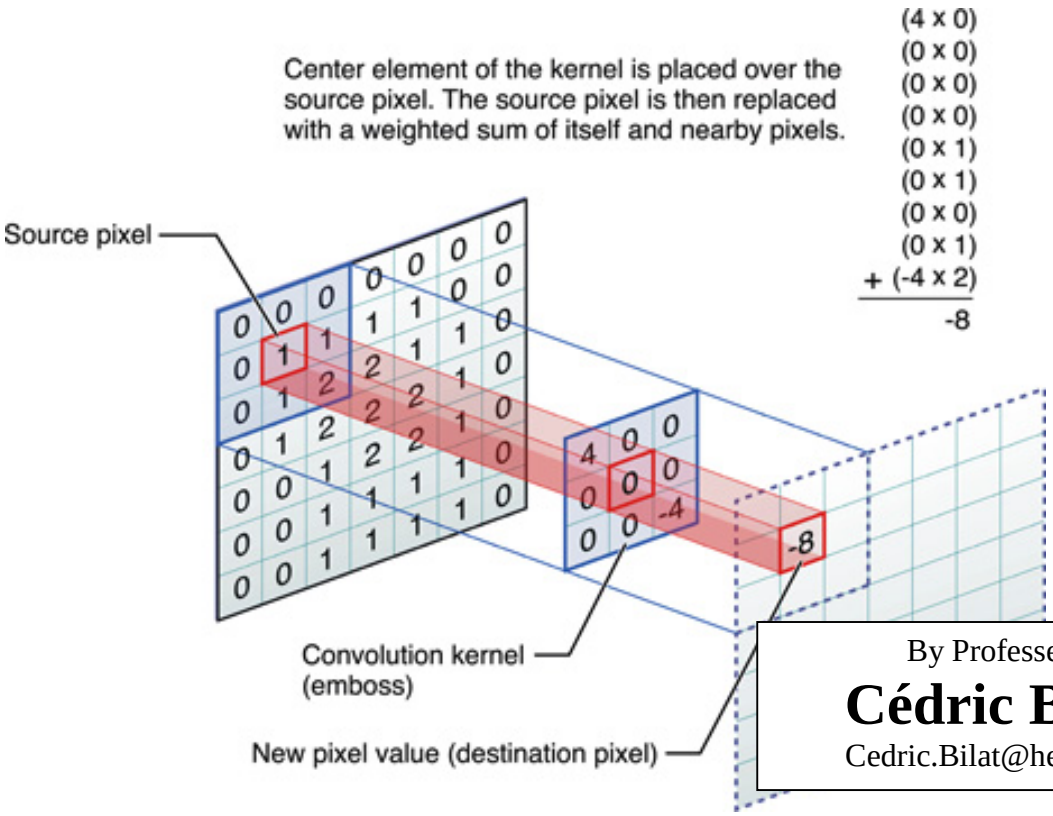
Problèmes Parallelisation



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

Convolution

Traitement d'image

Contexte

La discipline du traitement d'image consiste à transformer une image en lui appliquant des opérations successives de filtrage. Ces opérations sont de plusieurs types, nous pouvons citer en particulier :

- Les algorithmes traitant l'image **indépendamment** pixel par pixel
  Exemples : *Le seuillage*

- Les algorithmes traitant un pixel en tenant compte d'un **voisinage** donné.
  Exemples :
  - Filtre **linéaire** tels que le *floutage* ou mise en évidence des bords.
  - Filtre **non-linéaire** tels que la *morphologie mathématique*.

Filtre linéaire

Le filtrage d'une image par un filtre linéaire se fait de la façon suivante :

1. Choit du **noyau** de filtrage selon l'effet désiré.
2. Application du noyau sur l'image par **glissement** et **balayage**.

Définition *Noyau*

Un noyau est une matrice contenant des poids.

Application d'un noyau

Illustrons la technique sur un exemple !
Supposons qu'en input on ait une image 4x4

0	0	100	100
0	0	100	100
50	50	200	200
50	50	200	200

sur laquelle on souhaite appliquer le noyau 3x3

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

L'algorithme est itératif. Pour mettre à jour un pixel de l'image source, disons le jaune :

On positionne le centre du filtre (case verte) sur le pixel à updater jaune de l'image source. Le noyau *superpose* l'image. La valeur du pixel de sortie s'obtient alors ainsi:

- En multipliant les pixels de l'image source par les poids rencontrés dans le noyau, selon la règle : Un poids du noyau s'applique sur le pixel de l'image qu'il superpose.

- En sommant ensuite ces multiplications!

L'image de sortie complète s'obtient en appliquant le filtre sur chacun des pixels de l'image source (balayage). L'ordre de balayage des pixels de l'image n'a pas d'importance. Chaque pixel de l'image de sortie peut se calculer en parallèle.

Exemple

0	0	100	100
0	0	100	100
50	50	200	200
50	50	200	200

Image source

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Noyau

	56	94	
	78	122	

Image de Sortie

On constate que toute l'image n'a pas pu être parcourue. En effet les bords de l'image (cases grises) ne peuvent pas être updatées car le filtre sort de l'image pour ces zones là! L'image de sortie est donc plus petite que l'image d'entrée.

56=**1/9** * (0+0+100+0+0+100+50+50+200)

Correction dimension

Pour s'affranchir de ce problème de dimension, on peut virtuellement agrandir l'image source par **symétrie**.

0	0	100	100
0	0	100	100
50	50	200	200
50	50	200	200

Image source

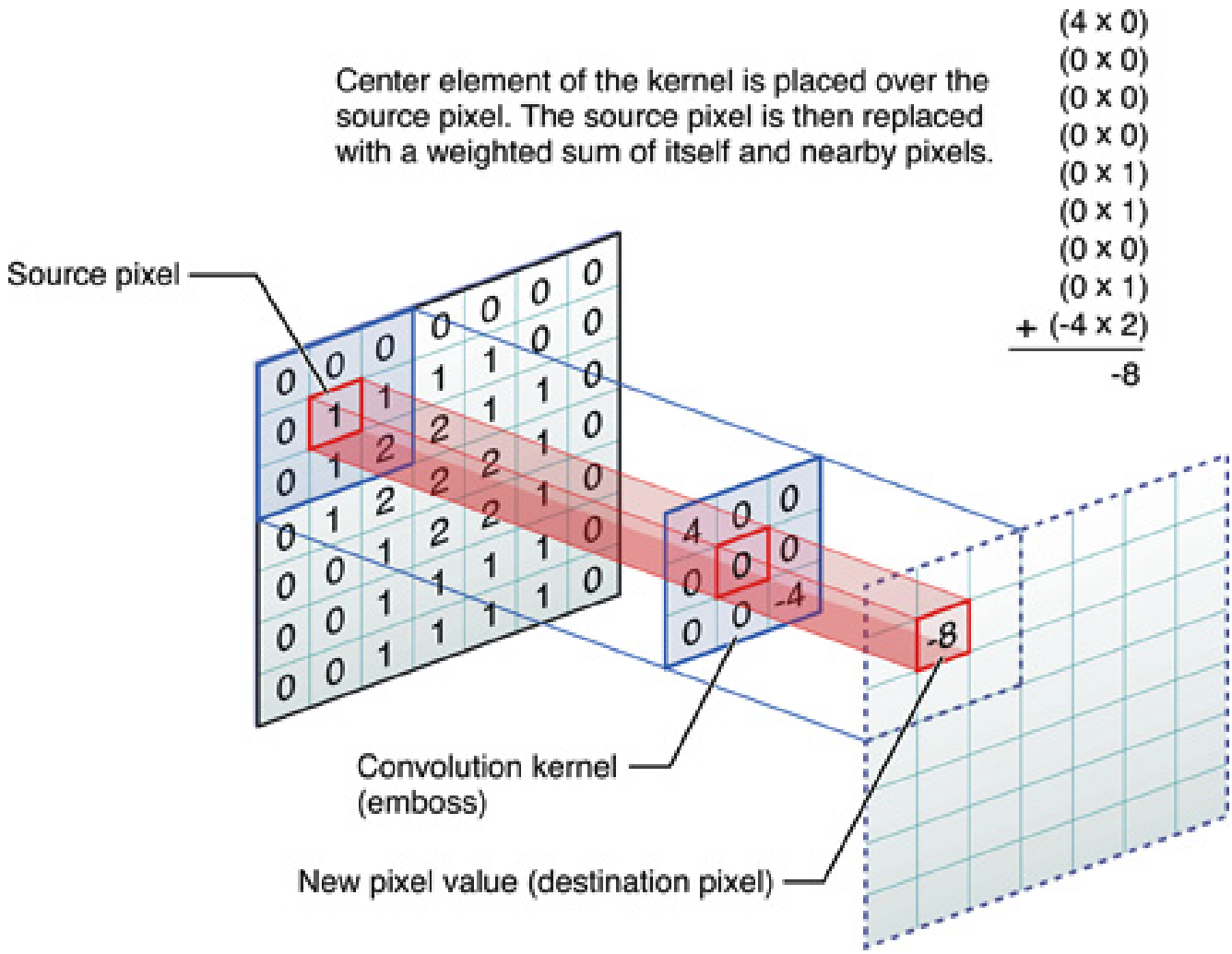
0	0	0	100	100	100
0	0	0	100	100	100
0	0	0	100	100	100
50	50	50	200	200	200
50	50	50	200	200	200
50	50	50	200	200	200

