编译技术Project2报告 - 第26组

组内分工

朱立人：基于表达式变换的简单自动求导

宋苑铭：基于语法树变换的简单自动求导

魏龙：求导语句中变量代换

自动求导技术设计

首先我们先考虑element-wise的情况，这说明表达式中的每一个符号都是一一对应的。例如`A<16,32>[i,j]=B<16,32>[i,j] * C<16,32>[i,j]`，我们可以对每一个`[i,j]`视为一个`A=B*C`的表达式，这样如果我们计算`dB`，可以根据多项式求导有

$$dB = \frac{\partial loss}{\partial B} = \frac{\partial loss}{\partial A}\frac{\partial A}{\partial B} = dA\frac{\partial A}{\partial B}$$

由于`dA`已知，所以只需要根据`A=B*C`对`B`求导即可，这样element-wise的情况就变成了简单的多元函数求导的过程

有了对于element-wise的情况的分析，我们就可以进一步探究表达式之间的关系。之后我们就会发现撤销了element-wise，其实只是没有了一一对应的关系而已，我们只需要对求导的变量和找出和哪些输出变量的有关即可。

如果类似矩阵乘法这样的情况，每个标识符其后的下标相同，也可以直接使用以上的方案求导，例如`A<16,32>[i,j]=B<16,16>[i,j] * C<16,32>[j,k]`也可以看成是`A=B*C`的多元函数求导，因为数学上，

$$dB_{i,j} = \frac{\partial loss}{\partial B_{i,j}} = \sum_j \frac{\partial loss}{\partial A_{i,j}}\frac{\partial A_{i,j}}{\partial B_{i,j}} = \sum_j dA_{i,j}C_{j,k}$$

最后我可以扩展到任意情况，目前所有的cases中如果标识符后面的下标不唯一，如case10，我们只需要将每一个对一个下标不同的标识符，在变量求导时视作不同的变量，所以对于case10我们可以将其看成`A=(B1+B2+B3)/3.0`，然后再将每个`B`对应的`A`相加即可生成求导的表达式。之后的报告将着重根据这个例子的简化版详细介绍我们的求导过程。

基于上面的分析，我们首先利用表达式变换设计求导的方案。由于这里的表达式运算以`+`和`*`为主，我们的设计以基本的求导规则为主针对`+`和`*`的情况求导。当然即使有除法也是可以解决的，但是由于测试集合中没有这种情况，没有特别设计，但是想要添加难度也不是太大。这里表达式的变换，其实也类似一种自顶向下的翻译方案，主要是通过字符串的替换和分析。得到表达式之后可以服用Project1的代码再生成代码。事实上通过表达式变换已经能够通过绝大多数的例子。但是为了更加直观的贴近编译技术，我们之后使用设计了基于语法树变换的自动求导方案。

所以我们最终的基本流程是，首先直接根据张量表达式变换设计了自动求导方案，然后改写得到基于语法树变换的简单自动求导编译器，并加入了对求导语句进行变量代换的功能，具体细节将在实现流程中详细叙述。

实现流程

以下总结的是基于语法树变换的自动求导技术实现流程。对于基于张量表达式变换的自动求导实现，参见project2/solution2.cc.

1. 使用jsoncpp解析输入json文件，得到测试数据的相关信息，并进行必要的预处理。复用project1中的代码，对表达式进行词法分析和语法分析，并构建IR树。
2. 对于每个“形式不同”的待求导变量，使用DiffMutator对语法树进行变换得到求导代码。比如，对 `A[i] = B[i] + B[i+1]`，`B[i], B[i+1]` 下标不完全相同，因而形式不同。根据求导的性质，将出现在表达式不同位置的变量看作不同变量求导，结果仍然正确。DiffMutator根据反向传播原理，将循环体内形如 `A[i] = A[i] + f(B[g(i)])` 的语句变为 `dB[g(i)] = dB[g(i)] + dA[i] * df(B[g(i)])`，不改变循环结构。求解 `df(B[g(i)])` 主要根据 `d(A + B) = dA + dB`，`d(AB) = AdB + BdA`，`d(kA) = kdA` 等求导规则逐层求导。使用SimplifyMutator对求导代码进行化简，以得到相对简洁的代码，并保证代码中出现的变量都会被实际用到。
3. 由于语句左侧下标索引上不能有运算，故需要进行变量代换。考虑到通用实现方法相对复杂，而测试用例形式较简单，故只考虑对下标上出现的单次加法、除法、取模进行相应代换。如对于 `A[i + j] = B[i] * C[j]`，令 `i + j = i0`，则原语句替换成 `A[i0] = B[i] * C[i0 - i]`；对于 `A[i / 16][i % 16] = B[i]`，令 `i / 16 = i0`，`i % 16 = i1`，则语句替换为 `A[i0][i1] = B[i0 * 16 + i1]`。整个过程基于 IRMutator 实现。
4. 在求导语句前添加初始化语句，生成函数签名，并复用project1中设计的IRPrinter，打印出kernel的C++代码。

实验结果

程序可以正确编译运行，并通过全部10个测试数据。分析出来的求导表达式是一个或多个赋值语句形式，每个语句左侧的下标索引上没有加减乘除等运算，简单修改输入也可以得到和有意义的符合逻辑的代码。

