深度学习中的高效计算方法 lab

韩金成　信息科学技术学院

2025 年 7 月 9 日

Q1. Q1

Solution.

表格太大，见下一页

表 1: 不同矩阵形状下矩阵乘法实现的平均运行时间对比

I	K	J	`matmul()`	`matmul__ikj()`	`matmul__AT()`	`matmul__BT()`
			平均运行时间 (ms)			
256	256	256	12.744	1.046	15.676	0.884
256	256	512	28.219	2.049	31.619	1.791
256	256	1024	51.268	4.126	70.614	3.633
256	512	256	26.588	2.093	31.686	1.755
256	512	512	57.991	4.064	63.521	3.550
256	512	1024	103.009	8.219	141.393	7.231
256	1024	256	54.074	4.178	65.399	3.492
256	1024	512	119.450	8.189	128.975	7.115
256	1024	1024	205.097	16.446	289.160	14.575
512	256	256	25.429	2.113	31.441	1.774
512	256	512	56.024	4.124	63.710	3.535
512	256	1024	102.119	8.245	141.032	7.104
512	512	256	52.976	4.207	63.753	3.457
512	512	512	117.128	8.220	128.335	6.999
512	512	1024	207.400	16.524	289.237	14.170
512	1024	256	109.414	8.405	129.297	6.958
512	1024	512	242.140	16.416	262.742	13.882
512	1024	1024	415.993	32.955	589.198	28.513
1024	256	256	51.943	4.264	70.329	3.571
1024	256	512	115.125	8.345	144.462	7.105
1024	256	1024	210.912	16.658	294.425	14.122
1024	512	256	106.282	6.837	140.154	6.895
1024	512	512	236.513	16.466	289.207	13.896
1024	512	1024	418.619	38.114	589.754	27.824
1024	1024	256	219.603	12.448	288.986	13.891
1024	1024	512	486.040	27.411	588.962	27.683
1024	1024	1024	841.603	65.980	1197.866	56.175

备注：实验平台:MacBook air、处理器型号:Apple M4、测试方法：运行 32 次取平均。

Q2. Q2

Solution. 以下是简单的 im2col 代码，其中：im2col 实现了核心的对输入特征图进行 im to col 操作，将输入为 $3 \times 56 \times 56$ 的特征图展开为 2916×27 的矩阵以方便与 filter 进行直接的矩阵运算.

unfold_filter 函数是将 filter 展开为能进行矩阵乘法的矩阵的辅助函数，matmul_correct 函数是将 im2col 后得到的矩阵与展开的 filter 进行矩阵乘法的辅助函数，之后 set_matrix_style 函数是将矩阵乘法后得到的临时结果转为 $64 \times 56 \times 56$ 标准卷积输出的函数。

```cpp
#include <iostream>
using namespace std;
int col[2916][27];
int unfolded_filter[27][64];
int unfolded_filter_T[64][27];
int temp_output[2916][64];
int output[64][54][54];
int (*im2col(const int X_in[3][56][56]))[27]{
    for (int i_1 = 0; i_1 < 54; i_1++){
        int bubu = i_1 * 54;
        for (int j_1 = 0; j_1 < 54; j_1++){
            for (int i = 0; i < 3; i++){
                int nn = i * 3;
                for (int j = 0; j < 3; j++){
                    int n1 = nn * 3;
                    int n2 = j * 3;
                    for (int k = 0; k < 3; k++){
                        col[bubu + j_1][n1 + n2 + k] = X_in[i][i_1 + j][j_1 +
                            k];
                    }
                }
            }
        }
    }
    return col;
}
int (*unfold_filter(const int filter[64][3][3][3]))[64]{
    for(int i = 0; i < 64; i++){
        for(int j1 = 0; j1 < 3; j1++){
            int temp1 = j1 * 3;
```

```
                for(int j2 = 0; j2 < 3; j2 ++){
                    int temp2 = j2 * 3;
                    int temp3 = temp1 * 3;
                    for(int j3 = 0; j3 < 3; j3 ++){
                        unfolded_filter[temp3 + temp2 + j3][i] =
                           filter[i][j1][j2][j3];
                    }
                }
            }
        }
        return unfolded_filter;
}
void matmul_correct(){
    memset(temp_output, 0, sizeof(temp_output));
    for (int i = 0; i < 2916; i++) {
        for (int j = 0; j < 64; j++) {
            for (int k = 0; k < 27; k++) {
                temp_output[i][j] += col[i][k] * unfolded_filter[k][j];
            }
        }
    }
}
void set_matrix_style(){
    for(int i = 0; i < 64; i++){
        for(int j = 0; j < 54; j++){
            int temp1 = j * 54;
            for(int k = 0; k < 54; k++){
                output[i][j][k] = temp_output[temp1 + k][i];
            }
        }
    }
}
```

