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

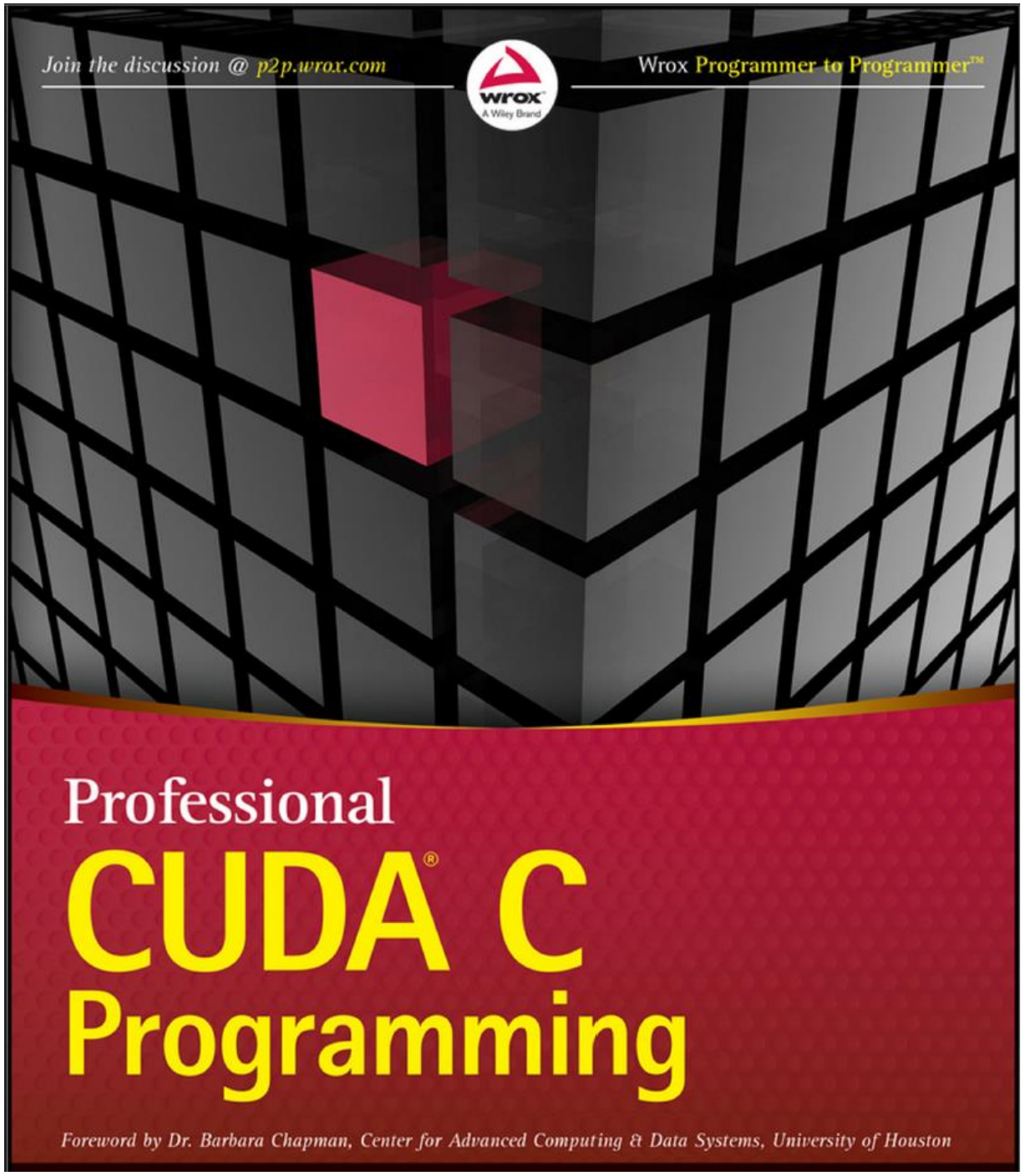
School of Electrical Engineering
and Computer Science

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 3 : Modèle d'exécution CUDA

Aperçu

- Développement de noyaux avec une approche axée sur les profils
- Comprendre la nature de l'exécution de la warp
- Exposer plus de parallélisme au GPU
- Maîtriser l'heuristique de configuration des grilles et des blocs
- Apprendre les différentes mesures et événements de performance CUDA
- Sonder le parallélisme dynamique et l'exécution imbriquée

Présentation du modèle d'exécution cuda

- Le modèle d'exécution CUDA expose une vue abstraite de l'architecture parallèle du GPU, vous permettant de raisonner sur la concurrence des threads.

- Au chapitre 2, vous avez appris que le modèle de programmation CUDA expose deux abstractions principales : une hiérarchie de mémoire et une hiérarchie de threads qui vous permettent de contrôler le GPU massivement parallèle.

- En conséquence, le modèle d'exécution CUDA fournit des informations utiles pour écrire un code efficace en termes de débit d'instructions et d'accès à la mémoire.

- Vous vous concentrerez sur le débit d'instructions dans ce chapitre et en apprendrez davantage sur les accès efficaces à la mémoire dans les chapitres 4 et 5.

Vue d'ensemble de l'architecture du GPU

- L'architecture du GPU est construite autour d'un réseau évolutif de multiprocesseurs de flux (SM).

- Le parallélisme matériel du GPU est obtenu par la réplication de ce bloc architectural.

- La figure 3-1 illustre les principaux composants d'un SM de Fermi :
  - Cœurs CUDA
  - Mémoire partagée/cache L1
  - Fichier d'enregistrement
  - Unités de chargement/stockage
  - Unités de fonction spéciales
  - Planificateur Warp

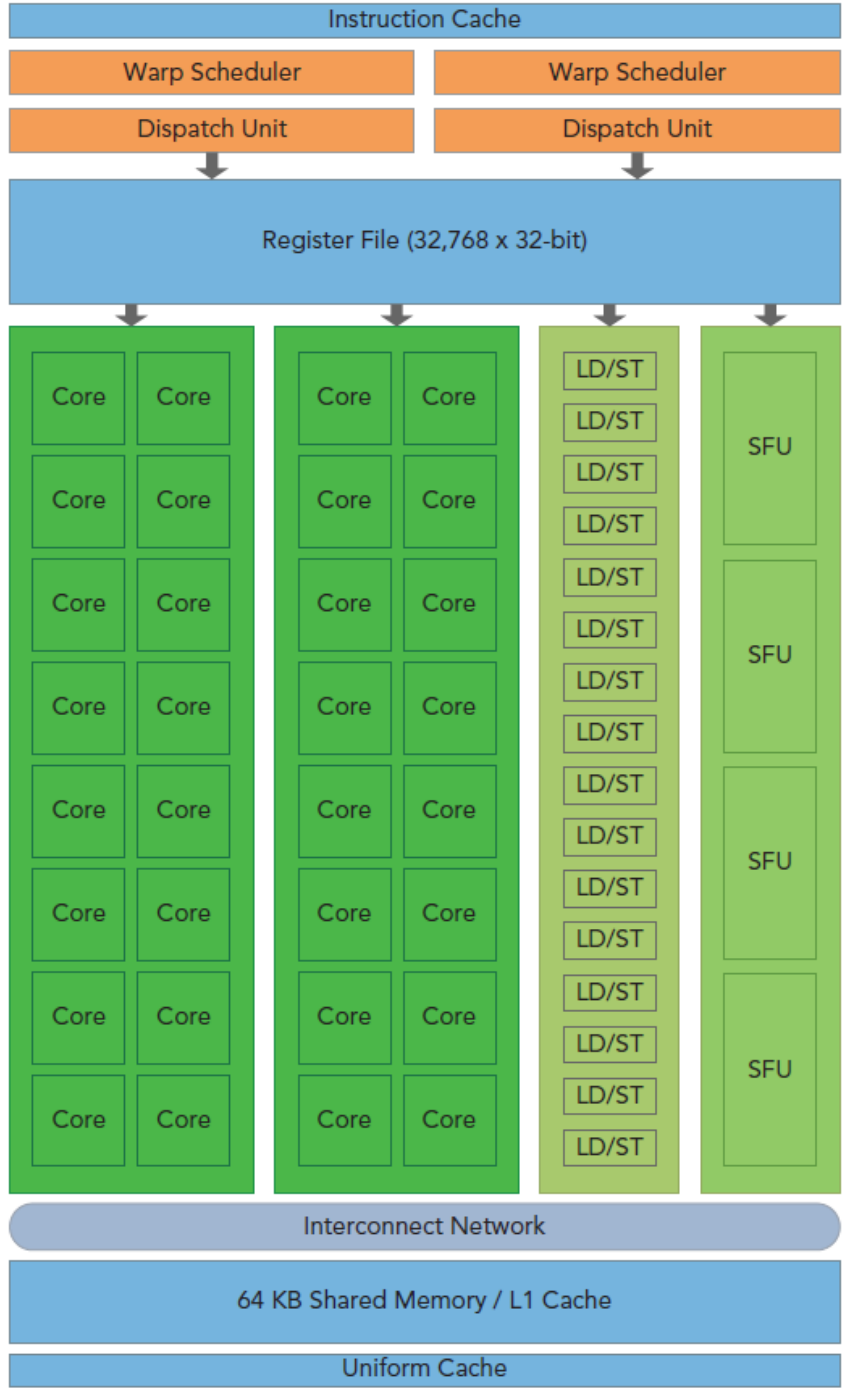

Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

FIGURE 3-1

Multiprocesseurs de flux (SM)

- Chaque SM d'un GPU est conçu pour prendre en charge l'exécution simultanée de centaines de threads, et il y a généralement plusieurs SM par GPU, de sorte qu'il est possible d'avoir des milliers de threads s'exécutant simultanément sur un seul GPU.

- Lorsqu'une grille de noyau est lancée, les blocs de threads de cette grille de noyau sont répartis entre les SM disponibles pour exécution.

- Une fois programmés sur un SM, les threads d'un bloc de threads ne s'exécutent simultanément que sur ce SM assigné.

- Plusieurs blocs de threads peuvent être assignés au même SM en même temps et sont programmés en fonction de la disponibilité des ressources du SM.

- Les instructions au sein d'un même thread sont mises en pipeline pour exploiter le parallélisme au niveau des instructions, en plus du parallélisme au niveau des threads que vous connaissez déjà avec CUDA.

Instruction unique et threads multiples (SIMT)

- CUDA utilise une architecture SIMT (Single Instruction Multiple Thread) pour gérer et exécuter des threads par groupes de 32, appelés warps.

- Tous les threads d'une warp exécutent la même instruction en même temps.

- Chaque thread possède son propre compteur d'adresses d'instructions et son propre état de registre, et exécute l'instruction en cours sur ses propres données.

- Chaque SM divise les blocs de threads qui lui sont attribués en 32 warps de threads qu'il planifie ensuite en vue de leur exécution sur les ressources matérielles disponibles.

Architecture SIMT

- L'architecture SIMT est similaire à l'architecture SIMD (Single Instruction, Multiple Data).
- SIMD et SIMT mettent en œuvre le parallélisme en diffusant la même instruction à plusieurs unités d'exécution.
- Une différence essentielle est que SIMD exige que tous les éléments vectoriels d'un vecteur s'exécutent ensemble dans un groupe synchrone unifié, alors que SIMT permet à plusieurs threads de la même warp de s'exécuter indépendamment.
- Même si tous les threads d'une warp démarrent ensemble à la même adresse de programme, il est possible que les threads individuels aient un comportement différent.
- SIMT vous permet d'écrire du code parallèle au niveau des threads pour des threads scalaires indépendants, ainsi que du code parallèle aux données pour des threads coordonnés.
- Le modèle SIMT comprend trois caractéristiques clés que SIMD n'a pas :
  - Chaque thread possède son propre compteur d'adresses d'instructions.
  - Chaque thread a son propre état de registre.
  - Chaque thread peut avoir un chemin d'exécution indépendant.

Composants correspondants de la vue logique et de la vue matérielle

- Un bloc de threads est programmé sur un seul SM.
- Une fois qu'un bloc de threads est programmé sur un SM, il y reste jusqu'à la fin de l'exécution.
- Un SM peut contenir plus d'un bloc de threads en même temps.
- La figure 3-2 illustre les composants correspondants de la vue logique et de la vue matérielle de la programmation CUDA.

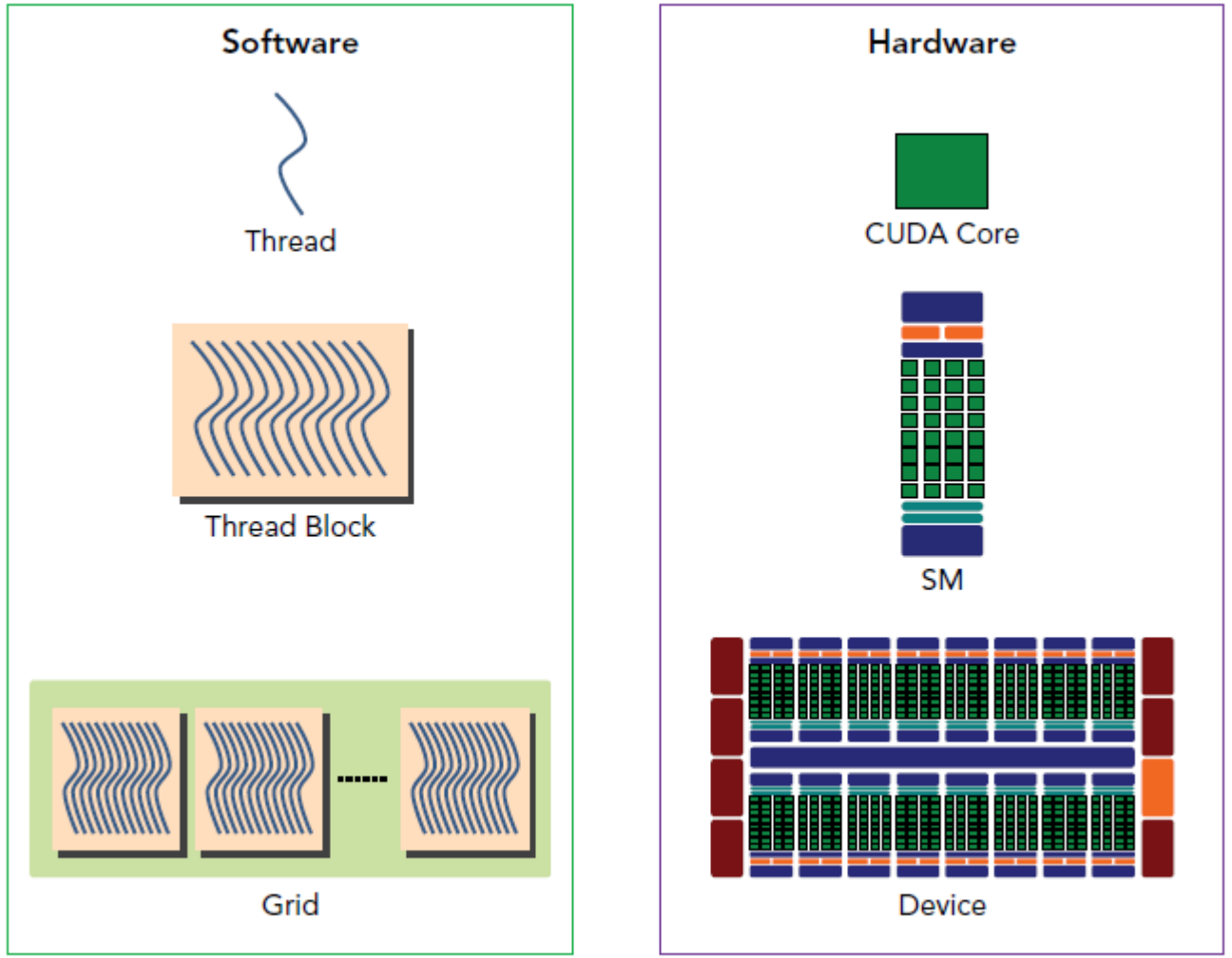


FIGURE 3-2

SM : Le cœur de l'architecture GPU

- Le multiprocesseur de flux (SM) est le cœur de l'architecture du GPU.
- Les registres et la mémoire partagée sont des ressources rares dans le SM.
- CUDA répartit ces ressources entre tous les threads résidant sur un SM.
- Par conséquent, ces ressources limitées imposent une restriction stricte sur le nombre de warps actives dans un SM, ce qui correspond à la quantité de parallélisme possible dans un SM.
- Connaître quelques éléments de base sur les composants matériels d'un SM vous aidera à organiser les threads et à configurer l'exécution du noyau afin d'obtenir les meilleures performances.

L'architecture Fermi

- La figure 3-3 illustre un schéma fonctionnel logique de l'architecture Fermi axée sur le calcul par le GPU, les composants spécifiques au graphisme étant largement omis.
- Fermi comprend jusqu'à 512 cœurs d'accélération, appelés cœurs CUDA.
- Chaque cœur CUDA dispose d'une unité logique arithmétique (ALU) entièrement en pipeline et d'une unité à virgule flottante (FPU) qui exécute une instruction entière ou à virgule flottante par cycle d'horloge.
- Les cœurs CUDA sont organisés en 16 multiprocesseurs de streaming (SM), chacun avec 32 cœurs CUDA.
- Fermi dispose de six interfaces de mémoire GDDR5 384 bits permettant d'atteindre un total de 6 Go de mémoire globale embarquée, une ressource de calcul essentielle pour de nombreuses applications.
- Une interface hôte relie le GPU au CPU via le bus PCI Express.
- Le moteur GigaThread (représenté en orange sur le côté gauche du diagramme) est un ordonnanceur global qui distribue les blocs de threads aux ordonnanceurs SM warp.

