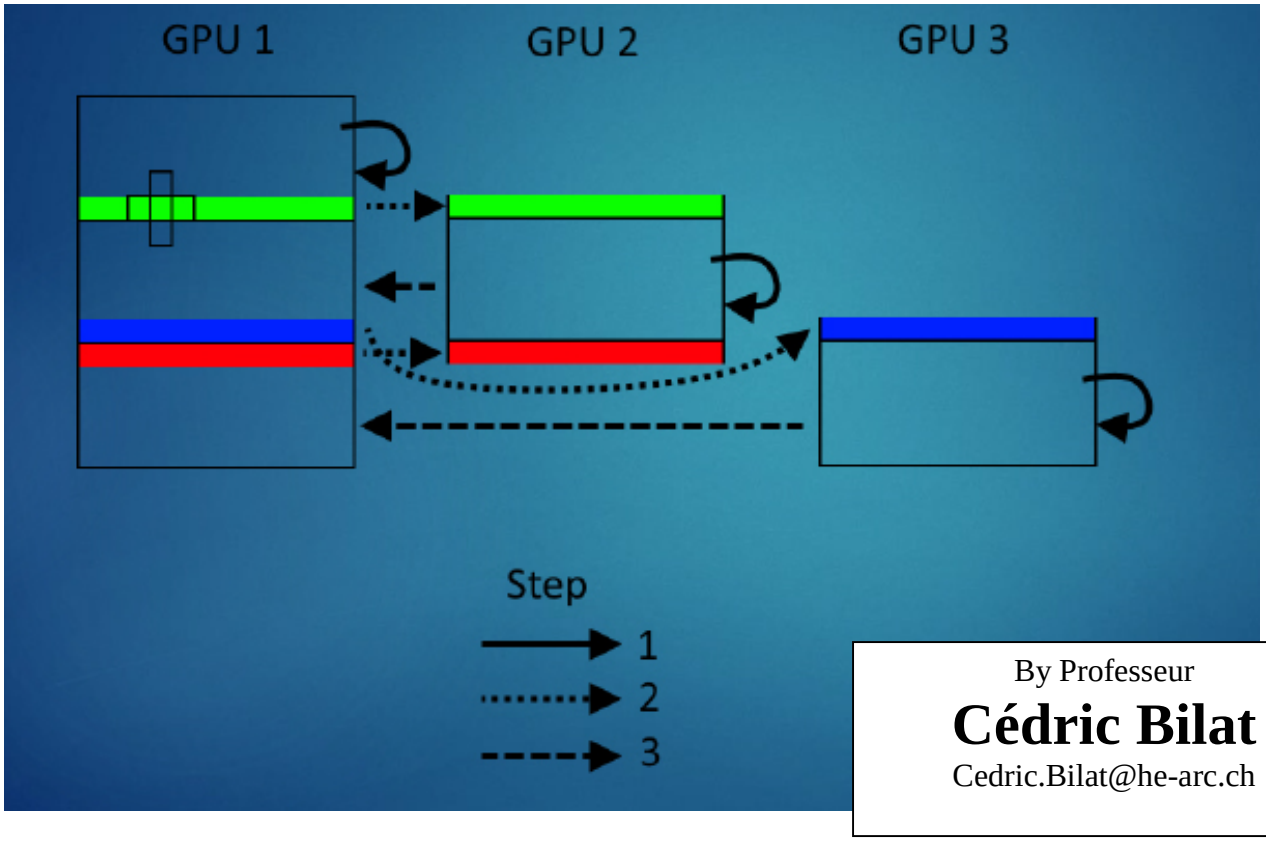
Problèmes Parallelisation



By Professeur
Cédric Bilat
Cedric.Bilat@he-arc.ch

Multi-GPU

Principe

Degré de parallélisme

Dans le cas d'une implémentation GPU, il serait intéressant de répartir la charge sur non pas un seul GPU, mais sur tous les GPU disponibles.

On aimerait employer tous les cores de tous les MP de tous les GPU !

