



进阶实验篇第2章

C矩阵乘法

1

性能分析：计算操作vs执行时间

1

原因：Python的解释执行机制

2

矩阵乘法的C语言实现

3

C的编译执行机制

4

编译器优化

5

性能分析



性能分析：计算操作vs执行时间

- 回顾一下Python的矩阵乘法：

```
1. for i in range(rows of matrix A):  
2.     for j in range(cols of matrix B):  
3.         for k in range(cols of matrix A or cols of matrix B):  
4.             C[i][j] += A[i][k] * B[k][j]
```

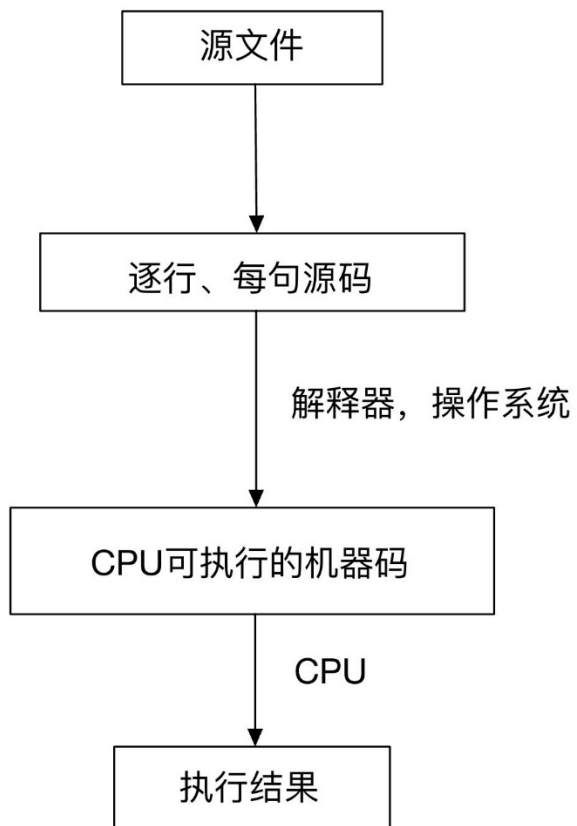
- 核心的运算在最后一行代码。
- 读3次, $C[i][j]$, $A[i][k]$, $B[k][j]$.
- 乘1次, $A[i][k]*B[k][j]$.
- 加1次, $C[i][j] + A[i][k]*B[k][j]$.
- 写1次, $C[i][j]$.
- 若是 $m*k$ 的矩阵与 $k*n$ 的矩阵相乘
- 这样的操作要重复 $m*n*k$ 次。

- 取加法消耗10周期，乘法20周期，读和写都是50周期
- CPU频率为2995.199MHz，一周期为0.00000000033s
- 令 $n \times n$ 的矩阵与 $n \times n$ 的矩阵相乘， n 分别取1024，2048和4096

矩阵大小	读次数	写次数	乘次数	加次数	预计周期	预计时间	实际时间
1024	30亿	10亿	10亿	10亿	2300亿	75.90s	196.85s
2048	240亿	80亿	80亿	80亿	18400亿	607.20s	1868.21s

- 而指令所消耗的周期数在现实中要比我们估计的周期数小，可即便如此，实际运行的时间仍然要比预计的时间长出很多。
- 为什么？

原因：Python的解释执行机制



源代码


```
a = 2
b = 3
c = a + b
c = c * 4
print(c)
```

分析、编译
识别变量等

字节码

```
a = 2
b = 3
c = a + b
c = c * 4
print(c)
```

字节码

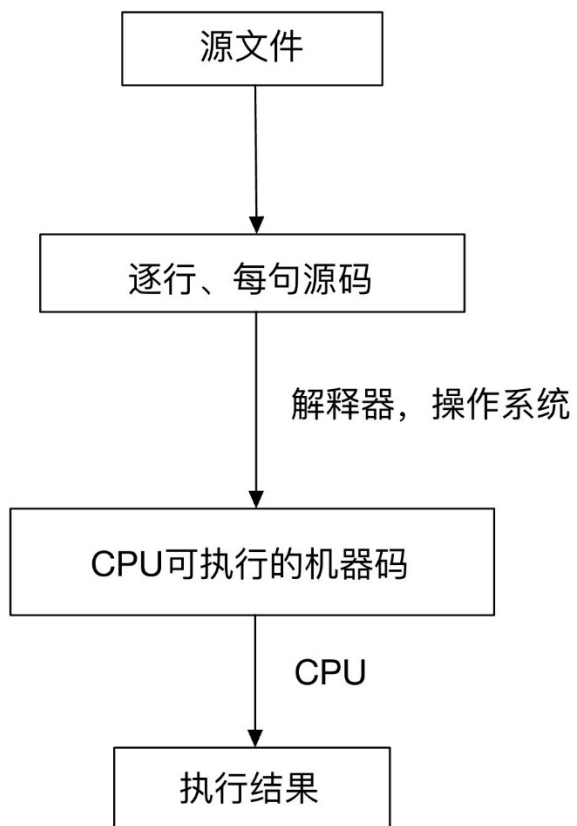


a = 2
b = 3
c = a + b
c = c * 4
print(c)

变量	值
a	2
b	3
c	5

未执行

已执行



- Python程序效率低的一个重要原因是其解释执行的机制。
- 解释器逐行逐句翻译源代码，翻译后的源码由操作系统翻译为目标CPU可执行的二进制机器码。
- CPU执行二进制机器码。
- 计时记录的时间不仅是CPU执行机器码的时间，还包括了解释器解释源码的时间。



