

Project2: Computing the dot product of two vectors

Contents

1. 实验总览	2.
2. Experiment 1	3.
2.1. 实验配置	3.
2.2. 实验预备：程序内计时	3.
2.3. 实验步骤与方法	5.
2.4. 实验思考	7.
2.4.1. Java 侧	7.
2.4.2. GCC 侧	11.
2.4.2.1. 代码的 O2 优化	12.
2.4.2.2. 代码的 O3 优化	13.
2.4.2.3. Linux 程序运行	14.
3. Experiment 2	17.
3.1. 实验配置	17.
3.2. 实验步骤与方法	17.
3.3. 实验结果	18.
3.4. 实验思考	19.
3.4.1. 程序的实例——进程与线程	19.
3.4.2. 多线程的数据竞争	22.
3.4.3. Java 的多线程实现初探	22.
4. Experiment 3	23.
4.1. 实验配置	23.
4.2. 实验步骤与方法	23.

4.3. 总结	30.
5. Experiment 4	30.
5.1. 循环展开	30.
5.2. SIMD 指令集 + register 变量	30.
5.3. CUDA 并行优化	34.
5.3.1. 性能对比	38.
5.4. 总结	39.
6. Extra	39.
7. 结论与反思	42.
Bibliography	43.
A GCC 优化等级对应选项细节	43.
AA O0	43.
AB O 和-O1	43.
AC O2	44.
AD O3	46.

§1. 实验总览

向量的点乘 $\mathbf{u} \cdot \mathbf{v}$ 在计算机学科中是非常基础的运算，小至计算一个力的在某段位移上所作的功，大至机器学习中的神经网络训练都需要用到它。我们很容易用 C 或 Java 这两个熟悉的语言来实现它们，但是这个看似简单的过程真的很简单吗？C 与 Java 在这个过程中表现孰优孰劣？我们又能如何根据观察到的现象与结论对点乘操作进行优化？为了解决以上疑问，我设计了几个小实验来具体探究点乘操作的性能表现。

1. Experiment 1: 使用 C 与 Java 实现朴素点乘操作，并用程序内计时比较不同优化等级下的 C 与 Java 的性能表现。
2. Experiment 2: 使用 C 与 Java 实现多线程版本的点乘操作，并用程序内计时分析不同线程数下的性能表现。
3. Experiment 3: 使用性能分析工具追踪 gcc 与 JIT 编译器的优化过程，分析它们的优化策略与效果。
4. Experiment 4: 探索几个常用优化技巧优化 C 点乘操作的性能
5. *Extra:* 延伸探究 Rust 语言的点乘操作性能表现与 cargo 的优化策略

§2. Experiment 1

§2.1. 实验配置

Experiment 1 在我笔记本本地的 WSL2 系统运行，配置如下：

Item	Details
System	5.15.167.4-microsoft-standard-WSL2 Ubuntu 22.04.2 LTS
OS Kernel	5.15.167.4-microsoft-standard-WSL2
gcc-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
g++-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
JDK-version	openjdk 21.0.2 2024-01-16 LTS
CPU	AMD Ryzen 5 5600H with Radeon Graphics
CPU-cores	6 cores
CPU-cache-size	512 KB
Memory	7.0278 GB

§2.2. 实验预备：程序内计时

在这个实验中，我们需要用到程序内计时来测量点乘操作的时间性能表现。对于 Java，我们可以直接使用 `System.nanoTime()` 来获取当前时间的纳秒数；而 C 中 `<time.h>` 定义了不同的时间函数来获取当前时间，比如 `clock()`、`clock_gettime()`、`time()` 等，project 文档中提到了用 `clock()` 方法返回的时间可能不准确，为了保障我们实验结果的准确性，究竟需要选择哪个函数呢？

为了统一时间测量指标，我们要先明确几个 Linux 系统下的时间参数：

- Wall-clock time: 也称为实时时间，表示从用户能体验到程序开始运行到结束所消耗的实际时间。这是我们最常用的时间测量指标。
- CPU time: 表示从程序开始运行到结束所消耗的 CPU 时间。CPU 时间不包含 I/O 操作、等待时间等。能够直接反映程序的计算性能。
- Process time: 表示从程序开始运行到结束所消耗的进程时间。它包含了用户态与内核态的时间。

我们可以设计程序分别测试这几个函数的测量一段时间的精度与稳定性。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

#define NUM_THREADS 4
#define TEST_DURATION 1
#define NUM_ITERATIONS 5
#define _POSIX_C_SOURCE 200809L

typedef struct {
    double clock_duration;
    double monotonic_duration;
}
```

```

    double process_cputime_duration;
} ThreadResult;

void perform_work() {

    volatile double sum = 0.0;
    for (int i = 0; i < 10000000; i++) {
        sum += i * 0.1;
    }

    usleep(TEST_DURATION * 1000);
}

void single_thread_test() {
    printf("\n==== Single Thread Test ====\n");

    for (int iter = 0; iter < NUM_ITERATIONS; iter++) {

        clock_t start_clock = clock();
        perform_work();
        clock_t end_clock = clock();
        double clock_duration = (double)(end_clock - start_clock) / CLOCKS_PER_SEC;

        struct timespec start_mono, end_mono;
        clock_gettime(CLOCK_MONOTONIC, &start_mono);
        perform_work();
        clock_gettime(CLOCK_MONOTONIC, &end_mono);
        double monotonic_duration = (end_mono.tv_sec - start_mono.tv_sec) +
                                     (end_mono.tv_nsec - start_mono.tv_nsec) / 1e9;

        struct timespec start_cpu, end_cpu;
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start_cpu);
        perform_work();
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end_cpu);
        double cpu_duration = (end_cpu.tv_sec - start_cpu.tv_sec) +
                             (end_cpu.tv_nsec - start_cpu.tv_nsec) / 1e9;

        printf("clock(): %.6fs\n", clock_duration);
        printf("CLOCK_MONOTONIC: %.6fs\n", monotonic_duration);
        printf("CLOCK_PROCESS_CPUTIME_ID: %.6fs\n", cpu_duration);
    }
}

// multithread test...

```

首先，`clock()`返回的结构体 `clock_t` 中包含了一个 `clock_t` 类型的成员变量 `_clock`。前后调用两次 `clock()` 函数，计算它们的差值就能得到程序运行的 CPU 时间。返回的时间单位为时钟周期数，除以 `CLOCKS_PER_SEC` 就能得到秒数。

而 `clock_gettime()` 函数返回的结构体 `timespec` 中包含了两个成员变量 `tv_sec` 与 `tv_nsec`，分别表示秒数与纳秒数。这个时间测量的精度达到了 1ns，我们可以用它来测量更精确的时间。`clock_gettime()` 函数的第一个参数是时钟类型枚举量，`CLOCK_MONOTONIC` 表示测量墙上时间，`CLOCK_PROCESS_CPUTIME_ID` 表示进程 CPU 时间。

Observation 2.2.1 (Result).

测试持续时间: 1 秒 x 5 次迭代 比较 `clock()` 和 `clock_gettime()` 的精度和行为差异, 测试持续时间: 1 秒 x 5 次迭代

==== 单线程测试 ====

- 迭代 1:

`clock(): 0.031471 秒`

`CLOCK_MONOTONIC: 0.032638 秒`

`CLOCK_PROCESS_CPUTIME_ID: 0.031759 秒`

- 迭代 2:

`clock(): 0.031260 秒`

`CLOCK_MONOTONIC: 0.032261 秒`

`CLOCK_PROCESS_CPUTIME_ID: 0.031270 秒`

- 迭代 3:

`clock(): 0.032118 秒`

`CLOCK_MONOTONIC: 0.032516 秒`

`CLOCK_PROCESS_CPUTIME_ID: 0.031544 秒`

- 迭代 4:

`clock(): 0.032118 秒`

`CLOCK_MONOTONIC: 0.032203 秒`

`CLOCK_PROCESS_CPUTIME_ID: 0.031861 秒`

- 迭代 5:

`clock(): 0.032078 秒`

`CLOCK_MONOTONIC: 0.033483 秒`

`CLOCK_PROCESS_CPUTIME_ID: 0.031055 秒`

经计算, 由 `CLOCK_PROCESS_CPUTIME_ID` 返回的时间样本方差小于 `clock()` 方法, 因而在时间测量中更为精确。另外一个现象是, `CLOCK_MONOTONIC` 返回的时间相比 `clock()` 方法和 `CLOCK_PROCESS_CPUTIME_ID` 方法的时间要大一些, 这是因为它包含了 `usleep()` 函数的线程睡眠时间。

§2.3. 实验步骤与方法

确定了时间测量方法后, 我们就可以开始实验了。这里我们选择 *performance* 作为实验的性能指标, 来衡量不同数据类型、不同优化等级下 C 与 Java 的性能表现, 其中 *performance* 定义为:

$$\text{performance} = \frac{1}{\text{CPU time}} \text{ (unit: } s^{-1})$$

数据大小上，选择了从 $10^2 \sim 10^8$ 的范围，阶长为 10。编译优化等级从 O0 到 Ofast。

C 程序数据收集脚本如下，这次我们延续上次的传统，依然采用 csv 格式来存储数据。

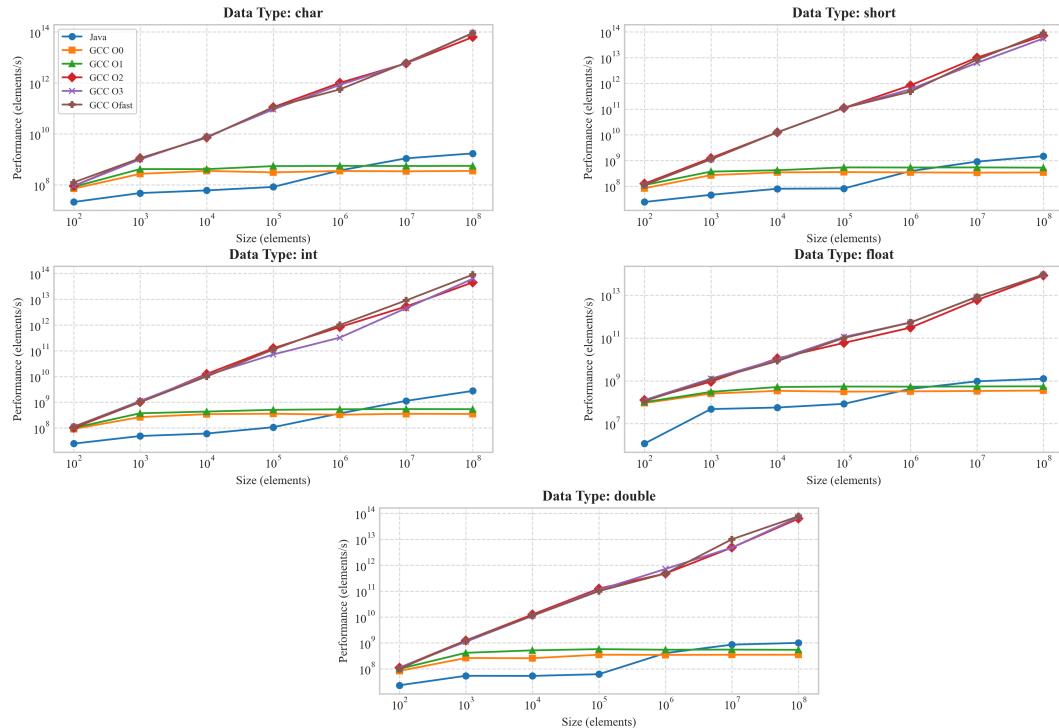
```
#!/bin/bash
DATA_TYPES=("float" "double" "int" "short" "char")
SIZES=(100 1000 10000 100000 1000000 10000000 100000000)
OPT_LEVELS=("00" "01" "02" "03" "Ofast")
SRC_DIR="../src"
BUILD_DIR="../build"
CSV_DIR="../results_1"
mkdir -p $CSV_DIR

for opt in "${OPT_LEVELS[@]}"; do
    gcc -$opt -march=native $SRC_DIR/dotproduct.c -o $BUILD_DIR/dot_$opt -Wall

    echo "Size,Data Type,Time" > $CSV_DIR/results_c_$opt.csv
    for dtype in {0..4}; do
        for size in "${SIZES[@]}"; do
            echo "Testing $opt ${DATA_TYPES[$dtype]} with size $size"
            echo -n "$size,${DATA_TYPES[$dtype]},," >> $CSV_DIR/results_c_$opt.csv
            $BUILD_DIR/dot_$opt $size ${DATA_TYPES[$dtype]} >> $CSV_DIR/$opt.csv
        done
    done
done
```

实验用到的 C 程序代码较长，不在此做展示；作为对照的 Java 程序由 Anthropic 的 Claude-3.7-Sonnet 模型辅助完成，为了实验完全控制变量，我尽量保证 Java 与 C 程序的逻辑基本一致，没有使用数组边界检查、泛型方法等可能影响效率的实现。实验的最终结果呈现如下：

Performance of dotproduct — C versus Java



Performance measured as size/time (higher is better)

我们从这个不同数据类型的点乘操作的性能表现中可以看出，在低数据量时，Java 程序性能不如无优化的 C 程序，但当数据规模达到 10^6 之后，Java 程序的性能甚至超过了 O0 优化的 C 程序，尽管 O2、O3、Ofast 优化等级下的 C 程序中，Java 程序的性能依然不如它们。而且另外一个观察是，Java、O0、O1 优化的程序性能随数据大小增加不明显，但 O2、O3、Ofast 优化的程序性能随数据大小线性增加，这个现象启发我们需要考察一下 gcc 编译器做了怎样的优化。对于不同数据类型，一个 General 的结论，是 $\text{double} > \text{float} \approx \text{int} > \text{short} \approx \text{char}$ ，这个和个数据类型的基本计算操作的复杂度是正相关的，数据位宽越大，计算复杂度越高，性能表现也就越差；而浮点数的计算复杂度又高于整数类型。

§2.4. 实验思考

上面这个出乎意料的结果的确使我对 JVM 的“运行龟速”旧印象改观了，那么为什么 Java 这种独特的“混合编译解释”的模式反而比无任何优化的 C 程序还要快呢？这就不得不深入探究一下这两种语言究竟是如何在背后处理我们的代码的。

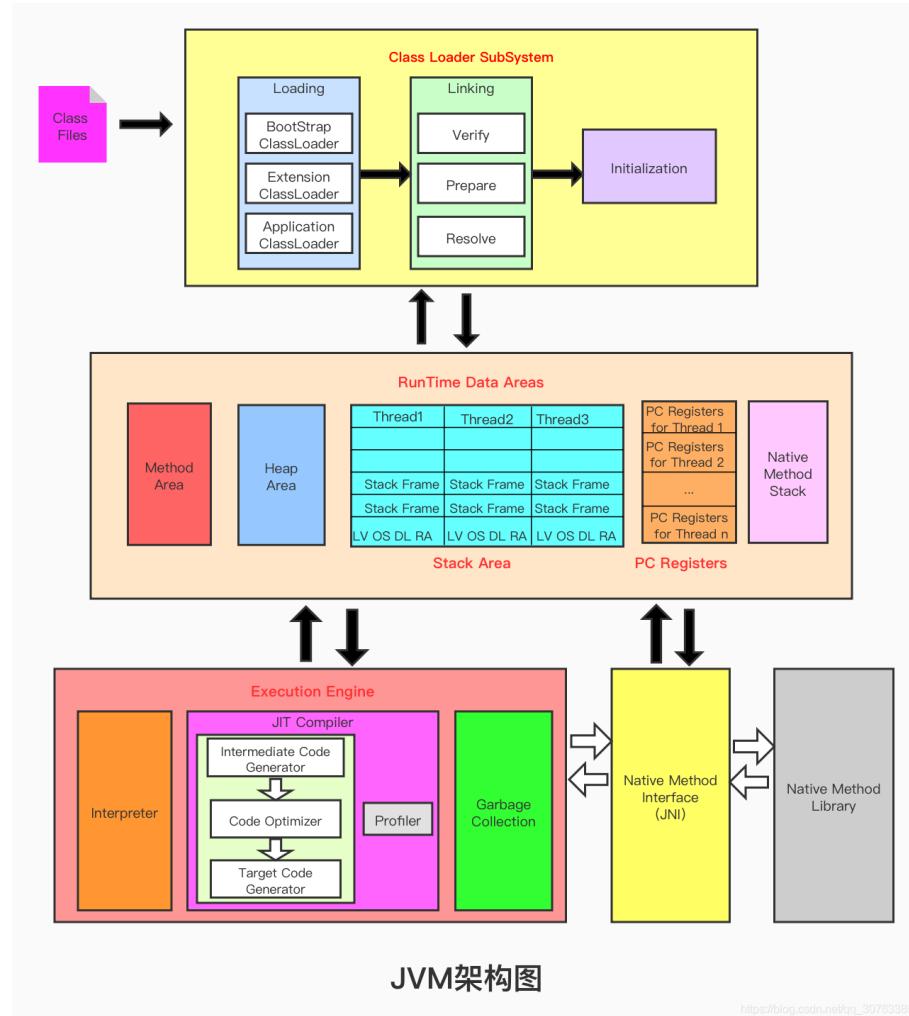
§2.4.1. Java 侧

Java 对我来说是一个又熟悉又陌生的语言，说它熟悉是因为 Java 是我第二个接触的编程语言，而且用了几乎一学期来学习它的语法与特性；说它陌生是因为从来没有深入了解 JVM 是怎么处理我的代码进行运作的，只需要知道如何用 javac 编译代码，java 运行 .class 就可以了。但是为了弄清楚这个实验结果的根源，我不得不学习一下 JVM 的核心机制——JIT(Just-In-Time)

与 C/C++ 不同，javac 编译器生成的字节码 class 文件实际上是一种 IR(intermediate Representation)。我们可以通过 Java 的 javap 命令来查看字节码的内容。

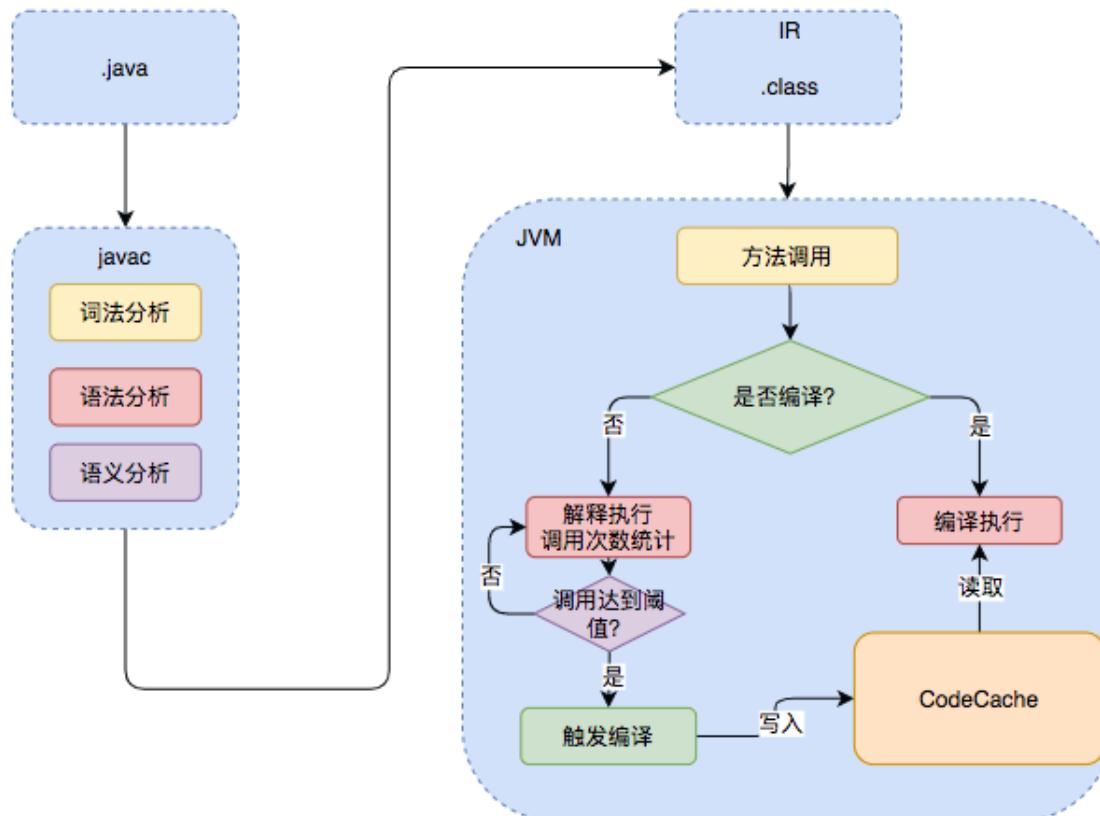
```
public static long dotProductPlainShort(short[], short[], int);
Code:
 0: lconst_0
 1: lstore_3
 2: iconst_0
 3: istore      5
 5: iload       5
 7: iload_2
 8: if_icmpge   30
11: lload_3
12: aload_0
13: iload       5
15: saload
16: aload_1
17: iload       5
19: saload
20: imul
21: i2l
22: ladd
23: lstore_3
24: iinc      5, 1
27: goto      5
30: lload_3
31: lreturn
```

.class 中间表示并不是直接可以被 CPU 执行的机器码，而是需要在运行时由 JIT 编译器将其转换为机器码或者交给 interpreter 解释执行，这也是 Java 为众多编程爱好者一直广为诟病的“慢”的原因之一。不过，Java 也并不是一无是处，JIT 编译器就是为了解决解释运行效率慢而设计，按照既定策略有需要地将字节码编译为机器码执行，不仅如此，JIT 编译器还会在运行时对代码分支进行分析，能够做到机器码级别的循环展开、内联等优化，甚至可以根据运行时的实际数据来决定使用哪种优化策略。下面这张图展示了一个.class 文件在 JVM 中处理的过程：



解释器在执行程序时，对于每一条字节码指令，都需要进行一次解释过程，然后执行相应的机器指令。这个过程在每次执行时都会重复进行，因为解释器不会记住之前的解释结果。

与此相对，JIT 会将频繁执行的字节码编译成机器码，这个机制叫做 codeCache。当代码块的执行次数超过一定阈值时，JIT 会将其编译为机器码并存储在 codeCache 中。这样，当下次执行相同的代码块时，JIT 就可以直接使用已经编译好的机器码，而不需要再次进行解释。



具体来说，JIT过程中，编译器会识别方法调用和循环结构的头尾，比如说我们的 `dotProductPlainShort` 函数 class 字节码反编译如下：

