Exploiting Modern C++ for Portable Parallel Programming in Lattice QCD Applications

Alexei Strelchenko

Data Science, Simulation and Learning Division @ Fermilab

July 31, 2023

Plan

- C++17 std::par algorithms
- Hierarchy of views in C++20/C++23
- A few notes on accelerator programming:
  - Utilizing C++23 cartesian products for the execution domain
  - C++17 Polymorphic Memory Resources (PMR) for memory buffer management
- Staggered stencil example
- Backup slides: C++20 Concepts

The standard library algorithms

- The quintessence of the C++ heterogeneous programming is the standard library algorithms, common programming patterns that can be used as building blocks for solving more complex problems.
- They are provided by <*algorithm*> header
- Most of the algorithms from std library operate only on *iterators*, and not on the *containers*, the feature that makes them truly generic constructions

The standard library algorithms

Examples of SL algorithms:

Iteration and transform:

std::for_each, std::for_each_n, std::transform,
std::transform_inclusive/exclusive_scan,
std::transform_reduce

Reduction:

std::reduce, std::transform_reduce
std::inclusive/exclusive_scan,
std::count, std::count_if,

Searching:

std::search, std::search_n,
std::find, std::find_if/if_not,
std::find_end, std::find_first_of

Memory managment:

std::copy, std::copy_n, std::copy_if,
std::move, std::fill, std::fill_n,
std::generate, std::generate_n

Removing and replacing elements:

std::remove, std::remove_if,
std::replace, std::replace_if,
std::unique, std::unique_copy,

Reordering elements:

std::sort, std::stable_sort,
std::partition, std::stable_partition,
std::merge, std::inplace_merge

C++17 std::execution::par

- C++17 introduced higher-level parallelism features that allow to request parallelization of (most) SL algos
- C++17 offers the execution policy parameter that is available for these algos:
  - `std::execution::seq` to execute an algo sequential
  - `std::execution::par` to execute an algo in parallel
  - `std::execution::unseq` to execute an algo with vector instructions
  - `std::execution::par_unseq` to execute an algo in parallel with vector instructions

The iterator-based algo (example)

Evolution of the for loop:

```
#include <algorithm>
#include <execution>
#include <ranges>

std::vector<float> x(N), y(N);

auto l = std::ranges::views::iota(0, N);

std::for_each(std::execution::par,
              std::ranges::begin(l),
              std::ranges::end(l),
              [x_ptr = x.data(), y_ptr = y.data()](const int& i){
                //do something with x_ptr and y_ptr
              });
```

The iterator-based algo (caveates)

- nvc++ uses UVM for automatic CPU-GPU data transfers
- only data dynamically allocated in CPU code compiled by nvc++ can be auto-managed.
- memory allocated in compute kernels is GPU-exclusive and unmanaged.
- CPU and GPU stack memory and global objects can't be auto-managed.
- data allocated outside nvc++-compiled units, even on CPU heap, isn't managed.
- pointers and objects within std::par algo invocations must refer to managed CPU heap data.
- dereferencing CPU stack pointers or global objects in GPU code leads to memory violations.

The iterator-based algo (example)

Evolution of the for loop:

```
#include <algorithm>
#include <execution>
#include <ranges>

std::vector<float> x(N), y(N);

auto l = std::ranges::views::iota(0, N);

std::for_each(std::execution::par,
              std::ranges::begin(l),
              std::ranges::end(l),
              [&x = x, &y = y](const int& i){
                //Illegal memory access!
              });
```

The iterator-based algo (example)

Evolution of the for loop:

```
#include <algorithm>
#include <execution>
#include <ranges>
#include <span>

std::vector<float> x(N), y(N);

auto l = std::ranges::views::iota(0, N);

std::for_each(std::execution::par,
              std::ranges::begin(l),
              std::ranges::end(l),
              [x_view = std::span{x}, y_view = std::span{y}](const int& i){
                //Okay, do something x_view, y_view
              });
```

C++20 Hierarchy of views

- `std::span` :
  - is a lightweight, non-owning reference to a contiguous sequence, or a span, of objects of a given type `T` in memory.
  - is only a *view* into a sequence: it does not control the lifetime of the objects it refers to, nor does it allocate or deallocate memory.

```
//Defined in header <span>
template<
    class T,
    std::size_t Extent = std::dynamic_extent
> class span;
```

C++20 Hierarchy of views (cont.)

