Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

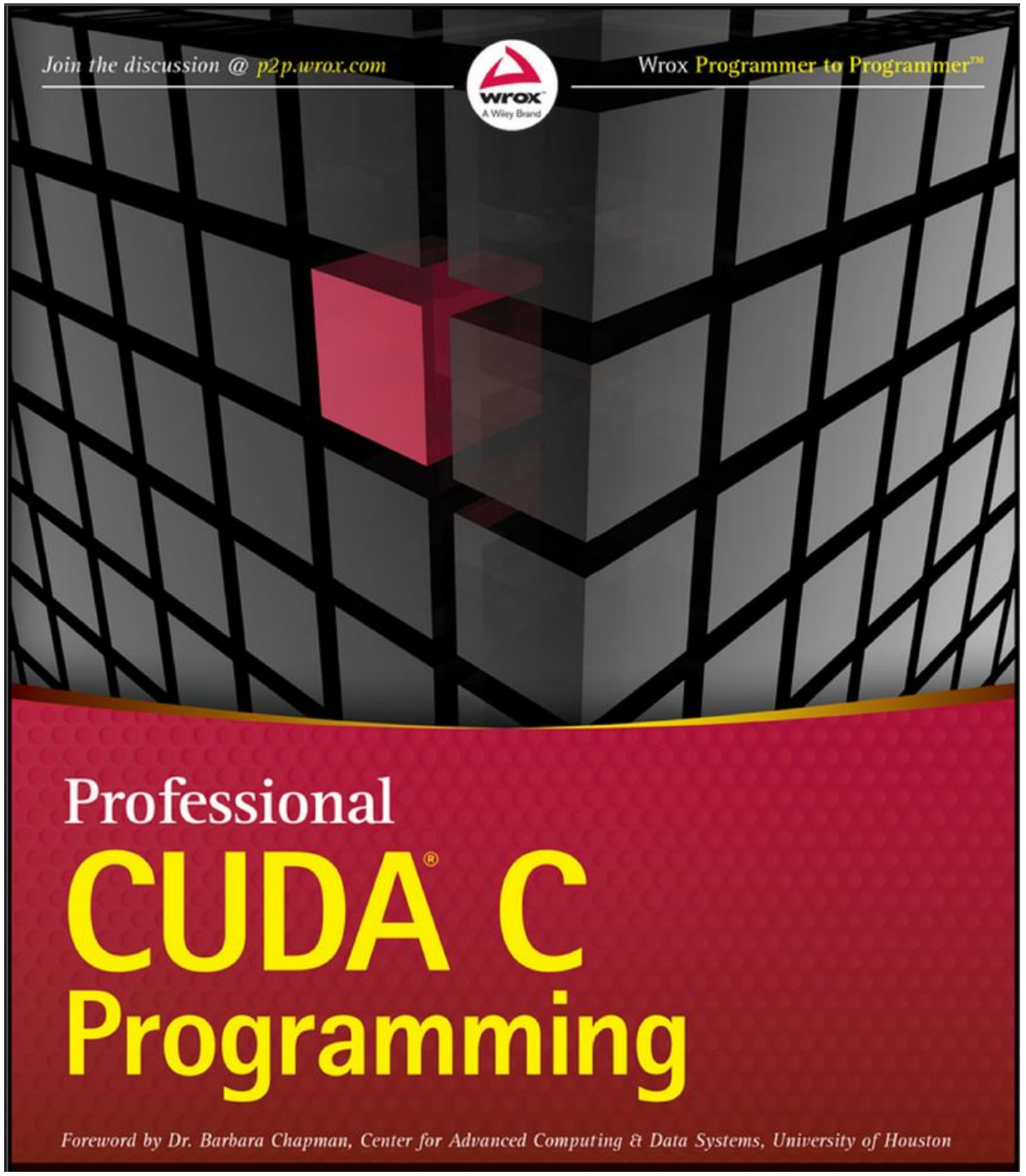
School of Electrical Engineering
and Computer Science

CEG 4536 Architecture des ordinateurs III

Automne 2024

Professor: Mohamed Ali Ibrahim, ing., Ph.D.

Source:


Join the discussion @ p2p.wrox.com
wrox
A Wiley Brand
Wrox Programmer to Programmer™
Professional
CUDA® C
Programming
Foreword by Dr. Barbara Chapman, Center for Advanced Computing & Data Systems, University of Houston

Chapitre 5 : Mémoire partagée et mémoire constante

Plan

- Apprendre comment les données sont organisées dans la mémoire partagée
- Maîtriser la conversion d'index de la mémoire partagée 2D à la mémoire globale linéaire
- Résolution des conflits de banques pour différents modes d'accès
- Mise en cache des données dans la mémoire partagée pour réduire les accès à la mémoire globale
- Éviter l'accès non coalisé à la mémoire globale en utilisant la mémoire partagée
- Comprendre la différence entre le cache constant et lecache en lecture seule
- Programmation avec l'instruction de brassage des chaînes

Mémoire partagée CUDA

- **Types de mémoire GPU** :
  - **Mémoire embarquée** (mémoire globale) : Grande, haute latence
  - **Mémoire sur puce** (mémoire partagée) : Petite, faible latence, grande largeur de bande

- **Utilisation de la mémoire partagée** :
  - **Communication intra-bloc** : Permet aux threads au sein d'un bloc de communiquer efficacement.
  - **Cache géré par le programme** : Agit comme un cache pour les données de la mémoire globale
  - **Mémoire de type "Scratchpad"** : Optimise les transformations de données pour améliorer l'accès à la mémoire globale
- **Exemples pratiques** :
  - **Noyau de réduction**
  - **Noyau de transposition de la matrice**

Mémoire partagée dans CUDA

- **Qu'est-ce que la mémoire partagée ?**
  - La mémoire partagée (SMEM) est un pool de mémoire sur puce à faible latence dans chaque multiprocesseur de flux (SM).
  - Partagé entre les threads au sein d'un bloc de threads, ce qui permet de réutiliser les données et d'accélérer la communication à l'intérieur du bloc.
  - Agit comme un **cache géré par le programme**, permettant un contrôle explicite sur le stockage des données.

- **Hiérarchie de la mémoire**
  - **Mémoire globale** : Temps de latence élevé, faible largeur de bande
  - **Mémoire partagée** : Faible latence, grande largeur de bande (latence 20 à 30 fois inférieure à celle de la mémoire globale).

Architecture de mémoire partagée (FIGURE 5-1)

- **Composants SM** :
  - **SMEM**(Mémoire partagée)
  - **Cache L1**
  - **Cache en lecture seule et cache constant**
- **Accès aux données** :
  - La mémoire partagée est allouée par bloc de threads et reste active pendant toute la durée de vie du bloc.
  - Efficace pour la réutilisation des données entre les threads au sein d'un bloc, ce qui réduit l'utilisation de la bande passante de la mémoire globale.

Architecture de mémoire partagée (FIGURE 5-1)

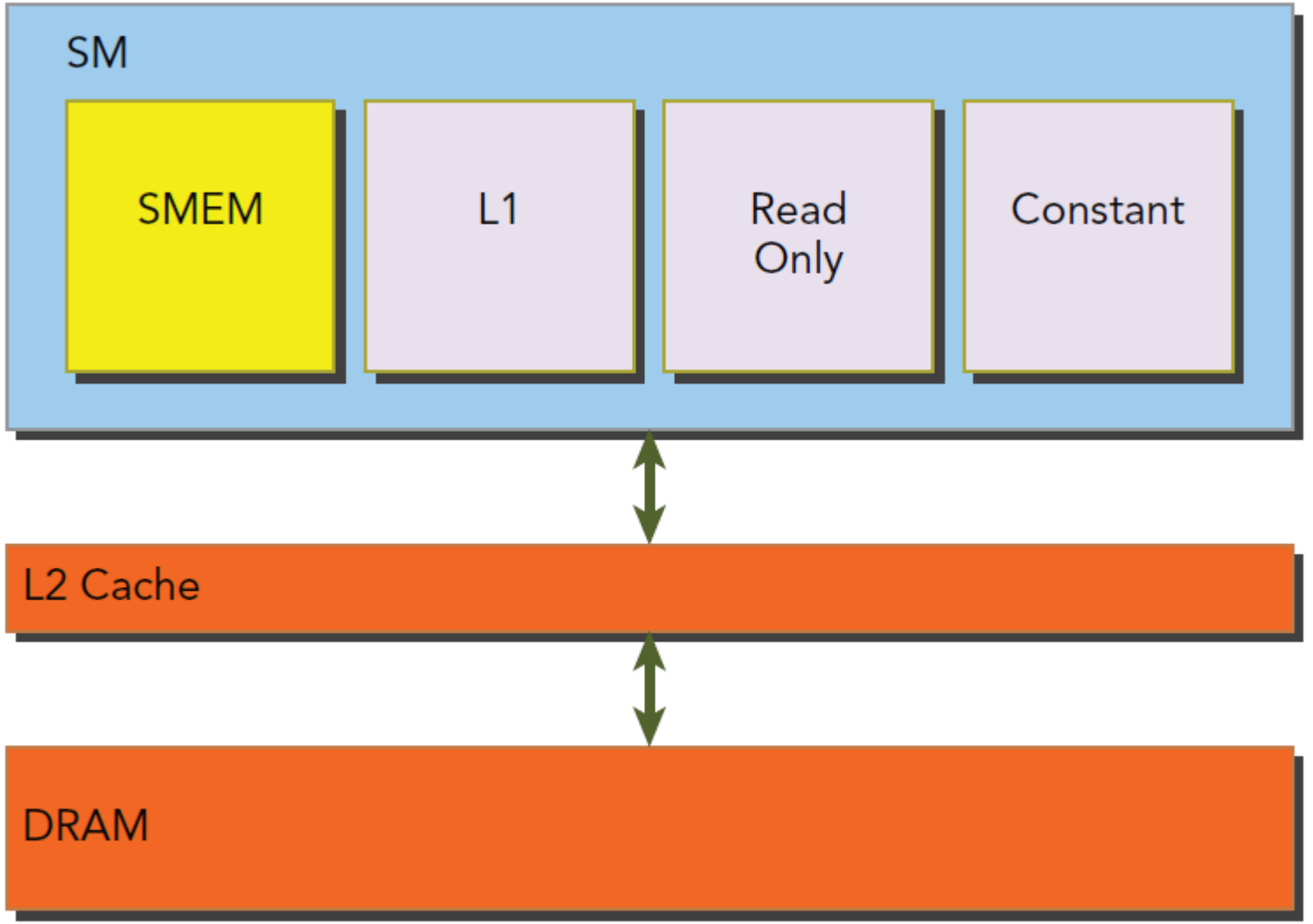


FIGURE 5-1

Cache géré par programme

- **Optimisations** :
  - Permet de **contrôler manuellement** le flux de données entre la mémoire globale et la mémoire partagée, améliorant ainsi la localité du cache.
  - Efficace pour les transformations de boucles afin d'augmenter la localité spatiale, idéal pour la programmation CUDA.
- **Avantages** :
  - Contrôle fin pour les développeurs afin d'optimiser les performances.
  - Permet de réduire les temps d'accès à la mémoire en améliorant le mouvement des données sur la puce et en réduisant les transactions hors puce.

