The Lab Component of ACM Summer School for AI/ML

17 June 2024 to 28 June 2024

Abstract

This document describes a small Domain Specific Language called AIDSL tailored specifically for Artificial Intelligence (AI) and Machine Learning (ML) applications. It has been designed for your lab component of the school. You are expected to write a compiler for this language. The document also suggests a plan for your implementation.

1 Overview

You are expected to write a compiler that will take a program in AIDSL and output a low level C++ program which can then be compiled by any C++ compiler.

You will be given the following:

1. An executable of the reference implementation. You can run this reference implementation to understand the expected output of the compiler.
2. A base source code. You can enhance this base source by adding the missing features of the language and that of the compiler.
3. A support library for the operations that you can use in the generated C++ code.

The generated C++ file can be compiled using `g++` command. Note that the header file "`api.h`" contains all the library functions. Class `Tensor` is a template-based class having public member functions and private variables (dimension, row and column, and a pointer variable). The parameterized constructor is responsible for allocating memory to the pointer variable.

2 Suggested Implementation Plan

You should begin by understanding the base code and extend it incrementally in the following steps:

1. Supporting the remaining operators and built-in functions described in Sections 3.3 and 3.4. These are `@`, `**`, all relational operators, `trans`, and `input` and `output` statements.
   (a) Add tensors as operands and support expressions without any type checking.
      i. In the first step, add the specifications of the tokens to the lex script. Use the options `-tn`; `t` shows the tokens identified and `n` suppresses the parser from being invoked.
      ii. In the second step, add grammar rules to the yacc script. Use the options `-tep`; `e` suppresses the semantic analysis, and `p` shows the rules used for reduction in parsing.
      iii. In the third step, add C++ code in the semantic actions (for the `lp_mode() == compiler` parts of the `if` statements in the `support.cc` file). Ignore the interpretation mode. You will need to extend the class hierarchy of the `Ast` class. Add the `print_ast` function for each class you create. You can see the documentation of class hierarchy by viewing `html/index.hml` file using any browser; this documentation has been generated using `doxygen`.
      Use the options `-tpa`; option `a` shows the AST constructed for each statement.

iv. In the fourth step, add a `gencode` function to each class which traverses an AST and generates (for now, approximate) C++ code by calling appropriate APIs from the support library. This code is approximate because you would not have the sizes of the tensors and hence all parameters to the APIs would not be emitted. The idea is not to get executable code but to get a feel of what it means to generate code.

(b) Add declaration statements (see Section 3.5), check types, and complete the generated code.

For this, you need to repeat steps i, ii, iii, and iv given above, to support declarations.

Steps i and ii would remain similar. However, in step iii, now you will not construct ASTs but will add the type and size information to a symbol table which is essentially a map from a name to this information stored in a suitable data structure. You will also check types in this step.

Note that type checking for tensor operations consists of three steps: first, the base type should be same; second, the number of dimensions should match and; third, the sizes of dimensions must be compatible. For example in order to multiply two matrices, `m1 ** m2`, they both must store the same type of elements, both must have 2 dimensions, and if the sizes are $r_1 \times c_1$ and $r_2 \times c_2$, c_1 must be same as r_2. You should generate the size of the resulting matrix as $r_1 \times c_2$ and generate the declaration of the temporary. Figure 2a provides an example.

In step iv, you will modify the `gencode` functions to complete the calls to APIs. You will also convert the output into a compilable `main` function.

With this you have a compiler that accepts sequence of the following statements: declaration statements, assignment statements, input statement, and output statement. It would generate a C++ code that you can compile and execute.

2. Supporting the remaining statements described in Section 3.5.

You will once again repeat the steps i, ii, iii, and iv and now be able to support the full AIDSL.

3 AIDSL Specifications

3.1 Data Types

There exist two distinct categories of data types in AIDSL.

- **Primitive Data Type:** The primitive data types of the given language are: `float32, float64, int8, int16, int32, int64`.

- **Tensor Data Type:** The tensor data type denoted as `T(n, pd)` is an n-dimensional array variable (named `T`) of elements of type `pd`, where `pd` can be any one of the primitive data types mentioned above. The number of elements at each tensor dimension can be represented using square brackets. For example, `t1(2, int32)[n][n]` represents a two-dimensional $n \times n$ matrix (called `t1`) where each element is of type `int32`. The actual size of the tensor (i.e. the number of elements in different dimensions) must be known at compile time (i.e., it must be provided as a constant in the declaration). The number of dimensions is known and is currently fixed to 1 and 2. This limitation comes from the use of support library.[1]

  Note that the indexing of tensor data types always starts from zero.

