FERMILAB-CONF-24-0628

PROCEEDINGS OF SCIENCE

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

Alexei Strelchenko[a,*]

[a] *Fermi National Accelerator Laboratory, Batavia, IL 60510, USA*

E-mail: astrel@fnal.gov

The evolution of ISO C++ standards increasingly serves the needs of scientific computing, offering potential benefits for developing portable applications. The recent revisions of C++ programming language, for instance, introduces a suite of algorithms capable of being executed on accelerators. Although this approach may not yield best performance, it can present a viable balance between code productivity and computational efficiency. In this report, we discuss the implementation of the HISQ operator utilizing a range of features from the C++17/20/23 standards and include an assessment of their performance.

The 40th International Symposium on Lattice Field Theory (Lattice 2023)
July 31st - August 4th, 2023
Fermi National Accelerator Laboratory

*Speaker

https://pos.sissa.it/

PoS(LATTICE2023)280

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

1. Introduction

One of the primary goals for exascale data-parallel programming interfaces is achieving performance portability, aiming to balance top-level performance at scale with efficient code development. While many HEP software libraries offer vendor-specific optimizations (see ,e.g., [1] and reference therein), the ideal scenario would be to establish a consistent, universal solution for this purpose. Additionally, some existing lattice QFT software, developed long ago, contain computational kernels that were written using outdated programming approaches and consequently need to be revisited.

The latest revisions in the C++ programming language had opened new opportunities for portable code development, particularly in scientific applications. Thus, with the introduction of the C++17 standard, the Standard Template Library (STL) underwent a substantial overhaul of its suite of algorithms, now updated with execution policies [2] to adapt across various computing architectures, including multi-core x86 systems and GPUs. Moreover, the polymorphic memory resource (PMR) feature introduced in C++17 [3] marked significant progress in memory management, offering developers enhanced flexibility and control. Subsequent iterations of the ISO C++ standard have further introduced a hierarchy of views, thereby improving data access and manipulation. In particular, C++20 introduced `std::span` [4], and C++23 added `std::mdspan` [5], facilitating convenient access to custom data objects. The C++20 standard also signified a milestone with *concepts* [6], a feature that bolstered the language's support for generic programming. Concepts provide a more expressive and precise way to define and use templates, making them particularly convenient for working with custom types and template specializations. They also play a key role in the C++ Ranges library, which represents a significant evolution in how C++ handles algorithms and iterates over sequences. The C++23 cartesian product view (`std::views::cartesian_product`) [7] enhances this further[1].

In summary, these developments showcase C++'s progress into a more modern, user-friendly language, while maintaining its core performance and flexibility. This report highlights these features through the practical implementation of the HISQ fermion operator, analysing their applicability and effectiveness.

2. Understanding the Code Design

The test code structure consists of three main components in the design of the primary (improved) staggered fermion operator application [9]. These components are: the setup of parameters through control structures, the compute kernel(s), and the kernel launchers. It is important to note the absence of a tuning infrastructure, as its implementation is not feasible within the standard C++ framework. In this section, we will present a concise overview of the key C++ features employed in the implementation of the fermion operator.

2.1 Generic programming with C++ concepts

C++20 concepts serve as a tool to specify semantic constraints on template arguments, which leads to clearer and more understandable code. They allow developers to define requirements that a template argument must meet, improving the diagnostics of template errors and making

[1] In our implementation, we relied on an adapted version of the custom implementation [8].

them more user-friendly. This results in more readable and maintainable code, as the intentions behind template parameters become more explicit and less prone to misinterpretation or misuse. For instance, a developer can define a concept to specify that a template function should only accept types that support certain operations. When a type that does not meet these requirements is used, the compiler can provide a clear and specific error message, rather than a generic template error. Listing 1 demonstrates a simplified example of the C++ concept of parity staggered fermion field objects. This concept is further utilized in defining the `D` method, which is specifically designed for parity staggered spinors: the type of its arguments must satisfy the constraints defined in the `ParitySpinor` concept.

```
template <typename T>
concept ParitySpinor = requires{
  // ...
  requires (T::Nspin()   == 1);
  requires (T::Ncolor()  == 3);
  requires (T::Nparity() == 1);
};
void D(ParitySpinor auto &y, const ParitySpinor auto &x)
{ /*apply dslash operator etc.*/  }
```

Listing 1: Concept of parity staggered spinors.