Allocation statique de la mémoire partagée

- **Déclaration de la mémoire partagée** :
  - Utilisez le qualificatif `__shared__` pour déclarer une mémoire partagée.
  - La mémoire partagée peut être constituée de **tableaux 1D, 2D ou 3D**.

- **Allocation statique** :
  - - Seuls les tableaux **1D** peuvent être alloués dynamiquement dans la mémoire partagée.
  - Les tableaux de taille connue peuvent être déclarés statiquement dans les noyaux.
  - Exemple :

    ```
    float tile[size_y][size_x] ;
    ```

  - Si elle est déclarée à l'intérieur d'un noyau, la portée est locale au noyau ; à l'extérieur, elle est globale à tous les noyaux.

Allocation dynamique de la mémoire partagée

- **Allocation dynamique** :
  - Pour les tailles inconnues au moment de la compilation, déclarer comme :

```
extern __shared__ int tile[] ;
```

  - Doit spécifier la taille au lancement du noyau comme troisième argument (en octets).

```
kernel<<<grid, block, isize * sizeof(int)>>>(...) ;
```

- **Remarque** :
  - Seuls les **tableaux 1D** peuvent être alloués dynamiquement dans la mémoire partagée.

Banques de mémoire partagée et mode d'accès

- **Optimisation des performances de la mémoire** :
  - Deux propriétés essentielles : **La latence** et la **bande passante**
  - La mémoire partagée est divisée en 32 banques pour maximiser la bande passante.
- **Banques de mémoire** :
  - Chaque banque peut être accédée simultanément par différents threads.
  - Des schémas d'accès appropriés améliorent l'utilisation de la bande passante.

Conflits entre les banques (voir les figures 5-2, 5-3 et 5-4)

- **Conflit bancaire** :
  - Se produit lorsque plusieurs threads d'une chaîne accèdent à la même banque.
- Types d'accès :
  - **Accès parallèle** : Pas de conflits ; accès à plusieurs banques.
  - **Accès en série** : Temps de latence élevé ; accès au sein de la même banque.
  - **Accès par diffusion** : Tous les threads lisent la même adresse dans une banque.

La figure 5-2 illustre le schéma d'accès parallèle optimal.

Chaque thread accède à un mot de 32 bits :

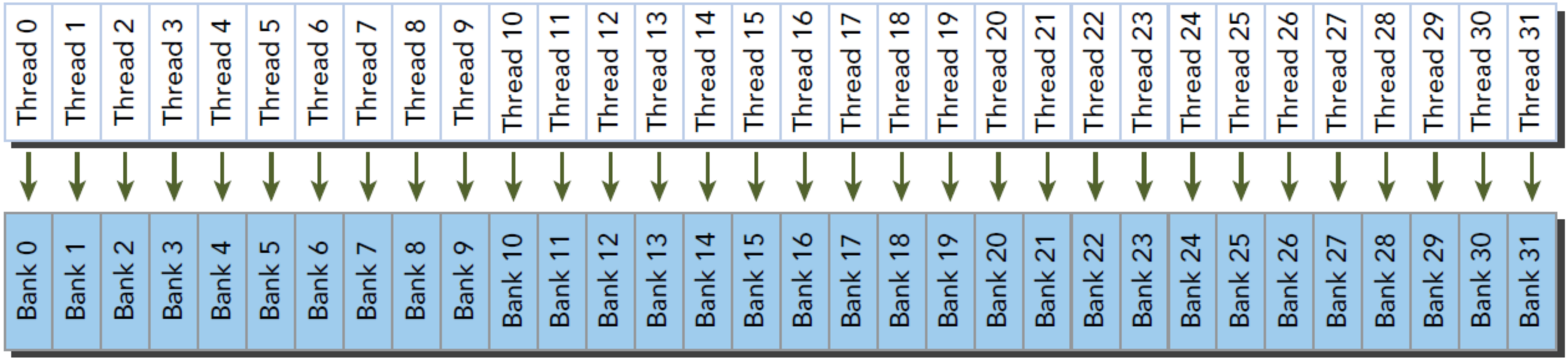


FIGURE 5-2

La figure 5-3 illustre un modèle d'accès irrégulier et aléatoire

Il n'y a toujours pas de conflit de banque, car chaque thread accède à une banque différente :

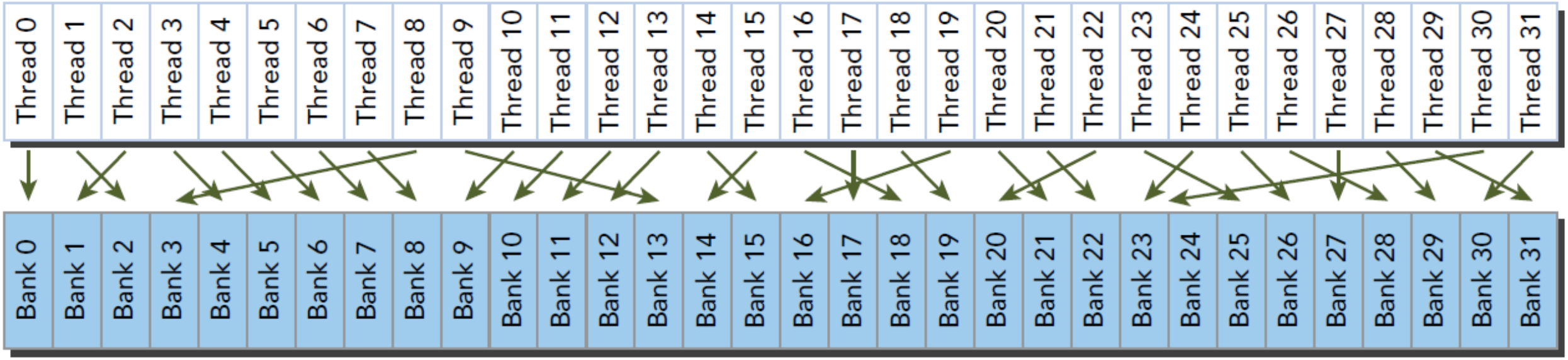


FIGURE 5-3

La figure 5-4 illustre un autre schéma d'accès irrégulier dans lequel plusieurs threads accèdent à la même banque

Il y a deux comportements possibles pour une telle demande :

- Accès par diffusion sans conflit si les fils accèdent à la même adresse au sein d'une banque
- Conflit d'accès à la banque si les threads accèdent à des adresses différentes au sein d'une banque

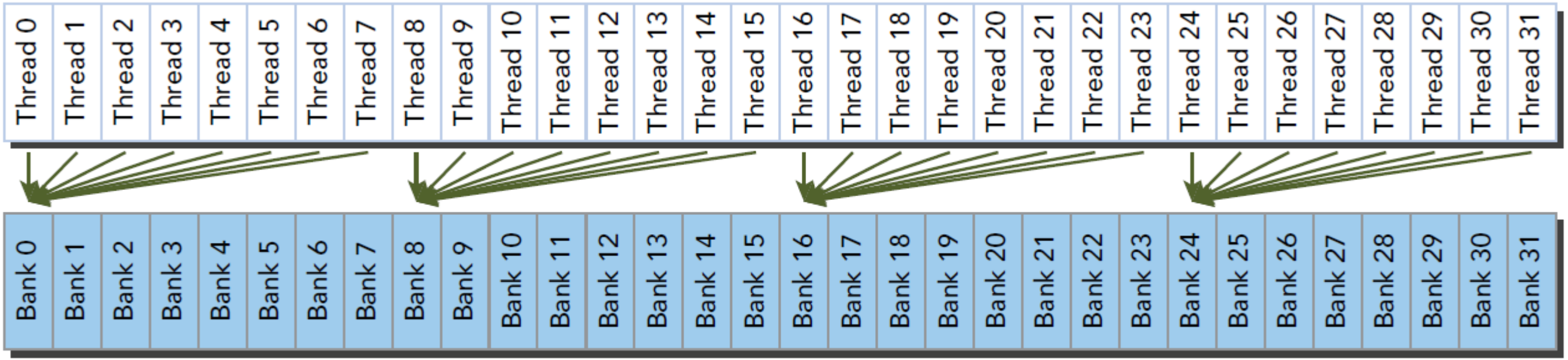


FIGURE 5-4

Mode d'accès et calcul de l'index de banque (voir figure 5-5)

- **Mode d'accès** :
  - Les appareils peuvent fonctionner en mode 32 bits ou 64 bits, ce qui influe sur la largeur des banques.
  - Calcul de l'indice bancaire :

    index de la banque = (adresse de l'octet/largeur de la banque) % 32

- **Fermi contre Kepler** :
  - Kepler dispose d'un mode 64 bits, ce qui réduit les conflits potentiels.

