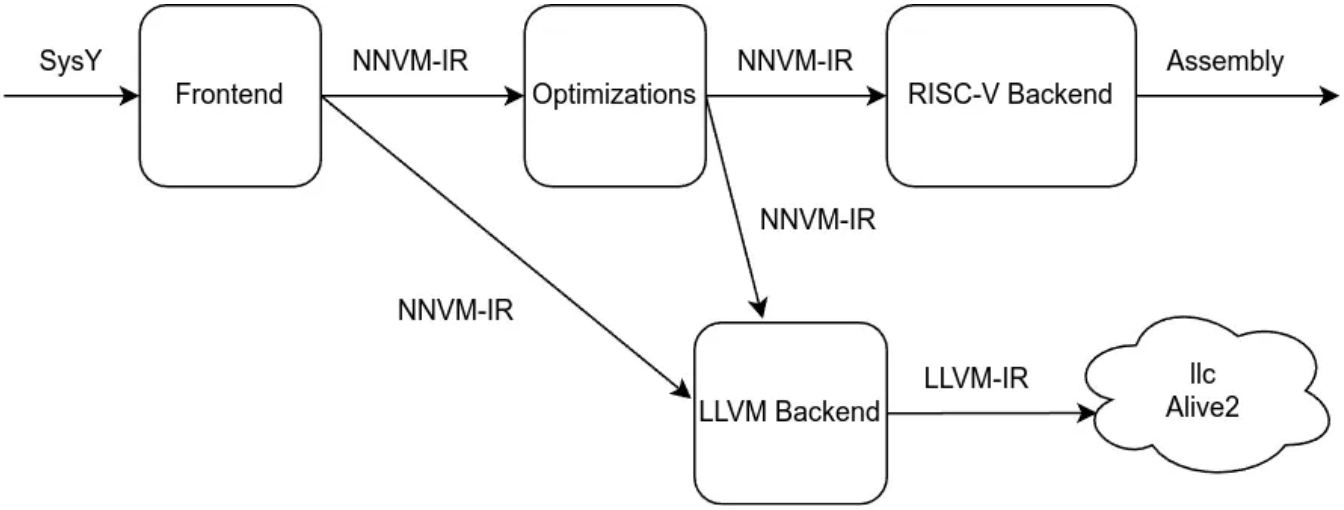
开发文档

1. 总体概要

项目名称：NNVM

取名启发自 LLVM，NNVM，全称 NJU (Nanjing University) New Virtual Machine，是一个基于 C++ 开发的可复用的优化编译器框架，其 workflow 如下：

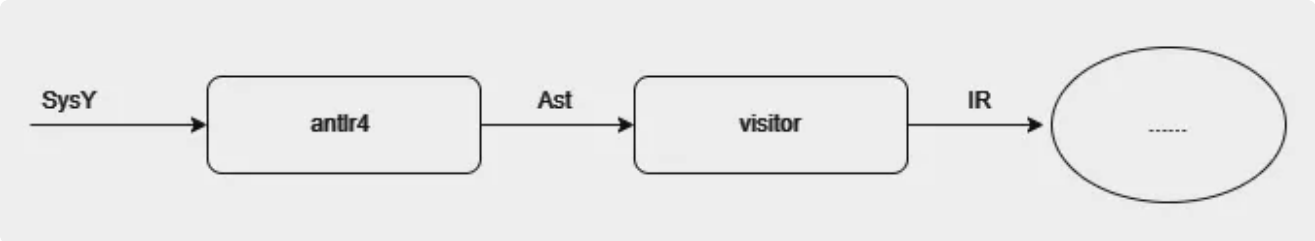


主流程由 "前端 – 中端优化 – RISC-V后端" 组成，是在比赛中运行的路线。

而 "前端 – LLVM后端" 与 "前端 – 中端优化 – LLVM后端" 两条路线共同用于差分测试，用于验证优化的正确性。

2. 前端

- 前端 workflow 如下:



- 为了利用 Csmith 进行测试，与 SysY2022 相比，额外支持以下语法
  - `for` 循环
  - `++/--` 运算符
  - `+= -= *= /= ...` 运算符
  - 扩展了 `expression` 概念
    - 赋值语句
    - 条件运算

        - ...

2.1. 词法分析, 语法分析

- 使用 `antlr4` 根据相应的 g4 文件进行，生成语法树

2.2. 错误输出

- 错误存储至 `list` 中，遍历完成后同一输出
- 错误内容：错误类型 + 错误位置
- 错误类型
    - 变量 & 函数 重复定义
    - 函数定义 & 声明 有误
    - 无效函数形参
    - 函数缺少返回值
    - 操作符左值 & 右值 错误
    - 变量无效引用
    - 操作符左值 & 右值类型不匹配

2.3. 语义分析

- 利用 `visitor` 模式遍历 `antlr4` 生成的语法树，调用 `IR's API` ，生成中端指令

3. NNVM-IR

采用SSA的形式，以下是几个主要的IR组成

3.1. Global variable

`global <type> <name> = <init>`

3.2. Function

```
<ret-type> <name> (<args>) {
<BasicBlocks>
}
```

3.3. Basic Block

A list of instructions, ending with only one terminator.

