華中科技大學

研究生课程报告

姓　　名：赖家晨

学　　号：M202474131

系、年级：计算机科技与技术系 2024 级

类　　别：课程报告

报告科目：高级系统结构实验

2025 年 1 月 15 日

1 算法介绍

1.1 暴力算法

设 $A=[a_{ij}], B=[b_{ij}] \in \mathbb{R}^{n\times n}$，则 $C=[c_{ij}]=AB$，其中

$$c_{ij}=\sum_{k=1}^{n} a_{ik}b_{kj}.$$

因此矩阵乘法最简单的方式就是对于每个 C_{ij} 的计算，都枚举 k 从 1 到 n，总运算量为：n^3（乘法）$+\ n^3$（加法）$=2n^3$。暴力算法伪代码如代码 1所示。

Algorithm 1 矩阵乘积: IJK 顺序

for $i=1$ to n do
    for $j=1$ to n do
        for $k=1$ to n do
            $c_{ij}=c_{ij}+a_{ik}b_{kj}$
        end for
    end for
end for

1.2 调换循环顺序

对于最普通的实现方式（顺序：ijk），它是依据计算 C 中的每个元素。当计算 C 中任何一个元素时，需要将 A 对应的行与 B 对应的列依次相乘加和。之前已假设过，矩阵 A 对应行不断向右移动时，内存访问是连续的。但 B 相对应的列已经断开间隔了 n 次，故 A 只有连续 1 次，故总共是 $n+1$ 次。这样计算 C 中所有元素，跳转了 n^2 次，但刚才没有计算 C 的跳转次数，加以上后是 n^3+n^2+n。(注意，在计算 C 中每行的最后一个元素时，A 是从对应行末尾转到下一行开头。而如果以顺序 ijk 实现，它将 C 中元素一行一行计算。当计算 C 中任意一行的第一个元素时，先访问 A 中相应行的第一列元素，和 B 第一列的第一行元素，然后 B 不断往右移，计算完成后转到一行，因此这样计算 C 的这一行元素后，恰好按顺序将 B 遍历一遍，间断了 n 次，但恰好从左在右遍历了 A 的对应行，间断了 1 次。故算完 C 的所有行后，跳转了 n^2+n 次，算上跳跃次数是 $2n^2+n$ 次。而内存访问的不连续，会导致 cache 命中率不高，从而影响程序的性能。IJK 顺序相较于 IKJ 顺序，有更少的跳跃次数，相应的访存会更连续，程序执行速度会更快。故，可以调换代码 1中的循环顺序，将原来的 IJK 顺序改成 IKJ 顺序。

Algorithm 2 矩阵乘积: IKJ 顺序

```
for i = 1 to n do
    for k = 1 to n do
        for j = 1 to n do
            c_ij = c_ij + a_ik b_kj
        end for
    end for
end for
```

1.3 循环展开

矩阵乘法通过循环展开和局部变量优化，实现了高效的矩阵乘法计算。具体的，可以将内层循环展开 8 次，一次性处理多个数据块，利用程序的局部性，减少循环控制的开销，从而提高计算效率。具体代码实现如下：

```
void MatrixMultiply() {
    for (int i = 0; i < MATRIXSIZE; ++i) {
        for (int j = 0; j < MATRIXSIZE; ++j) {
            int sum = 0; // 使用局部变量存储累加结果
            for (int k = 0; k < MATRIXSIZE; k += 8) {
                sum += matrix1[i][k]     * matrix2[k][j];
                sum += matrix1[i][k + 1] * matrix2[k + 1][j];
                sum += matrix1[i][k + 2] * matrix2[k + 2][j];
                sum += matrix1[i][k + 3] * matrix2[k + 3][j];
                sum += matrix1[i][k + 4] * matrix2[k + 4][j];
                sum += matrix1[i][k + 5] * matrix2[k + 5][j];
                sum += matrix1[i][k + 6] * matrix2[k + 6][j];
                sum += matrix1[i][k + 7] * matrix2[k + 7][j];
            }
            result[i][j] = sum; // 将结果存回 result[i][j]
        }
    }
}
```

