Calcul GPU – Cours 2: Mémoire et optimisation de performance

Jonathan Rouzaud-Cornabas

LIRIS / Insa de Lyon – Inria Beagle

Cours inspiré de ceux de Prof Wen-mei Hwu
(University of Illinois at Urbana–Champaign)

Références

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition

- NVIDIA, NVidia CUDA C Programming Guide, version 8.0, NVidia, 2016 (reference book)

Retour sur le noyau de floutage

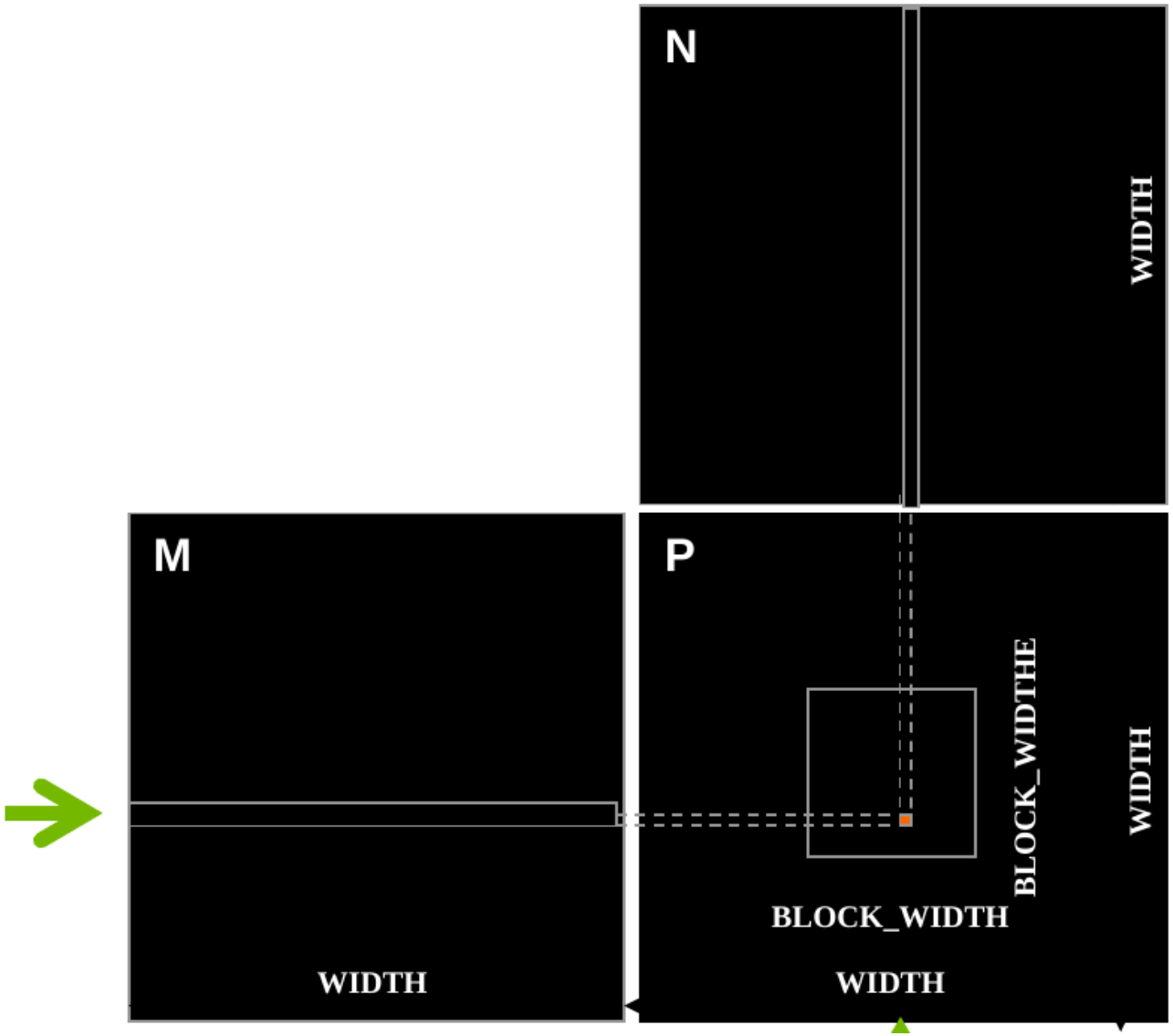
Lecture en mémoire : in[curRow * w + curCol]

```
int Col = blockIdx.x * blockDim.x + threadIdx.x;
int Row = blockIdx.y * blockDim.y + threadIdx.y;
if (Col < w && Row < h) {
  int pixVal = 0;
  int pixels = 0;
  for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
      int curRow = Row + blurRow;
      int curCol = Col + blurCol;
      if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
        pixVal += in[curRow * w + curCol];
        pixels++;
      }
    }
  }
  out[Row * w + Col] = (unsigned char)(pixVal / pixels);
}
```

Retour sur les performances d'un GPU

- Tous les threads accédent à la mémoire globale pour lire les éléments de matrice en entrée
  - Un accés mémoire (4 bytes) par addition de flottant
  - 4B/s de bande passante mémoire par FLOPS
- En supposant un GPU avec
  - Un pic de calcul flottant (FLOPS) de 1,500 GFLOPS avec une bande passante de DRAM de 200GB/s
  - $4 * 1,500 = 6,000$ GB/s nécessaire pour atteindre le pic de FLOPS
  - La bande passante mémoire de 200GB/s limite donc l'éxecution à 50 GFLOPS !
- Cette limite permet donc d'atteindre uniquement un pic d'exécution à 3.3% (50/1500) de la capacité théorique de la carte !
- Il est donc nécessaire de couper drastiquement la quantité d'accès mémoire pour se rapprocher de la vitesse théorique de 1,500 GFLOPS !

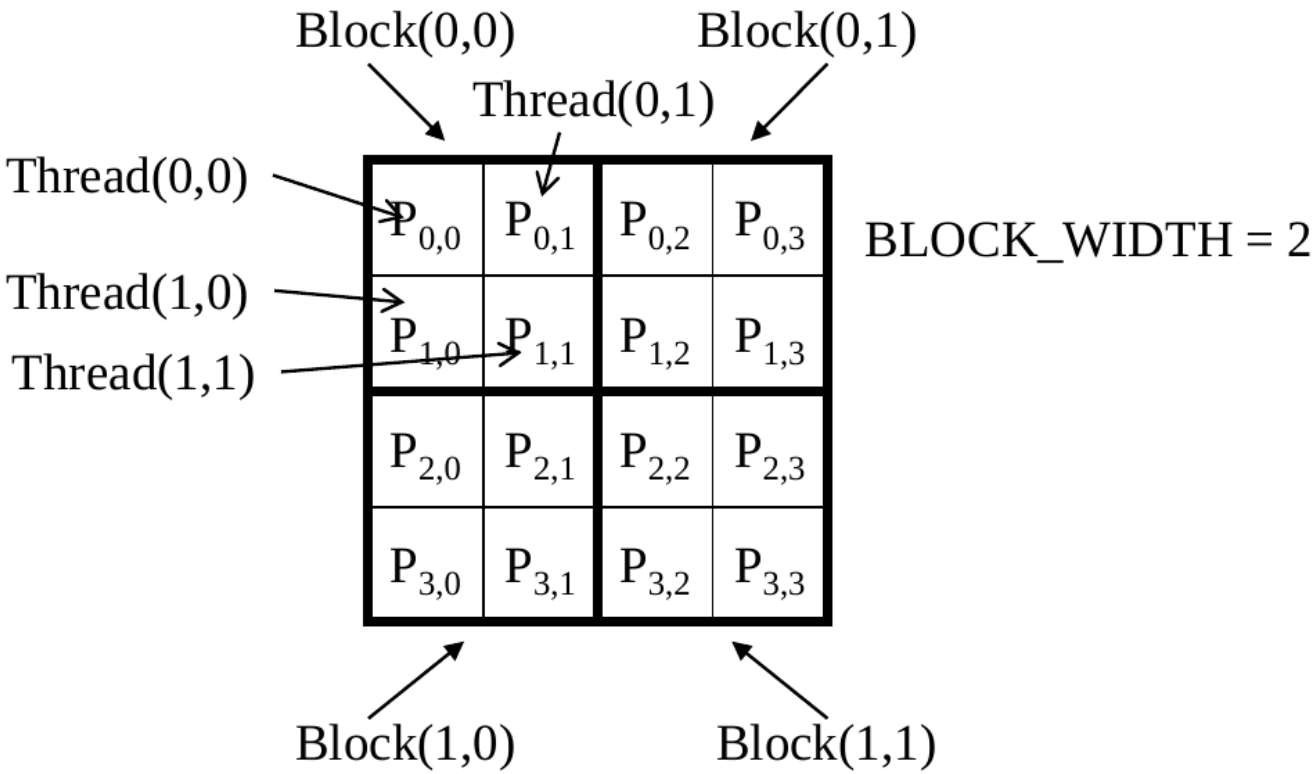
Passage à l'échelle transparent



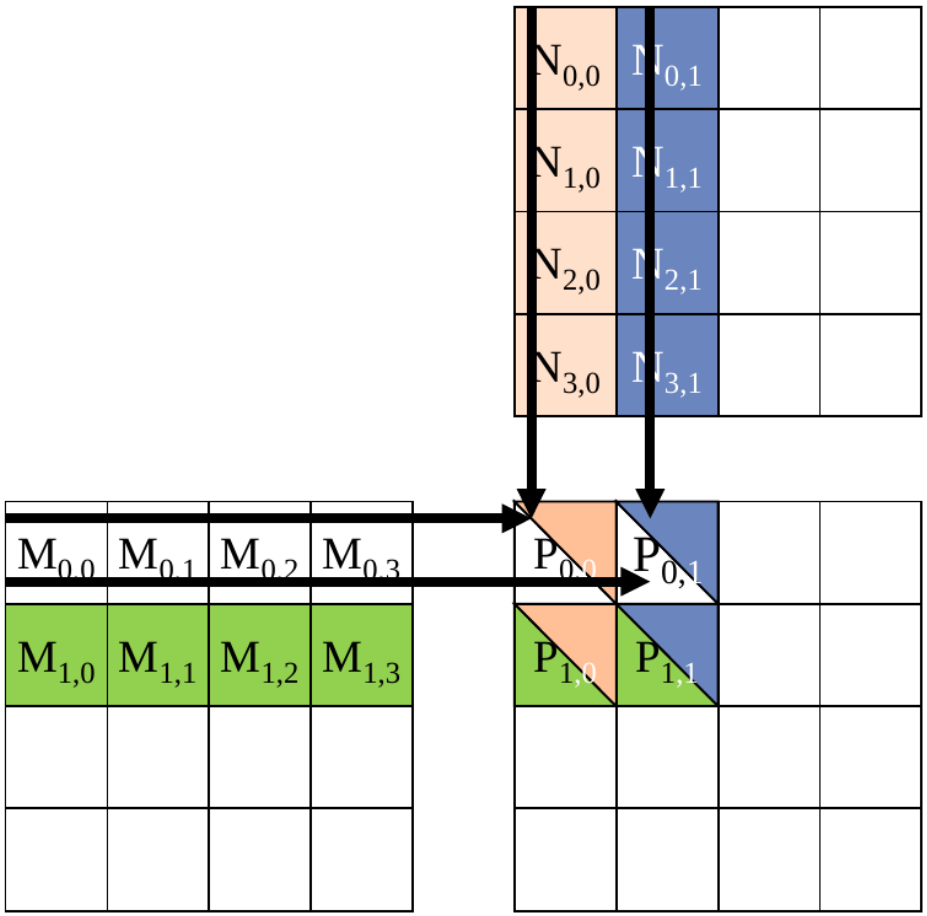
Multiplication de matrices : version simple

```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }
}
```

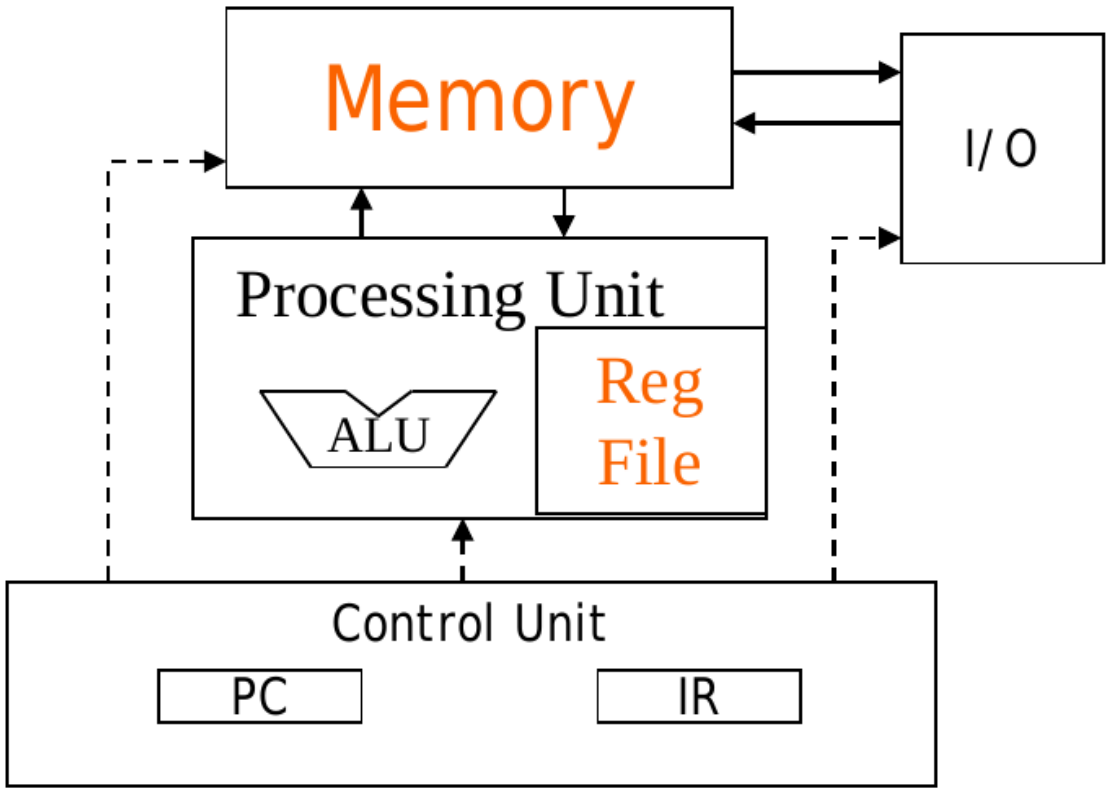
Multiplication de matrices : Répartition threads et données



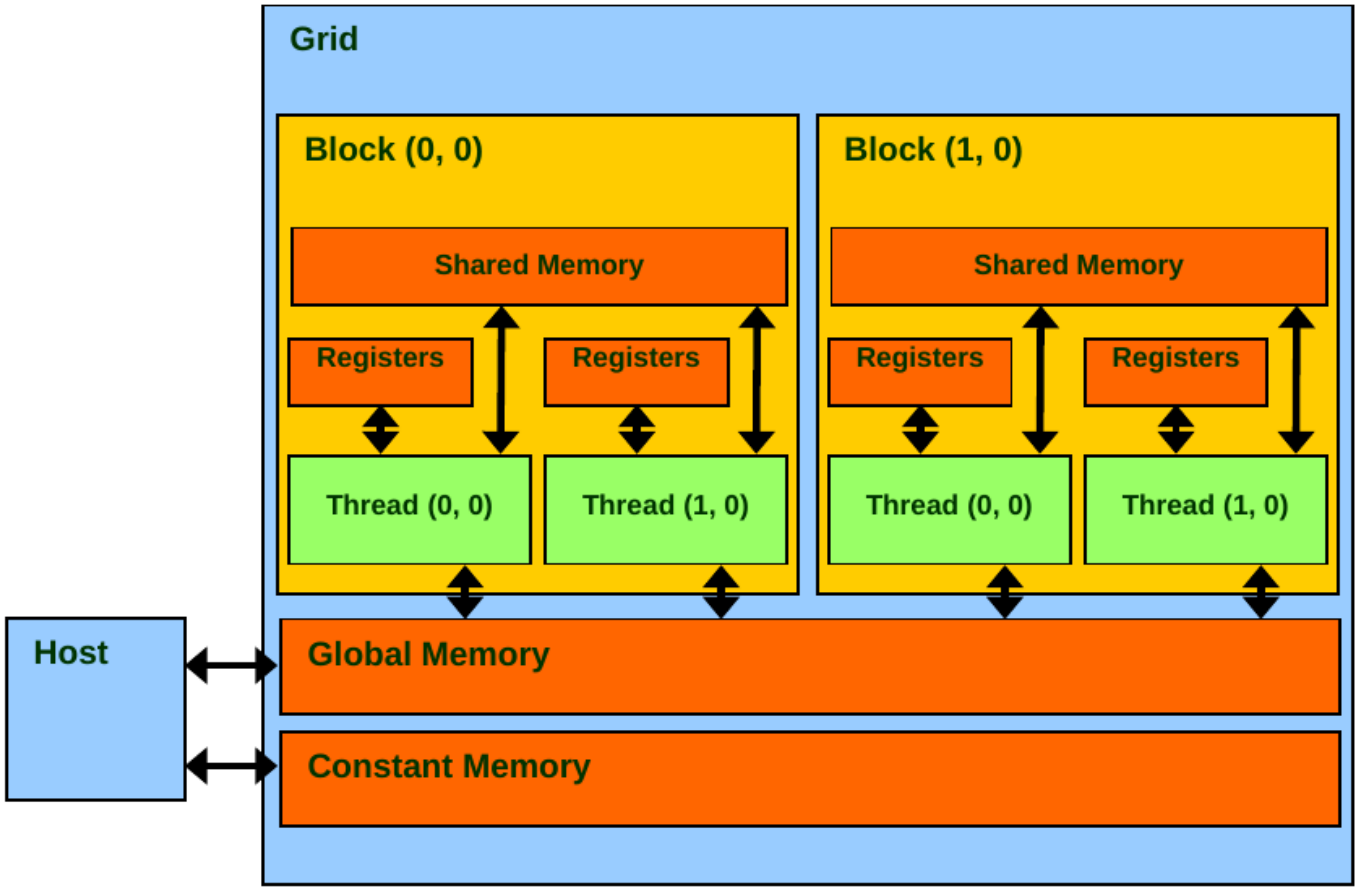
Multiplication de matrices : Calcul de $P_{0,0}$ et $P_{0,1}$



