成绩	

本科生《编译原理》课程实践报告

题　　目： 编译器认知实验

学　　院： 徐特立学院

专业名称： 计算机科学与技术

姓　　名： 陈照欣-1120191086

1. 实验目的

本实验的目的是了解工业界常用的编译器 GCC 和 LLVM，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容，了解编译器的优化效果，为编译器的学习和构造奠定基础。

2. 实验内容

本实验主要的内容为在 Linux 平台上安装和运行工业界常用的编译器 GCC 和 LLVM，如果系统中没有安装，则需要首先安装编译器，安装完成后编写简单的测试程序，使用编译器编译，并观察中间输出结果。

于不同的编译器 (GCC 和 LLVM)，分别完成查看编译器版本内容，程序编译的中间结果，查看汇编代码等任务，编写测试程序，运行编译器并进行观测。分别运行 GCC 和 LLVM 编译器，首先学习编译器的基本使用方法，其次通过编译选项查看分析编译器的中间结果，及其与输入源码之间的对应关系，最后使用-O0、-O1、-O2 和-O3 分别使用两个编译器对输入程序进行优化编译，并对比 GCC 和 LLVM 优化后程序的运行效率。

3. 具体实验步骤和操作

3.1 GCC

3.1.1 查看 GCC 版本

使用命令 gcc-V

```
charles@ubuntu:~/Documents/lab2$ gcc -v
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 9.3.0-17ubuntu1~
20.04' --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-ve
rsion-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shar
ed --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --
enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enab
le-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new
 --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-de
fault-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=au
to --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --wit
h-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-off
load-targets=nvptx-none=/build/gcc-9-HskZEa/gcc-9-9.3.0/debian/tmp-nvptx/usr,hs
a --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --ho
st=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

3.1.2 使用编译器编译单个文件

```
charles@ubuntu:~/Documents/lab2$ gedit hello.c
charles@ubuntu:~/Documents/lab2$ gcc hello.c -o hello.out
charles@ubuntu:~/Documents/lab2$ ./hello.out
Hello World!
```

hello.c 代码如下：

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

3.1.3 使用编译器链接多个文件

Step1. 建立 test.h、test.c、main.c

test.h：

```
#ifndef test_h__
#define test_h__

extern void test(void);

#endif
```

test.c：

```
#include<stdio.h>
void test(void)
{
      printf("This is a test library.");
      return;
}
```

main.c：

```
#include <stdio.h>
#include "test.h"
int main(void)
{
      puts("This is a shared library test...");
      test();
      return 0;
}
```

test.h 定义了一个接口连接我们的库，一个简单的函数，test()。test.c 包含了这个函数的实现，main.c 是一个用到我们库的驱动程序

Step2：编译无约束位代码

```
$ gcc -c -Wall -Werror -fpic test.c
```

Step3：从一个对象文件创建共享库

```
$ gcc -shared -o libtest.so test.o
```

Step4：连接共享库

```
$ gcc -Wall -o test main.c -ltest
/usr/bin/ld: cannot find -ltest
collect2: error: ld returned 1 exit status
```

出现报错，GCC 有一个默认的搜索列表，但我们的目录并不在那个列表当中。我们需要告诉 GCC 去哪里找到 libfoo.so。这就要用到-L 选项。在本例中，我们将使用当前目录

/home/charles/Documents/lab2：

```
$ gcc -L/home/charles/Documents/lab2 -Wall -o Test main.c -ltest
```

Step5：运行时使用库

