CS205 · C/C++ Programming

何泽安 (He Zean)
12011323

Project 3: Struct CMatrix

Oct. 8, 2021

Contents

1 需求分析 **2**

2 代码优化 **2**

2.1 减少解引用 . . . 2
2.2 寄存器变量 . . . 3
2.3 内联函数与宏 . . . 4

3 代码实现 **4**

3.1 结构体及相关函数 . . . 4
3.2 矩阵乘法 . . . 11
3.2.1 朴素乘法、调换顺序与分块访存优化 . . . 11
3.2.2 OpenBLAS、OpenCV . . . 13
3.2.3 向量化（SIMD）与循环展开 . . . 14
3.2.4 OpenMP 与多线程 . . . 16
3.3 主函数及辅助函数 . . . 17

4 测试样例及分析 **20**

4.1 矩阵乘法 . . . 21
4.1.1 基准：OpenBLAS 与朴素乘法 . . . 22
4.1.2 调换顺序 . . . 23
4.1.3 OpenCV . . . 24
4.1.4 SIMD 与循环展开 . . . 24
4.1.5 OpenMP 及多线程 . . . 25
4.2 部分相关函数 . . . 26
4.2.1 构造函数、随机矩阵及 MatRepr . . . 26
4.2.2 矩阵拷贝 . . . 27
4.2.3 保存至文件 . . . 28
4.2.4 转置矩阵及矩阵加减 . . . 28

5 困难及解决 **28**

5.1 面向过程编程 . . . 28
5.2 内联函数的使用 . . . 28
5.3 数组内存分配 . . . 29
5.4 内存管理 . . . 29

5.5 条件编译 29

6 总结 **29**

1 需求分析

本次项目需要实现 Matrix 结构体和提升矩阵乘法的性能。鉴于 Project2 中已经实现了六种矩阵乘，较为全面，本项目优先实现了一个“成员函数”[1]较全的结构体；并将上次的矩阵乘法翻译为 C 语言，更加深入探究其加速方案。此次我们不再关注算法带来的提升，故省略 Stressan，在上次 Project 的基础上，我们知道加速矩阵乘法主要有两方面值得我们努力：优化 IO 开销与提升并行计算能力。

本次项目与 Project 2 在存储数据方面最大的不同在于，Project 2 将矩阵元素存为二维数组[2]，而本次将 float 存为较长的一维数组：一方面在访问某一元素时可以节省一次解引用操作的时间，另一方面在调整计算顺序后能更好的减少跳转次数，做到对内存的连续访问（提高缓存命中率）。同时增加了对行主序和列主序的支持，使矩阵的存储更加自由高效。

在设计本项目所需实现的矩阵成员函数时，我参考 NumPy.array 和 cv::Mat 的 API，简单的实现了部分函数使 CMatrix 拥有了较为丰富的交互能力，出于时间考虑，略去了实现较为繁杂而并非体现代码能力的函数（如求行列式）。

此外，本次的代码中使用条件编译，具有更好的跨平台特性。

写在项目结束之际 早先受文章[3]启发，我们可以利用线性代数的计算法则来大幅度降低特殊矩阵（如方阵、对角阵、稀疏矩阵）的计算量。可惜初期认为要求中提及的矩阵乘法限制为随机生成的矩阵，其成为特殊矩阵的概率极低且需要为此写代码判断，故放弃了这种选择。以后的项目中可能会用上此方法。

2 代码优化

本节讨论实现代码中逐步发现的优化细节（语法层面的代码描述导致的计算机行为不同，而非加速方案），一般来说，在发现了某种写法优于原有方案后，后续会直接继续使用。

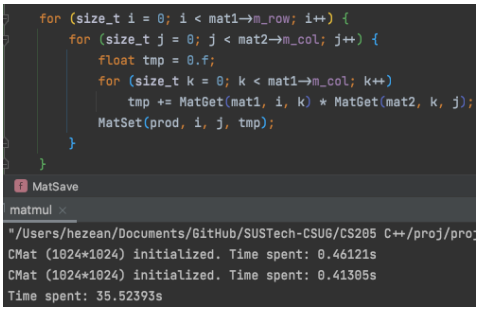
2.1 减少解引用

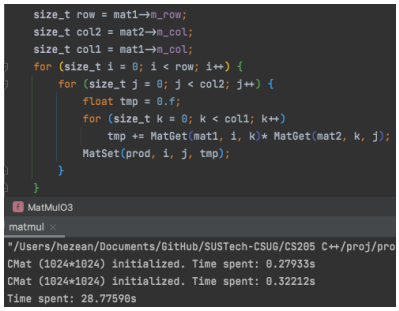
即尽量减少内存跳转数。在进行 for 循环中我们的条件语句可以直观的写为"i < mat->m_row" 的形式，但这意味着每次执行循环体前的检查都需要访问 mat 指针，再解引用跳转到其在堆内存里的元素；相较之下，我们可以只访问一遍改元素并将其保存为局部变量，这不仅带来了跳转数的减少的好处，也使得每次访问的是（尽管只有细微区别）较快的栈内存。

[1]面向过程编程严格来讲没有此说法，只是通过函数名前缀将与定义的结构体所关联起来的一种比喻。

[2]在此后的课上演示了在绝大部分情况下此操作是不必需的，如 4.2 节所述。但我们依然可以对一维数组的申请结果进行检查，一旦申请一维数组失败（malloc 返回 NULL），再使用二维数组初始化。出于时间考虑并未进行代码实现。

[3]zhuanlan.zhihu.com/p/227969338





如上图所示，减少解引用使 1024 阶矩阵乘法用时由 35.524s 提升至 28.776s（19.0% 的性能提升！），随矩阵阶数增加，for 语句执行次数增加，这个差别将更加明显。同理，在 *fill* 等对数据区有较多操作的函数里，我们不必对每个元素进行 *MatSet*，可以直接获取数据区指针并不断移动之。

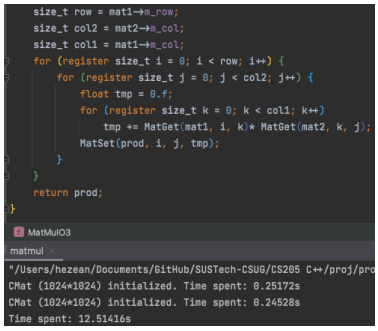
2.2 寄存器变量

C Primer Plus 中提到，在声明变量前加上 *register* 说明符可以请求编译器将变量保存在最快的可用内存中，一般来说是指 CPU 的寄存器中，但最终由编译器衡量是否保存至寄存器，也并非所有变量得以存入其中。这对于需要频繁访问的变量可以节省一些读写内存的 IO 开销。

以本段代码为例，如标识共有四处可能申请为寄存器变量：

```
1 for (/* register (1) */ size_t i = 0; i < row; i++) {
2     for (/* register (2) */ size_t j = 0; j < col2; j++) {
3         /* register (3) */ float tmp = 0.f;
4         for (/* register (4) */ size_t k = 0; k < col1; k++)
5             tmp += MatGet(mat1, i, k)* MatGet(mat2, k, j);
6         MatSet(prod, i, j, tmp);
7     }
8 }
```

下表所示为计算 1024 阶矩阵朴素乘法（经一次数据预热）所耗时，其中局部变量是指 Line 5 的 tmp，循环变量即 ijk。可见将访问频繁的数据申请为寄存器，仅在代码中加一个关键字，竟能带来约 50% 的性能提升！当然，这类工作编译器在开启 O3 选项后也会帮我们完成。虽然这里外层循环变量访问相对不频繁，但寄存器是否真正被申请到由编译器权衡决定，方便起见我们可以在大部分需要频繁更新的数据前加上此关键字。



	无寄存器变量	局部变量	循环变量	循环 + 局部
计算耗时 / s	26.0477	14.1541	13.3021	12.5141
节省时间 / %	-	45.66	48.93	51.96

2.3 内联函数与宏

课上提及，如果被调函数功能较为简单，调用函数时产生的保存上下文（将主调函数状态“存快照”）的开销相较于函数功能本身的开销就显得比重较大，得不偿失。与其相似的是定义宏，都是将函数体放入符号表并在调用处展开，但宏作为简单的文本替换，使用起来不如内联函数“安全”，在本项目中两种方法均会使用。

```c
void costTime() {  // is called in main for speed test
    for (register size_t i = 0; i < 10000000000; i++)
        getOne();
}

int getOne() { return 1; }  // Before: Time cost ~ 12.18s
static inline int getOne() { return 1; }  // (defined in header file) After: Timer cost ~ 10.86s
```