Distribution des banques de mémoire dans la mémoire partagée (Figure 5-5)

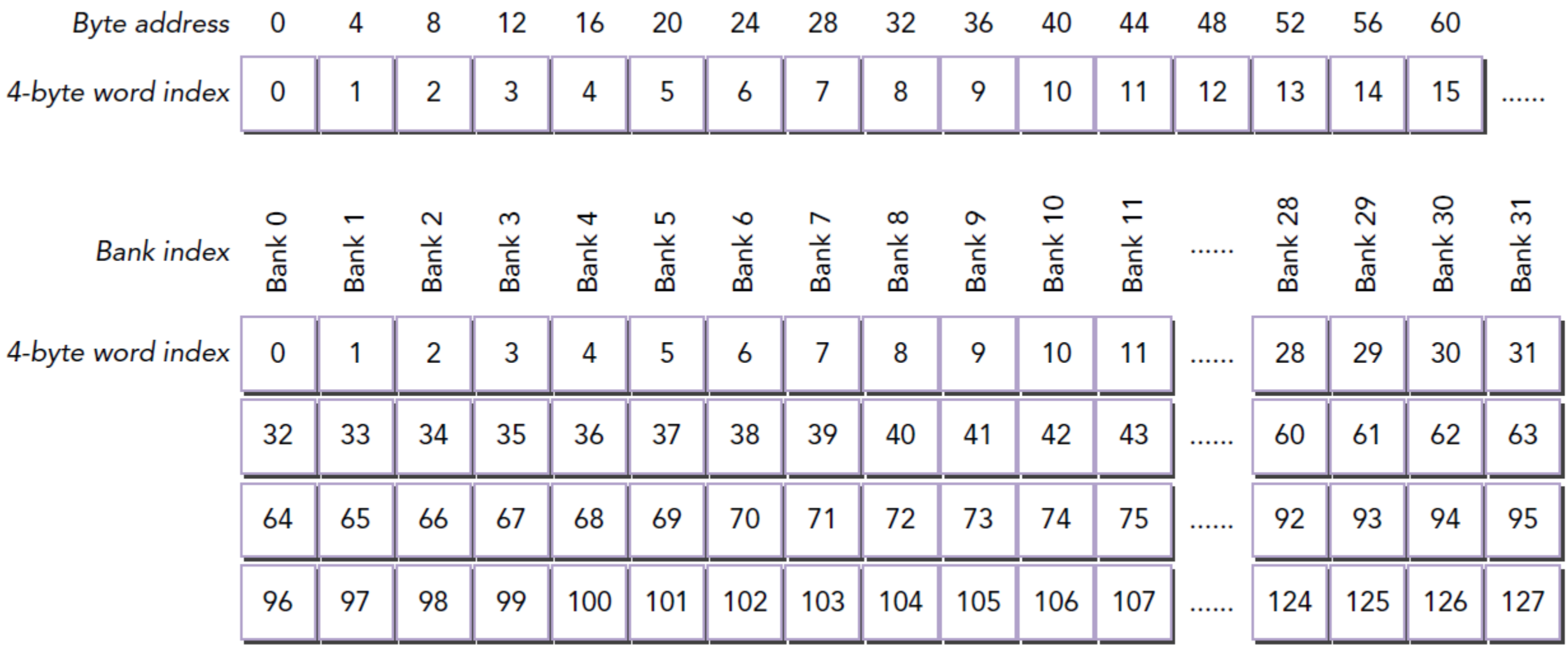


FIGURE 5-5

La figure 5-6 illustre la correspondance entre l'adresse de l'octet et l'index de banque pour le mode 32 bits.

- La figure du haut représente la mémoire partagée étiquetée avec des adresses d'octets et des indices de mots de 4 octets.
- La figure du bas montre la correspondance entre les indices de mots de 4 octets et les indices de banques.
- Bien que le mot 0 et le mot 32 se trouvent tous deux dans la banque 0, la lecture des deux dans la même demande de mémoire n'impliquerait pas de conflit de banque.

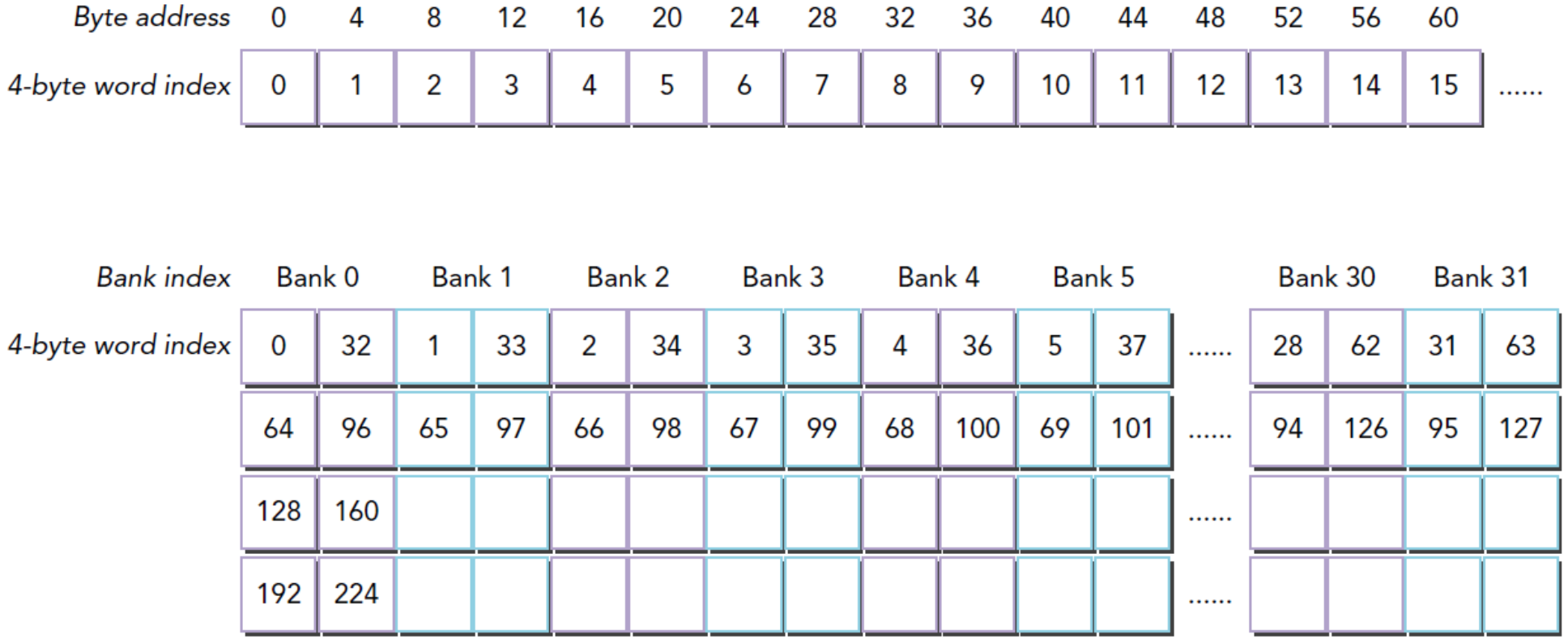


FIGURE 5-6

La figure 5-7 illustre un cas d'accès sans conflit en mode 64 bits, où chaque thread accède à des banques différentes.

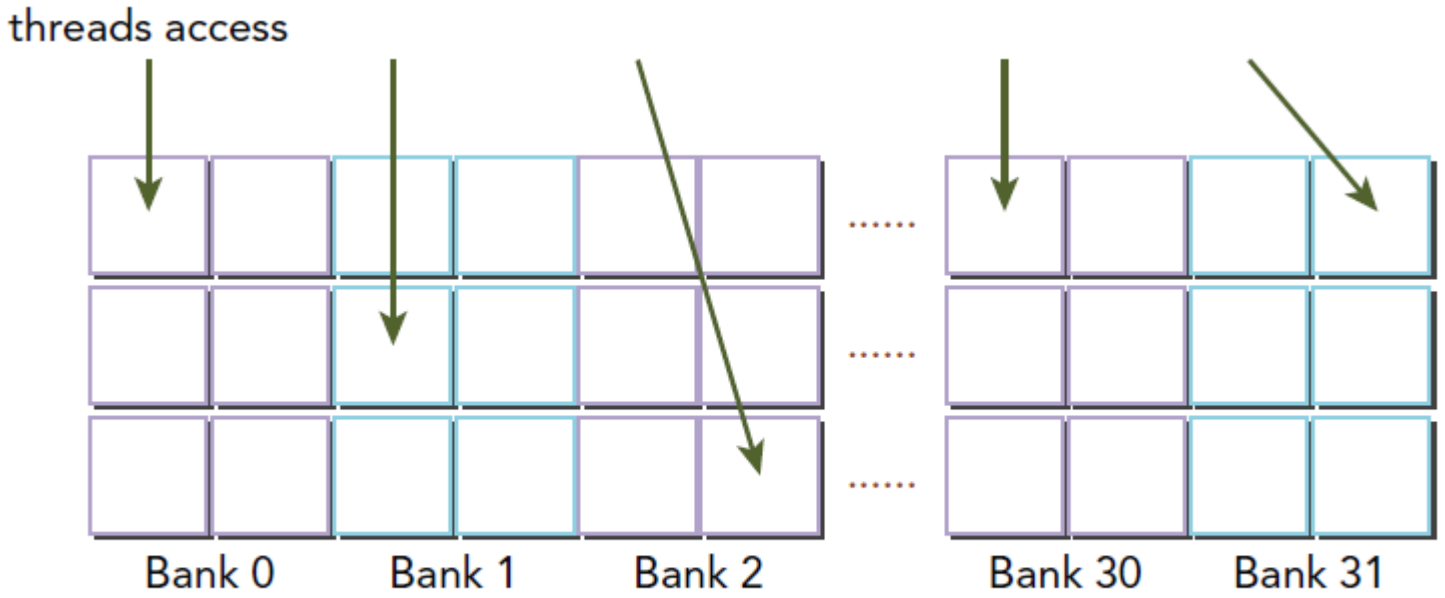


FIGURE 5-7

La figure 5-8 illustre un autre cas d'accès sans conflit en mode 64 bits, où deux threads accèdent à des mots dans la même banque et dans le même mot de 8 octets

