

Programmation GPGPU

Jonathan Caux

jcaux@adobe.com

Contexte du cours

Qui suis-je

- Jonathan Caux
- ZZ2 promo 2008
- Thèse au LIMOS soutenu en 2012
 - Cours C++
 - Cours GPGPU
- Depuis principalement dév' C++
 - Murex, progiciel de finance (dév' C++)
 - AVM Up, téléphonie (dév' C++)
 - Thales service, ESN du groupe Thales (tech lead C++)

Adobe 3D&I

- Senior dev' à Clermont depuis 2021 chez Adobe
- Division 3D&I (3D and Immersive) issue du rachat d'Allegorithmic
 - Allegorithmic, entreprise de texturing 3D créée à Clermont et d'abord hébergée à l'ISIMA
- Gamme de produit autour du texturing 3D
 - Designer, Painter (Clermont)
 - Sampler (Lyon)
 - Pleins d'autres logiciels et outils, ici et ailleurs

Adobe Substance 3D Designer

- Équipe (au sens large) de 11 personnes :
 - 1 tech lead + 4 dev' (dont 2 seniors) + 1 recrutement en cours
 - 2 PM (plus ou moins PO/scrum master)
 - 2 ingé' qualité
 - 1 UX/UI designer (partagé)
- Logiciel de création de texture 3D
 - C++ / Qt / Google Test
 - CMake / git / Jenkins

Contenu du cours

- Cours réalisé en 2010 pour la première fois
 - Domaine novateur à l'époque « Bémol de la nouvelle technologie : quelle pérennité ? » (cours GPGPU, 2010)
 - Domaine bien maîtrisé maintenant
- Explication du pourquoi du GPGPU + Programmation en CUDA
- Objectif double :
 - Savoir implémenter un algo sur GPU
 - Savoir quand et comment implémenter un algo sur GPU (force et faiblesse des GPU)

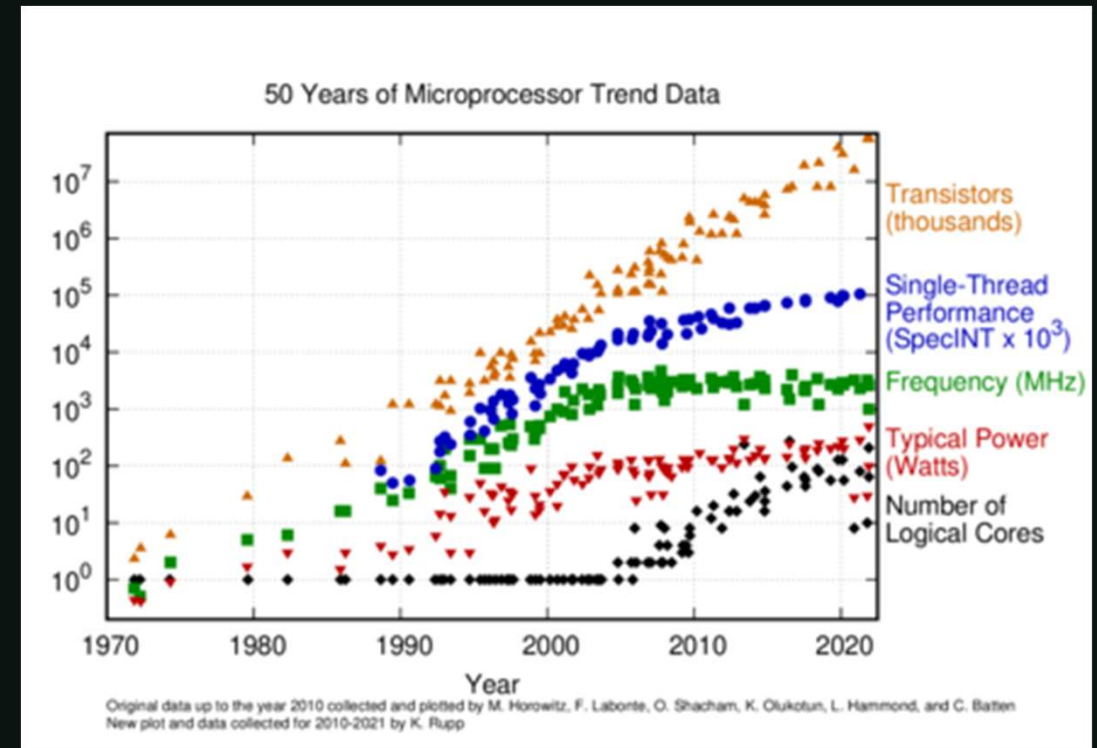
Contenu du cours

- N'hésitez pas à remonter vos remarques
- Loin de tout savoir, contenu très largement inspiré de la documentation CUDA
- Évaluation : TP + examen écrit + projet

Historique du contexte GPGPU

Evolution des microprocesseurs - courbes

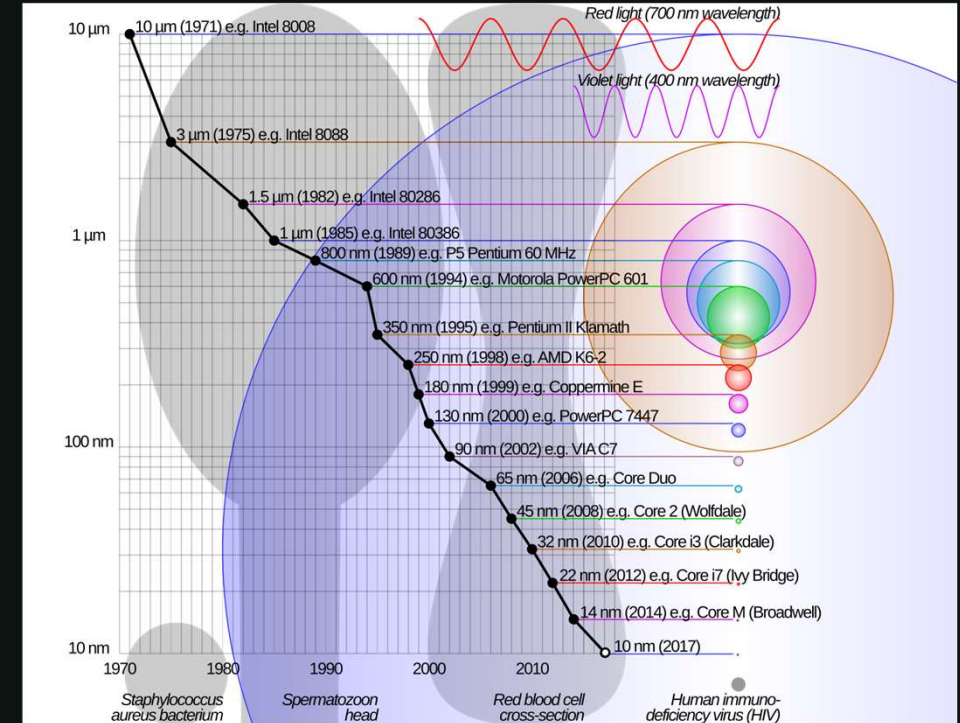
- Augmentation exponentielle jusqu'en 2005
- A partir de 2005 :
 - Augmentation + lente des perf' single thread
 - Multiplication des coeurs



