Mini Assignment - 1

Roll No : AI20BTECH11006

Write a short note on various options in common compilers: GCC and LLVM.

```c
// main.c
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

```
[OUT]: Hello World!
```

GCC

1. `-O0`
   This is same as no optimization, when no level is specified it is -O0 by default
2. `-O1` or `-O`
   It does minimal optimization by slightly reducing executaion time, and memory usage (code size), as a result the compilation time increases slightly.
3. `-O2`
   This optimizes more than `-O1`. This uses all the supported optimization techniques. However, it doesn't perform the optimization techniques that require a trade-off between memory usage and speed. The compilation time rises as a result and is more than that for `-O1` but the execution time is smaller. There are various flags that will be turned on by using this flag such as `-fexpensive-optimizations`
4. `-O3`
   This flag optimizes even more than `-O2`
5. `-Os`
   This flag is used to heavily optimize for size. On top of `-O2` it performs more optimizations that reduce the code size. It will turn off the flags in `-O2` that might increase the code size.
6. `-Oz`
   This flag is same as `-Os` but used only in Apple products. The MAN page has no reference for this flag.
7. `-S`
   This flag produces a human readable assembly code for given program.

```
[IN] gcc -S main.c
[OUT]
```

```
    .file   "main.c"
    .text
    .section    .rodata
.LC0:
    .string "Hello World!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5
0:
    .string  "GNU"
1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f
2:
    .long    0x3
3:
    .align 8
4:
```

8. `-o`

   This option is for output name

   `gcc main.c`

   This will compile the program, but it will name the executable as a.out. Instead we can use `-o` option to rename the executable

   `gcc main.c -o <name>`

9. `-v`

   This option is for verbose. It is used to get additional information about the stages involved in compilation. The man page of gcc says that this flag also prints the version number of compiler, preprocessor.

```
[IN] gcc -v main.c
[OUT]
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-
1ubuntu1~20.04.1' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --
enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr
--with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-
linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib -
-without-included-gettext --enable-threads=posix --libdir=/usr/lib --
enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-
libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-
object --disable-vtable-verify --enable-plugin --enable-default-pie --with-
system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-
multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-
multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-
offload-targets=nvptx-none=/build/gcc-9-Av3uEd/gcc-9-9.4.0/debian/tmp-
nvptx/usr,hsa --without-cuda-driver --enable-checking=release --
build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -quiet -v -imultiarch x86_64-linux-gnu
a.c -quiet -dumpbase a.c -mtune=generic -march=x86-64 -auxbase a -version -
fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-
security -fstack-clash-protection -fcf-protection -o /tmp/ccHWNEzF.s
GNU C17 (Ubuntu 9.4.0-1ubuntu1~20.04.1) version 9.4.0 (x86_64-linux-gnu)
        compiled by GNU C version 9.4.0, GMP version 6.2.0, MPFR version
4.0.2, MPC version 1.1.0, isl version isl-0.22.1-GMP

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/local/include"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/9/include-
fixed"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-
gnu/9/../../../../x86_64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/9/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
```

```
GNU C17 (Ubuntu 9.4.0-1ubuntu1~20.04.1) version 9.4.0 (x86_64-linux-gnu)
        compiled by GNU C version 9.4.0, GMP version 6.2.0, MPFR version
4.0.2, MPC version 1.1.0, isl version isl-0.22.1-GMP

