编译技术 Project1 Report

石弼钊 1700012955
滕明桂 1700012974
周劭博 1700013008

Group 27

摘要 本次 project 的任务将一个 json 文件中所描述的 kernel 语句编译为一个 C++ 语言的源代码，我们的思路就是通过解析 kernel 为其建立一个 IR tree，然后通过遍历树上的节点，打印出最后的 C++ 的源代码，我们的代码主要可以分成三块，第一个部分 (Sec 1) 是对于 json 文件进行预处理，建立相应的符号表和节点，第二部分 (Sec 2) 就是将这些节点连接为一颗 IR tree，第三部分 (Sec 3) 就是遍历树节点输出。所有成员的分工见 Sec 4。

1 数据读取和预处理

数据预处理的流程是使用 jsoncpp 读取 kernel 文件，对于 kernel 从头开始扫描第一个 pass，这个 pass 主要构造符号表和将 kernel 的表达式从中缀转为后缀表达式，同时需要具体的解析出解析 tensor 变量并粗略计算出他 index 的范围，具体的步骤如下：

1. 首先创建符号表和一个单独的索引表，用于存储和记录已经出现的 index 和其范围，并将 kernels 的 ins 和 out 中 tensor 加入到符号表中
2. 解析 kernel，通过正则表达式每提取一个 tensor 和常量，如果提取到 tensor 都要创建一个 myVaribles 变量，记录这个 tensor 的变量名称，数据类型，维度，是否为左值，其 index 信息，并有相应的成员函数，如图1.1所示，其中记录 tensor index 的 vector 中存放是我们自己定义的 myIndex 变量，symbols 中存放的是每一个维度的变量的表达式。记录完整之后将其标记 var_x，放入输出的 vector 当中，用于生成后续建立 IR tree 使用的后缀表达式。如果遇到常量，单独对于其 myImmediate 变量，这个数据结构会记录其名字和值，加入符号表，并将其标记为 imm_x 放入输出的 vector 当中。对于每一个 tensor 而言，如果他不是左值，我们

需要判定其是不是需要进行规约，如果他的 index 中出现了不同于左值 tensor 的 index，则该变量需要进行规约，例如左值出现的 index 为 i, j，而右值中的变量出现了 index k，则这个 tensor 是需要进行规约，要对其建立一个单独的循环的累加，然后再将其值赋值给左值。

```
class myVariables{
public:
    std::string name;
    int type;
    int dim;
    int is_left;
    int rw;
    Expr expr;
    std::vector<int> shape;
    std::vector<std::string> indexes;
    std::set<std::string> symbols;
    std::set<std::string> delta_symbols;

    myVariables() {}
    myVariables(std::string s, int data_type, int flag);
    void Print();
    void update_symbols(std::map<std::string, myIndex> &id);
};
```

图 1.1. myVaribles 变量的数据结构

```
class myIndex{
public:
    std::string name;
    int lb;
    int ub;
    int for_loop;
    Expr dom;
    Expr slf;
    std::set<std::string> symbols;
    myIndex();
    int CheckIndex(std::string s);
    void Print();
    void IfCheck();
    bool operator<(const myIndex x) const{
        return (bool)name.compare(x.name);
    }
};
```

图 1.2. myIndex 变量的数据结构

3. 对于 index 的解析，会将 index 分为三种，一种是单独的索引，如 $i, j, k, \cdots$，这种 index 都需要在输出为 C++ 代码的时候，添加单独的 for 循环; 另一种是复合的索引，如 $i + k, i + 2, \cdots$，这种 index 我们为了简化操作，

直接对其输出 if 语句来做 bound check；最后就是常量索引。具体的数据结构如图1.2所示。由于不同的 tensor 之间会共享索引，所以我们单独创建了一个索引表，如果当时解析到的索引已经存在于索引表当中，那我们只会检查索引的上下界，并取他们之间的交集，否则创建一个新的索引加入索引表当中。这样就能够完成 tensor 的 index 之间的范围的共享和更新。

4. 为了能够将中缀表达式转为后缀，我们首先建立一个栈，当我们解析 kernel 解析到运算符的时候，我们会对于将其和栈顶部的元素比较，如果该运算符的优先级大于栈顶运算符的优先级时，将其压栈，否则将栈顶运算符弹出并放入输出的 vector 当中，直到栈为空或者栈顶元素优先级较低。通过以上的操作，当解析完第一遍的 kernel 的时候，我们输出的 vector 中就存放好了当前 kernel 的后缀表达式，用于建立 IR tree，例如当输入的 kernel 为 A[i][j] = B[i][j] + C[i][j]，则通过第一遍 pass 之后，则在输出的 vector 中存放的顺序就是 [var_1,var_2,var_3,+,=]，其中 var_1,var_2,var_3 分别是变量 A,B,C。