Modèle de Von-Neumann avec les registres et la mémoire



Les mémoires CUDA vues par le programmeur



Déclarer les variables en CUDA

Déclaration de variable	Mémoires	Scope	Durée de vie
int LocalVar;	registre	thread	thread
__device__ __shared__ int SharedVar;	shared	bloc	bloc
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ est optionel quand utilisé avec __shared__ ou __constant__
- Par défaut, les variables sont affectées dans des registres
- Sauf pour les tableaux internes aux threads qui sont affectées à la mémoire globale

Multiplication de matrices : Déclarer une variable partagée

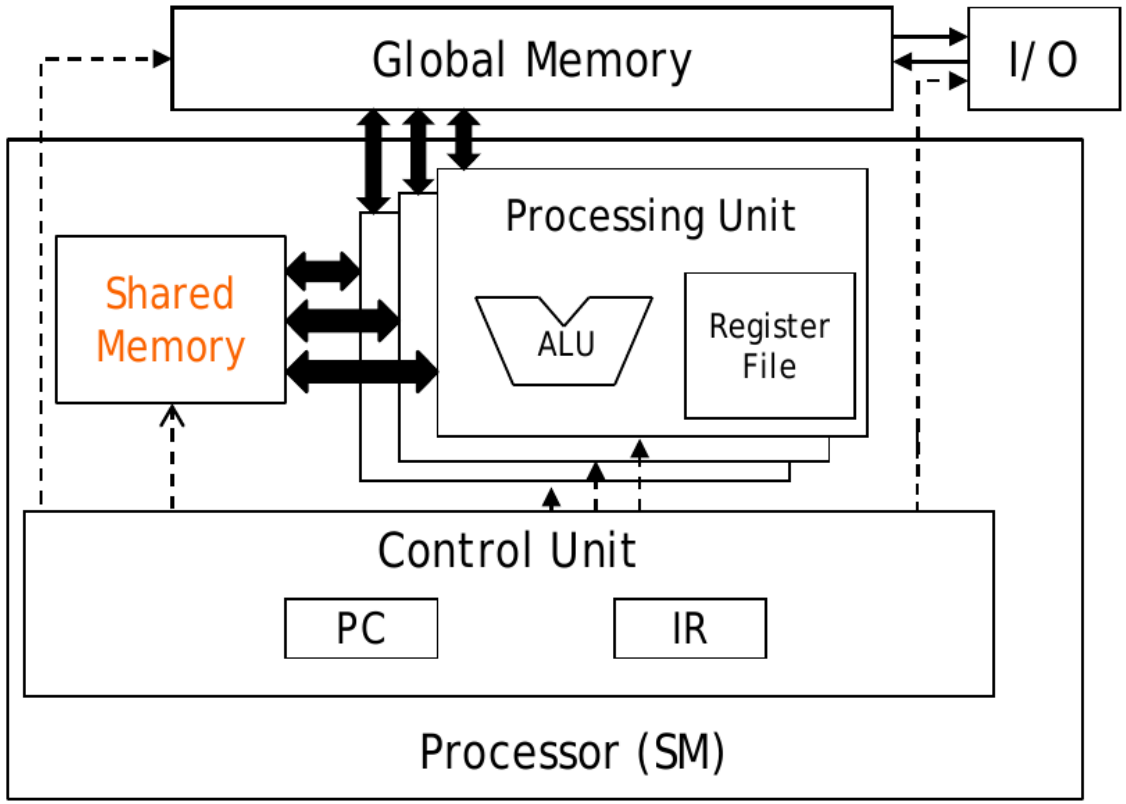
```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width) {
  __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];
  ...
}
```

La mémoire partagée en CUDA

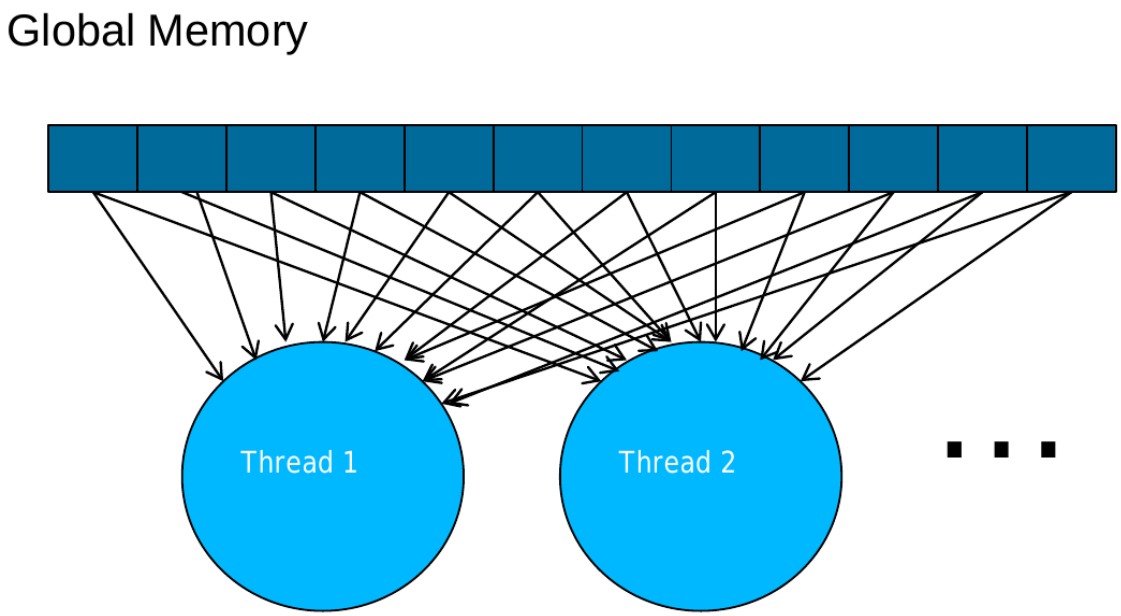
Un type spécial de mémoires dont le contenu est explicitement défini et utilisé dans le code source du noyau

- Une spécifique à chaque SM
- Les accés sont largement plus rapide (latence et bande passante) que la mémoire globale
- Les accés et les partages à cette mémoire sont limités aux threads d'un même bloc
- Le contenu disparait quand le dernier thread du bloc finit de s'exécuter
- Une forme de cache qui est géré par le programmeur

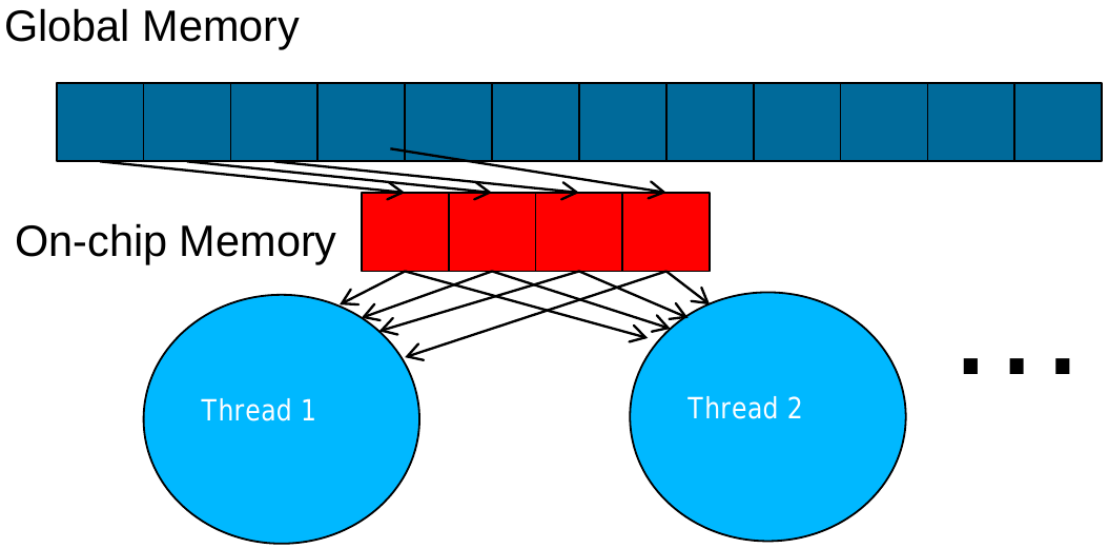
Modèle de Von-Neumann SIMD avec les mémoires CUDA



Accés à la mémoire globale pour le noyau multiplication de matrices

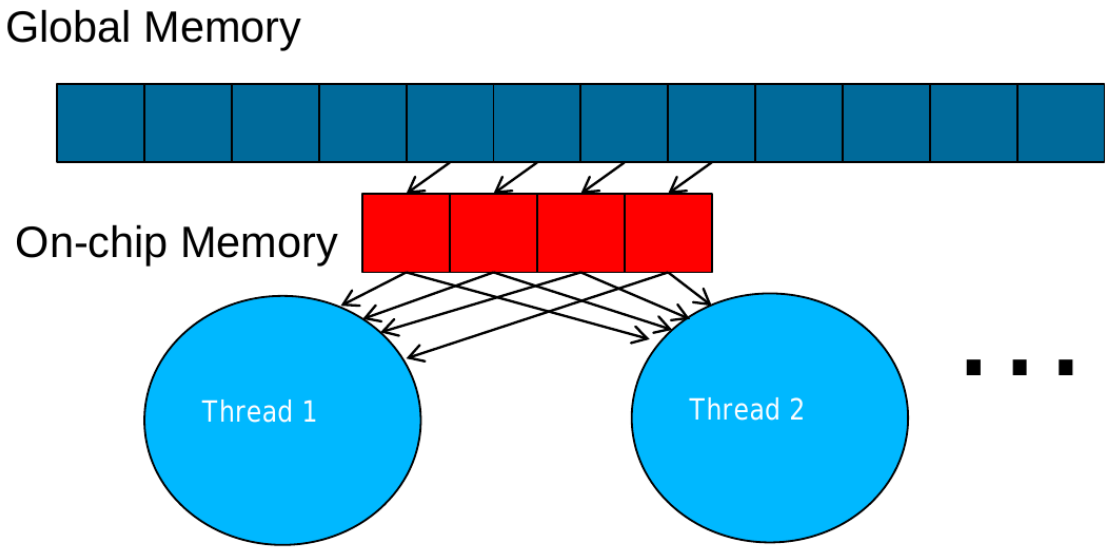


Tiling/Blocking : L'idée de base



- Diviser le contenu de la mémoire globale en tiles (carreau)
- Limiter les threads à calculer le contenu d'un (ou un petit nombre) de carreau à chaque moment dans le temps

Tiling/Blocking : L'idée de base



- Diviser le contenu de la mémoire globale en tiles (carreau)
- Limiter les threads à calculer le contenu d'un (ou un petit nombre) de carreau à chaque moment dans le temps

Le concept de base du tiling

Dans un système de traffic congestionné, une réduction significative du nombre de véhicules peut grandement améliorer le délai vu par tous les véhicules

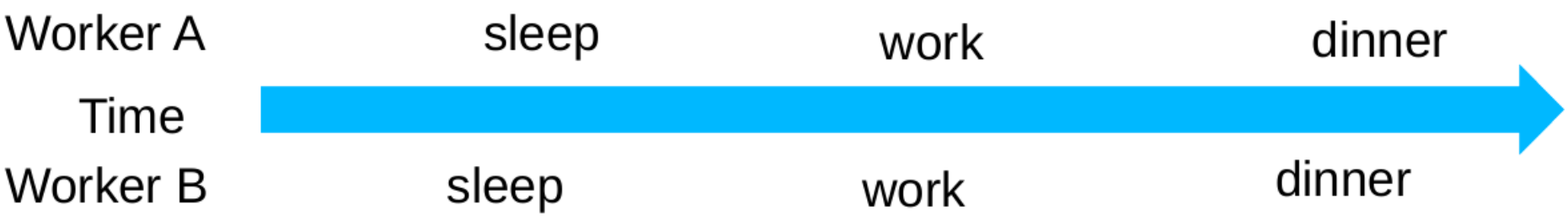
- Co-voiturage
- Tiling pour les accés à la mémoire globale
  - Conducteurs = threads accédant à leurs données en mémoire
  - Voitures = les demandes d'accès mémoire

Pas forcément facile d'utiliser le tiling

- Certains co-voiturages peuvent être plus simple à mettre en place que d'autres
  - Les participants doivent avoir des emplois du temps relativement similaires
  - Certains véhicules sont plus adaptés au co-voiturage
- Même raison et même difficulté avec le tiling

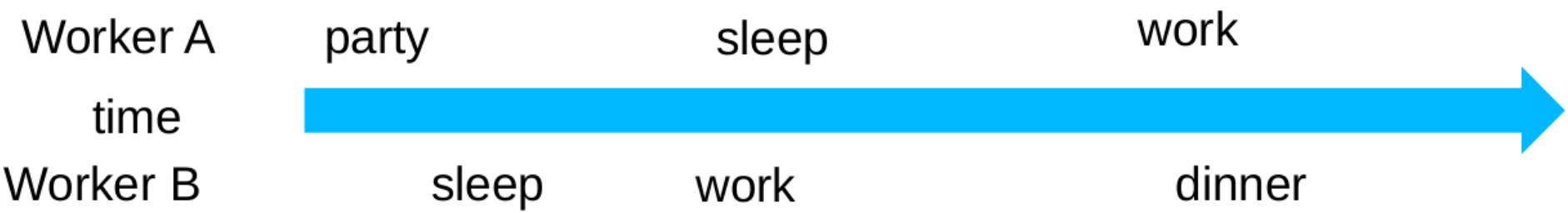
Le co-voiturage nécessite de la synchronisation

Tout se passe **BIEN** quand les personnes ont un emploi du temps similaire



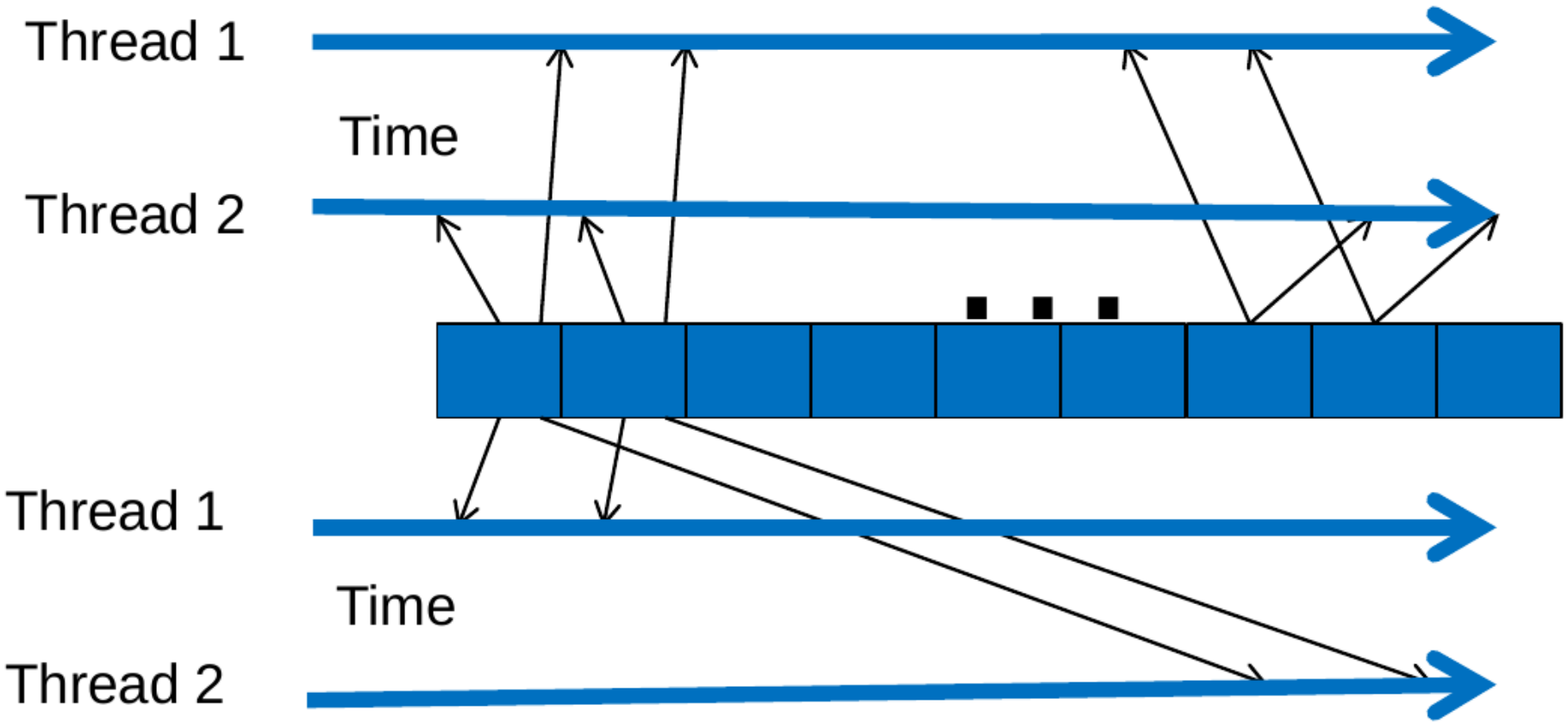
Le co-voiturage nécessite de la synchronisation