3 代码实现

3.1 结构体及相关函数

```c
// cmatrix.h

#ifndef MATMUL_CMATRIX_H
#define MATMUL_CMATRIX_H

#include <math.h>
#include <stdbool.h>
#include <stdio.h>

#include "util.h"

#define FLOAT_EQL(f1, f2) (fabsf((f1) - (f2)) > 1e-5)

typedef enum {
    RowMajor, ColMajor,
} kMatData;

typedef struct {
    size_t m_row;
    size_t m_col;
    kMatData m_storage;
    float *m_data;
} CMatrix;
```

```c

typedef const CMatrix *ReadOnlyMat;

/**
 * To simplify the implementation, only row-major matrices are supported
 */
typedef struct {
    size_t m_row;
    size_t m_col;
    kMatData m_storage;
    float **m_data_ptr;
} CMatrixView;

CMatrix *MatEmpty(size_t row, size_t col);
CMatrix *MatFromFile(size_t row, size_t col, FILE *f);
CMatrix *MatFromFileTrans(size_t ori_row, size_t ori_col, FILE *f);
CMatrix *MatClone(ReadOnlyMat ori);
CMatrix *MatZeros(size_t row, size_t col);
CMatrix *MatOnes(size_t row, size_t col);
CMatrix *MatEye(size_t size);  /* Creates an identify matrix (size*size) */
CMatrix *MatRand(size_t row, size_t col, float min, float max);

/**
 * Fill all the elems in the array to a diagonal matrix
 * While the other elements of such (size*size) matrix are set zeros
 */
CMatrix *MatDiag(size_t size, const float *diag_elems);

static inline void MatDelete(CMatrix *mat) {
    free(mat->m_data);
    free(mat);
}

/**
 * Getter of a CMatrix (m*n)
 * @param mat can be either row-maj or col-maj
 * @param row [0..m-1]
 * @param col [0..n-1]
 * @returns the value of mat[row][col]
 *          if access out of bound, returns NAN
 */
static inline float MatGet(ReadOnlyMat mat, size_t row, size_t col) {
    if (row > mat->m_row || col > mat->m_col) return NAN;
    if (mat->m_storage == RowMajor)
        return mat->m_data[row * mat->m_col + col];
    else return mat->m_data[col * mat->m_row + row];
}

/**
 * If the index out of bounds, do nothing
```

```
 */
static inline void MatSet(CMatrix *mat, size_t row, size_t col, float val) {
    if (row >= mat->m_row || col >= mat->m_col) return;
    if (mat->m_storage == RowMajor)
        mat->m_data[row * mat->m_col + col] = val;
    else mat->m_data[col * mat->m_row + row] = val;
}

/**
 * Copy the data from matrix <src> to <dst>
 * without changing the size of <dst>
 * To simplify the implementation, only row-major matrix is supported
 *
 * If <src> is bigger(row or col) than <dst> then
 *      to avoid data loss, will not do anything
 *      also prompt the user
 * If <src> is smaller(row or col) than <dst> then
 *      copy the available data, set the blank as zeros
 *
 * @param src of size (m*n), with m_data[m*n]
 * @param dst of size (x*y)
 * @return false if it is possible to loss data,
 *              caller shall check the return value
 *              to avoid further exceptions
 *         true otherwise
 */
bool MatCpy(ReadOnlyMat src, CMatrix *dst);

/**
 * Fill (overwrite) the data of a mat and fill them all with <elem>
 */
void MatFill(CMatrix *mat, float elem);

/**
 * Make a matrix in an original size m*n to behave as nrow*ncol
 *
 * @param mat the matrix to be resized
 * @param nrow the new size of row it should be
 * @param ncol should satisfies that nrow*ncol=m*n
 * @returns false if the new size mismatch, true otherwise
 */
bool MatResize(CMatrix *mat, size_t nrow, size_t ncol);

bool MatEquals(ReadOnlyMat self, ReadOnlyMat oppo);
void MatTrans(CMatrix **mat);

/**
 * Deep clone the sub-matrix (srow_len*scol_len)
 * It is equivalent to mat[rowst:rowst+srow_len][colst:colst+scol_len]
 */
```

```
CMatrix *MatSubCopy(ReadOnlyMat mat, size_t rowst, size_t srow_len, size_t colst, size_t
    scol_len);

CMatrixView *MatViewSub(ReadOnlyMat mat, size_t rowst, size_t srow_len, size_t colst, size_t
    scol_len);
void MatViewDel(CMatrixView *mv);

/**
 * mat1,2,3,4 must be the same size
 * @returns [[mat1] [mat2]
 *           [mat3] [mat4]]
 */
CMatrix *MatConcat(ReadOnlyMat mat1, ReadOnlyMat mat2, ReadOnlyMat mat3, ReadOnlyMat mat4);

/**
 * This function won't create any new CMatrix instance,
 * but updates the data of <to>
 * Aka. m1=a, m2=b; MatAdd(m1,m2); m1==a && m2==b+a;
 *
 * @param it a CMatrix instance, its data won't be affected
 * @param to a CMatrix instance, must have the same size as <this>
 */
void MatAdd(ReadOnlyMat it, CMatrix *to);

/**
 * This function won't create any new CMatrix instance,
 * but updates the data of <dst>
 * Aka. m1=a, m2=b; MatSub(m1,m2); m1==a && m2==b-a;
 *
 * @param by a CMatrix instance, its data won't be affected
 * @param dst a CMatrix instance, must have the same size as <this>
 */
void MatSub(ReadOnlyMat by, CMatrix *dst);

void MatShowRow(ReadOnlyMat mat, size_t r);
void MatRepr(ReadOnlyMat mat);
void MatSave(ReadOnlyMat mat, const char *filename);

#endif //MATMUL_CMATRIX_H
```

部分代码较为简单或相似，若有需要请查阅附件源码。以下仅展示部分函数的实现。

```
CMatrix *MatEmpty(size_t row, size_t col) {
    CMatrix *mat = (CMatrix *) malloc(sizeof(CMatrix));
    mat->m_row = row;
    mat->m_col = col;
    mat->m_storage = RowMajor;
    mat->m_data = (float *) malloc(sizeof(float) * row * col);
    return mat;
}
```

为了统一思路，我们将矩阵的数据区和矩阵实例均存入堆内存（因此返回指针是安全的），并在程序中始终操作指向矩阵实例的指针，这也使得传递参数的大小略微减少。

```
CMatrix *MatFromFile(size_t row, size_t col, FILE *f) {
    TIMER_START
    CMatrix *mat = MatEmpty(row, col);
    float tmp;
    float *cur = mat->m_data;
    for (register size_t i = 0; i < row; i++) {
        for (register size_t j = 0; j < col; j++) {
            if (fscanf(f, "%f", &tmp) == EOF) break;
            *(cur++) = tmp;
        }
    }
    printf("CMat (%lu*%lu) initialized. ", row, col);
    fclose(f);
    TIMER_END
    return mat;
}

void MatSave(ReadOnlyMat mat, const char *filename) {
    TIMER_START
    FILE *f = fopen(filename, "w+");
    size_t row = mat->m_row, col = mat->m_col;
    for (register int i = 0; i < row; i++) {
        for (register int j = 0; j < col; j++)
            fprintf(f, "%.4f ", MatGet(mat, i, j));
        fprintf(f, "\n");
    }
    if (fclose(f) != 0) {
        printf("Error saving CMat to <%s>: %s", filename, strerror(errno));
        return;
    }
    printf("Answer saved to <%s>\t", filename);
    TIMER_END
}
```

```
CMatrix *MatFromFileTrans(size_t ori_row, size_t ori_col, FILE *f) {
    TIMER_START
    CMatrix *mat = MatEmpty(ori_col, ori_row);
    mat->m_storage = ColMajor;
    float tmp;
    float *data = mat->m_data;
    for (register size_t i = 0; i < ori_row; i++) {
        for (register size_t j = 0; j < ori_col; j++) {
            if (fscanf(f, "%f", &tmp) == EOF) break;
            data[j * ori_col + i] = tmp;
```

```
        }
    }
    printf("Transposed CMat (%lu*%lu) initialized. ", ori_col, ori_row);
    fclose(f);
    TIMER_END
    return mat;
}
```

MatFromFileTrans 将矩阵直接初始化为转置矩阵（或理解为以列主序的方式存储原矩阵，只需对代码稍作修改）。

```
CMatrix *MatClone(ReadOnlyMat ori) {
    CMatrix *new = MatEmpty(ori->m_row, ori->m_col);
    new->m_storage = ori->m_storage;
    memcpy(new->m_data, ori->m_data, sizeof(float) * ori->m_row * ori->m_col);
    return new;
}

bool MatCpy(ReadOnlyMat src, CMatrix *dst) {
    if (src->m_row > dst->m_row || src->m_col > dst->m_col) {
        printf("Cannot copy matrix: src(%lu*%lu) -> dst(%lu*%lu)\n",
               src->m_row, src->m_col,
               dst->m_row, dst->m_col);
        return false;
    }
    if (src->m_row * src->m_col == dst->m_row * dst->m_col)
        memcpy(dst->m_data, src->m_data, sizeof(float) * src->m_row * src->m_col);
    else {
        for (int r = 0; r < src->m_row; r++) {
            memcpy(dst->m_data + r * dst->m_col,
                   src->m_data + r * src->m_col,
                   sizeof(float) * src->m_col);
            memset(dst->m_data + r * dst->m_col + src->m_col,
                   0, sizeof(float) * (dst->m_col - src->m_col));
        }
        printf("Copying matrix (%lu*%lu) -> (%lu*%lu), the blanks are set zero\n",
               src->m_row, src->m_col,
               dst->m_row, dst->m_col);
    }
    return true;
}
```

MatClone 是创建一个新副本，MatCpy 则将一个矩阵的数据拷贝到另一个矩阵中，不改变其大小。

```
void MatFill(CMatrix *mat, float elem) {
    float *cur = mat->m_data;
    register size_t cnt = mat->m_row * mat->m_col;
    while (cnt--) *(cur++) = elem;
}
```

```

CMatrix *MatDiag(size_t size, const float *diag_elems) {
    CMatrix *mat = MatZeros(size, size);
    for (register size_t i = 0; i < size; i++)
        MatSet(mat, i, i, diag_elems[i]);
    return mat;
}
```

MatFill 使用第二节的结论，通过减少解引用（保存数据区指针）和申请寄存器变量来加速覆盖矩阵数据区。MatZeros、MatOnes、MatEye 的实现与 MatDiag 类似，故不列出。

```
CMatrix *MatRand(size_t row, size_t col, float min, float max) {
    CMatrix *mat = MatEmpty(row, col);
    register size_t cnt = mat->m_row * mat->m_col;
    float *cur = mat->m_data;
    while (cnt--)
        *(cur++) = (rand() % (int) ((max - min) * 10000.f)) / 10000.f + min;
    return mat;
}
```

以本次实际对其调用为例，为使得多个随机矩阵互不相同，整个程序中只能出现一次 *srand(time(NULL))*，即在主函数中手动初始化一次种子。

```
bool MatEquals(ReadOnlyMat self, ReadOnlyMat oppo) {
    if (self->m_row != oppo->m_row ||
        self->m_col != oppo->m_col)
        return false;
    float *curSelf = self->m_data;
    float *curOppo = oppo->m_data;
    register size_t cnt = self->m_row * self->m_col;
    while (cnt--)
        if (!FLOAT_EQL(*(curSelf++), *(curOppo++))) return false;
    return true;
}
```

