

CS205: Project 3: Improved Matrix Multiplication in C

杨彦卓 (南方科技大学, 通识与学科基础部)

2024.04.26

摘 要

该文章从基础的暴力三重循环算法计算矩阵乘法入手, 尝试从并行, 多线程, 分块等多方面优化大规模的矩阵乘法运算。最后引入 OpenBLAS 库对矩阵乘法的优化, 并与之对比, 分析结果差异。

关键词: 矩阵乘法、大规模、算法优化

目录

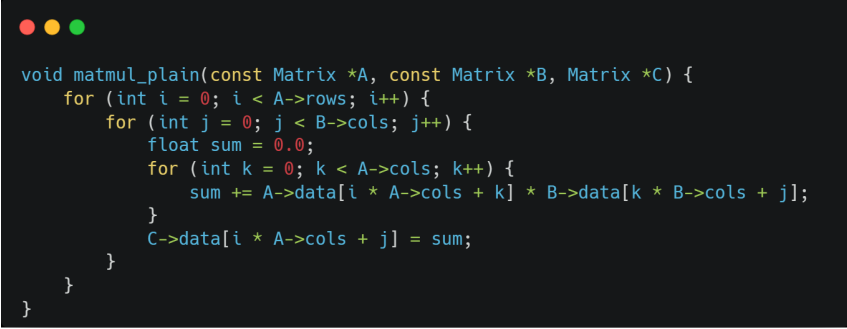
1	绪论	1
1.1	引言	1
1.2	声明	1
1.2.1	Linux Server 参数	1
1.2.2	特别声明	2
1.2.3	编译模式选择	2
1.2.4	函数参数合法检查	2
2	对于编译模式的改进	2
2.1	SIMD 指令集	2
2.1.1	SIMD 指令集优化原理	2
2.1.2	SIMD: 代码实现	4
2.1.3	SIMD: 改进算法与结果分析	5
2.2	OpenMP	6
2.2.1	OpenMp: 优化原理	6
2.2.2	OpenMP: 代码实现	7
2.2.3	OpenMP: 结果分析与改进	7
3	对于算法原理的改进	8
3.1	分块矩阵乘法	8
3.1.1	分块矩阵: 算法原理	8
3.1.2	分块矩阵: 再优化与实验结果	8
4	对比 OpenBLAS	10
4.1	OpenBLAS: 简介	10
4.2	OpenBLAS: 实验结果与对比分析	11
5	局限性分析	11
5.1	对于时间预测的局限性	12
5.2	手写函数效率的局限性	12
6	结语	13

6.1	其他方法尝试	13
6.2	心得	14

1 绪论

1.1 引言

由线性代数的矩阵乘法原理，我们很容易地得到一个三重循环的算法来计算矩阵乘法的乘积，具体实现步骤如下图：



```
void matmul_plain(const Matrix *A, const Matrix *B, Matrix *C) {
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->cols; j++) {
            float sum = 0.0;
            for (int k = 0; k < A->cols; k++) {
                sum += A->data[i * A->cols + k] * B->data[k * B->cols + j];
            }
            C->data[i * A->cols + j] = sum;
        }
    }
}
```

Figure 1: 三层循环暴力求解矩阵乘积的代码示例

这个算法的时间复杂度是 $O(n^3)$ 。当矩阵的尺寸达到 $64k * 64k$ 时，我们可以粗略地估计一下运行这个算法所需要的时间：该算法的循环需要迭代 2^{48} 次，即使是把每次乘法运算所需的时钟周期数 (Cycles) 视为 1，忽略处理器架构、内存访问，指令跳转，写入内存等等操作所花费的时间，至少也需要 30h。

这样的速度显然不是我们想要的结果，所以本次 Project 旨在尽可能提高计算矩阵乘法的效率。

另外，由于暴力算法的时间开销巨大，会占用大量的公共 Linux Server 资源，所以本次 Project 在实验阶段不再测量暴力算法的时间开销。

1.2 声明

1.2.1 Linux Server 参数

- CPU: Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, 24 Cores
- Memory: 128GB
- GPU: NVIDIA GeForce RTX 2080 Ti x 4
- OS: Ubuntu 22.04.4
- GCC: 11.4.0

