Introduction à Kokkos

Du Tiling manuel au Tiling automatique

M1 CSMI

Rappel : Le Problème du Tiling

Dans le cours précédent (OpenMP & Tiling), nous avons vu que pour la performance :

1. Il faut utiliser le cache efficacement (localité spatiale et temporelle).
2. Cela oblige à écrire des boucles imbriquées complexes ($ii, jj, kk, ...$).
3. Le code devient moins lisible et difficile à maintenir.

Le problème : Si on veut passer sur GPU, il faut tout réécrire (CUDA), changer les layouts mémoire, et gérer les transferts.

Kokkos est une librairie C++ qui résout ce problème par l'abstraction.

- **Vous écrivez** : L'équation mathématique (le kernel) et l'espace d'itération (le domaine).
- **Kokkos gère** :
  - Le Tiling (via `MDRangePolicy`).
  - Le Layout (disposition mémoire) adapté au matériel.
  - Le parallélisme (OpenMP/CUDA) à la compilation.

Philosophie : On écrit le code **une seule fois** ("Single Source").

Pour le TP3, nous allons résoudre l'équation des ondes 2D avec un schéma "Saute-Mouton" (Leapfrog) :

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^{n} + u_{i,j}^{n-1}}{\Delta t^2} = c^2 \Delta u_{i,j}^{n}$$

Ce qui donne l'algorithme de mise à jour :

$$u_{i,j}^{n+1} = 2u_{i,j}^{n} - u_{i,j}^{n-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 \left(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}\right) + \ldots$$

Comment implémenter cela efficacement sur CPU **et** GPU ?

Au lieu de `std::vector` ou `double**`, on utilise `Kokkos::View`.

```
Kokkos::View<float**, Kokkos::HostSpace>  u_host("u_h", nx, ny); // CPU
Kokkos::View<float**, Kokkos::CudaSpace> u_dev("u_d", nx, ny);   // GPU
```

Layout Automatique :

- Sur CPU, Kokkos choisit `LayoutRight` (Ligne par ligne) → Cache friendly.
- Sur GPU, Kokkos choisit `LayoutLeft` (Colonne par colonne) → Coalesced access.

Vous n'avez plus à inverser vos boucles `i,j` manuellement !

2. Le Calcul : Tiling Automatique

Au lieu d'écrire 4 boucles imbriquées pour le tiling, on utilise `MDRangePolicy`.

```
// 2D Range Policy: itère sur un domaine 2D (de {0,0} à {nx,ny})
using Policy = Kokkos::MDRangePolicy<Kokkos::Rank<2>>;

Kokkos::parallel_for("leapfrog", Policy({0, 0}, {nx, ny}),
  KOKKOS_LAMBDA(int i, int j) {
    // Écrire ici l'équation locale pour le point (i,j)
    // Kokkos se charge de découper (i,j) en tuiles (tiles) adaptées au GPU/
CPU
    u_next(i,j) = 2*u(i,j) - u_prev(i,j) + ... ;
});
```

Kokkos optimise la taille des tuiles (`TeamSize`, `VectorLength`) pour vous.

Rappel C++ : Les Lambdas

Une **lambda** est une fonction locale sans nom, très utilisée en C++ moderne.

Syntaxe : `[capture] (arguments) { corps }`

Exemple :

```cpp
int a = 1;      // Variable externe (capturée par valeur)
int b = 2;      // Variable externe (capturée par référence)

// [a, &b] : a est copiée, b est référencée
auto f = [a, &b] (int i) {
    int c = 3;  // Variable locale
    b = a + c + i;
};

f(10); // b vaut maintenant 1 + 3 + 10 = 14
```

Raccourcis : `[=, &b]` capture tout par valeur sauf `b`. `[&, a]` capture tout par référence sauf `a`. **(Le défaut `&` ou `=` doit toujours être le premier).**

Rappel C++ : Modes de Capture

La **capture** définit comment la lambda accède aux variables extérieures.

- `[&]` (**Par référence**) : Accès à la mémoire originale de la variable.
  - **Danger GPU** : La mémoire du CPU (stack) est invisible pour le GPU → **Crash**.
- `[=]` (**Par valeur**) : Crée une copie de la variable pour la lambda. **Requis pour Kokkos** : Les valeurs sont copiées vers le device (GPU).
- Dans Kokkos, on utilise `KOKKOS_LAMBDA` (macro pour créer une lambda avec le mode de capture `[=]`).

Note importante : Copier une `Kokkos::View` par valeur (`[=]`) est **correct et performant**. On copie juste le "pointeur", pas les données.

Attention : Le CPU et le GPU ont (souvent) des mémoires séparées.

- **Deep Copy** : Pour bouger les données, il faut être explicite.

```
Kokkos::deep_copy(u_dev, u_host); // Envoi vers le GPU
// ... Calcul Kernel sur GPU ...
Kokkos::deep_copy(u_host, u_dev); // Retour vers le CPU (pour sauvegarde/
affichage)
```

Règle d'or : Minimiser les copies. On initialise sur GPU, on calcule sur GPU, on ne ramène que le résultat final (ou une image).

Piège : Classes et GPU

Dans le TP3, vous aurez peut-être une classe `Grid2D`.

```
class WaveGrid2DSolver {
  Kokkos::View<float**> u; // Donnée membre
  void solve() {
    auto u_local = u; // 1. COPIE LOCALE OBLIGATOIRE
    Kokkos::parallel_for(..., KOKKOS_LAMBDA(int i, int j) {
       // 2. Utiliser u_local, PAS u (qui impliquerait 'this')
       u_local(i,j) = ... ;
    });
  }
};
```

Si vous capturez `this` par erreur, le GPU essaiera de lire l'objet `Grid2D` qui est dans la RAM du CPU $\rightarrow$ Crash (Segfault).