GPGPU: Utiliser la carte graphique pour accélérer vos applications

Maël Naccache Tüfekçi
@MaelTufekci

zenika
<animés par la passion>

Octobre 2019

DEVFEST NANTES 2019

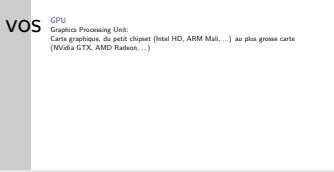
Bonjour,
Je m'appelle Maël et je suis consultant chez Zenika. Vous pourrez me retrouvez sur notre stand après le talk.
Aujourd'hui, je vais vous parlez de GPGPU, pourquoi et surtout comment.

Définitions

Commençons par quelques définitions.

GPU

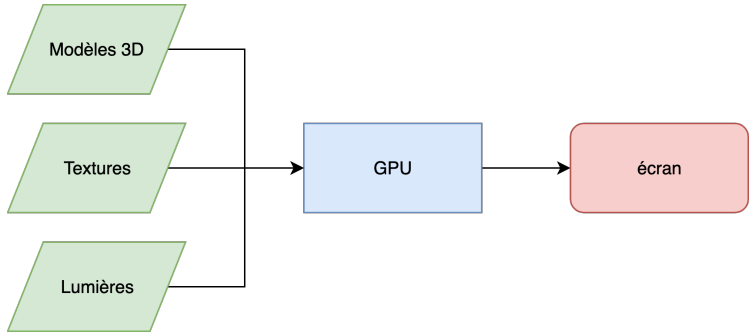
Graphics Processing Unit:
Carte graphique, du petit chipset (Intel HD, ARM Mali, ...) au plus grosse carte (NVidia GTX, AMD Radeon, ...)



Un GPU, ou carte graphique dans la langue de molière est un composant informatique dédié au rendu de graphiques.
Un GPU peut prendre différente forme, de simple puce intégrée directement au CPU, à la grosse carte dediée consommant des centaines de watt.
Dans ce talk, nous ne ferons pas la distinction entre les différents types.

GPU

Graphics Processing Unit:
Carte graphique, du petit chipset (Intel HD, ARM Mali, ...) au plus grosse carte (NVidia GTX, AMD Radeon, ...)

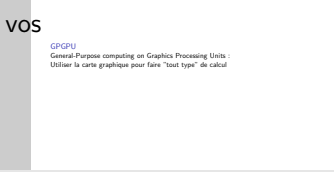


GPU
Graphics Processing Unit:
Carte graphique, du petit chipset (Intel HD, ARM Mali, ...) au plus grosse carte (NVidia GTX, AMD Radeon, ...)

On imagine un GPU un peu comme ça : On lui envoie des modèles 3D, des textures et d'autres données, et il nous ressort des graphismes à l'écran

GPGPU

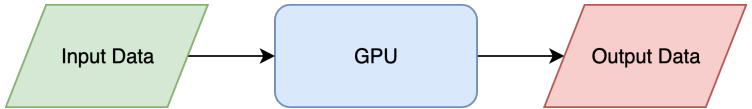
General-Purpose computing on Graphics Processing Units :
Utiliser la carte graphique pour faire "tout type" de calcul



Le GPGPU, pour General Purpose Computing on Graphics Processing Units est le nom donnée au fait de faire n'importe quel type de calcul sur un GPU et non plus juste des calculs destinée à afficher des graphismes à l'écran.

GPGPU

General-Purpose computing on Graphics Processing Units :
Utiliser la carte graphique pour faire "tout type" de calcul



L'idée est d'utiliser le GPU comme un processeur générique à qui ont fourni des données d'entrée, un programme, et on resoit des données de sortie.

Pourquoi faire du GPGPU ?

- CPU : 64 cœurs maximum

Mais pourquoi faire du GPGPU ?
Aujourd'hui, les tous derniers processeurs arrivent à avoir 64 cœurs. Et on parle de AMD Epyc a plus de 4000€.

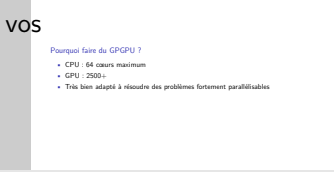
Pourquoi faire du GPGPU ?

- CPU : 64 cœurs maximum
- GPU : 2500+

En revanche une carte graphique haut de gamme comme une RTX 2080 à plus de 2500 cœurs pour un prix de 800 à 900€.

Pourquoi faire du GPGPU ?

- CPU : 64 cœurs maximum
- GPU : 2500+
- Très bien adapté à résoudre des problèmes fortement parallélisables

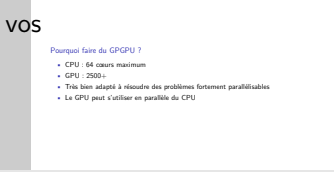


Évidemment, chacun de ces cœurs sont beaucoup moins puissants que ceux d'un CPU, mais le fait d'en avoir beaucoup fait que le GPU est très éfficaces quand il s'agit de traiter beaucoup de données en parrallèle.