-
- make: 4.3
 - cmake: 3.22.1

1.2.2 特别声明

- 在本次实验报告中，除非另有说明，所有单位术语 1k 都表示 $1024(2^{10})$ 而不是传统的 1000。原因是在 SIMD 优化时，我们需要矩阵的尺寸尽量为 8, 16, 32 的倍数。
- 在实验过程中，除非另有说明，所有的矩阵皆为方阵。属性名词“尺寸”表示方阵的行数或列数。

1.2.3 编译模式选择

由于时间开销巨大，所有的编译模式都会带上-O3 字段来优化编译结果。-O3 编译模式的优化原理在上一次 Project 中有所提及，此处仅做简要复述：-O3 提供了更积极的代码优化策略，如：编译器试图在循环内展开迭代以减少循环控制的开销，提高循环执行的效率。

1.2.4 函数参数合法检查

所有的初始化矩阵方法以及矩阵乘法方法，在实际运行之前都进行了参数合法性检查，但由于篇幅有限，在该报告的展示中均为列出。具体检查方式可查看 matrix.c 文件。

2 对于编译模式的改进

2.1 SIMD 指令集

2.1.1 SIMD 指令集优化原理

通过阅读百度百科的相关词条，以及 CSDN 等网站上提供的资料，我们得知单指令流多数据流 (Single Instruction Multiple Data) (SIMD) 是一种计算机处理技术，它允许一条指令同时处理多个数据点。这种方式在处理大量重复操作时非常有效。

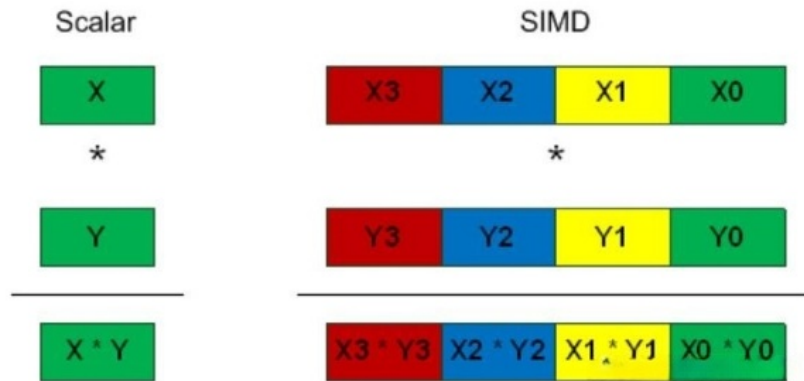


Figure 2: SIMD 原理示意图

在阅读了这些资料后，我尝试性地总结了 SIMD 优化运行速度的原理：

- **数据并行性:** SIMD 往往通过在单个操作中同时处理多个数据项来优化运行速度。在数据规模较大时，对于代码中每次迭代都会出现的单一操作，SIMD 将尝试在同一次迭代中多次重复该操作，从而达到减少迭代次数的效果。这意味着 B-type 和 R-type 指令这样的控制转移型指令数量减少。而这两种指令平均时钟周期 (CPI, Cycles Per Instruction) 通常都很高。尤其是当预测错误时。现代处理器使用分支预测来尝试优化这类指令的执行速度；然而，当预测失败时，处理器需要撤销错误的指令流，这会引入额外的周期。
- **提高 CPU 缓存和内存利用效率:** SIMD 指令通过同时处理多个数据点，可以更高效地利用缓存和内存带宽。当数据被加载到寄存器时，相同指令作用于所有这些数据，这减少了对内存的多次访问需求，从而降低了内存访问的瓶颈。

