**Simple Matrix Multiplication**

项目编号：project2

|  |  |
| --- | --- |
| **学　　号：** | **12210158** |
| **姓 名：** | **吴同馨** |
| **指导老师：** | **于仕琪** |
| **完成时间：** | **2024-03-30** |

**目 录**

[第1章 项目要求分析 1](#_Toc162538591)

[1.1 概述 1](#_Toc162538592)

[1.2 项目要求依据 1](#_Toc162538593)

[1.3 开发语言要求 1](#_Toc162538594)

[1.4 实现功能要求 1](#_Toc162538595)

[1.5 项目提交内容 1](#_Toc162538596)

[1.6 项目提交时间 1](#_Toc162538597)

[第2章 项目要点分析 2](#_Toc162538598)

[2.1 概述 2](#_Toc162538599)

[2.2 C/C++矩阵内存分配 2](#_Toc162538600)

[2.3 矩阵乘法朴素算法 3](#_Toc162538601)

[2.4 C/C++和Java的矩阵 3](#_Toc162538602)

[2.5 性能测试工具或方法 4](#_Toc162538603)

[2.6 其它工具的使用 4](#_Toc162538604)

[第3章 性能测试约定 5](#_Toc162538605)

[第4章 开发工具 6](#_Toc162538606)

[4.1 C/C++工具 6](#_Toc162538607)

[4.2 Java工具 6](#_Toc162538608)

[第5章 程序结构与项目构建 7](#_Toc162538609)

[5.1 程序结构 7](#_Toc162538610)

[5.2 C/C++目录结构 8](#_Toc162538611)

[5.3 C/C++ CMakeFileLists.txt文件 8](#_Toc162538612)

[5.4 Java包结构 10](#_Toc162538613)

[5.5 Maven引入JMH依赖 10](#_Toc162538614)

[第6章 源代码及阐述 11](#_Toc162538615)

[6.1 C/C++源代码 11](#_Toc162538616)

[6.2 Java源代码 14](#_Toc162538617)

[第7章 进程监视和内存泄漏 17](#_Toc162538618)

[7.1 矩阵乘法进程监视 17](#_Toc162538619)

[7.2 C/C++内存泄漏测试 18](#_Toc162538620)

[第8章 测时结果和分析 22](#_Toc162538621)

[8.1 矩阵乘法原始测试数据 22](#_Toc162538622)

[8.2 矩阵乘法耗时平均值 24](#_Toc162538623)

[8.3 -O3和非-O3对比 25](#_Toc162538624)

[8.4 C/C++（-O3）与Java的对比 25](#_Toc162538625)

[8.5 GoogleBenchmark和JMH对比 26](#_Toc162538626)

[8.6 Java的运行机制 27](#_Toc162538627)

[8.7 编译器 27](#_Toc162538628)

[8.8 运行时刻 27](#_Toc162538629)

[8.9 内存管理 28](#_Toc162538630)

[8.10 虚函数 28](#_Toc162538631)

[8.11 Javac优化措施 28](#_Toc162538632)

[8.12 JIT优化措施 29](#_Toc162538633)

[第9章 项目总结 30](#_Toc162538634)

[第10章 附录1 Valgrind使用方法 31](#_Toc162538635)

[第11章 附录2 cmake使用方法 33](#_Toc162538636)

[第12章 附录3 Benchmark使用方法 35](#_Toc162538637)

[第13章 附录4 Maven与JMH使用方法 37](#_Toc162538638)

[13.1 Maven安装与设置 37](#_Toc162538639)

[13.2 Maven中引入JMH 38](#_Toc162538640)

[第14章 附录5 JMH使用方法 40](#_Toc162538641)

[14.1 概述 40](#_Toc162538642)

[14.2 JMH注解说明 40](#_Toc162538643)

# 项目要求分析

## 概述

根据项目作业要求“Project2.pdf”文件，本项目的要求是用C/C++和Java分别实现矩阵乘法，重点是比较和分析C/C++的矩阵乘法和Java的矩阵乘法在计算时间及其它性能上的不同。

## 项目要求依据

project2.pdf文档。

## 开发语言要求

C/C++和Java。

## 实现功能要求

1. 用C/C++和Java分别实现矩阵乘法；
2. 分别对C/C++和Java的矩阵乘法计量其计算时间随着矩阵大小变化的变化；
3. 比较二者的性能并分析与解释其中原因；
4. 比较并分析二者其它方面的性能不同。

## 项目提交内容

1. 源代码: .c、.java文件格式

2. report: pdf文件格式

## 项目提交时间

截止时间：2024年3月31日23:59

# 项目要点分析

## 概述

通过实现本项目的目标，查阅各种资料，学习了C/C++的指针、双指针、函数指针的应用，同时学习了性能测试有关工具和手段。作为初学者更多关注的是基础知识和这些基础知识的细节，并且尽量对这些细节做好记录（包括记录在以后的各个章节），以便于今后能够方便查阅。

对于本项目，由于是比较矩阵乘法的C/C++程序和Java程序的性能，而不是对比矩阵乘法的不同算法之间的性能，因此在算法上采用朴素算法，将项目的学习重心投注于C/C++的基础知识和相关工具的学习上面。

本项目学习与开发的要点如下：

* C/C++中矩阵的内存分配运用双指针；
* C/C++中函数指针运用；
* 性能测试采用内联测试和外部工具测试；
* 各种测试工具：

内存泄漏测试工具

C/C++Benchmark测试

Java的benchmark测试

* 为了使用上述测试工具，还需要熟悉：

C/C++构建工具cmake

Java的构建工具maven

## C/C++矩阵内存分配

在上述C/C++的矩阵结构体中，采用双指针表达矩阵的二维数组形式。双指针使用时的主要问题在于分配内存时候容易出错，错误如下：

matrix.values = (float\*\*)malloc(rows\*cols\*sizeof(float));

实际分配内存的大小应该如下：

matrix.values = (float\*\*)malloc( rows \* sizeof(float\*) +

rows \* cols \* sizeof(float));

也就是说双指针的内存由2部分构成：存放矩阵行地址的内存+矩阵值的内存，这两部分内存的性质不一样：存地址的内存是每个地址8个字节(64位系统)，共有rows个；而存float数值的内存每个数值为4字节，共有rows \* cols个。

## 矩阵乘法朴素算法

由于本项目要求的是对比C/C++和Java程序之间的性能，两种程序的算法应保持一致，即相同算法下C/C++和Java程序之间的性能对比，不同的算法之间的对比不是主要考虑的对象。在这里，我们选择最简单的矩阵乘法的朴素算法，不考虑更为复杂优化的算法。

虽然没有采用一些优化的算法，但是在这里仍然记录一下查阅、学习过矩阵乘法的算法优化的情况。

1. 朴素算法，即直接按照矩阵乘法的定义进行计算。其时间复杂度为O(n^3)，其中n是矩阵的维度。对于较大的矩阵，这种方法的计算量会非常大。
2. Strassen算法：1969年，德国数学家Volker Strassen提出了Strassen算法，这是矩阵乘法算法优化历史上的一个重要里程碑，第一次把时间复杂度降到O(n^3)以下---这种算法的时间复杂度降低到了O(n^log7)，约等于O(n^2.807)。Strassen算法让人们意识到矩阵乘法问题有可能存在更低的维度解法，即可能存在更大的优化空间。
3. 后续优化与发展：在Strassen算法之后，更高效的矩阵乘法算法大多基于分治策略，即将大矩阵拆分成小矩阵，然后利用递归和合并的思想进行计算。如Coppersmith–Winograd算法，其时间复杂度为O(n^2.375)。

## C/C++和Java的矩阵

1. C/C++矩阵结构体

因为矩阵是一个二维数组形式，因此采用双指针方式来动态分配这个二维数组。具体采用用下的结构体形式：

typedef struct Matrix{

    int rows;

    int cols;

    float \*\* values;

    struct Matrix\* (\*matrix\_mul)(struct Matrix\* A,struct  Matrix\* B);

} Matrix;

1. Java的矩阵类

public class Matrix {  
 int rows;  
 int cols;  
 float[][] values;  
 public Matrix(){}  
 public Matrix(int rows,int cols) {}  
 public Matrix(float[][] values){}  
 public Matrix matrix\_mul(Matrix A,Matrix B){}  
 public void matrix\_print() {}

