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

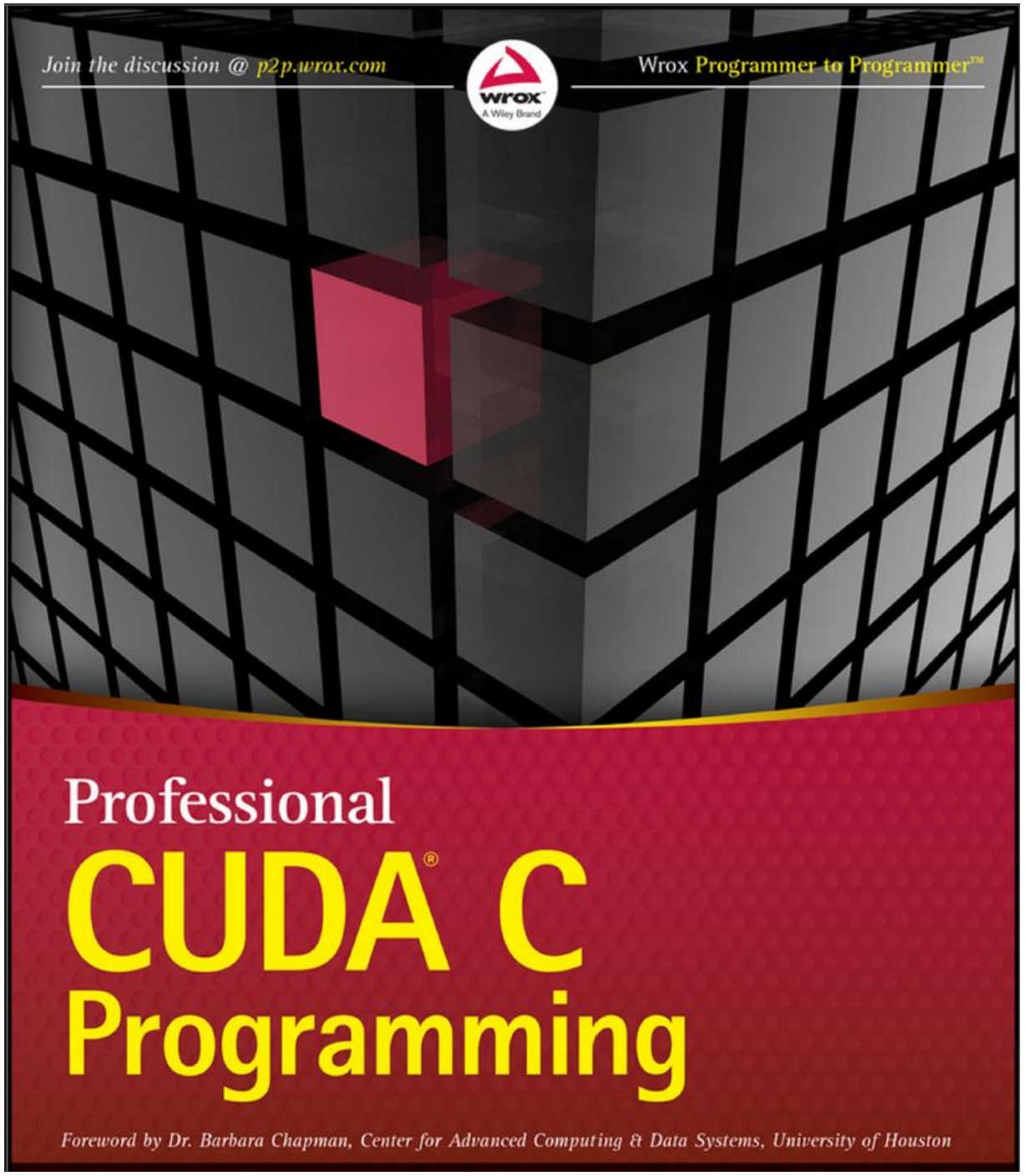
School of Electrical Engineering
and Computer Science

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 4 : La mémoire globale

Plan

- Apprendre le modèle de mémoire CUDA
- Gestion de la mémoire CUDA
- Programmation avec la mémoire globale
- Exploration des schémas d'accès à la mémoire globale
- Sonder l'agencement des données de la mémoire globale
- Programmation avec mémoire unifiée
- Maximiser le débit de la mémoire globale

Présentation du modèle de mémoire CUDA

- Point clé :
  - L'accès à la mémoire et sa gestion sont cruciaux dans la programmation, en particulier pour les calculs à haute performance avec des accélérateurs.

- Défis :
  - De nombreuses charges de travail sont limitées par la vitesse de chargement et de stockage des données.
  - La mémoire haute performance est coûteuse et n'est pas toujours réalisable.

- Solution :
  - Le modèle de mémoire CUDA permet d'optimiser la latence et la bande passante en unifiant les systèmes de mémoire de l'hôte et du périphérique, offrant ainsi un contrôle total sur la hiérarchie de la mémoire.

Avantages d'une hiérarchie de la mémoire

- Principe de localité :
  - Localité temporelle : Les données récemment consultées sont susceptibles d'être consultées à nouveau prochainement.
  - Localité spatiale : Les emplacements de mémoire proches sont susceptibles d'être accédés en même temps que l'un d'entre eux.

- Structure hiérarchique de la mémoire :
  - Optimise les performances en stockant les données dans une mémoire à faible latence lorsqu'elles sont utilisées activement, et dans une mémoire à haute latence lorsqu'elles ne sont pas utilisées.
  - Équilibre le coût, la capacité et la latence entre les différents niveaux de mémoire.

Types de mémoire dans la hiérarchie (Figure 4-1)

- Registres (le plus petit, le plus rapide)
  - Le plus cher et la plus faible capacité.
- Caches
  - Légèrement plus grand, il est utilisé pour les données fréquemment consultées.
- Mémoire principale
  - Basé sur la DRAM, capacité plus importante mais latence plus élevée.
- Mémoire du disque (la plus grande, la plus lente)
  - Utilisé pour le stockage de données à long terme avec la plus grande capacité mais la vitesse d'accès la plus lente.

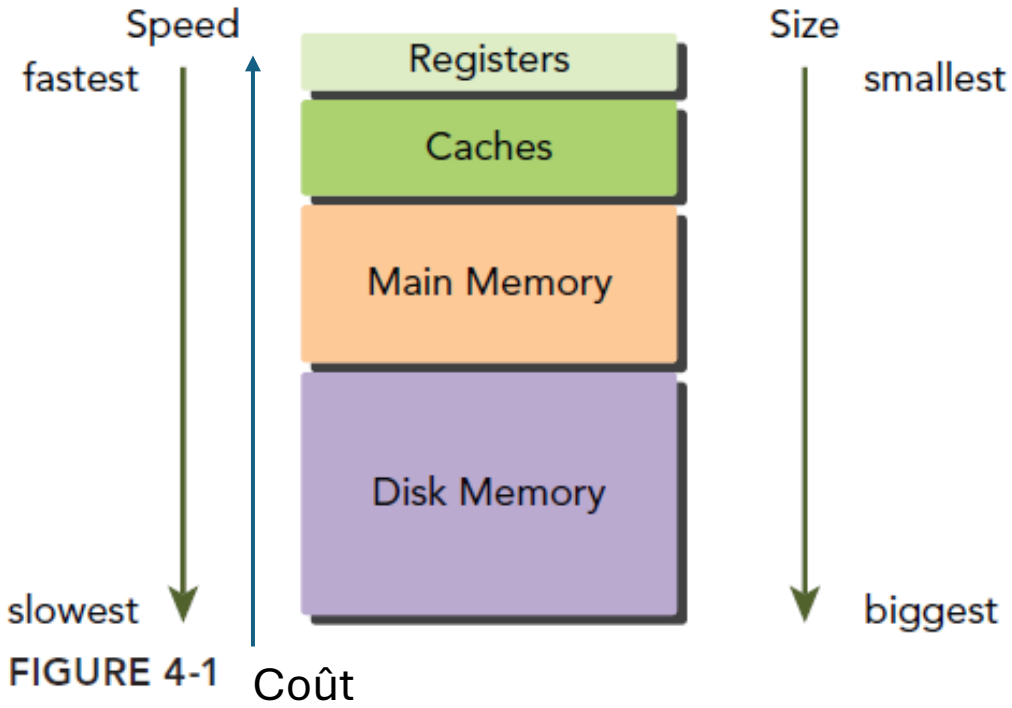


FIGURE 4-1

Présentation du modèle de mémoire CUDA

- Deux classifications de la mémoire :
  - Programmable : Contrôle explicite de l'emplacement des données.
  - Non programmable : Placement automatique des données, pas de contrôle par le programmeur.

- Mémoire CPU vs CUDA :
  - La hiérarchie de la mémoire de l'unité centrale comprend la mémoire non programmable (par exemple, les caches L1 et L2).
  - CUDA expose de nombreux types de mémoire programmable, tels que :
    - Registres
    - Mémoire partagée
    - Mémoire locale
    - Mémoire constante
    - Mémoire de texture
    - Mémoire globale

Hiérarchie de la mémoire dans CUDA (Figure 4-2)

- Registres :
  - L'espace mémoire le plus rapide et le plus petit, propre à chaque thread.

- Mémoire partagée :
  - Accessible à tous les threads au sein d'un même bloc de threads.

- Mémoire locale :
  - Espace mémoire privé pour les différents threads.

- Mémoire globale, mémoire constante, mémoire de texture :
  - Visible par tous les threads, optimisé pour différentes utilisations et persistant pendant toute la durée de vie de l'application.

Hiérarchie de la mémoire dans CUDA

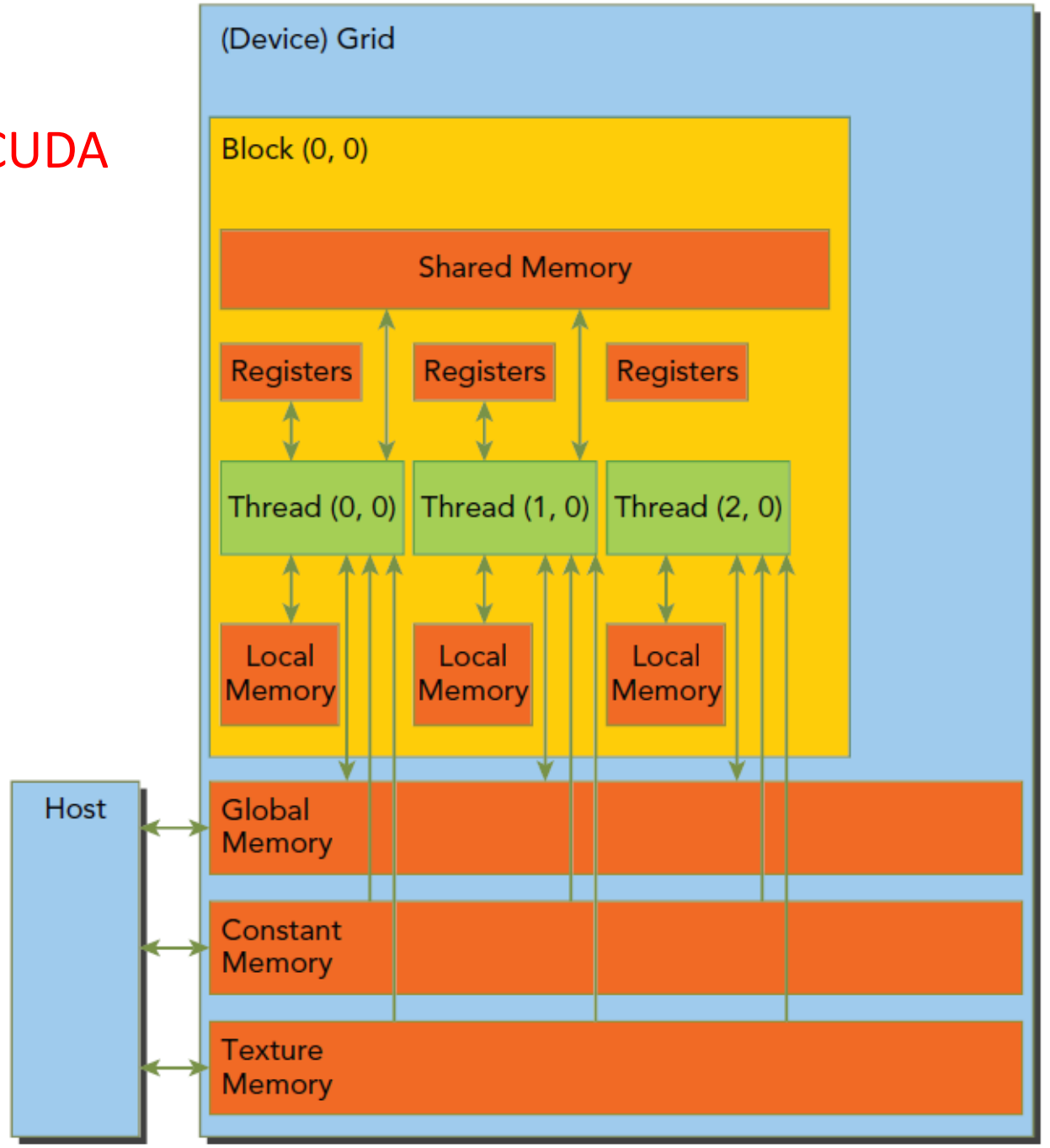


FIGURE 4-2

Registres dans CUDA

- Espace mémoire le plus rapide :
  - Les registres sont la mémoire la plus rapide d'un GPU.
  - Utilisé pour stocker les variables automatiques déclarées dans les noyaux.

- Du privé au thread de l'eau :
  - Chaque thread dispose de son propre ensemble de registres.
  - Les registres partagent leur durée de vie avec le noyau.

- Limites et gestion des ressources :
  - Nombre limité de registres par thread.
  - Une utilisation excessive peut entraîner un débordement du registre (les données sont stockées dans une mémoire locale plus lente).
  - L'optimisation de l'utilisation des registres peut améliorer les performances et l'occupation des blocs de threads.

Contrôle de l'utilisation des registres dans CUDA

- Déversement du registre :
  - Se produit lorsque le nombre de registres nécessaires est supérieur au nombre de registres disponibles.
  - Les performances sont réduites car les données se déversent dans une mémoire plus lente.