Q3. Q3

Solution.

I	K	J	ijk	ikj	AT	BT	Unrolled	Tiled	Write Opt.	SIMD (NEON)	BT Opt.	Strassen	Tiled OpenMP	Strassen OpenMP
			平均运行时间 (ms)											
1024	1024	1024	830.739	293.897	1151.937	304.868	152.130	366.742	302.890	74.999	300.814	243.280	87.407	59.727

表 2: 不同矩阵乘法优化方法在矩阵维度 $I = 1024, K = 1024, J = 1024$ 下的平均运行时间对比

在所有性能优化的讨论开始之前，我们必须先明确一个基本的计算机工作原理：局部性。这包括两个方面，一是“空间局部性”，即如果一个数据被访问，那么它物理地址相邻的数据也很有可能在不久后被访问；二是“时间局部性”，即一个数据被访问后，它很可能在短时间内被再次访问。CPU 的高速缓存机制就是围绕这两个局部性原理建立的。我们的代码如果能遵循这些规律，就能实现很高的缓存命中率，从而避免因访问慢速主内存而导致的性能瓶颈。本次实验的目的，正是通过各种手段去增强代码的局部性或利用其他硬件特性，来为 1024x1024 的矩阵乘法提速。我们选择 ikj 循环（293.897 毫秒）作为基准，因为它本身就通过将对矩阵 C 的写操作和对 A 的读操作放在内层，较好地维持了这两个矩阵的访存连续性，相比 ijk 顺序（830.739 毫秒）是一个更合理的出发点。

在此基础上，我们首先检验了一系列软件层面的缓存优化技术。

第一项“写入优化”，其原理是利用寄存器暂存循环内的不变量 A[i][k]，以达到在内层循环中减少内存访问的目的。实验结果（302.890 毫秒）与基准无异，这通常表明编译器在 O1 优化级别下已经自动执行了此项优化（标量替换），或者 cache 大小不够，不足以容纳所有的 A。

第二项“循环展开”将耗时缩减至 152.130 毫秒，效果显著。它的原理有二：一是减少了循环判断和索引递增等控制指令的执行次数，降低了程序开销；二是增大了循环体的指令数量，为 CPU 流水线进行指令级并行调度创造了更好的条件。

第三项“数组打包”，即预先转置矩阵 B，原理是将计算中对 B 的非连续列访问，转变为对 BT 的连续行访问，以改善空间局部性。其效果（300.814 毫秒）同样不明显，可以推断现代 CPU 的硬件预取器已经部分缓解了非连续访问的性能惩罚，使得此项优化的收益与转置开销大致相抵。

第四项“分块”，原理是通过处理能完全装入缓存的子矩阵，来增强时间局部性。但实验结果（366.742 毫秒）却是负优化，推断原因是块尺寸的大小设置不当，这揭示了该技术对块尺寸参数的敏感性，以及额外的多层循环嵌套本身带来的不可忽视的开销。

接下来，我们探索了两个能带来质变的优化维度。

其一是“Strassen 算法”，这是一种算法层面的优化。它的核心原理是通过分治和巧妙的代数变换，将计算复杂度从 $O(N^3)$ 降低至约 $O(N^{2.807})$，即从根本上减少了所需的乘法运算总量。对于 1024×1024 这样的大矩阵，其性能（243.280 毫秒）超越了所有纯软件层面的缓存优化，证明了更优算法复杂度的压倒性优势。

