# Project2 report

## 前言

这次Project2的任务是设计JAVA和C的数组乘法的简易程序，并分析发现的问题。我按照：进行实验——发现问题——寻找原因的步骤。我首先用JAVA和C进行了一系列的对比实验，得出C在数组较小时速度更快、java在数组较大时速度更快的结论，并且在实验中发现C的运行时间随数据量线性上升，而JAVA的运行时间上升逐渐变缓的特点。由于JAVA运行时间上升曲线跟预估不一致，我通过搜索资料、分析JIT的优化策略、查看优化后反编译的汇编码的方式，找到了JAVA运行时间上升曲线非线性的原因。

## 实验与提出问题

### 一、基本程序

C代码部分：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
clock_t start, finish;

int main()
{
  srand(1000);    // 修改随机数种子
  long n = 1000000; // 6个0
  // 以下三行计算时间
  for(int i = 0 ; i < 1; i++)  //如果需要进行多次循环，修改i的循环次数
  calculate_time(n);
  printf("\n");
}

int calculate_time(int n)
{
  int *a = (int *)malloc(sizeof(int) * n);  //动态申请空间，记得释放指针！
  int *b = (int *)malloc(sizeof(int) * n);
  int *c = (int *)malloc(sizeof(int) * n);
  for (int i = 0; i < n; i++) // 对每一个a和b进行赋值
  {
    a[i] = rand() % 65536+0;  //生成随机数范围0-2^16-1
    b[i] = rand() % 65536+0;
    // if(i % (n/10) == 0) printf("running %d times, max is %d\n",i, n);
  }
  printf("finished valuing\n");

  start = clock(); // 记录开启的时间
  for (int i = 0; i < n; i++)
  {
    c[i] = a[i] * b[i];
    // if(i % (n/10) == 0) printf("running %d times, max is %d\n",i, n);
  }
  finish = clock();//记录结束的时间

  free(a);  //释放指针！
  free(b);
  free(c);
  double time = (double)(finish - start) / CLOCKS_PER_SEC * 1000000; // 计算us
  printf("%lg ", time);  //输出时间，单位us
  return 0;
}
```

JAVA代码部分：

```
1 import java.util.Scanner;
2 import java.util.Random;
3
4 public class Compare {
5     public static void main(String[] args){
6         int n = 100000;
7         for(int i = 0; i < 1; i++)  //修改循环次数
8         calculate_time(n);
9         System.out.println();
```

```
 10      }
11
12      public static void calculate_time(int n){
13          Random r = new Random();
14          int[] a = new int[n];
15          int[] b = new int[n];
16          int[] c = new int[n];
17          for(int i = 0; i < n; i++){
18              a[i] = r.nextInt(1 << 16);  //生成随机数范围为0-2^16-1
19              b[i] = r.nextInt(1 << 16);
20          }
21          long start = System.nanoTime();  //开始的时间
22          for(int i= 0; i < n; i++){
23              c[i] = a[i] * b[i];
24          }
25          long finish = System.nanoTime();  //结束的时间
26          long time_difference = (finish - start)/1000;  //计算时间，单位us
27  //        System.out.printf("%d ",time_difference);
28      }
29  }
```

## 二、对比在一次运算中，二者花费时间（单独运行5次）

- n=10^4：C平均35us，JAVA平均125us，JAVA有时会到达400us，波动较大
- n=10^5：C平均330us，JAVA平均300，但是JAVA有时会出现特别高的比如900us的情况
- n=10^6：C平均用时3000us，JAVA平均用时2900us
- n=10^7：C平均用时30000us，JAVA平均用时11500us。测试时C的方差较大，普遍在35000和25000处还有分布，而JAVA在刚开始时分布不太稳定，后来基本稳定于11500数值
- n=10^8：C平均用时340000us，JAVA平均用时78558us：C偏差在10000内，JAVA开始时间逐渐减小，最终稳定在78000左右
- n=10^9：JAVA报OutOfMemoryError。搜集资料后发现：在JAVA中，数组长度不可以超过*Integer.MAX_VALUE*，即2^31 - 1个元素，如果尝试创建长度超过这个限制的数组，则会引发OutOfMemoryError异常；C的动态空间申请在64位系统上理论可达2^64-1个元素，远超过10^9，但是实际操作时发现会因程序超过内存上限而killed。

