Project5

12112504 郭城

Dgemm 优化

本程序主要亮点为：并行，大分块以及小分块的使用

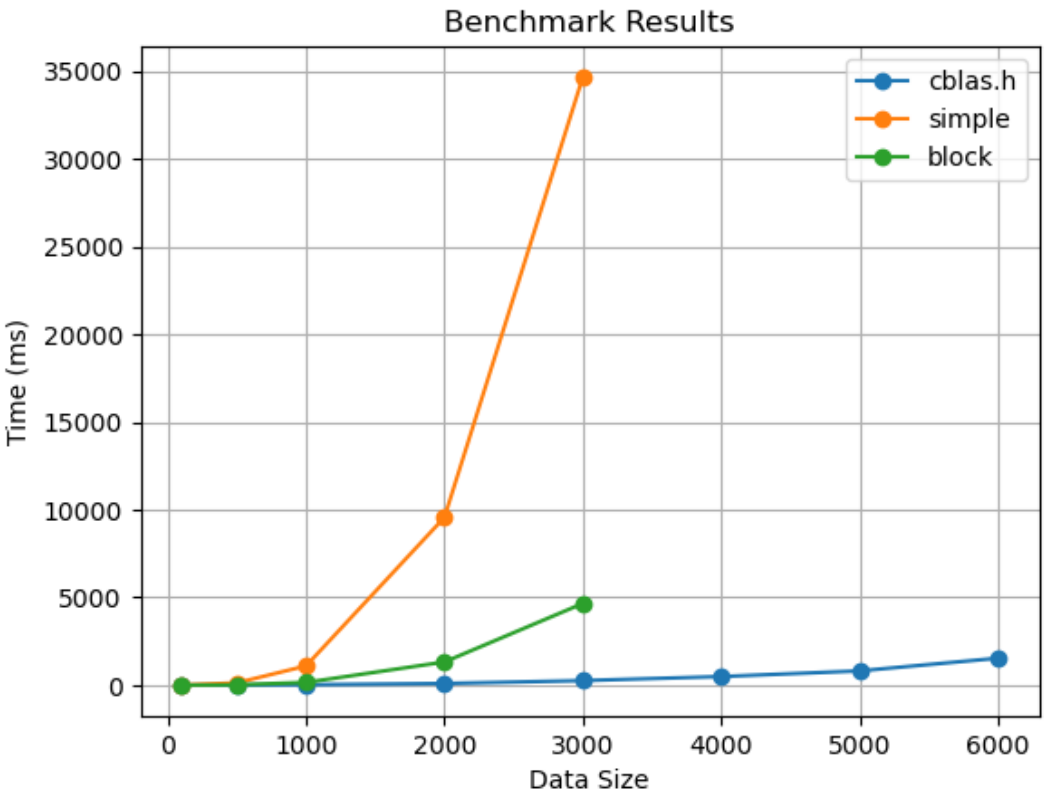
程序思路 :将 ABC 矩阵划分为大分块 **BLOCK_SIZE*BLOCK_SIZE(512)**, 以及小分块 **4*4**, 在 for 循环最外层加入并行运算 omp parallel。