```
$ ./Test
./Test: error while loading shared libraries: libtest.so: cannot open shared object file: No such file or directory
```

出现报错：加载器不能找到共享库。我们没有将它安装到标准位置，使用环境变量 LD_LIBRARY_PATH

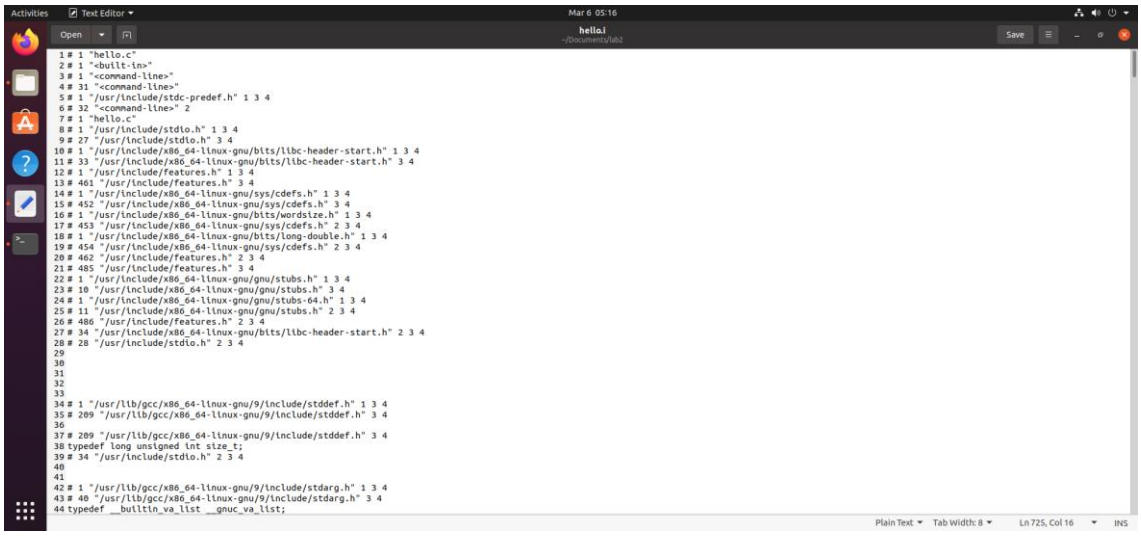
```
$ echo $LD_LIBRARY_PATH
$ LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
$ ./Test
```

```
charles@ubuntu:~/Documents/lab2$ gedit main.c
charles@ubuntu:~/Documents/lab2$ gedit test.h
charles@ubuntu:~/Documents/lab2$ gedit test.c
charles@ubuntu:~/Documents/lab2$ gcc -c -Wall -Werror -fPIC test.c
charles@ubuntu:~/Documents/lab2$ gcc -shared -o libtest.so test.o
charles@ubuntu:~/Documents/lab2$ gcc -L/home/charles/Documents/lab2 -Wall -o Test main.c -ltest
charles@ubuntu:~/Documents/lab2$ ./Test
./Test: error while loading shared libraries: libtest.so: cannot open shared object file: No such file or directory
charles@ubuntu:~/Documents/lab2$ echo $LD_LIBRARY_PATH

