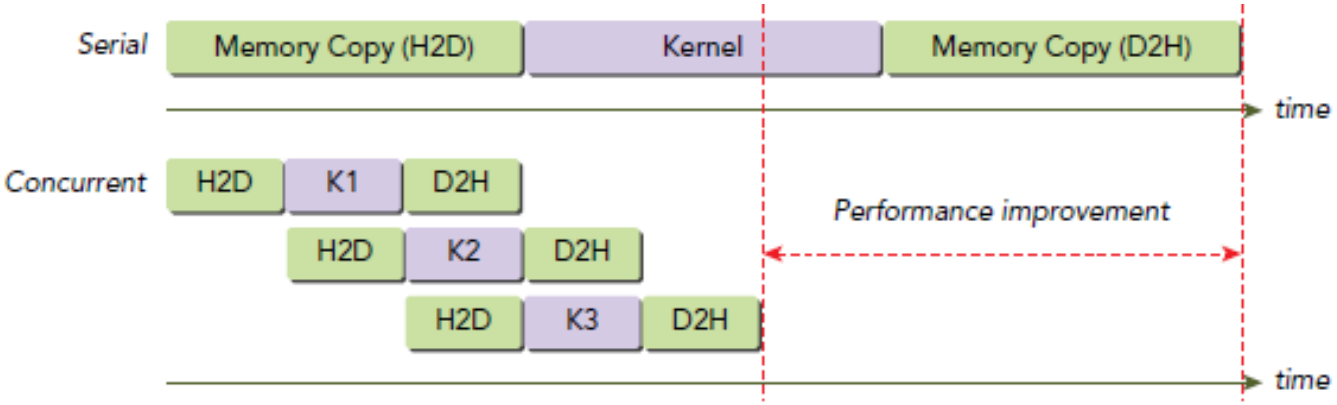
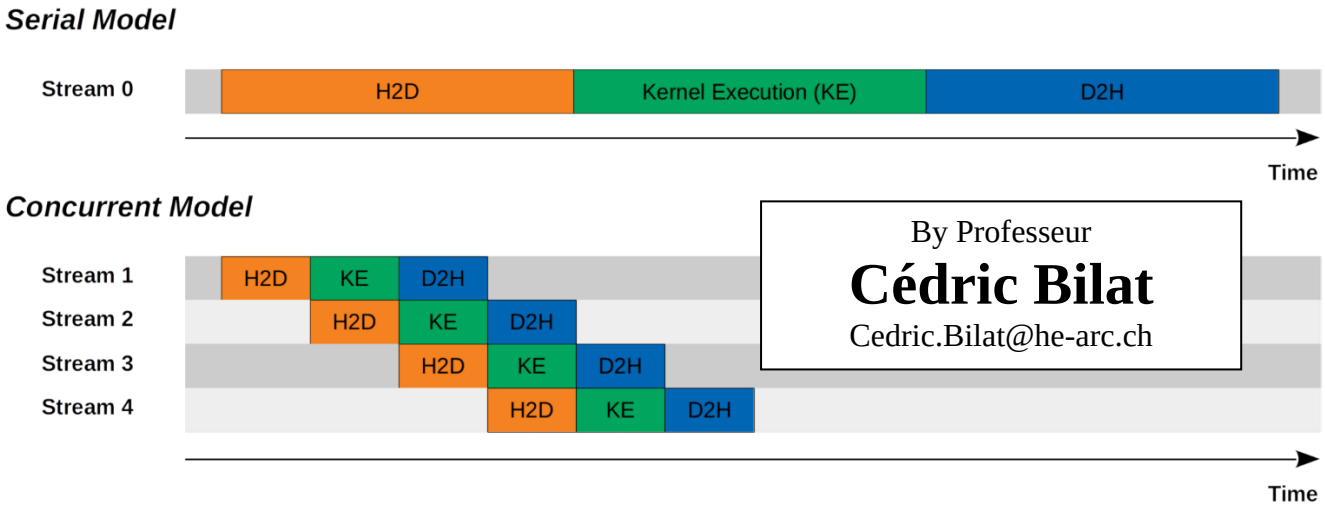
GPGPU





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

Vector

Stream

Contexte

Un GPU contient 3 engines pouvant travailler en parallèle :

