中山大学数据科学与计算机院本科生实验报告

（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 级计科 7 班	专业（方向）	计算机科学与技术
学号	18340166	姓名	王若琪
Email	Wangrq29@mail2.sysu.edu.cn	完成日期	2020.9.23

1. 实验目的(20 分)

（1）用 C 语言实现通用矩阵乘法，并记录计算的时间。

（2）实现用基于算法分析的方法对矩阵乘法进行优化，可选择 Strassen 算法或 Coppersmith–Winograd 算法，并与 GEMM 进行对比分析。

（3）实现用基于软件优化的方法对矩阵进行优化，如循环拆分向量化或内存重排，并与 GEMM 进行对比分析。

（4）考虑大规模矩阵计算优化，从性能方面和可靠性方面进行思考与分析。

（5）熟悉 linux 实验环境，学习、掌握、运用 makefile 的方法来编译实验代码。

（6）通过分析与实践各种矩阵乘法的优化方法，对比并分析实验结果，学习高性能计算编程优化的基本思想，熟悉高性能编程实验的实验环境和实验方法。

2. 实验过程和核心代码(40 分)

2.1 通用矩阵乘法的实现

首先，通过两次调用 initMatrix()函数，随机生成 M×N 和 N×K 的两个矩阵 A，B，由于需要防止结果超出范围，所以需要设置随机数的范围为 0~50 之间的整数。其中，initMatrix()函数核心代码如下，此函数可根据参数中的 r 行数，c 列数分配矩阵空间，并初始化矩阵值：

```
int** initMatrix(int r, int c) {  //初始化矩阵
    int **temp=(int**)malloc(sizeof(int*)*r);
    for (int i=0;i<r;++i)
        temp[i]=(int *)malloc(sizeof(int)*c);

    for(int i = 0; i < r; i++){
        for (int j = 0; j < c; j++){
            temp[i][j] = rand()%50;
        }
    }
    return temp;
}
```

接下来编写 gemm()函数来实现通用矩阵乘法。通用矩阵乘法的思路大致可用计算式表示为：

$$(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{ip} b_{pj}$$

我编写的 gemm()函数的主要功能为，先为结果矩阵申请空间，再实现通用矩阵乘法，求得结果，返回计算结果矩阵的指针，核心代码如下：

```
int** gemm(int** matrixA, int** matrixB) {   //通用矩阵乘法
    int **matrix=(int**)malloc(sizeof(int*)*M);
    for (int i=0;i<M;++i)
        matrix[i]=(int *)malloc(sizeof(int)*K);
    for (int i = 0; i < M; i++){
        for (int j = 0; j < K; j++){
            matrix[i][j] = 0;
            for (int l = 0; l < N; l++){
                matrix[i][j] += matrixA[i][l] * matrixB[l][j];
            }
        }
    }
    return matrix;
}
```

之后，根据题目要求添加输出矩阵和计算时间等功能，其中时间计算用到了<time.h>中的 clock()函数，此步骤较为简单，具体代码请见源文件 gemm.c，这里不做赘述。

2.2 通用矩阵乘法优化

（1）基于算法分析的矩阵乘法优化

选择 Strassen 算法对矩阵乘法进行优化。

Strassen 算法是一种基于分治思想的算法，Strassen 算法将矩阵分拆为更小的矩阵，其中有要求为 A、B、C 为行列数为 2 的整数次幂的方阵：

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

在分块后，Strassen 引入了七个如下所示的七个辅助中间矩阵：

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

$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\,\mathbf{B}_{2,2}$$
$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})\,(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$
$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})\,(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

最后再进行对中间结果的整合，得到最后的矩阵：

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

Strassen 方法实现的核心代码如下，主要分为拆分、分块计算、整合三大步骤，其中具体 divide(), subtractMat(), addMat(), concatMatrices()等函数的实现较为繁琐，具体请参考源代码 strassen.c：

```
int** Strassen(int** matrixA, int** matrixB, int size){  //strassen优化方法
    if(size <= 64){
        return gemm_sq(matrixA, matrixB, size);
    }
    else{
        int** a = divide(matrixA, size, 1);
        int** b = divide(matrixA, size, 2);
        int** c = divide(matrixA, size, 3);
        int** d = divide(matrixA, size, 4);
        int** e = divide(matrixB, size, 1);
        int** f = divide(matrixB, size, 2);
        int** g = divide(matrixB, size, 3);
        int** h = divide(matrixB, size, 4);

        int** p1 = Strassen(a, subtractMat(f, h, size/2), size/2);
        int** p2 = Strassen(addMat(a, b, size/2), h, size/2);
        int** p3 = Strassen(addMat(c, d, size/2), e, size/2);
        int** p4 = Strassen(d, subtractMat(g, e, size/2), size/2);
        int** p5 = Strassen(addMat(a, d, size/2), addMat(e, h, size/2), size/2);
        int** p6 = Strassen(subtractMat(b, d, size/2), addMat(g, h, size/2), size/2);
        int** p7 = Strassen(subtractMat(a, c, size/2), addMat(e, f, size/2), size/2);
        int** A = addMat(addMat(p4, p5, size/2), subtractMat(p6, p2, size/2), size/2);
        int** B = addMat(p1, p2, size/2);
        int** C = addMat(p3, p4, size/2);
        int** D = addMat(subtractMat(p1, p3, size/2), subtractMat(p5, p7, size/2),
size/2);
        return concatMatrices(A, B, C, D, size/2);
    }
}
```

