Project 4 Code Optimization

21307035 邓栩瀛

常量传播

`ConstantFolding.hpp`

`ConstantFolding.cpp`

1、建立变量到常量的映射：维护一个映射表，将变量和其确定的常量值关联起来。

```
std::unordered_map<Value*, Constant*> constMap;
```

2、替换常量：当确定一个变量为常量时，用该常量值替换程序中的所有使用该变量的地方。

3、重复直到收敛：多次迭代，直到不再发现新的常量传播为止。

处理二元运算指令：检查操作数是否在常量映射表中，如果是，则将该指令替换为常量计算结果

```
      if (auto* binOp = dyn_cast<BinaryOperator>(&inst)) {
        Value* lhs = binOp->getOperand(0);
        Value* rhs = binOp->getOperand(1);

        if (constMap.count(lhs) && constMap.count(rhs)) {
          ConstantInt* constLhs = dyn_cast<ConstantInt>(constMap[lhs]);
          ConstantInt* constRhs = dyn_cast<ConstantInt>(constMap[rhs]);

          if (constLhs && constRhs) {
            switch (binOp->getOpcode()) {
              case Instruction::Add:
                binOp->replaceAllUsesWith(ConstantInt::getSigned(
                    binOp->getType(), constLhs->getSExtValue() + constRhs->getSExtValue()));
                break;
              case Instruction::Sub:
                binOp->replaceAllUsesWith(ConstantInt::getSigned(
                    binOp->getType(), constLhs->getSExtValue() - constRhs->getSExtValue()));
                break;
              case Instruction::Mul:
                binOp->replaceAllUsesWith(ConstantInt::getSigned(
                    binOp->getType(), constLhs->getSExtValue() * constRhs->getSExtValue()));
                break;
              case Instruction::UDiv:
              case Instruction::SDiv:
                binOp->replaceAllUsesWith(ConstantInt::getSigned(
                    binOp->getType(), constLhs->getSExtValue() / constRhs->getSExtValue()));
                break;
              case Instruction::URem:
              case Instruction::SRem:
                binOp->replaceAllUsesWith(ConstantInt::getSigned(
                    binOp->getType(), constLhs->getSExtValue() % constRhs->getSExtValue()));
```

```
                break;
              default:
                break;
            }
            constFoldTimes++;
          }
        }
      }
```

处理存储指令：如果存储的值是常量，则更新常量映射表

```
if (auto* storeInst = dyn_cast<StoreInst>(&inst)) {
  Value* ptrOperand = storeInst->getPointerOperand();
  Value* valOperand = storeInst->getValueOperand();

  if (auto* constVal = dyn_cast<ConstantInt>(valOperand)) {
    constMap[ptrOperand] = constVal;
  }
}
```

常量折叠

ConstantPropagation.hpp

ConstantPropagation.cpp

1、遍历模块中的所有函数，然后遍历每个函数中的所有基本块，再遍历每个基本块中的所有指令

2、检查当前指令是否为二元运算指令（加、减、乘、除、取余）

```
if (auto binOp = dyn_cast<BinaryOperator>(&inst))
```

3、检查操作数是否为常量，如果二元运算指令的左右操作数都是常量，则进行常量折叠

```
            Value* lhs = binOp->getOperand(0);
            Value* rhs = binOp->getOperand(1);
            auto constLhs = dyn_cast<ConstantInt>(lhs);
            auto constRhs = dyn_cast<ConstantInt>(rhs);
```

4、根据运算类型（如加法、减法等）计算出结果，并将二元运算指令替换为计算出的常量值

5、删除冗余指令

```
      for (auto& i : instToErase)
        i->eraseFromParent();
```

强度削弱

StrengthReduction.hpp

StrengthReduction.cpp

1、乘法优化

```
if (auto* mul = dyn_cast<BinaryOperator>(&inst)) {
        if (match(mul, m_Mul(m_Value(), m_ConstantInt()))) {
          Value* lhs = mul->getOperand(0);
          ConstantInt* rhs = cast<ConstantInt>(mul->getOperand(1));
```

