2432	33.07%	57.75%
2560*2560	36.99%	60.41%

表 2.3: 不同优化方法下的运行时间



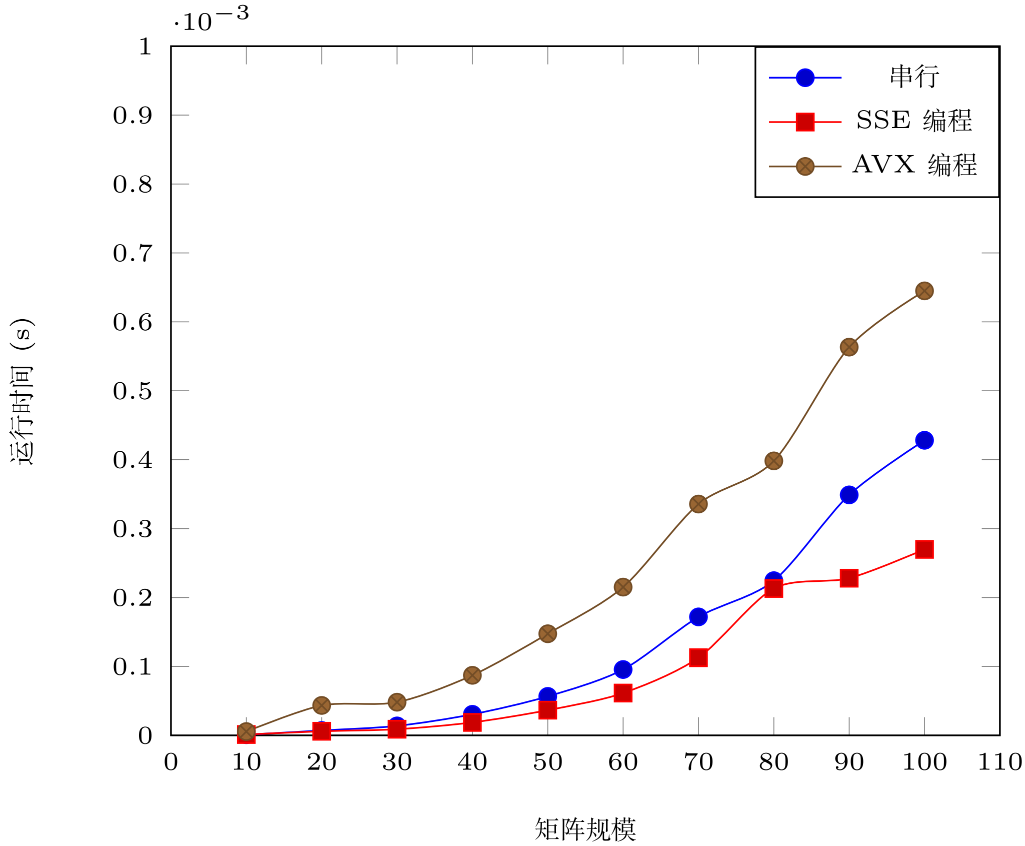
可以看出，SSE 编程性能在矩阵规模较大时能提升百分之四十左右，AVX 编程性能在矩阵规模较大时能提升百分之六十左右。

3.4 SSE 编程和 AVX 编程性能对比

为证明矩阵规模不大的时候,SSE 编程要优于 AVX 编程。统计矩阵规模 10*10 到 100*100 时三种高斯消元方法的计算时间。

矩阵规模	串行	SSE 编程	AVX 编程
10*10	9.5e-07	1.225e-06	5.53e-06
20*20	7.085e-06	5.935e-06	4.3425e-05
30*30	1.3705e-05	8.9e-06	4.8115e-05
40*40	3.052e-05	1.878e-05	8.713e-05
50*50	5.654e-05	3.6565e-05	0.000147415
60*60	9.552e-05	6.136e-05	0.00021511
70*70	0.000171895	0.00011266	0.00033562
80*80	0.00022446	0.00021314	0.000398255
90*90	0.000348925	0.000228055	0.000563335
100*100	0.000428155	0.000269875	0.00064494

