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 8 日

DGEMM 优化

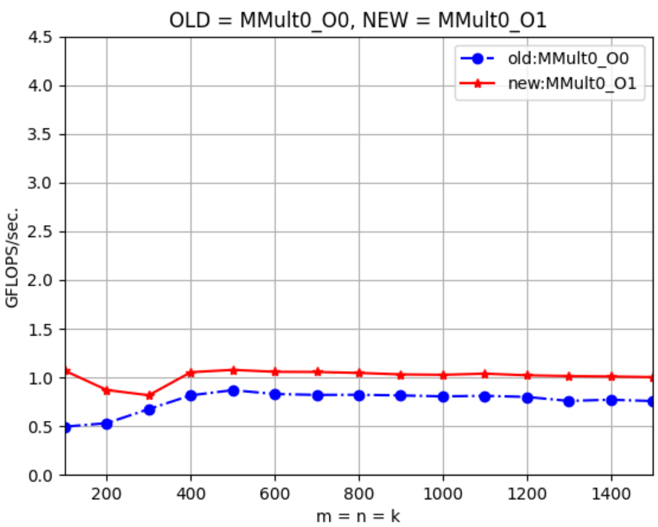
〇、朴素矩阵乘（MMult0.c）

描述：按 ijp 顺序迭代。

```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);
            }
        }
    }
}
```

优化一、利用指针、寄存器避免反复访问 `i`、`j`、`k`、`C(i, j)`的内存。

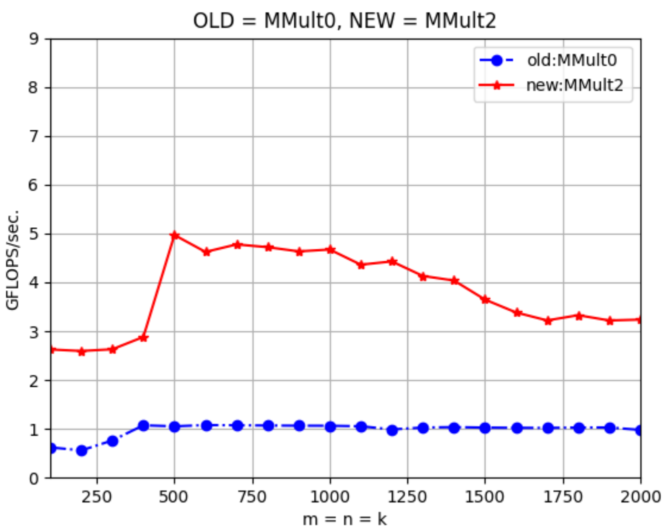
描述：为方便起见，用 O1 优化代替。后续优化都打开 O1 优化。后续优化中也会用 register 关键字指示编译器。



