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、单线程分块矩阵乘实现

一、实现方案

朴素矩阵乘原本是三层循环，分块矩阵乘改为六层循环。因为分块矩阵乘法的数学表达式和矩阵乘法的数学表达式在形式上是一样的，因此可以先用类似于朴素矩阵乘的三层循环，将分块矩阵乘法分解为许多两两相乘的矩阵块，然后对这些矩阵块做朴素矩阵乘。

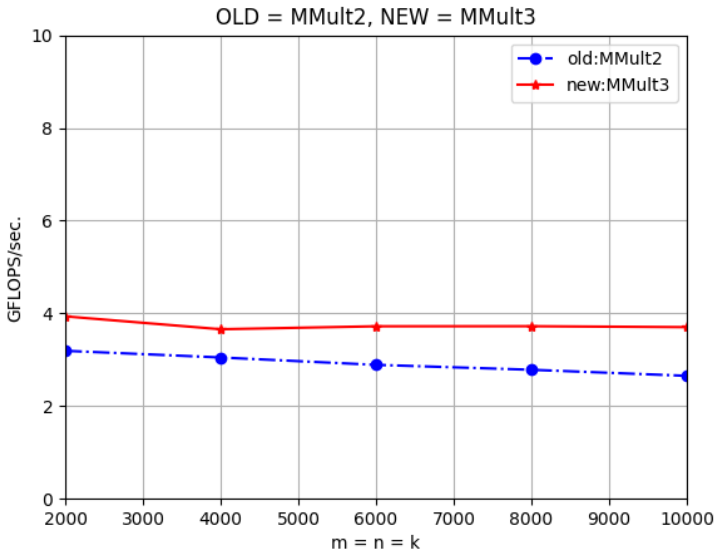
二、关键代码

```
for (i = 0; i <= m; i += BLOCK_SIZE) {
    for (j = 0; j <= n; j += BLOCK_SIZE) {
        for (p = 0; p <= k; p += BLOCK_SIZE) {
            for (int jj = j; jj < j + BLOCK_SIZE; jj++) {
                for (int pp = p; pp < p + BLOCK_SIZE; pp++) {
                    for (int ii = i; ii < i + BLOCK_SIZE; ii++) {
                        C(ii, jj) += A(ii, pp) * B(pp, jj);
                    }
                }
            }
        }
    }
}
```

注：由于上述代码只在矩阵规模刚好是 `BLOCK_SIZE` 的整数倍时才有效，因此实际实现时对边界条件做了很多处理，详细代码请参见文件 MMult3.c。

三、性能对比

下图为单线程分块矩阵乘（MMult3）与朴素矩阵乘（MMult2）的性能对比，可见二者的性能仅相差不大的常数因子。即使在规模很大时矩阵的一列元素已经超过了 L1 缓存大小，二者的 Gflops 数也无明显变化，说明这可能是因为缓存不是性能瓶颈。分块计算虽然在理论上能提高缓存利用率，但在此处效果不佳。



三、多线程分块矩阵乘实现

一、实现方案

将矩阵 B 和 C 都划分为从左到右 16 个部分（本机的逻辑 CPU 数是 16），每个线程计算一个部分：

$$C' = C + AB = \begin{pmatrix} C_0 & C_1 & \cdots & C_{R-1} \end{pmatrix} + A \begin{pmatrix} B_0 & B_1 & \cdots & B_{R-1} \end{pmatrix}$$
$$= \begin{pmatrix} C_0 + AB_0 & C_1 + AB_1 & \cdots & C_{R-1} + AB_{R-1} \end{pmatrix}$$

然后每一部分都按照上面实现的单线程分块矩阵乘进行计算。

实现时，由主进程划分这些部分，然后调用专门用来给子线程执行的入口函数，再由入口函数调用上面实现的单线程分块矩阵乘函数。也可以重写单线程分块矩阵乘函数，减少一次调用。

二、关键代码

用于传参的结构体

```
// Used to pass arguments to child threads
struct Arg {
    int m, n, k;
    int lda, ldb, ldc;
    double *a, *b, *c;
};
```

线程的创建和回收

```
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);
    }
}
```

子线程的入口函数，

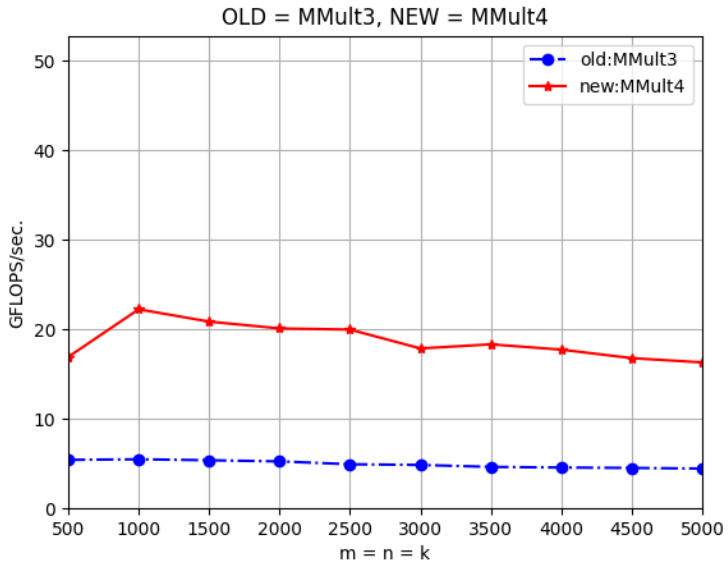
```
static void *dgemm_thread(void *_arg) {
    struct Arg *arg = _arg;
    int m, n, k;
    int lda, ldb, ldc;
    double *a, *b, *c;
    m = arg->m; n = arg->n; k = arg->k;
    lda = arg->lda; ldb = arg->ldb; ldc = arg->ldc;
    a = arg->a; b = arg->b; c = arg->c;
    block_dgemm(m, n, k, a, lda, b,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

单线程分块矩阵乘函数局部（为方便阅读起见，内三层循环改成了函数调用）

