Peking University

Compiler Project-2 Report

胡镇炜 1700012795
陈燕琼 1700012716
徐锦成 1700012919
刘陈晓 1700012942

20-06-2020

Contents

1. 输入处理与 IRParser

负责人：胡镇炜

1.1 设计思路

在 Project2 中，我们仍然需要从 json 文件读取数据，因此我们沿用了 Project1 中设计的 IRParser，由于 json 文件中新增了一个 “grad_to” 的键值，因此我们对 IRParser 在原来的基础上稍加修改，令其提取相应的 grad 信息。

1.2 实现方法

在 parse.cc 中有如下主要函数：

函数名	处理对象	输入	返回值
parse_kernel	完整式子	$A = B * C + D$	vec<vec<Expr> >
parse_factor	项	$B * C$	Expr
parse_av	下标	$[i + j, p - q]$	vec<Expr>

parse_kernel 以 $= +-$; 为分隔符，分割项调用 parse_factor 处理。

parse_factor，parse_av 都使用了表达式求值的算法，用两个栈 id，op 分别存储表达式和运算符，只是处理的粒度不同。parse_factor 遇到标识符，调用 parse_av 处理下标，返回后构造 Var 对象存入 id。parse_av 以 ,] 为分隔符，处理下标中的表达式。

parse.cc 中定义的一些辅助函数如下：

函数名	功能
getstr(char*, char*)	获取子串
parse_id(char*)	处理 ins,outs 和 grad_to
parse_int(char*, int&)	读取数字
isop(char)	判断是否为运算符
pri(char)	定义优先级
compute(Expr&, Expr&, char)	二元运算

parse_kernel 以 record 结构体的形式返回，并交由下一步处理:

```
struct record
{
    vector<string> in;
    vector<string> out;
    vector<string> grad;
    string name;
    string type;
    vector<vector<Expr>> vs; // 分割好的项，包括等号左边和右边
};
```

2. 求导规则的设计与 GradGen

负责人：陈燕琼

2.1 设计思路

在得到了 IRParser 输出的 record 结构之后，需要设计求导规则来计算出求导表达式。为了生成求导表达式所需的 record 结构，我们新增了 GradGen.h(cc) 和 MyMutator.h(cc) 文件。其中，MyMutator 类与 IRMutator 类相似，是对 IRMutator 类的继承。

2.2 实现方法

2.2.1 MyMutator.h(cc)：辅助完成最终求导语句中各项的构建

与 IRMutator 类相比，MyMutator 类在 MyMutator.h 中添加如下成员：

```
string grad;
Expr dx;
int cnt;
void set_grad(string& s)
```

其中 grad 记录当前求导变量，dx 记录需要求导变量的导数形式，cnt 用来表示当前试图对表达式中第几个 x 进行求导（一个表达式可能出现多个 x），set_grad 函数设置 grad 变量。

MyMutator 中涉及到的重要函数如下：

1. MyMutator::visit(Ref<const Binary> op): 该函数在遍历到一个二元表达式（即包含 +-*/）的时候，会根据二元运算符的种类进行区别，然后生成对应的求导表达式。对于 x+y 和 x-y 表达式，返回的求导表达式分别为 dx+dy 和 dx-dy；对于 x*y 表达式，返回的求导表达式为 dx*y+x*dy；对于 x/y 表达式，返回的求导表达式为 dx/y(经过简化，因为在 case 中 y 总是常数)。

2. MyMutator::visit(Ref<const Unary> op): 该函数在遍历到一个一元表达式（仅考虑-x 形式）的时候，返回求导表达式-dx。

3. MyMutator::visit(Ref<const Var> op)：该函数在遍历到一个变量的时候，会首先通过变量名判断该变量是不是需要求导的变量，如果不是则返回 0；否则令 cnt 减 1，此时如果 cnt 为 0 则说明是当前需要求导的变量，构建该变量的导数形式存储在 dx 中并返回 1，否则返回 0。

其余函数的实现与 IRMutator.cc 中的实现基本类似。

2.2.2 GradGen.h(cc)：基于对输入的 case 进行解析得到的 record 结构，生成相应的求导语句所需的 record 结构

我们新增了 GradGen.h 和 GradGen.cc 两个文件，用来声明并定义生成求导表达式的接口函数 GradGen(const record js, record result)。该函数接受一个 record 类型的引用 js 为参数，将生成求导表达式之后的结果存放到另一个 record 类型 result 中。

