haute école neuchâtel berne jura arc ingénierie saint-imier le locle porrentruy

Hes·so Haute Ecole Spécialisée de Suisse occidentale Fachhochschule Westschweiz University of Applied Sciences Western Switzerland

Cuda Practical Guide

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

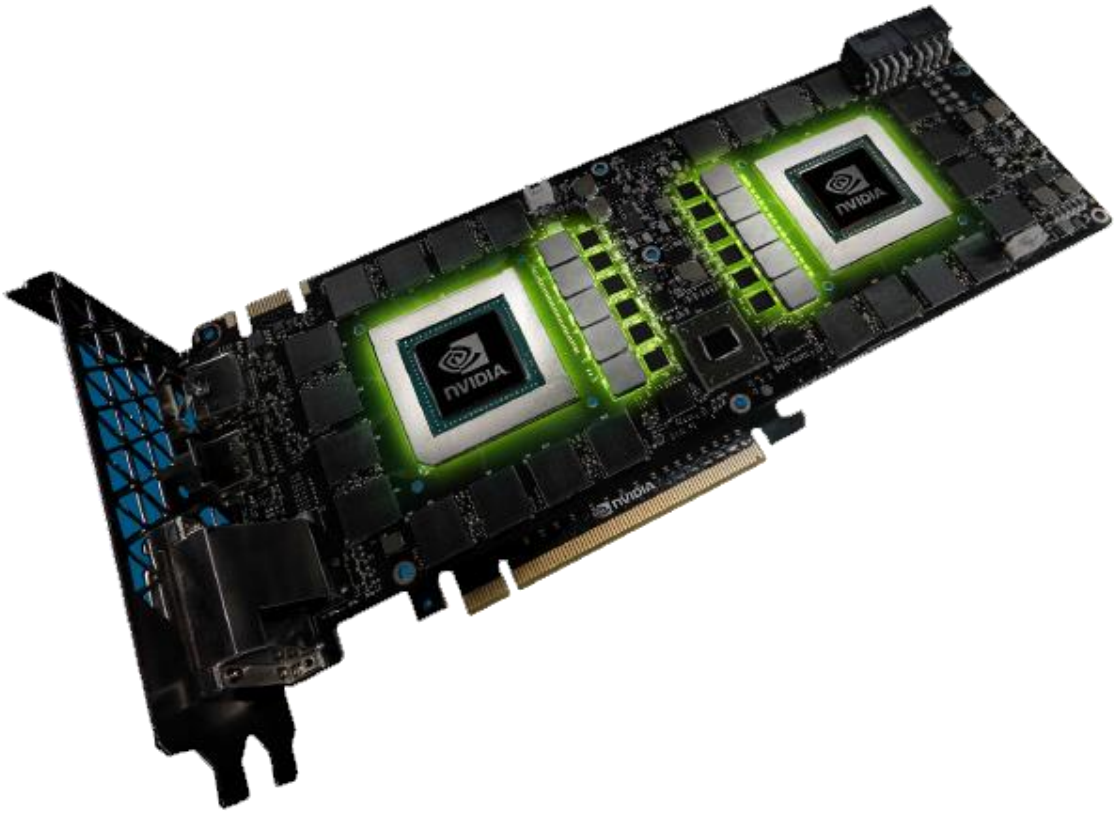


Table des matières

1 Memory Management (MM)

1.1 Principe

Pour écrire un code cuda, il faut coder deux parties

- Partie Host
- Partie Device

et réaliser du *memoryManagment* (MM) pour transférer les données entre ces deux parties et plus généralement entre

- le host et le device
- le device et lui-même
- le device et d'autre device

Le MM n'est utilisé que pour la gestion de la global memory (GM)

1.2 Syntaxe : Prototypes

L'utilisation du MemoryManagment (MM) et des méthodes qui vont suivre requière le fichier header cuda suivant :

- cuda_runtime.h

Allocation :

```
cudaError_t cudaMalloc(void** ptrDev, size_t size)
```

Libération :

```
cudaError_t cudaFree(void* ptrDev)
```

Copie :

```
cudaError_t cudaMemcpy(void* ptrDest,const void* ptrSrc, size_t count,
                       enum cudaMemcpyKind kind)
```

Moyen mnémotechnique :

En langage C l'opérateur d'affectation est dirigé de droite dans la gauche.

$$\overleftarrow{a = b}$$

La variable de droite remplit la variable de gauche. Avec *cudaMemcpy* le transfert est dirigé vers le même sens, de droite vers la gauche :

$$\text{cudaMemcpy } \overleftarrow{a, b}, \ldots$$

Initialisation :

```
cudaError_t cudaMemset (void *devPtr, int value, size_t count)
```

Il est vivement conseillé d'utiliser cette méthode pour mettre à zéro tout la zone mémoire GPU alloué avec un *cudaMalloc*. Vous pourriez avoir de mauvaise surprise entre 2 lancements successifs de la même application. En effet, la zone mémoire GPU pourrait contenir le contenu du lancement précèdent de l'application, et masqué des bugs pendant la phase de développement!

1.3 Syntaxe : Exemples

```
int n= …;
float* ptrData= new float[n];
float* ptrDevData= NULL;
size_t size= n*sizeof(float);
fill(ptrData);
HANDLE_ERROR(cudaMalloc((void**) &ptrDevData, size));

int initialValue = 0; //valeur en octet
HANDLE_ERROR(cudaMemset (ptrDevData,initialValue, size));

//Copie Host->Device
HANDLE_ERROR(cudaMemcpy(ptrDevData,ptrData,size,cudaMemcpyHostToDevice));
//Lancement du kernel

//Copie Device -> Host
HANDLE_ERROR(cudaMemcpy(ptrData, ptrDevData,size,cudaMemcpyDeviceToHost));
```

```
//Copie Device -> Device
float* ptrDevCopy= NULL;
HANDLE_ERROR(cudaMalloc((void**) &ptrDevCopy, size));
HANDLE_ERROR(cudaMemcpy(ptrDevCopy,ptrDevData,size,cudaMemcpyDeviceToDevice));
//Lancement du kernel

//Libération memoire
HANDLE_ERROR(cudaFree(ptrDevData));
HANDLE_ERROR(cudaFree(ptrDevCopy));
delete[] ptrData;
```

1.4 Détection d'erreur

Toutes les méthodes de MemoryManagment peuvent produire une erreur.

La macro **HANDLE_ERROR** permet de récupérer et d'afficher les erreurs dans la console.

Include :

```
#include "cudaTools.h"
```

Conseil :

> Prenez l'habitude de systématiquement entouré vos commande de MemoryManagment de la macro **HANDLE_ERROR**.
>
> Voir chapitre *CudaTools*.

1.5 Type simple et objet sans pointeur

Avant de connaitre le memory managmnet, on passait déjà des éléments entre le host et le device en utilisant les paramètres d'entrées du kernel :

```
myKernel<<<dg,db>>>(n,Banane) ;
```

Dans ce cas, les éléments sont copier ***octet par octet***. Si un objet occupe un espace continu en mémoire, le système pourra transférer l'objet sans problème entre le host et le device. Par contre, si l'objet contient un pointeur, c'est l'adresse du pointeur qui sera copié et non l'objet pointé !

Par exemple, on ne peut transférer un tableau en utilisant les paramètres d'entrée du kernel, car un tableau est un pointeur, et dans ce cas, ce serait le pointeur qui serait copié entre le host et le device, et non le contenu du tableau.

Dès qu'il y a des pointeurs, il faut donc utiliser les techniques de MM.

Par ailleurs, rappelons qu'en sortie un kernel a toujours *void* comme type de retour, car l'appel d'un kernel est ***assynchrone*** !

2 Gestion des erreurs

2.1 Memory Managment

Lorsque le Memory Managment de Cuda génère une erreur, on peut l'afficher dans la console grâce à la macro

HANDLE_ERROR

se trouvant dans le header

```
#include "cudaTools.h"
```

Voir fiche Memory Managment.

2.2 Erreur Cuda

On peut parfois récupérer certaine erreur cuda avec la méthode

```
Device::lastCudaError(const char* message)
```

Conseil :

Cette méthode ne semble pas couter de temps à l'exécution. Donc il est vivement conseillé de toujours utiliser cette méthode :

- avant/après chaque appel de ***kernel***
- après tout bloque de ***MemoryManagement***

Exemple *Kernel*

```
#include "Device.h"

void launchKernel(void)
    {
    …
    Device ::lastCudaError("KernelExemple (before)");
    kernelExemple<<<dg,db>>>(…);
    Device ::lastCudaError("KernelExemple (after)");
    …
    }
```

Exemple *MemoryManagement*

```
#include "Device.h"

void memoryManagment(void)
    {
    …
    HANDLE_ERROR(cudaMalloc(…));
    …
    Device ::lastCudaError("MemoryManagment (after)");
    }
```

2.3 Erreur d'algorithme

Afin de pouvoir détecter des erreurs dans notre algorithme, on peut choisir d'afficher dans la console des valeurs à l'aide de la procédure **printf**.

Syntaxe :

```
#include <stdio.h>
#include "Device.h"

___global___ void kernelExemple(….)
    {
    int tid=…;
    printf("hello %i", tid);
    …
    }

void launchKernel(void)
    {
    …
    kernelExemple<<<dg,db>>>(…);
    Device ::synchronize()); //pour le printf
    …
    }
```

Conseil :

Dans le cas ou le kernel travail sur une image, il est vivement conseillé de travailler avec un jeu de donné petit (par exemple 8x8), pour que la durée du kernel soit petite. Sinon il y a un risque d'arrêt des drivers graphiques.

Rappel :

Par défaut, après 2 secondes, si le kernel ne s'est pas terminé, l'OS tue les drivers de l'écran et les relances.

Attention :

Cette fonctionnalité est disponible que depuis l'architecture Nvidia *sm20*!

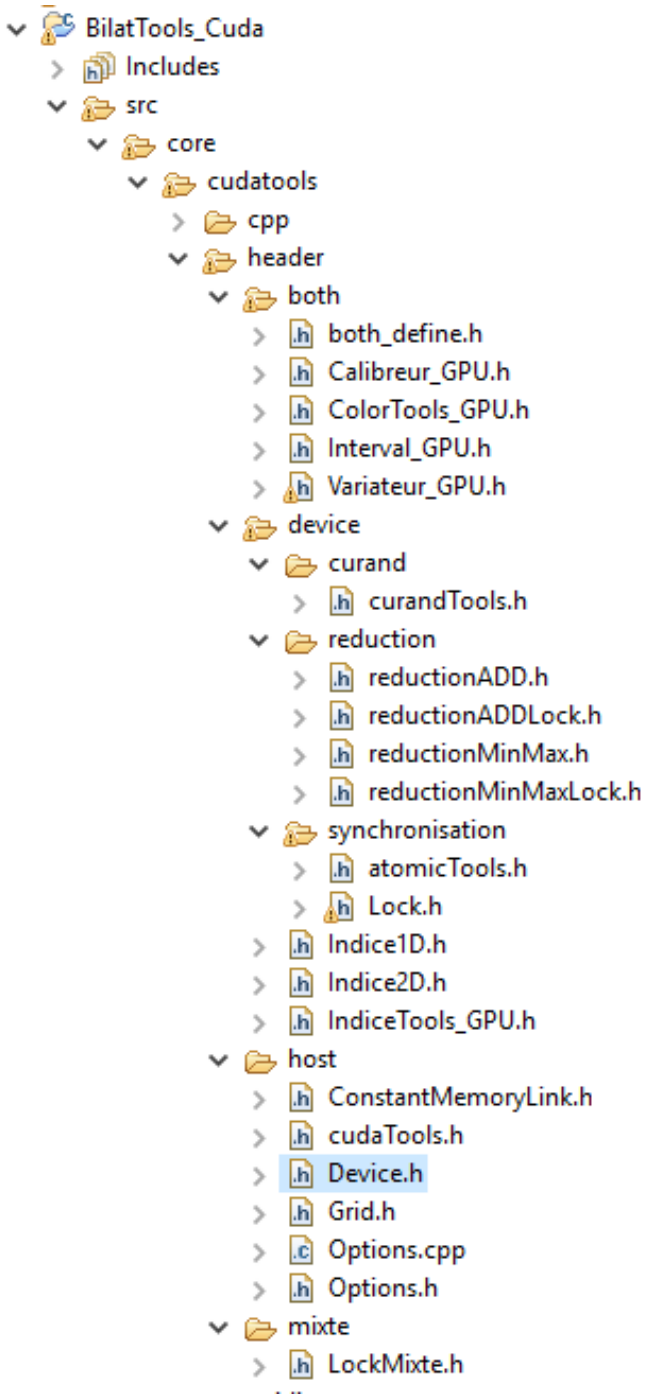
Option de compilation :

```
-arch=sm_20
```

3 CudaTools

3.1 Principe

Afin de facilité l'apprentissage et le codage avec Cuda, un certain nombre d'outils (*tools*) ont été développé. Ces outils ne se trouve donc pas dans Cuda mais dans le workspace fournit dans le cadre du cours. Les codes sources se trouvent dans le working set "*Tools*" dans le projet "*BilatToolsCuda*".



3.2 Classe Device

La classe *Device* permet de gérer tout ce qui concerne un ou plusieurs device. Elle se trouve dans le fichier header

```
#include "Device.h"
```

Exemple *Print*

```
Device::printAll();
Device::printAllSimple();
Device::printCurrent() ;

int deviceId=0 ;
Device::print(deviceId) ;
```

Exemple *LoadDriver*

Pour "pré charger" en mémoire les drivers *Cuda*, utilisez

```
Device::loadCudaDriver(i); //charge driver device i in [0,n[
Device::loadCudaDriverAll();
```

Très utile pour fiabiliser du *benchmraking* !

Exemple

Il est laissé au lecteur de découvrir les autres méthodes de la classe Device.

3.3 Classe Indice2D

La classe *Indice2D* se trouvant dans le fichier header

```
#include "Indice2D.h"
```

Cette classe fournit tous les outils pour utiliser le pattern d'entrelacement dans le cas d'une Grille 2D.

Exemple

```
int tid=Indice2D::tid();            // global
int tid=Indice2D::tidLocal();       // de block
int nbThread=Indice2D::nbThread();
```

Note

Dans le cas d'une grille 1D, utiliset la classe *Indice1D*

3.4 CudaTools

Le fichier header

```
#include "cudaTools.h"
```

est sans doute l'outil que vous allez le plus souvent utiliser. En effet il contient la macro

HANDLE_ERROR

Cette macro permet d'afficher l'erreur et l'endroit où elle s'est produite.

```
HANDLE_ERROR(xxx); //ou xxx est un memory managment
```

4 Chronométrage Cuda (Event)

4.1 Principe

Objectif :

On souhaite chronométrer précisément le temps d'exécution d'un kernel.

