Project5 Report

前言

本次project需要优化矩阵乘法，使用Cblas中GEMM的相同函数输入，实现C<-alpha*A*B+beta*C的计算，并使其尽可能加速以达到cblas_dgemm()的速度。在测试时发现cblas_dgemm()对于大矩阵的优化做的非常好，在矩阵规模达到10^3数量级的行和列后仍然能在1秒内算出结果。为了追赶上cblas_dgemm()的速度，本项目使用C语言首先对矩阵乘法进行了循环优化，然后针对不同类型的乘法使用SIMD进行加速，最后再通过使用OpenMP和编译器优化等多种手段进行加速，在计算大矩阵的时间上勉强与cblas_dgemm()的时间控制在10倍至20倍左右。

项目介绍

矩阵乘法

由线性代数基本知识可以知道，矩阵乘法A(mxk)*B(kxn)有四种常见的理解方式：

1. 行乘行：结果C中的每一行是B中行向量按照A中行向量线性组合的结果。即：

a1 a2 x b1 b2 = [a1[b1,b2] + a2*[b3,b4]*

a3 a4 b3 b4 a3[b1,b2] + a4*[b3,b4]]*

2. 行乘列：使用A的每一个行向量和B的每一个列向量做点乘，点乘结果是C中的一个元素。（最常规的理解）
3. 列乘行：A的列向量与B的行向量相乘，得到k个子矩阵，将这些矩阵求和得到C矩阵。
4. 列乘列：结果C中的每一列是A中列向量按照B中列向量线性组合的结果。即：

a1 a2 x b1 b2 = [b1[a1 + b3*[a2 , b2*[a1 + b4*[a2*

a3 a4 b3 b4 a3] a4] a3] a4]]

在本项目中，由于输入参数中包含了A和B的转置情况，为了保证内存读取的连续性，使用不同的矩阵运算方法可能有更好的效果。因此本项目实现了这四种矩阵运算方法，并分析了速度快慢。

获取程序运行时间

本项目使用linux系统自带的<sys/time.h>获取时间。这个函数可以获取当前的时区时间，并使用自定义的print_time_diff计算并打印两次时间的差值。

```
#include "sys/time.h"
struct timeval begin, end1;
gettimeofday(&begin, NULL);
/*
矩阵乘法
*/
gettimeofday(&end1, NULL);
print_time_diff(&begin, &end1);

void print_time_diff(struct timeval *begin, struct timeval *end)
{
    printf("%lf\n", (end->tv_sec - begin->tv_sec) + (double)(end->tv_usec - begin->tv_usec) / 1000000);
}
```

至于为什么不用<time.h>是由于在项目后期使用OpenMP并行计算时，<time.h>中的clock()方法会把所有并行系统所用的时间累加，导致时间测量不准确。

误差计算

在验证结果正确性中，我使用cal_error()方法来计算通过本项目的方法计算出来的矩阵C与用cblas_dgemm()计算的结果的差值。具体方法为计算两个结果矩阵的差值求和，并与cblas_dgemm()中矩阵求和的结果相除，得到错误率。

```
// 计算结果的误差
void cal_error(const double *C, const double *C_copy, size_t l)
{
    double calculate_error = 0.0;
    double sum = 0.0;
    for (int i = 0; i < l; i++)
    {
        calculate_error += abs(C[i] - C_copy[i]);
        sum += abs(C_copy[i]);
    }
    printf("calculate error is: %lf%\n", calculate_error / sum * 100);
}
```

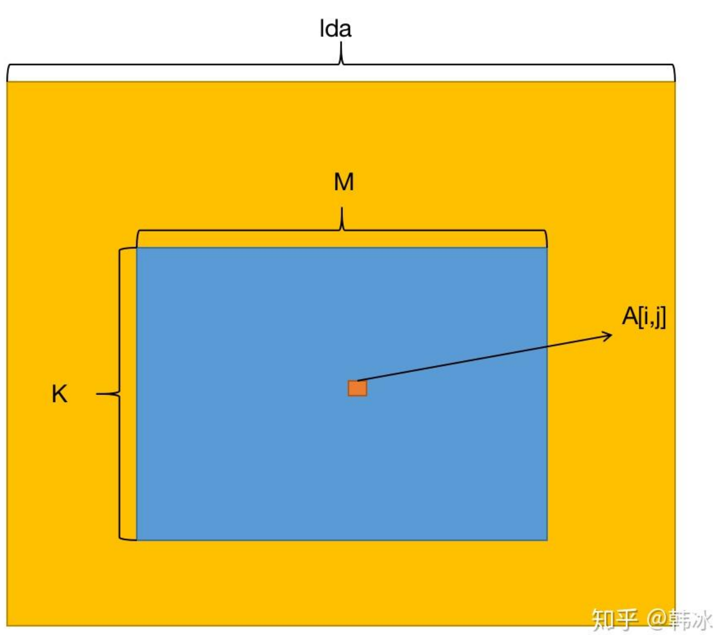
矩阵乘法及优化

参数解释

本项目的矩阵乘法方法与cblas_dgemm()所需输入参数相同

```
void cblas_dgemm(CBLAS_LAYOUT layout, CBLAS_TRANSPOSE TransA,
                 CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const double alpha, const double *A,
                 const int lda, const double *B, const int ldb,
                 const double beta, double *C, const int ldc);
```

- **CBLAS_LAYOUT layout：**选择行主序(CblasColMajor)还是列主序(CblasRowMajor)。
- **CBLAS_TRANSPOSE TransA, CBLAS_TRANSPOSE TransB：**分别控制A和B需要转置(CblasTrans)和不需要转置(CblasNoTrans)。
- **M,N,K：**假设我们要计算 A*B（A，B都为矩阵），则A为MxK的矩阵，B为KxN的矩阵。
- **alpha,beta,C：**C<-alpha*A*B+beta*C的对应参数，其中alpha, beta是常数，C是MxN矩阵。
- lda, ldb, ldc：表示矩阵到下一行需要跳过的元素个数。以lda为例进行解释，如果矩阵A是更大的矩阵（矩阵D）中的一部分，如下图所示（A是蓝色矩阵，D是黄色矩阵）：



如果此时要访问A[i, j]的元素，就需要`A [(i-1)*lda + j]`，其中lda>M，这便是lda存在的意义。

（图像和解释参考Blas矩阵乘法参数详解 - 知乎 (zhihu.com)）

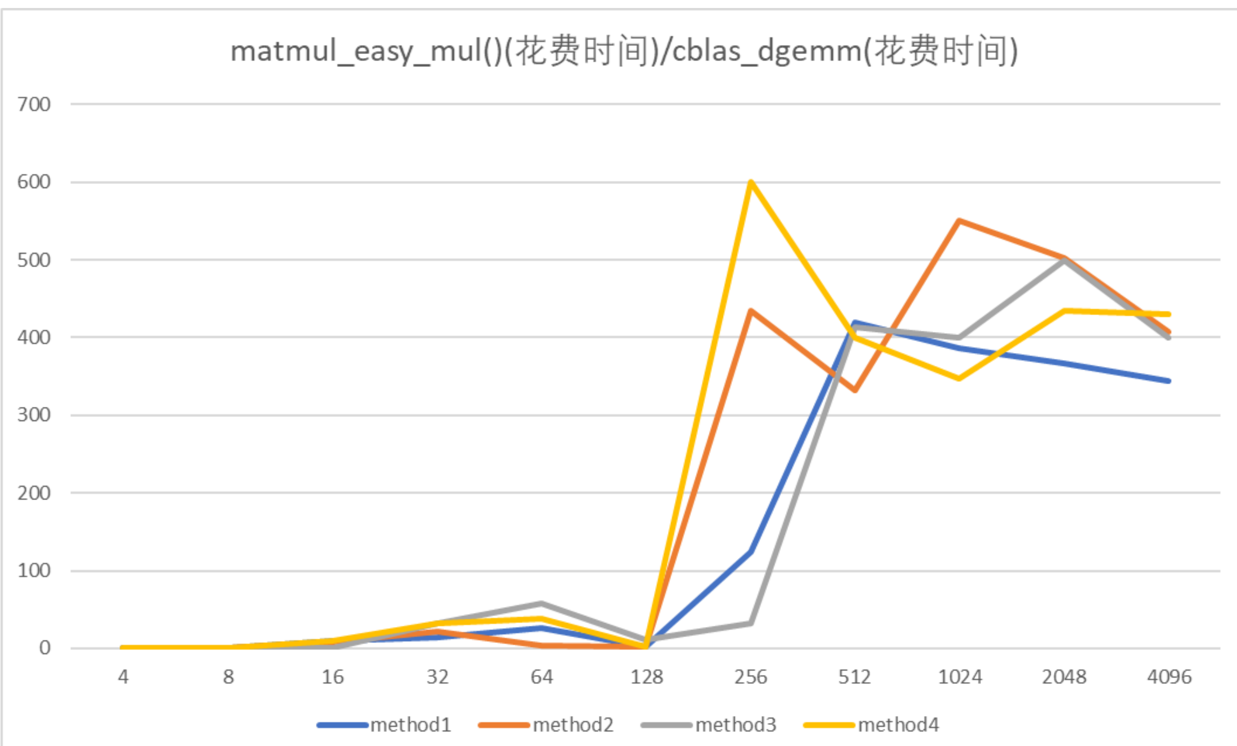
在实际使用该方法时，由于输入矩阵是A和B，实际结果仅有四种情况，即A*B,A*BT,AT*B,AT*BT，因此在后文讨论不同情况的矩阵输入时，**一般默认是CBLAS_LAYOUT layout按照行主序，分TransA和TransB的四种情况讨论。**