优化二、更改迭代顺序

描述：按 jpi 顺序迭代，提高缓存利用率。

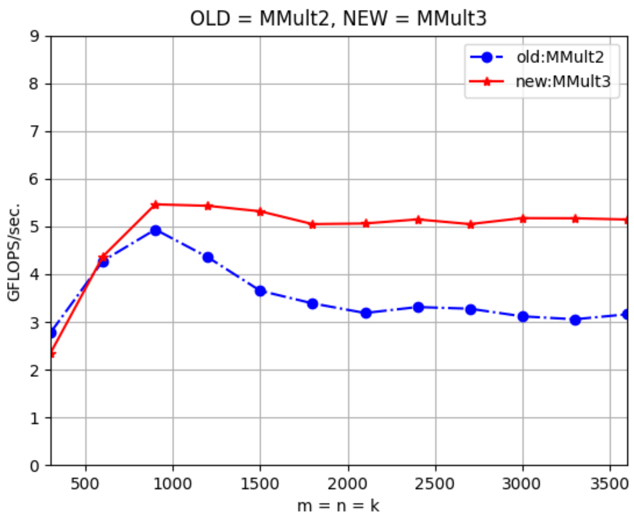
```
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb, double *c, int ldc) {
    int i, j, p;
    for (j = 0; j < n; j++) {
        for (p = 0; p < k; p++) {
            for (i = 0; i < m; i++) {
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
           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



优化三、分块矩阵乘（MMult3.c）

描述：外层分块按 ijp 顺序迭代，内层计算按 jpi 顺序迭代。每个分块都可以全部存入缓存，减少了矩阵规模增大造成的大量 cache miss。

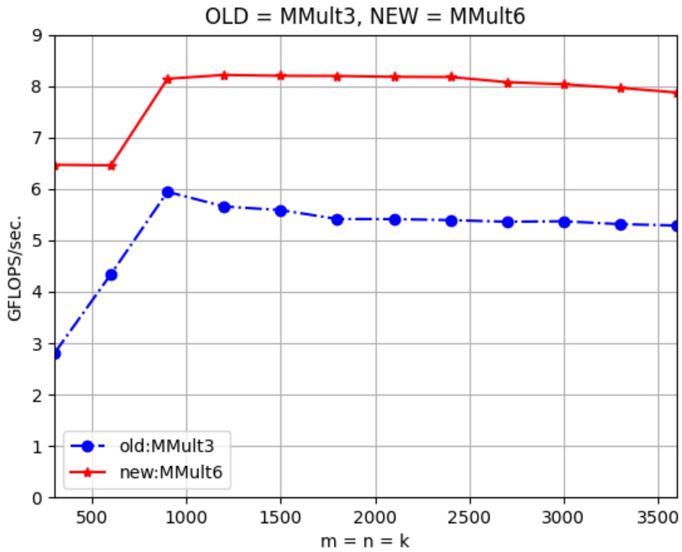
```c
inline static void block_dgemm(int m, int n, int k, double *a, int lda,
                               double *b, int ldb,
                               double *c, int ldc) {
    for (int i = 0; i < m; i += BLOCK_SIZE) {
        int len_i = BLOCK_SIZE < m - i ? BLOCK_SIZE : m - i;
        for (int j = 0; j < n; j += BLOCK_SIZE) {
            int len_j = BLOCK_SIZE < n - j ? BLOCK_SIZE : n - j;
            for (int p = 0; p < k; p += BLOCK_SIZE) {
                int len_p = BLOCK_SIZE < k - p ? BLOCK_SIZE : k - p;
                naive_dgemm(len_i, len_j, len_p, &A(i, p), lda, &B(p, j), ldb, &C(i, j), ldc);
            }
        }
    }
}

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



优化四：一次计算四组乘法和加法（MMult6.c）

描述：一次循环内更新 C(i,j)、C(i,j+1)、C(i,j+2)、C(i,j+3)，将寄存器读取 A(i,p)的次数减少到 1/4。

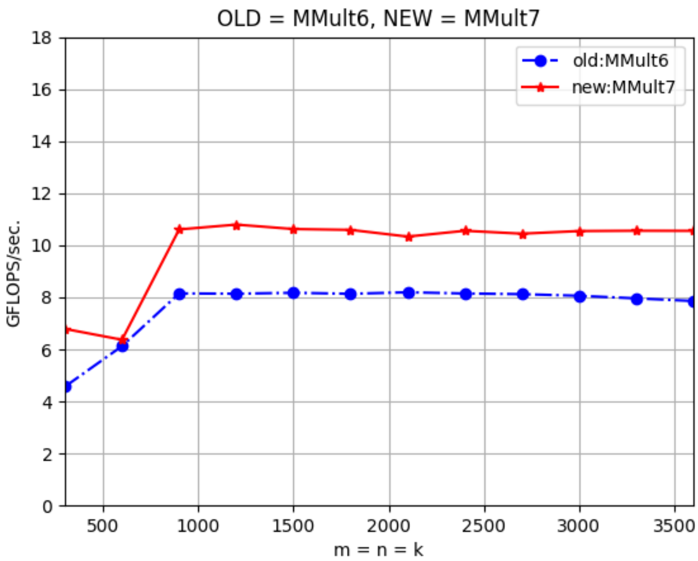
```c
int j;
for (j = 0; j <= n - 4; j += 4) {
    for (int p = 0; p < k; p++) {
        register
        double b_pj  = B(p, j),      b_pj1 = B(p, j + 1),
               b_pj2 = B(p, j + 2), b_pj3 = B(p, j + 3);
        for (int i = 0; i < m; i++) {
            register double a_ip = A(i, p);
            C(i, j)     += a_ip * b_pj;
            C(i, j + 1) += a_ip * b_pj1;
            C(i, j + 2) += a_ip * b_pj2;
            C(i, j + 3) += a_ip * b_pj3;
        }
    }
}
for (; j < n; j++) {
    for (int p = 0; p < k; p++) {
        for (int i = 0; i < m; i++) {
            C(i, j) += A(i, p) * B(p, j);
        }
    }
}
```