```
running 0 times, max is 1000000000
running 100000000 times, max is 1000000000
running 200000000 times, max is 1000000000
running 300000000 times, max is 1000000000
running 400000000 times, max is 1000000000
running 500000000 times, max is 1000000000
running 600000000 times, max is 1000000000
running 700000000 times, max is 1000000000
running 800000000 times, max is 1000000000
running 900000000 times, max is 1000000000
finished valuing
running 0 times, max is 1000000000
running 100000000 times, max is 1000000000
running 200000000 times, max is 1000000000
running 300000000 times, max is 1000000000
Killed
```

图1.数组乘法在3e^9-4e^9时被程序内存超过上限

**总结：C在处理较少数据的时候速度更快，JAVA在处理大量数据时更快。C的时间上升趋于线性，而JAVA的时间上升逐渐变缓慢，因此JAVA在处理大量数据时快于C。**

## 三、对比多次执行循环，二者花费时间

**n=10^5，循环100次**：取3次JAVA的输出绘图，可以看出：JAVA的程序开始时都是从一个较高的值骤减，直到稳定到一个比较低的值

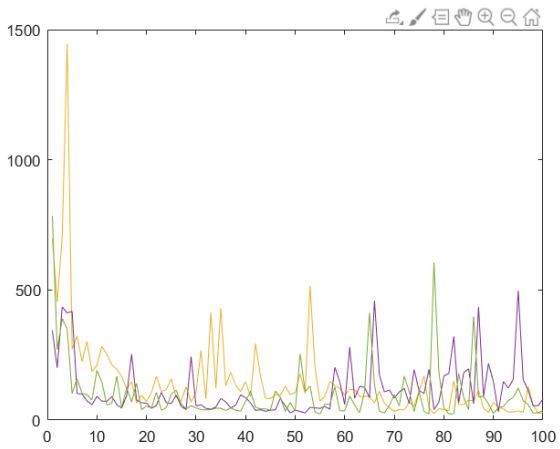



图2.JAVA程序在循环100次每次所需时间绘图

取5次C的100次循环输出绘图，发现图中在n较小时有部分异常点

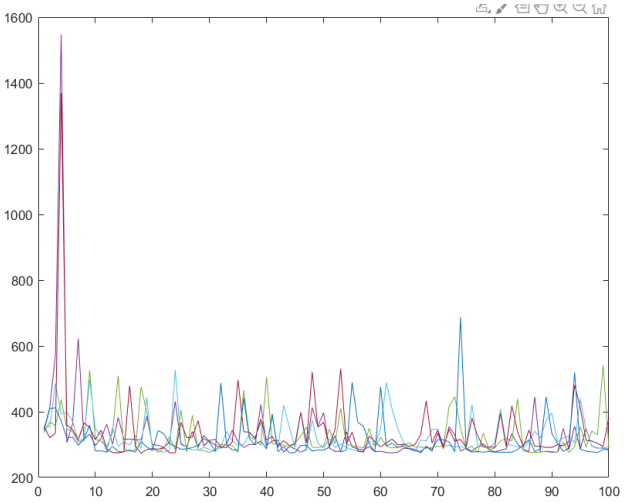



图3.C程序在循环100次每次所需时间绘图

去除异常点后，再次绘图

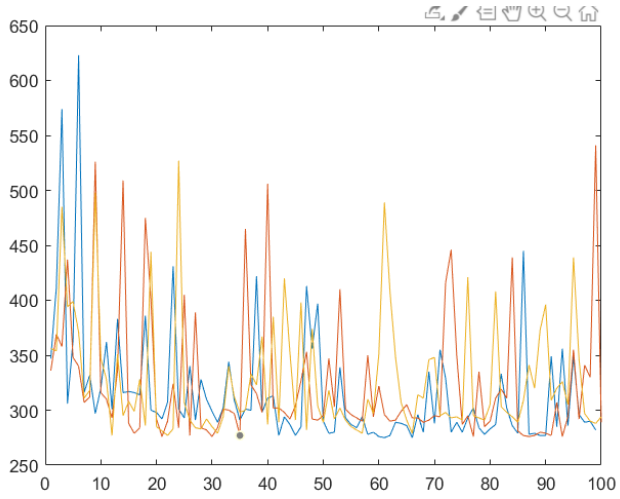
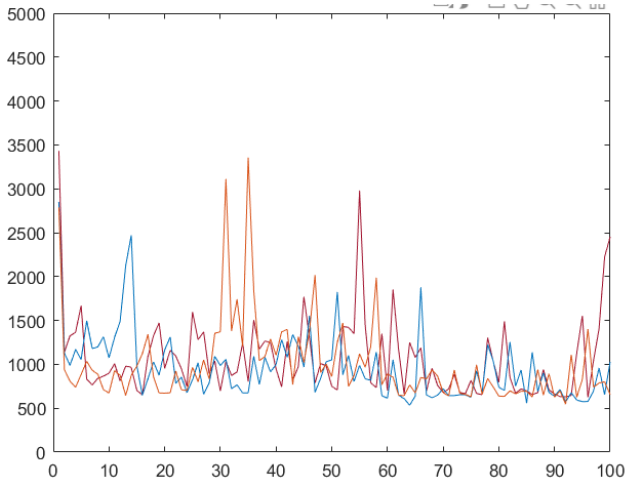



图4.处理后的C程序在循环100次每次所需时间绘图