矩阵乘法——最简单实现

在项目中，我定义了*matrix_easy_mul()*方法实现了最普通的矩阵乘法。乘法的思路即先判断A和B的转置情况，使用*Trans()*方法把A变成行排列，B变成列排列，然后使用行乘列的方式进行矩阵乘法。

```
double *Trans(double *A, int m, int n){
    double *temp = (double *)malloc(sizeof(double) * m * n);
    for (int i = 0; i < m; i++){
        for (int j = 0; j < n; j++){
            temp[j * m + i] = A[i * n + j];
        }
    }
    for (int i = 0; i < m * n; i++)
        A[i] = temp[i];
    free(temp);
    return A;
}
```

```
//矩阵乘法（行乘列）
    for (int i = 0; i < m; i++){
        for (int j = 0; j < n; j++){
            for (int l = 0; l < k; l++){
                sum[i * ldc + j] += A[i * lda + l] * B[j * ldb + l];
            }
        }
    }
```



- method1：行排列，A不转置，B不转置
- method2：行排列，A不转置，B转置

- method3：行排列，A转置，B不转置
- method4：行排列，A转置，B转置

由上面的图表可知：四种情况的时间差别不大，具体原因是method1和4相对于method2仅多了两次Trans()**（转置过去再转置回来）**，method3相对于method2多了四次Trans()，而Trans()方法仅涉及内存读取，因此速度较快，调用几次该方法对结果无太大影响。*matrix_easy_mul()*方法的速度在矩阵规模大于128*128之后就远小于cblas_dgemm()方法的计算速度了，且大矩阵的运行时间普遍差距在400倍左右。

矩阵乘法——不同情况的循环优化

由上面对于矩阵乘法的探讨可知，矩阵乘法有多种理解方式。那么对于用其他方式理解并实现的矩阵乘法，速度上是否会快于普通的行乘列的算法呢？
我定义了*matrix_mul()*方法，**为了让矩阵运算时读取的内存连续**以达到优化算法的最好效果，该方法对于四种不同情况的矩阵进行四种不同的处理方式。下面展示关键的计算方法并计算对应M = N = K = n时方阵的内存跳转的次数。

- 对于行输入，A不转置，B不转置的矩阵：使用行乘行的处理方法（method1）

sum内存跳转n次，B内存跳转n次，共内存跳转2n次

```
for (int i = 0; i < m; i++)
            {
                for (int l = 0; l < k; l++)
                {
                    for (int j = 0; j < n; j++)
                    {
                        sum[ldc * i + j] += A[i * lda + l] * B[l * ldb + j];
                    }
                }
            }
```

- 对于行输入，A不转置，B转置的矩阵：使用行乘列的处理方法（method2）**（与上面的简单方法一致）**

A内存跳转n次，B内存跳转n次，共内存跳转2n次

```
for (int i = 0; i < m; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    for (int l = 0; l < k; l++)
                    {
                        sum[i * ldc + j] += A[i * lda + l] * B[j * ldb + l];
                    }
                }
            }
```

- 对于行输入，A转置，B不转置的矩阵：使用列乘行的处理方法（method3）

sum内存跳转n次，B内存跳转n次，共内存跳转2n次

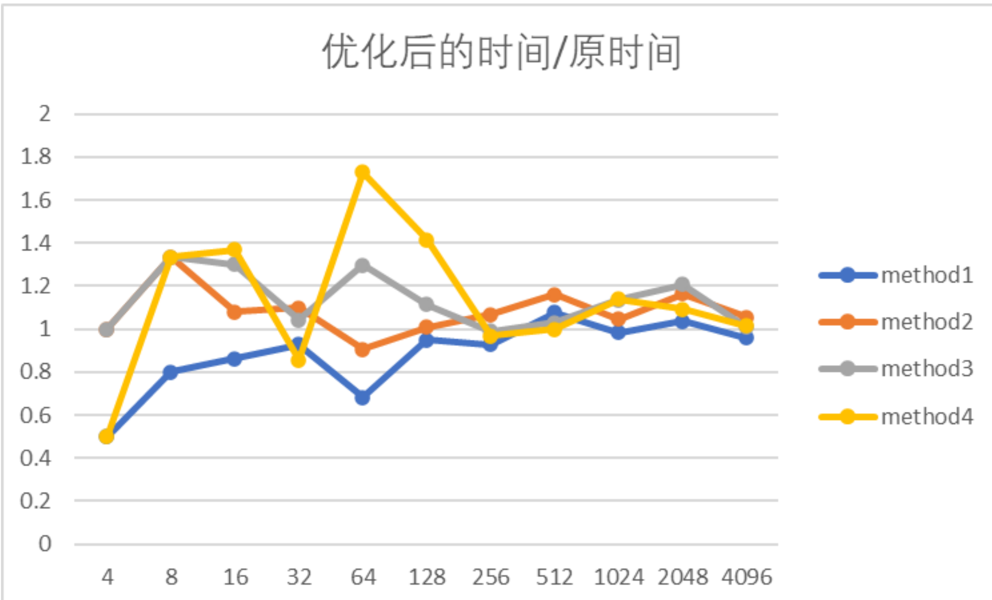
```
for (int l = 0; l < k; l++)
            {
                for (int i = 0; i < m; i++)
                {
                    for (int j = 0; j < n; j++)
                    {
                        sum[i * ldc + j] += A[l * lda + i] * B[j + ldb * l];
                    }
                }
            }
```

- 对于行输入，A转置，B转置的矩阵：使用列乘列的处理方法（method4）

sum内存跳转n次，A内存跳转n次，共内存跳转2n次

```
for (int j = 0; j < n; j++)
            {
                for (int l = 0; l < k; l++)
                {
                    for (int i = 0; i < m; i++)
                    {
                        sum[i + j * ldc] += A[l * lda + i] * B[j * ldb + l];
                    }
                }
            }
```

在使用了优化后的算法后，理论上四种方法的内存跳转次数一致，计算次数也一致，计算时间应该基本相同。



优化仅仅是省去了几次*Trans()*的时间，因此时间变化不大。但这些方法在接下来的SIMD优化中有很大作用。

矩阵乘法——SIMD加速

我定义了方法matrix_SIMD_mul作为更为快速的矩阵运算方法，使用SIMD的拓展语句其中对于四种不同类型矩阵分别进行了处理，加快矩阵乘法的速度。对于x86和Arm系统分别需要在matmul.c文件中解除注释，来开启对应的SIMD优化指令集。

```
// x86系统请解除注释以下代码
// #define Use_SIMD_mavx2
// Arm系统请解除注释以下代码
// #define Use_arm_NEON
```

