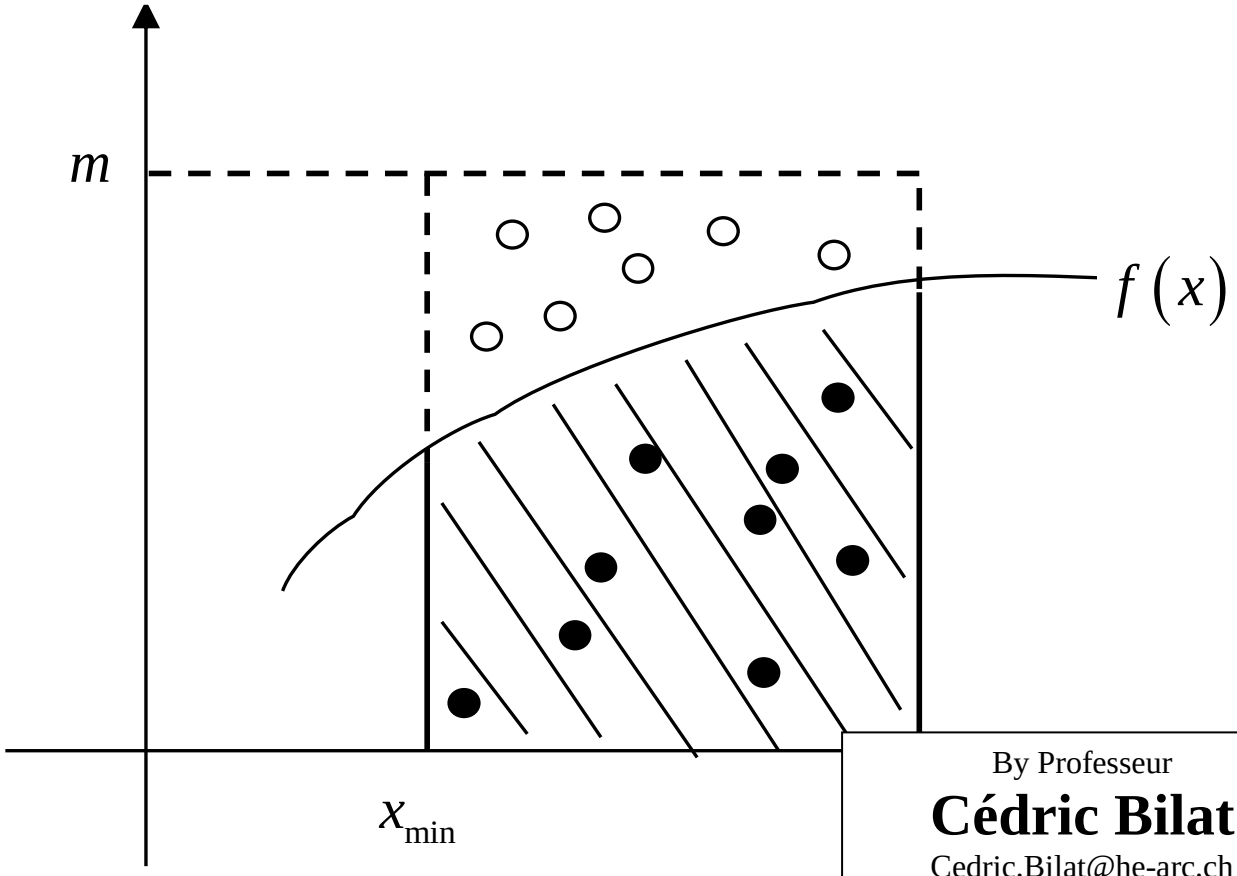
GPGPU



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

MonteCarlo

MULTI-GPU

Stream

Contexte

Prérequis

Réalisez d'abord le TP

Montecarlo MultiGPU_thread

notamment le dernier paragraphe « Performance Observation »

But

Le but est de corriger cette carence en performance lorsque le temps de l'execution des kernels est petit

Tactique

Comme le temps de création des threads coté host coute trop cher, on va supprimer ces threads ! Notre salut viendra de l'utilisation d'un outil Cuda : les streams !

Default Stream

Observation

Depuis le début du cours, on emploie le concept de *stream*, mais sans le savoir. On emploie à tout instant la *stream* par défaut.