- Engine de copy HToD
- Engine de copy DToH
- Engine de processing de kernel

Pour avoir le code le plus performant, il s'agit d'utiliser si possible ces 3 engines en même temps. Sur un GPU on essaie toujours d'employer tout le hardware en même temps : On essaye d'utiliser en même temps :

- tous les core (pas de core chômeur, cf heuristique I)
- tous les MP (pas de MP chômeur, cf heuristique II)
- tous les GPU
- toutes les SM

et dorénavant aussi les 3 engines ci-dessus. Cette optimisation ne s'applique qu'au problème ou il y a des données volumineuses à échanger entre le *host* et le *device* et réciproquement. On parle de programmation massivement parallèle,

Rappelons par ailleurs que le *pci-express* entre le *host* et le *device* est le goulet d'étranglement d'un code GPGPU. L'utilisations en parallèle des 3 engines ci-dessus, permet d'effectuer des transferts de portion de données, pendant qu'on process en même temps sur d'autre portions. Il s'agit donc de découper en ***slice*** nos données, de les transférer ***slice par slice*** coté *device*, et de les processer ***slice par slice.***

Mathématique

$$w(i) = v_1(i) + v_2(i) = \quad i \in [0, n[$$

Analyse

Pour utiliser les **3** engines en parallèles on va employant la technique des *streams*, plus précisément **3** *streams*, pour employer les 3 engines en même temps. Le déroulement de l'algorithme est schématisé juste un peu plus tard dans le document.

Petit problème en amont

L'addition de deux vecteurs est trop rapide sur un GPU, trop rapide, si bien que dans cet exemple d'addition vectoriel qui se veut simple, on ne verra pas l'effet des streams, car le temps nécessaire à l'engine de processer une tranche de vecteur est trop petite. Le TP passe son temps à transférer des data sur le bus pci-express.

Correction

On va donc artificiellement faire perdre du temps à l'engine de calcul, en luis faisant faire des calculs inutiles, avec la fonction ***loseTime***, qui renvoie l'input reçu, mais en perdant du temps

$$w(i) = v_1(i) + v_2(i) = \quad i \in [0, n[$$
$$return\ loseTime(w(i))$$

Fonction LoseTime

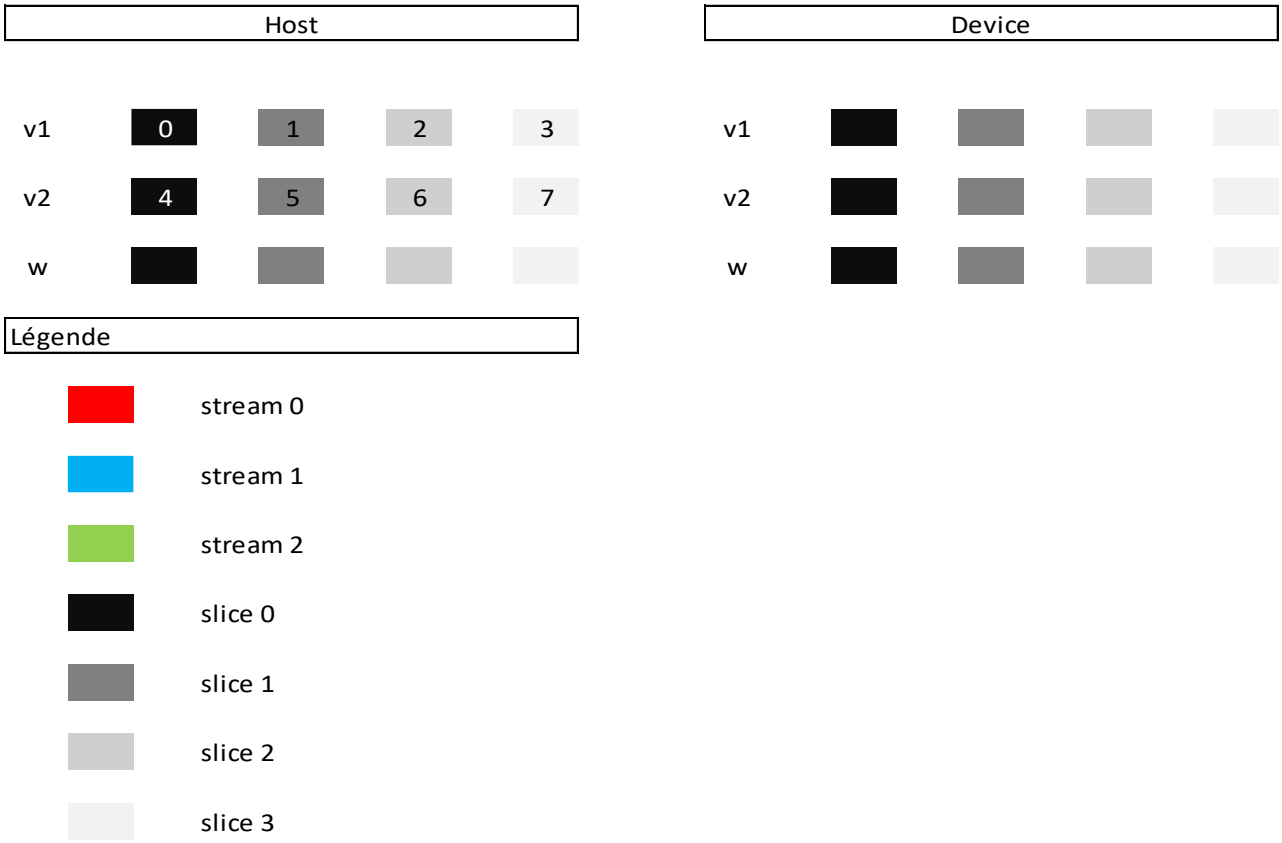
. Il n'est pas si évident de coder une fonction *loseTime*, car le compilateur ***nvcc*** recherche et enlève le « ***code mort*** ». Une astuce consiste à utiliser la variable résultat, à faire des calculs avec, puis à revenir sur nos pas. On code une fonction identité qui emploie la variable résultat, par exemple comme suit :

```
__device__ int loseTime(int u)
    {
    // Plus le GPU est performant plus il faut prendre grand
    const int N = 300; // chercher speed up de 2.1

    long uu=u;
    int t = 0;

    while (t < N)
          {
          t++;
          uu = uu +inc(t);
          }

    while(t>=1)
          {
          uu = uu - inc(t);
          t--;
          }

    return (int)uu;
   }
```

