RÉCONCILIER ERGONOMIE ET PERFORMANCE EN C++20 : EXEMPLE D'UNE BIBLIOTHÈQUE DE TABLEAUX MULTIDIMENSIONNELS

26 SEPTEMBRE 2024 *15E JOURNÉES INFORMATIQUES IN2P3/IRFU*

Sylvain JOUBE

Joël FALCOU, Hadrien GRASLAND, David CHAMONT.

LISN LABORATOIRE INTERDISCIPLINAIRE

IJCLab Irène Joliot-Curie

Problématiques de calcul pour la PHE : Ergonomie vs Performance

Problèmes d'accessibilité et d'ergonomie...

- Comment diminuer le temps de dév. des non experts en calcul (ex : physiciens) ?
- Comment garantir une maintenance simple par les scientifiques ?
- Comment s'abstraire de la complexité du matériel ?

...ou problèmes de performance

- Python : performances limitées

C++ : une semi-solution devenue complète ?

Zero Cost Abstraction via plusieurs mécanismes : *templates*, *constexpr*...

Mais les *templates* ont plusieurs problèmes...

- Difficiles à prendre en main et à écrire
- Messages d'erreurs longs et cryptiques
- Coût de compilation (CPU, RAM)
- Binaires énormes

Gains ergonomiques avec C++20

- *constexpr*
- *concepts C++*

Mon sujet de recherche

GPU et Portabilité des Performances - Approches et Applications hétérogènes

Aspects théoriques

Contraintes fines appliquées à la programmation générique...
=> Amélioration de l'ergonomie ?
=> Conservation des performances ?

Aspects pratiques

- Progrès ergonomiques récents pour C++ (normes 11, 14, 17, 20)
- Les nouveautés C++ permettent-elles un meilleur compromis ?
- Illustration via Kiwaku, une bibliothèque de tableaux nD
- *Programmation générique, méta-programmation, programmation générative*

Kiwaku

Caractéristiques techniques

- Bibliothèque de gestion de tableaux nD en C++20
- Allier performance et ergonomie
- Contexte hétérogène

Fonctionnalités principales de Kiwaku *actuelles et à venir*

- Algorithmes de parcours performants (row-major, column-major, Morton, Hilbert, ...)
- Opérations de base (fill, generate, iota, copy, find, count_if, replace_if, ...)
- Interface avec d'autres bibliothèques (algèbre linéaire, ...)
- Compatible avec diverses architectures (CPU, CPU vectoriel, CPU parallèle, GPU, distribué, ...)

Intérêt par rapport à std::md_span ?

std::md_span

- Extension de std::span (C++20)
- Dans le standard C++23
- Uniquement une aide pour le calcul des positions nD ↔ mémoire linéaire
- Vue sur la mémoire (ne possède pas la mémoire)

Kiwaku

- Tables et vues
- Algorithmes de base sur ses conteneurs
- Interface avec des bibliothèques de calcul
- Ergonomie inspirée de Numpy

Exemples simples

Numpy

```
in = np.arange(0, d0, 1, dtype=float)

out = in * 8 + 4
```

Kiwaku

```
auto in   = kwk::table{ kwk::of_size(d0), kwk::as<float> };
auto out  = kwk::table{ kwk::of_size(d0), kwk::as<float> };

// Initialisation du tableau
kwk::iota(in);

// Calcul élément par élément
kwk::for_each([](auto const& i, auto& o) { o = i * 8 + 4; }, in, out);
```

Contextes d'exécution Kiwaku

Caractéristiques principales

- Un contexte Kiwaku pour chaque architecture
- Interface commune, partagée par tous les contextes
- Ajout possible par les utilisateurs : ne recoder que le strict minimum

Décision de design

- Réutiliser un maximum le code déjà écrit :
  - Algorithmes chaînés (exemple : `none_of` → `!any_of` → `reduce` → `map`)
  - Fonction `for_each` / `map` à la base de quasiment tous les algos

- Support d'une nouvelle architecture => redéfinir `map` pour cette architecture

Contexte SYCL

Design général

- Le contexte SYCL redéfinit map et map_index
- Certains algorithmes sont sous-optimaux => redéfininition d'algorithmes spécifiques. Exemples :
  - `none_of` → `!any_of` → `reduce` spécialisé pour SYCL ~~-> `map`~~
  - `copy` spécialisé pour SYCL ~~-> `transform` → `map`~~

Proxy

- Optimisation SYCL : directionnalité des transferts de données
- Proxy pour l'accès aux conteneurs Kiwaku (lecture, écriture, lecture + écriture)
- Gestion de la mémoire via les buffers/accesseurs