- `std::span::subspan` :
  - creates a `std::span` that only refers to a portion of the elements of the original `std::span`, specified by starting index and length

```
//Defined in header <span>
constexpr std::span<element_type, std::dynamic_extent>
subspan( size_type Offset,
         size_type Count = std::dynamic_extent ) const;
```

Example 1

Field class with views:

```
template <GenericContainerTp container_tp, typename Arg>
class Field{
  private:
    const Arg arg;//copy of the arguments

    container_tp v;//could be std::vector, std::span (or subspan)

  public:

    decltype(auto) View() {
      // ...
      return Field(std::span{v}, arg);
    }

    decltype(auto) ParityView(const FieldParity parity ) {
      // define parity args etc...
      return Field(std::span{v}.subspan(parity_offset, parity_length), parity_arg);
    }
};
```

C++23 multi-dimensional view

- `std::mdspan` :
  - is a non-owning view into a contiguous sequence of objects that reinterprets it as a *multidimensional* array
  - (accepted) proposal can be found https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html
  - the reference implementation is due to Kokkos dev team https://github.com/kokkos/mdspan

```
//Defined in header <mdspan>
template<
    class T,
    class Extents,
    class LayoutPolicy   = std::layout_right,
    class AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

Example 2

Fine-grained accessors:

```
template<bool is_constant, size_t... dofs>
inline decltype(auto) mdaccessor(const std::array<size_t, (Ndim + sizeof...(dofs))> &strides) const {

using dext = stdex::dynamic_extent;

using Map = stdex::layout_stride::mapping<stdex::extents<size_t, dext, dext, dext, dext, dofs...>>;

using Extents= stdex::extents<size_t, dext, dext, dext, dext, dofs...>;

  if constexpr (is_constant){
    return stdex::mdspan<const data_tp, Extents, stdex::layout_stride>{
                    v.data(), Map{Extents{X[0], X[1], X[2], X[3]}, strides}} ;
  }
}

template<bool is_constant>  auto Accessor() const {
  if constexpr (Arg::type == FieldType::VectorFieldType) {
    if constexpr (nParity == 2) {
      return mdaccessor<is_constant, nColor, nColor, nDir, nParity>(strides);
    }
  }
}
```

C++20 Hierarchy of views (cont.)

- `std::ranges::ref_view` :
  - is a view of the elements of some other range
  - wraps a reference to that range
- `std::ranges::reverse_view` :
  - represents a view of underlying sequence with reversed order.

```
#include <ranges>

std::array<int, N> x{0,1,2,3};

std::ranges::ref_view y{x};//y=(0,1,2,3)

std::ranges::reverse_view z{x};//z=(3,2,1,0)
```

Example 3

Stencil computing with views:

```
void apply(SpinorView auto &out, const SpinorView auto &in, const auto idx, const Parity p) {

  std::array X = convert_coords(idx);
  //
  std::ranges::ref_view X_view{X};
  // or
  //auto X_view = X | std::views::all;

  auto out = FieldAccessor{out};
  const auto in = FieldAccessor<is_constant>{in};
  //
  auto res = compute_stencil(in, parity, X_view);

#pragma unroll
  for (int c = 0; c < Ncolor; c++){
    out(X_view,c) = res(c);
  }
}
```

C++23 cartesian product

- `std::view::cartesian_product` :
  - is a range adaptor that produces a view of tuples calculated by the n-ary cartesian product of the provided ranges
  - proposal can be found https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2374r4.html

```
//Defined in header <cartesian_product.hpp>
      auto X = std::views::iota(0, Nx);
      auto Y = std::views::iota(0, Ny);
      auto Z = std::views::iota(0, Nz);
      auto T = std::views::iota(0, Nt);
//T is the slowest index, X is the fastest
      auto ids = std::views::cartesian_product(T, Z, Y, X);
```

Example 4

Kernel launcher:

```
void launch(SpinorView auto &o, const SpinorView auto &i, const FieldParity p, const auto &idx) {
   //...
   std::for_each(std::execution::par_unseq,
                 ids.begin(),
                 ids.end(),
                 DslashKernel);
}

void operator()(Spinor auto &o, const Spinor auto &i, const FieldParity p){
   using spinor_tp    = typename std::remove_cvref_t<decltype(i)>;
   using container_tp = spinor_tp::container_tp;
   //Setup exe domain
   const auto [Nx, Ny, Nz, Nt] = o.GetCBDims(); //Get CB dimensions

   auto X = std::views::iota(0, Nx);
   //...
   auto T = std::views::iota(0, Nt);