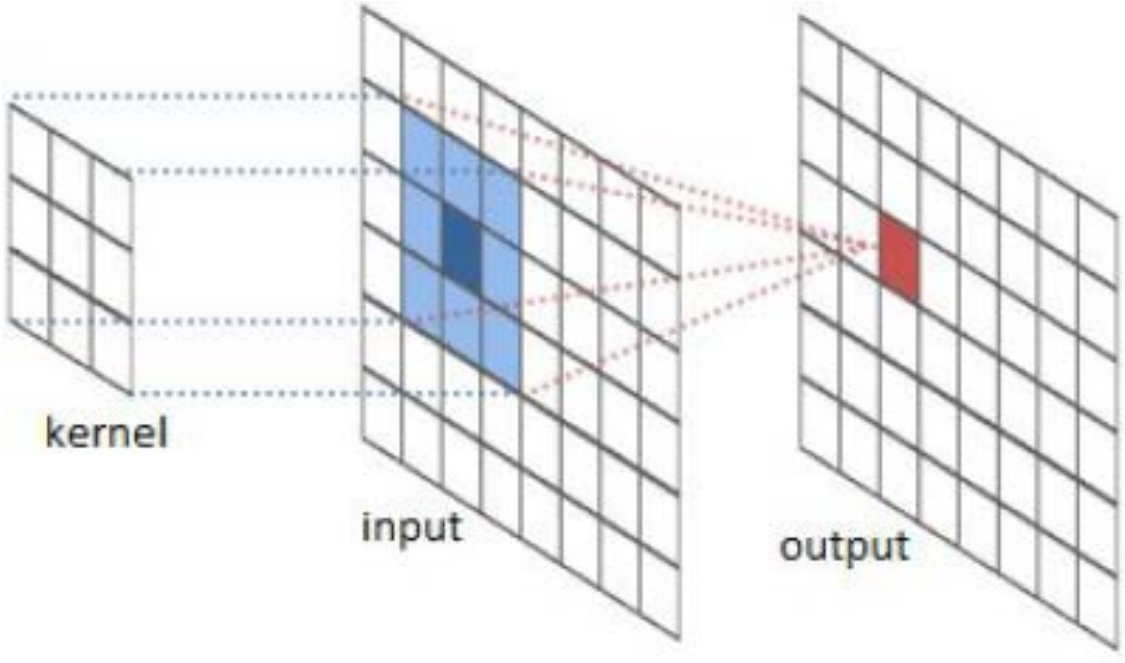
Image source virtuellement agrandie

D'un point de vue informatique, les textures bénéficient de cette propriété et évite le développeur de s'occuper des pixels particulier se trouvant dans les bords de l'image.

Convolution

Le principe est donc celui d'une convolution :


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)


kernel
input
output

Exemples

Filtre moyenneur

Noyau :

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

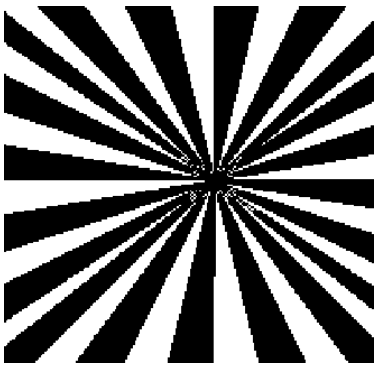
$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} = 1$$

$$c_{ij} = 1/n^2$$

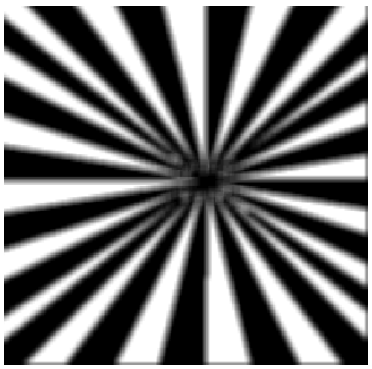
Effet :

Un filtre moyenneur est un filtre qui va introduire un flou sur une image. Ce flou sera d'autant plus prononcé que la dimension du filtre moyenneur est importante. Dans le domaine fréquentiel, le filtre moyenneur est un filtre passe-bas. Les zones de l'image contenant des hautes fréquences (les détails) seront donc très dégradées par ce filtre.

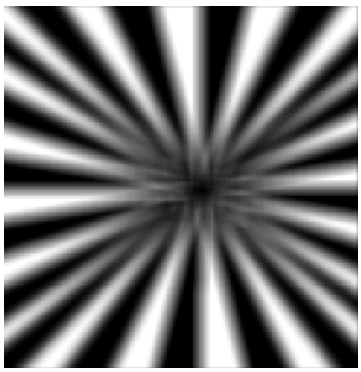
Exemple :



Source



Noyau 5x5



Noyau 10x10

Filtre dérivateur

Noyau :

-1/2	1/2

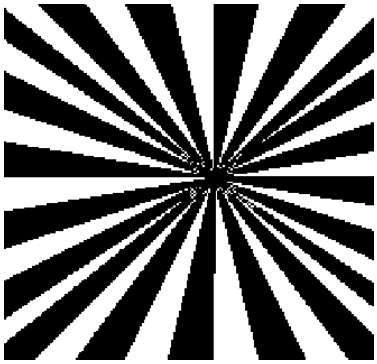
$$\sum_{i=1}^{n} c_i = 1$$

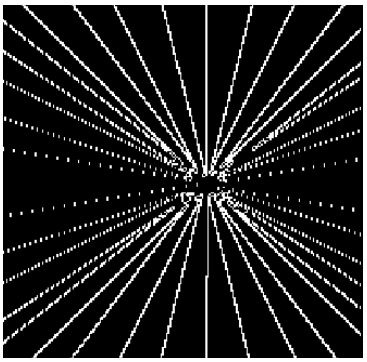
$$c_i = 1/n$$

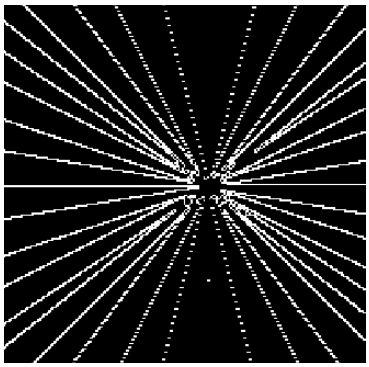
Effet :

Un filtre dérivateur met en évidence les variations sur l'axe vertical ou horizontal de l'image. Comme dans le domaine fréquentiel, la dérivation est une opération qui favorise les hautes fréquences. Des informations apparaitront dans l'image filtrée là où il y a de grande variation dans l'image source.