2.1.2 SIMD: 代码实现

```
void matmul_SIMD(const Matrix *A, const Matrix *B, Matrix *C) {
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->cols; j++) {
            __m256 sum = _mm256_setzero_ps();
            for (int k = 0; k < A->cols; k += 8) {
                __m256 a = _mm256_loadu_ps(&A->data[i * A->cols + k]);
                __m256 b = _mm256_loadu_ps(&B->data[k * B->cols + j]);
                sum = _mm256_add_ps(sum, _mm256_mul_ps(a, b));
            }
            float buffer[8];
            _mm256_storeu_ps(buffer, sum);
            float total = buffer[0] + buffer[1] + buffer[2] + buffer[3] +
                buffer[4] + buffer[5] + buffer[6] + buffer[7];
            C->data[i * C->cols + j] = total;
        }
    }
}
```

Figure 3: SIMD 指令集实现矩阵乘法

在暴力的算法中，每次循环迭代只处理一个元素的乘法和加法操作。根据 SIMD 的优化思路，通过使用 AVX 指令，每个操作同时处理 m 个浮点数。在该算法处理矩阵乘法时，一条 SIMD 指令完成的工作量相当于 `matmul_plain` 中 m 条指令的工作量。

那理论上是不是只要 m 能被 N 整除， m 值越大运行速度越快？实际并非如此，一般来说 AVX 指令集中是使用 256 位寄存器来加载和存储数据的，以此来达到每个寄存器同时处理 8 个单精度浮点数的效果（浮点数需要 8 个比特位）。所以说 m 的取值必然是有限的。

但我了解到 AVX 指令集同样支持 512 位寄存器来处理数据，这样每个寄存器可以同时处理 16 个单精度浮点数。于是改进了算法如下：

```
void matmul_SIMD_16(const Matrix *A, const Matrix *B, Matrix *C) {
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->cols; j++) {
            __m512 sum = _mm512_setzero_ps();
            for (int k = 0; k < A->cols; k += 16) {
                __m512 a = _mm512_loadu_ps(&A->data[i * A->cols + k]);
                __m512 b = _mm512_loadu_ps(&B->data[k * B->cols + j]);
                sum = _mm512_add_ps(sum, _mm512_mul_ps(a, b));
            }
            float buffer[16];
            _mm512_storeu_ps(buffer, sum);
            float total = 0;
            for (int n = 0; n < 16; n++) {
                total += buffer[n];
            }
            C->data[i * C->cols + j] = total;
        }
    }
}
```

Figure 4: 使用 SIMD 指令集同时处理 16 个浮点数

2.1.3 SIMD: 改进算法与结果分析

从上述两种算法提供的思路，我产生了这样的思考：能不能人为地让寄存器每次近似地处理更多浮点数？

我尝试创建两个独立的累加器，每个累加器每次近似同时处理 16 个浮点数，这样就能接近每次同时处理 32 个寄存器的效果。

```
void matmul_SIMD_32(const Matrix *A, const Matrix *B, Matrix *C) {
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->cols; j++) {
            __m512 sum0 = _mm512_setzero_ps();
            __m512 sum1 = _mm512_setzero_ps();
            for (int k = 0; k < A->cols; k += 32) {
                __m512 a0 = _mm512_loadu_ps(&A->data[i * A->cols + k]);
                __m512 a1 = _mm512_loadu_ps(&A->data[i * A->cols + k + 16]);
                __m512 b0 = _mm512_loadu_ps(&B->data[k * B->cols + j]);
                __m512 b1 = _mm512_loadu_ps(&B->data[(k + 16) * B->cols + j]);
                sum0 = _mm512_add_ps(sum0, _mm512_mul_ps(a0, b0));
                sum1 = _mm512_add_ps(sum1, _mm512_mul_ps(a1, b1));
            }
            sum0 = _mm512_add_ps(sum0, sum1);
            float buffer[16];
            _mm512_storeu_ps(buffer, sum0);
            float total = 0.0f;
            for (int n = 0; n < 16; n++) {
                total += buffer[n];
            }
            C->data[i * C->cols + j] = total;
        }
    }
}
```

Figure 5: 使用 SIMD 指令集近似同时处理 32 个浮点数

我们选用了这些矩阵尺寸，分别对以上三种算法的运行时间进行测量：{32, 128, 1k, 2k, 4k, 8k, 64k}。但在实际运行的过程中，我们发现第一个算法 (matmul_SIMD) 在矩阵尺寸为 4k 时，其运行时间就已经达到了 50s。在之后的时间测量中舍弃第一个算法，并得到如下时间数据：

