Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Dév GPGPU

Corrections TP n°2

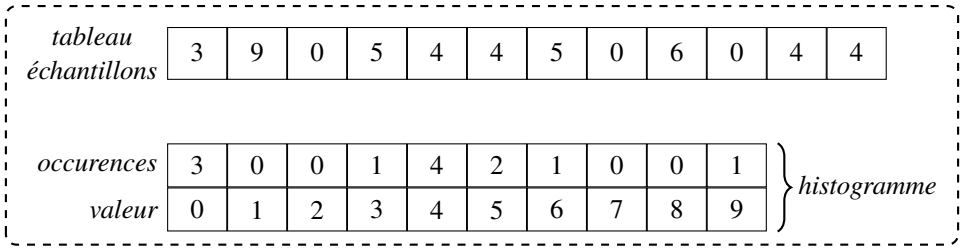
Cuda grilles, blocs & mémoire partagée

Histogramme

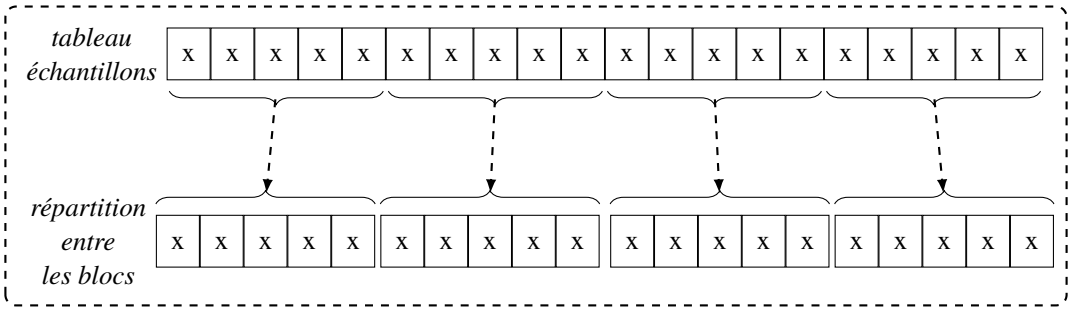
On a un tableau d'échantillons sur 10bits, c-à-d dont la valeur est comprise entre 0 et 1023 ($2^{10} = 1024$).

On veut en réaliser **l'histogramme**, c-à-d compter le nombre de fois où chaque valeur apparait :

- ▷ on parcourt les cases du tableau de valeurs ;
- ▷ pour chaque valeur rencontrée on augmente le nombre d'occurences associé.



On veut répartir le tableau d'échantillons entre différents blocs :



1 – Écrivez le programme réalisant le **calcul de l'histogramme** et affichant les 10 premières valeurs.

```
#!/usr/bin/python3

import random, struct, sys

nom_fichier = 'data.bin'
nombre_données = 33554432
taille_échantillon = 2**10

try:
    f = open(nom_fichier, 'rb')
except Exception as e:
    print(e.args)
    sys.exit(1)

histogramme = [0] * taille_échantillon

for i in range(nombre_données):
    entier = f.read(4)
    if not entier:
        break
    valeur = struct.unpack('i', entier)[0]
    histogramme[valeur] += 1
f.close()

print(histogramme[:10])
```

On choisit une taille pas trop grande pour certaines cartes des salles de TP
Ici $2^{25} = 33554432$

Première méthode

On veut utiliser la méthode où les *threads* :

- ▷ sont associées à une **tranche de valeurs** du tableau d'échantillons : une thread peut traiter seule une séquence de valeurs du tableau d'échantillons ;
- ▷ partagent l'accès au tableau histogramme.

