Computing the dot product of two vectors report

杨官宇涵12313614

由于代码和脚本较多，如需查看其余的代码和脚本以及测试的完整流程，请查看我的GitHub

(YangGuanyuhan github)

1. project 分析

分析project 找到一些注意点与关键点

1. 进行不同语言的比较注意控制变量（系统环境，电脑配置等，以及多个代码的优化程度相同）
2. 测量的时间应与向量长度的增加呈正相关的数据可视化
3. 不同数据类型的性能变化、语言特定优化的影响（对最基本的代码进行不断的优化，查看最终优化结果与一开始的差异）、内存使用特性（使用valgrind等工具判断内存差距）。
4. 注意结果的正确性验证

2. file structure

```
christylinux@christywindowscomputer:~/CS219/project2$ tree
.
├── build
├── inc
├── scripts
│   ├── c_performance_run_test.sh
│   ├── compare_c_optimization.sh
│   ├── compare_performance.sh
│   ├── java_performance_run_test.sh
│   └── temp_test_input.txt
├── src
│   ├── C_vector_dot_product.c
│   ├── JavaVectorDotProduct.class
│   ├── JavaVectorDotProduct.java
│   ├── Makefile
│   ├── c_vector_dot_product
│   ├── optimal_c_vector_product
│   ├── optimal_c_vector_product.c
│   ├── testcase_generator
│   └── testcase_generator.c
└── test
    ├── C_run_performance_test.sh
    ├── Makefile
    ├── compare_performance.sh
    ├── run_java_tests.sh
    ├── run_tests.sh
    ├── run_valgrind.sh
    ├── temp_test_input.txt
    ├── test_dot_product
    ├── test_dot_product.c
    ├── test_error_input.txt
```

```
    ├── test_input.txt
    └── vector_dot_product

6 directories, 26 files
```

3. 安全性验证

将写好并验证的c语言代码发给claude，给出如下prompt

```
1.注意代码的整体结构，符合开发规范。
2.注意代码规范，实现良好的方法和类的使用，实现良好的模块化使得代码能够拥有良好的移植性
3.保证代码的健壮性。处理好任何输入错误或边界条件。同时任何方法中都需要检查参数有效性 强调，对于任何指针操作，使用前都需要检查有效性
4.安全性代码应尽量减少漏洞风险，比如防止 SQL 注入、跨站脚本攻击 (XSS) 等问题，
```

进一步完善代码，提高健壮性

4. c程序解读

1. 使用宏来表示错误代码，便于报错后打印输出信息进行调试

```
// 定义错误代码
#define DOT_PRODUCT_SUCCESS 0
#define DOT_PRODUCT_NULL_PTR_ERROR 1
#define DOT_PRODUCT_INVALID_SIZE_ERROR 2
```

2. 计时机制

- 使用 `gettimeofday` 来测量时间，计时机制基于 **系统的实时时钟（RTC）**，提供微秒级（μs）精度的时间测量。
- 统一使用μs作为单位，便于后续脚本完成分析

```
struct timeval start, end;
long elapsed_time_us;  // 使用长整型存储微秒
      gettimeofday(&start, NULL);


      gettimeofday(&end, NULL);

    // 计算微秒差值
    elapsed_time_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec -
start.tv_usec);
```

3. 函数核心，这里以int为例

- 未使用任何优化的方法，最朴素的计算点乘的方法，便于后续和java做对比
- 维护代码茁壮性，对于函数的输入完成检测，规定空指针为不合法的输入

```
int dotProductInt(const int *vecA, const int *vecB, int size, int *result)
{
```

```
    // 错误检查
    if (vecA == NULL || vecB == NULL || result == NULL)
    {
        return DOT_PRODUCT_NULL_PTR_ERROR;
    }

    if (size <= 0)
    {
        return DOT_PRODUCT_INVALID_SIZE_ERROR;
    }

    // 计算点乘
    *result = 0;
    for (int i = 0; i < size; i++)
    {
        *result += vecA[i] * vecB[i];
    }

    return DOT_PRODUCT_SUCCESS;
}
```

4.错误信息打印函数

对于调用完成错误打印，便于后续调试

```
// 错误打印函数
void printError(int errorCode)
{
    switch (errorCode)
    {
    case DOT_PRODUCT_NULL_PTR_ERROR:
        printf("Error: Null pointer provided\n");
        break;
    case DOT_PRODUCT_INVALID_SIZE_ERROR:
        printf("Error: Invalid vector size\n");
        break;
    default:
        printf("Unknown error: %d\n", errorCode);
    }
}
```

5. 数据输入格式

第一行代表测试用例个数n
接下来n行
第一列代表数据类型，是一个string，可能的格式int，long，short，char，float，double
第二列代表vector的size
接下来分别代表不同的vector，格式为[3,4]等，两个vector中间使用空格分开
下面是一个具体的例子

```
6
int 3
[1,2,3] [4,5,6]
long 2
[1000000000,2000000000] [3000000000,4000000000]
short 2
[100,200] [300,400]
char 2
[10,20] [30,40]
float 2
[1.5,2.5] [3.5,4.5]
double 2
[1.5,2.5] [3.5,4.5]
```

6.main函数

使用malloc分配内存，同时在main方法中完成释放

```
scanf("%s %d", dataType, &size);

        // 检查size是否有效
        if (size <= 0)
        {
            printf("Error: Invalid vector size\n");
            continue;
        }

        // 根据数据类型处理
        if (strcmp(dataType, "int") == 0)
        {
            // 为int类型分配内存
            int *vecA = (int *)malloc(size * sizeof(int));
            int *vecB = (int *)malloc(size * sizeof(int));

            if (vecA == NULL || vecB == NULL)
            {
                printf("Error: Memory allocation failed\n");
                if (vecA)
                    free(vecA);
                if (vecB)
                    free(vecB);
                continue;
            }

            // 读取向量A
            char c;
            scanf(" %c", &c); // 读取'['
            for (int i = 0; i < size; i++)
            {
                scanf("%d", &vecA[i]);
                if (i < size - 1)
                {
                    scanf("%c", &c); // 读取','
                }
            }
```

```
            scanf("%c", &c); // 读取']'

            // 读取向量B
            scanf(" %c", &c); // 读取'['
            for (int i = 0; i < size; i++)
            {
                scanf("%d", &vecB[i]);
                if (i < size - 1)
                {
                    scanf("%c", &c); // 读取','
                }
            }
            scanf("%c", &c); // 读取']'

            // 计算点乘
            int result;
            int status = dotProductInt(vecA, vecB, size, &result);

            // 输出结果
            if (status == DOT_PRODUCT_SUCCESS)
            {
                printf("Int dot product result: %d\n", result);
            }
            else
            {
                printError(status);
            }

            // 释放内存
            free(vecA);
            free(vecB);
        }
```

