

编译原理 Lab1: 语言认知实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期: 2023 年 2 月 27 日

摘 要

本文为北京理工大学《编译原理与设计 2023》课程的 Lab1 实验报告。随着计算机领域的快速发展,多种高级语言进入主流市场。高级语言编写的程序是编译器的输入,了解它们的特性有助于更好的了解不同的编译程序及编译方式。本实验通过编写 C++、Java、Python、Haskell 和 x86 汇编语言版本的 300×300 的矩阵乘法,比较不同程序的编程效率和运行效率的差异性,进一步了解不同语言的特性、执行方式等等,为之后的课程学习打下良好的基础。

1 简介与准备工作

本节将在1.2给出 C++、Java、Python、Haskell、masm32 语言的简介。因为本实验需要编写矩阵乘法代码,故下文还将介绍朴素矩阵乘法算法。由于我的个人电脑之前没有 Haskell 的执行环境,

1.1 实验简介

根据实验文档 [2] 要求,本实验中主要任务为使用 C++、Java、Python、Haskell、masm32 五种语言完成矩阵乘法运算。

具体地,先编写数据生成程序,生成两个由 $1 \sim 100$ 的整数组成的规模为 300×300 的矩阵 A 和 B,生成文件 *A.txt* 和 *B.txt*。

在主程序中,先分别从 *A.txt* 和 *B.txt* 中读入两个 300×300 的矩阵。再通过基本的语句实现两矩阵的相乘,结果为矩阵 C。最后将矩阵 C 存入文件 *C_xxx.txt*,其中 xxx 为所用的语言名称。

为了比较不同程序之间的运行效率且尽可能排除更多因素的干扰,仅对矩阵相乘这一过程进行计时。为比较不同语言之间的易用性,在编写代码时不刻意压行和进行多余的换行。

1.2 语言简介

在本次实验中，我们用到了以下几种语言进行程序的编写：

1. C++: 编译型高级语言，支持面向过程、面向对象，命令式，静态弱类型。本实验中使用的 C++ 编译器为 C++14（代号为 201402）。
2. Java: 面向对象语言，通过将源代码编译成字节码，再使用 JVM 解释执行，命令式，静态强类型。本实验中使用的 java 版本为 jdk19.0.2。
3. Python: 解释型语言，动态强类型。本实验中的 Python 版本为 Python3.9。
4. Haskell: 函数式、声明式语言，静态强类型。其 GHC (Glasgow Haskell Compiler) 提供了编译和解释 (GHCi) 执行两种运行方式。本实验中使用的 GHC 版本为 9.2.5。
5. x86 汇编: x86 汇编是一种基于 Intel x86 CPU 体系结构的汇编语言，它使用的指令集为 CISC (Complex Instruction Set Computer)。本实验中我们使用 masm32 汇编器进行汇编程序的编写，运用了头文件 *irvine32.inc*，其定义的宏、过程函数的调用方法参考了 *Assembly Language for Intel-Based Computers* [1]。

1.3 朴素矩阵乘法算法

对于朴素的矩阵乘法，若有两个 $n \times m$ 、 $m \times k$ 的矩阵相乘，则算法时间复杂度为 $\mathcal{O}(nmk)$ ，具体的原理即为线性代数的矩阵乘法，见下图。

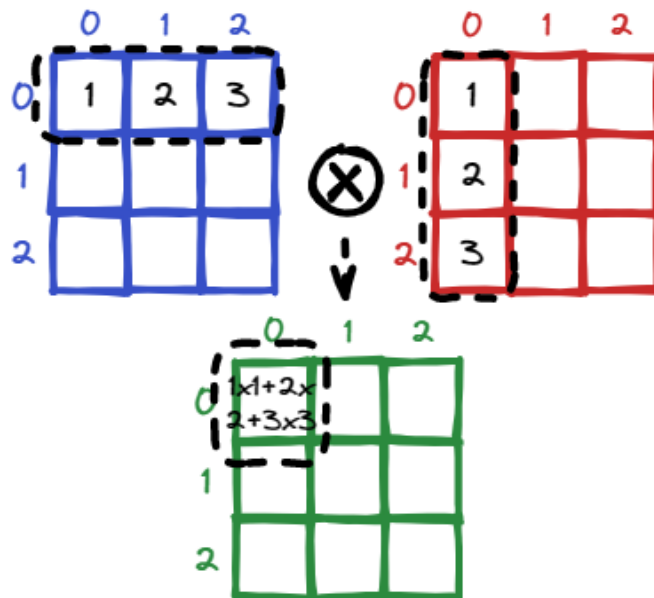


图 1: 矩阵乘法基本原理

据此，我们可以写出朴素矩阵乘法算法的伪代码。

Algorithm 1 朴素矩阵乘法算法**Input:** matrix A,B**Output:** matrix C = AB

```

1: for  $i = 1$  to  $n$  do
2:   for  $j = 1$  to  $n$  do
3:     for  $k = 1$  to  $n$  do
4:        $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$ 
5:     end for
6:   end for
7: end for
8: return matrix C

