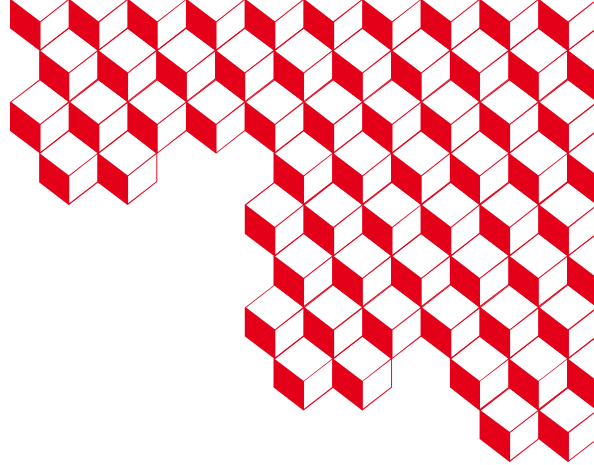


Programmation Rust et GPU

Exploration des capacités de Rust pour le calcul scientifique GPGPU

Journée des Stagiaires, 29 août 2023

Gabriel Dos Santos encadré par Cédric Chevalier
CEA/DAM DIF

Parcours académique

- 2018 – 2021 : Licence Informatique

- 2021 – 2023 : Master Calcul Haute Performance, Simulation
  Parcours Infomatique HPC

Sommaire

1. Introduction

Objectifs du stage

Déterminer la viabilité de Rust pour la programmation GPGPU

1. Établir un état de l'art exhaustif pour la programmation GPU en Rust
   - Support au niveau du langage
   - Bibliothèques et frameworks
   - Investiguer les limites de la détection de bogues par le compilateur
2. Évaluer les performances des méthodes disponibles pour la génération de code GPU
3. Portage d'algorithmes d'une application CEA sur GPU NVIDIA

Objectifs du stage

Déterminer la viabilité de Rust pour la programmation GPGPU

1. Établir un état de l'art exhaustif pour la programmation GPU en Rust
   - Support au niveau du langage
   - Bibliothèques et frameworks
   - Investiguer les limites de la détection de bogues par le compilateur
2. Évaluer les performances des méthodes disponibles pour la génération de code GPU
3. Portage d'algorithmes d'une application CEA sur GPU NVIDIA

Objectifs du stage

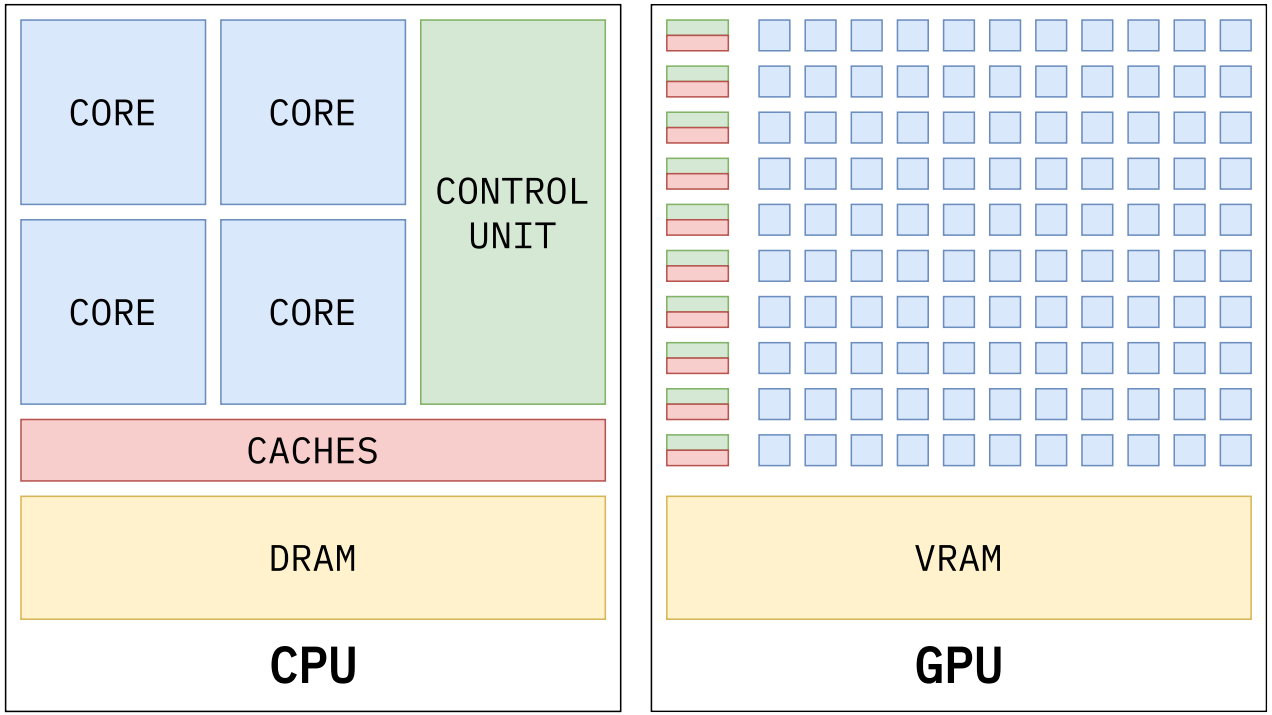
Déterminer la viabilité de Rust pour la programmation GPGPU

1. Établir un état de l'art exhaustif pour la programmation GPU en Rust
   - Support au niveau du langage
   - Bibliothèques et frameworks
   - Investiguer les limites de la détection de bogues par le compilateur
2. Évaluer les performances des méthodes disponibles pour la génération de code GPU
3. Portage d'algorithmes d'une application CEA sur GPU NVIDIA

Graphics Processing Unit

Qu'est-ce qu'un GPU ?

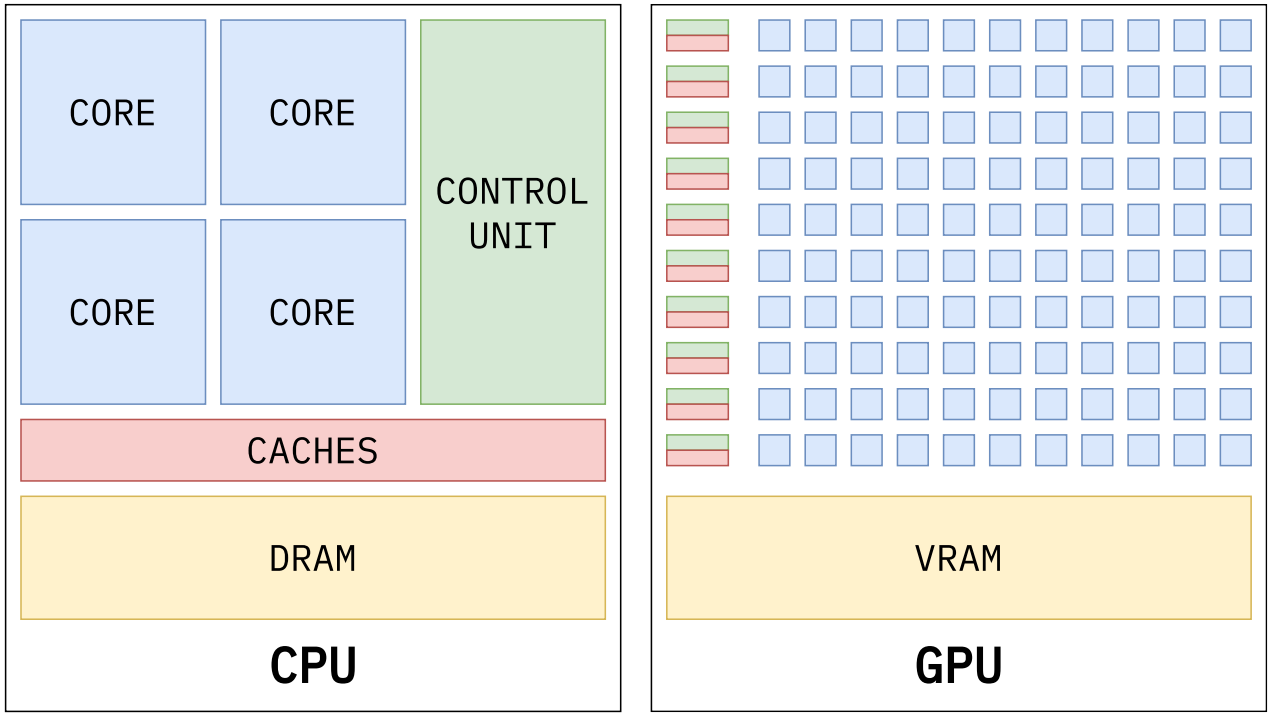
- Accélérateur matériel
  - Dédié à certains types de calculs
- Nombre de cœurs très élevé
- Transfert des données nécessaires entre mémoire CPU (*host*) et mémoire GPU (*device*)



Graphics Processing Unit

Qu'est-ce qu'un GPU ?