- 如果乘数是2的幂次方，可以用左移操作（shl）替代乘法操作。

```
if (rhs->getValue().isPowerOf2()) {
   auto* newInst = BinaryOperator::CreateShl(lhs, ConstantInt::get(rhs->getType(), rhs-
>getValue().logBase2()), "", &inst);
   mul->replaceAllUsesWith(newInst);
   instToErase.push_back(mul);
   ++strengthReductionCount;
}
```

- 对于非2的幂次方，通过位运算和加法结合的方式进行优化，减少乘法的使用

```
APInt rhsValue = rhs->getValue();
Value* result = nullptr;
bool first = true;
for (unsigned i = 0; i < rhsValue.getBitWidth(); ++i) {
    if (rhsValue[i]) {
       Value* shift = lhs;
       if (i > 0) {
          shift = BinaryOperator::CreateShl(lhs, ConstantInt::get(rhs->getType(), i), "",
&inst);
       }
       if (first) {
          result = shift;
          first = false;
       } else {
          result = BinaryOperator::CreateAdd(result, shift, "", &inst);
       }
     }
   }
if (result) {
   mul->replaceAllUsesWith(result);
   instToErase.push_back(mul);
   ++strengthReductionCount;
}
```

2、取余优化

- 将取余运算转换为除法和减法的组合

```
if (auto* rem = dyn_cast<BinaryOperator>(&inst)) {
    if (match(rem, m_URem(m_Value(), m_ConstantInt()))) {
        Value* lhs = rem->getOperand(0);
        ConstantInt* rhs = cast<ConstantInt>(rem->getOperand(1));
        auto* newInst = BinaryOperator::CreateUDiv(lhs, rhs, "", &inst);
        auto* subInst = BinaryOperator::CreateSub(lhs, BinaryOperator::CreateMul(newInst,
rhs, "", &inst), "", &inst);
        rem->replaceAllUsesWith(subInst);
        instToErase.push_back(rem);
        ++strengthReductionCount;
    }
}
```

3、除法优化

```
if (auto* div = dyn_cast<BinaryOperator>(&inst)) {
    if (match(div, m_UDiv(m_Value(), m_ConstantInt()))) {
        Value* lhs = div->getOperand(0);
        ConstantInt* rhs = cast<ConstantInt>(div->getOperand(1));
```

- 如果除数是2的幂次方，可以用右移操作（shr）替代除法操作。

```
if (rhs->getValue().isPowerOf2()) {
   auto* newInst = BinaryOperator::CreateLShr(lhs, ConstantInt::get(rhs->getType(), rhs-
>getValue().logBase2()), "", &inst);
    div->replaceAllUsesWith(newInst);
    instToErase.push_back(div);
    ++strengthReductionCount;
}
```

- 对于非2的幂次方，通过乘法和右移操作结合，利用预先计算好的魔数（magic number）和位移数（shift）来进行优化

```
uint64_t rhsValue = rhs->getValue().getZExtValue();
StrengthReduction::MagicInfo magicInfo = computeMagicAndShift(rhsValue);
auto* mulMagic = BinaryOperator::CreateMul(lhs, ConstantInt::get(rhs->getType(),
magicInfo.magic), "", &inst);
auto* newInst = BinaryOperator::CreateLShr(mulMagic, ConstantInt::get(rhs->getType(),
magicInfo.shift), "", &inst);
instToErase.push_back(div);
++strengthReductionCount;
```

代数恒等式

`AlgebraicIdentities.hpp`

`AlgebraicIdentities.cpp`

1、对于每个基本块中的每条指令，检查是否是二元运算指令（如加法、减法、乘法、除法等）

```
if (auto binOp = dyn_cast<BinaryOperator>(&inst))
```

2、获取二元运算指令的左右操作数，并尝试转换为常整数

```
Value* lhs = binOp->getOperand(0);
Value* rhs = binOp->getOperand(1);
auto constLhs = dyn_cast<ConstantInt>(lhs);
auto constRhs = dyn_cast<ConstantInt>(rhs);
bool simplified = false;
Value* newVal = nullptr;
```

3、应用代数恒等式

加法：包括x + 0 = x和0 + x = x两种情况

```
case Instruction::Add: {
     if (constRhs && constRhs->isZero()) {
         newVal = lhs;  // x + 0 = x
         simplified = true;
     } else if (constLhs && constLhs->isZero()) {
         newVal = rhs;  // 0 + x = x
         simplified = true;
     }
     break;
}
```

