

CS205 Project 2

1. 思路分析

本次project要求用C/C++和Java分别写一个计算向量点乘的程序，并计算向量点乘所花费的时间。重点在于比较C/C++和Java的程序运行时间并且探讨可能造成差异的原因。

我预计从以下几个方向进行比较：

- ①比较向量长度为**不同数量级**时Java和C++运行时间的差异；
- ②比较向量为**int类型（整数类型）**和**float类型（浮点数类型）**时计算时间的差异；
- ③比较**不同优化级别**下运行时间的差异；

2. int类型向量比较

2.1 基础代码

我在C++和Java的程序实现上采用了相同的思路：首先创建两个向量，以当前时间为种子为向量的每一个值赋予**0-99之间的随机整数**，再用for循环计算两个向量的点乘结果，并计算其运算时间。其中C++使用`ctime`库和`chrono`库记录程序运行时间，Java则使用`System.nanoTime()`方法记录程序运行时间，二者均以ns为单位。C++和Java均使用VScode编写并在WSL系统下运行。

C++的具体代码如下：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <chrono>

using namespace std;
using namespace chrono;

int main() {
    srand(time(NULL)); // 设置随机数种子

    long long int n;
    cout << "请输入向量的长度： ";
    cin >> n;
    // 动态分配内存
    int* vec1 = new int[n];
    int* vec2 = new int[n];
    for (int i = 0; i < n; i++) {
        vec1[i] = rand() % 99; // 生成0-99之间的随机整数
    }

    for (int i = 0; i < n; i++) {
```

```

        vec2[i] = rand() % 99; // 生成0-99之间的随机整数
    }

    double result = 0.0;

    // 获取当前时间
    auto start_time = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < n; i++){
        result += vec1[i] * vec2[i];
    }

    // 获取程序结束时间点
    auto end_time = std::chrono::high_resolution_clock::now();

    cout << "点乘结果为: " << result << endl;

    // 计算程序运行时间
    auto elapsed_ns = duration_cast<nanoseconds>(end_time - start_time);
    cout << "程序运行时间为: " << elapsed_ns.count() << "ns" << endl;

    delete[] vec1, vec2; // 释放动态分配的内存

    return 0;
}

```

Java的具体代码如下:

```

import java.util.Random;
import java.util.Scanner;

public class vector {
    public static void main(String[] args) {
        Random random = new Random(); // 创建一个随机数生成器

        System.out.print("请输入向量的长度: ");
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] vec1 = new int[n];
        int[] vec2 = new int[n];
        for (int i = 0; i < n; i++) {
            vec1[i] = random.nextInt(100);
            vec2[i] = random.nextInt(100);
        }