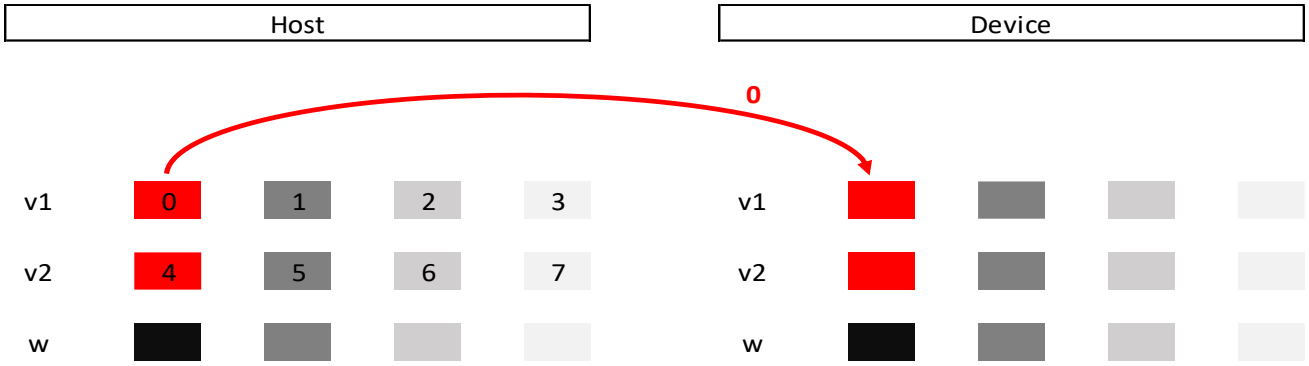
Algorithme avec Stream

On suppose ici de prendre 4 slices, représentés par les 4 colonnes grises ci-dessous. Dans les schémas ci-dessous, par commodité, les slices sont de size 1.

