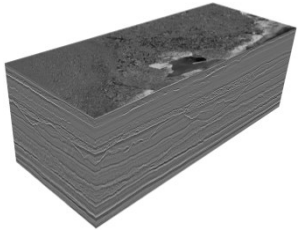
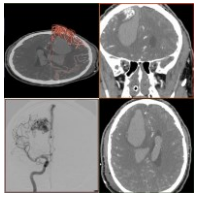
Calcul accéléré par GPU pour le TSI

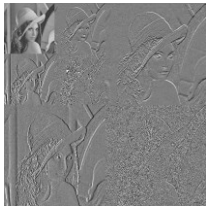
Marc Donias

Contexte (1/2)

- Gigantisme des données
  - Tomographie (sismique, IRM, …)
  - Flux vidéo

- Algorithmes itératifs
  - Filtrage bilatéral / Diffusion par EDP
  - Simulations de Monte-Carlo
  - Transformées (TFD, ondelettes, …)

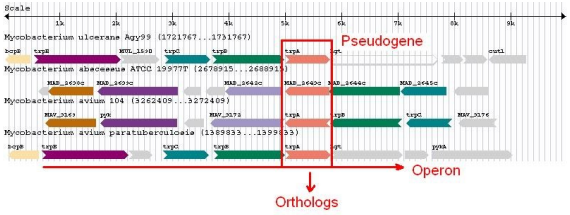


Nécessité de calculs TSI plus rapides

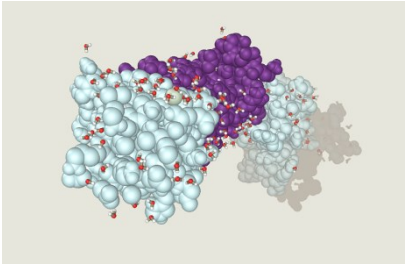
- Réponse temps-réel
  - Compression/Décompression Vidéo
  - Retouche Photo (rehaussement de contours, etc.)

Contexte (2/2)

- Recherche
- Défense
- Finance
- Production
- Multimédia
- Pétrole

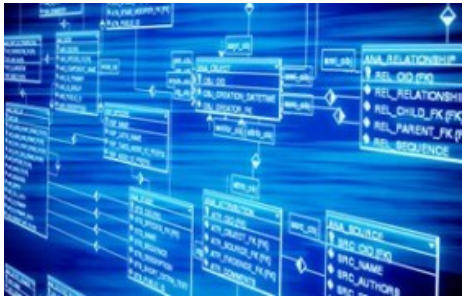


SeqNFind
Comparaison de génome (x400)



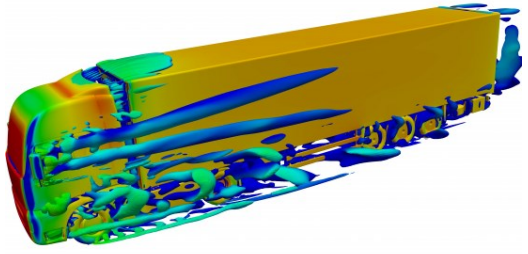
Abalone
Modélisation
biomoléculaire
de protéines (x29)

DigitalGlobe Advanced Ortho Series
Orthorectifcation (x50)

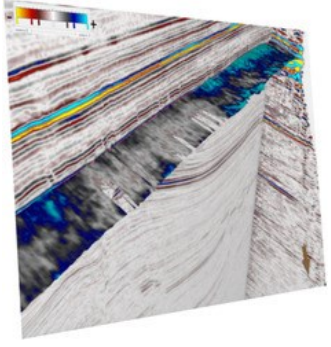


Tanay ZX Lib
Analyse financière (x500)

Adobe Premiere Pro CS6
Edition vidéo (x8)



FluiDyna LBultra
Solveur non-linéaire (x20)



Headwave Suite
Calcul d'attributs (×50)

Source : http://www.nvidia.com/object/gpu-applications.html

Calcul accéléré

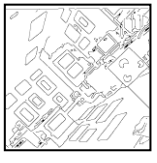
- Temps d'exécution réduit en calculant
  - Différemment
    - Changement de stratégie (exemple : $N^2 \rightarrow NlogN$)
  - Plus vite
    - Augmentation de la fréquence
    - Changement de technologie (à venir : calcul quantique)
  - A plusieurs
    - FPGA, DSP, …
    - Multi-cœurs/Multi-processeurs
    - **Cartes graphiques (GPU)**
    - Supercalculateurs
    - Calcul distribué (SETI@home, World Community Grid)

parallélisme supposé

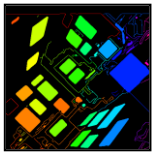
Calcul parallèle ? un exemple

Segmentation de régions contrastées et compactes

Détection de contours



Etiquettage et Remplissage



Sélection de formes

Conv. 2D | Conv. 2D — opérateur à voisinage local

barrière de synchronisation

Norme / Seuillage à hystérésis — opérateur point par point

barrière de synchronisation

Etiquettage et Remplissage — méthodes récursives

Descripteurs de régions Sélection — par forme

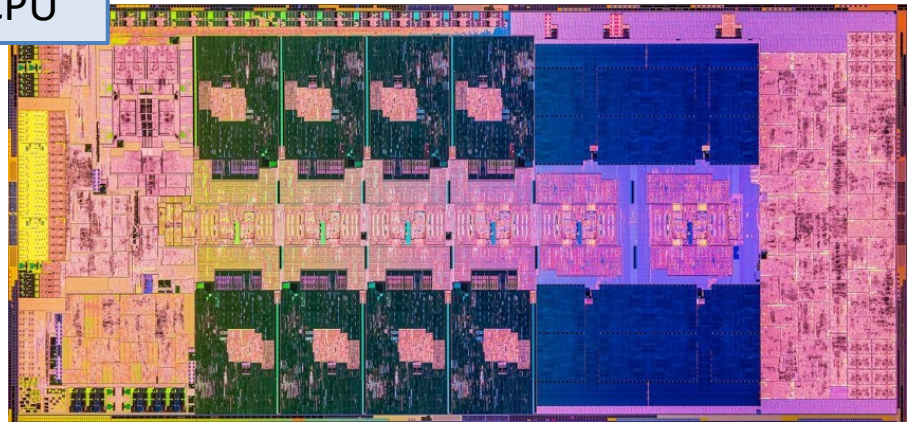
barrière de synchronisation

Parallèle

Séquentiel

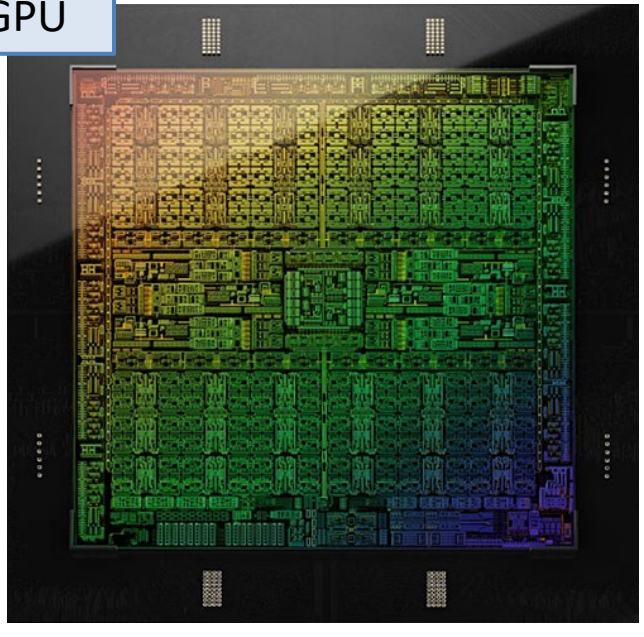
Architectures parallèles

CPU



Intel Raptor Lake Core i9-14900 (**2024**)
(10 nm – 257 mm^2)
24 Cœurs/32 Threads - 3,19 Ghz
64 ko/Core L1 - 36 Mo Shared L3

