CS205 · C/C++ Programming

Project2 Report: Matrix Multiplication

赖海斌
12211612

摘要：同样是矩阵乘法，Java 和 C 谁更快？在做 Project 之前，我会凭着经验和对于老师的信任大声告诉你答案。但是在做了本次 Project 后，我只能笑着不能告诉你答案。我一共做了四个实验，第一个实验发现 Java 在执行时会自动判定 float 精度，而 C 不会。我进而认为 Java 程序的 JVM 对 Java 影响很大，进而去探索了 JIT 对 Java 影响和被翻译成汇编的 Java 和 C 以及 C -O3 的 x86 汇编代码，发现 JIT 的 C1C2 编译器效果不明显，而-O3 编译优化将原本在栈上的 ijk 变量优化到放入寄存器中，尝试减少程序的汇编指令数以及减少花费高昂的内存读取指令的次数，因而降低了运行时间。随后实验 3 我进行了速度测试，发现在不同平台上 Java 和 C 的表现有所差异，在软硬件环境和编译器不同的情况下 Java 还真不一定比 C 慢。同时我也探测到 Java 多线程和 C OpenMP 在通信上的瓶颈，多线程并不一定比单线程强。实验 4 我探索了 Java 和 C 不同的启动方式以及 Linux 和 Windows 操作系统唤起程序的方式。同时我还用 Intel TopDown 性能分析法分析了 Java 和 C 程序的运行瓶颈，分析了程序 CPI 变化的原因。

在硬件、软件上的众多区别，使得我们的实验就像进入到《黑客帝国》里的 Matrix，复杂的环境下一切皆有可能发生。但是通过这次的理论和实验，作为 Mr.Anderson，我们离 Neo 更近了一步。

关键词：汇编分析；速度测试；微处理探索；Top Down 性能分析;

目录

Part 1: 实验需求分析

矩阵相乘是一个重要的计算机基础运算，这次 Project 要求我们计算矩阵相乘，并对 Java 和 C 语言的实现进行比较。我想老师希望我们从中学习的，是 **C 为什么性能高（Why）。**同时我也好奇，**C 如何提高性能(How)**，单纯的上 OpenMP 或者 CUDA 效果有多大？

所以，本次实验，我想建立这么几个观测目标：

1. **时间损耗**：这里边可能存在的问题是，1. 用 Profiler 统一测量时间好，还是在各自语言内写函数记录？2. 参考数据库 Project，JIT 是否会影响时间？3. 单核跟多核的性能是否会影响时间？

2. **性能跟踪**：这里我们可以引入并行计算里的理论来测量。可能的问题是，1. C 和 Java 程序分别是怎么跑起来的？2. 虚拟机与内部 GC 怎么影响我们的程序？3. 不同的处理器是否会使性能发生改变？

3. **运行加速**：1. Java 的多线程能够提升几倍？OpenMP 呢？为什么他们的效率不一样？GPU 提升的效率是怎么样的？2. 使用-O3,-O2 真的会使我们的程序加速吗，为什么？

4. **算法升级**：矩阵乘法真的只能在 $O(n^3)$实现吗？有没有更快的方法？他们应用的多吗？这部分只是针对这次 Project 的程序，也是给后面在优化上让我有更多的了解。

Part 2: 实验设备与仪器

本节介绍我们的实验“试管”和“试纸”——硬件设备和观测软件。

我们本次使用两个试管：一台 Linux 浪潮服务器和我的 Windows 华为笔记本电脑。

2.1 实验仪器 1

本次实验的仪器 1——“试管 1 号”服务器，我们使用 AMD Processor 作为我们的 CPU。感谢肖翊成和邱俊杰同学在装机和配置系统上的帮助。

CPU Architecture	x86_64
Model name	AMD EPYC 7773X 64-Core Processor
CPU family	25
Thread(s) per core	1
CPU(s)	128
Core(s) per socket	64
Memory	185GiB
Cache	1.5GiB
CPU Frequency	3.5GHz

表 2.1 “试管 1 号”硬件 CPU

2.2 实验仪器 2

为了方便运行 Java 和查看运行程序，我们的实验仪器 2——“试管 2 号”电脑就直接是我的笔记本。试管 1 号的单核 CPU 要比试管 2 号优秀，我们会在后面的效率比较实验（实验 3）时完全在试管 1 号上运行。在实验 2 和 4，我们会比较我们的程序在汇编层级的代码，此时我们将使用 Intel Vtune Profiler，因此我们将在试管 2 号上进行实验。

CPU Architecture	x86_64
CPU name	11th Gen Intel(R)Core i7-11370H @ 3.30GHZ
CPU family	6
Thread(s) per core	2
CPU(s)	8
Core(s) per socket	1
Memory	15.8GiB
Cache	12MiB

表 2.3 “试管 2 号”硬件 CPU

2.3 实验软件版本

本次主要实验软件为 Java jdk 和 C 编译器，具体版本如下表。

	试管 1 号	试管 2 号
Java jdk	openjdk 17.0.8 2023-07-18 LTS	openjdk 17.0.10 2024-01-16
gcc	gcc.exe (tdm64-1) 10.3.0	Ubuntu 13.2.0-4ubuntu3
nvcc	-	V12.4.99

表 2.4 实验软件版本

2.4 测量时间程序

在 Java 中，我们采用系统的 nanoTime 来记录时间，之后通过减法输出程序使用时间。而在 C 中，我们采用<time.h>里的 clock()函数，OpenMP 中 omp_get_wtime()记录当前时间。

组图 2.4.1 Java 与 C 测量时间程序

2.5 观测软件

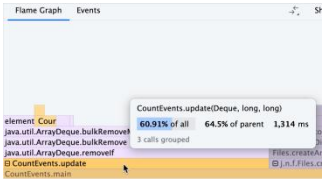
1. Intel® VTune™ Profiler

Intel VTune Profiler 是一个全平台性能分析工具，可以多角度分析程序性能瓶颈。我们将对我们的 C 程序分析，观察其在系统内的运行情况。

2. NVIDIA Nsight™ Compute

NVIDIA Nsight™ Compute 是 CUDA® 的交互式分析器。我们将在 CUDA C 程序上使用它，分析 GPU 上的 C 代码到底是怎么执行地，跟我们的 CPU 有什么区别。

3. IntelliJ Profiler



IDEA 的 Java 性能分析器。考虑到本次我们对 Java 关心的是 JVM 底层方面对性能的影响，我们用这个 Profiler 对 Java 程序的启动与终止进行分析。

Part 3: 矩阵乘法程序设计

在本次实验里程序结构如下，我们使用简单的 Python 矩阵生成器利用随机数生成矩阵，随后程序读入并运算，记录结果。我们将查看程序结果和运行时间，并分类讨论：如 read 和 write 的时间代表着程序在内存与存储间的 I/O 速度，对全部使用时间我们会分析程序的启动与终止过程。**（程序设计读写及框架由 GPT 生成）**

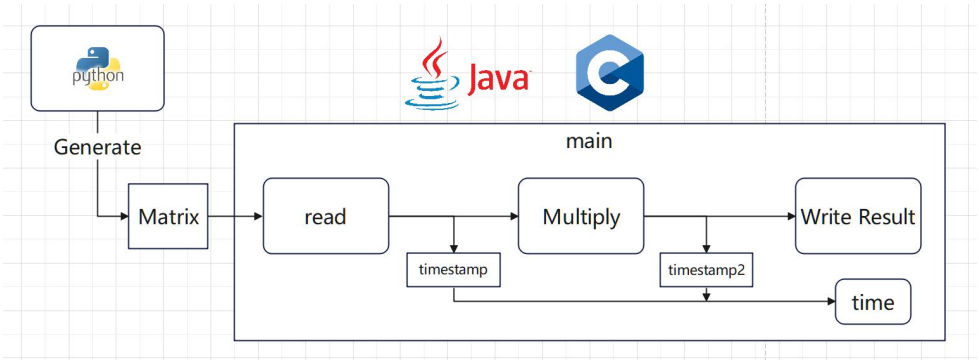


图 3.1 Java 与 C 程序架构

下面的函数是 Java 的读取和写矩阵函数。

```java
public static float[][] readMatrixFromFile(String filename, int n) {
    float[][] matrix = new float[n][n];

    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line;
        int row = 0;
        while ((line = br.readLine()) != null && row < n) {
            String[] values = line.trim().split( regex: "\\s+");
            for (int col = 0; col < n; col++) {
                matrix[row][col] = Float.parseFloat(values[col]);
            }
            row++;
        }
    } catch (IOException e) {
        System.out.println(e.getLocalizedMessage());
    }

    return matrix;
}
```

```java
public static void writeMatrix(float[][] matrix,String filePath) throws IOException {
    File file = new File(filePath);
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(file))) {
        for (float[] row : matrix) {
            for (float value : row) {
                bw.write(String.valueOf(value));
                bw.write( str: " ");
            }
            bw.newLine();
        }
    }
}
```

下面是 C 函数的读取和写函数。

```c
void readMatrixFromFile(char *fileName, float matrix[SIZE][SIZE]) {
    FILE *file = fopen(fileName, "r");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return;
    }

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            fscanf(file, "%f", &matrix[i][j]);
        }
    }

    fclose(file);
}
```

```c
void writeMatrixToFile(char *fileName, float matrix[SIZE][SIZE]) {
    FILE *file = fopen(fileName, "w");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return;
    }

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            fprintf(file, "%f ", matrix[i][j]);
        }
        fprintf(file, "\n");
    }

    fclose(file);
}
```

而我们对 Java 和 C 各设计了多种乘法程序，以供实验。具体如下图。

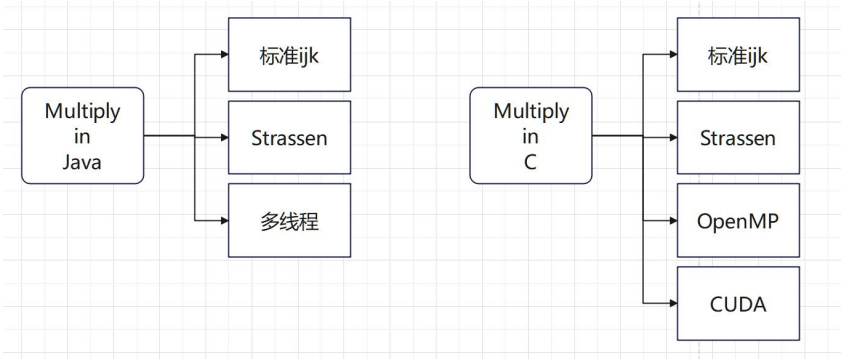


图 3.2 Java 与 C 矩阵乘法实现

首先是计算复杂度 $O(n^3)$ 的经典矩阵乘法，它主要有三个 for 循环，我称其为标准 ijk 算法。

```java
public static float[][] multiplyMatrices(float[][] matrix1, float[][] matrix2, int n) {
    float[][] result = new float[n][n];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    return result;
}
```

```c
#define SIZE 100

void matrixMultiplication(float matrix1[SIZE][SIZE], float matrix2[SIZE][SIZE],
                          float result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

图 3.3 Java 与 C 标准 ijk 矩阵乘法实现

Strassen 算法是一种特别的矩阵乘法，它通过多进行一些加法从而使矩阵乘法复杂度降低到 $O(n^{\log 7})$。但是算法设计还不是我们实验的主要内容。~~它到底性能如何，可能还要给读者留作练习。~~

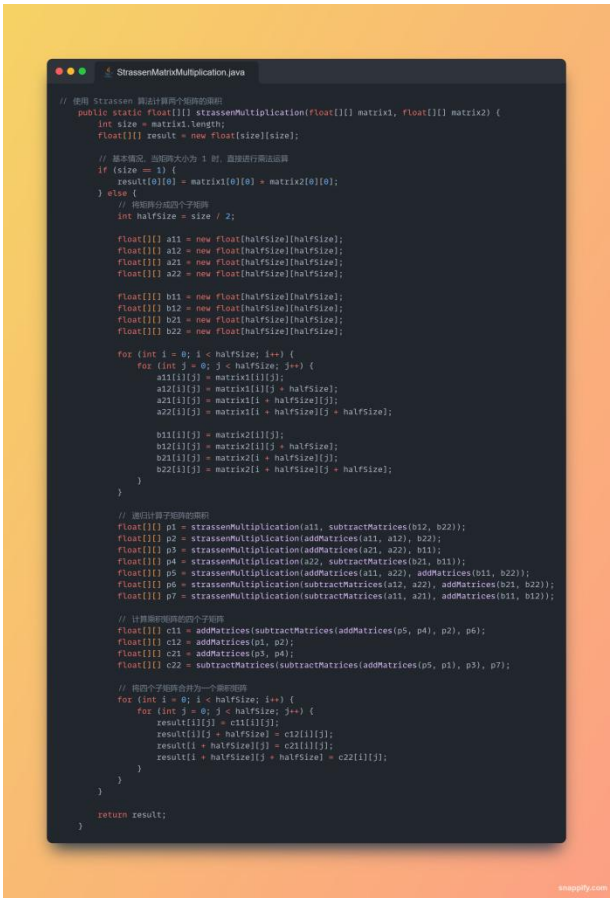


图 3.4 Java 与 C 的 Strassen 乘法实现

接着我们实现基于 Java 的 MultiThread 和 C 的 OpenMP 的矩阵乘法。我们将比较两种语言在多线程并行计算下的性能差异。另外，在 OpenMP 实现中，我应用了 SIMD 技术加速我们的程序。

```java
public MatrixMultiplicationTask(float[][] matrix1, float[][] matrix2, float[][] result, int startRow, int endRow) {
    this.matrix1 = matrix1;
    this.matrix2 = matrix2;
    this.result = result;
    this.startRow = startRow;
    this.endRow = endRow;
}

@Override
public void run() {
    int size = matrix1.length;

    for (int i = startRow; i < endRow; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

```c
void matrixMultiplicationInOpenMP(float *matrix1, float *matrix2,
                                  float *result, int size) {
    #pragma omp parallel for collapse(2) // 将两个内层循环并行化，同时减少显式的线程同步
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i * size + j] = 0;
            #pragma omp simd // 单指令多数据流，使矩阵加法操作在SIMD指令集上并行化
            for (int k = 0; k < size; k++) {
                result[i * size + j] += matrix1[i * size + k] * matrix2[k * size + j];
            }
        }
    }
}
```

图 3.5 Java 多线程与 C OpenMP 乘法实现

最后针对 CUDA，我们特地设计了矩阵乘法 CUDA 版本。然而很不幸的是，我们的 CUDA Driver 没能成功安装，所以最后我们没有实验 GPU 版本乘法。~~（我们将把这个留给读者当作练习）~~

```
__global__ void matrixMultiplication(float* A, float* B, float* C)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float value = 0.0;

    for (int i = 0; i < N; i++) {
        value += A[row * N + i] * B[i * N + col];
    }

    C[row * N + col] = value;
}
```

图 3.6 CUDA C 矩阵乘法的 CUDA 核函数实现

Part 4: 实验设计

本次实验分为 4 个子实验：

实验 1：运算正确性检验

首先我想确定的是，我们的算法计算出来的结果是否都是一致的，从而验证程序的运算有效性。这听起来比较基础，但是我随后发现，Java 和 C 运算出来的结果其实并不完全一致。我们将就此给出说明。

我们将选取两个 100 x 100 数值区间[0,2]的矩阵进行运算，投入我们跟 GPT 同志一起设计的算法，**比较计算结果**。

	Java	C
选用算法	标准 ijk，多线程	标准 ijk，C OpenMP
选用平台	试管 1 号，试管 2 号	试管 1 号，试管 2 号

问题规模 N	矩阵数值区间
100	[0,2]

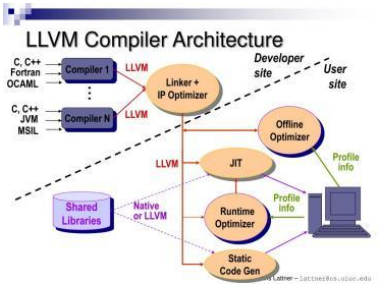
实验 2：编译优化实验

上学期数据库的 Project 经验告诉我们[5]，Java 会有基于 LLVM 的 JIT 编译引擎，它会标记热点代码并且对热方法编译成机器码，让我们的 Java 运行速度越来越快。而 C 的编译器也有类似的编译优化，像-O2 -O3 会对 C 编译出的机器码进行优化，比如调整分支语句、合并常量。他们的优化效果如何？我们将用**内置的时间记录**监测。

同样我们将选取两个 100 x 100 数值区间[0,2]的矩阵进行对比，查看计算结果。

问题规模 N	矩阵数值区间

100，1000	[0,2]



	Java	C
选用算法	标准 ijk	标准 ijk
选用平台	试管 2 号	试管 2 号

实验 3：矩阵相乘速度实验 & 程序 I/0 实验

同样是乘法，问题规模 N 变大以后，谁更快？我们将记录不同算法在速度上的记录。同时，我想观察 C、CUDA C 和 Java 在 I/O 上的时间区别，我们将用他们**内置的时间记录**监测他们的读写矩阵的速度，并进行比较。我们将在试管 1 号试管 2 号上一起进行实验。

N 的取值会影响我们的程序速度，同时合适的 N 可以让我们观察程序计算复杂度常数。再考虑电脑内存的影响，我们这次将 N 分为多个阶段进行比较。

问题规模 N	矩阵数值大小
2	[-1,1]
8	[-1,1]
16	[-1,1]
32	[-1,1]
64	[-1,1]
128	[-1,1]

问题规模 N	矩阵数值大小
256	[-1,1]
384	[-1,1]
512	[-1,1]
640	[-1,1]
768	[-1,1]
896	[-1,1]
1024	[-1,1]

问题规模 N	矩阵数值大小
2048	[-1,1]
3072	[-1,1]
4096	[-1,1]
5120	[-1,1]
6144	[-1,1]
7168	[-1,1]
8192	[-1,1]
9216	[-1,1]

选用 Java 算法	选用 C 算法
标准 ijk，Java 多线程	标准 ijk, C OpenMP

实验 4：运行时间 & 性能瓶颈实验

Java 和 C 启动时发生了什么？JVM 的启动和终止占程序的多少时间？传说中的 GC 对速度会有 drawback 吗？程序运行时的性能瓶颈在哪里，怎么分析？在 CUDA C 在运行时和纯 C 有什么区别？我们将用 **Profiler 进行监测**。

我们选取两个 100 x 100 数值区间[0,2]的矩阵进行对比，查看计算结果。

问题规模 N	矩阵数值区间
100，1000	[0,2]

附：实验样品（矩阵）选取考虑

对于多个实验中我们选用 100 x 100 的矩阵，选用它有这么几个好处：

1. 鲁棒性好，时间稳定，程序运算时不会因为矩阵过大或过小而出现时间范围大波动。

2. 实用性好，我们的几个实验是比较语言特性，100 x 100 满足我们的需求。

我们在实验中选择数值范围[0,2]和[-1,1]的矩阵，主要理由如下：

1. 合适性高。float 数据类型在[0,2]和[-1,1]之间的分布比较密集，可以更好地看到 float 的计算效果；

2. 可检验性强，[0,2]的数值范围内随机选择的数的期望是 1，而经过矩阵乘法运算后，结果矩阵上的数的期望为 N。我们可以由此判断矩阵乘法有没有正常运行。而在问题规模 N 上涨后，结果矩阵上的数也跟着上涨，我们转而采用[-1,1]之间的数进行计算。

3. 简化精度问题，这会在接下来的实验 1 结果中进行展现。

我们选择 N x N 的方阵作为我们算法的输入。这样有这么几个好处：

1. 标准化，N x M 矩阵差异比较大，我们不好准确定义问题规模。

2. 简易化，让我们的程序设计更简单。我们的目标不是拿下 ICPC 冠军，而是比较两门语言。简单的算法可以使我们花更多的心思在性能比较上。

我们在程序速度实验中，选择 N 为 2 的幂次方及其倍数，有这么几个好处：

1. 适配性好，Strassen 算法需要 N 为 2 的幂次方的矩阵输入，这样才适合我们进行比较。

2. 响应度高，阶梯式的问题规模可以更好地帮助我们查看程序运行速度变化情况。

Part 5: 实验结果

5.1 运算正确性实验结果

我们将相同的 100 x 100 的矩阵 Matrix1.txt 和 Matrix2.txt 喂给 C 和 Java 的乘法程序后，程序都成功输出了结果。表 5.1.1 是我们 C 和 Java 程序的输出部分结果。

首先我们发现，C 和 Java 程序输出的结果跟在试管 1 号或者 2 号平台的运行选择无关。根据矩阵大小和[0,2]的取值范围，我们的结果矩阵内的值确实在 100 左右波动。同时两个算法都算出了相近的结果。

但是，可以看出，C 程序计算出的所有的结果都是标准的 float 数值，打印出了小数点后 6 位。而 Java 则是有的是 6 位，有的则只有 5 位甚至是 4 位。

表 5.1 不同矩阵乘法的计算结果（部分）

经过检查，我排除了在 write 或者 printf 上的可能性。随后在调试 Java 的过程中，我发现了猫腻：在计算 C[0][0]的结果时，一开始数字保持着 6 位小数，随后随着运算的增加，突然在一次运算中就只会保留 5 位小数了。

我随后推测这是 float 精度的结果。根据 Project1 计算器里对 IEEE754 的研究，在 float 的指数位确定的情况下，32bit 的 float 最多可以表示 2^23 = 8388608 个数字，一共七位，这意味着最多能有 7 位有效数字，但绝对能保证的只有 6 位，即 float 的精度为 6~7 位有效数字。

而在矩阵乘法中 C[0][0]会等于 100 个 float 数的和。那么根据**不确定度的传递公式**，当两个 float x_1 与 x_2 相乘时，得到的新 float 的不确定度为：

$$\frac{u_{y_i}}{x_1 x_2} = \sqrt{(\frac{u_{x_1}}{x_1})^2 + (\frac{u_{x_2}}{x_2})^2} \ldots\ldots (5.1.1)$$

随后 100 个新 float 相加，他们的不确定度变为：

$$u_k = \sqrt{\sum_{i=1}^{100} {u_{y_i}}^2} \ldots\ldots (5.1.2)$$

我们通过 C 标准库中的 float.h 头文件里的常量 FLT_EPSILON，查得在[0,2]之间的 float 数值差距约为$u_0 = 0.0000001192$。令所有的$u_{x_i} = 0.0000001192$，$x_i = 1, y_i = 1$，我们得到结果矩阵的平均不确定度为：

$$u_k = \sqrt{100 \times 2 \times 0.0000001192^2} = \sqrt{200}u_0 = 1.68576 \times 10^{-6}$$

通过这个理论计算，我们发现在运算后结果 float 的不确定度会扩大到小数点第 6 位。而如果考虑真实 x1 与 x2 取值会有不同，会使得公式 5.1.1 中 x_1 和 x_2 对 y_i 的不确定度产生影响，从而使得不确定度发生变动。这可能使得不确定度大于或者小于u_k。这使得最后的不确定度在 4-6 位之间波动。

