**Ressources Allocation**

Avec l’API vulkan la gestion de la mémoire (graphique) est **gérée par l’application**, au temps en profiter. Il y a plusieurs espaces mémoire utilisable, plus ou moins performant. Il y a notamment **2 espaces**, l’espace sur la carte graphique (**DEVICE\_LOCAL**), et celui visible par l’application(**HOST\_VISIBLE**). Le premier n’est pas directement modifiable par l’application mais est plus rapide. Le second est modifiable par l’application mais est plus lent.

Pour modifier le premier **on doit créer deux buffers**, un **visible par l’application** et un **device\_local**. On écrit ensuite les données dans le buffer visible par l’app puis **on le copie vers le device\_local**. Puis on **supprime le host\_visible** puisqu’il ne sert plus rien.

Données à copier

HOST\_VISIBLE **BUFFER**

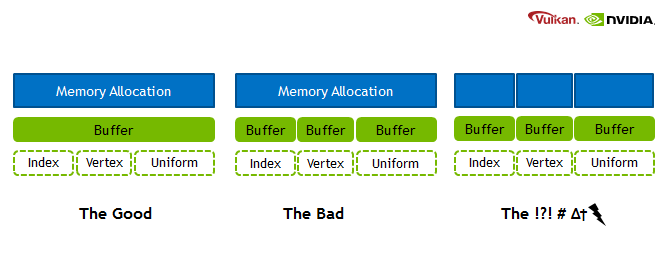
DEVICE\_LOCAL **BUFFER**

***vkMapMemory()***

vkCopyBuffer(Buffer\* src, Buffer\* dst, uint32\_t size, uint32\_t offset)

VULKAN est très performant quand le nombre de buffer est faible et en utilisant les offsets.

En effet il faut essayer de créer le moins possible de buffer (voir si dessous).



L’objectif est en fait de créer un **« gros » buffer** qui contient tout les vertex, indices et uniform buffer est autre. On va en fait simuler plusieurs buffers avec l’utilisation des offset. On se dit donc que ca va être la galère parce que je n’affiche pas tjrs le même nombre d’objet, qu’il y a des objets qui vont changer de taille et tout. On se rend compte qu’on ne veut pas détruire le gros buffer et en créer un a chaque fois qu’on ajoute un objet.

L’une des solutions pour éviter ce genre de problème c’est d’anticiper un minimum la taille du buffer.

Mon idée est donc de créer les objets en amont, puis créer le buffer de la bonne taille puis copier les données des objets apres.

Exemple :

std ::vector<Object\*> objects.

Object.pushback(new ObjectX(param)) ;

…

Renderer.init();

For (size\_t i=0;i<objects.size();i++){

objects[i]->load();

}