Préparer un code scientifique pour les architectures exascales

Etienne Malaboeuf (Cines)

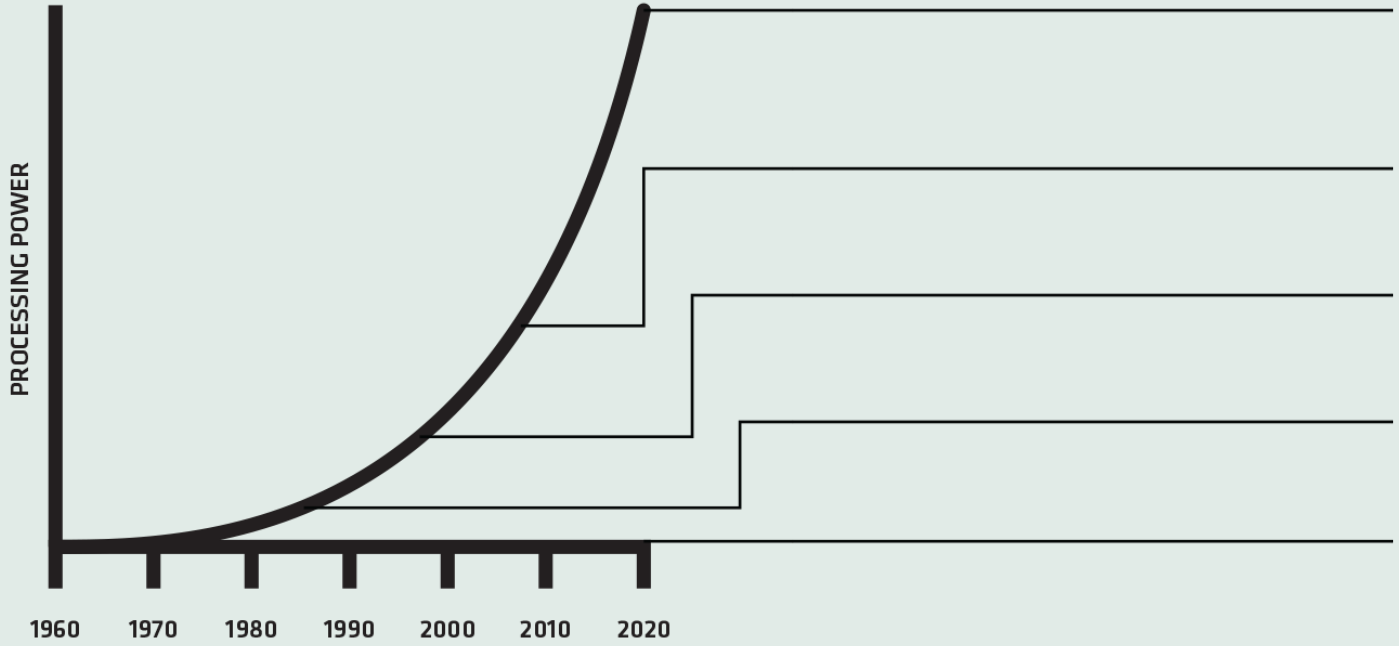
Plan de la présentation

- Pourquoi utiliser un GPU ?
- Quels outils pour utiliser le GPU ?
- Processus de portage

Pourquoi utiliser un GPU ?

Pourquoi utiliser un GPU ?

Nouvelle percée → Nouvelle technologie



GPU intégré ?

Assistance IA ?

Accélérateur GPU

Accélérateur Cell CPU

Mémoire distribuée (MPI)

Foreground processor & background coprocessor

Pipeline/Superscalaire

Vectoriel

Source de l'image: AMD

Le CPU est orienté latence (mémoire, flop)

- **Architectures ambiguës**
  - Faible latence mais de plus en plus de cœurs
- **Fréquence élevée**
- ***Caching* implicite**

Le GPU est orienté débit (mémoire, flop)

- **Efficience énergétique élevée** (Flop/W)
- **Bonne exploitation du parallélisme** (cœurs indépendants)
- **Débits mémoire agrégés élevés** (HBM2 3,2 Tio/s)

Ce qui « *marche mal* » sur GPU

- Beaucoup de transferts de données (*Host* <-> *Device*)
- Données en Array of Structure (AOS) plutôt que Structure of Array (SOA)
- Pas possible de *cache* les données ou mauvaise utilisation du *cache*
- Mauvaise utilisation de l'Instruction Level Parallelism (ILP)
- Gros noyaux
- La reproductibilité bit-à-bit avec les résultats CPU
- Trop de branches
- Les modules très couplés, complexes (ou avec effets de bord)

Accélération minimale viable (énergétiquement)

Pour ADASTRA = x4 en comparant 1 nœud CPU (AMD Genoa) vs 1 nœud GPU (AMD MI250x)

Bien sûr, plus est mieux !