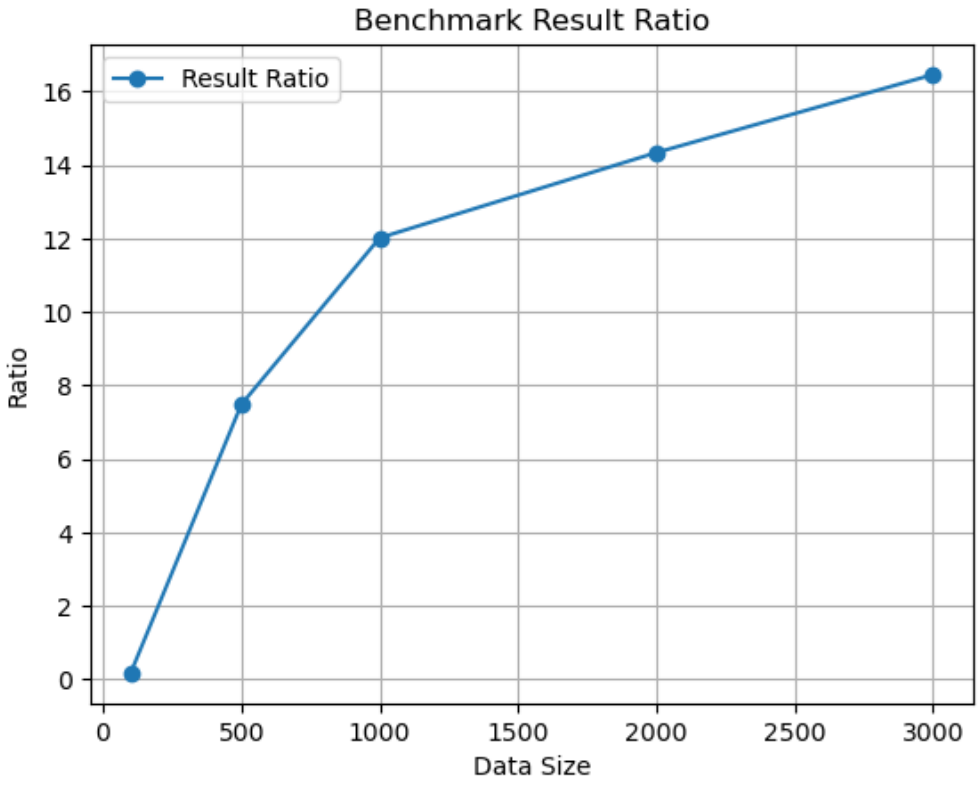
小分块：每次对 C 中的 4*4 大小的矩阵进行值更新，每次从 A 中拿出对应该 C4*4 矩阵的四个 a1, a2, a3, a4，并分别与 4 个对应的 b 值求积
例如 C（0，0）与 A（0，0），A（1，0），A（2，0），A（3，0），A（k，0）等等有关，又与 B（0，0），B（0，1），B（0，2），B（0，3），B（0，k）等等有关，
C（0，0）= A（k,0）*B(0,k)
而 A（0，0）同时又与 C（1，0），C（2，0），C（3，0），C（k，0）有关，则可以在一次取出 A（a，b）的值后对他进行 4 次使用，B 同理，以此大大降低 ABC 矩阵内存访问次数每个数据的读取至少降低为 1/4。以此提高效率
并且对 C 赋值前先使用**寄存器**将数值存储，因为每次循环都会对 C 值进行 4 次增加，使用寄存器求和之后在赋值给 C，也将大大减少内存读写。

代码实现：大量使用 for 循环是为后续 openmp 并行作铺垫，BLOCK_SIZE 为后续实现需要

```
for (int ii = i; ii < i + BLOCK_SIZE && ii < M; ii += 4)
{
    for (int jj = j; jj < j + BLOCK_SIZE && jj < N; jj += 4)
    {

        // Registers
        double sum[4][4] = {{0.0}};
        for (int kk = k; kk < k + BLOCK_SIZE && kk < K; ++kk)
        {
            // Load values from memory into registers
            double a0 = A[(A_TRANSPOSED ? kk : ii) * lda + (A_TRANSPOSED ? ii : kk)];
            double a1 = A[(A_TRANSPOSED ? kk : ii + 1) * lda + (A_TRANSPOSED ? ii + 1 : kk)];
            double a2 = A[(A_TRANSPOSED ? kk : ii + 2) * lda + (A_TRANSPOSED ? ii + 2 : kk)];
            double a3 = A[(A_TRANSPOSED ? kk : ii + 3) * lda + (A_TRANSPOSED ? ii + 3 : kk)];

            for (int x = 0; x < 4; ++x)
            {
                double b0 = B[(B_TRANSPOSED ? jj + x : kk) * ldb + (B_TRANSPOSED ? kk : jj + x)];

                sum[x][0] += a0 * b0;
                sum[x][1] += a1 * b0;
                sum[x][2] += a2 * b0;
                sum[x][3] += a3 * b0;
            }
        }

        // Update C directly from the registers
        for (int k = 0; k < 4; ++k) {
            for (int l = 0; l < 4; ++l) {
                C[(ii + l) * ldc + (jj + k)] = beta * C[(ii + l) * ldc + (jj + k)] + alpha * sum[k][l];
            }
        }
    }
```

