Improved Matrix Multiplication

项目编号：project3

学　　号：	**12210158**
姓　　名：	吴同馨
指导老师：	于仕琪
完成时间：	2024-04-27

目 录

第1章 项目要求分析

1.1 概述

根据项目作业要求“Project3.pdf”文件，本项目的要求是用 C 编写程序，提升矩阵乘法的速度。

1.2 项目要求依据

Project3.pdf 文档。

1.3 开发语言要求

C 语言。

1.4 实现功能要求

1. 以几个循环为基准，用简单的方法创建一个函数 matmul plain ( )；
2. 使用 SIMD、OpenMP 和其他技术实现函数 matmul improved( ) 以提高速度；
3. matmul improved( ) 与 matmul_plain( ) 进行比较；
4. 使用 16x16、128x128、1Kx1K、8Kx8K 和 64Kx64K 矩阵测试性能；
5. 实现 OpenBLAS 中的矩阵乘法并进行比较；
6. 在 X86 和 ARM 平台上测试程序；
7. 使用前要检查参数；
8. 检查内存泄漏；
9. 使用-O3 编译。

1.5 项目提交内容

1. 源代码: .c, .h 文件格式
2. report.pdf 文件格式

1.6 项目提交时间

截止时间：2024 年 4 月 28 日 23:59

第2章 项目要点分析

2.1 概述

本项目的目标是优化矩阵乘法计算，因此优化方向是首要问题。

一、优化首先设定优化标准。在矩阵乘法计算中，可以想见最慢的应该是朴素算法，即暴力计算；而 OpenBLAS 作为成熟的产品，计算一定是最快的。这样，就把矩阵乘法计算速度快慢的的上下限区间定下来了。

二、计算优化方向之一应该是从软件算法上进行。以暴力算法为起点，大致有分块算法、Strassen 算法、Coppersmith-Winograd 算法。Coppersmith-Winograd 算法实际上在现实情形下很少被用到，而且编程难度高，很难在有限时间内完成，因此在算法上的优化决定采用分块和 Goto 方式。

三、计算优化方向之二应该是实现方式的优化，其实项目要求中所记述的采用 SIMD 和 OpenMP 等方式就是从实现方式上面进行优化。

四、计算优化方向之三是从硬件方面入手进行优化。目前 AI 方面最火的硬件是 NVIDIA GPU，很多资料显示比特币挖矿和 AI 计算大多利用 GPU 上完成。通过查阅资料，发现 OpenCL 可以驱动 GPU 计算，因此可以尝试一下 GPU 的矩阵乘法计算。

综上所述并结合作业要求中提到的优化方法，优化矩阵乘法计算的基本思路如下：

- OpenBLAS 的使用（是优化的标准和方向）；
- 朴素算法；（起点）
- 调整循环次序；
- 分块算法；
- SIMD 的使用；
- OpenMP 的使用；
- ARM 的使用（考虑 ARM 架构下的速度）

另外，考虑到并行计算和 GPU 的优势，尝试了如下的方法：

- 自定义多线程进行并行计算；

- OpenCL 的使用；

2.2 OpenBlas 优化方法的学习

由于 OpenBlas 具有公认的矩阵乘法高速运算的能力，对我们而言是一个标准，因此在本项目中，重点准备采用 OpenBlas 方向性的优化手段，这样能避免走弯路。

根据查找的有关 OpenBlas 的资料，对 OpenBlas 速度快的原因总结如下：

- 高度优化的代码：OpenBLAS 的代码经过精心设计和优化，以充分利用现代处理器的特性，如 SIMD（单指令多数据）指令、多线程和缓存机制。这使得它在进行矩阵乘法等操作时能够更高效地利用计算资源。
- 多线程支持：OpenBLAS 支持多线程运算，这意味着它可以同时利用多个处理器核心进行矩阵乘法计算。这显著提高了计算速度，尤其对于大型矩阵乘法运算。
- 自动向量化：OpenBLAS 通过自动向量化技术，将多个数据元素组合成向量，并使用 SIMD 指令进行并行处理。这提高了数据吞吐量，减少了计算延迟。
- 内存访问优化：矩阵乘法运算涉及大量的内存访问。OpenBLAS 通过优化内存布局和访问模式，减少了内存延迟，提高了缓存命中率，从而提高了运算速度。
- 算法优化：OpenBLAS 还采用了多种算法优化技术，如分块乘法、循环展开等，以进一步提高矩阵乘法的性能。

  以上的方法也是我们对矩阵乘法优化的方向，后面的程序的改进基本依据这些思路进行。

注：以上内容重点参考以下文章的方法：

① 矩阵乘法 - Algorithmica

② 经典论文精读——《Anatomy of High-Performance matrix multiplication》-CSDN 博客

第3章 性能测试约定

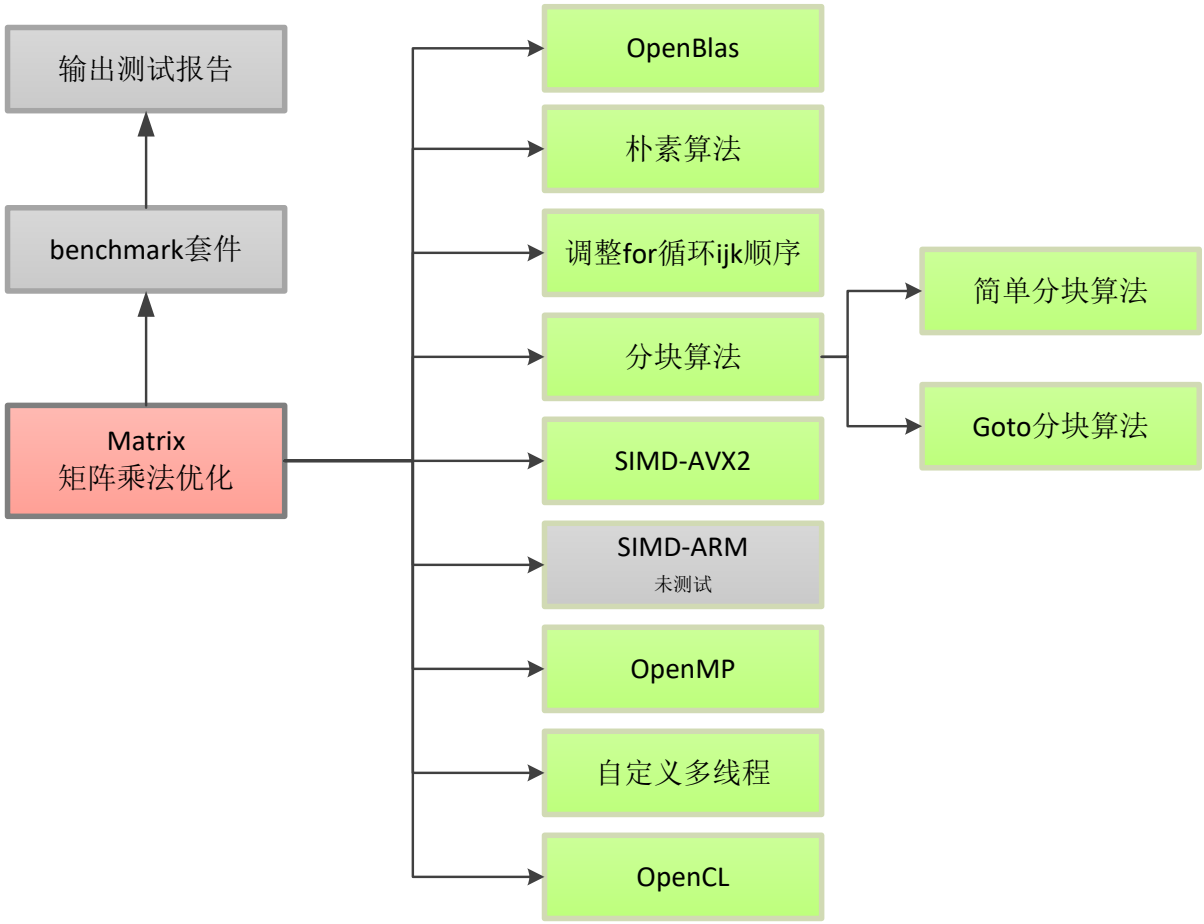
No	约定	内容
1	矩阵类型	正方形
2	矩阵大小	16、128、1024、8192、65536
3	矩阵值	0.0-0.1
4	测试次数	Googlebenchmark 自动设定
5	单次测试时间单位	微秒 μ s
6	测时起点	矩阵乘法前
7	测时终点	矩阵乘法后
8	机器	学校远端服务器以及个人电脑
9	单次测试记录	矩阵大小、用时
10	算法速度比较标准	均指 1024*1024 矩阵大小
11	矩阵算法 for 循环 ijk 的 index 含义	i - A 矩阵　j - B 矩阵　k - C 矩阵
12	矩阵乘法中的矩阵	C 矩阵 = A 矩阵 * B 矩阵

第4章 开发工具

项目	内容	参照
操作系统	ubuntu	-
开发语言	C/C++	-
集成开发工具	vi + gcc + gdb	-
项目构建工具	cmake	-
内存泄漏测试工具	valgrind	-
OpenBlas	OpenBlas 库	-
OpenCL	GPU 计算	参考附录

第5章 程序结构与项目构建

5.1 程序逻辑结构



程序结构图

程序结构分为两部分：各优化矩阵乘法和 benchmark。优化乘法被编译成静态库供 benchmark 调用。

5.2 程序目录结构

以下是本项目的目录结构，采用 cmake 进行构建管理。其中 src 目录下的程序将被编译成静态库 libmm.a，test 目录下 benchtest.cpp 调用 libmm.a 库中各个计算函数。

```
project03
    |---01c                    <!-- C/C++程序 -->
        |--- CMakeLists.txt
        |---bin                <!-- 执行程序被 cmake 拷贝至本目录 -->
        |---build              <!-- 编译结果 -->
        |---src
            |--- CMakeLists.txt
            |--- matrixMul_openBlas.c    <!-- Openblas 计算 -->
            |--- matrixMul.c             <!-- 朴素算法集 -->
            |--- matrixMul.h             <!-- 乘法函数定义 -->
            |--- matrixMul_SIMD.c        <!-- simd 计算 -->
            |--- matrixMul_openMP.c      <!-- OpenMP 计算 -->
            |--- matrixMul_threads.c     <!-- 自定义多线程计算 -->
            |--- matrixMul_opencl.c      <!—Opencl 计算 -->
            |--- matrixMul_opencl.h      <!—Opencl 定义 -->
        |---test
            |--- CMakeLists.txt
            |--- bentest.cpp           <!-- googlebenchmark 测试程序 -->
```

5.3 CMakeFileLists.txt 文件

- 根目录 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.6)
project(mm)
# 添加编译选项 -O3
add_compile_options(-O3)
add_definitions(-DWITH_AVX2)
add_definitions(-mavx)
add_definitions(-funroll-loops)
add_definitions(-ftree-vectorize)
add_definitions(-falign-loops=512)
add_definitions(-falign-functions=256)
add_definitions(-freorder-blocks-algorithm=stc)
#add_definitions(-DWITH_NEON)

# 设置编译类型为 Debug，供 gdb 使用
#set(CMAKE_BUILD_TYPE Debug)
```

```
# 设置编译类型为 Release
set(CMAKE_BUILD_TYPE Release)
include_directories(
    ${CMAKE_CURRENT_SOURCE_DIR}
)
add_subdirectory(src)
add_subdirectory(test)
```

- src 子目录 CMakeLists.txt

