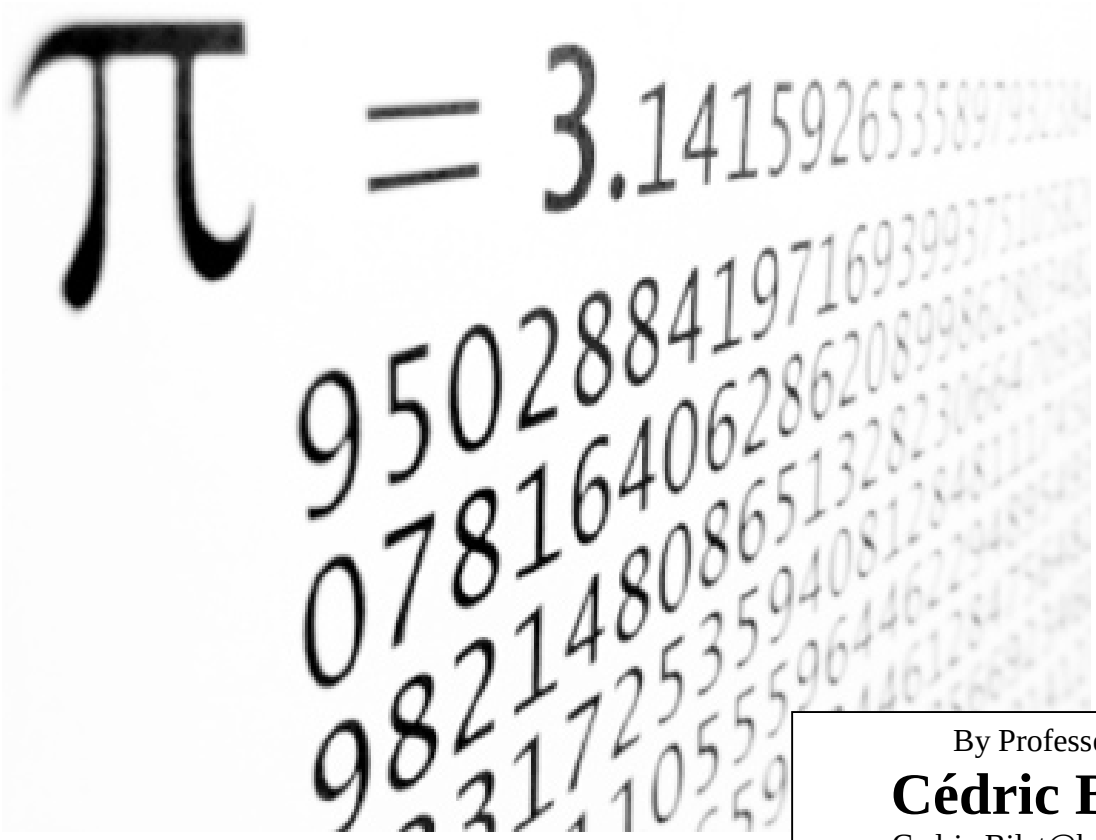
Problèmes Parallelisation



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

Slice

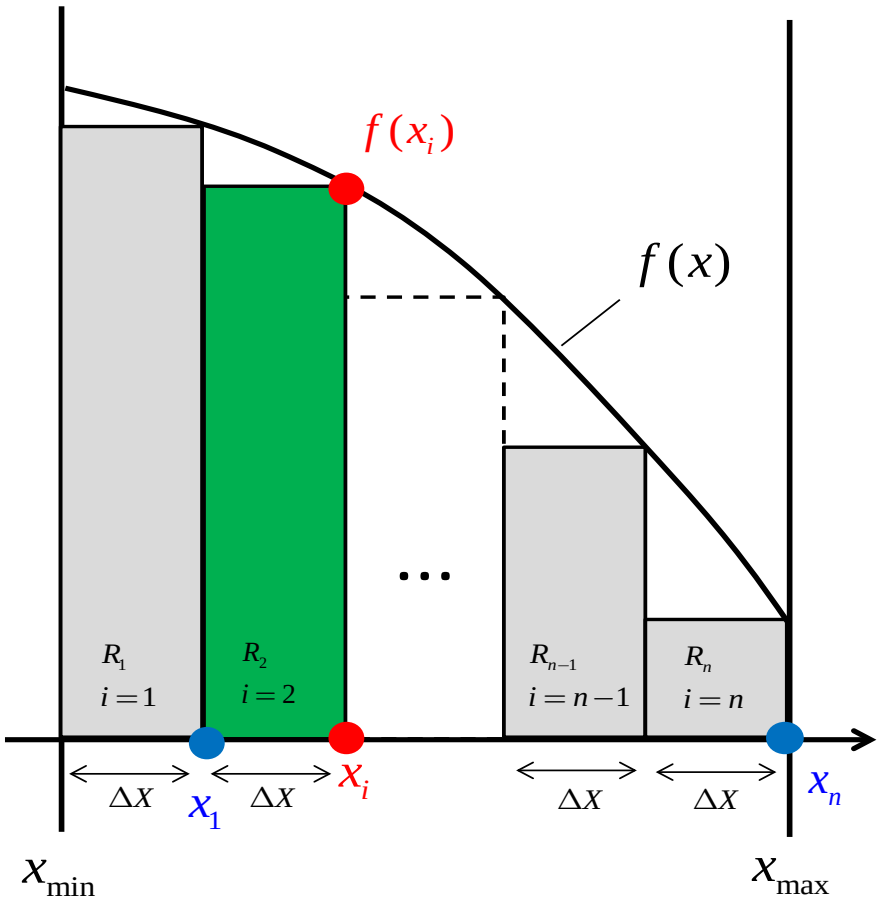
Intégration numérique *Slice*

Contexte

Il est possible de calculer une intégrale

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

en saucissonnant l'aire d'intégration en rectangles ou trapèzes :



Principe

On saucissonne en n tranches Δx le domaine d'intégration $\left[x_{\min}, x_{\max}\right]$.L'aire totale vaut alors la somme des aires sur chacune des tranches du saucisson. L'aire de chacune des tranches peut être approximée par un rectangle (ou un trapèze).

Formule

$$\int_{x_{\min}}^{x_{\max}} f(x)dx \cong \sum_{i=1}^{n} f(x_i)\Delta x$$

$$\cong \Delta x \sum_{i=1}^{n} f(x_i)$$

où x_i est définit par

$$\begin{cases} x_1 = x_{\min} \\ x_{i+1} = x_i + \Delta x \end{cases}$$

avec

$$\boxed{\Delta x = (x_{\max} - x_{\min}) / n}$$

On a aussi

$$\boxed{x_i = x_{\min} + i\Delta x \qquad \forall i \in [1, n] \subset \mathbb{N}}$$

Applications Calcul de 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 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 7 implémentations différentes (cf cours) :

- *Entrelacer_PromotionTab*
- *Entrelacer_Critique*
- *Entrelacer_Atomic*
- *For_Atomic*
- *For_Critical*
- *For_PromotionTab*
- *For_Reduction*

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 a décomposé l'intervalle d'intégration en *n* tranches. 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* rectangles dont on doit calculer l'air.

