# Compilers Homework 1

Name: Swetanjal Dutta
Roll Number: 20171077

14/08/2020

## 1 Problem 1

The version of **gcc** installed on my machine is: **7.5.0**. I found this out by running the bash command:

```
$ gcc --version
```

To update the **gcc** version to the latest version, I first checked the latest release version from gcc gnu compiler website. I found that **gcc-10** is the latest version of gcc released. Downloaded and installed gcc-10 using the following command:

```
$ sudo apt-get install gcc-10
```

Removed default gcc symlink and recreated it to link to gcc-10 compiler using the below commands:

```
$ sudo rm /usr/bin/gcc
$ sudo ln -s /usr/bin/gcc-10 /usr/bin/gcc
```

Rechecking the updated gcc version:

```
$ gcc --version
gcc (Ubuntu 10.1.0-2ubuntu1~18.04) 10.1.0
Copyright (C) 2020 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.
```

## 2 Problem 2

The Clang Compiler is required to compile LLVM Compiler. The compilation time to compile LLVM Compiler is approximately 3.5 hours. A better way to monitor the compilation time is using **time** utility command.

## 3 Problem 3

Installed Intel icc compiler from the official website. I chose to download Intel System Studio. To check the icc compiler version, run the following commands:

```
$ cd /opt/intel/sw_dev_tools/compilers_and_libraries/linux/bin
$ source compilervars.sh intel64
$ icc --version
icc (ICC) 19.1.2.254 20200623
Copyright (C) 1985-2020 Intel Corporation.  All rights reserved.
```

## 4 Problem 5

Compiled the code in **main.c** using the **x86_64 gcc** compiler by running the following command:

```
$ gcc main.c
```

The type of generated executable is checked using the command:

```
$ file a.out
```

The output for the same is as follows:

```
$ a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=29e301d70bf24d364c159f616da2d
9f46b5726a3, not stripped
```

Using the command

```
$ objdump -D -S
```

the following assembly code is generated for **x86_64** ISA corresponding to the *main()*:

```
5fa:    55                      push   %rbp
5fb:    48 89 e5                mov    %rsp,%rbp
5fe:    c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
605:    c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
60c:    eb 0a                   jmp    618 <main+0x1e>
60e:    8b 45 fc                mov    -0x4(%rbp),%eax
611:    01 45 f8                add    %eax,-0x8(%rbp)
614:    83 45 fc 01             addl   $0x1,-0x4(%rbp)
618:    81 7d fc 0f 27 00 00    cmpl   $0x270f,-0x4(%rbp)
61f:    7e ed                   jle    60e <main+0x14>
621:    b8 00 00 00 00          mov    $0x0,%eax
626:    5d                      pop    %rbp
627:    c3                      retq
628:    0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
62f:    00
```

Compiled the code in **main.c** using the **ARM gcc** compiler by running the following command:

```
$ arm-linux-gnueabi-gcc main.c
```

The type of generated executable is found to be as follows:

```
$ a.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.3, for
GNU/Linux 3.2.0, BuildID[sha1]=8cafb01866f5bf7e7aaeda900c497
aafe56400da, not stripped
```

Using the command

```
$ arm-linux-gnueabi-objdump -D -S
```

the following assembly code is generated for **ARM** ISA corresponding to the *main()*:

```
103c8:      e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
103cc:      e28db000        add     fp, sp, #0
103d0:      e24dd00c        sub     sp, sp, #12
103d4:      e3a03000        mov     r3, #0
103d8:      e50b300c        str     r3, [fp, #-12]
103dc:      e3a03000        mov     r3, #0
103e0:      e50b3008        str     r3, [fp, #-8]
103e4:      ea000006        b       10404 <main+0x3c>
103e8:      e51b200c        ldr     r2, [fp, #-12]
103ec:      e51b3008        ldr     r3, [fp, #-8]
103f0:      e0823003        add     r3, r2, r3
103f4:      e50b300c        str     r3, [fp, #-12]
103f8:      e51b3008        ldr     r3, [fp, #-8]
103fc:      e2833001        add     r3, r3, #1
10400:      e50b3008        str     r3, [fp, #-8]
10404:      e51b3008        ldr     r3, [fp, #-8]
```

```
10408:      e59f2018      ldr     r2, [pc, #24]    ; 10428 <main+0x60>
1040c:      e1530002      cmp     r3, r2
10410:      daffffff4     ble     103e8 <main+0x20>
10414:      e3a03000      mov     r3, #0
10418:      e1a00003      mov     r0, r3
1041c:      e28bd000      add     sp, fp, #0
10420:      e49db004      pop     {fp}             ; (ldr fp, [sp], #4)
10424:      e12fff1e      bx      lr
10428:      0000270f      andeq   r2, r0, pc, lsl #14
```

# 5 Problem 6

The microarchitecture of my processor **Intel(R) Core(TM) i5-8250U** is *Skylake microarchitecture.*

# 6 Problem 7

The following matrix multiplication **C** program(*matrix.c*) gives better performance using gcc -*march* compiler option when compared with default compilation.

```
#pragma GCC optimize("O3")
#include <stdio.h>
#include <time.h>

#define SIZE 1000

float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

void init(void)
{
    int i, j, k;
    for(i=0; i<SIZE; ++i)
    {
        for(j=0; j<SIZE; ++j)
        {
            a[i][j] = (float)i + j;
            b[i][j] = (float)i - j;
            c[i][j] = 0.0f;
        }
    }
}

void mult(void)
{
    int i, j, k;

    for(i=0; i<SIZE; ++i)
    {
        for(j=0; j<SIZE; ++j)
        {
            for(k=0; k<SIZE; ++k)
            {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
```

```
    }
}

int main(void)
{
    clock_t s, e;
    init();
    s=clock();
    mult();
    e=clock();
    printf("mult took %10d clocks\n", (int)(e-s));
    return 0;

}
```

Compiling the above code using gcc -march compiler option:

```
$ gcc matrix.c -march=skylake
```

The executable is run and timed as follows:

```
$ time ./a.out
```

The output on the terminal is as follows:

```
mult took      222646 clocks

real    0m0.234s
user    0m0.230s
sys     0m0.004s
```

Next, compiling the **matrix.c** file using default compilation:

```
$ gcc matrix.c
```

The executable is run and timed as follows:

```
$ time ./a.out
```

The output on the terminal is as follows:

```
mult took      401395 clocks

real    0m0.413s
user    0m0.405s
sys     0m0.008s
```

We can clearly see that the running time of the executable compiled using gcc -march compiler option is almost $2x$ faster than the running time of the executable compiled using default options.

GCC provides a range of platform-specific options for different types of CPUs. These options control features such as hardware floating-point modes, and the use of special instructions for different CPUs.

The features of the widely used Intel x86_64 families of processors(386, 486, Pentium, etc) can be controlled with GCC platform-specific options like *-march*. On these platforms, GCC produces executable code which is compatible with all the processors in the x86_64 family by default—going all the way back to the 386. However, it is also possible to compile for a specific processor to obtain better performance. Code produced using the *-march=CPU* option helps us achieve this.

I used the gcc *-march=skylake* option as the microarchitecture of my Intel processor is 'Skylake'. This microarchitecture possibly has a SIMD unit whose advantage was taken by the compiler by vectorising the matrix multiplication loops. Furthermore, the Skylake microarchitecture has multiple cores for execution and the compiler took advantage of this. All this makes the executable run faster. One must note that to distribute executable files for general use on Intel/AMD processors, they should be compiled without any '-march' options. Code produced using *-march* option will not run on other processors in x86_64 family other than the target specific processor.