- 对于method1和method4，二者都涉及向量乘以一个系数的方法，所以我定义了方法*matmul_SIMD_dotAdd_oneline()*使用SIMD加速向量乘以系数。该方法先定义了（n / 4）个__m256d类型向量数组用来储存结果，然后将A的第k个元素broadcast成__m256d向量，与B中第k行相乘。将对应的乘积累加，即可得到一条C中的行向量。在method1中输入为(A + lda * i, B, sum + n * i, m, n)计算行向量，method4中输入(B + k * j, A, sum + j * m, n, m)计算列向量。

```
#ifdef Use_SIMD_mavx2
    __m256d *result = (__m256d *)aligned_alloc(32, sizeof(__m256d) * (n / 4));
    for (int i = 0; i < (n / 4); i++){
        result[i] = _mm256_setzero_pd();
    }
    for (int i = 0; i < m; i++){
        for (int j = 0, l = 0; j < n - 3; j += 4, l++){
            result[l] = _mm256_add_pd(result[l], _mm256_mul_pd(_mm256_broadcast_sd(A + i), _mm256_loadu_pd(B + j + i * n)));
        }
    }
```

```
    for (int l = 0, j = 0; l < n / 4; l++, j += 4){
        _mm256_storeu_pd(sum + j, result[l]);
    }
    free(result);
#endif

matmul_SIMD_mul();
//method1
for (int i = 0; i < m; i++)
            {
                matmul_SIMD_dotAdd_oneline(A + lda * i, B, sum + n * i, m, n);
            }
//method4
for (int j = 0; j < n; j++)
            {
                matmul_SIMD_dotAdd_oneline(B + k * j, A, sum + j * m, n, m);
            }
```

- 对于method2行向量乘以列向量，我定义了*matmul_SIMD_dotAdd()*方法来使用SIMD计算两条向量的点乘。在将一组数据全部加起来后，通过水平加法把*__m256d*的四个数据加到**sum*里。

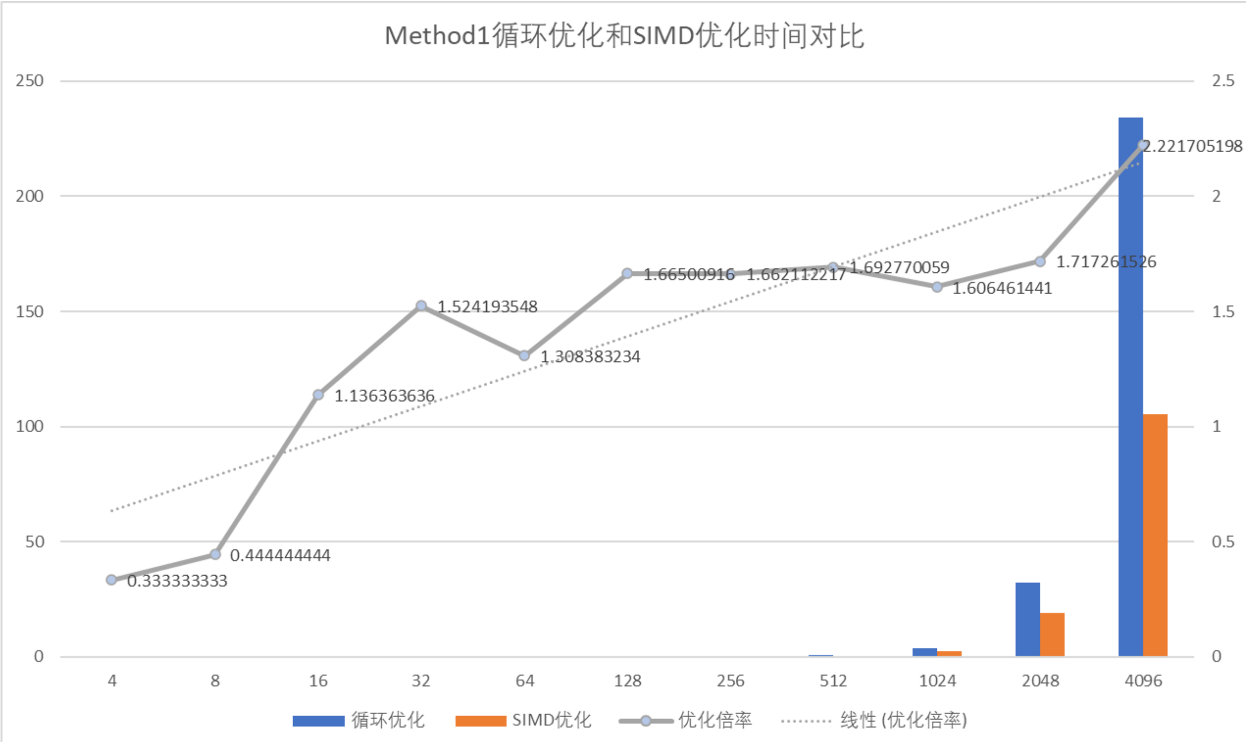
```
#ifdef Use_SIMD_mavx2
    __m256d result;
    result = _mm256_setzero_pd();
    for (int i = 0; i < k - 3; i += 4)
    {
        result = _mm256_add_pd(result, _mm256_mul_pd(_mm256_loadu_pd(A + i), _mm256_loadu_pd(B + i)));
    }
    __m256d sum256_hadd = _mm256_hadd_pd(result, result);
    __m256d sum256_hadd_permute = _mm256_permute2f128_pd(sum256_hadd, sum256_hadd, 1);
    __m256d sum256_hadd_permute2 = _mm256_hadd_pd(sum256_hadd_permute, sum256_hadd_permute);
    __m128d sum128 = _mm256_extractf128_pd(sum256_hadd_permute2, 0);
    sum128 = _mm_add_pd(sum128, _mm256_extractf128_pd(sum256_hadd_permute2, 1));
    *sum += _mm_cvtsd_f64(sum128);
    for (int i = 0; i < k % 4; i++)
        *sum += A[k - 1 - i] * B[k - 1 - i];
    *sum = *sum / 2.0;
#endif

matmul_SIMD_mul();
//method2
for (int i = 0; i < m; i++){
                for (int j = 0; j < n; j++){
                    matmul_SIMD_dotAdd(A + i * lda, B + j * ldb, sum + i * ldc + j, k);
                }
            }
```

- 对于method3列向量乘以行向量，我直接定义了方法matmul_SIMD_addDotToMatrix()使用SIMD计算列向量乘以行向量的结果。该方法定义了长度M*N/4的__m256d数组以储存每一处的结果，然后按照A矩阵的列和B矩阵的行进行遍历，将结果记录到不同的__m256d数组中，再把所有矩阵求和即得到结果矩阵。

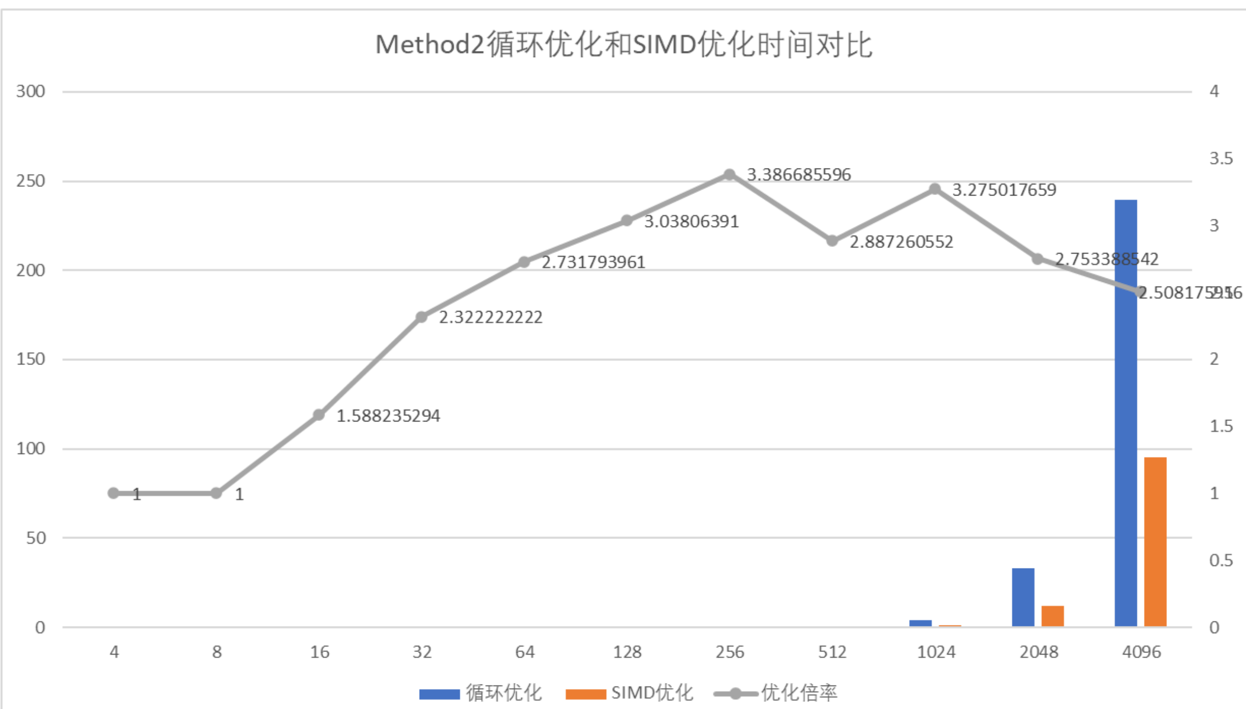
```
    #ifdef Use_SIMD_mavx2
    size_t moor = n / 4;  //__m256d of one row
    __m256d *result = (__m256d *)aligned_alloc(32, sizeof(__m256d) * m * moor);
    for(int i = 0; i < m * moor; i++){
        result[i] = _mm256_setzero_pd();
    }
    for(int l = 0; l < k; l++){
        for(int i = 0; i < m; i++){
            for(int j = 0, c = 0; j < n - 3; j += 4, c++){
                result[i * moor + c] = _mm256_add_pd(result[i * moor + c], _mm256_mul_pd(_mm256_broadcast_sd(A + i + l * m), _mm256_loadu_p
            }
        }
    }
    for(int i = 0, l = 0; i < m * n; i += 4, l++){
        _mm256_storeu_pd(sum + i, result[l]);
    }
    free(result);
    #endif

matmul_SIMD_mul();
// method3
matmul_SIMD_addDotToMatrix(A, B, sum, m, n, k);
```