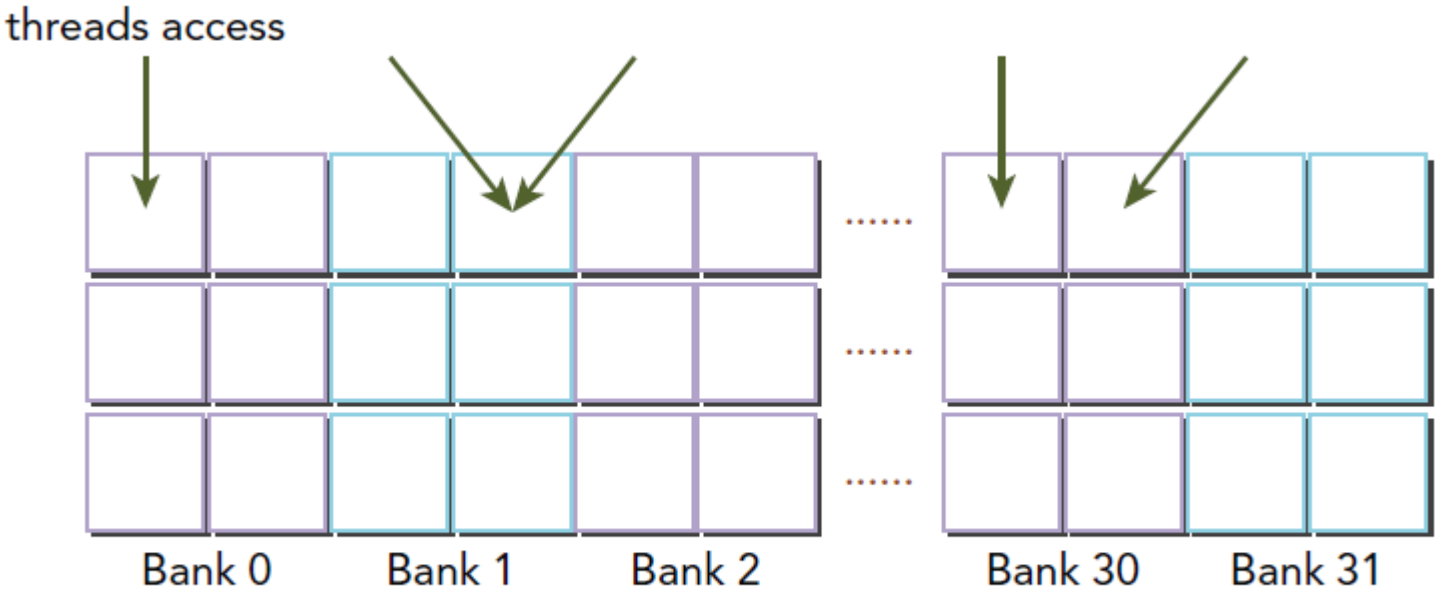


FIGURE 5-8

La figure 5-9 illustre un conflit de banque bidirectionnel dans lequel deux threads accèdent à la même banque, mais dont les adresses se situent dans deux mots de 8 octets différents

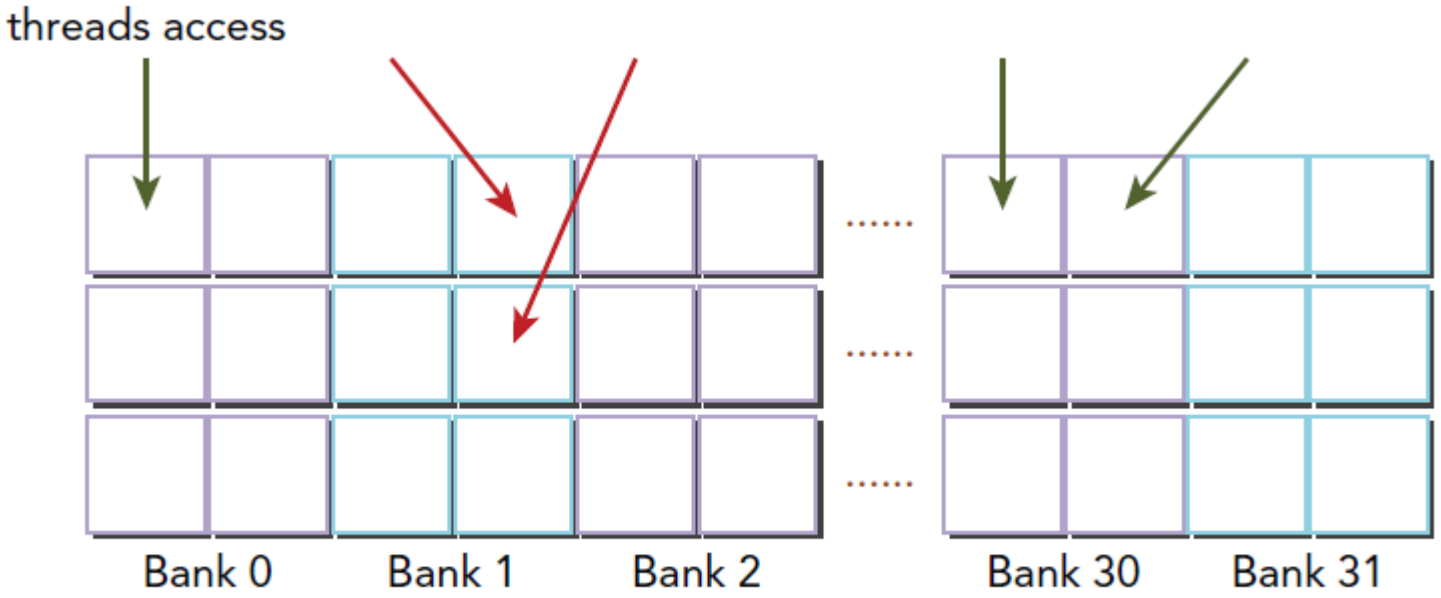


FIGURE 5-9

La figure 5-10 illustre un conflit de banque à trois voies, où trois threads accèdent à la même banque et où les adresses se trouvent dans trois mots de 8 octets différents.

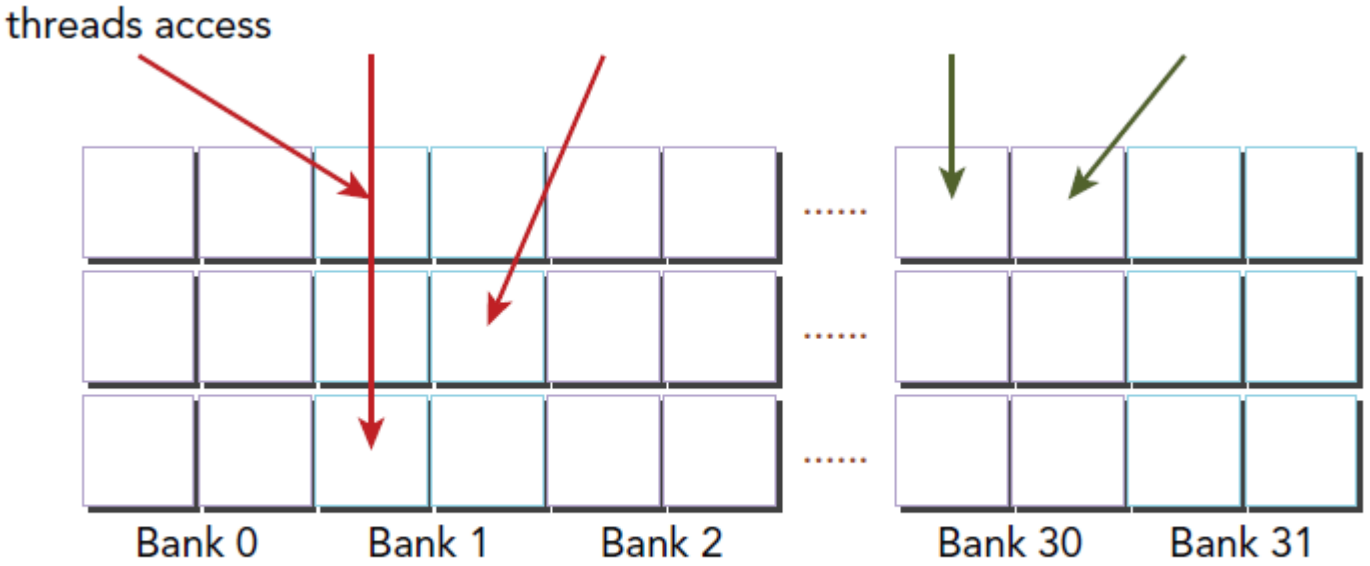


FIGURE 5-10

Rembourrage de la mémoire (voir figure 5-11)

- **Rembourrage de la mémoire** :
  - Ajoute de l'espace supplémentaire pour éviter les conflits bancaires.
  - Le remplissage répartit les données entre plusieurs banques, ce qui améliore l'efficacité de l'accès.
- **Mise en œuvre** :
  - Exemple : Ajouter un mot après chaque `N` éléments pour éviter les conflits.

La figure 5-11 illustre le remplissage de la mémoire dans un cas simple.

