几种常用线性方程组的直接解法探究

（注：这两周的工作主要在于查找资料和算法的代码实现；由于我们目前没有合适的实验设备，因此还未对算法进行测试，只是分析了其中的原理和实现方式；新添加的部分均用 * 表示出来）

求解线性方程组的方法分为**直接法**和**迭代法**两大类。迭代法采取逐次逼近的方法，从一个初始解出发，按照一定的计算格式，构造一个向量的无穷序列，其极限是方程组的精确解。但在上学期关于迭代法的研究与开发的过程中，系数矩阵的条件数、非零元素分布特征等因素对求解迭代次数的影响很大，相同维度的线性方程组的迭代次数可能是几次、几十次，也有可能是上万次。虽然直接法需要的计算量比较大，一般为方程组维数的三次方，但直接法的**计算量相对稳定**，不会出现迭代法中可能出现的远超方程组维数的三次方的计算量。因此，直接法求解线性方程组也十分具有研究价值。首先我们主要讨论一些最基本的直接法，并在此基础上讨论他们的各种改进以及矩阵分解的一些概念。

1 高斯消元法

1.1 定义

高斯消元法主要通过系数矩阵行向量加减消元，产生一个行阶梯形矩阵，通过行阶梯形矩阵的最后一行的等式，求解一个未知量，并不断向上回代求解，直到求解出所有变量。

1.2 限制条件

由于系数矩阵的所有主元 $a_{ii}^{(i)}$ 在消元时都会被当作除数，因此必须满足系数矩阵的所有主元 $a_{ii}^{(i)} \neq 0$。

1.3 计算过程

考虑 n 阶线性方程组

$$Ax = b$$

系数矩阵为 $\boldsymbol{A} = (a_{ij})_{n\times n}$， 右端向量和精确解分别为 $\boldsymbol{b} = (b_1, b_2, \cdots, b_n)^{\mathrm{T}}$，
$\boldsymbol{x} = \left(x_1, x_2, \cdots, x_n\right)^{\mathrm{T}}$, 它的分量形式为

$$\begin{cases} a_{11}x_1 + a_{12}x_1 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_1 + \cdots + a_{2n}x_n = b_2 \\ \qquad\qquad\qquad \vdots \\ a_{n1}x_1 + a_{n2}x_1 + \cdots + a_{nn}x_n = b_n \end{cases}$$

用高斯消元法求解上述线性方程组的计算过程如下：

分别记矩阵 $\boldsymbol{A}^{(1)} = \boldsymbol{A}$，向量 $\boldsymbol{b}^{(1)} = \boldsymbol{b}$，它们的元素分别为

$$a_{ij}^{(1)} = a_{ij}, \quad b_i^{(1)} = b_i \quad (i, j = 1, 2, \cdots, n)$$

1. 消元过程

   第一步，如果 $a_{11}^{(1)} \neq 0$，可对 $i = 2, 3, \ldots, n$ 做如下的运算，用数 $m_{i1} = -a_{i1}^{(1)}/a_{11}^{(1)}$ 依次乘以方程组的第一行，并加到第 i 行上去，可得到

   $$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ b_3^{(2)} \\ \vdots \\ b_n^{(2)} \end{pmatrix}$$

   其中，

   $$\begin{aligned} a_{ij}^{(2)} &= a_{ij}^{(1)} + m_{i1}a_{1j}^{(1)}, \quad i, j = 2, 3, \cdots, n \\ b_i^{(2)} &= b_i^{(1)} + m_{i1}b_1^{(1)}, \quad i = 2, 3, \cdots, n \end{aligned}$$

   第二步，如果 $a_{22}^{(2)} \neq 0$，可对 $i = 3, \ldots, n$ 做如下的运算，用数 $m_{i2} = -a_{i2}^{(2)}/a_{22}^{(2)}$ 依次乘以方程组的第二行，并加到第 i 行上去，可得到

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ b_3^{(3)} \\ \vdots \\ b_n^{(3)} \end{pmatrix}$$

其中，

$$\begin{aligned} a_{ij}^{(3)} &= a_{ij}^{(2)} + m_{i2} a_{2j}^{(2)}, \quad i,j = 3, \cdots, n \\ b_i^{(3)} &= b_i^{(2)} + m_{i2} b_2^{(2)}, \quad i = 3, \cdots, n \end{aligned}$$