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: c0c95c0b4209efec1c1892d5ff24030b
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 as -v --64 -o /tmp/ccDNEA6E.o /tmp/ccHWNEzF.s
GNU assembler version 2.34 (x86_64-linux-gnu) using BFD version (GNU
Binutils for Ubuntu) 2.34
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-
gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/9/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-
linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-
gnu/9/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-
linux-gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-
gnu/9/lto-wrapper -plugin-opt=-fresolution=/tmp/ccIY8l5I.res -plugin-opt=-
pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-
through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-
lgcc_s --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed
-dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/Scrt1.o
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-
gnu/9 -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu -
L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu -
L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -
L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. /tmp/ccDNEA6E.o -lgcc --push-
state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -
lgcc_s --pop-state /usr/lib/gcc/x86_64-linux-gnu/9/crtendS.o
/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
```

10. `-x`

    This option is used to forcefully specify the language of the prgram like c, c++, d etc.

    ```
    gcc -x c main.c
    ```

11. `-E`

    This option is used to stop after preprocessing, it doesn't run the compilation stage. The output is directed to stdout.

    ```
    gcc -E main.c
    ```

    The above results in more than 500 lines, in interest of space when using `<stdio.h>`, I have skipped the output

12. `-Wextra`

    Consider the following program

```c
#include <stdio.h>
int main()
{
    int i;
    printf("Hello World!");
    printf("%d",i);
    return 0;
}
```

This flag will return warnings, here `i` hasn't been initialised by the user, but it is being printed, so the flag will produce a warning.

`gcc -Wextra main.c`

The output is

```
main.c: In function ‘main’:
main.c:6:5: warning: ‘i’ is used uninitialized in this function [-
Wuninitialized]
    6 |     printf("%d",i);
      |     ^~~~~~~~~~~~~~
```

13. `--version`
    This returns the version, some information about LICENSE.

```
[IN] gcc --version
[OUT]
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 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.
```

14. `-g`
    This is the default debugging flag.

LLVM

---

1. `--version`
   prints the version of compiler
   `clang --version`
2. `--help`
   prints a list of commands with short description
   `clang --help`
3. `-o`
   used to change the name of output file.
   `clang main.c -o main`

4. `-g`
   This option enables debugging. source
5. `-O1`
   This is used to enable trivial optimizations
6. `-O2`
   This enables the default optimizations.
7. `-O3`
   This flag optimizes even more than `-O2`
8. `-Os`
   This is similar to `-O2` but it has extra optimizations to reduce code size
9. `-Oz`
   This is similar to `-Os` but it reduces code size even further.
10. `-Ofast`
    This is a more agressive option, it enables all the optimizations from `-O3` anf even those which might cause certain violations to language standards.
11. `-E`
    This option is used to stop after preprocessing, it doesn't run the compilation stage. The output is directed to stdout.
    Example Usage

```c
int main()
{
    int a = 0;
    for (int i = 0; i< 1000; i++)
    {
        a = 5;
    }
    // more comments
    for(int i = 0; i < 10000; i++)
    {
        a+=3;
        a-=2;
    }
    return 0;
}
```

[OUT]

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 341 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
int main()
{
    int a = 0;
```

```
    for (int i = 0; i< 1000; i++)
    {
        a = 5;
    }

    for(int i = 0; i < 10000; i++)
    {
        a+=3;
        a-=2;
    }
    return 0;
}
```

12. `-S`

    This flag produces a human readable assembly code for given program.

    Example Usage

```
int main()
{
    int a = 0;
    for (int i = 0; i< 1000; i++)
    {
        a = 5;
    }
    // more comments
    for(int i = 0; i < 10000; i++)
    {
        a+=3;
        a-=2;
    }
    return 0;
}
```

[OUT]

```
    .text
    .file   "main.c"
    .globl  main                    # -- Begin function main
    .p2align    4, 0x90
    .type   main,@function
main:                                   # @main
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    $0, -4(%rbp)
```

```
    movl    $0, -8(%rbp)
    movl    $0, -12(%rbp)
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
    cmpl    $1000, -12(%rbp)        # imm = 0x3E8
    jge .LBB0_4
# %bb.2:                                #   in Loop: Header=BB0_1 Depth=1
    movl    $5, -8(%rbp)
# %bb.3:                                #   in Loop: Header=BB0_1 Depth=1
    movl    -12(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -12(%rbp)
    jmp .LBB0_1
.LBB0_4:
    movl    $0, -16(%rbp)
.LBB0_5:                                # =>This Inner Loop Header: Depth=1
    cmpl    $10000, -16(%rbp)       # imm = 0x2710
    jge .LBB0_8
# %bb.6:                                #   in Loop: Header=BB0_5 Depth=1
    movl    -8(%rbp), %eax
    addl    $3, %eax
    movl    %eax, -8(%rbp)
    movl    -8(%rbp), %eax
    subl    $2, %eax
    movl    %eax, -8(%rbp)
# %bb.7:                                #   in Loop: Header=BB0_5 Depth=1
    movl    -16(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -16(%rbp)
    jmp .LBB0_5
.LBB0_8:
    xorl    %eax, %eax
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc
                                        # -- End function
    .ident  "clang version 10.0.0-4ubuntu1 "
    .section    ".note.GNU-stack","",@progbits
    .addrsig
```