Classes

On peut distinguer au moins trois classes de TP multi-GPU :

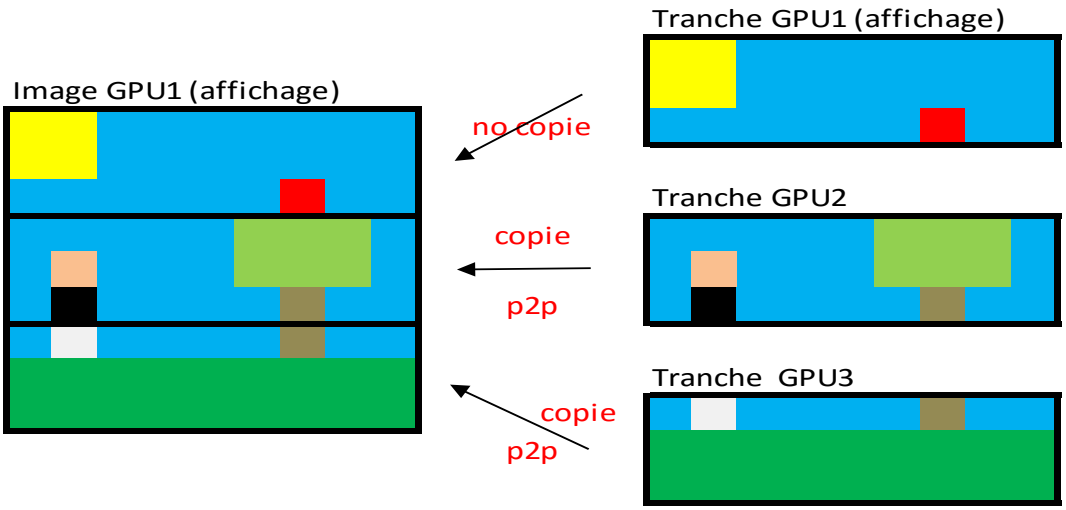
- Montearlo
- Mandelbrot
- Convolution

Classe ***Montecarlo***

C'est le cas le plus simple. Chaque GPU exécute le même kernel. Chaque GPU travaille indépendamment des autres. Seule à la fin une étape de synchronisation est nécessaire pour effectuer coté CPU une étape de pour consolider les résultats partielles de chacun des GPU.

Classe ***Mandelbrot***

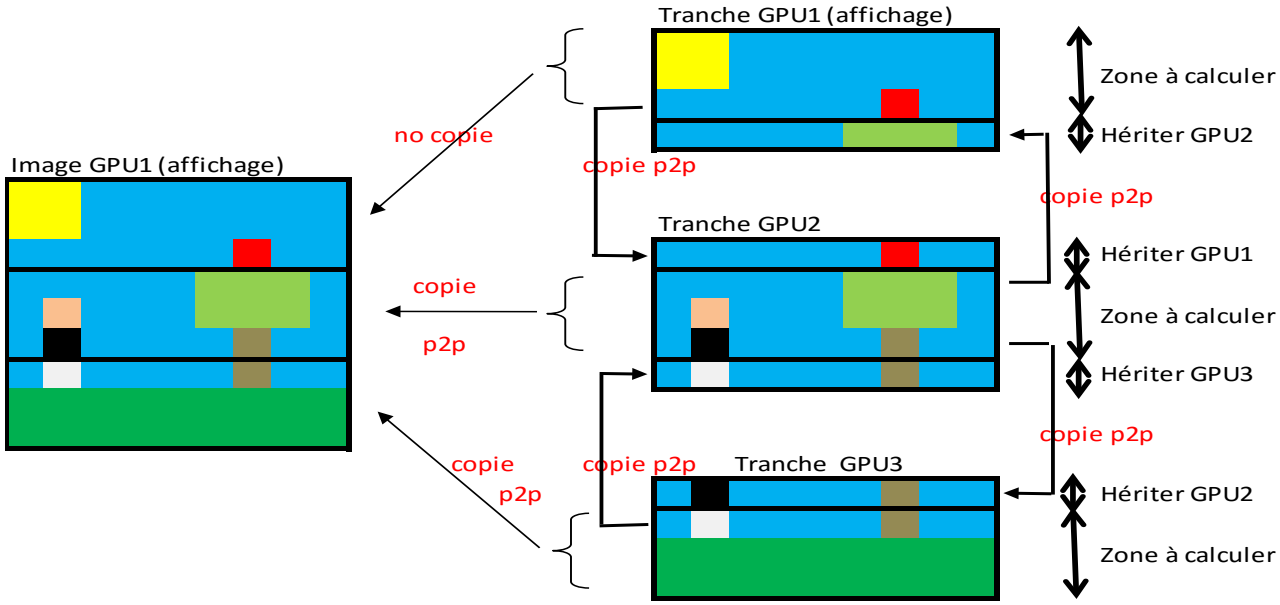
La couleur d'un pixel est **indépendante** de la couleur des pixels voisins.
Aucune synchronisation inter-GPU n'est donc nécessaire pour le calcul d'un pixel.
Chaque GPU travaille sur une portion indépendante de l'image (tranche)



