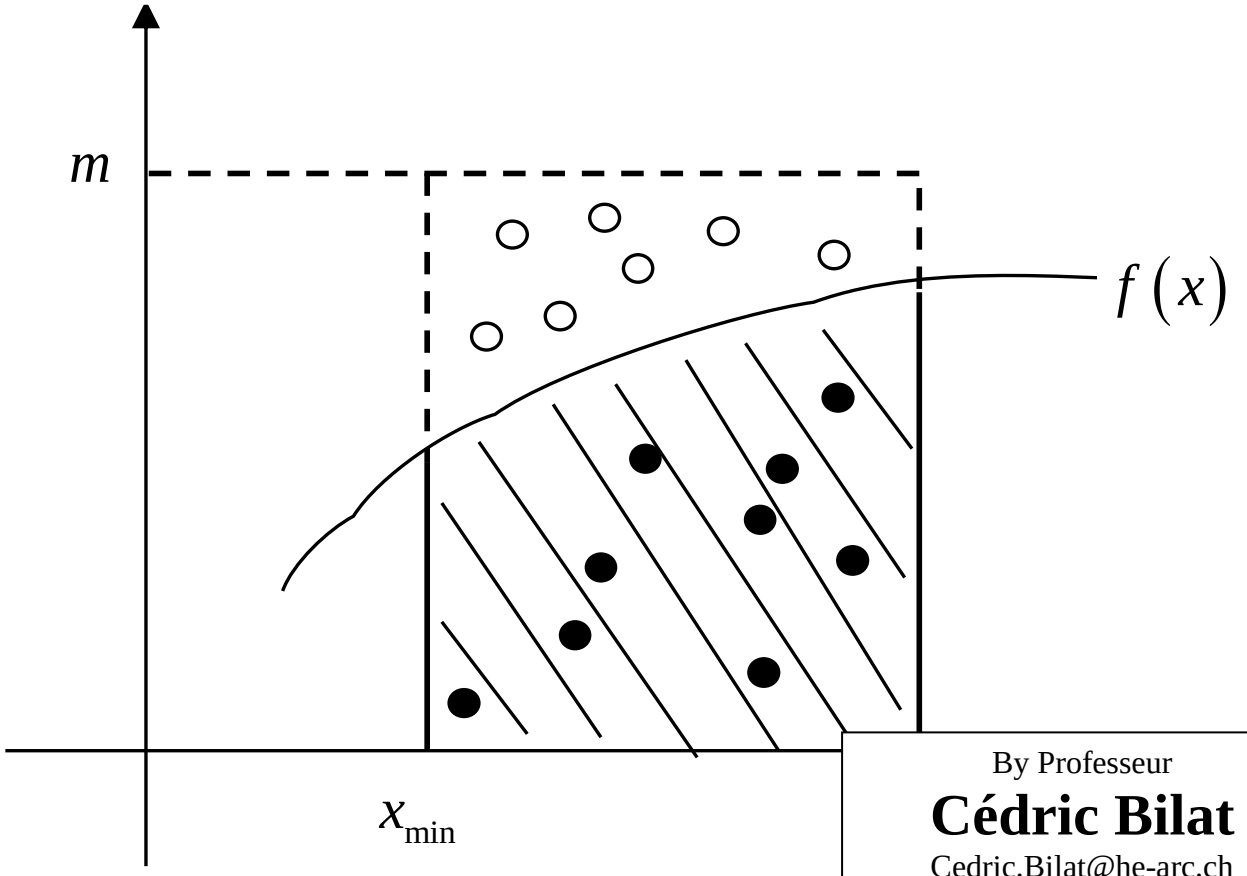
GPGPU



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

MonteCarlo

MULTI-GPU

Thread

Contexte

Réalisez d'abord le TP

Montecarlo MONO-GPU

L'idée est ici d'effectuer une implémentation MULTI_GPU. Si on 4 *devices*, ca doit aller 4 fois plus vite, car les GPU peuvent travailler en parallèle. Evidemment le nombre total de fléchettes reste globalement le même, mais avec 4 *device*, chaque *device* tira

nombre_Flechette_totale / nbDevice

Multi GPU Principe

Important

Le device « actif » est un état de thread.
On peut setter cet état avec

```
#include « Hardware.h »
. . .
Hardware::setDevice(id) ; // id in [0,…]
```

Principe

Coté *host* utilisez autant de thread que vous allez utiliser de GPU. On peut récupérer le nombre de GPU avec

```
int nbDevice=Hardware ::deviceCount() ;
```

Dans chaque thread (coté host) :

- Settez en « actif » en GPU différent.
- Effectuez le memory management (MM) nécessaire (pour le GPU actif du thread)
- Lancer votre kernel GPU (pour le GPU actif du thread)
- Effectuez le memory management (MM) nécessaire (pour le GPU actif du thread)

Finalement une étape séquentielle de **réduction** côté *host* pourra être nécessaire, pour consolider les résultats partiels de chacun des GPU en un résultat unique.