Tout se passe **MAL** quand les personnes ont un emploi du temps différent



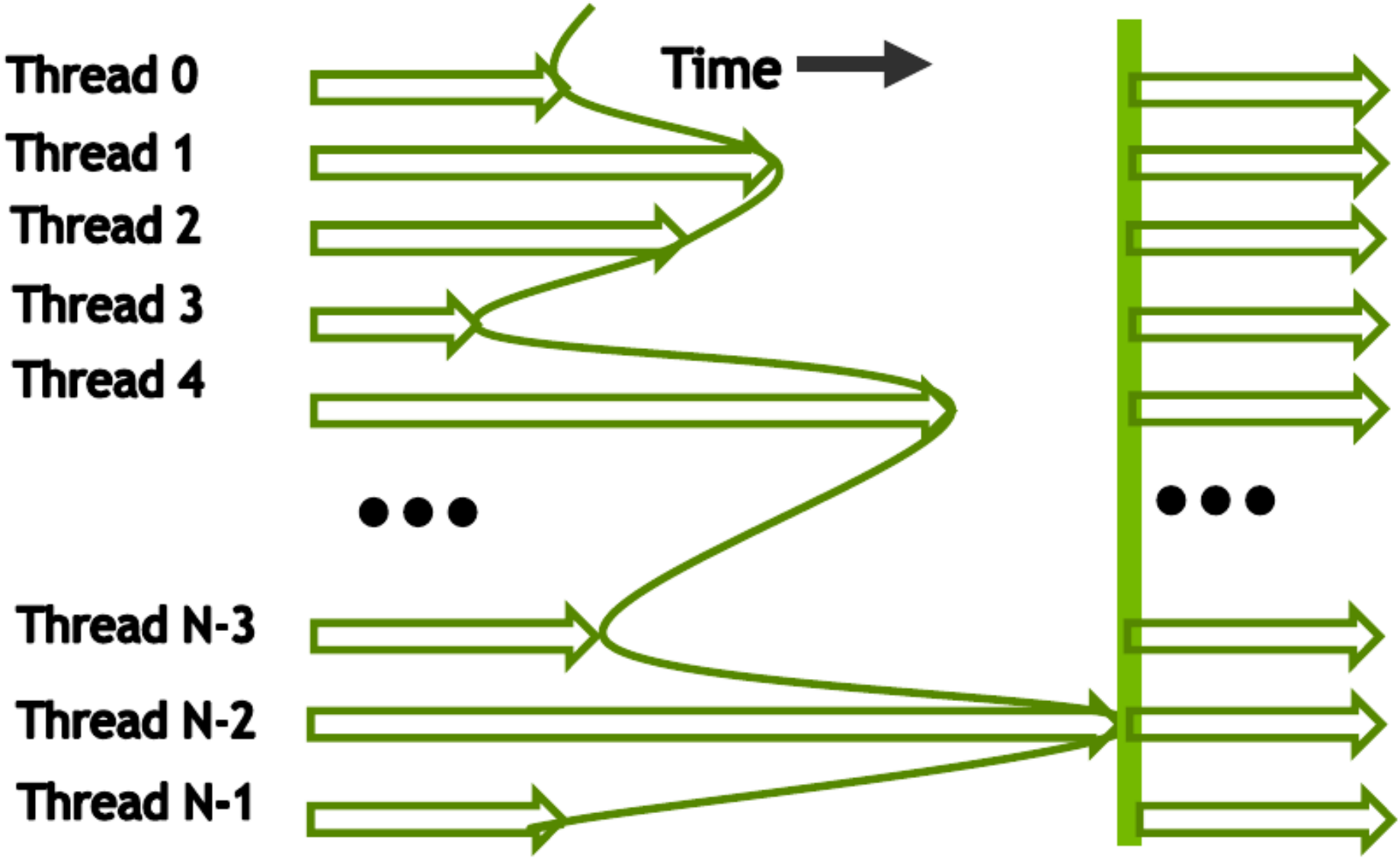
Même chose pour le tiling

Tout se passe **BIEN** quand les threads ont les mêmes chronologies d'accès



Tout se passe **MAL** quand les threads ont des chronologies différentes

Les barrières de synchronisation pour le tiling



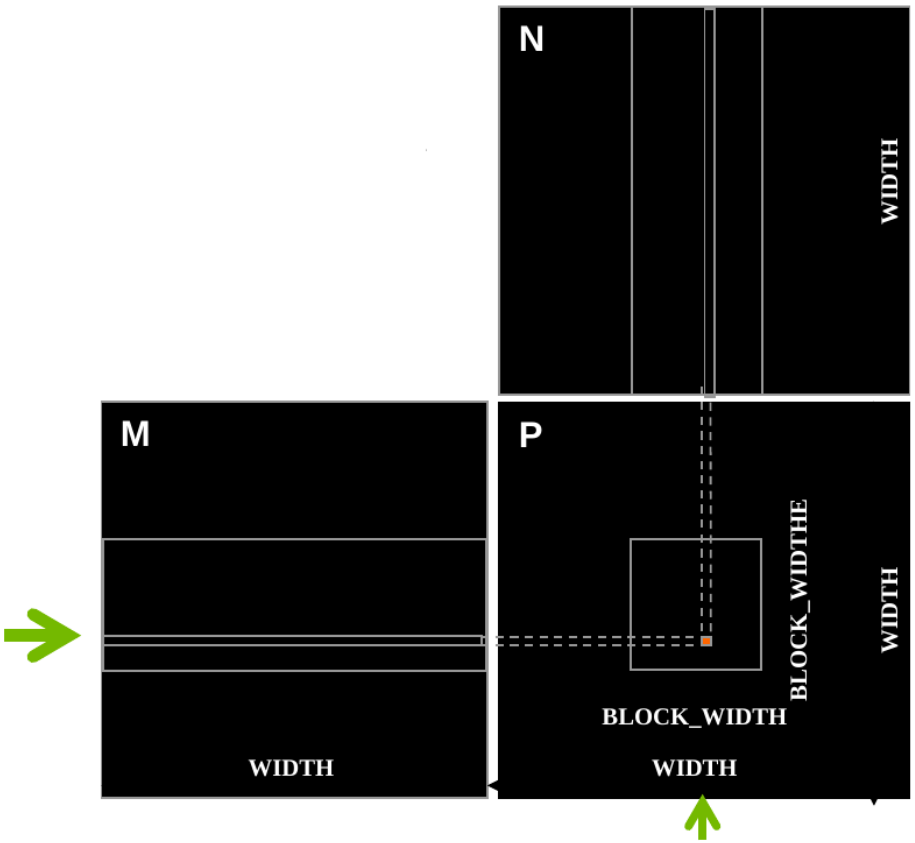
Profil de la technique de tiling

- Identifier un tile (un carreau, un bloc) de contenu de la mémoire globale qui est accédé par plusieurs threads
- Charger ce tile depuis la mémoire globale vers la mémoire on-chip
- Utiliser une barrière de synchronisation pour être sur que tous les threads sont prêts à démarrer l'étape
- Exécuter les multiples threads qui accédent aux données dans la mémoire on-chip
- Utiliser une barrière de synchronisation pour être sûr que tous les threads ont fini l'étape courante
- Avancer jusqu'au prochain tile

Multiplication de matrices

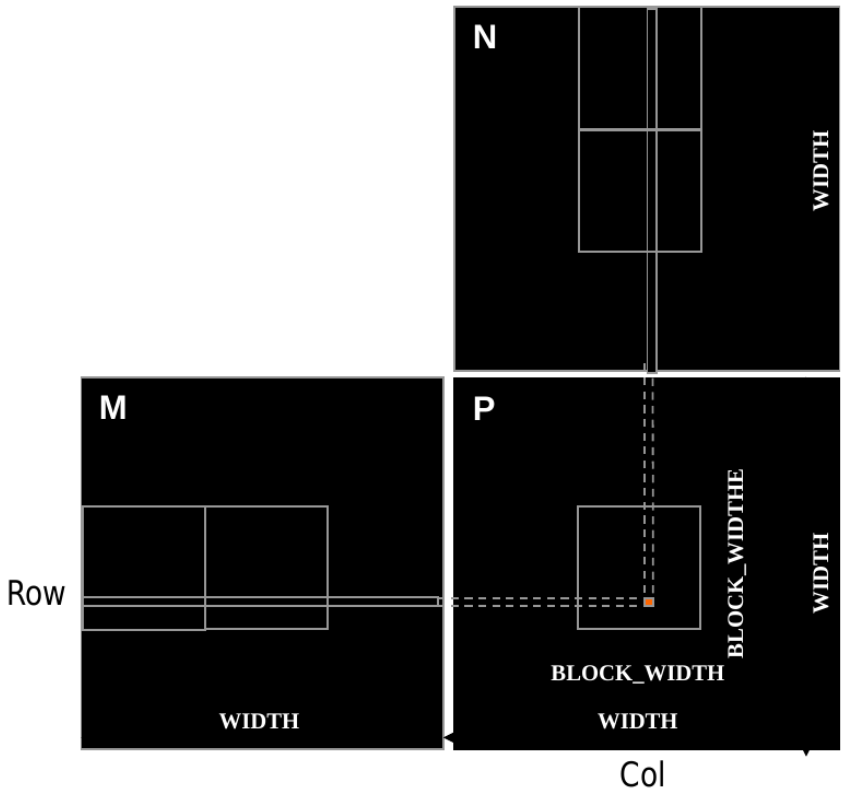
Motif d'accès aux données

- Chaque thread accéde à une ligne M et une colonne N
- Chaque bloc accéde à un ensemble de ligne et un ensemble de colonne



Multiplication de matrices tilisé

- Découper l'exécution de chaque thread en étape
- Pour que les accés aux données par le bloc de thread à chaque étape soient focalisés sur un carreau de M et un de N
- Le tile est de taille $BLOCK\_SIZE$ éléments dans chaque dimension

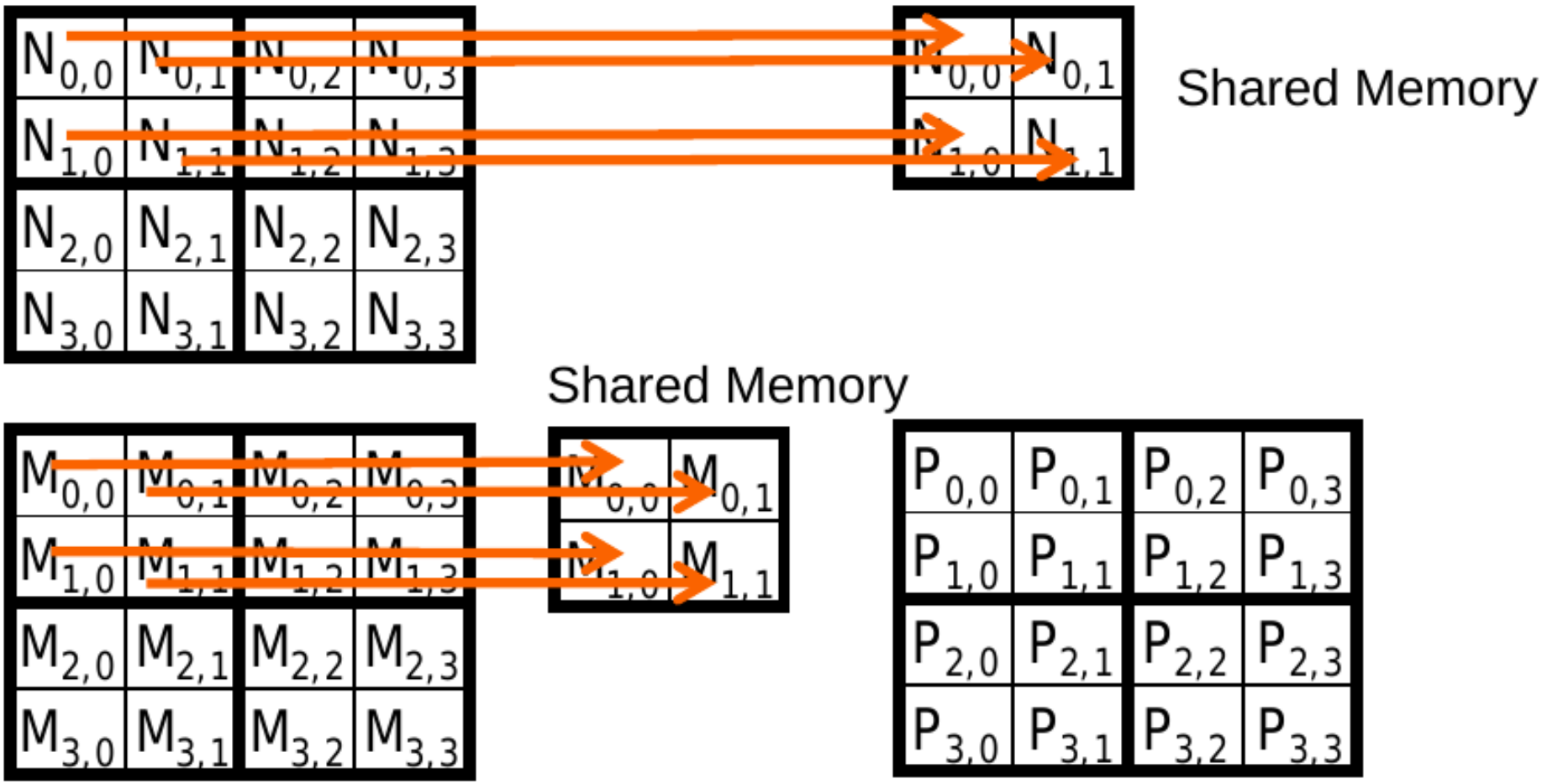


Chargement d'un carreau

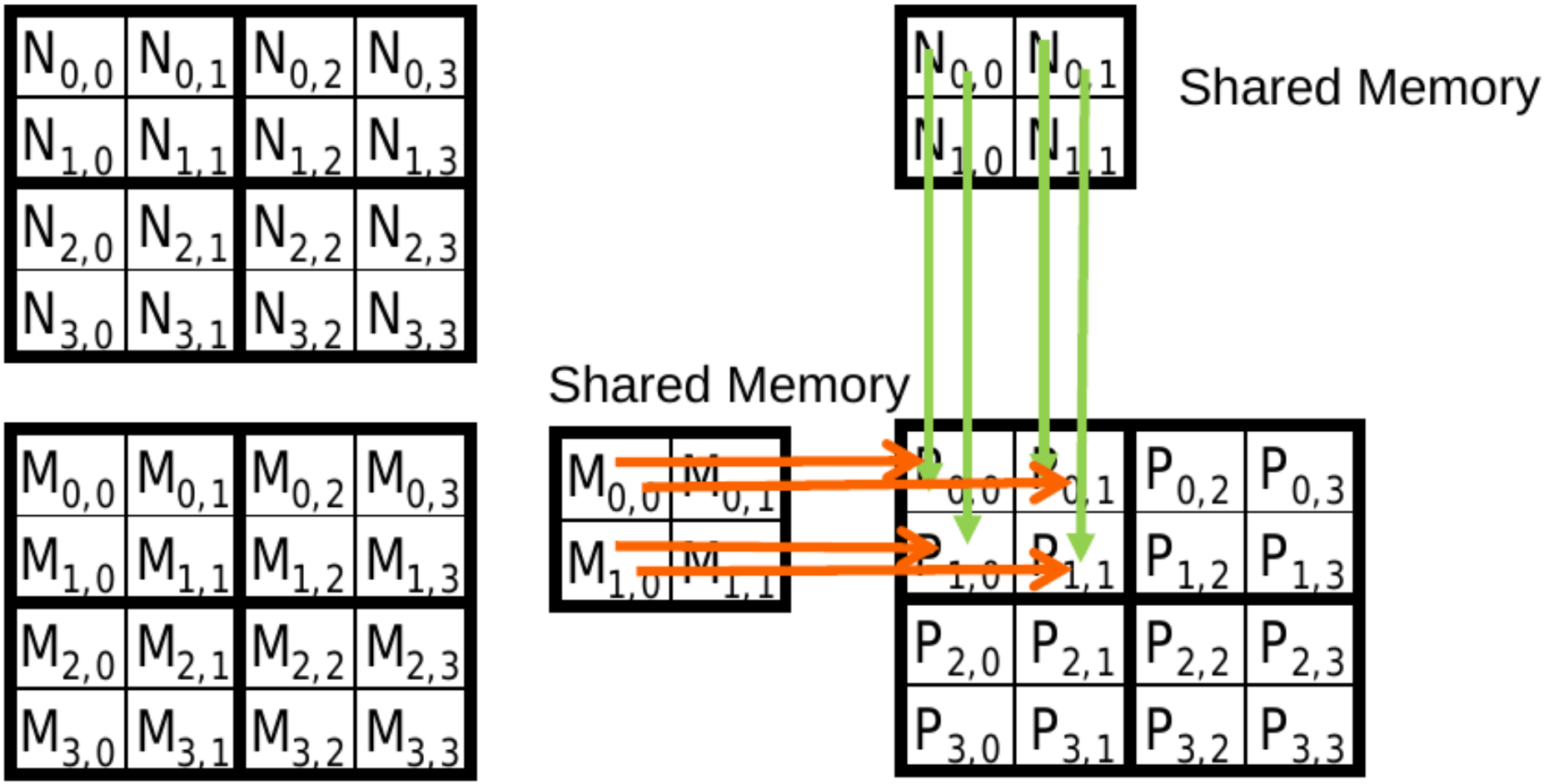
Tous les threads d'un bloc participent

- Chaque thread charge un élement de M et un élement de N dans le code tilisé

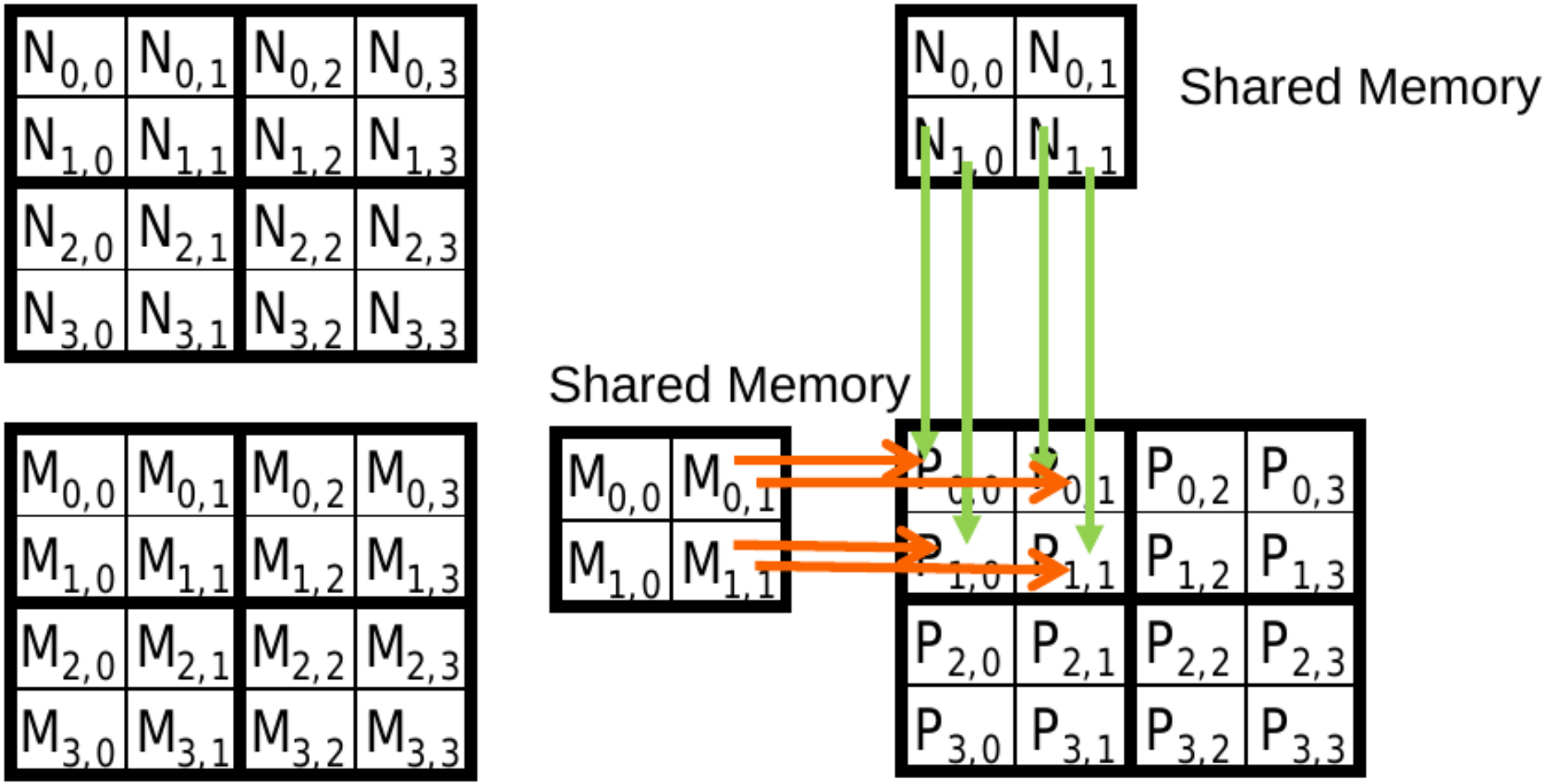
Phase 0 : Chargement du bloc (0,0)



