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.
- Why make an effort to standardize this?
  - Multiple ways libraries can interoperate.
  - One of the ways is this, if standard does it and if it is a good enough design, then other libraries might follow.

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

```
auto const a = alpha::func(); // std::matrix
auto const b = beta::func(); // b:matrix:
auto const c = beta::other_func(); // b::vector:
auto const d = p::compute_eigenvalues(a,b);
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.

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

Defining the Problem Space : Compatibility With?

Expression Templates

```
auto const a = alpha::func(); // std::matrix
auto const t1 = beta::transpose(a); // ET?
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.

```
auto a = alpha::func(); // Returned by Value
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

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

```
// returns by value
auto const a = alpha::tp(alpha::func());
// returns a view
auto const b = alpha::tp(a);
// returns a view
auto const c = alpha::tp(alpha::tp(b));
```

2. Engine-Aware Types

Engine-Aware Non-Owning Types

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

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

 private:
   OT* ot_;
 };
```

2. Engine-Aware Types

Engine-Aware Owning Types

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

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

  template <typename NE>
  auto change_engine() & -> view<NE, owning<E>>;

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

3. Engine and Engine-Associated Operators

`std::math` namespace

```
  struct serial_cpu_engine{};
```

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

Listing 1: Addition Engine Traits

```
template <
typename T,
typename U,
typename = std::enable_if<
std::math::uses_engine_v<serial_cpu_engine, T, U>>>
auto operator+(T&& t, U&& u) -> addition_traits_r<T&&, U&&>;
// Can also SFINAE on something like `is_linear_algebra_type`
```

Listing 2: Engine-Based Operator

3. Engine and Engine-Associated Operators

std::math namespace

```cpp
using sce = std::math::serial_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 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

```cpp
using sce = std::math::sce;
using pce = custom::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 4: Parallel Code for Multiplication of Two Vectors

```
template <
typename T,
typename U,
typename = std::enable_if<
  std::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: 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.
- Question of well-established practice: None of the Linear Algebra library has used this practice before, but STL is based on this idea. So, C++ wise, the methods are pretty established.
- Need to check compatibility of these techniques with linear algebra.

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.)