```

1.4 环境介绍与配置

1.4.1 实验硬件配置

实验时所用的计算机操作系统为 Windows 10 家庭中文版 64 位，四种高级语言的代码编辑器为 Microsoft VS Code，x86 汇编语言所用 IDE 为 Microsoft Visual Studio。实验机的硬件配置如下所示。

表 1: 计算机硬件配置

CPU 核数	CPU 主频	内存	Cache 大小
6	2.60GHz	16384MB	13.88MB

1.4.2 Haskell 环境配置

由于之前没有接触过 Haskell 语言，需要现配 Haskell 环境，并将其配置到 VS Code 中。

在 Windows 系统上的环境配置参考了博客《Windows10 下 VS Code 配置 Haskell 语言环境》[3]，采用安装 Stack 来管理和构建 Haskell 项目的依赖项。

首先在 [haskellstack 官网](#) 下载并安装 stack。再在 [stackage 官网](#) 找到适配版本为 9.2.5 的 GHC 的版本为 LTS20.11 的 Stackage。Stackage 提供和维护了很多不同版本的稳定兼容的 Haskell 包集，可以让我们在创建、运行项目时避免不同包不兼容、冲突等情况出现。最后通过命令 `stack config set resolver lts-20.11` 进行 GHC 的安装。在完成安装后，我们可以通过 `stack ghc` 进行程序的编译，或者通过 `stack ghci` 打开交互式解释器。

配置 haskell 环境到 VS Code。在 VS code 和 stack 两端分别安装扩展插件 ghc-

mod、hlint，并在 VS Code 中的用户配置文件中添加扩展的配置，在全局配置文件中添加 GHC 配置。至此，完成所有环境配置。

2 代码实现

2.1 算法实现

2.1.1 C++ 实现

C++ 的实现即用三重 for 循环，动态分配 MatC 的空间，具体代码如下：

```
int** MatC = new int*[numRowsC];
for(int i = 0; i < numRowsC; i++){
    MatC[i] = new int[numColsC];
    for(int j = 0; j < numColsC; j++){
        MatC[i][j] = 0;
        for(int k = 0; k < numColsA; k++){
            MatC[i][j] += MatA[i][k] * MatB[k][j];
        }
    }
}
```

2.1.2 Java 实现

Java 的实现与 C++ 类似。

```
int[][] MatC = new int[numRowsC][numColsC];
for(int i = 0; i < 300; i++){
    for(int j = 0; j < 300; j++){
        MatC[i][j] = 0;
        for(int k = 0; k < 300; k++){
            MatC[i][j] += MatA[i][k] * MatB[k][j];
        }
    }
}
```

2.1.3 Python 实现

python 中使用二维 List 存储矩阵，矩阵计算时逐行、逐列进行枚举，相乘得到每个结果矩阵的元素的值。

```
MatC = [[
    sum(a * b for a, b in zip(row_a, col_b))
    for col_b in zip(*MatB)] for row_a in MatA]
```

2.1.4 Haskell 实现

由于 Haskell 语言是声明式语言，所以我们需要编写 `mat_Mul` 函数。具体地，首先我学习并使用了 Haskell 运算符 `!!` 进行函数内容的编写，但因为线性索引查找符号每次都需要从头开始遍历寻找对应索引的数，所以会使程序运行速度大大减缓对运行效率影响较大，故后改用 `zipWith` 函数，其在编译过程中进行了优化。

```
mat_Mul :: Matrix -> Matrix -> Matrix
mat_Mul a b =
    let bt = transpose b
        n = length bt
    in [[sum $ zipWith (*) ar bc | bc <- bt] | ar <- a]
-- in [[sum [ar !! k * bc !! k | k <- [0..n-1]] | bc <- bt] | ar <- a]
```