这说明，Java 在计算时考虑了不确定度的影响，在计算结果上**保留了结果中的精确值**，省去了不确定值。但是 C 没有考虑，就 float 值的保留规则**完全保留了所有的 6 位**。我们在 C 中是**无法感受到不确定度扩大和精度损失扩大的影响**的。

同时，我们可以推导不确定度随着问题规模 N 和取值范围$[l_1, l_2]$的变化公式：

$$u_k = \sqrt{2N}\, u_0 \ldots\ldots (5.1.3)$$

根据公式 5.1.3，随着问题规模的增加，结果矩阵的不确定度会增加，而随着取值范围l_2的增大，u_0也会随之增大。但是如果l_1, l_2之间越接近，x_1 和 x_2 对 y_i 的不确定度产生的影响会下降，最终结果矩阵里每个数的不确定度间的差距会减小。

这是我们的第一个实验，但是结果却让我出乎意料的学到了许多。虽然我们接下来要比较的主要在性能，但是不确定度的计算让我对乘法这一基本运算有了更深的认识。另外我们也得到了 C 和 Java 的第一个差别：精度保留。

那么，这个精度保留所带来的开销，会不会使 Java 更慢呢？这很可能

5.2 编译优化实验结果

本节实验我们选择 Java 和 C 的 ijk 标准程序，比较 Java 的 JIT 以及 C 在不优化、-O2、-O3 的优化效果。

对于标准 100 x 100 矩阵，我们对 Java 执行运行第一次程序，复制文件并重复运行得到 5 次实验结果。之后在第一次运行后的基础上连续运行 15 次程序，得到多次运行程序的运算时间结果。

随后我们计算 5 次第一次运行程序与 15 次多次运行程序时间。我们可以看到，Java 在读矩阵所花费的时间用时最多，而在矩阵相乘上用时最短。比较有意思的事实是写矩阵的速度是快于读矩阵的速度而非我猜测的用时相等，这应该是我在读矩阵时为了图方便用了 BufferReader 和正则表达式等工具导致的，他们拖慢了程序的读速度。

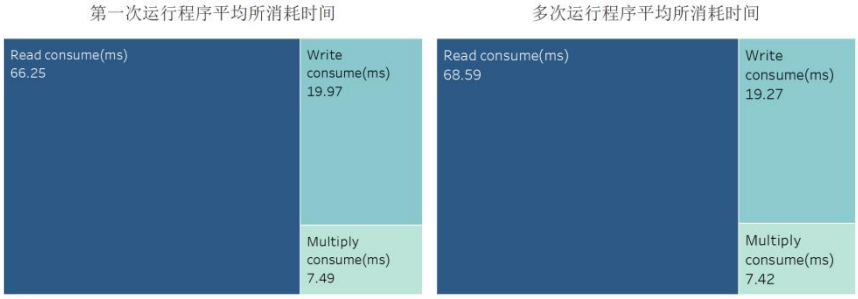


图 5.2.1 Java-i jk 矩阵乘法运行消耗时间

虽然在数据库的 Project 中我们可以清晰地发掘 JIT 带来的优化，但是面对基础的矩阵相

乘，JIT 带来的优化并不明显。

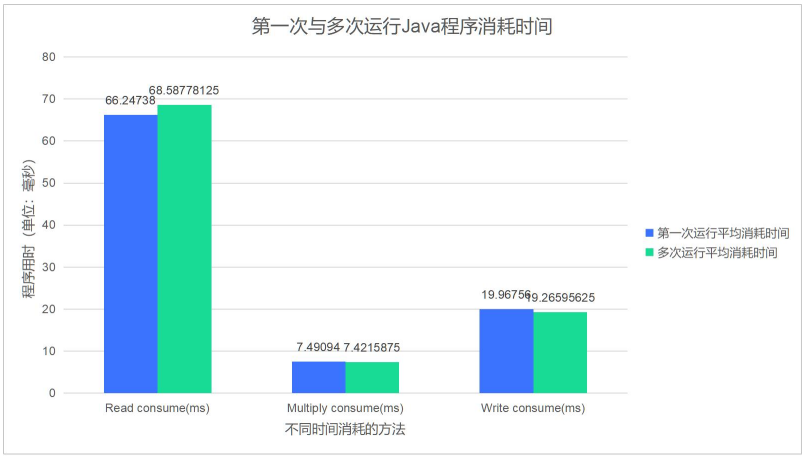


图 5.2.2 Java-ijk 矩阵乘法运行消耗时间对比

通过查看 15 次程序运行的时间，我们也发现这里边运行时间没有明显规律，三个方法的趋势线 R^2 值也低于 0.10，几乎没有趋势，这样的随机性说明 Java JIT 可能没有进行太多的优化。

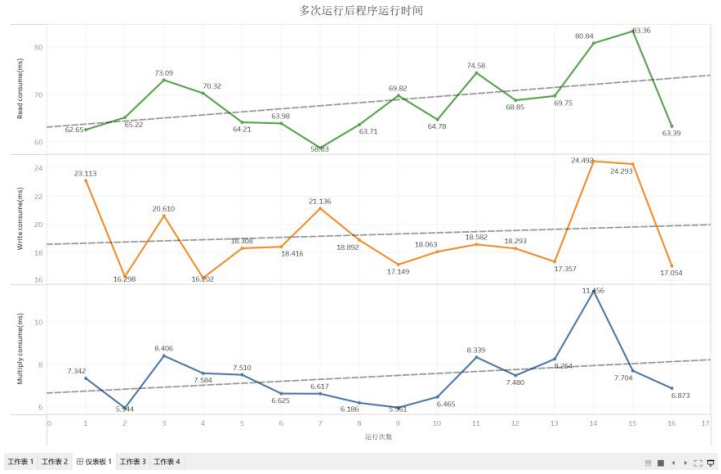


图 5.2.3 Java-ijk 矩阵乘法多次运行消耗时间趋势图

接下来我们测试 C 程序的优化效果，将普通 ijk 算法编译并运行 10 次，取程序所用时间均值。

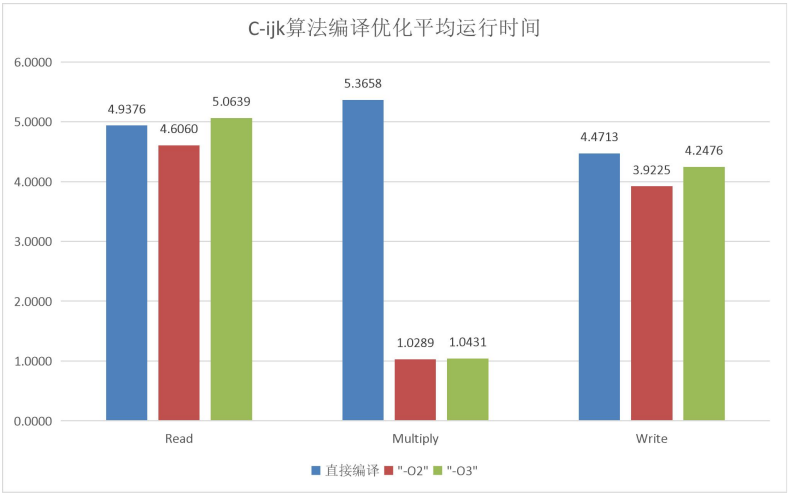


图 5.2.4 C-ijk 矩阵乘法运行平均消耗时间（单位：ms）

神奇的一幕发生了：虽然开启-O2 和-O3 选项没有明显改变 I/O 的所用时间，但是，**矩阵乘法所用的时间迅速下降了**！

这是怎么一回事？怎么 JIT 就做不到这样的优化？我要打开天窗一探究竟。

5.2.1 JAVA 加速：JIT 工作原理

在 JAVA 中我们一共会被编译两次：从 Java 到 JVM 看得懂的字节码，从字节码到计算机能运行的机器码。而我们的 Java 运行优化正是发生在字节码到机器码的过程中。

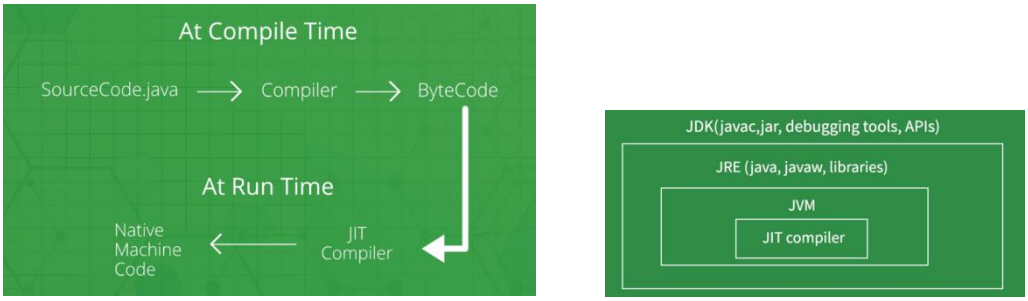


图 5.2.5 Java 编译过程，来源 https://www.geeksforgeeks.org/just-in-time-compiler/

在启动 Java 程序时，JVM 里的解释器就会开始工作。解释器会将.class 文件一行一行翻译之后再运行。它不会一次性把整个文件都翻译过来，而是翻译一句，执行一句，再翻译，再执行，所以解释器的程序运行起来会比较慢，每次都要解释之后再执行。

图 5.2.6 Java 解释器运行流程，来源：https://zhuanlan.zhihu.com/p/347564885

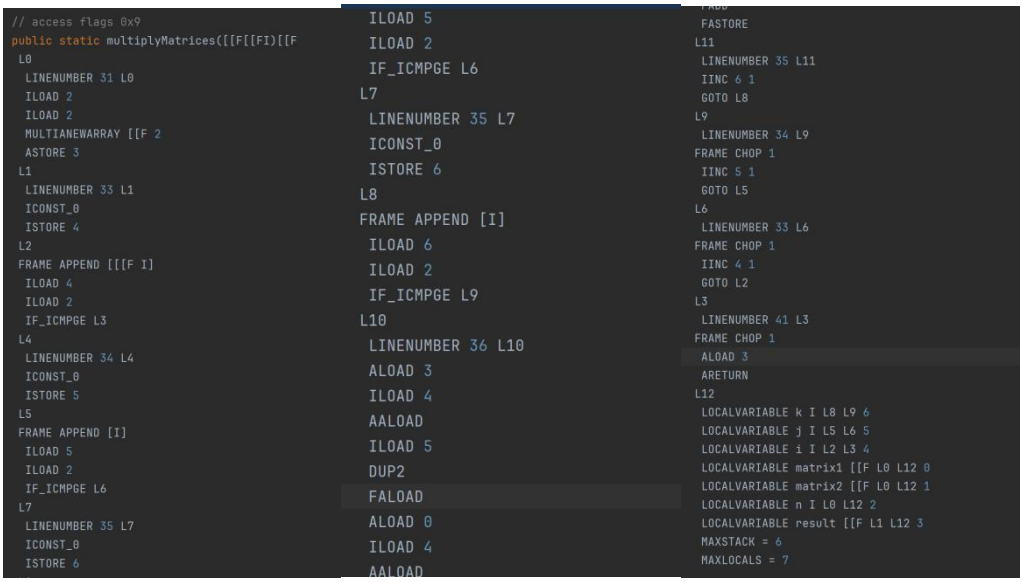


图 5.2.7 Java 矩阵乘法字节码

这里我找到了 Javac 编译出的矩阵乘法的字节码。可以看到，全程代码还是非常的长，且具体的 ijk 指针字节码花费了很多指令来进行。

所以，能不能把解释之后的内容缓存起来，就像数据库的缓存和 CPU 的缓存，就可以直接运行而不再需要解释了？但是，如果将每段代码都缓存起来，如将仅执行一次的代码也缓存起来，这太浪费内存了。由此我们引入一个新的运行时编译器，JIT 来解决这些问题，加速**热点代码**的执行。

图 5.2.8 Java JIT 编译器工作流程，来源：https://zhuanlan.zhihu.com/p/347564885

根据 ORACLE 官网上关于 JIT 的介绍，我了解到，JIT 会识别程序中的热点代码，随后，将这些代码直接翻译成机器码。这样，在 JVM 执行机器码到热点代码时，计算机将直接执行机器码，从而提高我们程序的运行效率。

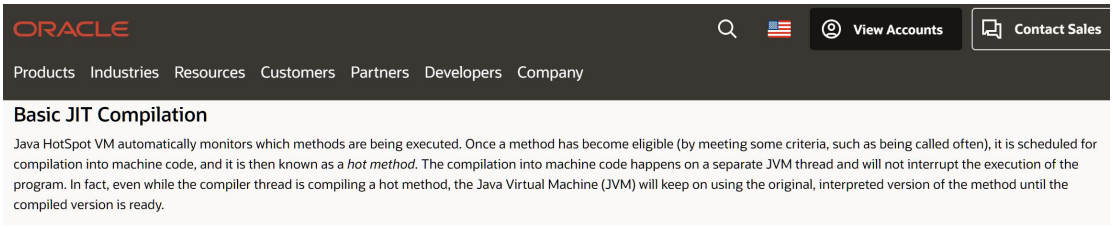


图 5.2.9 ORACLE 关于 JIT 介绍，来源：
https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html

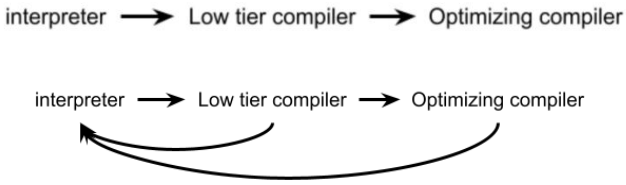
而热点代码、热方法的选择算法不同的 JVM 有不同实现，如基于方法，基于踪迹，基于区域。我们平时遇到的最典型的基于方法的 JIT 中，一般探测热点方法有基于采样的热点探测，即周期性的去检查线程的调用栈顶，如果方法经常出现在栈顶，那它就是热点方法。

这个操作有点像我们 Profiler 去监控程序。另一种是基于计数器的热点探测，这种会给每个方法建立计数器，用来统计方法的执行次数。超过阈值的就认为是热点方法。

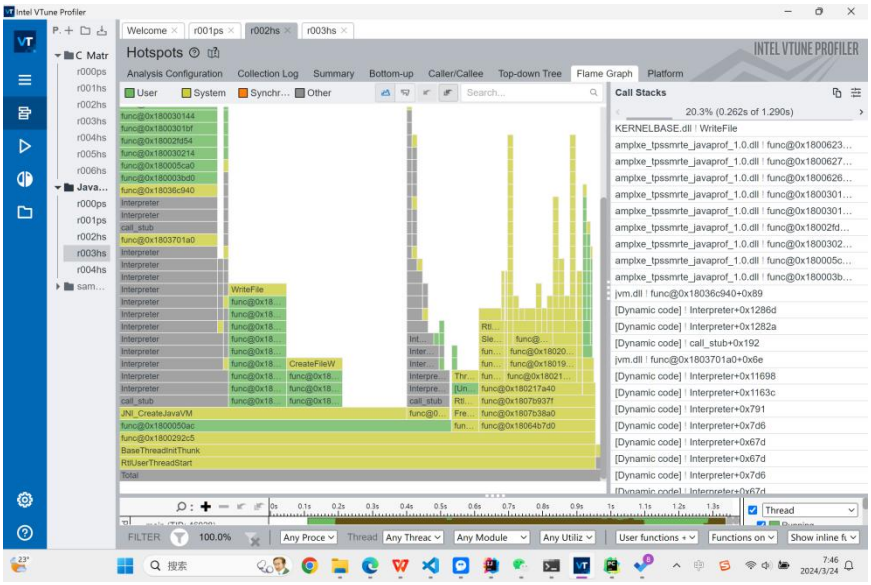
Multi-tiered execution

In HotSpot, the interpreter instruments the code that it executes; that is, it maintains a per-method count of the number of times a method is entered. Because hot methods usually have loops, it also collects the number of times a branch back to the start of a loop is taken. On method entry, the interpreter adds the two numbers and if the result crosses a threshold, it enqueues the method for compilation. A compiler thread running concurrently with threads executing Java code then processes the compilation request. While compilation is in progress, interpreted execution continues, including for methods in the process of being JIT'ed. Once the compiled code is available, the interpreter branches off to it.

而在对于检测出的热点代码，编译器也分为了两种：C1，C2。由于历史原因，C1 也被称为客户端编译器，C2 被称为服务器编译器。与 C1 编译器相比，C2 编译器对性能要求更高，会对代码做更加深层的优化，相应的也会比 C1 编译的时间更长。JVM 会根据代码情况，执行时间情况动态地选择 C1，C2 编译器。由于 JVM 会动态推测热点代码，假如一段代码使用了一段时间后不再是热点代码了（比如先读取矩阵，后面就不再使用此方法），JIT 可能会执行“去优化”，将停止执行编译的代码以切换到更慢的解释代码。

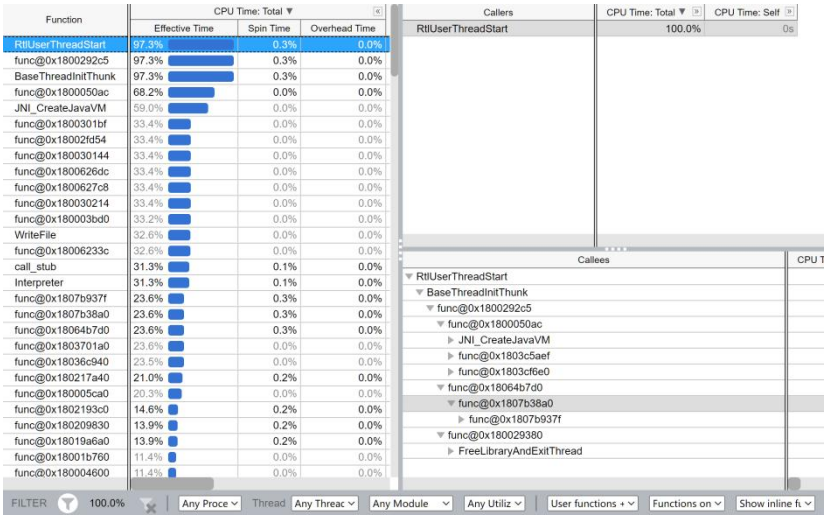


理论了这么多之后，我们来看在实际过程中 JIT 的执行结果。在第一次执行 Java 的时候，我们看到，编译器花了许多时间对程序进行编译，然后再运行编译好的代码。而这些编译的代码是动态的，他们在执行我们的读写函数时（根据用时判断应为 func@0x180003bd0 等函数）边进行编译。而对矩阵乘法函数的编译则出现的比较靠后，并且我们看到是在 call_stub 函数后进行编译的，我推测这个函数便是 JIT 检测热点代码的函数。同时我们可以看到，大部分 System 层面的运行都是在维持 JVM 的运转，程序在开始 CreateJavaJVM 和结束时调用了很多方法和线程。

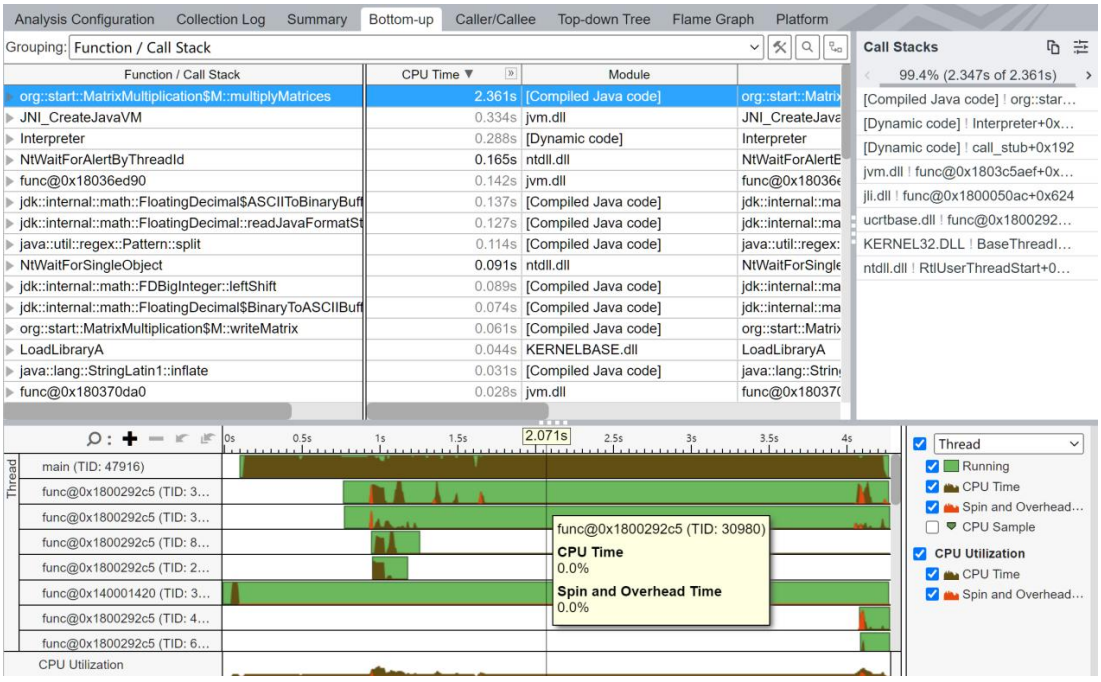


事实上我们也可以看到，在第一次运行中程序在启动 JVM 上的时间反而是占比最多的。当然这也和我们的矩阵规模选取有关，一旦规模增大，在方法上所耗用的时间应还是最多。

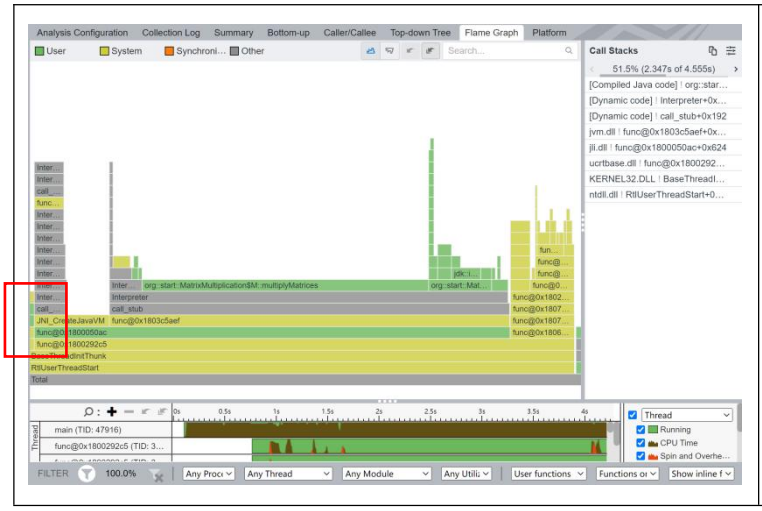
这里我们选择的矩阵规模为 100 x 100，可能还有很多参数 Profiler 来不及收集，于是我们决定选择一个 1000 x 1000 的矩阵。