注意到 float 的表示存在误差，两个相同的数在计算机里的 float 表示可能不同，故这里设置一个阈值，当两个 float 的值足够接近，即认为它们相等。

```
CMatrix *MatSubCopy(ReadOnlyMat mat, size_t rowst, size_t srow_len, size_t colst, size_t
    scol_len) {
    CMatrix *sub = MatEmpty(srow_len, scol_len);
    sub->m_storage = mat->m_storage;
    for (int i = 0; i < srow_len; ++i)
        for (int j = 0; j < scol_len; ++j)
            MatSet(sub, i, j, MatGet(mat, rowst + i, colst + j));
    return sub;
}
```

```c
CMatrixView *MatViewSub(ReadOnlyMat mat, size_t rowst, size_t srow_len, size_t colst, size_t
    scol_len) {
    if (mat->m_storage == ColMajor) return NULL;
    CMatrixView *view = (CMatrixView *) malloc(sizeof(CMatrixView));
    view->m_row = srow_len;
    view->m_col = scol_len;
    view->m_data_ptr = (float **) malloc(sizeof(float *) * srow_len);
    for (int i = 0; i < srow_len; ++i)
        view->m_data_ptr[i] = mat->m_data + (i * mat->m_col + colst);
    return view;
}

void MatViewDel(CMatrixView *mv) {
    free(mv->m_data_ptr);
    free(mv);
}
```

MatSubCopy 是将矩阵的某一块子矩阵进行深拷贝；而 MatViewSub 相当于为子矩阵 “开窗”，创建了指向其的指针，使用 View 虽然需要多解一次引用，但减少了深拷贝需要的大量时间及减少内存消耗，这可能对 Stressan 算法有所帮助。另外，销毁一个 View 对象时，只对保存多个指向原矩阵数据区的（外层）数组进行 free，这是因为我们不希望破坏原矩阵数据。

3.2 矩阵乘法

3.2.1 朴素乘法、调换顺序与分块访存优化

```c
CMatrix *MatMul03(ReadOnlyMat mat1, ReadOnlyMat mat2) {
    if (mat1->m_col != mat2->m_row) {
        printf("MathErr: Mat(%lu*%lu) * Mat(%lu,%lu) is invalid\n",
               mat1->m_row, mat1->m_col, mat2->m_row, mat2->m_col);
        return NULL;
    }
    CMatrix *prod = MatEmpty(mat1->m_row, mat2->m_col);

    size_t row = mat1->m_row;
    size_t col2 = mat2->m_col;
    size_t col1 = mat1->m_col;
    for (register size_t i = 0; i < row; i++) {
        for (register size_t j = 0; j < col2; j++) {
            register float tmp = 0.f;
            for (register size_t k = 0; k < col1; k++)
                tmp += MatGet(mat1, i, k) * MatGet(mat2, k, j);
            MatSet(prod, i, j, tmp);
        }
    }
    return prod;
}
```

与 Project2 的不同在于，矩阵使用一维数组存储，且使用第 2 节提及的三种语法层面的优化。

调换计算顺序的代码主体与朴素乘法相近，仅循环变量的顺序不同，故省略展示代码。从访存的角度来看[4]，使用不同顺序 Get 矩阵元素背后所影响的是缓存命中率的高低。方便起见，以下认为左右矩阵均为 n 阶方阵。

ijk 上述朴素乘法为了计算乘积矩阵中的一个元素，就需要访问左矩阵的一行（先从行尾跳转一次到行首，然后连续的 n 个）及右矩阵中（在内存中不连续的）n 个元素，现在对于乘积中的 n^2 个元素，总共跳转了 $n^3 + n^2$ 次，但考虑到计算完每行最后一列的元素后（C[i][n]），此时左矩阵刚访问完 A[i][n]，计算下一元素 C[i+1][1] 时左矩阵（一维数组形式）不需要跳转，故将跳转次数修正为 $n^3 + n^2 - n$。

ikj 类似于打印机的原理，对于 C 中每行都会累积叠加计算多次，这样每计算出 C 的一行需要遍历左矩阵中的对应行以及整个右矩阵。由于使用的是一维数组存储，左矩阵始终没有跳转，右矩阵在遍历结束，开始计算下一行时需要一次跳转，而矩阵 C 需要跳转一次。因此为了计算一行，跳转数为 n，对于总共的 n 行，需要 n^2 次跳转。

前面一直在讨论缓存命中率，现思考如何将缓存（主要指 L1）充分利用[5]。显然，对于特定计算机，缓存一定，矩阵越大，越“难”载入缓存中，高速缓存将缺失更多数据，要利用缓存大小的限制（局部性），我们考虑将矩阵分割成适合放入缓存的大小。

```
void do_block(size_t n, const float *m1dat, const float *m2dat, float *prodat) {
    for (int i = 0; i < BLOCKSIZE; i++) {
        for (int j = 0; j < BLOCKSIZE; j++) {
            float tmp = prodat[i * n + j];
            for (int k = 0; k < BLOCKSIZE; k++)
                tmp += m1dat[i * n + k] * m2dat[k * n + j];
            prodat[i * n + j] = tmp;
        }
    }
}

CMatrix *MatMulBlock(ReadOnlyMat mat1, ReadOnlyMat mat2) {
    if (mat1->m_col != mat2->m_row) {
        printf("MathErr: Mat(%lu*%lu) * Mat(%lu,%lu) is invalid\n",
               mat1->m_row, mat1->m_col, mat2->m_row, mat2->m_col);
        return NULL;
    }
    size_t sz = mat1->m_row;
    CMatrix *prod = MatZeros(sz, sz);

    float *dat1 = mat1->m_data;
    float *dat2 = mat2->m_data;
    float *dat3 = prod->m_data;
    for (size_t i = 0; i < sz; i += BLOCKSIZE) {
```

[4] C++ 加速矩阵乘法的最简单方法 (zhuanlan.zhihu.com/p/146250334)
[5] 实验设计思想参考：矩阵乘法的分块优化 (blog.csdn.net/weixin_40673608/article/details/88135041)

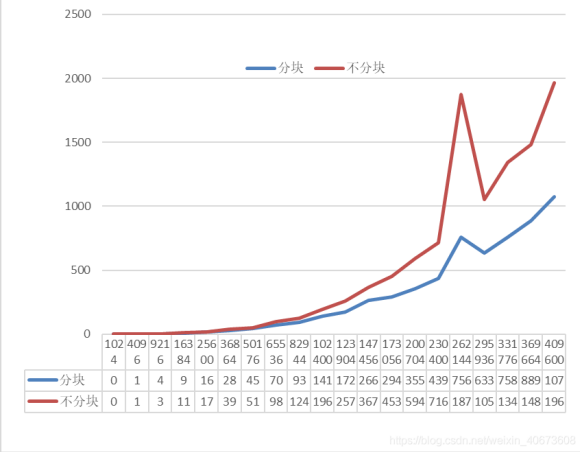
```
        for (size_t k = 0; k < sz; k += BLOCKSIZE) {
            for (size_t j = 0; j < sz; j += BLOCKSIZE) {
                do_block(sz, dat1 + i * sz + k,
                         dat2 + k * sz + j,
                         dat3 + i * sz + j);
            }
        }
    }
    return prod;
}
```

这里的 BLOCKSIZE 与 CPU 有关，本机（Core i5-1038NG7）的 L1 缓存为 4x32+4x48 kB，理论上将矩阵分割为

$$\sqrt{320_{\text{kb}} \times 1024_{\text{byte/kb}} \div 8_{\text{sizeof(float)}} \div 3_{\text{for left mat, right mat and prod}}}$$

大小的方阵能最大限度利用 L1 缓存（实际能使用的缓存远小于此，因为电脑并非只需将所有硬件性能交给我们的程序）。

时间所限未能对此进行完整的测速分析，已有的两组数据（2048 阶方阵：BLOCKSIZE=32，耗时 25.49s；BLOCKSIZE=1 模拟未分块，耗时 81.42s）说明分块合理利用缓存可以减少擦写，减少 IO 开销。别人的实验结果[6]能更好的验证此说法[7]：



3.2.2 OpenBLAS、OpenCV

```
CMatrix *MatMulBlas(ReadOnlyMat mat1, ReadOnlyMat mat2) {
    /* Validity checking, same as MatMul03 */
    CMatrix *prod = MatEmpty(mat1->m_row, mat2->m_col);

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                mat1->m_row, mat2->m_col, mat1->m_col,
                1.f,
                mat1->m_data, mat1->m_col,
```

[6]图源：blog.csdn.net/weixin_40673608/article/details/88135041

[7]“在矩阵大小小于 L1 缓存的时候的时候，矩阵可以完全装到 cache 中，所以 2 种方法速度差不多，但是由于分块有多余的函数调用开销，所以分块会比不分块慢些；在矩阵大小大于 L1 缓存的时候的时候，不分块矩阵没法完全装进 L1 缓存中，而分块矩阵可以装进 L1 缓存中，所以不分块矩阵乘法会比分块矩阵乘法慢。”

```
                mat2->m_data, mat2->m_col,
                0.f,
                prod->m_data, mat2->m_col);
    return prod;
}
```

在 Project 2 中，由于数据保存为二维数组，无法使用 sgemm，我使用 OpenBLAS 分别对乘积矩阵中的每个元素求向量点积，但极为低效；本次项目使用一维数组存储矩阵，也得以使用其“正统做法”：

$$\text{sgemm(mat1, mat2, alpha, mat3, beta)} \equiv \text{mat3} = \text{alpha} \times \text{mat1} \times \text{mat2} + \text{beta} \times \text{mat3}$$

```
CMatrix *MatMulCV(ReadOnlyMat mat1, ReadOnlyMat mat2) {
    /* Validity checking, same as MatMul03 */
    CMatrix *prod = MatEmpty(mat1->m_row, mat2->m_col);
    CvMat cvm1 = cvMat(mat1->m_row, mat1->m_col, CV_32F, mat1->m_data);
    CvMat cvm2 = cvMat(mat2->m_row, mat2->m_col, CV_32F, mat2->m_data);
    CvMat cvm3 = cvMat(mat1->m_row, mat2->m_col, CV_32F, prod->m_data);
    cvGEMM(&cvm1, &cvm2, 1, 0, 0, &cvm3, 0);
    // cvMatMul(&cvm1, &cvm2, &cvm3);  // we can use the macros instead
    return prod;
}
```