Write a note on the various frontends that these compilers support.

GCC

There are various frontends that this compiler support. Some of them are listed below

1. GNU Pascal Compiler (GPC)
   This is a free (Open-Source) software which supports all OS supported by GNU C, it is also compatible with GNU tools like debugger.

2. Mercury
   This is a declarative/functional language, which can directly produce assembly code by using gcc backend.

3. GHDL
   This is also open-source. It is used to compile and execute VHDL code.

4. GNU UPC
   It is a compilation and execution environment for programs written in unified Parallel C.

5. Cobol for GCC
   It is a free Cobol compiler compliant with COBOL 85 standard, integrated into GCC.

LLVM

---

1. CLANG
   It is used for C, C++, Objective C, OpenCl, CUDA etc.
2. flang
   This is front end for fortran
3. emscripten
   This is used for javascript
4. rubinius
   This is used for Ruby.

Use these compilers to generate code for various architectures using its various backends and report your findings.

GCC

GCC supports only x86 which we use to compile a program using say `gcc main.c`, I used it on x86_64. However, GCC can still be used on different architectures. For example, we can use `arm-linux-gcc` or `gcc-i686-linux-gnu` etc.

Listed below are some examples of assembly code generated for different architectures, I used this tool.

Code

```
int main()
{
    int a = 0;
    for(int i = 0 ; i < 5 ; i++)
    {
        a = 5;
    }
    return 0;
}
```

[OUT] x86_64

```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-8], 0
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        mov     DWORD PTR [rbp-8], 5
        add     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 4
        jle     .L3
        mov     eax, 0
        pop     rbp
        ret
```

[OUT] mips64

```
main:
        daddiu  $sp,$sp,-32
        sd      $fp,24($sp)
        move    $fp,$sp
        sw      $0,4($fp)
        sw      $0,0($fp)
        b       .L2
        nop

.L3:
        li      $2,5                            # 0x5
        sw      $2,4($fp)
        lw      $2,0($fp)
        addiu   $2,$2,1
        sw      $2,0($fp)
.L2:
        lw      $2,0($fp)
        slt     $2,$2,5
        bne     $2,$0,.L3
        nop

        move    $2,$0
        move    $sp,$fp
        ld      $fp,24($sp)
        daddiu  $sp,$sp,32
        jr      $31
        nop
```

[OUT] ARM64

```
main:
        sub     sp, sp, #16
        str     wzr, [sp, 8]
        str     wzr, [sp, 12]
        b       .L2
.L3:
        mov     w0, 5
        str     w0, [sp, 8]
        ldr     w0, [sp, 12]
        add     w0, w0, 1
        str     w0, [sp, 12]
.L2:
        ldr     w0, [sp, 12]
        cmp     w0, 4
        ble     .L3
        mov     w0, 0
        add     sp, sp, 16
        ret
```

Similarly, we can generate them for a variety of other compilers.