Size	SIMD_16 Time Cost	SIMD_32 Time Cost
1024	0.57s	0.66s
2048	3.89s	3.92s
4196	24.65s	27.83s
8192	412.99s	256.15s

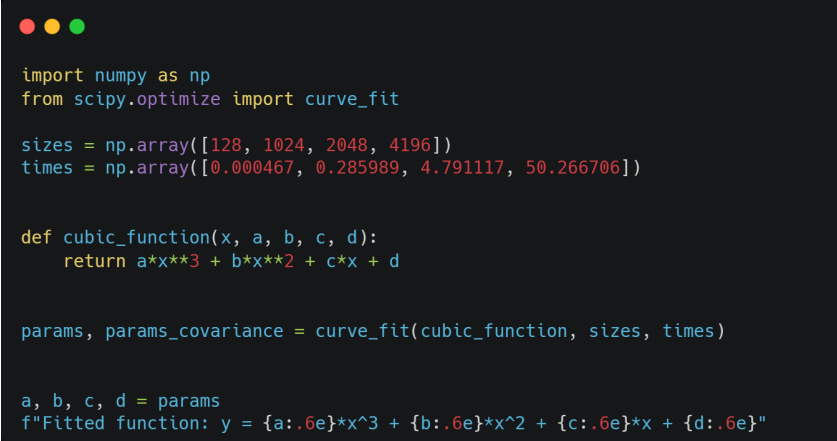
Table 1: 矩阵乘法在不同尺寸下两种算法耗时对照表

根据已有的数据与以下 Python 代码模板，我们可以粗略地得到运行时间关于矩阵尺寸的函数表达式：

$$y_1 = 9.77 * 10^{-9}x^3 - 3.62 * 10^{-7}x^2 - 1.94 * 10^{-4}x + 2.96 * 10^{-2}$$

$$y_2 = 1.70 * 10^{-9}x^3 - 1.04 * 10^{-5}x^2 + 2.26 * 10^{-2}x - 13.50$$

$$y_3 = 6.95 * 10^{-10}x^3 - 2.54 * 10^{-6}x^2 + 5.89 * 10^{-3}x - 3.45$$



```

import numpy as np
from scipy.optimize import curve_fit

sizes = np.array([128, 1024, 2048, 4196])
times = np.array([0.000467, 0.285989, 4.791117, 50.266706])

def cubic_function(x, a, b, c, d):
    return a*x**3 + b*x**2 + c*x + d

params, params_covariance = curve_fit(cubic_function, sizes, times)

a, b, c, d = params
f"Fitted function: y = {a:.6e}*x^3 + {b:.6e}*x^2 + {c:.6e}*x + {d:.6e}"

```

Figure 6: 利用 Python 预测运行时间关于矩阵尺寸的函数表达式

根据该函数表达式进行预测，当矩阵尺寸为 64k 时，三个算法的运行时间各自大概需要 763h, 120h, 51h。这样的速度仍然不够理想。不过这样的结果至少证明了：人为改进，使得寄存器近似同时处理 32 个浮点数的效率是更高的。

不过问题也相当明显：

- 当矩阵尺寸不是 32 的倍数时，我们需要额外处理边界情况；
- 当相乘的矩阵不是方阵时，需要更复杂的边界处理；
- 该算法对内存访问模式和缓存行对齐都有更高的要求。

2.2 OpenMP

2.2.1 OpenMp: 优化原理

OpenMP (Open Multi-Processing) 是一种用于多处理器编程的应用程序接口，主要用于编写在共享内存多处理器机器上运行的高性能并行程序。矩阵乘法是一个计算密集型任务，矩阵中的每个元素的计算都是独立的。我认为正是这样的独立性这使得矩阵乘法成为并行计算的理想选择。

2.2.2 OpenMP: 代码实现

使用 OpenMP，在最外层循环 (遍历 A 的行或 B 的列) 通过添加并行指令并行化矩阵乘法。如果在外层循环使用 `#pragma omp parallel for` 时，OpenMP 运行时将会自动把这些行或列的计算任务分配给多个线程，每个线程负责计算结果矩阵 C 的一部分。代码实现如下：

