CS205 Project5: General Matrix Multiplication

1.思路分析

本次project要求用C/C++实现GEMM，并且测试不同尺寸的矩阵下程序的效率与精度，尽可能的与OpenBLAS逼近。

我将从效率与精度两个方面与OpenBLAS进行比较，但重点将放在运行效率的优化上，效率的比较则以计算时间，效率的计算则调用benchmark完成。

由于CPU运算性能的限制，测试矩阵的大小为4x4，16x16，32x32，64x64.....1024x1024。

2.基础实现

首先，根据要求，我需要设计一个与`cblas_dgemm()`类似的函数，并且传入参数应该与其相同。

GEMM的定义为：$C \leftarrow \alpha AB + \beta C$

`cblas_dgemm()`的形式如下：

```
void cblas_dgemm(OPENBLAS_CONST enum CBLAS_ORDER Order, OPENBLAS_CONST enum
CBLAS_TRANSPOSE TransA, OPENBLAS_CONST enum CBLAS_TRANSPOSE TransB, OPENBLAS_CONST
blasint M, OPENBLAS_CONST blasint N, OPENBLAS_CONST blasint K,
         OPENBLAS_CONST double alpha, OPENBLAS_CONST double *A, OPENBLAS_CONST
blasint lda, OPENBLAS_CONST double *B, OPENBLAS_CONST blasint ldb, OPENBLAS_CONST
double beta, double *C, OPENBLAS_CONST blasint ldc);
```

其中，Order表示数据储存的顺序（行储存或列储存），TransA/B代表A/B是否转置，M,N,K则各自代表A,B的行列数，

alpha表示AB前的系数，beta表示C前的系数，ABC则为传入的指针，lda为A矩阵的leading dimension（列储存时为行的个数，行储存时为列的个数）。

根据这些参数，我设计了类似的函数`origin_GEMM()`。首先定义了两个enum，并设置了其对应的值。其次，我对传入的指针进行了检查，其余整数则设置为了size_t类型。然后用两个if语句判断储存类型，并根据是否转置选择合适的内存读取位置。

```
// 按行储存or按列储存
enum order{rowOrder=0, colOrder=1};
// 是否转置
enum transpose{noTrans=0, trans=1};
void origin_GEMM(enum order order, enum transpose TransA, enum transpose TransB,
size_t M, size_t N, size_t K, double alpha, double* A, size_t lda, double* B, size_t
ldb, double beta, double*C, size_t ldc)
{
    // 对传入指针进行检查
    if(C == NULL){
```

