# Computing the Dot Product of Two Vectors


Student ID: 12312710
Name: 胥熠/Brighton
Course: CS219: Advanced Programming
Date: 2025 年 3 月 30 日


---

## 目录

# 1 前言

无论是上 Java 课，还是 C/C++ 课，我们都被灌输 Java 就是要比 C 慢的一种理念。但真的是这样吗？在完成这个 Project 的过程中我的理解被一次次的击碎，那自称快的 C 怎么在某些场景会被 Java 拉开好几倍的速度差距？为了探索这个问题，我们必须了解 C 语言和 Java 的底层原理和为了加快程序运行速度采取的手段。同时，在这个探索过程中真正的理解 C 和 Java 这两种语言实现方式的根本差距。

# 2 需求分析

## 2.1 点乘实现

我们在初中就学过，向量 $A$ 点乘向量 $B$ 有：

$$A \cdot B = \sum A_i \times B_i$$

该算法的时间复杂度为 O(n)，且不存在进一步降低时间复杂度的可能。在这里，我们使用两个数组分别储存在两个数组中，使用 for 循环计算乘积并加到 result 中。

在这个实验中，我并不在意 result 的精度和溢出问题，我们只关注整个算法中乘法和加法的速度。但是我们必须要将乘积加到 result 中并最后打印 result，否则会在某些优化中整个算法被优化而无法实验。

## 2.2 如何记录时间？

既然要比较两个不同语言的处理速度，我们需要有一个精准记录时间的手段。Java 比较方便，直接调用 System.nanoTime() 就可以了。但是 C 语言中有个陷阱，*clock*() **测量的是 CPU 时间，而不是时钟时间（wall-clock time）**。在多线程或高负载系统下，CPU 时间和实际经过的时间会有差异。所以为了精确的记录时间，我们肯定不能用这个不稳定的 clock() 函数。同时，我还想要计数方法能够跨平台，于是，我找到了可以用以下方法实现：

```
#ifdef _WIN32
    #include <Windows.h>
    typedef LARGE_INTEGER TimePoint;
    void get_time(TimePoint *t) {
        QueryPerformanceCounter(t);
    }
    double time_diff_seconds(TimePoint start, TimePoint end) {
        LARGE_INTEGER freq;
        QueryPerformanceFrequency(&freq);
        return ((double)(end.QuadPart - start.QuadPart)) / freq.QuadPart;
    }
#else
    #include <time.h>
    #include <stdio.h>
```

```
    typedef struct timespec TimePoint;
    void get_time(TimePoint *t) {
        clock_gettime(CLOCK_MONOTONIC, t);
    }
    double time_diff_seconds(TimePoint start, TimePoint end) {
        return (end.tv_sec - start.tv_sec) +
               (end.tv_nsec - start.tv_nsec) / 1000000000.0;
    }
#endif
```

这个代码先是检测了运行环境，然后按不同的系统分别实现了能够精确计算经过时间的方法。在 Windows 系统下，我们直接利用 Windows.h 库下面的 QueryPerformaceCounter 方法，而 Linux 下则是使用 time.h 库。有了这个之后，我们就可以方便且准确的记录程序运行时间了。

## 2.3 向量从哪里来？

### 2.3.1 采用预文件读取还是现场生成？

本 Project 和两个矩阵相乘有所不同，两个向量相乘的时间复杂度就是 $O(n)$，非常快速。如果是使用文件 IO 的形式，读取文件速度可能会稍微慢，而且如前面所言有个问题——我的计算机很快，计算两个 100000000（一亿）int 类型的向量点积只需要大约 2 3 秒，而两个一亿 int 数组按每个数字十个字节算的话要占用接近 2GB 的空间！于是我在项目的一开始就决定了每次都现场生成随机数后点积，既节省了磁盘空间还省去了文件读取的繁琐，随生成随测。

### 2.3.2 如何快速生成随机数？

随之而来的一个问题，如何生成随机数？直接使用 C 或者 Java 内置的 random 函数固然方便，但是在 Windows 系统下，C 的 rand() 生成数字范围只有 0-32767。还有一个问题：有点慢。在这里，我们采用 xorshift 的随机方法，能够加速随机数的生成速度，而且生成范围可以覆盖指定的数据类型。

在 2*100,000,000（两亿）的数据规模下，两个方法的速度三次取平均值对比：

C 语言（直接使用 gcc 编译，不采用优化）：

```
    unsigned int xorshift32() {
    xorshift_state ^= xorshift_state << 13;
    xorshift_state ^= xorshift_state >> 17;
    xorshift_state ^= xorshift_state << 5;
    return xorshift_state;
}
...
for (int i = 0; i < size; i++) {
        a[i] = xorshift32();
        b[i] = xorshift32();
}
```

使用 XORShift：(2.314432 + 2.263346 + 2.481049)/3 = 2.35294s 使用 rand()：(3.181492 + 3.180839 + 3.225858)/3 = 3.19606s 速度提升了 35.84%，时间减少了 26.4%。额，快的不是很多，但也算提升了吧。

Java:

```
    public static int xorshift32(){
        xorshift32State ^= (xorshift32State << 13);
        xorshift32State ^= (xorshift32State >> 17);
        xorshift32State ^= (xorshift32State << 5);
        return xorshift32State;
}
...
for (int i = 0; i < size; i++) {
            a[i] =  xorshift32();
            b[i] =  xorshift32();
}
```

使用 XORShift：(0.567251 + 0.567911 + 0.568329)/3 = 0.567803s 使用 random.nextInt()：(3.497867 + 3.567083 + 3.440878)/3 = 3.50194s 速度提升了 193.41%，时间减少了 83.78%!

同时，C 语言肯定快于 Java 的论点甚至在我设计实验的时候就破碎了。如你可见：在上述两个逻辑完全一致的 XORShift 中，生成 2 亿个随机数 Java 比 C 快了三倍有余！这是为什么呢？我们在后面在讨论。

### 2.3.3 XORShift 会影响速度吗？

但是这又又衍生出来一个问题:两个同样都是 int 类型的数字相乘,但是一个范围在 [0,32767],一个范围在整个 int 表示范围，速度会有差异吗？我们也来做一下实验：

直接用 100000000 个样本计算一百次速度，为什么是一百次？因为这个数量就可以使用中心极限定理将近似看成正态分布了：我们可以运用在工程概率统计中的 Welch's t-test 计算两个方法时间是否可以被认为相同，详细数据见附录：

Python:

```
import scipy.stats as stats
xor = [*数据*]
ran = [*数据*]
# Welch's t-test
t_stat, p_value = stats.ttest_ind(xor, ran, equal_var=False)
print(f"t = {t_stat:.2f}, p = {p_value:.3f}")
# Cohen's d
import numpy as np
mean_diff = np.mean(xor) - np.mean(ran)
pooled_std = np.sqrt((np.std(xor, ddof=1)**2 + np.std(ran, ddof=1)**2) / 2)
cohens_d = mean_diff / pooled_std
```

```
print(f"Cohen's d = {abs(cohens_d):.2f}")
```

诶，奇怪的来了，设零假设为两种方法最后计算点积速度相同，我们计算出来的 p 值为 0.037，拒绝了零假设！也就是说我们有 96.6% 的把握，用 xor 生成出来的范围更大的数，进行点积速度还会比小范围的数还要快！我觉得这有点反直觉，因为两个更大的数相乘怎么会更加快呢。我发现，ran 中有几个极其极端的数据，有一个甚至接近平均值的两倍，这应该是问题数据，需要剔除。各剔除十个最大最小值后计算得：p= 0.217，在显著性水平 $\alpha$ = 0.05 下可以说无显著差异了。

