D1635R0 : An STL-like Design for Linear Algebra Library

Jayesh Badwaik

June 13, 2019

A linear algebra library is currently being considered for standardization in the C++ standard library through [1]. Among other things, the above paper puts forth the objective of providing facilities and techniques for customization that enable users to optimize performance for their specific needs in a manner that only requires them to implement a minimal set of types and functions to integrate it with the rest of the linear algebra libraries.

In this paper, we claim that a well-designed set of requirements imposed on the types and the functions allow us to write a linear algebra library which fulfills the above two objectives. We also claim that the design allows one to carry out operation between types of libraries designed as above without having to perform generate additional temporaries for the purpose of compatibility of the two libraries. As the name suggests, the design is motivated by the design of the Standard Template Library which also follows the similar philosophy of making functionality available to the user if they implement types which satisfy a certain set of requirements.

1 Introduction

Following from the motivation of [2], [1] presents a design of linear algebra library that has a set of objective.

Talk about objective, talk about how the objectives show themselves in terms of customization, operator algebra and other things. Discuss Blaze library multiplication. Discuss difficulties of move operations with a+b. and so on. Operators are important, why? Generic algorithm. Examples. How operators differ. Deciding which algorithm to use to implement operators. Why are operators important? Discuss expression templates. About expression templates refer to Klaus Igelberger paper. Move operations in C++, and problems with expresion templates. Finally, discuss the situation about different library types.

common vocabulary types to allow different code bases to use each other without having to make intermediate copies.

The problems arise when one tries to deal with operators. For example, how should the expression `a+b`. Should we return a view? Or an expression template? Or are we consuming a temporary and hence, should return by value. How to deal with `a+b` where `a` and `b` might belong to a different library. In current setups, there is no hope of interoperability, because none/both of the libraries might implement `a+b`. Both the situations are not ideal. Is there a way to provide some interoperability? Optimal interoperability will still require one to be careful, but can we provide a way to do it in a way that is consistent with other good practices in C++?

Single type/library design. Dilemna of mixing library types.

1. Comparison with ‘std::vector‘, ‘boost::vector‘, In most algorithms, one can use all the three without any functional difference.

2. Why should linear algebra be different?

3. At some point, people will need different types.

4. Much more beneficial in our opinion to allow a design which accomodates different types rather than ask everyone to use a single type, however good that type is.

5. Same goes for algorithms.

6. This also lowers the expectation on the standard library to implement all types. This ensures that people can keep using standard library types and algorithms for their daily use without having to completely throw them away in cases where they wish to use something else.

So, allow the user to be able to custom define operations like `a+b` where a and b belong to different libraries.

This also lowers the expectation on the standard library design itself. Because if the there is something which is not there in the standard library or if the standard library for some reason or the another is not the fastest option, then people can use other libraries without having to completely redesign their code/introduce expensive copies.

1.1 Some Comments about the Paper

This is not a proposal about including certain linear algebra types or functions in the standard library. Rather, it is a paper which presents how the types and functions can be designed to allow types from different library to interact with one another. Therefore the types presented in this paper are minimal in that, only the features required to highlight the functionality are implemented.

We are aware that a lot of language in the paper can be rewritten in the form of Concepts but we are not comfortable with the language and decided to use a traits based exposition in order to make sure that we do not introduce additional errors in the papers.

1.2 A Comment about the Code

2 Some Traits

We start with introducing some trait types which will be useful for the discussion

isvector trait ismatrix trait

```cpp
// Inherit from std::true_type if all the `Tp`s have a member typedef
// `engine_type` else inherit from std::false_type
template <class... Tp>
struct is_engine_aware {
};

// Inherit from std::true_type if all the `Tp`s have a member typedef
// `engine_type` and the type `engine_type` is convertible to `E` else inherit
// from std::false_type
template <typename E, typename... T>
struct uses_engine {
};


// Inherit from std::true_type if all the `Tp`s have a member typedef
// `is_owning_type` and the type `is_owning_type` is convertible to
// `std::true_type` else inherit from std::false_type
template <class Tp>
struct is_owning_type {
};

// Inherit from std::true_type if any one of the `T` satisfy both of the
// following properties:
// 1. std::is_owning_type_v<std::remove_cv_ref_t<T>> is true
// 2. std::is_rvalue_reference_v<T> is true
template <typename... T>
struct is_consuming_owning_type {
};
```