Une étape finale de memory management MM est nécessaire à la fin pour regrouper les différentes tranches calculées sur les GPU auxiliaires sur le GPU maitre qui affiche l'image globale.

Classe ***Convolution***

La couleur d'un pixel est **dépendante** de la couleur des pixels voisins.
Une synchronisation inter-GPU sera donc nécessaire pour le calcul des pixels dont le noyau de convolution interagit avec des pixels d'un autre GPU ! On travaille ici toujours en tranche, mais les tranches sont artificiellement gonflées, comme l'illustre le schéma suivant, dans le cas d'un voisinage 1-distant.



Le principe se généralise facilement dans le cas des voisins n-distants. Comme pour la classe *Mandelbrot*, une étape finale de memory management MM est nécessaire à la fin pour regrouper les tranches calculées sur les GPU auxiliaires sur le GPU maitre qui affiche l'image globale.

Cousin : *HeatTransfert*, ne s'agit-il d'ailleurs pas d'une convolution aussi ?!

Observation

(O1) Notons que dans toutes ces classes, le kernel de calcul ne change pas, il s'agit à l'exacte du même que celui de la version monoGPU !! Seul du memory management MM est nécessaire pour allouer les ressources adéquates sur tous les GPU en phase zéro d'initialisation, et nécessaire en phase finale pour le regroupement.

(O2) Dans le cas des images, on découpe l'image en tranche horizontale (et non verticale !), quitte à gonfler les tranches de telle sorte qu'il y ait un recouvrement entre elles. Ainsi un GPU peut accéder au pixel du GPU voisins, mais sans sortir de sa propre GRAM. Dans cette famille « Convolution », une phase de synchronisation inter-GPU est nécessaire à la fin de chaque cycle, de telle sorte que chaque GPU pousse les bords de ces tranches sur le GPU voisin, de telle sorte que les résultats se propage de GPU à GPU. Si on ne pratiquait pas ici, dans le *HeatTransfert*, la chaleur ne pourra pas de propager au-delà de la bande de

laquelle est issue. Une nouvelle fois la difficulté du multi-GPU va se reporter sur un calcul subtile d'indice pour connaitre l'adresse de départ et d'arrivée de la zone à copier !

Syntaxe

Lisez dans Bilat_CudaPratcicalGuide.pdf le chapitre concerné au multi-GPU !
En particulier, il est important d'utiliser des threads cotés host, thread *OMP* ou *Boost* peu importe !

Montecarlo-MultiGPU

Objectif

Pour l'implémentation Cuda, répartissez le travail sur les *k* GPUs que vous avez à disposition !

Principe

Si au total vous devez tirer *n* fléchettes, et que vous avez *k* gpu, chaque GPU devra tirer seulement *k/6* fléchettes au lieu de *k* en version mono GPU. Le temps d'exécution devrait donc être *k* fois plus court !

Après une synchronisation, il faut ensuite juste consolider le résultat de tous les GPUs.

Challenges

(C1) Réussir à répartir le travail sur les *k GPU*

(C2) Les GPUs doivent travailler en même temps, en parallèle et non en séquentiel !

Implémentation

Version 1 ***OMP***

Utilisez OMP, chaque thread OMP s'occupe de spécifier un *device* via

cudaSetDevice(deviceId));

puis lance le kernel avec *n/k* fléchettes à lancer !