GradGen 函数：首先基于原始的 record 结构构造出求导函数所对应的 in，out，type，name 等基本信息，然后对于每一个需要求导的变量，调用 get_grad 函数生成所需的求导语句对应的表达式项。

get_grad 函数：对于 js.vs 中记录的原语句等号右边的每一个爱因斯坦求和项，通过 3 层内层循环（一个表达式中同一个变量最多出现 3 次，判断此时在对第几个变量求导）使用辅助的 MyMutator 对象完成项的转换得到 Expr 类型的 ret，再将对应的求导项 ret*dy（y 是原语句的 dst 变量）添加到 result 中。

3. 中间节点的构建与 IRGenerator

负责人：徐锦成

3.1 设计思路

在经过 IRParser 对输入进行处理，以及设计了相应的求导法则之后，需要根据 record 结构中保存的变量来构造 IR 中间节点。在一个 record 结构中，in 和 out 指示了输入和输出变量的名字，grad 指示了要求导变量的名字，name 指示了要构造的函数体的名字，type 指示了函数体中的数据类型，vs 指示了所有的语句，对于 vs 中的每一条语句，是以 vector<Expr> 的形式存储的，其中第一个 Expr 是等号左边变量对应的表达式，之后的 Expr 是等号右边每一个爱因斯坦求和项对应的表达式。

为了对 vs 中的语句进行解析，需要修改 IRVisitor 类，为其添加相应的数据结构来记录语句中的相关信息，然后依次访问每一条语句中的每一个表达式，根据 visitor 记录下来的信息来构建最终的函数 kernel。

3.2 实现方法

在 IRVisitor.h 中为 IRVisitor 类添加如下成员变量：

```
std::unordered_map<std::string, std::pair<int, int> > index_mp;
std::vector<std::vector<std::pair<Expr, int> > > needIf;
std::unordered_map<std::string, std::vector<int> > var_dims;
std::vector<std::set<std::string> > termIndex;
std::vector<std::string> left_indexes;
bool enterR;
int ti;
bool needRep;
```

index_mp 中记录的是 < 下标名，下标范围 > 二元对；needIf 中记录的是每一项中所有需要进行范围检查的 < 下标表达式，检查上界 > 二元对；var_dims 中记录的是 < 变量名，变量大小 > 二元对；termIndex 中记录的是等号右边每一项中没有在等式左边出现过的下标名；left_indexes 记录的是等号左边出现过的下标名；enterR 用来判断当前在等号的左边还是右边；ti 用来记录当前处理的是第几项;needRep 记录左值下标是否出现复杂表达式。

在 IRVisitor.cc 中修改 IRVisitor::visit(Ref<const Var> op) 函数，首先对 op->shape 进行检查，判断当前变量是否为标量，如果是则不用进行后续的下标处理，将该标量及对应范围添加进 var_dims；否则，该变量指示的是一个数组，需要对其下标进行处理。对 op->args 中的每一项，即该变量的每个下标，首先判断它是否为 Index 节点类型，如果是则表明该下标是一个一元下标形式；否则为下标的组合形式如 i + j 等等。

对于下标的组合形式，需要在生成的函数体中为其添加界限检查，因此将其加入 needIf；此外与 project1 不同的是，赋值语句左边的变量下标可能出现复杂表达式的形式，因此也需要注意通过 visit 该复杂表达式，将表达式中的 index 变量添加到 left_indexes 中，并置 needRep 为 true。对于一元下标，会根据其在等号的左边或右边分别加入 left_indexes 或 term_indexes，然后为下标构造范围加入 index_mp，需要注意的是这里会不断更新 index_mp 中该下标的范围以使得该范围满足所有该下标出现时的要求。

IRGenerator.h 和 IRGenerator.cc 两个文件声明并定义了生成中间节点的接口函数 IRGenerator(record& js)。该函数接受一个 record 类型的引用 js 为参数，首先根据 js.type 确定函数体的数据类型。之后对于 js.vs 中的每一个表达式，调用 genStmt 函数，该函数的作用就是通过调用 IRVisitor 来遍历表达式，并根据遍历过程中保存的信息来构建下标信息和循环函数体。其具体的工作如下：

