Rust et Vulkan

Hadrien Grasland 2023-07-10

Introduction

- En physique des particules, on utilise principalement…
  - C++ si la performance d'exécution est importante
  - Python pour les interfaces haut niveau
  - CUDA (ou une couche d'abstraction *mince*) pour le GPU
- Dans cette présentation, je vais discuter de…
  - Rust comme alternative à C++
  - Vulkan comme alternative à CUDA, OpenCL, SYCL…
- Quelques raccourcis car j'ai 20min… A vos questions !

Rust

Carte d’identié

- Langage stabilisé récemment (v1 en 2015)
- **Créneau similaire à C++**
  - Orienté compilation AoT, sans ramasse-miettes
  - Typage strict et explicite
  - Contrôle bas niveau, métaprogrammation
  - Vocabulaire d’abstraction riche
  - Apprentissage long
- Voyons maintenant ce qui les distingue…

Compatibilité

- Le C++ est fortement **contraint par la compatibilité**
  - Avec lui-même (ex : std::vector<bool>, copie par défaut…)
  - Avec le C (ex : entiers de taille indéfinie, litérales typées…)
- Rust **s'affranchit** de cette contrainte
  - Types entiers de taille fixe sauf isize/usize + FFI vers le C
  - Déplacement par défaut + références et pointeurs si il faut
  - 0 n'est pas 32 bits, 0.0 n'est pas de double précision…

Comportement indéfini en C++

- En C++, le comportement indéfini (UB) est omniprésent :
  - **Nombres entiers** → Overflow*, shift > bits*, -INT_MIN, *cast*…
  - **Utilisation de tableau** → Accès hors borne
  - **Mémoire non initialisée** → Lecture indéfinie
    (y compris **destructeurs** : affectation, exceptions…)
  - **Boucles infinies** → Violation du théorème de Fermat
  - **Pointeurs, références** → Peut être nul ou invalide
- Le compilo suppose que ça n'arrive pas → **Effet imprévisible** !

* Enfin corrigé en C++20 si j'ai bien suivi. Plus qu'à attendre que la norme soit largement adoptée…

Sûreté en Rust

- En Rust, hors blocs *unsafe*, **pas de comportement indéfini** :
  - **Typage :** Les valeurs respectent les invariants de type
  - **Mémoire :** Références valides, mémoire initialisée
  - **Threads :** Pas d'accès concurrents non synchronisés
- En pratique, c'est un **bon compromis**
  - Besoin *d'unsafe* : tout n'est pas prouvable à la compilation
  - MAIS rarement requis au quotidien
  - Utilisation localisée → Code facile à auditer

Faiblesses de typage en C++

- **Comportements imprévus et erreurs incompréhensibles** résultant de l'interaction entre…
  - Conversions implicites
  - Surcharge de fonctions + arguments par défaut
  - Généricité via les *templates* + spécialisation
  - Méthodes virtuelles
- **Pas de vérification de typage*** dans les *templates*
  - Feeling comparable au *duck typing* de Python, Julia

* Je reviens sur la question des concepts dans un instant.

Typage fort en Rust

- Comportement **beaucoup plus prévisible**
  - Pas de conversion implicite
  - On ne supporte N types différents que via les *traits*
  - L'utilisation des *traits* est explicite
- **Généricité contrainte** par les *traits** :
  - Le code générique spécifie de quoi il a besoin
  - Utiliser autre chose est une erreur de compilation
  - A comparer aux concepts C++20 où c'est OK…

* Je détaille un peu plus loin.

Polymorphisme bancal en C++

- Deux mécanismes **complètement incompatibles**
  - Héritage et méthodes virtuelles
  - *Templates*, concepts et spécialisation
- Passer de l'un à l'autre est **très difficile**
  - Souvent requis : compromis coût compilation/exécution
  - Nécessite de tout réapprendre + réécrire beaucoup de code
- **Interopérabilité difficile** avec les types extérieurs

Traits en Rust*

- **Même formalisme** en polymorphisme statique & dynamique
  - Méthodes sans données, comme les interfaces Java
    + constantes et types associés
    + méthodes et types génériques
    + implémentations par défaut
  - Mais certaines choses ne sont supportées qu'en statique
- Implémentable pour un **objet extérieur** → Interopérabilité !

* L'idée a été popularisée par Haskell, la version Rust est un peu différente.

Erreurs chaotiques en C++

- Historiquement, fort accent sur les **exceptions**
  - Lancement/capture très coûteux en performances
  - Très difficile d'écrire du code correct en cas d'*unwinding*
  - Utilisation découragée dans les destructeurs → Que faire ?
- Hors de ce mécanisme, c'est **la jungle**
  - Valeurs spéciales ou int que personne ne lit, comme en C
  - Types exotiques spécifiques à chaque projet

Gestion d'erreurs en Rust

- Au quotidien, type variant **Result<T, E>**
  - Contient soit un résultat de type T, soit une erreur de type E
  - Pour accéder au résultat, il faut gérer l'erreur éventuelle
- ***Panic* pour les erreurs fatales** (ex : echec d'une assertion)
  - Implémenté soit comme une exception, soit abort() direct
  - Capture possible, mais rare et peu encouragée
- Fort **consensus communautaire** + erreurs bien documentées

Métaprogrammation

- En C++, on a beaucoup de ***template metaprogramming***
  - Basé sur un bug de compilateur anobli (SFINAE)
  - Accessible seulement à quelques initiés
  - Produit du code très difficile à maintenir
  - Alternative : Générateurs de code externes* (ex : ROOT)
- En Rust, outre les traits, on utilise beaucoup des **macros**
  - Parsing/génération de code intégré, assez facile à utiliser

* …avec ce que ça implique en termes de complexité du processus de construction.

