Compile Project 2 Report

编译大作业 2 报告

小组成员: 陈野 1700012834

余卓 1700012839

周思源 1700012768

目录

一、　小组分工 1

二、自动求导技术设计 1

三、具体实现流程 2

四、实验结果展示 3

五、所用知识总结 4

一、小组分工

余卓：求导表达式生成

陈野：下标变换

周思源：报告撰写

二、自动求导技术设计

1　矩阵里的每一个变量单独来看就是其他的变量经过运算然后累加求得。所以，对矩阵求导数，相当于对矩阵里的每一个变量求导数。

2　我们以最简单的变量表达式 A=B*C+D 为基础，直观可以看出，如果要对 B 求导，导数只跟 B 的乘因子有关，也可以看作是 B 的系数。基于这个简单的发现，我们可以简单设计对于复杂表达式的求导技术。我们首先找到需要求导的变量的位置，在得到变量的位置的基础上，往左往右扩展，如果碰到的符号是*/，说明下一个位置的变量为我们要求导的变量的系数，则将左端点左移到下一个符号之前；如果碰到的符号是+-号，说明下一个位置的变量以及之后的变量在求导的时候都会变成 0，则可以停止扩展；在扩展的过程中，如果碰到括号，则需要把括号里的变量当做一个整体，这部分用一个计数变量来实现括号匹配，里面的+-*/均需要保留。

3　需要求导的变量在表达式中可以会出现多次。根据求导规则，我们可以对变量的每一个位置单独对表达式求导，最后累加。比如 A = B*B，B 为我们要求导的变量。首先对第一个位置的 B 求导，得到 dB = dA * B，再对第二个位

置的 B 求导，得到 dB = dA * B， 累加即为 dB = dA * B + dA * B，与直接对 A=B*B 中的 B 变量求导结果一致。

4 一个变量在矩阵中有相应的位置，并且我们需要求得系数在其他矩阵当中的位置，即下标变换。求导后式子所表示的下标，与原式子中的该位置的数如何计算的出有关。根据爱因斯坦求和公式，我们可以得出结论：导数的下标与求导前保持一致即可，而未求导的变量则保持不变即可。但由于原式子中在右值中可能在矩阵下标中出现运算符号，如果对此矩阵求导，则导数会在左值中出现，因此需要进行变量替换。我们选择的方案与给出的样例不同。将左值中出现的运算符号整体变为一个新的下标，并将右值中对应的下标变为相反的运算操作，如左值中出现了 p+q 下标，整体替换为 z 后，右值中所有的 p 都会变为 z-q，其他运算同理。这样就能保证左值中不出现运算符号。但是这样会使右值中出现减号，可能造成越界的情况，因此需要修改 Project1 中的代码生成部分，在具体实现流程中解释。

三、具体实现流程

1 解析 json 文件。同 Project1，我们根据根据字符串匹配，读取出 case 当中我们需要的信息，比如表达式，需要求导的变量的名字。进一步解析表达式，通过语法规则，将 kernel 分成 LHS 和 RHS 两部分，将变量的名字和下标分别存储。

2 计算导数。使用自动求导技术，计算我们需要求导的变量的系数表达式，假设为 K，则表达式可以简单表示成 dA=dB*K。

3 计算下标变换。通过 json 文件中得到每个变量的下标和维度信息，如果在右

值中出现了需要求导的变量，则判断下该变量中的下标是否出现了运算符，如果有则根据上一节所讲记录每个需要求导变量对应的左值的导数的变换后的下标。之后顺序扫描 2 中生成的求导后表达式，为每个变量加上替换后的下标和维度。

4 根据 3 中得到的新的 kernel 输出一个新的求导后表达式的 json 文件。

5 修改 project1 中的代码生成文件，添加对于下标>=0 的判断，以及对于出现在左值中的下标，求导上界改为根据左值确定。

6 修改 CMakelist.txt，顺序执行生成 json 文件代码和生成 c 文件代码，利用生成的 json 文件得到最终的求导 c 代码。

四、实验结果展示

我们用下面的自行构造的样例进行展示：

```
{
    "name": "grad_case1",
    "ins": ["A"],
    "outs": ["B"],
    "data_type": "float",
    "kernel": "B<4, 16>[i, j] = A<4, 16>[i, j] * A<4, 16>[i, j]
* A<4, 16>[i, j] + 1.0;",
    "grad_to": ["A"]
}
```

我们会先对这个 json 进行解析，用第二节中提到的方法进行，把 kernel 中每一个需要求导的变量分别求导，得到新的求导后表达式，再进行下表变换，得到新的 kernel，如下：

```
dealing with case 11
kernel : B<4, 16>[i, j] = A<4, 16>[i, j] * A<4, 16>[i, j] * A<4, 16>[i, j] + 1.0;
grad_expr:B=A*A*A+1.0
grad_to_var:A grad_result:dB*A*A+A*dB*A+A*A*dB
kernel : B<4, 16>[i, j] = A<4, 16>[i, j] * A<4, 16>[i, j] * A<4, 16>[i, j] + 1.0;
```

之后生成新的 json 文件，如下：

```
{
    "name": "grad_case11",
    "ins": ["A", "dB"],
    "outs": ["dA"],
    "data_type": "float",
    "kernel": "dA<4, 16>[i,  j] = dB<4, 16>[i,  j] *A<4, 16>[i, j] *A<4, 16>[i,  j] +A<4, 16>[i,  j] *dB<4, 16>[i,  j] *A<4, 16>[i,  j] +A<4, 16>[i,  j] *A<4, 16>[i,  j] *dB<4, 16>[i,  j] ;",
}
```

在使用修改过后的代码生成文件，生成最终的 c 代码，如下(只截取了部分代码)：

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

void grad_case11(float (&A)[4][16],float (&dB)[4][16],float (&dA)[4][16])
{
  float tmp0[4][16];
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 16; ++j) {
      tmp0[i][j] = 0;
      if (0 <= j) {
        if (j < 16) {
          if (0 <= i) {
            if (i < 4) {
              if (0 <= j) {
                if (j < 16) {
                  if (0 <= i) {
                    if (i < 4) {
                      if (0 <= j) {
                        if (j < 16) {
                          if (0 <= i) {
                            if (i < 4) {
                              tmp0[i][j] = (tmp0[i][j] + ((dB[i][j] * A[i][j]) * A[i][j]));
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
      if (0 <= j) {
        if (j < 16) {
          if (0 <= i) {
            if (i < 4) {
```

这样就得到了最终的自动求导代码。

五、所用知识总结

1 在解析 json 文件中，我们运用到了词法分析和语法分析，根据提供的表达式规则，运用语法分析将表达式分成不同的终结符号，再运用词法分析，通过对运算符，变量的匹配，得到我们需要的信息。

2 在表达式转换的过程中，我们使用了表达式树的数据结构，将表达式转换成

语法树，表示不同运算符在表达式中的实际优先级。(括号里运算符的优先级>括号外的优先级)。

3 在 IR 树输出 c 代码过程中，我们运用到了语法制导翻译。在分析 IR 树的过程中，执行相应的操作，输出 c 代码。