中山大学数据科学与计算机学院本科生实验报告（2020学年秋季学期）

- 课程名称：**高性能计算程序设计**
- 任课教师：**黄聃**
- 年级/专业：**18级/计算机**
- 学号/姓名：**18340236/朱煜**
- Email：zhuy85@mail2.sysu.edu.cn
- 完成日期：**2020.09.25**

一、实验目的

1、通用矩阵乘法

- 数学上，一个 m x n 的矩阵是一个由 m 行 n 列元素排列成的矩形阵列。矩阵是高等代数中常见的数学工具，也常见于统计分析等应用数学学科中。矩阵运算是数值分析领域中的重要问题。
- 通用矩阵乘法（**GEMM**）通常定义为：

$$C = AB$$

$$C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$$

- 请根据定义用C语言实现一个矩阵乘法：

  题目：用语言实现通用矩阵乘法
  输入：M，N，K 三个整数（512 ~2048）
  问题描述：随机生成 M * N 和N * K 的两个矩阵A，B，对这两个矩阵做乘法得到矩阵C
  输出：A，B，C 三个矩阵以及矩阵计算的时间

2、通用矩阵乘法优化

- 对上述的矩阵乘法进行优化，优化方法可以分为以下两类：
  1. 基于算法分析的方法对矩阵乘法进行优化，典型的算法包括 Strassen 算法和 Coppersmith–Winograd 算法.
  2. 基于软件优化的方法对矩阵乘法进行优化，如循环拆分向量化和内存重排
- 实验要求：对优化方法进行详细描述，并提供优化后的源代码，以及与**GEMM**的计算时间对比

3、进阶：大规模矩阵计算优化

- 进阶问题描述：如何让程序支持大规模矩阵乘法？考虑两个优化方向
  1. 性能，提高大规模稀疏矩阵乘法性能；
  2. 可靠性，在内存有限的情况下，如何保证大规模矩阵乘法计算完成（M, N, K >> 100000），不触发内存溢出异常。对优化方法及思想进行详细描述，提供大规模矩阵计算优化代码可加分。
- References:
  [1]{GEMM 优化}
  https://jackwish.net/2019/gemm-optimization.html
  [2]{矩阵说明}
  https://zh.wikipedia.org/wiki/%E7%9F%A9%E9%98%B5

二、实验过程

1、通用矩阵乘法

- 通用矩阵乘法通过简单的三层循环计算结果，实现代码如下

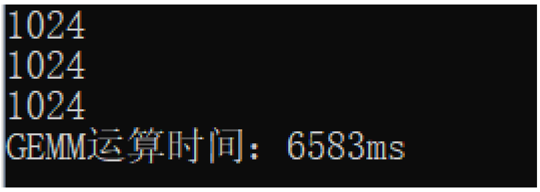
```
void Mul(int** A, int** B, int** R, int M, int N, int K)
{
    //通用矩阵乘法
    for (int i = 0; i < M; ++i)
    {
        for (int j = 0; j < K; ++j)
        {
            R[i][j] = 0;
            for (int z = 0; z < N; ++z)
                R[i][j] += A[i][z] * B[z][j];
        }
    }
}
```

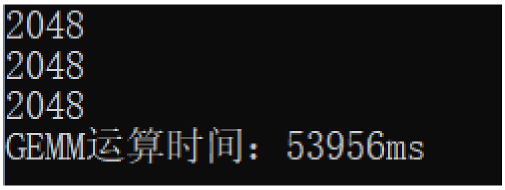
- 先输入较小的矩阵以验证正确性，可检验得到结果正确

```
6
5
4
Matrix A:
5 8 7 4 8
1 3 0 7 2
8 2 7 6 7
5 7 8 3 0
0 6 5 0 4
7 6 5 8 5
Matrix B:
2 0 2 0
6 4 8 1
7 3 2 6
2 3 6 2
3 7 2 1
Matrix C:
139 121 128 66
40 47 72 19
110 96 96 63
114 61 100 61
83 67 66 40
116 98 130 57
GEMM运算时间：0ms
```