Observation

Pour passer un TP de mono GPU à multiGPU, on peut utiliser le même kernel qu'en mono GPU, seul « les données » ou les « quantités » vont changer entre les différents GPU. Le travail à réaliser sera uniquement coté *host* !

Dans Montecarlo, le changement est de type « quantité ». Si vous avez 4 GPUs, et que vous souhaitez tirer au total disons 1000 fléchettes, chaque GPU ne devra tirer que 2500 fléchettes en version *multi-gpu*, contre 1000 en version *mono-gpu*.

Indication/Contrainte

Pour la gestion des threads coté *host*, utilisez OMP !

Multi GPU Structure de code

Hypothèse

Supposons que votre version *monoGPU* ait donné naissance à la classe **Montecarlo** :

```
int nbFlechette=4000;
double piHat=-1;
Montecarlo  montecarlo=Montecarlo(nbFlechette,&piHat); // update piHat
montecarlo.run();
```

Principe

On crée une classe **MonteCarloMultiGPU** de type

MontecarloMultiGPU
+ MontecarloMultiGPU(Grid,n,ptrPiHat)
+ run():void
+ getNbDarTotalEffective():entier

// n = nombre de dar a tirer

qui utilisera la classe **Montecarlo (mono gpu)** dans sa méthode *run*.

Multi GPU Précision Math

Voir chapitre du même nom dans le TP *Montecarlo* (mono GPU int). Ici la même correction est nécessaire, mais pour la réduction inter-GPU.

Il faut utiliser le nombre de fléchettes effectivement tirés par chaque GPU, pour obtenir le nombre correct de fléchettes totales tirés par tous les GPU.

Exemple

Avec

$$|Flechette|^{ask}_{totale} = 100$$
$$|GPU| = 3$$

Il faut par GPU

$$|Flechette|_{ByGPU} = \left\lfloor \frac{100}{3} \right\rfloor = 33$$

Ainsi on va tirer au total :

$$|Flechette|^{Empirique}_{totale} = 33*3 = 99$$

ce qui est différent de 100. Il s'agit d'une première erreur, qui se cumule avec une deuxième erreur. Si on a par exemple

$$|thread|^{ByGpu}_{totale} = 10$$

chaque GPU va tirer $\left\lfloor \frac{33}{10} \right\rfloor = 3$ fléchettes Si on cumule l'erreur 1 et l'erreur 2, il y a une différence de 10 fléchettes tirés en moins :

$$3*10*3 = 90 ! = 100$$

En soit ce n'est pas grave, mais il faut en tenir compte dans la formule de l'estimation de PI ! Utilisez à cet effet :

getNbDarTotalEffective

de la classe

Montecarlo
MontecarloMultiGPU

Pour améliore la précision mathématique, out tout simplement avoir un code juste, il y a deux réduction à faire :

- Une sur le nombre de fléchette sous la courbe
- Une sur le nombre de fléchette effectivement tirées

Cuda code Version I

```
void MontecarloMultiGPU ::run()
      {
      int nbDevice=Hardware ::getDeviceCount() ;
      int nbDarGPU=nbDarTotal/nbDevice;

      entier nbDarUnderCurve =0 ;
      entier nbDarTotalEffective=0:

      #pragma omp parallel for reduction
      for (int idDevice=0; idDevice< nbDevice ; idDevice ++)
            {
             Hardware::setDevice(idDevice);// idDevice n'intervient plus ensuite

             double piHatGPU=-1 ; // inutile mais obligatoire
             Montecarlo montecarlo(nbDarGPU,&piHat)
             montecarlo.run() ;// sur le device courant !

             #pragma omp critical (blabla)
                  {
                  nbDarUnderCurve += montecarlo.getNbDarUnderCurve() ;
                  nbDarTotalEffective+= montecarlo.getNbDarTotalEffective();
                  }
            }

      // Finalisation mathématique coté host
      this->piHat= nbDarUnderCurve * . . .
}
```

La section critique ne coute pas cher, elle est gratuite, car il y a peu de GPU !