1.4 Strassen 算法

德国数学家 Strassen 在 1969 年提出了计算矩阵乘积的快速算法，将运算量降为约 $O(n^{2.81})$。具体来说，Strassen 采用分而治之的思想，先将矩阵 A，B 进行 2 ×

2 分块，即

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

则 $C = AB$ 也可以写成 2×2 分块形式，即

$$\begin{aligned} C_{11} &= X_1 + X_4 - X_5 + X_7, \\ C_{12} &= X_3 + X_5, \\ C_{21} &= X_2 + X_4, \\ C_{22} &= X_1 + X_3 - X_2 + X_6, \end{aligned}$$

其中

$$\begin{aligned} X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ X_2 &= (A_{21} + A_{22})B_{11}, \\ X_3 &= A_{11}(B_{12} - B_{22}), \\ X_4 &= A_{22}(B_{21} - B_{11}), \\ X_5 &= (A_{11} + A_{12})B_{22}, \\ X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

需要 7 次子矩阵的乘积和 18 次子矩阵加法。假定采用普通方法计算子矩阵的乘积，即需要 $(n/2)^3$ 乘法和 $(n/2)^3$ 次加法，则采用 Strassen 方法计算 A 和 B 乘积的运算量为

$$7 \times ((n/2)^3 + (n/2)^3) + 18 \times (n/2)^2 = \frac{7}{4}n^3 + \frac{9}{2}n^2.$$

大约是普通矩阵乘积运算量的 $\frac{7}{8}$。在计算子矩阵的乘积时，我们仍然可以采用 Strassen 算法。依此类推，于是，由递归思想可知，则总运算量大约为（只考虑最高次项，并假定 n 可以不断对分下去）

$$7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807\ldots}$$

但是在实际代码中，Strassen 算法每次递归都要为 X_1、X_2、X_3、X_4、X_5、X_6 和 X_7 开辟内存空间，并且需要拷贝子矩阵，时间复杂度常数过大。在计算 1024 阶 32 位整数的矩阵乘法时，如果完全递归到 4 阶矩阵，代码运行时间甚至不如代码 2暴力算法。因此，在实现时，当阶数小于 512 阶时，采用代码 2中的算法，不继续进行递归。具体实现代码如下：

```
void Strassen(T **A, T **B, T **C, int size) {
    // 当矩阵大小小于等于256时，直接使用普通矩阵运算
    if(size == 256) {
        MatrixMultiply(A, B, C, size); // 直接调用普通矩阵乘法
```

```
        return;
    }

    // 将矩阵大小减半，递归拆分
    size /= 2;

    // 为每个子矩阵和中间结果分配内存
    T **A11 = new T*[size];
    T **A12 = new T*[size];
    T **A21 = new T*[size];
    T **A22 = new T*[size];
    T **B11 = new T*[size];
    T **B12 = new T*[size];
    T **B21 = new T*[size];
    T **B22 = new T*[size];

    T **X1 = new T*[size];
    T **X2 = new T*[size];
    T **X3 = new T*[size];
    T **X4 = new T*[size];
    T **X5 = new T*[size];
    T **X6 = new T*[size];
    T **X7 = new T*[size];

    T **T1 = new T*[size];
    T **T2 = new T*[size];

    // 分配内存并初始化子矩阵
    for(int i = 0; i < size; ++i) {
        A11[i] = new T[size];
        A12[i] = new T[size];
        A21[i] = new T[size];
        A22[i] = new T[size];
        B11[i] = new T[size];
        B12[i] = new T[size];
        B21[i] = new T[size];
        B22[i] = new T[size];
```

```
        X1[i] = new T[size];
        X2[i] = new T[size];
        X3[i] = new T[size];
        X4[i] = new T[size];
        X5[i] = new T[size];
        X6[i] = new T[size];
        X7[i] = new T[size];

        T1[i] = new T[size];
        T2[i] = new T[size];

        // 初始化所有矩阵元素为0
        memset(X1[i], 0, sizeof(T) * size);
        memset(X2[i], 0, sizeof(T) * size);
        memset(X3[i], 0, sizeof(T) * size);
        memset(X4[i], 0, sizeof(T) * size);
        memset(X5[i], 0, sizeof(T) * size);
        memset(X6[i], 0, sizeof(T) * size);
        memset(X7[i], 0, sizeof(T) * size);
    }

    // 将输入矩阵 A 和 B 拆分成 4 个子矩阵
    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < size; ++j) {
            A11[i][j] = A[i][j];                  // A 的左上子矩阵
            A12[i][j] = A[i][j + size];           // A 的右上子矩阵
            A21[i][j] = A[i + size][j];           // A 的左下子矩阵
            A22[i][j] = A[i + size][j + size];    // A 的右下子矩阵

            B11[i][j] = B[i][j];                  // B 的左上子矩阵
            B12[i][j] = B[i][j + size];           // B 的右上子矩阵
            B21[i][j] = B[i + size][j];           // B 的左下子矩阵
            B22[i][j] = B[i + size][j + size];    // B 的右下子矩阵
        }
    }

