# 编译技术project2报告

## 小组编号：7

## 小组成员

郭宇琪 1700012785

顾怿洋 1700012788

周博文 1700013028

袁皓晨 1700012959

## 求导原理描述

在本部分，我们使用以下这个简单的矩阵乘法运算例子，解释我们的导数计算方法。

> A<32, 16>[i, j] = C<32, 16>[i, k] * B<32, 16>[k, j] + 1.0

根据我们在project1中的代码实现，我们为这样的一个运算表达式生成了如下的计算代码。

```
void kernel_example(float (&B)[32][16], float (&C)[32][16], float (&A)[32][16]) {
  float temp1[32][16];
  for (int i = 0;i < 32;++i){
    for (int j = 0;j < 16;++j){
      temp1[i][j] = 0;
      for (int k = 0;k < 16;++k){
        if (0 <= i && i < 32) {
          if (0 <= k && k < 16) {
            if (0 <= k && k < 32) {
              if (0 <= j && j < 16) {
                temp1[i][j] += C[i][k] * B[k][j];
              }
            }
          }
        }
      }
      temp1[i][j] += 1;
    }
  }
  for (int i = 0;i < 32;++i){
    for (int j = 0;j < 16;++j){
      A[i][j] = temp1[i][j];
    }
  }
}
```

容易看出，我们为每一条计算表达式生成了三个语句块。

第一个语句块是临时数组的声明。

```
float temp1[32][16];
```

第二个语句块是一系列的循环，按照计算表达式中每一个项(item)出现的次序，进行计算，并将结果累加到临时数组中。

```
for (int i = 0;i < 32;++i){
    for (int j = 0;j < 16;++j){
      temp1[i][j] = 0;

      //item-1
      for (int k = 0;k < 16;++k){
        if (0 <= i && i < 32) {
          if (0 <= k && k < 16) {
            if (0 <= k && k < 32) {
              if (0 <= j && j < 16) {
                temp1[i][j] += C[i][k] * B[k][j];
              }
            }
          }
        }
      }

      //item-2
      temp1[i][j] += 1;
    }
  }
```

由于要求支持爱因斯坦求和约定，因此在每一项的计算过程中，可能出现更多的内层循环。在这个例子中，item-1的计算过程中就出现了关于k的内层循环。为了叙述方便，我们在此约定，将内层循环的循环体称为子项(subitem)。在本例中，item-1的子项为

> temp1[i][j] += C[i][k] * B[k][j]

第三个语句块也是一系列的循环。这部分则将临时数组中的结果复制到输出数组中，完成计算结果的输出。

```
for (int i = 0;i < 32;++i){
    for (int j = 0;j < 16;++j){
      A[i][j] = temp1[i][j];
    }
  }
```

基于以上的分析，我们可以给出我们在计算过程时遵循的数学表达式：

> $A = \sum_{i,j}(item1 + item2) = \sum_{i,j}(\sum_k subitem1 + item2)$

现在，假设我们需要求对B矩阵的导数。由链式求导法则可知

$$\frac{\partial Loss}{\partial B} = \sum_{m,n}(\frac{\partial Loss}{\partial A(m,n)} \times \frac{\partial A(m,n)}{\partial B})$$

其中

$$\frac{\partial Loss}{\partial A(m,n)} = dA(m,n)$$

已经给出，因此我们只需求

$$\frac{\partial A(m,n)}{\partial B}$$

将前面给出的A的计算表达式代入这个式子，得出

$$\frac{\partial A(m,n)}{\partial B} = \frac{\partial(\sum_{i,j}(\sum_k subitem1 + item2))}{\partial B}$$

化简可得

$$\frac{\partial A(m,n)}{\partial B} = \sum_{i,j}(\sum_k \frac{\partial subitem1}{\partial B} + \frac{\partial item2}{\partial B})$$

对比A的计算表达式，可以发现，A对B矩阵的导数表达式和A矩阵的计算表达式具有相似的结构，这意味着两者的代码结构也应高度相似，我们可以充分利用原有代码来构造求导计算代码。经过仔细分析，我们发现，只需将原来代码等式右边的部分对B求导，即可在不对代码结构做大幅度修改的情况下，计算出A对B的矩阵导数。