```
# 查找 OpenBLAS 库和头文件
find_library(OPENBLAS_LIBRARY NAMES libopenblas.so HINTS
/usr/lib/x86_64-linux-gnu)
find_path(OPENBLAS_INCLUDE_DIR NAMES cblas.h HINTS /usr/include)
find_package(OpenMP REQUIRED)
find_package(OpenCL REQUIRED)

# 包含 OpenBLAS 的头文件
include_directories(${OPENBLAS_INCLUDE_DIR})
# 添加头文件搜索路径
include_directories(${CMAKE_CURRENT_SOURCE_DIR})
# 添加源文件列表，并编译为库
set(SRC matrix_openBlas.c
    matrixMul.c
    matrixMul_SIMD.c
    #matrixMul_ARM.c
    matrixMul_openMP.c
    matrixMul_threads.c
    matrixMul_opencl.c
    )
set(SRC_H matrixMul.h matrixMul_opencl.h)
# 设置库的名字
add_library(mm STATIC ${SRC} ${SRC_H})
# 链接 OpenBLAS 库
target_link_libraries(mm ${OPENBLAS_LIBRARY})
target_link_libraries(mm  OpenMP::OpenMP_C)
```

- test 子目录 CMakeLists

```
# 在编译时如果不指定 opencl 的版本会有 note 提示产生，加上后则消失
add_definitions(-DCL_TARGET_OPENCL_VERSION=300)
# 添加头文件搜索路径
include_directories(${CMAKE_CURRENT_SOURCE_DIR} ../src)

```

```
# 链接之前生成的库
link_directories(${CMAKE_CURRENT_BINARY_DIR}/../src)
#自定义 benchmark
set(SRC1
    main.c
)
# googlebenchmark
set(SRC2
   bentest.cpp
)
# 查找 benchmark 包
#find_package(benchmark REQUIRED)

# 设置测试程序的名字
add_executable(myBench ${SRC1})  #自定义 benchmark
#add_executable(gBench ${SRC2})  #googlebenchmark
# 链接之前生成的库
target_link_libraries(myBench mm)
#target_link_libraries(gBench mm)
#target_link_libraries(gBench benchmark::benchmark)

############## copy 文件到 bin 目录
# 设置执行文件的输出路径
set(SOURCE_DIR "${CMAKE_SOURCE_DIR}/src")
set(DESTINATION_DIR "${CMAKE_SOURCE_DIR}/bin")

# 确保输出目录存在
file(MAKE_DIRECTORY ${DESTINATION_DIR})

message(STATUS "CMAKE_SOURCE_DIR is: ${CMAKE_SOURCE_DIR}")

# 复制编译结果到 bin 目录
add_custom_command(
    OUTPUT ${DESTINATION_DIR}/myBench
    COMMAND ${CMAKE_COMMAND} -E copy $<TARGET_FILE:myBench>
${DESTINATION_DIR}/myBench
    DEPENDS myBench
    COMMENT "吴同馨: Copying myBench to bin directory"
    VERBATIM
)

# 复制 mm_opencl.cl 到 bin 目录
add_custom_command(
OUTPUT ${DESTINATION_DIR}/mm_opencl.cl
```

```
    COMMAND ${CMAKE_COMMAND} -E copy ${SOURCE_DIR}/mm_opencl.cl
${DESTINATION_DIR}/mm_opencl.cl
    DEPENDS myBench
    COMMENT "吴同馨: Copying mm_opencl.cl to bin directory"
    VERBATIM
)
# 创建一个自定义目标，它依赖于所有自定义命令的输出
add_custom_target(
    copy_to_bin ALL
    DEPENDS ${DESTINATION_DIR}/myBench
          ${DESTINATION_DIR}/mm_opencl.cl
)
```

第6章 源代码阐述与相应 benchmark 数据

6.1 概述

本章记述源代码、benchmark 数据以及说明。

6.2 常用宏定义

- 源代码

```
#define FUNC_INIT \
    if (A == NULL || A->data == NULL || B == NULL || B->data == NULL) \
        return NULL; \
    if (A->cols != B->rows) \
        return NULL; \
    Matrix *C = matrix_malloc(A->rows, B->cols);\
    if (C == NULL) \
            return NULL; \
```

- **说明**

每个矩阵乘法函数都用到了这个宏定义进行初始化，减少重复代码。

6.3 Bechmark

6.3.1 BENCHMARK_INSTALL 宏函数

- 源代码

```
#define BENCHMARK_INSTALL(BENCHMARK_NAME, FUNC) \
    f = FUNC;  \
    BM_each_cal( BENCHMARK_NAME, f);
```

- 说明

  用于安装被测试程序到 benchmark，实际上是定义一个函数指针，使其等于被调用函数。

6.3.2 (*p_function)函数指针类型定义

- 源代码

```
typedef Matrix* (*p_function)(const Matrix*, const Matrix*);
```

- 说明

  因为本项目所有函数具有相同类型的返回值和输入参数，所以定义了一个名为：的函数指针类型，包括定义了它的返回值和输入参数。

6.3.3 Warm_up 函数

- 源代码

```
void BM_warm_up(char* name, p_function pf){
    Matrix * A;
    Matrix * B;
    A = matrix_malloc(BM_MATRIX_SIZE[0],BM_MATRIX_SIZE[0]);
    B = matrix_malloc(BM_MATRIX_SIZE[0],BM_MATRIX_SIZE[0]);
    if(A == NULL || B == NULL){
        printf("memory allocing fail, stop this function\n");
        return;
    }
    BM_random_value(A);
    BM_random_value(B);
    Matrix * C = pf(A, B);
    matrix_free(A);
    matrix_free(B);
    matrix_free(C);
}
```

在 benchmark 中使用 warm_up 函数，需要预热的原因是：

在程序中我使用了 OpenBlas、OpenMP、threads、OpenCL，这些库是动态链接库，它们是在运行时刻才调用，此时会从硬盘上读动态库，而读硬盘的时间消耗较大。Warm-up 函数让程序预先运行一遍动态库，这样进行矩阵乘法计算时就消除了读以提高测试的准确性。

6.3.4 Benchmark 函数

- 源代码

```
void BM_each_cal(char* name,p_function f){
    BM_warm_up(name, f);
    for(int i = 0; i < BM_SIZE_NUM; i++){
        Matrix * A;
        Matrix * B;
        A = matrix_malloc(BM_MATRIX_SIZE[i],BM_MATRIX_SIZE[i]);
        B = matrix_malloc(BM_MATRIX_SIZE[i],BM_MATRIX_SIZE[i]);
        struct timespec start, end;
        long long elapsed_nanos;
```

```
        // 获取开始时间
        if (clock_gettime(CLOCK_REALTIME, &start) == -1) {
            perror("clock_gettime");
            matrix_free(A);
            matrix_free(B);
            return;
        }
        Matrix *C = f(A,B);
        // 获取结束时间
        if (clock_gettime(CLOCK_REALTIME, &end) == -1) {
            perror("clock_gettime");
            matrix_free(A);
            matrix_free(B);
            return;
        }
        //纳秒
        elapsed_nanos = (end.tv_sec - start.tv_sec) * 1000000000LL +
(end.tv_nsec - start.tv_nsec);
        printf("\033[32m%s\033[0m",name);
        printf("%*s", (int)(24-strlen(name)),"");
        printf("%*d",10,BM_MATRIX_SIZE[i]);
        printf("%*lld\n",14,elapsed_nanos/1000);

        matrix_free(A);
        matrix_free(B);
        matrix_free(C);
    }
}
```

- 说明

  测试函数指针的执行时间。

6.4 main 函数

- 源代码

```
int main(int argc, char * argv[]){
    p_function f;
    //正式测试
printf("\033[32m-----------------------------------------------------\033
[0m\n");
    printf("\033[32mbenchmark              size          time(us)
\033[0m\n");
```

```
printf("\033[32m----------------------------------------------------\033
[0m\n");

    BENCHMARK_INSTALL("BM_OpenBlas", matmul_openblas)
    BENCHMARK_INSTALL("BM_PLAIN", matmul_plain)
    BENCHMARK_INSTALL("BM_PLAINikj", matmul_plain_ikj)
    BENCHMARK_INSTALL("BM_PLAINbk", matmul_plain_bk)
    BENCHMARK_INSTALL("BM_PLAINbkc", matmul_plain_bkc)
    BENCHMARK_INSTALL("BM_SIMD_bkc", matmul_SIMD_bkc)
    BENCHMARK_INSTALL("BM_SIMD_OpenMP", matmul_SIMD_bkc_omp)
    BENCHMARK_INSTALL("BM_OpenMP", matmul_bk_omp)
    BENCHMARK_INSTALL("BM_THREADS", matmul_threads)
    BENCHMARK_INSTALL("BM_OpenCL", matmul_opencl)

printf("\033[32m----------------------------------------------------\033
[0m\n");
    return EXIT_SUCCESS;
}
```

- 说明

main 函数 benchmark 的打印格式如下：

```
-----------------------------------------------------
benchmark                  size              time(us)
-----------------------------------------------------
BM_OpenBlas                          16             13
BM_PLAIN                             16              2
BM_PLAINikj                          16              1
BM_PLAINbk                           16              1
BM_PLAINbkc                          16              1
BM_SIMD_bkc                          16              3
BM_OpenMP                            16          72437
BM_THREADS                           16          11626
BM_OpenCL/16       kernel耗时： 43 μs
BM_OpenCL                            16         159515
-----------------------------------------------------
```

6.5 OpenBlas

- 源代码

```
Matrix * matmul_openblas(const Matrix* A, const Matrix* B) {
    FUNC_INIT

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                A->cols,B->rows, C->rows, 1.0f,
                A->data, C->rows, B->data, C->rows,
```

```
            0.0f, C->data,C->rows);
    return C;
}
```

- Benchmark 数据

benchmark 数据:	单位 us				
被测函数/矩阵尺寸	16	128	1024	8192	65536
matmul_openblas	1.00	1,262	14,721	511,331	499,189,704

- 说明

OpenBlas 作为成熟的计算库，其计算效率是作为标准存在，从实测效果来看，采用 OpenBlas 进行矩阵乘法计算的速度在矩阵变大时候速度比其它的优化要快，是最快的。其原因参考第 2 章。

在矩阵规模 128 时，其计算速度远低于除朴素算法以外的其他方法，原因应该是 OpenBlas 在利用多线程时有额外管理多线程的开销，而这些开销占用了大部分计算时间。

6.6 朴素算法

- 源代码

