优化实现概览

Mem2Reg实现

该优化的思想顾名思义，就是将内存操作尽可能转化为高效的寄存器操作，主要操作就是在支配边界插入PHI结点并重写一个基本块内的使用了Load指令返回值的指令操作数（类似定值传播）。

笔者的理解为什么没有PHI结点的情况下Load、Store操作会有这么多，这是因为一个基本块的前驱基本块可能都有对某变量的定值，而如果基本块间没有“通信手段”，那么就需要前驱基本块中将定值写入内存，当前基本块再从内存中读取；

插入PHI指令之后就不需要内存作为值传递的中介了，因为PHI指令可以根据从**哪个基本块跳转过来的**确定某变量的定值。接下来讲解PHI指令的插入过程：

1. 首先根据函数模块的控制流图获取支配树，支配树中结点A的后继节点为B，表示**从入口节点到基本块B一定会经过结点A**，那么在结点A中的定值就一定能够传递到结点B，因此可以进行值传递

   ```
   DominatorTree domTree;
   for (auto& func : mod) {
       if (func.isDeclaration())
           continue;
       domTree.recalculate(func);
       ......
   }
   ```

2. 然后获取支配边界，顾名思义支配边界就是某节点恰好支配不到的哪些结点，基本块A的定值有可能传递到后继基本块B也有可能传递不到，因为从入口节点到基本块B可能不经过基本块A（支配边界）。因此我们就是要在支配边界插入PHI结点，如果从基本块A跳转到支配边界B，PHI结点的返回值就是在基本块A上的定值，否则是其他基本块的定值

   计算支配边界的算法在此不再赘述，主要是找汇点（存在多个前驱基本块），且汇点中有前驱基本块不支配汇点

3. 下一步找出可以被优化的Alloca指令，由于数组操作的Mem2Reg的优化十分困难，所以笔者忽略数组操作的优化，另外如果Alloca指令的返回值用作了其他用途（指针操作等），那么也不优化；

   对于这些可以优化的Alloca指令，需要保存对Alloca返回空间的定值、引用指令所在的基本块，方便后面的传递

4. 再向支配边界上插入PHI结点，并保存基本块以及对应的Alloca信息

5. 最后是支配边界上的PHI结点的赋值以及结点的重命名操作：因为LLVM是静态单赋值的形式，所以重命名操作可以不用管，主要是将值传递到PHI结点需要用到一个工作集算法

Mem2Reg作为最重要的一个优化难度也非常高，需要很牢固的理论支撑以及代码能力，笔者实现的版本感觉没能完全消除可以消除的内存操作，所以后面都是使用助教提供的实现，在`TransformPass.hpp`中保留它的实现但不使用

LICM实现

循环不变量外提主要思想就是外提循环中不会改变返回值的计算，避免不必要的计算，在本实验中主要针对`while`循环中的不变量。整个优化算法的思想比较简单，就是识别循环不变量然后把它外提到循环入口结点前，但是实现起来许多需要注意的细节，**比如必须确认循环会执行再外提store指令，否则会改变程序原有逻辑**，需要不断调整代码以保证所有案例通过。下面简单讲解实现过程：

1. 首先利用LLVM提供的LoopAnalysis获取函数模块中的所有循环信息，然后获取函数模块的结点支配树，这会用在判断是否能外提上

```
LoopInfo& loopInfos = fam.getResult<LoopAnalysis>(func);
domTree.recalculate(func);
std::unordered_set<BasicBlock*> visited;    // 避免重复处理内循环
for (Loop* loop : loopInfos) {
    ......
}
```

2. 然后按照从内循环到外循环的顺序进行不变量外提，因为不变量从内循环提出后可能能够进一步外提

```
void processLoop(Loop* loop, std::unordered_set<BasicBlock*>& visited)
{
    const std::vector<Loop*>& subLoops = loop->getSubLoops();
    for (auto subLoop : subLoops)
        processLoop(subLoop, visited);
    for (auto block : loop->getBlocksVector()) {
        // 不处理内循环的基本块
        if (visited.find(block) != visited.end())
            continue;
        ......
        visited.insert(block);
    }
    ......
}
```

> 上述两步都没什么细节需要注意，主要是保证外循环**不要重复处理内循环的基本块**即可，因为处理内循环的时候已经处理过它的基本块

3. 接下来就是遍历循环基本块中的指令，判断它是不是循环不变量了。笔者将其分为两部分：
    1. 初步判断：如果指令是PHI、Br、Ret、ICmp、SExt指令，则一定不能外提，因为这些指令一般用于循环块跳转或者基本块之间的值传递(PHI)，所以不外提；
    2. 中阶判断：如果是**Store指令**，则判断该指令是否被所有循环退出块支配(对于经典`while`循环一般是`while.cond`或者`break`语句翻译来的基本块)；**如果没有被所有循环退出块支配则该Store指令有可能不执行，所以一定不能外提**；如果是**Call指令**，则判断Call的函数是否**幂等**，即多次调用对于同样的参数是否返回相同的结果、是否有副作用(修改全局变量、传入数组参数被修改等)等
    3. 最终判断：经过上述两步判断后，我们初步判断该指令合规，接下来就是判断指令的所有操作数是否会在循环中改变，**如果循环操作数对应指令已经被标记为循环不变量，或者操作数的计算本身就在循环外**，则指令可以外提