        double result = 0.0;
        long startTime = System.nanoTime(); // 记录程序开始执行的时间

        for (int i = 0; i < n; i++){

```

```

        result += vec1[i] * vec2[i];
    }

    long endTime = System.nanoTime(); // 记录程序结束执行的时间
    long duration = endTime - startTime; // 计算程序运行时间，单位为纳秒
    sc.close();
    System.out.println("点乘结果为: " + result);
    System.out.println("程序运行时间为: " + duration + " ns");
}
}

```

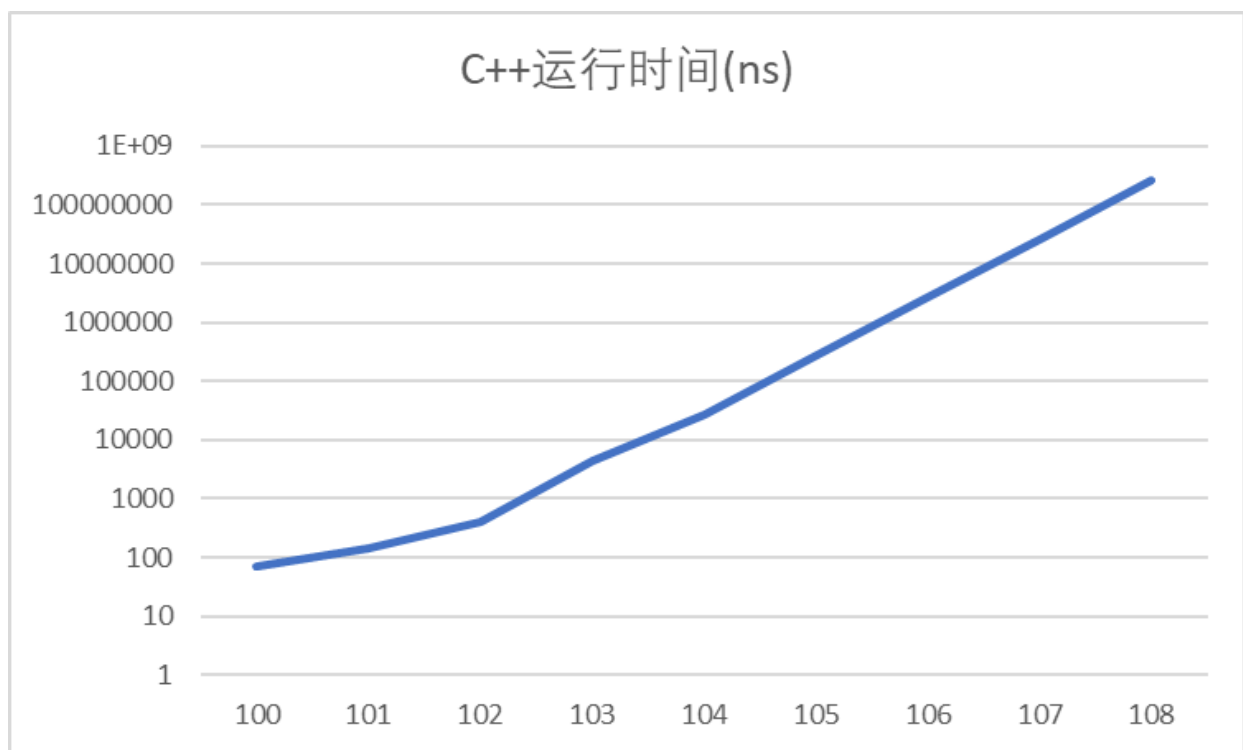
2.2 向量长度不同数量级时的点乘运行时间比较

接下来比较C++和Java在向量长度为不同数量级时的点乘运行时间，由于内存限制，向量长度为从 10^0 到 10^8 ，为确保测量数值准确，每个数量级均测量十次，去掉异常数据后取平均值。

C++测量结果如下：

数量级	运行时间(ns)
10^0	69.1
10^1	139.4
10^2	407.7
10^3	4417.3
10^4	26826.2
10^5	269704.1
10^6	2642419.2
10^7	25969873.5
10^8	263543026.1

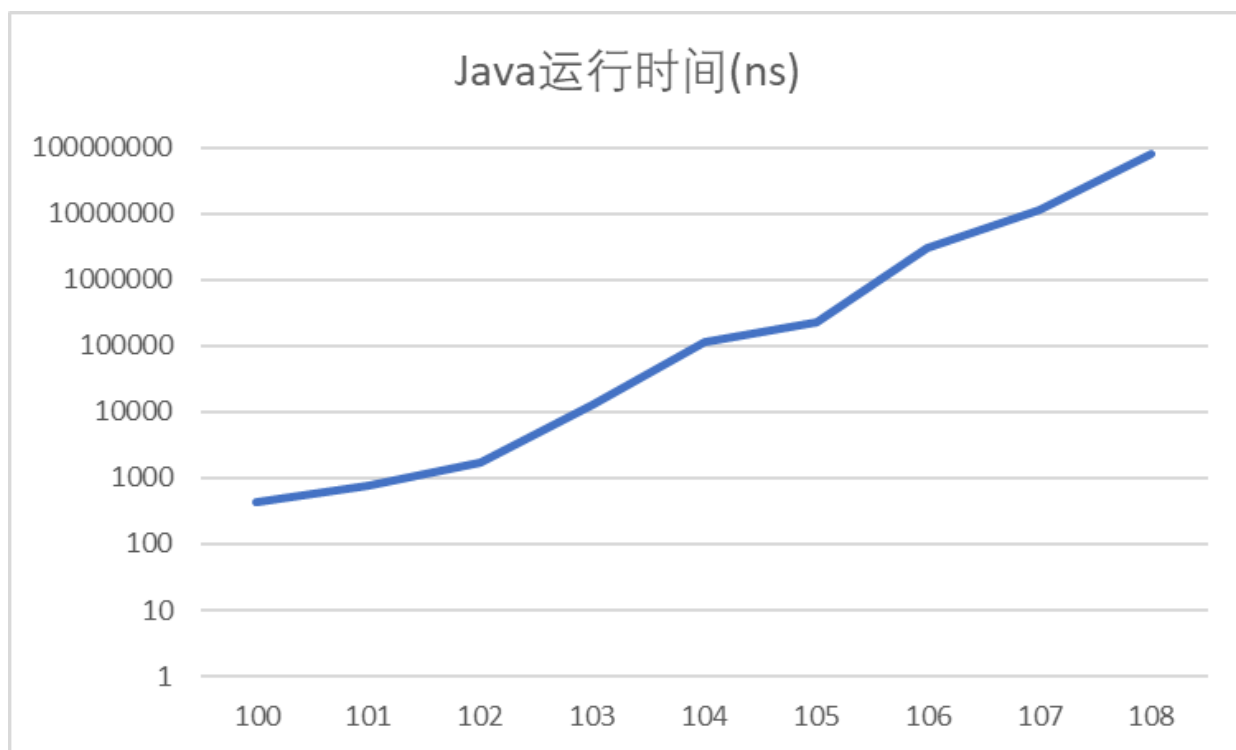
将其制作成折线图可以发现，运行时间与数量级成**强正比关系**。



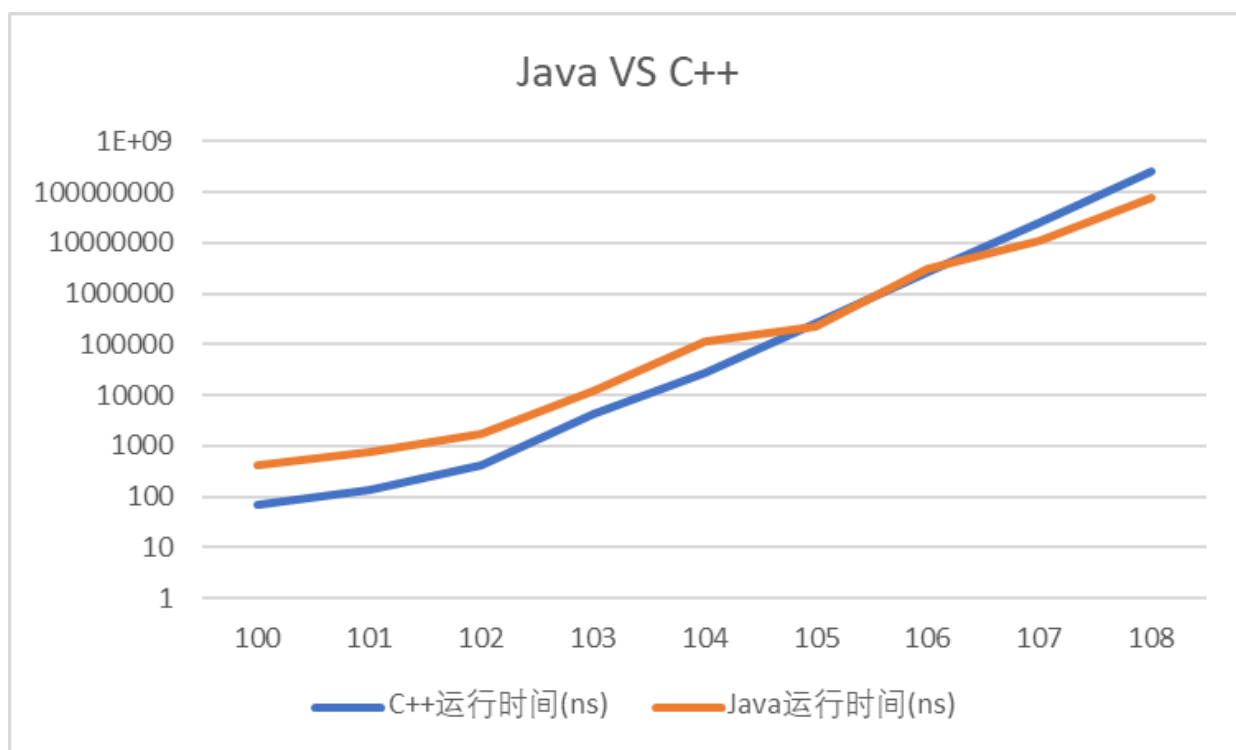
Java测量结果如下：

数量级	Java运行时间(ns)
10^0	419.5
10^1	767.4
10^2	1696.4
10^3	12580.1
10^4	114322.5
10^5	228687
10^6	3019567.8
10^7	11194649.4
10^8	78197288.6

将其制成折线图并且与C++对比可以发现，Java的运行时间与数量级并**不成强正比关系**。



将二者放在一起进行对比，我们可以发现，当数量级为 10^0 到 10^5 时，C++快于Java；当数量级为 10^5 到 10^6 时，C++与Java运行速度几乎持平；当数量级大于等于 10^7 时，Java反而快于C++。



2.3 原因分析

2.3.1 C++快于Java时

从前面测试的数据中我们可以发现，当数量级为 10^0 到 10^5 时，C++快于Java。为了探究其背后的原因，我向ChatGPT提出了我的疑问，他给了我以下答复：

12

为什么数量级小的时候，c++运行速度快于java



C++与Java都是编程语言，但它们的编译和运行方式不同，因此在某些情况下，C++可能比Java运行更快。

以下是一些可能导致数量级小的情况下C++运行速度快于Java的原因：

1. 编译方式不同：C++代码是通过编译器直接编译成机器码，而Java代码需要先编译成字节码，然后在运行时再由JIT编译器转换成机器码。因此，C++代码可以在编译时进行更多的优化，这可能会导致C++代码在运行时比Java代码更快。
2. 内存管理：Java有垃圾回收机制，这意味着它需要更多的时间来管理内存。而C++需要手动管理内存，这意味着它可以更好地控制内存使用，从而提高性能。
3. 优化能力：C++的代码可以进行更多的优化，因为它更接近底层语言。而Java的代码更加抽象化，使得优化变得更加困难。

因此我将从以上几个方面进行探讨。

2.3.1.1 编译方式的区别

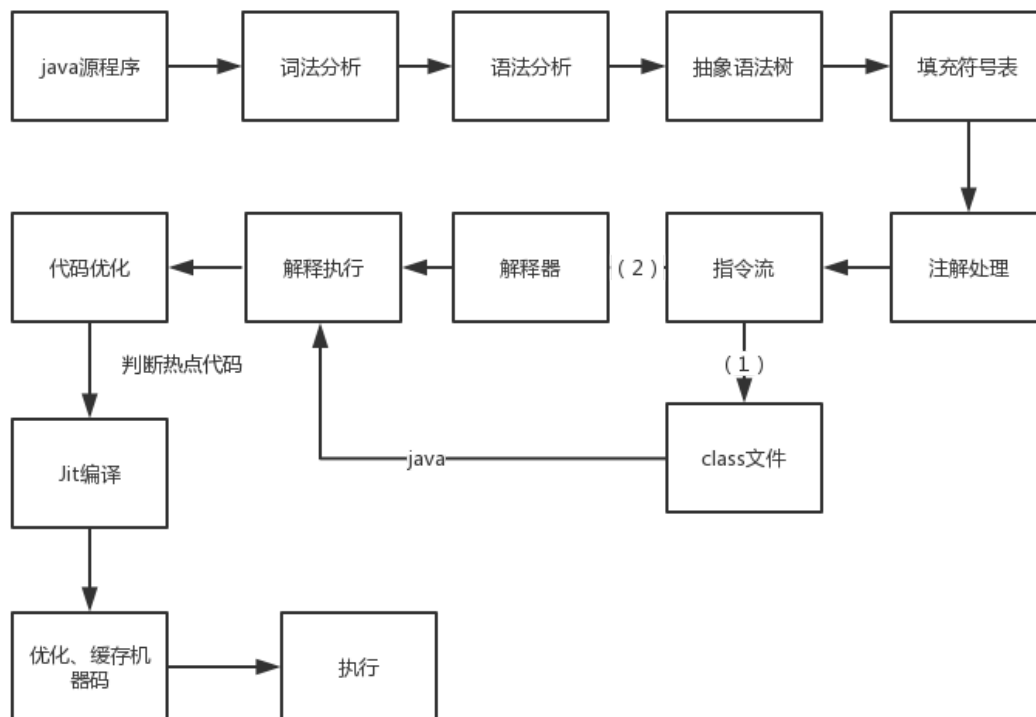
在计算两个向量点乘时，首先需要把代码编译或解释为机器可以识别的语言，然后再运行程序，因此程序的运行速度与编译方式有很大关系。

C++是编译型高级语言，程序要执行，必须要有编译器和链接器。编译过程分为四步：**预处理、编译、汇编、链接**。

1. **预处理**：C++编译器将读取源代码并对其中的以#开头的指令和特殊符号进行处理，如 `#define` 将会进行替换，`#include<...>` 将会导入头文件，而不必要的代码也会将其过滤掉。最后将处理结果存储在以 `“.i”` 为结尾的临时文件中。
2. **编译**：C++编译器将预处理完的文件进行词法、语法、语义分析和优化（如O1,O2,O3优化），在确认所有的指令都符合语法规则之后，将 `“.i”` 为结尾的临时文件翻译成 `“.s”` 结尾的汇编代码。template（函数模板和类模板）在编译阶段完成具现化。inline（内联）在编译阶段将调用函数的代码替换为函数本体，从而减少函数调用的开销。
3. **汇编**：将 `“.s”` 结尾的汇编代码转换成 `“.o”` 为结尾的二进制机器码。一般认为汇编和机器码是一一对应的。
4. **链接**：链接是将多个目标文件以及所需的库文件(如静态库和动态库)链接成最终的可执行文件。

Java则通常被认为是半编译半解释型的语言，Java文件编译过程包括两个阶段，第一阶段是**前端编译**（如Oracle的javac编译器）；第二阶段是**JIT编译**（如HotSpot中的C1、C2编译器）。

JVM整个编译过程如下图所示：