C'est en effet pour cela qu'il a été designé à la base, le rendu 3d temps réel demandant beaucoup de calculs similaire et répétitif.

Pourquoi faire du GPGPU ?

- CPU : 64 cœurs maximum
- GPU : 2500+
- Très bien adapté à résoudre des problèmes fortement parallélisables
- Le GPU peut s'utiliser en parallèle du CPU



Enfin, votre GPU reste inutilisé une majeur partie du temps alors que celui-ci peut être utiliser en parrallèle du CPU. Pourquoi ne pas lui déléguer certaines tâches ?

Un peu d'histoire

Un peu d'histoire

Avant d'aller plus loin, attardons nous un peu sur comment nous en sommes arriver la.

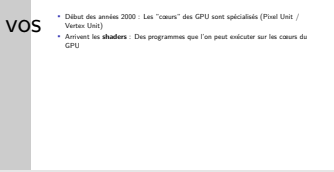
- Début des années 2000 : Les "cœurs" des GPU sont spécialisés (Pixel Unit / Vertex Unit)

- Début des années 2000 : Les "cœurs" des GPU sont spécialisés (Pixel Unit / Vertex Unit)

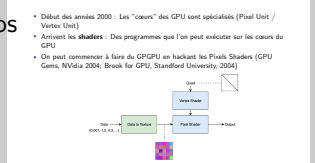
Le GPGPU est un phénomène relativement récent car nous n'avions pas les capacité techniques d'en faire.
Ce qu'il faut savoir c'est que au début des années 2000, chacun des cœurs du GPU était spécialisé. Soit en traitement de l'image (Pixel Unit), soit en traitement de la géométrie (Vertex Unit).

On pouvait configurer leurs opérations mais pas les programmer.

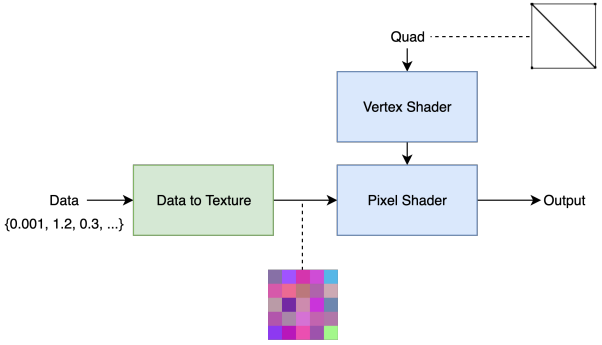
- Début des années 2000 : Les "cœurs" des GPU sont spécialisés (Pixel Unit / Vertex Unit)
- Arrivent les **shaders** : Des programmes que l'on peut exécuter sur les cœurs du GPU



Vers 2001, on commence à pouvoir utiliser des shaders: Des programmes qui vont s'éxécuter sur ces cœurs du GPU. Leurs capacités sont limités en fonctions du cœurs sur lequels ils s'éxécutent.



- Début des années 2000 : Les "cœurs" des GPU sont spécialisés (Pixel Unit / Vertex Unit)
- Arrivent les **shaders** : Des programmes que l'on peut exécuter sur les cœurs du GPU
- On peut commencer à faire du GPGPU en hackant les Pixels Shaders (GPU Gems, NVidia 2004; Brook for GPU, Standford University, 2004)



En 2004, les capacités des shaders ont suffisamment évolué pour que certains remarques que l'on peut commencer à faire du calculs génériques en "hackant" les pixels shaders.
L'idée est simple: à l'écran, on ne rend que un carré sur lequel on va appliqué une texture. Cette texture est en fait nos données, et c'est le pixels shaders qui va utiliser cette texture comme donnée d'entrée et réaliser les opérations que l'on souhaite dessus.
On récupère ensuite l'image de sortie et on la retransforme en image.
Cette technique à de gros inconvénient: On passe par plein d'étapes graphiques qui sont inutiles pour notre usage et on utilise très mal les cœurs de notre GPU. Les cours traitant la géométrie ne font presque rien.

- Depuis ∼2006, un GPU est composé de Compute Unit (core, shading unit)

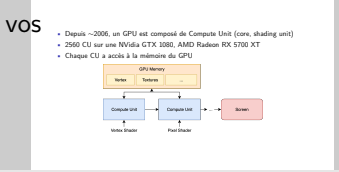
- Depuis ∼2006, un GPU est composé de Compute Unit (core, shading unit)

En 2006, les constructeurs de carte graphique finissent par supprimer la spécialisation des cœurs et utilise qu'un type de cœur unique, les "compute unit".
Les CU sont capables d'exécuter n'importe quel type de shader. Chaque CU à donc un accès à la mémoire du GPU (ce qui était plus ou moins réserver au pixel unit).