自动求导技术示例

我们讨论case 10的一个简化版，以解释设计的求导技术的可行性和正确性。

测试数据的变量类型为float，张量表达式为 `A<8, 8>[i, j] = (B<9, 8>[i, j] + B<9, 8>[i + 1, j]) / 2.0;`，希望求出 `dB`。根据求导的数学方法，我们有

$$dB[i,j] = \frac{\partial loss}{\partial B[i,j]} = \sum_k \frac{\partial loss}{\partial A[k,j]} \frac{\partial A[k,j]}{\partial B[i,j]} = \begin{cases} (dA[i-1,j] + dA[i,j])/2, & 0 < i < 9 \\ dA[i,j]/2, & i = 0 \\ dA[i-1,j]/2, & i = 9 \end{cases}.$$

下面是我们设计的自动求导技术求解 `dB` 的过程。首先，对表达式进行词法分析和语法分析，并构建IR树。得到的IR树对应的代码为

```
for (int i = 0; i < 8; i++) {
  if (i >= 0 && i < 8 && i < 9 && i + 1 >= 0 && i + 1 < 9) {
    for (int j = 0; j < 8; j++) {
      if (j >= 0 && j < 8) {
        A[i][j] = A[i][j] + ((float) (B[i][j] + B[i + 1][j])) / ((float) 2);
      }
    }
  }
}
```

注意生成的代码中，条件语句的谓词永远为真。可以看出，IR树对应的代码正确对应输入的张量表达式。使用DiffMutator分别对 `B[i][j], B[i+1, j]` 进行求导，生成两个与上面代码循环结构完全一致的LoopNest，但循环体分别变为

```
dB[i][j] = dB[i][j] + dA[i][j] * (0 + ((float) 1 + 0) / ((float) 2));
```

```
dB[i + 1][j] = dB[i + 1][j] + dA[i][j] * (0 + ((float) 0 + 1) / ((float) 2));
```

经过化简后，得到求导代码

```
for (int i = 0; i < 8; i++) {
  if (i >= 0 && i < 8 && i < 9 && i + 1 >= 0 && i + 1 < 9) {
    for (int j = 0; j < 8; j++) {
      if (j >= 0 && j < 8) {
        dB[i][j] = dB[i][j] + dA[i][j] * ((float) 1) / ((float) 2);
      }
    }
  }
}
for (int i = 0; i < 8; i++) {
  if (i >= 0 && i < 8 && i < 9 && i + 1 >= 0 && i + 1 < 9) {
    for (int j = 0; j < 8; j++) {
      if (j >= 0 && j < 8) {
        dB[i + 1][j] = dB[i + 1][j] + dA[i][j] * ((float) 1) / ((float) 2);
      }
    }
  }
}
```

目前，赋值语句的左侧下标还有加法运算。对第二个LoopNest，扫描到左侧下表有加法运算后，使用SubstituteMutator进行变量代换。令`i0 = i + 1`，得到

```
for (int i0 = 0; i0 < 9; i0++) {
  if (i0 - 1 >= 0 && i0 - 1 < 8 && i0 - 1 < 9 && i0 >= 0 && i0 < 9) {
    for (int j = 0; j < 8; j++) {
      if (j >= 0 && j < 8) {
        dB[i0][j] = dB[i0][j] + dA[i0 - 1][j] * ((float) 1) / ((float) 2);
      }
    }
  }
}
```

注意if语句将`i0`的取值范围限定到了`[1,9)`。 添加初始化语句和函数签名后，最终生成的完整求导代码为

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

void grad_case10(float (&dA)[8][8], float (&dB)[9][8]) {
  for (int i0 = 0; i0 < 9; i0++) {
    for (int i1 = 0; i1 < 8; i1++) {
      dB[i0][i1] = 0;
    }
  }
  for (int i = 0; i < 8; i++) {
    if (i >= 0 && i < 8 && i < 9 && i + 1 >= 0 && i + 1 < 9) {
      for (int j = 0; j < 8; j++) {
        if (j >= 0 && j < 8) {
          dB[i][j] = dB[i][j] + dA[i][j] * ((float) 1) / ((float) 2);
        }
      }
    }
  }
```

```
  }
  for (int i0 = 0; i0 < 9; i0++) {
    if (i0 - 1 >= 0 && i0 - 1 < 8 && i0 - 1 < 9 && i0 >= 0 && i0 < 9) {
      for (int j = 0; j < 8; j++) {
        if (j >= 0 && j < 8) {
          dB[i0][j] = dB[i0][j] + dA[i0 - 1][j] * ((float) 1) / ((float) 2);
        }
      }
    }
  }
}
```

与使用数学方法得到的求导表达式进行对比，可知生成的求导代码是正确的。

使用到的编译知识

- 词法分析，语法分析，语法树构建，语法树遍历，目标代码生成。这些编译知识在project1中已经使用过；详见project1的报告。
- 语法树变换，SDT。Project2使用的主要编译知识为语法制导翻译。IRMutator的工作原理可以看作后缀SDT，实现可以看作递归下降分析。对当前语法树结点进行mutate操作时，mutator首先访问当前结点的子节点，返回mutate之后的子树。然后，mutator根据设置好的语义动作修改当前结点的值，得到一个新的结点（以该结点为根的子树），并返回。有时，我们也使用visitor模式扫描语法树结点，并更新全局变量。全局变量可以看作当前语法树结点的继承属性，故visitor对应的是L型SDD的递归下降分析。

例如，下面的代码为SubstituteMutator中对Index结点实现的visit函数。我们扫描替换列表，查找当前index是否在替换列表中；如果在，则返回当前index的替换值，否则返回当前的index。

```
Expr visit(Ref<const Index> op) override {
  for (auto t : mp_sub) {
    if (printExpr(op) == t.first) {
      return t.second;
    }
  }
  return op;
}
```