دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

عنوان:

بررسی روش‌های مختلف بهینه سازی در برنامه‌ها

استاد:

دکتر جهانگیر

دانشجو:

مرتضی اسعدی

شماره دانشجویی:

۹۷۲۱۲۲۷۸

زمستان ۱۳۹۷

فهرست مطالب

۱. کلیات

۱-۱. چکیده

در این گزارش سعی شده روش‌های مختلف بهینه سازی برنامه شامل تغییر الگوریتم و تغییر سطح زبان برنامه نویسی بررسی گردد و تاثیر هر روش بر روی سرعت اجرای برنامه و تسریع حاصل شده از هر تغییر محاسبه شود.

۱-۲. ابزار های استفاده شده

برنامه های نوشته شده به زبان Python با مفسر نسخه ۳٫۷٫۱ این زبان اجرا گردیده‌اند. برای کامپایل برنامه‌های نوشته شده به زبان C از مجموعه ابزارهای GCC نسخه ۸٫۲٫۱ ۲۰۱۸۱۱۲۷ استفاده شده است. برای مشاهده اطلاعات مرتبط با سیستمی که برنامه‌ها بر روی آن اجرا شده‌اند، از ابزار screenFetch نسخه ۳٫۸٫۰ استفاده شد.

۱-۳. مشخصات سیستم

برنامه‌های نوشته شده بر روی سیستم شخصی اینجانب(بهترین سیستم در دسترس) اجرا شده‌اند. در زیر مختصری از مشخصات این سیستم ذکر شده است.

```
mak@MAK
 OS: Arch Linux
 Kernel: x86_64 Linux 4.20.0-arch1-1-ARCH
 Uptime: 5h 6m
 Packages: 2936
 Shell: zsh 5.6.2
 Resolution: 1366x768
 WM: bspwm
 GTK Theme: NumixSolarizedDarkBlue [GTK2/3]
 Icon Theme: Papirus-Dark
 Font: Noto Sans Regular 10
 CPU: Intel Core i7-4700MQ @ 8x 3.4GHz [56°C]
 GPU: intel
 RAM: 6558MiB / 7887MiB
```

سیستم دارای پردازنده مرکزی intel core i7-4700MQ و حافظه اصلی ۷۸۸۷ مگابایت می باشد. برنامه‌ها بر روی سیستم عامل ArchLinux مبتنی بر GNU/Linux با هسته نسخه ۴٫۲۰٫۰ اجرا می شوند.

۱-۴. برنامه تحت بهینه سازی

می توان شبه کد مربوط به برنامه تحت بهینه سازی را به صورت زیر نمایش داد.

```
/**
* Input: Numbers a > 0, b > 0
* Output: gcd(a, b)
*/

function gcd(a, b)
   if a >= b then
      m = a
      n = b
   else
      m = b
      n = a

   while n != 0 do
      m = p*n + z
      m = n
      n = z
   return m
```

این برنامه دو عدد را گرفته و بزرگترین مقسوم علیه مشترک دو عدد را باز می‌گرداند. در پیاده سازی برنامه سعی شده است تا جای ممکن خطای سربار سیستم عامل کاهش یابد. برای این منظور اینجانب در هر مرحله برنامه را ۱۰۰۰۰۰ بار بر روی ۱۰۰۰۰۰ مجموعه عددی که به صورت شبه تصادفی تولید شده‌اند، اجرا کرده و نتایج را گزارش نموده‌ام.