下面是SIMD加速的结果：

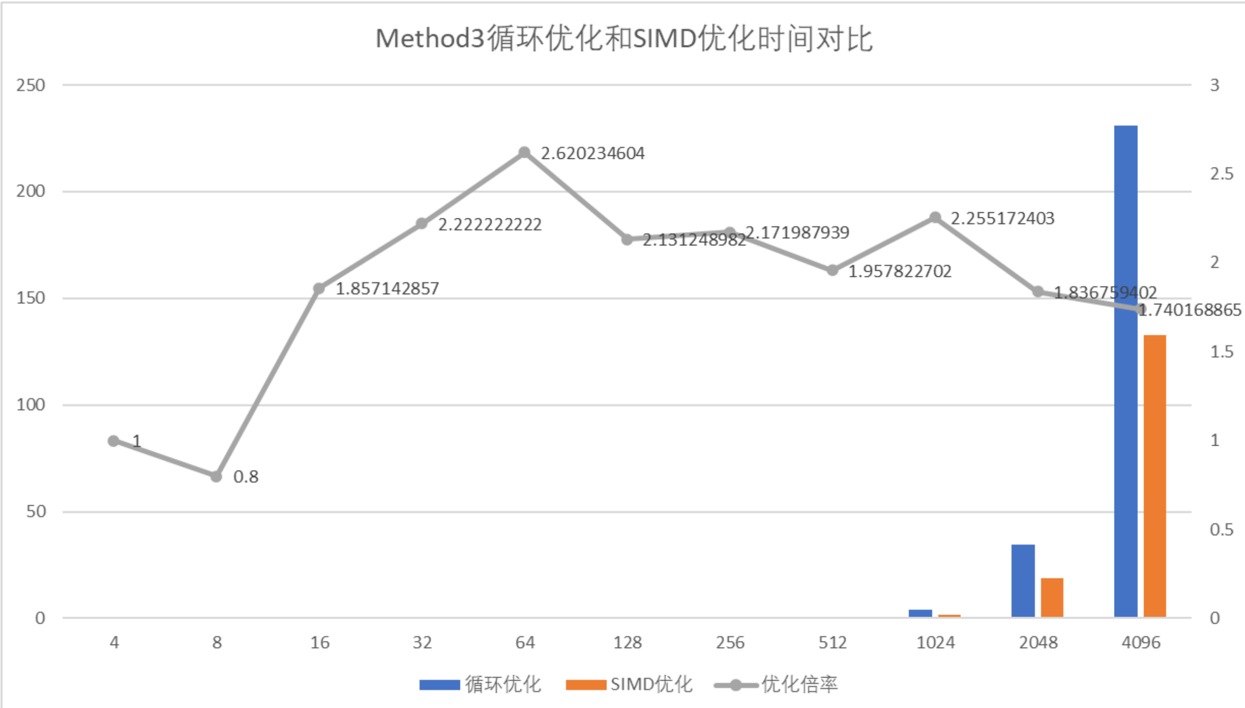
- Method1



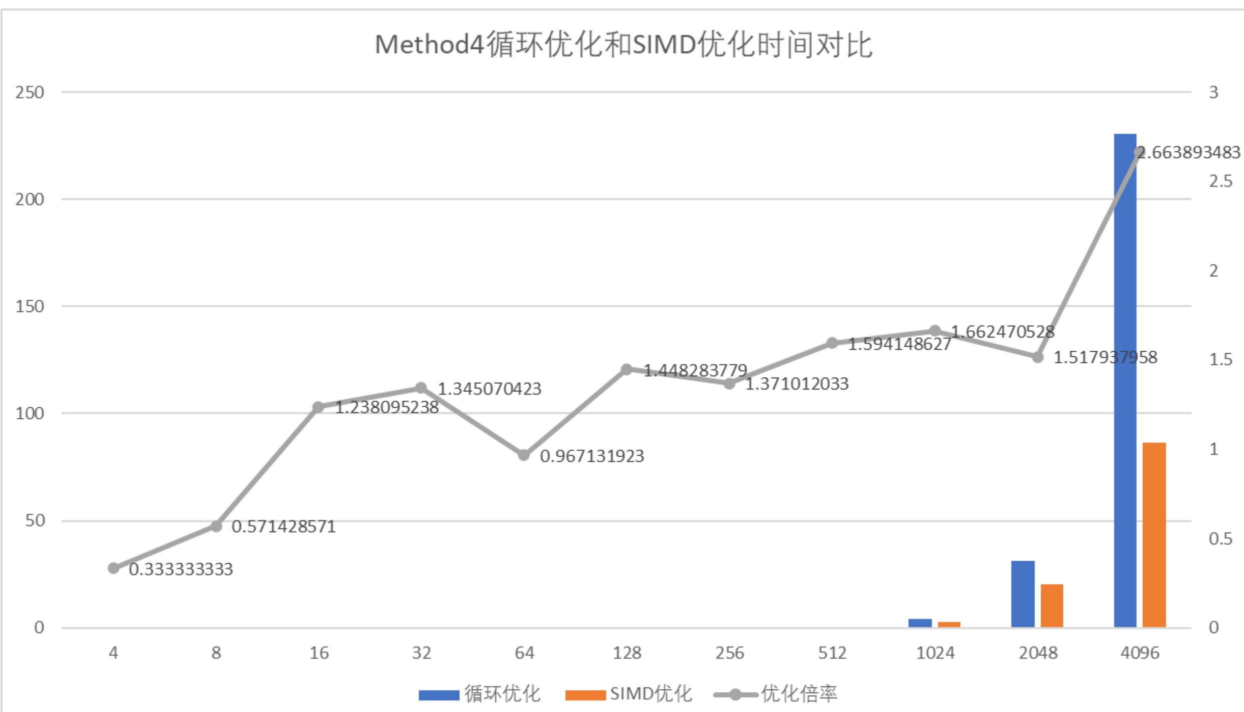
- Method2



- Method3



- Method4



通过上面的数据可以看到，在矩阵规模较小时优化效果不明显，但当矩阵规模较大的时候使用SIMD使得矩阵乘法速度提升了一倍多。method2对于矩阵计算加速明显，在大规模矩阵乘法是速度是原来的两至三倍。

