KIWAKU

A C++20 library for multidimensional arrays, applied to ACTS tracking

Sylvain JOUBE

Hadrien GRASLAND, David CHAMONT, Joël FALCOU.

LISN
LABORATOIRE INTERDISCIPLINAIRE DES SCIENCES DU NUMÉRIQUE

IJCLab
Irène Joliot-Curie
Laboratoire de Physique des 2 Infinis

Powered by Markdeep and Markdeep-Slides

Code Ergonomics vs. Performance

Challenges

- How to lessen development time for non-computer science experts?
- How to abstract the hardware and preserve portable performance?

A path forward: improve C++ templates ergonomics

- *Templates* have issues: long error messages, difficult to understand
- C++20 new features: concepts, if constexpr, constexpr functions
- Example: KIWAKU, a C++20 library for multidimensional arrays

Physics guinea pigs from ACTS with Kiwaku

- ACTS: C++17 experiment-independent toolkit for (charged) particle track reconstruction
- ATLAS magnetic field rendering
- Lorentz-Euler track propagation

Issues with Templates: Error Messages

Examples: function extracting the nth element of a container

```
1  template <typename T>
2  auto get_nth(T const& c, int n) // Returns n-th element value
3  {
4    assert(n < std::size(c));
5    auto b = std::begin(c);
6    for(int i = 0; i<n; ++i) b++;
7    return *b;
8  }
9
10 int main()
11 {
12     std::list<float> a{1, 2, 3, 4, 5};
13     auto x1 = get_nth(a, 2); // Compiles
14     auto x2 = get_nth(1, 2); // Very long and cryptic error message
15 }
```

Issues with Templates: Error Messages

```
main.cpp:22:14: error: no matching function for call to 'size'
  assert(n < std::size(c));
             ^~~~~~~~
/usr/include/assert.h:93:27: note: expanded from macro 'assert'
      (static_cast <bool> (expr)
                          ^~~~
main.cpp:36:14: note: in instantiation of function template specialization 'get_nth<int>' requested here
    auto x = get_nth(1,2);
             ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:264:5: note: candidate template ignor
    size(const _Container& __cont) noexcept(noexcept(__cont.size()))
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:274:5: note: candidate template ignor
    size(const _Tp (&)[_Nm]) noexcept
    ^
main.cpp:24:12: error: no matching function for call to 'begin'
  auto b = std::begin(c);
           ^~~~~~~~~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/initializer_list:90:5: note: candidate template ignored:
    begin(initializer_list<_Tp> __ils) noexcept
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:52:5: note: candidate template ignore
    begin(_Container& __cont) -> decltype(__cont.begin())
    ^                                     ~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:63:5: note: candidate template ignore
    begin(const _Container& __cont) -> decltype(__cont.begin())
    ^                                           ~
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:95:5: note: candidate template ignore
    begin(_Tp (&__arr)[_Nm]) noexcept
    ^
/opt/compiler-explorer/gcc-12.2.0/lib/gcc/x86_64-linux-gnu/12.2.0/../../../../include/c++/12.2.0/bits/range_access.h:113:31: note: candidate template ignc
  template<typename _Tp> _Tp* begin(valarray<_Tp>&) noexcept;
                              ^
```

Concepts: Constrained Genericity

General principles

- Idea: constrain generic template parameters
- Can be used to optimize/specialize functions according to types
- Faster compile time, better error messages

Example:

```
1  template<typename T>
2  concept basic_container = requires(T const& c)
3  {
4    { std::size(c)  }; // This expression must compile
5
6    // Expressions must compile and return a value that behaves as a std::forward_iterator
7    { std::begin(c) } -> std::forward_iterator;
8    { std::end(c)   } -> std::forward_iterator;
9  };
```

Concepts: Constrained Genericity

Code Using Concept

```
auto get_nth(basic_container auto const& c, int n)
{
  /*Exact same code*/
};
```