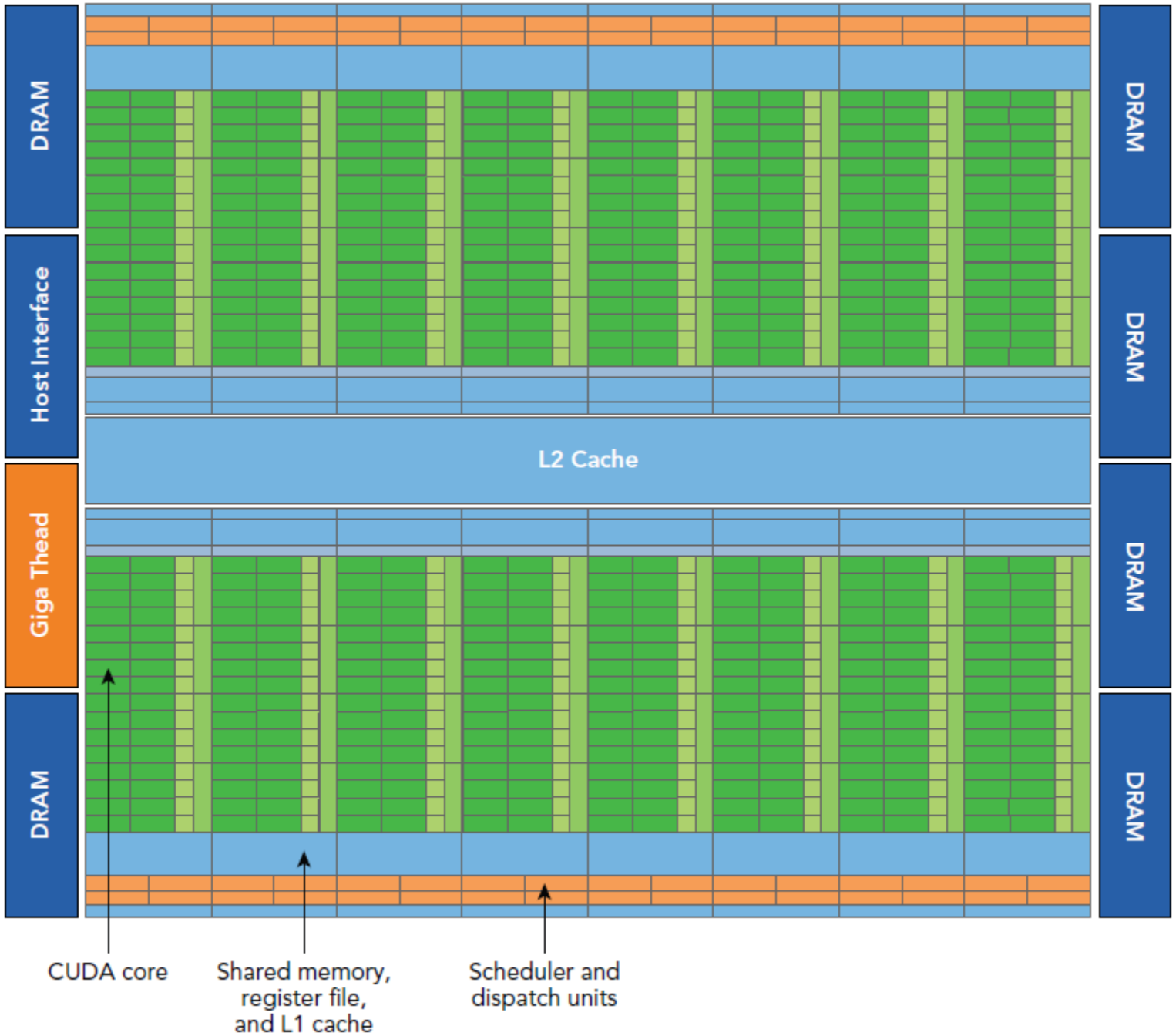

DRAM
Host Interface
Giga Thead
DRAM
DRAM
DRAM
DRAM
DRAM
L2 Cache
CUDA core
Shared memory, register file, and L1 cache
Scheduler and dispatch units

FIGURE 3-3

Multiprocesseurs de flux (SM)

- Chaque SM dispose de deux programmateurs de warp et de deux unités de distribution d'instructions.
- Lorsqu'un bloc de threads est attribué à un SM, tous les threads d'un bloc de threads sont divisés en warps.
- Les deux schedulers warp sélectionnent deux warps et émettent une instruction de chaque warp vers un groupe de 16 cœurs CUDA, 16 unités de chargement/stockage ou 4 unités de fonctions spéciales (illustré à la figure 3-4).
- L'architecture Fermi, capacité de calcul 2.x, peut gérer simultanément 48 warps par SM pour un total de 1 536 threads résidant dans un seul SM à la fois.

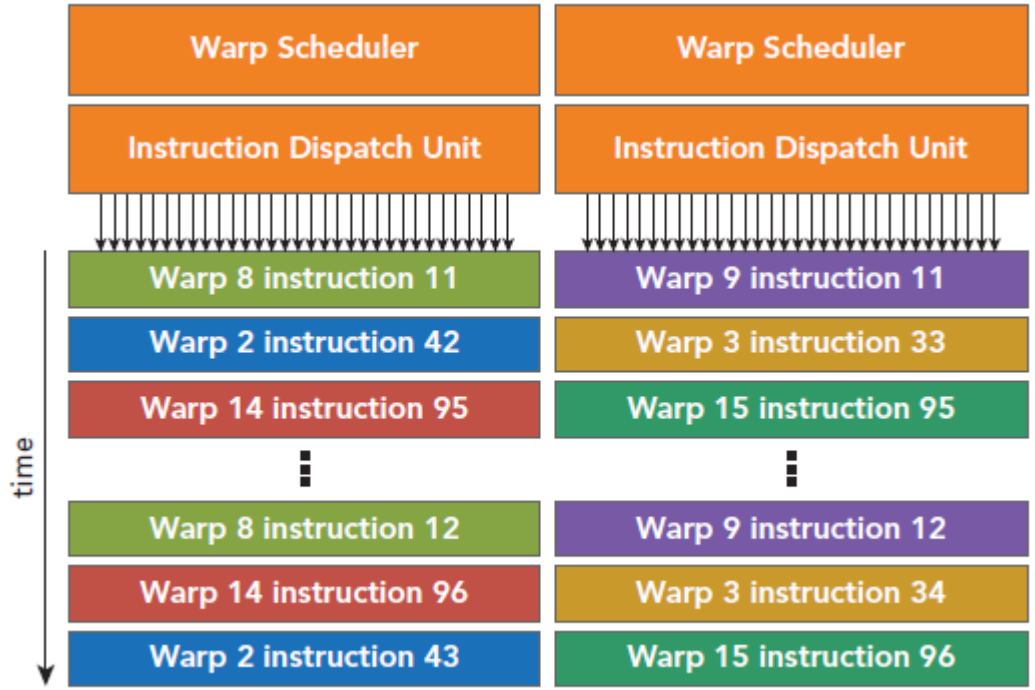


FIGURE 3-4

Exécution simultanée du noyau

- L'exécution simultanée de noyaux permet aux programmes qui exécutent un certain nombre de petits noyaux d'utiliser pleinement le GPU, comme l'illustre la figure 3-5.

- Fermi permet d'exécuter jusqu'à 16 noyaux en même temps sur l'appareil.

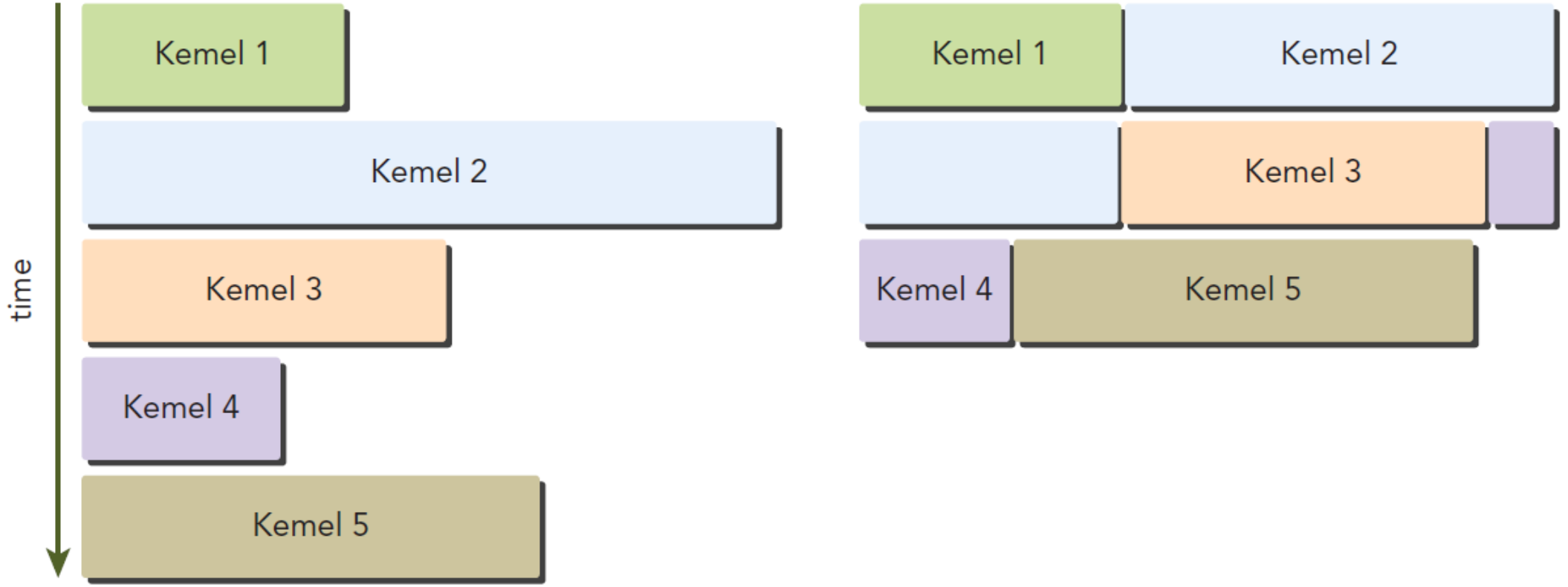


FIGURE 3-5

L'architecture Kepler

- La figure 3-6 illustre le schéma fonctionnel de la puce Kepler K20X, qui contient 15 multiprocesseurs en continu (SM) et six contrôleurs de mémoire 64 bits.

- L'architecture Kepler comporte trois innovations importantes :
  - SM améliorés
  - Parallélisme dynamique
  - Hyper-Q

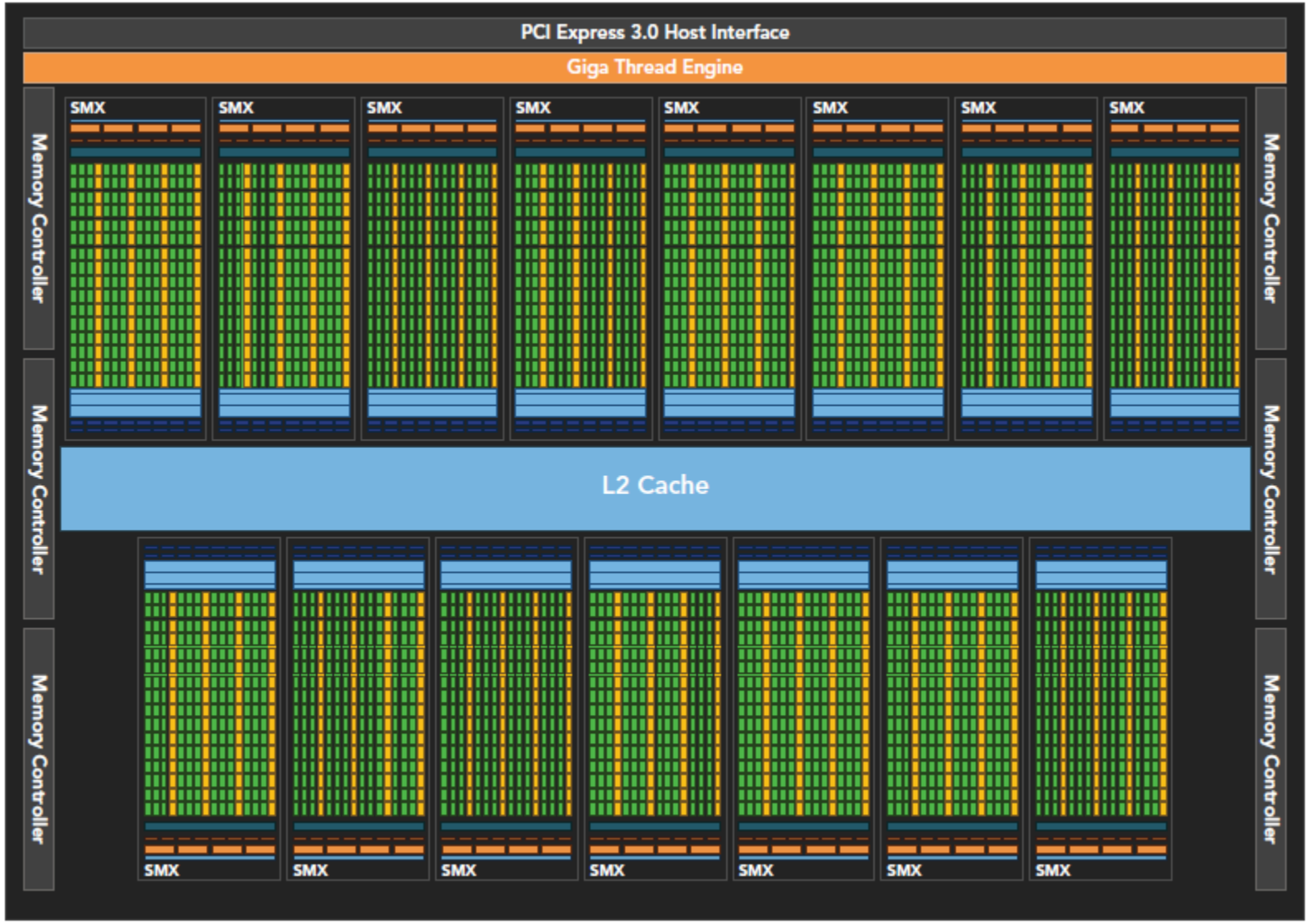


FIGURE 3-6

Multiprocesseurs en continu (SM)

- Au cœur du Kepler K20X se trouve une nouvelle unité SM, qui comprend plusieurs innovations architecturales améliorant la programmabilité et l'efficacité énergétique.

- Chaque unité SM Kepler se compose de 192 cœurs CUDA en simple précision, de 64 unités en double précision, de 32 unités de fonctions spéciales (SFU) et de 32 unités de chargement/stockage (LD/ST) (voir la figure 3-7).

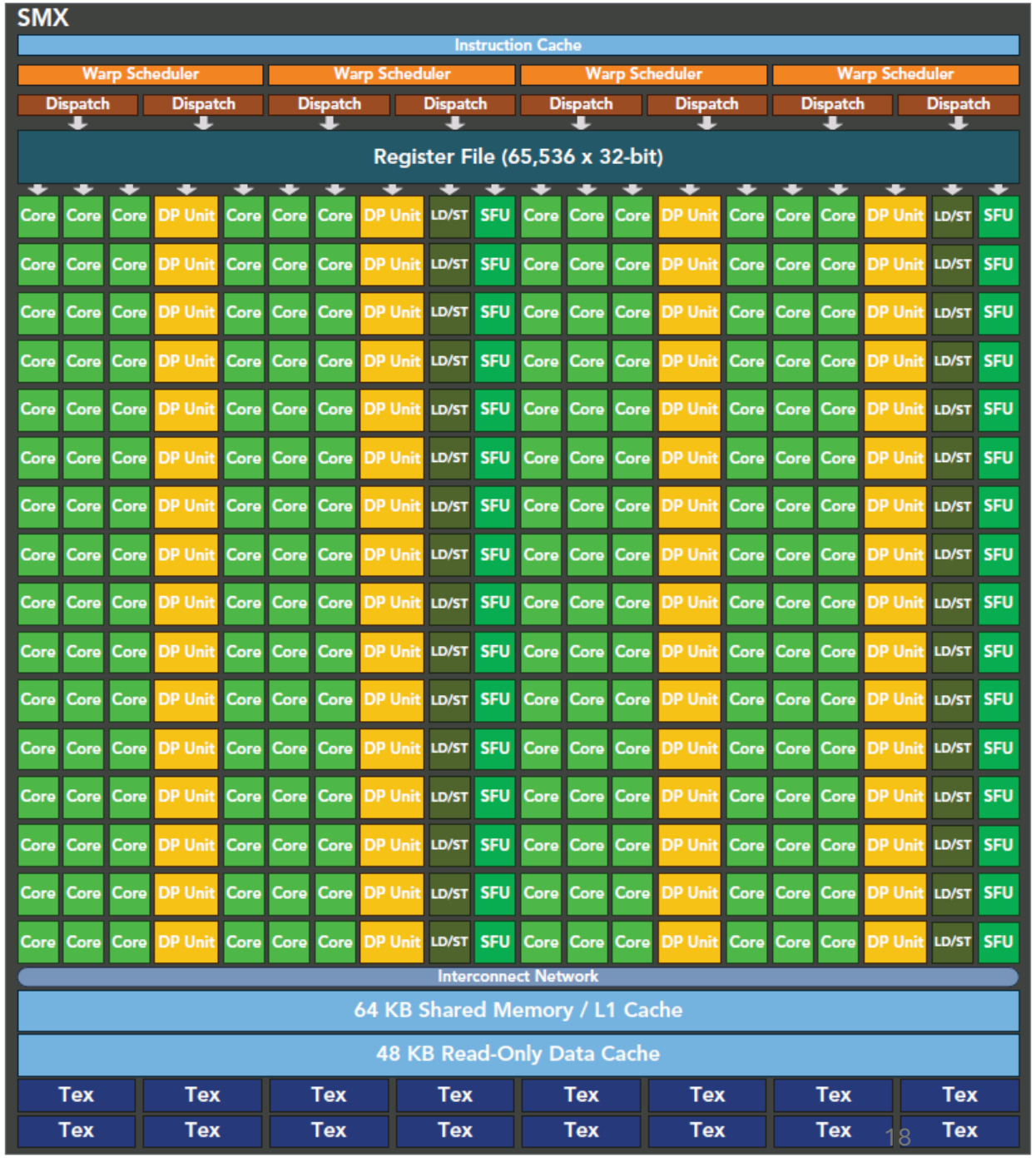


FIGURE 3-7

Parallélisme dynamique

- Le parallélisme dynamique est une fonctionnalité introduite avec les GPU Kepler qui permet au GPU de lancer de nouveaux noyaux de manière dynamique, ce qui permet à un noyau de gérer et de lancer d'autres noyaux.
- Il n'est donc pas nécessaire que l'unité centrale initie chaque lancement du noyau, ce qui facilite la mise en œuvre de schémas d'exécution récursifs et dépendants des données.
- En permettant le lancement de noyaux imbriqués directement à partir du GPU, cette fonctionnalité réduit le besoin de communication entre le GPU et le CPU, optimisant ainsi les performances pour certaines charges de travail.