矩阵乘法——OpenMP并行加速和-O3编译器优化

在上述SIMD加速算法的基础上，我为其中的循环进行了并行处理。在需要使用OpenMP进行加速时，需要将matmul.c中的宏定义的注释解除：

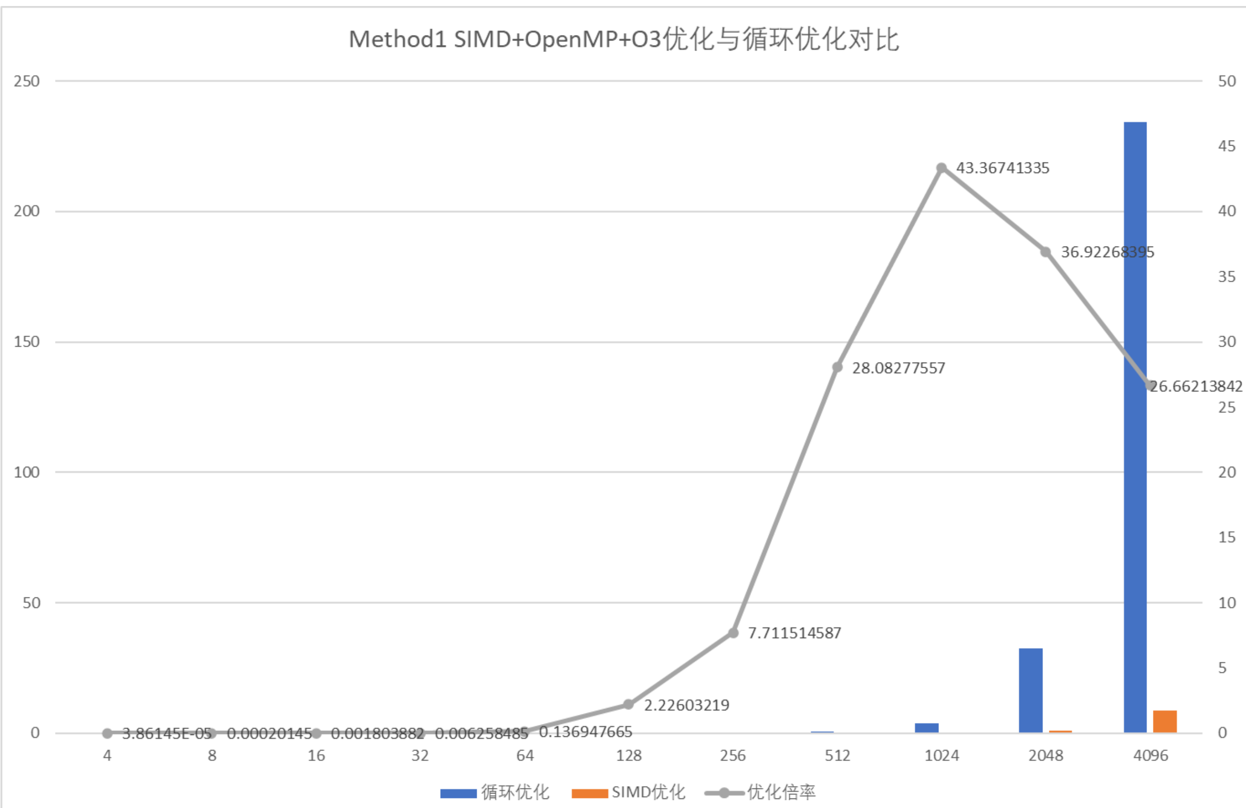
```
// 开启OpenMP请解除注释以下代码
 #define Use_openMP
```

并且在编译时加入了-O3编译器优化以进一步提升速度：

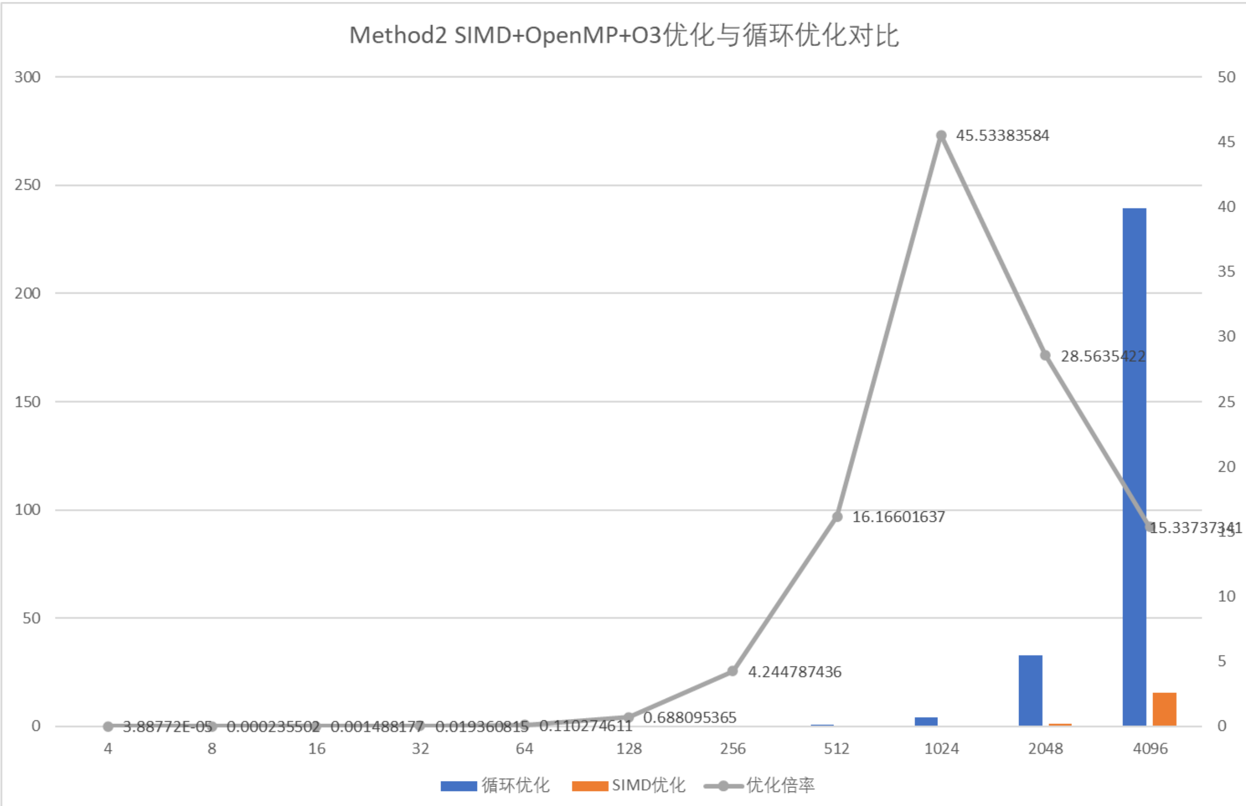
```
Makefile
main: main.c matmul.c
  gcc -o main main.c matmul.c -I /usr/include/ -L/usr/lib -lopenblas -lpthread -lgfortran -mavx2 -fopenmp -O3
```

