Project2

邬涵博 徐昊天 刘镇

概述与分工

Project2要求我们在之前的基础上实现C++语言的自动求导器。为了方便项目的并行，我们进行了如下的分工，这样分工的依据详见下面的问题分析。具体分工如下：

邬涵博(@EverNebula)：因子替换、Index转换

徐昊天(@Innov12942)：函数签名生成、链式求导、消除常数项

刘镇(@todoumon)：case分析、语句拆分

github：https://github.com/EverNebula/CompilerProject-2020Spring

（使用了submodule jsoncpp）

问题分析

因为对于向量求导不甚了解，我们小组选择首先对所有的case进行一个概览，并将其中需要实现的功能以及难点总结出来，方便代码的编写和分工。

- 因子左右替换(common) `A=B*C => dB=dA*C`
- 消除常数项(case 1) `A=B*C+1 => dB=dA*C`
- 多变量求导(case 4) `A=B*C => dB=dA*C;dC=dA*B`
- 语句拆分(case 10) `A=(B1+B2)/3 => dB=dA/3;dB+=dA/3`
- index转换(case 6,8,10) `h=p+r => r=p-h`

针对上述五个问题，由于需要多道pass，我们最终决定拆分为两个Mutator来解决，第一个Mutator为 **DerivMutator** ，用来处理前四个问题；第二个Mutator为 **IndexMutator** ，它们都位于 `project2/solution/derivmutator.cc` 中。代码的输出则依旧使用之前所编写的 **IRPrinter** 。下面将对于这两个Mutator的结构及实现作介绍。

DerivMutator

函数签名生成

观察函数签名的参数规律，发现顺序是输入矩阵、输出矩阵的导数、目标矩阵的导数。

输出矩阵的导数、目标矩阵的导数可以直接根据json文件描述得到，但输入矩阵需要进行一定的推断才能得到，如case2中的A既需要作为目标矩阵的导数也需要作为输入矩阵。

```
void grad_case2(float (& A)[4][16],float (& dB)[4][16],float (& dA)[4][16])
```

其他可能存在部分输入矩阵不需要等情况，解决方式是在通过DerivMutator修改AST后，再通过一个继承IRVisitor的子类MyVisitor来遍历新的AST，然后在重载的visit中统计所有用到的矩阵变量，并记录在usedVar中。

```
class MyVisitor : public IRVisitor{
    public:
        MyVisitor() : IRVisitor(), usedVar({}){};
        std::set<std::string> usedVar;
        void visit(Ref<const Var> op) override;
        void visit(Ref<const Kernel> op) override;
};
```

usedVar中记录的矩阵包括原有矩阵和导数矩阵，我们将其与input中的矩阵求交集，结果中的顺序按照input中的顺序，由此可以推断出参数中前半部分需要的矩阵有哪些，并可以生成完整的函数签名。

链式求导

因子左右替换、消除常数项都在求导过程中完成。

DerivMutator主要就是用于对原AST的求导过程，在其重载的visit函数中完成。

每次求导只针对一个目标，即多个目标就在多个DerivMutator中完成，得到多个求导后的AST，如case4，B、C就是分开求导，对应两个不同的AST，在最终结果中也是在不同循环体中。

```
void grad_case4(float (& B)[16][32],float (& C)[32][32],float (& dA)[16]
[32],float (& dB)[16][32],float (& dC)[32][32]){
  memset(dB, 0, sizeof dB);
  memset(dC, 0, sizeof dC);
  for (int i = 0; i < 16; ++i) {
    for (int j = 0; j < 32; ++j) {
      for (int k = 0; k < 32; ++k) {
        dB[i][k] += (dA[i][j]) * (C[k][j]);
      }
    }
  }
  for (int i = 0; i < 16; ++i) {
    for (int j = 0; j < 32; ++j) {
      for (int k = 0; k < 32; ++k) {
        dC[k][j] += (B[i][k]) * (dA[i][j]);
      }
    }
  }
}
```

求导过程：

- Imm 常数项：返回Expr(0)

- Var 矩阵变量：当访问到一个矩阵变量时，我们首先需要判断该矩阵是否是相关的矩阵，若是无关矩阵，则当作是常数项，若是目标矩阵或是输出矩阵，则生成新的导数矩阵变量，然后通过全局的记录进行左右交换。
- Binary 表达式：Binary是求导的关键，题目中的式子如

  ```
  C<4, 16>[i, j] = A<4, 16>[i, j] * B<4, 16>[i, j] + 1.0
  ```

  都可以看作多个Binary的复合，其中用到的运算符op包括+、-、*、/等，我们将一个Binary看作一个复合函数 $f = g\ op\ h$，然后根据op对应的求导法则进行求导，最终得到f的导数。在求导过程中可能会出现导数为0的情况，我们若发现g'或是f'为0，则将最终的求出的为0项直接消去，通过此种方法达到消除常数项的目的。