减法：x-0=x

```
case Instruction::Sub: {
     if (constRhs && constRhs->isZero()) {
         newVal = lhs;  // x - 0 = x
         simplified = true;
     }
     break;
}
```

乘法：包括x * 1 = x、1 * x = x、x * 0 = 0和0 * x = 0四种情况

```
case Instruction::Mul: {
     if (constRhs && constRhs->isOne()) {
         newVal = lhs;  // x * 1 = x
         simplified = true;
     } else if (constLhs && constLhs->isOne()) {
         newVal = rhs;  // 1 * x = x
         simplified = true;
     } else if (constRhs && constRhs->isZero()) {
         newVal = rhs;  // x * 0 = 0
         simplified = true;
     } else if (constLhs && constLhs->isZero()) {
         newVal = lhs;  // 0 * x = 0
         simplified = true;
     }
     break;
}
```

除法：x / 1 = x

```
case Instruction::UDiv:
case Instruction::SDiv: {
     if (constRhs && constRhs->isOne()) {
         newVal = lhs;  // x / 1 = x
         simplified = true;
      }
      break;
}
```

取余运算：$x \% 1 = 0$

```
case Instruction::URem:
case Instruction::SRem: {
if (constRhs && constRhs->isOne()) {
    newVal = ConstantInt::get(binOp->getType(), 0);  // x % 1 = 0
    simplified = true;
    }
    break;
}
```

3、替换和删除指令

```
if (simplified && newVal) {
    binOp->replaceAllUsesWith(newVal);
    instToErase.push_back(binOp);
    ++algebraicIdTimes;
}
for (auto& i : instToErase)
     i->eraseFromParent();
```

指令合并

InstructionCombining.hpp

InstructionCombining.cpp

1、遍历函数和基本块

```
for (auto& func : mod) {
    for (auto& bb : func) {
        if (combineInstructions(bb)) {
            ++combineTimes;
        }
    }
}
```

2、在给定的基本块中查找并尝试合并二元操作符指令，并调用 combineBinaryOperator 方法来检查和执行实际的合并操作

```
bool InstructionCombining::combineInstructions(BasicBlock& bb) {
    bool changed = false;
```

```
    std::vector<Instruction*> instToErase;

    for (auto it = bb.begin(), end = bb.end(); it != end; ++it) {
        if (auto* binOp = dyn_cast<BinaryOperator>(&*it)) {
            if (combineBinaryOperator(binOp, bb, instToErase)) {
                changed = true;
            }
        }
    }

    for (auto* inst : instToErase) {
        inst->eraseFromParent();
    }

    return changed;
}
```

3、合并具体二元操作符 combineBinaryOperator：根据二元操作符的类型（如加法、减法、乘法等）来尝试合并，如果发生了合并，则会更新 changed 标志，同时将需要删除的指令收集到 instToErase 向量中，在后续步骤中删除这些指令

```
bool InstructionCombining::combineBinaryOperator(BinaryOperator* binOp, BasicBlock& bb,
std::vector<Instruction*>& instToErase) {
    // 合并操作的具体实现，详见InstructionCombining.cpp文件
}
```

公共子表达式消除

CommonSubexpressionElimination.hpp

CommonSubexpressionElimination.cpp

1、建立表达式映射：对于每个基本块中的指令，如果是二元操作符，将其操作数和操作码构建为一个表达式对象 Expression

```
if (auto binOp = dyn_cast<BinaryOperator>(&inst)) {
    std::vector<Value*> operands;
    for (unsigned i = 0; i < binOp->getNumOperands(); ++i) {
        operands.push_back(binOp->getOperand(i));
    }
Expression expr{binOp->getOpcode(), binOp->getType(), operands};
```

2、查找与替换：使用 exprTable 哈希表存储已经遇到的表达式及其对应的指令，如果当前表达式已存在于哈希表中，说明这是一个公共子表达式，可以直接用之前的结果替换当前指令

```
auto it = exprTable.find(expr);
if (it != exprTable.end()) {
    // 公共子表达式已经存在，进行替换
    binOp->replaceAllUsesWith(it->second);
    instToErase.push_back(binOp);
    ++cseTimes;
} else {
    // 公共子表达式不存在，插入表中
    exprTable[expr] = binOp;
}
```

