P1635R0 : A Design for an Inter-Operable and Customizable Linear Algebra Library

Jayesh Badwaik

June 25, 2019

A linear algebra library is currently being considered for standardization in the C++ standard library through [1]. In this paper, we show how a well-designed set of requirements imposed on the types and the functions allow the users to write their linear algebra library that allows interoperability between libraries while preserving the objectives laid out for the library.

1 Introduction

In the past few years, there has been an effort to add a linear algebra library to the standard library. The motivations for the effort can be found in [2]. There are already quite a few papers in flight concerning the design, with the primary one being [1]. The primary idea behind the design in the paper is to provide a collection of standard linear algebra types and functions with facilities for customization for special purposes.

There is another approach which one can pursue, wherein, we ask the types implemented by the standard library and the user alike to satisfy certain requirements and then write generic code for those types. This is the idea which is primarily employed by the standard template library. In this paper, we take the second approach to the design of a linear algebra library.

Now, while the two approach do look different in principle, the practical considerations involved and the resulting design from the two approaches might end up being very similar. For example, in absence of well-designed types and functions in the standard library, the approach in this paper has a potential to create disparate and fragmented types and engines with possibly sub-par implementations. A well-designed standard library types in the spirit of [1] will go a long way towards acting as a deterrent to such implementations.

This reasoning is not very different from the reasoning today wherein we can confidently refer people to use `std::vector` for most purposes even though, technically, anyone can write their own `vector` implementation and it will work with other parts of standard library just fine.

For named functions, there is not anything significantly different about linear algebra types as to require a special attention in contrast with standard library types. The situation however changes when one begins talking about operators. Operators make the code quite readable and hence are considered quite desirable in the linear algebra community, however, this also means that specifying customization at the operator level in generic code can become quite cumbersome. For this reason, in this paper, we only devote our attention to the operator algebra between linear algebraic types. As a matter of exposition, we currently limit ourselves to only one operator, the `operator+` between vectors, but one can see that the design can very well extend to other operators as well. In the next versions of this paper, we plan to include more operators in the exposition.

The central idea of the paper is to use a configurable template parameter named as `engine` and then use that as a customization point to select correct algorithms. This also implies that instead of writing operators for specific types, we write operators for types with the corresponding engine parameter. This allows us

to remove the domain of binary operators from the namespaces of the types into the namespaces of the engine, giving users a better control over customization.

2 Some Uses Cases and Scenarios

In order to motivate the rest of the paper, we now lay out a couple of scenarios which are not generally supported by different linear algebra libraries today, but which we expect to be satisfied in the current scenario.

1. Operations Between Types of Different Libraries.

```
auto const x = a::f();  // Returns vector x of type a::vector
auto const A = b::g();  // Returns a matrix b of type b::matrix
auto const b = A * x;   // ERROR: operator* not defined!!
```

   In general, there will not be an `operator*` defined for the types `a::vector` and `b::matrix`. In fact, there is no good way to define an `operator*` in this case without invading on one of the two namespaces. A classical workaround would be to convert one of the types into a compatible type for another which might involve a copy. In this case, where a default behavior cannot be expected, we would like to design a solution wherein the user at the call site would be able to specify the which algorithm to use to carry out the `operator*`. The user should also be able to specify a custom implemented algorithm that the user has implemented.

2. Different Algorithms for the Same Operations on the Same Types

```
auto const x = a::f();  // Returns vector x of type a::vector
auto const A = a::g();  // Returns a matrix b of type a::matrix
auto const b = A * x;   // Assume that operator* is serial
```

   Assume that the `operator*` is serial in this case. But suppose that the user wants to allow the computation to be parallel (or probably offloaded to the GPU) in only some places in the code while requiring the computation to run in serial in other places. There can be quite a few scenarios where such a behavior is desirable. For example, it is possible that the operator which is expected to run in serial is already a part of a parallelly-executed code segment. In this design, we want to be able to use the same underlying storage, but carry out different algorithms on these types.

Another problem which is generally discussed in the context of modern linear algebra libraries the interaction of the `auto` keyword with the expression templates. Solutions to these issues do exists, for example, it has been suggested that one should detect if a temporary is being consumed by the operator and, if that is the case, then one should return an owning value instead of an expression or a view. We show how this can also be taken care of in the framework of operators which we present here.

3 Some Traits

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

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

Listing 1: Engine-Aware Trait

```cpp
// 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... Tp>
struct uses_engine ;
```

Listing 2: Uses-Engine Trait

```cpp
// 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 <typename... Tp>
struct is_owning_type ;

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

Listing 3: Supporting Traits

4 Engine Compatible Types

We next describe the engine compatible types. We list out the requirements which the types should satisfy in order to be interoperable with other similarly designed types. We describe the types in two flavors: an engine compatible owning type and a engine compatible non-owning type. The owning types are the types which have their own storage and hence cannot be cheaply copied. While the non-owning types are types which do not have their own storage, and instead point to other types or an expression of types. Examples are vector views and expression templates .

4.1 An Engine Compatible Non-Owning Type

These kinds of types consists of views and expression templates, which do not have any storage of their own. These objects should only be constructible from non-temporary objects (hence the requirement of references and const-references in the constructor).

