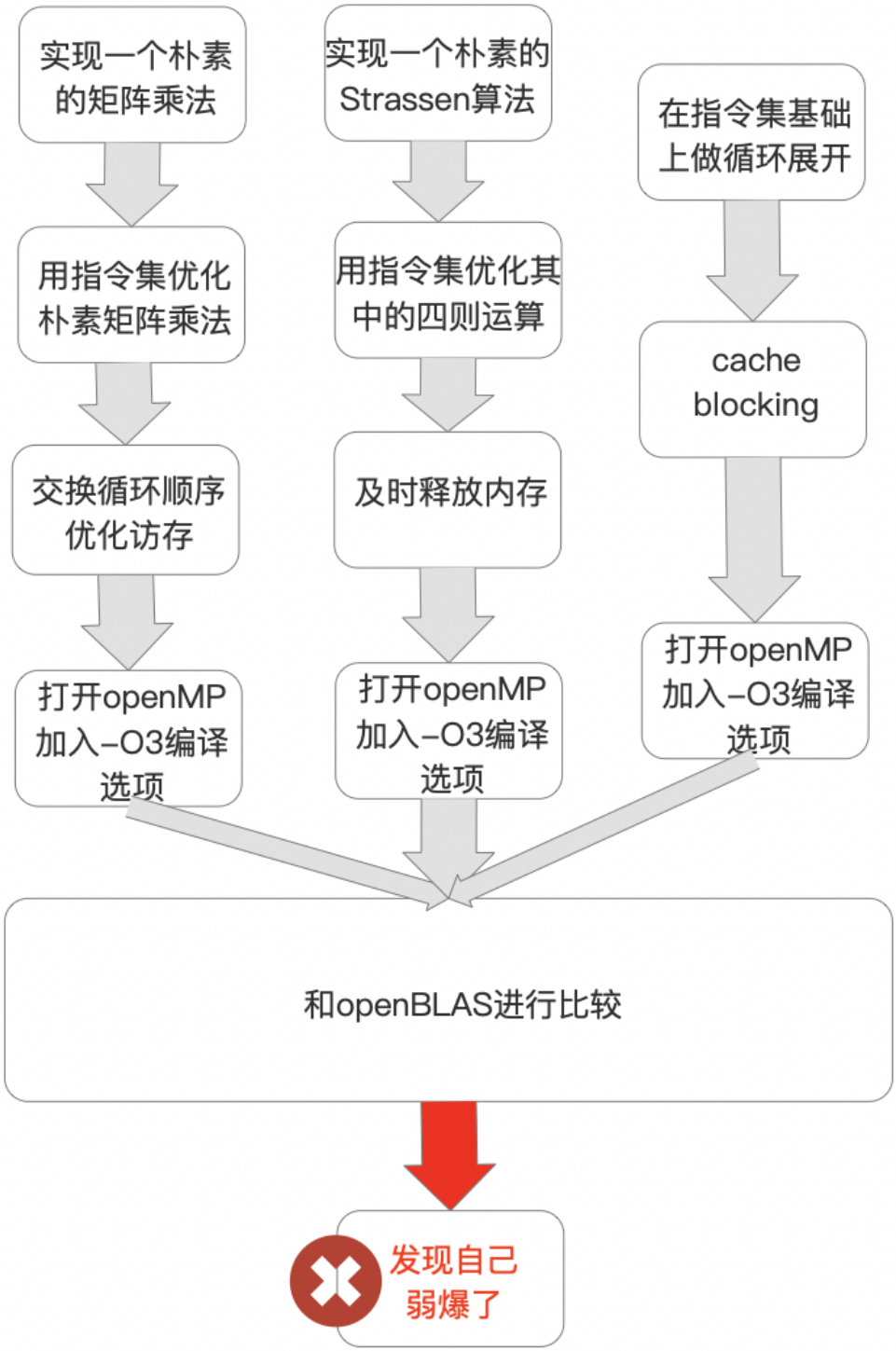
Project 4: Matrix Multiplication in C

Name: 周思呈
SID: 12110644

Part 01 - Analysis

工作主要内容梳理

下面这个流程图展示了我在本次project中完成的大部分工作，一共尝试了三种优化思路。我愿称之为“殊途同归”。



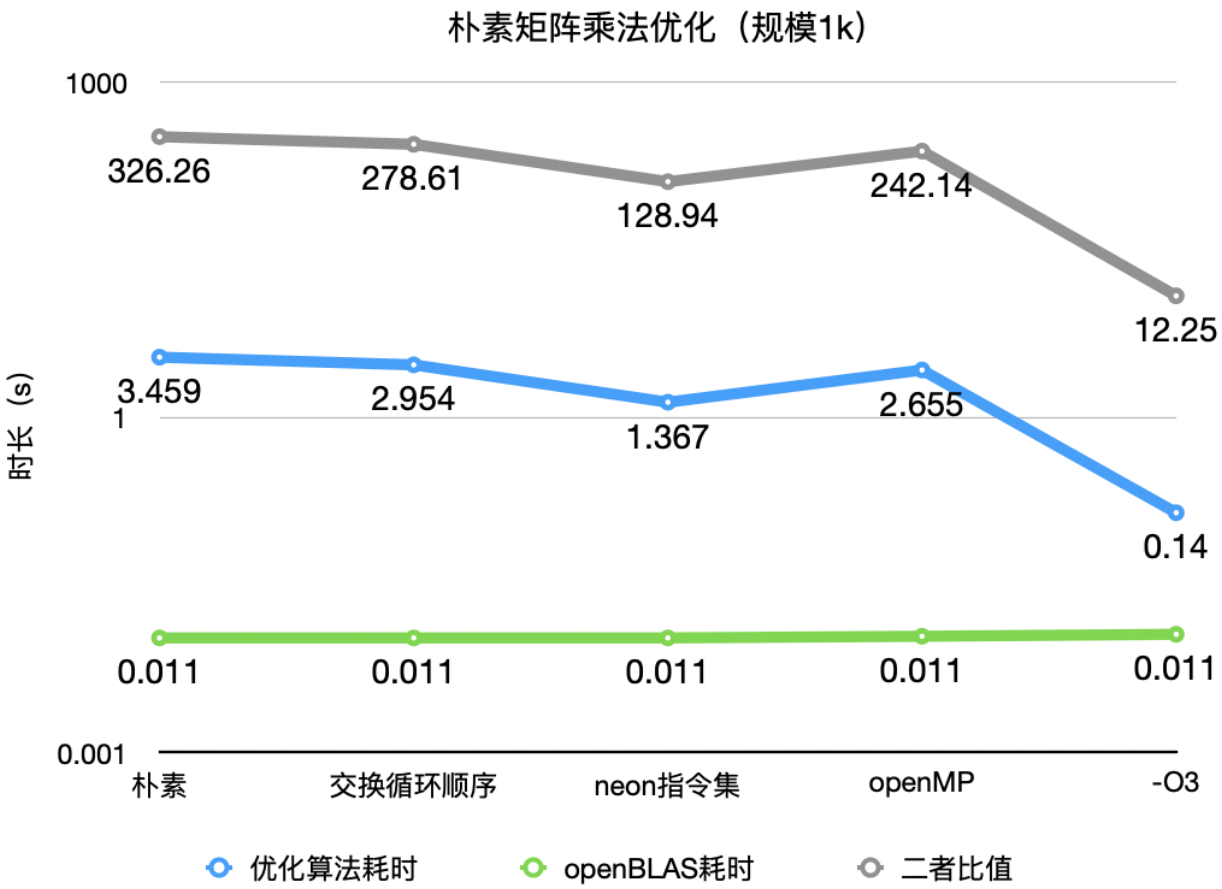
工作主要结果梳理

下面几张图表简要总结了优化所得的成果。在测试过程中发现，openBLAS在矩阵规模确定时运行情况较为稳定，因此将其作为优化结果的参考，以二者耗时比值大小判断优化成果好坏，比值越小优化效果越好，以消除测试过程中因为cpu计算频率波动等其他因素带来的影响。

可以看到，最开始朴素算法的计算时间是openBLAS计算时间的326.26倍，在朴素基础上优化得到的最好效果则缩短到了openBLAS计算时间的12.25倍，计算速度提升了24.71倍。