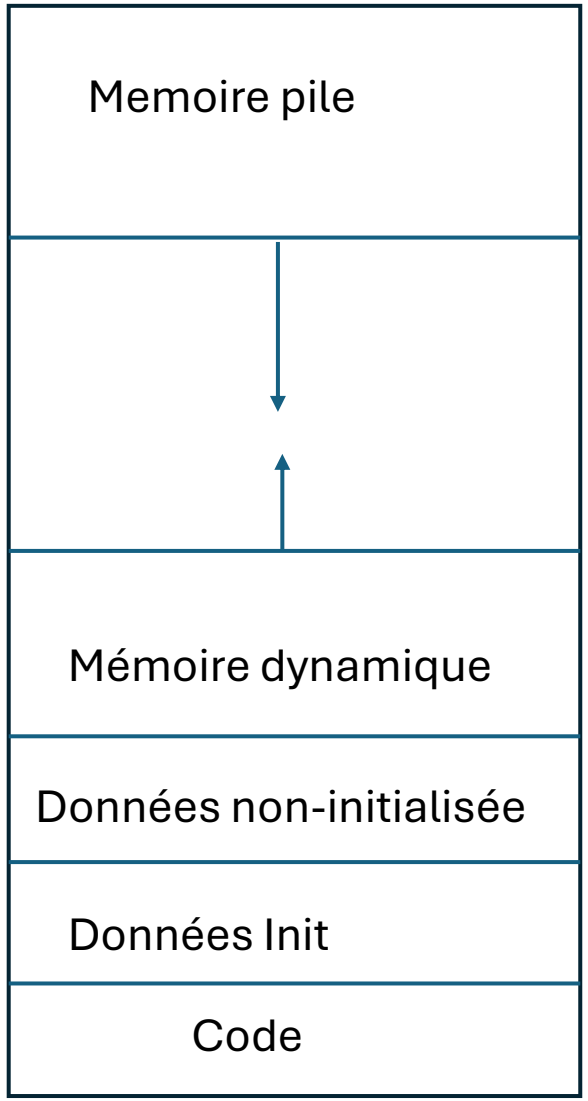
- Gestion de l'utilisation du registre :
  - Utilisez l'option `` `-Xptxas` `` pour vérifier l'utilisation du registre.
  - Contrôler les registres avec les limites de lancement et les options du compilateur comme `` `-maxrregcount`. ``

- Exemple :

```
__global__ void kernel(...) {
  // corps du noyau
}
(maxThreadsPerBlock, minBlocksPerMultiprocessor) ;
```

Mémoire constante dans CUDA

- Vue d'ensemble de la mémoire constante :
    - Mémoire en cache, en lecture seule, optimisée pour être utilisée par tous les threads d'une chaîne.
    - A utiliser de préférence lorsque tous les threads lisent à partir du même emplacement de mémoire.
- Déclaration :
    - Les variables constantes sont déclarées avec le qualificatif `__constant__`.
    - Peut uniquement être lu et doit être initialisé par l'hôte.
- Optimisation de l'accès à la mémoire :
    - Fonctionne mieux lorsque tous les threads d'une chaîne lisent les mêmes données afin de réduire les transactions en mémoire.


Memoire pile
Mémoire dynamique
Données non-initialisée
Données Init
Code

Mémoire de texture dans CUDA

- Qu'est-ce que la mémoire de texture ?
  - Mémoire cache, en lecture seule, optimisée pour la localité spatiale 2D.
  - Utilisé pour les applications nécessitant une interpolation, telles que le traitement d'images.

- Considération de la performance :
  - Utilisez la mémoire de texture pour les applications nécessitant un accès optimisé aux données 2D.
  - Peut être plus lent que la mémoire globale pour certaines applications.

Mémoire globale dans CUDA

- Caractéristiques :
  - Mémoire la plus grande et la plus couramment utilisée sur le GPU.
  - Temps de latence le plus élevé, mais accessible à tous les threads.

- Conseils d'optimisation :
  - Veiller à ce que les transactions de mémoire soient alignées (32, 64, 128 octets).
  - Un accès efficace à la mémoire globale dépend de la distribution des adresses mémoire entre les threads et de la garantie de l'alignement par transaction.

- Meilleures pratiques :
  - Minimiser le nombre de transactions par demande de mémoire.
  - Exploiter la localité des données pour améliorer l'efficacité du débit.

Caches du GPU dans CUDA

- Types de caches GPU :
  - Caches L1 et L2
  - Cache constant en lecture seule
  - Cache de texture en lecture seule

- Mécanisme de mise en cache :
  - Les caches L1/L2 sont utilisés pour stocker des données dans la mémoire locale/globale.
  - Les opérations de chargement de la mémoire peuvent être mises en cache, mais pas les opérations de stockage.
  - Les caches de constantes et de textures en lecture seule sont optimisés pour les espaces mémoire respectifs.

Événements et mesures

Dans le profilage CUDA :

- Les événements sont des activités dénombrables suivies par des compteurs matériels pendant l'exécution du noyau.
- Les métriques sont des caractéristiques d'un noyau, calculées sur la base d'un ou plusieurs événements.

Points clés :

- La plupart des compteurs sont rapportés par multiprocesseur de streaming (et non pour l'ensemble du GPU).
- Une seule exécution de profilage ne peut collecter que quelques compteurs, et certains compteurs s'excluent mutuellement, ce qui nécessite plusieurs exécutions pour collecter toutes les données nécessaires.
- Les valeurs des compteurs peuvent varier d'une exécution à l'autre en raison des différences d'exécution du GPU, telles que la programmation des blocs de threads et des warp.

Déclarations de variables CUDA

- Types de variables CUDA :
  - Registre : Stocké par thread.
  - Local : Stocké par thread, peut se déverser dans la mémoire locale.
  - Partagé : Accessible à tous les threads d'un bloc.
  - Global : visible par tous les threads de l'appareil.
  - Constante : Mémoire en lecture seule, mise en cache pour tous les threads.

- Caractéristiques de la mémoire (tableau 4-2) :
  - Registres : Sur la puce, accès rapide.
  - Local : Hors puce, portée des threads, accès plus lent.
  - Partagé : Sur la puce, par bloc, utilisé pour la communication inter-thread.
  - Global/Constant/Texture : Portée globale, mise en cache pour plus d'efficacité.

Les principales caractéristiques des différents types de mémoire sont résumées dans le tableau 4-2.

TABLEAU 4-2 : Caractéristiques principales de la mémoire des dispositifs

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

Déclaration de mémoire globale statique (exemple)

- Variable globale statique Exemple :

```
Dispositif__ float devData ;

__global__ void checkGlobalVariable() {
    printf("Value : %f", devData) ;
    devData += 2.0f ;
}
```

- Initialisation :
  - Utilisez `cudaMemcpyToSymbol` pour initialiser les variables globales du côté de l'hôte.

- Interaction entre le code de l'hôte et celui de l'appareil :
  - Le code hôte et le code de l'appareil ne peuvent pas accéder directement aux variables l'un de l'autre.
  - Il est possible d'accéder aux variables globales de la mémoire de l'appareil à l'aide d'appels API.

Accès à la mémoire de l'appareil à partir de l'hôte

- Accès aux variables globales de l'appareil :
  - Utilisez `cudaMemcpyToSymbol` ou `cudaGetSymbolAddress` pour accéder aux variables globales de l'hôte.

- Utilisation des pointeurs :
  - La fonction `cudaMemcpy` peut transférer des données vers la mémoire globale en utilisant des pointeurs.

- Notes clés :
  - L'hôte ne peut pas déréférencer directement les pointeurs de périphériques.
  - La mémoire épinglée est nécessaire pour permettre un accès direct à la mémoire de l'appareil hôte sans intervention de l'unité centrale.

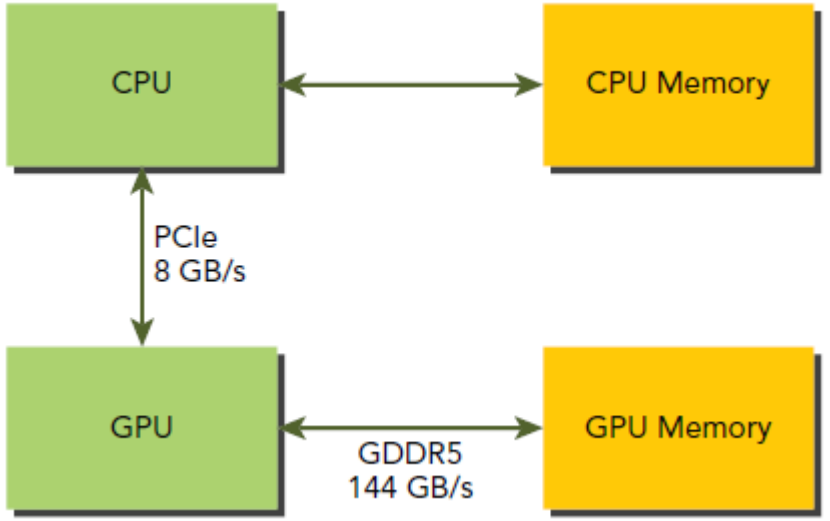
Transfert de mémoire dans CUDA

- Fonction cudaMemcpy
  - Objet : transfert de données entre l'hôte et l'appareil.
  - Fonction Signature :

```
cudaError_t cudaMemcpy(void *dst, const void
*src, size_t count, enum cudaMemcpyKind kind) ;
```

- Paramètres :
  - dst : Pointeur de destination.
  - src : Pointeur de source.
  - count : Nombre d'octets à copier.
  - kind : Spécifie la direction du transfert, peut être :
    - `cudaMemcpyHostToHost` (en anglais)
    - `cudaMemcpyHostToDevice` (en anglais)
    - `cudaMemcpyDeviceToHost` (appareil à l'hôte)
    - `cudaMemcpyDeviceToDevice`

- Synchronisation : La fonction est synchrone, ce qui signifie que les données sont transférées immédiatement et que l'exécution est bloquée jusqu'à ce que le transfert soit terminé.



La figure 4-3 illustre la connectivité de la mémoire du CPU et du GPU.

Exemple : Transfert de mémoire simple (memTransfer.cu)

- Objet : Démonstration du transfert de mémoire de l'hôte à l'appareil et vice-versa.
- Les étapes :
  1. Configurez le périphérique en utilisant `` `cudaSetDevice(dev)` ``.
  2. Allouer de la mémoire à l'hôte et à l'appareil :
     - Hôte : `float *h_a = (float *)malloc(nbytes) ;`
     - Dispositif : `cudaMalloc((float **)&d_a, nbytes) ;`
  3. Initialiser la mémoire de l'hôte avec des valeurs.
  4. Transférer des données de l'hôte à l'appareil :

```
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice) ;
```

  5. Transférer les données de l'appareil vers l'hôte :

```
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost) ;
```

  1. Libère la mémoire allouée.

Profilage du transfert de mémoire avec nvprof

- Exemple de profilage avec nvprof :
  - Le résultat du profilage montre les temps de transfert de données pour `HostToDevice (HtoD)` et `DeviceToHost DtoH).`

Bande passante de la mémoire dans le GPU et le CPU

- Figure 4-3 : Connectivité et largeur de bande
  - Cette figure illustre la bande passante entre le CPU et le GPU via le bus PCIe.
- Détails clés :
  - La mémoire GDDR5 du GPU a une bande passante théorique élevée de 144 Go/s.
  - Le bus PCI Express (PCIe) Gen2 offre une bande passante beaucoup plus faible de 8 Go/s entre le CPU et le GPU.

- À retenir : Il existe une disparité importante dans la bande passante de la mémoire entre le bus PCIe et la mémoire du GPU. Il est essentiel de minimiser les transferts de données entre l'hôte et l'appareil pour maximiser les performances.

La figure 4-3 illustre la connectivité de la mémoire du CPU et du GPU.

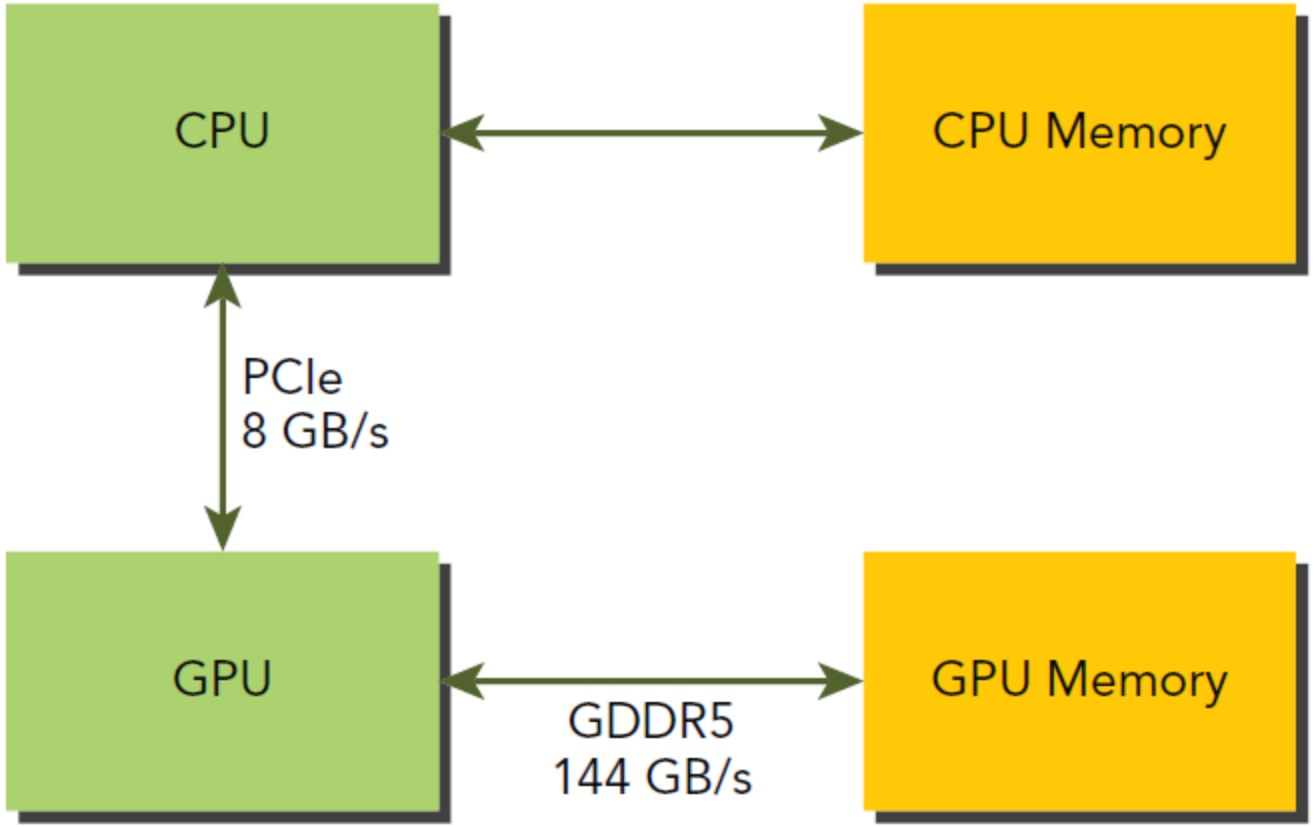


FIGURE 4-3

Mémoire épinglée

- **Problème de mémoire paginée :** La mémoire hôte est paginée, c'est-à-dire qu'elle peut être déplacée par le système d'exploitation, ce qui peut entraîner des ralentissements dus à des défauts de page.

- **Solution :** Utilisez la **mémoire épinglée** (également connue sous le nom de **mémoire verrouillée**) pour éviter les erreurs de page et permettre des transferts de mémoire plus rapides.

Figure 4-4: Transferts de mémoire paginée ou épinglée

- Cette figure compare la **mémoire paginée** et la **mémoire épinglée** :
  - **Transfert de mémoire paginée :** Plus lent en raison des erreurs de page potentielles et de la gestion de la mémoire virtuelle.
  - **Transfert de mémoire verrouillée :** Plus rapide car la mémoire est verrouillée dans la mémoire physique, ce qui permet un accès direct pour le transfert de données entre l'hôte et le GPU.