1. **前端编译**: 把Java源码文件 (.java) 编译成Class文件(.class)的过程, 即把满足Java语言规范的程序转化为满足JVM规范所要求格式的功能。许多Java语法新特性 (泛型、内部类等) 是靠前端编译器实现的, 而非虚拟机。编译成的class文件可以直接给JVM解释器解释执行, 省去编译时间, 加快启动速度。
2. **JIT编译**: 通过Java虚拟机 (JVM) 内置的即时编译器 (Just In Time Compiler, JIT编译器), 在运行时把Class文件字节码编译成本地机器码的过程。通过在运行时收集监控信息, 把“热点代码” (Hot Spot Code) 编译成与本地平台相关的机器码, 并进行各种层次的优化。

总体来说, C++代码被编译成本地机器码, 因此一般不能跨平台运行。而Java代码则被编译成Java字节码, 并在运行时被转换成机器码, 因此一般可以跨平台运行。由此可知, Java相比于C++多了编译成Java字节码的过程, 因此一般情况下, Java的运行速度是慢于C++的。

2.3.1.2 内存管理的区别

在计算两个向量点乘时, 需要从内存中读取向量的数据, 进行运算, 然后将结果写回内存。如果内存管理不好, 例如内存分配不充分或频繁的内存分配和释放等, 会导致内存访问产生较多的延迟, 从而影响计算的速度。因此程序的运行速度和内存管理也有很大关系。

C++使用**手动内存管理**, 即程序员需要显式地分配和释放内存。这种方式的好处是程序员可以控制内存的分配和释放, 可以优化内存使用, 避免不必要的内存分配和释放操作。当C++读取数组时, 往往通过指针访问内存地址, 这样的方式更加高效。

Java则使用**自动内存管理**，即由JVM自动进行内存分配和释放。JVM的垃圾回收机制可以自动释放不再使用的内存，避免了内存泄漏等问题。但是，垃圾回收机制需要花费一定的时间来扫描内存，查找不再使用的对象，导致程序运行速度下降。且在Java中，数组是一种引用类型，引用是一个存储在栈内存中的变量，它指向堆内存中的对象，Java是通过对数组对象的引用和索引计算得到的数组元素。在引用类型中也存在垃圾回收机制，因此需要花费一定时间扫描内存，运行速度会减慢。

综上所述，Java在进行内存管理时需要花费更多时间，但是相比于C++而言更加安全，因此Java会慢于C++。

2.3.1.3 优化方式的区别

在 C++ 中，g++编译器使用的优化级别是O0，也就是不开启任何优化选项。但编译器可以进行多种优化，包括但不限于：

1. **内联函数**：将函数调用替换为函数体，以减少函数调用的开销；
2. **循环展开**：将循环展开成一系列重复的语句，以减少循环的开销；
3. **常量折叠**：将表达式中的常量计算出结果并替换为结果，以减少表达式计算的开销；
4. **删除未使用的变量和函数**：将未使用的变量和函数从程序中删除，以减少程序的大小和复杂度；
5. **优化存储器访问**：将程序中频繁访问的变量放置在 CPU 缓存中，以加快存储器访问速度。

在 Java 中，Java 编译器 javac 默认会启用一些基本的优化选项。但编译器同样可以进行多种优化，包括但不限于：

1. **字节码优化**：编译器可以对 Java 代码进行优化，以生成更高效的字节码；
2. **JIT 编译**：在程序运行时，编译器可以对热点代码进行即时编译，以生成更高效的本地代码；
3. **自适应优化**：编译器可以根据程序运行时的情况对代码进行自适应优化，以提高程序性能；
4. **垃圾回收优化**：编译器可以优化垃圾回收算法和垃圾回收器的实现，以提高垃圾回收的效率和减少垃圾回收的影响；
5. **类加载优化**：编译器可以对类的加载和初始化进行优化，以提高程序启动速度和减少内存占用。

在运算数量级比较小的情况下，C++ 的优化效果会更突出。因为 C++ 在编译期间可以进行更加精细的优化，而且不受 JVM 的运行时环境限制。而且Java的JIT编译是在运行时进行的，所以在运算数量级小的时候，优化效果并不明显，甚至可能会有一些性能损失。因此在数量级小的时候，Java会慢于C++。

2.3.2 Java快于C++时

从先前的图中可以看出，Java的运行速度在逐渐提升甚至反超C++。为了获得后续探讨的方向，我同样向ChatGPT发出疑问，他的回答如下：

12

为什么运算数量级大的时候，C++运行速度反而比Java慢



在运算数量级大的情况下，Java 的运行速度可能会比 C++ 更快。这是因为 Java 的 JIT 编译器可以根据程序的实际情况，动态地生成优化代码，以适应不同的情况。这种动态优化的机制使得 Java 能够更好地利用现代计算机硬件的特性，比如多核处理器、高速缓存等，从而在运算数量级大的情况下表现更好。

另外，Java 运行时环境也提供了一些优化机制，比如垃圾回收、即时编译等。这些机制可以有效地减少内存管理方面的开销，从而提高程序的性能。与此相比，C++ 需要手动管理内存，这可能会增加一些开销，并且对程序员的技能水平有一定的要求。

需要注意的是，这只是一个概括性的结论，具体的情况可能会因为程序的具体实现方式、硬件环境等因素而有所不同。在实际应用中，应该根据具体的应用场景和需求，对程序进行综合考虑和优化，以达到最优的性能表现。

从图中可以看到，ChatGPT着重提及了Java的JIT编译器，这也是先前分析Java的编译方式时曾涉及到的，因此我将着重从这个方向进行分析。

2.3.2.1 JIT编译器的优化

在分析JIT编译器的功能之前，我们首先需要确认，JIT编译器是否真的对程序运行时间有影响。经过搜索，使用 `-Djava.compiler=NONE` 指令可以关闭JIT编译器，因此我们可以通过这个指令来探讨JIT编译器的影响。

当数量级为 10^8 时，正常运行程序得到的结果为：