5. 对c程序进行正确性校验（从输入和内存方面）

1. 使用bash脚本对程序进行确认性检验

具有三个测试单元

分别是`test_dot_product.c`完成错误处理，尤其是空指针的测试

```
// Test error handling
void test_error_handling()
{
    printf("Testing error handling...\n");

    int vecA[] = {1, 2, 3};
    int vecB[] = {4, 5, 6};
    int result;

    // Null pointer tests
    int status1 = dotProductInt(NULL, vecB, 3, &result);
    assert(status1 == DOT_PRODUCT_NULL_PTR_ERROR);

    int status2 = dotProductInt(vecA, NULL, 3, &result);
```

```
    assert(status2 == DOT_PRODUCT_NULL_PTR_ERROR);

    int status3 = dotProductInt(vecA, vecB, 3, NULL);
    assert(status3 == DOT_PRODUCT_NULL_PTR_ERROR);

    // Invalid size tests
    int status4 = dotProductInt(vecA, vecB, 0, &result);
    assert(status4 == DOT_PRODUCT_INVALID_SIZE_ERROR);

    int status5 = dotProductInt(vecA, vecB, -1, &result);
    assert(status5 == DOT_PRODUCT_INVALID_SIZE_ERROR);

    printf("Error handling tests passed\n");
}
```

接下来是对正确和错误输入的处理

```
christylinux@christywindowscomputer:~/CS219/project2/test$ ./run_tests.sh
rm -f test_dot_product vector_dot_product performance_test
gcc -Wall -Wextra -std=c99 -I../src -DTESTING -o test_dot_product
test_dot_product.c -lm
test_dot_product.c:1: warning: "TESTING" redefined
    1 | #define TESTING
      |
<command-line>: note: this is the location of the previous definition
gcc -Wall -Wextra -std=c99 -I../src -o vector_dot_product
../src/vector_dot_product.c -lm
Running unit tests...
./test_dot_product
Starting vector dot product unit tests
Testing int dot product...
Int dot product tests passed
Testing long long dot product...
Long long dot product tests passed
Testing short dot product...
Short dot product tests passed
Testing char dot product...
Char dot product tests passed
Testing float dot product...
Float dot product tests passed
Testing double dot product...
Double dot product tests passed
Testing error handling...
Error handling tests passed
All tests passed!

Running main program tests...
./vector_dot_product < test_input.txt
Int dot product result: 32
Long Long dot product result: -7446744073709551616
Short dot product result: -21072
Char dot product result: 76
Float dot product result: 16.500000
Double dot product result: 16.500000
程序运行时间: 9590 微秒
```

```
Running error handling tests...
./vector_dot_product < test_error_input.txt || true
Error: Invalid vector size
Error: Invalid vector size
Error: Invalid vector size
程序运行时间: 250 微秒
All tests completed!
```

2. 使用bash和valgrind进行全面内存分析

```
==8909== HEAP SUMMARY:
==8909==     in use at exit: 0 bytes in 0 blocks
==8909==   total heap usage: 14 allocs, 14 frees, 5,236 bytes allocated
==8909==
==8909== All heap blocks were freed -- no leaks are possible
==8909==
==8909== For lists of detected and suppressed errors, rerun with: -s
==8909== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

内存无溢出和泄露，代码安全

6. java程序解读（使用默认的垃圾回收、内存管理机制）

1.使用正常的scanner进行读取

2. 为了使得实验标准,使得java与c能够有着相似的代码执行顺序，相同的茁壮性

3.dotProductInt

```
 // Integer dot product
    public static int dotProductInt(int[] vecA, int[] vecB, int size, int[]
result) {
        // Error checking
        if (vecA == null || vecB == null || result == null) {
            return DOT_PRODUCT_NULL_PTR_ERROR;
        }

        if (size <= 0) {
            return DOT_PRODUCT_INVALID_SIZE_ERROR;
        }

        // Calculate dot product
        result[0] = 0;
        for (int i = 0; i < size; i++) {
            result[0] += vecA[i] * vecB[i];
        }

        return DOT_PRODUCT_SUCCESS;
    }
```

4.parseVector

解析数据存入数组

```
 private static Object parseVector(String vectorStr, String dataType, int size)
{
        // Remove [ and ] characters
        vectorStr = vectorStr.substring(1, vectorStr.length() - 1);
        String[] elements = vectorStr.split(",");
```

5.java也做了相应的正确性测试，详情请看github

7. testCaseGenerator

1.使用方法

生成5个随机类型的测试用例，向量长度为10：

```
./testcase_generator
```

生成10个int类型的测试用例，向量长度为5：

```
./testcase_generator -n 10 -t int -l 5
```

生成20个随机类型的测试用例，随机向量长度，并输出到文件：

```
./testcase_generator -n 20 -t random -r -o test_input.txt
```

生成指定类型（例如float）的测试用例：

```
./testcase_generator -t float -n 15 -l 8
```

获取帮助信息：

```
./testcase_generator -h
```

1. 自定义测试用例数量
2. 选择随机或特定的数据类型
3. 选择固定或随机的向量长度
4. 输出到文件或标准输出

2.高度自定义的随机输出，能够对于两个文件提供足够有代表性的输入或者完全随机

8. c语言与java的性能分析对比

1 . 测试环境说明