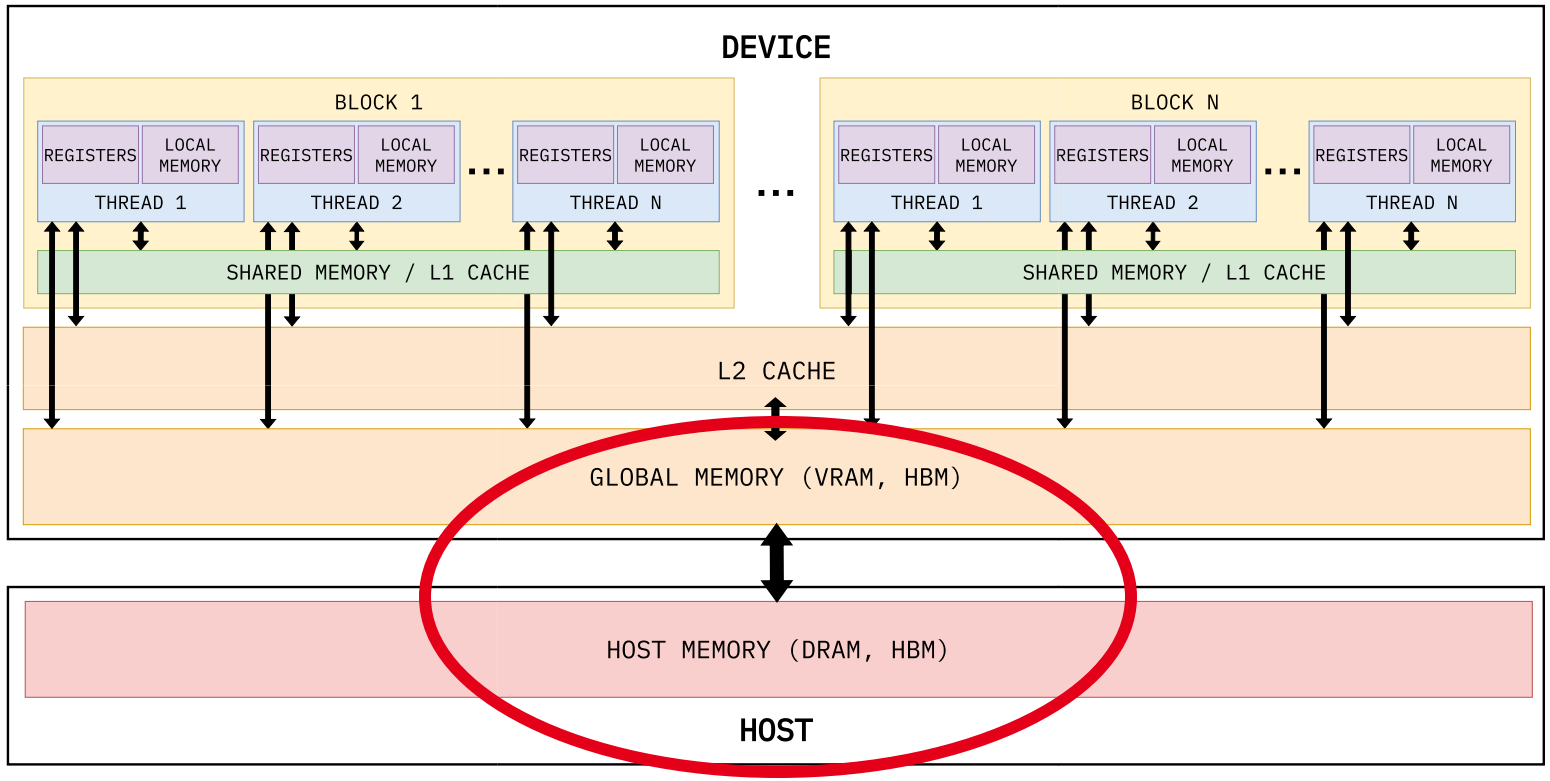
- Accélérateur matériel
  - Dédié à certains types de calculs
- Nombre de cœurs très élevé
- Transfert des données nécessaires entre mémoire CPU (*host*) et mémoire GPU (*device*)



Graphics Processing Unit

Qu'est-ce qu'un GPU ?

- Accélérateur matériel
  - Dédié à certains types de calculs
- Nombre de cœurs très élevé
- Transfert des données nécessaires entre mémoire CPU (*host*) et mémoire GPU (*device*)



Graphics Processing Unit

Quelle utilité en HPC ?

- Grand nombre de cœurs favorise la parallélisation
  - Idéal pour le calcul scientifique (algèbre linéaire)
  - Offre des performances accrues

- Performance crête des supercalculateurs modernes dépend des GPUs
  - Exemple : Frontier, #1 au TOP 500 (juin 2023)
    ≈98% de la performance provient des GPUs

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703

Graphics Processing Unit

Quelle utilité en HPC ?

- Grand nombre de cœurs favorise la parallélisation
  - Idéal pour le calcul scientifique (algèbre linéaire)
  - Offre des performances accrues

- Performance crête des supercalculateurs modernes dépend des GPUs
  - Exemple : Frontier, #1 au TOP 500 (juin 2023)
    ≈98% de la performance provient des GPUs

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703

Graphics Processing Unit

Quelle utilité en HPC ?

- Grand nombre de cœurs favorise la parallélisation
  - Idéal pour le calcul scientifique (algèbre linéaire)
  - Offre des performances accrues

- Performance crête des supercalculateurs modernes dépend des GPUs
  - Exemple : Frontier, #1 au TOP 500 (juin 2023)
    ≈98% de la performance provient des GPUs

→ **Programmation efficace des GPUs cruciale pour la performance**

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703

Modèles de programmation

Comment programmer un GPU pour du calcul scientifique ?

Différents modèles de programmation :

- Haut niveau :
  - OpenMP
  - SYCL
  - Kokkos
- Bas niveau :
  - NVIDIA CUDA
  - AMD HIP/ROCm
  - OpenCL

Modèles de programmation

Comment programmer un GPU pour du calcul scientifique ?

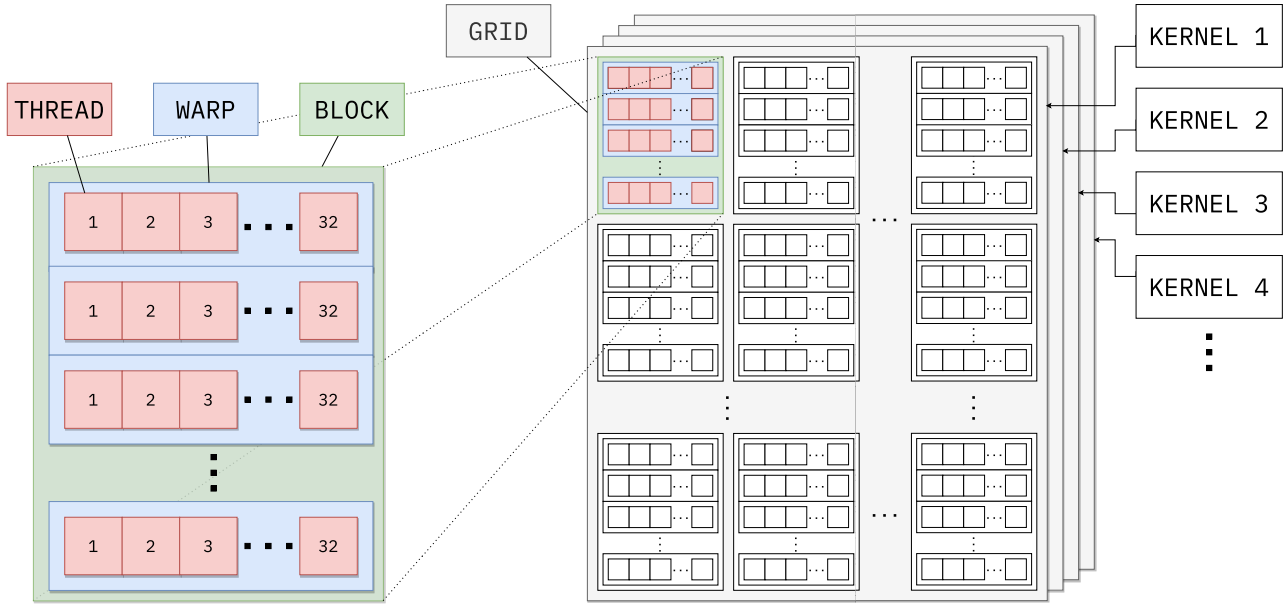
Différents modèles de programmation :

- Haut niveau :
  - OpenMP
  - SYCL
  - Kokkos
- Bas niveau :
  - NVIDIA CUDA
  - AMD HIP/ROCm
  - OpenCL

