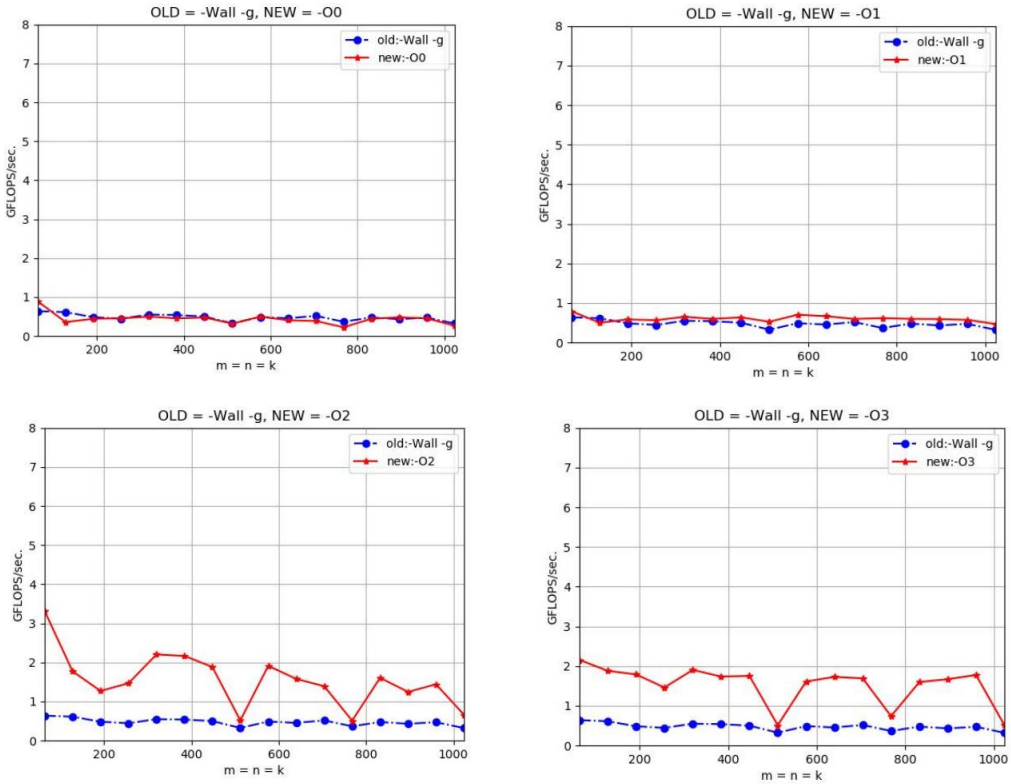
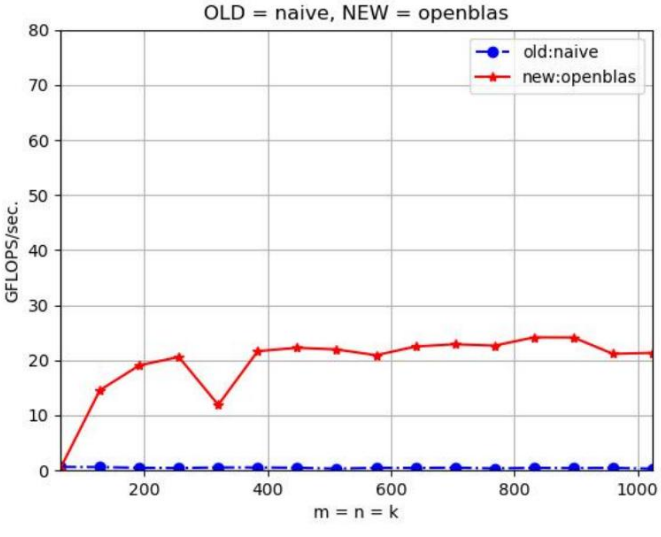
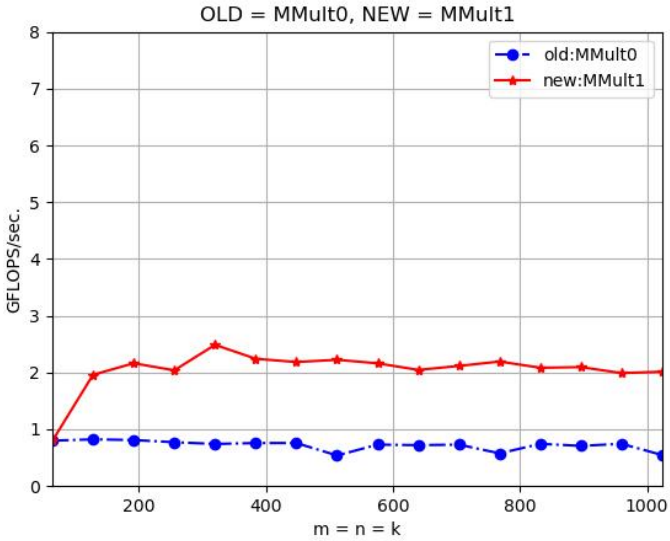


naive dgemm 与 O0、O1、02、03 四种编译优化

编译优化（Compiler Optimization）指通过编译器对源代码进行优化，以提高程序的执行效率和性能，减少程序的运行时间和资源消耗，同时保持程序的正确性和功能完整性。