```
public static long dotProductPlainShort(short[], short[], int);
Code:
 0: lconst_0
 1: lstore_3
 2: iconst_0
 3: istore      5
 5: iload       5
 7: iload_2
 8: if_icmpge   30
11: lload_3
12: aload_0
13: iload      5
15: saload
16: aload_1
17: iload      5
19: saload
20: imul
21: i2l
22: ladd
23: lstore_3
24: iinc      5, 1
27: goto      5
30: lload_3
31: lreturn
```

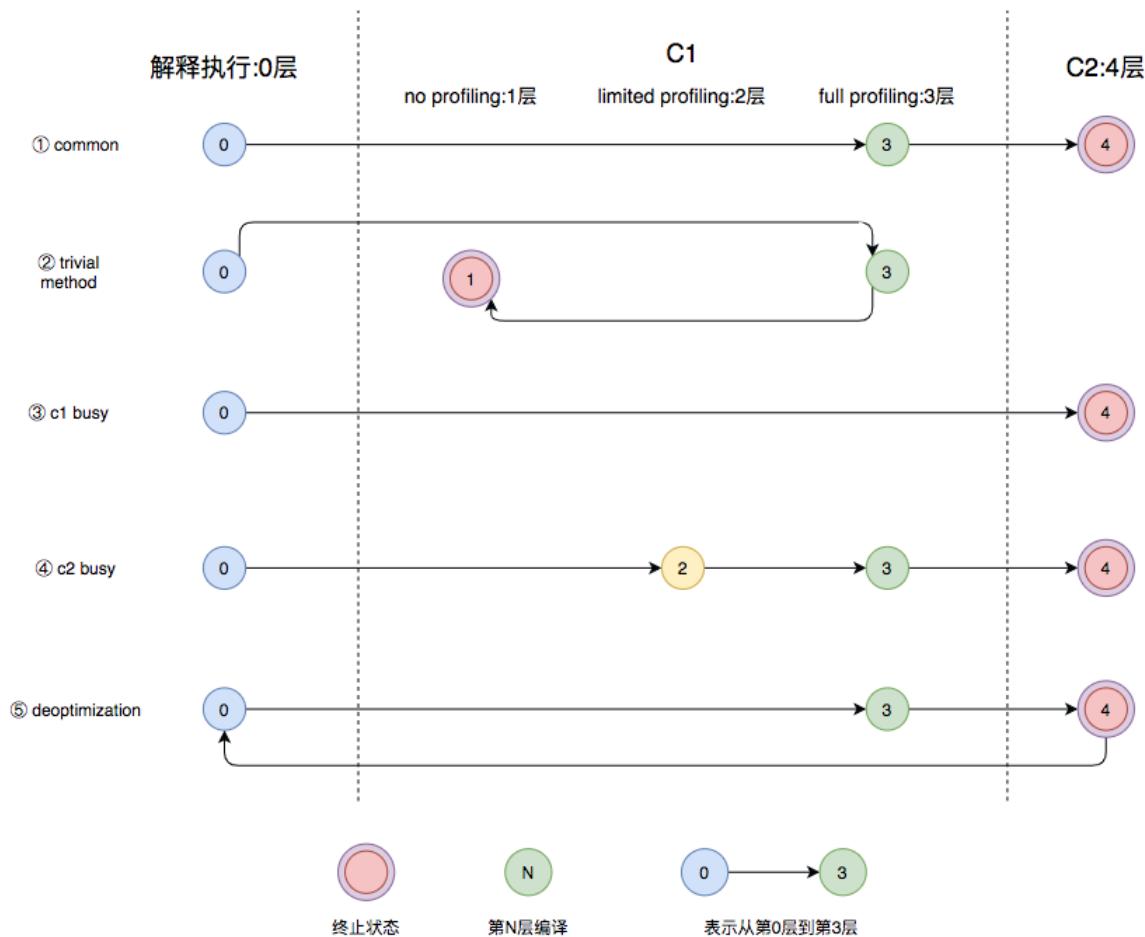
偏移量为 27 的 `goto` 指令会被 JIT 编译器识别为循环的开始，偏移量为 8 的 `if_icmpge` 指令会被识别为循环的结束，每次执行一次 `goto` 指令，JVM 会将该方法对应的循环回边计数器加 1，当

方法调用次数和循环回边次数的和超过参数 `XX:CompileThreshold` 时, JIT 编译器会将该方法编译为机器码并存储在 codeCache 中。

进一步地, JIT 的具体优化策略会依程序代码的编译层级而定, 这里我们先了解一下 JVM 的编译层级。

1. **Tier 0 - Interpreter:** 解释器直接执行字节码, 适用于小型程序或短时间运行的代码。
2. **Tier 1 - C1 with Simple Optimizations:** 代码由 C1 编译器(Client Compiler)编译为机器码, 进行一些简单的优化。适用于中等复杂度的代码。
3. **Tier 2 - C1 with Full Optimizations:** 代码仍由 C1 编译器处理, 但是如逃逸分析等复杂的优化会被启用, 可能需要更长的编译时间。适用于复杂的代码。
4. **Tier 3 - C1 with Profiler:** C1 编译器除了编译优化, 还会收集运行时的性能数据用于后续 C2 编译器的优化。
5. **Tier 4 - C2 Optimizations:** 代码由 C2 编译器编译为机器码, 会根据分析数据进行深入优化。适用于长时间运行的代码。

下面这张图就说明了不同状态下各编译层级的工作流程:



根据 JIT 的官方文档, 分层编译的触发条件为

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s && i + b > TierXCompileThreshold * s)
```

其中 `i` 为调用次数, `b` 为循环回边次数, `s` 为系数。

这里可以看出, JIT 的分层编译判定比较复杂。由于我们的点乘程序逻辑相对简单, 时间复杂度低, 很可能没有达到 C2 编译甚至 C1 满优化编译的条件, 所以 Java 程序虽然可以利用 codeCache

机制和一些简单的编译优化超过低优化等级的 C 程序，但与 O2、O3 优化的 C 程序相比还是会略逊一筹。

§2.4.2. GCC 侧

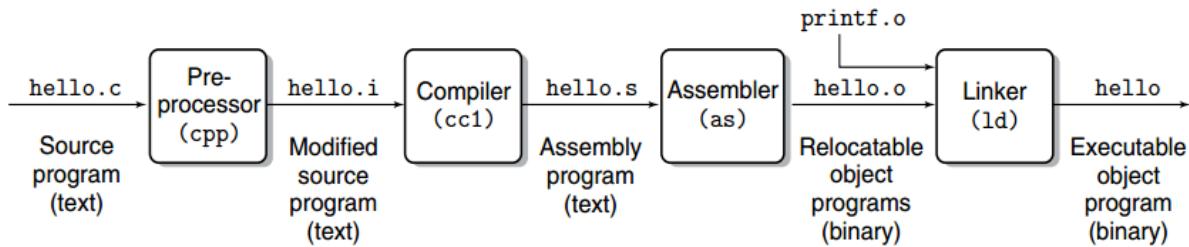


Figure 1.3 The compilation system.

我们一个熟知的完整的 C 代码编译与链接过程如上所示，如果想要了解代码究竟是如何被执行的，那么从编译器得到的汇编文件入手是最佳的，因为后续的汇编与链接过程都不会实际影响最终执行的逻辑。我以 char 类型的 dotproduct_plain 函数主体对应的汇编指令序列为例展示如下：

```

00000000000002d8 <dotproduct_plain_i8>:
 2dc: 55          push %rbp # save base-pointer
 2dd: 48 89 e5   mov %rsp,%rbp # %rbp is stack-pointer
 2e0: 48 89 7d e8 mov %rdi,-0x18(%rbp) # arg0: a -> %rdi
 2e4: 48 89 75 e0 mov %rsi,-0x20(%rbp) # arg1: b -> %rsi
 2e8: 48 89 55 d8 mov %rdx,-0x28(%rbp) # arg2: &result -> %rdx
 2ec: 48 89 4d d0 mov %rcx,-0x30(%rbp) # arg3: size -> %rcx
 2f0: 48 c7 45 f8 00 00 00 movq $0x0,-0x8(%rbp)
 2f7: 00
 2f8: eb 3d      jmp 337 <dotproduct_plain_i8+0x5f> # jump to check
boundary
 2fa: 48 8b 45 d8 mov -0x28(%rbp),%rax
 2fe: 48 8b 08   mov (%rax),%rcx
 301: 48 8b 55 e8 mov -0x18(%rbp),%rdx
 305: 48 8b 45 f8 mov -0x8(%rbp),%rax
 309: 48 01 d0   add %rdx,%rax
 30c: 0f b6 00   movzb (%rax),%eax
 30f: 0f be d0   movsb %al,%edx
 312: 48 8b 75 e0 mov -0x20(%rbp),%rsi
 316: 48 8b 45 f8 mov -0x8(%rbp),%rax
 31a: 48 01 f0   add %rsi,%rax
 31d: 0f b6 00   movzb (%rax),%eax
 320: 0f be c0   movsb %al,%eax
 323: 0f af c2   imul %edx,%eax
 326: 48 98       cltq
 328: 48 8d 14 01 lea (%rcx,%rax,1),%rdx
 32c: 48 8b 45 d8 mov -0x28(%rbp),%rax
 330: 48 89 10   mov %rdx,(%rax)
 333: 48 ff 45 f8 incq -0x8(%rbp)
 337: 48 8b 45 f8 mov -0x8(%rbp),%rax
 33b: 48 3b 45 d0 cmp -0x30(%rbp),%rax # if
 33f: 72 b9       jb 2fa <dotproduct_plain_i8+0x22>
 341: 90          nop
 342: 90          nop
 343: 5d          pop %rbp
 344: c3          ret

```

虽然南科大 Organization 课程学习的是 RV32 指令集，对于选择本课程的学生来说读懂以上 x86 指令应该是基本要求吧。我在这里稍微翻译一下这段汇编代码，首先出于 x86 寄存器保存规则，被调用者 Callee 需要保存 rbp 基址寄存器和四个 rcx, rdx, rsi, rdi 参数寄存器，这与 RISC-V 指令中 addi sp sp [offset] 然后 mv Calle-Save register 到栈中是一致的。

接下来 movq 初始化栈指针偏移 -0x8 的值为 0 (也就是 i)，通过一个 jmp 指令直接跳转到循环条件检查语句 337 处。在这里会先将 i 加载到 rax 中并通过下一个比较语句 cmp 比较 i 与 -0x30(rsp) 对应变量 (参数 size)，若小于则跳转到循环开始地址 2fa。

循环开始，函数首先读取 -0x28(rsp) (result 的地址) 并取出 result 的初值。之后便是分别读并扩展 a[i]、b[i] 到寄存器 edx 和 eax，使用 imul 指令计算乘积，lea 指令将 result 当前值与乘积相加存到 rdx 中，最后取 result 地址将新值写入，incq 指令执行 i++ 操作，这样就完成了一次循环。如果比较语句条件成立 ($i \geq size$)，就 pop 出 rbp 并 ret。

从上面编译出的汇编代码可以看出，虽然在 C 代码中只是一个简单的 for 循环，其涉及到的指令数也可能多的超乎想象。尤其是存在冗杂的指令操作。比如如果循环变量存到某个寄存器中，就可大大减少反复取值和存值的操作；而且每次循环只计算一个位置对应的内积，这种循环方式可能导致计算访存比很低，在规模较大时会造成效率下降和资源浪费。

这样我们通过编译出的汇编代码可以看出，gcc 编译时并没有按我们预想的那样生成高效的代码，而是生成了大量的冗余指令，导致了性能的下降。那么编译器优化究竟替我们做了什么呢？我们首先需要了解一下 gcc 的优化等级。¹

Optimizing Level	Brief Description
O0	不做任何优化
O1	优化会消耗较多的编译时间，它主要对代码的分支，常量以及表达式等进行优化
O2	会尝试更多的寄存器级的优化以及指令级的优化，gcc 将执行几乎所有的不包含时间和空间折中的优化
O3	在 O2 的基础上进行更多的优化。例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化
Os	在 O2 的基础之上，尽量的降低目标代码的大小
Ofast	在 O3 的基础上，开启一些不符合标准的优化选项，例如不检查浮点数的溢出等，一般不推荐开启
Og	会精心挑选部分与 -g 选项不冲突的优化选项，能提供合理的优化水平，同时产生较好的可调试信息和对语言标准的遵循程度

关于各等级对应的具体的优化选项，可参见附录 Appendix A。

以下我们以 O2 和 O3 的优化等级为例，分析它们的优化策略与效果。

§2.4.2.1. 代码的 O2 优化

我们使用如下命令编译代码，查看 O2 优化后的汇编代码：

```
gcc -c -Wall -O2 dotproduct.c -g -o dummy.o
objdump -s -d dummy.o > dotproduct_02.s

0000000000000002c0 <dotproduct_plain_i8>:
2c4: 49 89 fa          mov    %rdi,%r10
2c7: 48 85 c9          test   %rcx,%rcx
```

¹Description from <https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html#Optimize-Options>

```

2ca: 74 32          je    2fe <dotproduct_plain_i8+0x3e>
2cc: 4c 8b 02        mov   (%rdx),%r8
2cf: 31 ff          xor   %edi,%edi
2d1: 66 66 2e 0f 1f 84 00  data16 cs nopw 0x0(%rax,%rax,1)
2d8: 00 00 00 00
2dc: 0f 1f 40 00      nopl  0x0(%rax)
2e0: 41 0f be 04 3a  movsbl (%r10,%rdi,1),%eax
2e5: 44 0f be 0c 3e  movsbl (%rsi,%rdi,1),%r9d
2ea: 48 ff c7          inc   %rdi
2ed: 41 0f af c1      imul  %r9d,%eax
2f1: 48 98          cltq
2f3: 49 01 c0          add   %rax,%r8
2f6: 4c 89 02          mov   %r8,(%rdx)
2f9: 48 39 f9          cmp   %rdi,%rcx
2fc: 75 e2          jne   2e0 <dotproduct_plain_i8+0x20>
2fe: c3          ret

```

可以看出，O2 优化后的代码与 O0 的代码相比，指令数大幅精简，而且函数参数的传递方式也发生了变化，O2 优化后函数参数不再通过栈传递，而是通过寄存器传递。这使得循环体内只需要一次读取参数，减少了多次读取的开销。另外，O2 优化后还使用了 `movsbl` 指令来读取 `a[i]` 与 `b[i]`，而不是之前的 `movzbl` 指令，这样可以避免多余的扩展操作。这也就不难理解为什么 O2 优化后运行性能会比 O0 高了近 10 倍。

回想到 ACM-ICPC 比赛中，O2 优化等级经常使用的编译选项之一，因为它在保证代码性能的同时，仍然能够提供较好的可调试信息和对语言标准的遵循程度。不过，O2 优化的编译时间较长，对于大型项目来说，编译时间可能会显著增加。

§2.4.2.2. 代码的 O3 优化

类似地，我们使用如下命令编译代码，查看 O3 优化后的汇编代码：

```

gcc -c -Wall -O3 dotproduct.c -g -o dummy.o
objdump -s -d dummy.o > dotproduct_03.s

```

由于 `char` 类型对应的点乘函数 `dotproduct_plain_i8` 的与 O2 优化后的代码差别不大，这里我们直接展示变化较大的 `int` 类型对应的点乘函数 `dotproduct_plain_i32` 的汇编代码：

```

0000000000000470 <dotproduct_plain_i32>:
# ..... preparation for function call
4ad: 00 00 00
4b0: f3 0f 6f 0c 07      movdqu (%rdi,%rax,1),%xmm1
4b5: f3 0f 6f 1c 06      movdqu (%rsi,%rax,1),%xmm3
4ba: f3 0f 6f 04 07      movdqu (%rdi,%rax,1),%xmm0
4bf: f3 0f 6f 2c 06      movdqu (%rsi,%rax,1),%xmm5
4c4: 48 83 c0 10          add   $0x10,%rax
4c8: 66 0f 73 d3 20      psrlq $0x20,%xmm3
4cd: 66 0f 73 d1 20      psrlq $0x20,%xmm1
4d2: 66 0f f4 cb          pmuludq %xmm3,%xmm1
4d6: 66 0f f4 c5          pmuludq %xmm5,%xmm0
4da: 66 0f 70 c9 08      pshufd $0x8,%xmm1,%xmm1
4df: 66 0f 70 c0 08      pshufd $0x8,%xmm0,%xmm0
4e4: 66 0f 62 c1          punpckldq %xmm1,%xmm0
4e8: 66 0f 6f cc          movdqa %xmm4,%xmm1
4ec: 66 0f 66 c8          pcmpgtq %xmm0,%xmm1
4f0: 66 0f 6f d8          movdqa %xmm0,%xmm3
4f4: 66 0f 62 d9          punpckldq %xmm1,%xmm3
4f8: 66 0f 6a c1          punpckhdq %xmm1,%xmm0

```

```

4fc: 66 0f d4 d3          paddq %xmm3,%xmm2
500: 66 0f d4 d0          paddq %xmm0,%xmm2
504: 48 39 c1            cmp    %rax,%rcx
507: 75 a7                jne    4b0 <dotproduct_plain_i32+0x40>
# ..... process reamianing elements and return

```

相比 O2 优化后的代码，O3 优化后的指令与跳转数大幅增加，不过为什么效率反而更高呢？实际上如果我们仔细观察 O3 优化后的代码，会发现它减少了循环的次数，使用了 SIMD 指令 pmuludq 等来并行计算多个数据的乘积，并且使用了 pshufd 指令来对数据进行重排，这样可以减少访存次数，提高计算效率。

Intel® Intrinsics Guide 对 pmuludq 指令的描述如下：

```

FOR j := 0 to 1
  i := j*64
  dst[i+63:i] := a[i+31:i] * b[i+31:i]
ENDFOR

```

这个指令会将两个 128 位的 xmm 寄存器取每个 64-bit 段的低 32 位按无符号整数乘法相乘，并将结果作为无符号 64 位整数存储在目标寄存器的对应位置。这个指令一次可以计算两个 32 位整数的乘积，在循环体中，我们可以看到这个指令用了两次，意味着一次循环可以计算四个 32 位整数的乘积。当然，如果向量大小不是 4 的倍数，汇编中还会使用 imul 指令来计算剩余的元素乘积。