Exemple :


Source


Noyau Horizontal


Noyau Vertical

Note : Les pixels sont ici affichés en valeur absolue. Afin d'augmenter la visibilité de l'effet du filtre, un seuillage a également été appliqué après l'opération de filtrage.

Filtre extraction de contour *Laplacien*

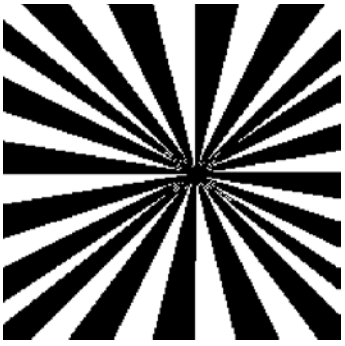
Noyau :

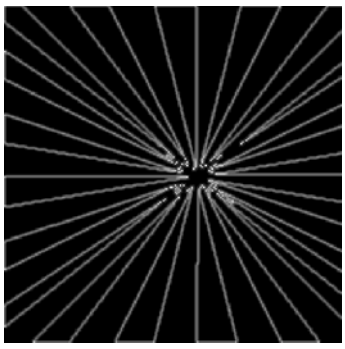
1/8	1/8	1/8
1/8	-1	1/8
1/8	1/8	1/8

Effet :

Faire apparaitre les contours des formes apparaissant dans une image

Exemple :


Source


Contour

Note : Les pixels sont ici affichés en valeur absolue. Appliquer un filtre moyenneur pour diminuer le bruit avant de faire la détection de contour.

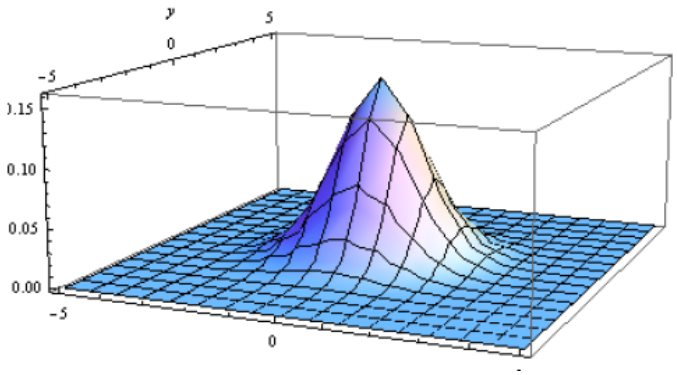
Applications

Contexte

On s'intéresse au **noyau 9x9** (*raw major linéarisé*) suivant :

0.0828, 0.1987, 0.3705, 0.5366, 0.6063, 0.5366, 0.3705, 0.1987, 0.0828, 0.1987, 0.4746, 0.8646, 1.1794, 1.2765, 1.1794, 0.8646, 0.4746, 0.1987, 0.3705, 0.8646, 1.3475, 1.0033, 0.4061, 1.0033, 1.3475, 0.8646, 0.3705, 0.5366, 1.1794, 1.0033, -2.8306, -6.4829, -2.8306, 1.0033, 1.1794, 0.5366, 0.6063, 1.2765, 0.4061, -6.4829, -12.7462, -6.4829, 0.4061, 1.2765, 0.6063, 0.5366, 1.1794, 1.0033, -2.8306, -6.4829, -2.8306, 1.0033, 1.1794, 0.5366, 0.3705, 0.8646, 1.3475, 1.0033, 0.4061, 1.0033, 1.3475, 0.8646, 0.3705, 0.1987, 0.4746, 0.8646, 1.1794, 1.2765, 1.1794, 0.8646, 0.4746, 0.1987, 0.0828, 0.1987, 0.3705, 0.5366, 0.6063, 0.5366, 0.3705, 0.1987, 0.0828

Les coefficients doivent encore être divisés par 100.



TODO

Appliquer le noyau ci-dessous a une vidéo Full HD.
Passer d'abord l'image en niveaux de gris !

Question

Cuda pemet-il de conserver la cadence initial de la vidéo ?
Que fait ce noyau ?

Amplification

Problème

L'image output est très sombre et on y voit pas grand-chose.

Solution 1 : *Translation*

Effectuez une translation de disons **+64**

Solution 2 : *Zoom*

Effectuez une amplification de ***50**

Solution 3 : *Transformation Affine*

Lancer un nouveau kernel qui calcul la valeur du pixel

- la plus grande
- la plus petite

puis appliquer la **transformation affine** appliquant

- la plus petite valeur sur noire
- la plus grande sur blanc

Le kernel devant recherchez le *min* et *max* nécessite une réduction, sympathique !

Programmez ce kernel vous-même ou utiliser les classes de réduction mises à disposition ! Attention, dans ce cas, ces classes de réduction effectue une réduction

- intra-block
- inter-block

A vous cependant de remplir préalablement les blocks en **shared memory SM**

Acquisition du flux video

Utiliser *OpenCV* !

Préliminaires

Vérifier qu' *OpenCV* est fonctionnel.

Etape 1

Ouvrez le projet *BilatToolsOpenCV* dans le workingset *Tools* d'Eclipse.
Les tools de haut niveau englobant des appels OpenCV sont mis à disposition pour effectuer la capture d'une video

CaptureVideo

ou d'une webcam

CaptureCamera

Les deux héritant de ***Capture_A***

Etape 2

Modifier le chemin vers les flux vidéo dans le fichier

main.cpp (du folder src/test)

Noter que des *videos* sont mises à disposition

/media/Data/Video

Etape 3

Compiler, runner et valider !

Acquistion

L'exemple ci-dessus effectue l'acquisition et le rendu avec *OpenCV*.

Dans votre projet *Convolution*, utiliser la classe de haut niveau ***CaptureVideo*** pour effectuer l'acquisition :

```
CaptureVideo  captureur(pathToVideo, titre);

Mat matImage=captureur.capturer(); // capture une image seulement ( à utiliser en boucle!)

uchar4* image= CaptureVideo::castToUChar4(&matImage); // format cuda
```

mais utilisez l'API *BilatImageCuda* pour effectuer le rendu !

Indications

(I1) Rappelez-vous qu'avec celle-ci, la méthode

fillImage(uchar4* prtDevImage)

est appelée en boucle en mode animation enclenchée.

Effecteur la capture de la video à l'intérieur de cette méthode *fillImage*, puis à l'aide de *Memory Management* copier l'image sur le device à l'adresse ***prtDevImage*** donnée par la méthode *fillImage*.

