实验报告

课程：高性能计算应用实践

姓名：王峻阳

学号：220110317

学院：计算机科学与技术学院

学期：2023 年秋季学期

实验日期：2023 年 10 月 7 日

一、实验环境

操作系统　Ubuntu 22.04.2 LTS（在 WSL2 上运行）

内存　7808.3 MiB（分配给 WSL2 的部分）

CPU　名称：12th Gen Intel(R) Core(TM) i5-12500H
指令集架构：x86-64
字长：64 位
物理核数：12 核，其中 4 核是性能核，8 核是能效核
线程数：16 线程，其中 8 线程来自性能核，8 线程来自能效核
基准频率：2.50 GHz（性能核）、1.8 GHz（能效核）
最大睿频频率：4.50 GHz（性能核）、3.30 GHz（能效核）

CPU 缓存　L1i　32 KiB (8-way, 64-byte line)
L1d　48 KiB (12-way, 64-byte line)
L2　1.25 MiB (10-way, 64-byte line)
L3　18 MiB (12-way, 64-byte line)
注：性能核和能效核的 L1、L2 级缓存据说不一样

二、朴素矩阵乘实现

一、实现方案

朴素算法就是三层循环，不过按循环变量的次序可分为六种，实验中实现了其中两种，`ijp`（MMult0.c）和 `ijp`（MMult0.c）。

二、关键代码

ijp（MMult0.c）

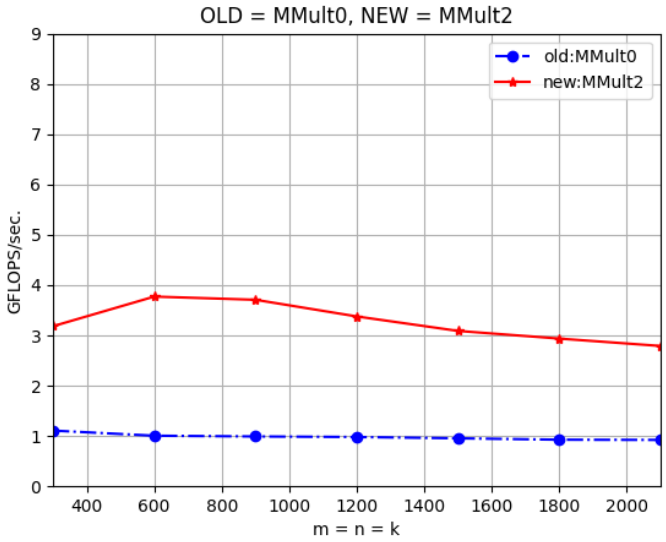
```
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb, double *c, int ldc) {
    int i, j, p;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            for (p = 0; p < k; p++) {
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
            }
        }
    }
}
```

jpi（MMult2.c）

```
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb, double *c, int ldc) {
    int i, j, p;
    for (j = 0; j < n; j++) {
        for (p = 0; p < k; p++) {
            for (i = 0; i < m; i++) {
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
            }
        }
    }
}
```

三、性能数据

ijp 版本的缓存 miss 数约为 9/8 mnk，jpi 版本的缓存 miss 数约为 1/4 mnk，因而后者比前者快几倍。（注：为方便起见，本报告中所有编译均打开 gcc 的 O1 优化，这会优化数组访问，并将合适的变量保持在寄存器中。）



三、OpenBLAS 实现

一、实现方案

只需调用 OpenBLAS 库的 cblas_dgemm 函数。实验三要求的不同优化之间的对比不重复在此列出，请参考实验三实验报告。

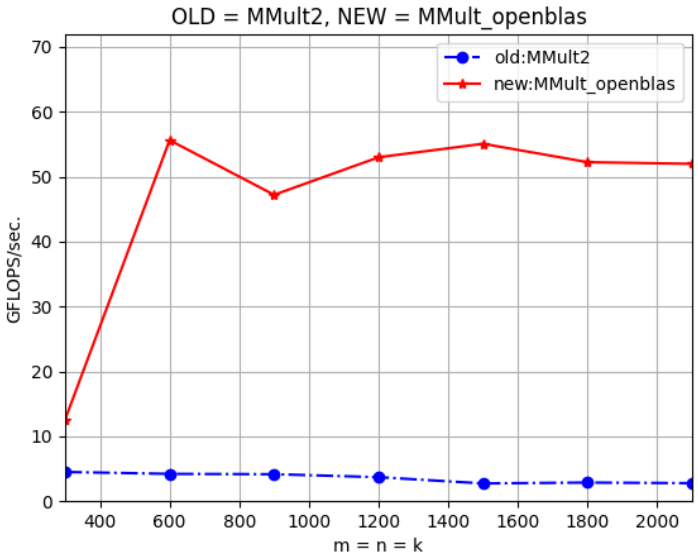
二、关键代码

OpenBLAS（MMult_openblas.c）

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc ) {
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, 1.0, a, lda, b, ldb, 1.0, c, ldc);
    return;
}
```

三、性能数据

OpenBLAS 的实现，Gflops 能达到朴素方法的十几倍。



四、Pthreads 多线程实现

一、实现方案

矩阵分块方法不在此重复列出，可参见实验五实验报告和MMult3.c。Pthreads多线程实现是，由主线程把矩阵分为几个部分，每个子线程用分块方法计算一个部分，然后主线程回收子线程。

二、关键代码

Pthreads（MMult4.c）

分配任务