Init et Légende

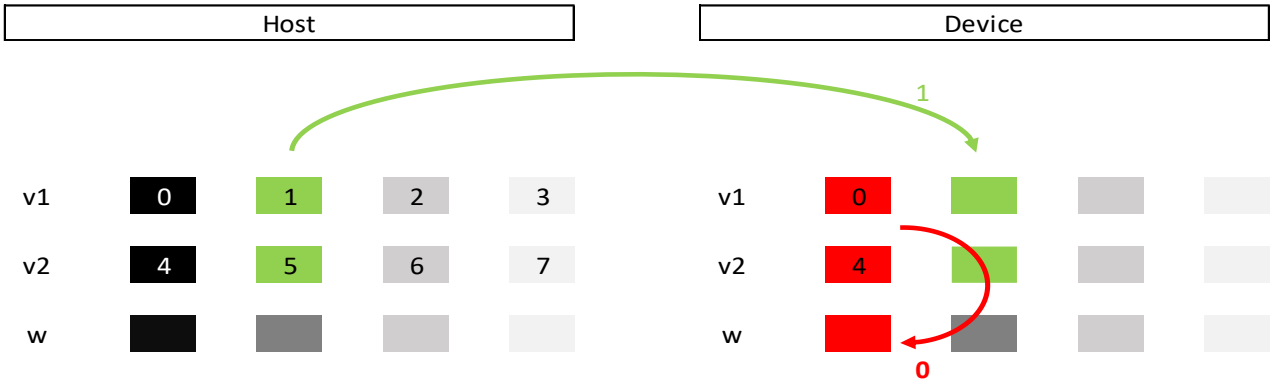


Step 1



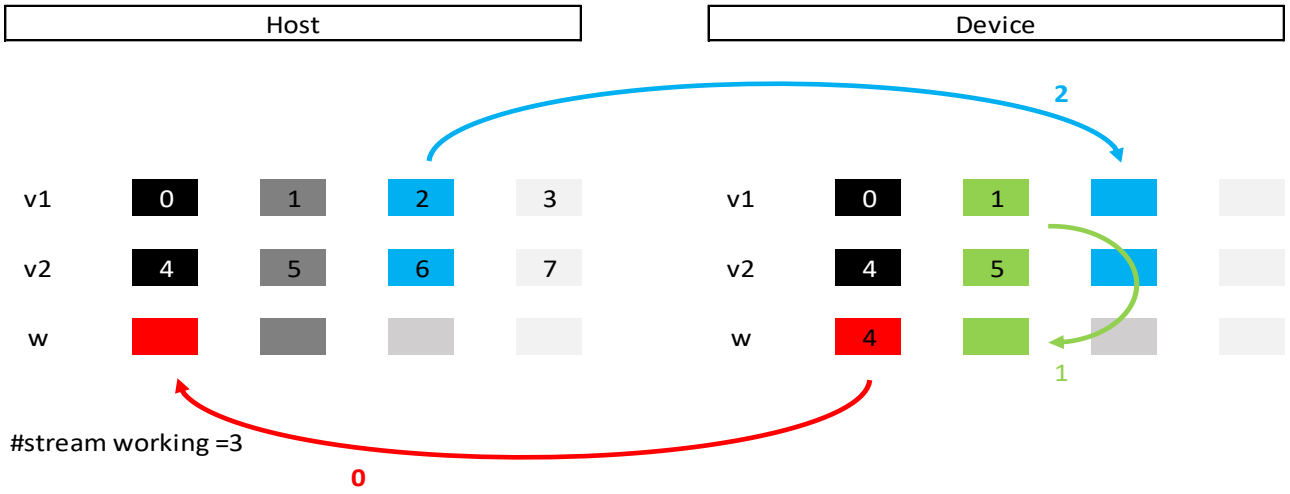
#stream working =1

Step 2



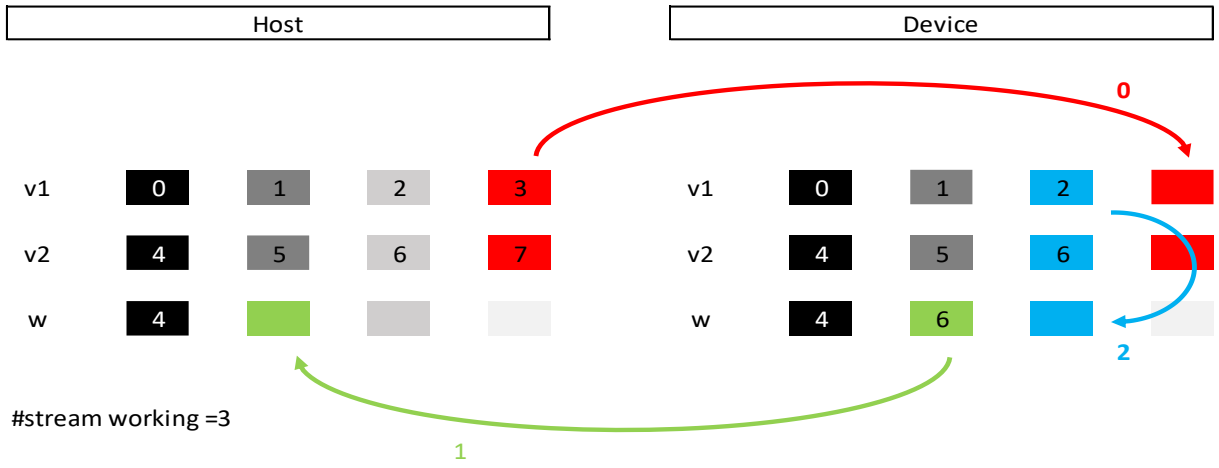