2 – a. Est-ce qu'un **problème** peut survenir lors du travail de chaque *thread* ?
Certaines threads vont entrer en « compétition *» pour l'accès aux cases du tableau* `histogramme` *et créer une «* race condition *» : deux threads trouvant la même valeur d'échantillon vont vouloir toutes les deux incrémenter la case du tableau.*
Pour organiser l'accès des threads à une même case de tableau on va utiliser l'instruction `atomicAdd(int *,int)` *qui évite les corruptions d'accès en définissant une opération d'addition* ***non interruptible*** *par une autre thread (exclusion mutuelle pour la modification de la variable).*

b. Écrivez le *kernel* CUDA utilisant le fichier de données générées par le programme Python et calculant l'histogramme sur ces données;
Vous ferez varier la définition de la grille, des blocs et des tranches, « *SLICE* » :

```
 1 #define N 33554432
 2 #define SAMPLE 1024
 3 #define filename "data.bin"
 4
 5 __global__ void histogramme( int *t, int *h, int slice ) {
 6   int pos = ((blockIdx.x * blockDim.x) + threadIdx.x) * slice;
 7   for(int i = 0; i < slice; i++)
 8   {
 9       atomicAdd(h+t[pos+i], 1);
10   }
11 }
```

décalage dans le tableau de données suivant le numéro du bloc

Ici, une thread traite une tranche de 512 échantillons.

Pour chaque variation vous noterez le temps d'exécution obtenu avec `nsys` suivant la syntaxe de `nvprof`.

```
xterm
$ nsys nvprof --print-gpu-trace ./compute_histo
```

c. Expliquez les différents résultats obtenus.
Pour chaque exécution vous afficherez les 10 premières valeurs de votre histogramme résultat pour le comparer aux résultats obtenus avec le programme Python afin de valider votre implémentation.

kernel		slice	nb opérations	temps
64	1024	512	$2^6 * 2^{10} * 2^9 = 2^{25}$	7ms
64	512	1024	$2^6 * 2^9 * 2^{10} = 2^{25}$	6.8ms
64	256	2048	$2^6 * 2^8 * 2^{11} = 2^{25}$	6.7ms

Ici, on fait varier le nombre d'échantillons que traite chaque thread, et on observe que les résultats sont identiques.

On peut essayer d'améliorer les performances en « économisant *» la boucle* `for`*. En effet, une boucle en assembleur pour le GPU consiste en une addition, incrémenter la valeur du compteur, un test, savoir si on a fini, et un branchement pour revenir en arrière si on a justement pas fini.*
Ces 3 instructions peuvent représenter un coût si par exemple on a une seule instruction dans la boucle car pour chaque occurence de la boucle on a 4 instructions (3 pour la boucle et 1 pour le travail).
Si on « déroule *» la boucle, c-à-d si on remplace plusieurs occurences de la bocule par les instructions de la boucle on diminue l'importance des instructions liées à la boucle.*
Exemple si on a 100 occurences de la boucle, on réalise, pour un travail de juste une instruction, un total de $100*4 = 400$ *instructions. Si on déroule la boucle en dupliquant le travail réalisé dans chaque occurence on a un total de seulement* $100 * 1 = 100$ *instructions (on a éliminé le test, l'incrément et le branchement).*
Par contre le code obtenu est presque 25 fois plus grand (100 instructions contre 4 initialement).
Pour le faire automatiquement on dispose d'une directive du compilateur `#pragma unroll` *avec le paramètre indiquant la valeur à dérouler. Cette valeur doit être supérieure à la valeur minimale possible de la boucle.*

```
__global__ void histogram_unroll( int *t, int *h, int slice ) {
  int pos = ((blockIdx.x * blockDim.x) + threadIdx.x) * slice;
#pragma unroll 512
  for(int i = 0; i < slice; i++)
  {
      atomicAdd(h+t[pos+i], 1);
  }
}
```

On peut vérifier dans le code assembleur le résultat de l'opération :

