编译原理研讨课实验PR003实验报告

任务说明

本次实验在前面两次实验的基础上完善从AST到LLVM IR的代码生成通路，将支持了elementWise操作构建的新的AST上的语句转化为LLVM IR。

成员组成

- 燕澄皓 2017K8009915041
- 蔡润泽 2017K8009908018

实验设计

设计思路

本次实验中我们将要实现的elementWise操作嵌套进入for循环结构中，通过理论课程上学习到的对于for循环结构的翻译与处理流程来完成数组的向量化运算。

在`PR003实验说明`中老师已经在LLVM-IR代码生成入口函数`EmitAnyExpr`中添加了支持整形静态数组加法运算的基本框架。但是示例中只演示了数组的单一位相加，我们要做的就是基于已有的框架，梳理清楚细节，支持for循环的嵌套结构以及赋值`=`、乘法`*`另外两种操作。

实验实现

1. **找到一般表达式的代码生成入口：`RValue CodeGenFunction::EmitAnyExpr`**

   该函数负责根据表达式的类型生成具体的代码计算并返回一个右值，我们要在函数的开头处添加对于整形静态数组这一非常规表达式类型的支持。
   修改文件位于`llvm/tools/clang/lib/CodeGen/CGExpr.cpp`，添加的代码如下：

```
RValue CodeGenFunction::EmitAnyExpr(const Expr *E,
                                    AggValueSlot aggSlot,
                                    bool ignoreResult) {
  if (E->getType()->getTypeClass() == Type::ConstantArray) {
    return RValue::get(EmitElementWiseExpr(E));
  }
  ...
}
```

添加的if分支就是用于检测elementWise类型的表达式，并交由`EmitElementWiseExpr`函数进行具体处理。

2. **在`EmitElementWiseExpr`中负责为类型为整形静态数组的表达式生成代码**

   修改文件位于`llvm/tools/clang/lib/CodeGen/CGExpr.cpp`，函数`EmitElementWiseExpr`的部分实现如下：

   1. **得到表达式左右两侧的基地址**

      如果运算符是赋值操作，则直接去符号表中查询LHSBase（左侧表达式的基地址）；如果运算符是`+`或者`*`操作，我们则要递归地先去为该运算生成代码并返回表达式的基地址。这里注意：RHSBase始终是递归生成的，而LHSBase取决于二元运算符是否为赋值操作而有所不同。在实现这部分代码时，用到了理论课上学习到的自顶向下分析的思想。

      ```
      if (BinaryOperator::classof(E)) {
        const BinaryOperator* bo = dyn_cast<BinaryOperator>(E);
        Expr* LHS = bo->getLHS();
        Expr* RHS = bo->getRHS();
        llvm::Value *LHSBase, *RHSBase;
        if (bo->getOpcode() == BO_Assign) {
          const DeclRefExpr *declRef = dyn_cast<DeclRefExpr>(LHS);
          const ValueDecl* decl = declRef->getDecl();
          LHSBase = LocalDeclMap.lookup((Decl*)decl);
        } else {
          LHSBase = EmitElementWiseExpr(LHS);
        }
        RHSBase = EmitElementWiseExpr(RHS);
      ```

   2. **分配用于循环索引的临时变量i并完成初始化**

      ```
      // Alloc temp val: i
      QualType Ty = getContext().UnsignedIntTy;
      llvm::Type *LTy = ConvertTypeForMem(Ty);
      llvm::AllocaInst *Alloc = CreateTempAlloca(LTy);
      Alloc->setName("compiler.for.i");
      Alloc->setAlignment(4);

      // init i, use Store_i
      // 0: use LLVM's API -> LLVM expr
      llvm::StoreInst *Store_i = Builder.CreateStore(llvm::ConstantInt::get(LTy, llvm
      Store_i->setAlignment(4);
      ```

   3. **生成四个基本块**

```
llvm::BasicBlock *ConditionBlock = createBasicBlock("compiler.for.condition");
llvm::BasicBlock *BodyBlock = createBasicBlock("compiler.for.body");
llvm::BasicBlock *IncBlock = createBasicBlock("compiler.for.inc");
llvm::BasicBlock *EndBlock = createBasicBlock("compiler.for.end");
```