LLVM

It supports various architectures, examples

```
clang -c -target arm64 main.c
```

```
clang -c -target mips main.c
```

Similarly, we can do this for various other architectures.
Here is the proper format of target triple `<arch><sub>-<vendor>-<sys>-<abi>`

- arch = x86_64, i386, arm, thumb, mips, etc.
- sub = for ex. on ARM: v5, v6m, v7a, v7m, etc.
- vendor = pc, apple, nvidia, ibm, etc.
- sys = none, linux, win32, darwin, cuda, etc.
- abi = eabi, gnu, android, macho, elf, etc.

The above list has been taken from the official Clang website.
An example of target-triple is `x86_64-pc-linux-gnu`

We can do what we did for GCC for LLVM. Here are few example

[OUT] RISC-V rv64gc clang 14.0.0

```
main:                                   # @main
        addi    sp, sp, -32
        sd      ra, 24(sp)                      # 8-byte Folded Spill
        sd      s0, 16(sp)                      # 8-byte Folded Spill
        addi    s0, sp, 32
        li      a0, 0
        sw      a0, -20(s0)
        sw      a0, -24(s0)
        sw      a0, -28(s0)
        j       .LBB0_1
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        lw      a1, -28(s0)
        li      a0, 4
        blt     a0, a1, .LBB0_4
        j       .LBB0_2
.LBB0_2:                                #   in Loop: Header=BB0_1 Depth=1
        li      a0, 5
        sw      a0, -24(s0)
        j       .LBB0_3
.LBB0_3:                                #   in Loop: Header=BB0_1 Depth=1
        lw      a0, -28(s0)
        addiw   a0, a0, 1
        sw      a0, -28(s0)
        j       .LBB0_1
.LBB0_4:
        li      a0, 0
        ld      ra, 24(sp)                      # 8-byte Folded Reload
        ld      s0, 16(sp)                      # 8-byte Folded Reload
        addi    sp, sp, 32
        ret
```

[OUT] armv8-a clang 11.0.1

```
main:                                   // @main
        sub     sp, sp, #16                     // =16
        str     wzr, [sp, #12]
        str     wzr, [sp, #8]
        str     wzr, [sp, #4]
.LBB0_1:                                // =>This Inner Loop Header:
Depth=1
        ldr     w8, [sp, #4]
        cmp     w8, #5                          // =5
        b.ge    .LBB0_4
        mov     w8, #5
        str     w8, [sp, #8]
        ldr     w8, [sp, #4]
        add     w8, w8, #1                      // =1
        str     w8, [sp, #4]
        b       .LBB0_1
.LBB0_4:
        mov     w8, wzr
```

```
        mov     w0, w8
        add     sp, sp, #16                     // =16
        ret
```

Compilers come with various optimization levels: Focusing on options O0, O1, O2, O3 as well as -Os, -Oz. Run various codes using these predetermined passes and report your findings.

GCC

Here is a short size comparison from `bzip2`

```
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -S main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
28377 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -S -O1 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
18985 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -S -O2 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
20303 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -S -O3 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
27380 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -S -Os main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
14353 main.s
```

We can see that the order of optimization in terms of length of code is

> O0>O3>O2>O1>Os

Now, we will compare the time of execution of the binary, we will compare based on `usr time`

```
cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end

________________________________________________________
Executed in    7.09 secs    fish           external
   usr time    6.25 secs   85.35 millis    6.16 secs
   sys time    0.86 secs  509.70 millis    0.35 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -O1 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end

________________________________________________________
```

```
Executed in    1.86 secs    fish           external
   usr time    1.15 secs   43.10 millis    1.11 secs
   sys time    0.74 secs  441.75 millis    0.30 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -O2 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end


________________________________________________________
Executed in    1.00 secs      fish           external
   usr time  572.83 millis   39.72 millis  533.10 millis
   sys time  492.24 millis  389.22 millis  103.02 millis

cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -O3 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end


________________________________________________________
Executed in  995.92 millis    fish           external
   usr time  556.80 millis   35.75 millis  521.05 millis
   sys time  478.20 millis  365.84 millis  112.35 millis

cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -Os main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end


________________________________________________________
Executed in    1.19 secs      fish           external
   usr time  669.19 millis   37.45 millis  631.75 millis
   sys time  560.59 millis  406.37 millis  154.23 millis

cmaspi in ~/s/c/c/assignment-1 </ main > $ gcc -Ofast main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 1000);
./a.out ;end


________________________________________________________
Executed in  980.96 millis    fish           external
   usr time  541.18 millis   36.01 millis  505.17 millis
   sys time  478.35 millis  372.38 millis  105.98 millis
```

