Exploitez efficacement l'architecture GPU

Approche spécifique pour les GPUs NVIDIA

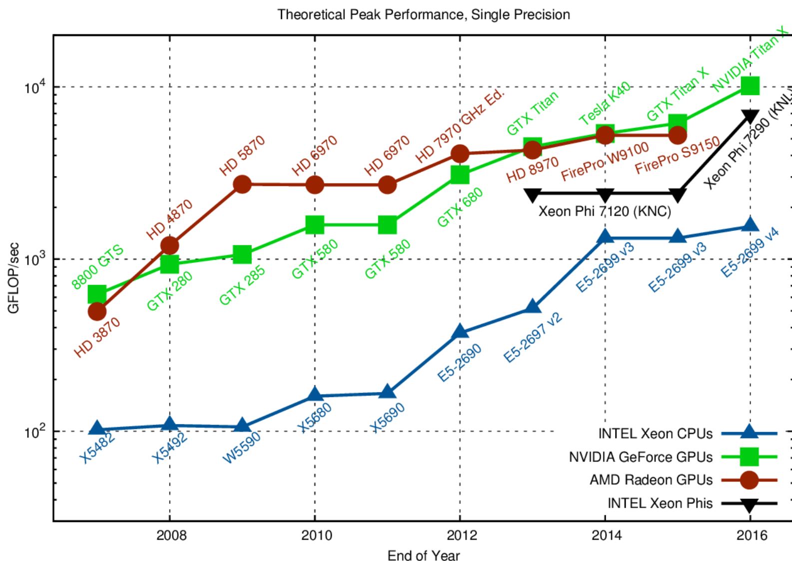
Florian Gouin

14 Novembre 2019

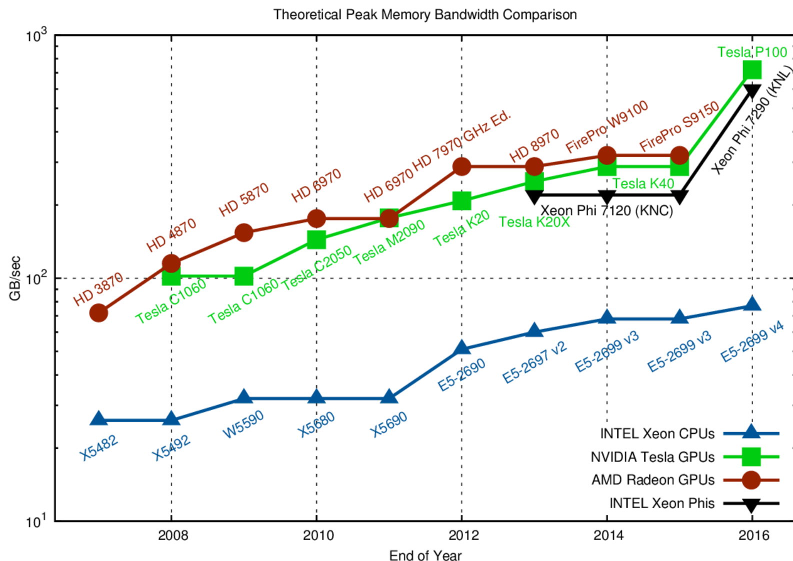
Journée Calcul, thème GPUs – Lyon

Parallélisme à niveaux multiples

Comparaison des performances théoriques calculatoires



Comparaison des performances théoriques de transfert mémoire

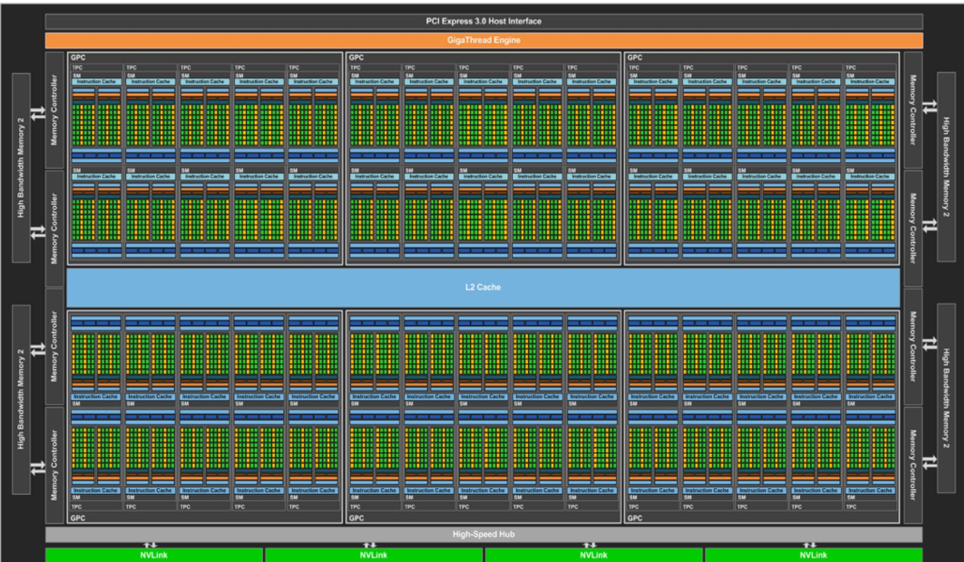


Volta Streaming Multiprocessor (SM)

Architecture GV100

Pascal Streaming Multiprocessor (SM)

Architecture GP100



Triangle des limitations architecturales

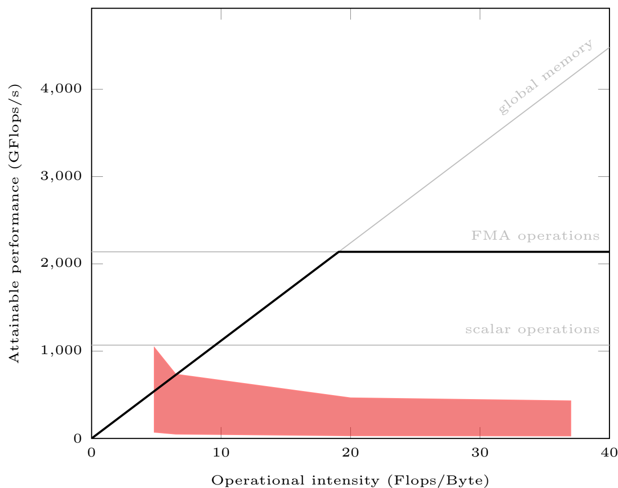
Limitations :

- Flot d'instructions
- Flot de communications mémoires
- Flot de calculs

Solutions :

- *Roofline model* de Patterson
- NVProf de Nvidia

Exemple : Roofline Model



Espace des instances de *thread*/d'itérations

Caractéristiques :

- fixe
- dense
- normalisé

Dimensions :

- Blocks : 3 dimensions *(par)*
- Threads : 3 dimensions *(séq)*
- Total : 6 dimensions

Execution :

- Block → SM
- Thread → Cuda Cores
- Warps : 32 threads

Taille de l'espace des blocks :

$$\begin{cases} b = b_0 \times b_1 \times b_2 \\ b_0 < 2\,147\,483\,647 \\ b_1 < 65535 \\ b_2 < 65535 \end{cases}$$

Taille de l'espace des threads :

$$\begin{cases} t = t_0 \times t_1 \times t_2 \\ t < 1024 \\ t_0 < 1024 \\ t_1 < 1024 \\ t_2 < 64 \\ t\%32 = 0 \\ t > 4 \times 32 \end{cases}$$

Vérifier la pression des registres !

- Nvidia Cuda Kernel Occupancy calculator (Excel)
- NVProf Nvidia

Strip Mining

```
1 __global__ void kernel(){
2         int global_coord = blockIdx.x * blockDim.x + threadIdx.x;
3 }
```

Tiling

```
1 __global__ void kernel(){
2         int3 global_coords = {
3                         blockIdx.x * blockDim.x + threadIdx.x,
4                         blockIdx.y * blockDim.y + threadIdx.y,
5                         blockIdx.z * blockDim.z + threadIdx.z};
6 }
```

