编译技术 Project2 Report

石弼钊 1700012955
滕明桂 1700012974
周劭博 1700013008

Group 27

摘要 Report 按照要求分为五个部分，首先是介绍我们的自动求导编译的算法设计 (Sec.1)，第二部分为具体的函数设计以及实验结果 (Sec.2)，第三部分为例子展示 (Sec.3)，第四部分为总结所使用的编译知识 (Sec.4)，第五部分为成员分工 (Sec.5)。

1 自动求导技术设计

自动求导技术设计的基本思路如下，伪代码表示如 Alg.1所示：

1. 通过正则表达式解析 json 文件中的 kernel 语句，建立符号表，记录每一个变量的名称，类型，大小，index 信息。利用 Project1 中所设计的数据结构，同时增加两个成员属性：need_grad 和 contain_grad 来记录当前的变量或者项是否需要求导和包含求导变量。
2. 调用 inver 函数将中缀表达式转为后缀表达式，便于后续操作。
3. 根据函数的求导规则，加减法的求导只要两个操作数分别求导相加/减即可，而乘除法的求导则需要对于每个因子求导之后相加，所以对于表达式进行合并操作，将所有乘法的操作数合并成一个变量，单独对其求导后记录他的微分形式，例如对于表达式 $A = B * C + D * E$，我们对于操作数进行合并变换之后应该得到 $A\ Var1\ Var2\ +\ =$，其中 $Var1$ 是代表的就是 $B * C$，他的全微分形式记录为 $Var1' = dB * C + B * dC$，此时需要注意，$Var1$ 的全微分形式并不是我们所需要，这个时候就需要利用 need_grad 属性，每次求微分的时候，都需要查看当前 need_grad 是否为 true, 如果不为 true，则说明当前微分的对象和我们的输出不符合，使用 0 来替代，同样对于上面这个例子，假设我们需要对 B 求导，那么在第一次求求微分的时候，need_grad 为 true，而第二次的时候，need_grad 为 false，所以最终得到的求导表达式的形式应该是 $Var1' = dB * C + 0$，对于 $Var2$ 的微分形式同理。

4. 计算得到所有项的微分形式之后，我们利用反向传播的原理得到我们最终需要的导数。在这个过程中，我们需要进行两项操作，对于求导表达式进行变换，使得等式的左边为我们需要求导的变量，第二步是需要完成下标的变化。
   (a) 首先完成求导表达式的变换，利用的方法梯度方向传播的公式，利用链式法则，首先在等式的左右两边除以需要微分的变量，也就是在计算当前的梯度，然后再乘上等式左边的变量的导数，也就是回传的梯度。同样以上述 $A = B*C+D*E$ 为例，如果对 B 求导，通过前面的操作，我们已经可以将等式化为 $dA = dB*C+0+0+0$，利用这个表达式来计算当前的梯度就是 $\frac{\partial A}{\partial B} = C$，然后再乘上回传的梯度也就是 dA, 最终就可以得到我们需要的 B 的导数 $dB = \frac{\partial A}{\partial B} * dA = C * dA$
   (b) 第二步我们需要进行下标的变换，我们需要记录需要求导的变量的下标的情况，如果在下标中存在运算符，则我们将他重新赋值为一个新的 index 变量，然后通过 broadcast 函数去修改其余所有可能需要变化的下标，例如对于表达式 $A[i] = B[i+1]$，通过上述步骤已经得到了求导的表达式 $dB[i+1] = dA[i]$，由于左边的下标不能带有运算符，所以我们需要进行下标替换，令 $i+1 = i'$，则 $i = i'-1$，然后我们用这个 $i'-1$ 去替代其余所有的 i 完成下标变换, 所以最终得到的求导表达式为 $dB[i'] = dA[i'-1]$
5. 在上述变换的过程中，我们都会保持计算式仍然是后缀表达式的形式，上面写成中缀表达式只是为了便于阅读。在符号表和表达式上完成上述变换操作之后，就相当于得到了求导表达式的后缀表达式形式，这个时候我们只需要利用 Project1 中的代码生成算法，确定他的 for 循环范围，然后输出成为 C 代码即可。

2 具体实现和实验结果

2.1 具体实现

相较于 Project1 的代码生成的实现，我们增加的部分是对 kernel 进行求导表达式计算和下标的转换，分别由如下两个函数 ir_extend 和 index_transform 来实现，具体的实现流程如下：