2 构建 IR tree

首先通过不断处理后缀表达式，就能够建立一个 stmt，对于 kernel 中用分号分割的句子，我们会建立两颗独立的 IR tree，并为其建立不同的 Stmt，最后将他们加入一个 group，形成一个完整 IR tree。从后缀表达式构建 Stmt 的算法如 Algorithm 1所示。如果遇到 tensor 或者常量就需要将其压入表达式栈中，如果遇到非 “+、−、=” 之外的运算符，则直接将其压栈，如果遇到 “+/−”, 则需要不断弹栈，直到找到其两个操作数，这个操作数可能是一个子表达式，然后对于两个操作数进行规约判断，如果他不需要规约，则直接利用运算符对其进行树节点的合并操作，否则需要就进行规约，根据爱因斯坦求和规则，需要创建一个临时变量，规约既是将将其求和赋值给一个临时变量，然后将这个临时变量的 Expr 返回，如果是 “=”, 则需要判断栈中元素的数目，如果栈中元素数目大于 2，则表明栈中还有多操作数，需要对其如上述规约判断和操作，完成之后，则栈中只剩下两个变量，然后对其进行等号赋值连接即可。遍历完整个后缀表达式的 vector，然后构建好了一句 kernel 语句的 StmtNode。

由上述不同部分，可以组成一个完整的算法如 Algorithm 2，首先就是将 kernel 根据分号分割开，然后对于子语句的表达式，我们通过 Sec 1的方法，

Algorithm 1 后缀表达式建立 stmt 的 IR Node

输入: 后缀表达式 SuffixVector

输出: IRNode::Stmt stmt

```
1: function ConstructStmtNode(SuffixVector)
2:     创建表达式栈 ExprStack，存放 tensor 和常量的 Expr
3:     for 遍历 SuffixVector do
4:         if 读到变量 var_x then
5:             ExprStack.push(var_x)
6:         else if 读到常量 imm_x then
7:             ExprStack.push(imm_x)
8:         else 读到运算符 op
9:             if op == "=" then
10:                if ExprStack.size ≠ 2 then
11:                    varVector = ExprStack[1:]
12:                    Expr var_1 = produce_reduction_code(varVector)
13:                    while ExprStack.size ≠ 1 do ExprStack.pop()
14:                    end while
15:                    Expr var_2 = ExprStack.pop()
16:                else
17:                    Expr var_1 = ExprStack.pop()
18:                    Expr var_2 = ExprStack.pop()
19:                end if
20:                Stmt stmt = Move::Make(var_2， var_1， MoveType::MemToMem)
21:            else
22:                if op == "+/-" then
23:                    find two operator varibles' set varVector_1,varVector_2 of op
24:                    pop the these varibles from ExprStack
25:                    Expr var_1 = produce_reduction_code(varVector_1)
26:                    Expr var_2 = produce_reduction_code(varVector_2)
27:                    exp = Binary::make(data_type, BinaryOpType::op, var_1, var_2)
28:                    ExprStack.push(exp)
29:                else
30:                    ExprStack.push(op)
31:                end if
32:            end if
33:        end if
34:    end for
35:    return stmt
36: end function
```

Algorithm 2 kernel 语句构建 IR tree

输入: kernel

输出: IRtree

```
1: function ConstructTree(kernel)
2:     StringVector ← kernel.split(";")
3:     for patchkernel : StringVector do
4:         SuffixVector = buildSuffixExpression(patchkernel)//构建后缀表达式
5:         Stmt stmt = ConstructStmtNode(SuffixVector)
6:         IfcheckBound = cond_gen(包含左值单字 index 的复合 index 子集)
7:         Stmt elseif = IfThenElse::make(IfcheckBound, stmt)
8:         Stmt main_stmt = LoopNest::make(左值 indexes, elseif)
9:         MainStatement.push_back(main_stmt)
10:    end for
11:    inputs,outputs = create_io(kernel) //用于创建函数签名
12:    Group IRtree = Kernel::make(kernel.name, inputs, MainStatement)
13:    return IRtree
14: end function
```

对其进行预处理，并得到对应的后缀表达式的 vector，然后通过 Algorithm 1 建立每一句的 stmt，然后为其加上 if 语句的 bound，然后添加其外层的 for 循环，形成一个 main stmt，当所有子语句都处理完成后，将其合并成一个 group，就构成成为了一棵 IR tree。

3 转为 C++ 代码

这个部分通过修改 IRPrint 代码，使其在访问不同节点的时候输出相应的语句，符合 C++ 的语法定义并去掉非必要的类型输出，对于每一棵构建好的 IR Tree，调用 IRprint 对其进行深度优先遍历，然后就能够得到 kernel 对应的 C++ 代码。

4 分工

三人共同讨论设计算法和相关数据结构，共同完成代码实现和调试以及报告的书写。所有成员全程参与。