```

```
    // 计算 Strassen 的 7 个中间矩阵
    MatrixAdd(A11, A22, T1, size);          // A11 + A22
    MatrixAdd(B11, B22, T2, size);          // B11 + B22
    Strassen(T1, T2, X1, size);             // X1 = (A11 + A22) * (B11 + B22)

    MatrixAdd(A21, A22, T1, size);          // A21 + A22
    Strassen(T1, B11, X2, size);            // X2 = (A21 + A22) * B11

    MatrixSub(B12, B22, T1, size);          // B12 - B22
    Strassen(A11, T1, X3, size);            // X3 = A11 * (B12 - B22)

    MatrixSub(B21, B11, T1, size);          // B21 - B11
    Strassen(A22, T1, X4, size);            // X4 = A22 * (B21 - B11)

    MatrixAdd(A11, A12, T1, size);          // A11 + A12
    Strassen(T1, B22, X5, size);            // X5 = (A11 + A12) * B22

    MatrixSub(A21, A11, T1, size);          // A21 - A11
    MatrixAdd(B11, B12, T2, size);          // B11 + B12
    Strassen(T1, T2, X6, size);             // X6 = (A21 - A11) * (B11 + B12)

    MatrixSub(A12, A22, T1, size);          // A12 - A22
    MatrixAdd(B21, B22, T2, size);          // B21 + B22
    Strassen(T1, T2, X7, size);             // X7 = (A12 - A22) * (B21 + B22)

    // 合并 7 个中间结果得到最终的矩阵 C
    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; ++j) {
            // C 的四个子矩阵
            C[i][j] = X1[i][j] + X4[i][j] - X5[i][j] + X7[i][j];
```

```
            C[i][j + size] = X3[i][j] + X5[i][j];
            C[i + size][j] = X2[i][j] + X4[i][j];
            C[i + size][j + size] = X1[i][j] + X3[i][j] - X2[i][j] + X6[i][j];
        }
    }

    // 释放动态分配的内存
    for(int i = 0; i < size; ++i) {
        delete[] A11[i];
        delete[] A12[i];
        delete[] A21[i];
        delete[] A22[i];
        delete[] B11[i];
        delete[] B12[i];
        delete[] B21[i];
        delete[] B22[i];

        delete[] X1[i];
        delete[] X2[i];
        delete[] X3[i];
        delete[] X4[i];
        delete[] X5[i];
        delete[] X6[i];
        delete[] X7[i];

        delete[] T1[i];
        delete[] T2[i];
    }
}
```

1.5 SIMD

SIMD 的全称叫做，单指令集多数据（Single Instruction Multiple Data）。最直观的理解就是，向量计算。比如一个加法指令周期只能算一组数（一维向量相加），使用 SIMD 的话，一个加法指令周期可以同时算多组数（n 维向量相加），二者用时基本相等，极大地提高了运算效率。现代 CPU（例如 Intel 的 AVX、AVX2、AVX-512 指