https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/La_loi_de_Moore_et_les_tendances_technologiques

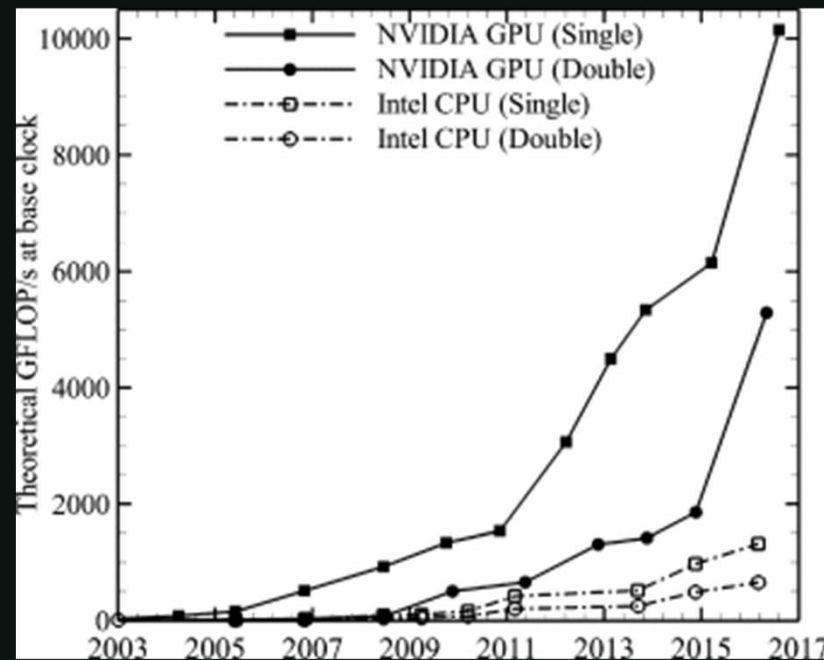
Evolution des microprocesseurs - gravure

- Augmentation exponentielle jusqu'en 2005
- Gravure de plus en plus fine
 - Toujours plus de transistor au m²
 - (5nm sans doute la limite)
- Problématique d'alimentation de ces transistors toujours plus fins.



https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/La_loi_de_Moore_et_les_tendances_technologiques

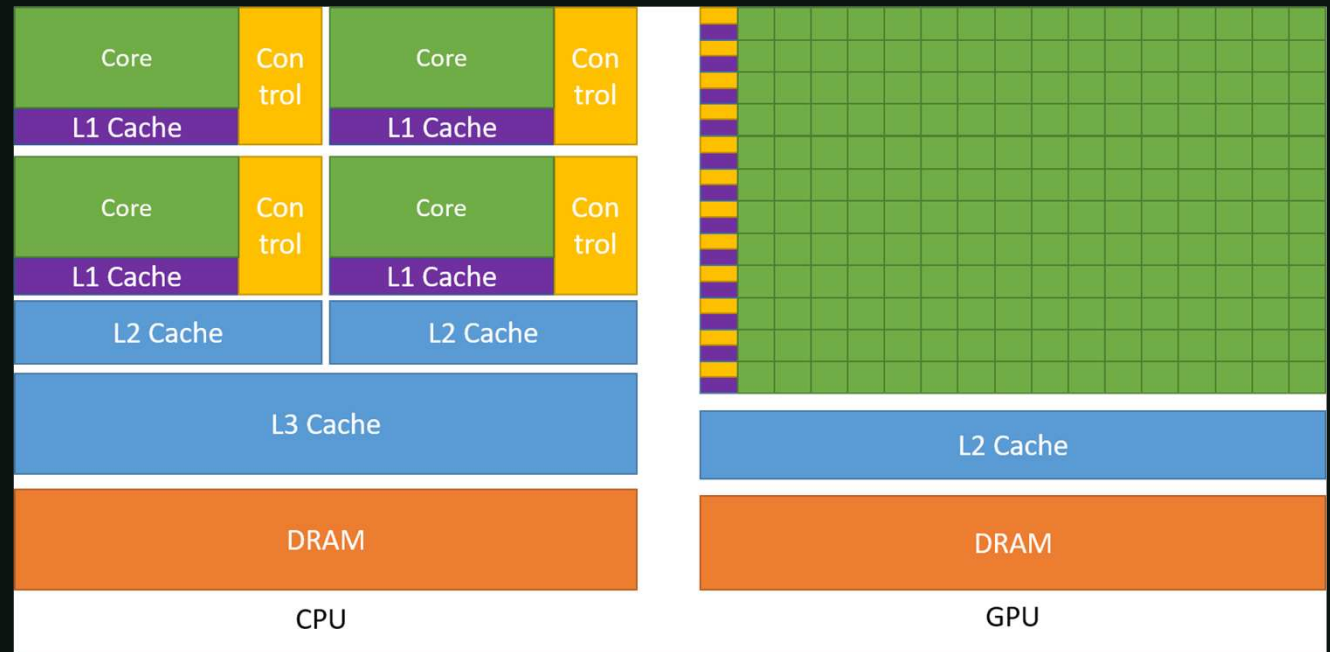
Puissance de calcul CPU vs GPU



An investigation of hybrid CPU-GPU solvers for supersonic reacting flow simulation with detailed chemical kinetics