Transferts de mémoire paginée ou épinglée

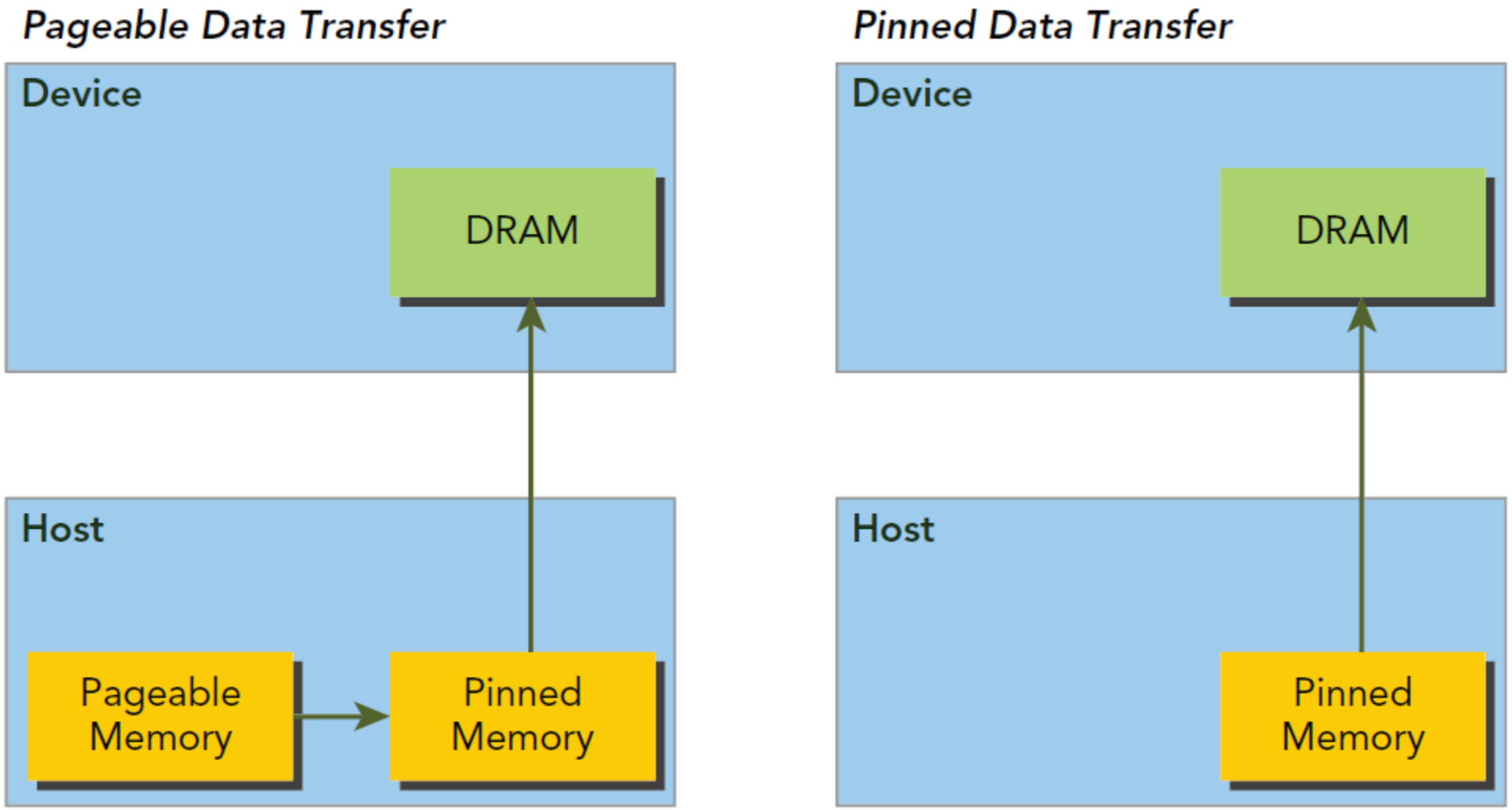


FIGURE 4-4

Utilisation de `cudaMallocHost` pour la mémoire épinglée

- **Fonction** :

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

- **Comportement** : alloue de la mémoire verrouillée sur l'hôte, ce qui permet des transferts directs et plus rapides vers et depuis le périphérique par rapport à la mémoire paginée.

Exemple de mémoire épinglée

- Allocation de la mémoire épinglée :

```
cudaMallocHost((void**)&h_aPinned, nbytes) ;
```

- Transférer des données entre la mémoire hôte épinglée et l'appareil.

- Mémoire libre épinglée :

```
cudaFreeHost(h_aPinned) ;
```

Transfert de mémoire entre l'hôte et le dispositif

- Mémoire épinglée :
  - Coût : L'allocation et la désallocation sont plus coûteuses que pour la mémoire paginée.
  - Avantage : permet d'augmenter le débit pour les transferts de données importants.

Performances avec la mémoire enfouie

- Accélération :
  - Les performances dépendent de la capacité de calcul de l'appareil.
  - Sur les appareils Fermi, la mémoire à broches est bénéfique pour le transfert de plus de 10 Mo de données.

- Optimisation
  - Regrouper les petits transferts en lots plus importants pour réduire la charge de travail

Chevauchement des transferts avec l'exécution du noyau

- Concurrence :
  - Les transferts de données peuvent être superposés à l'exécution du noyau pour améliorer les performances.
  - Recommandation : Réduire au minimum les transferts de données ou les faire se chevaucher chaque fois que cela est possible.

Aperçu de la mémoire à copie zéro

- Deux catégories d'architectures hétérogènes :
  - Intégré : Les CPU et les GPU partagent la même mémoire.
  - Discret : Les dispositifs sont connectés à l'hôte via le bus PCIe.

Avantages de la mémoire sans copie

- Architectures intégrées :
  - La mémoire sans copie est bénéfique à la fois pour les performances et la programmabilité.
  - Il n'est pas nécessaire de copier les données sur le bus PCIe.

Utilisation dans les systèmes discrets

- Systèmes discrets :
  - La mémoire à copie zéro n'est avantageuse que dans des cas particuliers.
  - Synchronisation nécessaire pour éviter les risques liés aux données lorsque plusieurs threads accèdent au même emplacement de mémoire.

Inconvénients de la mémoire à copie zéro

- Temps de latence élevé :
  - Les noyaux de périphériques qui lisent à partir d'une mémoire à copie zéro peuvent présenter une latence élevée.
  - Attention : Évitez d'abuser de la mémoire à copie zéro.

Aperçu de l'adressage virtuel unifié (Unified Virtual Addressing, UVA)

- Prise en charge à partir de CUDA 4.0 :
  - Introduit pour les appareils avec Compute Capability 2.0+.
  - Pris en charge sur les systèmes Linux 64 bits.
- Principale caractéristique : L'UVA permet à l'unité centrale, à la mémoire de l'hôte et à la mémoire de l'appareil de partager un seul espace d'adressage virtuel.

Comprendre la figure 4-5 :

- Sans UVA :
  - Plusieurs espaces mémoire distincts pour le CPU et les GPU (GPU0, GPU1).
  - Chaque espace mémoire doit être géré individuellement (gestion manuelle des pointeurs).

Avec UVA :

- Espace mémoire unifié unique pour le CPU et les GPU.
- La gestion de la mémoire est simplifiée et les pointeurs sont transparents.

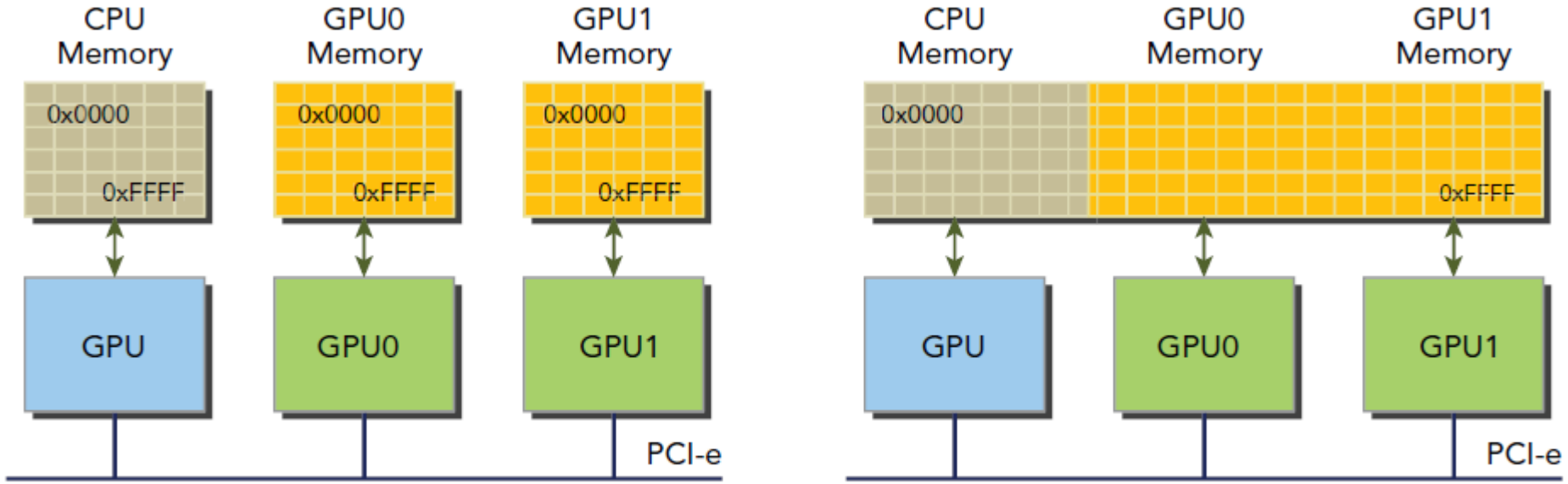


FIGURE 4-5

Avantages des UVA

- Avant l'UVA : les développeurs devaient gérer des pointeurs distincts pour la mémoire de l'hôte et celle du périphérique.
- Avec UVA :
  - Les pointeurs sont transparents : il n'est pas nécessaire de faire la distinction entre les pointeurs de la mémoire de l'hôte et ceux de la mémoire de l'appareil.
  - Code simplifié : Réduit la complexité de la gestion de la mémoire.

Mémoire zéro-copie avec UVA

- Mémoire d'hôte épinglée :
  - Alloué en utilisant `cudaHostAlloc` avec `cudaHostAllocMapped`.
  - Les pointeurs de l'hôte et du dispositif sont identiques.
  - Il n'est pas nécessaire de transmettre des pointeurs de périphériques distincts aux noyaux.

Exemple de code et compilation

- Exemple de code :

```
cudaHostAlloc((void **)&h_A, nBytes, cudaHostAllocMapped) ;
cudaHostAlloc((void **)&h_B, nBytes, cudaHostAllocMapped) ;

sumArraysZeroCopy<<grid, block>>>(h_A, h_B, d_C, nElem) ;
```

- Compilation :

```
$ nvcc -O3 -arch=sm_61 sumArrayZerocopyUVA.cu -o sumArrayZerocopyUVA
```

Code et performances simplifiés

- Résultats :
  - Performances identiques à celles des implémentations antérieures à l'UVA.
  - Avantages : Améliore la lisibilité et la maintenabilité du code sans perte de performance.

Introduction à la mémoire unifiée

- Introduit avec CUDA 6.0 pour simplifier la gestion de la mémoire.
- La mémoire unifiée crée un pool de mémoire gérée.
- Accessible à la fois sur le CPU et le GPU avec un seul pointeur.
- Le système migre automatiquement les données entre l'hôte et l'appareil, de manière transparente pour l'application.

Adressage virtuel unifié (UVA) et mémoire unifiée

- La mémoire unifiée s'appuie sur le soutien de l'UVA, mais ils sont différents :
  - L'UVA fournit un espace d'adressage de mémoire virtuelle unifié.
  - UVA ne migre pas les données, mais la mémoire unifiée le fait.

Avantages de la mémoire unifiée

- Offre un modèle de "pointeur unique sur les données".
- Découple les espaces de mémoire et d'exécution.
- Les données sont migrées de manière transparente entre l'hôte et l'appareil.
- Améliore la localité et les performances par rapport à la mémoire à copie zéro.

Mémoire gérée

- Désigne les allocations de mémoire unifiée gérées par le système.
- Interopérable avec les allocations spécifiques aux appareils comme `cudaMalloc`.
- Autorise les types de mémoire gérés et non gérés dans un noyau.
- Opérations de mémoire unifiée valables à la fois pour l'appareil et pour l'hôte.

Mémoire gérée

- Désigne les allocations de mémoire unifiée gérées par le système.
- Interopérable avec les allocations spécifiques aux appareils comme `cudaMalloc`.
- Autorise les types de mémoire gérés et non gérés dans un noyau.
- Opérations de mémoire unifiée valables à la fois pour l'appareil et pour l'hôte.

Déclaration de la mémoire gérée

- La mémoire gérée peut être déclarée statiquement avec le mot-clé `__managed__` :

```
__device__ __managed__ int y ;
```

- La mémoire gérée peut être allouée dynamiquement :

```
cudaMallocManaged(&devPtr, size, flags) ;
```

Principales caractéristiques de la mémoire gérée

- Migration automatique des données entre l'hôte et l'appareil.
- Il n'est pas nécessaire de dupliquer les pointeurs.
- Dans CUDA 6.0, le code du périphérique ne peut pas appeler `cudaMallocManaged`
  - il doit être alloué par l'hôte.

Exemple

- Exemple pratique d'addition matricielle avec la mémoire unifiée

Modèles d'accès à la mémoire

- Utilisation globale de la mémoire :
  - La plupart des accès aux données de l'appareil commencent dans la mémoire globale.
  - La bande passante de la mémoire est un facteur limitant courant dans les applications GPU.
  - Un réglage correct de la mémoire globale est essentiel pour les performances du noyau.
  - Un mauvais réglage de l'utilisation de la mémoire se traduira par des gains de performance négligeables grâce à d'autres optimisations.

Modèles d'accès à la mémoire

- Accès optimal à la mémoire :
  - Exécution par chaîne : Les instructions et les opérations de mémoire sont émises par chaîne.
  - Les fils d'une chaîne fournissent des adresses de mémoire pour le chargement et le stockage.
  - Les 32 threads d'une chaîne présentent une seule demande d'accès à la mémoire qui est traitée par une ou plusieurs transactions de mémoire.
  - Modèles de mémoire :
    - Les schémas d'accès sont déterminés par la répartition des adresses entre les fils d'une chaîne.
    - Différents modèles affectent le nombre de transactions de mémoire.
    - Objectif : obtenir un accès optimal à la mémoire globale grâce à des schémas de mémoire efficaces.

Accès aligné et coalescé

- Mise en cache de la mémoire globale :
  - Les accès à la mémoire globale sont échelonnés à travers les caches (L1 et L2).
  - Les données résident dans la DRAM et les transactions de mémoire sont traitées par des segments de 2 octets (cache L2) ou de 128 octets (cache L1).
  - Le cache L1 peut être activé/désactivé au moment de la compilation pour une meilleure mise en cache de la mémoire.

- Détails du cache L1 :
  - Taille de la ligne de cache L1 : 128 octets.
  - Les fils d'une chaîne demandent des valeurs de 4 octets, soit 128 octets par chaîne.

Accès aligné et coalisé

- Caractéristiques principales d'accès à la mémoire :
  - Accès à la mémoire alignée : Les adresses de données sont alignées sur la granularité du cache (32 ou 128 octets).
  - Accès à la mémoire coalescée : Les 32 threads d'une chaîne accèdent à des blocs de mémoire contigus.
  - Objectif : obtenir un accès aligné et coalescé à la mémoire pour une efficacité maximale.

Figure 4-6

- Montre les étapes de la mise en cache de la mémoire, où le SM (Streaming Multiprocessor) utilise les caches L1, L2 et la DRAM pour les transactions de mémoire.

