Architecture et programmation d'accélérateurs matériels

Julien Jaeger [*] Patrick Carribault

Table des matières

[*] julien.jaeger@cea.fr

1 Introduction

Le but sera à la fois l'introduction aux architectures des accélérateurs matériels et leur exploitation par le langage CUDA.

Définition 1 (Accélérateur matériel)**.** *Une Accélérateur matériel est une ressource dédiée spécialisé pour une utilisation spécifique, qui est plus rapide. En contrepartie, il est moins général que le processeur.*

Les accélérateurs sont reliés au(x) processeur(s) grâce à une interconnexion, soit via le bus système (passer par la carte mère) ou via le bus processeur. Ils agissent comme de périphériques, il faut donc explicitement en faire usage.

Il faut distinguer *architecture* et *micro-architecture*. La première est tout ce qui est visible depuis l'extérieur du CPU (le jeu d'instruction par exemple ; quand la seconde est totalement transparente à l'utilisateur (notamment les caches). La micro-architectures est tout un tas de systèmes complexes prévus pour accélérer. Les accélérateurs utilisent très peu de choses en micro-architectures : ils sont plus complexe extérieurement, mais l'optimisation sera à réaliser par l'utilisateur (la performance crête augmente mais la complexité de développement également).

L'accélérateur est un périphérique *esclave*. Il en existe de différentes sorte, dont les GPU. A la base, ils servaient aux jeux vidéos (afficher des pixels sur un écrans), car les projections utilisées pour afficher les images 2D étaient trop couteuses pour les CPU. Depuis quelques temps, on utilise ces GPU avec une pile logicielle différente pour faire du calcul uniquement.

Intel avait voulu concurrencer NVIDIA avec *Larabee*, une carte graphique sur jeu d'instruction x86. Le projet n'a pas vu commercialement le jour, et a été réutilisé en Xeon Phi (KNH : premiers proto distribué à peu de gens (dont le CEA) puis KNC (public) à qui succède KNL). On parle alors de MIC (*Many Integrated Core*).

Le processeur CELL d'IBM (dans la Playstation 3) est constitué d'un processeur généraliste accolé à 8 SPE (Synergistic Processor Element) permettant les calculs vectoriel. Initialement, le CELL devait être le seul processeur graphique, mais il a du être épaulé d'un accélérateur AMD pour épauler le processeur.

Les FGPA sont des accélérateur programmables par une configuration HDL (Hardware Description Language). Il s'agit de bouts de silicium programmables ; on commence à voir des processeurs possédant des portions programmables.

Focus sur les cartes graphiques NVIDIA

La puissance crête est désormais bien plus grande sur un GPU qu'un CPU. De même, la bande passante est bien plus grande sur un GPU, qui est à la base là pour transmettre directement les données. La brique de base est un cœur de calcul très simple prévu pour fonctionner sur un flux, c'est pourquoi ils sont regroupé en *streaming multiprocessor* ou *multiprocesseur de flux*.

L'architecture Kepler Kepler possède 15 SMs (nommés SMx), qui se comportent comme des coeurs de calculs ; chacun ne pouvant pas exécuter des instructions différentes à la fois. Chaque SM possède 192 cœurs compatibles CUDA (unité ALU et unité flottante simple précision), 64 unités flottantes double précision, 32 unités spéciales (SFU, capables de faire des logarithmes, exponentielles, etc), et 32 unités *load/store*.

Le instructions sont de type SIMT : *single instruction multiple thread*, similaire au SIMD, avec une exécution synchrone des cœurs de calcul. Chaque SMX gère et exécute les threads, en les divisant par paquet de 32 qui exécutent la même chose au même moment : il s'agit de *warp* de thread, chaque SMX contenant 4 ordonnanceurs de warps : on extrait un paquet de 32 threads et on le map sur 32 cœurs.

Le Gigathread Engine est spécialement conçu pour créer et détruire les threads (il est rentable de créer un thread uniquement pour une addition) : le switch de contexte est donc quasiment gratuit.

Il y a une hiérarchie mémoire ; la carte graphique possédant sa propre mémoire séparée de celle du processeur. L'accès se fait par un `load` ou un `store`, via deux caches (L1 et L2). Il existe un *scratch pad* qui possède le même temps d'accès que du L1 (c'est physiquement au même endroit) de 64 ko par SM ; le cache L2 étant partagé pour toute la carte.

Utilisation Langages possibles :
— CUDA
— OpenCL
Ou encore ajout de directives de programmation :
— OpenMP avec les `pragma target` permettant l'exécution du le device
— OpenACC

2 NVIDIA CUDA

CUDA utilise de la programmation par *kernel*, en deux parties : une partie hôte et des noyaux de calcul. L'application hôte sur CPU fera des appels via l'API CUDA.

