

# lab6 & lab9

- 实验环境
  - 系统版本
  - 编译器版本
  - CPU 物理核数及频率
- 四种矩阵乘实现
  - Naive
  - Openblas
  - Pthread
  - Openmp
- Gflops
- 截图
  - Naive
  - openblas
  - Pthread
  - Openmp
- Lab 3 & Lab 5
  - Lab 3 - optimize-gemm
    - 问题
  - Lab 5 - thread
    - 截图
- lab 9
  - 单核优化方法概述
    - 循环顺序优化
    - 向量化 SIMD
      - SIMD 是什么
      - 使用 SIMD 优化 gemm
    - 矩阵分块
    - 矩阵分块后数据重排
  - 实验结果

## 实验环境

### 系统版本

```
Linux LAPTOP-BGRVTJ4L 5.15.153.1-microsoft-standard-WSL2 #1 SMP Fri Mar 29 23:14:13 UTC
2024 x86_64 x86_64 x86_64 GNU/Linux
```

```
Ubuntu 22.04.4 LTS
```

### 编译器版本

```
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
```

## CPU 物理核数及频率

```
CPU(s):           16
On-line CPU(s) list: 0-15
Thread(s) per core: 2
Core(s) per socket: 8
```

```
CPU: 3792.655MHz
```

## 四种矩阵乘实现

### Naive

Naive gemm 是**最简单**的矩阵乘实现，**并未进行包括分块和多线程在内的优化**，因此也是四种实现中效率最低的实现。

Naive gemm 的 C 矩阵每个元素的计算公式如下：

$$C[i, j] = C[i, j] + A[i, p] \times B[p, j]$$

Naive gemm 通常使用最简单的三重循环实现，下面是其核心代码：

```
/* Macros for row-major order */
#define A(i, j) a[(i) * lda + (j)]
#define B(i, j) b[(i) * ldb + (j)]
#define C(i, j) c[(i) * ldc + (j)]

/* Routine for computing C = A * B + C */
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    int i, j, p;

    for (i = 0; i < m; i++) /* Loop over the rows of C */
    {
        for (j = 0; j < n; j++) /* Loop over the columns of C */
        {
            for (p = 0; p < k; p++)
            { /* Update C( i,j ) with the inner product of the ith row of A and the jth
              column of B */
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
            }
        }
    }
}
```

## Openblas

Openblas 是 **BLAS（基础线性代数程序集）** 的一种开源实现。

使用 openblas 中的 `cblas_dgemm` 函数实现 gemm，核心代码如下：

```
void MY_MMult(int m, int n, int k, double *a, int lda,
             double *b, int ldb,
             double *c, int ldc)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                1.0, a, lda, b, ldb, 1.0, c, ldc);
}
```

## Pthread

**POSIX线程**（英语：POSIX Threads，常被缩写为pthreads）是POSIX的线程标准，定义了创建和操纵线程的一套API。实现POSIX线程标准的库常被称作**pthreads**。

使用 pthread 库进行多线程计算，实现 gemm，核心代码如下：

```
#include "defs.h"
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
#include <math.h>

struct MatrixThreadArgs {
    int m;
    int n;
    int k;
    double *a;
    int lda;
    double *b;
    int ldb;
    double *c;
    int ldc;
    int section_x_begin;
    int section_x_end;
    int section_y_begin;
    int section_y_end;
};

void *MatrixThreadCalculate(void *arg) {
    struct MatrixThreadArgs matrixThreadArgs = *((struct MatrixThreadArgs *) arg);
    int k = matrixThreadArgs.k;
    double *a = matrixThreadArgs.a;
    int lda = matrixThreadArgs.lda;
    double *b = matrixThreadArgs.b;
    int ldb = matrixThreadArgs.ldb;
    double *c = matrixThreadArgs.c;
    int ldc = matrixThreadArgs.ldc;
    int section_x_begin = matrixThreadArgs.section_x_begin;
    int section_x_end = matrixThreadArgs.section_x_end;
    int section_y_begin = matrixThreadArgs.section_y_begin;
    int section_y_end = matrixThreadArgs.section_y_end;
```

```

    const int block_size = min(64, (max(section_x_end - section_x_begin, section_y_end
- section_y_begin)));
    int block_column_num = ceil((section_x_end - section_x_begin) / block_size);
    int block_row_num = ceil((section_y_end - section_y_begin) / block_size);

    for (int block_x = 0; block_x < block_column_num; block_x++) {
        for (int block_y = 0; block_y < block_row_num; block_y++) {
            int block_base_x = section_x_begin + block_x * block_size;
            int block_base_y = section_y_begin + block_y * block_size;
            int block_end_x = min(section_x_end, block_base_x + block_size);
            int block_end_y = min(section_y_end, block_base_y + block_size);

            for (int i = block_base_x; i < block_end_x; i++) {
                for (int j = block_base_y; j < block_end_y; j++) {
                    for (int p = 0; p < k; p++) {
                        C(i, j) = C(i, j) + A(i, p) * B(p, j);
                    }
                }
            }
        }
    }
    return NULL;
}

```

```

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc) {
    const int x_seperate = 4, y_seperate = 4, thread_num = x_seperate * y_seperate;

    pthread_t threads[thread_num];
    struct MatrixThreadArgs sectionMatrixThreadArgs[thread_num];
    int rc;

    int section_weight = ceil(n / x_seperate);
    int section_height = ceil(m / y_seperate);

    int thread_index = 0;
    for (int section_x = 0; section_x < x_seperate; section_x++) {
        for (int section_y = 0; section_y < y_seperate; section_y++) {
            int section_x_begin = section_x * section_weight;
            int section_x_end = min(n, section_x_begin + section_weight);
            int section_y_begin = section_y * section_height;
            int section_y_end = min(m, section_y_begin + section_height);

            sectionMatrixThreadArgs[thread_index] = (struct MatrixThreadArgs){
                .m = m,
                .n = n,
                .k = k,
                .a = a,
                .lda = lda,
                .b = b,
                .ldb = ldb,
                .c = c,
                .ldc = ldc,
                .section_x_begin = section_x_begin,
                .section_x_end = section_x_end,
                .section_y_begin = section_y_begin,