GPU



Nvidia Geforce RTX 4090 Ti (Ada Lovelace AD102-400-A1, **2023**)
(4 nm - 609 mm^2 - 76 10^9 T)
18176 CUDA Cores – 2,325 Ghz
24 Go GDDR6X - 96 Mo L2 - 18432 ko L1 - 1,5 Ghz

architecture généraliste / architecture spécialisée

Graphics Processing Unit (GPU) – 1/4

Schéma-bloc complet

AD102 Full

- 12 GPC (Graphic Processing Cluster)
- Mémoire
  - Bus 384 bits
  - 96 Mo L2 cache

Graphics Processing Unit (GPU) – 2/4

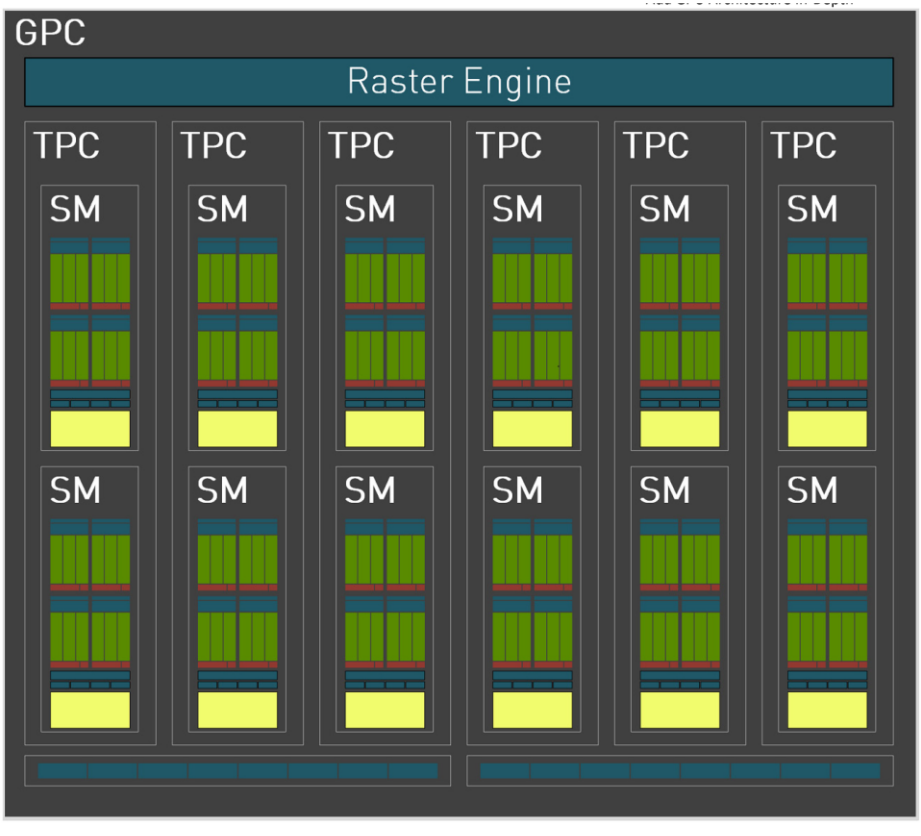


Schéma-bloc d'un GPC

- 6 TPC (Texture Processing Cluster)
  - 2 SM (Single Multiprocessor) par TPC
- 1 unité de rastérisation
- 16 ROP (Render Output unit)

Graphics Processing Unit (GPU) – 3/4

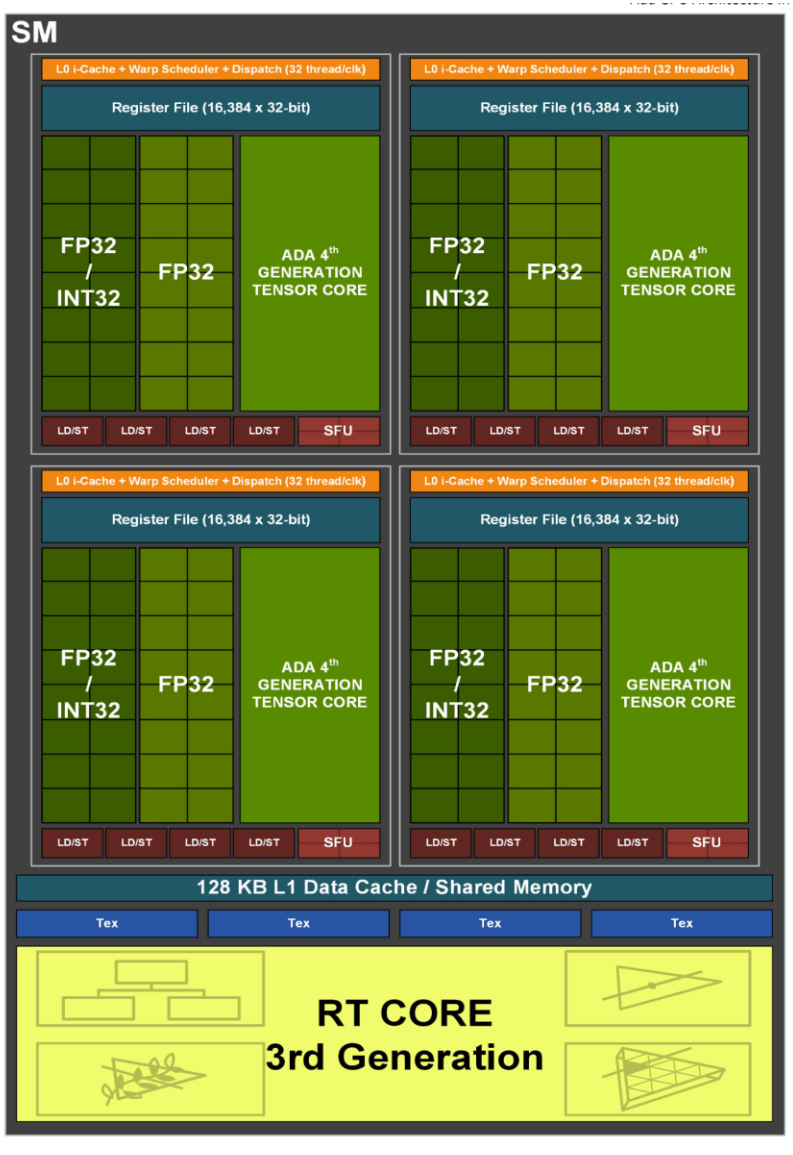
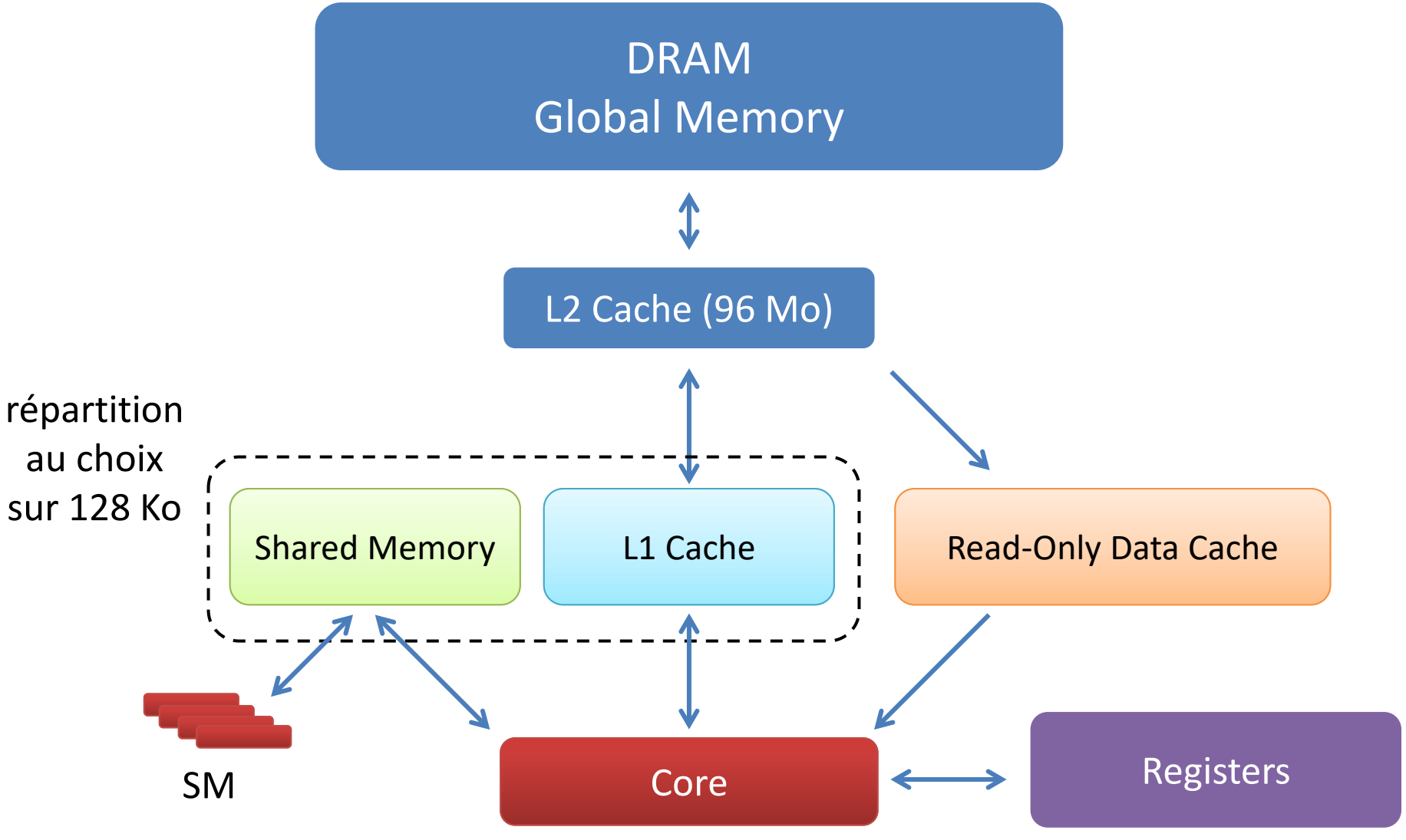


