

# Architecture et programmation d'accélérateurs matériels

Cours 3 : APIs CUDA et Programmation  
Multi-GPUs

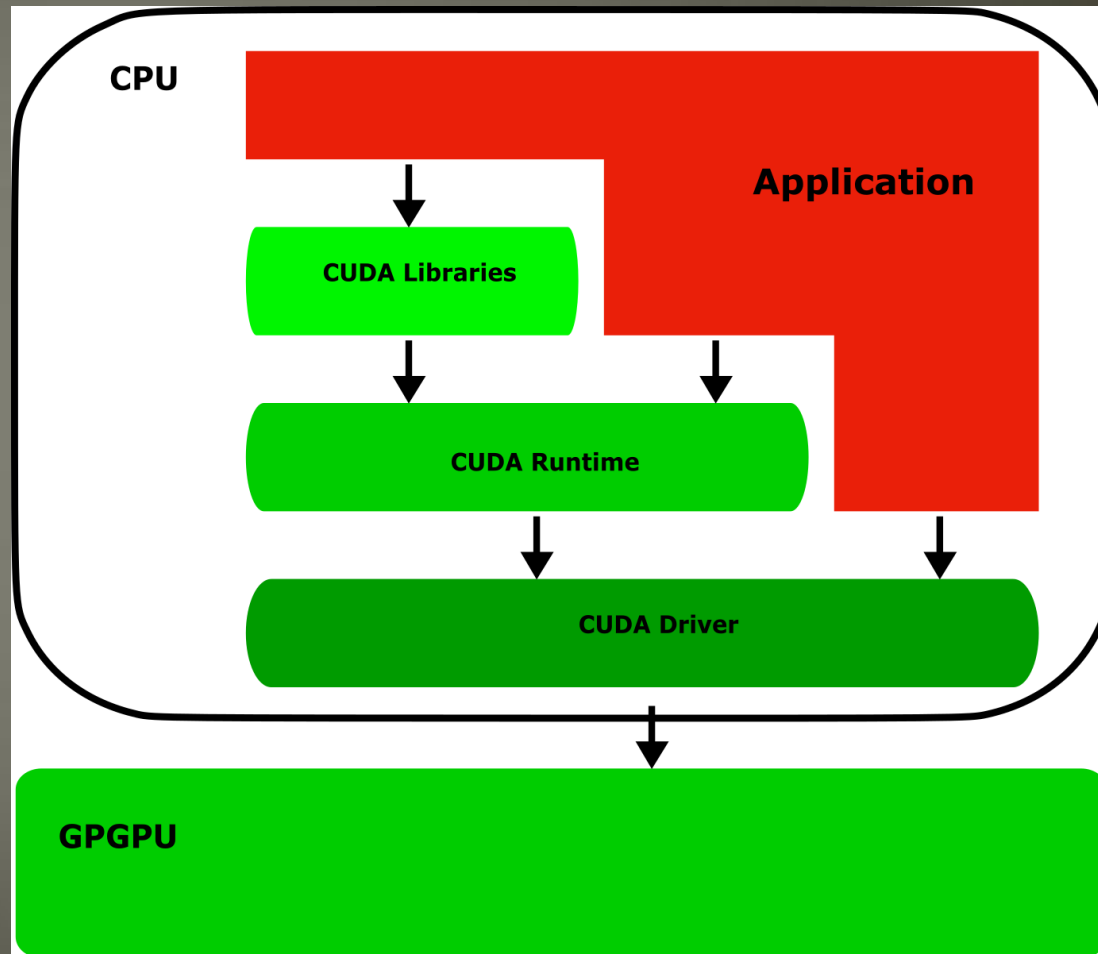
Adrien Roussel  
[adrien.roussel@cea.fr](mailto:adrien.roussel@cea.fr)

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- Programmation Multi-GPUs

# Les différentes APIs pour CUDA

# Les différents niveaux de programmation CUDA



# Plan du cours

- Les différentes APIs pour CUDA
  - API Runtime
  - API Diver
- Notion de contextes CUDA
- Programmation Multi-GPUs

# API Runtime (1)

- C'est l'API des utilisateurs finaux
  - Les exemples du cours et les fonctions utilisées en TDs jusqu'à présent font parties de l'API Runtime
- Fonctions préfixées par « cuda »
- Elle fournit les fonctions de base pour la programmation CUDA
  - `cudaMalloc`, `cudaFree`, `cudaMemcpy`, ...

## API Runtime (2)

- API de haut niveau
- Niveau d'abstraction élevé
- Permet de ne pas exposer tous les mécanismes internes de CUDA aux utilisateurs

# Plan du cours

- Les différentes APIs pour CUDA
  - API Runtime
  - API Driver
- Notion de contextes CUDA
- Programmation Multi-GPUs



# API Driver (1)

- API de bas niveau
- C'est l'API des développeurs de bibliothèques
  - Besoin d'une expertise plus importante pour l'utiliser
  - Permet d'isoler les développements CUDA dans une bibliothèque du reste du programme.

# API Driver (2)

- Fonctions préfixées par « cu »
- Elle fournit des fonctions équivalentes à celle de l'API Runtime
  - `cudaMalloc` → `cuMemAlloc`
  - `cudaMemcpy` → `cuMemcpyDtoH`  
`cuMemcpyHtoD`  
`cuMemcpyHtoH`  
`cuMemcpyDtoD`

# Contextes CUDA

# Contextes (1)

- Structure interne de CUDA attaché à un GPU
  - Lorsqu'une opération demandant le concours d'un GPU est réalisée, le runtime CUDA « regarde » dans cette structure pour savoir quel GPU doit être utilisé
- Créé lorsqu'on demande à s'attacher à un nouveau GPU
  - Si un contexte n'existe pas déjà pour ce GPU

## Contextes (2)

- Encapsule tous les objets relatifs au bon fonctionnement de CUDA comme:
  - Toutes les allocations mémoires avec son propre espace d'adressage.
    - Ainsi, seuls les threads partageant le même contexte partagent le même espace d'adressage
  - Les streams CUDA.
  - Les événements CUDA.
  - ...

