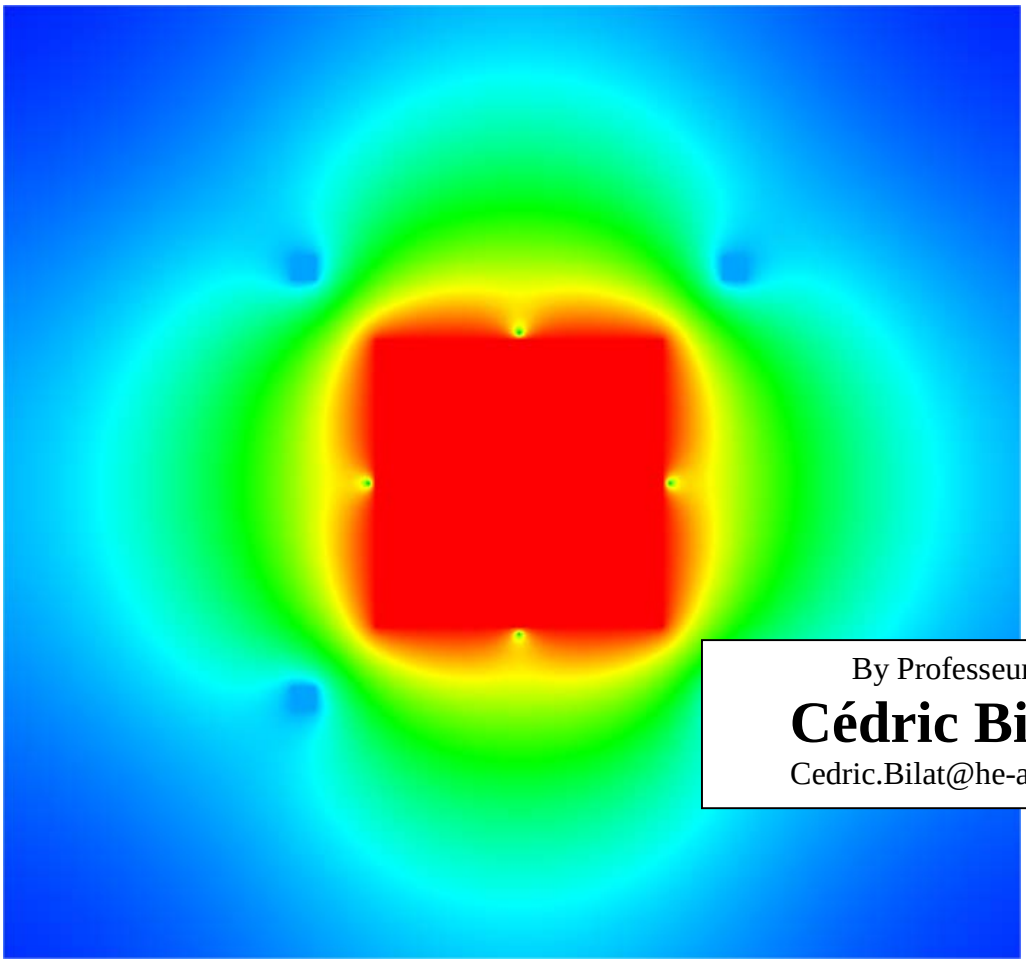
Problèmes Parallelisation



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

Heat Transfert

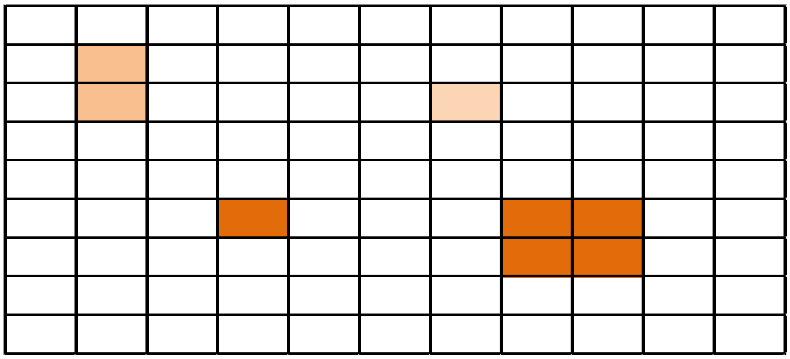
Contexte

Contexte

On s'intéresse à la diffusion de la chaleur sur une *surface*. Cette surface contient des *Heaters* et des *Coolers*. Chacun d'eux est une soure d'énergie et diffuse au tour de lui de la chaleur. La température du *Heater* peut varier au cours du temps, ou être constante. Un *cooler* n'est qu'un *heater* fournissant une température froide. Dans la suite on n'emploiera donc *Heater* au sens large.