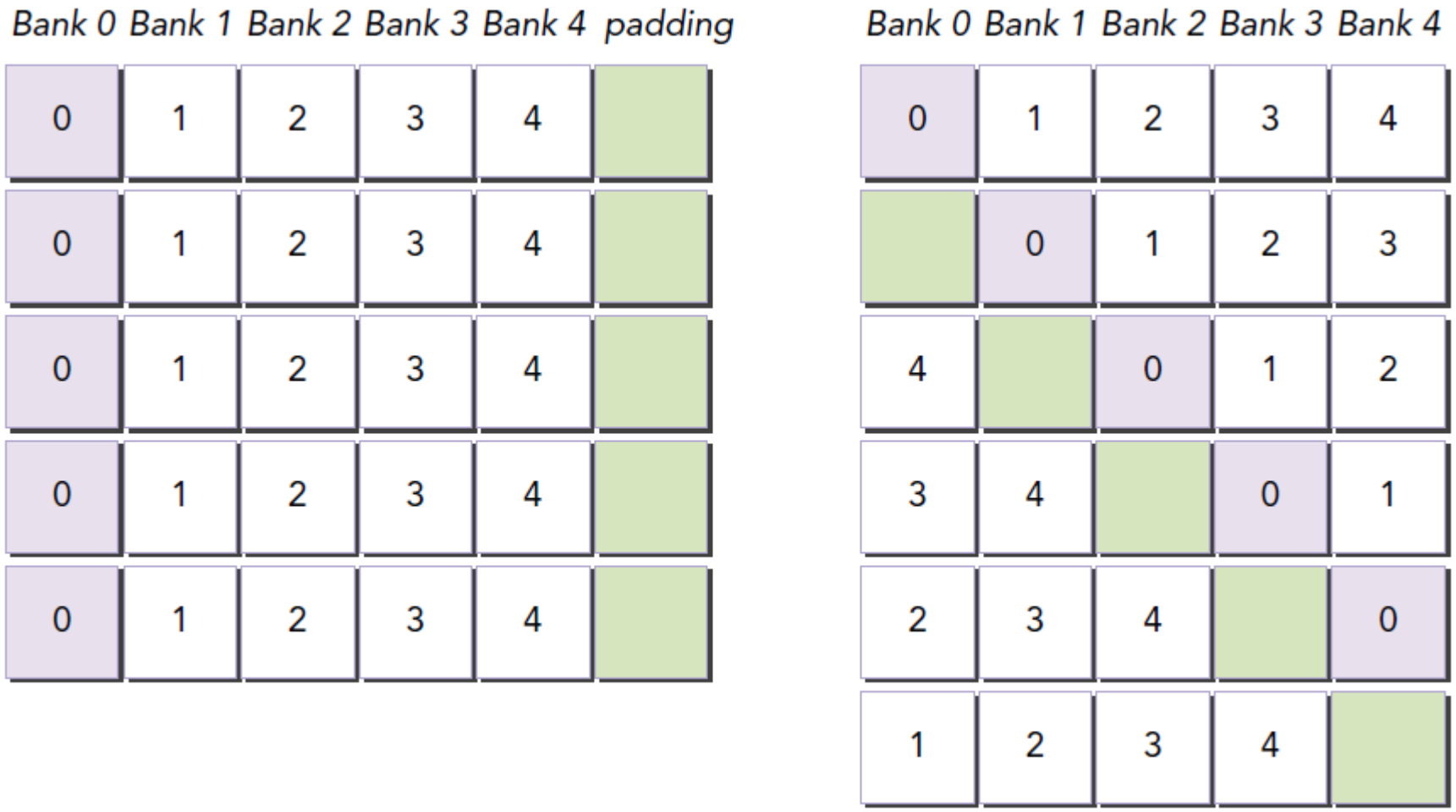


FIGURE 5-11

Configuration du mode d'accès à Kepler

- **Configuration du mode d'accès** :
  - **Dispositifs Kepler** : Prise en charge des modes 4 et 8 octets.
  - Utilisez les fonctions de l'API CUDA pour demander ou définir des configurations :

```
cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte) ;
```

- **Impact sur les performances** :
  - Les configurations ont une incidence sur la largeur de bande, l'occupation et les conflits potentiels entre banques.

Configuration de la mémoire partagée dans CUDA

- **Mémoire CUDA sur puce** : Chaque multiprocesseur de streaming (SM) dispose de 64 Ko de mémoire sur puce, partagée entre le cache L1 et la mémoire partagée.
- **Options de configuration** :
  - **Configuration par appareil** : S'applique à tous les noyaux de l'appareil.
  - **Configuration par noyau** : Spécifique à chaque noyau.
- **Fonction de configuration** :
  - `cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig) ;`
  - **Configurations du cache** :
    - `cudaFuncCachePreferNone : Pas de préférence (par défaut).`
    - `cudaFuncCachePreferShared : 48 Ko de mémoire partagée, 16 Ko de cache L1.`
    - `cudaFuncCachePreferL1 : 16 Ko de mémoire partagée, 48 Ko de cache L1.`
    - `cudaFuncCachePreferEqual : 32 KB pour la mémoire partagée et le cache L1.`

Choisir la bonne configuration de cache

- **Recommandations d'utilisation** :
  - **Plus de mémoire partagée** : Préférer la mémoire partagée lorsque les noyaux ont besoin d'un taux d'occupation élevé.
  - **Plus de cache L1** : Préférer le cache L1 pour les noyaux utilisant plus de registres afin d'éviter les débordements de registres.

- **Considérations** :
  - **Cache du GPU contre cache du CPU** :
    - Le comportement du cache du GPU est différent en raison de la concurrence des threads.
    - La mémoire partagée permet un contrôle explicite de la localité des données à un SM.
  - **Conseil de performance** : Une grande mémoire cache L1 peut être utile aux noyaux qui utilisent beaucoup de registres ou qui ont besoin de mémoire locale.

Synchronisation dans CUDA

- **Besoin de synchronisation** : Garantit que la mémoire partagée à laquelle accèdent plusieurs threads reste cohérente.
  - **Synchronisation intra-bloc** : Réalisée à l'aide de **barrières** et de **clôtures de mémoire**.
- **Mécanismes de synchronisation** :
  - **Barrières** : Les threads s'attendent les uns les autres à un point d'arrêt. Exemple : __syncthreads() `__syncthreads()`
  - **Clôtures de mémoire** : Contrôlez la visibilité des écritures en mémoire entre les threads, avec des portées de bloc, de grille ou de système.

Modèle de mémoire faiblement ordonnée et barrières explicites

- Mémoire faiblement ordonnée :
  - Les accès à la mémoire peuvent se faire dans le désordre afin d'optimiser les performances.
  - Utiliser des **clôtures de mémoire** et des **barrières** pour faire respecter l'ordre si nécessaire.

- **Barrière explicite** (`__syncthreads()`):
  - Assure que tous les threads d'un bloc atteignent le même point avant de continuer.
  - Doit être utilisé avec précaution pour éviter un comportement indéfini (par exemple, dans un code conditionnel).

Barrière de mémoire et qualificatif volatile

- **Les clôtures de la mémoire** :
  - Assurer la visibilité de la mémoire après une écriture.
  - Types :
    - `__threadfence_block()` : A l'intérieur d'un bloc.
    - `__threadfence()` : A l'intérieur d'une grille.
    - `__threadfence_system()` : Sur tous les appareils et l'hôte.

- **Qualificatif volatile** :
  - Empêche l'optimisation du compilateur et garantit un accès direct à la mémoire.
  - Utile pour les variables partagées/globales qui peuvent changer de manière inattendue.

Optimisation de l'agencement des données de la mémoire partagée dans CUDA

- **Considérations clés** :
  - **Forme du réseau** : Matrices carrées ou rectangulaires
  - **Modèle d'accès** : Majeur de ligne ou majeur de colonne
  - **Déclarations de mémoire** : Mémoire partagée statique ou dynamique
  - **Champ d'application** : Mémoire partagée à l'échelle du fichier ou à l'échelle du noyau
  - **Remboursement de la mémoire** : Avec ou sans rembourrage pour éviter les conflits entre banques

- **Focus sur la conception** :
  - **Cartographie des banques** : Mappage efficace des données entre les banques de mémoire
  - **Mappage d'index** : Aligner les index des threads sur les décalages de la mémoire partagée

- **Objectif** : réduire les conflits entre banques pour maximiser les avantages de la mémoire partagée.

Optimisation des schémas d'accès à la mémoire partagée carrée (Figure 5-12)

- **Concept** : La figure 5-12 illustre la mise en cache de données globales dans la mémoire partagée à l'aide d'une disposition carrée, ce qui facilite les calculs efficaces de décalage de mémoire 1D pour les indices de threads 2D.
- **Disposition de la mémoire** :
  - Le diagramme du haut montre la disposition des données 1D.
  - Le diagramme du bas présente une vue logique en 2D avec des éléments de 4 octets répartis sur les banques de mémoire.
- **Modèles d'accès** :
  - **Optimal** : `tile[threadIdx.y][threadIdx.x]`
  - **Sous-optimal** : `` `tile[threadIdx.x][threadIdx.y] ``
- **Recommandation** :
  - L'accès en tant que `tuile [threadIdx.y][threadIdx.x]` réduit les **conflits de banque** en alignant les threads voisins d'une chaîne sur des banques de mémoire distinctes, ce qui optimise les performances.

Optimisation des schémas d'accès à la mémoire partagée carrée (Figure 5-12)