Quels outils pour utiliser le GPU ?

Utilisez les librairies spécialisées

Exemple AMD: rocBLAS, rocFFT, rocSOLVER, rocALUTION, MIOpen, etc.

Programmez le GPU avec un langage spécialisé

Si possible, éviter ce choix

Cherchez à faciliter le portage futur

Différents niveaux d'accès à la ressource

Simple

- Moins de contrôle sur le matériel
- 0 à ~70 % des fonctionnalités*

Intermédiaire

- 0 à ~80 % des fonctionnalités*

Avancé

- Verbeux, pointilleux
- 0 à 100 % des fonctionnalités*

**: anecdotique*

Pour ~90% du code de calcul*

- Utilisez sur une technologie **simple** ou **intermédiaire**

Attention

- Portabilité ***hardware*** != portabilité de la ***performance***
  - Il faut parfois changer de méthode ou reformuler le problème
- **Pérennité**: Aux solutions exotiques
  - Pensez à long terme (+5-8 ans)
- A ne pas avoir trop de dépendances

**: anecdotique*

Quels outils pour utiliser le GPU ?

- **Simple**
  - OpenMP | AMD, Nvidia, Intel, etc. | (C/C++/Fortran)
  - OpenACC | AMD (Fortran), Nvidia (C/C++/Fortran)
- **Intermédiaire**
  - Kokkos, Raja | AMD, Nvidia, Intel, etc. | (C++)
  - Thrust | AMD, Nvidia | (C++)
- **Avancé**
  - HIP | AMD, Nvidia, Intel ? | (C++)
  - Alpaka | AMD, Nvidia, Intel, etc. | (C++)
  - OpenCL, Sycl | AMD, Nvidia, Intel, etc. | (C, C++)
  - Cuda | Nvidia | (C++)

A favoriser

A éviter

Quels outils pour utiliser le GPU ?

- **Compilation**
  - ROCM (AMD)
    - amdclang
    - hipcc
  - Cray Compiling Environnement (CCE)
    - clang

  - Compilateurs basés sur clang/flang donc llvm
  - Tous avec support pour GPU AMD (OpenMP/HIP)

- **Profilage**
  - rocProf (AMD)
  - Cray-PAT/Apprentice2
  - Hpctoolkit
  - …

- **Débogage**
  - Comprendre le hardware
  - printf / commentaires
  - gstack
  - rocGDB
  - …

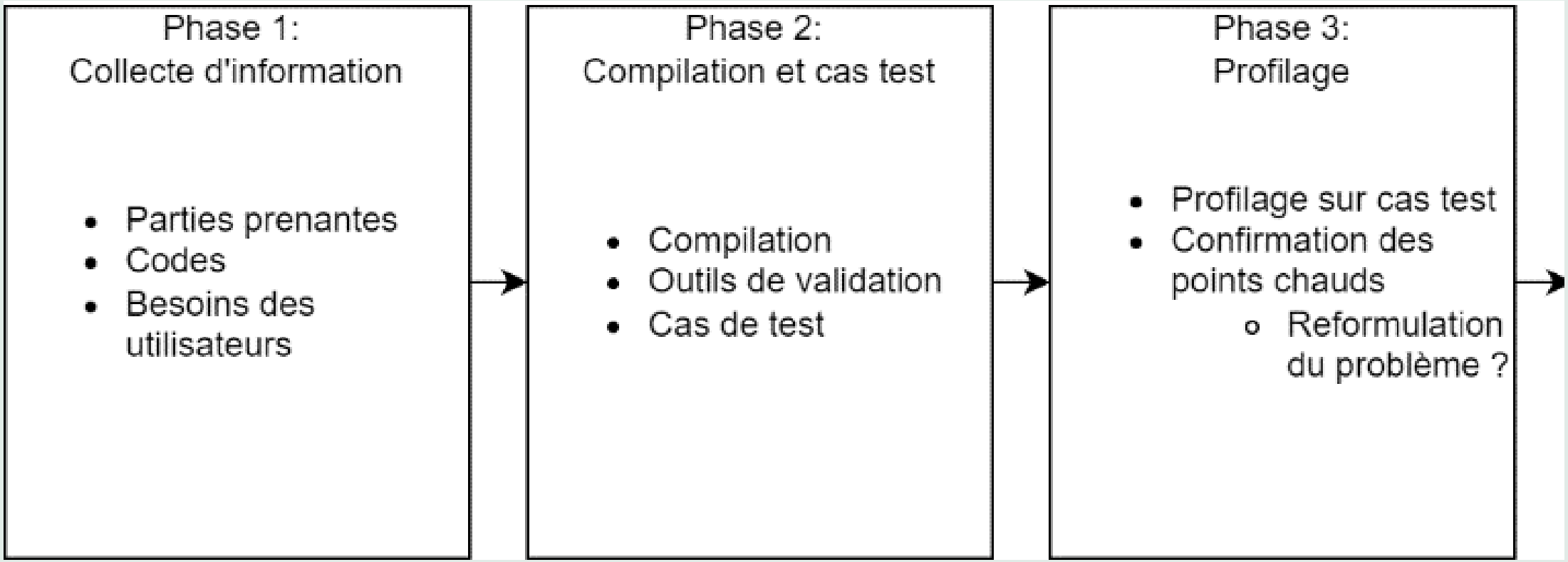
- **MPI**
  - **Cray-MPI** (basé sur MPICH):
    - GPU aware: transfert des buffers GPU
    - GPU direct: lire/écrire depuis/dans la mémoire d'un GPU
- **Gestionnaire de ressource**
  - **SLURM**
    - Réservez des GPU: --gpus-per-task=N
    - S'assurer du placement des taches: --gpu-bind=closest