2.2 Data managment

Further, ISO C++20 introduced an enhanced approach to handling views of data, among which `std::span` is a notable addition. It represents a view of a contiguous sequence of objects, akin to a non-owning reference. The concept of views in C++20, including creating subviews (*subspan*), is a step towards safer and more efficient handling of sequences of data. Listing 2 illustrates the flexibility of `std::span` in referencing both complete objects, as seen with the `View()` method, and specific segments like parity components. This feature is highly beneficial in GPU offloading tasks, where managing lightweight objects efficiently is crucial. While `std::span` in C++20 provided a view into a single-dimensional array, `std::mdspan`, introduced in C++23, extends this concept to multidimensional arrays, which is particularly useful in handling of matrices, tensors, and other complex data layouts.

```
template <ContainerTp container_tp, typename Arg>
class Field{
  private:
    const Arg arg;//copy of the field object arguments
    container_tp v;//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);
    }
};
```

Listing 2: generic field object with views.

Like `std::span`, it is designed to be a lightweight, flexible, and safer alternative to raw pointers. Specifically, `std::mdspan` is a template that takes several parameters, including the element type, an extents type that describes the shape of the multidimensional array, as well as (optionally) a layout policy, which describes how multidimensional indices map to linear memory (such as row-major or column-major), and an accessor policy, which controls how elements are accessed, potentially allowing for more complex

```
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}} ;
  }
}
```

Listing 3: field accessor via `std::mdspan`.

operations like bounds checking or proxy references. Listing 3 gives an example of a generic field accessor, where `dof` stands for (packed) internal degrees of freedom.

2.3 Kernel execution

ISO C++17 standard brought in advanced features for parallelism, enabling the parallel execution of Standard Library algorithms. This is achieved by specifying an execution policy as the initial parameter in algorithms that are compatible with these policies. In the current framework, kernel execution is managed using the `std::for_each` algorithm only, and the compute kernels themselves are passed for the execution via lambda expressions. Within the adopted design, the implementation of compute kernels is quite generic; that is, with minor modifications ,they could also be launched using vendor-specific execution mechanisms, for example, via the CUDA triple-chevron syntax with specified execution parameters[2].

Listing 4 demonstrates how the execution domain parameter is naturally defined using Cartesian coordinates through the `std::views::cartesian_product` construct. In this case, this construct provides a view that represents the Cartesian product of four ranges. In general, `std::views::cartesian_product` enables iteration over the Cartesian product of multiple ranges without the need to materialize the product, thus offering a memory-efficient approach to handling such combinations. For instance, this makes it straightforward to iterate over a range of timeslices or a single timeslice. Note also, that the `launch` method takes views of objects rather than the objects themselves, a critical aspect for GPU-offloading tasks (although irrelevant for the fully mameory-coherent systems like NVIDIA Grace-Hopper architecture).

```
void launch(ParitySpinorView auto &out,
            const ParitySpinorView auto &in,
            auto &&transformer,
            const FieldParity p, const auto &idx) {
  // define DslashKernel lambda here ...
  std::for_each(std::execution::par_unseq,
           ids.begin(), ids.end(), DslashKernel);
}
void operator()(ParitySpinor auto &out,
                const ParitySpinor auto &in,
                auto  &&transformer,
                const FieldParity p) {
  using spinor_tp    =
    typename std::remove_cvref_t<decltype(in)>;
  using container_tp = spinor_tp::container_tp;
  //Setup exe domain
  const auto [Nx, Ny, Nz, Nt] = out.GetCBDims();
  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&& out_view      = out.View();
    const auto&& in_view = in.View();
    launch(out_view, in_view, p, idx);
  }
}
```

Listing 4: kernel execution.

Observe also `transformer` agrument in the `launch` method that we will explain in the next subsection.

2.4 Dslash-transform kernel design

Let's now examine the implementation strategies for `DslashKernel`, as highlighted in Listing 4, where it is used as the final argument in the `std::for_each` construct. The actual definition of this argument is as follows:

```
auto DslashKernel = [=, &dslash_tranform_kernel = *dslash_transform_kernel_ptr] (const auto &cartesian_prod_view) {
                      dslash_transform_kernel.template apply<dagger>(out, in, in, transformer, cartesian_prod_view, parity);
                    };
```

[2]One would need the nvc++ compiler with the `-cuda` option which supports recent versions of C++.

