高性能计算导论实验1

通用矩阵乘法基于串行编程的实现及优化

秋季2022

# 总述

与矩阵相关的函数，比如创建矩阵，初始化矩阵等见matrix\_tool.cpp。

本人完成实验亮点：

1. 学习chrono库进行高精度计时。
2. 工程化思维，工程中优通用库，多个主函数完成不同需求。中级makefile文件编写，涉及自动依赖生成，vpath和环境变量，windows和linux跨平台的makefile文件。windows出现的make固有异常以及解决方案。
3. 实现了同一个二维数组，支持A[i\*stride+j]和A[i][j]两种访问模式。妈妈再也不同担心我因为接口不同、二维数组传参问题发愁啦。
4. 作图展示和分析数据。
5. 软件优化以及稀疏矩阵的讲解深入浅出，篇幅比参考文章短，却更容易让人理解。

# 通用矩阵乘法

## 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。

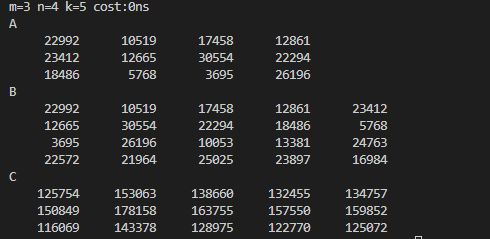
## 编写思路：

1. 使用vector存储矩阵
2. 使用c++标准库chrono来计时（需要学习）
3. 定义一个函数来用随机数填充矩阵。
4. 定义一个函数来用通用矩阵乘法计算。入参为矩阵A,B,出参为矩阵C。
5. A,B,C的大小都要事先固定。
6. 考虑到后续会不断添加文件，编写makefile指导编译。Makefile涉及的难点有：自动依赖生成，vpath和环境变量，windows和linux跨平台的makefile文件。windows出现的make固有异常。因为自动依赖生成和默认规则结合在一起犯了错。

## 代码见：

算法实现函数见universial.cpp，主函数见main1.cpp。

当m=3,n=4,k=5时运行结果，用时0ns，这个是由于数据过小，运行时间少于最高计时精度造成的。



当m=100 n=100 k=100 用时14ms



# 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](https://en.wikipedia.org/wiki/Strassen_algorithm)和 [Coppersmith–Winograd 算法](https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm).
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Strassen，Coppersmith–Winograd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。

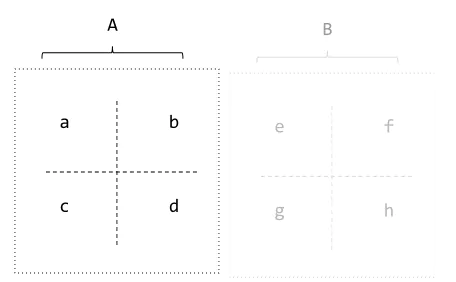
## 难点：

由于现成的代码接口不同，有些是用一维数组表示二维数组，有些是用支持形如A[i][j]这样操作的二维数组，代码量非常庞大，统一接口不太现实，于是我想到了一个开辟内存的方法，使得这个数组既可以支持用一维数组表示二维数组的操作，A[i\*N+j]，又支持A[i][j]操作。

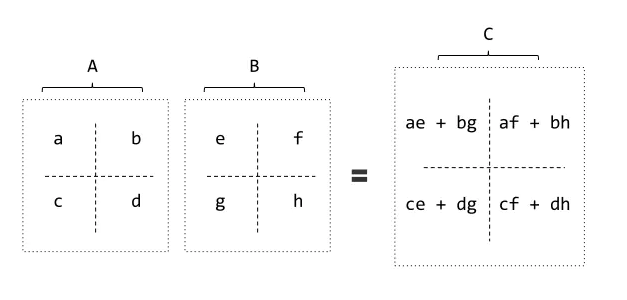
## 基于算法

### Strassen:

一般说来，当数据量较大时，我们往往会把大的数据分割成小的数据，各个分别处理。遵此思路，如果给我们一个很大的两个矩阵呢，是否可以考虑分治的方法循序渐进处理各个小矩阵的相乘，因为我们知道一个矩阵是可以分成更多小的矩阵的。  
如下图，当给定一个两个二维矩阵A B时：



