UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

# M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2022-2023)

# TP4

Multi-GPU
Mémoire Unifiée

adrien.roussel@cea.fr
julien.jaeger@cea.fr

## I MPI+CUDA et OPENMP+CUDA

Les fichiers *mpi_cuda_addvector.cu* et *openmp_cuda_addvector.cu* rálisent l'addition de deux vecteurs. Le calcul est effectué deux fois : une fois en répartissant le calcul sur les rangs MPI (respectivement OpenMP), puis via un kernel CUDA. Les deux temps de calculs (transferts compris) sont affichés. N'hésitez pas à les exécuter une fois qu'ils seront compilés.

**Q.1:** Ces fichiers contiennent à la fois du code MPI (respectivement OpenMP) et du code CUDA. Il faut donc spécifier le compilateur à utiliser pour le code CPU afin de supporter les fonctions MPI (respectivement les pragmas et fonctions OpenMP). Dans l'aide de nvcc, trouver l'option permettant de spécifier le compilateur à utiliser. Que faut-il faire de plus pour OpenMP ?

**Q.2:** Quel est l'autre façon de compiler ces codes ?

Nous allons maintenant testé le partage de mémoire GPU entre processus MPI et threads OpenMP.

**Q.3:** Écrire un code MPI+CUDA pour lequel le processus de rang 1 dans `MPI_COMM_WORLD` alloue et écrit des données sur le GPU 0, puis envoie le pointeur vers les données CUDA au rang 0. Le rang reçoit le pointeur, puis lis les données CUDA. Les données lues sont-elles les mêmes que les données écrites ? Pourquoi ?

**Q.4:** Écrire un code OpenMP réalisant les mêmes opérations. Les données lues sont-elles les mêmes que les données écrites ? Pourquoi ?

**Q.5:** Reprendre le code OpenMP précédent, mais utiliser la fonction `cuCtxCreate` pour la sélection du GPU 0. Les données lues sont-elles les mêmes que les données écrites ? Pourquoi ?

# II Mémoire Unifiée

La Mémoire Unifiée est une fonctionnalité qui rend plus facile la gestion des données (allocation et mouvement) dans un système hétérogène. Elle est apparue avec la génération Pascal de GPUs Nvidia, mais elle est supportée, dans une forme plus limitée, sur les architectures plus anciennes (Kepler, Maxwell).

Concrétement, la mémoire unifiée est un espace d'adressage accessible depuis n'importe quel processeur présent dans le système (CPU ou GPU). Depuis l'architecture Pascal, un "Page Migration Engine" est présent au niveau hardware. Cela a pour effet de n'allouer les pages mémoires physiques qu'une fois le premier accès à la donnée effectuée, et de pouvoir migrer des pages mémoires d'une mémoire à l'autre (host ou device). Plus d'informations dans le White Paper lié à l'architecture Pascal, disponible à l'adresse :
`https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`

**Astuce :** Essayez d'allouer vos ressources sur des machines disposant de GPUs intégrant l'architecture Pascal ou ultérieure (Volta), en changeant de partitions pour vos réservations de noeuds.

Dans cet exercice, nous allons effectuer un produit scalaire entre deux vecteurs, dont l'algorithme est le suivant :

```
1 float A[N];
2 float B[N];
3 float sum = 0;
4
5 for (int i = 0; i < N; i++)
6   sum += A[i] * B[i];
```

Ce pseudo-code se décompose de 2 manières : une opération de multiplication de 2 entrées de chaque vecteur, puis une réduction sur la valeur résultat.

**Q.6:** Proposer un code CUDA permettant d'effectuer la multiplication des vecteurs sur le GPU, et ensuite la réduction sur le CPU.

**Q.7:** Utilisons maintenant la mémoire unifiée afin de s'abstraire des mouvements de données entre le CPU et le GPU. Transformez votre code à l'aide de la fonction `cudaMallocManaged` pour allouer vos vecteurs dans la mémoire unifiée.
NB : Êtes-vous sûrs que le GPU a fini ses traitements avant de commencer votre calcul sur CPU ?

**Q.8:** Évaluation des performances : encapsulez le calcul de votre produit scalaire, afin de rejouer un nombre suffisant de fois l'opération. Mesurez le temps moyen pour effectuer une opération de produit scalaire.

**Q.9:** Comparez vos résultats obtenus avec et sans l'utilisation de mémoire unifiée. Que remarquez-vous ?

**Q.10:** Dans cet exemple très simplifié, comment peut-on modifier le code pour contourner le problème ? Implémenter cette solution, et comparer les résultats avec les résultats précédents.

**Q.11:** Que retenir de cet exemple ?

**Q.12:** Plusieurs manières existent afin de charger les données (allouées avec la mémoire unifiée) dans la mémoire du GPU. L'une d'elle consiste à effectuer les opérations d'initialisation des données sur GPU (politique *first-touch*). Une deuxième consiste à faire une opération de pré-fetch entre l'initialisation des données et leur première utilisation sur GPU. Transformez votre code afin d'utiliser la fonction `cudaPrefetchAsync` pour précharger les données sur GPU.