- 输入大型矩阵（512~2048）进行计算，为了方便调试，不打印矩阵，只进行时间的输出

```
512
512
512
GEMM运算时间：698ms
```





由**GEMM**的复杂度为$O(N^3)$，运算时间大致呈三次方增长。

2、通用矩阵乘法优化

（1）Strassen算法

- Strassen算法采用的是分治的方法，每个$n \times n$的矩阵都可以分割为四个$n/2 \times n/2$的矩阵,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

  其中$A_{i,j}$，$B_{i,j}$，$C_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$。

  根据矩阵的运算法则，拆分后的各矩阵的运算关系如下，需要进行八次的小矩阵乘法与四次小矩阵的加法。

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{2,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{2,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

  考虑其递归式如下,可以得到其时间复杂度为$O(n^3)$，因此简单的分治并没有效果。

$$T(n) = \begin{cases} 8T(n/2) + O(n^2) & ,n > 1 \\ O(1) & ,n = 1 \end{cases}$$

- 引入以下七个如下所示的用以辅助计算的中间矩阵

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_4 &:= \mathbf{A}_{1,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \end{aligned}$$

  得到中间矩阵后，将其组合得到最后的矩阵

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{aligned}$$

  由于进行了七次乘法与十八次加法，则减少了一次小矩阵乘法，则$Strassen$算法将矩阵乘法的复杂度降低到了$O(n^{log_2 7})$。

- 完全应用$Strassen$算法的一个局限是其要求矩阵乘的规模为 2^n，这在现实情况中不容易满足。一种解决方法是将规模分解为$2^n X$其中 X 无法被 2整除，那么可以应用$Strassen$算法不断递归拆分计算直到小矩阵规模为X。此时可以用朴素算法计算小矩阵；或者将 X补零为 2^n 再继续应用Strassen算法（亦可直接对大矩阵补零）。在本次实验中，直接对大矩阵补零，因此在某些非2^n的输入情况下，$Strassen$算法效果较差。$Strassen$算法具体实现代码如下

```
void Strassen(int** A, int** B, int** C, int length)
{
    int Half = length / 2;
    //当规模小于64时，直接采用通用矩阵乘法
    if (length <= 64)
    {
        Mul(A, B, C, length, length, length);
        return;
    }
    int** A11 = new int* [Half];
    int** A12 = new int* [Half];
    int** A21 = new int* [Half];
    int** A22 = new int* [Half];
    int** B11 = new int* [Half];
    int** B12 = new int* [Half];
    int** B21 = new int* [Half];
    int** B22 = new int* [Half];
    int** C11 = new int* [Half];
    int** C12 = new int* [Half];
    int** C21 = new int* [Half];
    int** C22 = new int* [Half];
    int** M1 = new int* [Half];
    int** M2 = new int* [Half];
    int** M3 = new int* [Half];
    int** M4 = new int* [Half];
    int** M5 = new int* [Half];
    int** M6 = new int* [Half];
    int** M7 = new int* [Half];

    int** TempA = new int* [Half];
    int** TempB = new int* [Half];

    for (int i = 0; i < Half; ++i)
    {
        A11[i] = new int[Half];
        A12[i] = new int[Half];
        A21[i] = new int[Half];
        A22[i] = new int[Half];
        B11[i] = new int[Half];
        B12[i] = new int[Half];
        B21[i] = new int[Half];
        B22[i] = new int[Half];
        C11[i] = new int[Half];
        C12[i] = new int[Half];
        C21[i] = new int[Half];
        C22[i] = new int[Half];
        M1[i] = new int[Half];
        M2[i] = new int[Half];
        M3[i] = new int[Half];
        M4[i] = new int[Half];
        M5[i] = new int[Half];
        M6[i] = new int[Half];
        M7[i] = new int[Half];