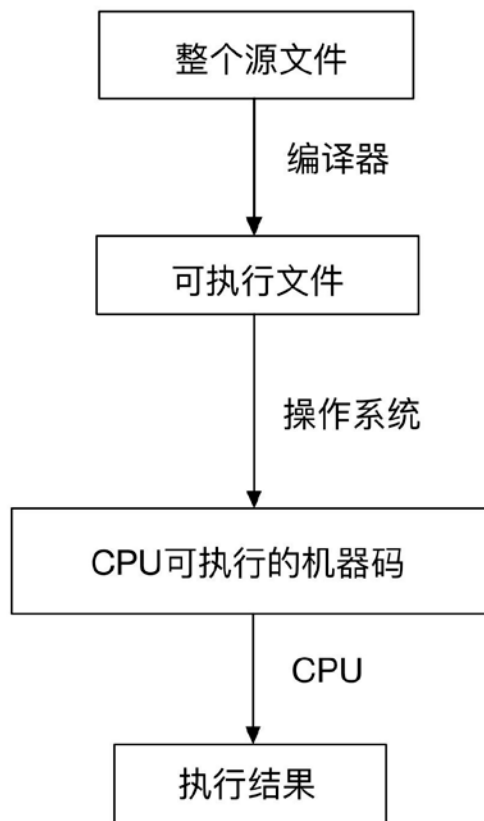
矩阵乘法的C语言实现



```
void matrix_mul(float* matrix_A, float* matrix_B, float* matrix_C, int m, int k, int n){  
    float(*A)[k] = (float(*)[k]) matrix_A;  
    float(*B)[n] = (float(*)[n]) matrix_B;  
    float(*C)[n] = (float(*)[n]) matrix_C;  
  
    memset(C, 0, m*n*sizeof(float));  
  
    int mi, ki, ni;  
    for(mi = 0; mi < m; mi++){  
        for(ni = 0; ni < n; ni++){  
            for(ki = 0; ki < k; ki++){  
                C[mi][ni] += A[mi][ki] * B[ki][ni];  
            }  
        }  
    }  
}
```

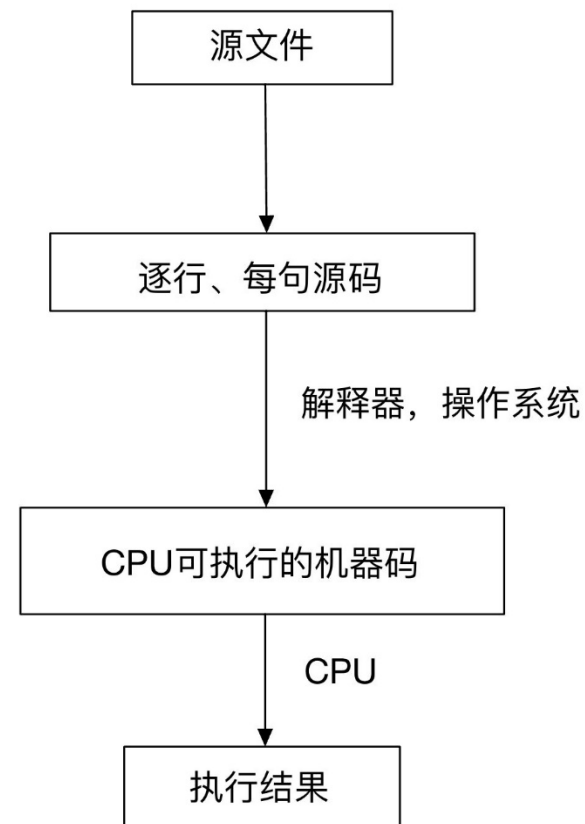
- C语言实现矩阵乘法的方式与Python实现完全一致，只是语言用法存在一定差异。
- C语言使用二维数组来存储二维矩阵。