۲. بهینه سازی برنامه زبان Python

۲–۱. برنامه اولیه

برنامه اولیه با استفاده از زبان Python به صورت زیر نوشته شده است:

```
import os
import time

def bytes_to_int(bytes):
    result = 0

    for b in bytes:
        result = result * 256 + int(b)

    return result
```

```
11.
12. def gcd(a,b):
13.     gcd = 1
14.     if (a == 0):
15.         return b
16.     if (b == 0):
17.         return a
18.
19.     # base case
20.     if (a == b):
21.         return a
22.
23.     for i in range(1,min(a,b)):
24.         if a%i==0 and b%i==0:
25.             gcd = i;
26.     return gcd
27.
28.
29. if __name__=="__main__":
30.     timingOfThousandRun = 0
31.     numbersToGCD = []
32.     numberOfRuns = 100000
33.     for i in range(0,numberOfRuns):
34.         numbersToGCD.append((bytes_to_int(os.urandom(1)),bytes_to_int(o
    s.urandom(1))))
35.     for i in numbersToGCD:
36.         t0 = time.process_time()
37.         gcdVal = gcd(i[0], i[1]);
38.         t1 = time.process_time()
39.         timingOfThousandRun += (t1-t0)
40.
41.     print("Program run for "+str(timingOfThousandRun)+" seconds.")
42.     print("Each GCD calculation took "+str(timingOfThousandRun/numberOf
    Runs)+" seconds.")
```

توجه نمایید که قسمت مورد اندازگیری زمانی قرار گرفته از برنامه، تابع gcd می باشد.

```
1. def gcd(a,b):
2.     gcd = 1
3.     if (a == 0):
4.         return b
5.     if (b == 0):
6.         return a
7.
8.     # base case
9.     if (a == b):
10.         return a
11.
12.     for i in range(1,min(a,b)):
13.         if a%i==0 and b%i==0:
```

```
14.             gcd = i;
15.     return gcd
```

بقیه بخش‌های برنامه برای تولید اعداد شبه تصادفی و اندازه گیری زمان اجرای تابع gcd مورد نیاز می باشند. در بخش‌های دیگر این گزارش از ارایه کل برنامه اجتناب کرده و فقط به نمایش تابع gcd بسنده خواهیم نمود.

۲-۲. استفاده از الگوریتم ۱

```
1. def gcd(a,b):
2.     if (a == 0):
3.         return b
4.     if (b == 0):
5.         return a
6.
7.     # base case
8.     if (a == b):
9.         return a
10.
11.     # a is greater
12.     if (a > b):
13.         return gcd(a-b, b)
14.     return gcd(a, b-a)
```

این الگوریتم با استفاده از فراخوانی تودرتو[1] gcd دو عدد را به دست می آورد. مشکل اصلی الگوریتم محدودیت تعداد دفعات فراخوانی تودرتو در زبان Python بر حسب نوع سیستم می باشد. مثلا برای سیستم اینجانب حداکثر میزان فراخوانی تودرتو یک تابع برابر ۲۹۰۵ مرتبه می باشد.

۲-۳. استفاده از الگوریتم ۲

```
1. def gcd(a,b):
2.     if a == 0 and b == 0:
3.         b = 1;
4.     elif b == 0:
5.         b = a;
6.     elif a != 0:
7.         while a != b :
8.             if a <b:
9.                 b -= a;
10.             else:
11.                 a -= b;
12.
13.     return b;
```

[1] Recursive

این الگوریتم مشکل الگوریتم فراخوانی تودرتو را نداشته و نسبت به الگوریتم اولیه نیز سریع تر اجرا می شود.

۴-۲. جدول تسریع برنامه های نوشته شده به زبان Pyhton

نتایج اجرای ۱۰ باره برنامه های فوق بر حسب میلی ثانیه در جدول زیر ذکر شده و تسریع الگوریتم ۱ و ۲ نسبت به برنامه اولیه محاسبه شده است.

جدول ۱- تسریع الگوریتم ۱ و۲ نسبت به برنامه اولیه

ردیف	برنامه اولیه	الگوریتم ۱		الگوریتم ۲	
	زمان اجرا	زمان اجرا	تسریع	زمان اجرا	تسریع
۱	۶۰۸	۵۴۷	۱٫۱۸	۲۴۴	۲٫۶۵
۲	۶۲۸	۵۵۰	۱٫۱۴	۲۷۰	۲٫۳۲
۳	۶۲۴	۵۴۲	۱٫۱۵	۲۵۸	۲٫۴۱
۴	۶۲۹	۵۴۸	۱٫۱۴	۲۲۰	۲٫۸۵
۵	۶۴۲	۵۳۹	۱٫۱۹	۲۶۸	۲٫۳۹
۶	۶۱۴	۵۵۲	۱٫۱۱	۲۵۷	۲٫۳۸
۷	۶۰۳	۵۴۷	۱٫۱۰	۲۶۲	۲٫۳۰
۸	۶۱۸	۵۴۳	۱٫۱۳	۲۴۷	۲٫۵۰
۹	۶۱۳	۵۵۱	۱٫۱۱	۲۵۰	۲٫۴۵
۱۰	۶۰۹	۵۵۰	۱٫۱۰	۲۴۰	۲٫۵۳
میانگین	۶۱۸	۵۴۶	۱٫۱۳	۲۵۱	۲٫۴۶