Compilation et dépendances

- En C++, on a **CMake** et les paquets des distributions Linux
  - Insatisfaction quasi-unanime vis à vis de ces outils
  - Résultat : Peu de réutilisation de code
- En Rust, on a **cargo** en standard
  - Combine gestion de compilation + dépendances
  - Simple pour un projet de petite taille, passe bien à l'échelle
  - Gestion des dépendances triviale → Partage de code !

Rust du présent = C++ du futur ?

- **C++17** vu de **Rust v1 (2015)**
  - **De nombreux rattrapages :**
    filesystem, any, optional, string_view, byte, aligned_alloc
  - **Des équivalents moins ergonomiques :**
    std::tuple, structures bindings, CTAD, std::variant
- Même tendance en **C++20** :
  - **Rattrapages :** Ranges, <=>, consteval, { .a }, format, span, endian, <bit>, barrier, latch, jthread, assume_aligned
  - **Equivalents douteux :** Coroutines, modules, concepts

En conclusion

- 2 bonnes raisons de lancer un projet C++ en 2023
  - Lié à un gros code C++ existant qu'on ne veut pas réécrire
  - Bibliothèques C++ plus matures pour votre domaine
- Si vous n'êtes pas dans ces cas, **essayez Rust**
  - Rare d'être bloqué par un écart de fonctionnalité langage
  - Ergonomie infiniment supérieure
  - Moins de debug → Plus de fonctionnalités & optimisations
  - Moins obscur et mieux supporté que C++2x !

Vulkan

Audience

- Vous voulez **coder pour GPU en optimisant beaucoup**
  - Besoin d'accès fin aux fonctionnalités matérielles
  - Les couches d'abstraction ne doivent pas les cacher
- Vous avez besoin de **portabilité** entre OSs, matériels…
- Vous exigez une certaine **qualité/maturité** de l'implémentation
  - Installation/maintenance facile, en dev et en prod
  - Pas trop de bugs, d'APIs brisées tous les quelques mois…
- Une **API moins ergonomique** est un compromis acceptable

Pourquoi une API graphique ?

- Conçues pour permettre de **tirer le maximum** d'un GPU
  - Expose tout ce qu'on trouve dans une API calcul, et plus
- **Audience** jeux vidéos, CAO >> Calculs GPGPU
  - Installation facile, moins de bugs, API rarement changée
  - Outillage abondant (mais parfois trop orienté 3D)
- Certaines APIs conçues pour la **portabilité** OS/matériel
  - Bonnes perfs possibles à >95 % de code partagé
  - Grande audience → Les fabricants *doivent* les supporter

Pourquoi pas OpenGL ?

- **Conception obsolète** héritée de IrisGL de SGI (1992)
  - Pensé pour du matériel très différent des GPU modernes
  - Faible support du parallélisme (etat global implicite)
- Evolution limitée par la **rétro-compatibilité**
  - N façons de faire la même chose, dont beaucoup à éviter
  - Implémentations « futées » ⟶ Imprévisibles et buggées
- **Perte de vitesse** des évolutions en faveur de Vulkan

Vulkan vs OpenGL

- Maintenu par les mêmes auteurs (groupe Khronos)
- **Suppression** de nombreuses fonctionnalités obsolètes
- Pas d'état global → **Parallélisme** grandement facilité
- **Meilleur contrôle** de l'ordonnancement et des allocations
- Meilleur support de **mémoire unifiée**, **dallage**, **multi-GPU**
- **Pilote simplifié** → Moins de bugs, moins de surprises
- Contrepartie : **Très lourd** → Abstraire selon ses besoins !

Pourquoi pas WebGPU ?

- API **graphique bas niveau** (type Vulkan, Metal, Direct3D 12)
- Backends pour les 3 grandes APIs → **Support HW/OS optimal**
- Prévue pour le Web → **API simplifiée, pas d'UB possible**
- MAIS très jeune → **Fonctionnalités réduites, API mouvante**
- Ma recommandation future quand elle aura maturé ?
  - En attendant, Vulkan + un peu d'abstraction fait le travail

En résumé

	Support OS / Installation			Support matériel			API	
	Windows	macOS	Linux	Nvidia	AMD	Intel	Facile	Complet
CUDA	✔	✘	⚠	✔	✘	✘	✔	✔
HIP	?	✘	⚠	⚠	✔	✘	✔	✔
SYCL	⚠	✘	⚠	⚠	⚠	✔	✔	⚠
OpenCL	⚠	💀	⚠	💀	⚠	⚠	⚠	✘
Direct3D	✔	✘	⚠	✔	✔	✔	✘	✔
Metal	✘	✔	✘	💀	💀	💀	⚠	✔
OpenGL	⚠	💀	✔	✔	✔	✔	⚠	⚠
Vulkan	⚠	⚠	✔	✔	✔	✔	✘	✔
WebGPU	✔	✔	✔	✔	✔	✔	⚠	⚠

Conclusion générale

- D’un côté, on a une API graphique bas niveau
  - Support matériel excellent
  - Utilisation difficile car riche en UB
- De l’autre, un langage qui rend le bas niveau ergonomique
- Essayons de combiner les deux !

Merci de votre attention !