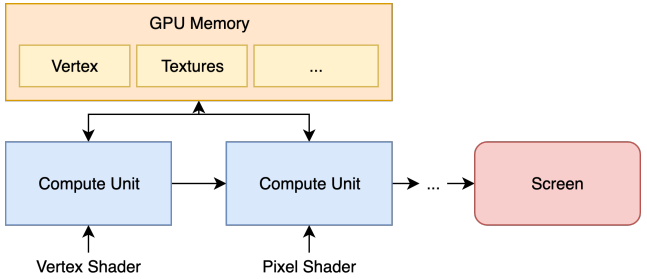
- Depuis ∼2006, un GPU est composé de Compute Unit (core, shading unit)
- 2560 CU sur une NVidia GTX 1080, AMD Radeon RX 5700 XT

- Depuis ∼2006, un GPU est composé de Compute Unit (core, shading unit)
- 2560 CU sur une NVidia GTX 1080, AMD Radeon RX 5700 XT

En 2006, les constructeurs de carte graphique finissent par supprimer la spécialisation des cœurs et utilise qu'un type de cœur unique, les "compute unit".
Les CU sont capables d'exécuter n'importe quel type de shader. Chaque CU à donc un accès à la mémoire du GPU (ce qui était plus ou moins réserver au pixel unit).



- Depuis ~2006, un GPU est composé de Compute Unit (core, shading unit)
- 2560 CU sur une NVidia GTX 1080, AMD Radeon RX 5700 XT
- Chaque CU a accès à la mémoire du GPU



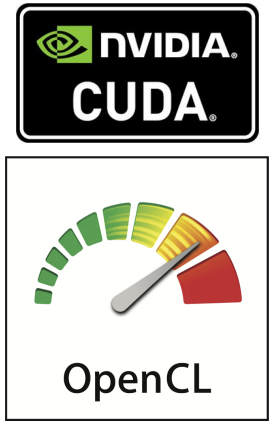
En 2006, les constructeurs de carte graphique finissent par supprimer la spécialisation des cœurs et utilise qu'un type de cœur unique, les "compute unit".
Les CU sont capables d'exécuter n'importe quel type de shader. Chaque CU à donc un accès à la mémoire du GPU (ce qui était plus ou moins réserver au pixel unit).

- En 2007, NVidia lance CUDA
- Première API dédiée au GPGPU

- En 2007, NVidia lance CUDA
- Première API dédiée au GPGPU
NVIDIA CUDA

Grâce à ces changements d'architecture, NVidia lance CUDA en 2007 qui est la première API dédié au GPGPU qui n'utilise pas le hack des pixels shader.

- En 2007, NVidia lance CUDA
- Première API dédiée au GPGPU
- En 2009, Khronos Group (OpenGL) lance OpenCL
- Permet de faire du GPGPU sur n'importe quelle carte et n'importe quel OS



Enfin, en 2009 sort OpenCL, à l'origine développé par Apple et confié à Khronos Group (responsable d'OpenGL). OpenCL est une spécification (comme OpenGL), il est donc libre à chaque constructeur de carte graphique / OS de réaliser une implémentions. Cela permet à OpenCL d'être supporté par toute les cartes et tout les OS.

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU

La totalité des API traditionnellement fait pour faire du graphisme supporte maintenant le fait du faire du calcul pur.
WebGL le supporte depuis avril 2019 mais via une extension qui n'est implémenté par presque personne.

En revanche, cela fait partie de intégrante de la prochaine API, webGPU.

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU
- Des gammes de produits et des outils dédiés au GPGPU : NVidia Telsa/Jetson, AMD Radeon Instinct, NVidia NSight Compute, AMD ROCm

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU
- Des gammes de produits et des outils dédiés au GPGPU : NVidia Telsa/Jetson, AMD Radeon Instinct, NVidia NSight Compute, AMD ROCm

On a maintenant des gammes de produits dédié entièrement au GPGPU, comme les AMD Instinct ou Nvidia Telsa qui sont des GPU fait pour être intégré sur des serveurs dans des datacenter.

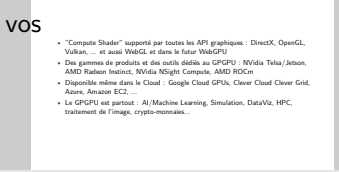
On a aussi des outils comme NVidia NSight qui permettent de faciliter le développement GPGPU, notamment lorsqu'il s'agit de faire du debugging.

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU
- Des gammes de produits et des outils dédiés au GPGPU : NVidia Telsa/Jetson, AMD Radeon Instinct, NVidia NSight Compute, AMD ROCm
- Disponible même dans le Cloud : Google Cloud GPUs, Clever Cloud Clever Grid, Azure, Amazon EC2, ...

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU
- Des gammes de produits et des outils dédiés au GPGPU : NVidia Telsa/Jetson, AMD Radeon Instinct, NVidia NSight Compute, AMD ROCm
- Disponible même dans le Cloud : Google Cloud GPUs, Clever Cloud Clever Grid, Azure, Amazon EC2, ...

Tout les grands fournisseurs de solutions cloud propose désormais des instances avec des GPU. Pas besoin d'aller investir dans une RTX2080 pour un one-shot. En revanche, pour le moment, les prix font qu'il est très intéressant d'acheter du matériel. C'est notamment du au prix des licence "serveur" de NVidia (et peut être AMD).