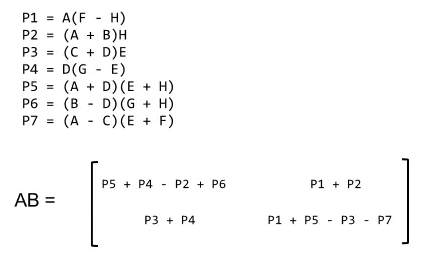
这两个矩阵A B相乘时，我们发现在相乘的过程中，有8次乘法运算，4次加法运算：



矩阵乘法的复杂度主要就是体现在相乘上，而多一两次的加法并不会让复杂度上升太多。

可以让矩阵乘法的运算过程中乘法的运算次数减少，从而达到降低矩阵乘法的复杂度。

用上文A B两个矩阵相乘的例子演示Strassen，定义了7个变量



这样计算只需要计算7次规模减半的乘法和若干次加法。时间复杂为

T(n/2)+O(n2) 展开为O(nlog27) 。

代码见：Strassen.cpp

### Coppersmith–Winograd

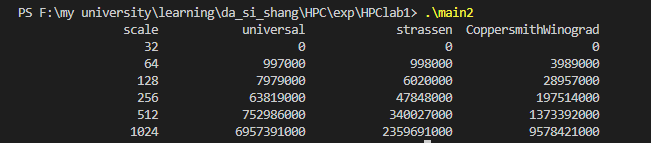
理解难度过大，找到现成代码，修改为与Stassen和GEMM代码相同的接口，用于时间对比。

代码见CoppersmithWinograd.cpp

参考自：<https://github.com/YYYYYW/Matrix-Multiplication/blob/main/MatrixMultiplication.cpp>

### 测试结果

不知道是因为这个作者没实现好还是其他什么的原因，Coppersmith-Winograd效果很差，比通用矩阵乘法用时都多。



单位 ns

|  |  |  |  |
| --- | --- | --- | --- |
| scale | universal | strassen | CoppersmithWinograd |
| 32 | 0 | 0 | 0 |
| 64 | 997000 | 998000 | 3989000 |
| 128 | 7979000 | 6020000 | 28957000 |
| 256 | 63819000 | 47848000 | 197514000 |
| 512 | 752986000 | 340027000 | 1373392000 |
| 1024 | 6957391000 | 2359691000 | 9578421000 |

## 基于软件

### 讲解分析（定量）

for(int i=0;i<m;i++)

    {

        for(int j=0;j<k;j++)

        {

            C[i][j]=0;

            for(int s=0;s<n;s++)

            {

                C[i][j]+=A[i][s]+B[s][j];

            }

        }

    }

这里，总共执行m\*n\*k次C[i][j]+=A[i][s]+B[s][j];

这条语句对C矩阵访问两次（一次读一次写），A矩阵访问一次，B矩阵访问一次。整个代码段会访存(2+1+1)\*(m\*n\*k)=4mnk次。

通过循环展开，可以减少访存次数。

先看伪代码：

**for** (**int** m **=** 0; m **<** M; m **+=** 4)

{

**for** (**int** n **=** 0; n **<** N; n **+=** 4)

{

C[m **+** 0..3][n **+** 0..3] **=** 0;

C[m **+** 0..3][n **+** 0..3] **=** 0;

C[m **+** 0..3][n **+** 0..3] **=** 0;

C[m **+** 0..3][n **+** 0..3] **=** 0;

**for** (**int** k **=** 0; k **<** K; k **+=** 4)

{

C[m **+** 0..3][n **+** 0..3] **+=** A[m **+** 0..3][k **+** 0] **\*** B[k **+** 0][n **+** 0..3];