表 2.4: 不同优化方法下的运行时间



在矩阵规模不大的时候，SSE 编程要优于 AVX 编程 (甚至串行都比 AVX 编程快)。而当矩阵规模越来越大时，AVX 编程的性能会逐渐超过 SSE 编程。变化曲线详见 3.1。

4 实验结论及心得体会

本次实验学习了高斯消元以及回代的基本步骤，并实现了高斯消元的向量化，对 SSE/AVX 编程有了一个更深刻的认识。

附录

1. 矩阵乘法测试源码

```cpp
#include <iostream>
#include <pmmintrin.h>
#include <stdlib.h>
#include <algorithm>
#include <windows.h>
using namespace std;
const int maxN = 10000;
const int T = 32;
float a[maxN][maxN];
float b[maxN][maxN];
float c[maxN][maxN];
long long head, tail, freq;
int n;
double mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) {
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    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];
            }
        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)freq.QuadPart;
    cout << "串行算法耗时：" << time << "s" << endl;
    return time;
```

```

}
double trans_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) {...
    // Cache优化 O(n) = n^3
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
    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)
            swap(b[i][j], b[j][i]);

    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "Cache优化耗时: " << time << "s" << endl;
    return time;
}
double sse_mul(int n, float a[][maxN], float b[][maxN], float c[][maxN]) { ...
    //SSE版本 O(n) = (n^3)/4
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    __m128 t1, t2, sum;  //__m128 == float
```

```
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            c[i][j] = 0.0;
            sum = _mm_setzero_ps();  //Initialize
            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];
            }
        }
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);

    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)freq.QuadPart;
    cout << "SSE版本耗时: " << time << "s" << endl;
    return time;
}
double sse_tile(int n, float a[][maxN], float b[][maxN], float c[][maxN], int T) {
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);

    __m128 t1, t2, sum;
```

```
    float t;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);
    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 = 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];
            }
        }
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            swap(b[i][j], b[j][i]);

    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
    cout << "分片策略耗时：" << time << "s" << endl;
    return time;
}
int main()
{
    n = 128;
    int guimo = 200;
```

```
    int x = 0;
    double result[100][4];
    for (int i = 0;i < 100;i++) {
        for (int j = 0;j < 4;j++)
            result[i][j] = 0;
    }
    while (n <= 2048)
    {

        cout << "矩阵规模为" << n << "*" << n << "时" << endl;
        x = 0;

            while (x++ < 20)
            {
                cout << "第" << x << "次矩阵乘法" << endl;
                cout << endl;
                for (int i = 0; i < n; i++)
                {
                    for (int j = 0; j < n; j++) {
                        a[i][j] = rand();
                        b[i][j] = rand();
                    }

                }
                result[n / 128][0] += mul(n, a, b, c);
                result[n / 128][1] += trans_mul(n, a, b, c);
                result[n / 128][2] += sse_mul(n, a, b, c);
                result[n / 128][3] += sse_tile(n, a, b, c, T);

                cout << endl;
            }
        result[n / 128][0] /= 20;
        result[n / 128][1] /= 20;
        result[n / 128][2] /= 20;
        result[n / 128][3] /= 20;
        cout << result[n / 128][0] << "  " << result[n / 128][1] << "  " << ...
            result[n / 128][2] << "  " << result[n / 128][3] << endl;


        n += 128;
        cout << endl;

    }
```

```


    return 0;


}
```