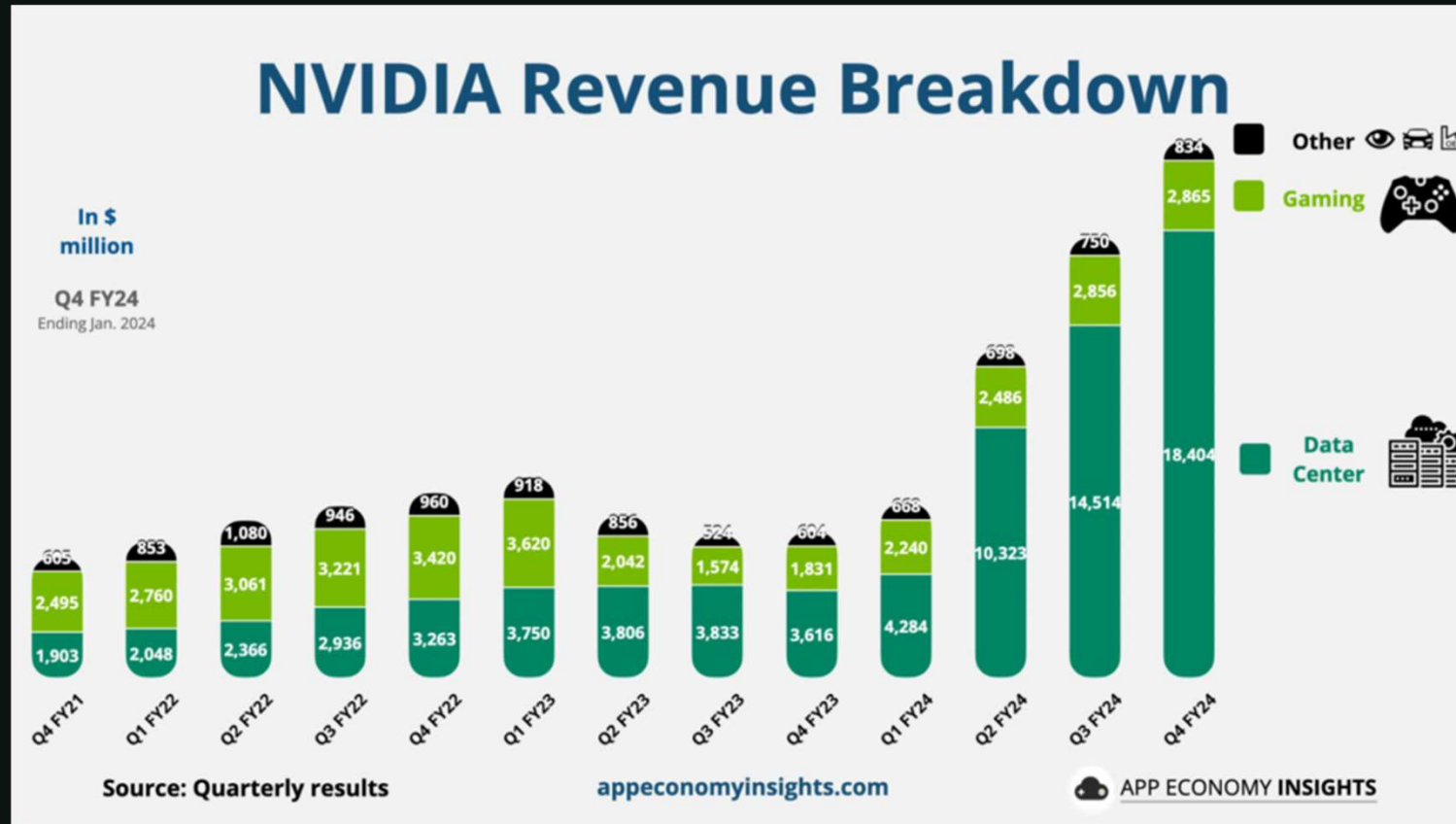
Organisation CPU vs GPU - aperçu

- CPU optimisé pour le contexte switching
- GPU optimisé pour le traitement d'un même jeu d'instruction sur des données différentes