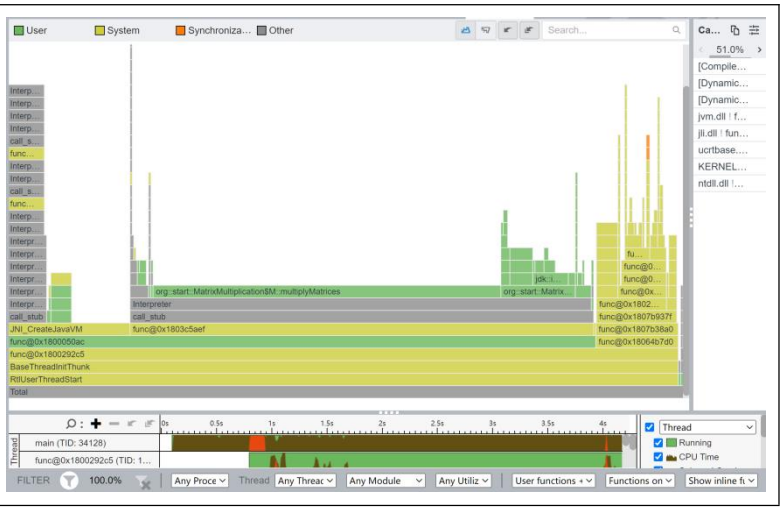


在 1000 x 1000 矩阵的输入下，程序将大部分时间花在了运算上，不过 CreateJavaJVM,Interpreter 还是占用很多时间，接近 0.6 秒。 另外一个比较有趣的观察是，JVM 有会有自己的数学库：jdk::internal::math。

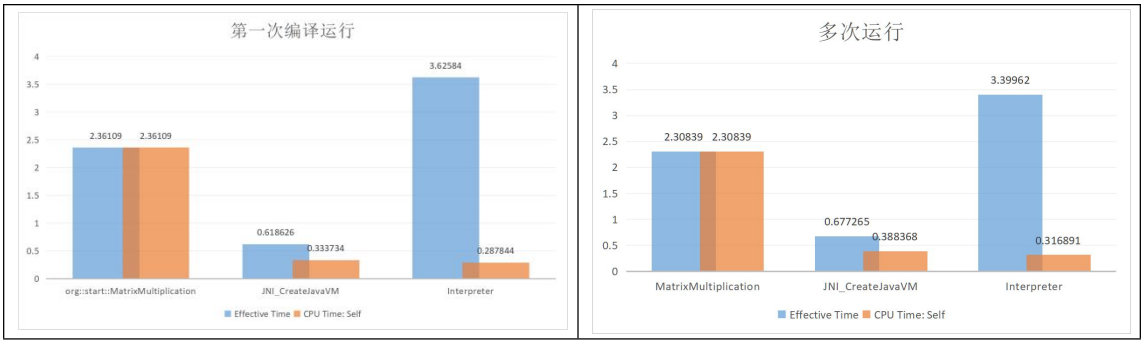


同时，JVM 还是会先进行编译，然后在我们的程序执行过程中动态地编译程序。这里我将第一次运行的程序热力图放在左边，多次运行的程序热力图放在右边。可以看到，程序运行的基本流程是大致相同的，但是多次运行的程序会省略掉部分子程序，有部分程序会被提前执行、更多精细的程序会被调用等等。

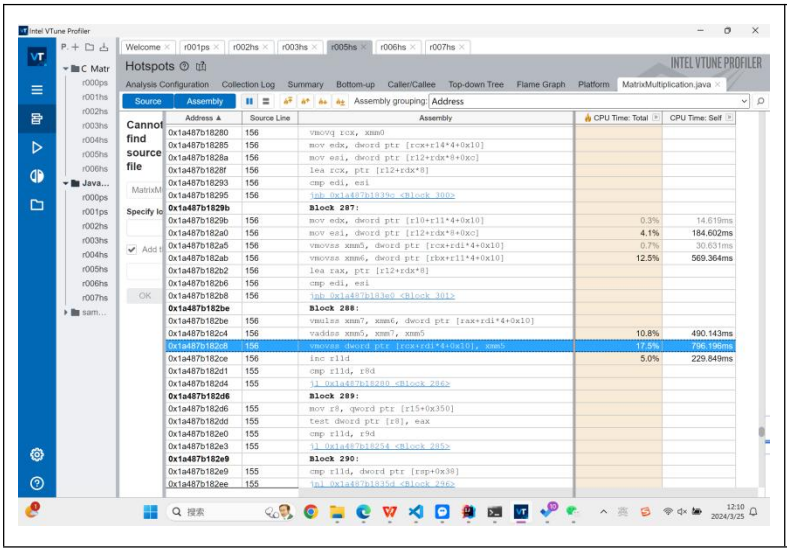


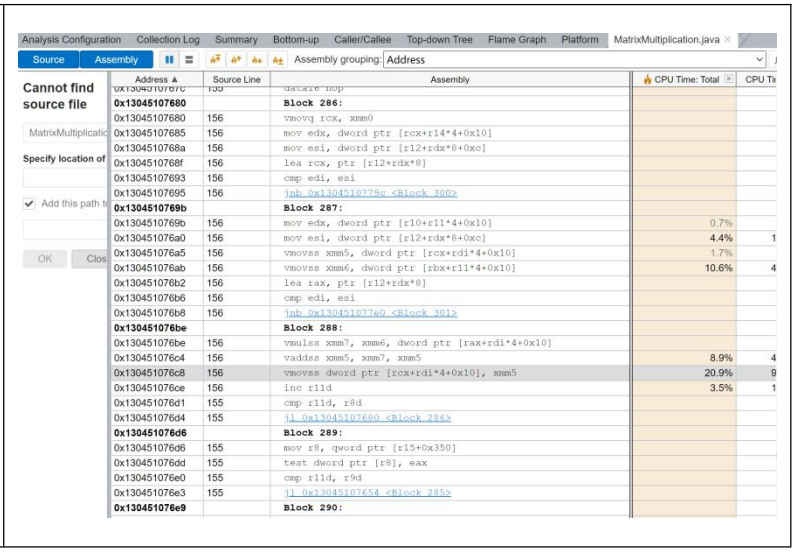


在多次运行的情况下， 程序编译的有效时间减少了，矩阵乘法的运行速度也有了微量提升。



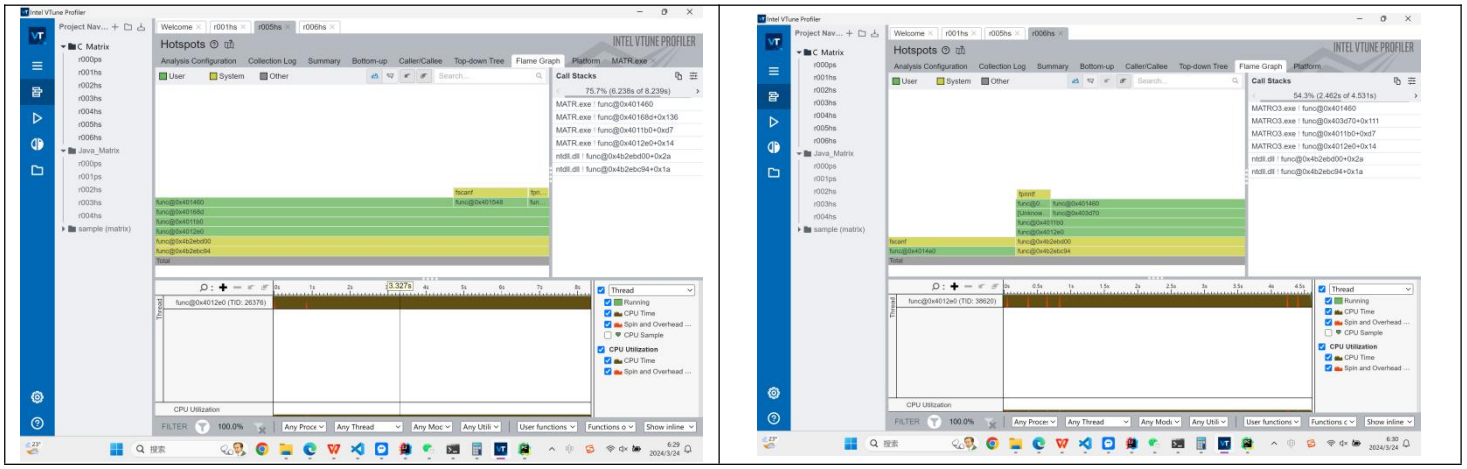
我还是最关心 MatrixMultiply 方法被编译成了什么。在这里我们成功地查看到 JVM 翻译出的汇编代码层。我们可以看出，虽然在执行时间上差距不大，但是汇编层面.....居然也没有优化，并且全是奇妙的 caller callee 指令。我在这里用了多种方法，尝试让编译器重复计算 MatrixMultiply，但是很遗憾的，所有的尝试 JVM 出来的汇编代码都是一样的。也就是说我们的 JIT 优化对于我们的矩阵乘法效果上在汇编层面**没有优化汇编**。



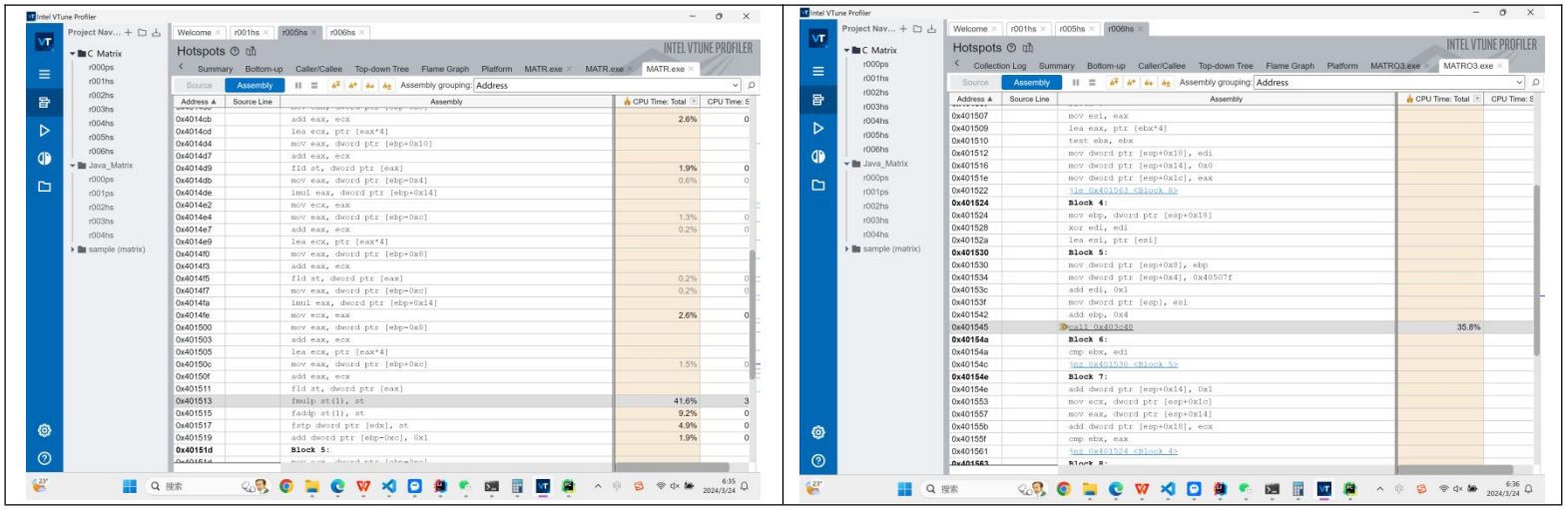


5.2.2 C 编译加速：-O3 到底做了什么？

我们接下来查看 C 的矩阵乘法的效果。由于 C 在运行 100 x 100 时太高速了，Profiler 没有捕捉到程序的太多信息。于是我也给它投喂了 1000 维的矩阵，查看运行的信息。左边的表格是标准编译，右边则是使用了-O3 进行了编译。



我们可以看出，跟 Java 的复杂比起来，C 的代码执行的就很干脆利落。并且，编译带来的优化是显而易见的，连程序运行的结构都发生了改变。



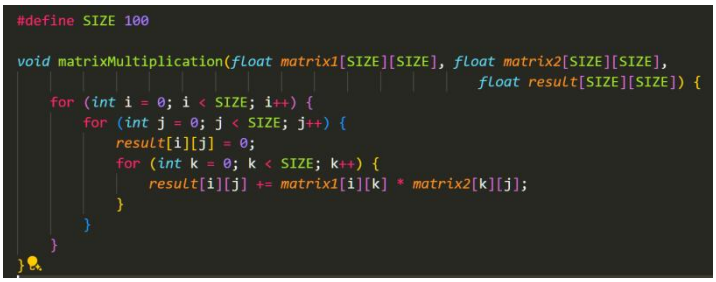
我们接下来找到并对比 MatrixMultiply 的部分汇编代码。可以看到，**最多优化的就是代码 Block4**。我们可以把 block4 的代码 copy 下来进行分析：

首先是直接编译，根据计组的知识我们可以知道，mov 是移动,eax,edx 是 x86 上的寄存器，dword 是 double word，lea 是装载有效地址，fld 是加载浮点数到寄存器，imul 是执行有符号乘法，fmulp,faddp 是浮点数的乘法和加法。黑色字体的注释是我写的。（啊，感谢计组）

```
1.  Address Source Line Assembly CPU Time: Total CPU Time: Self
2.  0x4014a5 Block 4:
3.  0x4014a5 mov eax, dword ptr [ebp-0x4] 2.1% 0.170s
4.  0x4014a8 imul eax, dword ptr [ebp+0x14] 0.2% 0.016s
```

```
0x4014ac mov edx, eax
0x4014ae mov eax, dword ptr [ebp-0x8]
0x4014b1 add eax, edx 1.9% 0.156s  # eax = edx
0x4014b3 lea edx, ptr [eax*4]      # edx = eax * 4 目的是得到地址 i
0x4014ba mov eax, dword ptr [ebp+0x10]
0x4014bd add edx, eax
0x4014bf mov eax, dword ptr [ebp-0x4] 2.1% 0.171s
0x4014c2 imul eax, dword ptr [ebp+0x14]
0x4014c6 mov ecx, eax
0x4014c8 mov eax, dword ptr [ebp-0x8]
0x4014cb add eax, ecx 2.6% 0.217s
0x4014cd lea ecx, ptr [eax*4]  # ecx = eax * 4 目的是得到地址 j
0x4014d4 mov eax, dword ptr [ebp+0x10]
0x4014d7 add eax, ecx
0x4014d9 fld st, dword ptr [eax] 1.9% 0.154s
0x4014db mov eax, dword ptr [ebp-0x4] 0.6% 0.047s
0x4014de imul eax, dword ptr [ebp+0x14]
0x4014e2 mov ecx, eax
0x4014e4 mov eax, dword ptr [ebp-0xc] 1.3% 0.109s
0x4014e7 add eax, ecx 0.2% 0.016s
0x4014e9 lea ecx, ptr [eax*4]   # ecx = eax * 4 目的是得到地址 k
0x4014f0 mov eax, dword ptr [ebp+0x8]
0x4014f3 add eax, ecx
0x4014f5 fld st, dword ptr [eax] 0.2% 0.016s # 加载 st matrix[i][k]
0x4014f7 mov eax, dword ptr [ebp-0xc] 0.2% 0.016s
0x4014fa imul eax, dword ptr [ebp+0x14]
0x4014fe mov ecx, eax 2.6% 0.214s
0x401500 mov eax, dword ptr [ebp-0x8]
0x401503 add eax, ecx
0x401505 lea ecx, ptr [eax*4]
0x40150c mov eax, dword ptr [ebp+0xc] 1.5% 0.125s
0x40150f add eax, ecx
0x401511 fld st, dword ptr [eax]  # 加载 st matrix[k][j]
0x401513 fmulp st(1), st 41.6% 3.424s   #  st = matrix[i][k] * matrix[k][j]
0x401515 faddp st(1), st 9.2% 0.761s        #  result = result + st
0x401517 fstp dword ptr [edx], st 4.9% 0.406s
0x401519 add dword ptr [ebp-0xc], 0x1 1.9% 0.156s
```

这段代码比较长，但是其实不难理解或者猜测它的行为。结合 GPT，大概是这样的：ebp 是我们的基址指针，在 x86 架构中，通常使用基址指针来帮助访问栈上的局部变量和函数参数。随后，我们在栈上存储了我们矩阵的 ijk 三个循环变量，matrix1，matrix2，result 的基地址。我们依靠 ebp 取出 matrix[i][k]和 matrix[k][j]，然后使用 fmulp 和 faddp 完成矩阵的乘法。



```c
#define SIZE 100

void matrixMultiplication(float matrix1[SIZE][SIZE], float matrix2[SIZE][SIZE],
                          float result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            result[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

这段代码从注释上可以看到，它是有冗余的，比如在得到地址 ijk 上前面的步骤非常的多。当然，它不失为一种好方法，毕竟这种方法算是翻译了 C 语言，如果在计组课上这可能会被当成一个 example。

我们再来看看开了-O3 后编译器编译出的汇编代码：