此外，用 Strassen 算法时，需要注意先把矩阵通过补 0，扩展成为边长为 2 的整数次幂的方阵，才能够方便使用分治方法。并且，当分块小到了一个临界值（此处设定为 64）时，如果继续拆分会导致效率低下，于是在分到边长为 64 的时候，开始使用 GEMM 方法计算结果。

（2）基于软件优化方法的矩阵乘法优化

选用循环计算拆分法，对矩阵乘法的运算进行拆分，以求改进程序运行时内存访问的局部性。

①在 K 维度上进行拆分展开的代码如下，可以看出这里最内侧计算使用的 matrix 元素是一定的，所以可以实现 4 次数据复用：

```
int** loop_divide_1(int** matrixA, int** matrixB) {   //第一种拆分方法，K维度上进行拆分展开
    int **matrix=(int**)malloc(sizeof(int*)*M);
    for (int i=0;i<M;++i)
        matrix[i]=(int *)malloc(sizeof(int)*K);
    for (int i = 0; i < M; i++){
        for (int j = 0; j < K; j+=4){
            matrix[i][j+0] = 0;
            matrix[i][j+1] = 0;
            matrix[i][j+2] = 0;
            matrix[i][j+3] = 0;
            for (int l = 0; l < N; l++){
                matrix[i][j+0] += matrixA[i][l] * matrixB[l][j+0];
                matrix[i][j+1] += matrixA[i][l] * matrixB[l][j+1];
                matrix[i][j+2] += matrixA[i][l] * matrixB[l][j+2];
                matrix[i][j+3] += matrixA[i][l] * matrixB[l][j+3];
            }
        }
    }
    return matrix;
}
```

②继续拆分输出的 M 维度，从而在内侧循环中计算 4*4 的输出，伪代码如下，其中 0..3 代表代入值 0、1、2、3 各执行一次。为了提高性能，所以具体代码篇幅较长，具体代码详见源代码 loop_divide.c 中的 loop_divide_2()函数。伪代码：

```
for (int i = 0; i < M; i+=4){  //伪代码
        for (int j = 0; j < K; j+=4){
            matrix[i+0][j+0..3] = 0;
            matrix[i+1][j+0..3] = 0;
            matrix[i+2][j+0..3] = 0;
            matrix[i+3][j+0..3] = 0;
            for (int l = 0; l < N; l++){
                matrix[i+0][j+0] += matrixA[i+0][l] * matrixB[l][j+0..3];
                matrix[i+1][j+0] += matrixA[i+1][l] * matrixB[l][j+0..3];
                matrix[i+2][j+0] += matrixA[i+2][l] * matrixB[l][j+0..3];
                matrix[i+3][j+0] += matrixA[i+3][l] * matrixB[l][j+0..3];
            }
        }
    }
```

③继续拆分 N 维度，原理同上，伪代码如下，其中 0..3 代表代入值 0、1、2、3 各执行一次。为了提高性能，所以具体代码篇幅较长，具体代码详见源代码 loop_divide.c 中的 loop_divide_3()函数。伪代码：

```
for (int i = 0; i < M; i+=4){  //伪代码
    for (int j = 0; j < K; j+=4){
        matrix[i+0][j+0..3] = 0;
        matrix[i+1][j+0..3] = 0;
        matrix[i+2][j+0..3] = 0;
        matrix[i+3][j+0..3] = 0;
        for (int l = 0; l < N; l+=4){
            matrix[i+0][j+0] += matrixA[i+0][l+0..3] * matrixB[l+0..3][j+0..3];
            matrix[i+1][j+0] += matrixA[i+1][l+0..3] * matrixB[l+0..3][j+0..3];
            matrix[i+2][j+0] += matrixA[i+2][l+0..3] * matrixB[l+0..3][j+0..3];
            matrix[i+3][j+0] += matrixA[i+3][l+0..3] * matrixB[l+0..3][j+0..3];
        }
    }
}
```