<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Revenu NVIDIA



<https://blogs.alphanso.ai/blogs/the-remarkable-rise-of-nvidia/>

Cours action NVIDIA



<https://lemediadelinvestisseur.fr/bourse/acheter-action-nvidia>

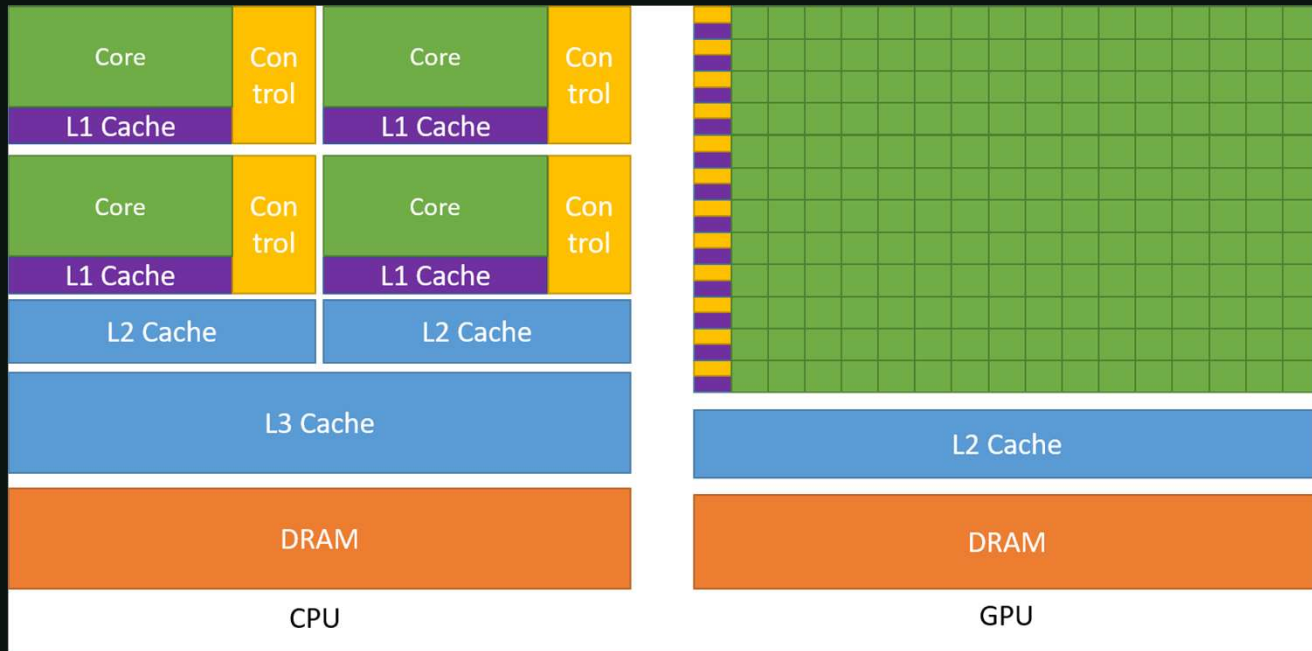
Introduction aux GPGPU

Historique des GPU

- Manque de puissance sur les PC & partie rendu graphique de plus en plus gourmande
- Objectifs :
 - Déléguer au GPU des tâches importantes (en termes de temps de calcul) mais répétitives
 - Effectuer ces tâches bien plus rapidement qu'avec un CPU + libérer le CPU
- Méthodes :
 - Optimiser pour réaliser des calculs en virgule flottante
 - Optimiser pour réaliser la même opération sur des données différentes

Principales différences architecturales des CPU et des GPU

- CPU :
 - + de contrôles
 - + de caches
- GPU :



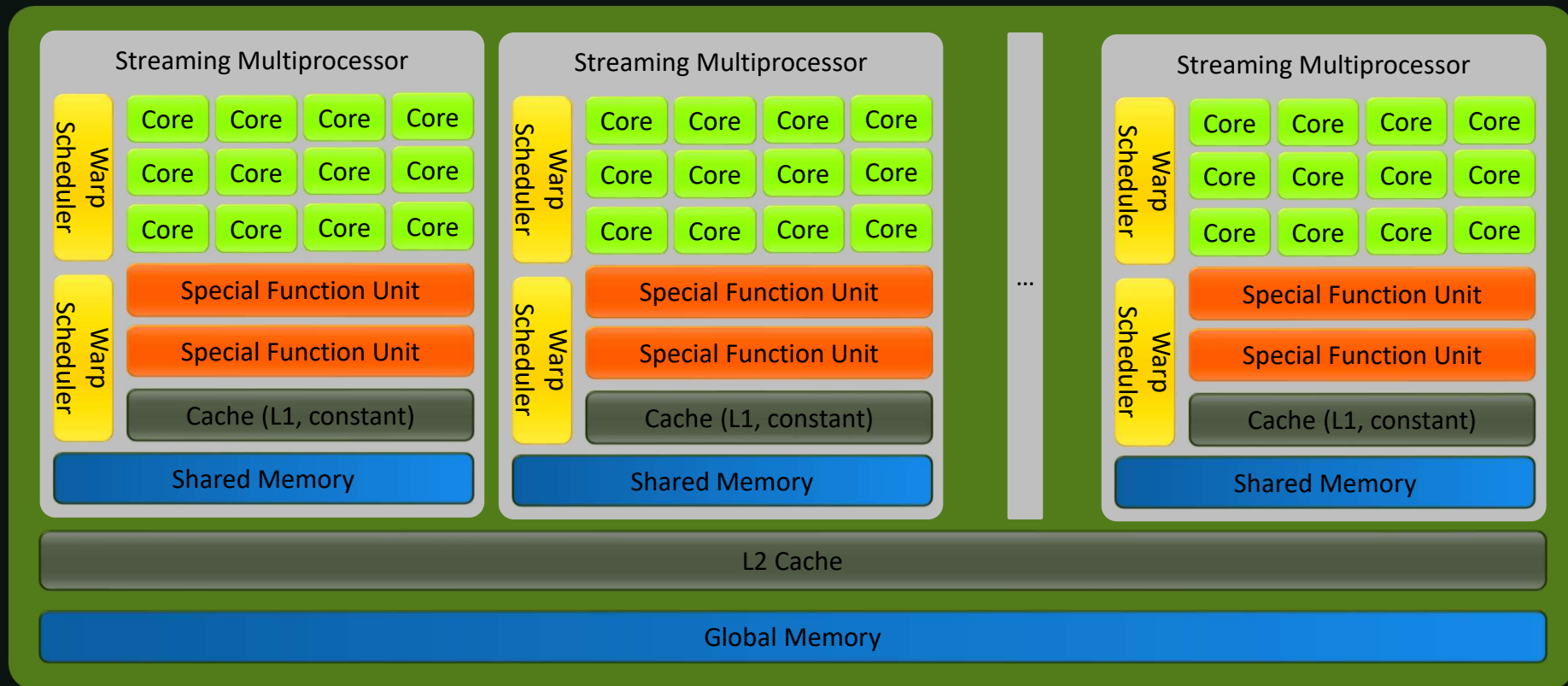
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Impacts des différences architecturales sur la prog' GPGPU

- Plus d'unités de calcul => Capacité de calcul brut plus élevés
- Moins de contrôles => Moins de facilité à effectuer des tâches diverses
- Moins de caches => Accès mémoire plus lents