3.2.3 向量化（SIMD）与循环展开

```
CMatrix *MatMulSimd(ReadOnlyMat mat1, ReadOnlyMat mat2) {
    CMatrix *m2trans = MatClone(mat2);
    MatTrans(&m2trans);

    size_t sz = mat1->m_row;
    CMatrix *prod = MatEmpty(sz, sz);
    float *data1 = mat1->m_data;
    float *data2 = m2trans->m_data;

#if defined(_AVX512)
    /* The core idea is same to <AVX2>, only the functions called are different */
#elif defined(_AVX2)
    float *tmp = aligned_alloc(sizeof(float) * 8, 256);
    for (register size_t i = 0; i < sz; i++) {
        for (register size_t j = 0; j < sz; j++) {
            __m256 c = _mm256_setzero_ps();
            for (register size_t k = 0; k < sz; k += 8) {
                c = _mm256_add_ps(_mm256_mul_ps(
                                        _mm256_load_ps(data1 + i * sz + k),
                                        _mm256_load_ps(data2 + j * sz + k)),
                                c);
            }
            _mm256_store_ps(tmp, c);
            MatSet(prod, i, j, tmp[0] + tmp[1] + tmp[2] + tmp[3]
```

```
                                + tmp[4] + tmp[5] + tmp[6] + tmp[7]);
        }
    }
#elif defined(_NEON)  // Acknowledgement: ShiqiYu@libfacedetection
    /* The core idea is same to <AVX2>, only the functions called are different */
#else
    printf("Must select one from AVX2, AVX512 and NEON. quitting...\n");
    exit(-1);
#endif

#if defined(_AVX512) || defined(_AVX2)
    free(tmp);
#endif
    MatDelete(m2trans);
    return prod;
}
```

在 Project 2 的基础上修改部分代码使之符合一维数组的存储方式，另外使用了条件编译增加了对 ARM 平台的支持[8]。

出于简化实现考虑，我们假设输入方阵在点乘方向上的向量长度均为 8 的倍数（AVX2，float 装入 __m256 寄存器）或 16 的倍数（AVX512）或 4 的倍数（NEON），且矩阵不会过小而存在内存对齐问题（可以增加判断，对于小矩阵直接使用朴素乘法）。时间关系，我们依然使用简单的按行列计算，以后可能考虑实现 kernel。

与之类似的，我们想到可以将循环展开：

```
/* Codes */
for (register size_t j = 0; j < col2; j += 8) {
    prod->m_data[i * col2 + j + 0] += s * mat2->m_data[k * col2 + j + 0];
    prod->m_data[i * col2 + j + 1] += s * mat2->m_data[k * col2 + j + 1];
    prod->m_data[i * col2 + j + 2] += s * mat2->m_data[k * col2 + j + 2];
    prod->m_data[i * col2 + j + 3] += s * mat2->m_data[k * col2 + j + 3];
    prod->m_data[i * col2 + j + 4] += s * mat2->m_data[k * col2 + j + 4];
    prod->m_data[i * col2 + j + 5] += s * mat2->m_data[k * col2 + j + 5];
    prod->m_data[i * col2 + j + 6] += s * mat2->m_data[k * col2 + j + 6];
    prod->m_data[i * col2 + j + 7] += s * mat2->m_data[k * col2 + j + 7];
}
/* Codes */
```

“循环展开最常用来降低循环开销，为具有多个功能单元的处理器提供指令级并行，也有利于指令流水线的调度”[9]。上面将循环展开 8 次，当然，我们可以继续增加循环展开次数以追求进一步提高程序运行速度，但是这个增加循环展开次数也是有限度的，“当达到了 CPU 的最高吞吐量之后，继续增加循环展开次数是没有意义的。”

但以上手动展开导致源代码膨胀，可读性下降。实际上我们可以通过编译指令完成等效操作：

```
#pragma UNROLL(8)
```

[8]由于我没有 ARM 架构的计算机，暂无法对代码进行验证及测速。
[9]C++ 性能榨汁机之循环展开 (zhuanlan.zhihu.com/p/37582101)

```
    for (register size_t j = 0; j < col2; j ++) {
        prod->m_data[i * col2 + j] += s * mat2->m_data[k * col2 + j];
    }
```

这是在不开启优化的情况下明确告诉编译器需要展开，而在开启了编译器优化选项的时候，编译器会自动对我们的循环代码进行循环展开（因此会将此类的展开指令覆盖，是否展开、展开规模由编译器权衡），让我们可以在保持了代码可读性的同时，又能享受到循环展开对我们程序性能的提升。

3.2.4 OpenMP 与多线程

这里简单的对 ikj 循环使用并行循环，对于三重 for 循环，下文的性能测试分别展示了 omp parallel 设置在不同循环的区别（即子任务粒度的区别）。

```
CMatrix *MatMulMP(ReadOnlyMat mat1, ReadOnlyMat mat2) {
/* Codes */
#pragma omp parallel for  // Case 1
    for (register size_t i = 0; i < row; i++) {
#pragma omp parallel for  // Case 2, Optional
        for (register size_t k = 0; k < col1; k++) {
            float s = mat1->m_data[i * mat1->m_col + k];
#pragma omp parallel for  // Case 3, Not recommanded
            for (register size_t j = 0; j < col2; j++) {
                prod->m_data[i * col2 + j] += s * mat2->m_data[k * col2 + j];
            }
        }
    }
    return prod;
}
```

OpenMP 强大且使用方便，与之相对的是传统的手动实现多线程。

```
typedef struct {
    ReadOnlyMat mat1, mat2;
    CMatrix *prod;
    size_t i, sz;
} Params;

void subMul(Params *p) {
    size_t i = p->i;
    size_t sz = p->sz;
    const float *data1 = p->mat1->m_data;
    const float *data2 = p->mat2->m_data;
    float *data3 = p->prod->m_data;
    for (register size_t k = 0; k < sz; k++) {
        float s = data1[i * sz + k];
        for (register size_t j = 0; j < sz; j++)
            data3[i * sz + j] += s * data2[k * sz + j];
    }
}

CMatrix *MatMulThreads(ReadOnlyMat mat1, ReadOnlyMat mat2) {
```

```
    CMatrix *m2trans = MatClone(mat2);
    MatTrans(&m2trans);

    size_t sz = mat1->m_row;
    CMatrix *prod = MatZeros(sz, sz);
    int threadNum = omp_get_max_threads();
    pthread_t *threads = (pthread_t *) malloc(threadNum * sizeof(pthread_t));
    for (int i = 0; i < sz / threadNum; ++i) {
        for (int t = 0; t < threadNum; ++t) {
            Params *arg = (Params *) malloc(sizeof(Params));
            arg->mat1 = mat1;
            arg->mat2 = mat2;
            arg->prod = prod;
            arg->sz = sz;
            arg->i = i * threadNum + t;
            pthread_create(&threads[i], NULL, subMul, arg);
        }
        for (int i = 0; i < threadNum; ++i)
            pthread_join(threads[i], NULL);
    }

    MatDelete(m2trans);
    return prod;
}
```

一方面，我们需要注意控制线程数量、另一方面，这里需要注意控制划分子任务的粒度大小。在 Project 2 中，我使用了更加方便的 ThreadPool，其帮我们管理的任务与线程的关系，理应效果优于手动实现，但当时使用多线程计算每个向量的点积，即对于 (n×m) 的结果矩阵，我们总共划分出 $n \times m$ 个子任务，这反而使每个子任务的管理耗时高于直接计算子任务的时间（类似于 inline 函数在时间与代码长度上的权衡不当），而本次 Project 我们粗粒度的划分任务，达到了较好的效果。

3.3 主函数及辅助函数

辅助函数（util.c / util.h）包括了对读文件的检查和计时器的宏定义。

```
// util.h

#ifndef MATMUL_UTIL_H
#define MATMUL_UTIL_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/errno.h>

#define MAX_LEN 40960

#ifdef _WIN32
#include <windows.h>

```

```
DWORD start, end;
#define TIMER_START start = GetTickCount();
#define TIMER_END end = GetTickCount();        \
                  printf("Time spent: %.5fs\n",\
                  (end - start) / 1000.f);
#else
#include <sys/time.h>

#define US_PER_SEC 1000000.f

struct timeval start, end;
#define TIMER_START gettimeofday(&start, NULL);
#define TIMER_END gettimeofday(&end, NULL);     \
                   printf("Time spent: %.5fs\n",\
                   end.tv_sec - start.tv_sec +  \
                  (end.tv_usec - start.tv_usec) / US_PER_SEC);
#endif

/**
 * General input file validity check
 */
FILE *OpenFile(const char *filename);

/**
 * The buffer (char*) has a max length and can be reused in many calls
 * here we must assume the length of a line is shorter than this
 *
 * @param f a txt file that contains a float matrix
 * @param buf generated by the caller, let the size be 40960
 * @return the column size of input
 */
size_t CheckCol(FILE *f, char *buf);

/**
 * Scan a line each time (assume the buffer is long enough)
 * until reaches EOF
 *
 * @param f a txt file that contains a float matrix
 * @param buf generated by the caller, let the size be 40960
 * @return the row size of input
 */
size_t CheckRow(FILE *, char *buf);

#endif //MATMUL_UTIL_H
```

初版代码我使用的是 clock() 来获取运行函数执行前后的时钟滴答数之差，但发现其在 OpenMP 和 sleep() 的测试下计时表现异常[10]：对于串行程序，可以用（单个 CPU 核心的）滴答数之差来计时，但对于

[10]C API (clock): clock time may advance faster or slower than the wall clock, depending on the execution resources given to the program by the operating system... if the current process is multithreaded and more than one execution core is available, clock time