§2.4.2.3. Linux 程序运行

相比 Java 需要在虚拟机 JVM 中创建程序进程，C 程序可以直接在 Linux 系统中运行。GCC 最后编译得到的二进制文件格式叫做 ELF(Executable and Linkable Format)，这个格式是 Linux 系统中可执行文件的标准格式。比如这是我通过 readelf 命令查看 dot 的 ELF 文件头部信息：

ELF Header:

```

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: DYN (Position-Independent Executable file)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Entry point address: 0x1220
Start of program headers: 64 (bytes into file)
Start of section headers: 23808 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 13
Size of section headers: 64 (bytes)
Number of section headers: 31
Section header string table index: 30

```

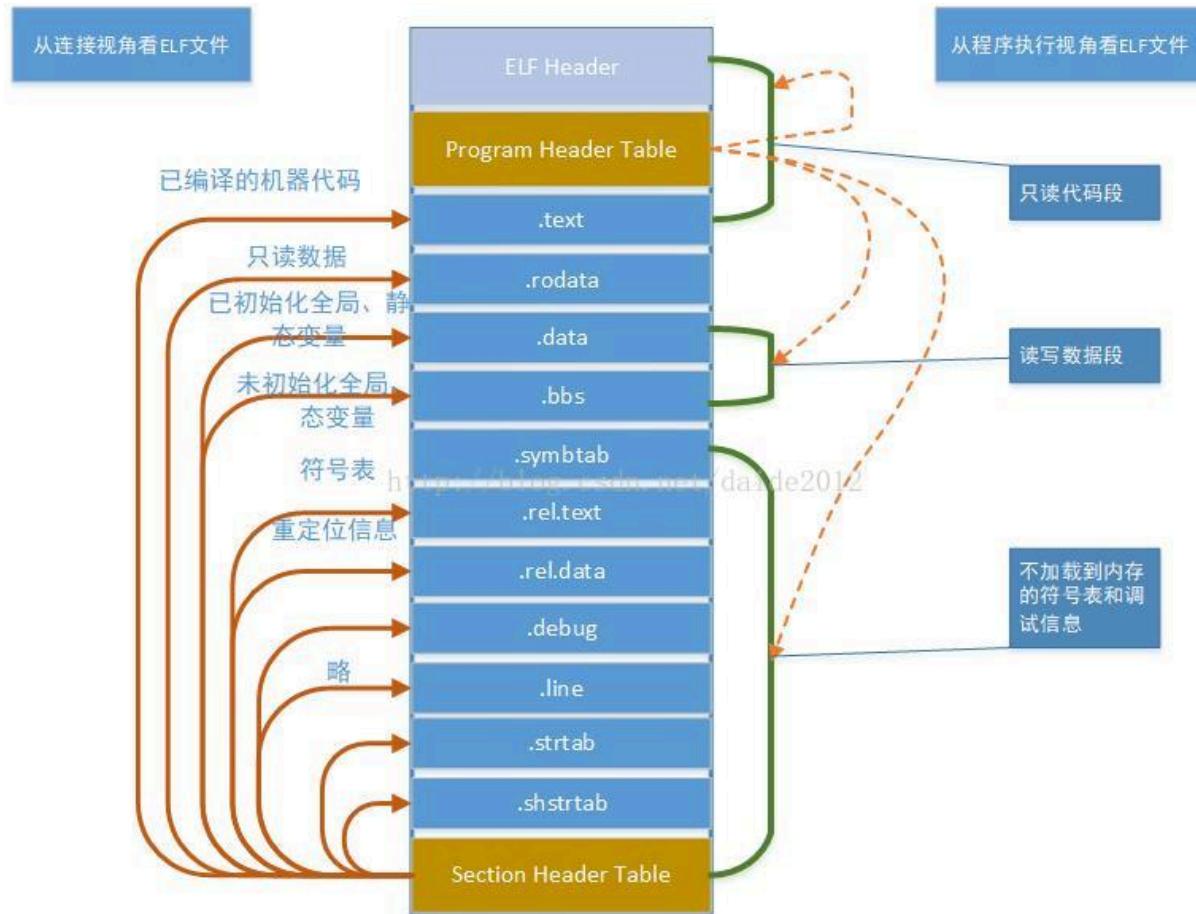
Section Headers:

```
... Sections ...
```

Program Headers:

```
... Program Headers ...
```

首先 ELF Header 中包含了 ELF 文件的基本信息，包括文件类型、机器架构、入口地址、节区表偏移等。

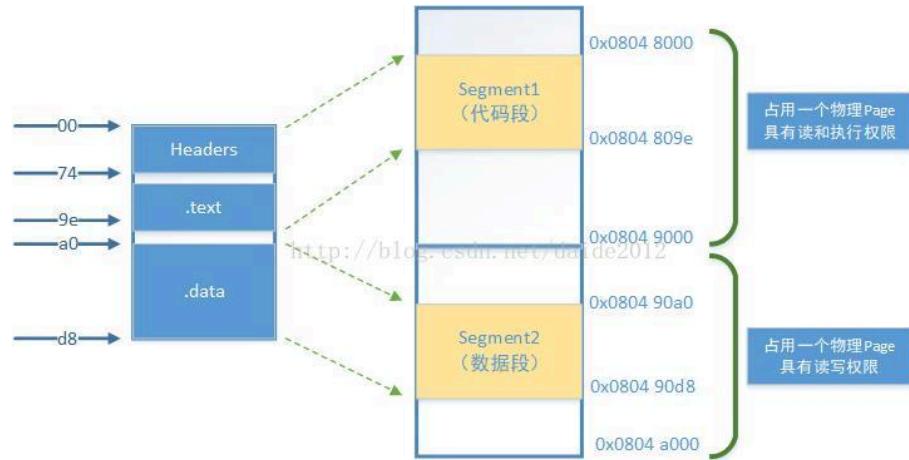


其中，这里告知了两个比较重要的 Header 的偏移量：Section Header 和 Program Header。Section Header 是面向汇编器和链接器的 table，包含了所有的节区信息，主要的节区

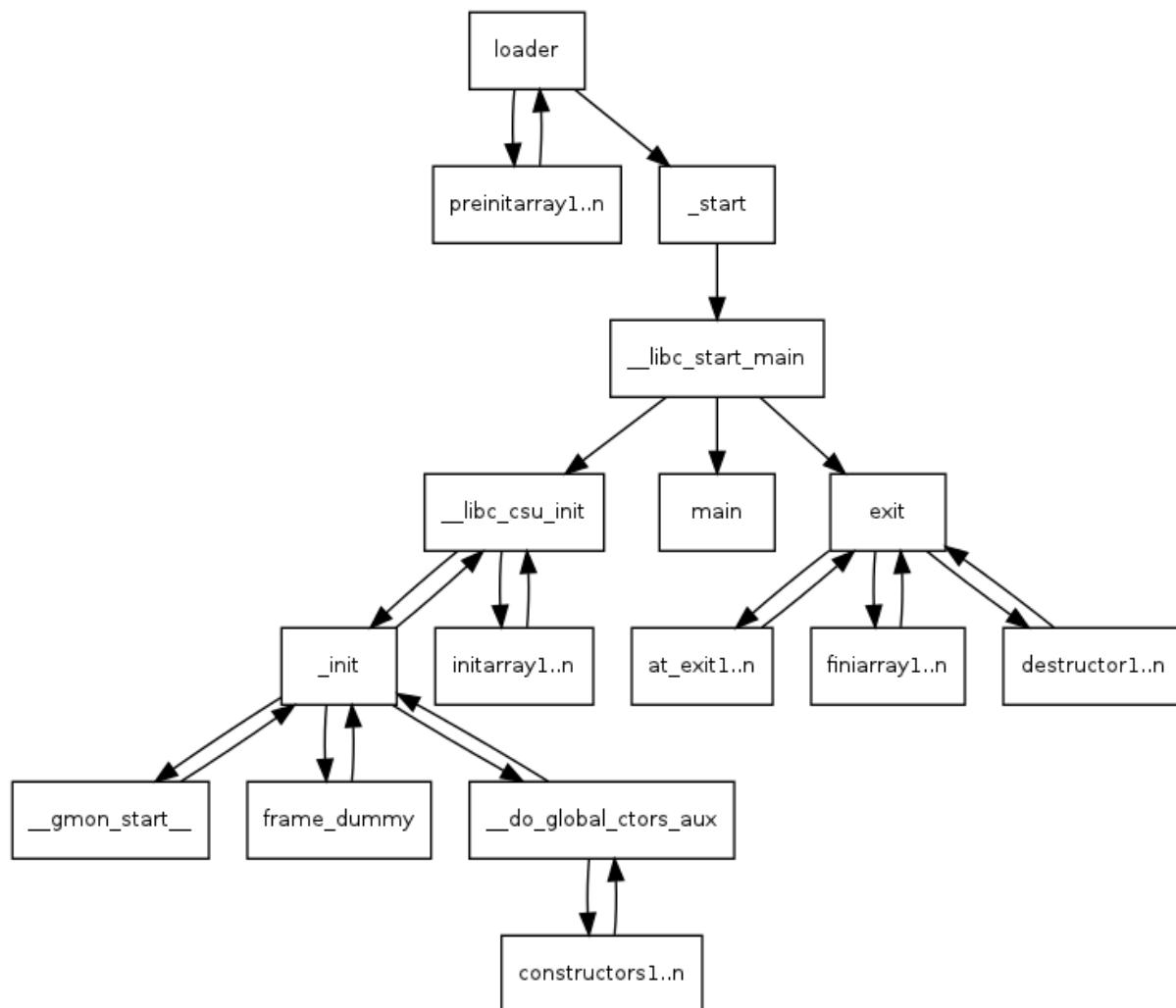
- .text: 存放程序的代码段
- .rodata: 存放只读的数据
- .data: 存放初始化的全局变量和静态变量
- .bss: 存放未初始化的全局变量，这里的变量在程序运行时会被初始化为 0
- .symtab: 存放符号表
- .shstrtab: 存放节区名字字符串
- ...

这些节区为链接器通过.h 文件将不同的源文件链接到一起提供相关信息。比方说我们所熟知错误是一个头文件中声明了一个函数，但是没有在源文件中定义，或者函数签名与定义不一致，链接器就会报错 `undefined reference to xxx`。

而 Program Header 则是面向加载器的 table，包含了所有的段信息，主要的段相比节区多了一些权限信息，比如可读、可写、可执行等。根据内存分页机制，不同权限的段都会在被映射到内存中的不同页，这时符号表中的相对地址也会相应修改为系统中的实际地址。



接下来，在我们敲下 `./dot` 命令后，shell 程序会执行 Linux `execve()` 系统调用来加载并执行这个 ELF 文件。这意味着一个新的子进程会被创建，然后 Linux 内核会分配一些进程资源，比如进程 ID、文件描述符、内存空间等。等到一切准备就绪后，系统特权级别会从用户态切换到内核态，然后进入 `libc` 库提供的 `_libc_start_main` 函数，这个函数会初始化一些环境变量，然后调用我们的 `main` 函数。这个示意图展示了整个过程：



我们对比 Java 和 C 程序的运行过程，可以看出 Java 程序的运行过程中多了一层 JVM 虚拟机来模拟一个执行环境，而 C 程序则直接在 UNIX 系统中运行，并且支持各种底层系统调用，这也

是我们普遍认知中 C 程序运行速度快的原因之一。并且相较于 Java 的 JIT 分层优化需要多次执行才能达到最优状态，C 程序的编译器优化则可以在编译时就指定优化等级，这样可以更好地控制程序的性能。

不过这种高效率并非没有代价，由于各种内存操作、指针操作等都会直接在内核态中执行，如果发生错误，就会直接被操作系统捕获并终止程序。除此之外，为何 C 程序发生的内存非法操作经常被报告为 Segmentation Fault(段错误)也容易理解了——因为程序使用的内存都来自于该进程被分配的虚拟内存段，如果程序访问了段之外的内存或者栈指针越界，就会收到系统的 SIGSEGV 信号，进程被终止。

§3. Experiment 2

§3.1. 实验配置

由于我的两台电脑 CPU 核心数都比较少，为了能观察到更高线程数的运行结果，我直接选择使用了南科大的服务器进行实验。服务器配置如下：

Item	Details
System	Ubuntu 20.04.2 LTS
OS Kernel	5.4.0-205-generic
gcc-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
g++-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
JDK-version	openjdk version “11.0.26” 2025-01-21
CPU	Intel(R) Xeon(R) E5-2680 v4 CPU @ 2.30GHz
CPU-cores	14 cores
CPU-cache-size	35840 KB
Memory	251.77GB

§3.2. 实验步骤与方法

我选择了从 1 到 32 的线程数，数据规模阶数从 10^2 到 10^8 ，向量值随机生成范围仍为 $[-2, 1]$ ，使用以下脚本进行测试与数据收集：

```
#!/bin/bash
DATA_TYPES=("char" "short" "int" "float" "double")
SIZES=(100 1000 10000 100000 1000000 10000000 100000000)
OPT_LEVELS=("00")
NUM_THREADS=(1 2 4 8 16 32)
SRC_DIR="..../src"
BUILD_DIR="..../build"
CSV_DIR="..../results_2"
mkdir -p $CSV_DIR

for opt in "${OPT_LEVELS[@]}"; do
    gcc -$opt -march=native -fopenmp $SRC_DIR/dotproduct.c -o $BUILD_DIR/dot_$opt -
    Wall

    # print header for .csv
    echo "Num_Threads,Size,Data Type,Time" > $CSV_DIR/results_c_$opt.csv
```

```

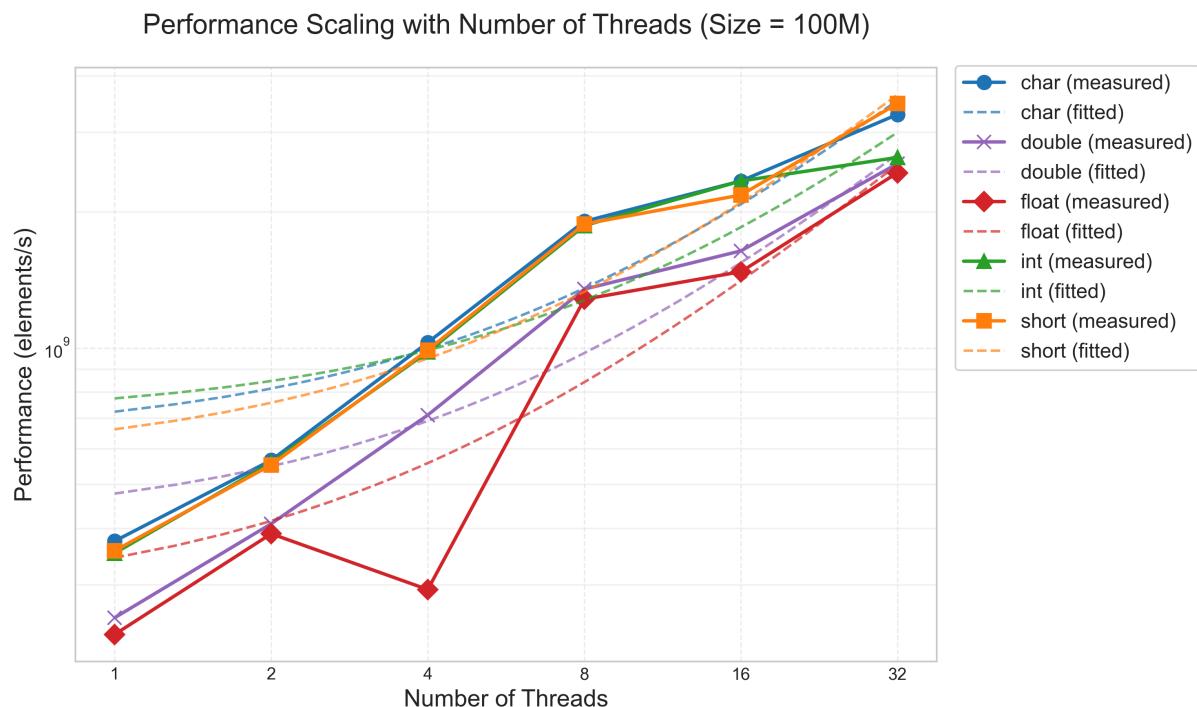
for threads in "${NUM_THREADS[@]}"; do
    for dtype in {0..4}; do
        for size in "${SIZES[@]}"; do
            echo "Testing $opt ${DATA_TYPES[$dtype]} with size $size and $threads
threads"
            echo -n "$threads,$size,${DATA_TYPES[$dtype]}," >> $CSV_DIR/
results_c_$opt.csv
$BUILD_DIR/dot_$opt $size $dtype $threads >> $CSV_DIR/
results_c_$opt.csv
        done
    done
done
done
done

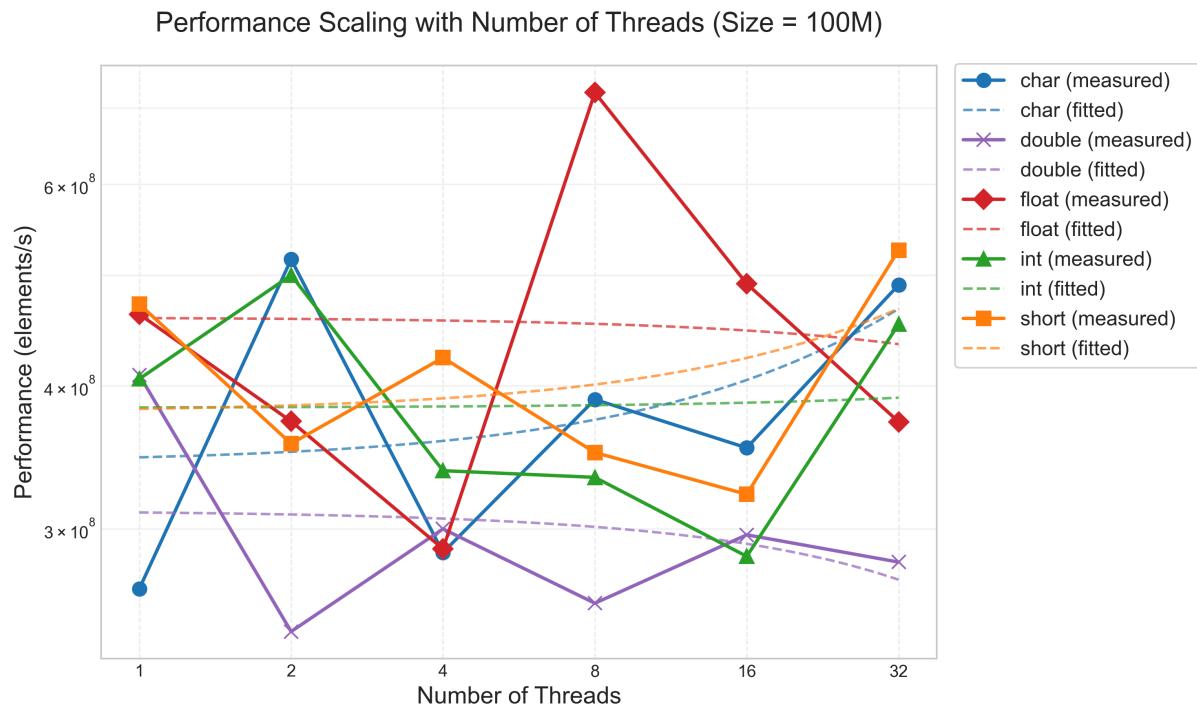
```

注：C 程序的多线程的计时方法相比与单线程做了一些调整，由于多线程的 CPU time 计算的是所有线程的 CPU 时间之和，所以我使用了 `omp_get_wtime()` 函数来计算多线程的 Wall-time 运行时间，这样可以更好地反映多线程的实际运行效果。

§3.3. 实验结果

以线程数为横坐标，性能表现为纵坐标，绘制了不同数据类型下的 C 和 Java 程序的性能曲线图：





可以看出，对于 C 程序，随着线程数增加，程序性能基本上保持线性增长，不同数据类型之间差别不大；而对于 Java 程序，随着线程数增加，性能增长并不明显，甚至在线程数较高时性能反而下降。

这个结果又让我困惑了，同样的系统，同样的硬件，Java 的 JVM 进程难道不能像 C 一样充分利用多个 CPU 核心来分配计算任务吗？我决定针对这个稍显特别的结果进行一些学习与研究。

§3.4. 实验思考

§3.4.1. 程序的实例——进程与线程

在研究多线程编程之前，我们先来了解一下它的基础——进程。在一个操作系统上，一个进程就是被加载到内存中并在 CPU 上执行的程序实例。为了决定一个进程的执行顺序，操作系统会为每个进程分配一个进程控制块(Process Control Block, PCB)。^[1]进程控制块 PCB 相当于一个结构体，用来存储进程的各种信息，比如

- **Process state**
- Program counter
- CPU registers
- CPU scheduling information
- ...