死代码消除

DeadCodeElimination.hpp

DeadCodeElimination.cpp

1、死代码识别和移除：从模块中的每个函数开始遍历，然后检查每个指令是否为死代码，如果指令没有副作用且没有被使用，则被认定为死代码，将其收集到 instToErase 中

```
for (auto& func : mod) {
  for (auto& inst : instructions(func)) {
    if (!hasSideEffects(inst) && !isUsedOutsideOfDef(inst)) {
      instToErase.push_back(&inst);
    }
  }
}
for (auto* inst : instToErase) {
  inst->eraseFromParent();
  ++deadCodeCount;
}
```

2、检查指令是否具有副作用，可能是 volatile 的加载/存储、调用或其他具有副作用的操作，或者是终结指令

```
bool DeadCodeElimination::hasSideEffects(const Instruction& inst) const {
  return inst.mayHaveSideEffects() || inst.isTerminator();
}
```

3、检查指令是否被外部使用，即指令的使用是否为空

```
bool DeadCodeElimination::isUsedOutsideOfDef(const Instruction& inst) const {
  return !inst.use_empty();
}
```

死存储消除

DeadStorageElimination.hpp

DeadStorageElimination.cppp

1、识别和处理未使用的局部变量（alloca 指令）

```
if (auto alloca = dyn_cast<AllocaInst>(&inst))
```

2、检查是否所有使用都是无效的

```
bool allUsesDead = true;
for (auto& use : alloca->uses()) {
     Instruction* userInst = dyn_cast<Instruction>(use.getUser());
     if (!userInst) continue;
```

3、检查使用方式是否是无效的

```
        if (isa<StoreInst>(userInst) || isa<LoadInst>(userInst) || userInst->isTerminator()) {
            if (isUsedMeaningfully(userInst)) {
                allUsesDead = false;
                break;
            }
        } else {
            allUsesDead = false;
            break;
        }
```

4、如果所有使用都无效，则将其标记为待删除

```
if (allUsesDead) {
   // 收集要删除的指令
   for (auto& use : alloca->uses()) {
        Instruction* userInst = dyn_cast<Instruction>(use.getUser());
        if (userInst) instToErase.push_back(userInst);
   }
   instToErase.push_back(alloca);
   ++deadStorageElimTimes;
}
```

循环无关变量移动

LICM.hpp

LICM.cpp

1、构建支配树和循环信息：支配树用于确定基本块之间的支配关系，循环信息则提供了关于函数中所有循环的信息。

```
DominatorTree dt(func);
LoopInfo li(dt);
```

2、处理每个循环，获取循环的预头基本块，如果没有preheader基本块则跳过该循环

```
for (auto *loop : li) {
    if (!loop->isInnermost())
        continue;
    // ...
}
```

3、查找循环preheader

```
BasicBlock *preheader = loop->getLoopPreheader();
if (!preheader)
    continue;
```

4、收集循环出口基本块

```
SmallVector<BasicBlock *, 8> exitBlocks;
loop->getExitBlocks(exitBlocks);
```

5、对循环内每个基本块执行LICM

```
for (auto *bb : loop->getBlocks()) {
    if (bb == preheader)
        continue;
    // ...
}
```

6、检查并移动不变代码：在每个基本块中，遍历指令，检查是否可以被移动到preheader基本块以优化性能

```
for (auto &I : *bb) {
    // 检查指令是否为循环不变量
    if (I.isBinaryOp() || I.isCast() || I.isShift() || isLogicalOp(I.getOpcode())) {
        bool isInvariant = true;
        // 检查操作数是否为循环不变量
        for (Use &U : I.operands()) {
            Value *v = U.get();
            Instruction *opInst = dyn_cast<Instruction>(v);
            if (opInst && loop->contains(opInst->getParent())) {
                isInvariant = false;
                break;
            }
        }
        // 如果是循环不变量，加入到待移动列表
        if (isInvariant) {
            toHoist.push_back(&I);
        }
    }
}
```

