代码优化

功能入口相关类放在 `./optimize` 文件夹下，实际上具体代码实现都在 `./mid/IrBasicBlock.cpp` 中，因为优化需要以基本块为单位分析，处理各中间代码指令。最主要的优化为 mem2reg 和 寄存器分配。

设计部分

Mem2Reg

llvm 指令全是静态单赋值（SSA），但实际上没优化时，它对非数组变量都是通过 load 和 store 这样的内存操作，实现了伪静态单赋值。mem2Reg 就是消除非数组变量的 alloca、load、store 指令，并用变量的**到达定义**作为要使用的值；也就是说其他指令使用到达定义替换 load 的值。

但由于分支语句的存在，变量的到达定义可能有多种情况，因此我们引入了 `phi` 指令，格式类似 `%v_3 = phi i32 [ 1, %b_0 ], [ %v_1, %b_2]`，从不同的前驱基本块进入当前基本块，到达定义的选择也会不同。

这样完成到 `phi` 的转换后，生成的 llvm 中间代码就是标准的 SSA，从而可以进行其他优化。下面是具体的实现步骤：

准备阶段

需要建立各个基本块之间的关系，包括控制流程图（CFG, Control Flow Graph），严格支配关系、直接支配关系、支配边界、直接支配树。由于后续的优化可能有基本块删除和插入的操作，基本块关系需要**多次计算**。因此在计算前**先清空关系**。下面是基本块内存储的关系数据结构：

```
std::vector<IrBasicBlock *> preBlocks;  // CFG数据流图前驱节点
std::vector<IrBasicBlock *> sucBlocks;  // 后继节点
std::set<IrBasicBlock *> domers;  // 支配this的节点
IrBasicBlock *idomer;  // dom tree中直接支配this节点的节点
std::vector<IrBasicBlock *> idomees;  // this节点直接支配的节点
std::set<IrBasicBlock *> DF;  // 支配边界(phi辅助使用)

// 用于变量活性分析，有可能是IrInstr, IrGlobalVar, IrParam
std::set<IrValue *> def;
std::set<IrValue *> use;
std::set<IrValue *> in;
std::set<IrValue *> out;

/* 重复分析时要清空 */
void IrBasicBlock::clearCFG() {
    preBlocks.clear(); sucBlocks.clear();
    domers.clear(); idomer = nullptr; idomees.clear(); DF.clear();
    def.clear(); use.clear(); in.clear(); out.clear();
}
```

控制流程图：遍历所有基本块，根据基本块最后一个跳转指令就能确定；

删除不会到达的基本块：得到控制流图后，可以 DFS 得到所有能从入口基本块到达的基本块，除此之外的基本块无法到达，需要删除，否则会影响后续分析；

计算支配关系：从函数的入口基本块开始，一个基本块的支配者是它的前驱基本块的支配者的交再加上自身；如果它的某个前驱基本块还没计算支配者，则它的支配者集合当做全集处理。如果当前基本块的支配者集合更新，则更新它的所有后继基本块的支配者集合。

计算直接支配关系：直接利用定义：严格支配n，且不严格支配任何严格支配n的节点的节点；

计算支配边界（DF, Dominance Frontier）：也是为后面插入 `phi` 做准备的最重要的步骤。使用现成算法，代码实现如下：

```
void IrBasicBlock::genDF() {
    IrBasicBlock *domer = this;
    for (auto suc : sucBlocks) {
        // 有概率domer==suc
        while (domer && !domer->strictlyDom(suc)) {
            domer->DF.insert(suc);
            domer = domer->idomer;
        }
    }
}
```