۳. استفاده از زبان C

می توان برنامه‌های نوشته شده به زبان C را به دو صورت استفاده از بهینه سازی کامپایلر و بدون استفاده از بهینه سازی کامپایلر پیاده سازی کرد. برای کامپایل برنامه بدون بهینه سازی از دستور:

```
1. $ gcc -O0 gcd.c -o gcd
```

و برای استفاده از حداکثر بهینه سازی از دستور:

```
1. $ gcc –O3 gcd.c -o gcd
```

استفاده نموده ایم. علاوه بر مطالب فوق از پارامتر S– کامپایلر gcc برای به دست آوردن کد اسمبلی برنامه نوشته شده استفاده نموده ایم.

برنامه نوشته شده به زبان C به صورت زیر می باشد:

```c
#include<stdio.h>
#include <stdlib.h>
#include<time.h>
#include <sys/time.h>

#define numberSets 100000

unsigned int gcd (unsigned int a, unsigned int b){
    if (a == 0 &&b == 0)
        b = 1;
    else if (b == 0)
        b = a;
    else if (a != 0)
        while (a != b)
            if (a <b)
                b -= a;
            else
                a -= b;

    return b;
}

int main(void) {

    unsigned char numbersToGCD[numberSets][2];
    double totalTime;
    time_t t;

    /* Intializes random number generator */
    srand((unsigned) time(&t));


    for(int i = 0 ; i < numberSets ; i++) {
        for(int j = 0 ; j<2; j++)
            numbersToGCD[i][j]= (unsigned char)(rand());
    }

    struct timeval stop, start;

    for(int i = 0 ; i < numberSets ; i++) {

        gettimeofday(&start, NULL);
        gcd(numbersToGCD[i][0],numbersToGCD[i][1]);
```

```
        gettimeofday(&stop, NULL);
        totalTime += (stop.tv_usec - start.tv_usec);
    }

    printf("%f\n",totalTime/CLOCKS_PER_SEC);

}
```

در تابع gcd از الگوریتم ۲ نوشته شده در بخش python برای پیاده سازی برنامه استفاده نموده ایم. توجه نمایید که در این برنامه نیز فقط زمان اجرای تابع gcd را اندازه گرفته شده و در محاسبه تسریع اعمال می نماییم.