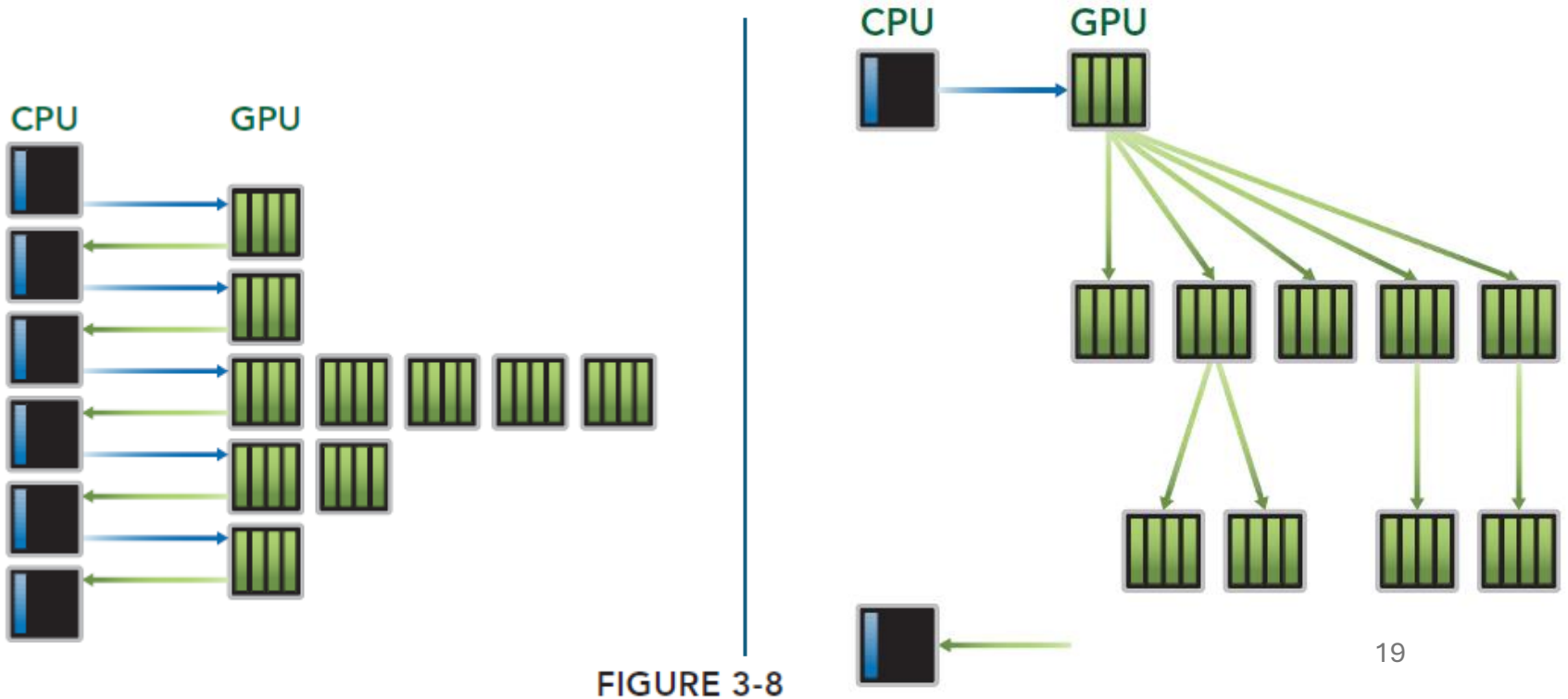


FIGURE 3-8

Parallélisme dynamique

- Hyper-Q est une fonction introduite avec les GPU Kepler qui augmente le nombre de connexions matérielles simultanées entre le CPU et le GPU, ce qui permet à plusieurs cœurs de CPU d'exécuter des tâches sur le GPU en même temps.
- Cela permet d'améliorer l'utilisation du GPU et de réduire le temps d'inactivité du CPU.
- En revanche, les GPU Fermi utilisent une seule file d'attente, ce qui peut entraîner le blocage des tâches les unes derrière les autres.
- Hyper-Q résout ce problème en fournissant 32 files d'attente matérielles, ce qui permet d'augmenter la concurrence, de maximiser l'utilisation du GPU et d'améliorer les performances globales.

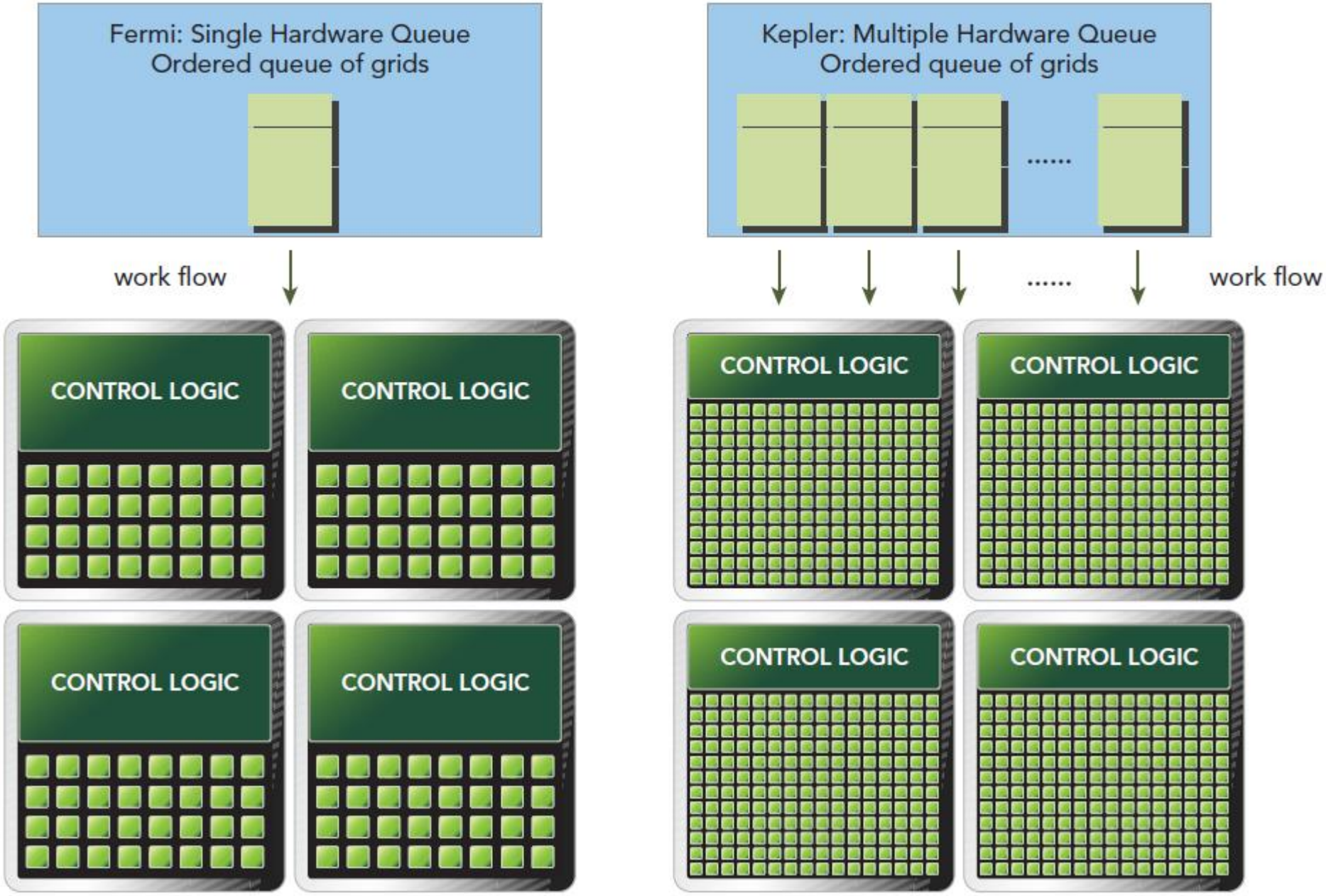


FIGURE 3-9

Optimisation en fonction du profil

- Le profilage est l'acte d'analyser la performance d'un programme en le mesurant :
  - L'espace (mémoire) ou la complexité temporelle du code de l'application
  - L'utilisation d'instructions particulières
  - La fréquence et la durée des appels de fonction

- Le profilage est une étape critique dans le développement d'un programme, en particulier pour optimiser le code d'une application de calcul à haute performance (HPC).
- Le profilage nécessite souvent une compréhension de base du modèle d'exécution d'une plateforme pour aider à prendre des décisions d'optimisation de l'application.
- Le développement d'une application HPC comporte généralement deux grandes étapes :
  1. Développer le code pour qu'il soit correct
  2. Améliorer le code pour le rendre plus performant

Optimisation en fonction du profil

Le développement axé sur les profils est essentiel dans la programmation CUDA pour plusieurs raisons :

1. Les implémentations naïves du noyau manquent souvent de performances optimales, et les outils de profilage permettent d'identifier les goulets d'étranglement critiques dans le code.
2. Le partitionnement des ressources de calcul de CUDA entre plusieurs blocs de threads peut limiter les performances.
3. Les outils de profilage permettent de savoir comment les ressources sont utilisées.
4. CUDA fait abstraction de l'architecture matérielle pour contrôler la simultanéité des threads.
5. Les outils de profilage permettent de mesurer, de visualiser et d'optimiser cette concurrence.

CUDA fournit deux outils de profilage principaux :

- `nvvp` : Un profileur visuel autonome.
- `nvprof` : Un profileur en ligne de commande.

Ces outils permettent d'analyser les performances du noyau et d'identifier les goulets d'étranglement.

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).
- Un seul cycle de profilage ne peut collecter que quelques compteurs, et certains compteurs s'excluent mutuellement, ce qui nécessite plusieurs cycles 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.

Connaître les détails des ressources matérielles

- Dans la programmation CUDA C, la compréhension des ressources matérielles est cruciale pour optimiser les performances du noyau.

- Si l'écriture d'un code correct ne nécessite pas une connaissance détaillée des caractéristiques de la mémoire cache, l'optimisation des performances, elle, en nécessite une.

- Même si le compilateur CUDA optimise le code dans une certaine mesure, une connaissance de base de l'architecture des GPU permet aux développeurs d'écrire un code plus efficace et de mieux utiliser le matériel.

- Les sections suivantes du chapitre abordent la manière dont les détails du matériel sont liés aux mesures de performance et comment ces mesures peuvent guider les efforts d'optimisation.

Comprendre la nature de l'exécution de la warp

1. Exécution parallèle de threads dans le lancement du noyau
   - Du point de vue du logiciel, il semble que tous les threads fonctionnent en parallèle.
   - Logiquement, c'est vrai, mais les limitations matérielles empêchent l'exécution simultanée de tous les threads.

2. Comprendre l'exécution Warp
   - Les threads sont regroupés en unités d'exécution appelées "warps".
   - Une warp se compose de 32 threads, exécutés en parallèle à l'intérieur de la warp.

3. Parallélisme matériel ou logique
   - Bien que logiquement tous les threads semblent s'exécuter ensemble, le matériel les traite par groupes (warps).
   - Tous les warps ne peuvent pas se dérouler en même temps en raison de contraintes de ressources.

4. Warp Execution Insights
   - Comprendre comment les warps s'exécutent au niveau matériel est essentiel pour optimiser la conception des noyaux.
   - Ces connaissances peuvent contribuer à améliorer les performances en guidant la gestion et l'exécution des threads.

Les warps et les blocages de threads

1. Les warps en tant qu'unités d'exécution de base
   - Les warps sont les unités fondamentales d'exécution au sein d'un multiprocesseur de flux (SM).

2. Distribution des blocs de threads
   - Lors du lancement des blocs de threads, ceux-ci sont répartis sur plusieurs SM.
   - Chaque bloc de thread est divisé en warps.

3. Structure de la warp
   - Une warp se compose de 32 threads consécutifs.
   - Les threads d'une warp s'exécutent ensemble.

4. Instruction unique, threads multiples (SIMT)
   - Les threads d'une warp exécutent la même instruction en parallèle.
   - Chaque thread effectue son opération sur ses propres données privées.

5. Vue logique et matérielle
   - Cette structure met en évidence la relation entre les blocs logiques de threads et leur exécution matérielle (voir la figure 3-10 dans la diapositive suivante).

Vue logique et matérielle

- Les blocs de threads dans CUDA peuvent être configurés en une, deux ou trois dimensions, mais le matériel les organise en une seule dimension.
- Chaque thread possède un identifiant unique au sein du bloc.
- Dans un bloc unidimensionnel, l'identifiant du thread est stocké dans threadIdx.x, et les threads consécutifs sont regroupés en warps.
- Par exemple, un bloc de 128 threads sera divisé en 4 warps, avec 32 threads dans chaque warp, où les threads sont regroupés par leurs valeurs threadIdx.x :
  - warp 0 : thread 0, thread 1, thread 2, ... thread 31
  - warp 1 : thread 32, thread 33, thread 34, ... thread 63
  - warp 2 : thread 64, thread 65, thread 66, ... thread 95
  - warp 3 : thread 96, thread 97, thread 98, ... thread 127

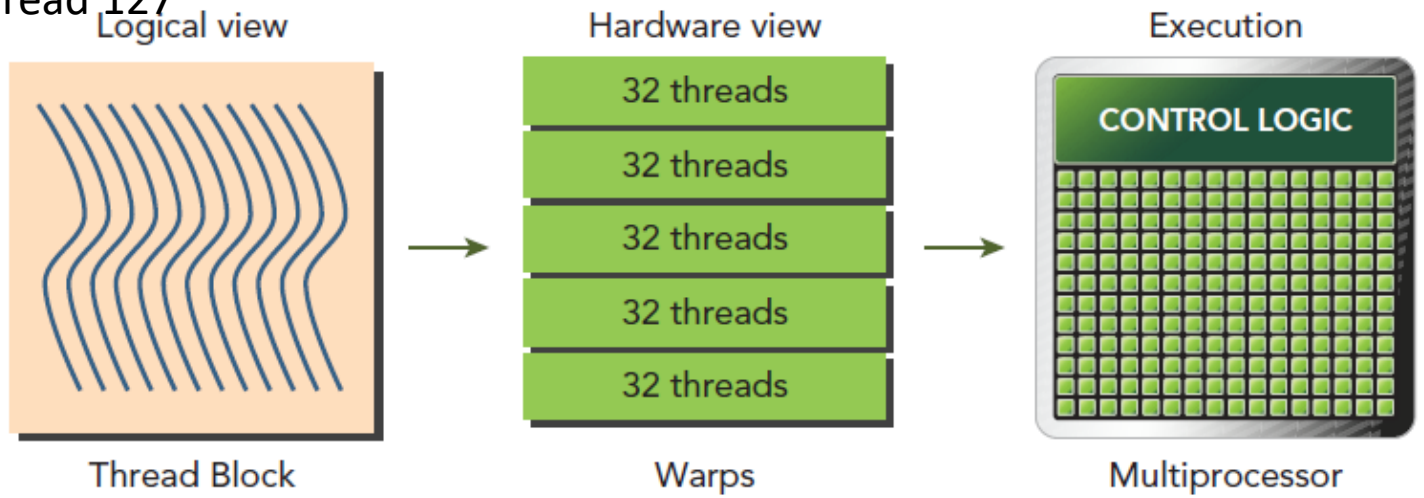


FIGURE 3-10

Vue logique et matérielle

1. Disposition logique des blocs de threads
   - Les blocs de threads peuvent être convertis d'une disposition logique multidimensionnelle en une disposition physique unidimensionnelle.
   - Dimensions :
     - x : le plus proche
     - y : deuxième dimension
     - z : ultrapériphérique

2. Calcul de l'identifiant unique du bloc de thread 2D
   ```
   threadIdx.y * blockDim.x + threadIdx.x
   ```

3. Calcul de l'identifiant unique du bloc de thread 3D
   ```
   threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x
   ```

4. Calcul du nombre de warps par bloc
   ```
   WarpsPerBlock = ceil(ThreadsPerBlock / warpSize)
   ```

5. Allocation Warp
   - Le matériel alloue un nombre discret de warps par bloc.
   - Les warps ne sont jamais divisées en blocs de threads.

Exemple : un bloc de threads en 2D avec 80 threads

Fils inactifs dans la dernière warp

- Si la taille du bloc de threads n'est pas un multiple exact de la taille de la warp, certains threads de la dernière warp peuvent être inactifs.
- Exemple : un bloc de threads 2D avec 80 threads nécessite 3 warps (96 threads matériels), bien que la dernière warp puisse avoir des threads inactifs (voir figure 3-11).

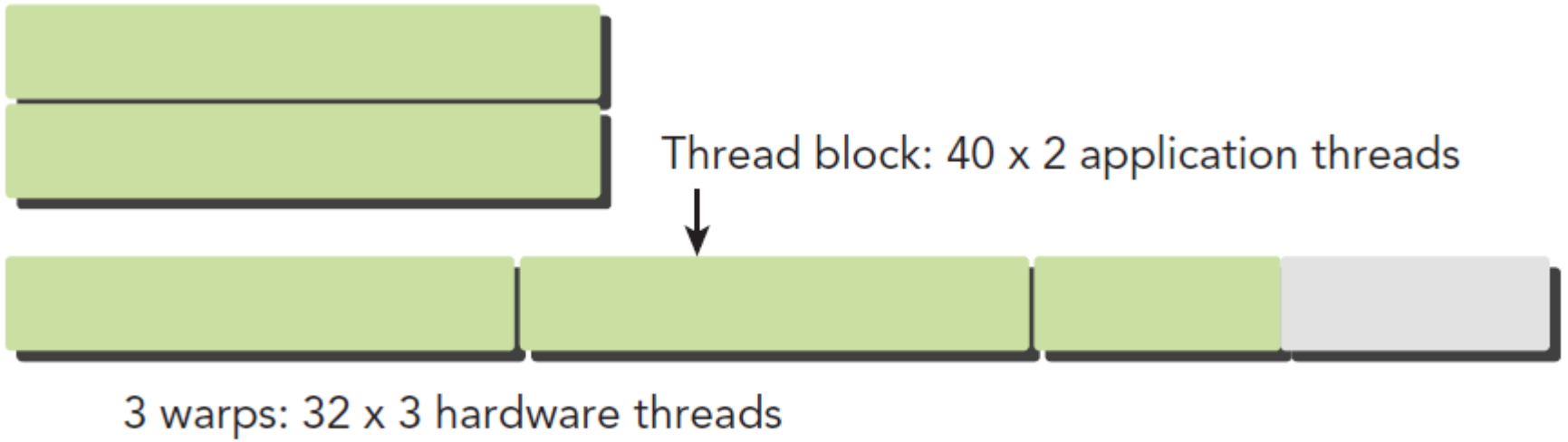


FIGURE 3-11

Bloc de thread : vue logique ou vue matérielle

- D'un point de vue logique, un bloc de threads est un ensemble de threads organisés selon une disposition 1D, 2D ou 3D.
- Du point de vue matériel, un bloc de thread est une collection 1D de warps.
- Les threads d'un bloc de threads sont organisés selon une disposition 1D, et chaque ensemble de 32 threads consécutifs forme une warp.