3.2 Values

Values in AIDSL can be of three kinds: (1) Constants, (2) Scalar variables of primitive data types, and (3) Tensor variables of primitive data types. Like C language, variable identifiers begin with a letter or underscore and can be followed by a sequence of letters, underscores or digits. Identifiers are case-sensitive.

[1]Please think more about it and ask a TA to explain it to you :-)

```
//tensor variables
t1(1, int32)[10];
_t2(1, int32)[10];
int32 var; //scalar variable
//Error: tensor size-dimension mismatch
t1(2, int32)[10];
```

Figure 1: Declaration of tensors and scalars

3.3 Operators

1. **Binary Arithmetic**: $\boxed{E\, op\, E}$ where E is an expression (which could be a variable V).

   The list of valid binary arithmetic operators is: addition(`+`), subtraction(`-`), multiplication(`*`), matrix multiplication (`**`), division (`/`), and convolution (`@`). Both operands may be of scalar or tensor types, except for convolution and matrix multiplication, where both operands must be tensors. For other binary operations, one variable can be a scalar variable or a constant, while the other is a tensor.

   A binary operation between a tensor and a scalar is equivalent to the same operation on all the tensor elements. Multiplication (`*`) and division (`/`) operations between tensors represent element-wise multiplication and division. Matrix multiplication (**) multiplies two two-dimensional tensors (following the matrix multiplication rule) and returns a single two-dimensional tensor.

2. **Unary Arithmetic**: $\boxed{op\, E}$ where E is an expression (which could be a variable V).

   We have only a single unary operator: negation(`-`). The variables can be both scalar or tensor types.

3. **Relational Operators**: $\boxed{E\, op\, E}$ where E is an expression (which could be a variable V).

   List of valid Relational operators is: greater-than (`>`), less-than (`<`), greater-than-or-equal (`>=`), less-than-or-equal (`<=`), equal (`==`), and not-equal (`!=`). Each relational operator operates on two operands and returns true (1) or false (0). Note that both the operands must be of the same variable type (either scalar or tensor). For tensor data types, the first four relational operators (`>, <, >=, <=`) only compare the length of the given two tensors. The equal (`==`) and not-equal (`!=`) operations not only compare the length of two tensors but also perform an element-wise comparison.

3.4 Builtin Functions

AIDSL supports three builtins with the following APIs:

- The API $\boxed{\texttt{input}\ (V)}$ takes only a single variable as an argument. The argument must be a scalar or tensor variable but not a constant. It reads input from the standard input device and stores it in the argument variable. For a tensor, this API reads input for each tensor element.

- The API $\boxed{\texttt{output}\ (V)}$ displays output to the standard output device. It also takes a single variable as an argument to the API call. For a tensor variable, it prints the value of each element of the tensor, and for a scalar, it prints the value of the scalar to the standard output device.

- The API for transpose operation, $\boxed{\texttt{trans}\ (V)}$ The argument, V, represents the tensor variable, which is to be transposed. The `trans` API returns a tensor as an output.

3.5 Statements

AIDSL supports the following statements. There are no functions or procedures so an AIDSL program is just a sequence statements.

- Declaration statements. Figure 1 contains some examples of declarations of scalars and tensors. They may appear anywhere but a variable must be declared before its use. For simplicity, only one variable name can appear in a declaration statement and declaration cannot contain initialization.
- Assignment statements. They have the usual syntax $\boxed{V = E;}$ where E is an expression containing the scalars, tensors, the operators (Section 3.3), and the `trans` builtin function (Section 3.4).
- There are two forms of the IF statement: $\boxed{\texttt{if } (E) \texttt{ then } S \texttt{ else } S}$ and $\boxed{\texttt{if } (E) \texttt{ then } S}$ where S is any statement and E is an expression.
- AIDSL also has a WHILE statement of the form $\boxed{\texttt{while } (E)\ S}$ where S is any statement and E is an expression.
- AIDSL supports a compound statement which is a sequence of statements in a pair "{" and "}".
- The input and output statements use the builtins described in Section 3.4. They must be terminated by a semicolon.