Stratégie :

On a deux stratégies :

- Chronométrer avec le CPU
- Chronométrer avec le GPU.

Le chronomètre GPU est bien plus précis, mais plus lourd à mettre en place.

4.2 Chronomètre CPU

Syntaxe :

```
#include "cudaTools.h" //for HANDLE_ERROR
#include "Chrono.h"
#include <iostream>
using std::cout;
using std::endl;

__global__ static void myKernel (void);

void exemple(void)
    {
    Chrono chrono;
    myKernel<<<blockPerGrid,threadPerBlock>>>(); // Asynchrone
    Device::synchronize(); //barriere de synchronisation
    chrono.stop();
    cout << "ElapseTime" << chrono.getDeltaTime()<< " (s)" << endl;
    }
```

```
__global__ void myKernel (void)
    {
    //doSomethingGPU
    }
```

Note :

Si on oublie la barrière explicite de synchronisation, le chrono affiche un chiffre très petit, ce qui prouve que l'appel d'un kernel est asynchrone.

4.3 Chronomètre GPU

Syntaxe :

```
#include "cudaTools.h" //for HANDLE_ERROR
#include <iostream>

using std::cout;
using std::endl;

__global__ static void myKernel (void);

void exemple(void)
    {
    cudaEvent_t start;
    cudaEvent_t stop;
    float elapseTimeMS;
    cudaStream_t idStream=0; //valeur par defaut(not explain here)

    HANDLE_ERROR(cudaEventCreate(&start));
    HANDLE_ERROR(cudaEventCreate(&stop));
    HANDLE_ERROR(cudaEventRecord(start, idStream));

    myKernel<<<blockPerGrid,threadPerBlock>>>(); // Asynchrone

    HANDLE_ERROR(cudaEventRecord(stop, idStream));
    HANDLE_ERROR(cudaEventSynchronize(stop));
    HANDLE_ERROR(cudaEventElapsedTime(&elapseTimeMS, start, stop));
    HANDLE_ERROR(cudaEventDestroy(start));
    HANDLE_ERROR(cudaEventDestroy(stop));
```

```
    cout << "ElapseTime<<" << elapseTimeMS << " (ms)" << endl;
    }

__global__ void myKernel (void)
    {
    //doSomethingGPU
    }
```

Principe :

On utilise l'objet

cudaEvent_t

celui-ci permet de stocker un "temps système". On le crée avec la méthode

cudaEventCreate

Le temps système est attrapé avec la méthode

cudaEventRecord

En jouant avec deux event (start et stop) on peux mesurer un temps écoulé grâce à la méthode

cudaEventElapsedTime

Les events se détruisent avec

cudaEventDestroy

5 Monitoring

5.1 Stratégies

Utiliser un profiler graphique ou un outil de monitoring en ligne de commande.

5.2 Ligne de commande

Syntaxe

Sur les GPU Tesla et Quadro haut de gamme, l'utilitaire ***nvidia-smi*** permet de monitorer les GPUs du système :

```
nvidia-smi
```

ou en boucle avec l'options *-l*

```
nvidia-smi -l
nvidia-smi --loop=1
```

ou pour plus de détails :

```
nvidia-smi -q
```

Pour se concentrer sur la température, le taux d'utilisation et la mémoire

```
nvidia-smi -q --display=TEMPERATURE,UTILIZATION,MEMORY
```

Pour de l'aide

```
nvidia-smi -h
```

Utilité

Dans le cadre d'un parallélisme étendu sur plusieurs GPU, cela permet en monitorant la ***température*** ou **l'utilisation** de vérifier que les kernels sont bien répartis sur plusieurs GPU. Un *taux d'utilisation* de 0% serait alors suspect, tout comme une *température* constante pas plus haute que celle à l'état Idl (au repos)

Note

Le taux d'utilisation des GPU avec l'API *BilatImageCuda* est toujours assez bas. En effet, le rendu prend du temps ! Comme la zone mémoire

est partagée entre OpenGL et Cuda, aucun kernel cuda ne peut calculer pendant qu'OpenGL lit la zone mémoire contenant les données à rendre. Avec le TP *Heat Transfert* et les images fantômes, la situation devrait changer, avec les TP sans GUI, le taux devrait encore grimper ! Quel valeur max arrivez-vous obtenir !!

Exemples

Exemple 1 : *nvidia-smi*

```
+------------------------------------------------------------------------------+
| NVIDIA-SMI 4.304.64   Driver Version: 304.64                                 |
|-------------------------------+----------------------+----------------------+
| GPU  Name                     | Bus-Id        Disp.  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage         | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Quadro 7000              | 0000:16:00.0     On  |                  Off |
| 30%  58C   P0   82W / 250W    |   4%  223MB / 6143MB |    31%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Quadro 7000              | 0000:15:00.0     Off |                  Off |
| 30%  54C   P0   89W / 250W    |   2%  105MB / 6143MB |    16%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla M2090              | 0000:0C:00.0     Off |                    0 |
| N/A  47C   P0   76W / 225W    |   2%  110MB / 5375MB |     0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla M2090              | 0000:0B:00.0     Off |                    0 |
| N/A  47C   P12  29W / 225W    |   0%   10MB / 5375MB |     0%      Default |
+-------------------------------+----------------------+----------------------+
|   4  Tesla M2090              | 0000:88:00.0     Off |                    0 |
| N/A  47C   P12  31W / 225W    |   0%   10MB / 5375MB |     0%      Default |
+-------------------------------+----------------------+----------------------+
|   5  Tesla M2090              | 0000:87:00.0     Off |                    0 |
| N/A  47C   P12  30W / 225W    |   0%   10MB / 5375MB |     0%      Default |
+-------------------------------+----------------------+----------------------+

+------------------------------------------------------------------------------+
| Compute processes:                                               GPU Memory  |
|  GPU       PID  Process name                                     Usage       |
|==============================================================================|
|    0     16255  ...Targets/TutoCuda_Image/bin/TutoCuda_Image64.run   370MB  |
|    1     16255  ...Targets/TutoCuda_Image/bin/TutoCuda_Image64.run   370MB  |
|    2     16255  ...Targets/TutoCuda_Image/bin/TutoCuda_Image64.run   370MB  |
+------------------------------------------------------------------------------+
```

Exemple 2 : *nvidia-smi -q --display=TEMPERATURE,UTILIZATION,MEMORY*

```
Attached GPUs                 : 6

GPU 0000:16:00.0
    Memory Usage
        Total                 : 6143 MB
        Used                  : 47 MB
        Free                  : 6096 MB
    Utilization
        Gpu                   : 0 %
        Memory                : 4 %
    Temperature
        Gpu                   : 53 C

GPU 0000:15:00.0
    Memory Usage
        Total                 : 6143 MB
        Used                  : 12 MB
        Free                  : 6131 MB
    Utilization
        Gpu                   : 0 %
        Memory                : 0 %
    Temperature
        Gpu                   : 49 C

...
```

6 Constant Memory (CM)

6.1 Présentation

Définition

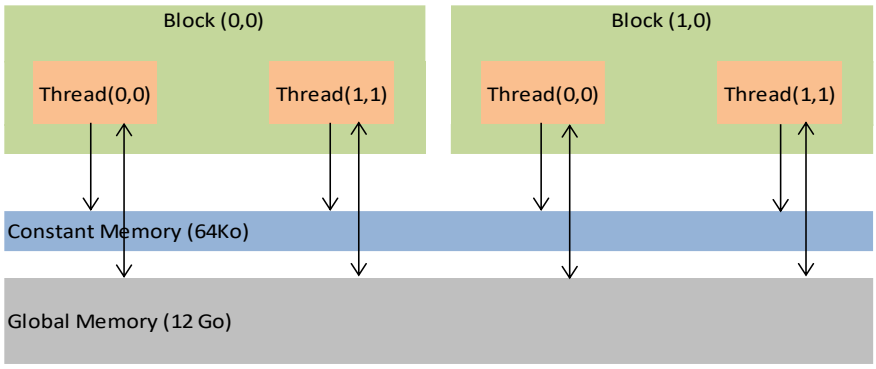
Cuda met à notre disposition un type de mémoire particulier pour des données constantes au cours du temps : la

constante Memory (CM!)

Elle n'est pas en quantité infini. Pour les cartes graphiques du type *Maxwell* la quantité de *constante Memory CM* est de l'ordre de

64 ko

Il s'agit d'une mémoire **read-only**. Il y a **une instance** pour toute l'application.



Performance

Vue son aspect *read-only*, le compilateur peut effectuer des opérations de mise en cache agressives. Les performances de la *constante memory* sont plutôt intéressantes, notamment en accès de type « Broadcasting », ie lorsque tous les threads accèdent en même temps à une même donnée.

Notes

(N1) Est fonctionnelle pour les types simples comme pour les objets.

(N2) Le contenu de la CM peut changer entre l'appel de deux kernels ! Par contre son contenu est forcément de type read-only é l'intérieur d'un kernel !

6.2 Syntaxe (mono fichier)

Déclaration

```
// Déclaration Constante globale
#define LENGTH 2      //ou const int LENGTH =2;
__constant__ float TAB_CM[LENGTH];
```

device.cu

Contraintes :

(C1) **TAB_CM** est déclaré **globalement à un fichier** .

Sa portée est de fichier.
Elle ne peut être importée depuis un autre fichier.

(C2) Cette déclaration doit être dans le même fichier ***.cu*** qui l'utilise.

Remplissage

On utilise du memory managment. Attention le classique

```
cudaMemcpy
```

ne fonctionne que sur la GM (Global Memory). Il ne peut donc être utilisé ici. Son pendant pour la CM (constant Memory) est

```
cudaMemcpyToSymbol
```

Exemple *Mono Fichier*

```
// Déclaration Constante globale
#define LENGTH 2
__constant__ float TAB_CM[LENGTH];

// Hyp : meme fichier que la déclaration de TAB_CM
__host__ void fillCM(…)
      {
      float tabValue[LENGTH]={1,2};
      uploadGPU (tabValue);
      }

// Hyp : meme fichier que la déclaration de TAB_CM
__host__ void uploadGPU (float* tabValue)
      {
      size_t size= LENGTH *sizeof(float);
      int offset=0;

      HANDLE_ERROR(cudaMemcpyToSymbol(TAB_CM,tabValue,size,
                        offset,cudaMemcpyHostToDevice));
      }

// Hyp : meme fichier que la déclaration de TAB_CM
__global__ void myKernel(…)
      {
      … TAB_CM …
      }
```

Device&Host.cu

Utilisation

TAB_CM peut être vu comme un pointeur et utilisé comme tel dans un kernel.

Exemple

Un code utilisant de la GM

```
__global__ void myKernel(float* tabDevGM, int n)
      {
      work(tabDevGM,n); // use GM
      }

__device__ void work(float* tabDev, int n)
      {
      … // read only
      }
```

Le même code utilisant de la CM. Notez que l'utilisation de la méthode work est la même que ci-dessus, pratique!

```
// Déclaration Constante globale
#define LENGTH 2
__constant__ float TAB_CM[LENGTH];

__global__ void myKernel(int n)
      {
      work(TAB_CM, LENGTH);   // use SM
      }

// Le meme que ci-dessus utilisant de la GM
__device__ void work(float* tabDev, int n)
      {
      … // read only
      }
```

6.3 Syntaxe (multifichier)

Problème

Dans le cas standard d'une approche multifichiers, où on a séparé le code *Host* et le code *Device*, l'utilisation de la CM est plus complexe. En effet, **TAB_CM** n'est pas externalisable comme le serait une constante du langage C standard. En effet :

```
extern __constant__ float TAB_CM[];
```

ne compile pas en cuda !

Solution

Deux solutions existent aux problèmes :

- Technique des *Symboles Cuda*
- Techniques du *Service externe*

Techniques du service externe

TAB_CM n'étant utilisable que dans le fichier *device.cu* dans lequel il a été déclaré, l'appel de ***cudaMemcpyToSymbol*** pour transférer les données entre le host et la CM (device) ne peut se faire que dans ce fichier *device.cu*

L'idée est ici de fournir une méthode effectuant cet *upload* en CM dans une méthode, puis d'exporter cette méthode dans la partie host du code. L'exemple mono-fichier a déjà été architecturé sous cette forme pour pouvoir utilisez ce pattern du « service externe »

Exemple *Device Side*

Comme *cudaMemcpyToSymbol* doit connaitre *TAB_CM* pour travailler, on place la méthode *uploadGPU* dans le code côté device, bien qu'elle sera executée par le host.

```
// Déclaration Constante globale
#define LENGTH 2
__constant__ float TAB_CM[LENGTH];

__host__ void uploadGPU(float* tabValue)
      {
      size_t size= LENGTH *sizeof(float);
      int offset=0;

      HANDLE_ERROR(cudaMemcpyToSymbol(TAB_CM,tabValue,
                  size,offset,cudaMemcpyHostToDevice));
      }

__global__ void myKernel(int n)
      {
      work(TAB_CM, LENGTH);   // use SM
      }

__device__ void work(float* tabDev, int n)
      {
      … // read only
      }
```

device.cu

puis coté host on declare extern cette méthode *uploadGPU*, afin de pouvoir l'utiliser.

Exemple *Host Side*

```
extern __global__ void myKernel(…) ;
extern __host__ void uploadGPU (…) ;



__host__ void fillCM(…)
      {
      float tabValue[LENGTH]={1,2};
      uploadGPU(tabValue);
      }

__host__ void runGPU(…)
      {
      …
      fillCM(..);// dans un constructeur normalement !
      …
      myKernel<<dg,db>>(…) ;
      …
      }
```

host.cu

6.4 Cuda et Symbole

Définition

Une variable ou constante globale déclaré dans un *.cu* est considéré en *Cuda* comme un **symbole.**

Pour mettre un contenu dans un symbole on a deux possibilités :

(P1) Utiliser la méthode ***cudaMemcopyToSymbol*** (utilisable en mono fichier)

(P2) Obtenir l'adresse du symbol avec ***CudaGetSymbolAddress*** puis utiliser un *cudaMemcopy* standard (utile en multi fichier)

Note :

Une déclaration de procédure *__device__* serait aussi un symbole !

6.5 Symbole et CM

On considère l'objet CMLink suivant

```
struct CMLink
    {
        void** ptrDevTabCM;   // void** pour la généricité
        int n;                // nb case du tableau
        size_t sizeAll;       // #octets de tout le tab
    };
```

L'idée est ici de représenter ***TAB_CM*** par cet objet. Cet objet *CMLink* sera créer coté

Device

et utilisez coté

Host

Il s'agit d'une autre solution pour utiliser la CM dans une application ou le code *Host* a été séparé du code *Device* dans plusieurs fichiers différents.