Code du kernel « déroulé *» :*

```
   Function : _Z16histogram_unrollPiS_i
.headerflags  @"EF_CUDA_SM86 EF_CUDA_PTX_SM(EF_CU
DA_SM86)"
   /*0000*/   MOV R1, c[0x0][0x28] ;
   /*0010*/   S2R R4, SR_CTAID.X ;
   /*0020*/   MOV R0, c[0x0][0x170] ;
   /*0030*/   S2R R3, SR_TID.X ;
   /*0040*/   ISETP.GE.AND P0, PT, R0, 0x1, PT ;
   /*0050*/  @!P0 EXIT ;
   /*0060*/   IADD3 R2, R0.reuse, -0x1, RZ ;
   /*0070*/   ULDC.64 UR4, c[0x0][0x118] ;
   /*0080*/   LOP3.LUT R0, R0, 0x1ff, RZ, 0xc0, !PT ;
   /*0090*/   IMAD R4, R4, c[0x0][0x0], R3 ;
   /*00a0*/   ISETP.GE.U32.AND P1, PT, R2, 0x1ff, PT ;
   /*00b0*/   ISETP.NE.AND P0, PT, R0, RZ, PT ;
   /*00c0*/   MOV R5, RZ ;
   /*00d0*/  @!P1 BRA 0x6190 ;
   /*00e0*/   IADD3 R6, -R0, c[0x0][0x170], RZ ;
   /*00f0*/   MOV R5, RZ ;
   /*0100*/   MOV R8, 0x4 ;
   /*0110*/   IMAD R3, R4, c[0x0][0x170], R5 ;
   /*0120*/   IMAD.WIDE R2, R3, R8, c[0x0][0x160] ;
   /*0130*/   LDG.E R11, [R2.64] ;
   /*0140*/   MOV R7, 0x1 ;
   /*0150*/   IMAD.WIDE R10, R11, R8, c[0x0][0x168] ;
   /*0160*/   RED.E.ADD.STRONG.GPU [R10.64], R7 ;
   /*0170*/   LDG.E R13, [R2.64+0x4] ;
   /*0180*/   IMAD.WIDE R12, R13, R8, c[0x0][0x168] ;
   /*0190*/   RED.E.ADD.STRONG.GPU [R12.64], R7 ;
   /*01a0*/   LDG.E R15, [R2.64+0x8] ;
     ...       ...
   /*6100*/   IMAD.WIDE R14, R9, R8, c[0x0][0x168] ;
   /*6110*/   RED.E.ADD.STRONG.GPU [R14.64], R7 ;
   /*6120*/   LDG.E R9, [R2.64+0x7fc] ;
   /*6130*/   ISETP.NE.AND P1, PT, R6, RZ, PT ;
   /*6140*/   IADD3 R5, R5, 0x200, RZ ;
   /*6150*/   IMAD.WIDE R8, R9, R8, c[0x0][0x168] ;
   /*6160*/   RED.E.ADD.STRONG.GPU [R8.64], R7 ;
   /*6170*/  @!P1 CALL.REL.NOINC 0x6190 ;
     ...       ...
```

Ici le code du kernel a été coupé de l'adresse `0x01a0` *à* `0x6100` *ce qui représente* 23966 *octets, alors qu'à droite le code est complet.*

On remarque dans le code non déroulé de droite, un peu de déroulage par le compilateur pour optimiser les possibilités d'exploitation d'ILP, « Instruction Level Parallelism *» car on remarque la répétition des instrcutions* `IMAD`, `RED`, `LDG`.