```
Matrix* matmul_plain(const Matrix *A, const Matrix *B) {
    FUNC_INIT

    for (size_t i = 0; i < A->rows; i++){
        for (size_t j = 0; j < B->cols; j++){
            C->data[i*B->cols+j] = 0;//初始化结果矩阵
            for (size_t k = 0; k < A->cols; k++){
                C->data[i*B->cols+j] += A->data[i*A->cols+k] *
                                        B->data[k*B->cols+j];
            }
        }
    }
    return C;
}
```

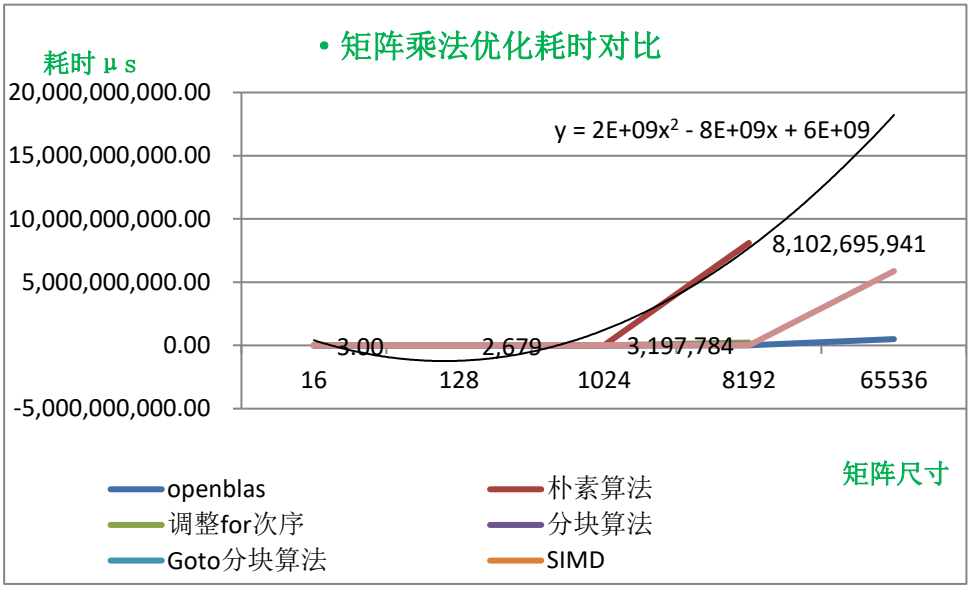
- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_plain	3.00	2,679	3,197,784	8,102,695,941	8,589,410,310,000,000,000

*注：绿色代表拟合数据，下同。

- **说明**

该算法平均效率最低，计算 8k、64k 矩阵时程序陷于停滞。因此，采用拟合的方式预测 64k 矩阵乘法运所需时间。



拟合朴素算法的趋势线如上，根据其公式预测 64k 矩阵乘法的运行时间。

预测时间：8.58941031E+18 us。

6.7 调整循环次序

- 源代码

```
Matrix* matmul_plain_ikj(const Matrix *A, const Matrix *B) {
    FUNC_INIT

    for (size_t i = 0; i < A->rows; i++){
        for (size_t k = 0; k < A->cols; k++){
            for (size_t j = 0; j < B->cols; j++){
                C->data[i*B->cols+j] += A->data[i*A->cols+k] *
                                        B->data[k*B->cols+j];
            }
        }
    }
    return C;
}
```

- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_plain_ikj	2.00	271	177,059	228,757,051	2,424,987,308,330,000

- **说明**

在矩阵乘法的三层 for 循环中，可以按照 i、j、k 嵌套进行循环，这是最普通的方式，还可以对 i、j、k 进行组合式改变，改变循环次序后，计算效率有提升也有下降，其中，i、k、j 方式进行循环，程序速度提升最快，在 1024 矩阵规模时 i、j、k 的朴素算法有 10 倍以上的提高（这次计算速度的提升幅度是所有方法中最大的，所以，后面的循环次序均以 ikj 方式进行）。之所以 i、k、j 的循环方式速度得到显著提升，其原因应该有如下几个：

1) 比起 ijk 的顺序，编译器将 AVX2 的指令编译进了执行程序，如向量乘法指令 vmulps、向量加法指令 vaddps 等，这就为矩阵乘法提速一次，这也能够解释为什么在显式地运用 avx2 指令编程时计算速度并没有什么提升。

采用如下反汇编指令：

```
objdump -d myBench> x.txt
```

在对应函数代码段可以看到 AVX2 的汇编指令：

```
b9b6:   0f 86 89 01 00 00       jbe    bb45 <matmul_plain_ikj+0x365>
b9bc:   c4 e2 7d 18 0e          vbroadcastss (%rsi),%ymm1
b9c1:   31 d2                  xor    %edx,%edx
b9c3:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
b9c8:   c5 f4 59 04 11          vmulps (%rcx,%rdx,1),%ymm1,%ymm0
b9cd:   c5 fc 58 04 10          vaddps (%rax,%rdx,1),%ymm0,%ymm0
b9d2:   c5 fc 11 04 10          vmovups %ymm0,(%rax,%rdx,1)
    b9d7:   48 83 c2 20             add    $0x20,%rdx
```

2) 不同的循环次序矩阵在缓存的位置可能导致缓存未命中。

就 i, j, k 循环顺序而言，这种顺序的好处是它保持了 A 矩阵行的连续性访问，但每次内部循环时，它都需要跳到 B 矩阵的不同列，这可能导致缓存未命中，因为列元素在内存中可能不是连续存储的。如果矩阵 B 不是按列主序存储的（即不是列优先），那么这种访问模式可能会导致较差的缓存局部性。

就 i, k, j 循环顺序而言，这种顺序保持了 A 矩阵行的连续性访问，同时对于每个 k，它连续访问 B 矩阵的列元素。如果矩阵 B 是按列主序存储的，那么这种访问模式可以最大化缓存的利用率，因为 B 的列元素在内存中是连续存储的。对于 C 矩阵，每次内部循环都会更新同一行的不同列，这也有利于缓存的利用，因为行元素在内存中通常是连续存储的。也就是说，调整循环次序的实质是内存访问优化。

关于缓存的问题，参考附录 4。

同时，我也对 kij 循环算法进行了尝试，但是效果没有 ikj 好，它和 plain 算法在

运行时间上基本没有区别，因此不进行过多描述。

6.8 分块算法

将矩阵分成小块，如果数据块的大小与缓存大小相匹配，那么可以将整个数据块加载到缓存中，从而减少从主存中读取数据的次数，提高计算效率。分块算法允许我们根据缓存的大小来优化数据块的大小。

6.8.1 简单分块算法

- 源代码

```
 Matrix* matmul_plain_bk(const Matrix *A, const Matrix *B) {
    FUNC_INIT

    for (size_t i0 = 0; i0 < A->rows; i0 += 16){
        for (size_t k0 = 0; k0 < B->cols; k0 += 16){
            for (size_t i = 0; i < 16; i++){
                for (size_t k = 0; k < 16; k++){
                    for (size_t j = 0; j < B->cols; j++){
                        C->data[(i0+i)*B->cols+j] +=
A->data[(i0+i)*B->cols+k0+k] * B->data[(k0+k)*B->cols+j];
                    }
                }
            }
        }
    }
    return C;
}
```

- benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_plain_bk	13.00	233	106,602	74,910,348	34.102.924.340.389

- **说明**

简单分块算法，因为时间关系并没有根据特定机器的缓存来设计分块的大小，而是笼统地采用同样的分块。

分块算法在矩阵乘法中能够提升计算效率的主要原因在于它有效地减少了计算过程中的内存访问次数，并且可以帮助提高计算的并行性。

6.8.2 Goto 法

- 源代码

```
Matrix* matmul_plain_bkc(const Matrix *A, const Matrix *B) {
    FUNC_INIT

    float temp[16*16];
    float *A1 = aligned_alloc(1024, A->rows*16*sizeof(float));
    float *C1 = aligned_alloc(1024, A->rows*A->rows*sizeof(float));

    for (size_t k0 = 0; k0 < A->rows; k0+=16) {

        for (size_t i = 0; i < A->rows; i++){
            for (size_t k = 0; k < 16; k++){
                A1[i*16+k] = A->data[i*B->cols+k0+k];
            }
        }
        for (size_t j0 = 0; j0 < B->cols; j0 += 16) {
            for (size_t j = 0; j < 16; j++){
                for (size_t k = 0; k < 16; k++){
                   temp[k*16+j] = B->data[(k0+k)*B->cols+j0+j];
                }
            }
            float *p = C1 + j0*A->rows;
            for (size_t i = 0; i < A->rows; i++) {
                for (size_t j = 0; j < 16; j++) {
                    float sum = 0;
                    for (size_t k = 0; k < 16; k++){
                        sum += A1[i*16+k]*temp[k*16+j];
                    }
                    *p+=sum;
                    p++;
                }
            }
        }
    }

    float *p = C1;
    for (size_t j0 = 0; j0 < B->cols; j0+=16){
        for (size_t i = 0; i < A->cols; i++){
            for (size_t j = 0; j < 16; j++){
                C->data[i*B->cols+j0+j] = *p;
                p++;
            }
```

```
        }
    }
    free(A1);
    free(C1);
    return C;
}
```

- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_plain_bkc	2.00	211	97,430	45,292,514	1,260,059,324,888

- 说明

根据网上的介绍，Goto 的算法是优化的 Strassen 算法。他针对 Strassen 算法进行了优化，通过减少加法和减法操作的次数，提高了算法的实际性能。这些优化使得 Goto 的算法在大规模矩阵乘法运算中表现出色。在这里，采用他的 GEPB 算法。

使用了该算法的计算效率得到了 10%的提升。这个提升似乎不是很高。

6.9 SIMD-X86

- 源代码

```
Matrix* matmul_SIMD_bkc(const Matrix *A, const Matrix *B) {
#ifdef WITH_AVX2
    FUNC_INIT

    float temp[16*16];

    float *A1 = aligned_alloc(1024, A->rows*16*sizeof(float));
float *C1 = aligned_alloc(1024, A->rows*A->rows*sizeof(float));

    __m256 a1,t;
    for (size_t k0 = 0; k0 < A->rows; k0+=16) {

        for (size_t i = 0; i < A->rows; i++){
            for (size_t k = 0; k < 16; k++){
                A1[i*16+k] = A->data[i*B->cols+k0+k];
            }
        }
        for (size_t j0 = 0; j0 < B->cols; j0 += 16) {
            for (size_t j = 0; j < 16; j++){
                for (size_t k = 0; k < 16; k++){
                    temp[k*16+j] = B->data[(k0+k)*B->cols+j0+j];
                }
```

```
            }
            float *p = C1 + j0*A->rows;

            for (size_t i = 0; i < A->rows; i++) {
                for (size_t j = 0; j < 16; j++) {
                    __m256 acc = _mm256_setzero_ps();
                    for (size_t k = 0; k < 16; k+=8){
                        //sum += A1[i*16+k]*temp[k*16+j];
                        a1 = _mm256_loadu_ps(A1+i*16+k);
                        t = _mm256_loadu_ps(temp+k*16+j);
                        acc = _mm256_add_ps(acc, _mm256_mul_ps(a1, t)); // 累加
乘积
                    }

                    float result[8]; // 存储 8 个浮点数的结果
                    _mm256_storeu_ps(result, acc); // 将累加器的值存储到数组中
                    for(int l = 0; l < 8 && i * A->rows+l < A->rows && j+l < A->rows;
++l) {
                        p[i*A->rows+j+l] += result[l]; // 将结果累加到 C 的相应位置
                    }
                }
            }
        }
    }

    float *p = C1;
    for (size_t j0 = 0; j0 < B->cols; j0+=16){
        for (size_t i = 0; i < A->cols; i++){
            for (size_t j = 0; j < 16; j++){
                C->data[i*B->cols+j0+j] = *p;
                p++;
            }
        }
    }

    free(A1);
    free(C1);
    return C;
#endif
}
```

- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_SIMD_bkc	2.00	268	113,755	51,179,320	1,429,585,104,532

- 说明

采用 SIMD 进行优化并没有带来计算速度的提升，相反却降低了计算速度。原因估计是编译器对不是 SIMD 的代码进行了 SIMD 化，那么再显式地写 SIMD 等于是做了重复劳动，会造成计算速度下降。具体内容参考 6.6 调整循环次序中的说明。

6.10 OpenMP

6.10.1 OpenMP 优化分块乘法

- 源代码