1. 构造将 dst 变量赋值给新建 tmp 变量的表达式，这里需要注意为 dst 变量创建新的循环变量；

2. 对于每一个爱因斯坦求和项，利用 IRVisitor 遍历后保存的信息 index_mp 和 term_index 来构建该项私有的下标表达式，并在 tmp 变量上累加该项，随后使用 IfThenElse 语句嵌套构造 needIf 中所需的 bound check，若需要内部循环则根据该求和项私有的下标构造 LoopNest 语句；这里还需要注意的是，由于要求左值下标不允许为复杂表达式，因此需要生成临时下标变量替换复杂表达式；

3. 构造将 tmp 赋值回给 dst 变量的表达式（若 dst 为数组则需要循环体）。

在 genStmt 函数返回之后，根据 visitor 中保存的 var_dims 来构建函数签名中的输入和输出（注意处理重复变量），生成最终的函数 kernel 并返回。

4. 中间节点的遍历与 MyPrinter

负责人：刘陈晓

4.1 设计思路

在构建好中间节点之后，MyPrinter.h 和 MyPrinter.cc 通过对中间结点进行遍历，生成 C/C++ 源代码。其中，MyPrinter 类与 IRPrinter 类相似，是对 IRVisitor 类的继承。

4.2 实现方法

与 IRPrinter 类相比，MyPrinter 类在 MyPrinter.h 中添加如下成员变量：

```
bool print_claim;
bool get_begin;
```

print_claim 用来判断当前语句是否为声明语句（需要与 Move 语句相区别），get_begin 判断当前的函数参数是否为开头的第一个参数。

在 MyPrinter.cc 中，当访问对象为 Kernel 时，首先打印函数签名，参数使用引用形式传入，输入放在输出之前，再逐一访问 stmt_list，打印函数主体。涉及到的重要函数如下：

1. MyPrinter::visit(Ref<const Move> op) 函数：print_claim 的值由 op->src.defined() 确定，为真时打印声明语句，即访问 op->dsc；否则打印 move 语句，即 op->dsc = op->src。

2. MyPrinter::visit(Ref<const LoopNest> op) 函数：依次打印循环的条件和内容，即访问 op->index_list 和 op->body_list

3. MyPrinter::visit(Ref<const Index> op) 函数：若 print_range 为真，需要生成范围，则打印循环的括号里内容，包括访问 op->name 和 op->dom，否则仅打印 op->name。

4. MyPrinter::visit(Ref<const Var> op) 函数：分为打印参数、打印声明语句、打印其它语句三种情形。当打印参数时，需要打印参数类型，使用其引用格式，并通过 op->shape.size() 判断其为一维变量或是数组；当打印声明语句时，需要打印变量类型及变量名，并通过 op->shape.size() 判断其为一维变量或是数组；当打印其它语句时，打印变量名，并通过 op->args.size() 打印维度信息。

其余函数的实现与 IRPrinter.cc 中的实现基本类似。

5. 实验结果及举例分析

5.1 实验结果

实验共提供 10 个测试用例，在 solution2.cc 的 main 函数中依次读入测试用例的 json 数据，通过以上四个模块的处理，将最终的源代码输出到 kernels 目录下相应的文件。经测试，可以在 kernels 目录下产生正确的源代码文件内容，且测试用例全部通过。

5.2 举例分析

下面以测试用例中的 case10 为例解释所设计的求导技术的可行性和正确性。

case10.json 的内容如下：

```
{
    "name": "grad_case10",
    "ins": ["B"],
    "outs": ["A"],
    "data_type": "float",
    "kernel": "A<8, 8>[i, j] = (B<10, 8>[i, j] + B<10, 8>[i + 1, j] + B<10, 8>[i + 2, j]) / 3.0;",
    "grad_to": ["B"]
}
```

指明该用例的原始式的输入变量 B（10*8 的数组），输出变量 A（8*8 的数组）以及符合爱因斯坦求和规范的赋值语句，并指明需要求导的变量为 B。值得注意的是，这里虽然只有一个需要求导的变量，但是在原式中 B 共出现三次，因此需要在分别求导后进行加和。

根据我们设计的求导规则以及处理流程，具体来看对 B 进行求导的过程：

1. 在 GradGen 中进入以 “B” 为参数的 get_grad 函数，在其内部创建一个 MyMutator 的对象，设置该对象当前的求导变量为 “B”，并准备好变量 “dA”（A 即原式的目的变量）