效率随数据规模变化图





效率提升：（与简单递归，openblas 相比）数据大小在 3000 以内

在数据大小为 3000 时效率比纯递归提升 7.5 倍，为 openblas 十六分之一，随数据量增大，与 openblas 效率比将降低

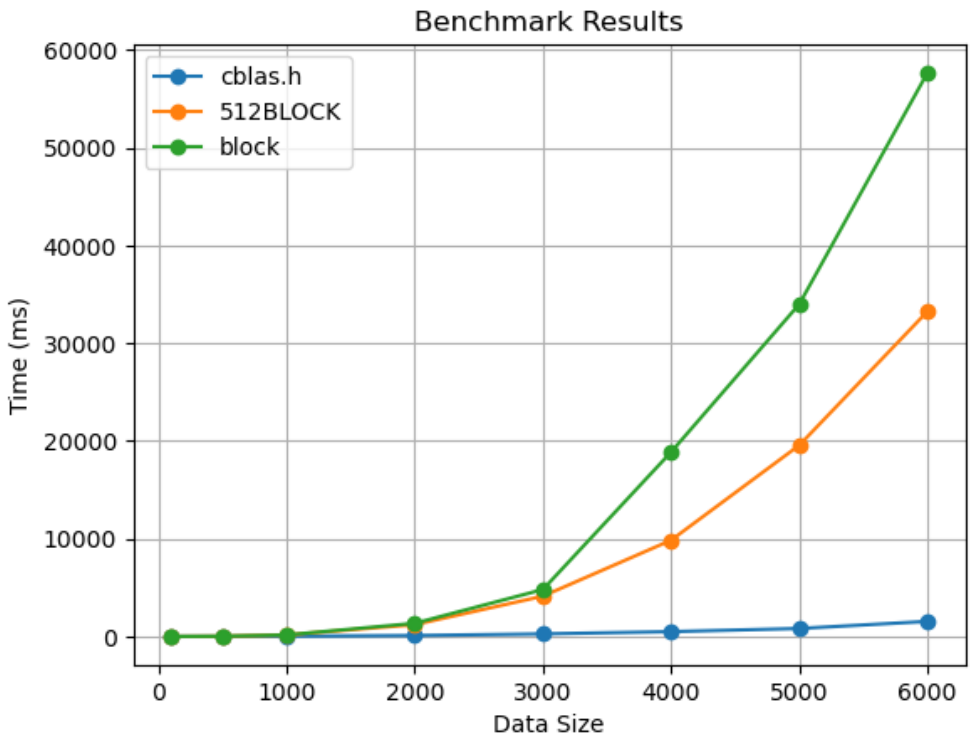
大分块：在小分块基础上，对整个 ABC 矩阵进行大分块，BLOCK_SIZE 为 32-512 最佳，大分块将 ABC 三个矩阵分为以 BLOCK_SIZE 为边长的“大方块”，以此达到每次移动之后，比如 B 矩阵的分块移动之后，A 矩阵对应的分块不需要移动就可以进行下一轮运算，A 矩阵的对应区域内存读取减少。