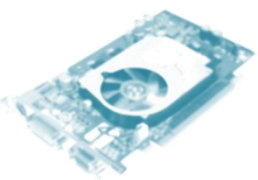
Schéma-bloc d'un SM

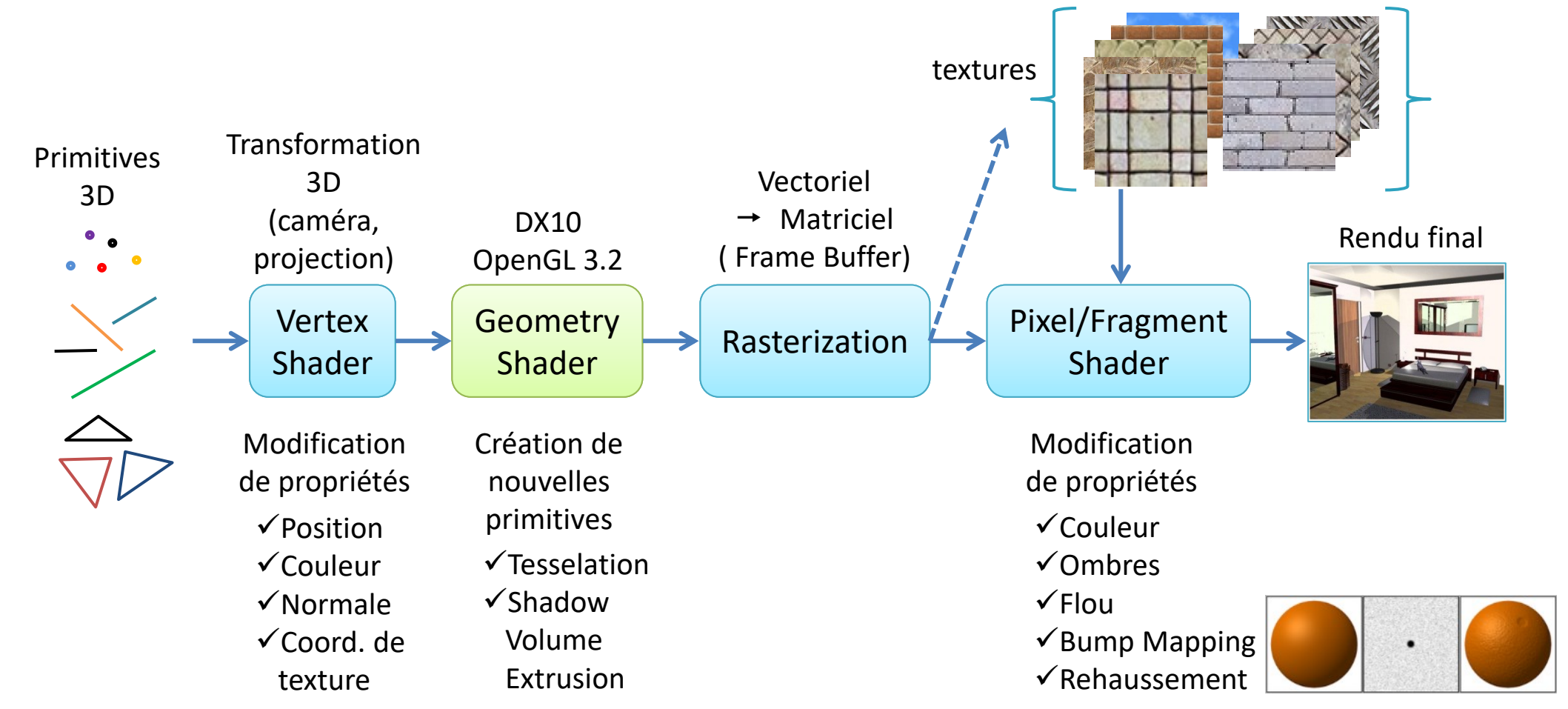
- 4 partitions
  - 16 FP32 (Cuda Core)
  - 16 FP32 en chemin partagé INT32
  - 16 INT32 en chemin partagé FP32
  - 1 SFU (Special Function Units)
  - Quelques FP64 (débit 1/64 du FP32)
  - 4 unités LD/ST (Load/Store)
  - 1 Tensor Core de 4ème génération
  - 1 ordonnanceur/dispatcheur (warp de 32 threads)
- 4 unités de Texturing
- 1 RT Core de 3ème génération (raytracing)

Graphics Processing Unit (GPU) – 4/4



Chaîne graphique 3D





Vertex par vertex ← parallélisme → **Pixel par pixel**

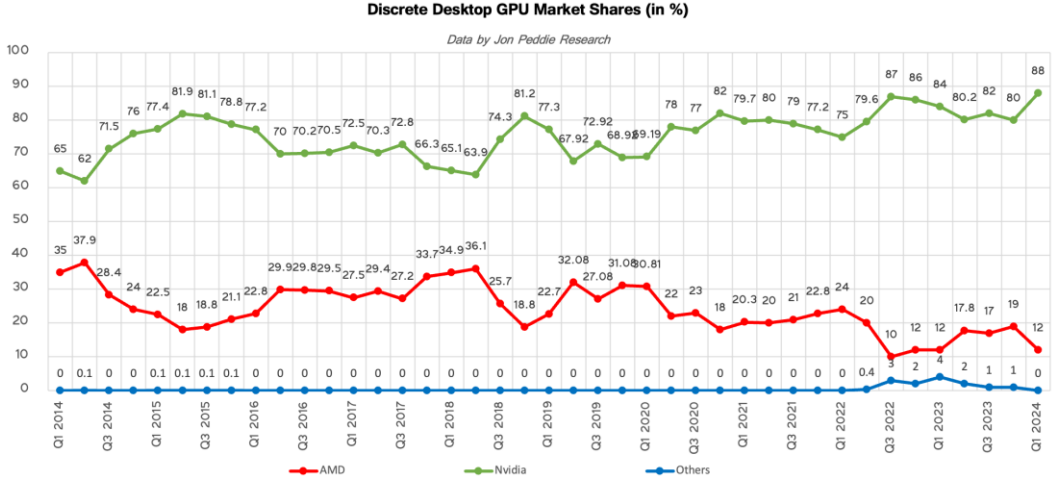
Shader : calculs programmables permettant le rendu plus réaliste, l'ajout d'effets spéciaux (explosion, rendu « toon »), …

