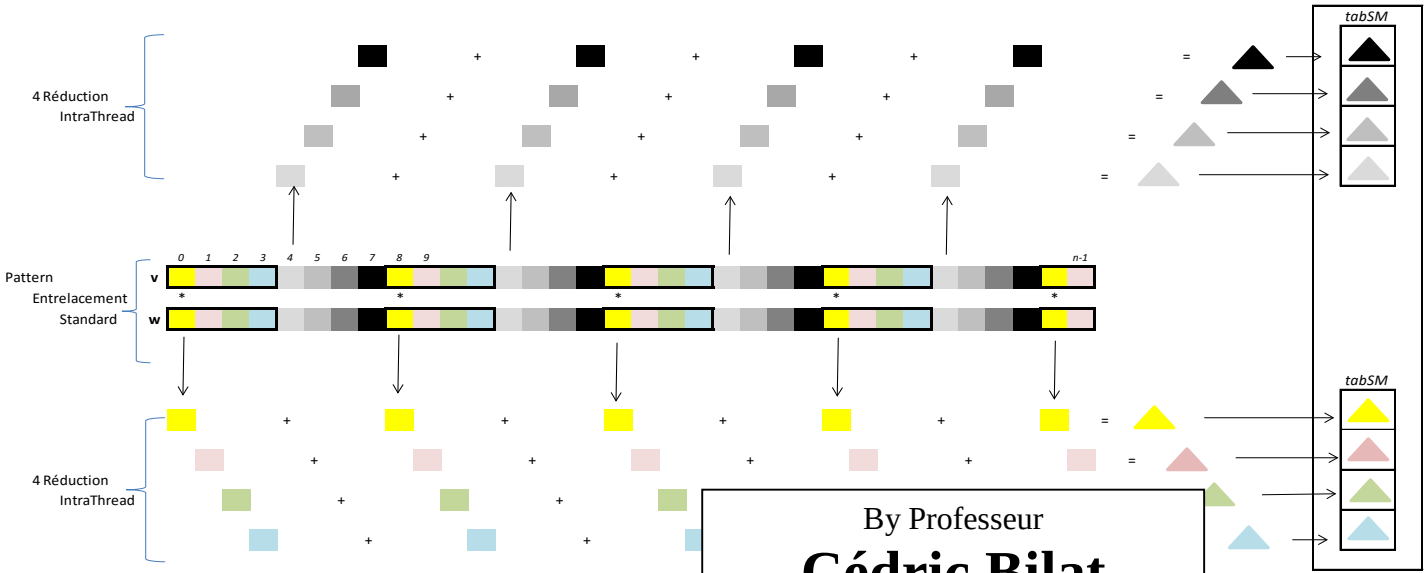
# Problèmes Parallelisation




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

# Produit Scalaire

# Produit Scalaire

**Définition**

Soit $v=(v_i)_{i\in[1,n]} \in \mathbb{R}^n$ et $w=(w_i)_{i\in[1,n]} \in \mathbb{R}^n$ deux vecteurs à $n$ dimension. Le produit scalaire $\langle v|w\rangle \in \mathbb{R}$ est définit par

$$\boxed{\langle v|w\rangle = \sum_{i=1}^{n}(v_i w_i) \in \mathbb{R}}$$

# Version Fonctionnelle

**Fonction**

Afin de pouvoir prendre un nombre de composante arbitrairement grand, les vecteurs ne seront pas stockés sous forme de tableaux, mais définit de manière **fonctionnelle**.

Par ailleurs, pour garantir un temps de calcul significativement différent de 0, on simule ici un calcul complexe avec les deux paramètres `M_W` et `M_V`. De plus, le résultat du calcul a l'avantage de pouvoir être obtenu avec une formule directe, ce qui permettra de facilement valider les résultats !

```
#define M_W 50 // ou plus grand !!
#define M_V 50 // ou plus grand !!


double v(long i)
    {
    double x = 1.5+ abs(cos((double) i));

    for (long j = 1; j <= M_V; j++)
       {
       double xCarre=x*x;
       x = x - (xCarre * x - 3) / (3*xCarre);
       }

    return (x/VI)*sqrt((double)i);
   }
```

```
double w(long i)
    {
    double x = abs(cos((double) i));

    for (long j = 1; j <= M_W; j++)
       {
       x = x - (cos(x) - x) / (-sin(x) - 1);
       }

    return (x/WI)*sqrt((double)i);
   }


double resultatTheorique(long n)
    {
    n--; // si la boucle du produit scalaire va de [0,n[
    return (n/(double)2) *(n+1);
    }
```

Les constantes `VI` et `WI` valent respectivement

Pour CPU OMP (calcul en double)

```
#define VI 1.442249570307408
#define WI 0.7390851332151607
```

Pour GPU Cuda (calcul en double)

```
#define VI 1.4422495703074083
#define WI 0.7390850782394409
```

# Analyse Mathématique

**Adaptation des constantes** : *Pour les curieux*

Si vous avez des problèmes de précisions de vos résultats finaux, vous pouvez éventuellement adapter les constantes `VI` et `WI` à votre code selon la règle suivante

Exemple `VI`

Recette :

Lancer votre code.
Afficher *x*.