C[m **+** 0..3][n **+** 0..3] **+=** A[m **+** 0..3][k **+** 1] **\*** B[k **+** 1][n **+** 0..3];

C[m **+** 0..3][n **+** 0..3] **+=** A[m **+** 0..3][k **+** 2] **\*** B[k **+** 2][n **+** 0..3];

C[m **+** 0..3][n **+** 0..3] **+=** A[m **+** 0..3][k **+** 3] **\*** B[k **+** 3][n **+** 0..3];

}

}

}

证明该代码的正确性略，下面计算它的访存次数。

循环总共进行（m/4)\*(n/4)\*(k/4)=(1/64)mnk次。

看到最内层的循环：

研究C矩阵，在不考虑编译器优化的情况下，C矩阵被访问了32\*4=128次（每行4\*4=16次读，4\*4=16次写）。如果考虑编译器优化，最内层循环只是对16个数进行操作，也就是

C[m **+** 0..3][n **+** 0..3]这16个数。编译器完全可以把这16个数放到寄存器中，那么只需要进行32次访存，（16次读，16次写）。再仔细观察，发现这16个数是不随k变化而变化的，总体来看，对C矩阵的访问，只进行了32\*m\*n次访问（其中16\*m\*n次从内存读到寄存器，16\*m\*n次把寄存器的值写回内存）。

研究A矩阵，共进行了16次内存读。B矩阵也是，共进行了16次内存读。

因此总体上，进行了

32\*m\*n次访问C矩阵内存，16\*[(1/64)mnk]=1/4mnk次A矩阵内存读，16\*[(1/64)mnk]=1/4mnk次B矩阵内存读。

共1/2mnk+32\*mn次内存。比原来的4mnk次内存访问要快接近8倍。不过这需要编译器提供帮助。把一些变量延迟写回内存。

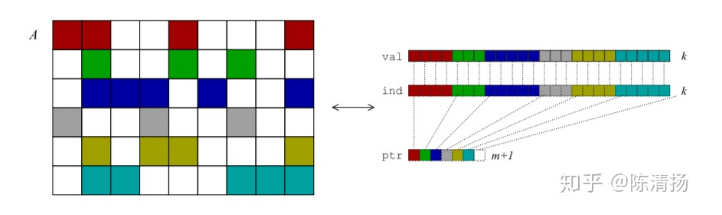
### 总结

通过循环展开，让编译器能够发现数据的重用，从而把一些数据延迟写回内存，大幅度减少访存次数，从而减少运行时间。

# 进阶：大规模矩阵计算优化

大规模稀疏矩阵的优化

优化方法是改变矩阵的表示方法，稀疏矩阵大部分是0，把这些0放在矩阵用占用空间事小，最重要是他们很多计算都没有意义，白白耗费时间。可以用CSR来存储这个稀疏矩阵。CSR的存储方式如下：



val存储的是非0数据。ind存在的是对应val位置的非0数据在原矩阵中的列编号。ptr存储的是原矩阵对应位置的行的第一个非0元素在val数组的位置。原矩阵有m行，ptr数据就有m+1行。

下面来看用二维数组和用CSR方式存储稀疏矩阵计算y<-y+Ax的区别(y是m个数据的一维数组，逻辑上是m维列向量，A是m行n列的二维数组，逻辑上是mxn的矩阵。x是n个数据的一维数组，逻辑上是n维列向量。

**用二维数据方式计算：**

for i=0 to m-1 do

    for j=0 to n-1 do

        y[i]=y[i]+A[i][j]\*x[j]