朴素矩阵乘法优化（规模1k）

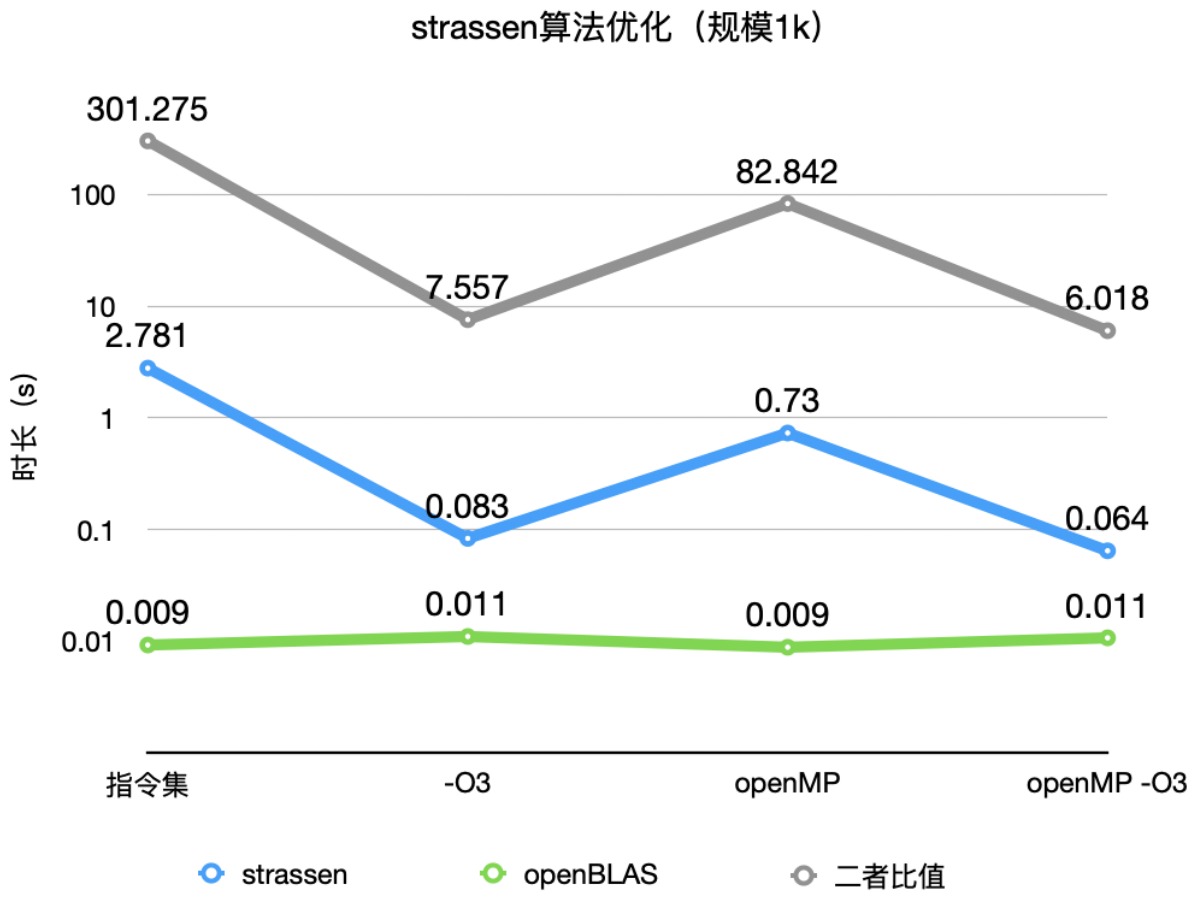
	优化算法耗时	openBLAS耗时	二者比值
朴素	3.45897	0.01060	326.26
交换循环顺序	2.95385	0.01060	278.61
neon指令集	1.36704	0.01060	128.94
openMP	2.65507	0.01097	242.14
-O3	0.13997	0.01142	12.25



最开始strassen算法的计算时间是openBLAS计算时间的301.27倍，在strassen算法基础上优化得到的最好效果则缩短到了openBLAS计算时间的6.01倍，计算速度提升了50.04倍。

strassen算法优化（规模1k）

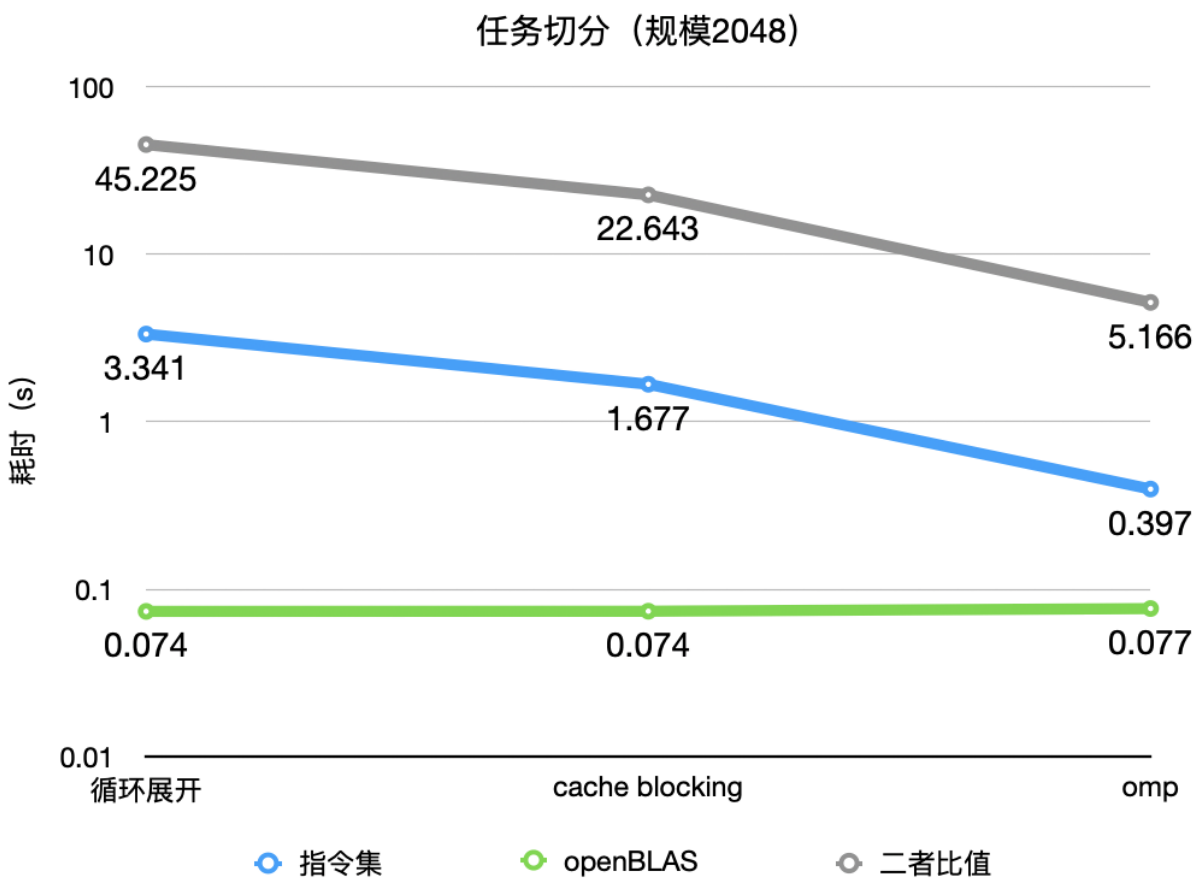
1024	strassen	openBLAS	二者比值
指令集	2.780767	0.009230	301.27486
-O3	0.083138	0.011001	7.55731
openMP	0.730085	0.008813	82.84182
openMP -O3	0.064373	0.010696	6.01842



发现strassen算法优化程度有限之后重新对于朴素算法进行任务切分等优化。最开始朴素算法的计算时间是openBLAS计算时间的326.26倍，经过任务切分、cache命中等优化之后的最好效果则缩短到了openBLAS计算时间的5.16倍，计算速度提升了63.23倍。在8k规模的大矩阵计算中，这种优化方式甚至跑出了仅为openBLAS耗时1.748倍的良好效果。

任务切分（规模2048）

2048	指令集	openBLAS	二者比值
循环展开	3.341438	0.073884	45.22546
cache blocking	1.676857	0.074055	22.64340
omp	0.396893	0.076831	5.16579
8k	14.244533	8.149054	1.74800



但是总体来说，无论哪种优化方式都没能达到openBLAS的计算速度。去查了一些资料看openBLAS的实现方式，了解到其实现分为三个层次，分别是 Interface, driver, kernel 层，Interface 层指定要执行的分支；Driver 层具体实现，任务切分，数据重排，以达到多核并行以及高效利用 cache 来提升 CPU 利用率的目的；到kernel 层才进行核心运算，此时数据已经都在 cache 上，通过使用一些汇编或者是 intrinsic 指令来高效利用 CPU 的计算能力。[1] 不得不说开源库人多力量大还是很厉害的。

Part 02 - Code

这一部分阐述了测试方法、三种优化思路中涉及到各种优化方法的实现思路和具体代码。

project3订正

去b站上看了于老师对于project3实现方法的分析和讲解，深感自己project3写的不好，于是在project4中相似部分进行“订正”。

`deleteMatrix` 方法在释放内存前先检查传入的指针是否为空，如果为空则在标准错误输出打印错误信息，返回false，告诉用户释放失败。

```
bool deleteMatrix(Matrix *matrix)
{
    //但是无法判断地址是否有效
    if (!matrix) {
        fprintf(stderr, "the pointer matrix is NULL!\n");
        return false;
    }
    if (matrix->value) {
        free(matrix->value);
        matrix->value = NULL;
    }
    free(matrix);
    return true;
}
```