通过查询得知，C 语言编译后使用 CPU 通过二进制乘法指令（如 mul 或 imul。在 CPU 中，可能又直接用组合逻辑电路或者 Carry-Save Adder 等获取乘积结果。总而言之，只要是 int 类型整数，无论里面存的数无论大还是小，乘法操作并不会优化，32 位都要参与运算（除非除以一些常数如 2 的幂，编译器可能会优化成位运算）。这似乎告诉我们选择数据类型需要更加谨慎，能用 short 甚至 char 的就不要用 int 了。对于浮点数，同时我也查阅了 IEEE754 标准，“对于所有浮点数，无论数值的大小如何，其内部表示均严格遵循上述格式”，因此处理数字的时候，无论数值大小如何，CPU 内部读取和操作的都是固定长度的数据字，也是按照固定的算法计算。同时，在 Java 虚拟机的计算中，于是，后面的计算中，我会直接大量的使用 xor 方法生成随机数——**反正不会影响计算速度**。

# 3 正式实验

## 3.1 运行环境

Lenovo Legion Y9000P 运行 x86 下的 Windows 11, 使用 Intel i9-13900HX，24GB DDR5 运存。运行前保证 CPU 负载不超过 20%，同时内存余量大于 10GB。

## 3.2 实验使用的软件

在 Windows 系统中，C 语言程序使用 Powershell，使用下列代码编译和运行：

```
PS *path*> gcc -o dotproduct dotproduct.c
        或者 gcc -O3 dotproduct.c -o dotproduct
PS *path*> ./dotproduct
```

如果是较老版本 linux 系统，则需要在 gcc 后面加入 $-lrt$ 链接到实时库。

而对于反汇编 C 语言机器码到汇编代码，我们在同环境运行如下：

```
PS *path*> gcc -g -o dotproduct dotproduct.c
PS *path*> objdump -d dotproduct.exe > code.txt
```

随后我们就可以在同目录中 code.txt 查看机器码了。

Java 语言程序则在 Powershell 中，使用下列代码编译和运行，’X’ 为 0-5 的某个整数：

```
PS *path*> javac Dotproduct.java
PS *path*> java Dotproduct
       或者 java XX:TieredStopAtLevel='X' Dotproduct
```

在两亿乘两亿的数据量下不会产生 heap out of memory error 的问题，因此不用特意的调整 JVM 堆的大小。

观测软件方面，我们使用 Intel VTune Profiler，主要用于观察程序性能瓶颈以及采取各种优化手段后的 Bound 的改变。

### 3.3 实验数据 1—Java vs 无优化 C

如果没有很极限的性能需求下，其实我们一般不会特意的使用编译器的优化功能。这个实验数据的目的就是为了测试一下普通情况下，Java 和 C 的差距。下面五张图片分别代表了五种不同数据类型下，面对不同数据规模，Java 与 C 的运行速度变化曲线和对比：