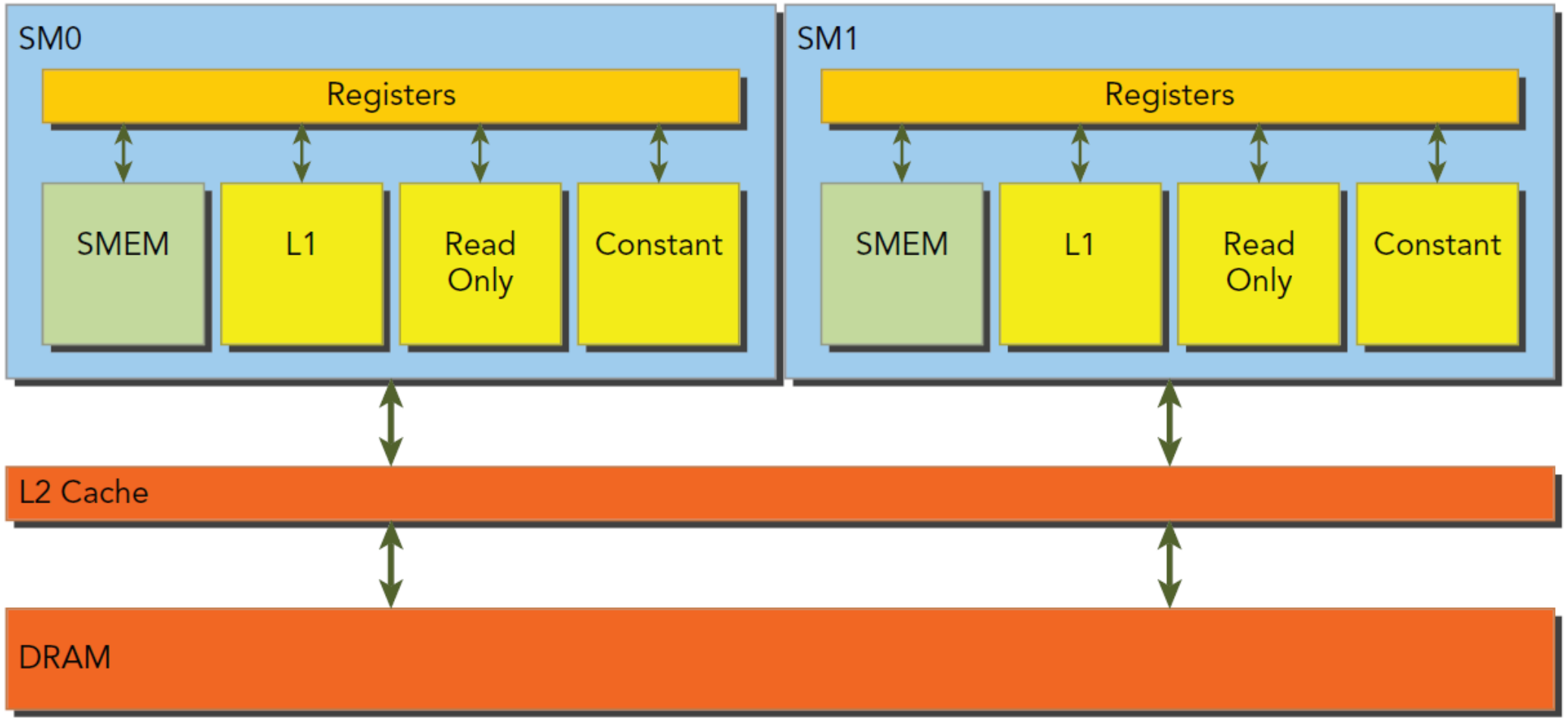


FIGURE 4-6

Accès à la mémoire alignée

- Optimisation de l'accès aligné :
  - L'alignement des adresses mémoire permet d'éviter le gaspillage de la bande passante.
  - Un accès mal aligné entraîne des transactions supplémentaires, ce qui réduit le débit.

Figure 4-7 :

- Illustration de l'accès à la mémoire alignée où les 32 threads accèdent à des adresses mémoire alignées et contiguës (128 octets de données sont récupérés de manière efficace).

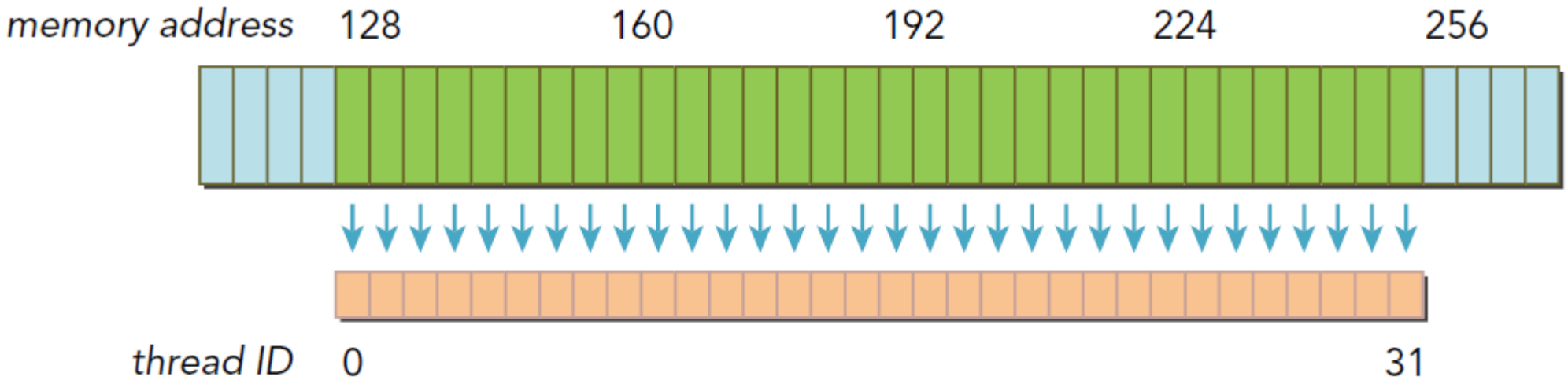


FIGURE 4-7

Accès désaligné et non coalisé

- Accès à la mémoire mal aligné :
  - Les threads accèdent à la mémoire non alignée, ce qui nécessite plusieurs transactions de mémoire.
  - Il en résulte un gaspillage de la bande passante et une utilisation inefficace de la mémoire globale.

Figure 4-8 :

- Démontre un accès mal aligné et non coordonné, où plusieurs transactions sont nécessaires pour répondre aux demandes de mémoire, ce qui entraîne un gaspillage de ressources.

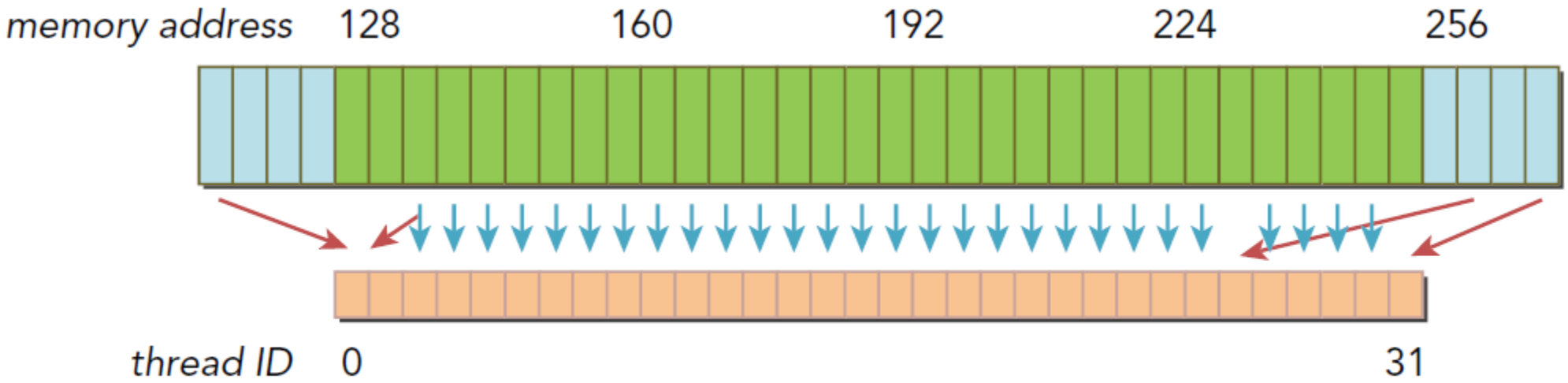


FIGURE 4-8

Aperçu de la lecture des mémoires globales

- **Titre** : Vue d'ensemble de la lecture des mémoires globales
- **Points clés** :
  - Dans un SM, les données sont acheminées en pipeline à travers trois chemins potentiels de cache/tampon :
  - Cache L1/L2
  - Cache constant
  - Cache en lecture seule
- L1/L2 Cache est le chemin par défaut.
- Pour faire passer des données par d'autres chemins, une gestion explicite est nécessaire.
- La question de savoir si les opérations de chargement de la mémoire globale passent par le cache L1 dépend de ce qui suit :
  - **Capacité de calcul de l'appareil**
  - **Options du compilateur**

Comportement du cache et options du compilateur

- **Titre** : Comportement du cache et options du compilateur
- **Points clés** :
  - Pour les GPU Fermi (capacité 2.x) et les GPU Kepler K40 (capacité 3.5+) :
  - La mise en cache L1 des charges de mémoire globale peut être activée/désactivée par les drapeaux du compilateur.
  - **Par défaut** :
    - Le cache L1 est activé pour les charges de mémoire globale sur Fermi.
    - Désactivé sur Kepler K40 et les versions ultérieures.
- **Désactivation du cache L1** :
  - Indicateur du compilateur : `-Xptxas -dlcm=cg`
- **Activation du cache L1** :
  - Indicateur du compilateur : `-Xptxas -dlcm=ca`

Comportement des caches L1 et L2

- **Titre** : Comportement des caches L1 et L2
- **Points clés** :
  - Avec le cache L1 désactivé :
    - Toutes les demandes sont directement adressées à L2.
    - Si L2 rate sa cible, les demandes sont traitées par la DRAM.
  - Avec le cache L1 activé :
    - Demandes de charge première tentative L1.
    - En cas d'échec, les demandes sont transmises à L2, puis à la DRAM.

Utilisation du cache L1 sur les GPU Kepler

- **Titre** : Utilisation du cache L1 sur les GPU Kepler
- **Points clés** :
  - Sur les GPU Kepler K10, K20, K20x :
    - L1 **n'**est **pas** utilisé pour mettre en cache les charges de la mémoire globale.
    - L1 est exclusivement utilisé pour mettre en cache les débordements de registres dans la mémoire locale.

Modèles d'accès à la charge de la mémoire

- **Titre** : Modèles d'accès à la charge de la mémoire
- **Points clés** :
    - Deux types de chargements de mémoire :
        - En cache (cache L1 activé)
        - Non mis en cache (cache L1 désactivé)
    - Modèles d'accès caractérisés par :
        - **Mise en cache et non mise en cache**
        - **Aligné ou mal aligné** : Un chargement est aligné si la première adresse mémoire est un multiple de 32 octets.
        - **Coalesced vs Uncoalesced** : Coalescé si le warp accède à un bloc de données contigu.

Impact sur les performances

- **Titre** : Impact sur les performances des schémas d'accès à la mémoire
- **Points clés** :
  - L'impact des schémas d'accès à la mémoire sur les performances du noyau sera examiné dans les sections suivantes.

Chargements en cache

- **Points clés** :
  - Les opérations de chargement en cache passent par le cache L1.
  - Les transactions de mémoire sont traitées avec une granularité de 128 octets.
  - Les charges mises en cache peuvent être :
    - Aligné/Misaligné
    - Coalescé/non coalescé
  - **FIGURE 4-9** : Illustration d'un cas idéal où les accès à la mémoire sont alignés et coalisés. Tous les threads de la chaîne se trouvent à l'intérieur d'une ligne de cache, ce qui maximise l'utilisation du bus.

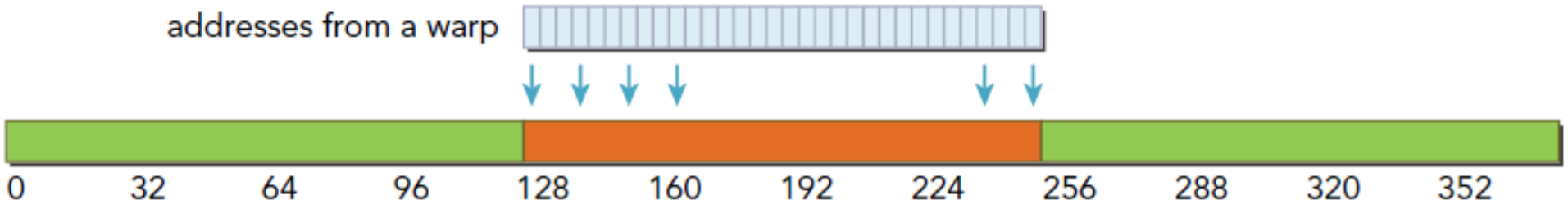


FIGURE 4-9

Accès aligné avec randomisation

- **Titre** : Accès aligné avec des ID de threads aléatoires
- **Points clés** :
  - Les adresses sont aléatoires mais se situent toujours à l'intérieur d'une ligne de cache.
  - Une seule transaction de 128 octets est nécessaire.
  - L'utilisation des bus reste à 100%.
  - **FIGURE 4-10**: Visualise ce cas aligné mais aléatoire.

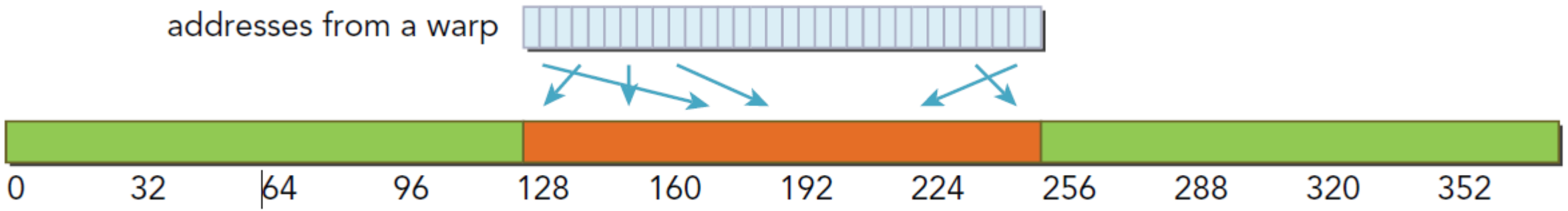


FIGURE 4-10

Accès désaligné

- **Titre** : Accès désaligné (50% d'utilisation)
- **Points clés** :
  - Les fils demandent 32 éléments de données consécutifs qui ne sont pas alignés.
  - Deux transactions de 128 octets sont nécessaires en raison d'un mauvais alignement.
  - L'utilisation des bus tombe à 50 %.
  - **FIGURE 4-11** : Illustration de l'impact du désalignement sur les opérations de chargement de la mémoire.

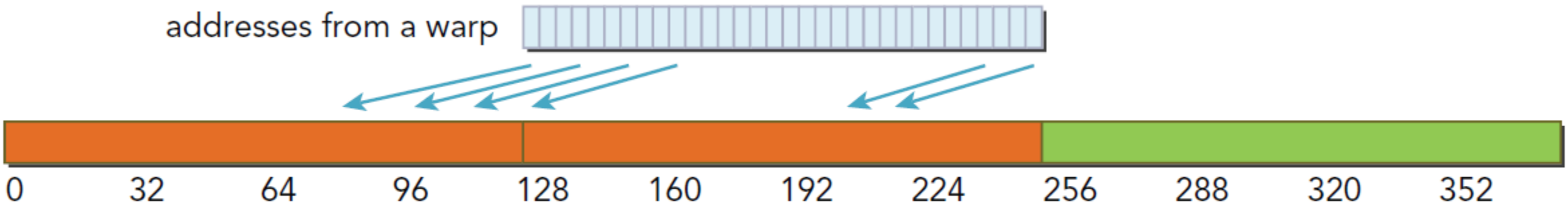


FIGURE 4-11

Tous les fils demandent la même adresse

- **Titre** : Faible utilisation de la demande de la même adresse
- **Points clés** :
  - Les fils demandent la même adresse.
  - L'utilisation des bus est très faible - seulement 3,125 %.
  - **FIGURE 4-12** : Démontre comment l'utilisation du bus chute lorsque tous les threads demandent la même adresse.