```cpp
  template<typename E, typename OT>
  class egnot {
  public:
    using engine_type = E;
    using is_owning_type = std::false_type;

  public:
    template <typename NE>
    auto change_engine() const -> egnot<NE, OT>;

    egnot(OT & ot);
    egnot(OT const& it);
  };
```

Listing 4: An Engine Compatible Non-Owning Type

As we can see, the type requires us to export an `engine_type` and specify that it is not an owning type by specifying `is_owning_type` as `std::false_type`. Given a non-owning object with an engine parameter `E`, one should be able to construct another non-owning object from it, which points to the same

underlying storage in an identical manner but which exports a different engine type `NE`. This is the facility that should be provided by the `change_engine` method.

4.2 A Engine Compatible Non-Owning Type

These kind of types actually own the storage. Just as with non-owning types, they export the engine type, they assert that they are an owning type. Unlike the non-owning types, owning types need multiple different versions `change_engine` method. When operating on a temporary, the method should return another owning type which exports the updated engine parameters. When operating on a non-temporary instead, it returns a non-owning type referring to the underlying owning type with proper const qualifications.

```
1   template<typename E>
2   class egot {
3   public:
4     using engine_type = E;
5     using is_owning_type = std::true_type;
6
7   public:
8     template <typename NE>
9     auto change_engine() && -> egot<NE>;
10
11    template <typename NE>
12    auto change_engine() & -> egnot<NE, egot<E>>;
13
14    template <typename NE>
15    auto change_engine() const& -> egnot<NE, egot<E> const>;
16  };
```

Listing 5: An Engine Compatible Owning Type

5 Engine and Engine-Based Operators

Now that we have described how the types in this paradigms should look, we now describe how the `operator+` can be implemented using these principles. All of the code implemented below is in the same namespace. We first define an engine named `serial_cpu_engine`. As the name suggest, the storage and the computations are both done on the CPU and the algorithm is serial. The engine above is stateless, but it doesn't have to be, and customizations can be made based on the underlying hardware.

```
1   struct serial_cpu_engine{};
```

Listing 6: A Serial CPU Engine

Next we define the `addition_traits`. The purpose of `addition_traits` is to determine the return type of the expression. Here is the place where one can determine whether to return an owning type (by value) or a non-owning type (a view or an expression). This can be achieved, for example, by checking if either of `T` and `U` satisfies the `std::is_rvalue_reference<T>` trait and the `is_consuming_owning_type` trait mentioned before.

```
1 template <typename T, typename U>
2 struct addition_traits {
3   using result_type = X ; // X can be determined using any logic required
4 };
```

Listing 7: Addition Engine Traits

Finally, we describe the engine based operator for the above engine. Given types `T` and `U`, the operator described below can be found by ADL by the virtue of being in the same namespace as `serial_cpu_engine`. The way we can ensure that this is the best match is by ensuring that there are no explicit operators

defined for T and U. The similar operators in the same namespace but attached to different engines are ruled out by the condition in the template parameters.

```
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&&>;
```

Listing 8: Engine-Based Operator

In case one wants the operator to inherit the behavior of an operator attached to a different engine, then one can implement the above operator as below, where we can see that the lifetime issues take care of themselves.

```
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 9: Inheriting Behavior of Operator from Another Engine

6 Example Code

In this example, we show how the code works with a combination of vectors from different libraries and of different structure. `sce` is an engine which has an associated operator `operator+` which computes addition of two vectors stored on CPU in a serial manner. `pce` is an engine which has an associated operator `operator+` which computes the addition in a multi-threaded environment. `std::math::vector` is the vector implemented in `std::math` namespace which satisfies the requirements placed in the paper on the owning types and `another_lib::vector` is another vector implemented in `another_lib` namespace with similar qualities.

6.1 Serial Code for Multiplication of Two Vectors

```
  using sce = std::math::sce;
  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 ov4 = stdvec<sce>(args...) + ov1;
```

Listing 10: Serial Code for Multiplication of Two Vectors

In the above code, because `ov1` and `ov2` are not temporaries, the addition will return a view or an expression. But then in the second operation, we will get a return by a value.

6.2 Parallel Code for Multiplication of Two Vectors

Assume that the function `f()` returns an object of type owning type by value but exports engine `sce` in its engine type.

```
using sce = std::math::sce;
using pce = std::math::pce;
using stdvec = std::math::vector;
using custvec = another_lib::vector;

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

auto const view_3 = ov1.change_engine<pce>() + ov2;
auto const ov4 = f().change_engine<pce>() + ov2.change_engine<pce>();
```

Listing 11: Parallel Code for Multiplication of Two Vectors

7 Conclusions and Musings

So, in this paper, we have described design decisions which make the code shown in the two examples work. The paper is still in its very early design phase. While we can show that the proof of concept works, the question on whether it remains a good design with manageable complexity still remains. While we believe that the complexity is manageable, it has not yet been shown in a proof of concept code for the same. Conditional on the response to the design here, we will invest more time in writing a more details examples which take into account other factors such as sparsity of the matrix, static dimensions of the matrix, GPGPU storage among other things.

8 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. I would also like to thank various people in Slack C++ workspace for answering a lot of my questions about C++ in relation to the above paper.

9 A Comment about the Code

The code for the paper is being currently developed at https://github.com/liblac/proof-of-concept.

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.