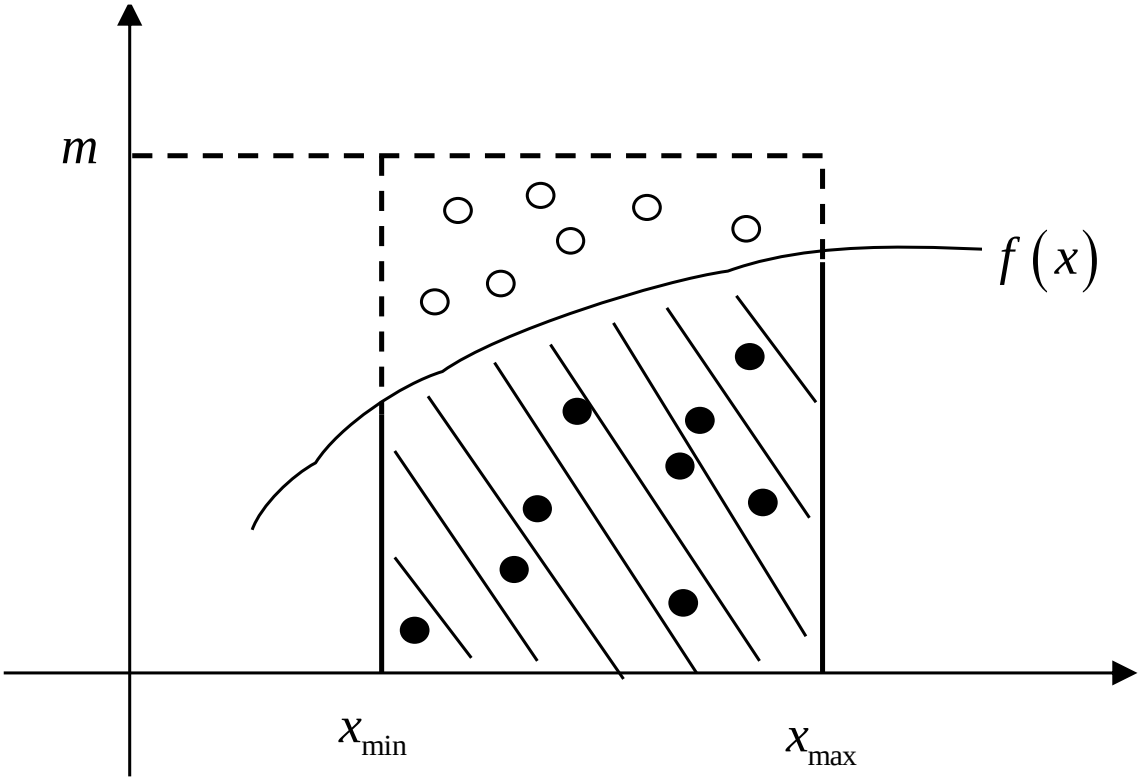
GPGPU



Cédric Bilat

MonteCarlo

MONO-GPU

Int

Prérequis

Lecture du pdf Montecarlo Algo

Curand Concept

Il existe une api pour tirer des nombres aléatoires en *Cuda* :

Curand

Elle est très complète, et complexe ! Il est laissé le soin au lecteur de l'étudier en détails. Voici néanmoins une introduction suffisante pour la réalisation du TP.

Introduction curand

Ca se passe en deux étapes :

Etape 1 : *Générateurs*

On lance un kernel pour créer des générateurs de nombres aléatoires.
Du memory management (MM) sera nécessaire pour créer de la place en global memory (GM) pour ces générateurs.

Attention *Qualité de l'aléatoire*

Chaque générateur doit générer une séquence de nombre différent. On associera à chaque thread un générateur. Il faut impérativement que chaque thread génère une série de nombre différents, que les threads soit dans le même block, ou non, dans le même GPU ou non !

Etape 2 Utilisation des générateurs

On utilise les générateurs de l'étape précédente dans un kernel métier
On passe à notre kernel métier le pointeur en GM sur le tableau de générateur crée à l'étape 1 :

prtGenerateurGM

Notons que les éléments en GM sont utilisables dans plusieurs kernels différents
Leur durée de vie est déterminée par la paire

cudaMalloc
cudaFree

Curand En pratique

Coté device

Voici le kernel pour créer les générateurs.
Le type d'un générateur est « curandState»

```
#include <curand_kernel.h>
#include <limits.h>
#include <Indice1D.h>

// Each thread gets same seed, a different sequence number
// no offset
// host side : Device::getDeviceId();
__global__
void createGenerator(curandState* tabGeneratorGM,int deviceId)
      {
      // Customisation du generator:
      // Proposition, au lecteur de faire mieux !
      // Contrainte : Doit etre différent d'un GPU à l'autre
      // Contrainte : Doit etre différent d'un thread à l'autre

      const int TID = Indice1D::tid();
      int deltaSeed=deviceId* INT_MAX/10000;
      int deltaSequence=deviceId *100;
      int deltaOffset=deviceId *100;
      int seed=1234+deltaSeed;
      int sequenceNumber= TID +deltaSequence;
      int offset=deltaOffset;

      curand_init(seed,sequenceNumber,offset,& tabGeneratorGM [TID]);
      }
```

