ORGANISATION MÉMOIRE DE DONNÉES STRUCTURÉES

KIWAKU, CONTAINERS WELL MADE

January 14, 2026

Adrien HENROT

Joël FALCOU, Hadrien GRASLAND, David CHAMONT.

LISN
LABORATOIRE INTERDISCIPLINAIRE DES SCIENCES DU NUMÉRIQUE

IJCLab
Irène Joliot-Curie
Laboratoire de Physique des 2 Infinis

Sujet

- Accès unifié aux structures de données du HPC/HTC
- Uniformiser leurs définitions/usages
- Faciliter les changements d'organisation interne
- Tirer partie des hiérarchies mémoires

Objectif: Ergonomie de la performance

- Accéder aux données doit etre **efficace** et **simple**
- Définition logique non contrainte par l'optimisation
- Structure physique modifiable sans changer l'algorithme
- Idée : DSL de description d'organisation mémoire

> Sujet de recherche actif notamment au CERN

Outil : C++

- Utilisé en HPC/HTC, langage puissant et généraliste
- **Métaprogrammation** : Abstraction à moindre coût
- **Concepts** : Contraintes sur le code générique
- Programmation à la compilation (constexpr, ...)

Problématique

Problématique

<table>
<tr><th></th><th>HPC</th><th>Physique</th></tr>
<tr><td>Structures de donnée</td><td><pre><code>float matrice[300][40];
float vecteur[1200];</code></pre></td><td><pre><code>struct Composites;

namespace CompositeTags {
  // Tag types
  struct PID : Event::int_field {};
  struct nDoF : Event::int_field {};
  struct Chi2 : Event::float_field {};
  // Child relations -- two int columns per child.
  struct ChildIndices
      : Event::vector_field<Event::index2D_field<
            std::tuple<ChargedBasics const, NeutralBasics const, Composites const>, std::index_sequence<6, 24>>> {};
  struct ChildRels : public Event::vector_field<Event::ints_field<2>> {
    enum { IndexOffset = 0, ZipFamilyOffset, NumColumns };
  };
  /** Descendant unique IDs -- one int column per descendant.
   *
   * By descendant we mean any decay product (including intermediate
   * states). e.g. For Bs0 -> J/psi(-> mu+ mu-) K*0(892)(-> K+ pi-) the
   * J/psi has as descendants (mu+, mu-) and the Bs0 has (J/psi, mu+, mu-,
   * K*0(892), K+, pi-).
   *
   * Unique IDs must be created using LHCb::UniqueIDGenerator (unless for test
   * and development scripts).
   */
  struct DescendantUniqueIDs : Event::vector_field<Event::int_field> {};
  // Position vector (3D)
  struct Position : Event::Vec_field<3> {};
  // Position covariance (3D symmetric)
  struct PositionCov : Event::MatSym_field<3> {};
  // Four-momentum (4D...)
  struct FourMom : Event::Vec_field<4> {};
  // Four-momentum covariance (4D symmetric)
  struct FourMomCov : Event::MatSym_field<4> {};
  // Position-four-momentum covariance (4x3)
  struct PositionFourMomCov : Event::Mat_field<4, 3> {};

  template <typename T>
  using base_t = Event::SOACollection<T, PID, nDoF, Chi2, Position, PositionCov, FourMom, FourMomCov,
                                      PositionFourMomCov, ChildIndices, ChildRels, DescendantUniqueIDs>;
} // namespace CompositeTags</code></pre></td></tr>
<tr><td>Représentation mémoire</td><td>f32 | f32 | f32 | f32 | f32 | f32 | f32 | f32 | f32 | f32</td><td>?</td></tr>
</table>

- Stocker efficacement les structures issues de la physique ?
- Généralisation au delà ?

Structures de Données Classiques

- **Array of Structure** (AoS)
  - Localité spatiale par structure
  - Accès par index puis par champ
- **Structure of Arrays** (SoA)
  - Localité spatiale par champs
  - Accès par champ puis par index
- **Tiled Structure of Arrays**
  - Localité des données mixte
  - Indexage complexe, accès par tuile puis dans la tuile

Example

```
struct pixel
{
    int r, g, b;
};
std::array<pixel,10> aos{};
```

```
for(int i=0;i<10;++i)
    print("{}", aos.r[i]);
```

```
template<std::size_t N>
struct pixels
{
    std::array<int,N> r, g, b;
};
pixels<10> soa{};
```

```
for(int i=0;i<10;++i)
    print("{}", soa[i].r);
```

```
constexpr std::size_t simd_width = 2;
struct block
{
    std::array<int,simd_width> r, g, b;
};
std::array<block,5> aosoa{};
```

```
for(int i=0;i<5;++i) for(int j=0;j<simd_width;++j)
    print("{}", aosoa[i].r[j]);
```