# Contextes (3)

- Deux types de contextes
  - Selon le niveau CUDA supporté
  - Primaire et « classique »
- Chacun accessible depuis une API différente
  - Primaire: API Runtime
  - « classique »: API Driver

# Programmation Multi-GPUs

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- **Programmation Multi-GPUs**
  - Sélection d'un GPU dans CUDA
  - UVA
  - MPI+CUDA
  - OpenMP + CUDA
  - NCCL



# Sélection d'un GPU en CUDA

- Il existe deux fonctions pour choisir le GPU sur lequel les prochaines opérations hétérogènes vont être exécutées
  - API Runtime: `cudaSetDevice(...)`
  - API Driver: `cuCreateCtx(...)`

# API Runtime: cudaSetDevice

- `__host__ cudaError_t cudaSetDevice (int device)`
  - Set device to be used for GPU executions.
- Parameters
  - device
    - Device on which the active host thread should execute the device code.

# Utilisation de cudaSetDevice

```
Int main()  
{  
...  
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut  
    (device 0)  
    cudaSetDevice(1); // Sélectionne le device 1 pour les  
    prochaines opérations CUDA  
    cudaMalloc(&d_b, ...) // malloc sur le device 1  
...  
}
```

# cudaSetDevice: détails (1)

- `cudaSetDevice` vérifie si un contexte est déjà associé au device demandé
  - Si il existe une pile de contexte « classique »
    - Sélectionne le contexte correspondant
    - Crée un nouveau contexte « classique » associé à ce device
  - Sinon, sélectionne le contexte primaire correspondant, ou en crée un nouveau

# cudaSetDevice: détails (1)

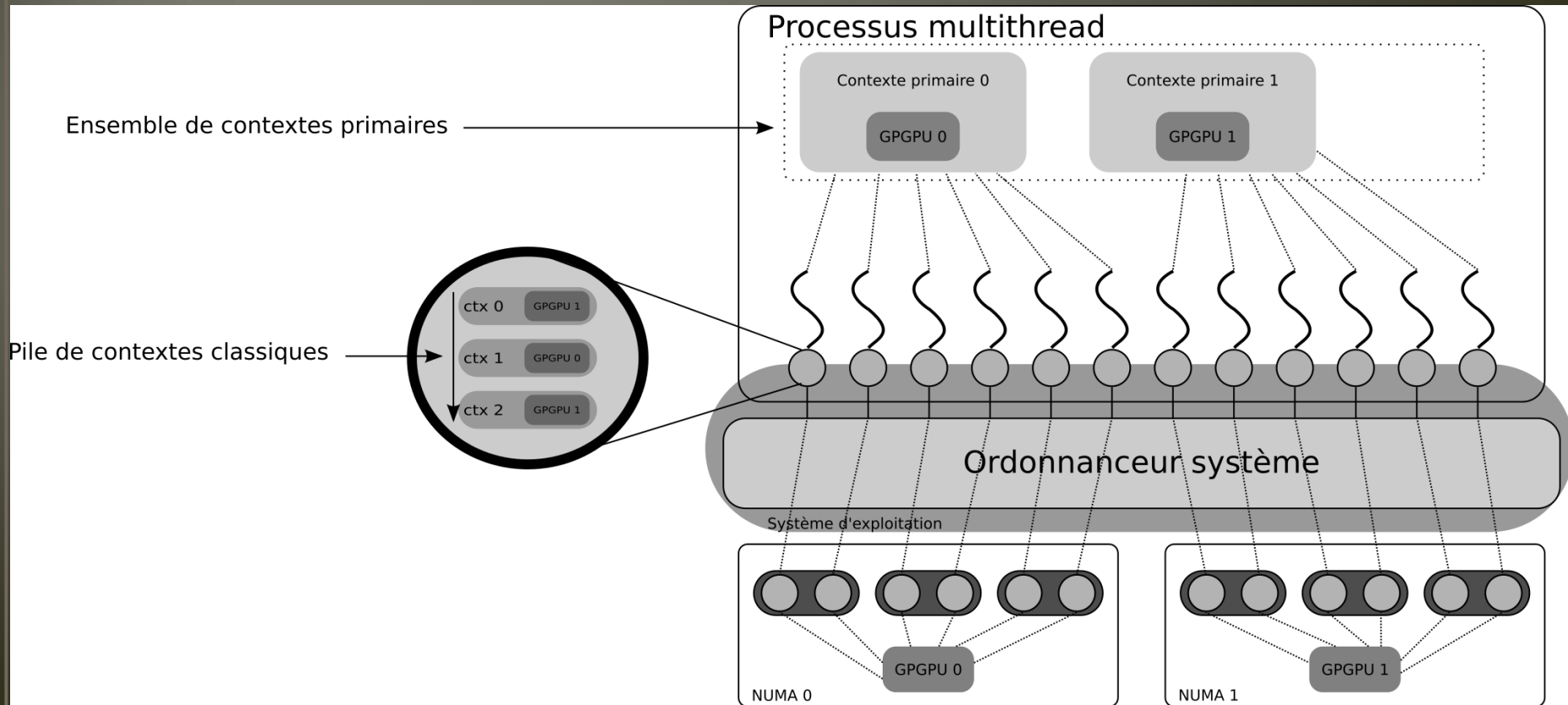
- `cudaSetDevice` vérifie si un contexte est déjà associé au device demandé
  - Si il existe une pile de contexte « classique »
    - Sélectionne le contexte correspondant
    - Crée un nouveau contexte « classique » associé à ce device
  - Sinon, sélectionne le contexte primaire correspondant, ou en crée un nouveau
  - Si oui, il s'associe au context primaire correspondant
  - Sinon, il crée un contexte primaire associé au GPU demandé puis s'y associe

