编译技术 Project 2 报告

贺义鸣　沈若冰　姜欣睿　徐逸飞

一、小组分工

贺义鸣、沈若冰负责 Project 1；姜欣睿、徐逸飞负责 Project 2。在 Project 2 中，贺义鸣、沈若冰提供此前的代码，姜欣睿主要负责分析反向传播表达式，完成带限制的线性坐标变换部分的代码；徐逸飞主要负责为表达式分析构建简化版的语法树，并完成导数表达式的翻译代码。大量设计、debug 和优化方面的讨论由两人共同完成。

二、自动求导技术设计

本次 Project 需要根据给定表达式给出计算导数的 C 语言原代码。由于 Project 1 的工作能够分析一般表达式，并给出符合 C 语言结构的语法树，并翻译成 C 代码，因此一部分工作可以基于此上修改完成。

1. 构建适合求导分析的 AST

Project1 中构建的语法树过于复杂，包含大量循环、判断和赋值语句结点，不适合进行求导分析。因此需要对此前的代码进行修改，对等式左侧的 Var（默认唯一）直接解析并连接在 Move 左侧，对 Move 右侧的 Expr 根据运算符优先级生成子树，连接在 Move 右侧，此过程中对 Var 的 Index 和 Dom 子孙结点类似原 Parser 进行保留。随后构建无分支的 Move-LoopNest-Kernel 树根，完成 AST。

2. 分析该 AST，得到反向传播表达式

得到易于处理的 AST 之后，我们就可以开始着手设计 IRMutator 的派生类了。我们的思路稍有些不同，不是通过传入一个 kernel 返回一个 kernel 然后直接输出，而是传入一个 kernel 去 visit，然后得到一个 statement adjointStmt，这个是我们需要的表达式。这之后我们通过这个表达式还原出 ins, outs, gradto 等变量，最后通过 project1 里面的 parser 去得到我们所需要的代码。

这里面最核心的就是 adjointExpr，表示访问到某个节点时积累的梯度的表达式。他们相对于每个 visit 而言都是全局变量。当我们访问一个节点的时候，我们先保存，然后更新

adjointExpr 的内容，递归 visit 每个子部分，最后将 adjointExpr 的内容还原。当 adjointExpr 到达了需要反向传播的变量时，我们通过一些处理，将它归入到已有的 adjointStmt 里面来。所以我们的处理方法是基于回溯的。

对于反向传播的梯度表达式更新，我们只关注两种节点：Binary 和 Var。

Visit Binary：当我们 visit Binary 节点是，我们需要根据操作类型对 adjointExpr 进行修改。比如，对于表达式$B[i,j] = A[i,j] * C[i,j]$，此时 adjointExpr 为$dB[i,j]$，Binary 的操作类型是 BinaryOpType::Mul，而我们马上要 visit A 节点。这时我们就把剩余部分乘到 adjointExpr 里面来，那么 adjointExpr 就变成了$dB[i,j] * C[i,j]$。注意，此时我们没有对 gradTo 的节点作任何假设，不管需要 gradTo 的是 A 还是 C，这一部分的处理不受到影响，只会在访问到 var 的时候才会处理。得到新的 adjointExpr，我们就访问 A 节点，更新 adjointStmt，然后返回到当前的 Binary 节点。这时我们会还原 adjointExpr，变成$dB[i,j]$，然后准备访问 C 节点，此时 adjointExpr 又变成了$A[i,j] * dB[i,j]$，然后访问 C 节点，更新 adjointStmt。

Visit Var：当梯度到达了 var 节点，我们首先看这个 var 是否是 gradto 的节点；如果不是，我们就不做操作，直接返回。否则，我们就需要更新 adjointStmt。如果此时 adjointStmt 为空，说明这是第一次访问 gradto 节点，那么我们将 adjointStmt 设置成 var=adjointExpr 的形式，并记录下当前的 var 和 adjointExpr，然后返回；否则，我们通过一些处理，将当前的 adjointExpr 和原有的 adjointExpr 相加，一起作为 adjointStmt 的右部，从而实现梯度在单个变量上的累积。

3. 下标变换

此部分与遍历 AST 得到表达式同时完成。我们假定//和%总是成对出现的，且除数和模数大小相等。我们不对他们的先后顺序作假设，也不对他们中间是否有其他变量作假设。这些情况我们都可以处理。下标变换有两个部分，一个是"grad var"的 index 存在表达式的情况；一个是合并不同的 adjointExpr 的情况。

1) "grad var"的 index 存在表达式

核心思想：新建一个 index，将(原 index -> 反向表达式) 加入到 rules 中。最后 visit adjointExpr，通过 rules 里面的转换规则，对 adjointExpr 里面所有的下标进行遍历和转换。