```

```

        .section_y_end = section_y_end
    };
    rc = pthread_create(&threads[thread_index], NULL, MatrixThreadCalculate,
&sectionMatrixThreadArgs[thread_index]);
    assert(rc == 0);
    thread_index++;
}
}
for (int i = 0; i < thread_num; i++) {
    rc = pthread_join(threads[i], NULL);
    assert(rc == 0);
}
}

```

## Openmp

**OpenMP** (Open Multi-Processing) 是一套支持跨平台共享内存方式的多线程并发的编程 API。

核心代码如下：

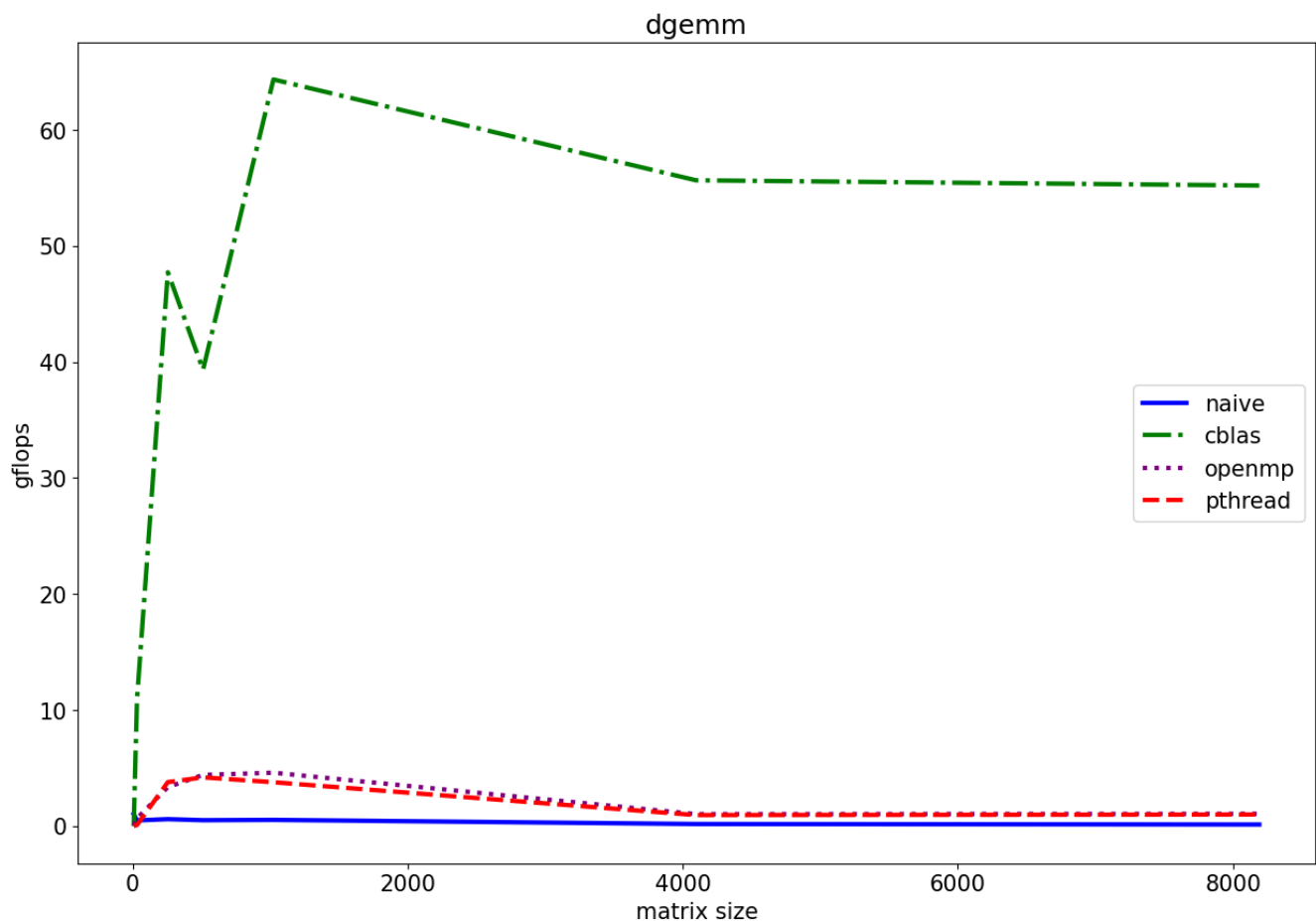
```

void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    int i, j, p;
    #pragma omp parallel for private(j, p)
    for (i = 0; i < m; i++) /* Loop over the rows of C */
    {
        for (j = 0; j < n; j++) /* Loop over the columns of C */
        {
            for (p = 0; p < k; p++)
            { /* Update C( i,j ) with the inner product of the ith row of A and the jth
column of B */
                C(i, j) = C(i, j) + A(i, p) * B(p, j);
                // printf("Thread %d: i=%d, j=%d, p=%d\n", omp_get_thread_num(), i, j, p);
            }
        }
        // printf("Thread %d: i=%d\n", omp_get_thread_num(), i);
    }
}

