对矩阵乘法的加速及其在卷积神经网络中的应用

数学上，一个$m \times n$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。实验目的是尽可能的加速通用矩阵乘法。通用矩阵乘法（GEMM）通常定义为：

$$C = A \times B$$

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

一. 实验内容

1.基于算法的分析优化

①使用 Strassen 算法进行优化。Strassen 算法使用的是分治的思想，当矩阵的阶很大时会采取一个递推式进行计算相关递推式，描述如下：A_{11}，A_{12}，A_{21}，A_{22}分别为 A 的 4 个子矩阵，B、C 同理，计算如下表达式：

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \\ S_4 &= B_{21} - B_{11} \\ S_5 &= A_{11} + A_{22} \\ S_6 &= B_{11} + B_{22} \\ S_7 &= A_{12} - A_{22} \\ S_8 &= B_{21} + B_{22} \\ S_9 &= A_{11} - A_{21} \\ S_{10} &= B_{11} + B_{12} \end{aligned} \qquad \begin{aligned} P_1 &= A_{11} \times S_1 \\ P_2 &= S_2 \times B_{22} \\ P_3 &= S_3 \times B_{11} \\ P_4 &= A_{22} \times S_4 \\ P_5 &= S_5 \times S_6 \\ P_6 &= S_7 \times S_8 \\ P_7 &= S_9 \times S_{10} \end{aligned}$$

则可得到最终的矩阵结果：

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

具体代码见 Matrix_Strassen.cpp

②使用 Coppersmith－Winograd 算法进行优化。算法中符号表示与之前的 Strassen 相同，通过计算：

$$\begin{aligned} S1 &= A21 + A22 \\ S2 &= S1 - A11 \\ S3 &= A11 - A21 \\ S4 &= A12 - S2 \end{aligned} \qquad \begin{aligned} T1 &= B21 - B11 \\ T2 &= B22 - T1 \\ T3 &= B22 - B12 \\ T4 &= T2 - B21 \end{aligned}$$

$$\begin{aligned} M1 &= A11 * B11 \\ M2 &= A12 * B21 \\ M3 &= S4 * B22 \\ M4 &= A22 * T4 \\ M5 &= S1 * T1 \\ M6 &= S2 * T2 \\ M7 &= S3 * T3 \end{aligned} \qquad \begin{aligned} U1 &= M1 + M2 \\ U2 &= M1 + M6 \\ U3 &= U2 + M7 \\ U4 &= U2 + M5 \\ U5 &= U4 + M3 \\ U6 &= U3 - M4 \\ U7 &= U3 + M5 \end{aligned}$$

$$\begin{aligned} C11 &= U1 \\ C12 &= U5 \\ C21 &= U6 \\ C22 &= U7 \end{aligned}$$

可得到结果：

具体代码见 Matrix_Winograd.cpp

2. 使用 AVX256 指令进行循环拆分向量化的优化

AVX 指令集是 Sandy Bridge 和 Larrabee 架构下的新指令集。AVX 是在之前的 128 位扩展到 256 位的单指令多数据流。而 Sandy Bridge 的单指令多数据流演算单元扩展到 256 位的同时数据传输也获得了提升，所以从理论上看 CPU 内核浮点运算性能提升到了 2 倍。实验核心代码如下：

```
__m256d vec_res = _mm256_setzero_pd();
__m256d vec_1 = _mm256_setzero_pd();
__m256d vec_2 = _mm256_setzero_pd();
start=clock();
//printf("here1\n");
for (int a = 0; a < m; a++){
        for (int b = 0; b < n; b++){
                //printf("here2\n");
                vec_1 = _mm256_set1_pd(ma[a*n+b]);
                //printf("here3\n");
                for (int c = 0; c < k; c += 4){
                //printf("m=%d n=%d k=%d a=%d b=%d c=%d a*n+b=%d b*k+c=%d a*k+c=%d\n",m,n,k,a,b,c,a*n+b,b*k+c,a*k+c);
                //printf("%p\n",&mb[b*k+c]);
                //printf("here4\n");
                        vec_2 = _mm256_loadu_pd(&mb[b*k+c]);
                        //printf("%p\n",&mb[b*k+c]);
                        //printf("here5\n");
                        vec_res = _mm256_loadu_pd(&mc[a*k+c]);
                        //printf("here6\n");
                        vec_res = _mm256_add_pd(vec_res ,_mm256_mul_pd(vec_1, vec_2));
                        //printf("here7\n");
                        _mm256_storeu_pd(&mc[a*k+c], vec_res);
                        //printf("here8\n");
                }
        }
}
end=clock();
```