Voici un exemple de kernel d'utilisation des générateurs

```
#include <curand_kernel.h>

__global__
void useGenerator(curandState* tabGeneratorGM, long n,…)
      {
      ...
      example(tabGeneratorGM,n, …) ;
      ...
      }
```

```
__device__
void example(curandState* tabGeneratorGM, long n,…)
    {
    const int TID = Indice1D::tid();

    // Global Memory -> Register (optimization)
    curandState generatorThread= tabGeneratorGM [TID];

    float xAlea;
    float yAlea;
    for (long i = 1; i <= n; i++)
       {
       xAlea01 = curand_uniform(&generatorThread); // in [0,1[
       yAlea01 = curand_uniform(&generatorThread); // in [0,1[
       …
       work(xAlea01,yAlea01);
       …
       }

    //Register -> Global Memory
    //Necessaire si on veut utiliser notre generator
    // - dans d'autre kernel
    // - avec d'autres nombres aleatoires !
    tabGeneratorGM [TID] = generatorThread;
    }
```

Coté Host

Effectuer du MM (memory management) avant de lancer le kernel de création des générateurs !

```
#include « GM.h »
. . .
int nbThread = grid.threadCounts();
size_t sizeOctetGeneratorGM= nbThread*sizeof(curandState);// en octets
curandState* tabGeneratorGM =NULL;

GM::malloc(&tabGeneratorGM, sizeOctetGeneratorGM);
```

Où?

Dans le constructeur de Montercarlo pour le *malloc*

Dans le destructeur de Montercarlo pour le *free*

Cuda Implémentation

La réduction

La réduction est toujours divisée en 3 étapes qui se jouent coté device :

Etape 1 : reductionIntraThread (rempli la sm)

Etape 2 : reductionIntraBlock

Etape 3 : reductionInterBlock

} **ReductionAdd**

Structurer votre code comme dans les TP précédents, sous la forme :

```
__global__ void montecarlo(…)
            {
            __shared__ extern int tabSM[] ;

            reductionIntraThread(tabSM, ….) ;
            . . .
            ReductionAdd::reduce(tabSM, …) ;
            }
```

Lien avec le paragraphe currand ci-dessus

La méthode **example** est l'équivalent de la méthode **reductionIntraThread**

```
__device__
void example(curandState* tabGeneratorGM, long n,…)
      {
      // TODO ReductionIntraThread

      // La réduction intraThread consiste à tirer des
      // fléchettes dans une boucle, à compter celle qui sont
      // au-dessous de la courbe. On utilise un registre pour
      // effectuer ce comptage, puis on dépose le compteur
      // dans « sa » case en SM.
      }
```

Renommer la méthode *example* :

example → reductionIntraThread

Updater la signature du prototype selon les besoins du TP, notamment en passant *tabSM :*

```
__device__
void reductionIntraThread(int tabSM, int nbDarsByThread,
                                curandState* tabGeneratorGM)
      {
      . . .
      int sumThread=0 ;
      for(int i=1 ; i<= nbDarsByThread; i++)
            {
            sumThread +=   ;// incrémenter par le thread courant
            }

      tabSM[ … ] = sumThread; // 1x, ecrasement!
      }
```

01

Les nombres aléatoires tirés par *curand* sont entre $[0,1[$. On peut les transformer en des nombres compris dans $[0, N[$

```
xAlea01 = curand_uniform(&localGenerator); // in [0,1[
yAlea0N = curand_uniform(&localGenerator)*N; // [0,1[ -> [0,N[
```

Cuda Type Entier

Performance et astuce

Le *device* ne doit pas calculer une estimation de PI, mais juste le nombre *nb* de fléchette sous la courbe! Coté *device* on utilise donc des ***int*** uniquement ! La fin du calcul peut se faire avantageusement cotée host ! Le device ne va travailler qu'en nombre

ENTIER

et compter uniquement le nombre de fléchettes obtenus sous la courbe !