Divergence de la warp

1. Flux de contrôle dans la programmation
   - Les constructions de flux de contrôle (par exemple, "if", "for", "while") sont fondamentales dans la programmation.
   - Les GPU prennent en charge le contrôle de flux traditionnel de type C.

2. Prédiction des branches de l'unité centrale
   - Les unités centrales de traitement utilisent un matériel complexe de prédiction de branche pour prévoir le flux de contrôle.
   - Les prédictions correctes n'entraînent qu'une pénalité minimale en termes de performances.
   - Les prédictions incorrectes provoquent des blocages de l'unité centrale et des vidanges de pipeline.

3. Modèle d'exécution GPU
   - Les GPU ne disposent pas de mécanismes complexes de prédiction des branches comme les CPU.
   - Tous les threads d'une warp doivent exécuter des instructions identiques au cours du même cycle.

4. Divergence de la warp
   - Si un thread d'une warp exécute une instruction différente, tous les threads de la warp doivent également exécuter cette instruction.
   - Cela pose des problèmes si les threads d'une même warp empruntent des chemins de contrôle différents (par exemple, dans une instruction "if-else").

Divergence de la warp

- La figure 3-12 illustre la divergence de la warp.
- Tous les threads d'une warp doivent emprunter les deux branches de l'instruction `if...then`.
- Si la condition est `vraie` pour un thread, celui-ci exécute la clause `if` ; dans le cas contraire, le thread se bloque en attendant la fin de l'exécution.

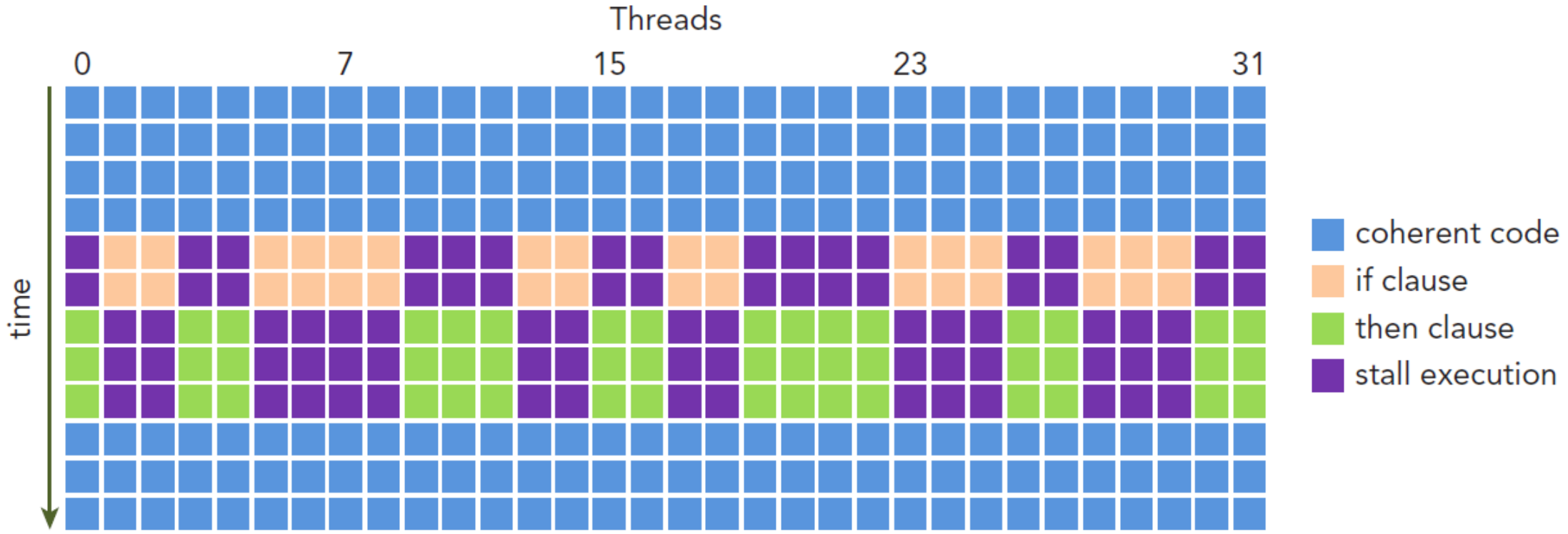


FIGURE 3-12

Exemple de code de divergence simple (1)

```
// Noyau à l'origine de la divergence de la warp
__global__ void simpleDivergence(int* data) {
    int tid = threadIdx.x ;

    // Condition simple : la moitié des threads prendront un chemin différent.
    if (tid < 16) {
        // La première moitié des threads effectue ceci
        data[tid] *= 2 ;
    }
    else {
        // La seconde moitié des threads effectue ceci
        data[tid] += 1 ;
    }
}
```

Exemple de code de divergence simple (2)

```
const int size = 32 ; // Taille de la déformation
    const int arraySize = size * sizeof(int) ;

    // Allocation de la mémoire de l'hôte
    int h_data[size] ;

    // Initialisation du tableau des hôtes
    for (int i = 0 ; i < size ; i++) {
        h_data[i] = i ;
    }

    // Allocation de la mémoire du dispositif
    int* d_data ;
    cudaMalloc((void**)&d_data, arraySize) ;

    // Copie des données de l'hôte vers l'appareil
    cudaMemcpy(d_data, h_data, arraySize, cudaMemcpyHostToDevice) ;
```

Exemple de code de divergence simple (3)

```
// Lancer le noyau avec un bloc de 32 threads (une seule warp)
    simpleDivergence <<<1, taille >>> (d_data) ;

    // Copie du résultat de l'appareil vers l'hôte
    cudaMemcpy(h_data, d_data, arraySize, cudaMemcpyDeviceToHost) ;

    // Imprimer le résultat
    printf("Result:\n") ;
    for (int i = 0 ; i < size ; i++) {
        printf("h_data[%d] = %d\n", i, h_data[i]) ;
    }

    // Libère la mémoire de l'appareil
    cudaFree(d_data) ;

    retour 0 ;
}
```

Exemple de code de divergence simple (4)

1. Aperçu du noyau :
    - Le noyau simpleDivergence est conçu pour introduire la divergence de warp.
    - Dans CUDA, chaque warp se compose de 32 threads et l'index des threads (tid) est utilisé pour déterminer le chemin d'exécution de chaque thread.
    - Les threads dont l'index est inférieur à 16 (tid < 16) exécuteront un bloc de code différent de ceux dont l'index est >= 16. Étant donné que les threads d'une warp doivent exécuter la même instruction simultanément, le fait d'avoir des chemins différents entraîne une divergence de la warp.

2. Code de l'hôte :
    - Nous créons un tableau de 32 entiers, nous l'initialisons avec des valeurs égales à leur index, puis nous transmettons ce tableau au noyau CUDA.
    - Après l'exécution du noyau, les valeurs mises à jour sont copiées sur l'hôte et imprimées.

Exemple de code de divergence simple (5)

1. Aperçu du noyau :
    - Le noyau simpleDivergence est conçu pour introduire la divergence de warp.
    - Dans CUDA, chaque warp se compose de 32 threads et l'index des threads (tid) est utilisé pour déterminer le chemin d'exécution de chaque thread.
    - Les threads dont l'index est inférieur à 16 (tid < 16) exécuteront un bloc de code différent de ceux dont l'index est >= 16. Étant donné que les threads d'une warp doivent exécuter la même instruction simultanément, le fait d'avoir des chemins différents entraîne une divergence de la warp.

2. Code de l'hôte :
    - Nous créons un tableau de 32 entiers, nous l'initialisons avec des valeurs égales à leur index, puis nous transmettons ce tableau au noyau CUDA.
    - Après l'exécution du noyau, les valeurs mises à jour sont copiées sur l'hôte et imprimées.

explication du fichier simpleDivergence.cu (6)

1. Warp Divergence :
    - Les threads de la warp sont divisés en deux groupes, ce qui entraîne une divergence.
    - Alors que le premier groupe multiplie ses valeurs par 2, le second groupe y ajoute 1.
    - En raison de ce branchement conditionnel, la warp doit exécuter les deux chemins séquentiellement pour les deux groupes, ce qui réduit l'efficacité de l'exécution parallèle.

Lecture_3_exemple_4 : simpleDivergence.cu

Optimiser les performances de CUDA : Éviter la divergence Warp

1. Éviter la divergence de warp :
   - Pour des performances optimales, veillez à ce que tous les threads d'une même warp suivent le même chemin d'exécution.

2. Affectation à la warp :
   - L'affectation des threads à la warp dans un bloc est déterministe.
   - Il est difficile mais possible de partitionner les données de manière à ce que tous les threads d'une même warp suivent le même flux de contrôle.

3. Exemple de divergence de branche :
   - Considérons deux branches dans un noyau.
   - Un mauvais partitionnement peut entraîner des divergences entre les threads d'une même warp, c'est-à-dire des threads qui suivent des chemins différents.

4. Partitionnement pair/impair des threads :
   - Exemple de condition : tid % 2 == 0
   - Les threads de discussion dont le numéro est pair prennent la branche if.
   - Les threads impairs prennent la branche "else".
   - Cela entraîne une divergence de la warp, car les threads d'une même warp exécutent des branches différentes.

Rappels clés

- La divergence de la warp se produit lorsque les threads d'une warp empruntent des chemins de code différents.
- Les différentes branches if-then-else sont exécutées en série.
- Essayez d'ajuster la granularité des branches pour qu'elle soit un multiple de la taille de la warp afin d'éviter la divergence des warps.
- Différents warps peuvent exécuter des codes différents sans que les performances n'en pâtissent.

Partitionnement des ressources (1)

- Le contexte d'exécution local d'une warp de production se compose principalement des ressources suivantes :
  - Compteurs de programmes
  - Registres
  - Mémoire partagée
- Le contexte d'exécution de chaque warp traité par un SM est maintenu sur la puce pendant toute la durée de vie du warp.
- Par conséquent, le passage d'un contexte d'exécution à un autre n'entraîne aucun coût.

Partitionnement des ressources (2)

- Chaque multiprocesseur de flux (SM) possède un ensemble de registres de 32 bits stockés dans un fichier de registres.
- Les registres sont répartis entre les différents threads.
- Une quantité fixe de mémoire partagée est disponible, répartie entre les blocs de threads.
- Le nombre de blocs de thread et de warps qui peuvent résider sur un SM dépend de ce qui suit :
  - Nombre de registres disponibles.
  - Quantité de mémoire partagée disponible sur le SM.
  - Exigences du noyau en matière de registres et de mémoire partagée.

Répartition des ressources (3)

1. Registres par SM :
    - Kepler : 64K registres
    - Fermi : 32K registres
2. Plus de threads avec moins de registres par thread :
    - Permet le traitement simultané d'un plus grand nombre de bandes.
3. Moins de threads avec plus de registres par thread :
    - Le traitement simultané d'un moins grand nombre de bandes est possible.
4. Mémoire partagée par SM :
    - Kepler et Fermi : Jusqu'à 48K de mémoire partagée.
5. Plus de blocs avec moins de mémoire partagée par bloc :
    - Permet à plus de blocs de threads d'être traités simultanément par le SM.
6. Moins de blocs et plus de mémoire partagée par bloc :
    - Limite le nombre de blocs de threads qui peuvent être traités simultanément.

Partitionnement des ressources (4)

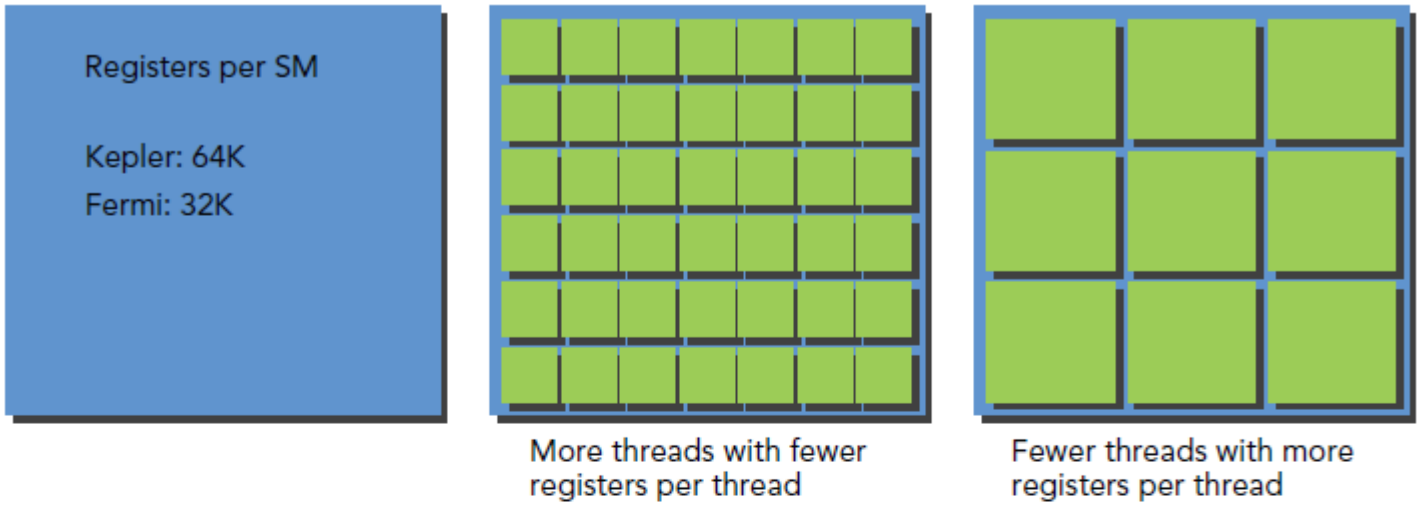


FIGURE 3-13

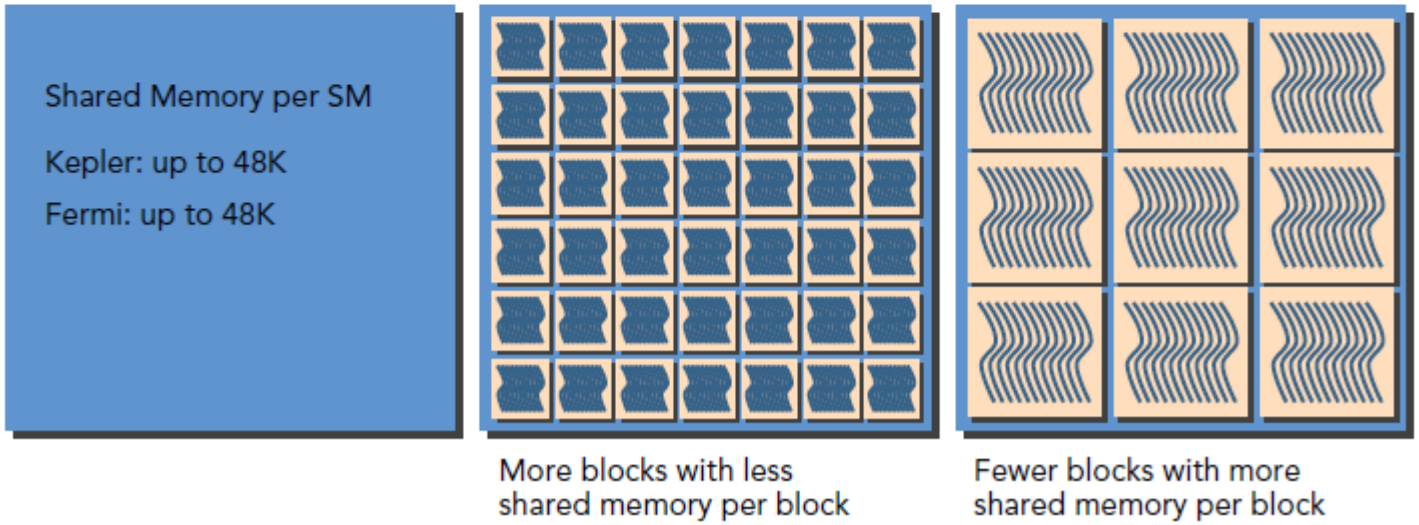


FIGURE 3-14

Partitionnement des ressources (5)

TABLEAU 3-2 : Limites de ressources relatives à la capacité de calcul

TECHNICAL SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Maximum number of threads per block	1,024			
Maximum number of concurrent blocks per multiprocessor	8		16	
Maximum number of concurrent warps per multiprocessor	48		64	
Maximum number of concurrent threads per multiprocessor	1,536		2,048	
Number of 32-bit registers per multiprocessor	32 K		64 K	
Maximum number of 32-bit registers per thread	63		255	
Maximum amount of shared memory per multiprocessor	48 K			

Capacité de calcul

- Pour les GPU ayant une capacité de calcul de 8.9, comme la série GeForce RTX 40xx (y compris les RTX 4090, 4080 et autres), les spécifications des ressources diffèrent considérablement des générations précédentes comme 3.0 ou 3.5.
- Voici les principales caractéristiques techniques de ces GPU modernes :
  - Nombre maximum de threads par bloc : Jusqu'à 1 024 threads (comme pour les générations précédentes).
  - Nombre maximum de blocs simultanés par multiprocesseur de streaming (SM) : Jusqu'à 32 blocs, contre 16 dans les modèles précédents comme compute capability 3.0 (NVIDIA Docs) (NVIDIA Developer).
  - Mémoire partagée maximale par multiprocesseur : Augmentation à 164 Ko pour certains modèles, contre 48 Ko pour les anciens GPU (NVIDIA Developer).
  - Nombre de registres 32 bits par SM : ce nombre est passé à 64 000 ou plus dans les GPU modernes, en fonction des configurations spécifiques (NVIDIA Developer).
  - Nombre maximum d'éléments par multiprocesseur : Capacité de 64 warps par SM, ce qui est plus efficace pour le parallélisme (AI Product Scaling).
  - Ces améliorations permettent de meilleures performances dans des tâches telles que l'apprentissage profond, le ray tracing et le calcul haute performance (HPC), en utilisant des fonctionnalités telles que les Tensor Cores et des opérations à virgule flottante améliorées pour la capacité de calcul.