具体代码见 Matrix_Avx.cpp

3. 使用 MPI 进行运算优化

使用 MPI 集合通信方式，主进程将矩阵 A 按行划分为与进程数量相同的块并发送给其他进程，将矩阵 B 全部发送，每一个进程计算 A 的一块与 B 进行矩阵乘法的结果，并将其发送回主进程汇总得到结果。实验核心代码如下（主进程和其他进程）：

```
MPI_Scatter(a, line*n, MPI_DOUBLE, buffer, line*n, MPI_DOUBLE, 0, MPI_COMM_WORLD );
MPI_Bcast(b, n*k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (int i = 0; i < line; i++) {
        for (int j = 0; j < k; j++) {
                double temp = 0;
                for (int t = 0; t < n; t++)
                        temp += a[i * n + t] * b[t * k + j];
                ans[i * k + j] = temp;
        }

}
MPI_Gather( ans, line*k, MPI_DOUBLE, c, line*k, MPI_DOUBLE, 0, MPI_COMM_WORLD );
stop = MPI_Wtime();
if(isprint){
        printf("矩阵c：\n");
        for(int i=0;i<m;i++){
                for(int j=0;j<k;j++)
                        printf("%.2f \t",c[i*k+j]);
                printf("\n");
        }
}
```

```
double * buffer = new double [ n * line ];
MPI_Scatter(a, line*n, MPI_DOUBLE, buffer, line*n, MPI_DOUBLE, 0, MPI_COMM_WORLD );
MPI_Bcast( b, n * k, MPI_DOUBLE, 0, MPI_COMM_WORLD );
for(int i=0;i<line;i++){
        for(int j=0;j<k;j++){
                double temp=0;
                for(int t=0;t<n;t++)
                        temp += buffer[i*n+t]*b[t*k+j];
                ans[i*k+j] = temp;
        }
}
MPI_Gather(ans, line*k, MPI_DOUBLE, c, line*k, MPI_DOUBLE, 0, MPI_COMM_WORLD );
delete [] buffer;
```

具体代码见 Matrix_MPI.cpp

4. 使用 CUDA 和 cuBLAS 数学库对矩阵乘法进行优化

实验核心代码如下：

```
__global__ void gemm_gpu(double* a, double* b, double* c, int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    double tmp = 0;
    if (col<k && row<m)
    {
        for (int i = 0; i < n; i++)
        {
            tmp += a[row*n+i] * b[i*k+col];
        }
        c[row*k+col] = tmp;
    }
}
```

使用 cuBLAS 函数优化如下：

```
  cublasDgemm(
      handle,                 // blas 库对象
      CUBLAS_OP_T,            // 矩阵 A 属性参数
      CUBLAS_OP_T,            // 矩阵 B 属性参数
      M,                      // A, C 的行数
      M,                      // B, C 的列数
      N,                      // A 的列数和 B 的行数
      &a,                     // 运算式的 α 值
      device_A,               // A 在显存中的地址
      N,                      // lda
      device_B,               // B 在显存中的地址
      M,                      // ldb
      &b,                     // 运算式的 β 值
      device_C,               // C 在显存中的地址(结果矩阵)
      M                       // ldc
  );
```

具体代码见 Matrix_CUDA.cu 与 Matrix_cuBLAS.cu

5.使用 im2col 方法结合 GEMM 实现卷积操作

在信号处理、图像处理和其他工程和科学领域，卷积是一种广泛使用的技术。在深度学习领域，卷积神经网络（CNN）这种模型架构就得名于这种技术。操作流程可查看 CNN.gif。

实验使用 im2col 的方式对 Input 进行卷积,这里只实现 2D,height*width,通道 channel（depth）设置为 3，Kernel（Filter）大小设置为 3*3*3，个数为 3。不考虑 bias（b），bias 设置为 0，步幅（stride）分别设置为 1，2，3。im2col 核函数代码如下：

```
__global__ void im2colOnDevice(unsigned int n, float* matAc, float* matA, int radiusF, int countF, int L, int M, int K, int C, int H)
{
    for (int idx = blockIdx.x*blockDim.x+threadIdx.x; idx < n; idx += blockDim.x*gridDim.x)
    {
        int m = (idx/C)/L;
        int l = (idx/C)%L;
        int r = idx%C;
        if (m < M)
        {
            int w = m+radiusF;
            if (l < L)
            {
                int h = l+radiusF;
                for (int q = 0, oq = -1*radiusF; oq <= radiusF; q++, oq++)
                {
                    for (int p = 0, op = -1*radiusF; op <= radiusF; p++, op++)
                    {
                        if (r < C)
                        {
                            matAc[(r+C*(p+K*q))+countF*(l+L*m)] = matA[r+C*((h+op)+H*(w+oq))];
                        }
                    }
                }
            }
        }
    }
}
```