        TempA[i] = new int[Half];
        TempB[i] = new int[Half];
    }
```

```
    //初始化各矩阵
    for (int i = 0; i < Half; i++)
    {
        for (int j = 0; j < Half; j++)
        {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + Half];
            A21[i][j] = A[i + Half][j];
            A22[i][j] = A[i + Half][j + Half];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + Half];
            B21[i][j] = B[i + Half][j];
            B22[i][j] = B[i + Half][j + Half];
        }
    }
    //M1 = (A11 + A22)*(B11 + B22)
    Add(A11, A22, TempA, Half);
    Add(B11, B22, TempB, Half);
    Strassen(TempA, TempB, M1, Half);
    //M2 = (A21 + A22)*B11
    Add(A21, A22, TempA, Half);
    Strassen(TempA, B11, M2, Half);
    //M3 = A11*(B12 - B22)
    Sub(B12, B22, TempA, Half);
    Strassen(TempA, A11, M3, Half);
    //M4 = A12*(B21 - B11)
    Sub(B21, B11, TempA, Half);
    Strassen(A22, TempA, M4, Half);
    //M5 = (A11 + A22)*B22
    Add(A11, A12, TempA, Half);
    Strassen(TempA, B22, M5, Half)
    //M6 = (A21 - A11)*(B11 + B22)
    Sub(A21, A11, TempA, Half);
    Add(B11, B12, TempB, Half);
    Strassen(TempA, TempB, M6, Half);
    //M7 = (A12 - A22)*(B21 + B22)
    Sub(A12, A22, TempA, Half);
    Add(B21, B22, TempB, Half);
    Strassen(TempA, TempB, M7, Half);

    //C11 = M1 + M4 - M5 + M7;
    Add(M1, M4, TempA, Half);
    Sub(M7, M5, TempB, Half);
    Add(TempA, TempB, C11, Half);
    //C12 = M3 + M5;
    Add(M3, M5, C12, Half);
    //C21 = M2 + M4;
    Add(M2, M4, C21, Half);
    //C22 = M1 + M3 - M2 + M6;
    Add(M1, M3, TempA, Half);
    Sub(M6, M2, TempB, Half);
    Add(TempA, TempB, C22, Half);

    for (int i = 0; i < Half; ++i)
    {
        for (int j = 0; j < Half; ++j)
        {
            C[i][j] = C11[i][j];
```

```
            C[i][j + Half] = C12[i][j];
            C[i + Half][j] = C21[i][j];
            C[i + Half][j + Half] = C22[i][j];
        }
    }
    for (int i = 0; i < Half; i++)
    {
        delete[] A11[i];delete[] A12[i];delete[] A21[i];delete[] A22[i];
        delete[] B11[i];delete[] B12[i];delete[] B21[i];delete[] B22[i];
        delete[] C11[i];delete[] C12[i];delete[] C21[i];delete[] C22[i];
        delete[] M1[i];delete[] M2[i];delete[] M3[i];delete[] M4[i];
        delete[] M5[i];delete[] M6[i];delete[] M7[i];

        delete[] TempA[i];
        delete[] TempB[i];
    }
    delete[] A11;delete[] A12;delete[] A21;delete[] A22;
    delete[] B11;delete[] B12;delete[] B21;delete[] B22;
    delete[] C11;delete[] C12;delete[] C21;delete[] C22;

    delete[] M1;delete[] M2;delete[] M3;delete[] M4;
    delete[] M5;
    delete[] M6;
    delete[] M7;

    delete[] TempA;
    delete[] TempB;
    return;
}
```

为了减少对大矩阵补零的误差，输入采用2^n，将**GEMM**与$Strassen$算法进行比较，结果如下

```
512
512
512
GEMM运算时间：1231ms
Strassen运算时间：914ms
```

