编译技术 Project2实验报告

小组ID：30

组员: 张家硕、曹浩威、游凌云

构建

该项目在 Ubuntu 18.04 (x64) gcc 7.5.0 环境下通过编译

```
mkdir build
cd build
cmake ..
make -j 12
```

小组分工

三人共同讨论，游凌云主要负责代码编写，曹浩威主要负责编译环境搭建和输入输出部分，张家硕主要负责实验报告编写。

自动求导技术设计

在传统的深度学习框架求导中，求导通常通过以计算图为基础，利用链式法则反向传播求导。每次计算时需要依赖预先定义的算子来计算反向计算导数，这限制了深度学习框架的可扩展性。事实上，不管是何种算子，最终都会通过带有最基础的加减乘除运算的数学表达式实现，因此，在IR层，通过对于最底层加减乘除运算的求导，我们就可以直接得到计算相应导数的代码。

每一个kernel中的stmt，都会翻译成若干层循环中的一条赋值语句。在前向传播中，我们通过这条语句来计算张量中每一个元素的值。在求导时，通过对该赋值语句求导即可得到求导代码。一般的说，对于$Output = expr(Input_1, Input_2, \ldots, Input_n)$（1），根据链式法则有，
$dInput_i = \frac{\partial loss}{\partial Input_i} = \frac{\partial loss}{\partial Output} \cdot \frac{\partial Output}{\partial Input_i}$，对（1）两边求导，即有
$dInput_i = \frac{\partial loss}{\partial Input_i} = \frac{\partial loss}{\partial Output} \cdot \frac{\partial expr}{\partial Input_i}$，而在IR中，expr中的计算都为简单的加减乘除运算，直接使用最为基础的加减乘除求导法则即可得到$\frac{\partial expr}{\partial Input_i}$，进而完成自动求导的功能。

自动求导实例

以case1为例，其kernel为：$C < 4, 16 > [i, j] = A < 4, 16 > [i, j] * B < 4, 16 > [i, j] + 1.0;$ 将其翻译为IR后，其核心赋值语句即为C[i,j]=A[i,j]*B[i,j]+1.0，对该等式进行求导，即知$\frac{\partial C[i,j]}{\partial A[i,j]} = B[i, j]$。由上述推导知，$dA = dC \cdot \frac{\partial C[i,j]}{\partial A[i,j]} = dC \cdot B[i, j]$。因此，生成的计算导数的代码应形如：

```
for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 16; ++j) {
            dA[i][j] = dC[i][j] * B[i][j];
        }
    }
```

实现流程

本实验利用了project1中的代码来构建IR语法树。具体逻辑为输入kernel字符串，通过词法、语法分析，构建出kernel的抽象语法树（借助antlr完成），在遍历语法树的过程中初步构建IR抽象语法树。这一部分大致流程如下：

```
      std::ifstream stream;
      stream.open("input.kernel");
      ANTLRInputStream input(stream);
      kernelLexer lexer(&input);
      CommonTokenStream tokens(&lexer);
      kernelParser parser(&tokens);
      kernelParser::ProgContext* tree = parser.prog();
      Kernel2IRVisitor kvisitor;
      vector<Stmt> stmtList = kvisitor.visit(tree).as<vector<Stmt> >();
```

之后，通过IRDiffer对各个stmt进行求导（IRDiffer继承自IRMutator），求导完成后，使用project1中实现的IRvisitor来推断每一个index的范围，通过IRmutator来组装外层循环，完成完整的求导后的IR抽象语法树的构造。

```
        vector<Stmt> bodyList;
        map<string, vector<size_t> > allTempList;
        IRVisitor visitor;
        set<string> ins, outs;

        for (auto stmt : stmtList) {
            vector<Stmt> newStmtList;
            for (auto g : grad_to) {
                IRDiffer differ;
                differ.grad_to = g;
                Stmt newStmt = differ.mutate(stmt).as<Stmt>();
                newStmtList.push_back(newStmt);
            }

            for (auto newStmt : newStmtList) {
                newStmt.visit_stmt(&visitor);
                cout << endl;
                for (auto in : visitor.in)
```

```
                    ins.insert(in);
                for (auto out : visitor.out)
                    outs.insert(out);

                IRMutator mutator;
                mutator.boundTable = visitor.boundTable;
                vector<Stmt> tmp = mutator.mutate(newStmt).as<vector<Stmt> >
();
                for (Stmt s : tmp)
                    bodyList.push_back(s);
                for (auto p : mutator.tempList)
                    allTempList.insert(p);
            }
        }
```