```
● wyuuu@LAPTOP-AF3F78HF:/mnt/d/资料/C++/project/project2$ java vector
请输入向量的长度：100000000
点乘结果为：2.45031407592E11
程序运行时间为：77027451 ns
```

此时运行时间约为77ms。

关闭JIT编译器后，得到的结果为：

```
● wyuuu@LAPTOP-AF3F78HF:/mnt/d/资料/C++/project/project2$ java -Djava.compiler=NONE vector
请输入向量的长度：100000000
点乘结果为：2.45040647672E11
程序运行时间为：1002853828 ns
```

此时运行时间约为1.04s，与启用JIT编译器得到的结果只比约为13.5倍。而在数量级为 10^8 时，用C++测量出来的时间约为0.26s。这说明，当未启用JIT编译器时，Java运行速度是慢于C++的，符合我们先前的判断。因此，接下来我将探讨JIT编译器究竟有什么功能。

JIT编译器 (just in time 即时编译器)，当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为**热点代码**(Hot Spot Code)，为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化。说JIT比解释快，并非说“编译”这个操作比“解释”操作更快，而是说执行JIT编译后的机器码会更快。

那么为何JIT只对热点代码进行优化呢，原因如下：

1. 时间

对只执行一次的代码做JIT编译再执行，可以说是得不偿失。对只执行少量次数的代码，JIT编译带来的执行速度的提升也未必能抵消掉最初编译带来的开销。只有对频繁执行的代码，JIT编译才能保证有正面的收益。

2. 空间

对一般的Java方法而言，编译后代码的大小相对于字节码的大小，膨胀比达到10x是很正常的。同上面说的时间开销一样，这里的空间开销也是，只有对执行频繁的代码才值得编译，如果把所有代码都编译则会显著增加代码所占空间，导致“代码爆炸”。

那么什么是**热点代码**呢，一般有两种：

1. 被多次调用的方法
2. 被多次执行的循环体

在我的代码中，计算向量点乘的方式正是通过多次循环，因此满足热点代码的条件，JIT会对其进行优化。那么JIT是怎么判断这段代码是热点代码的呢？根据我的搜索，JIT大概率是运用了热点探测技术中的**回边计数器**，它的作用就是统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”。作用是在遇到回边（可以狭义的理解为循环，但并非所有循环都是回边）时，虚拟机会检测改代码是否存在已编译版本 - 是就执行，否加调用计数器+1，并检测是否超过阈值。

经过这样的探测，当回边计数器到达一定标准时，JIT将会把代码转为本地机器码，并对其进行优化，因此**当运算数量级大时，Java的运行速度反而快于C++**。

2.3.2.2 内存访问模式的区别

经过搜索与学习，除了JIT的优化外，Java与C++内存访问模式的区别也可能会导致二者之间运行速度的差异。

C++ 可以使用指针进行低级别的内存访问，即直接操作内存地址。这种方式可以让程序员更加灵活地控制内存访问的顺序和方式，对于小数据集可能会比Java 更快。但是，当程序中频繁使用指针进行内存访问时，可能会出现以下问题：

1. **内存碎片**：指针在内存中访问数据时，如果内存中的空间是不连续的，那么就会产生内存碎片。这会导致内存分配效率低下，内存分配和释放的速度变慢。
2. **缓存未命中**：在处理大数据集时，由于内存访问模式的不连续性，程序在访问内存时可能会频繁地触发缓存未命中，从而导致程序性能的下降。

因此C++ 的指针访问可能会导致内存访问模式的不连续性，即程序访问内存的顺序不连续，导致缓存命中率降低，程序性能将会受到影响。

相比之下，Java 采用的是自动内存管理机制，可以通过垃圾回收机制来减少内存碎片的产生，从而提高内存的连续性，进而提高程序的性能。此外，Java 也提供了数组和集合等高级数据结构，这些数据结构在内存中是连续存储的，可以更好地支持数据访问。因此，在运算数量级较大时，Java反而比C++更快。

3. float类型向量比较

3.1 基础代码

比较完int类型的向量，接下来比较float类型的向量。

C++中向量生成方式由 `int* vec1 = new int[n];` 更改为 `float* vec1 = new float[n];`。

Java中则由 `int[] vec1 = new int[n];` 更改为 `float[] vec1 = new float[n];`。

同时生成的向量值为0~1之间的随机浮点数，C++具体代码如下：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <chrono>

using namespace std;
using namespace chrono;

int main() {
    srand(time(NULL)); // 设置随机数种子

    int n;
    cout << "请输入向量的长度: ";
    cin >> n;
    // 动态分配内存
    float* vec1 = new float[n];
    float* vec2 = new float[n];

    for (int i = 0; i < n; i++) {
        vec1[i] = static_cast<float>(rand()) / RAND_MAX; // 生成0-1之间的随机浮点数
    }

    for (int i = 0; i < n; i++) {
        vec2[i] = static_cast<float>(rand()) / RAND_MAX; // 生成0-1之间的随机浮点数
    }

    double result = 0.0;

    // 获取当前时间
    auto start_time = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < n; i++){
        result += vec1[i] * vec2[i];
    }

    // 获取程序结束时间点
    auto end_time = std::chrono::high_resolution_clock::now();
```

```

cout << "点乘结果为: " << result << endl;

// 计算程序运行时间
auto elapsed_ns = duration_cast<nanoseconds>(end_time - start_time);
cout << "程序运行时间为: " << elapsed_ns.count() << "ns" << endl;

delete[] vec1,vec2; // 释放动态分配的内存

return 0;
}