Figure 3.3 Process control block (PCB).

而 Process state 则决定 CPU 在调度队列中是如何处理这个进程的，通常有以下几种状态：

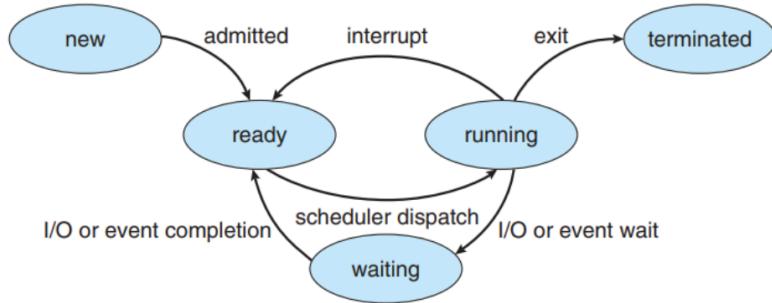
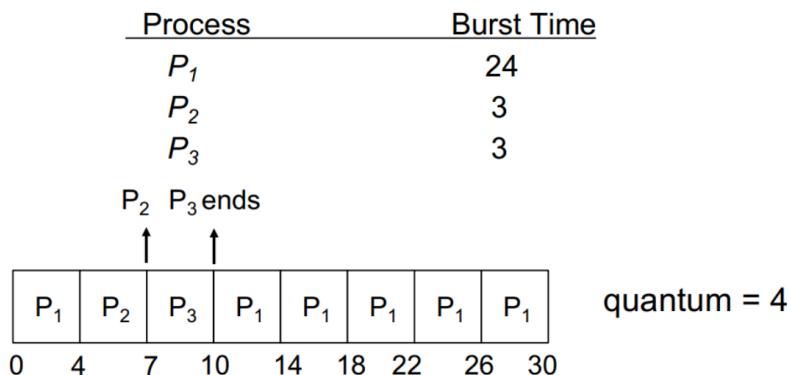


Figure 3.2 Diagram of process state.

当我们只有一个处理器时，如果只是同步地按顺序执行多个进程，那么 CPU 在执行一个进程时，其他进程只能处于 ready 状态，等待 CPU 的调度。这种情况对计算密集型任务来说是非常低效的，因为 CPU 在执行一个进程时，其他进程的计算资源就被浪费了。

这时 CPU 分片机制出现了，它规定了一个最小的执行时间单位，然后 CPU 在这个时间单位内执行一个进程，然后切换到另一个进程，这样就可以实现多个进程的并发执行。这种机制叫做时间片轮转调度算法。



这种复用方式看似可取，但我们到现在还没有引入一个不可忽视的问题，就是进程切换时的时空开销。

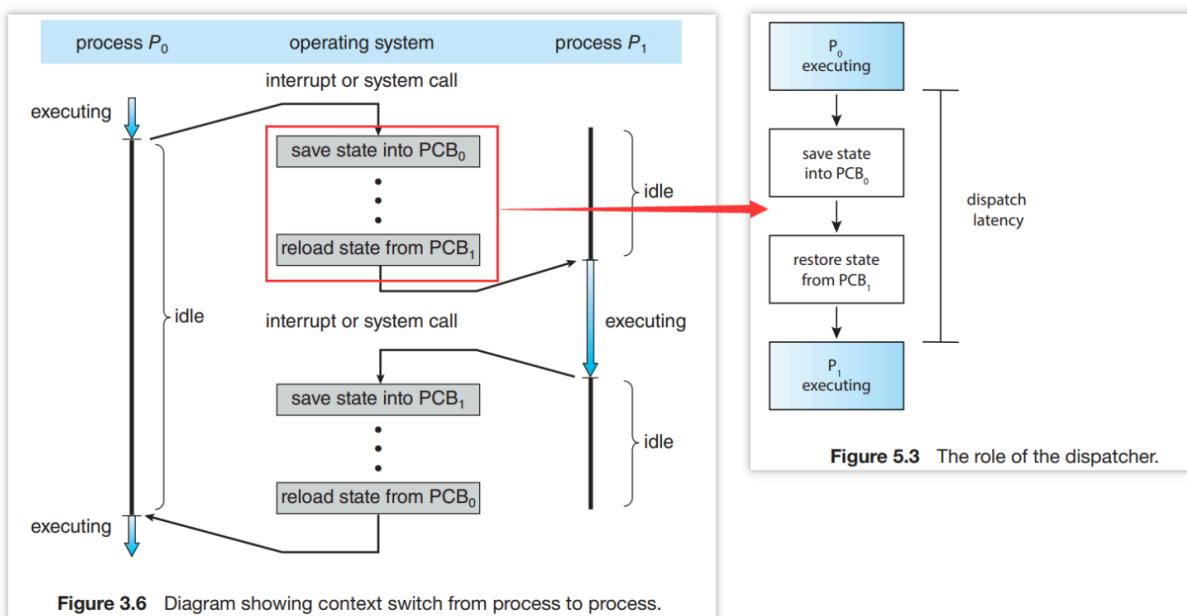
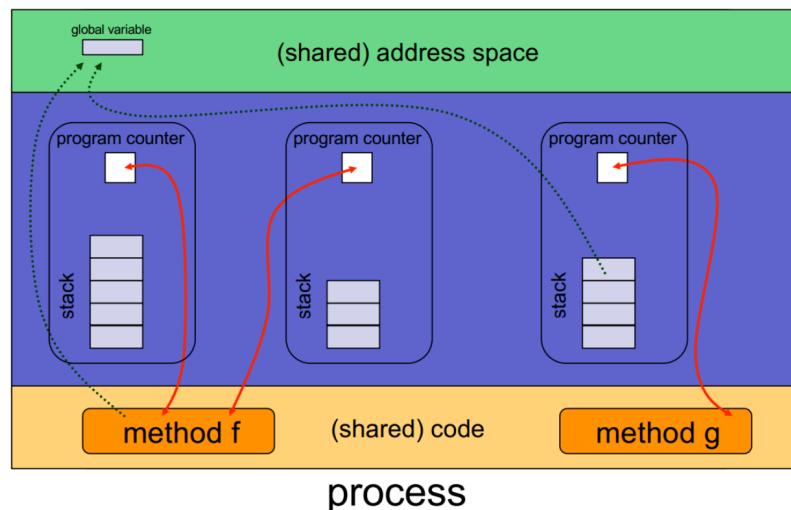


Figure 3.6 Diagram showing context switch from process to process.

进程切换的过程被称为上下文切换(Context Switch)，它包括了保存当前进程的状态、加载新进程的状态等操作，这些操作都需要耗费 CPU 时间，并且进程的上下文信息保存在内存中也需要占用大量的空间。而且进程切换时，CPU 的 cache 中的数据也会被清空，这样会导致大量的 cache miss，进而影响程序的性能。

为了简化进程所需要的资源，系统设计者模仿进程的概念，引入了更加轻量化的线程。线程(Thread)是进程的一部分，它也是一个程序的执行实例，拥有自己的 thread ID、register set、stack、和 state。但是线程与进程的区别在于，线程共享进程的资源，比如进程的内存空间、文件描述符等。这样，线程的创建、销毁、切换的开销都会比进程小很多。下面这个图就表示了线程与对应的进程之间的关系：



多线程相比单线程程序有以下几个优势：

- Economy: 线程的创建、销毁、切换的开销都比进程小很多
- Resource sharing: 线程共享进程的资源，比如内存空间、文件描述符等
- Responsiveness: 线程可以提高程序的响应速度，比如一个线程阻塞时，其他线程仍然可以继续执行
- Scalability: 线程可以更好地利用多核 CPU，提高程序的性能

线程的实现方式依赖于操作系统的支持，它分为两种： **User-level threads** 和 **Kernel-level threads**。

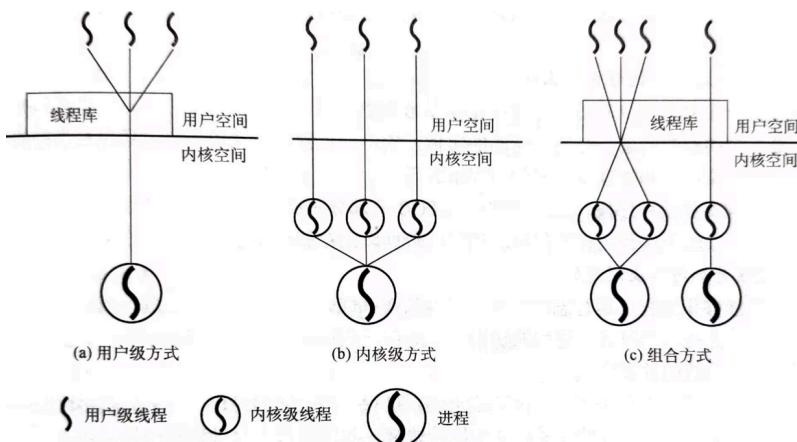


图 2.5 用户级和内核级线程

User-level threads 是由用户程序库实现的，操作系统并不知道线程的存在，而且实际上每个线程都对应到一个进程，这样线程的切换只涉及到 User Mode 下的调度算法，切换开销很小。但是这种方式的最大的缺陷是，由于所有线程都在同一个进程中，那么自然也不能使用 CPU 的多核资源来分配计算任务。

而 Kernel-level threads 则是由操作系统内核实现的，每个线程都会被映射到一个内核线程，这种方式可以更好地利用多核 CPU，提高程序的性能。

§3.4.2. 多线程的数据竞争

我们之前一直接触的是同步编程(Synchronous Programming)，这种编程方式是顺序执行的编程方式，PC 的转移只由当前单个 CPU 时钟进行驱动，很明显不会涉及到两段代码操作同一块内存产生冲突的问题；不过多线程编程就可能引发 race condition 和 deadlock 问题，这些都可能影响我们最后计算得到的结果

比如对于 Kernel 的 CriticalSection 问题：考虑有 n 个线程，每个线程都执行一段代码，这段代码会对一个共享的变量进行操作，这段代码称为 Critical Section。如果两个线程同时进入了 Critical Section，那么就会产生数据竞争，这样就会导致程序的结果不确定。

```

while (true) {
    entry section
    critical section
    exit section
    remainder section
}
```

Figure 6.1 General structure of a typical process.

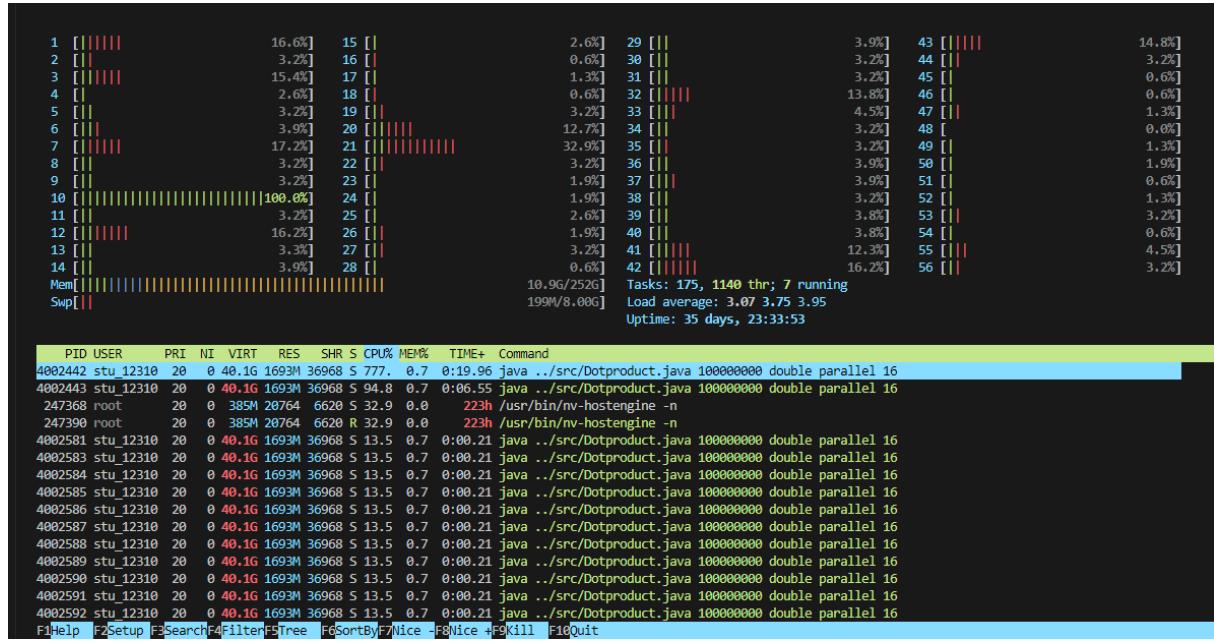
为了解决 Critical Section 问题，我们可以让各个线程进行同步，即规定线程之间的执行顺序，从而避免非法情况的发生。常见的同步方式有：

- Mutex: 互斥锁，一次只允许一个线程进入临界区
- Semaphore: 信号量，可以控制多个线程同时访问临界区
- Condition Variable: 条件变量，可以让线程等待某个条件满足后再继续执行
- Barrier: 屏障，可以让多个线程在某个点上同步

当然，在线程内部定义一个私有的变量，直接消除 Critical Section 也是一种解决方案，这种方式叫做 Thread-local Storage。我们在 C 的 openmp 中使用 reduction 子句进行线程规约就是这种方式的应用。reduction 子句需要指定规约变量和规约操作，然后编译器会自动为每个线程分配一个私有的变量，最后将所有线程的结果合并到一个全局变量中。Java 里需要手动实现这种方式，这就是需要提前定义一个长度为 num_threads 的缓存数组的原因，在每个线程中会在计算后把线程结果存入缓存的对应位置，最后再合并到全局变量中。

§3.4.3. Java 的多线程实现初探

我原本以为，Java 线程数增加而程序性能几乎没有变化，是因为 JVM 虚拟机内部的多线程实现属于 User-level threads，这也就对上了在核心数较多的服务器上运行，也没有发挥出多线程的优势。不过，在我继续扩大数据规模并观察服务器运行脚本时系统占用的情况时，发现并非如此：



第一个高亮显示的是运行参数为 Size = 1e9, Data Type = double, Num_Threads = 16 的 Java 点乘程序进程，可以看到 CPU 占用率达到了 777%，我们知道 CPU 占用率的计算是 $cpu\ usage\ rate = ((utime+stime) - (lastutime + laststime)) / (period * sysconf(_SC_CLK_TCK))$ ，如果超过了 100%，这意味着不止一个 CPU 逻辑核心在运行这个程序。上面的猜想不攻自破，Java 程序的多线程实现并不是 User-level threads，而是实实在在的 Kernel-level threads。

目前看来想要仅通过程序内计时分析是无法窥探 Java 程序性能下降的原因的，我决定使用更加专业的工具来分析 Java 程序的多线程是如何在 JVM 中进行的。

§4. Experiment 3

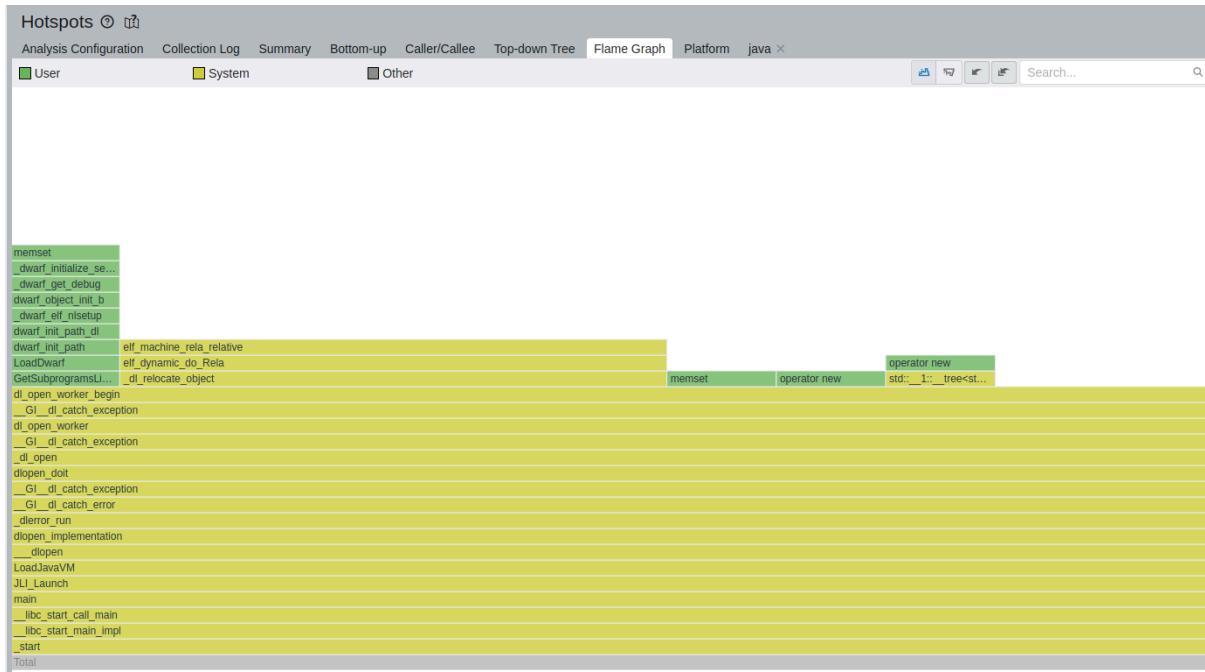
§4.1. 实验配置

该实验在台式机的 Ubuntu 系统上进行，配置如下：

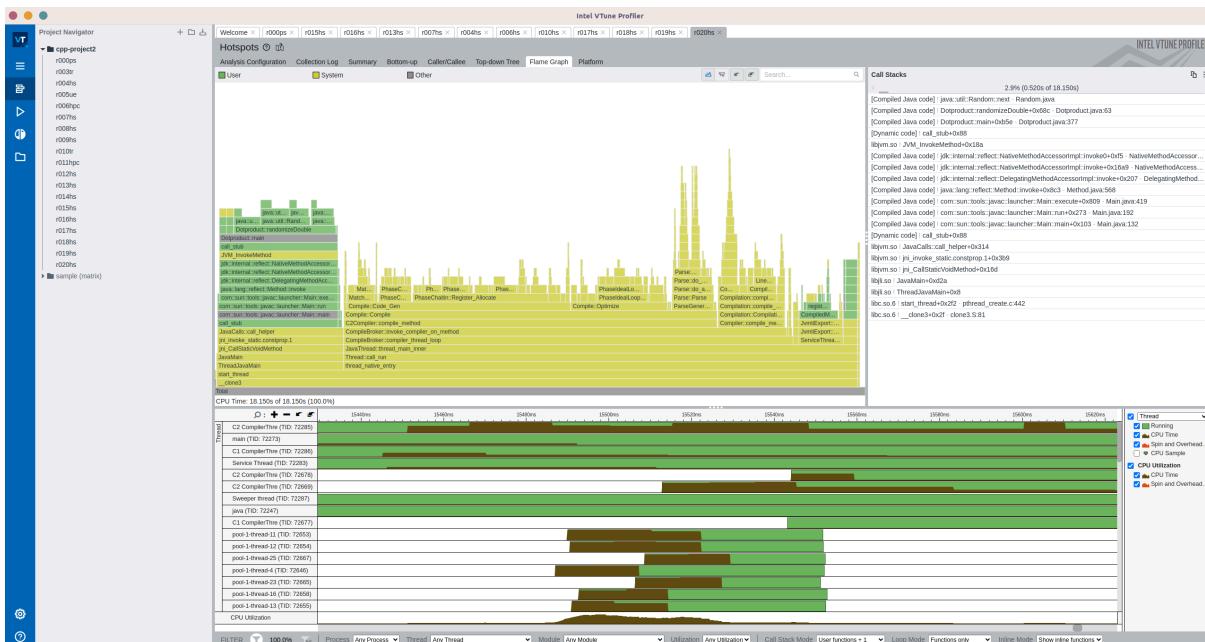
Item	Details
System	6.8.0-52-generic Ubuntu 22.04.1 LTS
OS Kernel	5.15.167.4-microsoft-standard-WSL2
gcc-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
g++-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
vtune-version	2025.1.0
JDK-version	openjdk 21.0.2 2024-01-16 LTS
CPU	Intel(R) Core(TM) i5-13600KF CPU 2.60GHz
CPU-cores	14 cores
CPU-cache-size	24576 KB
Memory	31.968 GB