3.6 Reserved Keywords

The reserved keywords in AIDSL language are: {`trans, input, output, if, then, else, while`}. These keywords cannot be used as variable names.

4 Support Library for Use in the Generated C++ Code

The AIDSL compiler translates AIDSL code into C++ code, leveraging a dedicated set of library functions and data types to handle tensors. This section details the structure of the generated C++ code and describes the provided C++ library.

These APIs cannot be used in an AIDSL program; the AIDSL compiler does not understand them.

4.1 Declaring Tensors

The tensors have been implemented in the provided C++ library as a template-based class. This class supports basic arithmetic operations, comparison operations, matrix multiplication, and convolution for 1D and 2D tensors. The tensors can be declared using the following syntax:

```
Tensor<int32_t> matrix1(dim, row, col);
```

- `Tensor<int32_t>`: This specifies the type of the tensor. `Tensor` is a template class that can hold elements of any specified type. In this case, `int32_t` (a 32-bit integer) is the type of the elements stored in the tensor.
- `matrix1`: This is the name of the tensor variable being declared. The tensor will be referenced by this name in the code.
- `(dim, row, col)`: The first parameter `dim` specifies the number of dimensions of corresponding tensor. `row` and `col` specifies the size of the row-wise and column-wise size of the tensor. Note that `dim`, `row`, and `col` always must be of type `int32_t`.

For example, in Figure 2a, at line 2, a 2D tensor of size 2×3 has been declared. Note that when declaring a one-dimensional array, the last argument is ignored. For example, Line 13, and 14 in Figure 2a both are valid statements to declare an array of size 2.

4.2 Getters and Setters

One can initialize an empty tensor using the following syntax:

```
    Tensor<int32_t> matrix2;
```

However, before using an empty tensor, it is necessary to define its dimensions and specify the size along each dimension using the `set` function.

```
    matrix2.set(dim, row, col);
```

Our library supports three different getter functions: (1) `getDim()`, (2)`getRow()`, and (3)`getCol()`. Each of these getter functions returns an integer value, which represents the dimension, the number of rows, and the number of columns of a tensor, respectively. The syntax of the getter functions are as follows:

```
    int d = matrix1.getDim();
    int r = matrix1.getrow();
    int c = matrix1.getCol();
```

4.3 Utility Functions

(a) `input()`: It reads tensor data from stdin. Each data element in the input must be separated by newline character.
Syntax:

```
    matrix1.input();
```

Return Type: `void`

(b) `output()`: It writes the tensor data to stdout.
syntax:

```
    matrix1.output();
```

Return Type: `void`

(c) `trans()`: It returns the transpose of the corresponding tensor. The number of rows and columns of the input are exchanged in the result.
Syntax:

```
    matrix1.trans();
```

Return Type: `tensor`
Example: In Figure 2a, at line 15 transpose of `matrix1` is assigned to tensor `t0`.

(d) `scalar_mul(var)`: It multiplies a scalar variable with each element of the corresponding tensor and returns the resultant tensor.
Syntax:

```
    matrix1.scalar_mul(var);
```

Return Type: `tensor`
Example: In Figure 2a, at line 17 each element of `matrix1` is multiplied by 5, and the resultant tensor is assigned to tensor `t1`.
Similar Functions: We have provided similar library functions for addition, subtraction and division named as `scalar_add`, `scalar_sub`, and `scalar_div`, respectively.
Syntax:

```
    matrix1.scalar_add(var);
    matrix1.scalar_sub(var);
    matrix1.scalar_div(var);
```

(e) `equals(tensor)`: It checks whether the given two tensors are equal (dimension-wise, size-wise and element-wise) or not.
Syntax:

```
    matrix1.equals(matrix2);
```

Return Type: `boolean`
Example: In Figure 2a, at line 19, boolean variable `flag` indicates whether tensor `result1` and `matrix1` are equal.
Similar Functions: We have provided library functions for other relational operations also. The library functions have similar syntax as `equals` and are named as `lt` (less-than), `gt` (greater-than), `leq` (less-than-equal), and `geq` (greater-than-equal).
Syntax:

```
    matrix1.lt(matrix2);
    matrix1.gt(matrix2);
    matrix1.leq(matrix2);
    matrix1.geq(matrix2);
```

(f) `add(tensor)`: It adds two given tensors and returns a resultant tensor.
Syntax:

```
    matrix1.add(matrix2);
```

Return Type: `tensor`
Example: In Figure 2a, at line 21, two tensors (`matrix1` and `result1`) are added and the resultant tensor is stored in tensor `t2`.
Similar Functions: We have provided library functions for other tensor arithmetic operations also. The library functions have similar syntax as `add` and are named as `sub` (tensor-subtraction), `mul` (tensor-element-wise-multiplication), and `div` (tensor-element-wise-division).
Syntax:

```
    matrix1.sub(matrix2);
    matrix1.mul(matrix2);
    matrix1.div(matrix2);
```

Note: In arithmetic operations, the resultant tensor must have same number of rows and same number of columns as that of the input tensor.

