D1635R0 : An STL-like Design for Linear Algebra Library

Jayesh Badwaik

June 5, 2019

A linear algebra library is being currently being considered for standardization. In this paper, we present some of the techniques that will allow us to design a library that will allow interoperability with other suitably-designed linear algebra libraries. The design is inspired from the design of the standard template library in C++ and its ability to allow the user to use custom types with standard algorithms and vice versa.

1 Introduction

For some years now, there have been efforts to add a linear algebra library in the C++ standard library. One of the motivations has been to provide common vocabulary types to allow different code bases to use each other without having to make intermediate copies. The other factor which is generally important is to recognize the sparsity in the data structures and . The way The focus of those papers mainly deals with the types and algorithms which would be useful in a standardized linear algebra library.

The named functions themselves are often not a problem because one almost always, one can choose the correct algorithms based on the context. Furthermore, choosing the correct algorithm is straightforward as well, often as simple as

```
std::some_algorithm(custom_execution_policy, arguments...);
custom_lib::custom_algorithm(custom_execution_policy, arguments...);
```

Listing 1: Calling a Custom Function

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++?

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 What this paper is not?

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 them to interact with one another. As a result, the types used in this paper are minimal in order to highlight the requirements that we wish to impose on the types.

2 Some Traits

```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 2: Supporting Traits

3 Engine-Aware Types

```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 3: 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 4: 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 5: Engine and Engine-Based Operator

5 Some Musings on Design Decisions

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

```

using sce = std::math::serial_cpu_engine;
using pce =custom_lib::parallel_cpu_engine;
using stdvec = std::math::vector;
using custvec = another_lib::vector;

auto const ov1 = stdvec<sce>(arg...);
auto const ov2 = custvec<sce>(arg...);

auto const view_3 = ov1 + ov2;

auto const view_4 = ov1.change_engine<pce>();
auto const ov5 = (custvec<sce>(args...)).change_engine<pce>();

auto const view_6 = view_4 + ov5;

auto const ov7 = stdvec<pce>(args...) + ov1;
```

Listing 6: Example of How the Code Might Look

7 Acknowledgement

A lot of ideas have been gathered by participating in the SG15 Linear Algebra SIG discussions, reading the codebases like Eigen, MTL, Blaze and others etc. While I have tried to credit the ideas wherever necessary, if you find your ideas being used in the code without credit, I apologize in advance. Also, I would be grateful if you could bring them to my attention so that I could write them as such.

The idea of offloading the computations to an engine is inspired from [**GuyDavidson2018**]. The idea to use execution policy is inspired from MTL .

is?