KIWAKU, CONTAINERS WELL MADE

GENERATIVE PROGRAMMING FOR EFFICIENT HPC DATA STRUCTURES

April 8, 2022 REPRISES

Sylvain JOUBE

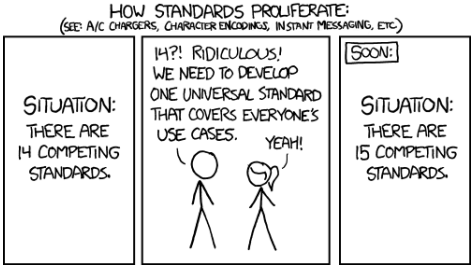
Joël FALCOU, Hadrien GRASLAND, David CHAMONT.

Context

nD Arrays : a pervasive tool

- Main data structure in **numerical simulations**
- Expected to be **efficient**
- Expected to be **easy to use** yet **expressive**

Problem already addressed many times :



Context

nD Arrays : a pervasive tool

- Main data structure in **numerical simulations**
- Expected to be **efficient**
- Expected to be **easy to use** yet **expressive**

Problem already addressed many times :

Libraries	C++ Standard ?
`std::vector`, `std::valarray`, `std::mdspan`	✓
EIGEN, Armadillo, xtensor, Blaze, Kokkos, ...	✗

What if the real issue was a software design problem?

nD Array Design Space

A small sampling

- Owning or non-owning array ?
- Are my dimensions runtime or compile-time ? or both ?
- Storage order : C, FORTRAN, arbitrary, ... ?
- Indexes start at 0 ? 1 ? −3 ? any user value ?
- Memory allocation : Allocator-based ? Which allocator model ? Stack or Heap ?

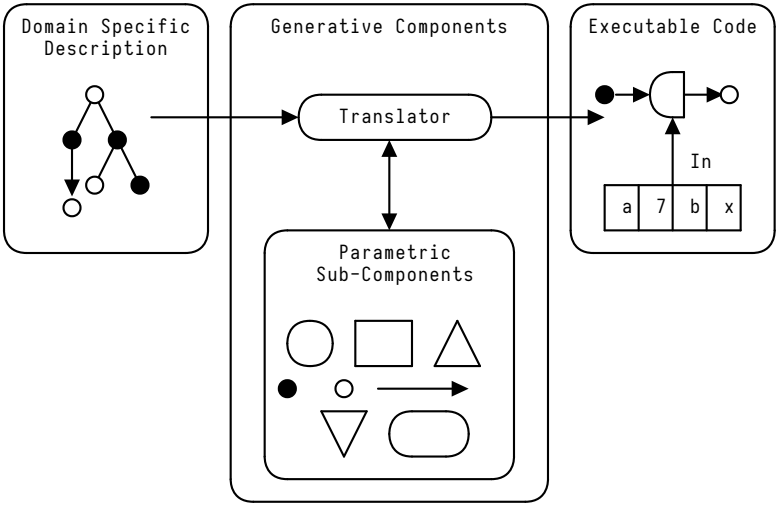
One implementation to rule them all ?

- Monolithic implementation leads to unmaintainable code
- Arbitrary restrictions on API is not a solution

