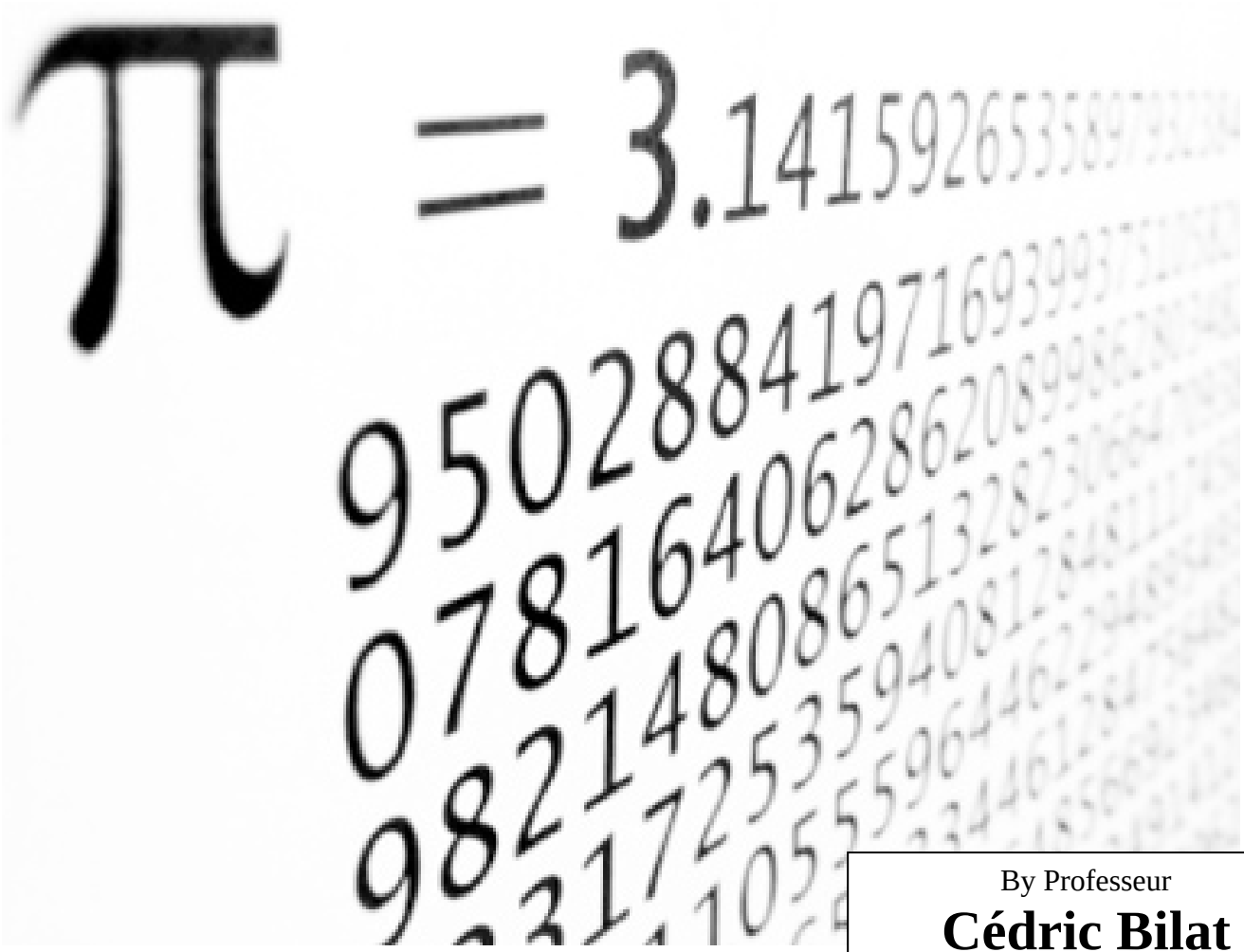
GPGPU



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

Slice

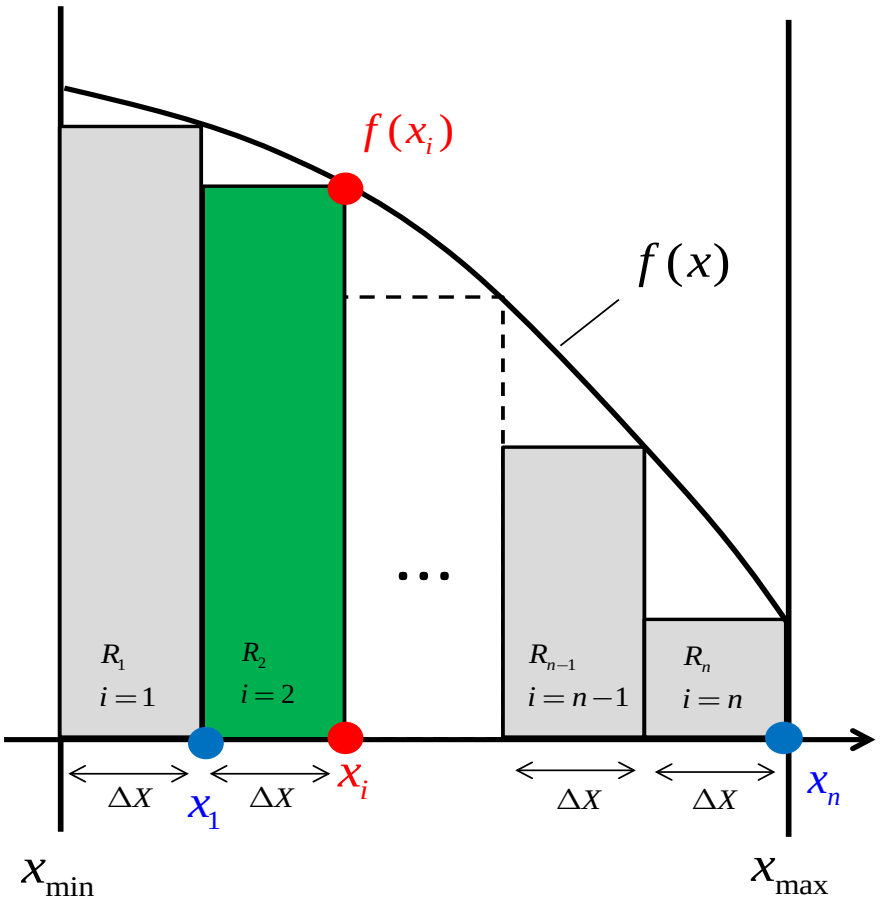
Intégration numérique *Slice*

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

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

OpenMP Implémentation

Effectuez 7 implémentations différentes (cf cours) :

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

et comparer les performances ! Pour l'étude des performances, varier aussi les compilateurs! N'oubliez pas l'implémentation séquentielle (que l'on obtient de puis la version for_auto en mettant en commentaire la ligne #pragma omp