```

## Gflops

下图为四种 gemm 实现在不同矩阵规模下的 gflops 曲线图：



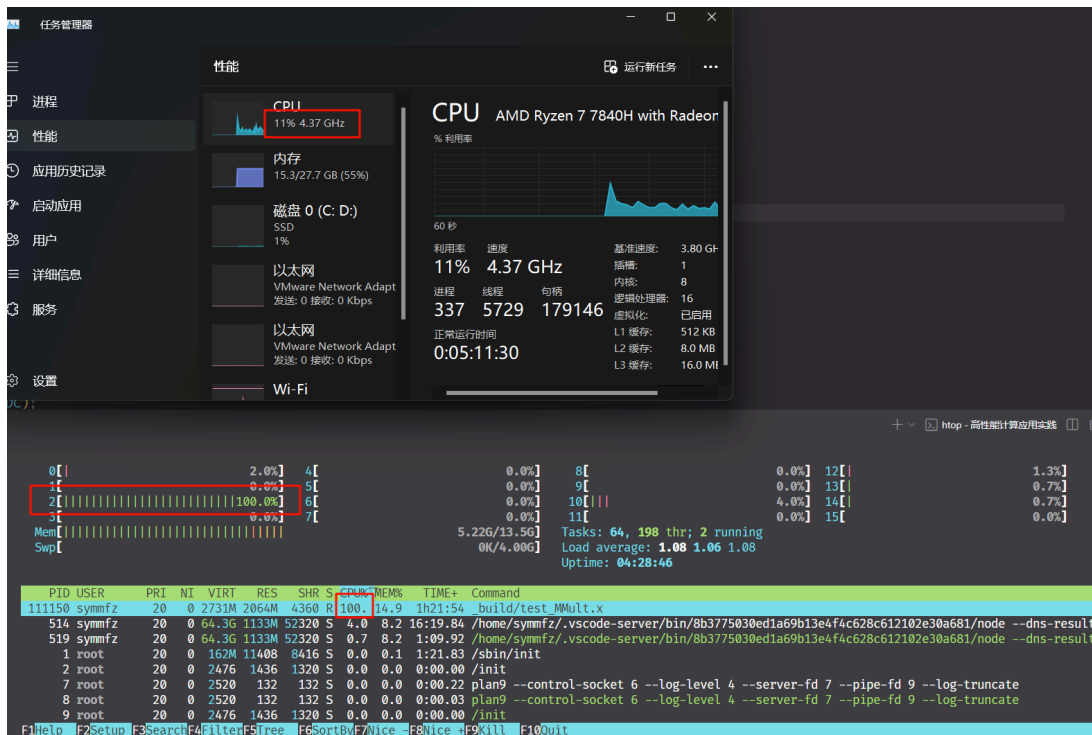
根据上图，不难发现以下结论：

- Cblas 实现的 gemm 在绝大多数矩阵规模下，gflops 都显著高于其他四种实现，最高可达 naive 实现的 500 倍以上
- Cblas 实现在较低矩阵规模时 gflops 较低，随着矩阵规模的上涨 gflops 值先快速上涨然后稳定。Gflops 峰值出现在  $1024 \times 1024$  的矩阵规模附近，峰值大小约为 64.35
- Openmp 和 pthread 实现的 gflops 曲面相似，从 gflops 的大小上看，大于 naive 实现并显著小于 cblas 实现；从曲线的变化上看，随着矩阵规模的增大，gflops 先增后减，峰值出现在  $512 \times 512$  或  $1024 \times 1024$  附近，峰值 gflops 约为 4.4
- Naive 实现的 gflops 值在矩阵规模大于  $32 \times 32$  时最低。从趋势上看，gflops 值大致随着矩阵规模的增大而减小