优化五：展开循环（MMult7.c）

描述：把循环顺序修改为 jip，在一次计算四组乘法和加法的基础上，对循环做展开（unrolling），变为一次计算八组乘法和加法。

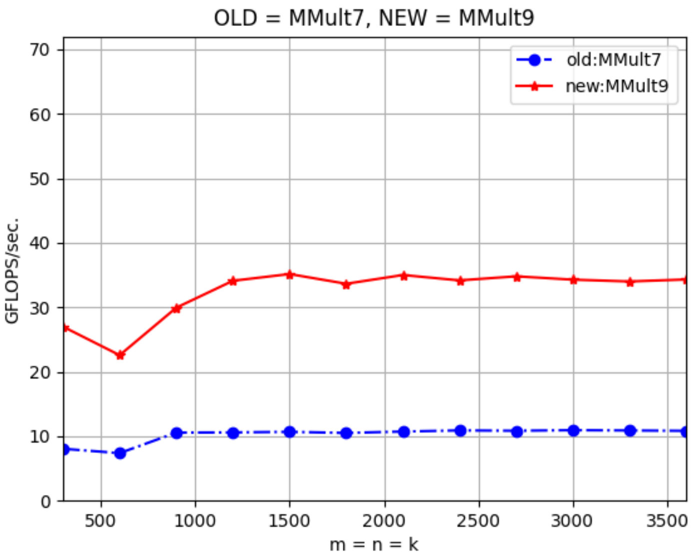
```
for (j = 0; j <= n - 4; j += 4) {
    int i;
    for (i = 0; i <= m - 2; i += 2) {
        register
        double c_i0j0 = C(i, j),         c_i0j1 = C(i, j + 1),
               c_i0j2 = C(i, j + 2),     c_i0j3 = C(i, j + 3),
               c_i1j0 = C(i + 1, j),     c_i1j1 = C(i + 1, j + 1),
               c_i1j2 = C(i + 1, j + 2), c_i1j3 = C(i + 1, j + 3);
        for (int p = 0; p < k; p++) {
            register
            double a_i0p0 = A(i, p), a_i1p0 = A(i + 1, p);
            c_i0j0 += a_i0p0 * B(p, j);
            c_i0j1 += a_i0p0 * B(p, j + 1);
            c_i0j2 += a_i0p0 * B(p, j + 2);
            c_i0j3 += a_i0p0 * B(p, j + 3);
            c_i1j0 += a_i1p0 * B(p, j);
            c_i1j1 += a_i1p0 * B(p, j + 1);
            c_i1j2 += a_i1p0 * B(p, j + 2);
            c_i1j3 += a_i1p0 * B(p, j + 3);
        }
        C(i, j)         = c_i0j0, C(i, j + 1)     = c_i0j1,
        C(i, j + 2)     = c_i0j2, C(i, j + 3)     = c_i0j3,
        C(i + 1, j)     = c_i1j0, C(i + 1, j + 1) = c_i1j1,
        C(i + 1, j + 2) = c_i1j2, C(i + 1, j + 3) = c_i1j3;
    }
```