Here the `apply` method is represented in the following code snippet:

```
 void apply(ParitySpinorView auto &out, const ParitySpinorView auto &in, const ParitySpinorView auto &in,
            auto &&transformer, const auto cartesian_prod_view, const FieldParity parity) {
      using S = typename std::remove_cvref_t<decltype(out)>;
      // convert cartesian view into std::array  X
      auto X  = convert_coords<nDim>(cartesian_prod_view);
      // create alias view of X:
      auto X_view = X | std::views::all;
      // call (local) accessors:
      auto out = FieldAccessor<S>{out};
      const auto in = FieldAccessor<S, is_constant>{in};
      const auto aux= FieldAccessor<S, is_constant>{in};
      // apply site stencil:
      auto res = compute_parity_site_stencil<dagger>(in, parity, X_view);
      // apply post transformer (e.g., axpy operation)
      transformer(out, aux, res);
}
```

Note that in defining the `apply` method, we utilized the `ParitySpinorView` concept to ensure that its argument matches the parity spinor view type. This consideration is also important because we used capture by value in the lambda expression (and that is why `launch` method also requires view types). The transformer can be any local linear operation over the spinor fields, such as an AXPY-type operation. One can define other forms to create a fused dslash plus transform operator, akin to C++'s '`std::transform_reduce`'. This approach can be useful, for instance, in Krylov solver implementations, where fusing matrix-vector operations with linear algebra operations can enhance the overall performance of the algorithm. Generalizing this to multi-source dslash-transform operator is also straightforward.

2.5 Memory management with STL Polymorphic Memory Resource

In the final subsection, we'll discuss C++17's Polymorphic Memory Resource (PMR), a feature in the <`memory_resource`> header aimed at improving memory management, particularly in scenarios requiring custom allocation strategies.

```
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;
      //...
      explicit vector(size_type n, std::pmr::memory_resource* res) : data_(nullptr), size_(n), alloc_(res) {
        data_ = alloc_.allocate(n);//allocate memory for n elemnts of type T
        if (init_pmr_space != TargetMemorySpace::None ) {
          auto zero_ = zero<T>();
          if (init_pmr_space == TargetMemorySpace::Device) {
            std::fill(std::execution::par_unseq, this->begin(), this->end(), zero_);
          } else if (init_pmr_space == TargetMemorySpace::Host) {
            std::fill(this->begin(), this->end(), zero_);
          }
        }
    };
  }//end pmr namespace
}//end impl namespace
```

Listing 5: custom pmr vector container.

Initially, our implementation of the field class utilized STL containers, suitable for systems with Unified Virtual Memory support. However, the C++ standard mandates initializing allocator-aware containers, leading to host initialization for every object, even in scenarios where such kind of initialization is suboptimal, e.g., as with temporary objects in Krylov solvers. This limitation

prompts the need for custom, type-compatible vector containers. PMR addresses this by introducing `std::pmr::polymorphic_allocator`, a flexible, type-erased allocator compatible with both standard and custom containers and allowing these containers to use different memory allocation strategies without changing their type. An example of a custom PMR container is demonstrated in Listing 5. At the core of PMR is the `std::pmr::memory_resource` abstract class, which is evident as the second argument in the constructor of the `vector` class. The class provides a uniform interface for memory allocation, allowing objects that use polymorphic allocators to remain agnostic of the underlying memory allocation mechanism. Note that the constructor provides the option for either non-initialization or initialization on the target device using the `std::fill` algorithm. In our typical scenario, we create a custom common memory pool using PMR's memory resource functionality, and then utilize this pool for creating instances of the field class. This approach is especially convenient for objects like block spinors, as it eliminates the need for manual adjustments of memory parameters, such as offsets and the like.

3. Performance summary

We conducted test runs on the FNAL LQ cluster compute node using the NVIDIA nvc++ compiler version 23.07. Each compute node is equipped with NVIDIA A100 accelerators. The compiler options employed were '-stdpar=gpu -gpu=cc80 -gpu=managed -gpu=fma -gpu=fastmath -gpu=autocollapse -gpu=loadcache:L1 -gpu=unroll'. The source code for these tests is available in our GitHub repository [9]. It should be noted that the code adheres to the QUDA layout [10], with the lattice index as the fastest and color as the slowest, and used an even-odd decomposition. Additionally, we did not analyze multi-process implementation since the current C++17 standard does not allow for asynchronous execution. This capability is expected to be included in future revisions of the ISO C++ standard. The results for various lattice sizes are presented in Figure 1. As a reference implemntation we used HISQ dslash from the QUDA library [11]. In the chart, the blue bar represents the performance of the QUDA reference dslash, the light green bar indicates the performance of the single-source dslash, and the dark green bar shows the performance for the single component of the multi-source Dslash, with number of sources $N = 8$. Note a significant drop in performance for the C++ single-source version for the smallest lattice size, which warrants further investigation to determine the underlying cause.