其二是“向量化（SIMD)”，这属于硬件层面的优化。它的原理是利用 CPU 的单指令多数据流（Single Instruction, Multiple Data）单元，用一条指令同时对多个数据元素（例如 4 个整数）执行运算，实现了真正的数据级并行。其结果（74.999 毫秒）非常突出，体现了直接利用硬件并行计算能力的巨大优化能力。

最后，我们通过 OpenMP 将问题扩展到多核心处理，即线程级并行。

将并行化应用于分块算法（Tiled OpenMP)，通过数据并行的思想，让各核心分担不同数据块的计算，取得了 87.407 毫秒的成绩。

而将并行化与 Strassen 算法结合（Strassen OpenMP)，则是利用了任务并行的思想，让各核心分担递归分解后的 7 个独立子任务。最终 59.727 毫秒的成绩是全场最佳，因为它实现了两种强大优化原理的叠加：首先通过 Strassen 算法减少了宏观的运算总量，接着又通过 OpenMP 将剩余的计算任务分配给多个核心同时执行，充分说明了顶尖的性能往往来源于算法与并行化策略的深度结合。

Note. Q3 用到的代码如下：

编译环境：

编译器：

Apple clang version 17.0.0 (clang-1700.0.13.5)

Target: arm64-apple-darwin24.5.0

Thread model: posix

硬件环境：

Apple M4(10 cores)