代码实现：分别对 AB 的长宽循环，每次步长为 BLOCK_SIZE，在一个 BLOCK 迭代结束后再进行下一步，内部为小分块。

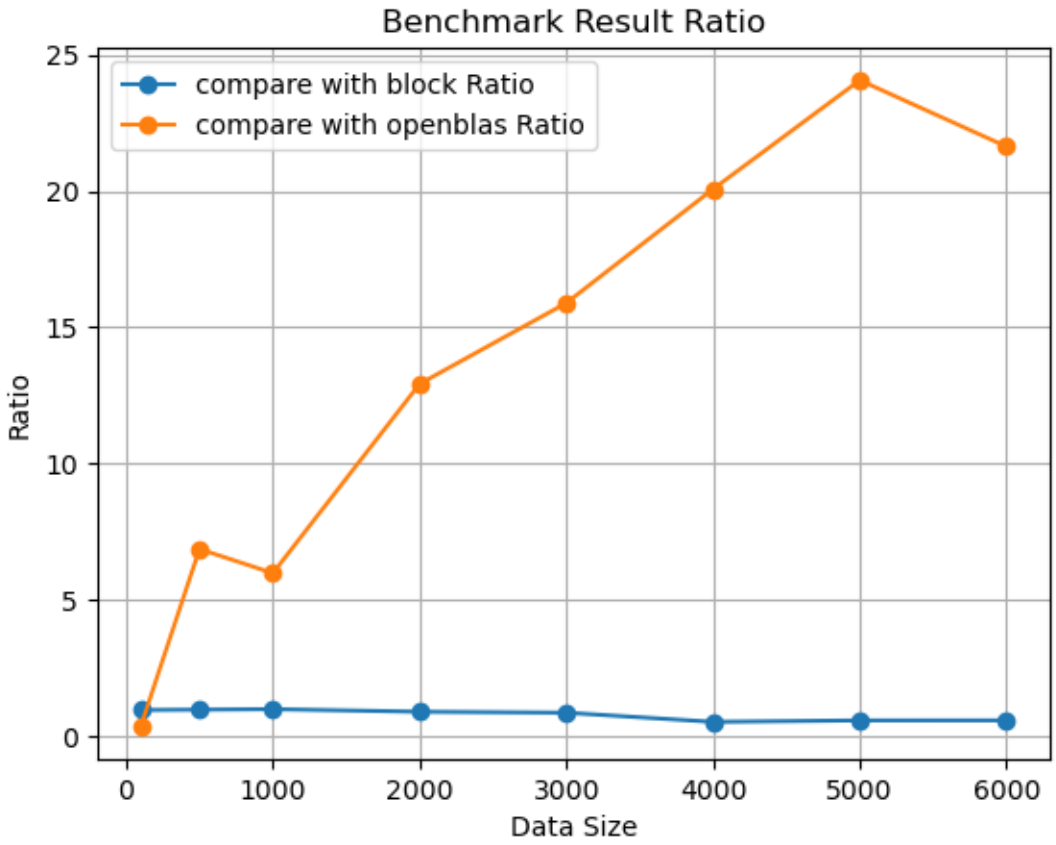
```
const int BLOCK_SIZE = 128; // Set the block size

for (int i = 0; i < M; i += BLOCK_SIZE)
{
    for (int j = 0; j < N; j += BLOCK_SIZE)
    {
        for (int k = 0; k < K; k += BLOCK_SIZE)
        {
            for (int ii = i; ii < i + BLOCK_SIZE && ii < M; ii += 4)
            {
                for (int jj = j; jj < j + BLOCK_SIZE && jj < N; jj += 4)
                {
```