Objectif

Connaissant la chaleur initiale en tout point de la plaque, et la position des *Heaters*, on aimerait calculer la diffusion de la chaleur induit par les *Heaters* au cours du temps en tout point de la plaque. Ce transfert de chaleur sera calculé de proche en proche, la surface étant discrétiser en *pixel*.

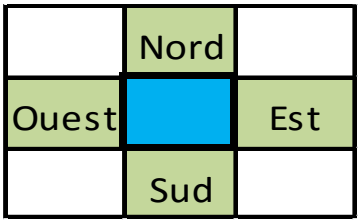


Surface discrétisée en pixel, avec des Heaters de différentes températures

Modèle

L'objectif n'est pas ici ne donner le modèle physique le plus fin et le plus réaliste, mais de donner un modèle simple permettant d'entrainer les techniques de parallélismes !

Notre modèle travaillera de proche en proche, chaque cellule de la grille influencera la température des cellules qui lui sont directement voisine. On se contentera d'un voisinage à 4 voisins comme suit :



Un **pixel** influencera ses 4 voisins cardinaux :

Nord, **Sud**, **Est** et **Ouest**.

Plusieurs modèles de propagation peuvent être utilisés. Ici on en propose deux très simples. Par ailleurs, on supposera que les *Heaters* sont constant au cours du temps. Mais rien ne vous empêche de les faire varier selon une fonction temporelle.

Modèle 1

Voici une règle de propagation simple :

$$\boxed{T_{new} = T_{old} + \sum_{\text{voisins}} k\left(T_{\text{Voisin}} - T_{old}\right)} \qquad (*)$$

Noton que dans ce modèle, la température varie dans $[0,1]$. La température maximum est ici 1 et la température minimum 0. Le paramètre k est une constante de vitesse de propagation de la chaleur, $k \subset [0, 0.25]$. Plus k est grand, plus la chaleur se diffusera rapidement dans la plaque. Le modèle (*) peut aussi se récrire :

$$\boxed{T_{new} = T_{old} + k\left(T_{sud} + T_{nord} + T_{est} + T_{ouest} - 4T_{old}\right)} \qquad (**)$$

Modèle 2

Une autre règle de propagation simple sans paramètre pourrait être :

$$\boxed{T_{new} = \frac{\sum_{\text{voisins}} \left(T_{\text{Voisin}} + T_{old}\right)}{2\left(\text{voisins}\right)}} \qquad (\#)$$

Implémentations

Indications

Utiliser quatre images

- ***ImageHeater*** Contient uniquement la température des *Heaters*, le reste à zéro
- ***ImageInit*** Contient la température initiale de la grille en tout point.
- ***ImageA*** *ImageA = ImageHeater + ImageInit*
- ***ImageB*** *ImageA* à laquelle on a appliqué le modèle de diffusion (**)

Ensuite on fusionne *ImageHeater* avec *ImageB*, et on applique le modèle de diffusion (**) à *ImageB* qui deviendra *ImageA*, etc. Tantôt *ImageA* est un **input**, tantôt un **output**. Idem pour *ImageB*. Le rôle de *ImageA* et *ImageB* croise au cours du temps.

Utiliser une variable

NB_ITERATION_AVEUGLE

qui est le nombre d'itération où on calcul la diffusion sans l'afficher! Si NB_ITERATION_AVEUGLE = 1, on affiche toujours le résultat de la diffusions.

On vous propose

NB_ITERATION_AVEUGLE = 50.

Libre à vous de faire varier ce paramètre.

Définition

Appelons **E** l'opérateur d'écrasement entre *ImageHeater* et une *Image* (les *Heaters* écrase le reste)
Appelons **D** l'opérateur de diffusion thermique modéliser par (**) ou (#).