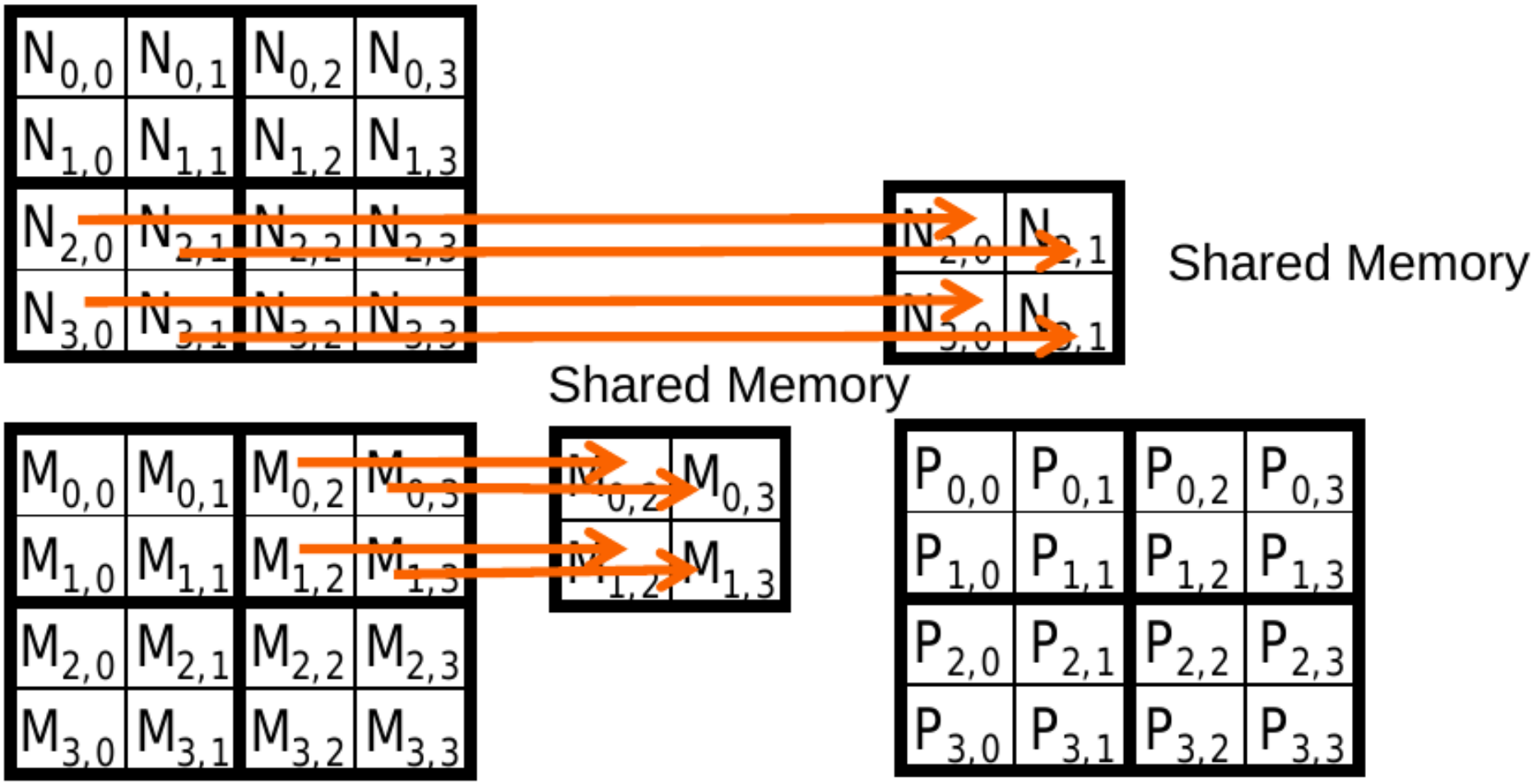
Phase 0 : Utilisation du bloc (0,0) (iteration 0)



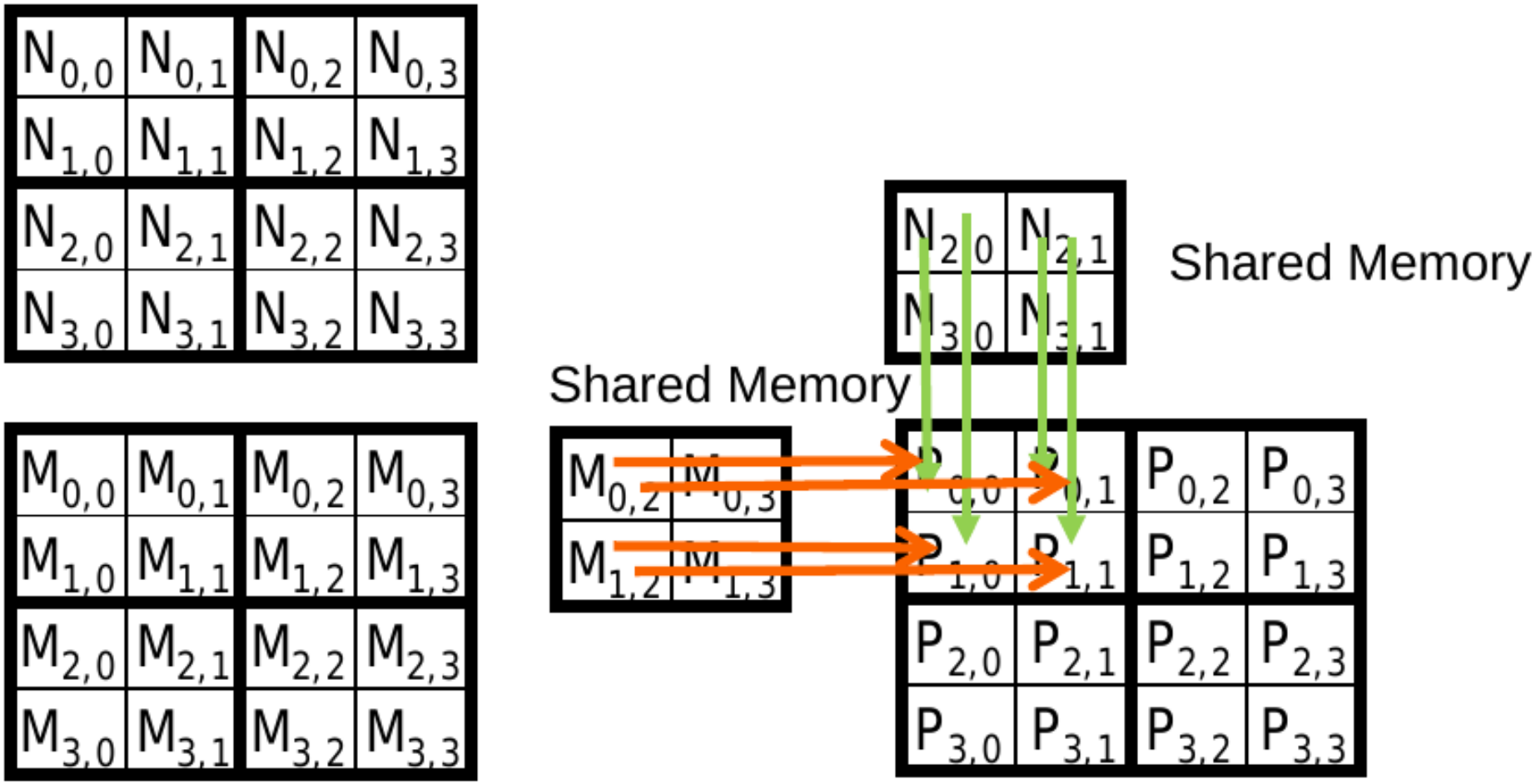
Phase 0 : Utilisation du bloc (0,0) (iteration 1)



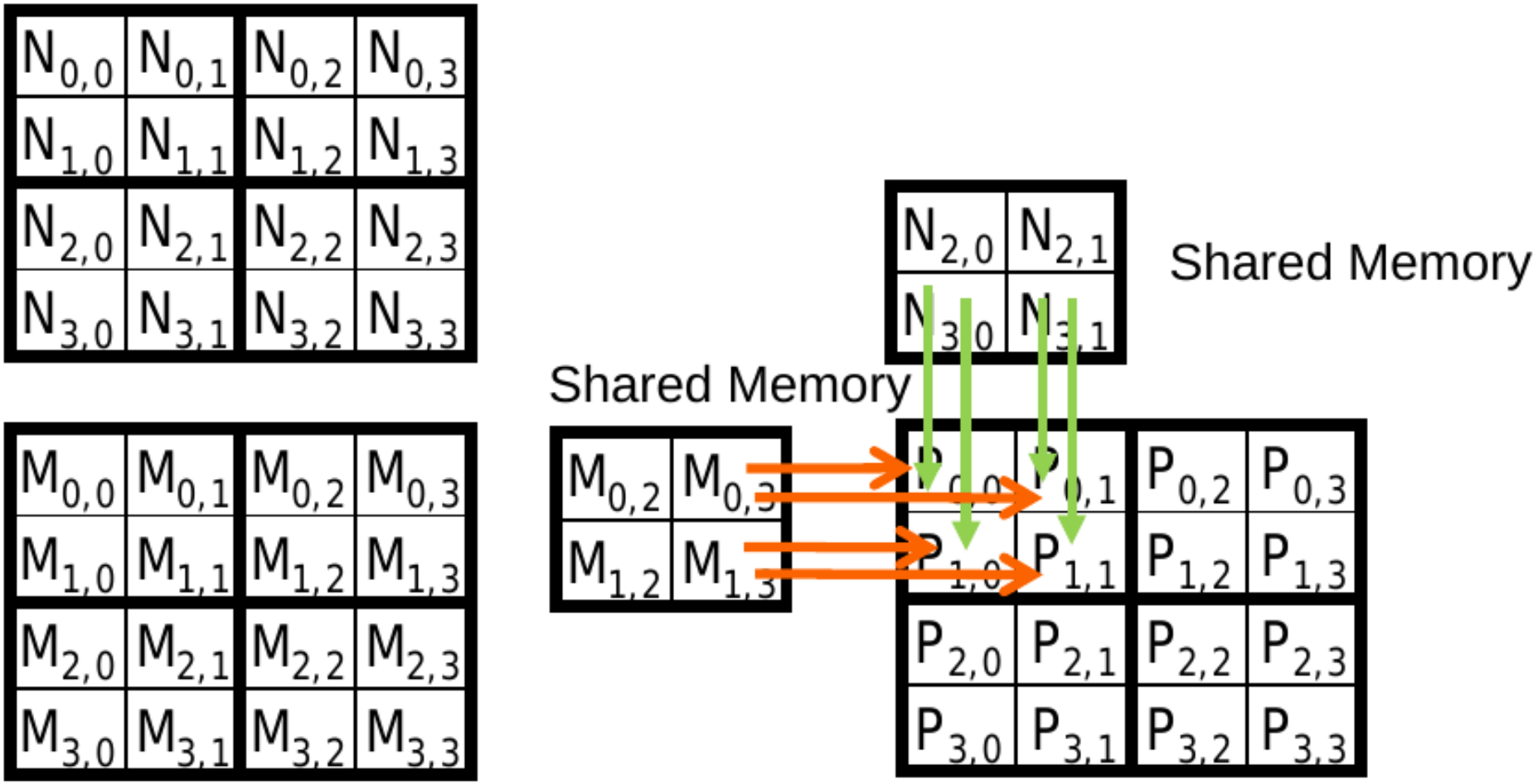
Phase 1 : Chargement du bloc (0,0)



Phase 1 : Utilisation du bloc (0,0) (iteration 0)



Phase 1 : Utilisation du bloc (0,0) (iteration 1)



Exemple d'exécution (1/2)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Exemple d'exécution (2/2)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

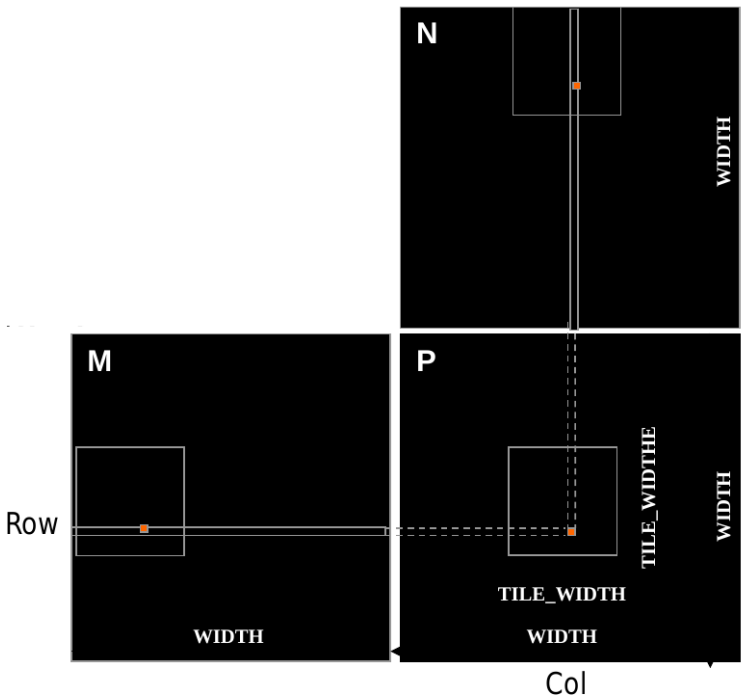
La mémoire partagée permet pour chaque variable d'être accéder par différents threads

Barrière de synchronisation

- Synchroniser tous les threads d'un bloc : __syncthreads()
- Tous les threads d'un même bloc doivent avoir atteint le __syncthreads() avant de pouvoir continuer l'exécution
- La meilleur manière de coordonner la phase d'exécution des algorithmes tilisés
  - est de vérifier que tous les éléments d'un carreau soient chargés au début d'une phase
  - est de vérifier que tous les éléments d'un carreau ont été utilisés à la fin d'une phase

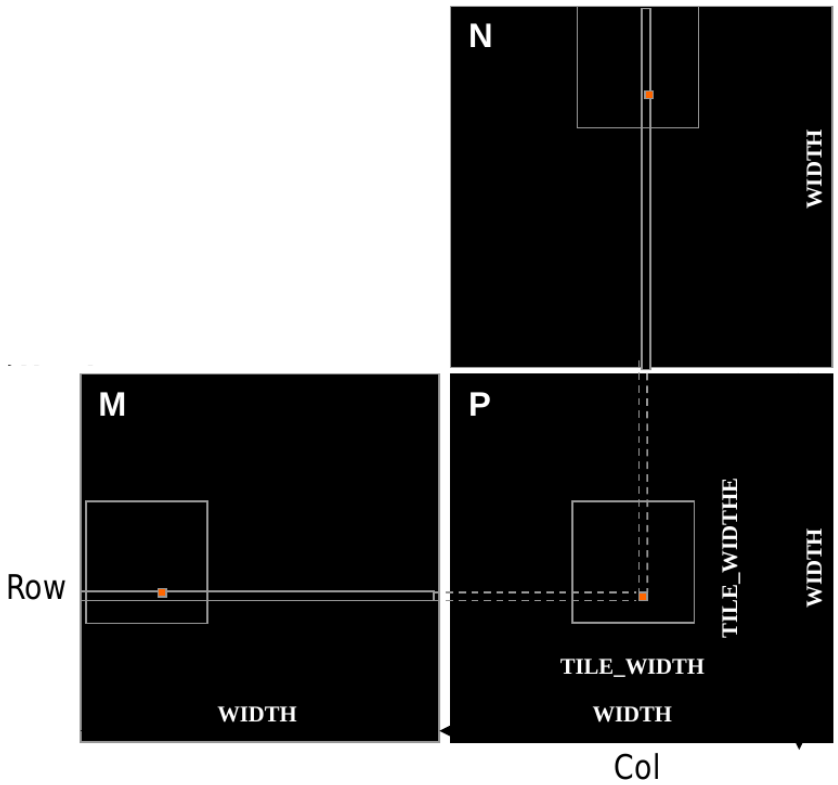
Chargement des entrées du carreau 0 de M et N (Phase 0)

- Chaque thread charge un élément de M et un de N qui seront utilisés pour calculer un élément de P
- int Row = by * blockDim.y + ty;
- int Col = bx * blockDim.x + tx;
- Index 2D pour accéder au carreau 0: $M[Row][tx]$ et $N[ty][Col]$



Chargement des entrées du carreau 1 de M et N (Phase 1)

- Index 2D pour accéder au carreau 0: $M[Row][1 * TILE_W IDTH + tx]$ et $N[1 * TILE_W IDTH + ty][Col]$



M et N sont alloués dynamiquement – Utilisation d'un index 1D

- $M[Row][p * TILE_W IDTH + tx]$ devient
  $M[Row * Width + p * TILE_W IDTH + tx]$
- $N[p * TILE_W IDTH + ty][Col]$ devient
  $N[(p * TILE_W IDTH + ty) * Width + Col]$