## 性能测试工具或方法

1. 程序中内联手动计时；
2. C/C++：google Benchmark
3. Java：JMH

具体的安装使用方法在附录中记录。

## 其它工具的使用

由于编译要用到的参数很多，从命令行输入几乎不可能，因此采用项目构建工具。

C/C++的构建工具采用：Cmake

Java的构建工具采用：maven

# 性能测试约定

|  |  |  |
| --- | --- | --- |
| No | 约定 | 内容 |
|  | 矩阵类型 | 正方形 |
|  | 矩阵大小 | 10，20，50，100，200，500，1000，1200，1500，2000 |
|  | 矩阵值 | 0.0-0.1 |
|  | 测试次数 | 6 |
|  | 单次测试时间单位 | 纳秒 |
|  | 测时起点 | 矩阵乘法前 |
|  | 测时终点 | 矩阵乘法后 |
|  | 机器 | 同一台机器 |
|  | 单次测试记录 | 矩阵大小、轮次、用时 |

# 开发工具

根据上一章分析，采用如下的开发环境和工具。具体这些工具软件的使用方法记录在附录中供今后备查。

## C/C++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 | 附录2 |
| 内存泄漏测试工具 | valgrind | 附录1 |
| 性能测试工具 | google benchmark | 附录3 |

## Java工具

|  |  |  |
| --- | --- | --- |
| 项目 | 内容 | 参照 |
| 操作系统 | Windows11 | - |
| 开发语言 | Java | - |
| 集成开发工具 | idea | - |
| 项目构建工具 | maven | 附录4 |
| 性能测试工具 | JMH | 附录5 |

# 程序结构与项目构建

## 程序结构



程序结构图

本项目中，矩阵作为一个对象，被置于中心位置，根据需要将自定义的benchmark和第三方benchmark套件分别挂在程序架构上对矩阵对象进行benchmark测试，互相之间没有干扰。

构建工具作为项目构建的基础支撑，将矩阵对象和自定义benchmark和benchmark套件构建在一起，使得编译、打包等变得简单易操作，并且加强了项目的可扩展性。

## C/C++目录结构

|---01c <!-- 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.c       <!-- 矩阵乘法实现 -->

|---matrixMul.h       <!-- 矩阵相关定义 -->

|---test

|---CMakeLists.txt

|---bentest.cpp       <!-- googlebenchmark测试程序 -->

|---main.c            <!-- 自定义测试程序 -->

## C/C++ CMakeFileLists.txt文件

* 根目录CMakeLists.txt

cmake\_minimum\_required(VERSION 3.6)

project(mm)

**# 添加编译选项 -O3 ，此处可以添加其它编译优化选项**

add\_compile\_options(-O3)

**# 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，供发布使用**

set(CMAKE\_BUILD\_TYPE Release)

include\_directories(

    ${CMAKE\_CURRENT\_SOURCE\_DIR}

)

**#添加2个子目录**

add\_subdirectory(src)

add\_subdirectory(test)

* src子目录CMakeLists.txt

**# 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 matrixMul.c )

set(SRC\_H matrixMul.h)

**# 设置库的名字**

add\_library(mm STATIC ${SRC} ${SRC\_H})

* test子目录CMakeLists

**# 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)

**# 添加源文件列表，并编译为测试程序**

set(SRC1  main.c  )

set(SRC2  bentest.cpp  )

**# 查找benchmark包**

find\_package(benchmark REQUIRED)

**# 设置测试程序的名字**

add\_executable(main ${SRC1})

**# 设置测试程序的名字**

add\_executable(myBench ${SRC2})

**# 链接之前生成的库**

target\_link\_libraries(main mm)

target\_link\_libraries(myBench mm)

target\_link\_libraries(myBench benchmark::benchmark)

**# 设置执行文件的输出路径**

set(OUTPUT\_DIR "${CMAKE\_CURRENT\_SOURCE\_DIR}/../bin")

**# 确保输出目录存在**

file(MAKE\_DIRECTORY ${OUTPUT\_DIR})

**# 添加一个自定义目标来复制可执行文件到指定目录**

add\_custom\_target(

    copy\_exe ALL

    COMMAND ${CMAKE\_COMMAND} -E copy $<TARGET\_FILE:main> ${OUTPUT\_DIR}/

    COMMAND ${CMAKE\_COMMAND} -E copy $<TARGET\_FILE:myBench> ${OUTPUT\_DIR}/

    COMMENT "txwu copying executable to ${OUTPUT\_DIR}"

)

## Java包结构

用maven构建：

|---02Java                             <!-- java程序 -->

|---pj2

|---src

|---main

|---java

|---com.pj2.matrix

|--- Matrix         <!-- 矩阵类含矩阵乘法 -->

|---test

|---java

|---com.pj2.benchmarks

|--- MyBenchmark <!-- JMH测试程序 -->

|---  Main                        <!-- 自定义测试程序 -->

## Maven引入JMH依赖

<!-- 添加JMH依赖 -->  
<dependency>  
 <groupId>org.openjdk.jmh</groupId>  
 <artifactId>jmh-core</artifactId>  
 <version>1.37</version>  
</dependency>  
  
<dependency>  
 <groupId>org.openjdk.jmh</groupId>  
 <artifactId>jmh-generator-annprocess</artifactId>  
 <version>1.37</version>  
</dependency>

# 源代码及阐述

## C/C++源代码

### 矩阵结构与矩阵乘法

矩阵结构

#ifndef \_MM\_H\_

#define \_MM\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#ifdef \_\_cplusplus

extern "C" {

#endif

**typedef** struct Matrix{

int rows;

int cols;

float \*\* values;

struct Matrix \* (\*f\_mul)(struct Matrix \* A,struct Matrix \* B);

}Matrix;

//为矩阵分配内存

Matrix \* matrix\_malloc(const int rows, const int cols);

// 释放矩阵内存

void matrix\_free(Matrix \* matrix);

Matrix \* matrix\_mul(Matrix \* A,Matrix \* B);

#ifdef \_\_cplusplus

}

#endif

#endif

矩阵内存分配与释放

//为矩阵分配内存

Matrix \* matrix\_malloc(const int rows, const int cols){

Matrix \* matrix = (Matrix\*)malloc(**sizeof**(Matrix));

matrix->rows = rows;

matrix->cols = cols;

// 分配一个指向float指针的数组

matrix->values = (float \*\*)malloc(matrix->rows \* **sizeof**(float \*));

**if** (matrix->values == **NULL**) {

perror("memory allocation failed for row");

**return** **NULL**;

}

// 为每一个指针分配一个float数组

**for** (int i = 0; i < matrix->rows; ++i) {

matrix->values[i] = (float \*)malloc(matrix->cols \* **sizeof**(float));

**if** (matrix->values[i] == **NULL**) {

perror("memory allocation failed for col");

// 某元素内存分配失败，需要释放之前已经分配的内存

**while** (i > 0) {

free(matrix->values[--i]);

}

free(matrix->values);

**return** **NULL**;

}

}

**return** matrix;

}

// 释放矩阵内存

void matrix\_free(Matrix \* matrix){

**for** (int i = 0; i < matrix->rows; ++i) {

free(matrix->values[i]);

}

free(matrix->values);

free(matrix);

}

矩阵乘法

Matrix \* matrix\_mul(Matrix \* A,Matrix \* B){

**if** (A->cols != B->rows){

**return** **NULL**;

}

Matrix \* C = matrix\_malloc(A->rows,B->cols);

**for** (int i = 0; i < A->rows; i++){

**for** (int j = 0; j < B->cols; j++){

C->values[i][j] = 0;

**for** (int k = 0; k < A->cols; k++){

C->values[i][j] += A->values[i][k] \* B->values[k][j];

}

}

}

**return** C;

}

### 自定义benchmark

计时

int main(int argc, char \* argv[]){

struct timespec start, end;

long long elapsed\_nanos;

printf("--------------------------------------\n");

printf("size | turn | time\n");

printf("--------------------------------------\n");

// 获取开始时间

**if** (clock\_gettime(CLOCK\_REALTIME, &start) == -1) {

perror("clock\_gettime");

**return** EXIT\_FAILURE;

}

**for**(int i = 0;i < BM\_SIZE\_NUM;i++){

**for**(int j = 0;j < BM\_NUM;j++){

BM\_each\_cal(i,j);