## cudaSetDevice: détails (2)

- Précédence sur les contextes primaires
  - Ctx « classique » choisit avant ctx primaire

## Contexte « classique » (4)

- **Précédence sur les contextes primaires**
  - Ctx « classique » choisit avant ctx primaire



## cudaSetDevice: détails (3)

- Si le device demandé n'existe pas (device 3 alors qu'il n'y a que 2 devices sur le nœud)
  - Pas d'erreur ni de segfault
  - Les prochains codes CUDA seront exécutés sur le device par défaut (device 0)



# API Runtime: cudaGetDeviceCount

- `__host____device__cudaError_t cudaGetDeviceCount`  
`(int *count)`
  - Returns the number of compute-capable devices.
- Parameters
  - Count
    - Returns the number of devices with compute capability greater or equal to 2.0

# Utilisation de cudaSetDevice

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...);
    cudaGetDeviceCount(&nbGPUs); // Récupère le nombre
de GPUs disponibles
    cudaSetDevice(1%nbGPUs); // Modulo sur le nombre de
GPUs permet de choisir un « vrai » GPU
    cudaMalloc(&d_b, ...); //
    ...
}
```

# API Driver: cuCtxCreate

- *CUresult* **cuCtxCreate** (*CUcontext* \*pctx, unsigned int flags, *CUdevice* dev)
  - Create a CUDA context.
    - Ce nouveau contexte est « pushé » en tête de pile du thread appelant
- Parameters
  - Pctx: Returned context handle of the new context
  - Flags: Context creation flags
  - Dev: Device to create context on

# API Driver: cuCtxPopCurrent

- *CUresult* **cuCtxPopCurrent** (*CUcontext* \*pctx)
  - Pops the current CUDA context from the current CPU thread.
- Parameters
  - Pctx
    - Returned new context handle

# API Driver: cuCtxPushCurrent

- *CUresult* **cuCtxPushCurrent** (*CUcontext ctx*)
  - Pushes a context on the current CPU thread.
- Parameters
  - Ctx
    - Context to push

# Utilisation de cuCtxCreate

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
    (device 0)
    cuCtxCreate(&myctx, flags, 1); // Sélectionne le device 1
    pour les prochaines opérations CUDA
    cudaMalloc(&d_b, size) // malloc sur le device 1
    cuMemAlloc(&d_c, size) // malloc sur le device 1
    ...
}
```

# Utilisation de `cuCtxPopCurrent` et `cuCtxPushCurrent`

```
Int main()
{
    cuCtxCreate(&myctx, flags, 1);
    cudaMalloc(&d_a, ...); // malloc sur le device 1
    cuCtxPopCurrent(&tmpctx); // Retire myctx du dessus de la pile des
contextes classiques
    cudaMalloc(&d_b, ...); // malloc sur le device précédent
    cuCtxPushCurrent(tmpctx); // On remet le contexte sur le dessus de
la pile
    cudaMalloc(&d_c, ...); // malloc sur le device 1
}
```

## cuCtxCreate: détails (1)

- `cuCtxCreate` va créer un nouveau contexte pour le GPU demandé quel que soit le contenu de la pile
  - Il est possible d'avoir dans la même pile de contextes « classiques » plusieurs contexte attaché au même GPU
  - Lors d'un appel à `cudaSetDevice`, il va sélectionner le premier contexte attaché au GPU demandé trouvé lors du parcours de la pile



## cuCtxCreate: détails (1)

- `cuCtxCreate` va créer un nouveau contexte pour le GPU demandé quel que soit le contenu de la pile
  - Il est possible d'avoir dans la même pile de contextes « classiques » plusieurs contexte attaché au même GPU
  - Lors d'un appel à `cudaSetDevice`, il va sélectionner le premier contexte attaché au GPU demandé trouvé lors du parcours de la pile

# API Driver: Autres fonctions

- `CUresult cuCtxGetCurrent (CUcontext *pctx)`
  - Returns the CUDA context bound to the calling CPU thread.
    - Ce contexte n'est pas « poppé »

# API Driver: Autres fonctions

- `CUresult cuCtxGetCurrent (CUcontext *pctx)`
  - Returns the CUDA context bound to the calling CPU thread.
- `CUresult cuCtxGetDevice (CUdevice *device)`
  - Returns the device ID for the current context.

# API Driver: Autres fonctions

- `CUresult cuCtxGetCurrent (CUcontext *pctx)`
  - Returns the CUDA context bound to the calling CPU thread.
- `CUresult cuCtxGetDevice (CUdevice *device)`
  - Returns the device ID for the current context.
- `CUresult cuCtxSetCurrent (CUcontext ctx)`
  - Binds the specified CUDA context to the calling CPU thread.
    - En fait, effectue un remplacement (pop la tête de la pile, push ctx en tête de pile)

# Gestion fine des contextes « classiques » (1)

- Pour éviter de créer plusieurs contextes « classique » attachés au même GPU:
  - Parcours « à la main » de la pile de contextes « classiques »
    - Avant de créer un nouveau contexte, parcours de toute la pile en « poppant » les contextes.
    - A chaque étape, un appel à `cuCtxGetDevice()` vous donnera l'ID du device auquel le contexte courant est attaché

## Gestion fine des contextes « classiques » (2)

- Si c'est le GPU voulu, « poppez » le, « poussez » tous les contextes avec le GPU voulu en dernier
- Si vous ne trouvez pas le GPU voulu, alors faites votre appel à `cuCtxCreate`
- Ou laisser faire l'API Runtime
  - Créer un premier contexte « classique » pour votre thread
  - Ensuite chaque appel à `cudaSetDevice` cherchera dans la pile de contexte « classique » un contexte correspondant, ou bien créera un nouveau context « classique »

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- **Programmation Multi-GPUs**
  - Sélection d'un GPU dans CUDA
  - UVA
  - MPI+CUDA
  - OpenMP + CUDA
  - NCCL

# Utilisation de la même adresse sur des devices différents

```
Int main()
{
...
    cudaMalloc(&d_a, ...) //
    •
    cudaSetDevice(1); //
    •
    cudaMemcpy(&d_a, ...) //
    •
...
}
```



# Utilisation de la même adresse sur des devices différents

```
Int main()
{
...
    cudaMalloc(&d_a, ...) //
•
    cudaSetDevice(1); //
•
    cudaMemcpy(&d_a, ...) //
•
...
}
```

Que se pass-t-il dans ce cas ?

