LLVM的使用以及LLVM IR的理解

1.下载并编译LLVM源代码

- 机器配置:
  - 内存：7.63GB
  - 共12个processor(0-11号)

```
//一个processor的信息
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 158
model name  : Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
stepping    : 10
microcode   : 0xde
cpu MHz     : 3433.568
cache size  : 9216 KB
physical id : 0
siblings    : 12
core id     : 0
cpu cores   : 6
apicid      : 0
initial apicid  : 0
fpu     : yes
fpu_exception   : yes
cpuid level : 22
wp      : yes
bugs        : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips    : 4399.99
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

- 操作系统版本号

```
Linux micheallei-TUF-GAMING-FX504GE-FX80GE 5.4.0-53-generic #59~18.04.1-Ubuntu SMP Wed Oct 21 12:14:56 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

- 下载的llvm版本号:11.0.0
- 编译配置模式:Release
- 编译耗费时间:50min
- 编译时内存占用:2GB--7GB不等 ,采用htop工具查看
- 编译后占的硬盘资源大小:
  - llvm:约582MB
  - llvm-build:3.3GB
  - llvm-install:2.2GB
- 实验中遇到的问题:

- 一开始看https://clarazhang.gitbooks.io/compiler-f2018/content/environment.html上的安装方式,发现安装llvm的版本不对,而且修改命令版本号后会报URL move permanently的错误,所以在网上搜到一个安装教尝试下载源码,然而由下面命令
- `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
- 运行了一个小时也没把文件下载下来.与助教沟通后发现可能是因为对应的源没咋维护了或者确实下载的东西有偏差.最后考虑在官网下载了最新版本的tar文件,解压后按gitbook上教程成功编译并配好环境变量.
- 编译时遇到一个问题就是,我一共12个核,我`make -j11 install`跑的时候,编译到20%左右直接电脑卡死.吸取教训后,重启电脑用8个核跑就还行,其余电脑资源还能较为正常运作.

2.理解C与LLVM IR的对应关系

- LLVM IR的基本信息
  - 主要特征
    - RISC风格的三地址代码
    - SSA格式、无限的虚拟寄存器
    - 简单、低级的控制流系统
    - load/store指令带类型化指针：是强类型系统，所有value都有其自身的类型

      IR类型系统及与c语言类型的对应
      - i1:bool（1位宽整数）
      - i8:char (8位宽整数)
      - i32：int（32位宽整数）
      - float:单精度浮点数
      - pointer：指针
      - label：basicblock标识符类型
      - 结构体：{i32,i8}对应struct{int ,char}
  - 一些语句的解释
    - load：读内存
    - store：写内存
    - ret：将控制流从函数返回到调用者
    - br：控制流转移到当前功能里的另一个基本块，有条件分支和无条件分支两种形式
    - alloca：在当前执行函数的堆栈帧上分配内存，当该函数返回时将自动释放该内存。
    - getelementptr:指向数组的元素和指向结构体成员的指针，仅仅计算地址，不访问内存
    - add：求和
    - sub：求差
    - Icmp：比较两个同类型整数，返回布尔值
      - 比较条件：eq(=),ne(!=),sgt(>),sge(>=),slt(<),sle(<=)
    - call:转移控制流到指定函数，并传入参数。由ret返回后，将从该位置继续执行，函数返回值会绑定到result参数
- 自己编写的c程序与对应llvm IR关系分析
  - 使用命令`clang -S -emit-llvm testi.c`生成c程序对应的.ll文件
  - test1.c：函数调用
    - 函数调用过程首先是初始化参数(分配空间并赋值)

      注意到caller函数里为变量%1分配了空间并为其存储值4

- 然后调用call来调用函数，括号里对应传入实参，如`i32* %1`,实参类型与c中一致。然后也会用一个变量(%2)来存放函数返回值
- callee函数里会先把参数赋值给一个新分配的同类型局部变量。然后注意到对与指针类型的数据，需要连续两次load才能真正取出指针实际对应的数据
- ret返回到调用者时也会一并传回返回值。

- test2.c: switch结构,for循环结构,数组变量
  - 首先是数组变量分配：`%3 = alloca [3 x i32], align 4`对应`int a[3]`
  - 调用了一个scanf函数
  - switch结构里：首先是对%6(对应c中in变量的值)做选择，根据不同的值跳转到不同的块。（default调到label %22对应分支）
    - block7：对应case1：此时什么都不做，break直接到block23，即跳出switch
    - block8：对应case2：因为case2中是一个for循环语句，会单独生成一个block，所以又再次跳转
    - block9：对for循环赋值并判断for条件是否成立，利用icmp指令和br指令来判断是否是进入循环(block12)，还是退出循环(到block21)
    - block12：for循环内部语句，实现`a[i]=i+2`。i的值存在变量%2中

      ```
      %15 = load i32, i32* %2, align 4
      %16 = sext i32 %15 to i64
      %17 = getelementptr inbounds [3 x i32], [3 x i32]* %3, i64 0, i64 %16
      ```