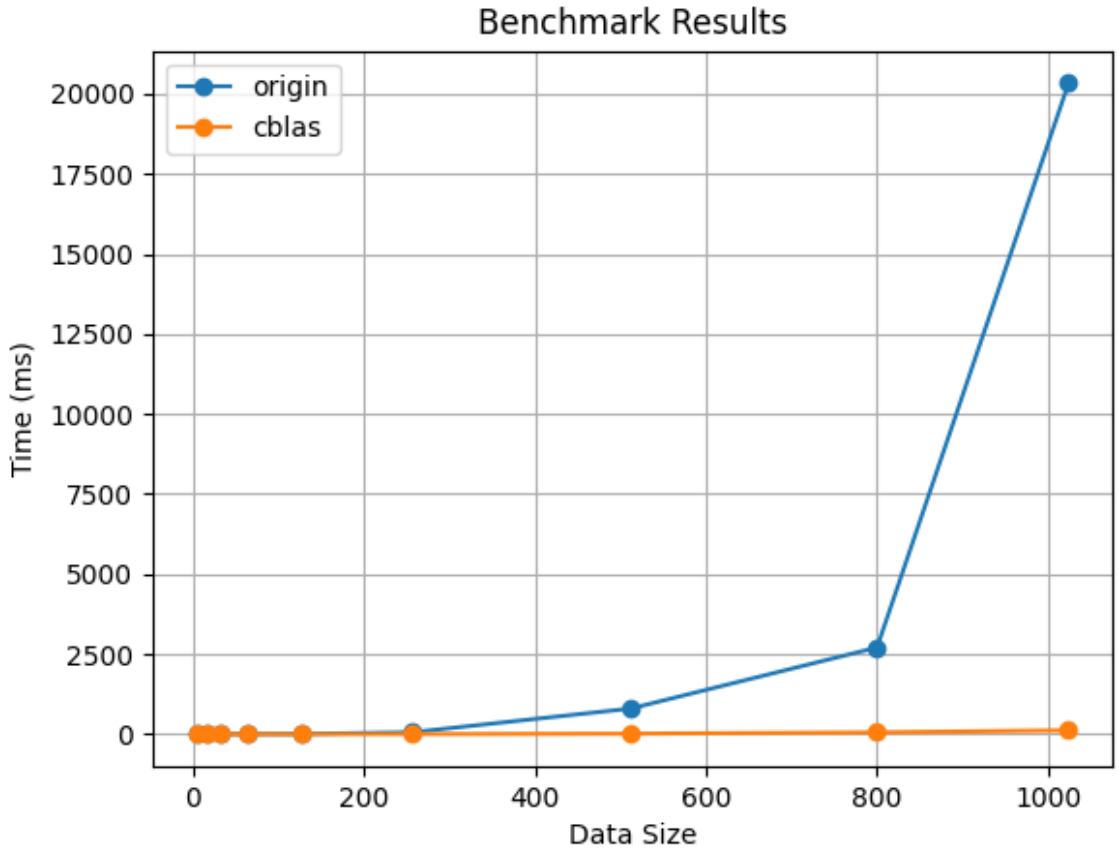
```
        printf("C是空指针!");
    }
    else if(A == NULL){
        printf("A是空指针!");
    }
    else if(B == NULL){
        printf("B是空指针!");
    }else{
    // 按行储存时
    if (order == 0)
    {
        for(int i = 0; i < M * N ; i++){
            C[i] = beta * C[i];
        }
        for (int i = 0; i < M; i++)
        {
            for (int j = 0; j < N; j++)
            {
                double sum = 0.0;
                for (int k = 0; k < K; k++)
                {
                    double a = (TransA == noTrans) ? A[i * lda + k] : A[k * lda +
i];
                    double b = (TransB == noTrans) ? B[k * ldb + j] : B[j * ldb +
k];
                    sum += a * b;
                }
                C[i * ldc + j] = alpha * sum + beta * C[i * ldc + j];
            }
        }

    }

    // 按列储存时
    if (order == 1)
    {
        for(int i = 0; i < M * N ; i++){
            C[i] = beta * C[i];
        }
        for (int i = 0; i < M; i++)
        {
            for (int j = 0; j < N; j++)
            {
                double sum = 0.0;
                for (int k = 0; k < K; k++)
                {
                    double a = (TransA == noTrans) ? A[k * lda + i] : A[i * lda +
k];
                    double b = (TransB == noTrans) ? B[j * ldb + k] : B[k * ldb +
j];
                    sum += a * b;
                }
```

```
                C[j * ldc + i] = alpha * sum + beta * C[j * ldc + i];
            }
        }
    }
}
```

随机矩阵的生成则利用随机数种子生成。

```
// 生成随机矩阵
void random_matrix(size_t row, size_t col, double *A)
{
  // 检查空指针
  if (A == NULL)
  {
    printf("A是空指针!");
  }
  else
  {
    double minValue = 0.0;  // 随机数范围的最小值
    double maxValue = 10.0; // 随机数范围的最大值

    struct timeval tv;
    gettimeofday(&tv, NULL);
    unsigned int seed = (unsigned int)tv.tv_usec;
    srand(seed);

    for (int i = 0; i < row * col; ++i)
    {
      double randomNum = minValue + ((double)rand() / RAND_MAX) * (maxValue -
minValue);
      A[i] = randomNum;
    }
  }
}
```