3.4. Arithmetic operator

- add, sub, mul, div, srem, urem
- fadd, fsub, fmul, fdiv
- icmp, fcmp
- and, or, xor, shl, lshr, ashl, ashr

3.5. Memory operator

- stack allocation:

  `%a = stack <number of bytes>`

- load:

  `%v = load <type>, <pointer>`

- store:

  `store <type> <value>, <pointer>`

- index of pointer:

  `%ptr_new = ptradd <pointer>, <number of bytes>`

3.6. Terminator

- Unconditional branch:

  `br <label1>`

- Conditional branch:

  `br <cond>, <label1>, <label2>`

- Return from function:

  `ret <value>`

3.7. Type

- void
- i1
- i8
- i32
- i64
- float
- ptr
- array

4. 中端优化 & 分析

4.1. 优化

我们参考了各种资料并设计/实现了一系列优化，包括：

- Mem2Reg，识别局部变量，将栈上的变量变为寄存器以及 ϕ 指令
- CSE (Common Sub-expression Elimination)，消除公共子表达式，我们的实现还包括了简单的 load elimination
- GVNHoist，将相同的指令提取到最近的共同支配位置，并合并为同一条指令
- TailElim，即尾递归调用优化，将尾递归转换成循环，减小函数调用的开销，同时使尾递归函数可以被完全内联
- MemProp，复用先前store或load的值，避免不必要的内存访问
- DeadStoreElim，删除没有被复用的dead stores
- ConstantFold，常量折叠，如"1+1" --> "2"
- LoopCanon，将循环转换为适合优化的标准形式，主要包括分割关键边，生成preheader等控制流形式的转化
- LICM，循环不变量提取，我们分别实现了 No-SideEffect 的指令的提取(如加减法等算数运算)，和 Store / Load 的不变性分析与提取

- LoopLoadStorePair，将循环中相同目标的load-store对提取到循环外，内部用phi代替
- StaticUnroll，将一些有固定迭代次数的循环完全展开，以暴露更多的优化机会
- Rotate，将 while 循环转化为 do – while 循环
- Combiner，实现了基本的算数优化，如"a * 2" --> "a << 1"
- CFGCombiner，实现了基本的控制流图化简，包括合并基本块，DCE(死代码消除)等优化
- Inliner，函数内联
- GlobalVarOpt，优化全局变量，包括优化编译单元内部的只读变量以及只写变量
- GlobalAttributor，识别纯函数等函数性质，有助于其它优化进行分析

4.2. 分析

同时我们还实现了一系列的基本分析，包括：

- DomTreeAnalysis，即支配树的分析，我们使用了Semi-NCA算法构建支配树
- PostDomTreeAnalysis，即后支配树的分析，由于infinite loop在后支配树中的特殊性，我们使用了简单的可达性算法来判断后支配以保证正确性
- LoopAnalysis，基于DomTreeAnalysis实现了循环以及子循环的识别，同时我们还额外分析了preheader，latch等循环的必要组成部分
- AliasAnalysis，即指针别名分析，精确地分析了指针之间可能的别名信息，以及不同内存对象的差异
- MemAccAnalysis，基于AA实现了关于内存指令间影响的分析，主要包括识别 Clobber, Def, Use 关系
- SCEV (Scalar Evolution)，基于 Recurrence 的数学模型分析循环induction variable的变化趋势
- LoopBoundAnalysis，基于 SCEV 分析 Trip Count 等循环运行的信息
- Liveness Analysis，后端关于寄存器的活跃变量分析，使用了 worklist 进行加速

4.3. TODOs

- SCCP, Loop Fusion, Induction Simplify, Layout Optimization

5. 后端代码生成

- **Lower**: 将 NNVM-IR 转化为 LowIR (更低一个级别的IR)
- **Instruction Selection**: 将指令合法化，并转化为更快的形式，包括展开DIV
- **Phi Resolution**: 翻译 Phi，解决 Last Copy Problem 和 Swap Problem.