COND块进行for循环的条件检查，根据循环索引变量i与数组长度的大小关系来控制跳转

```
// load compiler.i as offset
// LLVM returns Value, StoreInst, AllocInst
// LoadInst is Value
llvm::LoadInst *idx = Builder.CreateLoad((llvm::Value*)Alloc, "");
idx->setAlignment(4);

// type Promote to Int
llvm::Value *idxPromoted = Builder.CreateIntCast(idx, IntPtrTy, false, "idxprom

const ConstantArrayType *CATy = dyn_cast<ConstantArrayType>(TyR);
llvm::Value *ConditionCmpRHS = llvm::ConstantInt::get(IntPtrTy, CATy->getSize()

// create condition branch
llvm::Value *ConditionCmp = Builder.CreateICmpULT(idxPromoted, ConditionCmpRHS,
Builder.CreateCondBr(ConditionCmp, BodyBlock, EndBlock);
```

BODY块为真正的循环体生成代码，核心代码实现如下所示：

- 加法乘法需要先维护好结果数组C的偏移，通过`CreateAdd`或者`CreateMul`计算加法的值，并将运算的结果通过`CreateStore`存入数组C之中
- 赋值操作则直接通过`CreateLoad`与`CreateStore`完成赋值操作

```
if(bo->getOpcode() == BO_Add || bo->getOpcode() == BO_Mul){
  llvm::LoadInst *valueA = Builder.CreateLoad(addrL, "");
  llvm::LoadInst *valueB = Builder.CreateLoad(addrR, "");
  valueA->setAlignment(4);
  valueB->setAlignment(4);
  tempC = CreateTempAlloca(LTyR);
  tempC->setAlignment(4);

  llvm::Value *baseC = (llvm::Value *)tempC;
  llvm::Value *arrayPtrC = MakeAddrLValue(baseC, TyR).getAddress();
  llvm::Value *addrC = Builder.CreateInBoundsGEP(arrayPtrC, Args, "arrayidx");
  llvm::Value *op_result;

  if (bo->getOpcode() == BO_Add) {
    op_result = Builder.CreateAdd((llvm::Value *)valueA, (llvm::Value *)valueB,
  } else if (bo->getOpcode() == BO_Mul) {
    op_result = Builder.CreateMul((llvm::Value *)valueA, (llvm::Value *)valueB,
  }
  llvm::StoreInst *valueC = Builder.CreateStore(op_result, (llvm::Value *)addrC
  valueC->setAlignment(4);
} else if (bo->getOpcode() == BO_Assign) {
  llvm::LoadInst *load = Builder.CreateLoad(addrR, "");
  load->setAlignment(4);
  llvm::StoreInst *valueC = Builder.CreateStore((llvm::Value*)load, addrL, fals
  valueC->setAlignment(4);
}
```

INC块负责循环索引变量i的自增

load->add->store的操作方式完成i++的操作，随后无条件跳转到COND块的起始处进行循环条件检查

```
llvm::Value *One = llvm::ConstantInt::get(Int32Ty, 1);
llvm::LoadInst *IncI = Builder.CreateLoad((llvm::Value*)Alloc, "");
IncI->setAlignment(4);
llvm::Value* IncAdd = Builder.CreateAdd((llvm::Value*)IncI, One, "");
llvm::StoreInst *StoreI = Builder.CreateStore(IncAdd, (llvm::Value*)Alloc, fals
StoreI->setAlignment(4);
Builder.CreateBr(ConditionBlock);
```