- Avec p qui est le numéro de séquence de la phase courante

Multiplication de matrices : Déclarer une variable partagée

```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, Int Width) {
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;
  for (int p = 0; p < n/TILE_WIDTH; ++p) {
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();
    for (int i = 0; i < TILE_WIDTH; ++i)
      Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Considérations sur la taille des carreaux

- Chaque bloc de threads doit avoir autant de threads
  - *TILE_WIDTH* de 16 donne $16 * 16 = 256$ threads
  - *TILE_WIDTH* de 32 donne $32 * 32 = 1024$ threads
- Pour 16, à chaque phase, chaque bloc effectue $2 * 256 = 512$ chargement de flottant depuis la mémoire globale pour $256 * (2 * 16) = 8192$ mul/add operations (16 opérations flottantes pour chaque chargement mémoire)
- Pour 32, à chaque phase, chaque bloc effectue $2 * 1024 = 2048$ chargement de flottant depuis la mémoire globale pour $1024 * (2 * 32) = 65536$ mul/add operations (32 opérations flottantes pour chaque chargement mémoire)

Mémoire partagée et threads

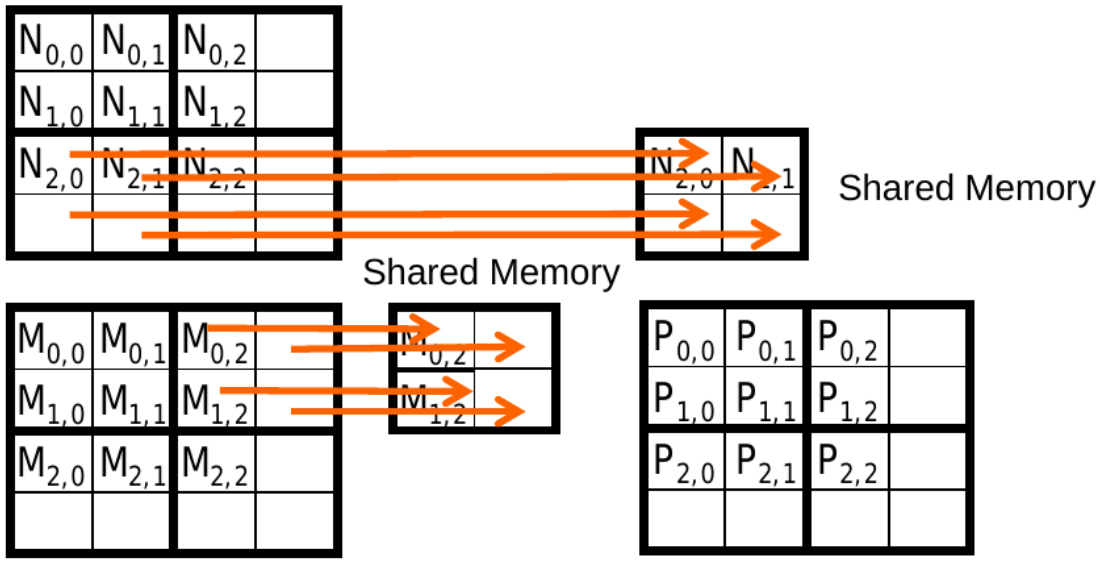
- Pour un SM avec 16KB de mémoire partagée
  - La taille de la mémoire partagée est dépendant de la génération de cartes (entre 16 et 96KB)
  - Pour un $TILE\_WIDTH = 16$, chaque bloc de threads utilise $2 * 256 * 4B = 2KB$ de mémoire partagée
  - Pour 16KB de mémoire partagée, il est potentiellement possible d'avoir 8 bloc de threads qui s'exécutent en parallèle. Cela permet d'avoir jusqu'à $8 * 512 = 4096$ chargement en attente (2 par thread, 256 threads par bloc)
  - Dans le cas de $TILE\_WIDTH = 32$, il y a $2 * 32 * 32 * 4B = 8KB$ de mémoire partagée utilisée par bloc de threads. Cela permet d'avoir jusqu'à 2 blocs de threads actifs en même temps
  - Mais, avec un GPU où le nombre de threads est limité à 1,536 threads par SM, le nombre de blocs par SM est réduit à 1 !
- Chaque `__syncthreads()` peut réduire le nombre de threads actifs pour un bloc.
- Par conséquence, plus de bloc de threads peut être intéressant !

Prendre en charge des matrices de taille arbitraire

- Le noyau de calcul présenté juste avant ne permet que de prendre en charge des matrices carrées dont les dimensions (*Width*) est un multiple de la taille du carreau ($TILE_W IDTH$)
- Pour autant, les vraies applications nécessitent de pouvoir prendre en charge des matrices de tailles arbitraires.
- Une des approches classiques est d'ajouter des éléments (padding) aux lignes et colonnes pour atteindre un multiple de la taille du carreau mais cela nécessite d'utiliser un plus grand espace mémoire et d'augmenter le sur-coût dû au temps de transfert.
- Nous allons voir une autre approche

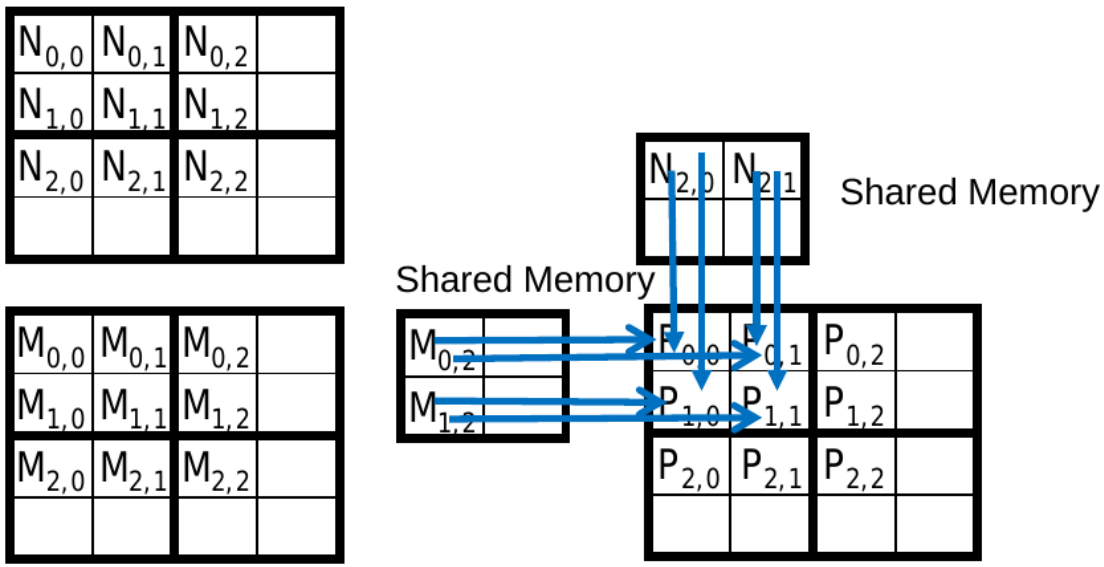
Phase 1 Chargement du bloc (0,0) pour une matrice 3x3

Les threads (1,0) et (1,1) nécessitent un traitement spécial pendant le chargement du carreau N

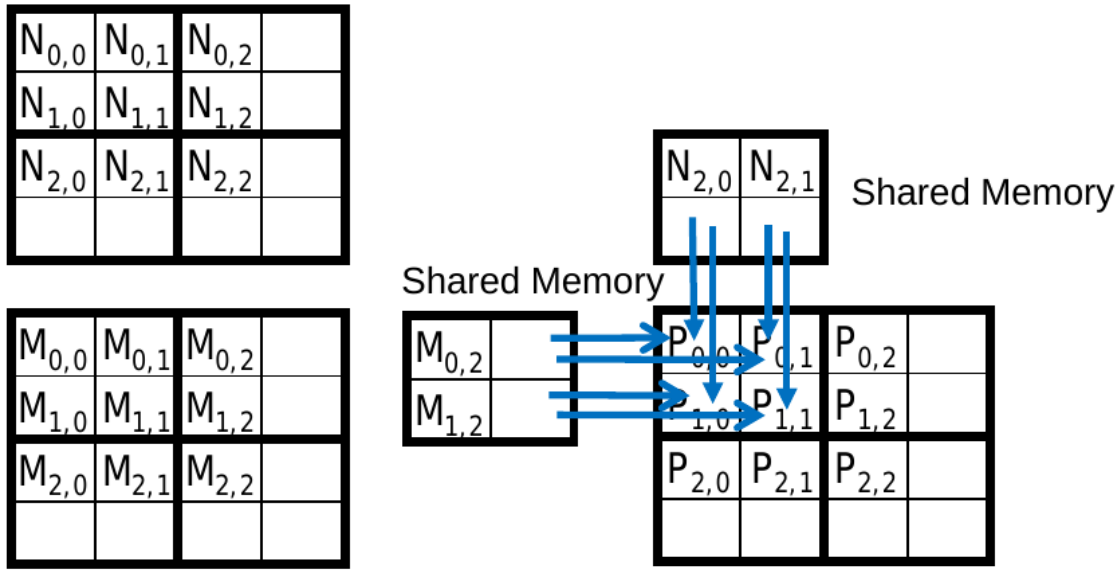


Les threads (0,1) et (1,1) nécessitent un traitement spécial pendant le chargement du carreau M

Phase 1 Utilisation du bloc (0,0) (iteration 0)



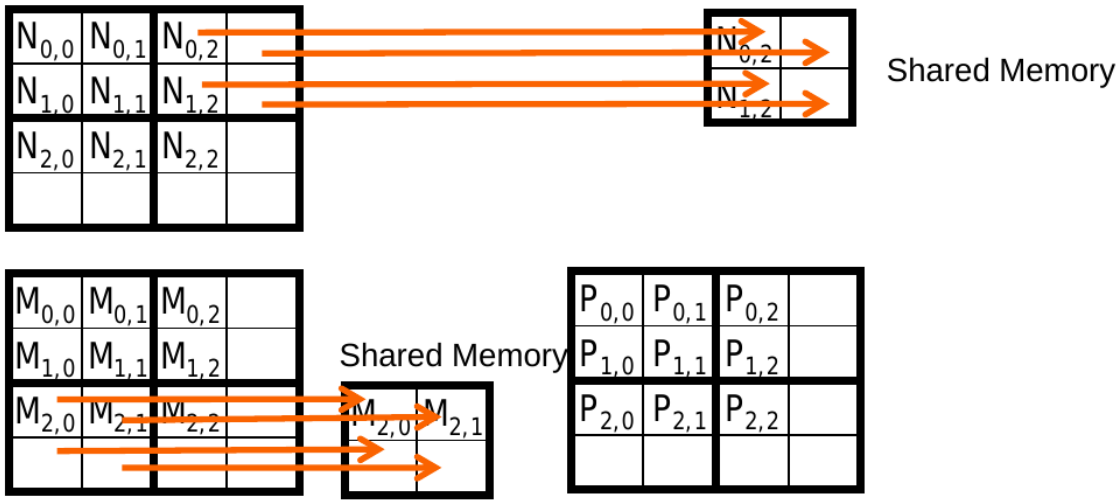
Phase 1 Utilisation du bloc (0,0) (iteration 1)



Tous les threads nécessitent un traitement spécial. Aucun doit introduire des contributions invalide à leurs éléments dans P.

Phase 1 Chargement du bloc (1,1) pour une matrice 3x3

Les threads (0,1) et (1,1) nécessitent un traitement spécial pendant le chargement du carreau N



Les threads (1,0) et (1,1) nécessitent un traitement spécial pendant le chargement du carreau M

Retour sur les cas d'exception

- Les threads qui ne calculent pas d'élements valides de P mais participent tout de même à charger des carreaux
  - Phase 0 du bloc (1,1), le thread (1,0), assigné à calculer un élément non-existant $P[3,2]$ mais participe au chargement de l'élément du carreau $N[1,2]$

- Les threads qui calculent un élement valides de P et qui pourrait tenter de charger des éléments non-existants au moment du chargement du carreau
  - Phase 0 du bloc (0,0), le thread (1,0), assigné à calculer un élément existant $P[1,0]$ mais tente de charger un élément non existant $N[3,0]$

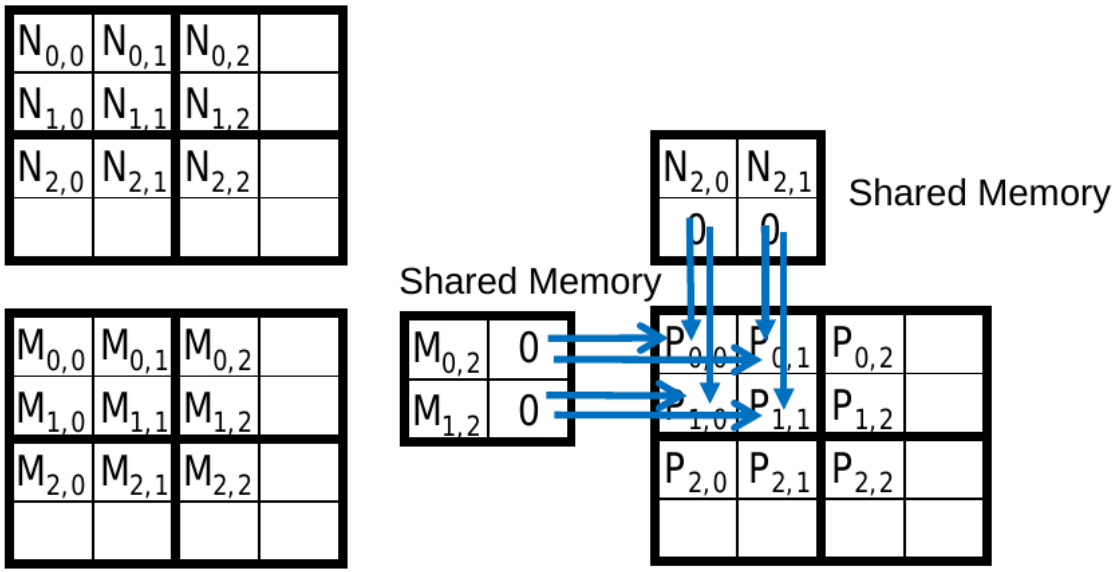
Une solution simple

- Quand un thread charge un élément en entrée, tester si il est dans l'ensemble valide des indexes
  - Si valide, alors effectuer le chargement
  - Sinon, ne pas charger et juste écrire 0

- Raisonnement : une valeur 0 permet de s'assurer que l'étape de multiplication-addition ne va pas affecter la valeur finale de l'élément de sortie

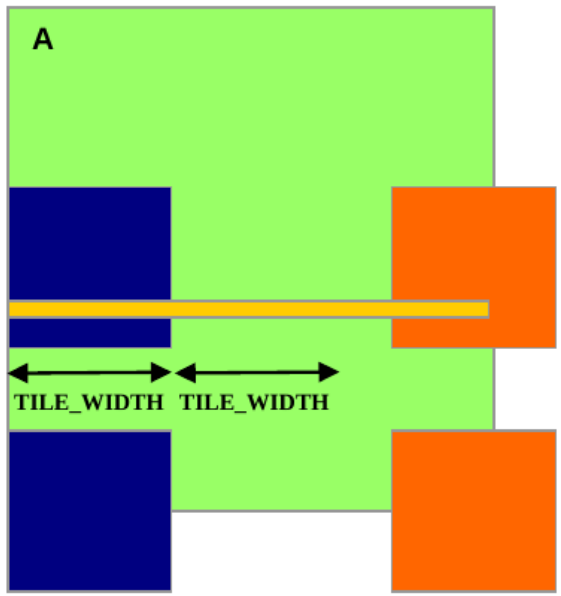
- La condition testé pour le chargement des éléments en entrée est différente du test pour calculer l'élément de sortie P
  - Un thread qui ne calcule pas d'élément valide de P peut tout de même participer aux chargement des données du carreau

Phase 1 Chargement du bloc (0,0) (iteration 1)



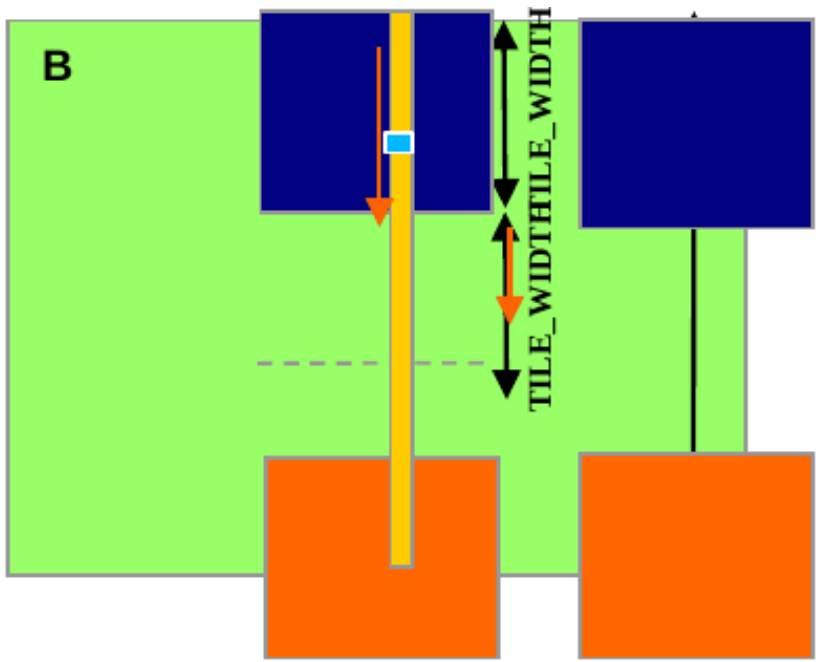
Condition limite pour le carreau chargeant l'entrée *M*

- Chaque thread charge
  - M[Row][p∗TILE_WIDTH+tx]
  - M[Row∗Width + p∗TILE_WIDTH+tx]
- Chaque thread doit tester
  - (Row < Width) && (p∗TILE_WIDTH+tx < Width)
  - Si vrai, alors charge l'élément depuis *M*
  - Sinon, charge 0



Condition limite pour le carreau chargeant l'entrée *N*

- Chaque thread charge
  - N[p∗TILE_WIDTH+ty][Col]
  - N[(p∗TILE_WIDTH+ty)∗Width+ Col]
- Chaque thread doit tester
  - (p∗TILE_WIDTH+ty $<$ Width) && (Col$<$ Width)
  - Si vrai, alors charge l'élément depuis *N*
  - Sinon, charge 0



Chargement des éléments avec vérification des bornes

```
for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
  if(Row < Width && t * TILE_WIDTH+tx < Width) {
    ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
  } else {
    ds_M[ty][tx] = 0.0;
  }

  if (p*TILE_WIDTH+ty < Width && Col < Width) {
    ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
  } else {
    ds_N[ty][tx] = 0.0;
  }

  __syncthreads();
```

Multiplication-Addition modifié

```
if(Row < Width && Col < Width) {
  for (int i = 0; i < TILE_WIDTH; ++i) {
    Pvalue += ds_M[ty][i] * ds_N[i][tx];
  }