最后，生成函数签名和打印代码，完成求导流程:

```
        result << "void " << name << '(';
        bool first = true;

        cout << endl;
        for(auto i = ins.begin();i != ins.end();++i) {
            string arg_name = *i;
            auto shape = visitor.varShapeTable[arg_name];
            if(!first)
                result << ", ";
            first = false;
            if(shape.size() == 1 && shape[0] == 1){
                result << data_type << " &" << arg_name;
                continue;
            }
            result << data_type << " (&" << arg_name << ")";
            for(auto l : shape) {
                result << "[" << l << "]";
            }
        }
        for(auto i = outs.begin();i != outs.end();++i) {
            string arg_name = *i;
            if(!first)
                result << ", ";
            first = false;
            auto shape = visitor.varShapeTable[arg_name];
            if(shape.size() == 1 && shape[0] == 1){
                result << data_type << " &" << arg_name;
                continue;
            }
            result << data_type << " (&" << arg_name << ")";
            for(auto l : shape) {
                result << "[" << l << "]";
```

```
            }
        }
        result << ") {\n";

        for (auto p : allTempList) {
            if(p.second.size() == 1 && p.second[0] == 1) {
                result << " int " << p.first << ";\n";
                continue;
            }
            result << " " << data_type << " " << p.first;
            for (size_t l : p.second)
                result << "[" << l << "]";
            result << ";\n";
        }

        for (Stmt s : bodyList) {
            IRPrinter printer;
            string code = printer.print(s);
            result << code;
        }

        result << "}\n";
        result.close();
    }
```

自动求导代码实现

IRDiffer的输入是一个赋值的Stmt和一个要对其求导的变量名g，输出是一个求导后的表达式。在Move节点，对于dst成员，我们可以处理出$\frac{\partial loss}{\partial Output}$（即dOutput）。对于src成员，我们递归地mutate，遍历顺序大致是Move (-> Binary) -> Var (-> Binary) -> Index。在Binary节点，判断一下是否是Index的运算，如果不是，就说明是Var之间的运算。如果是Var之间的运算，如果运算符是加减法，那么就对两个运算数分别求导再组合。如果运算符是乘法，那么根据链式法则求导。这样递归地求导，总会到Var或者Imm。对于每个Var，如果这个Var的名字是g，那么应该返回1，否则应该返回0，对于每个Imm，返回0。考虑到最后还要乘上$\frac{\partial loss}{\partial Output}$（即dOutput），因此实现中没有直接返回1，而是返回前面处理出来的dOutput，并且注意到最后的求导表达式的dst应是dg，所以也要处理出来保存。对于返回0的情形，要对整个表达式化简，例如在Binary节点，运算是加法的情形，对a+b求导，那么应该返回a的导数+b的导数，为了把0清除掉，当a的导数和b的导数都是0的时候，返回0；当a的导数非0，b的导数为0的时候，返回a的导数；当a的导数为0，b的导数非0的时候，返回b的导数；当a的导数和b的导数都不是0的时候，才返回a的导数+b的导数。对其他的运算或节点也有类似的逻辑。这样可以在生成的求导表达式中清除掉所有的0。

实验结果

由于时间与精力限制，我们组放弃了对于case10中复杂下标的处理，通过了9/10的测试用例。

实现中的编译原理

- 词法分析。 实验中使用的词法分析为antlr工具自动生成的词法分析器，通过编写文法与词法文件，可以指导antlr自动生成语法分析器。
- 语法分析。kernel的语法分析器是由antlr自动生成的，其与Yacc不同，采用LL(k)的语法分析，使用该语法分析器可以直接的到kernel的语法分析树，通过遍历该语法分析树可以完成后续工作。
- SDT。本project中，IR语法树的构建，index的范围推导，核心stmt的求导修改，外层循环的组装，以及从IR语法树打印C语言代码，都使用了SDT，具体实现中，我们采用visitor开发模式，在自顶向下遍历语法分析树的过程中完成各项工作。
- 中间代码生成。本次实验中的主要工作是在IR层进行的（虽然本project使用的IR与课程提到的三地址代码有一定差距）。在IR层相比于kernel更加底层，能够将kernel的计算过程完整的表达出来，正是依赖于此，对于kernel的自动求导才成为可能。