UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2024-2025)

TP3

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* réalisent 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, trouvez l'option permettant de spécifier le compilateur à utiliser pour la partie host du code. (Utilisez par exemple `nvcc -h | grep -C 1 compiler`). Que faut-il faire de plus pour OpenMP ?

Correction

L'option pour changer le compilateur est `- - compiler-bindir`.
Pour activer OpenMP, nous devons spécifier le flag `-fopenmp` au compilateur host avec `- - compiler-options`.
MPI : `- -compiler-bindir=mpicxx`
OpenMP : `- -compiler-options=-fopenmp`

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

Correction

Il faut passer les paths vers les fichiers d'include, les paths vers les libs, et les appels aux libs. Pour OpenMP, il est encore nécessaire de passer l'option -fopenmp. Voir un exemple dans *../CORR/Partie2/Makefile*.

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

Q.3: Écrivez 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 0 reçoit le pointeur, puis lit les données CUDA sur le même GPU 0. Les données lues sont-elles les mêmes que les données écrites ? Pourquoi ?

Correction

Voir fichier *../CORR/Partie1/mpi_cuda_read_memory.cu.*
Les données ne sont pas les mêmes. Mêmes s'ils utilisent le même GPU, les processus MPI ont chacun leur propre contexte associé à ce GPU, donc des espaces d'adressage différents. L'adresse récupérée par le rang 0 ne pointe pas au même endroit en mémoire.

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

Correction

Voir fichier *..CORR/Partie2/openmp_cuda_read_memory.cu.*
Les données sont les mêmes. Les threads OpenMP utilise le même contexte primaire associé au GPU 0.

Q.5: Reprenez le code OpenMP précédent, mais utilisez les fonctions `cuInit` et `cuCtxCreate` pour la sélection du GPU 0. Les données lues sont-elles les mêmes que les données écrites ? Pourquoi ? Existe-t-il une différence selon les GPUs et la version de CUDA ?

Pensez à compiler avec `-lcuda` lorsque vous vous servez du driver CUDA.

Correction

Ça dépend du GPU et de la version CUDA utilisée !
Si vous utilisez un GPU avec une version de CUDA ancienne (e.g. < 4.0), alors les données ne devraient pas les mêmes. Mêmes s'ils utilisent le même GPU, les threads OpenMP ont maintenant chacun leur propre contexte associé à ce GPU, donc des espaces d'adressage différents. L'adresse récupérée par le rang 0 ne pointe pas au même endroit en mémoire.
Par contre, si vous utilisez une version CUDA récente, alors les contextes sont forcément primaires, ce qui signifie que tous les threads d'un même processus partagent le même espace d'adressage mémoire sur un même GPU. Ainsi les données lues sont les mêmes.

Q.6: Reprenez la question précédente, mais utilisez la fonction `cuCtxDestroy` pour le thread 1 après avoir transmis le pointeur vers le tableau de données dans le GPU.

Les données lues par le thread 0 sont-elles les mêmes que les données écrites ? Pourquoi ?

Correction

Les données lues ne sont pas les mêmes. Ici, le thread 1 a détruit son contexte avant que le thread 0 ne puisse lire les données en GPU. Donc toutes les ressources liées à ce contexte ont été détruites et nettoyées, en particulier le tableau de données alloué par le thread 1 en mémoire GPU.

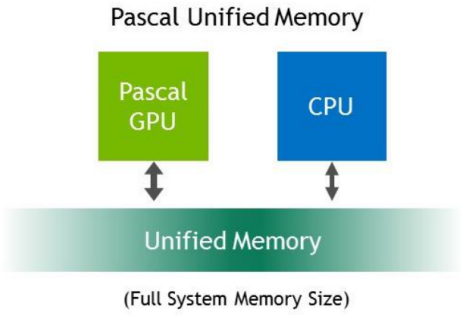

FIGURE 1 – Pascal GP100 Unified Memory

II Mémoire Virtuelle Unifiée

La mémoire virtuelle unifiée est une fonctionnalité qui rend plus facile la gestion des données (allocation et transferts) 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).

Concrètement, 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, et d'où nous tirons la figure **??** 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 nœuds.

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

```
float A[N];
float B[N];
float sum = 0;

for (int i = 0; i < N; i++)
  sum += A[i] * B[i];
```

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

Q.7: Proposer un code CUDA permettant d'effectuer la multiplication des vecteurs sur le GPU, et ensuite la réduction sur le CPU.

Q.8: Utilisez 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 sûr(e) que le GPU a bien fini ses traitements avant de commencer votre calcul sur CPU ?

Nous allons désormais tenter d'évaluer les performances de nos deux programmes. Le calcul final n'aura pas forcément de sens mathématiques. Nous cherchons uniquement à découvrir l'influence de l'utilisation de la mémoire unifiée sur la performance d'un programme aussi simple.

Q.9: Dans vos deux versions du programme (avec et sans mémoire unifiée), encapsulez le calcul du produit scalaire dans une boucle for afin de rejouer un nombre suffisant de fois le même produit scalaire. Mesurez le temps moyen pour effectuer un produit scalaire.

Q.10: Comparez vos résultats obtenus avec et sans l'utilisation de mémoire unifiée. Que remarquez-vous ?

Correction

Avec cet énoncé, les étudiants vont effectuer à chaque itération de boucle les instructions suivantes :

1. Multiplication de vecteurs.
2. Synchronisation du device.
3. Calcul de la réduction.

Entre chaque multiplication et réduction, le driver CUDA va transférer les données entre CPU et GPU de manière transparente avec la mémoire unifiée. Ce mouvement de données répété a un coût non négligeable, ce qui détériore grandement les performances.

Q.11: Dans cet exemple très simplifié, où on ne fait que répéter les mêmes opérations en boucle, comment peut-on modifier le code pour contourner le problème posé par la question précédente tout en répétant les mêmes opérations d'additions et de réductions ? Implémentez cette solution et comparez vos résultats avec les résultats précédents.

Correction

On peut regrouper toutes les opérations de multiplication de vecteurs dans une boucle, ainsi que toutes les opérations de réduction dans une autre boucle, en mettant une barrière de synchronisation entre les 2 boucles. On réduit ainsi considérablement le nombre de copies mémoires entre CPU et GPU, et donc on améliore les performances de l'application.

Q.12: Que retenir de cet exemple ?

Correction

Bien que masquées par l'utilisation de la mémoire unifiée, les copies de données entre CPU et GPU sont toujours omniprésentes. Il faut donc bien en avoir conscience lors de la conception d'une application utilisant de la mémoire unifiée.

Q.13: 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.

La signature de la fonction pour pré-fetch est la suivante : `cudaError_t cudaMemPrefetchAsync(const void *devPtr, size_t count, int dstDevice, cudaStream_t stream = (cudaStream_t)0)`. Pour charger sur le CPU, il faut utiliser la constante `cudaCpuDeviceId` pour `dstDevice`.

Transformez votre code afin d'utiliser la fonction `cudaPrefetchAsync` pour précharger les données sur GPU avant le kernel, puis de les précharger sur CPU avant la réduction. Comparez les performances à la première version de votre programme utilisant la mémoire unifiée.

Correction

Cf code fourni.