Kiwaku, a C++20 library for multidimensional arrays

Application to ACTS tracking

Sylvain Joube[1,2,*], *Hadrien* Grasland[1,**], *David* Chamont[1,***], and *Joël* Falcou[2,****]

[1]Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405, Orsay, France
[2]Université Paris-Saclay, CNRS, Laboratoire interdisciplinaire des sciences du numérique, 91405, Orsay, France

Abstract.
C++ is a vital part of particle physics as it allows for high level abstractions while offering state of the art performance. In this article we will first introduce the *C++20 concepts*, a major C++ usability enhancement. We will then introduce Kiwaku, a new multidimensional arrays library taking advantage of the most advanced C++ features at the time of writing, providing the user with intuitive API while retaining state of the art performance. Multidimensional arrays are a basic building block for many scientific experiments and simulations, and particle physics is no exception. Using examples borrowed from *Covfie* and *ACTS* libraries, and using data from the *ATLAS CERN* experiment, we will show how Kiwaku offers good usability while having a negligible impact on performance compared to using classic C++ *std::array*s.

1 Introduction

Track reconstruction, also known as tracking, is a vital part of the HEP event reconstruction process, and one of the largest consumers of computing resources. The upcoming HL-LHC upgrade will increase the need for software able to make efficient use of the underlying heterogeneous hardware. However, this evolution should not imply the production of code unintelligible to most of its maintainers, hence the need to take care of usability for both end users and developers.

C++ has long been a language of choice for efficient scientific computing tasks. The *Generative Programming paradigm* [1], which relies on heavy type based *template metaprogramming*, provides a powerful solution for supporting multiple execution contexts [2]. Yet *templates* are also usually blamed for unnecessarily large binary files, high code complexity and difficult-to-understand error messages.

In this article, we will discuss recent developments made to the C++ language, helping to define a new process for constructing libraries that are both efficient and easy to use. We will first illustrate how C++20 concepts makes for better error reporting and overall code clarity. We will then introduce Kiwaku [3], a new *C++20* multidimensional arrays library taking advantage of the most recent C++ usability improvements, yet providing portable

*e-mail: sylvain.joube@lisn.fr
**e-mail: hadrien.grasland@ijclab.in2p3.fr
***e-mail: david.chamont@ijclab.in2p3.fr
****e-mail: joel.falcou@lisn.fr

performance [4] on various hardware such as CPUs and GPUs. We will finally discuss a few proofs of concept, based on use-cases borrowed from the *ACTS project* [5], and more specifically the Covfie [6] library, which provides magnetic field computation and *Lorentz-Euler* track propagation.

2 A short glimpse into C++ usability

Inheritance-based object oriented programming can be easy to understand, write and maintain, at the expense of runtime performance. To make C++ code use modern hardware more efficiently, template meta-programming proved necessary. Yet templates may turn into a nightmare for non computer science experts, as the templated code can be very difficult to grasp. It notoriously leads to painfully long and cryptic error messages and impedes code readability and maintainability.

As an example, consider figure 1, which defines a function template get_nth, invokes std::begin() on the parameter c. An argument of type *std::list<float>* will successfully compile, since it has such a function. But calling our function with an *int* will lead to an error message that prints the entire call stack and ultimately refers to fragments of code unknown to the developer (the C++ standard library in this case). Part of the resulting error message is reproduced in figure 2.

```
template <typename T>
auto get_nth(T const& c, int n) // Returns n-th element value
{
  assert(n < std::size(c));
  auto b = std::begin(c);
  for(int i = 0; i<n; ++i) b++;
  return *b;
}

int main()
{
    std::list<float> a{1, 2, 3, 4, 5};
    auto x1 = get_nth(a, 2); // Compiles
    auto x2 = get_nth(1, 2); // Very long and cryptic error message
}
```

Figure 1. The templated *get_nth* function leads to a non-user-friendly error message during compilation when called with unsupported arguments, such as an int.

To mitigate this issue, the long awaited *C++ concepts* were introduced in C++20. They provide a powerful and usable solution for adding constraints to templates, constraining the set of arguments that are accepted as *template arguments*. When a templated function is called with an argument not modeling the *concept*, the compiler will stop at the function header and print an error message, clearly stating the location of the error and its cause. It will not inspect the function implementation and instantiate the entire call stack as it would have done otherwise.