2.1.5 x86 汇编实现

我们按照 C++ 版的代码逐一进行转换，不难得到 x86 汇编语言的矩阵相乘代码。

```
matMul PROC USES eax ebx edx ecx esi
    LOCAL i: DWORD, j: DWORD, k: DWORD, index: DWORD,
           sum: DWORD, p1: DWORD, p2: DWORD
    mov i, 0
    mov j, 0
    mov k, 0
    mov index, 0
    mov sum, 0
    mov p1, 0
    mov p2, 0
    mov esi, index
mulbegin:
    mov eax, i
    cmp eax, numRowsA
    jae quit
L1: mov eax, j
    cmp eax, numColsB
```

```
    mov sum, 0          ;sum=0
    jae plusJ
L2:  mov eax, k
    cmp eax, numColsA
    jae plusK
    mov eax, i
    mov ebx, numColsA
    mul ebx
    add eax, k
    mov eax, MatA[eax*4]
    mov p1, eax         ;p1=A[i][k]
    mov eax, k
    mov ebx, numColsB
    mul ebx
    add eax, j
    mov eax, MatB[eax*4]
    mov p2, eax         ;p2=B[k][j]
    mov eax, p1
    mov ebx, p2
    imul ebx
    add sum, eax        ;sum+=p1*p2
    add k, 1
    jmp L2
plusK:
    mov k, 0
    add j, 1
    mov eax, sum
    mov MatC[esi*4], eax
    inc esi
    jmp L1
plusJ:
    mov j, 0
    add i, 1
    jmp mulbegin
quit:
    ret
```

```
matMul ENDP
```

2.2 文件读写

因为本实验需要对不同语言所编写的相同功能的代码进行运行时间的比较，数据规模较大，故我们可以使用文件读写的方式进行输入输出，操作简便并节约时间，具体的，我们从 *A.txt* 中读入矩阵 **A**，从 *B.txt* 中读入矩阵 **B**，进行矩阵乘法后得到矩阵 **C**，并将矩阵 **C** 写入文件 *C.txt* 中。附录A给出不同语言的具体实施方案。

2.3 其他部分

2.3.1 随机数据生成

编写一个 C++ 代码，设置随机种子，生成两个大小为 300×300 的、每个元素为 $[0, 100]$ 的整数的矩阵，分别写入文件 *A.txt* 和 *B.txt*，代码如下：

```
int main() {
    srand(time(0));
    int numRows, numCol;
    numRows = numCol = 300;

    ofstream file_OPA("A.txt");
    for(int i = 0; i < numRows; i ++){
        for(int j = 0; j < numCol; j ++){
            file_OPA << rand() % 100 + 1 << " ";
        }
        file_OPA << endl;
    }

    ofstream file_OPB("B.txt");
    for(int i = 0; i < numRows; i ++){
        for(int j = 0; j < numCol; j ++){
            file_OPB << rand() % 100 + 1 << " ";
        }
        file_OPB << endl;
    }

    file_OPA.close();
```

```
file_OPB.close();  
}
```

2.3.2 运行时间计时

由于我们需要对不同语言下的程序进行性能分析，故需要对核心算法部分进行计时。具体的各语言版本的计时语句可见附录B。需要注意的是，由于 Haskell 语言是惰性语言 [2]，它会延迟求值，所以在 `mat_Mul` 函数前后计时只能得到一个接近 0ms 的时间。改正方法可以在输出结果后再计时，或者为了减小误差可以使用 `evaluate` 函数后计时。

3 实验结果与分析

3.1 运行过程

五种语言的运行截图见附录C

3.2 语言易用性和程序规模对比分析

3.2.1 语言易用性分析

从学习难度上来看，由于我学习 C++ 和 python 时间较长，所以对于我而言编写本次 C++ 和 Python 代码显然是很简单的。又由于本次实验用到的 java 语法和 C++ 语法比较相似，所以学习、编写难度也较低。汇编代码易于上手，但 Haskell 语言在思维上对我而言比较颠覆。

从编程效率上，这次 Haskell 语言给我造成了很大的困难。首先，我之前所接触过的编程语言基本都是命令式的，而非像 Haskell 一样是声明式的、编写函数式的，同时很多常用的函数我需要从零学习，大概花了一整天的时间才能编写本次实验代码。

对我而言更困难的是汇编。虽然上学期的汇编接口课写了不少汇编代码，但是文件输入输出还是让我不知所措。又由于有关的文献资料较少，在我花了大量的时间后才逐渐了解 `Irvine32.lib` 和其中的一些过程和宏的功能和使用方法。编写代码时 debug 十分费劲，大概耗时 20h 才完成汇编代码的编写。