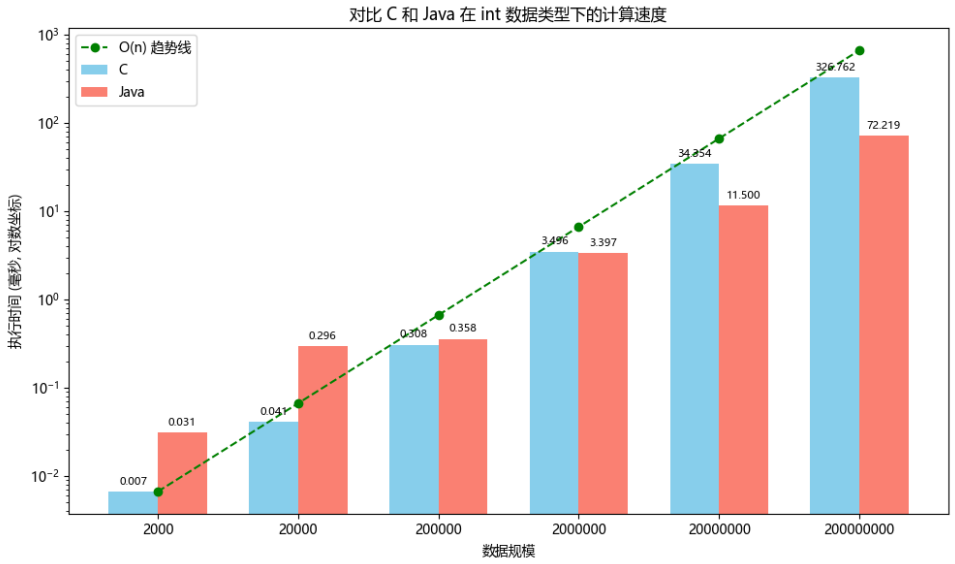



图 1: int 数据类型

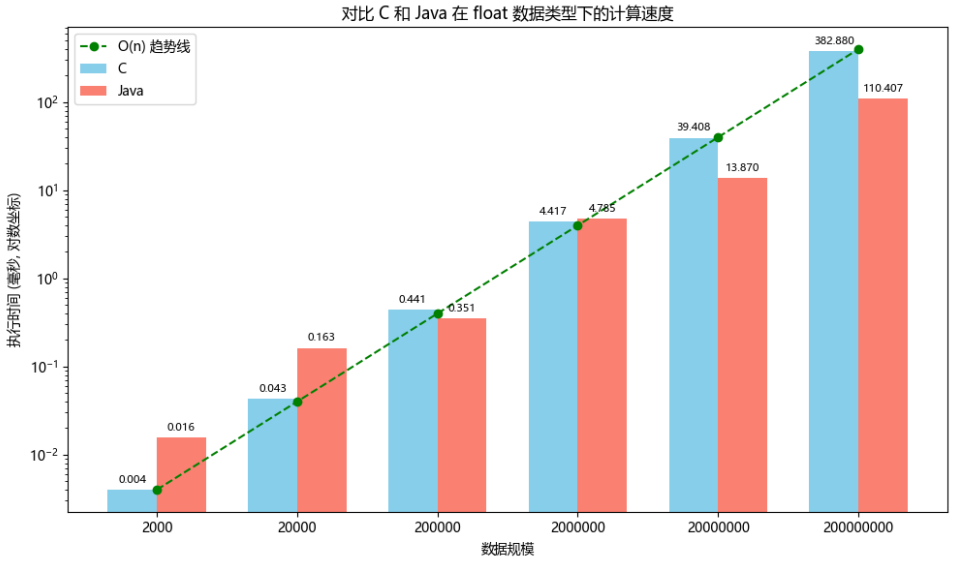



图 2: float 数据类型

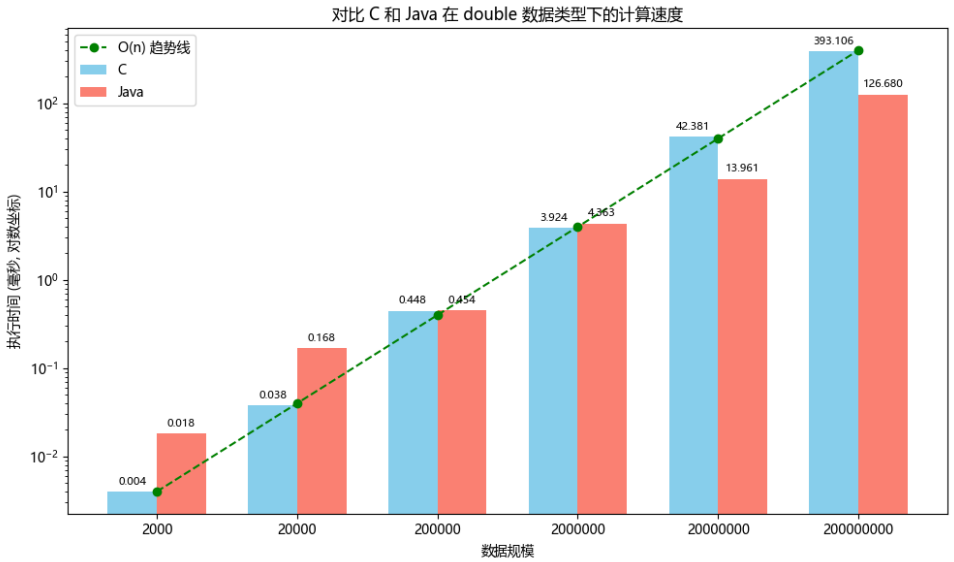



图 3: double 数据类型


图 4: char 数据类型

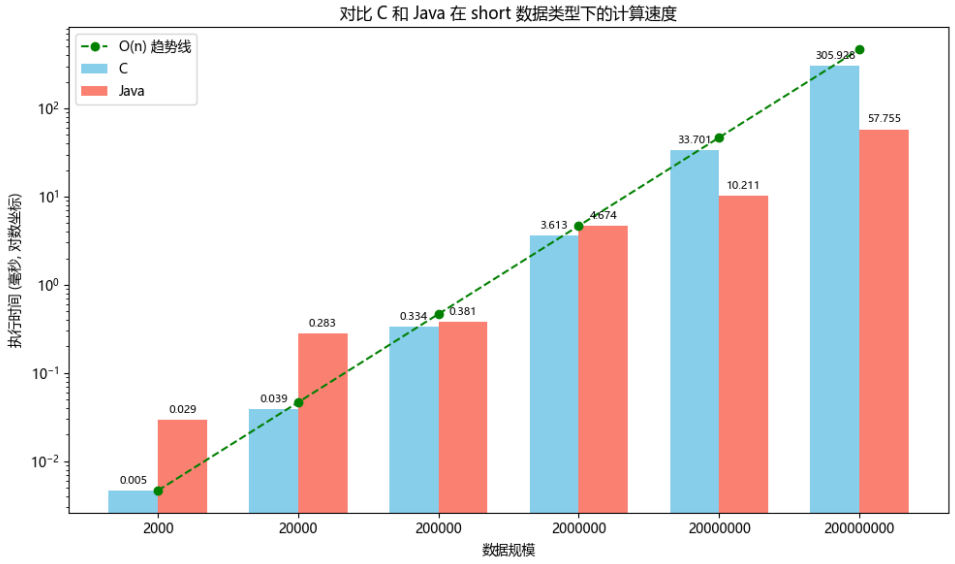



图 5: short 数据类型

我们发现了一些有趣的事实。首先对 C 程序来说，随着数据规模的扩大十倍，相应的消耗的时间也扩大成原来的十倍，图上也可以看出来呈现的是相对线性的关系，这符合我们的预期。计算线性拟合的 $R^2$ 发现对于所有的数据，$R^2 > 0.99$，非常接近完美的线性关系。同时对于 C，不同数据类型计算速度差不多是 $double \approx float > int \approx short \approx char$ ，数据类型的之间相差的速度其实比较小，可能也就 30%。造成差距的原因也很好解释，float 和 double 调用的都是 CPU 中的 Floating Point Unit，而整数类型调用的 ALU。为什么 char 比 int 从数据位数上来看少四倍，但是最终时间相似？现在 CPU 的 ALU 基本都是面向 32 位计算和 64 位计算，char 和 short 底层运算可能是和 int 同一套逻辑。就算不共享一套逻辑，32 位 ALU 一次计算可能也只用一个 clock，已经到速度的极限了。

但是 Java 程序就稍微有点反常了，首先，他的时间好像不是那么的线性。数据规模增大十倍，它的运行时间有的时候翻 10 倍左右，有的时候就只翻一两倍，肯定不是简单的线性关系。总体来说对不同的数据类型速度有 $double \approx float > short \approx char >= int$，数据类型之间的差距依旧比较小，原因和 C 有点相似。唯一不同的是这次 char 甚至比 int 还要慢一点，这是因为执行 short 和 char 的时候 JVM 会将他们提升到 int，这个转换过程还会消耗一点时间。

在一开始的时候，Java 是要缓慢比 C 慢不少的，但是随着数据规模的扩大，最后的速度比 C 快很多倍!

### 3.4 实验数据 2—Java vs O3 优化 C

**我无法接受 *Java* 居然会比 *C* 快的事实，这对我的世界观冲击太大了**。为了重振 C，在使用 gcc 的时候我们可以选择优化等级。分别有 O1/2/3/fast 选项，优化等级逐级提升而且逐级变得极端，到了 Ofast 甚至还会忽视一些 constraints。

在速度提升方面，真的会有效果吗? 我们来做个实验，这次我们采用 O3 优化，即常规优化的最高等级，并且直接采用 int 数据类型，因为其他的数据类型在上面的实验可以看出本质上是大同小异的:

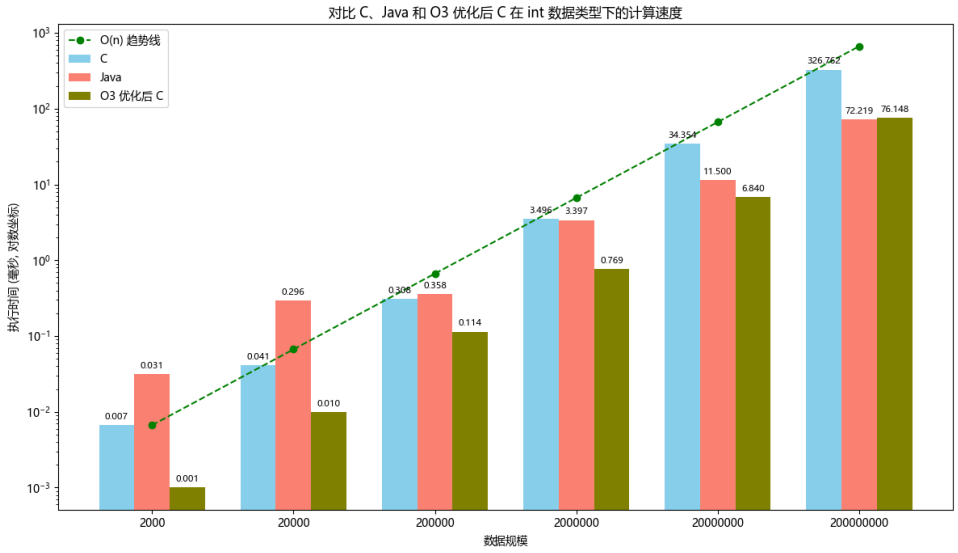



图 6: O3 优化, int 数据类型

O3 程序的运行时间增长依旧比较符合线性关系，说明 O3 优化没有对时间复杂度方面做出什么调整。但是惊人的是，在 O3 优化等级下，无论是生成随机数，还是乘积的速度直接提升了几倍! O3 编译器为什么可以让程序跑这么快? 同时，注意到数据量变得非常巨大的时候，Java 甚至以微弱的优势再次战胜了 O3 优化的 C，JVM 做了什么可以让程序跑这么快?