Raison 1 :

On calcul *nb* côté device, puis coté host on finalise avec la formule :

$$\int_{x_{\min}}^{x_{\max}} f(x)dx \cong \frac{n_b}{n} aire(cible)$$
$$\cong \frac{n_b}{n}(x_{\max} - x_{\min})H$$

Il suffit de finaliser une fois ! Chaque thread n'a pas besoin de le faire.

Raison 2

Cette approche sera très utile pour l'approche multiGPU.

Raison 3

Un gpu calcule plus vite en entier qu'en floatant.

Type Entier

Le type ***entier*** qui peut cacher

- des int
- des long
-

Il est défini dans le fichier *entier.h* à la racine des du folder *MonteCarlo*, sous la forme d'un *typedef*.

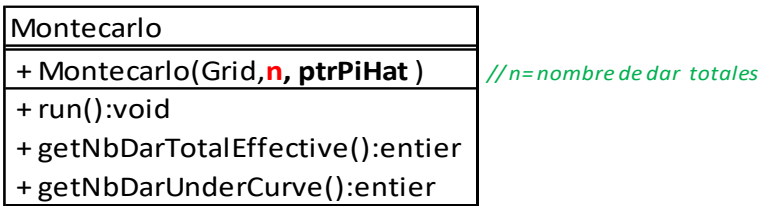
Dans ce TP on s'échauffe en *int* car c'est plus simple. Dans le TP suivant on passera en long.

Cuda ReductionAdd

Voir le TP séparé traitant de cette problématique !

Modélisation

Diagramme de classe



Le user spécifiera

n=le nombre de fléchettes totales

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)

Cuda Check

Check GM

N'oubliez pas la création et surtout l'**initialisation à zéro** de la variable contenant le résultat final :

```
#include « GM.h »

int* ptrResultGM =NULL;

GM::mallocInt0(&ptrResultGM) ;
```

Check SM

Faut-il initialiser les *tabSM* à zéros lors d'une réduction additive ? Oui et non, tout dépend de votre technique de « peuplement ». Lors de la réduction *intraThread*, dont le but est le peuplement des *tabSM*, utiliser une variable locale dans les registres pour :

- Optimiser les performances de votre code
- Eviter l'initialisation de tabSM

Cette variable va accumuler la réduction du thread auquel elle est rattachée, et une fois sa valeur finale obtenue, il faut la déverser une seule fois, par écrasement dans la « bonne » case du tableau en SM.

Chaque thread à une case en SM, « sa » case, elle est identifiée par son TID. Mais lequel : global ou local ? A vous de jouer !

Syncthread

N'oubliez pas les syncthreads, mais soyez minimaliste !

```
__syncthreads() ; // barrière pour tous les threads d'un même block
```

Math Précision

L'utilisateur spécifie le nombre de fléchettes à tirer.
Selon le nombre de thread dans la grille, le nombre de fléchettes total demandées ne pourra pas être honoré exactement.

La classe *Montecarlo* va donc ressortir avec un getter le nombre de fléchettes effectivement tirées ! Cette valeur sera utilisée pour le calcul de l'estimation de PI

Exemple

Avec

$$|Flechette|^{ask}_{totale} = 1009$$
$$|thread|_{totale} = 100$$

On obtient

$$|Flechette|_{thread} = \left\lfloor \frac{1009}{100} \right\rfloor = 10$$

Le nombre de fléchettes effectivement tirées sera

$$|Flechette|^{Empirique}_{totale} = 10 * 100 = 1000$$

On tire donc 9 fléchettes de moins, ce qui en soit n'est pas grave, mais il faut dès lors appliquer la formule avec le nombre de fléchettes effectivement tirés, ie 1000 au lieu de 1009 dans la formule de l'estimation de PI :

$$\begin{aligned} \int_{x_{\min}}^{x_{\max}} f(x)dx &\cong \frac{n_b}{n} aire(cible) \\ &\cong \frac{n_b}{n}(x_{\max} - x_{\min})m \\ &\cong \frac{|Flechette|_{sous\ la\ courbe}}{|Flechette|^{Empirique}_{totale}}(x_{\max} - x_{\min})m \end{aligned}$$

Cuda Variation

Deux variations parmi tant d'autres :

(A1) SM Partielle

Le tableau en SM sera de dimension 1. Il n'y aura ainsi pas besoin de *réduction intraBlock.* Tous les threads du même block écrivent leur résultat directement dans cette unique case. Le problème de concurrence se résout avec un

```
atomicAdd(&tabSM[0],sumTidLocal) ;
```

ou de manière équivalente

```
atomicAdd(tabSM, sumTidLocal) ;
```

Que peut-on dire des performances ?

(A2) Sans SM

En utilisant pas de SM, mais uniquement la GM. . Le problème de concurrence sera contourné avec un `atomicADD`.

Que peut-on dire des performances ?

End