Modèles de programmation

Programmation GPU bas niveau

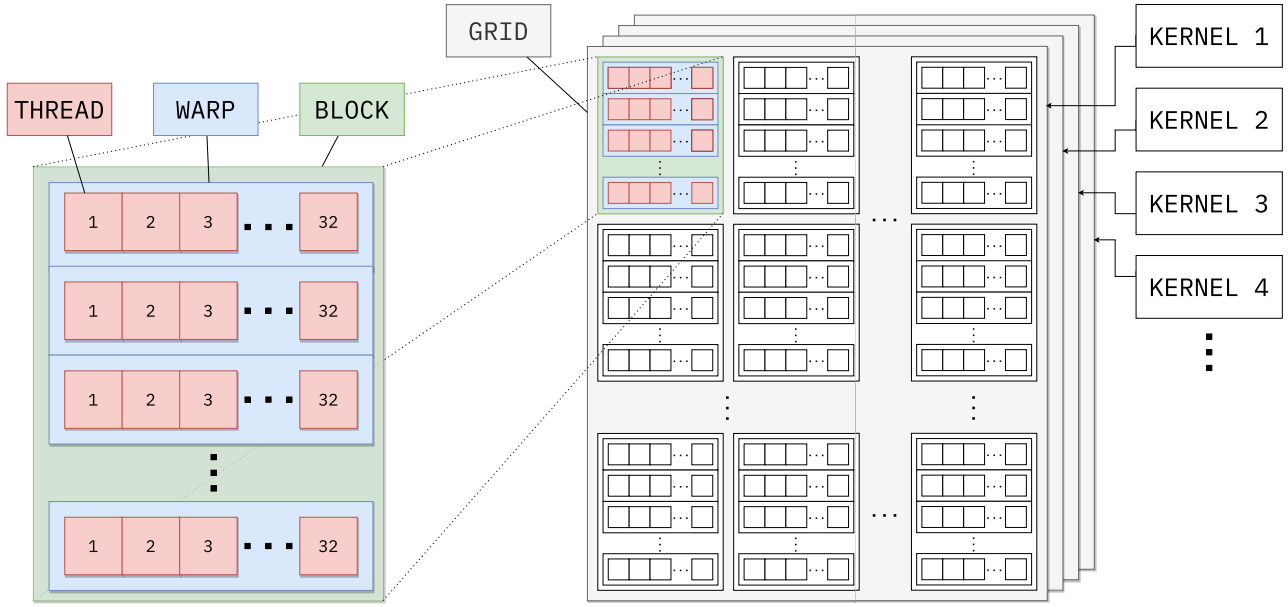
- **Kernel** = fonction qui s'éxecute sur GPU
- Abstraction hiérarchique du matériel :
  - Grilles
  - Blocs
  - Threads



Modèles de programmation

Programmation GPU bas niveau

- **Kernel** = fonction qui s'éxecute sur GPU
- Abstraction hiérarchique du matériel :
  - Grilles
  - Blocs
  - Threads



Langage Rust

Qu'est-ce que Rust ?

- Langage conçu pour la **programmation système** :
  - **Performance :** langage compilé, pas de ramasse-miette, typage statique fort
  - **Sécurité / sûreté :** code, mémoire, typage, parallélisme
  - **Concurrence** (parallélisme)

The Rust Programming Language

Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

1. Chaque valeur a un *propriétaire*

```
1 // `a` possède la valeur "Hello, world!"
2 let a = String::from("Hello, world!");
```

Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

1. Chaque valeur a un *propriétaire*

```
1 // `a` possède la valeur "Hello, world!"
2 let a = String::from("Hello, world!");
```

2. Une valeur ne peut avoir qu'un seul propriétaire à la fois

```
1 let a = String::from("Hello, world!");
2 // `a` cède la propriété de "Hello, world!" à `b`
3 let b = a;
```

Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

1. Chaque valeur a un *propriétaire*

```
1 // `a` possède la valeur "Hello, world!"
2 let a = String::from("Hello, world!");
```

2. Une valeur ne peut avoir qu'un seul propriétaire à la fois

```
1 let a = String::from("Hello, world!");
2 // `a` cède la propriété de "Hello, world!" à `b`
3 let b = a;
```

3. Une valeur est automatiquement désallouée lorsque son propriétaire est hors de portée

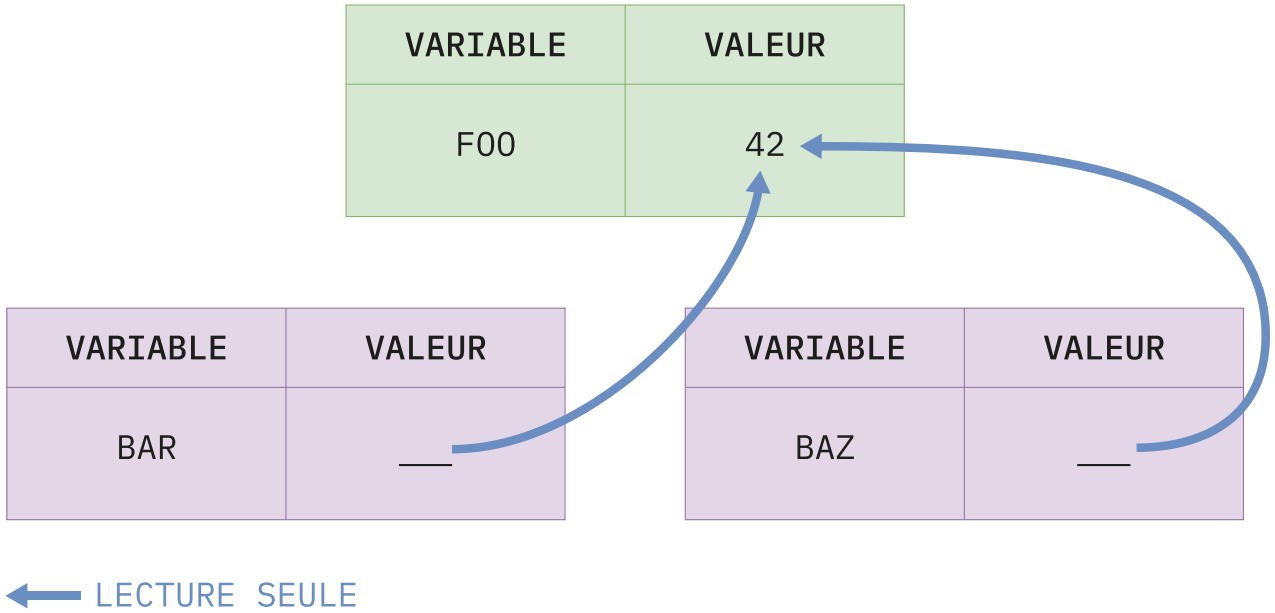
```
1 {
2     let a = String::from("Hello, world!");
3 }
4 // La valeur "Hello, world!" est automatiquement désallouée
5 // car son propriétaire (`a`) est hors de portée
```

Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

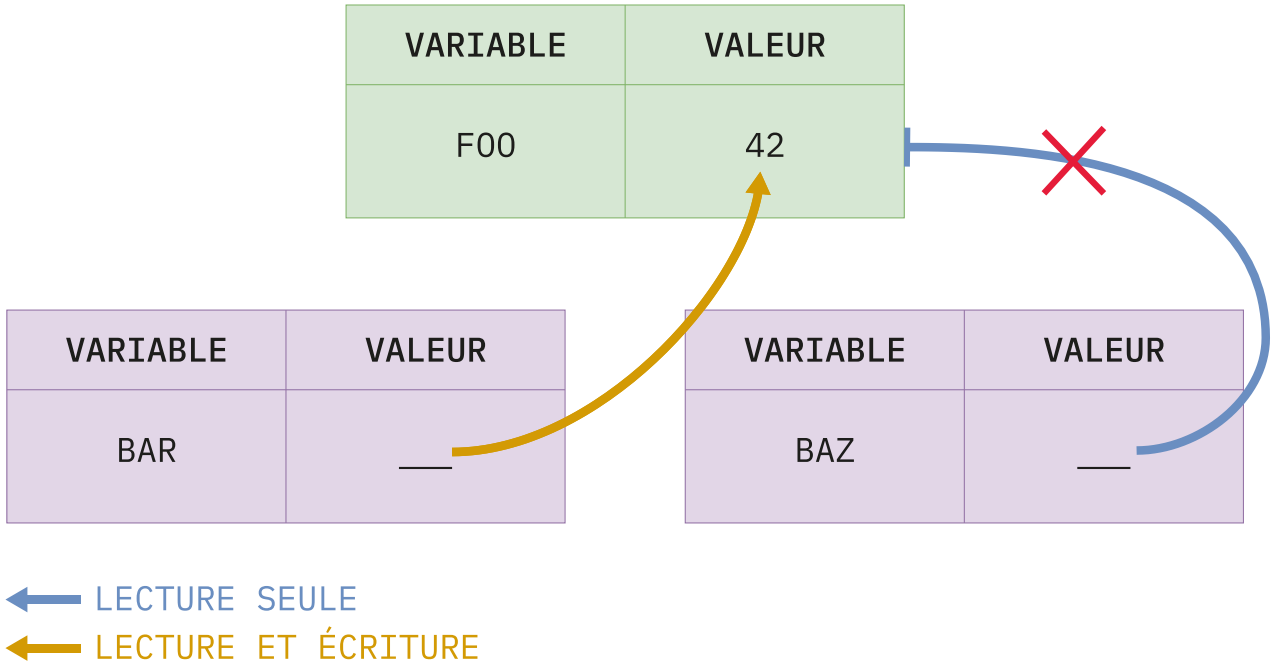


Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)



Langage Rust

Pourquoi Rust en programmation scientifique ?