§4.2. 实验步骤与方法

我按照 Intel 官网的方法，在 Ubuntu 系统中安装了 Intel VTune Profiler，并且以 HotSpot 分析器的方式来分析 Java 程序的在多线程下的性能表现。



最开始的函数调用是一个叫 `_start` 的函数，这就和我们之前在 Linux 系统中的进程启动过程相符，说明 JVM 的启动函数是 C 语言实现的。然后是 `_libc_start_main` 函数，这个函数是 C 语言程序的入口函数，它会初始化一些环境变量，然后调用我们的 `main` 函数。进而启动了 Java 虚拟机，这个过程中会加载 JVM 的一些库文件，比如 `libjvm.so`，然后通过 `_clone3` 系统调用来创建一个新的线程来执行 Java 程序。



可以看到，Java 程序运作时，Profiler 追踪到了 JVM 启动的大量线程，包含了 C2 Compiler, Service Thread, GC Thread 等，这些线程占去了大部分的 User Time。接下来 `thread_native_entry` 这个函数耗时最长，不过这个函数并不会是我们主类 `DotProduct` 的入口函数。根据资料，它其实是 JVM 通过 `pthread_create` 传入的函数指针，用以创建 Java Native 线程。

```

bool os::create_thread(Thread* thread, ThreadType thr_type,
                      size_t req_stack_size) {
    // 创建操作系统线程
    // ...

    ThreadState state;
    {
        pthread_t tid;
        // =====
        // 调用系统库创建线程, thread_native_entry为本地Java线程执行入口
        // =====
        int ret = pthread_create(&tid, &attr, (void* (*)(void*)) thread_native_entry,
                                thread);

        if (ret != 0) {
            // Need to clean up stuff we've allocated so far
            thread->set_osthread(NULL);
            delete osthread;
            return false;
        }

        // Store pthread info into the OSThread
        osthread->set_pthread_id(tid);

        // Wait until child thread is either initialized or aborted
        // ...
        return true;
    }
}

```

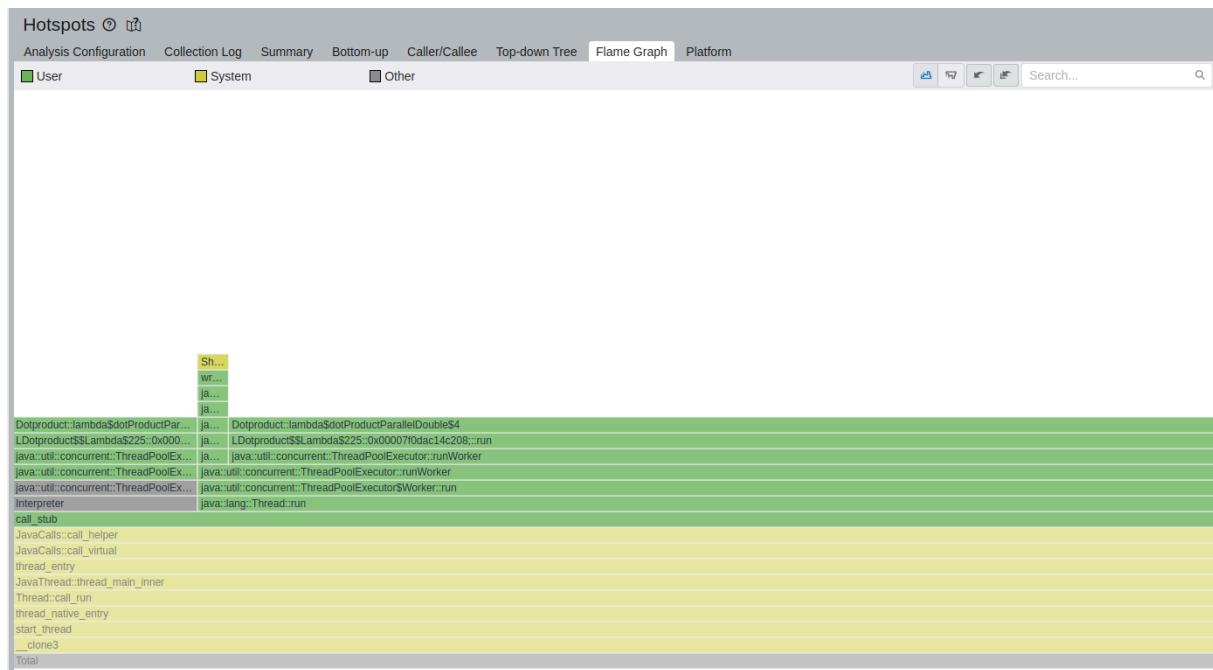
`pthread_create` 函数作用是创建一个线程，它的第三个参数是线程运行函数的起始地址，第四个参数是运行函数参数。`pthread_create` 实际上是调用了 `clone()` 完成系统调用创建线程的，所以目前 Java Native 线程在 Linux 操作系统下为 1:1 的内核级线程模型。随即，被创建的线程会一路调用到 `JavaThread::thread_main_inner` 函数，这个函数是 Java 线程的入口函数。

```

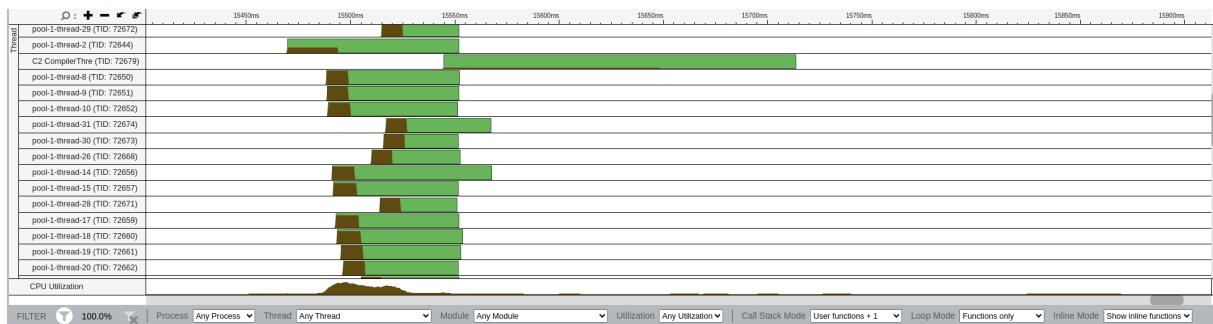
void JavaThread::thread_main_inner() {
    if (!this->has_pending_exception() &&
        !java_lang_Thread::is_stillborn(this->threadObj())) {
    {
        ResourceMark rm(this);
        this->set_native_thread_name(this->get_thread_name());
    }
    HandleMark hm(this);
    this->entry_point()(this, this);
}
DTRACE_THREAD_PROBE(stop, this);
this->exit(false);
delete this;
}

```

`entry_point` 函数对应 `thread_entry` 函数，这个函数会通过 `JavaCalls::call_virtual` 启动 Java 线程的 `run` 方法，这样 Java 线程就开始执行了。在这个调用栈中，可以看到除了 `java.lang.Thread.run` 函数外被执行，还有标识为 `Interpreter` 的函数，这说明，Java 程序的确会进行一部分解释执行，而不会全部编译成机器码执行。最后终于到达了实际执行计算操作的匿名函数 `DotProduct::lambda$dotProductParallel$4`，他所占的 CPU time 比例也是最大的。

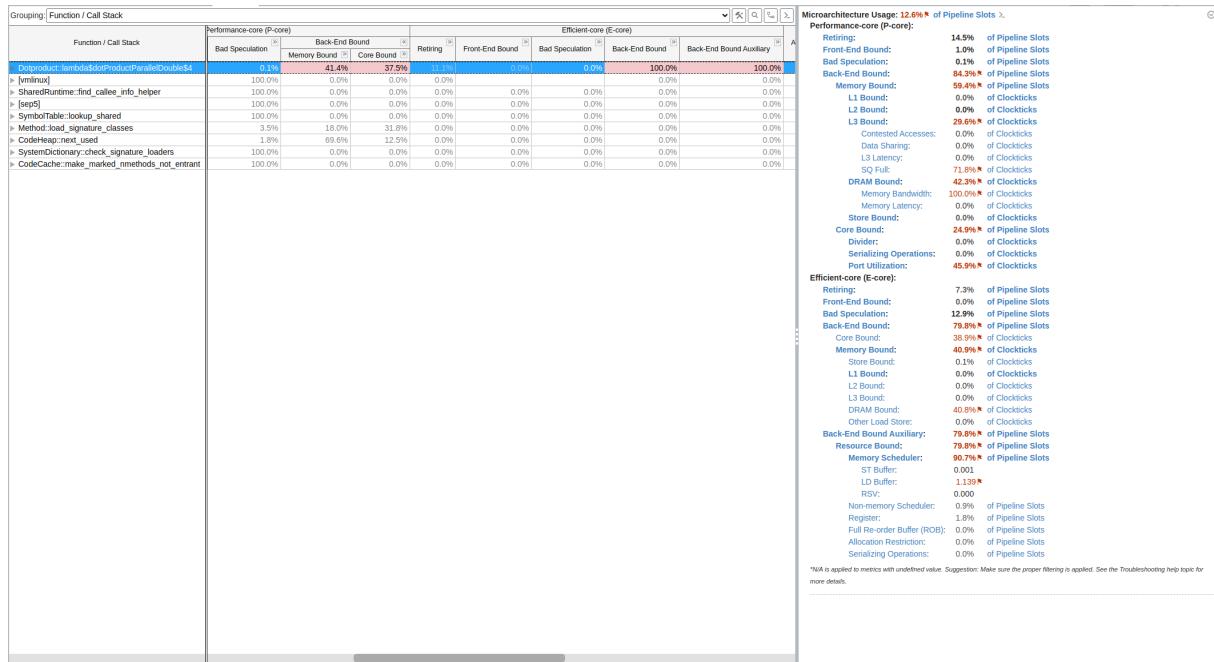


那么说了这么多，Java 程序的多线程性能为何提升不明显？我在这里找到了答案



在这个线程 CPU time 分析中，pool-1-thread-N 对应线程池中第 N 个线程，而棕色的部分就是线程的 CPU utilization 比例，可以看到，线程池中的线程并不是同时开始的，由于线程分配时间开销大，这些启动时间差甚至达到了几十毫秒，而主方法需要等待所有线程都结束后才能继续执行，这就导致了总的线程时间开销很难减少；同时，我们可以明显观察到，在线程数为 32 时，单个线程 CPU utilization 并不高，线程基本上处在等待状态，有些线程甚至几乎没有被使用，这就导致了线程数增加时实际上没有很好地利用多核 CPU 的来提高效率。

同样地，我对并行计算过程展开了 Microarchitecture Usage 分析，右侧即为计算中的各种微结构的运行瓶颈，



Microarchitecture Usage 分析是根据 Intel 公司的 Top-Down Model 理论所构建的，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。通过硬件性能计数器来分析程序的性能瓶颈，这个模型分为四个层次：

- **Frontend Bound:** 指令前端瓶颈，主要是指令缓存、分支预测、指令解码等
- **Bad Speculation:** 错误预测，主要是分支预测错误导致的性能损失
- **Retiring:** 指令退休，主要是指令执行的瓶颈
- **Backend Bound:** 指令后端瓶颈，主要是数据缓存、数据总线等

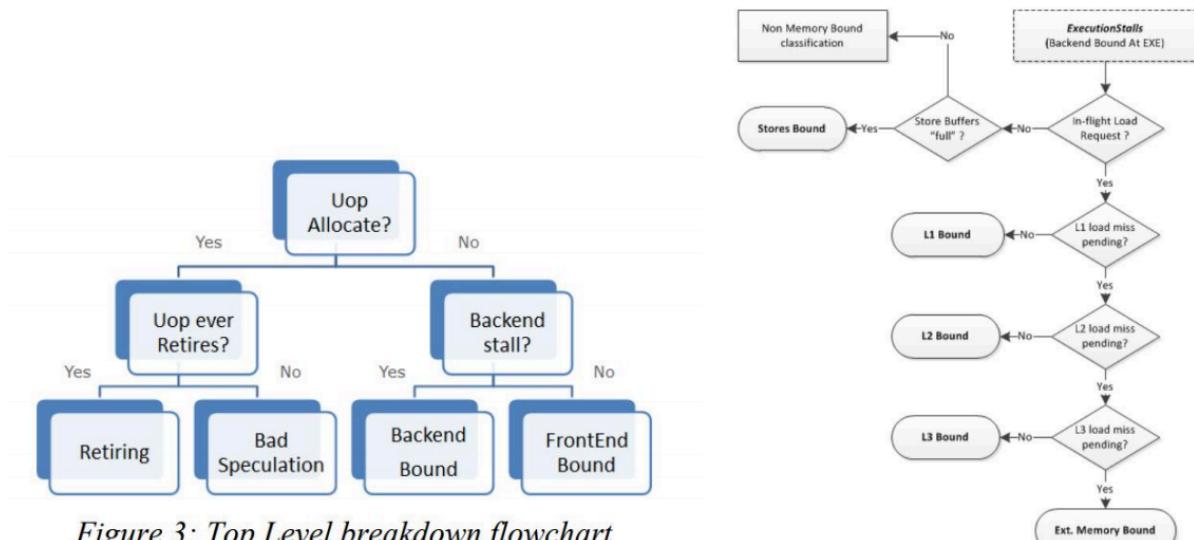
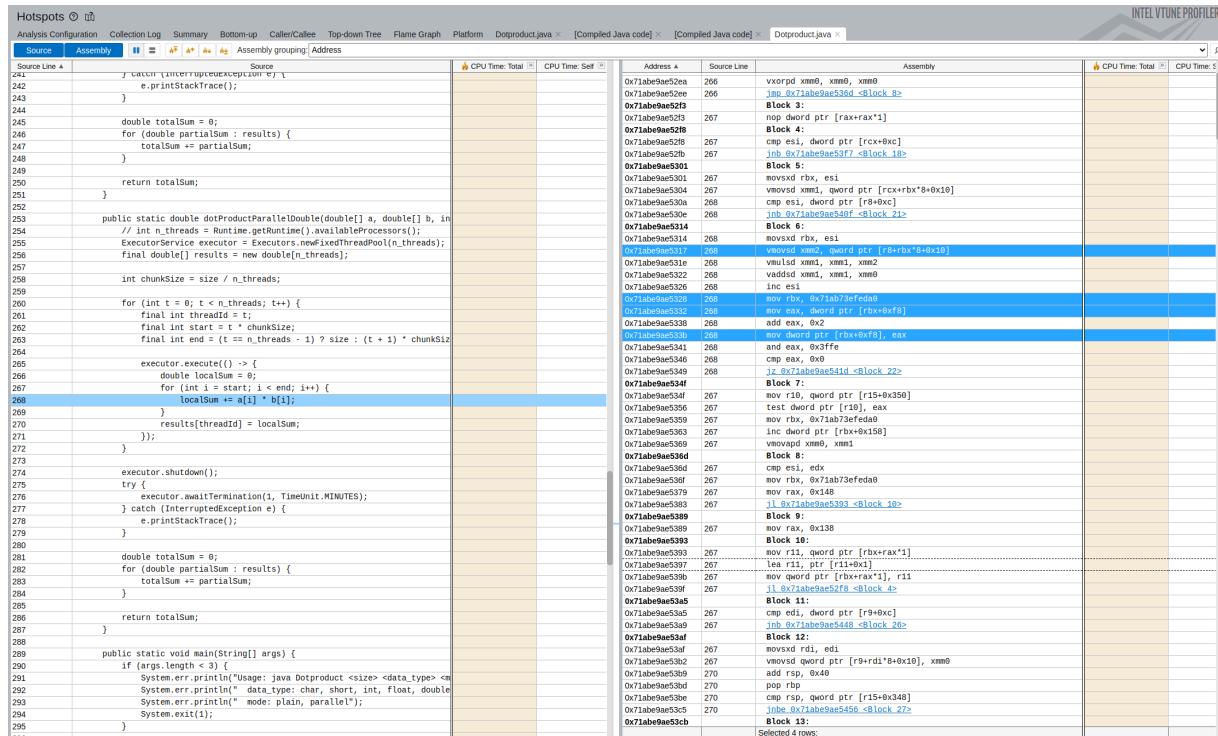
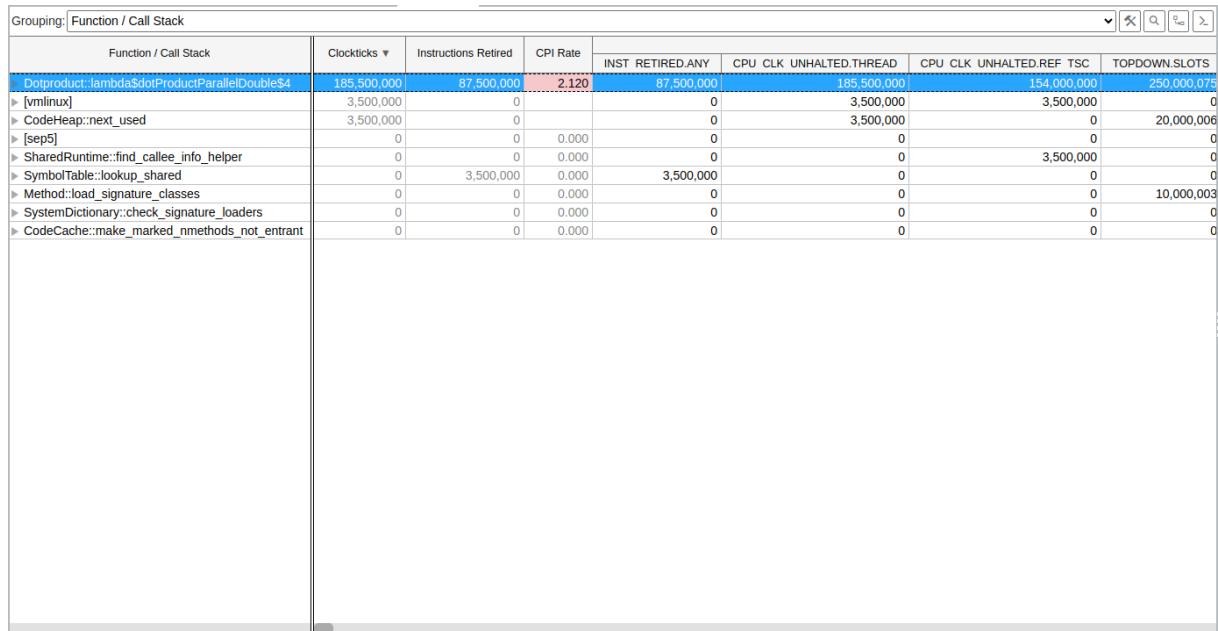


Figure 3: Top Level breakdown flowchart

Java 程序的性能瓶颈主要在 Core Bound 和 Memory Bound 上，Core Bound 主要是由于 CPU Port utilization 并不高，一个 CPU 周期内只有少量 Port 用来执行，缺乏并行，通常使用 Vectorization 技术可以提高 CPU Port 的利用率；Memory Bound 在 Performance Core 上来源于 L3 cache bound 和 DRAM bound，我们可以通过下面的汇编代码来看看 Java 程序的访存情况：



Block 6 这部分中涉及到很多 `mov` 指令将数据从 DRAM 读取到寄存器中，由于 DRAM 访问 cycle 很高，这就导致了 Memory Bound 的性能瓶颈。