## 截图

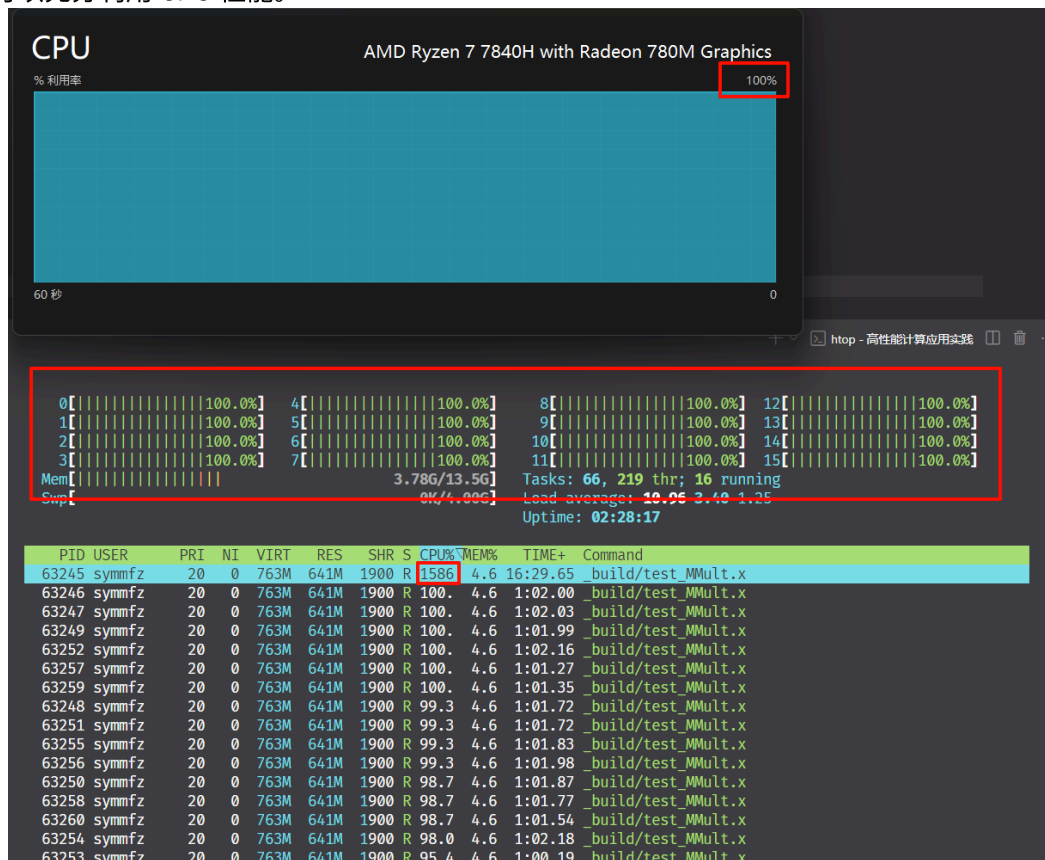
### Naive

Naive 为单线程运行，无法充分利用 CPU 的性能。



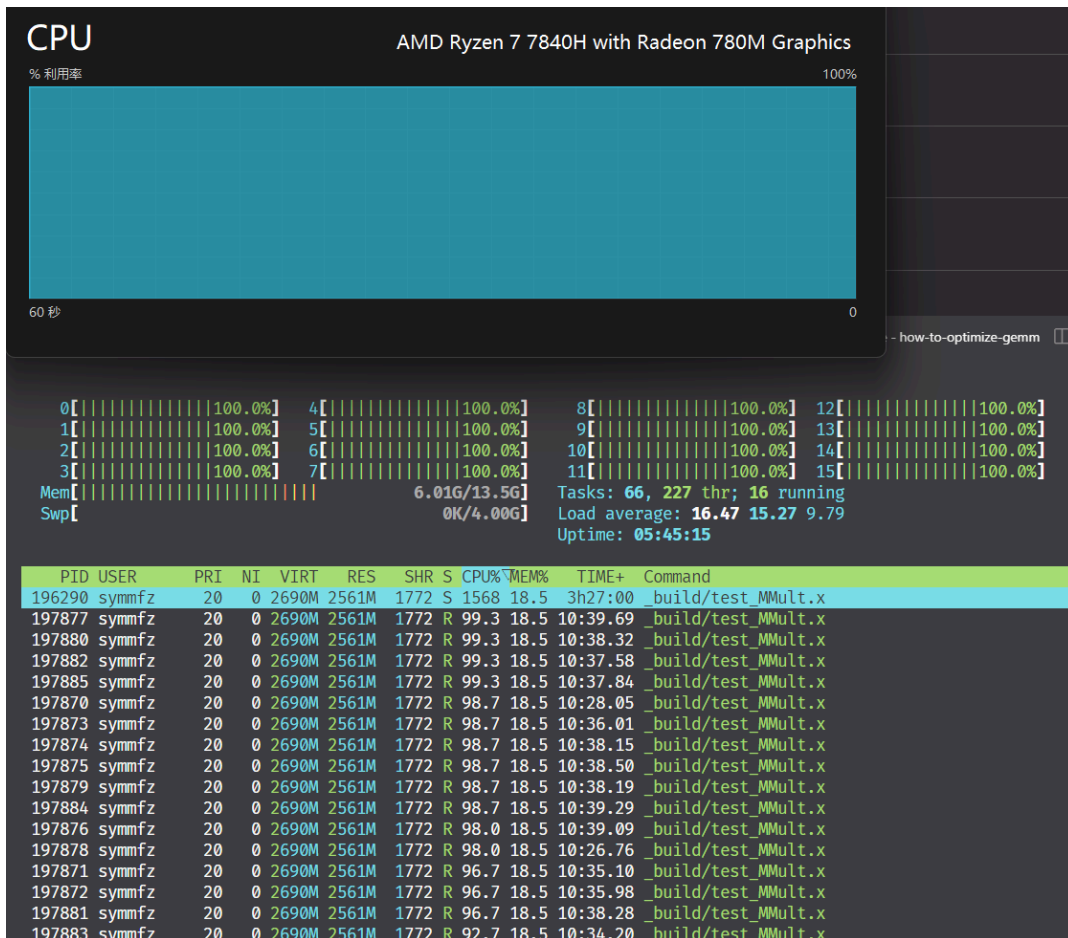
## openblas

Openblas 可以利用 CPU 性能。



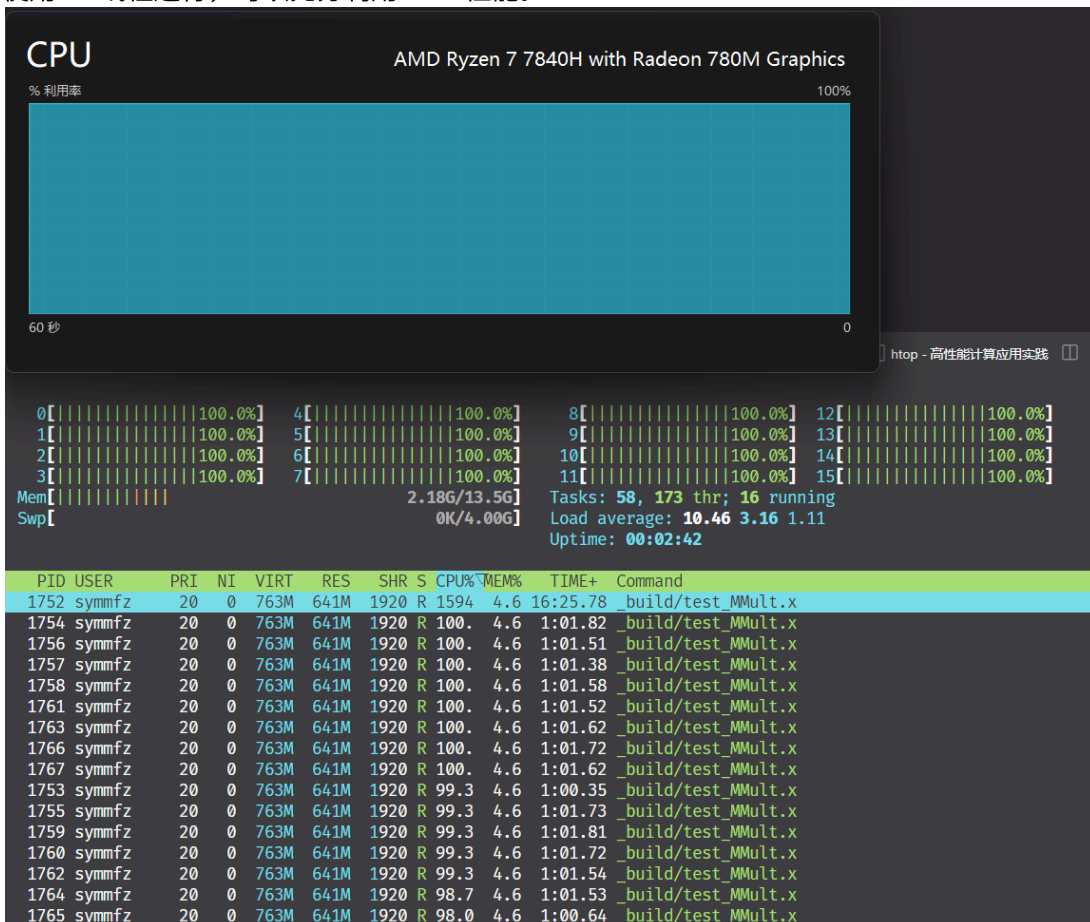
## Pthread

Pthread 16 线程运行



## Openmp

Openmp 使用 16 线程运行，可以充分利用 CPU 性能。





```
top - 23:24:49 up 2:31, 1 user, load average: 2.12, 0.79, 0.42
Tasks: 61 total, 3 running, 58 sleeping, 0 stopped, 0 zombie
%Cpu(s): 98.4 us, 1.1 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.4 si, 0.0 st
MiB Mem : 13824.3 total, 11115.6 free, 2294.9 used, 413.9 buff/cache
MiB Swap: 4096.0 total, 4096.0 free, 0.0 used. 11256.0 avail Mem

  PID USER      PR  NI   VIRT   RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 88539 symmfz    20   0 289908 166092 1920 R   1582   1.2   3:30.61 test_MMult.x

fish—sh—sh—sh—node—node—fish—test_MMult.x—15*[{test_MMult.x}]
                             |
                             |—fish—pstree
                             |   |
                             |   |—2*[{fish}]
                             |   |
                             |   |—12*[{node}]
                             |   |
                             |   |—node—12*[{node}]
                             |   |
                             |   |—node—cpptools—25*[{cpptools}]
                             |   |   |
                             |   |   |—node—10*[{node}]
                             |   |   |
                             |   |   |—2*[{node—6*[{node}]]
                             |   |   |
                             |   |   |—16*[{node}]
                             |   |   |
                             |   |   |—10*[{node}]
```

## Lab 3 & Lab 5

### Lab 3 - optimize-gemm

#### 问题

##### ? Question 1

多个 c 代码中有相同的 MY\_MMult 函数，怎么判断可执行文件调用的是哪个版本的 MY\_MMult 函数？是 makefile 中的哪行代码决定的？

##### Answer

MY\_MMult 函数的版本由 makefile 文件决定，具体来说是由下面这行代码决定的：

```
NEW := openblas_MMult
```

这个决定是在以下这行代码中实现的：

```
OBJS := $(BUILD_DIR)/util.o $(BUILD_DIR)/REF_MMult.o $(BUILD_DIR)/test_MMult.o
$(BUILD_DIR)/$(NEW).o
```