Notez: les processus d'un nœud pour un job donné doivent pouvoir accéder à tous les GPU pour que MPI GPU fonctionne.

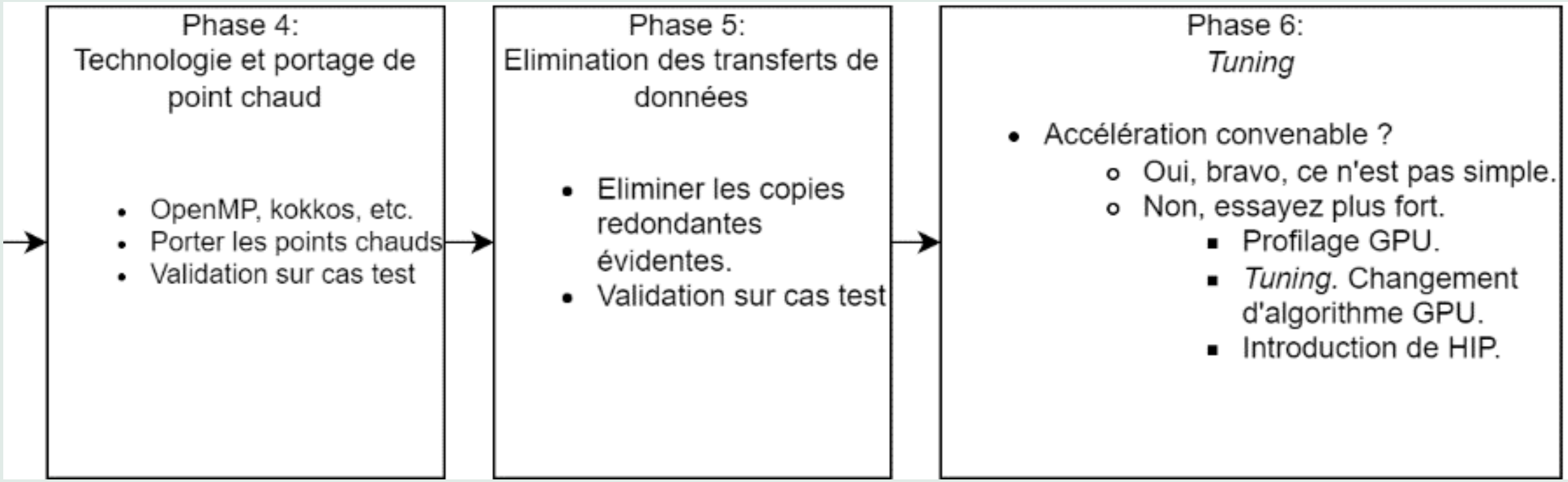
```
export ROCR_VISIBLE_DEVICES=$SLURM_LOCALID
exec "$@"
```

Processus de portage

Préparation



Portage


Phase 4:
Technologie et portage de point chaud
OpenMP, kokkos, etc.
Porter les points chauds
Validation sur cas test
Phase 5:
Elimination des transferts de données
Eliminer les copies redondantes évidentes.
Validation sur cas test
Phase 6:
Tuning
Accélération convenable ?
Oui, bravo, ce n'est pas simple.
Non, essayez plus fort.
Profilage GPU.
Tuning. Changement d'algorithme GPU.
Introduction de HIP.