```
设备名称    christywindowscomputer
处理器 13th Gen Intel(R) Core(TM) i9-13900HX   2.20 GHz
机带 RAM  16.0 GB (15.7 GB 可用)
设备 ID   E57C76E3-CD9E-4C4C-A6B5-AC733E4CF6CC
产品 ID   00342-31475-80601-AAOEM
系统类型    64 位操作系统, 基于 x64 的处理器
笔和触控    没有可用于此显示器的笔或触控输入
```

```
openjdk version "17.0.14" 2025-01-21
OpenJDK Runtime Environment (build 17.0.14+7-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 17.0.14+7-Ubuntu-124.04, mixed mode, sharing)
```

```
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
PRETTY_NAME="Ubuntu 24.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.2 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-
policy"
UBUNTU_CODENAME=noble
LOGO=ubuntu-logo
```

由于java中存在一些特定的机制来优化运行速度例如：

1. Take advantage of the JVM's JIT compiler to optimize bytecode to native machine code at runtime, improving performance.
2. Consider using GraalVM's native image feature to compile Java applications ahead of time, resulting in faster startup times and reduced memory footprint.

所以在C语言的编译选项中统一使用O1作为优化

因为java存在预热，使得jit生效，所以以下都是对10000组样例进行分析测试之后取平均值

统一使用该脚本进行对比分析

- 保证同一个输入
- 忽视编译和运行时间
- 为了更加直观，使用ratio Java/C时间比率: RATIO (比值越大说明Java相对C越慢)

```
# 使用相同输入测试C程序
echo "使用相同输入测试C程序..."
C_RESULT=$($C_VECTOR_DOT < $OUTPUT_FILE)
```

```
# 使用相同输入测试Java程序
echo "使用相同输入测试Java程序..."
SCRIPT_DIR="$(pwd)"
JAVA_RESULT=$(cd ../src && java JavaVectorDotProduct <
"${SCRIPT_DIR}/${OUTPUT_FILE}")

# 从C程序输出中提取执行时间
C_TIME=$(echo "$C_RESULT" | grep -o "Program execution time: [0-9]\+
microseconds" | grep -o "[0-9]\+")
if [ -z "$C_TIME" ]; then
    # 尝试其他格式
    C_TIME=$(echo "$C_RESULT" | grep -o "执行时间: [0-9]\+ 微秒" | grep -o "[0-
9]\+")
    if [ -z "$C_TIME" ]; then
        C_TIME=$(echo "$C_RESULT" | tail -n 1 | grep -o "[0-9]\+")
    fi
fi

# 从Java程序输出中提取执行时间
JAVA_TIME=$(echo "$JAVA_RESULT" | grep -o "程序运行时间: [0-9]\+ 微秒" | grep -o "
[0-9]\+")
if [ -z "$JAVA_TIME" ]; then
    # 尝试其他格式
    JAVA_TIME=$(echo "$JAVA_RESULT" | grep -o "Program execution time: [0-9]\+
microseconds" | grep -o "[0-9]\+")
    if [ -z "$JAVA_TIME" ]; then
        JAVA_TIME=$(echo "$JAVA_RESULT" | tail -n 1 | grep -o "[0-9]\+")
    fi
fi

# 检查是否成功提取时间
if [ -z "$C_TIME" ] || [ -z "$JAVA_TIME" ]; then
    echo "警告: 无法提取执行时间，显示原始输出结果"
    echo "C程序输出的最后几行:"
    echo "$C_RESULT" | tail -n 5
    echo ""
    echo "Java程序输出的最后几行:"
    echo "$JAVA_RESULT" | tail -n 5
    exit 1
fi

# 计算平均执行时间
C_AVG=$(echo "scale=2; $C_TIME / $NUM_CASES" | bc)
JAVA_AVG=$(echo "scale=2; $JAVA_TIME / $NUM_CASES" | bc)

# 计算差异
RATIO=$(echo "scale=2; $JAVA_TIME / $C_TIME" | bc)

# 输出比较结果
echo "========================================="
echo "性能比较结果:"
echo "C总执行时间: $C_TIME us (平均每个测试案例: $C_AVG us)"
echo "Java总执行时间: $JAVA_TIME us (平均每个测试案例: $JAVA_AVG us)"
echo "Java/C时间比率: $RATIO (比值越大说明Java相对C越慢)"
```