GPU et constructeurs

- Différents types de GPU (ordinateur, tablette, smartphone)
  - dGPU (discrete graphical processor unit)
    - Carte graphique (extension sur bus)
  - iGPU (integrated graphical processor unit)
    - Puce spécialisée de type SoC (insérée sur la carte mère)
  - APU (accelerated processor unit)
    - Hybridation CPU et iGPU

Plus de 1 milliard par an

- Parts de marché des constructeurs principaux

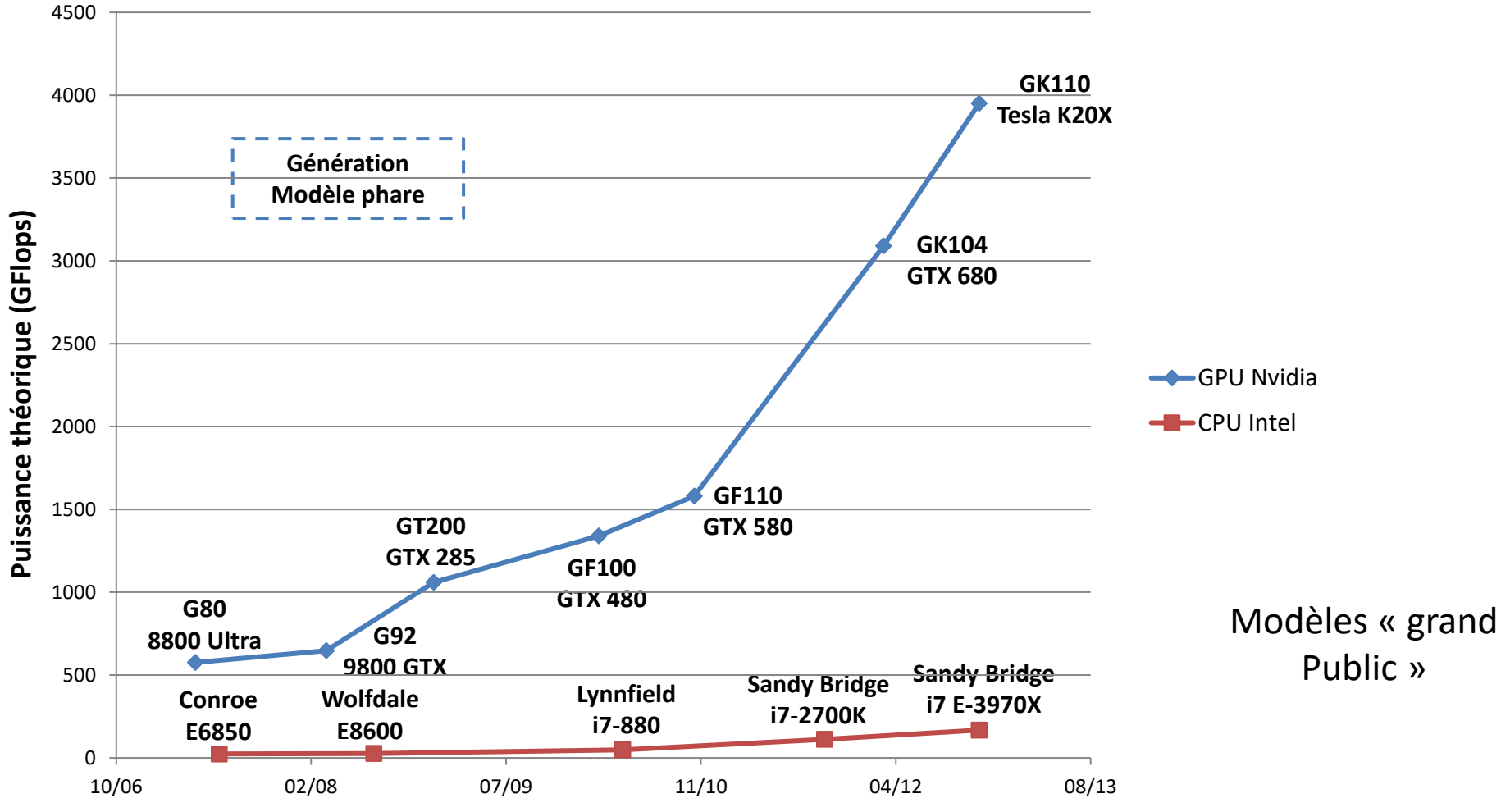


Leader technologique

Source : Jon Peddie Research (2024)

CPU vs GPU

Puissances de calcul comparées en virgule flottante simple précision



Modèles « grand Public »

Gamme NVIDIA

RTX 4090 Ti (95 TFlops SP) ~2000€
AD102, 18176 CUDA Cores - 24 Go GDDR6X

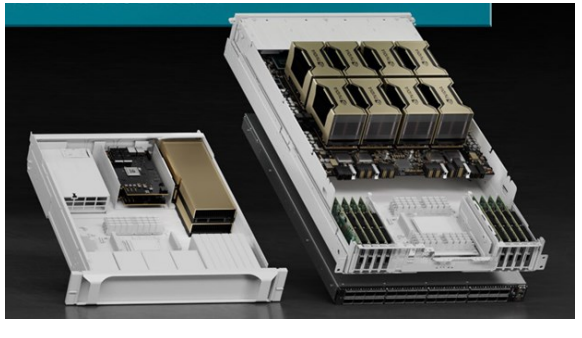
Carte graphique

Carte de calcul (pas de sortie vidéo)

2023

Cluster HPC

Hopper H100 SXM 80GB (67 TFlops SP) ~35k€
16896 CUDA Cores - 80 Go HBM3



256 × H100 8GPU (~17,1 PFlops SP)
~4,3 M CUDA Cores - 20 To HBM3

Supercalculateurs

www.top500.org

TITAN, Cray XK-7 (USA, ~61ème en 2024)

2012

Nvidia Tesla K20X + AMD Opteron 6274

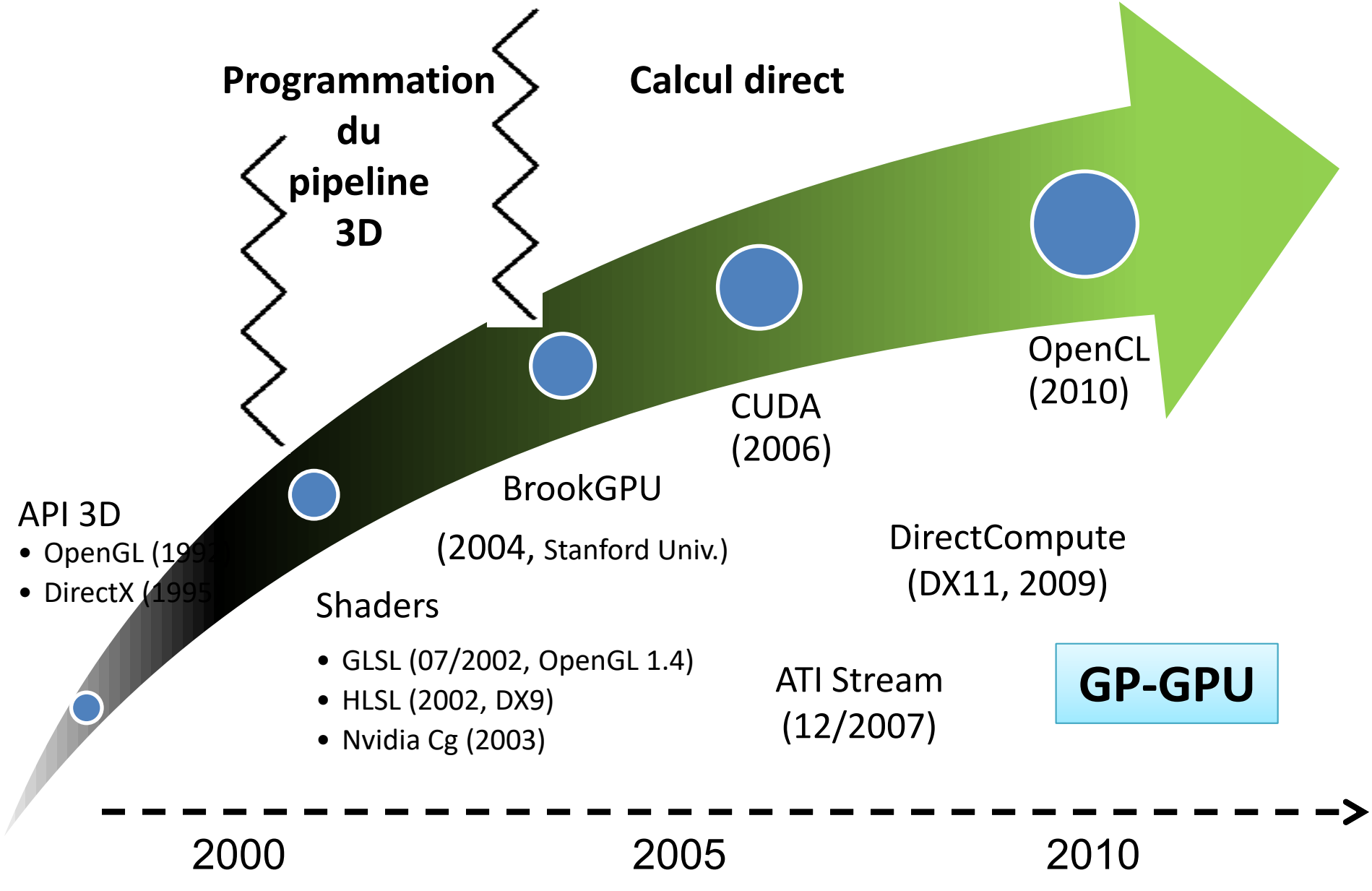
FRONTIER, HPE CRAY EX235A (USA, 1er en 2024)

