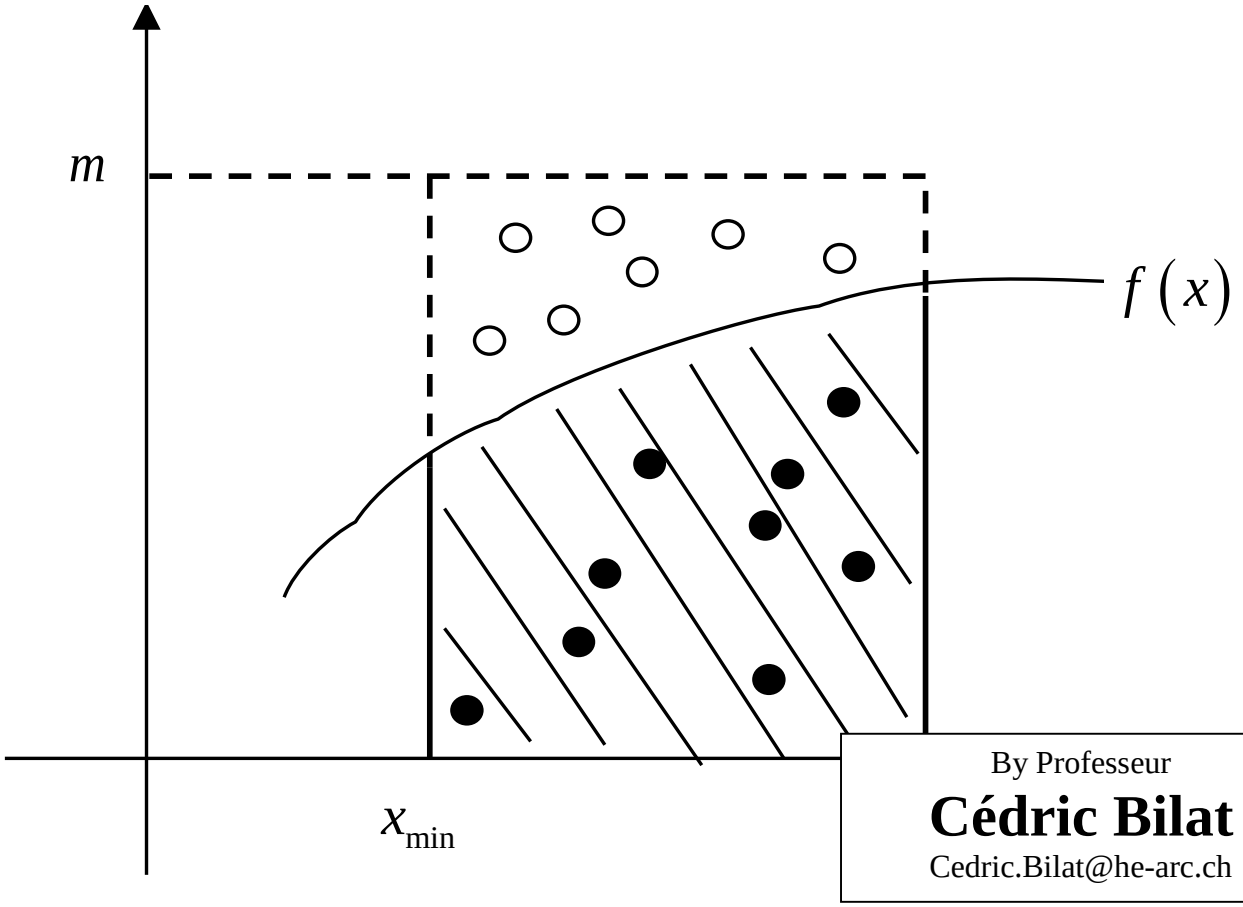
Problèmes Parallelisation



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

MonteCarlo

Intégration numérique

Contexte

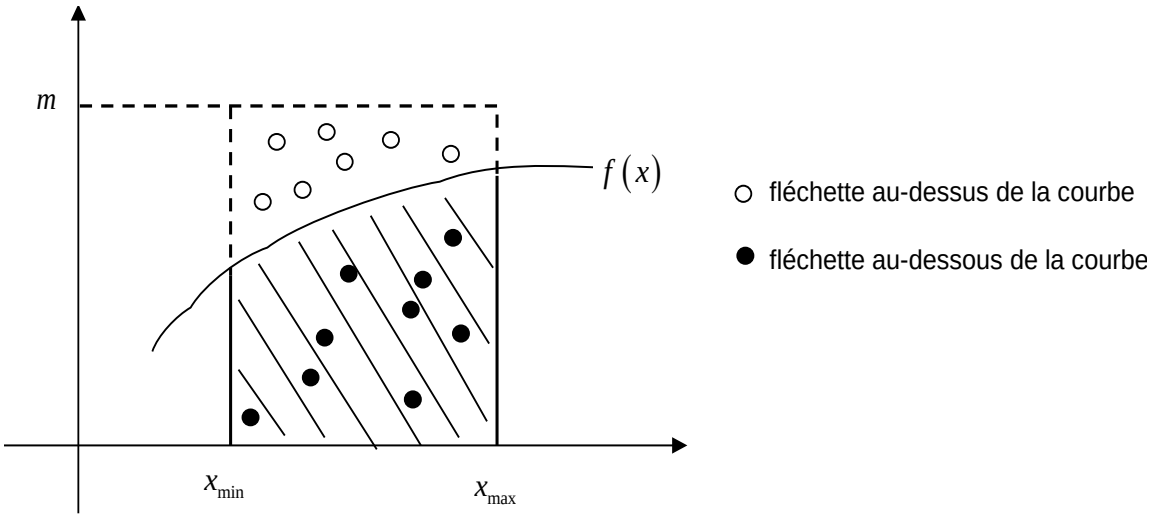
Il est possible de calculer une intégrale

$$\int_{xMin}^{xMax} f(x)dx = ?$$

à l'aide de nombres aléatoires. Le principe est le suivant:

Principe

On tire aléatoirement n fléchettes sur la cible $[x_{\min}, x_{\max}]x[0, m]$.



où

m est une borne supérieur quelconque de f sur $[x_{\min}, x_{\max}]$

Posons

$n_b = \#$ fléchettes sous la courbe $y = f(x)$

$n_h = \#$ fléchettes au-dessus de la courbe $y = f(x)$

$n = n_h + n_b = \#$ fléchettes totales lancées aléatoirement sur la cible.

Constatations

Soit P la probabilité qu'une fléchette arrive sous la courbe, si on la tire aléatoirement sur la cible. Dans cette expérience aléatoire, la fléchette est sur d'arriver sur la cible, mais pas forcément sous la courbe.

- Cette probabilité peut clairement être approximée par un rapport d'aire :