# Utilisation de la même adresse sur des devices différents

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
                           (device 0)
    cudaSetDevice(1); // Sélectionne le device 1 pour les
                     prochaines opérations CUDA
    cudaMemcpy(&d_a, ...) // Erreur! d_a est alloué sur le
                           device 0, et ne veut rien dire pour le device 1
    ...
}
```

Que se pass-t-il dans ce cas ?

# Utilisation de la même adresse sur des devices différents

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
                           (device 0)
    cudaSetDevice(1); // Sélectionne le device 1 pour les
                      prochaines opérations CUDA
    cudaMemcpy(&d_a, ...) // Erreur! d_a est alloué sur le
                           device 0, et ne veut rien dire pour le device 1
    ...
}
```

C'est ce qui arrivait avant UVA

# UVA: Unified Virtual Addressing

- UVA est arrivée avec CUDA 4.0
- UVA est un espace mémoire unifié virtuel
  - Les mémoires de l'hôte et des GPUs sont indépendants et ont des plages d'adresse mémoire différentes
  - UVA présente ces mémoires dans le meme espace d'adressage
- Le host ou un device peut interpréter correctement une adresse mémoire venant d'un autre device
  - Il peut savoir de quel device est issue cette adresse mémoire

# Ce qui se passe avec UVA

```
Int main()
{
...
    cudaMalloc(&d_a, ...) //
•
    cudaSetDevice(1); //
•
    cudaMemcpy(&d_a, ...) //
•
•
...
}
```

# Ce qui se passe avec UVA

```
Int main()
{
...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
(device 0)
    cudaSetDevice(1); //
•
    cudaMemcpy(&d_a, ...) //
•
•
...
}
```

# Ce qui se passe avec UVA

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
    (device 0)
    cudaSetDevice(1); // Sélectionne le device 1 pour les
    prochaines opérations CUDA
    cudaMemcpy(&d_a, ...) //
    .
    .
    ...
}
```

# Ce qui se passe avec UVA

```
Int main()
{
    ...
    cudaMalloc(&d_a, ...) // malloc sur le device par défaut
                           (device 0)
    cudaSetDevice(1); // Sélectionne le device 1 pour les
                      prochaines opérations CUDA
    cudaMemcpy(&d_a, ...) // Le runtime reconnaît que d_a
                           est une adresse sur le GPU 0, et va donc copier les données sur ce
                           GPU
    ...
}
```



# UVA: Unified Virtual Addressing

- Il est tout a fait possible de copier des données vers, ou depuis, un autre GPU que celui auquel on est associé
  - Le runtime peut reconnaître les localisations des adresses
  - Plus besoin de spécifier le sens de la copie

# UVA: Unified Virtual Addressing

- Il est tout a fait possible de copier des données vers, ou depuis, un autre GPU que celui auquel on est associé
  - Le runtime peut reconnaître les localisations des adresses
  - Plus besoin de spécifier le sens de la copie
    - `CudaMemcpyHostToDevice`
    - `CudaMemcpyDeviceToHost...`

# UVA: Unified Virtual Addressing

- Il est tout a fait possible de copier des données vers, ou depuis, un autre GPU que celui auquel on est associé
  - Le runtime peut reconnaître les localisations des adresses
  - Plus besoin de spécifier le sens de la copie
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost...`
  - `cudaMemcpyDefault`
    - Le runtime détermine seul le sens de la copie au fonction des adresses passées en argument de `cudaMemcpy`

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- **Programmation Multi-GPUs**
  - Sélection d'un GPU dans CUDA
  - UVA
  - MPI+CUDA
  - OpenMP + CUDA
  - NCCL

# MPI+CUDA (1)

```
Int main()
{
    MPI_Init(...)//
    ...
    cudaMalloc(...); // Tous les processus MPI utilisent le
    même GPU par défaut
    ...
    MPI_Finalize(...)
}
```

# MPI+CUDA (2)

```
Int main()
{
    MPI_Init(...)//
    ...
    cudaSetDevice (1); // Assigne le GPU 1 à chaque
    processus
    cudaMalloc(...); // Tous les processus MPI utilisent le
    même GPU 1
    ...
    MPI_Finalize(...)
}
```

# MPI+CUDA (3)

```
Int main()
{
    MPI_Init(...)//
    MPI_Comm_rank(MCW, &rank);
    cudaSetDevice (rank); // Assigne un GPU unique à chaque
    processus MPI. Pb si rank>nb GPU.
    cudaMalloc(...); //

    ...