`createMatrix` 方法用于创建有初始值矩阵。创建之前先对输入参数行数、列数进行检查，若不合法则打印错误信息并返回空指针。接着尝试申请结构体内存，申请之后进行检验（因为如果其他地方造成内存泄漏可能申请失败），若申请失败则输出错误信息返回空指针，同理申请结构体中数组内存。但要注意如果数组内存申请不成功则需要释放掉前面申请的结构体内存再返回空指针。申请成功后对矩阵值进行随机初始化。最后返回申请完毕的结构体指针。

```
Matrix *createMatrix(const size_t row, const size_t col){
    if (row == 0 || col == 0){
        fprintf(stderr, "rows and/or cols is 0.\n");
        return NULL;
    }
    Matrix *matrix = (Matrix *) malloc(sizeof(Matrix));
    if (matrix == NULL){
        fprintf(stderr, "fail to allocate memory!\n");
        return NULL;
    }
    matrix->col = col;
    matrix->row = row;
    matrix->value = (float *) aligned_alloc(128, row * col * sizeof(float));
    if (matrix->value == NULL){
        fprintf(stderr, "failed to allocate memory for value!\n");
        free(matrix);
        return NULL;
    }
    memset(matrix->value, 1, col * row * sizeof(float));
        matrix->value[7] = 1.1;
    return matrix;
}
```

`matmul_plain` 朴素矩阵乘法，只进行计算，不进行内存管理。计算前进行输入检验，若果有错误返回false表示计算不成功。

```
bool matmul_plain(const Matrix *matrixLeft,
                                const Matrix *matrixRight, Matrix * result)
{
    if (!(matrixLeft && matrixRight && result
        && matrixLeft->value && matrixRight->value && result->value)){
        fprintf(stderr, "pointer NULL!\n");
        return 0;
    }
    if (matrixLeft->col != matrixRight->row)
    {
        fprintf(stderr,
        "two matrices can not be multiplied due to illegal size!\n");
```

```
        return 0;
    }
    else
    {
        size_t row = matrixLeft->row;
        size_t col = matrixRight->col;
        size_t mul = matrixLeft->col;
        memset(result->value, 0, row * col * sizeof(float));
#pragma omp parallel for
        for (size_t i = 0; i < row; i++)
        {
            for (size_t k = 0; k < mul; k++)
            {
                for (size_t j = 0; j < col; j++)
                {
                    result->value[i * col + j] +=
                            matrixLeft->value[i * mul + k] *
                            matrixRight->value[k * col + j];
                }
            }
        }
        return 1;
    }
}
```

正确性检验

将优化所得计算结果与plain计算结果进行比较，检验优化之后正确性。由于本次测试中所用数据类型为float，数据精度较低，在多次四则运算后可能出现误差，因此采用误差百分比来判断误差是否在可接受范围内。幸运的是，在后续测试过程中，优化算法大部分能保证计算结果正确性，也就是打印“the biggest percentage of mistake is 0.000000 %”。

```
float isCorrect(const Matrix *correctMatrix, const Matrix *resultMatrix)
{
    if (!correctMatrix || !resultMatrix){
        fprintf(stderr, "matrix pointer is/are NULL!\n");
        return -1;
    }
    if (!correctMatrix->value || !resultMatrix->value){
        fprintf(stderr, "value pointer is/are NULL!\n");
        return -1;
    }
    if (correctMatrix->col != resultMatrix->col ||
            correctMatrix->row != resultMatrix->col){
        fprintf(stderr, "two matrices have different sizes!\n");
        return -1;
    }
    size_t row = correctMatrix->row;
    size_t col = correctMatrix->col;
    size_t size = row * col;
    float max = 0;
    float original = 0;
    float * p1 = correctMatrix->value;
    float * p2 = resultMatrix->value;
    for (size_t i = 0; i < size; i++){
        float result = *(p1++) - *(p2++);
        if (result > max){
            max = result;
            original = *(p1-1);
        }else if (-result > max){
            max = -result;
            original = *(p1-1);
        }
    }
    if (original == 0)
```

```
    {
        return 0;
    }
    else
    {
        printf("the correct result is %f while the biggest difference is %f\n",
         original, max);
        return max*(float)100/original;
    }
}
```

测试主函数

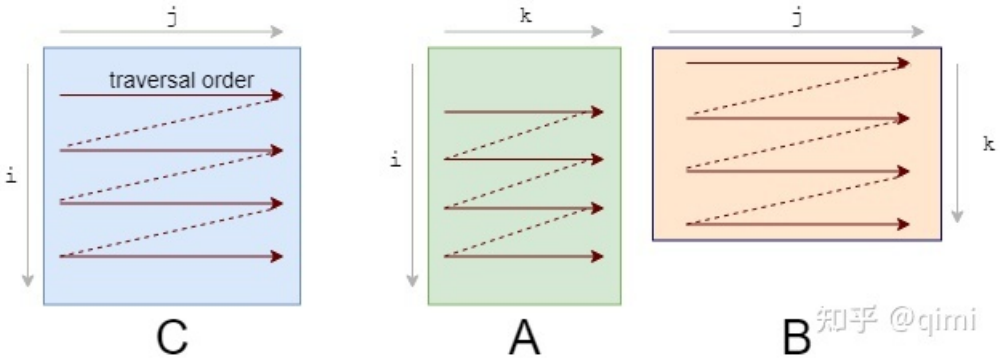
正式测试前先采用两次规模为1k的矩阵乘法进行热身。计时采用`gettimeofday`函数。进行计算的函数只负责计算，不负责内存管理，所以计算之前需要`createPlainMatrix`分类内存，计算之后需要`deleteMatrix`释放内存。

```
#include "matrix.h"
#include <stdio.h>
#include <sys/time.h>

int main()
{
    struct timeval start,end;
    int size[5] = {16, 128, 1000, 8000, 64000};
    Matrix *matrixx = createMatrix(1000, 1000);
    //热身
    Matrix *result = createPlainMatrix(1000, 1000);
    printf("warm up\n");
    if (matmul_plain(matrixx, matrixx, result)){
        printf("warm up 2 done\n");
    }
    deleteMatrix(result);
    result = NULL;
    result = createPlainMatrix(1000, 1000);
    if (matmul_plain(matrixx, matrixx, result)){
        printf("warm up 1 done\n");
    }
    deleteMatrix(result);
    result = NULL;
    deleteMatrix(matrixx);
    matrixx = NULL;

    for (int i = 0; i < 5; ++i) {
        int tempSize = size[i];
        printf("-----------size = %d ---------------\n", tempSize);
        matrixx = createMatrix(tempSize, tempSize);

        printf("this is the plain result\n");
        result = createPlainMatrix(tempSize, tempSize);
        gettimeofday(&start, NULL);
        matmul_plain(matrixx, matrixx, result);
        gettimeofday(&end, NULL);
        long timeuse = 1000000 * (end.tv_sec - start.tv_sec) +
                end.tv_usec - start.tv_usec;
        printf("time=%f\n", timeuse / 1000000.0);

        printf("this is the advanced result\n");
        Matrix *advanced = createPlainMatrix(tempSize, tempSize);
        gettimeofday(&start, NULL);
        neon_unroll(matrixx, matrixx, advanced);
        gettimeofday(&end, NULL);
        timeuse = 1000000 * (end.tv_sec - start.tv_sec) +
                end.tv_usec - start.tv_usec;
        float mistake = isCorrect(result, advanced);
```

```
        printf("time=%f\n", timeuse / 1000000.0);
        printf("the biggest percentage of mistake is %f %%\n", mistake);
        deleteMatrix(advanced);

        printf("this is the opebBLAS result\n");
        advanced = createPlainMatrix(tempSize, tempSize);
        gettimeofday(&start, NULL);
        cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                matrixx->col, matrixx->col, matrixx->col, 1,
            matrixx->value, matrixx->col, matrixx->value,
            matrixx->col, 0, advanced->value, matrixx->col);
        gettimeofday(&end, NULL);
        timeuse = 1000000 * (end.tv_sec - start.tv_sec) +
                end.tv_usec - start.tv_usec;
        mistake = isCorrect(result, advanced);
        printf("time=%f\n", timeuse / 1000000.0);
        printf("the biggest percentage of mistake is %f %%\n", mistake);
        deleteMatrix(advanced);

        printf("this is the strassen result\n");
        gettimeofday(&start, NULL);
        advanced = Strassen(0, matrixx, 0, matrixx, matrixx->col);
        gettimeofday(&end, NULL);
        timeuse = 1000000 * (end.tv_sec - start.tv_sec) +
                end.tv_usec - start.tv_usec;
        mistake = isCorrect(result, advanced);
        printf("time=%f\n", timeuse / 1000000.0);
        printf("the biggest percentage of mistake is %f %%\n", mistake);

        deleteMatrix(advanced);
        deleteMatrix(matrixx);
    }

    return 0;
}
```

ikj访存优化

原本的矩阵乘法外层循环遍历A的行和B的列，最内层遍历A的列和B的行，那么在访问B中元素时就会出现不连续访问的情况，这会导致cache命中率不高，使得数据读写速度变慢。但如果交换j和k的顺序就能巧妙地使B中元素访问连续，提高运算效率。[2]这部分的代码已经展示在了前文 project_4_matrixInC/report/report > project3订正 中，此处不再重复粘贴。