综上所述，对我而言语言易用性 $C++ = Python > Java > Haskell = asm$ 。

3.2.2 程序规模对比分析

下面以代码行数为度量单位对程序规模进行对比分析。在没有压行的情况下，各语言实现矩阵乘法部分的代码行数如下所示。

表 2: 不同语言核心代码行数

C++	Java	Python	Haskell	x86
10	10	1	9	55

其中 Python 用了比较"pythonic"的风格, 所以只有一行。从上表可以看出, 四种高级语言的程序规模相差不大, 但是汇编语言作为更接近硬件的语言实现起来就比较复杂, 程序规模较大。

最后再谈一下 Haskell, 作为函数式编程, 如果我们需要完成的任务有现成的函数或者函数的组合就能完成, 那么码量是很小的; 但是一旦有一些比较细小的“定制化”任务需要手写, 那么 Haskell 语言会比较费劲。

3.3 运行效率对比

表 3: 不同语言矩阵乘法运行时间

次数	C++	Java	Python	Haskell		x86	次数	C++	Java	Python	Haskell		x86
				编译	解释						编译	解释	
1	108ms	48ms	2.51s	1.15s	1.47s	74ms	6	95ms	48ms	2.35s	1.22s	1.58s	73ms
2	95ms	42ms	2.48s	1.16s	1.56s	73ms	7	96ms	48ms	2.41s	1.16s	1.77s	72ms
3	97ms	38ms	2.41s	1.19s	1.63s	72ms	8	96ms	42ms	2.37s	1.26s	1.67s	84ms
4	98ms	39ms	2.40s	1.16s	1.40s	72ms	9	96ms	49ms	2.52s	1.17s	1.55s	77ms
5	106ms	47ms	2.51s	1.18s	1.37s	71ms	10	106ms	44ms	2.49s	1.22s	1.60s	72ms

对上面 10 组实验的运行时间分别求平均, 如下表:

表 4: 不同语言运行平均时间

C++	Java	Python	Haskell		x86
			编译	解释	
99.3ms	44.5ms	2.45s	1.18s	1.57s	74.0ms

由上表可以看出, 运行最快的是 Java 程序, 其次是汇编语言和 C++ 程序。Haskell 的解释运行比编译运行略快一些, 最慢的为 Python 程序。

这与我的预期略有不符, 因为从抽象层次高低来看, 更底层的汇编语言应该比高级语言更快, 然后对于高级语言而言 C++ 应比 Java 快一些。究其原因, 可能是因为 Java 的编译器优化能力比 g++ 更强, 所以在简单的程序中体现不出 C++ 程序的性能优势。另一方面, x86 语言中用了一些伪指令和一些由 C 编写的函数接口, 可能会使速度变慢。

综上所述，汇编程序的运行效率比高级语言更高；编译型语言的运行效率比解释型语言更高。

4 实验心得体会

这次试验虽然是一个“认知”实验，但是却让我耗时 5 天时间完成。从编写 C++、Java、Python 程序的顺风顺水，到 Haskell 的环境配置和语言学习，再到痛苦的汇编的编写和调试，已经三者的经历也让我在这次实验中学到了很多东西，尤其是“速成”了 Java 和 Haskell。

从环境配置上，原来刚学习 python 的时候配置 anaconda 并部署到 vs code 上让我十分痛苦，而且当时一头雾水的我把各种环境、包管理的十分混乱，而有了一些经验后再去配置 java 环境和 haskell 环境就轻松了许多，在这次的环境配置中比较顺利。

虽然实现的代码内容十分简单，但是通过这个平平无奇的代码也带我领略了不同类、不同级语言在实现过程、运行过程中的不一样。被汇编折磨了一天一夜后也让我更加“珍惜”如今市场上广泛使用的高级语言。而作为高级语言和机器语言之间的“桥梁”，汇编器的“神秘”也吸引着我接下来的课程中学习、了解它。

参考文献

- [1] Kip R. Irvine. *Assembly Language for Intel-Based Computers*. 2014. URL: <https://csc.csudh.edu/mmccullough/asm/help/>.
- [2] Lab1-程序设计语言认知实验. zh. 2023.
- [3] Windows10 下 VS Code 配置 Haskell 语言环境. zh. 2022. URL: <https://blog.csdn.net/myRealization/article/details/118878150>.

