Advanced Compiler CS540400

name:莊岱倫 studentID:114062616

Implementation by solving diophantine equation

1. **Extracting Linear Array Accesses from LLVM IR**

   Loop 內，我搜尋所有load / store並對每個GEP 的最後一個 index，我嘗試將其解析為線性形式

   ```
   index= a x i + b
   ```

   code

   ```
   for (auto &I : instructions(F)) {
           auto *BB = I.getParent();
           if (!LI.getLoopFor(BB)) continue;

           if (auto *GEP = dyn_cast<GetElementPtrInst>(&I)) {
               Instruction *nextInst = GEP->getNextNode();
               if (!nextInst) continue;
               bool isStore = isa<StoreInst>(nextInst);
               bool isLoad  = isa<LoadInst>(nextInst);
               if (!isStore && !isLoad) continue;

               AccessInfo info;
               info.array = getArrayNameFromGEP(GEP);
               info.isStore = isStore;
               info.inst = nextInst;

               Value *idx = GEP->getOperand(GEP->getNumOperands() - 1);
               std::string vname; long long a=0,b=0;
               if (!parseLinear(idx,vname,a,b)) continue;
               info.varName=vname; info.a=a; info.b=b;
               if (!vname.empty()) indVarNames.insert(vname);

               accesses.push_back(info);
           }
    }
   ```

2. **Finding Loop Bounds**

   從 IR 中解析 induction variable 的初始值與比較條件

   ```
   auto bounds = detectLoopBounds(F, indVar);
   loopStart = bounds->first;
   ```

```
loopEnd   = bounds->second;
```

若無法解析，我 fallback 為 [0, 64)。

這樣可以讓後續的 Diophantine 解只在合法的 iteration 範圍內搜尋

3. **Constructing Dependence Equation**

   建立起兩次訪問同一記憶體位置的必要條件

```
long long A = L1.a;
long long B = -L2.a;
long long C = L2.b - L1.b;
```

   之後的GCD function會用這個這個線性模型去解

4. **Solving the Diophantine Equation**

   我寫了一個ext_gcd function找出特殊解

```
static long long ext_gcd(long long a, long long b, long long &x, long
long &y) {
    if (b == 0) { x = (a >= 0 ? 1 : -1); y = 0; return std::llabs(a);
}
    long long x1=0, y1=0; long long g = ext_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

   將求出的特殊解轉成通解

```
long long ip = x0 * (C/g);
long long jp = y0 * (C/g);
i = ip + (B/g) * t;
j = jp - (A/g) * t;
```

   利用迴圈界限推導 t 的合法範圍

```
i ∈ [loopStart, loopEnd)
j ∈ [loopStart, loopEnd)
```

5. **Program Order Constraints**

   就算找出index相同也要判斷statement再迴圈中的先後順序

```
if (iIter > jIter) continue;
if (iIter == jIter) {
          if (L1.stmtNo > L2.stmtNo) continue;
          if (L1.stmtNo == L2.stmtNo && L1.inStmtOrder >=
L2.inStmtOrder){
               continue;
          }
}
```

Bouns

Why Mixin Pattern

- 編譯期多型（Compile-time Polymorphism）相較於傳統的虛擬函式多型，CRTP 能在編譯期決定要呼叫哪個函式，不需要透過虛擬表（vtable）在執行期動態查找，效能更高。
- 消除執行期開銷由於 LLVM pass 經常被重複呼叫在大型 IR 上進行分析與優化，任何動態分派開銷都會被放大。CRTP 讓函式可以被 inline，提升執行效能並減少函式呼叫開銷。
- 統一 Pass 註冊與介面PassInfoMixin 提供統一的 run() 與註冊介面，使不同種類的 Pass都能以相同方式整合進 PassBuilder pipeline。
- 降低樣板程式碼（Boilerplate）新增一個 Pass 只需繼承自 PassInfoMixin，就能自動獲得 LLVM 的 PassMetadata 與註冊邏輯，不需重複撰寫登錄程式

LLVM ADT

SmallVector<T, N>	**小型固定容量的向量，對小量資料不會配置堆積記憶體**
DenseMap<Key, Value>	高效能的雜湊表，比 std::unordered_map 更節省記憶體
StringSet	一種針對字串鍵值的集合型別。你用來儲存迴圈變數名稱。
SmallPtrSet	優化型的集合，針對 pointer 資料或少量元素情境
ArrayRef	提供read only陣列參照，避免不必要的複製

作業中有用到的

1. SmallVector<AccessInfo, 16>
    - **大量小陣列操作時效能明顯較佳**

      IR 分析中訪問通常不多（16 或 32 個以內），使用 SmallVector 避免頻繁 heap allocation。
    - **能避免碎片化與 allocation overhead**

      尤其在 pass 被多次 run 時，差異更明顯。

      ```
      SmallVector<AccessInfo, 16> accesses;
      SmallVector<LinExpr, 16> lin;
      SmallVector<DependenceRecord, 32> flow, anti, output;
      ```

2. StringSet<> indVarNames; + StringRef

    - **提供 O(1) 查找** → 內部使用 StringMap
    - **避免重複字串儲存** → autodedupe
    - 只需要key就能查詢，不用完整的sed::string

3. ArrayRef

    用來建立read only的物件，任何型別都接受非常方便