      这三句计算数组a[i]的地址，然后再用store语句存入该地址
    - block18:for循环结束块，实现i自增：load->add->store，再跳回block9开始下一轮循环
- test3.c:结构体、if-else语句
  - 首先是为结构体做定义和为结构体变量分配空间：

    ```
    %struct.book = type { [50 x i8], i32 }

    %2 = alloca %struct.book, align 4
    ```

  - if语句结构：

    把变量i的值load到%4,然后做比较icmp。得到布尔值用在br指令中跳转
    - block6:if语句成立。此时需对结构体变量赋值

      `%7 = getelementptr inbounds %struct.book, %struct.book* %2, i32 0, i32 1`

      注意到这里因为是结构体，所以参数有两层偏移:`i32 0`指向结构体，`i32 1`才指向book_id这个内部成员。

      且注意到表达式`(3+4/2)&&1`的值在.ll中是计算好的，即直接`store i32 1, i32* %7, align 4`
    - block8：else语句分支，简单调用printf函数

3.人工翻译

```
//源代码
int fib(int n) {
  int r;
```

```
  if (n == 0)
    r = 0;
  else if (n == 1)
    r = 1;
  else
    r = fib(n - 1) + fib(n - 2);
  return r;
}
int main() {
  int x = 0;
  float n = 8;
  for (int i = 1; i < (int)n; ++i) {
    x += fib(i);
  }
  return x;
}
```

- 非一次性成功的片段：
  - `float n = 8;` 的赋值 `store float 8.000000e+00, float* %3, align 4`
  - 类型转换 `%8 = fptosi float %7 to i32`
  - 当然还有机器翻译里对应的开头、结尾的一些语句，以及对前驱块是什么的注释 `; preds = %1` 这种
- 翻译时的注意点：
  - 因为是仿照之前在第二问看过的llvm IR文件特点来人工翻译，所以在写的过程中有以下几点考量
    - 被调用函数里要为传进的参数分配空间
    - 为.c中声明了的变量分配空间
    - 除了load，store指令外，其余指令的操作均在寄存器上进行，即要用到对应内存中的变量值时，先load出来，再加、减等操作，用完要更新内存则再store
    - label和临时变量名称的分配问题：与llvm IR翻译一致，考虑按顺序分配，在同一个函数里label的号数与变量的号数不能重复。而且注意到，main函数里号数从1开始用。而其他函数里号数从x开始用(0到x-2号是参数，x-1号保留不用，是函数最开始初始化变量部分的号码，隐式占用。x根据参数个数变化)
- 结果验证:

```
michealleiemichealleiei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/llvm_pre_lab » lli fib
.ll
michealleiemichealleiei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/llvm_pre_lab » echo $?

33
michealleiemichealleiei-TUF-GAMING-FX504GE-FX80GE:~/snap/Compiler/llvm_pre_lab »
```

- 代码如下

```
define dso_local i32 @fib(i32 %0) #0 {
  %2 = alloca i32, align 4  ;存放传入的参数n
  %3 = alloca i32, align 4  ;为r分配内存
  store i32 %0, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = icmp eq i32 %4, 0;比较n==0
  br i1 %5, label %6, label %7;if语句判断并跳转

6:                                ; r=0的赋值
  store i32 0, i32* %3, align 4
  br label %19
```

```
7:                                ; else if语句的判断
  %8 = load i32, i32* %2, align 4
  %9 = icmp eq i32 %8, 1
  br i1 %9, label %10, label %11

10:                               ; r=1的赋值
  store i32 1, i32* %3, align 4
  br label %19

11:                               ;else语句块
  %12 = load i32, i32* %2, align 4
  %13 = sub nsw i32 %12, 1
  %14 = call i32 @fib(i32 %13);递归调用函数
  %15 = load i32, i32* %2, align 4
  %16 = sub nsw i32 %15, 2
  %17 = call i32 @fib(i32 %16);递归调用函数
  %18 = add nsw i32 %14, %17
  store i32 %18, i32* %3, align 4;把值存入r
  br label %19

19:
  %20 = load i32, i32* %3, align 4
  ret i32 %20
}


define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4  ;int x
  %2 = alloca float, align 4  ;int n
  %3 = alloca i32, align 4    ;int i
  store i32 0, i32* %1, align 4
  store float 8.000000e+00, float* %2, align 4
  store i32 1, i32* %3, align 4
  br label %4

4:
  %5 = load i32, i32* %3, align 4
  %6 = load float, float* %2, align 4
  %7 = fptosi float %6 to i32    ;(int)n的强制类型转换
  %8 = icmp slt i32 %5, %7    ;比较i与(int)n大小
  br i1 %8, label %9, label %17

9:                                ; for语句内部
  %10 = load i32, i32* %3, align 4
  %11 = call i32 @fib(i32 %10)  ;调用fib()函数
  %12 = load i32, i32* %1, align 4
  %13 = add nsw i32 %12, %11
  store i32 %13, i32* %1, align 4
  br label %14

14:                               ; for语句结尾，实现++i并跳转到for循环开头
  %15 = load i32, i32* %3, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %3, align 4
  br label %4

17:                               ;main结束，加载返回值x并return
  %18 = load i32, i32* %1, align 4
```

```
  ret i32 %18
}
```