```
1.  Address Source Line Assembly CPU Time: Total CPU Time: Self
2.  0x401492 Block 4:
3.  0x401492 fst dword ptr [ecx], st
4.  0x401494 mov edx, edi
5.  0x401496 xor eax, eax
6.  0x401498 fld st, st(0)
7.  0x40149a lea esi, ptr [esi]  # 矩阵
8.  0x4014a0 Block 5:
9.  0x4014a0 fld st, dword ptr [ebx+eax*4] 7.5% 0.339s # 加载第一个矩阵元素到浮点寄存器栈中的栈顶
10. 0x4014a3 add eax, 0x1  # 将 eax 加 1
11. 0x4014a6 fmul st, dword ptr [edx]  # 将栈顶的浮点数与地址为 edx 的浮点数乘起来
12. 0x4014a8 add edx, esi 31.3% 1.416s # 将 edx 加上 esi
13. 0x4014aa cmp ebp, eax 1.4% 0.062s  # 将 ebp 与 eax 进行比较
14. 0x4014ac faddp st(1), st  # 将栈顶的两个浮点数相加，并弹出栈顶的浮点数
15. 0x4014ae fst dword ptr [ecx], st 12.2% 0.555s # 将栈顶的浮点数存储到地址为 ecx 的内存中
16. 0x4014b0 jnz 0x4014a0 <Block 5> 2.0% 0.091s # 如果比较结果非零，则跳转到 Block 5，否则继续执行下一条指令，这里应该是比较的是 k 和 size
1.  0x4014b2 Block 6:
2.  0x4014b2 fstp st(0), st
3.  0x4014b4 add ecx, 0x4  # ecx 是地址，地址+=1，应该是 i，代表 matrix1 的 i 行
4.  0x4014b7 add edi, 0x4  # edi 的地址+=1 应该是 j，代表 matrix2 的 j 行
5.  0x4014ba cmp ecx, dword ptr [esp]
6.  0x4014bd jnz 0x401492 <Block 4> # 回到 block4
```

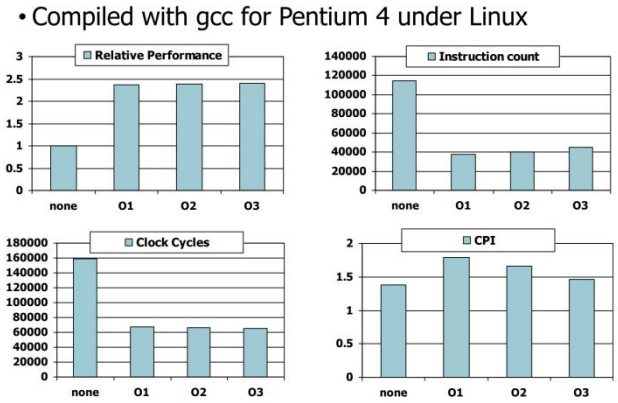
这段代码就没有那么直白，但是通过仔细分析我发现，-O3 确实是在 ijk 循环上面下了功夫，**直接把原本栈上的 i jk，SIZE 变量直接存储在 CPU 的寄存器上**，减少了我们的 I/O 读写次数，同时，add ecx 0x4 这个操作也简化了地址的效率。这样，不仅是我们的总指令数 IC（Instructions Count）减少了，因为我们简化了取地址和指针的汇编程序。而且，我们的平均 CPI（Clock cycles per instruction）也减少了，因为我们向栈内存上的访问次数减少了。

根据计组里的 CPU 性能估算公式，对于一个程序，它的运行时间跟 IC，CPI 和时钟频率挂钩。在 CPU 时钟频率不变的情况下，**IC 的下降和 CPI 的下降，让我们的程序运行时间下**

降了。（ps，这是我做实验 2 时的分析，而当我在做实验 4 时我发现，**真实情况是 IC 下降，CPI 上升了**，不过由于-O3 指令的优化，所以总时钟周期数下降了，才导致了整个 CPU Time 下降了。**我在这里做了一个错误的猜测**）

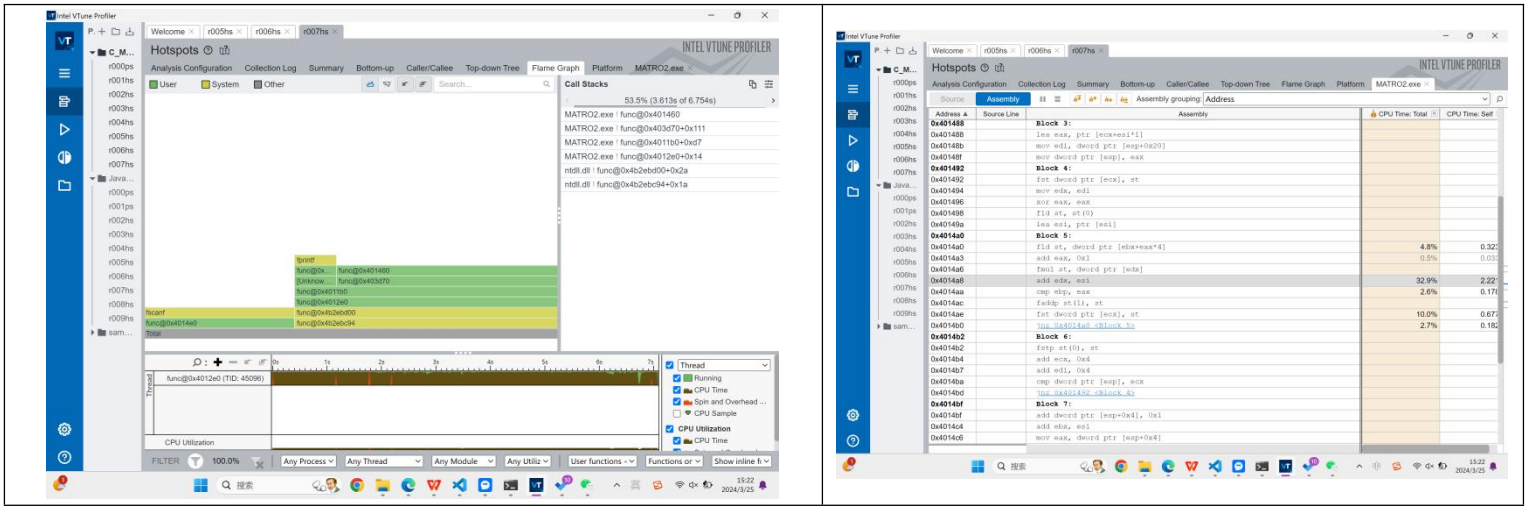
$$\text{CPU Time} = \text{Instruction Count(IC)} \times \text{Cycles per Instruction(CPI)} \times \text{Clock Period}(T_c)$$

计组课白老师也在奔腾 4 CPU 上做了实验，发现确实对于不同的编译器，性能出现了一定程度的提升，在此作一个引用。

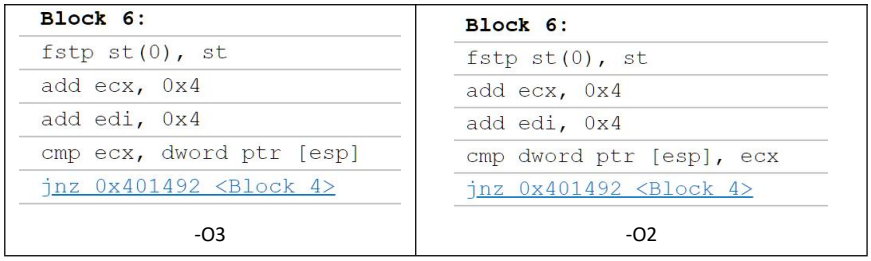


Source:https://bb.sustech.edu.cn/bbcswebdav/pid-455556-dt-content-rid-15826365_1/courses/CS214-30022126-2024SP/CompOrg_24S_Lec5_Performance.pdf

当然，这里我也尝试-O2 的效果，这里编译出来的结构和-O3 完全一致，如果比较汇编，会发现代码结构也基本一致。



比较不同的汇编代码在于 Block 6，-O2 导入的还是 dword 到寄存器中，但是到-O3 就直接存储的是指针 ecx。可以看出，-O3 会优化到极致，能用寄存器就用寄存器，可以用指针就用指针。



在 CSDN 上这篇文章：#linux# gcc 编译优化-O0 -O1 -O2 -O3 -Os -CSDN 博客，给出了-O0,-O1,-O2,-O3 的优化范围。比如下面是-O2 的优化范围：

- l -fdefer-pop：延迟栈的弹出时间。当完成一个函数调用，参数并不马上从栈中弹出，而是在多个函数被调用后，一次性弹出。
- l -fforce-mem：在做算术操作前，强制将内存数据copy到寄存器中以后再执行。这会使所有的内存引用潜在的共同表达式，进而产出更高效的代码，当没有共同的子表达式时，指令合并将排出个别的寄存器载入。这种优化对于只涉及单一指令的变量，这样也许不会有很大的优化效果. 但是对于再很多指令(必须数学操作)中都涉及到的变量来说，这会时很显著的优化，因为和访问内存中的值相比 ,处理器访问寄存器中的值要快的多。
- l -fstrength-reduce：这种优化技术对循环执行优化并且删除迭代变量。 迭代变量是捆绑到循环计数器的变量，比如使用变量，然后使用循环计数器变量执行数学操作的for-next循环。

我们在刚刚的-O3 的汇编中已经感受到了这两个优化，对栈指令的优化，数据拷贝中寄存器访问的减少和 ijk 循环变量的优化。除此之外，-O2 还会执行课上老师所讲的 inline 操作，将小方法内联到我们的大方法中，优化取指令的操作。

而在-O3 下，gcc 会执行下面这个优化，我推测这是我们 block6 汇编改变的原因。

- l -fweb：构建用于保存变量的伪寄存器网络。 伪寄存器包含数据，就像他们是寄存器一样，但是可以使用各种其他优化技术进行优化，比如cse和loop优化技术。这种优化会使得调试变得更加的不可能，因为变量不再存放于原本的寄存器中。

所以，通过这个实验，我深刻地从最基层的体验到 Java 和 C 在执行层面上的区别。Java 会启动 JVM 并且进行及时编译运行代码，这里边要启动 JVM 的各种线程。**JIT** 计算会对热点代码进行 C1 或者 C2 编译以提升 Java 的运行速度。然而可惜的是 Java 面对矩阵乘法，JIT 在**汇编指令上的优化是有限的**，因而其所在乘法算法上做的优化很有限。但是 C 语言中 gcc 的**汇编指令上的优化是明显的**，它会尝试减少程序的汇编指令数，尝试减少花费高昂的内存读取指令的次数，尝试用指针代替实际的 word，减少程序的 IC 和 CPI，从而减少程序的运行时间。

同时我们也知道，JVM 是在边编译边执行机器码，这样其实是占用了程序一定的时间，并且这些编译在有时并没有取到非常好的效果，他们的汇编代码没有 gcc 编译出的那么优化。而这是我认为在乘法程序执行上 Java 会比 C 缓慢的一个重要原因。

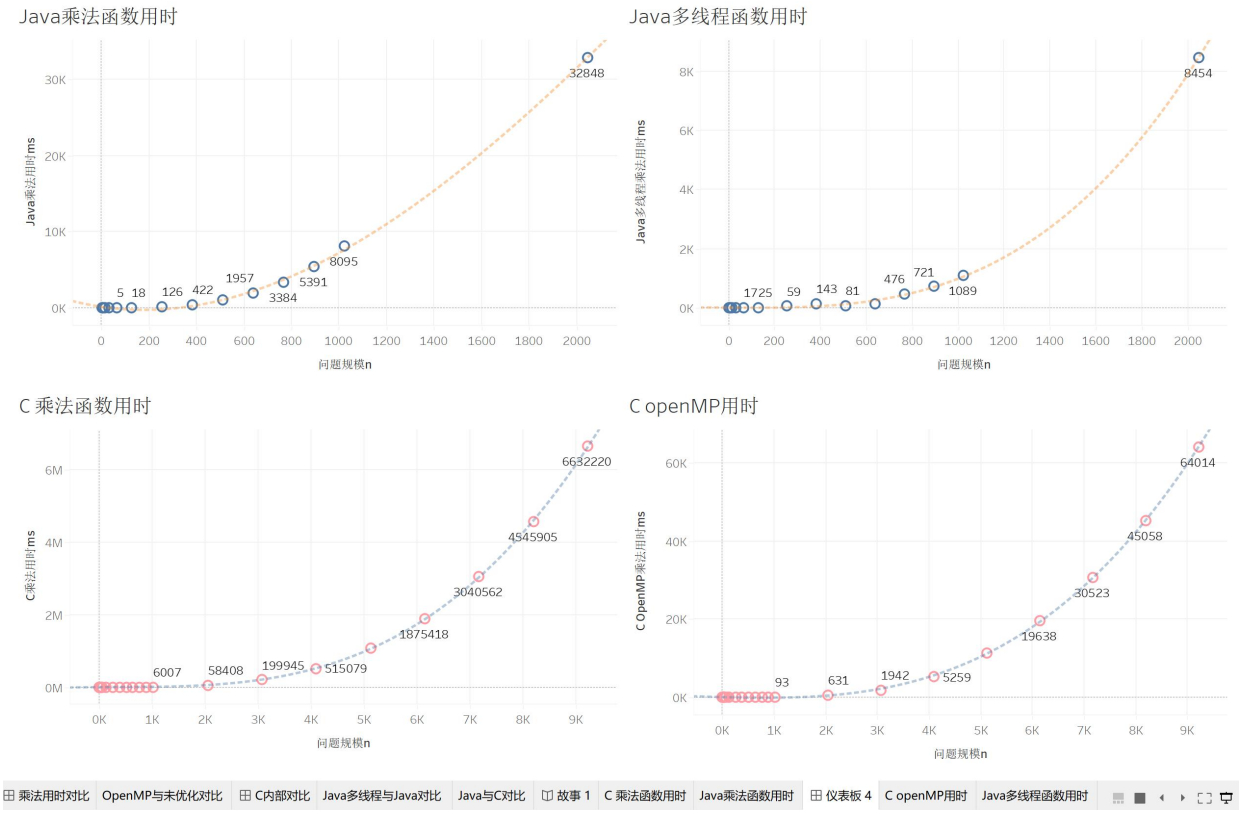
5.3 矩阵相乘速度实验 & 程序 I/O 实验结果

我们将 21 个矩阵案例输入到我们的 Java 和 C 程序中，分别在我们的试管 1 号和试管 2 号上运行。最终得到了我们的测试结果。我们将查看程序中矩阵乘法的消耗时间。

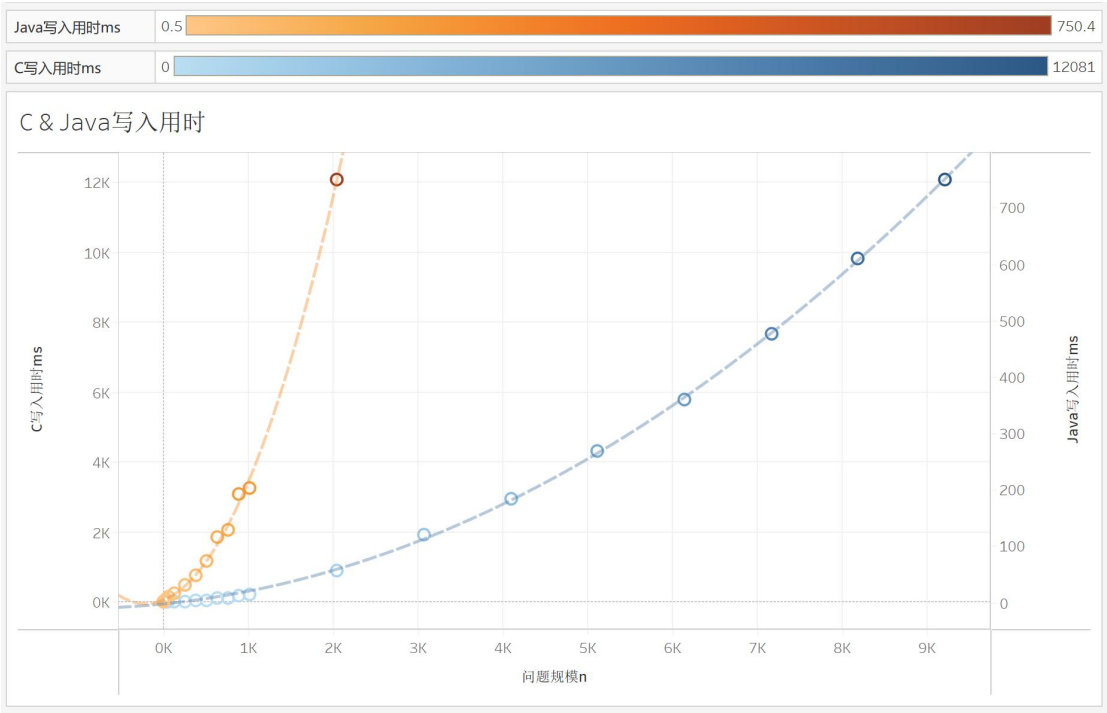
这里我们的试管 2 号由于系统限制，JVM 的内存不能开太大。我们就暂时测量到 N=2048 的矩阵。

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project2/Excutable/data$ java -cp .:data Matrix
Enter the size of the matrix:
3072
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2452 out of bounds for length 2452
        at Matrix.readMatrixFromFile(Matrix.java:27)
        at Matrix.main(Matrix.java:152)
```

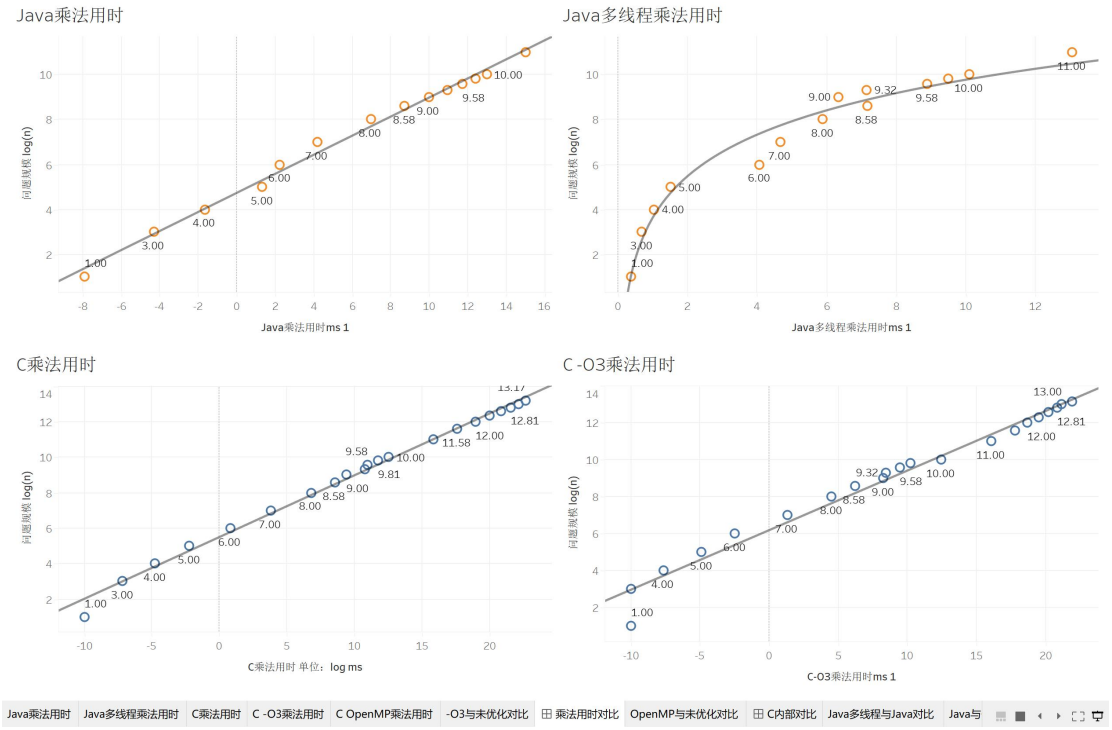
我们先来分析一下试管 1 号的程序结果。受到数据的影响，我将问题规模取了对数 log(N)，程序用时也取对数进行分析，得到下面的乘法用时图。



我们用 N^3 曲线去拟合我们的时间-矩阵维数散点图，得到的R^2值均在 0.999 以上，P 值小于 0.001。说明我们的函数的计算复杂度都在 $O(N^3)$，符合理论。而像读取和写入矩阵，使用N^2可以得到很好的拟合效果，这也说明我们的读写的复杂度就在 $O(N^2)$。



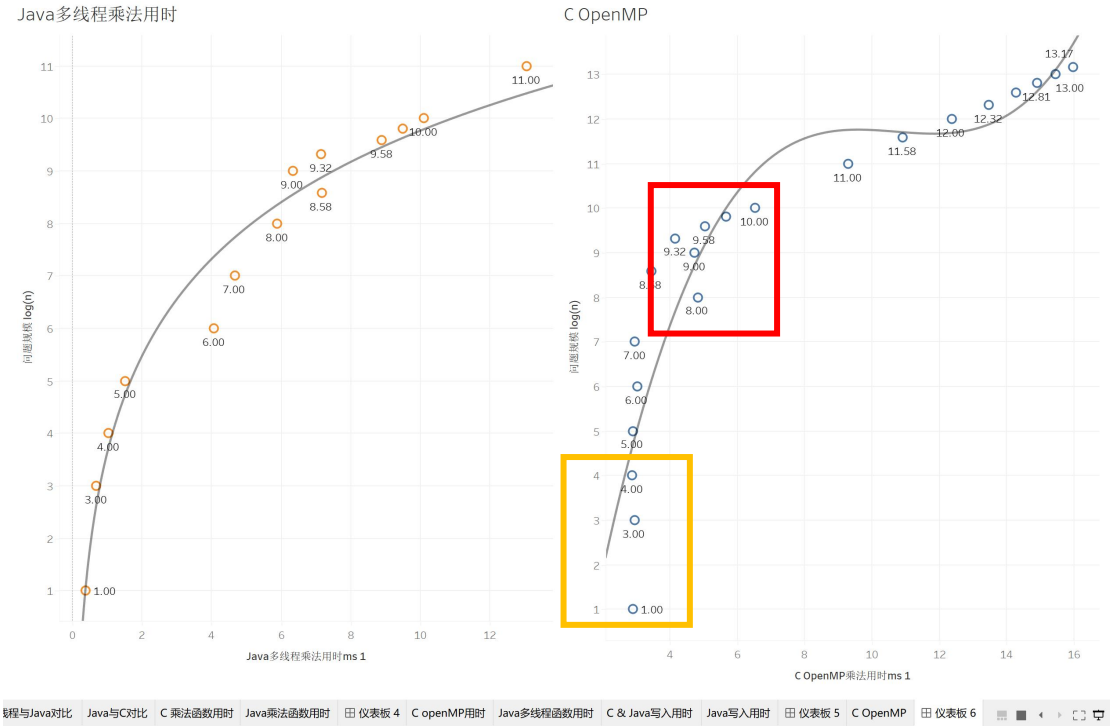
不过，我在执行数据处理的时候，开始时我并没有选择直接将数据展现成图表。我是先对我们的问题规模和用时取了对数，然后做成了下图：



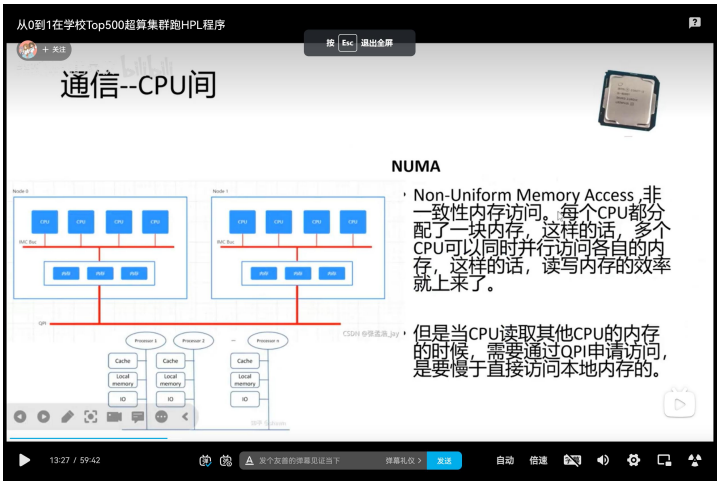
从图中我们可以得知，C 乘法，C-O3 乘法，Java 乘法的程序运行时间都满足 $\log(T(N)) \sim \log(N)$，这是可以根据刚刚的公式推出来的。但是！**Java 多线程在两者 log 之后，问题规模和乘法用时是呈现一种对数关系**！

类似的事情也发生在 OpenMP 上，它也不是一个简单的直线而是一个偏对数偏三次函数的曲线。这说明，在**经过并行处理后的程序，他们的计算复杂度可能不是$O(N^3)$！**

我认为这是一个比较重要的发现。执行了 4 个线程的 Java 程序，它的计算时间是一定大于原来程序的四分之一的。但是具体大多少，这和通信有关。而我们目前的发现是可以展现出 Java 语言和 C 语言在多线程技术下的 CPU 间通信开销。如果这块研究深入，我们甚至可以使用阿姆达尔定律来计算它们的加速比。~~（给读者留作练习）~~



另外值得一提的是我对 OpenMP 的推测。OpenMP 在单核情况下是没有特别多通信阻碍的，但是一旦多核起来就会**增加一层 L3 缓存上的开销，使得程序运行时间发生突变**。我认为这是在上图中的**橙色**方框里发生的过程。而我们的 OpenMP 是在 1 号服务器上进行的测试，它有两个 CPU 构成 NUMA 节点。当我们的 OpenMP 需要让 CPU0 和 CPU1 在内存上进行通信时，**NUMA 架构会使程序的运行时间发生第二次突变**，而我认为这就是上图中**红色**方框发生的事情。

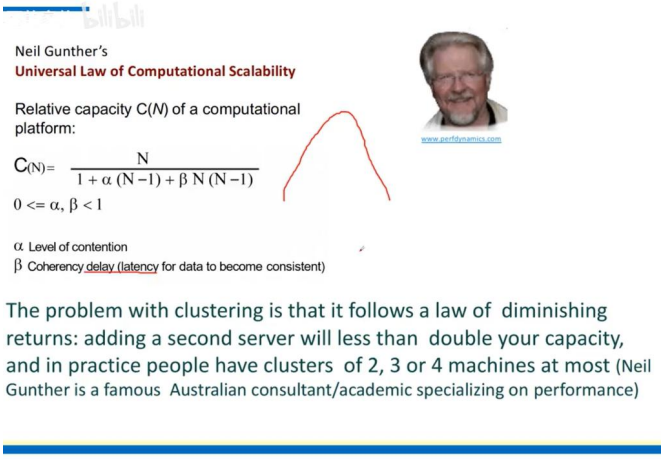


顺带一提，B 站上有个 UP 视频讲这个讲的挺好的，是南科大大二计算机系的：

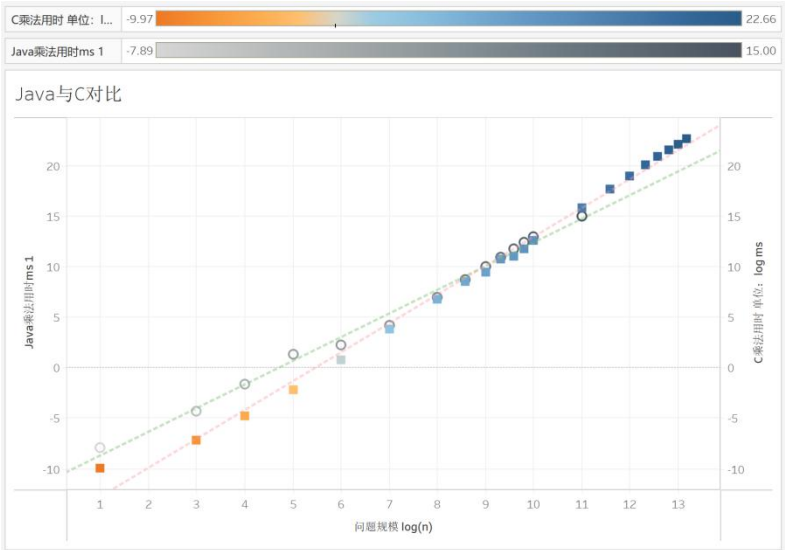
https://www.bilibili.com/video/BV1et4y1f7KK **狠狠关注！**

回来，我们还有更重要的问题没有解决：为什么多线程程序的用时会随着问题规模的增大而增加的更多，从而变成一个 $f'(x) > 0, f''(x) < 0$ 的函数呢？我推测还是通信的问题。

上学期教我数据库的老师告诉我，随着机器数量增加，计算性能并不能像是一条直线往上蹭蹭涨。当系统的通信开销大于我们的计算性能增长时，程序的执行效率就开始下降了。虽然我的数据库考试考了一坨稀饭，但是我对这个还是有很深的印象。所以，我认为造成这个曲线的原因在于通信。当然，更多的数据研究需要去做，~~留作给读者的练习~~。