具体代码见 Matrix_im2col.cu

二．实验结果

实验的加速效果如下：

Strassen 算法：

```
scy@manager:~$ ./Matrix_Strassen
请依次输入m，n，k的值（范围512～2048）：1024 1024 1024
GEMM通用矩阵乘法已完成，用时：7336.917000 ms.
Strassen优化矩阵乘法已完成，用时：1937.480000 ms.
```

Winograd 算法：

```
scy@manager:~$ ./Matrix_Winograd
请依次输入m，n，k的值（范围512～2048）：1024 1024 1024
GEMM通用矩阵乘法已完成，用时：7402.908000 ms.
Coppersmith-Winograd优化矩阵乘法已完成，用时：1939.331000 ms.
scy@manager:~$
```

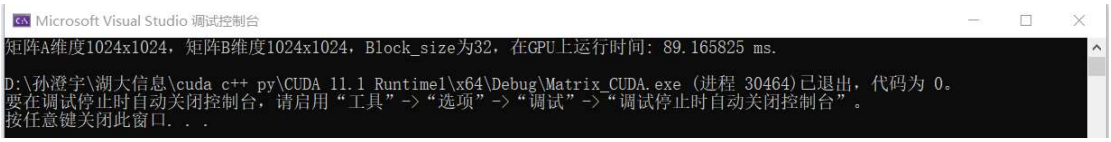
AVX 指令集优化：

```
scy@manager:~$ ./Matrix_Avx
1024 1024 1024
GEMM通用矩阵乘法已完成，用时：10892.801000 ms.
AVX优化矩阵乘法已完成，用时：1654.554000 ms.
scy@manager:~$
```

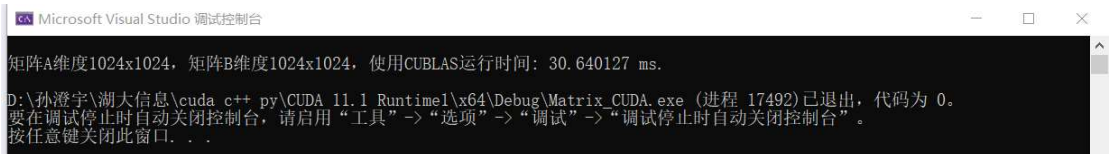
MPI 并行优化（分别运行 2,4,8,16 核心）：

```
scy@manager:~$ mpirun -np 8 ./Matrix_MPI 1024 1024 1024
计算所用时间：2804.502010 ms
scy@manager:~$ mpirun -np 16 ./Matrix_MPI 1024 1024 1024
计算所用时间：3292.614698 ms
scy@manager:~$ mpirun -np 4 ./Matrix_MPI 1024 1024 1024
计算所用时间：4073.179960 ms
scy@manager:~$ mpirun -np 2 ./Matrix_MPI 1024 1024 1024
计算所用时间：8062.340021 ms
scy@manager:~$ mpirun -np 1 ./Matrix_MPI 1024 1024 1024
计算所用时间：10312.184334 ms
scy@manager:~$
```

CUDA 优化：



CUDA cuBLAS 函数库优化：



im2col 方法结合优化 GEMM 方法实现卷积：

三．分析与反思

Strassen 算法和 Winograd 算法是对矩阵乘法在算法级的优化，截止目前时间复杂度最低的矩阵乘法算法是 Coppersmith-Winograd 方法的一种扩展方法，其算法复杂度为 $\theta(n^{2.375})$，代码实现可以发现，1024×1024 的矩阵相乘，两种算法约可以加速 3.8 倍。可以预期，矩阵维度越大，其加速效果越好。

AVX 是从指令集架构层面优化，主要针对的是浮点运算，可以看到 1024×1024 的矩阵相乘，使用 AVX 指令集编程约可以加速 6.6 倍。

MPI 是目前使用并行计算使用最多的模型，针对多核多节点进行并行加速，可以看到 1024×1024 的矩阵相乘，使用 8 核并行约可以加速 3.7 倍。8 核效果最好，因为我的计算机本身是 8 核，可以预期，矩阵维度越大，也就是数据越多，使用并行加速效果越好。

基于 CUDA 的 GPU 编程可以充分发挥 GPU 的计算性能，可以使程序使用成百上千的运算单元高度并行化，可以看到，在矩阵乘法中，CUDA 的计算性能远超 CPU，而经过优化的 cuBLAS 库函数又能使运算速度更快。1024×1024 的矩阵相乘仅需约 30ms。

如果能够将 MPI 与 GPU CUDA 编程结合，对大规模矩阵运算还会有更大的性能提升。