C的编译执行机制

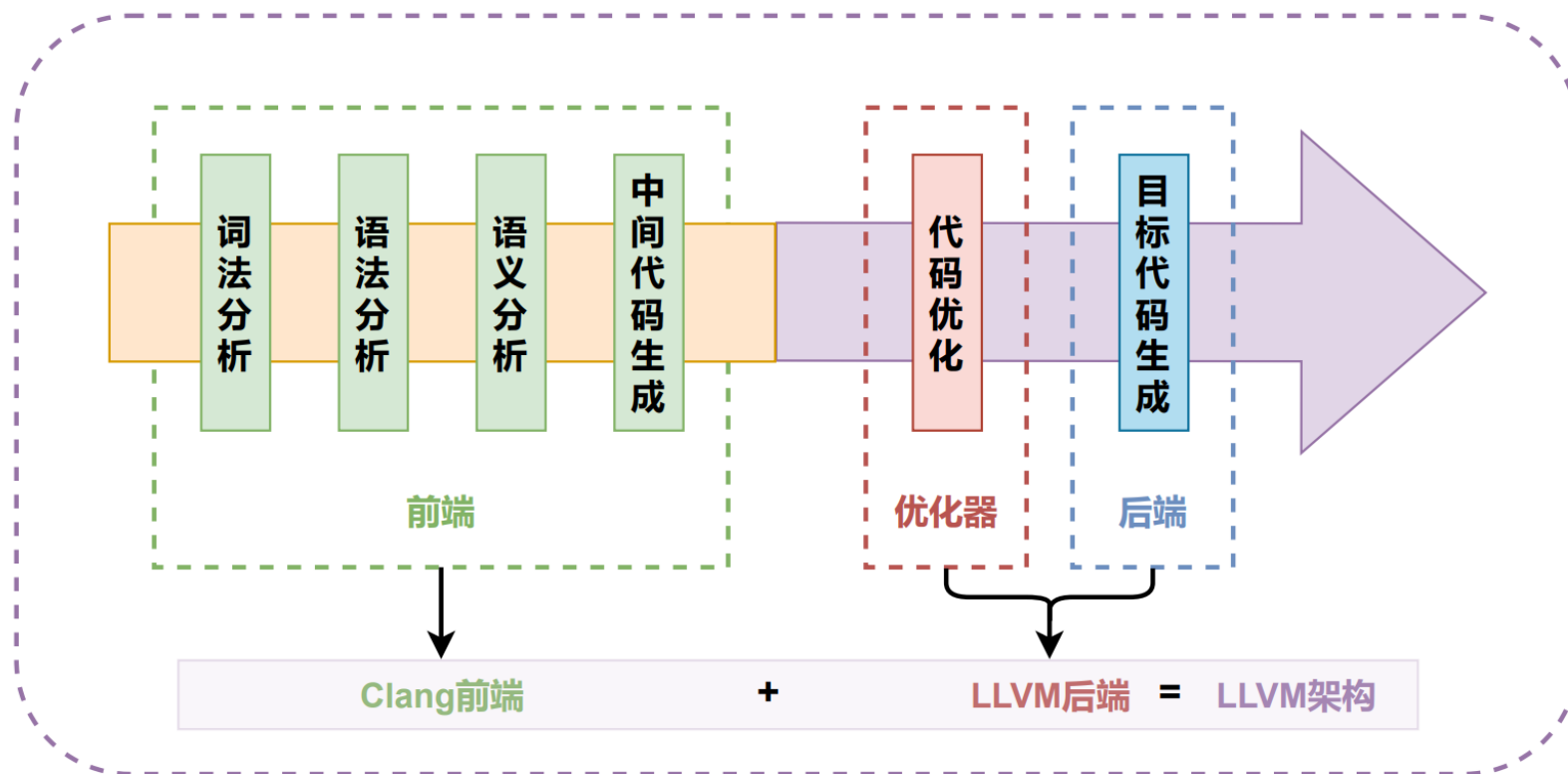


编译执行

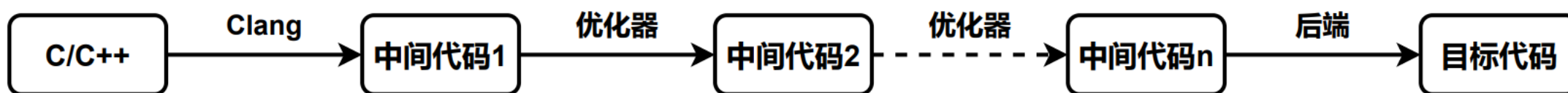
- 编译器直接将源文件编译为可执行文件。
- 相比于解释执行，编译执行下CPU可以更快得到需要执行的指令，效率更高。



解释执行

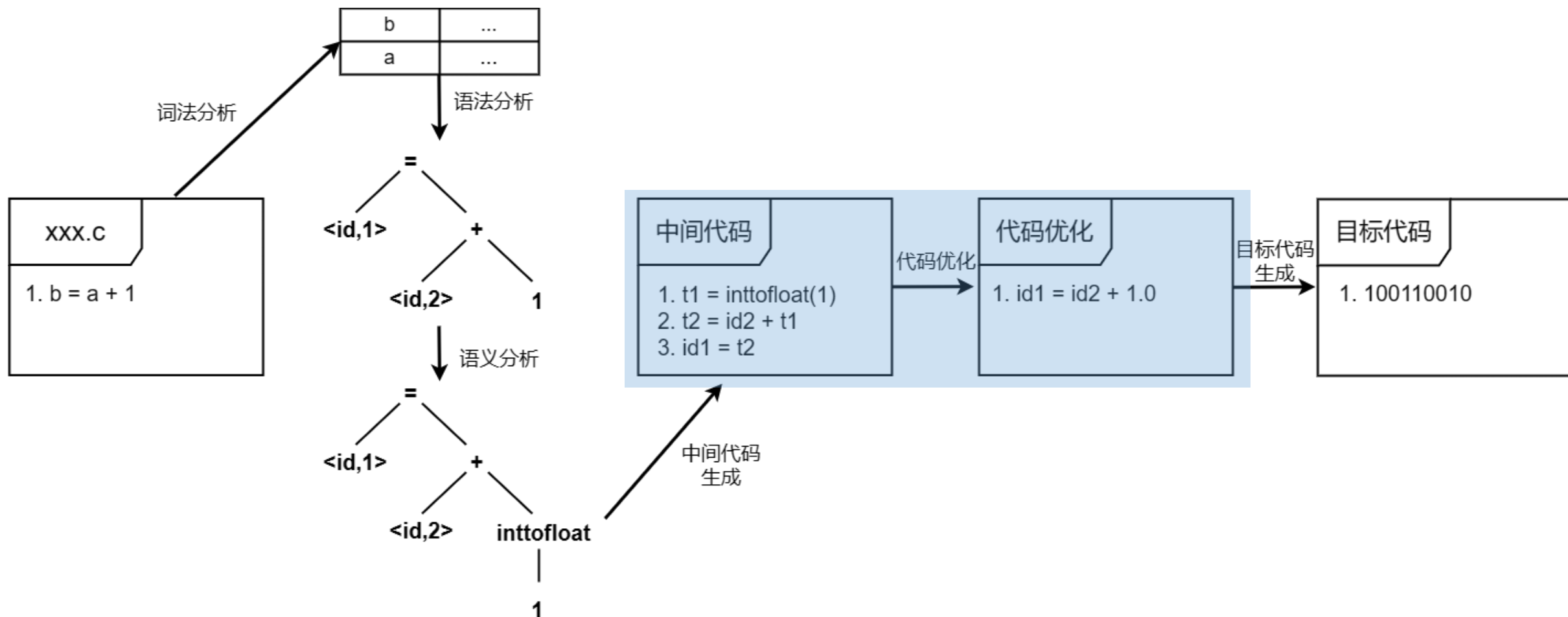


- 词法分析提取词素，识别变量、运算符等。
- 语法分析判断语法是否符合规范。
- 语义分析进行类型检查等任务。
- 中间代码生成将源代码使用中间代码表示。
- 代码优化在中间代码层面使代码更高效。
- 目标代码生成将中间代码翻译为目标机器代码。