效率提升：在数据规模 6000 时，达到前者的 1.6 倍，达到 openblas 效率 4.6%

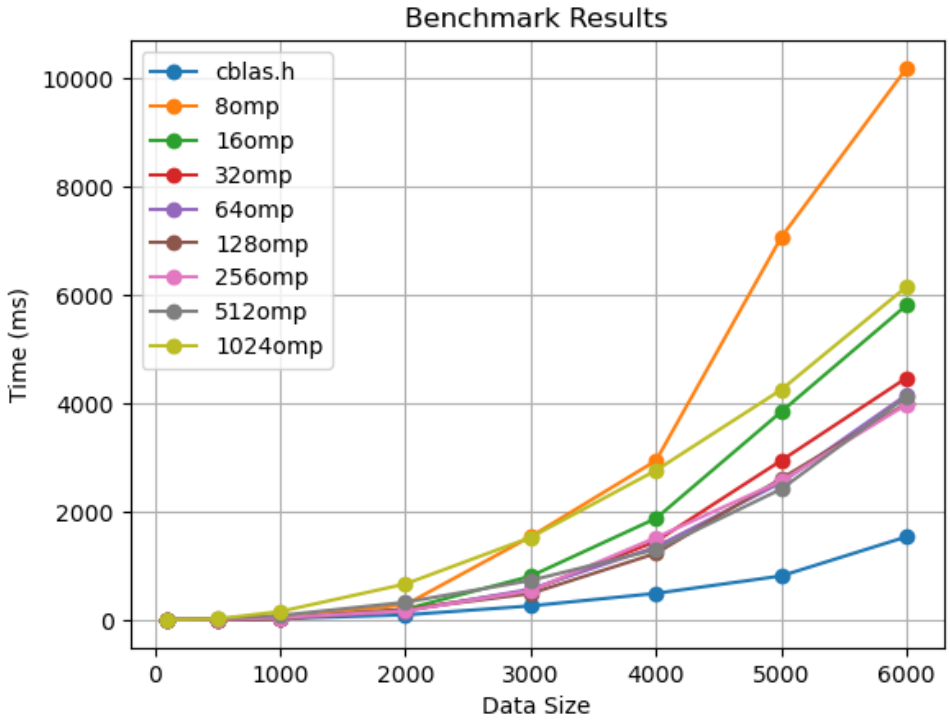
效率随数据规模变化图



比值



不同分块大小效率对比（8-1024）该结果是使用并行优化后结果：

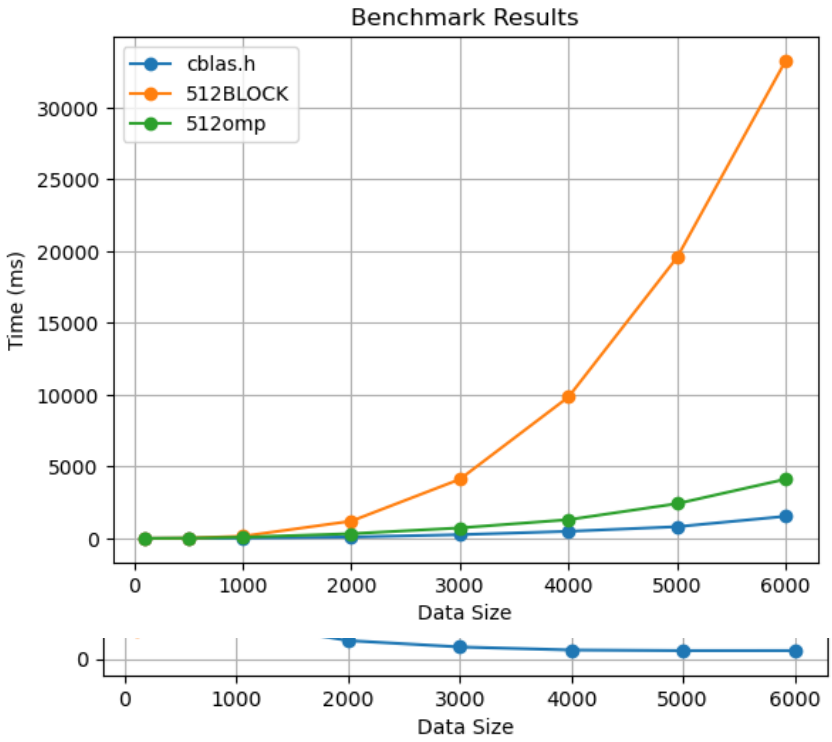


结论：分块大小在 32 到 512 之间最优，范围内在 6000 数据大小内差距不大。优化效果与小分块类似，是由减少内存访问得到的。

Openmp：在每次循环前加入*#pragma omp parallel for*，以 omp.h 库的并行优化代码