# 4 性能分析

## 4.1 编译器神力—O3 优化干了什么

### 4.1.1 逆向代码分析

我们用指令 objdump -d program 指令观察一下编译后的，未经优化和优化后的代码有什么区别（为了读取方便，我这里采用了简化版程序）：

在不进行 gcc 优化前的部分函数:

```
    操作1: XORShift最后一步，左移五位，取异或后返回函数值
401588: c1 e0 05              shl    \$0x5,%eax            ; 左移5位
40158b: 89 c2                 mov    %eax,%edx           ; 复制到EDX
40158d: 8b 05 7d 1a 00 00     mov    0x1a7d(%rip),%eax   ; 加载当前状态到EAX
401593: 31 d0                 xor    %edx,%eax           ; EAX = EAX ^
    EDX (第三次异或更新)
401595: 89 05 75 1a 00 00     mov    %eax,0x1a75(%rip)   ; 最终状态写回内存
40159b: 8b 05 6f 1a 00 00     mov    0x1a6f(%rip),%eax   ;
    将最终状态加载到EAX (作为返回值)
4015a1: c3                    retq                        ; 返回调用者
```

操作 2：为数组中某个变量进行赋值

```
4015e8: e8 63 ff ff ff        callq  401550 <xorshift32> ;
    调用生成随机数 (结果在EAX)
4015ed: 89 c1                 mov    %eax,%ecx           ; 保存随机数到ECX
4015ef: 8b 45 f4              mov    -0xc(%rbp),%eax    ; 加载i到EAX
4015f2: 48 98                 cltq                       ; 扩展为64位索引
4015f4: 48 8d 14 85 00 00 00  lea    0x0(,%rax,4),%rdx   ; RDX = i *
    4 (计算数组偏移)
4015fb: 00
4015fc: 48 8b 45 e8           mov    -0x18(%rbp),%rax    ; 加载数组1基地址
401600: 48 01 d0              add    %rdx,%rax            ; RAX = 数组1[i]的地址
401603: 89 ca                 mov    %ecx,%edx           ; EDX = 随机数
401605: 89 10                 mov    %edx,(%rax)         ; 数组1[i] = 随机数
```

操作 3：点积计算

```
401647: 8b 45 f0              mov    -0x10(%rbp),%eax   ; 加载j到EAX
40164a: 48 98                 cltq                       ; 扩展为64位索引
40164c: 48 8d 14 85 00 00 00  lea    0x0(,%rax,4),%rdx   ; RDX = j * 4
401654: 48 8b 45 e8           mov    -0x18(%rbp),%rax   ; 加载数组1基地址
401658: 48 01 d0              add    %rdx,%rax           ; RAX = 数组1[j]地址
40165b: 8b 10                 mov    (%rax),%edx         ; EDX = 数组1[j]的值
```

```
40165d: 8b 45 f0              mov    -0x10(%rbp),%eax   ; 加载j
401660: 48 98                 cltq                      ; 扩展为64位索引
401662: 48 8d 0c 85 00 00 00  lea    0x0(,%rax,4),%rcx   ; RCX = j * 4
40166a: 48 8b 45 e0           mov    -0x20(%rbp),%rax   ; 加载数组2基地址
40166e: 48 01 c8              add    %rcx,%rax          ; RAX = 数组2[j]地址
401671: 8b 00                 mov    (%rax),%eax        ; EAX = 数组2[j]的值

401673: 0f af c2              imul   %edx,%eax          ; EAX = EAX *
    EDX (有符号乘法)
401676: 48 98                 cltq                      ; 扩展乘积到64位
401678: 48 01 45 f8           add    %rax,-0x8(%rbp)    ; sum += 乘积
40167c: 83 45 f0 01           addl   \$0x1,-0x10(%rbp)  ; j++
```

嗯，没有经过优化的汇编代码其实就是 C 语言的翻版而已，其指令序列与 C 源码保持高度对应。让我们来看看经过 O3 优化后的代码长什么样。

指令：gcc -O3 x.c -o x，这里没有更进一步的采用-march=native 因为这个指令生成的机器码是根据当前处理器架构再优化的，不够通用。

操作 1：XORShift 最后一步，左移五位，取异或后返回函数值

```
401564: 89 d0                 mov    %edx,%eax          ; 转移结果到EAX
401566: c1 e0 05              shl    \$0x5,%eax          ;
    EAX左移5位（优化：合并操作顺序）
401569: 31 d0                 xor    %edx,%eax          ; EAX ^= EDX (第三次异或)
40156b: 89 05 9f 1a 00 00     mov    %eax,0x1a9f(%rip)  ; 更新全局状态
401571: c3                    retq                      ; 返回（删除冗余指令）
```

嗯，确实优化掉了几次 move 操作，而且删掉了最后写入内存的操作。

操作 2：为数组中某个变量进行赋值

```
4015c3: 89 c2                 mov    %eax,%edx          ; EDX = 当前状态
*省去详细： 左移13位 异或，右移17位 异或，左移5位 异或，复制结果到EDX*
4015d8: 89 c2                 mov    %eax,%edx          ; 复制结果到EDX
4015da: 89 04 8e              mov    %eax,(%rsi,%rcx,4) ; 存储到数组1[i]
```

我们惊人发现，其实程序根本没有调用 xorshift 函数！而是直接内嵌进循环中了，这样可以减少跳转的开支，在两亿的数据量下，这个小小的跳转优化确实可以节省巨量的时间。

操作 3：点积计算。在搬运代码前有个趣事，我在简略版的代码中进行点积运算后，下一步直接 free 掉两个内存空间然后 return 跑路了。然后，我在 O3 优化后的汇编代码中找了好久，没有找到点积计算这一步。原来是 O3 看我根本没有使用 sum，就干脆把乘积那个 for 循环直接删掉了。这是高效但危险的一步，高效在删除后确实没有影响运行结果，速度变快了；危险在在这种模式下会很难理解机器码，因为你根本不知道机器到底为了优化做了哪些极端的手段。

```
401630: f3 0f 6f 14 06        movdqu (%rsi,%rax,1),%xmm2 ; 加载数组1[i:i+3]到XMM2
401635: f3 0f 6f 1c 03        movdqu (%rbx,%rax,1),%xmm3 ; 加载数组2[i:i+3]到XMM3
40163a: 48 83 c0 10           add    $0x10,%rax          ;
    偏移量增加16字节（4个int）
40163e: 66 0f 6f ca           movdqa %xmm2,%xmm1         ; 复制XMM2到XMM1
401642: 66 0f 73 d2 20        psrlq  $0x20,%xmm2        ; XMM2右移32位（提取高32位）
401647: 48 39 c2              cmp    %rax,%rdx           ; 检查是否处理完所有向量块
40164a: 66 0f f4 cb           pmuludq %xmm3,%xmm1        ; XMM1 =
    低32位乘积（数组1[i]*数组2[i]）
40164e: 66 0f 73 d3 20        psrlq  $0x20,%xmm3        ; XMM3右移32位（提取高32位）
401653: 66 0f f4 d3           pmuludq %xmm3,%xmm2        ; XMM2 = 高32位乘积
401657: 66 0f 70 c1 08        pshufd $0x8,%xmm1,%xmm0   ; 重排结果：[低0,低2,0,0]
40165c: 66 0f 6f cd           movdqa %xmm5,%xmm1         ; 初始化符号扩展掩码
401660: 66 0f 70 d2 08        pshufd \$0x8,%xmm2,%xmm2   ; 重排结果：[高0,高2,0,0]
401665: 66 0f 62 c2           punpckldq %xmm2,%xmm0     ;
    合并低32位结果：[低0,高0,低2,高2]
401669: 66 0f 66 c8           pcmpgtd %xmm0,%xmm1        ;
    生成符号位掩码（处理有符号扩展）
40166d: 66 0f 6f d0           movdqa %xmm0,%xmm2         ; 复制乘积结果
401671: 66 0f 62 d1           punpckldq %xmm1,%xmm2     ; 低64位符号扩展
401675: 66 0f d4 e2           paddq  %xmm2,%xmm4         ; 累加到XMM4（低64位和）
401679: 66 0f 6a c1           punpckhdq %xmm1,%xmm0     ; 高64位符号扩展
40167d: 66 0f d4 e0           paddq  %xmm0,%xmm4         ; 累加到XMM4（高64位和）
401681: 75 ad                 jne    401630              ;
    继续循环直到处理完所有向量块
```