语句拆分

对于case10，如下

```
A<8, 8>[i, j] = (B<10, 8>[i, j] + B<10, 8>[i + 1, j] + B<10, 8>[i + 2, j]) /
3.0;
```

我们发现B有三项，但它们下标不同，我们并不能将这三项同等对待，相比于推断出每项对应的dA下标并合并一次求出B，我们选择在三条语句中完成B的计算。

```
for (int dB0 = 0; dB0 < 10; ++dB0) {
    for (int j = 0; j < 8; ++j) {
      if ((dB0 - 2 >= 0) && (dB0 - 2 < 8)) {
        dB[dB0][j] += ((dA[dB0 - 2][j]) * (3)) / ((3) * (3));
      }
      if ((dB0 - 1 >= 0) && (dB0 - 1 < 8)) {
        dB[dB0][j] += ((dA[dB0 - 1][j]) * (3)) / ((3) * (3));
      }
      if ((dB0 >= 0) && (dB0 < 8)) {
        dB[dB0][j] += ((dA[dB0][j]) * (3)) / ((3) * (3));
      }
    }
}
```

解决方法如下：

首先，在第一次访问到LHS=RHS的Move结点时，若我们遇到第一个B时，则记录下它为当前的目标curTargetVar，然后遇到其他的B，我们实现了比较函数compare，发现这些B不同，于是将他们保存在allTargetVars中。

```
bool exprcompare(Expr a, Expr b);
bool varcompare(Ref<const Var> a, Ref<const Var> b);
bool binarycompare(Ref<const Binary> a, Ref<const Binary> b);
bool strimmcompare(Ref<const StringImm> a, Ref<const StringImm> b);
bool intimmcompare(Ref<const IntImm> a, Ref<const IntImm> b);
bool indexcompare(Ref<const Index> a, Ref<const Index> b);
```

在完成一次访问Move后，我们在LoopNest中检查allTargetVars，发现不为空，则更新当前目标，并将其从allTargetVars中移除，再次访问Move，直到全部访问完。于是我们的到多个求导后的新Move，在上述例子中我们的到三个新的求导后的Move，于是在最终结果中我们得到三条赋值语句计算B。

IndexMutator

具体效果

尽管DerivMutator输出的结果已经可以正确得到答案，但是对于赋值语句左边的Index会存在计算，这对于时间局部性是很不友好的。因此有了IndexMutator再次对AST遍历，解决最后的index转换问题。

```
  for (int k = 0; k < 8; ++k) {
    for (int n = 0; n < 2; ++n) {
      for (int p = 0; p < 5; ++p) {
        for (int q = 0; q < 5; ++q) {
          for (int c = 0; c < 16; ++c) {
            for (int r = 0; r < 3; ++r) {
              for (int s = 0; s < 3; ++s) {
                dB[n][c][p + r][q + s] += (dA[n][k][p][q]) * (C[k][c][r][s]);
              }
            }
          }
        }
      }
    }
  }
```

上面是DerivMutator输出的结果，而经过IndexMutator，它将会变为：

```
  for (int k = 0; k < 8; ++k) {
    for (int n = 0; n < 2; ++n) {
      for (int p = 0; p < 5; ++p) {
        for (int q = 0; q < 5; ++q) {
          for (int c = 0; c < 16; ++c) {
            for (int dB2 = 0; dB2 < 7; ++dB2) {
              for (int dB3 = 0; dB3 < 7; ++dB3) {
                if (((dB2 - p >= 0) && (dB2 - p < 3)) && ((dB3 - q >= 0) &&
(dB3 - q < 3))) {
                  dB[n][c][dB2][dB3] += (dA[n][k][p][q]) * (C[k][c][dB2 - p]
[dB3 - q]);
                }
```

```
                }
              }
            }
          }
        }
      }
    }
```

可以发现，它用`dB2,dB3`代替了原来的`r,s`，消除了左边下标的计算。同时对应下表的变换，加入了If语句来防止数组访问越界。

牺牲Index

我们目的是消除左值向量下标计算，对于这些存在计算的下标。所以我们的思路是，对于每一维，寻找参与下标计算的一个Index作为牺牲者，将这一维用一个新的Index来替换。比如对于`dB[n][c][p+r][q+s]`中，存在两个维度有下标计算，因此我们定义两个新Index，分别为`dB2=p+r`，`dB3=q+s`。通过这两个式子我们可以反推出`r=dB2-p`，`s=dB3-q`。

`r`与`s`的反推我们使用如下的方法。为了更加具体地介绍，下面的例子为通过`a=b+c*4`反推`c=(a-b)/4`。`b+c*4`的AST需要进行的变换如下：

```
   +                /
  / \              / \
 b   *     =>     -   4
    / \          / \
   c   4        a   b
```

我们通过栈记录下从根到牺牲节点c的一条路径，并同时记录下另一边节点的值，栈中信息为[$, (*, 4), (+, b)]。可以发现，变换后的AST其实就是按 **栈的顺序** 使用 **逆运算** 构建而成。根据栈信息，在`Var`中一次性构建AST即可。

