CentraleSupélec

Mineure CalHau2

CUDA best practices

Stéphane Vialle

CentraleSupélec université PARIS-SACLAY ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC) RISEGrid Région lorraine Grand Est ALSACE CHAMPAGNE-ARDENNE LORRAINE

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

CUDA best pratices

- **Respect de la « coalescence »**
  - ***Thread* lisant 1 donnée sur tableau 1D**
  - *Thread* lisant 1 colonne sur tableau 2D
  - *Thread* lisant 1 ligne sur tableau 2D
  - *Thread* lisant *n* données sur tableau 1D
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

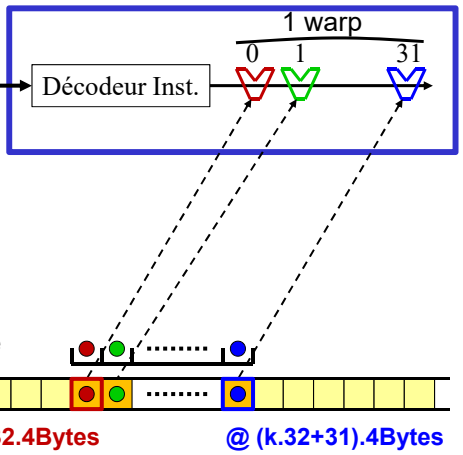
Respect de la coalescence

Thread lisant 1 donnée sur tableau 1D

Exemple 1D coalescent :

Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
float data = Tab[idx];
….
```



32 data accédées en parallèle
(1 chargeur)

Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes **@ (k.32+31).4Bytes**

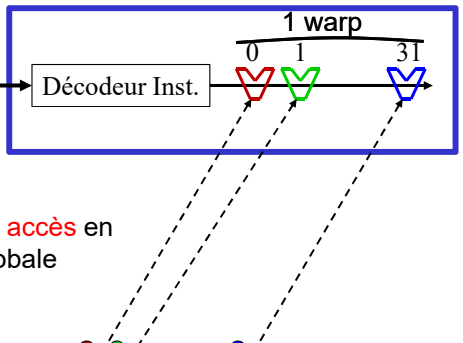
Respect de la coalescence

Thread lisant 1 donnée sur tableau 1D

Exemple 1D coalescent :

Code de chaque thread

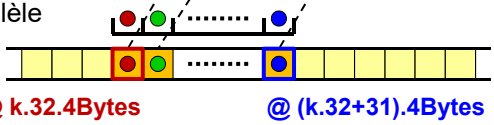
```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
float data = Tab[idx];
….
```



Coalescence parfaite !

Temps d'**un** accès en mémoire globale

32 data accédées en parallèle
(1 chargeur)



Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes **@ (k.32+31).4Bytes**

CentraleSupélec

CUDA best pratices

- **Respect de la « coalescence »**
  - *Thread* lisant 1 donnée sur tableau 1D
  - ***Thread* lisant 1 colonne sur tableau 2D**
  - *Thread* lisant 1 ligne sur tableau 2D
  - *Thread* lisant *n* données sur tableau 1D
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

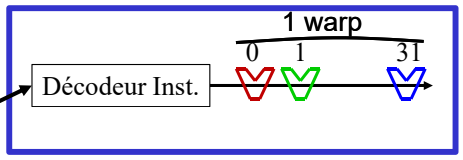
Thread lisant 1 colonne sur tableau 2D

Exemple 2D coalescent :

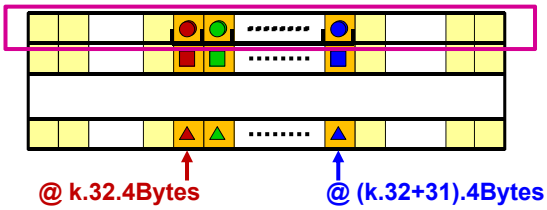
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[n][idx];
  ….
}
```

Chaque thread lit une colonne



n = 0 :



Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

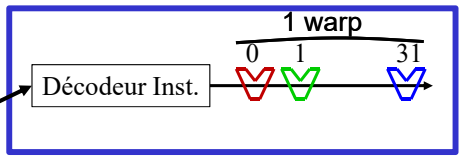
Respect de la coalescence

Thread lisant 1 colonne sur tableau 2D

Exemple 2D coalescent :

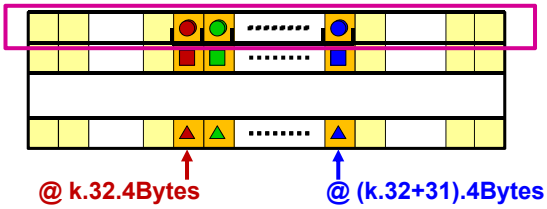
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[n][idx];
  ….
}
```



Chaque thread lit une colonne

n = 0 : 32 accès en parallèle



Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

Respect de la coalescence

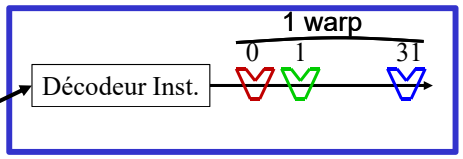
Thread lisant 1 colonne sur tableau 2D

Exemple 2D coalescent :

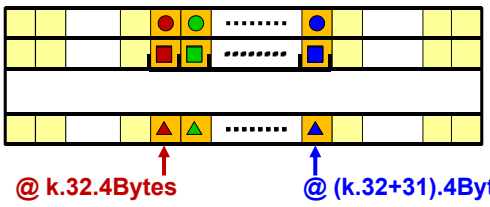
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[n][idx];
  ….
}
```

Chaque thread lit une colonne



n = 0 : 32 accès en parallèle
n = 1 :



Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

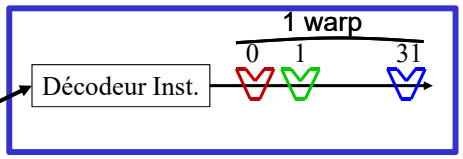
Thread lisant 1 colonne sur tableau 2D

Exemple 2D coalescent :

Code de chaque thread

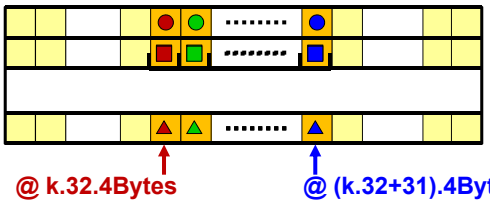
```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[n][idx];
  ....
}
```

Chaque thread lit une colonne



Coalescence parfaite !

n = 0 : 32 accès en parallèle
n = 1 : **32 accès en parallèle**



Conditions :
- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

CUDA best pratices

- **Respect de la « coalescence »**
  - *Thread* lisant 1 donnée sur tableau 1D
  - *Thread* lisant 1 colonne sur tableau 2D
  - ***Thread* lisant 1 ligne sur tableau 2D**
  - *Thread* lisant *n* données sur tableau 1D
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

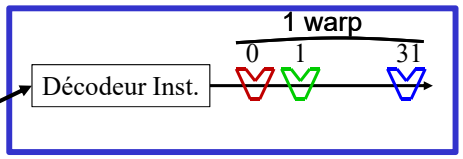
Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

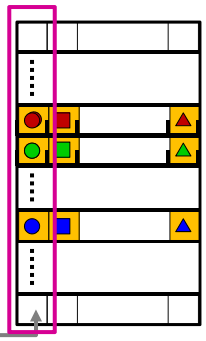
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```



Chaque thread lit une ligne

n = 0 : ....



Hypothèse (peu importante) :

- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes

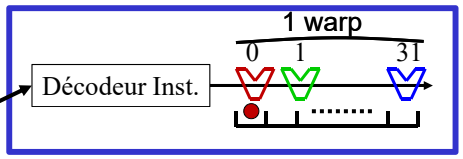
Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

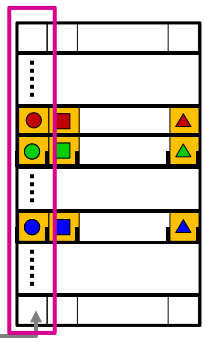
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```



Chaque thread lit une ligne

n = 0 : ....