类似地，这样的运算过程一直做到第 $n-1$ 步，最后就把原方程组转化为一个上三角形方程组

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1,n-1}^{(1)} & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{1,n-1}^{(2)} & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1}^{(n-1)} & a_{n-1,n}^{(n-1)} \\ 0 & 0 & \cdots & 0 & a_{nn}^{(n)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_{n-1}^{(n-1)} \\ b_n^{(n)} \end{pmatrix}$$

2. 回代过程

如果 $a_{nn}^{(n)} \neq 0$，可从上述三角形方程组逐次回代计算出线性方程组的解。

$$\$ \begin{cases} x_n = b_n^{(n)} / a_{nn}^{(n)} \\ x_i = \left(b_i^{(i)} - \sum_{j=i+1}^{n} a_{ij}^{(i)} x_j \right) / a_{ii}^{(i)}, \quad (i = n-1, \cdots, 2, 1) \end{cases}$$

1.4 伪代码

```
i = 1
j = 1
while (i ≤ m and j ≤ n) do
    Find pivot in column j, starting in row i
    maxi = i
    for k = i+1 to m do
        if abs(A[k,j]) > abs(A[maxi,j]) then
            maxi = k
    if A[maxi,j] ≠ 0 then
        swap rows i and maxi, but do not change the value of i
        Now A[i,j] will contain the old value of A[maxi,j]
        divide each entry in row i by A[i,j]
        Now A[i,j] will have the value 1
        for u = i+1 to m do
            subtract A[u,j] * row i from row u
            A[u,j] will be 0, since A[u,j]-A[i,j]*A[u,j]=A[u,j]-1*A[u,j]=0.
```

```
        i = i + 1
    j = j + 1
```

1.5 算法分析

高斯消元法主要为浮点数的乘除法运算，我们对两个过程分别进行计算量分析：

1. 消元过程的第 k 步，对矩阵需要做 $(n-k)^2$ 次乘法运算及 $(n-k)$ 次除法运算，对右端向量需作 $(n-k)$ 次乘法运算，所以消元过程总的乘除法运算工作量为

$$\sum_{k=1}^{n-1}(n-k)^2+\sum_{k=1}^{n-1}(n-k)+\sum_{k=1}^{n-1}(n-k)=\frac{1}{3}n^3+\frac{1}{2}n^2-\frac{5}{6}n$$

2. 回代过程中，计算每个 x_k 需作 $(n-k+1)$ 次乘除法运算，其工作量为

$$\sum_{k=1}^{n}(n-k+1)=\frac{1}{2}n(n+1)$$

因此，高斯消元法计算线性方程组所需要的总的计算量为

$$\frac{1}{3}n^3+n^2-\frac{1}{3}n$$

1.6 加速思路

求解线性方程组时，消元这一操作的运算量是最大的，因而需要实现并行化。矩阵按行存取，将矩阵中的元素与内核函数中的线程一一对应，即矩阵中编号为 (i,j) 元素对应线程块中编号为 (i,j) 的线程。在找到主行后，线程块中主行以后的所有线程同时启动，对矩阵中相应位置的元素进行消元操作。

