Project4: Matrix Multiplication in C

- 作者：罗嘉诚（Jiacheng Luo）
- 学号：12112910

1 项目介绍

本项目是 `Project3` 的提升版本，矩阵的定义、创建、简单矩阵运算均复用 `Project3` 的 `Matrix` 类进行实现。

本次 `Project4` 中，我们重点进行矩阵乘法效率优化方面的探究。

我会在接下来的报告中向您详细地介绍我是如何一步步优化 `matmul_plain()`（朴素的平凡矩阵乘法）到 `matmul_improved()`（高效率矩阵乘法）的。

在优化过程中，我将会使用 `matmul_method()`（使用某些方法进行优化的矩阵乘法）进行测试，向您展示一些优化矩阵乘法效率的 `idea` 是如何工作的，它们的效果究竟如何。

我会尽可能地使用所探究内容进行效率优化，尝试实现一个我能实现的效率最高的 `matmul_improved()` 并将其在不同数据量范围内与 `OpenBLAS` 中的 `sgemm` 进行对比。

我将在三个平台测试平台中（`x86 架构 PC 机`、`x86 服务器 Ser`、`arm 架构 PC 机`）进行相关测试。

2 描述矩阵乘法效率

我们主要通过两种方式来描述矩阵乘法算法的效率：进行一定规模矩阵运算所需时间、进行一定规模矩阵运算过程中单位时间内的浮点运算（两个浮点数 `float` 进行一次四则运算）的次数。

2.1 时间

显然，衡量算法是否具有高效率，测量其运行时间是最直观的方式。

进行相关测试时，可以使用`Linux`系统库下的`time.h`，需要 `#include "sys/time.h"`

所使用测试的代码如下：

```
const int testCases = 10000;
const int matrixSize = 1;
Matrix *a = createMatrixRandom(matrixSize, matrixSize);
Matrix *b = createMatrixRandom(matrixSize, matrixSize);
Matrix *c;
struct timeval start,end; long timeuse;

// matrixSize is small
timeuse = __LONG_MAX__;
gettimeofday(&start, NULL);
for (int i = 1; i <= testCases; i++) {
    c = matmul_plain(a, b);
}
gettimeofday(&end, NULL);
timeuse = 1000000*(end.tv_sec - start.tv_sec) + end.tv_usec-start.tv_usec;
```

```
printf("Time_avg: %.10f ms\n", timeuse / 1000.0 / testCases);

// matrixSize is large
timeuse = __LONG_MAX__;
for (int i = 1; i <= testCases; i++) {
    gettimeofday(&start, NULL);
    c = matmul_plain(a, b);
    gettimeofday(&end, NULL);
    long tmp = 1000000*(end.tv_sec - start.tv_sec) + end.tv_usec-start.tv_usec;
    if (tmp < timeuse) timeuse = tmp;
    // printf("Test Case#%d: %.10f ms, Min Time: %.10f\n", i, tmp / 1000.0, timeuse / 1000.0);
}
printf("Time_min: %.10f\n", timeuse / 1000.0);
```

上述代码中，有如下两个关键 `idea`，目的是减少单次实验的误差：

- 本项目不使用 `time.h` 中的 `clock()`，而使用 `Linux` 系统库下的`time.h`。

这是由于后续使用`OpenMP`优化时，会使用多线程。`clock()`函数作为最常用的计时函数，返回`CPU`时钟计时单元数（clock tick）。只适用于单线程或单核心运行计时，但使用多线程运行时，`clock()`计时会引发问题，这是因为`clock()`在多线程返回的是`end-begin`的多个核心总共执行的时钟周期数，所以造成时间会偏大。

- 对于较小矩阵进行矩阵乘法，连续进行多组相同计算，求其耗时的平均值。较小矩阵单次矩阵乘法所耗时间较小，难以直接测量，测量误差很大，连续进行多次计算求平均值，可显著降低误差。
- 对于较大矩阵进行多次实验，求若干次实验中最小耗时，使得 `CPU` 能充分发挥峰值效能。