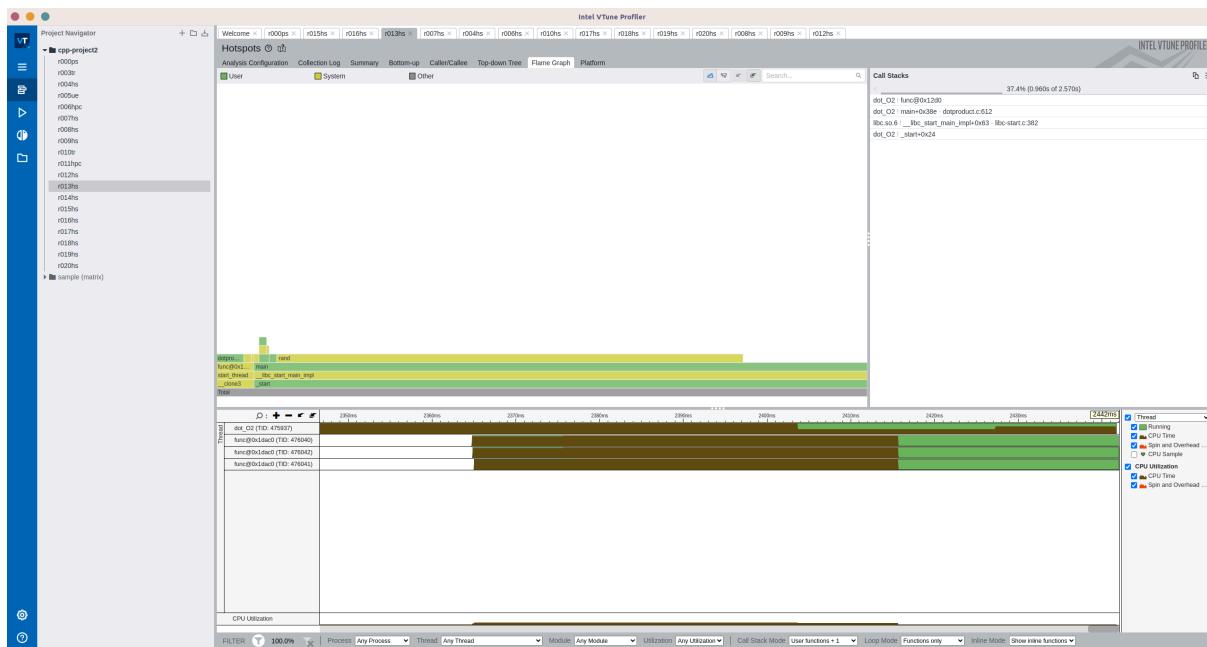


CPI(Cycles Per Instruction)是一个很好的评估程序性能的指标，它是指完成一个指令所需要的 CPU 周期数，CPI 越小，程序性能越好。在这个分析中，我们可以看到 Java 程序的 CPI 值在 2.1 左右，根据 CPU time 计算公式

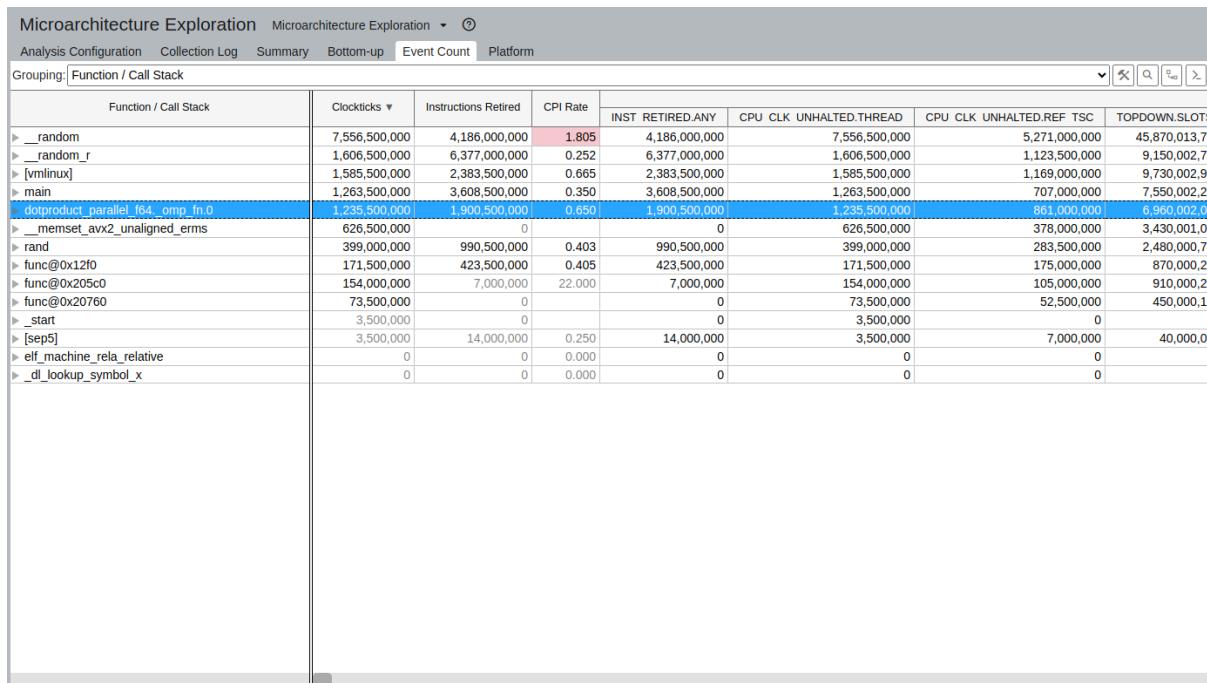
$$\text{CPU time} = \text{IC} * \text{CPI} * \text{Cycle Time}$$

如果 IC(Instruction Count)不变，CPI 越小，CPU time 就越小，程序性能就越好。

那么分析完了 Java 程序的多线程性能，我们再用类似的流程来看看 C 程序这边怎么样



可以看到 C 程序这里的调用栈相当简单，基本和我们最开始的讨论一致，也是从 `_start` 函数开始，然后调用 `main` 函数，最后到达 `dotproduct_parallel_f64` 函数，这个函数是我们的多线程计算函数的入口。随后在 `omp` 原语的作用下，系统通过 `libgomp` 库调用了 `GOMP_parallel` 函数，这个函数会创建一个线程池，然后调用 `GOMP_parallel_start` 函数来启动线程任务。和 Java 不同的是，C 程序的主线程也会承担一个线程的计算，所以实际上只需创建 3 个子线程，并且可以从热力图中看到，这三个线程启动时间差别不大，CPU utilization 都很高，这也说明了 C 程序的多线程实现相对于 Java 程序更加高效。



再来看看计算循环的 CPI 值，这个值小于 1，可以看出来指令的 retiring 率还是很高的，而且 CPI 值小也有利于更好的 Backend 侧的分支预测和指令执行。

§4.3. 总结

这个小实验虽然只是探索了 Intel VTune Profiler 的一部分功能，但是通过这个工具，我们可以更加直观地看到程序的各种函数调用情况和代码执行热力分布。通过这个工具，我们可以更好地了解程序的性能瓶颈，明确了 Java 程序多线程性能提升不明显的原因，来源于线程分配时间开销大，线程 CPU utilization 不高，以及 Memory Bound 的性能瓶颈。而 C 程序明显处理地更加直接，各计算线程几乎同时启动，CPU utilization 很高，CPI 值也很低，这些都是 C 程序多线程符合预期的原因。

§5. Experiment 4

讨论了这么多关于 Java 的优化，我们重新把视角拉回到本课程主要语言，看看如何用一些优化技巧把 C 程序的性能提升到媲美 O2 甚至 O3 优化等级的程度。

§5.1. 循环展开

联系到 O0 优化编译出的汇编代码，我们可以看到循环体内的计算是一个一个元素地进行的，这样会导致访存比很低，但如果我们将减少循环的次数，将一次循环的工作量增加，就可以减少循环中反复 read 和 write 内存的开销。这种方法就是循环展开(loop unrolling)。在做循环展开时，我们需要考虑步长的设置，这个其实和缓存的大小有关，对于速度比较快的 L1 缓存来说，一般是 64B(x86_64 平台)，也就是 8 个 int 型数据，所以我们可以设置步长为 8，这样可以在一次循环中正好读入一个 cache line 的数据，增加 cache 的命中率。

```
void dotproduct_unwind_i32(i32 *a, i32 *b, scalar_i *result, size_t size) {
    for (int i = 0; i < size; i += CACHE_SIZE) {
        *result += a[i] * b[i];
        *result += a[i + 1] * b[i + 1];
        *result += a[i + 2] * b[i + 2];
        *result += a[i + 3] * b[i + 3];
        *result += a[i + 4] * b[i + 4];
        *result += a[i + 5] * b[i + 5];
        *result += a[i + 6] * b[i + 6];
        *result += a[i + 7] * b[i + 7];
    }
    for (int i = size - size % CACHE_SIZE; i < size; i++) {
        *result += a[i] * b[i];
    }
}
```

对于主循环中剩余的元素，我们对其进行循环计算即可。其运算结果如下：

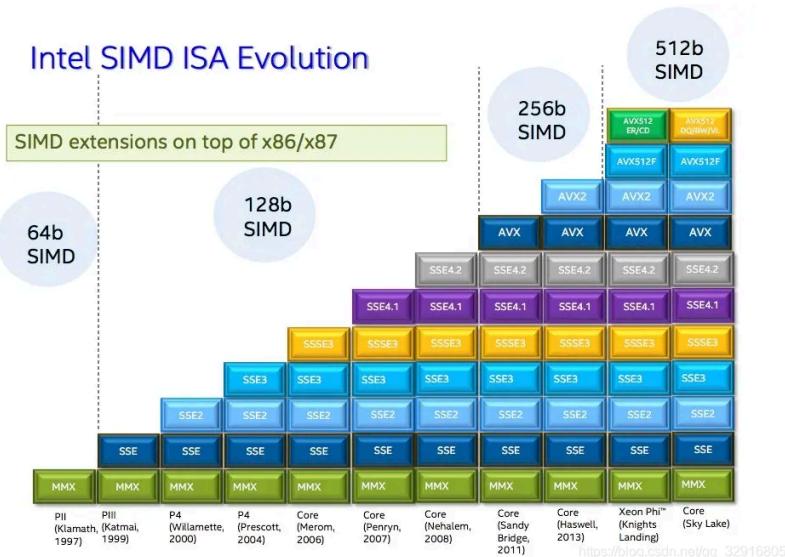
```
(~/code/CPP/project2)
(21:47:08 on main *)-> make run SIZE=100000000 TYPE=double
Running the program...
./build/dot 100000000 double 4
    == Plain ==
0.716468900
Dot product result: 24987831.547229
    == Unroll ==
0.653222900
Dot product result: 24987831.547229
```

不过可以看出，这种优化技巧对于 O0 优化的程序性能提升并不明显，而且实际上开到 O3 优化，循环展开也会影响 gcc 编译时对循环的优化，所以该方法想要达到最优性能可能性不大。

§5.2. SIMD 指令集 + register 变量

所谓指令集，就是CPU中用来计算和控制计算机系统的一套指令的集合，一般来说，每个ISA(Instruction Set Architecture)都有自己的指令集，我们在计组课程里学习的RISC-V32指令集就是一个例子。不过相比于这种比较精简的指令集，CISC(Complex Instruction Set Computer)则是一种指令集更加复杂的架构，比如用于第一代Intel处理器的x86指令集以及后续为提高浮点数学处理能力而设计的x87指令集。这两个指令集统称为x86指令集，这也是x86-64这个架构名字的由来。

不过即便对于64位的系统，一个寄存器的大小也是64bit，如果想要同时计算多个数据就显得不足了。为了解决这个需求，Intel一如了单指令多数据(Single Instruction, Multiple Data)的指令集，也就是SIMD指令集。SIMD指令集可以同时对多个数据进行操作，这样可以提高程序的并行度，提高程序的性能。SIMD指令集经过历史的演化完善，所包括的子指令集也逐渐丰富，主要有以下几种：



- MMX: 64-bit SIMD 整数指令集。主要用于处理音频、视频等多媒体数据
- SSE: 128-bit SIMD 指令集。主要用于处理整型数据以及双精度浮点数的计算，并且提供了各种类型之间的转换指令
- AVX: 256-bit SIMD 指令集。主要用于处理浮点数的计算，提供了更多的寄存器和更多的指令
- AVX-512: 512-bit SIMD 指令集。主要用于处理高性能计算(HPC)、数据压缩以及深度学习的大规模数据计算

而对于每种 SIMD 指令集，他们使用的寄存器也不同，比如 SSE 使用的是 XMM 寄存器，AVX 使用的是 YMM 寄存器，AVX-512 使用的是 ZMM 寄存器。对应到 C 语言中，我们可以使用 `_m128`、`_m256`、`_m512` 等类型来表示这些寄存器。

在这里，我们主要使用 SSE 对应的 `_m128` 类型来进行 SIMD 计算，不过我们实际使用的是 `_m128i` 和 `_m128d` 类型，分别用来表示整型和双精度浮点数。他们在头文件 `emmintrin.h` 中定义如下：

```
typedef union __declspec(intrin_type) __CRT_ALIGN(16) __m128i {
    __int8      m128i_i8[16];
    __int16     m128i_i16[8];
    __int32     m128i_i32[4];
    __int64     m128i_i64[2];
    unsigned __int8 m128i_u8[16];
    unsigned __int16 m128i_u16[8];
}
```

```

    unsigned __int32    m128i_u32[4];
    unsigned __int64    m128i_u64[2];
} __m128i;

typedef struct __declspec(intrin_type) __CRT_ALIGN(16) __m128d {
    double             m128d_f64[2];
} __m128d;

```

从定义的结构体可以看出，`__m128i` 和 `__m128d` 在 `__CRT_ALIGN(16)` 下，开启了 16 字节对齐，这样可以保证这两个类型的变量在内存中的地址起始位与 RAM 单元整对齐，更便于 CPU 的读取。

在 Intel® Intrinsics Guide 中，我找到了最关键的并行乘法指令 `_mm_mul_ep32`，这个指令可以将两个 `__m128i` 类型的寄存器中的数据扩展为 signed 64-bit 向量进行并行乘法，然后将结果存储到一个新的寄存器中。其过程描述如下：

```

FOR j := 0 to 1
    i := j*64
    dst[i+63:i] := SignExtend64(a[i+31:i]) * SignExtend64(b[i+31:i])
ENDFOR

```

不过这个指令只能一次处理两个 32-bit 整数，所以在计算 offset 为 0 和 offset 为 1 的两个 32-bit 整数乘积后，我们需要右移 32 位，然后再次调用 `_mm_mul_ep32` 指令，这样就可以实现一次计算 4 个 32-bit 整数的乘积。结合之前 O0 汇编的观察，由于很多循环中使用的局部变量都会存到栈上，读取时效率很低，所以我们可以使用 `register` 关键字来声明这些变量，这种技巧可以提示编译器将这些变量存储到寄存器中，可以减少 DRAM 访问的开销。

```

void dotproduct_simdi(register i32 *a, register i32 *b,
                      register scalar_i *result, register size_t size) {

    register __m128i lo = _mm_setzero_si128();
    register __m128i hi = _mm_setzero_si128();

    register __m128i* pa = (__m128i*)a;
    register __m128i* pb = (__m128i*)b;

    // Process 4 elements at a time (SSE registers are 128-bit = 4 x int32)
    register size_t i = 0;
    for (; i + 3 <= size; i += 4) {
        // Load 4 int32 values from each array
        register __m128i va = _mm_lddqu_si128(pa + (i >> 2ULL));
        register __m128i vb = _mm_lddqu_si128(pb + (i >> 2ULL));

        // Multiply with extended precision: 32-bit * 32-bit -> 64-bit
        // _mm_mul_ep32 only multiplies odd indices

        // Multiply odd indices (1, 3)
        register __m128i mul_odd = _mm_mul_ep32(va, vb);

        // Shift right to get even indices (0, 2) in position
        va = _mm_srl_epi64(va, 32);
        vb = _mm_srl_epi64(vb, 32);

        // Multiply even indices (0, 2)
        register __m128i mul_even = _mm_mul_ep32(va, vb);

        lo = _mm_add_epi64(lo, mul_odd);
        hi = _mm_add_epi64(hi, mul_even);
    }
}

```

```

    }

    lo = _mm_add_epi64(lo, hi);

    // Extract results from the __m128i
    *result += _mm_extract_epi64(lo, 0) + _mm_extract_epi64(lo, 1);

    // Handle remaining elements
    for (; i < size; i++) {
        *result += (int64_t)a[i] * b[i];
    }
}

```

当然，这种 SIMD 指令只能使用在 x86 平台上，如果我们想添加对 ARM 平台的支持，我们可以使用 NEON 指令集，这个指令集和 SSE 指令集类似，也是一种 SIMD 指令集，我们可以增加条件编译来让这个函数支持多种平台：

```

void dotproduct_simdi(register int32 *a, register int32 *b,
                      register scalar_i *result, register size_t size) {

#ifdef SSE_
    // SSE implementation
#elif defined(NEON_)
    int64x2_t sum_0 = vdupq_n_s64(0);
    int64x2_t sum_1 = vdupq_n_s64(0);

    // Process 4 elements at a time (2 elements per loop iteration per vector)
    size_t i = 0;
    for (; i + 3 < length; i += 4) {
        // Load 2 int32 values into each vector
        int32x2_t va_0 = vld1_s32(&a[i]);
        int32x2_t vb_0 = vld1_s32(&b[i]);
        int32x2_t va_1 = vld1_s32(&a[i+2]);
        int32x2_t vb_1 = vld1_s32(&b[i+2]);

        // Multiply with extended precision: 32-bit * 32-bit -> 64-bit
        int64x2_t prod_0 = vmull_s32(va_0, vb_0);
        int64x2_t prod_1 = vmull_s32(va_1, vb_1);

        // Accumulate the products
        sum_0 = vaddq_s64(sum_0, prod_0);
        sum_1 = vaddq_s64(sum_1, prod_1);
    }

    // Combine the partial sums
    int64x2_t sum = vaddq_s64(sum_0, sum_1);

    // Extract and sum the two 64-bit values
    *result = vgetq_lane_s64(sum, 0) + vgetq_lane_s64(sum, 1);

    for (; i < length; i++) {
        *result += (int64_t)a[i] * b[i];
    }
#else
    printf("No SIMD support available. Falling back to plain dot product.\n");
    // Fallback to plain dot product if SIMD is not available
    for (size_t i = 0; i < size; i++) {
        *result += a[i] * b[i];
    }
#endif
}

```

```

}
#endif
}

(~code/CPP/project2)
[21:47:03 on main * ]-> make
gcc -c -Wall -march=native -fopenmp -D_EPA src/dotproduct.c -o build/dotproduct.o -I./inc
gcc -fopenmp -o build/dot ./build/dotproduct.o
(~code/CPP/project2)
[21:47:08 on main * ]-> make run SIZE=100000000 TYPE=double
Running the program...
./build/dot 100000000 double 4
    Plain ===
0.718468900
Dot product result: 24987831.547229
    Unroll ===
0.653222900
Dot product result: 24987831.547229
    SIMD ===
0.158735700 O0 优化 SIMD
Dot product result: 24987831.547229
Program finished running
Output file: ./build/dot
Output file location: ./build/dot
Output file size: 28K
(~code/CPP/project2)
[21:47:19 on main * ]-> make O2
gcc -c -Wall -march=native -fopenmp -D_EPA -O2 -c ./src/dotproduct.c -o ./build/dotproduct.o
gcc -fopenmp -o ./build/dot ./build/dotproduct.o
Optimized with O2
(~code/CPP/project2)
[21:47:33 on main * ]-> make run SIZE=100000000 TYPE=double
Running the program...
./build/dot 100000000 double 4
    Plain ===
0.444248200 O2 优化
Dot product result: 24989757.525325
    Unroll ===
0.190800000
Dot product result: 24989757.525325
    SIMD ===
0.059433100
Dot product result: 24989757.525325
Program finished running
Output file: ./build/dot
Output file location: ./build/dot
Output file size: 24K