Device Side *Symbole to Adresse*

```
#include "CMLink.h"

CMLink wrappingCM(void)
    {
    float* ptrTabCM;
    size_t size = LENGTH * sizeof(float);

    HANDLE_ERROR(cudaGetSymbolAddress((void**)&ptrTabCM, TAB_CM));

    CMLink cmLink ={(void**) ptrTabCM, LENGTH, size};

    return cmLink;
     }
```

device.cu

Host Side *Use Adress*

```
#include "CMLink.h"

// Hyp : TAB_CM déclaré dans un autre fichier
// On récupère l'adesse de ce symbole,
// puis on travaille comme avec une adresse standard
__host__ void fillCM(…)
      {
      // Recupération de la constantMemory CM du device
     CMLink cmLink = wrappingCM();
     float* ptrDevTabCM = (float*)cmLink.ptrDevTabCM;
     size_t size = cmLink.size; // octet
      int n = cmLink.n;

      // Création et remplissage coté host
     float* ptrTabValue=new float[n] ;
     fill(ptrTabValue,n);// fill with data ptrtabValue

     // transfert Host->Device
      HANDLE_ERROR(cudaMemcpy(ptrDevTabCM
          ptrTabValue,sizeALL,cudaMemcpyHostToDevice));

     delete[] ptrTabValue ;
      }
```

host.cu

7 Texture avec Cuda

7.1 Principe

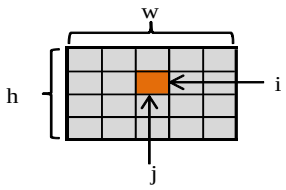
Définition : *texture*

Une texture est une vue d'une zone mémoire du GPU alloué avec par exemple un *cudaMalloc*. La texture est une manière de voir différemment cette zone mémoire.

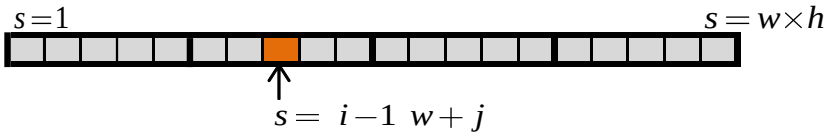
7.1.1 Exemple

Hypothèse :

On a une image finie $1,w\ x\ 1,h$:



stockée sur le GPU sous la forme row-major linéarisé :



E1) Texture wrapping (masque)

On a une image finie row-major linéarisé stockée sur le GPU. On peut déposer un "masque" de lecture sur cette zone mémoire sous la forme d'une texture. Grace à ce masque, on peux accéder aux pixels de l'image avec un couple i,j d'indice, avec $i\in 1,h$ et $j\in 1,w$. Pratique!

Intérêt :

S'affranchir de la problématique du row-major linéarisé, c.à.d. s'affranchir des problématiques de conversion d'indice $2D\rightarrow 1D$

$$i,j\ \rightarrow s$$

E2) Interpolation

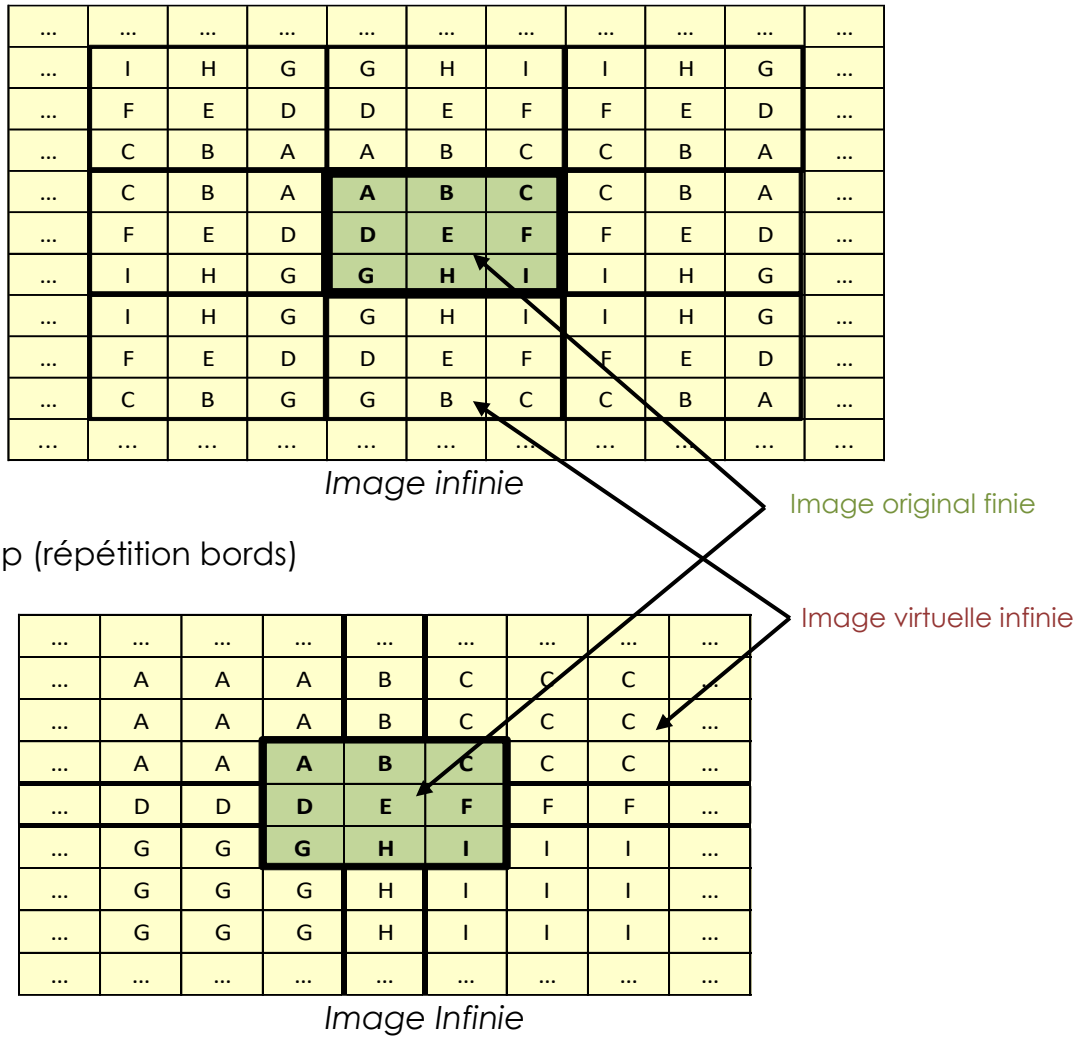
On a une image finie row-major linéarisé stockée sur le GPU. Si on map cette zone mémoire avec une texture, l'image possède une "infinité" de pixel. Par exemple, le pixel (1.6,3.8) n'existe pas dans l'image d'origine, mais dans la texture, oui! Ce pixel sera calculé selon l'un des 2 modes suivants :

- Interpolation linéaire des texels en coordonné entières les plus proches
- Par le texel en coordonnée entières le plus proche.

E3) Mode d'adressage

On a une image **finie** $1,w\ x\ 1,h$ row-major linéarisé stocké sur le GPU. La texture permet d'accéder ici aux pixels se trouvant en dehors de l'image, selon un des modes d'adressages suivant :

- Mirror



- Clamp (répétition bords)

E4) Texture et normalisation

Définition : *normalisation*

Soit x une variable appartenant à a,b . Normaliser x équivaut à effectuer une transformation affine de telle sorte que le spectre de valeur atterrisse dans $0,1$

Deux types :

Il existe deux sortes de normalisation

- Normalisation sur les indices d'accès à la texture.
- Normalisation de la "sortie" de la texture.

Normalisation des indices :

On a une image finie $1,w\ x\ 1,h$ row-major linéarisé stockée sur le GPU. On peut déposer un "masque" de lecture sur cette zone mémoire sous la forme d'une texture.

Accès en mode standard :

Grace à ce masque, on peux accéder aux pixels de l'image avec un couple i,j d'indice, avec

- $i \in 1,h$
- $j \in 1,w$

Accès en mode Normalisé

En mode normalisé, la paire d'indice i,j ne dot plus varier sur les dimensions de l'image, mais sur l'intervalle $0,1$:

- $i \in 0,1$
- $j \in 0,1$

Normalisation de la sortie :

Lecture en mode standard :

En mode standard, on obtient le contenu même de la mémoire. Il n'y aucune transformation entre le contenu de la mémoire et ce que l'on reçoit avec l'objet texture.

Lecture en mode normalisé :

En mode normalisé, avec l'objet texture on obtient le contenu normalisé dans $0,1$ de la zone mémoire mappée.

E5) Combinaisons

Les propriétés d'une texture ont été présenté séparément. Cependant en pratique elles se combinent toutes! Que l'on n'utilise ou pas l'interpolation, cette fonctionnalité est disponible. Que l'on n'utilises ou pas le mode d'adressage (miroir, clamp), cette fonctionnalité est disponible.

7.1.2 Propriétés des textures

En fait, les 3 notions

- Texture wrapping
- Interpolation
- Mode d'adressage

sont toujours les 3 disponibles. A la création ont peux cependant choisir :

- Les propriétés du mode d'interpolation (linéaire, plus proche).
- Les propriétés du mode d'adressage (miroir et clamp).
- La normalisation ou non des indices.
- La normalisation ou non de la sortie de la texture.

La notions de texture wrapping est quand a elle un axiome même de la texture et est toujours active. En particulier, le masque de lecture pratique en indice 2D (entier) est toujours utilisable. Comme l'interpolation est toujours présente dans une texture, le masque de lecture en indice 2D (*float*) est lui aussi toujours disponible.

7.1.3 Bindings

On à une zone mémoire GPU créé avec un *cudaMalloc*, une texture est un "objet" qui va binder (mapper) cette zone mémoire pour l'utiliser. Après utilisation, la texture doit relâcher cette zone mémoire en faisant un unbind (unmap).

En conséquence, une zone mémoire a (en tout cas) 2 états :

- bind
- unbind

7.1.4 Ecriture dans les textures

Les textures sont des sortes de masques offrant un certain nombre de fonctionnalités intéressantes, **en lecture**! On ne peut pas écrire dans une texture à l'aide de l'objet texture. On peut cependant écrire dans la zone mémoire wrapé par la texture grâce au pointeur sur le tableau original 1D. Ceci est vrai quelque soit l'état

- bind
- unbind

de la mémoire. Surprenant, mais fonctionnel!

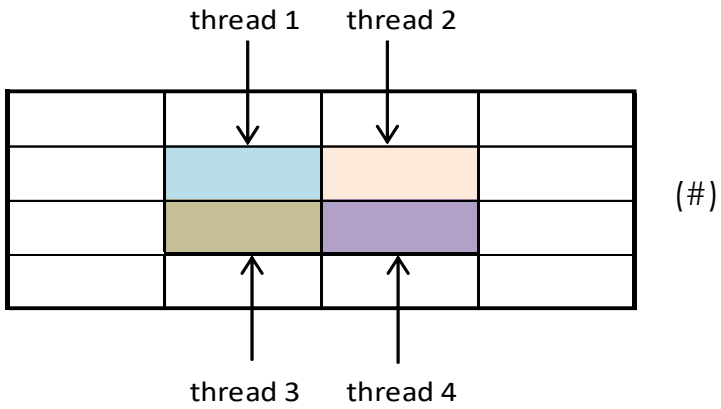
7.1.5 Texture 1D et 2D

Les textures sont souvent utilisé lorsque l'on travail avec des images. On utilise dans ce cas tout naturellement des textures 2D. Il existe cependant des textures 1D et 3D. Il n'y a aucune différence de performances entre-elles. Seule la logique d'accès aux données varie (c.f. paragraphe à venir syntaxe).

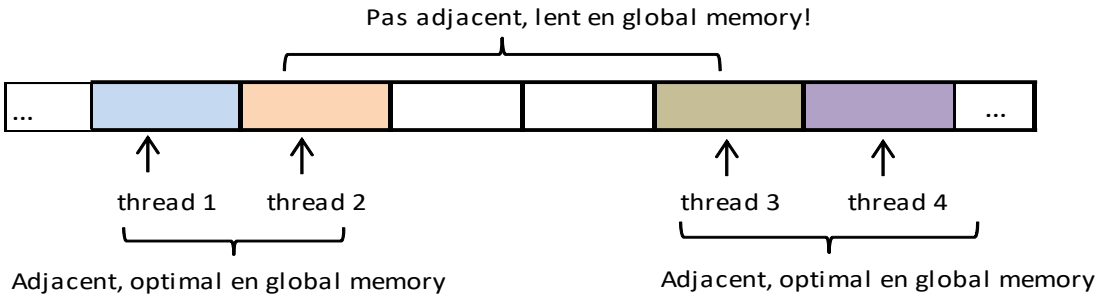
7.1.6 Performance

En théorie, les textures offrent de meilleures performances que la *global memory*, lorsque des threads voisins utilisent des "pixels" géographiquement proche comme ceci :

Vue logique (vue texture 2D) :



Vue Row-major linéarisé :



En global memory, l'accès par des threads voisins d'un *warp* à des adresses mémoire non adjacentes coûte beaucoup de transactions, donc de temps. En global memory, l'accès aux données jaune et bleu par les threads voisins thread 2 et thread 3 est lent. Avec les textures on n'a pas ce problème! Les textures amènent la performance sur le voisinage vertical. Le pattern d'entrelacement faisait déjà très bien le travail sur le voisinage horizontal.

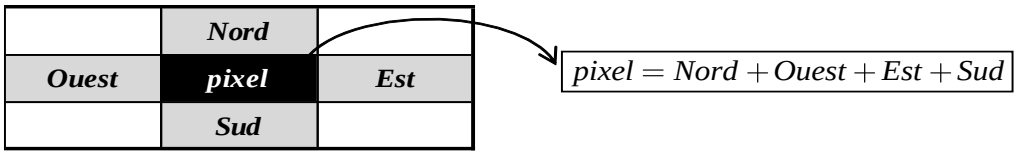
Observation :

En pratique, est-il possible de parcourir facilement une image avec le pattern d'accès (#) optimum pour les textures? Avis de l'auteur : pas trop!

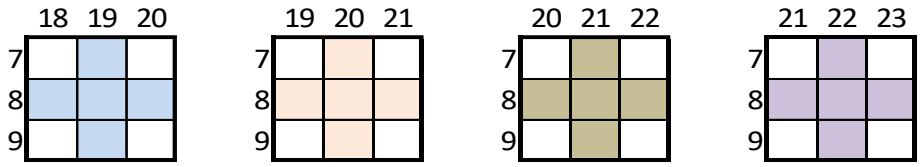
L'exemple ci-dessous semble être un scénario idéal pour les textures. Néanmoins le **pattern d'entrelacement** fait une nouvelle fois des merveilles.