C++ concepts can be refined using a pattern close to interface inheritance, meaning a concept can be refined by another, and extended at will. The concept thusly defined subsumes the concepts of its ancestors plus any directly specified concepts. It is then possible to overload a function constrained by concepts, as the strongest constraint will always be chosen.

As an example, described in figure 3, let's take our previous *get_nth* function returning the nth element of an array-like structure. We can avoid the previous cryptic error message by defining a concept *basic_container* and requesting that our *get_nth* function only

```
main.cpp:24:12: error: no matching function for call to 'begin'
  auto b = std::begin(c);
           ^~~~~~~~~~
/gcc-12.2.0/include/c++/12.2.0/initializer_list:90:5: note: candidate
template ignored: could not match 'initializer_list<_Tp>' against 'int'
    begin(initializer_list<_Tp> __ils) noexcept
    ^
... 14 lines hidden ...
/gcc-12.2.0/include/c++/12.2.0/bits/range_access.h:113:31: note: candidate
template ignored: could not match 'valarray<_Tp>' against 'const int'
  template<typename _Tp> _Tp* begin(valarray<_Tp>&) noexcept;
                             ^
/gcc-12.2.0/include/c++/12.2.0/bits/range_access.h:114:37: note: candidate
template ignored: could not match 'const valarray<_Tp>' against 'const int'
  template<typename _Tp> const _Tp* begin(const valarray<_Tp>&) noexcept;
                                    ^
```

Figure 2. Part of the resulting very long and cryptic error message when figure 1 code gets compiled with *gcc-12.2.0*. We hid 14 lines out of the 29 lines of the full error message to save space.

accepts arguments that model this concept. When our function gets called with an argument not matching the concept, the compiler will stop there and print an error message, without descending further into the call hierarchy.

It is then possible to refine our previous concept by making a stronger concept asking for the array-like argument to support random access. As shown on figure 3, we can now overload our function *get_nth* with our more specific concept *random_access_container*, with its code optimized for this kind of structure, since the strongest constraint will always be chosen by the compiler. This is a very welcomed and easy to use way to specialize functions according to fine-grained types.

3 KIWAKU: Applied C++20 for Data Management

Multidimensional arrays are an integral part of scientific computing. Research on libraries for multidimensional arrays has been abundant, giving rise to many designs.

- Some libraries are intended for a single use-case, such as *Fastor* [7] for tensors, Covfie for vector field benchmarks and *md_span* [8] for managing conversions between n-dimensional positions and in-memory locations.
- Some are exclusively dedicated to mathematical computations and linear algebra, like the *Matrix Template Library* [9], *Intel MKL* [10], *Eigen* [11], *NT2* [12] and the *GNU Scientific Library* [13].
- Some of these libraries also act as C++ wrappers for other linear algebra libraries, like *Armadillo* [14] which can leverage *OpenBLAS* [15], *LAPACK* [16], *ARPACK* [17] and *SuperLU* [18].
- Finally, some can be seen as complete ecosystems, having both mathematical, storage and analysis capabilities like the *Kokkos* ecosystem [19] or the *Dlib* [20] library. These libraries all support CPU calculations, with automatic or manual vectorization. Some of them also support offloading work to a GPU or distributing it over a computing cluster.

With KIWAKU, we made the choice to explore what C++20 could allow in terms of usability while preserving performance. KIWAKU is only compatible with C++20 and onward and does not offer compatibility with previous C++ standards, allowing it to take advantage

```cpp
// basic_container concept definition
template<typename T>
concept basic_container = requires(T const& c)
{
  { std::size(c)   };
  { std::begin(c) } -> std::forward_iterator;
  { std::end(c)   } -> std::forward_iterator;
};

auto get_nth(basic_container auto const& c, int n) // General case
{
  assert(n < std::size(c));
  auto b = std::begin(c);
  for(int i = 0; i<n; ++i) b++;
  return *b;
};

// Refinement of basic_container into a more specific concept
template<typename T>
concept random_access_container = basic_container<T>
                                  && requires(T const& c, int i)
{
  // All the above from concept basic_container
  { c[i] }; // And this expression must compile
};

// Specialization, only for random_access_container types
auto get_nth(random_access_container auto const& c, int n)
{
  assert(n < std::size(c));
  return c[n];
};

std::list<float>     b;  get_nth(b, 2); // Call with basic_container
std::array<float, 4> r;  get_nth(r, 2); // Call with random_access_container
```