呃呃，一头雾水，编译器在干什么？怎么把一行就搞定的 for 循环弄的这么复杂？我也是这个学期在学计组，里面实在太多高级指令了，我让 AI 解释一下吧：

**1.SIMD 并行计算：使用宽寄存器批量加载 4 个整数，通过 PMULUDQ 指令同时执行多组乘法，将 4 次标量计算合并为 1 次向量运算。哦，是某种高级的向量化加速！**

**2. 数据重组与累加：通过位移/重排指令分离高低位乘积，利用 PADDQ 实现 64 位有符号累加，避免溢出。额，这一条好像和效率没啥关系。**

**3. 循环展开与剩余处理：主循环每次处理 4 个元素，剩余元素用标量指令补充，最大化利用指令级并行和内存带宽。嗯，一次读四个元素确实可以利用内存带宽。** 这确实体现了编译器极其强大的能力，我之前听说过什么编译器的优化水平秒杀 99.9% 的程序员，这么看可能还真不夸张。

#### 4.1.2 Profiler 分析

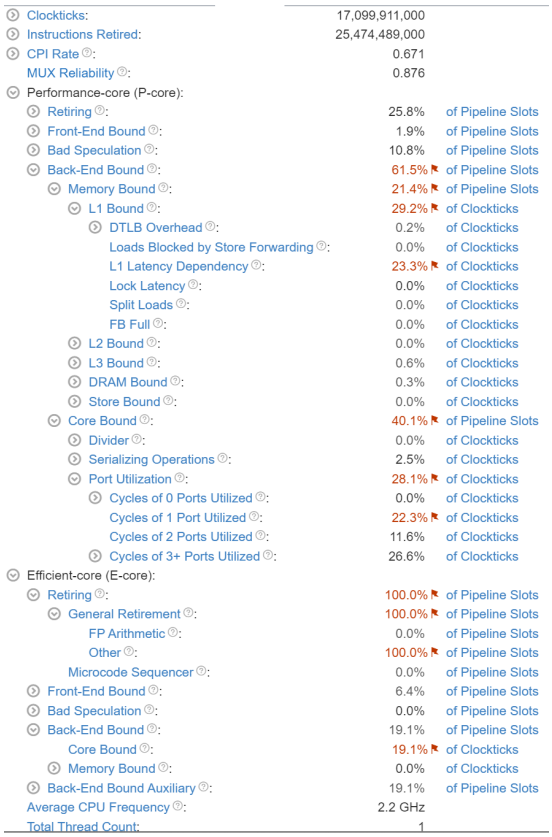
只看汇编可不够，我们继续使用 Intel VTune Profiler 工具深入的分析：究竟在运行的时候，这些优化究竟带来了什么。

首先，Inter VTune Profiler 将分析结果分成了 Performance-core 和 Efficient-core 两种结果。这张图中有几个重要指标，我们着重分析各种 Memory Bound，L1 Bound 等限制运行速度的数据。

我们知道：一个木桶能承多少水，由木桶最短的木板决定；一个程序能运行多块，由最慢的步骤决定。但是这个最慢的步骤是有理论下限的，例如 DRAM 的读写上限，CPU 核心运算单元的上限等，而这些 Bound 就告诉了我们究竟是哪个因素是当前程序的最短板。

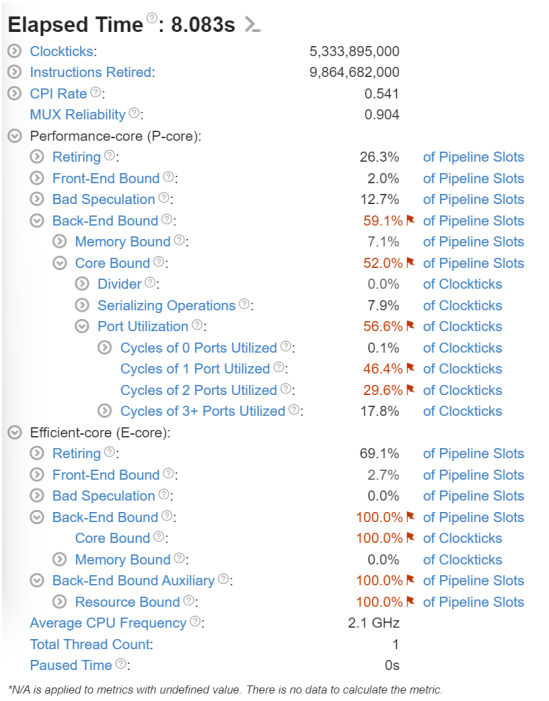
对于我的向量点乘程序来说，两个大小为 1 亿的向量总共需要占据大约 2GB 的内存空间。我的测试设备配备了双通道 DDR5 4800MT/s，理论吞吐上限为 76.8GB/s。也就是说，理论内存的下限为 26ms，程序不可能比这个程序更快。但是，我们注意到就算是 O3 优化的程序计算两个点乘也用了 76ms，**也就是我们可以说如果速度没有逼近 26ms 的话，不应该有很多的 Memory Bound**。同时，对于 i9-13900HX 来说，我没有找到 Intel 官方公布的 L1,L2,L3 Cache 的具体吞吐量数据。但一般来说，Cache 速度是要比内存快的。也就是说，L1 Cache Bound 也不应该很大，至少不应该比 Memory Bound 大。**如果 L1 Cache Bound 很大，说明程序在读取数据上面有很大的短板**。

这分别是由 Inter VTune Profile 分析出的未经优化、O3 优化的 C 程序计算 2*1 亿向量的报告：



| | | |
|---|---|---|
| Clockticks: | 17,099,911,000 | |
| Instructions Retired: | 25,474,489,000 | |
| CPI Rate: | 0.671 | |
| MUX Reliability: | 0.876 | |
| Performance-core (P-core): | | |
| Retiring: | 25.8% | of Pipeline Slots |
| Front-End Bound: | 1.9% | of Pipeline Slots |
| Bad Speculation: | 10.8% | of Pipeline Slots |
| Back-End Bound: | 61.5% | of Pipeline Slots |
| Memory Bound: | 21.4% | of Pipeline Slots |
| L1 Bound: | 29.2% | of Clockticks |
| DTLB Overhead: | 0.2% | of Clockticks |
| Loads Blocked by Store Forwarding: | 0.0% | of Clockticks |
| L1 Latency Dependency: | 23.3% | of Clockticks |
| Lock Latency: | 0.0% | of Clockticks |
| Split Loads: | 0.0% | of Clockticks |
| FB Full: | 0.0% | of Clockticks |
| L2 Bound: | 0.0% | of Clockticks |
| L3 Bound: | 0.6% | of Clockticks |
| DRAM Bound: | 0.3% | of Clockticks |
| Store Bound: | 0.0% | of Clockticks |
| Core Bound: | 40.1% | of Pipeline Slots |
| Divider: | 0.0% | of Clockticks |
| Serializing Operations: | 2.5% | of Clockticks |
| Port Utilization: | 28.1% | of Clockticks |
| Cycles of 0 Ports Utilized: | 0.0% | of Clockticks |
| Cycles of 1 Port Utilized: | 22.3% | of Clockticks |
| Cycles of 2 Ports Utilized: | 11.6% | of Clockticks |
| Cycles of 3+ Ports Utilized: | 26.6% | of Clockticks |
| Efficient-core (E-core): | | |
| Retiring: | 100.0% | of Pipeline Slots |
| General Retirement: | 100.0% | of Pipeline Slots |
| FP Arithmetic: | 0.0% | of Pipeline Slots |
| Other: | 100.0% | of Pipeline Slots |
| Microcode Sequencer: | 0.0% | of Pipeline Slots |
| Front-End Bound: | 6.4% | of Pipeline Slots |
| Bad Speculation: | 0.0% | of Pipeline Slots |
| Back-End Bound: | 19.1% | of Pipeline Slots |
| Core Bound: | 19.1% | of Clockticks |
| Memory Bound: | 0.0% | of Clockticks |
| Back-End Bound Auxiliary: | 19.1% | of Pipeline Slots |
| Average CPU Frequency: | 2.2 GHz | |
| Total Thread Count: | 1 | |

