

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

批改人：

年级+班级	18 计科 8 班	专业（方向）	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成日期	2020 年 9 月 20 日

1. 实验目的(20 分)

熟悉 GEMM，并学习一些算法对计算进行优化以节省运行时间，并为以后做并行化打下基础。

2. 实验过程和核心代码(40 分)

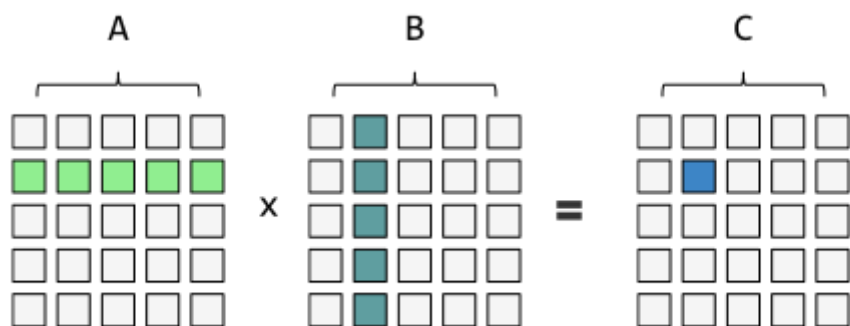
实验环境

操作系统：macOS

编译器：Apple clang version 11.0.3 (clang-1103.0.32.29)

问题 1

矩阵 A 规模为 $n \times m$ ，矩阵 B 规模为 $m \times p$ ，现需要你求 $A \times B$ 。矩阵相乘的定义： $n \times m$ 的矩阵与 $m \times p$ 的矩阵相乘变成 $n \times p$ 的矩阵，令 $a_{i,k}$ 为矩阵 A 中的元素， $b_{k,j}$ 为矩阵 B 中的元素，则相乘所得矩阵 C 中的元素。乘法的方式如下：



即对于 C 中的每个元素 c_{ij} ，可以用下面的式子计算：

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}$$

在这里我用 C 语言的 malloc 申请的二维数组来存储矩阵。然后调用我写好的矩阵乘法的函数，对结果进行计算。计算的结果同样存储在用 C 语言的 malloc 申请的二维数组中。

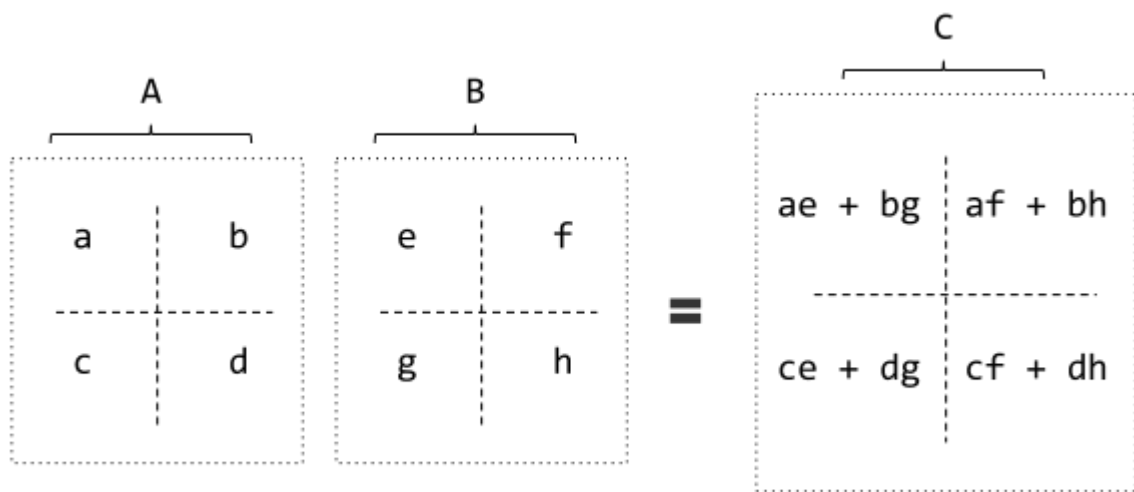
核心的代码如下：

```
void calculate(int ** Matrix_a,int ** Matrix_b,int ** res,int m,int k,int n){
    for(int i = 0;i < m;i++){
        for(int j = 0;j < k;j++){
            for(int q = 0;q < n;q++){
                res[i][j] += Matrix_a[i][q] * Matrix_b[q][j];
            }
        }
    }
}
```