```
void MY_MMult(int m, int n, int k, double *a, int lda,
                                   double *b, int ldb,
                                   double *c, int ldc) {
    pthread_t tid[NUM_THREADS];
    struct Arg arg[NUM_THREADS];
    for (int t = 0; t < NUM_THREADS; t++) {
        int n_start = t * n / NUM_THREADS;
        int n_end = (t + 1) * n / NUM_THREADS;
        arg[t].m = m; arg[t].n = n_end - n_start; arg[t].k = k;
        arg[t].a = a; arg[t].b = &B(0, n_start); arg[t].c = &C(0, n_start);
        arg[t].lda = lda; arg[t].ldb = ldb; arg[t].ldc = ldc;
        Pthread_create(&tid[t], NULL, dgemm_thread, (void *) &arg[t]);
    }
    for (int t = 0; t < NUM_THREADS; t++) {
        Pthread_join(tid[t], NULL);
    }
}
```

子线程入口函数

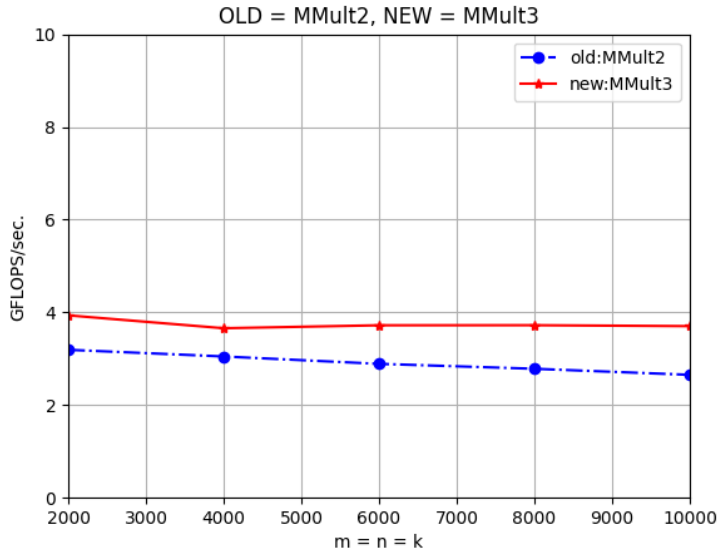
```
static void *dgemm_thread(void *_arg) {
    struct Arg *arg = _arg;
    int m, n, k;
    int lda, ldb, ldc;
    double *a, *b, *c;
    m = arg->m; n = arg->n; k = arg->k;
    lda = arg->lda; ldb = arg->ldb; ldc = arg->ldc;
    a = arg->a; b = arg->b; c = arg->c;
    block_dgemm(m, n, k, a, lda, b, ldb, c, ldc);
    return NULL;
}
```

（其中 `block_dgemm` 是分块矩阵乘函数）

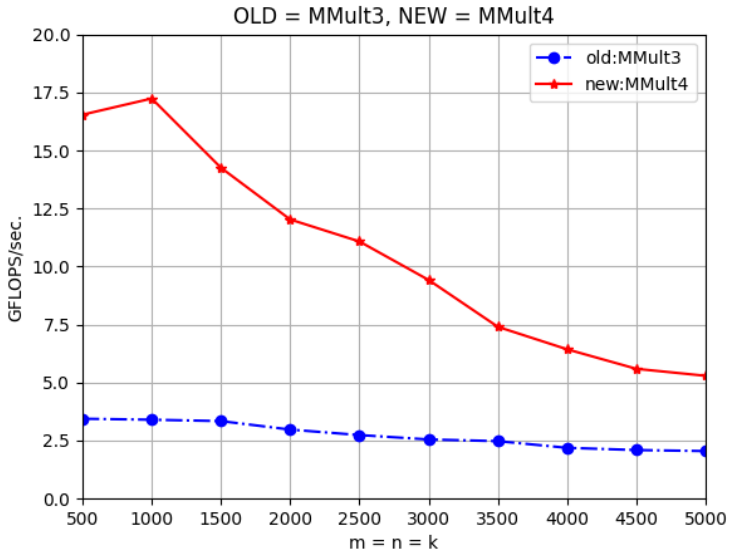
三、性能数据

相对详细的数据分析说明请见实验五的实验报告，下面仅做简单描述。

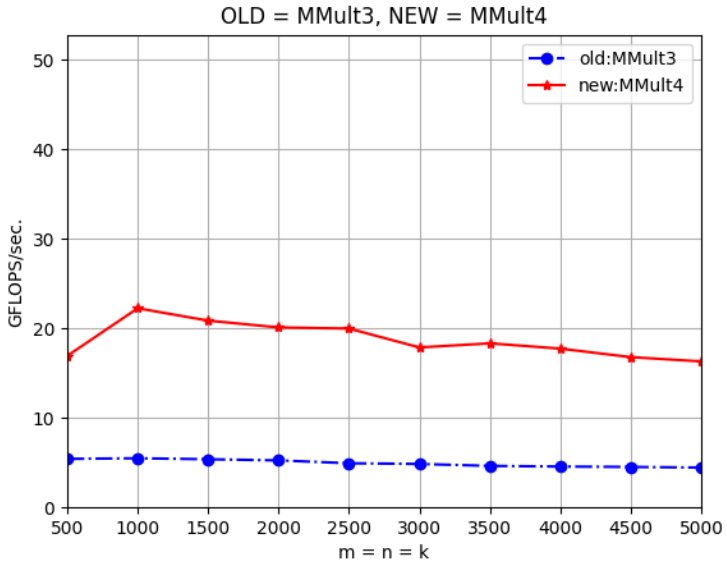
单线程分块矩阵乘（MMult3）相对朴素矩阵乘（MMult2）的性能提升不多：



如果分块实现不正确，不能有效利用缓存，那么多线程的性能提升效果会在矩阵规模增长时下滑（下图 MMult4 为实现过程中错误分块的多线程实现）：



分块正确的多线程实现中，分块和多线程的优势都能充分体现出来：



四、OpenMP 多线程实现

一、实现方案

方案同第三部分 Pthreads 多线程实现，但任务分配和线程回收工作由 OpenMP 来实现。

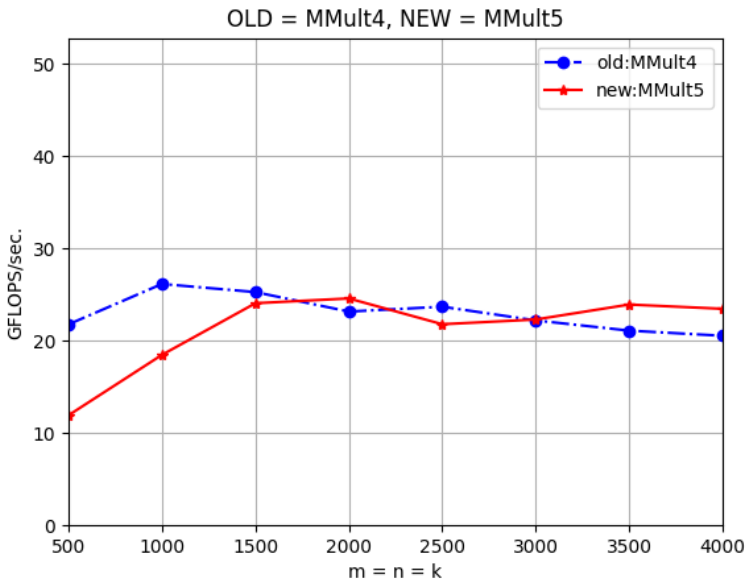
二、关键代码

OpenMP（MMult5.c）局部