```
static void block_dgemm(int m, int n, int k, double *a, int lda,
                        double *b, int ldb,
                        double *c, int ldc) {
    int i = 0, j = 0, p = 0;
    for (i = 0; i <= m - BLOCK_SIZE; i += BLOCK_SIZE) {
        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);
        }
    }
```

三、性能对比

下图为 16 线程分块矩阵乘（MMult4）与单线程分块矩阵乘（MMult3）的性能对比。16 线程版本的运算速度能达到单线程版本的 3 到 4 倍，加速效果明显，而且在矩阵规模很大时也没有出现明显的性能下滑。下图图线有轻微的下降趋势，主要应该是多组测试后 CPU 温度上升，系统降低 CPU 频率所致，可以测试证明这一点。



四、查看 CPU 占用率

在 16 线程分块矩阵乘程序运行时，用 top 工具查看 CPU 占用率等信息：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
25063		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25064		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25065		20	0	766764	634776	1784	R	99.9	7.9	0:07.69	time_MMult_new.
25066		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25067		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25068		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25069		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25070		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25071		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25072		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25073		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25074		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25075		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25076		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
25077		20	0	766764	634776	1784	R	99.9	7.9	0:07.69	time_MMult_new.
25078		20	0	766764	634776	1784	R	99.9	7.9	0:07.70	time_MMult_new.
24934		20	0	766764	634776	1784	S	0.0	7.9	0:01.30	time_MMult_new.

观察发现总共有 17 个线程，也就是主线程（PID=24934）和它创建的 16 个子线程。16 个子线程均为 running 状态，而且 CPU 占用率很高；主线程处于 sleeping 状态，几乎不占用 CPU。

下图是主线程 running 并回收子线程的瞬间。

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
25710		20	0	535604	106756	1644	R	99.9	1.3	0:00.78	time_MMult_new.

二、遇到的问题和解决办法

问题一：实现的单线程分块矩阵乘法没有性能提升，甚至比素朴算法更慢。

在网上搜索相似问题得知，如果内存访问不构成性能瓶颈，那么即便用分块提升缓存利用率，也没多少效果。

问题二：实现的多线程分块矩阵乘法的性能提升效果也很差。

搜索相似问题得知，我原先采用的分块算法不妥当。原先的算法如下，只有5个循环（令A的几整行与B的几整列相乘，而非A的小块与B的小块相乘）：

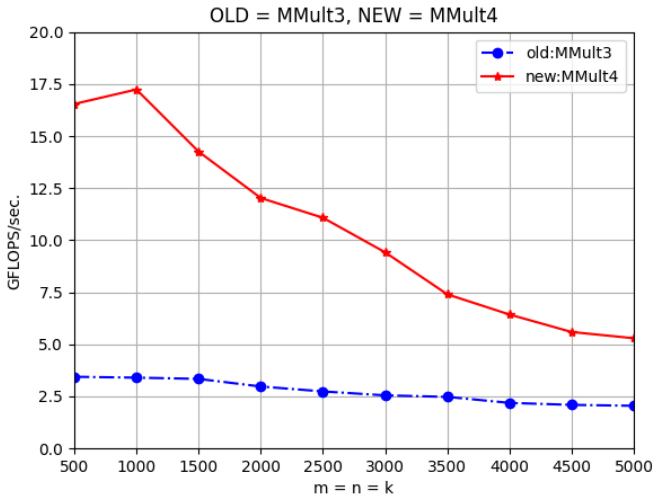
```
for (int i = 0; i < y; i++) {
    for (int j = 0; j < x; j++) {
        block_c = &C(i * BLOCK_SIZE, j * BLOCK_SIZE);
        block_a = &A(i * BLOCK_SIZE, 0);
        block_b = &B(0, j * BLOCK_SIZE);
        full_block_mult(k, block_a, lda, block_b, ldb, block_c, ldc);
    }
}
```

其中函数 `full_block_mult` 是朴素矩阵乘。

这个算法虽然也分块了，但其实几乎不能提高缓存利用率。每次调用朴素矩阵乘函数，大约发生$B^2k/8$次cache miss（一条缓存行能存放8个双精度浮点数），其中B是每块边长。有mn/B^2块，那么总共会发生mnk/8次cache miss。可是朴素算法也要发生这么多次cache miss，因此缓存利用率没有提高。实际上，这种分块方法，由于矩阵A和B仍然太大，只有矩阵C有可能可以享受到分块的好处，而这个优化是微乎其微的。

采用正确的6层循环的分块方法，如果分块后的矩阵A、B、C都可以完全放入缓存中，可以算出总共只会发生大约3mnk/8B次cache miss，理论上只要B大于3，就能明显降低miss rate，因此可以用来优化矩阵乘法。

按照原先的错误算法，得到的性能图线如下（让我烦恼了很久）：



注：与上面的图不同，这张图中的单线程分块算法（MMult3）的分块方法是错误的，因而也更慢。

图线表明，虽然在矩阵规模较小时它的性能接近正确算法，但随矩阵规模增大，缓存逐渐成为性能瓶颈，而错误算法并没有改善缓存问题，因此其性能逐步下滑。正确算法，由于解决了缓存问题，能在规模增大时保持住提升性能的效果。