优化五：向量化（MMult9.c）

描述：使用 AVX2 和 FMA 指令集提供的 256 位向量运算和融合乘加运算。

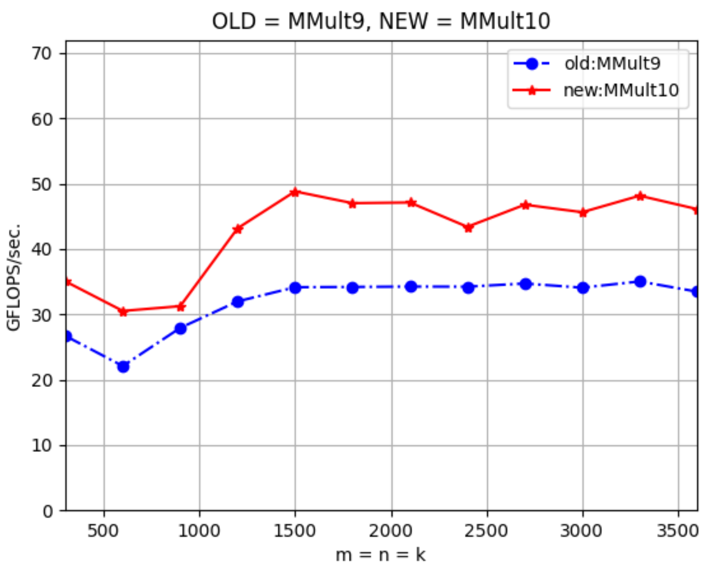
```
for (i = 0; i <= m - 4; i += 4) {
    __m256d c_i0123j0, c_i0123j1, c_i0123j2, c_i0123j3;
    c_i0123j0 = _mm256_loadu_pd(&C(i, j));
    c_i0123j1 = _mm256_loadu_pd(&C(i, j + 1));
    c_i0123j2 = _mm256_loadu_pd(&C(i, j + 2));
    c_i0123j3 = _mm256_loadu_pd(&C(i, j + 3));

    for (int p = 0; p < k; p++) {
        __m256d b_p0j0, b_p0j1, b_p0j2, b_p0j3;
        b_p0j0 = _mm256_movedup_pd(_mm256_broadcast_pd((__m128d *) &B(p, j)));
        b_p0j1 = _mm256_movedup_pd(_mm256_broadcast_pd((__m128d *) &B(p, j + 1)));
        b_p0j2 = _mm256_movedup_pd(_mm256_broadcast_pd((__m128d *) &B(p, j + 2)));
        b_p0j3 = _mm256_movedup_pd(_mm256_broadcast_pd((__m128d *) &B(p, j + 3)));
        __m256d a_i0123p0;
        a_i0123p0 = _mm256_loadu_pd(&A(i, p));

        c_i0123j0 = _mm256_fmadd_pd(a_i0123p0, b_p0j0, c_i0123j0);
        c_i0123j1 = _mm256_fmadd_pd(a_i0123p0, b_p0j1, c_i0123j1);
        c_i0123j2 = _mm256_fmadd_pd(a_i0123p0, b_p0j2, c_i0123j2);
        c_i0123j3 = _mm256_fmadd_pd(a_i0123p0, b_p0j3, c_i0123j3);
    }
    _mm256_storeu_pd(&C(i, j),     c_i0123j0);
    _mm256_storeu_pd(&C(i, j + 1), c_i0123j1);
    _mm256_storeu_pd(&C(i, j + 2), c_i0123j2);
    _mm256_storeu_pd(&C(i, j + 3), c_i0123j3);
}
```