Contextes Kiwaku : built-in (cpu, sycl)

CPU / SYCL

```
// Initialisation
kwk::iota(in);

// Exécution par défaut...
kwk::for_each(          [](auto const& i, auto& o) { o = i * 8 + 4; }, in, out);

// ...ou sur CPU...
kwk::for_each(kwk::cpu,                   ^ idem ^                            );

// ...ou via SYCL
kwk::for_each(kwk::sycl,                  ^ idem ^                            );
```

Contextes Kiwaku : chaînage

Sur chaque noeud MPI, exécuter un noyau SYCL : version manuelle (Kokkos-like)

```
// Lambda SYCL
auto process_tile_sycl =
  [](auto const& tile_in, auto& tile_out) {
    kwk::for_each( kwk::sycl
                 , [](auto const& i, auto& o) { o = i * 8 + 4; }
                 , tiles(tile_in , of_size(m,m))
                 , tiles(tile_out , of_size(m,m))
                 );
  };

// MPI -> SYCL
kwk::for_each( kwk::mpi
             , process_tile_sycl
             , tiles(kwk_table_in , of_size(n,n))
             , tiles(kwk_table_out , of_size(n,n))
             );
```

Contextes Kiwaku : chaînage

Sur chaque noeud MPI, exécuter un noyau SYCL : version raccourcie

```
// Chaînage MPI -> SYCL
kwk::for_each( kwk::mpi(kwk::sycl)
             , [](auto concepts::view const& i, auto concepts::view & o) { o = i * 8 + 4; }
             , tiles(tiles(kwk_table_in , of_size(n,n)), of_size(m,m))
             , tiles(tiles(kwk_table_out, of_size(n,n)), of_size(m,m))
             );
```

Performance

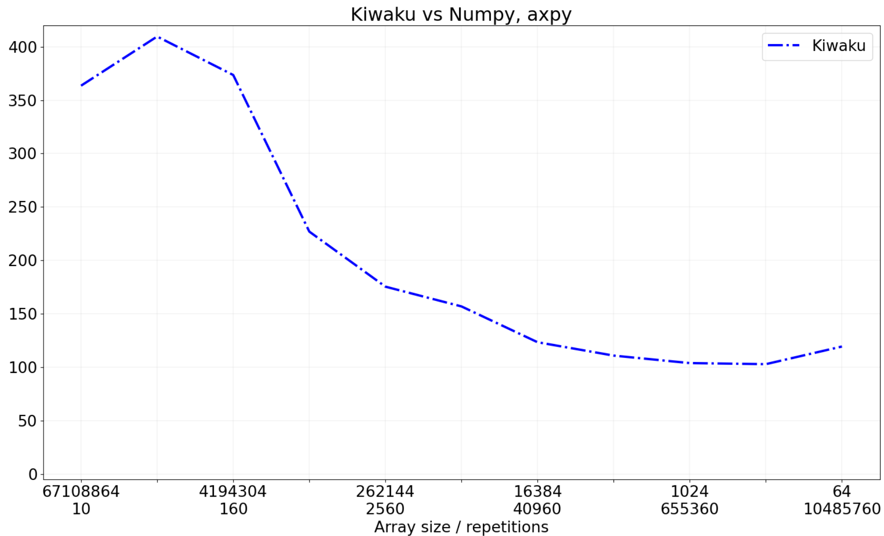
C++

```
auto tab   = kwk::table{ kwk::of_size(/*taille du tableau*/), kwk::as<float> };
kwk::iota(in);

// Démarrage timer
for (std::size_t i = 0; i < /*nombre de répétitions*/; ++i)
  kwk::for_each([](auto& e) { e = e * 8 + 4; }, tab);
// Arrêt timer
```