* 1.7 代码实现

- CPU 版本

```
#include <iostream>
#include <vector>
using namespace std;

vector<vector<double>> gaussianElimination(vector<vector<double>> matrixIn,
int n)
{
    vector<vector<double>> matrix = matrixIn;

    // Rearrange rows to make sure none of the values on the main diagonal
are 0.
    for (int row = 0; row < n; ++row)
    {
        if (matrix[row][row] == 0.0)
        {
            // Iterates to find suitable row to swap with.
            int rowToSwapWith = 0;
```

```
            while (rowToSwapWith < n)
            {
                if (matrix[rowToSwapWith][row] != 0.0)
                {
                    break;
                }
                ++rowToSwapWith;
            }

            // Swaps rows
            vector<double> rowTemp = matrix[row];
            matrix[row] = matrix[rowToSwapWith];
            matrix[rowToSwapWith] = rowTemp;
        }
    }

    /*
    Elimination from top to bottom:
        1. Divide the row by a number such that the value on the main diagonal becomes 1.
        2. Subtract a multiple of the row from all rows under it to set all the values under the aforementioned value of the main diagonal to 0.
        3. The matrix should end up in row echelon form.
    */
     for (int row = 0; row < n; ++row)
    {
        double mainDiagonalValue = matrix[row][row];
        for (int col = 0; col < n + 1; ++col)
        {
            matrix[row][col] /= mainDiagonalValue;
        }

        if (row != n - 1)
        {
            for (int rowUnder = row + 1; rowUnder < n; ++rowUnder)
            {
                double multiple = matrix[rowUnder][row];
                for (int col = 0; col < n + 1; ++col)
                {
                    matrix[rowUnder][col] -= matrix[row][col] * multiple;
                }
            }
        }
    }

   /*
    Back substitution from bottom to top:
        1. Subtract a multiple of the row from all rows above it to set all the values above the value on the main diagonal to 0.
```

```
        2. The matrix should end up in reduced row echelon form.
    */
    for (int row = n - 1; row > 0; --row)
    {
        for (int rowAbove = row - 1; rowAbove >= 0; --rowAbove)
        {
            double multiple = matrix[rowAbove][row];
            for (int col = 0; col < n + 1; ++col)
            {
                matrix[rowAbove][col] -= matrix[row][col] * multiple;
            }
        }
    }

    return matrix;
}
```