}

}

// 获取结束时间

**if** (clock\_gettime(CLOCK\_REALTIME, &end) == -1) {

perror("clock\_gettime");

**return** EXIT\_FAILURE;

}

// 计算经过的纳秒数

elapsed\_nanos = (end.tv\_sec - start.tv\_sec) \* 1000000000LL + (end.tv\_nsec - start.tv\_nsec);

printf("Total time: %lld ns\n", elapsed\_nanos);

**return** EXIT\_SUCCESS;

}

### Google Benchmark

测试

void BM\_MatrixMultiply(benchmark::State& state) {

int size = state.range(0); // 获取当前测试的尺寸

Matrix\* a = matrix\_malloc(size, size);

Matrix\* b = matrix\_malloc(size, size);

BM\_rand\_value(a);

BM\_rand\_value(b);

**for** (auto \_ : state) {

Matrix\* c = matrix\_mul(a,b);

matrix\_free(c);

}

// 清理

matrix\_free(a);

matrix\_free(b);

}

// 使用BENCHMARK宏定义基准测试

BENCHMARK(BM\_MatrixMultiply)->Arg(10)

->Arg(20)

->Arg(50)

->Arg(100)

->Arg(200)

->Arg(500)

->Arg(1000)

->Arg(1200)

->Arg(1500)

->Arg(2000);

BENCHMARK\_MAIN();

## Java源代码

### 矩阵结构与矩阵乘法

public Matrix matrix\_mul(Matrix A,Matrix B){

Matrix C = **new** Matrix(A.rows,B.cols);

**if**(A.cols != B.rows){

System.out.println("The matrix can‘t multiple.");

**return** **null**;

}

**for** (int i = 0;i < A.rows;i++){

**for** (int j = 0;j < B.cols;j++){

C.values[i][j] = 0;

**for**(int k = 0; k < A.cols; k++){

C.values[i][j] += A.values[i][k] \* B.values[k][j];

}

}

}

**return** C;

}

### 自定义benchmark

计时

public static void main(String[] args) {

long startTime = System.nanoTime();

**for**(int t = 0;t < BM\_SIZE\_NUM;t++) {

**for** (int u = 0;u < BM\_NUM;u++){

BM\_cal(t,u);

}

}

long endTime = System.nanoTime();

// 计算执行时间（纳秒）

long durationInNanos = endTime - startTime;

System.out.println("Total time: " + durationInNanos);

}

### JMH

关于JMH注解见附件5

@Warmup(iterations = 3,time = 1,timeUnit = TimeUnit.SECONDS)

@Measurement(iterations = 1, time = 1, timeUnit = TimeUnit.SECONDS)

@State(Scope.Benchmark)

@BenchmarkMode(Mode.AverageTime)

@OutputTimeUnit(TimeUnit.NANOSECONDS)

public class MyBenchmark {

@Param({"10","20", "50","100", "200", "500","1000","1200","1500","1800"}) // 定义矩阵大小

public String matrixSize;

Matrix A;

Matrix B;

@Setup

public void setup() {

Random random;

random = **new** Random(5); // 使用固定的种子以确保每次测试的数据相同

int size = Integer.parseInt(matrixSize);

float [][]a = **new** float[size][size];

float [][]b = **new** float[size][size];

// 填充矩阵A和B

**for** (int i = 0; i < size; i++) {

**for** (int j = 0; j < size; j++) {

a[i][j] = random.nextFloat();

}

}

**for** (int i = 0; i < size; i++) {

**for** (int j = 0; j < size; j++) {

b[i][j] = random.nextFloat();

}

}

A = **new** Matrix(a);

B = **new** Matrix(b);

}

//@Threads(2)

//@Fork(2)

@Benchmark

public void BM\_test() {

Matrix m = **new** Matrix();

m.matrix\_mul(A, B);

}

public static void main(String[] args) **throws** Exception {

//org.openjdk.jmh.Main.main(args);

Options opt = **new** OptionsBuilder()

.include(MyBenchmark.class.getSimpleName())

//.resultFormat(ResultFormatType.JSON)

//.result("/D:/result.json")

.forks(1)

.build();

**new** Runner(opt).run();

}

}

# 进程监视和内存泄漏

## 矩阵乘法进程监视

在测试程序的过程中，不管是由于程序的逻辑问题导致死循环，还是矩阵过大导致计算过慢引起程序一时无法退出，都需要监视当前进程状态。用top监视进程状态。

为了把某个瞬间进程监控的实时屏幕信息记录到文件，采用如下指令：

top -b -n 1 > top\_output.txt

以下是该指令的结果：

top - 10:19:54 up 2:03, 0 users, load average: 1.00, 0.73, 0.35

Tasks: 13 total, 2 running, 11 sleeping, 0 stopped, 0 zombie

%Cpu(s): 8.8 us, 0.0 sy, 0.0 ni, 91.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

MiB Mem : 3845.7 total, 3192.1 free, 507.9 used, 145.6 buff/cache

MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 3151.6 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

324 txwu 20 0 400152 397796 1260 R 100.0 10.1 6:19.37 main

1 root 20 0 1824 1180 1108 S 0.0 0.0 0:00.04 init

11 root 20 0 2172 372 0 S 0.0 0.0 0:00.00 init

12 root 20 0 2180 372 0 S 0.0 0.0 0:03.23 init

13 txwu 20 0 6172 5180 3392 S 0.0 0.1 0:00.16 bash

49 root 20 0 2172 372 0 S 0.0 0.0 0:00.00 init

50 root 20 0 2180 372 0 S 0.0 0.0 0:00.21 init

51 txwu 20 0 6208 5220 3440 S 0.0 0.1 0:00.09 bash

85 root 20 0 2172 372 0 S 0.0 0.0 0:00.00 init

86 root 20 0 2180 372 0 S 0.0 0.0 0:00.72 init

87 txwu 20 0 6208 5184 3408 S 0.0 0.1 0:00.04 bash

101 txwu 20 0 18740 10472 6532 S 0.0 0.3 0:01.79 vi

347 txwu 20 0 7788 3140 2776 R 0.0 0.1 0:00.00 top

上述第1行可以看到，main（即矩阵乘法计算）CPU满负荷了，运用排除法查找原因，发现是矩阵的尺寸可能设计大了，最大是10000，这么大尺寸按照矩阵乘法的朴素算法是O(n3)，因此非常耗时。经过多次尝试，当矩阵尺寸超过2000乘法计算时间就非常长了。因此把尺寸的上限定在2000。

## C/C++内存泄漏测试

因为C/C++学习时间不长，在实际编程中真实产生了不少内存泄漏。由于采用了内存错误检测工具valgrind，因此很好地检测出了所有的内存泄漏情况。

以下是valgrind指令、valgrind错误检测报告、程序相应的错误和最终全部正确的检测报告。

* 编译器参数的要求valgrind都会检测到内存错误，但是如果编译器参数设置为-g（命令行gcc –g ） 或者 CMakeLists.txt 内设置成：

set(CMAKE\_BUILD\_TYPE Debug)

valgrind不仅可以检测到内存错误，并且可以精确定位到源代码的内存错误产生的位置。

编译器参数无论是什么，

* 执行指令：

'''bash

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

'''

* 执行上述命令的结果：

01: ==1120== Memcheck, a memory error detector

02: ==1120== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.