7、移动指令到preheader基本块

```
for (auto *I : toHoist) {
    I->moveBefore(preheader->getTerminator());
    ++moveCount;
}
```

函数内联

Inliner.hpp

Inliner.cpp

1、获取可内联的函数集合 (getFunctionsToInline 函数)

- 使用LLVM的 CallGraph 分析模块来构建调用图

```
CallGraph CG(M);
```

- 通过SCC迭代器遍历调用图，确定哪些函数可以被内联

```
for (scc_iterator<CallGraph *> I = scc_begin(&CG); !I.isAtEnd(); ++I) {
  const std::vector<CallGraphNode *> &SCCNodes = *I;
  bool SCCHasExternalCall = false;

  for (CallGraphNode *CGN : SCCNodes) {
    Function *F = CGN->getFunction();

    if (!F || F->isDeclaration())
      continue;

    if (CGN->getNumReferences() > 0) {
      SCCHasExternalCall = true;
      break;
    }

    Result.insert(F);
  }
```

- 判断一个函数是否可以内联的条件包括：不是外部声明的函数，并且没有被其他外部函数调用

```
  if (SCCHasExternalCall) {
    for (CallGraphNode *CGN : SCCNodes) {
      Function *F = CGN->getFunction();
      if (F && !F->isDeclaration())
        Result.erase(F);
    }
  }
```

2、执行内联操作 (run 函数)

- 首先调用 getFunctionsToInline 函数获取可以内联的函数集合

```
SmallPtrSet<Function *, 8> FunctionsToInline = getFunctionsToInline(M);
```

- 遍历这些函数，对每个函数内部的每个基本块和每条指令进行检查，如果发现可以内联的函数调用指令（即调用了一个本地定义的函数），则尝试使用 InlineFunction 函数进行内联操作，如果内联成功，则相应地更新代码；否则，记录内联失败的情况。

```
for (Function *F : FunctionsToInline) {
    if (!F->isDeclaration()) {
      InlineFunctionInfo IFI;
      for (auto &BB : *F) {
        for (auto &I : BB) {
          if (CallBase *CB = dyn_cast<CallBase>(&I)) {
            if (Function *Callee = CB->getCalledFunction()) {
```

```
            if (!Callee->isDeclaration()) {
              InlineResult IR = InlineFunction(*CB, IFI);
              if (IR.isSuccess()) {
                LLVM_DEBUG(dbgs() << "Function " << Callee->getName()
                                  << " was inlined into " << F->getName() << "\n");
              } else {
                LLVM_DEBUG(dbgs() << "Inlining failed for call to "
                                  << Callee->getName() << " in function "
                                  << F->getName() << ": " << IR.getFailureReason() <<
"\n");
              }
            }
          }
        }
      }
    }
  }
```

循环展开

LoopUnrolling.hpp

LoopUnrolling.cpp

1、遍历所有基本块中的指令，并检查是否为循环条件的分支指令，如果是，则进一步获取循环的结束条件。

```
if (auto *brInst = dyn_cast<BranchInst>(&inst)) {
  if (brInst->isConditional()) {
    if (auto *cmpInst = dyn_cast<ICmpInst>(brInst->getCondition())) {
      if (cmpInst->getPredicate() == ICmpInst::ICMP_SLT) {
        if (auto *loopVar = dyn_cast<PHINode>(cmpInst->getOperand(0))) {
          if (auto *loopBound = dyn_cast<LoadInst>(cmpInst->getOperand(1))) {
            if (auto *loopBoundValue = dyn_cast<ConstantInt>(loopBound->getPointerOperand()))
{
              int loopBoundInt = loopBoundValue->getSExtValue();
```

2、展开循环体：使用IRBuilder在原位置插入展开的循环体副本，builder.Insert(clonedInst)将复制的指令插入到当前位置。

```
IRBuilder<> builder(&inst);
for (int i = 0; i < loopBoundInt; ++i) {
  for (auto &bbInst : *brInst->getSuccessor(0)) {
    auto *clonedInst = bbInst.clone();
    builder.Insert(clonedInst);
    instToErase.push_back(clonedInst);
  }
}
```

3、移除原循环

```
  instToErase.push_back(&inst);
  for (auto& i : instToErase)
    i->eraseFromParent();
```