__syncthreads();
}

if (Row < Width && Col < Width)
  P[Row*Width + Col] = Pvalue;
```

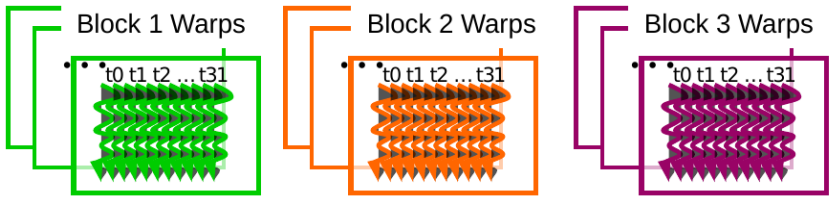
Quelques points important

- Pour chaque thread, les conditions sont différentes pour
  - le chargement des éléments de M
  - le chargement des éléments de N
  - Calculer et stocker les éléments en sortie
- L'effet de la divergence sur le contrôle devraient être minimale avec des grandes matrices

Cas général de manipulation des matrices rectangulaires

- En générale, la multiplication de matrice est définie en terme de matrices rectangulaires
  - Une matrice *jxk* M multiplié par une matrice *kxl* N avec le résultat dans une matrice *jxl* P
- Dans ce cours, il a été présenté le cas particulié de la matrice carrée
- La fonction noyau nécessite d'être généralisé pour pouvoir manipuler tous les types de matrices rectangulaires
  - L'argument *Width* doit être remplacé par 3 arguments : j, k et l
  - Quand *Width* est utiliser pour se référer à la hauteur de *M* ou de *P*, remplacer par *j*
  - Quand *Width* est utiliser pour se référer à la largeur de *M* ou la hauteur de *N*, remplacer par *k*
  - Quand *Width* est utiliser pour se référer à la largeur de *N* ou la largeur de *P*, remplacer par *l*

Warps comme unité d'ordonnnancement



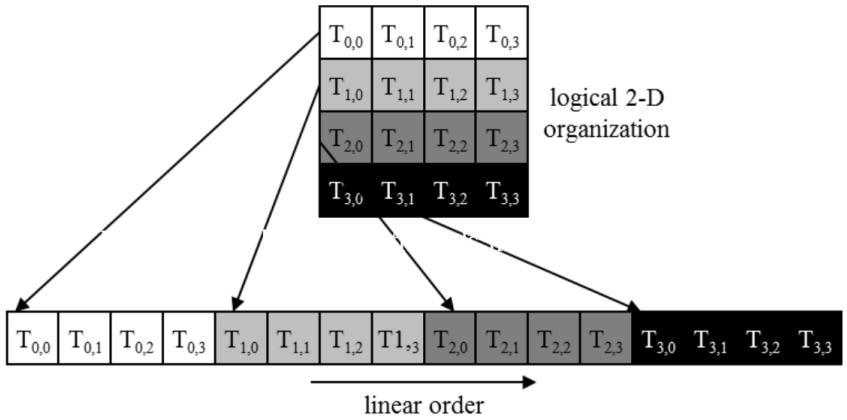
Chaque bloc est découpé en warp de 32 threads

- La partie technique du découpage ne fait pas parti de CUDA mais est un détail d'implémentation
- Les warps sont les unité de base d'ordonnnancement sur les SMs
- Les threads composant un warp s'exécutent de la même manière que du SIMD
- Le nombre de threads au sein d'un warp pourrait évoluer dans le futur

Les warps et les bloc de threads multidimensions

Les blocs de threads sont déjà linéarisés en une ligne ordonnée 1D

- D'abord suivant la dimension x, puis y et finalement z

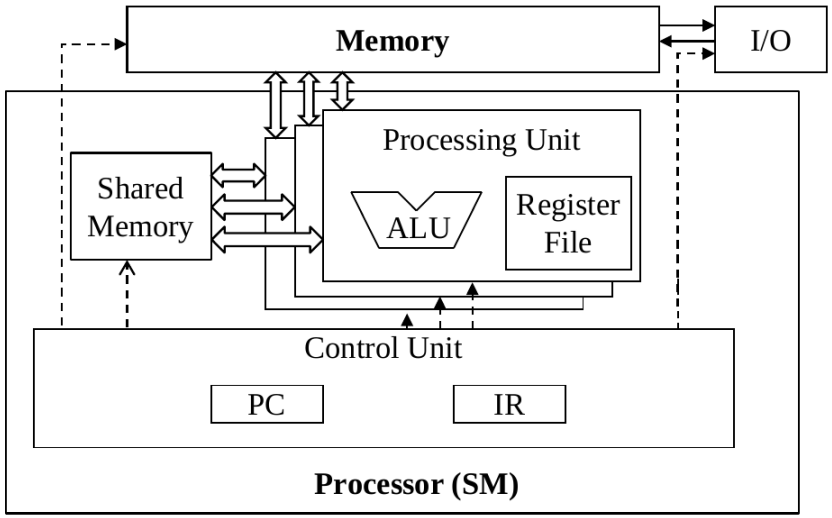


Les blocs sont partionnés après la linéarisation

- Les blocs linéarisé de threads sont partionnés
  - Les indexes de threads au sein d'un warp sont consécutifs et croissant
  - Warp 0 commence par le thread 0
- Le même schéma de partionnement est utilisé sur tous les devices
  - Vous pouvez donc utiliser cette connaissance lors de la définition du flux de contrôle
  - Pour autant, d'une génération à l'autre de carte ce nombre pourrait changer
- NE PAS SE BASER sur un ordre quelconque au sein ou entre les warps
  - Si il y a une dépendance entre les threads, il reste nécessaire d'utiliser `__syncthreads()`

Les SMs sont des processeurs SIMD

L'unité de contrôle qui charge les instructions, les décode et les contrôle est partagée entre les différentes unités de calcul



Exécution SIMD et Threads dans un Warp

- A tout moment, tous les threads d'un warp doivent exécuter la même instructions
- Cela fonctionne efficacement si tous les threads suivent le même chemin dans le flux de contrôle
  - Toutes les déclarations if-then-else prennent la même décision
  - Toutes les boucles doivent avoir le même nombre d'itérations

Contrôle de la divergence

- La divergence du flux de contrôle arrive quand les threads d'un warp prennent différent chemins dans le flux de contrôle en prennant des décisions de contrôles différentes
  - Certains prennent le chemin then et d'autres le chemin else pour une même déclaration if
  - Certains threads ont un nombre d'itérations de boucle différent des autres

- L'exécution des threads prenant un chemin différent est actuellement sérialisée sur GPU
  - Les chemins de contrôle pris par les threads au sein d'un warp sont parcouru un à un jusqu'à ce qu'il n'y en ait plus
  - Durant l'exécution de chaque chemin, tous les threads prenant ce chemin sont exécutés en parallèle
  - Le nombre de chemins différents peut être grand lorsqu'on considére les déclarations de flux de contrôle imbriqué (*e.g.*, récursion)

Exemples de cas de divergence

- La divergence peut apparaitre quand les conditions de branchement et/ou de boucle se font en fonction des indexes des threads

- Exemple de la déclaration d'un noyau avec divergence
  - if (threadIdx.x > 2)
  - Cela créé 2 chemins différents de contrôle pour les threads au sein d'un bloc
  - La granularité des décisions < la taille du warp; threads 0, 1 et 2 suivent un chemin et les autres un autre alors qu'ils font parti d'un même warp

- Example sans divergence
  - if (blockIdx.x > 2)
  - La granularité de décision est un multiple de la taille des blocs, tous les threads dans n'importe quel warp suivent le même chemin

Exemple : Noyau d'addition de vecteurs

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C,
int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

Analyse pour un vecteur de 1,000 éléments

- Nous supposons une taille de bloc de 256 threads
  - 8 warps par bloc
- Tous les threads des blocs 0, 1 et 2 sont dans une gamme valide
  - i valeurs de 0 à 767
  - Il y a 24 warps dans ces 3 blocs, aucun n'auront de divergence de contrôle
- La plupart des warps du bloc 3 n'ont pas de divergence de contrôle
  - Les threads dans les warps 0-6 ont tous des gammes valides et donc pas de divergence de contrôle
- Un warp dans le bloc 3 aura une divergence de contrôle
  - Les threads avec des valeurs de i entre 992 et 999 seront au sein d'une gamme valide
  - Les threads avec des valeurs de i entre 1000 et 1023 seront en dehors d'une gamme valide
- L'effet de la sérialisation sur le contrôle de la divergence devrait être faible
  - 1 parmi 32 warps subira une divergence de contrôle
  - L'impact sur les performances devrait être inférieur à 3%

Impact sur les performances de la divergence de contrôle

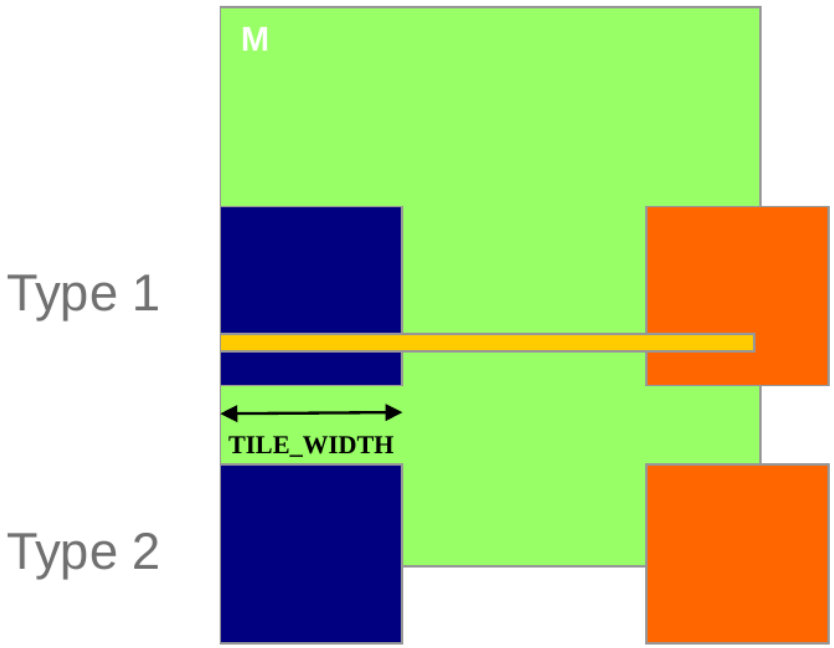
La vérification des conditions de borne est vitale pour avoir un code parallèle complet fonctionnel et robuste

- Le noyau de multiplication de matrices tilisé contient de nombreuses vérifications de bornes
- Le problème est que ces vérifications peuvent provoquer des dégradations significatives de performance
- Par exemple, quand le code du carreau suivant est chargé