Gestion de la mémoire innovante :

1. Principe d'***ownership*** (propriété)
2. Principe de ***borrowing*** (emprunt)

→ Suppression à la **compilation** de nombreux bogues relatifs à la **mémoire** et au **parallélisme**

Problématique

Les besoins du HPC d'aujourd'hui et de demain

- Avenir du HPC : les GPUs
  - Nécessité d'exploiter efficacement leur architecture
  - Repousser les limites du calcul scientifique
- Codes massivement parallèles : Rust, candidat idéal
  - Accent sur la performance
  - Exactitude des codes garantie par le compilateur

Problématique

Les besoins du HPC d'aujourd'hui et de demain

- Avenir du HPC : les GPUs
  - Nécessité d'exploiter efficacement leur architecture
  - Repousser les limites du calcul scientifique
- Codes massivement parallèles : Rust, candidat idéal
  - Accent sur la performance
  - Exactitude des codes garantie par le compilateur

Problématique

Les besoins du HPC d'aujourd'hui et de demain

- Avenir du HPC : les GPUs
  - Nécessité d'exploiter efficacement leur architecture
  - Repousser les limites du calcul scientifique
- Codes massivement parallèles : Rust, candidat idéal
  - Accent sur la performance
  - Exactitude des codes garantie par le compilateur

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

2. Contributions

État de l'art : Support natif à Rust

Quelles sont les possibilités natives au langage ?

Cible officiellement supportée par rustc : **architecture NVIDIA "nvptx64"**

- Fonctions et primitives de base pour l'écriture de kernels en Rust

État de l'art : Support natif à Rust

Quelles sont les possibilités natives au langage ?

Cible officiellement supportée par rustc : **architecture NVIDIA "nvptx64"**

- Fonctions et primitives de base pour l'écriture de kernels en Rust

En pratique : **Inutilisable**

- Compilation des kernels produit des exécutables invalides
- Pas de support pour les abstractions disponibles en Rust standard :
  - Nécessité d'utiliser des **pointeurs bruts**
  - Dépendance intrinsèque aux **blocs de code *unsafe***
  - Équivalent à CUDA C++ : **pas de garanties du compilateur Rust**
- Aucun moyen de lancer l'exécution des kernels sur GPU nativement

État de l'art : Support natif à Rust

Quelles sont les possibilités natives au langage ?

Cible officiellement supportée par rustc : **architecture NVIDIA "nvptx64"**

- Fonctions et primitives de base pour l'écriture de kernels en Rust

En pratique : **Inutilisable**

- Compilation des kernels produit des exécutables invalides
- Pas de support pour les abstractions disponibles en Rust standard :
  - Nécessité d'utiliser des **pointeurs bruts**
  - Dépendance intrinsèque aux **blocs de code *unsafe***
  - Équivalent à CUDA C++ : **pas de garanties du compilateur Rust**
- Aucun moyen de lancer l'exécution des kernels sur GPU nativement

État de l'art : Support natif à Rust

Quels sont les possibilités natives au langage ?

Cible officiellement supportée par rustc : **architecture NVIDIA "nvptx64"**

- Fonctions et primitives de base pour l'écriture de kernels en Rust

En pratique : **Inutilisable**

- Compilation des kernels produit des exécutables invalides
- Pas de support pour les abstractions disponibles en Rust standard :
  - Nécessité d'utiliser des **pointeurs bruts**
  - Dépendance intrinsèque aux **blocs de code *unsafe***
  - Équivalent à CUDA C++ : **pas de garanties du compilateur Rust**
- Aucun moyen de lancer l'exécution des kernels sur GPU nativement

État de l'art : Support natif à Rust

Quels sont les possibilités natives au langage ?

Cible officiellement supportée par rustc : **architecture NVIDIA "nvptx64"**

- Fonctions et primitives de base pour l'écriture de kernels en Rust

En pratique : **Inutilisable**

- Compilation des kernels produit des exécutables invalides
- Pas de support pour les abstractions disponibles en Rust standard :
  - Nécessité d'utiliser des **pointeurs bruts**
  - Dépendance intrinsèque aux **blocs de code *unsafe***
  - Équivalent à CUDA C++ : **pas de garanties du compilateur Rust**
- Aucun moyen de lancer l'exécution des kernels sur GPU nativement

→ Approche **non retenue**

État de l'art : Langages de *shading*

Un bon choix pour la programmation GPGPU en Rust ?

Approche la plus populaire pour la programmation GPU en Rust :

- Plusieurs *crates* activement maintenus : wgpu-rs, Arrayfire, …

État de l'art : Langages de *shading*

Un bon choix pour la programmation GPGPU en Rust ?

Approche la plus populaire pour la programmation GPU en Rust :

- Plusieurs *crates* activement maintenus : wgpu-rs, Arrayfire, …

Cependant :

- Destiné aux applications graphiques (*rendering*, interface web, jeux vidéos)
- Nécessité d'écrire les kernels en langage de *shading*
  - Validation de l'exactitude des kernels par rustc impossible
  - Manque de contrôle bas niveau
  - Pas adapté au calcul scientifique

État de l'art : Langages de *shading*

Un bon choix pour la programmation GPGPU en Rust ?

Approche la plus populaire pour la programmation GPU en Rust :

- Plusieurs *crates* activement maintenus : wgpu-rs, Arrayfire, …

Cependant :

- Destiné aux applications graphiques (*rendering*, interface web, jeux vidéos)
- Nécessité d'écrire les kernels en langage de *shading*
  - Validation de l'exactitude des kernels par rustc impossible
  - Manque de contrôle bas niveau
  - Pas adapté au calcul scientifique

État de l'art : Langages de *shading*

Un bon choix pour la programmation GPGPU en Rust ?

Approche la plus populaire pour la programmation GPU en Rust :

- Plusieurs *crates* activement maintenus : wgpu-rs, Arrayfire, …

Cependant :

- Destiné aux applications graphiques (*rendering*, interface web, jeux vidéos)
- Nécessité d'écrire les kernels en langage de *shading*
  - Validation de l'exactitude des kernels par rustc impossible
  - Manque de contrôle bas niveau
  - Pas adapté au calcul scientifique

→ Approche **non retenue**

État de l'art : OpenCL

API OpenCL : *bindings*

Plusieurs *crates* implémentant l'API OpenCL 3 en Rust :

- Gestion de l'environnement OpenCL :
  - Plateformes, *devices*, *program-queues*, kernels
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - API idiomatique : **code très compact** par rapport à OpenCL en C/C++

État de l'art : OpenCL

API OpenCL : *bindings*

Plusieurs *crates* implémentant l'API OpenCL 3 en Rust :

- Gestion de l'environnement OpenCL :
  - Plateformes, *devices*, *program-queues*, kernels
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - API idiomatique : **code très compact** par rapport à OpenCL en C/C++