2.2 单位时间浮点运算次数

对于矩阵 $A_{n\times k}$ 和矩阵 $B_{k\times m}$，如果采用朴素平凡算法进行矩阵乘法：

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j<= m; j++)
        for (int p = 1; p <= k; p++)
            C[i][j] = C[i][j] + A[i][p] * B[p][j];
```

共计进行 $2\times n\times m\times k$ 次浮点运算，不同大小的矩阵进行矩阵乘法的效率，可以用单位时间浮点运算次数来表示。

设 1 单位时间（即 1 s）进行矩阵乘法程序，相对于朴素平凡算法进行浮点运算次数可以表示为 $\frac{2\times n\times m\times k}{time}$，其中 $time$ 表示该方法进行计算的时间，我们用 `flops` 表示。

由于直接采用`flops`表示的值太大，我们通常采用 `Gflops` 作为单位，换算关系为：1 Gflops = 10^9 flops。

2.3 不同平台上矩阵乘法效率

为了衡量不同平台上运行同一矩阵乘法效率，考虑在如下三个平台中进行测试：

- x86 架构 PC 机

项目	信息
架构	x86_64
CPU	1 个 8 核 CPU， AMD Ryzen 7 5800H with Radeon Graphics @ 3.20 GHz
内存	16 GB
高速缓存	512 KB （L1 Cache） 4.0 MB （L2 Cache）16.0 KB （L3 Cache）

- x86 服务器 Ser

项目	信息
架构	x86_64
CPU	4 个 14 核 CPU， Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GH
内存	512 GB
高速缓存	896 KB （L1 Cache）28.0 MB（L2 Cache）38.5 MB（L3 Cache）

- arm 架构 PC 机

项目	信息
架构	arm_64
CPU	1 个 8 核 CPU，型号为 Apple M1 Pro @ 3.35 GHz
内存	16 GB
高速缓存	192 KB（L1 Cache） 4.0 MB （L2 Cache）

注意：如果没有特殊注明，程序将在 x86 服务器 Ser 测试，不采用任何编译参数优化。

3 矩阵乘法优化方法探究

3.1 优化一：更改循环次序优化

3.1.1 优化思想

为了方便表示，我们假设参与矩阵乘法的矩阵大小为 $n \times n$。

使用朴素的实现方式（顺序：i, j, k ），它依次计算 $c_{i,j}$，每次将 a 的第 i 行和 b 的第 j 列。在 `Matrix` 库中，矩阵是按行存储的，所以在 a 中相应行中不断向右移动时，内存访问是连续的，但 b 相应列不断向下移动时，内存是不连续的，计算一个$c_{i,j}$时，b 相应列已经间断访问了 n 次，而 a 只间断访问 1 次（跳转到开头）。所以，一共的跳转次数为$n^2(n+1) = n^3 + n^2$次。又由于矩阵是按照一维数组存储了，计算完毕 $c_{i,n}$ 时，不必跳转到本行的开头，所以总的跳转次数为$n^3 + n^2 - n$ 次。

如果我们更改循环顺序（顺序：i, k, j），它依次计算$c_{i,j}$，每次将 a 的第 i 行和 b 的第 j 列。在 `Matrix` 库中，矩阵是按行存储的，所以在 a 中相应行中不断向右移动时，内存访问是连续的，但 b 相应列不断向下移动时，内存也是连续的。所以总的跳转次数为n^2次。

同理，按照循环顺序的不同，我们计算各种循环顺序，总的跳转次数：

循环顺序	跳转次数
i,k,j	n^2
k,i,j	$2n^2$
j,i,k	n^3+n^2+n
i,j,k	n^3+n^2-n
k,j,i	$2n^3$
j,k,i	$2n^3+n^2$

3.1.2 程序实现

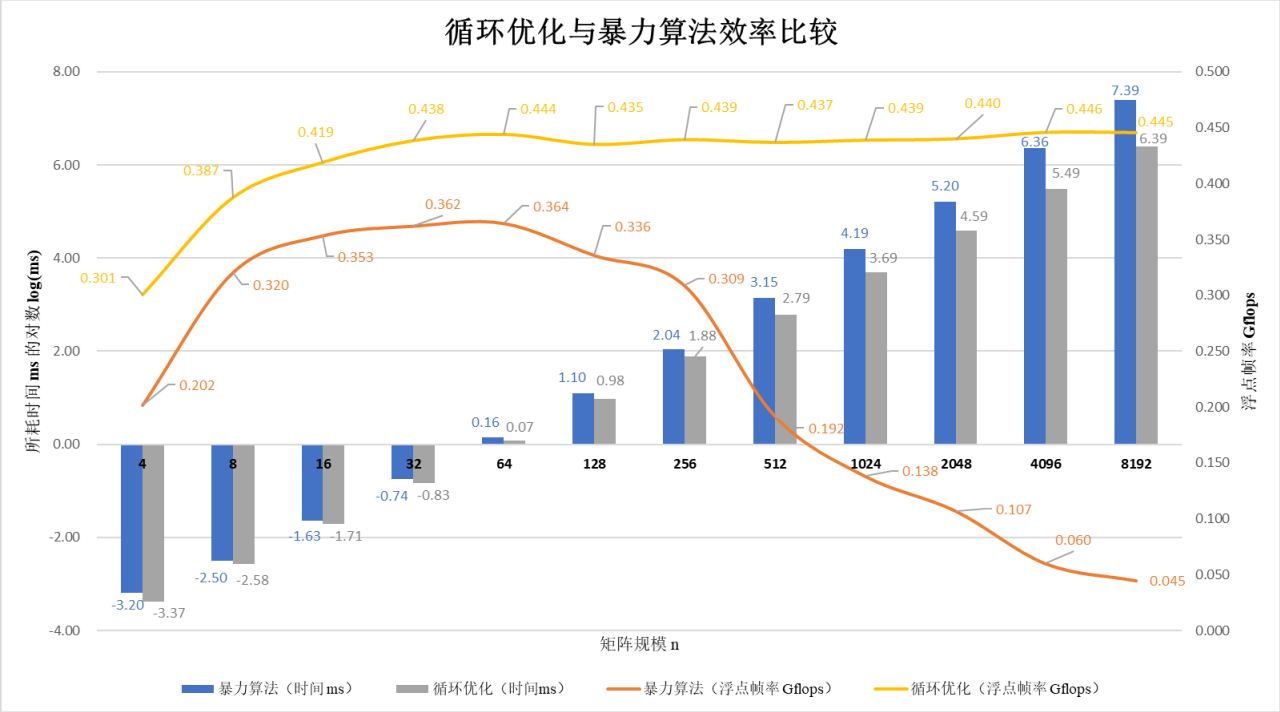
调用函数 `Matrix *matmul_plain(Matrix *matrix1, Matrix *matrix2);` 使用朴素方案。

调用函数 `Matrix *matmul_changeCycleOrder(Matrix *matrix1, Matrix *matrix2)` 使用更改循环次序优化方案。

3.1.3 实验验证

选取 i,k,j 和 j,k,i 两种方式进行实验，前者（循环优化）速度显著快于后者（暴力算法），与理论相符。

矩阵规模	暴力算法耗时（ms）	循环优化耗时（ms）	提升倍率
1	1.76×10^{-4}	1.08×10^{-4}	1.63
2	2.52×10^{-4}	1.32×10^{-4}	1.91
4	6.35×10^{-4}	4.26×10^{-4}	1.49
8	3.20×10^{-3}	2.64×10^{-3}	1.21
16	2.32×10^{-2}	1.96×10^{-2}	1.19
32	1.81×10^{-1}	1.50×10^{-1}	1.21
64	1.44	1.18	1.22
128	12.4	9.65	1.29
256	1.09×10^{2}	7.64×10^{1}	1.42
512	1.40×10^{3}	6.15×10^{2}	2.27
1024	1.55×10^{4}	4.90×10^{3}	3.17
2048	1.60×10^{5}	3.91×10^{4}	4.10
4096	2.29×10^{6}	3.08×10^{5}	7.43
8192	2.45×10^{7}	2.47×10^{6}	9.95



从上图中，我们可以得到如下结果：

- 随着矩阵规模的增大，两种算法所耗时间均增大，使用循环优化算法较朴素暴力算法耗时更短。

  这可能是由于两种算法的复杂度均为$O(n^3)$，而循环优化算法在访问内存的连续性上比暴力算法好。

- 随着矩阵规模的增大，两种算法浮点帧率变化有所不同：

  朴素暴力算法浮点帧率先增大后减小，而循环优化算法浮点帧率先增大后逐渐稳定。

  这可能是由于朴素暴力算法在矩阵规模较大时，其访问内存连续性问题成为制约计算速度的一大原因，而循环优化后，则不存在这个问题。通过理论分析，循环优化后程序在“访问内存连续性”方面的效率大致是朴素暴力算法的 n 倍，在实验上展现出的效果，随着数据规模的增大而越来越明显。

3.1.4 实验结论

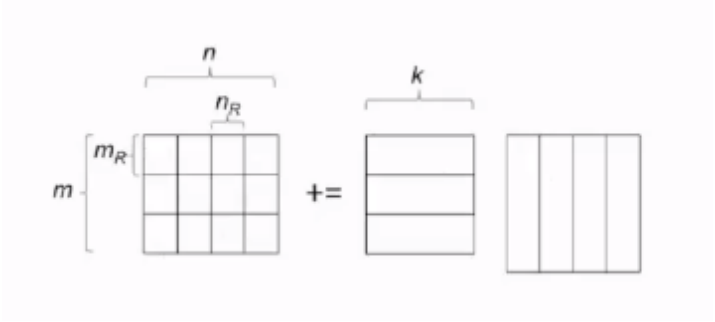
- 访问内存连续性，会很大程度地影响程序的运行效率。
- 尽量让程序连续地访问内存，有助于提高效率，最大发挥出 `CPU` 的计算效能。

3.2 优化二：矩阵分块加速

3.2.1 优化思想

对于矩阵 $A_{m\times k}$ 与矩阵 $B_{k\times n}$ 进行矩阵乘法，得到结果矩阵 $C_{m\times n}$，采用批量将矩阵 A 的行（设一批次为m_R行），矩阵B 的列（设一批次为n_R列）进行对应向量的乘法，得到大小为$m_R \times n_R$的结果矩阵，该矩阵就是结果矩阵C中的一部分。

示意图如下：



取 $m_R = 2, n_R = 2$ 我们设计了一个应用此思想的算法 `matmul_blockMatrix_2x2(matrix1, matrix2);`

在实现上述算法时，我们还进行了一定的访存优化，如：循环展开、将2×2的计算结果存储在寄存器内。

需要注意的是：由于分块的过程与朴素循环相比常数稍大，如果不加入上述访存优化，实验效果将不够明显。

我们还设计了较为一般的矩阵分块计算矩阵乘法的算法`matmul_blockMatrix(matrix1, matrix2);`

由于其内层是一个大小为$m_R \times n_R$的不定大小的矩阵，因此无法进行循环展开和中间结果保存在寄存器内。

因此，该算法在定义$m_R = n_R = 2$时，由于常数方面的问题，其运行效率应该略低于上面的算法。

这个算法的进步意义在于，我们可以自主定义分块矩阵的规模，观察分块矩阵规模大小与程序效率的关系。

3.2.2 程序实现

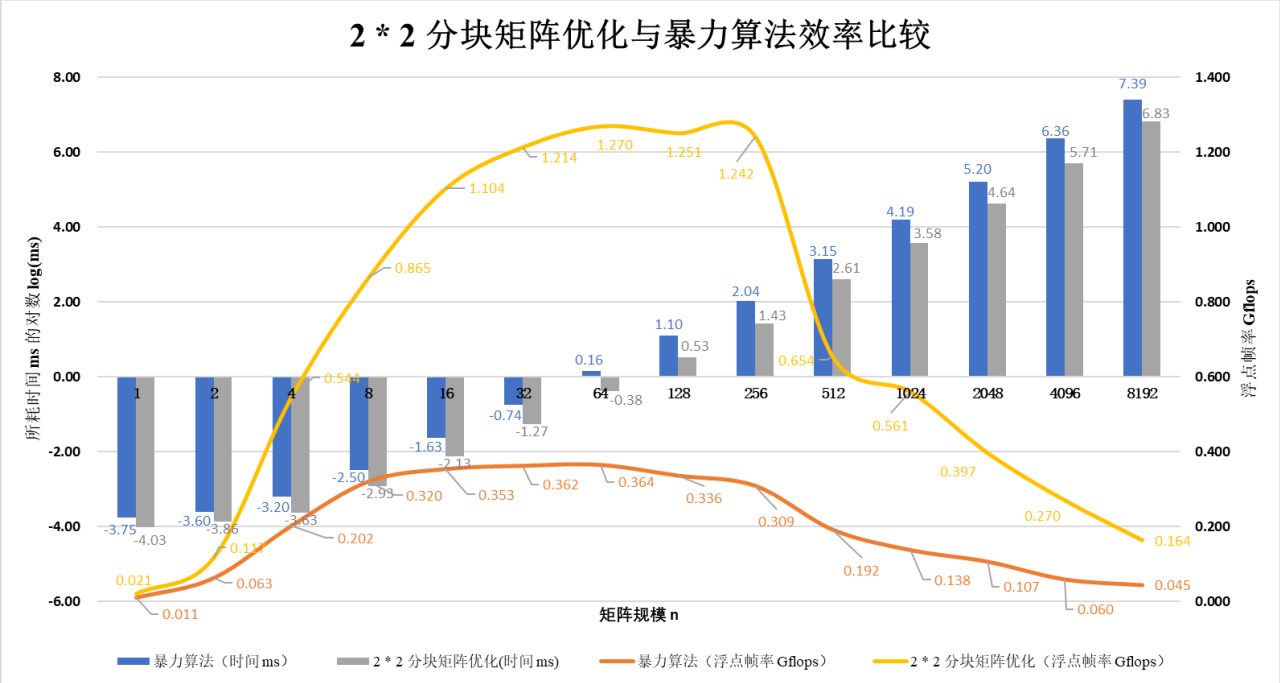
您可以使用 `Matrix *matmul_blockMatrix(Matrix *matrix1, Matrix *matrix2)` 将矩阵按照`NR * MR`进行分块计算优化（该函数仅作为通用方案陈述，由于效率方面的原因，我们不推荐您使用该函数）。

您可以使用 `Matrix *matmul_blockMatrix_2x2(Matrix *matrix1, Matrix *matrix2)` 将矩阵按照`2 * 2`进行分块计算优化（该函数中进行了上述循环展开、将2×2的计算结果存储在寄存器内等访存优化方案，该优化方案是 2.3 节使用矩阵分块优化矩阵乘法的最快速版本，我们推荐您使用该函数）。

您可以使用 `Matrix *matmul_blockMatrix_4x4(Matrix *matrix1, Matrix *matrix2)` 将矩阵按照`4 * 4`进行分块计算优化（该函数中进行了上述循环展开、将4×4的计算结果存储在寄存器内等访存优化方案，但由于程序实现常数方面问题，优化方案相较于前者仍然有一定差距，我们不推荐您使用该函数）。

3.2.3 实验验证

矩阵规模	暴力算法耗时（ms）	2 * 2矩阵分块优化耗时（ms）	提升倍率
1	1.76×10^{-4}	9×10^{-5}	1.87