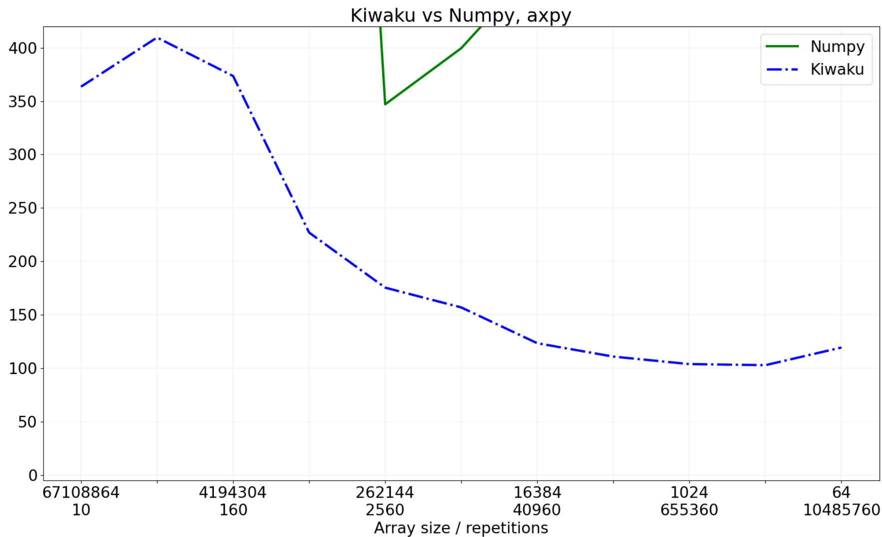
Numpy

```
tab = np.arange(0, '''taille du tableau''', 1, dtype=float)

# Démarrage timer
for i in range('''nombre de répétitions'''):
  tab = 8 * tab + 4
# Arrêt timer
```

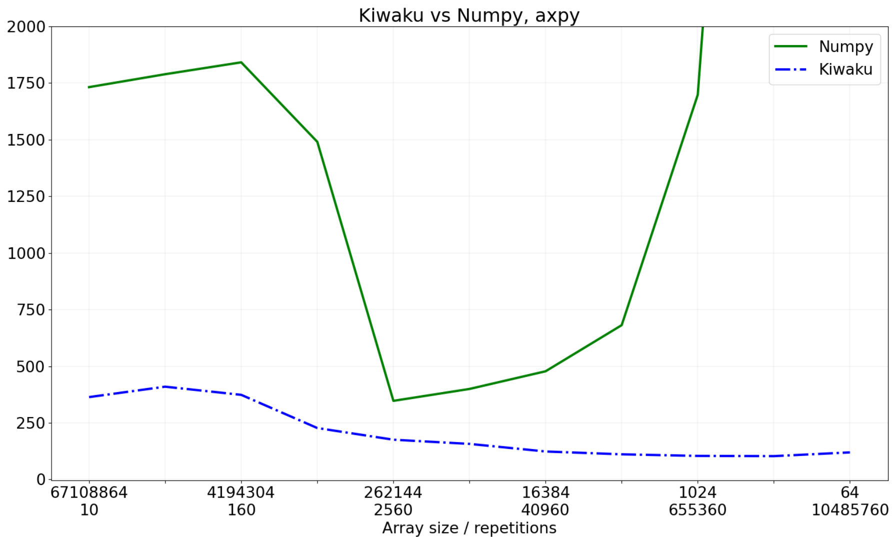
Performance : Kiwaku seul



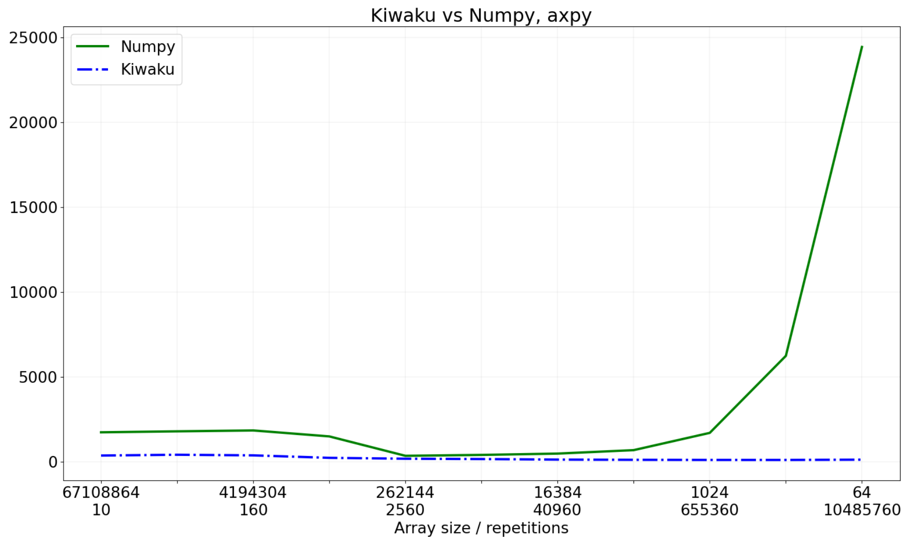
Performance : Kiwaku + Numpy



Performance : Kiwaku + Numpy



Performance : Kiwaku + Numpy



Merci !

- https://github.com/jfalcou/kiwaku
- sylvain.joube@ijclab.in2p3.fr
- https://sjoube.net/