Code du kernel « non déroulé *» :*

```
   Function : _Z9histogramPiS_i
.headerflags    @"EF_CUDA_SM86 EF_CUDA_PTX_SM(EF_CU
DA_SM86)"
/*0000*/ IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28] ;
/*0010*/ IMAD.MOV.U32 R5, RZ, RZ, c[0x0][0x170];
/*0020*/  S2R R12, SR_CTAID.X ;
/*0030*/  S2R R3, SR_TID.X ;
/*0040*/  ISETP.GE.AND P0, PT, R5, 0x1, PT ;
/*0050*/ @!P0 EXIT ;
/*0060*/  IADD3 R2, R5.reuse, -0x1, RZ ;
/*0070*/  ULDC.64 UR4, c[0x0][0x118] ;
/*0080*/  LOP3.LUT R0, R5, 0x3, RZ, 0xc0, !PT ;
/*0090*/  IMAD R12, R12, c[0x0][0x0], R3 ;
/*00a0*/  ISETP.GE.U32.AND P1, PT, R2, 0x3, PT ;
/*00b0*/  ISETP.NE.AND P0, PT, R0, RZ, PT ;
/*00c0*/  MOV R13, RZ ;
/*00d0*/ @!P1 BRA 0x280 ;
/*00e0*/  IMAD.MOV.U32 R15, RZ, RZ, 0x4 ;
/*00f0*/  IADD3 R14, -R0, c[0x0][0x170], RZ ;
/*0100*/  IMAD R2, R12, R5, 0x2 ;
/*0110*/  MOV R13, RZ ;
/*0120*/  IMAD.WIDE R2, R2, R15, c[0x0][0x160] ;
/*0130*/  IMAD.MOV.U32 R10, RZ, RZ, R2 ;
/*0140*/  MOV R11, R3 ;
/*0150*/  LDG.E R2, [R10.64+-0x8] ;
/*0160*/  IMAD.MOV.U32 R17, RZ, RZ, 0x1 ;
/*0170*/  IMAD.WIDE R2, R2, R15, c[0x0][0x168] ;
/*0180*/  RED.E.ADD.STRONG.GPU [R2.64], R17 ;
/*0190*/  LDG.E R4, [R10.64+-0x4] ;
/*01a0*/  IMAD.WIDE R4, R4, R15, c[0x0][0x168] ;
/*01b0*/  RED.E.ADD.STRONG.GPU [R4.64], R17 ;
/*01c0*/  LDG.E R6, [R10.64] ;
/*01d0*/  IADD3 R14, R14, -0x4, RZ ;
/*01e0*/  IMAD.WIDE R6, R6, R15, c[0x0][0x168] ;
/*01f0*/  RED.E.ADD.STRONG.GPU [R6.64], R17 ;
/*0200*/  LDG.E R8, [R10.64+0x4] ;
/*0210*/  ISETP.NE.AND P1, PT, R14, RZ, PT ;
/*0220*/  IADD3 R2, P2, R10, 0x10, RZ ;
/*0230*/  IADD3 R13, R13, 0x4, RZ ;
/*0240*/  IADD3.X R3, RZ, R11, RZ, P2, !PT ;
/*0250*/  IMAD.WIDE R8, R8, R15, c[0x0][0x168] ;
/*0260*/  RED.E.ADD.STRONG.GPU [R8.64], R17 ;
/*0270*/  @P1 BRA 0x130 ;
/*0280*/ @!P0 EXIT ;
/*0290*/  IMAD.MOV.U32 R7, RZ, RZ, 0x4 ;
/*02a0*/  IMAD R2, R12, R5, 0x2 ;
/*02b0*/  IMAD.WIDE R2, R2, R7, c[0x0][0x160] ;
/*02c0*/  IMAD.MOV.U32 R5, RZ, RZ, R3 ;
/*02d0*/  MOV R4, R2 ;
/*02e0*/  LDG.E R2, [R4.64] ;
/*02f0*/  IADD3 R0, R0, -0x1, RZ ;
/*0300*/  MOV R9, 0x1 ;
/*0310*/  ISETP.NE.AND P0, PT, R0, RZ, PT ;
/*0320*/  IADD3 R4, P1, R4, 0x4, RZ ;
/*0330*/  IMAD.X R5, RZ, RZ, R5, P1 ;
/*0340*/  IMAD.WIDE R2, R2, R7, c[0x0][0x168] ;
/*0350*/  RED.E.ADD.STRONG.GPU [R2.64], R9 ;
/*0360*/  @P0 BRA 0x2e0 ;
/*0370*/  EXIT ;
/*0380*/  BRA 0x380;
/*0390*/  NOP;
```

le branchement allant vers `0x130`.

Au niveau des performances, on ne note pas d'amélioration :

kernel	slice	nb opérations	temps
64 1024	512	$2^6 * 2^{10} * 2^9 = 2^{25}$	7.2ms
64 512	1024	$2^6 * 2^9 * 2^{10} = 2^{25}$	6.9ms
64 256	2048	$2^6 * 2^8 * 2^{11} = 2^{25}$	6.7ms

Résultats similaires aux précédents.