改变 NEW 的值即可改变调用的 MY\_MMult，例如，上面 NEW 的值为 openblas\_MMult，表示 MY\_MMult 将调用 openblas 实现的版本

##### ? Question 2

性能数据 \_data/output\_MMult0.M 是怎么生成的？C 代码中只是将数据输出到终端并没有写入文件。

##### Answer

性能数据 \_data/output\_MMult0.M 是通过将运行 \$(BUILD\_DIR)/test\_MMult.x 的输出重定向到文件来生成的。这在 run 目标中的以下行完成：

```
$(BUILD_DIR)/test_MMult.X >> $(DATA_DIR)/output_$(NEW).M
```

上面这行代码表示将程序的输出追加到性能数据文件中。

# Lab 5 - thread

## 截图

```
top - 00:44:48 up 1:09, 1 user, load average: 2.24, 1.59, 1.11
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 52.3 us, 0.6 sy, 0.0 ni, 45.7 id, 0.0 wa, 0.0 hi, 1.4 si, 0.0 st
MiB Mem : 13824.3 total, 11132.9 free, 2135.1 used, 556.3 buff/cache
MiB Swap: 4096.0 total, 4096.0 free, 0.0 used. 11426.7 avail Mem
```


PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
37461	symmfz	20	0	232364	165904	1776	S	800.0	1.2	3:11.61	test_MMult.x

```
top - 00:40:05 up 1:04, 1 user, load average: 2.72, 1.56, 0.97
Threads: 9 total, 8 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 49.6 us, 0.3 sy, 0.0 ni, 48.8 id, 0.0 wa, 0.0 hi, 1.2 si, 0.0 st
MiB Mem : 13824.3 total, 11130.1 free, 2138.7 used, 555.5 buff/cache
MiB Swap: 4096.0 total, 4096.0 free, 0.0 used. 11423.2 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
35215	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35216	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35217	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35218	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35219	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35220	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35221	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
35222	symmfz	20	0	232364	165884	1760	R	99.9	1.2	0:13.17	test_MMult.x
34099	symmfz	20	0	232364	165884	1760	S	0.0	1.2	1:39.55	test_MMult.x

## lab 9

实验九选择任务 1，具体实验方向如下所示：

 任务 1

在 how-to-optimize-gemm 框架进一步探索单核的优化方法，包括调整 ijk 顺序、向量化 SIMD、矩阵分块、分块后数据重排，ijk 顺序组合有：ijk, ikj, jik, jki, kij, kji。

## 单核优化方法概述

### 循环顺序优化

在矩阵乘法的实现中，循环的顺序对内存访问模式有着显著影响。gemm 中的循环顺序会影响计算的缓存命中率，从而影响矩阵计算的速度和性能。

以循环循序为 ijk 的 naive gemm 为例：