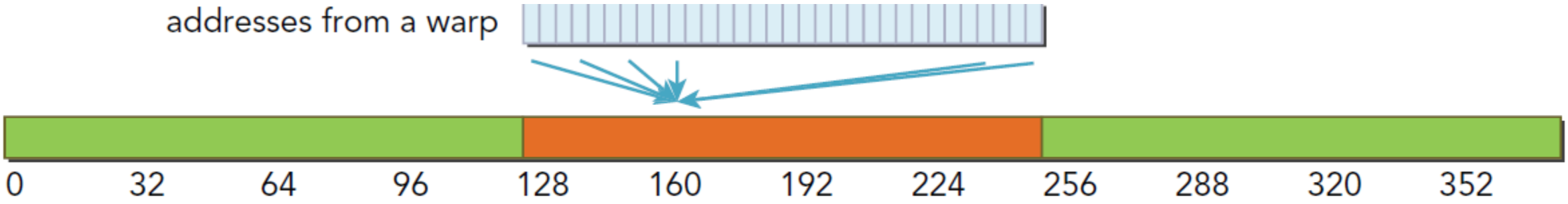


FIGURE 4-12

Scénario le plus défavorable (32 threads, 32 lignes de cache)

- **Points clés** :
- Les threads demandent 32 adresses de quatre octets réparties sur plusieurs lignes de cache.
  - Plusieurs transactions de mémoire sont nécessaires, ce qui réduit l'efficacité.
  - **FIGURE 4-13** : Ce scénario catastrophe est illustré par des adresses dispersées sur plusieurs lignes de cache.

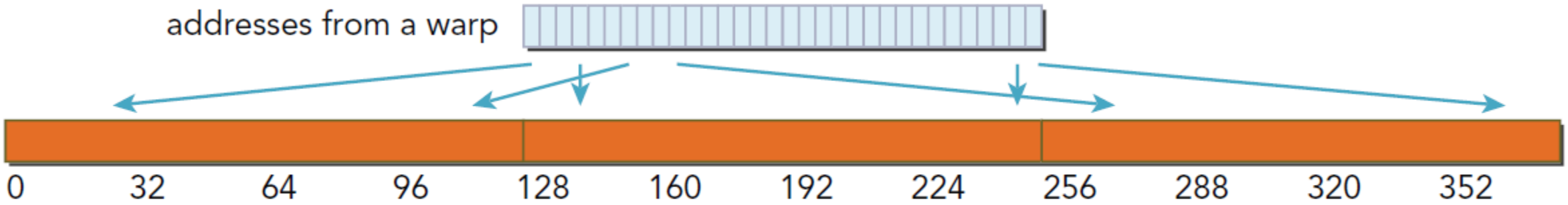


FIGURE 4-13

Charges non mises en cache

- **Les charges non mises en cache** ne passent pas par le cache L1 et opèrent à une granularité plus fine : des segments de mémoire de 32 octets, et non des lignes de cache de 128 octets. Cela peut conduire à une meilleure utilisation du bus pour les accès à la mémoire mal alignés ou non coalisés.

- **FIGURE 4-14**
  - **Accès à la mémoire alignée et coalisée** :
    - Dans ce cas idéal, les adresses sont réparties en quatre segments (32 octets chacun).
    - L'**utilisation du bus** est de **100 %** car les adresses de la mémoire sont alignées et accédées de manière groupée.

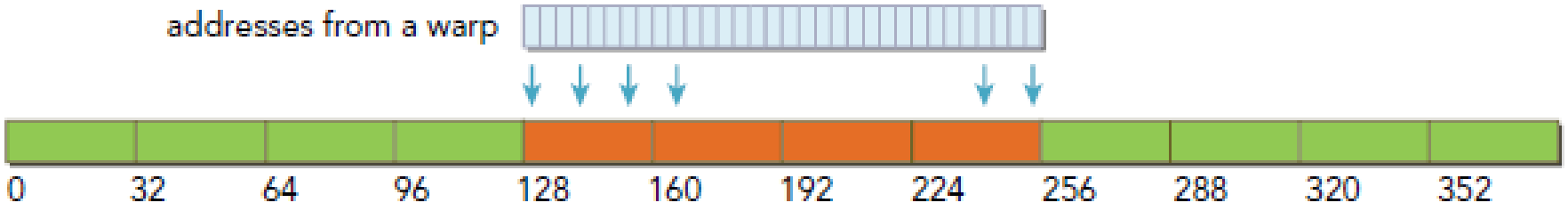


FIGURE 4-14

FIGURE 4-15

- **Accès aléatoires mais alignés** :
  - sont randomisés dans une plage de 128 octets.
  - Tant que chaque thread accède à des adresses uniques, les données seront toujours récupérées efficacement dans quatre segments de mémoire.
  - Les performances restent élevées avec une bonne exécution du noyau.

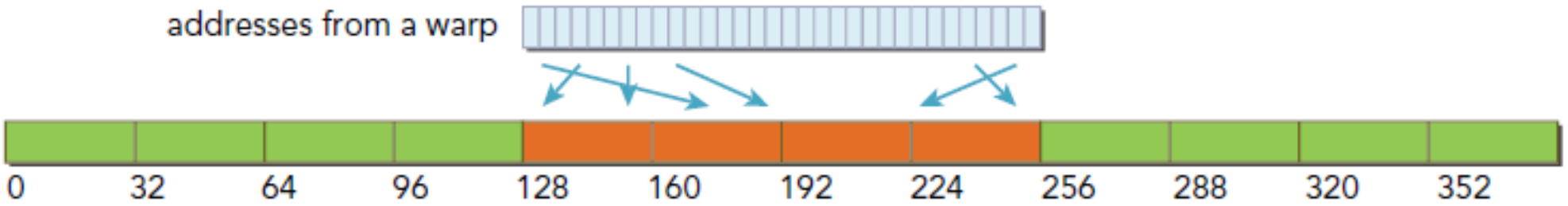


FIGURE 4-15

Charges mal alignées et dispersées

- **Impact des schémas d'accès non séquentiels** sur les performances de la mémoire.
- **FIGURE 4-16**
  - **Éléments consécutifs de 4 octets avec une charge désalignée** :
    - Une chaîne demande 32 éléments consécutifs de 4 octets, mais la charge de mémoire n'est pas alignée sur une frontière de 128 octets.
    - La transaction de 128 octets s'effectue sur cinq segments de mémoire, ce qui permet d'atteindre un **taux d'utilisation du bus d'**au moins **80 %**.

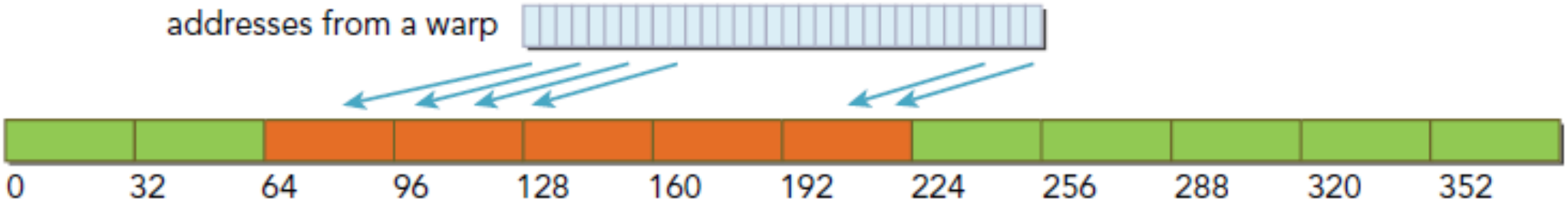


FIGURE 4-16

Différence entre le cache L1 du CPU et le cache L1 du GPU

- **Titre** : Cache L1 du CPU vs. du GPU
- **Points clés** :
  - Cache L1 du processeur : Optimisé pour la localité spatiale et temporelle.
  - Cache L1 du GPU : Optimisé uniquement pour la localité spatiale.
  - L'accès fréquent à un emplacement de mémoire cache ne garantit **pas** que les données resteront dans le cache L1 du GPU.

FIGURE 4-18

- **Le pire des scénarios** :
  - La chaîne demande 32 mots de 4 octets répartis dans la mémoire globale.
  - Dans ce cas, les données sont réparties sur plusieurs lignes de cache de 128 octets, ce qui se traduit par une faible efficacité. Toutefois, ce cas le plus défavorable est encore amélioré par rapport aux charges mises en cache.

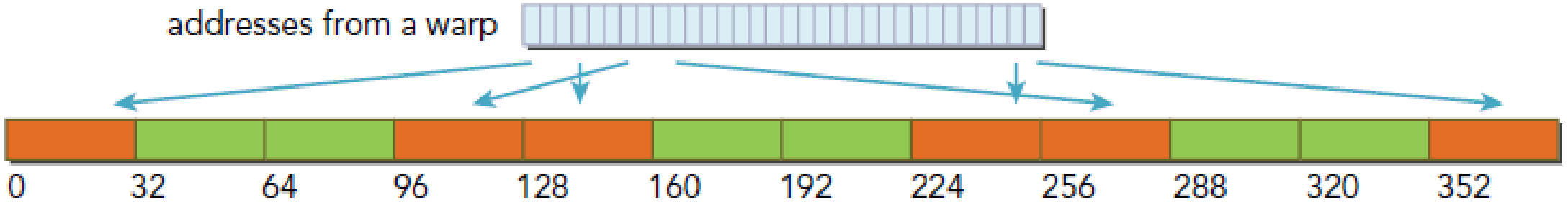


FIGURE 4-18

Exemple de lectures mal alignées et de leur impact sur les performances

- **Accès à la mémoire mal aligné** : Se produit lorsque les charges de mémoire provenant des tableaux sont décalées, ce qui rend l'accès à la mémoire inefficace.
- **Stratégies d'alignement** : L'alignement correct des accès à la mémoire permet d'améliorer les performances, en particulier dans les systèmes informatiques parallèles tels que CUDA.

Illustration de l'impact du désalignement

- **Modification du noyau** : Un noyau CUDA est ajusté pour démontrer l'effet des charges désalignées, notamment en introduisant un décalage pour désaligner les accès aux tableaux `A` et `B`.
  - *Cas aligné (décalage = 0) :* Les accès à la mémoire sont regroupés et efficaces.
  - *Cas non aligné (décalage = 11) :* Les accès à la mémoire ne sont pas alignés, ce qui entraîne un manque d'efficacité.

Impact sur les performances d'un accès mal aligné

- **Métrique : Efficacité globale de la charge (gld_efficiency) :**
  - Formule

$$\text{gld\_efficiency} = \frac{\text{Requested Global Memory Load Throughput}}{\text{Required Global Memory Load Throughput}}$$

  - Cas aligné (offset = 0 et 128) : L'efficacité est de 100 %.
  - Cas de désalignement (décalage = 11) : L'efficacité tombe à ~49,81%, ce qui indique que les transactions en mémoire doublent.

Résultats du profilage des NVPROF

- Transactions de charge globale (gld_transactions) : Nombre de transactions en mémoire.
  - Offset = 0 : 65 184 transactions.
  - Décalage = 11 : 131 039 transactions (doublées).
  - Décalage = 128 : 65 744 transactions.

Effet de la désactivation du cache L1

- Commande : -Xptxas -dlcm=cg désactive le cache L1 pour les chargements de mémoire globale.
- Impact :
  - Les performances sont légèrement inférieures sans le cache L1.
  - gld_efficiency :
    - Décalage = 0 et 128 : Reste à 100 %.
    - Décalage = 11 : Amélioration jusqu'à 80 % (mieux que 49,81 %).

Exemple_5_segment de lecture

Écritures en mémoire globale

FIGURE 4-19

- Cas idéal : Tous les threads de la chaîne accèdent à une plage consécutive de 128 octets.
  - Cette opération est gérée par une transaction de 4 segments, ce qui garantit une utilisation à 100 % du bus.

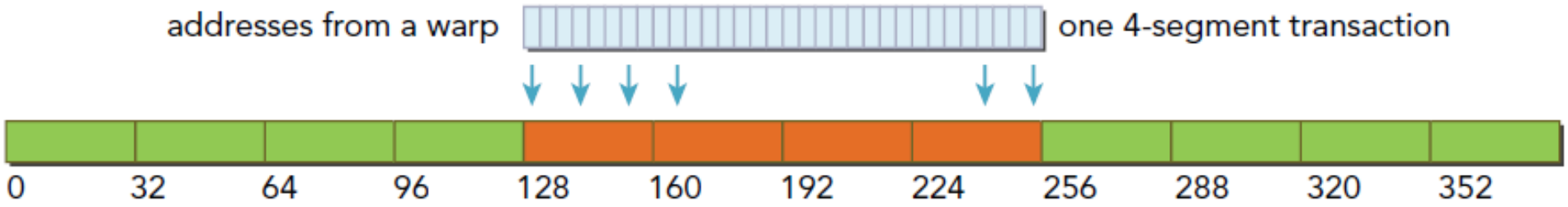


FIGURE 4-19

Écritures en mémoire globale

FIGURE 4-20

- Accès aligné mais dispersé : Les fils de la chaîne accèdent à une plage de 192 octets, qui s'étend sur plusieurs segments.
  - Il en résulte trois transactions à un segment, ce qui réduit l'efficacité.

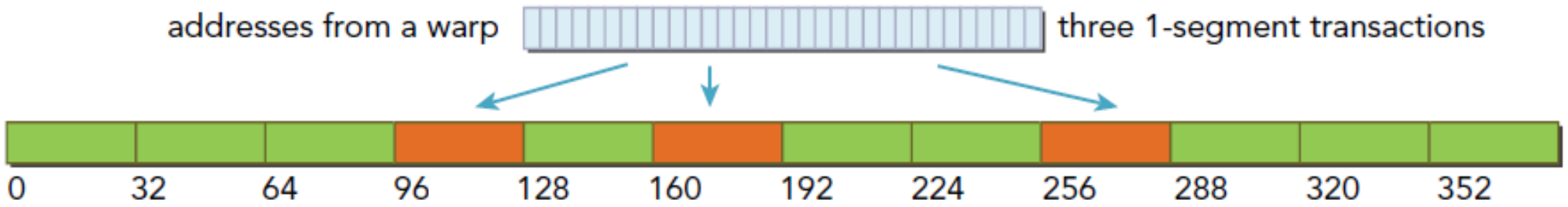


FIGURE 4-20

Écritures en mémoire globale

FIGURE 4-21

- Transaction à deux segments : Lorsque les accès à la mémoire sont alignés mais couvrent une plage de 64 octets, la demande est traitée par une transaction à deux segments.
  - Il en résulte une efficacité accrue par rapport aux écritures mal alignées ou dispersées.

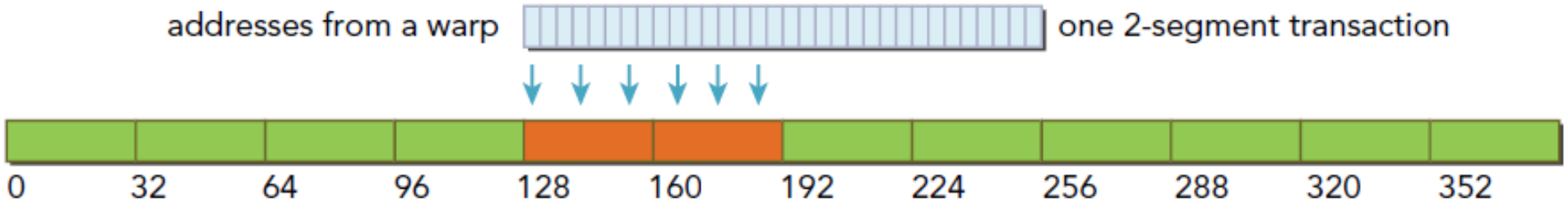


FIGURE 4-21

Écritures mal alignées dans CUDA

- Les écritures mal alignées affectent l'efficacité du stockage en mémoire dans CUDA.
- Cet exemple modifie un noyau d'addition vectorielle pour démontrer cet effet.