    MPI_Finalize(...)
}
```

# MPI+CUDA (4)

```
Int main()
{
    MPI_Init(...)//
    MPI_Comm_rank(MCW, &rank);
    cudaGetDeviceCount(&nbGPU);
    cudaSetDevice (rank % nbGPU); // Assigne un GPU à
chaque processus MPI en Round-Robin.
    cudaMalloc(...);
    ...
    MPI_Finalize(...)
}
```



# Affectation des GPUs en Round-Robin (1)

- Maximise la répartition des processus MPI sur les différents GPUs disponibles
  - Ne maximise par forcément l'occupation des ces GPUs
    - Il faudrait pour cela affecter les GPUs aux processus MPI en fonction de la charge de travail présente... très compliqué à mettre en place
- Les GPUs affectés aux processus MPI ne sont pas forcément les plus proches
  - Problèmes d'effet NUMA

## Affectation des GPUs en Round-Robin (2)

- Les processus utilisant le même GPU ne partagent pas le même espace d'adressage
  - Il faudrait partager le même contexte... très compliqué à mettre en place
    - Process-based: bcast du contexte du GPU choisi aux processus impliqués -> non prévu par CUDA (la taille et le contenu d'un contexte n'est pas accessible aux utilisateurs de CUDA)
    - Thread-based: utilisation de contextes primaires

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- **Programmation Multi-GPUs**
  - Sélection d'un GPU dans CUDA
  - UVA
  - MPI+CUDA
  - OpenMP + CUDA
  - NCCL

# OpenMP+CUDA (1)

```
Int main()
```

```
{
```

```
    #pragma omp parallel
```

```
    {
```

```
        rank = omp_get_thread_num();
```

```
        cudaGetDeviceCount(&nbGPU);
```

```
        cudaSetDevice (rank % nbGPU); // Chaque thread choisit son GPU.
```

Le GPU courant n'est modifié que pour le thread courant. Mémoire partagée pour les threads sélectionnant le même GPU (utilisation de contexte primaire).

```
        cudaMalloc(...);
```

```
    }
```

```
}
```

# OpenMP+CUDA (2)

```
Int main()
{
    #pragma omp parallel
    {
        rank = omp_get_thread_num();
        cudaGetDeviceCount(&nbGPU);
        cuCtxCreate (rank % nbGPU); // Créer et assigne, un nouveau
        contexte Indépendant pour chaque thread associé en Round-Robin à un GPU
        disponibles. Mémoire du GPU non partagée.
        cudaMalloc(...);
    }
}
```

# OpenMP+CUDA (2)

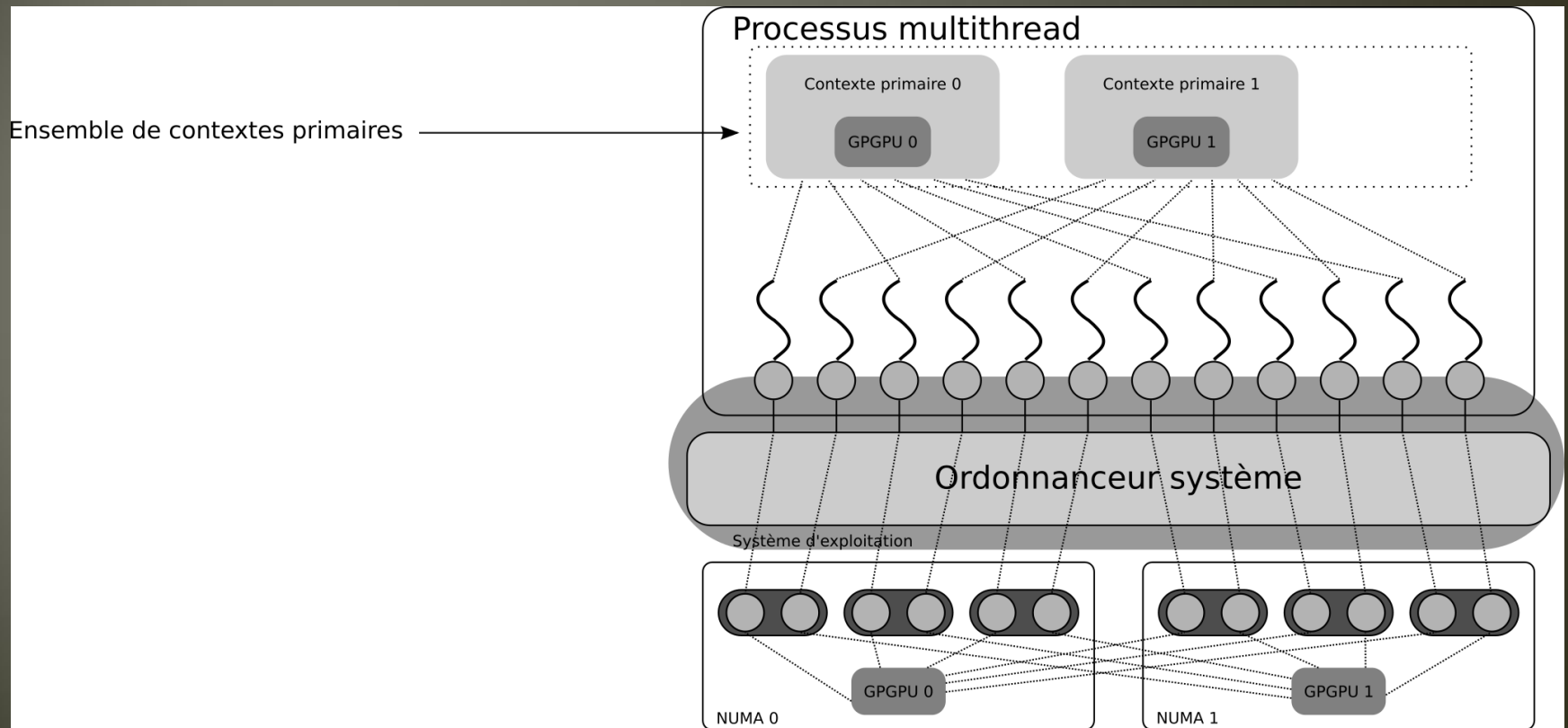
```
Int main()
{
    #pragma omp parallel
    {
        rank = omp_get_thread_num();
        cudaGetDeviceCount(&nbGPU);
        cuCtxCreate (rank % nbGPU);
        cudaMalloc(&d_a,...);