Scénario

Initialisation

```
ImageInit=E(ImageHeater, ImageInit)
ImageA=D(ImageInit)
ImageA=E(ImageHeater, ImageA)
```

Itération (Une)

```
ImageB=D(ImageA)
ImageB=E(ImageHeater, ImageB);
ImageA=D(ImageB)
ImageA=E(ImageHeater, ImageA)
```

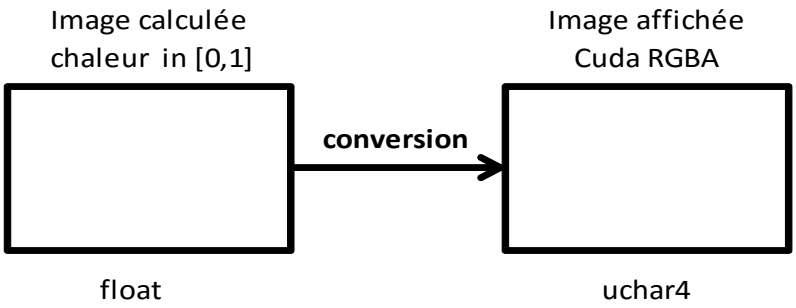
Type

Les images seront stockées à l'aide de tableau de *float*.

Pour l'implémentation avec l'*APIGLImageCuda*, il s'agira de convertir l'image calculée (en *float**) en une image *Cuda* RGBA (*uchar4**), dont on reçoit le pointeur GPU dans la méthode :

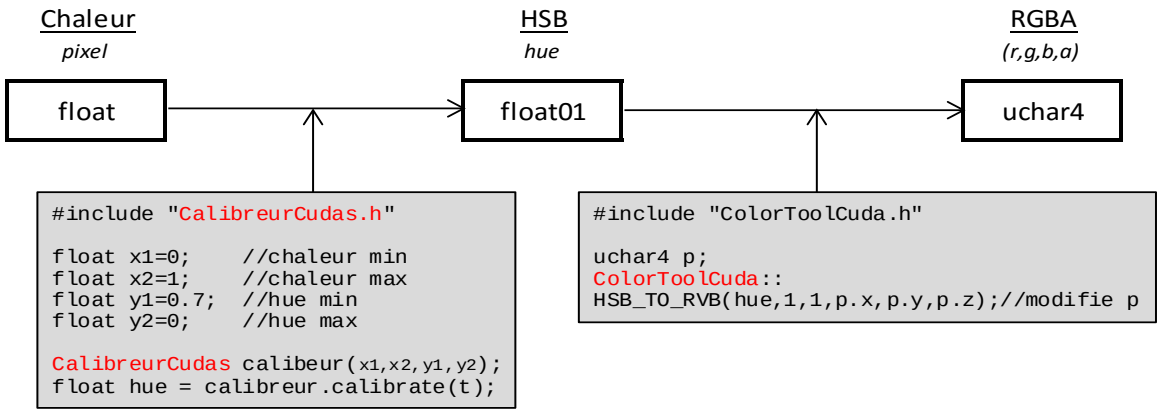
fillImageGL(uchar4 ptrDevImageGL, int w, int h)*

Dans le cas des textures en cuda, les textures vont "binder" les images en *float**.

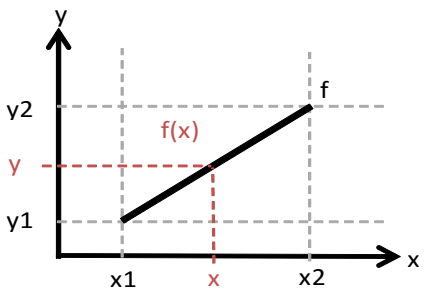


Pour être le plus efficace possible, en *Cuda* ce travail de conversion sera effectué de manière parallèle dans un kernel.