Figure 3. Examples of C++20 concept definition and refinement. Depiction of function overloading according to these newly defined concepts.

of all the new features from the C++20 standard, including the C++ concepts we previously introduced and older C++ features improved by C++20 such as *constexpr if* and *constexpr functions*. Furthermore, to obtain satisfactory performance, KIWAKU is based on *generative meta-programming* using C++ templates.

KIWAKU provides both owning tables and non-owning views. The former manages its own memory allocation, like *std::vector*, while the latter points to a pre-existing allocation, like *std::span*. Those containers are defined with expressive and high level arguments. Linear algebra, *expression templates* and complex calculations are all out of scope for KIWAKU, as it is designed to solely be an efficient multidimensional arrays library providing optimized and easy to use traversal algorithms. It aims to provide a compatibility layer with pre-existing calculation libraries such as Intel MKL, Eigen and Kokkos. Multiple execution contexts are or will be also supported, including traditional CPUs, most GPUs with the help of *SYCL* [21], distributed computations through *MPI* [22] and vectorization using the *Eve* library [23] [24].

Although KIWAKU may at first look like md_span, it differs greatly from it. Indeed, md_span only offers *views* on pre-existing allocations whereas KIWAKU offers views and *tables*, the later being allocated by default or with a user-defined allocator, useful for situations

where GPU or distribution contexts are needed. Unlike md_span, KIWAKU also offers optimized traversal algorithms on its views and tables.

3.1 KIWAKU: Usability

When a generic function is invoked with an argument of invalid type, we want the error message to be as clear and concise as possible. As introduced earlier, C++20 concepts are particularly suited for this kind of task as they restrain the set of arguments allowed as template arguments, and immediately lead to an error when the given argument does not model the concept, without going down the whole call hierarchy, hence KIWAKU relies on them for its usability.

Let's say we want to make a *square_each* function that squares every element of a 2D KIWAKU view of *floats*, as depicted on figure 4. How does this generic function express what it expects as an argument? To start with, a KIWAKU view is represented by the concept *kwk::concepts::view*. We then add two more constraints for the view: we want the view to be 2-dimensional, and to contain floats. Then, we call the *for_each* function of KIWAKU to finally square each element of the view. Without C++20, we would not have been able to define clean concepts, thus a *square_each* function would have been much more complex to write for the end user.

```
#include <kwk/kwk.hpp>

// Only expects a 2D kiwaku view of floats: constraint via C++20 concepts
void square_each(kwk::concepts::view<kwk::_2D, kwk::as<float>> auto& view)
{
  // For each value of v: square the value
  kwk::for_each( [](auto& e) { e *= e; }, view);
}

void demo_kiwaku_view(float* data, int width, int height)
{
  kwk::view v { kwk::size = kwk::of_size(width, height), kwk::source = data };
  square_each(v);
}
```

Figure 4. Definition of a function taking a KIWAKU view as argument, and applying the KIWAKU *for_each* to each element o the view.

KIWAKU is engineered to give an easy to write and understand interface with other libraries and native C++ code. The creation of a KIWAKU view from a *C++ array* can be achieved in a single line of code, by specifying the *shape* of the desired view and its *source*. The data type is then automatically deduced from the source. Once the KIWAKU view is created, it can be used by other functions taking such a view as argument. The code of function *demo_kiwaku_view* from figure 4 demonstrates this process.

3.2 KIWAKU: Real life examples

Performance-wise, it is important that the use of KIWAKU only adds a barely noticeable overhead, if any. To assess this, we conducted two experiments: starting from pre-existing physics code examples of the Covfie magnetic field handling library, we evaluated the relative performance between equivalent code written with KIWAKU and classic standalone C++ code using std::arrays. The relevant code is available at [25].