État de l'art : OpenCL

IMPORTS
VARIABLES GLOBALES

SÉLECTION DU DEVICE

CONSTRUCTION DU PROGRAMME

CRÉATION DU CONTEXTE
CRÉATION DE LA QUEUE

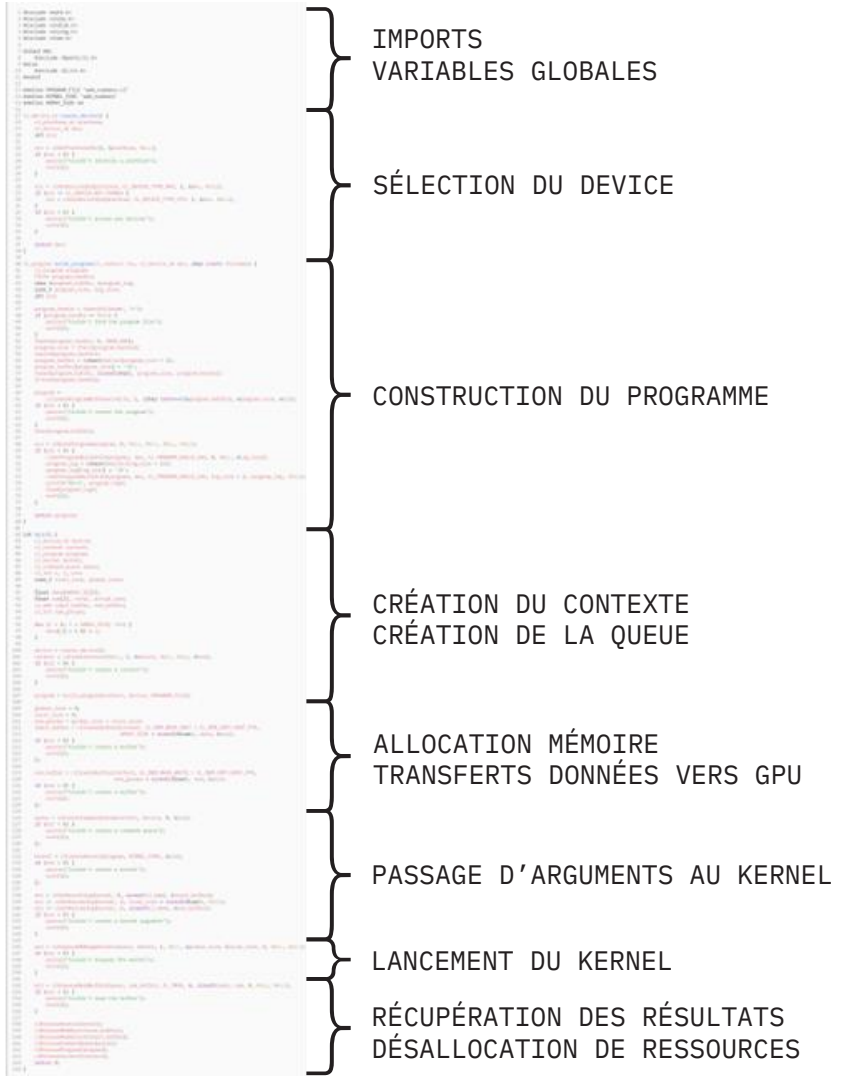
ALLOCATION MÉMOIRE
TRANSFERTS DONNÉES VERS GPU

PASSAGE D'ARGUMENTS AU KERNEL

LANCEMENT DU KERNEL

RÉCUPÉRATION DES RÉSULTATS
DÉSALLOCATION DE RESSOURCES

État de l'art : OpenCL



IMPORTS
VARIABLES GLOBALES

SÉLECTION DU DEVICE

CONSTRUCTION DU PROGRAMME

CRÉATION DU CONTEXTE
CRÉATION DE LA QUEUE

ALLOCATION MÉMOIRE
TRANSFERTS DONNÉES VERS GPU

PASSAGE D'ARGUMENTS AU KERNEL

LANCEMENT DU KERNEL

RÉCUPÉRATION DES RÉSULTATS
DÉSALLOCATION DE RESSOURCES

```rust
extern crate ocl;
use ocl::ProQue;

const ARRAY_SIZE: usize = 64;
static SRC: &str = r#"/* ... */"#;

fn main() -> ocl::Result<()> {
    let pro_que = ProQue::builder().src(SRC).dims(ARRAY_SIZE).build()?;
    let input = pro_que.create_buffer::<f32>()?;
    let d_output = pro_que.create_buffer::<f32>()?;

    let kernel = pro_que
        .kernel_builder("fma")
        .arg(&input)
        .arg(10.0f32)
        .arg(&d_output)
        .build()?;

    unsafe {
        kernel.enq()?;
    }

    let mut h_output = vec![0.0f32; d_output.len()];
    d_output.read(&mut h_output).enq()?;

    Ok(())
}
```

165 lignes ➜ 27 lignes

État de l'art : OpenCL

API OpenCL : *bindings*

Plusieurs *crates* implémentant l'API OpenCL 3 en Rust :

- Gestion de l'environnement OpenCL :
  - Plateformes, *devices*, *program-queues*, kernels
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - API idiomatique : **code très compact** par rapport à OpenCL en C/C++

Limites :

- Nécessité d'écrire les **kernels en OpenCL C**
  - Détection par compilateur de bogues dans les kernels impossible
- Multiples implémentations des *bindings* :
  - **APIs Rust incompatibles** entre *crates*

État de l'art : OpenCL

API OpenCL : *bindings*

Plusieurs *crates* implémentant l'API OpenCL 3 en Rust :

- Gestion de l'environnement OpenCL :
  - Plateformes, *devices*, *program-queues*, kernels
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - API idiomatique : **code très compact** par rapport à OpenCL en C/C++

Limites :

- Nécessité d'écrire les **kernels en OpenCL C**
  - Détection par compilateur de bogues dans les kernels impossible
- Multiples implémentations des *bindings* :
  - **APIs Rust incompatibles** entre *crates*

État de l'art : OpenCL

API OpenCL : *bindings*

Plusieurs *crates* implémentant l'API OpenCL 3 en Rust :

- Gestion de l'environnement OpenCL :
  - Plateformes, *devices*, *program-queues*, kernels
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - API idiomatique : **code très compact** par rapport à OpenCL en C/C++

Limites :

- Nécessité d'écrire les **kernels en OpenCL C**
  - Détection par compilateur de bogues dans les kernels impossible
- Multiples implémentations des *bindings* :
  - **APIs Rust incompatibles** entre *crates*

➜ Approche **retenue**

État de l'art : CUDA

Projet Rust-CUDA : *Toolchain* pour le support de CUDA en Rust

- Implémentation de kernel en Rust via **nouveau *backend* compilateur rustc**
- Intégration de kernels en CUDA C++ (compatible bibliothèques NVIDIA)
- Abstractions classiquement disponibles en CUDA :
  - Coordonnées des threads
  - Primitives de synchronisation
  - Gestion de mémoire partagée
  - *Intrinsics* et support pour la programmation en assembleur PTX
- Gestion de l'environnement CUDA
  - Séléction des *devices*, création de *streams*, lancement de kernels, etc...
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - Désallocation automatique des ressources

État de l'art : CUDA

Projet Rust-CUDA : *Toolchain* pour le support de CUDA en Rust

- Implémentation de kernel en Rust via **nouveau *backend* compilateur rustc**
- Intégration de kernels en CUDA C++ (compatible bibliothèques NVIDIA)
- Abstractions classiquement disponibles en CUDA :
  - Coordonnées des threads
  - Primitives de synchronisation
  - Gestion de mémoire partagée
  - *Intrinsics* et support pour la programmation en assembleur PTX
- Gestion de l'environnement CUDA
  - Séléction des *devices*, création de *streams*, lancement de kernels, etc...
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - Désallocation automatique des ressources

État de l'art : CUDA

Projet Rust-CUDA : *Toolchain* pour le support de CUDA en Rust

- Implémentation de kernel en Rust via **nouveau *backend* compilateur rustc**
- Intégration de kernels en CUDA C++ (compatible bibliothèques NVIDIA)
- Abstractions classiquement disponibles en CUDA :
  - Coordonnées des threads
  - Primitives de synchronisation
  - Gestion de mémoire partagée
  - *Intrinsics* et support pour la programmation en assembleur PTX
- Gestion de l'environnement CUDA
  - Séléction des *devices*, création de *streams*, lancement de kernels, etc...
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - Désallocation automatique des ressources

État de l'art : CUDA

Projet Rust-CUDA : *Toolchain* pour le support de CUDA en Rust

- Implémentation de kernel en Rust via **nouveau *backend* compilateur rustc**
- Intégration de kernels en CUDA C++ (compatible bibliothèques NVIDIA)
- Abstractions classiquement disponibles en CUDA :
  - Coordonnées des threads
  - Primitives de synchronisation
  - Gestion de mémoire partagée
  - *Intrinsics* et support pour la programmation en assembleur PTX
- Gestion de l'environnement CUDA
  - Séléction des *devices*, création de *streams*, lancement de kernels, etc...
  - Allocation sur GPU et transferts de données *host* <=> *device*
  - Désallocation automatique des ressources

État de l'art : CUDA

Projet Rust-CUDA : certaines limites

- Kernels écrits dans des **blocs de code *unsafe***
- **Utilisation de pointeurs bruts :**
  - Tableaux mutables
  - Zone de mémoire partagée
- **Non maintenu depuis fin 2021**
  - Projet non-officiel : ni Rust, ni NVIDIA
  - Selon propriétaire : travaux devraient être menés directement au sein de rustc
  - Incompatible avec les versions récentes du compilateur Rust
  - Support versions CUDA récentes : initialement incompatible avec CUDA 12, mais...
    Contributions *open-source* patchant le projet