Observations

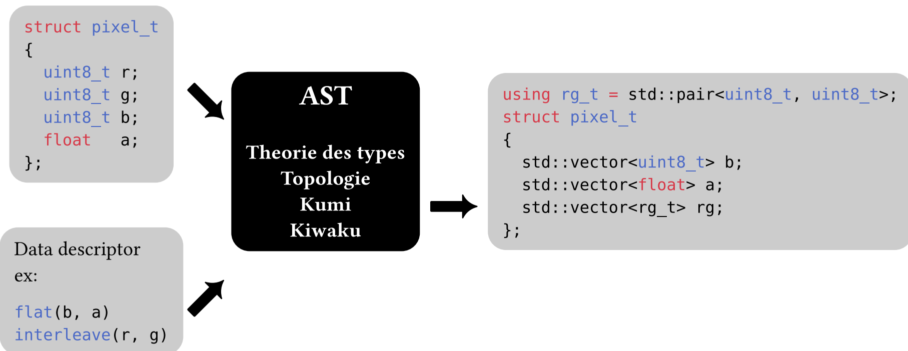
- Champs fixes : tuple(R,G,B)

- Espace similaire :
  - Logique : définition utilisateur
  - Physique : realisation en mémoire

$$nb_slots = \sum composants * \sum pixels = 3 * nb_pixels$$

- Définition/représentation mémoire différente
  - Fonction d'accès aux champs/éléments variable

Processus



- Comment manipuler une structure définie par l'utilisateur ?
- Comment rendre la structure finale utilisable ?

Une affaire de topologie

- **Variété topologique** : Espace abstrait défini par l'utilisateur
- **Fibré** : Forme canonique de la structure
- **Cartes** : Description de l'organisation interne
  - Organisation mémoire d'un sous-ensemble des données
- **Atlas** : Description totale de l'espace

L'aplatissage

- Passage sous forme canonique
  - Les champs sont fusionnés et les tableaux propagés
- Une structure est composée de :
  - Scalaires
  - Tableaux → constructeur de type
  - Tuples → type produit
  - Structs → type record

```
using info = std::pair<char,int>;              struct tracks {
struct hits {                                      std::vector<float> momentum;
    std::vector<info> data;          →             std::vector<std::vector<char>> hit_collection.data.0;
};                                                 std::vector<std::vector<int>> hit_collection.data.1;
struct tracks {                                };
    std::vector<hits> hit_collection;
    std::vector<float> momentum;
};
```

Considération des spécifications

- Les spécifications de l'utilisateur sont considérées
  - Les champs sont reindexés
  - Les tableaux réattribués
  - Spécification valide si isomorphisme

```
constexpr auto layout =
interleave(
    tracks.momentum
    tracks.hit_collection.data.1
)
```

→

```
using type = std::pair<float, std::vector<int>>
auto names = std::pair{"momentum", "hit_collection.data.1"};
struct chart1 {
    std::vector<type> combined;
    std::pair<string,string> id = names;
}
```

Traiter les champs manquants

- Les champs non précisés par l'utilisateur sont traités

```
struct chart1{
    float momentum;
    std::vector<int> hit.collection.data.1;
};

struct final {
    std::vector<chart1> chart;
    std::vector<std::vector<char>> hit_collection.data.0;
};
```

Finalement

- On construit la description totale de l'espace
  - Indice + champ logique → Elément physique

```
template<typename T> constexpr auto resolve(std::string path, T indexes)
{
    if ( path == "hit.collection.data.0" )
        return this.hit_collection.data.0[get<0>(indexes)][get<1>(indexes)];
    else if ( path == "momentum" )
        return this.chart[get<0>(indexes)].momentum;
    else
        return this.chart[get<0>(indexes)].hit_collection.data.1[get<1>(indexes)];
};

auto item = my_struct.resolve("hit_collection.data.1", std::pair{0, 0});
```

Tout ceci est résolu à la compilation, l'organisation d'une structure ne peut dépendre de paramètres non connus statiquement.

Kiwaku

Bibliothèque de tableaux multidimensionnels

Deux composants principaux :

- kwk::view et kwk::table (notion d'*ownership*)

Des options pour définir les propriétés

Options	Valeurs
`kwk::source`	pointeurs, tableaux statiques, ranges contigues
`kwk::size`	`kwk::of_size(dn...)`, entiers : 1D
`kwk::label`	une valeur sous forme de string
`kwk::allocator`	un type modélisant `kwk::concepts::allocator`

Example

Hadamard.cpp

```cpp
9   void hadamard(auto &C0, auto const& C1, auto const& C2)
10  {
11    using namespace kwk;
12
13    auto shape = of_size(30,10);
14    auto va = view{ source = C0, shape };
15    auto vb = view{ source = C1, shape };
16    auto vc = view{ source = C2, shape };
17
18    kwk::for_each([](auto &a, auto const& b, auto const &c)
19    {
20      a = b * c;
21    }, va, vb, vc);
22  }
23
24
25
26
27
28
29
30
```