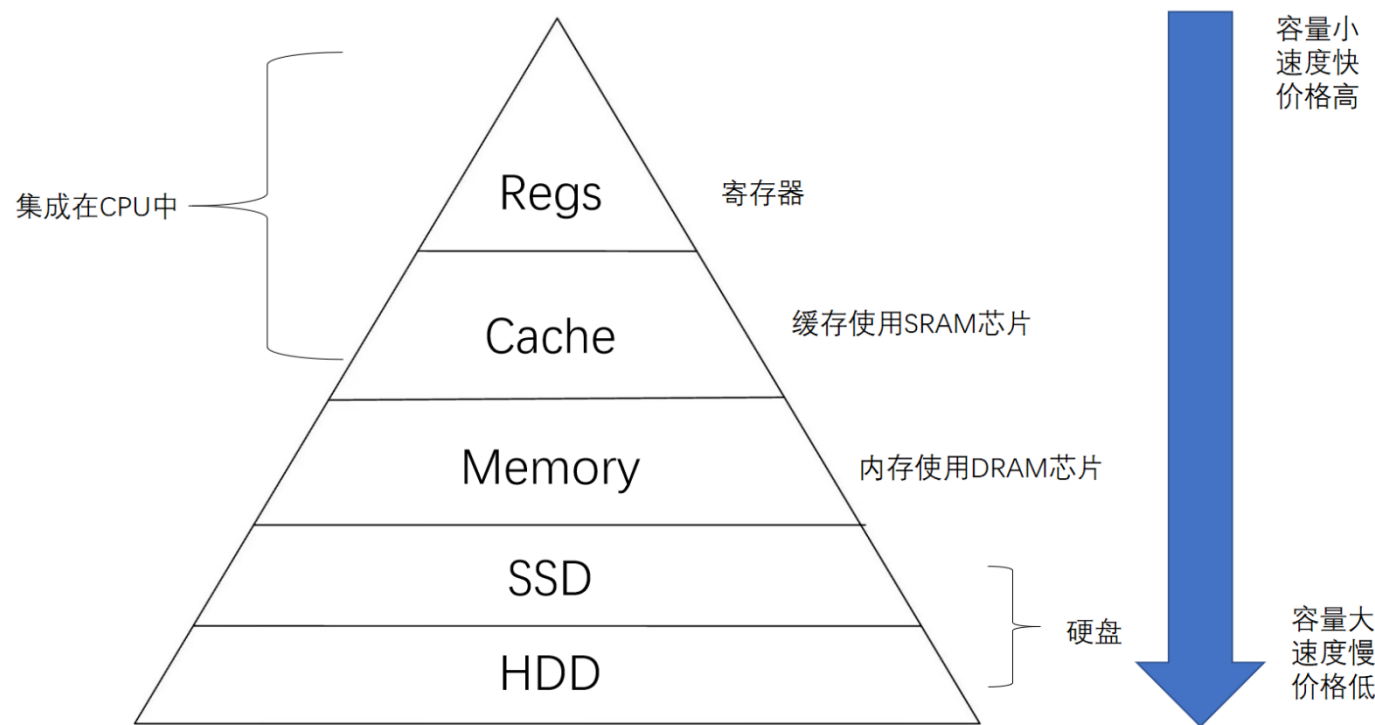
编译器优化



- 编译器编译源码会经过多个阶段，代码优化部分有助于生成高效的目标代码。

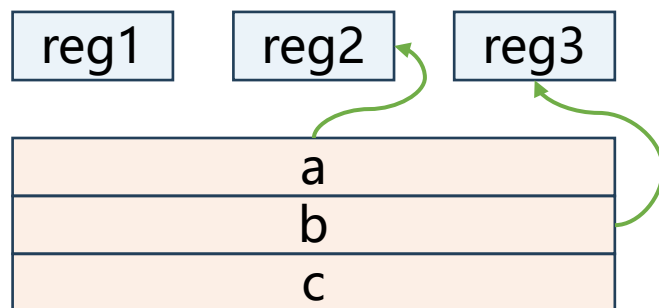
- 为何要编译?

1. CPU只能识别一些简单的机器指令，即01串
2. 高级语言对于CPU来说十分复杂，需要被分解为很多条简单的**汇编指令**，例如加法、乘法、取值、跳转等
3. 汇编指令最终会被翻译为机器指令对应的01串，传给CPU识别并执行。当然我们只讨论汇编，毕竟01串对于人来说很不友好，而且从语义上来讲，汇编指令与机器指令是一一对应的
4. 为保证高频率的CPU能够发挥出应有的性能，汇编指令所使用的**操作数**应被放到寄存器中