Model Cuda naïf

Depuis le début du cours, on a dit :

(D1) L'appel d'un kernel est asynchrone

```
Chrono chrono;
k1<<<dg.db>>>( ...) ;  // asynchrone
cout<<chrono.stop();    // minuscule
```

(D2) Une action de memory management (MM) de la GM (memcpy)est une barrière de synchronisation implicite

```
k1<<<dg.db>>>( ...) ; // asynchrone
GM::memcpyDtoH(...);// barrière de synchronisation implicite
```

GM ::memcpyDtoH attend que le kernel ait fini

(D3) L'appel d'un second kernel attend que le premier soit fini.

```
k1<<<dg.db>>>( ...) ;  // asynchrone
k2<<<dg.db>>>( ...) ;  // attend que k1 soit fini
```

Observations

Tout ceci est vrai, car tout ceci se passe dans la même *stream*, la *stream* par défaut. En pratique on peut créer plusieurs *streams*, et seul les éléments processer dans une même *stream*, sont affectés par les synchronisation ci-dessus. Deux éléments de deux *streams* différents ne sont jamais synchronisées.

Conséquences

(C1) On peut donc lancer 2 kernels en parallèle, il s'agit juste de les lancer sur des *streams* différents.

(C2) On peut effectuer des copies en parallèle, il suffit juste de le faire sur des *streams* différents.

(C3) On peut lancer deux kernels en parallèle sur des GPUs différents, il suffit juste de le faire sur des *streams* différents et sur des *deviceID* différent.

Rappel

deviceId est un état que l'on peut changer avec

```
Hardware ::setDevice(deviceID) ;
```

Stream Syntaxe

Création/destruction

Les streams se créent et se détruisent :

```
#include "Stream.h"

cudaStream_t stream;
Stream::streamCreate(&stream);
Stream::streamDestroy(stream);
```

Appel d'un kernel sur une stream

On utilise les <<< >>> pour spécifier la stream auquel sera associé le kernel :

```
#include "Stream.h"

cudaStream_t stream;
Stream::streamCreate(&stream);
k<<<dg,db,sizeSM,stream>>>(...); // au pire on met sizeSM=0
```

Par défaut on a

```
k<<<dg,db>>>(...);
// idem que
k<<<dg,db,0,0>>>(...);
```

MemoryManagement sur une stream

```
#include « GM.h »

GM::memcpyAsyncDToH(ptrDestHost, ptrGM, sizeOctet, stream); // async
```

Stream Example

But

Exécuter deux kernels sur 2 GPU différents, en parallèle !

```
#include "Stream.h"
#include "GM.h"

const int DEVICE_ID_ORIGINAL=Hardware::getDeviceId(); // bakup du deviceId original

cudaStream_t stream0;
cudaStream_t stream1;

// Creation des stream
    {
    Hardware::setDevice(0); // obligatoire
    Stream::streamCreate(&stream0);

    Hardware::setDevice(1); // obligatoire
    Stream::streamCreate(&stream1);
    }

// kernel
    {
     Hardware::setDevice(0);        // obligatoire
     k<<<dg,db,0, stream0>>>(...); // non bloquant

     Hardware::setDevice(1);        // obligatoire
     k<<<dg,db,0, stream1>>>(...); // non bloquant
    }
```

```
// DtoH
    {
    //Hardware::setDevice(0);                       // facultatif
    GM::memcpyAsyncDToH(..., ..., ..., stream0);    // non bloquant

    //Hardware::setDevice(1);                       // facultatif
    GM::memcpyAsyncDToH(..., ..., ..., stream1);    // non bloquant
    }

work(...);      // avec les résultats recupérés coté host

// synchronisation
    {
    //Hardware::setDevice(0);                       // facultatif
    Stream::streamSynchronize(stream0);             // attend fin actions steram0

    //Hardware::setDevice(1);                       // facultatif
    Stream::streamSynchronize(stream1);             // attend fin actions steram1
    }

// destroy stream
    {
    //Hardware::setDevice(0);                       // facultatif
    Stream::destroy(stream0);

    //Hardware::setDevice(1);                       // facultatif
    Stream::streamDestroy stream1);
    }

Hardware::setDevice(DEVICE_ID_ORIGINAL);// restauration du deviceId original
```

Obligatoire/Facultatif

```
Hardware::setDevice(...);
```

Obligatoire

(O1) Avant la création de la stream

(O2) Avant l'appel d'un kernel

Facultative

(F1) Avant GM::memcpyAsyncDToH(..., ..., ..., stream);

(F2) Avant Destruction de la stream

(F3) Avant La synchronisation de chacune des streans

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)

Monitoring

Vérifier que les GPU sont bien utiliser en parallèle, avec dans une console

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

Faites le plutot avec la version long, pour avoir le temps de monitorer !

Performance Observation

Cette fois-ci, en int comm en long, avec des kernels très rapide, ou des kernels plus long, les performances sont à la hauteur de nos attentes :

Avec 3 GPU, ca va 3X plus vite (dans ce TP)

End