多线程的程序，滴答数实际代表的是 CPU 时间，即所有线程的时间分别相加，因此观察到多线程计时与单线程耗时相近，故需选用其余方式实现计时器。有意思的是，我使用了 _WIN32 这个宏的状态来判断运行平台，对于 Unix/Linux/macOS 则使用 <sys/time.h> 的函数；若为 Windows 平台则采用 <windows.h> API。这种方式能正常且具有合适精度的完成需求。

```c
// util.c

#include "util.h"

FILE *OpenFile(const char *filename) {
    FILE *f = fopen(filename, "r");
    if (f == NULL) {
        printf("Failed to open <%s>: %s", filename, strerror(errno));
        exit(1);
    }
    return f;
}

size_t CheckCol(FILE *f, char *buf) {
    int col = 0;
    fscanf(f, "%[^\n]", buf);
    for (int i = 0; i < strlen(buf); i++)
        if (buf[i] != ' ' &&
            (buf[i + 1] == ' ' ||
             buf[i + 1] == '\0'))
            col++;
    rewind(f);
    return col;
}

size_t CheckRow(FILE *f, char *buf) {
    int row = 0;
    while (!feof(f)) {
        if (fgets(buf, MAX_LEN, f) != NULL)
            row++;
    }
    rewind(f);
    return row;
}
```

```c
// main.c

#include "cmatrix.h"
#include "cmat_mul.h"
#include "util.h"

```

may advance faster than wall clock.

```c
int main(int argc, const char **argv) {
    if (argc != 4) {
        printf("3 arguments expected (input 1, input 2, output), got %d\n", argc - 1);
        return -1;
    }
    char *in_buf = (char *) malloc(sizeof(char) * 40960);

    FILE *f1 = OpenFile(argv[1]);
    FILE *f2 = OpenFile(argv[2]);
    size_t r1 = CheckRow(f1, in_buf);
    size_t r2 = CheckRow(f2, in_buf);
    size_t c1 = CheckCol(f1, in_buf);
    size_t c2 = CheckCol(f2, in_buf);
    if (c1 != r2) {
        printf("The input two matrices cannot be multiplied: "
               "Mat 1 (%lu*%lu)\tMat 2 (%lu*%lu)\n",
               r1, c1, r2, c2);
        return -1;
    }

    CMatrix *mat1 = MatFromFile(r1, c1, f1);
    CMatrix *mat2 = MatFromFile(r2, c2, f2);
    printf("\n");

    TIMER_START
    CMatrix *prod = MatMul(mat1, mat2, Naive);
    TIMER_END

    MatRepr(prod);
    MatSave(prod, argv[3]);
    MatDelete(mat1);
    MatDelete(mat2);
    MatDelete(prod);
    return 0;
}
```

主函数包含了一套标准的从检测输入有效性（输入参数数量错误、输入文件读取异常会提示用户并退出，与 Project 2 一致）、矩阵乘法、打印结果并保存文件的流程。这里仅演示了其中一种矩阵乘法的调用，测试其余函数时需要修改 31 行的代码；且视函数要求，可能需要将 14 行的 mat2 按列主序读取存储。此外，结构体相关其余函数将在第四节另外演示。

4 测试样例及分析

```
Platform: CMake 3.20.4  OpenBLAS 0.3.15_1  OpenCV 3.4.15  OpenMP 13.0.0
Hardware: MacBook Pro 13, 2020 (2 GHz Quad-Core Intel Core i5, 16 GB 3733 MHz LPDDR4X)
$ cd /path/to/project/ && cmake . && make && ./matmul <3 args>
```

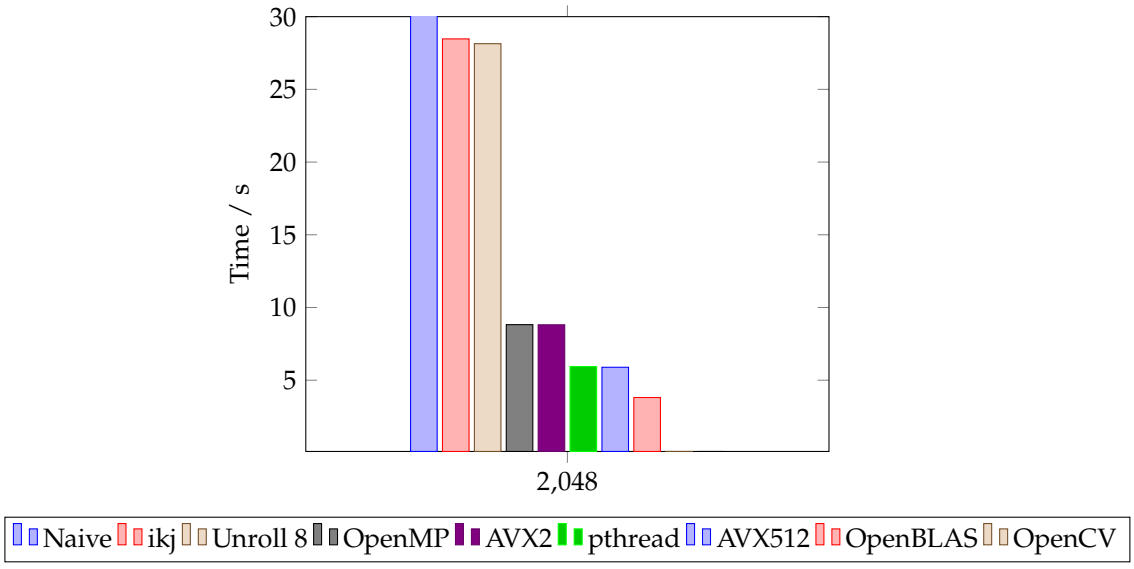
```
cmake_minimum_required(VERSION 3.20)
project(matmul C)

set(CMAKE_C_STANDARD 11)
set(CMAKE_C_COMPILER /usr/bin/gcc)

find_package(OpenMP REQUIRED)
if(NOT TARGET OpenMP::OpenMP_C)
    add_library(OpenMP_TARGET INTERFACE)
    add_library(OpenMP::OpenMP_C ALIAS OpenMP_TARGET)
    target_compile_options(OpenMP_TARGET INTERFACE ${OpenMP_C_FLAGS})
    find_package(Threads REQUIRED)
    target_link_libraries(OpenMP_TARGET INTERFACE Threads::Threads)
    target_link_libraries(OpenMP_TARGET INTERFACE ${OpenMP_C_FLAGS})
endif()

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -mavx -I/usr/local/opt/openblas/include")

set(LDFLAGS "-L/usr/local/opt/openblas/lib")
set(USE_BLAS OpenBLAS)
set(BLAS_LIBRARY /usr/local/opt/openblas/lib/libblas.dylib)

set(OpenCV_DIR /usr/local/Cellar/opencv@3/3.4.15/share/OpenCV)
find_package(OpenCV REQUIRED)
include_directories(/usr/local/Cellar/opencv@3/3.4.15/include)

set(SRC main1.c cmatrix.c util.c)
add_executable(matmul ${SRC})

target_link_libraries(matmul OpenMP::OpenMP_C)
target_link_libraries(matmul ${OpenCV_LIBS})
target_link_libraries(matmul ${BLAS_LIBRARY})
```

```
#include "cmatrix.h"
#include "cmat_mul.h"
#include "util.h"

//#define MODE OpenCv
#define MODE Blas

void benchmark(const char *f1, const char *f2, size_t sz, kMatMulMode m, int t) {
    CMatrix *ma = MatFromFile(sz, sz, OpenFile(f1));
    CMatrix *mb = MatFromFile(sz, sz, OpenFile(f2));
    if (m == Order) {
        MatMulChangeOrd(ma, mb, t);
        return;
    }
    CMatrix *ans = MatMul(ma, mb, m, t);
//    MatRepr(ans);
    MatSave(ans, filename: "out2048.txt");
    MatDelete(ans);
    MatDelete(ma);
    MatDelete(mb);
    printf("\n");
}

int main(int argc, const char **argv) {
    benchmark( f1: "mat-A-32.txt", f2: "mat-B-32.txt", sz: 32, m: MODE, t: 20);
    benchmark( f1: "mat-A-64.txt", f2: "mat-B-64.txt", sz: 64, m: MODE, t: 20);
    benchmark( f1: "mat-A-128.txt", f2: "mat-B-128.txt", sz: 128, m: MODE, t: 20);
    benchmark( f1: "mat-A-256.txt", f2: "mat-B-256.txt", sz: 256, m: MODE, t: 20);
    benchmark( f1: "mat-A-512.txt", f2: "mat-B-512.txt", sz: 512, m: MODE, t: 20);
    benchmark( f1: "mat-A-1024.txt", f2: "mat-B-1024.txt", sz: 1024, m: MODE, t: 20);
    benchmark( f1: "mat-A-1500.txt", f2: "mat-B-1500.txt", sz: 1500, m: MODE, t: 20);
    benchmark( f1: "mat-A-2048.txt", f2: "mat-B-2048.txt", sz: 2048, m: MODE, t: 20);
    benchmark( f1: "mat-A-2500.txt", f2: "mat-B-2500.txt", sz: 2500, m: MODE, t: 20);
    benchmark( f1: "mat-A-3000.txt", f2: "mat-B-3000.txt", sz: 3000, m: MODE, t: 20);
    benchmark( f1: "mat-A-3500.txt", f2: "mat-B-3500.txt", sz: 3500, m: MODE, t: 20);
    benchmark( f1: "mat-A-4096.txt", f2: "mat-B-4096.txt", sz: 4096, m: MODE, t: 20);
```