État de l'art : CUDA

Projet Rust-CUDA : certaines limites

- Kernels écrits dans des **blocs de code *unsafe***
- **Utilisation de pointeurs bruts :**
  - Tableaux mutables
  - Zone de mémoire partagée
- **Non maintenu depuis fin 2021**
  - Projet non-officiel : ni Rust, ni NVIDIA
  - Selon propriétaire : travaux devraient être menés directement au sein de rustc
  - Incompatible avec les versions récentes du compilateur Rust
  - Support versions CUDA récentes : initialement incompatible avec CUDA 12, mais... Contributions *open-source* patchant le projet

État de l'art : CUDA

Projet Rust-CUDA : certaines limites

- Kernels écrits dans des **blocs de code *unsafe***
- **Utilisation de pointeurs bruts :**
  - Tableaux mutables
  - Zone de mémoire partagée
- **Non maintenu depuis fin 2021**
  - Projet non-officiel : ni Rust, ni NVIDIA
  - Selon propriétaire : travaux devraient être menés directement au sein de rustc
  - Incompatible avec les versions récentes du compilateur Rust
  - Support versions CUDA récentes : initialement incompatible avec CUDA 12, mais... Contributions *open-source* patchant le projet

État de l'art : CUDA

Projet Rust-CUDA : certaines limites

- Kernels écrits dans des **blocs de code *unsafe***
- **Utilisation de pointeurs bruts :**
  - Tableaux mutables
  - Zone de mémoire partagée
- **Non maintenu depuis fin 2021**
  - Projet non-officiel : ni Rust, ni NVIDIA
  - Selon propriétaire : travaux devraient être menés directement au sein de rustc
  - Incompatible avec les versions récentes du compilateur Rust
  - Support versions CUDA récentes : initialement incompatible avec CUDA 12, mais...
    Contributions *open-source* patchant le projet

État de l'art : CUDA

Projet Rust-CUDA : certaines limites

- Kernels écrits dans des **blocs de code *unsafe***
- **Utilisation de pointeurs bruts :**
  - Tableaux mutables
  - Zone de mémoire partagée
- **Non maintenu depuis fin 2021**
  - Projet non-officiel : ni Rust, ni NVIDIA
  - Selon propriétaire : travaux devraient être menés directement au sein de rustc
  - Incompatible avec les versions récentes du compilateur Rust
  - Support versions CUDA récentes : initialement incompatible avec CUDA 12, mais... Contributions *open-source* patchant le projet

→ Approche **retenue**

Évaluation de performances

HARP : Outil d'évaluation de performance de Rust sur GPU

HARP : *Hardware Accelerated Rust Profiling*

- Projet *open-source* pour le CEA
- Évaluation et comparaison de performances sur noyaux de calculs simples :
  - AXPY, GEMM, réduction
  - Simple et double précision
- Plusieurs implémentations des kernels :
  - Rust parallèle (via le *crate* rayon)
  - Rust OpenCL (via *bindings*)
  - Rust-CUDA (via kernels en Rust & en CUDA C++)
- Aggrégation et visualisation des résultats
  - Plusieurs métriques
  - Vérification de la stabilité des mesures

Évaluation de performances

HARP : Outil d'évaluation de performance de Rust sur GPU

HARP : *Hardware Accelerated Rust Profiling*

- Projet *open-source* pour le CEA
- Évaluation et comparaison de performances sur noyaux de calculs simples :
  - AXPY, GEMM, réduction
  - Simple et double précision
- Plusieurs implémentations des kernels :
  - Rust parallèle (via le *crate* rayon)
  - Rust OpenCL (via *bindings*)
  - Rust-CUDA (via kernels en Rust & en CUDA C++)
- Aggrégation et visualisation des résultats
  - Plusieurs métriques
  - Vérification de la stabilité des mesures

Évaluation de performances

HARP : Outil d'évaluation de performance de Rust sur GPU

HARP : *Hardware Accelerated Rust Profiling*

- Projet *open-source* pour le CEA
- Évaluation et comparaison de performances sur noyaux de calculs simples :
  - AXPY, GEMM, réduction
  - Simple et double précision
- Plusieurs implémentations des kernels :
  - Rust parallèle (via le *crate* rayon)
  - Rust OpenCL (via *bindings*)
  - Rust-CUDA (via kernels en Rust & en CUDA C++)
- Aggrégation et visualisation des résultats
  - Plusieurs métriques
  - Vérification de la stabilité des mesures

Évaluation de performances

HARP : Outil d'évaluation de performance de Rust sur GPU

HARP : *Hardware Accelerated Rust Profiling*

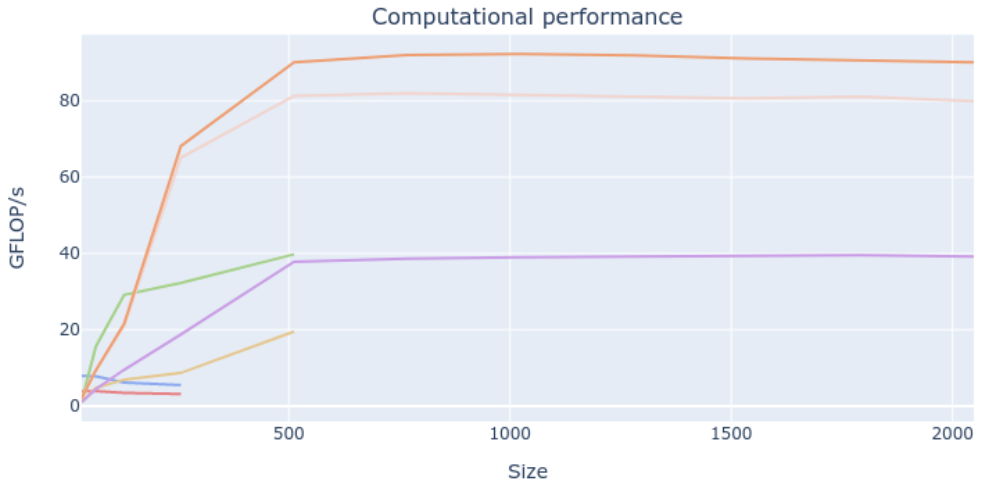
- Projet *open-source* pour le CEA
- Évaluation et comparaison de performances sur noyaux de calculs simples :
  - AXPY, GEMM, réduction
  - Simple et double précision
- Plusieurs implémentations des kernels :
  - Rust parallèle (via le *crate* rayon)
  - Rust OpenCL (via *bindings*)
  - Rust-CUDA (via kernels en Rust & en CUDA C++)
- Aggrégation et visualisation des résultats
  - Plusieurs métriques
  - Vérification de la stabilité des mesures

Évaluation de performances

HARP : Résultats et observations sur GEMM

- Implémentations CUDA >2x plus performantes qu'OpenCL :