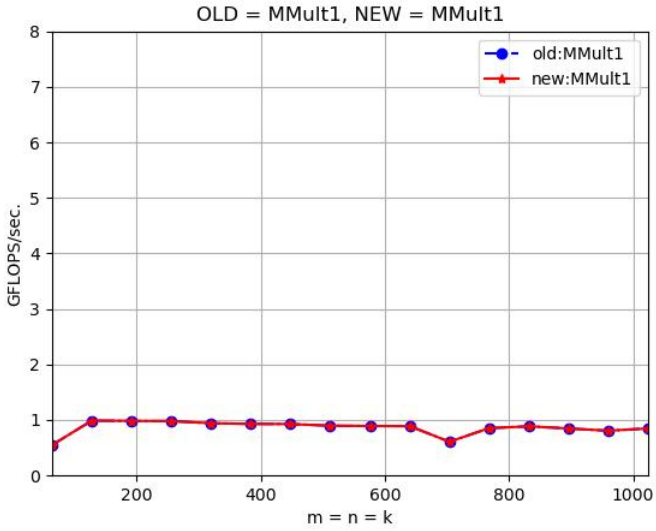
naive dgemm 与 openblas



单线程与多线程

OpenBLAS（Open Basic Linear Algebra Subprograms）是一个开源的优化的基本线性代数子程序库，用于高效执行线性代数运算。

多线程优化（Multithreading Optimization）是指通过使用多线程来并行执行程序的不同部分，以提高程序的并发性和性能。多线程优化可以在多核处理器上实现并行计算，从而加快程序的运行速度。



Openmp（两条线均为 omp）

OpenMP（Open Multi-Processing）是一种并行编程模型，它为共享内存系统提供了一套简单易用的接口，使得程序员能够方便地将串行程序并行化。OpenMP 使用指令的方式来标识并行区域，并通过自动创建和管理线程来实现并行计算。OpenMP 优化可以通过使用不同的 OpenMP 指令和选项来实现。此次实验中，使用了#pragma omp parallel for

```
42
43  void dgemm_block_multithread(int n, double *A, double *B, double *C, int block_size, int num_threads) {
44      pthread_t threads[num_threads];
45      ThreadData data[num_threads];
46
47      int rows_per_thread = n / num_threads;
48
49      for (int t = 0; t < num_threads; t++) {
50          data[t].n = n;
51          data[t].A = A;
52          data[t].B = B;
53          data[t].C = C;
54          data[t].block_size = block_size;
55          data[t].start_row = t * rows_per_thread;
56          data[t].end_row = (t + 1) * rows_per_thread;
57
58          if (t == num_threads - 1) {
59              // 最后一个线程处理剩余的行
60              data[t].end_row = n;
61          }
62
63          pthread_create(&threads[t], NULL, dgemm_block_thread, (void *)&data[t]);
64      }
65
66      for (int t = 0; t < num_threads; t++) {
67          pthread_join(threads[t], NULL);
68      }
69  }
```

```
23
24  void *dgemm_block_thread(void *arg) {
25      ThreadData *data = (ThreadData *)arg;
26      int n = data->n;
27      double *A = data->A;
28      double *B = data->B;
29      double *C = data->C;
30      int block_size = data->block_size;
31      int start_row = data->start_row;
32      int end_row = data->end_row;
33      #pragma omp parallel for num_threads(10)
34      for (int i = start_row; i < end_row; i += block_size) {
35          for (int j = 0; j < n; j += block_size) {
36              for (int k = 0; k < n; k += block_size) {
37                  // 对每个block进行矩阵乘法
38                  for (int ii = i; ii < fmin(i + block_size,n); ii++) {
39                      for (int jj = j; jj < fmin(j + block_size,n); jj++) {
40                          for (int kk = k; kk < fmin(k + block_size,n); kk++) {
41                              C[ii * n + jj] += A[ii * n + kk] * B[kk * n + jj];
42                          }
43                      }
44                  }
45              }
46          }
47      }
48
49      pthread_exit(NULL);
50  }
51
```