令集）提供了 SIMD 指令集，允许一次性处理 256 位或 512 位的数据。这些指令可以在一条指令内对多个元素进行操作，显著加速计算。

使用 SIMD 进行加速的过程：

1. **数据并行化**：将矩阵中的数据分成多个块，利用 SIMD 一次性处理多个数据块。比如，使用 256 位的 AVX 指令集，可以一次处理 8 个整数或 8 个浮点数（假设每个元素为 32 位整型或单精度浮点型）。

2. **并行化乘法**：对于每个 $C[i][j]$，我们将 $A[i][k]$ 和 $B[k][j]$ 逐个乘积并累加，使用 SIMD 来同时计算多个乘积。例如，使用 256 位寄存器存储多个元素，使用 SIMD 指令同时计算多个乘积。

3. **并行化加法**：SIMD 还可以加速累加过程。计算 $C[i][j]$ 时，多个乘积的结果需要加和，可以通过 SIMD 指令来并行化加法运算（例如，使用 _mm256_add_epi32 来执行多个整数的加法）。

4. **数据加载与存储**：使用高效的内存加载和存储指令（如 loadu 和 storeu）从内存中加载矩阵元素，避免数据访问延迟。这是利用 SIMD 加速矩阵运算的关键，因为内存访问的速度往往成为计算瓶颈。

由于使用的 CPU 支持 AVX2 和 AVX512 两种 SIMD 指令，所以这里对这两种方法都进行了实验。

1.5.1 AVX2

AVX2（Advanced Vector Extensions 2）是 Intel 提供的 SIMD（Single Instruction, Multiple Data）指令集，用于提高计算密集型操作的性能。AVX2 支持 256 位的向量操作，允许同时处理 8 个 32 位整数或 4 个 64 位整数。代码通过并行化矩阵乘法中的元素乘加运算，显著提高了效率。我们可以在循环展开的基础上，利用 AVX2 指令集将 A，B 矩阵元素加载到 AVX 向量寄存器中，一次执行 4 个元素的乘积和累加运算，最后将结果写回内存中的目标矩阵 C。具体流程如下：

- **初始化结果矩阵**：首先，矩阵 C 被初始化为零。使用 memset 函数对矩阵 C 的每一行进行清零操作。

- **外循环遍历矩阵 A 和 C**：接下来，代码使用两个外层循环，m 和 k，分别遍历矩阵 A 和 C，每次处理 4 个元素。这样做是为了使得矩阵块的大小与 AVX2 向量的大小匹配，从而更好地利用 AVX2 寄存器进行并行计算。

- **AVX 向量的定义和初始化**：为了执行向量化操作，代码定义了 4 个 AVX2 向量 $C0v, C1v, C2v, C3v$ 来分别存储矩阵 C 中的结果。这些向量将存储 $C[m : m+3][k : k+3]$ 中的四个结果。

- **内循环计算矩阵元素**：
  - 对于矩阵 B 和 A 的每一对元素，代码使用 _mm256_loadu_si256 加载 B 矩阵的列和 A 矩阵的行到 AVX2 向量中。
  - 对于 A 矩阵中的每一行 $A[m][n], A[m+1][n], A[m+2][n], A[m+3][n]$，使用 _mm256_set1_epi64x 将这些元素扩展成 256 位的向量（每个向量中都包含 4 个相同的值）。
  - 使用 _mm256_mullo_epi64 指令执行逐元素乘法操作，计算矩阵元素乘积。
  - 然后使用 _mm256_add_epi64 对计算结果进行累加，将每个结果添加到 $C0v, C1v, C2v, C3v$ 中。
- **将结果存回矩阵 C**：最后，代码使用 _mm256_storeu_si256 将计算结果存回 C 矩阵的对应位置。