Pareil, si on déroule partiellement à la main :

```
__global__ void histogram_ilp( int *t, int *h, int
slice ) {
  int pos = ((blockIdx.x * blockDim.x)
                + threadIdx.x) * slice;
  for(int i = 0; i < slice; i+=4)
  {   atomicAdd(h+t[pos+i], 1);
      atomicAdd(h+t[pos+i+1], 1);
      atomicAdd(h+t[pos+i+2], 1);
      atomicAdd(h+t[pos+i+3], 1);
  }
}
```

kernel		slice	nb opérations	temps
128	512	512	$2^7 * 2^9 * 2^9 = 2^{25}$	6.9ms

3 – Écrivez une nouvelle version de votre kernel CUDA utilisant la **mémoire partagée** :

- ◇ les threads d'un même bloc utilisent un histogramme **local au bloc** ;
- ◇ à la fin les différents histogrammes sont **combinés** dans **l'histogramme final**.

```
 1 __global__ void histogramme_cache( int *t, int *h, int slice ) {
 2   __shared__ int histo[SAMPLE];
 3   int pos = ((blockIdx.x * blockDim.x) + threadIdx.x) * slice;
 4   int rang = threadIdx.x;
 5
 6   while(rang < SAMPLE)
 7   {
 8     histo[rang] = 0;
 9     rang += blockDim.x;
10   }
11   __syncthreads(); ❷
12   for (int i = 0; i < slice; i++)
13   {
14     atomicAdd(histo+t[pos+i], 1);
15   }
16   __syncthreads(); ❸
17
18   rang = threadIdx.x;
19
20   while(rang < SAMPLE)
21   {
22     atomicAdd(h + rang, histo[rang]);
23     rang += blockDim.x;
24   }
25 }
```

❶⇒ **initialiser** le tableau d'échantillon si le bloc contient moins de threads que sa taille ;
❷⇒ **garantir** que toutes les threads ont fini d'initialiser **l'histogramme local**.
❸⇒ **garantir** que toutes les threads ont fini de mettre à jour **l'histogramme local**.

kernel		slice	nb opérations	temps
2048	32	512	$2^{11} * 2^5 * 2^9 = 2^5$	1.3ms
2048	64	256	$2^{11} * 2^6 * 2^8 = 2^{25}$	1.3ms
2048	128	128	$2^{11} * 2^7 * 2^7 = 2^{25}$	1.1ms
4096	32	256	$2^{12} * 2^5 * 2^8 = 2^{25}$	0.9ms
4096	64	128	$2^{12} * 2^6 * 2^7 = 2^{25}$	1.0ms
8192	64	64	$2^{13} * 2^6 * 2^6 = 2^{25}$	0.9ms

Si on essaye de supprimer les « slices *», c-à-d que chaque thread ne traite qu'une case des échantillons :*

```
__global__ void histogram_sans_slice( int *t, int *h ) {
  int pos = ((blockIdx.x * blockDim.x) + threadIdx.x);
  atomicAdd(h+t[pos], 1);
}
```

Le kernel est tout petit...

kernel		nb opérations	temps
32768	1024	$2^{15} * 2^{10} = 2^{25}$	6.6ms
65536	512	$2^{16} * 2^9 = 2^{25}$	6.6ms
131072	256	$2^{17} * 2^8 = 2^{25}$	6.6ms
262144	128	$2^{18} * 2^7 = 2^{25}$	6.6ms
524288	64	$2^{19} * 2^6 = 2^{25}$	6.6ms
1048576	32	$2^{20} * 2^5 = 2^{25}$	6.6ms

Et les performances :

Seconde méthode

On veut utiliser la **seconde méthode** suivante :

▷ chaque thread est associé à une valeur possible des échantillons, c-à-d à une case de l'histogramme.