- **Génie logiciel**
  - **Séparez** bien les données et leurs traitements
  - Adoptez une vision *kernel* des boucles
  - Pensez programmation générique [1]
- **Pipeline de Continuous Integration (CI)**
  - Importance d'un **script de validation**
- **Cas tests rapide (20s), court (1h), long (4h+)**
- ***Git flow***
- ***Sanitizer* (CPU)**
  - *Leak* (détecte les fuites de mémoire)
  - *Undefined behavior*
  - *Address* (détecte les problèmes d'adressabilité)

- **Notion d'*offloading* avec OpenMP**

CPU

```
const int N = 256;
#pragma omp parallel for
for(int i = 0; i < N; ++i) {
    // calcul
}
```

Instanciation des fils d'une *team*

Distribution sur les fils d'une *team*

GPGPU

```
const int N = 256;
#pragma omp target
#pragma omp teams
#pragma omp distribute parallel for
for(int i = 0; i < N; ++i) {
    // calcul
}
```

Sur GPU

Instanciation des *teams*

Distribution sur les *teams*

Instanciation des fils des *teams*

Distribution sur les fils des *teams*

- **Notion sur les copies mémoire avec OpenMP**

```
void DoWork(int* input, size_t size) {
    for(size_t j = 0; j < 10; ++j) {
        #pragma omp target map(tofrom: input[0:size])
        #pragma omp teams distribute parallel for
        for(size_t i = 0; i < size; ++i) {
            input[i] += 1;
        }
    }
}
void CallDoWork(int* input, size_t size) {
    #pragma omp target data map(tofrom: input[0:size])
    DoWork(input, size);
}
```

Conclusion

- **Les architectures changent**
- **Les données ou leurs représentations doivent changer**
- **Les codes doivent donc s'adapter**
  - La plateforme sur laquelle le code fonctionne est bien le hardware, et pas seulement une pile logiciel.

Questions ?

[1]: Elements of programming, Stepanov, Alexander A and McJones, Paul, Addison-Wesley Professional

Annexes

- **Nœud de calcul**
  - 1 CPU Trento par nœud
  - 4 cartes MI250x par nœud
  - 382 Tflop/s par nœud (SGEMM/DGEMM)
  - 191 Tflop/s par nœud (*opération scalaire*)
  - Infinity fabric (PCIe propriétaire pour faciliter le MCM, pensez au NVLink)

- **E/S**
  - Réseau d'interconnexion Slingshot 11 : Dragonfly
  - 4 NIC par nœud accéléré à 200 Gib/s
  - SCRATCH: 1,3 Tio lecture | 0,74 Tio écriture (agrégé)

- **Pile logiciel**
  - RedHat 8.6
  - ROCm 5
  - Cray Compiling Environnement (CPE)
  - Cray-MPI (MPICH)

} Compilateurs basés sur clang/flang donc llvm

• GPU AMD MI250x

- 2 Graphics Compute Die (GCD) visibles comme 2 GPU
  - 8 GCD par nœuds
- 110 Compute Unit (CU) par GCD
- 32 wavefront par CU (max)
- Local Data Store (LDS) == Shared memory
  - 64 Kio/CU
- 64 Gio de VRAM / GCD
- 64 K VGPR / CU | 1 VGPR == 4 o
- Débit VRAM 1,6 Tio/s / GCD
- Mémoire unifiée
  - Similaire à Nvidia + Power9
  - Utilisation par un simple malloc

• CPU AMD Trento

- 64 cœurs
- 256 Gio RAM

OpenMP	Matériel AMD	AMD/HIP	Cuda
Thread	SIMD *lane/work-item/thread*	*Thread/work-item*	*Thread*
SIMD	16 SIMD *lanes* pour des opérations 64bits	*Wave*	*Wave*
Sans équivalent	*Wavefront* (64 fils)	*Wavefront* (64 fils)	*Wrap* (32 fils)
Team	*Workgroup*	*Workgroup*	*Thread block*
Sans équivalent	CU	CU	Streaming Multi-processor
num_teams	Quantité de *work-groups*	Quantité de *work-groups*	Quantité de *thread blocks*
thread_limit	Taille du *work-group*	Taille du *work-group*	Taille du *thread block*

Processus complet de portage