```
echo "========================================="
```

运行结果展示

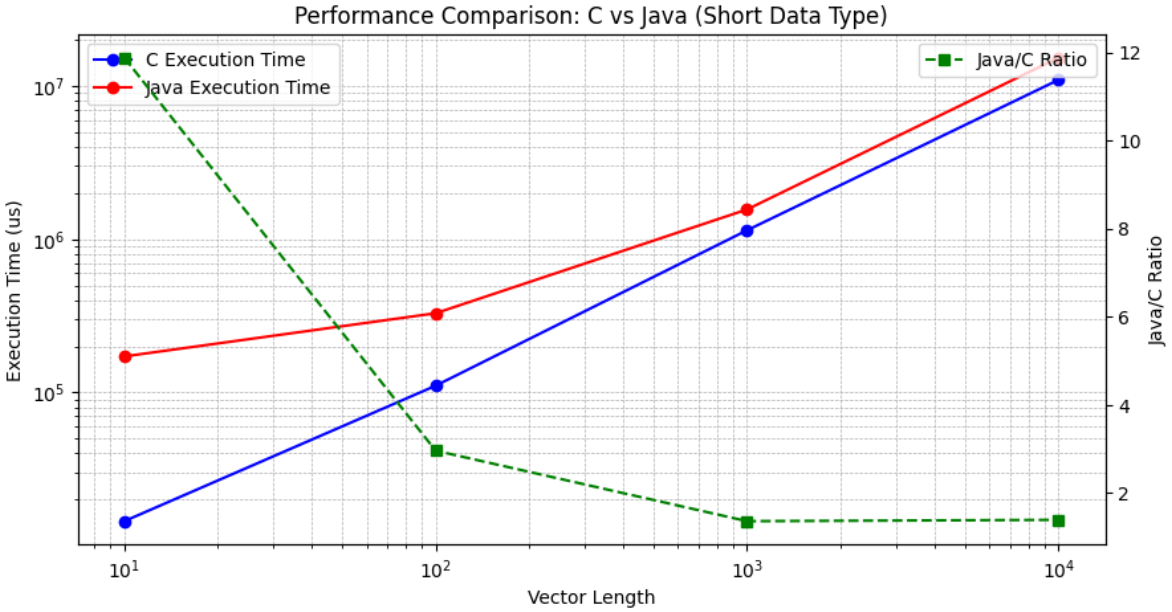
```
christylinux@christywindowscomputer:~/CS219/project2/scripts$
./compare_performance.sh
Compiling Java program...
=========================================
性能比较测试
测试配置:
- 测试案例数量: 10000
- 数据类型: double
- 向量长度: 1000
- 随机长度: false
=========================================
生成测试用例中: 10000 个 double 类型, 向量长度 1000
Test cases written to 'temp_test_input.txt'
使用相同输入测试C程序...
使用相同输入测试Java程序...
=========================================
性能比较结果:
C总执行时间: 2420574 us (平均每个测试案例: 242.05 us)
Java总执行时间: 3497539 us (平均每个测试案例: 349.75 us)
Java/C时间比率: 1.44 (比值越大说明Java相对C越慢)
=========================================
```

2.对于int类型的比较

向量长度	C总执行时间 (us)	Java总执行时间 (us)	Java/C 时间比率
10	16095	165265	10.26
100	110264	325788	2.95
1000	1289870	1467086	1.13
10000	10657444	14370772	1.34

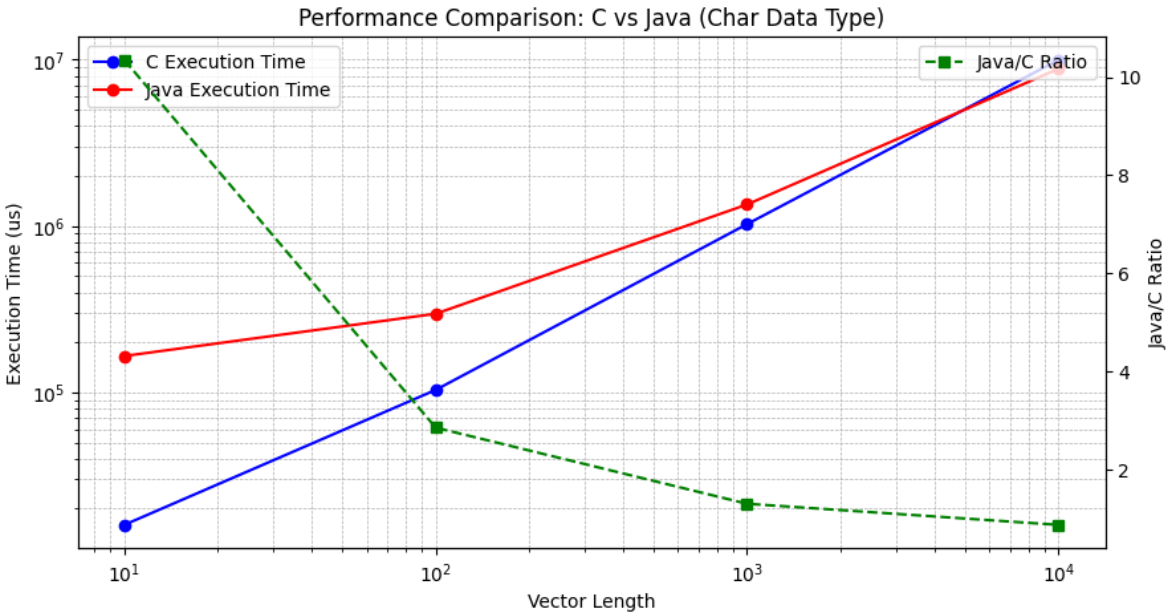
3.对于short类型的比较

Vector Length	C Execution Time (us)	Java Execution Time (us)	Java/C Ratio
10	14448	171786	11.88
100	110682	328643	2.96
1000	1143727	1563470	1.36
10000	10956098	15282541	1.39



4.对于char类型的比较

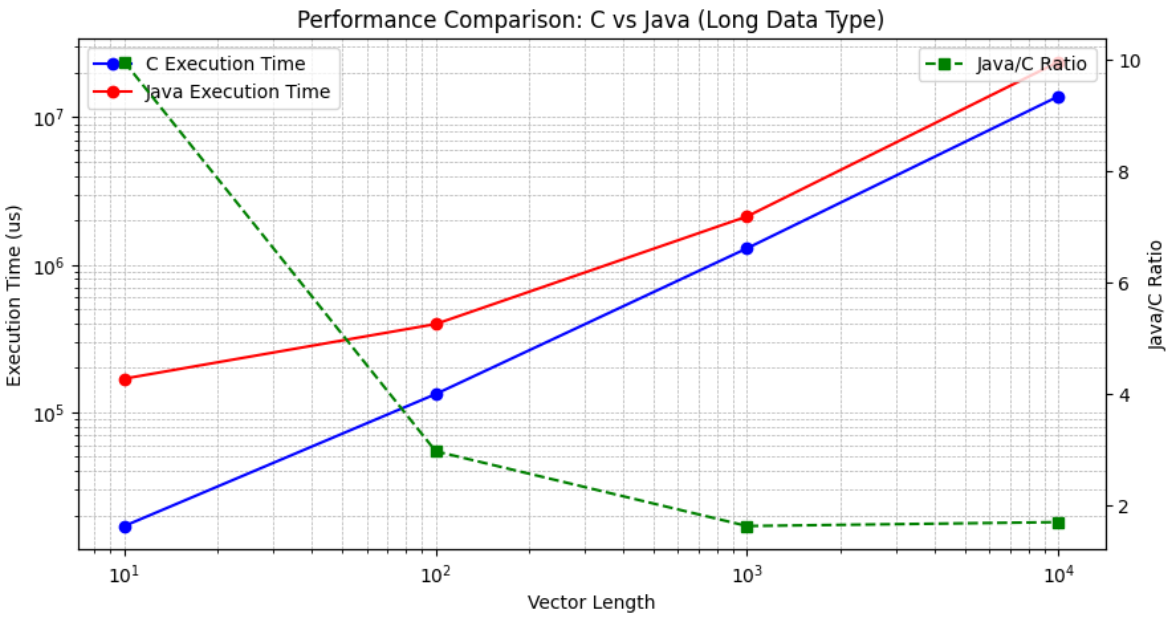