- "Compute Shader" supporté par toutes les API graphiques : DirectX, OpenGL, Vulkan, ... et aussi WebGL et dans le futur WebGPU
- Des gammes de produits et des outils dédiés au GPGPU : NVidia Telsa/Jetson, AMD Radeon Instinct, NVidia NSight Compute, AMD ROCm
- Disponible même dans le Cloud : Google Cloud GPUs, Clever Cloud Clever Grid, Azure, Amazon EC2, ...
- Le GPGPU est partout : AI/Machine Learning, Simulation, DataViz, HPC, traitement de l'image, crypto-monnaies...



Aujourd'hui, on ne s'en rend pas forcément compte mais on retrouve du GPGPU partout: La plupart des gros jeux vidéo en font pour certaines choses comme la gestion de l'AI ou la génération procédurale.
On en retrouve de manière générale dans tout ce qui est AI, machine learning, data mining, ... beaucoup dans le DataViz et le traitement de l'image. De plus en plus dans les simulations scientifiques et dans l'High Performance Computing.

Et bien sur, ça a été très médiatisé, on en retrouve dans le minage de crypto-monnaie.

Votre premier programme

Votre premier programme

Avant de commencer, il nous faut définir quelques pré-requis :

Avant de commencer, il nous faut définir quelques pré-requis :

On va maintenant passer au cœur du sujet, mais avant, il faut que l'on remplisse 2 pré-requis.

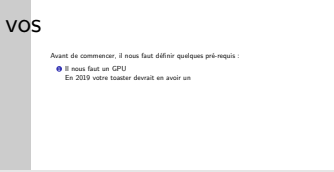
En premier lieux il nous faut une machine avec un GPU. En 2019, n'importe quoi devrait en avoir. Même mon MacBook Pro de 2014 à un chipset intégré Intel qui supporte le GPGPU.

Enfin, il faut que l'on choisisse une API.

Avant de commencer, il nous faut définir quelques pré-requis :

1. Il nous faut un GPU

On va maintenant passer au cœur du sujet, mais avant, il faut que l'on remplisse 2 pré-requis.
En premier lieux il nous faut une machine avec un GPU. En 2019, n'importe quoi devrait en avoir. Même mon MacBook Pro de 2014 à un chipset intégré Intel qui supporte le GPGPU.
Enfin, il faut que l'on choisisse une API.



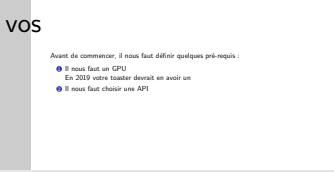
Avant de commencer, il nous faut définir quelques pré-requis :

1. Il nous faut un GPU
   En 2019 votre toaster devrait en avoir un

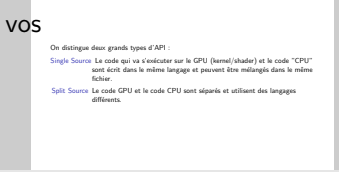
On va maintenant passer au cœur du sujet, mais avant, il faut que l'on remplisse 2 pré-requis.
En premier lieux il nous faut une machine avec un GPU. En 2019, n'importe quoi devrait en avoir. Même mon MacBook Pro de 2014 à un chipset intégré Intel qui supporte le GPGPU.
Enfin, il faut que l'on choisisse une API.

Avant de commencer, il nous faut définir quelques pré-requis :

1. Il nous faut un GPU
   En 2019 votre toaster devrait en avoir un
2. Il nous faut choisir une API



On va maintenant passer au cœur du sujet, mais avant, il faut que l'on remplisse 2 pré-requis.
En premier lieux il nous faut une machine avec un GPU. En 2019, n'importe quoi devrait en avoir. Même mon MacBook Pro de 2014 à un chipset intégré Intel qui supporte le GPGPU.
Enfin, il faut que l'on choisisse une API.



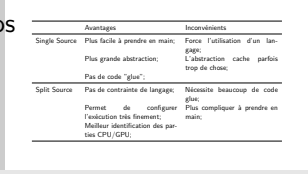
On distingue deux grands types d'API :

Single Source Le code qui va s'exécuter sur le GPU (kernel/shader) et le code "CPU" sont écrit dans le même langage et peuvent être mélangés dans le même fichier.

Split Source Le code GPU et le code CPU sont séparés et utilisent des langages différents.

Pour choisir notre API, on doit choisir entre deux grands types: Le single source ou le split source.
Avec une API single source, le code "CPU" et le code qui va être exécuté sur le GPU est écrit dans le même langage et peuvent être écrit dans le même fichier. Par exemple, CUDA peut être utiliser avec cette approche et utilise du C++ avec des annotations qui permettent de définir quels sont les partie "GPU".
Le split source en revanche utilisent en générale un langage dédié pour écrire le code "GPU" (le shader), qui doit ensuite être compiler et envoyer au GPU via l'API. C'est le cas de toute les API "graphique" (DirectX, OpenGL, ...).