```

Java具体代码如下:

```

import java.util.Random;
import java.util.Scanner;

public class vector {
    public static void main(String[] args) {
        Random random = new Random(); // 创建一个随机数生成器

        System.out.print("请输入向量的长度: ");
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        float[] vec1 = new float[n];
        float[] vec2 = new float[n];
        for (int i = 0; i < n; i++) {
            vec1[i] = random.nextFloat();
            vec2[i] = random.nextFloat();
        }

        double result = 0.0;
        long startTime = System.nanoTime(); // 记录程序开始执行的时间

        for (int i = 0; i < n; i++){
            result += vec1[i] * vec2[i];
        }

        long endTime = System.nanoTime(); // 记录程序结束执行的时间
        long duration = endTime - startTime; // 计算程序运行时间, 单位为纳秒
        sc.close();
        System.out.println("点乘结果为: " + result);
        System.out.println("程序运行时间为: " + duration + " ns");

    }
}

```

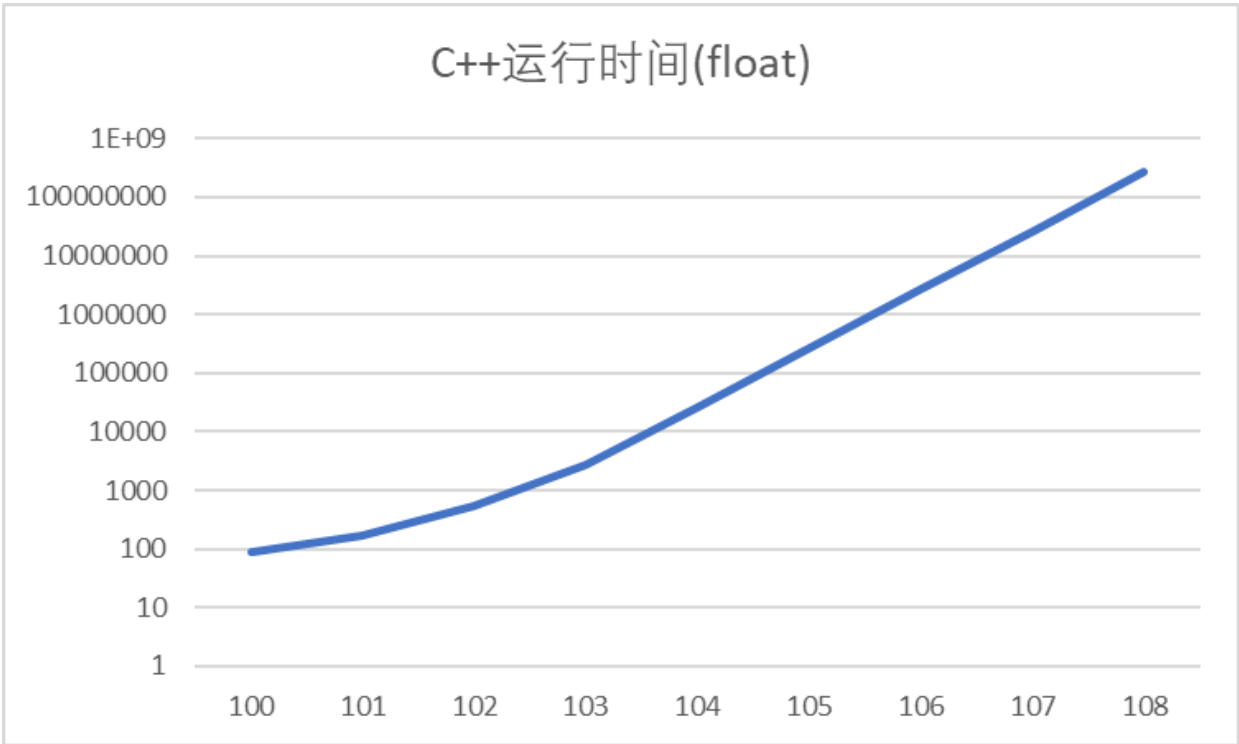
3.2 Java与C++运行时间的比较

同先前一样，向量长度为从 10^0 到 10^8 ，为确保测量数值准确，每个数量级均测量十次，去掉异常数据后取平均值。

C++测量结果如下：

数量级	C++运行时间(ns)
10^0	88.1
10^1	173.1
10^2	556.2
10^3	2671
10^4	25809.2
10^5	264960.4
10^6	2672185.6
10^7	26809701.4
10^8	267407915.2