Vue d'ensemble de l'ordonnancement Warp(1)

- Bloc actif : Un bloc auquel ont été attribuées des ressources telles que des registres et de la mémoire partagée.
- Warps actifs : Warps contenus dans un bloc actif.
- Les warp actives sont classées comme suit :
  - warp de production sélectionnée : Actuellement en cours d'exécution sur les cœurs CUDA.
  - warp bloquée : Pas prêt pour l'exécution.
  - Déformation éligible : Prêt à exécuter mais en attente de disponibilité.

Vue d'ensemble de l'ordonnancement Warp(2)

- Planification de la warp de production :
  - Chaque multiprocesseur de flux (SM) sélectionne les warps à chaque cycle et les distribue aux unités d'exécution.
  - Warp sélectionnée : Une warp qui s'exécute activement.
  - Warp éligible : Prêt à être exécuté mais pas encore sélectionné.
  - Warp bloqué : Pas prêt en raison de dépendances non satisfaites ou d'une indisponibilité des ressources.

Vue d'ensemble de l'ordonnancement Warp(3)

- Conditions d'exécution des warp : Un warp est éligible à l'exécution si
  - 32 cœurs CUDA sont disponibles.
  - Tous les arguments de l'instruction en cours sont prêts.
- Exemple - Architecture Kepler :
  - Maximum 64 warps simultanés sur un Kepler SM.
  - A chaque cycle, jusqu'à 4 warps peuvent être sélectionnées pour être exécutées.
  - Si une warp se bloque, le planificateur de warps sélectionne une warp éligible pour la remplacer.
- La clé d'une programmation CUDA efficace :
  - Minimiser les blocages de warp en maintenant un grand nombre de warps actives.
  - Maximiser les warps actives pour masquer la latence causée par les décrochages de warp et optimiser l'utilisation du GPU.

Masquage des temps de latence (1)

- Qu'est-ce que le masquage de latence ?
  - Dans la programmation CUDA, la latence fait référence au délai entre l'émission d'une instruction et sa réalisation.
  - Le masquage de la latence garantit que les ressources de calcul du GPU sont pleinement utilisées en exécutant les instructions d'autres warps pendant qu'un warp attend des données ou des résultats.
- Concept clé :
  - Sur un multiprocesseur de flux (SM), le parallélisme au niveau des threads maximise l'utilisation de ses unités fonctionnelles.
  - La latence est masquée par le maintien de plusieurs warps actifs et la sélection des warps éligibles à l'exécution pendant que les autres sont bloqués.
  - La latence des instructions est masquée par les calculs effectués par d'autres groupes.

Masquage des temps de latence (2)

- Comparaison avec la programmation par CPU :
  - Les processeurs s'efforcent de minimiser la latence pour un ou deux threads à la fois.
  - Les GPU sont conçus pour gérer un grand nombre de threads légers afin de maximiser le débit.
  - L'objectif est d'exécuter plusieurs warps simultanément pour masquer les temps de latence.
- Types de latence :
  - Temps de latence des instructions arithmétiques :
    - Temps écoulé entre le début d'une opération arithmétique et le moment où son résultat est disponible.
  - 10 à 20 cycles d'horloge pour les opérations arithmétiques.
  - Latence d'instruction de la mémoire :
    - Temps écoulé entre l'émission d'une opération de mémoire et la réception des données.
  - 400-800 cycles d'horloge pour l'accès à la mémoire globale.

Masquage des temps de latence (3)

- Ordonnancement Warp et dissimulation des temps de latence :
  - Le planificateur de warp sélectionne dynamiquement les warps éligibles à l'exécution.
  - Si un warp est bloqué (par exemple, en attente de mémoire), le planificateur choisit d'autres warps à exécuter, ce qui garantit une utilisation continue des ressources de calcul.

- Exemple (figure 3-15) :
  - Warp 0 se bloque en raison d'une opération de mémoire à longue durée de vie.
  - Le planificateur de warp exécute le Warp 2, le Warp 3 et d'autres warps jusqu'à ce que le Warp 0 soit à nouveau éligible pour être exécuté.
  - Ce processus minimise le temps d'inactivité du SM.

Masquage des temps de latence (4)

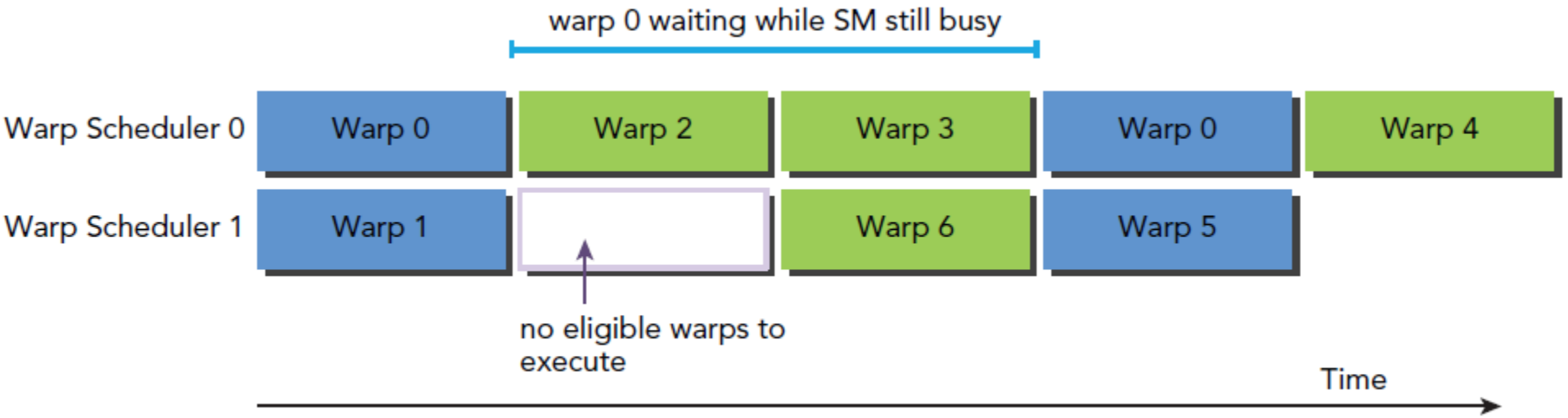


FIGURE 3-15

Masquage des temps de latence (5)

- Comment estimer le nombre de warps nécessaires ?
  - La loi de Little peut être appliquée pour estimer le nombre de warps actives nécessaires pour masquer la latence :
    - Nombre de séquences requises = temps de latence × débit

- Cette formule permet de calculer le nombre optimal de warps nécessaires pour que le GPU soit pleinement utilisé et que la latence soit masquée de manière efficace.

Loi de Little (1)

- Visualisation de la loi de Little (figure 3-16) :
  - La loi de Little montre la relation entre la latence et le débit.
  - Exemple : Si la latence moyenne pour une instruction est de 5 cycles, et que vous visez un débit de 6 warps par cycle, vous avez besoin d'au moins 30 warps en cours pour que le GPU soit pleinement utilisé.

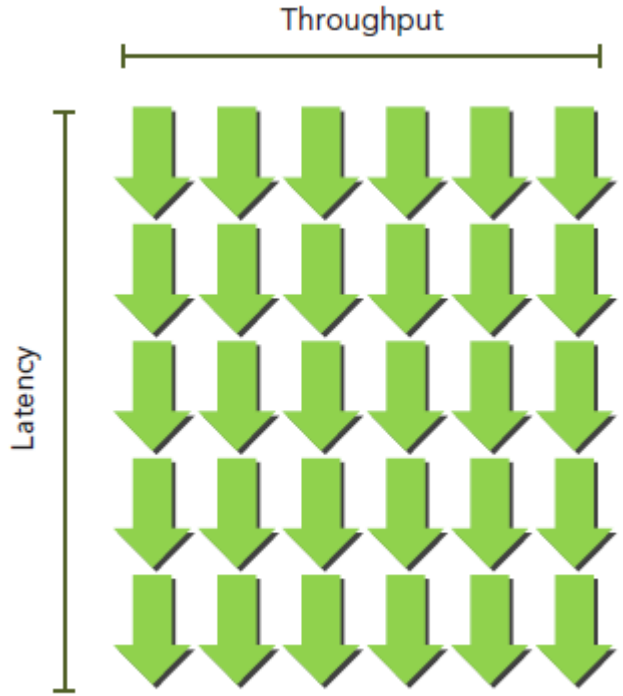


FIGURE 3-16

Loi de Little (2)

- Débit et largeur de bande :
  - Débit : Le taux d'opérations effectuées par unité de temps (par exemple, les déformations exécutées par cycle).
  - Bande passante : taux de transfert de données le plus élevé possible, se référant généralement à la mémoire ou au transfert d'E/S.

  - Ces deux indicateurs sont souvent utilisés pour mesurer les performances, mais ils se concentrent sur des aspects différents :
    - Débit : Le nombre d'opérations ou d'instructions exécutées par unité de temps.
    - Bande passante : taux de transfert de données théoriques ou réels maximums.

Loi de Little (3)

- Opérations arithmétiques et parallélisme :
  - Pour les opérations arithmétiques, le parallélisme requis est exprimé par le nombre d'opérations nécessaires pour masquer la latence arithmétique.
  - L'exemple fourni montre une opération d'addition multipliée en virgule flottante de 32 bits (a + b × c), illustrant comment le débit arithmétique varie en fonction du nombre d'opérations par cycle d'horloge par multiprocesseur de flux (SM).

Maintien d'une utilisation complète de l'arithmétique et de la mémoire (1)

- Débit arithmétique et parallélisme :
  - Pour maintenir une utilisation arithmétique complète sur un SM, le nombre d'opérations par cycle doit correspondre à la capacité de débit du GPU.
  - Parallélisme au niveau de la warp :
    - Une warp exécutant une instruction correspond à 32 opérations.
    - Par conséquent, le nombre requis de warps est basé sur le débit divisé par le nombre d'opérations par warp.
  - L'architecture Fermi, par exemple, nécessite 640 opérations par cycle, ce qui correspond à 20 warps pour maintenir une pleine utilisation (comme le montre le tableau 3-3).

- Deux façons d'augmenter le parallélisme :
  - Parallélisme au niveau des instructions (ILP) :
    - Plus d'instructions indépendantes au sein d'un même thread.
  - Parallélisme au niveau des threads (TLP) :
    - Plus de threads éligibles simultanément à l'exécution.

Maintien d'une utilisation complète de l'arithmétique et de la mémoire (2)

TABLEAU 3-3 : Parallélisme SM nécessaire pour maintenir une utilisation arithmétique complète

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	THROUGHPUT (OPERATIONS/CYCLE)	PARALLELISM (OPERATIONS)
Fermi	20	32	640
Kepler	20	192	3,840

Maintien d'une utilisation complète de l'arithmétique et de la mémoire (3)

- Débit de mémoire et parallélisme :
  - Le parallélisme de la mémoire est défini par le nombre d'octets nécessaires pour masquer la latence de la mémoire.
  - Le débit de la mémoire est généralement exprimé en gigaoctets par seconde (Go/s), mais il doit être converti en octets par cycle pour calculer le parallélisme requis pour les opérations liées à la mémoire.

- Exemple de calcul :
  - Un GPU Fermi avec une bande passante mémoire de 144 Go/sec et une fréquence de 1,566 GHz produit 92 octets par cycle.
  - À l'aide de ces informations, le parallélisme nécessaire peut être calculé pour maintenir une utilisation complète de la mémoire pour le périphérique (comme indiqué dans le tableau 3-4).

Maintien d'une utilisation complète de l'arithmétique et de la mémoire (3)

TABLEAU 3-4 : Parallélisme des dispositifs nécessaire pour maintenir une utilisation complète de la mémoire

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	BANDWIDTH (GB/SEC)	BANDWIDTH (B/CYCLE)	PARALLELISM (KB)
Fermi	800	144	92	74
Kepler	800	250	96	77

Valeurs du tableau 3-3 (1)

- Temps de latence des instructions (cycles) :
  - La latence des instructions représente le nombre de cycles d'horloge nécessaires pour effectuer une opération arithmétique donnée.
  - Pour les architectures Fermi et Kepler, il s'agit de 20 cycles.
  - Cette valeur est généralement déterminée par les caractéristiques matérielles de l'architecture du GPU.

- Débit (opérations/cycle) :
  - Le débit correspond au nombre d'opérations arithmétiques pouvant être exécutées par cycle.
  - Elle dépend du matériel du GPU, en particulier du nombre d'unités fonctionnelles (par exemple, les cœurs CUDA) disponibles pour traiter les instructions arithmétiques en parallèle.
  - Pour Fermi, le débit est de 32 opérations par cycle, ce qui signifie qu'il peut traiter 32 opérations arithmétiques en un seul cycle d'horloge.
  - Pour Kepler, le débit est nettement plus élevé, avec 192 opérations par cycle, grâce aux améliorations architecturales et à l'augmentation du nombre de cœurs CUDA.

Valeurs du tableau 3-3 (2)

- **Parallélisme (opérations)** :
  - Le parallélisme fait référence au nombre d'opérations qui doivent être en cours (traitées activement) pour maintenir la pleine utilisation des unités arithmétiques du SM.
  - Cela permet au GPU de ne pas se bloquer et de continuer à exécuter les opérations de manière efficace.

- **Formule de calcul du parallélisme :**

$$\text{Parallelism} = 20 \times 32 = 640 \text{ operations}$$

Cela signifie que pour une pleine utilisation, le GPU Fermi a besoin de 640 opérations en cours.

- **Pour Kepler :**

$$\text{Parallelism} = 20 \times 192 = 3,840 \text{ operations}$$

Cela signifie que Kepler nécessite un degré de parallélisme beaucoup plus élevé - 3 840 opérations en cours - pour utiliser pleinement son plus grand nombre d'unités arithmétiques.

Valeurs du tableau 3-3 (3)

Résumé :

- La latence d'instruction (20 cycles) est la même pour Fermi et Kepler, car elle représente le temps nécessaire à l'exécution d'une opération arithmétique.

- Le débit diffère en fonction du nombre d'unités arithmétiques sur chaque architecture.

- Le parallélisme est calculé en multipliant la latence des instructions par le débit, ce qui donne le nombre total d'opérations qui doivent être exécutées simultanément pour que le SM soit entièrement occupé.

Ces calculs garantissent que le GPU est pleinement utilisé, ce qui minimise les temps d'inactivité et maximise les performances.

Valeurs du tableau 3-4 (1)

- Temps de latence des instructions (cycles) :
  - La latence des instructions représente le nombre de cycles d'horloge nécessaires pour effectuer une opération de mémoire (par exemple, charger ou stocker des données).
  - Dans ce cas, la latence de la mémoire pour Fermi et Kepler est de 800 cycles.
  - Il s'agit du temps nécessaire pour que les données soient extraites ou écrites dans la mémoire globale du GPU.

- Largeur de bande (GB/sec) :
  - La bande passante est la vitesse maximale à laquelle les données peuvent être transférées entre la mémoire du GPU et les unités de traitement.
  - Elle est généralement exprimée en gigaoctets par seconde (Go/sec).
  - Pour Fermi, la bande passante de la mémoire est de 144 GB/sec, et pour Kepler, elle est de 250 GB/sec.
  - Une bande passante plus large permet de transférer plus de données par seconde, ce qui améliore les performances pour les tâches nécessitant beaucoup de mémoire.

Valeurs du tableau 3-4 (1)

- **Largeur de bande (B/cycle)** :
  - La largeur de bande en octets par cycle (B/cycle) est une mesure plus précise qui permet de comprendre la quantité de données pouvant être transférées au cours de chaque cycle d'horloge.
  - Elle est calculée en convertissant la bande passante de gigaoctets par seconde (Go/sec) en octets par cycle d'horloge en utilisant la fréquence d'horloge de la mémoire du GPU.

Formule pour convertir la largeur de bande en B/cycle :

$$\text{Bandwidth (B/cycle)} = \frac{\text{Bandwidth (GB/sec)}}{\text{Memory Clock Frequency (GHz)}}$$

- Pour **Fermi** :

$$\frac{144\,\text{GB/sec}}{1.566\,\text{GHz}} \approx 92\,\text{B/cycle}$$

Cela signifie que Kepler peut transférer **96 octets par cycle d'horloge**.

Valeurs du tableau 3-4 (2)

- **Parallélisme (KB) :**
  - Le parallélisme fait référence au nombre d'opérations de mémoire (en Ko) nécessaires pour que le système de mémoire du GPU soit pleinement utilisé et que la latence de la mémoire soit masquée.
  - Pour calculer le parallélisme nécessaire, nous multiplions la latence de l'instruction mémoire (800 cycles) par le nombre d'octets pouvant être transférés par cycle (B/cycle) :

Formule de calcul du parallélisme :

$$\text{Parallelism (KB)} = \frac{\text{Instruction Latency (cycles)} \times \text{Bandwidth (B/cycle)}}{1024}$$

- Pour **Fermi**

$$\text{Parallelism} = \frac{800 \times 92}{1024} \approx 74\,\text{KB}$$

Cela signifie que Fermi a besoin de **74 Ko** d'opérations de mémoire en cours pour masquer totalement la latence de la mémoire.

Valeurs du tableau 3-4 (3)

- Pour **Kepler** :

$$\text{Parallelism} = \frac{800 \times 96}{1024} \approx 77\,\text{KB}$$

Kepler nécessite **77 Ko d'**opérations de mémoire en cours pour atteindre une pleine utilisation de la mémoire.

Résumé :

- La latence des instructions pour les opérations de mémoire est la même pour Fermi et Kepler (800 cycles).
- Fermi dispose d'une bande passante mémoire plus faible (144 Go/s), transférant 92 octets par cycle, ce qui nécessite 74 Ko d'opérations mémoire parallèles pour masquer la latence.
- Kepler dispose d'une bande passante mémoire plus élevée (250 Go/s), transférant 96 octets par cycle, et nécessite 77 Ko d'opérations mémoire parallèles pour masquer la latence.

Ce tableau indique la quantité de trafic de mémoire parallèle nécessaire pour utiliser pleinement le sous-système de mémoire du GPU et éviter les blocages dus à la latence de la mémoire.

Architectures modernes de GPU

- Les architectures modernes de GPU ont évolué au-delà de Fermi et Kepler.

- Ces deux architectures ont été développées par NVIDIA à des moments différents.

- Fermi (GF100) a été introduit en 2010, suivi par Kepler (GK110) en 2012, qui a représenté une amélioration significative de Fermi en termes d'efficacité énergétique et de performance.