2022

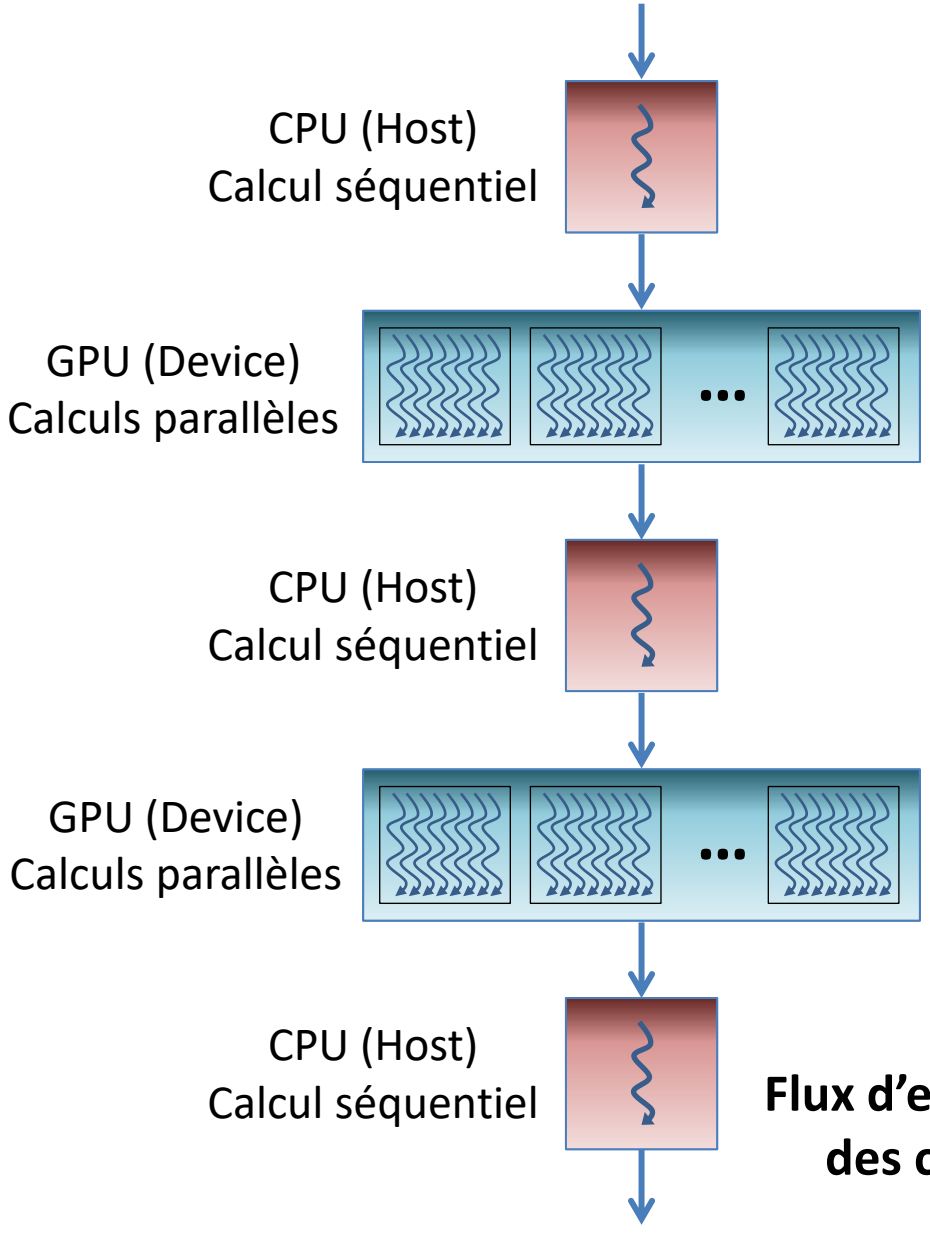
AMD Instinct MI250X + AMD EPYC 64C

	TITAN	FRONTIER
« Processeurs »	560640	8699904
Puissance (DP)	17,6 PFlops	1,21 ExFlops
Consommation	8,2 MW	22,8 MW
Efficacité (GFlops/W)	2,14	53,1

General Purpose-Processing on GPU

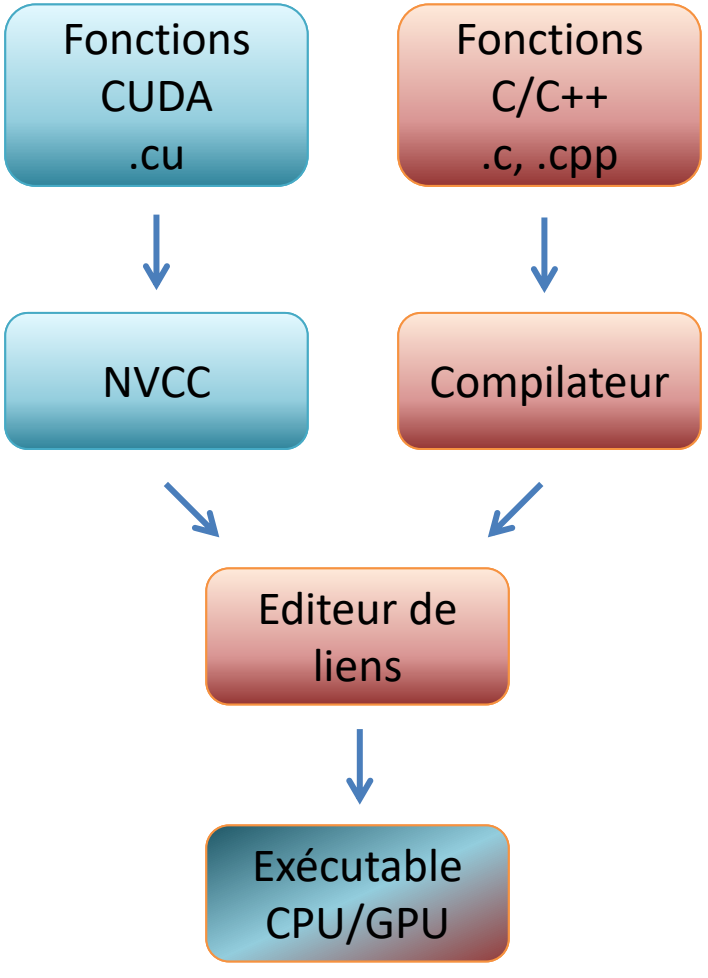


Modèle de programmation CUDA



Flux d'exécution des calculs

Du code à l'exécutable



Du GPU au thread…

- Le CPU (host) exécute des fonctions séquentielles et lance des calculs parallèles sur le GPU (device)
- Des passerelles (host/device) permettent l'échange de données entre le CPU et le GPU
- Le GPU exécute des fonctions (kernel) sous la forme d'instances parallèles (threads)
- Les instances sont organisées, selon une hiérarchie double, en grilles de blocs de threads (hiérarchie triple récente…)
- Les threads accèdent uniquement à la mémoire GPU dans différents modes (sauf accès direct à la DRAM…)

...du thread au GPU

Matériel

Core (SP)

SM

GPU

Logiciel

Registers

Local Memory

Thread

Global Memory

Bloc

Shared Memory

Global Memory

Grille

...