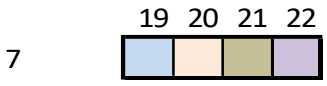
Exemple : *Exploitation spatial de données*

Algo :

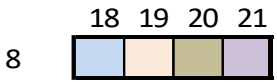


A cause du SIMD des *GPU*, les 4 threads accède au même cycle d'horloge au pixel nord :

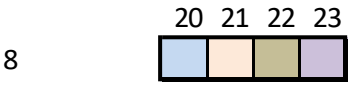




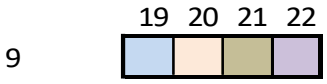
Puis accèdent en même temps au pixel Ouest :



Puis au pixel Est :



Puis au pixel Sud :



On laisse le soin au lecteur de vérifier expérimentalement que les textures n'amène pas de meilleurs performances que le pattern d'entrelacement. Par contre, le mode d'adressage (*clamp*) simplifie énormément la gestion des bords. Les textures n'amène pas forcément de meilleurs performances, mais elles apportent une simplification du code.

7.2 Syntaxe

Contraintes :

Les objets textures

- Ne peuvent pas être transmis aux méthodes.

Doivent impérativement être déclarés comme des variables globales en tête du ficher *.cu* contenant le kernel!

- Zone mémoire mappé ne doit pas être nécessairement une puissance de 2 (supérieur ou égal à 8). Par contre si tel était le cas, les performances pourraient être meilleures.

Syntaxe simplifié

Par défaut :

- Le mode d'interpolation est du type "texel le plus proche".
- Le mode d'adressage est de type "clamp".
- Les coordonnées de texture i, j ne sont pas normalisées.
- La "sortie" de la texture n'est pas normalisée.

La syntaxe simplifiée est :

Texture 2D :

```
texture<float, 2> textureRef;
```

Texture 1D :

```
texture<float, 1> textureRef;
```

Syntaxe longue :

Texture 2D :

```
texture<float,2,cudaReadModeElementType> textureRef;
textureRef.normalized=false; //pour coordonnée texture (i,j)
textureRef.filterMode = cudaFilterModePoint;
textureRef.addressMode[0] = cudaAddressModeClamp;
textureRef.addressMode[1] = cudaAddressModeClamp;
```

Texture 1D :

```
texture<float,1,cudaReadModeElementType> textureRef;
textureRef.normalized=false; //pour coordonnée texture (i,j)
textureRef.filterMode= cudaFilterModePoint;
textureRef.addressMode[0] = cudaAddressModeClamp;
```

7.2.1 Customisation des propriétés des textures

Interpolation

Par défaut, le mode d'interpolation est le mode texel le plus proche.

Texture 2D :

```
texture<float,2> textureRef;

//version texel le plus proche
textureRef.filterMode = cudaFilterModePoint;

//version interpolation (pour des float uniquements!)
textureRef.filterMode = cudaFilterModeLinear;
```

Texture 1D :

```
texture<float,1> textureRef;

//version texel le plus proche
textureRef.filterMode = cudaFilterModePoint;

//version interpolation (pour des float uniquements!)
textureRef.filterMode = cudaFilterModeLinear;
```

Contrainte :

Pour une interpolation linéaire la texture doit être de type *float* ou *float3*.

Mode d'adressage

Texture 2D :

```
texture<float,2> textureRef;

//version miroir (coordonnées texture normalisé obligatoire)
```

```
textureRef.normalized=true;//pour coordonnée texture (i,j)
textureRef.filterMode[0] = cudaAddressModeMirror;
textureRef.filterMode[1] = cudaAddressModeMirror;

//version clamp
//textureRef.normalized=false; //par defaut
//textureRef.addressMode[0] = cudaAddressModeClamp;
//textureRef.addressMode[1] = cudaAddressModeClamp;
```

Texture 1D :

```
texture<float,1> textureRef;

//version miroir(coordonnées texture normalisé obligatoire)
textureRef.normalized=true; //pour coordonnée texture i
textureRef.addressMode[0] = cudaAddressModeMirror;

//version clamp
//textureRef.normalized=false;//pour coordonnée texture i
//textureRef.addressMode[0] = cudaAddressModeClamp;
```

Limitation :

Le mode adressage Miroir nécessite des coordonnées de textures (indices d'accès i et j) normalisé, c.à.d. dans $0,1$. Par contre le mode d'adressage clamp ne souffre pas de cette contrainte.

Normalisation

Normalisation des indices :

Texture 2D :

```
textureRef.normalized=true; //pour coordonnée texture (i,j)
```

Texture 1D :

```
textureRef.normalized=true; //pour coordonnée texture i
```

La valeur par défaut est *false*.

Normalisation de la sortie :

Texture 2D :

```
texture<short,2,cudaReadModeNormalizedFloat> textureRef;
```

Texture 1D :

```
texture<short,1,cudaReadModeNormalizedFloat> textureRef;
```

Limitation :

La normalisation de la sortie n'est valable que pour les textures du type "entier" :

- unsigned char, uchar4
- short, unsigned short, ushort4

7.2.2 Exemples

- Miroir et interpolation linéaire

```
texture<float,2> textureRef;

textureRef.cudaTextureFilterMode= cudaFilterModeLinear;

textureRef.normalized=true; //pour coordonnée texture (i,j)
textureRef.addressMode [0] = cudaAddressModeMirror;
textureRef.addressMode [1] = cudaAddressModeMirror;
```

Interpolation

miroir

- Clamp et interpolation par texel le plus proche

```
texture<float,2> textureRef;

textureRef.filterMode= cudaFilterModePoint;

textureRef.normalized=false; //ou true coordonnée texture (i,j)
textureRef.addressMode[0] = cudaAddressModeClamp;
textureRef.addressMode[1] = cudaAddressModeClamp;
```

Interpolation

clamp

7.2.3 Bind

Texture 2D :

```
size_t pitch = w * sizeof(float); //taille en octets d'une ligne
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTexture2D(NULL, textureRef,ptrDevImage,channelDesc,w,h,pitch);
```

Texture 1D :

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTexture(NULL, textureRef,ptrDevImage,channelDesc,size);
```

7.2.4 Unbind

Texture 1D, 2D et 3D :

```
cudaUnbindTexture(textureRef);
```

7.2.5 Lecture

Texture 2D :

```
float valueXY = tex2D(texture, x, y); //x et y float!
```

Notez que le paire d'indice (x,y) est en float. Le mode interpolation est toujours disponible sur une texture.

x varie sur la partie horizontal de l'image, càd,
$x \in 0, w$

y varie sur la partie vertical de l'image, càd,
$y \in 0, h$

Texture 1D :

```
float valueX = tex1D (texture, x) //x float!
```

Notez que l'indice est en float. Le mode interpolation est toujours disponible sur une texture.

ou

```
float valueX = tex1Dfetch(texture, x) //x int!
```

Notez que l'on a avantage a utiliser *tex1Dfetch* lorsque l'interpolation ne nous intéresse pas et que nos indices sont entier (évite une conversion de type int to float). Dans ce cas, plus performant?

7.2.6 Diagramme de séquence

Voici une séquence standard d'une utilisation d'une texture :

1) Allocation mémoire GPU (*cudaMalloc*)
2) Texture binding
3) Texture use
4) Texture unbinding
5) Déallocation mémoire GPU (*cudaFree*)

Evidemment, l'objet texture doit être créée comme variable global avant.

7.3 Exemple

Supposons que l'on doit implémenter l'algorithme suivant :

$$\text{imageInput} \xrightarrow[algo]{} \text{imageOutput}$$

L'*imageInput* sera mappé avec une texture ayant les propriétés suivantes :

- Clamp.
- Interpolation par texel proche.
- Pas de normalisation d'indice.
- Pas de normalisation de sortie.

```
//Déclaration

texture<float,2,cudaReadModeElementType> textureInput;
static __global__ void kernelUseTexture(float* ptrDevOutput,int w,int h)
static __host__ void work(float* ptrOut,int w,int h);

//Implémentation

void useHelloWorld(float* ptrIn,float* ptrOut,int w,int h)
    {
    float* ptrDevIn = NULL;
    float* ptrDevOut = NULL;

    size_t size = w * h * sizeof(float);
    HANDLE_ERROR(cudaMalloc((void**) &ptrDevOut,size);
    HANDLE_ERROR(cudaMemset(ptrDevOut, 0, size);
    HANDLE_ERROR(cudaMalloc((void**) &ptrDevIn, size));
    HANDLE_ERROR(cudaMemcpy(ptrDevIn, ptrIn,size,cudaMemcpyHostToDevice));

    //Configuration texture, valeurs par defaut !
    textureInput.addressMode[0] = cudaAddressModeClamp;  //par defaut
    textureInput.addressMode[1] = cudaAddressModeClamp;  //par defaut
    textureInput.filterMode = cudaFilterModePoint;       //par defaut
    textureInput.normalized = false; //coordonnée texture //par defaut

    size_t pitch = w * sizeof(float); //size ligne
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
    HANDLE_ERROR(cudaBindTexture2D(NULL,textureInput,ptrDevIn,channelDesc,w,h,pitch));
```

```
    dim3 grid = dim3(8, 8);       //disons
    dim3 block = dim3(16, 16, 1); //disons
    kernelUseTexture<<<grid,block>>>(ptrDevOut, w, h); //fill ptrDevOut
    HANDLE_ERROR(cudaUnbindTexture(textureInput));
    HANDLE_ERROR(cudaMemcpy(ptrOut,ptrDevOut,size,cudaMemcpyDeviceToHost));

     HANDLE_ERROR(cudaFree(ptrDevOut));
     HANDLE_ERROR(cudaFree(ptrDevIn));
    work(ptrOut,w,h);
    }

/**
 * Attention
 * L'image input mappé par la texture n'est pas passé comme paramètre
 * au kernel.
 * Rappelons que la texture est un variable globale (contrainte).
 */
__global__ void kernelUseTexture(float* ptrDevOutput, int w, int h)
    {
    float pixelI=…;
    float pixelJ=…;
    …
    float valueIJ= tex2D(textureInput,pixelJ,pixelI);
    …
    ptrDevOutput=…;
    }

__host__ void work(float* ptrOut,int w,int h)
    {
    //doSomething!
    }
```

8 Nombre aléatoire Cuda (Curand)

8.1 Mathématique et nombre aléatoire

Mathématiquement il n'est pas facile d'écrire un algorithme parfait générant des nombres aléatoire (pseudo, quasi). On peut d'ailleurs ce poser la question quelle est la différence entre ces deux sorte de nombre aléatoire. Si la formule est difficile en séquentielle, elle l'est encore plus en parallèle.

Il existe une multitude d'algorithmes. Le défi de ces algorithmes est de tirer des nombres aléatoires le plus "aléatoire" possible. Il est laissé au lecteur le choix du **générateur** à utiliser. A chaque algorithme correspond un générateur. En Cuda on a par exemple les générateurs suivant :

- **CURAND_RNG_PSEUDO_DEFAULT**
- **CURAND_RNG_PSEUDO_XORWOW**
- **CURAND_RNG_PSEUDO_MRG32K3A**
- **CURAND_RNG_PSEUDO_MTGP32**

8.2 Stratégie d'utilisation

On vous propose 2 stratégies :

(S1) Tirer un nombre et l'exploiter (directement)
(S2) Tirer un nombre et le ranger (tableau)

8.3 Tirer un nombre et l'exploiter

Principe

L'idée est ici de tirer les nombres aléatoires directement dans le kernel, là où on en a besoin. Chaque thread devra donc générer une séquence de nombre différents. Il ne faudra donc pas un générateur pour tout le kernel, mais un ensemble de générateurs : un part thread !

Analyse

Avantage : Pas de limitation sur la quantité de nombres générés.

Inconvénient : Aucune utilisation CPU possible!

Technique

Il faut lancer au minimum deux kernels :

- Un pour la création des générateurs
- Un pour l'utilisation des générateurs

Les Générateurs

Les générateurs seront stockés en GM dans un tableau de type

curandState*

de longueur *nbThread* (un générateur par thread).

Comme chaque générateur de ce tableau doit générer des nombres aléatoires différents, la création d générateur doit impérativement dépendre de

- *tid*
- *deviceId*

si l'on souhaite utiliser plusieurs GPU.

API Currand

Nvidia met à disposition l'API ***curand*** qui permet la création de nombre aléatoire sur GPU. Le principe est le suivant :

- Fabriquer des générateurs (un par thread)
- Utiliser les générateurs.

Deux header sont nécessaires :

- Dans un cpp : **curand.h**
- Dans un kernel : **curand_kernel.h**

Kernel indirect

Même si aucun kernel n'est apparant dans l'API curand, des kernels sont appelé indirectement par les fonctions de cette API.

8.3.1 Etape

Trois grandes étapes sont nécessaire

- Memory Management pour faire de la place pour les générateurs !

Puis :

- Lancement du kernel de création des générateurs
- Lancement du ou des kernles d'utilisation des générateurs

Type

Un générateur sera représenté par le type ***curandState****

8.3.2 Code : création des générateurs

```
#include <curand_kernel.h>
#include <limits.h>

// Each thread gets same seed , a different sequence number , no offset
__global__
void setup_kernel_rand(curandState* tabDevGenerator,int deviceId)
    {
    const int TID = Indice1D::tid();

    //Customisation du generator: Proposition (au lecteur de faire mieux)
    // Contrainte : Doit etre différent d'un GPU à l'autre
    int deltaSeed=deviceId* INT_MAX/10000;
    int deltaSequence=deviceId *100;
    int deltaOffset=deviceId *100;

    int seed=1234+deltaSeed;
    int sequenceNumber= TID +deltaSequence;
```

```
    int offset=deltaOffset;
    curand_init(seed, sequenceNumber, offset, & tabDevGenerator [tid]);
    }
```

Note :

Deux générateurs se distinguent lors de leur création par les 3 paramètres suivant :

- *Seed* : doit être le même pour chaque thread.
- *SequenceNumber* : doit être différent pour chaque thread et GPU.
- *Offset* : doit être différent pour chaque GPU.

Exemple :

GPU 1

Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
NumSeq=1	NumSeq=2	NumSeq=3	NumSeq=4	NumSeq=5
seed=12	seed=12	seed=12	seed=12	seed=12
offset=1	offset=1	offset=1	offset=1	offset=1

GPU 2

Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
NumSeq=6	NumSeq=7	NumSeq=8	NumSeq=9	NumSeq=10
seed=6	seed=6	seed=6	seed=6	seed=6
offset=4	offset=4	offset=4	offset=4	offset=4

8.3.3 Code : Utilisation des générateurs