优点：直接访问，顺序访问，对cache和处理器流水线很友好。而且能够轻松地做并行计算。（只有y[i]需要写，而且写y[i]不依赖于任何在过程中会修改的值。）

缺点：时间复杂度为O(m\*n),很多的计算都没有意义。

**用CSR方式计算**

foreach row i do

    for l=ptr[i] to ptr[i+1]-1 do

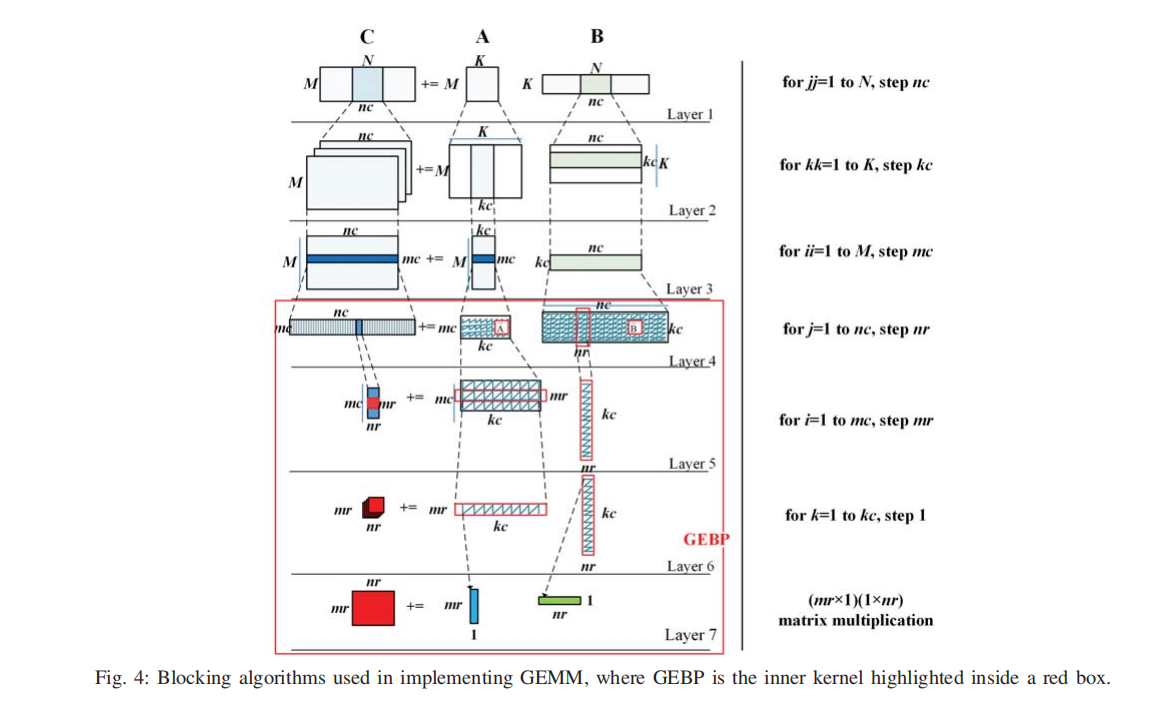
        y[i]=y[i]+val[l]\*x[ind[l]]

优点：时间复杂度为O(l) l为非0数据的个数。而且能够轻松地做并行计算。

缺点：对x的访问为间接访问，对处理器流水线不够友好。

# 进阶2：通过block对矩阵进行计算优化

优化思路：通过分块，增加cache的命中率

矩阵乘法要多次访问同一个元素。通用矩阵乘法对于同一个元素连续两次访问的时间差比较大。比如计算c11和c12都要访问A11,第二次访问a11时，已经是遍历完A的第一行之后的事情了。

计算机的存储结构中有cache，会保存最近使用的数据，如果在短时间内连续访问两次同一个数据，由于第一次访问时把数据放入了cache并且由于时间短还没被移出cache，第二次访问时从cache取数据而不是从内存取数据，所以会比较快，称为cache命中。把矩阵分块，可以减小连续访问同一个元素的时间差，分块后，第二次访问A11是在遍历完分块大小之后，而是遍历完一行之后。分块就是通过优化时间局部性来达到效果的。

但是分块是需要额外开销的，分块越小，分块就越多，额外开销就越多，但对cache越友好，分块如果过大，就达不到减少两次访问时间差的目的。在不同处理器上，要达到最优的效果，需要不断地尝试不同的分块大小。