OUR POSITION

Apply a different design strategy to the problem : generative programming

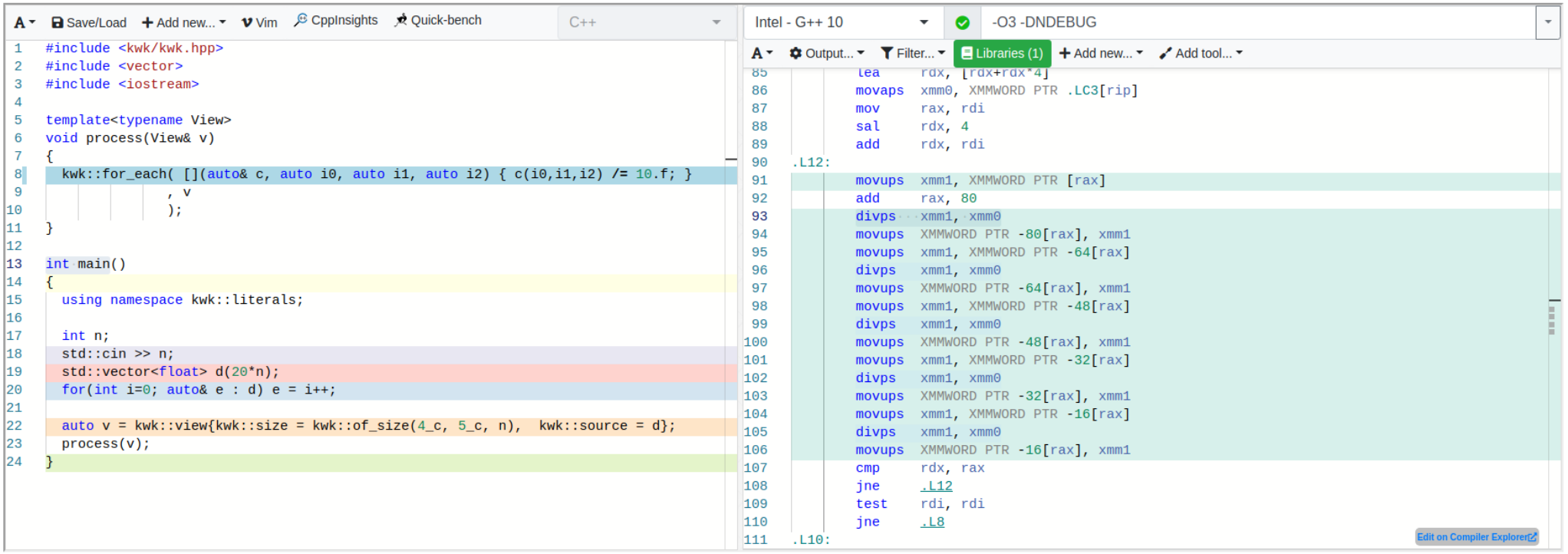
Generative Programming [Czarnecki et al. : 2002]



Generative Programming for nD arrays

Advantages of Generative programming

- Reduce the number of implementations from $|X| \cdot |Y| \cdot |Z|$ to $|X| + |Y| + |Z|$
- Generative programming in C++ is based on **template metaprogramming** and **code generation**




```cpp
#include <kwk/kwk.hpp>
#include <vector>
#include <iostream>

template<typename View>
void process(View& v)
{
  kwk::for_each( [](auto& c, auto i0, auto i1, auto i2) { c(i0,i1,i2) /= 10.f; }
               , v
               );
}

int main()
{
  using namespace kwk::literals;

  int n;
  std::cin >> n;
  std::vector<float> d(20*n);
  for(int i=0; auto& e : d) e = i++;

  auto v = kwk::view{kwk::size = kwk::of_size(4_c, 5_c, n),  kwk::source = d};
  process(v);
}
```

```
        lea     rdx, [rdx+rdx*4]
        movaps  xmm0, XMMWORD PTR .LC3[rip]
        mov     rax, rdi
        sal     rdx, 4
        add     rdx, rdi
.L12:
        movups  xmm1, XMMWORD PTR [rax]
        add     rax, 80
        divps   xmm1, xmm0
        movups  XMMWORD PTR -80[rax], xmm1
        movups  xmm1, XMMWORD PTR -64[rax]
        divps   xmm1, xmm0
        movups  XMMWORD PTR -64[rax], xmm1
        movups  xmm1, XMMWORD PTR -48[rax]
        divps   xmm1, xmm0
        movups  XMMWORD PTR -48[rax], xmm1
        movups  xmm1, XMMWORD PTR -32[rax]
        divps   xmm1, xmm0
        movups  XMMWORD PTR -32[rax], xmm1
        movups  xmm1, XMMWORD PTR -16[rax]
        divps   xmm1, xmm0
        movups  XMMWORD PTR -16[rax], xmm1
        cmp     rdx, rax
        jne     .L12
        test    rdi, rdi
        jne     .L8
.L10:
```

Existing solutions

Overview

What do users want?

- Small and non-ambiguous interface
- High discoverability
- Clear and concise documentation

What do users deal with?

- Some APIs suffer from feature creep
- Some APIs require strong programming skills
- Some (most) APIs are sometimes counterintuitive and verbose

What is missing?