charles@ubuntu:~/Documents/lab2$ LD_LIBRARY_PATH=/home/charles/Documents/lab2:$LD_LIBRARY_PATH
charles@ubuntu:~/Documents/lab2$ export LD_LIBRARY_PATH=/home/charles/Documents/lab2:$LD_LIBRARY_PATH
charles@ubuntu:~/Documents/lab2$ ./Test
This is a shared library test...
This is a test library.charles@ubuntu:~/Documents/lab2$
```

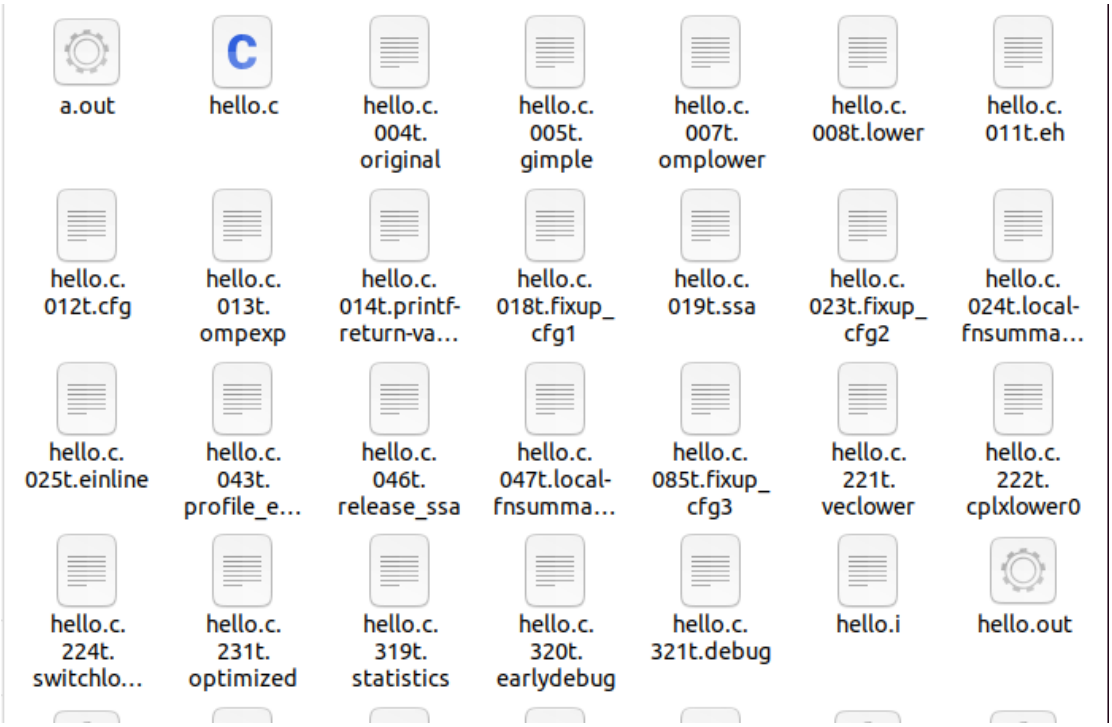
3.1.4 查看预处理结果：gcc -E hello.c -o hello.i

生成 hello.i

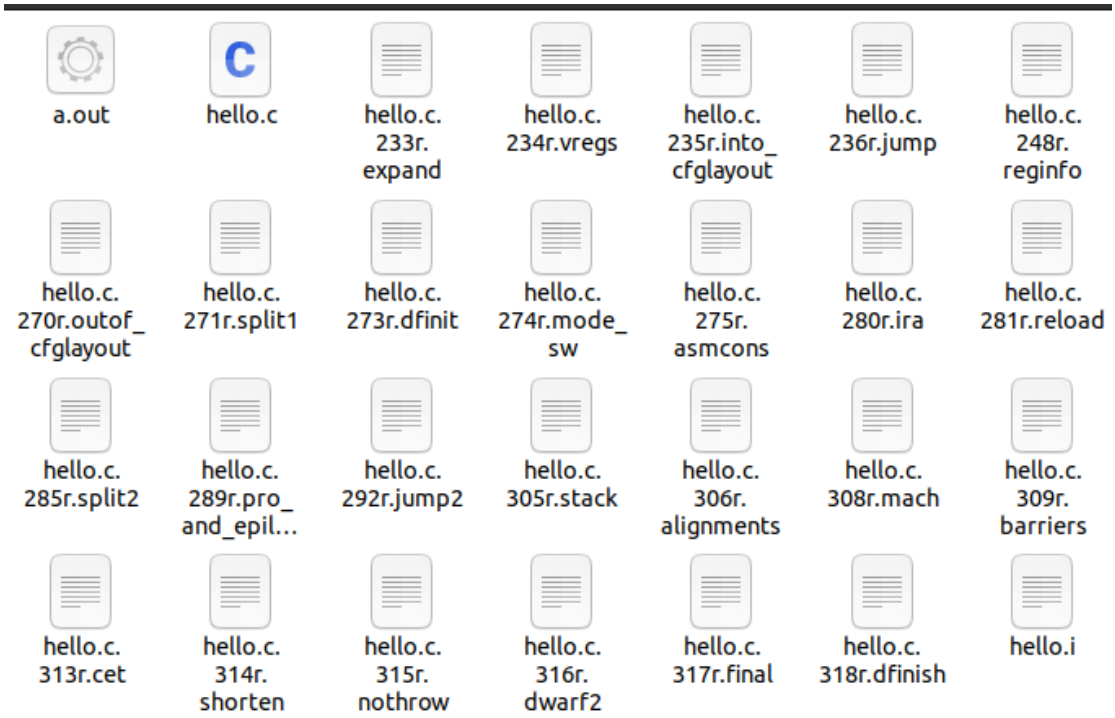


内容如上

3.1.5 查看语法分析树：gcc -fdump-tree-all hello.c



3.1.6　查看中间代码生成结果：gcc -fdump-rtl-all hello.c



3.1.7　查看生成的目标代码（汇编代码）：gcc -S hello.c -o hello.s