Vector Length	C Execution Time (us)	Java Execution Time (us)	Java/C Ratio
10	15990	165558	10.35
100	103544	297092	2.86
1000	1027831	1348904	1.31
10000	9976080	8822160	0.88



5.对于long类型的比较

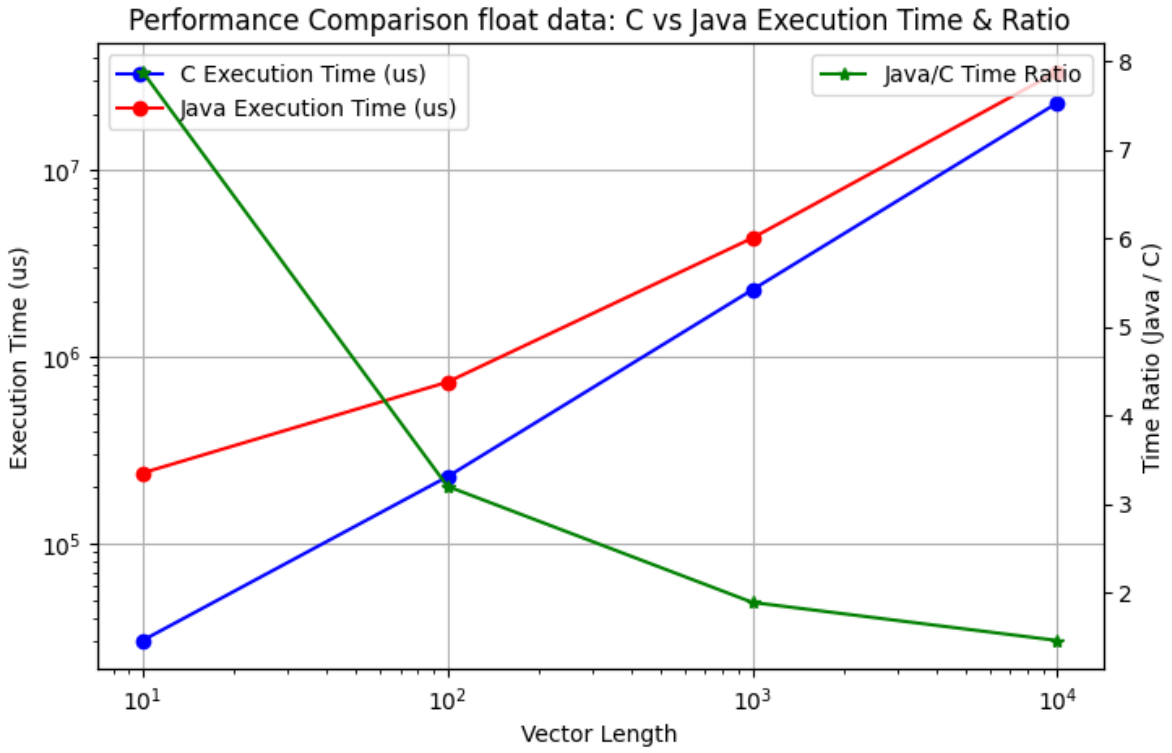
Vector Length	C Execution Time (us)	Java Execution Time (us)	Java/C Ratio
10	16992	169331	9.96
100	133174	397012	2.98
1000	1295504	2129333	1.64

Vector Length	C Execution Time (us)	Java Execution Time (us)	Java/C Ratio
10000	13874525	23758381	1.71



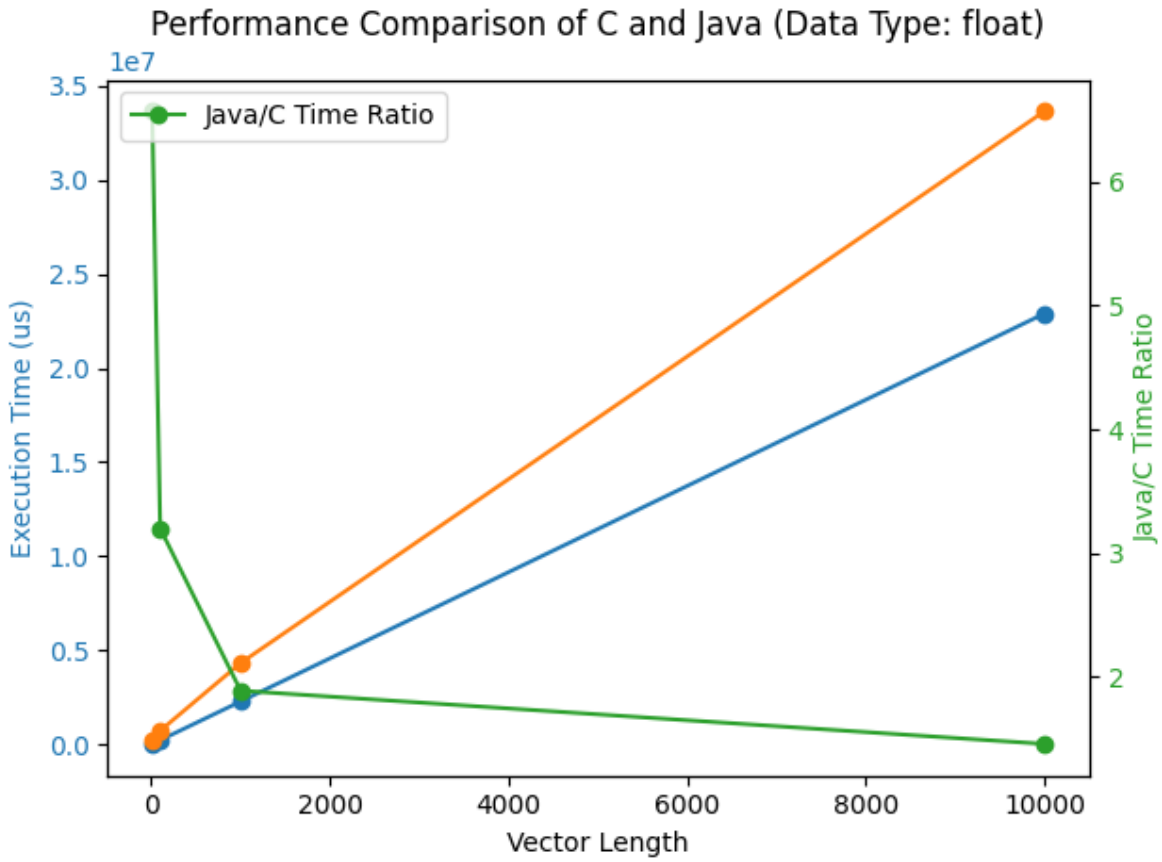
6.对于float类型的比较

Vector Length	C Execution Time (us)	Java Execution Time (us)	Java/C Ratio
10	30301	238953	7.88
100	228231	732595	3.20
1000	2285849	4325484	1.89
10000	22894146	33641110	1.46



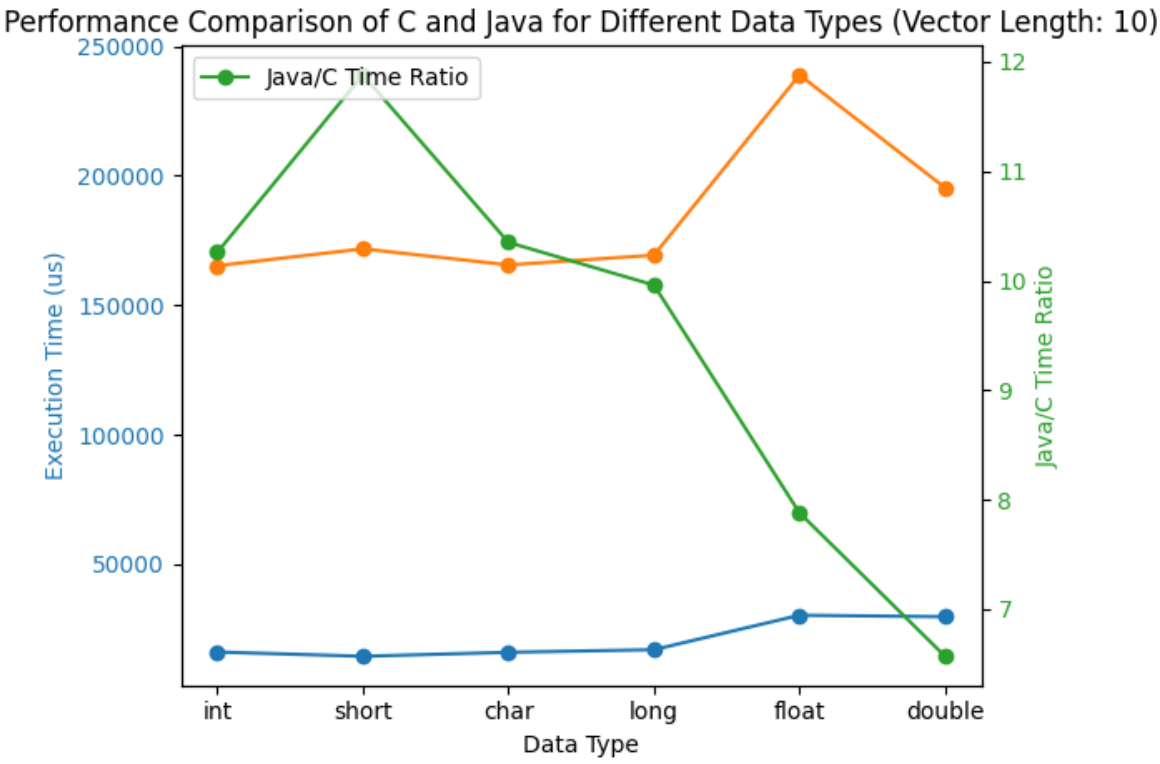
7.对于double类型的比较

Vector Length	C Total Execution Time (us)	Java Total Execution Time (us)	Java/C Time Ratio
10	29721	195362	6.57
100	255677	589070	2.30
1000	2420574	3497539	1.44
10000	22651244	26804110	1.18



8.Comparison of Execution Time on different types (Vector Length: 10)

Data Type	Vector Length	C Total Execution Time (us)	Java Total Execution Time (us)	Java/C Time Ratio
int	10	16095	165265	10.26