Pour une implémentation efficace, placer l'unique instance *captureur* comme attribut de votre classe *ImageConvolutionCuda* possédant la méthode *fillImage*. Instancier *captureur* dans le constructeur!

(I2) Une fois que vous arrivez réussi l'acquisition avec *OpenCV* et le rendu avec l'API *BilatImageCuda*, lancer un premier kernel passant l'image en noir et blanc, par exemple par moyennage de RVB. Observez-vous une réduction des *fps* avec la vidéo *FullHD* mis à disposition ?

(I3) Implémenter le kernel de convolution 9x9 mis à disposition pour le détourage. Les 81 valeurs du kernel se trouvent dans un fichier *.txt* sur le server *ftp*. Ce noyau doit s'appliquer sur une image noir et blanc. Vous aurez donc deux kernels successifs à lancer dans *fillImage*, avec peut-être une synchronisation entre les deux !

Contraintes

Utiliser impérativement des **chemins absolus** vers les vidéos fournies dans

/media/Data/Video

Implémentation Cuda

Contraintes Cuda

Effectuer différentes implémentations en *Cuda*

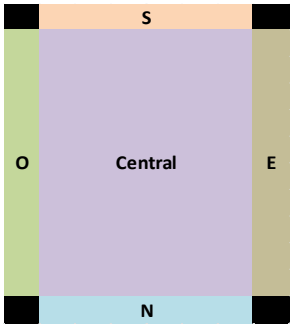
- Tout en ***global memory GM***
- L'image dans une ***texture (***pour s'affranchir de la problématique des bords de l'image)
- Le noyau en ***constant memory CM***
- Le noyau en ***shared memorySM (cache)***
- Le noyau dans une ***texture***

Note :

La dimension d'une texture ne doit pas nécessairement être une puissance de 2 (de taille supérieur à 8), mais la dimension pourrait influencer les performances.

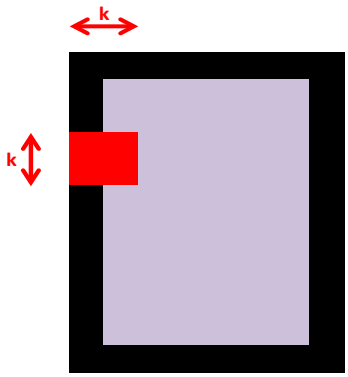
Indication

(I1) Dans le cas d'une implémentation coté ***Host*** ou ***Device*** sans ***TextureMemory TM***, partitionnez l'image en 9 classes d'équivalences selon la topologie du voisinage:



Pour tenir compte de ces 9 régions, votre code devra contenir 9 branchements conditionnels ***if***, ce qui est fastidieux, surtout que la majorité des pixels se trouvent dans la partie centrale ! On se contentera donc de produire un code ne marchant que pour les pixels appartenant à la partie centrale, et on ne fera aucun calcul pour les pixels du bords qui resteront noir. L'image calculée sera donc plus petite que l'image originale.

La bordure noire sera d'une largeur de *k/2* pixels où *k* est la dimension (impaire) du **noyau**.



Dans le cas *Cuda* et de l'utilisation d'une texture, vous n'aurez pas cette bordure **noire** !

Difficulté Cuda

Difficulté : Gagner la guerre des indices !

Hypothèse :

(H1) Le noyau est carré de dimension kxk , avec k impaire.
(H2) L'image comme le noyau sont row-major linéarisé.

Notation :

Soit s l'indice du pixel de l'image row major linéarisé à calculer
Soit (i, j) la paire d'indice indexant le point de l'image à calculer, où

- i représente la ligne
- j représente la colonne

On a évidemment

$$s=j+(i*w)$$

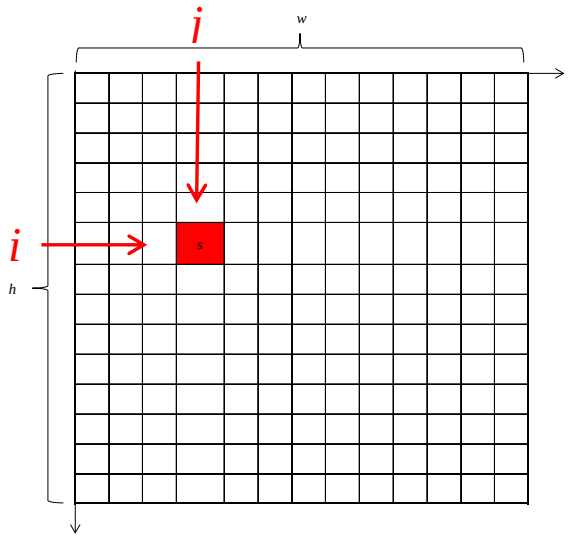


Image wxh

Problème :

Il faut parcourir tout le noyau et l'appliquer à l'image autour du pixel *s*. Ainsi on obtient la nouvelle valeur du pixel *s*.

Indication Cuda *balayage*

Objectif

Expliquer une technique de balayage (générique), applicable tant au noyau qu'à l'image

Notation

Notons *ss* le point central du noyau de côté *k* impair
On a évidemment

$$ss = k\frac{k}{2}$$

Exemple : *k=3*

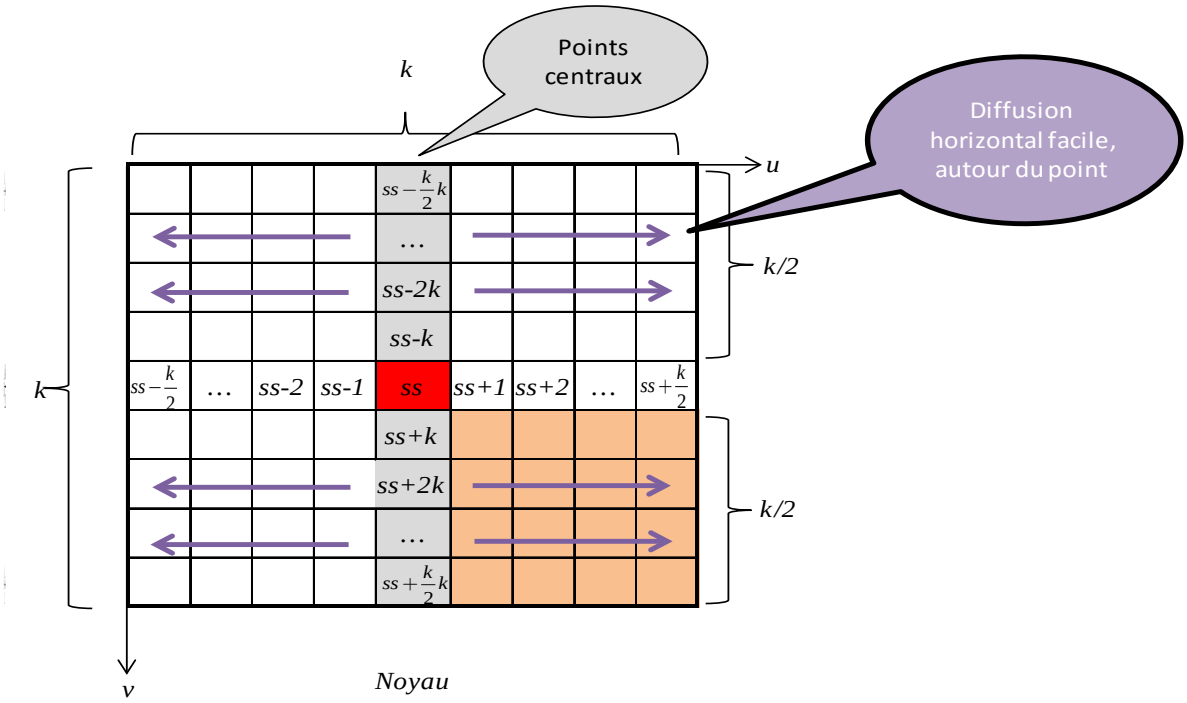
0	1	2
3	4	5
6	7	8

$$ss = k\frac{k}{2} = 3\frac{3}{2} = \frac{9}{2} = 4 \qquad \text{(en } integer\text{)}$$

Technique de balayage :

Une des possibilités est de partir du point central du noyau *ss* (et non du coin supérieur gauche).