初步版本的效率图如下：

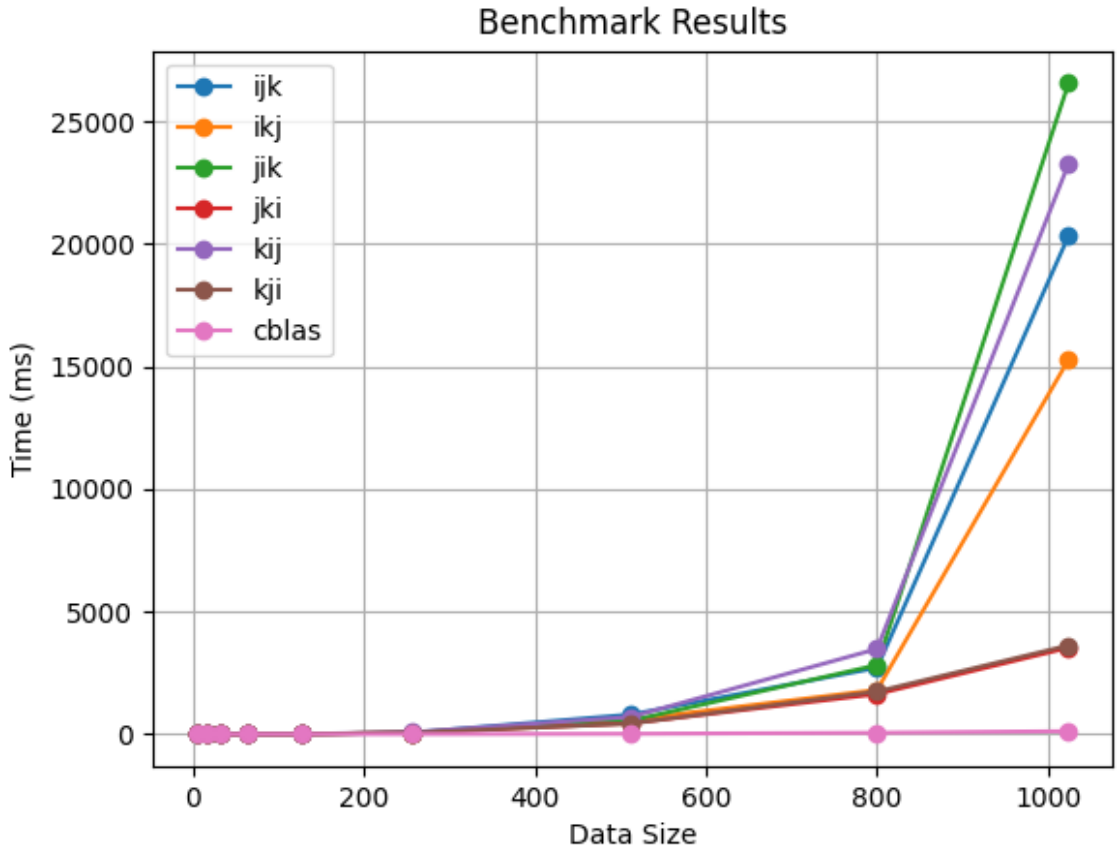


可以看到，cblas的运行效率远远超过我编写的程序。

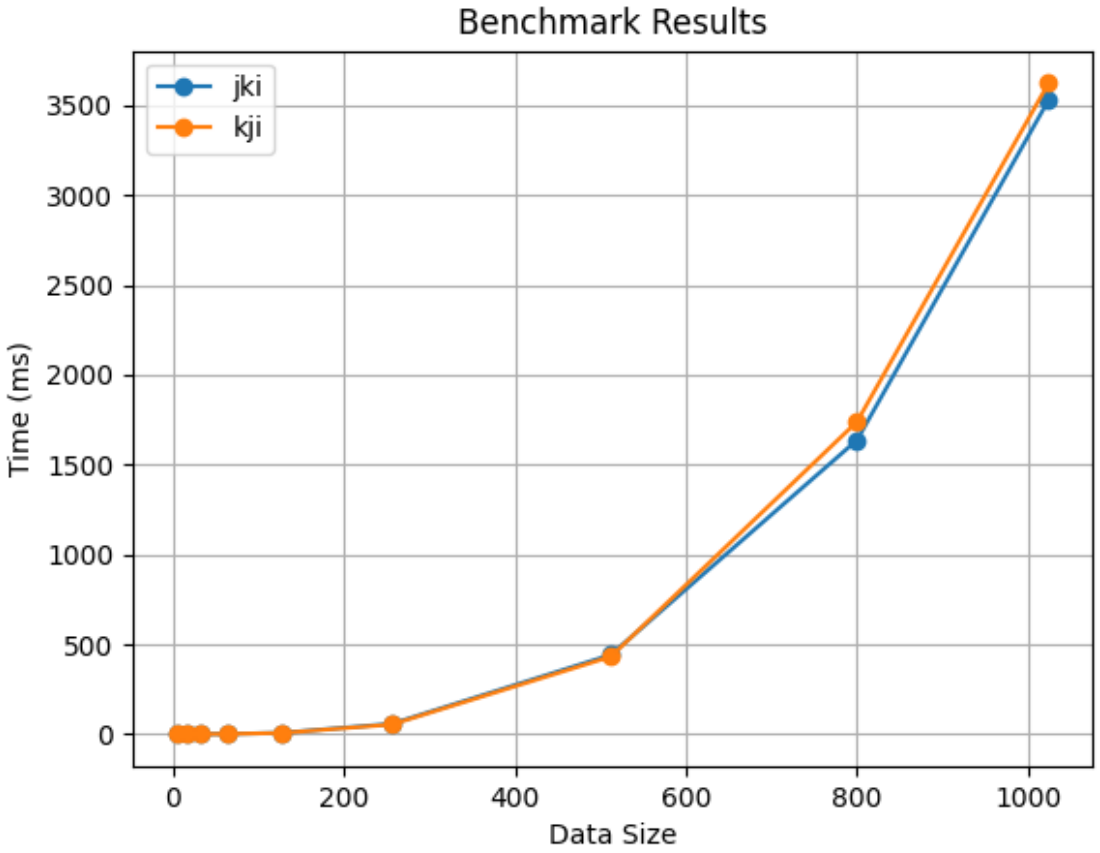
3. 效率优化