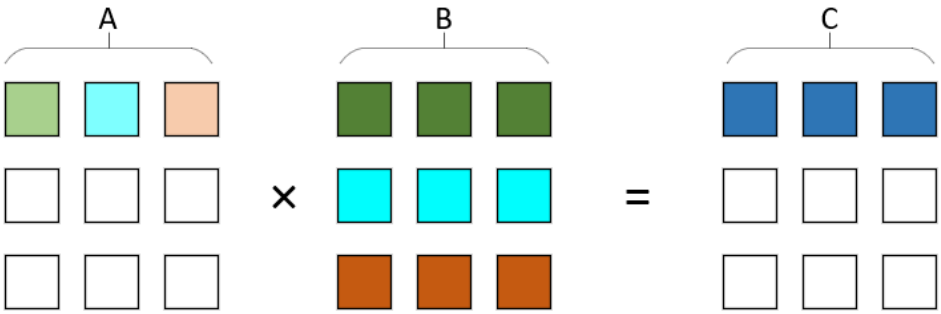
SIMD指令集

在访存优化的基础上应用SIMD指令集加快计算速度。
SIMD（Single Instruction Multiple Data，单指令多数据流），使用一个控制器控制多个处理单元，同时对一组数据中的每一个数据执行相同的操作。[3]以ARM架构下的NEON指令集为例，主要实现方法为，定义三个能储存四个32位浮点数的寄存器`float32x4_t`，取出A[i][j]元素并复制四份存在va里，取出B[j][k]以及该地址往后数的四个元素存在vb中，将

这二者相乘后加到结果矩阵C[i][k]以及该地址往后数的四个元素vc中。x86架构下的AVX指令集对应寄存器为包含8个float类型的`__m256`，每次同时对8个元素进行操作。

以下是使用到的部分NEON指令集中函数说明：

```
float32x4_t res = vdupq_n_f32(0.0f); // 存储的四个float32都初始化为0

float32x4_t res1 = vmlaq_f32(q0, q1, q2); // q0 + q1*q2

float32x4_t q0 = vld1q_f32(d0); // 加载 d0 地址起始的 4 个 float 数据到 q0
```



$$C[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0] + A[0][2] * B[2][0]$$

$$C[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1] + A[0][2] * B[2][1]$$

$$C[0][2] = A[0][0] * B[0][2] + A[0][1] * B[1][2] + A[0][2] * B[2][2]$$

$$\begin{pmatrix} C[0][0] \\ C[0][1] \\ C[0][2] \end{pmatrix} = A[0][0]* \begin{pmatrix} B[0][0] \\ B[0][1] \\ B[0][2] \end{pmatrix} + A[0][1]* \begin{pmatrix} B[1][0] \\ B[1][1] \\ B[1][2] \end{pmatrix} + A[0][2]* \begin{pmatrix} B[2][0] \\ B[2][1] \\ B[2][2] \end{pmatrix}$$

```
bool matmul_improved(const Matrix *matrixLeft,
                                        const Matrix *matrixRight, Matrix * result)
{
    if (!(matrixLeft && matrixRight && result
          && matrixLeft->value && matrixRight->value && result->value)){
        fprintf(stderr, "pointer NULL!\n");
        return 0;
    }
    if (matrixLeft->col != matrixRight->row)
    {
        fprintf(stderr,
                "two matrices can not be multiplied due to illegal size!\n");
        return 0;
    }
    else
    {
        size_t M = matrixLeft->row;
        size_t N = matrixRight->col;
        size_t K = matrixLeft->col;
        memset(result->value, 0, M * N * sizeof(float));
        float t = 0;
#ifdef WITH_NEON
        float32x4_t va, vb, vc;
#pragma omp parallel for
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                float t = *(matrixLeft->value + M * i + j);
                for (int k = 0; k < K; k+=4) {
                    vb = vld1q_f32(matrixRight->value + N * j + k); //B[j][k]
```

```
                    vc = vld1q_f32(result->value + N * i + k); //C[i][k]
                    vc = vmlaq_n_f32(vc, vb, t);
                    vst1q_f32(result->value + N * i + k, vc);
                }
            }
        }
        return 1;
#elifdef WITH_AVX2
        __m256 vecA, vecB, vecC;
#pragma omp parallel for
        for (int i = 0; i < matrixRight->col; i++) {
            for (int j = 0; j < matrixLeft->row; j++) {
                vecA = _mm256_set1_ps(
                        *(matrixLeft->value + matrixLeft->col * i + j)); //A[i][j]
                for (int k = 0; k < matrixLeft->col; k+=8) {
                    vecB = _mm256_loadu_ps(
                            matrixRight->value + matrixRight->col * j + k); //B[j][k]
                    vecC = _mm256_loadu_ps(
                            result->value + result->col * i + k); //C[i][k]
                    vecC = _mm256_fmadd_ps(va, vb, vc);
                    _mm256_storeu_ps(
                            result->value + result->col * i + k, vecC);
                }
            }
        }
        return 1;
#else
        printf( "NEON and AVX are both not supported\n" );
        return 0;
#endif
    }
}
```

Strassen算法

发现访存和指令集对计算速度提升较有限之后，转而试图利用strassen算法降低计算复杂度以提升计算效率。

公式

“Strassen 算法的核心思想是令递归树稍微不那么茂盛一点儿，即只递归进行7次而不是8次1/2×n/2矩阵的乘法。减少一次矩阵乘法带来的代价可能是额外几次1/2×2/2 矩阵的加法，但只是常数次。”[4] 主要实现公式如下。

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

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \\ S_4 &= B_{21} - B_{11} \\ S_5 &= A_{11} + A_{22} \\ S_6 &= B_{11} + B_{22} \\ S_7 &= A_{12} - A_{22} \\ S_8 &= B_{21} + B_{22} \\ S_9 &= A_{11} - A_{21} \\ S_{10} &= B_{11} + B_{12} \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\ P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\ P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A12 \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\ P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \end{aligned}$$

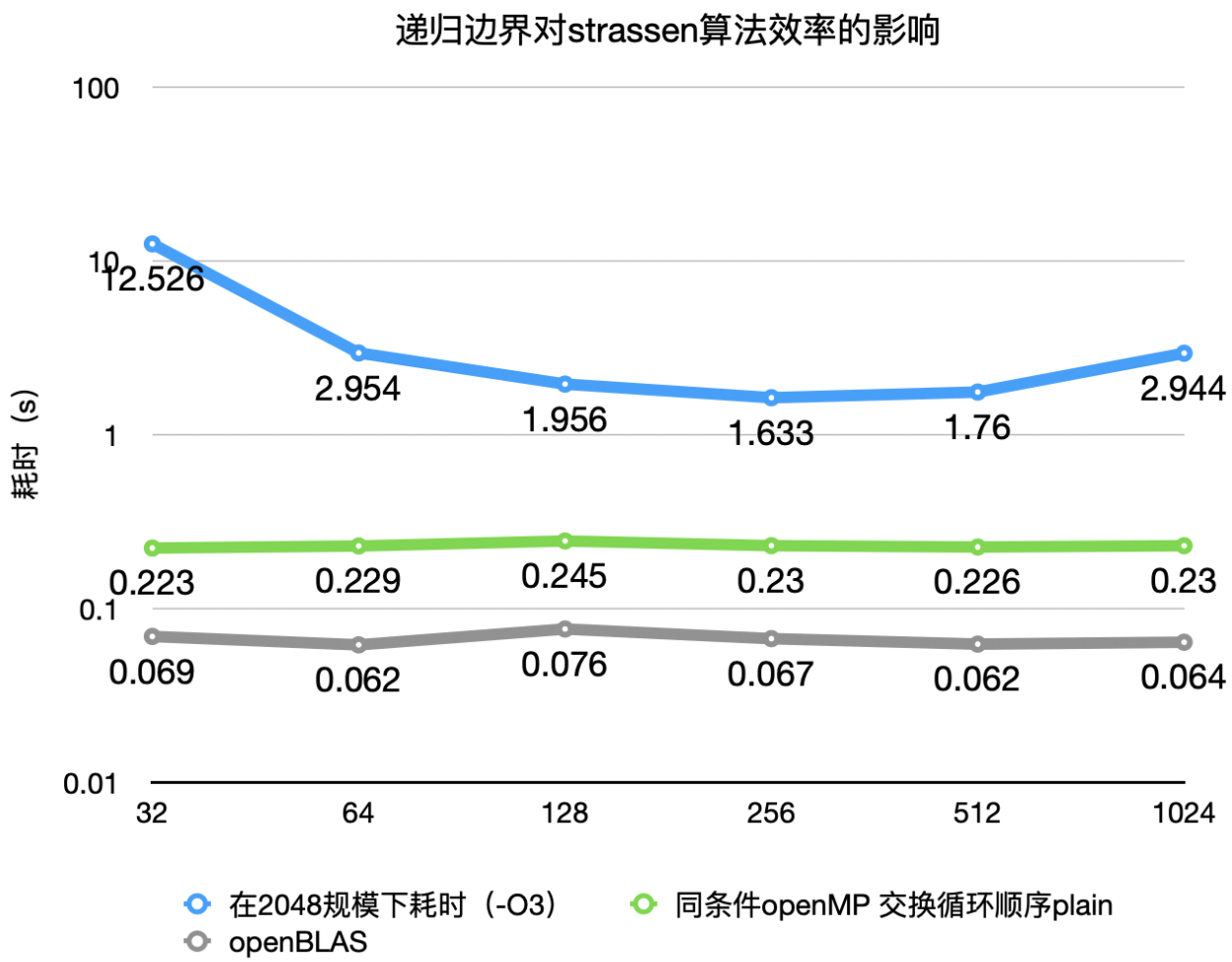
$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