2. 高斯消元以及回代测试源码

```
#include <iostream>
#include <immintrin.h>
#include <stdlib.h>
#include <algorithm>
#include <windows.h>
#include <thread>

using namespace std;
const int maxN = 5000;
const int N = 8192;
float a[maxN][maxN];
float b[maxN][maxN];
float c[maxN][maxN];
float answer[maxN];
float answer2[maxN];
float answer3[maxN];
float x[maxN];
double LU(int n, float a[][maxN]) {
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    for (int i = 0; i <= n - 1; i++) //对每一行
    {
        for (int j = i + 1;j <= n;j++)//对这一行的每一个数
        {
            a[i][j] = a[i][j] / a[i][i];
        }
        a[i][i] = 1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
        {
```

```
                float tem = a[j][i] / a[i][i];
                for (int k = i + 1; k ≤ n; k++) //这个是列，列是n+1个（有b）
                {
                    a[j][k] -= a[i][k] * tem;
                }
                a[j][i] = 0.00;
            }
        }
        QueryPerformanceCounter(&endTime);

        double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
        cout << "普通方法耗时：" << time << "s" << endl;
        return time;


}
double generation(int n, float a[][maxN], float x[])
{
        LARGE_INTEGER freq;
        LARGE_INTEGER beginTime;
        LARGE_INTEGER endTime;

        QueryPerformanceFrequency(&freq);
        QueryPerformanceCounter(&beginTime);
        for (int i = n - 1; i ≥ 0; i--) {
            x[i] = a[i][n];
            for (int j = i + 1; j < n; j++) {
                x[i] -= a[i][j] * x[j];
            }
            x[i] /= a[i][i];
        }
        QueryPerformanceCounter(&endTime);

        double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
        cout << "普通方法回代：" << time << "s" << endl;
        return time;

}
double LU_SSE(int n, float a[][maxN]) {
        LARGE_INTEGER freq;
        LARGE_INTEGER beginTime;
```

```
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m128 t1, t2, sub, tem2;
    for (int i = 0; i <= n - 1; i++) //对每一行
    {
        for (int j = i + 1;j <= n;j++)//对这一行的每一个数
        {
            a[i][j] = a[i][j] / a[i][i];
        }
        a[i][i] = 1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
        {
            float tem = a[j][i] / a[i][i];
            tem2 = _mm_set1_ps(tem);
            for (int k = i + 1; k <= n; k += 4) //这个是列，列是n+1个（有b）

            {
                if (k + 3 > n)break;
                t1 = _mm_loadu_ps(a[i] + k);
                t2 = _mm_loadu_ps(a[j] + k);
                sub = _mm_sub_ps(t2, _mm_mul_ps(t1, tem2));
                _mm_storeu_ps(a[j] + k, sub);
            }
            for (int k = n - (n - i) % 4 + 1; k <= n; k += 1) {
                a[j][k] -= a[i][k] * tem;
            }

            a[j][i] = 0.00;


        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "SSE高斯消元耗时：" << time << "s" << endl;
    return time;

}
double LU_AVX(int n, float a[][maxN]) {
```

```
    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m256 t1, t2, sub, tem2;
    for (int i = 0; i <= n - 1; i++) //对每一行
    {
        for (int j = i + 1;j <= n;j++)//对这一行的每一个数
        {
            a[i][j] = a[i][j] / a[i][i];
        }
        a[i][i] = 1.0;
        for (int j = i + 1; j < n; j++)//每一行的下面的行
        {
            float tem = a[j][i] / a[i][i];
            tem2 = _mm256_set1_ps(tem);
            for (int k = i + 1; k <= n; k += 8) //这个是列, 列是n+1个 (有b)

            {
                if (k + 7 > n)break;
                t1 = _mm256_loadu_ps(a[i] + k);
                t2 = _mm256_loadu_ps(a[j] + k);
                sub = _mm256_sub_ps(t2, _mm256_mul_ps(t1, tem2));
                _mm256_storeu_ps(a[j] + k, sub);
            }
            for (int k = n - (n - i) % 8 + 1; k <= n; k += 1) {
                a[j][k] -= a[i][k] * tem;
            }

            a[j][i] = 0.00;


        }
    }
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "AVX高斯消元耗时: " << time << "s" << endl;
    return time;

