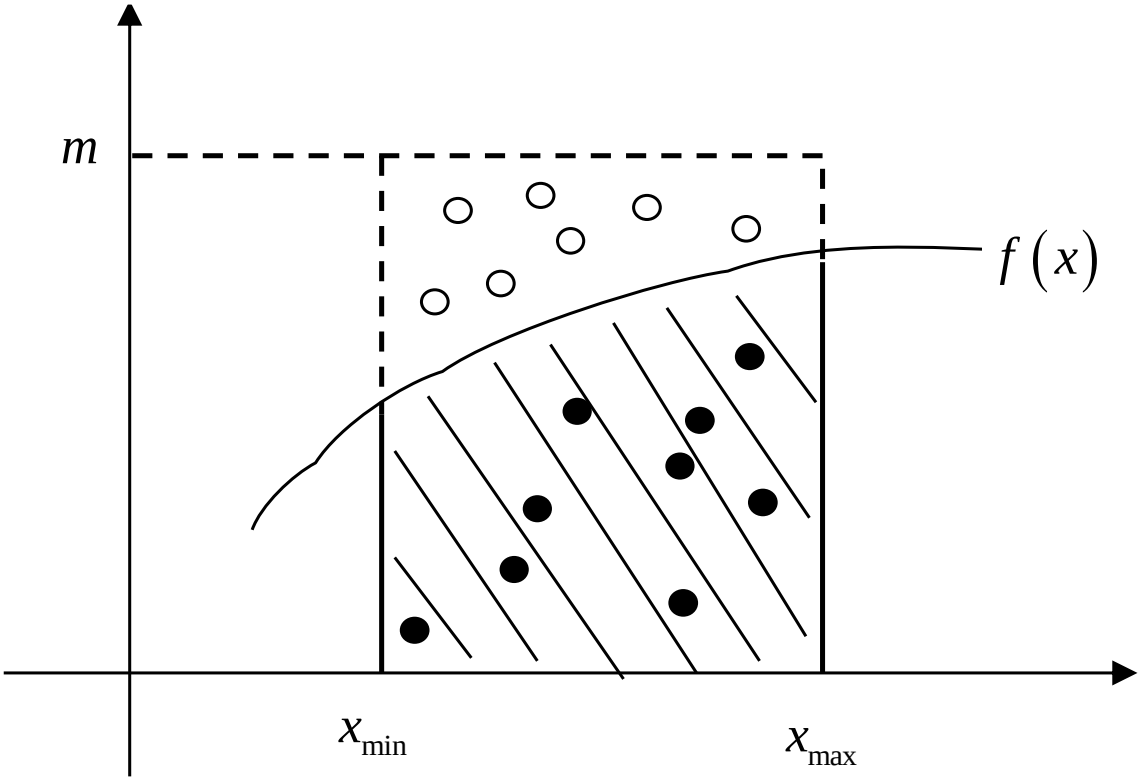
GPGPU



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

MonteCarlo

MONO-GPU

Long

Contexte

En Int

Dans le TP de base, on a tiré le nombre de fléchettes en *int*.

Mathématiquement

Le type *int* est largement suffisant pour obtenir une approximation de PI. D'ailleurs, comme la qualité de la technique de *Montecarlo* est borné par une asymptote horizontale, après un certain seuil, tirer 2x plus de fléchettes n'améliore quasi plus la qualité, et le jeu n'en vaut plus la chandelle.

Performance

Ca va tellement vite en *int*, qu'on par exemple pas le temps de monitorer le GPU, avec

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

ni de voir l'influence de diverses optimisation. Le temps écoulé est si court qu'ont atteint presque la granularité minimum dont dispose le système de chronométrage. Les mesures du chrono commencent à être entachées de bruit.

En long

Le but n'est pas d'obtenir plus de décimales de PI, mais de pouvoir mettre arbitrairement à genoux le GPU, et ainsi monitorer et quantifier l'impact de différentes optimisations. Il s'agit donc d'un exercice de style, et non une nécessité

Cuda Réduction Long

Problème

La partie de réduction inter-block souffre d'un problème de concurrence, qui été résolu avec un

```
atomicAdd
```

qui est la technique la plus efficace. Malheureusement, selon la version de cuda ou du hardware, cette méthode n'est pas disponible pour tous les types simples. Elle est bien présente pour les *int*, mais pas toujours pour les *long* !

Challenge : On va faire l'hypothèse que *atomicAdd* n'existe pas !

Solution

Utiliser votre classe de ***Reduction*** ; qui elle est générique.

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 ***mainUse.cpp***.