Resulting Error Message

```
main.cpp:29:14: error: no matching function for call to 'get_nth'
    auto x = get_nth(1,2);
             ^~~~~~~
main.cpp:13:6: note: candidate template ignored: constraints not satisfied [with c:auto = int]
auto get_nth(basic_container auto const& c, int n)
     ^

main.cpp:13:14: note: because 'int' does not satisfy 'basic_container'
auto get_nth(basic_container auto const& c, int n)
             ^
main.cpp:8:5: note: because 'std::begin(c)' would be invalid: no matching function for call to 'begin'
  { std::begin(c) } -> std::forward_iterator;
    ^
1 error generated.
```

Concepts: Combinaison and Refinement

General Principles

- A concept can be refined (add constraints)

```
1  template<typename T> // Same container as before
2  concept basic_container = requires(T const& c)
3  {
4    { std::size(c)  }
5    { std::begin(c) } -> std::forward_iterator;
6    { std::end(c)   } -> std::forward_iterator;
7  };
8
9  template<typename T> // Refinement:
10 concept random_access_container = basic_container<T> && requires(T const& c, int i)
11 {
12   // All the above from concept container
13   { c[i] }; // And this expression must compile
14 };
```

Concepts: Combinaison and Refinement

Application to function overload

```
1  auto get_nth(basic_container auto const& c, int n) // General case
2  {
3    assert(n < std::size(c));
4    auto b = std::begin(c);
5    for(int i = 0; i<n; ++i) b++;
6    return *b;
7  };
8
9  auto get_nth(random_access_container auto const& c, int n) // Specialization
10 {
11   assert(n < std::size(c));
12   return c[n];
13 };
14
15 std::list<float>     b;  get_nth(b, 2); // Call with basic_container
16 std::array<float, 4> r;  get_nth(r, 2); // Call with random_access_container
17
```

Compile-time Computation: Constexpr functions

Before `constexpr`: recursive variadic templates

```
1  template<int N> struct factorial    { static const int value = N * factorial<N-1>::value; };
2  template<>      struct factorial<0> { static const int value = 1; };
3
4  std::array<float, factorial<4>::value>> x; // call in a compile-time context
```

`constexpr` functions to the rescue

```
1  constexpr int factorial(int n)
2  {
3    int r = 1;
4    for(int i=2; i<=n; ++i) r *= i;
5    return r;
6  }
7
8  std::array<float, factorial(4)> a; // call in a compile-time context
9  int f = factorial(7);              // call in a runtime context
```

Code Selection at Compile Time

The old way...

```
1  #include <type_traits>
2
3  template<typename T, typename Enable = void>
4  struct has_size : std::false_type {};
5
6  template<typename T>
7  struct has_size<T, std::void_t<decltype(std::declval<T>().size())>> : std::true_type {};
8
9  template<typename T> auto size(T const& t, std::false_type) { return 1; }
10
11 template<typename T> auto size(T const& t, std::true_type) { return t.size(); }
12
13 template<typename T> auto size(T const& t){ return size(t,typename has_size<T>::type{}); }
14
15 std::vector<int> v(58);
16 int              n = size(v);  //  n == 58
17 int              m = size(8);  //  m == 1
```

Compile-time Computation: `if constexpr`

With `if constexpr`

- `if constexpr` allows code selection at compile-time
- Combination with constraints on types

```
1  template<typename T>
2  auto size(T const& t)
3  {
4    if constexpr( requires(T const& t) { t.size(); } ) // If t has a method named size()
5      return t.size();
6    else // If not, returns 1
7      return 1;
8  }
9
10 std::vector<int> v(58);
11 int              n = size(v);  //  n == 58
12 int              m = size(8);  //  m == 1
```

KIWAKU: Applied C++20 for Data Management

Goal and Basic Components

- **C++20** library for multidimensional *(non owning)* views and *(owning)* tables
- Containers definition with high-level parameters
- Generative programming with `templates`, `concepts`, `constexpr`

Current status

- Out of scope: linear algebra, expression templates, complex calculations
- WIP: Optimized and ergonomic traversal algorithms
- WIP: Support for multiple execution contexts (CPU, GPU, distribution, vectorization...)