- GPU版本

```
typedef struct {
    unsigned int num_columns;
    unsigned int num_rows;
    unsigned int pitch;
    float* elements;
} Matrix;

int main(int argc, char** argv)
{
    Matrix  A; // The NxN input matrix
    Matrix  U; // The upper triangular matrix
    struct timeval start, stop;
    srand(time(NULL));

    if(argc > 1){
        printf("Error. This program accepts no arguments. \n");
        exit(0);
    }

    // Allocate and initialize the matrices
    A  = allocate_matrix(MATRIX_SIZE, MATRIX_SIZE, 1);
    U  = allocate_matrix(MATRIX_SIZE, MATRIX_SIZE, 0);

    // Perform Gaussian elimination on the CPU
    Matrix reference = allocate_matrix(MATRIX_SIZE, MATRIX_SIZE, 0);
    gettimeofday(&start, NULL);
    int status = compute_gold(U.elements, A.elements, A.num_rows);
    gettimeofday(&stop, NULL);
    printf("Execution time gold = %fs. \n", (float)(stop.tv_sec -
start.tv_sec +\\
```

```
                (stop.tv_usec - start.tv_usec)/(float)1000000));
    if(status == 0){
        printf("Failed to convert given matrix to upper triangular. Try
again.      Exiting. \n");
        exit(0);
    }

    // Check that the principal diagonal elements are 1
    status = perform_simple_check(U);
    if(status == 0){
        printf("The upper triangular matrix is incorrect. Exiting. \n");
        exit(0);
    }
    printf("Gaussian elimination on the CPU was successful. \n");

    // Perform the vector-matrix multiplication on the GPU. Return the
result in U
    gauss_eliminate_on_device(A,U);
    int num_elements = MATRIX_SIZE*MATRIX_SIZE;

    int res = checkResults(U.elements, U.elements, num_elements, 0.001f);
    printf("Test %s\n", (1 == res) ? "PASSED" : "FAILED");

    gauss_eliminate_on_device_optimized(A, U);
    // check if the device result is equivalent to the expected solution
    res = checkResults(U.elements, U.elements, num_elements, 0.001f);
    printf("Test %s\n", (1 == res) ? "PASSED" : "FAILED");

    // Free host matrices
    free(A.elements); A.elements = NULL;
    free(U.elements); U.elements = NULL;
    free(U.elements); U.elements = NULL;

    return 0;
}

void gauss_eliminate_on_device(const Matrix A, Matrix U)
{
    struct timeval start,stop;
    Matrix gpu_u = allocate_matrix_on_gpu( U );

    //Copy matrices to gpu, copy A right into U
    copy_matrix_to_device( gpu_u, A );

    int num_blocks = 1;
    int threads_per_block = 512;
    int ops_per_thread = MATRIX_SIZE / (threads_per_block*num_blocks);

    printf("== GPU (Slow) ==\n");
```

```
77      printf("    Threads per block: %d\n",threads_per_block);
78      printf("    Number of blocks: %d\n",num_blocks);
79      printf("    Operations per thread: %d\n",ops_per_thread);
80
81      dim3 thread_block(threads_per_block, 1, 1);
82      dim3 grid(num_blocks,1);
83
84      gettimeofday(&start, NULL);
85
86      // Launch the kernel <<<grid, thread_block>>>
87      gauss_eliminate_kernel<<<grid, thread_block>>>
(gpu_u.elements,ops_per_thread);
88
89      //Sync at end and check for errors
90      cudaThreadSynchronize();
91      checkCUDAError("FAST KERNEL FAILURE");
92      gettimeofday(&stop, NULL);
93      printf("Execution time GPU = %fs. \n", (float)(stop.tv_sec -
start.tv_sec +\\
94                  (stop.tv_usec - start.tv_usec)/(float)1000000));
95
96      //Copy data back
97      copy_matrix_from_device(U, gpu_u);
98
99      //Free memory on device
100     cudaFree(gpu_u.elements);
101 }
102
103 void  gauss_eliminate_on_device_optimized(const Matrix A, Matrix U){
104     printf("== GPU (Fast) ==\n");
105     Matrix gpu_u = allocate_matrix_on_gpu( U );
106
107     //Copy matrices to gpu, copy A right into U
108     copy_matrix_to_device( gpu_u, A );
109
110     //Each thread within a block will take some j iterations
111     int threads_per_block = 256;
112     struct timeval start,stop;
113     int stride = threads_per_block;
114     printf("    Threads per block / stride: %d\n",threads_per_block);
115
116     int k;
117     gettimeofday(&start, NULL);
118     for(k = 0; k < MATRIX_SIZE; k++)
119     {
120         int isize = (MATRIX_SIZE-1) - (k+1) + 1;
121         int num_blocks = isize;
122         if(num_blocks <= 0)
123         {
```

```
            num_blocks = 1;
        }

        dim3 thread_block(threads_per_block, 1, 1);
        dim3 grid(num_blocks,1);

        gauss_eliminate_kernel_optimized_div<<<grid, thread_block>>>(
            gpu_u.elements,
            k,
            stride);
        gauss_eliminate_kernel_optimized<<<grid, thread_block>>>(
            gpu_u.elements,
            k,
            stride);
            cudaThreadSynchronize();

        checkCUDAError("FAST KERNEL FAILURE");
    }
    gettimeofday(&stop, NULL);
    printf("Execution time GPU = %fs. \n", (float)(stop.tv_sec -
start.tv_sec +\\
                (stop.tv_usec - start.tv_usec)/(float)1000000));


    //Sync at end
    cudaThreadSynchronize();

    //Copy data back
    copy_matrix_from_device(U, gpu_u);

    //Free memory on device
    cudaFree(gpu_u.elements);


    int i, j;
    for(i = 0; i < MATRIX_SIZE; i++)
        for(j = 0; j < i; j++)
        {
            U.elements[i * MATRIX_SIZE + j] = 0.0;
        }
}
```

2 LU分解

2.1 定义

LU分解在本质上高斯消元法的一种表达形式。实质上是将n阶矩阵分解为一个上三角矩阵和单位下三角矩阵的乘积。

2.2 限制条件

被分解矩阵A为n阶矩阵，且所有顺序主子式子均不为0

2.3 计算过程

LU分解的运算过程和高斯消元类似，首先通过杜利托尔算法将A变成LU，该算法先算U的第一行再算L的第一列：

$$u_{1j} = a_{1j}, \quad j = 1, 2, \cdots, n$$
$$l_{i1} = a_{i1}/u_{11}, \quad j = 2, 3, \cdots, n$$

然后再第二行第二列，依次计算下去，若以求出U的前k-1行和L的前k-列，则有：

$$u_{kj} = a_{kj} - \left(l_{k1}u_{1j} + \cdots + l_{k,k-1}u_{k-1,j}\right), \quad j = k, k+1, \cdots, n$$
$$l_{ik} = \left(a_{ik} - l_{i1}u_{1k} - \cdots - l_{i,k-1}u_{k-1,k}\right)/u_{kk}, \quad i = k+1, \cdots, n$$