- **Instruction Scheduling**: 重排列指令以最大化 ILP 并隐藏 内存延迟.
- **Register Allocation**: 将虚拟寄存器替换为实际的物理寄存器，我们分别实现了 Linear Scan 与 Graph Coloring 两个版本的寄存器分配
- **Peephole Optimizations:** 一些简单的窥孔优化
- **Stack Allocation**: 将对栈对象的引用替换为实际栈指针 sp 的引用，该过程要合理排列栈，同时还要插入 prologue 和 epilogue
- **Emit**: 将 LowIR 以汇编的形式输出

6. 测试

6.1. 概述

在整个开发周期中，团队自行编写用例以实现单元测试，利用官方提供的用例仓库与 `sylib` 库实现集成测试。此外，随着开发日趋完善，团队一直反复进行回归测试，并利用开源工具实现模糊测试、代码规约等。通过多种测试形式，团队得以及时发现NNVM的正确性问题和性能瓶颈，进而快速迭代编译器实现。

6.2. 自动化测试

根据编译器的测试需求，团队将多种测试功能集成于一个Python脚本，通过命令行参数将相对独立的功能结合应用。主要功能如下：

1. 编译运行执行官方测试用例，与官方提供的输出结果对比以判定正确性。
    a. 可在命令行传入正则表达式指定或排除特定路径名，使得用例选取更加灵活便利。
    b. 可通过命令行参数控制测试时的输出（包含brief、normal、verbose三个级别），根据需要精准获取不同测试信息。
    c. 对多个用例的测试可以并行，以提高测试效率。
2. 将SysY源码转为IR后传入LLVM完成后端处理，以单独测试编译器的前端实现。
3. 将NNVM编译程序运行的输出与GCC编译程序运行的输出而非官方提供的结果对比，实现 differential test，使得非官方测试用例也能快速得到测试。
4. 调用Csmith生成测试用例，经过脚本处理后传入NNVM编译运行，结合difftest实现模糊测试。

6.3. 其他工具

6.3.1. Csmith

开源工具Csmith可用于生成随机的C语言测试用例。通过参数调整、SysY语法拓展、正则表达式替换等方式，生成出的测试用例可通过NNVM编译，最终集成于测试脚本以实现模糊测试，使得测试更加充分。

6.3.2. Creduce

开源工具Creduce可用于规约测试用例。通过编写interestingness test脚本设定规约条件，测试用例中的正确部分可被缩减，使得更容易从所编译程序的failure溯源到相应的程序error，最终锁定编译器实现中的fault。

7. Assign

陈泓宇：RISC-V Backend & Middle-End Optimization

刘治元：Fuzzing & Test & Graph Coloring Register Allocation

徐天行：Frontend & Constant Fold

刘洪源：Error Report / Diagnostic in Frontend

8. Reference

1. Linear Scan Register Allocation: https://dl.acm.org/doi/10.1145/330249.330250
2. Loop Optimization: https://webdocs.cs.ualberta.ca/~amaral/thesis/ChristopherBartonMSc.pdf
3. Andrew W. Appel. Modern Compiler Implementation in Java[M]. Cambridge University Press, 1998.
4. Csmith: https://github.com/csmith-project/csmith
5. Creduce: https://github.com/csmith-project/creduce
6. Semi-NCA: https://qaqcxh.github.io/Blogs/graph%20theory/DominatorTheory.html
7. LLVM: https://github.com/llvm/llvm-project

   注：我们参考了 LLVM 的 IR 形式， 故我们 IR 的形式与 LLVM 相近，但实际上也有很多不同，比如我们去除了 GEP，而改为实际在字节层面位移的 ptradd，同样把基于类型的 alloca 改为了基于字节数的 stack ，同时我们只保留了必要的算数指令，并没有引进各种不必要的 intrinsics，这些改动帮助了我们进行优化与分析。在其它方面的设计，NNVM与LLVM几乎没有关系。优化层面我们采

用了更激进的策略，尝试了编译更慢但效果更好的方法，比如 LLVM 基于 MemorySSA 进行死赋值的消除以及一些内存量的传播，但我们并不建立中间的表示，而是直接基于别名分析多次扫描全部指令以确定赋值的可达性以及 Load 的可复用性，而 LLVM 后端采用了与机器无关的代码生成器，使用TableGen生成目标的信息，而 NNVM Backend 则专注于 RISC-V，在后端中有许多 ISA-specific 的元素，且 NNVM 后端的 IR 类似于维护 def-use 关系的三地址码，与 LLVM 的 SDAG 或者 GlobalISel 的设计完全不同，而且 LLVM 采用了 Greedy 的寄存器分配作为其主要分配手段，我们则分别参考了一些论文实现了线性扫描寄存器分配，并参考了《Modern Compiler Implementation in Java》 实现图着色寄存器分配，可以说在实现上，LLVM 聚焦于 spill 的更好，而我们则追求更少的 spill 。总之， NNVM 是完全不同于 LLVM 的一个编译器设计。

8. CMMC: https://github.com/dtcxzyw/cmmc/tree/main/cmmc

   我们参考了 CMMC 的测试方法，包括差分测试以及模糊测试，但测试的具体逻辑仍然是自己完成

9. Loop Regnition: https://blog.csdn.net/qq_43391414/article/details/110876772
10. SCEV: https://sbaziotis.com/compilers/introduction-to-scalar-evolution.html
11. List Scheduling：https://www.cs.cmu.edu/afs/cs/academic/class/15745-s11/public/lectures/L17-List-Scheduling.pdf