3.1 更改循环顺序

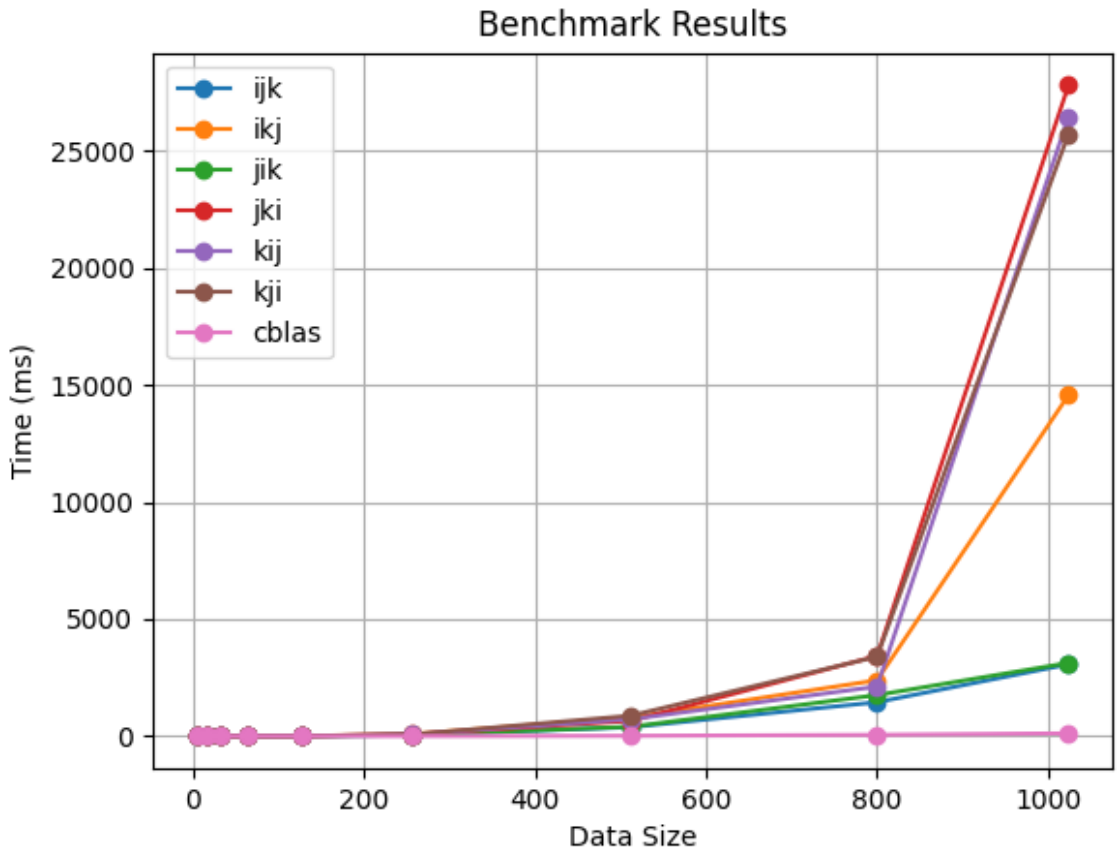
经过查询资料，我发现原有的顺序下，A的内存访问是连续的，但B的内存访问是不连续的，为了进一步探究，我将总共六种顺序的效率都画在了一张图上。