END块负责循环结束后的返回处理，完成子表达式的处理后返回到上层调用（可能是一个递归过程的返回）

如果操作符是赋值操作，则返回左侧表达式基地址

如果是加或乘操作，则返回保存了运算结果的临时结果数组C

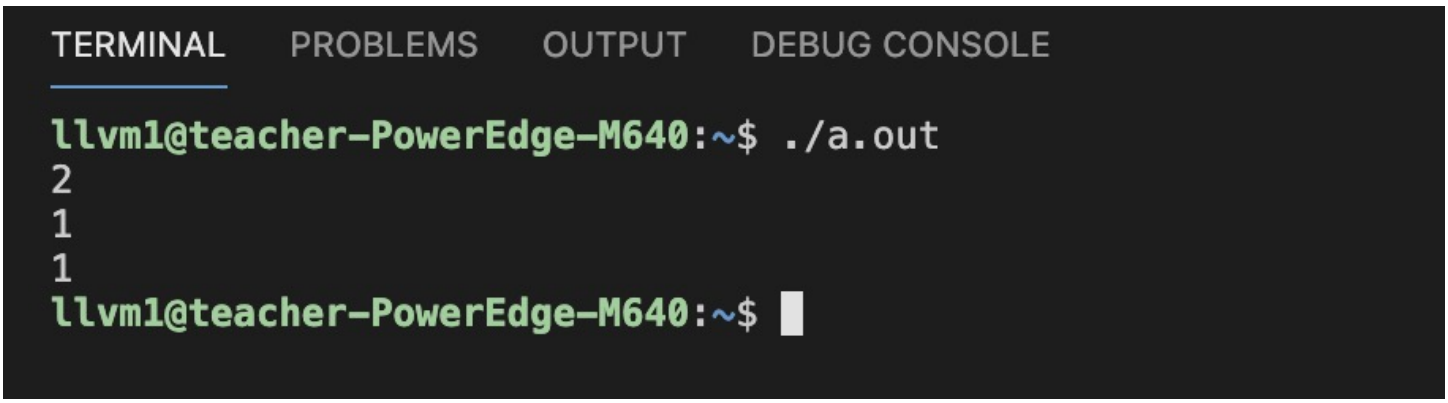
```
if (bo->getOpcode() == BO_Assign) {
  return LHSBase;
} else {
  return (llvm::Value*)tempC;
}
```

总结

实验结果总结

Sample code如下：

```
#pragma elementWise
#include <stdio.h>
void foo1(){
  int A[1000];
        int B[1000];
        int C[1000];
        for(int i = 0; i < 1000; i++){
                A[i] = i;
                B[i] = i;
        }
        C = A + B;
        printf("%d\n", C[1]);
        C = A * B;
        printf("%d\n", C[1]);
  C = B;
        printf("%d\n", C[1]);
}
int main(){
        foo1();
  return 0;
}
```