在将等式右边部分对B求导后，我们得到了如下的代码结构。其中index1和index2是B矩阵的下标，这段代码计算的是A矩阵对B[index1,index2]的导数。

```
for (int i = 0;i < 4;++i){
  for (int j = 0;j < 16;++j){
    temp1[i][j] = 0;

      //item-1
      if (0 <= i && i < 4) {
        if (0 <= j && j < 16) {
          if (0 <= i && i < 4) {
            if (0 <= j && j < 16) {
                temp1[i][j] += (( index1 == i && index2 == j ) ? ( 1 ) : ( 0 )) *
B[i][j];
            }
          }
        }
      }

      //item-2
      temp1[i][j] += 0;
    }
  }
```

随后，将这一结果与dA矩阵相乘累加，即可完成dB[index1,index2]的计算。最后，在最外层增加关于index1和index2的循环，即可完成整个dB矩阵的计算。完整版本的代码如下。

```
void grad_example(float (&C)[32][16], float (&dA)[32][16], float (&dB)[32][16]) {
  for (int index1 = 0;index1 < 32;++index1){
    for (int index2 = 0;index2 < 16;++index2){
      float temp1[32][16];
      dB[index1][index2] = 0.0;
      for (int i = 0;i < 32;++i){
        for (int j = 0;j < 16;++j){
          temp1[i][j] = 0;
          for (int k = 0;k < 16;++k){
            if (0 <= i && i < 32) {
              if (0 <= k && k < 16) {
                if (0 <= k && k < 32) {
                  if (0 <= j && j < 16) {
                    temp1[i][j] += C[i][k] * (( index1 == k && index2 == j ) ? ( 1
) : ( 0 ));
                  }
                }
              }
            }
          }
          temp1[i][j] += 0;
        }
      }
      for (int i = 0;i < 32;++i){
        for (int j = 0;j < 16;++j){
          dB[index1][index2] += dA[i][j] * temp1[i][j];
        }
      }
    }
  }
}
```

# 实现流程

由于我们的project2内容很大一部分是基于project1，所以实现流程中很多与project1相同。主要的工作在设计思路中的第三部分。

### 设计思路：

使用lex进行词法分析，yacc进行文法分析，在规则末段添加动作以建立原始抽象语法树【第一部分】，再将原始抽象语法树转换成c抽象语法树【第二部分】，然后将c抽象语法树进行转换实现求导【第三部分】，最后打印c抽象语法树【第四部分】。

### 实现方法：

第一部分·建立原始抽象语法树

这一部分完成的主要工作是利用lex和yacc工具完成词法和文法的分析，并通过编写语义动作，完成原始抽象语法树的建立。具体实现为，首先根据提供的文法，在yacc文件夹下编写lex能够识别的词法文件first.l，以及yacc能够识别的文法文件second.y。随后，为每一个终结符号和每一条文法生成式编写语义动作，每一种语义动作

被封装成一个函数，终结符号的语义动作位于tokens.cc下，文法生成式的语义动作位于actions.cc下。最后，在second.y中对应文法生成式的末端填写所编写的函数，并使用lex和yacc工具对first.l和second.y进行编译，生成c语言文件y.lex.c和y.tab.c。在这些工作完成之后，main函数即可调用生成的的yyparse()函数对输入字符串进行解析，并返回构造出的抽象语法树。

为了给后续步骤提供方便，在构造原始抽象语法树的阶段，还实现了三个辅助功能。一个辅助功能是变元范围的推定。在自底向上的归约过程中，对于所有下标表达式中出现的变元，其取值范围会在适当的时机进行判定，并推送到全局变量global_map中。另一个辅助功能是对下标表达式中出现变元种类的收集。我们为所有类型的节点增加了一个属性variables，这个属性存储了以该节点为根的子表达式中所有在下标表达式中出现过的变元。最后一个辅助功能是数组变量shape信息的记录。对于扫描过程中遇到的每一种数组变量(tensor或scalar)，我们会将其shape信息推送到全局变量global_shape_map中。这些功能为第二部分树结构转换提供了必要的信息。