Indications :



Calibreur :

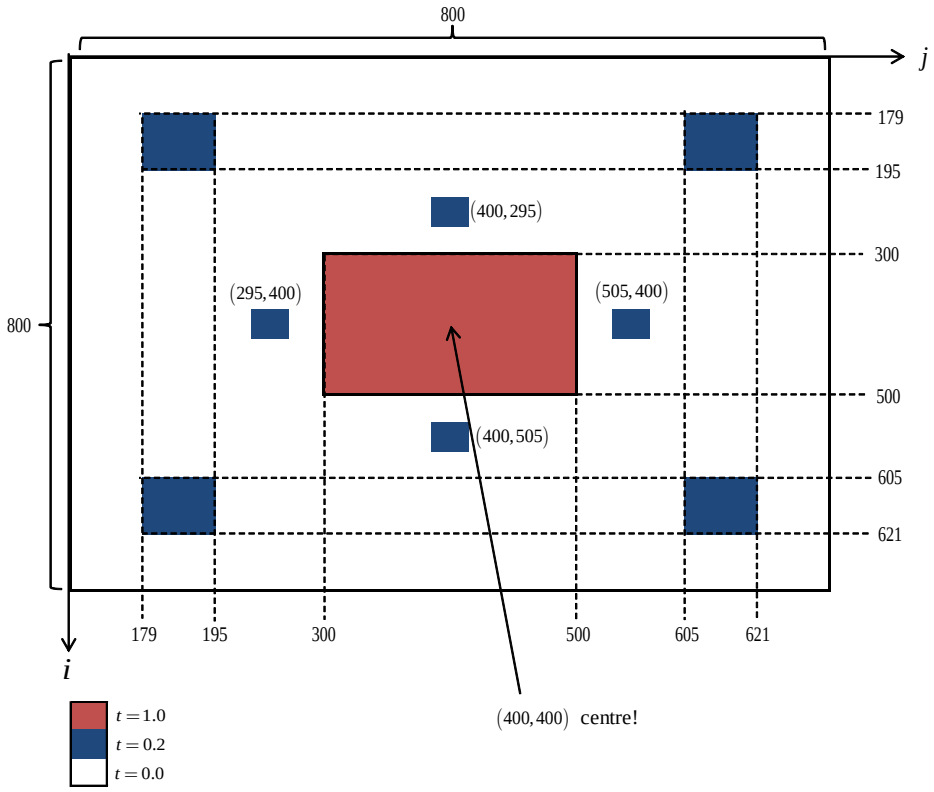


Conseil :

Ne travaillez pas sur le spectre HSB complet $[0,1]$ mais sur l'intervalle $[0.7,0]$. Notez aussi le croisement!

Start Point

Prenez comme *imageInit* $w \times h = 800 \times 800$, une grille de température uniformément remplie à 0.0. Pour les *heaters*, on propose :



Mais libre à vous de créer un *startPoint* plus attractif!

Implémentation

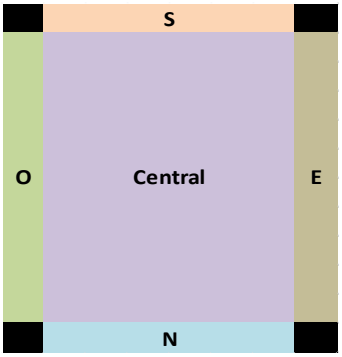
Contraintes Cuda

Effectuer deux implémentations différentes en *Cuda*

- Avec ***global memory GM***
- Avec ***texture***

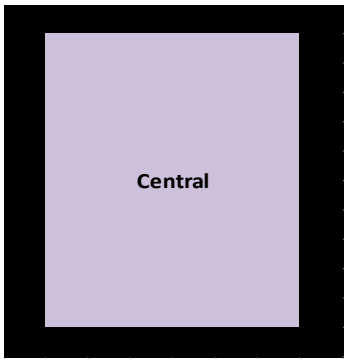
Indication

(I1) Dans le cas d'une implémentation coté ***Host*** ou ***Device*** avec ***Global Memory GM***, 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 de mise à jour pour les pixels du bords qui garderont leur chaleur originale !

Vu la grandeur de l'image, cette bordure de 1 pixel de *large* autour de l'image est en effet négligeable. Ne pas tenir compte de cette bande ne va pas péjorer la qualité de la cartographie globale du diffusément thermique, mais simplifier grandement votre code !



Updatez donc seulement les pixels de la partie centrale, ayant tous la même propriété de posséder 4 voisins dans les 4 points cardinaux !