2	2.52×10^{-4}	1.36×10^{-4}	1.85
4	6.35×10^{-4}	2.35×10^{-4}	2.70
8	3.20×10^{-3}	1.18×10^{-3}	2.70
16	2.32×10^{-2}	7.42×10^{-3}	3.12
32	1.81×10^{-1}	5.40×10^{-2}	3.35
64	1.44	4.13×10^{-1}	3.48
128	12.4	3.35	3.72
256	1.09×10^{2}	2.70×10^{1}	4.02
512	1.40×10^{3}	4.11×10^{2}	3.40
1024	1.55×10^{4}	3.83×10^{3}	4.06
2048	1.60×10^{5}	4.33×10^{4}	3.70
4096	2.29×10^{6}	5.08×10^{5}	4.51
8192	2.45×10^{7}	6.70×10^{6}	3.67



有上面的数据和图表可知，使用`2 * 2 分块矩阵优化`能在一定程度上加速矩阵乘法，相较于朴素暴力算法，其效率在较大数据规模上有显著提升。

但是上述算法的浮点帧率随着矩阵规模的增大先快速提高，再逐渐下降，这有可能由于我们在访问$m_R \times K$和$K \times n_R$的子矩阵时候，会造成严重的cache miss，，在大数据规模的矩阵计算时，同不进行循环优化的朴素暴力算法一样，内存访问逐渐成为制约计算速率的重要因素之一。

我们选取矩阵规模为 1024 的三个矩阵，计算分块大小为 $2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$ 时的运算速率。

分块矩阵规模	**1024 * 1024** 矩阵耗时（ms）
1	5142.302
2	5281.307
4	5282.674
8	5222.789
16	5231.800
32	5282.674
64	5280.259
128	5124.433
256	5483.004
512	5058.718
1024	5018.117

可以发现，由于常数问题的原因，分块矩阵的规模大小几乎不影响矩阵乘法的耗时。

3.2.4 实验结论

- 使用带访存优化（循环展开、使用寄存器）的`2 * 2 分块矩阵优化`算法可以有效加快矩阵运算效率，但在大规模矩阵计算时，内存问题依然会造成浮点帧率下降，无法发挥`CPU`的计算效能。
- 不使用访存优化的`分块矩阵优化`算法，虽然与暴力算法相比，能有效加快矩阵运算效率，但与上述带访存优化（循环展开、使用寄存器）的`2 * 2 分块矩阵优化`相比仍然稍稍逊色。

3.3 优化三：Strassen 算法

3.3.1 优化思想

假设矩阵 A, B 都是 $N \times N (N = 2^n)$ 的方阵，若求矩阵乘法 $C = A \times B$。

对矩阵进行分块，$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

那么，

- $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
- $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{21}$
- $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
- $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

如果递归的使用上述算法，计算 $n \times n$ 的矩阵相乘需要两个 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵进行8次乘法，和4次加法。

那么时间复杂度 $T(n) = 8T(\frac{n}{2}) + O(n^2)$，所以时间复杂度为 $O(n^3)$。

使用 Strassen 算法将减少进行乘法的次数，使其进行乘法的次数从 8 次减小到 7 次。

定义 $S_1, \ldots, S_{10}$ 为：

$S_1 = B_{12} - B_{22}$ $S_2 = A_{11} + A_{12}$ $S_3 = A_{21} + A_{22}$ $S_4 = B_{21} - B_{11}$ $S_5 = A_{11} + A_{22}$

$S_6 = B_{11} + B_{22}$ $S_7 = A_{12} - A_{22}$ $S_8 = B_{21} + B_{22}$ $S_9 = A_{11} - A_{21}$ $S_{10} = B_{11} + B_{12}$

定义 $P_1, \ldots, P_7$ 为：

$P_1 = A_{11}S_1$ $P_2 = S_2B_{22}$ $P_3 = S_3B_{11}$ $P_4 = A_{22}S_4$ $P_5 = S_5S_6$ $P_6 = S_7S_8$ $P_7 = S_9S_{10}$

因此，可以得到：

$C_{11} = P_5 + P_4 - P_2 + P_6$ $C_{12} = P_1 + P_2$ $C_{21} = P_3 + P_4$ $C_{22} = P_5 + P_1 - P_3 - P_7$

上述算法计算 $n \times n$ 的矩阵相乘需要两个 $\frac{n}{2} \times \frac{n}{2}$ 的矩阵进行7次乘法，和18次加法。

那么时间复杂度 $T(n) = 7T(\frac{n}{2}) + O(n^2)$，所以时间复杂度为 $O(n^{\log_2 7})$。

如果使用上述算法，矩阵乘法的时间复杂度由$O(n^3)$优化到$O(n^{\log_2 7})$。

此外，如果我们将 Strassen 算法优化和 更改循环次序优化结合起来（即在分块后矩阵的行数小于一定值时，直接采用`更改循环次序`暴力求解 ，在进行矩阵大小大于 128 测试时，这个值将被被设置为 128）。

3.3.2 程序实现

调用函数`Matrix *matmul_strassen(Matrix *matrix1, Matrix *matrix2)` 使用`Strassen`优化方案。

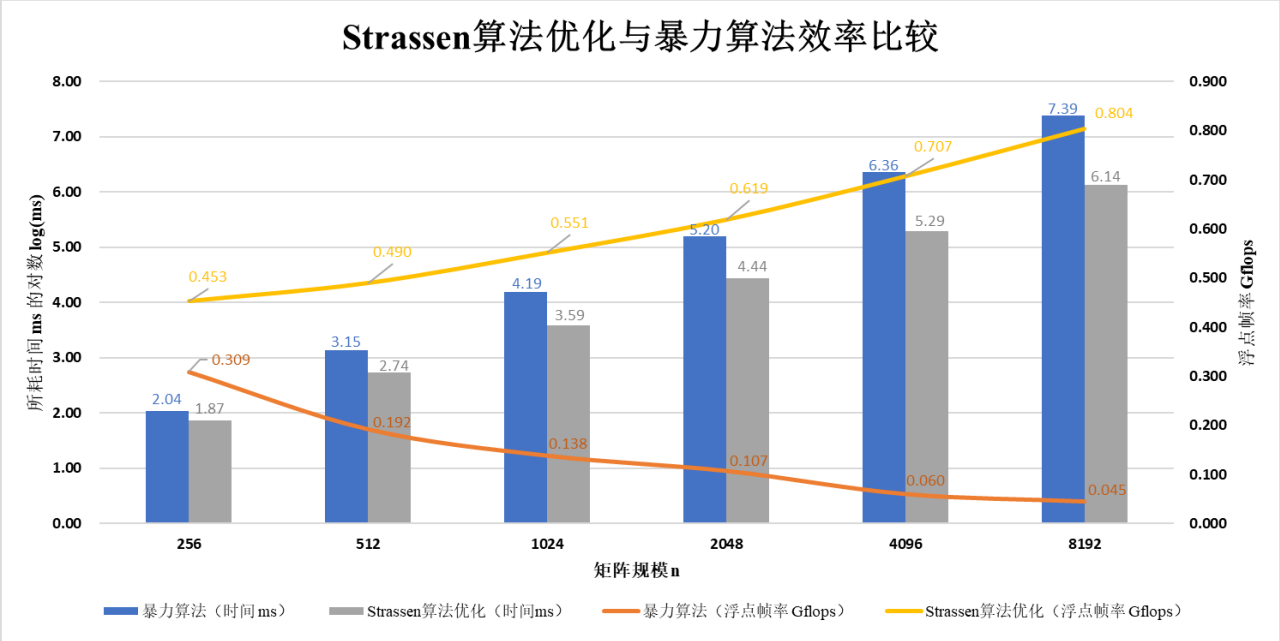
3.3.3 实验验证

进行矩阵大小小于等于 128 的矩阵乘法计算（直接采用Strassen 算法）中，我们可以得到如下数据：

矩阵规模	暴力算法耗时（ms）	Strassen算法耗时（ms）	提升倍率
1	1.76×10^{-4}	1.82×10^{-4}	0.967
2	2.52×10^{-4}	4.00×10^{-3}	0.063
4	6.35×10^{-4}	3.41×10^{-2}	0.018
8	3.20×10^{-3}	2.35×10^{-1}	0.014
16	2.32×10^{-2}	1.70	0.014
32	1.81×10^{-1}	1.17×10^{1}	0.015
64	1.44	8.29×10^{1}	0.017
128	12.4	5.83×10^{2}	0.021

我们发现，使用 Strassen 算法后，其效率反而比暴力算法要差，我们认为这是由于在算法执行过程中定义了很多中间矩阵变量，加之较少数据量情况下该算法优化效果不明显，所以在执行更大规模的矩阵乘法时，我们将 Strassen 算法执行过程中的，处理两个小于 128 大小矩阵的乘法的计算，使用前面`更改循环次序`的$O(n^3)$算法求解。实验结果表明，该优化可以大大提高 Strassen 算法的效率。

矩阵规模	暴力算法耗时（ms）	Strassen算法耗时（ms）	提升倍率
256	1.09×10^{2}	7.41×10^{1}	1.47
512	1.40×10^{3}	5.48×10^{2}	2.54
1024	1.55×10^{4}	3.89×10^{3}	3.99
2048	1.60×10^{5}	2.78×10^{4}	5.77
4096	2.29×10^{6}	1.94×10^{5}	11.80
8192	2.45×10^{7}	1.37×10^{6}	17.96



3.3.4 实验结论

- 采用 `Strassen` 算法将时间复杂度从$O(n^3)$ 降低到 $O(n^{\log_2 7})$，在大规模矩阵计算中能大大提高效率。
- 在较小矩阵规模下，采用 `Strassen` 算法，由于常数原因，其速度较慢，应该采用前面优化效果较好的方案进行代替。
- 采用 `Strassen` 算法，其浮点帧率随着数据规模的增大稳定增加。

3.3.5 其他

Strassen 算法因为有很多固有的缺点：

1. 常数因子远远大于 $O(n^3)$的朴素算法，在小数据规模下尤其明显。
2. 递归实现时，系统栈中生成的子矩阵开销较大，所需消耗更多的空间。
3. 数值计算有极大的不稳定性，无法保证结果的可靠性，在大数据规模下尤其明显。

正因为如此，在大多数矩阵运算库中，我们并不使用它。

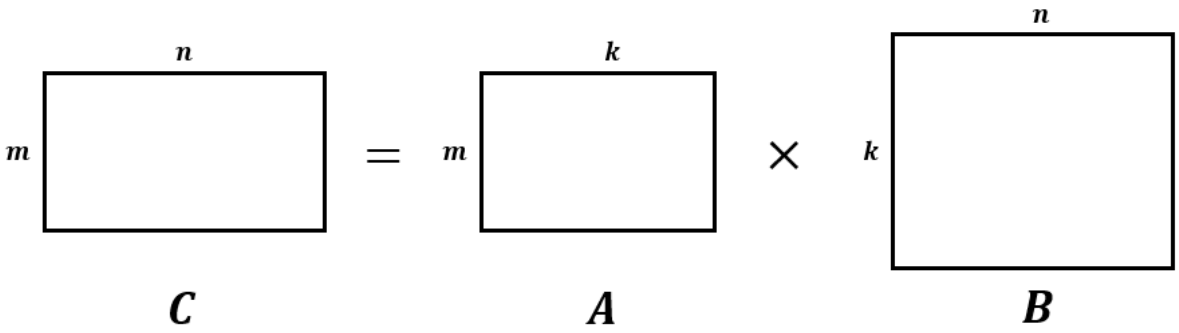
3.4 优化四：矩阵打包优化

3.4.1 优化思想

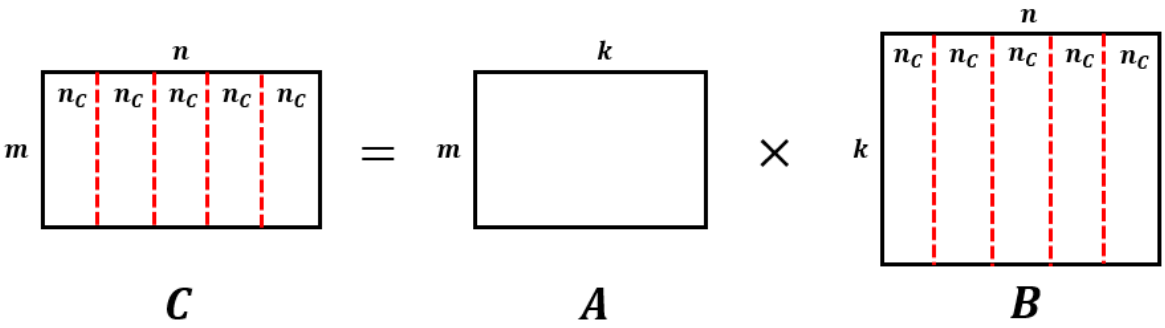
在 3.2 节中，我们将矩阵试图放入到寄存器中，不断复用寄存器中的数据，实现一定程度的优化。但当矩阵较大以后，由于 K 的值较大，在访问 $m_R \times K$ 的子矩阵和$K \times n_R$ 的子矩阵时，一样有很重的 cache miss。您能从实验测试中看到，当矩阵规模超过 256 以后，此优化的效果并不明显。