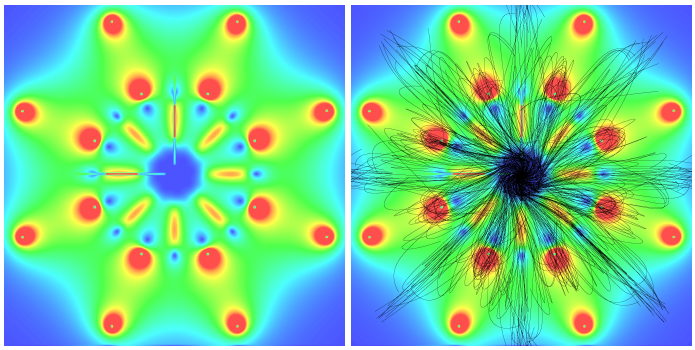

Figure 5. Magnetic field rendering for the ATLAS detector, with a few particle tracks added to the right image.

The first significant example we chose is a magnetic field rendered into a 2D image for the ATLAS detector [26], as seen on figure 5. The left image is a slice of the ATLAS magnetic field: colors indicate the magnetic field strength. A few charged particle track simulations with the Lorentz-Euler algorithm have been added to the right image. The vector field data is made available by the ACTS library. The benchmark consists in measuring the elapsed time for filling a 1024x1024 image with pixel colors representing the intensity of the magnetic field at each point of the detector.

In this example, the Kiwaku code is a little shorter than our standalone C++ code since Kiwaku provides the necessary functions to interpolate between its container's cells when a floating point position is asked by the user code. We were pleased yet a bit intrigued by the results of our benchmark, since the Kiwaku version proved to be slightly faster than the standalone version, consistently by about 10%, for every slice. This is most likely due to the compiler being able to optimize further the code generation by its knowledge of the full context in which each function was called for the Kiwaku version. Of course, we always automatically checked that the rendered image was exactly the same in both cases.

As another example, we choose the Lorentz-Euler algorithm for charged particle track simulation in a magnetic field. The particles were moving in the same ACTS magnetic field, and our benchmark measured the elapsed time for a given quantity of particles moved a given number of iterations. Like before, we ensured that the results between our two code versions were identical. This time, the standalone version proved to be slightly faster by about 3%.

3.3 Kiwaku: Future work

The Kiwaku library is designed to offer optimized traversal algorithms for its tables and views, in the most usable way possible. It is not made to handle linear algebra calculations so it will provide an interface with other math libraries such as Intel MKL, Kokkos and Eigen. It will offer multiple specific traversal algorithms such as *Hilbert*, *Morton*, *row/column-major* and its extension to 3 and more dimensions. The *slicing* of views and tables according to user-specified dimensions is already supported. Kiwaku currently only supports CPU execution, but multiple execution contexts will soon be available, such as parallel CPU execution with an underlying *OpenMP* [27] layer, vectorization with the SYCL library and GPU execution with the SYCL backend. It will also provide the user with tools to create his own execution context, extending Kiwaku's basic *context* types at will. In the more distant future, Kiwaku will also use MPI to allow execution on a cluster of servers, allowing execution contexts

to be stacked on top of each other, having for example a GPU SYCL context run on each distributed node of an MPI context.

4 Conclusion

We have shown how the recent developments made to the *C++ standard* enable libraries to be more usable and easier to understand and maintain, while preserving performance. This is partly achieved by using C++ concepts, constexpr functions and constexpr if, based on the Generative Programming paradigm and template metaprogramming. We then introduced Kiwaku, a C++20 library for multidimensional arrays using the most recent available and supported features of C++ at the time of writing. It is aimed at maximizing usability while retaining the runtime performance for which C++ is known. We showed that, on top of offering good usability, the performance of Kiwaku for two physics examples borrowed from the ACTS and Covfie libraries were on par with C++ code only using std::arrays [25].