```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
  ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
} else {
  ds_M[ty][tx] = 0.0;
}
if (p*TILE_WIDTH+ty < Width && Col < Width) {
  ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
} else {
  ds_N[ty][tx] = 0.0;
}
```

2 types de blocs lors du chargement des carreaux de M

1. Les blocs dont les carreaux sont entièrement dans des gammes valides jusqu'à la phase finale
2. Les blocs dont les carreaux sont partiellement en dehors des gammes valides



Analyse de l'impact de la divergence de contrôle

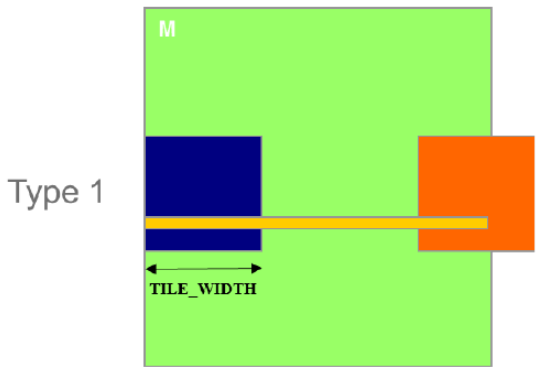
- Supposons des carreaux de 16x16 et des blocs de threads de 256
- Chaque bloc de threads a 8 warps (256/32)
- Supposons des matrices carrées de 100x100
- Chaque thread doit passer par 7 phases (*ceil*(100/16))

- Il y a 49 blocs de threads (7 dans chaque dimension)

Divergence de contrôle lors du chargement (Type 1)

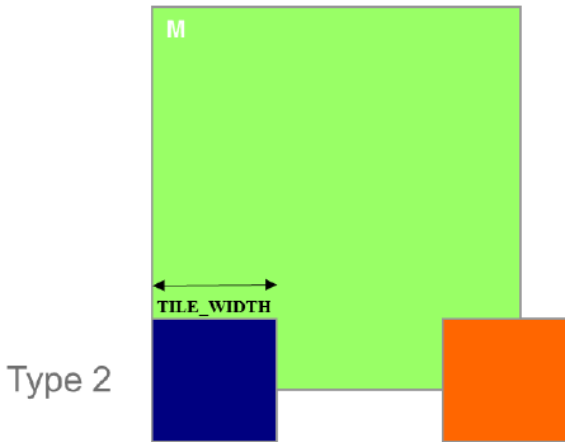
- Supposons des carreaux de 16x16 et des blocs de threads de 256
- Chaque bloc de threads a 8 warps (256/32)
- Supposons des matrices carrées de 100x100
- Chaque thread doit passer par 7 phases (*ceil*(100/16))

- Il y a 42 (6*7) blocs de de Type 1 pour un total de 336 ($8 * 42$) warps
- Ils ont chacun 7 phases et donc il y a 2,352 ($336 * 7$) phases de warps
- Les warps ont une divergence de contrôle seulement lors de leur dernière phase
- 336 phases de warps ont une divergence de contrôle



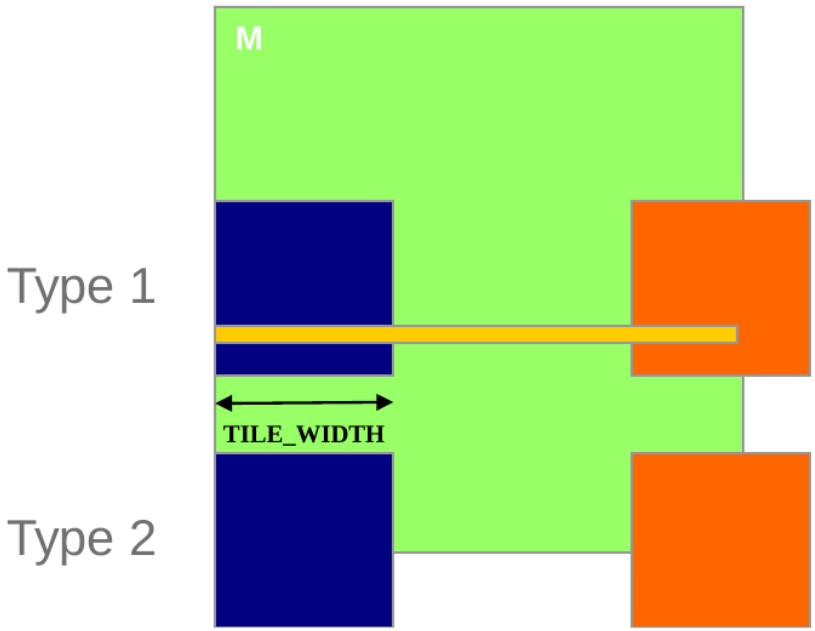
Divergence de contrôle lors du chargement (Type 2)

- Type 2 : les 7 blocs assignés au chargement des carreaux en bas pour un total de 56 $(8 * 7)$ warps
- Ils ont tous 7 phases, il y a donc 392 $(56 * 7)$ phases de warps
- Les 2ers warps de chaque bloc de Type 2 resteront dans une gamme valide jusqu'à la dernière phase
- Les 6 warps restants restent au sein de la gamme valide
- Il y a donc seulement 14 $(2 * 7)$ phases de warps qui ont de la divergence de contrôle



Impact global de la divergence de contrôle

- Blocs de Type 1 : 336 sur 2,532 phases de warps ont de la divergence de contrôle
- Blocs de Type 2 : 14 sur 392 phases de warps ont de la divergence de contrôle
- L'impact sur la performance est prévu d'être inférieur à 12% (350/2944 ou $(336 + 14)/(2352 + 392)$)



Quelques remarques

- Le calcul de l'impact de la divergence de contrôle dans le cas du chargement des carreaux N n'est pas équivalent à celui de M
- L'impact estimé sur les performances est dépendant des données
  - Pour les grandes matrices, l'impact devrait être significativement plus petit
- En général, l'impact de la divergence de contrôle pour les conditions de vérification de bornes sur les larges ensembles de données en entrée est considéré négligeable
  - Il ne faut donc pas hésiter à utiliser ces vérifications de borne pour s'assurer d'un code complétement fonctionnel
- Le fait qu'un noyau soit composé de plein de structures de contrôle de flux ne veut pas forcément dire que son exécution provoquera beaucoup de divergence de contrôle
- Nous verrons plus tard des algorithmes qui naturellement provoque de la divergence de contrôle (parallel reduction)

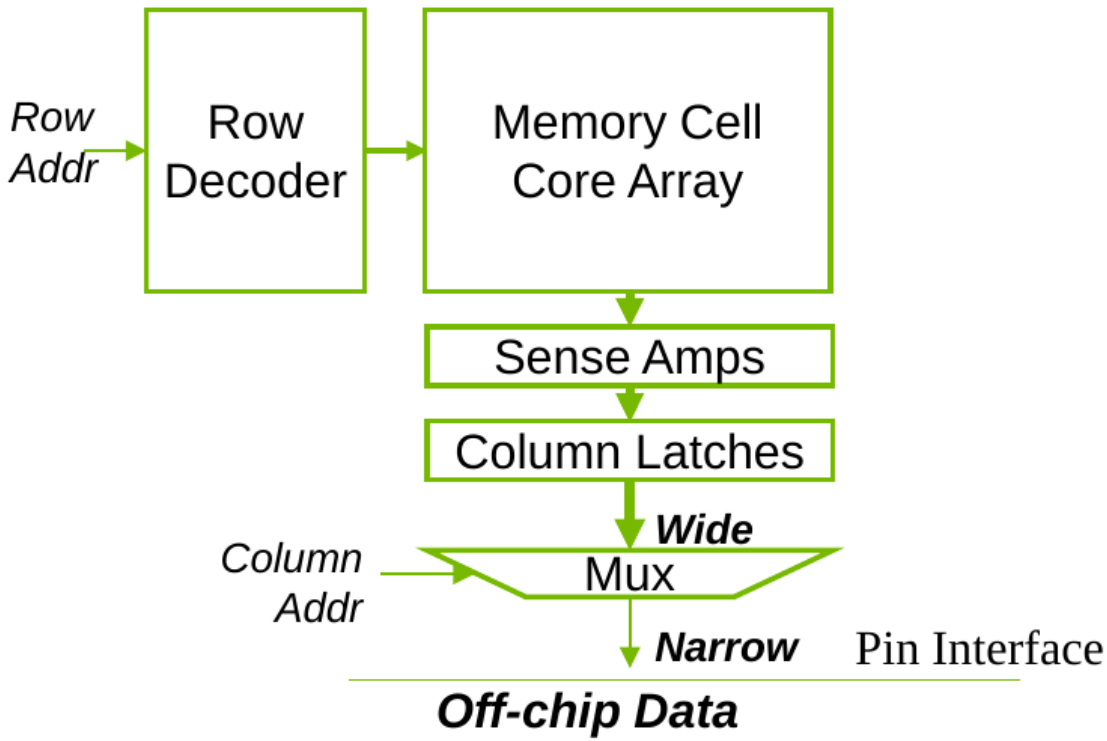
Bande passante mémoire globale (DRAM)

Ideal

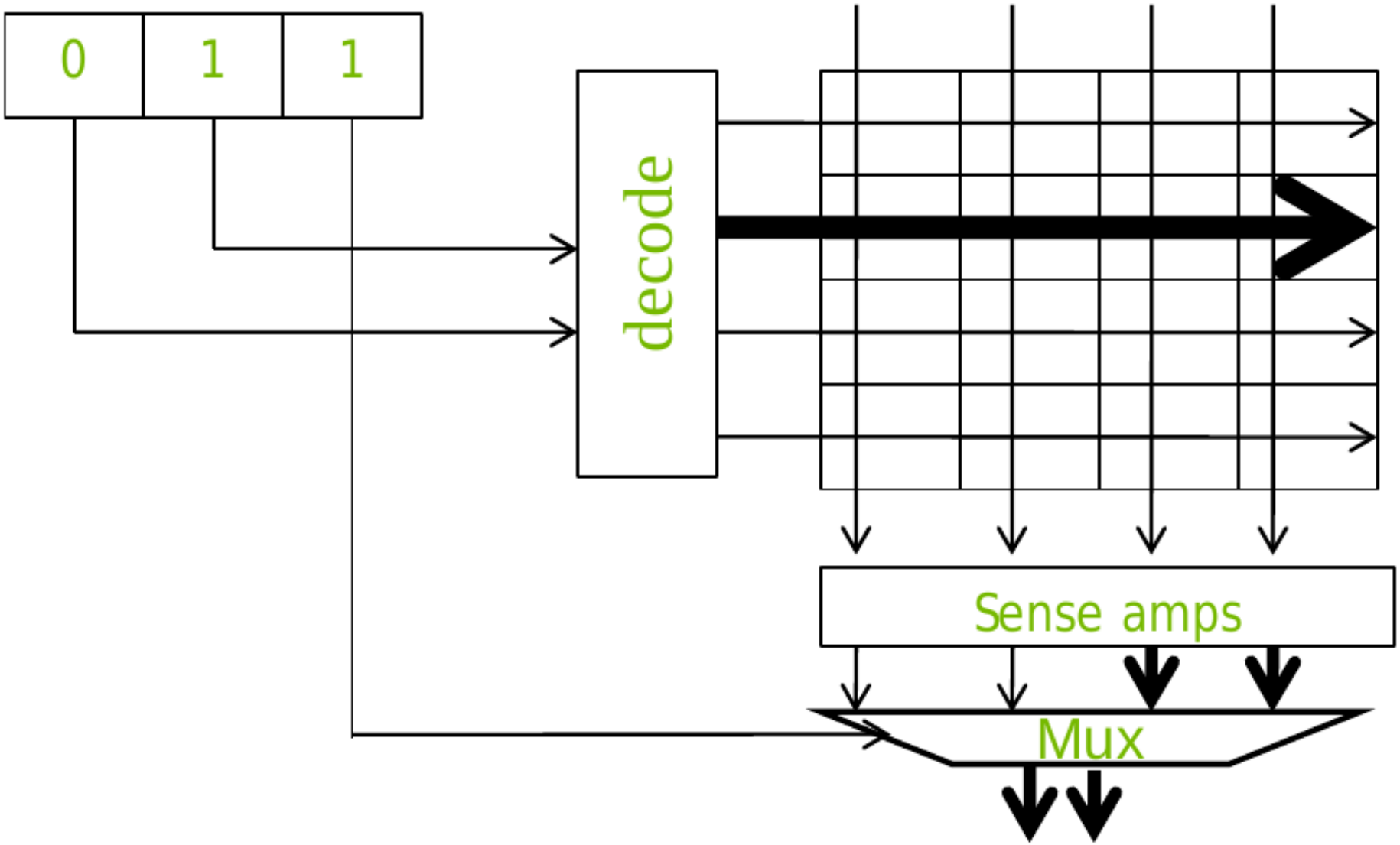
Reality

Organisation par tableaux de la DRAM

- Chaque tableau au sein de la DRAM est composé de 16M de bits
- Chaque bit est stockée sur un petit condensateur composé d'un transistor



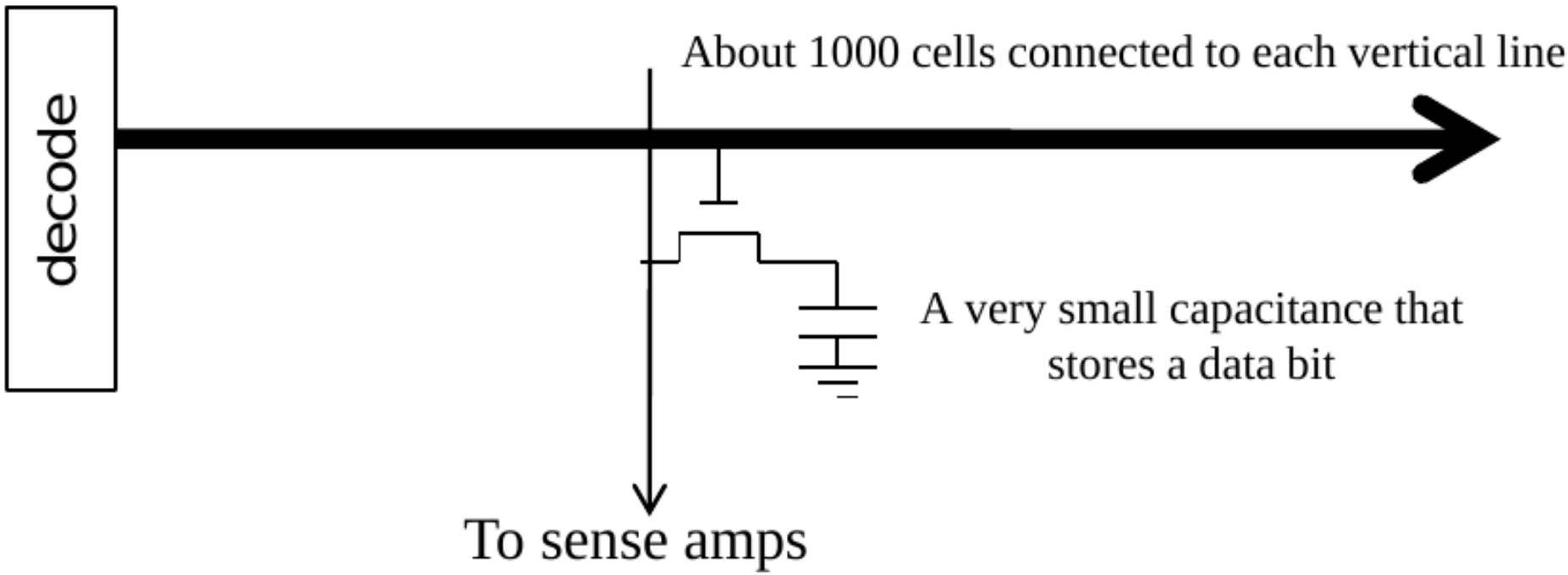
Un tout petit tableau de 8x2-bit en DRAM