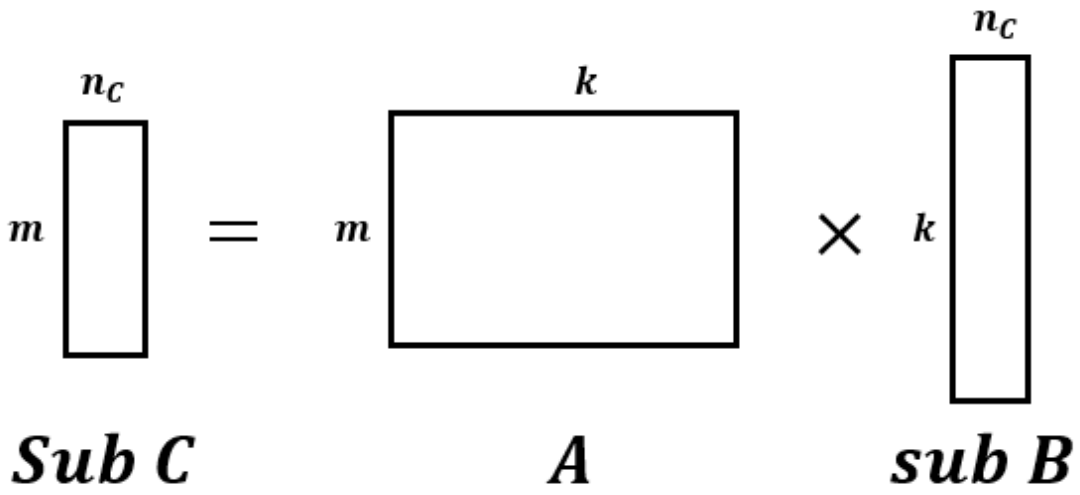
在本节内容中，我们尝试将不连续的数据，拷贝到连续的内存中，在计算时从连续的内存中读取，这样能降低 cache miss 的概率，从而提高程序的效率。

假设我们要实现 $A_{m\times k}$ 与 $B_{k\times n}$ 两个矩阵的乘法，得到矩阵 $C_{m\times n}$，如下图所示：

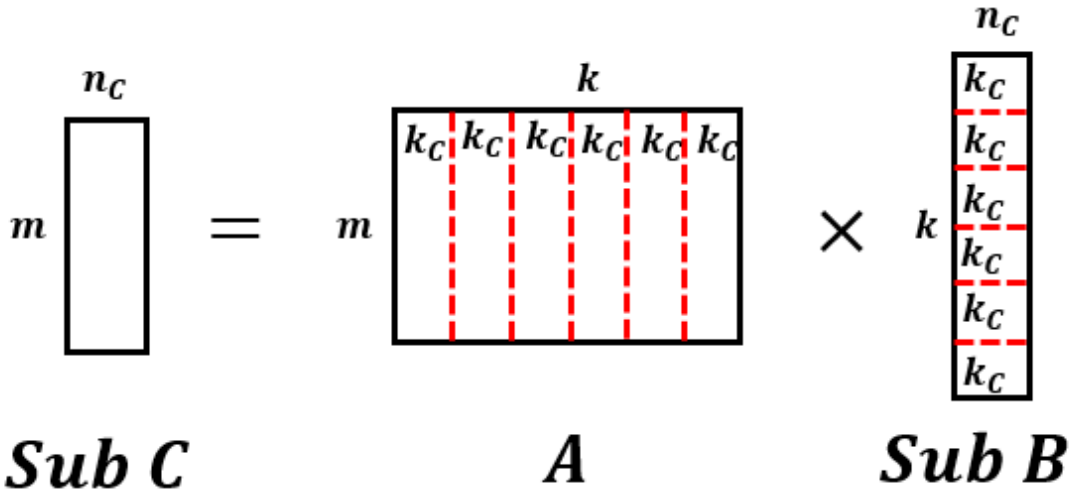


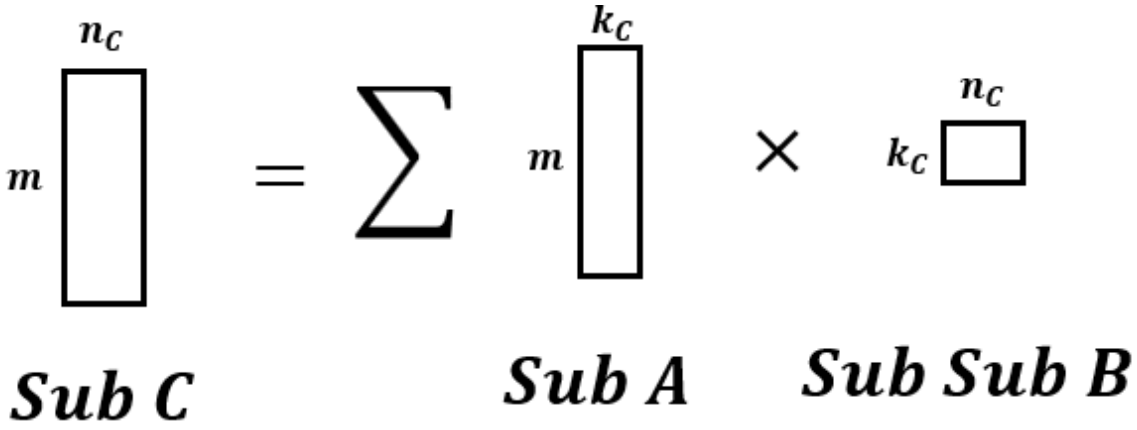
首先，我们对 n 这一维度，进行大小为 n_C 的分块，分出的每一块分别进行处理。



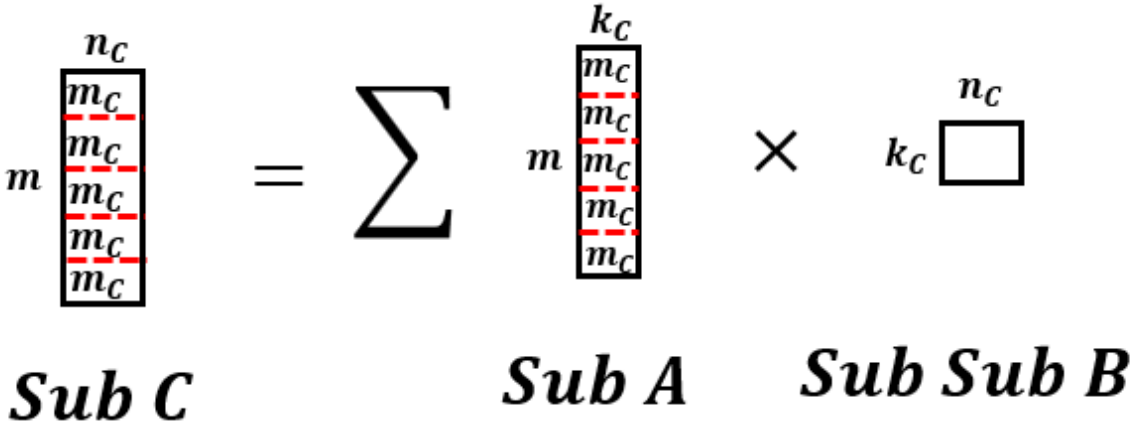


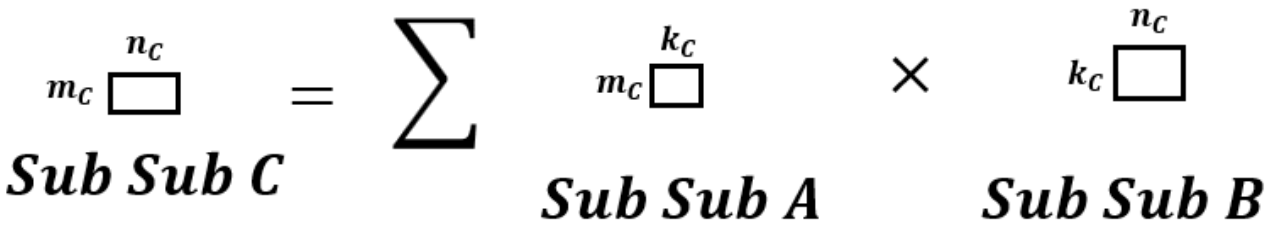
然后，我们对 k 这一维度，进行大小为 k_C 的分块，分出的每一块分别进行处理。





然后，我们对 m 这一维度，进行大小为 m_C 的分块，分出的每一块分别进行处理。

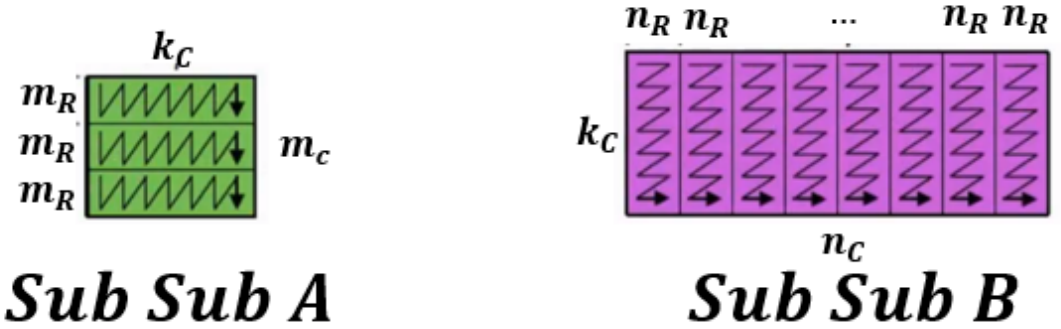


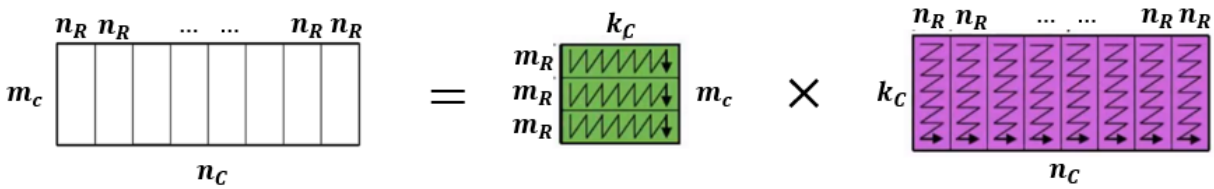


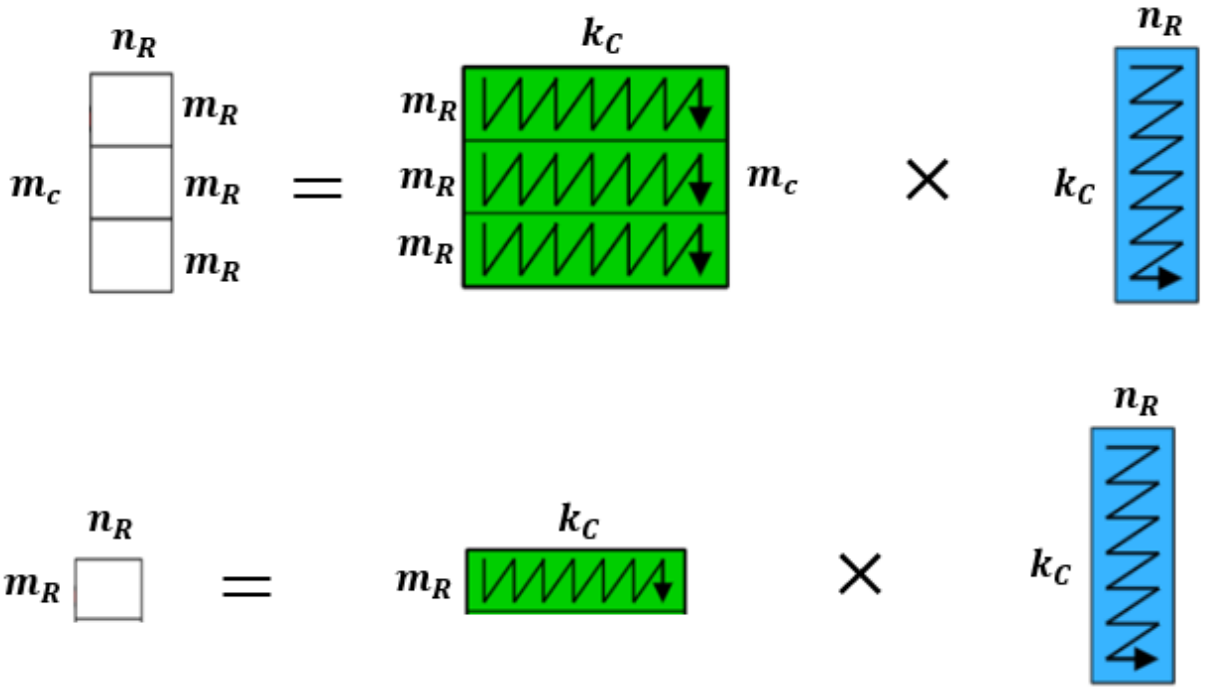
在计算 $m_C \times k_C$ 大小的矩阵和 $k_C \times n_C$ 大小的矩阵时，我们使用 2.3 中的的 `2 * 2 分块矩阵优化` （即定义 $n_R = m_R = 2$，更一般地，我们可以自定义n_R和 m_R的值，再对两个矩阵进行分块优化，由于 3.2 中的探究表明，`2 * 2` 分块优化的实现效率最高，故在此处使用）。

我们可以通过设置m_C, n_C, k_C的参数，使得每次计算能发挥 `2 * 2 分块矩阵优化` 算法的最大帧频率。

上面提到的将不连续的数据，拷贝到连续的内存中，在计算时从连续的内存中读取。我们可以通过对 `sub Sub A` 和 `sub sub B` 进行如下的打包。







计算$n_R \times m_R$可以先将结果放置在寄存器内，然后等 k_C 次浮点运算结束后，再将寄存器内结果移植到原内存。

值得注意的是，进行打包矩阵 `pack Sub Sub A` 和 `pack Sub Sub B` 进行内存申请时，我们不建议直接使用 `malloc`，而是使用`posix_memalign`，后者能实现动态内存的对齐，这会使得 Cache 的读取更快，并且在后续进行向量化读取指令进行优化时，效率能进一步增加。

3.4.2 程序实现

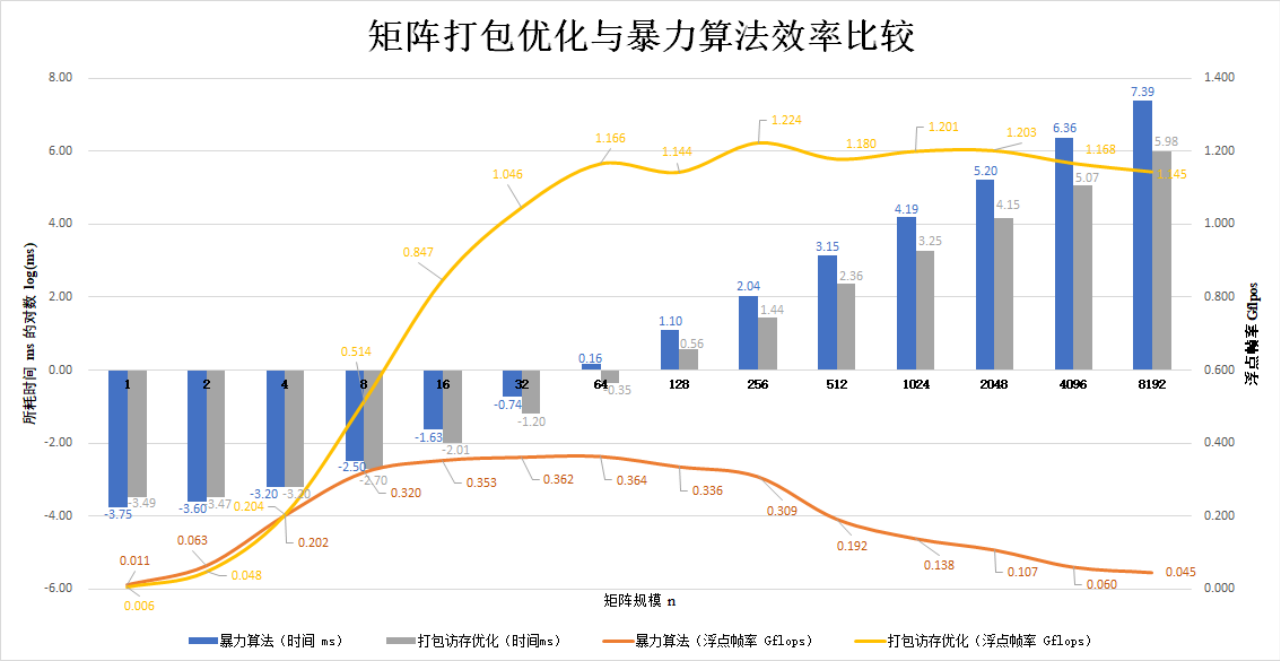
调用函数 `Matrix *matmul_package(Matrix *matrix1, Matrix *matrix2)` 使用`矩阵打包`优化方案。

在我的实现中，依据 3.2 节的内容，我将 $m_R = n_R = 2$，这在`_matmul_package_addDot_2x2` 中有所体现，我也实现了一个`_matmul_package_addDot_MRxNR` 用于通用地实现其他情况（虽然效果并不够好），您可以根据您地需求，完成 m_R, n_R 是其他值的实现，以获得在您的计算机中更好的效果。

除此以外，依据简单的实验，我将其他参数分别定为：$n_C = 72, m_C = 4080, k_C = 256$。这并不意味着这些参数在您的本地能跑出最好的效果，您可以依据您的喜好调节参数，以获得最大效率。

3.4.3 实验验证

矩阵规模	暴力算法耗时（ms）	矩阵打包优化耗时（ms）	提升倍率
1	1.76×10^{-4}	3.26×10^{-4}	0.54
2	2.52×10^{-4}	3.35×10^{-4}	0.75
4	6.35×10^{-4}	6.27×10^{-4}	1.01
8	3.20×10^{-3}	1.99×10^{-3}	1.60
16	2.32×10^{-2}	9.67×10^{-3}	2.40
32	1.81×10^{-1}	6.27×10^{-2}	2.89
64	1.44	4.49×10^{-1}	3.20
128	12.4	3.67	3.40
256	1.09×10^{2}	2.74×10^{1}	3.96
512	1.40×10^{3}	2.28×10^{2}	6.15
1024	1.55×10^{4}	1.79×10^{3}	8.69
2048	1.60×10^{5}	1.43×10^{4}	11.22
4096	2.29×10^{6}	1.18×10^{5}	19.47
8192	2.45×10^{7}	9.60×10^{5}	25.56



3.4.4 实验结论

- 采用 矩阵打包优化 对访存做了极大的优化，在大规模矩阵计算中能大大提高效率，所耗时间下降非常明显。
- 采用 矩阵打包优化 优化，其浮点帧率随着数据规模的增大稳定增加，并且浮点频率较高，并且在较大数据规模时依然非常稳定，所以它是一种优秀的方法。

3.5 优化五：使用 OpenMP 进行多线程并行计算

3.5.1 优化思想

上面的优化，我们都只使用了计算机的一个核心 CPU，或者是一个进程，并没有把计算机所有核心都使用上。有上面的一些算法（如调整循环次序、矩阵分块、矩阵打包）可知，我们每次都是一部分一部分地计算矩阵的每一个部分，而这些部分确实互相独立的，这非常符合可以进行并行计算的特点。

如果有指令，能够使得这些部分并行地进行执行，显然能大大加快运行效率。

幸运的是，很多编译器都支持`OpenMP`，不需要额外安装，只需要加上相关编译参数即可使用。

如在本项目中，我在上述所有优化中，都使用了`#pragma omp parallel for schedule(dynamic)`让计算机自动分配计算资源，从而最大化进程效率，并使用宏定义`#define UseParallelOptimization`实现开关操作。

如果您需要比较使用或者不使用`OpenMP`的前后差异比较，您可以注释或者恢复相关宏定义。

需要特别指出的是，`Stressen`算法由于其诸多确定，虽然在理论上复杂度确实是最优的，但是在实际工程中，由于其缺点太过明显（不精确、不稳定、需要额外空间），我们并不经常使用它，所以我们不再使用并行计算优化它。

3.5.2 代码实现

需要使用`#include "omp.h"`库，编译时需要加入`-fopenmp`编译参数。

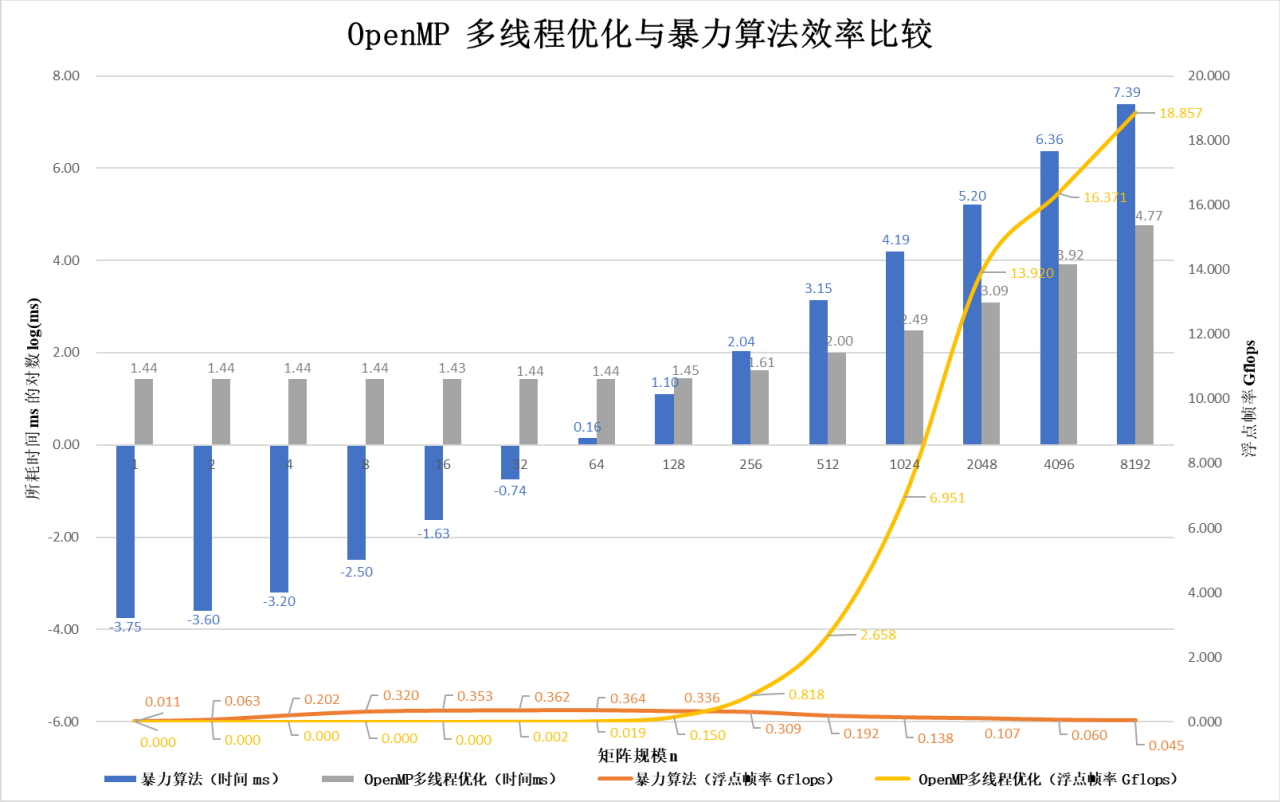
定义宏`UseParallelOptimization`则对每个矩阵乘法函数（除了`matmul_stressen`）都应用并行计算。

注释宏`UseParallelOptimization`则对每个矩阵乘法函数（除了`matmul_stressen`）都不应用并行计算。

3.5.3 实验验证

下表和下图数据均是在优化三基础上，进行优化四进行实验获得的数据。

矩阵规模	暴力算法耗时（ms）	矩阵打包 + OpenMP 多线程优化耗时（ms）	提升倍率
1	1.76×10^{-4}	2.73×10^{1}	0.00
2	2.52×10^{-4}	2.72×10^{1}	0.00
4	6.35×10^{-4}	2.73×10^{1}	0.00
8	3.20×10^{-3}	2.73×10^{1}	0.00
16	2.32×10^{-2}	2.72×10^{1}	0.00
32	1.81×10^{-1}	2.73×10^{1}	0.01
64	1.44	2.73×10^{1}	0.05
128	12.4	2.80×10^{1}	0.45
256	1.09×10^{2}	4.10×10^{1}	2.65
512	1.40×10^{3}	1.01×10^{2}	13.83
1024	1.55×10^{4}	3.09×10^{2}	50.27
2048	1.60×10^{5}	1.23×10^{3}	129.81
4096	2.29×10^{6}	8.40×10^{3}	272.96
8192	2.45×10^{7}	5.83×10^{4}	421.03



3.5.4 实验结论

- 采用 `OpenMP` 优化使得程序能充分使用计算机的所有 `CPU` 资源，并行化地进行计算，大大提高了计算机处理数据的效率。
- 采用 `OpenMP` 优化，其浮点帧率随着数据规模的增大稳定增加，并且浮点频率相当高，并且在较大数据规模时依然非常稳定，所以它是一种很优秀的方法。
- 由于在较低数据量下并行计算实现常数较大，所以效果上远不如使用暴力算法计算，而在较大数据规模下，并行计算的优势淋漓尽致地得到了体现。

3.6 优化六：使用 SIMD / NEON 优化

3.6.1 优化思想

在计算 $m_R \times k_C$ 和 $k_C \times n_R$ 两个小矩阵的时，可以看成 m_R 个行向量和 n_R 个列向量分别做点积运算（每个向量的长度均为k_C）。这可以使用 `SIMD / NEON` 指令集进行优化。

由于 A 矩阵提供行向量，B 矩阵提供列向量，所以在进行打包时，我们在延用 3.4 的打包方式的同时，对 `sub sub A` 的打包使用**行优先**而非列优先，对 `sub sub B` 的打包使用**列优先**而非行优先。

如果暴力地计算两个向量的点积，加入到目标中，代码如下：