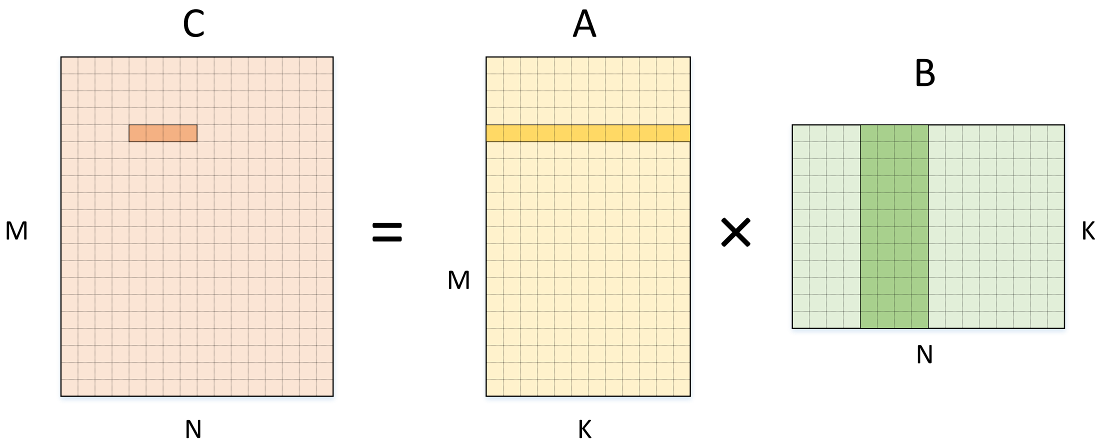
```
1024
1024
1024
GEMM运算时间：12631ms
Strassen运算时间：5980ms
```

```
2048
2048
2048
GEMM运算时间：95615ms
Strassen运算时间：38924ms
```

比较可得，当矩阵规模增大时，$Strassen$算法能有效地提高矩阵乘法的运算速度。

（2）循环拆分向量化和内存重排

- 在矩阵运行过程中，有大量的矩阵元素重复使用，因此可通过减少访存次数来提高矩阵乘法的运算速度，如进行1×4的矩阵乘法，需要用到A矩阵的1行与B矩阵的4列，



该计算的伪代码如下，由伪代码可知，C[m][n+0..3]，A[m][k]进行了重复的访存，因此可以通过将上述重复访存的元素保存在寄存器中，以提高运算的效率。

```
for (int m = 0; m < M; m++) {
  for (int n = 0; n < N; n += 4) {
    C[m][n + 0] = 0;
    C[m][n + 1] = 0;
    C[m][n + 2] = 0;
    C[m][n + 3] = 0;
    for (int k = 0; k < K; k++) {
      C[m][n + 0] += A[m][k] * B[k][n + 0];
      C[m][n + 1] += A[m][k] * B[k][n + 1];
      C[m][n + 2] += A[m][k] * B[k][n + 2];
      C[m][n + 3] += A[m][k] * B[k][n + 3];
    }
  }
}
```

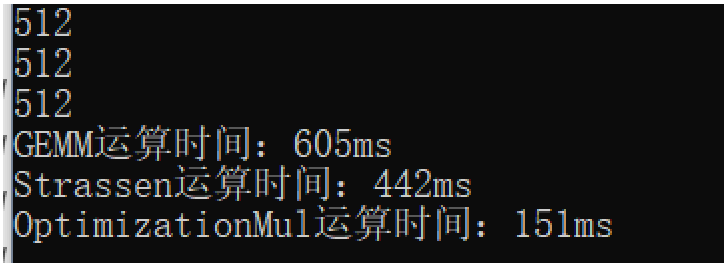
同样道理，优化4×4的矩阵乘法，通过对A矩阵与B矩阵的访存元素复用，可减少访存次数，其中4×4的访存从原来的$4MNK$减少到了$\frac{5}{2}MNK$，具体实现代码如下，