三次打印输出的结果分别为`2,1,1`，如上图，符合预期，生成的LLVM IR如下：

```
; ModuleID = 'test.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @foo1() #0 {
entry:
  %A = alloca [1000 x i32], align 16
  %B = alloca [1000 x i32], align 16
  %C = alloca [1000 x i32], align 16
  %i = alloca i32, align 4
  %compiler.for.i = alloca i32, align 4
  %tmp = alloca [1000 x i32], align 4
  %compiler.for.i8 = alloca i32, align 4
  %compiler.for.i19 = alloca i32, align 4
  %tmp26 = alloca [1000 x i32], align 4
  %compiler.for.i31 = alloca i32, align 4
  %compiler.for.i43 = alloca i32, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32* %i, align 4
  %cmp = icmp slt i32 %0, 1000
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i32* %i, align 4
  %2 = load i32* %i, align 4
  %idxprom = sext i32 %2 to i64
  %arrayidx = getelementptr inbounds [1000 x i32]* %A, i32 0, i64 %idxprom
  store i32 %1, i32* %arrayidx, align 4
  %3 = load i32* %i, align 4
  %4 = load i32* %i, align 4
  %idxprom1 = sext i32 %4 to i64
  %arrayidx2 = getelementptr inbounds [1000 x i32]* %B, i32 0, i64 %idxprom1
  store i32 %3, i32* %arrayidx2, align 4
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %5 = load i32* %i, align 4
  %inc = add nsw i32 %5, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  store i32 0, i32* %compiler.for.i, align 4
  br label %compiler.for.condition

compiler.for.condition:                           ; preds = %compiler.for.inc, %for.enc
  %6 = load i32* %compiler.for.i, align 4
  %idxprom3 = zext i32 %6 to i64
```

```
  %compiler.for.condition.cmp = icmp ult i64 %idxprom3, 1000
  br i1 %compiler.for.condition.cmp, label %compiler.for.body, label %compiler.for.end

compiler.for.body:                                ; preds = %compiler.for.condition
  %arrayidx4 = getelementptr inbounds [1000 x i32]* %A, i32 0, i64 %idxprom3
  %arrayidx5 = getelementptr inbounds [1000 x i32]* %B, i32 0, i64 %idxprom3
  %7 = load i32* %arrayidx4, align 4
  %8 = load i32* %arrayidx5, align 4
  %arrayidx6 = getelementptr inbounds [1000 x i32]* %tmp, i32 0, i64 %idxprom3
  %add = add i32 %7, %8
  store i32 %add, i32* %arrayidx6, align 4
  br label %compiler.for.inc

compiler.for.inc:                                 ; preds = %compiler.for.body
  %9 = load i32* %compiler.for.i, align 4
  %10 = add i32 %9, 1
  store i32 %10, i32* %compiler.for.i, align 4
  br label %compiler.for.condition

compiler.for.end:                                 ; preds = %compiler.for.condition
  store i32 0, i32* %compiler.for.i8, align 4
  br label %compiler.for.condition9

compiler.for.condition9:                          ; preds = %compiler.for.inc15, %compi
  %11 = load i32* %compiler.for.i8, align 4
  %idxprom10 = zext i32 %11 to i64
  %compiler.for.condition.cmp11 = icmp ult i64 %idxprom10, 1000
  br i1 %compiler.for.condition.cmp11, label %compiler.for.body12, label %compiler.for.

compiler.for.body12:                              ; preds = %compiler.for.condition9
  %arrayidx13 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 %idxprom10
  %arrayidx14 = getelementptr inbounds [1000 x i32]* %tmp, i32 0, i64 %idxprom10
  %12 = load i32* %arrayidx14, align 4
  store i32 %12, i32* %arrayidx13, align 4
  br label %compiler.for.inc15

compiler.for.inc15:                               ; preds = %compiler.for.body12
  %13 = load i32* %compiler.for.i8, align 4
  %14 = add i32 %13, 1
  store i32 %14, i32* %compiler.for.i8, align 4
  br label %compiler.for.condition9

compiler.for.end16:                               ; preds = %compiler.for.condition9
  %arrayidx17 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 1
  %15 = load i32* %arrayidx17, align 4
  %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32
  store i32 0, i32* %compiler.for.i19, align 4
  br label %compiler.for.condition20

compiler.for.condition20:                         ; preds = %compiler.for.inc28, %compi
  %16 = load i32* %compiler.for.i19, align 4
  %idxprom21 = zext i32 %16 to i64
  %compiler.for.condition.cmp22 = icmp ult i64 %idxprom21, 1000
  br i1 %compiler.for.condition.cmp22, label %compiler.for.body23, label %compiler.for.
```

```
compiler.for.body23:                              ; preds = %compiler.for.condition20
  %arrayidx24 = getelementptr inbounds [1000 x i32]* %A, i32 0, i64 %idxprom21
  %arrayidx25 = getelementptr inbounds [1000 x i32]* %B, i32 0, i64 %idxprom21
  %17 = load i32* %arrayidx24, align 4
  %18 = load i32* %arrayidx25, align 4
  %arrayidx27 = getelementptr inbounds [1000 x i32]* %tmp26, i32 0, i64 %idxprom21
  %mul = mul i32 %17, %18
  store i32 %mul, i32* %arrayidx27, align 4
  br label %compiler.for.inc28

compiler.for.inc28:                               ; preds = %compiler.for.body23
  %19 = load i32* %compiler.for.i19, align 4
  %20 = add i32 %19, 1
  store i32 %20, i32* %compiler.for.i19, align 4
  br label %compiler.for.condition20

compiler.for.end29:                               ; preds = %compiler.for.condition20
  store i32 0, i32* %compiler.for.i31, align 4
  br label %compiler.for.condition32

compiler.for.condition32:                         ; preds = %compiler.for.inc38, %compi
  %21 = load i32* %compiler.for.i31, align 4
  %idxprom33 = zext i32 %21 to i64
  %compiler.for.condition.cmp34 = icmp ult i64 %idxprom33, 1000
  br i1 %compiler.for.condition.cmp34, label %compiler.for.body35, label %compiler.for.

compiler.for.body35:                              ; preds = %compiler.for.condition32
  %arrayidx36 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 %idxprom33
  %arrayidx37 = getelementptr inbounds [1000 x i32]* %tmp26, i32 0, i64 %idxprom33
  %22 = load i32* %arrayidx37, align 4
  store i32 %22, i32* %arrayidx36, align 4
  br label %compiler.for.inc38

compiler.for.inc38:                               ; preds = %compiler.for.body35
  %23 = load i32* %compiler.for.i31, align 4
  %24 = add i32 %23, 1
  store i32 %24, i32* %compiler.for.i31, align 4
  br label %compiler.for.condition32

compiler.for.end39:                               ; preds = %compiler.for.condition32
  %arrayidx40 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 1
  %25 = load i32* %arrayidx40, align 4
  %call41 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i
  store i32 0, i32* %compiler.for.i43, align 4
  br label %compiler.for.condition44

compiler.for.condition44:                         ; preds = %compiler.for.inc50, %compi
  %26 = load i32* %compiler.for.i43, align 4
  %idxprom45 = zext i32 %26 to i64
  %compiler.for.condition.cmp46 = icmp ult i64 %idxprom45, 1000
  br i1 %compiler.for.condition.cmp46, label %compiler.for.body47, label %compiler.for.

compiler.for.body47:                              ; preds = %compiler.for.condition44
  %arrayidx48 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 %idxprom45
  %arrayidx49 = getelementptr inbounds [1000 x i32]* %B, i32 0, i64 %idxprom45
```

```
  %27 = load i32* %arrayidx49, align 4
  store i32 %27, i32* %arrayidx48, align 4
  br label %compiler.for.inc50

compiler.for.inc50:                               ; preds = %compiler.for.body47
  %28 = load i32* %compiler.for.i43, align 4
  %29 = add i32 %28, 1
  store i32 %29, i32* %compiler.for.i43, align 4
  br label %compiler.for.condition44

compiler.for.end51:                               ; preds = %compiler.for.condition44
  %arrayidx52 = getelementptr inbounds [1000 x i32]* %C, i32 0, i64 1
  %30 = load i32* %arrayidx52, align 4
  %call53 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i
  ret void
}

declare i32 @printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  call void @foo1()
  ret i32 0
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"
attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame
```