效率提升：在数据规模 6000 时，达到前者的 8 倍，达到 openblas 效率三分之一,效率提升显著。

效率随数据规模变化图



比值

结论：使用并行后，代码运行速度得到大幅提高，由于并行代表的是利用处理器多核的特性充分利用多条通道同时运算，得到成倍提升的效率。

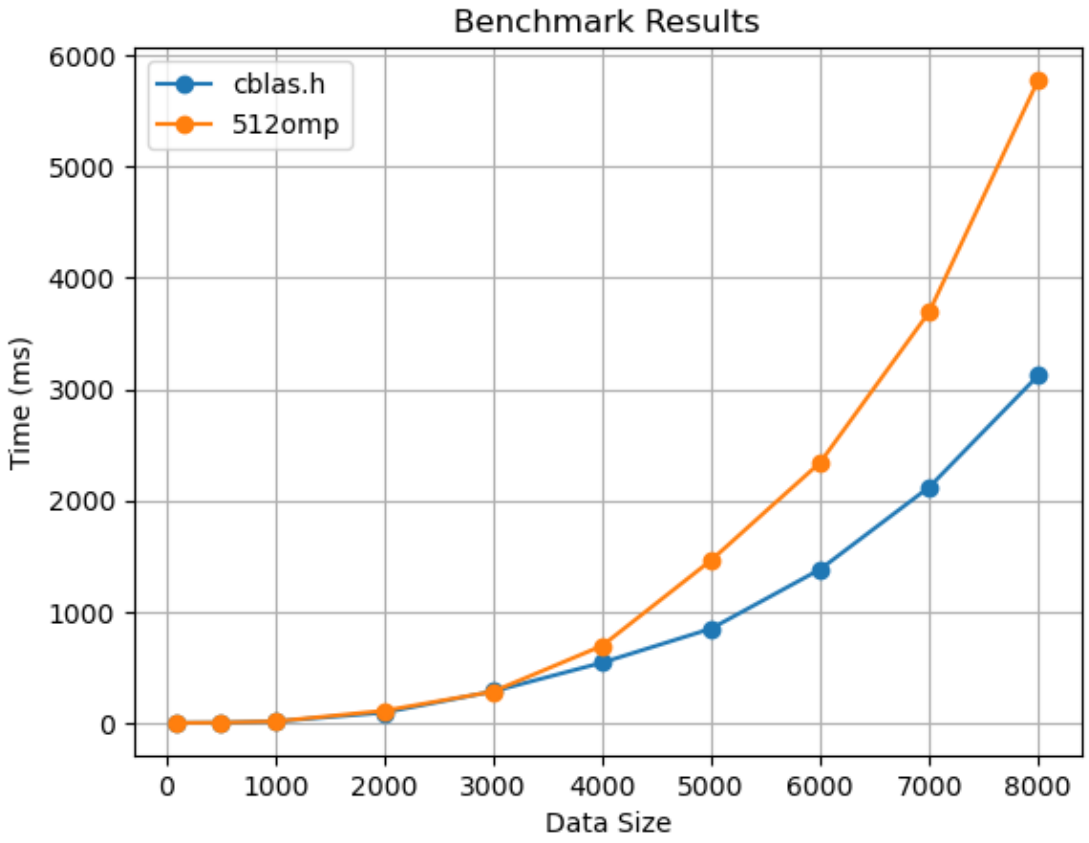
代码细节：

列优先功能是由行优先演化而来，在对 AB 矩阵转制之后进行运算，最后再将得到的 C 矩阵转制以得到最终结果。

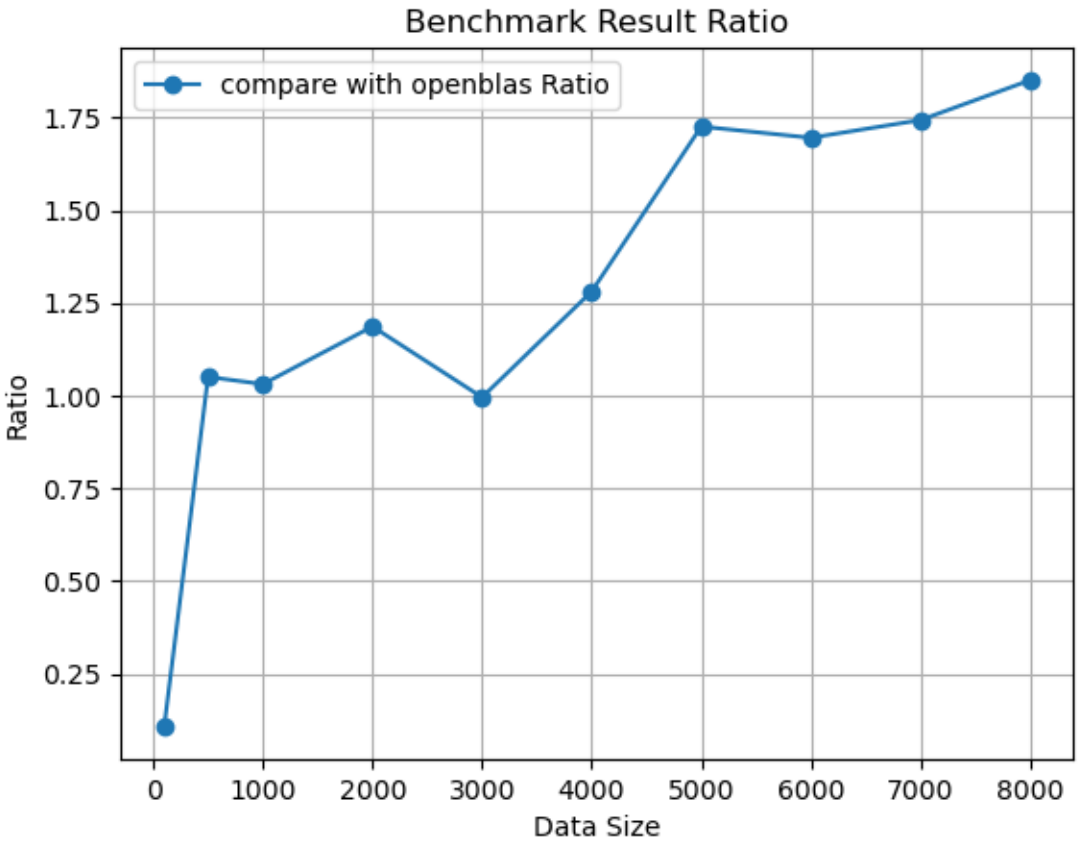
AB 矩阵是否转制，若 A 或 B 则在代码初始阶段将 A，B 转化为其转制矩阵，再进行后续计算

最终优化结果：将循环内部 if 语句放到循环外（A_Transposed？），利用转制解决，使用最佳大分块大小 128 后。

效率随数据规模变化图



比值



最终得到代码效率达到 openblas 的 55.6%（数据规模 8000），在单人两周尝试学习下，已经得到不错成果。

代码注意事项：benchmark2.cpp，为多数据规模下测试结果效率，test.cpp 为测试结果准确性。

代码核心为 **cblas_dgemmOptimized_update5** 部份。