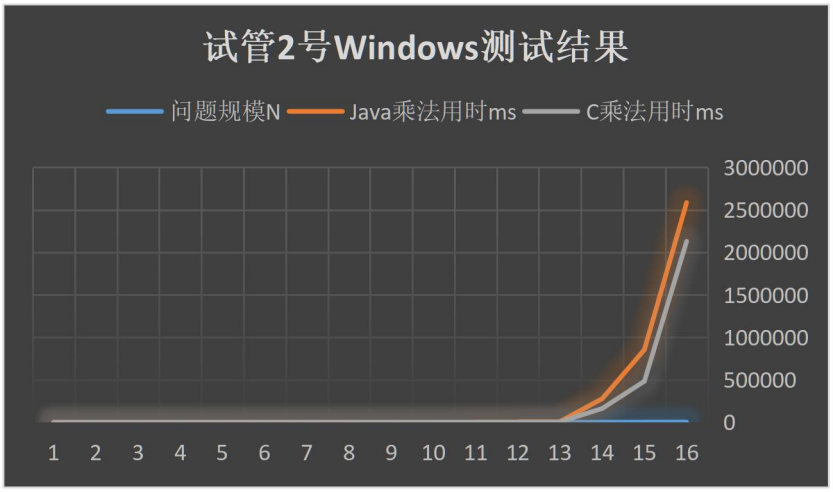


我们继续来看一些有趣的事情，在试管 1 号 Linux 服务器上，Java 乘法和 C 乘法的对比。开始 C 是占上风的，但是，**如果按照我们的趋势线走，似乎 Java 将战胜 C**。诶？Java 可以更强吗？



诶，但是我们看看试管 2 号的结果：C 还是比 Java 强大！这是怎么回事？原来在不同的

硬件平台，不同的操作系统上结果还能不一样的吗！

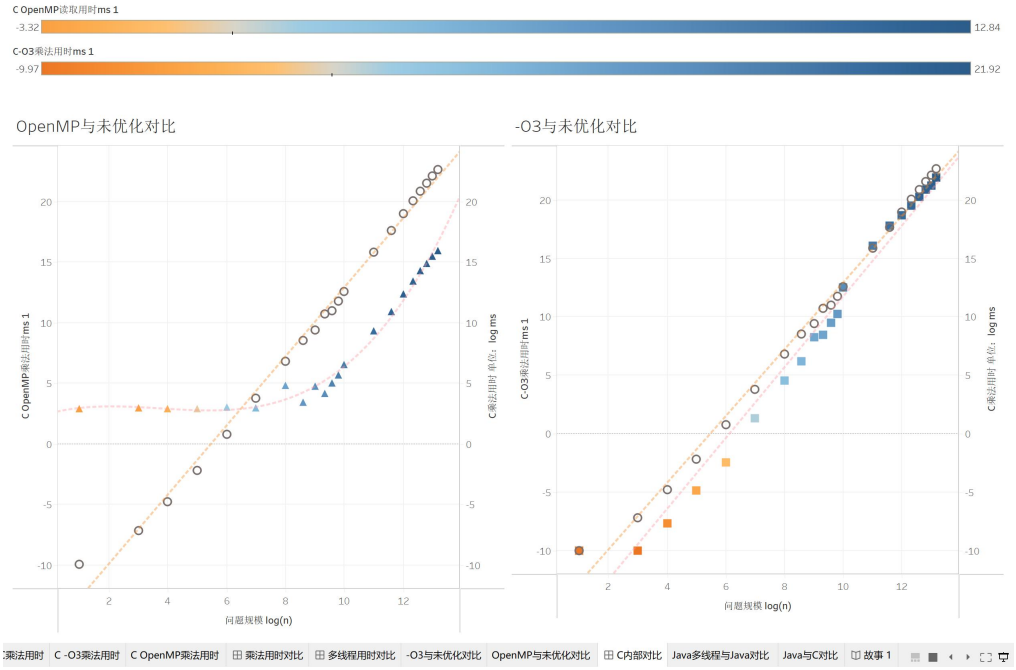


我认为这里的差异跟好几个因素有关：

- 硬件差异：试管 1 号是 AMD CPU，试管 2 号是 Intel CPU，**他们执行着不同的指令集**，面对不同的程序可能有不同的效果。
- 操作系统差异：Windows 的 dll 库和 Linux 的动态库可能有差异，操作系统对程序的管理也可能导致程序在 CPU 运行的 idle time，effect time 有所不同。
- 环境差异：试管 2 号上安装了 MinGW，Intel HPC Base Toolkit，VS，还有从 2008 年到 2019 年全部 C++运行环境和 400MB 的 dll 补全库，以及最新版 Intel MKL，而试管 1 号服务器上的 C 环境可能没有试管 2 号上那么“优秀”（玩太多游戏配的）。
- 编译器差异：两个试管的编译器不同，造成的优化可能也不同。

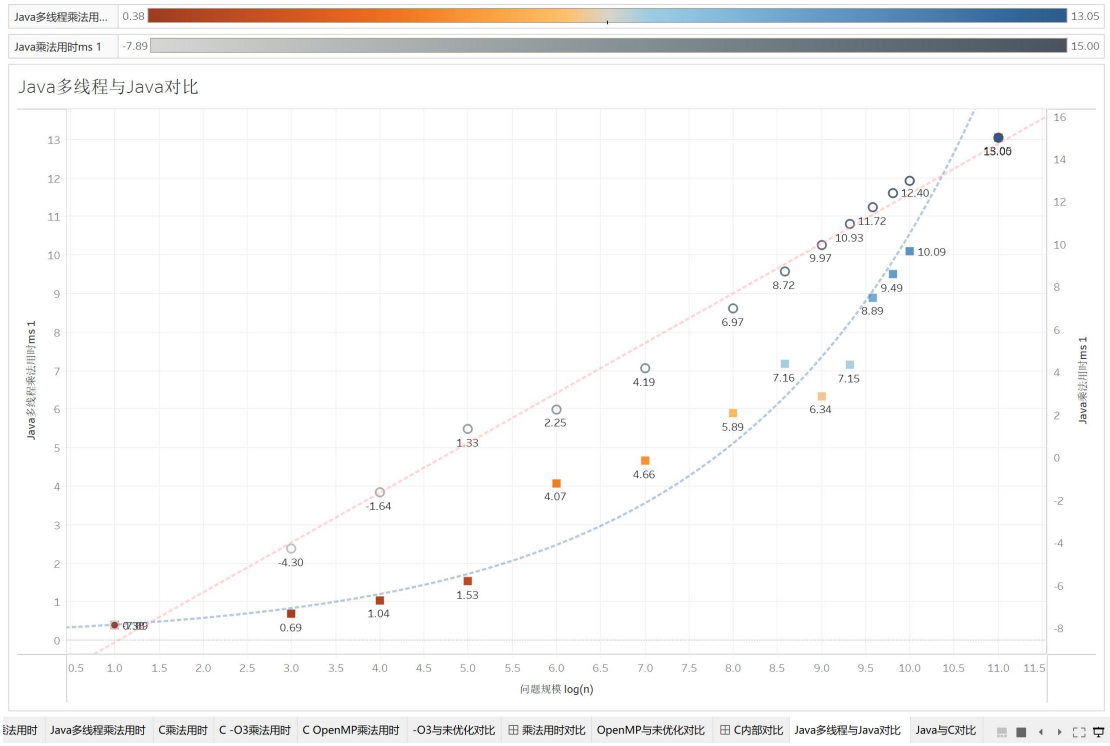
具体发生了哪个，还需要我们进一步探索，这里就给读者留作一道……

我们来看 C 自身的优化：OpenMP 方面，前期 OpenMP 慢于普通 C 程序，推测是多线程开销所致。而后期在多核处理效果快于多线程开销后，OpenMP 就比 C 程序快了。



O3 的也挺有意思的，一开始 O3 的执行速度带来的优化是比较明显的，但是随着问题规模指数的上涨，O3 的优化逐渐缩小了。这并不代表 O3 废了，而是比如说一开始 O3 优化了二分之一，开了 log 很明显，但是后面优化在指数爆炸下就越来越不明显了。这说明 **O3 的优化是类似于接近常数的优化，它不能改变程序的计算复杂度**。

最后我们来看看开了 log 处理的 Java 多线程和 Java。这里我们就让 Java 最多开 4 个线程，相当于最多优化到 4 倍。可以观察到 Java 多线程的优化效果有点像 OpenMP 和 O3 的结合：前期因为创建线程导致速度不如正常 Java，中期因为线程优化大于通信和创建线程，优化效果越来越大。然而多线程是类似于常数上的优化，所以像 O3 一样，多线程 Java 跟普通 Java 越来越接近，直到后面通信的成本让普通 Java 再次成功。



总结

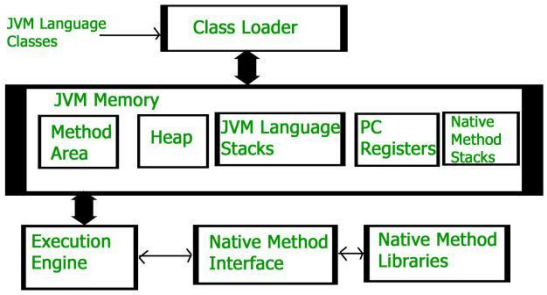
这个实验其实很打破我的想象。首先，我们观测到了 Java 多线程和 C OpenMP 里通信对程序执行时间的影响。第二，平台和环境以及编译器不同，Java 不一定就比 C 慢。接着，OpenMP 和多线程的优化其实是有个区间的，多线程并不意味着就比单线程的快，我们的目标是要找到这个合理的区间。

5.4 运行时间 & 性能瓶颈实验结果

5.4.1 运行时间实验结果

我们选取几个 Profiler 的结果对我们的程序进行分析。首先我们将分析 Java。

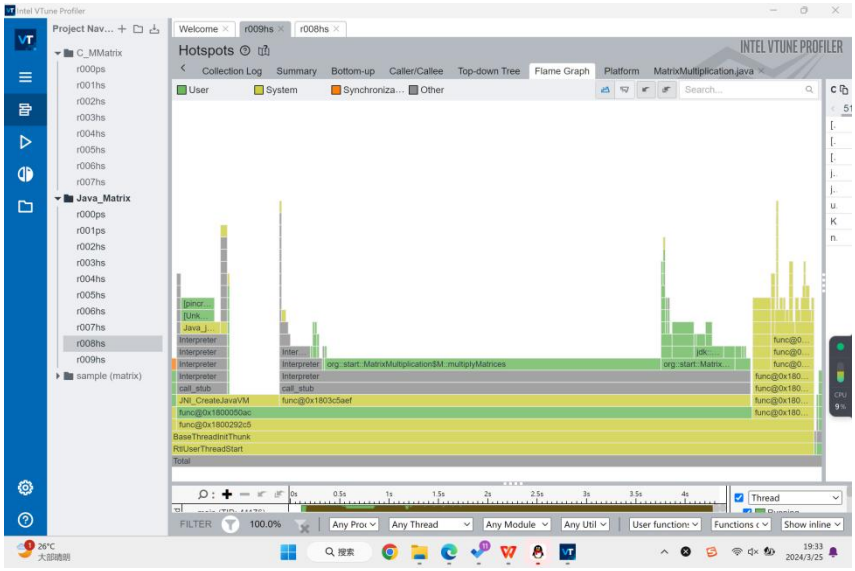
我们根据以往我们的 Java 知识，Java 程序在启动后，随即会启动一个 JVM 进程。JVM 会导入字节码文件，这个过程叫类加载，把类放入到方法区中，并且启动程序计数器，JVM 堆，JVM 栈。随后 JVM 就像一个小 CPU 一样，读取指令，执行程序直到 Main 函数结束。

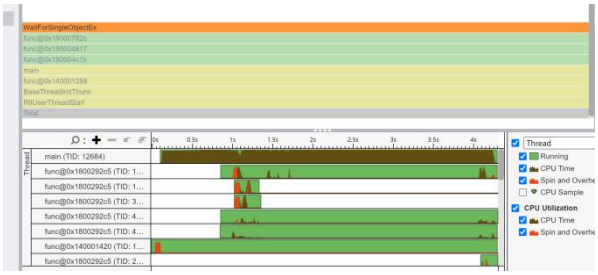


我们看看 Profiler 里程序的运行情况。发现程序在开始时，JVM 并非是直接启动的。反而是 WaitForSingleObjectEx 函数先开始执行（橙色函数部分）。这似乎是操作系统层面同步的一个函数，我们根据微软的介绍，这个函数函数将检查指定对象的当前状态。

https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject

并且此时，一个叫 main 的函数启动，初始化程序。我无法确认 main 是否就是 Java 里的 main，因为此时 JVM 还没有启动，由此我推测，假如这个 JVM 是 C 写的，这可能是启动 JVM 的 C 语言 Main 函数。





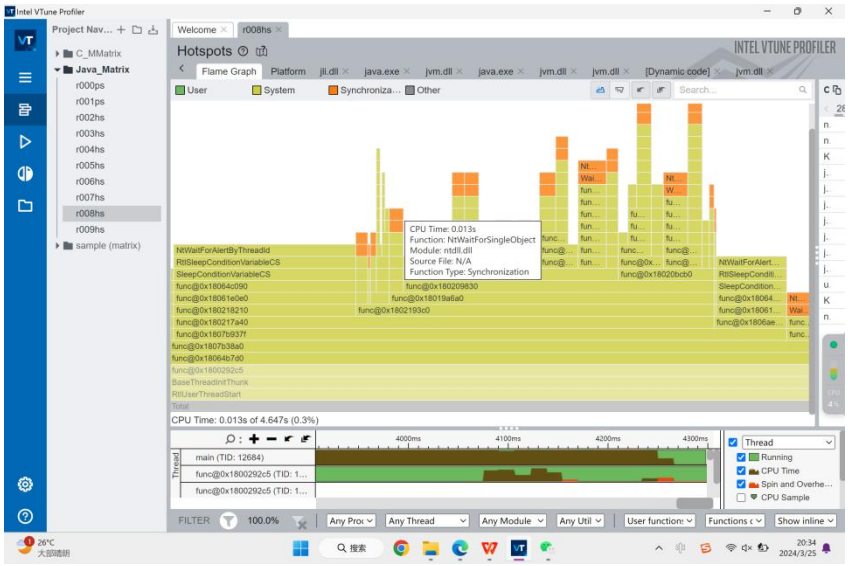
接着是 CreateJavaJVM。虽然它用时很长，但是它的汇编指令就那么短短几行。主要的花费在 call 这条指令上。这个方法我看不到具体的源码，但是根据 Profiler 里 System 的标注，我推测是唤起了系统层内的一个函数。

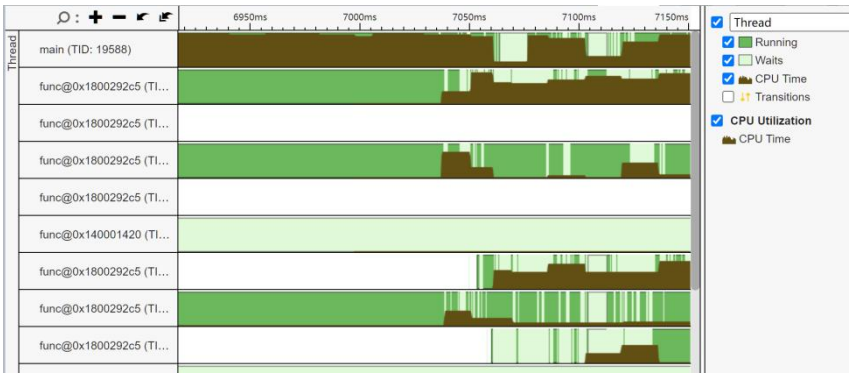
Address ▲	Source Line	Assembly	CPU Time: Total
0x1803d6030		**Block 1:**	
0x1803d6030		sub rsp, 0x38	
0x1803d6034		mov dword ptr [rsp+0x20], 0xffffffff	
0x1803d603c		> call 0x1803d4570	8.4%
0x1803d6041		**Block 2:**	
0x1803d6041		mov dword ptr [rsp+0x20], eax	
0x1803d6045		jmp 0x1803d604b <Block 4>	
0x1803d6047		**Block 3:**	
0x1803d6047		mov eax, dword ptr [rsp+0x20]	
0x1803d604b		**Block 4:**	
0x1803d604b		add rsp, 0x38	
0x1803d604f		ret	

随后 JVM 开始了编译，具体到汇编层面可能就有点难看，对于每个方法，JVM 会动态调用编译方法，将代码编译并存储到相应的地址，随后我们的 Program Counter 就可以一条一条地找到他们。

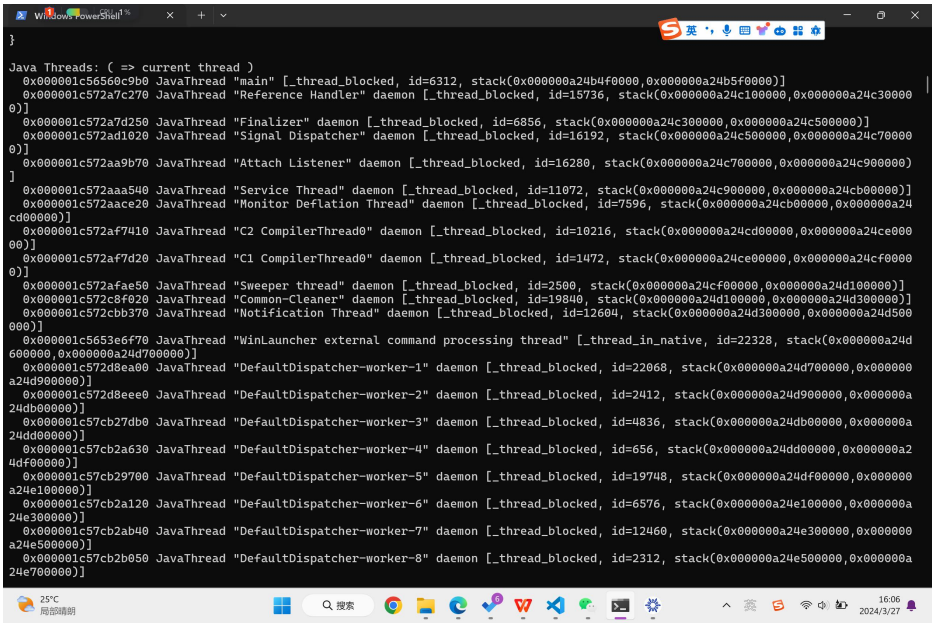
0x23b170cee08	**Block 5218:**	
0x23b170cee08	mov r10, 0x7fff4d6c01a0	
0x23b170cee12	>call r10	5.1%

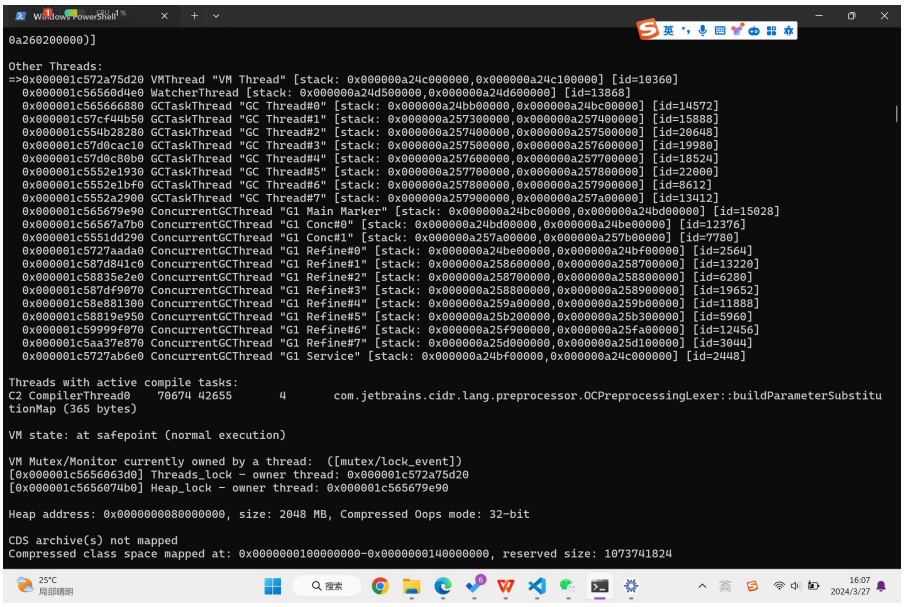
我们接下来看程序终止的时刻，这里有一堆程序在做事情，根据橙色的同步函数，我推测他们是 JVM 在终止自己的虚拟机部分，将不同的类进行终止。这个部分会占用 300ms，所以其实和我们的时间占用并不大。





我们打开 JVM 的日志程序，看看 JVM 的内部启动了什么线程：






```
0x00007ffcbca90000 - 0x00007ffcbcca6000         C:\WINDOWS\SYSTEM32\ntdll.dll
0x00007ffcbaf70000 - 0x00007ffcbb034000         C:\WINDOWS\System32\KERNEL32.DLL
0x00007ffcb9ec0000 - 0x00007ffcba266000         C:\WINDOWS\System32\KERNELBASE.dll
```

可以看到，JVM 执行了我们的熟知的 Main 线程，连带着 Compiler C1,C2，I/O 池，GC 垃圾回收器，导入的动态系统库，以及各种线程的操作记录。这说明刚刚我们的 Vtune 分析是正确的。

我们接下来看 C 的执行过程。可以看到，C 执行的比较纯粹，但是我主要想看这个 ntdll.dll 是什么，它跟我们的程序有什么影响。

根据多方面的信息，ntdll.dll 是重要的 Windows NT 内核级文件。描述了 windows 本地 NTAPI 的接口。我接着看到，像 strcpy 这样的函数都是存在 ntdll.dll 内的。看来它应该像是一个数学库一样的代码库，我们在执行程序时调用了它的 api。那么在 Linux 里怎么办？我想可能还有别的对应的代码库。

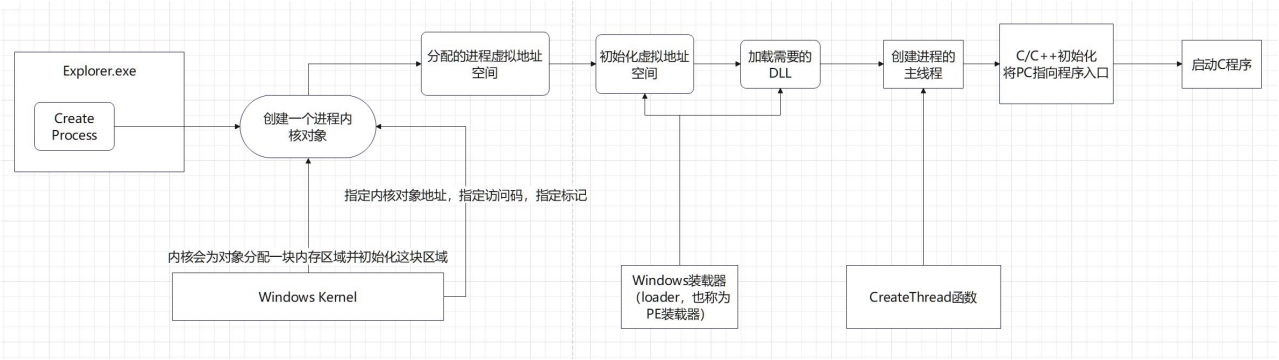
但是 Windows 是怎么启动我们的 C 程序的？下面这个链接里的文章是这么提出的。

https://blog.csdn.net/cpp_mybest/article/details/80194158

当我们启动电脑进入桌面时，系统会创建 Explorer.exe 进程。Explorer.exe 是 Windows 程序管理器 或者叫文件资源管理器，用于管理 Windows 图形壳，删除该程序会导致 Windows 图形界面无法使用。