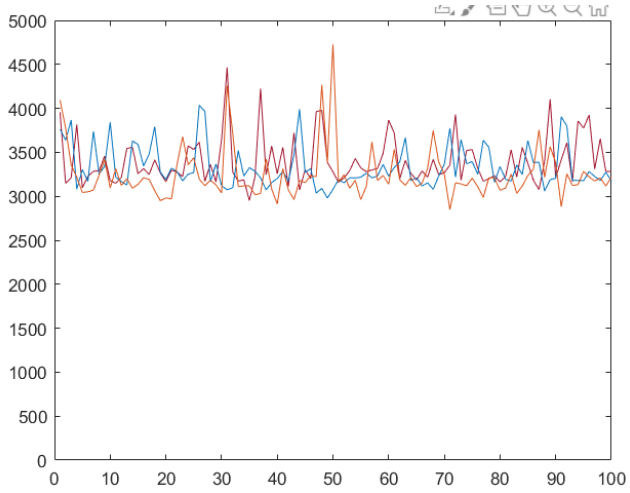
对比可发现JAVA程序开始时的花费时间普遍较长，而在循环十次左右后，稳定时的值都在100 us附近。而C的稳定值变化不大，始终在300us左右，因此C的运行时间远高于JAVA的运行时间。

**n=10^6时，循环100次**：

取三次JAVA的运行时间并绘图：可见：三次JAVA的图线也是从一个比较高的值（3000附近）骤减稳定到1000us左右。




取三次C的数据并绘图：数据基本稳定与3300us左右




**分析：JAVA的运行时间存在一个大幅下降的过程，下降后的时间远小于C语言。在这一系列实验之后，不难认为JAVA应该存在某些优化机制，能够减小多次循环的运行时间。**

## 资料搜集与做出假设

根据以上几组测验，并结合网上搜索的资料，发现JAVA的JIT不仅会将JAVA程序变为机器码，还会对代码进行多方面的优化，从而使代码运行速度加快。

结合网上搜索的JIT的优化类型，我猜想对于本项目，JIT所做的代码优化可能有以下几种：

- 热点代码优化：JIT编译器会将频繁执行的代码编译成本地代码以提高执行速度。这些频繁执行的代码被称为“热点代码”
- 内联函数：JIT编译器会将函数调用直接替换为函数本身的代码，从而消除了函数调用的开销。
- 循环展开：JIT编译器会将循环中的代码重复展开多次，从而减少循环的迭代次数，提高程序的执行速度。
- 缓存分析：JIT编译器会分析程序访问内存的模式，从而将数据预取到缓存中，提高程序的执行速度。

我猜想，JIT在前几次循环中把编译的代码优化了，并且把优化的代码储存在缓存中。后续循环使用缓存中的代码，因此提高了效率。

**因此，接下来的目标就是证实JIT对代码进行了优化，并尽量弄清楚优化了哪些内容。**

### 实验验证

**如果想要知道JAVA到底优化了哪些内容，最简单的方法就是把优化前的代码和优化之后的代码进行对比，然后找不同。**为此，我进行了如下操作：

1. 通过网上资料搜集，了解到JIT中有两种编译器，C1代表的是Client Compiler,C2代表的是Server Compiler。其中C1只是简单的编译，而C2在收集到更多信息之后，会进行更加深入的编译和优化。
2. 使用HIDIS通过反汇编来查看优化后的内容，使用JITWatch进行可视化分析。

于是，我尝试使用jitwatch来查看JIT对我代码进行的优化，jitwatch能将优化后的机器码反编译为汇编代码，因此通过查看该代码就可以知道JIT对代码进行了哪些优化。

**我对n=10^6,循环1次进行了优化查看：**