Modification du noyau pour le désalignement

- La fonction noyau modifiée `writeOffset` utilise deux indices :
  - i : Index aligné pour la lecture des tableaux A et B.
  - k = i + offset : Index mal aligné pour l'écriture dans le tableau C.
- Introduit un désalignement potentiel en fonction de la valeur du décalage.

Ajustement du code hôte

- La fonction hôte `sumArraysOnHost` a été révisée pour intégrer le décalage :
  - Boucle sur les indices du tableau en utilisant i + offset pour le tableau C.
- Fait correspondre les changements de noyau pour des résultats cohérents.

Compilation et exécution

- Commande de compilation : `nvcc -O3 -arch=sm_61 writeSegment.cu -o writeSegment`.
- Exécuter avec différents décalages (0, 11, 128) pour observer les différences de performance.

Résultats des écritures mal alignées

- Exemples de sorties pour différents décalages :
  - Offset 0 : Aligné, exécution rapide (100% d'efficacité).
  - Décalage 11 : désaligné, plus lent en raison d'une efficacité réduite (80 %).
  - Décalage 128 : Aligné, haute efficacité similaire à l'offset 0.

Mesures d'efficacité avec nvprof

- Analyse de l'efficacité à l'aide de nvprof pour les métriques globales de charge/stockage.
- Commandes :

```
nvprof --metrics gld_efficiency --metrics gst_efficiency ./writeSegment $OFFSET
```

- Les résultats :
  - Les décalages 0 et 128 ont une efficacité de chargement/stockage de 100 %.
  - Le décalage 11 ne présente qu'une efficacité de stockage de 80 % en raison d'un mauvais alignement.

Exemple_6_writeSegment

Cause de la baisse d'efficacité à l'offset 11

- Les écritures mal alignées (offset 11) entraînent des transactions de mémoire fragmentées :
  - La demande d'écriture de 128 octets se divise en une transaction de 4 segments et une transaction de 1 segment.
  - L'accès à 160 octets au lieu de 128 se traduit par une efficacité de 80 % seulement.

Conclusion

- Le désalignement a un impact significatif sur les performances de CUDA.
- L'alignement de l'accès aux données est essentiel pour optimiser les opérations de mémoire.

Réseau de structures (AoS) vs. Structure de réseaux (SoA)

- Options de présentation des données
  - Réseau de structures (AoS) :
    - Stocke des champs connexes (par exemple, `x` et `y`) dans une même structure.
    - Convient à la localité du cache de l'unité centrale, mais moins efficace pour les unités de traitement graphique.
  - Structure des réseaux (SoA) :
    - Sépare chaque champ en tableaux indépendants, en stockant toutes les valeurs `x` dans un tableau et toutes les valeurs `y` dans un autre.
    - Optimisé pour l'accès à la mémoire du GPU grâce à des schémas d'accès à la mémoire coalisés.

La figure 4-22 illustre l'agencement de la mémoire dans les approches AoS et SoA

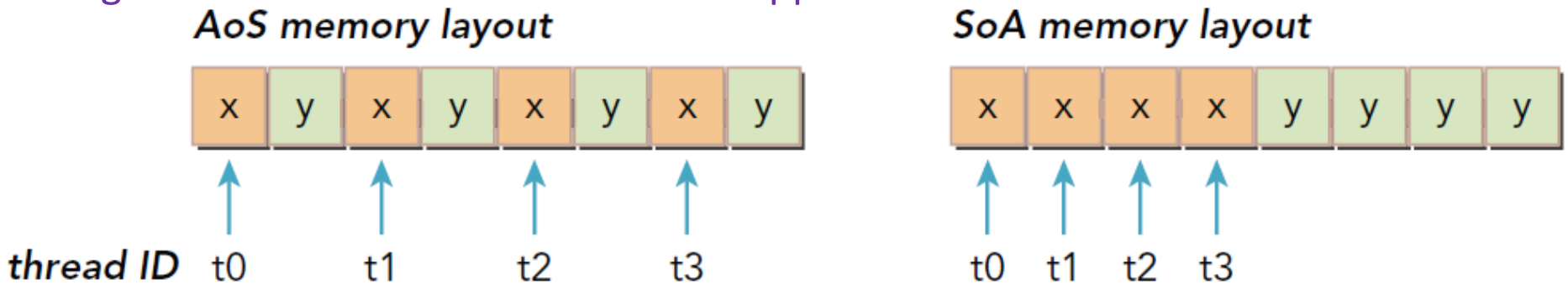


FIGURE 4-22

Exemples de définitions

- Définir les structures de l'AoS et de la SoA
  - Définition de la mise en page de l'AoS :
    - Définir `innerStruct` avec `float x` et `float y`.
    - Tableau de structures : `innerStruct myAoS[N] ;`
  - Définition de la mise en page SoA :
    - Tableaux séparés pour `x` et `y` : `float x[N] ; float y[N] ;`
    - Variable utilisant cette structure : `struct innerArray moa ;`

Différences d'agencement de la mémoire

- Visualisation des schémas de mémoire AoS vs. SoA
  - Mise en page de l'AoS :
    - Les threads chargent à la fois x et y même si seul x est nécessaire.
    - Entraîne une utilisation inefficace de la bande passante sur le GPU.
  - Mise en page SoA :
    - Les champs stockés indépendamment permettent un accès coalisé à la mémoire.
    - Maximise l'efficacité de la bande passante de la mémoire sur le GPU.

Implications pour la programmation parallèle

- Préférence pour la programmation SIMD/CUDA
  - Accès à la mémoire coalescente avec SoA :
    - L'agencement SoA permet un accès plus efficace à la mémoire.
  - Impact de l'AoS :
    - La disposition AoS est bénéfique pour le cache de l'unité centrale, mais elle est moins efficace dans les environnements GPU en raison de la dispersion des accès à la mémoire.

Mise en œuvre de l'agencement AoS dans CUDA

- Mise en œuvre du noyau
  - Fonction du noyau (`testInnerStruct`):
    - Traite les données à l'aide de la présentation AoS.
    - Lit les champs `x` et `y`, effectue des opérations, stocke les résultats.

Allocation de la mémoire CUDA et configuration

- Mémoire CUDA et initialisation
  - Allocation globale de mémoire :
    - Allocation de mémoire pour les structures AoS dans CUDA.
  - Initialisation du réseau d'hôtes :
    - Initialiser les tableaux sur l'hôte avec `initialInnerStruct`.

Exécution Configuration et échauffement

- Configuration de l'exécution
  - Configurer l'exécution :
    - Définir la taille des grilles et des blocs pour l'exécution du noyau.
  - Noyau d'échauffement :
    - Exécution d'un noyau d'échauffement pour réduire les frais de démarrage de CUDA.

Exécution du noyau et mesure des performances

- Exécution et performance du noyau
  - Exécution du noyau :
  - Exécuter le noyau `testInnerStruct` et mesurer le temps écoulé.
  - Exemple de sortie :
    - Exemple de résultat : `0.000286 sec` sur Fermi M2070.

Exemple_7_simpleMathAoS

Exemple_8_simpleMathSoA

Conclusion

- Comparaison entre AoS et SoA
  - Résumé de l'AoS :
    - Bénéfique pour le CPU mais moins efficace pour les GPU.
  - Résumé de la SoA :
    - Maximise l'efficacité du GPU grâce à des schémas d'accès coalisés, ce qui le rend préférable dans les applications CUDA.

Objectifs de l'optimisation des performances

- **Objectif** : Optimiser l'utilisation de la bande passante de la mémoire des appareils.
- **Stratégies clés** :
  - **Accès aligné et groupé** : Réduit le gaspillage de la bande passante en assurant un chargement efficace des données.
  - **Opérations simultanées en mémoire** : Maximise le débit de la mémoire en faisant se chevaucher les opérations pour masquer les temps de latence.
- **Domaines d'intervention** :
  - Aligner et fusionner les accès à la mémoire.
  - Augmenter les opérations de mémoire indépendantes par thread pour un meilleur parallélisme.

Techniques de déroulement

- **Objectif du déroulement des boucles** : Ajoute des opérations de mémoire indépendantes, augmentant ainsi l'accès simultané.
- **Exemple** : Noyau `readOffsetUnroll4` modifié avec 4 chargements de mémoire indépendants par thread.
- **Résultat** : Plus d'accès simultanés possibles, ce qui réduit les temps morts et améliore l'utilisation de la bande passante.

Impact de l'Unrolling sur les performances

- **Gains de performance** : Le désenroulement peut permettre de tripler la vitesse par rapport aux noyaux non désenroulés.
- **Mesures d'efficacité (à l'aide de** `nvprof`**)** :
  - **Cas aligné** : Efficacité élevée de la mémoire.
  - **Cas de désalignement** : Le noyau déroulé présente une meilleure efficacité de charge même en cas de désalignement.

Exemple_9_déroulement du segment de lecture

Exposer plus de parallélisme

- **Taille optimale des blocs** : Expérimentation avec 128, 256 et 512 threads par bloc.
- **Résultats** : les 256 threads par bloc ont donné les meilleurs résultats dans le noyau déroulé, en tirant parti d'un plus grand nombre de blocs par SM.
- **Limitations matérielles** : Sur les GPU Fermi, la simultanéité maximale est limitée par les limites du warp et les ressources SM.

Conclusion - Maximiser l'utilisation de la bande passante

- **Facteurs clés** :
  - **Alignement et regroupement de la mémoire** : Essentiel pour réduire le gaspillage de la bande passante.
  - **Augmentation des opérations indépendantes** : Le déroulement des boucles et l'optimisation de la configuration de l'exécution permettent d'accroître le parallélisme.
- **À retenir** : Aligner et dérouler les schémas d'accès à la mémoire pour optimiser les performances du GPU.

Bande passante dans les performances du noyau

- **Mesures de performance des noyaux** :
  - **Latence de la mémoire** : Temps nécessaire pour satisfaire une demande de mémoire individuelle.
  - **Bande passante de la mémoire** : Taux d'accès à la mémoire du dispositif par SM, en octets par unité de temps.
- **Méthodes d'amélioration des performances des noyaux** :
  - **Cacher la latence de la mémoire** : Maximiser l'exécution des déformations pour augmenter la saturation du bus et les accès à la mémoire en vol.
  - **Maximiser l'efficacité de la bande passante de la mémoire** : Alignement et coalescence appropriés des accès à la mémoire.
- **Les défis de la conception des noyaux** :
  - Traiter les mauvais schémas d'accès pour une performance sous-optimale.
  - Importance de la mise au point pour obtenir des performances réalisables dans des conditions réelles.

Comprendre la bande passante de la mémoire

- **Types de bande passante de la mémoire** :
  - **Largeur de bande théorique** : Largeur de bande maximale possible avec le matériel actuel.
  - **Largeur de bande effective** : Largeur de bande mesurée réelle obtenue par un noyau.

- **Formule de la largeur de bande effective** :

$$\text{Effective Bandwidth (GB/s)} = \frac{(\text{bytes read} + \text{bytes written}) \times 10^{-9}}{\text{time elapsed}}$$

- **Exemple de calcul** (pour une matrice de 2048x2048 d'entiers de 4 octets) :

$$\text{Effective Bandwidth (GB/s)} = \frac{2048 \times 2048 \times 4 \times 2 \times 10^{-9}}{\text{time elapsed}}$$

- **Objectif** : Mesurer et régler la largeur de bande effective d'un noyau de transposition de matrice.

Problème de transposition de matrice

- **Qu'est-ce que la transposition matricielle** ?
  - Opération courante en algèbre linéaire.
  - Il s'agit de permuter les lignes et les colonnes d'une matrice.

- **Exemple** :
  - Matrice (à gauche) et Matrice transposée (à droite), comme le montre la **figure 4-23**.
- **Applications** :
  - Largement utilisé dans les calculs scientifiques et techniques.

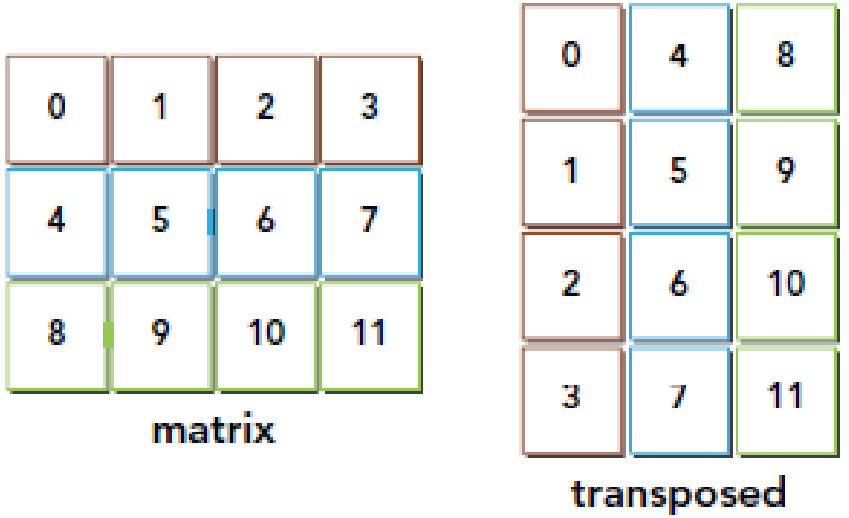


FIGURE 4-23

Mise en œuvre de la transposition basée sur l'hôte

- **Exemple de code :**
  - La fonction hôte `transposeHost()` effectue une transposition hors place en utilisant des nombres flottants en simple précision.
  - Prend une matrice stockée dans un format de tableau 1D.

- **Disposition de la matrice :**
  - La matrice d'entrée `in` et la matrice de sortie `out` sont stockées sous forme de tableaux 1D.
  - Illustré dans la **figure 4-24** :
    - **Disposition des données de la matrice origin**
    - **Disposition des données de la matrice trans**

FIGURE 4-24

Observations sur les schémas d'accès

- **Modèles d'accès** :
  - Lecture : Accès par les lignes de la matrice d'origine pour un accès groupé.
  - Écritures : Accès par les colonnes de la matrice transposée, ce qui entraîne un accès en escalier.

- **Accès enjambé** :
  - Réduit les performances sur les GPU en raison de l'accès non coalisé à la mémoire.
  - Viser à optimiser les schémas de lecture/écriture dans la conception du noyau.

Améliorer l'utilisation de la bande passante

- **Deux approches pour le noyau de transposition** :
  - **Approche 1** : Lecture par lignes et stockage par colonnes (figure 4-25 à gauche).
  - **Approche 2** : Lecture par colonnes et stockage par lignes (figure 4-26 à droite).

- **Implications en termes de performances** :
  - Avec le cache L1 activé pour les charges :
    - L'approche 2 peut améliorer les performances en mettant en cache les lectures.
  - Sans cache L1 pour la mémoire globale :
    - La différence de performance est minime sur les appareils Kepler K10, K20 et K20x.