03: ==1120== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info

04: ==1120== Command: ./main A.txt B.txt

05: ==1120==

06: Elapsed time: 26005600 nanoseconds

07: ==1120==

08: ==1120== HEAP SUMMARY:

09: ==1120==     in use at exit: 472 bytes in 1 blocks

10: ==1120==   total heap usage: 9 allocs, 8 frees, 8,768 bytes allocated

11: ==1120==

12: ==1120== 472 bytes in 1 blocks are still reachable in loss record 1 of 1

13: ==1120==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload\_memcheck-amd64-linux.so)

14: ==1120==    by 0x48E064D: \_\_fopen\_internal (iofopen.c:65)

15: ==1120==    by 0x48E064D: fopen@@GLIBC\_2.2.5 (iofopen.c:86)

16: ==1120==    by 0x109520: matrix\_fread (matrixMul.c:42)

17: ==1120==    by 0x1091BB: main (main.c:23)

18: ==1120==

19: ==1120== LEAK SUMMARY:

20: ==1120==    definitely lost: 0 bytes in 0 blocks

21: ==1120==    indirectly lost: 0 bytes in 0 blocks

22: ==1120==      possibly lost: 0 bytes in 0 blocks

23: ==1120==    still reachable: 472 bytes in 1 blocks

24: ==1120==         suppressed: 0 bytes in 0 blocks

25: ==1120==

26: ==1120== For lists of detected and suppressed errors, rerun with: -s

27: ==1120== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

上述第10行、16行、17行明确指示了错误在源代码中所产生的位置，根据所指示的源代码位置，打开所在的源代码，

* matrixMul.c源代码1：

// 从矩阵文件读入矩阵值

Matrix \* matrix\_fread(const char \* file\_name){

    FILE \* fp = fopen(file\_name,"r");

    if(fp == NULL){

        perror(file\_name);

        return NULL;

    }

    int m;

    int n;

    int r = fscanf(fp, "%d %d", &m, &n);

    Matrix \* matrix = matrix\_malloc(m, n);

    for(int i = 0; i < m; i++){

        for(int j = 0; j < n; j++){

            r = fscanf(fp, "%f", &(matrix->values[i][j]));

        }

    }

    return matrix;

}

第42行文件打开后没有关闭该指针对应的文件，修改为：

// 从矩阵文件读入矩阵值

Matrix \* matrix\_fread(const char \* file\_name){

    FILE \* fp = fopen(file\_name,"r");

    if(fp == NULL){

        perror(file\_name);

        return NULL;

    }

    int m;

    int n;

    int r = fscanf(fp, "%d %d", &m, &n);

    Matrix \* matrix = matrix\_malloc(m, n);

    for(int i = 0; i < m; i++){

        for(int j = 0; j < n; j++){

            r = fscanf(fp, "%f", &(matrix->values[i][j]));

        }

    }

fclose(fp);

    return matrix;

}

* matrixMul.c源代码2：

Matrix \* matrix\_mul(Matrix \* A,Matrix \* B){

    if (A->cols != B->rows) {

        return NULL;

    }

    Matrix \* C = matrix\_malloc(A->rows,B->cols);

    for (int i = 0; i < A->rows; i++) {

        for (int j = 0; j < B->cols; j++) {

            C->values[i][j] = 0;

            for (int k = 0; k < A->cols; k++) {

                C->values[i][j] += A->values[i][k] \* B->values[k][j];

            }

        }

    }

}

本函数没有返回Matrix指针，修改为：

Matrix \* matrix\_mul(Matrix \* A,Matrix \* B){

    if (A->cols != B->rows) {

        return NULL;

    }

    Matrix \* C = matrix\_malloc(A->rows,B->cols);

    for (int i = 0; i < A->rows; i++) {

        for (int j = 0; j < B->cols; j++) {

            C->values[i][j] = 0;

            for (int k = 0; k < A->cols; k++) {

                C->values[i][j] += A->values[i][k] \* B->values[k][j];

            }

        }

    }

return C;

}

* 将修改的程序保存后编译，然后重新执行内存检测，得到内存检测正确的报告：

01: ==1213== Memcheck, a memory error detector

02: ==1213== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.

03: ==1213== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info

04: ==1213== Command: ./main A.txt B.txt

05: ==1213==

06: Elapsed time: 29330500 nanoseconds

07: ==1213==

08: ==1213== HEAP SUMMARY:

09: ==1213==     in use at exit: 0 bytes in 0 blocks

10: ==1213==   total heap usage: 9 allocs, 9 frees, 8,768 bytes allocated

11: ==1213==

12: ==1213== All heap blocks were freed -- no leaks are possible

13: ==1213==

14: ==1213== For lists of detected and suppressed errors, rerun with: -s

15: ==1213== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

从上述报告的10、12、15行可以看出，根据之前检测出来位置进行修改后，程序的内存泄漏问题被解决了。

这是非常方便、高效的内存错误问题的解决方法。

# 测时结果和分析

## 矩阵乘法原始测试数据

单位：纳秒(ns)

| 矩阵 | 轮次 | C运行(非-O3) | C运行（-O3） | Java第1次 | Java第2次 |
| --- | --- | --- | --- | --- | --- |
| 10 | 1 | 3300 | 1000 | 32100 | 1000 |
| 10 | 2 | 2900 | 900 | 28800 | 700 |
| 10 | 3 | 2700 | 800 | 28400 | 800 |
| 10 | 4 | 2900 | 800 | 28100 | 700 |
| 10 | 5 | 2800 | 900 | 27700 | 700 |
| 10 | 6 | 2600 | 900 | 27800 | 700 |
| 20 | 1 | 20600 | 6400 | 213200 | 4200 |
| 20 | 2 | 20600 | 4500 | 211900 | 4100 |
| 20 | 3 | 20400 | 4400 | 208700 | 4000 |
| 20 | 4 | 22900 | 4400 | 209400 | 4000 |
| 20 | 5 | 20900 | 4400 | 208900 | 4000 |
| 20 | 6 | 18800 | 4400 | 213200 | 4100 |
| 50 | 1 | 288900 | 67600 | 657900 | 57900 |
| 50 | 2 | 301700 | 66100 | 260800 | 61500 |
| 50 | 3 | 296200 | 60700 | 266400 | 62200 |
| 50 | 4 | 289700 | 61200 | 259000 | 57700 |
| 50 | 5 | 298000 | 69200 | 259200 | 58500 |
| 50 | 6 | 296800 | 66400 | 257600 | 58500 |
| 100 | 1 | 2329100 | 479800 | 2136400 | 457900 |
| 100 | 2 | 2334400 | 442900 | 908600 | 474500 |
| 100 | 3 | 2302600 | 457000 | 2021200 | 445000 |
| 100 | 4 | 2307000 | 444400 | 2173400 | 445700 |
| 100 | 5 | 2462600 | 883500 | 523400 | 441500 |
| 100 | 6 | 2379300 | 452000 | 517500 | 450500 |
| 200 | 1 | 19045500 | 3963800 | 4502400 | 3813500 |
| 200 | 2 | 19366000 | 4225200 | 4206200 | 3338100 |
| 200 | 3 | 19970800 | 3591000 | 4303400 | 3380100 |
| 200 | 4 | 20545200 | 3901700 | 4060300 | 3453100 |
| 200 | 5 | 19579500 | 3472900 | 4183300 | 3481800 |
| 200 | 6 | 19767300 | 3339900 | 4091900 | 3381700 |
| 500 | 1 | 328355800 | 62978600 | 104612500 | 92339500 |
| 500 | 2 | 334024800 | 66885800 | 124768900 | 94344500 |
| 500 | 3 | 325719200 | 65974200 | 106587100 | 91493100 |
| 500 | 4 | 326929100 | 67596600 | 98834500 | 92883400 |
| 500 | 5 | 347510500 | 64365000 | 101289900 | 91253200 |
| 500 | 6 | 328553100 | 64646400 | 94797100 | 91434600 |
| 1000 | 1 | 5218242900 | 830009300 | 803961500 | 787436600 |
| 1000 | 2 | 5647301800 | 807881100 | 803701100 | 801236600 |
| 1000 | 3 | 5645007700 | 828987200 | 805662400 | 794015700 |
| 1000 | 4 | 5572996900 | 1132033900 | 792811300 | 796195300 |
| 1000 | 5 | 5703764700 | 1138437300 | 798196600 | 802331600 |
| 1000 | 6 | 5719347900 | 962817000 | 799781600 | 794431800 |
| 1200 | 1 | 10680564000 | 1952137900 | 1495594100 | 1416750900 |
| 1200 | 2 | 10599544000 | 1952286600 | 1488229000 | 1433085500 |
| 1200 | 3 | 10686509700 | 1984744300 | 1503174100 | 1402704700 |
| 1200 | 4 | 10868658900 | 1979381300 | 1511885700 | 1398041300 |
| 1200 | 5 | 10561131100 | 1906955400 | 1499769500 | 1508463500 |
| 1200 | 6 | 10665467700 | 2018603500 | 1591195400 | 1495054500 |
| 1500 | 1 | 19205284600 | 3965905400 | 4153571800 | 5011175900 |
| 1500 | 2 | 20350670500 | 3966878400 | 4803697900 | 3790009100 |
| 1500 | 3 | 19741289800 | 4031446000 | 4864472400 | 4778419300 |
| 1500 | 4 | 19952351800 | 3981619500 | 4868533300 | 3872484200 |
| 1500 | 5 | 20005228700 | 3954087600 | 4807785400 | 4782391900 |
| 1500 | 6 | 20178873100 | 3964993600 | 4900150100 | 5050203600 |
| 2000 | 1 | 56198441900 | 10312819100 | 28900110600 | 27600449000 |
| 2000 | 2 | 56386953900 | 10320833900 | 31199327400 | 27222419900 |
| 2000 | 3 | 55476190500 | 10349043500 | 29120882900 | 31202767500 |
| 2000 | 4 | 55783614100 | 10324782200 | 27265226700 | 28117718500 |
| 2000 | 5 | 56066526100 | 9905742200 | 30876537300 | 30485806100 |
| 2000 | 6 | 55779028500 | 9704276100 | 26687695800 | 30470855700 |