1. **ir_extend** 函数：函数的目标就是计算出表达式的各个项的微分形式。我们操作后缀表达式，从前往后遍历，如果发现 “+/-” 则需要找到加法

Algorithm 1 kernel 语句的求导函数自动生成

输入: kernel

输出: 自动求导函数 C++ 代码

```
function ConstructTree(kernel)
    StringVector ← kernel.split(";")
    for patchkernel : StringVector do
        SuffixVector = inver(patchkernel)//构建后缀表达式
        //操作后缀表达式完成自动求导
        dSuffixVector = ir_extend(SuffixVector)
        //完成求导表达式生成和下标变换
        final_SuffixVector = index_transform(dSuffixVector)
        //利用 Project1 代码，完成从后缀表达式到 C++ 代码的生成
        Stmt stmt = ConstructStmtNode(final_SuffixVector)//生成 IRNode
        //根据表达式的下标生成 checkBound
        IfcheckBound = cond_gen(包含左值单字 index 的复合 index 子集)
        //建立表达式树
        Stmt elseif = IfThenElse::make(IfcheckBound, stmt)
        Stmt main_stmt = LoopNest::make(左值 indexes, elseif)
        MainStatement.push_back(main_stmt)
    end for
    inputs = create_io(kernel) //用于创建函数签名
    //建立完整的抽象语法树
    Group IRtree = Kernel::make(kernel.name, inputs, MainStatement)
    String Code = IRVistor(IRtree)//遍历抽象语法树生成 C++ 代码
    return Code
end function
```

的两个操作数，对他们分别进行求微分的操作，这两个操作数可能是单个变量 A 或者通过包含乘除法的项 $A*B*C$ 或者是一个常数，求导操作就是分别对于这些项目进行求导

(a) 对于单个的变量，如果他是 need_grad 为 true，也就是需要求导的变量，那么其实只需要将其微分形式记录为 “d+ 变量名”，否则赋值为 0。

(b) 对于包含乘除法的项，我们参考复合函数的求导规则，对于每一个变量单独求导然后相加，但是需要注意当前需要求微分的变量是否为我们需要当前需要求导的变量，如果不是，则就将这一项赋为 0，例如对于项 $A*B*C$, 我们需要计算 B 的导数，我们首先应该计算对 A 的微分，但是因为 A 不是我们需要的，所以对 A 的微分就赋值为 0，第二次计算对 B 的微分，这个时候是我们需要的，所以微分形式记录为 $A*dB*C$，第三项对于 C 微分，同对 A 微分一样，赋值为 0。然后我就获得了关于 B 的导数为 $0+A*dB*C+0$。并且这样的方式对于表达式 $A*A$ 计算对 A 的导数同样适用，按照上述的方式，就可以得到导数为 $dA*A+A*dA$

(c) 对于常数的微分就直接赋为 0 即可。

通过上述方式，我们计算 “+/-” 的两个操作数的导数之后，我们只需要用他们的导数表达式去替代原来的操作数，这样就可以获得当前的梯度的表达式。下一步就是需要通过回传的梯度计算出最后需要输出的导数。

2. **index_transform** 函数：这个函数需要完成求导表达式的变换和下标变换，这两个部分的操作：

(a) 首先是求导表达式的变换，根据反向传播的原理，我们只需要使用当前的梯度乘上回传的梯度即可，在代码实现上，我们可以简单的理解为交换左右变量的位置，即在等式左边替换为需要求导的变量的微分，在等式右边，用原来等式左边的变量的微分形式去替换需要求导的变量的微分。例如对于表达式 $A=B*C$，完成上述操作后得到 $dA=dB*C$，在等式左边替换为 dB，然后在等式右边用 dA 替换 dB，得到 $dB=dA*C$，这就是我们需要的求导表达式。

(b) 但是完成上述变量换位之后，由于规定在等式左边的下标中不能存在运算符，所以如果需要求导的变量的下标中存在运算符，则需要进行下标变化。即 $dB[i+1]=dA[i]$ 的形式不允许存在，需要变换成为 $dB[i]=dA[i-1]$，但是下标变换是一个 NP 问题，我们根据样例进行设计完成了一些简单的变换：

i. 对于需要求导的变量的下标中存在//和% 运算时，我们默认这两个操作必须同时存在表达式中，且他们的第二个操作数必须相同，这样才能通过整除和取模的结果逆变换回原来的数。所以我们在存储时会将他们绑定在一起。在完成求导表达式的生成之后，我们首先将 x//num 替换为一个变量 z_1，然后再将 x%num 替换为一个变量 z_2，最后再将 x 替换为 $z_1 * num + z_2$。替换的过程就是遍历整个表达式，然后遇到 x//num 就将其替换为 z_1，其余同理。

ii. 对于下标中有 +/-操作的，也同样将他们设为一个新的 index 变量，然后计算得到新的 index 和原来 index 之间的关系，然后到表达式中去替换掉原来的 index 即可。我们对于两种特殊的情况进行了处理如下

   – 可能会出现 $i+1$,$i+2$,$i+3$ 都出现在要求导的变量的下标中，所以我们选择记录第二个操作数最大的变量，即 i+3 设为 z，然后也是遍历表达式，将 i 替换为 $z-3$ 即可。