插入 `phi` 阶段

原 llvm 代码的 `store` 就是对应 `alloca` 变量的到达定义，它所在基本块的支配节点都可以直接使用 store 的值作为到达定义，但支配边界的那些基本块可能有多个到达定义，我们需要在这些地方插入 `phi`，插入的 `phi` 也是新的该变量的到达定义，从而需要继续根据支配边界插入 `phi`。具体代码实现如下：

```
void IrBasicBlock::insertPhis() {
    for (auto it = instrs.begin(); it != instrs.end();/* ++it*/) {
        IrInstr *instr = *it;
        /* 对每个变量:
         * 找到所有的定义/重新赋值block
         * 然后用算法在这些block的递归DF block首部插入phi
         * 最后对所有block的每个load,store,phi,alloca指令重命名
         * 过程中要记录(最新)到达定义
         * 并将后继block中phi更新填充
         * 然后遍历func所有block进行重命名 */
        if (instr->getIrInstrType() == IrInstrType::ALLOCA &&
                dynamic_cast<IrAllocaInstr *>(instr)->getTargetType() ==
IrValueType::INT32) {

            // 找到所有的定义/重新赋值block
            std::set<IrBasicBlock *> storeBBs;
            for (IrUser *t : instr->getUsers()) {
                auto *user = dynamic_cast<IrInstr *>(t);
                assert(user != nullptr);
                if (user->getIrInstrType() == IrInstr::IrInstrType::STORE) {
                    storeBBs.insert(user->getInBB());
                }
            }

            // 用算法在各个block(有store的和store的DFs和继续递归)首部插入phi
            std::set<IrBasicBlock *> addedBBs;
            std::set<IrBasicBlock *> defBBs(storeBBs.begin(),
storeBBs.end());
```

```
27
28              while (!defBBs.empty()) {
29                  auto _it = defBBs.begin();
30                  IrBasicBlock *defBB = *_it;
31                  defBBs.erase(_it);
32                  for (auto df : defBB->DF) {
33                      if (addedBBs.find(df) == addedBBs.end()) {
34                          new IrPhiInstr(IrBuilder::getInstance()-
>genLocalVarName(df),
35                                         df, instr);
36                          addedBBs.insert(df);
37                          if (storeBBs.find(df) == storeBBs.end()) {
38                              defBBs.insert(df);
39                          }
40                      }
41                  }
42              }
43
44              // 按支配树上的dfs顺序对所有block的每个load,store,phi,alloca指令重命名
45              inFunc->rename2SSA(instr);
46              it = instrs.erase(it);
47          } else {
48              ++it;
49          }
50      }
51  }
```

插入的逻辑在 24~42 行，对应下面的算法：

Algorithm 3.1: Standard algorithm for inserting ϕ-functions

```
1  for v: variable names in original program do
2      F ← {}                          ▷ set of basic blocks where φ is added
3      W ← {}                          ▷ set of basic blocks that contain definitions of v
4      for d ∈ Defs(v) do
5          let B be the basic block containing d
6          W ← W ∪ {B}
7      while W ≠ {} do
8          remove a basic block X from W
9          for Y: basic block ∈ DF(X) do
10             if Y ∉ F then
11                 add v ← φ(...) at entry of Y
12                 F ← F ∪ {Y}
13                 if Y ∉ Defs(v) then
14                     W ← W ∪ {Y}
```

变量重命名阶段

也就是删除 `alloca` `load` `store` 指令；过程中记录变量的（最新）到达定义，并将后继基本块中的 `phi` 指令更新填充；最后继续按照支配树 DFS 遍历重命名。实现代码如下：

```
1  void IrBasicBlock::rename2SSA(IrInstr *alloca, IrValue *newV) {
2      IrValue *reachDef = newV;
3      for (auto it = instrs.begin(); it != instrs.end();/* ++it*/) {
4          IrInstr *instr = *it;
```

```
        if (instr->is(IrInstr::IrInstrType::PHI) && instr->defAlloca(alloca)) {
            reachDef = instr;
        } else if (instr->is(IrInstr::IrInstrType::STORE) && instr->defAlloca(alloca)) {
            reachDef = instr->getOperand1();
            it = instrs.erase(it);
        } else if (instr->is(IrInstr::IrInstrType::LOAD) && instr->useAlloca(alloca)) {

            instr->repValsInUsers(reachDef);
            it = instrs.erase(it);
        } else if (instr == alloca) {
            // 交给insertPhis做
//            it = instrs.erase(it);
        }
        // 如果没有删除，则手动++it
        if (instr == *it) ++it;
    }
    // 对后继block的phi更新填充
    for (auto block : sucBlocks) {
        for (auto instr : block->instrs) {
            if (instr->is(IrInstr::IrInstrType::PHI) && instr->defAlloca(alloca)) {
                auto *phi = dynamic_cast<IrPhiInstr *>(instr);
                assert(phi != nullptr);
                phi->addOption(reachDef, this);
                break;
            }
        }
    }

    for (auto block : idomees) {
        block->rename2SSA(alloca, reachDef);
    }
}
```