```
for (int i=0; i<m; i++) {
    for (int j=0; j<n; j++) {
        for (int k=0; k<p; k++) {
            C[i * n + j] += A[i * n + p] * B[p * n + j]
        }
    }
}
```

在最内层的循环中， $k$  值不断变化，意味着矩阵  $B$  是逐列读取的。由于矩阵数据是按照行优先顺序存储的，因此按列访问需要频繁跨越整行数据。如果矩阵  $B$  的尺寸较大，频繁的列访问可能会导致较高的缓存未命中率。

简而言之，对于行主序储存的矩阵，如果以  $ijk$  顺序循环，会导致在读取矩阵  $B$  元素时内存访问地址跳跃较大，容易导致缓存缺失，缓存命中率较低。

由于 CPU 缓存的速度快于内存，优化矩阵乘法的性能的一个基本方法就是提高缓存命中率。于是可以通过改变循环顺序提高计算过程的缓存命中率，从而提高计算性能。

```
// ikj 顺序实验结果，其中三列数字分别为 矩阵大小 gflops 计算误差
date = 'Sat Oct 5 21:19:27 CST 2024';
version = 't1_ikj';
MY_MMult = [
16 8.192000e-01 0.000000e+00
32 8.192000e-01 0.000000e+00
64 7.872192e-01 0.000000e+00
128 8.267897e-01 0.000000e+00
256 8.171452e-01 0.000000e+00
512 8.387350e-01 0.000000e+00
768 8.633438e-01 0.000000e+00
1024 8.539657e-01 0.000000e+00
2048 8.573356e-01 0.000000e+00
];
```

综合来看，在所有的循环顺序组合中， $ikj$  循环顺序的  $gflops$  在不同矩阵规模中均能达到较高水平。在实验中，其他的循环顺序或者在矩阵规模较大时  $gflops$  明显下降，或者在矩阵规模较小时  $gflops$  较低。相对的， $ijk$  的计算顺序总能保持在较高水平。

这里简单以  $ikj$  循环顺序为例分析循环顺序如何影响缓存命中率，并且解释为何某些循环顺序下（如  $ijk$ ）随着矩阵规模的增大， $gflops$  值下降明显。

```
// ikj 循环顺序
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc) {
    int i, j, p;
    for (i = 0; i < m; i++) {
        for (p = 0; p < k; p++) {
            for (j = 0; j < n; j++) {
                C(i, j) += A(i, p) * B(p, j);
            }
        }
    }
}
```

此时，最内层循环中的  $j$  值发生变化，这意味着矩阵  $B$  将按行读取，而非  $ijk$  循序的按列读取，这极大减少了缓存未命中的概率。同时矩阵  $A$  和矩阵  $C$  的内存读取循序虽然发生了变化，但对其性能影响并不大，综合来说  $ikj$  的计算顺序提高了算法的综合缓存命中率，提高了计算的性能。

前文提到，在某些循环顺序下，计算的  $gflops$  值会随着矩阵规模的增加有明显下降。这是因为矩阵规模的增加使缓存缺失明显增加，整体的缓存命中率下降。例如，在  $ijk$  循环顺序时，矩阵  $B$  的元素读取方式是按列读取，而储存是按行储存（假设为行主序）。当矩阵规模增大时，矩阵  $B$  每一行元素的个数增多，这意味着按列读

取每个元素的过程中内存地址的跳跃性增大，因此更容易出现缓存缺失的情况。于是，随着矩阵规模的增大，gflops 值逐渐下降。

## 向量化 SIMD

### SIMD 是什么

#### SIMD

**单指令流多数据流**（英语：**Single Instruction Multiple Data**，缩写：**SIMD**）是一种采用一个**控制器**来控制多个**处理器**，同时对一组数据（又称“**数据向量**”）中的每一个分别执行**相同**的操作从而实现空间上的**并行**性的技术。

SIMD 的作用如其名称所示，通过单条指令同时处理多个数据元素。我们可以用 SIMD 提高 gemm 的性能。

gemm 的朴素实现逐个处理矩阵元素，换言之，在最内层循环只做两个两个浮点数的乘法，效率较低。*SIMD 优化通过单条指令同时处理多个数据元素，加速运算。*相比之下，naive GEMM 在每次计算时需要频繁读取和处理单个元素，而 SIMD 可以并行计算多组数据，减少循环次数和内存访问开销。这种优化方式充分利用了现代处理器的并行计算能力，大幅提高了矩阵乘法的效率，特别是在大规模计算中表现显著。

### 使用 SIMD 优化 gemm

SIMD 256 是指 CPU 同时对 256 bit 的数据进行读写或者运算，使用 SIMD 需要 CPU 支持 SIMD 的指令集。

对于 intel x86 架构的 CPU 来说，这个指令集通常是 AVX2.0，它支持 SIMD 256，可以同时处理 4 个 double 类型的浮点数进行读写或者运算。

在 C 语言中使用 AVX2.0 进行矩阵计算需要导入 immintrin.h 库，然后才能够调用 AVX2.0 的 api 和数据类型，具体的代码如下所示。