```
#include <curand_kernel.h>

__global__
void working_kernel(curandState* tabDevGenerator, long n,…)
    {
    const int TID = Indice1D::tid();

    // Global Memory -> Register
    curandState localState = tabDevGenerator [TID]; //Optimisation

    float xAlea;
    float yAlea;
    for (long i = 1; i <= n; i++)
          {
          xAlea = curand_uniform(&localState);
          yAlea = curand_uniform(&localState);
           …
          work(xAlea,yAlea);
          …
          }
```

```
    //Register -> Global Memory
    //Necessaire si on utilise ensuite d'autrekernel avec nombre alea
    tabDevGenerator [tid] = localState;
    }
```

Note

Nvidia propose sur son site web une documentions complète pour l'API *curand*. Bon courage au lecteur!

8.4 Tirer un nombre et le ranger

Principe

On tire des nombres aléatoires sur le GPU et on les stocke dans un tableau GRAM. Avec des actions de Memory Managment, on peut ensuite exploiter ces nombres aléatoires soit sur le CPU soit sur le GPU dans d'autre kernel.

Analyse

Avantage : Utilisation possible des nombres générée dans des codes CPU standard.

Inconvénient : Limitation sur la quantité de nombres générés, de par la taille des tableaux.

Api Curand

Currand propose les méthodes suivantes :

- **curandCreateGenerator**
- **curandSetPseudoRandomGeneratorSeed**
- **curandGenerateUniform**
- **curandDestroyGenerator**

8.4.1 Code :

```
#include "curand.h"

//Création du générateur
```

```
// int seed=time(NULL);
int seed = 1234; //The same seed always produces the same sequence of results.
curandGenerator_t generator;

curandRngType_t generatorType= CURAND_RNG_PSEUDO_MTGP32;
/* Generateur disponible :
*     CURAND_RNG_PSEUDO_DEFAULT
*     CURAND_RNG_PSEUDO_XORWOW
*     CURAND_RNG_PSEUDO_MRG32K3A
*     CURAND_RNG_PSEUDO_MTGP32
*/

HANDLE_ERROR(curandCreateGenerator(&generator, generatorType));
HANDLE_ERROR(curandSetPseudoRandomGeneratorSeed(generator, seed));

//Memory Managment pour ptrDevXAlea, cudaMalloc

// Utilisation du générateur
HANDLE_ERROR(curandGenerateUniform(generator,ptrDevXAlea,n));

// Destruction du générateur
HANDLE_ERROR(curandDestroyGenerator(generator));


//V1 Utilisation CPU
//Memory Managment pour recupérer ptrDevXAlea

//V2 Utilisation GPU via un kernel

//Memory Managment libération de ptrDevXAlea
```

9 CPU Memory

9.1 Rappel

Pour rappel, il y a deux endroits mémoire où peuvent se trouver les tableaux :

- Sur le tas (tableau dynamique)
- Sur la pile (tableau automatique)

Les tableaux dynamique sont créés avec l'opérateur ***new*** et doivent explicitement être détruit après leur utilisation avec l'opérateur ***delete[]***. Les tableaux automatique sont quant à eut automatiquement détruit.

Syntaxe :

Utilisation de tableaux dynamique :

```
int n=…
int* ptrV=new int[n];
delete[] ptrV;
```

Utilisation de tableaux automatique :

```
const int N=…
int ptrV[N];
```

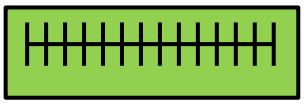
9.2 Concept

9.2.1 Principe de la RAM standard

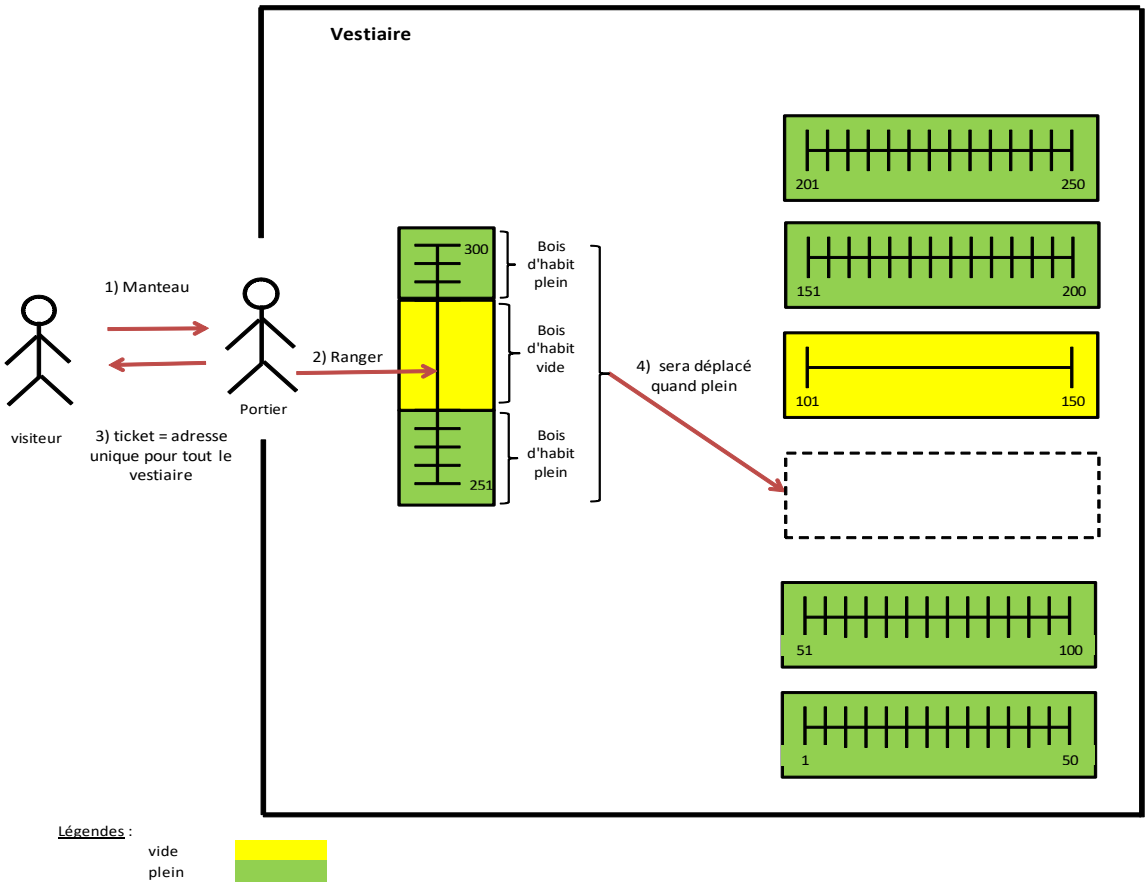
Analogie : *vestiaire – théâtre*

Contexte

Le vestiaire est un ensemble d'armoire mobile de manteaux rangés de manière compliquée. Une armoire mobile manteau peut contenir x manteaux. Elle sera représentée comme suite :



Le trait horizontal représente une tringle supportant des cintres. Les traits verticaux représente des cintre avec des manteaux.



Equivalence :

- Vestiaire = Mémoire
- Ticket = adresse

- Porte manteaux mobile = Page mémoire
- Portier = CPU

Observation :

Tout comme les portes manteaux mobiles bougent dans le vestiaire au cours du temps, les pages mémoires peuvent bouger aussi!

Exemple : *swap*

Lorsque la ram celle-ci est étendu sur le hdd.

Schéma (*)

Pour récupérer son manteau (lire le contenu d'une case mémoire), il faut demander au portier (CPU) d'aller chercher le manteau dans le vestiaire privé. Seul lui à le droit d'y aller, et seul lui connaît le porte manteau adéquat susceptible de contenir votre veste

- ➔ Il y a un **intérmédiaire**, le portier (CPU).
- ➔ Il y a un double accès à la mémoire. Le portier lui-même tiens un registre à jour contenant le lieu des portes manteaux mobiles.
- ➔ Lent et CPU sollicité.

De manière analogue, pour écrire en mémoire (déposer son manteau), il faut aussi passer par le gestionnaire du vestiaire (portier). Ici aussi, il y a un intermédiaire, un accès indirect et une sollicitation du CPU. C'est lent!

Avantage : La taille de la mémoire peut augmenter dynamiquement grâce au disque dur.

Inconvénient : Lent. Mais utilisé par tous!

9.2.2 Direct Memory Access (DMA ou pinned memoy)

Le but de la pinned memory est d'éviter de devoir passer par un intermédiaire, donc de gagner du temps et d'économiser des ressources CPU.

Analogie : *Vestiaire - Piscine*

Contrairement au théâtre, on peut ranger soit même ses habits dans les casiers disponibles. Il n'y a pas d'intermédiaire, c'est donc rapide (*en écriture*)! Les casiers étant fixe, on peut les retrouver tout seul puisqu'ils ne vont pas bouger de place. La *lecture* ne nécessite donc pas non plus d'intermédiaire. Le CPU n'est ici pas du tout utilisé.

Avantage : Performance !

Utile seulement pour copier des données entre la ram et un device ! La GRAM est un device ! N'affecte pas les performance de type RAM-RAM, mais de type RAM-GRAM. Augmente les performances surtout pour les grands volumes de données !

Inconvénient : Quand les casiers disponibles sont pleins, plus personne ne peut rentrer. Out of Memory!

Conséquence : A utilisé avec parcimonie.

9.2.3 Zero copy memory (map memory)

Contexte :

Lorsque le GPU n'a besoin d'accéder qu'**une seule fois** a des éléments d'un set de data volumineux, et que la GRAM est déjà bien pleine par le reste des besoins de l'application Cuda, on peut utiliser la ***zero copy memory***

Principe :

Comme son nom l'indique, la *zero memory copy* ne demande aucun transfert de data entre le *host* (CPU) et le *device* (GPU). Les datas resteront tout le temps du côté *host* sur la RAM. Du côté du *device*, le GPU va mapper une zone mémoire du *host*.

Technique :

Côté host (CPU)

Soit ***ptrData*** un pointeur sur le *host* adressant des données sur la RAM. On aimerait que le GPU puisse accéder à ces données via le pointeur ***ptrDevData***. Ce pointeur ***ptrDevData*** est un pseudo pointeur que l'on construit comme suit :

```
int vaNotUse=0; //doit être à zéro.
HANDLE_ERROR(cudaHostGetDevicePointer(    &ptrDevData,
                                          ptrData,vaNotUse));
```

Côté device (GPU)

On passera au GPU notre pseudo pointeur

ptrDevData

qui sera perçu comme s'il était le résultat d'un *cudaMalloc* standard. En effet le développeur à charge d'implémenter le kernel utilisant ce set de data spécial, travaillera de manière tout à fait habituel, comme si les données été bien présentes sur le GPU.

```
myKernel<<<dg,db>>>(float* ptrDevData,int n)
    {
    //comme d'habitude
    }
```

Les drivers Cuda s'occuperont d'aller chercher eux-mêmes les data sur le *host*!

Les data sur le host doivent être de type pinned memory. En effet, sinon elles risqueraient de changer de place, ce que le GPU ne verrait pas. Par ailleurs, sans cette accès de type DMA les performances seraient en plus catastrophiques.

Si deux thread accèdent à un même élément du set de data, les performances s'écroule par rapport au pattern standard *cudaMalloc* et utilisation de la global memory.

➔ Chaque élément du set de data doit être accédé qu'une seule fois.

Exemple : *Calcul d'un histogramme*

Input : tabNombres

Output : tabFrequences

Observation :

Chaque case de tabNombre doit être lu qu'une seule fois. Dans ce contexte particulier, la zero copy memory peut être utilisé pour le stockage de tabNombres. Les performances sont assées remarquables. A essayer!

9.2.4 WriteCompineMemory

Contexte :

Soit ***ptrData*** un pointeur adressant une zone mémoire du host. Cette zone mémoire peut être utilisant en même temps par le CPU et le GPU.

Exemple : *Pinned memory avec map memory enclenché*

Il peut y avoir donc des problèmes de concurrence, notamment si une partie des données se trouvent dans le cache du CPU. Cette concurrence liée à l'utilisation des caches est géré par le système lui-même. On peut cependant choisir cet algorithme de gestion. Il y en a

deux avec des performances totalement différentes. Soit on favorise le travail du CPU en autorisant l'utilisation de ces caches, soit on favorise le travail du GPU en interdisant ces caches CPU :

- Cache CPU enable
- Cache CPU disable

pour et uniquement pour les données de la *zero copy memory*.

Optimisation : *Côté host (CPU)*

Si une zone mémoire est utilisée abondamment par le CPU, mieux vaut laisser les caches du CPU activé. Si au contraire, seul le GPU utilise cette zone mémoire, il peut être intéressant de désactivé les caches CPU sur cette zone.

Note :

En désactivant les caches CPU

(N1) On a observé des temps traitement CPU 120x plus lent.

(N2) On a pas observé pour autant de performance GPU meilleurs.

➔ Approfondissez le sujet ou n'utilisez pas la *writeCombineMemory* (Cache CPU disable).
A priori on ne fait que perdre en performance.

9.3 Syntaxe

L'utilisation du MemoryManagment et des méthodes qui vont suivre requière le header suivant :

```
#include "cuda_runtime.h"
```

9.3.1 Direct Memory Access (DMA ou pinned memoy)

```
#include "cuda_runtime.h"

const int n=…
int* ptrV;

// Allocation
size_t size=n*sizeof(int);
HANDLE_ERROR(cudaHostAlloc((void**)&ptrV, size,cudaHostAllocDefault));

// Utilisation
// Utilisation de ptrV comme un pointeur standard CPU.
// En pratique remplire cette zone mémoire avec les données qui
// devrait être transféré côté GPU à l'aide du standard memcpy.