Filtre moyenneur (CPU)

```
1  #define IMG_W 1920
2  #define IMG_H 1080
3  #define K_RAY_X 4
4  #define K_RAY_Y 4
5  #define K_LINE (K_RAY_X*2+1)
6  #define K_SIZE (K_LINE*K_LINE)
7
8  for (int y = K_RAY_Y; y < IMG_H - K_RAY_Y; ++y){
9          for(int x = K_RAY_X; x < IMG_W - K_RAY_X; ++x){
10                 int val = 0;
11                 for (int ky = -K_RAY_Y; ky <= K_RAY_Y; ++ky){
12                         for(int kx = -K_RAY_X; kx <= K_RAY_X;
                           ↪  ++kx){
13                                 val += in[(y+ky) * IMG_W +
                                   ↪  (x+kx)];
14                         }
15                 }
16                 out[y * IMG_W + x] = val / (K_SIZE);
17         }
18 }
```

Filtre moyenneur (GPU / Tiling)

```
1   dim3 threads = {32,32,1};
2   dim3 blocks = {ceil(IMG_W/threads.x),ceil(IMG_H/threads.y),1};
3   kernel<<<blocks,threads>>>(in, out);
4           ...
5   __global__ void kernel(char in*, char out*){
6           int2 pos = { blockIdx.x * blockDim.x + threadIdx.x,
7                        blockIdx.y * blockDim.y + threadIdx.y};
8           if ( pos.x <K_RAY_X || pox.x >= IMG_W - K_RAY_X ||
9                pos.y <K_RAY_Y || pos.y >= IMG_H - K_RAY_Y )
10                  return;
11
12          int val = 0;
13          for (int ky = -K_RAY_Y; ky <= K_RAY_Y; ++ky){
14                  for(int kx = -K_RAY_X; kx <= K_RAY_X; ++kx){
15                          val += in[(y+ky) * IMG_W + (x+kx)];
16                  }
17          }
18          out[pos.y * IMG_W + pos.x] = val / K_SIZE;
19  }
```

Filtre moyenneur (GPU)

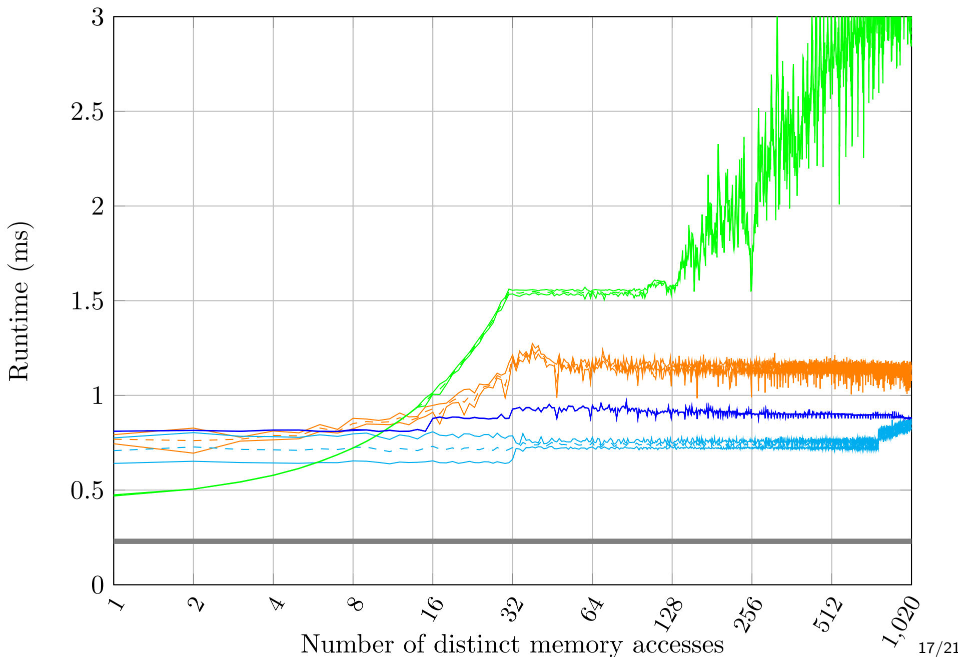
```
dim3 threads = {K_LINE, K_LINE, 1};
dim3 blocks = {IMG_W, IMG_H, 1};
kernel<<<blocks,threads>>>(in, out);
        ...
__global__ void kernel(char in*, char out*){
        if(blockIdx.x < K_RAY_X || blockIdx.x >= IMG_W-K_RAY_X ||
           blockIdx.y < K_RAY_Y || blockIdx.y >= IMG_H-K_RAY_Y)
                return;

        int gPos = blockIdx.y * IMG_W + blockIdx.x;
        int kPos = (blockIdx.y+threadIdx.y-K_RAY_Y-1) * IMG_W +
        ↪  (blockIdx.x+threadIdx.x-K_RAY_X-1);
        out[gPos] = atomicAdd(in, kPos);

        __syncthreads();
        if((threadIdx.x | threadIdx.y) == 0)
                out[gPos] = out[gPos] / K_SIZE;
}
```