$$
\text{综合可得如下递归式：} T(n) = \begin{cases} \Theta(1) & \text{若 } n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta\left(n^2\right) & \text{若 } n > 1 \end{cases}
$$

$$
\text{进而求出时间复杂度为：} T(n) = \Theta\left(n^{\log_2^7}\right)
$$

递归边界

strassen算法的核心在于分配计算任务，在递归到一定大小之后使用朴素算法求解。下图展示了递归边界大小对于strassen算法效率的影响，选用的测试矩阵规模为2048。可以看到，将边界设置为256时出现了耗时最短的拐点。因此在后续优化过程中均采用256作为递归边界。



及时释放内存

按照上述《算法导论》中展示的公式，所有中间过程生成的Matrix指针应该在计算完成之后、方法结束之前释放所占用的内存，但是在实际实现过程中可以通过调整计算顺序及时释放内存，达到更高的计算效率。

防止内存拷贝

因为中间过程需要生成多个子矩阵，如果进行大量内存拷贝不仅浪费资源，也会大大降低计算速度。于是采用类似老师上课讲到的openCV里面的“step”来避免内存拷贝。具体实现方法为，在计算乘法时传入 `size_t indexA, Matrix * A, size_t indexB, Matrix * B, size_t size)` 这五个参数，其中index表示需要选取的数据的起始位置，size表示矩阵规模（因为本次作业只需要进行方阵计算，在实现过程中将这部分进行了简化，如果需要计算非方阵则需要传入sizeRow和sizeCol）。同理，在矩阵加减法的实现过程中也传入了多个参数，分别为

`Matrix * A, size_t indexA, Matrix * B, size_t indexB, Matrix * C, size_t indexC, size_t size` 。具体计算时调用见下文代码。

创建一个全零矩阵指针

```
Matrix* createPlainMatrix(const size_t row, const size_t col){
    if (row == 0 || col == 0){
        fprintf(stderr, "rows and/or cols is 0!\n");
        return NULL;
    }
    Matrix *matrix = (Matrix *) malloc(sizeof(Matrix));
    if (matrix == NULL){
        fprintf(stderr, "failed to allocate memory!\n");
        return NULL;
    }
    matrix->col = col;
    matrix->row = row;
    matrix->value = (float *) aligned_alloc(128, row * col * sizeof(float));
    if (matrix->value == NULL){
        fprintf(stderr, "failed to allocate memory for value!\n");
        free(matrix);
        return NULL;
    }
    memset(matrix->value, 0, col * row * sizeof(float));

    return matrix;
}
```

strassen专用的加减法

```
void strAdd(Matrix * A, size_t indexA, Matrix * B, size_t indexB,
                        Matrix * C, size_t indexC, size_t size){
#ifdef WITH_NEON
    float32x4_t va, vb, vc;
#pragma omp parallel for
    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; j+=4){
            va = vld1q_f32(A->value + indexA + i * A->col + j);
            vb = vld1q_f32(B->value + indexB + i * B->col + j);
            vc = vaddq_f32(va, vb);
//#pragma omp critical
            vst1q_f32(C->value + indexC + i * C->col + j, vc);
        }
    }
    if (size % 4 != 0){
        for (int j = 0; j < size % 4; j++){
            C->value[indexC + (size-1) * C->col + j] =
                    A->value[indexA + (size-1) * A->col + j] +
                    B->value[indexB + (size-1) * B->col + j];
        }
    }
#else
    printf( "NEON and AVX are both not supported\n" );
        return NULL;
#endif
}

void strSubtract(Matrix * A, size_t indexA, Matrix * B, size_t indexB,
                                Matrix * C, size_t indexC, size_t size){
#ifdef WITH_NEON
    float32x4_t va, vb, vc;
#pragma omp parallel for
    for (size_t i = 0; i < size; ++i) {
```

```
        for (size_t j = 0; j < size; j+=4){
            va = vld1q_f32(A->value + indexA + i * A->col + j);
            vb = vld1q_f32(B->value + indexB + i * B->col + j);
            vc = vsubq_f32(va, vb);
//#pragma omp critical
            vst1q_f32(C->value + indexC + i * C->col + j, vc);
        }
    }
    if (size % 4 != 0){
        for (int j = 0; j < size % 4; j++){
            C->value[indexC + (size-1) * C->col + j] =
                    A->value[indexA + (size-1) * A->col + j] -
                    B->value[indexB + (size-1) * B->col + j];
        }
    }
#else
    printf( "NEON and AVX are both not supported\n" );
        return NULL;
#endif
}
```

分块和递归边界内部实现

```
Matrix *Strassen(size_t indexA, Matrix * A,
                                size_t indexB, Matrix * B, size_t size){
#ifdef WITH_NEON
    if (size <= 256) {
        Matrix *ret = createPlainMatrix(size, size);
        memset(ret->value, 0, ret->col * ret->row * sizeof(float));
        float32x4_t va, vb, vc;
#pragma omp parallel for
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                float t = *(A->value + indexA + A->col * i + j);
                for (int k = 0; k < size; k += 4) {
                    vb = vld1q_f32(B->value + indexB + B->col * j + k); //B[j][k]
                    vc = vld1q_f32(ret->value + ret->col * i + k); //C[i][k]
                    vc = vmlaq_n_f32(vc, vb, t);
                    vst1q_f32(ret->value + ret->col * i + k, vc);
                }
            }
        }
        return ret;
    }
#else
    if (size <= 256){
        Matrix * ret = createPlainMatrix(size, size);
        memset(ret->value, 0, ret->col * ret->row * sizeof(float));
#pragma omp parallel for
        for (size_t i = 0; i < size; i++)
        {
            for (size_t k = 0; k < size; k++)
            {
                for (size_t j = 0; j < size; j++)
                {
                    ret->value[i * ret->col + k] +=
                    A->value[indexA + i * A->col + j] *
                    B->value[indexB + j * B->col + k];
                }
            }
        }
        return ret;
    }
#endif
```

```
//分块
size_t newSize = (size >> 1);
size_t a11 = indexA, a12 = indexA + newSize,
a21 = a11 + newSize * A->col, a22 = a12 + newSize * A->col;
size_t b11 = indexB, b12 = indexB + newSize,
b21 = b11 + newSize * B->col, b22 = b12 + newSize * B->col;

//S1 = B12 - B22
Matrix * S1 = createPlainMatrix((B->row >> 1), (B->col >> 1));
strSubtract(B, b12, B, b22, S1, 0, newSize);
Matrix * P1 = Strassen(a11, A, 0, S1, newSize);
deleteMatrix(S1);

//S3 = A21 + A22
Matrix * S3 = createPlainMatrix((A->row >> 1), (A->col >> 1));
strAdd(A, a21, A, a22, S3, 0, newSize);
Matrix * P3 = Strassen(0, S3, b11, B, newSize);
deleteMatrix(S3);

//S5 = A11 + A22
Matrix * S5 = createPlainMatrix((A->row >> 1), (A->col >> 1));
strAdd(A, a11, A, a22, S5, 0, newSize);
//S6 = B11 + B22
Matrix * S6 = createPlainMatrix((B->row >> 1), (B->col >> 1));
strAdd(B, b11, B, b22, S6, 0, newSize);
Matrix * P5 = Strassen(0, S5, 0, S6, newSize);
deleteMatrix(S5);
deleteMatrix(S6);

//S9 = A11 - A21
Matrix * S9 = createPlainMatrix((A->row >> 1), (A->col >> 1));
strSubtract(A, a11, A, a21, S9, 0, newSize);
//S10 = B11 + B12
Matrix * S10 = createPlainMatrix((B->row >> 1), (B->col >> 1));
strAdd(B, b11, B, b12, S10, 0, newSize);
Matrix * P7 = Strassen(0, S9, 0, S10, newSize);
deleteMatrix(S9);
deleteMatrix(S10);

Matrix * C = createPlainMatrix(size, size);
size_t c11 = 0, c12 = c11 + newSize,
c21 = c11 + newSize * size, c22 = c21 + newSize;
strAdd(P5, 0, P1, 0, C, c22, newSize);
strSubtract(C, c22, P3, 0, C, c22, newSize);
strSubtract(C, c22, P7, 0, C, c22, newSize);
deleteMatrix(P7);


//S2 = A11 + A12
Matrix * S2 = createPlainMatrix((A->row >> 1), (A->col >> 1));
strAdd(A, a11, A, a12, S2, 0, newSize);
Matrix * P2 = Strassen(0, S2, b22, B, newSize);
deleteMatrix(S2);
strAdd(P1, 0, P2, 0, C, c12, newSize);
deleteMatrix(P1);


//S4 = B21 - B11
Matrix * S4 = createPlainMatrix((B->row >> 1), (B->col >> 1));
strSubtract(B, b21, B, b11, S4, 0, newSize);
Matrix * P4 = Strassen(a22, A, 0, S4, newSize);
deleteMatrix(S4);
strAdd(P3, 0, P4, 0, C, c21, newSize);
deleteMatrix(P3);
```

```
    //S7 = A12 - A22
    Matrix * S7 = createPlainMatrix((A->row >> 1), (A->col >> 1));
    strSubtract(A, a12, A, a22, S7, 0, newSize);
    //S8 = B21 + B22
    Matrix * S8 = createPlainMatrix((B->row >> 1), (B->col >> 1));
    strAdd(B, b21, B, b22, S8, 0, newSize);
    Matrix * P6 = Strassen(0, S7, 0, S8, newSize);
    deleteMatrix(S7);
    deleteMatrix(S8);
    strAdd(P5, 0, P4, 0, C, c11, newSize);
    strSubtract(C, c11, P2, 0, C, c11, newSize);
    strAdd(C, c11, P6, 0, C, c11, newSize);

    deleteMatrix(P2);
    deleteMatrix(P4);
    deleteMatrix(P5);
    deleteMatrix(P6);

    return C;

}
```