4.1 矩阵乘法

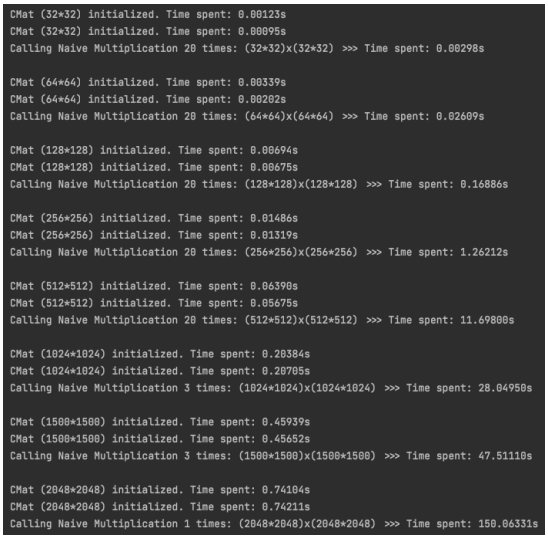
本次计算出所有结果均通过了 Project 2 中的 Python 脚本的正确性验证（将计算结果与 NumPy 结果对比，随机数据 RMSE≤1，官方样例 RMSE≤10，与 Project 2 结论保持一致，可认为程序无逻辑错误，仅存在 float 误差），不重复附图。

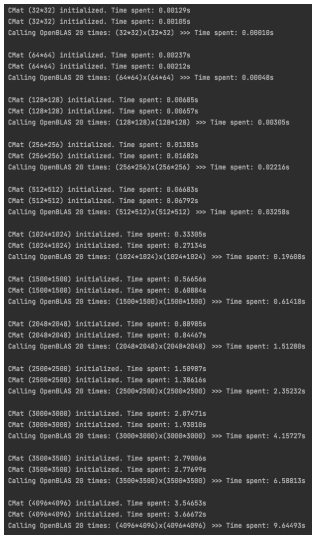
Project 2 中我们已经见识到 g++/gcc 的 O3 优化角度之多、提升之大（且对于 OpenBLAS、OpenMP 等已经较为完善的库函数所能提升的比例不如对较为简单的源码的比例大），故省略此项对比。

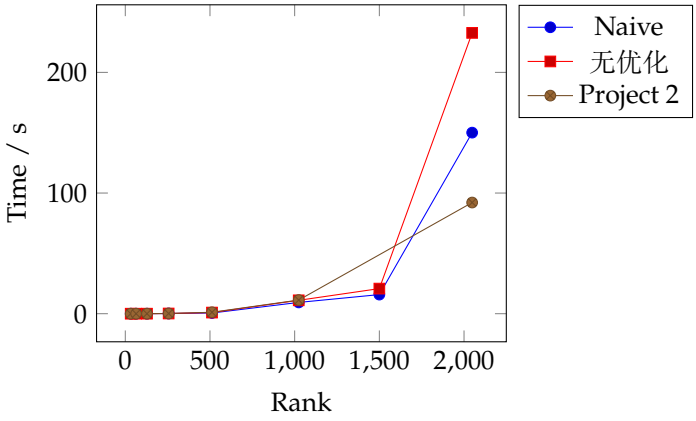


2048 阶矩阵乘法性能横向对比

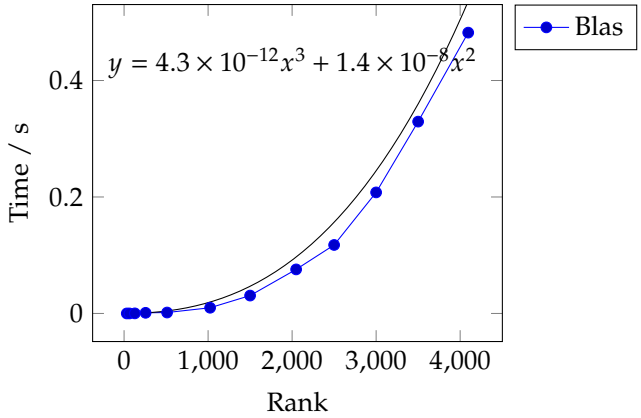
4.1.1 基准：OpenBLAS 与朴素乘法







上图对比了 Project 2、Project 3（应用了第二节提及的三种优化方案）及无优化版本的朴素乘法耗时。确实能体现语法层面的区别导致计算机底层行为不同，在大矩阵运算是能体现较大的性能差异（无优化 2048：232.729s，语法优化使性能提升 30.31%）。但优化后的 C 版本依然慢于 C++ 版本，这可能是因为相较于本次使用的 gcc，g++ 更为高级会自动帮我们优化一些语言细节[11]，或者 C++ 从语言层面上不需要注意这些。



可以看出 OpenBLAS 耗时的增长趋势与朴素乘法一致，均为 $O(n^3)$，由此我们可以推出 OpenBLAS 也是使用基础的乘法，只是在各方面都做了优化（以下图为例，可以看出 OpenBLAS 使用了多线程），而

[11] 或因为 C 更接近底层，其语法更直接的影响了计算机的行为。

并非凭借在算法上的复杂度优势，通过节约计算开销（无效缓存、内存对齐、分页等）及充分利用计算资源（多线程等）得以将整体效率提升。另外，Project 2 中我用 OpenBLAS 计算向量点乘来分布计算每个元素的值，由于此方法与正常使用 OpenBLAS 的方法想去甚远（甚至先创建矩阵的一维数据副本，再使用 sgemm 的总耗时依然远小于此），其时间相对于使用 sgemm 多出数百倍，无对比价值，但 RMSE = 0.256，小于 sgemm 算出的 RMSE = 0.549，即 sgemm 可能将整个矩阵的运算中的 float 误差积累了下来。

```
PID    COMMAND      %CPU  TIME     #TH   #WQ  #PORTS MEM    PURG   CMPRS  PGRP  PPID  STATE
50515  matmul       652.6 00:44.48 8/8   0    17     91M    0B     0B     50515 456   running
```

如表所示，OpenBLAS 的性能几乎是朴素乘法的 50-100 倍

Time(s) \Rank	32	64	128	256	512	1024	2048
Naive (C, Optimized)	0.00026	0.0021	0.021	0.17	1.20	11.32	92.13
Blas	0.000005	0.000024	0.00015	0.0011	0.0016	0.0098	0.075
Blas / Naive (%)	1.92	1.14	0.71	0.64	13.33	8.66	8.14

4.1.2 调换顺序

“在内存使用上，程序访问的内存地址之间连续性越好，程序的访问效率就越高；相应地，程序访问的内存地址之间连续性越差。所以，我们应该尽量在行优先机制的编译器，比如 C/C++，CUDA 等等上，采用行优先的数据存储方式。”[12]因此我们有两种选择：一是直接遵循线性代数的运算法则，这可以通过调换乘法顺序来提高缓存命中率；或者在读矩阵时直接将右矩阵转置（使用 MatFromFileTrans 函数）存储，这样原矩阵的 “列” 就变成了内存中连续存储的 “行”，等效的可以依然将此函数规模认为是原规模，但在数据区 “转置”，也即*列主序*。理论分析后两种方法的跳转数与使用 *ikj* 顺序乘法的跳转数一致[13]。由下表可以看出，在计算次数（即 CPU 负荷）一致的时候，整体耗时与内存跳转次数呈正相关，即与缓存命中率呈负相关。

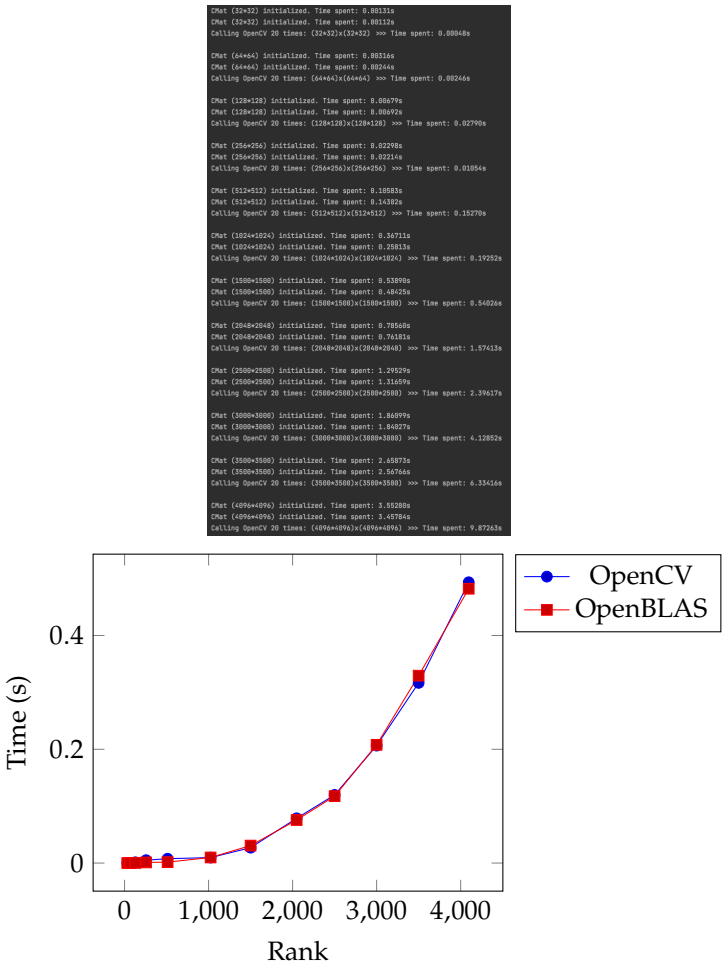
```
CMat (256*256) initialized. Time spent: 0.01292s
CMat (256*256) initialized. Time spent: 0.01311s
Calling [ikj] 20 times: (256*256)x(256*256) >>> Time spent: 1.18025s
Calling [kij] 20 times: (256*256)x(256*256) >>> Time spent: 1.15437s
Calling [jik] 20 times: (256*256)x(256*256) >>> Time spent: 1.40319s
Calling [kji] 20 times: (256*256)x(256*256) >>> Time spent: 2.75928s
Calling [jki] 20 times: (256*256)x(256*256) >>> Time spent: 2.78074s
CMat (512*512) initialized. Time spent: 0.10020s
CMat (512*512) initialized. Time spent: 0.08173s
Calling [ikj] 20 times: (512*512)x(512*512) >>> Time spent: 9.51450s
Calling [kij] 20 times: (512*512)x(512*512) >>> Time spent: 8.77461s
Calling [jik] 20 times: (512*512)x(512*512) >>> Time spent: 13.23500s
Calling [kji] 20 times: (512*512)x(512*512) >>> Time spent: 19.32561s
Calling [jki] 20 times: (512*512)x(512*512) >>> Time spent: 19.45552s
CMat (1024*1024) initialized. Time spent: 0.21771s
CMat (1024*1024) initialized. Time spent: 0.22117s
Calling [ikj] 3 times: (1024*1024)x(1024*1024) >>> Time spent: 10.84493s
Calling [kij] 3 times: (1024*1024)x(1024*1024) >>> Time spent: 10.19168s
Calling [jik] 3 times: (1024*1024)x(1024*1024) >>> Time spent: 26.51898s
Calling [kji] 3 times: (1024*1024)x(1024*1024) >>> Time spent: 80.18508s
Calling [jki] 3 times: (1024*1024)x(1024*1024) >>> Time spent: 91.62509s
CMat (1500*1500) initialized. Time spent: 0.50452s
CMat (1500*1500) initialized. Time spent: 0.50146s
Calling [ikj] 3 times: (1500*1500)x(1500*1500) >>> Time spent: 34.39339s
Calling [kij] 3 times: (1500*1500)x(1500*1500) >>> Time spent: 32.55033s
Calling [jik] 3 times: (1500*1500)x(1500*1500) >>> Time spent: 38.01279s
Calling [kji] 3 times: (1500*1500)x(1500*1500) >>> Time spent: 84.48177s
Calling [jki] 3 times: (1500*1500)x(1500*1500) >>> Time spent: 81.22594s
```