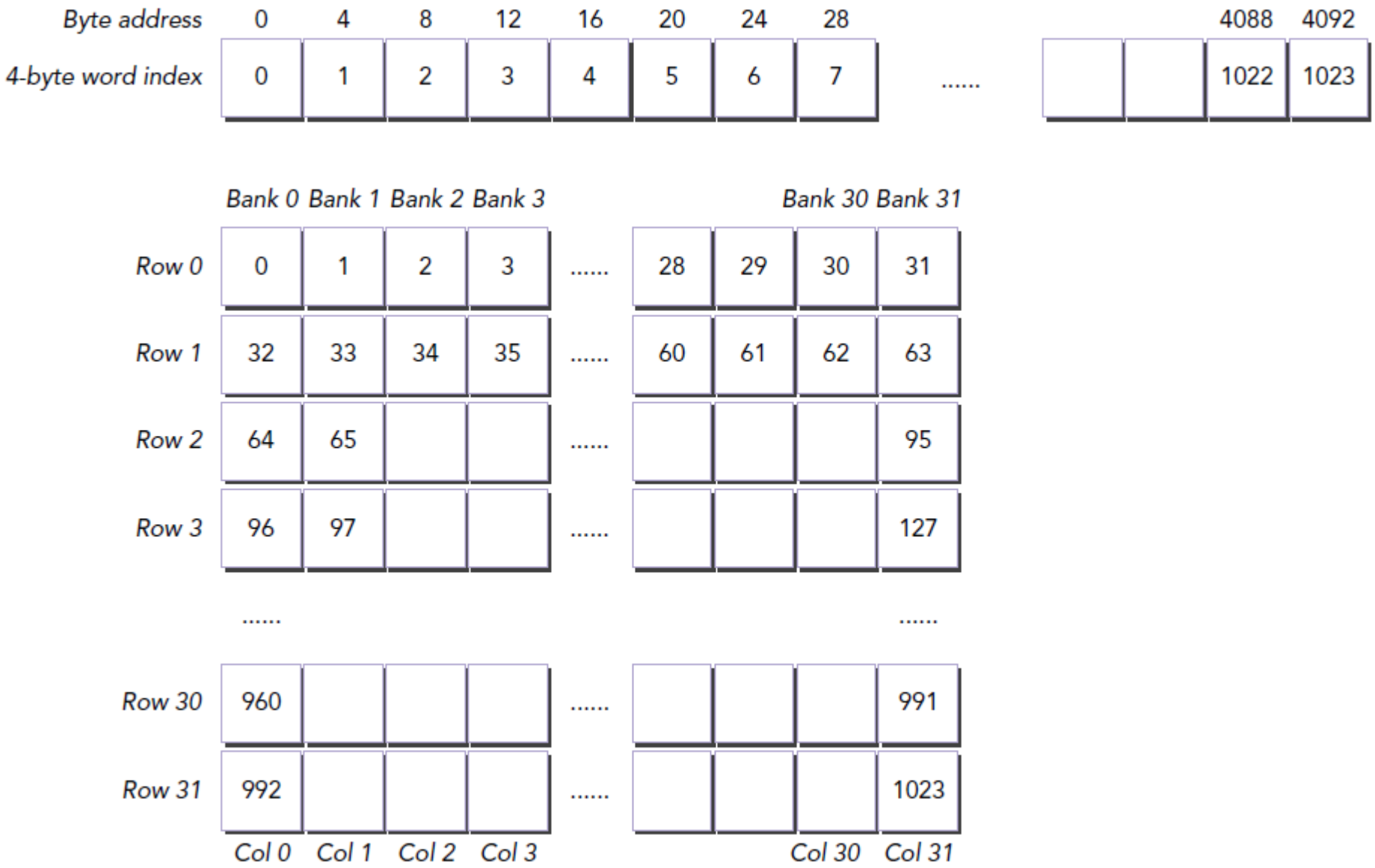


FIGURE 5-12

Accès aux lignes majeures et aux colonnes majeures dans CUDA

- **Énoncé du problème** : Accéder efficacement à la mémoire partagée dans CUDA.
- **Configuration** :
  - Dimensions du bloc : `BDIMX = 32, BDIMY = 32`
  - Configuration d'exécution : `bloc dim3 (BDIMX, BDIMY) ; grille dim3 (1,1) ;`
- **Opérations** :
  - Écrire les index globaux des threads dans un tableau de mémoire partagée en 2D dans l'ordre des rangées majeures.
  - Lire les valeurs de la mémoire partagée dans l'ordre des lignes pour les stocker dans la mémoire globale.

Code du noyau et opérations sur la mémoire

- **Code du noyau pour l'accès aux lignes** (`setRowReadRow`) :
  - Permet de faire correspondre les index des threads à la mémoire globale.
  - L'écriture et la lecture de la mémoire partagée s'effectuent dans l'ordre des lignes majeures, ce qui minimise les conflits entre les banques.
- **Code du noyau pour l'accès aux colonnes (**`setColReadCol`**) :**
  - Permet de faire correspondre les index des threads à la mémoire globale.
  - L'accès dans l'ordre des colonnes majeures peut entraîner des conflits de banques.
- **Opérations de mémoire** :
  - Opération de stockage en mémoire partagée
  - Opération de chargement de la mémoire partagée
  - Opération de mémorisation globale

Analyse des performances à l'aide de `nvprof`

- **Résultats des tests de performance (Tesla K40c)** :
  - L'accès à la rangée majeure (`setRowReadRow`) offre de meilleures performances en évitant les conflits entre banques.
  - L'accès à la colonne principale (`setColReadCol`) entraîne des conflits entre les banques à 32 et 16 voies.
- **Conflits entre banques (à l'aide de `nvprof`)**
  - `setRowReadRow` : une seule transaction par demande, pas de conflit.
  - `setColReadCol` : 16 transactions par demande, confirmant les conflits bancaires.
- **Conclusion** : L'ordre ligne-majeur améliore les performances d'accès à la mémoire partagée en réduisant les conflits entre banques.

Écriture d'une ligne majeure et lecture d'une colonne majeure dans CUDA

- **Objectif du noyau** : Gérer efficacement la mémoire partagée avec des écritures majeures en ligne et des lectures majeures en colonne.
- **Détails de la mise en œuvre** :
  - **Écriture dans la mémoire partagée dans l'ordre des rangées principales** :
    - Code : `tile[threadIdx.y][threadIdx.x] = idx ;`
  - **Lecture de la mémoire partagée dans l'ordre colonne-majeur** :
    - Code : `out[idx] = tile[threadIdx.x][threadIdx.y] ;`
- **Concept clé** :
  - L'utilisation de l'ordre majeur de rangée pour l'écriture permet de s'aligner sur la structure des banques de la mémoire partagée, alors que la lecture majeure de colonne peut introduire des conflits.

Écriture d'une ligne majeure et lecture d'une colonne majeure dans CUDA

- **Objectif du noyau** : Gérer efficacement la mémoire partagée avec des écritures majeures en ligne et des lectures majeures en colonne.
- **Détails de la mise en œuvre** :
  - **Écriture dans la mémoire partagée dans l'ordre des rangées principales** :
    - Code : `tile[threadIdx.y][threadIdx.x] = idx ;`
  - **Lecture de la mémoire partagée dans l'ordre colonne-majeur** :
    - Code : `out[idx] = tile[threadIdx.x][threadIdx.y] ;`
- **Concept clé** :
  - L'utilisation de l'ordre majeur par rangée pour l'écriture permet de s'aligner sur la structure des banques de la mémoire partagée, alors que la lecture majeure par colonne peut introduire des conflits.

Analyse des performances et Figure 5-13

- **Illustration (figure 5-13) :**
  - Représentation d'une configuration de mémoire partagée à cinq banques.
  - **Position d'écriture** : `(iy, ix)` - Accès majeur à la rangée, sans conflit.
  - **Position de lecture** : `(ix, iy)` - Colonne - accès principal, risque de conflits avec les banques.
- **Analyse des performances (résultats `nvprof`) :**
- **Mesures** :
  - `shared_load_transactions_per_request = 16` (conflit de banque à 16 voies pendant la lecture).
  - `shared_store_transactions_per_request = 1` (pas de conflit pendant l'écriture).
- **Conclusion** :
  - L'écriture en rangée majeure est efficace, tandis que la lecture en colonne majeure introduit un conflit de banque à 16 voies, ce qui a un impact sur les performances.

La figure 5-13 illustre les deux opérations de mémoire à l'aide d'une implémentation simplifiée de la mémoire partagée à cinq banques.

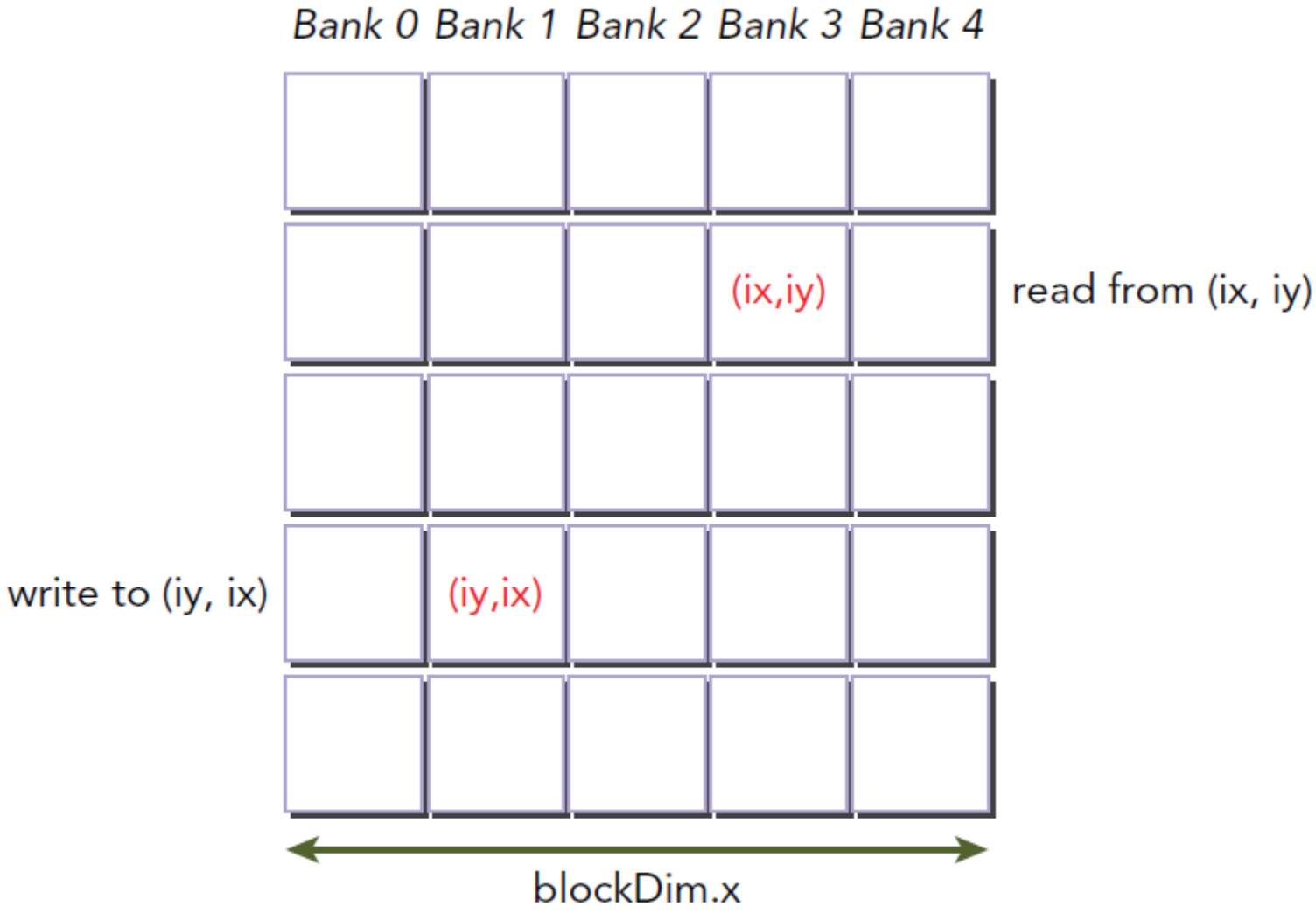


FIGURE 5-13

Mémoire partagée dynamique dans CUDA

- **Concept** : La mémoire partagée dynamique permet une certaine flexibilité dans l'allocation de la mémoire pour les noyaux.
- **Utilisation** :
  - Déclarer la mémoire partagée dynamique comme un tableau 1D non dimensionné.
  - Calculer les indices d'accès à la mémoire à partir des indices de threads 2D pour les ordres majeurs en ligne et en colonne.
- **Calcul de l'indice** :
  - `row_idx` : décalage de la mémoire de la rangée majeure 1D à partir des indices des threads 2D.
  - `col_idx` : décalage de la mémoire principale de la colonne 1D par rapport aux indices des fils 2D.
- **Opérations** :
  - Écriture dans la mémoire partagée dans l'ordre majeur des rangées : `tile[row_idx] = row_idx ;`
  - Lecture de la mémoire partagée dans l'ordre des colonnes et affectation à la mémoire globale :

```
out[row_idx] = tile[col_idx] ;
```