#stream working =2

Step 3

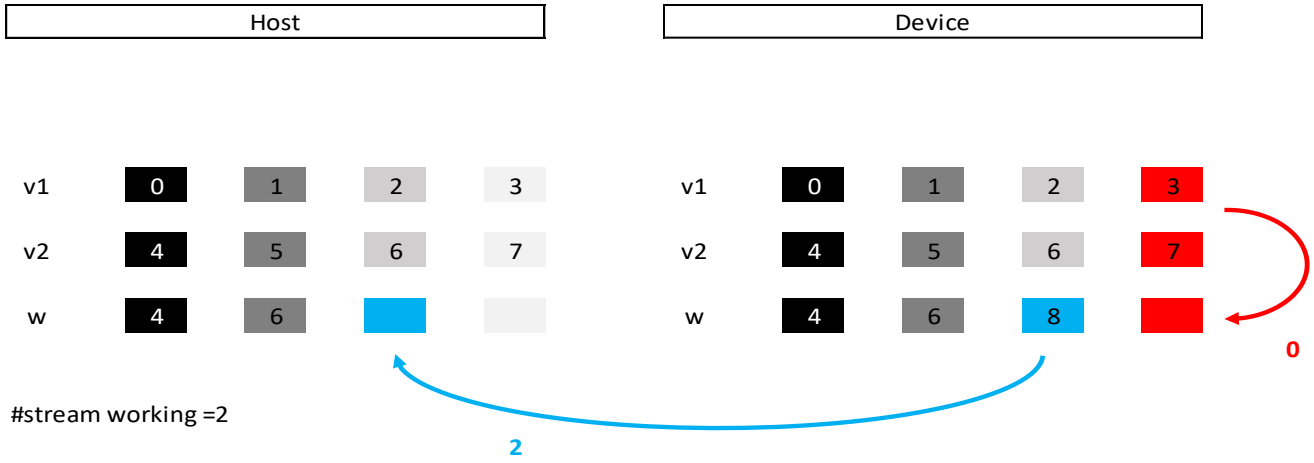


#stream working =3

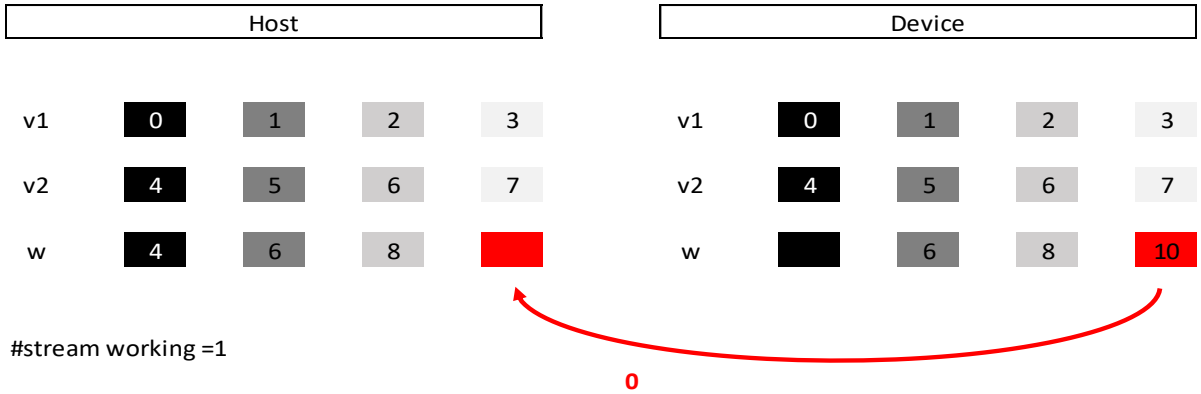
Step 4



Step 5



Step 6



Warning :