Global Memory

Threads (1/2)

- Un thread s'exécute sur un seul processor (SP/Core)
- Un bloc s'exécute sur un seul multi-processeur (SM)
  - Les threads s'exécutent de manière concurrente (parallèle) par groupes de 32 (warp)
  - Un multi-processeur peut exécuter plusieurs blocs
  - Les threads d'un bloc peuvent partager des données à travers la mémoire globale ou la mémoire partagée (durée de vie égale à celle du bloc), et peuvent se synchroniser efficacement
- Une grille s'exécute sur un seul GPU
  - Les blocs s'exécutent en parallèle ou en série dans un ordre quelconque
  - Les threads de différents blocs ne peuvent pas échanger de données et ne sont pas tous « en vie » simultanément

Threads (2/2)

- Les threads sont associés à un identifiant unique au sein d'un bloc selon un pavage 1D, 2D ou 3D

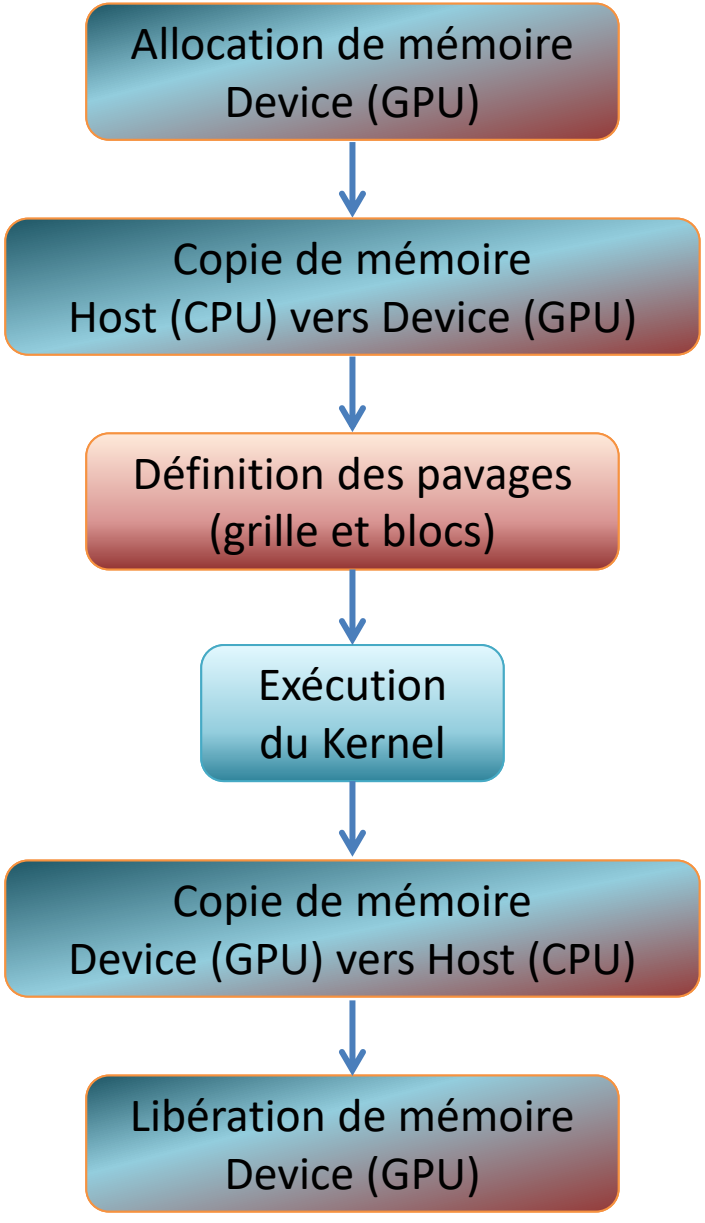
Max 1024 threads/bloc

- Les blocs sont associés à un identifiant unique au sein d'une grille selon un pavage 1D, 2D ou 3D

- Des variables « built-in » (« intégrées ») permettent d'accèder aux identifiants
  - threadIdx(.x, .y, .z) numéro du thread (dans la dimension)
  - blockDim(.x, .y, .z) nombre de threads (dans la dimension)
  - blockIdx(.x, .y, .z) numéro du bloc (dans la dimension)
  - dimGrid(.x, .y, .z) nombre de blocs (dans la dimension)

- Dimensions choisies selon une adéquation calculs/données

Etapes d'un calcul sur GPU



Opérateur point à point

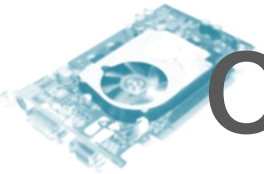
```
void gpu_square_main(float * v_in, int size, float * v_out)
{
   int nbThread = 512;
   dim3 dimBlock(nbThread);
   dim3 dimGrid((size+nbThread-1)/nbThread);
   float * v_in_cuda = NULL;

   cudaMalloc((void **)&v_in_cuda, size*sizeof(float));
   cudaMemcpy(v_in_cuda, v_in, size*sizeof(float), cudaMemcpyHostToDevice);

   gpu_square<<<dimGrid, dimBlock>>>(v_in_cuda, size);
   cudaDeviceSynchronize();

   cudaMemcpy(v_out, v_in_cuda, size*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(v_in_cuda);
}
```

```
__global__ void gpu_square(float * v, int size)
{
   int id =  blockIdx.x*blockDim.x+threadIdx.x;
   if  ( id < size ) v[id] = v[id]*v[id];
}
```



Opérateur à voisinage local (1 /4)

```
void gpu_conv2d_main(float * img_in, int width, int height, float * filter2d, int tx, int ty, float * img_out)
{
   int size = width*height, nbThreadx = 16, nbThready = 16;
   dim3 dimBlock(nbThreadx, nbThready);
   dim3 dimGrid((width+nbThreadx-1)/nbThreadx, (height+nbThready-1)/nbThready);
   float * img_in_cuda = NULL, * filter2d_cuda = NULL, * img_out_cuda = NULL;

   cudaMalloc((void **)&img_in_cuda, size*sizeof(float));
   cudaMemcpy(img_in_cuda, img_in, size*sizeof(float), cudaMemcpyHostToDevice);
   cudaMalloc((void **)&img_out_cuda, size*sizeof(float));
   cudaMemcpy(img_out_cuda, img_out, size*sizeof(float), cudaMemcpyHostToDevice);
   cudaMalloc((void **)&filter2d_cuda, tx*ty*sizeof(float));
   cudaMemcpy(filter2d_cuda, filter2d, tx*ty*sizeof(float), cudaMemcpyHostToDevice);

   gpu_conv2d<<<dimGrid, dimBlock>>>(img_in_cuda, width, height, filter2d_cuda, tx, ty, img_out_cuda);
   cudaDeviceSynchronize();

   cudaFree(img_in_cuda);
   cudaFree(filter2d_cuda);
   cudaMemcpy(img_out, img_out_cuda, size*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(img_out_cuda);
}
```

Opérateur à voisinage local (2/4)

```
__global__ void gpu_conv2d(float * img_in, int width, int height,
                           float * filter2d, int tx, int ty,
                           float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x,
       j =  blockIdx.y*blockDim.y+threadIdx.y, k, l, dtx = tx/2, dty = ty/2;
   float s = 0.0f;

   if  ( i >= dtx && i < width-dtx && j >= dty && j < height-dty )
      {
         for (l=-dty; l<=dty; l++)
            for (k=-dtx; k<=dtx; k++)
               s += img_in[(j-l)*width+i-k]*filter2d[(l+dty)*tx+k+dtx];
         img_out[j*width+i] = s;
      }
}
```