2.3 进阶：大规模矩阵计算优化

（1）性能，提高大规模稀疏矩阵乘法性能

答：由于题目中要考虑的是大规模的稀疏矩阵，所以可以从矩阵的稀疏性出发，优化程序性能。由于稀疏矩阵中非零元素较少，零元素较多，因此可以采用只存储非零元素的方法来进行压缩存储。

由于非零元素分布没有任何规律，所以在进行压缩存储的时侯需要存储非零元素值的同时还要存储非零元素在矩阵中的位置，即非零元素所在的行号和列号，也就是在存储某个元素比如 a[i][j]的值的同时，还需要存储该元素所在的行号 i 和它的列号 j，这样就构成了一个三元组(i,j,a[i][j])的线性表。

三元组可以采用顺序表示方法，也可以采用链式表示方法，这样就产生了对稀疏矩阵的不同压缩存储方式。

可以使用这种方法，在做乘法时，就无须考虑结果为 0 的运算，可以节省许多内存和时间，实现大规模稀疏矩阵乘法性能上的优化。

（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。

答：因为内存有限，为防止溢出，可以将大规模矩阵进行拆分，拆分成许多个小矩阵再作乘法，最后再将结果整合在一起即可。也可以通过 spark 等并行架构来实现。

3. 实验结果(30 分)

3.1 通用矩阵乘法的实现结果

编译 gemm.c，运行程序 gemm，输入 M、N、K 的值，运行结果如下：先输入 M、N、K 为 800、900、1024：

```
wrq@wrq-virtual-machine:~/hw1$ ./gemm
Please enter 3 integers (512~2048) :
800 900 1024
```

输入后开始等待运算，运算结束后，按照实验要求，先分别输出矩阵的值，最后输出运行时间。

输出矩阵截图（部分）：

```
 547824 550456 538285 538363 534223 533387 546732 536656 540093 528930 545345 55
1309 532476 535413 568888 535131 523083 553863 546433 554908 547416 571348 54466
7 551818 547815 536230 539472 536548 543923 566476 533414 548423 544776 541689 5
31217 552283 546532 535950 558080 560095 566273 538354 546540 543652 537084 5555
89 571549 552319 525128 558044 556229 538580 532970 513111 573167 579595 535906
537135 565963 549904 552993 552559 573675 561687 545955 556486 554177 545629 528
261 533077 553413 541900 555646 570327 560170 529400 542356 564698 533113 525082
 545814 557614 550934 551434 551896 552828 550050 546627 584705 527801 537687 55
7443 553028 532594 493834 550258 571241 541008 554736 568723 538303 551798 54857
7 569889 542290 520344 558101 538873 535791 545937 542018 563660 576144 539589 5
57970 548134 534301 554509 546992 570968 565796 540552 551633 539882 550080 5479
10 527298 558521 561373 541016 547321 557730 536126 544083 552994 554266 548056
554439 546609 536234 516725 552686 556742 547349 550572 560376 554078 563047 565
062 527450 550600 550206 558887 551459 546572 555809 533490 578188 557553 557319
 554906 542108 560682 572317 538804 539777 540415 541013 530779 549200 557364 56
6787 555989 536018 550083 559192 543567 559218 549039 520105 563416 526526 57107
1 535228 563955 554984 573279 550832 539496 570215 541107 561079 543628 553696 5
```

输出运行时间截图：

```
time of gemm: 15.334545 s
```

可见，用 GEMM 求 800*900 以及 900*1024 的两个矩阵的积，本次用时大约在 15.334545 秒左右。

3.2 通用矩阵乘法优化结果

（1）Strassen 方法运行结果及比较

因为 Strassen 方法需要给不规则矩阵补 0，使其变为 2 的整数次幂边长的方阵，所以用 Strassen 方法求矩阵与 GEMM 相比较，对于不同类型的矩阵会有不同的优化效果，现结果举例和分析如下。

在同一程序同一次运行中，分别用 GEMM 和 Strassen 方法求 800*900 以及 900*1024 的两矩阵乘积，结果如下图：

```
wrq@wrq-virtual-machine:~/hw1$ ./strassen
Please enter 3 integers (512~2048) :
800 900 1024
time of gemm:     14.753640 s
time of strassen: 7.955359 s
```

这种情况下，加速比约为 1.85.

分别用 GEMM 和 Strassen 方法求 1024*1024 以及 1024*1024 的两矩阵乘积，结果如下图：

```
wrq@wrq-virtual-machine:~/hw1$ ./strassen
Please enter 3 integers (512~2048) :
1024 1024 1024
time of gemm:     22.654827 s
time of strassen: 9.994885 s
```

这种情况下，加速比约为 2.27.