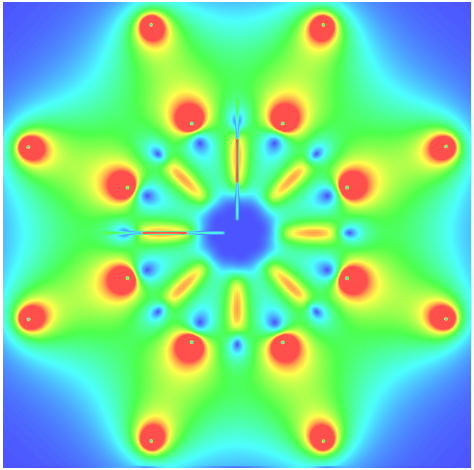
KIWAKU: Example

Creating and manipulating views with concepts

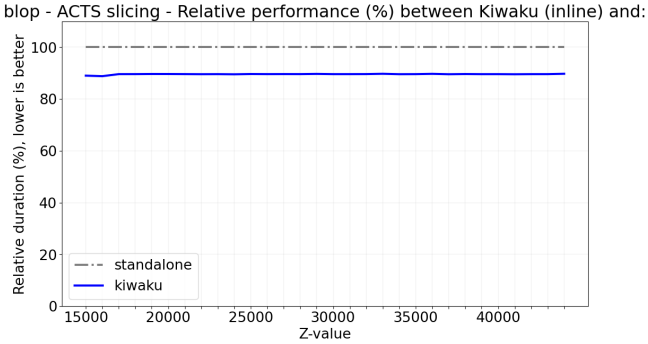
```
#include <kwk/kwk.hpp>

// Expects only 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)
{
  // Construction of a 2D view of floats from a pointer
  kwk::view v { kwk::size = kwk::of_size(width, height), kwk::source = data };
  square_each(v);
}
```

KIWAKU Performance: ATLAS Magnetic Field Rendering

Original code from COVFIE

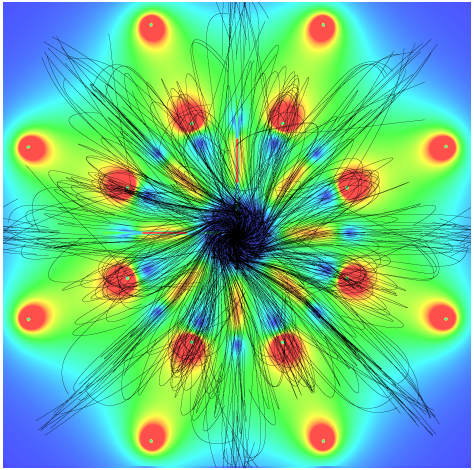


- 2D slice of the ATLAS magnetic field

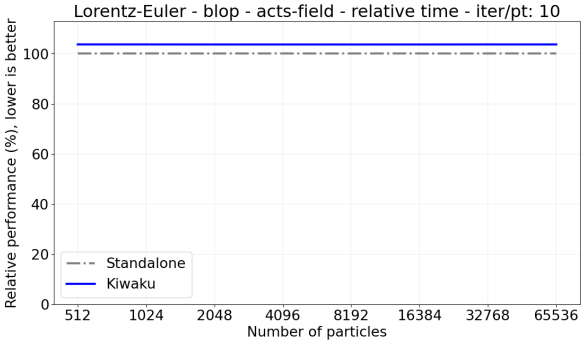


KIWAKU Performance: Lorentz-Euler Algorithm

Original code from COVFIE



- Computed trajectories of charged particles in the ATLAS vector field



Conclusion

Better Ergonomics With C++20

- Concepts:
  - Constrained genericity via compile-time predicates
  - Easy to combine, powerful function specialization (strongest constrain chosen)
- Constexpr:
  - Resembles to classic C++, more ergonomic than macros
  - Allows passing of more complex values as template parameters

Future Work on KIWAKU

- Optimized traversal algorithms for multidimensional arrays in C++20
- Ergonomic for maintainers and users
- Future support for GPU execution contexts via SYCL and distributed via MPI

Thank You!

- **KIWAKU**
  - Github: `https://github.com/jfalcou/kiwaku`
- **COVFIE** *special thanks to Stephen Nicholas Swatman*
  - Github: `https://github.com/acts-project/covfie`
  - ACAT'22: `https://indico.cern.ch/event/1106990/contributions/4991271/`

Backup slides