4 – a. Écrivez un kernel CUDA **simple** pour cette méthode et comparez les résultats à ceux donnés par la première méthode.

```
1  // 8192 entiers -> 32768 octets
2  // taille -> 8192
3
4  __global__ void histogramme_inverse(int *t, int *h, int taille) {
5    extern __shared__ int donnees_locales[];
6    __shared__ int histo[SAMPLE];
7    int pos = blockIdx.x*taille;
8    int rang = threadIdx.x;
9    histo[rang] = 0;
10
11   while(rang < taille)
12   {
13     donnees_locales[rang] = t[pos+rang];   ❶
14     rang += blockDim.x;❷
15   }
16
17   __syncthreads();❸
18
19   rang = threadIdx.x;
20   for (int i = 0; i < taille; i++)
21     if (rang == donnees_locales[i])   ❹
22       histo[rang]++;
23
24   atomicAdd(h + rang,  histo[rang]);❺
25 }
```

❶⇒ **initialiser** le tableau local de valeurs que chaque des 2^{10} va traiter ;
❷⇒ on se décale de `blocDim.x` qui est égale à `SAMPLE` ou 2^{10}.
❸⇒ **garantir** que toutes les threads ont fini de mettre à jour **les données locales** ;
❹⇒ chaque thread parcours le tableau local de données pour sa valeur d'échantillon ;
❺⇒ chaque thread met à jour sa case dans l'histogramme local, mais elle n'est pas toute seule à le faire car on lance 2^{16} blocs s'occupant chacun de 2^{13} cases de données.

Au niveau des performances :

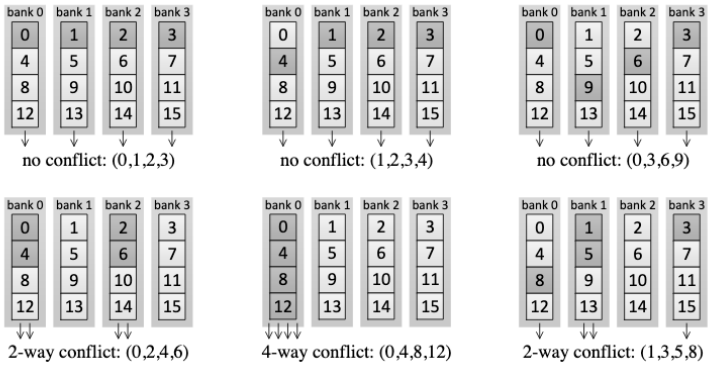
kernel		shared memory	nb opérations	temps
4096	1024	8192	$2^{12} * 2^{13} = 2^{25}$	121.4ms
8192	1024	4096	$2^{13} * 2^{12} = 2^{25}$	121.4ms
16384	1024	2048	$2^{14} * 2^{11} = 2^{25}$	121.2ms
32768	1024	1024	$2^{15} * 2^{10} = 2^{25}$	121.4ms

Ici, on a toujours 1024 threads par bloc, chacune de ces threads sert à reconnaitre « sa » valeur d'échantillon.

*Les performances sont **très mauvaises** par rapport à la première version !*
⟹ Il y a de la place pour de l'optimisation !

b. Comment **tirer parti au maximum** de l'organisation parallèle offerte par CUDA ?

On sait que l'accès à la mémoire partagée doit se faire en accord avec des « banks *» (accès par un réseau «* crossbar *»).*
Ces « banks *» sont au nombre de 32, pour des accès sur 32 bits (soit un entier par exemple) et le compilateur «* enroule *» les données entre les différentes banques :*



On sait aussi que toutes les threads peuvent demander simultanèment la même valeur sur une « bank *» ce qui donne lieu à un «* broadcast *», diffusion, de la valeur vers toutes les threads en ayant demander l'accès.*

c. Est-ce que l'utilisation de la **mémoire partagée** peut être intéressante ?

Oui, si on tient compte de ses contraintes pour son accès suivant les banques.

d. Écrivez une version de kernel **efficace** utilisant la **mémoire partagée** et comparez les résultats obtenus.