使用 SIMD 加速矩阵乘法代码如下：

```
void MatrixMultiplyAVX(T **A, T **B, T **C, int size) {
    // 遍历矩阵 A 和 B
    for (int m = 0; m < size; m += 4) {
        for (int k = 0; k < size; k += 4) {
            // 定义 AVX 向量
            __m256i C0v, C1v, C2v, C3v;
            __m256i B0v;

            // 初始化累加器
            C0v = _mm256_setzero_si256();
            C1v = _mm256_setzero_si256();
            C2v = _mm256_setzero_si256();
            C3v = _mm256_setzero_si256();

            // 对矩阵 B 和 A 进行遍历，计算乘积并累加到 C[m:m+3][k:k+3]
            for (int n = 0; n < size; ++n) {
                // 加载 B[n][k:k+3] 到 AVX 向量
                B0v = _mm256_loadu_si256((__m256i*)&B[n][k]);

                // 加载 A[m:m+3][n] 到 AVX 向量
                __m256i vecA0 = _mm256_set1_epi64x(A[m][n]);
                __m256i vecA1 = _mm256_set1_epi64x(A[m + 1][n]);
```

```
                __m256i vecA2 = _mm256_set1_epi64x(A[m + 2][n]);
                __m256i vecA3 = _mm256_set1_epi64x(A[m + 3][n]);

                // 逐元素乘法并累加
                C0v = _mm256_add_epi64(C0v, _mm256_mullo_epi64(vecA0, B0v));
                C1v = _mm256_add_epi64(C1v, _mm256_mullo_epi64(vecA1, B0v));
                C2v = _mm256_add_epi64(C2v, _mm256_mullo_epi64(vecA2, B0v));
                C3v = _mm256_add_epi64(C3v, _mm256_mullo_epi64(vecA3, B0v));
            }

            // 将结果存回 C[m:m+3][k:k+3]
            _mm256_storeu_si256((__m256i*)&C[m][k], C0v);
            _mm256_storeu_si256((__m256i*)&C[m + 1][k], C1v);
            _mm256_storeu_si256((__m256i*)&C[m + 2][k], C2v);
            _mm256_storeu_si256((__m256i*)&C[m + 3][k], C3v);
        }
    }
}
```

1.5.2 AVX512

AVX-512（Advanced Vector Extensions 512）是 Intel 提供的 SIMD（Single Instruction, Multiple Data）指令集，支持 512 位的寄存器，可以同时处理 8 个 64 位整数或 16 个 32 位整数。它显著提高了大规模数据处理的效率，特别是矩阵运算等计算密集型任务。与 AVX2 相比，AVX-512 拥有更大的向量寄存器宽度，因此能够同时处理更多数据，进一步提高计算效率。主要思路是通过将矩阵元素加载到 AVX-512 向量寄存器中，一次执行 8 个元素的乘积和累加运算，最后将结果写回内存中的目标矩阵 C。具体流程如下：

- **初始化矩阵 C**：首先，矩阵 C 被初始化为零。为了确保每个元素被初始化为零，代码对每一块（8x8）的子矩阵进行处理。
- **外循环遍历矩阵 A 和 C**：使用两个外循环，m 和 k，分别遍历矩阵 A 和 C。每次处理 8 个元素的矩阵块。这样做是为了确保 AVX-512 寄存器的 512 位宽度能够充分利用，通过同时处理多个矩阵元素来加速计算。

- **定义 AVX-512 向量**:定义了 8 个 AVX-512 向量 $C0v, C1v, C2v, C3v, C4v, C5v, C6v, C7v$ 来存储矩阵 C 中的计算结果。这些向量将存储 $C[m:m+7][k:k+7]$ 的计算结果，每次 8 个元素进行并行计算。
- **内循环计算矩阵元素**：
  - **加载矩阵 B 和 A 的元素**：
    * B 矩阵的每一列（$B[n][k:k+7]$）被加载到 $B0v$ 向量中。
    * A 矩阵的每一行（$A[m:m+7][n]$）被加载到 8 个不同的向量中（$vecA0, vecA1, ..., vecA7$）。每个 $vecA$ 向量包含了 4 个相同的 64 位整数。
  - **逐元素乘法并累加**：使用 _mm512_mullo_epi64 对矩阵 A 和 B 中的元素进行逐元素乘法操作，结果会被累加到 8 个 C 向量中（$C0v, C1v, ..., C7v$）。
  - **存储结果**：使用 _mm512_storeu_si512 将计算结果存回矩阵 C 中。
- **将结果存回矩阵 C**：在内循环结束后，8 个向量 $C0v, C1v, ..., C7v$ 被存回矩阵 $C[m:m+7][k:k+7]$ 中。