	Avantages	Inconvénients
Single Source	Plus facile à prendre en main; Plus grande abstraction; Pas de code "glue";	Force l'utilisation d'un langage; L'abstraction cache parfois trop de chose;
Split Source	Pas de contrainte de langage; Permet de configurer l'exécution très finement; Meilleur identification des parties CPU/GPU;	Nécessite beaucoup de code glue; Plus compliquer à prendre en main;



Chaque approche à des avantages et inconvénients. Globalement, les API single source on tendance à être plus facile a prendre en main car elle cache beaucoup de détails rébarbatif, mais en contrepartie de forcer le choix du langage et d'abstraire parfois des choses qui peuvent être important pour le développeur.

En revanche, le split source va demander beaucoup de code, de connaissance de l'API et du fonctionnement d'un GPU au développeurs, mais permet d'être beaucoup plus précis sur l'exécution du shader, l'envoie des donneés au GPU et autres. De plus, on peu utiliser n'importe qu'elle langage, tant que des binding existe (les API sont le plus souvent en C).

Pour cette démo, j'ai choisi Vulkan :

- Cross-plateform et cross-vendor
- Split source
- Bas niveau

Pour cette démo, j'ai choisi Vulkan :

- Cross-plateform et cross-vendor
- Split source
- Bas niveau

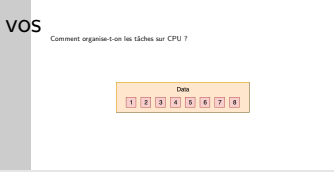
La démo utilisera mon framework open-source Wyzoid.

Pour cette démo, j'ai choisi Vulkan :
- Cross-plateform et cross-vendor
- Split source
- Bas niveau

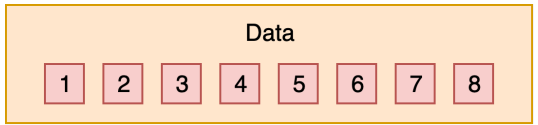
La démo utilisera mon framework open-source Wyzoid.

Exercice 1:
Faire un programme qui double chaque élément d'une liste de nombres flottants 32bits

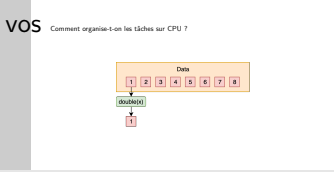
Notre premier programme va être un équivalent du hello world : On va prendre une liste de nombres et les doubler.



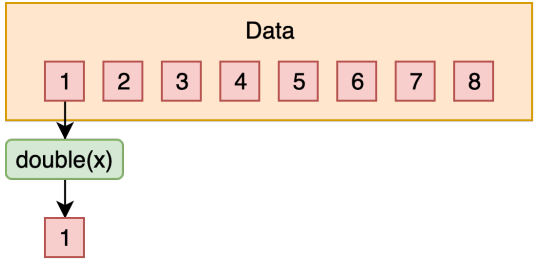
Comment organise-t-on les tâches sur CPU ?



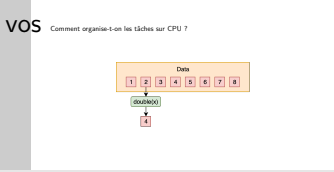
Fondamentalement, un CPU est toujours séquentiel: On exécute une opération à la fois.
On a un accès complet à notre mémoire et on a l'entière contrôle de notre pointeur d'exécution.
On va donc itérer sur nos données une à une et exécuter notre instruction. On peu aller en avant, en arrière, sauter des données, ... on contrôle entièrement notre processus séquentiel.



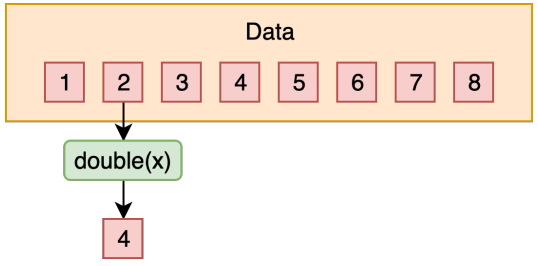
Comment organise-t-on les tâches sur CPU ?



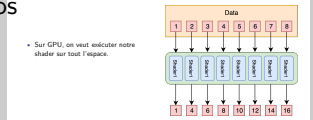
Fondamentalement, un CPU est toujours séquentiel: On exécute une opération à la fois.
On a un accès complet à notre mémoire et on a l'entière contrôle de notre pointeur d'exécution.
On va donc itérer sur nos données une à une et exécuter notre instruction. On peu aller en avant, en arrière, sauter des données, ... on contrôle entièrement notre processus séquentiel.