// Libération
HANDLE_ERROR(cudaFreeHost(ptrV));
```

9.4 Zero copy memory (map memory)

```
#include "cuda_runtime.h"

const int n=…
float* tabData;
float* ptrDev_data;

// Config
HANDLE_ERROR(cudaSetDevice(deviceID));
HANDLE_ERROR(cudaSetDeviceFlags(cudaDeviceMapHost));

// Allocation
size_t size= n * sizeof(float)
//V1 : Enable CPU cache
HANDLE_ERROR(cudaHostAlloc((void**) &tabData,size, cudaHostAllocMapped)

//V2 : Disable CPU cache
HANDLE_ERROR(cudaHostAlloc((void**) &tabData, size,
                cudaHostAllocMapped | cudaHostAllocWriteCombined));
```

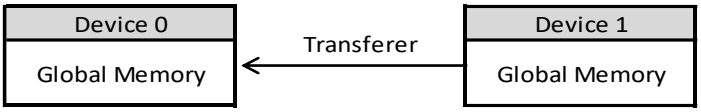
```
// Maping
int flagInutiliser=0; //doit être à zéro.
HANDLE_ERROR(cudaHostGetDevicePointer(&ptrDev_data, tabData,flagInutiliser));


// Utilisation
myKernel<<<dg,db>>>( ptrDev_data,n);


// Libération
HANDLE_ERROR(cudaFreeHost(tabData));
```

10 Multi GPU P2P

10.1 Objectif

Objectif :

Effectuer des transferts de données entre la Global Memory des GPU.



Contraintes :

Sans exploiter le Host (CPU et RAM) comme intermédiaire.

Pour y parvenir il faut activer la propriété *P2P* des *devices*.

Note

Depuis l'existence de l'UVA (espace d'adressage unifié), sans *P2P, on peut effectuer* des copy de *device* à *device*, avec la syntaxe naturelle suivante

```
HANDLE_ERROR(cudaMemcpy(ptrDevDestB, ptrDevSrcA,
                        size,cudaMemcpyDeviceToDevice));
```

Le transfert est alors plus lent qu'avec le *P2P* activé, mais bien fonctionnel. Le transfert de données transite alors par la ram du *host*, et le CPU est ainsi sollicité.

Device → Host →device

Avec le mode *P2P* enclenché, la copie devient alors *CPU-free* et *Ram-free*! Il est intéressant de souligner qu'en mode *P2P* comme en mode standard, la syntaxe pour la copie est la même. Seule une pré-configuration des GPU est nécessaire ! Il est donc facile de passer d'un code *P2P* à un code standard !

10.2 Performances

Contexte :

GPU :	Tesla M2090
MotherBoard :	PCI-Express x16 x16 Gen 2 (Théorique 8 Go/sec)
Host ←→ Device :	environ 7 Go/sec (avec PinedMemory)
Host ←→ Device :	environ 1.5 Go/sec (sans PinedMemory)

Exemple 1 :

Sur le même GPU :

Device ←→ Device : environ 60 Go/sec
Un speed up d'environ **10**!

Exemple 2 :

Sur deux GPU différents relié par le PCI-E de la carte mère

Device ←→ Device : environ 5 Go/sec

Exemple 3 :

Sur deux GPU différents d'un même boitier externe (QuadroPlex 7000)

Device ←→ Device : environ 5 Go/sec

Exemple 4 : *Sur une carte BI-GPU*

Device ←→ Device : TODO (pas de matériel)

Analyse :

(A1) Le transfert entre deux GPU ne peut excéder
8 Go/s en pci-express gen 2
16 Go/s en pci-express gen 3

(A2) Les données ne passent pas par le connecteur SLI

(A3) Si on ne fait pas du *P2P* entre 2 gpus, il faut passer par la ram du host et deux transferts sont réalisés au lieu de un seul.

GRAM→RAM→GRAM
ou

Device→host→Device

Même si le débit P2P entre 2 gpus est sensiblement plus faible qu'entre RAM ←→ GRAM, comme il n'y a qu'une copie à faire, au lieu de 2, c'est plus rapide !

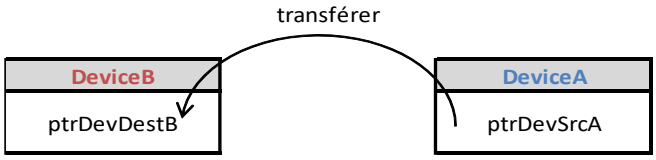
10.3 Exemple

Objectif :

Dans cet exemple on souhaite transférer sur le **deviceB** le contenu d'une zone mémoire du **deviceA**.

$$\overleftarrow{\text{ptrDevOutput} = \text{ptrDevInput}}$$

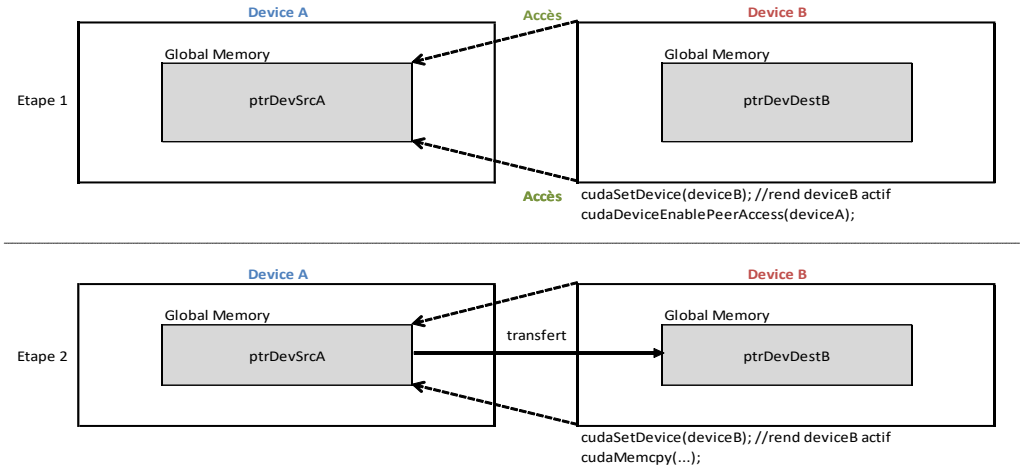
ptrDevDestB =ptrDevSrcA



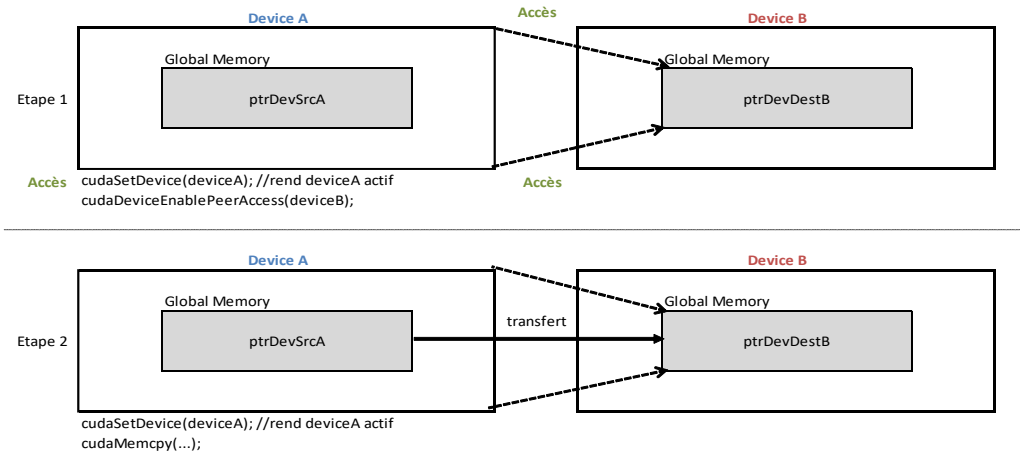
Deux Stratégies :

Deux stratégies possibles :

Stratégie 1 : *DeviceA* partage sa mémoire avec *DeviceB*, *DeviceB* lance la copie.



- Stratégie 2 : *Device* B partage sa mémoire avec *DeviceA*. *DeviceA* lance la copie.



Code : *Stratégie 1*

```
// Commun
int deviceA=0;
int deviceB=1;
size_t size=…;
float* ptrDevSrcA=NULL;
float* ptrDevDestB=NULL;
int flagInutiliser=0; //doit être a 0!

// DeviceA
HANDLE_ERROR(cudaSetDevice(deviceA));

HANDLE_ERROR(cudaMalloc((void**) & ptrDevSrcA, size));
HANDLE_ERROR(cudaMemset(ptrDevSrcA, 0, size));

// DeviceB
HANDLE_ERROR(cudaSetDevice(deviceB));
HANDLE_ERROR(cudaDeviceEnablePeerAccess(deviceA, flagInutiliser));//(#)
HANDLE_ERROR(cudaMalloc((void**) & ptrDevDestB, size));

//Obligatoire : passer deviceB actif avant la copie!
HANDLE_ERROR(cudaSetDevice(deviceB));
HANDLE_ERROR(cudaMemcpy(ptrDevDestB, ptrDevSrcA,
                    size,cudaMemcpyDeviceToDevice));

// Liberation DeviceB
HANDLE_ERROR(cudaSetDevice(deviceB)); //Inutile mais possible
HANDLE_ERROR(cudaFree(ptrDevDestB));

//Liberation DeviceA
HANDLE_ERROR(cudaSetDevice(deviceA));
HANDLE_ERROR(cudaFree(ptrDevSrcA));
```

Code : *Stratégie 2*

```
// Commun
int deviceA=0;
int deviceB=1;
size_t size=…;
```

```
float* ptrDevSrcA=NULL;
float* ptrDevDestB=NULL;
int flagInutiliser=0; //doit être a 0!

// DeviceA
HANDLE_ERROR(cudaSetDevice(deviceA));
HANDLE_ERROR(cudaDeviceEnablePeerAccess(deviceB, flagInutiliser)); //(#)

HANDLE_ERROR(cudaMalloc((void**) & ptrDevSrcA, size));
HANDLE_ERROR(cudaMemset(ptrDevSrcA, 0, size));

// DeviceB
HANDLE_ERROR(cudaSetDevice(deviceB));
HANDLE_ERROR(cudaMalloc((void**) & ptrDevDestB, size));

//Obligatoire : passer deviceA actif avant la copie!
HANDLE_ERROR(cudaSetDevice(deviceA));
HANDLE_ERROR(cudaMemcpy(ptrDevDestB, ptrDevSrcA,
                    size,cudaMemcpyDeviceToDevice));

// Liberation DeviceB
HANDLE_ERROR(cudaSetDevice(deviceB)); //Inutile mais possible
HANDLE_ERROR(cudaFree(ptrDevDestB));

//Liberation DeviceA
HANDLE_ERROR(cudaSetDevice(deviceA));
HANDLE_ERROR(cudaFree(ptrDevSrcA));
```

Important :

La méthode **cudaDeviceEnablePeerAccess** est asymétrique.

Un moyen mnémotechnique pour utiliser correctement cette asymétrie

cudaDeviceEnablePeerAccess= à le droit d'accéder à

Monitoring

L'outil ***nvidia-smi*** qui permet de monitorer le système peut être utile ici ! Voir le chapitre traitant du sujet.

10.4 Résolution de problèmes

Problème 1 *l'interopérabilité OpenGL - - Ordonancement*

Avec l'interopérabilité OpenGL vous risquez d'avoir le problème suivant:

```
[CUDA ERROR]:cannot set while device is active in this process
```

Dans le cas, essayer d'activer le p2*p-access* après l'activation *OpenGL* du device :

```
int deviceGL =0;
int deviceB=1;// non gl-device
HANDLE_ERROR(cudaSetDevice(deviceGL));
HANDLE_ERROR(cudaDeviceEnablePeerAccess(deviceB,0));      //Après
```

Problème 2 *l'interopérabilité OpenGL - - Choix du device maitre*

Dès qu'un *device* est partagé entre *OpenGl* et *Cuda* via la commande

```
HANDLE_ERROR(cudaGLSetGLDevice(deviceGL));
```

Il est possible que lors d'une connexion P2P avec le *gl-device* vous ayez le message d'erreur suivant :

```
[CUDA ERROR]: invalid resource handle
```

Dans ce cas, le *gl-device* doit impérativement être le **maître**!

Par conséquence, c'est le *gl-device* qui doit être actif pour lancer la copie. Le *gl-device* peut accéder à la mémoire des autres devices, mais pas l'inverse!

```
int deviceGL=0;// gl-device
int deviceB=1;// non gl-device
HANDLE_ERROR(cudaSetDevice(deviceGL));
HANDLE_ERROR(cudaDeviceEnablePeerAccess(deviceB,0));
```

Interprétation

Il s'agit vraisemblablement d'un mécanisme de protection. Il semble assez logique de ne pouvoir écrire dans une zone mémoire *OpenGL* à un moment quelquonque. L'intégrité de l'affichage des données de cette zone mémoire serait alors compromise. *OpenGL* va *lockée* cette zone mémoire lors du rendu, puis passer la main à *Cuda* pour une éventuelle modification des datas. *Cuda* n'a le droit de modifier les datas que quand il a la main.

10.5 Compatibilité Hardware

Selon la carte mère, la connexion P2P n'est pas nécessairement disponible entre tous les devices. On peut tester cette connectivité ainsi :

```
int flag01;
int devA=0; // par exemple
int devB=1; // par exemple
HANDLE_ERROR(cudaDeviceCanAccessPeer(&flag01,devA,devB));
```

Exemples

Server Cuda 1 *P2P compatibility : symetric matrix (6x6):*

	1	1	1	0	0
1		1	1	0	0
1	1		1	0	0
1	1	1		0	0
0	0	0	0		1
0	0	0	0	1	

où $a_{ij} = 1$ ssi P2P-enable entre gpu_i et gpu_j, 0 sinon

Server Cuda 2 *P2P compatibility : symetric matrix (3x3):*

	0	0
0		0
0	0	

où $a_{ij} = 1$ ssi P2P-enable entre gpu_i et gpu_j, 0 sinon

Dual IOH chipsets

Avec des cartes mères de type Dual IOH chipsets, vous pourriez vous trouver avec une matrice de compatibility nulle

http://stackoverflow.com/questions/6905549/cuda-4-0-peer-to-peer-access-confusion
(cuda 5, avril 2012)

10.6 P2P sur le terrain

Stratégie

En pratique, il vous est proposer d'activer tous les connexions *P2P* disponibles selon votre hardware, via :

```
Device ::p2pEnableALL() ;
```

On peut par ailleurs visualiser la matrice symétrique des connexions P2P disponibles, via :

```
Device ::printP2PmatrixCompatibility() ;
```

Il vous suffit ensuite d'effectuer du memory management de manière tout à fait standard :

```
HANDLE_ERROR cudaMemcpy(ptrDevDestB, ptrDevSrcA,
                                size,cudaMemcpyDeviceToDevice));
```

Note

La copie *deviceToDevice* est fonctionnel que le *P2P* soit activé ou non. Dans le cas où le *P2P* est activé, la copie est plus performante, et le CPU n'est pas sollicité !

11 Multi-GPU Parallélisme

11.1 Objectif

Objectif

Exploitèrent simultanément tous les GPU disponibles, en dirigeant sur ceux-ci l'exécution de kernels. Il peut s'agir :

- (C1) D'un même kernel avec des inputs différents.
- (C2) De kernels dont le code est totalement différent.

Principe

Utiliser son *API* préférée de thread coté host (*omp*, *boost*,…) et spécifier le *device* sur lequel on souhaite travailler

```
HANDLE_ERROR(cudaSetDevice(deviceId));
```

Spécifier ce *device* avant

- le memory management
- le lancement du kernel

11.2 OMP

Si l'on connait l'api *OMP*, le code à produire est trivial et ressemble à quelque chose du type :

Exemple

```
int nbDevice=Devices ::getDeviceCount();
#pragma omp parallel for
for(int deviceId=0 ; deviceId<nbDevice ;deviceId++)
    {
    HANDLE_ERROR(cudaSetDevice(deviceId));
    // memory management
    …
    k<<<dg,db>>>(…) ;
    //memoryManagement
    …
    } // barrrière de synchronisation implicite OMP
consolidation(…) ;
```

11.3 Boost

Considération technique

Problème

Le compilateur *nvcc* de Cuda pourrait avoir de la peine à compiler un code *boost* contenu dans un fichier *.cu*. Ce problème a été constaté entre autres avec Cuda5.0.

Solution

Il suffit pour s'en sortir de séparer la problématique *boost* et *Cuda* en répartissant les rôles dans deux fichiers différents :

- ***.cpp*** pour *boost*
- ***.cu*** pour *Cuda*

Exemple

L'exemple ci-dessous est volontairement très indirect et Objet, sinon il est vivement conseillé d'utiliser OMP.