后端消除 Phi

`phi` 并不能直接转为 MIPS 指令，需要最终转为 `move` 指令。

`phi` 转 `pcopy` 阶段

`phi` 之间是并行赋值，对于 `%v_3 = phi i32 [ 1, %b_0 ], [ %v_1, %b_2]`，可以在 `%v_3` 所在基本块前驱基本块各放置这样的赋值语句：`move %v_3, 1` 和 `move %v_3, %v_1`。如果当前基本块有多个 `phi` 指令，各前驱基本块也要放置多个 `move` 指令。`pcopy` 就是暂时存储这些 `move` 指令，之后这些 `move` 指令之间还会因为并行赋值下遇到循环赋值有冲突，需要在 `pcopy` 转 `move` 的步骤中处理。

特别的，如果某个前驱基本块有多个后继基本块，我们需要插入一个中间基本块在当前块和原本的前驱基本块之间。

`pcopy` 转 `move` 阶段

这里就要解决循环赋值的问题。对于下面的 `move` 序列：

```
move %v_0, %v_1
move %v_2, %v_0
```

我们需要将 `%v_2` 赋值为 `%v_0` 被修改前的值，因此需要做如下修改：

```
move %v_0_tmp, %v_1
move %v_0, %v_1
move %v_2, %v_0_tmp
```

下面是代码的具体实现：

```
/* 将pcopy存储的多个<phi, value>对变为多个move, 最后删除pcopy
 * 由于pcopy是并行赋值，因此要解决move中循环赋值的问题 */
void IrBasicBlock::pcopy2Move() {
    if (instrs.size() < 2 || !instrs.at(instrs.size() - 2)->is(IrInstrType::PCOPY)) return;

    IrBuilder::getInstance()->setCurFunc(inFunc);  // localVarName
    auto pcopy = dynamic_cast<IrPcopyInstr *>(instrs.at(instrs.size() - 2));
    std::vector<IrValue *> dstList = pcopy->getDstList(), srcList = pcopy->getSrcList();
    std::vector<IrMoveInstr *> moveList;
    // 注意pcopy原本是并行赋值
    // 应该再将用dst作为src的改为用dst_tmp
    for (int i = 0; i < srcList.size(); ) {
        IrValue *dst = dstList.at(i), *midV = nullptr;
        for (auto src : srcList) {
            // 如果有loopAssign，新插入a'<-a
            if (src == dst) {
                midV = new IrValue(IrValueType::INT32, dst->getName() + "_tmp");
                auto move = new IrMoveInstr(IrBuilder::getInstance()->genLocalVarName(this), midV, dst, this);
                moveList.push_back(move);
                break;
            }
        }
        auto move = new IrMoveInstr(IrBuilder::getInstance()->genLocalVarName(this), dst, srcList.at(i), this);
        moveList.push_back(move);
        // 从pcopy列中移除
        srcList.erase(srcList.begin()), dstList.erase(dstList.begin());
        // 后续的也替换
        if (midV != nullptr) {
            for (auto &src_tmp : srcList) {
                if (src_tmp == dst) {
                    src_tmp = midV;
                }
            }
        }
    }
    // 用moveList的指令代替pcopy指令
```

```
37      instrs.erase(instrs.end() - 2);
38      for (auto move : moveList) {
39          instrs.insert(instrs.end() - 1, move);
40      }
41  }
```