- Cependant, les GPU modernes d'aujourd'hui utilisent des architectures beaucoup plus récentes telles que Pascal, Volta, Turing, Ampere et Ada Lovelace (pour les cartes graphiques de la série RTX 4000, par exemple).

- Ces architectures offrent des performances accrues, notamment en termes de capacité de calcul, de gestion de la consommation d'énergie et de prise en charge des dernières technologies telles que le ray tracing en temps réel.

Microarchitecture Pascal

Introduction :

- Lancé en 2016, Pascal (P100) a été le premier grand GPU pour datacenter de NVIDIA, conçu spécifiquement pour le deep learning et le calcul haute performance.

- Amélioration majeure : Prise en charge des calculs de demi-précision (FP16), essentiels pour l'IA et les simulations scientifiques.

- Largement utilisé dans des appareils tels que le Jetson TX2 et dans plusieurs applications de haute performance.

Microarchitecture Pascal

Points forts de l'architecture :

1. Multiprocesseur de flux (SM) :
   - Chaque SM est composé de deux blocs de traitement, formant un multiprocesseur complet.
   - Le GPU P100 contient 56 SM, chacun avec des cœurs CUDA et des unités fonctionnelles spécialisées.

2. Composants clés :
   - Blocs verts : Représentent les cœurs CUDA pour les calculs parallèles à usage général.
   - Blocs jaunes : Dédiés aux calculs en virgule flottante à double précision (importants dans les charges de travail HPC).
   - SFUs : Unités de fonctions spéciales pour les fonctions complexes (par exemple, sinus, log, exponentielle) pour les tâches d'intelligence artificielle.
   - Unités LD/ST : Effectuer des opérations de mémoire (chargement/enregistrement).

3. Caractéristiques supplémentaires :
   - Mémoire de texture : Prend en charge les tâches de rendu 3D avancées.
   - Fichier de registre : Fournit de la mémoire pour les threads qui est partagée à travers le système.

Pascal GP100 SM Unit

Microarchitecture Pascal

Applications :

- Idéal pour les charges de travail dans le domaine de l'apprentissage profond et du calcul à haute performance (HPC).

- Elle est encore présente dans des systèmes comme le P100 et le Jetson TX2, mais elle est peu à peu remplacée par des architectures plus récentes comme Turing et Ampere.

Microarchitecture Pascal

Conclusion :

- Pascal représente une étape clé dans l'évolution des GPU, permettant l'apprentissage en profondeur et l'amélioration de la précision de la virgule flottante.

- Bien que le P100 arrive à la fin de son cycle de vie, il reste une architecture importante pour les systèmes existants.

Microarchitecture Volta

- **Introduction** : Volta est sorti fin 2017 et s'est largement répandu en 2018, succédant à l'architecture Pascal.
- **Améliorations clés** :
  - Volta a présenté les **Tensor Cores**, des unités spécialisées dans les tâches d'apprentissage profond, permettant des opérations **Multiply-Accumulate (MAC)** pour la multiplication matricielle, largement utilisée dans l'IA et l'apprentissage profond.
  - **Opérations MAC** : Effectuer l'opération D = A x B + C pour des matrices 4 x 4, avec des entrées en **FP16** et des sorties en **FP32**, en optimisant la précision.
- **Instruction Fused Multiply-Add (FMA)** :
  - L'**instruction FMA** de Volta combine la multiplication et l'addition en un seul cycle d'horloge.
  - Cela améliore les opérations **en précision mixte**, rendant Volta jusqu'à **9 fois plus rapide** dans certaines charges de travail d'IA par rapport à Pascal.
- **Cas d'utilisation** :
  - Largement utilisé pour l'**apprentissage profond**, la formation de réseaux neuronaux et les **couches convolutives** dans les applications modernes d'IA.
  - Les **cœurs tensoriels** de Volta effectuent à la fois la multiplication et l'addition au cours du même cycle d'horloge, ce qui accroît l'efficacité des charges de travail informatiques.

Microarchitecture Turing

- La microarchitecture Turing, présentée fin 2018, est conçue pour les applications de jeu et d'IA.
- Il s'appuie sur Volta et introduit la prise en charge des types de données INT8 et INT4, ce qui permet d'effectuer des calculs plus rapides qu'avec le FP16.
- Les Turing Tensor Cores offrent des améliorations significatives en termes de performances, ce qui rend ces GPU idéaux pour les tâches d'apprentissage en profondeur.
- L'inférence en INT8 sur les GPU Turing est environ deux fois plus rapide qu'en FP16 et dix fois plus rapide qu'en FP32.
- Les GPU de la série RTX 20 sont des exemples de l'utilisation de l'architecture Turing, offrant des accélérations significatives dans la recherche sur l'IA et les jeux.

Microarchitecture Ampère

- **Ampere** a été introduit en mai 2020, succédant à Volta.
  - Le **GPU** phare **A100** est un puissant GPU de centre de données compatible avec CUDA, conçu pour les tâches d'apprentissage profond, avec **108 SM** (multiprocesseurs de streaming).
- **Principales innovations** :
  - **Troisième génération de cœurs tenseurs** : Ces Tensor Cores prennent en charge le **TF32** (Tensor Float 32) et offrent un **débit 20 fois supérieur** à celui de Volta, en particulier pour les tâches d'apprentissage en profondeur.
  - **Sparsité structurée (SS)** : Une nouvelle méthode pour réduire la complexité de calcul dans les réseaux neuronaux en élaguant les poids redondants, conduisant à une **accélération de 2 x** pour les multiplications de matrices.
  - Ampere permet une formation sans mélange de précision, ce qui constitue un avantage significatif par rapport aux architectures précédentes.
- **Applications** :
  - Avec **Ampere**, les praticiens de l'apprentissage profond peuvent utiliser **TF32** et l'espacement structuré pour une formation et une inférence efficaces. Ces améliorations réduisent la demande en matériel et augmentent l'efficacité de calcul, en particulier pour les grands modèles de réseaux neuronaux.

Microarchitecture Hopper (2022)

Principales caractéristiques de l'architecture de la trémie :

1. **Prise en charge du format de données FP8** :
    - Hopper introduit la prise en charge de deux variantes d'opérations en virgule flottante 8 bits (FP8) avec différentes configurations d'exposant et de mantisse.
    - Ils sont utiles pour l'entraînement de grands modèles linguistiques tels que GPT-3 ou PaLM, car le FP8 réduit les frais généraux de conversion, améliorant ainsi l'efficacité sans perte de performance.
2. **Cœurs tensoriels de quatrième génération** : Les cœurs tensoriels de Hopper doublent les performances de ceux d'Ampere, ce qui lui permet d'effectuer plus d'opérations par cycle d'horloge. Avec la prise en charge du FP8, le GPU H100 de Hopper atteint un débit matriciel six fois supérieur à celui du GPU A100 d'Ampere.
3. **Accélérateurs de mémoire tensorielle (TMA)** : Ces accélérateurs aident à gérer les goulets d'étranglement de la mémoire pour les modèles comportant des milliards de paramètres, en permettant des transferts de données plus rapides entre la mémoire globale et la mémoire partagée sans attendre la fin des calculs.
4. **Instructions DPX pour la programmation dynamique** : Ces instructions sont particulièrement utiles pour les tâches lourdes en termes de calcul, telles que la génomique et la robotique, car elles accélèrent le calcul des chemins optimaux jusqu'à 7 fois par rapport à Ampere.
5. **Clusters de blocs de threads (TBC)** : Les TBC permettent un contrôle plus fin des ressources SM en autorisant les blocs d'un même cluster à partager la mémoire, ce qui améliore l'efficacité de la communication entre les SM.
6. **Moteur de transformateur (TE)** : Cette fonctionnalité est conçue pour optimiser les couches d'attention dans les modèles tels que les Transformers en utilisant des calculs de précision mixte (FP16/FP8). Le TE analyse intelligemment les activations et adapte la conversion de format pour éviter toute perte de précision, ce qui accélère considérablement l'inférence de l'apprentissage profond.

Mise en évidence d'un parallélisme suffisant

- Concept clé :
  - Le GPU peut passer d'une warp à l'autre avec une **surcharge minimale** (1 à 2 cycles), grâce à sa capacité à conserver l'état requis sur la puce.

- L'utilité du parallélisme :
  - Si le nombre de **threads actifs** est suffisant, le GPU peut rester occupé à chaque étape du pipeline, à chaque cycle.
  - La latence d'une warp peut être masquée par l'**exécution d'autres** warps.
  - Le GPU est ainsi pleinement utilisé, ce qui réduit les temps d'inactivité.

Exposer le parallélisme suffisant (2)

- **Formule pour le parallélisme requis** :
  - Pour calculer le **parallélisme requis**, il faut multiplier le nombre de cœurs par multiprocesseur de flux (SM) par la **latence d'instruction** d'une opération arithmétique unique.
  - **Exemple** (pour l'architecture Fermi) :
    - Fermi dispose de **32 voies de pipeline de virgule flottante** par SM.
    - La **latence** d'une instruction arithmétique est de **20 cycles**.
    - Nombre de threads requis = **32 cœurs × 20 cycles = 640 threads** par SM.

Cela garantit qu'il y a suffisamment de parallélisme pour que l'appareil soit occupé, bien que ce nombre soit considéré comme une **limite inférieure**.

Conclusion :

- **L'exposition d'un parallélisme suffisant** garantit que le GPU peut exécuter efficacement les warps sans attendre d'autres opérations, ce qui est essentiel pour maximiser les performances.

Occupation(1)

- **Concept clé** :
  - L'**occupation** est le rapport entre le nombre de **warps actives** et le **nombre maximal de warps** qu'un multiprocesseur de streaming (SM) peut prendre en charge.
  - Il est essentiel de maintenir les cœurs CUDA occupés et de masquer les temps de latence.

- **Détermination des déformations maximales sur SM** :
  - **Fonction CUDA** :
    - Utilisez 'cudaGetDeviceProperties' pour obtenir le nombre maximum de warps par SM.
    - La variable "maxThreadsPerMultiProcessor" indique le nombre maximum de threads par SM.

  - **Calcul de la warp** :
    - Warps maximum = 'maxThreadsPerMultiProcessor / 32' (puisqu'un warp est composé de 32 threads).
    - Exemple : Pour un GPU **Tesla M2070**, avec **1536 threads par SM**, le nombre maximum de warps = **48 warps** par SM.

Occupation (2)

- **Calculateur d'occupation CUDA :**
  - CUDA Toolkit fournit une feuille de calcul **de l'**occupation pour aider à maximiser l'occupation en sélectionnant les **dimensions** optimales **de la grille** et du **bloc**.
  - Les informations requises sont les suivantes :
    - **1. threads par bloc**
    - **2. Registres par thread**
    - **3. Mémoire partagée par bloc**

- **Lignes directrices pour la taille des grilles et des blocs** :
  - **Conservez les blocs de thread comme multiples de la taille de la warp** (32).
  - **Évitez les petits blocs** ; commencez avec au moins **128 ou 256 threads** par bloc.
  - **Équilibrer la taille des blocs** en fonction des besoins en ressources du noyau.
  - **Maximiser les warps actives par SM** pour un parallélisme optimal.

Occupation (3)

- **Note finale** :
  - Bien que l'occupation soit essentielle, **une occupation complète ne garantit pas toujours de meilleures performances.**
  - L'optimisation basée uniquement sur l'occupation peut négliger d'autres facteurs de performance tels que la bande passante de la mémoire et la latence.

Synchronisation (1)

- **Définition** :
  - **Synchronisation à barrière** : Une primitive qui garantit que les threads dans la programmation parallèle atteignent un certain point avant de continuer.
  - Dans **CUDA**, la synchronisation s'effectue à deux niveaux :
    - **Au niveau du système** : Attendez que les tâches de l'hôte et du périphérique soient terminées.
    - **Au niveau du bloc** : Attendre que tous les threads d'un bloc atteignent le même point d'exécution.

- **Synchronisation au niveau du système** :
  - Fonction : 'cudaDeviceSynchronize()'
    - Bloque l'application hôte jusqu'à ce que toutes les opérations CUDA (par exemple, les noyaux, les copies de mémoire) soient terminées.
    - Utile pour rattraper les erreurs des opérations asynchrones précédentes.

Synchronisation (2)

- **Synchronisation au niveau des blocs** :
  - - Fonction : '__syncthreads()'
    - Force tous les threads d'un bloc de threads à attendre jusqu'à ce qu'ils atteignent tous la barrière.
    - S'assure que tous les accès à la mémoire globale et partagée sont terminés avant de continuer.
    - **Limitation** : Peut avoir un impact négatif sur les performances en provoquant l'inactivité de certaines warps.

- **Risques liés à la mémoire partagée** :
  - - Les threads d'un même bloc **partagent** la **mémoire** et les **registres**.
  - - Nécessité d'éviter les **conditions de course** :
    - Exemple : Risque de lecture après écriture - lorsqu'un thread lit un emplacement de mémoire après qu'un autre thread y a écrit sans synchronisation appropriée.
    - Les risques peuvent également inclure des scénarios d'écriture après lecture et d'écriture après écriture.

Synchronisation (3)

- **Synchronisation entre les blocs** :
  - **Pas de synchronisation** entre les threads dans différents blocs.
  - La seule façon de synchroniser les blocs est d'utiliser un **point de synchronisation global** :
    - Termine le noyau actuel et en lance un nouveau pour poursuivre l'exécution.

- **Importance de l'exécution indépendante des blocs** :
  - Le fait d'empêcher la synchronisation entre les blocs permet aux GPU d'exécuter les blocs dans n'importe quel ordre.
  - Cela garantit l'évolutivité des programmes CUDA sur des GPU massivement parallèles.

Évolutivité (1)

- **Définition** :
  - **Évolutivité** : La capacité d'une application parallèle à utiliser des ressources matérielles supplémentaires pour améliorer efficacement les performances.
  - Par exemple, une application CUDA est **évolutive** si son exécution sur deux multiprocesseurs en continu divise par deux le temps d'exécution par rapport à l'exécution sur un seul multiprocesseur.

- **Caractéristiques** :
  - **Code parallèle** :
    - Possibilité de faire évoluer le système en utilisant davantage de cœurs de calcul.
    - **Utilisation efficace des ressources** : Un programme évolutif maximise l'utilisation de toutes les ressources informatiques disponibles.
  - **Code de série** :
    - **Non extensible** : L'ajout de cœurs n'a aucun impact sur les performances d'une application séquentielle à un seul thread.

Évolutivité (2)

- **Évolutivité transparente** :
  - **Définition** : Capacité d'exécuter le même code d'application sur un nombre variable de cœurs de calcul sans modification.
  - Avantages :
    - **Des cas d'utilisation plus larges** : Extension des possibilités d'utilisation des applications sur différents matériels.
    - **Réduit les efforts des développeurs** : Les développeurs n'ont pas besoin de réécrire le code pour différentes configurations matérielles.

- **Importance de l'évolutivité** :
  - **Charges de travail plus importantes** : Les systèmes évolutifs peuvent gérer des charges de travail plus importantes en ajoutant simplement des cœurs de matériel supplémentaires.
  - **Risque d'inefficacité** : Un système inefficace et non évolutif peut rapidement atteindre ses limites de performance.

Évolutivité (3)

- **Exécution du noyau CUDA** :
  - **Blocs de threads** :
    - Réparti entre plusieurs SM.
    - **Exécution indépendante** : Peut être exécutée en parallèle ou en série, ce qui permet de l'**étendre à n'importe quel nombre de cœurs de calcul**.

- **Exemple d'évolutivité de CUDA (figure 3-18)** :
  - **GPU avec 2 SM** : Exécute deux blocs en même temps.
  - **GPU avec 4 SM** : Exécute quatre blocs à la fois.
  - Sans modifier le code, les applications peuvent s'adapter automatiquement à différentes configurations de GPU, le temps d'exécution dépendant des ressources disponibles.

- Cette évolutivité permet aux programmes CUDA de s'adapter efficacement à des configurations matérielles plus vastes et plus complexes.

Évolutivité (4)

Exemple d'évolutivité de CUDA

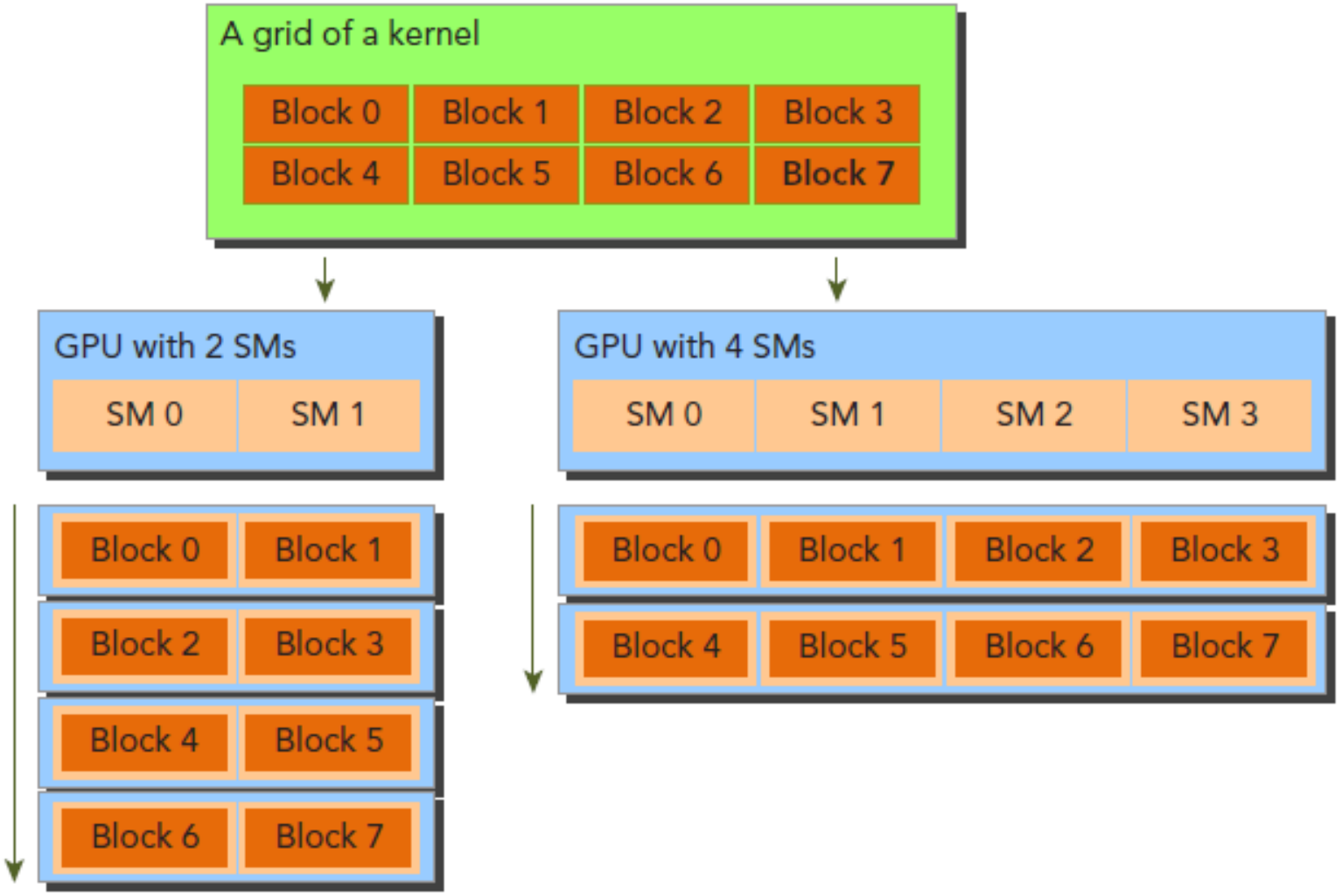


FIGURE 3-18

Vérification des warps actives avec 'nvprof'

Vue d'ensemble

- **Objectif** :
  - Mesurer et comparer les performances du noyau en testant différentes configurations de blocs de threads en utilisant 'sumMatrix' sur un GPU Tesla M2070.
- **Approche** :
  - Évaluer les tailles de blocs de threads de base : (32,32), (32,16), (16,32), (16,16).
  - Comparer le temps écoulé pour différentes configurations.
  - Utilisez 'nvprof' pour analyser le taux d'occupation atteint, qui est le rapport entre le nombre de warps actifs par cycle et le maximum possible.