在处理后，对比不同优化结果花费的时间和优化倍率：

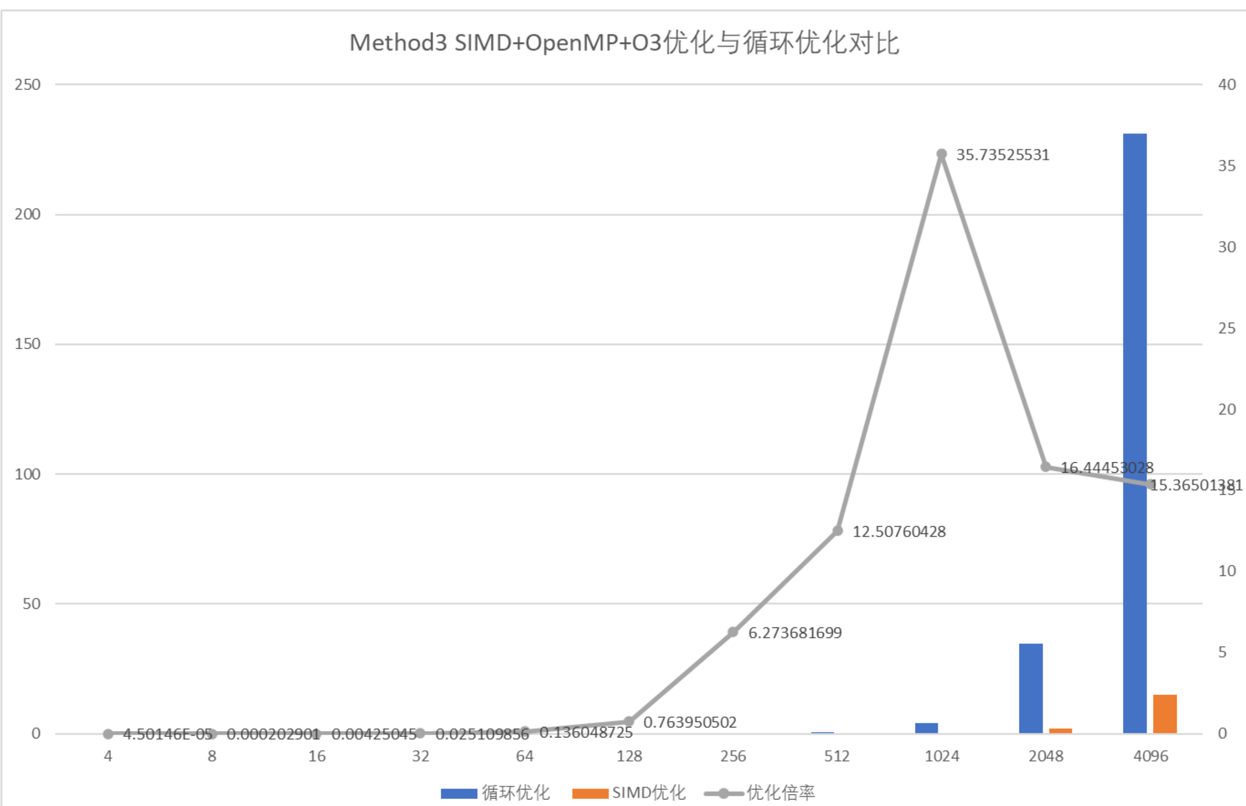
- method1



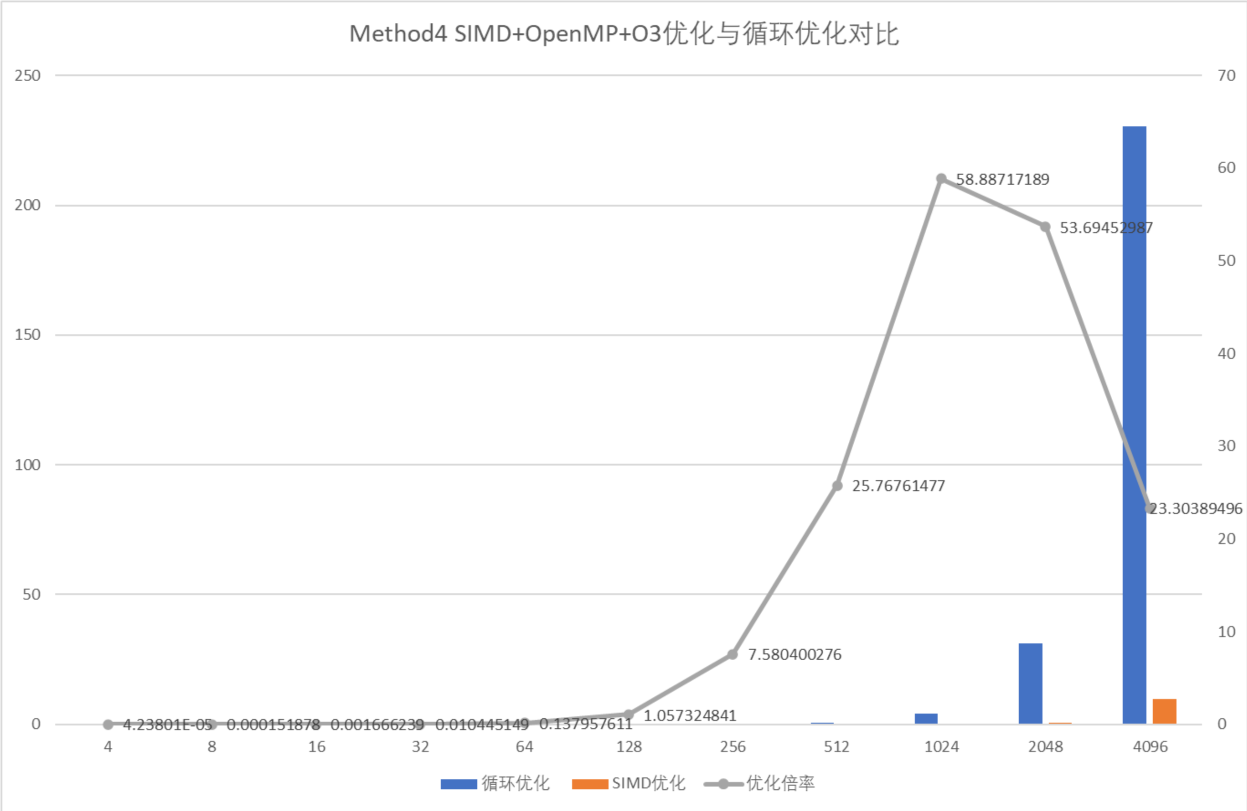
- method2


Method2 SIMD+OpenMP+O3优化与循环优化对比
300
250
200
150
100
50
0
50
45
40
35
30
25
20
15
10
5
0
3.88772E-05
0.000235502
0.001488177
0.019360815
0.110274611
0.688095365
4.244787436
16.16601637
45.53383584
28.5635422
15.33737341
4
8
16
32
64
128
256
512
1024
2048
4096
循环优化
SIMD优化
优化倍率

- method3


Method3 SIMD+OpenMP+O3优化与循环优化对比
250
200
150
100
50
0
40
35
30
25
20
15
10
5
0
4.50146E-05
0.000202901
0.00425045
0.025109856
0.136048725
0.763950502
6.273681699
12.50760428
35.73525531
16.44453028
15.36501381
4
8
16
32
64
128
256
512
1024
2048
4096
循环优化
SIMD优化
优化倍率

- method4



由上图可得，method1和method4在SIMD+OpenMP+O3优化下效率达到最高，在矩阵规模在1024^2和2048^2时达到了50倍，并在矩阵规模到达4096^2时达到20多倍，远高于method2和method3的优化方法。

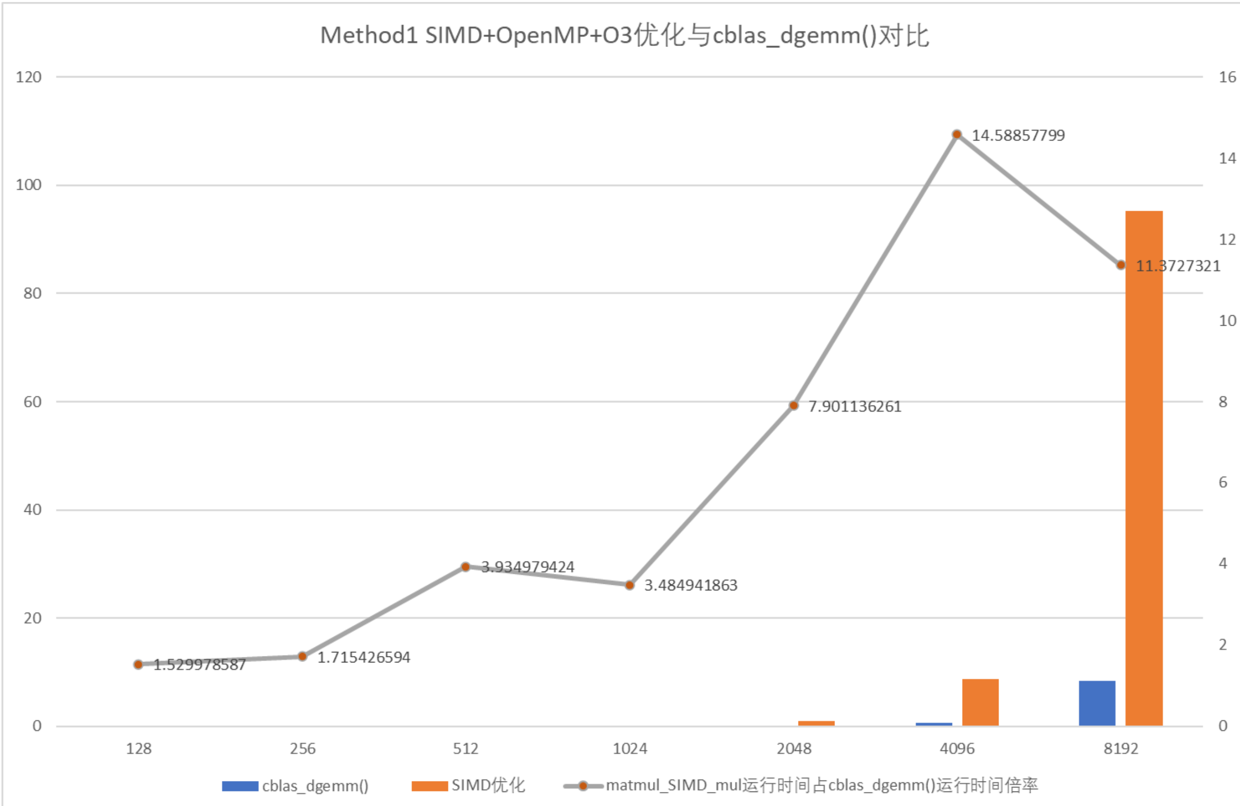
可以看到，在进行较大规模的矩阵乘法时，我的*matmul_SIMD_mul()*方法可以将时间控制在和cblas_dgemm()相差在一个数量级左右。