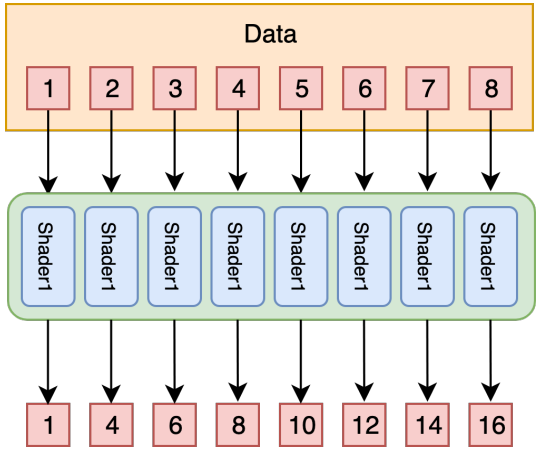
Comment organise-t-on les tâches sur CPU ?



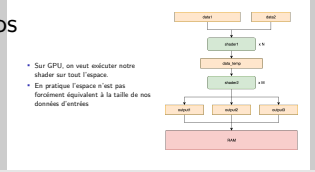
Fondamentalement, un CPU est toujours séquentiel: On exécute une opération à la fois.
On a un accès complet à notre mémoire et on a l'entière contrôle de notre pointeur d'exécution.
On va donc itérer sur nos données une à une et exécuter notre instruction. On peu aller en avant, en arrière, sauter des données, ... on contrôle entièrement notre processus séquentiel.



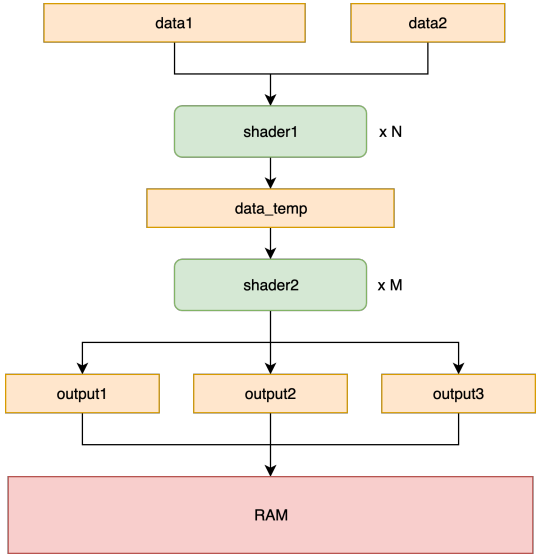
- Sur GPU, on veut exécuter notre shader sur tout l'espace.



En revanche, sur GPU, nous voulons nous exécuter sur l'ensemble de nos données en même temps. Cela fait que l'on peut pas contrôler notre pointeur d'exécution comme on le veut.



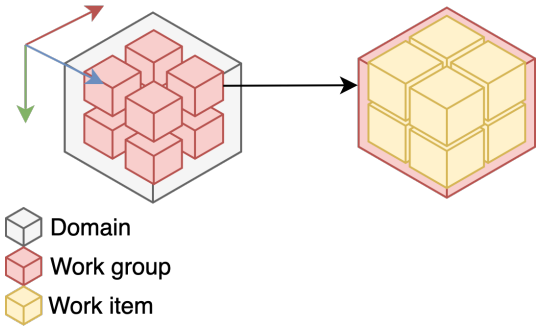
- Sur GPU, on veut exécuter notre shader sur tout l'espace.
- En pratique l'espace n'est pas forcément équivalent à la taille de nos données d'entrées



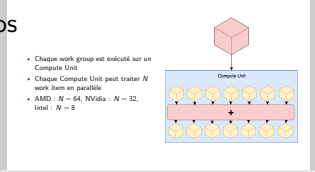
Qui plus est, notre espace de travail ou d'exécution n'est pas forcément équivalent à la taille de nos données d'entrée. On peut avoir N exécution d'un shader qui prend plusieurs données d'entrées et génère des données de sortie qui seront eu même utiliser par un shader qui sera exécuté M fois, ...

Il nous faut donc un modèle qui nous permet de gérer tout ces cas.

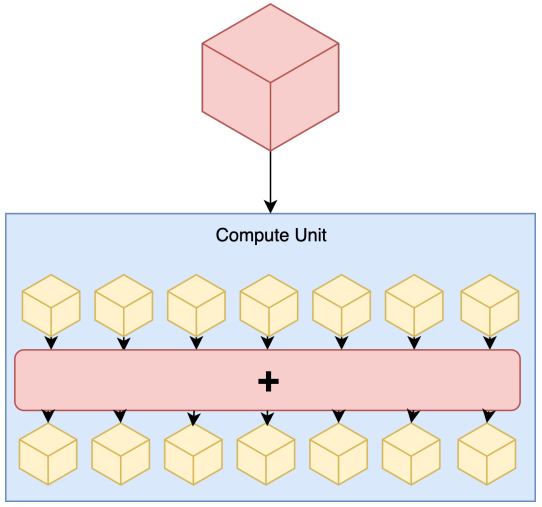
- L'espace de travail est un espace 3D
- Il est divisé en work group (thread group/groupe local) (wX, wY, wZ)
- Chaque groupe contient (lX, lY, lZ) work item (invocation du shader)
- Le nombre total d'invocation est égale à $(lX \times lY \times lZ) \times (wX \times wY \times wZ)$
- Chaque invocation a un identifiant unique (accessible avec gl_GlobalInvocationID en GLSL)