Le schéma classique de programmation CUDA est le suivant :
— Initialisation sur la partie hôte (allocation, lectures, etc)
— Allocation mémoire sur le GPU (manuellement !)
— Transférer les données utiles sur le GPU
— Exécuter un noyau de calcul
— Transfert de données du GPU vers l'hôte

Comment compiler ? Le code est en deux parties : code hôte (C/C++) et device (CUDA, sur-ensemble du C99). En pratique, ces deux codes peuvent être mélangée au sein d'un même fichier.

On va utiliser le compilateur `nvcc`.

Sur l'hôte, on a accès à la fonction `cudaMalloc` et `cudaMemcpy` pour respectivement allouer et transférer des données vers de device ou l'hôte (ou diverses variantes).

Chaque thread CUDA va lancer la même fonction ! Tous les threads auront la même fonction (comme pour les directives OpenMP). Par défaut, tout sera parallèle !

Pour connaître l'identifiant d'un thread, CUDA definit une notion de *block* et *grid* les hiérarchisant. Les grilles contiennent des blocs qui contiennent des threads, la grillé pouvant avoir de 1 à 3 dimensions (séparation logique et non physique. On repère les blocs par leur coordonnées cartésiennes dans la grille, et les threads leurs coordonnées cartésiennes dans le bloc.

Un kernel est une fonction ne retournant rien se déclarant par le préfixe `__global__`, indiquant le point d'entrée du nœud de calcul. On peut accéder aux indices/dimensions du block/thread par les structures `blockIdx`, `blockDim`, `threadIdx`, `gridDim` via leurs membres `x`, `y`, `z`.

Pour invoquer le kernel, on va l'appeller de la manière suivante :

`my_kernel«<Dg, Db»>(arg1, arg2, ..)` où `Dg` et `Db` sont les dimensions et la taille de la grille et des blocs (de type `dim3`).

Une fois l'invocation lancée, on ne sait pas où cela en est. on peut attendre la fin de l'exécution par l'appel de `cudaThreadSynchronize();`.

Il y a cependant bien une file d'attente appelée *stream* : les noyaux sont exécutés dans l'ordre. On peut déclarer d'autres streal et ainsi appeler `my_kernel«<Dg, Db, streamToUse»>(arg1, arg2, ..)` (les streams devant être déclarés au préalable).

Timing et suivis du programme Le type `cudaEvent_t` permet de mesurer les écarts de temps entre différentes exécutions de kernels. Sur le GPU, il n'y a pas de variables statiques et mieux vaut éviter la récursion. Les mots-clefs `__device__`, `__constant__` et `__shared__` existent pour diriger le comportement de stockage des variables.

On peut également définir des fonction dédiée sur l'hôte avec le mot-clef `__host__` et sur le device avec le mot clef `__device__`.

Le mot-clef `volatile` permet de générer un accès mémoire à chaque utilisation de la variable, ce qui permet au compilateur de ne pas optimiser, ce qui est utile pour les synchronisations. CUDA définie des nouveau type de données vectorielles tel `int2`.

La documentation CUDA est bien faite, il est conseillé d'y aller pour voir les variables disponibles dans un thread par exemple (indices de position, taille de warp, etc).

Au niveau des barrières mémoires, on peut synchroniser les threads d'un même bloc ; attention au flot de contrôle par contre !

Attention, pour les calculs mathématiques, l'ordre des opérations diffère entre CPU et GPU : on peut donc avoir des résultats différents suite au portage des applications.

Il est possible de timer sur les threads via la fonction `clock()`.

Il n'y a qu'un nombre restreint d'unité de chargement mémoire. Certes, il y a du *Load coalescing* permettant la fusion de requêtes de `load` (et stores) contigües.

Le nombre de registre est limité dans un bloc (il est limité par SM). Pour optimiser les perf d'un thread, on a tendance à utiliser beaucoup de registre, mais du coup on perd en nombre de thread par bloc. On peut définir le nombre de registre par bloc utilisés à la compilation et utiliser des attributs donnant le nombre max de thread à l'exécution.

Lors de chemins d'exécutions différents au sein d'un même warp Tous les chemins d'exécution sont effectués, mais du coup certains pour rien....

3 OpenCL

Chaque device contient des *Compute Units* qui contiennent des *elements*. Le principe d'exécution est le même que CUDA. L'appel d'un noyau de calcul suit le même principe, et on utilise des grilles et work-group.

Le code des kernels est compilés à la volé à l'exécution. De plus, le code est plus verbeux, portable et bas niveau. Cependant OpenCL est moins rapide que CUDA (volonté de NVIDIA...).