举例：A[n, k, p, q] = B[k, c, p + r, q + s] * C[n, c, r, s] .当我们访问 B 节点是，adjointExpr 存放的内容是 dA[n,k,p,q]*C[n,c,r,s]，通过访问 B 节点，我们新建两个 index：h, w，并建立规则(p -> h-r)，(q -> w-s)，然后我们通过这个规则修改 adjointExpr，变成了 dA[n,k,h-r,w-s]*C[n,c,r,s]，然后建立表达式 dB[k,c,h,w]= dA[n,k,h-r,w-s]*C[n,c,r,s]

2) 合并不同的 adjointExpr

核心思想：以第一个 adjointStmt 作为 index 的标准，对于后面的每一个 var 和 adjointExpr，建立(index->标准 index)的 rules，通过 rules 去 visit 并修改 adjointExpr。最后通过加法将新的 adjointExpr 加到原来的 adjointStmt 的右边去。

举例：A[i,j]=(B[i,j] + B[i+1, j])/2，当访问第二个 B 节点时，已有的 adjointStmt 为 dB[i,j]=dA[i,j]/2, 在经过了下标表达式的消除后，现在的 var 是 dB[k,j],adjointExpr 是 dA[k-1,j]/2，我们建立规则(k->i), (j->j)，然后对 adjointExpr 进行修改，最后加到原有的 adjointStmt

上，即有 dB[i, j]= dA[i,j]/2+ dA[i-1,j]/2

3) 局限性

a) 无法处理 tensorA / tensorB 的对 B 反向传播的表达式
b) 无法处理多对//和%在下标中出现的情况
c) 无法处理下标中有多个运算的情况(e.g. A[a+b+c][d][e])

4. 将表达式翻译成 C 代码

对于新表达式，首先根据各个变量下标的 Dom，将带有下标域字符串添加在变量标识符后，打印成 String，作为新 kernel 的输入。然后将这条 kernel 所用到的变量名按照函数声明的规则排序，将 grad_to 最后一个变量名作为新 outs，其他依次加入 ins 集合中，以保证新的输入可以兼容 Project 1 中的翻译方案。然后对 Project 1 翻译方案稍作修改，删除 if 语句的构建过程，在每个变量取值时添加以下标在相应域内作为条件的 select 语句翻译，若不满足则为 0。直接打印出 C 代码。

三、实现流程

1. **Json_Parser**（json_parser.h）：同 Project 1，解析 json 输入文件。
2. **DParser**（DParser.h）：表达式 AST 生成器，简化版的 Parser，根据输入 Json 信息构建简单 AST。
3. **gradBuilder**（gradBuilder.h）：反向传播分析器，IRMutator 的派生类，visit 上述 AST，生成导数表达式 adjointStmt。
4. **formChanger**（gradBuilder.h）：gradBuilder 的成员，辅助改变 adjointExpr 里的下标。
5. **varMutator**（gradBuilder.h）：反向传播分析器，IRMutator 的派生类，遍历 adjointStmt，生成可直接交给 Parser 翻译的 kernel 和用于修改 J.ins/outs 的变量名表。
6. 对变量名排序，修改 J.ins/outs。
7. **Parser**（Parser.h）：C 代码 AST 生成器，相比 Project 1 稍加修改的 Parser，输出带有 Select 下标边界限制结点的 C 代码 kernel。
8. **IRPrinter**（IRPrinter.h）：C 代码翻译器，相比 Project 1 稍加修改的 IRPrinter，打印上述 kernel，将 Select 结点翻译为(? :)语句。