分别用 GEMM 和 Strassen 方法求 600*1020 以及 1020*512 的两矩阵乘积，结果如下图：

```
wrq@wrq-virtual-machine:~/hw1$ ./strassen
Please enter 3 integers (512~2048) :
600 1020 512
time of gemm:     7.099572 s
time of strassen: 9.550345 s
```

这种情况下，加速比仅为 0.74.

分别用 GEMM 和 Strassen 方法求 512*600 以及 600*520 的两矩阵乘积，结果如下图：

```
wrq@wrq-virtual-machine:~/hw1$ ./strassen
Please enter 3 integers (512~2048) :
512 600 520
time of gemm:     3.138941 s
time of strassen: 7.907135 s
```

这种情况下，加速比仅为 0.40.

对比这四次实验：

M	N	K	T_gemm	T_Strassen	加速比
800	900	1024	14.753640	7.955359	1.85
1024	1024	1024	22.654827	9.994885	2.27
600	1020	512	7.099572	9.550345	0.74
512	600	520	3.138941	7.907135	0.40

可以对比分析出，当矩阵规模较大，长宽差距较小，并且稍小于或者等于 2 的幂次方边长的矩阵时，Strassen 算法的优化效果明显。然而当矩阵规模较小时分治的开销占比增大，或者某边长稍大于 2 的幂次方时，由于 Strassen 算法需要补很多 0，所以开销增大，所以这些情况下 Strassen 的优化效果不好，甚至会比 GEMM 更差。

（2）基于软件优化方法的结果

针对上文中的三种循环拆分模式进行测试和比较，结果如下：

M、N、K 分别为 520、540、560 时的运行结果：

```
wrq@wrq-virtual-machine:~/hw1$ ./loop_divide
Please enter 3 integers (512~2048) :
520 540 560
time of gemm:       3.575252 s
time of loop_divide_1: 2.883610 s
time of loop_divide_2: 2.557175 s
time of loop_divide_3: 2.882375 s
```

M、N、K 分别为 1000、1100、1200 时的运行结果：

```
Please enter 3 integers (512~2048) :
1000 1100 1200
time of gemm:       27.444235 s
time of loop_divide_1: 22.225538 s
time of loop_divide_2: 16.206596 s
time of loop_divide_3: 18.080817 s
```

M、N、K 分别为 1300、1400、1500 时的运行结果:

```
wrq@wrq-virtual-machine:~/hw1$ ./loop_divide
Please enter 3 integers (512~2048) :
1300 1400 1500
time of gemm:       60.647134 s
time of loop_divide_1: 50.368103 s
time of loop_divide_2: 35.424102 s
time of loop_divide_3: 56.899989 s
```

对比以上三次实验:

M	N	K	T_gemm	T_ld_1	T_ld_2	T_ld_3	加速比 1	加速比 2	加速比 3
520	540	560	3.57525	2.88361	2.55718	2.88238	1.24	1.40	1.24
1000	1100	1200	27.44424	22.22554	16.20660	18.08082	1.23	1.69	1.52
1300	1400	1500	60.64713	50.36810	35.42410	56.89999	1.20	1.71	1.07

通过表格对比，可以看出每一种循环拆分方法都对程序性能有着较为明显的优化效果，其中第二种拆分方法优化效果最好。

3.3 进阶：大规模矩阵计算优化

具体算法设计及分析请见上文 2.3。

4. 实验感想(10 分)

本次实验过程中遇到了一些值得记录的问题和思考，大致总结为以下三点:

（1）在实现 Strassen 算法时，由于要在不足 2 的整数次幂的矩阵中补 0，所以如何求得大于某个数的最小 2 的整数次幂的值就难到了我，通过查阅资料。我学会并在此次实验中运用了位操作的方法来求这个数，非常巧妙方便。

（2）Strassen 算法拆分的粒度也是需要考虑的问题，如何设置不继续拆分的界限，需要不断地去实验尝试，经过多次测试，我发现以 64 为不继续拆分的界限效果最好。

（3）循环拆分法是我通过这次实验学到的新方法，之前只是理论上地学习过访存局部性，这次将访存局部性应用到实践中去，达到了优化程序性能的效果，并且实现也比较简单，这是我学到的新知识。

总而言之，第一次做高性能程序设计实验，我感到自己之前学习到的基本的编程方式还是有很大的优化空间，之后的编程实践中，我可能不会仅仅把正确性作为目的和指标，而是会多多考虑自己设计程序的性能，有时候改变一点算法或思路，程序性能真的能提升很多，我需要更多地实践和学习。