Vérifier les Warps actifs avec 'nvprof'

- **Résultats du test (temps écoulé)**

```
> sumMatrix.exe 32 32
sumMatrixOnGPU2D << (512,512), (32,32) >>> durée 60 ms

> sumMatrix.exe 32 16
sumMatrixOnGPU2D << (512,1024), (32,16) >>> durée 38 ms

> sumMatrix.exe 16 32
sumMatrixOnGPU2D << (1024,512), (16,32) >>> durée 51 ms

> sumMatrix.exe 16 16
sumMatrixOnGPU2D << (1024,1024),(16,16) >>> durée 46 ms
```

- **Principales observations** :
  - **Le plus lent** : (32, 32) -> 60 ms
  - **Le plus rapide** : (32,16) -> 38 ms

Vérifier les Warps actifs avec 'nvprof'

- **Occupation réalisée à l'aide de 'nvprof'**

```
> nvprof --metrics achieved_occupancy sumMatrix.exe 32 32
sumMatrixOnGPU2D <<(512,512), (32,32)>>> Occupation atteinte 0.501071

> nvprof --metrics achieved_occupancy sumMatrix.exe 32 16
sumMatrixOnGPU2D <<(512,1024), (32,16)>>> Occupation atteinte 0.736900

> nvprof --metrics achieved_occupancy sumMatrix.exe 16 32
sumMatrixOnGPU2D <<(1024,512), (16,32)>>> Occupation atteinte 0.766037

> nvprof --metrics achieved_occupancy sumMatrix.exe 16 16
sumMatrixOnGPU2D <<(1024,1024),(16,16)>>> Occupation atteinte 0.810691
```

- **Principales observations** :
  - **Taux d'occupation le plus élevé :** (16,16) -> 0.810691
  - **Occupation la plus faible :** (32,32) -> 0.501071

Vérifier les Warps actifs avec 'nvprof'

Conclusions

1. **Plus de blocs de threads = taux d'occupation plus élevé et meilleures performances** (comme le montre la deuxième configuration).

2. **Le taux d'occupation n'est pas le seul déterminant de la performance** (comme le montre la quatrième configuration, qui a le taux d'occupation le plus élevé mais n'est pas la plus rapide).

Vérifier les opérations de mémoire avec 'nvprof'

Vue d'ensemble

- **Objectif** :
  - Évaluer l'efficacité des opérations de mémoire dans le noyau `'sumMatrix'` (C[idx] = A[idx] + B[idx]).

- **Opérations de mémoire :**
  - Deux charges de mémoire
  - Une seule mémoire

- **Outils** :
  - Utilisez `'nvprof'` pour examiner les performances de la mémoire via des métriques telles que `'gld_throughput'` et `'gld_efficiency'`.

Vérifier les opérations de mémoire avec 'nvprof'

Efficacité de la lecture de la mémoire avec 'gld_throughput' (en anglais)

- **Sortie de la ligne de commande** :

```
> nvprof --metrics gld_throughput sumMatrix.exe 32 32
sumMatrixOnGPU2D << (512,512), (32,32)>>> Débit de charge global 35.908GB/s

> nvprof --metrics gld_throughput sumMatrix.exe 32 16
sumMatrixOnGPU2D << (512,1024), (32,16)>>> Débit de charge global 56,478GB/s

> nvprof --metrics gld_throughput sumMatrix.exe 16 32
sumMatrixOnGPU2D << (1024,512), (16,32)>>> Débit de charge global 85,195GB/s

> nvprof --metrics gld_throughput sumMatrix.exe 16 16
sumMatrixOnGPU2D << (1024,1024),(16,16)>>> Débit de charge global 94.708GB/s
```

- **Principales observations** :
  - Le **quatrième cas** a le débit le plus élevé mais est **plus lent que le deuxième**.
  - **Un débit de charge plus élevé ≠ de meilleures performances** grâce au fonctionnement des transactions en mémoire.

Vérifier les opérations de mémoire avec 'nvprof'

Efficacité globale de la charge avec 'gld_efficiency'

- **Sortie de la ligne de commande :**

```
> nvprof --metrics gld_efficiency sumMatrix.exe 32 32
sumMatrixOnGPU2D <<< (512,512), (32,32)>>> Efficacité de la charge de mémoire globale 100,00%

> nvprof --metrics gld_efficiency sumMatrix.exe 32 16
sumMatrixOnGPU2D << (512,1024), (32,16)>>> Efficacité de la charge de mémoire globale 100,00%

> nvprof --metrics gld_efficiency sumMatrix.exe 16 32
sumMatrixOnGPU2D << (1024,512), (16,32)>>> Efficacité de la charge de mémoire globale 49,96%.

> nvprof --metrics gld_efficiency sumMatrix.exe 16 16
sumMatrixOnGPU2D << (1024,1024),(16,16)>>> Efficacité de la charge de la mémoire globale 49,80%
```

- **Principales observations** :
  - Les deux dernières configurations (**16, 32** et **16, 16**) sont **deux fois moins efficaces** que les deux premières**.**
  - **L'efficacité de la charge est liée à la taille du bloc de thread** (les dimensions de la demi-lame affectent les performances)

Vérifier les opérations de mémoire avec 'nvprof'

Conclusions

- La métrique **gld_efficiency** révèle que l'efficacité des opérations de chargement de la mémoire est beaucoup plus faible pour les petites dimensions (16, 32 et 16, 16) que pour les grandes (32, 32 et 32,16), ce qui explique leurs performances plus lentes malgré un débit plus élevé.

Exposer le parallélisme (1)

- **Vue d'ensemble** :
  - **Objectif** : Comprendre l'exécution du warp en examinant un noyau de sommation de matrice 2D (sumMatrixOnGPU2D).
  - **Outils** : Utilisez `nvprof` pour établir le profil des mesures et connaître les dimensions optimales des grilles/blocs.
  - L'exercice permet d'acquérir de l'expérience en matière **d'heuristique de grille et de bloc**.

  - **Code du noyau** :

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY)
{
  unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x ;
}
```

  - Le noyau traite les éléments d'une matrice 2D en utilisant les dimensions du bloc et de la grille pour mapper les threads.

Exposer le parallélisme (2)

- **Taille de la matrice** :
  - Définir une grande matrice avec 16 384 éléments dans chaque dimension :

```
int nx = 1<<14 ;
int ny = 1<<14 ;
```

- **Configuration du bloc** :
  - Un extrait de code permet de définir les dimensions du bloc via la ligne de commande :

```
if (argc > 2) {
  dimx = atoi(argv[1]) ;
  dimy = atoi(argv[2]) ;
}
dim3 block(dimx, dimy) ;
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y) ;
```

Éviter la divergence des branches

- **Problème** :
  - **Divergence de branche** : Se produit lorsque le flux de contrôle dépend des indices de threads, ce qui entraîne une **exécution conditionnelle au sein d'une warp**.
  - Cela peut dégrader les performances du noyau, car les différents threads d'une même warp empruntent des chemins d'exécution différents.
- **Solution** :
  - **Réorganiser les schémas d'accès aux données** : L'optimisation de la disposition des données peut minimiser ou éviter complètement la divergence de la warp.
- **Exemple** :
  - Dans cette section, la **réduction parallèle** est utilisée comme exemple pour démontrer les techniques permettant d'éviter la divergence des branches.

En gérant efficacement le flux de contrôle, vous pouvez améliorer de manière significative les performances du warp et garantir une exécution efficace entre les threads.

Le problème de la réduction parallèle (1)

- **Objectif** :
  - Calculer la somme d'un tableau d'entiers à N éléments de manière efficace en parallèle à l'aide de CUDA.
- **Approche séquentielle** :
  - Dans le cas d'une exécution séquentielle, la somme est calculée comme suit :

```
int sum = 0 ;
pour (int i = 0 ; i < N ; i++)
    sum += array[i] ;
```

- **Approche parallèle** :
  - Pour accélérer la sommation pour un grand nombre d'éléments :
    1. **Partitionner** le vecteur d'entrée en plus petits morceaux.
    2. Affecter un **thread** au calcul de la somme partielle pour chaque morceau.
    3. **Combinez** les sommes partielles pour obtenir le résultat final.

Le problème de la réduction parallèle (2)

- **Addition parallèle par paire** :
  - **Somme par paire** : À chaque itération, les éléments sont appariés et des sommes partielles sont calculées, ce qui permet de mettre à jour l'entrée sur place. Ce processus se poursuit jusqu'à ce qu'il ne reste plus qu'une somme finale.

- **Deux approches** :
  1. **Paire voisine** : Les éléments adjacents sont appariés (par exemple, le premier et le deuxième).
  2. **Paire entrelacée** : Les éléments sont appariés avec une foulée qui les sépare (par exemple, le 1er avec le 3e).

- **Code de réduction récursive (paire entrelacée)** :

```
int recursiveReduce(int *data, int const size) {
    if (size == 1) return data[0] ; // cas de base
    int const stride = taille / 2 ;
    for (int i = 0 ; i < stride ; i++) {
        data[i] += data[i + stride] ; // réduction sur place
    }
    return recursiveReduce(data, stride) ; // appel récursif
}
```

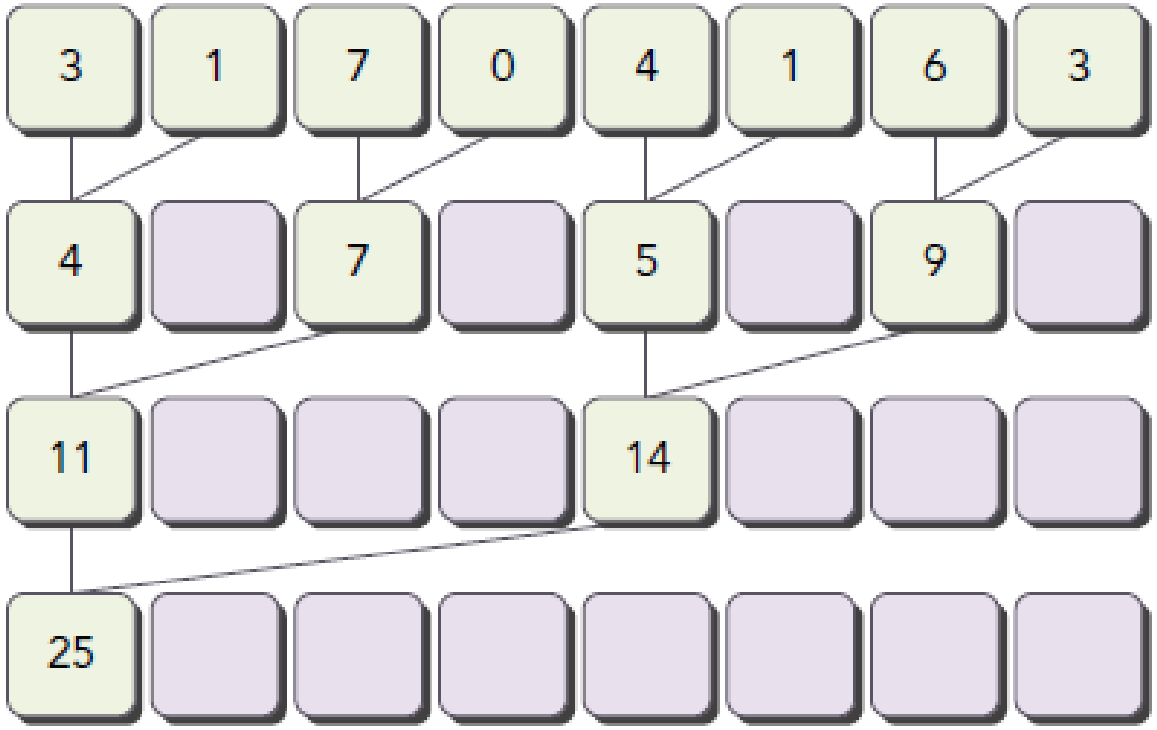
Le problème de la réduction parallèle (3)

- **Généralisation** :
  - La **réduction** peut s'appliquer à toute opération associative et commutative (par exemple, max, min, moyenne, produit), et pas seulement à l'addition.
  - La **réduction parallèle** est un modèle clé pour les algorithmes parallèles, essentiel pour effectuer efficacement des opérations sur de grands ensembles de données.

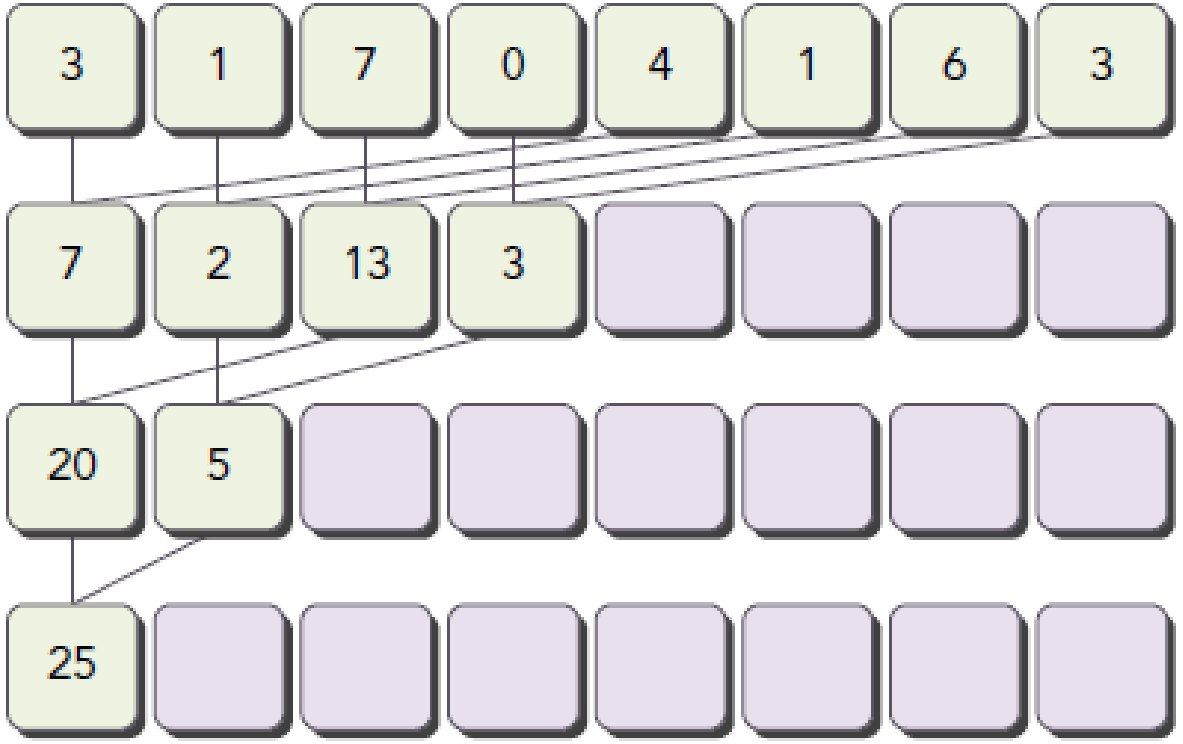
- **Prochaines étapes** :
  - Implémenter différents **noyaux de réduction parallèles** et évaluer leur impact sur les performances d'exécution à l'aide des outils CUDA.
  - Cela permettra de comprendre comment les différentes configurations affectent l'efficacité du traitement parallèle.

Ce problème illustre l'importance des **modèles parallèles** dans l'optimisation des calculs à grande échelle.

Le problème de la réduction parallèle (4)



La figure 3-19 illustre la mise en œuvre d'une paire voisine



La figure 3-20 illustre la mise en œuvre d'une paire entrelacée

Mise en œuvre récursive de l'approche par paires entrelacées

La fonction C suivante est une implémentation récursive de l'approche par paires entrelacées :

```
int recursiveReduce(int *data, int const size)
{

    // mettre fin au contrôle
    if (size == 1) return data[0] ;

    // renouveler la cadence
    int const stride = taille / 2 ;

    // réduction sur place
    for (int i = 0 ; i < stride ; i++) {
           data[i] += data[i + stride] ;
    }