编译命令：

```
>mac&&user clang++ matmul.cpp -o matmul_optimized -std=c++17 -O1 -Wall -DNDEBUG

-Xpreprocessor -fopenmp -L/opt/homebrew/opt/libomp/lib

-I/opt/homebrew/opt/libomp/include -lomp

(上面是一条命令拆成了 3 行)

>mac&&user ./matmul_optimized
```

```cpp
#include <sys/time.h>
#include <iostream>
#include <cstring>
#include <cassert>
#include <random>
#include <vector>
#include <algorithm>
#ifdef __ARM_NEON__
#include <arm_neon.h>
#endif
// 为了Strassen简化，这里假设I, K, J都是1024，并且为2的幂
constexpr int I = 1024;
constexpr int K = 1024;
constexpr int J = 1024;
// 缓存块大小
```

```
constexpr int BLOCK_SIZE_I = 32;
constexpr int BLOCK_SIZE_K = 32;
constexpr int BLOCK_SIZE_J = 32;
constexpr int STRASSEN_THRESHOLD = 64;
alignas(16) int A[I][K];
alignas(16) int B[K][J];
alignas(16) int BT[J][K]; // 转置B
alignas(16) int AT[K][I]; // 转置A
alignas(16) int C[I][J];
alignas(16) int C_groundtruth[I][J];
alignas(16) int S1[I/2][J/2];
double get_time() {
  struct timeval tv;
  gettimeofday(&tv, nullptr);
  return tv.tv_sec + 1e-6 * tv.tv_usec;
}
void init() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 10);
    for (int i = 0; i < I; i++) {
        for (int j = 0; j < K; j++) {
            A[i][j] = distrib(gen);
        }
    }
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < J; j++) {
            B[i][j] = distrib(gen);
        }
    }
    for (int i = 0; i < I; i++) {
        for (int j = 0; j < J; j++) {
            long long sum = 0;
            for (int k = 0; k < K; k++) {
                sum += (long long)A[i][k] * B[k][j];
            }
            C_groundtruth[i][j] = static_cast<int>(sum);
        }
    }
}
void test() {
  for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
      assert(C[i][j] == C_groundtruth[i][j]);
    }
  }
}
// 原始的ijk顺序矩阵乘法
void matmul_ijk() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
      for (int k = 0; k < K; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
// 原始的ikj顺序矩阵乘法
void matmul_ikj() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < I; i++) {
    for (int k = 0; k < K; k++) {
      for (int j = 0; j < J; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
// 原始的AT矩阵乘法
void matmul_AT() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < K; i++) {
    for (int j = 0; j < I; j++) {
      AT[i][j] = A[j][i];
    }
  }
  for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
      for (int k = 0; k < K; k++) {
        C[i][j] += AT[k][i] * B[k][j];
      }
    }
  }
}
// 原始的BT矩阵乘法
void matmul_BT() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < J; i++) {
    for (int j = 0; j < K; j++) {
      BT[i][j] = B[j][i];
    }
  }
  for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
      for (int k = 0; k < K; k++) {
        C[i][j] += A[i][k] * BT[j][k];
      }
    }
  }
}
// 循环展开 (Loop Unrolling) - 以ikj顺序为例，展开最内层循环
void matmul_ikj_unrolled() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < I; i++) {
    for (int k = 0; k < K; k++) {
      for (int j = 0; j < J; j += 4) { // J假设能被4整除
        C[i][j] += A[i][k] * B[k][j];
        C[i][j+1] += A[i][k] * B[k][j+1];
        C[i][j+2] += A[i][k] * B[k][j+2];
        C[i][j+3] += A[i][k] * B[k][j+3];
      }
    }
  }
```

```
}
// 分块/切片 (Tiling)
void matmul_tiled() {
  memset(C, 0, sizeof(C));
  for (int ii = 0; ii < I; ii += BLOCK_SIZE_I) {
    for (int jj = 0; jj < J; jj += BLOCK_SIZE_J) {
      for (int kk = 0; kk < K; kk += BLOCK_SIZE_K) {
        for (int i = ii; i < std::min(ii + BLOCK_SIZE_I, I); i++) {
          for (int j = jj; j < std::min(jj + BLOCK_SIZE_J, J); j++) {
            for (int k = kk; k < std::min(kk + BLOCK_SIZE_K, K); k++) {
              C[i][j] += A[i][k] * B[k][j];
            }
          }
        }
      }
    }
  }
}
// 写入缓存优化 (Writing Caching)
void matmul_ikj_write_optimized() {
    memset(C, 0, sizeof(C));
    for (int i = 0; i < I; ++i) {
        for (int k = 0; k < K; ++k) {