        cudaGetDeviceCount(&nbGPU);
        cuCtxCreate (rank % nbGPU);
        cudaMemcpy(&d_a,...); // Erreur! Même si c'est le même GPU, contexte
différent du malloc, donc espace d'adressage différent!
    }
}
```

# Affectation des GPUs en Round-Robin

- Nous avons les mêmes contraintes en OpenMP+CUDA (et en threads+CUDA en général) qu'en MPI+CUDA
  - Ne maximise pas forcément l'occupation des GPUs
  - Les GPUs affectés aux threads ne sont pas forcément les plus proches
  - Les threads utilisant le même GPU ne partagent pas forcément le même espace d'adressage
    - Cela dépend de l'utilisation de contexte classiques ou primaires

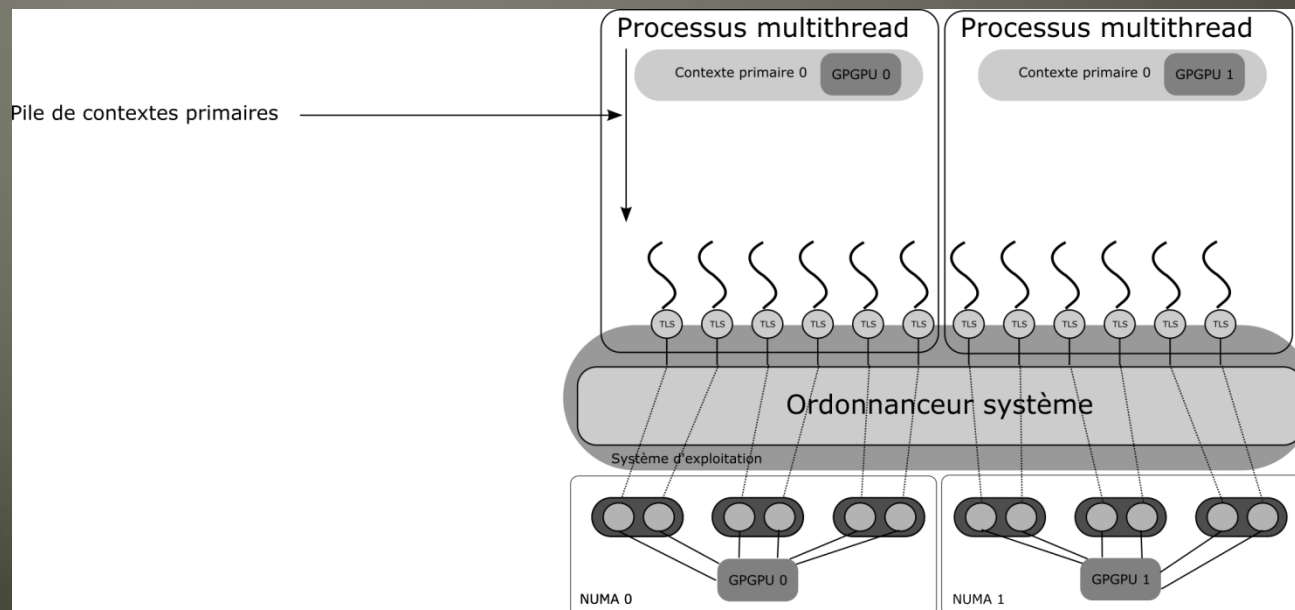
# Affectation des GPUs en Round-Robin





# MPI+OpenMP+CUDA (1)

- Meilleur cas ... presque (pour les GPUs)
  - 1 processus MPI/GPU disponible



# MPI+OpenMP+CUDA (2)

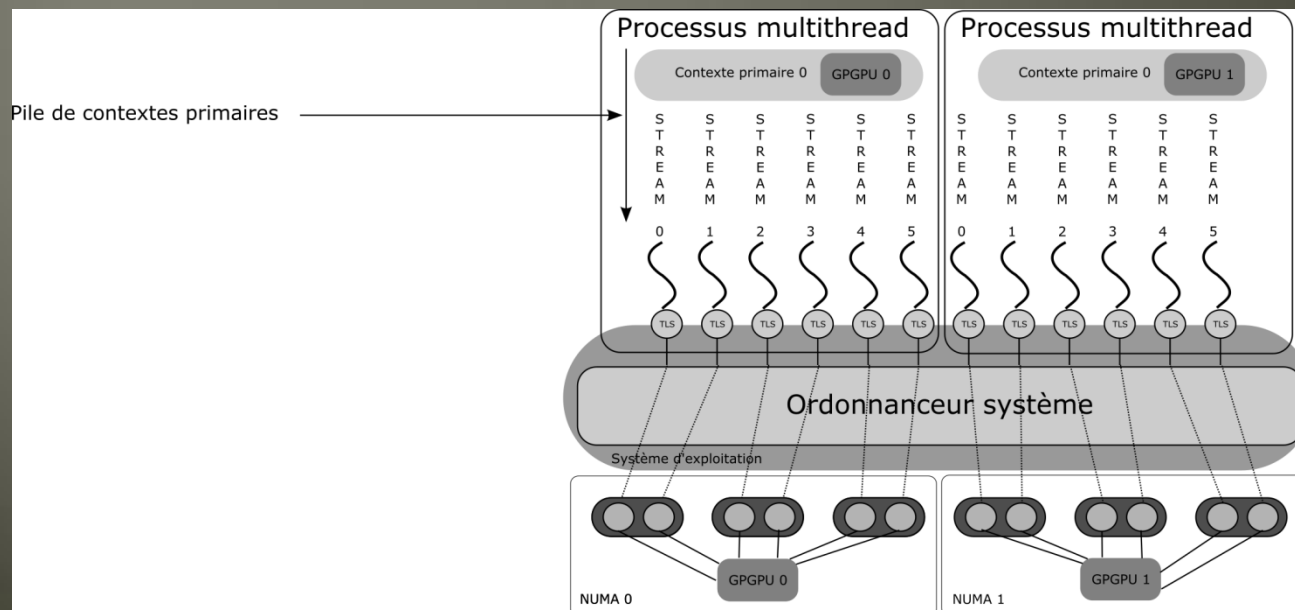
- Utilise tous les GPUs disponibles
- Tous les threads d'un même processus MPI partage le même espace d'adressage sur GPU
  - Possible de faire alloc+memcpy avant une région parallèle, puis chaque thread lance un kernel sur ses données
- Possible avec l'API Runtime
  - Pas besoin des fonctions compliquées de l'API Driver
- Fonctionne avec user-level threads

# MPI+OpenMP+CUDA (3)

- `/!\` Chaque appel CUDA dans un même processus MPI sera sérialisé avec les autres appels des autres threads
  - Utilisation des streams: associer un stream unique à chaque threads
    - Les streams sont indépendants: pas de sérialisation, et il est alors possible pour deux threads utilisant seulement la moitié du GPU de l'utiliser au même moment
    - Attention: nombre de streams limités (16)

# MPI+OpenMP+CUDA (4)

- Meilleur cas (pour les GPUs)
  - 1 processus MPI/GPU disponible + streams



# MPI+OpenMP+CUDA (5)

