tensor-library

Second Assignment for Advanced algorithms and programming methods [CM0470] course.

We made some changed to an existing Tensor Library to allow operations using the Einstein Notation. The special Einstein Notation for tensors allows summation, subtraction and multiplication using variables, by rendering implicit the summation. According to this notation, repeated variables in a tensorial expression are implicitly summed over, so the expression

$a_{ijk}b_j$

represents a rank 2 tensor c parameterized by i and k such that

$c_{ik} = \sum_j a_{ijk}b_j$

The notation allows for simple contractions

$\mathrm{Tr}(a) = a_{ii}$

as well as additions (subtractions) and multiplications.

In particular, we respect these essential rules of Einsten notation:

1. Repeated variables are implicitly summed over;
2. Each variable can appear at most twice in any term;
3. Each term must contain identical non-repeated variables.

Of course, when we define a same variable on two tensors, the dimension on that variable must be the same.

Usage

The library adds a new method `ein()` to the tensor class. The method does not take parameters, but only the variables as template parameters. This example performs the expression described before:

$c_{ik} = a_{ijk}b_j$

```
tensor::tensor<int> a({3, 3, 3});
tensor::tensor<int> b({3});

auto exp = a.ein<'i', 'j', 'k'>() * b.ein<'j'>();
tensor::tensor<int> c = exp.evaluate<'i', 'k'>();
```

The `ein()` method does not calculate anything but returns an opaque `tensor_expression` object, which can be turned into a resulting tensor using the `evaluate()` method. You have to pass the resulting variables to the method as template parameters. This way it can be used also to transpose a tensor:

```
tensor::tensor<int> a({3, 3});
tensor::tensor<int> transpose =
    a.ein<'i', 'j'>().evaluate<'j', 'i'>();
```

If the operations returns a scalar (for example the trace) it will be wrapped on a single-element tensor:

```
tensor::tensor<int> a({3, 3});
auto exp = a.ein<'i', 'i'>();
tensor::tensor<int> trace = exp.evaluate<>();
// std::cout << trace({0});
```

Of course, since the variables are templates, it's possibile to choose them only at compile-time. **No run-time expressions are possible.** We did this to have better performance, since run-time dependant expressions are very rare. This allows also to check expression validity at compile-time:

```
tensor::tensor<int> a({3, 3});
tensor::tensor<int> transpose =
    a.ein<'i', 'j'>().evaluate<'k', 'p'>();

// error: static_assert failed due to requirement
match_vars<tensor::expressions::vars<'k', 'p'> tensor::expressions::vars<'i',
'j'>, void>::value' "The free variables on both the sides of an equation must
match"
```

You can perform operations with tensor_expressions, such as sum, multiplication and negation:

```
tensor::tensor<int> a({2, 2});
tensor::tensor<int> b({2, 2});

auto exp = a.ein<'i', 'j'>() + b.ein<'i', 'j'>();
tensor::tensor<int> c = exp.evaluate<'i', 'j'>();
```

We suggest to use inference with `auto` for the expression type, since it can be extremely complicated due to compile-time checks and variables.

How it works

First of all, when we assign variables to the indices of a given tensor though the `ein()` method, a `tensor_constant` (which is a tensor_expression) object is created.

We can perform operations such as addition ($A + B$), negation ($-A$) and multiplications ($A * B$) between `tensor_expressions` in order to get more complex expressions, and for each one of these operations, a `tensor_addition` / `tensor_negation` / `tensor_multiplication` object (which is also a `tensor_expression`) is created. This is achieved using C++ **operator**

overloading. We can also achieve subtraction by combining an addition and a negation ($A - B = A + (-B)$).

This builds an expression tree, and every term is copied inside the expression object, to avoid losing references.

When a `tensor_expression` is created, at compile-time the compiler checks if such expression is valid or not, by performing a static verification on the expression variables. For e.g., a `tensor_constant` is valid if there are at most two variables with the same identifier. We also use template types to statically obtain the free variables and the repeated (dummy/bound) variables involved in the expression. Afterwards, at run-time, we check if all the variables with some identifier are referred to variables with the same dimension; this is because we don't know the information about the dimensions of a tensor at run-time.