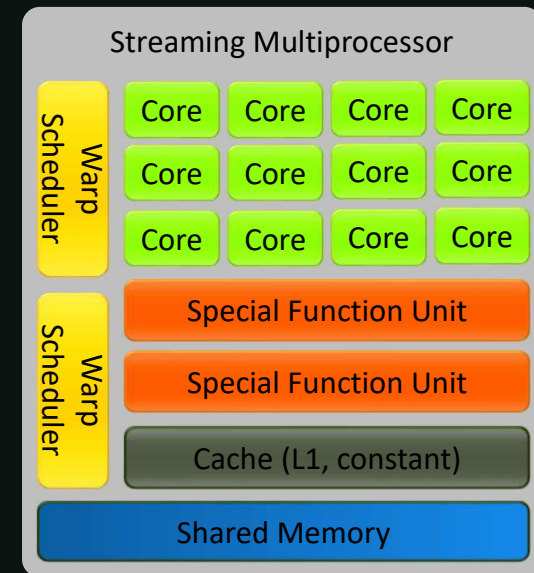
Architecture simplifiée d'un GPGPU 1/2

- /!\ Dépend des « Compute Capability » du GPU ↔ de sa génération



Architecture simplifiée d'un GPGPU 2/2

- Core** ▪ Cœur de calcul arithmétique
- Special Function Unit** ▪ Cœur de calcul des fonctions transcendante (cos, sin..., exp...)
- Warp Scheduler** ▪ Unité de contrôle
- Cache (L1, constant)** ▪ Cache le plus performant
- Shared Memory** ▪ Mémoire R/W la plus performante
- L2 Cache** ▪ Cache global au GPU
- Global Memory** ▪ Mémoire R/W la plus large



Exemple de génération de carte NVIDIA

- 5.X :
 - 128 Core
 - 32 SFU
 - 4 Warp Scheduler
 - 24KB Cache L1/constant
 - 64/96KB Shared Memory
 - L2 Cache et Global Memory dépendent du modèle
 - Exemple Tesla M40 : 3MB de cache L2 et 12Go de mémoire globale

Résumé

- Beaucoup de puissance de calcul
- Pas beaucoup de cache
 - => Accès mémoire potentiellement complexe
- Pas beaucoup de contrôle
 - => Limité le changement de contexte

Approche SIMD

Architecture de calcul parallèle

- De nombreuses architectures différentes existent :
 - SMP (Symmetric MultiProcessing) : au moins deux processeurs identiques connectés à une même mémoire
 - Cluster (ferme de calcul) : groupe de composants de calcul localement interconnectés entre eux
 - Grille de calcul : groupe de composants de calcul interconnectés entre eux pouvant être très fortement éloignés géographiquement
 - FPGA (Field-Programmable Gate Array) : circuit intégré programmable permettant la réalisation de calcul parallèle
 - GP-GPU

Classification des architectures d'ordinateurs

- La taxonomie de Flynn propose une classification en quatre catégories :

- SISD (Single Instruction, Single Data stream)

Ex : Processeur mono-coeur traditionnel.

- SIMD (Single Instruction, Multiple Data streams)

Ex : GPU.

- MISD (Multiple Instruction, Single Data stream)

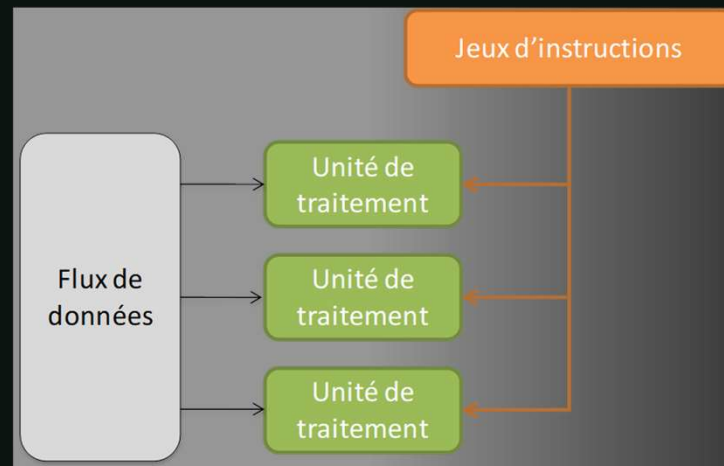
Ex : Utilisé pour la détection d'erreur dans des applications très sensible.

- MIMD (Multiple Instruction, Multiple Data streams)

Ex : SMP, Cluster ou Grille de calcul.

Single Instruction, Multiple Data streams

- Exécution de la même instruction en parallèle sur différents flux de données
- Également utilisé dans les instructions SSE (Streaming SIMD Extensions) des processeurs actuels



Exemple de pseudo-code SIMD

```
int    vecteur_A[ 4 ] = { 2, 4, 6, 8 };
int    vecteur_B[ 4 ] = { 2, 3, 5, 7 };
int    vecteur_C[ 4 ];
```

// Sur une architecture non SIMD :

```
vecteur_C[ 0 ] = vecteur_A[ 0 ] + vecteur_B[ 0 ]; // 1er cycle
vecteur_C[ 1 ] = vecteur_A[ 1 ] + vecteur_B[ 1 ]; // 2ème cycle
vecteur_C[ 2 ] = vecteur_A[ 2 ] + vecteur_B[ 2 ]; // 3ème cycle
vecteur_C[ 3 ] = vecteur_A[ 3 ] + vecteur_B[ 3 ]; // 4ème cycle
```

// Sur une architecture SIMD avec quatre unités de traitement,
// chaque unité de traitement effectue en parallèle :

```
vecteur_C[ numero_unite ] = vecteur_A[ numero_unite ] +
                             vecteur_B[ numero_unite ]; // 1er et unique cycle
```

SIMD pour les GP-GPU : SIMT

- NVIDIA parle de Single Instruction, Multiple Threads (SIMT)
- Exécution de plusieurs threads en parallèle exécutant le même code mais traitant des données différentes
- Chaque thread possède un identifiant unique lui permettant d'accéder à une donnée propre

Gestion des threads – les kernels

Introduction sur les kernels

- Kernel : fonction invoquée par l'hôte (le CPU) et exécutée plusieurs fois en parallèle sur le GPU
 - ⇔ Fonctions sur GPU
- Paramétré lors de l'appel par le nombre de threads utilisés (en plus des paramètres habituels)
- Accède à la mémoire du GPU uniquement
 - => Impossibilité d'accéder à la mémoire du CPU