Mise en œuvre du noyau et analyse des performances

- **Code du noyau (`setRowReadColDyn`) :**
  - Déclarer la mémoire partagée dynamique : `extern __shared__ int tile[] ;`
  - - Écrire dans l'ordre des lignes en utilisant `` `row_idx` ``, lire dans l'ordre des colonnes en utilisant `col_idx`.
  - - Assurer la synchronisation avec `__syncthreads()`.
- **Lancement du noyau** :
  - Spécifiez la taille de la mémoire partagée : `setRowReadColDyn<<grid, block, BDIMX * BDIMY * sizeof(int)>>>(d_C) ;`
- **Analyse des performances (résultats `nvprof`) :**
- **Mesures** :
  - `shared_load_transactions_per_request = 16` (16-way bank conflict on load).
  - `shared_store_transactions_per_request = 1` (sans conflit sur le magasin).
- **Conclusion** : Résultats similaires à ceux de la mémoire partagée statique avec des conflits sur les lectures majeures de colonnes, mais utilisation d'une mémoire déclarée dynamiquement pour plus de flexibilité.

Remplissage de la mémoire partagée déclarée statiquement dans CUDA

- **Objectif du rembourrage** :
  - L'ajout d'un tampon aux tableaux dans la mémoire partagée permet d'éviter les conflits de banques.
  - En ajoutant une colonne supplémentaire, les éléments sont répartis sur différentes banques, ce qui permet d'éviter les conflits lors de la lecture et de l'écriture.
- **Mise en œuvre** :
  - Déclarez la mémoire partagée rembourrée : `__shared__ int tile[BDIMY][BDIMX+1] ;`
  - **Code du noyau** (`setRowReadColPad`) :
    - - Opération d'écriture : `tile[threadIdx.y][threadIdx.x] = idx ;`
    - - Opération de lecture : `out[idx] = tile[threadIdx.x][threadIdx.y] ;`
    - - Utilisez `__syncthreads()` pour la synchronisation.
- **Résultats des performances** (`nvprof`) :
  - **Mesures** :
    - `shared_load_transactions_per_request = 1.000000`
    - `transactions_du_magasin_partagé_par_requête = 1.000000`
  - **Conclusion** : Le rembourrage élimine les conflits entre banques et permet un accès efficace à la mémoire partagée pour les opérations de lecture et d'écriture.
- **Note spécifique à l'appareil** :
  - Pour les dispositifs Fermi, une colonne doit être remplie ; pour les dispositifs Kepler, les exigences de remplissage varient en fonction des éléments de données et du mode d'accès à 64 bits.

Mémoire partagée dynamique dans CUDA

- **Concept** : La mémoire partagée dynamique offre une certaine souplesse en allouant la mémoire partagée sous la forme d'un tableau 1D, ce qui lui permet de s'adapter à des besoins de mémoire variables.
- **Indexation de la mémoire** :
  - **Index de ligne** (`row_idx`) : Calcule le décalage 1D pour l'ordre majeur des lignes à partir des indices de fils 2D.
  - **Index de colonne** (`col_idx`) : Calcule le décalage 1D pour l'ordre majeur des colonnes à partir des indices de fils 2D.
- **Modèle d'accès à la mémoire** :
  - **Écriture dans l'ordre des lignes principales** : `tuile[row_idx] = row_idx ;`
  - **Lecture dans l'ordre des colonnes principales** : `out[row_idx] = tile[col_idx] ;`
- **Remarque** : la figure 5-14 illustre la disposition de la mémoire partagée dynamique et le schéma d'accès.

La figure 5-14 illustre la disposition de la mémoire partagée dynamique et le schéma d'accès.

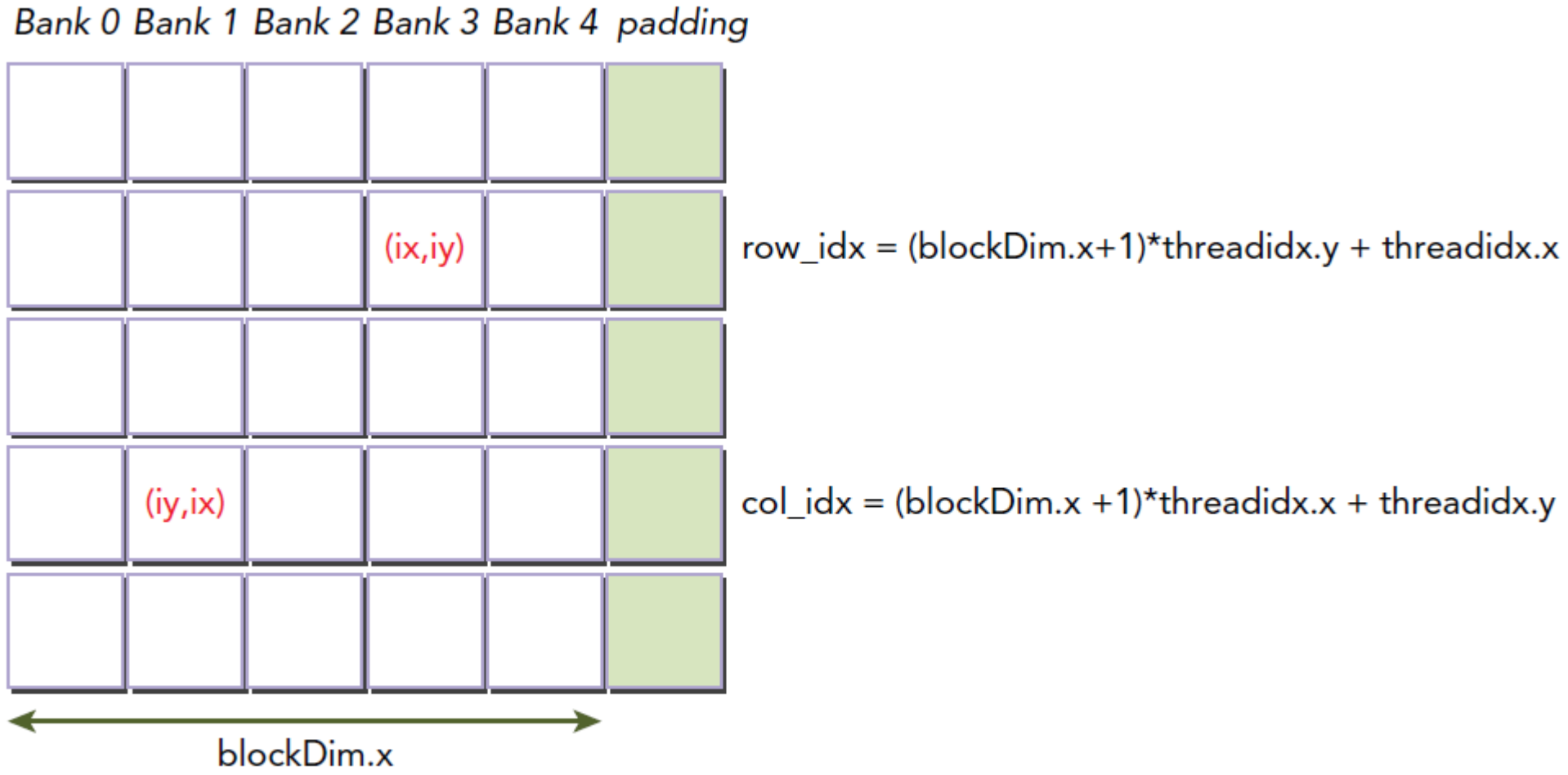


FIGURE 5-14

Mise en œuvre du noyau et analyse des performances

- **Code du noyau** (`setRowReadColDyn`) :
  - Déclarer la mémoire partagée dynamique : `extern __shared__ int tile[] ;`
  - Calculer les `valeurs row_idx` et `col_idx` pour l'écriture majeure en ligne et la lecture majeure en colonne.
  - Assurer la synchronisation en utilisant `__syncthreads()`.
- **Lancement du noyau avec la taille de mémoire spécifiée :**
  - `setRowReadColDyn<<grid, block, BDIMX * BDIMY * sizeof(int)>>>(d_C) ;`
- **Analyse des performances** (résultats `nvprof`):**
  - **Mesures** :
    - `shared_load_transactions_per_request = 16` (16-way bank conflict on load).
    - `shared_store_transactions_per_request = 1` (pas de conflit sur le magasin).
  - **Conclusion** : Comme dans le cas de la mémoire partagée statique, l'allocation dynamique introduit des conflits de banque pendant les lectures de colonnes majeures.

Comparaison des performances des noyaux à mémoire partagée carrés

- **Objectif** : Évaluer les performances des noyaux avec différentes configurations de mémoire partagée.
- **Principales conclusions** :
  - **Rembourrage** : Les noyaux utilisant le padding présentent des améliorations de performance en raison de la réduction des conflits de banque.
  - **Allocation dynamique** : La mémoire partagée déclarée dynamiquement ajoute un léger surcoût.
- **Commandement pour mesurer la performance** :

```
> nvprof smemSquare.exe
```

- **Résultats des performances (Tesla K40c)** :
  - `setColReadCol(int*)` - Moyenne : 3.6160us
  - `setColReadColDyn(int*)` - Moyenne : 3.1040us
  - `setRowReadRow(int*)` - Moyenne : 2.1440us
  - `setRowReadColPad(int*)` - Moyenne : 2.1440us
  - D'autres configurations présentent des tendances similaires, mettant en évidence les avantages du rembourrage et les inconvénients de l'allocation dynamique.

