Compiler Project Part 2

Jisen Ren, Haoran Wang, Xiao Han

June 20, 2020

1 Preliminaries

1.1 Reusable Code

Since this project is inherited from the first part, the following parts of the previous code could be reused to save the workload:

1. The modules for lexical and syntax analysis can be reused with little change;
2. The modules for C++ code generation based on grammar tree can be reused, but for reasons explained later, some modifications are needed to adapt to the second part.

And the module exclusive to project 2 is, to transform the grammar tree from its original to the differentiation form, which will be inserted into the code generation tool.

1.2 Cases Observation

By observing the test cases and evaluation tool, we discovered the following features which the cases share in common:

1. Unlike project 1, in project 2 the cases all have data type "float"; there is always exactly one single output and the output tensor is never among the input list; scalar multipliers like α, β never occur; each kernel has exactly one statement (however, since `grad_to` might contain multiple elements, the output is actually a list of statements).
2. The kernel of each cases is a linear combination of "indivisible sub-expressions" which are either single tensors or tensors connected by multiplication. Case 10 is special in that it contains one pair of parentheses; the rest of the cases are all simple additions of "indivisible sub-expressions". Therefore, the main issue is how to deal with these sub-expressions.
3. In terms of index, cases 6, 8 and 10 contain expressions other than single variables, but the loop ranges can always be judged from the tensor dimensions associated with single variables. In the following context, we aim to find a universal solution for dealing with index expressions, though.

2 Automatic Differentiation

Here we propose our design for theoretical model of automatic differentiation. As mentioned above, this section will cover mostly the differentiation scheme for the "sum of multiplication chains of tensors" expressions.

2.1 Mathematical Foundation

The differentiation of functions and matrices satisfy the following properties:

1. The differentiation of constant number is zero, $(c)' = 0$;
2. The differentiation of additions of sub-expressions is the sum of their separated differentiation, $(f + g)' = f' + g'$;
3. The differentiation of multiplications of sub-expression is the sum of all such parts: replace one sub-expression by its differentiation and let the rest remain unchanged, $(u \cdot v \cdot w)' = u' \cdot v \cdot w + u \cdot v' \cdot w + u \cdot v \cdot w'$;
4. Since matrix multiplication is a linear operator, its differentiation is simply calculated by replacing the matrix with its changing rate, $d(X \cdot Y) = dX \cdot Y + X \cdot dY$.

2.2 Differentiation of Multiply Chains

Consider the "multiplication chain" $O = I_1 I_2 \dots I_n$ or

$$O = \prod_{k=1}^{n} I_k$$

And we want to compute the gradient to G, or dG, then:

$$dG = \sum_{k:I_k=G} \left[\prod_{j=1}^{k-1} I_j \cdot dO \cdot \prod_{j=k+1}^{n} I_j \right]$$

Among each component of summation, the tensor whose name is identical to G is simply replaced by dO.

2.3 Issues of Index

While replacing G with dO, its index must be considered since not all indexes are expressed with single variables. Supposing G is a d dimensional vector and appears in the expression as $G[f_0(\mathcal{I}), f_1(\mathcal{I}), \dots, f_{d-1}(\mathcal{I})]$ where $\mathcal{I}$ is the set of index variables (in a text case it might appear more than once), then:

$$dG[i_0, i_1, \dots i_{d-1}] = \sum_{k:I_k=G} \left[\prod_{j=1}^{k-1} I_j \cdot \text{select} \left(\bigwedge_{j=0}^{d-1} i_j = f_j(\mathcal{I}), dO, 0 \right) \cdot \prod_{j=k+1}^{n} I_j \right]$$

The issue of index occur when $f_j(\mathcal{I}) \neq i_j$, namely, the index expression is more than a single variable.

3 Experimental Procedure

3.1 Grammar Tree Traversal

In the grammar described in project 1, R is a summation chain of E, and each E is a multiplication chain of T (except for case 10, where such property is only satisfied inside the parentheses).

Combined with the knowledge in previous sections we implement the following traversal scheme: first find out whether a pair of parentheses exist; if it does we set p to the immediate node inside the parentheses; otherwise set p to the root of the right-hand expression. For

each child c in p, if c is R then call Transform(c) recursively; if c is E then replace c with Differentiate(c).