4.请结合第2和3题涉及的IR特征，根据 LLVM IR及其构建 提供的线索，熟悉 LLVM IR的相关表示以及 IRBuilder 的工作逻辑，并总结你的理解。为日后开展实验做好准备。

- 熟悉LLVM IR的相关表示：LLVM IR 是 LLVM 所约定的、与硬件架构无关的中间代码表示。它通过一些抽象的指令来表达程序。LLVM 在定义了 IR 规范的同时也提供了用于表示和构建 IR 的库和相关数据结构。

  结合2,3问涉及的IR特征我们可以看到LLVM IR表示的一些特点，总结如下

  - RISC风格的三地址代码
  - SSA格式、无限的虚拟寄存器：可以看到.ll文件中无限制的使用新编号的变量
  - 强类型系统，所有value都有其自身的类型，需在指令里显示标识
  - c程序中变量的声明对应在LLVM IR中会为其分配空间(alloca)
  - 不可对内存空间直接op，只能通过load/store访问
  - 编号问题：在一个函数里，传入参数、内部变量、block的标记的标号是连续且无限的。
  - LLVM IR是由许多basicblock组成的，bb内部语句顺序执行，在其结尾由br或ret指令跳转到其他bb，与课上所讲控制流图概念相对应。
- IRBuilder工作逻辑
  - 概述 : IRBuilder是LLVM中专门提供用来生产Instruction命令的，提供了更加友好的封装，可用各种独立的接口创建各种 IR 指令,并将它们插入基本块中。
  - 具体工作方式
    - 类 IRBuilder及其超类中声明的各个 `Create*` 方法与 LLVM IR 的指令相对应，例如 `CreateRetVoid` 用于创建返回值为空的 `ReturnInst` 指令。
    - 在 LLVM 框架中，生成 BB 的接口是 `BasicBlock::Create` ，一个函数可能包含多个BB
    - 指定当前 BB , 可插入一些指令，如可调用 `builder.CreateAdd` 插入一条加指令 。每次插入新指令时，如果该指令会产生一个结果 ,则用于创建和插入指令的方法调用会有一个类型为 `llvm :: Value` 的返回值 , 用来代表对应的SSA Value
    - 当完成了一个BB后 , 调用 `builder.SetInsertPoint` 把当前 BB 设置为要跳转到的 BB
    - IRBuilder 类模板的实例可用于跟踪当前插入指令的位置
    - 在生成代码之前必须先设置好 Builder 对象,指明写入代码的位置

学习记录

- pkg-config是一个linux下的命令，用于获得某一个库/模块的所有编译相关的信息。
- LLVM（low level virtual machine）从本质上来说，是一个开源编译器框架，能够提供程序语言的编译期优化、链接优化、在线编译优化、代码生成。LLVM有两个特点：
  1. LLVM有一个特定指令格式的IR语言，我们可以通过书写Pass来对其IR进行优化。
  2. 可以作为多种语言的后端，提供与编程语言无关的优化和针对多种CPU的代码生成功能。LLVM项目是模块化和可重用的编译器和工具链技术的集合；

  LLVM是构架编译器(compiler)的框架系统，以C++编写而成。用于优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)。对开发者保持开放，并兼容已有脚本。
  - LLVM的组成部分

    LLVM主要由Clang前端、IR优化器（Pass）和LLVM后端构成。其功能分别是：
    - clang前端：将平台相关的源码生成与平台无关的IR（llvm Bitcode）。
    - IR优化器：主要对IR进行优化。

        - llvm后端：将优化后的IR转换为与平台相关的汇编代码或者机器码。
    - Clang前端：
      Clang前端以.c文件为输入，经语法词法分析后解析为抽象语法数，最后通过LLVM内联API变为LLVM IR。其功能为：词法分析器：把输入的程序代码切成token；语法分析器：接收token流解析为AST。
    - IR优化器：
      LLVM IR包含三种格式：一种是在内存中的编译中间语言；一种是硬盘上存储的二进制中间语言（以.bc结尾），最后一种是可读的中间格式（以.ll结尾）。这三种中间格式是完全相等的。LLVM IR是LLVM优化和进行代码生成的关键。根据可读的IR，我们可以知道再最终生成目标代码之前，我们已经生成了什么样的代码。我们通过Pass来对IR进行相应的优化。
    - LLVM后端：
      llvm clang编译器主要是将各平台源代码编译成与平台无关的IR指令集，这将支撑对IR的优化及转换操作，而llvm后端的主要工作是优化IR指令，并将这些与平台无关的IR指令转换成目标设备相关的指令。

参考文档

- https://blog.csdn.net/qq_42570601/article/details/107581608#store_552
- https://blog.csdn.net/rikeyone/article/details/100020145