Observation :

Quel que soit i, la valeur de x est la même !

Action :

Le *define* doit correspondre à cette valeur constante de *x* que vous observez.
Autrement dit *(x/VI)* doit toujours valoir 1.

Math :

L'algorithme modélisé par la boucle est un algorithme de Newton convergeant toujours au même endroit `VI`, quel que soit le point de départ, modulo une petite adaptation :

```
double x = 1.5+ abs(cos((double) i));
```

Evidemment plus *M_V* et *M_W* sont grands plus la convergence sera robuste. N'hésitez donc pas à prendre des grandes valeurs pour *M_V et M_W* !

# Typage

Attention le résultat théorique est très sensible à ces constantes ! Il faut impérativement utiliser les bons types pour effectuer les calculs !

Pour CPU OMP :

Tout en **double** !

Pour GPU Cuda :

Tout en **double** mais alors assez lent ! GPU plus performant en float !
Essayer d'être plus minimaliste !
Pour le calcul de *w* essayez de passer *x* en *float* et d'utiliser

**cosf**
**sinf**

au lieu de

**cos**
**sin**

Les 2 premières implémentations travaillent en *float*, alors que sans le ***f*** elles travaillent en *double*. Pour la racine utiliser toujours la version *double*

**sqrt( (double)i )**

# Implémentation OpenMP

Effectuez 7 implémentations différentes (cf cours) :

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

Comparer les performances ! Pour l'étude des performances, vous pouvez varier :

- Les compilateurs
- Les OS
- La taille *n* des vecteurs *v* et *w*
- Le nombre d'itération de Newton *M_V* >50 et *M_W>50*
- Le code séquentiel

# Principe Cuda

## Contexte

Supposons un vecteur de taille quelquonque *n*. 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

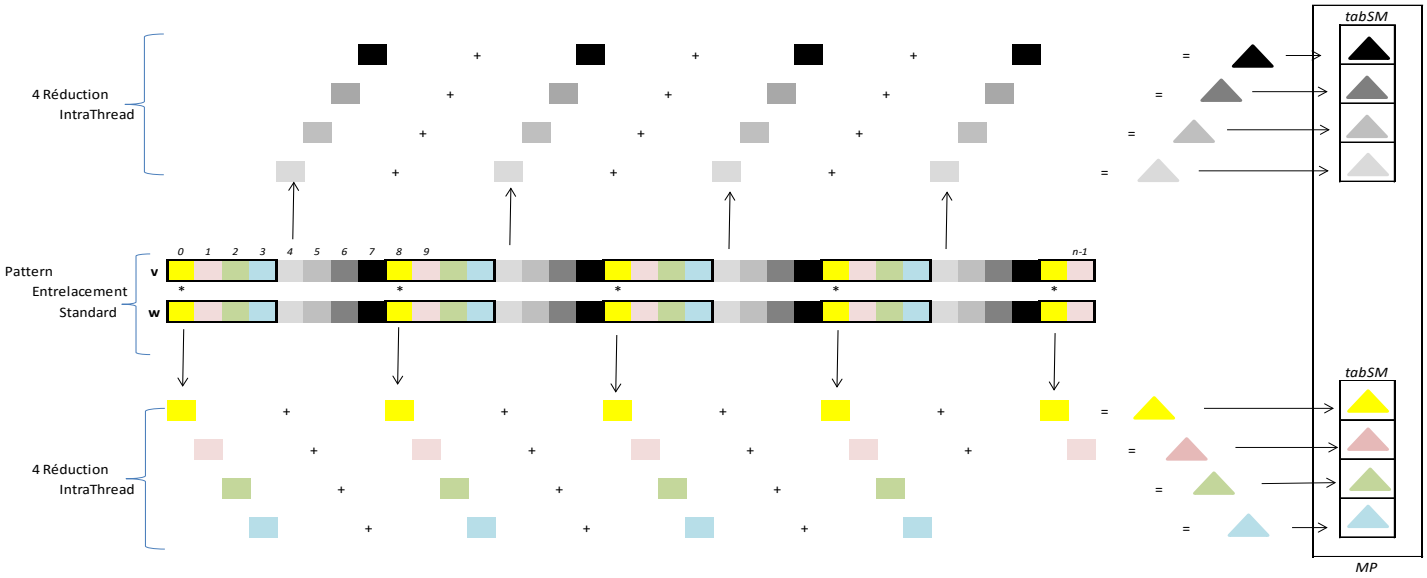
```
const int tid=threadIdx.x+(blockIdx.x*blockDim.x) ;       //global à la grille
const int tidLocal=threadIdx.x;                           //local à un block
const int NB_THREAD= blockDim.x*gridDim.x;                //nbThreadTotal

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

      s+=NB_THREAD;
      }
```

## Réduction IntraThread

Le thread jaune s'occupe alors de multiplier les cases de *v et w* 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 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

Important

Afin de pouvoir effectuer une **réduction intra-bloc** parallèle 2 à 2, par dichotomie successive, vous vous arrangerez pour que la taille des tableaux en **shared memory** SM ait comme taille une puissance de 2 !