            int temp_A_ik = A[i][k];
            for (int j = 0; j < J; ++j) {
                C[i][j] += temp_A_ik * B[k][j];
            }
        }
    }
}
// 向量化 (Vectorization (SIMD)) - 使用 ARM NEON Intrinsics
void matmul_simd() {
    memset(C, 0, sizeof(C));
    constexpr int SIMD_WIDTH = 4; // 128位NEON向量包含4个32位整数

    for (int i = 0; i < I; ++i) {
        for (int k = 0; k < K; ++k) {
            int32x4_t a_val = vdupq_n_s32(A[i][k]);
            for (int j = 0; j < J; j += SIMD_WIDTH) {
                int32x4_t c_vec = vld1q_s32(C[i] + j);
                int32x4_t b_vec = vld1q_s32(B[k] + j);
                int32x4_t prod_vec = vmulq_s32(a_val, b_vec);
                c_vec = vaddq_s32(c_vec, prod_vec);
                vst1q_s32(C[i] + j, c_vec);
            }
        }
    }
}
// 数组打包 (Array packing) - 通过转置BT来优化B的访问模式
void matmul_BT_optimized() {
  memset(C, 0, sizeof(C));
  for (int i = 0; i < J; i++) {
    for (int j = 0; j < K; j++) {
      BT[i][j] = B[j][i];
    }
  }
  for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
      for (int k = 0; k < K; k++) {
        C[i][j] += A[i][k] * BT[j][k];
      }
    }
  }
}
// 加法 C = A + B
void matrix_add(int* A_ptr, int* B_ptr, int* C_ptr, int dim) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            C_ptr[i * dim + j] = A_ptr[i * dim + j] + B_ptr[i * dim + j];
        }
    }
}
// 减法 C = A - B
void matrix_sub(int* A_ptr, int* B_ptr, int* C_ptr, int dim) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
```

```cpp
                C_ptr[i * dim + j] = A_ptr[i * dim + j] - B_ptr[i * dim + j];
            }
        }
}
// C = A * B
void _matmul_ijk_base(const int* A_ptr, const int* B_ptr, int* C_ptr, int dim)
    {
     for (int i = 0; i < dim; ++i) {
         for (int j = 0; j < dim; ++j) {
             C_ptr[i * dim + j] = 0; // 初始化
             for (int k = 0; k < dim; ++k) {
                 C_ptr[i * dim + j] += A_ptr[i * dim + k] * B_ptr[k * dim + j];
             }
         }
     }
}
void _strassen_matmul_recursive(int* A_start, int a_stride,
                                int* B_start, int b_stride,
                                int* C_start, int c_stride,
                                int current_dim) {
    // 递归终止条件
    if (current_dim <= STRASSEN_THRESHOLD) {

        std::vector<int> tempA(current_dim * current_dim);
        std::vector<int> tempB(current_dim * current_dim);
        std::vector<int> tempC(current_dim * current_dim);

        for(int i = 0; i < current_dim; ++i) {
            for(int j = 0; j < current_dim; ++j) {
                tempA[i * current_dim + j] = A_start[i * a_stride + j];
                tempB[i * current_dim + j] = B_start[i * b_stride + j];
            }
        }
        _matmul_ijk_base(tempA.data(), tempB.data(), tempC.data(),
            current_dim);
        for(int i = 0; i < current_dim; ++i) {
            for(int j = 0; j < current_dim; ++j) {
                C_start[i * c_stride + j] = tempC[i * current_dim + j];
            }
        }
        return;
    }
    int half_dim = current_dim / 2;
    int* A11 = A_start;
    int* A12 = A_start + half_dim;
    int* A21 = A_start + half_dim * a_stride;
    int* A22 = A_start + half_dim * a_stride + half_dim;
    int* B11 = B_start;
    int* B12 = B_start + half_dim;
    int* B21 = B_start + half_dim * b_stride;
    int* B22 = B_start + half_dim * b_stride + half_dim;
    int* C11 = C_start;
    int* C12 = C_start + half_dim;
    int* C21 = C_start + half_dim * c_stride;
    int* C22 = C_start + half_dim * c_stride + half_dim;
    alignas(16) static int M1[I/2][J/2], M2[I/2][J/2], M3[I/2][J/2], M4[I/2][J/2],
                           M5[I/2][J/2], M6[I/2][J/2], M7[I/2][J/2];
    alignas(16) static int T1[I/2][J/2], T2[I/2][J/2]; // 临时矩阵用于加减
    // 计算 M1 - M7
    // M1 = (A11 + A22) * (B11 + B22)
    matrix_add((int*)A11, (int*)A22, (int*)T1, half_dim); // T1 = A11 + A22