Appel de kernel 1/2

```
__global__ void monPremierKernel()
{
}
```

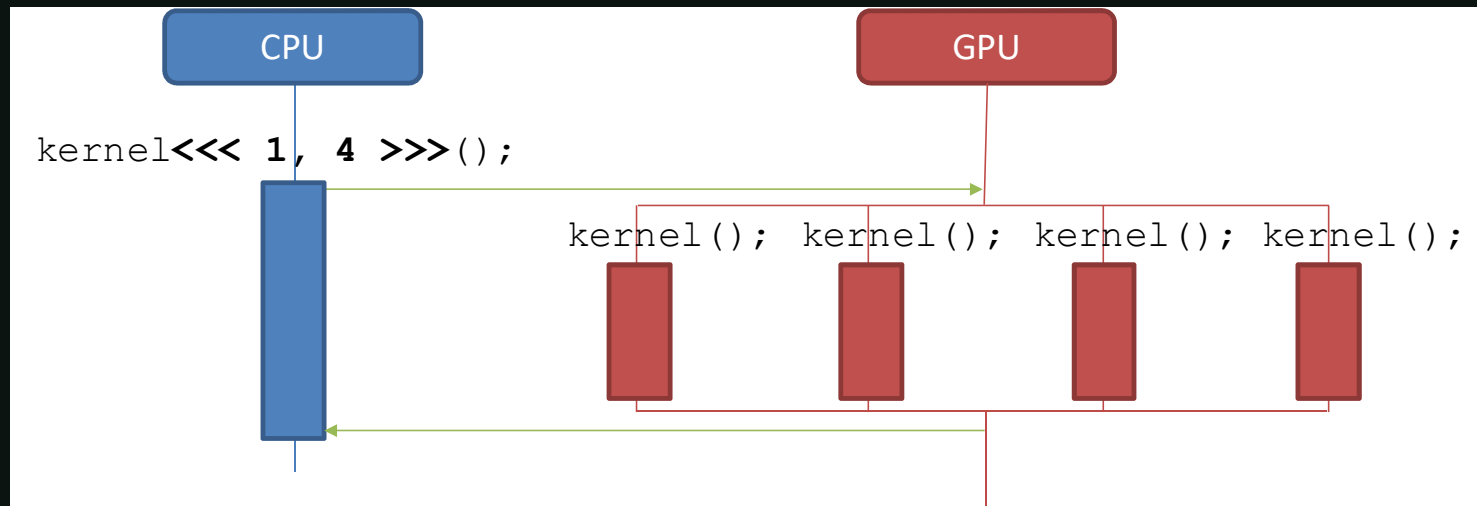
```
int main( int, char *[] )
{
    monPremierKernel<<< 1, 10 >>>();
}
```

- Appel standard (simplifié) d'un kernel :

```
monKernel<<< nbBloc, nbThdParBloc >>>(
    /*paramètres*/
);
```

Appel de kernel 2/2

```
__global__ void kernel()  
{  
int main( int, char *[] )  
{  
    kernel<<< 1, 4 >>>();  
}
```



Déclaration d'un kernel

- Mot clé CUDA : `__global__`.
 - Spécifie que la fonction est un kernel.
 - Appelé depuis l'hôte (le CPU).
 - Exécuté sur le GPU.

- Mot clé CUDA : `__device__`.
 - Appelé et exécuté sur le GPU.

- Exemple :

```
__global__ void multiMatrice( float * inMatrice_A,
                             float * inMatrice_B,
                             float * outMatrice_C );
```

Notion de warp 1/2

- Notion logicielle utilisée par NVIDIA et qui correspond au regroupement de 32 threads « identiques »
- Exécution obligatoire d'un nombre rond de warp
 - Dans le cas contraire, arrondi au nombre de warp supérieur
- Exécution sur le GPU par warp (par groupe de 32 threads)

Notion de warp 2/2

- Tous les threads d'un warp jouent le même jeu d'instruction (SIMT)
 - => Si flux d'instructions différents, cycle différents


```
// Mult x2 + 1 si data pair ou + 2 si impair
res = data[threadId] * 2;
if (data[threadId]%2 == 0) {
    res += 1; // certains threads
} else {
    res += 2; // les autres threads
}
```

- Impact important sur le comportement du GPU
 - => Impact tout aussi important sur l'implémentation
 - Nouvelles archi' GPU de plus en plus permissives

Gestion des threads – les blocs

Introduction sur les blocs

- Regroupement sous forme de blocs des threads exécutés sur le GPU
- « Exécution simultanée » des threads d'un même bloc
 - Possibilités de communication entre ces threads
 - Possibilités de partage de mémoire entre ces threads
- Limitation matérielle du nombre de threads par bloc
 - Évolue maintenant peu avec les versions de GPU (1024 depuis près de 10 ans)
 - Du à la mémoire nécessaire pour faire du scheduling de threads

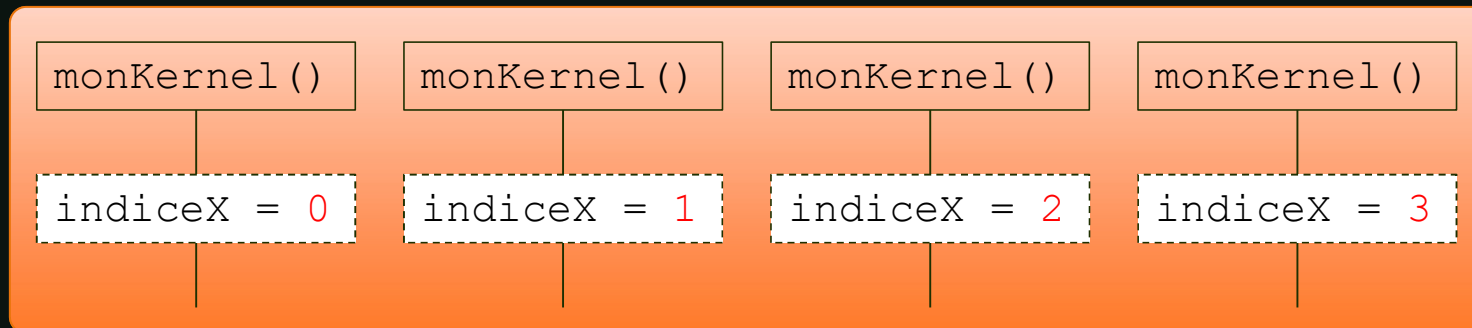
- `kernel<<< 1, 10 >>>();`

 Dimension des blocs utilisés

Différenciation des threads d'un bloc : threadIdx

- Sans différenciation des threads, travail similaire effectué par tous les threads
 - threadIdx : variable CUDA identifiant de manière unique un thread dans un bloc

```
__global__ void monKernel() { int indiceX = threadIdx.x; }
```

```
monKernel<<< 1, 4 >>>();
```