并且我在计算函数的前后都加入了 gettimeofday 函数，用于测量计算所需花费的时间。

问题 2

在这里我使用了 Strassen 算法对矩阵乘法进行优化。由于对矩阵做乘法的时间和开销远远大于对矩阵做加法所需的时间和开销，所以应该尽可能减少乘法在整个运算中的比重。Strassen 算法将方阵划分为 4 等分，然后对 4 个 $\frac{1}{4}$ 矩阵只需要做 7 次矩阵乘法 and 很多次矩阵加法就可以完成运算，相当于将部分的矩阵乘法转换成了矩阵加法，将矩阵乘法的时间复杂度从 $O(n^3)$ 降到了 $O(n^{\log 7})$ 。计算的方法如下：



具体的实现比较复杂，核心部分的代码如下：

```
add( A11,A22,AResult, HalfSize);
add( B11,B22,BResult, HalfSize);
Strassen( HalfSize, AResult, BResult, M1 );
add( A21,A22,AResult, HalfSize);
Strassen(HalfSize, AResult, B11, M2);
sub( B12,B22,BResult, HalfSize);
Strassen(HalfSize, A11, BResult, M3);
sub( B21, B11, BResult, HalfSize);
Strassen(HalfSize, A22, BResult, M4);
add( A11, A12, AResult, HalfSize);
Strassen(HalfSize, AResult, B22, M5);
sub( A21, A11, AResult, HalfSize);
add( B11, B12, BResult, HalfSize);
Strassen( HalfSize, AResult, BResult, M6);
sub(A12, A22, AResult, HalfSize);
add(B21, B22, BResult, HalfSize);
Strassen(HalfSize, AResult, BResult, M7);
add( M1, M4, AResult, HalfSize);
sub( M7, M5, BResult, HalfSize);
add( AResult, BResult, C11, HalfSize);
add( M3, M5, C12, HalfSize);
add( M2, M4, C21, HalfSize);
add( M1, M3, AResult, HalfSize);
sub( M6, M2, BResult, HalfSize);
add( AResult, BResult, C22, HalfSize);
```

也就是我们对小矩阵进行乘法和加法的过程。

问题 3

在这里可以采用 MapReduce 的编程模型，对于大规模的矩阵 M 和 N 相乘，先做一次 Map 操作，对每个矩阵元素 m_{ij} 产生键值对 $(j, (M, i, m_{ij}))$ ，对每

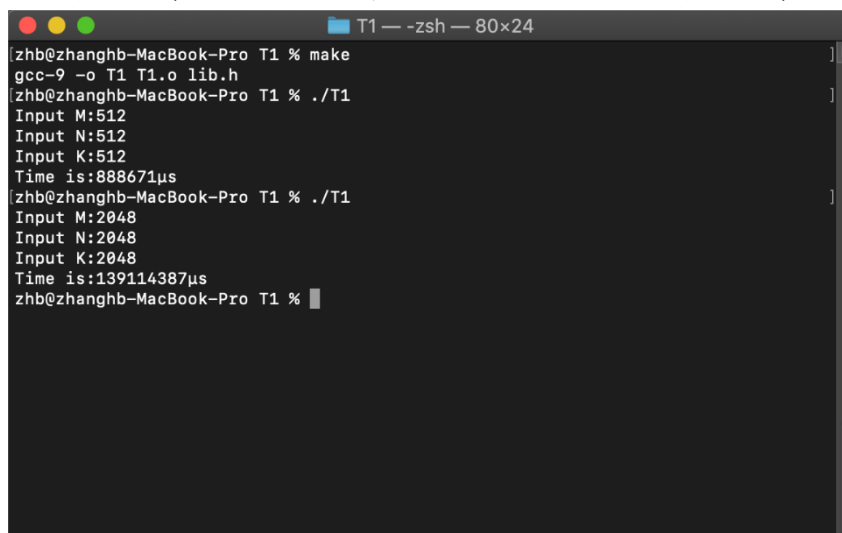
个矩阵元素 n_{jk} 产生键值对 $(j, (N, k, n_{jk}))$ 。再做一次 Reduce 操作：对每个键 j ，对 (M, i, m_{ij}) 和 (N, k, n_{jk}) ，产生键值对 $((i, k), m_{ij}n_{jk})$ 。最后再做一次 Reduce 操作，对每个键 (i, k) ，计算与此键相关联的所有值的和，得到结果。