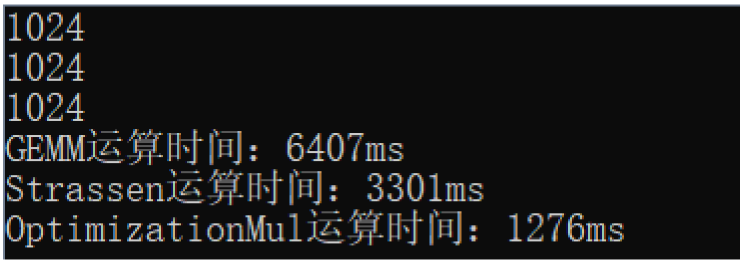
```
void OptimizationMul(int** A, int** B, int** C, int length)
{
    //对矩阵进行4*4的拆分
    for (int i = 0; i < length; i += 4)
    {
        for (int j = 0; j < length; j += 4)
        {
            VectorMul(A, B, C, i, j, length);
        }
    }
}
void VectorMul(int** A, int** B, int** C, int row, int col, int length)
{
    //使用寄存器保存各结果
    register int c_00_reg,
        c_01_reg, c_02_reg, c_03_reg, c_10_reg, c_11_reg, c_12_reg,
c_13_reg, c_20_reg, c_21_reg, c_22_reg, c_23_reg, c_30_reg, c_31_reg,
c_32_reg, c_33_reg,
```

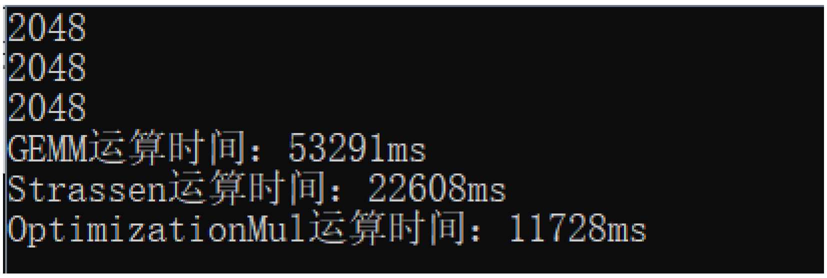
```
        a_0i_reg, a_1i_reg, a_2i_reg, a_3i_reg, b_i0_reg, b_i1_reg,
b_i2_reg, b_i3_reg;

    c_00_reg = 0.0;
    c_01_reg = 0.0;
    c_02_reg = 0.0;
    c_03_reg = 0.0;
    c_10_reg = 0.0;
    c_11_reg = 0.0;
    c_12_reg = 0.0;
    c_13_reg = 0.0;
    c_20_reg = 0.0;
    c_21_reg = 0.0;
    c_22_reg = 0.0;
    c_23_reg = 0.0;
    c_30_reg = 0.0;
    c_31_reg = 0.0;
    c_32_reg = 0.0;
    c_33_reg = 0.0;


    for (int i = 0; i < length; ++i)
    {
        //保存重复访存的元素
        a_0i_reg = A[row][i];
        a_1i_reg = A[row + 1][i];
        a_2i_reg = A[row + 2][i];
        a_3i_reg = A[row + 3][i];

        b_i0_reg = B[i][col];
        b_i1_reg = B[i][col + 1];
        b_i2_reg = B[i][col + 2];
        b_i3_reg = B[i][col + 3];

        c_00_reg += a_0i_reg * b_i0_reg;
        c_01_reg += a_0i_reg * b_i1_reg;
        c_02_reg += a_0i_reg * b_i2_reg;
        c_03_reg += a_0i_reg * b_i3_reg;

        c_10_reg += a_1i_reg * b_i0_reg;
        c_11_reg += a_1i_reg * b_i1_reg;
        c_12_reg += a_1i_reg * b_i2_reg;
        c_13_reg += a_1i_reg * b_i3_reg;

        c_20_reg += a_2i_reg * b_i0_reg;
        c_21_reg += a_2i_reg * b_i1_reg;
        c_22_reg += a_2i_reg * b_i2_reg;
        c_23_reg += a_2i_reg * b_i3_reg;

        c_30_reg += a_3i_reg * b_i0_reg;
        c_31_reg += a_3i_reg * b_i1_reg;
        c_32_reg += a_3i_reg * b_i2_reg;
        c_33_reg += a_3i_reg * b_i3_reg;

    }
    //将结果输出到目标矩阵
    C[row][col] += c_00_reg;
    C[row][col + 1] += c_01_reg;
```

```
        c[row][col + 2] += c_02_reg;
        c[row][col + 3] += c_03_reg;

        c[row + 1][col] += c_10_reg;
        c[row + 1][col + 1] += c_11_reg;
        c[row + 1][col + 2] += c_12_reg;
        c[row + 1][col + 3] += c_13_reg;

        c[row + 2][col] += c_20_reg;
        c[row + 2][col + 1] += c_21_reg;
        c[row + 2][col + 2] += c_22_reg;
        c[row + 2][col + 3] += c_23_reg;

        c[row + 3][col] += c_30_reg;
        c[row + 3][col + 1] += c_31_reg;
        c[row + 3][col + 2] += c_32_reg;
        c[row + 3][col + 3] += c_33_reg;
    }
```