Pattern Entrelacement standard

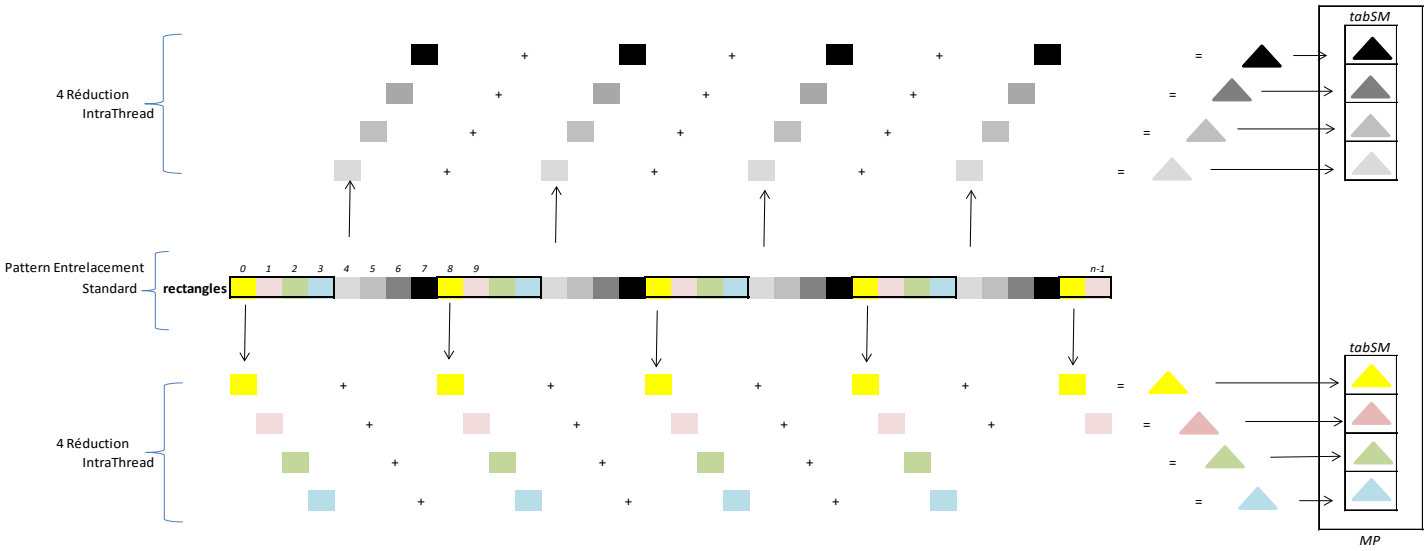
```
const int tid=threadIdx.x+(blockIdx.x*blockDim.x) ;       //global à la grille
const int tidLocal=threadIdx.x;                           //local à un block
const int NB_THREAD= blockDim.x*gridDim.x;                //nbThreadTotal

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

      s+=NB_THREAD;
      }
```

Réduction IntraThread

Le thread jaune s'occupe alors de calculer et d'additionner les rectangles jaunes de la surface d'intégration qu'il parcoure, et stocke par soucis de performance ces résultats intermédiaires dans une **variable locale** (dans les registres). Cette première réduction, dite réduction **intraThread** fournit le triangle jaune dans le schéma suivant :



Il suffit ensuite de copier ce triangle jaune en **shared memory** (SM)

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 du nombre *n* de rectangles! 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 !

L'algorithme de réduction intraBlock fait juste l'hypothèse que `tabSM` est une puissance de 2. A vous de choisir une dimension de block idéal satisfaisant cette contrainte. Cette hypothèse n'est pas réductrice. Il n'y a aucune relation entre cette taille *db.x* et le nombre *n* de rectangle.

Réduction Inter block

Cf cours. Cet algorithme est aussi générique !

Implémentation Cuda

Piège

N'oubliez pas l'initialisation de

- La variable en GM contenant le résultat final.
- Des `tabSM` locaux

Dans le premier cas, un `cudaMemset` faira l'affaire.

```
size_t sizeOctet=sizeof(float);
HANDLE_ERROR(cudaMemset(ptrDevAireGM,0,sizeOctet)) ;
```

Dans le second cas, si vous travailler avec une variable locale pour stocker les résultats intermédiaires obtenus par le thread Jaune, le peuplement de `tabSM` se fera par écrasement. Cette technique est la meilleur car les variables locales sont dans les registres, et l'écrasement évite l'oubli de l'initialisation.

Attention

N'oubliez pas les

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

Mais soyez minimaliste !

Variation 1

Utilisez en SM un tableau de 1 case. 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],aireTidLocal) ;
```

ou de manière équivalente

```
atomicAdd(tabSM,aireTidLocal) ;
```

Que peut-on dire des performances ?

Variation 2

N'utilisez plus du tout de SM. Chaque thread ira updater une unique variable en GM. Le problème de concurrence se résout avec un

```
atomicAdd(&tabGM[0],aireTidLocal) ;
```

ou de manière équivalente

```
atomicAdd(tabGM,aireTidLocal) ;
```

Que peut-on dire des performances ?

Speedup

Mesurer les coefficients de *speedup* des différentes implémentations.

Pour canevas, utiliser le document

speedup_simple.xls.

Au besoin adapter ce canevas.

End