根据上述过程，杜利托尔算法最终表述如下：

(1)对 $k = 1, 2, \cdots, n$，做 $(2)u_{kj} = a_{kj} - \sum_{s=1}^{k-1} l_{ks}u_{sj}, \quad j = k, k+1, \cdots, n$
$(3)l_{ik} = \left(a_{ik} - \sum_{s=1}^{k-1} l_{is}u_{sk}\right)/u_{kk}, \quad j = k+1, \cdots, n$

然后将原方程$Ax = b$变为下式子进行求解：

$$\begin{cases} \boldsymbol{Ly} = \boldsymbol{b} \\ \boldsymbol{Ux} = \boldsymbol{y} \end{cases}$$

由此可得计算公式：

$$\begin{cases} y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j, \quad i = 1, 2, \cdots, n \\ x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)/u_{ii}, \quad i = n, n-1, \cdots, 1 \end{cases}$$

2.4 代码

```
%求解三角方阵
for r = 1 : n
    for j = r : n
        if r > 1;
            for k = 1 : r-1
                A(r,j) = A(r,j) - A(r,k)*A(k,j);
            end
        end
    end
    for i = r+1 : n
        if r < n;
            for k = 1 : r-1
```

```
                A(i,r) = A(i,r) -A(i,k)*A(k,r);
            end
            A(i,r) = A(i,r)/A(r,r);
        end
    end
    disp(['A(', r, '): ']); A
end

%最后回代
```

2.5 算法分析

由杜利托尔算法描述中可以计算出LU 分解的运算量 (加减乘除) 为：

$$\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n}1+\sum_{j=i+1}^{n}\sum_{k=i+1}^{n}2\right)=\sum_{i=1}^{n-1}\left(n-i+2(n-i)^2\right)=\frac{2}{3}n^3+O\left(n^2\right)$$

再加上回代过程的运算量 $O(n^2)$, 总运算量为: $\frac{2}{3}n^3+O\left(n^2\right)$

从上面的运算复杂度可以看出它要高于普通的高斯消去法。因为一般来说，如果LU分解只是为了单纯求一个非齐次方程组，则没有任何优势可言，但是如果想要求解具有一些结果扰动的方程，即$AX=b$，b有很多情况，但这些情况只是细微的不同，此时，LU分解则在算法复杂度上具有一定的优势，因为当Ax = b频繁地变成$Ax=b'$，此时高斯消元就需要全部重新计算（高斯消元用增广矩阵消元，变化过程是[A, b]→[U, b']），这对大型矩阵来说及其耗时。反观LU分解，因为它不依赖于b，所以计算一次后就可以存储U和L^{-1}，在输出变化后也只是需要简单的相乘。

2.6 并行加速思路

根据LU的分解公式：

$$u_{ri}=a_{ri}-\Sigma_{k=1}^{r-1}l_{rk}u_{ki},i=r,r+1,\ldots,n$$
$$l_{ri}=\left(a_{ir}-\Sigma_{k=1}^{r-1}l_{ik}u_{kr}\right)/u_{rr},i=r+1,\ldots,n$$

可以得到下列计算过程：

$$\begin{bmatrix}a_{11}&a_{12}&a_{13}&a_{14}\\a_{21}&a_{22}&a_{23}&a_{24}\\a_{31}&a_{32}&a_{33}&a_{34}\\a_{41}&a_{42}&a_{43}&a_{41}\end{bmatrix}=\begin{bmatrix}1&0&0&0\\l_{21}&1&0&0\\l_{31}&l_{32}&1&0\\l_{41}&l_{42}&l_{43}&1\end{bmatrix}\begin{bmatrix}u_{11}&u_{12}&u_{13}&u_{14}\\0&u_{22}&u_{23}&u_{44}\\0&0&u_{33}&u_{34}\\0&0&0&u_{44}\end{bmatrix}$$
$$=\begin{bmatrix}u_{11}&u_{12}&u_{13}&u_{14}\\l_{21}u_{11}&l_2u_{12}+u_{22}&l_{21}u_{13}+u_{23}&l_{22}u_{14}+u_{24}\\l_{31}u_{11}&l_{31}u_{12}+l_{32}u_{22}&l_{31}u_{13}+l_{32}u_{23}+u_{33}&l_3u_{14}+l_{32}u_{24}+u_{34}\\l_{41}u_{11}&l_{41}u_{12}+l_{42}u_{22}&l_{41}u_{13}+l_{42}u_{23}+l_{43}u_{33}&l_{41}u_{14}+l_{42}u_{24}+l_{43}u_{34}+u_{44}\end{bmatrix}$$