```
for (int p = 0; p < len; p++) {
    *dest += v1[p] * v2[p];
}
```

我们在优化四的基础上，进一步使用 `SIMD / NEON` 进行优化。

- 对于 `x86` 平台上的指令集，可以选用 `avx2` 指令集、`avx512f` 指令集， 需要导入库 `#include "immintrin.h"`
- 对于 `arm` 平台上的指令集，可以选用 `neon` 指令集，需要导入库 `#include <arm_neon.h>`

使用 `_m256` 向量化编程（每 8 个实数作为一个向量），使用如下代码能得到相同的结果，并加速计算：

```
float sum[8] = {0};
__m256 sum256 = _mm256_setzero_ps();
register int p;
register float *p1 = v1, *p2 = v2;
for (p = 0; p + 7 < len ; p += 8, p1 += 8, p2 += 8) {
    sum256 = _mm256_add_ps(sum256, _mm256_mul_ps(_mm256_loadu_ps(p1),
_mm256_loadu_ps(p2)));
}
sum256 = _mm256_hadd_ps(sum256, sum256);
sum256 = _mm256_hadd_ps(sum256, sum256);
_mm256_storeu_ps(sum, sum256);
*dest += sum[0] + sum[4];
for (; p < len; p++, p1++, p2++) {
    *dest += (*p1) * (*p2);
}
```

使用 `_m512` 向量化编程（每 16 个实数作为一个向量），使用如下代码能得到相同的结果，并加速计算：

```
__m512 sum512 = _mm512_setzero_ps();
register int p;
for (p = 0; p + 15 < len ; p += 16, v1 += 16, v2 += 16) {
    sum512 = _mm512_add_ps(sum512, _mm512_mul_ps(_mm512_loadu_ps(v1),
_mm512_loadu_ps(v2)));
}
*dest += _mm512_reduce_add_ps(sum512);
for (; p < len; p++, v1++, v2++) {
    *dest += (*v1) * (*v2);
}
```

而在 `ARM` 架构中，则需要使用到 `float32x4_t` 进行向量化。