   auto idx = std::views::cartesian_product(T, Z, Y, X);

   if constexpr (is_allocator_aware_type<container_tp>) {
     auto&& o_view       = o.View();
     const auto&& i_view = i.View();

     launch(o_view, i_view, p, idx);
   }
}
```

C++17 Polymorphic Memory Resource

- C++17 PMR provided a "polymorphic" interface to memory management:
  - allow you to allocate memory in a way that is independent of the underlying allocator
  - useful for applications that need to control how memory is allocated, or need to be able to run on multiple platforms with different allocators
  - includes polymorphic allocator-aware versions of several SL containers, e.g.,
    `std::pmr::vector`, `std::pmr::string`, and
    `std::pmr::unordered_map`

C++17 Polymorphic Memory Resource

- When is it useful?
  - Performance improvement
    - can reduce allocation overhead and improve cache utilization
  - Memory usage control
    - has more control over how memory is allocated (deallocated).
  - Memory tracking and debugging
    - can be used to implement custom memory resources that track memory usage for debugging
  - Custom memory pools
    - allows the creation of memory pools with specific allocation strategies, which can be useful for cases where one has objects of the same size being frequently created (destroyed), i.e., can help reduce fragmentation and the overhead of frequent allocations.

C++17 Polymorphic Memory Resource

- Memory resources:
  - `std::pmr::memory_resource`
    - the base class for all memory resources in the PMR library.
  - `std::pmr::monotonic_buffer_resource`
    - a memory resource that allocates memory from a single buffer
  - `std::pmr::unsynchronized_pool_resource`
    - a memory resource that allocates memory from a pool of pre-allocated blocks
  - `std::pmr::synchronized_pool_resource`
    - same as above but with synchronization for concurrent access.

Example 5

PMR in action:

```
#include <memory_resource>

// ptr is a pre-allocated memory pool
// bytes = 2*N*sizeof(float);

std::pmr::monotonic_buffer_resource pmr_buff(ptr, bytes);

std::pmr::vector<float> x(N, &pmr_buff);//populate N*sizeof(float) bytes
std::pmr::vector<float> y(N, &pmr_buff);//populate next N*sizeof(float) bytes

auto l = std::ranges::views::iota(0, N);

std::for_each(std::execution::par,
              std::ranges::begin(l),
              std::ranges::end(l),
              [x_view = std::span{x}, y_view = std::span{y}](const int& i){
                //Okay, do something x_view, y_view
              });
```

Example 5

PMR in action (allocation in stack):

```
#include <memory_resource>

std::array<std::bytes, 2*N> stack_buff;

std::pmr::monotonic_buffer_resource pmr_buff(stack_buff.data(), stack_buff.size(),
                                             std::pmr::null_memory_resource());

std::pmr::vector<float> x(N, &pmr_buff);//populate N*sizeof(float) bytes
std::pmr::vector<float> y(N, &pmr_buff);//populate next N*sizeof(float) bytes
```

Example 5 (cont.)

PMR in action:

```
std::pmr::vector<float> x(N, &pmr_buff);//populate N*sizeof(float) bytes
std::pmr::vector<float> y(N, &pmr_buff);//populate next N*sizeof(float) bytes

// ... do something with x, y
x.resize(0);//destroy x container
y.resize(0);//destroy y container

// reset pmr buffer:
pmr_buff.release();

// now create new containers:
std::pmr::vector<float> u(N, &pmr_buff);//populate N*sizeof(float) bytes
std::pmr::vector<float> v(N, &pmr_buff);//populate next N*sizeof(float) bytes
```

C++17 Polymorphic Memory Resource

- Caveat:
  - C++ standard requires that any (allocator-aware) container must be initialized
- Possible solution:
  - create a custom container with a given initialization pattern (e.g., target specific)

Example 6

Custom pmr container:

```
namespace impl {
  namespace pmr {
  template <typename T>
  class vector {
    public:
      using allocator_type  = std::pmr::polymorphic_allocator<T>;
      //
      using value_type      = typename std::allocator_traits<allocator_type>::value_type;
      using size_type       = typename std::allocator_traits<allocator_type>::size_type;
      ...
    private:
      pointer        data_;
      size_type      size_;
      allocator_type alloc_;
  };
}//end of pmr
}//end of impl
```

Example 6

Custom pmr container (cont.):