- User friendly for non computer experts
- Keep API scope reasonable

Painful example : std::vector

Higher dimensions vector declaration

```
// 5D vector declaration and creation
std::vector<std::vector<std::vector<std::vector<std::vector<float>>>>>
  my_vect(  d0,
            std::vector<std::vector<std::vector<std::vector<float>>>>(d1,
            std::vector<std::vector<std::vector<float>>>(d2,
            std::vector<std::vector<float>>(d3,
            std::vector<float>(d4)
            )))
          );

// Access to fields
my_vect[i0][i1][i2][i3][i4] = some_float;
```

- For a $d_1 \times d_2 \times \ldots \times d_n$ array, $1 + d_1 \times d_2 \times \ldots \times d_{n-1}$ allocations occurs
- Data access induces spurious cache misses
- Memory is not contiguous, leading to sub-par performance

Painful example : std::mdspan

Extent definition

```
std::extents<4,3> fixed_4x3;                                          // Compile-time known extent
std::extents<std::dynamic_extent,std::dynamic_extent> dynamic(4,3); // Runtime known extent

std::extents< std::dynamic_extent,std::dynamic_extent,std::dynamic_extent,std::dynamic_extent
            , std::dynamic_extent,std::dynamic_extent,std::dynamic_extent,std::dynamic_extent // Ughhhh
            > dyn8d;
```

Type interface

```
template< typename T, typename Extents
        , typename LayoutPolicy = std::layout_right, typename Accessor = std::accessor_basic
        >
class mdspan;
```

Painful example : Expression template libraries

API inconsistencies and traps

```
auto unexpected_lazy_evaluation(matrix const& A, matrix const& B)
{
  auto C = A * B; // The type of C is not matrix
  return C;
}

template<typename T> auto dangerous_reference_to_temporary(T const& t)
{
  matrix m = t + t;
  return m / t; // returns a reference to a temporary
}

template<typename T> auto silent_performance_bug(T const& t)
{
  T that = t * t; // may or may not be of type T
  return that;
}
```

Kiwaku

Kiwaku design - Overview

Scope reduction limits feature creep

- Monolithic designs are bound to fail
- Kiwaku design focuses on a small set of closely relates features...

Kiwaku's features

- Owning and non-owning containers (`kwk::view`, `kwk::table`) with proper allocators
- Slicing of containers (soon)
- Algorithmic skeletons for order n containers

Kiwaku's non-features

- No linear algebra
- No expression templates
- No computation

Kiwaku design

`kwk::view`

- Never owns any memory
- Wraps existing memory in a nD-array like interface
- Is designed to be as small as possible in 'resting position'

`kwk::table`

- Owns memory allocated via an allocator or on the stack
- Wraps said memory in a nD-array like interface
- Is designed to be as small as possible in 'resting position'

Two sides of the same coin

- A `kwk::table` is a `kwk::view` over the memory it owns
- Efficient `kwk::view` leads to efficient `kwk::table`

Kiwaku design

Supported options

- Kiwaku options are defined as orthogonal policies
- can be used as positional arguments in constructors
- options are exploited both at compile-time and run-time

Options	Values
`kwk::source`	pointers, static arrays, contiguous ranges
`kwk::size`	`kwk::of_size(dn...)`, integers are 1D-shapes
`kwk::base_index`	`kwk::index<i>`, `kwk::indexes<i0,i1,...>`
`kwk::label`	any string-like values
`kwk::allocator`	any types modeling `kwk::concepts::allocator`

Simple view construction

```cpp
// simple_view.cpp
#include <kwk/kwk.hpp>
#include <iostream>

int main()
{
  using namespace kwk;

  double data[6] = {1.2,2.3,3.4,4.5,5.6,6.7};

  auto vd = view{ source = data };
  std::cout << vd.shape() << "\n";
  std::cout << vd.rank() << "\n";

  for(std::size_t i = 0; i < vd.size();++i)
    vd(i) *= 10.;

  for(auto e : data)
    std::cout << e << " ";
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 [ 6 ]
 1
 12 23 34 45 56 67
```