Approche 1 : Lecture par lignes et stockage par colonnes (figure 4-25 à gauche)

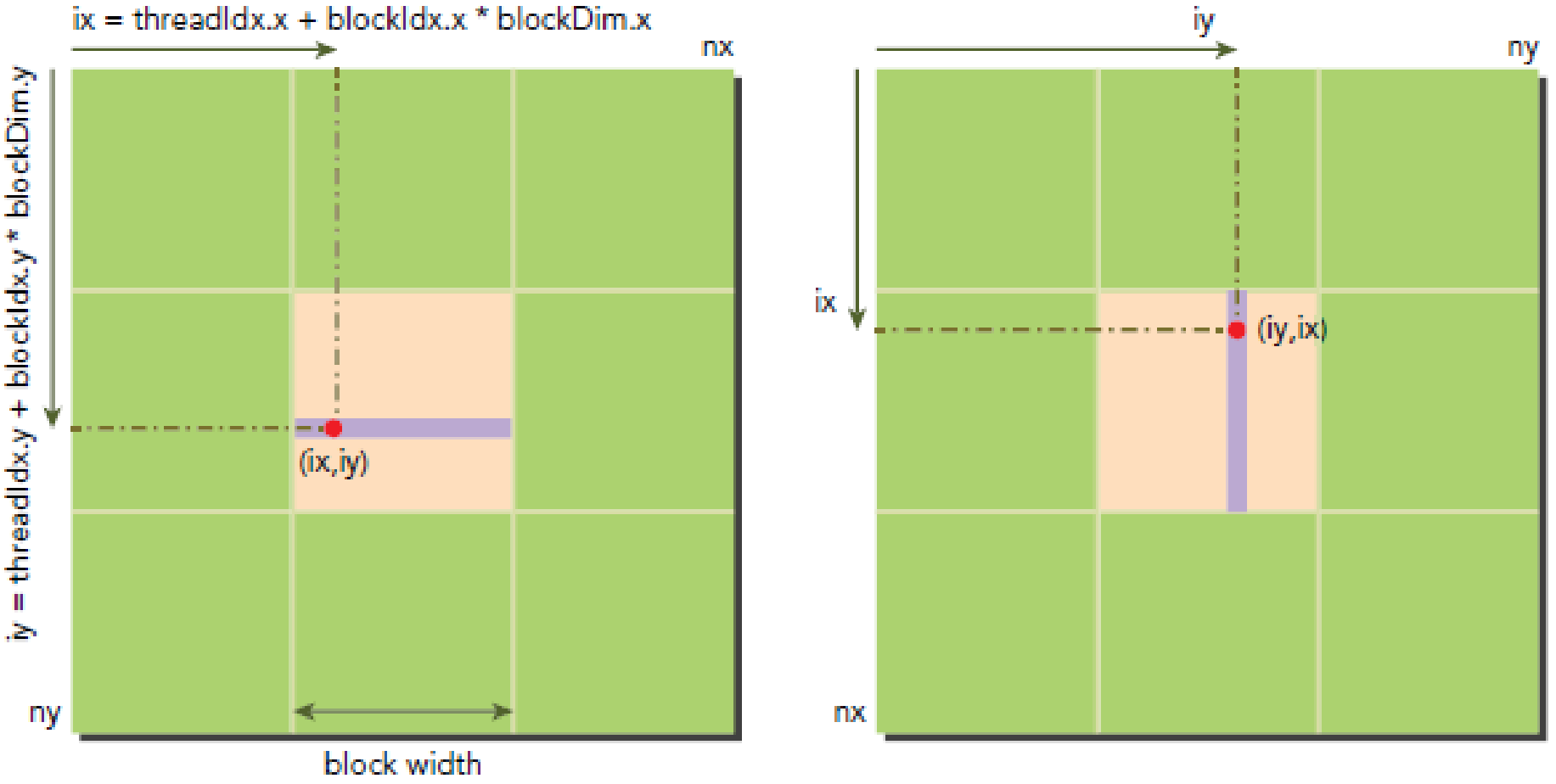


FIGURE 4-25

Approche 2 : Lecture par colonnes et stockage par lignes (Figure 4-26 droite)

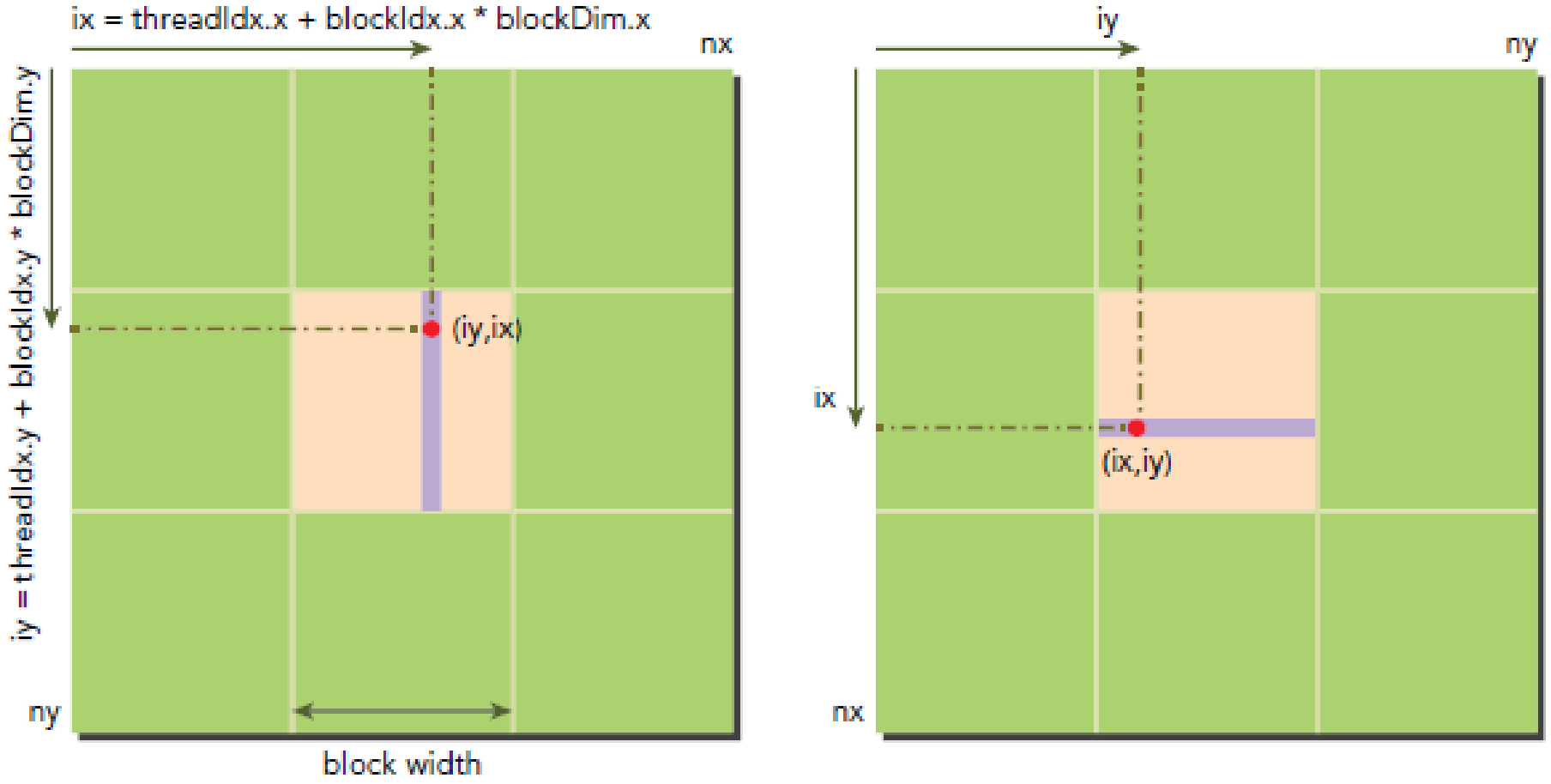


FIGURE 4-26

Présentation de CUDA Matrix Transpose

- **Objectif** : Transposer efficacement une matrice sur le GPU à l'aide de CUDA.
- **Configuration du code** :
  - Définit l'exécution du noyau avec une instruction de `commutation` pour la sélection du noyau.
  - Utilise l'API CUDA pour la configuration des périphériques, l'allocation de mémoire et le transfert de données.
  - Mesure la performance des noyaux.

Variantes du noyau pour l'opération de transposition

- **Noyaux mis en œuvre** :
  - `CopyRow` : Chargement/stockage par ligne.
  - `CopyCol` : Chargement/stockage par colonnes.
  - `NaiveRow` : Lit les lignes, stocke les colonnes.
  - `NaiveCol` : lit les colonnes, stocke les lignes.
- **Commandes d'exécution** :
  - `./transpose 0` pour CopyRow
  - `./transpose 1` pour CopyCol
  - `./transpose 2` pour NaiveRow
  - `./transpose 3` pour NaiveCol

Analyse des performances - Cache L1 activé (tableaux 4-4 et 4-5)

- **Comparaison de la largeur de bande :**
  - **`CopyRow`** : 125.67 GB/s (70.76% du pic)
  - **`CopyCol`** : 58.76 GB/s (33.09% du pic)
  - **`NaiveRow`** : 64,16 Go/s (36,13 % du pic)
  - **`NaiveCol`** : 81,64 Go/s (45,97% du pic)
  - **Observation :** `NaiveCol` est plus performant que `NaiveRow` en raison d'une meilleure utilisation du cache.

Analyse des performances - cache L1 désactivé (tableau 4-6)

- Largeur de bande avec L1 désactivé :
  - **`CopyRow`** : 128,07 GB/s (limite supérieure)
  - **`CopyCol`** : 40,42 GB/s (limite inférieure)
  - **`NaiveRow`** : 63,79 Go/s
  - **`NaiveCol`** : 47,13 Go/s
- **Conclusion** : La désactivation de la mémoire cache L1 réduit considérablement les performances, en particulier pour les schémas d'accès en escalier.

Efficacité de la charge et du stockage (tableaux 4-7 et 4-8)

- **Débit et efficacité** :
  - **`CopyRow`** permet d'obtenir une grande efficacité en matière de chargement et de stockage grâce à un chargement basé sur les lignes.
  - **`NaiveCol`** souffre d'inefficacité dans les lectures en escalier mais bénéficie de la mémoire cache L1.
  - **Aperçu principal** : L'utilisation efficace du cache est cruciale pour améliorer l'efficacité du chargement et du stockage dans les opérations liées à la mémoire du GPU.

Transposition diagonale dans CUDA : Lecture des lignes et des colonnes

- **Aperçu du concept** :
  - Dans la programmation CUDA, les grilles de blocs de threads sont distribuées entre les multiprocesseurs de flux (SM).
  - Chaque bloc de fils a un identifiant unique, `bid`, basé sur un ordre ligne-majeur.
- **Figure 4-27** : Affiche les coordonnées cartésiennes et les identifiants de bloc correspondants pour une grille de 4 x 4.

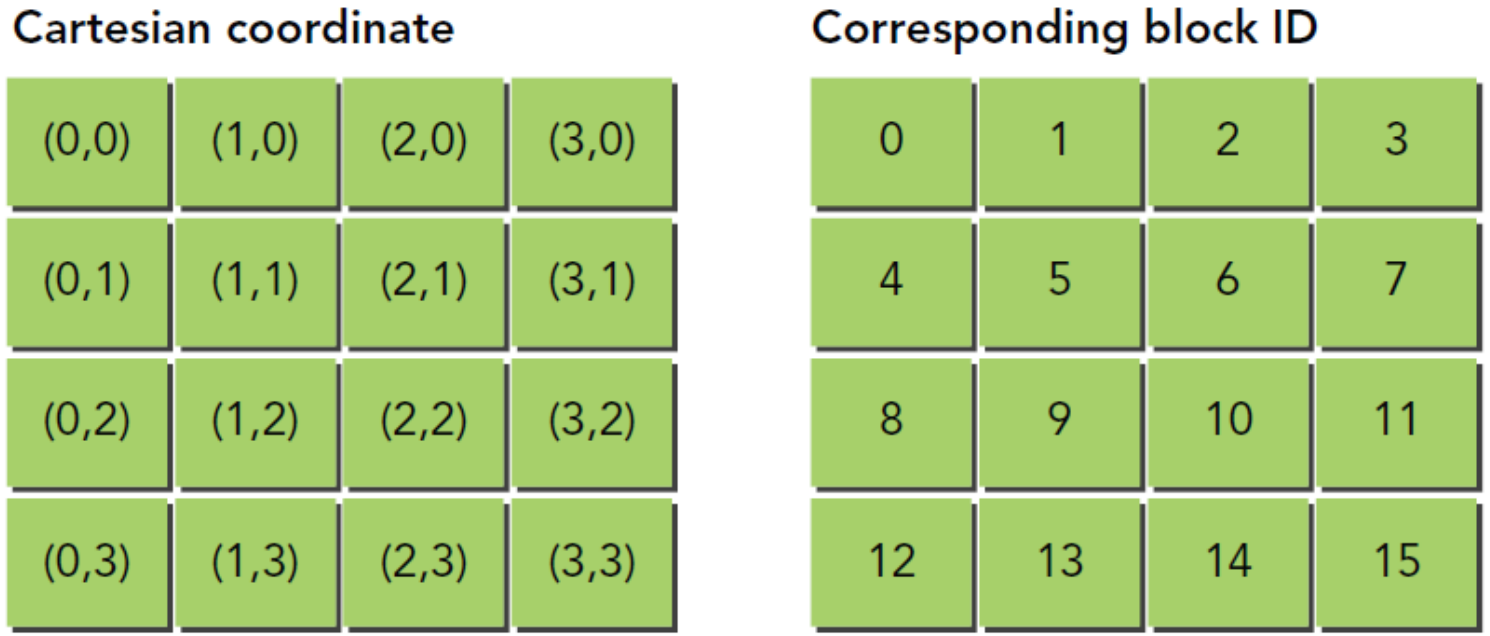


FIGURE 4-27

Coordonnées cartésiennes et diagonales

- **Différence dans les systèmes de coordonnées** :
  - Les coordonnées cartésiennes sont traditionnelles ; les coordonnées diagonales offrent une cartographie alternative.
- **Figure 4-28** : Démontre les coordonnées diagonales avec une nouvelle disposition de l'ID du bloc.

Diagonal coordinate

(0,0)	(0,1)	(0,2)	(0,3)
(1,3)	(1,0)	(1,1)	(1,2)
(2,2)	(2,3)	(2,0)	(2,1)
(3,1)	(3,2)	(3,3)	(3,0)

Corresponding block ID

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

FIGURE 4-28

Implémentation des coordonnées diagonales dans CUDA

- **Explication de la mise en correspondance** :
  - Les identifiants des blocs sont convertis en coordonnées cartésiennes à partir de la diagonale pour un accès précis aux données.
  - **Exemple de code** : Montre le calcul des coordonnées diagonales et le noyau de transposition pour l'accès par ligne.
- **Formule de cartographie** :
  - `block_x = (blockIdx.x + blockIdx.y) % gridDim.x ;`
  - `block_y = blockIdx.x ;`

Noyau de transposition diagonale basé sur les colonnes

- **Noyau alternatif** :
  - Noyau de transposition basé sur les colonnes et utilisant des coordonnées diagonales.
- **Exemple de code** : Noyau de transposition avec accès par colonne.
- **Commande d'exécution** : Exécutez `./transpose 6` pour une transposition basée sur les lignes et `./transpose 7` pour une transposition basée sur les colonnes.

Comparaison des performances (tableau 4-10)

- **Efficacité de la bande passante** :
  - Basé sur la rangée : 73,42 Go/s (41,32 % du pic)
  - Basé sur des colonnes : 75,92 Go/s (42,72 % du pic)
- **Conclusion** :
  - Le noyau basé sur les colonnes est plus performant grâce à une utilisation plus efficace de la partition de la mémoire.