۳-۱. اسمبلی برنامه C بدون بهینه سازی کامپایلر

```
    .file   "gcd.c"
    .text
    .section    .rodata
.LC1:
    .string "%f\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB7:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $200080, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -200064(%rbp), %rax
    movq    %rax, %rdi
    call    time@PLT
    movl    %eax, %edi
    call    srand@PLT
    movl    $0, -200076(%rbp)
    jmp .L2
.L5:
    movl    $0, -200072(%rbp)
    jmp .L3
.L4:
    call    rand@PLT
    movl    %eax, %ecx
    movl    -200072(%rbp), %eax
    cltq
    movl    -200076(%rbp), %edx
```

```
    movslq  %edx, %rdx
    addq    %rdx, %rdx
    addq    %rbp, %rdx
    addq    %rdx, %rax
    subq    $200016, %rax
    movb    %cl, (%rax)
    addl    $1, -200072(%rbp)
.L3:
    cmpl    $1, -200072(%rbp)
    jle .L4
    addl    $1, -200076(%rbp)
.L2:
    cmpl    $99999, -200076(%rbp)
    jle .L5
    movl    $0, -200068(%rbp)
    jmp .L6
.L7:
    leaq    -200032(%rbp), %rax
    movl    $0, %esi
    movq    %rax, %rdi
    call    gettimeofday@PLT
    movl    -200068(%rbp), %eax
    cltq
    movzbl  -200015(%rbp,%rax,2), %eax
    movzbl  %al, %edx
    movl    -200068(%rbp), %eax
    cltq
    movzbl  -200016(%rbp,%rax,2), %eax
    movzbl  %al, %eax
    movl    %edx, %esi
    movl    %eax, %edi
    call    gcd@PLT
    leaq    -200048(%rbp), %rax
    movl    $0, %esi
    movq    %rax, %rdi
    call    gettimeofday@PLT
    movq    -200040(%rbp), %rdx
    movq    -200024(%rbp), %rax
    subq    %rax, %rdx
    movq    %rdx, %rax
    cvtsi2sdq   %rax, %xmm0
    movsd   -200056(%rbp), %xmm1
    addsd   %xmm1, %xmm0
    movsd   %xmm0, -200056(%rbp)
    addl    $1, -200068(%rbp)
.L6:
    cmpl    $99999, -200068(%rbp)
    jle .L7
    movsd   -200056(%rbp), %xmm0
    movsd   .LC0(%rip), %xmm1
```

```
87.     divsd   %xmm1, %xmm0
88.     leaq    .LC1(%rip), %rdi
89.     movl    $1, %eax
90.     call    printf@PLT
91.     movl    $0, %eax
92.     movq    -8(%rbp), %rcx
93.     xorq    %fs:40, %rcx
94.     je  .L9
95.     call    __stack_chk_fail@PLT
96. .L9:
97.     leave
98.     .cfi_def_cfa 7, 8
99.     ret
100.          .cfi_endproc
101.    .LFE7:
102.         .size   main, .-main
103.         .section    .rodata
104.         .align 8
105.    .LC0:
106.         .long   0
107.         .long   1093567616
108.         .ident  "GCC: (GNU) 8.2.1 20181127"
109.    .section    .note.GNU-stack,"",@progbits
```

کامپایلر سعی در حفظ ماهیت برنامه نوشته شده توسط برنامه نویس نموده و در ساختار برنامه تغییرات بنیادی نداده است. این امر هرچند که باعث خواناتر شدن کد اسمبلی حاصل برای برنامه نویس می‌شود ولی کیفیت برنامه نهایی کاهش می‌یابد.