A 不同语言文件输入输出的实现

A.1 C++ 实现

添加头文件 `fstream`，对根据 `filepath` 得到的文件创建 `ofstream` 类对象，通过插入和提取运算符完成对文件的读、写，具体代码如下：

```
ifstream fileA("A.txt");    // 创建 ifstream 类
fileA >> MatA[i][j];        // 读入矩阵
ofstream fileC("C.txt");    // 创建 ofstream 类
fileC << MatC[i][j] << " "; // 写出矩阵
```

A.2 Java 实现

在 Java 中，通过对给定路径名对应的文件创建一个 `File` 类对象，并对这个 `File` 类对象创建一个 `Scanner` 类对象，用于实现文件的读入；使用 `PrintStream` 类创建一个输出流，并将 `FileOutputStream` 类创建的输出流作为参数传入，实现文件的写入。具体代码如下：

```
// 创建 Scanner 类对象
Scanner fileA = new Scanner(new File("A.txt"));
MatA[i][j] = fileA.nextInt();
// 创建输出流对象
PrintStream printS = new PrintStream(new FileOutputStream("C_java.txt"));
printS.print(MatC[i][j] + " ");
fileA.close()
printS.close()
```

A.3 Python 实现

Python 的文件打开、关闭实现较为简单，通过 `split()` 函数可以得到一个每个数作为一个字符串的字符串列表，`map` 函数和 `list` 进一步将其转化成二维的列表。

```
# 读入矩阵
with open('A.txt', 'r') as file:
    lines = file.readlines()
    MatA = [list(map(int, line.split())) for line in lines]
# 写出矩阵 C
with open('C_py.txt', 'w') as file:
```

```
for row in MatC:
    file.write(' '.join(map(str, row)) + '\n')
```

A.4 Haskell 实现

Haskell 从文件读入的函数需要一个类型为 `FilePath` 的参数，并返回一个类型为 `IO Matrix` (`Matrix` 为我定义的两为 `Int` 类型的列表类型) 的数据。

通过自带的 `readFile` 函数将文件内容导入，`lines` 函数对内容按行切割得到行数个字符串，`words` 函数将如 "1 2 3 4" 的字符串按空格切分成 ["1", "2", "3", "4"]，最后 `map read` 函数将字符串转换成整数，即转为 [1, 2, 3, 4]。

将结果写入文件的操作类似上面的逆向操作。`map show` 函数将整数转成字符串，`unwords` 函数将每行的字符串以空格连接起来，`unlines` 函数将不同行的字符串以换行符链接起来，最后由 `writeFile` 函数写入相应的文件。

```
-- 读取文件转成矩阵
readMatrix :: FilePath -> IO Matrix
readMatrix file = do
    contents <- readFile file
    return (map (map read . words) (lines contents))

-- 将矩阵写入文件
writeMatrix :: FilePath -> Matrix -> IO ()
writeMatrix file matrix =
    writeFile file (unlines (map (unwords . map show) matrix))
```

A.5 x86 汇编实现

因为导入了 `Irvine32.lib`，故根据使用文档 [1] 使用到了其中几个函数，较为频繁使用的函数极其功能如下：

- `ReadFromFile`: 将文件内容读入缓冲区。EAX 为打开文件句柄，EDX 为输入缓冲区的地址，ECX 为最大读入字节。返回是否读入成功。
- `OpenInputFile`: 将文件打开。EDX 为文件名的地址。返回 EAX 为打开文件的句柄。
- `closeFile`: 关闭文件。EAX 为打开文件句柄。

由于汇编实现较为麻烦且将结果写入文件不必须，故我没有编写写入文件的代码。

```
;-----
; function name: GetMyFile
```

```
; description: 将 txt 文件内容以长字符串形式读入 tableBuffer, 用 ASCII 码存
; Params: 1) PtrFilename 文件名指针
;-----
GetMyFile PROC USES edx eax ecx,
PtrFilename: PTR DWORD
LOCAL FileHandle: DWORD ; 句柄
mov edx, PtrFilename
call OpenInputFile
mov FileHandle, eax
mov edx, OFFSET tableBuffer
mov ecx, DWORD PTR BufferSize
call ReadFromFile
mov eax, FileHandle
call closeFile
ret
GetMyFile ENDP

;-----
; function name: ParseMat
; description: 将 tableBuffer 中的 ASCII 码转换为十进制数
; Params: 1) PtrMat: 矩阵指针 2) numRows: 矩阵行数指针 3) numCols: 矩阵列数指针
;-----
ParseMat PROC USES eax esi edx ebx ecx,
PtrMat: DWORD,
numRows: DWORD,
numCols: DWORD
LOCAL index: DWORD, tmpRow: DWORD, tmpCol: DWORD,
digit: DWORD, sum: DWORD
mov index, 0
mov tmpRow, 0
mov tmpCol, 0
mov esi, 0
.while tableBuffer[esi] != 0
    .if tableBuffer[esi] == 20h ; 空格
        .if tmpRow < 1
            add tmpCol, 1
```

```
.endif
inc esi
.elseif tableBuffer[esi] == 0dh ;CL(回车)
add tmpRow, 1
add esi, 2^^I; 跳过换行
.else ; 找数
mov sum, 0
mov digit, 0
.while tableBuffer[esi] >= 30h && tableBuffer[esi] <= 39h
mov edx, 0h
movzx eax, tableBuffer[esi]
mov ecx, 30h
div ecx
mov digit, edx
mov eax, sum
mov ebx, 10
mul ebx
mov sum, eax
mov edx, digit
add sum, edx
inc esi
.ENDW
mov ecx, PtrMat
add ecx, index
mov eax, sum
mov [ecx], eax
add index, 4
.if index == 360000 ; 读完跳出
inc tmpRow
jmp jpl
.endif
.ENDIF
.ENDW
jpl:
mov eax, tmpCol
mov esi, numCols
```

```
mov [esi], eax
mov eax, tmpRow
mov esi, numRows
mov [esi], eax
ret
ParseMat ENDP
```