循环展开

写完strassen相关内容并完成strassen相关优化之后进行测试，发现测试结果不尽人意，于是尝试模仿openBLAS进行循环展开、cache命中等优化[5] 。矩阵分块之后，每次计算时，CPU可以把更规整的把部分内存搬到缓存中，提高缓存命中率[6]。在循环展开的基础上还利用了neon指令集进行加速。

```
#define UNROLL 4
void dgemm_neon_unroll(size_t n, float *A, float *B, float *C)
{
    for (size_t i = 0; i < n; i += 4*UNROLL) {
        for (size_t j = 0; j < n; j++) {
            float32x4_t c[UNROLL];
            for (int x = 0; x < UNROLL; x++) {
                c[x] = vld1q_f32(C + i + x * 4 + j * n);
            }
            for (size_t k = 0; k < n; k++) {
                float32x4_t b = vdupq_n_f32(B[k + j * n]);
                for (int x = 0; x < UNROLL; x++) {
                    c[x] = vaddq_f32(c[x],
                        vmulq_f32(vld1q_f32(A + i + k * n + x * 4), b));
                }
            }
            for (int x = 0; x < UNROLL; x++) {
                vst1q_f32(C + i + x * 4 + j * n, c[x]);
            }
        }
    }
}
```

cache blocking

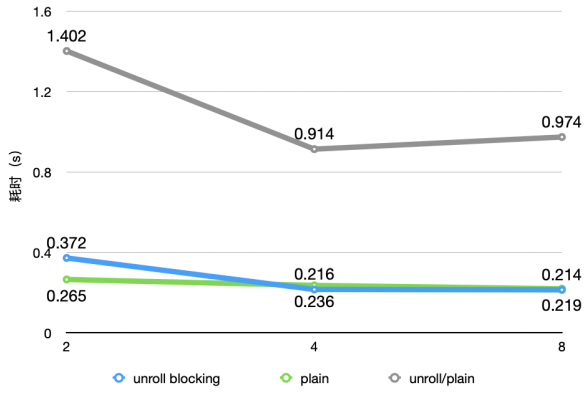
cache不变，矩阵越大，越难装进cache中，高速缓存缺失更多；矩阵大小不变，cache越大，越容易装更多的数据，高速缓存缺失更少。于是在unroll基础上拆分任务高效利用cache。

在测试过程中发现UNROLL和BLOCKSIZE值的大小会对计算效率产生影响，于是测试几组参数以寻求最优结果。先固定BLOCKSIZE为32测试UNROLL值对计算效率影响，获得最优UNROLL值后再调整BLOCKSIZE值比较计算速度，最后确定最优参数。测试过程中发现，UNROLL = 4时出现性能拐点，BLOCKSIZE = 128时出现性能拐点，于是后续测试过程

中均采用这组参数。以下图表中矩阵规模均为2048。

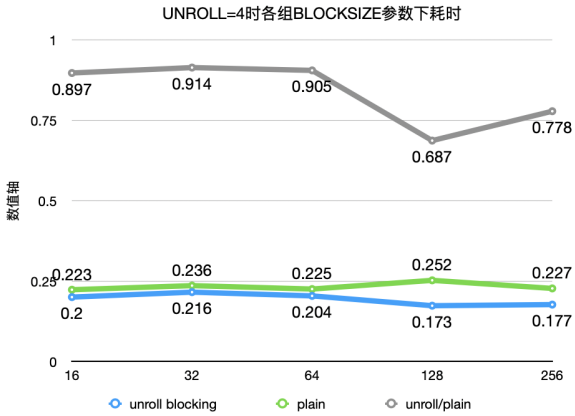
BLOCKSIZE = 32时各组UNROLL参数下耗时

unroll	unroll blocking	plain	unroll/plain
2	0.372353	0.265495	1.402486
4	0.215523	0.235761	0.914159
8	0.213691	0.219307	0.974392



UNROLL=4时各组BLOCKSIZE参数下耗时

blocksize	unroll blocking	plain	unroll/plain
16	0.200049	0.222959	0.897246
32	0.215523	0.235761	0.914159
64	0.203782	0.225089	0.905340
128	0.173242	0.252197	0.686931
256	0.176706	0.227034	0.778324