    matrix_add((int*)B11, (int*)B22, (int*)T2, half_dim); // T2 = B11 + B22
    _strassen_matmul_recursive((int*)T1, half_dim, (int*)T2, half_dim, (int*)M1, half_dim, half_dim);
    // M2 = (A21 + A22) * B11
    matrix_add((int*)A21, (int*)A22, (int*)T1, half_dim); // T1 = A21 + A22
    _strassen_matmul_recursive((int*)T1, half_dim, (int*)B11, b_stride, (int*)M2, half_dim, half_dim);
    // M3 = A11 * (B12 - B22)
    matrix_sub((int*)B12, (int*)B22, (int*)T1, half_dim); // T1 = B12 - B22
    _strassen_matmul_recursive((int*)A11, a_stride, (int*)T1, half_dim, (int*)M3, half_dim, half_dim);
    // M4 = A22 * (B21 - B11)
    matrix_sub((int*)B21, (int*)B11, (int*)T1, half_dim); // T1 = B21 - B11
    _strassen_matmul_recursive((int*)A22, a_stride, (int*)T1, half_dim,
            (int*)M4, half_dim, half_dim);
    // M5 = (A11 + A12) * B22
    matrix_add((int*)A11, (int*)A12, (int*)T1, half_dim); // T1 = A11 + A12
    _strassen_matmul_recursive((int*)T1, half_dim, (int*)B22, b_stride,
            (int*)M5, half_dim, half_dim);
    // M6 = (A21 - A11) * (B11 + B12)
    matrix_sub((int*)A21, (int*)A11, (int*)T1, half_dim); // T1 = A21 - A11
    matrix_add((int*)B11, (int*)B12, (int*)T2, half_dim); // T2 = B11 + B12
    _strassen_matmul_recursive((int*)T1, half_dim, (int*)T2, half_dim,
            (int*)M6, half_dim, half_dim);
    // M7 = (A12 - A22) * (B21 + B22)
    matrix_sub((int*)A12, (int*)A22, (int*)T1, half_dim); // T1 = A12 - A22
    matrix_add((int*)B21, (int*)B22, (int*)T2, half_dim); // T2 = B21 + B22
    _strassen_matmul_recursive((int*)T1, half_dim, (int*)T2, half_dim,
            (int*)M7, half_dim, half_dim);
    // 计算 C11, C12, C21, C22
    // C11 = M1 + M4 - M5 + M7
    matrix_add((int*)M1, (int*)M4, (int*)T1, half_dim); // T1 = M1 + M4
    matrix_sub((int*)T1, (int*)M5, (int*)T2, half_dim); // T2 = T1 - M5
    matrix_add((int*)T2, (int*)M7, (int*)C11, half_dim); // C11 = T2 + M7
    // C12 = M3 + M5
    matrix_add((int*)M3, (int*)M5, (int*)C12, half_dim);
    // C21 = M2 + M4
    matrix_add((int*)M2, (int*)M4, (int*)C21, half_dim);
    // C22 = M1 - M2 + M3 + M6
    matrix_sub((int*)M1, (int*)M2, (int*)T1, half_dim); // T1 = M1 - M2
    matrix_add((int*)T1, (int*)M3, (int*)T2, half_dim); // T2 = T1 + M3
    matrix_add((int*)T2, (int*)M6, (int*)C22, half_dim); // C22 = T2 + M6
}
// Strassen矩阵乘法的公共接口
void matmul_strassen() {
    memset(C, 0, sizeof(C));
    assert(I == K && K == J && (I & (I - 1)) == 0);
    _strassen_matmul_recursive((int*)A, K, (int*)B, J, (int*)C, J, I);
}
// 多线程分块矩阵乘法
void matmul_tiled_openmp() {
  memset(C, 0, sizeof(C));
```

```
 for (int ii = 0; ii < I; ii += BLOCK_SIZE_I) {
  for (int jj = 0; jj < J; jj += BLOCK_SIZE_J) {
   for (int kk = 0; kk < K; kk += BLOCK_SIZE_K) {
    for (int i = ii; i < std::min(ii + BLOCK_SIZE_I, I); i++) {
     for (int j = jj; j < std::min(jj + BLOCK_SIZE_J, J); j++) {
      for (int k = kk; k < std::min(kk + BLOCK_SIZE_K, K); k++) {
       C[i][j] += A[i][k] * B[k][j];
      }
     }
    }
   }
  }
 }
}
void matmul_strassen_openmp() {
  memset(C, 0, sizeof(C));
  assert(I == K && K == J && (I & (I - 1)) == 0); // 检查是否为2的幂且方阵
  int half_dim = I / 2;
  int* A11 = (int*)A;
  int* A12 = (int*)A + half_dim;
  int* A21 = (int*)A + half_dim * K;
  int* A22 = (int*)A + half_dim * K + half_dim;
  int* B11 = (int*)B;
  int* B12 = (int*)B + half_dim;
  int* B21 = (int*)B + half_dim * J;
  int* B22 = (int*)B + half_dim * J + half_dim;
  int* C11 = (int*)C;
  int* C12 = (int*)C + half_dim;
  int* C21 = (int*)C + half_dim * J;
  int* C22 = (int*)C + half_dim * J + half_dim;
  alignas(16) static int M1_par[I/2][J/2], M2_par[I/2][J/2],
     M3_par[I/2][J/2], M4_par[I/2][J/2],
             M5_par[I/2][J/2], M6_par[I/2][J/2],
               M7_par[I/2][J/2];
  alignas(16) static int T1_par[I/2][J/2], T2_par[I/2][J/2];
  // 计算 C11, C12, C21, C22