short	10	14448	171786	11.88
char	10	15990	165558	10.35
long	10	16992	169331	9.96
float	10	30301	238953	7.88
double	10	29721	195362	6.57



结论

这真的太神奇了，Java 在短向量处理上较慢，但在大规模数据时有一定优化。甚至于比c还快，这可能和我的c写的一般并且做了特别多安全性检验有关。但是总的来说c确实会更快，尤其对于短的向量特别明显，尤其是能够快到10倍以上。因为我们已经忽视了java的预热，所以对于c如果把向量长度大的部分做好优化，也能够比java快很多倍，至少十倍

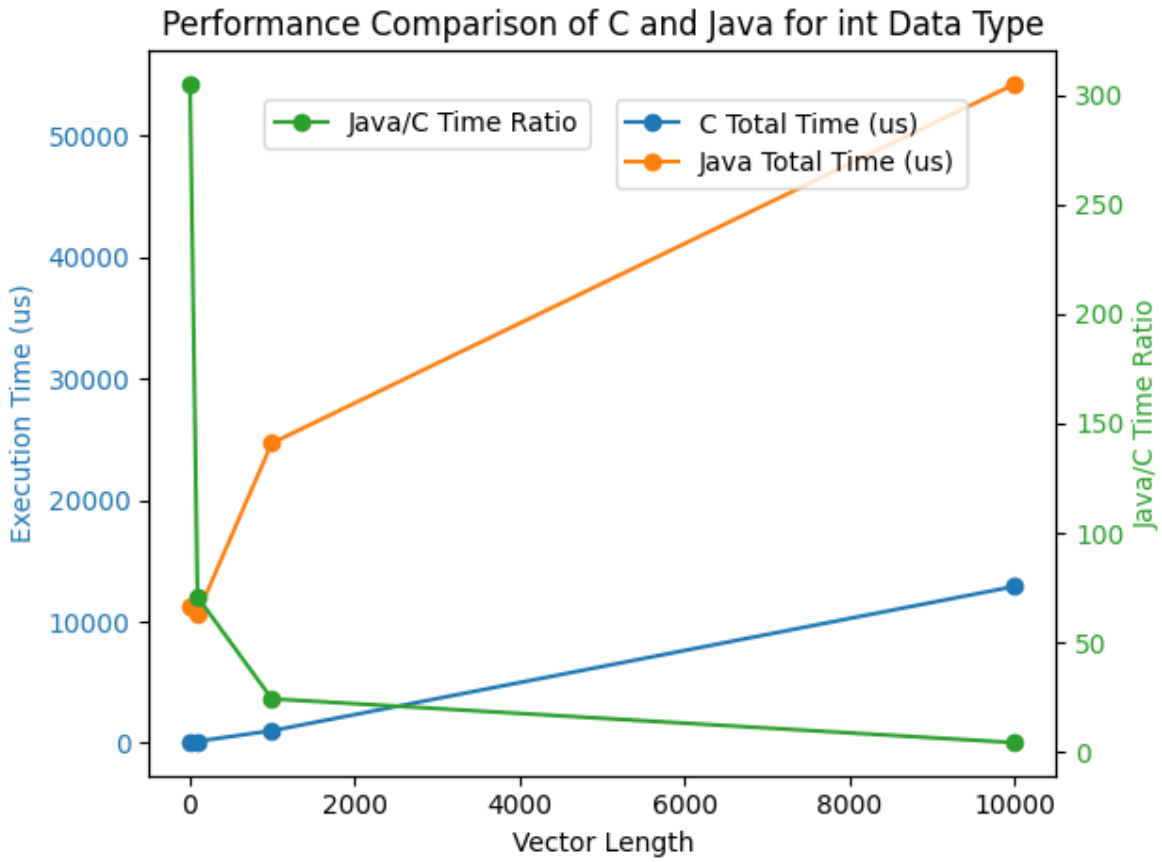
1. 在vectorsize较低的情况下 C 实现在速度上比 Java 有显著优势。是一个10倍左右的优势，但是当size变大，优势会变小
2. 对于不同的数据类型之间，从整数到浮点数需要的时间缓慢变大，在两个语言中呈现出同样的整体趋势，但是奇怪的一点是1bytechar和4byte的inter整体上时间查不多，可能是int是最为常用的数据类型，所以编译器对于int优化在两种语言中都比较多，其余都比较符合直觉
3. java相对于c可以有jvm做出优化，所以当长度变大，优化越明显
4. C语言是编译型语言，直接编译成机器码，运行时没有额外的虚拟机开销，所以对于短的向量长度会特别的快，而Java是运行在JVM上的，虽然IT编译器能优化热点代码，但启动时可能会有一些预热时间。
5. 测量的时间与向量长度的增加呈正相关，在c中是特别明显的线性相关，但是在java中由于size变大带来的优化作用，使得是一个曲线，斜率表示运行速度，说明了运行速度的增加

9. c语言与java的性能分析对比（考虑预热时间）

NUM_CASES=10，考虑java的预热时间，以int为例进行分析

Vector Length	C Total Time (us)	Java Total Time (us)	Java/C Time Ratio
10	37	11293	305.21
100	150	10620	70.80
1000	1019	24684	24.22

Vector Length	C Total Time (us)	Java Total Time (us)	Java/C Time Ratio
10000	12902	54261	4.20



结论

在NUM_CASES=10的情况下，预热时间使得c比java快恐怖的300倍，这就是不要虚拟机带来的优势，但是这也使得c的移植性较差

10. 优化后的c语言与优化前的对比

优化部分

注意编译选项变为