(g) `negate()`: It performs a negation operation on the given tensor (element-wise).
Syntax:

```
    matrix1.negate();
```

Return Type: `tensor`

(h) `mm(tensor)`: It multiplies two 2D tensors using matrix-multiplication technique.
Syntax:

```
    matrix1.mm(matrix2);
```

Return Type: `tensor`
Example: In Figure 2a, at line 25, two tensors (`matrix1` and `matrix2`) are multiplied (using matrix-multiplication rule) and the resultant tensor is stored in `t3`.
Note: The resultant tensor must have the size of $(r_1 \times c_2)$, where r_1 is the number of rows in `matrix1` and `c2` is the number of columns in `matrix2`. Moreover, both the tensors must be two-dimensional tensors.

(i) `conv(tensor)`: It performs 2D convolution operation.
Syntax:

```
1  int main(){
2      Tensor<int32_t> matrix1(2, 2, 3);
3      Tensor<int32_t> matrix2(2, 3, 2);
4      Tensor<int32_t> t0(2, 3, 2);
5      Tensor<int32_t> t1(2, 2, 3);
6      Tensor<int32_t> result(2, 3, 2);
7      Tensor<int32_t> result1(2, 2, 3);
8      Tensor<int32_t> mm_res(2, 2, 2);
9      Tensor<int32_t> conv_res(2, 2, 3);
10     Tensor<int32_t> t2(2, 2, 3);
11     Tensor<int32_t> t3(2, 2, 2);
12     Tensor<int32_t> t4(2, 2, 3);
13     Tensor<int32_t> array1(1, 2, 0);
14     Tensor<int32_t> array2(1, 2);
15     t0 = matrix1.trans();
16     result = t0;
17     t1 = matrix1.scalar_mul(5);
18     result1 = t1;
19     bool flag = result1.equals(matrix1);
20     if (flag==0) goto L0:
21     t2 = matrix1.add(result1);
22     result1 = t2;
23     goto L1;
24     L0:
25     t3 = matrix1.mm(matrix2);
26     mm_res = t3;
27     L1:
28     t4 = matrix1.conv(array1);
29     conv_res = t4;
30     return 0;
31 }
32
```

(a) Example of C++ code corresponding to AIDSL code

```
    matrix1(2, int32)[2][3];
    matrix2(2, int32)[3][2];
    result(2, int32)[3][2];
    result1(2, int32)[2][3];
    mm_res(2, int32)[2][2];
    conv_res(2, int32)[2][3];
    array1(1, int32)[2];
    array2(2, int32)[2];
    result = trans(matrix1);
    result1 = matrix1 * 5;
    bool flag = result1==matrix1;
    if (flag){
        result1 = matrix1 + result1;
    }
    else
    {
        mm_res = matrix1 ** matrix2;
    }
    conv_res =  matrix1 @ array1;

```

(b) Example of AIDSL code

Figure 2

```
    matrix1.conv(matrix2);
```

Return Type: `tensor`
Example: In Figure 2a, at line 28, we perform a convolution operation on `matrix1`. Here, `array1` is the kernel matrix and `matrix1` is the input tensor for the convolution operation.
Note: The resultant tensor must have the same row size and column size as the input tensor.

5 Credits

AIDSL was designed by Soumik Kumar Basu, IIT Hyderabad, (`cs21resch11004@iith.ac.in`) under the guidance of Jyothi Vedurada, IIT Hyderabad, (`jyothiv@cse.iith.ac.in`). Soumik also implemented the support library. The base code of the compiler was implemented by Uday Khedker, IIT Bombay, (`uday@cse.iitb.ac.in`). It was extended to the full-fledged reference implementation by Bhavya Hirani, SVNIT Surat, (`u21cs100@coed.svnit.ac.in`).