- 由于进行了4×4的矩阵拆分，因此要求矩阵的行与列为4的倍数，为了方便比较，使用512、1024、2048三种情况的方矩阵，运算结果如下，可见使用循环向量拆分能有效地减少矩阵运算的时间。

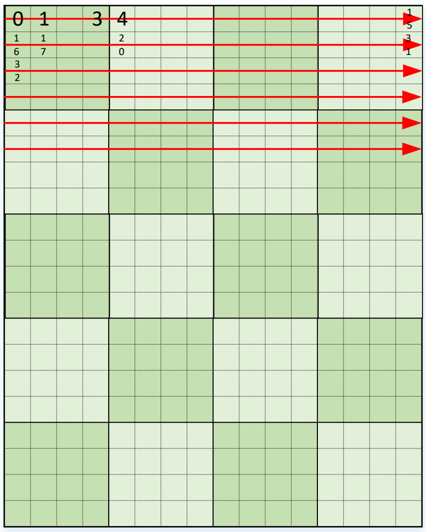


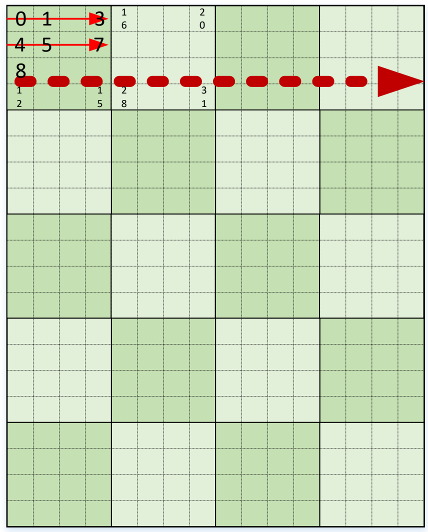




（2）内存优化

- 由于数组元素在内存中时连续存取，而我们进行4×4矩阵向量拆分时，需要在同一个循环内访问不同行的内容，会降低Cache的命中率，因此进行矩阵重排可提高访问矩阵时的Cache命中率。由于我们进行的是4×4的矩阵重排，因此对于$4 \times col$的矩阵，通过以下的内存排布，理想情况下，可以提高Cache的命中率。