```
float32x4_t a, b;
float32x4_t c = vdupq_n_f32(0);
float sum[4] = {0};
int p;
for (p = 0; p + 4 < len; p += 4) {
    a = vld1q_f32(v1 + p);
    b = vld1q_f32(v2 + p);
    c = vaddq_f32(c, vmulq_f32(a, b));
}
vst1q_f32(sum, c);
*dest+=(sum[0]+sum[1]+sum[2]+sum[3]);
for (; p < len; p ++) {
    *dest += v1[p] * v2[p];
}
```

3.6.2 代码实现

对于`x86`架构，需要使用库：`#include "immintrin.h"`，编译时需加`-mavx2`或者`-mavx512f`参数。

对于`arm`架构，需要使用库：`#include <arm_neon.h>`。

我们可以使用宏定义`#define Use_x86_SIMD_mavx2`表示使用`x86`的`SIMD`的`mavx2`，以此类推可以使用宏定义控制`#define Use_x86_SIMD_mavx512f`、`#define Use_arm_NEON`。

调用函数`Matrix *matmul_package_SIMD_NEON(Matrix *matrix1, Matrix *matrix2);`使用`SMID or NEON`优化方案。

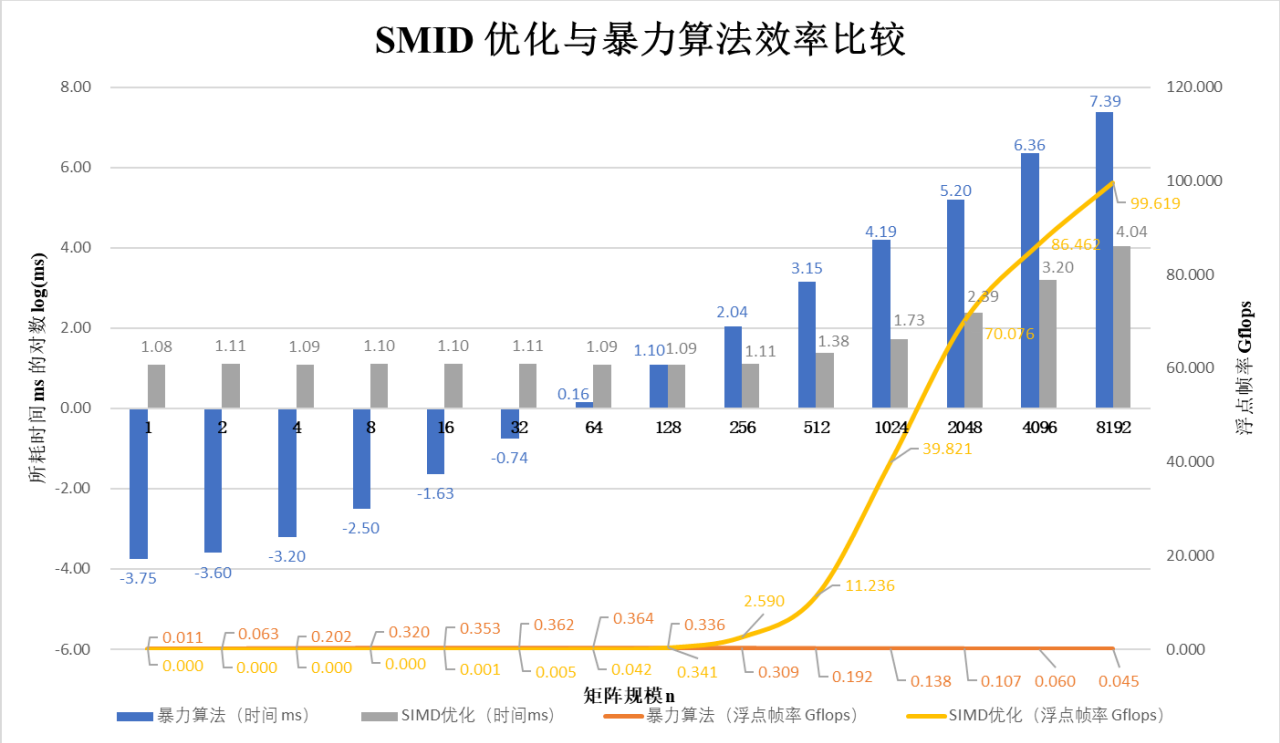
该函数基于`_matmul_package_addDot_2x2_SIMD_NEON`函数实现（根据 2.3 的讨论，2×2 的分块效率较高），您也可以调整分块的大小，使用更为特殊的`_matmul_package_addDot_4x4_SIMD_NEON`函数，或更为一般的`_matmul_package_addDot_MRxNR_SIMD_NEON`函数。

3.6.3 实验验证

该优化是本次项目优化的最终版本，它结合了`3.1 循环优化`、`3.2 分块优化`的思想（尽可能让内存连续、尽可能提高Cache的命中率），吸取了`3.3 Stressen 算法`局限性（过分追求理论复杂度的降低往往会带来复杂性的上升）带来的教训，使用`3.4 矩阵打包优化`、`3.5 OpenMP 多线程优化`、`3.6 SIMD指令集优化`。

我们认为，这是一个非常优秀的矩阵乘法方案，您将会在接下来的测试中看出，在较大的矩阵规模下，使用该方法可以极高地提高矩阵乘法的效率。

矩阵规模	暴力算法耗时（ms）	矩阵打包 + OpenMP 多线程 + SIMD优化耗时（ms）	提升倍率
1	1.76×10^{-4}	1.21×10^{1}	0.00
2	2.52×10^{-4}	1.28×10^{1}	0.00
4	6.35×10^{-4}	1.21×10^{1}	0.00
8	3.20×10^{-3}	1.26×10^{1}	0.00
16	2.32×10^{-2}	1.26×10^{1}	0.00
32	1.81×10^{-1}	1.29×10^{1}	0.01
64	1.44	1.24×10^{1}	0.12
128	12.4	1.23×10^{1}	1.01
256	1.09×10^{2}	1.30×10^{1}	8.39
512	1.40×10^{3}	2.39×10^{1}	58.47
1024	1.55×10^{4}	5.39×10^{2}	288.00
2048	1.60×10^{5}	2.45×10^{2}	653.48
4096	2.29×10^{6}	1.59×10^{3}	1441.56
8192	2.45×10^{7}	1.10×10^{4}	2224.27



3.6.4 实验结论

我们使用了上述所有的思想方法，结合一些编程技巧，获得一种对于大规模矩阵乘法，计算较为快速的算法。无论是其耗时方面的表现还是在浮点帧率的表现都非常优秀。

美中不足的是在小数据规模下，该算法的时间并不优秀，所以最终实现函数时，会综合小数据规模下较为优秀的算法（即 `OpenMP` + 循环顺序优化）和本算法，实现一个较为快速的库函数。

4 组装矩阵乘法优化方案

4.1 正确性分析

4.1.1 矩阵最大误差

我们将使用 `OpenBLAS` 库计算生成的矩阵作为基准矩阵$B_{m\times n}$，设需要与其计算误差的待测矩阵为$X_{m\times n}$。

我们将待比较矩阵与基准矩阵各个位置数值相对误差的最大值，作为误差 ϵ，公式为 $\epsilon = \max\limits_{1\le i\le m, 1\le j\le n} \left|\frac{X_{i,j}-B_{i,j}}{B_{i,j}}\right|$。

4.1.2 代码实现

在 `matrix.h` 我们实现了一个 `float MatrixMaxRelativeError(Matrix *matrixBase, Matrix *matrixInput);` 函数用来计算输入矩阵 `matrixInput` 与基准矩阵 `matrixBase` 的相对误差。

使用如下代码求出最大相对误差：

```
1 MatrixMaxRelativeError(matmul_openBLAS(a, b), matmul_package_SIMD(a, b));
```

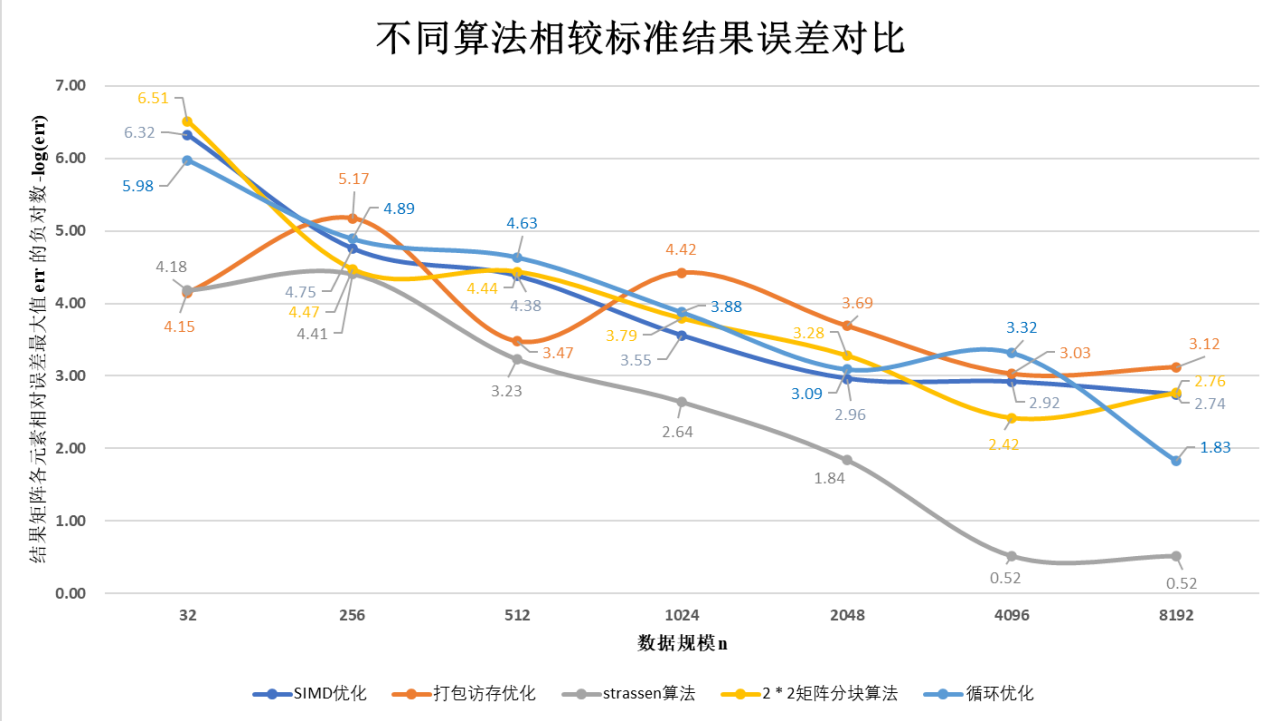
在 `matrix.h` 我们实现了一个 `Matrix *createMatrixRandom(const int rows, const int cols, const float leftBound, const float rightBound);` 矩阵创建函数，用来生成大小为 $rows \times cols$ 的矩阵，矩阵每个元素是 $(leftBound, rightBound]$ 的随机浮点数。

使用如下代码生成测试矩阵：

```
Matrix *a = createMatrixRandom(matrixSize, matrixSize, -1e5, 1e5);
Matrix *b = createMatrixRandom(matrixSize, matrixSize, -1e5, 1e5);
```

4.1.3 实验验证

对上述所有提出的优化方法都使用对应的函数进行计算，并求出误差。



4.1.4 实验结论

上述所有优化都能顺利执行，能在一定数据规模下相较于 `OpenBLAS` 库结果能保证一定程度的精确性。

上述优化算法中， `strassen` 算法精度较为欠缺，上面也分析过其原因。

算法处理的数据规模越大，精确程度略有下降，这是由于`float`精度造成的，但实验表明，该误差在可以接受的范围内。

4.2 速度分析

4.2.1 算法选择

为了避免小规模矩阵下使用 `OpenMP` 多线程导致的效率降低，我们在处理小规模矩阵时，只使用`SIMD`进行处理，而处理较大规模矩阵时，使用`打包优化 + OpenMP + SIMD` 进行处理。

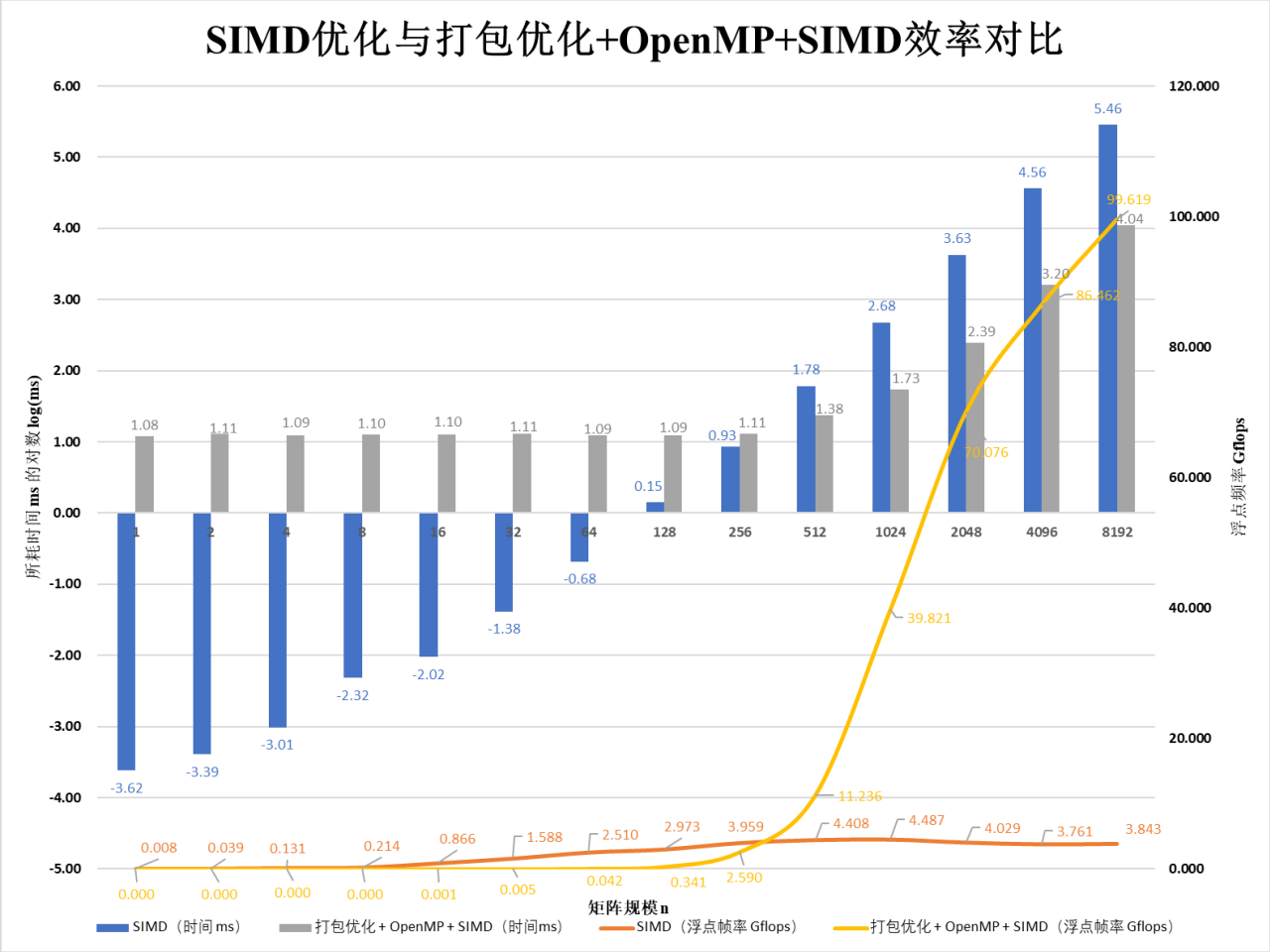
因此需要确定多小的矩阵使用前者，多大的矩阵使用后者，这需要进一步的实验。

4.2.2 代码实现

使用 `Matrix *matmul_plain_SIMD(Matrix *matrix1, Matrix *matrix2);` 使用`SIMD`优化方案。

使用 `Matrix *matmul_package_SIMD(Matrix *matrix1, Matrix *matrix2);` 使用`打包优化 + OpenMP + SIMD` 优化方案。

4.2.3 实验验证



4.2.4 实验结论

上述实验表明，在矩阵大小为 256 及以下使用 `SIMD` 优化，256 以上使用 `打包优化 + OpenMP + SIMD` 能达到较好的效果。

4.3 矩阵乘法优化方案

根据 4.2 的实验结果，我们实现 `Matrix *matmul_improved(Matrix *matrix1, Matrix *matrix2);`

- 当矩阵大小为 256 及以下，调用函数 `Matrix *matmul_plain_SIMD(Matrix *matrix1, Matrix *matrix2);` 使用 `SIMD` 优化方案计算。
- 当矩阵大小为 256 以上，`Matrix *matmul_package_SIMD(Matrix *matrix1, Matrix *matrix2);` 使用 `打包优化 + OpenMP + SIMD` 优化方案计算。

根据 4.1 的准确性说明，能够说明其相较于 `OpenBLAS`，具有较高的准确性。

5 与 OpenBLAS 比较

5.1 OpenBLAS 库的安装与使用

5.1.1 安装

使用 `git clone https://github.com/xianyi/OpenBLAS.git` 将库代码安装到本地

使用 `cd OpenBLAS && make` 进行构建

使用 `make PREFIX=/installation_directory install`，其中 `/installation_directory` 表示安装到的绝对路径。

删除安装文件 `OpenBLAS`

5.1.2 使用

在你上述安装目录`/installation_directory`下，你将会看到：`bin`、`include`、`lib`

在引用`cblas.h`库的时候，您需要引用`/installation_directory/include/cblas.h`。

在进行编译时候，您可能需要使用编译命令：

`gcc -static -o run xxx.c -I /installation_directory/include -L/installation_directory/lib -lopenblas -lpthread`

5.1.3 配合

为了使得 OpenBLAS 库 能和自定义的矩阵`Matrix`类配合起来，我在`matrix.h` 加入 `Matrix *matmul_openBLAS(Matrix *matrix1, Matrix *matrix2);` 函数，表示使用`OpenBLAS` 库中的 `cblas_sgemm` 进行计算。

5.2 正确性与效率测试

5.2.1 脚本编写

依据 `Project3` 中的要求，我们对如下特殊矩阵进行正确性测试，测试代码如下：