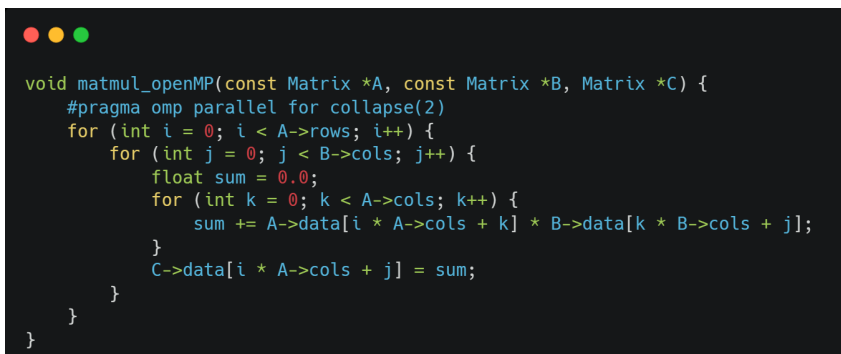


Figure 7: OpenMP 实现矩阵乘法

2.2.3 OpenMP: 结果分析与改进

然而直接使用这个 `matmul_OpenMP` 方法计算矩阵乘法的结果不尽如人意，因为矩阵尺寸为 4k 时，运行时间大约需要 70s，这个结果显然不如我们之前改进的 SIMD 算法实现。

我认为可以根据上文提到的 SIMD 优化思路对其进行进一步优化——将两者结合，在 SIMD 每次处理多个浮点数的基础之上，再应用 OpenMP 的并行处理：

```

void matmul_SIMD_openMP(const Matrix *A, const Matrix *B, Matrix *C) {
    #pragma omp parallel for
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->cols; j++) {
            __m512 sum0 = _mm512_setzero_ps();
            __m512 sum1 = _mm512_setzero_ps();
            for (int k = 0; k < A->cols; k += 32) {
                __m512 a0 = _mm512_loadu_ps(&A->data[i * A->cols + k]);
                __m512 a1 = _mm512_loadu_ps(&A->data[i * A->cols + k + 16]);
                __m512 b0 = _mm512_loadu_ps(&B->data[k * B->cols + j]);
                __m512 b1 = _mm512_loadu_ps(&B->data[(k + 16) * B->cols + j]);
                sum0 = _mm512_add_ps(sum0, _mm512_mul_ps(a0, b0));
                sum1 = _mm512_add_ps(sum1, _mm512_mul_ps(a1, b1));
            }
            sum0 = _mm512_add_ps(sum0, sum1);
            float buffer[16];
            _mm512_storeu_ps(buffer, sum0);
            float total = 0.0f;
            for (int n = 0; n < 16; n++) {
                total += buffer[n];
            }
            C->data[i * C->cols + j] = total;
        }
    }
}

```

Figure 8: 结合 SIMD 和 OpenMD 实现矩阵乘法

运行速度得到了显著提升，这是我们得到的函数表达式：

$$y = 5.27 * 10^{-10}x^3 - 3.36 * 10^{-6}x^2 + 4.70 * 10^{-3}x - 0.90$$

两者结合得到了更快的速度，但当矩阵尺寸达到 64k 时，该算法预计仍需要 37.2h。这样治标不治本的优化措施始终无法达到预期，我们应该把优化重心转向算法原理。

3 对于算法原理的改进

3.1 分块矩阵乘法

3.1.1 分块矩阵：算法原理

分块矩阵乘法是一种优化矩阵乘法计算的技术，它通过将大型矩阵分解成更小的子矩阵 (块) 来改善计算过程的内存访问模式。它的每次计算都尽可能地使用缓存中的数据，减少对慢速主内存的访问。当一个矩阵块被加载到缓存中后，所有的计算都可以直接利用这些数据，这样可以显著减少数据传输的开销。