Le numéro colorer sous les flèches sont l'id de la stream (in [0,2]

Pédagogie

Vous devez fournir 3 implémentations pour cette addition :

- GPU naïve (sans stream)
- BiStream, 2 slices
- TriStream, muti slices (generic)

La première servira de base line pour la comparaison des performances, et la détermination du ***speedUP***.

La deuxième est là pour s'échauffer tranquillement avec ces *streams* et ces slices. Le cas échéant refaite un schéma comme ci-dessus, mais avec 2 slices et 2 *stream*.

La dernière est l'implémentation générique que l'on vise, qui doit marcher quelque soit le nombre de slice. Dans les squelettes de codes il vous est conseillé de vous échauffer, encore, et de traiter d'abord les cas particuliers ci-dessous :

- 3 slices
- 4 slices
- 5 slices
- Puis générique

Pour voir la justesse de votre code, travailler avec le ***mainUse***, activer l'affichage en inversant provisoirement les flags de verbosité comme suit (rouge) :

```
AddVectorUse addVectorUse(!IS_VERBOSE);
AddVectorBistreamUse addVectorBistreamUse(!IS_VERBOSE);

isOk &= addVectorUse.isOk(!false);
isOk &= addVectorBistreamUse.isOk(!false);

for (int nbSlice = 3; nbSlice <= 3; nbSlice++)
      {
      AddVectorTristreamUse addVectorTristreamUse(nbSlice, !IS_VERBOSE);
      bool isOkSlice = addVectorTristreamUse.isOk(!false);
      cout << nbSlice << "\t" << isOkSlice << endl;
      isOk &= isOkSlice;
      }
```

Et prenez peu de cases, en modifiant provisoirement la taille ***n*** du nombre de case, dans

```
VecteurTools ::n()
```

Prenez des valeurs comme 4, 6 , ou 10 selon le nombre de slice.

Hypothèse Simplificatrice

On va se faciliter la vie, pour la taille des slices soit identiques, autrement que :

$$N\%nbSlice = 0$$

où N est le nombre de case du vecteur. A cet effet, la taille du vecteur proposé est

N = 2 * 3 * 4 * 5 * 7 * 9 * 11 * 13 * 5

Ainsi N est divisible en particulier par :

nbSlice in [2,15]
nbSlice allant de 15 à 75 par pas de 5

Votre implémentation généric doit donc fonctionner pour le N ci-dessus. Pour les *warmup*, prenez un N petit pour pouvoir afficher le résultat. Par exemple :

nbSlice=3 →N=6
nbSlice=4 →N=8
nbSlice=5 →N=10

DeviceSide Piège

On effectue comme d'habitude un pattern d'entrelacement :

```
__global__ void addVector(int* ptrDevV1 , int* ptrDevV2 , int* ptrDevW
                          , int n , int sOffset = 0)
    {
    const int NB_THREAD = Thread1D::nbThread();
    const int TID = Thread1D::tid();

    int s = TID;
    while (s < n)
            {
            process(ptrDevV1, ptrDevV2, ptrDevW, s, sOffset);
            s += NB_THREAD;
            }
    }
```

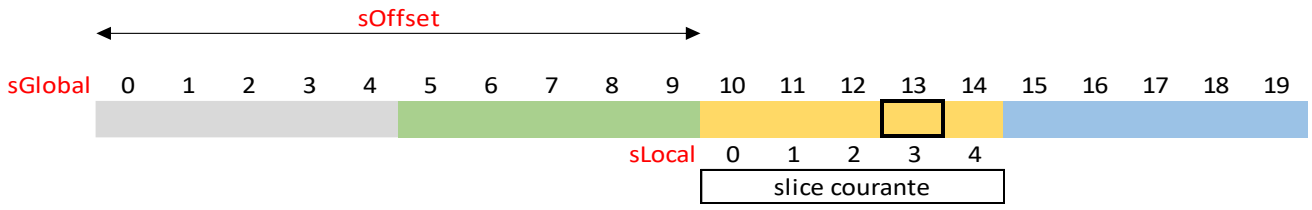
mais sur une slice seulement. Notre variable de travail s, ne fait que parcours la slice courante. On pourrait presque dire que

s est local, local à la slice

Or il se pourrait qu'on ait besoin de savoir où on se trouve dans le vecteur complet. On aimerait peut-être avoir besoin

du s global au vecteur

A cet effet, on passe la variable ***sOffset*** à notre kernel.