clang 20 (Editor #1) — clang 20 — -std=c++20 -march=skylake-avx512 -O3 -DNDEBUG

```asm
30  vmovups ymm0, ymmword ptr [r14 + rax - 112]
31  vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 112]
32  vmovups ymmword ptr [rbx + rax - 112], ymm0
33  vmovsd xmm0, qword ptr [r14 + rax - 80]
34  vmovsd xmm1, qword ptr [r15 + rax - 80]
35  vmulps xmm0, xmm0, xmm1
36  vmovlps qword ptr [rbx + rax - 80], xmm0
37  vmovups ymm0, ymmword ptr [r14 + rax - 72]
38  vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 72]
39  vmovups ymmword ptr [rbx + rax - 72], ymm0
40  vmovsd xmm0, qword ptr [r14 + rax - 40]
41  vmovsd xmm1, qword ptr [r15 + rax - 40]
42  vmulps xmm0, xmm0, xmm1
43  vmovlps qword ptr [rbx + rax - 40], xmm0
44  vmovups ymm0, ymmword ptr [r14 + rax - 32]
45  vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 32]
46  vmovups ymmword ptr [rbx + rax - 32], ymm0
47  vmovsd xmm0, qword ptr [r14 + rax]
48  vmovsd xmm1, qword ptr [r15 + rax]
49  vmulps xmm0, xmm0, xmm1
50  vmovlps qword ptr [rbx + rax], xmm0
```

Travail en cours

- **kwk::tiles**
  - Permet de tuiler un espace
  - Chaque tuile possède son pattern d'itération
- **Entrelacement**
  - Notion d'ordre mémoire pour les champs d'une structure
  - Notion de hiérarchie
- **Données hétérogènes**
  - Types hétérogènes (tuples)
  - Tailles hétérogènes (*jagged arrays*)
- **Contextes**
  - Composition via une construction **with**

Preuve de concept

- Définition habituelle de la structure
- Réorganisation facile à définir

```cpp
using namespace kumi;
using namespace kwk;

constexpr auto pixel = record{ "r"_f = 0, "g"_f = 0, "b"_f = 0, "a"_f = 0.0f };
using pixel_map      = std::vector<pixel>;

constexpr auto layout = interleave("g"_f, "a"_f);
// constexpr auto layout = interleave("r"_f, "b"_f);

int main(int argc, char *argv[])
{
    auto pixels = structure(pixel_map{}, layout, argc);
}
```

L'accès aux données

- Accès uniforme entre différentes organisations
- Algorithme indépendant de l'organisation physique
- Garantit un accès aux données efficace

```cpp
auto to_grayscale = [](auto &elt){
    auto [r,g,b,a] = elt;
    int gray = static_cast<int>(0.2126f * r + 0.7152f * g + 0.0722f * b);
    r = g = b = gray;
    return elt;
};

int main()
{
  auto context    = kwk::standard;
  auto rng        = pixels.as_range(context);
  std::ranges::transform(rng, rng.begin(), to_grayscale);
}
```

Supports de différents contextes

- Kiwaku supporte des contextes d'exécution
  - simd : EVE pour la vectorisation
  - kiwaku : Algorithmes natifs de Kiwaku
  - standard : Fonctions de la STL

- Différents contextes → algorithmes de parcours spécifiques

```cpp
auto context = kwk::simd;
auto context = kwk::kiwaku;
auto context = kwk::standard;
```

> A terme Kiwaku vise à supporter d'autres contextes, par exemple un contexte *multi-thread* par dessus un contexte SIMD

Hiérarchisation

- Un contexte par niveau de hiérarchie

```
map(mpi{MPI_COMM_WORLD}[openmp{4}],
  [](auto const& i)
  {
    auto o = kwk::table{i.extent()};
    auto n = i.size();
    with( sycl{}.when(n > 10'000).otherwise(simd{}),
      [](auto& oo, auto const& ii)
      {
        auto x = count_if(ii, [](auto e){ return e > 4; });
        map([&](auto &e){ return e - x; }, oo, ii );
      }
      , o,  i["x"_f]
    )
  }, out, in
);
```

Conclusion : Apports de Kiwaku

- Abstraction haut niveau n'impactant pas les performances
- Changement d'organisation physique simplifié
- Problématiques optimisations/algorithmes mieux séparées
- Simplification de l'écriture de code générique
- Performance portable et equivalente à un code spécialisé

Perspectives

- Collections internes de tailles hétérogènes (*jagged arrays*)
- Adaptation/écriture des différents algorithmes de parcours
- Support des types somme (unions, std::variant)
- Automatiser la sélection de layout ? (réflexion C++26)

Avec la réflexion C++26

```cpp
struct rgb {
    int r, g, b;
    float alpha;
    struct meta {
        int id; char value;
    } infos;
} my_rgb;

int size   = 42;
auto pixels = kwk::table(my_rgb, size);

auto kernel = [](auto &elt){
    auto [r, g, b, ...] = elt;
    int gray = static_cast<int>(0.2126f * r + 0.7152f * g + 0.0722f * b);
    r = g = b = gray;
    return elt;
}

eve::algo::transform_inplace(pixels.as_range(kwk::simd), kernel);
```

Merci de votre attention

- Kiwaku : https://github.com/jfalcou/kiwaku/
- Kumi : https://github.com/jfalcou/kumi/