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...).

4 APIs CUDA et Programmation Multi-GPU

4.1 Différentes APIs pour CUDA

Il existe en fait deux APIs pour CUDA, une haut et une bas niveau, la bas niveau étant plutôt réservée aux programmeurs de bibliothèques. On peut même exécuter directement des bibliothèques très optimisées sans passer par CUDA (transparence totale, comme cuBLAS). L'API runtime est celle utilisée en TP. Elle est haut niveau (cudamalloc, cudafree, etc). Il existe également l'API driver, plus verbeuse et destinée à des utilisateurs plus avertis.

Les fonctions de l'API runtime sont préfixés par `cuda`. Le niveau d'abstraction est élevé, et permet de ne pas exposer tous les mécanismes CUDA aux utilisateurs.

A contrario, l'API driver nécessite une plus grande expertise, et peut isoler les développements CUDA dans un bibliothèque du reste du programme. L'idée est de pouvoir totalement s'abstraire de ce qui est fait dans le reste du code, afin de ne pas être gêné par le code de l'utilisateur (il ne faudrait pas que les perf sont amoindries à cause d'opération a priori indépendantes de l'utilisation de l'a bibliothèque).

Les fonction de l'API driver sont préfixés par `cu` : `cuMemAlloc`, `cuMemcpyDtoH`, ...

4.2 Contextes CUDA

Il s'agit de la structure interne de CUDA attaché à un GPU. De base, tout va s'exécuter sur le GPU 0. En fait, le runtime crée un contexte par défaut, attaché au GPU 0. A chaque appel de CUDA, les fonctions vont chercher dans le contexte actuelle sur quel GPU appliquer les fonctions.

Un contexte encapsule tous les objets relatifs au bon fonctionnement de CUDA : allocations mémoires, espace d'adressage, stream CUDA, événements CUDA.

Il existe deux types de contextes : les contextes (calssiques) et les contextes *primaires*. L'API runtime ne manipule que des contextes primaires, alors que l'API driver utilise des contextes classiques. Chaque type de contexte est stocké comme dans une pile (le runtime CUDA n'est pas open source donc on ne sais pas exactement le fonctionnement interne).

4.2.1 Contextes primaires

Les contextes primaires sont disponibles depuis CUDA 4.0. Il s'agit d'un contexte commun à tous les threads d'un processus ciblant le même GPU. Cela permet de réutiliser les données d'un thread à l'autre et éviter de stocker plusieurs fois les mêmes données sur GPU. On peut créer un nouveau contexte primaire par l'appel de `cudaSetDevice()`. Si un contexte attaché au GPU existe déjà, on va simplement aller le rechercher.

On peut le voir comme :

— Une pile de contexte par processus UNIX, qui fonctionne comme une variable globale

— (Pas dans les versions les plus récentes de CUDA) tous les threads d'un processus sont attachés au même contexte. Actuellement il s'agirait plutôt d'une liste.

4.2.2 Les contextes classique

Il s'agit des contextes par défaut de l'API avant CUDA 4.0. Le contexte est privé à chaque thread OS. On peut créer un nouveau contexte par l'appel à `cuCtxCreate()`, en spécifiant le GPU attaché. *Peu importe l'état de la pile*, on crée bien *toujours* un *nouveau* contexte. Il faut donc bien faire attention du contexte dans lequel on a appelé de kernel !

Il existe une pile de contextes par threads systèmes, stockée dans la TLS (thread local storage) du thread noyeau. Il faut donc faire la gestion des contextes *à la main*.

Il y a une *précédence* sur les contextes primaires : les contextes classiques sont prioritaires sur les contextes primaires. Il faut donc faire attentions aux situations de famine...