Il faut en tenir compte pour obtenir de bonnes, « meilleures *», performances :*

```
__global__ void histogram_banks( int *t, int *h ) {
  __shared__ int values[SAMPLE];
  /* int pos = (blockIdx.x * blockDim.x); */
  int pos = (blockIdx.x * SAMPLE);
  int v = threadIdx.x;
  int occurences = 0;
  int i = 0;

  values[v] = t[pos+v];
  __syncthreads();

  for(i = 0; i < SAMPLE; i++)
  {
    /* if (v == values[i]) occurences++; */
    occurences += (v == values[i]?1:0);
  }
  atomicAdd(h + v, occurences);
}
```

blockDim.x est forcèment égal à SAMPLE !

Petite amélioration : utiliser un opérateur ternaire
et un `xor`, car si deux valeurs sont identiques alors leur `xor` donne zéro

Et les performances ?

kernel		shared memory	nb opérations	temps
32768	1024	1024	$2^{15} * 2^{10} = 2^{25}$	21.6ms

On a accéléré par 6 nos performances ! On est passé de 120ms à 20ms !

On essayant de « dérouler *» :*

```
__global__ void histogram_banks2( int *t, int *h ) {
  __shared__ int values[SAMPLE];
  /* int pos = (blockIdx.x * blockDim.x); */
  int pos = (blockIdx.x * SAMPLE);
  int v1 = threadIdx.x;
  int v2 = threadIdx.x+SAMPLE/2;
  int occurences1 = 0;
  int occurences2 = 0;
  int i = 0;

  values[v1] = t[pos+v1];
  values[v2] = t[pos+v2];
  __syncthreads();

  for(i = 0; i < SAMPLE; i++)
  {
    /* occurences += (v == values[i]?1:0); */
    occurences1 += (v1 ^ values[i]?0:1);
    occurences2 += (v2 ^ values[i]?0:1);
  }
  atomicAdd(h + v1, occurences1);
  atomicAdd(h + v2, occurences2);
}
```

Les performances sont similaires ?

kernel		shared memory	nb opérations	temps
32768	512	1024	$2^{15} * 2^{10} * 2 = 2^{25}$	20.7ms

Et si on continue ?

```
__global__ void histogram_banks4( int *t, int *h ) {
  __shared__ int values[SAMPLE];
  /* int pos = (blockIdx.x * blockDim.x); */
  int pos = (blockIdx.x * SAMPLE);
  int v1 = threadIdx.x;
  int v2 = v1 + SAMPLE/4;
  int v3 = v2 + SAMPLE/4;
  int v4 = v3 + SAMPLE/4;
  int occurences1 = 0;
  int occurences2 = 0;
  int occurences3 = 0;
  int occurences4 = 0;
  int i = 0;

  values[v1] = t[pos+v1];
  values[v2] = t[pos+v2];
  values[v3] = t[pos+v3];
  values[v4] = t[pos+v4];
  __syncthreads();

  /* for(i = 0; i < SAMPLE; i++) */
  /* { */
  /*   if (v == values[i]) */
  /*     occurences++; */
  /* } */
  for(i = 0; i < SAMPLE; i++)
  {
    /* occurences += (v == values[i]?1:0); */
    occurences1 += (v1 ^ values[i]?0:1);
    occurences2 += (v2 ^ values[i]?0:1);
    occurences3 += (v3 ^ values[i]?0:1);
    occurences4 += (v4 ^ values[i]?0:1);
  }
  atomicAdd(h + v1, occurences1);
  atomicAdd(h + v2, occurences2);
  atomicAdd(h + v3, occurences3);
  atomicAdd(h + v4, occurences4);
}
```

Les performances sont encore similaires ?

kernel		shared memory	nb opérations	temps
32768	256	1024	$2^{15} * 2^{10} * 4 = 2^{25}$	20.5ms

Attention

Pour compiler dans la salle de TP211 avec les carte RTX 3060, il faut choisir l'architecture `sm_86`:

xterm

```
$ nvcc -arch=sm_80 \
-gencode=arch=compute_80,code=sm_80 \
-gencode=arch=compute_86,code=sm_86 \
-gencode=arch=compute_86,code=compute_86 \
-o compute_histo compute_histo.cu
```