[12]行优先和列优先的问题 (blog.csdn.net/Canhui_WANG/article/details/52242496)

[13]时间关系，在实现完列主序的存储和转置初始化矩阵的函数后，未对其进行测速。

计算顺序	jki	kji	ijk	jik	kij	ikj
理论跳转数	$2n^3+n^2$	$2n^3$	n^3+n^2-n	n^3+n^2+n	$2n^2$	n^2
1024 阶方阵耗时 (s)	91.625	80.185	21.664	26.519	10.192	10.845

4.1.3 OpenCV

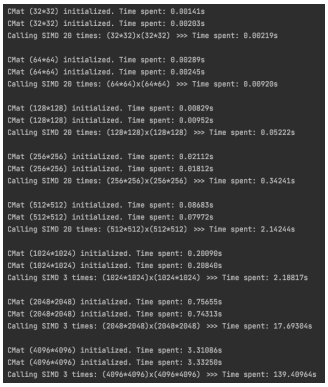


可见 OpenCV 与 OpenBLAS 的速度极为接近（考虑到电脑性能浮动，可认为两者等同）——在维持复杂度的情况下，两者均将优化做到了极致。经对比，两者结果的 RMSE 也极为接近。

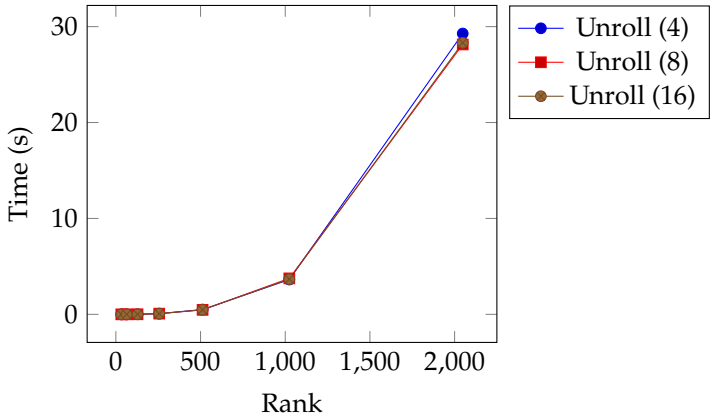
4.1.4 SIMD 与循环展开

Project 2 中仅用 SIMD 进行向量点乘，且每准备右矩阵的列向量均需要花费大量时间，本次先将右矩阵保存一份转置副本并在其上进行跳转数较少的运算。可以看到，本次使用 SIMD 的方式较为正确（且可以通过直接传入转置的右矩阵进一步提升效率），与 Project 2 中较为愚蠢！的调用方法相比性能提升近十倍。硬件限制，以下仅对 AVX2 测速。

不出所料的，利用数据向量化一次计算 8 个数据，我们将朴素乘法的性能提升十倍左右。与此同理，AVX512 又能将 AVX2 的性能翻倍。



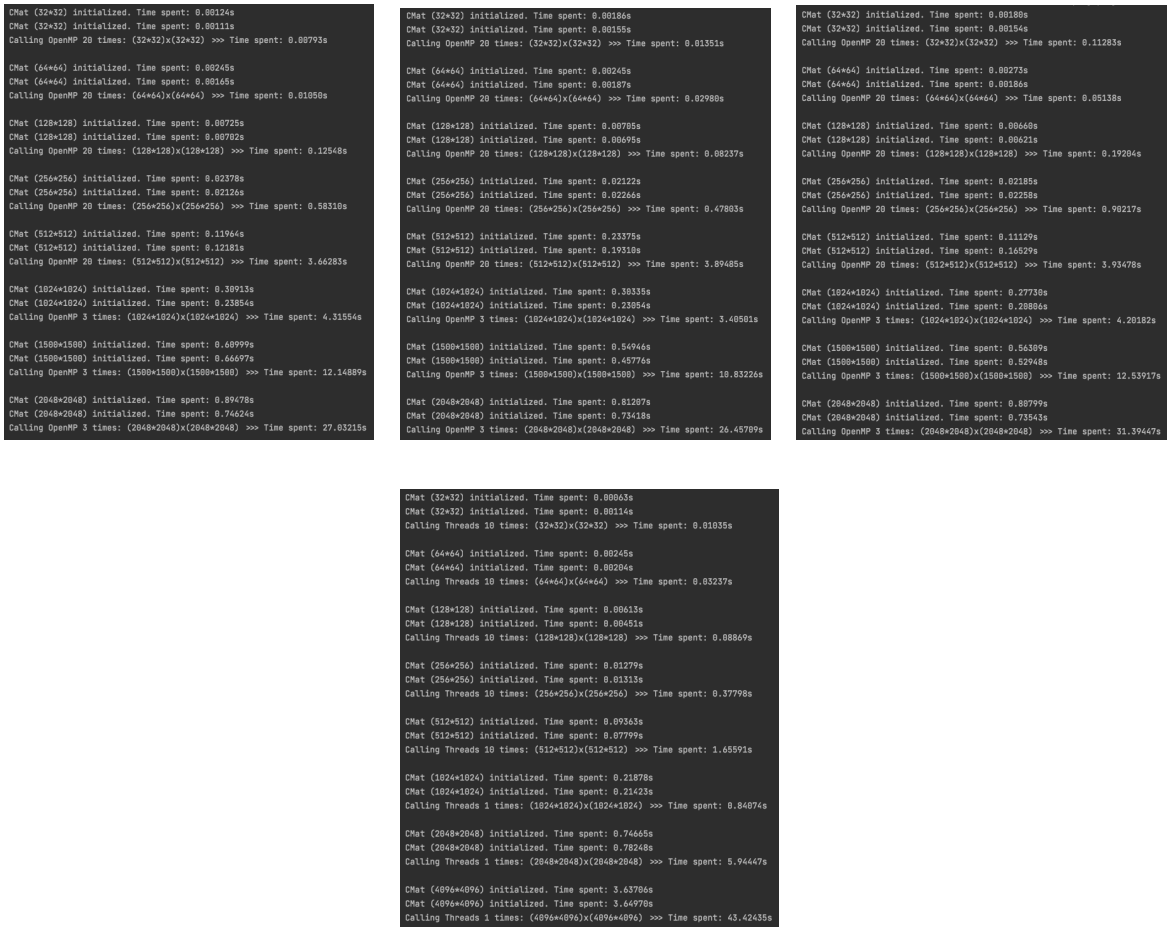
Time(s) \Rank	32	64	128	256	512	1024	2048	4096
Project 2	0.00082	0.0042	0.028	0.16	1.19	11.17	51.56	-
Project 3 (AVX2)	0.00011	0.00046	0.0026	0.017	0.11	0.73	5.89	46.47
Project 3 (AVX512)	0.000045	0.00021	0.0013	0.0081	0.053	0.44	3.67	29.18
Unroll（8）	0.00013	0.0010	0.0071	0.074	0.47	3.75	28.14	-
Naive (C, Optimized)	0.00026	0.0021	0.021	0.17	1.20	11.32	92.13	-



上图展示了使用编译命令展开循环的性能，展开 16 次相比起展开 8 次无明显区别，可能是上文提及的 CPU 吞吐量问题。

4.1.5 OpenMP 及多线程

虽然 OpenMP 强大且“智能”，与多线程同理，它并不可以滥用，“OpenMP 和多线程能否加速要看任务的粒度，粗粒度有利于加速，细粒度甚至会减速。”下图中（右）将三个 for 循环均标识为 omp parallel for，耗时反而较使用两个外层 parallel for（中）和最外层 parallel for（左）的多。