四、实验结果内容

1. varMutator 生成的 kernel、ins、outs

case4 原则上有两个 outs，但是考虑到 Parser 接收格式问题将其移入 ins，不影响计算

结果。可见所有 case 的导数表达式 kernel 均正确：

```
grad_case6:
dB<2, 16, 7, 7>[n, c, a, b]=(dA<2, 8, 5, 5>[n, k, (a - r), (b - s)] * C<8, 16, 3, 3>[k, c, r, s]);
ins: C dA
outs: dB
grad_case7:
dA<32, 16>[j, i]=dB<16, 32>[i, j];
ins: dB
outs: dA
grad_case8:
dA<2, 16>[a, b]=dB<32>[((a * 16) + b)];
ins: dB
outs: dA
grad_case4:
dB<16, 32>[i, k]=(dA<16, 32>[i, j] * C<32, 32>[k, j]);
dC<32, 32>[k, j]=(B<16, 32>[i, k] * dA<16, 32>[i, j]);
ins: B C dA dB
outs: dC
grad_case3:
dA<4, 16>[i, k]=(dC<4, 16>[i, j] * B<16, 16>[k, j]);
ins: B dC
outs: dA
grad_case5:
dB<16, 32, 4>[i, k, l]=((dA<16, 32>[i, j] * D<4, 32>[l, j]) * C<32, 32>[k, j]);
ins: C D dA
outs: dB
grad_case9:
dA<4>[i]=dB<4, 6>[i, j];
ins: dB
outs: dA
grad_case2:
dA<4, 16>[i, j]=((dB<4, 16>[i, j] * A<4, 16>[i, j]) + (A<4, 16>[i, j] * dB<4, 16>[i, j]));
ins: A dB
outs: dA
grad_case10:
dB<10, 8>[i, j]=(((dA<8, 8>[i, j] / 3) + (dA<8, 8>[(i - 1), j] / 3)) + (dA<8, 8>[(i - 2), j] / 3));
ins: dA
outs: dB
grad_case1:
dA<4, 16>[i, j]=(dC<4, 16>[i, j] * B<4, 16>[i, j]);
ins: B dC
outs: dA
```

2. 最终生成的代码

以 case4 为例，所有 case 都可以正确求导：

```
#include "../run2.h"
void grad_case4(float (&B)[16][32], float (&C)[32][32], float (&dA)[16][32], float (&dB)[16][32],float (&dC)[32][32]) {
  float tmp[32][32];
  float ret[32][32];
  for (int i=0;i<16;i++){
    for (int k=0;k<32;k++){
      ret[i][k]=0;
      tmp[i][k]=0;
      for (int j=0;j<32;j++){
        tmp[i][k]=(tmp[i][k] + ((j < 32? (j >= 0? (i < 16? (i >= 0? dA[i][j]: 0): 0): 0): 0) * (j < 32? (j >= 0? (k < 32? (k >= 0? C[k][j]: 0): 0): 0): 0)));
      }
      ret[i][k]=(ret[i][k] + tmp[i][k]);
      dB[i][k]=ret[i][k];
    }
  }
  for (int k=0;k<32;k++){
    for (int j=0;j<32;j++){
      ret[k][j]=0;
      tmp[k][j]=0;
      for (int i=0;i<16;i++){
        tmp[k][j]=(tmp[k][j] + ((k < 32? (k >= 0? (i < 16? (i >= 0? B[i][k]: 0): 0): 0): 0) * (j < 32? (j >= 0? (i < 16? (i >= 0? dA[i][j]: 0): 0): 0): 0)));
      }
      ret[k][j]=(ret[k][j] + tmp[k][j]);
      dC[k][j]=ret[k][j];
    }
  }
}

```

3. 测试成绩

所有测试样例全部通过：

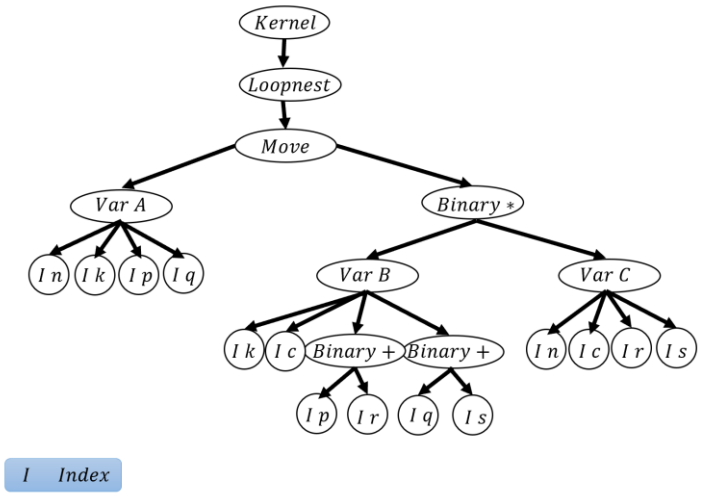
```
xuyf@LabServer1:~/compiler/build/project2$ ./test2
Random distribution ready
Case 1 Success!
Case 2 Success!
Case 3 Success!
Case 4 Success!
Case 5 Success!
Case 6 Success!
Case 7 Success!
Case 8 Success!
Case 9 Success!
Case 10 Success!
Totally pass 10 out of 10 cases.
Score is 15.
```