图 7: 未经过优化



**Elapsed Time: 8.083s**

| | | |
|---|---|---|
| Clockticks: | 5,333,895,000 | |
| Instructions Retired: | 9,864,682,000 | |
| CPI Rate: | 0.541 | |
| MUX Reliability: | 0.904 | |
| Performance-core (P-core): | | |
| Retiring: | 26.3% | of Pipeline Slots |
| Front-End Bound: | 2.0% | of Pipeline Slots |
| Bad Speculation: | 12.7% | of Pipeline Slots |
| Back-End Bound: | 59.1% | of Pipeline Slots |
| Memory Bound: | 7.1% | of Pipeline Slots |
| Core Bound: | 52.0% | of Pipeline Slots |
| Divider: | 0.0% | of Clockticks |
| Serializing Operations: | 7.9% | of Clockticks |
| Port Utilization: | 56.6% | of Clockticks |
| Cycles of 0 Ports Utilized: | 0.1% | of Clockticks |
| Cycles of 1 Port Utilized: | 46.4% | of Clockticks |
| Cycles of 2 Ports Utilized: | 29.6% | of Clockticks |
| Cycles of 3+ Ports Utilized: | 17.8% | of Clockticks |
| Efficient-core (E-core): | | |
| Retiring: | 69.1% | of Pipeline Slots |
| Front-End Bound: | 2.7% | of Pipeline Slots |
| Bad Speculation: | 0.0% | of Pipeline Slots |
| Back-End Bound: | 100.0% | of Pipeline Slots |
| Core Bound: | 100.0% | of Clockticks |
| Memory Bound: | 0.0% | of Clockticks |
| Back-End Bound Auxiliary: | 100.0% | of Pipeline Slots |
| Resource Bound: | 100.0% | of Pipeline Slots |
| Average CPU Frequency: | 2.1 GHz | |
| Total Thread Count: | 1 | |
| Paused Time: | 0s | |

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

图 8: O3 优化

我们可以看到，对于没有经过优化的程序，最大的三个 Bound 分别为：

$$CodeBound : 40.1\%, L1Bound : 29.2\%, MemoryBound : 21.4\%$$

说明未优化的代码在数据访问（尤其是 L1 缓存）和指令并行调度方面表现糟糕，CPU 大量周期在等待数据。这反映出来的问题是内存没有发挥其带宽优势，CPU 也没有被高效的利用。

而经过 O3 优化后的程序，这三个 Bound 分别变成：

$$MemoryBound7.1\%, L1Bound : 6.9\%, CodeBound : 52.0\%$$

Code Bound 变大了，说明核心瓶颈更多在算术执行段，我们充分的利用了 CPU 某一个核心的运算能力，这是我们期望的结果。这也验证了上面机器码中，O3 优化一次加载 4 个元素的优化是实打实的提升了程序速度的。

同时，我们注意到上述汇编将 4 次标量计算合并成了 1 次向量计算，以及直接砍掉调用 XORShift 函数直接将其移植到循环内部的优化。这样做的目的肯定是为了减少整体的 Instruction 数。同样是计算两个向量的点乘，未经优化的使用了 254 亿条指令，而 O3 优化后的只使用了 98 亿条，这也是一个非常大的优化。

但有点可惜的是，两个程序的 Total Thread Count 都是 1，GCC 没有帮我进行多线程优化，如果有了多线程优化，程序速度还会翻上好几倍。

## 4.2 JVM 神力—底层干了什么

### 4.2.1 Java 越跑越快，真的假的？

JVM 是一个非常神奇的东西，其中一个神奇的特性就是 Java 程序越执行越快。

在我们上面对比运行速度的时候，可以发现，java 从一个数据规模跳到下一个数据规模的时候，不像 C 比较稳定的翻十倍。而是偶尔翻十倍，偶尔翻三四倍。口说无凭，我们依旧来做一下实验，这次我们用 for 循环五次，计算两个一百万 int 向量点乘，五次实验取平均值，看看速度的变化：

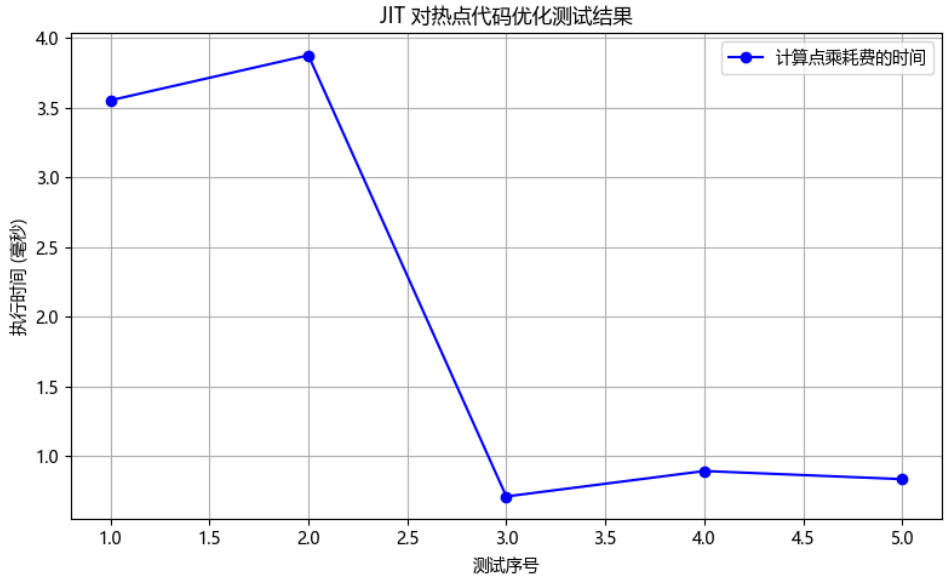



图 9: 相同 Java 代码块运行时间

可以看到，这里在第二次到第三次的转换中，明明都是计算同样的数据量，速度却加快了五倍多。Java 真的越跑越快了，这到底是什么原因呢?

### 4.2.2 优化 1：JIT 热点代码优化

Java 能够越跑越快的一个非常关键的要素就是 JIT 对热点代码的优化。什么是 JIT? 全称是 Just-In-Time Compiler，是 JVM 中自带的一部分。JIT 通过方法调用计数器和 loop-back-edge 计数器，统计代码中某一块代码的运行次数。假如说某一段代码的运行次数达到了一个 JIT 的阈值，JIT 就会针对热点代码进行编译和优化。