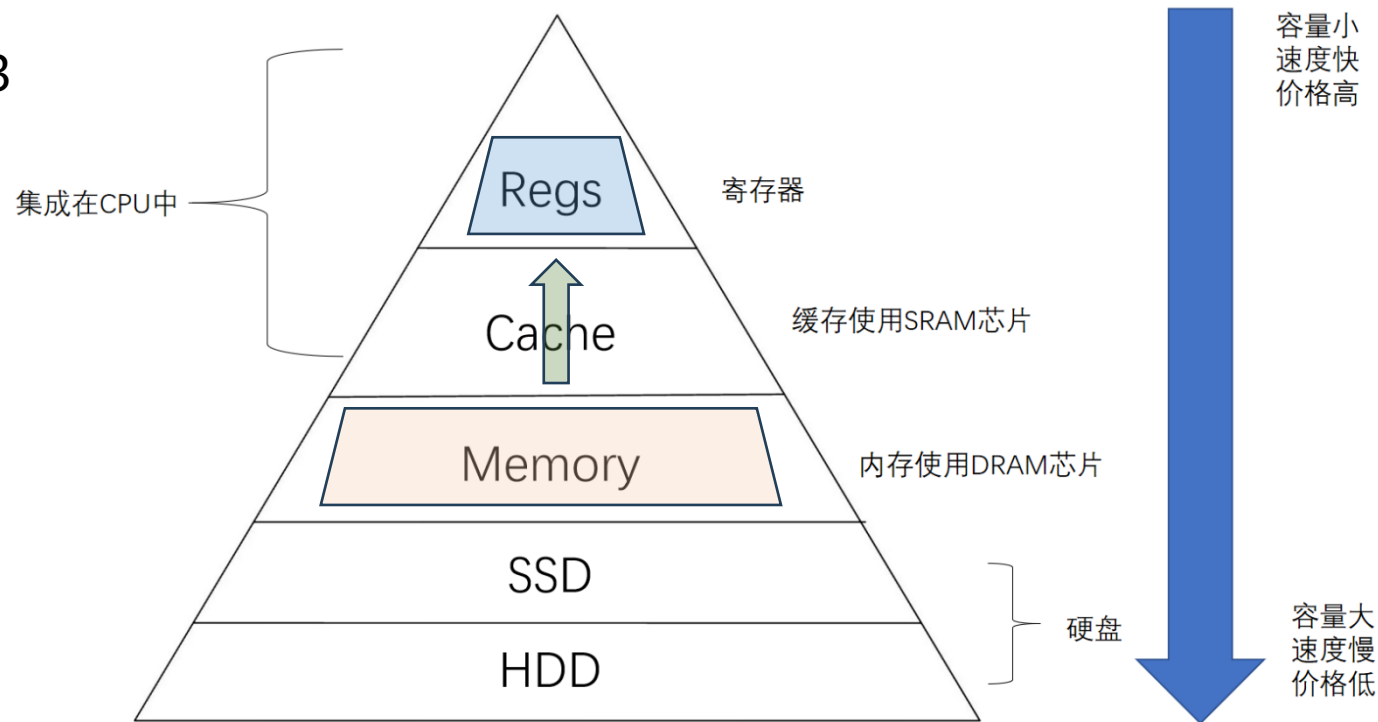


- 计算的前提：数据获取

`int c = a + b; <---> add reg1, reg2, reg3`

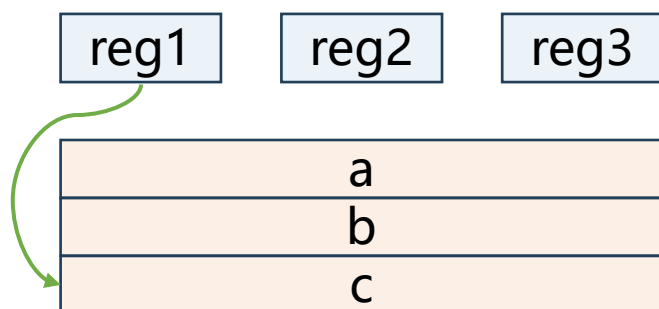


load reg2, &a
load reg3, &b

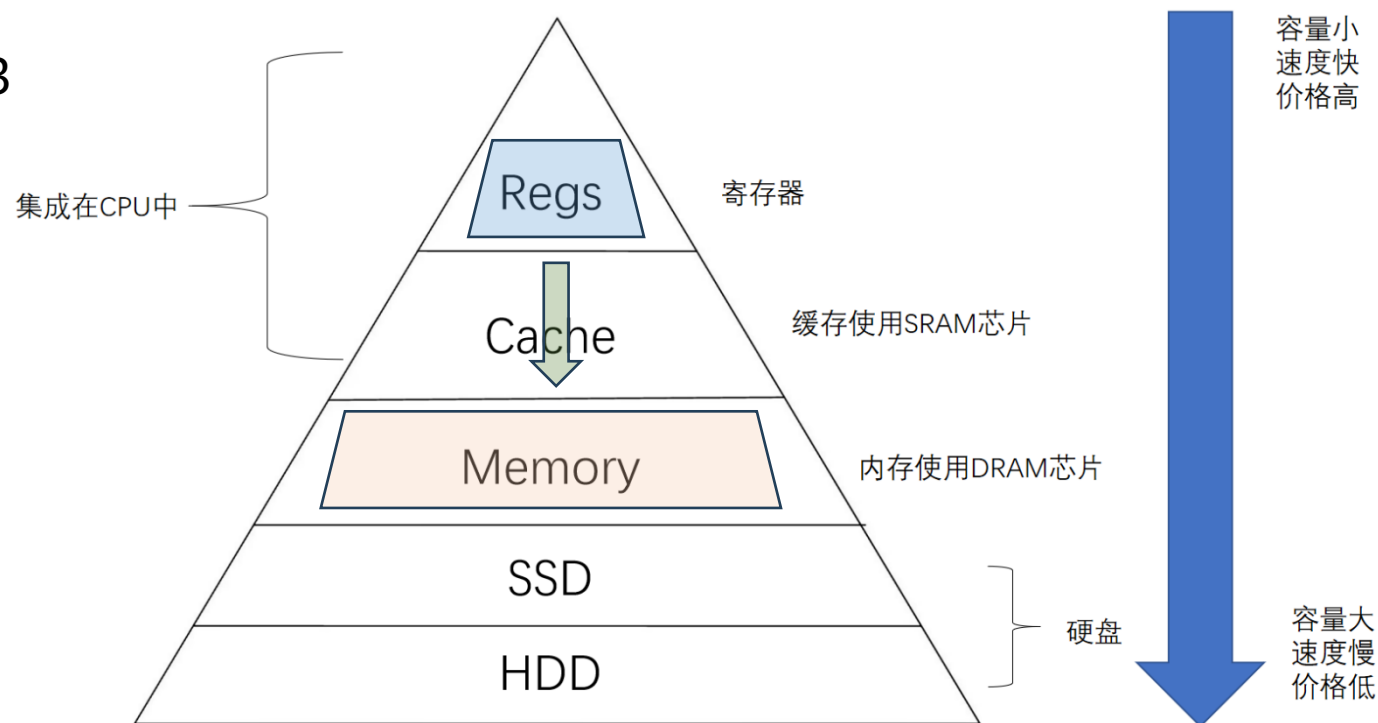


- 计算后的数据存储

`int c = a + b; <---> add reg1, reg2, reg3`



`store reg1, &c`



- 利用寄存器暂存，替代内存读写操作
- 识别出相同的计算结果，尽可能的利用数据以避免重复计算
- 识别一个计算或一组计算，为其选择合适的指令

- 汇编怎样实现循环?

```
void matrix_mul(float* matrix_A, float* matrix_B, float* matrix_C, int m, int
k, int n){
    float(*A)[k] = (float(*)[k]) matrix_A;
    float(*B)[n] = (float(*)[n]) matrix_B;
    float(*C)[n] = (float(*)[n]) matrix_C;

    memset(C, 0, m*n*sizeof(float));

    int mi, ki, ni;
    for(mi = 0; mi < m; mi++){
        for(ni = 0; ni < n; ni++){
            for(ki = 0; ki < k; ki++){
                C[mi][ni] += A[mi][ki] * B[ki][ni];
            }
        }
    }
}
```