Hypothèse (peu importante) :

- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes

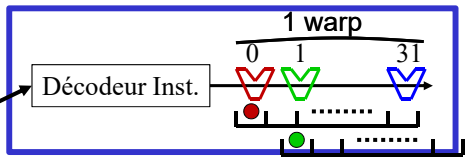
Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

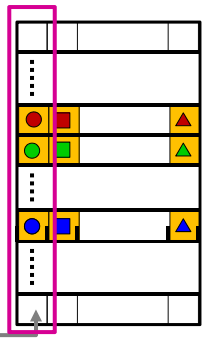
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```



Chaque thread lit une ligne

n = 0 : ....



Hypothèse (peu importante) :

- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes

Respect de la coalescence

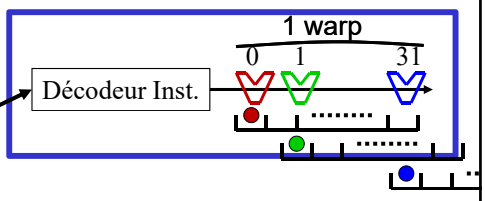
Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

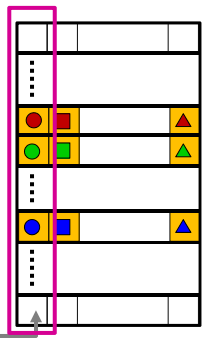
Code de chaque thread

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```

Chaque thread lit une ligne



n = 0 : 32 accès en séquentiel !!



Aucune coalescence !

Hypothèse (peu importante) :

- Démarrage à une adresse multiple de 32 mots mémoire

@ k.32.4Bytes

CentraleSupélec

CUDA best pratices

- **Respect de la « coalescence »**
  - *Thread* lisant 1 donnée sur tableau 1D
  - *Thread* lisant 1 colonne sur tableau 2D
  - *Thread* lisant 1 ligne sur tableau 2D
  - ***Thread* lisant *n* données sur tableau 1D**
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

CentraleSupélec

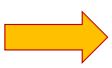
Thread lisant *n* données sur tableau 1D

Modèle d'exécution SIMT

Un warp suit un modèle d'exécution SIMD
- UN décodeur d'instruction
- Chaque thread hardware (ou ALU) fait la même chose que les autres en même temps, ou bien ne fait rien

Un bloc (de warps) de threads suit un modèle SIMT (NVIDIA)
- Les threads sont synchronisés par warps
- Les warps ne sont pas synchronisés
- Le bloc se termine quand tous les warps sont terminés

Parfois il est plus simple de raisonner sur le bloc (en SIMT) plutôt que sur les warps (SIMD) pour concevoir un code coalescent



Thread lisant *n* données sur tableau 1D

Kernel utilisant la mémoire globale et des registres

Une barre de threads par bloc, et une barre de blocs par grille (un choix)

Un thread réalise **n** calculs séparés en traitant **n** données.

Hyp : Nd = k*(BLOCK_SIZE_X*n)

```
Db = {BLOCK_SIZE_X,1,1}
Dg = {Nd/(BLOCK_SIZE_X*n),1,1}
```