第二部分·转成c抽象语法树

这一部分主要通过继承自IRVisitor的类IRTermFinder实现。首先将最顶层的RHS以+-号为界分割成单独循环的项，用visitor构建每一个项对应的循环、条件和赋值语句，它们的指针都存在类term_group这个数据结构中。这时候各个语句都还是分组并列存储的。然后将每个项的这些语句串起来，以构建c抽象语法树在outmost LoopNest stmt层次上的子树，对应于单条原始赋值语句。最后将这些子树的root顺序存入kernel结点的stmt_list vector，以形成最终的kernel层次上的c抽象语法树。这一部分的主要类型声明在include/IRTermFinder.h文件中，实现在solution/IRTermFinder.cc文件中。

第三部分·转换c抽象语法树实现求导

这一部分主要通过继承自IRMutator的类IRMutator_grad实现。由于在生成函数签名的时候只能出现使用过的矩阵，所以求导时需要记录使用过的矩阵。为此我们在IRMutator_grad中设置了一些全局变量，用于记录当前正在对哪个矩阵求导、求导过程用到了哪些矩阵等信息。我们为ExprNode结点添加了is_zero属性，如果某个结点求导为0，那么就不需要将其内部细节翻译出来，只需要将其转化为一个int立即数结点0。此外，我们还为ExprNode结点添加了一个元素为string的set，名为var_used，用于记录当前结点求导过程中用到的变量名。下面是具体求导过程。

首先是对于表达式的求导。对于立即数，求导得0。对于Unary结点，对其操作数求导，并加上Unary的操作符。对于Binary结点，为它的两个操作数求导，分加减乘除讨论，计算最后的返回结点。如果计算得到0，那么返回一个立即数结点0。在乘除法求导中如果计算结果不止一项，那么得到的计算式结点外部应该套上一层括号，即返回操作符为Bracket的Unary结点。除法求导中还有返回操作符为Neg的Unary结点的情况，具体见代码。在求导过程中同时会维护结点的var_used集合。对于Var结点，如果name与当前求导的矩阵名不同，那么直接返回立即数结点0；如果相同则返回一个Select结点，判断条件为下标是否对应相等，相等得1，不相等得0。

下面是对于Move结点的求导。从前面的求导原理中可以看出，Move结点的求导有两种，一种是对第三个语句块（将临时数组中的结果复制到输出数组中）中的Move结点进行求导，另一种是对第一（临时数组初始化）、二（临时数组计算）个语句块中的Move结点进行求导。后者的操作很简单，只需要对Move结点的src进行求导，作为新的Move结点的src即可。前者的操作则是关键。根据前面的求导原理，此处应该对Move结点进行转换，结点的原src为临时数组temp，原dst为json文件outs中的数组，结点的新dst应该为一个新的数组：当前求导的矩阵名前面加上'd'，其维度和原矩阵一致，而结点的新src应该为对outs数组的求导与临时数组temp的乘积。

对于Kernel结点的转换，更新了函数签名，然后为每个待求导的矩阵生成新的语句块，每一个语句块中要把原来的temp矩阵更改名称和维度，增加对待求导矩阵的声明，并为其增加外层循环。

第四部分·打印c抽象语法树

这一部分主要通过继承自IRPrinter的类IRPrinter_genCcode实现，主要操作是重载IRPrinter中对一些结点的visit函数。由于变量声明只出现在参数表中和临时变量声明时，所以对于四种类型的立即数直接输出它们的值，在Kernel部分增加了参数类型的输出；Unary部分增加了括号项，Binary部分增加了整除；Var结点分别根据是张量还是常量进行不同的输出；对于IfThenElse结点，由于我们得到的c语法树结构中不会出现else部分，所以将其去掉；对于LoopNest结点，我们调整了index结点和Dom结点的输出方式；对于Move结点，根据我们自己定义的属性Move操作类型MoveOp进行不同的输出，如果操作是Declare（声明），输出变量类型和名称，如果操作是Zero（初始化，置0），根据之前的处理，只需直接进行赋值，其他几种情况（=、+=、-=）分别输出对应的操作符；对于Select结点，将其翻译为问号表达式。这一部分的主要类型声明在include/IRPrinter_genCcode.h文件中，实现在solution/IRPrinter_genCcode.cc文件中。

# 组内分工

郭宇琪负责Kernel结点的转换，顾怿洋负责Move结点的转换，周博文和袁皓晨负责表达式部分的求导转换。

# 实验结果

```
PS D:\CompilerProject\build\project2> .\test2.exe
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.
```