```
 1  import java.util.Scanner;
 2  import java.util.Random;
 3
 4  public class Compare {
 5      public static void main(String[] args){
 6          int n = 100000;
 7          for(int i = 0; i < 1; i++)
 8          calculate_time(n);
 9          System.out.println();
10      }
11
12      public static void calculate_time(int n){
```

```
13          Random r = new Random();
14          int[] a = new int[n];
15          int[] b = new int[n];
16          int[] c = new int[n];
17          for(int i = 0; i < n; i++){
18              a[i] = r.nextInt(1 << 15) - 1 << 15;
19              b[i] = r.nextInt(1 << 15) - 1 << 15;
20          }
21          long start = System.nanoTime();
22          for(int i= 0; i < n; i++){
23              c[i] = a[i] * b[i];
24          }
25          long finish = System.nanoTime();
26          long time_difference = (finish - start)/1000;
27  //        System.out.printf("%d ",time_difference);
28      }
29  }
```

经过JIT优化(C2)的对应汇编代码为(只放部分，源码见附C2.txt)：

```
1 # {method} {0x0000020e3b2005e0} 'calculate_time' '(I)V' in 'Compare'
2 [Entry Point]
3 0x0000020e2298e820: int3
4 0x0000020e2298e821: data16 data16 nopw 0x0(%rax,%rax,1)
5 0x0000020e2298e82c: data16 data16 xchg %ax,%ax
6 0x0000020e2298e830: mov %eax,-0x6000(%rsp)
7 0x0000020e2298e837: push %rbp
8 0x0000020e2298e838: sub $0x70,%rsp
9 0x0000020e2298e83c: mov 0x18(%rdx),%r13d
10 0x0000020e2298e840: mov 0x20(%rdx),%r10
....
```

除此之外，还有未经优化的汇编代码（只放部分，源码见附C1.txt）