$$P \cong \frac{Aire(hachurée)}{Aire(cible)} \qquad (*)$$

Exemple : $m = 1$
$y = f(x) = 0.3$ (fonction constante)
$p = 30\%$

- Si n est grand, cette probabilité peut clairement être approximée par

$$P \cong \frac{n_b}{n} \qquad (**)$$

Finalement

Par (*) et (**) on a

$$\boxed{Aire(hachurée) \cong \frac{n_b}{n} Aire(cible)} \quad (\#)$$

où

$$\begin{cases} n = \#\text{couple } (x', y') \text{ aléatoire tirés dans } [x_{min}, x_{max}]x[0, m] \\ n_b = \#y' \text{ sous la courbe } f \\ Aire(cible) = (x_{max} - x_{min}) * m \end{cases}$$

Notation

Posons

$$\boxed{MonteCarlo^{m}_{x \in [x_{min}, x_{max}]}(f, n) := \frac{n_b}{n}(x_{max} - x_{min}) * m}$$

l' approximation par MonteCarlo de l'intégrale de $f : \mathbb{R} \longrightarrow \mathbb{R}^+$ sur $[x_{min}, x_{max}]$ avec

$$\begin{cases} n = \#\text{couple } (x', y') \text{ aléatoire tirés dans } [x_{min}, x_{max}]x[0, m] \\ n_b = \#y' \text{ sous la courbe } f \end{cases}$$