```
Int main()
{
    MPI_Init(...)//
    MPI_Comm_rank(MCW, &mpirank);
    cudaGetDeviceCount(&nbGPU);
    cudaSetDevice (mpirank % nbGPU); // modulo nécessaire pour associer
chaque rang au bon GPU en multi-noeud
    #pragma omp parallel
    {
        omprank = omp_get_thread_num();
        cudaMalloc(..., omprank % 16); // appel à cudaMalloc sur le
stream n° (omprank % 16).
    }
    MPI_Finalize(...)
}
```

# MPI+OpenMP+CUDA (6)

- Pour de meilleures performances, il faut tenir compte de la position des GPUs pour placer les processus MPI et les threads générés de façon à éviter les effets NUMA
- Parfois, il n'est pas possible d'avoir une hiérarchie sans effets NUMA
  - Exemple: 2 sockets, avec les 2 GPUs attachés à la même socket

# MPI+OpenMP+CUDA (7)

- /!\ La meilleure répartition processus MPI/threads OpenMP pour l'utilisation des GPUs n'est pas forcément celle apportant les meilleures performances sur CPUs
- /!\ Des bibliothèques externes peuvent manipuler les GPUs/les contextes CUDA et « casser » votre répartition optimale

# Plan du cours

- Les différentes APIs pour CUDA
- Notion de contextes CUDA
- **Programmation Multi-GPUs**
  - Sélection d'un GPU dans CUDA
  - UVA
  - MPI+CUDA
  - OpenMP + CUDA
  - NCCL



# NCCL (1)

- NCCL (« Nickel »): Nvidia Collective Communications Library
- But: permettre des échanges point-à-points et collectifs de données entre plusieurs GPUs présents sur un même nœud.
  - Possible de le faire entre GPUs sur nœuds distincts si ceux-ci sont reliés par les cartes réseaux Nvidia Mellanox Infiniband

# NCCL (2)

- Très proche de la façon de faire de MPI
- Une « clique » regroupe les GPUs mis en jeu dans une collective (pas forcément tous)
- Un communicateur est initialisé pour chaque GPU dans la « clique » voulue
  - Soit un appel par GPU
    - Cette initialisation fait intervenir une barrière: il est nécessaire de faire cette initialisation en parallèle
    - Processus MPI différents ou threads différents
  - Soit un appel global

# Fonctions d'initialisation

- `ncclResult_t ncclCommInitRank (ncclComm_t* comm, int nGPUs, ncclUniqueId cliqueld, int rank);`
  - Initialise le rang « rank »
- **Parameters**
  - Comm: communicateur CUDA
  - nGPUs: nombre de GPU dans la clique
  - cliqueld: ID unique pour la clique
    - Un rang fait appel à `ncclGetUniqueId()` puis broadcast (`MPI_Bcast, ...`)
  - Rank: ID unique pour le GPU courant

# Fonctions d'initialisation

- `ncclResult_t ncclCommInitAll (ncclComm_t* comms, int nGPUs, int* devList);`
  - Initialise directement les nGPUs GPUs
- **Parameters**
  - Comms: tableau de comm, un pour chaque GPU
  - nGPUs: nombre de GPU dans le communicateur
  - devList: quel CUDA device est associé à quel rang

# Utilisation de ncclCommInitRank (MPI)

```
Int main()
{
    ...

    MPI_Init();

    MPI_Comm_rank(MCW, &rank);
    ncclCommInitRank(&gpucomm, nGPUs, cUID, getGPU(rank)); // Seuls
    les rangs choisis initialisent leur communicateur GPU.
    MPI_Finalize();

    ...
}
```

# Utilisation de `ncclCommInitRank` (OpenMP)

```
Int main()
{
    ...

