ENSIEE - Programmation GPU

(2023-2024)

TP0

Vérification des erreurs et débuggage

adrien.roussel@cea.fr
mickael.boichot@cea.fr

Dans ce TP nous allons prendre en main l'API runtime CUDA et les outils NVIDIA disponibles pour trouver les caractéristiques des GPUs disponibles, vérifier les retours d'erreurs et débugguer un programme CUDA.

Les objectifs de ce TP sont :

- Trouver les caractéristiques des GPUs disponibles grâce au SDK CUDA
- Vérifier les erreurs de l'API runtime CUDA et des appels de kernel
- Utiliser le débuggueur GPU cuda-gdb
- Utiliser l'outil Compute Sanitizer pour trouver l'origine d'un accès mémoire illégal dans un kernel

I Caractéristiques des GPUs disponibles

Q.1: Nous pouvons obtenir des informations basiques concernant les GPUs installés sur une machine en lançant *Nvidia System Management Interface* avec la commande *nvidia-smi*. Lancez cette commande pour lister les GPUs disponibles sur votre machine. Quelles autres informations renvoie cette commande ?

Q.2: Le SDK CUDA dispose de fonctions pour trouver les caractéristiques des GPUs disponibles sur une machine. Par exemple, nous pouvons utiliser

- *cudaError_t cudaGetDeviceProperties(cudaDeviceProp* prop, int device)* qui retourne les propriétés d'un GPU dans la structure *prop*
- *cudaError_t cudaDeviceGetAttribute(int* value, cudaDeviceAttr attr, int device)* qui retourne une information concerant un GPU dans *value*

Le code *device_query/prog.c* recherche et affiche les informations des GPUs disponibles sur une machine. Compilez ce code et lancez le binaire sur votre machine. Quel est le nombre maximal de threads par bloc pour votre GPU ?

Q.3: Complétez le code *device_query/prog.c* pour afficher la bande passante maximale de la mémoire globale de votre GPU.

Q.4: Complétez le code *device_query/prog.c* pour afficher le nombre maximal d'opérations flottantes simple précision (FP32) par secondes que peut réaliser votre GPU. Pour cela, considérez qu'à chaque cycle chaque coeur de votre GPU peut réaliser deux opérations flottantes FP32 (grâce aux instructions Fused Multiply-Add).

II Vérification des erreurs

Dans cet exercice, nous allons apprendre à vérifier les erreurs dans un programme CUDA.

Q.5: Tous les appels au SDK CUDA retourne un code d'erreur de type *cudaError_t*. *cudaError_t* est un type *enum* couvrant toutes les codes d'erreurs possibles. *cudaGetErrorString(cudaError_t err)* convertit un code d'erreur en une chaîne de caractères compréhensible par l'utilisateur.

Dans le header *error_checking/cuda_helper.h* nous avons défini une macro vérifiant le code d'erreur d'un appel au SDK CUDA.

La bonne pratique est de toujours vérifier les retours d'erreurs de vos appels au SDK CUDA. C'est ce que nous avons mis en place dans le fichier *error_checking/exemple1.cu* qui additionne deux vecteurs sur GPU et vérifie sur CPU que le résultat est correct. Compilez et lancez le programme *error_checking/exemple1.exe*. Que se passe-t-il ? A votre avis, où s'est produite l'erreur ?

Q.6: Un lancement de kernel *kernel«<...»>(...)* n'est pas un appel au SDK CUDA et ne retourne pas de code d'erreur. Les lancements de kernel CUDA sont asynchrones. Le kernel ne commence pas nécessairement son exécution sur GPU tout de suite. De son côté, le thread CPU qui lance le kernel continue son exécution sans attendre que le kernel termine sur GPU.

Un lancement de kernel CUDA peut produire deux types d'erreur

— synchrone, i.e. détectable au lancement

— asynchrone, i.e. se produisant pendant l'exécution du kernel sur GPU

Les erreurs de type synchrone peuvent être détectées juste après un lancement de kernel gâce à *cudaGetLastError()* ou *cudaPeekAtLastError()*.

Modifiez le fichier *error_checking/exemple1.cu* pour vérifier les erreurs synchrones après le lancement de kernel. L'erreur présente dans ce programme était-elle bien synchrone ? Si oui corrigez la.

Q.7: Le programme *error_checking/exemple2.cu* lance un kernel qui produit une erreur asynchrone.

Le seul moyen de détecter l'erreur produite par le kernel est de forcer le CPU à attendre la fin du kernel puis de vérifier l'erreur.