MapReduce 本身就是为大数据设计的，因此擅长处理这种大规模的问题。

3. 实验结果(30 分)

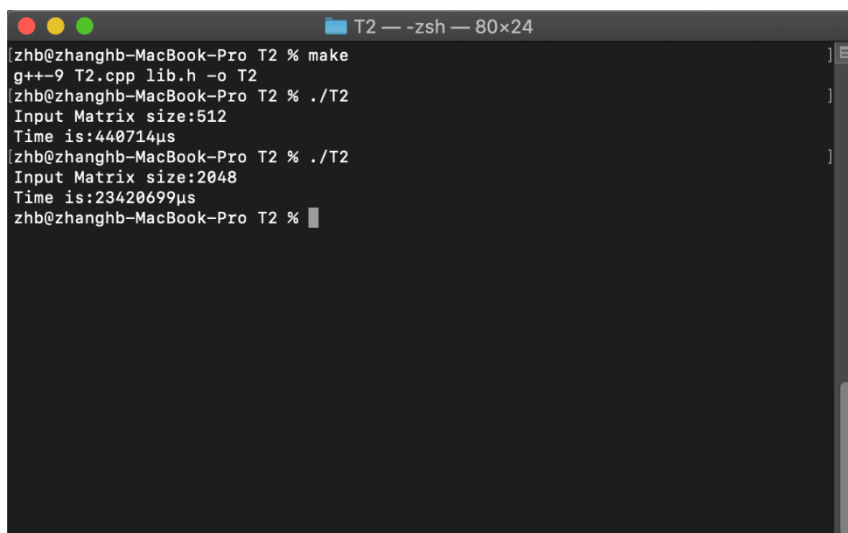
问题 1 的结果

在命令行输入 make 后，输入问题的规模，在这里我运行了两次，一次让两个 512*512 的方阵相乘，一次让两个 2048*2048 的方阵相乘，结果如下：



```
T1 — -zsh — 80x24
zhb@zhanghb-MacBook-Pro T1 % make
gcc-9 -o T1 T1.o lib.h
zhb@zhanghb-MacBook-Pro T1 % ./T1
Input M:512
Input N:512
Input K:512
Time is:888671µs
zhb@zhanghb-MacBook-Pro T1 % ./T1
Input M:2048
Input N:2048
Input K:2048
Time is:139114387µs
zhb@zhanghb-MacBook-Pro T1 %
```

问题 2 的结果



```
zhb@zhanghb-MacBook-Pro T2 % make
g++-9 T2.cpp lib.h -o T2
zhb@zhanghb-MacBook-Pro T2 % ./T2
Input Matrix size:512
Time is:440714µs
zhb@zhanghb-MacBook-Pro T2 % ./T2
Input Matrix size:2048
Time is:23420699µs
zhb@zhanghb-MacBook-Pro T2 %
```

可以看到，使用 Strassen 算法所需要的时间开销远远小于 GEMM 的时间开销。说明我们的优化效果较好。

4. 实验感想(10 分)

这次实验的难点主要在于对矩阵乘法的优化，虽然我采用的 Strassen 算法思路比较简单，但是具体实现的时候还是比较麻烦的，尤其是对矩阵进行划分的时候非常容易出错。

这次实验虽然不难，但是也让我了解了 MapReduce 的工作原理，并且让我熟悉了 Makefile 的使用，以及了解了一些基本的优化的方法，为接下来的实验打下一定的基础。