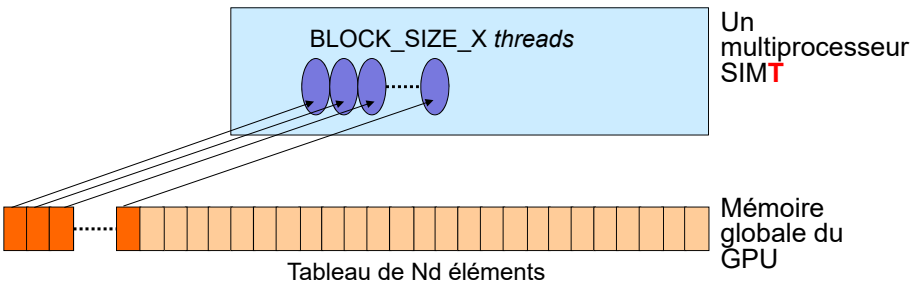
```
__global__ void f1(void)
{
  int offset = 0;                    // Registers
  float data = 0.0f, res = 0.0f;
  // Compute intitial data idx of the thread
  offset = threadIdx.x + blockIdx.x*(BLOCK_SIZE_X*n);
  // Loop with contiguous accesses to data tables
  for(int i = offset;
      i < offset + BLOCK_SIZE_X*n;
      i += BLOCK_SIZE_X) {
    // - Read one value from the global memory
    data = InGPU[i];
    // - Compute one result
    res = (data + 1.0f)*data …;
    // - Write one result in the global memory
    OutGPU[i] = res;
  }
}
```

Stratégie d'accès alignés aux données →

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :



Step 1:

BLOCK_SIZE_X accès contigus, et synchronisés par warp

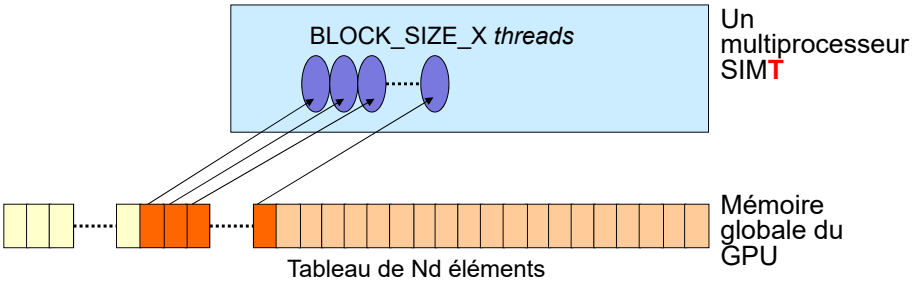
1 warp : modèle *Single Instruction Multiple Data*

1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :



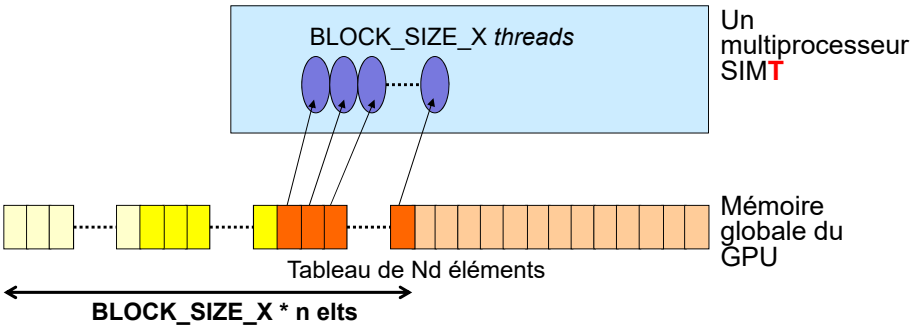
Step 2:

BLOCK_SIZE_X accès contigus, et synchronisés par warp
1 warp : modèle *Single Instruction Multiple Data*
1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :



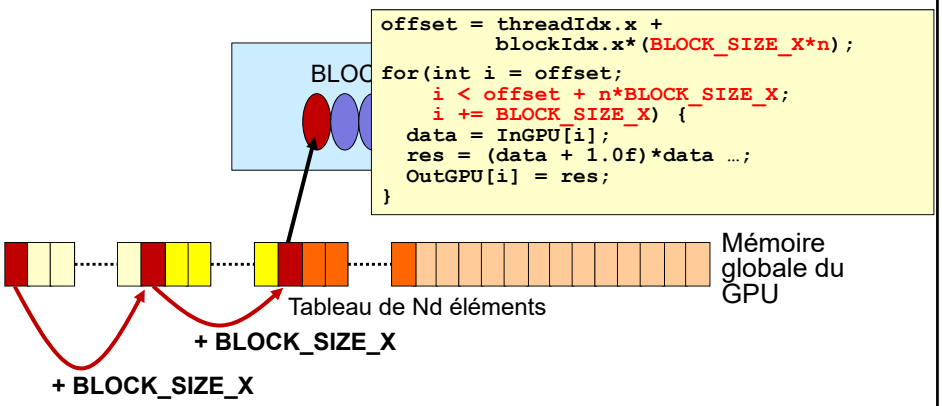
Step 3:

BLOCK_SIZE_X accès contigus, synchro par warp
1 warp : modèle *Single Instruction Multiple Data*
1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :

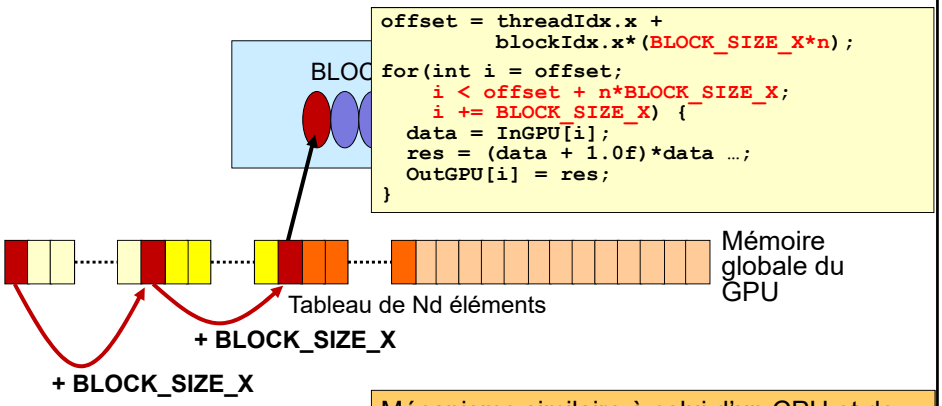


→ Pour que le bloc de threads synchronisés fasse des accès contigus en mémoire, il faut que chaque thread fasse des accès non contigus !!

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :

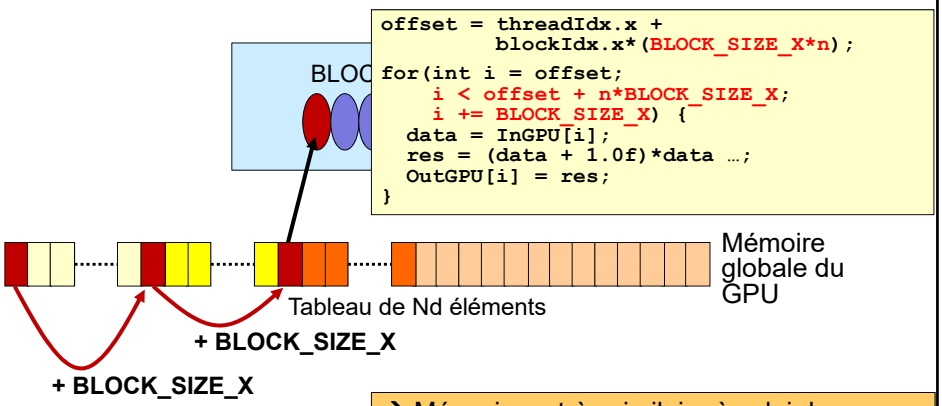


Mécanisme similaire à celui d'un CPU et de ses accès mémoires à travers son cache…
…**mais une boucle invisible est faite par des threads synchronisés**.

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :




```
offset = threadIdx.x +
         blockIdx.x*(BLOCK_SIZE_X*n);
for(int i = offset;
    i < offset + n*BLOCK_SIZE_X;
    i += BLOCK_SIZE_X) {
  data = InGPU[i];
  res = (data + 1.0f)*data …;
  OutGPU[i] = res;
}
```

→ Mécanisme très similaire à celui des **unités vectorielles (AVX)** d'un CPU

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Kernel utilisant la mémoire globale et des registres

Une barre de threads par bloc, et une barre de blocs par grille (un choix)

Un thread réalise **n** calculs séparés en traitant **n** données.