    上述三步表现在代码上有不同的实现方式，笔者实现了最直观的方式，特殊处理所有可能出现的指令，且实现的循环不变量外提并不激进，比如对数组的Store操作笔者没有作进一步判断(因为无法判断Store操作是否会执行，所以保守起见不要外提)

算法伪代码如下：

```
function isLoopInvariant(inst, curLoop) -> bool
    // 根据指令类型判断是否可能被外提，如果为Br、PHI、ICmp这些指令则一律不外提
    if inst.opcode == PHI or inst.opcode == Br or inst.opcode == ICmp or inst.opcode == SExt:
        return false
    // 判断Store指令所在基本块是否被所有退出块支配，Call指令是否幂等等等
    if inst.opcode == Store:
        if some exitingBlocks of curLoop don't dominates inst.parent:
            return false
    if inst.opcode == Call:
        if the corresponding callee isn't idempotent:
            return false
    // 判断指令所有操作数是否在循环内被修改
    for all operands of inst:
        if operand are modified in loop:
            return false
    return true
```

4. 最后外提循环不变量即可，LLVM中只需要一句代码：`inst->moveBefore(loop->getLoopPreheader()->getTerminator())`，把不变量提到循环入口处即可(即`preheader basic block`)

本实验中循环不变量外提的优化巨大，而且该优化算法并不算难，只需要理清每一步写代码，然后调试保证所有案例正确性以及优化可行性即可

函数内联实现

函数内联就是将函数代码迁移到函数被调用指令处，同时进行形参到实参的替换和返回值的替换，当然还要判断函数调用是否在环中（直接或间接递归）、判断函数调用是否幂等（多次调用对同样的参数是否返回值一样，是否有副作用）等等。

该优化算法并不算难，因此思路非常直观：

1. 对于每条CallInst指令，首先判断Callee是否在一条函数调用环中，也就是是否存在**直接递归、间接递归**，如果是递归则不能内联；另外为了保证代码体积，笔者建议函数代码量太大的也不宜内联，因为这些函数可能被多次调用，内联后体积会非常大

   这里判断函数调用点是否在环内有很多方法，深搜、广搜、拓扑排序都可以。笔者这里使用广搜来完成，主要是从函数调用点开始，用队列保存Callee中的调用其他函数的情况，如果回到了Callee，则Callee在环内，不能内联，否则可以内联。具体代码实现参见源代码的`bool hasCallLoop(Function* callOne);`

2. 接下来就是一些自定义的内联设置了，笔者设置如果函数指令跳数大于40条、或者存在循环跳转则不进行内联
3. 再就是函数内联步骤了，因为函数有形参、有返回值，我们需要处理这两个情况；另外函数内也可能有局部变量的Alloca，我们外提的时候也要判断是否和Caller中的变量重名(不过LLVM中不需要注意该问题，IRBuilder在创建指令的时候会保证当前Module中不会有重名变量)：

   1. **建立形参到实参映射**

2. **根据映射复制语句**：主要是创建在当前的Module下的IRBuilder，将插入点设置在Call指令处，就可以肆意复制指令了；

   当Callee中的指令操作数是形参，那么换成Caller传入的实参；在复制过程中我们也需要**保存从Callee内指令到复制指令的映射**，原因很简单，后面需要复制的指令又不是以Callee中指令结果为操作数，而是以前面复制出来的指令的结果为操作数

3. **替换返回值为Callee中的ret语句值**：只需要在复制到`ret`语句的时候调用`replaceAllUsesWith`即可替换Caller中所有使用到函数返回值的地方

4. **删除Call指令**：`eraseFromParent`即可，注意如果是用迭代器遍历所有指令，则不应该删除因为会破坏迭代器，在完成所有判断后再删除

4. 在所有内联完成后，可以判断函数模块是否被调用，删除所有没有被调用的函数模块

5. 最后讲一个小细节，复制的Callee指令应该被插入到Caller的Call指令后面；**因为函数内联是可以递归进行的，Callee中的Call指令可能有可以内联**，将复制指令插入到后面迭代器就可以遍历到