The order of performance is

> Ofast > O3 > O2 > Os > O1 > O0

I compared Compilation time using bzip2 source code, the results are as follows

```
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc
main.c; end


________________________________________________________
Executed in   18.44 secs    fish           external
```

```
   usr time   16.88 secs   15.51 millis   16.87 secs
   sys time    1.51 secs   11.73 millis    1.50 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc -
O1 main.c; end


________________________________________________________
Executed in   60.57 secs    fish           external
   usr time   58.47 secs    3.02 millis   58.46 secs
   sys time    2.02 secs   28.57 millis    1.99 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc -
O2 main.c; end


________________________________________________________
Executed in  106.44 secs    fish           external
   usr time  103.85 secs    8.06 millis  103.84 secs
   sys time    2.39 secs   20.94 millis    2.37 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc -
O3 main.c; end


________________________________________________________
Executed in  144.35 secs    fish           external
   usr time  141.32 secs   17.25 millis  141.30 secs
   sys time    2.81 secs   13.43 millis    2.79 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc -
Os main.c; end


________________________________________________________
Executed in   65.49 secs    fish           external
   usr time   63.55 secs    2.83 millis   63.55 secs
   sys time    1.78 secs   26.64 millis    1.76 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); gcc -
Ofast main.c; end


________________________________________________________
Executed in  145.74 secs    fish           external
   usr time  142.57 secs   15.32 millis  142.56 secs
   sys time    2.96 secs   14.96 millis    2.95 secs
```

compilation time comparison is given below

> Ofast > O3 > O2 > Os > O1 > O0

LLVM

First we will compare the compilation time

```
cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
main.c; end

_____________________________________________________
Executed in    9.44 secs    fish           external
   usr time    7.46 secs   14.83 millis    7.44 secs
   sys time    1.45 secs   18.21 millis    1.44 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
-O1 main.c; end

_____________________________________________________
Executed in   46.80 secs    fish           external
   usr time   44.15 secs   28.57 millis   44.13 secs
   sys time    1.68 secs   20.07 millis    1.66 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
-O2 main.c; end

_____________________________________________________
Executed in   94.59 secs    fish           external
   usr time   90.81 secs    6.91 millis   90.80 secs
   sys time    1.78 secs   26.98 millis    1.75 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
-O3 main.c; end

_____________________________________________________
Executed in   99.89 secs    fish           external
   usr time   96.28 secs    7.14 millis   96.28 secs
   sys time    1.71 secs   27.07 millis    1.68 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
-Os main.c; end

_____________________________________________________
Executed in   89.90 secs    fish           external
   usr time   86.06 secs   10.76 millis   86.05 secs
   sys time    1.82 secs   23.10 millis    1.80 secs

cmaspi in ~/s/c/c/assignment-1 </ main > $ time for i in (seq 1 50); clang
-Oz main.c; end

_____________________________________________________
Executed in   73.34 secs    fish           external
   usr time   69.57 secs    0.77 millis   69.57 secs
   sys time    1.77 secs   33.20 millis    1.74 secs
```

The observed time comparison is

> O3>O2>Os>Oz>O1>O0

Now, for the size of code

```
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
27845 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S -O1 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
17665 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S -O2 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
23384 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S -O3 main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
24916 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S -Os main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
20243 main.s
cmaspi in ~/s/c/c/assignment-1 </ main > $ clang -S -Oz main.c
cmaspi in ~/s/c/c/assignment-1 </ main > $ wc -l main.s
17366 main.s
```

Comparison

> O0 > O3 > O2 > Os > O1 > Oz