```
# include "matrix.h"
# include "matmul.h"
# include "time.h"
# include "stdio.h"
# include "stdlib.h"
# include "sys/time.h"
int main(int argc, char** argv) {
    srand(time(0));
    printf("total Test Case = %d\n", argc - 1);
    for (int i = 1; i < argc; i++) {
        struct timeval start,end; long timeuse;
        int matrixSize = atoi(*(argv + i));
        printf("-------------------------------------------------\n");
        printf("Case #%d: matrixSize = %d\n", i,  matrixSize);
        Matrix *a = createMatrixRandom(matrixSize, matrixSize, -1e5, 1e5);
        Matrix *b = createMatrixRandom(matrixSize, matrixSize, -1e5, 1e5);
        Matrix *c_openblas, *c_improve;
        gettimeofday(&start, NULL);
        c_openblas = matmul_openBLAS(a, b);
        gettimeofday(&end, NULL);
        timeuse = 1000000*(end.tv_sec - start.tv_sec) + end.tv_usec-start.tv_usec;
        printf("time_openBLAS: %.3f ms\n", timeuse / 1000.0);
        gettimeofday(&start, NULL);
        c_improve = matmul_improved(a, b);
        gettimeofday(&end, NULL);
        timeuse = 1000000*(end.tv_sec - start.tv_sec) + end.tv_usec-start.tv_usec;
        printf("time_improve: %.3f ms\n", timeuse / 1000.0);
```

```
28          printf("maximum relative error: %.2f%%\n", 100.0 *
    MatrixMaxRelativeError(c_openblas,c_improve));
29      }
30      printf("-----------------------------------------------------\n");
31      return 0;
```

5.2.2 脚本编译

使用编译命令：`gcc -O0 -o matmulTest matmul.c matrix.c test.c -I/installation_directory/include -L/installation_directory/lib -lopenblas -mavx512f -fopenmp` 进行脚本编译。

- 其中 `-O0` 是编译优化参数，有`-O0, -O1, -O2, -O3` 四级优化可选。
- 其中 `/installation_directory` 表示前面将`OpenBLAS`安装到的绝对路径。
- 其中 `-mavx512f` 是进行向量化运算指令集的相关编译参数。

5.2.3 脚本运行

使用命令：`./matmulTest 16 128 1000 8000 64000`，运行脚本。

- 其中 `matmulTest` 表示前面编译产生的可执行文件。
- 其中 `16 128 1000 8000 64000` 为测试列表，表示随机生成$(10^5, 10^5]$ 浮点数的方阵，进行测试。

5.3 测试结果

使用 `-O3` 进行优化，即使用`gcc -O3 -o matmulTest matmul.c matrix.c test.c -I/installation_directory/include -L/installation_directory/lib -lopenblas -mavx512f -fopenmp` 进行脚本编译。

使用命令：`./matmulTest 16 128 1000 8000 64000`，运行脚本，结果如下：

```
(base) daoan@cvip-NULL:~/beginer_project/CppLearning/SUSTech_cpp_Project04_matrix_multiplication_in_C/src$ ./matmulTest 16 128 1000 8000 64000
total Test Case = 5
-----------------------------------------------------
Case #1: matrixSize = 16
time of matmul_openBLAS: 0.042 ms
time of matmul_improve: 0.044 ms
maximum relative error: 0.00%
-----------------------------------------------------
Case #2: matrixSize = 128
time of matmul_openBLAS: 1.944 ms
time of matmul_improve: 1.268 ms
maximum relative error: 0.00%
-----------------------------------------------------
Case #3: matrixSize = 1000
time of matmul_openBLAS: 77.711 ms
time of matmul_improve: 104.683 ms
maximum relative error: 0.02%
-----------------------------------------------------
Case #4: matrixSize = 8000
time of matmul_openBLAS: 1050.309 ms
time of matmul_improve: 3316.838 ms
maximum relative error: 0.05%
-----------------------------------------------------
Case #5: matrixSize = 64000
time of matmul_openBLAS: 270111.072 ms
time of matmul_improve: 1402219.828 ms
maximum relative error: 0.87%
-----------------------------------------------------
```

使用表格表示上述数据：

矩阵大小	mat_openBLAS 时间（ms）	mat_improve 时间 （ms）	相对误差（%）
16×16	0.042	0.044	0.00%
128×128	1.944	1.268	0.00%
$1k \times 1k$	77.711	104.683	0.02%
$8k \times 8k$	1050.309	3316.838	0.05%
$64k \times 64k$	270111.072	1402219.828	0.87%

5.4 结果分析

5.4.1 精度分析

从上面的测试结果可以看出，分别使用 $16 \times 16, 128 \times 128, 1k \times 1k, 8k \times 8k, 64k \times 64k$ 五个矩阵进行测试，`matmul_improve()` 函数与 `matmul_openBLAS()` 函数计算矩阵乘法的结果，最大元素相对误差值，小于 1%，说明本项目实现的算法，在精度上可以达到相关要求。

5.4.2 效率分析

从上面的测试结果可以看出，分别使用 $16 \times 16, 128 \times 128, 1k \times 1k, 8k \times 8k, 64k \times 64k$ 五个矩阵进行测试，`matmul_improve()` 函数与 `matmul_openBLAS()` 函数计算矩阵乘法运行时间相比，均在同一数量级内，可以说明本项目实现的算法，在效率上较高。

6 ARM 平台 和 x86 平台执行效率对比

6.1 实验过程

本代码支持 `x86` 平台和 `arm` 平台的跨平台执行，前者使用`SIMD`系列指令集进行优化，后者使用`NEON`系列指令集进行优化，都使用`openMP` 进行多线程优化。

我们分别在两个平台，三个设备上进行实验，三台设备的环境配置如下：

- x86 架构 PC 机

项目	信息
架构	x86_64
CPU	1 个 8 核 CPU， AMD Ryzen 7 5800H with Radeon Graphics @ 3.20 GHz
内存	16 GB
高速缓存	512 KB （L1 Cache） 4.0 MB （L2 Cache）16.0 KB （L3 Cache）

- x86 服务器 Ser

项目	信息
架构	x86_64
CPU	4 个 14 核 CPU， Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GH
内存	512 GB
高速缓存	896 KB （L1 Cache）28.0 MB（L2 Cache）38.5 MB（L3 Cache）

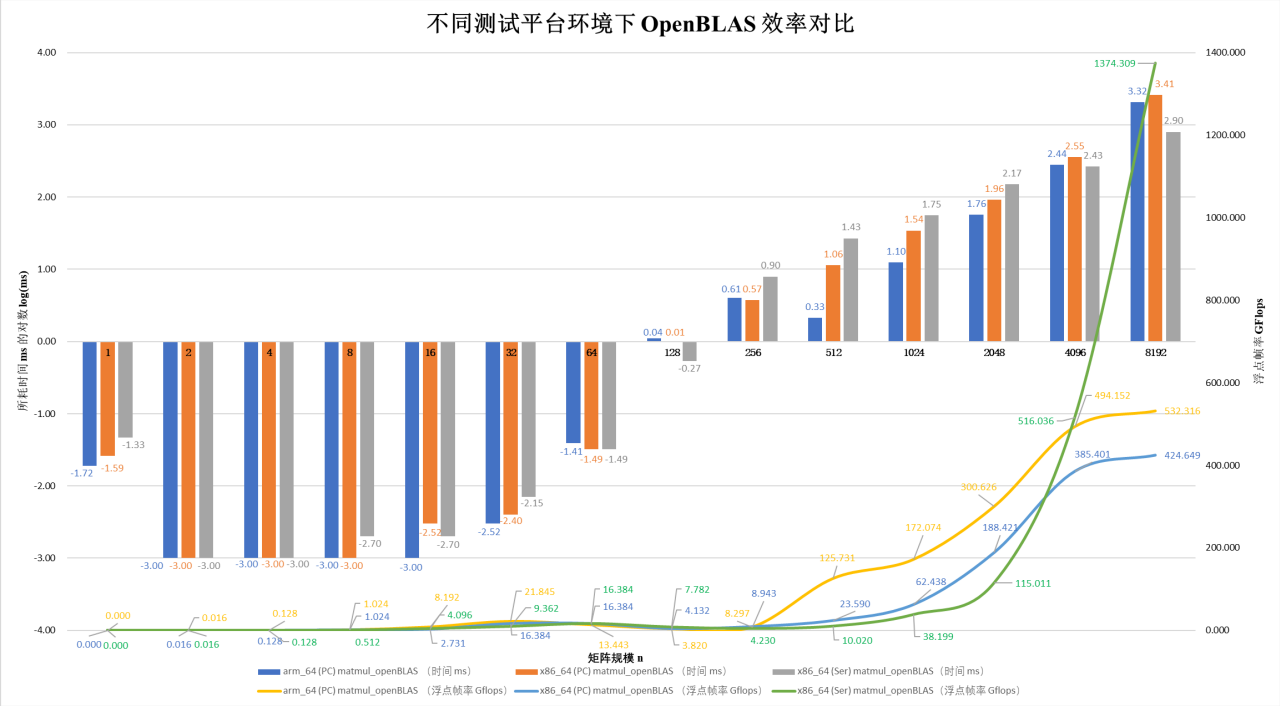
- arm 架构 PC 机

项目	信息
架构	arm_64
CPU	1 个 8 核 CPU，型号为 Apple M1 Pro @ 3.35 GHz
内存	16 GB
高速缓存	192 KB（L1 Cache） 4.0 MB （L2 Cache）

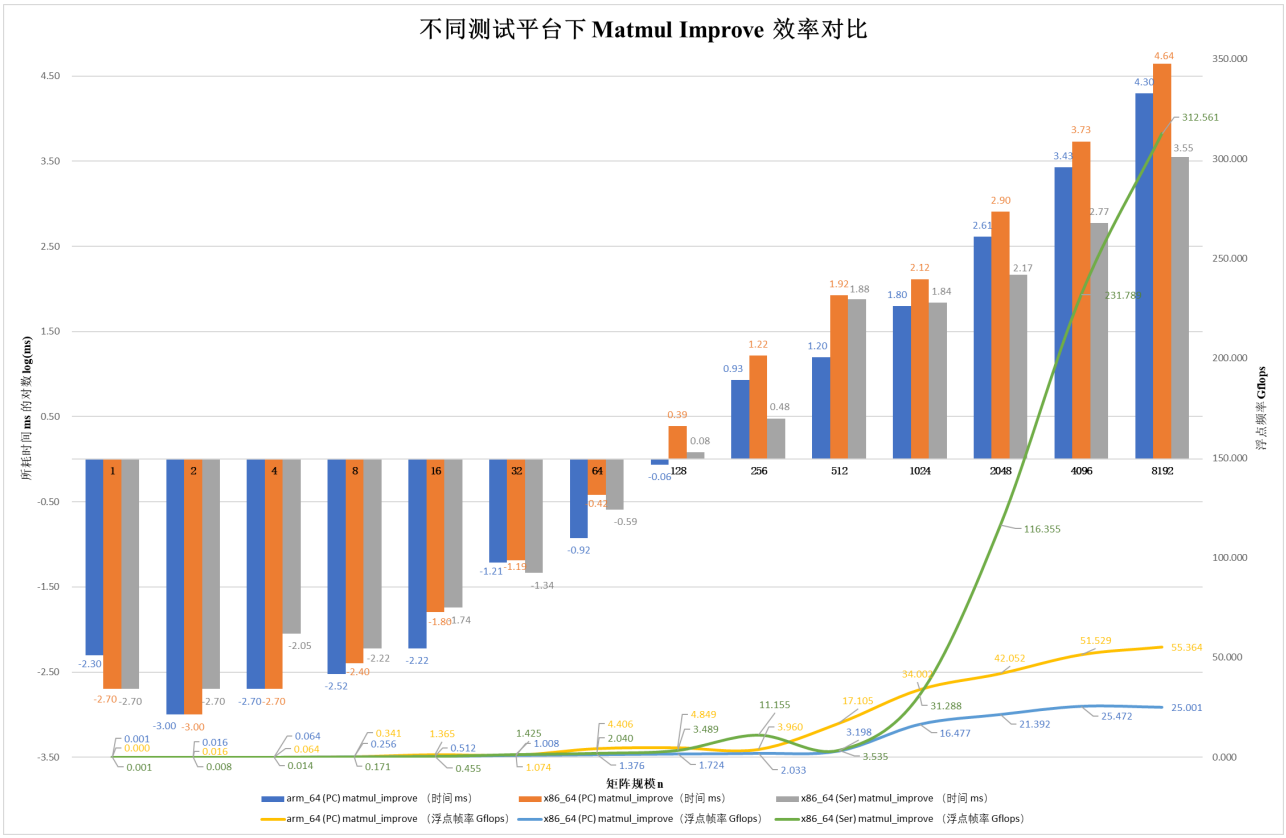
分别测得 `OpenBLAS` 和 `Matmul Improve` 两个库的运行效率（包括运行时间，计算浮点帧率）

6.2 实验验证

- 使用 `OpenBLAS` 在上述两个平台、三个设备上进行实验，开启 `O3` 优化。



- 使用 `Matmul Improve` 在上述两个平台、三个设备上进行实验，开启 `O3` 优化。



6.3 实验结论

- 无论使用 `OpenBLAS` 还是 `Matmul Improve`，较小数据规模的矩阵进行计算时，在 `arm` 平台上进行矩阵乘法的效率较高，而在 `x86` 尤其是核心数相对较多的平台上，效率较低；较大数据规模的矩阵进行计算时，多线程起的作用明显，CPU核心数越多效率显著越高，同等核心数下，`arm` 平台效率略高于 `x86` 平台。
- 不同测试平台下，`OpenBLAS` 和 `Matmul Improve` 都对矩阵乘法做了较大的优化，并且在较少核心数的 `x86` 或者是 `arm` 平台上出现了瓶颈现象，其浮点帧率趋于稳定，处于同一数量级上。

7 项目及代码

这个项目已经放在 `GitHub` 上开源了，您可以在 `src` 中访问我的代码，本项目主要将上述所有的源代码都在 `matmul.c` 和 `matmul.h` 中给出，矩阵库依旧复用 `project3` 的矩阵库。

项目链接：https://github.com/Maystern/SUSTech_cpp_Project04_matrix-multiplication-in-C。

项目推荐在 `Ubuntu` 环境下运行。

1. 使用命令 `git clone https://github.com/Maystern/SUSTech_cpp_Project04_matrix-multiplication-in-C.git` 将项目下载到当前目录。