Tout d'abord, on défini un espace de travail en 3D. La raison pour laquelle cet espace est en 3D est en dehors du champs de ce talk, mais vous pouvez venir me voir à notre stand si cela vous intéresse.
Cet espace est divisé en work group (aussi appelé thread group/groupe local).
De plus, chaque work group est lui aussi divisé en work item. Chaque work item est une exécution du shader.
Le nombre totale d'éxécution du shader est donc égale à la multiplication du nombre de work group par le nombre de work item par work group.
Si on revient vers notre exemple, GI_InvocationID nous donne l'identifiant unique de chaque exécution du shader. On a aussi accès à d'autre variable qui permette notamment de récupérer l'id du work group dans lequel est notre work item



- Chaque work group est exécuté sur un Compute Unit
- Chaque Compute Unit peut traiter N work item en parallèle
- AMD : $N = 64$, NVidia : $N = 32$, Intel : $N = 8$



Maintenant, pourquoi la taille du work group est important ?
Un work group est exécuté sur un compute unit et chaque compute unit peut exécuté N donnée en même temps.
Ce nombre de donnée varie en fonction du constructeur, par exemple, on a 64 pour AMD et 32 pour NVidia

Si on veut utiliser tout le compute unit, il faut donc que notre nombre de work item par work group soit égale au nombre d'item que peut traiter notre CU

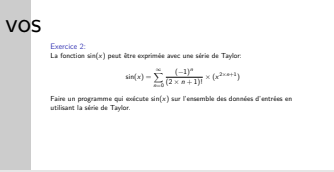
Pour maximiser l'occupancy:
- Nombre de work group $\geq$ Nombre compute unit
- Nombre de work item par work group $= N$
- Rule of thumb : $(IX \times IY \times IZ) = 64$

Pour maximiser l'occupancy:

- Nombre de work group $\geq$ Nombre compute unit
- Nombre de work item par work group $= N$
- Rule of thumb : $(IX \times IY \times IZ) = 64$

Donc, si l'on souhaite maximisé notre "occupency", c'est à dire, le pourcentage d'utilisation du GPU, on doit respecter les régles suivantes: On doit avoir plus ou autant de work group que l'on a de CU. Le nombre de work item par work group doit être égale au nombre d'item traitable par notre CU.
Si on a le luxe de connaître le matériel sur lequel on s'exécute, on peut choisir le bon nombre de work item, sinon, 64 est une valeurs qui va en règle générale produire des performances correcte sur tout les GPU.
On peut ce demander pourquoi on respecte pas systématiquement ces règles. Il y a des raisons mais que je n'aurais pas le temps d'expliquer dans ce talk. Donc passer me voir au stand zenika ;)

Allons plus loin



Pour aller un peu plus loin, on va vouloir cette fois exécuter la fonction sin sur toute nos données d'entrée.

Celle-ci peut être exprimer via une série de taylor défnie par l'équation suivante.

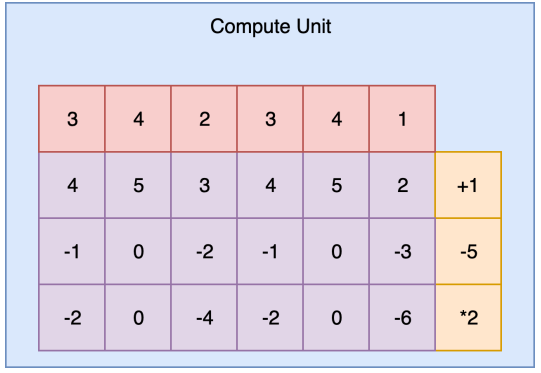
Exercice 2:

La fonction $\sin(x)$ peut être exprimée avec une série de Taylor:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2 \times n + 1)!} \times (x^{2 \times n + 1})$$

Faire un programme qui exécute $\sin(x)$ sur l'ensemble des données d'entrées en utilisant la série de Taylor.

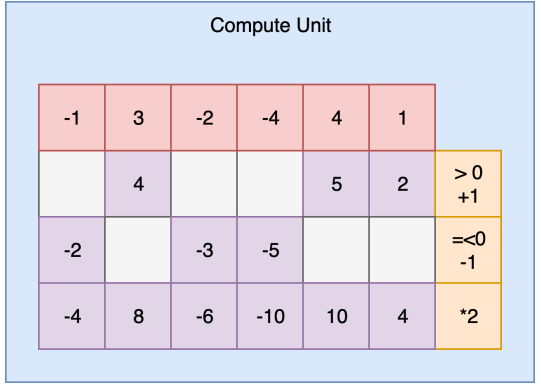
- Un Compute Unit est un processeur SIMD, Single Instruction Multiple Data
- Il exécute une opération sur l'ensemble des données en même temps



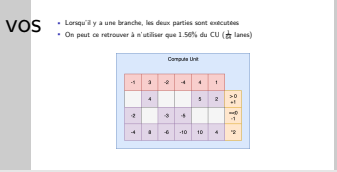
Un CU est un processeur dit SIMD pour Single Instruction Multiple Data. Les instructions SIMD, qui existe aussi sur CPU, sont des instructions qui vont s'exécuter sur plusieurs données à la fois.