```
        .file   "hello.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
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
        .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
```

3.2 LLVM

3.2.1 查看编译器版本

输入命令

```
clang -v
```

得到 clang 版本号:

```
charles@ubuntu:~/Desktop$ clang -v
clang version 10.0.0-4ubuntu1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
Found candidate GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/9
Found candidate GCC installation: /usr/lib/gcc/x86_64-linux-gnu/9
Selected GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/9
Candidate multilib: .;@m64
Selected multilib: .;@m64
```

3.2.2 使用编译器编译单个文件

输入命令:

```
clang hello.c -o hello_clang.out
```

得到 hello_clang.out，执行文件得到输出:

```
charles@ubuntu:~/Documents/lab2$ clang hello.c -o hello_clang.out
charles@ubuntu:~/Documents/lab2$ ./hello_clang.out
Hello World!
charles@ubuntu:~/Documents/lab2$
```

3.2.3 使用编译器编译链接多个文件

此处采用与 gcc 实验中不同的方法实现:

hello.c

```c
#include<stdio.h>
void test();
int main()
{
    printf("Hello World!\n");
    test();
```

```
    return 0;
}
```

test_clang.c

```
void test()
{
    printf("Test\n");
    return;
}
```

输入命令

```
clang -o a.out test_clang.c hello.c
./a.out
```

得到输出

```
charles@ubuntu:~/Documents/lab2$ ./a.out
Hello World!
Test
```

3.2.4 查看编译流程和阶段：clang -ccc-print-phases hello.c -c

```
charles@ubuntu:~/Documents/lab2$ clang -ccc-print-phases hello.c -c
         +- 0: input, "hello.c", c
      +- 1: preprocessor, {0}, cpp-output
   +- 2: compiler, {1}, ir
+- 3: backend, {2}, assembler
4: assembler, {3}, object
```

3.2.5 查看词法分析结果：clang hello.c -Xclang -dump-tokens

以如下语句为例：

```
int m = 123456;
```

词法分析结果如下：

```
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<hello.c:4:2>
identifier 'm'   [LeadingSpace] Loc=<hello.c:4:6>
equal '='        [LeadingSpace] Loc=<hello.c:4:8>
numeric_constant '123456'        [LeadingSpace] Loc=<hello.c:4:10>
semi ';'                 Loc=<hello.c:4:16>
```

3.2.6 查看词法分析结果 2：clang hello.c -Xclang -dump-raw-tokens

以如下语句为例：