    #pragma omp parallel
    {
        rank = omp_get_thread_num();
        ncclCommInitRank(&gpucomm, nGPUs, cUID, getGPU(rank)); // Seuls
les rangs choisis initialisent leur communicateur GPU.
    }

    ...
}
```

# Utilisation de `ncclCommInitRank`

- `/!\` Si vous plus d'un processus MPI/thread affecté à un GPU, il faut faire bien attention de n'appeler qu'une seul fois `ncclCommInitRank` pour ce GPU.
- L'argument passé à `ncclCommInitRank` est l'identifiant du GPU, et non le rang du processus MPI/thread.

# Utilisation de ncclCommInitAll (MPI)

```
int clique = {0,3,1,5}
```

```
int main()
```

```
{
```

```
    ncclComm_t gpucomm [4]; // Autant de communicateurs nccl que de GPUs  
    dans la clique
```

```
    MPI_Init();
```

```
    MPI_Comm_rank(MCW, &rank);
```

```
    if(rank == constante)
```

```
    {
```

```
        ncclCommInitAll(&gpucomm, nGPUs, clique); // on n'appelle cette fonction  
    qu'une seule fois
```

```
    }
```

```
    MPI_Finalize();
```

```
}
```



# Utilisation de ncclCommInitAll (OpenMP)

```
int clique = {0,3,1,5}
```

```
int main()
```

```
{
```

```
    ncclComm_t gpucomm [4]; // Autant de communicateurs nccl que de GPUs  
    dans la clique
```

```
    ncclCommInitAll(&gpucomm, nGPUs, clique); // On n'appelle cette fonction  
    qu'une seule fois
```

```
    #pragma omp parallel
```

```
    {
```

```
        ...
```

```
    }
```

```
}
```

# NCCL (3)

- Les fonctions de communications collectives ont des signatures quasiment équivalentes à MPI
- Chaque GPU doit faire appel à la même fonction
- Ce sont des appels asynchrones
  - Il est possible que le même processus/threads fassent tous les appels
  - Sélection du GPU + appel de fonction

# Fonctions collective: allreduce

- `ncclResult_t ncclAllReduce(  
 void* sendoff,  
 void* recvbuff,  
 int count,  
 ncclDataType_t type,  
 ncclRedOp_t op,  
 ncclComm_t comm,  
 cudaStream_t stream);`

# Fonctions collective: allreduce

## NCCL

- `ncclResult_t ncclAllReduce(  
 void* sendbuff,  
 void* recvbuff,  
 int count,  
 ncclDataType_t type,  
 ncclRedOp_t op,  
 ncclComm_t comm,  
 cudaStream_t stream);`

## MPI

- `int MPI_Allreduce(  
 void *sendbuf,  
 void *recvbuf,  
 int count,  
 MPI_Datatype datatype,  
 MPI_Op op,  
 MPI_Comm comm);`

# Utilisation de fonction de communications coll. (MPI 1)

```
int clique = {0,3,1,5}
Int main()
{
    MPI_Init();
    MPI_Comm_rank(MCW, &rank);
    ... // Initialisation des communicateurs nccl
    if(inClique (getGPU(rank)) // Soit les rangs concernés font l'appel
    {
        ncclAllReduce(&sendbuf, &recvbuf, count, type, op, gpucomm);
    }
    MPI_Finalize();
}
```

# Utilisation de fonction de communications coll. (MPI 1)

```
int clique = {0,3,1,5}
Int main()
{
    MPI_Init();
    MPI_Comm_rank(MCW, &rank);
    ... // Initialisation des communicateurs nccl
    if(inClique (getGPU(rank)) // Soit les rangs concernés font l'appel
    {
        ncclAllReduce(&sendbuf, &recvbuf, count, type, op,
gpucomm[getGPU(rank)]);
    }
    MPI_Finalize();
}
```

# Utilisation de fonction de communications coll. (MPI 2)

```
int clique = {0,3,1,5}
Int main()
{
    MPI_Init();
    MPI_Comm_rank(MCW, &rank);
    ... // Initialisation des communicateurs nccl
    if(rank==0) // Soit un seul rang réalise les appels de tous les GPUs concernés
    {
        for(i=0; i<nbGPUs; i++)
        {
            if(inClique (i))
            {
                ncclAllReduce(&sendbuf, &recvbuf, count, type, op,
gpucomm);
            }
        }
    }
    MPI_Finalize();
}
```

# Utilisation de fonction de communications coll. (MPI 3)

```
// Si tous les GPUs sont concernés, il n'y a plus de sélection/vérification à faire: code plus simple
Int main()
{
    MPI_Init();
    MPI_Comm_rank(MCW, &rank);
    ... // Initialisation des communicateurs nccl
    if(rank==0)
    {
        for(i=0; i<nbGPUs; i++)
        {
            ncclAllReduce(&sendbuf, &recvbuf, count, type, op, gpucomm);
        }
    }
    MPI_Finalize();
}
```



# Utilisation de ncclCommInitAll (OpenMP 1)

```
int clique = {0,3,1,5}
Int main()
{
    ... // Initialisation des communicateurs nccl
    #pragma omp parallel
    {
        rank = omp_get_thread_num();
        if(inClique (getGPU(rank)) // Soit les rangs concernés font l'appel
        {
            ncclAllReduce(&sendbuf, &recvbuf, count, type, op,
                           gpucomm);
        }
    }
}
```

# Utilisation de ncclCommInitAll (OpenMP 2)

```
int clique = {0,3,1,5}
int main()
{
    ... // Initialisation des communicateurs nccl
    for(i=0; i<nGPUs, i++)
    {
        if(inClique (i) // Soit les rangs concernés font l'appel
        {
            ncclAllReduce(&sendbuf, &recvbuf, count, type, op, gpucomm);
        }
    }
    #pragma omp parallel
    {
    }
}
```

# Etat actuel de NCCL 2.16.2

- Collectives

- Broadcast
- All-Gather
- Reduce
- All-Reduce
- Reduce-Scatter

- Point-to-point

- Send / recv
- One-to-all (scatter)
- All-to-one (gather)
- All-to-all
- Neighbor exchange

- Key Features

- Single-node and multi-nodes
- Host-side API
- Asynchronous/non-blocking interface
- Multi-thread, multi-process support
- In-place and out-of-place operation
- Automatic Topology Detection
- NVLink & PCIe/QPI\*support
- InfiniBand verbs, libfabric, RoCE and IP Socket internode communication