Travaux dirigés de Programmation CUDA

Exercice 1.

Ecrire la fonction

`void vecADD(float *A, float *B, float *C, int n)`

qui calcule la somme des vecteurs A et B de dimension n et range le résultat dans le vecteur C.
Cette fonction fera appel à un kernel avec des blocs de 1024 threads.

Exercice 2

On veut écrire la fonction

`void blur(unsigned char *in, unsigned char *out, int width, int height)`

qui floute l'image pointée par `in` de (`height` $\times$ `width`) pixels et place le résultat à l'adresse pointée par `out`.

Un pixel de l'image résultat est égal à la moyenne des voisins du pixel correspondant dans l'image initiale. Les voisins considérés sont les pixels qui se trouvent dans un carré de dimension $2 \times$ `BLURSIZE + 1`. Remarque : les pixels qui se trouvent proches des bordures ont moins de voisins que les autres.

Le code de la fonction est donné ci-dessous. Ecrire le kernel `blurkernel`.

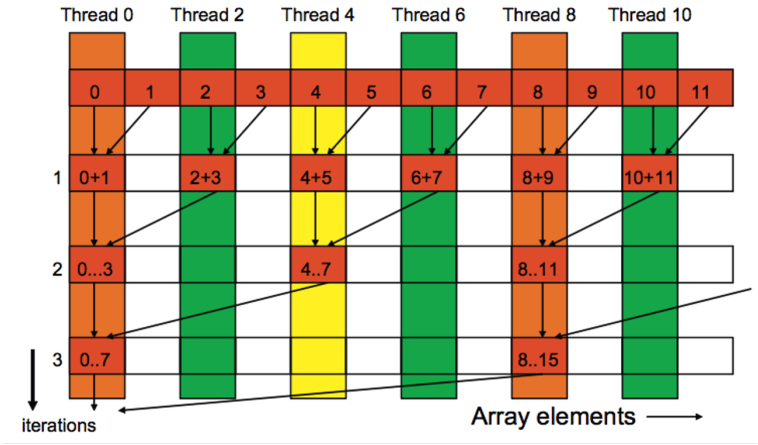
```
#define BLURSIZE 3
#define BSIZE 32

void blur(unsigned char *in, unsigned char *out, int width, int height)  {
    int numbw = (width + BSIZE - 1) / BSIZE;
    int numbh = (height + BSIZE - 1) / BSIZE;
    dim3 gdim(numbw, numbh, 1);
    dim3 bdim(BSIZE, BSIZE, 1);
    unsigned char *din, *dout;
    int bytes = width*height*sizeof(unsigned char);

    cudaMalloc((void **)&din, bytes) ;
    cudaMalloc((void **)&dout, bytes) ;
    cudaMemcpy(din, in, bytes, cudaMemcpyHostToDevice) ;

    blurkernel<<<gdim, bdim>>>(din, dout, width, height);

    cudaMemcpy(out, dout, bytes, cudaMemcpyDeviceToHost);
    cudaFree(din); cudaFree(dout);
}
```

Exercice 3

On considère la fonction suivante qui calcule la somme des éléments du vecteur vec de taille size $\leq$ 1024, égale à une puissance de 2, et la renvoie dans sum :

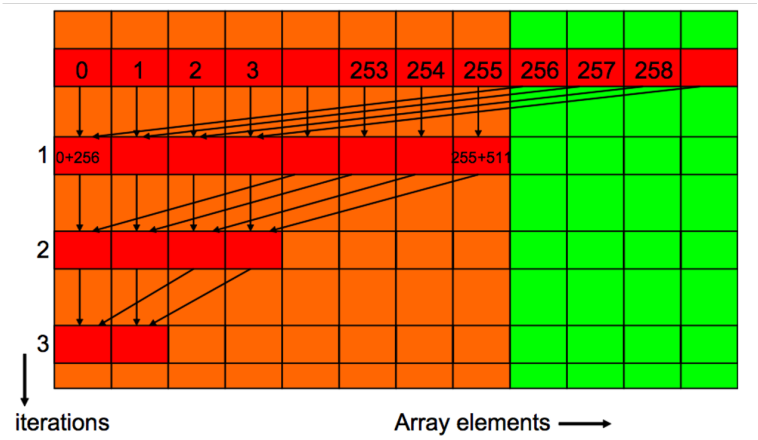
```
void reduce(float *vec, float *sum, int size)  {
    float *d_vec;
    int bytes = size*sizeof(float);

    cudaMalloc((void **)&d_vec, bytes) ;
    cudaMemcpy(d_vec, vec, bytes, cudaMemcpyHostToDevice) ;

    kreduce<<<1, size>>>(d_vec, size);

    cudaMemcpy(sum, d_vec, sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_vec);
}
```

Cette fonction appelle le kernel kreduce qui calcule la somme selon le schéma ci-dessous, de sorte que le résultat final se trouve dans d_vec[0].