3.1.2 分块矩阵：再优化与实验结果

该算法涉及六个循环：

定义三个外层循环的每个循环步长是 `BLOCK_SIZE`，这表示在进行乘法计算时，矩阵被切分为了 `BLOCK_SIZE x BLOCK_SIZE` 的小块。我们根据实际情况 (考虑最终需要处理矩阵尺寸为 64k 的情形)，将 `BLOCK_SIZE` 调整为 64。

定义三个内层循环对每一个确定的块进行具体的乘法操作。这些循环实现了矩阵块内部的乘法累加，即 $C_{i0,j0} += A_{i0,k0} \times B_{k0,j0}$

```
void matmul_block(const Matrix *A, const Matrix *B, Matrix *C) {
    int i, j, k, i0, j0, k0;
    for (i = 0; i < A->rows; i += BLOCK_SIZE) {
        for (j = 0; j < B->cols; j += BLOCK_SIZE) {
            for (k = 0; k < A->cols; k += BLOCK_SIZE) {
                for (i0 = i; i0 < i + BLOCK_SIZE && i0 < A->rows; i0++) {
                    for (j0 = j; j0 < j + BLOCK_SIZE && j0 < B->cols; j0++) {
                        float sum = C->data[i0 * C->cols + j0];
                        for (k0 = k; k0 < k + BLOCK_SIZE && k0 < A->cols; k0++) {
                            sum += A->data[i0 * A->cols + k0] * B->data[k0 * B->cols + j0];
                        }
                        C->data[i0 * C->cols + j0] = sum;
                    }
                }
            }
        }
    }
}
```

Figure 9: 分块矩阵算法实现矩阵乘法

同样地，我们结合之前介绍的 SIMD 和 OpenMP 优化思路对普通的分块矩阵乘法进行再优化。

```
void matmul_SIMD_openMP_block(const Matrix *A, const Matrix *B, Matrix *C) {
    int i, j, k, ii, jj, kk;

    #pragma omp parallel for private(i, j, k, ii, jj, kk) collapse(2) schedule(static)
    for (i = 0; i < A->rows; i += BLOCK_SIZE) {
        for (j = 0; j < A->cols; j += BLOCK_SIZE) {
            for (k = 0; k < B->cols; k += BLOCK_SIZE) {
                for (ii = i; ii < i + BLOCK_SIZE && ii < A->rows; ii++) {
                    for (jj = j; jj < j + BLOCK_SIZE && jj < A->cols; jj++) {
                        __m512 sum0 = _mm512_setzero_ps();
                        for (kk = k; kk < k + BLOCK_SIZE && kk < B->cols; kk += 16) {
                            __m512 a0 = _mm512_loadu_ps(&A->data[ii * A->cols + kk]);
                            __m512 b0 = _mm512_loadu_ps(&B->data[kk * B->cols + jj]);
                            sum0 = _mm512_add_ps(sum0, _mm512_mul_ps(a0, b0));
                        }
                        __m512 c0 = _mm512_loadu_ps(&C->data[ii * C->cols + jj]);
                        c0 = _mm512_add_ps(c0, sum0);
                        _mm512_storeu_ps(&C->data[ii * C->cols + jj], c0);
                    }
                }
            }
        }
    }
}
```

Figure 10: 同时利用 SIMD 与 OpenMD 再优化

显然，分块算法在矩阵尺寸较大、较规整 (指尺寸为 256 的倍数) 时效果最好。我们选定如下尺寸进行时间测量：{512, 1024, 2048, 4196, 8192, 16384, 65536}。我们结合之前的 `matmul_SIMD_OpenMP` 的数据进行对比，得到的结果如下：

Matrix Size	matmul_SIMD_OpenMP	matmul_SIMD_OpenMP_block
512	0.03s	0.094s
1024	0.155s	0.163s
2048	1.36s	1.291s
4196	10.551s	10.943s
8192	198.423s	85.917s
16384	—	746.373s
65536	37.2h (预计)	14.4h (预计)

Table 2: 矩阵乘法在不同尺寸下分块与否耗时对照表

4 对比 OpenBLAS

4.1 OpenBLAS: 简介

通过阅读百度百科的相关词条，以及 CSDN 等网站上提供的资料，我们得知 OpenBLAS 是一个优化的基础线性代数库 (Basic Linear Algebra Subprograms)，它利用了多种算法和优化技术来加速是矩阵运算。OpenBLAS 的算法选择依赖于多个因素，简单总结如下：