- 具体实现代码如下，在本次实验中，由于采用的是二维指针，因此可能在内存中矩阵的排布并不是连续的，降低了空间局部性，中间增加了打包与新索引过程，因此可能会出现负加速的情况，效果并没有想象的明显，因此不做比较。

```
int** PacketMatrix(int** A, int M, int K, int length)
{
    //重新打包
    int** PA = new int* [length];
    for (int i = 0; i < length; ++i)
        PA[i] = new int[length];
    //根据计算公式得到新位置
    for (int i = 0; i < length; i += 4)
    {
        for (int j = i; j < i + 4; j++)
        {
            for (int l = 0; l < length; l++)
            {
                PA[i + l % 4][j % 4 * 4 + l % 4] = A[j][l];
            }
        }
    }
    return PA;
}

void StorageVectorMul(int** PA, int** B, int** C, int row, int col, int
length)
{
    register int c_00_reg, c_01_reg, c_02_reg, c_03_reg,
        c_10_reg, c_11_reg, c_12_reg, c_13_reg,
        c_20_reg, c_21_reg, c_22_reg, c_23_reg,
        c_30_reg, c_31_reg, c_32_reg, c_33_reg,
        a_0i_reg, a_1i_reg, a_2i_reg, a_3i_reg,
        b_i0_reg, b_i1_reg, b_i2_reg, b_i3_reg;

    c_00_reg = 0; c_01_reg = 0; c_02_reg = 0; c_03_reg = 0;
    c_10_reg = 0; c_11_reg = 0; c_12_reg = 0; c_13_reg = 0;
    c_20_reg = 0; c_21_reg = 0; c_22_reg = 0; c_23_reg = 0;
    c_30_reg = 0; c_31_reg = 0; c_32_reg = 0; c_33_reg = 0;


    for (int i = 0; i < length; ++i)
    {
```

```
        //找到打包后的新下标位置
        a_0i_reg = PA[row + i % 4][row % 4 * 4 + i % 4];
        a_1i_reg = PA[row + i % 4][(row + 1) % 4 * 4 + i % 4];
        a_2i_reg = PA[row + i % 4][(row + 2) % 4 * 4 + i % 4];
        a_3i_reg = PA[row + i % 4][(row + 3) % 4 * 4 + i % 4];

        b_i0_reg = B[i][col];
        b_i1_reg = B[i][col + 1];
        b_i2_reg = B[i][col + 2];
        b_i3_reg = B[i][col + 3];


        c_00_reg += a_0i_reg * b_i0_reg;
        c_01_reg += a_0i_reg * b_i1_reg;
        c_02_reg += a_0i_reg * b_i2_reg;
        c_03_reg += a_0i_reg * b_i3_reg;

        c_10_reg += a_1i_reg * b_i0_reg;
        c_11_reg += a_1i_reg * b_i1_reg;
        c_12_reg += a_1i_reg * b_i2_reg;
        c_13_reg += a_1i_reg * b_i3_reg;

        c_20_reg += a_2i_reg * b_i0_reg;
        c_21_reg += a_2i_reg * b_i1_reg;
        c_22_reg += a_2i_reg * b_i2_reg;
        c_23_reg += a_2i_reg * b_i3_reg;

        c_30_reg += a_3i_reg * b_i0_reg;
        c_31_reg += a_3i_reg * b_i1_reg;
        c_32_reg += a_3i_reg * b_i2_reg;
        c_33_reg += a_3i_reg * b_i3_reg;

    }
    C[row][col] += c_00_reg;
    C[row][col + 1] += c_01_reg;
    C[row][col + 2] += c_02_reg;
    C[row][col + 3] += c_03_reg;

    C[row + 1][col] += c_10_reg;
    C[row + 1][col + 1] += c_11_reg;
    C[row + 1][col + 2] += c_12_reg;
    C[row + 1][col + 3] += c_13_reg;

    C[row + 2][col] += c_20_reg;
    C[row + 2][col + 1] += c_21_reg;
    C[row + 2][col + 2] += c_22_reg;
    C[row + 2][col + 3] += c_23_reg;

    C[row + 3][col] += c_30_reg;
    C[row + 3][col + 1] += c_31_reg;
    C[row + 3][col + 2] += c_32_reg;
    C[row + 3][col + 3] += c_33_reg;

}
```

3、进阶：大规模矩阵计算优化

（1）大规模稀疏矩阵

- 假设矩阵A与B都为稀疏矩阵，则可以通过简单地遍历A或B中的非0元素来才进行矩阵运算，这种方式仅考虑两个矩阵A与B中较为稀疏的一个，以A为例，将GEMM的三层循环改变顺序，并增加对A元素的判断，实现代码如下，这是较为简单的稀疏矩阵的乘法

```
void SparseMatrixMul(int** A, int** B, int** C, int length)
{
    //对稀疏矩阵A进行遍历
    for (int i = 0; i < length; i++)
    {
        for (int k = 0; k < length; k++)
        {
            if (!A[i][k])
                continue;
            for (int j = 0; j < length; j++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

- 假设矩阵A与B都为稀疏矩阵，则可使用结构体存储A与B矩阵中非0元素的坐标，在计算时只需要将所有非0元素，考虑计算即可，实现的结构体与计算的代码如下