分成员总结

蔡润泽：
本次实验主要负责写相关的代码。个人觉得有几个比较关键的要点：

- 需要将`elementWise`的计算操作转换成for循环的中间代码。而for循环的生成代码在理论课上有所学习，可以通过设置label（此实验中具体为生成block）、判断跳转和索引加一这几部分构成。
- 可以先利用llvm和clang生成标准for循环的中间代码(.s文件)，进行参考。根据标准for循环的代码可知，llvm for循环的中间代码包含四个block，分别是`condition`、`body`、`inc`以及`end`。四个部分分别对应了条件跳转、循环体代码、加一代码以及结束标签这几个部分。
- 参考任务书可以学习到如何进行相关的变量类型转换、左值和右值的处理。

综上所属，本次实验难度适中，在承接前两次实验的基础上最终实现了中间代码的生成，让我们对于`llvm`和`clang`的编译流程有了更进一步的理解。

燕澄皓：

本次实验中主要负责实验报告的撰写。本次实验难度总体适中，理解难度以及实现难度都不是很大。在有了老师提供的示例代码以及理论课的基础知识铺垫之后，我们在实验中要做的主体内容就是通过基本块的控制来实现for循环结构的翻译。在编译理论知识的高级层面需要我们认识到各个基本块的职责以

及跳转的控制；根据运算符是赋值操作还是算数运算，还要分析当前处理的是否是一个子表达式从而引入递归的函数实现方式。

个人认为实验中比较让人困惑的点还是在于需要对于`llvm`中各个类型的存储、调用有一个较为全面的认识。在每一个层面上我们都要注意许多细节，譬如当前应该用什么类型存储变量，是否需要进行类型转换，是否正确处理了偏移量以及地址，是否进行了对齐操作等等。

总体而言整个实验项目对于我们巩固理论知识、提高源码阅读以及实践能力有很大的帮助。