```
    Matrix* matmul_bk_omp(const Matrix *A, const Matrix *B) {
    FUNC_INIT

    #pragma omp parallel
    for (size_t i0 = 0; i0 < A->rows; i0 += 16){
        for (size_t k0 = 0; k0 < B->cols; k0 += 16){
            size_t i;
            size_t j;
            size_t k;
            #pragma omp for private(i,j,k)
            for (i = 0; i < 16; i++){
                for (k = 0; k < 16; k++){
                    for (j = 0; j < B->cols; j++){
                        C->data[(i0+i)*B->cols+j] +=
A->data[(i0+i)*B->cols+k0+k] * B->data[(k0+k)*B->cols+j];
                    }
                }
            }
        }
    }
    return C;
}
```

- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_bk_omp	6802	152055	33,207,878	463,623,135	37,937,469,570

- **说明**

分块算法 omp 优化比简单分块算法运行时间更慢，在矩阵小的时候可能是因为 OpenMP 需要为每一个并行区域创建和销毁线程。如果分块太小或者并行区域太小，那么线程创建和销毁的开销可能会超过并行化带来的性能提升。而且在使用 OpenMP 进行并行化时，如果任务的负载不均衡，即某些线程的工作量比其他线程大，那么可能导致一些线程处于空闲状态，从而降低了整体的并行效率。

6.10.2 OpenMP 的 SIMD 优化

- 源代码

```
Matrix* matmul_SIMD_bkc_omp(const Matrix *A, const Matrix *B) {
#ifdef WITH_AVX2
    FUNC_INIT

    float temp[16*16];

    float *A1 = aligned_alloc(1024, A->rows*16*sizeof(float));
    float *C1 = aligned_alloc(1024, A->rows*A->rows*sizeof(float));
    __m256 a1,t;
    #pragma omp parallel for
    for (size_t k0 = 0; k0 < A->rows; k0+=16) {
        for (size_t i = 0; i < A->rows; i++){
            for (size_t k = 0; k < 16; k++){
                A1[i*16+k] = A->data[i*B->cols+k0+k];
            }
        }
        for (size_t j0 = 0; j0 < B->cols; j0 += 16) {
            for (size_t j = 0; j < 16; j++){
                for (size_t k = 0; k < 16; k++){
                   temp[k*16+j] = B->data[(k0+k)*B->cols+j0+j];
                }
            }
            float *p = C1+j0*A->rows;
            for (size_t i = 0; i < A->rows; i++) {
                for (size_t j = 0; j < 16; j++) {
                    __m256 acc = _mm256_setzero_ps();
                    for (size_t k = 0; k < 16; k+=8){
                        a1 = _mm256_loadu_ps(A1+i*16+k);
                        t = _mm256_loadu_ps(temp+k*16+j);
                        acc = _mm256_add_ps(acc, _mm256_mul_ps(a1,t));
                    }
                    float result[8] = {0};
```

```
                _mm256_storeu_ps(result, acc);
                for (size_t l = 0; l < 8 && i*A->rows+l < A->rows && j+1 <
A->rows; ++l)
                {
                    p[i*A->rows+j+l] += result[l];
                }
            }
        }
    }

    float *p = C1;
    for (size_t j0 = 0; j0 < B->cols; j0+=16){
        for (size_t i = 0; i < A->cols; i++){
            for (size_t j = 0; j < 16; j++){
                C->data[i*B->cols+j0+j] = *p;
                p++;
            }
        }
    }
    free(A1);
    free(C1);
    return C;
#endif
}
```

- Benchmark **数据**

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_SIMD_bkc_omp	21.00	823	243,940	20,649,597	38,537,479,661

- **说明**

1. OpenMP 是采用了多线程，但奇怪的是其效率没有明显提升，反而不如后面自定义多线程算得快。(也可能是写法稍有问题) 但是相比于 SIMD，OpenMP 在运行较大矩阵像 8k*8k 矩阵时速度稍快。
2. OpenMP 我在远端服务器测了几组数据，但是每次耗时都略有区别。这可能是因为我写的 OpenMP 程序运行不太稳定，且在远端服务器运行时，可能因为线上运行人数不同造成运行时间不同，但是变化范围较为合理。这里我只放了一次的数据，以下数据均采用该测试结果。
3. OpenMP 的 SIMD 优化较 OpenMP 的分块算法优化快一些，但是效果不是特

别明显。下文均采用 OpenMP 的 SIMD 优化的数据作为 OpenMP 优化方法的数据。

6.11 自定义多线程

- 源代码

```
typedef struct {
    const Matrix *A;
    const Matrix *B;
    Matrix *C;
    size_t start_col;
    size_t end_col;
} ThreadArgs;

// 线程函数，用于计算矩阵乘法的一部分
void *matmul_thread(void *arg) {
    struct ThreadArgs *thread_args = (struct ThreadArgs *)arg;
    const Matrix *A = thread_args->A;
    const Matrix *B = thread_args->B;
    Matrix *C = thread_args->C;
    size_t start_row = thread_args->start_row;
    size_t end_row = thread_args->end_row;
    size_t A_cols = A->cols;
    size_t B_cols = B->cols;

    for (size_t i = start_row; i < end_row; i++) {
        for (size_t k = 0; k < A_cols; k++) {
            for (size_t j = 0; j < B_cols; j++) {
                C->data[i * B_cols + j] += A->data[i * A_cols + k] * B->data[k
* B_cols + j];
            }
        }
    }

    pthread_exit(NULL);
}

Matrix* matmul_threads(const Matrix *A, const Matrix *B) {
    FUNC_INIT
    // 线程参数数组
    struct ThreadArgs *thread_args = malloc(NUM_THREADS * sizeof(struct
ThreadArgs));
    pthread_t *threads = malloc(NUM_THREADS * sizeof(pthread_t));
```

```
    // 分配每个线程处理的行数
    size_t rows_per_thread = A->rows / NUM_THREADS;
    size_t remainder = A->rows % NUM_THREADS;

    size_t start_row = 0;
    for (size_t t = 0; t < NUM_THREADS; t++) {
        size_t end_row = start_row + rows_per_thread;
        if (t < remainder) {
            end_row++; // 将剩余的行分配给前面的线程
        }
        thread_args[t].A = A;
        thread_args[t].B = B;
        thread_args[t].C = C;
        thread_args[t].start_row = start_row;
        thread_args[t].end_row = end_row;

        pthread_create(&threads[t], NULL, matmul_thread, &thread_args[t]);
        start_row = end_row;
    }
    // 等待所有线程完成
    for (size_t t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }
    // 释放线程参数和线程句柄数组
    free(thread_args);
    free(threads);

    return C;
```

- Benchmark 数据

benchmark 数据:	单位 us				
项目/矩阵尺寸	16	128	1024	8192	65536
matmul_threads	1,954.00	1,991	26,778	8,133,615	5,874,706,800

- **说明**

由于使用 OpenMP 的优化效果不明显，因此我又实现了自定义多线程的优化。

在自定义多线程中，32 个线程进行并行计算，将计算速度比 OpenMP 提升了 60%，多个线程去完成同一件事情比单个线程肯定要快很多。

第7章 内存泄漏检测

C/C++内存泄漏测试采用 valgrind 指令。以下是 valgrind 指令命令形式、valgrind 检测结果报告。

- 执行指令：

```
valgrind --leak-check=full --show-leak-kinds=all ./main > result.txt 2>&1
```

- 执行上述命令的结果：