从上表非常醒目地看出Java程序的一个特点：在到矩阵尺寸100、第4轮的时候，Java第1次跑的程序和第2次跑的程序有数量级的差别，但之后二者之间的差别缩小。这应该是反应了Java的2个特点：

* 1 Java程序在第一次调用class文件的时候，JIT需要对其进行动态编译成机器码，这需要消耗很多时间。而之后不需要再次进行动态编译，直接跑已编译出来的机器码，因此后来的速度加快。
* 2 JIT会在运行时刻比如在尺寸100、第四轮不断对Java程序优化，使得程序逐步稳定下来。JIT对Java程序的优化也不是一次完成，而是在运行过程中不断优化，这就造成了第一次运行的前20多轮的计算和第二次运行时候的耗时有很大的差异。这或许就是JIT的热点代码优化。

## 矩阵乘法耗时平均值

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 矩阵/平均用时 | C运行(非-O3) | C运行（-O3） | Java第1次 | Java第2次 |
| 10 | 2867 | 883 | 28817 | 767 |
| 20 | 20700 | 4750 | 210883 | 4067 |
| 50 | 295217 | 65200 | 326817 | 59383 |
| 100 | 2352500 | 526600 | 1380083 | 452517 |
| 200 | 19712383 | 3749083 | 4224583 | 3474717 |
| 500 | 331848750 | 1360883 | 2166283 | 1990700 |
| 1000 | 5584443650 | 950027633 | 800685750 | 795941267 |
| 1200 | 10676979233 | 1965684833 | 1514974633 | 1442350067 |
| 1500 | 19905616417 | 3977488417 | 4733035150 | 4547447333 |
| 2000 | 55948459167 | 10152916167 | 29008296783 | 29183336117 |

从图表上可以看出：

1. C/C++（-O3）的矩阵乘法速度最快，尤其是矩阵越来越大时；
2. Java程序经过JIT动态编译后在1200以下的速度平均快于C/C++，但是到了矩阵较大时候，比C/C++程序慢；
3. C/C++程序如果不经过-O3的编译优化，速度最慢。

## -O3和非-O3对比

|  |  |  |  |
| --- | --- | --- | --- |
| 矩阵/平均用时 | C运行(非-O3) | C运行（-O3） | 百分比差 |
| 10 | 2867 | 883 | 69% |
| 20 | 20700 | 4750 | 77% |
| 50 | 295217 | 65200 | 78% |
| 100 | 2352500 | 526600 | 78% |
| 200 | 19712383 | 3749083 | 81% |
| 500 | 331848750 | 1360883 | 100% |
| 1000 | 5584443650 | 950027633 | 83% |
| 1200 | 10676979233 | 1965684833 | 82% |
| 1500 | 19905616417 | 3977488417 | 80% |
| 2000 | 55948459167 | 10152916167 | 82% |
|  |  | 平均百分差 | 81% |

从上表可以看出用-O3编译，比起不用-O3编译，运行时间缩短了80%。因此-O3编译优化选项对性能的提升是非常大的。

## C/C++（-O3）与Java的对比

选择Java第2次运行的结果进行对比，因为此时class文件已经完全被JIT从字节码动态编译成了机器码。而在运行时刻JIT不会反复动态编译class文件，只会在第1次读到class文件时把字节码编译成机器码。所以一般第1次运行到java的class时速度会比较慢。

|  |  |  |  |
| --- | --- | --- | --- |
| 矩阵/平均用时 | C运行（-O3） | Java第2次 | 相差百分比 |
| 10 | 883 | 767 | -15% |
| 20 | 4750 | 4067 | -17% |
| 50 | 65200 | 59383 | -10% |
| 100 | 526600 | 452517 | -16% |
| 200 | 3749083 | 3474717 | -8% |
| 500 | 1360883 | 1990700 | 32% |
| 1000 | 950027633 | 795941267 | -19% |
| 1200 | 1965684833 | 1442350067 | -36% |
| 1500 | 3977488417 | 4547447333 | 13% |
| 2000 | 10152916167 | 29183336117 | 65% |

从数据来看，矩阵乘法的Java程序耗时和C++比较起来没有明显的区别，只是当矩阵规模到了2000的时候，差距拉开。

## GoogleBenchmark和JMH对比

| 矩阵/平均用时 | googlebenchmark | JMH | 百分比差 |
| --- | --- | --- | --- |
| 10 | 541 | 725 | 25% |
| 20 | 3735 | 4332 | 14% |
| 50 | 82278 | 59433 | -38% |
| 100 | 919395 | 461509 | -99% |
| 200 | 7386855 | 3776593 | -96% |
| 500 | 144998314 | 87328608 | -66% |
| 1000 | 1215909250 | 810213600 | -50% |
| 1200 | 2162033800 | 1426498800 | -52% |
| 1500 | 4071664600 | 7618772200 | 47% |
| 2000 | 9855713100 | 24117537600 | 59% |

GoogleBenchmark的数据和自测的数据差距不大，而JMH的数据也和自测的数据差距不大。由于GoogleBenchmark和JMH在测试方面具有专业性和稳定性，因此，可以以它们作为标准性能测试工具。

从测试数据可以看出，在矩阵规模低于1200时，Java程序的速度要比C/C++快，但是矩阵规模大了以后，Java程序速度比C/C++慢，应该是Java的内存回收机制在内存变大时效率不如C/C++，因为C/C++的内存是直接释放而没有判断的。

总得来说，C/C++和Java在运行速度上的差距没有那么明显，以下从几个方面对比较C/C++和Java的运行机制等方面不同。

## Java的运行机制

在进行C/C++和Java的比较时，不得不首先提到Java虚拟机，因为这是理解这两种语言所谓性能差别的主要原因。Java程序要想运行起来，必须加载到Java虚拟机上。

作为一个平台，Java包括Java虚拟机（JVM）、Java运行时环境（JRE）和Java开发工具包（JDK）。JVM是Java程序运行的核心，它能够将Java字节码转换为特定平台上的机器码并执行。JRE提供了运行Java程序所需的基本类和库，而JDK则包含了开发Java程序所需的编译器、调试器和其他工具。

## 编译器

编译器是C/C++和Java的重要区别所在，也是对性能产生主要影响的原因。

这里C/C++的编译器具体指的的是gcc/g++，Java的编译器是javac和JIT。

C/C++的编译器的编译结果就是机器码，也就是机器直接能执行的代码。

Java的编译器实际上是两部分构成javac和JIT：

1. javac

这个编译器的编译结果是字节码而不是机器码，也就是说，编译出来的执行文件.class文件并不能被计算机直接运行。需要在Java虚拟机上才能运行。

1. JIT

在运行时刻，Java虚拟机会驱动JIT对class文件的字节码进行动态编译成机器码。

## 运行时刻

从上节可以看出，Javac编译出来的程序并不是直接就能运行在机器上，必须在运行时刻经过JIT再次动态编译才能成为机器码运行在机器上。而C/C++编译出来的程序直接运行在机器上，这应该是造成C++程序效率高于Java的主要原因。

## 内存管理

C/C++的内存分配和内存释放是由程序员完成，因此无需额外的判断就可以做到内存的管理。

而Java是自动管理内存，程序员很方便，不用考虑内存管理。但产生的代价就是Java虚拟机必须频繁进行动态检查，如对象实例访问时检查空指针、数组元素访问时检查上下界范围、类型转换时检查继承关系等等。对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但是总体上仍然要消耗着不少的运行时间。