```
struct Point
{
    //坐标
    int i, j;
    Point(int ia, int ja)
    {
        i = ia;
        j = ja;
    }
};

vector<Point> getNonZeroPoints(int** matrix,int length) {
    vector<Point> nonZeroPoints;
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < length; j++) {
            if (matrix[i][j] != 0) {
                Point* temp = new Point(i, j);
                nonZeroPoints.push_back(*temp);
            }
        }
    }
    return nonZeroPoints;
}

void SparseMatrixMul(int** A, int** B, int** C, vector<Point> VA,
vector<Point> VB, int length)
{
    //计算非0位置的值
    for (Point pA : VA) {
        for (Point pB : VB) {
```

```
            if (pA.j == pB.i) {
                C[pA.i][pB.j] += A[pA.i][pA.j] * B[pB.i][pB.j];
            }
        }
    }
}
```

- 构建稀疏矩阵A、B，验证以上两种方法的加速效果，当稀疏比为0.05时，选择512、1024、2048规模的方形矩阵，结果如下。由于第二种方法使用了`vector`类，在`point`遍历时影响了效率。

```
512
512
512
GEMM运算时间: 912ms
Strassen运算时间: 635ms
OptimizationMul运算时间: 198ms
OptimizationStorageMul运算时间: 278ms
SparseMatrixMul运算时间: 13ms
SparseMatrixMulv2运算时间: 54ms
```

```
1024
1024
1024
GEMM运算时间: 9392ms
Strassen运算时间: 4645ms
OptimizationMul运算时间: 2136ms
OptimizationStorageMul运算时间: 2168ms
SparseMatrixMul运算时间: 88ms
SparseMatrixMulv2运算时间: 165ms
```

```
2048
2048
2048
GEMM运算时间: 79180ms
Strassen运算时间: 31987ms
OptimizationMul运算时间: 18578ms
OptimizationStorageMul运算时间: 19999ms
SparseMatrixMul运算时间: 477ms
SparseMatrixMulv2运算时间: 6665ms
```

（2）大规模矩阵

- 可以采用PC集群并行计算进行大规模的矩阵乘法，将大规模矩阵计算任务划分成一些小的任务, 尽可能的开拓并行执行的机会，采用客户/服务器结构，构成一个集群并行计算环境。在PC集群计算环境下, 对矩阵进行划分, 然后指派给不同的处理器，客户端将数据依次发送给每个服务器, 提高集群之间的通信速度，获得更好效果。

三、实验结果

- 经过检测，以上实现的各种矩阵优化具体运算时间（单位：ms）比较如以下表格

方法\矩阵规模	256	512	1024	2048
GEMM	76	727	7048	51733
Strassen	72	480	3562	22833
循环向量拆分	22	179	1448	12079
内存重排	20	198	1571	12956
稀疏矩阵优化(只考虑A)	3	12	43	149
稀疏矩阵优化(同时考虑A，B)	12	29	133	461

- 根据表格可看到，$Strassen$算法在算法层面上对矩阵运算进行了优化，在大规模矩阵运算的情况下提升效果更好，由而循环拆分向量提升矩阵运算速度的效果比$Strassen$算法的更好，这说明矩阵运算有大量的时间用作访存，可使用$Strassen$与循环向量拆分相结合的方式结合两者的优点，即使用$Strassen$进行矩阵规模的缩小，当矩阵规模小于一定值时，使用循环向量拆分计算矩阵。
- 内存重排在本次实验中效果并不好，原因在于使用的是二维指针，有待修改。
- 稀疏矩阵计算优化效果显著，而第二种使用了`vector`类保存`point`，在使用迭代器遍历时会降低效果，否则计算时间在理想状态下应优于第一种。

四、实验感想

- 本次实验充分体现了程序优化的两个方向，一是算法层面的优化，选择合适的算法能够有效地提高程序运行的效率。二是软件优化，而程序运行很大程度上依赖于程序代码的有效软件优化，这样能有效地减少程序不必要的如访存等行为，在本次实验中循环向量拆分有效地提高了程序运行的效率，这是平常在代码编写过程中没有去注意到的关键。