   – 如果下标是两个 index 变量的相加，如 $i+j$，这样令他们为 z 之后，我们只需要任意去替换一个变量即可，例如我们就可以用 $z-i$ 去替换所有的下标 j 即可。

2.2 实验结果

测试的十个样例均通过，结果如图2.1所示

```
root@iZ2zeanyt65achi3vuubtdZ:~/compiler-final/build/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.
```

图 2.1. 测试结果

3 具体例子演示

具体例子演示以 $B<32>[i]=A<2,16>[i//16,i\%16]$ 为例:

1. 首先解析表达式，记录数组大小和下标等信息，然后转换为后缀表达式 $BA=$
2. 使用 ir_extend 对于等式对于各个项进行求全微分，得到微分表达式 $dBdA=$
3. 使用 index_transform 完成求导表达式的生成，首先是对于微分形式进行变换，得到我们需要的变量的导数计算式 $dAdB=$，然后对于下标进行变化，首先是将所有的 $i//16$ 替换为 z_1, 然后将 $i\%16$ 替换为 z_2，第三轮替换是将原来的 i 替换为 z_1*16+z_2，通过三次调用 broadcast 函数就完成了下标的替换，得到最终需要的求导表达式的后缀形式为[1]：

$$dA[z_1,z_2]dB[z_1*16+z_2]=$$

4. 利用 Project1 的算法进行分析，确定 for 循环的范围和 bound-check，然后生成的语法树大致结构如图3.1所示

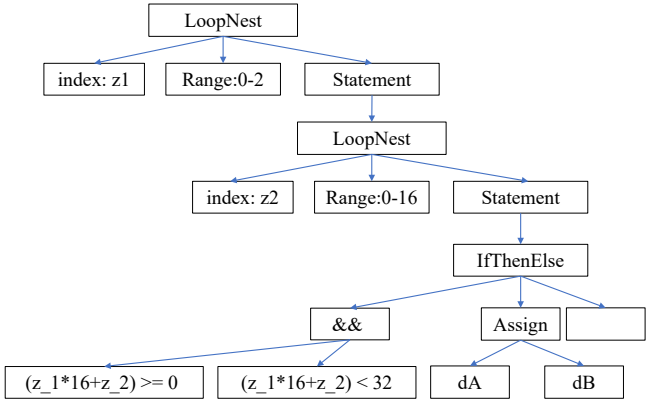

图 3.1. $B<32>[i]=A<2,16>[i//16,i\%16]$ 的求导表达式的语法树，由于页面原因，所以对于语法树中的部分进行了合并和删除，只是为了展示语法树的基本架构

5. 遍历语法树输出成为 C 代码，输出代码中的 tmp_1 是因为存在 bound-check，需要判断等式右边的各个数组的 index 是否越界，如果越界则 tmp_1 为 0，结果如图3.2所示。

[1] 在我们的栈中，并没有存储下标信息, 他们都是变量的一个属性值，便于阅读将其写出

```
#include "../run2.h"
void grad_case8(float (&dB)[32], float (&dA)[2][16]) {
  for(int z_1 = 0; z_1 < 2; ++z_1){
    for(int z_2 = 0; z_2 < 16; ++z_2){
      float tmp_1 = 0;
      if (((z_1*16+z_2) >= 0 && (z_1*16+z_2) < 32)) {
        tmp_1 = (tmp_1 + ((float) 0  + dB[(z_1*16+z_2)]));
      } else {
      }
      dA[z_1][z_2] = tmp_1;
    }
  }
}
```

图 3.2. $B<32>[i] = A<2,16>[i//16, i\%16]$ 生成的求导函数

4 总结

在自动求导的编译器实现过程中，我们所用到的编译知识如下：

1. 词法分析和语法分析：借助于 segrex 库通过正则表达式，从 json 的 kernel 中提取 token，并建立符号表。并将 kernel 中的中缀表达式转换为后缀表达式。
2. 中间表示形式：自定义中间表示形式，将表达式中的变量进行整合，将表达式根据 $+/-$ 分割表达式，将各个部分合并为一个自定义的项，使得不同项之间仅有加减法操作，可以直接进行微分的加减法。后续只需要在这些节点上进行操作。
3. SDT: 设计一个 SDT 用于计算每一个项的微分形式。由于每个项要么就是单独的变量或常量，要么只有乘除法，所以我们对于项中的每一个变量求微分，然后加和得到这个项的全微分形式。设计另外一个 SDT 完成求导表达式的生成和下标变换。
4. 语法树构建：利用计算完成的求导表达式的后缀表达式进行建树操作。
5. 语法树遍历和代码生成：遍历语法树，在相应的结点输出对应的代码，完成代码生成。

5 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。