B 不同语言计时的实现

B.1 C++ 实现

include 头文件 `ctime`, 使用 `time()` 函数获得当前时间, 最后差值除以 `CLOCKS_PER_SEC`。

```
clock_t st = clock();
clock_t ed = clock();
cout << "total running time: " <<
    double(ed - st) / CLOCKS_PER_SEC << "sec" << endl;
```

B.2 Java 实现

通过 `System.currentTimeMillis()` 函数获取当前时间, 单位为毫秒。

```
double st = System.currentTimeMillis();
double ed = System.currentTimeMillis();
System.out.println("total running time: " + totalTime/1000.0 + "s");
```

B.3 Python 实现

直接导入 `time` 包获取当前时间即可。

```
st = time.time()
ed = time.time()
print("total running time: %.3fs" % (ed - st))
```

B.4 Haskell 实现

调用 `getTime` 函数即可, 注意此时获取的时间单位为纳秒。

```
st <- getTime Monotonic
ed <- getTime Monotonic
```

```
let tot_t = fromIntegral(toNanoSecs $ diffTimeSpec ed st) / (10 ^ 9)
putStrLn $ "total running time: " ++ show tot_t ++ " s"
```

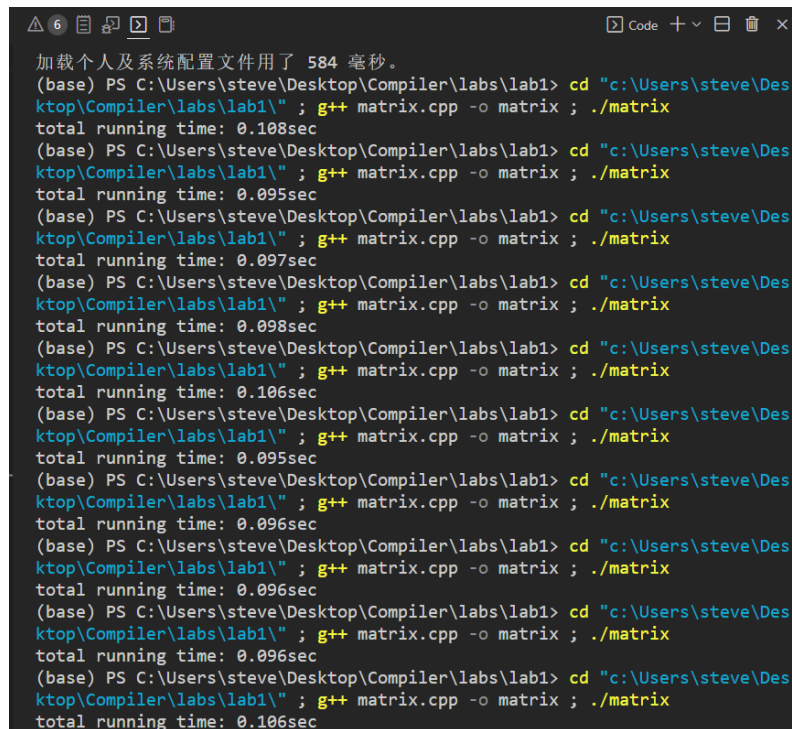
B.5 x86 实现

调用 GetMSeconds 过程, 返回 EAX 为 $EAX = ((hours * 3600) + (minutes * 60) + (seconds)) * 1000 + milliseconds$ 。