```
==14496== LEAK SUMMARY:
==14496==    definitely lost: 0 bytes in 0 blocks
==14496==    indirectly lost: 0 bytes in 0 blocks
==14496==    possibly lost: 6,080 bytes in 19 blocks
==14496==    still reachable: 6,192 bytes in 4 blocks
==14496==         suppressed: 0 bytes in 0 blocks
```

从内存检测的结果来看，没发生内存泄漏。

Possibly lost 和 still reachable 是由 openMP 导致的问题，并不影响运行结果。

第8章 测时结果对比与综合分析

8.1 矩阵乘法耗时对比表格

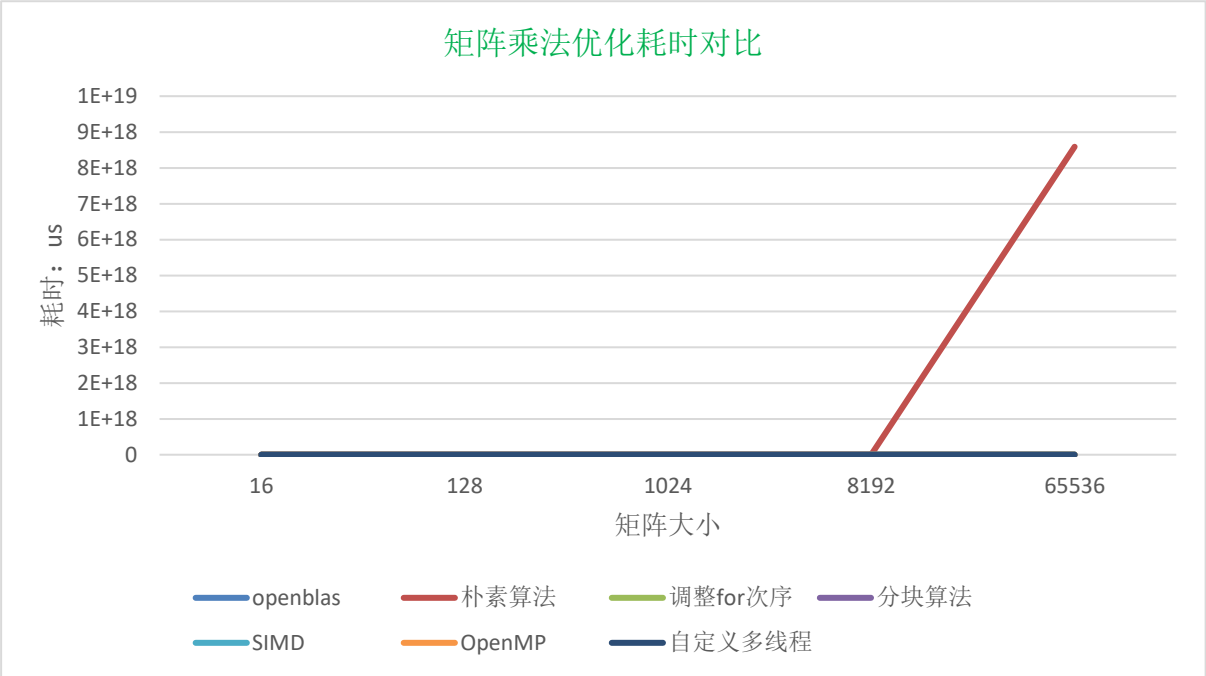
根据表格可以看出，从朴素算法开始直到自定义多线程，计算时间逐步缩短向 openblas 的计算速度靠近。其中单次提速最快的是改变 for 循环 ijk 的次序为 ikj，速度提升了 10 倍以上；优化速度最快的是自定义线程，计算速度达到了 OpenBlas 的 10~50% 左右。（根据矩阵大小的不同而不同）

被测函数/矩阵尺寸	16	128	1024	8192	65536
matmul_openblas	1	1,262	14,721	511,331	499,189,704
matmul_plain	3	2,679	3,197,784	8,102,695,941	8,589,410,310,000,000,000
matmul_plain_ikj	2	271	177,059	228,757,051	2,424,987,308,330,000
matmul_plain_bk	13	233	106,602	74,910,348	34.102.924.340.389
matmul_plain_bkc	2	211	97,430	45,292,514	1,260,059,324,888
matmul_SIMD_bkc	2	268	113,755	51,179,320	1,429,585,104,532
matmul_SIMD_bkc_omp	21	823	243,940	20,649,597	38,537,479,661
matmul_threads	1,954.00	1,991	26,778	8,133,615	5,874,706,800

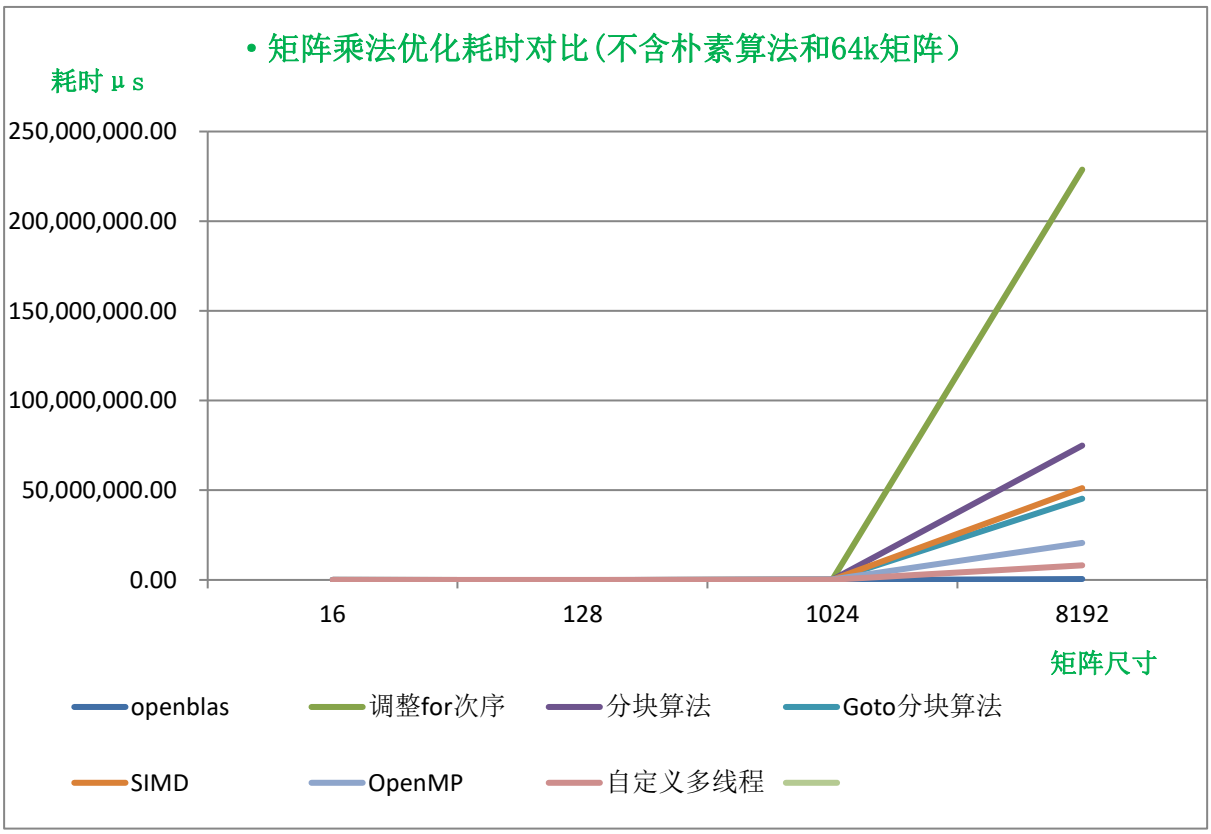
*注：绿色字体为拟合估计消耗时间

8.2 矩阵乘法耗时对比曲线

本曲线是将上述表格图形化。由于朴素算法和 64k 计算耗时巨大，造成了其它优化程序的曲线被压扁，看不清。



8.3 矩阵乘法耗时对比曲线（不含朴素算法和 64k 矩阵）



8.4 对比表格和对比曲线的基本解读

OpenBlas 在矩阵大小为 128 时，速度慢于其它几种优化，其原因是 openBlas 的诸多优化比如多线程需要额外的时间，其时间远大于 128 的朴素算法；

调整 ijk 循环顺序对速度的提升很大，同样的算法，只是朴素算法的约 1/40；分块算法对计算速度有 10%左右的提升；

多线程的提速效果明显。

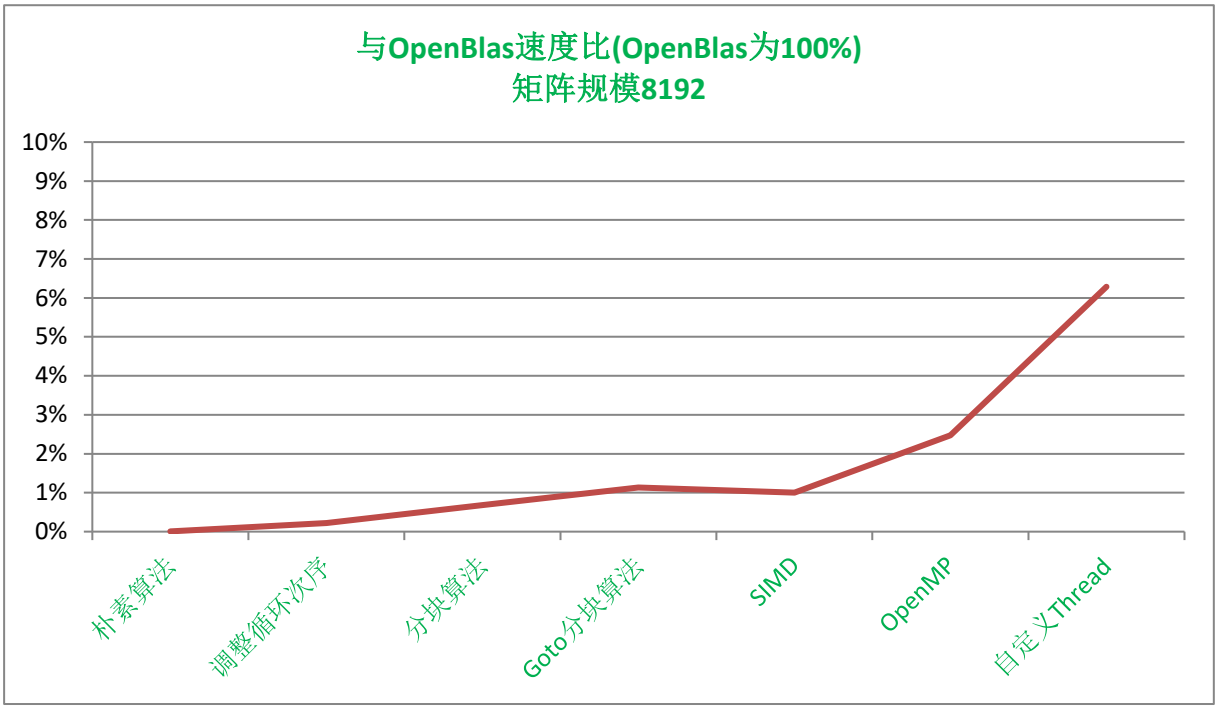
8.5 优化方法的提速效果分析

8.5.1 原始数据对比表

benchmark 数据:	单位 us	机器：学校远端服务器		
被测函数/矩阵尺寸	8192	提速比	与 OpenBlas 比	备注
matmul_openblas	511,331			openblas
matmul_plain	8,102,695,941		0%	朴素算法
matmul_plain_ikj	228,757,051	97%	0%	调整循环次序
matmul_plain_bk	74,910,348	67%	1%	分块算法
matmul_plain_bkc	45,292,514	40%	1%	Goto 分块算法
matmul_SIMD_bkc	51,179,320	-13%	1%	SIMD
matmul_SIMD_bkc_omp	20,649,597	60%	2%	OpenMP
matmul_threads	8,133,615	61%	6%	自定义 Thread
提速比指的是下一个优化对上一个优化的计算速度提升比率				

8.5.2 与 OpenBlas 速度比曲线

以下是各优化程序的速度对比曲线图，OpenBlas 作为标准是 100%。



由图中我们可以看到，采用自定义多线程方式优化，达到了自定义优化方法的最快速度，根据矩阵大小的不同，计算速度能达到 OpenBlas 的 10%~50%之间，这显示出内存优化、分块算法、多线程对矩阵乘法的叠加优化效应。

8.5.3 提速力度差分曲线

从提速力度的曲线图可以直观地看到，有 3 个点对提速的效果尤其明显：

第一，最大幅度提速发生在 for 循环的 ijk 变为 ikj，这一步提升了 90%以上的速度。for 循环次序的改变其实是内存布局，使得减少了内存延迟，提高了缓存命中率，从而提高了运算速度。这里可以理解为内存布局在高性能计算的软件算法中具有至关重要的决定性作用。

第二，提速幅度很大的是多线程，提升幅度在之前优化的基础提升 70%的幅度。这说明，多线程并行计算对计算速度的提升帮助是很大的。这很容易理解：本来由一个线程计算一个矩阵，现在分成多个线程同时计算每一小块矩阵，然后再拼起来，理论上不考虑线程管理开销的话，有 N 个线程，速度能提升 N-1 倍。由此我们可以推理：如果使用多台计算机进行矩阵乘法并行计算的话，速度会提升更大。所以不难理解深蓝下国际象棋、AlphaGo 下围棋其背后 应该有多台计算机同时在进行计算。当然多台计算机并行计算和单台计算机多线程并行计算也存在着线程管理和网络通信需要数据传输的时间问题，在小规模矩阵计算时是不合算的，但当矩阵规模大到一定程度时，多台计算机并行计算应该能极大地提升计算速度。

第三，分块算法对提速贡献了 20%的力量。不用对矩阵乘法采用复杂的 Strassen 算法、Coppersmith-Winograd 算法，采用分块算法再配合以 SIMD 等方法，也能有效地提升矩阵乘法的计算速度。

最后，我们看到 SIMD 对提升速度帮助不是很大，通过反汇编程序我们发现，编译器实际上将大多数即便没有显式用 AVX2 指令的 C 程序也编译成了含有 AVX2 指令的程序，也就是说 SIMD 已经被编译器安排进我们的程序了，再显式地用 SIMD 指令也就起不到进一步提速的作用了。

8.6 优化方法提速效果重要结论

通过本项目的 benchmark 数据，有以下结论：

- 内存访问优化对性能优化的贡献最大；对应就是改变 ijk 循环次序。
- 多线程技术对性能优化的贡献其次；
- 自动向量化技术对性能优化也有贡献，只是编译器在很多情况下对程序自动做了向量化，显式的向量化技术对速度的影响往往是增加了执行时间；
- 分块算法对优化的作用也很显著。
- 上述结论和 2.2 OpenBlas 优化方法中所阐述的内容高度一致并做了量化。

这些结论表明：openBlas 所采用的优化方法对提速确实有很好的指导作用。

8.7 编译优化参数对耗时影响

为了提升矩阵乘法的运算速度，经过反复测试，最终在-O3 的基础上增加了如下

的编译参数：