Notons qu'il est très facile d'obtenir les pixels voisins de droite et gauche en partant des pixels centraux!



Pour parcourir le quart inférieur droit du noyau (partie orange), on peut ainsi commencer par faire varier la variable *v* (vertical) et à l'intérieure de la boucle, effectuer la diffusion horizontal "droite-gauche" :

```
for v ∈ [1, k/2]          // Diffusion verticale
        {
        //parcours horizontal, diffusion "droite-gauche" ± 1
        }
```

Pour le parcours horizontal :

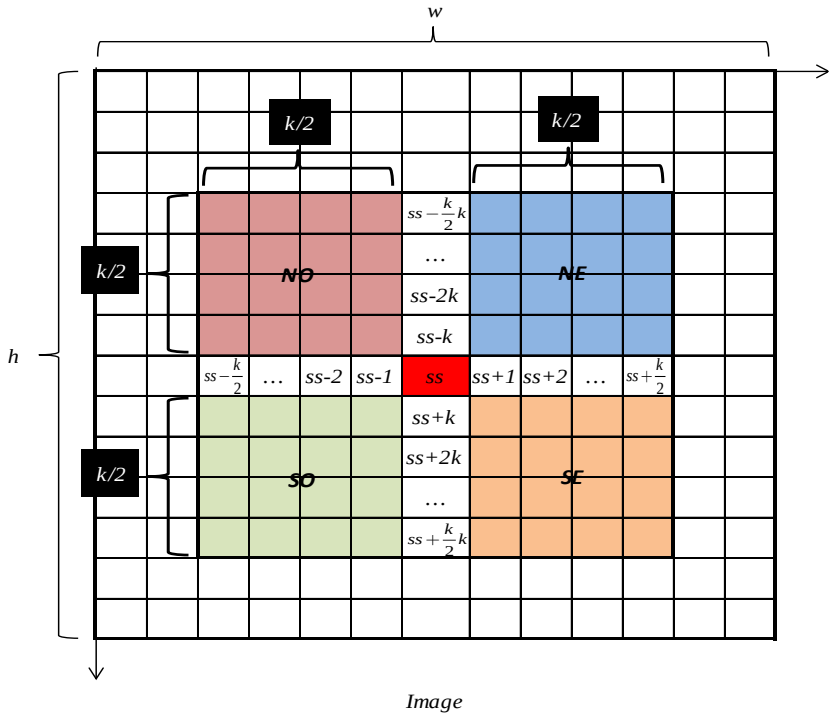
```
for u ∈ [1, k/2]          // Diffusion horizontale
        {
        int indice1D = [ ss + (vk) ] + u
                       (points centraux)   (diffusion horizontal)
        work(indice1D);
        }
```

Saut de *k* pour passer d'une ligne à une autre

Pour parcourir tout le noyau, et pas seulement le quart inférieur droite, il suffit juste de jouer sur les signes. Partitionnant le noyau en quatre zones géographique :

- Sud Est = SE
- Sud Ouest = SO
- Nord Est = NE
- Nord Ouest = NO

que l'on visualise comme suit :



Image

Le parcours complet s'obtient alors

```
for v ∈ [1, k/2]          // Diffusion verticale
        {
        for u ∈ [1, k/2]          // Diffusion horizontale
                {
                 int indice1D_SE = ss + (vk) + u;
                 int indice1D_SO = ss − (vk) + u;
                 int indice1D_NE = ss + (vk) − u;
                 int indice1D_NO = ss − (vk) − u;

                work(indice1D_SE);
                work(indice1D_SO);
                work(indice1D_NE);
                work(indice1D_NO);
                }
        }
```

Par la suite on simplifiera les notations comme suit :

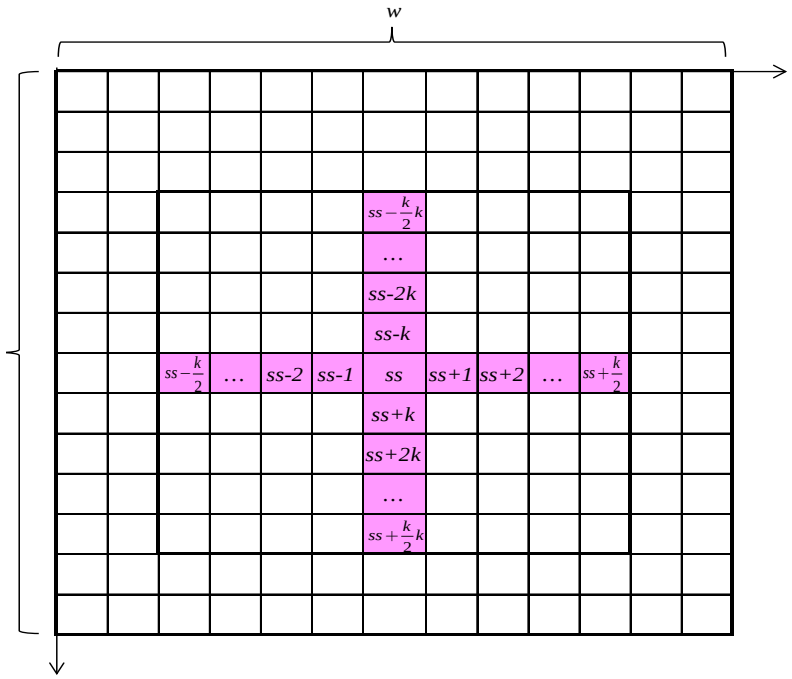
```
for v ∈ [1, k/2]          // Diffusion verticale
        {
        for u ∈ [1, k/2]          // Diffusion horizontale
                {
                int indice1D = [ ss ± (vk) ] ± u          //4 cas au total
                work(indice1D);
                }
        }
```

(points centraux : $ss \pm (vk)$; diffusion horizontal : u)