使用 SIMD 加速矩阵乘法代码如下：

```
void MatrixMultiplyAVX512(T **A, T **B, T **C, int size) {
    // 遍历矩阵 A 和 C
    for (int m = 0; m < size; m += 8) {
        for (int k = 0; k < size; k += 8) {
            // 定义 AVX-512 向量
            __m512i C0v, C1v, C2v, C3v, C4v, C5v, C6v, C7v;
            __m512i B0v;

            // 初始化累加器
            C0v = _mm512_setzero_si512();
            C1v = _mm512_setzero_si512();
            C2v = _mm512_setzero_si512();
            C3v = _mm512_setzero_si512();
            C4v = _mm512_setzero_si512();
            C5v = _mm512_setzero_si512();
            C6v = _mm512_setzero_si512();
            C7v = _mm512_setzero_si512();

            // 对矩阵 B 和 A 进行遍历，计算乘积并累加到 C[m:m
               +7][k:k+7]
```

```
            for (int n = 0; n < size; ++n) {
                // 加载 B[n][k:k+7] 到 AVX-512 向量
                B0v = _mm512_loadu_si512((__m512i*)&B[n][k]);

                // 加载 A[m:m+7][n] 到 AVX-512 向量
                __m512i vecA0 = _mm512_set1_epi64(A[m][n]);
                __m512i vecA1 = _mm512_set1_epi64(A[m + 1][n]);
                __m512i vecA2 = _mm512_set1_epi64(A[m + 2][n]);
                __m512i vecA3 = _mm512_set1_epi64(A[m + 3][n]);
                __m512i vecA4 = _mm512_set1_epi64(A[m + 4][n]);
                __m512i vecA5 = _mm512_set1_epi64(A[m + 5][n]);
                __m512i vecA6 = _mm512_set1_epi64(A[m + 6][n]);
                __m512i vecA7 = _mm512_set1_epi64(A[m + 7][n]);

                // 逐元素乘法并累加
                C0v = _mm512_add_epi64(C0v, _mm512_mullo_epi64(vecA0, B0v));
                C1v = _mm512_add_epi64(C1v, _mm512_mullo_epi64(vecA1, B0v));
                C2v = _mm512_add_epi64(C2v, _mm512_mullo_epi64(vecA2, B0v));
                C3v = _mm512_add_epi64(C3v, _mm512_mullo_epi64(vecA3, B0v));
                C4v = _mm512_add_epi64(C4v, _mm512_mullo_epi64(vecA4, B0v));
                C5v = _mm512_add_epi64(C5v, _mm512_mullo_epi64(vecA5, B0v));
                C6v = _mm512_add_epi64(C6v, _mm512_mullo_epi64(vecA6, B0v));
                C7v = _mm512_add_epi64(C7v, _mm512_mullo_epi64(vecA7, B0v));
            }

            // 将结果存回 C[m:m+7][k:k+7]
            _mm512_storeu_si512((__m512i*)&C[m][k], C0v);
            _mm512_storeu_si512((__m512i*)&C[m + 1][k], C1v);
            _mm512_storeu_si512((__m512i*)&C[m + 2][k], C2v);
```

```
                _mm512_storeu_si512((__m512i*)&C[m + 3][k], C3v);
                _mm512_storeu_si512((__m512i*)&C[m + 4][k], C4v);
                _mm512_storeu_si512((__m512i*)&C[m + 5][k], C5v);
                _mm512_storeu_si512((__m512i*)&C[m + 6][k], C6v);
                _mm512_storeu_si512((__m512i*)&C[m + 7][k], C7v);
            }
        }
}
```