- 循环变量的读取, 计算和存储


```
for(ki = 0; ki < k; ki++){  
    C[mi][ni] += A[mi][ki] * B[ki][ni];  
}
```

- 每次循环都需要load k, ki
- 但是似乎, 并不需要?

```
1. load reg1, &ki  
2. mov reg1, reg_zero  
3. store reg1, &ki  
4. Loop_begin:  
5.     load reg1, &ki  
6.     load reg2, &k  
7.     cmp reg1, reg2  
8.     beq Loop_end  
9.     #-----  
10.    C[mi][ni] += A[mi][ki] * B[ki][ni];  
11.    #-----  
12.    load reg1, &ki  
13.    add reg1, reg1, 1  
14.    store reg1, &ki  
15.    jmp Loop_begin  
16. Loop_end:
```

减少循环体中的load和store

```
1. load reg1, &ki
2. mov reg1, reg_zero
3. store reg1, &ki
4. Loop_begin:
5.     load reg1, &ki
6.     load reg2, &k
7.     cmp reg1, reg2
8.     beq Loop_end
9.     #-----
10.    C[mi][ni] += A[mi][ki] * B[ki][ni];
11.    #-----
12.    load reg1, &ki
13.    add reg1, reg1, 1
14.    store reg1, &ki
15.    jmp Loop_begin
16. Loop_end:
```



```
1. load reg1, &ki
2. mov reg1, reg_zero
3. store reg1, &ki
4. load reg2, &k
5. Loop_begin:
6.     cmp reg1, reg2
7.     beq Loop_end
8.     #-----
9.     C[mi][ni] += A[mi][ki] * B[ki][ni];
10.    #-----
11.    add reg1, reg1, 1
12.    jmp Loop_begin
13. Loop_end:
14.     store reg1, &ki
```

```
1. int i, j, k;  
2. for (i = 0; i < rows of matrix A; i++) {  
3.     for (j = 0; j < cols of matrix B; j++) {  
4.         for (k = 0; k < cols of matrix A or cols of matrix B; k++) {  
5.             C[i][j] += A[i][k] * B[k][j];  
6.         }  
7.     }  
8. }
```

- O0

- C[i][j]的变更涉及到一次读和一次写。
- C[i][j]的读写都需要地址。
- 读和写时各计算一次地址。

- O1

- C[i][j]的读和写使用同一地址。
- 读和写只计算一次地址。

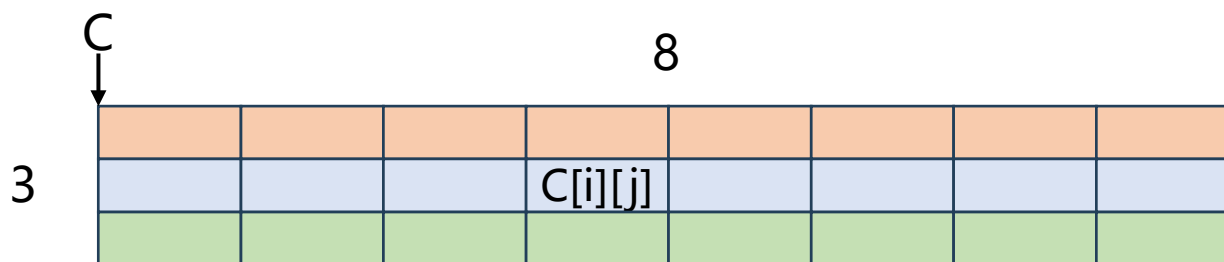
编译器优化：使用寄存器代替对同一个位置的读写以减少访存时间

```

1. int i, j, k;
2. for (i = 0; i < rows of matrix A; i++) {
3.     for (j = 0; j < cols of matrix B; j++) {
4.         for (k = 0; k < cols of matrix A or cols of matrix B; k++) {
5.             C[i][j] += A[i][k] * B[k][j];
6.         }
7.     }
8. }

```

- C[i][j]如何获取?



- 确定C[i][j]为数组中第几个元素:
 $\text{index} = i * 8 + j$
- 确定地址偏移offset:
 $\text{offset} = \text{index} * 4$
- 数组的首地址与offset相加得到C[i][j]地址:
 $\text{dst} = C + \text{offset}$


```
1. load reg1, &ki
2. mov reg1, reg_zero
3. store reg1, &ki
4. load reg2, &k
5. Loop_begin:
6.     cmp reg1, reg2
7.     beq Loop_end
8.     #-----
9.     C[mi][ni] += A[mi][ki] * B[ki][ni];
10.    #-----
11.    add reg1, 1
12.    jmp Loop_begin
13. Loop_end:
14.    store reg1, &ki
```

```
1. load reg1, &mi
2. load reg2, &ni
3. mul reg3, reg1, 8
4. add reg4, reg3, reg2
5. mul reg5, reg4, 4
6. add reg6, reg5, C
7. load reg7, reg6
```

循环体中每次获得C[mi][ni]都要算一次偏移，但是这个偏移每次都是一样的，造成很多重复计算。提到循环外就可以避免重复计算。

```
1. int i, j, k;
2. for (i = 0; i < rows of matrix A; i++) {
3.     for (j = 0; j < cols of matrix B; j++) {
4.         for (k = 0; k < cols of matrix A or cols of matrix B; k++) {
5.             C[i][j] += A[i][k] * B[k][j];
6.         }
7.     }
8. }
```

- 当C[i][j], A[i][k]和B[k][j]分别加载到reg1,reg2和reg3中后, 进行的运算应当如下:

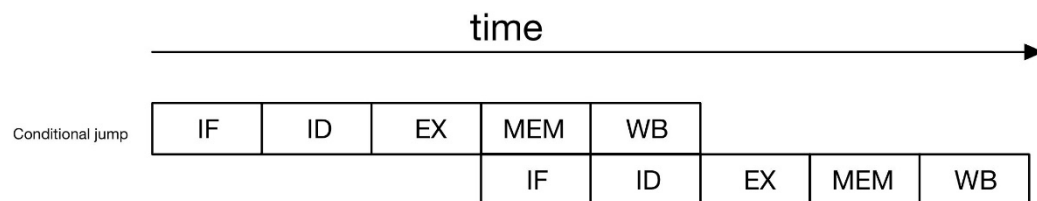
```
1. mul reg4, reg2, reg3  #reg4 <-- reg2*reg3
2. add reg1, reg1, reg4  #reg1 <-- reg1+reg4
```

- 当存在乘加指令时, 可以将两条指令合并在一起

循环展开前

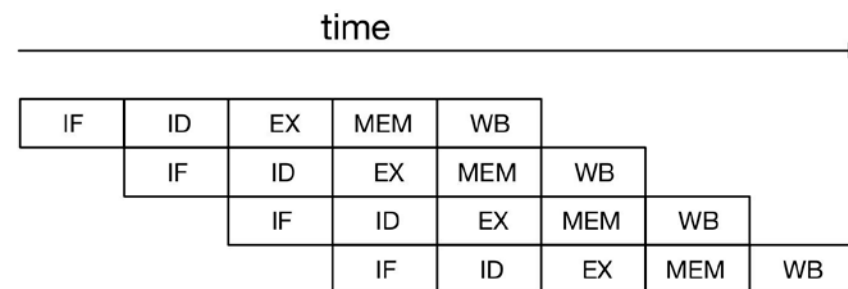
```
int i;
for(i = 0; i < loop_size; i++) {
    array[i]++;
}
```

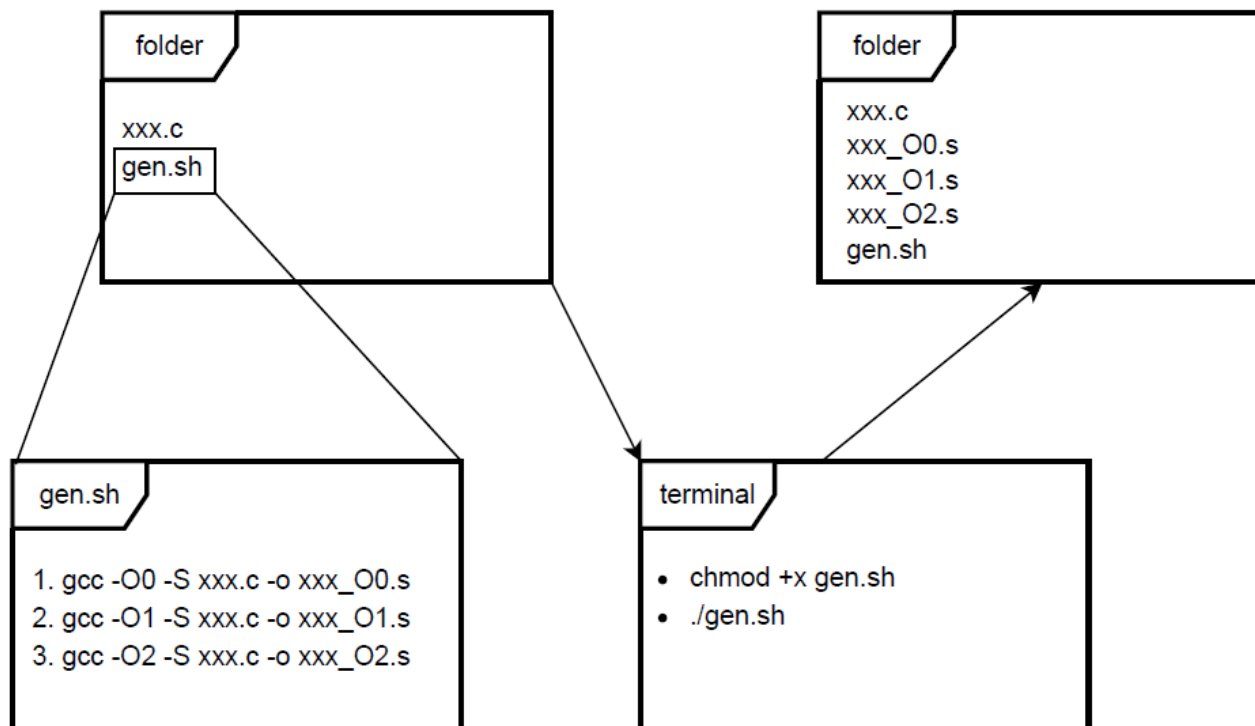
- 循环展开有助于流水线排布
- 除此之外编译器还有其他的优化.....



循环展开后

```
int i;
for(i = 0; i < loop_size; i+=4) {
    array[i+0]++;
    array[i+1]++;
    array[i+2]++;
    array[i+3]++;
}
```





- 通过gcc 命令进行编译。
- `gcc -O1 -S xxx.c -o xxx_O1.s`中:
 - O1表示优化等级为O1
 - S表示生成汇编文件
 - o用于设定生成文件名称
- `chmod +x gen.sh`赋予 `gen.sh`执行权限。

```
xxx_O0.s
.....
20. or      $r12,$r9, $r0
.....
26. st.w    $r12, $r22, -68
.....
79. ld.w    $r12, $r22, -68
80. blt     $r13, $r12, .L5
.....
```

```
xxx_O1.s
.....
21. addi.w   $r20, $r9, -1
.....
79. bne      $r20, $r13, .L4
.....
```

- \$r9存储一个参数，在O0下直接存放在栈中，\$r22为栈底指针每次循环都从栈中进行读取参数，与循环变量比较。
- 而O1优化下，该参数存放到\$r20寄存器中，每次循环可以直接进行比较，节省时间开销。