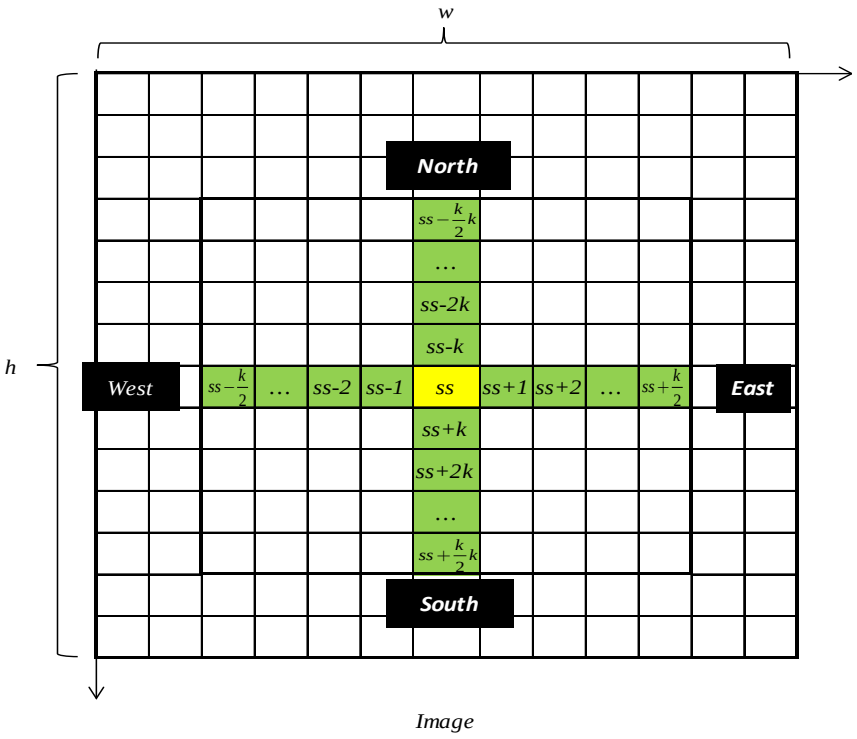
Saut de k pour passer d'une ligne à une autre

Parcours croix

Comme la numérotation des indices de boucles sur v et u commence à 1, il faut encore parcourir la *croix central* rose qui a échappé aux parcours jusqu'ici :



Si on avait commencé à 0 les indices de boucles sur v et u, le point central jaune aurait été parcouru 4 fois, et les bras vert de la croix 2 fois.



Au final

Soit *s* l'indice pixel central, et *k* le nombre de pixel à sauter pour passer d'une ligne à l'autre. Si *k* est impaire, et le noyau *raw major linéarisé*, on parcourt toute le noyau comme suit.

```
for v ∈ [1, k/2]
        {
        for u ∈ [1, k/2]
                {
                int indice1D = [ ss ± (vk) ] ± u          //4 cas au total
                work(indice1D);
                }

        // Parcours des bras de la croix central
        int indiceBrasSouthNorth = ss ± (vk)
        int indiceBrasEastWest   = ss ± (v)   //v car u ne varie pas ici

        work(indiceBrasSouthNorth);
        work(indiceBrasEastWest);
        }
// Parcours croix centre
int indiceCentre = ss
work(indiceCentre)
```

Saut de *k* pour passer d'une ligne à une autre

points centraux

diffusion horizontal

Indication Cuda *Sans Texture*

Objectif

Application du noyau ***RowMajor*** sur l'image ***RowMajor*** sans texture.

Solution

Il faut parcourir ici en même temps :

- l'image
- le noyau

La technique de balayage est la même que présenté précédemment. Il faut juste l'appliquer en même temps sur le noyau et sur l'image, en tenant compte des bonnes dimensions! En effet le nombre de colonnes de ces 2 container n'est pas le même (*k* pour le noyau et w pour l'image). Pour balayer verticalement, il faut effectuer des sauts de *k* pour le noyau, alors que pour l'image le saut sera de *w*.

Notation

Soit *k* le nombre de colonnes du noyau (k impaire).
Soit *w* le nombre de colonnes de l'image.
Soit *ss* le point central du noyau (row-major linéarisé). Il est **fixe** et vaut $ss = k(k/2)$.
Soit *s* le point de l'image à calculer (row-major linéarisé). Il est **fixe** aussi.

Update *pixel s de l'image row-major linéarisé*

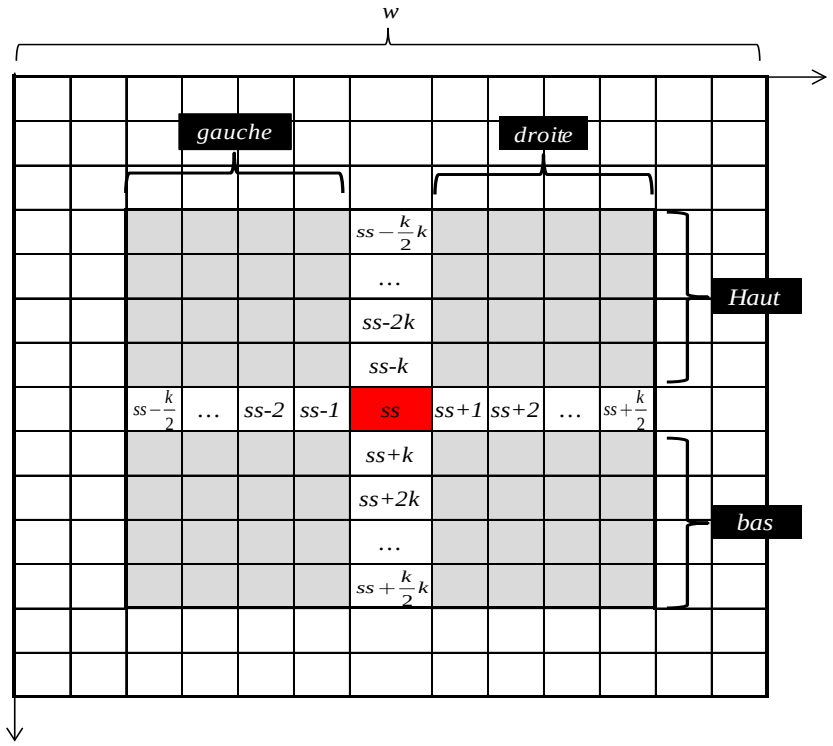
```
float sum=0
for v ∈ [1, k/2]      // Diffusion verticale
    {
    for u ∈ [1, k/2]
        {
        sum+ = noyau[ (ss ± vk) ± u ] *image[ (s ± vw) ± u ]   (#)
        }
    // Parcours des bras de la croix central :
    // Bras east et Bras west :
    sum+ = noyau[ ss ± v ] *image[ s ± v ]      // v car u ne varie pas ici !
    // Bras south et Bras north :
    sum+ = noyau[ (ss ± vk) ] *image[ (s ± vw) ]
    }
// Parcours centre croix
sum+ = noyau[ (ss) ] *image[ (s) ]
// Update pixel s image raw major lineariser
imageOut[s] = sum
```

Haut / Bas : $(ss \pm vk)$ — Gauche / Droite : $\pm u$ — Haut / Bas : $(s \pm vw)$ — Gauche / Droite : $\pm u$

Observation :

(O1) Seule ***u*** et ***v*** varient dans le parcours ci-dessus. Le reste est fixe!

(O2) Notons que dans (#), il y a 4 cas au total!
Afin de choisir les bonnes paire de + et de – à associer,
on peut séparer le noyau en 4 zones de + et de - comme suit :



Ce qui donne les 4 cas suivant en changeant l'association des signes afin de parcourir tout les points cardinaux:

$$\sum_{u,v} noyau\left[(ss+vk)+u\right] * image\left[(s+vw)+u\right]$$ *Sud Est* *(SE)*

$$\sum_{u,v} noyau\left[(ss+vk)-u\right] * image\left[(s+vw)-u\right]$$ Sud Ouest (SO)

$$\sum_{u,v} noyau\left[(ss-vk)+u\right] * image\left[(s-vw)+u\right]$$ Nord Est (NE)

$$\sum_{u,v} noyau\left[(ss-vk)-u\right] * image\left[(s-vw)-u\right]$$ Nord Ouest (NO)

La seule différence entre le parcours du noyau et de l'image est le pixel central et les sauts de lignes :

	Pixel Central	Saut de ligne
Noyau	ss	k
Image	s	w

Indication Cuda *Avec Texture*

Objectif

Application du noyau RowMajor sur l'image RowMajor avec texture

Solution

On met l'image dans une texture, mais pas le noyau qui trouve plus naturellement sa place en *constant memory CM*.