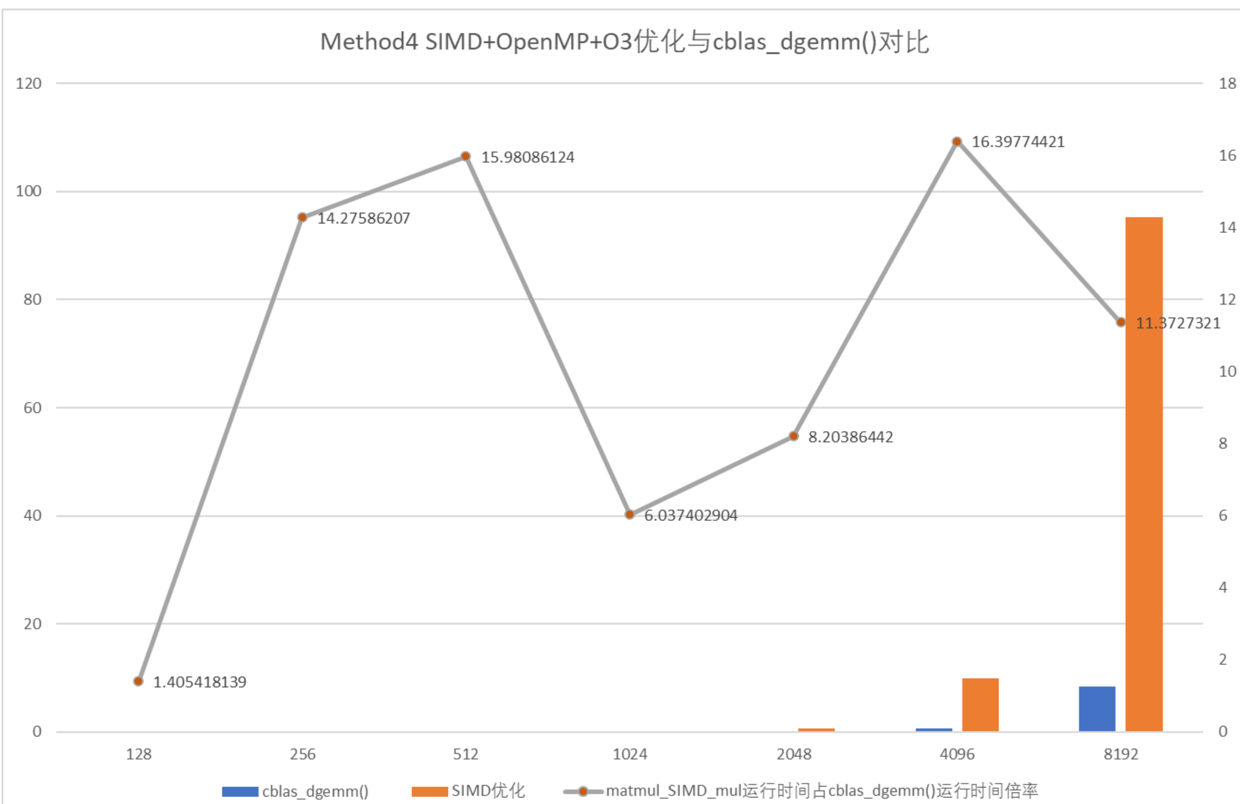
矩阵乘法——最终版

在上述测试过后，我发现使用SIMD+OpenMP+O3的对应method1和method4方法较快，method2方法虽然没有1和4好但相差不大，但是method3速度明显较慢。因此，为了使整体算法提升，我使用Trans()的方法先将method3转换成method4的情况，再用method4的情况求解。我定义了*matrix_SIMD_final_mul()*方法，这是本项目优化后的最快的方法。

```
void matrix_SIMD_final_mul(CBLAS_LAYOUT layout, CBLAS_TRANSPOSE TransA,
                           CBLAS_TRANSPOSE TransB, const int m, const int n,
                           const int k, const double alpha, const double *A,
                           const int lda, const double *B, const int ldb,
                           const double beta, double *C, const int ldc)
{
    if ((layout == CblasRowMajor && TransA == CblasNoTrans) || (layout == CblasColMajor && TransA == CblasTrans))
    {
        if (TransB == CblasNoTrans) // 行乘行
        {
            double *sum = (double *)malloc(sizeof(double) * m * n);
            for (int i = 0; i < m * n; i++)
                sum[i] = 0.0;
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
#endif
            for (int i = 0; i < m; i++)
            {
                matmul_SIMD_dotAdd_oneline(A + lda * i, B, sum + n * i, m, n);
            }
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
```

```
#endif
            for (int i = 0; i < m * n; i++)
                C[i] = alpha * sum[i] + beta * C[i];
            free(sum);
        }
        else // 行乘列
        {    // complete
            double *sum = (double *)malloc(sizeof(double) * m * n);
            for (int i = 0; i < m * n; i++)
                sum[i] = 0.0;
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
#endif
            for (int i = 0; i < m; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    matmul_SIMD_dotAdd(A + i * lda, B + j * ldb, sum + i * ldc + j, k);
                }
            }
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
#endif
            for (int i = 0; i < m * n; i++)
                C[i] = alpha * sum[i] + beta * C[i];
            free(sum);
        }
    }
    // 行排列转置后直接矩阵乘法
    else
    {
        if(TransB == CblasNoTrans) Trans(B, k ,n);
        double *sum = (double *)malloc(sizeof(double) * m * n);
        for (int i = 0; i < m * n; i++)
            sum[i] = 0.0;
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
#endif
        for (int j = 0; j < n; j++)
        {
            matmul_SIMD_dotAdd_oneline(B + k * j, A, sum + j * m, n, m);
        }
#ifdef Use_openMP
#pragma omp parallel for schedule(dynamic)
#endif
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                C[i * n + j] = alpha * sum[j * n + i] + beta * C[i * n + j];
            }
        }
        free(sum);
        if(TransB == CblasNoTrans) Trans(B, k ,n);
    }
}
```