这里严格执行了下面的算法：

Algorithm 3.6: Replacement of parallel copies with sequences of sequential copy operations.

```
1  let pcopy denote the parallel copy to be sequentialized
2  let seq = () denote the sequence of copies
3  while ¬[∀(b ← a) ∈ pcopy, a = b] do
4      if ∃(b ← a) ∈ pcopy s.t. ∄(c ← b) ∈ pcopy then        ▷ b is not live-in of pcopy
5          append b ← a to seq
6          remove copy b ← a from pcopy
7      else                                                  ▷ pcopy is only made-up of cycles; Break one of them
8          let b ← a ∈ pcopy s.t. a ≠ b
9          let a′ be a freshly created variable
10         append a′ ← a to seq
11         replace in pcopy b ← a into b ← a′
```

寄存器分配

由于时间不够，没有使用图着色，而是实现更简单的线性扫描。也就是对基本块根据当前寄存器分配情况继续分配寄存器。

在这种方法中，代码不会变成图形。取而代之的是，对所有变量进行线性扫描以确定其实时范围，以区间表示。一旦确定了所有变量的实时范围，就会按时间顺序遍历区间。尽管这种遍历可以帮助识别其实时范围会干扰的变量，但不会构建干扰图，并且变量是以贪婪的方式分配的。

遗憾的是，这个方法目前还有 bug 还没改完，在代码生成二中只能得到 98 分，在竞速中也只能过 4 个点。

其他优化

简化常量计算

直接在中间代码的 `IrAlueInstr` 类里使用函数 `IrValue *IrAluInstr::getSimplify()` 得到简化后的返回值，如果和原本的 `IrAluInstr` 值不同，则将所有使用该指令的指令，替换使用的 value 为简化后的值。

简化的范围有：两个常量计算；只有一个常量，并且该常量为 0 或 1。

死代码删除

如果一个指令不能被删除，则它使用的指令也不能被删除，依此递归。需要注意的是，如果没有详细分析，数组变量的 `store` 也不能被删除。

编码完成后的修改

有一些优化前没有 bug，但优化后会出现 bug 的地方。究其根本，是 `move` 指令的引入，导致我采用之前的顺序（从函数的入口基本块开始，无论是支配树前序 DFS 还是根据 CFG 前序 DFS）遍历时，可能出现使用的 value 还没定义，从而没有在栈里分配空间。

问题 1：调用函数时如何保存寄存器

如果直接将使用的寄存器以及 `$sp` 和 `$ra` 寄存器保存后，加载变量到形参里（`$a1~$a3` 以及被调用函数的栈空间），可能需要 `load` 加载传入的值。我之前的策略是如果传入的值没有在栈里分配空间，则在这里为其分配空间。但此时分配空间将会分配到保存寄存器的栈空间的后面，也就是 `$sp` 和 `$ra` 在栈里的后面的位置，这明显是错误的。