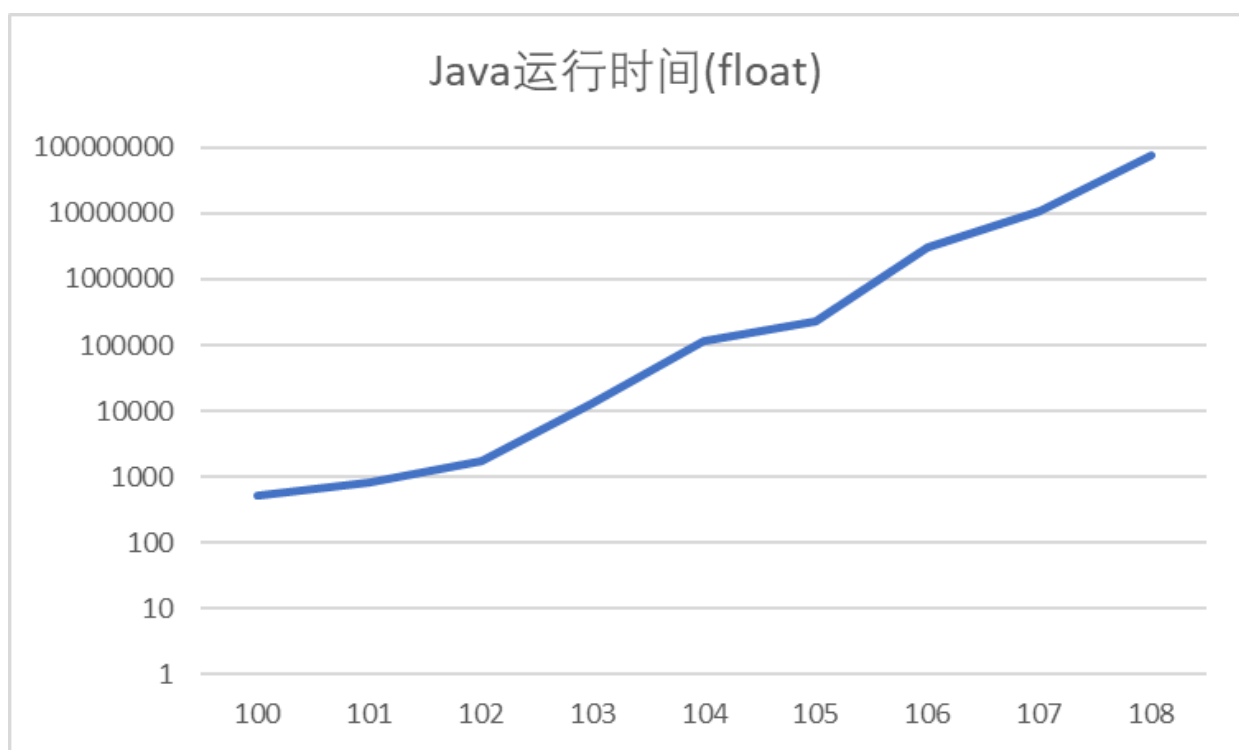
绘制折线图，数量级和运行时间仍然是**强正比**的关系。



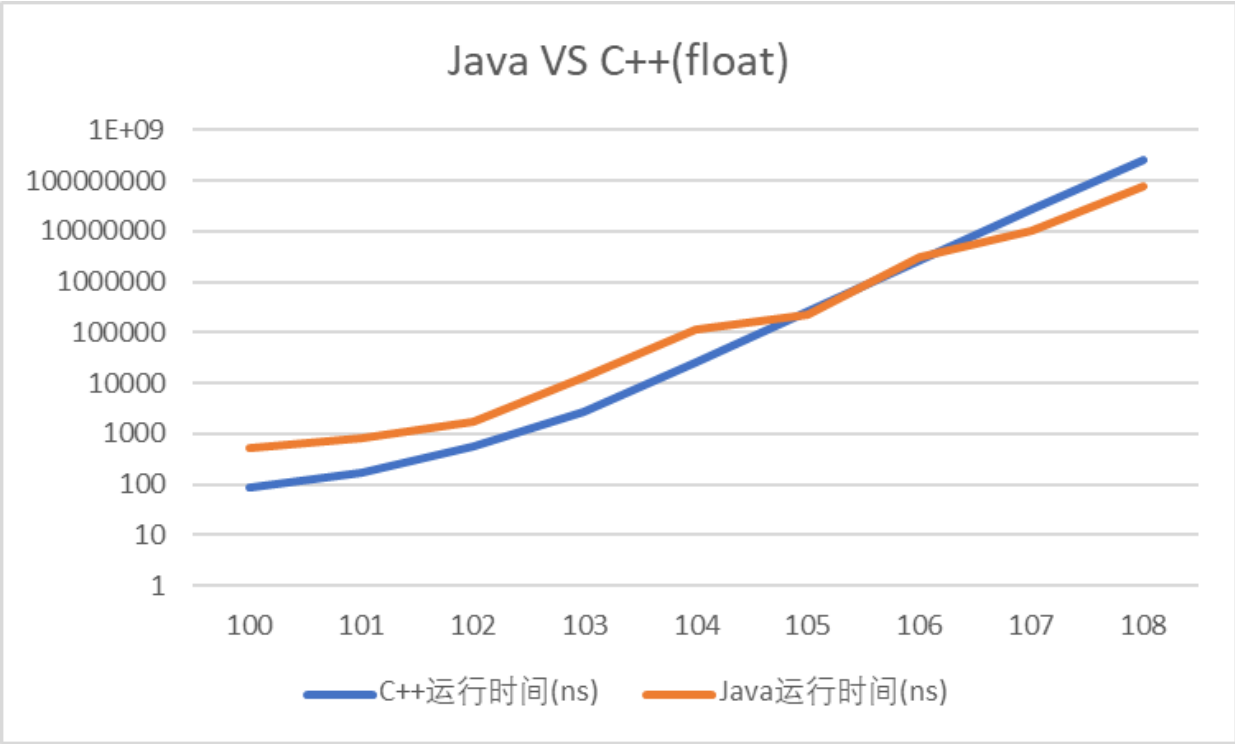
Java测量结果如下：

数量级	Java运行时间(ns)
10^0	512.5
10^1	841.4
10^2	1733.4
10^3	12706
10^4	117033.6
10^5	232584.4
10^6	3120073.6
10^7	10653880.2
10^8	75899289.6

绘制折线图：



将二者放在一起对比，可以看出与int类型时没有明显区别。

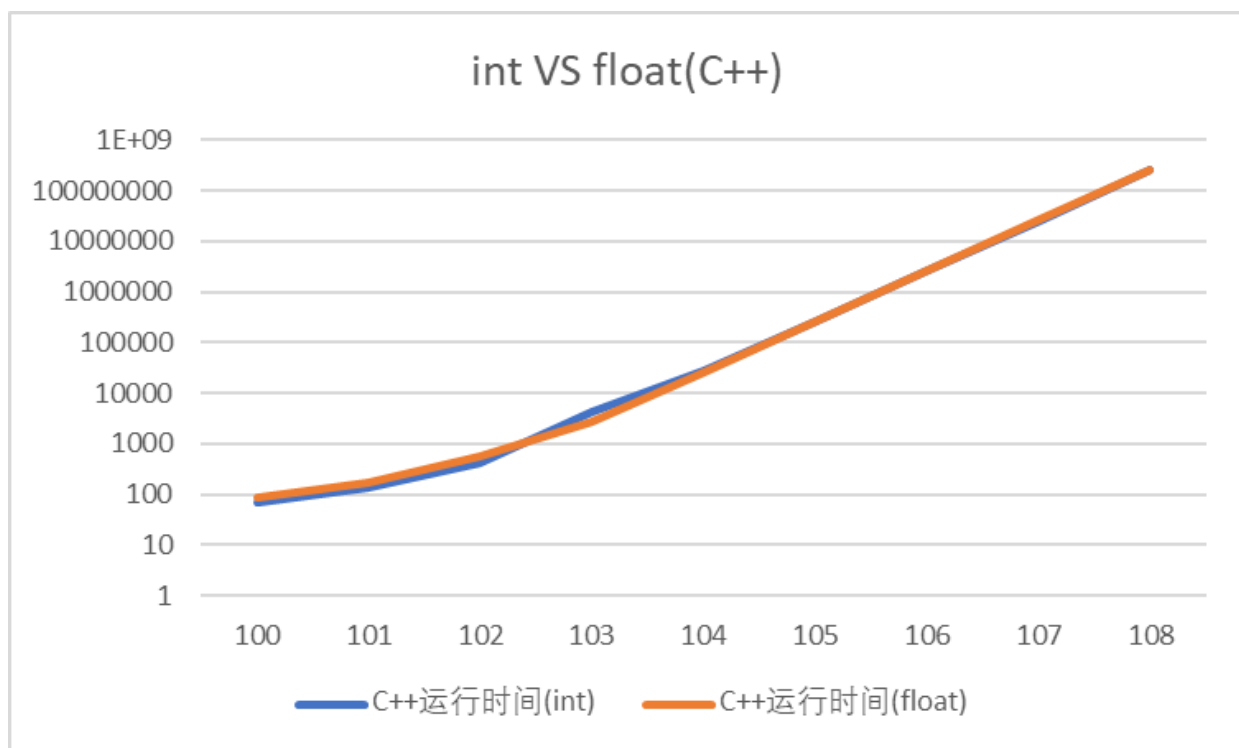


3.3 C++中int与float的比较

将C++中的int类型与float类型放在一起进行比较，表格如下：

数量级	C++运行时间(int)	C++运行时间(float)
10^0	69.1	88.1
10^1	139.4	173.1
10^2	407.7	556.2
10^3	4417.3	2671
10^4	26826.2	25809.2
10^5	269704.1	264960.4
10^6	2642419.2	2672185.6
10^7	25969873.5	26809701.4
10^8	263543026.1	267407915.2