2 实验

实验平台：CPU 为 Intel(R) Xeon(R) Platinum 8352V 32 核 2.10GHz，60GB 内存，操作系统为 Ubuntu 20.04，编译器为 g++ 13.1.0，编译时开启 O3 优化。所有实验均使用上述实验平台进行测试，并且每个算法使用随机种子生成的小规模矩阵（32 阶，每个元素 32 位）和大规模矩阵（4096 阶，每个元素 64 位）测试 5 次，计算平均执行时间。报告实现的所有算法都使用 C++ 编写，在github 仓库均可找到。各个算法运行结果如表 2.1所示。在小规模矩阵乘法中，各个算法的运行时间相差无几，基本在 6 μs 左右。在大规模矩阵乘法中，各个算法之间的差距体现出来了，暴力算法和循环展开的算法运行时间都在 200s 以上，Strassen 算法凭借时间复杂度的优势，以 28s 的运行速度领先其他算法。而基于硬件的优化方式 AVX2 和 AVX512 优化效果也很显著，其中基于 AVX2 的代码执行时间为 181s，基于 AVX512 的代码执行时间为 113s，均比暴力算法和循环展开快。AVX512 比 AVX2 有更大的寄存器宽度，因此可以同时处理更多数据，进一步提高了计算效率。

	IJK 顺序	IKJ 顺序	循环展开	Strassen 算法	AVX2	AVX512
小规模矩阵	8μs	7μs	5μs	5μs	7μs	8μs
大规模矩阵	230s	227s	221s	28s	181s	113s

表 2.1 算法运行时间比较

对各个算法选取大规模矩阵进行详细的性能分析。通过 perf 工具查看算法运行的 CPU-cycle 数量，时钟周期，分支数量，cache 缺失数等信息，精确比较算法的执行信息，得到各个算法的实验结果如下表 2.1所示。传统算法（如 IJK 顺序和 IKJ 顺序）的性能相对较低，具体表现为高 CPU 时钟和 CPU 周期，以及较多的 L1 数据缓存缺失和页面错误。例如，IKJ 顺序的 CPU 周期和缓存缺失达到非常高的数值，说明这些传统算法在计算密集型操作中效率较低，主要是因为它们的内存访问模式没有得到优化，导致缓存未能有效利用。

相比之下，Strassen 算法在计算量较大时表现出了显著的性能优势。它通过采用分治法将矩阵乘法问题分解为多个较小的矩阵乘法，从而减少了乘法运算的次数。这种优化策略直接减少了 CPU 时钟和 CPU 周期，同时由于操作数减少，缓存缺失和页面错误也得到了有效降低。Strassen 算法通过减少计算步骤和优化内存访问，提高了矩阵乘法的计算效率。

此外，AVX2 和 AVX512 是基于 SIMD（单指令多数据）技术的硬件加速算法，它们通过并行处理多个数据元素来显著提升计算速度。AVX2 和 AVX512 使用 256 位和 512 位的寄存器，允许一次性加载和处理更多的数据，从而减少了每个计算步骤所需的时间。这些硬件加速算法在所有性能指标上都显著优于传统算法，尤其是在 CPU 周期和缓存缺失上，它们大大减少了 L1 缓存缺失和页面错误的发生。AVX512 更是提供了

更高的并行度和数据吞吐量，其 CPU 时钟和计算周期数量比 Strassen 算法更低，展现出了硬件加速在大规模矩阵运算中的巨大优势。

总体而言，Strassen 算法通过减少运算次数来优化计算流程，提供了性能的显著提升，而 AVX2 和 AVX512 则通过硬件级的并行计算加速了数据处理过程，进一步减少了运算时间和资源消耗。在这些优化算法中，硬件加速的 AVX512 算法无疑在所有性能指标上表现最佳，特别是在大规模矩阵运算时，硬件加速的优势更加明显。

	IJK 顺序	**IKJ 顺序**	**循环展开**	**Strassen 算法**	**AVX2**	**AVX512**
cpu-clock	63,524.4	61,423.5	60,3124.7	5,354.4	34,344.4	24,764.8
cycles	1.5×10^{12}	1.3×10^{12}	1.1×10^{12}	1.2×10^{10}	8.4×10^{11}	6.5×10^{11}
L1-dcache-load-misses	8467054	6783456	6432974	327597	5436896	4648325
page-faults	390	285	274	146	197	174

表 2.2 算法性能比较