Lire dans les tableaux au sein de la DRAM est lent

Lire une cellule dans un tableau est un processus très lent

- DDR : Vitesse = 1/2 de la vitesse de l'interface
- DDR2/GDDR3 : Vitesse = 1/4 de la vitesse de l'interface
- DDR3/GDDR4 : Vitesse = 1/8 de la vitesse de l'interface
- Vraisemblablement pire dans le futur

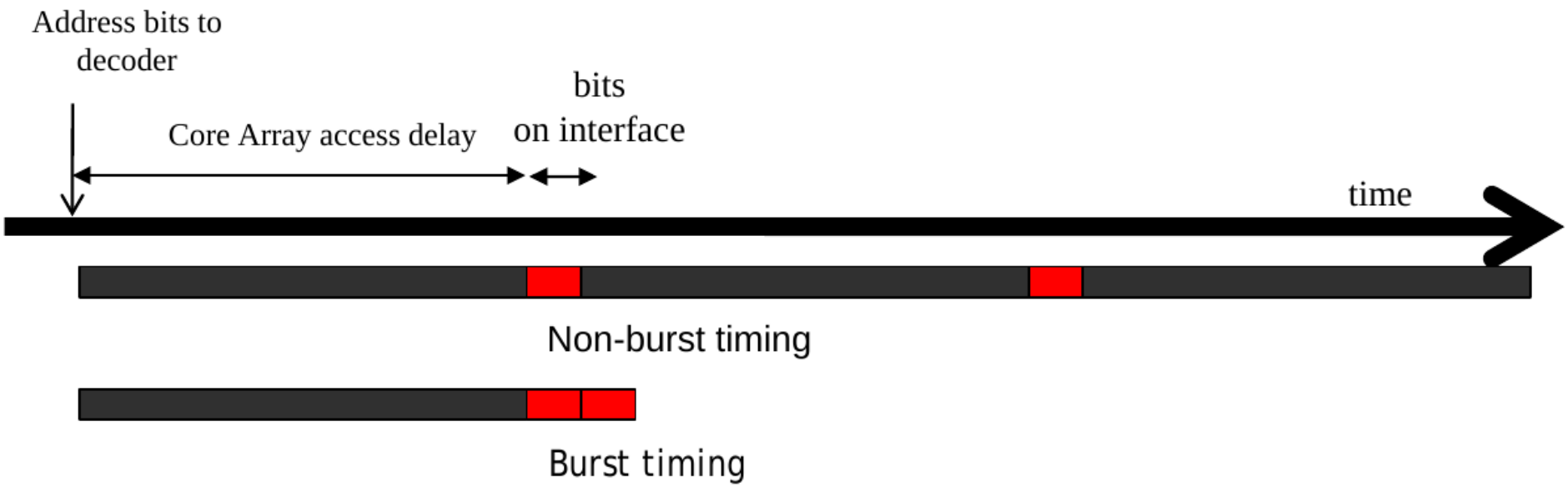


DRAM Bursting

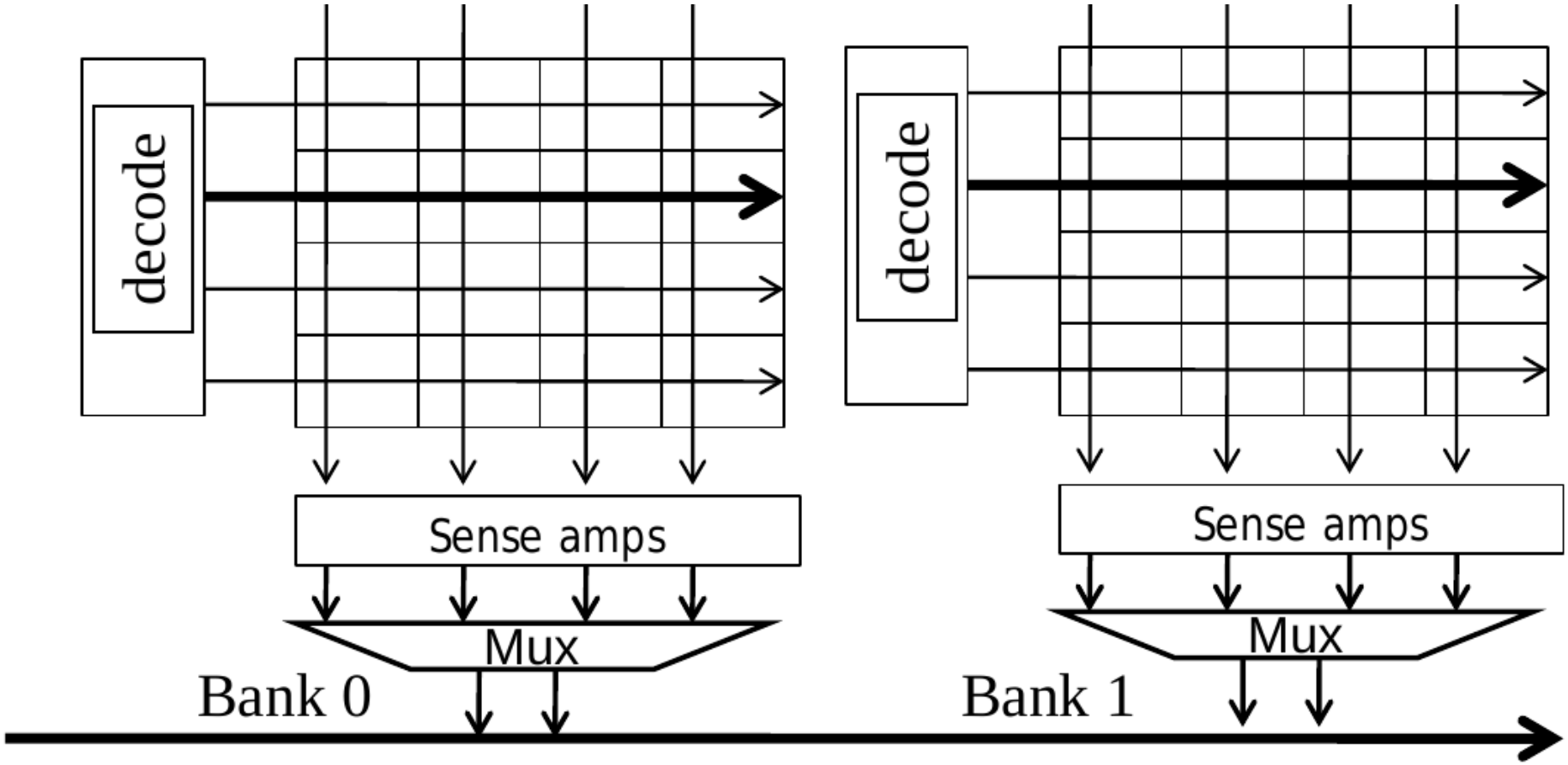
Pour la DDR2,3, les coeurs SDRAM sont calibrés à $1/N$ de la vitesse de l'interface

- Chargement de (N x largeur de l'interface) DRAM bits depuis la même ligne au sein d'un buffer interne puis transfer de N étapes à la vitesse de l'interface
- DDR3/GDDR4 : la taille du buffer = 8X la largeur de l'interface

DRAM Bursting: Exemple



Plusieurs barrettes de DRAM



DRAM Bursting avec plusieurs barrettes

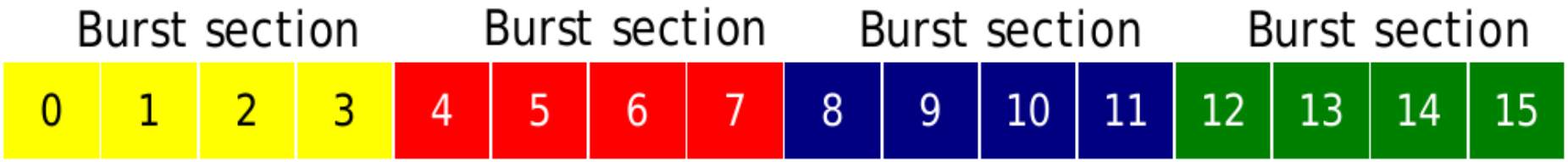
Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

Sous système de mémoire hors chip

- NVidia GRX280 GPU: Bande passante maximum de la mémoire globale = 141.7GB/s
- Mémoire globale (GDDR3) avec une interface à 1.1GHz
  - Vitesse des coeurs 276Mhz
  - Pour une interface typique de 64-bit, il est possible de maintenir uniquement 17.6GB/s (2 transferts par cycle d'horloge)
  - Il est nécessaire d'avoir beaucoup plus de bande passante (141.7GB/s) alors 8 canaux mémoires

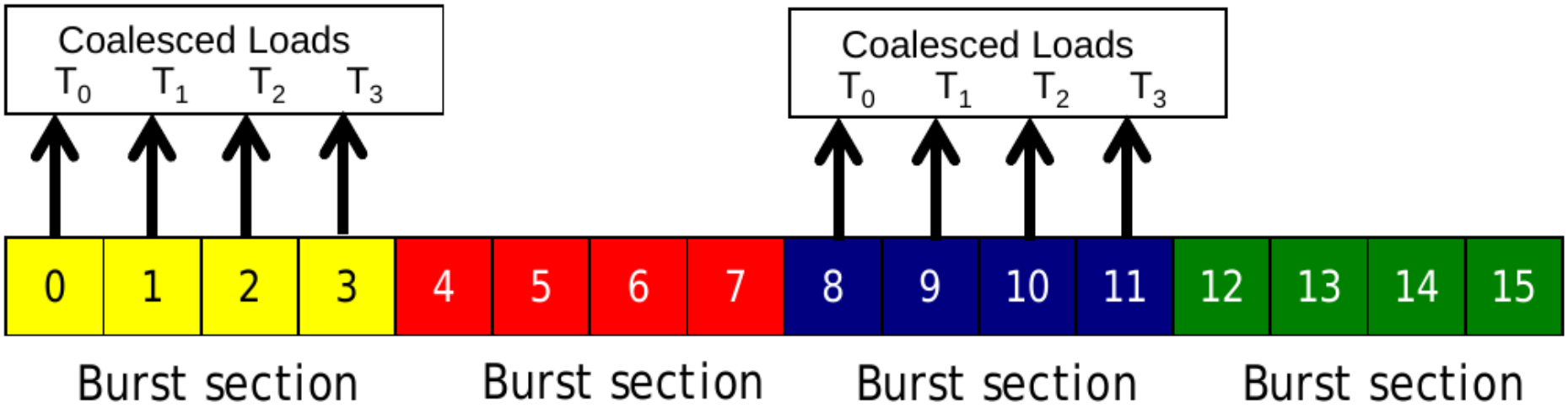
DRAM Burst : une vue système



- Chaque espace d'adresse est partitionné en sections de burst
  - Chaque fois qu'une localité est accédée, toutes les autres localité dans la même section sont également fournies au processeurs

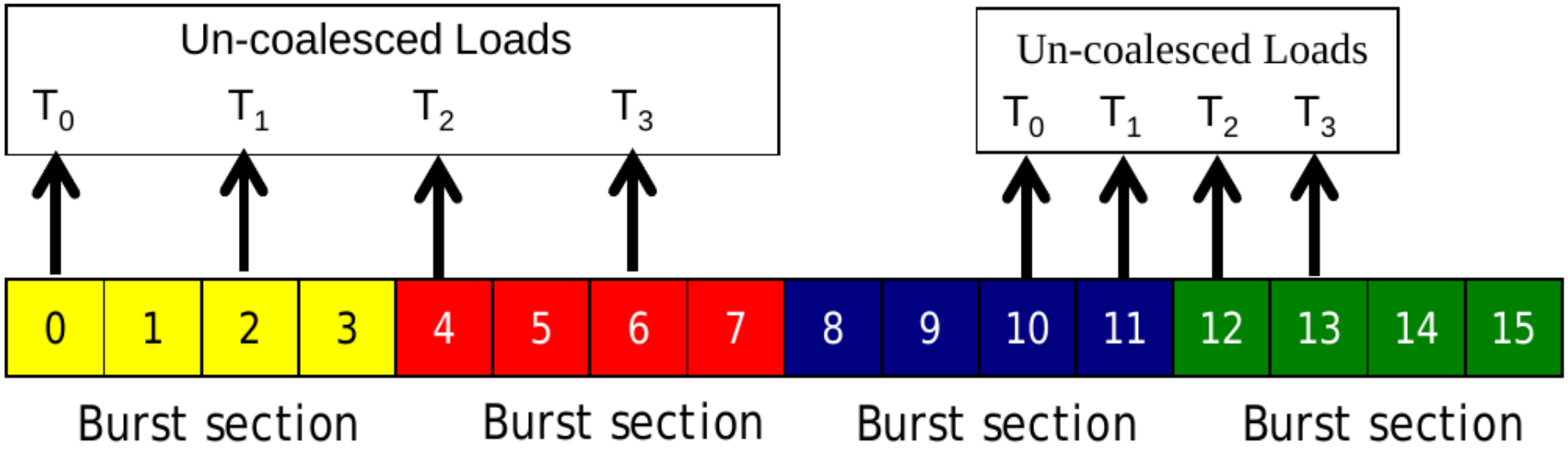
- Exemple simple : un espace d'adressage de 16-byte, des sections burst de 4-byte
  - En pratique, nous avons des espaces d'adressage de 4GB et des sections de bursts de 128 ou plus

Groupement mémoire (memory coalescing)



Quand tous les threads d'un warp exécute la même instruction de chargement, si tous les localités accédées tombent dans la même région de burst, seulement une requête DRAM sera effectuée et l'accés est dit complétement groupé (fully coalesced).

Accés non-groupé (un-coalesced)



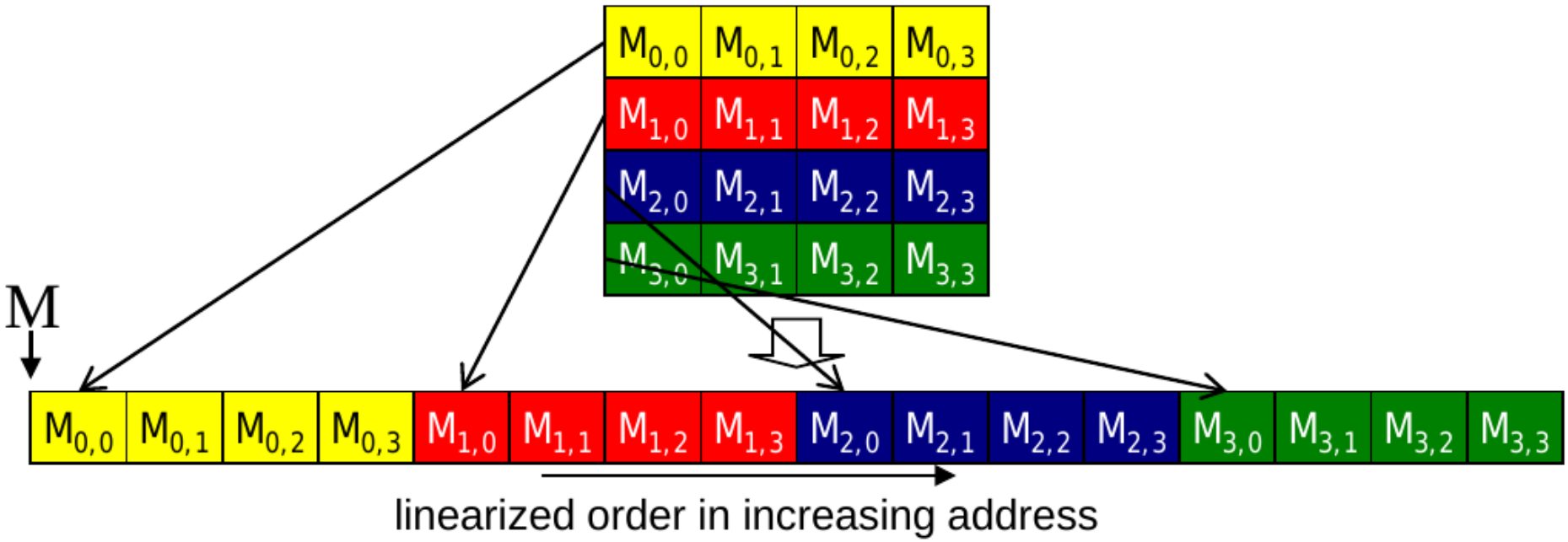
- Quand les localités accédées sont réparties entre plusieurs sections de burst
  - Le regroupement échoue
  - Plusieurs requêtes DRAM sont effectuées
  - L'accés n'est pas complétement groupé

- Certains bytes accédés et transférés ne sont pas utilisés par les threads

Comment savoir si un accés est groupé

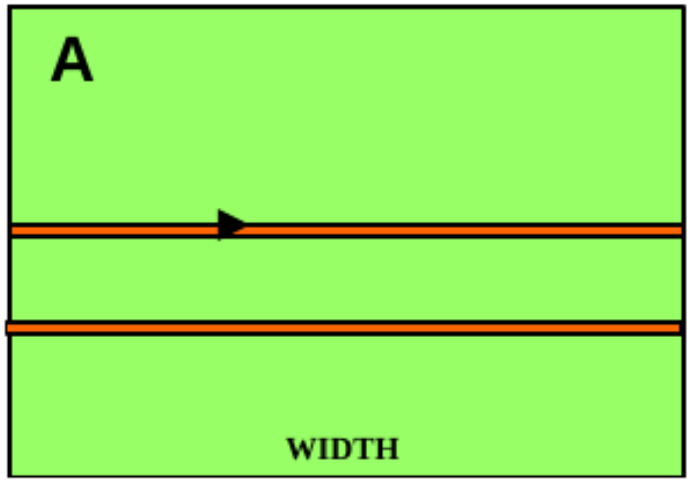
- Les accés dans un warp sont dans des localités consécutives si l'index d'un accés dans un tableau est de la forme suivante:
  - A[(expression with terms independent of threadIdx.x) + threadIdx.x];

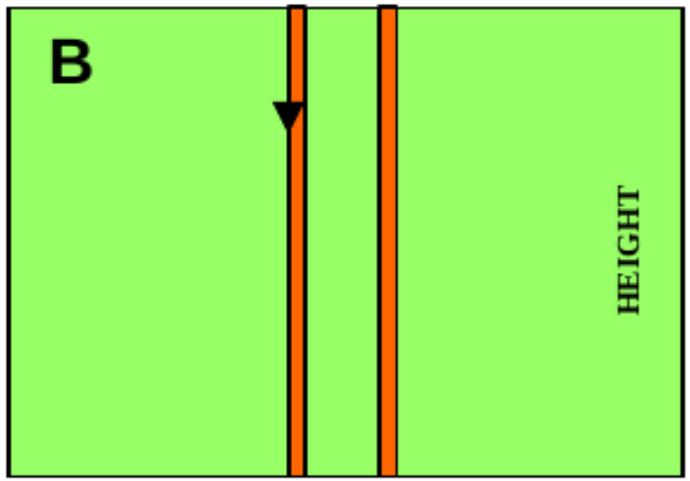
Un tableau 2D en C dans un espace mémoire linéaire



Deux modèles d'accès pour la multiplication de matrices

Thread 1
Thread 2



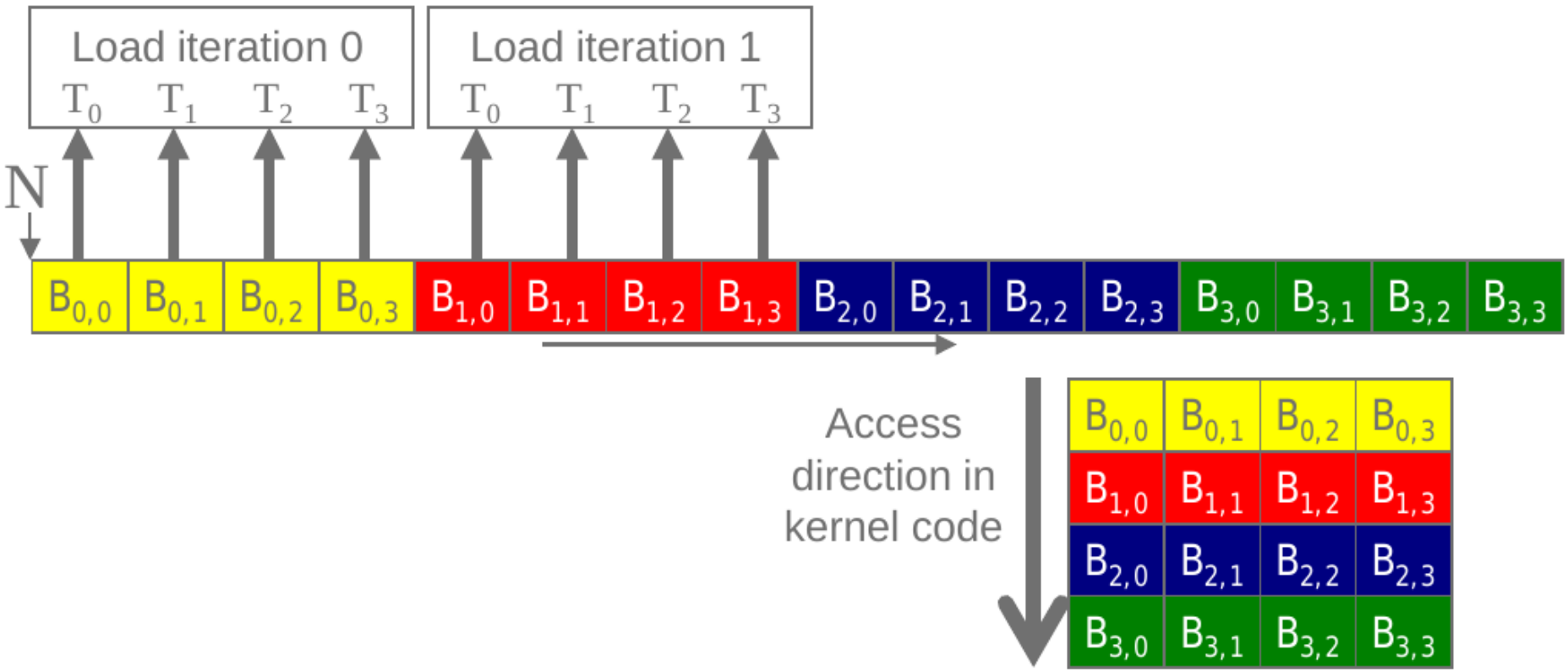


A[Row*n+i]

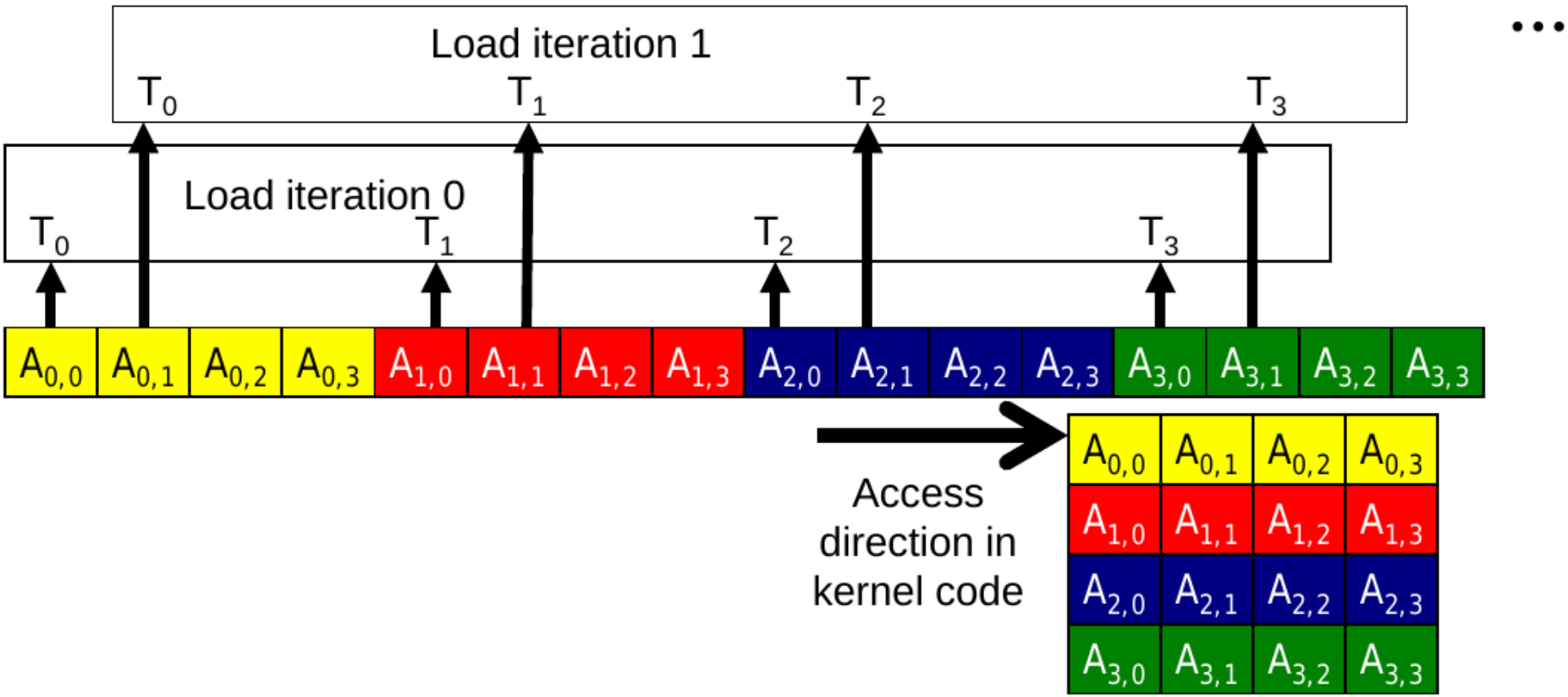
B[i*k+Col]

- *i* est le compteur de boucle dans la boucle de produit interne du code noyau
- *A* est de taille m x n
- *B* est de taille n x k
- $Col = blockIdx.x * blockDim.x + threadIdx.x$

Les accés à B sont groupés



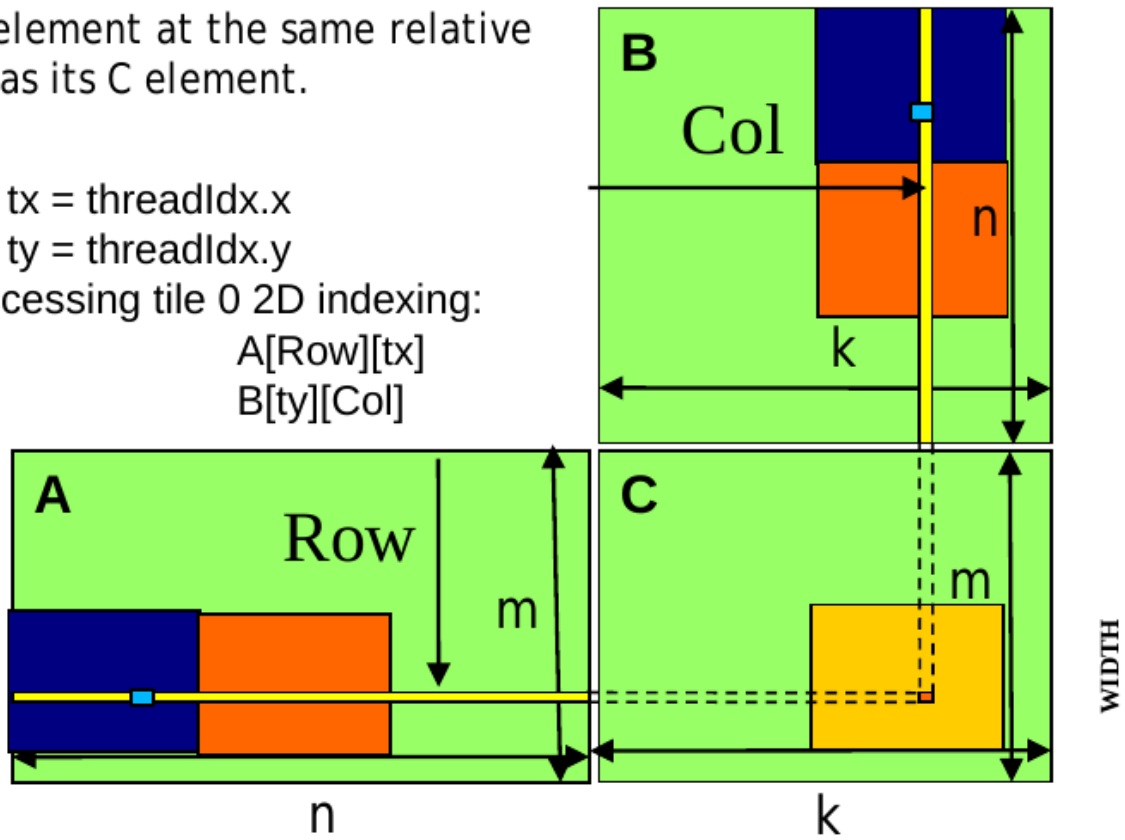
Les accés à 1 NE sont PAS groupés



Chargement d'un carreau en entrée

Have each thread load an A element and a B element at the same relative position as its C element.

int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
A[Row][tx]
B[ty][Col]



Optimisation des coins