```
int i = 0, j = 0, p = 0;
omp_set_num_threads(NUM_THREADS);
for (i = 0; i <= m - BLOCK_SIZE; i += BLOCK_SIZE) {
    #pragma omp parallel for private(p)
    for (j = 0; j <= n - BLOCK_SIZE; j += BLOCK_SIZE) {
        for (p = 0; p <= k - BLOCK_SIZE; p += BLOCK_SIZE) {
            naive_dgemm(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE,
                        &A(i, p), lda, &B(p, j), ldb, &C(i, j), ldc);
        }
        naive_dgemm(BLOCK_SIZE, BLOCK_SIZE, k - p,
                    &A(i, p), lda, &B(p, j), ldb, &C(i, j), ldc);
    }
}
#pragma omp parallel for lastprivate(j) private(p)
for (j = 0; j <= n - BLOCK_SIZE; j += BLOCK_SIZE) {
    for (p = 0; p <= k - BLOCK_SIZE; p += BLOCK_SIZE) {
        naive_dgemm(m - i, BLOCK_SIZE, BLOCK_SIZE,
                    &A(i, p), lda, &B(p, j), ldb, &C(i, j), ldc);
    }
    naive_dgemm(m - i, BLOCK_SIZE, k - p,
                &A(i, p), lda, &B(p, j), ldb, &C(i, j), ldc);
}
```

三、性能对比

Pthreads 实现和 OpenMP 实现的性能相仿。



四、遇到的问题和解决过程

问题一：用 OpenMP 实现之后计算结果不对。

想到应该是变量的共享和私有问题，找到了介绍 OpenMP 指令的入门级手册，仔细学习了各个指令之后用 private 和 lastprivate 子句解决了这个问题。

问题二：用了理论上是错误的变量共享和私有设置，计算结果却是正确的。

突然想到可能是 O1 优化造成了影响，关掉 O1 优化后发现确实发生错误，是 O1 优化恰巧“改正”了程序（实际上这相当于更改了我的设置，应该是“改错”了程序）。对相关变量加上 volatile 修饰符后计算结果错误，说明应该是编译器把变量保持在寄存器中导致之前的“改正”的。