4. Conclusion and outlook

In this report, we have explored the application of various new C++ features to standard LQCD compute kernels. Although our primary focus was on utilizing C++17 parallel algorithms, it's noteworthy that many of these features are versatile and could be beneficially integrated into existing LQCD libraries. When discussing `std::execution::par` algorithms, it's important to highlight two significant limitations: the lack of execution parameters and the absence of asynchronous versions. The former drawback limits the ability to fine-tune and control parallel execution workflow, which can be crucial for optimizing performance in certain HPC applications, while the latter implies that these algorithms cannot be easily integrated into non-blocking, concurrent workflows, and thus impacting performance on large scale. The second issue, however, is expected to be addressed in

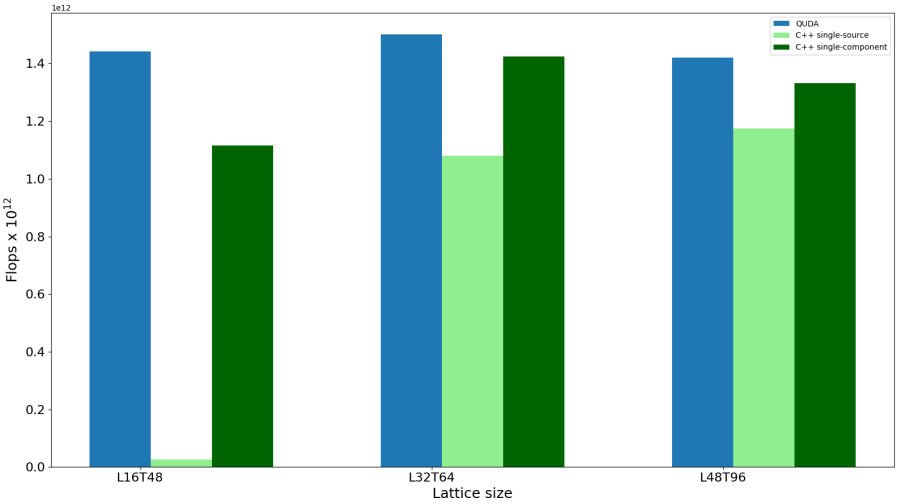

Figure 1: Performance summary on A100 GPU, single precision HISQ dslash.

future revisions of the ISO C++ standard. In particular, this will be through the introduction of *sender* model of asynchronous programming [12] (see also [13]), a feature that NVIDIA has already implemented experimentally.

5. Acknowledgements

The author expresses gratitude for the valuable discussions with colleagues from the USQCD ECP portability team. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Special thanks to the developers of ChatGPT, an AI language model by OpenAI. The tool was especially helpful in exploring the new features of the recent ISO C++ revisions.

References

[1] V. D. Elvira, S. Gottlieb, O. Gutsche, et al., *The Future of High Energy Physics Software and Computing,* (2022) [arXiv:2210.05822]

[2] ISO, ISO/IEC JTC1 SC22 WG21 N4507 *Programming languages — Technical Specification for C++ Extensions for Parallelism*, May 2015
https://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2015/n4507.pdf

[3] P. Halpern, SC22 WG21 N3916, *Polymorphic Memory Resources*, February 2014
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3916.pdf

[4] N. MacIntosh, S. T. Lavavej, SC22 WG21 P0122R7, *span: bounds-safe views for sequences of objects*, March 2018
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0122r7.pdf

[5] C. Trott et al, SC22 WG21 P0009r18, *MDSPAN*, July 2022
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html

[6] A. Sutton, SC22 WG21 P0734R0, *Wording Paper, C++ extensions for Concepts*, July 2017
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf

[7] S. Brand, M. Dominiak, SC22 WG21 P2374R4, *views::cartesian_product*, July 2022
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2374r4.html

[8] https://github.com/TartanLlama/ranges
https://sc22.supercomputing.org/presentation/?id=tut149&sess=sess220

[9] https://github.com/alexstrel/dataparallel-lqft-cpp2x

[10] M. A. Clark et al., *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs*, Comput. Phys. Commun. **181**, 1517-1528 (2010)

[11] https://github.com/lattice/quda

[12] M. Dominiak et al, SC22 WG21 P2300R7, *std::execution*, April 2023
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r7.html

[13] R. Arutyunyan, A. Kukanov, SC22 WG21 P2500R1, *C++ parallel algorithms and P2300*, May 2023
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2500r1.html