Visualisation de la sortie du noyau et de la transformation de la matrice

- **Visualisation de la matrice 2D** : Pour des raisons de simplicité, la tuile de mémoire partagée est fixée à `4x4`.
  - Définir les dimensions dans le code :

```
#define BDIMX 4

#define BDIMY 4
```

- **Exécuter la commande pour obtenir la matrice de sortie** :

```
> smemSquare.exe 1
```

- **Exemple de sortie** :
  - Montre les transformations où les ordres de lecture et d'écriture diffèrent (par exemple, lecture majeure de la ligne, écriture majeure de la colonne), résultant en une matrice transposée.
- **Conclusion** : Ces noyaux de base peuvent servir d'éléments de construction pour des opérations de transposition de matrices plus complexes, avec des avantages en termes de performances visibles grâce à des schémas d'accès à la mémoire soignés.

Introduction à la mémoire partagée rectangulaire

- **Définition** : La mémoire partagée rectangulaire est un type de mémoire partagée en 2D dont les lignes et les colonnes sont de dimensions inégales.
  - Exemple : `__shared__ int tile[Row][Col] ;`
- **Défi majeur** :
  - Contrairement à la mémoire partagée carrée, le simple échange des coordonnées des threads pour effectuer une transposition peut entraîner des violations de l'accès à la mémoire.
  - Nécessité de recalculer les indices d'accès pour traiter les dimensions non carrées.
- **Exemple de configuration** :
  - Dimensions d'un réseau rectangulaire :

```
#define BDIMX 32 // Colonnes
#define BDIMY 16 // Lignes
```

  - Allocation de mémoire :

```
__shared__ int tile[BDIMY][BDIMX] ;
```

Configuration du noyau pour une mémoire partagée rectangulaire

- **Configuration du lancement du noyau** :
  - Utilisez une seule grille et un bloc 2D correspondant aux dimensions du tableau rectangulaire :
    - `bloc dim3 (BDIMX, BDIMY) ;`
    - `dim3 grid (1,1) ;`

- **Considérations relatives à la sécurité de l'accès à la mémoire** :
  - Recalculer les indices d'accès pour les faire correspondre à la disposition rectangulaire afin d'éviter les violations d'accès à la mémoire.
  - La permutation directe des coordonnées (telle qu'elle est utilisée dans les tableaux carrés) n'est pas sûre ; à la place, il convient d'adapter les schémas d'accès aux opérations majeures par ligne ou par colonne, selon les besoins.
- **Conclusion** : La gestion de la mémoire partagée rectangulaire dans CUDA nécessite une configuration minutieuse et des indices recalculés pour un accès sûr et efficace, en particulier lors d'opérations telles que la transposition.

Accès aux lignes et aux colonnes dans la mémoire partagée

- **Noyaux** :
  - `setRowReadRow(int *out)` : Accès à la rangée principale, conflits de banque minimaux.
  - `setColReadCol(int *out)` : Accès majeur à la colonne, conflits de banque importants.

- **Résultats des conflits bancaires (K40)** :
  - `setRowReadRow` : 1 transaction par chargement/stockage (pas de conflit).
  - `setColReadCol` : 8 transactions par chargement/stockage (conflit à 8 voies).

- **Conclusion** : L'accès par rangée majeure évite les conflits de banque, tandis que l'accès par colonne majeure provoque des conflits en raison de la structure à 8 banques de Kepler K40 (figure 5-6).

Écriture d'une ligne majeure et lecture d'une colonne majeure dans une mémoire partagée

- **Objectif** : Transposer efficacement des données à l'aide de la mémoire partagée dans CUDA.
- **Opérations de mémoire** :
  - **Écriture** : Ecritures majeures en ligne dans la mémoire partagée pour éviter les conflits de banque.
  - **Lire** : Les lectures principales de la colonne pour effectuer la transposition.
  - **Écriture en mémoire globale** : Écritures groupées de chaque chaîne.
- **Calcul de l'indice** :
  - Convertit l'index de thread 2D en un ID de thread global 1D (`idx`).
  - Transposer les coordonnées :
    - `unsigned int irow = idx / blockDim.y ;`
    - `unsigned int icol = idx % blockDim.y ;`
- **Initialisation** : Stocker les identifiants globaux des threads dans la tuile de mémoire partagée :

```
tile[threadIdx.y][threadIdx.x] = idx ;
```

Code du noyau et analyse des performances

- **Code du noyau** (`setRowReadCol`) :
  - Déclaration de mémoire partagée : `` `__shared__ int tile[BDIMY][BDIMX];` ``
  - Opérations de stockage et de lecture avec synchronisation :

```
tile[threadIdx.y][threadIdx.x] = idx ;
__syncthreads() ;
out[idx] = tile[icol][irow] ;
```

- **Résultats des performances** (`nvprof`) :
  - **Mesures** :
    - `shared_load_transactions_per_request = 8` (conflit à 8 sur la charge).
    - `shared_store_transactions_per_request = 1` (sans conflit).
- **Commande d'impression de la matrice** : Exécuter `smemRectangle.exe 1` pour vérifier que les données de la mémoire globale sont transposées.

- **Conclusion** : Les écritures en ligne majeure sont exemptes de conflits, mais les lectures en colonne majeure introduisent des conflits de banque sur un K40 en raison de la structure à 8 banques.

Calculs de l'objet et de l'index pour la mémoire partagée dynamique capitonnée

- **Objectif** : Utiliser le padding dans la mémoire partagée allouée dynamiquement pour éviter les conflits de banques dans les tableaux rectangulaires.
- **Trois indices clés** :
  - `row_idx` : Index majeur dans la mémoire partagée pour accéder aux lignes de la matrice.
  - `col_idx` : Index majuscule de colonne dans la mémoire partagée pour l'accès aux colonnes de la matrice.
  - `g_idx` : Index permettant d'accéder à la mémoire globale de manière groupée.
- **Code de calcul de l'indice** :

```
// Mapper l'index des threads à la mémoire globale
unsigned int g_idx = threadIdx.y * blockDim.x + threadIdx.x ;

// Conversion de l'index global en ligne et colonne transposées
unsigned int irow = g_idx / blockDim.y ;
unsigned int icol = g_idx % blockDim.y ;

// Calculer les indices de la mémoire partagée
 unsigned int row_idx = threadIdx.y * (blockDim.x + IPAD) + threadIdx.x ;
unsigned int col_idx = icol * (blockDim.x + IPAD) + irow ;
```

Code du noyau et analyse des performances

- **Code du noyau** (`setRowReadColDynPad`) :
  - Déclarer une mémoire partagée dynamique avec remplissage :

```
extern __shared__ int tile[] ;
```

- Opérations de stockage et de chargement :

```
tile[row_idx] = g_idx ; // Écriture dans une mémoire partagée rembourrée
__syncthreads() ;// Synchronisation des threads
out[g_idx] = tile[col_idx] ; // Lecture à partir d'une mémoire rembourrée avec accès transposé
```

- **Résultats des performances** (`nvprof`) :
  - `shared_load_transactions_per_request = 1.000000`
  - `transactions_magasin_partagé_par_requête = 1.000000`
- **Conclusion** : Le rembourrage élimine avec succès les conflits entre banques, ce qui permet un accès efficace à la mémoire en réduisant le nombre de transactions par requête.

Comparaison des performances des noyaux à mémoire partagée rectangulaire

- **Objectif** : Comparer les temps d'exécution de différents noyaux à mémoire partagée avec des tableaux rectangulaires afin d'analyser les effets du remplissage et de l'utilisation dynamique de la mémoire.
- **Commandement pour mesurer la performance** :

```
>nvprof smemRectangle.exe
```

- **Résultats des performances (Tesla K40c)** :
  - `setRowReadColDyn(int*)` - Moyenne : 2.4000 µs
  - `setRowReadColDynPad(int*)` - Moyenne : 2.2400 µs
  - `setRowReadCol(int*)` - Moyenne : 2.1760 µs
  - `setRowReadColPad(int*)` - Moyenne : 2.1120 µs
  - `setRowReadRow(int*)`- Moyenne : 1.8240 µs

- **Conclusion** :
  - Le remplissage améliore les performances en réduisant les conflits entre les banques.
  - La mémoire partagée dynamique introduit un léger surcoût.

Visualisation de la transposition d'une matrice avec une mémoire partagée rectangulaire

- **Objectif** : Afficher la transformation du contenu pour chaque noyau afin de vérifier les opérations de transposition.
- **Mise en place** :
  - Définir des dimensions réduites pour la visualisation :

```
#define BDIMX 8
#define BDIMY 2
```

- **Commande pour afficher les résultats** :

```
> smemRectangle.exe 1
```

- **Exemple de sortie** :
  - `setRowReadRow` : Matrice originale (pas de transposition).
  - Autres noyaux : Matrices transposées utilisant diverses configurations de rembourrage et de mémoire dynamique.

- **Aperçu** : Les sorties de la matrice visuelle confirment que chaque noyau effectue une transposition, les configurations de rembourrage présentant des transformations plus efficaces.

Réduction de l'accès à la mémoire globale dans les noyaux CUDA

- **Objectif de la mémoire partagée** :
  - Mettre les données en cache sur la puce afin de minimiser les accès à la mémoire globale.
  - Améliore les performances en réduisant la latence d'accès à la mémoire.

- **Focus sur les noyaux de réduction parallèles (chapitre 3) :**
  - **Éviter les divergences entre les chaînes** : Réorganiser les schémas d'accès aux données pour s'assurer que les fils d'une chaîne suivent des chemins similaires.
  - **Déroulement de boucle** : Dérouler les boucles pour maintenir les opérations en vol et optimiser l'utilisation des instructions et de la bande passante de la mémoire.

- **Objectif actuel** : réexaminer les noyaux de réduction parallèles utilisant la mémoire partagée comme cache géré par le programme afin de réduire davantage l'accès à la mémoire globale et d'améliorer l'efficacité et les performances.