Cuda Principe

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

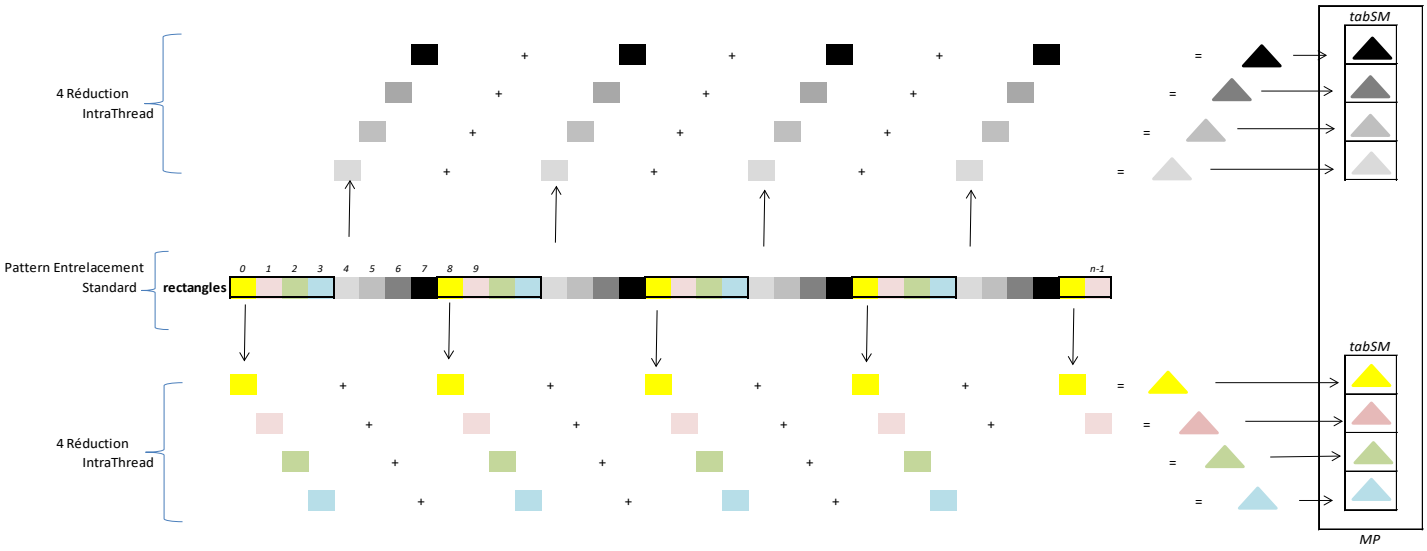
```
const int TID=threadIdx.x+(blockIdx.x*blockDim.x) ;       //global à la grille
const int TID_LOCAL=threadIdx.x;                          //local à un block
const int NB_THREAD= blockDim.x*gridDim.x;                //nbThreadTotal

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

      s+=NB_THREAD;
      }
```

Réduction IntraThread

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)

Cuda Rappel

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 !

Tip

Si vous avez déjà implémenté la classe *ReductionTools*, utilisez-là !

Cuda ReductionTools

Voir le TP séparé traitant de cette problématique !

Cuda Implémentation

Indication générales

(I1) Utiliser votre classe *ReductionTools*

(I2) Structurer votre code comme suit

```
__global__ void XXX(…)
        {
        __shared__ extern float tabSM[] ;

        reductionIntraThread(tabSM, ….) ;

        ReductionTools. reductionADD(tabSM, …) ;

        }
```

(I3) La fin du calcul peut se faire avantageusement cotée host de manière séquentielle

Pièges

(P1) N'oubliez pas la création et surtout l'**initialisation à zéro** de la variable contenant le résultat final :

```
float* ptrDevResultGM =NULL;
size_t sizeOctet=sizeof(float);
HANDLE_ERROR(cudaMalloct(&ptrDevResultGM,sizeOctet)) ;
HANDLE_ERROR(cudaMemset(ptrDevResultGM,0,sizeOctet)) ;
```

(P3) Faut-il initialiser les *tabSM* à zéros lors d'une réduction additive ? Oui et non, tout dépend de votre technique de « peuplement »

Conseil : *Peuplement par écrasement*

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 ! La variable locale au thread doit, elle, être initialisé, puis peuplé, mais pas la case en SM :

```
__device__ void reductionIntraThread(…)
        {
        …
        int sumThread =0 ;

        for(int i=1 ; i<=m; i++)
                {
                sumThread +=   ;// incrémenter par le résultat du thread courant

                }

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

Parcimonie

N'oubliez pas les

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

Mais soyez minimaliste !

Cuda Variation

Deux variations possibles parmies tant d'autres :

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 ?

End