References

[1] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen, *Generative Programming and Active Libraries*, in *Generic Programming*, edited by M. Jazayeri, R.G.K. Loos, D.R. Musser (Springer, Berlin, Heidelberg, 2000), Lecture Notes in Computer Science, pp. 25–39, ISBN 978-3-540-39953-7

[2] I. Masliah, M. Baboulin, J. Falcou, *Meta-Programming and Multi-stage Programming for GPGPUs*, in *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC 2016)* (2016), p. 369

[3] J. Falcou, *Kiwaku github repository*, https://github.com/jfalcou/kiwaku

[4] S.J. Pennycook, J.D. Sewall, V.W. Lee, *A metric for performance portability* (2016), 1611.07409

[5] X. Ai, C. Allaire, N. Calace, A. Czirkos, M. Elsing, I. Ene, R. Farkas, L.G. Gagnon, R. Garg, P. Gessinger et al., *A Common Tracking Software Project*, in *Computing and Software for Big Science* (2022), Vol. 6, p. 8

[6] S.N. Swatman, A.L. Varbanescu, A. Pimentel, A. Salzburger, A. Krasznahorkay, *Systematically Exploring High-Performance Representations of Vector Fields through Compile-Time Composition*, in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Association for Computing Machinery, New York, NY, USA, 2023), ICPE '23, pp. 55–66, ISBN 9798400700682

[7] R. Poya, A.J. Gil, R. Ortigosa, *A High Performance Data Parallel Tensor Contraction Framework: Application to Coupled Electro-Mechanics*, in *Computer Physics Communications* (2017), Vol. 216, pp. 35–52

[8] D.S. Hollman, B. Adelstein-Lelbach, H.C. Edwards, M. Hoemmen, D. Sunderland, C.R. Trott, *Mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards*, in *CoRR* (2020), Vol. abs/2010.06474, 2010.06474

[9] J.G. Siek, A. Lumsdaine, in *Computing in Object-Oriented Parallel Environments*, edited by D. Caromel, R.R. Oldehoeft, M. Tholburn (Springer Berlin Heidelberg, Berlin, Heidelberg, 1998), Vol. 1505, pp. 59–70, ISBN 978-3-540-65387-5 978-3-540-49372-3

[10] *Accelerate Fast Math with Intel oneAPI Math Kernel Library*

[11] *Eigen (libeigen) C++ linear algebra library*, https://eigen.tuxfamily.org

[12] P. Estérie, J. Falcou, M. Gaunard, J.T. Lapresté, L. Lacassagne, Journal of Parallel and Distributed Computing **74** (2014)

[13] *GSL - GNU Scientific Library - GNU Project - Free Software Foundation*, https://www.gnu.org/software/gsl/

[14] C. Sanderson, R. Curtin, *Armadillo: A Template-Based C++ Library for Linear Algebra*, in *Journal of Open Source Software* (The Open Journal, 2016), Vol. 1, p. 26

[15] Z. Xianyi, *OpenBLAS: An optimized BLAS library*, https://github.com/xianyi/OpenBLAS

[16] *LAPACK Users' Guide – Third Edition*, https://www.netlib.org/lapack/lug/

[17] R.B. Lehoucq, D.C. Sorensen, C. Yang, *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods* (SIAM, 1998), ISBN 978-0-89871-407-4

[18] X.S. Li, P. Lin, Y. Liu, P. Sao, *Newly Released Capabilities in the Distributed-Memory SuperLU Sparse Direct Solver*, in *ACM Transactions on Mathematical Software* (2023), Vol. 49, pp. 1–20

[19] *Kokkos Ecosystem – Part of the Exascale Project*, https://kokkos.org/

[20] *Dlib C++ Library*, http://dlib.net/

[21] *SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload" The Khronos Group, Jan. 20, 2014.*, https://www.khronos.org/sycl/

[22] *Open MPI: Open Source High Performance Computing*, https://www.open-mpi.org/

[23] J. Falcou, J. Serot, *E.V.E., An Object Oriented SIMD Library*, in *Scalable Computing: Practice and Experience* (2005), Vol. 6

[24] P. Estérie, J. Falcou, M. Gaunard, J.T. Lapresté, *Boost.SIMD: Generic Programming for Portable SIMDization*, in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing* (Association for Computing Machinery, New York, NY, USA, 2014), WPMVP '14, pp. 1–8, ISBN 978-1-4503-2653-7

[25] S. Joube, *Reference code used in this article, with full instructions for reproducible results*, https://github.com/SylvainJoube/CHEP2023-kiwaku

[26] *ATLAS Experiment at CERN*, https://atlas.cern/

[27] *The OpenMP API specification for parallel programming*, https://www.openmp.org/