Observons qu'ici que :

(O1) Nous ne somme plus **row-major linéarisé** avec un indice 1D sur l'image, mais que la texture nous fait travailler avec un couple d'indice 2D pour accéder aux éléments de l'image.

(O2) La fonction tex qui permet d'accéder aux éléments d'une texture possède un couple d'indice croisé

$\boxed{\text{tex2d}(\text{texture}, j, i)}$ // Attention d'abord j !!!!!

Le premier indice est celui de la **colonne**, et le seconde celui de la **ligne** !

Notation

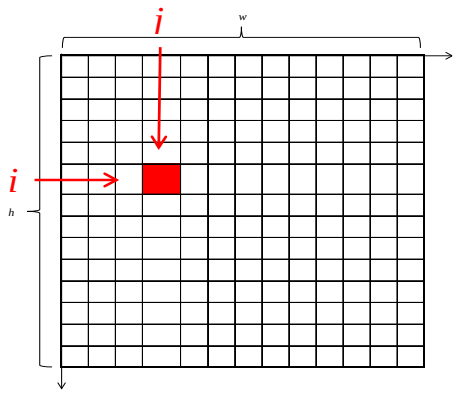
Soit k le nombre de colonnes du noyau, k impaire.

Soit w le nombre de colonnes de l'image.

Soit ss le centre du noyau. Il est fixe est vaut $ss = k(k/2)$.

Soit (i, j) la paire d'indice indexant le point de l'image à calculer, où

- i représente la ligne
- j représente la colonne



float sum=0

for $v \in [1, k/2]$ // Diffusion verticale

{

for $u \in [1, k/2]$ // Diffusion horizontale

{

$$sum+ = noyau\left[(ss \pm vk) \pm u\right] * tex2d_{image}\left(j \pm u \;,\; (i \pm v)\right)$$

Points centraux : $(ss \pm vk)$ — Haut / Bas

diffusion horizontale : u — Gauche / Droite

diffusion horizontale : $j \pm u$ — Gauche / Droite

diffusion verticale : $(i \pm v)$ — Haut / Bas

}

// Parcours des bras de la croix central :

// Bras east et Bras west :

$$sum+ = noyau\left[(ss \pm v)\right] * tex2d_{image}\left(\hat{j} \;,\; (i \pm v)\right)$$

// *v* car *u* ne varie pas ici !

// Bras south et Bras north :

$$sum+ = noyau\left[(ss \pm vk)\right] * tex2d_{image}\left(j \pm v \;,\; \hat{i}\right)$$

}

// Parcours centre croix

$$sum+ = noyau\left[(ss)\right] * tex2d_{image}\left(j, i\right)$$

// Update pixel *s* image raw major linéariser

$s = j + (i * w)$

imageOut[s]=sum // On ne peut pas écrire dans texture, mais dans zone mémoire binder, oui !

Observation :

(O1) Notons que seule ***u*** et ***v*** varient dans le parcours ci-dessus. Le reste est fixe!

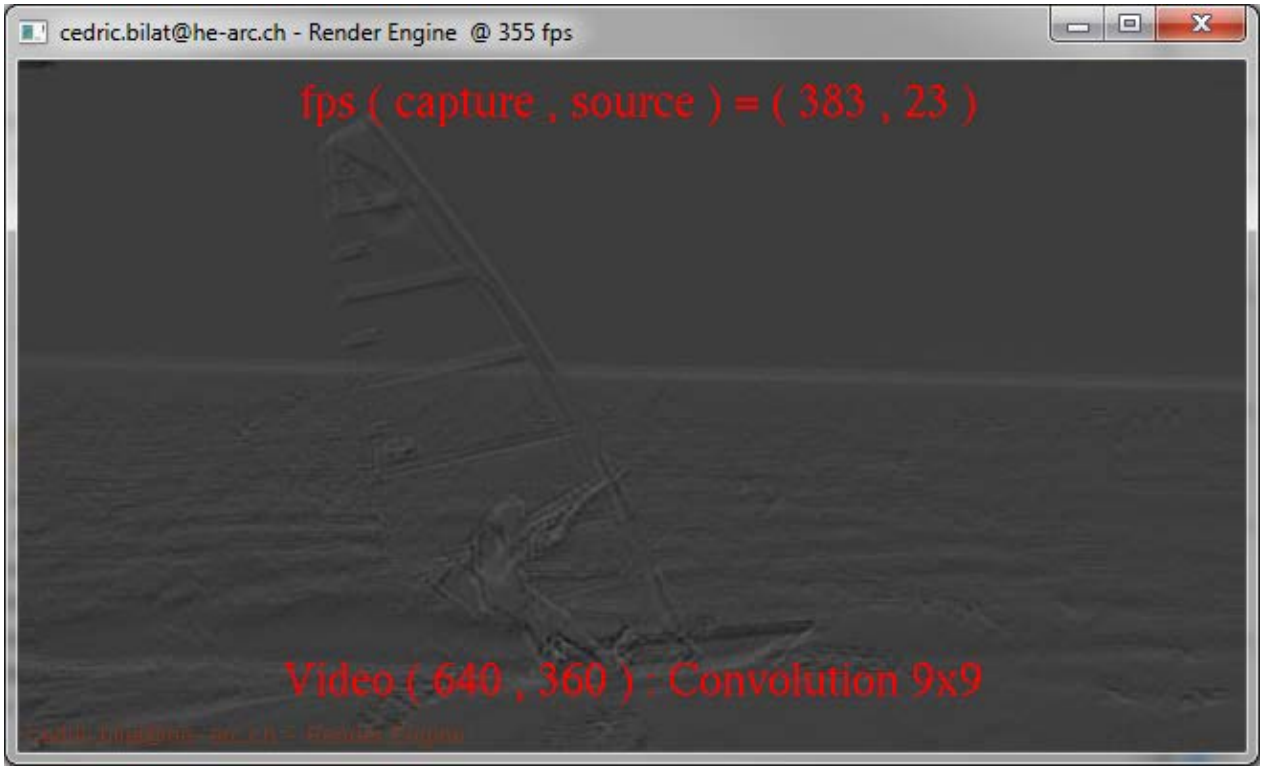
(O2) Comme on ne peut pas écrire dans une texture, on doit écrire directement dans la zone mémoire représentant l'image. L'image étant row-major linéarisé, il faut passer d'un couple d'indice 2D (i, j) à un indice 1D :

