hw1_109062320_report

How to run your program

Since I used `getBounds` in my implementation, I modified the default rule to the custom rule that TA announced in eeclass.

```
[testcase].ll: [testcase].c
    $(CXX) $(IRFLAGS) -o $@ $<
    $(OPT) -S -passes=mem2reg,loop-rotate,loop-simplify $@ -o $@
```

As for the remaining parts in my makefile, they are identical to the makefile sample in the Spec, with the exception of the path to llvm_build.

cases handled by my implementation

In my implementation, I handle cases involving 1-D arrays with array indices in the form of `ai + b,` where both `a` and `b` are integers, and `i` is the loop induction variable. Furthermore, the loop induction variable of the `1-layer for loop` is updated with a fixed step of `+1` in each iteration.

Experiment report

ResultData

I use a custom structure to store information about data dependencies, which includes the following components:

1. Array Name: The name of the array associated with the data dependency.
2. Statement Number: The specific statement within the code that has the data dependency (e.g., 'S1' represents the first statement).
3. Results: The results calculated using the `diophatine_solver` function, which describe the nature and properties of the data dependencies for this statement.

This structure enables me to systematically organize and display information related to data dependencies within the code. This information includes the array name, statement number, calculated results, coefficients and constants of the index. For instance, in the case of an expression like A[2 * i + 3], the coefficient is 2, and the constant is 3.

```
struct ResultData {
    std::string name;
    SolEquStru solEquStru;
    int state0;
    int state1;
    int XCoeff;
    int YCoeff;
    int X0;
    int Y0;
};
```

processInst

In this function, I iterate through all the instructions in `BB` and send the load/store instructions to `processLoadInst` and `processStoreInst` for data handling. After processing, I store the load/store instructions in the `MemInstr` vector. Additionally, I maintain a map called `instructionToLineNumberMap` to keep track of the statement number associated with each instruction in this context.

processLoadInst, processStoreInst

Since the implementations of these two functions are similar, I will only explain `processLoadInst` here. Additionally, to facilitate data dependency checking, I have two additional maps: one to store the array index (for example, for `A[2i+3]`, I map the load/store instruction to a pair `(2, 3)`), and the other to store the array name corresponding to the load/store instruction.
Since When loading data from an array, we may have the instructions like the following instructions appear in the .ll file:

```
%1 = load i32, ptr %i, align 4
%idxprom = sext i32 %1 to i64
%arrayidx = getelementptr inbounds [20 x i32], ptr %C, i64 0, i64 %idxprom
%2 = load i32, ptr %arrayidx, align 4
```

In my implementation, taking the above instructions as an example, I create an instruction called `idx` associated with `%idxprom` and check its operands:

1. If the name of the first operand is identical to the induction variable, I set the pair to `(1, 0)` and map it to the load instruction, which is then stored in the 'arrayIdx' map.
2. If the first operand is a BinaryOperator, I examine its type (Sub, Add, Mul). If the index has the form `ai+b` the first operand will be an Add instruction. In this case,

I inspect the first operand of the BinaryOperator. If it's a Mul instruction, I extract the exact value and set it as the first element of the pair. The value of the second operand of the BinaryOperator becomes the second element of the pair. This pair, representing `(a, b)` is then mapped to the load instruction.

Besides, I will also associates the name of the array with its corresponding load instruction, storing this information in the `Arr_name` map.

`PreservedAnalyses HW1Pass::run(Function &F, FunctionAnalysisManager &FAM)`

In the `run` function inside the `HW1Pass` structure, I iterate through the BasicBlocks in function `F`. When I encounter a BasicBlock named `for.body` , I use `getBounds` to extract loop-related information, including `InitVal` (the initial induction variable value), `IndVal` (the induction variable), and `FinalVal` (the final induction variable value). I then pass references to the current BasicBlock, `MemInstr` (a `ValueVector` used to store load/store instructions), and `IndVal` to another function called processInst for further processing.
After processing all the instructions, a nested two-layer for loop is utilized to iterate through all the elements in the `MemInstr` vector. The outer loop's iterator is named `I`, and the inner loop's iterator is denoted as `J`. Within each iteration, the instruction pointed to by `I` is referenced as `Src`, while the instruction pointed to by `J` is referred to as `Dst`. This structure enables the comparison of different pairs of instructions in the `MemInstr` vector, facilitating further analysis and processing.
When the following conditions are met, I employ the `diophantine_solver` to determine the presence of data dependencies:

1. `Arr_name[Src] == Arr_name[Dst]`
2. `J != I`
3. Dst is not a `LoadInst`, or Src is not a `LoadInst`.

To examine data dependencies, I identify the corresponding pairs of Dst and Src instructions in the `arrayIdx` map. I then use the first element in these pairs as the first and second arguments for the `diophantine_solver` function. As for the third argument, it is calculated as the difference between the second elements in the two pairs.
If result calculated shows there exist solution, then there will be two conditions, and I have another three vectors to store the ResultData.

1. `isa<StoreInst>(Src) && isa<StoreInst>(Dst)`: exist output dependency
2. the others: exist flow dependency or anti dependency

```
ResultData* new_dep = new ResultData;
new_dep->name = Arr_name[Src];
new_dep->solEquStru = f;
new_dep->state0 = instructionToLineNumberMap[Src];
new_dep->state1 = instructionToLineNumberMap[Dst];
new_dep->XCoeff = src_first;
new_dep->YCoeff = dst_first;
new_dep->X0 = src_second;
new_dep->Y0 = dst_second;
// output dep
if(isa<StoreInst>(Src) && isa<StoreInst>(Dst)){
  output_dep.push_back(*new_dep);
}
else{// flow dep && anti dep
  flow_dep.push_back(*new_dep);
  anti_dep.push_back(*new_dep);

}
```

In the end, I iterate through the elements in `flow_dep`, `anti_dep`, and `output_dep` to print the results. Within each loop, I have an inner loop that iterates through 'i' from `InitVal` to `FinalVal`, which are obtained using `getBounds`. I determine the values of 'XCoeff' and 'X0' based on the statement with the smaller number. The printing conditions for each of these dependencies are as follows:

- When iterating through `flow_dep`, I print if `i * it->YCoeff + it->Y0 <= i * it->XCoeff + it->X0`.
- When iterating through `anti_dep`, I print if `i * it->YCoeff + it->Y0 > i * it->XCoeff + it->X0`.
- When iterating through `output_dep`, I print the results directly.