```

```
}
double generation_SSE(int n, float a[][maxN], float x[]) {

    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m128 t1, t2, t3, sub, tem;
    //转置


    for (int i = n - 1; i >= 0; i--) {
        x[i] = a[i][n];
        for (int j = i + 1; j < n; j += 4) {
            if (j + 3 >= n)break;
            t1 = _mm_load_ps(a[i] + j);
            t2 = _mm_load_ps(x + j);
            t3 = _mm_mul_ps(t1, t2);
            t3 = _mm_hadd_ps(t3, t3);
            t3 = _mm_hadd_ps(t3, t3);
            x[i] -= _mm_cvtss_f32(t3);

        }
        for (int j = n - (n - i-1) % 4 ; j < n; j++) {

            x[i] -= a[i][j] * x[j];

        }




    }


    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "SSE方法优化回代：" << time << "s" << endl;
    return time;
```

```
}
double generation_AVX(int n, float a[][maxN], float x[]) {

    LARGE_INTEGER freq;
    LARGE_INTEGER beginTime;
    LARGE_INTEGER endTime;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&beginTime);
    __m256 t1, t2, t3,tem;


    for (int i = n - 1; i >= 0; i--) {
        x[i] = a[i][n];
        for (int j = i + 1; j < n; j += 8) {
            if (j + 7 >= n)break;
            t1 = _mm256_load_ps(a[i] + j);
            t2 = _mm256_load_ps(x + j);
            t3 = _mm256_mul_ps(t1, t2);
            t3 = _mm256_hadd_ps(t3, t3);
            t3 = _mm256_hadd_ps(t3, t3);
            x[i] -= (_mm256_cvtss_f32(t3) + _mm256_cvtss_f32(...
                _mm256_permute2f128_ps(t3, t3, 1)));

        }
        for (int j = n - (n - i - 1) % 8; j < n; j++) {

            x[i] -= a[i][j] * x[j];

        }




    }


```

```
    QueryPerformanceCounter(&endTime);

    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
    cout << "AVX方法优化回代: " << time << "s" << endl;
    return time;






}
int main()
{
    int n;
    n = 128;
    int x = 0;


    double result[100][8];
    for (int i = 0;i < 100;i++) {
        for (int j = 0;j < 7;j++)
            result[i][j] = 0;
    }
    while (n ≤ 2560)
    {

        cout << "矩阵规模为" << n << "*" << n << "时" << endl;
        x = 0;


        while (x++ < 20)
        {
            cout << "第" << x << "次高斯消元" << endl;
            cout << endl;
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n + 1; j++) {
                    a[i][j] = (rand() % 100000) / 100.00;
                    b[i][j] = a[i][j];
                    c[i][j] = a[i][j];
                    answer[j] = 0.0;
```

```
                        answer2[j] = 0.0;
                        answer3[j] = 0.0;
                    }

                }
                result[n / 128][0] += LU(n, a);
                result[n / 128][1] += LU_SSE(n, b);
                result[n / 128][2] += LU_AVX(n, c);
                result[n / 128][3] += generation(n, a, answer);
                result[n / 128][4] += generation_SSE(n, b, answer2);
                result[n / 128][5] += generation_AVX(n, c, answer3);


            }
            result[n / 128][0] /= 20;
            result[n / 128][1] /= 20;
            result[n / 128][2] /= 20;
            result[n / 128][3] /= 20;
            result[n / 128][4] /= 20;
            result[n / 128][5] /= 20;

            cout << result[n / 128][0] << "  " << result[n / 128][1] << "  " << ...
                result[n / 128][2] << "  " << result[n / 128][3] << "  " << ...
                result[n / 128][4] << "  " << result[n / 128][5] << endl;


            n += 128;
            cout << endl;

        }
        return 0;

}
```

3. 高斯消元正确性验证代码以及结果

```
int main()
{
    for (int i = 0;i <10;i++)
        for (int j = 0;j < 11;j++)
        {
            cin >> a[i][j];
```

```
        }
    for (int i = 0;i < 10;i++)
    {
        for (int j = 0;j < 11;j++)
        {
            cout << a[i][j] << " ";

        }
        cout << endl;
    }
        cout << endl;
    LU_SSE(10, a);
    for (int i = 0;i < 10;i++)
    {
        for (int j = 0;j < 11;j++)
        {
            cout << a[i][j] << " ";
            b[i][j] = a[i][j];
            c[i][j] = a[i][j];
        }
        cout << endl;

    }



    generation(10, a, x);
    for (int i = 0;i < 10;i++) {
        cout << x[i] << " ";
        x[i] = 0;
    }
    cout << endl;


    generation_AVX(10, b, x);
    for (int i = 0;i < 10;i++) {
        cout << x[i] << " ";
        x[i] = 0;
    }
    cout << endl;
    generation_SSE(10, c, x);
    for (int i = 0;i < 10;i++) {
        cout << x[i] << " ";
```

```
            x[i] = 0;
        }

}
```