```
gcc -O3 -march=native -fopenmp -mavx2 -mfma optimal_c_vector_product.c -o
optimal_c_vector_product
```

SIMD指令集优化

- 使用SSE2指令集加速整数计算
- 使用AVX/AVX2指令集加速浮点计算
- 每次处理4个数据元素，实现并行计算

```
// 使用SIMD指令的整数向量点积 (SSE2)
long long dot_product_int_simd(int *a, int *b, int length)
{
    __m128i sum_vec = _mm_setzero_si128(); // 初始化向量寄存器为0
    long long sum = 0;
    int i;
```

```
    // 使用SIMD指令并行处理4个整数
    for (i = 0; i <= length - 4; i += 4)
    {
        __m128i a_vec = _mm_loadu_si128((__m128i *)&a[i]);
        __m128i b_vec = _mm_loadu_si128((__m128i *)&b[i]);

        // SSE中没有直接的整数乘法累加，使用专用指令完成
        __m128i mul_lo = _mm_mullo_epi32(a_vec, b_vec);
        sum_vec = _mm_add_epi32(sum_vec, mul_lo);
    }

    // 将向量寄存器中的4个整数相加
    int result[4] ALIGN;
    _mm_store_si128((__m128i *)result, sum_vec);

    for (int j = 0; j < 4; j++)
    {
        sum += result[j];
    }

    // 处理剩余的元素
    for (; i < length; i++)
    {
        sum += a[i] * b[i];
    }

    return sum;
}
```

多线程并行计算

- 利用OpenMP实现多核并行处理
- 使用reduction子句避免线程间竞争

```
// 多线程优化的整数向量点积
long long dot_product_int_parallel(int *a, int *b, int length)
{
    long long sum = 0;