Version 2 ***Boost***

Idem que OMP, mais ici on utilise Boost comme library de thread.
Regarder à cet effet le projet *Tuto* mis à disposition !

Version 3 ***Stream Cuda***

Observation

Pour répartir des kernels sur plusieurs GPUs en parallèles, les *streams* ne servent à rien ! En effet, rappelons que les *streams* contrôlent la séquence d'exécution côté *device* essentiellement, pas coté *host* !

But des streams

Les *streams* permettent de lancer sur un même GPU des kernels en parallèles, ou du memory management en parallèle.

Rappel

Rappelons que côté *host*, l'appel d'un kernel est *assynchrone*, jusqu'à une barrière de synchronisation, explicite ou implicite. Parmi les barrières implicites, on trouve

- Les actions de memory management (de la même *stream* que le kernel)
- Le lancement de tout kernel (sur le même *device* ou un autre *device*)

Exemple

```
cudaSetDevice(0));
k0<<<dg,db,0,stream0>>>(…) ;// assynchrone

cudaSetDevice(1) ;
k1<<<dg,db,0,stream1>>>(…) ;// (#) barrière pour k0 !!
```

On s'attend à ce que *k1* débute « en même temps » que *k0*. Il n'en est rien. Il y a ici séquentialisation côté host! En effet (#) est une Barrière de synchronisation pour *k0***.** Ceci est vrai même si la *stream* dans laquelle on lance *k0* n'est pas la même que *k1*. Pour faire partir *k1* avant que *k0* ne soit terminé, il faut impérativement utiliser une *API* de thread comme *OMP* ou *Boost*.

Controller *La répartition multi-gpu*

Quel que soit la version contrôler la répartition multi-gpu avec les indicateurs suivants :

(I1) Controller le parallélisme avec

Device ::printCurrent() ;

juste avant l'appel du kernel

(I2) Controller avec l'outil console

nvidia-smi –l

ou pour spécifier la fréquence de rafraichissement

nvidia-smi --loop=1

Vérifier alors

- Les processus attachés au différent GPU
- Leur taux d'occupation
- Leur température
- …

(I3) Si vous avez *k* GPU, le temps total doit être *k* fois plus court si les GPU travaillent bien en parallèle ! Chronométrer et afficher le temps nécessaire !

HeatTransfert-MultiGPU

It's up to you !
Bon courage !
Il s'agit exactement du schéma du chapitre principe ci-dessus avec des voisins 1-distant

Convolution-MultiGPU

It's up to you !
Bon courage !
Ici les voisins soit 4 distants, avec notre noyau de convolutions de taille 9.

Speed-up MultiGPU

QUAND

Il est intéressant de vérifier depuis « quand » le multi-GPU devient intéressant !

Par exemple, dans le cas Mandelbrot, effecteur une animation mais avec une profondeur de convergence fixe, mesurer les fps, puis recommencer avec une profondeur de convergence plus grande. Plus la profondeur de convergence est grande, plus le multi-GPU risque de devenir intéressant, car les calculs deviennent vraiment longs.

Si les calculs prennent très peu de temps, le multi-GPU risque d'être défavorable, car les copies p2p prendront une part trop grande du temps total.

PCI-EXPRESS

Sur cuda2 le pci-express est de type :

gen3 x16 x16

alors qu'il n'est que

gen2x16x16

sur cuda1, voyez-vous des différences d'interprétations de vos résultats ?

P2P

Tout le hardware mis à disposition n'est pas p2p compatible. Utiliser la classe Device fournit pour afficher la matrice de p2p disponible. Cette même classe Device permet d'ailleurs d'activer toutes

les connexions p2p disponibles. Est-il intéressant d'utiliser que les GPU p2p compatibles, quitte à en pendre moins ?

PATTERN

Est-il intéressant de varier un peu le pattern multi-GPU est de n'utiliser le GPU d'affichage que pour l'affichage, et de transférer le calcul de la tranche1 à un autre GPU ? Quelle performance on obtiendrait si on employait un GPU pour le calcul et un GPU pour l'affichage ?

End