Listing 1: Supporting Traits

3 Engine-Aware Types

The dilemna for defining ‘a+b‘ without copy comes from the fact that if one follows the good practices of not declaring operator for in the namespace of someone else, neither ‘a‘ nor ‘b‘ nor the user ‘c‘ can define the ‘operator+(a,b)‘. We resolve this difficulty as follows:

```cpp
  template<typename E>
  class vector {
  public:
    using engine_type = E;
    using is_owning_type = std::true_type;

  public:
    template <typename NE>
    auto change_engine() && -> vector<NE>;

    template <typename NE>
    auto change_engine() & -> vector_view<NE, vector<E>>;

    template <typename NE>
    auto change_engine() const& -> vector_view<NE, vector<E> const>;
  };
```

Listing 2: An Engine-Aware Owning Type

```
  template<typename E>
  class vector_view {
  public:
    using engine_type = E;
    using is_owning_type = std::false_type;

  public:
    template <typename NE>
    auto change_engine() const -> vector_view<NE, owning_type>;
  };
```

Listing 3: An Engine-Aware View/Expression Type

4 Engine and Engine-Based Operators

```
struct serial_cpu_engine;

template <typename T, typename U>
struct addition_traits {
  using result_type = X ; // X can be determined using any logic required
};

template <typename T, typename U>
struct addition_engine {
  auto operator()(T&& t, U&& u) -> addition_traits_r<T, U>
  {
    auto const size = t.size();
    auto result = addition_traits_r<T, U>(size);
    for (std::size_t i = 0; i < size; ++i) {
      result(i) = t(i) + u(i);
    }
  }
};

template <typename T,
          typename U,
          typename = std::enable_if<
            std::experimental::math::uses_engine_v<serial_cpu_engine, T, U>>>
auto operator+(T&& t, U&& u) -> addition_traits_r<T&&, U&&>
{
  auto constexpr ae = addition_engine<T, U>();
  return ae(std::forward<T>(t), std::forward<U>(u));
}

template <typename T,
          typename U,
          typename = std::enable_if<
            std::experimental::math::uses_engine_v<serial_cpu_engine, T, U>>>
auto operator+(T&& t, U&& u) -> addition_traits_r<T&&, U&&>
{
  return t.change_engine<some_other>() + u.change_engine<some_other>();
}
```

Listing 4: Engine and Engine-Based Operator

5 Some Musings on Design Decisions

Compile Times due to SFINAE IsVector trait

5.1 Use of ADL in Detection of Correct Operator Namespace

Justification and Pitfalls (Could also be thought of as good design decisions.)

Good practice is not to write operators for linear algebra types but instead write them for the engines.

5.2 More Fine Tuned Traits

Arbitrarily more finely tuned traits can be taken care of the `addition_traits`. But we should be careful as to prevent trait explosion.

1. 'Sparsity' trait definitely needed
2. 'Host/Device' trait should be useful
3. Access Pattern Traits

6 Example

```
1
2   using sce = std::math::serial_cpu_engine;
3   using pce =custom_lib::parallel_cpu_engine;
4   using stdvec = std::math::vector;
5   using custvec = another_lib::vector;
6
7   auto const ov1 = stdvec<sce>(arg...);
8   auto const ov2 = custvec<sce>(arg...);
9
10  auto const view_3 = ov1 + ov2;
11
12  auto const view_4 = ov1.change_engine<pce>();
13  auto const ov5 = (custvec<sce>(args...)).change_engine<pce>();
14
15  auto const view_6 = view_4 + ov5;
16
17  auto const ov7 = stdvec<pce>(args...) + ov1;
```

Listing 5: Example of How the Code Might Look

7 Acknowledgement

A lot of ideas have been gathered by participating in the SG14 Linear Algebra SIG discussions, reading the codebases like Eigen, MTL, Blaze and others etc.

References

[1] Bob Steagall Guy Davidson. **P1385R1: A proposal to add linear algebra support to the C++ standard library**. 2019. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1385r1.html (visited on 06/12/2019).

[2] Bob Steagall Guy Davidson. **What do we need from a linear algebra library?** 2019. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1166r0.pdf.