从图中可以看出，jki与kji的速度远远快于其他情况。单独画出这二者的图像，可以发现，jki的顺序还是更胜一筹。



但是在这个例子中，我使用的储存顺序都是列储存？那么如果我将输入的参数都改为行储存，结果会是怎样的，我对此又进行了一次研究。



更改顺序之后可以发现，当顺序为ijk和jik时反而更快。

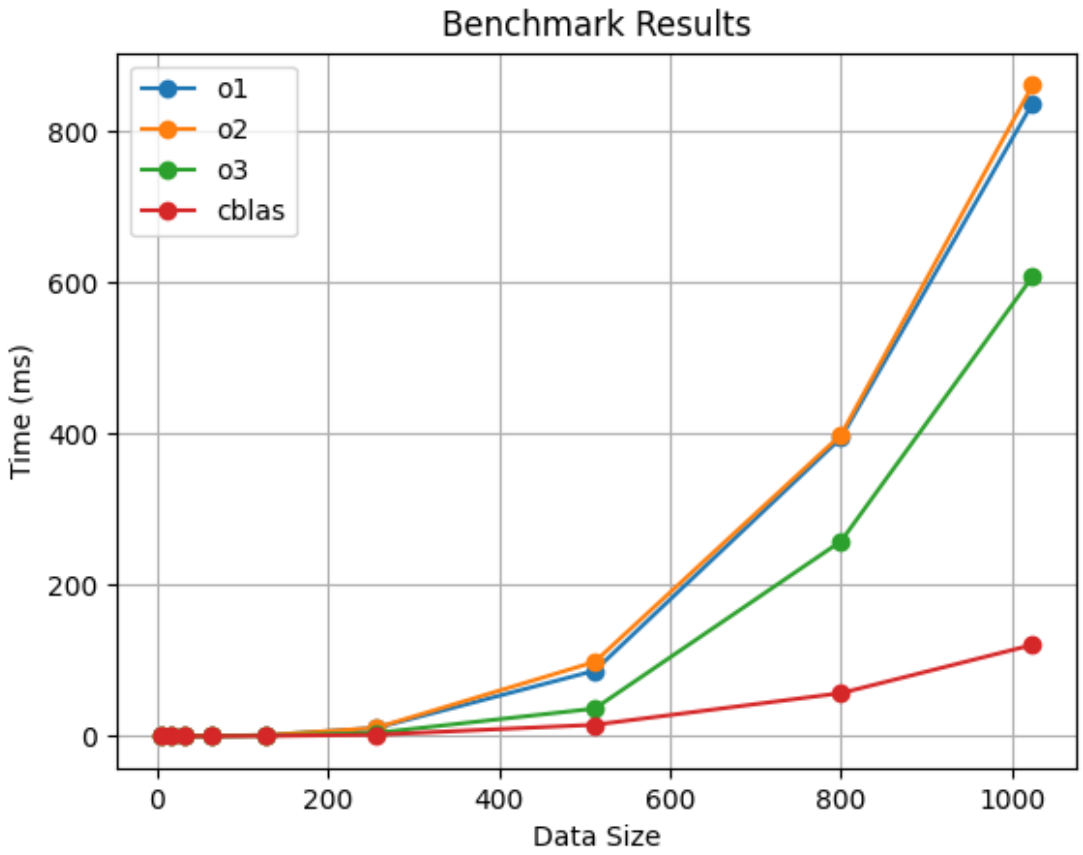
根据上述情况，我将代码修改成，当输入为行储存时，采用ijk的计算顺序，而当输入为列储存时，采用jki的计算顺序。代码如下：

```
// 按行储存时
    if (order == 0)
    {
        for(int t = 0; t < M * N ; t++){
            C[t] = beta * C[i];
        }

        for (int i = 0; i < M; i++)
        {
            for (int j = 0; j < N; j++)
            {
                double sum = 0.0;
                for (int k = 0; k < K; k++)
                {
                    double a = (TransA == noTrans) ? A[i * lda + k] : A[k * lda +
i];
                    double b = (TransB == noTrans) ? B[k * ldb + j] : B[j * ldb +
k];
                    sum += a * b;
                }
                C[i * ldc + j] = alpha * sum + beta * C[i * ldc + j];
```

```
            }
        }

    }

    // 按列储存时
    if (order == 1)
    {
        for(int t = 0; t < M * N ; t++){
            C[t] = beta * C[t];
        }

        for (int j = 0; j < N; j++)
        {
            for (int k = 0; k < K; k++)
            {
                double b = (TransB == noTrans) ? B[j * ldb + k] : B[k * ldb + j];
                for (int i = 0; i < M; i++)
                {
                    double a = (TransA == noTrans) ? A[k * lda + i] : A[i * lda +
k];
                    C[j * ldc + i] += alpha * a * b;
                }
            }
        }

    }
```