Attention

La formule

$$\int_{x_{\min}}^{x_{\max}} f(x)\,dx = \lim_{n \longrightarrow \infty} \frac{n_b}{n} Aire(cible)$$

$$= \lim_{n \longrightarrow \infty} MonteCarlo^{m}_{[x_{\min}, x_{\max}]}(f, n)$$

marche que si f est positive, ie du type

$$f : \mathbb{R} \longrightarrow \mathbb{R}^{+}$$

En pratique, la fonction f ne sera pas nécessairement positive sur tout l'intervalle d'intégration $[x_{\min}, x_{\max}]$. Il faut alors adapter l'algorithme !

Applications PI

Aire du Cercle

Soit $f:[-1,1]\subset\mathbb{R}\longrightarrow\mathbb{R}^+, x\rightarrow y=\sqrt{1-x^2}$ l'équation du demi cercle de rayon 1. On a

$$\int_{-1}^{1} f(x)dx = \frac{\pi * 1^2}{2}$$

et aisni

$$\boxed{\pi = 2\int_{-1}^{1} f(x)dx} \quad \text{avec} \quad \boxed{f(x)=\sqrt{1-x^2}}$$

On peut obtenir une estimation de π avec MonteCarlo :

$$\boxed{\widehat{\pi} = 2 * MonteCarlo_{x\in[-1,1]}^{m=1}(f,n)}$$

On a aussi

$$\boxed{\pi = 4\int_{0}^{1} f(x)dx} \quad \text{avec} \quad \boxed{f(x)=\frac{1}{1+x^2}}$$

Implémentation OpenMP

Effectuez 3 implémentations différentes (cf cours) :

- *Entrelacer_PromotionTab*
- *Entrelacer_Atomic*
- *For_atomic*

et comparer les performances ! Pour l'étude des performances, varier aussi les compilateurs, et les OS ! N'oubliez pas l'implémentation séquentielle.

Principe Cuda

Contexte

Supposons que l'on tiureun nombre quelquonque *n* de féchettes. Prenons pas soucis de commodité pour le schéma :

- 4 threads par blocs
- 2 blocks par multi processeurs (MP)

Utilisons une organisation hiérarchique des threads monodimensionnel tant pour la *grille* que pour les *blocks*. On applique le pattern entrelacement standard afin de parcourir les *n* éléments de nos deux vecteurs *v* et w.

Pattern Entrelacement standard

```
const int tid=threadIdx.x+(blockIdx.x*blockDim.x) ;       //global à la grille
const int tidLocal=threadIdx.x;                           //local à un block
const int NB_THREAD= blockDim.x*gridDim.x;                //nbThreadTotal

int s=tid;
while(s<n)
      {
      // work with : s

      s+=NB_THREAD;
      }
```

Réduction IntraThread

Chaque thread s'occupe de tirer un certains nombres de fléchettes, plus précisément

```
n/ NB_THREAD
```

et stocke par soucis de performance son résultat intermédiaire dans une **variable locale** (dans les registres). Cette première *réduction*, dite réduction **intraThread** fournit le nombre de fléchettes tirées par exemple sous la courbe par le thread incriminé. Celui-ci stocke enfin sa comptabilité en *shared memory* (SM).

Une réduction *intraBlock* puis *interBlock* consolide le résultat de chacun des threads.

Réduction Intra block

Au cours, on a vu comment effectuer une *réduction IntraBlock.* L'algorithme expliqué est générique et indépendant du TP ! Il s'agit juste de l'appliquer ! Le seul paramètre d'entrée est tabSM et rien d'autres !

```
/**
* Hyp : dimension(tabSM) est une puissance de 2
/*
__device__ void reductionIntraBlock(float* tabSM) ;
```

La dimension de celui-ci pourra se récupérer à l'intérieur avec

```
blockDim.x
```

Cet algorithme de *réduction intraBlock* est indépendant de la taille *n* du vecteur ! Il s'occupe juste de réduire `tabSm`, quel que soit la manière dont `tabSM` a été peuplé. Cet algorithme de *réduction intraBlock* pourra être repris de TP en TP : il est générique !