Rank 3 view from standard range

```cpp
#include <kwk/kwk.hpp>
#include <vector>
#include <iostream>

int main()
{
  using namespace kwk;

  std::vector<double> data(18);

  auto vd = view{ source = data
                , of_size(2,3,3)
                , label = "base_freq"
                };

  std::cout << vd.shape() << "\n";
  std::cout << dim<0>(vd) << " x "
            << dim<1>(vd) << " x "
            << dim<2>(vd) << "\n";
  std::cout << vd.rank() << "\n";

  for(std::size_t i2 = 0; i2 < dim<2>(vd);++i2)
    for(std::size_t i1 = 0; i1 < dim<1>(vd);++i1)
    for(std::size_t i0 = 0; i0 < dim<0>(vd);++i0)
      vd(i0,i1,i2) = (i0+1) + 10*(i1+1) + 100*(i2+1);

  for(auto e : data)
    std::cout << e << " ";
  std::cout << "\n\n";

  std::cout << vd << "\n";
}
```

```
.LC0:
        .string "["
.LC1:
        .string " "
.LC2:
        .string " ]"
```

```
Execution build compiler returned: 0
Program returned: 0
 [ 2 3 3 ]
 2 x 3 x 3
 3
 111 112 121 122 131 132 211 212 221 222 231 232 311 312 321 322 331 332

 base_freq:
   [ 111 112 ]
   [ 121 122 ]
   [ 131 132 ]

   [ 211 212 ]
   [ 221 222 ]
   [ 231 232 ]

   [ 311 312 ]
   [ 321 322 ]
   [ 331 332 ]
```

Code generation comparison with standard C

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

template<typename T, auto... Os>
void g( kwk::view<T,Os...> v)
{
  for(std::ptrdiff_t i=0;i<v.size();++i)
    v(i) *= 2.354f;
}

void f_kwk(float* ptr, int n)
{
  g( kwk::view{kwk::source = ptr, kwk::size = n});
}
```

```cpp
#include <cstddef>

void f_c(float* ptr, int n)
{
  for(std::ptrdiff_t i=0;i<n;++i)
    ptr[i] *= 2.354f;
}
```

Left: Intel - G++ 10 -O3 -DNDEBUG.. Assembly

```
1 -f_kwk(float*, int):
2         endbr64
3         movsx   rsi, esi
4         test    rsi, rsi
5         jle     .L1
6         lea     rax, -1[rsi]
7         cmp     rax, 2
8         jbe     .L6
9         mov     rdx, rsi
10        movaps  xmm1, XMMWORD PTR .LC0[ri
11        mov     rax, rdi
12        shr     rdx, 2
13        sal     rdx, 4
14        add     rdx, rdi
15 .L4:
16        movups  xmm0, XMMWORD PTR [rax]
17        add     rax, 16
18        mulps   xmm0, xmm1
19        movups  XMMWORD PTR -16[rax], xmm
20        cmp     rax, rdx
21        jne     .L4
22        mov     rax, rsi
23        and     rax, -4
24        test    sil, 3
25        je      .L9
26 .L3:
27        lea     rdx, 0[0+rax*4]
28        movss   xmm0, DWORD PTR .LC1[rip]
29        lea     rcx, [rdi+rdx]
30        movss   xmm1, DWORD PTR [rcx]
31        mulss   xmm1, xmm0
32        movss   DWORD PTR [rcx], xmm1
```

Right: Intel - G++ 10 -O3 -DNDEBUG.. Assembly

```
1 +f_c(float*, int):
2         endbr64
3         movsx   rsi, esi
4         test    rsi, rsi
5         jle     .L1
6         lea     rax, -1[rsi]
7         cmp     rax, 2
8         jbe     .L6
9         mov     rdx, rsi
10        movaps  xmm1, XMMWORD PTR .LC0[ri
11        mov     rax, rdi
12        shr     rdx, 2
13        sal     rdx, 4
14        add     rdx, rdi
15 .L4:
16        movups  xmm0, XMMWORD PTR [rax]
17        add     rax, 16
18        mulps   xmm0, xmm1
19        movups  XMMWORD PTR -16[rax], xmm
20        cmp     rax, rdx
21        jne     .L4
22        mov     rax, rsi
23        and     rax, -4
24        test    sil, 3
25        je      .L9
26 .L3:
27        lea     rdx, 0[0+rax*4]
28        movss   xmm0, DWORD PTR .LC1[rip]
29        lea     rcx, [rdi+rdx]
30        movss   xmm1, DWORD PTR [rcx]
31        mulss   xmm1, xmm0
32        movss   DWORD PTR [rcx], xmm1
```

Edit on Compiler Explorer

Conclusion

Conclusion

Current work

- `Kiwaku on Github`
- Allocators : adaptation of Alexandrescu's model
- Slicers : complete the implementation

Future of my PhD

- Support for tiling and non-trivial walkthrough (Morton, ...)
- Support for data locality in distributed computing scenarios
- Support for heterogenous platforms (GPU, FPGA, ...)
- Seamless integration with SYCL

Thanks for your attention !