采用right-looking算法，伪代码如下：

Algorithm 1 Hybrid Columen-Based Right-Looking Algorithm

1: **for** $k = 1$ to n **do**
2: /* *Compute column k of L matrix* */
3: **for** $i = k + 1$ to n where $A_s(i,k) \neq 0$ **do**
4: $A_s(i,k) = A_s(i,k)/A_s(k,k)$
5: **end for**
6: /* *Update the submatrix for next iteration* */
7: **for** $j = k + 1$ to n where $A_s(k,j) \neq 0$ **do**
8: **for** $i = k + 1$ to n where $A_s(i,k) \neq 0$ **do**
9: $A_s(i,j) = A_s(i,j) - A_s(i,k) * A_s(k,j)$
10: **end for**
11: **end for**
12: **end for**

加速方法：

在一轮迭代中（即最外层的循环一次中，下面以第一次循环为例），我们可以得到下列值，其中后三列的值计算仅依赖L的第一列，他们之间并没有依赖关系，所以在求出L第一列的值后，后面三列的值更新就可以并行计算；同时每一列中个个行元素的之间也是没有依赖关系的，同列不同行间也可以并行计算：

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12} & l_{21}u_{13} & l_{21}u_{14} \\ l_{31}u_{11} & l_{31}u_{12} & l_{31}u_{13} & l_{31}u_{14} \\ l_{41}u_{11} & l_{44}u_{12} & l_{41}u_{13} & l_4u_{14} \end{bmatrix}$$

* 2.7 代码实现

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "device_functions.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda.h>
#include <time.h>
#include <string>
#include <iostream>
#include <fstream>
using namespace std;
BLOCK_SIZE = 1024
//BLOCK_SIZE = 128
__global__ void  lu_decomposition_base(float *a)
{
  __shared__ float sharedBlock[BLOCK_SIZE][BLOCK_SIZE];

  for (int idx = threadIdx.x; idx<BLOCK_SIZE; idx += blockDim.x){
```

```
        sharedBlock[idx][threadIdx.y] =  a[idx*BLOCK_SIZE+threadIdx.y];
  }
  __syncthreads();

  int i,j;
  for(i=0; i < BLOCK_SIZE-1; i++)
   {
      if ( threadIdx.y>i && threadIdx.x==0 )//calculate L part
      {
        for(j=0; j < i; j++){
          sharedBlock[threadIdx.y][i] -= sharedBlock[threadIdx.y]
[j]*sharedBlock[j][i];
        }

      }
      else if(threadIdx.y>=i   && threadIdx.x==1 ) //calculate U part
      {
        for(j=0; j < i; j++){
          sharedBlock[i][threadIdx.y] -= sharedBlock[i][j]*sharedBlock[j]
[threadIdx.y];
        }
      }
      __syncthreads();
      if(threadIdx.y>i && threadIdx.x==0)//L part process
      {
        sharedBlock[threadIdx.y][i] /= sharedBlock[i][i];
      }
      __syncthreads();
  }
  for (int idx = threadIdx.x; idx<BLOCK_SIZE; idx += blockDim.x){
        a[idx*BLOCK_SIZE+threadIdx.y]=sharedBlock[idx][threadIdx.y];
  }
}

__global__ void  lu_decomposition_right_looking(float *a)
{
    __shared__ float sharedBlock[BLOCK_SIZE][BLOCK_SIZE];
    for (int idx = threadIdx.x; idx<BLOCK_SIZE; idx += blockDim.x)
    {
      sharedBlock[idx][threadIdx.y] =  a[idx*BLOCK_SIZE+threadIdx.y];
    }
    __syncthreads();

    for (int k=0; k< BLOCK_SIZE-1; k++)
    {
      if (threadIdx.y>k && threadIdx.x==0){
        sharedBlock[threadIdx.y][k]=sharedBlock[threadIdx.y][k]/sharedBlock[k]
[k];
      }
```