C'est ce que nous avons effectuer dans le fichier *error_checking/exemple2.cu* avec un appel à *cudaDeviceSynchronize()*. Quelle est l'erreur produite par le programme ?

Remarques :

— Rajouter un appel à *cudaDeviceSynchronize()* modifie cependant le comportement asynchrone du code et est donc à réserver au débuggage du code.

— En fixant la variable d'environnement *CUDA_LAUNCH_BLOCKING* à 1, vous pouvez forcer les lancements de kernel à être synchrone.

Q.8: Rajouter un appel à *cudaMalloc* après l'appel de kernel dans *error_checking/exemple2.cu* et vérifier le code retour d'erreur de cet appel à l'API CUDA. L'objectif est de déterminer si l'API runtime CUDA est toujours utilisable après l'erreur provoqué par le kernel.

Est-ce que l'appel à *cudaMalloc* que vous avez rajouté a réussi ? Si non, quelle est l'erreur reportée ? Qu'en concluez-vous ?

Q.9: Le programme *error_checking/exemple3.cu* effectue volontairement une allocation sur GPU dépassant la capacité du GPU avant d'effectuer d'autres appels à l'API CUDA et un lancement de kernel.

Est-ce que les appels à l'API CUDA effectués après l'allocation illégale sur GPU se déroule normalement ? Qu'en concluez-vous ?

Remarque : Un récapitulatif des notions vues dans cet exercice est disponible ici.

III Compute Sanitizer et CUDA-gdb

Compute-Sanitizer est un outil fourni avec le SDK CUDA pour vérifier la correction de vos programmes CUDA. Quand votre code est lancé avec l'outil, les erreurs de l'API runtime sont automatiquement vérifiées même si votre code ne le fait pas.

L'outil est composé de plusieurs sous-outils :

- **memcheck** : détecte les actions illégales dans votre code (instructions illégales, accès mémoire illégaux, etc.)
- **racecheck** : détecte les accès mémoire concurrents dans la shared memory sans synchronisation
- **initcheck** : détecte les accès mémoire non intialisée à la mémoire globale
- **synccheck** : détecte l'utilisation illégale de primitive de synchronisation comme *___syncthreads()*.

Les options de ligne de commande de l'outil sont décrites ici.

Q.10: Nous allons tout d'abord utiliser l'outil *memcheck*. Notez que pour avoir des localisation d'erreur, il faut compiler avec *-lineinfo*. Cet outil a un impact sur le temps d'exécution des kernels. Compilez le code *compute-sanitizer/memcheck.cu* et lancez l'outil sur le binaire. Le code ne fait volontairement pas de vérification d'erreur pour que vous constatiez que Compute-Sanitizer le fait pour vous. Où est localisée l'erreur dans le programme ? Si l'erreur se produit dans un kernel, quel thread CUDA est fautif ?

Q.11: CUDA-gdb est une extension de gdb qui permet de débugguer du code CPU et GPU. Le code doit être compilé avec les flags

- *-g* pour avoir les informations de debug du code host
- *-G* pour avoir les informations de debug du code device

Le flag *-G* a un impact significatif sur les performances du code device. Il ne doit être utilisé qu'à des fins de debug.

Des commandes spécifiques à CUDA sont ajoutés aux commandes classiques de gdb. Par exemple

- *set cuda* permet d'activer des options
  - *launch_blocking (on/off)* rend les lancements de kernels synchrones (i.e. le thread host se met en pause et attend la fin du kernel)
  - *break_on_launch (option)* demande au débuggueur de faire un break à la première instruction de chaque kernel lancé.

— *info cuda* permet d'obtenir des informations générales sur la configuration du système
— *cuda* permet d'inspecter ou de sélectionner quelle device et quel thread on suit
  — *cuda device sm warp lane block thread* affiche quel thread on suit
  — *cuda block (1) thread (3)* permet de suivre le thread 1 du bloc 3.

La documentation complète de cuda-gdb est disponible ici.

Compilez *compute-sanitizer/memcheck.exe* avec les bonnes options et lancez le avec cuda-gdb. Activez l'option pour faire un break à chaque lancement de kernel.

Testez les options permettant d'inspecter et de sélectionner quel thread le débuggueur suit.

Q.12: Lancez *compute-sanitizer/memcheck.exe* et fixez un breakpoint conditionnel sur le thread qui cause l'erreur. Corrigez l'erreur.

Q.13: Le code *compute-sanitizer/initcheck.cu* lit une variable en mémoire globale non initialisée. Compilez le programme et lancez l'outil *initcheck*. La lecture de variable non initialisée est-elle bien détectée ? Initialisez la variable et relancez l'outil.