Java语言中的对象内存分配都是堆上进行，只有方法中的局部变量才在栈上分配。而C/C++的对象则有多种内存分配方式，既可能在堆上分配，也可能在栈上分配，如果可以把线程私有的对象在栈上分配，将可以减轻内存回收的压力，也不需要考虑内存屏障方面的问题。

## 虚函数

Java语言中虽然没有virutal关键字，但是使用虚方法的频率却远远大于C/C++语言，这意味着运行时对方法接收者进行多态选择的频率要远远大于C/C++语言， 也意味着JIT编译器在进行一些优化，如方法内联时难度要远大于C/C++的静态优化编译器。Java语言是可以动态扩展的语言，运行时加载新的类可能改变程序类型继承关系，这使得很多全局的优化都难以进行，因为编译器无法看见程序的全貌。

## Javac优化措施

* 常量折叠和优化：如果编译器能够确定某个表达式的值在程序运行期间是恒定不变的，它会将这个表达式替换为相应的常量值。
* 局部变量优化：尽量使用局部变量，因为它们存储在栈中，访问速度较快。而静态变量和实例变量则存储在堆中，访问速度相对较慢。。
* 内联优化：对于短小且频繁调用的方法，编译器可能会进行内联优化，将方法调用替换为方法体，从而减少方法调用的开销。

## JIT优化措施

* 热点探测：识别出频繁执行的代码段（热点代码），并对其进行优化编译。
* 方法内联：对于频繁调用的方法，JIT编译器可能会将其内联到调用点，以减少方法调用的开销。
* 逃逸分析：分析对象是否可能在方法外部被引用，从而决定是否可以将对象在栈上分配，以减少堆内存的使用。

# 项目总结

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

1. 学习了C/C++指针、双指针的运用，矩阵非常适合运用指针；
2. 学习了C/C++编译器的编译选项如-O3等；
3. 学习了内存泄漏的检测方法；
4. 学习了第三方benchmark工具googlebenchmark和Java的JMH；
5. 学习了cmake的CMakelist.txt的编程方法及其相关项目构建；
6. 学习了maven项目构建，并学习了在maven中引入JMH的方法，虽然JMH是Java的benchmark工具，但是对比C/C++下google benchmark，发现二者其实在方法和思想上是一致的；
7. 学习了benchmark测试涉及到的指标；
8. 学习了C语言和Java在性能上面的成因。

由于项目中涉及到工具和知识很多，大部分时间处于查阅相关资料、试验这些工具和知识是否适用和怎么使用上面，难以熟练、深入地掌握它们，更多的是浅尝则止，追求广泛的接触，为今后进一步学习开拓一下知识面。

# 附录1 Valgrind使用方法

1. 安装

sudo apt install valgrind

1. 作用

检查C/C++的内存泄漏。

1. 使用方法

* main.c：int型指针p分配了内存，但是没有释放内存。

#include <stdio.h>

#include <stdlib.h>

int main() {

    int \*p = malloc(sizeof(int));

    \*p = 10;

    printf("%d\n", \*p);

    return 0;

}

* 编译：

gcc -g -o test test.c

采用-g编译选项是为了能够定位到源代码哪一行造成了内存泄漏。

* 执行

valgrind --leak-check=full ./test1 > result.txt 2>&1

注解：采用了重定向将结果输出到result.txt文件中 2>&1把stderr重定向到stdout的重定向位置即result.txt。

valgrind --leak-check=full --show-leak-kinds=all ./test1 > result.txt 2>&1

* result.txt的结果

01: ==1120== Memcheck, a memory error detector

02: ==1120== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.

03: ==1120== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info

04: ==1120== Command: ./main A.txt B.txt

05: ==1120==

06: Elapsed time: 26005600 nanoseconds

07: ==1120==

08: ==1120== HEAP SUMMARY:

09: ==1120==     in use at exit: 472 bytes in 1 blocks

10: ==1120==   total heap usage: 1 allocs, 0 frees, 8,768 bytes allocated

11: ==1120==

12: ==1120== 472 bytes in 1 blocks are still reachable in loss record 1 of 1

13: ==1120==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload\_memcheck-amd64-linux.so)

17: ==1120==    by 0x1091BB: main (main.c:5)

18: ==1120==

19: ==1120== LEAK SUMMARY:

20: ==1120==    definitely lost: 0 bytes in 0 blocks

21: ==1120==    indirectly lost: 0 bytes in 0 blocks

22: ==1120==      possibly lost: 0 bytes in 0 blocks

23: ==1120==    still reachable: 472 bytes in 1 blocks

24: ==1120==         suppressed: 0 bytes in 0 blocks

25: ==1120==

26: ==1120== For lists of detected and suppressed errors, rerun with: -s

27: ==1120== ERROR SUMMARY: 1 errors from 0 contexts (suppressed: 0 from 0)

第10、17行指示了源代码的第5行指针p被malloc但没有被free造成了内存泄漏。

# 附录2 cmake使用方法

1. 下载安装

sudo apt install cmake

1. 主要命令

* add\_compile\_options：设置编译选项
* add\_executable：添加编译结果执行文件
* add\_library：添加编译结果库文件
* target\_link\_libraries：设置连接库

1. 一些C/C++编译器选项

下面是在学习-O3时接触的其它编译器选项，记录在此：

* -Wall：启用几乎所有的警告信息，用于检测常见的编程错误。这个选项可以帮助开发者在代码编译时发现潜在的问题。
* -Wextra：启用额外的警告信息，这些警告通常不是默认启用的，但可以帮助提高代码的质量和安全性。例如，未使用的变量和函数、潜在的缓冲区溢出、不安全的类型转换等都会触发这个选项的警告。
* -pedantic：要求编译器严格遵循ANSI C标准。它会拒绝任何不符合标准的代码，并且会发出警告。这有助于确保代码的可移植性和标准一致性。
* -Wcast-align：当指针类型强制转换导致目标要求的对齐比源更严格时发出警告。这有助于检测可能导致性能下降或运行时错误的指针强制转换。
* -Wcast-qual：检查类型转换时是否丢失了const或volatile限定符。这有助于维护代码的const正确性，防止意外修改不应该被修改的数据。
* -Wdisabled-optimization：当编译器因为某些原因禁用了优化时发出警告。这有助于开发者了解哪些代码可能由于某种原因没有被优化。
* -Wformat=2：检查函数调用中的格式字符串是否与其参数匹配，级别2表示进行更严格的检查。这有助于防止因格式字符串错误导致的运行时错误。
* -Winit-self：检查是否将变量初始化为其自身。这通常是一个错误，应该避免。
* -Wmissing-include-dirs：检查是否包含了必要的头文件目录。这有助于确保编译时能找到所有需要的头文件。
* -Wredundant-decls：检查是否有冗余的声明或定义。这有助于保持代码的清晰和简洁。
* -Wshadow：检查局部变量是否隐藏了外部变量或函数。这有助于防止因变量名冲突导致的错误。
* -Wstrict-overflow=5：在编译时进行更严格的溢出检查，级别5表示进行最严格的检查。这有助于检测可能导致未定义行为的整数溢出。
* -Wundef：在预处理时检查是否使用了未定义的宏。这有助于确保所有使用的宏都已经被正确定义。
* -Wno-unused：禁止显示未使用变量和函数的警告。在某些情况下，可能有些变量或函数是故意留空的，使用这个选项可以避免不必要的警告。
* -Wno-variadic-macros：禁止显示关于可变参数宏的警告。这可以用于避免由可变参数宏引起的某些常见警告。
* -Wno-parentheses：禁止显示关于可能需要括号的表达式的警告。这有时用于避免由于编译器对操作符优先级的理解不同而产生的警告。
* -fdiagnostics-show-option：在诊断信息中显示可以用于禁用该警告或错误的编译器选项。这有助于开发者了解如何控制特定的警告或错误。
* -march=native：根据运行编译器的机器的架构生成代码。这有助于优化代码以充分利用目标机器的特性。
* -flto：启用链接时优化（Link Time Optimization）。这可以在链接阶段进行跨文件的优化，进一步提高代码的性能。
* -Ofast：启用一系列可能违反C/C++标准的优化。这包括假设没有数学错误（如除以零或溢出）的优化，以及忽略某些浮点数的精度要求。使用这个选项可以显著提高代码的运行速度，但也可能引入难以察觉的错误。因此，在使用这个选项时需要特别小心。<!-- -O3的超集，除了启用所有-O3的优化外，还会额外启用一些可能会破坏IEEE浮点标准和其他标准的优化。因此，使用-Ofast时，需要特别小心可能引入的难以察觉的错误。-->