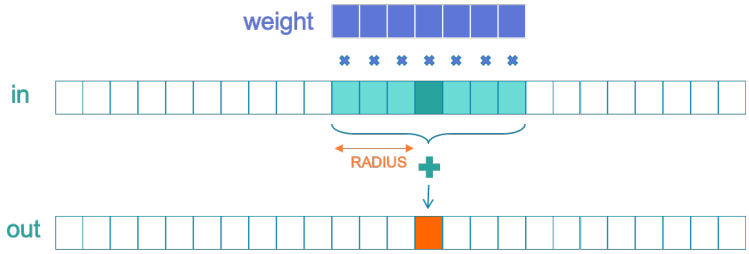
Question 1. Ecrire ce kernel, puis analyser la divergence des flots de contrôle.

Question 2. Ecrire une nouvelle version du kernel, inspirée du schéma suivant. Analysez la divergence.



Exercice 4

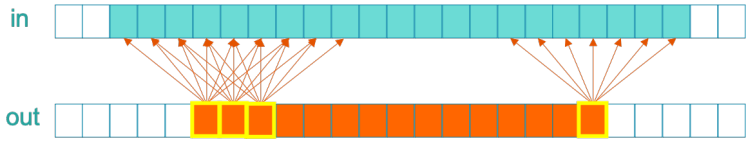
On veut réaliser une opération de type *stencil* dans laquelle chaque élément du vecteur résultat (`out`) est la somme pondérée de lui-même et ses voisins à une distance inférieure à `RADIUS` dans le vecteur initial (`in`). Les poids sont spécifiés par le vecteur `weight`. Pour les points proches des bordures, les voisins manquants sont supposés égaux à 0.



Ecrire la fonction suivante, qui fait appel à un kernel :

```
void convolution(float *in, float *out, float *weight, int size);
```

Le rayon sera défini comme une constante symbolique. Le tableau `weight` sera placé dans la mémoire constante du GPU. Les éléments du tableau `in` étant réutilisés par plusieurs threads dans un bloc, on le copiera dans la mémoire partagée.

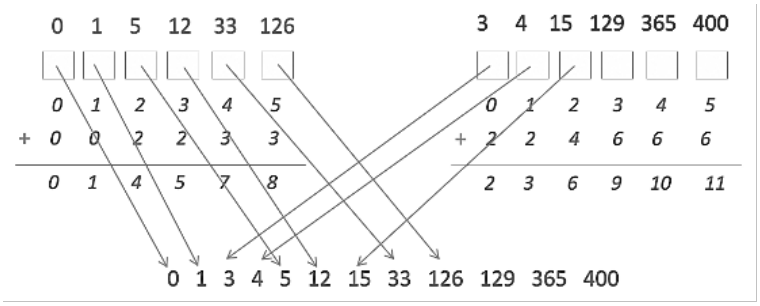


Exercice 5

On veut écrire un kernel qui réalise la fusion de deux vecteurs triés (sans doublons). Pour cela, pour chaque élément d'un des deux vecteurs :

- On cherche la position qu'il aurait dans le second vecteur, en utilisant la fonction suivante :
  ```
  __device__ int search(int val, int *vect, int size) ;
  ```
- On ajoute à cela sa position dans son vecteur d'origine.
- Le résultat est la position de l'élément dans le vecteur fusionné.

Exemple :



Les paramètres de ce kernel sont :

- pointeur sur le vecteur résultat
- pointeurs sur les vecteurs à fusionner
- taille des vecteurs à fusionner (le vecteur résultat étant deux fois plus long). On suppose que cette taille est inférieure à 1024.

Le kernel sera lancé sur un seul bloc de threads. Chaque thread traite un élément de chaque vecteur en entrée.

Exercice 6

Un problème de type *N bodies* consiste à simuler l'évolution d'un système de corps (ou objets) dont chacun interagit en permanence avec tous les autres. Par exemple, dans le domaine de l'astrophysique, un corps peut représenter une étoile : les étoiles s'attirent les unes les autres selon la force gravitationnelle.
Chaque objet/corps b_i est caractérisé par sa masse m_i, sa position $\mathbf{x}_i$[1], sa vitesse $\mathbf{v}_i$ et son accélération $\mathbf{a}_i$. La force $\mathbf{f}_{ij}$ appliquée par l'objet b_j à l'objet b_i est définie par :

$$\mathbf{f}_{ij} = G.\frac{m_i \dot{m}_j}{||\mathbf{r}_{ij}||^2}.\frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}$$

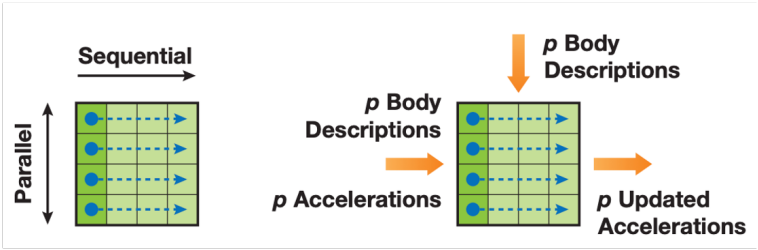
avec $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$. G est la constante gravitationnelle.
La force totale $\mathbf{F}_i$ subie par l'objet b_i est :

$$\mathbf{f}_i = \sum_{0 \leq j < N, j \neq i} \mathbf{f}_{ij} = G.m_i. \sum_{0 \leq j < N, j \neq i} \frac{m_j.\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||^3}$$