3.2 O1,O2,O3优化

众所周知，c中有O1,O2,O3三个优化级别，那么在开启这三种优化以后，哪种优化效果更好呢，同样可以绘制成图：



不出所料的，O3的优化效果最佳。正如project2曾查到过的资料所说：

1. **-O1**：
   编译器优化后，增加编译时间，并处理大函数时会占用更大的内存。
   优化结果，使得程序文件变小，执行时间变短。
   开启一些基本的优化，如去除无用代码、简化表达式等。

2. **-O2**：
   比O1优化更多。g++会尽可能的引入不造成空间-时间（不为了降低执行时间而增大使用内存，或降低使用内存而增加执行时间）影响的优化。和没有优化相比，这选项增加了编译时间，同时提高了代码执行性能。
   Level 2优化打开了所有Level 1打开的选项，并且在 -O1 的基础上增加一些较为复杂的优化，如函数内联、循环展开等。

3. **-O3**：
   在 -O2 的基础上增加更多的优化，如自动向量化、函数调用优化等。

O3在O2的基础上增加了更多的优化，利用向量化等操作同时处理多个数据元素，提升了矩阵的运算效率，因此效果更佳。可以看到，此时的时间差距已经不算太大了。

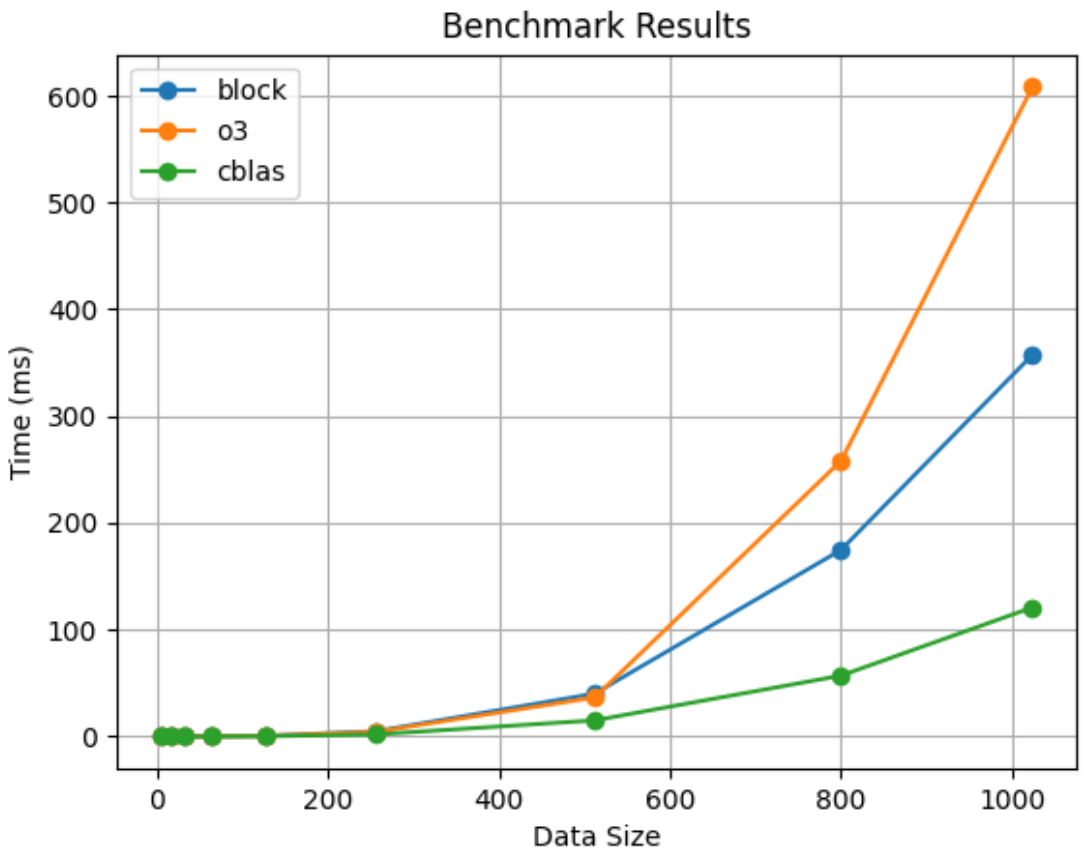
3.3 矩阵分块

对于矩阵本身来说，除了改变for循环的顺序外，还可以考虑通过分块的方式提高程序效率，由于输入数据均为4的倍数，我将矩阵分块的大小定为4，具体代码如下：

```
#define min(a, b) ((a) < (b) ? (a) : (b))
// 按行储存时
    if (order == 0)
    {
      for (int i = 0; i < M * N; i++)
      {
          C[i] = beta * C[i];
      }

      for (int ii = 0; ii < M; ii += 4)
      {
          for (int jj = 0; jj < N; jj += 4)
          {
              for (int kk = 0; kk < K; kk += 4)
              {
                  for (int i = ii; i < min(ii + 4, M); i++)
                  {
                      for (int j = jj; j < min(jj + 4, N); j++)
                      {
                          double sum = 0.0;
                          for (int k = kk; k < min(kk + 4, K); k++)
                          {
                              double a = (TransA == noTrans) ? A[i * lda + k] : A[k
* lda + i];
                              double b = (TransB == noTrans) ? B[k * ldb + j] : B[j
* ldb + k];
                              sum += a * b;
                          }
                          C[i * ldc + j] += alpha * sum;
                      }
                  }
              }
          }
      }
    }
    // 按列储存时
    if (order == 1)
    {
      for (int i = 0; i < M * N; i++)
      {
          C[i] = beta * C[i];
      }

      for (int jj = 0; jj < N; jj += 4)
      {
          for (int kk = 0; kk < K; kk += 4)
```

```
            {
                for (int j = jj; j < min(jj + 4, N); j++)
                {
                    for (int k = kk; k < min(kk + 4, K); k++)
                    {
                        double b = (TransB == noTrans) ? B[j * ldb + k] : B[k * ldb +
j];

                        for (int i = 0; i < M; i++)
                        {
                            double a = (TransA == noTrans) ? A[k * lda + i] : A[i *
lda + k];

                            C[j * ldc + i] += alpha * a * b;
                        }
                    }
                }
            }
        }
    }
```