```
// Code boost dans un .cpp

#include <boost/thread.hpp>

void exemple(Worker* tabWorker)
      {
      int nbDevice=Devices ::getDeviceCount();
      thread** tabThread = new thread*[nbDevice];

      for (int deviceId = 0; deviceId < nbDevice; deviceId++)
          {
          Worker* ptrWorker = tabWorker[deviceId];

          //HANDLE_ERROR(cudaSetDevice(deviceId));
          //Not necessary : already inlcude into Worker
          tabThread[deviceId] = new thread(&Worker::runGPU,
                                                  ptrWorker);
         // les threads host sont crées et démarrés ici
         }

      for (int deviceId = 0; deviceId < nbDevice; deviceId++)
         {
         tabThread[deviceId]->join();     // bloquant
         delete tabThread[deviceId];
         }
      delete[] tabThread;

      doSomething(tabWorker) ;
      }
```

.cpp

```
// Code cuda dans un .cu

class Worker
      {
      public:

      Worker (const Inputs& input,int deviceId)
          {
          this->input=input ;
          this->deviceId=deviceId;

          // Memory management
          HANDLE_ERROR(cudaSetDevice(deviceId));
          HANDLE_ERROR(cudaMalloc(…));
          …
          }

      virtual ~ Worker (void)
         {
          // Memory management
          HANDLE_ERROR(cudaSetDevice(deviceId)); // optional
          HANDLE_ERROR(cudaFree(…));
           …
          }

      void runGPU(void)
          {
          HANDLE_ERROR(cudaSetDevice(deviceId));
          k<<dg,db>>>(ptrDev, …) ;
          // Memory management
          HANDLE_ERROR(cudaMemcpy(…));
           …
          }

      Outputs getOutput(void)
          {
          return this->output;
          }

      private:

            // Inputs
            Inputs input ;
            int deviceId ;

            // Outputs
            Outputs output;

            // Tools
            … ptrDev …
        };
```

.cu

Notes

(N1) Le code la classe *Worker* pourrait sans autre être splitté en deux fichiers :

- .h
- .cu

ce qui est vivement conseillé (pas réalisé ici par soucis de place).

(N2) La classe *Worker* contient le *deviceId* sur lequel elle doit s'exécuter ! A elle de l'utiliser pour le *memory management* et le lancement du kernel.

11.4 OMP versus Boost

OMP

- Positif : simplicité, aucune installation.
- Négatif : peut en certaines circonstances manqués de souplesse (*)

Boost

- Positif : souplesse.
- Négatif : installation et configuration nécessaires.
  Code plus difficile (pièges sournois).
  Pas facilement intégrable à l'intérieur d'un *.cu*.

Conclusion

La plus part du temps *OMP* à suffisamment de souplesse. Il est vivement recommandé de l'utiliser ! La compilation est en effet facile, tout comme le code à produire et aucune installation n'est nécessaire Néanmoins, dans certains cas particuliers la souplesse de Boost peut être utile. Le prix à payer est alors une étape d'installation supplémentaire !

(*) On doit attendre la fin de l'exécution de tous les threads, donc de tous les kernels avant que le fil du programme ne puisse reprendre son cours. Ceci est dû à la barrière de synchronisation implicite à la fin de toute section parallèle. Les callback sont ainsi difficile à implémenter !

11.5 Thread versus Stream (Piège)

Concept

Une API de thread (comme Boost et OMP) permet de faire du parallélisme de kernel **inter-device**, les *streams* (voir chapitre dédié) permettent de faire du parallélisme de kernel **intra-device**

Thread	Parallélisme **inter-device**
Stream	Parallélisme **intra-device**

Voir le chapitre dédié au *stream*

Piège

La notion de *stream* offerte par cuda ne remplace pas les threads coté host :

```
cudaSetDevice(0));
k0<<<dg,db,0,stream0>>>(…) ;// assynchrone

cudaSetDevice(1) ;
k1<<<dg,db,0,stream1>>>(…) ;// (#) barrière pour k0 !!
```

Attente

On pourrait s'attendre à ce que *k1* débute « en même temps » que *k0*.

Observation

Il n'en est rien.

Il y a ici séquentialisation côté host!

Explication

En effet (#) est une **Barrière de synchronisation** pour *k0*. Ceci est vrai même si la *stream* dans laquelle on lance *k0* n'est pas la même que *k1*.

Par contre, si le device était le même, k0 et k1 serait executée en paralléle sur le device, mais comme le device n'est pas le même, il parte en séquentielle.

```
cudaSetDevice(0));


kA<<<dg,db,0,streamA>>>(…) ;// assynchrone
kB<<<dg,db,0,streamB>>>(…) ;// assynchrone
```

Ici kB est exécuté en parallèle avec kA sur le device 0

Conséquence

Dans le premier exemple, pour faire partir *k1* avant que *k0* ne soit terminé, il faut impérativement utiliser une *API* de thread comme *OMP* ou *Boost*.

12 Stream (Intra-device)

12.1 Objectif

Là où les API de thread coté host comme *OMP* et *Boost* permette de répartir l'exécution en parallèle sur des *devices* différents, les *stream* permettent d'effectuer en parallèle plusieurs kernels en simultanée sur le même *device*.

Thread	Parallélisme **inter-device**
Stream	Parallélisme **intra-device**

Notons que les *stream* permettent aussi d'effectuer du memory management en parallèle sur un même *device*. On peut effecteur des copies *deviceToHost* en même temps que des copies *hostToDevice* (requiert coté host de la DMA)

12.2 Principe

Hypothèse

Les *stream* servent à faire du parallélisme **intra-device**. On suppose donc ici disposer d'un seul GPU.

Modèle

Il faut voir une *stream* un peu comme le pendant du « *thread* » mais coté *device* :

Deux kernels dans 2 *streams* différents sur un même device s'exécutent en parallèle

Attention

Mais si vous lancer deux kernels dans deux threads cotés host sur le même *device*, ceux-ci vont se retrouver dans la même file d'attente sur le GPU et être exécuté de manière séquentielle sur le GPU.

Queue

Chaque *stream* possède sa file d'attente sur le *device*. Toute opération dans une *stream* (memory management, kernel) atterrit dans sa file

d'attente sur le device. Cette file d'attente est ordonnée, et les éléments de cette file d'attente sont exécutés les uns après les autres de manières séquentiels, en respectant la chronologie temporelle induit par l'ordre d'insertion.

Hardware

Tous les GPU ne possède de

AsyncEngineCount

Les files d'attentes des *streams* seront traités en parallèles selon les capacités du hardware. Il y a séquentialisation partielle ou totale selon le nombre *AsyncEngineCount* à disposition coté device *!*

```
Devices ::getAsyncEngineCount();
Devices ::print() ;
```

Stream par Defaut

Même si vous n'utilisez pas de *stream explicitement dans vos codes*, tout se fait dans un *stream* quand même : la *stream* par défaut d'id valant 0.

Barrière de synchronisation

Rappelons que

(R1) Le lancement d'un kernel est **asynchrone**

(R2) Une action de memory management précédent l'appel d'un kernel est une **barrière de synchronisation**, pour autant que les deux se trouvent dans la même *stream* !

(R3) Le lancement d'un nouveau kernel est une **barrière de synchronisation**, pour le kernel lancé précédemment dans le même stream !

Stream

Dans un *stream* on peut :

(P1) Lancer un kernel (syntaxe longue)

(P2) Effectuer du memory management (syntaxe modifiée)

Evidemment, on ne peut faire du memory management dans un *stream* s_1 avec des donnés calculés avec un kernel lancé dans le *stream* s_2, sauf évidemment après des étapes de synchronisation. On peut effectuer des synchronisation explicites comme suit

- Inter-stream : **cudaDeviceSynchronize()**
- Intra-stream : **cudaStreamSynchronize(stream)**

Pour mettre en pratique un *stream*, il suffit de suivre la «*trilogie*» suivante

On crée un stream s_i.

On effectue du memory management dans s_i

On lance un kernel k_i dans s_i

On effectue du memory management dans s_i

Cas pratique *Application mono-gpu*

Objectif

On aimerait lancer *n* kernel en parallèle sur un GPU.

Hypothèse

On suppose ici que les *n* kernels k_i possèdent une logique de fonctionnement indépendantes les uns des autres, et que les outputs des uns ne sont pas des inputs des autres !

Solution

Il suffit de travailler avec *n* stream et de suivre la «*trilogie*» ci-dessus. Une fois les *n* threads terminés on peut alors consolider les résultats des uns et des autres

12.3 Syntaxe

Création

```
// Activer d'abord le device sur lequel est destiné le stream
HANDLE_ERROR(cudaSetDevice(deviceId));

// Synchrone
cudaStream_t streamId ;
HANDLE_ERROR(cudaStreamCreate(&streamId));
```

Lancement d'un kernel

```
// Assynchrone
size_t sizeSM=0;
myKernel<<<dg,db,sizeSM, streamId >>>( … );
```

Memory management

```
// Assynchrone
HANDLE_ERROR (cudaMemcpyAsync (ptrA,ptrDevA,sizeA,
                               cudaMemcpyHostToDevice, streamId);
```

Synchronisation

```
// Assynchrone
HANDLE_ERROR (cudaStreamSynchronize(streamId));
```

Destruction

```
// Synchrone
HANDLE_ERROR(cudaStreamDestroy(streamId));
```

Rappel

```
// synchroniser tout le device courant
HANDLE_ERROR (cudaDeviceSynchronize());
```

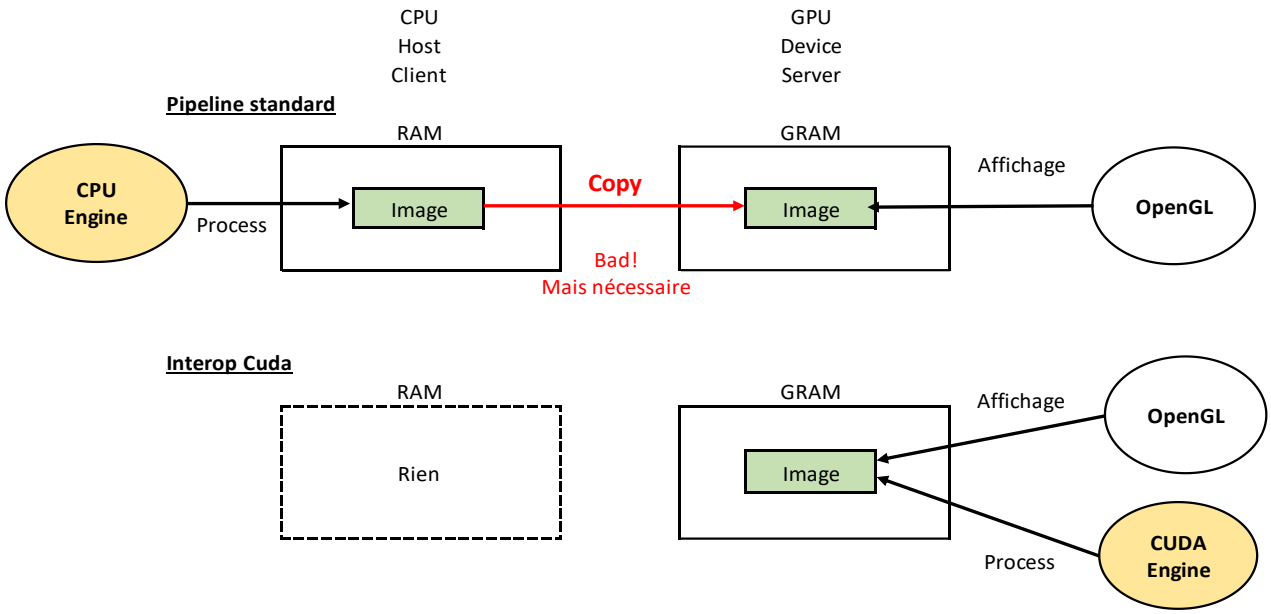
13 Interopérabilité GL

Pré requis :

Il faut impérativement lire le document **bilatRappelOpenGL** avant!

13.1Principe

Objectif :



L'objectif est de pouvoir utiliser des ressources créé par OpenGL directement dans un kernel Cuda. Ce qui devrait permettre d'économiser des transferts mémoire entre le CPU et le GPU.

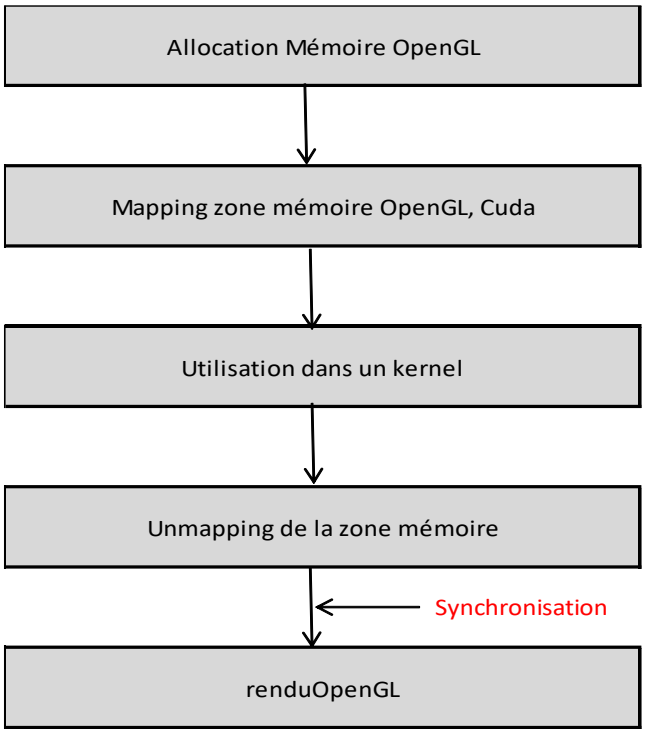
Idée général :

Cuda propose un mécanisme permettant d'utiliser des ressources OpenGL directement avec un kernel Cuda. Ce mécanisme s'appel l'interopérabilité OpenGL.

Etapes :

- Allocation mémoire OpenGL standard via des VBO, PBO, textures.
- Mapping OpenGL, Cuda

- Utilisation dans un kernel Cuda
- Unmapping des ressources
- Rendu OpenGL utilisant la mémoire alloué précédemment.



L'étape synchronisation de la zone mémoire partagée par Cuda et OpenGL est gérée par Cuda de manière transparente. Dès qu'une zone mémoire OpenGL est partagée avec Cuda, la synchronisation pour le rendu est garantie. OpenGL ne va pas lire les données tant que Cuda n'aura pas redonnées la main (ie fait un unmapping).

13.2Syntaxe

Includes :

Windows :

```
#include <windows.h>
#include <GL/gl.h>
#include "cuda.h"
#include "cuda_runtime.h"
#include "cuda_gl_interop.h"
```

Linux :

```
#include <GL/gl.h>
#include "cuda.h"
#include "cuda_runtime.h"
#include "cuda_gl_interop.h"
```

Contraintes :

Il faut impérativement ajouter les includes OpenGL avant les includes Cuda, et sous Windows, il faut ajouter les includes propres à Windows avant ceux d'OpenGL.

Objectifs :

L'objectif est de partager une zone mémoire du GPU entre OpenGL et Cuda.

Inputs :

On dispose d'un VBO OpenGL via un VBO ID

```
GLuint vboID;
```

On connaît aussi la taille du VBO

```
size_t sizeVBO;
```

et le type des éléments qu'il contient.