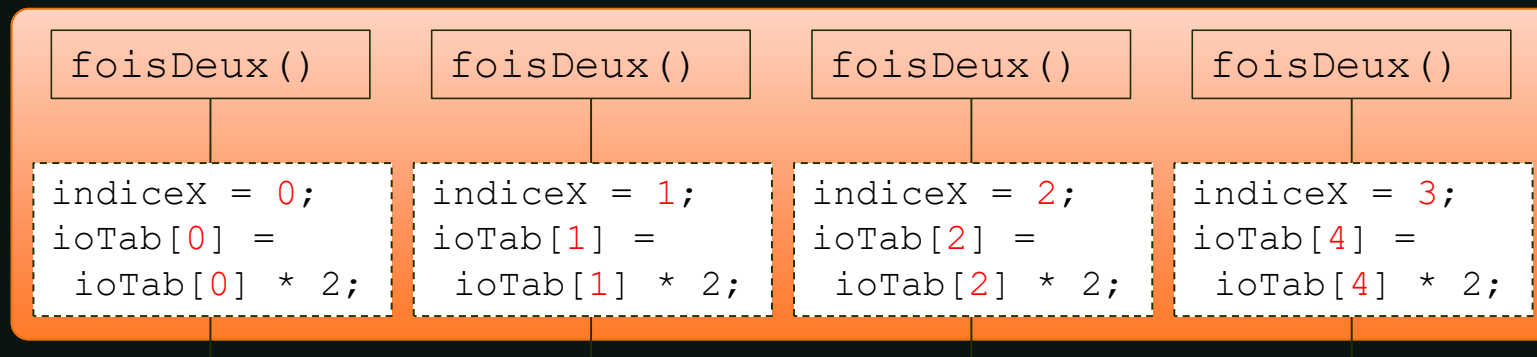


Première application « utile »

```
__global__ void foisDeux( float * inoutTab )
{
    int indiceX = threadIdx.x;
    inoutTab[ indiceX ] = inoutTab[ indiceX ] * 2;
}
```

```
float tableau[] = { 2, 4, 6, 8 };
foisDeux<<< 1, 4 >>>( tableau );
```

/*! Code incorrect
utilisation de la mémoire CPU sur le GPU



Premier exercice : addition de vecteur

```
__global__ void ajout_vecteur( )
{
    // Calcul de l'indice
    int idX = ;
    // Réalisation de l'ajout
    ;
}

int main( int, char *[] )
{
    float tableau_A[ 5 ] = { 2, 4, 6, 8, 10 };
    float tableau_B[ 5 ] = { 1, 2, 3, 4, 5 };
    float tableau_C[ 5 ];

    // Appel du kernel
    ;
}
```

**/*! Code incorrect
utilisation de la mémoire CPU sur le GPU**

Premier exercice : correction

```
__global__ void ajout_vecteur( float * inTabA, float * inTabB, float * outTabC )
{
    int idX = threadIdx.x;
    outTabC[ idX ] = inTabA[ idX ] + inTabB[ idX ];
}
```

```
int main( int, char *[] )
{
    float tableau_A[ 5 ] = { 2, 4, 6, 8, 10 };
    float tableau_B[ 5 ] = { 1, 2, 3, 4, 5 };
    float tableau_C[ 5 ];

    ajout_vecteur<<< 1, 5 >>>( tableau_A, tableau_B, tableau_C );
}
```

/*! Code incorrect
utilisation de la mémoire CPU sur le GPU

Bloc à plusieurs dimensions 1/3

- Prototype (presque exact) d'un appel de kernel :

```
nom_du_kernel<<< dim3, dim3 >>>( [parametres] );
```

- dim3, type basé sur uint3 :

```
struct uint3 { unsigned int x, y, z; };
```

- Pour chaque dimension :

- Valeur par défaut : 1
- Valeur maximal : dépend des versions (1024 pour les dernières générations)

- Ex :

```
kernel<<< 1, 10 >>>() ⇔ kernel<<< 1, dim3( 10, 1, 1 ) >>>();
```

Bloc à plusieurs dimensions 2/3

- `blockDim` : fournit, à l'exécution, les dimensions du bloc dans lequel le kernel est exécuté
- Fonctionnement similaire à `threadIdx` :
 - Également de type `dim3`
 - Accès à la taille de chaque dimension à l'aide des composantes `x`, `y` et `z`

Bloc à plusieurs dimensions 3/3

- Multiplication d'un tableau par 2 + ajout de chaque valeur d'un second tableau
- N valeurs au départ => nb de valeurs du second tableau * N valeurs à l'arrivée

Tableau A :

1	2	3	4	5
---	---	---	---	---

Tableau B :

0	3
---	---

Résultat :

2	4	6	8	10	5	7	9	11	13
---	---	---	---	----	---	---	---	----	----

Bloc à plusieurs dimensions 3/3

```
__global__ void tabAfoisDeuxPlusTabB( float * inTabA, float * inTabB, float * outTabC ) {
    int idX      = threadIdx.x;
    int idY      = threadIdx.y;
    int idGlobal = idX + idY * blockDim.x;

    outTabC[ idGlobal ] = inTabA[ idX ] * 2 + inTabB[ idY ];
}

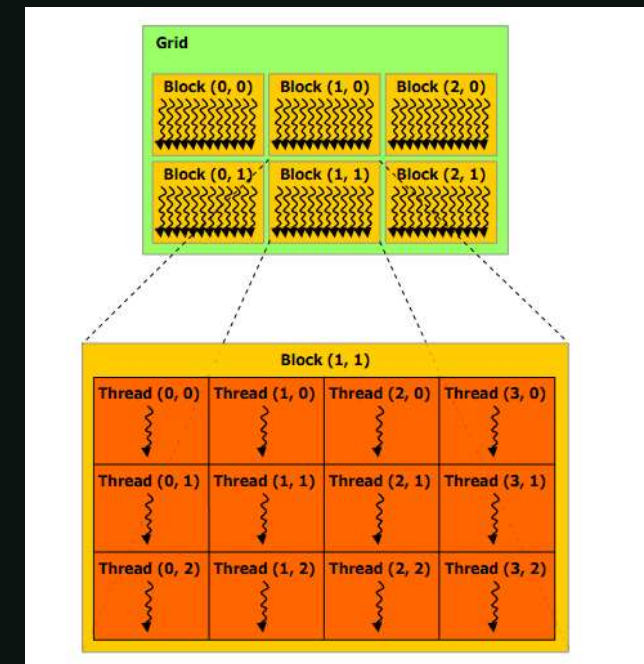
int main( int, char *[] ) {
    float tabA[ 5 ] = { 1, 2, 3, 4, 5 };
    float tabB[ 2 ] = { 0, 3 };
    float tabC[ 10 ];

    dim3 dimBloc( 5, 2 );
    tabAfoisDeuxPlusTabB<<< 1, dimBloc >>>( tabA, tabB, tabC );
}
```