```
add_definitions(-funroll-loops)
add_definitions(-ftree-vectorize)
add_definitions(-falign-loops=1024)
add_definitions(-falign-functions=1024)
```

其中-falign-functions 能较大幅度的提升速度，大约提速 20%。具体数据就不一一列出了。

第9章 OpenCL 矩阵乘法基本方法

9.1 概述

由于我没有找到支持 ARM 架构的机器进行 ARM 优化的测试，因此我只是写了使用 ARM 的代码，但是并没有进行测试。（具体 ARM 代码见上传的文件 matrixMul_ARM.c，就不在这里放了。）不过我尝试使用了 OpenCL 进行矩阵乘法。

由于区块链和 AI 的计算经常提及到运用 GPU 进行高速计算，因此想到了矩阵乘法是否可以用到 GPU 来优化性能。经过查阅资料，了解到了如何利用 OpenCL 进行 GPU 编程，因此在本项目中也尝试用了 OpenCL 来进行矩阵乘法的计算。

由于机器用的是集成线卡，所以计算速度并非象期待得那样快，更多地是了解了 OpenCL 的使用方法。

OpenCL 的使用方法和编程框架参考附录 12、13。

9.2 C 代码

OpenCL 代码看起来很长，但很多是为了与 GPU 通信而进行的设置，设置的目的在于把内核代码动态编译后送给 GPU 去执行（原理参考附录 3），然后把 GPU 计算结果也就是 C 矩阵返回给 C 程序。此处给出主体部分，包含了对内核代码编译并提交编译后的代码给 GPU 执行：

```
Matrix* matmul_opencl( const Matrix* A, const Matrix* B ) {
    cl_context context = 0;
    cl_device_id device = 0;
    cl_command_queue commandQueue = 0;
    cl_program program = 0;
    cl_kernel kernel = 0;
    cl_mem memObjects[3] = { 0, 0, 0 };
    cl_int errNum;
    cl_event events[1];
    cl_event event;
    const char* filename = "mm_opencl.cl";

    // 1 选择 OpenCL 平台并创建一个上下文
    context = CreateContext();
```

```
    // 2 创建设备并创建命令队列
    commandQueue = CreateCommandQueue(context, &device);
    // 3 创建和构建程序对象
    program = CreateProgram(context, device, filename);
    // 4 创建 OpenCL 内核并分配内存空间
    kernel = clCreateKernel(program, "matrixMul_kernel", NULL);
    // 5 创建内存对象
    if (!CreateMemObjects(context, memObjects, A, B)) {
        Cleanup(context, commandQueue, program, kernel, memObjects);
        return NULL;
    }
    // 6 设置内核数据
    errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
    errNum = clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
    errNum = clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
    errNum = clSetKernelArg(kernel, 3, sizeof(int), &(A->rows));
    errNum = clSetKernelArg(kernel, 4, sizeof(int), &(B->cols));
    errNum = clSetKernelArg(kernel, 5, sizeof(int), &(B->rows));
    size_t globalWorkSize[2];
    globalWorkSize[0] = A->rows;
    globalWorkSize[1] = B->cols;
    // 7 执行内核
    errNum = clEnqueueNDRangeKernel(commandQueue,
                                    kernel,
                                    2,
                                    NULL,
                                    globalWorkSize,
                                    NULL,
                                    0,
                                    NULL,
                                    &events[0]);
    if (errNum != CL_SUCCESS) {
        perror("clEnqueueNDRangeKernel");
        return NULL;
    }

    // 阻塞，由于执行 GPU 计算是非阻塞方式，这里采取阻塞，保证程序时间计算正确
    errNum = clWaitForEvents(1, &events[0]);
    if (errNum != CL_SUCCESS) {
        perror("clWaitForEvents");
        return NULL;
    }
    // -------- 获取 OpenCL kernel 事件的时间戳
    cl_ulong kernel_start, kernel_end;
```

```
    clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &kernel_start, NULL);
clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &kernel_end, NULL);

    // 8 读取执行结果
    Matrix* C;
    if(errNum == 0){
        C = matrix_malloc(A->cols, B->rows);
    }else{
        return NULL;
    }
    errNum = clEnqueueReadBuffer(commandQueue,
                                 memObjects[2],
                                 CL_TRUE,
                                 0,
                                 B->cols * A->rows * sizeof(float),
                                 C->data,
                                 0,
                                 NULL,
                                 NULL);

    errNum = clReleaseEvent(events[0]);
    if (errNum != CL_SUCCESS) {
        perror("clReleaseEvent");
        return NULL;
    }
    // 9 释放 OpenCL 资源
    Cleanup(context, commandQueue, program, kernel, memObjects);

    //------------------------------- 以下，打印 Kernel 时间
-------------------------------

    // -------- 打印标题，红色
    printf("\033[31mBM_OpenCL/%ld\033[0m     ",A->rows);

    // -------- 打印 GPU 内核计算所需时间 品红色
    cl_ulong _kernel_elapsed_time_ns = (kernel_end - kernel_start)/1000; //
纳秒->微秒
    printf("\033[35mkernel 耗时: %ld μs\033[0m     \n",
_kernel_elapsed_time_ns);

    return C;
}
```

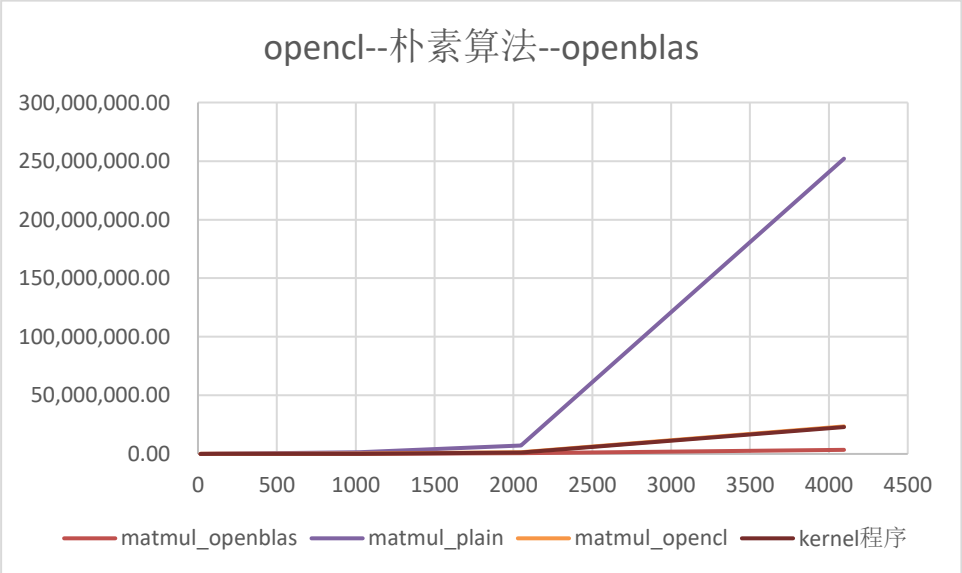
上面的代码完成了找到 GPU 设备、准备内存、执行内核代码的编译和执行、读取计算结果、释放 OpenCL 资源。

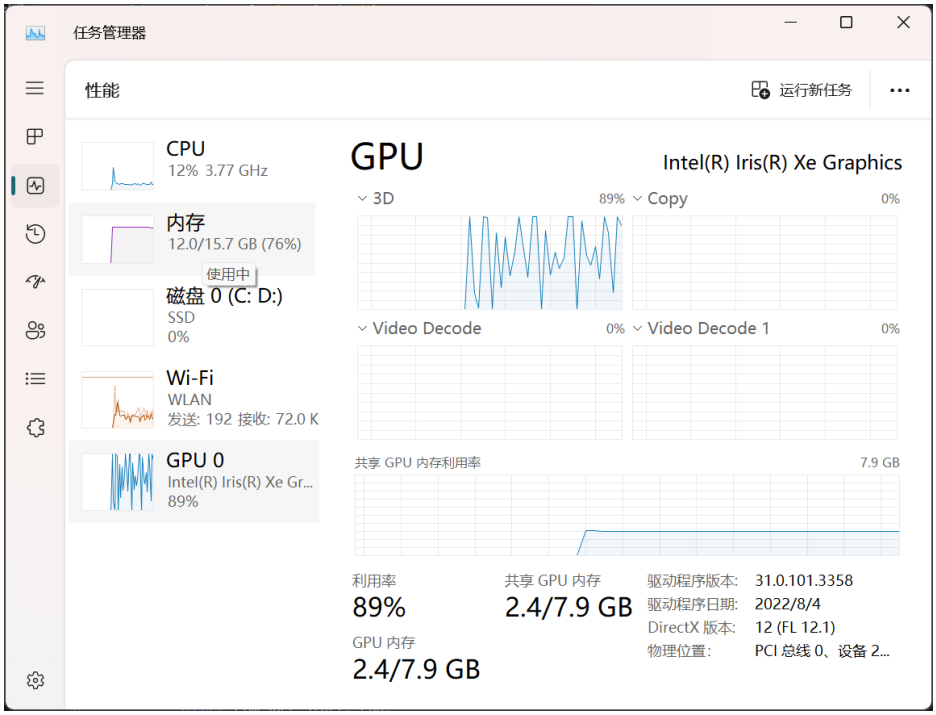
9.3 内核代码