# 附录3 Benchmark使用方法

1. 下载、编译和安装

```bash

# Check out the library.

$ git clone https://github.com/google/benchmark.git

# Go to the library root directory

$ cd benchmark

# Make a build directory to place the build output.

$ cmake -E make\_directory "build"

# Generate build system files with cmake, and download any dependencies.

$ cmake -E chdir "build" cmake -DBENCHMARK\_DOWNLOAD\_DEPENDENCIES=on -DCMAKE\_BUILD\_TYPE=Release ../

# or, starting with CMake 3.13, use a simpler form:

# cmake -DCMAKE\_BUILD\_TYPE=Release -S . -B "build"

# Build the library.

$ cmake --build "build" --config Release

cmake -E chdir "build" ctest --build-config Release

sudo cmake --build "build" --config Release --target install

```

摘自benchmark的readme.md

1. 测试程序样例
2. void BM\_MatrixMultiply(benchmark::State& state) {
3. int size = state.range(0); // 获取当前测试的尺寸
4. Matrix\* a = matrix\_malloc(size, size);
5. Matrix\* b = matrix\_malloc(size, size);
6. BM\_rand\_value(a);
7. BM\_rand\_value(b);
8. **for** (auto \_ : state) {
9. Matrix\* c = matrix\_mul(a,b);
10. matrix\_free(c);
11. }
12. // 清理
13. matrix\_free(a);
14. matrix\_free(b);
15. }
16. // 使用BENCHMARK宏定义基准测试
17. BENCHMARK(BM\_MatrixMultiply)->Arg(10)
18. ->Arg(20)
19. ->Arg(50)
20. ->Arg(100)
21. ->Arg(200)
22. ->Arg(500)
23. ->Arg(1000)
24. ->Arg(1200)
25. ->Arg(1500)
26. ->Arg(2000);
27. BENCHMARK\_MAIN();
28. 编译测试程序

* 采用g++

g++ -o my\_benchmark test.cpp -lbenchmark -lpthread

* 如果用cmake工具，CMakeLists.txt如下设置：

cmake\_minimum\_required(VERSION 3.10)

project(mytest)

# Find Benchmark

find\_package(benchmark REQUIRED)

# Add your source file to the project

add\_executable(mytest test.cpp)

# Link Benchmark library to your test executable

target\_link\_libraries(mytest benchmark::benchmark)

# 附录4 Maven与JMH使用方法

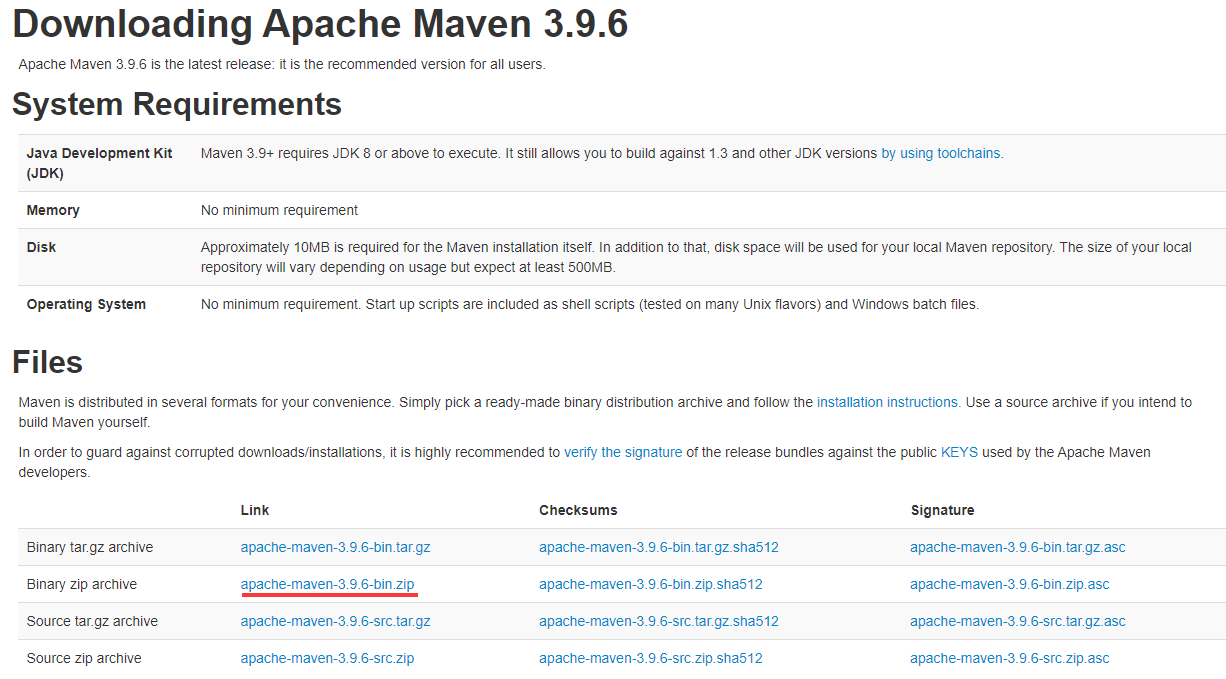
## Maven安装与设置

由于Java采用了Maven作为构建工具，所以把maven安装的说明记录如下。

1. 下载最新maven的版本apache-maven-3.9.6，下载地址：

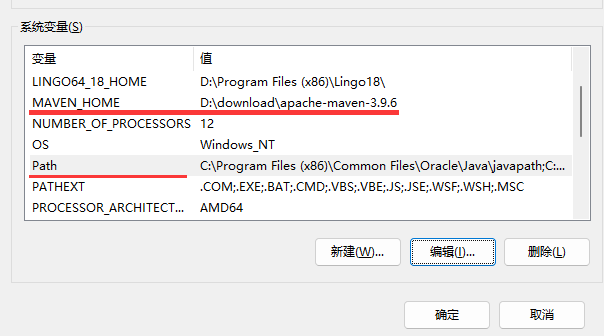
[Maven – Download Apache Maven](https://maven.apache.org/download.cgi)

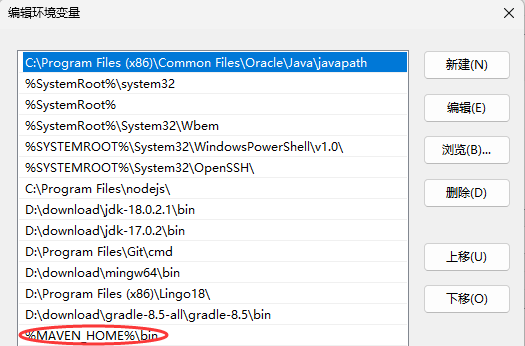
点击红色标注的版本。



将下载包解压到安装路径下，如：D:\download\apache-maven-3.9.6

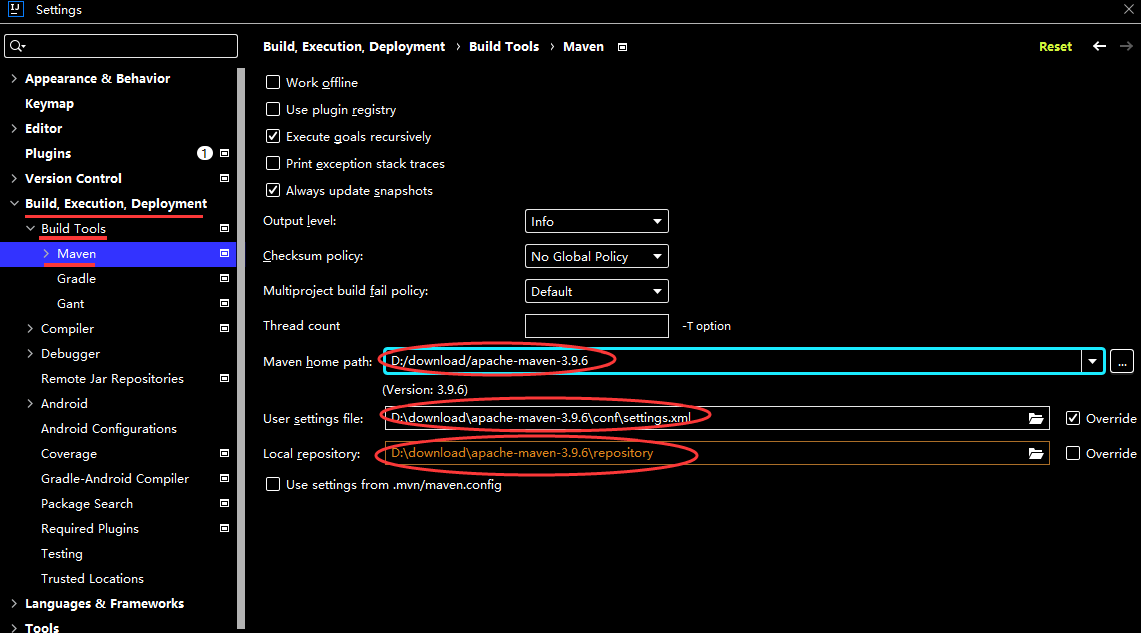
1. 在windows11高级系统设置中设置环境变量：
2. 设置环境变量中的系统变量：