我的处理是将 `$a1~$a3` 的值保存到当前函数栈底，也就是保存从右到左的参数的地方（一般从第 4 个或第 5 个参数开始才保存到栈里的这个位置，不过我们还是为前几个参数分配了栈里的空间）。接下来加载函数传参。最后保存其他寄存器。代码的实现也比较丑陋了：

```
1  void MipsBuilder::storeRegs(std::vector<IrValue *> args) {
2      std::vector<IrValue *> &params = MipsBuilder::getInstance()-
   >curFuncParams;
3      std::map<IrValue *, Register> &v2r = MipsBuilder::getInstance()-
   >var2reg;
4      std::set<Register> regs;
5      for (const auto &pair : v2r) {
6          if (pair.second >= Register::A1 && pair.second <= Register::A3)
   continue;
7          regs.insert(pair.second);
8      }
9      // 先保存$a1~$a3
10     for (int i = 0; i < std::min(3, (int)params.size()); ++i) {
11         new MemAsm(MemAsm::Op::SW, MipsBuilder::getRegOf(params.at(i)),
   Register::SP, getPosInStack(params.at(i)));
12     }
13     // 然后传实参
14     for (int i = 0; i < std::min(3, (int)args.size()); ++i) {
15         Register targetReg = RegTool::getReg(Register::A1, i);
16         Register argReg = MipsBuilder::genRegOf(args.at(i), targetReg);
17         if (argReg == targetReg) continue;  // 如果是一样的a寄存器，或者已经赋值给
   a寄存器，则不用再赋值了
18         else if (argReg >= Register::A1 && argReg <= Register::A3) {  // 如果
   实参寄存器argReg是其他的$ai寄存器，需要LW
19             new MemAsm(MemAsm::Op::LW, targetReg,
   Register::SP,MipsBuilder::getPosInStack(args.at(i)));
20         } else {  // 如果是其他寄存器，如$t1,$s0
21             new MoveAsm(targetReg, argReg);
22         }
23     }
24     // 继续保存寄存器
25     int offset = MipsBuilder::getCurStackOffset();
26     for (auto reg : regs) {
27         offset -= 4;
28         new MemAsm(MemAsm::Op::SW, reg, Register::SP, offset);
29     }
30     // 最后保存$sp和$ra
31     new MemAsm(MemAsm::Op::SW, Register::SP, Register::SP, offset - 4);
32     new MemAsm(MemAsm::Op::SW, Register::RA, Register::SP, offset - 8);
```

```
    MipsBuilder::getInstance()->stackOffset = offset - 8;
    // SW剩下的实参
    offset = MipsBuilder::getCurStackOffset();
    for (int i = (int)args.size() - 1; i > 2; --i) {
        offset -= 4;
        Register paramReg = MipsBuilder::genRegOf(args.at(i), Register::K0);
        if (paramReg >= Register::A1 && paramReg <= Register::A3) {  // 如果call传入的实参是当前函数的实参，需要LW
            new MemAsm(MemAsm::Op::LW, (paramReg = Register::K0), Register::SP, MipsBuilder::getPosInStack(args.at(i)));
        }
        new MemAsm(MemAsm::Op::SW, paramReg, Register::SP, offset);
    }
}
```

问题 2：为指令分配栈空间的问题

前面提到，`%v_3 = add i32 %v_1, %v_2` 时如果还没遍历到 `%v_1` 定义的地方，意味着暂时没有为它分配栈空间。因此会在解析 `%v_3` 指令时为 `%v_1` 分配栈空间。

但到了 `%v_1` 定义时，如果再为其分配栈空间，就会出现变量和栈内偏移不一致的情况，**表现为该输出正常数字时输出 0**。这是有很低的概率会发生的，我评测遍了 github 上学长留下的几乎所有样例也没找到 bug，最后还是运气成分偏多猜到了 bug 原因。

代码实现原本为：

```
// 如果没有为该值分配寄存器，则将该寄存器的值存/更新到栈上
if (MipsBuilder::getRegOf(this) == Register::ZERO) {
    int offset = MipsBuilder::getPosInStack(this);
    new MemAsm(MemAsm::Op::SW, targetReg, Register::SP, offset);
}
```

修改后为：

```
// 如果没有为该值分配寄存器，则将该寄存器的值存/更新到栈上
if (MipsBuilder::getRegOf(this) == Register::ZERO) {
    int offset = MipsBuilder::getPosInStack(this);
    if (offset > 0) {
        MipsBuilder::allocaMem2Var(this, 4);
        offset = MipsBuilder::getCurStackOffset();
    }
    new MemAsm(MemAsm::Op::SW, targetReg, Register::SP, offset);
}
```