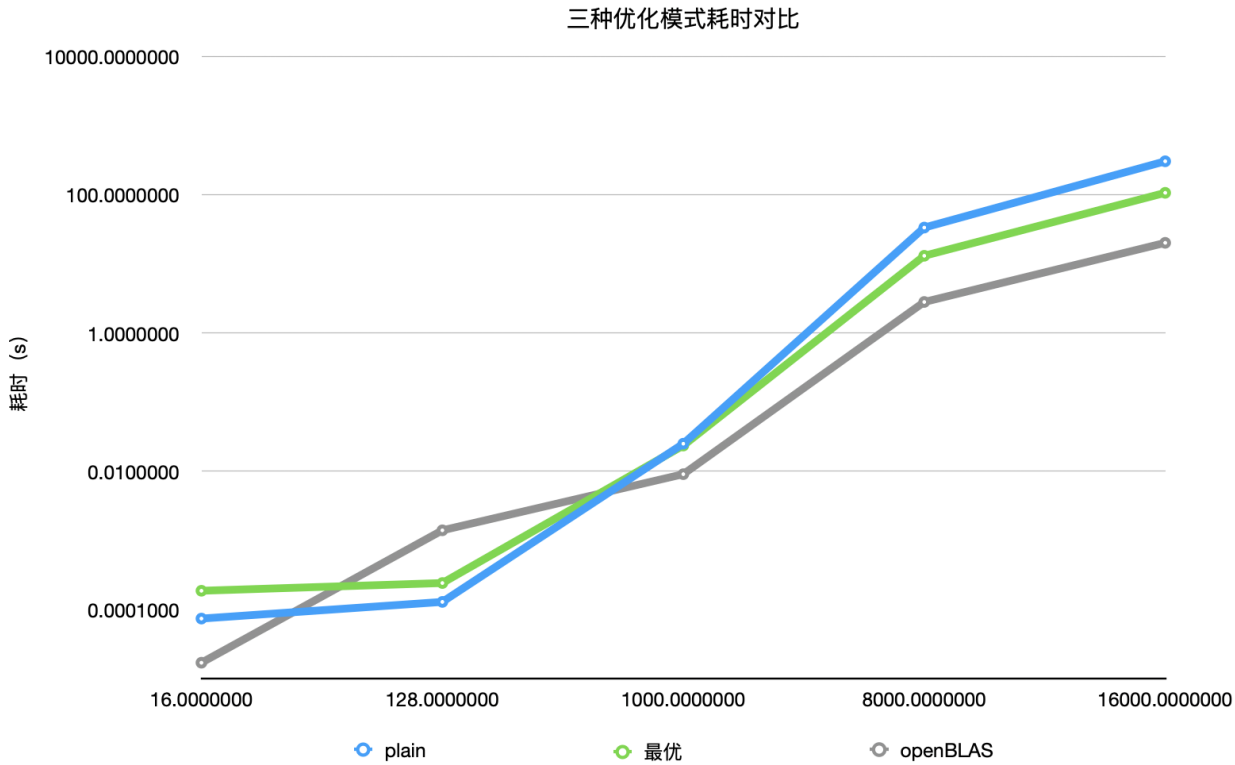

```
#define UNROLL 4
#define BLOCKSIZE 128

static inline void do_block(size_t n, size_t si, size_t sj, size_t sk,
                            float *A, float *B, float *C)
{
    for (size_t i = si; i < si + BLOCKSIZE; i += UNROLL*4) {
        for (size_t j = sj; j < sj + BLOCKSIZE; j++) {
            float32x4_t c[UNROLL];
            for (size_t x = 0; x < UNROLL; x++) {
                c[x] = vld1q_f32(C+i+x*4+j*n);
            }
            for (size_t k = sk; k < sk + BLOCKSIZE; k++) {
                float32x4_t b = vdupq_n_f32(B[k+j*n]);
                for (size_t x = 0; x < UNROLL; x++) {
                    c[x] = vaddq_f32(c[x],
                            vmulq_f32(vld1q_f32(A+n*k+x*4+i), b));
                }
            }

            for (size_t x = 0; x < UNROLL; x++) {
                vst1q_f32(C+i+x*4+j*n, c[x]);
            }
        }
    }
}

void dgemm_neon_unroll_blk(size_t n, float *A, float *B, float *C)
{
#pragma omp parallel for
    for (size_t sj = 0; sj < n; sj += BLOCKSIZE) {
        for (size_t si = 0; si < n; si += BLOCKSIZE) {
            for (size_t sk = 0; sk < n; sk += BLOCKSIZE) {
                do_block(n, si, sj, sk, A, B, C);
            }
        }
    }
}

bool neon_unroll(const Matrix *matrixLeft,
```

```
                                const Matrix *matrixRight, Matrix *result){
    if (!(matrixLeft && matrixRight && result && matrixLeft->value
            && matrixRight->value && result->value)){
        fprintf(stderr, "pointer NULL!\n");
        return 0;
    }
    if (matrixLeft->col != matrixRight->row)
    {
        fprintf(stderr,
                "two matrices can not be multiplied due to illegal size!\n");
        return 0;
    }
    size_t M = matrixLeft->row;
    size_t N = matrixRight->col;
    memset(result->value, 0, M * N * sizeof(float));
    dgemm_neon_unroll_blk(M, matrixLeft->value,
            matrixRight->value, result->value);
    return 1;
}
```

Part 03 - Result & Verifification

比较16x16, 128x128, 1Kx1K, 8Kx8K, 64Kx64K这五种规模下plain、最高速优化模式、openBLAS三者的效率。其中，由于plain在8k以上规模计算速度过慢，此处采用ikj算法和openMP对plain进行优化，并且在编译选项中加入-O3以提升测试速度。可以看到，在8k大规模测试中，最优计算速度达到了openBLAS计算速度的0.21457，在16k大规模测试中，最优计算速度达到了openBLAS计算速度的0.18890，也就是说差不多都达到openBLAS计算速度的五分之一左右水平。

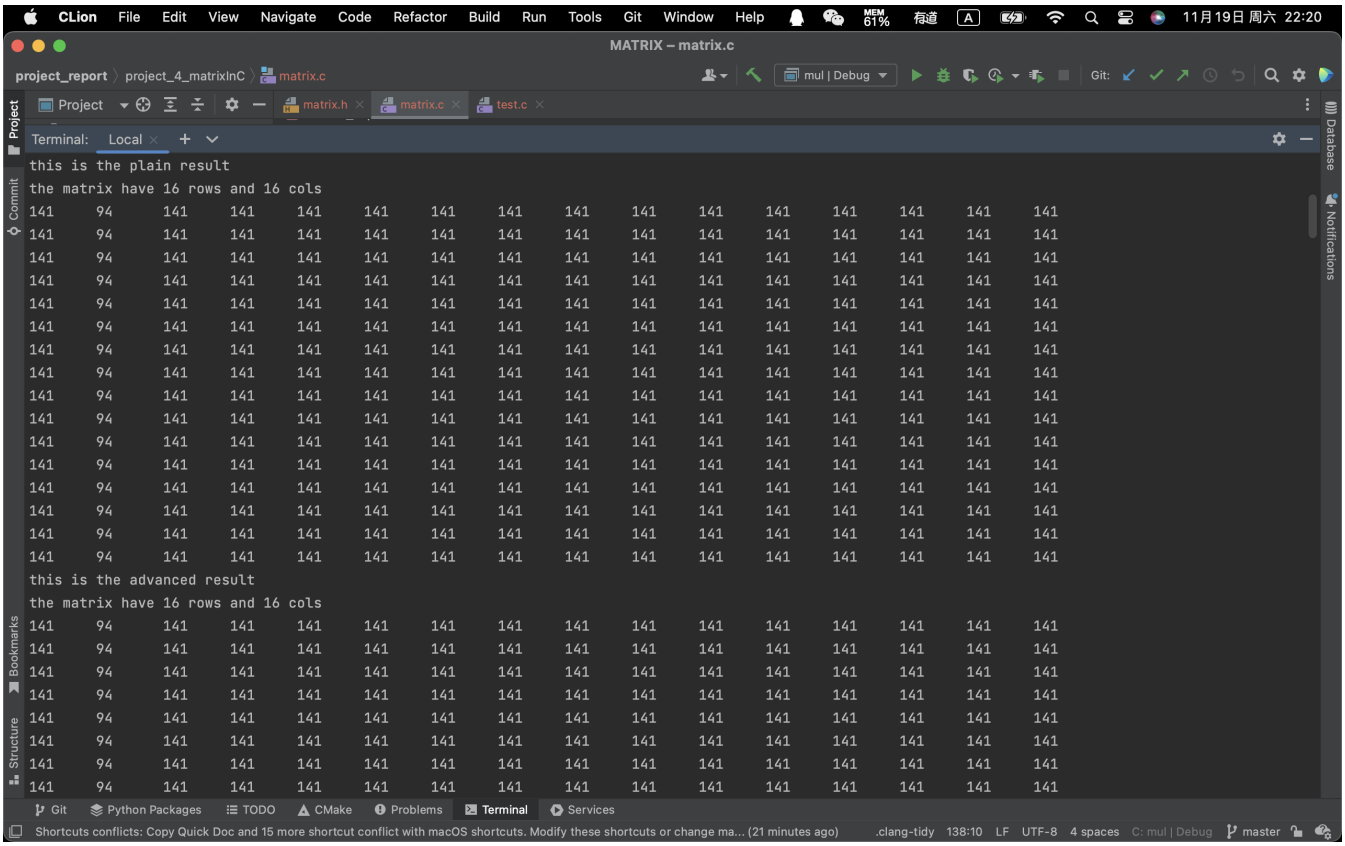
矩阵规模	plain	最优	openBLAS
16.0000000	0.0000740	0.0001870	0.0000170
128.0000000	0.0001290	0.0002420	0.0014020
1000.0000000	0.0251390	0.0236270	0.0091050
8000.0000000	33.732976	13.176344	2.827360
16000.0000000	307.329394	107.422931	20.292655