Simple précision

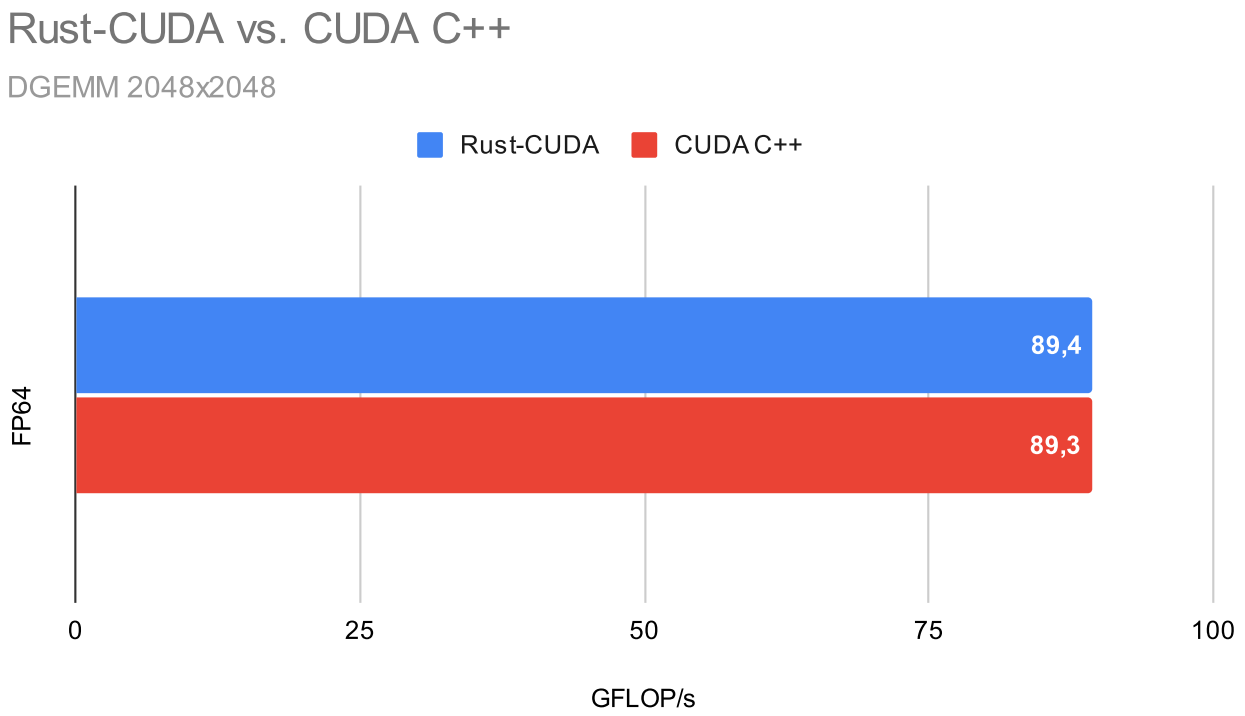


Double précision

Évaluation de performances

HARP : Résultats et observations sur GEMM

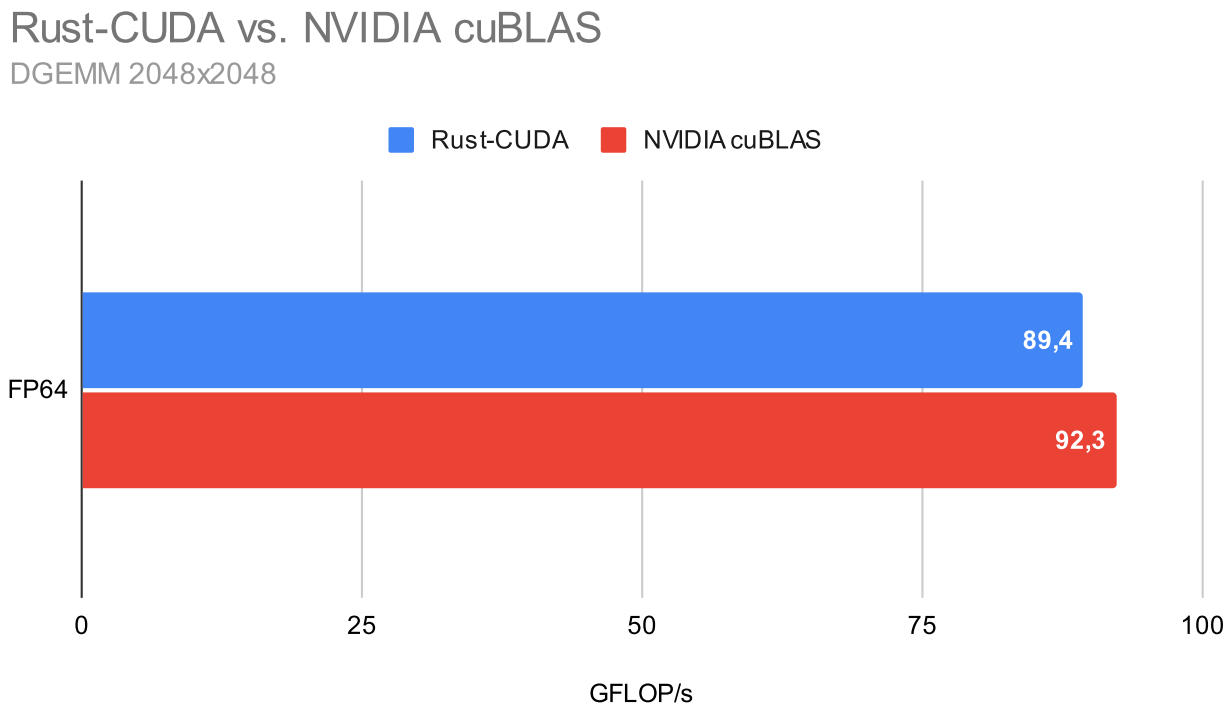
- Rust-CUDA : kernels Rust aussi performants que versions CUDA C++



Évaluation de performances

HARP : Résultats et observations sur GEMM

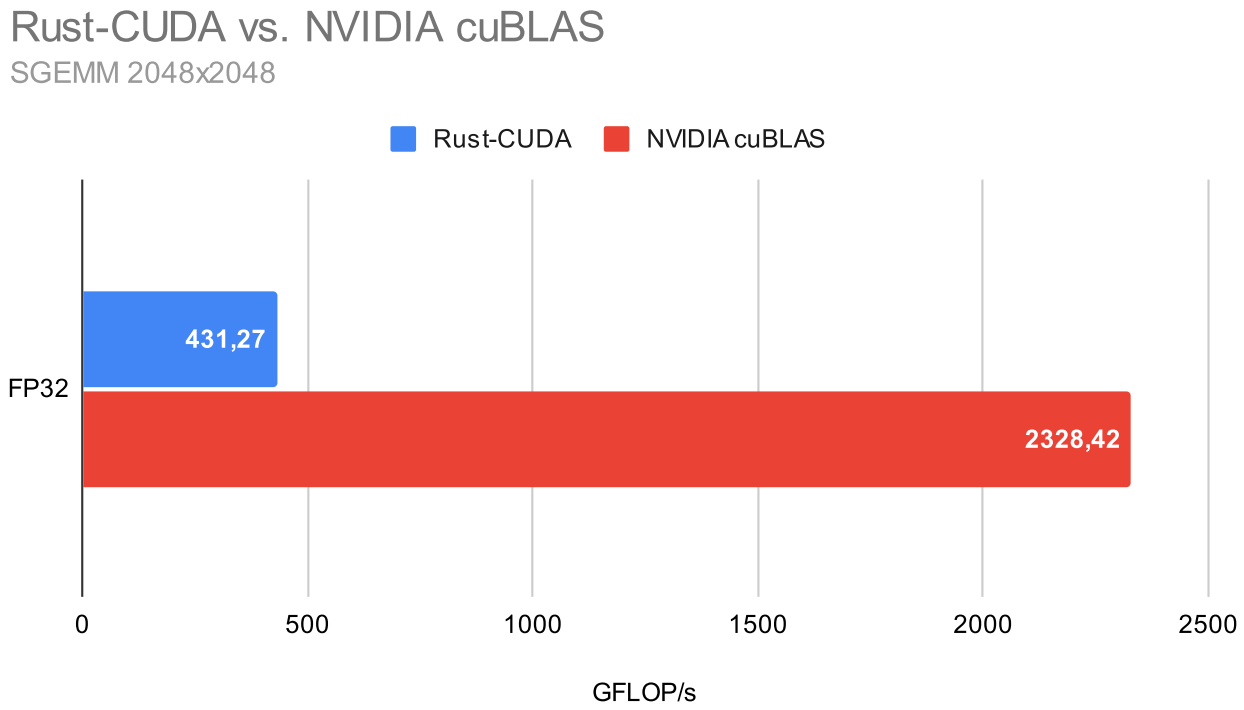
- Comparaisons aux bibliothèques NVIDIA :
  - Double précision



Évaluation de performances

HARP : Résultats et observations sur GEMM

- Comparaisons aux bibliothèques NVIDIA :
  - Simple précision



3. Conclusion

Conclusion

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

Éléments prometteurs :

- Simplicité d'utilisation en comparaison à C/C++
- Performance comparable à CUDA C++ sur les cas étudiés

Mais des manques importants pour faciliter l'adoption :

- Support natif au langage virtuellement inexistant
- Besoin de bibliothèques avec routines nécessaires aux algorithmes plus complexes
  1. API identique à celle de la bibliotèque standard Rust (approche rayon)
  2. *Bindings* vers des bibiliothèques existantes (NVIDIA CUB, Thrust, etc...)

Conclusion

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

Éléments prometteurs :

- Simplicité d'utilisation en comparaison à C/C++
- Performance comparable à CUDA C++ sur les cas étudiés

Mais des manques importants pour faciliter l'adoption :

- Support natif au langage virtuellement inexistant
- Besoin de bibliothèques avec routines nécessaires aux algorithmes plus complexes
  1. API identique à celle de la bibliotèque standard Rust (approche rayon)
  2. *Bindings* vers des bibiliothèques existantes (NVIDIA CUB, Thrust, etc...)

Conclusion

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

Éléments prometteurs :

- Simplicité d'utilisation en comparaison à C/C++
- Performance comparable à CUDA C++ sur les cas étudiés

Mais des manques importants pour faciliter l'adoption :

- Support natif au langage virtuellement inexistant
- Besoin de bibliothèques avec routines nécessaires aux algorithmes plus complexes
  1. API identique à celle de la bibliotèque standard Rust (approche rayon)
  2. *Bindings* vers des bibiliothèques existantes (NVIDIA CUB, Thrust, etc...)