Il s'agit d'une hypothèse technique non réductrice. La taille des vecteurs *n* reste elle quelconque !

Contrainte

On utilisera la syntaxe longue pour le lancement du kernel

```
size_t sizeSM=sizeof(double)*db.x ; // hyp : db 1D (puissance de 2)
kernel<<<dg,dg, sizeSM >>>(….);
```

qui permet de ne pas spécifier la taille du tableau `tabSM` en **shared memory** à la compilation. Ce tableau `tabSM` se déclarera alors dans le kernel comme suit

```
extern __shared__ double tabSM[]  ;
```

C'est ce tableau `tabSM` qui recevra le résultat de la réduction **intra-thread**

A ce niveau le type *float* pourrait peut-être suffire!

Rappel

Il y a un **tabSM** par block !
Deux threads d'un même block peuvent donc coopérer via **tabSM**.

Maladresse

La taille des tableaux tabSM en Shared Memory n'a rien à voir avec la taille *n* des vecteurs !

```
size_t sizeVector=sizeof(double)*n ;   // hyp : n  = taille des vecteurs
size_t sizeSM=sizeof(double)*db.x ;    // hyp : db 1D (puissance de 2)
```

Evitez cette confusion!

Piège

N'oubliez pas d'initialiser à zéro le zone en global memory GM contenant le résultat final. Par exemple, utiliser un **cudaMemset** après le **cudaMalloc** pour mettre cette zone à 0.

```
size_t sizeOctet=sizeeof(float)
float value=0;
HANDLE_ERROR(cudaMemset(prtDevOutput, value,sizeOctet)) ;
```

Vérifier si ce problème d'initialisation pour les tableaux en SM doit être effectué ou non.

Attention

Votre code contient certainement trois procédures

```
reductionIntraThread
reductionIntraBlock
reductionInterblock
```

Réfléchissez bien au

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

qui représente une barrière de synchronisation pour tous les threads d'un même block.

Soyez minimaliste ! Mais votre code doit être robuste !

Validation

Votre code doit être robuste quelque soit :

- La taille du vecteur n
- La dimension de la grillle *dg* *(mono dimensuonelle)*
- La dimension des blocks *db* *(mono dimensionelle, puissance de 2)*

Faites donc varier ces paramètres pour illustrer d'éventuels problèmes de votre code. N'ayez pas peur d'exécuter deux fois de suite le même code, pour valider la justesse de vos initialisations ! A cette effet, enlever provisoirement le **cudaMemset** et observer les dégâts !

Variations A

(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 ?

Depuis cuda 5, `atomicADD` a été « super optimisé » en GM. Vous en pensez quoi ?

Variations B

(B1) Promotion tab

En effectuant une réduction intraBlock avec la technique des promotion de tableau et en effectuant la réduction coté CPU.

(B2) Lock (see BilaToolsCuda)

En effectuant une réduction intraBlock en utilisant la classe ***Lock*** mis à disposition, et en effectuant la réduction complète côté device !

(B3) LockGPU (see BilaToolsCuda)

En effectuant une réduction intraBlock en utilisant la classe ***LockGPU*** mis à disposition, et en effectuant la réduction complète côté device ! Contrairement à la classe *Lock* qui laisse une trace coté host, la classe *LockGPU* ne laisse aucune trace côté cpu. En effet cette classe est instancier coté *device* seulement !

(B4) API réduction génériques (see BilaToolsCuda)

En utilisant les classes de réduction génériques (inter et intra block) mises à disposition.

Optimisation

Dans les *makefile* proposées l'option suivante

**-use_fast_math**

qui est équivalente à

-ftz=true -prec_div=false -prec_sqrt=false

est déjà activée ! Essayer sans !

# Speedup

Mesurer les coefficients de *speedup* des différentes implémentations.

Pour canevas, utiliser le document

***speedup_simple.xls.***

Au besoin adapter ce canevas.

Essayer en particulier pour

int n=INT_MAX/10;

Par rapport à un code GCC séquentiel, un speedup d'environ **400** fois devrait être obtenu.

N'utilisez pas le même *device* que votre collègue, ni le *device* 0 que l'on utilise pour les TP graphiques. On peut spécifier le device avec

```
HANDLE_ERROR(cudaSetDevice(i) );
```

Vous pouvez vérifier le taux d'occupation du GPU avec l'utilitaire *nvidia-smi*. Dans une console, en parallèle à votre code :

nvidia-smi --loop=2

# 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)

# End