优化六：增加向量计算密度（MMult10.c）

描述：进一步展开循环，一次内层循环计算三十二组乘加运算，进行八次 256 位向量运算。这能提高向量计算对向量读取的操作次数的比值，提高向量运算单元利用率。

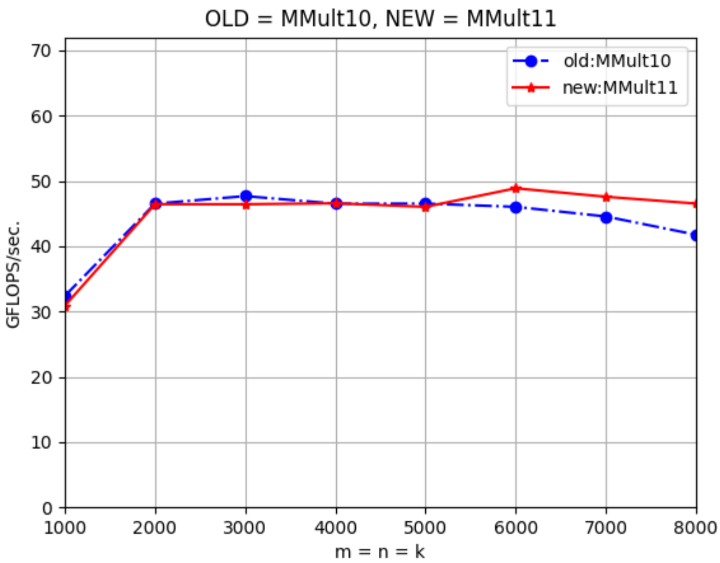
```
for (int p = 0; p < k; p++) {
    __m256d b_p0j0, b_p0j1, b_p0j2, b_p0j3;
    b_p0j0 = _mm256_set1_pd(B(p, j));
    b_p0j1 = _mm256_set1_pd(B(p, j + 1));
    b_p0j2 = _mm256_set1_pd(B(p, j + 2));
    b_p0j3 = _mm256_set1_pd(B(p, j + 3));
    __m256d a_i0123p0, a_i4567p0;
    a_i0123p0 = _mm256_loadu_pd(&A(i, p));
    a_i4567p0 = _mm256_loadu_pd(&A(i + 4, p));

    c_i0123j0 = _mm256_fmadd_pd(a_i0123p0, b_p0j0, c_i0123j0);
    c_i0123j1 = _mm256_fmadd_pd(a_i0123p0, b_p0j1, c_i0123j1);
    c_i0123j2 = _mm256_fmadd_pd(a_i0123p0, b_p0j2, c_i0123j2);
    c_i0123j3 = _mm256_fmadd_pd(a_i0123p0, b_p0j3, c_i0123j3);
    c_i4567j0 = _mm256_fmadd_pd(a_i4567p0, b_p0j0, c_i4567j0);
    c_i4567j1 = _mm256_fmadd_pd(a_i4567p0, b_p0j1, c_i4567j1);
    c_i4567j2 = _mm256_fmadd_pd(a_i4567p0, b_p0j2, c_i4567j2);
    c_i4567j3 = _mm256_fmadd_pd(a_i4567p0, b_p0j3, c_i4567j3);
}
```



优化七：内存对齐（MMult11.c）

描述：将矩阵 A、B、C 按 64 个字节（缓存行大小）进行内存对齐。这样可以将向量读取、写入指令由非对齐改为对齐，并减少一些 cache miss，不过实际效果不佳。

```
int lda_aligned = (m + 7) & 0xfffffffc;
double *a_aligned = (double *) aligned_alloc(64, lda_aligned * k * sizeof(double));
#define A_ALN(i, j) a_aligned[(i) + lda_aligned * (j)]
for (int j = 0; j < k; j++) {
    for (int i = 0; i < m; i++) {
        A_ALN(i, j) = A(i, j);
    }
}
```



优化八：多线程（MMult12.c）

用 OpenMP 提供的编译指令，多线程并行计算。

```
#pragma omp parallel for schedule(dynamic) num_threads(16)
for (int j = 0; j < n; j += Nc) {
    int len_j = Nc < n - j ? Nc : n - j;
    for (int p = 0; p < k; p += Kc) {
        int len_p = Kc < k - p ? Kc : k - p;
        for (int i = 0; i < m; i += Mc) {
            int len_i = Mc < m - i ? Mc : m - i;
            inner_dgemm(len_i, len_j, len_p, &A_ALN(i, p), lda_aligned,
                        &B_ALN(p, j), ldb_aligned, &C_ALN(i, j), ldc_aligned);
        }
    }
}
```