Portable C++ for Modern, Heterogenous Mixed-Precision Methods

Neil Lindquist

SIAM CSE23

March 1, 2023

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Modern Numerical Methods

Precision

- Double
- Single
- Half
- Bfloat16
- Integer?
- Posits?

Hardware

- CPU
- GPUs
  - NVIDIA
  - AMD
  - Intel

Algorithm variants

- GMRES vs FGMRES
- Real vs Complex

Mixed Precision Experiments

- Many combinations of precisions to test
  - My GMRES work: 14 combinations in paper
    - Only single, double
  - GMRES-IR5[1]: 112 “meaningful” combinations
    - Bfloat16, half, single, double, quad

 [1]Amestoy et al., “Five-Precision GMRES-based iterative refinement” 2021

Implementing

- Templates allow generic implementation
  - Portability layers, e.g., Kokkos
- Ideally: retain performance of separate codes
  - E.g., avoid “conversions” in uniform-precision
- Readable and portable

→ C++ features

Avoiding Duplicate Storage

- Sometimes need array in high and low prec.
  - In uniform-precision, just use reference
- Easy for types with reference semantics
  - E.g., Kokkos View
  - Just need different constructor

Avoiding Duplicate Storage

```
using hi_t = ...; using lo_t = ...;

Kokkos::View<hi_t, ...> w_hi = ...;
Kokkos::View<lo_t, ...> w_lo;
if (std::is_same_v<hi_t, lo_t>) {
  w_lo = w_hi;
} else {
  w_lo = Kokkos::View<lo_t, ...>(w_hi.extent(0));
}
```

Error when hi_t≠lo_t

Compile time if statement

- `if constexpr (condition)`
  - Since C++17
- Condition must be `constexpr`
- No template substitution for untaken branches
  - Can use type-specific functions

Avoiding Duplicate Storage

```
using hi_t = ...; using lo_t = ...;

Kokkos::View<hi_t, ...> w_hi = ...;
Kokkos::View<lo_t, ...> w_lo;
if constexpr (std::is_same_v<hi_t, lo_t>) {
  w_lo = w_hi;
} else {
  w_lo = Kokkos::View<lo_t, ...>(w_hi .extent(0));
}
```

Compile time if statement

```
...
if constexpr (is_complex<T>) {
  phase = val / std::abs(val);
} else {
  phase = val >= T(0) ? 1 : -1;
}
...
```

Avoiding Duplicate Storage

- Sometimes need array in high and low prec.
  - In uniform-precision, just use reference
- Harder for types with value semantics
  - Want to avoid copy operator

    → conditionally make it a reference

Avoiding Duplicate Storage

```
using hi_t = ...; using lo_t = ...;
constexpr bool same_type_p = std::is_same_v<hi_t, lo_t>;
using vec_lo_t = std::conditional_t<same_type_p,
                                    Vect<hi_t>&, Vect<lo_t>>;
Vect<hi_t> w_hi = ...;
vec_lo_t w_lo;
if constexpr (same_type_p){
   w_lo = w_hi;
} else {
   w_lo = Vect<lo_t>(w_hi.n());
}
```

Error when hi_t=lo_t

Move to helper function

Conditional types for real vs complex

- In rank revealing factorizations
  - SVD gives real-valued singular values
  - Pivoted QR gives complex-valued estimates
  - `std::conditional_t` allows same postprocessing code

Optional arguments

- Optional keyword arguments
  - Easily add new functionality w/out new routines
  - Users can focus on relevant args
  - Name-value pairs help clarity
- C++ only supports this via structs

Optional Arguments in SLATE

```
using namespace slate;

Options opts;
opts[Option::Target]          = Target::Devices;
opts[Option::Lookahead]       = 2;
opts[Option::PivotThreshold] = 0.1;
lu_solve(A, b, opts);
```

Optional Arguments in SLATE

- C++ list initialization give better API

```
using namespace slate;

lu_solve(A, b,
         {{Option::Target, Target::Devices},
          {Option::Lookahead, 2},
          {Option::PivotThreshold, 0.1}});
```

Conclusions

- Combinatorial explosions of implementations
  - Precisions, accelerators, algorithmic variants
- Modern C++ features help
  - Reduce code duplication
  - Retain performance

Avoiding Duplicate Storage

```
template<class T, bool useFirst, class U, class V>
T make_from_selector(U arg1, V arg2) {
    // U, V are std::tuple<...>
    if constexpr (useFirst) {
        return std::make_from_tuple<T>(arg1);
    } else {
        return std::make_from_tuple<T>(arg2);
    }
}
```

Avoiding Duplicate Storage

```
using hi_t = ...; using lo_t = ...;
constexpr bool same_type_p = std::is_same_v<hi_t, lo_t>;
using vec_lo_t = std::conditional_t<same_type_p,
                                    Vect<hi_t>&, Vect<lo_t>>;
Vect<hi_t> w_hi = ...;
vec_lo_t w_lo = make_from_selector
                          <vec_lo_t, same_type_p>
                          (std::tuple(w), std::tuple(n));
```

Caveats

- Too many templates
  - Increases compile times
  - Increases mental burden
  - Results in poor compiler error messages
- Can’t easily instantiate many combinations of template parameters
  - Often results in larger headers/repeated compilation