On a la relation :

$$sGlobal = sLocal + sOffset$$
$$avec\ sOffset = indexSlice * nBySlice$$

Exemple (du schéma ci-dessus)

$$nBySlice = 5$$
$$indexSlice = 2$$
$$sOffset = 10$$
$$sGlobal = sOffset + sLocal = 10 + 3 = 13$$

HostSide Astuce

Coté host, il peut être utile de s'aider des trois méthodes utilitaires :

```
/**
 * copyHtoD: la slice sliceIndex pour
 *        v1
 *        v2
 * ou
 *        sliceIndex in [0,nbSlice[
 */
void AddVectorTristream::copyHtoD(int sliceIndex , cudaStream_t stream)
    {
    const int OFFSET_SLICE = sliceIndex * N_BY_SLICE;
    GM::memcpyAsyncHToD(…);
    GM::memcpyAsyncHToD(…);
    }

/**
 * copyDtoH: la slice sliceIndex pour
 *        w
 * ou
 *        sliceIndex in [0,nbSlice[
 */
void AddVectorTristream::copyDtoH(int sliceIndex , cudaStream_t stream)
    {
    const int OFFSET_SLICE = sliceIndex * N_BY_SLICE;
    GM::memcpyAsyncDToH(…);
    }

/**
 * lance le kernel de calcul pour la slice sliceIndex
 * ou
 *  sliceIndex in [0,nbSlice[
 */
void AddVectorTristream::kernelSlice(int sliceIndex , cudaStream_t stream)
    {
    const int OFFSET_SLICE = sliceIndex * N_BY_SLICE;
    addVector<<<dg,db,0,stream>>>(… OFFSET_SLICE);
}
```

Host Side Généralisation

Après vous êtes échauffés avec 3 slice, puis 4, voir 5, il s'agira de produire un code générique fonctionnant quel que soit le nombre de slice. Le code est décomposé en 3 parties :

- Partie Init
- Partie Centrale (la généralisation est ici, boucle, 3 stream en //)
- Partie Finale

Partie centrale

Il y a deux choses à généralisé :

L'index de slice

Aucune difficulté, logique !

Stream à utiliser

Il suffit juste d'observer qu'il faut effectuer une ***permutation circulaire***

Partie Finale

L'index de slice

Facile !

Stream à utiliser

Il faut utiliser les streams de *sortie* de la *partie centrale.*

Optimalité

Avec la taille **N** fixé ci-dessus et 3 *streams*, combien doit on prendre de slice pour obtenir le max de fps ?

Warning

Selon le nombre de slice, la taille de la grille optimale n'est pas forcément toujours la même, embêtant ! Ca vous oblige presque a faire un bruteForce pour chaque grandeur de slice.

SpeedUP

Comparer la version naïve sans *stream*, avec votre version optimale. On vise une ***speedUP*** de l'ordre de 2, ie la version avec *stream* doit aller

> **2x** plus vite

que la version sans *stream*.

Conseil

Après avoir optimisé la taille de la grille et le nombre de slice, lancer

> ***mainBenchmarking.cpp***

Oublier de mettre le bon nombre de slice dans la méthode

> ***addvectorTristream***

Utiliser et updatre à cet effe *addvectorTristream* de ***mainBruteForce.cpp*** pour faire varier le nombre de slice de [0,75] par pas de 0.5. Oublier pas d'updater ensuite :

> *Grid AddVectorTristreamUse::createGrid(int nbSlice)*

Tests Unitaires

Les ***tests unitaires*** sont déjà codés. Il teste :

- Justesse
- Performance

Il ne reste plus qu'a les lancer !

Squelettes de codes

Des squelettes de codes vous sont fournis dans le workspace.

Complétez les ***TODO*** (requiert de comprendre le code fournit et son organisation)

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.

End