```
1 # {method} {0x0000020e3b2005e0} 'calculate_time' '(I)V' in 'Compare'
[Entry Point]
2 0x0000020e22991100: mov %eax,-0x6000(%rsp)
3 0x0000020e22991107: push %rbp
4 0x0000020e22991108: sub $0x150,%rsp
5 0x0000020e2299110f: movabs $0x20e3b200790,%rsi  ;   {metadata(method data for {method} {0x0000020e3b2005e0} 'calculate_time' '(I)V' in 'C
6 0x0000020e22991119: mov 0xdc(%rsi),%edi
7 0x0000020e2299111f: add $0x8,%edi
8 0x0000020e22991122: mov %edi,0xdc(%rsi)
9 0x0000020e22991128: movabs $0x20e3b2005d8,%rsi  ;   {metadata({method} {0x0000020e3b2005e0} 'calculate_time' '(I)V' in 'Compare')}
10 0x0000020e22991132: and $0x1ff8,%edi
11 0x0000020e22991138: cmp $0x0,%edi
12 0x0000020e2299113b: je L0023
13             L0000: mov %edx,0xe8(%rsp)
14 0x0000020e22991148: movabs $0x7c0045658,%rdx  ;   {metadata('java/util/Random')}
15 0x0000020e22991152: mov 0x60(%r15),%rax
16 0x0000020e22991156: lea 0x20(%rax),%rdi
17 0x0000020e2299115a: cmp 0x70(%r15),%rdi
18 0x0000020e2299115e: ja L0024
19 0x0000020e22991164: mov %rdi,0x60(%r15)
20 0x0000020e22991168: mov 0xa8(%rdx),%rcx
21 0x0000020e2299116f: mov %rcx,(%rax)
22 0x0000020e22991172: mov %rdx,%rcx
23 0x0000020e22991175: shr $0x3,%rcx
24 0x0000020e22991179: mov %ecx,0x8(%rax)
25 0x0000020e2299117c: xor %rcx,%rcx
26 0x0000020e2299117f: mov %ecx,0xc(%rax)
27 0x0000020e22991182: xor %rcx,%rcx
28 0x0000020e22991185: mov %rcx,0x10(%rax)
29 0x0000020e22991189: mov %rcx,0x18(%rax)  ;*new  ; - Compare::calculate_time@0 (line 13)
30             L0001: mov %rax,%rbx
31 0x0000020e22991190: movabs $0x20e3b200790,%rsi  ;   {metadata(method data for {method} {0x0000020e3b2005e0} 'calculate_time' '(I)V' in '
32 0x0000020e2299119a: addq $0x1,0x108(%rsi)
33 0x0000020e229911a2: movabs $0x20e3b04ae78,%rbx  ;   {metadata(method data for {method} {0x0000020e3b048598} '<init>' '()V' in 'java/util
34 0x0000020e229911ac: mov 0xdc(%rbx),%esi
...
```

C2代码相较于C1代码长度大幅缩短，运行时间从19ms变成10ms。部分能看出的代码优化如下：

- 方法内联：将calculate_time方法内联到main方法中，对应C2代码的第4-9行，将calculate_time的代码直接插入到了main方法中，避免了方法调用的开销。可在C1代码中发现多次调用方法的记录，而在C2代码中由于方法内联，没有发现调用方法。
- 热点代码优化：对循环计算c数组的热点代码进行了优化，对应C2代码的第13-28行。JIT编译器将这段热点代码编译为本地机器码，并针对机器架构进行了优化，如使用向量化指令和CPU寄存器等。此外，JIT编译器还进行了一些优化，如数组边界检查消除和循环展开等。
- 数组边界检查消除：对于数组访问，JIT编译器进行了数组边界检查消除，对应C2代码的第17-20行。由于数组a和b的长度均为n，且在循环中的访问下标从0到n-1，因此不必进行边界检查，JIT编译器可以直接使用内存地址进行访问，避免了数组访问时的开销。

可见，经过这些优化，JAVA在后续的处理中速度大幅提升，也就造成了JAVA在循环了几次后运行时间大幅减少的情况。

### 关于在处理少量数据时，C比JAVA快的原因分析

1. JAVA和C数组的存储方式不同，C是按照指针的位置将数字按顺序堆叠，而JAVA中数组时引用实体变量，呈树状结构，彼此之间毫无关系。这可能导致JAVA在调用数据时更慢。
2. JAVA的JIT使用也会给程序带来限制，如启动时间延时、内存占用、编译开销等等，JIT对于比较小的程序基本无优化效果，且JIT的反而会使程序变慢，因此在少量数据即优化不明显时，C的速度快于JAVA的速度。

## 总结

至此，我对比了JAVA和C在运行数组乘法时的表现情况，并着重JIT对于JAVA优化的角度进行了探究。这次Project我拓展了很多关于JAVA的底层的知识，了解了JIT、JVM、Maven、HIDIS、Jitwatch、汇编语言还有诸多相关的内容，感叹于一个看似简单的循环和乘法竟然可以有这么多的细节可以挖掘。