```
问题  输出  调试控制台  终端  JUPYTER

(base) root@MaysternLaptop:/home/cpp_fall2022# git clone https://github.com/Maystern/SUSTech_cpp_Project04_matrix-multiplication-in-C.git
Cloning into 'SUSTech_cpp_Project04_matrix-multiplication-in-C'...
remote: Enumerating objects: 198, done.
remote: Counting objects: 100% (198/198), done.
remote: Compressing objects: 100% (120/120), done.
remote: Total 198 (delta 79), reused 178 (delta 67), pack-reused 0
Receiving objects: 100% (198/198), 12.44 MiB | 6.16 MiB/s, done.
Resolving deltas: 100% (79/79), done.
(base) root@MaysternLaptop:/home/cpp_fall2022#
```

2. 在当前目录下执行 `cd SUSTech_cpp_Project04_matrix-multiplication-in-C` 进入项目根目录。

```
问题  输出  调试控制台  终端  JUPYTER

(base) root@MaysternLaptop:/home/cpp_fall2022# cd SUSTech_cpp_Project04_matrix-multiplication-in-C
(base) root@MaysternLaptop:/home/cpp_fall2022/SUSTech_cpp_Project04_matrix-multiplication-in-C#
```

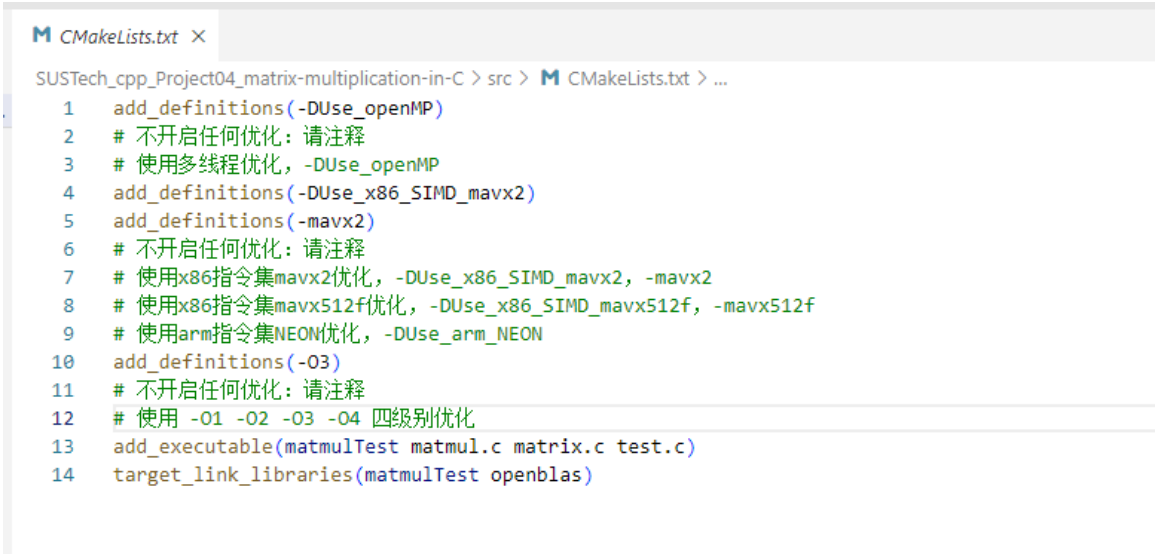
3. 在项目根目录执行`sh Run_Matmul.sh`命令，该命令执行后，将在`./build`中自动使用`cmake`编译原代码文件，并将位于`./build/bin`目录下的生成的二进制可执行程序`matmulTest`移动至项目根目录处。

```
问题  输出  调试控制台  终端  JUPYTER

(base) root@MaysternLaptop:/home/cpp_fall2022/SUSTech_cpp_Project04_matrix-multiplication-in-C# sh Run_Matmul.sh
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
libopenblas-dev is already the newest version (0.3.20+ds-1).
0 upgraded, 0 newly installed, 0 to remove and 25 not upgraded.
-- The C compiler identification is GNU 11.2.0
-- The CXX compiler identification is GNU 11.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Warning (dev) in CMakeLists.txt:
  No cmake_minimum_required command is present.  A line of code such as

    cmake_minimum_required(VERSION 3.22)

  should be added at the top of the file.  The version specified may be lower
  if you wish to support older CMake versions for this project.  For more
  information run "cmake --help-policy CMP0000".
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /home/cpp_fall2022/SUSTech_cpp_Project04_matrix-multiplication-in-C/build
[ 25%] Building C object bin/CMakeFiles/matmulTest.dir/matmul.o
[ 50%] Building C object bin/CMakeFiles/matmulTest.dir/matrix.o
[ 75%] Building C object bin/CMakeFiles/matmulTest.dir/test.o
[100%] Linking C executable matmulTest
[100%] Built target matmulTest
(base) root@MaysternLaptop:/home/cpp_fall2022/SUSTech_cpp_Project04_matrix-multiplication-in-C#
```

4. 您可以在项目根目录中，使用`./matmulTest matmulSizeLists`运行所有$matmulSize \times matmulSize$大小的矩阵乘法，作为例子。

```
问题  输出  调试控制台  终端  JUPYTER

(base) root@MaysternLaptop:/home/cpp_fall2022/SUSTech_cpp_Project04_matrix-multiplication-in-C# ./matmulTest 10 100 500 1000
total Test Case = 4
------------------------------------------------------
Case #1: matrixSize = 10
time of matmul_openBLAS: 0.025 ms
time of matmul_improve: 0.002 ms
maximum relative error: 0.00%
------------------------------------------------------
Case #2: matrixSize = 100
time of matmul_openBLAS: 4.186 ms
time of matmul_improve: 0.190 ms
maximum relative error: 0.00%
------------------------------------------------------
Case #3: matrixSize = 500
time of matmul_openBLAS: 1.208 ms
time of matmul_improve: 15.480 ms
maximum relative error: 0.00%
------------------------------------------------------
Case #4: matrixSize = 1000
time of matmul_openBLAS: 15.518 ms
time of matmul_improve: 106.621 ms
maximum relative error: 0.00%
------------------------------------------------------
```

5. 您可以在 `src` 下的 `CMakeLists.txt` 中选择是否开启多线程优化、指令集优化、编译优化，需要重新 `build`。



8 总结

这个项目还是非常有挑战性的，也非常有意思。

其中最有意思的事情在于，我们仅仅通过了访存优化，而不涉及到新的算法，就能使得程序的效率提高几千倍。这也让我更有理由相信，硬件地革新非常有助于人类科技的发展。

当然，本次项目也借鉴和参考了一些课程和文件：

- 于仕琪老师的 B 站账号中的系列视频课程。
- 澎峰科技开启算力时代 的 B 站账号中关于 OpenBLAS 优化方案的系列讲座。
- BLISlab 系列实验课程：加速矩阵乘法 https://github.com/flame/blislab。
- 矩阵乘法加速原理：A Sandbox for Optimizing GEMM https://github.com/flame/blislab/blob/master/tutorial.pdf。

这次的项目，真正实现了从零开始，由自己编写所有代码，去不断更新、迭代，尝试更优的做法，并且做实验，让自己的代码效率提高了数千倍，非常有成就感。

这次项目，从学期开始的 4 个项目中内容最多的一个，我也花了很多时间和精力去写代码、做实验、撰写报告，所以报告内容较多，也希望老师能理解。

当然，因为时间原因，本项目还有很多地方需要完善，还希望老师能多多指点。

谢谢您看到最后！