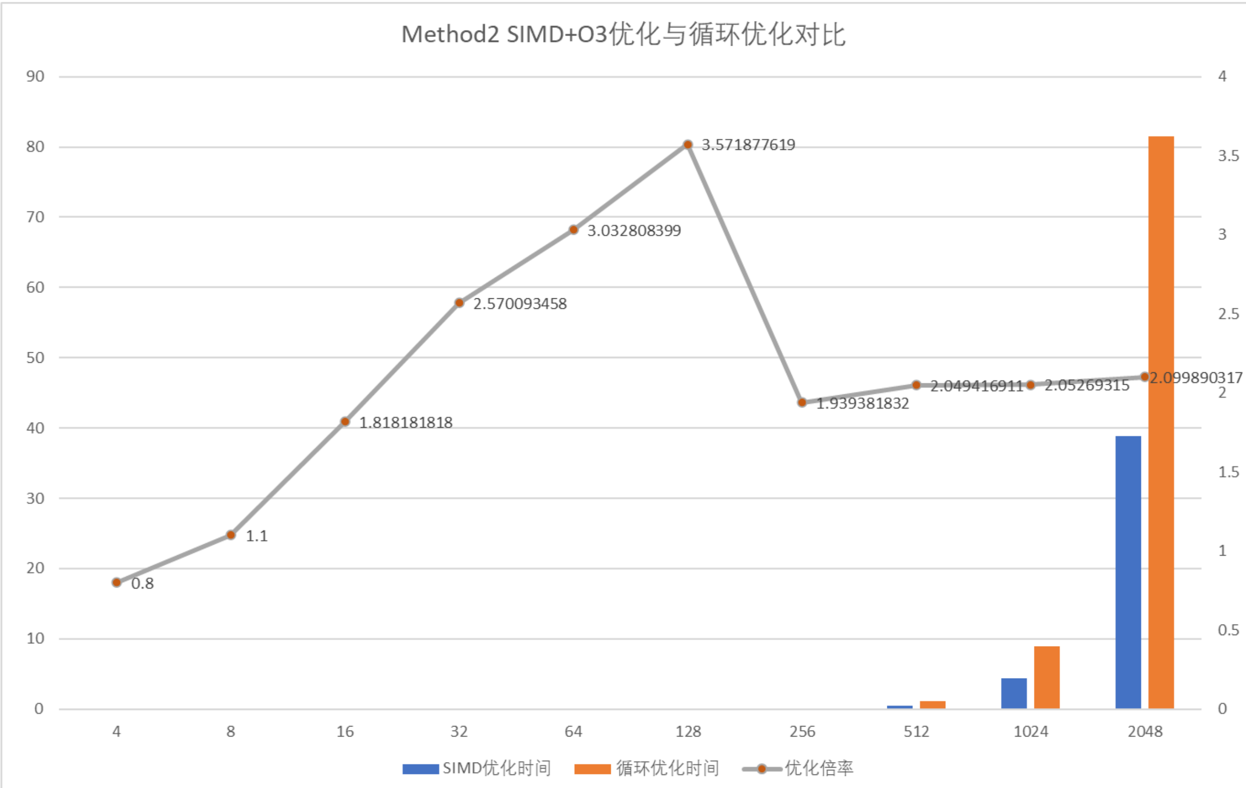

Method1 SIMD+OpenMP+O3优化与cblas_dgemm()对比
120
100
80
60
40
20
0
16
14
12
10
8
6
4
2
0
128
256
512
1024
2048
4096
8192
1.529978587
1.715426594
3.934979424
3.484941863
7.901136261
14.58857799
11.3727321
cblas_dgemm()
SIMD优化
matmul_SIMD_mul运行时间占cblas_dgemm()运行时间倍率


Method4 SIMD+OpenMP+O3优化与cblas_dgemm()对比
120
100
80
60
40
20
0
18
16
14
12
10
8
6
4
2
0
128
256
512
1024
2048
4096
8192
1.405418139
14.27586207
15.98086124
6.037402904
8.20386442
16.39774421
11.3727321
cblas_dgemm()
SIMD优化
matmul_SIMD_mul运行时间占cblas_dgemm()运行时间倍率

可以看到最终本项目优化完的算法在计算大矩阵乘法时速度大约是*cblas_dgemm()*的十分之一，成功将速度差缩小到一个数量级左右。

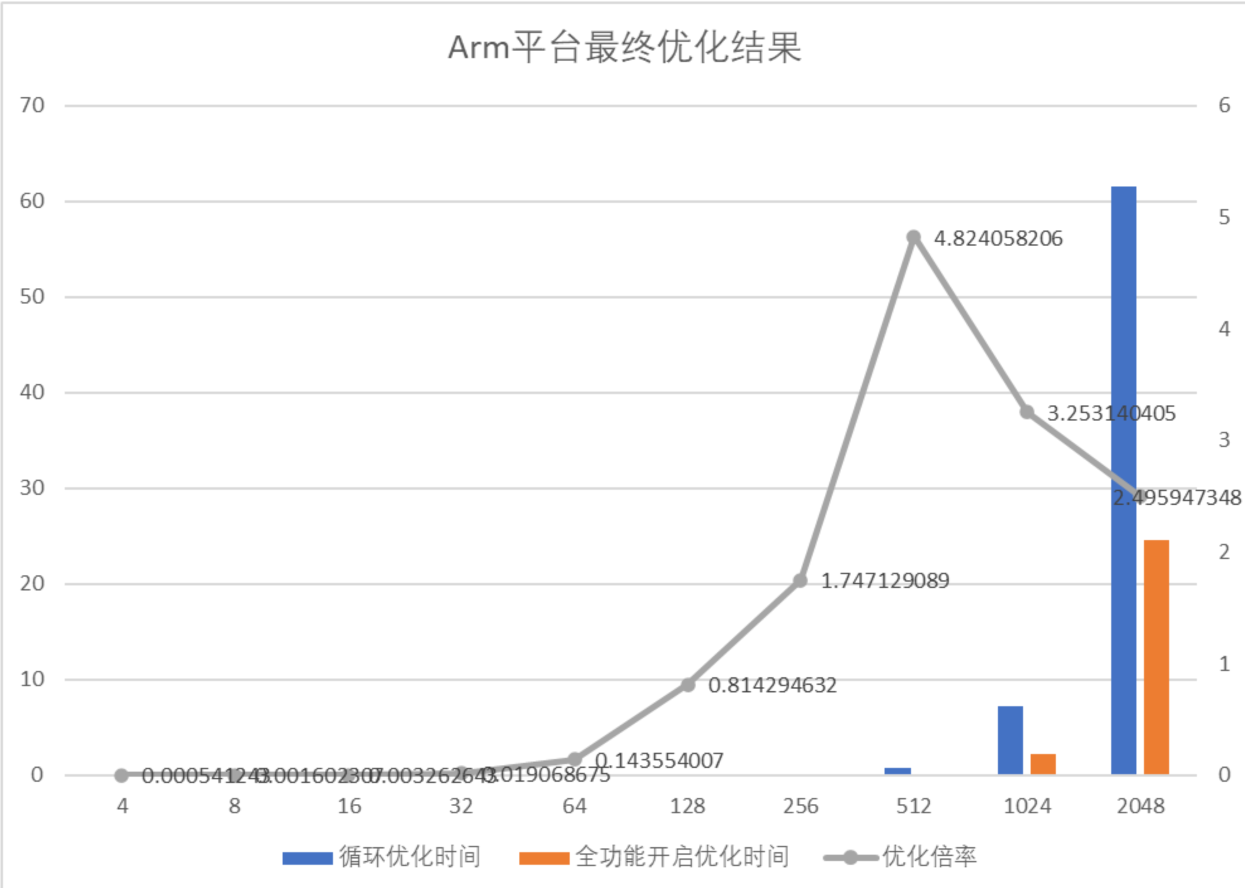
在Arm平台的测试

使用EAIDK-310进行测试，发现SIMD优化效果符合预期。选取优化效果最好的method2方法展示SIMD的效果：



可以看到使用Arm-neon指令集也可以加速矩阵乘法的计算，但是效果并不如x86的效果好。原因是在x86系统上我们使用的是__m256d向量，可以一次进行四个数字的计算，而在Arm平台上，由于硬件限制我使用float64x2_t变量，一次仅仅可以支持两个数字的计算，所以在效率上不如x86好。并且由于CPU本身算力的区别，使用EAIDK-310进行相同的计算比在我的电脑上慢了大约10倍。

开启OpenMP并行处理，由于EAIDK-310本身仅有四核，加速效果并不是很明显，速度大约提升了30%



由于本身硬件限制，在Arm平台的优化测试结果难以展示，优化对矩阵乘法速度有所提升，但不如x86平台上的优化结果。

使用方式

- 使用最简单的算法，请在main.c中设置a的值表示矩阵规模，并使用matmul_easy_mul()方法。
- 使用循环优化的算法，请在main.c中设置a的值表示矩阵规模，并使用matmul_mul()方法。
- 使用仅SIMD优化的算法，请在main.c中设置a的值表示矩阵规模，在matmul.c文件中注释掉*#define Use_openMP*，并使用matmul_SIMD_mul()方法。
- 使用SIMD+OpenMP+编译器优化的算法，请在main.c中设置a的值表示矩阵规模，在matmul.c文件中解除注释*#define Use_openMP和#define Use_SIMD_mavx2（x86）或#define Use_arm_NEON(Arm)*，并使用matmul_SIMD_final_mul()方法。

总结

这就是本次project的全部内容了，感谢您读到这里。本project通过循环优化+SIMD+OpenMP优化实现了比较快速的矩阵乘法，在大矩阵运算上速度大约能到达cblas_dgemm的十分之一。本project支持x86和Arm平台的使用，并对应做了优化。

感悟

到这里这门课所有Project都结束了。想想从第一次做大数字计算器时的一无所知，到现在能够完成矩阵乘法的优化，在project中学到了数不尽的算法和计算机底层的知识。虽然在这五个projects上花费了无数的时间和精力，但我真正在做project的过程中感受到了解决实际问题的快乐和热情，这是课本学习所永远学不到的，也是一段珍贵难得的体验。有时半夜想project的问题甚至感觉回到了高三，那个充满压力但也充满热情的时候。总之，这些project教会我的不仅仅是知识，还有面对问题的态度。我希望这些project带给我的能力能成为未来遇到其他问题时的开门钥匙。