记录下这个牺牲Index的计算表达式后，将它存入Map `replaceExpr`中，之后在右值中访问到这个Index时直接替换为计算后的Expr。

取模的处理

在case 8中，出现了对于下标取模的处理，`flatten`等函数中才能使它有意义，在这里我们默认出现取模就是发生了shape的改变。因此如果出现式子`x=Expr mod k`，会直接取处理`Expr`而先忽略`k`。比如case 8的`dA[i/16][i%16]`，我们要将其变为`dA[dA0][dA1]`。在遍历`dA0=i/16`时，我们得到了`i`的计算表达式`dA0*16`，在`dA1=i%16`中，我们发现`i`再次成为了牺牲Index，并且这次运算包含取模，因此就在原计算式基础上加上这次的结果，表示让原计算式取模结果正确，`i=dA0*16+dA1`。

由于shape发生改变，上面检测到重复的牺牲Index时，LoopNest中也要检查并加上所有的新Index。

边界条件

下标发生了变换，需要额外的边界检查，比如case 10中同时有`i`,`i+1`,`i+2`，如果粗暴地转换为`dB`,`dB-1`,`dB-2`，而外层循环还是从0-10，就会发生数组越界。

解决方法是如果右值中出现了一个牺牲Index，原范围为[a,b]，那么就在这个Move外套一个If语句，条件为`Expr>=a && Expr<=b`，`Expr`为之前得到的计算表达式。

```
Stmt IndexMutator::visit(Ref<const Move> op)
{
  ...
  return IfThenElse::make(cond, mov, Stmt());
}
```

具体例子与实验结果

实验代码在Mac OSX操作系统上，编译器为clang-1100.0.33.17。

case 10是一个比较全面的例子，下面给出case 10在不同阶段的代码：

- 初始表达式如下

```
  for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      A[i][j] += (B[i][j] + B[i + 1][j] + B[i + 2][j]) / (3);
    }
  }
```

- 求导并拆分语句

```
for (int i = 0; i < 8; ++i) {
    for (int j = 0; j < 8; ++j) {
      dB[i + 2][j] += ((dA[i][j]) * (3)) / ((3) * (3));
      dB[i + 1][j] += ((dA[i][j]) * (3)) / ((3) * (3));
      dB[i][j] += ((dA[i][j]) * (3)) / ((3) * (3));
    }
  }
```

- Index转换后

```
for (int dB0 = 0; dB0 < 10; ++dB0) {
    for (int j = 0; j < 8; ++j) {
      if ((dB0 - 2 >= 0) && (dB0 - 2 < 8)) {
        dB[dB0][j] += ((dA[dB0 - 2][j]) * (3)) / ((3) * (3));
      }
      if ((dB0 - 1 >= 0) && (dB0 - 1 < 8)) {
        dB[dB0][j] += ((dA[dB0 - 1][j]) * (3)) / ((3) * (3));
      }
      if ((dB0 >= 0) && (dB0 < 8)) {
        dB[dB0][j] += ((dA[dB0][j]) * (3)) / ((3) * (3));
      }
    }
  }
```

对于不同情况的解释详见上文举出的各种例子。

实验结果为十个测试点全部通过：

```
➜  build git:(master) ./project2/test2
Random distribution ready
Case 1 Success!
Case 2 Success!
Case 3 Success!
Case 4 Success!
Case 5 Success!
Case 6 Success!
Case 7 Success!
Case 8 Success!
Case 9 Success!
Case 10 Success!
Totally pass 10 out of 10 cases.
Score is 15.
```

总结

在整个Project中，我们活用一学期学习的编译知识，搭建了一个简易可行的自动求导器。具体涉及知识点如下：

- 词法分析：主要在Project1中。我们首先对输入串进行了一定的处理，比如将占两个字符的`//`运算符替换为`$`方便处理。由于输入比较有规律性，生成Token的过程与语法分析是同步进行的。
- 语法分析：主要在Project1中。我们使用语法规律人工实现了一个简单的解析器，位于`parser.cc`中。
- 中间代码与语法制导翻译：通过上面自顶向下的语法分析，我们生成了一棵抽象语法树作为中间代码表示。为了对语句进行求导，则需要对AST进行多道pass，并更改一部分结构。需要注意的是，翻译过程并不一定是L型的，因为我们有时会先访问左值，有时会先访问右值，但我们的设计保证

了其不存在循环的求值。代码位于`derivmutator.cc`中。

- 代码生成：有了改造后的AST，通过简单的遍历就可以输出对应的C++语句，位于`IRPrinter.cc`中。

通过一学期编译技术的学习和实践，我们感受到了编译原理的理性之美，也对计算机系统结构有了更深层次的认识和探讨，为以后的学习更是建立了扎实的基础。

在这里特别感谢梁云老师和各位助教的悉心教导与帮助，感谢小组各位成员之间的团结协作、互帮互助！