- 体系结构特定优化：OpenBLAS 对不同的处理器架构进行了特定的优化，具体情况尚未了结；
- 多线程和并行化：OpenBLAS 使用多线程来加速计算，并在可能的情况下利用处理器的多核特性；对于支持 SIMD 的处理器，如使用 AVX、AVX2 或 AVX-512 指令集进行矩阵操作的向量化；
- 缓存优化：OpenBLAS 进行算法调整以优化缓存使用，如使用分块矩阵乘法（blocking）技术来增强数据在缓存中的局部性。

简要了解了之后才发现，除了体系结构特定优化之外，OpenBLAS 的优化思路和我们现有的思路如此惊人的相似！

以下内容尝试实现调用 OpenBLAS 库函数测量矩阵乘法运行时间并分析结果差异。

4.2 OpenBLAS：实验结果与对比分析

用命令行运行以下指令来获取该库：`git clone https://github.com/OpenMathLib/OpenBLAS.git`
用以下代码实现，得到结果：

```
void matmul_openBLAS(const Matrix *A, const Matrix *B, Matrix *C) {
    int M = A->rows;
    int K = A->cols;
    int N = B->cols;

    float alpha = 1.0;
    float beta = 0.0;

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                M, N, K,
                alpha, A->data, K,
                B->data, N,
                beta, C->data, N);
}
```

Figure 11: OpenBLAS 实现矩阵乘法

Matrix Size	matmul_openBLAS	matmul_SIMD_OpenMP_block
512	0.008s	0.094s
1024	0.039s	0.163s
2048	0.047s	1.291s
4196	0.495s	10.943s
8192	1.012s	85.917s
16384	5.883s	746.373s
65536	603.22s	14.4h(预计)

Table 3: 矩阵乘法在不同尺寸下调用 OpenBLAS 库与手写算法耗时对照表

5 局限性分析

手写函数的效率和库函数相比有很大差异。但相较于初代还是有很大的提升。

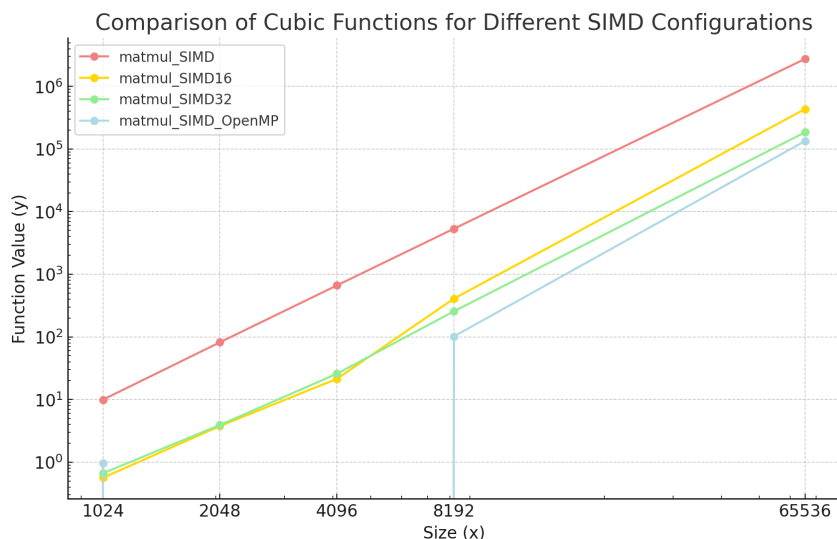


Figure 12: 手写函数的运行时间对比图

后文将分析本次实验的局限性。

5.1 对于时间预测的局限性

我们知道所有的矩阵乘法时间复杂度都是 $O(n^3)$ 的，在得到数据后我们对其进行拟合，并根据函数表达式预测了后续函数值。但这些数据点数量几乎都聚集在一个小区域 (128 至 8k，很少到 16k) 内，模型可能无法准确预测远离该区域的值 (64k)。这种情况下对于拟合的区间之外的预测（外推），三次多项式可能会表现得不稳定。