Réduction Inter block

Cf cours. Cet algorithme est aussi générique !

Implémentation Cuda

Indications

(I1) Utilisez l'API *currand* pour tirer des nombres aléatoires.
Voir ***BilatCudaPracticalGuide.pdf*** pour en savoir plus.

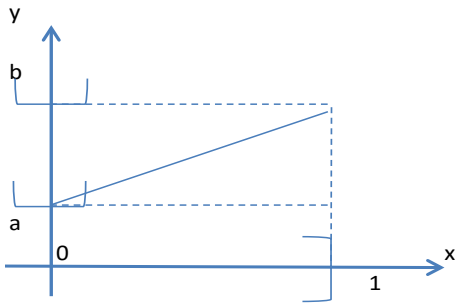
(I2) Indirectement des kernels sont appelés par l'api *curand*.

(I3) Préférez la version ou on tire des nombres aléatoires en les exploitants tout de suite. Ca permet d'en tirer plus, car dans le cas où on les stocke dans des tableaux, on est limité par la taille des tableaux.

On lance un premier kernel fabriquant les générateurs.
On lance ensuite un deuxième kernel utilisant les générateurs.

Le challenge est que tous les threads de tous les *MP* de tous les GPU tirent une suite de nombres aléatoires différents. Pour cette raison, on essaye d'utiliser les variables *deviceID* et *tid* lors de la fabrication des générateurs !

(I4) Si les nombres aléatoires tirés sont tirés entre $[0,1[$, on peut les transformer en des nombres compris dans $[a,b[$ par la transformation affine suivante



dont la forme algébrique est

$$y = \frac{b-a}{1}x + a \qquad \left(y \in [a,b[\;\; si \;\; x \in [0,1[\right)$$

Variations

(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 ?

Optimisation

(O1) *Unroll loop*

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

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

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

Cuda GPU Timeout

Rappels

(R1) 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*. Notion de ***wathdog***

(R2) Le nombre de thread maximum n'est pas arbitrairement grand. Il est limité par l'architecture du GPU. Pour obtenir ces informations, utiliser la classe ***Devices***. Par exemple la méthode *printALL* vous donne toutes les informations dont vous avez besoin pour tous les devices disponibles.

Trucs

(T1) Pour voir si *device* à un timeout, 2 méthodes :

- Utilisez la classe ***Devices*** et la méthode *printALL*
- Utilisez l'utilitaire ***nvidia-smi –q***
  (disponible seulement sur GPU haut de gamme)

Cuda Multi GPU

Voir le TP Multi-GPU.
A réaliser plus tard …

Speedup

Mesurer les coefficients de *speedup* des différentes implémentations.

Pour canevas, utiliser le document

speedup_simple.xls.

Au besoin adapter ce canevas.

Pertinence du critère de speed-up

La technique de *MonteCarlo* a ses limites, en partie du fait aussi que les nombres aléatoires parfait n'existe pas. Par ailleurs, la suite de nombres tirés par un générateur est finie, et étendue de manière cyclique. On retombe alors sur les mêmes nombres aléatoires déjà tirés (périodicité).

Fléchettes par seconde

Pour ces raisons, il peut être intéressant de faire du speed-up sur la quantité de nombres que l'on peut tirer sur une période *dt* (par exemple *dt*=1) , et non pas sur le temps nécessaire à obtenir *x* décimal de *PI*.

Octet par seconde

Sachant qu'une fléchette coute 2 floats (un pour x, l'autre pour y) soit 8 octets, on peut ainsi faire du benchmarking sur le nombre d'octets par seconde que l'on peut tirer :

- En mono GPU
- En multi GPU

Pertinence du chronométrage

Au début de *main*, forcer le chargement des drivers sur tous les GPU avec

Devices :: loadCudaDriverDeviceAll()

afin de ne pas fausser le chronométrage !

Dimension

Essayer par exemple avec

```
#ifdef _WIN32
```

```
    long n=LONG_MAX; // sous windows INT=LONG
#else
    long n =((long)INT_MAX)*80;
    //long n =INT_MAX ; // same value windows
#endif
```

End