۳-۲. اسمبلی برنامه C با بهینه سازی کامپایلر

```
1.      .file   "gcd.c"
2.      .text
3.      .section    .rodata.str1.1,"aMS",@progbits,1
4.  .LC1:
5.      .string "%f\n"
6.      .section    .text.startup,"ax",@progbits
7.      .p2align 4,,15
8.      .globl  main
9.      .type   main, @function
10. main:
11. .LFB23:
12.     .cfi_startproc
13.     pushq   %r13
14.     .cfi_def_cfa_offset 16
15.     .cfi_offset 13, -16
16.     pushq   %r12
17.     .cfi_def_cfa_offset 24
18.     .cfi_offset 12, -24
```

```
    pushq   %rbp
    .cfi_def_cfa_offset 32
    .cfi_offset 6, -32
    pushq   %rbx
    .cfi_def_cfa_offset 40
    .cfi_offset 3, -40
    subq    $200088, %rsp
    .cfi_def_cfa_offset 200128
    movq    %fs:40, %rax
    movq    %rax, 200072(%rsp)
    xorl    %eax, %eax
    leaq    24(%rsp), %rdi
    leaq    64(%rsp), %r13
    call    time@PLT
    leaq    200000(%r13), %rbx
    movq    %r13, %rbp
    movl    %eax, %edi
    call    srand@PLT
    .p2align 4,,10
    .p2align 3
.L2:
    call    rand@PLT
    addq    $2, %rbp
    movb    %al, -2(%rbp)
    call    rand@PLT
    movb    %al, -1(%rbp)
    cmpq    %rbp, %rbx
    jne .L2
    leaq    48(%rsp), %r12
    leaq    32(%rsp), %rbp
    jmp .L7
    .p2align 4,,10
    .p2align 3
.L3:
    xorl    %esi, %esi
    movq    %rbp, %rdi
    addq    $2, %r13
    call    gettimeofday@PLT
    movq    40(%rsp), %rax
    pxor    %xmm0, %xmm0
    subq    56(%rsp), %rax
    cvtsi2sdq   %rax, %xmm0
    addsd   8(%rsp), %xmm0
    movsd   %xmm0, 8(%rsp)
    cmpq    %rbx, %r13
    je  .L21
.L7:
    xorl    %esi, %esi
    movq    %r12, %rdi
    call    gettimeofday@PLT
```

```
    movzbl  1(%r13), %eax
    movzbl  0(%r13), %edx
    movl    %eax, %esi
    orb %dl, %sil
    sete    %sil
    testl   %eax, %eax
    sete    %cl
    orb %cl, %sil
    jne .L3
    testl   %edx, %edx
    je  .L3
    cmpl    %edx, %eax
    je  .L3
    .p2align 4,,10
    .p2align 3
.L4:
    cmpl    %edx, %eax
    jbe .L5
.L22:
    subl    %edx, %eax
    cmpl    %edx, %eax
    je  .L3
    cmpl    %edx, %eax
    ja  .L22
.L5:
    subl    %eax, %edx
    cmpl    %eax, %edx
    jne .L4
    jmp .L3
.L21:
    leaq    .LC1(%rip), %rdi
          movl    $1, %eax
          divsd   .LC0(%rip), %xmm0
          call    printf@PLT
          xorl    %eax, %eax
          movq    200072(%rsp), %rbx
          xorq    %fs:40, %rbx
          jne .L23
          addq    $200088, %rsp
          .cfi_remember_state
          .cfi_def_cfa_offset 40
          popq    %rbx
          .cfi_def_cfa_offset 32
          popq    %rbp
          .cfi_def_cfa_offset 24
          popq    %r12
          .cfi_def_cfa_offset 16
          popq    %r13
          .cfi_def_cfa_offset 8
          ret
```

```
.L23:
    .cfi_restore_state
    call    __stack_chk_fail@PLT
    .cfi_endproc
.LFE23:
    .size   main, .-main
    .section    .rodata.cst8,"aM",@progbits,8
    .align 8
.LC0:
    .long   0
    .long   1093567616
    .ident  "GCC: (GNU) 8.2.1 20181127"
    .section    .note.GNU-stack,"",@progbits
```

در حالت فعال بودن بهینه سازی توسط کامپایلر، هر چند کد اسمبلی تولید شده دارای تعداد دستور بیشتری ولی تعداد دستورات شرطی و پرش و فراخوانی کاهش یافته است. این امر باعث اجرای سریع تر و بهینه تر برنامه نهایی می‌گردد.

۳-۳. جدول تسریع

نتایج تسریع در دو حالت بدون بهینه سازی کامپایلر و با بهینه سازی کامپایلر برای برنامه زبان C نسبت به برنامه Python اولیه مطابق جدول زیر می باشد.

جدول ۲- تسریع برنامه نوشته شده به زبان C نسبت به برنامه اولیه نوشته شده به زبان Python

ردیف	برنامه اولیه به زبان Python	برنامه نوشته شده به زبان C			
		بدون بهینه سازی کامپایلر		با بهینه سازی کامپایلر	
	زمان اجرا	زمان اجرا	تسریع	زمان اجرا	تسریع
۱	۶۰۸	۱۰	۶۰٫۸۰	۶	۱۰۱٫۳۳
۲	۶۲۸	۱۱	۵۷٫۰۹	۹	۶۹٫۷۷
۳	۶۲۴	۱۲	۵۲٫۰۰	۶	۱۰۴٫۰۰
۴	۶۲۹	۱۰	۶۲٫۹۰	۶	۱۰۴٫۸۳
۵	۶۴۲	۱۰	۶۴٫۲۰	۶	۱۰۷٫۰۰
۶	۶۱۴	۱۲	۵۱٫۱۶	۸	۷۶٫۷۵
۷	۶۰۳	۱۰	۶۰٫۳۰	۶	۱۰۰٫۵۰
۸	۶۱۸	۱۱	۵۶٫۱۸	۷	۸۸٫۲۸
۹	۶۱۳	۱۱	۵۵٫۷۲	۷	۸۷٫۵۷

۱۰	۶۰۹	۱۳	۴۶٫۸۴	۶	۱۰۱٫۵۰
میانگین	۶۱۸	۱۱	۵۶٫۱۸	۶٫۷۰	۹۲٫۲۳

برنامه نوشته شده به زبان C با فعال بودن بهینه سازی توسط کامپایلر به طور متوسط ۹۲٫۲۳ برابر سریع تر از یک برنامه (با شبه کد مشترک) نوشته شده به زبان Python می باشد.