```
      __syncthreads();
      for (int idx = threadIdx.x; idx<BLOCK_SIZE; idx += blockDim.x)
      {
        if(idx>k && threadIdx.y>k){
            sharedBlock[idx][threadIdx.y] -= sharedBlock[idx][k]*sharedBlock[k]
[threadIdx.y];
        }
        __syncthreads();
      }
    }

    for (int idx = threadIdx.x; idx<BLOCK_SIZE; idx += blockDim.x){
        a[idx*BLOCK_SIZE+threadIdx.y]=sharedBlock[idx][threadIdx.y];
    }
}

int main(int argc, char *argv[]){
    float *a;
    float *b;
    float *x;
    float *y;
    float * l_u;
    int N = 1024;

    a = (float *)malloc(sizeof(float)*N*N);
    b = (float *)malloc(sizeof(float)*N);
    y = (float *)malloc(sizeof(float)*N);
    x = (float *)malloc(sizeof(float)*N);
    l_u = (float *)malloc(sizeof(float)*N*N);

    float * dev_a,dev_b;
    cudaMalloc ( (void*)&dev_a, N*N* sizeof (float) );
    //cudaMalloc ( (void*)&dev_b, N* sizeof (float) );

    cudaMemcpy(dev_a, a, N*N* sizeof (float), cudaMemcpyHostToDevice);

    dim3 dimGrid(N/BLOCK_SIZE);
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
    //LU decomposition
    lu_decomposition_right_looking<<<dimGrid, dimBlock>>>(dev_a);

    cudaMemcpy(l_u, dev_a, N*N* sizeof (float), ,cudaMemcpyDeviceToHost);

   // Ly=b && Ux = y, no parallel acceleration yet
    for (i = 1; i<N; i++){
        y[i] = 0;
        x[i] = 0;
    }
    y[0] = b[0];
```

```
    int i,k;
    for (i = 1; i<N; i++)
    {
        float sum = 0;
        for (k = 0; k<i; k++)
            sum += l_u[i * N + k] * y[k];
        y[i] = b[i] - sum;
    }

    x[N - 1] = y[N - 1] / l_u[(N - 1)*(N - 1)];
    for (i = N - 2; i >= 0; i--)
    {
        float sum = 0;
        for (k = i + 1; k<N; k++)
            sum += l_u[i * N + k] * x[k];
        x[i] = (y[i] - sum) / l_u[i * N + i];
    }

    cudaFree(dev_a);
    free(a);
    free(b);
    free(x);
    free(y);
    free(l_u);
}
```

3 奇异值分解

3.1 定义

SVD也是对矩阵进行分解，但是和特征分解不同，SVD并不要求要分解的矩阵为方阵。假设我们的矩阵A是一个m×n的矩阵，那么我们定义矩阵A的SVD为：

$$A = U\Sigma V^T$$

其中U是一个 $m \times m$ 的矩阵，Σ是一个 $m \times n$ 的矩阵，除了主对角线上的元素以外全为0，主对角线上的每个元素都称为奇异值，V是一个$n \times n$的矩阵。U和V满足
$U^TU = I, V^TV = IU^TU = I, V^TV = I$。

3.2 特征值和特征向量

A为 n 阶矩阵，若数λ和n维非0列向量x满足Ax=λx，那么数 λ 称为A的**特征值**，x 称为 A 的对应于特征值λ的**特征向量**。式$Ax = \lambda x$也可写成 $(A - \lambda E)x = 0$，并且$|\lambda E - A|$叫做A 的**特征多项式**。当特征多项式等于0的时候，称为A的特征方程，特征方程是一个齐次线性方程组，求解特征值的过程其实就是求解特征方程的解。SVD奇异值分解就是讲左右奇异奇异矩阵分别转换A^TA和 AA^T的特征方差进行计算，消除了对A矩阵是方阵的限制。

3.3 计算过程

对方程组

$$Ax = b$$

$$A = U\Sigma V^T = (U_1, U_2)\begin{pmatrix}\Sigma_1 \\ 0\end{pmatrix}V^H$$

记$V^H x = y, U_1^H b = b_1, U_2^H b = b_2$ 则

$$\| Ax - b \|_2^2 = \| U\Sigma y - b \|_2^2 = \| \Sigma y - U^H b \|_2^2 = \left\| \begin{pmatrix}\Sigma_1 \\ 0\end{pmatrix} y - \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right\|$$

当 A, b 给定，则$\| b_2 \|_2^2 = \| U_2^H b \|_2^2$ 是一个常数，为使得$\| Ax - b \|_2^2 = min,$

只需要 $\Sigma_1 y = b_1 \Leftrightarrow y = \Sigma_1^{-1} b_1 = \Sigma_1^{-1} U_1^H b$

最终得到线性方程组的最小二乘解

$$x = V\Sigma_1^{-1} U_1^H b = \sum_{k=1}^{N} \frac{u_k^H b}{\sigma_k} v_k$$

3.4 算法分析

这是标准的按照线性代数理论进行分解的方法，复杂度最高的操作即是矩阵乘法操作 $O(n^3)$, 所以时间复杂度是$max(m, n)^3$，可以通过并行，或者提取$Top\ k$ 特征的方式转化为 $O(n^2)$.

3.5 加速思路

奇异值分解目前大多时候不用于解线性方程组，因为他只能提取方程的前几个成分，而无法满足精确到指定位数的要求,目前主要用于图像降噪和数据特征降维。如果只要求前$Top\ k$个未知数的解，可以采用PCA主成分分析的方式进行加速。

对于奇异值,它跟我们特征分解中的特征值类似，在奇异值矩阵中也是按照从大到小排列，而且奇异值的减少特别的快，在很多情况下，前10%甚至1%的奇异值的和就占了全部的奇异值之和的99%以上的比例。也就是说，我们也可以用最大的k个的奇异值和对应的左右奇异向量来近似描述矩阵。也就是说：

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n} \approx U_{m\times k}\Sigma_{k\times k}V^T_{k\times n}$$

* 3.6 代码实现

- 头文件

  代码过长，具体可见于svd-impl.h

- 源文件