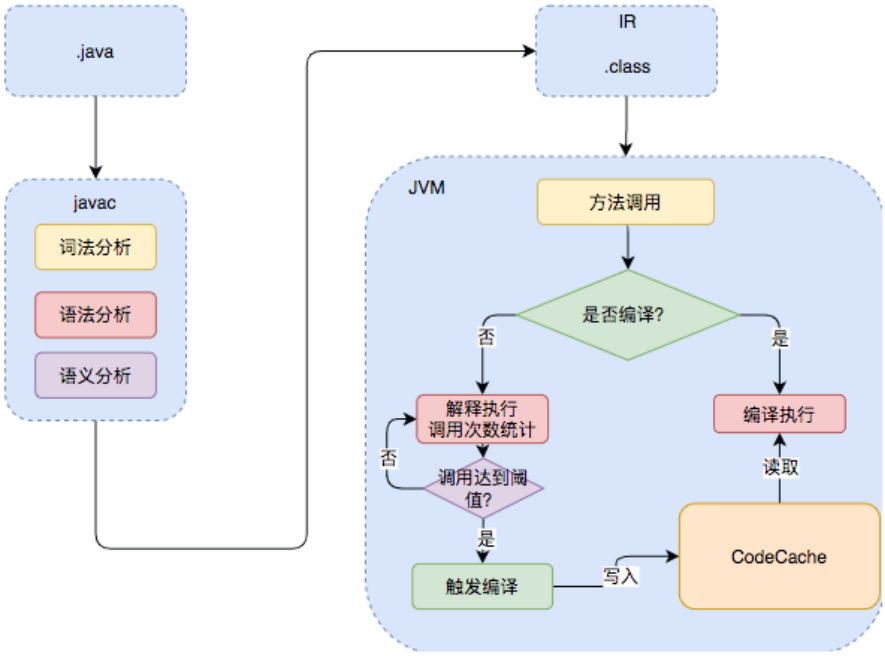



图 10: Java 程序运行结构

Java 虚拟机中，存在功能不同的编译器。其中 Client Compiler 的设计目的是为了快速响应，用较低的编译开销和短的响应时间满足轻量级的应用。而 Server Compiler 的设计目的是为了性能，用更长的编译时间换取更高的优化效果。**类比来说，其实也就是 GCC 中开不开优化的区别**。而具体 Server Compiler 的编译优化过程，则与 GCC 不同，比较抽象的用了某种控制流和数据流结合的图数据结构。在 JDK9 后 Server Compiler 甚至还有两种，这里由于篇幅原因就不详细展开。

但是又与 C 不同的是，在上图 JVM 其实是使用了叫 Tiered Compilation 的策略。这种策略先使用 Client Compiler 快速响应生成机器码，使程序能够快速响应。然后 JIT 监控方法的运行情况后，使用 Server Compiler 编译热点代码进一步的优化性能。这是一个非常聪明的策略，他可以同时保证程序响应快和程序效率高。

同时，Java 编译器也允许从一开始就使用 $-XX:TieredStopAtLevel=X$ 指令实现类似 GCC -Ox 的编译层面优化。X 数字越大，代表优化等级越高。X=0 时候，代表程序通过解释器逐行解释执行，这样的好处就是响应快，坏处是性能差。当 X=5 的时候是直接使用 Server Complier。我们分别采用不同 X 进行性能测试，看看优化程序究竟如何：

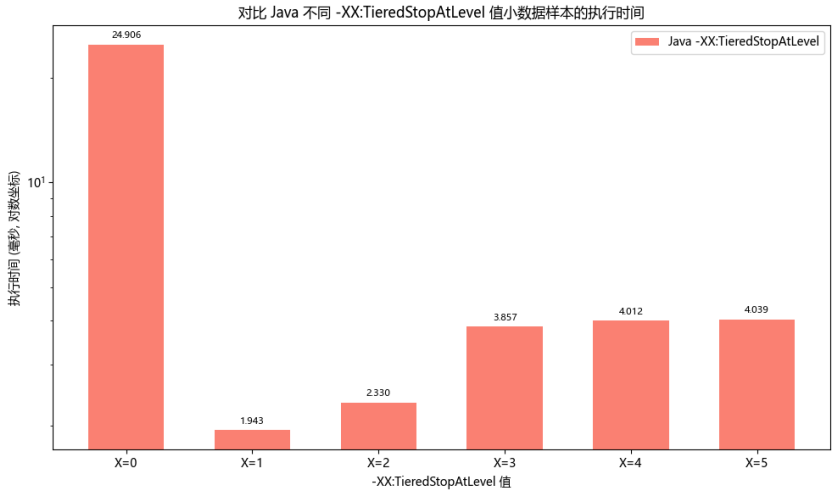



图 11: 数据量为 100w 时，不同优化等级运行时间

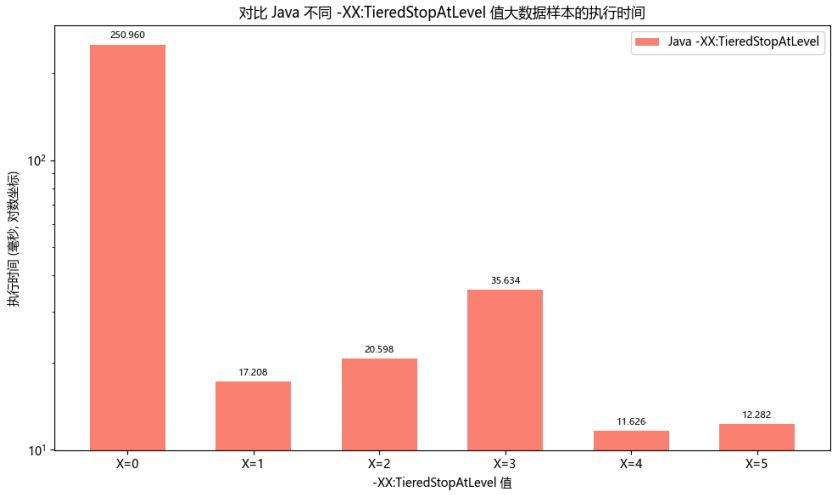



图 12: 数据量为 1000w 时，不同优化等级运行时间

可以看到，只要采用了优化，速度都比不采用优化快上好多倍。但是最高级的优化反而在数据量小 (2*100W) 的时候占据了下风，原因就是因为 Server Compiler 的响应速度并没有 Client Complier 快，在数据量小的时候还不如不优化。但一旦数据量大，运行时长显著长于响应时长的时候，Server Compiler 的速度优势就逐渐显示出来了。

#### 4.2.3 优化 2：调度优化

搜索资料得知，JVM 不会自动的把一个单线程任务改变成多线程，除此之外会不会有其他优化呢？这里，我们继续使用 Intel VTune Profiler，用 Java VisualVM 捕获运行 JVM 进程的 pid 后，使用 Profiler 的 Attach to Process 功能捕获 JVM 的运行数据。由于捕获 JVM 的结果其实是并不是程序运行本身，而是套壳虚拟机后的结果，最终可能不能特别代表 JIT 优化本身，但也能获取不少数据。