```
float3
```

Observation :

C'est OpenGL qui partage sa zone mémoire et non cuda. La zone mémoire est donc allouée par la technique des VBO d'OpenGL.

Solution :

Il nécessaire de connaître ces deux phases :

- Initialisation
- Utilisation

Initialisation

```
// Input from OpenGL
GLuit vboID;
```

Le VBO OpenGL sera vue du côté Cuda comme une *cudaGraphicsResource*

```
cudaGraphicsResource * ptrCudaResource;
unsigned int flagMapping= cudaGraphicsMapFlagsNone; //Par defaut
cudaGraphicsGLRegisterBuffer(&ptrCudaResource, vboID, flagMapping);
```

Il y aura d'autre type de flag de mapping.

Flag	Droit d'accès Cuda
cudaGraphicsMapFlagsNone	Lecture et écriture
cudaGraphicsRegisterFlagsReadOnly	Lecture uniquement
cudaGraphicsRegisterFlagsWriteDiscard	Ecriture uniquement

Utilisation :

Pour éviter les problèmes de concurrence avec OpenGL, cuda doit "Mapper" la ressource. Dès lors, OpenGL n'y aura plus accès.

```
cudaStream_t stream = NULL;
cudaGraphicsMapResources(1, &ptrCudaResource, stream));
```

Récupération d'un pointeur sur la zone mémoire

```
float3* ptrDevVBO; //Pointeur qui sera utilisé dans un kernel Cuda
size_t sizeVBO;
cudaGraphicsResourceGetMappedPointer((void**)&ptrDevVBO, &sizeVBO,
    ptrCudaResource)
```

Utilisation du pointeur sur le device :

```
dim3 dg(8, 8);
dim3 db(16, 16, 1);
myKernel<<<dg,db>>>(ptrDevVBO,sizeVBO);
```

Libération de la ressource, OpenGL peut y avoir accès pour y effectuer le rendu

```
cudaGraphicsUnmapResources(1, &ptrCudaResource, stream);
```

Obligatoire avant cuda 5.0 *cudaGLSetGLDevice*

En cuda le device est un état (par thread) que l'on choisit en début de code par

```
int deviceID = 0;
cudaSetDevice(deviceID)
```

pour l'intéropérabilité entre OpenGL et Cuda, la commande ci-dessus est remplacé avant cuda 5.0 par

```
cudaGLSetGLDevice(deviceID);
```

Résolution de problème :

Il se peut que vous rencontriez le message suivant :

```
[CUDA ERROR] : cannot set while device is active in this process in
src/cpp/core/xxx.cpp at ligne yy
```

Ceci peux se produire si vous le *glDevice* après l'avoir déjà utilisé en Cuda.

Exemple : *interdit*

```
cudaMalloc(...) //on device deviceID
cudaGLSetGLDevice(deviceID);
```

Solution :

```
cudaGLSetGLDevice(deviceID);
glutCreateWindow(...);
cudaMalloc(...) //on device deviceID
```

13.3Exemple

Hypothèse :

On suppose que l'on dispose d'un VBO de sommet. Autrement dit on a un ID du VBO

```
GLuint vboID; //Construit par OpenGL
```

son type

```
float3
```

et sa taille

```
size_t sizeVBO;
```

Informations :

- `Code propre à OpenGL`
- `Code standard`

Code :

```
#include <GL/glew.h> //prise en charge VBO
#include <GL/glut.h> //fenetrage, include <GL/gl.h>
#include "cuda.h"
#include "cuda_runtime.h"
#include "cuda_gl_interop.h"

// Cuda
extern void useVBOCuda (float* ptrDevVBO, size_t size);

static void initWindow();
static void display();
static void initVBO();

// OpenGL
static GLuint vboID;

//Cuda InteropGL
static cudaGraphicsResource* ptrCudaResource;
```

```
void main(int argc,char** argv)
    {
    glutInit(&argc,argv);

    int deviceID = 0;
    cudaSetDevice(deviceID);
    initWindow();
    initVBO();

    // Phase d'initialisation
          {
          cudaStream_t stream = NULL;
          unsigned int flag = cudaGraphicsMapFlagsNone;
          cudaGraphicsGLRegisterBuffer(&ptrCudaResource, vboID,flag);
          }

    // Utilisation
          {
          float3* ptrDevVBO; //Pointeur utilisé dans le kernel Cuda
          size_t sizeVBO;
          cudaGraphicsMapResources(1, &ptrCudaResource, stream));
          cudaGraphicsResourceGetMappedPointer((void**)&ptrDevVBO,
                &sizeVBO, ptrCudaResource);

          useVBOCuda(ptrDevVBO, sizeVBO);
          cudaGraphicsUnmapResources(1, &ptrCudaResource, stream);
          }

    glutMainLoop(); //bloquant!
    }
```

```
void initWindow()
    {
// Create FreeGLUT Window
    int width = 1200;
    int height = 750;
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
    // GLUT_DOUBLE double Buffering (utile lors d'animation)
    // GLUT_DEPTH  tampon de profondeur (utile pour 3D)
    // GLUT_RGBA couleur RGBA.
```

```
    glutInitWindowSize(width, height);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Tuto OpenGL VBO");
    glewInit(); // pour VBO

    // Assignation des callbacks
    glutDisplayFunc(display);
    }

void initVBO()
    {
    // Creation ID VBO
    glGenBuffers(3, &vboID);
    // Association Array - VBO
    glBindBuffer(GL_ARRAY_BUFFER, vboID);

    // Copie Host->Device
    size_t sizeVBO = NB_SOMMETS_TRIANGLES*NB_COMPOSANTE_XYZ;
    sizeVBO *=sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, sizeVBO,0,GL_STREAM_DRAW);
    }

void display()
    {
    …
    glBindBuffer(GL_ARRAY_BUFFER, vboID);
    // draw vbo
    …
    }.cpp
```

```
void useVBOCuda(float3* ptrDevSommet, size_t sizeVBO);
static __global__ void useVBO(float3* ptrDevSommet, size_t sizeVBO);

void useVBOCuda(float3* ptrDevVBO, size_t sizeVBO)
    {
    dim3 dg(8, 8);
    dim3 db(16, 16, 1);

    useVBO <<<dg,db>>>( ptrDevVBO, sizeVBO);
    }
```

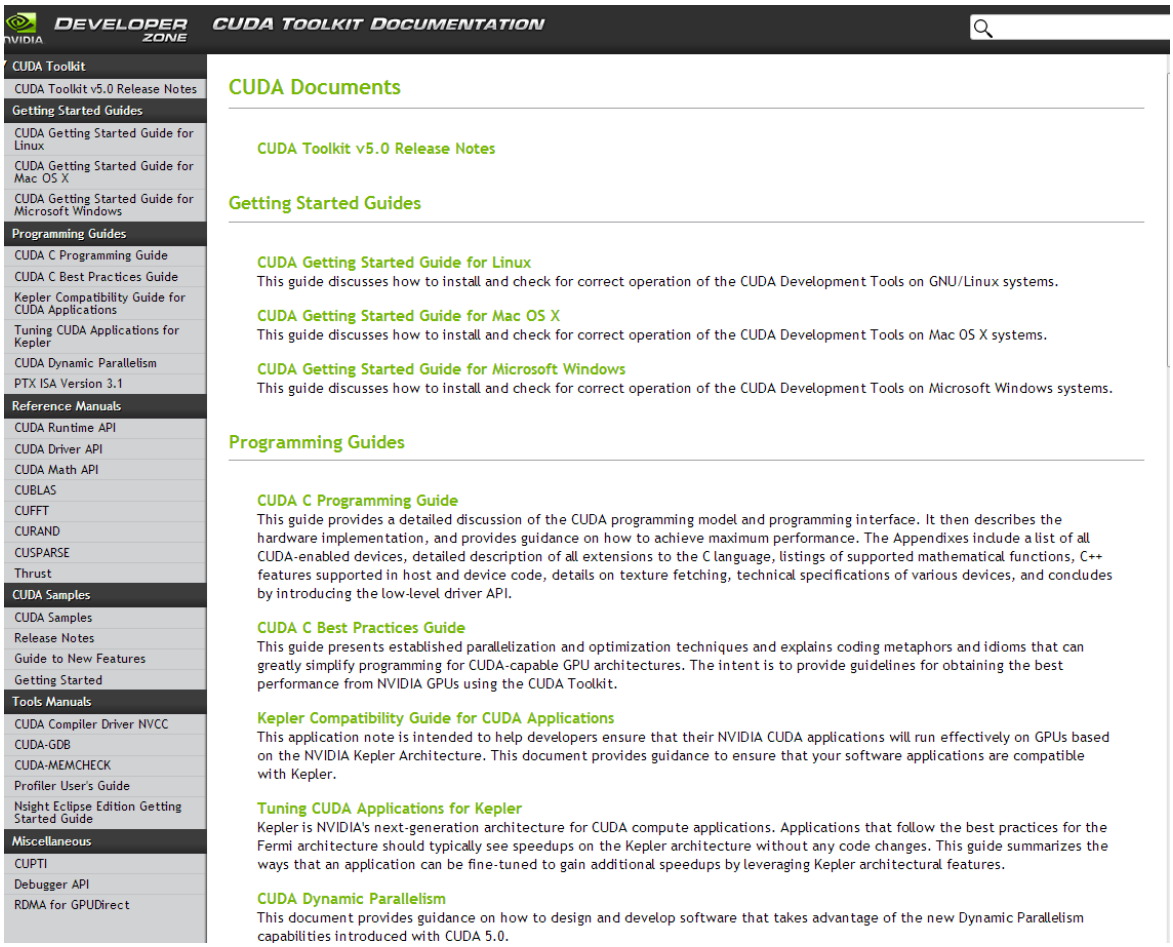
.CU

14 Bibliographie

14.1 Site Web

Documentation Officielle

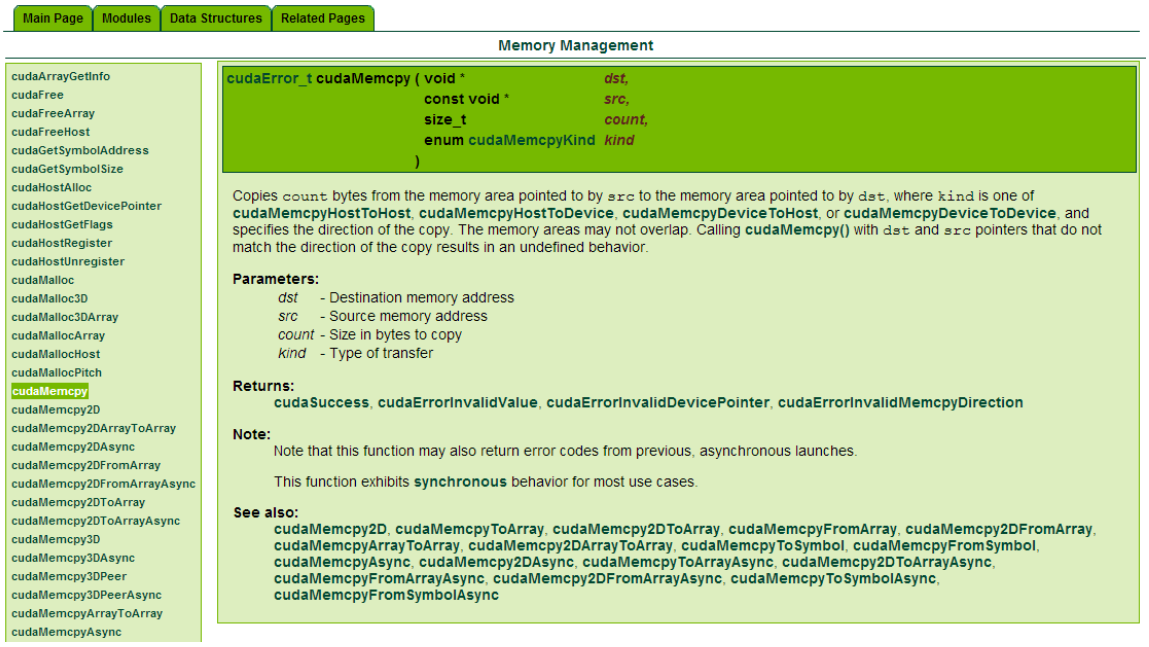
http://docs.nvidia.com/cuda/#axzz41ZSRnCBL



Prototype

Pour la syntaxe, nous 'invitons le lecteur à consulter ce site web :

http://horacio9573.no-ip.org/cuda/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html



Très pratique !

Truc

En tapant **cudamemcpy** dans *googl* vous devriez tomber dessus, même si l'adresse change