Piège host-parallélisme

Il faut impérativement charger les drivers de tous les gpus dans la méthode *main.cpp :*

Avant

```
Hardware::loadCudaDriver(deviceId);
//Hardware::loadCudaDriverAll();// Force driver to be load for all GPU
```

Apres

```
// Hardware::loadCudaDriver(deviceId);
Hardware::loadCudaDriverAll();// Force driver to be load for all GPU
```

Sinon le chargement des drivers est lazzy, et se fait lors de la première utilisation du GPU. Dans notre cas le chargement se ferait sous forme *lazzy ,juste in time* dans notre boucle :

```
#pragma omp parallel for
```

Or, comme le chargement des drivers des GPU ne peut être réalisé en parallèle, ce chargement lazzy est une barrière de synchronisation qui rend notre boucle parallèle, séquentielle, et ruine tous nos efforts. Les deux GPU sont utilisés, mais en séquentiel !

Squelettes de codes

Des squelettes de codes vous sont fournis dans le workspace.

1) Complétez les ***TODO*** (requiert de comprendre le code fournit et son organisation)
2) Faites passer les *uses*
3) Faites passer les *tests unitaires*

Concept

Les squelettes de codes vous sont fournis pour gagner du temps de saisie de code c++ standard. L'objectif dans ce cours est de mettre l'accent sur Cuda,

Il vous incombe néanmoins de comprendre les codes fournis, leur interaction et les structures sous-jacente.

Tests Unitaires

Les ***tests unitaires*** sont déjà codés.

Procédure

Dans le workspace, on peut spécifier deux types d'exécution :

- ***Use***
- ***Test***

dans la méthode ***main*** à la racine du projet.

Etape 1

Passer le workspace en mode *test* dans ***main.cpp***, à la racine du projet.

Etape 2

Mettez en commentaire les tests qui ne vous intéressent pas dans ***mainTest.cpp***
Inutile de tous les lancer à chaque fois !
Idem pour la méthode ***mainUse.cpp***.

Etape 3

Choisissez une rendu *console* ou *html* dans ***mainTest.cpp***
(html conseillé)

Nombre de Fléchettes Long

Travailler en long, non pas pour avoir une meilleur précision de PI, mais pour mettre le GPU a genoux, et par exemple avoir le temps de monitorer ! N'exagérez pas non plus et laisser des ressources aux autres utilisateurs ! Tirer plus que INT_MAX fléchettes, mais pas trop non plus quand même.

Exemple

Essayer par exemple avec

```
long nbFlechettesTotal= MAX_INT ;
```

puis augmenter progressivement :

```
long nbFlechettesTotal=((long) MAX_INT) *8;
```

puis

```
long nbFlechettesTotal=((long) MAX_INT) *900;
```

etc …

Conseil

Voir le paragraphe *Limite/Danger* de *MonteCarloMonoGpuLong.pdf*

TimeOut

Voir le paragraphe *Limite/Danger* de *MonteCarloMonoGpuLong.pdf*

Multi GPU Valider

Voici différentes stratégies pour Controller que votre code s'exécute bien en parallèle sur plusieurs devices.

Mieux que rien

Controller le changement de device avec

```
Hardware ::printCurrent() ;
```

juste avant l'appel du kernel

Monitoring

Controller avec l'outil console

```
nvidia-smi  --loop=1
```

Vérifier alors :

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

Notez que vu la performance des GPU récents, le temps d'exécution est très, voir trop court pour pouvoir le monitorer avec l'outil proposé ci-dessus ! Utilisez donc votre implémentation mono gpu en long. Dans ce cas, on peut arbitrairement ralentir le kernel, en mettant beaucoup de fléchettes ! N'exagérez pas non plus !

Intéressant

Observez la différence avec *nvidia-smi –loop=1*, entre une version :

- Multi gpu séquentielle
- Multi gpu parallèle

Rappel : En OMP, on passe facilement de l'une à l'autre !

Performance Observation

En Int

Théorie

Avec 3 GPU, cela devrait aller quasiment 3 fois plus vite. En effet, il n'y a pas de perte de temps à copier des résultats cotés host et une grosse consolidation à faire des résultats de chacun des GPU. Il s'agit juste de transférer le nombre de fléchettes sous la courbe de chacun des gpus.

En pratique

Les performances ne sont pas mieux, voir même moins bonne, aie aie aie !

Explications

La création des threads cotés host coute trop cher par rapport à l'exécution du kernel.

En Long :

Cette fois les performances obtenues sont à la hauteur des attentes.
La création des threads coté host ne sont pas plus rapide, mais ce temps de création devient négligeable par rapport à la durée d'exécution des kernels.

Amélioration

Pour avoir une version multiGPU efficace dans tous les cas, il vaut mieux ne pas utiliser de thread coté host, mais des ***stream*** !

Pratiquez à cet effet le ***TP Montecarlo_MultiGPU_stream***

End