The function Differentiate, then, returns a grammar-tree representation of the sub-expression (whose root is passed as the parameter) having the form of multiplication chain.

3.2 Assigning Additional Variables

Supposing we aim to compute the gradient to G, for each dimension of G, if an index expression occurs at least once on this dimension, then: replace this dimension of left side dG with an additional loop variable, and attach an equality constraint to the corresponding right-hand sub-expression.

This method is somewhat inefficient, but sufficient to solve the cases in the project, in it satisfies the requirement that left-side tensors do not contain index expressions, and the method does not make use of the cases' names. Moreover, within the scale of the given cases, the loops can be executed without taking much time.

3.3 List of Function Parameters

After obtaining the processed grammar tree, we determine the function parameters in the following order:

1. The tensor name(s) among the list `ins` which appear in the right-hand expressions at least once;
2. The character 'd' plus the tensor name in `outs`;
3. The character 'd' plus the tensor name(s) in `grad_to`.

3.4 Ideas for Refinement

The pursuit for efficiency led us to consider ideas for refinement. It would be better if extraneous loops can be eliminated by computing the inverse operator. For example, the inverse of $p+r$ is $p-r$, when r does not appear in the tensor to be "graded" over. However, the inversion of integer division and mod operator is confusing, since each of these operators itself is not even reversible. If we regard the combination $(//, \%)$ as an operator which maps an integer to a pair of integers, then this operator is indeed reversible.

However, how should we know that // and % always appear in pairs without examining the ground truth? If they do appear in pairs, how should we find out that truth efficiently? What about more complicated cases, should they also be considered? These issues can be too complicated to solve. The method based on extra loop variables and equality constraints, however, solves these issues universally. That is why we apply this method in the submitted solution.

4 Concrete Example

In this section we demonstrate our differentiation scheme using a concrete example. We choose the most complicated example, a combination of matrix multiplication and convolution.

In case 6, the input string `kernel` is: `A<2, 8, 5, 5>[n, k, p, q] = B<2, 16, 7, 7>[n, c, p + r, q + s] * C<8, 16, 3, 3>[k, c, r, s];` and we need to compute its gradient to `B`. The right side itself is an indivisible sub-expression without parentheses, so we begin transformation from the root; from the differentiation rule, if $A = B * C$, then $dB = dA * C$.

In terms of index, since the third and fourth dimensions contain expressions, we assign an extra loop variable to each of these dimensions and attach an equality constraint to the right-side component. The answer is thus: `dB<2, 16, 7, 7>[n, c, i2, i3] = (i2 == p + r && i3 == q + s ? dA[2, 8, 5, 5]<n, k, p, q> : 0) * C<8, 16, 3, 3>[k, c, r, s];`.

Among all cases, this is the one which needs to execute the loop for most iterations. Even so, a total of 2822400 iterations finish in a matter of milliseconds while running `test2`.

5 Knowledge Applied

Here we give a brief summary of the compiler knowledge applied in the project:

1. Lexical Analysis. After reading the string from JSON files, the lexical analysis is first performed to obtain a sequence of tokens;
2. Syntax Analysis. We apply LR automaton based syntax analysis scheme to the (actually ambiguous) grammar, and resolve conflicts properly to construct the grammar tree.
3. Syntax Directed Definition and Translation. For each tensor reference as well as the whole expression, we associate semantic actions to build the `for` loops, the `if` conditions, as well as the C++ representation of tensor references together with the conditions related to its indexes.
4. Knowledge Exclusive to Project 2. The "Differentiation" action is also syntax directed, which is the foundation of grammar tree transformation.

A Task Distribution

All group members have played a significant role in the progress of projects, hence it is suggested that all members be given score equally.

A.1 Jisen Ren

Performed LR based syntax analysis in project 1 and the majority of effort of syntax transformation in project 2; responsible for assembling the report documents.

A.2 Zhiyi Cheng, who withdrew the course halfway

Read input from JSON files using third-party library; performed lexical analysis to obtain the token sequence.

A.3 Haoran Wang

Completed the code generation part, in which the established syntax tree is translated into the target C++ code; provided useful and convenient auxiliary functions.

A.4 Xiao Han

Made an attempt to refine the second project aiming at eliminating redundant loops, by exploiting the principles of "inverse operators".