```
int m = 123456;
```

词法分析结果如下：

```
raw_identifier 'int'     [StartOfLine]  Loc=<hello.c:4:2>
unknown ' '              Loc=<hello.c:4:5>
raw_identifier 'm'               Loc=<hello.c:4:6>
unknown ' '              Loc=<hello.c:4:7>
equal '='                Loc=<hello.c:4:8>
unknown ' '              Loc=<hello.c:4:9>
numeric_constant '123456'                Loc=<hello.c:4:10>
semi ';'                 Loc=<hello.c:4:16>
unknown '
        '                Loc=<hello.c:4:17>
```

3.2.7　查看语法分析结果：clang hello.c -Xclang -ast-dump

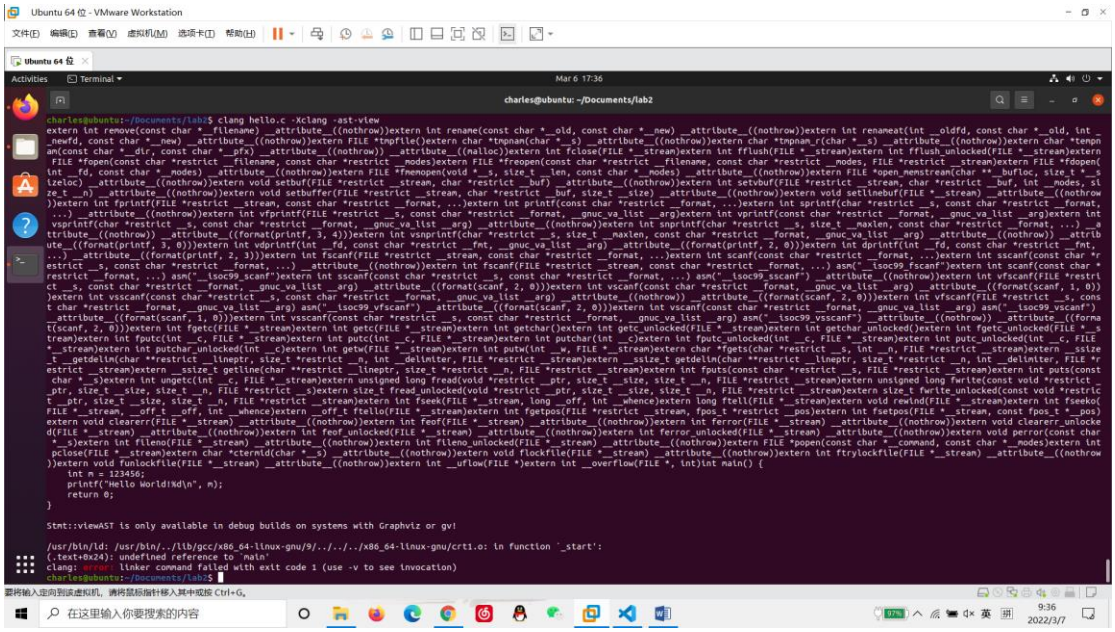
以如下语句为例：

```
int m = 123456;
```

词法分析结果如下：

```
`-VarDecl 0x1c85558 <col:2, col:10> col:6 used m 'int' cinit
  `-IntegerLiteral 0x1c855c0 <col:10> 'int' 123456
-CallExpr 0x1c856c0 <line:5:2, col:29> 'int'
 |-ImplicitCastExpr 0x1c856a8 <col:2> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
```

3.2.8　查看语法分析结果 2：clang hello.c -Xclang -ast-view



3.2.9　查看编译优化的结果：clang hello.c -S -mllvm -print-after-all

```
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
*** IR Dump After Exception handling preparation ***
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 123456, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]* @.str, i64 0, i64 0), i32 %3)
  ret i32 0
}
*** IR Dump After Safe Stack instrumentation pass ***
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 123456, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]* @.str, i64 0, i64 0), i32 %3)
  ret i32 0
}
```

对应代码

```
int m = 123456;
```

将整型常量 123456 存入一个寄存器中

3.2.10　查看生成的目标代码结果：clang hello.c –S

以语句

```
int m = 123456;
```

为例，对应的汇编语句为：

```
movl    $123456, -8(%rbp)        # imm = 0x1E240
```

即将常量 123456 移动到堆栈基指针 (rbp) 偏移一个整型常量 (-8) 的寄存器中

4. 具体实验步骤和操作

对于同一个程序 (矩阵乘法)，在相同的配置（Linux）下，经过不同的编译器的不同优化，执行时间如表所示：可以从下表看到，编译器不同级别的优化会对运行时间产生显著的影响。

编译器	优化	时间/s
GCC	-O0	2.55
GCC	-O1	1.852
GCC	-O2	1.788
GCC	-O3	1.186
LLVM	-O0	2.135
LLVM	-O1	1.912
LLVM	-O2	1.890
LLVM	-O3	1.518

5. 具体实验步骤和操作

本次实验让我体会到了高级语言“抽象”的优点，底层代码实在过于复杂，不利于程序员直接实现自己的逻辑与想法。同时也让我认识到了不同编译器之间的区别，以及不同的编译优化对函数运行的时间等有很大的影响。最后，也体会到了编译器设计的不容易，在看生成的中间代码的时候，几乎是所有语句一开始都摸不着头脑，但大致上对 GCC 的编译过程有了一个较为“理性”的认知，希望在后续的课程中能够继续学习相关内容。