Hyp : Nd ≠ k*(BLOCK_SIZE_X*n)

```
Db = {BLOCK_SIZE_X,1,1}
if (Nd%(BLOCK_SIZE_X*n) == 0)
  Dg = {Nd/(BLOCK_SIZE_X*n),1,1}
else
  Dg = {Nd/(BLOCK_SIZE_X*n) + 1,1,1}
```

```
__global__ void f1(void)
{
  int offset = 0;
  float data = 0f, res = 0f;
  // Compute intitial data idx of the thread
  offset = threadIdx.x + blockIdx.x*(BLOCK_SIZE_X*n);
  // Loop with contiguous accesses to data tables
  for(int i = offset;
      i < offset + BLOCK_SIZE_X*n  &&  i < Nd;
      i += BLOCK_SIZE_X) {
    // - Read one value from the global memory
    data = InGPU[i];
    // - Compute one result
    res = (data + 1.0f)*data …;
    // - Write one result in the global memory
    OutGPU[i] = res;
  }
}
```

CentraleSupélec

CUDA best pratices

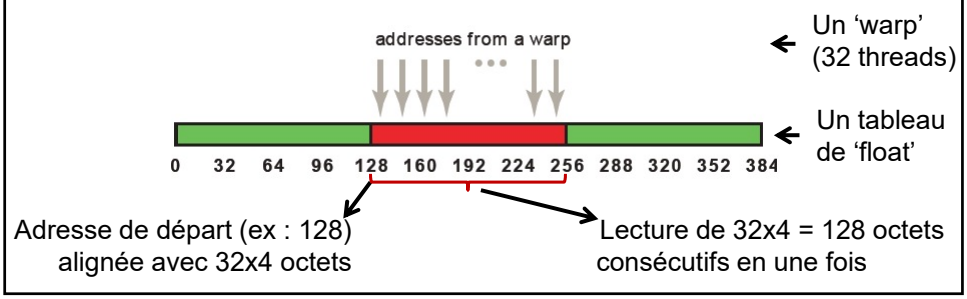
- **Respect de la « coalescence »**
  - *Thread* lisant 1 donnée sur tableau 1D
  - *Thread* lisant 1 colonne sur tableau 2D
  - *Thread* lisant 1 ligne sur tableau 2D
  - ***Thread* lisant *n* données sur tableau 1D**
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

Sensibilité de la coalescence

Principes complets de la *coalescence* :

- Les threads sont activés par « warp » de 32 consécutifs dans la dimension « x » de leur 3D-bloc
- Ils doivent accéder à des données consécutives en mémoire
- Le premier thread doit accéder à une donnée alignée avec un multiple de 32 mots mémoires de 4 octets.

→ Le warp récupère alors ses 32 données de 4 octets en « une fois »
→ Il récupère 128 octets en parfaite coalescence

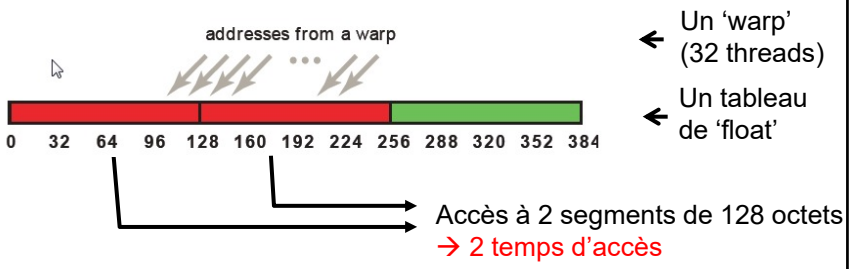


Respect de la coalescence

Sensibilité de la coalescence

Impact du « désalignement » :

- Hypothèse : les 32 threads d'un warp accèdent à 32 données consécutive, MAIS la première adresse n'est pas un multiple de 32x4o = 128o….
- Alors il y aura lecture de **DEUX** segments de 128 octets (au lieu d'un)