五、具体例子

在设计部分已用多个例子阐述自动求导中间关键过程，此处仅简单展示以下 kernel grad to B 的变换过程（部分深层结点、属性省略或简略表示），以解释该设计的可行性和正确性。

$$\boldsymbol{A[n,k,p,q] = B[k,c,p + r,q + s] * C[n,c,r,s]}$$

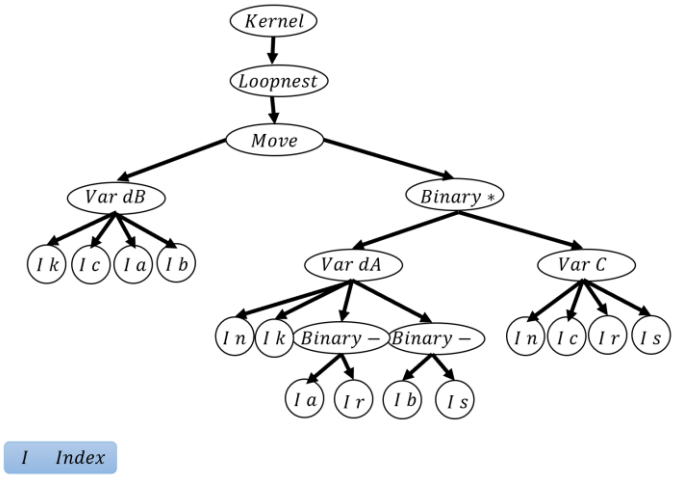
其经过 DParser 构建的 AST 如下：



根据本报告自动求导技术设计部分，经过 gradBuilder 的遍历，分析 AST，进行下标变换，得到的反向传播表达式为（a、b 为新建下标，此时尚未得到下标域）：

$$dB[n,c,a,b] = (dA[n,k,(a - r),(b - s)] * C[k,c,r,s]);$$

其 AST 如下：



经过 varMutator 二次遍历，插入下标域，重构 ins/outs 后得到如下新 kernel：

dB<2, 16, 7, 7>[n, c, a, b]=(dA<2, 8, 5, 5>[n, k, (a - r), (b - s)] * C<8, 16, 3, 3>[k, c, r, s]);

ins: C dA

outs: dB

利用 Parser 和 IRPrinter 翻译，得到以下可以正确求导的 C 代码（自动生成代码单行过长，已手动换行）。

```
#include "../run2.h"
void grad_case6(float (&C)[8][16][3][3], float (&dA)[2][8][5][5],float (&dB)[2][16][7][7]) {
  float tmp[2][16][7][7];
  float ret[2][16][7][7];
  for (int n=0;n<2;n++){
    for (int c=0;c<16;c++){
      for (int a=0;a<7;a++){
        for (int b=0;b<7;b++){
          ret[n][c][a][b]=0;
          tmp[n][c][a][b]=0;
          for (int k=0;k<8;k++){
            for (int r=0;r<3;r++){
              for (int s=0;s<3;s++){
                tmp[n][c][a][b]=(tmp[n][c][a][b] +
                  (((b - s) < 5? ((b - s) >= 0?
                  ((a - r) < 5? ((a - r) >= 0?
                   (k < 8? (k >= 0?
                    (n < 2? (n >= 0?
                     dA[n][k][(a - r)][(b - s)]: 0): 0): 0): 0): 0): 0): 0): 0)
                 * (s < 3? (s >= 0?
                  (r < 3? (r >= 0?
                   (c < 16? (c >= 0?
                    (k < 8? (k >= 0?
                     C[k][c][r][s]: 0): 0): 0): 0): 0): 0): 0): 0)));
              }
            }
          }
          ret[n][c][a][b]=(ret[n][c][a][b] + tmp[n][c][a][b]);
          dB[n][c][a][b]=ret[n][c][a][b];
        }
      }
    }
  }
}
```

六、知识总结

本次 Project 使用到的编译知识主要是抽象语法树（AST）的构建和分析、语法制导定义（SDD）、语法制导翻译（SDT） 的设计以及贯穿全程的 Visitor 访问模式。

其中，为了简化结点，方便反向传播分析，我们用 DParser 重构了输入表达式的

AST。遍历 AST 时遇到的每个结点对应一个产生式，我们事先为每个产生式设计了用于生成导数表达式的 SDD。我们在构建 gradBuilder 编写各个结点 visit 代码过程中构建翻译动作及其顺序，完成 SDT 的设计。在编写 varMutator 时也同样完成了一对简单的 SDD-SDT 的设计。Parser 重新构建了 C 代码的 AST，大部分在 Project 1 实现的 IRPrinter 同样也是 SDD-SDT&Visitor 模式的典型应用。

通过这次 Project，我们对课程中停留于纸面的 AST、SDD、SDT 等抽象的概念落实到代码的过程有了具体的认识，同时对 Visitor 访问模式（在编译实习课程中也有大量实践）有了更加熟练的应用。

最后，感谢老师和助教在这一个学期的辛苦付出！