2. 遍历原式每一个爱因斯坦求和项，这里只有一项，即 “((B<10, 8>[i, j] + B<10, 8>[i + 1, j]) + B<10, 8>[i + 2, j]) / 3.0”

3. 对当前求和项，内层循环共 3 次，第 i 次循环对项中第 i 个 B 变量进行求导

4. 第 1 次循环：对于第 1 个 B 求导，使用 MyMutator 对求和项进行 mutate。

（1）首先进入 MyMutator::visit(Ref<const Binary> op)，对 x/y 进行处理。这里我们对返回的求导表达式进行过简化，即只返回 dx/y，事实上这里由于 y=3.0 是常数，因此简化处理不会出现问题。返回前对 x（即 (B<10, 8>[i, j] + B<10, 8>[i + 1, j]) + B<10, 8>[i + 2, j]）的 mutate 进入了（2）。

（2）进入 MyMutator::visit(Ref<const Binary> op)，对 x+y 进行处理（x = B<10, 8>[i, j] + B<10, 8>[i + 1, j]，y = B<10, 8>[i + 2, j]）。对 x 与 y 分别完成 mutate 之后（（3）及（6）），返回 dx+dy（此处 dy = 0.0）。

（3）进入 MyMutator::visit(Ref<const Binary> op)，对 x+y 进行处理（x = B<10, 8>[i, j]，y = B<10, 8>[i + 1, j]）。与（2）大致相同，对 x 与 y 分别完成 mutate 之后（（4）及（5）），返回 dx+dy（此处 dx = 1.0，dy = 0.0）。

（4）进入 MyMutator::visit(Ref<const Var>，处理 B<10, 8>[i, j]。由于该变量是需要求导的变量（而且是这一轮循环中需要求导的第一个 B），因此创建一个相应的 dB 变量存入 MyMutator 的成员变量 dx 备用，并返回 1.0。

（5）进入 MyMutator::visit(Ref<const Var>，处理 B<10, 8>[i + 1, j]。由于该变量不是需要求导的变量（是第 2 个 B），因此返回 0.0。

（6）进入 MyMutator::visit(Ref<const Var>，处理 B<10, 8>[i + 2, j]。由于该变量不是需要求导的变量（是第 3 个 B），因此返回 0.0。

（7）最后经过 MyMutator 得到的表达式即为 “(((1.0 + 0.0) + 0.0) / 3.0)”

5. 与第 1 次循环类似，第 2 次循环和第 3 次循环得到的表达式分别为 “(((0.0 + 1.0) + 0.0) / 3.0)” 和 “((0 + 1.0) / 3.0)”

6. 在一次循环中得到 MyMutator 输出的表达式 ret 之后，创建 Binary 表达式 ret*dA，比如第 3 次循环中创建得到 “(((0 + 1.0) / 3.0) * dA)”，则对于当前求导的第 i 个变量，构造得到了求导语句所需的左值（即 MyMutator 存储的 dx）以及各个爱因斯坦求和项

7. 由于对于 3 个 B 的求导语句的左值是相同的，在 IRGenerator 中会产生 tmp 临时变量存储原值并对值进行累加，因此最终可以实现对原语句的 B 变量的求导

最终生成的源代码如下：

```
#include "../run2.h"

void grad_case10(float (&dA)[8][8], float (&dB)[10][8]) {
  float tmpdB1[10][8];
  for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 8; ++j) {
      tmpdB1[i][j] = dB[i][j];
    }
  }
  for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      tmpdB1[i][j] = (tmpdB1[i][j] + ((((1.0 + 0.0) + 0.0) / 3.0) * dA[i][j]));
    }
  }
```

```
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 8; ++j) {
    dB[i][j] = tmpdB1[i][j];
  }
}
float tmpdB2[10][8];
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 8; ++j) {
    tmpdB2[i][j] = dB[i][j];
  }
}
for (int i = 0; i < 8; ++i) {
  for (int j = 0; j < 8; ++j) {
    int u;
    u = (i + 1);
    int v;
    v = j;
    if (((i + 1) < 10 && (i + 1) >= 0)) {
      tmpdB2[u][v] = (tmpdB2[(i + 1)][j] + ((((0.0 + 1.0) + 0.0) / 3.0) * dA[i][j]));
    }
  }
}
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 8; ++j) {
    dB[i][j] = tmpdB2[i][j];
  }
}
float tmpdB3[10][8];
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 8; ++j) {
    tmpdB3[i][j] = dB[i][j];
  }
}
for (int i = 0; i < 8; ++i) {
  for (int j = 0; j < 8; ++j) {
    int u;
    u = (i + 2);
    int v;
    v = j;
    if (((i + 2) < 10 && (i + 2) >= 0)) {
      tmpdB3[u][v] = (tmpdB3[(i + 2)][j] + (((0 + 1.0) / 3.0) * dA[i][j]));
    }
  }
}
for (int i = 0; i < 10; ++i) {
  for (int j = 0; j < 8; ++j) {
    dB[i][j] = tmpdB3[i][j];
  }
```

```
63   }
64 }
```