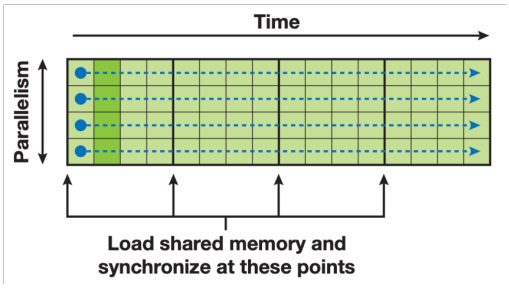
Pour limiter l'amplitude de la force lorsque deux corps sont très proches l'un de l'autre, on introduit généralement un facteur d'atténuation ϵ^2, de sorte que la force est approchée par :

$$\mathbf{f}_i \approx G.m_i. \sum_{0 \leq j < N} \frac{m_j.\mathbf{r}_{ij}}{||(\mathbf{r}_{ij}||^2 + \epsilon^2)^{3/2}}$$

L'accélération de l'objet b_i est calculée par $\mathbf{a}_i = \mathbf{F}_i/m_i$ et est utilisée pour mettre à jour sa vitesse et sa position.
Une approche possible pour la simulation est une technique de force brute qui évalue les interactions au sein de toutes les paires possibles d'étoiles, avec une complexité en (N^2) si N est le nombre d'étoiles. On peut voir cette approche comme le calcul de chaque $\mathbf{f}_{ij}$ dans une grille $N \times N$. La force totale $\mathbf{F}_i$ subie par l'objet b_i (ou son accélération $\mathbf{a}_i$) est alors la somme de tous les éléments de la ligne i.
On pourrait calculer les N^2 éléments en parallèle, mais pour limiter les accès à la mémoire globale, on va procéder autrement. On introduit la notion de *tuile* de dimensions $p \times p$. Les données de seulement $2p$ objets sont nécessaires pour calculer les p^2 éléments $(\mathbf{f}_{ij})$ de la tuile. Ces données peuvent être copiées dans la mémoire partagée. Par ailleurs, on peut calculer l'effet des forces sur les accélérations au fur et à mesure du calcul. En outre, pour optimiser la réutilisation des données, on va organiser le calcul de sorte que p threads, constituant un bloc, calculent la tuile, chacun une ligne.

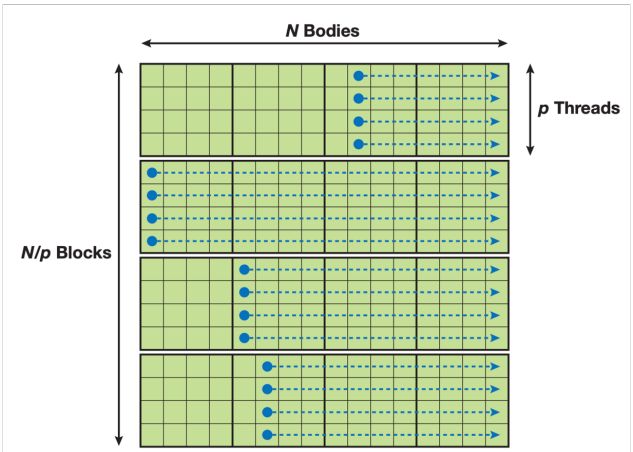


Chaque bloc de threads va calculer successivement toutes les tuiles couvrant les mêmes lignes. Avant chaque nouvelle tuile, il faudra copier les données correspondantes dans la mémoire partagée.



La simulation complète appelle, à chaque itération, un kernel qui calcule la nouvelle accélération de chaque objet. Ce kernel est lancé pour N/p blocs de p threads qui calculent l'ensemble des éléments de la grille :

[1] Les vecteurs 3D sont notés en **gras**.



Ecrire le kernel. On pourra utiliser la fonction suivante qui calcule l'accélération qui résulte de la force appliquée à l'objet b_i par l'objet b_j. Les objets sont représentés par des `float4`: les champs `.x`, `.y` et `.z` définissent la position, et le champ `.w` la masse.

```
__device__ void interaction(float4 *bi, float4 *bj, float3 *ai){
    float3 r;
    r.x = bj.x - bi.x;
    r.y = bj.y - bi.y;
    r.z = bj.z - bi.z;
    float dist_sqr = r.x*r.x + r.y*r.y + r.z*r.z + EPS2;
    float dist_sixth = dist_sqr + dist_sqr * dist_sqr;
    float inv_dist_cube = 1.0f / sqrtf(dist_sixth);
    float s = bj.w * inv_dist_cube;
    ai.x += r.x * s;
    ai.y += r.y * s;
    ai.z += r.z * s;
}
```