Evaluation

When we want to get the result of an expression (which is always a tensor of rank known at compile-time), the `evaluate()` method is called, which calculates, for each combination of indexes over the free variables of the expression, the summation of the expression over the repeated variables, and assigns the resulting scalar to the correct position of the resulting tensor:

```
tensor<T, rank<result_rank>> result(dims);

size_t free_vars_values_count =
    count_vars_values(result_vars::id, result_vars::size);

for (size_t i = 0; i < free_vars_values_count; i++) {
  auto free_vars_values =
      get_nth_vars_values(i,
        result_vars::id,
        result_vars::size
      );

  result(
    to_indexes(
      result_vars::id,
      result_vars::size,
      free_vars_values
    )
  ) = evaluate_summation(free_vars_values);
}

return result;
```

In particular, `evaluate_summation()` is the method which performs the summation of the expression over its own repeated variables:

```
T result = evaluate_direct(bind_vars_values(
    vars_values,
```

```
    get_nth_vars_values(0, repeated_vars::id, repeated_vars::size)));

size_t vars_values1_count =
    count_vars_values(repeated_vars::id, repeated_vars::size);

for (size_t i = 1; i < vars_values1_count; i++) {
  result += evaluate_direct(bind_vars_values(
      vars_values,
      get_nth_vars_values(i,
        repeated_vars::id,
        repeated_vars::size)
    ));
}

return result;
```

The effective evaluation of an expression, given some values of variables, is performed by the `evaluate_direct()` method, whose implementation depends on the specific type of the expression:

- for a **tensor_constant**, `evaluate_direct()` accesses the tensor at the point specified by replacing the variables with the actual values;
- for a **tensor_addition**, `evaluate_direct()` performs the addition between the summations (i.e. `evaluate_summation()`) of the two inner expressions;
- for a **tensor_negation**, `evaluate_direct()` negates the value of `evaluate_direct()` of the inner expression;
- for a **tensor_multiplication**, `evaluate_direct()` performs the multiplication between the values of `evaluate_direct()` of the two inner expressions.

Variables

As we said, variables are passed as template parameters, and this allows static checking and evaluation. This is done by exploiting template specialization.

The `ein()` method takes a variadic sequence of chars as template parameters:

```
template <char... Is>
inline tensor_constant<T, vars<Is...>> ein() {
  return tensor_constant<T, vars<Is...>>(*this);
}
```

This way we are passing our variables to the `vars` utility type, which represents a set of variables. The `vars` type is defined as following:

```
template <char...>
struct vars;

template <char... Is>
```

```
struct vars {
  constexpr static size_t size = sizeof...(Is);
  constexpr static char id[sizeof...(Is)] = {Is...};
};

template <char... Is>
constexpr char vars<Is...>::id[sizeof...(Is)];

template <>
struct vars<> {
  constexpr static size_t size = 0;
  constexpr static char id[1] = {'\0'};
};

constexpr char vars<>::id[1];
```

We also have some utility types that allows operations on variables, such as `concat_vars`, `single_vars` and `match_vars`. They are used to find free and repeated variables, and to validate the expressions. Every expression type defines their free and repeated variables using a member type. For example:

```
template <typename T, typename Derived>
class tensor_expression {
 public:

  using free_vars =
    typename single_vars<
      typename term_multi_vars<Derived>::value>::value;

  using repeated_vars =
      typename double_vars<
        typename term_multi_vars<Derived>::value>::value;

  constexpr static size_t result_rank = free_vars::size;

  //...
};
```

Another template, named `match_vars`, uses *SFINAE* recursively to check if two `vars` types have the same variables inside:

```
template <typename, typename, typename = void>
struct match_vars;

template <>
struct match_vars<vars<>, vars<>> {
  constexpr static bool value = true;
};
```

```
template <char... As, char... Bs>
struct match_vars<
    vars<As...>, vars<Bs...>,
    typename std::enable_if<
      sizeof...(As) != sizeof...(Bs)>::type> {
  constexpr static bool value = false;
};

template <char A, char... As, char... Bs>
struct match_vars<
    vars<A, As...>, vars<Bs...>,
    typename std::enable_if<
      1 + sizeof...(As) == sizeof...(Bs)>::type> {
  constexpr static bool value =
      match_vars<
          typename remove_var<A, vars<As...>>::value,
          typename remove_var<A, vars<Bs...>>::value
      >::value;
};
```

Expression validation in compile-time is done using the `validate_expression` type, which takes a `vars` type checking the validity recursively on all child expression. For example:

```
template <typename T, typename A, typename B>
struct validate_expression<tensor_multiplication<T, A, B>> {
  constexpr static bool value =
      validate_expression<A>::value &&
      validate_expression<B>::value &&
      at_most_2_equals_vars<typename term_multi_vars<
          tensor_multiplication<T, A, B>>::value>::value;
};

// on tensor_multiplication class
tensor_multiplication() {
  static_assert(
    validate_expression<Derived>::value,
    "Invalid expression"
  );
}
```