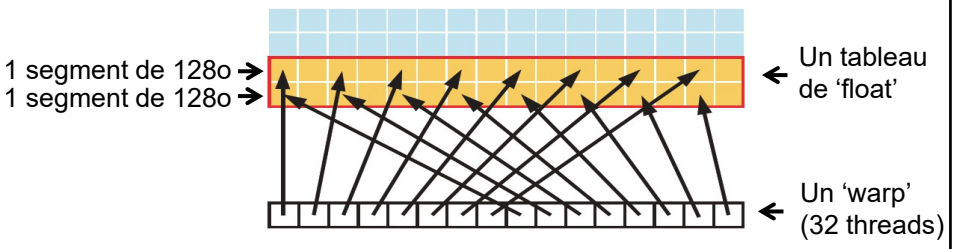
Respect de la coalescence

Sensibilité de la coalescence

Impact du « stride » (> 1) :

- Stride de 2 : le thread i accède à Tab[offset+ (2*i)]

  Rmq : on suppose le point de départ aligné (offset = k x 128)

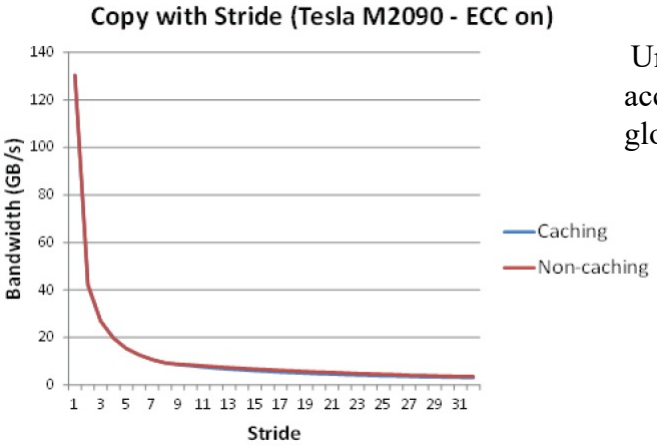


- On accède à 2 segments de 128 octets
  → 2 temps d'accès

Sensibilité de la coalescence

Impact du « stride » (> 1) :

- Dégradation de bande-passante applicative observée par NVIDIA



Un *stride* dans les accès à la mémoire globale se paie très vite très fort

CUDA best pratices

- **Respect de la « coalescence »**
  - *Thread* lisant 1 donnée sur tableau 1D
  - *Thread* lisant 1 colonne sur tableau 2D
  - *Thread* lisant 1 ligne sur tableau 2D
  - ***Thread* lisant *n* données sur tableau 1D**
  - Impact du désalignement et du *stride*
  - Règles de développement pour la coalescence
- Limitation de la « divergence »
- Conseils de développement

Respect de la coalescence

Règles de développement

Mise en place de la coalescence :

- La coalescence reste le premier « souci » du développement en CUDA
- Les caches améliorent les performances mais ne masquent pas les défauts de coalescences

→ **Il faut soigner la coalescence dès la conception de l'algorithme**

→ Il faut concevoir un ensemble **'stockage des données et accès'** qui vérifie la coalescence

Utiliser la 'shared memory' peut aider (voir plus loin), car on écrit un algorithme de cache dédié au problème.

Respect de la coalescence

Règles de développement

Vérifier la coalescence lors de la conception :

Démarche :

- **Considérer deux threads successifs en X dans un warp**
  (leur *threadIdx.x* sont des entiers successifs)
- **Identifier leurs accès en mémoire globale**

Analyse :

- Si le second accède à la case suivante du premier lors de chaque accès, alors : très bonne coalescence
- Si les deux accèdent à la même case : acceptable
  (gaspillage de Bw mémoire, mais un seul accès mémoire)
- Si les 32 threads du warp accèdent globalement à des données situées dans le même segment de 32 mots mémoire : bonne coalescence (supportée par les GPU modernes)
- Sinon … **revoir les structures de données et les codes de calculs** pour améliorer la coalescence.