1. 在idea中导入maven

File 🡪 Setting



## Maven中引入JMH

JMH是一个用于构建、运行和分析基于Java及其他JVM语言的基准测试工具，是OpenJDK项目的一部分。以下说明在本项目中使用JMH的过程。

1. 首先建立一个maven项目来管理整个项目；
2. 在Maven的pom.xml中，添加如下依赖：

<dependencies>  
 <!-- 添加JMH依赖 -->  
 <dependency>  
 <groupId>org.openjdk.jmh</groupId>  
 <artifactId>jmh-core</artifactId>  
 <version>1.37</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.openjdk.jmh</groupId>  
 <artifactId>jmh-generator-annprocess</artifactId>  
 <version>1.37</version>  
 </dependency>  
</dependencies>

3 生成测试类，引入JMH的注释，如下图所示：

import org.openjdk.jmh.annotations.\***;**import java.util.concurrent.TimeUnit**;**@BenchmarkMode(Mode.*AverageTime*)  
@OutputTimeUnit(TimeUnit.*NANOSECONDS*)  
@State(Scope.*Thread*)  
public class MyBenchmark {  
 @Benchmark  
 public void testMethod() {  
 // 你的基准测试代码  
 }  
}

# 附录5 JMH使用方法

## 概述

JMH（Java Microbenchmark Harness）是一个用于编写、运行和分析Java微基准测试的框架。它提供了一套注解、工具和API，帮助开发者编写高质量的基准测试代码，并生成准确、可靠的性能数据。

使用JMH进行Java微基准测试的一般步骤如下：

* 编写基准测试代码：使用JMH提供的注解和API编写基准测试类和方法。这些注解和API可以帮助你定义测试的状态、配置预热和测量阶段等。
* 构建和运行基准测试：使用JMH提供的构建和运行工具，将基准测试代码编译成可执行的JAR文件，并运行基准测试。
* 分析测试结果：JMH会生成详细的测试结果报告，包括吞吐量、延迟、标准差等性能指标。你可以使用这些报告来评估代码的性能，并找出可能的优化点。

## JMH注解说明

### @Warmup和@Measurement

@Warmup和@Measurement是类级别注解

@Warmup(iterations = **1,** time = **1,** timeUnit = TimeUnit.*SECONDS*)  
@Measurement(iterations = **1,** time = **1,** timeUnit = TimeUnit.*SECONDS*)  
public class JMHSample\_02\_BenchmarkModes {

@Warmup注解用于配置预热阶段，即基准测试开始前的准备阶段。预热阶段的主要目的是让系统（包括JIT编译器、垃圾回收器、缓存等）达到一个稳定的状态，以便在后续的测量阶段中获得更准确的性能数据。

@Measurement注解则用于配置基准测试的测量阶段。在测量阶段，基准测试方法会被执行多次，并且这些执行的结果会被计入最终的测量数据中。测量阶段的目标是获取基准测试方法的性能数据，包括吞吐量、延迟等。

### @BenchmarkMode

@Benchmark  
@BenchmarkMode(Mode.*SampleTime*)  
@OutputTimeUnit(TimeUnit.*MICROSECONDS*)  
public void measureSamples() throws InterruptedException {

@BenchmarkMode是方法级别注解。

### @OutputTimeUnit

|  |  |
| --- | --- |
| TimeUnit.NANOSECONDS | 纳秒经常用于高性能基准测试中 |
| TimeUnit.MICROSECONDS | 提供更易于阅读和理解的基准测试结果，尤其是在纳秒级别的变化不太显著时。 |

### @State

@State(Scope.*Benchmark*)  
public static class BenchmarkState {  
 volatile double x = Math.*PI***;**}

@State是类级别注解。

| @ State | 测量值 |
| --- | --- |
| Scope.Benchmark | 这是默认的作用域。状态对象对于每个基准测试方法都是唯一的。这意味着每个基准测试方法都会有一个新的状态对象实例。这通常用于存储与单个基准测试方法相关的状态。 |
| Scope.Thread | 状态对象在每个线程中是唯一的。这适用于多线程基准测试，其中每个线程都有自己的状态对象实例。这对于需要线程本地存储的场景非常有用。 |

### @Setup和TearDown

@Setup  
public void prepare() {

x = Math.*PI***;**}

@TearDown  
public void check() {  
 assert x > Math.*PI* : "Nothing changed?"**;**}

@Setup注解用于标记在基准测试方法执行之前应该运行的方法。它可以有多个方法，具体来说，@Setup注解有几个重要的参数：

| @ Setup | 测量值 |
| --- | --- |
| Level.Trial | 在每个试验（trial）之前执行。这是默认级别，如果没有指定其他级别，就会使用这个级别。 |
| Level.Invocation | 在每个方法调用之前执行。这意味着对于基准测试方法的每次迭代，都会先执行带有@Setup(Level.Invocation)注解的方法。 |
| Level.Iteration | 在每个迭代（iteration）之前执行，迭代是包含多个方法调用的序列。 |
| Level. Benchmark | 在整个基准测试之前执行一次。 |

@TearDown注解用于标记在基准测试方法执行之后应该运行的方法。其注解的参数与@Setup相同。

### BlackHole

Blackhole是一个特殊的工具类，用于吸收（即消费但不执行任何操作）基准测试中的结果，从而避免编译器或JIT（Just-In-Time）编译器优化掉这些结果。这确保了基准测试的结果不会被优化器所影响，从而提供了更准确的性能度量。

@Benchmark  
public void measureRight\_2(Blackhole bh) {  
 bh.consume(compute(x1))**;** bh.consume(compute(x2))**;**}

### @Fork

@Fork 注解用于控制基准测试方法执行的次数以及执行的环境。通过使用@Fork注解，您可以指定JMH在多个JVM实例中运行基准测试，这有助于减少JVM预热、JIT编译和其他潜在干扰因素的影响，从而提供更稳定、更准确的性能度量。

@Benchmark  
@Fork(**1**)  
public int measure\_4\_forked\_c1() {  
 return measure(c1)**;**}

使用不同的进程进行测试，可以降低benchmark之间的干扰比如解决JVM优化失效问题。

### @ Param

@Param({"10,10,10"**,** "50,50,50"**,** "100,100,100"}) // 矩阵大小aRows,aCols,bCols  
public String matrixSize**;**

放在参数上方，为基准测试方法定义一组参数值。这样，当你运行基准测试时，JMH 会为每个参数值运行一次测试，并收集结果。

### @Group和@GroupThreads

@Benchmark  
@Group("g")  
@GroupThreads(**3**)  
public int inc() {  
 return counter.incrementAndGet()**;**}

@Group 注解用于将多个基准测试方法组织成一个逻辑组。

@GroupThreads 注解用于指定执行某个基准测试组时使用的线程数。它通常与 @Group 注解一起使用，以控制并行执行基准测试方法时的线程分配。

### 报表可视化

Options opt = new OptionsBuilder()  
 .include(JMHSample\_01\_2\_Warmup.class.getSimpleName())  
 .resultFormat(ResultFormatType.*JSON*)  
 .result("/D:/result.json")  
 .forks(**1**)  
 .build()**;**

在程序中加入报表结果的文件格式和保存位置，此例是保存成json格式文件。

然后将json文件拖动到如下网站形成可视化表格：

<https://jmh.morethan.io/>

形成如下的图表