实现了正确的自动求导，可见所设计的求导技术是可行且正确的。

6. 编译知识与总结

6.1 相关编译知识

实现本次大作业的基本思想来自于书本知识。在课程中，我们分析了如何实现一个简单的语法制导翻译器，其输入为程序设计语言，如 Java 语言，通过该编译器，翻译得到三地址代码，再由三地址代码生成机器指令。书本中的编译器，通过词法分析、语法分析、中间代码生成、目标代码生成等步骤，一步步实现生成机器指令的目标。

大作业的编译器，与书本中的编译器不同之处，在于其输入 json 形式文件，其目标是生成自动求导的式子。但在实现思路上，我们需要应用书本上的理论知识，并在实践中对书本方法进行调整和改进，使得算法能够正确执行。

总的来看，自动求导的编译器中应用的书本知识可以分为以下几个方面：

1. 词法分析

大作业参照常规的编译器，首先对给定的 json 文件进行词法分析。常规的编译器中，词法分析这一步，通常包括剔除空白和注释、预读、处理常量、识别关键词和标识符等等，并创建符号表，以此保存有关源文件构造的各种信息。在大作业中，我们应用书上的知识，从 name、ins 等等域中提取出想要的关键信息，并通过 Record 的数据结构进行保存。即实践了书上的词法分析步骤。

2. 语法分析

在书本中，语法分析将词法分析出来的内容转化成树形的表达形式，构建出树形结构。大作业依据这一思想，基于 record 结构创建一个新的包含求导表达式的抽象语法树。在“求导规则的设计与 GradGen”部分，得到了 IRParser 输出的 record 结构之后，需要设计求导规则来计算出求导表达式，为了生成求导表达式，我们创建新的包含求导表达式的 AST，这体现出语法树构建的思想。

3. 中间代码生成

书中的“中间代码生成”一章，介绍了翻译方案，包括类型检查、翻译声明语句、翻译控制流等等。依据书中的思路，在大作业中，我们通过 MyMutator 和 IRGenerator，对语法树进行了遍历，并据此生成中间表示。其中充分体现了 Visitor 模式的设计思想，分离了类和操作，将不同类的相同操作汇总在一个 visitor 中，用 visitor 中函数的重载替换类的多态。以往的设计方式，为一个类增加操作的时候，需要为父类增加一个抽象方法，所有子类再实现各自的方法，这样就需要修改所有的子类，极为不便。采用 Visitor 模式，极大便利了修改，只需要新设计一个 visitor，在该 visitor 中实现不同的操作即可。通过此次大作业，我们更加深入地理解了 Visitor 设计模式。

4. 目标代码生成

由书本知识，我们了解到编译器的前端构造出源程序的中间表示，后端根据这个中间表示生成目标程序。中间表示分为两种，一种是树形结构，包括语法分析树和抽象语法树，另一种是线性表示形式，如三地址代码。在本次大作业中，我们参照书中的内容，从抽象语法树的中间表示生成自动求导的式子。同样利用 Visitor 设计模式，我们使用 MyPrinter 对语法树遍历，翻译为目标代码，对编译器的后端进行了实践。

6.2 总结

通过大作业，我们了解到各个深度学习的实际应用中的求导方式，包括 element-wise 的乘法、矩阵乘法、dense MTTKRP、二维普通卷积、转置、flatten、broadcast、blur。与此同时，将书本中知识与工程实践相结合，我们加深了对编译原理的理解，是对于平时学习的很好补充。