4.3 Programmation Multi-GPU

4.3.1 Sélection d'un GPU en CUDA

Comme vu précédemment :

— `__host__ cudaError_t cudaSetDevice(int device)` où `device` est le numéro du device utilisé pour les exécutions. Attention, ce numéro est l'identifiant CUDA ! Il faut également ne pas oublier que tous les pointeur utilisé auparavant deviennent *non valide*. Cette fonction vérifie s'il n'existe pas déjà un contexte déjà associé au device demandé. Si il existe une pile de contextes classiques, il va prendre le premier contexte classique associé au GPU ou en créer un s'il n'y en a pas ! Sinon, il sélectionne le contexte primaire correspondant ou en créer un s'il n'en existe pas. On ne peut donc pas utiliser le même GPU sur le même noeud avec le même processus (surtout pour utiliser le même GPU que celui en train d'être exploité par une bibliothèque CUDA). Si le device n'existe pas, il semble que les codes sont exécutés sur le device par défaut (0).
Pour obtenir le nombre de devices utilisables, on utilise `__host__device__cudaError_t cudaGetDeviceCount(int * count)` retournant le nombre de cartes supportant CUDA 2.0.

— `CUresult cuCtxCreate(CUcontext *pctx, unsigned int flags, CUdevice dev)` où `pctx` le handler du nouveau contexte, `flags` les options de créations et `dev` le périphérique à utiliser. Le contexte est alors pushé en tête de pile de thread appelant. Pour manipuler la pile, on utilise les fonctions `CUresult cuCtxPopCurrent(CUcontext *pctx)` enlevant le denier contexte de la pile et le renvoyeant dans `pctx`. A l'inverse, la fonction `CUresult cuCtxPushCurrent(CUcontext ctx)` permet de mettre `ctx` sur la pile. Les appels de fonctions se font donc sur le contexte en tête de pile.
D'autres fonctions sont disponibles : `cuCtsGetCurrent` retournant le contexte courant, `cuCtxGetDevice()` retournant l'ID du device courant et `cuCtxSetCurrent()` pour remplacer le contexte courant (pop + push)
La gestion fine des contextes classique doit donc se faire à la main, à moins d'utiliser l'API runtime (`cudaSetDevice()` va chercher tout seul dans la pile !)

4.3.2 CUDA et MPI

Par défaut, tous les processus MPI utilisent le GPU par défaut (GPU 0). Au sein d'un noeud, le *même* GPU sera tout le temps utilisé ! Le problème ne change pas si l'on fait un setdevice constant. On peut donc tenter un `cudaSetDevice(rank)`. Il y a cependant un léger soucis si le rang maximal est plus grand que le nombre de GPU, que l'on peut facilement contourner par un modulo (Roud-Robin). Cela permet de maximiser la répartition des processus MPI sur les différents GPUs disponibles, mais cela ne maximise pas forcément l'occupation de ces GPU. L'affectation la plus efficace est donc machine-dépendant à cause de l'effet NUMA.

de plus, les processus utilisant le même GPU ne partagent pas le même espace d'adressage : il faudrait donc partager le même contexte, ou mettre en place des redondance de données. Dans l'idéal, il faudrait mettre en place un broadcast du contexte, non prévu par CUDA. Il faudrait sinon utiliser des thread primaires, à condition d'utiliser des threads OS (APMI ou MPC).

4.3.3 CUDA et OpenMP

Deux méthodes sont possible.

— Avec l'API runtime, avec des contextes primaires, sans soucis de communications inter-threads
— Avec l'API drivers, on récupère le rang et on crée le contexte ne fonction du rang. Le contexte sera par contre indépendant par thread, la mémoire du GPU ne sera pas partagée !

On retrouvera les mêmes problématiques qu'en MPI+CUDA. Cela ne maximise pas forcément l'utilisation des GPU (nécessite de connaître la charge de travail).

Un bonne manière est d'utiliser MPI + OpenMP + CUDA :

— Utiliser un process MPI par GPU disponible
— Répartir sur les threads via OpenMP
— On utilise directement l'API Runtime et des contextes primaires.