Effets de bords gérés → latences

Opérateur à voisinage local (3/4)

```
…
dim3 dimGrid((width-tx+1+nbThreadx-1)/nbThreadx, (height-ty+1+nbThready-1)/nbThready);
…
```

```
__global__ void gpu_conv2d(float * img_in, int width, int height,
                           float * filter2d, int tx, int ty,
                           float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x,
       j =  blockIdx.y*blockDim.y+threadIdx.y, k, l, dtx = tx/2, dty = ty/2;
   float s = 0.0f;

   if  ( i < width-tx+1 && j < height-ty+1 )
      {
         for (l=-dty; l<=dty; l++)
            for (k=-dtx; k<=dtx; k++)
               s += img_in[(j+dty-l)*width+i+dtx-k]*filter2d[(l+dty)*tx+k+dtx];
         img_out[(j+dty)*width+i+dtx] = s;
      }
}
```

« Pas » d'effets de bords

Opérateur à voisinage local (4/4)

```
__global__ void gpu_conv2d(float * img_in, int width, int height,
                           float * filter2d, int tx, int ty,
                           float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x,
       j =  blockIdx.y*blockDim.y+threadIdx.y,
       n = threadIdx.y*blockDim.x+threadIdx.x, k, l, dtx = tx/2, dty = ty/2;
   float s = 0.0f;
   __shared__ float f2d[1000];

   if ( n < tx )
      for (l=0; l<ty; l++) f2d[l*tx+n] = filter2d[l*tx+n];
   __syncthreads();
   if  ( i < width-tx+1 && j < height-ty+1 )
      {
         for (l=-dty; l<=dty; l++)
            for (k=-dtx; k<=dtx; k++)
               s += img_in[(j+dty-l)*width+i+dtx-k]*f2d[(l+dty)*tx+k+dtx];
         img_out[(j+dty)*width+i+dtx] = s;
      }
}
```

Filtre en mémoire partagée

Performances

Intel i7-13700H - 2,4 Ghz – 14/20 Cores/Threads - 192 GFlops (Ordinateur portable)
vs
Nvidia Geforce RTX 4070M - 4608 Cuda Cores – 1230/2175 Mhz - 15,6 TFlops

Transfert Host/Device inclus ~16,6 ms

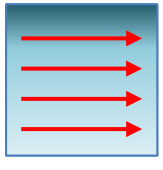
	3×3	11×11	21×21	31×31	41×41
CPU mono-thread (écart-type)	107 ms (0,273 ms)	802 ms (2,21 ms)	3,06 s (7,23 ms)	7,02 s (8,07 ms)	12,5 s (6,97 ms)
GPU avec bords (écart-type)	17,6 ms (0,064 ms)	20,4 ms (0,069 ms)	31,1 ms (0,208 ms)	44,1 ms (0,171 ms)	65,6 ms (0,063 ms)
GPU sans bords (écart-type)	17,6 ms (0,038 ms)	20,3 ms (0,117 ms)	31,5 ms (0,211 ms)	44,4 ms (0,066 ms)	65,4 ms (0,111 ms)
GPU SharedMemory (écart-type)	17,8 ms (0,027 ms)	20,5 ms (0,029 ms)	32,1 ms (0,044 ms)	44,7 ms (0,081 ms)	

Image 4096×4096 en virgule flottante simple précision, temps moyennés sur 10 séries de 250 itérations

Ecueils – Coalescence (1/2)

Mémoire globale

- Parcours en ligne, colonne par colonne (accès coalescent ou contigu)

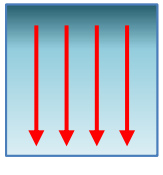


Rapide

```
__global__ void gpu_distance(float * img, int width, int height)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x, j =  blockIdx.y*blockDim.y+threadIdx.y;

   if  ( i < width && j < height ) img[j*width+i] = (float)(i*i+j*j);
}
```

- Parcours en colonne, ligne par ligne (accès non coalescent)



```
__global__ void gpu_distance(float * img, int width, int height)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.y, j =  blockIdx.y*blockDim.y+threadIdx.x;

    if  ( i < width && j < height ) img[j*width+i] = (float)(i*i+j*j);
}
```

Ecueils – Coalescence (2/2)

Intel i7-13700H - 2,4 Ghz – 14/20 Cores/Threads - 192 GFlops (Ordinateur portable)
vs
Nvidia Geforce RTX 4070M - 4608 Cuda Cores – 1230/2175 Mhz - 15,6 TFlops

	Distance	Sinus(Distance)
CPU mono-thread	3,61 ms	128 ms
GPU accès coalescent	0,269 ms	1,36 ms
GPU accès non coalescent	0,335 ms	1,36 ms

Transfert Host/Device ~16,6 ms

coalescence marginalisée par des calculs « lourds »

Image 4096×4096 en virgule flottante simple précision, temps moyennés sur 10 séries de 50 itérations

Ecueils – Mémoire partagée (1 / 5)

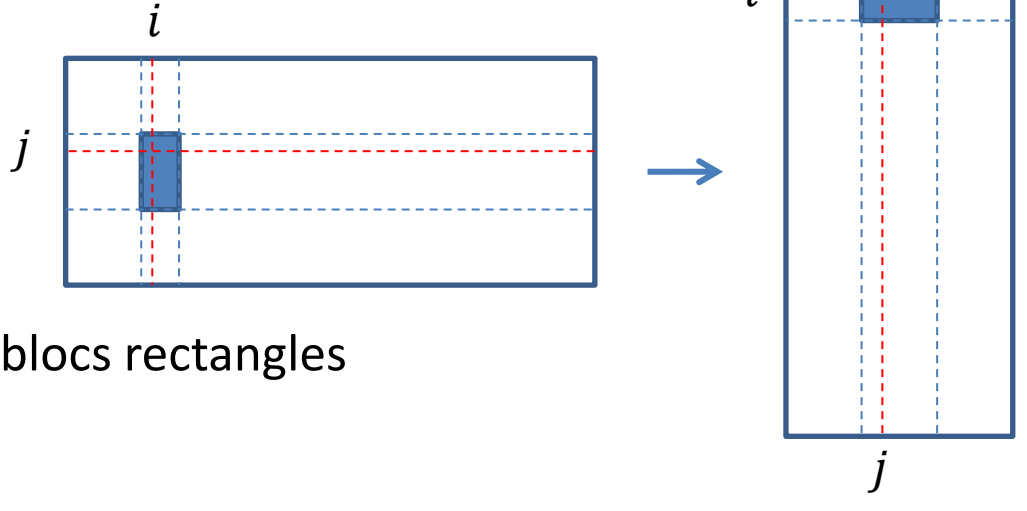
Transposition

```
__global__ void gpu_transposition(float * img_in, int width, int height, float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x, j =  blockIdx.y*blockDim.y+threadIdx.y;

   if  ( i < width && j < height ) img_out[i*height+j] = img_in[j*width+i];
}
```

accès non coalescent

accès coalescent



blocs rectangles

Version naïve

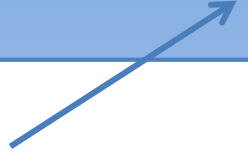
Ecueils – Mémoire partagée (2/5)

```
…
#define BLOCK_SIZE 16
…
```

```
__global__ void gpu_transposition(float * img_in, int width, int height, float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x, j =  blockIdx.y*blockDim.y+threadIdx.y;
   __shared__ tile[BLOCK_SIZE][BLOCK_SIZE];

   if ( i < width && j < height ) tile[threadIdx.y][threadIdx.x] = img_in[j*width+i];
    __syncthreads();

   i = blockIdx.y*blockDim.y+threadIdx.x;
   j = blockIdx.x*blockDim.x+threadIdx.y;
   if ( j < width && i < height ) img_out[j*height+i] = tile[threadIdx.x][threadIdx.y];
}
```



accès coalescent

- Inversion du rôle de x et y (threads)
  - Blocs nécessairement carrés
  - Accès séquentiels à la mémoire globale en écriture