```
__kernel void matrixMul_kernel(__global const float *A,
                               __global const float *B,
                               __global float *C,
                               int  result_matrix_row,
                               int  result_matrix_col,
                               int  compute_size){
    int row = get_global_id(0);
    int col = get_global_id(1);
     float sum = 0;
    for(int i=0; i<compute_size; i++){
        sum += A[row*compute_size+i] * B[i*result_matrix_col+col];
    }
     C[row*result_matrix_col+col] = sum;
}
```

9.4 Benchmark 数据

benchmark 数据:	单位 us	机器: 我的电脑			
被测函数/矩阵尺寸	16	128	1024	2048	4096
matmul_openblas	6.00	4,953	126,005	685,967	3,480,000
matmul_plain	8.00	1,848	1,508,198	7,334,326	252,072,262
matmul_opencl	118,925.00	123,905	281,683	1,411,367	23,494,152
kernel 程序	12.00	406	137,488	1,211,159	22,935,767





GPU 进行矩阵运算时的状态。

9.5 说明

Kernel 代码就是一个矩阵乘法的朴素算法代码，它由 C 程序动态读入并编译，然后提交 GPU 执行，GPU 据此完成矩阵乘法运算。

可以看到：

1. 矩阵规模在 16 和 128 时，opencl 耗时很多；在规模为 1024 时耗时未增加很多。可能原因是：opencl 在进行计算时，与 GPU 数据交换时会占用较多时间，在小规模矩阵计算时这部分时间占比较大；当矩阵规模为 1024 时，计算速度就变得较快；
2. Kernel 程序本身耗时更少更贴近 openblas，说明 GPU 计算速度高于 CPU；
3. 但是整个 opencl 没想象中快，可能因为是集成显卡而不是独立显卡。

限于时间有限，并未深入了解 GPU 和 opencl，没有采用更加合适的优化设计。

第10章 项目总结

本次项目通过实现对矩阵乘法的优化，学习了计算机内存布局、利用缓存提速，多线程编程，还学习了 GPU 的使用方法，认识到了计算机软件与硬件在计算效率方面的相互作用。

通过本项目的实施，在 project1 和 project2 的基础上，继续学习了 C/C++基本编程方法。

1. 巩固了第三方 benchmark 工具 googlebenchmark 的使用方法；
2. 学习了 SIMD 的基本编程方法；
3. 学习了 ARM 的基本编程方法；
4. 学习了 OpenMP 的基本方法；
5. 学习了自定义多线程的基本编程方法；
6. 学习了 OpenCL 的基本编程方法。

通过本项目的实施，对矩阵乘法的优化方法有了相对系统性的认识，这些方法也可以被今后遇到的其它算法所借鉴，这些优化方法包括：

1. 内存访问优化
2. 多线程支持
3. 自动向量化
4. 算法优化

今后可以借鉴以上的方法来帮助改进其它算法。

通过本项目的实施，利用 GPU 进行了矩阵乘法计算，虽然计算速度并不理想，但为今后利用 GPU 进行计算准备了一些基础知识。

总体来说，更多地是从广度上学习了多种优化方法，开阔了视野；而从深度上来说，每种优化方法运用地还是很粗糙，一方面时间来不及，一方面理解的程度不够。今后需要更加精细地深入学习这些优化方法。

第11章 附录 1 项目知识要点备忘清单

No	要点	说明	项目
1	OpenBlas	基础线性代数程序集	3
2	SIMD	多数据流处理技术,能够同时对多个数据进行操作，intel avx2 指令集	3
3	ARM SIMD	neon 选项	3
4	OpenMP	共享存储并行编程	3
5	Thread	多线程编程技术	3
6	OpenCL	运用 GPU 进行计算	3
7	objdump	将执行文件反汇编成汇编程序和 gcc -S -mavx matrixMul_SIMD.c 作用类似但更简单，因为 gcc 反汇编可能要参数	3
8	函数指针	用于简化 benchmark 代码	3

第12章 附录 2 GPU 的 ubuntu 驱动安装方法

12.1 目的

因为要用 GPU 进行矩阵计算首先需要安装支持 OpenCL 的显卡驱动，其次才是安装 OpenCL，所以本章记录 ubuntu 操作系统上常见的支持 OpenCL 的显卡驱动安装方法以备后查。

12.2 Intel 集成显卡

1 安装驱动（支持 OpenCL3.0）

```
sudo apt-get install -y intel-opencl-icd intel-level-zero-gpu level-zero intel-media-va-driver-non-free libmfx1 libmfxgen1 libvpl2 libegl-mesa0 libegl1-mesa libegl1-mesa-dev libgbm1 libgl1-mesa-dev libgl1-mesa-dri libglapi-mesa libgles2-mesa-dev libglx-mesa0 libigdgmm12 libxatracker2 mesa-va-drivers mesa-vdpau-drivers mesa-vulkan-drivers va-driver-all
```

2 安装 OpenCL

参考附录 3 OpenCL 使用方法

3 检查显卡对 openCL 支持情况

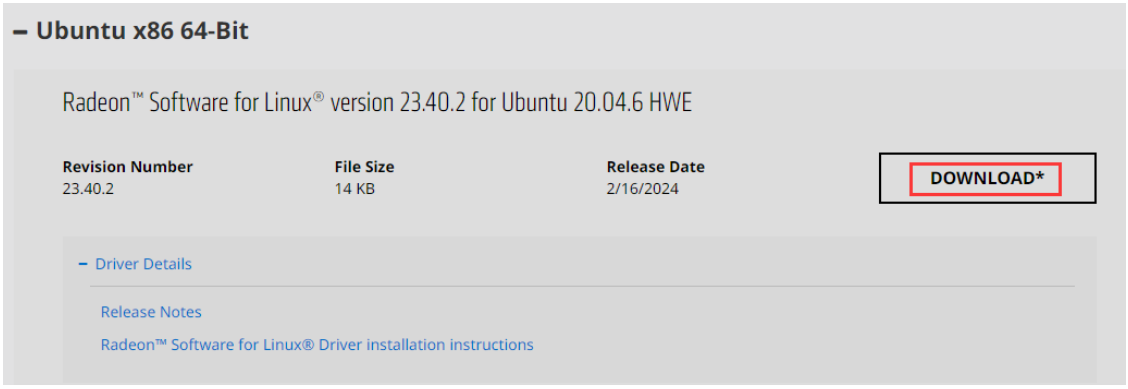
```
clinfo
```

```
Number of platforms                               1
Platform Name                                     Intel(R) OpenCL HD Graphics
Platform Vendor                                   Intel(R) Corporation
Platform Version                                  OpenCL 3.0
Platform Profile                                  FULL_PROFILE
```

12.3 AMD GPU

1 下载 AMD GPU 驱动（ubuntu 版本）

```
https://www.amd.com/en/support/linux-drivers
```



2 安装 AMD GPU 驱动

```
sudo apt install ./amdgpu-install_5.4.50403-1_all.deb
amdgpu-install
```

3 安装 Mesa3D 图形库工具

```
sudo apt install mesa-utils
```

4 检查 AMD GPU 驱动是否成功安装

```
glxinfo | grep rendering
direct rendering: Yes
```

5 AMD opencl SDK 下载安装

```
wget
https://github.com/ghostlander/AMD-APP-SDK/releases/download/v2.9.1/AMD-APP-SDK-v2.9.1-lnx64.tar.xz
tar xvJf AMD-APP-SDK-v2.9.1-lnx64.tar.xz
cd AMD-APP-SDK-v2.9.1-lnx64
sudo ./install.sh
Installing to /opt/AMDAPPSDK-2.9-1
Exported LD_LIBRARY_PATH=/opt/AMDAPPSDK-2.9-1/lib/x86_64/ via
/etc/profile.d/AMDAPPSDK.sh
```

6 下载安装 OpenCL

```
参考附录 3 OpenCL 使用方法
```

7 检查显卡对 openCL 支持情况

clinfo

```
Number of platforms:                         1
  Platform Profile:                          FULL_PROFILE
  Platform Version:                          OpenCL 1.2 AMD-APP (1445.5)
  Platform Name:                             AMD Accelerated Parallel
Processing
  Platform Vendor:                           Advanced Micro Devices, Inc.
  Platform Extensions:                        cl_khr_icd
cl_amd_event_callback cl_amd_offline_devices cl_amd_hsa
  Platform Name:                             AMD Accelerated Parallel
Processing
Number of devices:                           1
  Device Type:                               CL_DEVICE_TYPE_CPU
  Vendor ID:                                 1002h
  Board name:
  Max compute units:                          12
  Max work items dimensions:                   3
    Max work items[0]:                        1024
    Max work items[1]:                        1024
    Max work items[2]:                        1024
Max work group size:                         1024
```

8 重启计算机

9 Sample

/opt/AMDAPPSDK-2.9-1/samples/opencl/bin/x86_64 下有 sample，可以运行

12.4 NVIDIA GPU

虽然我的笔记本电脑并没有装 NIVIDIA GPU 的独立显卡，但由于 NVIDIA 芯片是当前最流行、最重要的 AI 芯片，因此依然把它的安装使用方法和步骤记录如下以备查。

1 安装显卡驱动

sudo apt install nvidia-driver-XXX

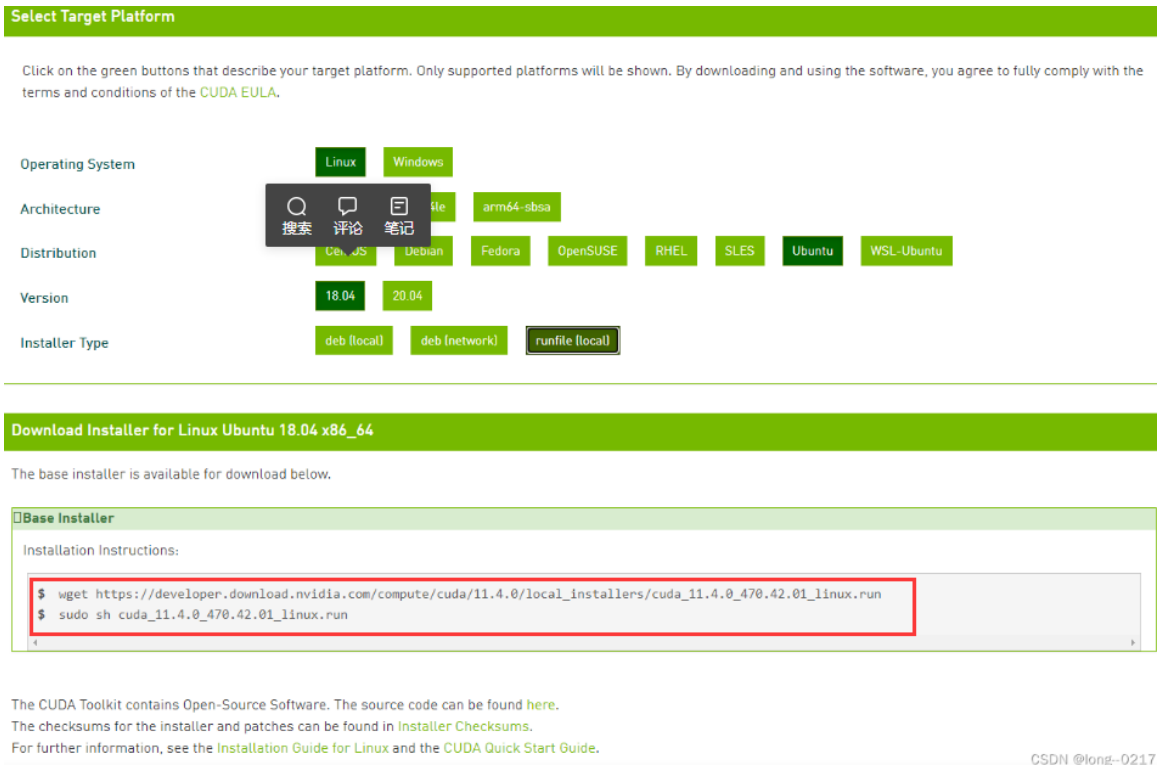
2 重启系统

3 查看显卡信息