Sur cette exemple, on va avoir nos données d'entrée qui sont stocker dans les registres de notre CU, et on va exécuter plusieurs instruction dessus.

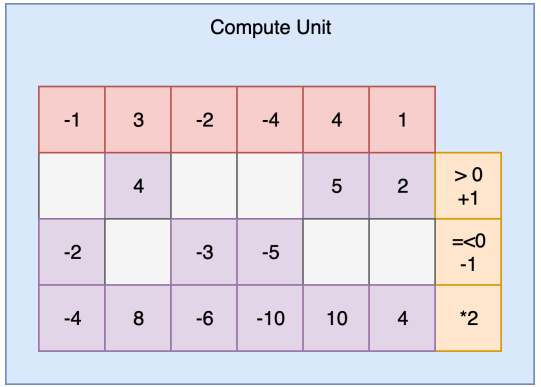
- Lorsqu'il y a une branche, les deux parties sont exécutées



Les problèmes commences lorsque l'on a du branching (c'est à dire des if).
Lorsqu'il y a des branches, on est obligé d'exécuter les deux parties de la branche. Dans le pire des cas, on peut ce retrouver à n'utiliser que un très faible pourcentage de notre CU si on a une branche avec une seul donnée. Si la branche est courte, c'est a dire que l'on a juste quelques instruction par branches, le coût n'est pas prohibitif, par contre, si nos branches son longues, on sous utilise notre CU la majeur partie du temps.



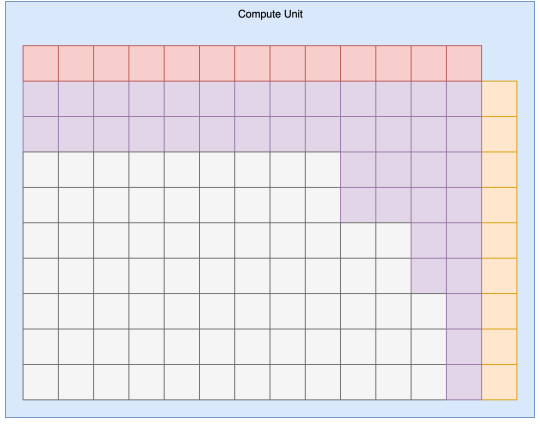
- Lorsqu'il y a une branche, les deux parties sont exécutées
- On peut ce retrouver à n'utiliser que 1.56% du CU ($\frac{1}{64}$ lanes)

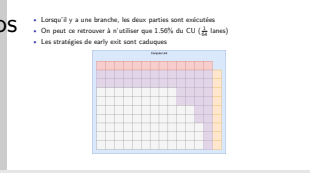


Les problèmes commences lorsque l'on a du branching (c'est à dire des if).

Lorsqu'il y a des branches, on est obligé d'exécuter les deux parties de la branche. Dans le pire des cas, on peut ce retrouver à n'utiliser que un très faible pourcentage de notre CU si on a une branche avec une seul donnée. Si la branche est courte, c'est a dire que l'on a juste quelques instruction par branches, le coût n'est pas prohibitif, par contre, si nos branches son longues, on sous utilise notre CU la majeur partie du temps.

- Lorsqu'il y a une branche, les deux parties sont exécutées
- On peut ce retrouver à n'utiliser que 1.56% du CU ($\frac{1}{64}$ lanes)
- Les stratégies de early exit sont caduques

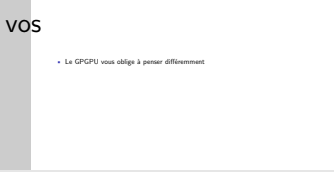




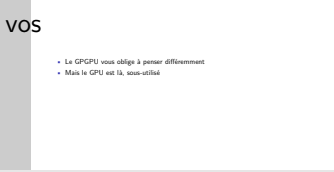
Cela à d'autre conséquence, par exemple, cela signifie que les stratégies de early exit sont complètement caduc car si un seul élément tombe dans le cas le plus coûteux, on va forcément aller jusqu'au bout.

Si on ne peut garantir que l'on va systématiquement tombé dans l'une au l'autre côté d'une branche, il vaut donc mieux directement utilisé la version la plus précise / coûteuse.

Conclusion



- Le GPGPU vous oblige à penser différemment



- Le GPGPU vous oblige à penser différemment
- Mais le GPU est là, sous-utilisé

- Le GPGPU vous oblige à penser différemment
- Mais le GPU est là, sous-utilisé
- Vous pouvez très vite être plus performant grâce au GPGPU !

Slides et demos :

Slides et demos :