这里，我们捕获两个计算：计算 100w 和计算 1000w（因为上图我们发现，JIT 的优化大约在两百万次数左右开始起效），下面是我们的捕获结果：

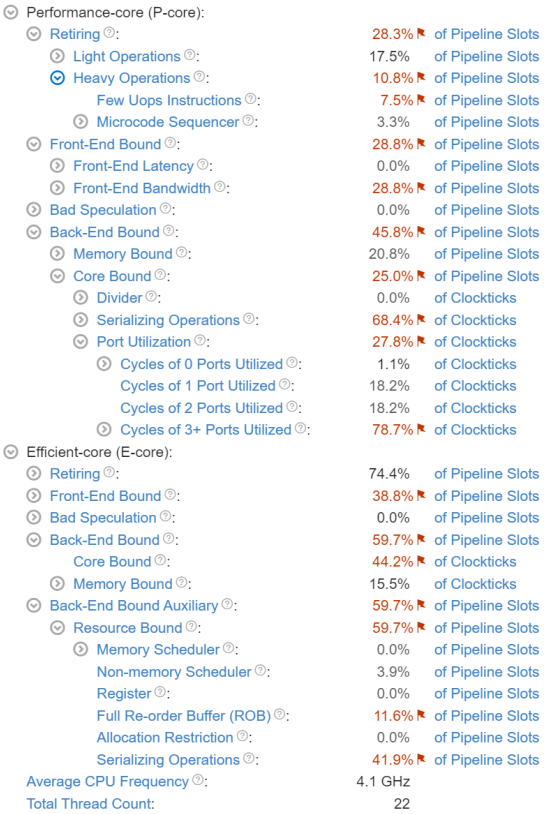



图 13: 数据量为 100w 时，Profiler 报告

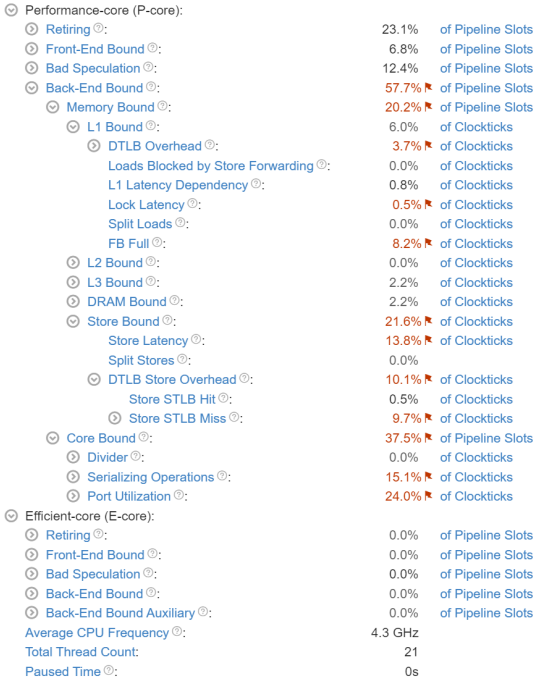



图 14: 数据量为 1000w 时，Profiler 报告

首先我们发现了个比较巨大的改变，当数据量大的时候，E 核直接不工作了！我一开始以为是测试有误，然后连续测试了几次之后，E 核依旧不工作。我不太清楚这是系统层面的调度还是 JVM 层面的优化。但这确实我们发现了 JVM 在 JIT 以外的优化。同时我们观察到，当数据量大的时候，Code Bound 的确从 25.0% 提升到了 37.5%，这是我们希望看到的结果，也说明了 JIT 对热点代码的优化确实提升了 CPU 的使用率。

### 4.3 C 真的比 Java 快吗

C 比 Java 快，对也不对。我们发现，O3 优化在数据规模为 2 亿的情况下被 Java 以微小的差距反超了。这其实并不代表 Java 语言运行的就比 C 更加快。事实上根据 Profiler 给出的 Instruction 量来看，C 语言的程序使用的 Instruction 数是 98 亿，而 Java 是 46 亿。说明了但从单指令速度来看，C 语言还是要更快的，只不过 JVM 内部可能采用的优化策略更加激进。所以说，我们不能一棒子敲死 Java 慢，毕竟现在 JVM 确实已经进化到了一个非常智能的水平了。

# 5 AI 辅助了哪些工作

在这次 Project 我依旧按照惯例，简单阐述一下我在这个 Project 中在哪里让 AI 参与了我的工作。

## 5.1 将 C 转换成 Java

我在一开始按照大概需求用 C 语言写好了测试程序，但是在测试的时候意识到还要将同样的逻辑用 Java 写一遍，非常没有意义而且麻烦。但是没有关系，Copilot Cookbook 恰巧给出了对应 prompt:"Tell me how I can convert this C Program to Java. The functionality and output of the new script should be exactly the same as the existing one."。然后将程序丢给 Copilot，1 分钟结束，转换结果无可挑剔。

## 5.2 Python 统计数据程序生成

上述所有统计图，和 T-test 均是由 Copilot 内置的 4-o 生成的 Python 程序绘制。我使用的 prompt 为:

```
我现在需要绘制一个条形图，目的是对比int数据类型下C和Java计算速度。
数据大小...转换成毫秒再画图，不要预设数据，我要现场命令行输入的数据。
画一条线以示意这个算法的时间复杂度符合O(n)。时间轴不要用线性轴，差距有数万倍。
...
现在，需求稍微变化了一点，要求再增加一个O3优化后的C速度对比。
现在需求稍微改变了，我现在在测试Java不同'-XX:TieredStopAtLevel的速度...
```

我用极短的时间就获得了所有正确的绘图 Python，并且这个流程已经深度嵌入我的其他工作流程。

## 5.3 解释汇编代码

上述逆向 exe 文件后获取的汇编码后面的中文注释，全部由 DeepSeek R1 生成。流程如下:

```
逐行的解释这个汇编码： *代码*
不要偷懒，逐行解释每一行。
你做的很好，现在按照上面的规格再对这个经过O3级别优化的汇编码解释:
那int_dot_product的乘积循环呢？（其实这里被O3优化掉了，R1精准的指了出来）
我修改了一下源码，现在有了。再次按照上面的规格对这个汇编码进行逐行解读。
展开讲讲编译器究竟是怎么使用SIMD指令提升计算性能的
详细展开讲讲编译器是怎么加快这个向量点积运算的，我需要技术细节
用大约两三句话核心的概括这个的优化逻辑或方法（生成内容为上述点积优化解释）
```

就算计算机组成原理学的再好，O3 优化后的汇编代码也并非正常人能轻易读懂。这里我们合情合理的运用 AI 解读汇编代码，事实上他确实做的非常漂亮，加了中文注释后我们的确能够轻松理解 O3 在干什么了。

### 5.4 Latex 快速模仿

介于我已经有了现成的 Project1 的模版，这次 Project 的部分 latex 部分都是直接要求 AI 模仿生成。例如下面的 Reference 部分，我直接把新链接和之前的某个例子丢给 AI 帮我快速生成 latex 源码，就不用麻烦照着之前的引用逐行修改了。

## 6 实验总结

本实验通过向量点乘为手段，详细分析了 Java 与 C 语言在不同数据类型和数据规模下的运行速度。我们发现，GCC 确实可以在编译层面极大地加速 C 语言运行的速度，我们通过解读机器码和分析运行瓶颈验证了这个事实。以及 JVM 现在的优化机制已经非常强大了，强大到在数据规模大的情况下已经可以超越 O3 优化后的 C 程序。但是，Java 的 instructions per second 数确实较 C 慢上不少。**因此，我们不能一概而论的说哪个编程语言更快**。在不同的场景和优化级别下，Java 可能会比 C 快上几倍，或是慢上几倍。

在实验的过程中，的确出现了很多冲击我之前认知的东西，也被现代编译器的优化能力和 JVM 其貌不扬的实力所震撼。总的来说，学到的东西还算不少，如果时间能够给的再宽裕一点，我应该还会再深入研究一下 JVM 更加内部的一些优化实现原理，~~两周的时间确实有点短（逃）~~。同时建议对这方面有兴趣的去看看赖海斌同学的 report（见 Reference），他在这个基础之上还增加了跨平台的测试，也比较有意思。

**感谢各位能读到这里**。

# References

# 参考文献

[1] Brighton *All the original experiment data including in this report*. Southern University of Science and Technology, 2025. [Source code]. Available: github.com/BrightonXX/SUSTech-CPP-Project

[2] *IEEE Standard for Floating-Point Arithmetic (IEEE 754)*. the Institute of Electrical and Electronics Engineers, 1985. [Source code]. Available: wikipedia.org/IEEE_754

[3] Lai H.B.*CS205 Project2: Simple Matrix Multiplication*. Southern University of Science and Technology, 2024. [Source code]. Available: github.com/HaibinLai/CS205_Project2

[4] GitHub. *Copilot Chat Cookbook*. Documentation, 2025. Available: docs.github.com/copilot-cookbook

[5] Ben Evans. *Understanding Java JIT Compilation with JITWatch, Part 1*. Technical Article, Oracle, July 2014. Available: oracle.com/technical-resources/articles/java/architect-evans-pt1.html

[6] Javabetter.cn. *JVM JIT 编译器解析*. Technical Article, Javabetter.cn, n.d. Available: javabetter.cn/jvm/jit.htmljvm-的编译器