```
#include <immintrin.h>

/* SIMD 256 with unroll 4 */
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    int i, j, p, t;
    __m256d cm[4], a0, b0;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j += 4 * 4) {
            for (t = 0; t < 4; t++) {
                cm[t] = _mm256_load_pd(c + i * n + j + t * 4);
            }
            for (p = 0; p < k; p++) {
                a0 = _mm256_broadcast_sd(a + i * k + p);
                cm[0] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * n + j));
                cm[1] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * n + j + 4));
                cm[2] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * n + j + 8));
                cm[3] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * n + j + 12));
            }
            for (t = 0; t < 4; t++) {
                _mm256_store_pd(c + i * n + j + t * 4, cm[t]);
            }
        }
    }
}
```

```
}  
}
```

实验得到的结果如下所示：

```
date = 'Sat Oct 5 21:36:47 CST 2024';  
version = 't1_SIMD_with_unroll4';  
MY_MMult = [  
16 2.730667e+00 0.000000e+00  
32 4.681143e+00 0.000000e+00  
64 4.332959e+00 0.000000e+00  
128 4.359983e+00 0.000000e+00  
256 4.464400e+00 0.000000e+00  
512 4.466480e+00 0.000000e+00  
768 4.499231e+00 0.000000e+00  
1024 4.500836e+00 0.000000e+00  
2048 2.623453e+00 0.000000e+00  
];
```

可以发现，经过 SIMD 优化，gflops 明显提高。

## 矩阵分块

矩阵分块可以提高矩阵乘法中的缓存命中率，其思想是使经常访问的数据更加集中，分析方法与循环顺序优化部分类似，故这里不做赘述。

矩阵分块的方法非常直观，即将矩阵分成数个小矩阵，然后依次计算每个小矩阵的每个元素，这样集中计算小矩阵有利于提到计算的缓存命中率。

代码如下：

```
/* Macros for row-major order */  
#define A(i, j) a[(i) * lda + (j)]  
#define B(i, j) b[(i) * ldb + (j)]  
#define C(i, j) c[(i) * ldc + (j)]  
const int BLOCK_SIZE = 64;  
  
void MY_MMult(int m, int n, int k, double *a, int lda,  
              double *b, int ldb,  
              double *c, int ldc) {  
    int i, j, p;  
    for (i = 0; i < m; i += BLOCK_SIZE) {  
        for (j = 0; j < n; j += BLOCK_SIZE) {  
            for (p = 0; p < k; p += BLOCK_SIZE) {  
                int i_block = i + BLOCK_SIZE > m ? m : i + BLOCK_SIZE;  
                int j_block = j + BLOCK_SIZE > n ? n : j + BLOCK_SIZE;  
                int p_block = p + BLOCK_SIZE > k ? k : p + BLOCK_SIZE;  
  
                for (int i1 = i; i1 < i_block; i1++) {  
                    for (int j1 = j; j1 < j_block; j1++) {  
                        for (int p1 = p; p1 < p_block; p1++) {  
                            C(i1, j1) += A(i1, p1) * B(p1, j1);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    }
}

}
}
}

```

这里将矩阵分块和 `ijk` 顺序 gemm 的实验数据进行对比。

下面为 `ijk` 顺序的 naive gemm：

```

date = 'Sat Oct 5 21:16:17 CST 2024';
version = 't1_ijk';
MY_MMult = [
16 7.447273e-01 0.000000e+00
32 7.992195e-01 0.000000e+00
64 6.472691e-01 0.000000e+00
128 7.105377e-01 0.000000e+00
256 5.454054e-01 0.000000e+00
512 5.092647e-01 0.000000e+00
768 6.397476e-01 0.000000e+00
1024 3.978009e-01 0.000000e+00
2048 1.989489e-01 0.000000e+00
];

```

矩阵分块计算：

```

date = 'Sat Oct 5 21:38:28 CST 2024';
version = 't1_MultiBlocks';
MY_MMult = [
16 8.192000e-01 0.000000e+00
32 8.295696e-01 0.000000e+00
64 8.282591e-01 0.000000e+00
128 5.694141e-01 0.000000e+00
256 6.478690e-01 0.000000e+00
512 5.888499e-01 0.000000e+00
768 6.626533e-01 0.000000e+00
1024 6.165061e-01 0.000000e+00
2048 4.704800e-01 0.000000e+00
];

```

观察实验数据可以发现，*矩阵分块后 gflops 值有所增大，并且矩阵规模越大，矩阵分块优化对 gflops 的提升越大*。这是因为随着矩阵规模增大，缓存缺失的问题将会越来越明显，因此更能体现矩阵分块提高缓存利用率的功能。

## 矩阵分块后数据重排

矩阵分块可以提高缓存命中率，数据重排后进行 SIMD 计算可以减少运算指令数，综合使用可以进一步提高矩阵计算的性能。

代码如下：

```

#include <stdio.h>
#include <immintrin.h>
#include <math.h>
#define A(i, j) a[(i) * lda + (j)]
#define B(i, j) b[(i) * ldb + (j)]
#define C(i, j) c[(i) * ldc + (j)]
const int BLOCK_SIZE = 64;

void calculateBlock(int m, int n, int k, double *a, int lda,
                   double *b, int ldb,
                   double *c, int ldc)
{
    int i, j, p, t;
    __m256d cm[4], a0, b0;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j+=4 * 4) {
            for (t = 0; t < 4; t++) {
                cm[t] = _mm256_load_pd(c + i * ldc + j + t * 4);
            }
            for (p = 0; p < k; p++) {
                a0 = _mm256_broadcast_sd(a + i * lda + p);
                cm[0] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * ldb + j));
                cm[1] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * ldb + j + 4));
                cm[2] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * ldb + j + 8));
                cm[3] += _mm256_mul_pd(a0, _mm256_load_pd(b + p * ldb + j + 12));
            }
            for (t = 0; t < 4; t++) {
                _mm256_store_pd(c + i * ldc + j + t * 4, cm[t]);
            }
        }
    }
}