调用过程 WriteString 输出字符串, 调用 WriteDec 过程输出运行时间。

```
invoke GetMSeconds
mov startTime, eax
invoke GetMSeconds
sub eax, startTime
mov startTime, eax
mov edx, OFFSET msg1
call WriteString
mov edx, startTime
call WriteDec
mov edx, OFFSET msg2
call WriteString
```


C 运行过程截图

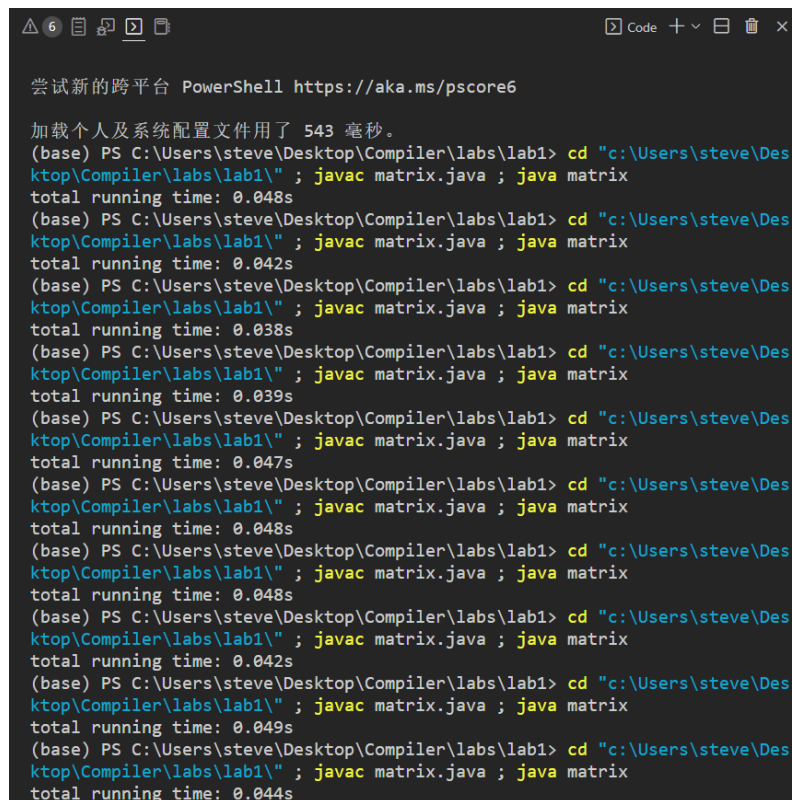


```

加载个人及系统配置文件用了 584 毫秒。
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.108sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.095sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.097sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.098sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.106sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.095sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.096sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.096sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.096sec
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; g++ matrix.cpp -o matrix ; ./matrix
total running time: 0.106sec

```

图 2: C++ 代码运行截图



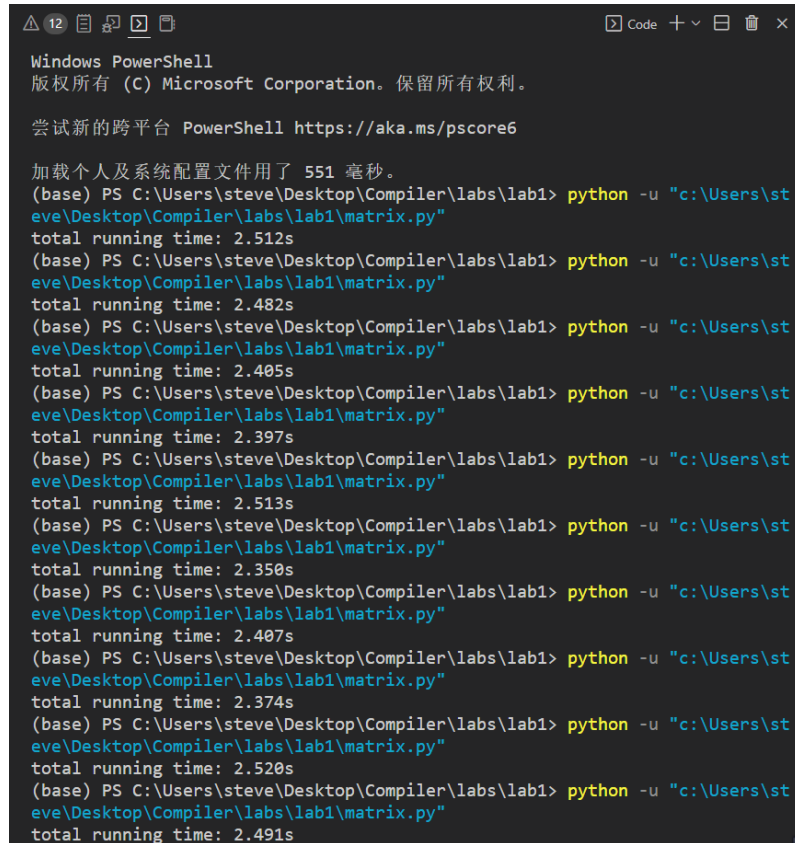
```

尝试新的跨平台 PowerShell https://aka.ms/pscore6

加载个人及系统配置文件用了 543 毫秒。
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.048s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.042s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.038s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.039s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.047s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.048s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.048s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.042s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.049s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Users\steve\Desktop\Compiler\labs\lab1\" ; javac matrix.java ; java matrix
total running time: 0.044s