```
#include <matrix.h>
namespace splab
{

    template <typename Real>
    class SVD
```

```
    {

    public:

        SVD();
        ~SVD();

        void dec( const Matrix<Real> &A );
        Matrix<Real> getU() const;
        Matrix<Real> getV() const;
        Matrix<Real> getSM();
        Vector<Real> getSV() const;

        Real norm2() const;
        Real cond() const;
        int  rank();

    private:

        // the orthogonal matrix and singular value vector
        Matrix<Real> U;
        Matrix<Real> V;
        Vector<Real> S;

        // docomposition for matrix with rows >= columns
        void decomposition( Matrix<Real>&, Matrix<Real>&,
                            Vector<Real>&, Matrix<Real>& );

    };
    // class SVD


    #include <svd-impl.h>

}
// namespace splab


#endif
// SVD_H
```

- **使用 Eigen 矩阵库**

  求解最小二乘问题最精确的解法应该是`SVD`分解法。`Eigen`中提供了多种解法，官方推荐的是`BDCSVD`方法。下面直接看`SVD`方法使用示例，有如下超定方程组：

$$\begin{cases} 1x_1 + 2x_2 = 3 \\ 5x_1 - 3x_2 = 7 \\ 7x_1 + 10x_3 = 1 \end{cases}$$

对这个方程进行求解的代码如下：

```
# include<iostream>
# include<Eigen\Dense>

using namespace std;
using namespace Eigen;

int main(){
    MatrixXf A(3, 2);
    Vector3f b;
    Vector2f x;
    A << 1, 2, 5, -3, 7, 10;
    b << 3, 7, 1;
    x = A.bdcSvd(ComputeThinU | ComputeThinV).solve(b);
    cout << x << endl;
    system("pause");
    return 0;
}
```

最小二乘求解结果如下：

$$\begin{cases} x_1 = 1.02755 \\ x_2 = -0.56257 \end{cases}$$

```
这样得到的 $x$ 就是通过最小二乘算出来的。这里`.bcdScd()`函数里面的参数 `ComputeThinU | ComputeThinV` 必 须要写，否则会报。

将得到的解带回方程会发现其并不是严格成立的，即使对于不满足有解条件的方程组,也会给出一个近似,有时可能还会相差较大。这是因为对于超定方程，采用最小二乘法得出的解并不一定对每一个方程都严格成立，其确保的是当前解在所有方程上的总误差最小。
```