Etape 3

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

Int vs Long entier

entier.h

On joue sur le ***typedef*** entier qui vaut soit un ***int***, soit un ***long.***

On spécifie ça dans le fichier ***entier.h*** à la racine du folder montecarlo

```
/*----------------------*\
 |* public               *|
 \*----------------------*/

// Choose one of the two (either/either):
#define DAR_INT
//#define DAR_LONG

/*----------------------*\
 |* private               *|
 \*----------------------*/

#ifdef DAR_INT
#define entier int
#endif

#ifdef DAR_LONG
#define entier long
#endif
```

Code¨

On remplace ensuite partout *int* par *entier*

Device side

Selon si on travaille en *int* ou en *long*, on utilise :

Reduction
ReductionGeneric

Le reste du code est commun et basé sur le typedef entier

```
__global__ void montecarlo(curandState* tabGeneratorGM ,
                           entier nbDarByThread ,
                           entier* ptrNbDarAboveGM ,
                           float h)
    {
    // TODO Montecarlo
    //      declarer tabSM
    //      reductionIntraThread

      #ifdef DAR_INT
             //TODO Montecarlo ReductionAdd
      #endif

      #ifdef DAR_LONG
             //TODO Montecarlo Reduction
      #endif
    }
```

Un seul des deux blocs rouges sera compiler, il n'y a donc aucune perte de performance !

Limite Danger

Technique

Il est maladroit de définir un nombre de fléchettes sous la forme

```
int n=100000 ;          // Dangerous, débordement du type ?
long n=10000000 ;       // Dangerous, débordement du type ?
```

car on ne sait jamais si on déborde du type ou pas. Le mieux est de prendre la valeur *max*, puis de la diviser par un multiple de 10 pour la rendre plus petite.

Exemple :

```
#include « limit.h »

int n=MAX_INT/10;
```

On peut aussi utiliser la classe ***LimitTools.h*** puis faire du code complétion dessus.

```
#include « Limits.h »

Limits::rappelTypeSize(); // affiche le max des types
Limits::MAX_INT ;
Limits::MAX_LONG ;
Limits::MAX_LONG ;
Limits::MAX_SHRT ;
```

Windows

Sous *Windows*, il faut faire gaffe !

```
#include « limit.h »

#ifdef _WIN32
    long n=LONG_MAX; // sous windows INT=LONG
#else
    long n =((long)INT_MAX)*80; // sous linux on peut prendre plus grand
#endif
```

Cuda GPU Timeout

Rappels

Sous linux, le temps d'exécution max d'un *kernel* est de 2s si un écran est branché au *device*, et arbitrairement grand si aucun écran n'est connecté au *device*.

Trucs

Dans une configuration standard c'est souvent le gpu de deviceId=0 qui a un écran et donc un timeout. Pour voir si *device* à un timeout, deux méthodes :

- Utilisez la classe ***Devices*** et la méthode *printALL*
- Utilisez l'utilitaire

```
nvidia-smi
```

ou

```
nvidia-smi -q
```

Note *:*

nvidia-smi n'est disponible seulement sur GPU haut de gamme, de type *quadro*.

Exemple *nvidia-smi*

Si il y a un **ON**, il y a un écran et un timeout.

```
+-----------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|=============================+======================+======================|
|   0  Quadro M6000        Off  | 0000:03:00.0     On |                  Off |
| 26%   39C    P8    16W / 250W |     80MiB / 12205MiB |      0%      Default |
+-----------------------------+----------------------+----------------------+
|   1  Quadro M6000        Off  | 0000:04:00.0     Off |                  Off |
| 26%   39C    P8    16W / 250W |      1MiB / 12206MiB |      0%      Default |
+-----------------------------+----------------------+----------------------+
|   2  Quadro M6000        Off  | 0000:83:00.0     Off |                  Off |
| 26%   41C    P8    16W / 250W |      1MiB / 12206MiB |      0%      Default |
+-----------------------------+----------------------+----------------------+
|   3  Quadro M6000        Off  | 0000:84:00.0     Off |                  Off |
| 26%   39C    P8    15W / 250W |      1MiB / 12206MiB |      0%      Default |
+-----------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                             Usage      |
|=============================================================================|
|    0      1462    G   /usr/lib/xorg/Xorg                            79MiB |
+-----------------------------------------------------------------------------+
bilat@cuda2:~$
```

device 0 est-on, donc à un timeout

Cuda Optimisation

Deux optimisations parmi tant d'autres :

Unroll loop

Essayer de voir l'impact sur les performances de la variation ci-dessous :

Unroll loop

```
                                        n=n/4 ;
for (int i=1 ; i<=n ; i++)              for (int i=1 ; i<=n ; i++)
            {                                   {
            work( );                            work( );
            }                                   work( );
                                                work( );
                                                work( );
                                                }
```

Générateur

La création des générateurs coute très cher, mais on peut les utiliser dans plusieurs kernels différents. Par contre une fois créé ces générateurs sont très rapide. Vu l'association

Thread < -- > Générateur

Pour une fois, n'utilisez pas trop de thread, pour minimiser le temps nécessaire à la création des générateurs. Une fois de plus, il faut trouver le bon compromis.

End