Camping de partition et schémas d'accès à la mémoire

- **Partition Camping Issue** :
  - Les coordonnées cartésiennes entraînent une utilisation inégale des partitions de mémoire, ce qui conduit au camping des partitions.
  - Le mappage diagonal répartit les accès de manière plus homogène entre les partitions de la mémoire.
- **Figures 4-29 et 4-30** : montrent les schémas d'accès aux partitions pour les mappages cartésiens et les mappages diagonaux.

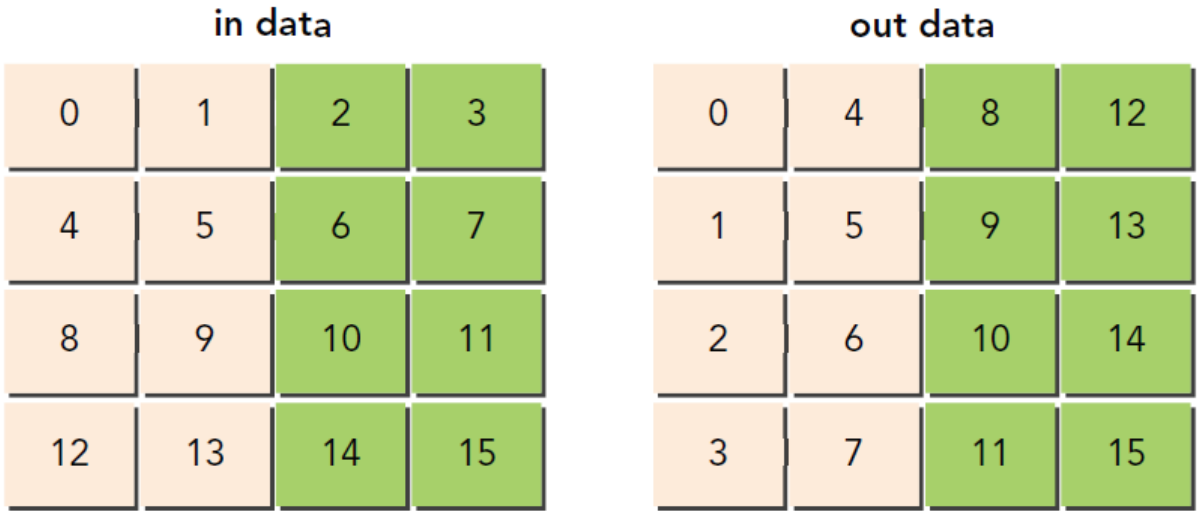


FIGURE 4-29

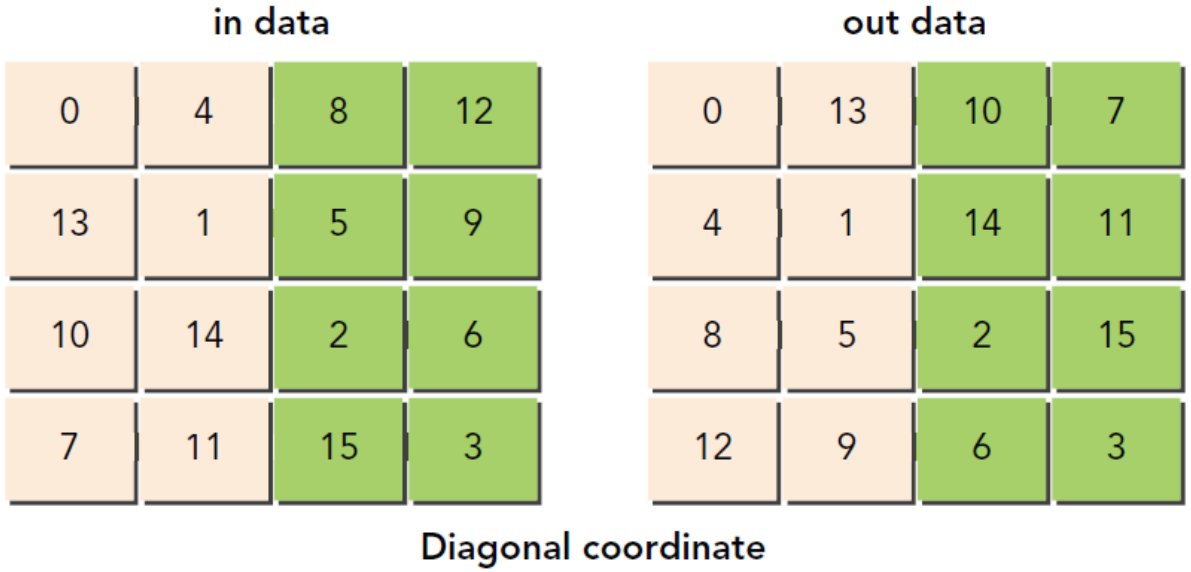


FIGURE 4-30

Exposer plus de parallélisme avec les Thin Blocks

- **Vue d'ensemble** :
  - L'ajustement de la taille des blocs est une stratégie efficace pour augmenter le parallélisme.
  - L'expérimentation du noyau `NaiveCol` (basé sur les colonnes) avec différentes tailles de blocs montre des différences de performance.
- **Tableau 4-11** : Bande passante effective des noyaux avec différentes tailles de blocs (cache L1 désactivé).

KERNEL	BLOCK SIZE	BANDWIDTH
NaiveCol	(32, 32)	38.13 GB/s
NaiveCol	(32, 16)	51.46 GB/s
NaiveCol	(32, 8)	54.82 GB/s
NaiveCol	(16, 32)	73.42 GB/s
NaiveCol	(16, 16)	80.27 GB/s
NaiveCol	(16, 8)	70.34 GB/s
NaiveCol	(8, 32)	102.76 GB/s
NaiveCol	(8, 16)	82.64 GB/s
NaiveCol	(8, 8)	59.59 GB/s

Taille optimale des blocs pour une largeur de bande maximale

- **Meilleure taille de bloc :**
  - La taille de bloc `(8, 32)` fournit la bande passante la plus élevée (102,76 Go/s), malgré le même parallélisme que `(16, 16)`.
  - - Cette amélioration est due à l'effet de bloc "fin", qui accroît l'efficacité des opérations en magasin.
- **Figure 4-31** : Représentation visuelle des blocs minces et de la largeur des blocs.

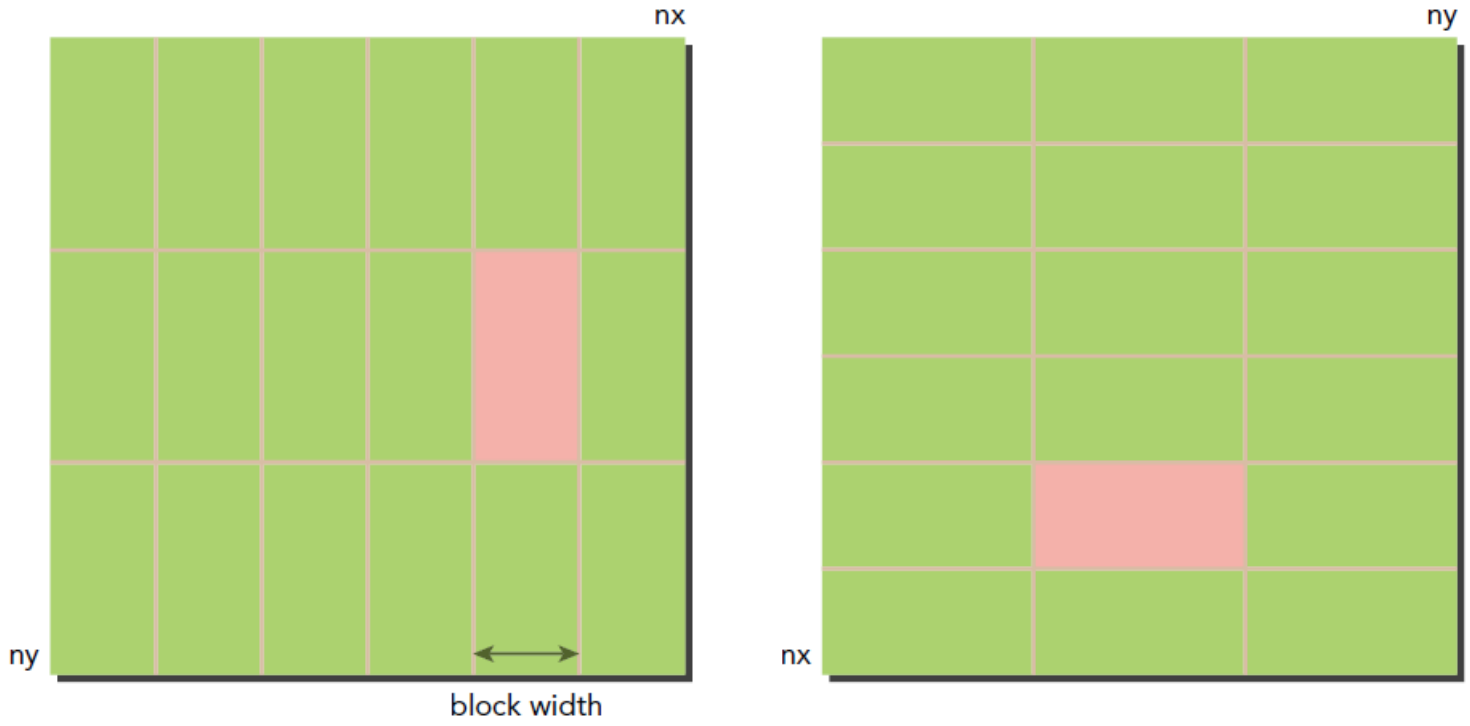


FIGURE 4-31

Comparaison des débits de chargement et de stockage

- **Utilisation de métriques pour le débit :**
  - - Mesuré à l'aide de `nvprof` pour évaluer le débit de chargement et de stockage avec différentes tailles de blocs.
- **Tableau 4-12** : Comparaison du débit de chargement et de stockage entre les tailles de blocs `(16, 16)` et `(8, 32)`.

EXECUTION CONFIGURATION	LOAD THROUGHPUT	STORE THROUGHPUT
(16, 16)	660.89 GB/s	41.11 GB/s
(8, 32)	406.43 GB/s	50.80 GB/s

Comparaison des performances des implémentations du noyau

- **Test de différents noyaux :**
  - Commandes pour l'exécution de différentes implémentations du noyau avec une taille de bloc `(8, 32)`.
- **Aperçu des performances :**
  - Le noyau `Unroll4Col` obtient la meilleure bande passante effective, surpassant le noyau copy.
- **Tableau 4-13 :** Bande passante effective de différents noyaux avec une taille de bloc `(8, 32)` (cache L1 désactivé).

KERNEL	BANDWIDTH	RATIO TO PEAK BANDWIDTH
Theoretical peak bandwidth	177.6	
`CopyRow` : Load/store using rows	102.30	57.57%
`NaiveRow` : Load rows/store columns	95.33	53.65%
`NaiveCol` : Load columns/store rows	101.99	57.39%
`Unroll4Row` : Load rows/store columns	82.04	46.17%
`Unroll4Col` : Load columns/store rows	113.36	63.83%

Conclusion et points clés à retenir

- **Thin Blocks et efficacité de la bande passante** :
  - Les configurations en blocs minces (comme `(8, 32)`) améliorent l'accès aux données et maximisent la bande passante.
  - Les optimisations du noyau (comme `Unroll4Col`) peuvent atteindre une bande passante élevée, proche de 60 à 80 % de la bande passante théorique maximale.
- **Implications pratiques** :
  - La taille optimale des blocs et les techniques de déroulement sont essentielles pour maximiser les performances du noyau CUDA.

Introduction à la mémoire unifiée

- **Objectif** : simplifier la gestion de la mémoire dans CUDA en combinant les espaces mémoire de l'hôte et du périphérique.
- **Concepts clés** :
  - La mémoire unifiée permet la migration automatique des données entre le CPU et le GPU.
  - Élimine le besoin de copies de mémoire explicites (par exemple, `cudaMemcpy`).
  - Les allocations de mémoire gérées réduisent les pointeurs en double pour l'hôte et l'appareil.

Mise en œuvre de l'addition matricielle avec la mémoire unifiée

- **Étapes de la mise en œuvre :**

1. Allocation de la mémoire gérée pour les matrices `A`, B et `gpuRef` :

```
cudaMallocManaged((void**)&A, nBytes) ;
```

2. Initialiser les matrices sur l'hôte.
3. Lancer le noyau CUDA avec des pointeurs de mémoire gérés :

```
sumMatrixGPU<<grid, block>>(A, B, gpuRef, nx, ny) ;
```

4. Synchroniser avec `cudaDeviceSynchronize()` pour s'assurer que les données sont prêtes sur l'hôte.

Comparaison des performances (mémoire gérée et mémoire non gérée)

- **Configuration du test** :
  - Exécutez les versions gérées et non gérées sur un GPU NVIDIA Tesla K40.
- **Résultats** :
  - **Mémoire gérée** :
    - Initialisation : Ralentissement dû à la migration initiale des données entre le CPU et le GPU.
    - Exécution du noyau : Légèrement plus lent mais simplifie le code.
  - **Mémoire non gérée** :
    - Nécessite des copies de mémoire explicites mais offre une initialisation plus rapide.
- **Mesure clé** : La mémoire unifiée simplifie le code avec un coût de performance minimal.

Profilage avec **nvprof**

- **Profilage de la mémoire unifiée** :
  - Utilisez `nvprof --unified-memory-profiling per-process-device ./managed` pour analyser les transferts de mémoire.
  - Suivi des transferts d'hôte à périphérique, de périphérique à hôte et des erreurs de page.
- **Perspectives** :
  - Temps de migration des données plus élevé pour les grandes matrices.
  - Les défauts de page se produisent lorsque le CPU accède à des données qui se trouvent sur le GPU, ce qui déclenche une migration des données.

Principaux enseignements

- **Avantages** :
  - La mémoire unifiée réduit la complexité de la programmation en automatisant la gestion de la mémoire.
  - Idéal pour les applications où la facilité de développement est prioritaire par rapport à la précision des performances.
- **Considérations** :
  - Peut entraîner des frais généraux supplémentaires pour les grands ensembles de données en raison des erreurs de page et des migrations de données.
  - Le profilage et l'optimisation sont essentiels pour maximiser les performances des applications à mémoire unifiée.

Résumé

- **Principales caractéristiques de la gestion de la mémoire CUDA**
  - **Contrôle de la hiérarchie de la mémoire du GPU** : Le modèle de programmation CUDA donne un accès direct à la hiérarchie de la mémoire du GPU, ce qui permet une gestion fine des données et des performances accrues.

  - **Caractéristiques de la mémoire globale** :
    - Mémoire la plus importante, latence la plus élevée.
    - Accès par des transactions de 32 ou 128 octets.
    - L'utilisation efficace de la mémoire globale est cruciale pour la performance des applications.

Résumé

Lignes directrices pour l'optimisation de l'utilisation de la bande passante

- **Stratégies clés** :
  1. **Maximisez les accès simultanés à la mémoire** : Gardez plusieurs demandes de mémoire en vol pour une plus grande efficacité.
  2. **Utiliser la mémoire sur puce** : Optimiser le transfert de données entre la mémoire globale et la mémoire interne.
- **Modèles d'accès à la mémoire** :
  - S'efforcer d'**aligner et de regrouper les accès** afin de minimiser les temps de latence.
  - Utiliser des techniques telles que le **déroulement** et les **ajustements de grille/blocs** pour améliorer le parallélisme.
  - **Camping de partition** : à éviter en gérant les schémas d'accès aux distorsions.
- **Mémoire unifiée** :
  - Simplifie la programmation en gérant automatiquement le transfert de données entre l'hôte et l'appareil.
  - CUDA 6.0+ maintient la cohérence des données, optimisant la facilité d'utilisation par rapport aux performances brutes.