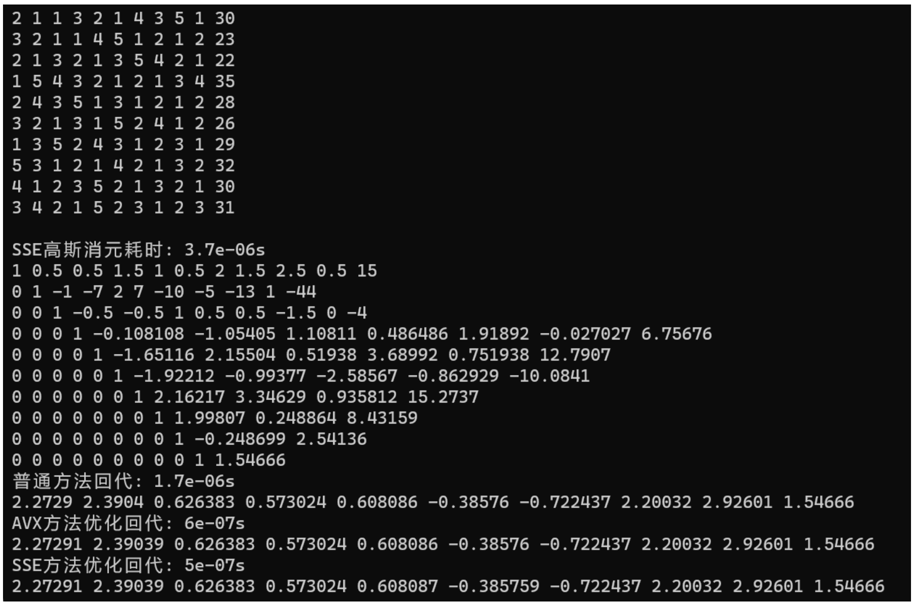
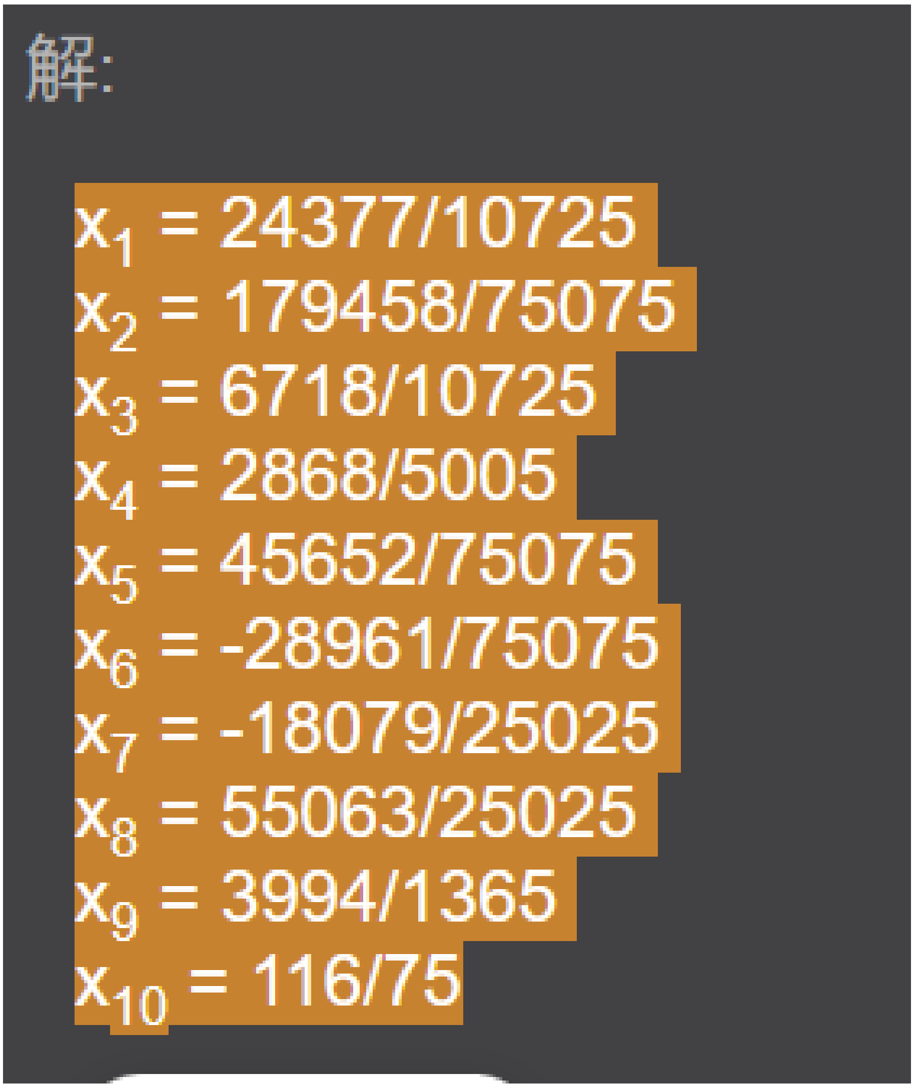