Filtre moyenneur (GPU / Tiling + Texture)

```
1  texture <char, 2, CudaReadModeElementType> in;
2  dim3 threads = {32,32,1};
3  dim3 blocks = {ceil(IMG_W/threads.x),ceil(IMG_H/threads.y),1};
4  kernel<<<blocks,threads>>>(out);
5          ...
6  __global__ void kernel(char out*){
7          int2 pos = { blockIdx.x * blockDim.x + threadIdx.x,
8                       blockIdx.y * blockDim.y + threadIdx.y};
9          if ( pos.x <K_RAY_X || pox.x >= IMG_W - K_RAY_X ||
10              pos.y <K_RAY_Y || pos.y >= IMG_H - K_RAY_Y )
11                 return;
12
13         int val = 0;
14         for (int ky = -K_RAY_Y; ky <= K_RAY_Y; ++ky){
15                 for(int kx = -K_RAY_X; kx <= K_RAY_X; ++kx){
16                         val += tex2D(in,y+ky,x+kx);
17                 }
18         }
19         out[pos.y * IMG_W + pos.x] = val / K_SIZE;
20 }
```


Runtime (ms)
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
32
64
128
256
512
1,020
Number of distinct memory accesses

Filtre moyenneur (GPU / Tiling + Texture + Constant Mem)

```
1  __constant__ char kernel[K_LINE*K_LINE];
2  texture <char, 2, CudaReadModeElementType> in;
3  dim3 threads = {32,32,1};
4  dim3 blocks = {ceil(IMG_W/threads.x),ceil(IMG_H/threads.y),1};
5  kernel<<<blocks,threads>>>(out);
6          ...
7  __global__ void kernel(char out*){
8          int2 pos = { blockIdx.x * blockDim.x + threadIdx.x,
9                       blockIdx.y * blockDim.y + threadIdx.y};
10         if ( pos.x <K_RAY_X || pox.x >= IMG_W - K_RAY_X ||
11              pos.y <K_RAY_Y || pos.y >= IMG_H - K_RAY_Y )
12                 return;
13
14         int val = 0;
15         for (int ky = -K_RAY_Y; ky <= K_RAY_Y; ++ky){
16                 for(int kx = -K_RAY_X; kx <= K_RAY_X; ++kx){
17                         val += tex2D(in,y+ky,x+kx) *
                           ↪  kernel[(ky+K_RAY_Y) * K_LINE +
                           ↪  (kx+K_RAY_X)] ;
18                 }
```

Optimisation du flot de contrôle

- Branching
- Concurrence CPU/GPU
- Concurrence GPU/GPU
- Kernel Occupancy
- Dépendance registres
  - 24 warps par block (768 threads)
- Unrolling
- Transferts asynchrones de données (Direct Memory Access (DMA)) Concurrence Mémoire/GPU

Selection des espaces mémoire

- Mémoire globale Coalescence des accès mémoire + alignement
- Shared Memory
  - Données creuses
  - Mémoire Cache manuelle
- Constant Memory
  - Réutilisation de données dans un warp
- Texture / Surface Memory
  - interpolation
  - gestion des bords
  - cache optimisé 2D

Exploitez efficacement l’architecture GPU

Approche spécifique pour les GPUs NVIDIA

Florian Gouin

14 Novembre 2019

Journée Calcul, thème GPUs – Lyon