并且实际数据往往包含噪声，实际情况中影响运行时间的因素有很多。此次的运行环境是 Linux Server，在使用人数过多、受网络波及时等等情况下很可能会数据的参考性产生影响，从而对拟合的准确性产生影响。异常值对高阶多项式的影响尤为显著，可能导致模型拟合得很糟糕。

所以我们对运行时间在尺寸为 64k 时的预测实践上是不准确的。预测结果的参考价值更多的在于，相对于其他算法，当前是否得到了优化。

5.2 手写函数效率的局限性

经过查询资料与分析，我认为手写的函数尽管效率提升了很多，但相对与 OpenBLAS 来说还是有很大差距的原因包括但不限于以下几点。

- 精细化：据了解，OpenBLAS 根据具体的 CPU 架构优化其算法，包括深度优化的向量化处理和多线程控制，以充分利用现代 CPU 的所有功能；

-
- 分块大小选择：OpenBLAS 对于块大小的选择经过精心调整以匹配 CPU 的缓存系统。在手写的代码中，很难考虑到对于某一个具体的尺寸来说，如何调整块大小会使得效率达到最优。而不合适的块大小可能会导致缓存未命中率增高，从而影响性能。而经过实验，简单地调整块大小在提高效率方面并不显著，并且目前选择的块大小已经是相对较好的值。块大小的选择需要大量测量数据分析；
 - 数据访问和内存带宽：OpenBLAS 能够更有效地管理内存访问模式。这一点我的知识储备有限，无法做到更多优化，我认为这也是导致效率差异的重要原因；
 - 动态调度：OpenBLAS 可能使用更加复杂的动态任务调度算法来保证多线程工作的负载均衡。实际情况往往复杂很多，简单地利用 OpenMP 还无法保证每个线程工作量均衡，
 - 线程开销和同步：OpenMP 并行块的效率还取决于线程的创建、销毁和同步的开销。如果这些开销较大，可能会抵消并行带来的收益。

6 结语

6.1 其他方法尝试

OpenBLAS 的速度实在太快，我的任何其他手写的方法都相形见绌。但在自己一步步不断优化矩阵乘法之前，我确实没有读过任何涉及 OpenBLAS 原理层面的信息，最后发现思路其实相当接近。

在过程中，我还尝试了其他的优化策略：(但效果甚微，于是没有在上文详细列出)

- Morton Order: 使用 Z-order curve(也称为 Morton order) 优化矩阵存储，他在预处理矩阵信息的时候往往需要更高的时间开销，而且效率提高的效果也不够显著。除非我们需要对数量很多的矩阵做多次乘法操作，并且预处理的时间可以被忽略，否则我认为这样的方法在目前的情形中是舍本逐末的；
- Strassen 算法: 该算法在上一个 Project 中也有所提及，只是并未实现。它使用分治的思想处理原始大矩阵，分治之后递归地处理子矩阵，试图以此方法把时间复杂度降低至 $O(n^{\log_2 7})$ 。我认为导致该算法效率低下的原因有很多：
 - 每次递归开始之前需要为子矩阵分配内存；
 - 分治之后的合并操作需要大量时间开销；

-
- 递归结束之后需要手动释放子矩阵的内存;
 - 我们前文中提高效率的途径 SIMD、OpenMD 很难被应用于最关键乘法的部分, 只能适用于简单的合并操作。而且经实验发现, 即使是合并操作, 在矩阵尺寸仅为 2k 时就会发生分段错误 Segmentation fault (core dumped)。在尺寸更大时, 舍去这两个途径之后对于运行效率来说便是雪上加霜。

除了考虑时间以外, 分治思想往往还具有很高的空间复杂度, 在矩阵尺寸达到 64k 量级时, 内存开销也必然会成为问题。经过我的粗略计算, 在递归树递归至最深时, 即使每次递归结束之后都进行内存释放, 对于处理 64k 量级的矩阵仍是不够的。

这些尝试可以在源代码中找到。

6.2 心得