2 1 1 3 2 1 4 3 5 1 30
3 2 1 1 4 5 1 2 1 2 23
2 1 3 2 1 3 5 4 2 1 22
1 5 4 3 2 1 2 1 3 4 35
2 4 3 5 1 3 1 2 1 2 28
3 2 1 3 1 5 2 4 1 2 26
1 3 5 2 4 3 1 2 3 1 29
5 3 1 2 1 4 2 1 3 2 32
4 1 2 3 5 2 1 3 2 1 30
3 4 2 1 5 2 3 1 2 3 31
SSE高斯消元耗时: 3.7e-06s
1 0.5 0.5 1.5 1 0.5 2 1.5 2.5 0.5 15
0 1 -1 -7 2 7 -10 -5 -13 1 -44
0 0 1 -0.5 -0.5 1 0.5 0.5 -1.5 0 -4
0 0 0 1 -0.108108 -1.05405 1.10811 0.486486 1.91892 -0.027027 6.75676
0 0 0 0 1 -1.65116 2.15504 0.51938 3.68992 0.751938 12.7907
0 0 0 0 0 1 -1.92212 -0.99377 -2.58567 -0.862929 -10.0841
0 0 0 0 0 0 1 2.16217 3.34629 0.935812 15.2737
0 0 0 0 0 0 0 1 1.99807 0.248864 8.43159
0 0 0 0 0 0 0 0 1 -0.248699 2.54136
0 0 0 0 0 0 0 0 0 1 1.54666
普通方法回代: 1.7e-06s
2.2729 2.3904 0.626383 0.573024 0.608086 -0.38576 -0.722437 2.20032 2.92601 1.54666
AVX方法优化回代: 6e-07s
2.27291 2.39039 0.626383 0.573024 0.608086 -0.38576 -0.722437 2.20032 2.92601 1.54666
SSE方法优化回代: 5e-07s
2.27291 2.39039 0.626383 0.573024 0.608087 -0.385759 -0.722437 2.20032 2.92601 1.54666

图 2.1: 计算结果



解:

$x_1 = 24377/10725$
$x_2 = 179458/75075$
$x_3 = 6718/10725$
$x_4 = 2868/5005$
$x_5 = 45652/75075$
$x_6 = -28961/75075$
$x_7 = -18079/25025$
$x_8 = 55063/25025$
$x_9 = 3994/1365$
$x_{10} = 116/75$

图 2.2: 正确答案