当双击某个图标时，Explorer.exe 进程的一个线程会侦测到这个操作，它根据注册表中的信息取得文件名，然后 Explorer.exe 以这个文件名调用 CreateProcess 函数。注册表中有相关的项保存着双击操作的信息，如 exe 文件关联、启动 exe 的 Shell 是哪个。PC 中的大多其它的进程都是 Explorer.exe 的子进程，因为它们都是由 Explorer.exe 进程创建的。

接下来，我用一张图片大致总结了 C 程序的流程：Windows 操作系统给程序创建进程线程，并将指针指向程序入口。

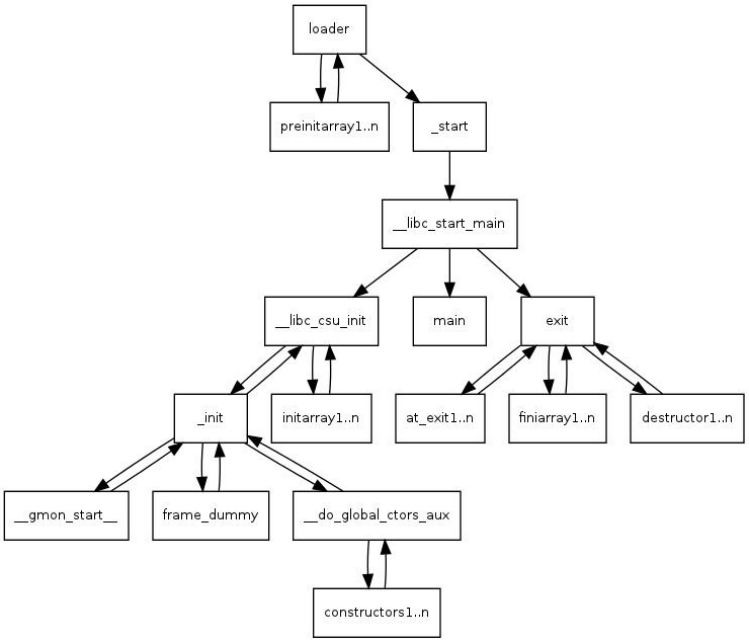


对于一个 C 程序，它在操作系统重将是一个进程块，有进程信息，线程，内存等资源。

那么，Linux 操作系统会怎么做呢？下面这篇文章给出了他的实验结果：

http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

在 Linux 中，每当我们回车输入./Matrix 时，shell 会让 Linux 调用 execve 函数，它将读取 shell 中的输入参数，把他们放到 argc，argv 中，然后 Linux Kernel 会给你像 Windows 一样配置进程，Loader 会给你分配栈。




```
execve() executes the program referred to by pathname.  This causes the program that is currently being run by
the  calling  process  to be replaced with a new program, with newly initialized stack, heap, and (initialized
and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a line of the form:

    #!interpreter [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its command-line arguments.   By  conven-
tion,  the  first  of these strings (i.e., argv[0]) should contain the filename associated with the file being
executed.  The argv array must be terminated by a NULL pointer.  (Thus, in the new program, argv[argc] will be
NULL.)

envp is an array of pointers to strings, conventionally of the form key=value, which are passed as  the  envi-
ronment of the new program.  The envp array must be terminated by a NULL pointer.

This  manual page describes the Linux system call in detail; for an overview of the nomenclature and the many,
often preferable, standardised variants of this function provided by libc, including ones that search the PATH
environment variable, see exec(3).

The argument vector and environment can be accessed by the new program's main function, when it is defined as:

    int main(int argc, char *argv[], char *envp[])
```

接下来，就来到了_start 函数。根据汇编我们可以推测，_start 函数会配置好 eax,esp,edx 以及栈指针，将我们的 argv = %ecx 等等。接下来，它将执行__libc_start_main。

```
080482e0 <_start>:
80482e0:       31 ed                   xor    %ebp,%ebp
80482e2:       5e                      pop    %esi
80482e3:       89 e1                   mov    %esp,%ecx
80482e5:       83 e4 f0                and    $0xfffffff0,%esp
80482e8:       50                      push   %eax
80482e9:       54                      push   %esp
80482ea:       52                      push   %edx
80482eb:       68 00 84 04 08          push   $0x8048400
80482f0:       68 a0 83 04 08          push   $0x80483a0
80482f5:       51                      push   %ecx
80482f6:       56                      push   %esi
80482f7:       68 94 83 04 08          push   $0x8048394
80482fc:       e8 c3 ff ff ff          call   80482c4 <__libc_start_main@plt>
8048301:       f4                      hlt
```

值	__libc_start_main参数	内容
%eax	未知	不关心
%esp	void (*stack_end)	已被对齐的栈指针
%edx	void (*rtld_fini)(void)	加载器传到edx中的动态链接器的析构函数。 被__libc_start_main函数通过__cxat_exit()注册， 为我们已经加载的动态库调用FINI section
0x8048400	void (*fini)(void)	__libc_csu_fini——程序的析构函数。 被__libc_start_main 通过 __cxat_exit()注册
0x80483a0	void (*init)(void)	__libc_csu_init——程序的构造函数。 于main函数之前被__libc_start_main函数调用
%ecx	char **ubp_av	argv相对栈的偏移值
%esi	arcg	argc相对栈的偏移值
0x8048394	int(*main)(int, char**, char**)	我们程序的main函数，被__libc_start_main函数调用 main函数的返回值被传递给exit(), 由其控制我们的程序

__libc_start_main 程序会在启动程序主线程，传入我们的 argc 和 argv 函数，调用 glibc（GNU C Library），存储环境变量，根据指针地址启动 Main 函数。

1. 处理关于setuid、setgid程序的安全问题
2. 启动线程
3. 注册用户程序的 `fini` 和 `rtld_fini` 参数，然后被at_exit调用，从而完成用户程序和加载器的负责清理工作的函数
4. 调用其 `init` 参数
5. 调用 `main` 函数，并把 `argc` 和 `argv` 参数、环境变量传递给它
6. 调用 `exit` 函数，并将main函数的返回值传递给它

来源：https://zhuanlan.zhihu.com/p/52054044

到这里，我们的 C 程序就基本上进入到 Main 函数里了。

我们可以看到，在本次实验中，跟 Java 相比，C 的执行更加地直接和迅速，所涉及的程序也是直接在操作系统层而非虚拟机层上。同时 C 的代码会被 CPU 直接执行，而 Java 代码会在编译中共同进行。

5.4.2 程序瓶颈试分析

通过这几个实验，我们现在知道了我们 C 和 Java 程序的基本上全部的运行流程。那么，假如我想提高我们的程序的效率，我应该从哪些方面分析我们程序的瓶颈，让我们的程序加速呢？这里我们还是选择 1000 * 1000 的矩阵进行查看。

传统的分析方法是分析程序的 CPU 执行情况，分析我们的内存状况，分析 IO。我们刚刚通过 Profiler 查看程序执行也基本上是服从这个传统分析方法。通过实验 2 我们知道，Java 在执行程序上对字节码的优化还有所乏力，那么我们的 **Java 程序瓶颈可能就在 CPU 方面**。在 C 程序中我们的 I/O 读取判断逻辑似乎没有 Java 写的那么优秀，那么我们 **C 程序的 I/O 读取逻辑可能就需要提升**。而如果说我们的矩阵继续变大，大到我们的内存不够用时，**矩阵增大将导致内存可能会变成新的瓶颈**，操作系统很有可能会开启内存虚拟化技术，将我们的一部分矩阵存储在系统盘中，从而让我们的程序成功执行，并且使得我们的时间增加的系数增加。

对于矩阵范围增大导致的内存瓶颈，在软件上我们似乎没有太多能优化的地方，更直接的解决方案是换更快更宽更大的 SSD，插上 PCIe4.0。但是，像 CPU 方面的优化我们就可以

动许多手脚。此时我们已经进入比较微观的视角，需要一套新的分析方案。

这里我找到了 Ahmand Yasin 的 IEEE 论文“A top-down method for performance analysis and counter architercture”。这是 Intel 公司提出的一套方法论叫 **Top-Down** 模型，它让每个 CPU 微指令对应不同的**系统微资源**的利用依赖度。他们建立了 4 种系统微资源的指令倾向类型，然后用 4 种类型去评估我们的 CPU 微指令。最终，我们汇总一个程序所有的微指令，让一个应用程序展现出 4 种不同的倾向性中的一种。使我们对程序在多核、高频、缓存上的使用有新的理解。

根据论文，这四种分别是：

- Frontend bound（前端依赖）首先需要注意的是这里的前端并不是指 UI 的前端，这里的前端指的是 x86 指令解码阶段的耗时。
- Backend bound（后端依赖）同样不同于其他“后端”的定义，这里指的是传统的 CPU 负责处理实际事务的能力。由于这一个部分相对其他部分来说，受程序指令的影响更为突出，这一块又划分出了两个分类。core bound（核心依赖）意味着系统将会更多的依赖于微指令的处理能力。memory bound（存储依赖）我这里不把 memory 翻译成内存的原因在于这里的 memory 包含了 CPU L1～L3 缓存的能力和传统的内存性能。
- Bad speculation（错误的预测）这一部分指的是由于 CPU 乱序执行预测错误导致额外的系统开销。
- Retiring（拆卸）字面理解是退休的意思，事实上这里指的是指令完成、等待指令切换，模块重新初始化的开销。

在这四种模型上，Intel 工程师们将不同的 Bound 细分到各个模型中，比如分支预测错误归结到 Bad speculation, Memory Bound 归结到 Backend Bound。具体见下图。

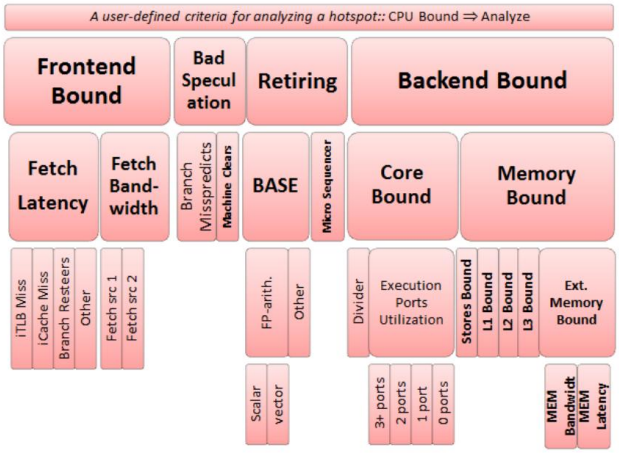


Figure 2: The Top-Down Analysis Hierarchy

论文认为，我们在寻找程序的最大瓶颈时，可以像**树搜索**一样，先从 4 种程序瓶颈里找到最大的，然后再在这一种程序瓶颈里再找到问题最严重的，随后一步一步解决问题。比如在文章中，作者先给出 4 种瓶颈的判断方法，然后再细分查看各种瓶颈，比如 Backend Bound 里可能会有 L1,L2,L3 缓存的 Bound，我们在确定分析 Backend 的情况下分析起来定位问题就会比较高效。

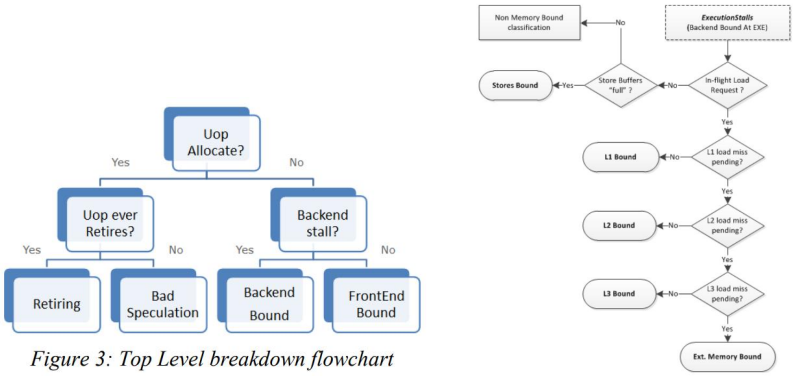
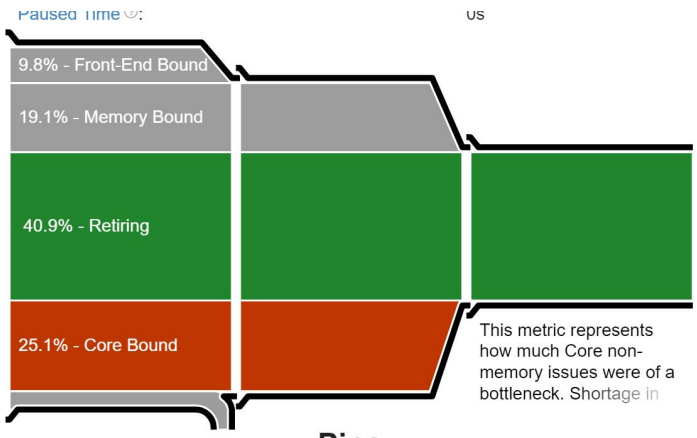
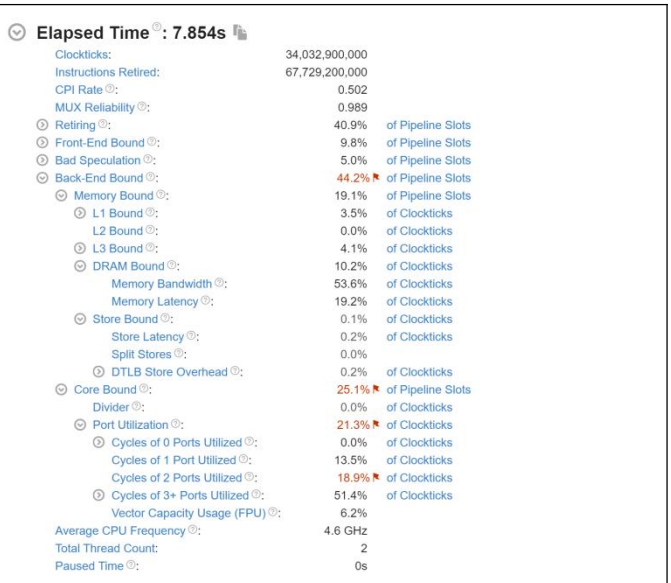


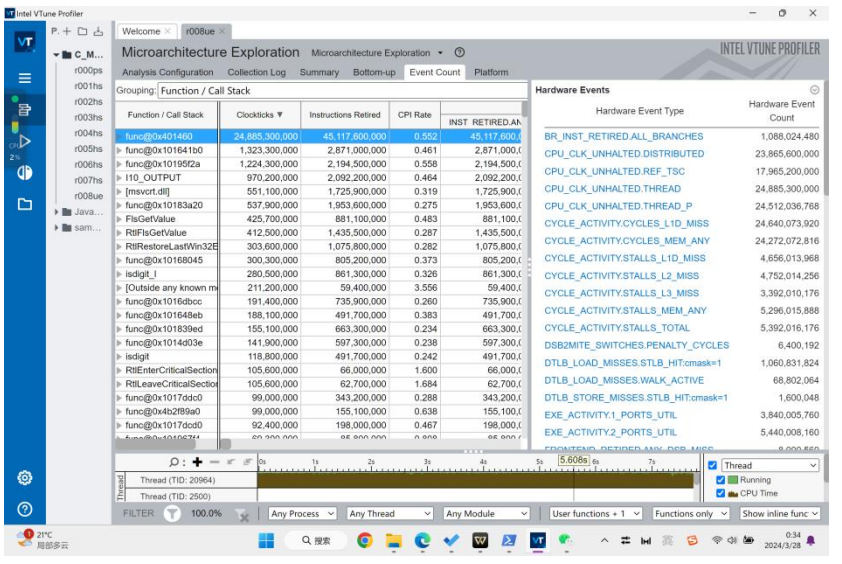
Figure 3: Top Level breakdown flowchart

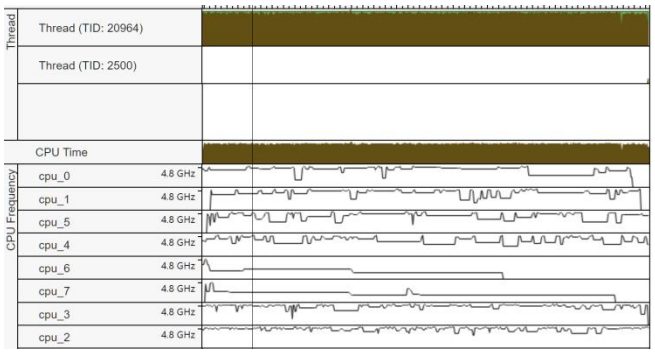
一位在英特尔中国的软件工程师[10]提出他们在 Vtune 上实现了这一模型。我随后在电脑上安装了完整的 OneAPI HPC Toolkit 查看我们的程序。

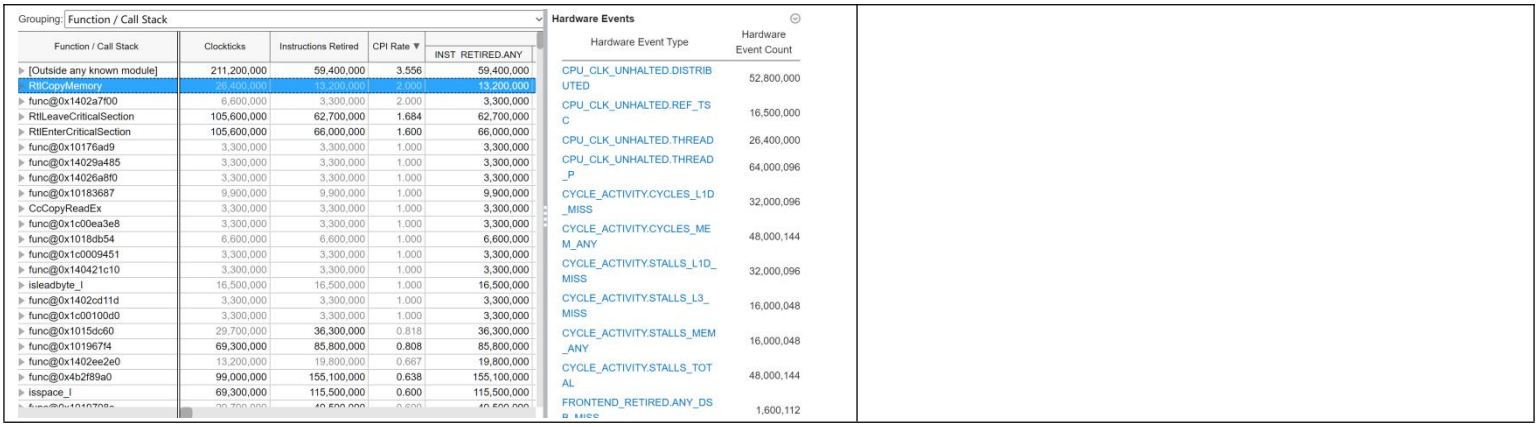
我首先对我们的直接编译的 C 程序进行查看，Vtune 返回了很多结果，从四种模型的 Bound，程序的 CPI，各种细分的 Bound，不同函数的所用时钟数，指令数，CPI，CPU 频率。











Grouping: Function / Call Stack

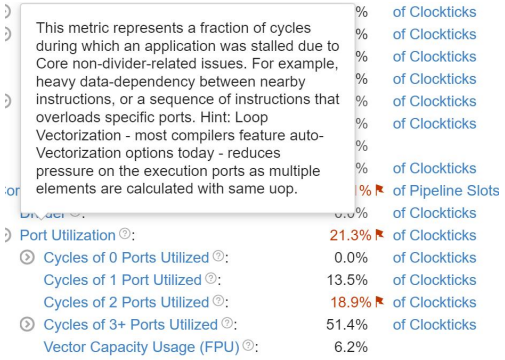
Function / Call Stack	Clockticks	Instructions Retired	CPI Rate ▼	INST_RETIRED.ANY
[Outside any known module]	211,200,000	59,400,000	3.556	59,400,000
RtlCopyMemory	26,400,000	13,200,000	2.000	13,200,000
func@0x1402a7f00	6,600,000	3,300,000	2.000	3,300,000
RtlLeaveCriticalSection	105,600,000	62,700,000	1.684	62,700,000
RtlEnterCriticalSection	105,600,000	66,000,000	1.600	66,000,000
func@0x10176ad9	3,300,000	3,300,000	1.000	3,300,000
func@0x14029a485	3,300,000	3,300,000	1.000	3,300,000
func@0x14026a8f0	3,300,000	3,300,000	1.000	3,300,000
func@0x10183687	9,900,000	9,900,000	1.000	9,900,000
CcCopyReadEx	3,300,000	3,300,000	1.000	3,300,000
func@0x1c00ea3e8	3,300,000	3,300,000	1.000	3,300,000
func@0x1018db54	6,600,000	6,600,000	1.000	6,600,000
func@0x1c0009451	3,300,000	3,300,000	1.000	3,300,000
func@0x140421c10	3,300,000	3,300,000	1.000	3,300,000
isleadbyte_l	16,500,000	16,500,000	1.000	16,500,000
func@0x1402cd11d	3,300,000	3,300,000	1.000	3,300,000
func@0x1c00100d0	3,300,000	3,300,000	1.000	3,300,000
func@0x1015dc60	29,700,000	36,300,000	0.818	36,300,000
func@0x10196714	69,300,000	85,800,000	0.808	85,800,000
func@0x1402ee2e0	13,200,000	19,800,000	0.667	19,800,000
func@0x4b2f89a0	99,000,000	155,100,000	0.638	155,100,000
isspace_l	69,300,000	115,500,000	0.600	115,500,000

Hardware Events

Hardware Event Type	Hardware Event Count
CPU_CLK_UNHALTED.DISTRIBUTED	52,800,000
CPU_CLK_UNHALTED.REF_TSC	16,500,000
CPU_CLK_UNHALTED.THREAD	26,400,000
CPU_CLK_UNHALTED.THREAD_P	64,000,096
CYCLE_ACTIVITY.CYCLES_L1D_MISS	32,000,096
CYCLE_ACTIVITY.CYCLES_MEM_ANY	48,000,144
CYCLE_ACTIVITY.STALLS_L1D_MISS	32,000,096
CYCLE_ACTIVITY.STALLS_L3_MISS	16,000,048
CYCLE_ACTIVITY.STALLS_MEM_ANY	16,000,048
CYCLE_ACTIVITY.STALLS_TOTAL	48,000,144
FRONTEND_RETIRED.ANY_DSB_MISS	1,600,112