Gestion des threads – la grille

Introduction

- Nombre total de threads par bloc limité (1024 sur les GPU actuels)
 - Très limité en comparaison de la capacité de calcul en parallèle
 - => Utilisation d'une grille contenant plusieurs blocs de dimension identique
- Dimension maximale pour un GPU récent :
 - De la grille : en 'x' : $2^{31}-1$; en 'y' et 'z' : 65535
 - D'un bloc : en 'x' et 'y' : 1024; en 'z' : 64
 - Nombre maximum de threads par bloc : 1024
 - => Jusqu'à plus de 9×10^{21} threads.



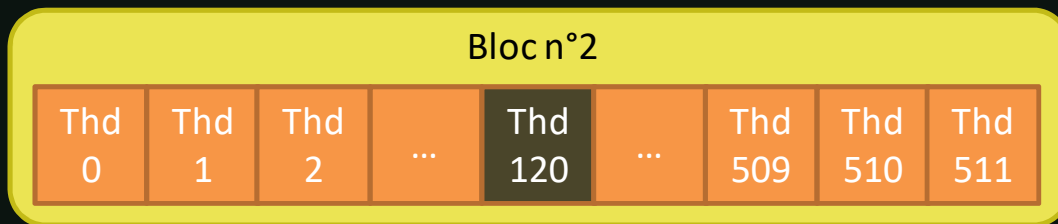
Source NVIDIA

blockIdx et gridDim

- blockIdx : à l'exécution, dans un kernel, position du bloc auquel le thread appartient
 - Fonctionnement similaire à threadIdx
- gridDim : à l'exécution, dans un kernel, dimension de la grille
 - Fonctionnement similaire à blockDim.




Multiplication de grands tableaux

```
__global__ void doubleGrandTableau( float * inoutTab )
{
    int idX    = threadIdx.x + blockIdx.x * blockDim.x;
    inoutTab[ idX ] = inoutTab[ idX ] * 2;
}
```



```
threadIdx.x = 120
blockIdx.x  = 2
blockDim.x  = 512
idx = 120 + 2 * 512 = 1144
```

Deuxième exercice : utilisation de grilles à deux dimensions

```
__global__ void doubleImmenseTab( float * inoutTab )
{
    int idX      =  ;
    int idY      =  ;
    int idGlobal =  ;
    inoutTab[ idGlobal ] = inoutTab[ idGlobal ] * 2;
}

int main( int, char *[] )
{
    dim3 dimGrille( 1000, 1000 );
    dim3 dimBloc( 512, 2 );
    float tab[ 1024000000 ];
    // Initialisation du tableau
    [...]
    doubleImmenseTab<<< dimGrille, dimBloc >>>( tab );
}
```

/*! Code incorrect
utilisation de la mémoire CPU sur le GPU

Deuxième exercice : correction

```
__global__ void doubleImmenseTab( float * inoutTab )
{
    int idX      = threadIdx.x + blockIdx.x * blockDim.x;
    int idY      = threadIdx.y + blockIdx.y * blockDim.y;
    int idGlobal = idX + blockDim.x * gridDim.x * idY;
    inoutTab[ idGlobal ] = inoutTab[ idGlobal ] * 2;
}
int main( int, char *[] )
{
    dim3  dimGrille( 1000, 1000 );
    dim3  dimBloc( 512, 2 );
    float tab[ 1024000000 ];
    // Initialisation du tableau
    [...]
    doubleImmenseTab<<< dimGrille, dimBloc >>>( tab );
}
```

/*! Code incorrect
utilisation de la mémoire CPU sur le GPU

Blocs et grilles à multiple dimension

- Nécessaire dans les premières versions de CUDA pour manipuler de grands indices
- Nécessaire maintenant pour manipuler de très grands indices (maximum 10^7 en 'x', 10^{15} maintenant)
- Toujours pratique pour manipuler des tableaux à deux dimensions ou faire deux traitements transversaux

TD1 – calcul $nx^2 + my$

- TD car on ne gère pas la mémoire pour le moment
- Proposer le code pour calculer sur GPU :
 - L'ensemble des n valeurs $nx^2 + my$
 - Où 'n' et 'm' sont saisis par l'utilisateur
 - Pour un très grand nombre de valeurs (taille des tableaux 'tabX' et 'tabY' > 1000000)

TD2 – conversion RGB vers niveau de gris

- TD car on ne gère pas la mémoire pour le moment
- Proposer le code pour calculer sur GPU :
 - La conversion de l'ensemble des pixels de l'image de RGB vers niveau de gris ($0,299 \times R + 0,587 \times G + 0,114 \times B$)
 - Image d'entrée et de sortie, tableau de char (1 octet pour coder de 0 à 255 R, G ou B) à trois dimensions
 - 1^{ère} pour la dimension x
 - 2^{ème} pour la dimension y
 - 3^{ème} pour les valeurs R, G et B

Mémo

- `__global__` + fonction => kernel
- `dim3` : champs x, y et z disponible
- `threadIdx` : indice du thread dans le bloc
- `blockIdx` : indice du bloc dans la grille
- `blockDim` : dimension des blocs
- `gridDim` : dimension de la grille
- TD1 : $nx^2 + my$ avec n et m saisie par l'utilisateur
- TD2 : RGB vers niveau de gris ($0,299xR + 0,587xG + 0,114xB$)

Fin 1^{er} cours