```
nvidia-smi
```

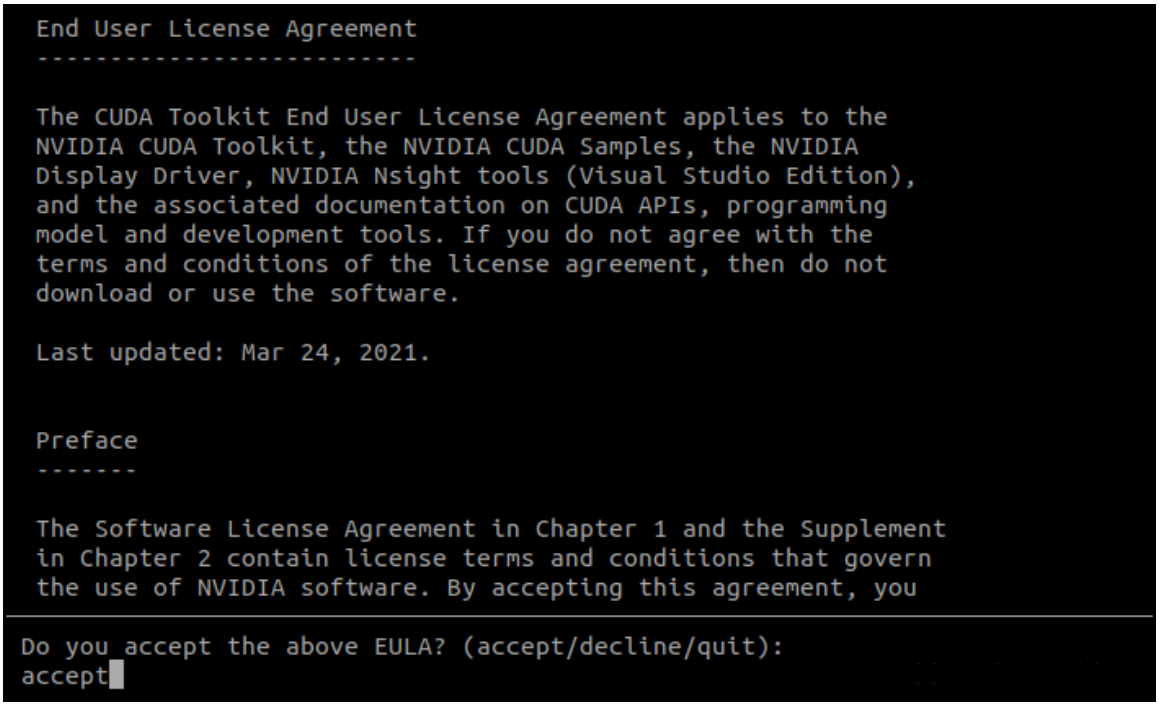
4 安装 CUDA

到 nivida 官网下载自己可安装的版本




```
wget https://developer.download.nvidia.com/compute/cuda/11.4.0/local_installers/cuda_11.4.0_470.42.01_linux.run
```

```
sudo sh cuda_11.4.0_470.42.01_linux.run
```



5 配置环境变量

```
vi ~/.bashrc
export PATH=$PATH:/usr/local/cuda/bin
export CUDA_HOME=$CUDA_HOME:/usr/local/cuda
```

注：路径根据实际情况

6 检查是否安装成功

```
nvcc -V
```

7 添加 cuda lib 路径

```
sudo echo '/usr/local/cuda-11.4/lib64/' >> /etc/ld.so.conf
sudo ldconfig
```

8 OpenCL 安装

参考附录 3 OpenCL 使用方法

9 写代码测试安装过程是否成功

第13章 附录 3 OpenCL 使用方法

13.1 OpenCL 安装及环境设置

1 安装

```
sudo apt install opencl-headers
sudo apt install ocl-icd-libopencl1
sudo apt install ocl-icd-opencl-dev
sudo apt install clinfo
```

或者直接使用如下指令：

```
sudo apt-get install opencl-headers ocl-icd-libopencl1 ocl-icd-opencl-dev clinfo
```

2 安装显卡驱动和显卡对应的 OpenCL 驱动

```
参考附录 2
```

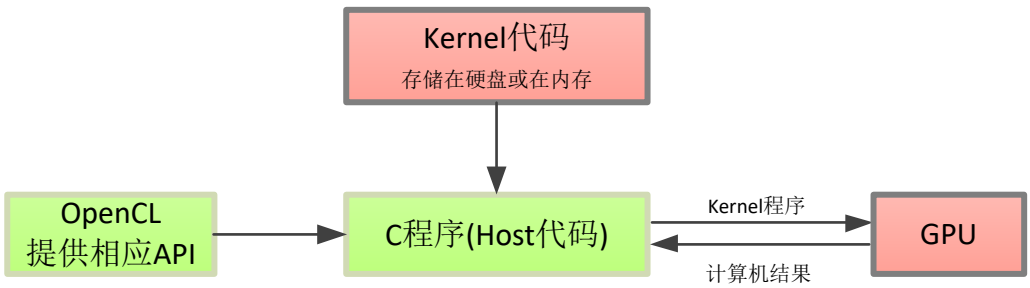
3 执行 clinfo

```
clinfo
```

4 编程验证

13.2 OpenCL 编程框架理解

OpenCL 的编程框架的基本结构可以理解为一句话（虽不准确但比较容易记忆）：C 语言程序将计算任务动态编译成 GPU 机器码并抛给 GPU，GPU 计算后将结果返回给 C 语言程序。如下图：



OpenCL 的编程框架

之所以 C 程序要动态编译 Kernel 代码是因为 Kernel 代码是供 GPU 执行的，GPU 的执行机器码和 CPU 执行机器码肯定是不一样的，所以需要在运行时刻进行编译。而这一点正好是 OpenCL 的主要功能之一。

为了让 C 程序正确而方便地把 Kernel 程序抛给 GPU 进行计算，并能从 GPU 得到计算结果，OpenCL 提供了一系列的数据类型和函数来帮助做到这一点，这些数据类型和函数包含如下功能：

1) 选择 OpenCL 平台并创建一个上下文；
2) 创建设备并创建命令队列；
3) 创建和构建程序对象：这部分就是把 kernel 程序读进来并实时编译成 GPU 能执行的程序；
4) 创建 OpenCL 内核并分配内存空间；
5) 创建内存对象；
6) 设置内核数据并执行内核；
7) 读取执行结果并释放 OpenCL 资源。

综上所述，上述的这些功能总结起来就是访问 GPU 的初始化，将 kernel 程序动态编译并传送给 GPU 执行，并准备好内存接收 GPU 的计算结果。

13.3 Kernel 源代码特点

OpenCL 的 kernel 程序的语法和 C 语言语法的区别：

- 地址空间限定符：OpenCL 引入了地址空间限定符（如__global，__local，__private 等），用于指示变量的存储位置和生命周期。
- 内置函数和类型：OpenCL 提供了一些内置的函数和类型，用于处理向量运算、内存访问和其他与并行计算相关的任务。
- 并行执行模型：OpenCL 的 kernel 程序是为并行执行而设计的，这意味着它们可以同时在多个工作项（work-items）上执行，而标准 C 程序通常按顺序执行。
- 工作组和内存层次结构：OpenCL 引入了工作组（work-groups）的概念，以及一个包括全局内存、局部内存和私有内存的内存层次结构。
- 没有指针运算：OpenCL kernel 中不允许进行指针运算，这是为了避免复杂的数据依赖和并行化问题。

第14章 附录 4 L1、L2、L3 缓存

14.1 概述

L1、L2 和 L3 缓存对矩阵乘法的优化如此重要，因此将其基本知识记录如下。

L1、L2 和 L3 缓存是 CPU 中用于存储临时数据以提高计算机程序性能的内存层次结构。它们各自的作用如下：

- L1 缓存（一级缓存）：

作用：L1 缓存是距离处理器核心最近的缓存层，用于存储最常用的数据和指令。

容量与速度：常见的 L1 缓存大小通常在几 KB 到几十 KB 之间。

分类：L1 缓存进一步分为数据缓存（L1 DCache）和指令缓存（L1 ICache）。数据缓存存储处理器核心正在处理的数据，而指令缓存则存储处理器正在执行的指令。

特点：由于 L1 缓存直接与处理器核心相连，因此其访问速度非常快，对于提高 CPU 的性能至关重要。

- L2 缓存（二级缓存）：

作用：L2 缓存位于 L1 缓存和主内存之间，用于存储更多的数据和指令，以便在 L1 缓存未命中时提供更多的备份。

容量与速度：L2 缓存的容量通常比 L1 大，在数百 KB 到几 MB 之间，速度也较快，但相对于 L1 缓存来说稍慢一些。

特点：L2 缓存的存在使得处理器在访问主内存之前，能够有更多的机会从缓存中获取所需的数据和指令，从而减少了访问主内存的次数，提高了程序的执行效率。

- L3 缓存（三级缓存）：

作用：L3 缓存是位于处理器核心之间的共享缓存，多个处理器核心可以共享相同的 L3 缓存。这有助于减少核心之间的数据传输时延。

容量与速度：L3 缓存的容量通常比 L2 大，在现代高端处理器中，L3 缓存的大小经常达到或超过 8MB，甚至更高，但访问速度相对于 L1 和 L2 缓存来说可能稍慢一些。

功能与特点：

数据存储和访问加速：L3 缓存能够保存最近访问的数据，当处理器需要读取或写入这些数据时，可以直接从 L3 缓存中获取，无需等待慢速主内存的访问，从而大大提高处理器的数据访问速度和整体性能。

减少内存访问的延迟：通过在 L3 缓存中存储频繁使用的数据和指令，处理器可以更快地获取到这些数据，从而减少内存访问的延迟。

提高数据局部性：处理器在执行指令时，往往会连续访问相邻的内存地址，即数据的局部性原理。L3 缓存的存在使得处理器能够更好地利用数据的局部性，连续读取和写入相关的数据，从而提高整体的性能。

改善多核处理器的处理能力：在现代计算机系统中，多核处理器已经成为主流。L3 缓存作为多个处理器核心共享的缓存，有助于协调各核心之间的数据访问，提高多核处理器的整体性能。

14.2 程序如何获得 L1、L2、L3 大小

C 语言在 ubuntu 上获得 L1、L2、L3 缓存的代码段示例（引自 ChatGPT）：

```
    // 遍历所有的 CPU 核心
    while (1) {
        snprintf(line, sizeof(line),
"/sys/devices/system/cpu/cpu%d/cache/index*/size", cpu_num);
        file = fopen(line, "r");
        if (file == NULL) {
            break; // 没有更多的 CPU 核心或缓存信息
        }

        while (fgets(line, sizeof(line), file)) {
            if (sscanf(line, "cache size\t: %9s %19s", cache_type, cache_size)
== 2) {
                unsigned long size = strtoul(cache_size, NULL, 16); // 将十六
进制字符串转换为无符号长整数
                if (strcmp(cache_type, "L1") == 0) {
                    l1_size += size; // 累加 L1 缓存大小
                } else if (strcmp(cache_type, "L2") == 0) {
                    l2_size += size; // 累加 L2 缓存大小
                } else if (strcmp(cache_type, "L3") == 0) {
                    l3_size += size; // 累加 L3 缓存大小
                }
            }
        }
```

```
        fclose(file);
        cpu_num++;
}
```

*备注：测试机器的 L1、L2、L3 如下：

线程数：20（即逻辑处理器数为 20）

L1：1.2MB

L2：11.5MB

L3：24.0MB

第15章 附录 5 Wsl1 升级至 Wsl2 方法

由于我的电脑一开始安装的是 wsl1 版本，但是 opencl 需要用到 wsl2 版本的功能，因此在这里记录从 wsl1 到 wsl2 的升级过程。

- Powershell 指令

```
wsl -l -v
dism.exe /online /enable-feature
/featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all
/norestart
wsl --set-default-version 2
wsl -install
wsl --unregister Ubuntu-20.04
```

- Ubuntu 确认指令

```
cat /etc/os-release
```

- Ubuntu 下工具安装

```
sudo apt install cmake
cmake ..
sudo apt install g++
cmake ..
history
sudo apt install libopenblas-dev
```