```

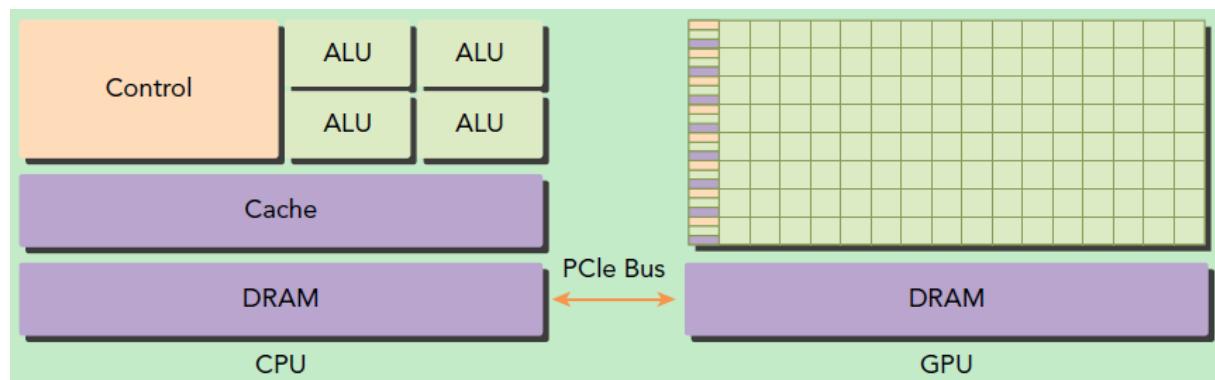
我们只测试了 x86 平台的 SIMD 指令优化函数，其中注意到 Data Type 为 `double` 时，SIMD 计算的结果虽然和朴素实现不太一致，但计算结果的误差在可接受范围内。计算时间上，可以看到 SIMD 指令集的计算时间比 O2 优化的程序还要快，说明 SIMD 的优化策略还是很不错的。实际上 O3 优化的程序也是通过组合 SIMD 指令集来实现的，因此这样的结果也在意料之中。

§5.3. CUDA 并行优化

CUDA 是 NVIDIA 公司推出的一种并行计算平台和编程模型，它可以让程序员使用 C 语言来编写 GPU 的程序，然后通过 NVIDIA 的 CUDA 编译器来编译成 GPU 的指令集，从而实现 GPU 的并行计算。

CUDA 的并行计算模型是 SIMT(Single Instruction, Multiple Threads)，它将一个线程块(Block)中的线程划分为若干个线程束(Warp)，每个线程束中的线程会同时执行一条指令，这样可以提高 GPU 的并行度，提高程序的性能。

GPU 在 CUDA 计算模型中并非一个独立的计算设备，而是一个协处理器，它需要和 CPU 协同工作，CPU 负责程序的控制流，GPU 负责程序的数据流。CPU 所在位置称为 Host，GPU 所在位置称为 Device，Host 和 Device 之间的数据传输是通过 PCIe 总线来实现的，如下图所示



GPU 中比 CPU 多了大量的计算核心，特别适合于大规模的并行计算，比如矩阵乘法、图像处理等。而 CUDA 的语法十分接近 C 语言，因此我们可以将一个.cu 文件编译成 GPU 的指令集，然后在 Host 上调用这个指令集来实现 GPU 的点乘并行计算。

Host 和 Device 的协作基本上遵循这个流程

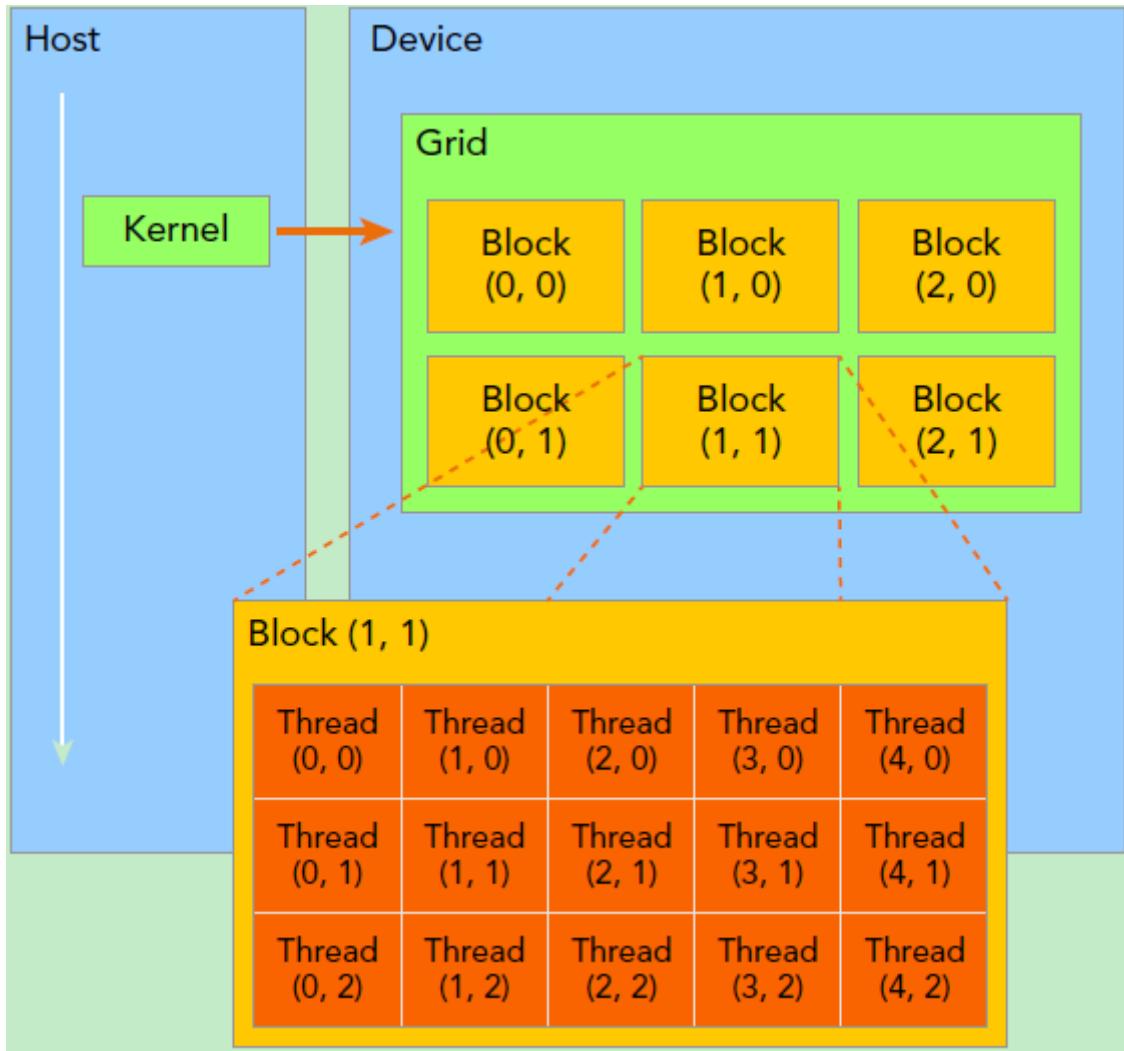
1. 在 Host 上分配内存空间，并进行初始化
2. 分配 Device 上的内存空间，并将 Host 上的数据拷贝到 Device 上
3. 调用 CUDA 的 Kernel 函数来执行 GPU 的计算
4. 将 Device 上的数据拷贝到 Host 上
5. 释放 Device 和 Host 上的内存空间

上面流程中最重要的一个过程是调用 CUDA 的核函数来执行并行计算，kernel 是 CUDA 中一个重要的概念，kernel 是在 device 上线程中并行执行的函数，核函数用 global 符号声明，在调用时需要用<<<grid, block>>>来指定 kernel 要执行的线程数量，在 CUDA 中，每一个线程都要执行核函数，并且每个线程会分配一个唯一的线程号 thread ID，这个 ID 值可以通过核函数的内置变量 threadIdx 来获得。

CUDA 中区分 Host 和 Device 的函数是通过 CUDA 的函数限定词实现的，主要有以下几种：

- `_global_`: 在 Device 上执行的函数，可以被 Host 调用，返回值为 void
- `_device_`: 在 Device 上执行的函数，只能被 Device 调用，返回值可以是任意类型
- `_host_`: 在 Host 上执行的函数，仅在 Host 上执行，返回值可以是任意类型，一般省略不写

CUDA 中最重要的概念是 kernel 的线程层次结构。一个 kernel 所启动的线程群为一个网格 (grid)，grid 中的线程被划分为若干个线程块(block)，一个线程块中包含若干个线程(thread)。同一个网格上的线程共享同一块共享内存，同一个线程块上的线程共享同一块共享内存，同一个线程块上的线程可以通过共享内存进行通信。



一个线程需要两个参数来确定自己的位置，一个是 blockIdx，表示线程块的索引，另一个是 threadIdx，表示线程在线程块中的索引。它们都是 dim3 类型，这个类型是一个三维的向量，可以用来表示线程的三维位置。比如图中的 Thread (1, 1) 就有 blockIdx = (1, 1, -) 和 threadIdx = (1, 1, -)。通过这两个参数，我们可以确定一个线程在整个网格中的位置，然后通过这个位置来计算线程要处理的数据。

理论了这么多，我们来尝试编写一个 CUDA 的核函数来实现 GPU 的点乘并行计算

```
__global__ void dotProd_kernel_i32(int *a, int *b, long long *c, unsigned long long
size) {
    __shared__ int cache[256];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    int tmp = 0;
    while (tid < size) {
        tmp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = tmp;
}

__syncthreads();
```

```

// do reduction in shared mem
int i = blockDim.x >> 1;
while (i != 0) {
    if (cacheIndex < i) {
        cache[cacheIndex] += cache[cacheIndex + i];
    }
    __syncthreads();
    i >>= 1;
}

if (cacheIndex == 0) {
    c[blockIdx.x] = cache[0];
}
}

__global__ void dotProd_kernel_f64(double *a, double *b, double *c, unsigned long
long n) {
    // similar to the int version
    // ...
}

```

我们在这里分配了定义了共享内存为`__shared__`的缓存向量大小为 256，他们用来存储每个线程块的计算结果，然后在核函数中，我们使用了`__syncthreads()`函数来同步线程块中的线程，最后将缓存向量中的结果累加到结果中。

由于 kernel 函数只能在 Device 上执行，我们如果想调用它，需要编写相应的 Host 函数来封装这个 kernel。在最开始，我们定义每个 Block 内的线程数为 256，然后依据向量 Size 大小计算出比较合适的 Block 数。接着，我们调用 `cudaMalloc` 函数来在 Device 上分配内存，然后调用 `cudaMemcpy` 函数将 Host 上的数据拷贝到 Device 上。调用完核函数后，我们再次调用 `cudaMemcpy` 函数将 Device 上的数据拷贝到 Host 上，同时处理剩余的元素。最后需要使用 `cudaFree` 函数来释放 Device 上的内存。

```

int dotproduct_cuda_i32(int *a, int *b, long long *result, unsigned long long size) {
    int *d_a, *d_b;
    long long *d_c, *partial_results;

    int threadsPerBlock = 256;
    int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;
    if (blocksPerGrid > 65535) blocksPerGrid = 65535;

    cudaMalloc((void**)&d_a, size * sizeof(int));
    cudaMalloc((void**)&d_b, size * sizeof(int));
    cudaMalloc((void**)&d_c, blocksPerGrid * sizeof(long long));

    cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    partial_results = (long long*)malloc(blocksPerGrid * sizeof(long long));

    dotProd_kernel_i32<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, size);

    cudaDeviceSynchronize();
    cudaMemcpy(partial_results, d_c, blocksPerGrid * sizeof(long long),
    cudaMemcpyDeviceToHost);

    // reduce the partial results on the host
}

```

```

*result = 0;
for (int i = 0; i < blocksPerGrid; i++) {
    *result += partial_results[i];
}

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(partial_results);

return 0;
}

int dotproduct_cuda_f64(double *a, double *b, double *result, unsigned long long
size) {
    // similar to the int version
    // ...
}

```

§5.3.1. 性能对比

在这里我们使用 nvprof 工具来分析 CUDA 程序的性能表现，nvprof 是 NVIDIA 公司提供的一个性能分析工具，它可以用来分析 CUDA 程序的 GPU 使用情况，包括 GPU 的内存使用情况、核函数的执行时间、核函数的调用次数等，以及各种 API 的运行时间。

测试环境依然和实验 2 一样，我们在这里使用大小为 100000000 的向量来测试 CUDA 程序的性能表现，分别测试了 int32 和 float32 两种数据类型的点乘程序。

```

stu_12310823@nb1516:~$ cd proj2-cpp/
stu_12310823@nb1516:~/proj2-cpp$ nvprof ./build/dot 100000000 int
    == Plain ==
0.282278192
Dot product result: 25035180
    == Unroll ==
0.235637909
Dot product result: 25035180
    == SIMD ==
0.162898889
Dot product result: 25035180
    == CUDA Parallel ==
==1185708== NVPROF is profiling process 1185708, command: ./build/dot 100000000 int
0.688484673
Dot product result: 25035180
==1185708== Profiling application: ./build/dot 100000000 int
==1185708== Profiling result:
      Type  Time(%)     Time    Calls      Avg      Min      Max  Name
GPU activities:  99.00% 127.41ms      2  63.706ms  61.606ms  65.805ms [CUDA memcpy HtoD]
               0.97% 1.2432ms      1  1.2432ms  1.2432ms  1.2432ms dotProd_kernel_i32(int*, int*, __int64*, __int64)
               0.03% 41.791us      1  41.791us  41.791us  41.791us [CUDA memcpy DtoH]
API calls:    78.31% 559.40ms      3  186.47ms  3.9141ms  542.69ms cudaMalloc
               17.96% 128.29ms      3  42.764ms  518.37us  66.010ms cudaMemcpy
               1.79% 12.770ms    798  16.002us   317ns  3.5574ms cuDeviceGetAttribute
               1.69% 12.049ms      3  4.0162ms  222.50us  10.955ms cudaFree
               0.17% 1.2429ms      1  1.2429ms  1.2429ms  1.2429ms cudaDeviceSynchronize
               0.06% 399.35us      1  399.35us  399.35us  399.35us cuLibraryLoadData
               0.01% 69.804us      7  9.9720us  7.5900us  20.027us cuDeviceGetName
               0.01% 66.819us      1  66.819us  66.819us  66.819us cudaLaunchKernel
               0.00% 24.657us      7  3.5220us  1.5830us  9.8210us cuDeviceGetPCIBusId
               0.00% 6.4120us     14  458ns   363ns  1.2030us cuDeviceGet
               0.00% 4.4670us      7  638ns   517ns  1.0000us cuDeviceTotalMem
               0.00% 3.2170us      7  459ns   403ns   584ns cuDeviceGetUuid
               0.00% 2.7380us      3  912ns   517ns  1.6970us cuDeviceGetCount
               0.00% 847ns         1  847ns   847ns   847ns cuModuleGetLoadingMode

```

```
stu_12310823@nb1516:~/proj2-cpp$ nvprof ./build/dot 100000000 double
    == Plain ==
0.441778193
Dot product result: 25016279.015934
    == Unroll ==
0.512793131
Dot product result: 25016279.015934
    == SIMD ==
0.197730524
Dot product result: 25016279.015940
    == CUDA Parallel ==
==1186530== NVPROF is profiling process 1186530, command: ./build/dot 100000000 double
0.729143474
Dot product result: 25016279.015934
==1186530== Profiling application: ./build/dot 100000000 double
==1186530== Profiling result:
      Type  Time(%)     Time    Calls      Avg      Min      Max   Name
GPU activities:  98.97%  242.17ms      2  121.09ms  119.87ms  122.30ms  [CUDA memcpy HtoD]
               1.01%  2.4668ms      1  2.4668ms  2.4668ms  2.4668ms  dotProd_kernel_f64(double*, double*, double*, __int64)
               0.02%  42.239us      1  42.239us  42.239us  42.239us  [CUDA memcpy DtoH]
API calls:    63.57%  455.37ms      3  151.79ms  405.88us  453.46ms  cudaMalloc
               33.93%  243.03ms      3  81.010ms  557.23us  122.44ms  cudaMemcpy
               1.42%  10.205ms      3  3.4016ms  186.06us  8.8719ms  cudaFree
               0.66%  4.7301ms    798  5.9270us  285ns  323.60us  cuDeviceGetAttribute
               0.35%  2.4779ms      1  2.4779ms  2.4779ms  2.4779ms  cuDeviceSynchronize
               0.04%  308.05us      1  308.05us  308.05us  308.05us  cuLibraryLoadData
               0.01%  64.750us      7  9.2500us  6.2680us  19.467us  cuDeviceGetName
               0.01%  55.332us      1  55.332us  55.332us  55.332us  cudaLaunchKernel
               0.00%  23.064us      7  3.2940us  1.9400us  8.0950us  cuDeviceGetPCIBusId
               0.00%  5.7910us     14  413ns   290ns  997ns   cuDeviceGet
               0.00%  3.8670us      7  552ns   437ns  920ns   cuDeviceTotalMem
               0.00%  3.0110us      7  430ns   355ns  680ns   cuDeviceGetUuid
               0.00%  2.4710us      3  823ns   432ns  1.5220us  cuDeviceGetCount
               0.00%  767ns        1  767ns   767ns  767ns   cuModuleGetLoadingMode
```

可以看到，相比与上面三种优化方法，CUDA 程序的总计算时间是最久的。不过如果细看 nvprof 的 report，可以注意到真正的核函数执行时间是很短的。数据类型为 int32 的程序，核函数执行时间只有 1.2ms，而数据类型为 float32 的程序，核函数执行时间只有 2.5ms。而最大的时间开销来自于数据拷贝，这个结果并不奇怪，因为 GPU 和 CPU 之间的数据传输是通过 PCIe 总线进行 I/O 的，传输像 100000000 这么大的数据量，自然会消耗很多时间。

§5.4. 总结

这个实验中，我们尝试了三种优化方法来提高 C 程序的性能，分别是循环展开、SIMD 指令集和 CUDA 并行优化。可以看出，循环展开在一定程度上减少了点乘计算的时间，但效果并不理想。CUDA 并行优化虽然在核函数执行时间占优，但由于数据拷贝的时间开销甚至超过了核函数执行时间，所以总体性能并不理想。而 SIMD 指令集则是最有效的优化方法，通过使用 SIMD 指令集，我们真正做到了在不引入其他代价的情况下通过 CPU 的并行计算来提高程序的性能，并且在数据类型为 double 的情况下，SIMD 指令集的性能甚至超过了 O2 优化的程序，这个结果是比较令人满意的。

§6. Extra

关于这部分其实是我的私货，不过鉴于这门课最后也会接触一些 Rust，我们可以通过简单的性能测试来学习以下 Rust 的特性。Rust 是一个对标 C++ 的系统级编程语言，它以更加安全的指针管理和复杂借用规则著称，同时也支持高性能的底层计算。为了更好地了解 Rust 的性能表现，我尝试用 Rust 实现了一个类似的计算点乘的程序，并且使用了 Rayon 库来实现多线程并行计算。

首先，这个朴素算法和 SIMD 优化和 C 语言实现基本一致，Rust 的 `std::arch` 库提供了 SIMD 指令集的支持，我们完全可以像 C 语言一样使用 SIMD 指令集来实现并行计算。

```
pub fn dotproduct<T>(a: &[T], b: &[T], c: usize) -> T
where
    T: std::ops::Mul<Output = T> + std::ops::AddAssign + Copy + Default
{
    let mut result = T::default();
    for i in 0..c {
```

```

        result += a[i] * b[i];
    }
    result
}

#[cfg(target_arch = "x86_64")]
pub fn dotproduct_simd(a: &[f32], b: &[f32], c: usize) -> f32 {
    use std::arch::x86_64::*;
    let mut result = unsafe {__mm_setzero_ps()};
    let mut i = 0;
    unsafe {while i + 4 <= c {
        let a_vec = __mm_loadu_ps(&a[i]);
        let b_vec = __mm_loadu_ps(&b[i]);
        result = __mm_add_ps(result, __mm_mul_ps(a_vec, b_vec));
        i += 4;
    }}
    let mut sum = [0.0; 4];
    unsafe {__mm_storeu_ps(&mut sum[0], result)};
    sum.iter().sum::<f32>() + dotproduct(&a[i..], &b[i..], c - i)
}

#[cfg(target_arch = "aarch64")]
pub fn dotproduct_simd(a: &[f32], b: &[f32], c: usize) -> f32 {
    use std::arch::aarch64::*;
    // similar to c version
    // ...
}

```

为了方便计算各种类型，我在这里使用了泛型来定义点乘函数。由于 Rust 的泛型是通过编译时的类型推导来实现的，在运行时几乎零成本，所以不需要考虑泛型会带来额外的性能开销。