#pragma omp parallel reduction(+ : sum)
    {
#pragma omp for schedule(static)
        for (int i = 0; i < length; i++)
        {
            sum += (long long)a[i] * b[i];
        }
    }

    return sum;
}
```

循环优化

- 8倍循环展开减少循环控制开销
- 预先计算边界条件以减少分支判断

```
// 循环展开优化的整数向量点积
long long dot_product_int_unrolled(int *a, int *b, int length)
{
    long long sum = 0;
    int i;

    // 8倍循环展开，减少循环开销
    for (i = 0; i <= length - 8; i += 8)
    {
        sum += (long long)a[i] * b[i] +
               (long long)a[i + 1] * b[i + 1] +
               (long long)a[i + 2] * b[i + 2] +
               (long long)a[i + 3] * b[i + 3] +
               (long long)a[i + 4] * b[i + 4] +
               (long long)a[i + 5] * b[i + 5] +
               (long long)a[i + 6] * b[i + 6] +
               (long long)a[i + 7] * b[i + 7];
    }

    // 处理剩余的元素
    for (; i < length; i++)
    {
        sum += (long long)a[i] * b[i];
    }

    return sum;
}
```

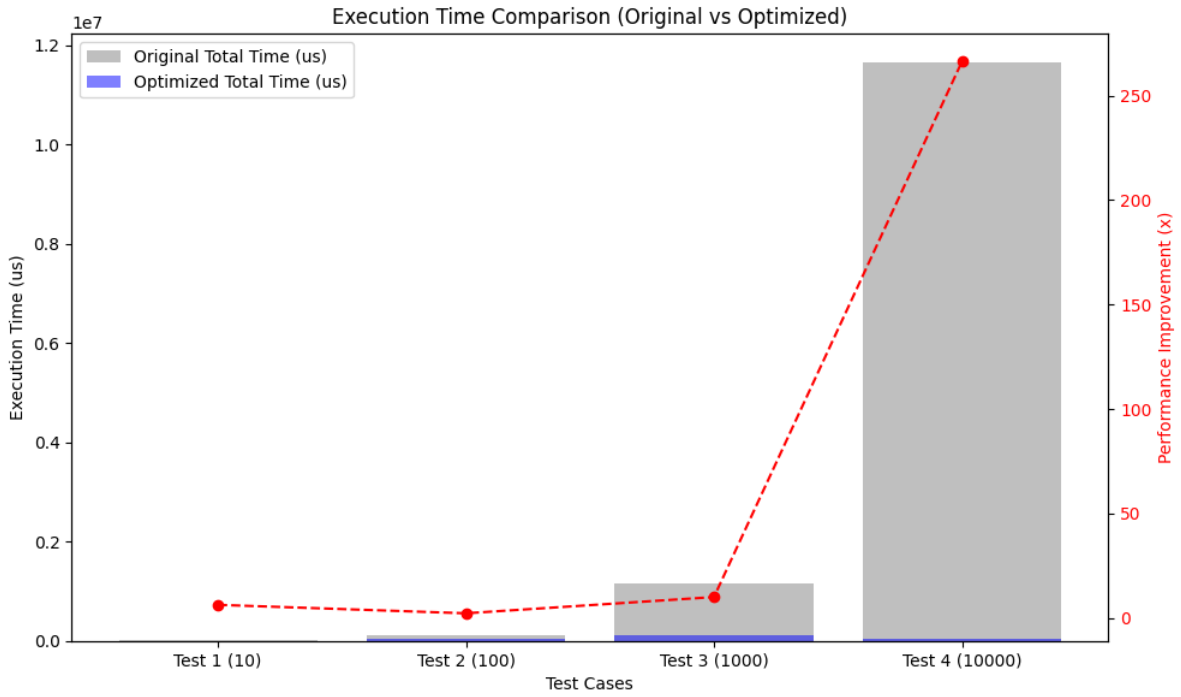
内存优化

- 内存对齐以优化SIMD加载/存储操作
- 使用posix_memalign确保32字节对齐
- 复用已分配内存，避免重复分配

```
// 内存对齐宏，优化内存访问
#define ALIGN __attribute__((aligned(32)))
#define MAX_VECTOR_SIZE 1000000
```

Test Case	Vector Length	Original Total Time (us)	Optimized Total Time (us)	Performance Improvement (x)
Test 1	10	16105	2543	6.33
Test 2	100	110398	48029	2.29

Test Case	Vector Length	Original Total Time (us)	Optimized Total Time (us)	Performance Improvement (x)
Test 3	1000	1161239	115203	10.07
Test 4	10000	11641687	43721	266.27



结论

优化后相对于原来能够有200倍的性能提升，并且vectorsize越大越明显，上一个section中size变大，java和c的差距减小，但是通过优化，可以进一步扩大java和c的速度差异，弥补了java内置虚拟机的优化差异，总之良好的c语言对于java有十分显著的速度优势

11.不足

1. 使用字符串代表数据类型，使得比较中有很少的时间是不同语言对于加工字符串的问题，但主要还是在比较两种语言对于计算方面的差异。
2. 可以完善两种语言的基本特性和实现方式，例如垃圾回收、内存管理机制、和内存使用等。

12. 结论

c在实现上相比java有极为显著的优势

但是c很底层，虽然已经很注意，但是还是有很多意向不到的错误，debug是面向内存的，这真的好痛苦啊，而且特别耗时间

相比之下java很友好，而且运行的时间差距在面对数据量较小时可以接受

btw I love program

I am the master of artificial intelligence

引用

Wikipedia contributors. (n.d.). *GraalVM*. Wikipedia. Retrieved March 30, 2025, from https://en.wikipedia.org/wiki/GraalVM

Stack Overflow user. (n.d.). *Profiling - Java performance tips*. Stack Overflow. Retrieved March 30, 2025, from https://stackoverflow.com/questions/938683/java-performance-tips

GitHub. (n.d.). *Copilot*. GitHub. Retrieved March 30, 2025, from https://github.com/settings/copilot

OpenAI. (n.d.). *ChatGPT*. OpenAI. Retrieved March 30, 2025, from https://chatgpt.com/