```
namespace impl {
  namespace pmr {
  template <typename T>
  class vector {
    public:
      ...
      explicit vector(size_type n, std::pmr::memory_resource* mr) : data_(nullptr),
                                                                    size_(n),
                                                                    alloc_(mr) {
              data_ = alloc_.allocate(n);
              //
              if ( init_pmr_space == TargetMemorySpace::Device ) {
                std::fill(std::execution::par_unseq,
                          this->begin(),
                          this->end(),
                          zero_);

              }
  };
}//end of pmr
}//end of impl
```

Example 7

Container Zoo:

```
std::vector<float>        x(N);
std::pmr::vector<float>  y(N, &pmr_buff);
impl::pmr::vector<float> z(N, &pmr_buff);

static_assert( std::is_same_v<decltype(x), decltype(y)> == false);
static_assert( std::is_same_v<decltype(x), decltype(z)> == false);
static_assert( std::is_same_v<decltype(y), decltype(z)> == false);

auto u = std::span{x};
auto v = std::span{y};
auto w = std::span{z};

static_assert( std::is_same_v<decltype(v), decltype(u)> == true);
static_assert( std::is_same_v<decltype(v), decltype(w)> == true);
static_assert( std::is_same_v<decltype(u), decltype(w)> == true);
```

Stencil tests (WIP)

- Platform :
  - nvc++ compiler ver 23.5
  - NVIDIA A100 node
  - link: https://github.com/alexstrel/dataparallel-lqft-cpp2x
- 2d Schwinger model stencil (2048×2048):
  - init launch : 0.022sec.
  - pmr buffer reuse: 0.002sec.
  - msrc init launch ($N = 8$) : 0.15sec.
  - msrc pmr buffer reuse ($N = 8$): 0.019sec.
- 4d staggered stencil ($L = 16$, $T = 16$):
  - init launch : 0.0011sec.
  - pmr buffer reuse: 0.0005sec.

Conclusion

- `std::execution::par` :
  - express parallel algorithm with standard language
  - plain C++ code
  - memory operation via UVM
- limitations :
  - does not specify launch parameters
  - no async versions
- beyond C++20 :
  - range-based parallel algos
  - support for async. execution (sender/receiver)

Beyond C++20

- nvc++ experimental features (require *–experimental-stdpar* flag)

Table 1. Experimental features information

Feature	Recommended	Limited support	Standard proposal	Other notes
Multi-dimensional spans (mdspan)	C++23	C++17	P0009	https://github.com/NVIDIA/libcudacxx
Slices of multi-dimensional spans (submdspan)	C++23	C++17	P2630	https://github.com/NVIDIA/libcudacxx
Multi-dimensional arrays (mdarray)	C++23	C++17	P1684	https://github.com/kokkos/mdspan
Senders and receivers	C++23	C++20	P2300	https://github.com/NVIDIA/stdexec
Linear algebra	C++23	C++17	P1673	https://github.com/kokkos/stdblas

C++20 concepts

- C++20 concepts:
  - allow to specify constraints on template arguments at *compile* time
  - can be used to select the most appropriate function overloads and template specializations
  - (a lot of) built-in concepts defined in the <concepts> header (e.g. `std::copy_constructible`, `std::destructible` etc.)

```
template <typename T>
concept Complex = std::is_floating_point_v<typename T::value_type>
and requires {
    { std::declval<T>().real() } -> std::convertible_to<typename T::value_type>;
    { std::declval<T>().imag() } -> std::convertible_to<typename T::value_type>;
};
//
decltype(auto) foo(const auto x, auto y)                 {/*do something*/}
// same as above but now restricted to complex floating point
decltype(auto) foo(const Complex auto x, Complex auto y) {/*do something*/}
```

C++20 concepts (more examples)

```
template <typename T>
concept GenericContainer = requires{
    typename T::value_type;
    typename T::size_type;
    typename T::iterator;

    { std::declval<T>().data()  }  -> std::same_as<typename T::value_type*>;
    { std::declval<T>().size()  }  -> std::same_as<typename T::size_type  >;
    //
    { std::declval<T>().begin() }  -> std::convertible_to<typename T::iterator>;
    { std::declval<T>().end()   }  -> std::convertible_to<typename T::iterator>;
};

template <typename T>
concept GenericSpinorField = requires{
  //here container_tp stands for std::vector<>, std::span<> or std::subspan<>
  requires GenericContainer<typename T::container_tp>;

  requires (T::Nspin()    == 1ul or T::Nspin()  == 4ul);
  requires (T::Ncolor()   == 3ul);
};
```