Mémoire partagée

Ecueils – Mémoire partagée (3/5)

```
…
#define BLOCK_SIZE 32
…
```

```
__global__ void gpu_transposition(float * img_in, int width, int height, float * img_out)
{
   int i =  blockIdx.x*blockDim.x+threadIdx.x, j =  blockIdx.y*blockDim.y+threadIdx.y;
   __shared__ tile[BLOCK_SIZE][BLOCK_SIZE+1];

   if ( i < width && j < height ) tile[threadIdx.y][threadIdx.x] = img_in[j*width+i];
    __syncthreads();

   i = blockIdx.y*blockDim.y+threadIdx.x;
   j = blockIdx.x*blockDim.x+threadIdx.y;
   if ( j < width && i < height ) img_out[j*height+i] = tile[threadIdx.x][threadIdx.y];
}
```

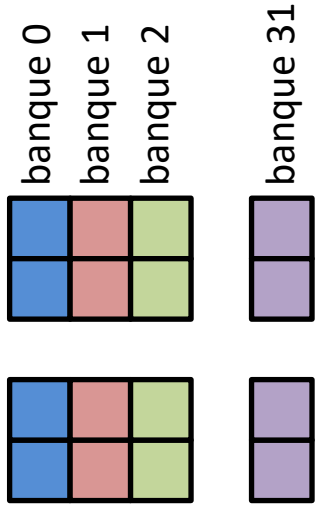
- Elimination/Limitation des latences
  - Allocation adéquate de mémoire partagée
  - Accès threads/mémoire partagée non concurrentiels

Mémoire partagée sans conflits d'accès

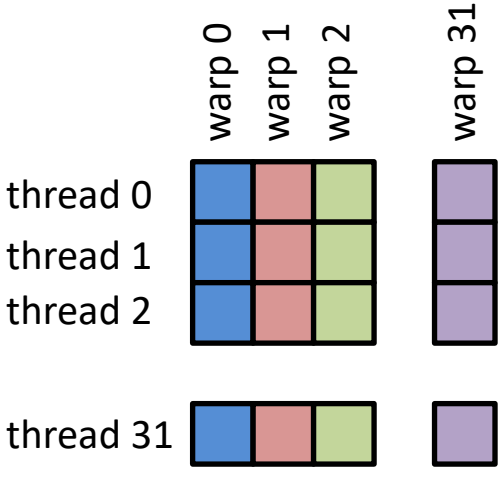
Ecueils – Mémoire partagée (4/5)

Organisation en banques

- Mémoire partagée structurée en groupes (banques)
  - Alternance de banques de dimension fixe
  - Bande passante d'une banque limitée par cycle

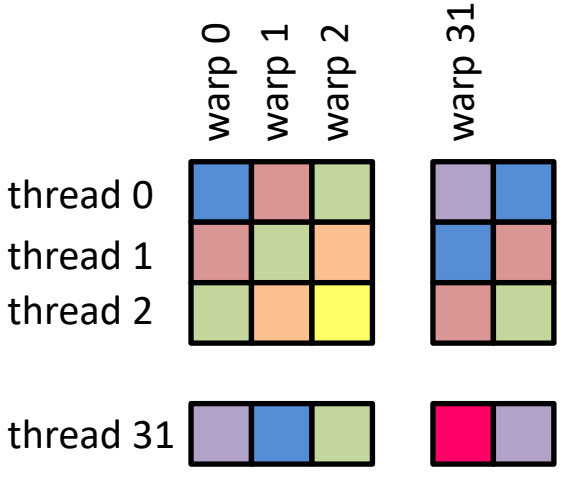


`tile[32][32]`

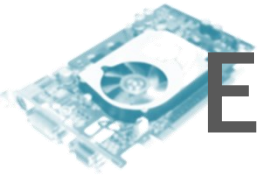


Accès à la même banque
des threads d'un warp
→ conflits et latences

`tile[32][33]`



Accès à des banques différentes
des threads d'un warp
→ pas de conflits



Ecueils – Mémoire partagée (5/5)

Intel i7-13700H - 2,4 Ghz – 14/20 Cores/Threads - 192 GFlops (Ordinateur portable)
vs
Nvidia Geforce RTX 4070M - 4608 Cuda Cores – 1230/2175 Mhz - 15,6 TFlops

	Transposition
CPU (mono-thread)	419 ms
GPU version naïve	0,741 ms
GPU mémoire partagée avec conflits	0,573 ms
GPU mémoire partagée sans conflits	

Image 4096×4096 en virgule flottante simple précision, temps moyennés sur 10 séries de 50 itérations

Mémoire unifiée (1 /2)

CPU

```
…
matA = (float *)calloc(widthA*heightA, sizeof(float)); // idem pour matB et matC
// Initialisation de matA et matB,
mat_mul_gpu_main(matA, widthA, heightA, matB, widthB, matC);
…
```

CPU/GPU

```
void mat_mul_gpu_main(float * matA, int widthA, int heightA, float * matB, int widthB, float *matC)
{
   int sizeA = widthA*heightA, sizeB = widthB*widthA, sizeC = widthB*heightA, nbThreadx = 16, nbThready = 16;
   float * cuMatA = NULL, * cuMatB = NULL, * cuMatC = NULL;
   dim3 dimBlock(nbThreadx, nbThready);
   dim3 dimGrid((widthB+nbThreadx-1)/nbThreadx, (heightA+nbThready-1)/nbThready);

   cudaMalloc((void **)&cuMatA, sizeA, sizeof(float));
   cudaMemcpy(cuMatA, matA, sizeA*sizeof(float), cudaMemcpyHostToDevice);
   cudaMalloc((void **)&cuMatB, size*sizeof(float));
   cudaMemcpy(cuMatB, matB sizeB*sizeof(float), cudaMemcpyHostToDevice);
   cudaMalloc((void **)&cuMatC, sizeC*sizeof(float));
   mat_mul_gpu<<<dimGrid, dimBlock>>>(cuMatA, widthA, heightA, cuMatB, widthB, cuMatC);
   cudaDeviceSynchronize();
   cudaMemcpy(matC, (void **)&cuMatC, sizeC*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(cuMatA); cudaFree(cuMatB); cudaFree(cuMatC);
}
```

Mémoire unifiée (2/2)

```
…
cudaMallocManaged((void **)&matA, widthA*heightA*sizeof(float)); // idem pour matB et matC
// Initialisation de matA et matB,
mat_mul_gpu_main(matA, widthA, heightA, matB, widthB, matC);
…
```

CPU/GPU

```
void mat_mul_gpu_main(float * matA, int widthA, int heightA, float * matB, int widthB, float *matC)
{
   int sizeA = widthA*heightA, sizeB = widthB*widthA, sizeC = widthB*heightA, nbThreadx = 16, nbThready = 16;
   dim3 dimBlock(nbThreadx, nbThready);
   dim3 dimGrid((widthB+nbThreadx-1)/nbThreadx, (heightA+nbThready-1)/nbThready);

   mat_mul_gpu<<<dimGrid, dimBlock>>>(matA, widthA, heightA, matB, widthB, matC);
   cudaThreadSynchronize();
}
```

CPU/GPU

A partir de la version CUDA 6.5

Pour aller plus loin (1/2)

- Librairies spécialisées
  - cuBLAS (calculs algébriques)
  - cuFFT (transformées de Fourier)
  - cuRAND (nombres aléatoires)
  - cuSPARSE (matrices creuses)
  - NPP (algorithmes TSI)

- Outils d'aide au développement
  - Cuda-Memcheck (erreurs d'accès à la mémoire GPU)
  - Cuda-GDB, Nvidia Nsight VS/Eclipse Edition (débogueurs)
  - Nvidia Visual Profiler (optimisation)
  - CUDA Occupancy Calculator (taux d'activité parallèle)

Pour aller plus loin (2/2)

- Voies d'optimisation
  - Respecter au mieux les spécifications matérielles (nombre de threads par bloc, …)
  - Eviter les divergences de calculs (branchements à temps de traitement variable)
  - Limiter les transferts mémoire CPU (Host)/GPU (Device)
  - Adresser des blocs de mémoire contigus et alignés en mémoire globale
  - Utiliser la mémoire partagée (! conflits de banques)
  - Mettre en œuvre des ensembles (calculs et transferts) par parties asynchrones (stream)
  - …
- Alternatives à CUDA : OpenCL, OpenAcc, …

A retenir

- GPU ? (définition)
- Adéquation avec le TSI accéléré (structure interne et parallélisme)
- Possibilités (performances comparées, histoire et solutions disponibles)
- Mise en œuvre (principes et bases, outils logiciels)