$$s = j + (i * w)$$

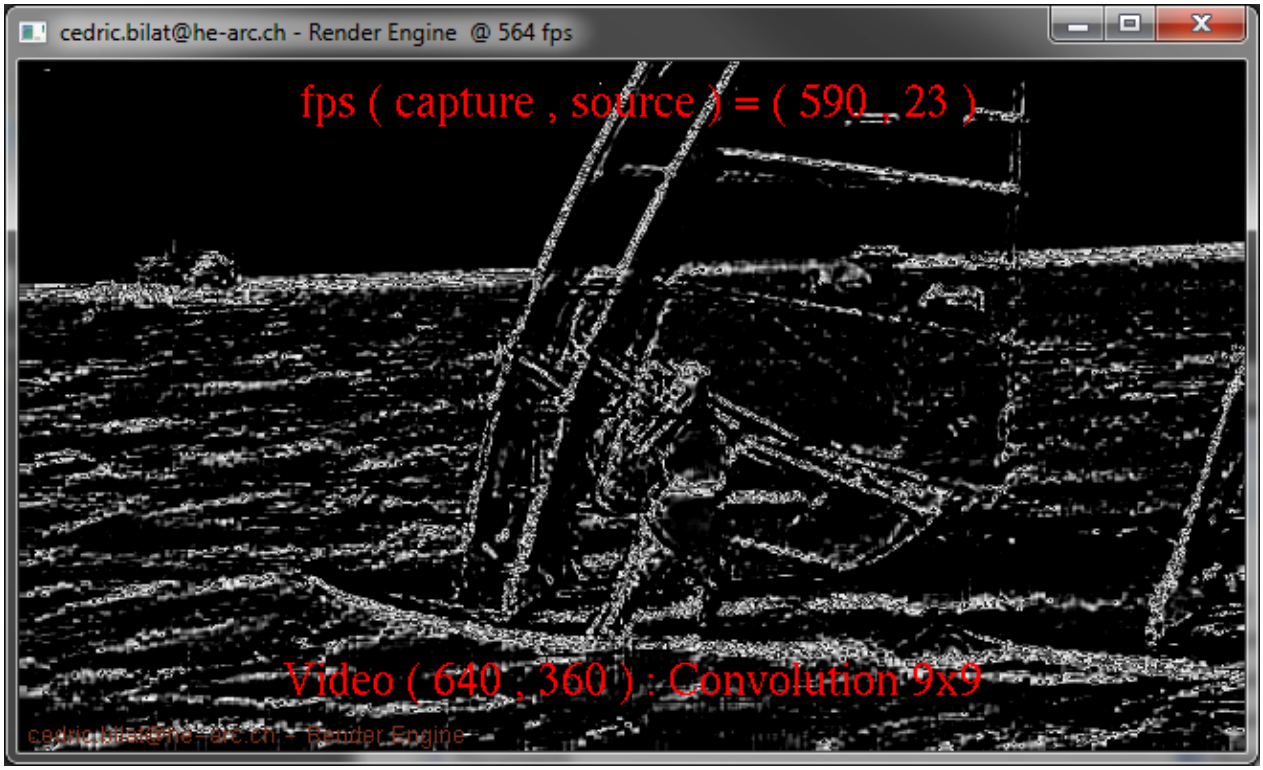
En changeant l'association des signes comme dans le cas 1, on obtient un balayage sur les quatre régions géographique avec les formules suivantes :

$\sum_{u,v} \text{noyau}[\ (ss + vk) + u] * \text{tex2d}_{\text{image}}[(j + u, i + v)]$	Sud Est	(SE)
$\sum_{u,v} \text{noyau}[\ (ss + vk) - u] * \text{tex2d}_{\text{image}}[(j + u, i - v)]$	Sud Ouest	(SO)
$\sum_{u,v} \text{noyau}[\ (ss - vk) + u] * \text{tex2d}_{\text{image}}[(j - u, i + v)]$	Nord Est	(NE)
$\sum_{u,v} \text{noyau}[\ (ss - vk) - u] * \text{tex2d}_{\text{image}}[(j - u, i - v)]$	Nord Ouest	(NO)

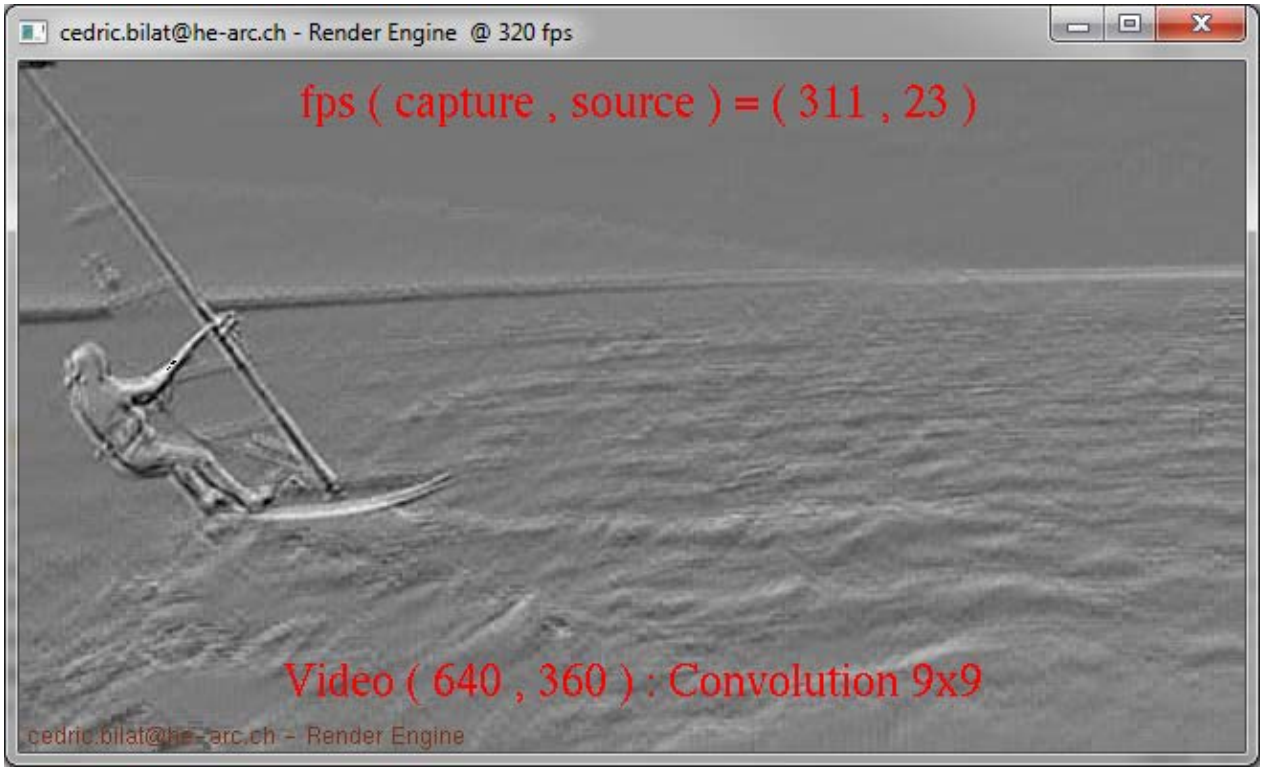
Output *Exemples*



Amplification par translation+64



*Amplification par scaling *50*



Amplification affine [min,max] -> [0,255] a chaque image !
Réduit considérablement les fps !
Mais il suffirait de la faire pour une image !

Speedup

Quelques pistes :

(I1) Comparer le temps nécessaire à faire l'acquisition de l'image, au temps nécessaire au traitement. Quel est le ratio ? moitié-moitié, , ¼ ¾, …

Note : On ne peut pas obtenir plus de fps que ne le permet l'acquisition ! Cette information nous donne une *borne supérieure* que l'on ne pourra pas dépasser, mais qu'il s'agira d'approcher le plus possible !

(I2) Pour ce TP transformer les *FPS* en *ms* (milliseconde) peut être intéressant :

- Combien de ms pour l'acquisition ?
- Combine de ms pour le transfert sur GPU
- Combien de ms pour les kernels cuda ?

(I3) Tuner les *db,dg* différemment pour chacun des kernels. Il n'y a aucune raison que la valeur optimal soit la même pour tous vos kernels !

(I4) Essayer de parcourir le noyau de convolution de manière plus standard, en partant en hauzt à gauche, puis en défilant vers la droite, …

End