函数内联虽然简单但是也给很多优化提供了前置条件，比如循环展开、循环不变量外提等等，因为内联后可能是循环不变量，或者循环展开后又能被公共子表达式消除优化所优化。因此函数内联优化十分有必要

循环展开实现

循环展开是本次实验所有优化中第二难实现的（最难是Mem2Reg），主要思想是提前获知循环执行的次数n，然后将循环体展开n次，随后删除循环相关基本块即可。**该优化为公共子表达式删除等提供了重大优化空间**

下面简单介绍实现细则，具体实现参见源代码：

1. 首先判断循环是否可以展开、是否适合展开：这一步可以自定义，笔者选择的是比较保守的展开方式，展开条件如下：

```
/**
* 判断循环是否可以展开，条件如下：
*     1.循环退出块只有一个，后继块也只有一个，latch块也只有一个
*     2.循环没有内循环，且循环迭代次数有限，这里设置不超过80次
*     3.循环内有其他跳转也不展开
*/
```

2. 获取循环迭代次数：如果不能获知循环执行次数，则不进行展开。获取方式也很简单，观察LLVM的IR代码可以发现规律，循环迭代变量常常在循环入口块中以PHI结点的形式出现：

   1. 如果是初次进入循环，循环迭代变量初始值常常是一个常数
   2. 如果是从latch block回到循环入口，则是自增后的值

   根据循环迭代变量初始值、每次自增的量、以及循环判断条件，三者结合判断即可得到循环迭代次数

3. 开始展开：主要步骤是在循环的`preheader`块，也就是循环入口块后面新建一个承接循环展开后指令的基本块，将循环体指令复制n次插入到这个新创建的基本块内，并建立这个新基本块和其他基本块的正确连接即可

   这一步代码实现也有很多方式，不过核心仍然是**映射图（类似函数内联中的`valueMap`）**、**复制指令插入点**、**新产生的基本块和已有基本块的连接**、**原本循环块的删除**等，这些细节笔者不赘述，参见源码`processLoop`和`loopBodyCopy`两个函数

4. 最后需要注意先展开内循环再展开外循环即可

下面介绍一些笔者在实现过程中遇到的LLVM相关的点：

1. `getExitingBlock`、`getExitBlock`和`getLoopLatch`**的区别**：latch block是指从该基本块有一条边回到循环的Header，在`while`循环中latch block常常是循环体中的最后一个基本块(`while.body`)；exiting block指的是从该基本块有一条边到循环外部基本块，exiting block常常是循环条件所在基本块(`while.cond`)；exit block是循环退出后到达的块，在`while`循环中常常是`while.end`等
2. **循环展开的一大细节就是判断操作数均为常数的情况，因为展开之后循环变量是已知的常数**，因为这种情况下不用创建指令，只需要保存映射。像`BinaryOperator`和`SExtInst`这两个操作都需要考虑操作数均为常数的情况
3. 还有一个注意的问题就是循环迭代变量更新的时机需要注意，并不是所有`i=i+1`都放在循环末尾，**因此在复制到迭代变量自增语句的时候再进行循环迭代变量的重新计算映射，而不是在循环体开始复制就更新映射。**笔者在实现的时候忽略了这点，所以占用了很多时间

虽然循环展开看似实现起来简单，但实际上要经历较长时间的debug过程，**对特殊情况的考虑或者LLVM指令的不熟悉都会导致难以调试的错误**，所以请一定谨慎考虑代码实现。

控制流简化

该优化主要针对在编译阶段就已知基本块跳转情况时，**将条件跳转改为直接跳转、合并可以合并的基本块，以减少代码的跳转次数。**该优化的实现比较简单(不过笔者还是和助教因为一个非常简单的错误调试了一个小时左右~~，该错误就是`insertBefore`这个坑爹API，下面会说明)，步骤如下：

1. **首先获取函数的基本块跳转情况**：在IR中的跳转常见无非是三种情况：
    1. 非条件跳转：一般在翻译`while`和`if`这些语句的时候产生，**因为需要生成有回边的基本块**；
    2. 条件跳转：在`while`和`if`等的`header`块中生成
    3. 返回语句：本优化不涉及
2. **判断优化条件**：非条件跳转自然不用说，如果跳转到的块没有回边（经过优化后回边可能会被优化掉，比如循环展开），那么就删除这个非条件跳转并合并当前块和跳转到的块，这样可以减少跳转次数；

    如果是条件跳转，我们要判断**是否在编译时期就已知跳转到哪个块(比如比较的两个数是常数)**，这样就可以优化为非条件跳转，然后非条件跳转就能判断是否可以合并基本块
3. **合并基本块的条件**：并不是所有直接跳转都能合并基本块，还需要**判断跳转到的基本块有没有回边，如果有则不能合并**
4. **修改跳转语句以及合并基本块**：修改跳转语句比较简单，就是创建直接跳转语句并插入到基本块末尾，删除原本的跳转语句即可；对于合并基本块，需要将跳转到的块中的所有指令复制到当前基本块末尾，同时删除原本的直接跳转语句，最后删除所有已经被合并的块（**如果是用迭代器遍历基本块的时候注意不要边遍历边删除**）