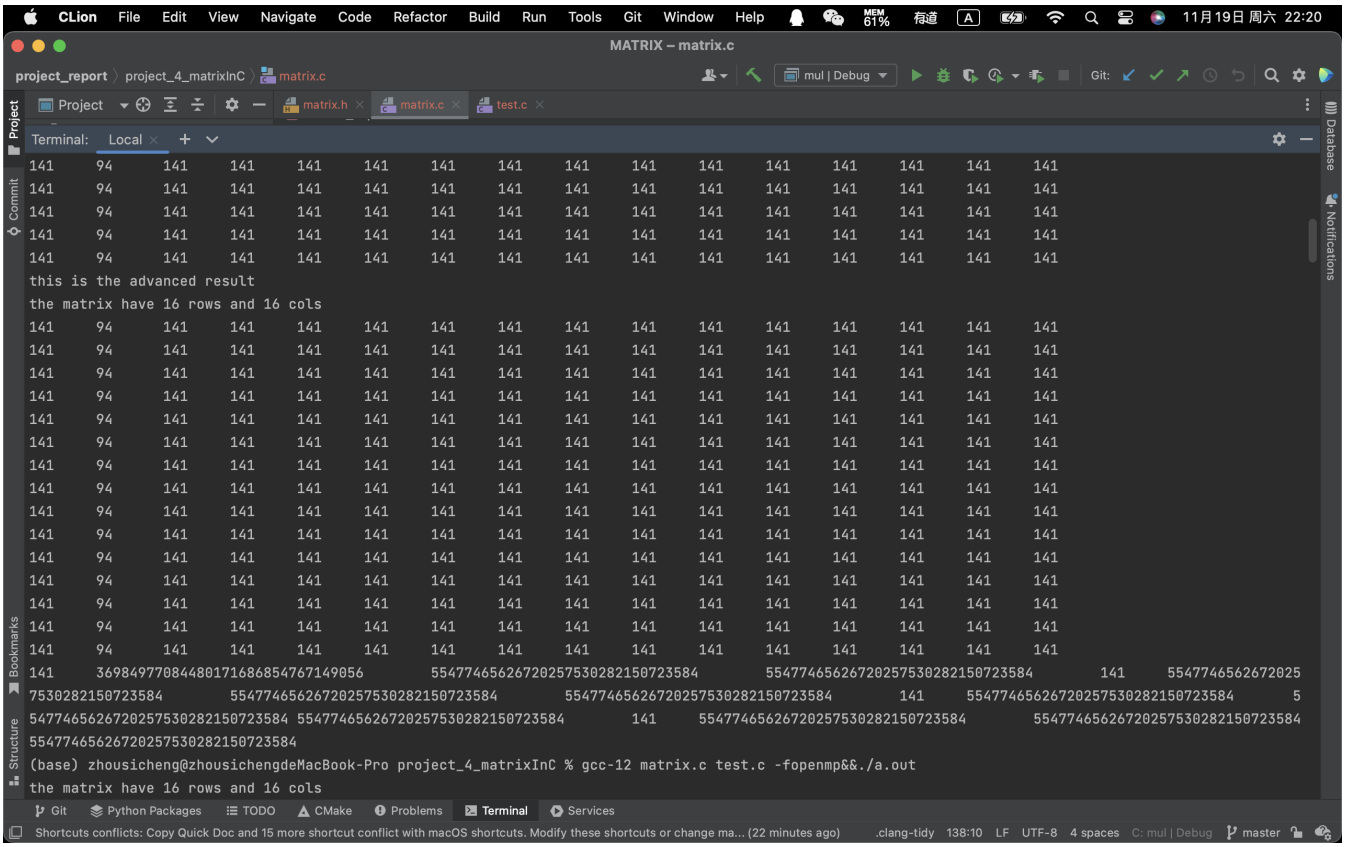


Part 04 - Difficulties & Solutions

内存管理

在进行优化追求计算速度之前，首先需要保证矩阵乘法计算正确性，因此用规模较小的矩阵作为例子打印其计算结果判断正确性。实现了使用SIMD指令的基本算法之后进行小规模测试，发现前面大部分计算结果都正确，但在最后一行出现了奇怪的错误。





上面一个矩阵是没有使用NEON指令集的普通计算函数的计算结果，下一个是使用之后的。二者对比发现，前面几行计算结果均正确，但最后一行使用指令集之后出现了奇怪的大数。

百思不得其解，让大佬帮忙看代码之后，大佬一针见血地指出：是不是内存没有对齐？看看自己的代码，确实，在`matmul_improved`里面给结果数组`result->value`分配内存的时候沿用的是`matmul_plain`里面的`malloc`，并不会自动对齐内存。指令集加速的本质是同时操作多个数据，比如simd256就是指同时操作256bit的数据。因此simd256技术要求所操作的数据的首地址是内存对齐32字节。这样的话，原先的malloc就不够用了，因为它不能分配给我们内存对齐于32字节的。[7]

用`aligned_alloc`代替`malloc`获得连续内存之后获得了正确的计算结果。

多核运行计时

在c中计时有多种方法。[8]

一开始本人使用的是最简单的`clock()`，该函数返回值是硬件滴答数，代码如下。

```
clock_t start,end;
start = clock();
//…calculating…
end = clock();
printf("time=%f\n",(double)end-start)/CLK_TCK);
```

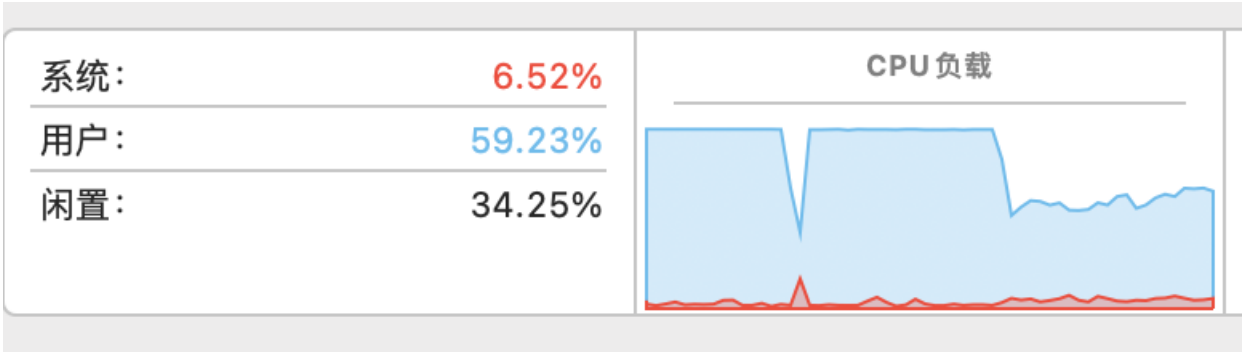
比较`matmul_plain`和仅添加指令集优化结果时，这个计时方法还能给出相对正确的结果，我们能看到随着矩阵规模的提升，指令集所带来的速度提升越来越大。但是，在使用openMP的情况下，却发现无论是`matmul_plain`还是`matmul_improved`，计算耗时都增加了几倍。和同学讨论并查阅资料后发现，`clock()`计算的是"the CPU time used so far"，所以在打开openMP多个CPU同时运行的时候，他会计算出更长的时间。

换成`gettimeofday()`之后发现openMP终于给出了令人欣慰的加速结果。

```
struct timeval start,end;
gettimeofday(&start, NULL );
//…calculating…
gettimeofday(&end, NULL );
long timeuse =1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
printf("time=%f\n",timeuse /1000000.0);
```

strassen算法的局限性

1. 丢失精度。由于strassen算法涉及多次矩阵拆分和加减法，在运算过程中可能造成精度的丢失，尤其这次使用的数据类型为float，更是容易出现丢失精度的情况。在测试过程中发现，当用于相乘的两个初始矩阵中的值均为整数时（float类型的整数，可能小数点后四五位才出现非零的数），strassen计算不出现误差。但当初始值有带小数后，就会出现较大的计算错误。如果将float改为double可能能稍稍改善这个问题。
2. 实际计算效率不高。理论上strassen算法的复杂度仅为 $T(n) = \Theta\left(n^{\log_2^7}\right)$ ，但是实际运行的时候却发现其计算速度比ikj+omp算法速度还要慢一些，且矩阵规模在512、1024、2048时（256是递归边界）均如此。下图展示了活动监视器在运行测试程序时现实的CPU负载情况，其中：最左侧是在计算最优的情况；第一个谷值是最优算法计算完成，程序在验证计算结果正确性；第二块峰值是openBLAS进行计算；右侧部分则是strassen算法运行时的CPU负载情况。可以看到，其负载完全没有达到前两种算法的负载水平，对于CPU的利用率相比之下非常低。猜想原因，应该是算法将大部分运行精力投放在了分解任务上，真正进行乘法计算的部分却很有限。可能是我的实现还有不完备之处，后续可以研究一下如何完善。



---

1. # OpenBLAS学习一：源码架构解析&GEMM分析 https://blog.csdn.net/frank2679/article/details/113243044 ↩
2. # 矩阵乘法&优化方法 - CPU篇 https://zhuanlan.zhihu.com/p/438173915 ↩
3. # AVX指令集加速矩阵乘法 http://t.csdn.cn/DzZfD ↩
4. 算法导论 第四章-分治策略 ↩
5. # 矩阵乘法优化过程（DGEMM） https://zhuanlan.zhihu.com/p/76347262 ↩
6. # 矩阵乘法的优化 http://t.csdn.cn/bqdBd ↩
7. # 【学习体会】aligned_malloc实现内存对齐 原文链接： https://blog.csdn.net/jin739738709/article/details/122992753 ↩
8. # C语言中常用计时方法总结 http://t.csdn.cn/V9V6z ↩