Conclusion

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

Éléments prometteurs :

- Simplicité d'utilisation en comparaison à C/C++
- Performance comparable à CUDA C++ sur les cas étudiés

Mais des manques importants pour faciliter l'adoption :

- Support natif au langage virtuellement inexistant
- Besoin de bibliothèques avec routines nécessaires aux algorithmes plus complexes
  1. API identique à celle de la bibliotèque standard Rust (approche rayon)
  2. *Bindings* vers des bibiliothèques existantes (NVIDIA CUB, Thrust, etc...)

Conclusion

Peut-on se servir de Rust et de ses propriétés pour écrire des codes GPUs performants et exacts ?

Éléments prometteurs :

- Simplicité d'utilisation en comparaison à C/C++
- Performance comparable à CUDA C++ sur les cas étudiés

Mais des manques importants pour faciliter l'adoption :

- Support natif au langage virtuellement inexistant
- Besoin de bibliothèques avec routines nécessaires aux algorithmes plus complexes
  1. API identique à celle de la bibliotèque standard Rust (approche rayon)
  2. *Bindings* vers des bibiliothèques existantes (NVIDIA CUB, Thrust, etc...)

Travaux futurs

- Reprise active du projet Rust-CUDA
  - Mise à jour avec les dernières versions de rustc
  - Extensions du projet (abstractions, bibliothèques, etc...)
  - Intégration *upstream* au compilateur Rust
- Étude sur GPUs AMD
  - Comparaison de performance : OpenCL Rust vs. HIP C++
  - Investiguer la viabilité d'un projet Rust-HIP

Travaux futurs

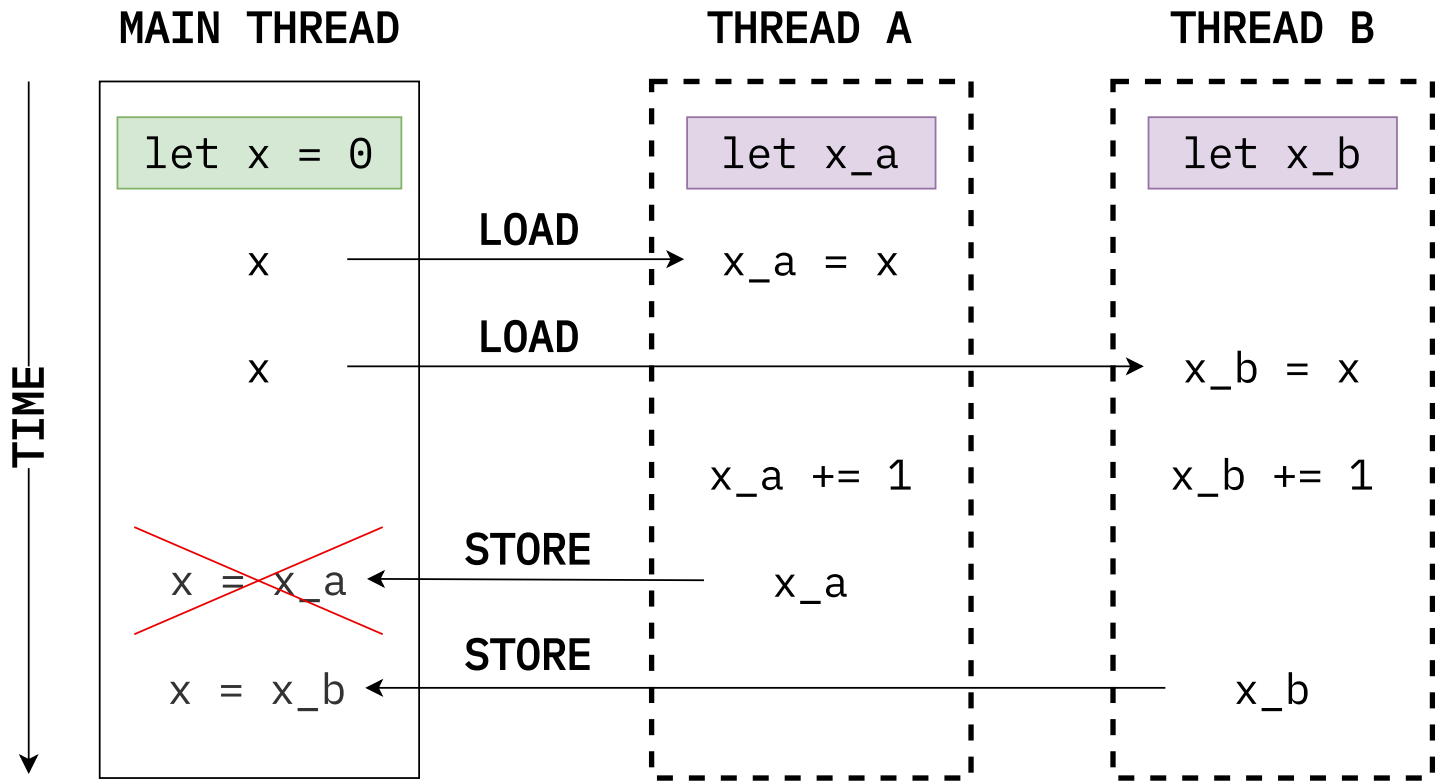
- Reprise active du projet Rust-CUDA
  - Mise à jour avec les dernières versions de rustc
  - Extensions du projet (abstractions, bibliothèques, etc...)
  - Intégration *upstream* au compilateur Rust
- Étude sur GPUs AMD
  - Comparaison de performance : OpenCL Rust vs. HIP C++
  - Investiguer la viabilité d'un projet Rust-HIP

Merci de votre attention

4. Annexe

Race condition

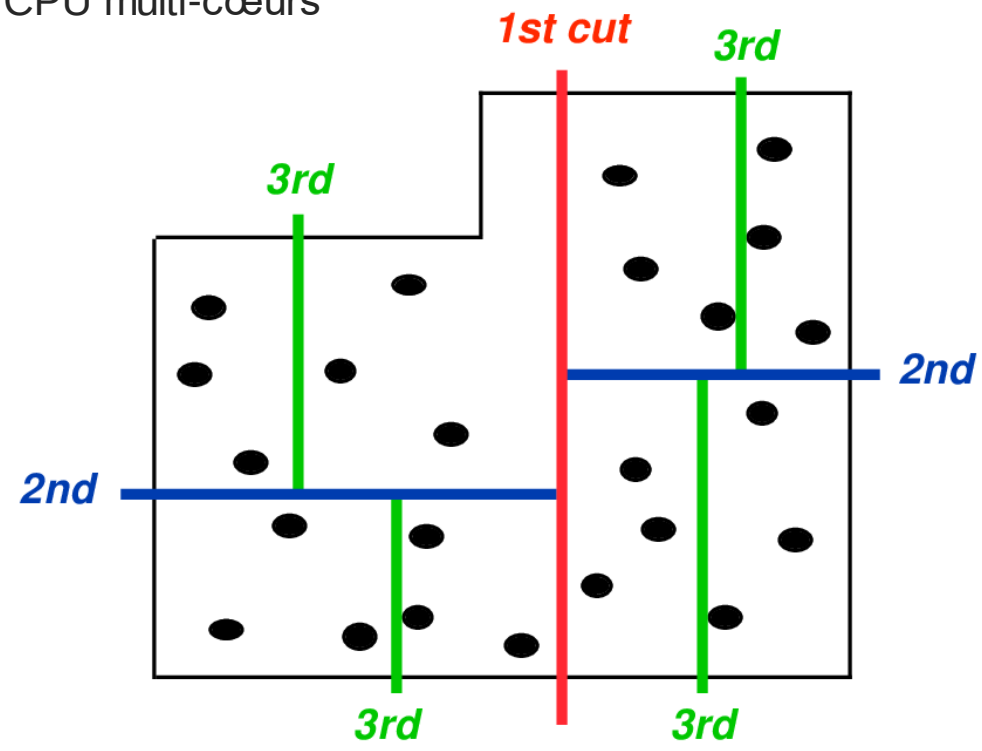
Comment Rust peut-il détecter une *race condition ?*



Portage d'algorithmes sur application CEA

Coupe, partitionneur de maillage en Rust (application CEA/DAM, LIHPC)

- Nombreux algorithmes de partitionnement implémentés pour CPU multi-cœurs
- Portage de l'algorithme RCB sur GPU :
  - *Recursive Coordinate Bisection*
  - Idéal pour architecture GPU
- Portage basé sur Rust-CUDA pour GPU NVIDIA A100



Portage d'algorithmes sur application CEA

Portage de RCB : Observations

Difficultés à porter le code :

- Routines de base indisponibles parmis les bibliothèques compatibles Rust-CUDA :
  - Implémentation manuelle
- Incompatibilité entre kernels CUDA C++ et Rust-CUDA :
  - Erreurs de segmentation à l'exécution
  - Difficile à investiguer (incompatibilité entre ABI ?)
- Code actuel encore non fonctionnel
  - Mais des pistes pour implémentation avec une approche différente