在使用 Rayon 库时，主要是用了 `par_chunk` 和 `sum` 函数。`par_chunk` 函数是 Rayon 库提供的一个基础 api，它可以将一个向量分成若干个 chunk，然后在每个 chunk 上启动一个线程来执行迭代器的操作，我们只需要写一个闭包来定义迭代器的操作，然后使用 `sum` 函数来将迭代器返回的结果 reduce 到一个标量形式中。

```

pub fn dotproduct_parallel<T>(a: &[T], b: &[T], size: usize, num_threads: usize) -> T
where
    T: Send + Sync + Copy + Add<Output = T> + std::iter::Sum + Mul<Output = T> +
Default,
{
    let chunk_size = (size + num_threads - 1) / num_threads;
    a.par_chunks(chunk_size)
        .zip(b.par_chunks(chunk_size))
        .map(|(chunk1, chunk2)| {
            chunk1.iter()
                .zip(chunk2.iter())
                .map(|(&a, &b)| a * b)
                .sum()
        })
        .sum::<T>()
}

```

²<https://zhuanlan.zhihu.com/p/402478044>

然后下面就到了性能测试的部分。我在这里²找到了 Rust 最好用的 benchmark 库是 Criterion，相比于官方的 benchmark 库，Criterion 不需要将编译链升级为 rust nightly 版本，并且他能结合 gnuplot 生成更加直观的性能图表。使用方法也比较简单，首先在 Cargo.toml 中加入如下内容

```
[dev-dependencies]
criterion = { version = "0.5.1", features = ["html_reports"] }

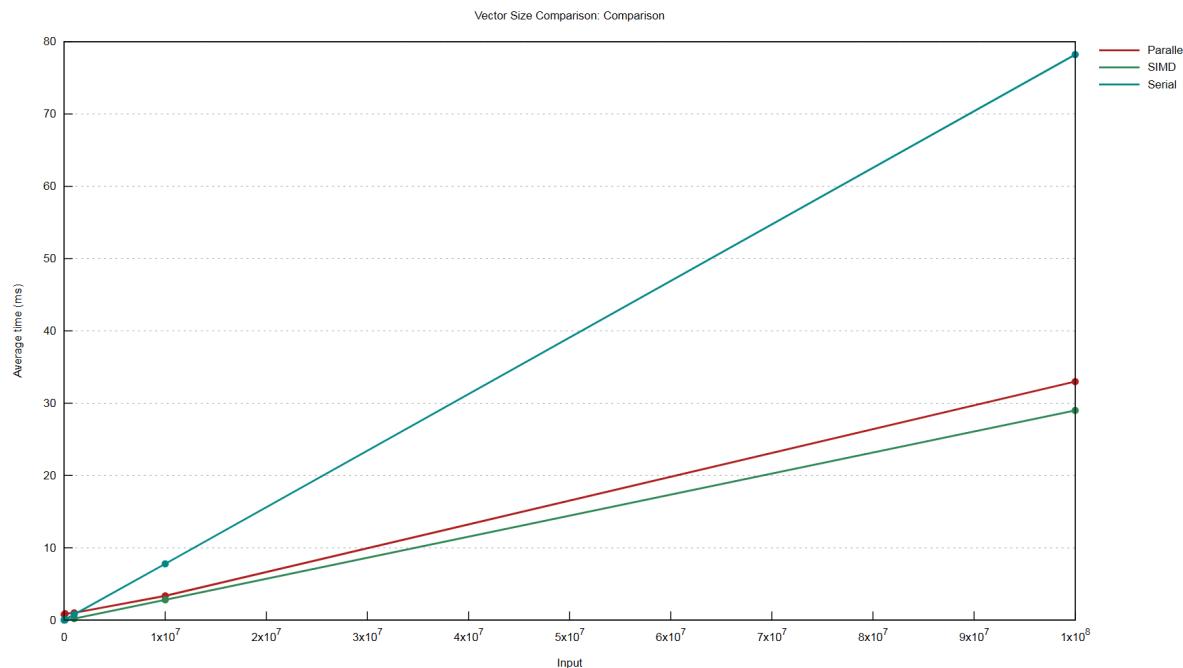
[[bench]]
name = "dotprod"
harness = false
```

其中选择 `html_reports` 特性是为了生成更加直观且美观的性能报告 html 文件。`[[bench]]` 部分是为了告诉 Criterion 我们要在 `benches` 目录下写一个 `dotprod.rs` 文件来实现性能测试。

我在 benchmark 文件中定义了两个测试，一个是将朴素算法、SIMD 优化和 Rayon 并行计算的性能进行对比，数据大小为 10000 到 100000000(在我的笔记本上最多开这么大)；另外一个测试是单独测试 Rayon 不同线程数并行计算的性能，数据大小为 100000000，线程数为 1 到 16。两个测试由于时间原因均只选择了 float32 类型的数据进行测试。

最后，我们可以通过 `cargo bench` 命令来运行 benchmark 测试，然后在 `target/criterion/report/index.html` 中查看性能报告如下

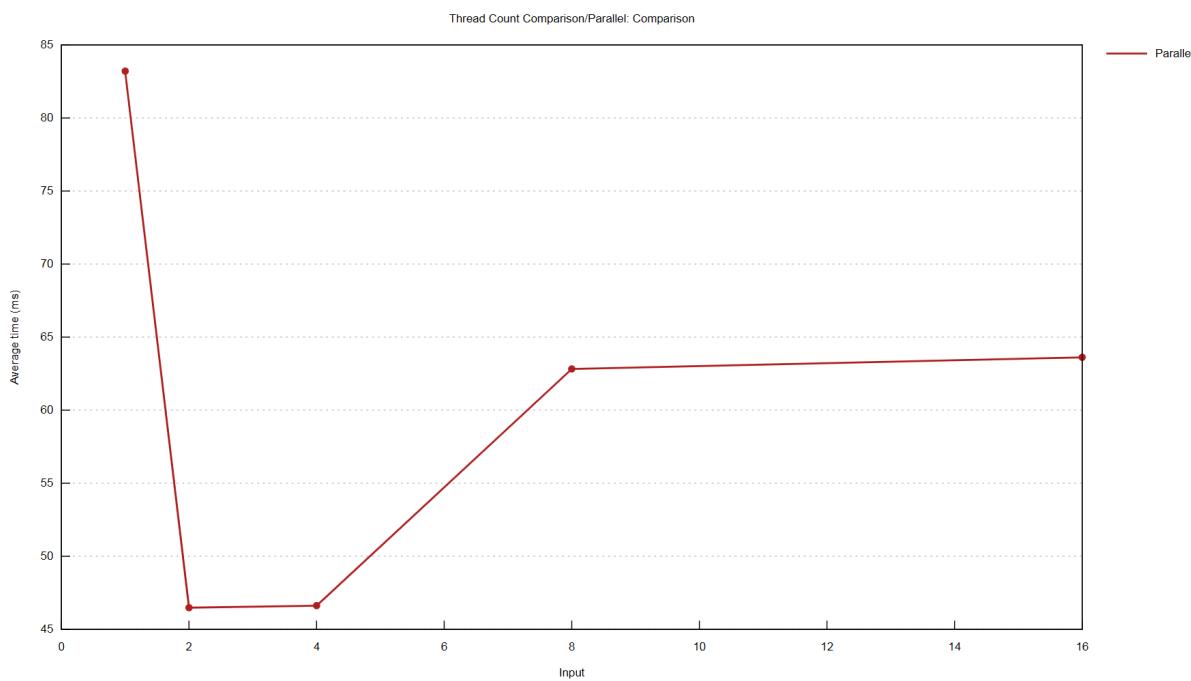
Line Chart



This chart shows the mean measured time for each function as the input (or the size of the input) increases.

在数据量比较小的情况下(低于 10^7)，Parallel 计算时间最不占优，但随着数据量的增加，Parallel 的计算时间逐渐减少，最终在 10^8 的数据量下，Parallel 的计算时间和 SIMD 优化的计算时间几乎一致，考虑到我们对并行计算函数几乎没有做任何优化，可以看出 Rayon 库的并行计算性能是非常优秀的。

Line Chart



This chart shows the mean measured time for each function as the input (or the size of the input) increases.

接着是 Rayon 不同线程数的性能测试，可以看到随着线程数的从 1 增加到 2，计算时间几乎减少了一半，而随着线程数的增加，计算时间并没有继续减少，反而到线程数为 8 的时候就发生了反弹，这可能是笔记本上的 CPU 核心数有限，一些线程并不是 1:1 映射到 CPU 核心上，导致线程切换的开销增加，从而影响了计算性能。

§7. 结论与反思

在这四个实验里，我们较为综合的比较了 Java 和 C 两种语言对点乘计算的实现和性能表现。首先，Java 的 JIT 和 codeCache 机制处理多循环的情况时，表现十分优秀，在依赖 JVM 虚拟环境运行和 GC 回收机制的影响下中依然能够超过 C 语言的性能表现，这个结果改变了我对 Java 语言性能的一些看法，并且对现代编程语言的优化策略有了更深的理解。

接下来是多线程的优化，在进行多次试验后可以发现 Java 的多线程性能并不理想，这主要是因为 Java 的线程分配时间开销大，线程启动不一致，而且线程 CPU utilization 不高，这些都是 Java 多线程性能提升不明显的原因，而 C 语言的 openmp 用法比较简单，在实际执行时也起到了很好的效果，这也说明了 C 程序的多线程实现相对于 Java 程序更加高效。

Intel VTune Profiler 为我们提供了更加细致入微的性能分析，通过这个工具，Java 的具体执行过程和性能瓶颈一目了然，而 C 程序的调用栈和性能瓶颈也可以清晰地展现出来，我们在这里分析了 Java 程序多线程的启动过程和性能瓶颈的原因，以及 C 程序的 CPI 值和循环展开的优化方法，

有了上面的经验，我们采取了一些优化方法，比如循环展开、SIMD 指令集和 CUDA 并行优化，有些优化方法在一定程度上提高了 C 程序的性能，但效果并不理想；而 SIMD 指令集则是最有效的优化方法，它确实能够与 O2 优化的程序相媲美，向量化的思想也是很多高性能计算库的核心。

综合来看，通过一系列实验，我认为 Java 和 C 在点乘计算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。

最后，我们尝试了 Rust 语言的性能测试，通过 Rust 的泛型和 Rayon 库的并行计算，我们可以看到 Rust 的性能表现也确实名不虚传，Rayon 库简洁的 API 和高效的并行计算能力，可以从中窥见 Rust 在系统级编程和高性能计算方面有着很好的表现。而且在这个过程中，接触了一些新的 Rust benchmark，Criterion，这个工具可以直接集成到项目中，并且允许我们通过直观的图表来查看性能报告，这对于我们进行性能测试和优化是非常有帮助的。

不过还有一些问题值得发掘，比如我们在点乘中同时引入了整型和浮点数计算，那么不同类型的数据计算在底层是怎么优化的？它们各自有哪些瓶颈？以及 Java 程序中只是分析了多线程性能下降的原因，那么如何提高 Java 程序的多线程性能？Rust 作为借鉴了函数式语言的现代语言是怎么解决底层计算的优化来保证其与原生 C 语言几乎相同并且更加优秀的性能的？它与 clang-c++ 都使用 llvm 作为后端，是否在 LIR 生成上有类似之处或者有所改进？这些问题都是值得进一步研究的。

我们在平时编程中几乎不会考虑一个加法操作这样简单的计算会产生多少性能开销。但是在这个实验中，我们要结合数据的类型和规模，实现的语言特性，深入到汇编层面、运行层面甚至硬件微结构层面来分析一个操作的性能瓶颈，去看我们的代码是怎么被编译器优化的，是怎么被 CPU 执行的，在程序执行中又会有那些影响性能的因素。我很欣慰我在操作系统课上学到的一些概念和原理在这个实验中得到了更加深邃的理解，并且从一个优化的角度去应用这些知识，这也是这个 project 最大的收获。

Bibliography

- [1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating System Concepts 10th.* 2018.

Appendix

A GCC 优化等级对应选项细节	43.
AA O0	43.
AB O 和-O1	43.
AC O2	44.
AD O3	46.

§A GCC 优化等级对应选项细节

§AA O0

无

§AB O 和-O1

打开的优化选项：

- -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。
- -fmerge-constants: 尝试横跨编译单元合并同样的常量(string constants and floating point constants)
- -fthread-jumps: 如果某个跳转分支的目的地存在另一个条件比较，而且该条件比较包含在前一个比较语句之内，那么执行本项优化。根据条件是 true 或者 false，前面那条分支重定向到第二条分支的目的地或者紧跟在第二条分支后面。
- -floop-optimize: 执行循环优化，将常量表达式从循环中移除，简化判断循环的条件，并且 optionally do strength-reduction，或者将循环打开等。在大型复杂的循环中，这种优化比较显著。
- -fif-conversion: 尝试将条件跳转转换为等价的无分支型式。优化实现方式包括条件移动，min, max，设置标志，以及 abs 指令，以及一些算术技巧等。
- -fif-conversion2 基本意义相同。
- -fdelayed-branch: 这种技术试图根据指令周期时间重新安排指令。它还试图把尽可能多的指令移动到条件分支前，以便最充分的利用处理器的治理缓存。
- -fguess-branch-probability: 当没有可用的 profiling feedback 或 builtin_expect 时，编译器采用随机模式猜测分支被执行的可能性，并移动对应汇编代码的位置，这有可能导致不同的编译器会编译出迥然不同的目标代码。
- -fcprop-registers: 因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性(两个段要求使用相同的寄存器)并且删除不必要的寄存器复制操作。

§AC O2

在 O1 的基础上，增加了以下优化选项：

- -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。
- -foptimize-sibling-calls: 优化相关的以及末尾递归的调用。通常，递归的函数调用可以被展开为一系列一般的指令，而不是使用分支。这样处理器的指令缓存能够加载展开的指令并且处理他们，和指令保持为需要分支操作的单独函数调用相比，这样更快。
- -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。
- -fcse-follow-jumps: 在公用子表达式消元时，当目标跳转不会被其他路径可达，则扫描整个的跳转表达式。例如，当公用子表达式消元时遇到 if...else... 语句时，当条件为 false 时，那么公用子表达式消元会跟随着跳转。
- -fcse-skip-blocks: 与 -fcse-follow-jumps 类似，不同的是，根据特定条件，跟随着 cse 跳转的会是整个的 blocks
- -frerun-cse-after-loop: 在循环优化完成后，重新进行公用子表达式消元操作。

- `-frerun-loop-opt`: 两次运行循环优化 1. `-fgcse`: 执行全局公用子表达式消除 pass。这个 pass 还执行全局常量和 copy propagation。这些优化操作试图分析生成的汇编语言代码并且结合通用片段，消除冗余的代码段。如果代码使用计算性的 goto, gcc 指令推荐使用`-fno-gcse` 选项。
- `-fgcse-lm`: 全局公用子表达式消除将试图移动那些仅仅被自身存储 kill 的装载操作的位置。这将允许将循环内的 load/store 操作序列中的 load 转移到循环的外面（只需要装载一次），而在循环内改变成 copy/store 序列。在选中`-fgcse` 后，默认打开。
- `-fgcse-sm`: 当一个存储操作 pass 在一个全局公用子表达式消除的后面，这个 pass 将试图将 store 操作转移到循环外面去。如果与`-fgcse-lm` 配合使用，那么 load/store 操作将会转变为在循环前 load，在循环后 store，从而提高运行效率，减少不必要的操作。
- `-fgcse-las`: 全局公用子表达式消除 pass 将消除在 store 后面的不必要的 load 操作，这些 load 与 store 通常是同一块存储单元（全部或局部）
- `-fdelete-null-pointer-checks`: 通过对全局数据流的分析，识别并排出无用的对空指针的检查。编译器假设间接引用空指针将停止程序。如果在间接引用之后检查指针，它就不可能为空。
- `-fexpensive-optimizations`: 进行一些从编译的角度来说代价高昂的优化（这种优化据说对于程序执行未必有很大的好处，甚至有可能降低执行效率，具体不是很清楚）
- `-fregmove`: 编译器试图重新分配 move 指令或者其他类似操作数等简单指令的寄存器数目，以便最大化的捆绑寄存器的数目。这种优化尤其对双操作数指令的机器帮助较大。
- `-fschedule-insns`: 编译器尝试重新排列指令，用以消除由于等待未准备好的数据而产生的延迟。这种优化将对慢浮点运算的机器以及需要 load memory 的指令的执行有所帮助，因为此时允许其他指令执行，直到 load memory 的指令完成，或浮点运算的指令再次需要 cpu。1
- `-fschedule-insns2`: 与`-fschedule-insns` 相似。但是当寄存器分配完成后，会请求一个附加的指令计划 pass。这种优化对寄存器较小，并且 load memory 操作时间大于一个时钟周期的机器有非常好的效果。
- `-fsched-interblock`: 这种技术使编译器能够跨越指令块调度指令。这可以非常灵活地移动指令以便等待期间完成的工作最大化。
- `-fsched-spec-load`: 允许一些 load 指令进行一些投机性的动作。相同功能的还有`-fsched-spec-load-dangerous`，允许更多的 load 指令进行投机性操作。这两个选项在选中`-fschedule-insns` 时默认打开。
- `-fcaller-saves`: 通过存储和恢复 call 调用周围寄存器的方式，使被 call 调用的 value 可以被分配给寄存器，这种只会在看上去能产生更好的代码的时候才被使用。（如果调用多个函数，这样能够节省时间，因为只进行一次寄存器的保存和恢复操作，而不是在每个函数调用中都进行。）
- `-fpeephole2`: 允许计算机进行特定的观察孔优化（这个不晓得是什么意思），`-fpeephole` 与`-fpeephole2` 的差别在于不同的编译器采用不同的方式，有的采用`-fpeephole`，有的采用`-fpeephole2`，也有两种都采用的。
- `-freorder-blocks`: 在编译函数的时候重新安排基本的块，目的在于减少分支的个数，提高代码的局部性。
- `-freorder-functions`: 在编译函数的时候重新安排基本的块，目的在于减少分支的个数，提高代码的局部性。这种优化的实施依赖特定的已存在的信息：`.text.hot` 用于告知访问频率较高的函数，`.text.unlikely` 用于告知基本不被执行的函数。

- **-fstrict-aliasing:** 这种技术强制实行高级语言的严格变量规则。对于 c 和 c++ 程序来说，它确保不在数据类型之间共享变量。例如，整数变量不和单精度浮点变量使用相同的内存位置。
- **-funit-at-a-time:** 在代码生成前，先分析整个的汇编语言代码。这将使一些额外的优化得以执行，但是在编译器间需要消耗大量的内存。(有资料介绍说：这使编译器可以重新安排不消耗大量时间的代码以便优化指令缓存。)
- **-falign-functions:** 这个选项用于使函数对准内存中特定边界的开始位置。大多数处理器按照页面读取内存，并且确保全部函数代码位于单一内存页面内，就不需要叫化代码所需的页面。
- **-falign-jumps:** 对齐分支代码到 2 的 n 次方边界。在这种情况下，无需执行傀儡指令 (dummy operations)
- **-falign-loops:** 对齐循环到 2 的 n 次幂边界。期望可以对循环执行多次，用以补偿运行 dummy operations 所花费的时间。
- **-falign-labels:** 对齐分支到 2 的 n 次幂边界。这种选项容易使代码速度变慢，原因是需要插入一些 dummy operations 当分支抵达 usual flow of the code.
- **-fcrossjumping:** 这是对跨越跳转的转换代码处理，以便组合分散在程序各处的相同代码。这样可以减少代码的长度，但是也许不会对程序性能有直接影响

§AD O3

在 O2 的基础上，增加了以下优化选项：

- **-finline-functions:** 内联简单的函数到被调用函数中。由编译器启发式的决定哪些函数足够简单可以做这种内联优化。默认情况下，编译器限制内联的尺寸，3.4.6 中限制为 600 可以通过-finline-limit=n 改变这个长度。这种优化技术不为函数创建单独的汇编语言代码，而是把函数代码包含在调度程序的代码中。对于多次被调用的函数来说，为每次函数调用复制函数代码。虽然这样对于减少代码长度不利，但是通过最充分的利用指令缓存代码，而不是在每次函数调用时进行分支操作，可以提高性能。
- **-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。
- **-frename-registers:** 在寄存器分配后，通过使用 registers left over 来避免预定代码中的虚假依赖。这会使调试变得非常困难，因为变量不再存放于原本的寄存器中了。
- **-funswitch-loops:** 将无变化的条件分支移出循环，取而代之的将结果副本放入循环中。