  // C11 = M1 + M4 - M5 + M7
  matrix_add((int*)M1_par, (int*)M4_par, (int*)T1_par, half_dim);
```

```
    matrix_sub((int*)T1_par, (int*)M5_par, (int*)T2_par, half_dim);
    matrix_add((int*)T2_par, (int*)M7_par, (int*)C11, half_dim);
    // C12 = M3 + M5
    matrix_add((int*)M3_par, (int*)M5_par, (int*)C12, half_dim);
    // C21 = M2 + M4
    matrix_add((int*)M2_par, (int*)M4_par, (int*)C21, half_dim);
    // C22 = M1 - M2 + M3 + M6
    matrix_sub((int*)M1_par, (int*)M2_par, (int*)T1_par, half_dim);
    matrix_add((int*)T1_par, (int*)M3_par, (int*)T2_par, half_dim);
    matrix_add((int*)T2_par, (int*)M6_par, (int*)C22, half_dim);
}
int main() {
  init();
  constexpr int RUN_TIMES = 3;
  double total_time_ijk = 0.0f;
  double total_time_ikj = 0.0f;
  double total_time_AT = 0.0f;
  double total_time_BT = 0.0f;
  double total_time_unrolled = 0.0f;
  double total_time_tiled = 0.0f;
  double total_time_write_optimized = 0.0f;
  double total_time_simd = 0.0f;
  double total_time_BT_optimized = 0.0f;
  double total_time_strassen = 0.0f;
  double total_time_tiled_openmp = 0.0f;
  double total_time_strassen_openmp = 0.0f;
  printf("Running␣%d␣times␣for␣averaging...\n", RUN_TIMES);
  for (int run = 0; run < RUN_TIMES; ++run) {
    auto t = get_time();
    matmul_ijk();
    total_time_ijk += (get_time() - t);
    test();
    t = get_time();
    matmul_ikj();
    total_time_ikj += (get_time() - t);
    test();
    t = get_time();
    matmul_AT();
    total_time_AT += (get_time() - t);
    test();
    t = get_time();
    matmul_BT();
    total_time_BT += (get_time() - t);
    test();
    t = get_time();
    matmul_ikj_unrolled();
    total_time_unrolled += (get_time() - t);
    test();
    t = get_time();
    matmul_tiled();
    total_time_tiled += (get_time() - t);
    test();
    t = get_time();
    matmul_ikj_write_optimized();
    total_time_write_optimized += (get_time() - t);
    test();
    t = get_time();
    matmul_simd();
    total_time_simd += (get_time() - t);
    test();
    t = get_time();
    matmul_BT_optimized();
    total_time_BT_optimized += (get_time() - t);
    test();
    t = get_time();
    matmul_strassen();
    total_time_strassen += (get_time() - t);
    test();
    t = get_time();
    matmul_tiled_openmp();
    total_time_tiled_openmp += (get_time() - t);
    test();
    t = get_time();
    matmul_strassen_openmp();
    total_time_strassen_openmp += (get_time() - t);
    test();
```

```
  }
  printf("Average␣times␣over␣%d␣runs:\n", RUN_TIMES);
  printf("Original␣ijk␣matmul␣time:␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_ijk /
     RUN_TIMES * 1000);
  printf("Original␣ikj␣matmul␣time:␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_ikj /
     RUN_TIMES * 1000);
  printf("Original␣AT␣matmul␣time:␣␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_AT /
     RUN_TIMES * 1000);
  printf("Original␣BT␣matmul␣time:␣␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_BT /
     RUN_TIMES * 1000);
  printf("IKJ␣Loop␣Unrolled␣matmul␣time:␣␣␣␣%f␣ms\n", total_time_unrolled /
     RUN_TIMES * 1000);
  printf("Tiled␣matmul␣time:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_tiled /
     RUN_TIMES * 1000);
  printf("IKJ␣Write␣Optimized␣matmul␣time:␣␣%f␣ms\n",
     total_time_write_optimized / RUN_TIMES * 1000);
  printf("SIMD␣matmul␣time␣(NEON):␣␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_simd /
     RUN_TIMES * 1000);
  printf("BT␣Optimized␣matmul␣time␣(Array␣Packing):␣%f␣ms\n",
     total_time_BT_optimized / RUN_TIMES * 1000);
  printf("Strassen␣matmul␣time:␣␣␣␣␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_strassen /
     RUN_TIMES * 1000);
  printf("Tiled␣OpenMP␣matmul␣time:␣␣␣␣␣␣␣␣␣%f␣ms\n", total_time_tiled_openmp
     / RUN_TIMES * 1000);
  printf("Strassen␣OpenMP␣matmul␣time:␣␣␣␣␣␣%f␣ms\n",
     total_time_strassen_openmp / RUN_TIMES * 1000);
  return 0;
}
```