Cependant, chaque appel CUDA dans un processus MPI sera sérialisé avec les autres appels des autres threads. Au sein d'un stream, tout est sérialisé ! Les kernels sur le même stream vont juste se faire enfiler dans la file du stream en attendant de se faire exécuter. Pour palier à ce problème, on peut utiliser un stream par unique à chaque thread. On ne peut cependant utiliser que 16 streams maximum !

Il faut (toujours) régler l'effet NUMA en plaçant les processus sur les coeurs attaché au GPU. Parfois ce n'est pas possible (2 GPU sur la même socket).

La répartition des processus n'est donc pas toujours celle qui maximise les performances CPU ! De plus, des bibliothèques externes peuvent manipuler les GPU et contextes cassant la répartition optimale.

4.4 NCCL

Permet, à la manière de MPI, de déplacer des données d'un GPU à un autre. On regroupe les GPU dans des *clique*. On reprend les même collectives et quasiment les même signatures qu'MPI. Les appels sont asynchrones, et il faut faire bien attention à qui fait quoi lors d'une utilisation conjointe avec MPI.

5 Programmation par directives et Task-Scheduling

Programmation par directives : le code est annoté par l'utilisateur de manière à ce que le code soit enveloppé à la compilation par certaines fonctions.

5.1 Programmation par directives

5.1.1 OpenMP

Modèle fork-join : au début d'une région parallèle, on réveille/crée les threads, on exécute puis on les tue/endors.

La programme est très simple, il suffit de connaitre un pragma et deux fonctions. Il faut cependant synchroniser manuellement, et d'autre options sont possible afin de d'exécuter séquentiellement certaines sections ou changer certains options.

Pour envoyer des données, on utilise la directive `#pragma omp target data if() device() map() use_device_ptr()`. Elle a un scope (accolade ouvrante et fermante).

Pour mettre à jour les données, on utilise `#pragma omp target update ...`.

Pour exécuter la région, on utilise `#pragma omp target if() device() private() firstprivate() map() is_device_ptr() defaultmap() nowait depend()`. Elle contient un `omp target data` pour envoyer les données avant exécution. Les variables privées ne seront pas partagées entre les threads. Firstprivate permet en plus d'initialiser à la valeur précédente.

CUDA propose trois niveau de parallélisme, alors qu'OpenMP ne propose que deux niveaux : Team et Thread, le troisième étant implicite (nombre de team). La directive `#pragma omp teams num_teams() thread_limit() default() private() firstprivate() shared() reeduction()`.

Jusqu'à présent, on utilisait `parallel` pour lancer une région parallèle. En fait, cette directive crée une seul team de threads, et chacun exécute la région parallèle. On a donc besoin de nouvelles directives. Le même problème se porte avec la directive `for`.

On utilise alors la directive `#pragma omp distribute private etc` qui se place avant une directive for. Cela permet de réparti la boucle sur les différentes teams.

5.1.2 OpenACC

Etait censé être un standard temporaire pour alléger la programmation CUDA+OpenCL, pour être intégré plus tard dans OpenMP. Les directives pour accélérateurs étaient tout de même légèrement différentes, et depuis, les deux standards coexistent, bien qu'il y ait des discussions de merge à ce sujet.

Il s'agit une fois encore de pragma. Voir les slides pour les détails d'implémentation.

La hiérarchie d'OpenACC définit des Gang qui contiennent des workers qui contiennent des vectors.

Attention, la hiérarchie des niveaux dépend totalement de l'implémentation du runtime !

Il n'y a donc que deux directives, mais il faut bien savoir les utiliser !

OpenACC possède en plus une directive `kernel` qui permet d'automatiser le parallélisme dans une région.

5.2 Task-scheduling

StarPU est un runtime à base de tâches, organisées dans un graph. StarPu reconnait les architectures hétérogènes, et permet l'ordonnancement dynamique en évitant au maximum les transfert inutiles.