在分块以后，绘制程序图（同样开了O3）：

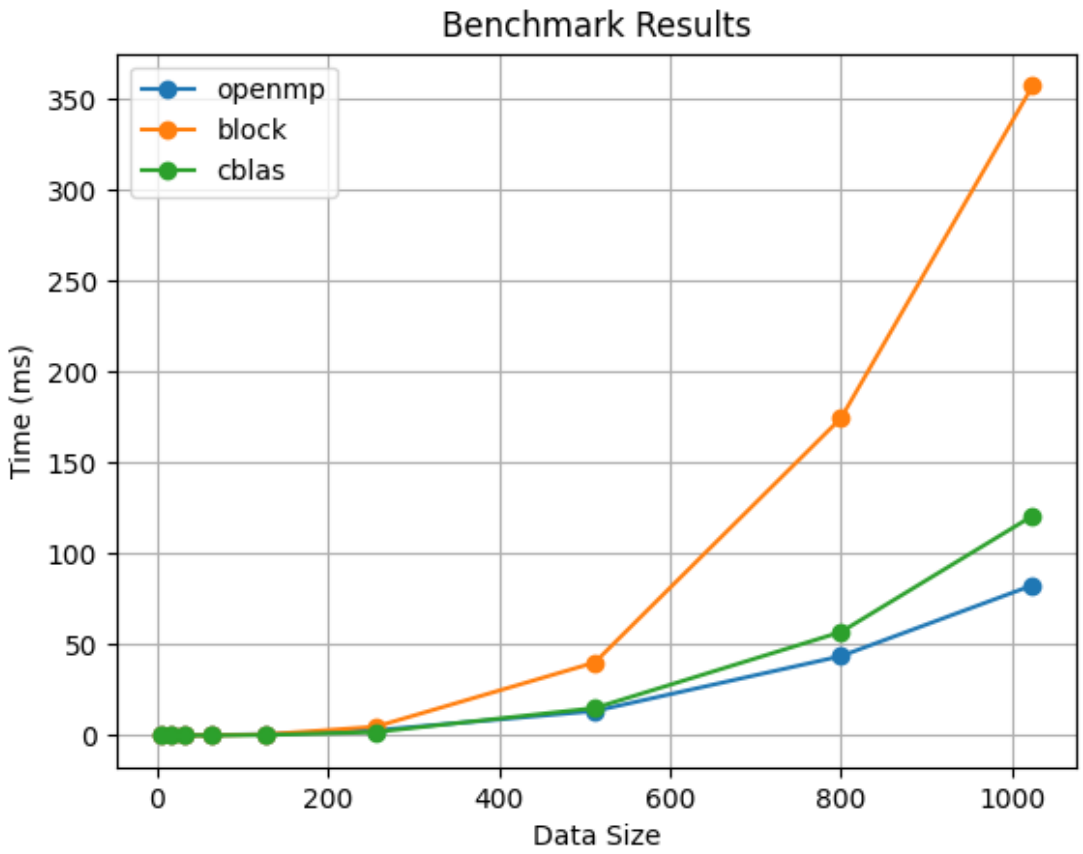


可以发现，在分块之后，在矩阵大小为1000x1000时，效果已经十分明显，运行时间仅是cblas的三倍了。

3.4 OpenMP优化

上面的优化，都只是使用了计算机的一个核心或一个进程，那么如果使用OpenMP指令进行多线程并行计算会怎样呢?

我在每个for循环前，添加了`#pragma omp parallel for schedule(dynamic)`的指令，并且在编译时添加了`-fopenmp`，得到的结果如图:



可以看出，加入了openmp的指令后，程序运行时间已经比cblas还要短了，尽管这并不能说明效率就已经高于cblas，并且没有进行GFLPOS的比较，但已经是可喜的进步了。

4. 精度

效率的优化已经基本完成，接下来需要确认精度上的差异。为了确认精度上的差异，我写了一个for循环遍历两种方法计算出来的结果的绝对值之差，并将其全部相加，若得出来的误差很小，则可视作精度大致相同。

代码如下:

```
#include <iostream>
#include <cmath>
#include <cstring>
#include <cblas.h>
#include "multiplication.h"
using namespace std;
```

```
int main() {
    double* C1 = new double[256*256];
    double* C2 = new double[256*256];
    double* random1 = new double[256*256];
    random_matrix(256, 256, random1);
    double* random2 = new double[256*256];
    random_matrix(256, 256, random2);

    origin_GEMM(colOrder, noTrans, trans, 256, 256, 256, 1, random1, 256, random2,
256, 1, C1, 256);

    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasTrans, 256, 256, 256, 1, random1,
256, random2, 256, 1, C2, 256);

    double error = 0.0;
    for(int i = 0; i < 16; i++){
        error += fabs(C1[i] - C2[i]);
    }

    cout << error << endl;

    delete[] C1;
    delete[] C2;
    delete[] random1;
    delete[] random2;
}
```

当矩阵大小为256x256时，绝对误差为如下：

当矩阵大小为512x512时，绝对误差如下：

当矩阵大小为1024x1024时，绝对误差如下：

因此看近似看作没有误差。

5. 总结

这次是最后一次project，在解脱的同时也学到了很多神奇的东西。比如benchmark的使用，以及结果的导出与画图，在寻求多方帮助以后，才总算弄好这些看似不重要但是又很有用的事情。

回到正题，在矩阵的效率优化上，我采用了循环顺序变换、矩阵分块、O3优化、OpenMP优化的方法，最终在某一数量级的运算时间达到了大致相当的程度。但仍然有很多力有未贷的地方，比如SIMD的指令优化，尝试失败，因时间所限，终究是没有再进一步。又比如O3优化究竟优化了哪些步骤，由于我不是计算机系的学生，对于这方面就没有再做更深层的研究。

不过总而言之，还是一次收获满满的project，也庆祝自己cpp的旅途终于结束了吧！（哭）