جدول ۳- تسریع برنامه نوشته شده به زبان C نسبت به برنامه نوشته شده به زبان Python با الگوریتم ۲

ردیف	برنامه الگوریتم ۲ به زبان Python (الگوریتم برابر)	برنامه نوشته شده به زبان C			
		بدون بهینه سازی کامپایلر		با بهینه سازی کامپایلر	
	زمان اجرا	زمان اجرا	تسریع	زمان اجرا	تسریع
۱	۲۴۴	۱۰	۲۴٫۴۰	۶	۴۰٫۶۶
۲	۲۷۰	۱۱	۲۴٫۵۴	۹	۳۰٫۰۰
۳	۲۵۸	۱۲	۲۱٫۵۰	۶	۴۲٫۶۶
۴	۲۲۰	۱۰	۲۲٫۰۰	۶	۳۶٫۶۶
۵	۲۶۸	۱۰	۲۶٫۸۰	۶	۴۴٫۶۶
۶	۲۵۷	۱۲	۲۱٫۴۱	۸	۳۲٫۰۰
۷	۲۶۲	۱۰	۲۶٫۲۰	۶	۴۳٫۶۶
۸	۲۴۷	۱۱	۲۲٫۴۵	۷	۳۵٫۲۸
۹	۲۵۰	۱۱	۲۲٫۷۲	۷	۳۵٫۷۱
۱۰	۲۴۰	۱۳	۱۸٫۴۶	۶	۴۰٫۰۰
میانگین	۲۵۱	۱۱	۲۲٫۸۱	۶٫۷۰	۳۷٫۴۶

در صورت پیاده سازی یک الگوریتم به صورت یک برنامه با زبان C و با فعال بودن بهینه سازی توسط کامپایلر ، این برنامه به طور متوسط ۳۷٫۴۶ برابر سریع تر از پیاده سازی همان الگوریتم با زبان Python اجرا خواهد شد.

۳-۴. استفاده از پارامتر های register

دراین بخش تابع نوشته شده به زبان C را تغییر داده و از register برای پارامترها استفاده می نماییم.

```
1. unsigned int gcd (register unsigned int  a, register unsigned int b){
2.     if (a == 0 &&b == 0)
3.         b = 1;
4.     else if (b == 0)
5.         b = a;
```

```
6.      else if (a != 0)
7.          while (a != b)
8.              if (a <b)
9.                  b -= a;
10.             else
11.                 a -= b;
12.
13.     return b;
14. }
```

با توجه به عدم تغییر نتایج با اعمال بهینه سازی کامپایلر نسبت به حالت قبل(با بهینه سازی کامپایلر) در این بخش اینجانب تسریع را نسبت به حالت بدون بهینه سازی توسط کامپایلر به شرح زیر گزارش می‌نمایم.

جدول ٤- تسریع برنامه C با پارامتر register نسبت به برنامه C اولیه

ردیف	برنامه اولیه نوشته شده به زبان C بدون بهینه سازی کامپایلر	برنامه نوشته شده به زبان C با استفاده از register	
	زمان اجرا	زمان اجرا	تسریع
۱	۱۰	۸	۱٫۲۵
۲	۱۱	۷	۱٫۵۷
۳	۱۲	۷	۱٫۷۱
۴	۱۰	۹	۱٫۱۱
۵	۱۰	۷	۱٫۴۲
۶	۱۲	۷	۱٫۷۱
۷	۱۰	۸	۱٫۲۵
۸	۱۱	۸	۱٫۳۷
۹	۱۱	۷	۱٫۵۷
۱۰	۱۳	۸	۱٫۶۲
میانگین	۱۱	۷٫۶۰	۱٫۴۴

اعمال کلید واژه register به پارامترهای تابع به طور متوسط باعث افزایش ۱٫۴۴ برابری سرعت اجرای برنامه می شود.