

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

Jayesh Badwaik

SG14 Linear Algebra SIG

What this paper is trying to be?

- Not a library proposal.
- Rather a collection of concepts/techniques which enable the library writers to write inter-operable libraries.

Linear Algebra

Why think about Inter-Operability between Linear Algebra Libraries?

- Mixing of **vector** in current code.
- Standard will always move slowly with regards to current practice.
- There will be other linear algebra libraries which will do things standard cannot do.
- Some libraries will use these **other** libraries.
- These other libraries **might not** use the vocabulary-like types.
- Well-designed linear algebra types in STL will prevent drive-by implementations of library, but custom library implementers might still go off and do their own thing.

Impacts of Non-Interoperability on User Code

- Run-Time Performance

If I use a library which uses standard linear algebra library and another library which uses another standard library, then, there will always be a copy between the two.

This will result in:

- ▶ Reimplementing the library by performance-conscious people.
- ▶ Lack of performance by copies permeating through the code base.

- Ecosystem Effects

- ▶ There is a possibility (possibly small) of the standard linear algebra ecosystem falling enough behind that people stop using it, and use other libraries instead.

Defining the Problem Space

Restriction to Operator Algebra

```
1 auto const a = alpha::func(); // std::matrix
2 auto const b = beta::func(); // b:matrix:
3 auto const c = beta::other_func(); // b::vector:
4 auto const d = p::compute_eigenvalues(a,b);
5 auto const e = p::gaussian_elimination(a,c);
```

What do we not have today that prevents this from happening?

- $a+b$, $a*b$, $a[i] * b[j]$ (vectors)
- Pessimization of operators in face of structured matrices, sparse matrices etc.

Defining the Problem Space : Compatibility With?

Heterogenous Computing

Talk a little heterogenous computing.

```
1 auto const a = alpha::func(); // std::matrix
2 auto const c = beta::other_func(); // b::vector:
3 // Compiler Error?
4 auto const e = gpu::gaussian_elimination(a,c);
5 // Copy and Compute (What if a and c are already on gpu?)
6 auto const f = gpu::gaussian_elimination(a,c);
```

Defining the Problem Space : Compatibility With?

Expression Templates

```
1 auto const a = alpha::func(); // std::matrix
2 auto const t1 = beta::transpose(a); // ET?
3 auto const t2 = beta::transpose(t1*alpha::g()); // Return By V
```

Tools to Solve the Problem

Three Techniques

- Non-owning Types aka Expression Templates / Views
- Engine-Aware Types
- Engines and Engine-Associated Operator

1. Non-owning Types aka ET/Views

Intent : Not Wording

A type is said to be a non-owning type if an object of this type needs an independent object to exist separately for it to be well-behaved.

```
1 auto a = alpha::func(); // Returned by Value
2 auto view= alpha::transpose(a); // A Non-owning Type
```

So, all such types should export

```
using is_owning_type=std::false_type;
```

1. Non-owning Types aka ET/Views

Intent : Not Wording

Assume `alpha::func()` returns `a::matrix`. Now, there exists a proof of concept which does the following

```
1 // returns by value
2 auto const a = alpha::tp(alpha::func());
3 // returns a view
4 auto const b = alpha::tp(a);
5 // returns a view
6 auto const c = alpha::tp(alpha::tp(b));
```

2. Engine-Aware Types

Engine-Aware Non-Owning Types

```
1  template<typename E, typename OT>
2      class view {
3      public:
4          using engine_type = E;
5          using is_owning_type = std::false_type;
6
7      public:
8          template <typename NE>
9              auto change_engine() const -> view<NE, OT>;
10
11     private:
12         OT* ot_;
13     };
```

2. Engine-Aware Types

Engine-Aware Owning Types

```
1  template<typename E>
2  class owning {
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() && -> owning<NE>;
10
11     template <typename NE>
12     auto change_engine() & -> view<NE, owning<E>>;
13
14     template <typename NE>
15     auto change_engine() const& -> view<NE, owning<E> const>;
16 };
```

3. Engine and Engine-Associated Operators

`std::math` namespace

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

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

Listing 1: Addition Engine Traits

```
1 template <
2 typename T,
3 typename U,
4 typename = std::enable_if<
5 std::math::uses_engine_v<serial_cpu_engine, T, U>>>
6 auto operator+(T&& t, U&& u) -> addition_traits_r<T&&, U&&>;
```

Listing 2: Engine-Based Operator

3. Engine and Engine-Associated Operators

`std::math` namespace

```
1  using sce = std::math::serial_cpu_engine;
2  using stdvec = std::math::vector;
3  using custvec = another_lib::vector;
4
5  auto const ov1 = stdvec<sce>(arg...);
6  auto const ov2 = custvec<sce>(arg...);
7
8  auto const view_3 = ov1 + ov2;
9
10 auto const ov4 = stdvec<sce>(args...) + ov1;
```

Listing 3: Serial Code for Multiplication of Two Vectors

- Discuss the ADL-based lookup and how we can ensure that it does not go rogue.
- DO NOT DECLARE OPERATORS FOR TYPES, ONLY FOR ENGINES.

3. Engine and Engine-Associated Operators

`std::math` namespace

```
1  using sce = std::math::sce;
2  using pce = custom::pce;
3  using stdvec = std::math::vector;
4  using custvec = another_lib::vector;
5
6  auto const ov1 = stdvec<sce>(arg...);
7  auto const ov2 = custvec<pce>(arg...);
8
9  auto const view_3 = ov1.change_engine<pce>() + ov2;
10
11 auto const ov4 = f().change_engine<pce>()
12                 + ov2.change_engine<pce>();
```

Listing 4: Parallel Code for Multiplication of Two Vectors

3. Engine and Engine-Associated Operators

`std::math` namespace

```
1  template <
2  typename T,
3  typename U,
4  typename = std::enable_if<
5      std::math::uses_engine_v<serial_cpu_engine, T, U>>>
6  auto operator+(T&& t, U&& u) -> addition_traits_r<T&&, U&&>
7  {
8      return t.change_engine<some_other>()
9          + u.change_engine<some_other>();
10 }
```

Listing 5: Inheriting Behavior of Operator from Another Engine

Some Additional Points

- Less pressure on standard library to have all functions.
- Graceful migration of functions from one library to another.
- A weak standard library will spawn a creation of lot of "engines" and types. But with this design, a growth of stronger standard library can potentially reduce the problem.

So, we can (potentially) solve our problems this way.

- Proof of concept at <https://github.com/liblac/proof-of-concept>
- Questions/Polls: Does SG14 consider this a viable exploration of design for linear algebra library? (Implying that current design be modified in accordance with these ideas.)