    在移动语句的时候一定要先调用`removeFromParent`再调用`insertBefore`，因为`insertBefore`并不会删除再原本块中的引用，笔者再移动之后发现被合并的基本块中仍然有这条指令，后续在删除基本块的时候容易造成错误导致死循环~~

```
for (auto succInst : succInsts) {
    succInst->removeFromParent();
    succInst->insertBefore(branchInst);
}
```

该优化对`if-combine*`有很大提升，当然合并基本块的行为对很多案例都有一定的提高，而且该优化实现并不复杂，推荐实现当作练手。

其他优化实现

1. **常量传播实现**：主要针对全局常量和局部常量：
    1. 如果是全局变量需要遍历完所有函数模块以确认该全局变量是否会被修改
    2. 如果是局部变量则用定值-引用链进行传播
2. **常量折叠实现**：该优化助教已经实现完毕，主要针对两个常数的二元操作进行传播
3. **强度削弱实现**：乘除法变移位、取余操作变为移位和加减法操作，这些都是很简单的操作不再赘述
4. **代数恒等式**：0作乘数、0作被除数、0作加法操作数、0作减法操作数、1作取余操作数，**注意任何数取余于1应该等于0（笔者脑一抽取余于1返回1了，debug了好久~~）**
5. **常量数组消除实现**：主要针对一个局部数组的操作在编译时期都是显式的，比如**往一个已知的位置存入一个常数**，**往一个已知位置取数**，**该数组没有作为函数参数被外部访问**等，这些都表示**对这个数组的store存值操作可以传递到load取值操作那里**

    事实上笔者实现的是比较保守的优化方式，具体实现参照源代码`TransformPass.hpp`末尾
6. **公共子表达式消除实现**：

    事实上这个优化提升也挺大的，不过该优化的实现比较简单而且不具有普适性(正确性是可以保证的，只是可能应对的情况比较少)，在源代码中笔者也写了详细的注释表明优化的目标和实现方式，这里简单说明一下：
    1. **消除`i1+i2+...+in`这种加法多次出现**：观察IR就能发现规律了，当前加法的第一个操作数是**上一个指令的结果**，第二个操作数**周期出现**，然后就用模拟的算法找出重复的操作以及重复次数，将加法化简为乘法或者其他传播操作即可。主要针对`hoist-*`案例
    2. **消除重复的load和store操作**：这里主要针对全局变量，因为保守起见如果全局变量有赋值操作，那么它不会被`Mem2Reg`和`ConstantPropagate`优化掉，所以笔者将其放在这里处理，，主要针对下面这种形式：

        ```
        store i32 %i.0, ptr @global, align 4
        %1 = load i32, ptr @global, align 4
        %2 = add i32 0, %1
        store i32 %i.0, ptr @global, align 4
        %3 = load i32, ptr @global, align 4
        %4 = add i32 %2, %3
        ......
        ```

        消除思路也很简单，只要保存基本块中最后一次对全局变量的store操作的`valueOperand`，在需要的时候传播它即可
    3. **消除i=i+1; i=i+1; .....这种形式**：也是模拟+自动机的算法，将连续的加法转化为乘法

4. **删除多余的GEP指令**：判断GEP指令是否等价主要判断索引是否都相等、操作地址是否一样，GEP指令能传播主要判断**出现了GEP指令的基本块是否支配需要传播到的基本块，如果支配则传播并删除那个多余的**
5. **两个操作数相等的情况的传播**：`a = x op y; b = x op y;`这种情况，主要还是用哈希表保存`x op y`然后判断能否传播

7. **死代码消除实现**：如果指令结果没有被使用，或者`store`操作的地址、`alloca`分配的内存没有被使用，那么这些指令都应该被**级联删除**，笔者保守起见没有对数组进行优化

实验感想

本次实验中笔者先了解了编译优化相关理论知识，然后着手开始从提升性能高的优化开始实现，一开始的`Mem2Reg`确实让笔者非常困惑，直到最后也是只实现了一个提升不高的版本，这个优化也是本次实验中最重要的优化，好在助教后来给出了实现；然后笔者开始着手循环不变量外提、函数内联、循环展开这些重要的优化，虽然说思想并不难但是要想考虑到全部情况，真的需要非常长时间的Debug和代码调整，助教给出的优化示例实现给了笔者很大的启发；最后希望在学习到理论的时候能对自己的代码做一些调整，优化代码的鲁棒性和高效性吧。