编译器优化：使用寄存器代替对同一个位置的读写以减少访存时间

xxx_O1.s

```
.....  
21. fmul.s    $f1, $f0, $f1  
22. fld.s     $f0, $r12, 0  
23. fadd.s    $f0, $f0, $f1  
.....
```

xxx_O2.s

```
.....  
21. fmadd.s   $f0, $f2, $f1, $f0  
.....
```

- O1优化下使用3条指令来完成矩阵乘法核心的乘后加操作。
- 而O2优化下，这3条指令可以用一条更高效的乘加指令替换，效率更高。

编译器优化：使用指令集中的特定指令代替普通指令的功能以减少指令条数

性能分析

```
1. # include<time.h>
2. ....
3. clock_t start, end;
4. start = clock();
5. mm_mul(A, B);
6. end = clock();
7. int result_clock = end - start;
```

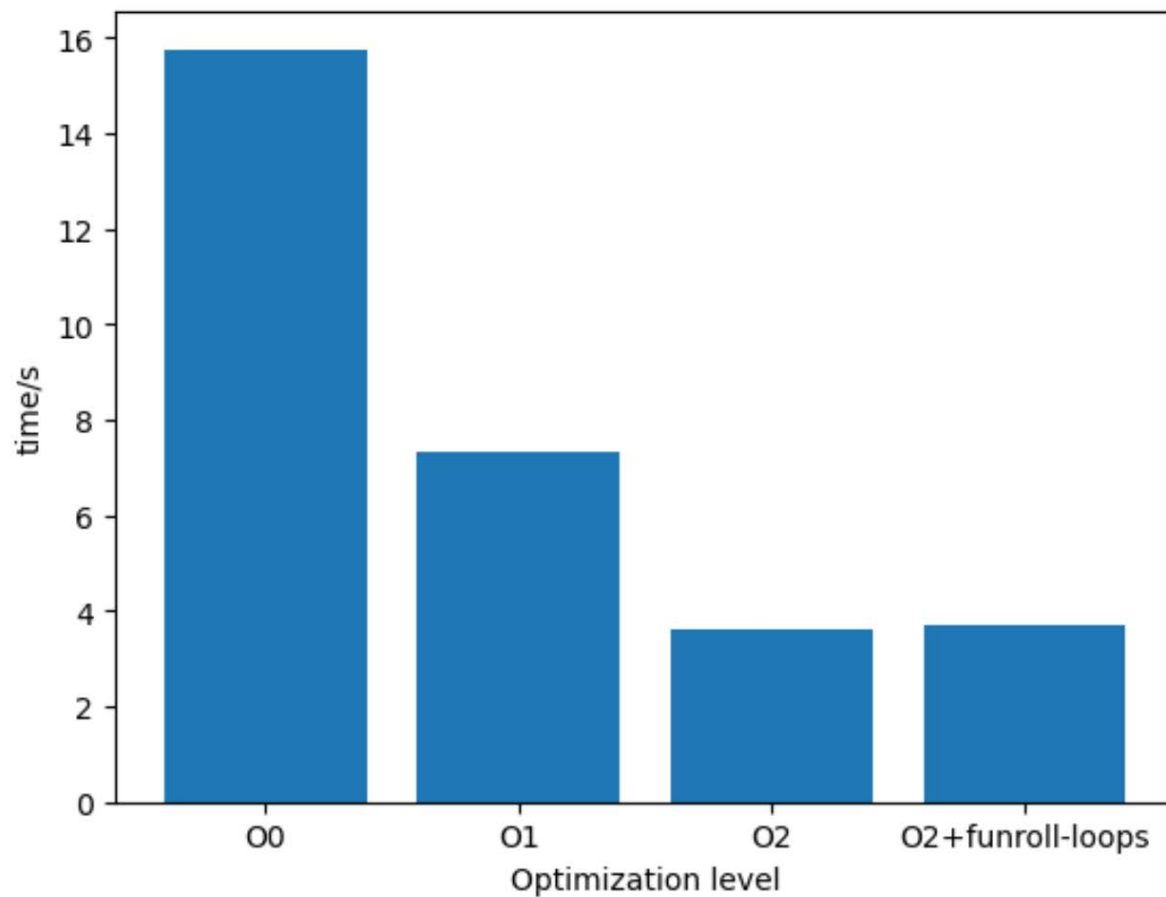
- C语言的计时方法与Python一致，使用time.h中提供的计时函数clock获取函数执行前后的时间，最后相减得到函数耗时。


```
1. # include<time.h>
2. ....
3. clock_t start, end;
4. start = clock();
5. // mm_mul(A, B);
6. end = clock();
7. int syscall_clock = end - start;
```

- 然而，只要是指令就都需要消耗时间来执行，计时函数的调用也不例外。
- 将待测函数注释掉进行测试，得到函数调用产生的耗时，这一部分时间应当被排除在矩阵乘法时间之外，以提高计时精度。

优化等级	O0	O1	O2	O2+funroll-loops
矩阵乘法耗时/s	15.75397774	7.32194929	3.63377837	3.71166412

- O2+funroll-loops选项强制开启循环展开，但对于现代支持乱序发射的CPU，循环展开会导致指令的乱序发射代价增多，从而影响性能。
- 可以粗略看到随着优化等级的提高，矩阵乘法性能越来越高。
- 但随着数据的增多，表格对数据的呈现将并不友好，尤其显示数值相近的数据时人眼分辨困难。



- 将数据绘制为表格，可以更清晰地比较数据之间的大小关系。
- 当对比一系列数据的相同指标时，可以使用柱状图进行更直观的表达。



本章作业

程序设计

利用C语言编程实现矩阵乘法

1. 利用C语言的循环语句（不要使用库函数）实现矩阵乘法
2. 测量不同尺寸的矩阵的乘法运算时间，并使用曲线呈现结果
3. 测量不同优化等级对于同一规模矩阵的运行时间影响，并使用柱状图呈现结果

观察并完成实验报告

1. 利用编译器生成汇编语言，观察不同的优化等级在指令生成上的差异
2. 观察不同的指令集和不同的编译器在代码生成和优化等级上的差异



南开大学
Nankai University

感谢阅读