从折线图中，我们可以看到，二者的运行时间几乎没有差距。

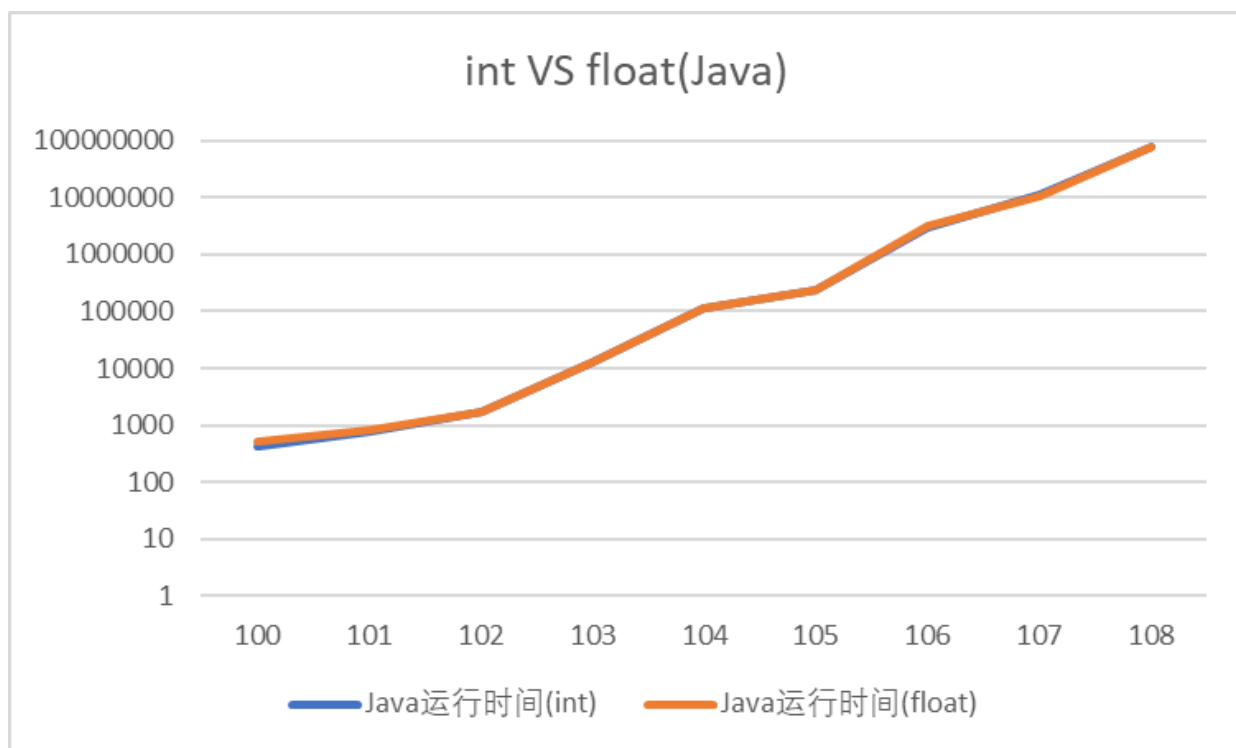


3.4 Java中int与float的比较

将Java中的int类型与float类型放在一起比较，表格如下：

数量级	Java运行时间(int)	Java运行时间(float)
10^0	419.5	512.5
10^1	767.4	841.4
10^2	1696.4	1733.4
10^3	12580.1	12706
10^4	114322.5	117033.6
10^5	228687	232584.4
10^6	3019567.8	3120073.6
10^7	11194649.4	10653880.2
10^8	78197288.6	75899289.6

同样，从折线图中，我们可以发现，二者运行时间差距忽略不计。



4. 不同优化级别比较

默认情况下, C++编译器g++使用的优化级别是O0, 也就是不开启任何优化选项。Java 编译器 javac 默认会启用一些基本的优化选项, 例如去除没有使用的代码、常量折叠、代码内联等。而JVM中的JIT编译器也会在程序运行时动态地将热点代码进行优化。

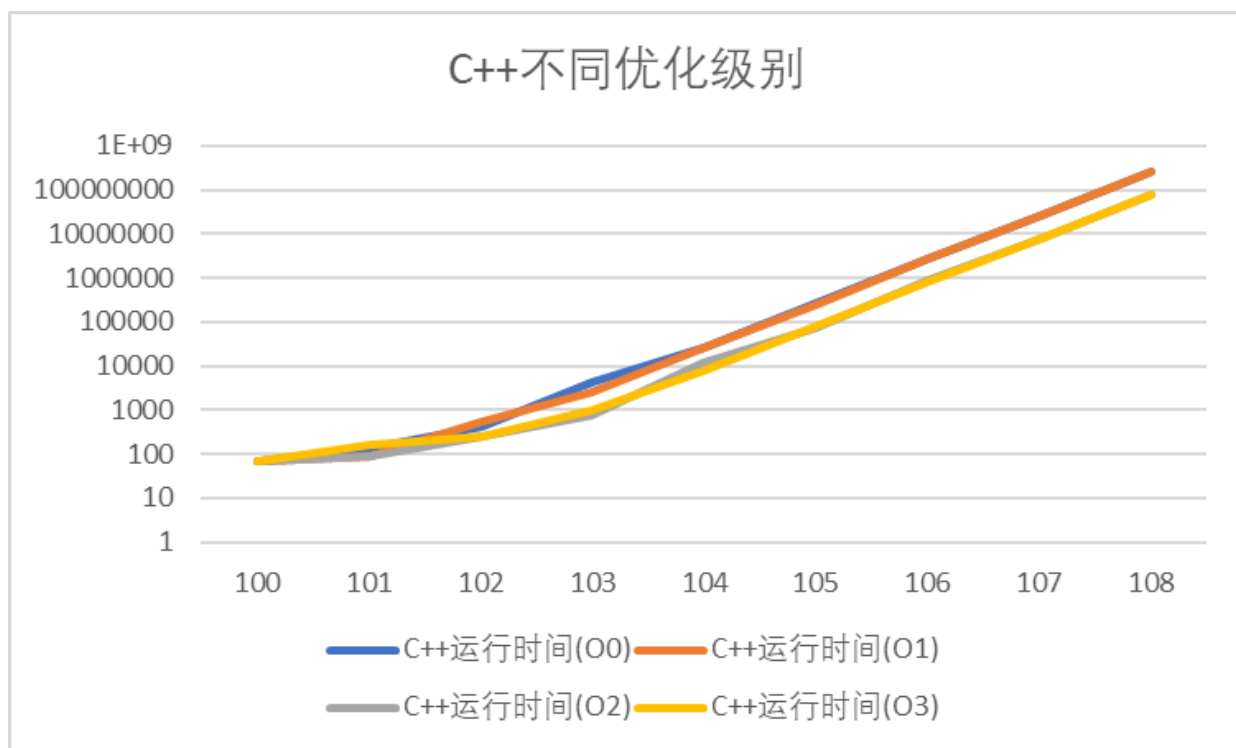
接下来分别讨论向量为int类型时, 优化级别为O1、O2、O3的情况。

4.1 C++自身各个优化级别的比较

分别使用O1、O2、O3级别优化, 得出的数据如下:

数量级	C++运行时间(O0)	C++运行时间(O1)	C++运行时间(O2)	C++运行时间(O3)
10^0	69.1	66.2	67.4	70.2
10^1	139.4	86.2	91	162.6
10^2	407.7	541	246.4	245.3
10^3	4417.3	2601	765.1	990
10^4	26826.2	25749.4	11534	8081.8
10^5	269704.1	253598.2	73832.4	80684.2
10^6	2642419.2	2631257.2	894865.6	833497.8
10^7	25969873.5	25698513.6	7777481.4	7701707.5
10^8	263543026.1	261459756.4	76771429.4	78991748.6

制成折线图可以发现, 当优化级别为O1时, 运行时间和未优化时相差不大, 而优化级别为O2时, 运行时间明显有所降低, 而O2与O3的差异却不甚明显。



根据调查，g++的三个优化级别功能如下：

1. -O1:

编译器优化后，增加编译时间，并处理大函数时会占用更大的内存。

优化结果，使得程序文件变小，执行时间变短。

开启一些基本的优化，如去除无用代码、简化表达式等。

2. -O2:

比O1优化更多。g++会尽可能的引入不造成空间-时间（不为了降低执行时间而增大使用内存，或降低使用内存而增加执行时间）影响的优化。和没有优化相比，这选项增加了编译时间，同时提高了代码执行性能。

Level 2优化打开了所有Level 1打开的选项，并且在 -O1 的基础上增加一些较为复杂的优化，如函数内联、循环展开等。

3. -O3:

在 -O2 的基础上增加更多的优化，如自动向量化、函数调用优化等。

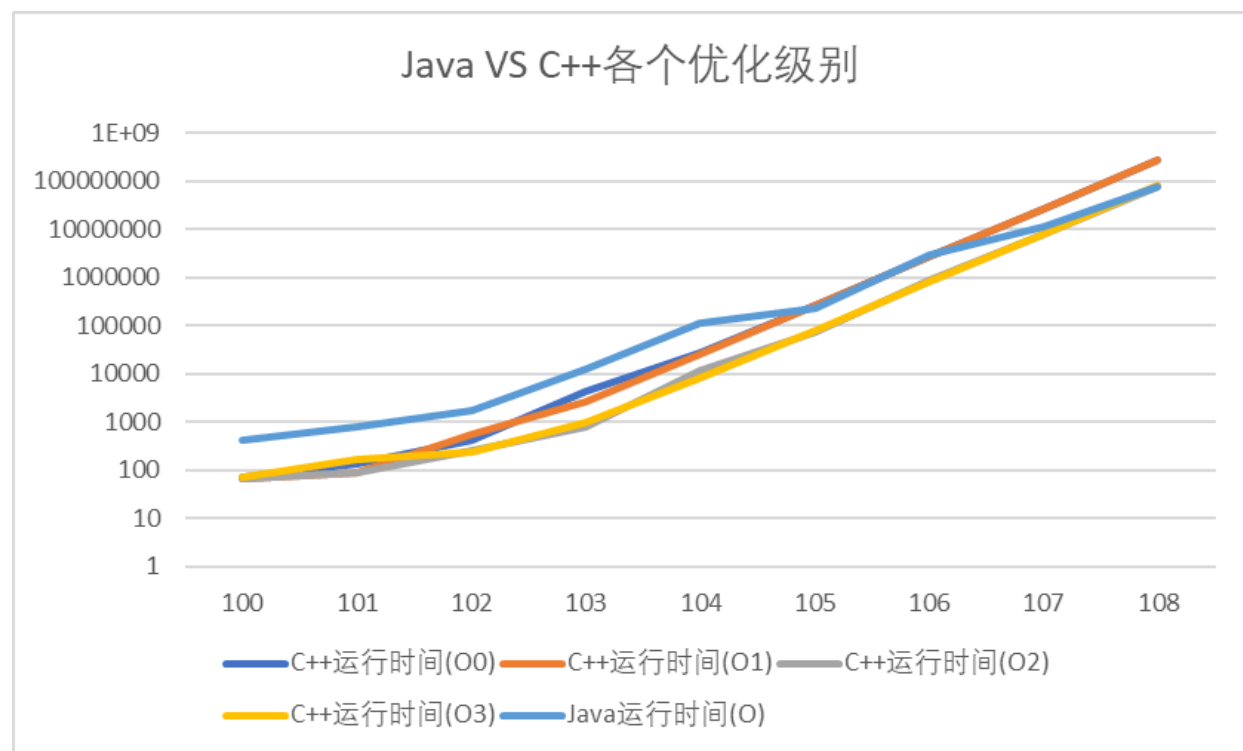
由上述结果我们可以发现，对多次循环的优化很可能是在O2时进行优化，因此对for循环计算向量点乘的运行速度有较大的提升。

4.2 Java与C++各个优化级别的比较

将Java与C++各个优化级别进行比较的数据如下：

数量级	C++运行时间(O0)	C++运行时间(O1)	C++运行时间(O2)	C++运行时间(O3)	Java运行时间(O)
10^0	69.1	66.2	67.4	70.2	419.5
10^1	139.4	86.2	91	162.6	767.4
10^2	407.7	541	246.4	245.3	1696.4
10^3	4417.3	2601	765.1	990	12580.1
10^4	26826.2	25749.4	11534	8081.8	114322.5
10^5	269704.1	253598.2	73832.4	80684.2	228687
10^6	2642419.2	2631257.2	894865.6	833497.8	3019567.8
10^7	25969873.5	25698513.6	7777481.4	7701707.5	11194649.4
10^8	263543026.1	261459756.4	76771429.4	78991748.6	78197288.6

制成折线图，我们可以发现，当C++开启O2以上级别的优化时，C++的运行速度往往是快于Java的，即使当数量级增大到 10^8 时，二者也只是持平。



5. 总结

在Java与C++的运行速度比较上，C++一般都是快于Java的，只有当数量级很大且C++没有开启优化时，Java才有可能比C++更快。

这是因为Java相比于C++，有自身独特的自动内存管理和垃圾回收机制，使得开发人员可以不必关心程序的内存管理，但这也造成了程序的性能降低。并且Java相比于C++多了将代码转换为字节码的步骤，因此在执行上往往比C++要慢。而Java的JIT编译器也有其自身独特的优势，在程序运行过程中，可以优化程序的性能，从而获得更快的运行速度，但是其分析过程往往也需要一定时间。

因此，对于开发程序而言，如果对性能要求比较高，可以选择C++；如果更看重开发效率与安全性，可以选择Java。