/* Routine for computing C = A * B + C */
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb,
              double *c, int ldc)
{
    int i, j, p;

    for (i = 0; i < m; i += BLOCK_SIZE)
    {
        for (j = 0; j < n; j += BLOCK_SIZE)
        {
            for (p = 0; p < k; p += BLOCK_SIZE)
            {
                int i_block = i + BLOCK_SIZE > m ? m : i + BLOCK_SIZE;
                int j_block = j + BLOCK_SIZE > n ? n : j + BLOCK_SIZE;
                int p_block = p + BLOCK_SIZE > k ? k : p + BLOCK_SIZE;

                calculateBlock(i_block - i, j_block - j, p_block - p, a + i * lda + p, lda,
                              b + p * ldb + j, ldb,
                              c + i * ldc + j, ldc);
            }
        }
    }
}

```

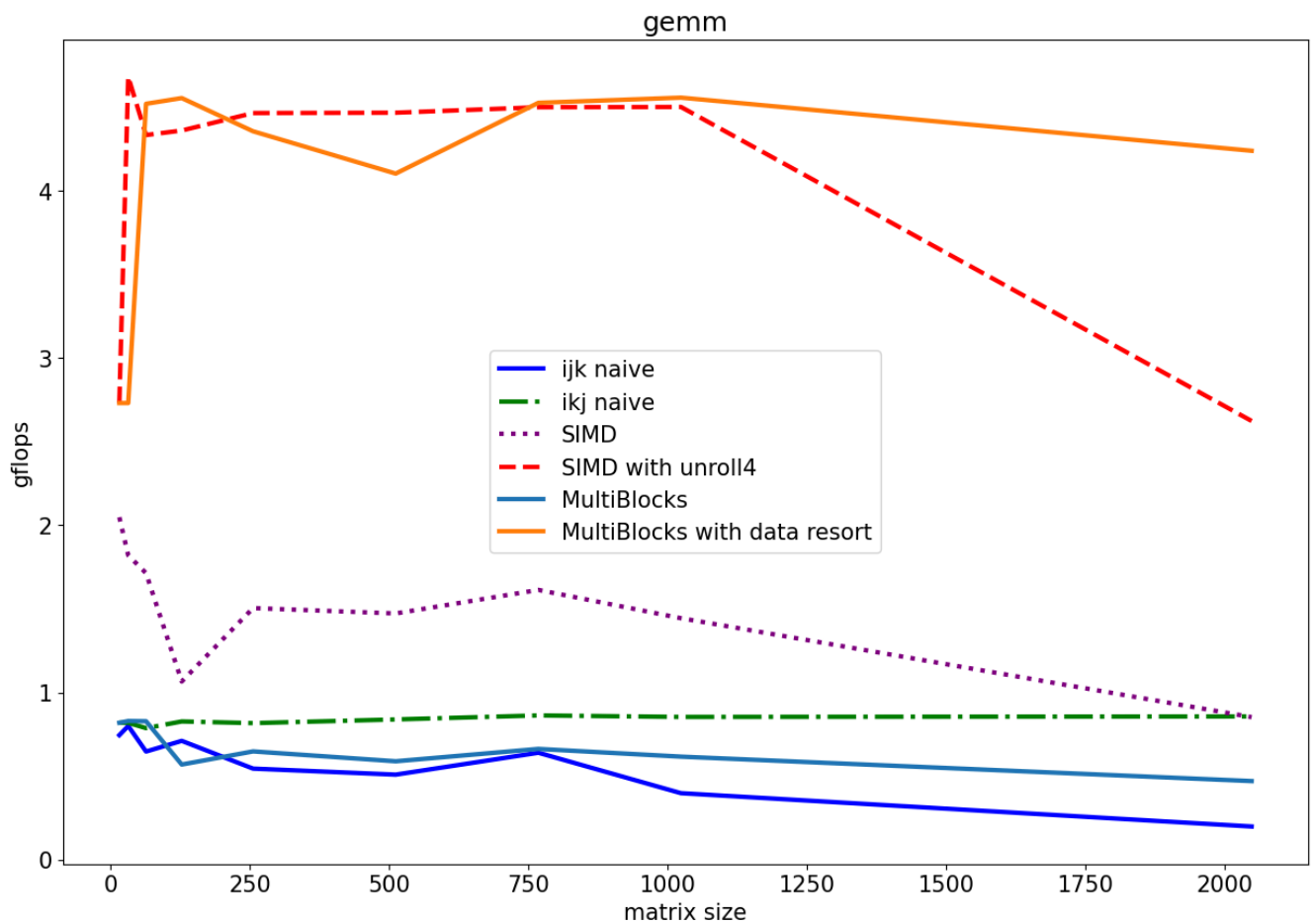
```
}
```

实验结果如下所示：

```
date = 'Sat Oct 5 21:40:44 CST 2024';  
version = 't1_MultiBlocks_data_resort';  
MY_MMult = [  
16 2.730667e+00 0.000000e+00  
32 2.730667e+00 0.000000e+00  
64 4.519724e+00 0.000000e+00  
128 4.554076e+00 0.000000e+00  
256 4.355456e+00 0.000000e+00  
512 4.102573e+00 0.000000e+00  
768 4.525526e+00 0.000000e+00  
1024 4.556666e+00 0.000000e+00  
2048 4.238878e+00 0.000000e+00  
];
```

可以发现，分块后数据重排后 gflops 明显增大。

## 实验结果





可以发现分块后数据重排的 gflops 值最高，SIMD with unroll 4 其次。其余的 4 种优化 gflops 相对价低，从大到小分别为 SIMD、ikj naive、MultiBlocks、ijk naive。