(I2) Utiliser la version *bitmap* de l'API Image fournie. Il n'est pas nécessaire ici d'utiliser la version *fonctionnelle*.

(I3) N'instancier qu'une seule fois côté *Host* le ***CalibreurCuda***

Stocker le comme attribut de la classe *HeatTransfertImageCudaMOO.*

Pour la construction (problème de syntaxe C++) , voir l'annexe.

Passer ensuite par référence l'objet *calibreurCuda* à votre *launcher* de *Kernel.*

Passer le ensuite par valeur à votre *Kernel* de calcul Cuda.

Note

Comme l'objet *calibreurCuda* ne contient pas d'attribut de type pointeur, il occupe une suite d'octet adjacents et contigus en Ram, il peut donc être copié byte par byte automatiquement coté *Device* (car sa taille total est connu par le système).

Vous n'avez donc pas besoin d'effectuer du MM (memory management) de type *cudaMalloc* avec lui. Il se passe comme un type simple !

(I4) Fabriquer coté *Host* de manière séquentielle les deux images :

ImageInit
ImageHeater

Puis par memory management MM copier ces deux images côtés *Device.* Stocker les 4 pointeurs comme attributs de la classe

HeatTransfertImageCudaMOO

L'allocation se fait dans le constructeur, et la destruction dans le destructeur de cette classe

HeatTransfertImageCudaMOO

On transmet ensuite les 2 pointeurs *Devices*

float* ptrDevImageInit
float* ptrDevImageHeater

aux objets en ayant besoin.

(I5) Utilisez 3 kernels

Kernel 1 : ***diffusion***

Input : *ptrDevImageA*

Output : *ptrDevImageB*

Important : On lit dans l'input, on écrit dans l'output (et on écrit surtout pas dans l'input !)

Au prochain passage, *l'input* devient *l'output*. On échange les rôles. Système de *double buffering !*

Kernel 2 : ***ecrasement***

Input : image output du kernel précédent de diffusion

Output : image input modifiée

Important : algorithme sur place !!

Les *heaters* reprennent leur place au–dessus de l'image thermique calculé (car eux ne changent jamais de température)

Kernel 3 : ***toScreenImageHSB***

Input : output du kernel précédent d'écrasement

Outputs : p*trDevImage* updatée (où p*trDevImage* est le pointeur de la zone mémoire partagée par *OpenGL* et *Cuda*).

Important On lit l'input, on converti t en HSB et on stocke dan *ptrDevImage*

Input type: **float***
Output type : **uchar4***

Observer le changement de type !

L'animation est obtenue en itérant sur ces 3 kernels. Attention au changement de buffer pour le *kernel 1*. Utilisez un flag indiquant quel image contient le résultat du *kernel 2*. Cette image sera alors l'input du *kernel 1*

bool isImageAInput ;

Amélioration

Placer des *Heaters* au cours du temps de manière évènementielle avec la souris (Cf *BilatGLImage Advanced Guide*)

Validation

Effectuer les variations suivantes :

- Taille de l'image (rectangulaire, pas carrée !)
- dg et db

Pour taille de l'image prenez par exemple

```
int dw = 16 * 80;
int dh = 16 * 60;
```

Pour les contraintes à satisfaire sur *dg* et *db*, utilisez

```
Devices ::printAll() ;
```

Speedup

Mesurer les coefficients de *speedup* des différentes implémentations.

Pour canevas, utiliser le document

speedup_simple.xls.

Au besoin adapter ce canevas.

Annexe

Rappel C++ *(par l'exemple)*

Comment construire un attribut de type Objet ?

Exemple 1

Soit B avec un attribut de type « tools » T

Header

```
class T
        {
        public :
                T(float x);
        …
        private:
                float x;
        }

class B
        {
        public :
                B(…) ;
        …
        private :
                // Tools
                T t ;
        }
```

Implémentation

```
B ::B(…) :t(3.14)
        {
        …
        }
```

Exemple 2

Même exemple que ci-dessus, mais en plus B dérive de A

Header

```
class B : A
        {
        public :
                B(…) ;
        …
        private :
                T t ;
        }
```

Implémentation

```
B ::B(…) :A(…) , t(3.14)
        {
        …
        }
```

Le séparateur est la virgule !

End