CentraleSupélec

CUDA best pratices

- Respect de la « coalescence »
- **Limitation de la « divergence »**
- Conseils de développement

Exécution d'un « if…then…else »

Les divergences sont sources de ralentissement sur les archi. SIMD :

Ex : `if (x < 10) then {….} else {…}`

Exécution dans un warp :

1. Tous les threads testent la condition `(x < 10)`
2. Tous les threads qui doivent exécuter le *then* le font en parallèle
3. Tous les threads qui doivent exécuter le *else* le font en parallèle

Temps d'exécution :

- Si tous les threads exécutent le *then* :
  T(condition) + T(then)
- Si tous les threads exécutent le *else* :
  T(condition) + T(else)
- Sinon au moins un *then* et au moins un *else* sont exécutés :
  T(condition) + T(then) + T(else)

Limitation de la divergence

Exécution d'un « if…then…else »

Divergence très couteuse :

```
if (threadIdx.x % 2 == 0)
  {…}
else
  {…}
```

Chaque warp exécutera le *then* puis le *else*

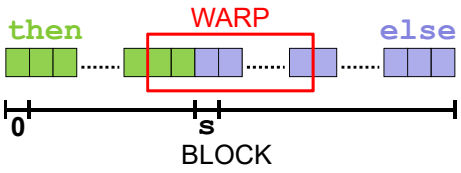
→ exécution lente sur tout le bloc !

Divergence moins couteuse :

```
if (threadIdx.x < s)
  {…}
else
  {…}
```

→ exécution lente dans un seul warp

En 1D un seul warp exécutera le *then* puis le *else* :



Limitation de la divergence

Exécution d'un « if…then…else »

Divergence très couteuse :

```
if (threadIdx.x % 2 == 0)
  {…}
else
  {…}
```

Chaque warp exécutera le *then* puis le *else*

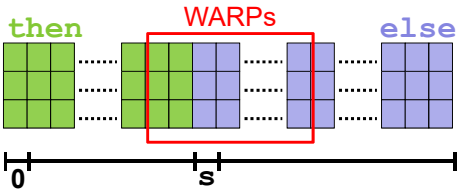
→ exécution lente sur tout le bloc !

Divergence moins couteuse :

```
if (threadIdx.x < s)
  {…}
else
  {…}
```

→ exécution lente dans une seule colonne de warps

En 2D une seule colonne de warps exécutera le *then* puis le *else* :



CentraleSupélec

CUDA best pratices

- Respect de la « coalescence »
- Limitation de la « divergence »
- **Conseils de développement**

CentraleSupélec

Conseils de développement

Développer et tester *step by step* !

- Vérifiez que chaque transfert de données CPU-GPU retourne bien CUDA_SUCCESS
- Développez, testez et validez successivement chaque kernel.
- Comparez les résultats d'un nouveau kernel GPU :
  - avec les résultats d'un kernel CPU

  ou :
  - avec les résultats d'un kernel GPU déjà validé
- Utilisez un profiler pour analyser la performance de chaque kernel GPU

Surcoût possible de développement:

T_{dev} sur GPU = T_{dev} kernels GPU + T_{dev} kernels de comparaison

Conseils de développement

Certaines erreurs de calcul ne sont pas des bugs :

- Deux codes parallèles différents ne font pas toujours les calculs dans le même ordre, et leurs arrondis peuvent être différents
- Les GPU ne font pas les arrondis des opérations « fma » comme les CPU !
- Les calculs en simple précision (`float`) sur CPU et GPU divergent rapidement !

Erreur d'arrondi ou Bug dans le code ?

Parfois difficile à différencier !

Conseils de développement

Compromis volontaire précision/vitesse de calcul

- Positionner certains flags de compilation pour profiter de la vitesse de traitement avec une précision limitée :

  Compilation avec : `--ftz=true` `--prec-sqrt=false` `--prec-div=false` `--fmad=true`

→ Recherche fine des limitations de précision possible sans grosse perte de précision

- Forcer l'utilisation de la bibliothèque « fast math » pour accélérer les calculs si la précision n'est pas critique :

  Compilation avec `--use_fast_math`

→ Réduction globale de la précision des calculs

CentraleSupélec

CUDA best practices

End