从上表我们注意到，使用 pthread 实现的粗粒度矩阵乘法反而比 OpenMP 快，可能是 OpenMP 自带了互斥锁或者我对其了解不够深而导致没写好。

4.2 部分相关函数

4.2.1 构造函数、随机矩阵及 MatRepr

演示生成全零矩阵、全一矩阵、单位矩阵、对角矩阵与随机生成指定范围元素的矩阵。

以下“输入”指演示该函数时在 main 函数里调用其的主要代码（省略 MatDelete 等）。构造空白矩阵函数被大量复用，以下函数无异常可验证 MatEmpty 无异常；多次运行程序后查看电脑内存消耗，发现无明显增加，即可表明 MatDelete 工作正常。

```
>>> CMatrix *m1 = MatZeros(3, 5);
CMatrix (dtype: float, size: 3*5)
[[0.0000  0.0000  0.0000  0.0000  0.0000],
 [0.0000  0.0000  0.0000  0.0000  0.0000],
 [0.0000  0.0000  0.0000  0.0000  0.0000]]
```

```
>>> CMatrix *m3 = MatEye(3);
CMatrix (dtype: float, size: 3*3)
[[1.0000  0.0000  0.0000],
 [0.0000  1.0000  0.0000],
 [0.0000  0.0000  1.0000]]
```

```
>>> CMatrix *m2 = MatOnes(7, 4);
CMatrix (dtype: float, size: 7*4)
[[1.0000  1.0000  1.0000  1.0000],
 [1.0000  1.0000  1.0000  1.0000],
 [1.0000  1.0000  1.0000  1.0000],
 ...
 [1.0000  1.0000  1.0000  1.0000],
 [1.0000  1.0000  1.0000  1.0000],
 [1.0000  1.0000  1.0000  1.0000]]
```

```
>>> float elem[] = {1.f, 2.f, 3.f, 4.f, 5.f};
>>> CMatrix *m4 = MatDiag(5, elem);
CMatrix (dtype: float, size: 5*5)
[[1.0000  0.0000  0.0000  0.0000  0.0000],
 [0.0000  2.0000  0.0000  0.0000  0.0000],
 [0.0000  0.0000  3.0000  0.0000  0.0000],
 [0.0000  0.0000  0.0000  4.0000  0.0000],
 [0.0000  0.0000  0.0000  0.0000  5.0000]]
```

```
>>> CMatrix *m5 = MatRand(4, 3, -5, 5);
CMatrix (dtype: float, size: 4*3)
[[-3.3193  2.5249  0.0073],
 [-0.6342  -4.1070  -3.8728],
 [-2.2456  0.0878  2.7923],
 [-1.2291  1.4440  -1.1835]]
```

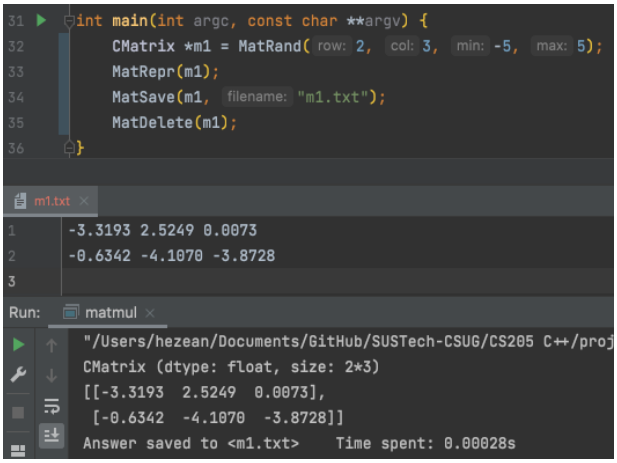
4.2.2 矩阵拷贝

一种理解是根据已有矩阵生成一个副本（MatClone）；另一种理解是将一个矩阵的数据覆盖地拷贝到另一个矩阵，下面演示了目标矩阵与原矩阵相比 ① 同行同列或总元素数量一致时直接拷贝；② 行数或列数比原矩阵小，可能丢失数据，禁止拷贝；③ 目标矩阵更大，将原矩阵填入左上角并将其余置零。

```c
int main(int argc, const char **argv) {
    CMatrix *m1 = MatRand( row: 2,  col: 3,  min: -5,  max: 5);
    CMatrix *m2 = MatClone(m1);
    MatRepr(m1);
    MatRepr(m2);

    CMatrix *m3 = MatEmpty( row: 2,  col: 3);
    CMatrix *m4 = MatEmpty( row: 3,  col: 2);
    CMatrix *m5 = MatEmpty( row: 3,  col: 4);
    CMatrix *m6 = MatEmpty( row: 2,  col: 2);
    if (MatCpy(m1, m3)) MatRepr(m3);
    if (MatCpy(m1, m4)) MatRepr(m4);
    if (MatCpy(m1, m5)) MatRepr(m5);
    if (MatCpy(m1, m6)) MatRepr(m6);

    MatDelete(m1);
    MatDelete(m2);
    MatDelete(m3);
    MatDelete(m4);
    MatDelete(m5);
    MatDelete(m6);
}
```

```
CMatrix (dtype: float, size: 2*3)
[[-3.3193  2.5249  0.0073],
 [-0.6342  -4.1070  -3.8728]]
CMatrix (dtype: float, size: 2*3)
[[-3.3193  2.5249  0.0073],
 [-0.6342  -4.1070  -3.8728]]
CMatrix (dtype: float, size: 2*3)
[[-3.3193  2.5249  0.0073],
 [-0.6342  -4.1070  -3.8728]]
Cannot copy matrix: src(2*3) → dst(3*2)
Copying matrix (2*3) → (3*4), the blanks are set zero
CMatrix (dtype: float, size: 3*4)
[[-3.3193  2.5249  0.0073  0.0000],
 [-0.6342  -4.1070  -3.8728  0.0000],
 [0.0000  0.0000  0.0000  0.0000]]
Cannot copy matrix: src(2*3) → dst(2*2)
```

4.2.3 保存至文件



4.2.4 转置矩阵及矩阵加减

```
>>> CMatrix *m1 = MatRand(2, 3, -5, 5);
>>> MatRepr(m1);
CMatrix (dtype: float, size: 2*3)
[[-3.3193  2.5249  0.0073],
 [-0.6342  -4.1070  -3.8728]]

>>> MatTrans(&m1);
>>> MatRepr(m1);
CMatrix (dtype: float, size: 3*2)
[[-3.3193  -0.6342],
 [2.5249  -4.1070],
 [0.0073  -3.8728]]
```

```
>>> CMatrix *m1 = MatOnes(3, 3);
>>> CMatrix *m2 = MatEye(3);
>>> MatAdd(m1, m2);
>>> MatRepr(m2);
CMatrix (dtype: float, size: 3*3)
[[2.0000  1.0000  1.0000],
 [1.0000  2.0000  1.0000],
 [1.0000  1.0000  2.0000]]

>>> CMatrix *m3 = MatEye(4);
>>> MatAdd(m1, m3);
MathErr: Mat(3*3) + Mat(4,4) is invalid
```

5 困难及解决

5.1 面向过程编程

与面向对象编程最大的区别在于，C 语言不支持定义类，也自然不支持将一些函数封装为成员函数的做法。这主要影响了编写代码时的优雅性，但在参考 NASA C Style Guide 和 UMD C Style Guidelines 后，我通过 *StructNameFunctionName()* 的命名规范来解决此问题。但这依然意味着“对象”数据难以真正封装保护起来，在本项目中，我选择使用 Getter/Setter 访问矩阵元素（另外带来了较为方便地访问行主序与列主序的元素的优势）。

5.2 内联函数的使用

最初我尝试将函数的 inline 声明写在头文件及源文件中，但出现链接错误。经查阅[14]：

[14]Why are C++ inline functions in the header? (stackoverflow.com/questions/5057021/why-are-c-inline-functions-in-the-header)

"The definition of an inline function doesn't have to be in a header file but, because of the one definition rule (ODR) for inline functions, an identical definition for the function must exist in every translation unit that uses it."

"The easiest way to achieve this is by putting the definition in a header file."

考虑到需要内联的函数较为简短，确实可以放入头文件，故本项目用此方法解决。此外还有如 *static / extern inline*[15]等知识点我尚未研究清楚，或许在后续的课程中我们会更加深入的学习此方面的内容。

5.3 数组内存分配

在上一次项目中，我“为了防止无法申请超长数组导致异常”而使用二维数组存储矩阵数据，且这会造成缓存命中率更低及增加解引用的时间开销。但查阅资料发现 malloc/new 分配的内存实际上是*在虚拟内存（或逻辑地址，由 OS 管理）中连续，而在物理地址上可能是不连续的*[16]，另一方面，malloc 分配的堆区内存往往有数 GB 大小，故大部分情况下没有必要担心无法分配内存的错误。因此本项目可以用一维数组保存数据，并通过函数计算二维位置映射的下标来访问数据。

5.4 内存管理

C 语言没有类似 Java 通过引用计数管理的垃圾收集机制，也不能像上次项目利用类的析构函数帮助我们维护堆内存，因此，在本次项目中必须手动为每个 malloc 对应一个 free 来避免内存泄露。但也正因为在本程序中只将结构体放入堆内存中（栈内存式写法即不方便，也无优势），我们减少创建结构体变量并小心注意对应将其释放即可，一种简单可行的方法是在写下一个 malloc 时对应的马上写下 free（即对于每处构造的 MatEmpty，尤其是仅在函数局部中使用的，切记在合适位置 MatDelete），后续将其放到合适位置即可。

5.5 条件编译

如 3.2 节所提，为了实现跨平台的计时器，我使用 #ifdef 以在不同平台上调用不同 API，类似的，我们也可以用此使指令集加速跨平台[17]。

6 总结

首次写 C 语言，便能感受到其与 C++ 的巨大差别：C++ 中许多经常使用的函数在 C 中找不到对应；切实体会到面向过程远比不上面向对象的优雅等等。继上次 Project 给我带来的多线程及硬件优化这种计算机科技所能带来的巨大提升的震撼，我本次 Project 中了解到的语言层面的差别也能带来显著提升，可见这种接近底层的语言的强大之处。另外，我也在完成此项目中进一步加深了对指针、内存（堆区、栈区、分配与泄漏）等方面的理解与掌握[18]，也享受到了“把玩”OpenCV，OpenMP，OpenBLAS 等库的乐趣，见识计算机的强大。

[15] What's the difference between "static" and "static inline" function? (stackoverflow.com/questions/7762731/whats-the-difference-between-static-and-static-inline-function)

[16] Does malloc allocated fragmented chunks? (stackoverflow.com/questions/63077540/does-malloc-allocated-fragmented-chunks)

[17] 参考了 ShiqiYu@libfacedetection 的代码处理方式 (github.com/ShiqiYu/libfacedetection/blob/master/src/facedetectcnn.cpp)

[18] 下次 Project 能不要再做矩阵乘法吗 x 秃了