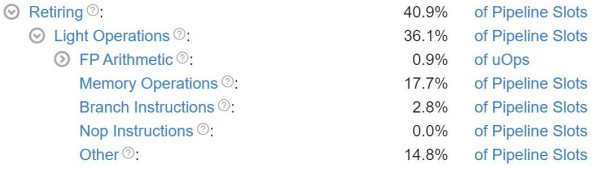
直接编译的 C 程序的 Top-Down 分析结果

Profiler 认为在程序中的最大瓶颈是 Core Bound（红色高亮）这属于 Backend Bound。我们查看 Backend 里的性能瓶颈，比较严重的是 Port Utilization，我们可以查看 Profiler 对这个的解释。它提出我们的端口之所以成为严重的瓶颈，很可能是相邻指令之间存在大量数据依赖性，或者一系列指令过度占用特定端口。接着它还提到一个提示提示：循环向量化 - 如今大多数编译器都具备自动向量化选项 - 可以减少对执行端口的压力，因为多个元素使用相同的 uop 进行计算。



回忆我们实验 2 所执行的直接编译的汇编代码，不难想到这部分描述的就是 ijk 三变量的瓶颈，他们由于存在内存中使得 CPU 花费很多的时间进行存储和读取。

从四种模型中我们也去查看 Retiring 的瓶颈，可以发现程序在取址和内存操作上是比较严重的。他们的完成速度慢，从而拖累了指令的 Retiring。根据实验 2 的汇编代码，我们可以推测因为大部分的取址是发生在 ijk 上，因而产生了这个瓶颈。



如果分析我们的具体各个函数，我们发现 CPI 最高的是 copyMemory 操作，这一点不难想象。而我们的主程序的 CPI 在 0.552。主程序的瓶颈主要是 Backend Bound，而这里边一个是 Memory Bound 的 DRAM Bound。不难推测出，这是我们的矩阵存储的地方，CPU 与 DRAM 的 I/O 自然会成为程序的重要瓶颈，如果我的电脑用了很多程序，使得 Intel 开启了内存虚拟化技术，我们的程序可能会跑得更慢。综上，I/O 使得 CPU 的端口使用比较紧张。不过想到这里，我想到我的电脑 CPU 是 UMA 架构，如果对于服务器里的 NUMA 架构，CPU 的 Bound 会不会增加呢？时间原因我没有测试。~~这个问题给读者留作练习（不是）。~~

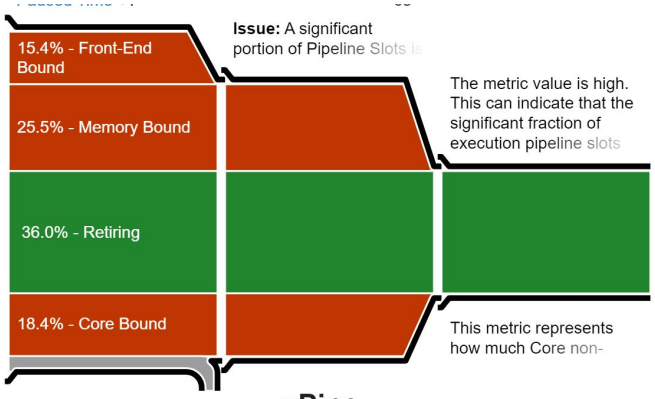
Function / Call Stack	und	Bad Speculation	Back-End Bound	Average CPU Frequency	Mod
func@0x401460	3.2%	1.6%	57.8%	4.6 GHz	matr.exe
[No call stack informatio	3.2%	1.6%	57.8%	4.6 GHz	
func@0x101641b0	40.6%	15.8%			msvcrt.dl
func@0x10195f2a	39.1%	33.3%			svcrt.dl
I10_OUTPUT	23.9%	39.4%			svcrt.dl
func@0x10183a20	12.0%	0.0%			svcrt.dl
[msvcrt.dll]	11.6%	21.9%			svcrt.dl
RtlFlsGetValue	9.3%	16.0%			dll.dll
RtlRestoreLastWin32Error	21.8%	16.8%			dll.dll
FlsGetValue	25.1%	0.0%			rnelba
isdigit_l	17.6%	0.0%			svcrt.dl
func@0x10168045	13.0%	11.1%			svcrt.dl
func@0x1016dbcc	18.1%	0.0%			svcrt.dl
func@0x101839ed	6.7%	12.6%			svcrt.dl
func@0x1014d03e	3.3%	27.4%			svcrt.dl
func@0x101648eb	32.0%	0.0%	0.0%	4.1 GHz	msvcrt.dl
isdigit	9.5%	11.3%	15.8%	2.3 GHz	msvcrt.dl
func@0x1017ddc0	37.3%	0.0%	0.0%	4.5 GHz	msvcrt.dl
func@0x1014e561	8.9%	37.8%	0.0%	2.8 GHz	msvcrt.dl
func@0x1017dcd0	15.2%	16.2%	0.0%	3.3 GHz	msvcrt.dl
func@0x101769b6	50.9%	20.0%	0.0%	1.5 GHz	msvcrt.dl
strcpy_s	94.1%	0.0%	0.0%	9.9 GHz	msvcrt.dl
func@0x1013474f	19.2%	42.4%	0.0%	4.6 GHz	msvcrt.dl

A significant portion of pipeline slots are remaining empty. When operations take too long in the back-end, they introduce bubbles in the pipeline that ultimately cause fewer pipeline slots containing useful work to be retired per cycle than the machine is capable to support. This opportunity cost results in slower execution. Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port (for example, more multiply operations arriving in the back-end per cycle than the execution unit can support).

µPipe

Retiring:	**37.4%**
Front-End Bound:	**3.2%**
Bad Speculation:	**1.6%**
Back-End Bound:	**57.8%**
Memory Bound:	**27.1%**
L1 Bound:	**2.6%**
L2 Bound:	0.0%
L3 Bound:	**5.5%**
L3 Latency:	0.0%
SQ Full:	0.0%
DRAM Bound:	**13.6%**
Memory Bandwidth:	73.2%
Memory Latency:	25.7%
Store Bound:	**0.0%**
Core Bound:	**30.6%**
Divider:	0.0%
Port Utilization:	**23.6%**
Cycles of 0 Ports Utilized:	**0.0%**
Cycles of 1 Port Utilized:	15.4%
Cycles of 2 Ports Utilized:	21.9%
Cycles of 3+ Ports Utilized:	**42.2%**
Vector Capacity Usage (FPU):	6.2%

那我们接下来将**测试-03 下的程序**。

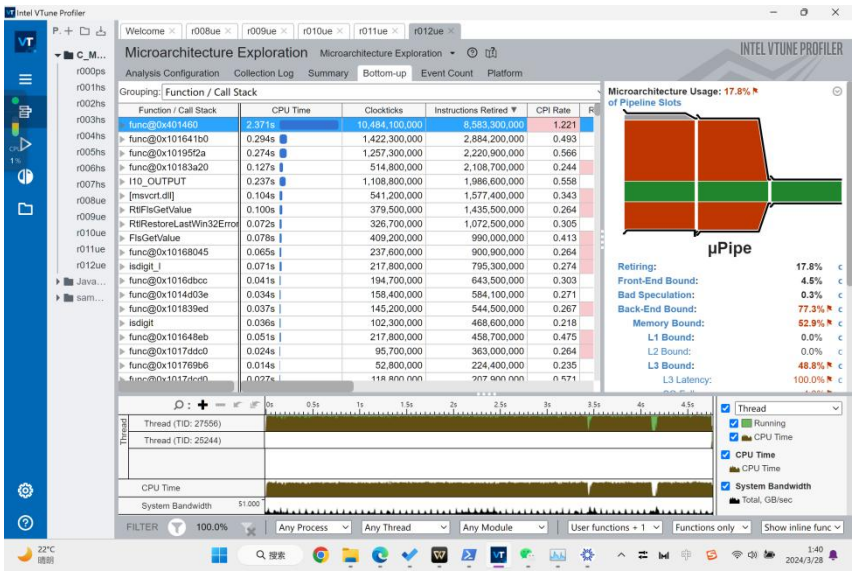


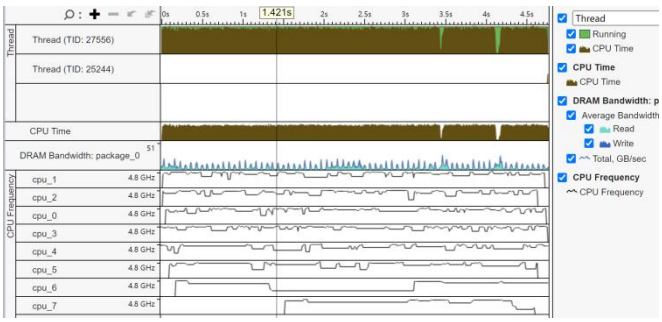
µPipe

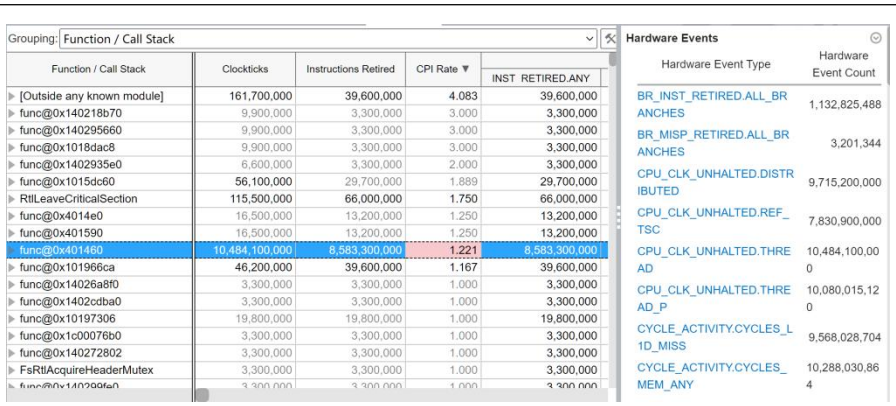
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Elapsed Time: 4.763s

Clockticks:	19,611,900,000	
Instructions Retired:	31,221,300,000	
CPI Rate:	0.628	
MUX Reliability:	0.979	
Retiring:	36.0%	of Pipeline Slots
Front-End Bound:	15.4%	of Pipeline Slots
Front-End Latency:	5.2%	of Pipeline Slots
Front-End Bandwidth:	10.2%	of Pipeline Slots
Front-End Bandwidth MITE:	4.9%	of Pipeline Slots
Front-End Bandwidth DSB:	4.8%	of Pipeline Slots
Front-End Bandwidth LSD:	3.6%	of Pipeline Slots
(Info) DSB Coverage:	51.5%	
(Info) LSD Coverage:	29.6%	
(Info) DSB Misses:	100.0%	of Pipeline Slots
Bad Speculation:	4.6%	of Pipeline Slots
Back-End Bound:	44.0%	of Pipeline Slots
Memory Bound:	25.5%	of Pipeline Slots
L1 Bound:	1.6%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	26.1%	of Clockticks
L3 Latency:	87.8%	of Clockticks
SQ Full:	2.1%	of Clockticks
DRAM Bound:	5.3%	of Clockticks
Store Bound:	0.2%	of Clockticks
Core Bound:	18.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	18.2%	of Clockticks
Average CPU Frequency:	4.4 GHz	
Total Thread Count:	2	
Paused Time:	0s	







Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate ▼	INST_RETIRED.ANY
[Outside any known module]	161,700,000	39,600,000	4.083	39,600,000
func@0x140218b70	9,900,000	3,300,000	3.000	3,300,000
func@0x140295660	9,900,000	3,300,000	3.000	3,300,000
func@0x1018dac8	9,900,000	3,300,000	3.000	3,300,000
func@0x1402935e0	6,600,000	3,300,000	2.000	3,300,000
func@0x1015dc60	56,100,000	29,700,000	1.889	29,700,000
RtlLeaveCriticalSection	115,500,000	66,000,000	1.750	66,000,000
func@0x4014e0	16,500,000	13,200,000	1.250	13,200,000
func@0x401590	16,500,000	13,200,000	1.250	13,200,000
func@0x401460	10,484,100,000	8,583,300,000	1.221	8,583,300,000
func@0x101966ca	46,200,000	39,600,000	1.167	39,600,000
func@0x14026a8f0	3,300,000	3,300,000	1.000	3,300,000
func@0x1402cdba0	3,300,000	3,300,000	1.000	3,300,000
func@0x10197306	19,800,000	19,800,000	1.000	19,800,000
func@0x1c00076b0	3,300,000	3,300,000	1.000	3,300,000
func@0x140272802	3,300,000	3,300,000	1.000	3,300,000
FsRtlAcquireHeaderMutex	3,300,000	3,300,000	1.000	3,300,000

Hardware Events

Hardware Event Type	Hardware Event Count
BR_INST_RETIRED.ALL_BRANCHES	1,132,825,488
BR_MISP_RETIRED.ALL_BRANCHES	3,201,344
CPU_CLK_UNHALTED.DISTRIBUTED	9,715,200,000
CPU_CLK_UNHALTED.REF_TSC	7,830,900,000
CPU_CLK_UNHALTED.THREAD	10,484,100,000
CPU_CLK_UNHALTED.THREAD_P	10,080,015,120
CYCLE_ACTIVITY.CYCLES_L1D_MISS	9,568,028,704
CYCLE_ACTIVITY.CYCLES_MEM_ANY	10,288,030,864

开了-O3 的 C 程序的 Top-Down 分析结果

从-O3 的结果上看，我们的程序在 FrontEnd，MemoryBound，和 BackEnd 全成了瓶颈。这听起来可能不太妙，不过往另一个角度想，是不是因为优化的很厉害，所以才使得原本不是瓶颈的 Bound 变成瓶颈呢？

不过，这回 MemoryBound 里的具体瓶颈不再是 DRAM 的瓶颈，相反是 L3 Cache 的瓶颈。我认为这是编译器优化的结果。原本程序将数据存储在 DRAM 中，但是-O3 会改变存储位置，尝试将数据导入到 L3 中。不过，这应该和我们的矩阵的问题规模有关，因为我们的 L3 大小只有 12MB，如果继续增加 N 维数，DRAM 可能还是会变成一个瓶颈。

虚拟化:	已启用
L1 缓存:	320 KB
L2 缓存:	5.0 MB
L3 缓存:	12.0 MB

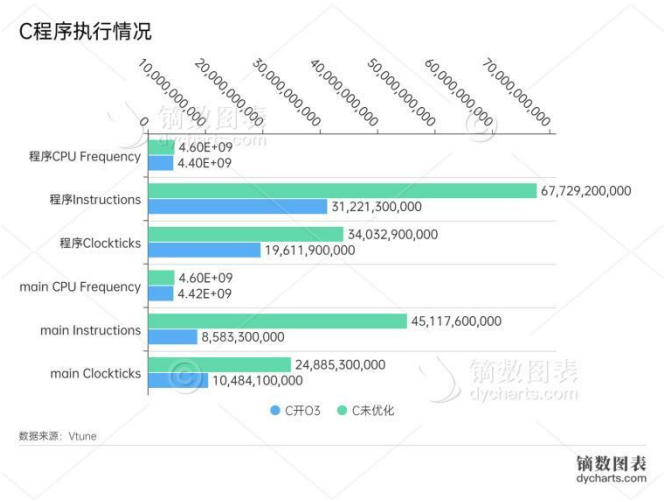
FrontEnd 的瓶颈似乎出现在各种 Cache 上。时间原因我就没有分析。

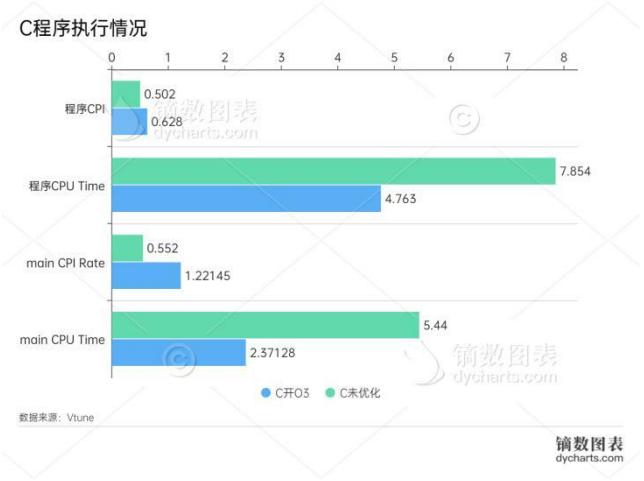
Backend 的 Core Bound 里 CPU 的端口瓶颈下降了，从 21.3%到 18.4%。

Core Bound ⓘ:	18.4%	of Pipeline Slots
Divider ⓘ:	0.0%	of Clockticks
Port Utilization ⓘ:	18.2%	of Clockticks

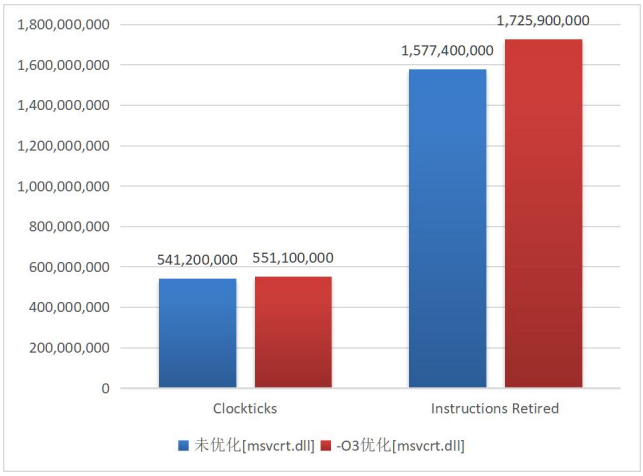
我们比较两个程序的执行情况。可以看到，**使用了 O3 以后程序和 main 函数的 CPI 反而增加了，也就是说程序执行一条汇编代码所花的平均时间反而变长了**，但是程序的总 instructions 几乎减少了一半，main 函数的 instruction 几乎减少到原来的 20%。这使得最后的 CPU 时钟花费在未优化的 42%。

由此我们可以得出 CPI 增长的原因：**程序的指令数大量减少，使得花费时间减少**。但是减少的指令很多执行的速度是小于 CPI 的，比如我们在实验 2 中所看到的，在我们操作 ijk 的时候，我们用了很多 add 指令。这些指令**花费的时间短，并且可以被优化**。因而在开启编译器优化后，剩下来的指令所需要的平均时间增加，CPI 也就增加了。这说明我在实验 2 的推测并不完全正确。其实从白老师做的实验也能看到，优化后的程序普遍 CPI 都发生了增长。

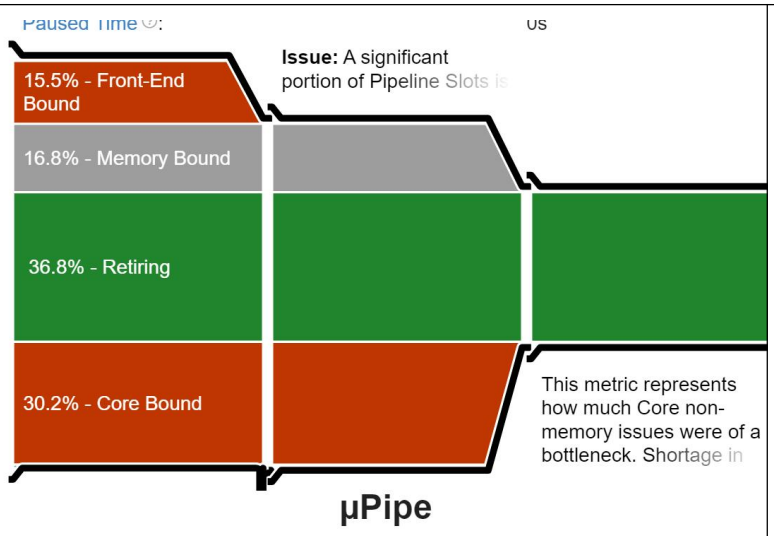




对比在 dll 库中使用的指令，-O3 同样也进行了优化，程序用 dll 的汇编代码数也下降了。

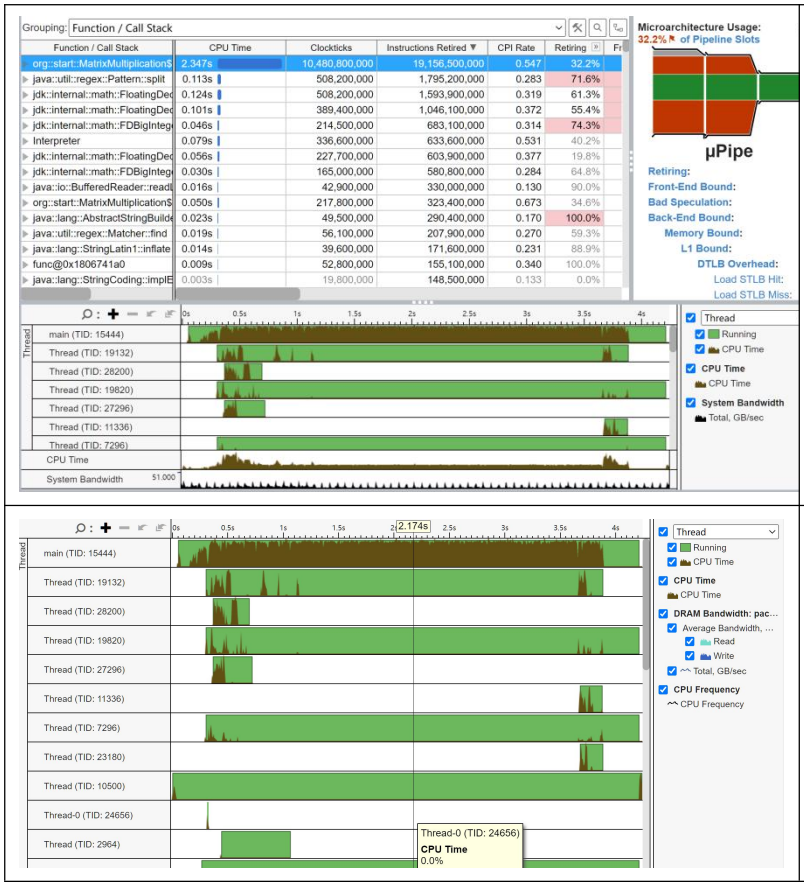


接下来我们查看 **Java** 的运行情况。



Elapsed Time: 4.240s

Clockticks:	17,915,700,000	
Instructions Retired:	33,270,600,000	
CPI Rate:	0.538	
MUX Reliability:	0.984	
Retiring:	36.8%	of Pipeline Slots
Front-End Bound:	15.5%	of Pipeline Slots
Bad Speculation:	0.8%	of Pipeline Slots
Back-End Bound:	46.9%	of Pipeline Slots
Memory Bound:	16.8%	of Pipeline Slots
Core Bound:	30.2%	of Pipeline Slots
Divider:	0.4%	of Clockticks
Port Utilization:	29.2%	of Clockticks
Cycles of 0 Ports Utilized:	0.2%	of Clockticks
Cycles of 1 Port Utilized:	20.4%	of Clockticks
Cycles of 2 Ports Utilized:	23.4%	of Clockticks
Cycles of 3+ Ports Utilized:	32.9%	of Clockticks
Vector Capacity Usage (FPU):	6.2%	
Average CPU Frequency:	4.3 GHz	
Total Thread Count:	31	
Paused Time:	0s	



Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	INST RETI
org::start::MatrixMultiplication$M::mu	10,480,800,000	19,156,500,000	0.547	19,15
jdk::internal::math::FloatingDecimal$	508,200,000	1,593,900,000	0.319	1,59
java::util::regex::Pattern::split	508,200,000	1,795,200,000	0.283	1,79
jdk::internal::math::FloatingDecimal::	389,400,000	1,046,100,000	0.372	1,04
Interpreter	336,600,000	633,600,000	0.531	63
jdk::internal::math::FloatingDecimal$	227,700,000	603,900,000	0.377	60
org::start::MatrixMultiplication$M::wr	217,800,000	323,400,000	0.673	32
jdk::internal::math::FDBigInteger::lef	214,500,000	683,100,000	0.314	68
jdk::internal::math::FDBigInteger::va	165,000,000	580,800,000	0.284	58
func@0x180356320	69,300,000	89,100,000	0.778	8
jdk::internal::math::FDBigInteger::lef	59,400,000	141,900,000	0.419	14
java::util::regex::Matcher::find	56,100,000	207,900,000	0.270	20
func@0x310e0	52,800,000	3,300,000	16.000	
func@0x1806741a0	52,800,000	155,100,000	0.340	15
func@0x18034ff00	49,500,000	39,600,000	1.250	3
java::lang::AbstractStringBuilder::ap	49,500,000	290,400,000	0.170	29

Java 程序的 Top-Down 分析结果

我们可以看到，Java 的瓶颈会出现在 Core 上，并且还是 CPU 端口的问题，这个问题就跟他汇编代码一样，程序进行了很多次 call，并将数据存储在 DRAM 中，因而出现了性能瓶颈。但是令人惊叹的是，Java 程序在 I/O 上速度比 C 要好，以至于整个时间是比 C 程序快的，这可能是因为 Java jdk 内部的函数被优化的很好，并且比我写的 C 好。在矩阵乘法函数上 Java 的 CPI 似乎就比较低，和 C 未优化的效果是一样的。我们可以猜测这里 Java 的编译器就直接像 C 一样翻译代码，从而效果就比较普通。

值得一提的是，Intel 官方也拿矩阵乘法做了测试，对不同的矩阵算法进行了分析，我们也可以对比他们的结果看看。

Metric	multiply1	multiply2	multiply3
Speedup	1.0x	11.8x	16.5x
IPC	0.17	1.19	0.80
Frontend Bound	0.00	0.07	0.02
Retiring	0.05	0.41	0.28
Bad Speculation	0.00	0.00	0.00
Backend Bound	0.95	0.52	0.70
├─+ Memory Bound	0.84	0.12	0.31
│ ├── L1 Bound	0.05	0.07	0.03
│ ├── L2 Bound	0.03	-	0.05
│ ├── L3 Bound	0.05	-	0.01
│ ├── MEM Bound	0.71	0.07	0.21
│ └── Stores Bound	-	-	-
└─+ Core Bound	0.15	0.64	0.55
├── Divider	-	-	-
└── Ports Utiliz.	0.15	0.64	0.55

小结

所以综上我们看到，未优化的 C 程序在端口取地址上存在瓶颈，同时未优化程序的汇编指令多，执行的速度就比较慢。而开了 O3 的 C 程序将 DRAM 存储主要改为了 L3 存储和寄存器存储，性能有了许多优化，但是可能在 I/O 方面函数写的比较朴素，函数依旧花费了比较多的时间。Java 程序和未优化的 C 程序类似，主要的瓶颈也是在汇编指令数和端口地址上，对汇编和地址的优化没有 C 编译器那么强大。

Part 6: 实验背后的理论

6.1 矩阵乘法的发展

这部分来源于我自己对各个文章的整理，可以作为我们的研究综述开头。

矩阵乘法是一门近代研究方向。1812 年，Binet 和 Cauchy 发现了最初了行列式乘法。1858 年，Cayley 在他们的基础上，研究出了我们今天线性代数课上所讲的矩阵乘法：result[i][j] += matrix1[i][k] * matrix2[k][j]; 如果你的研究方向不是理论计算机科学或者计算数学，这些知识足够应对现实的各种问题。

200 BC: Han dynasty, coefficients are written on a counting board [6]
1545 Cardan: Cramer rule for 2x2 matrices. [6]
1683 Seki and Leibnitz independently first appearance of Determinants [6]
1750 Cramer (1704-1752) rule for solving systems of linear equations using determinants [8]
1764 Bezout rule to determine determinants
1772 Laplace expansion of determinants
1801 Gauss first introduces determinants [6]
1812 Cauchy multiplication formula of determinant. Independent of Binet
1812 Binet (1796-1856) discovered the rule det(AB) = det(A) det(B) [1]
1826 Cauchy Uses term "tableau" for a matrix [6]
1844 Grassman, geometry in n dimensions [14], (50 years ahead of its epoch [14 p. 204-205]
1850 Sylvester first use of term "matrix" (matrice=pregnant animal in old french or matrix=womb in latin as it generates determinants)
1858 Cayley matrix algebra [7] but still in 3 dimensions [14]
1888 Giuseppe Peano (1858-1932) axioms of abstract vector space [12]

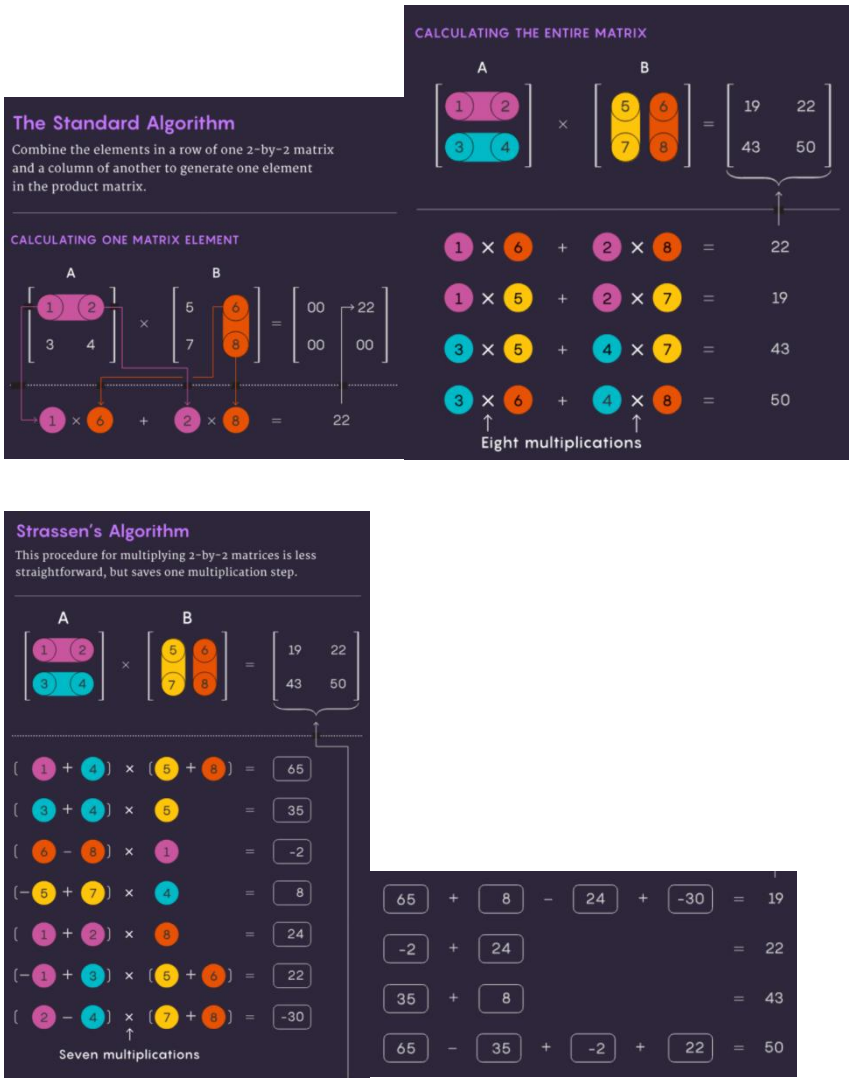
近代矩阵乘法的发展历史，来源： When was Matrix Multiplication invented? (harvard.edu)

随着计算机科学的发展，人们开始思考，能不能从最基础的乘法层面进行加速？

1969 年，Volker Strassen 发表文章提出一种渐进快于平凡算法的 n x n 矩阵相乘算法（n 为 2 的幂次方），引起巨大轰动。在此之前，很少人想过能快于平凡算法的方法。矩阵乘法的渐近上界自此被改进了。

矩阵乘法提升速度的关键在于减少乘法步骤的数量，尽可能将指数从 3（传统方法）降低。可能的最低值 n^2，就是写出答案所需的时间。计算机科学家把这个指数称为 Ω，或者 ω。n^ω 是当 n 越来越大时，成功将两个 $n \times n$ 矩阵相乘所需的最少步骤。

Strassen 算法的大致想法是，我们的传统做法是将大矩阵之间用最朴素地 ijk 三循环进行相乘。那么对于 2x2 的矩阵相乘，一共需要 2^3 即 8 次乘法运算。但是 Strassen 天才地发现，2 x 2 的小矩阵之间的计算可以只用 7 次乘法运算，随后 Shmuel Winograd 证明，我们找不到低于 7 次的运算方法。



标准乘法与 Strassen 的 2x2 矩阵乘法对比，来源： [New Breakthrough Brings Matrix Multiplication Closer to Ideal | Quanta Magazine]

1986 年，Strassen 取得了另一项重大突破，他推出了矩阵乘法的激光法。Strassen 用它确定了 ω 的上限值为 2.48。虽然该方法只是大型矩阵乘法的一个步骤，但却是最重要的步骤之一，计算机科学家一直在不断改进它。

激光法的大致工作原理是，将重叠的块标记为垃圾，并安排处理，而其他块被认为有价值并将被保存。

在做 Project 的这些天，刚好有一个新闻发了出来：[清华姚班本科生连发两作，十年来最大改进：矩阵乘法接近理论最优]

姚班学长的文章提出，Strassen 算法里有一些“hidden loss”。他们是激光法中被程序抛弃的矩阵块，而这些被标记为垃圾的块被发现还是有利用效率的。于是他们修改了激光法的标记方法，将理论复杂度下降到 n 的 2.371552 次方，较之前的方法低了 0.0001，但已经是这 10 年最强的进步。

不过，学长并没有在代码层面实现算法。不难理解，这些方法的常数已经让复杂度的降低变得很鸡肋，且算法的长度也杜绝了大部分程序员的尝试。但是，这些突破不断启发我们，哪怕是再寻常，再基础的计算，或许也有更快，更强的方法出现。

6.2 Strassen 矩阵算法详解

这里的原理主要源于 https://zhuanlan.zhihu.com/p/268392799。当然，我后来又看了《算法导论》，发现神书已经详细讲解了这个算法，并且还进行了多线程的实现。在读完整个过程后，我跪倒在地，不愧是神书！

首先我们会先将 A,B 两个大矩阵分成 4 个小矩阵，用时为 O(1)。我们用 C 表示 A x B 的结果：

$$A=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix},\ B=\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix},\ C=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}$$

紧接着，计算 10 个小矩阵之间的加减，存为 S 花费时间为 $O(n^2)$。

$$\begin{aligned}S_1&=B_{12}-B_{22}\\ S_2&=A_{11}+A_{12}\\ S_3&=A_{21}+A_{22}\\ S_4&=B_{21}-B_{11}\\ S_5&=A_{11}+A_{22}\\ S_6&=B_{11}+B_{22}\\ S_7&=A_{12}-A_{22}\\ S_8&=B_{21}+B_{22}\\ S_9&=A_{11}-A_{21}\\ S_{10}&=B_{11}+B_{12}\end{aligned}$$

随后计算 7 次矩阵乘法，得到 7 个 P 矩阵：

$$\begin{aligned}P_1&=A_{11}\cdot S_1=A_{11}\cdot B_{12}-A_{11}\cdot B_{22}\\ P_2&=S_2\cdot B_{22}=A_{11}\cdot B_{22}+A_{12}\cdot B_{22}\\ P_3&=S_3\cdot B_{11}=A_{21}\cdot B_{11}+A_{22}\cdot B_{11}\\ P_4&=A_{22}\cdot S_4=A_{22}\cdot B_{21}-A_{22}\cdot B_{11}\\ P_5&=S_5\cdot S_6=A_{11}\cdot B_{11}+A_{11}\cdot B_{22}+A_{22}\cdot B_{11}+A_{22}\cdot B_{22}\\ P_6&=S_7\cdot S_8=A_{12}\cdot B_{21}+A12\cdot B_{22}-A_{22}\cdot B_{21}-A_{22}\cdot B_{22}\\ P_7&=S_9\cdot S_{10}=A_{11}\cdot B_{11}+A_{11}\cdot B_{12}-A_{21}\cdot B_{11}-A_{21}\cdot B_{12}\end{aligned}$$

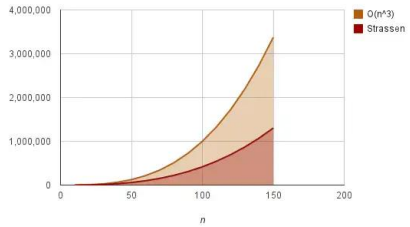
之后我们就可以得到 C 矩阵：

$$\begin{aligned}C_{11}&=P_5+P_4-P_2+P_6\\ C_{12}&=P_1+P_2\\ C_{21}&=P_3+P_4\\ C_{22}&=P_5+P_1-P_3-P_7\end{aligned}$$

这样我们就计算出 A x B 的值了。通过上面的计算过程，我们得到算法的时间递归式：

$$T(n)=\begin{cases}\Theta(1) & 若n=1\\ 7T(\frac{n}{2})+\Theta(n^2) & 若n>1\end{cases}$$

进而我们就可以得到我们的算法时间复杂度 $T(n)=\Theta(n^{\log_2 7})$。如果我们单纯把 n 的三次方和$\log_2 7$画图，还是能明显看到两者的区别。



可以看出，Strassen 算法确实在一定程度上降低了时间复杂度和运算时间，但是，每次矩阵运算都需要**大量的内存**用于临时存储 P 和 S 矩阵，当问题规模 N 很大时是非常致命的，这导致同样的集群我们可能算不了更大的矩阵。同时 Strassen 算法增加了多步加法来换取乘法步骤的减少，这让**算法的常数**增加，在运用中实际使得算法要在 N 大到一定程度时才有优势。可是综合来看，在算法有优势时集群没优势，会让这个算法的地位变得有点尴尬。但是，它确实对矩阵计算进行了加速，对于今天的图像处理等操作有着非常重要的意义。

6.3 Intel Top-down 性能分析模型

初始代码 multiply1() 极为内存受限，因为它以不利于缓存的方式遍历大型矩阵。在 multiply2() 中应用的循环交换优化大大提高了速度。尽管如此，经过优化的代码仍然是后端受限，但现在它从内存受限转变为核心受限。接下来，在 multiply3() 中尝试了向量化，因为它减少了端口利用率，减少了净指令数，从而实现了另一个加速。

5.5. Case Study 1: Matrix-Multiply

A matrix-multiply textbook kernel is analyzed with Top-Down. It demos the iterative nature of performance tuning.

Table 4: Results of tuning Matrix-Multiply case

Metric	multiply1	multiply2	multiply3
Speedup	1.0x	11.8x	16.5x
IPC	0.17	1.19	0.80
Frontend Bound	0.00	0.07	0.02
Retiring	0.05	0.41	0.28
Bad Speculation	0.00	0.00	0.00
Backend Bound	**0.95**	**0.52**	**0.70**
├-+ Memory Bound	0.84	0.12	0.31
| ├-- L1 Bound	0.05	0.07	0.03
| ├-- L2 Bound	0.03	-	0.05
| ├-- L3 Bound	0.05	-	0.01
| ├-- MEM Bound	0.71	0.07	0.21
| └-- Stores Bound	-	-	-
└-+ Core Bound	0.15	0.64	0.55
├-- Divider	-	-	-
└-- Ports Utiliz.	0.15	0.64	0.55

下面是一位英特尔工程师的分析：

1-2的优化分析:

- 首先是IPC和speedup部分接近等比例上升，说明算法部分的改变不大。同样，retiring的提升和IPC提升表明指令执行速度加快。这些是表象。
- memory bound占比减少，mem bound（dram）下降到1/10；core bound占比提升，且port的使用率提升4倍以上。说明应用在这个阶段主要通过优化memory cache，减少了内存的访问瓶颈，从L1的变化上看有极有可能是微调了数据结构，使得其大幅增加了L1的命中概率。

2-3的优化分析：

- IPC下降了50%，speedup上升了50%。看似有点不合理，好比搬砖速度下降了，但工作完成的时间减少了——要么是搬砖的人多了，要么是砖少了。说明算法上有了调整优化，指令数减少。
- memory bound提升不多，单mem bound 提升了3倍，说明读取的次数减少，但每次读取的数量增加，但这个信息结合speedup的提升反而变得没有参考性了。
- core bound/port utilization略降，但比例小于IPC下降的情况可以理解为平均每条指令的执行时间略有提升。指令更重，说明在这个阶段大概率使用了SIMD合并过指令。

--本文部分节选自开源小站2018/02/14发布的 Top-down性能分析模型 - 开源小站

Part 7: More Jobs can be done

1.对于实验 1，Java 内部代码是如何实现精度的估算的？
2.对于实验 2，Java 内部各个代码是如何实现的？由于通信的影响，我们的多线程函数什么时候是弊大于利的？
3.对于实验 3，能不能换一个更有代表性的矩阵读取和写入函数？不同程序唤起的时间记录函数不同，有没有什么好方法能让程序同时用同一个函数记录时间？
4.对于实验 4，让程序 CPI 上涨的函数为什么要花更多的时钟周期？为什么有的指令就不用那么多时钟周期？CPI 在我们这次的测试里是越高越好还是越低越好？
5.GPU 版本的测试与探索。GPU 里的代码如何执行？
6.随着 N 的增加，Cache，DRAM，乃至虚拟化的内存会不会成为新的瓶颈？UMA 和 NUMA 跨节点影响我们的程序运行时间的效果怎么样？OpenMP 多核时的程序运行时间是否和核的布置有关？
~~6.所有给读者留作的练习。~~

Java 真的跑的比 C 慢吗？我们可以看到其实刚刚在多核情况下，Java 是可以多线程启动的，而这相比于单线程 C 程序会有一定弥补。如果在 C 没优化的状态下，他们之间的差距会更加缩小，可能最后，程序执行我们平时交了这么多次 OJ，大家都喜欢 C++，但是在真实情况下 Java 带来的性能牺牲没有我们想象的那么大？

Part 8: 总结

我们在本次 Project 中共进行了 4 次实验，分别对 C 和 Java 在矩阵乘法程序中的精度、编译效率、运行时间和启动方法进行了观测。

通过实验，我们发现了几个关键的性能差异点：

1. **精度判定**：Java 在执行时会自动判定 float 精度，而 C 语言不会，这可能对性能产生一定影响。

2. **JVM 和编译器的影响**：Java 的性能受到 JVM 和 JIT 编译器的影响。JIT 的 C1C2 编译器效果不明显，而 C 语言使用-O3 编译优化可以将变量优化到寄存器中，减少汇编指令和内存读取指令，从而降低运行时间。

3. **平台差异**：在不同的软硬件环境和编译器下，Java 和 C 的表现有所差异，这意味着性能也可能受到具体实施环境的影响。

4. **多线程和通信瓶颈**：Java 的多线程和 C 的 OpenMP 在通信上可能存在瓶颈，多线程不一定比单线程性能更优。

5. **启动方式和操作系统的影响**：Java 和 C 的不同启动方式以及 Linux 和 Windows 的调用程序方式可能影响程序的性能。

6. **性能分析法的应用**：通过使用 Intel TopDown 性能分析法，我们能分析 Java 和 C 程序的运行瓶颈，以及 CPI 变化的原因。

综合来看，通过一系列实验，我认为 Java 和 C 在矩阵乘法操作中的性能差异多受编译器优化、运行时环境、编程语言特性以及操作系统等因素的影响。**在不同的实现细节和优化水平下，两者的性能表现可能会有很大差异，因此不能简单地断言哪种语言绝对更快**。

ChatGLM 补充到：我们在选择技术栈时，需要考虑具体的应用场景、性能需求和开发维护的复杂度等因素。

*我认为比较开心的一点是，这次 Project 我学到了很多性能分析操作，想明白了很多没想过的问题，**实验都是自己设计自己完成，并且找到了自己想证明的知识，并且还跑通了 Intel 工程师跑通的实验。**我对自己在以后优化程序上更加有信心，并且对整个优化要走的流程有了更系统的经验。我感受到一种探索的乐趣，这也一直让我坚信我确实应该要尝试做做科研。*

我在跑时间测试的时候去读了海子的诗
突然觉得 诗歌就是一段段汇编语言
指针跳转，语句走了一行又一行
内存翻了一页又一页
我突然想到
等程序跑完，我会躺在哪片麦地上呢
风扇呼呼 就像海子笔下麦子谦卑的呢喃
终于，我看到
目击众神死亡的草原上野花一片
于是，我写下
远在内存的指针比内存更远

Part 9: Acknowledgement

感谢肖翊成同学和邱俊杰同学在“试管 1 号”装机和配置系统上的帮助，从底层配置 BIOS，IB，学习 BMC 等等让我受益良多。感谢超算中心的硬件提供和网络支持。感谢马国恒同学，在我 Project 生病时帮我带饭和录了 lab 老师的讲解，我很荣幸拥有这样的舍友。感谢白雨卉老师在编译器优化上的实验和在计组课上给我的答疑。

Reference

[1] StrassenMultiplier.java A program that implement Strassen Algorithm: by JinGe Wang. Retrieved from Algorithms/src/matrix/StrassenMultiplier.java at master • jingedawang/Algorithms (github.com)

[2] Duan, R., Wu, H., & Zhou, R. (2022). Faster Matrix Multiplication via Asymmetric Hashing. 2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS), 2129-2138.

[3] New Breakthrough Brings Matrix Multiplication Closer to Ideal New Breakthrough Brings Matrix Multiplication Closer to Ideal | Quanta Magazine

[4] History of Matrix Multiplication When was Matrix Multiplication invented? (harvard.edu)

[5] 南方科技大学 2023Fall 数据库（H）Project1 DBMS 赖海斌.pdf (sustech.edu.cn)（哈哈，怎么还王婆卖瓜引用起自己来了）

[6] WaitForSingleObjectEx 方法解释
_https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject

[7] 白老师的实验结果
https://bb.sustech.edu.cn/bbcswebdav/pid-455556-dt-content-rid-15826365_1/courses/CS214-30022126-2024SP/CompOrg_24S_Lec5_Performance.pdf

[8] Gunther, N. J. (2008). A General Theory of Computational Scalability Based on Rational Functions. arXiv, https://doi.org/10.48550/arXiv.0808.1431

[9] Yasin, A. (2014, March). A top-down method for performance analysis and counters architecture. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 35-44). IEEE. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6844459

[10] Top-down 性能分析模型 https://zhuanlan.zhihu.com/p/34688930

[11] 从 0 到 1 在学校 Top500 超算集群跑 HPL 程序_哔哩哔哩_bilibili