```

图 3: Java 代码运行截图

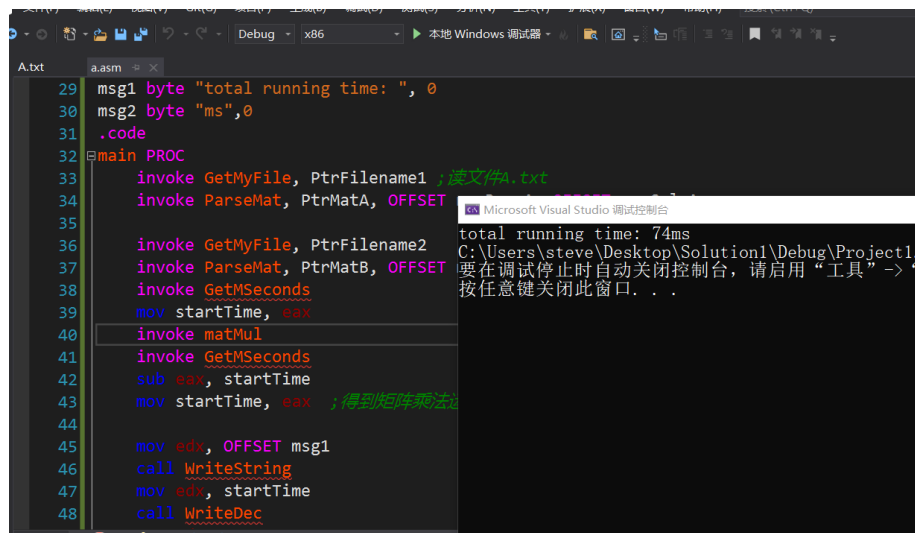


```
Windows PowerShell
版权所有 (c) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

加载个人及系统配置文件用了 551 毫秒。
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.512s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.482s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.405s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.397s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.513s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.350s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.407s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.374s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.520s
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> python -u "c:\Users\steve\Desktop\Compiler\labs\lab1\matrix.py"
total running time: 2.491s
```

图 4: Python 代码运行截图



```
A.txt
a.asm
29 msg1 byte "total running time: ", 0
30 msg2 byte "ms", 0
31 .code
32 main PROC
33     invoke GetMyFile, PtrFilename1 ; 读文件A.txt
34     invoke ParseMat, PtrMatA, OFFSET
35
36     invoke GetMyFile, PtrFilename2
37     invoke ParseMat, PtrMatB, OFFSET
38     invoke GetMSeconds
39     mov startTime, eax
40     invoke matMul
41     invoke GetMSeconds
42     sub eax, startTime
43     mov startTime, eax ; 得到矩阵乘法
44
45     mov edx, OFFSET msg1
46     call WriteString
47     mov edx, startTime
48     call WriteDec

Microsoft Visual Studio 调试控制台
total running time: 74ms
C:\Users\steve\Desktop\Solution1\Debug\Project1
要在调试停止时自动关闭控制台，请启用“工具”->“
按任意键关闭此窗口。...
```

图 5: x86 汇编代码运行截图



```
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
ell haskell.hs }  
total running time: 40.4750703 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 40.3563996 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 40.1883738 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 42.174806 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 42.5846366 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 42.2147277 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 42.3744375 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 42.5990241 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }  
total running time: 40.8389895 s  
(base) PS C:\Users\steve\Desktop\Compiler\labs\lab1> cd "c:\Use  
rs\steve\Desktop\Compiler\labs\lab1\" ; if ($?) { stack runhask  
ell haskell.hs }
```

图 6: Haskell 代码运行截图