    // appel récursif
    return recursiveReduce(data, stride) ;
}
```

Divergence dans la réduction parallèle (sommation parallèle)

- **Introduction à la réduction parallèle**
  - La réduction parallèle est un modèle fondamental utilisé en informatique parallèle pour calculer un résultat unique (par exemple, une somme, un maximum) à partir d'un grand ensemble de données en parallèle.
  - Il est généralement implémenté en CUDA pour obtenir des gains de performance sur les architectures GPU.
- **Approche : Réduction des paires de voisins**
  - **Concept** : Chaque thread est affecté à l'addition de deux éléments adjacents.
  - **Mise en œuvre** :
    - Le tableau est divisé en blocs, chacun étant géré par un bloc de threads.
    - Chaque thread effectue une étape de réduction à la fois, en additionnant deux éléments adjacents (par exemple, A[i] + A[i+1]).
    - Le résultat est enregistré dans la mémoire après chaque itération.
  - **Structure de la mémoire** : Il y a deux tableaux de mémoire :
    - Un grand tableau pour l'ensemble de l'entrée.
    - Un tableau plus petit qui contient les sommes partielles de chaque bloc.

Divergence dans la réduction parallèle (sommation parallèle)

- **Vue d'ensemble** :
  - La réduction parallèle est utilisée dans CUDA pour additionner efficacement de grands tableaux en exploitant des threads parallèles.
  - Chaque thread ajoute deux éléments adjacents, réduisant ainsi le tableau étape par étape jusqu'à ce qu'une somme finale soit atteinte.

- **Réduction des paires voisines (figure 3-21)** :
  - **Threads** : Chaque thread additionne deux éléments voisins de la mémoire globale.
  - **Processus** : Chaque étape divise par deux la taille du tableau en additionnant les paires adjacentes, réduisant ainsi le travail global pour l'itération suivante.
  - **Synchronisation** : "__syncthreads()" permet de s'assurer que tous les threads d'un bloc se terminent avant de continuer, évitant ainsi les conditions de course.

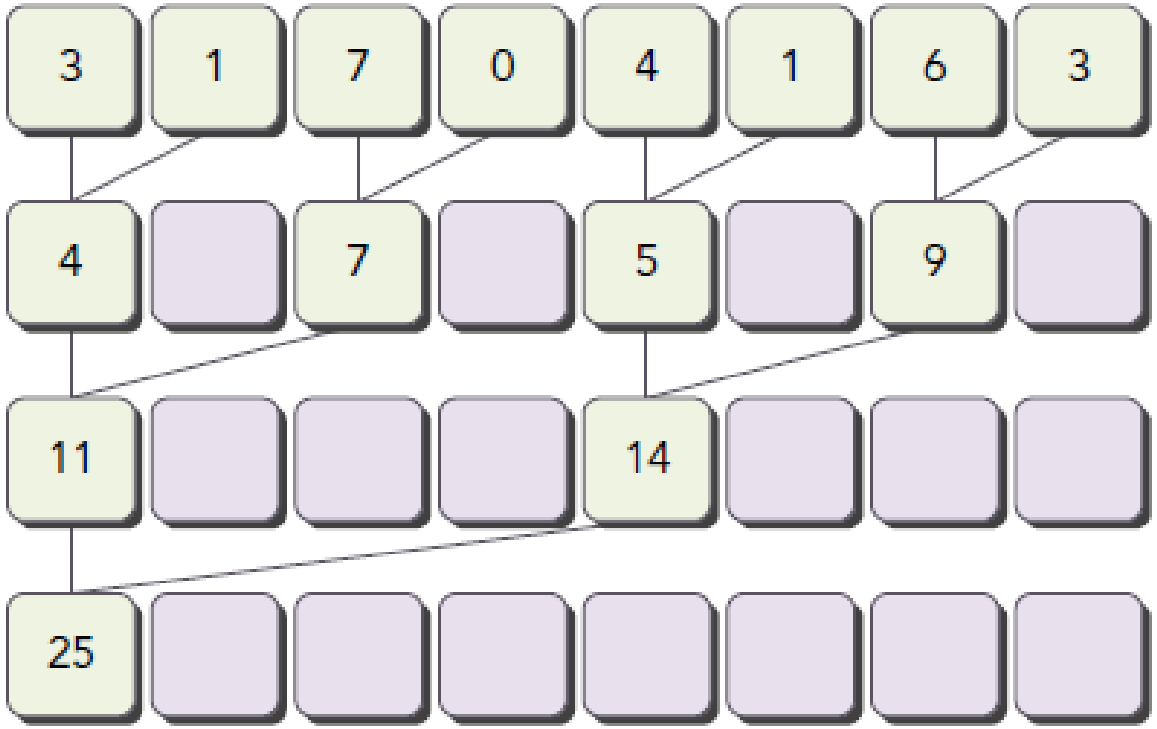
Divergence dans la réduction parallèle (sommation parallèle)

- **Code du noyau** :
  - Les threads additionnent les éléments adjacents avec des pas croissants, en écrivant le résultat du bloc final dans la mémoire globale.

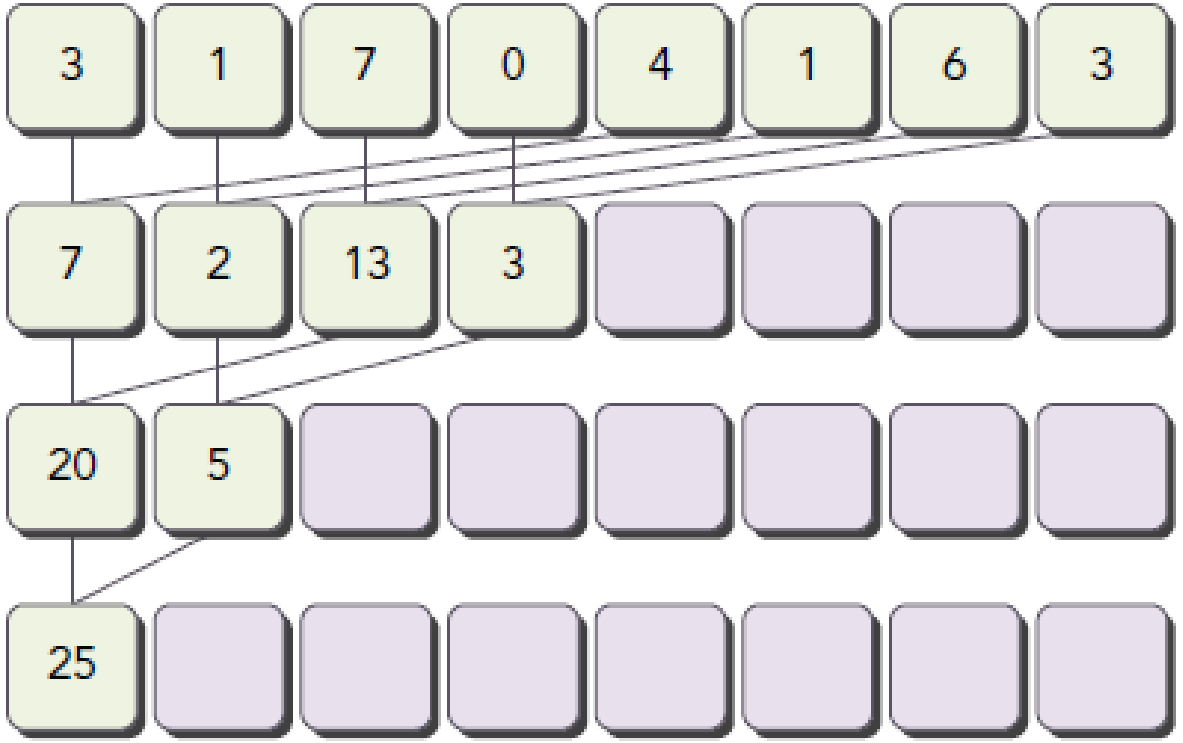
- **Performance** :
  - L'implémentation GPU est nettement plus performante que la réduction CPU (11 ms contre 29 ms pour 16 millions d'éléments).

Cette méthode est efficace pour les données volumineuses et permet des accélérations significatives en parallélisant le processus de sommation sur plusieurs blocs et threads sur le GPU.

Le problème de la réduction parallèle (4)



La figure 3-19 illustre la mise en œuvre d'une paire voisine



La figure 3-20 illustre la mise en œuvre d'une paire entrelacée

Mise en œuvre récursive de l'approche par paires entrelacées

La fonction C suivante est une implémentation récursive de l'approche par paires entrelacées :

```
int recursiveReduce(int *data, int const size)
{

    // mettre fin au contrôle
    if (size == 1) return data[0] ;

    // renouveler la foulée
    int const stride = taille / 2 ;

    // réduction sur place
    for (int i = 0 ; i < stride ; i++) {
           data[i] += data[i + stride] ;
    }

    // appel récursif
    return recursiveReduce(data, stride) ;
}
```

Divergence dans la réduction parallèle (sommation parallèle)

- **Introduction à la réduction parallèle**
  - La réduction parallèle est un modèle fondamental utilisé dans l'informatique parallèle pour calculer un résultat unique (par exemple, la somme, le maximum) à partir d'un grand ensemble de données en parallèle.
  - Il est généralement implémenté en CUDA pour obtenir des gains de performance sur les architectures GPU.
- **Approche : Réduction des paires de voisins**
  - **Concept** : Chaque fil est affecté à l'addition de deux éléments adjacents.
  - **Mise en œuvre** :
    - Le tableau est divisé en blocs, chacun étant géré par un bloc de threads.
    - Chaque thread effectue une étape de réduction à la fois, en additionnant deux éléments adjacents (par exemple, A[i] + A[i+1]).
    - Le résultat est enregistré dans la mémoire après chaque itération.
  - **Structure de la mémoire** : Il y a deux tableaux de mémoire :
    - Un grand tableau pour l'ensemble de l'entrée.
    - Un tableau plus petit qui contient les sommes partielles de chaque bloc.

Divergence dans la réduction parallèle (sommation parallèle)

- **Vue d'ensemble** :
  - La réduction parallèle est utilisée dans CUDA pour additionner efficacement de grands tableaux en exploitant des threads parallèles.
  - Chaque fil ajoute deux éléments adjacents, réduisant ainsi le tableau étape par étape jusqu'à ce qu'une somme finale soit atteinte.

- **Réduction des paires voisines (figure 3-21)** :
  - **Threads** : Chaque thread additionne deux éléments voisins de la mémoire globale.
  - **Processus** : Chaque étape divise par deux la taille du tableau en additionnant les paires adjacentes, ce qui réduit le travail global pour l'itération suivante.
  - **Synchronisation** : "__syncthreads()" permet de s'assurer que tous les threads d'un bloc se terminent avant de continuer, évitant ainsi les conditions de course.

Divergence dans la réduction parallèle (sommation parallèle)

La figure 3-21 illustre l'approche par paires de voisins

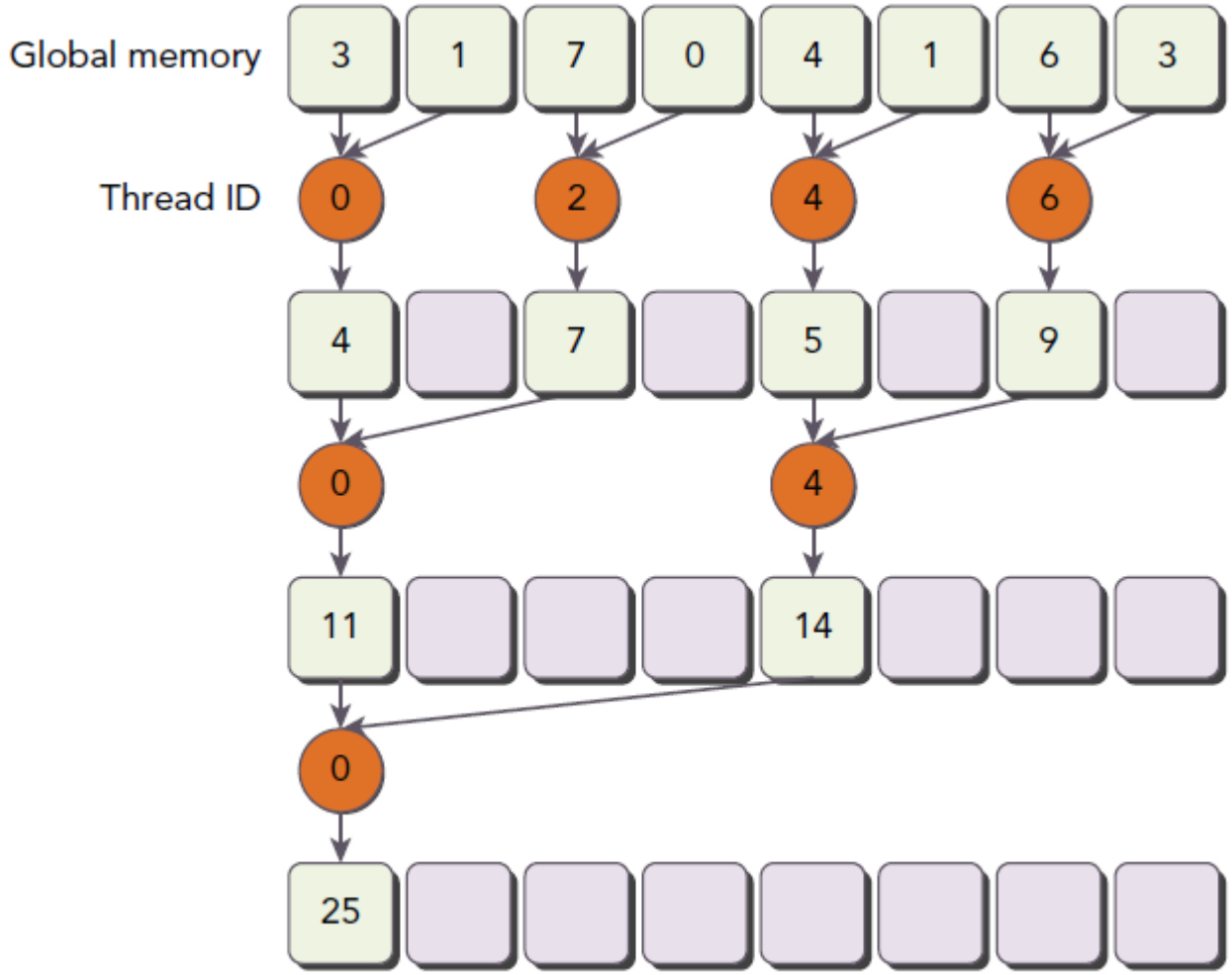


FIGURE 3-21

Divergence dans la réduction parallèle (sommation parallèle)

- **Code du noyau** :
  - Les threads additionnent les éléments adjacents avec des pas croissants, en écrivant le résultat du bloc final dans la mémoire globale.

- **Performance** :
  - L'implémentation GPU est nettement plus performante que la réduction CPU (11 ms contre 29 ms pour 16 millions d'éléments).

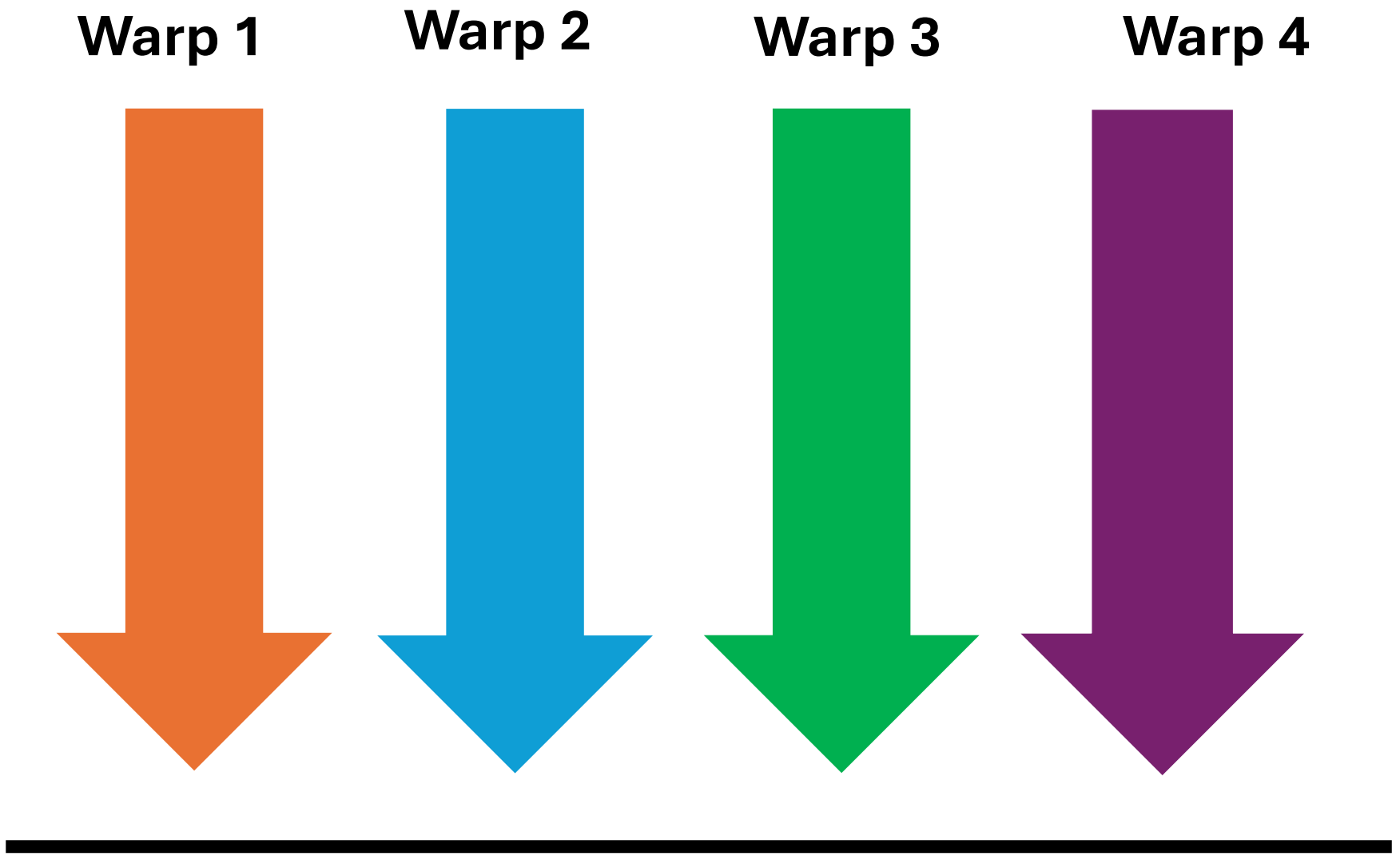
Cette méthode est efficace pour les données volumineuses et permet des accélérations significatives en parallélisant le processus de sommation sur plusieurs blocs et threads sur le GPU.

Synchronisation

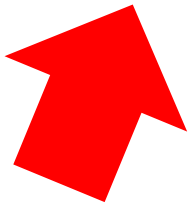
- cudaDeviceSynchronize
- __syncthreads

Introduire un point de synchronisation global dans le code de l'hôte

Synchronisation avec dans un bloc


Warp 1
Warp 2
Warp 3
Warp 4
__syncthreads

Réduction parallèle



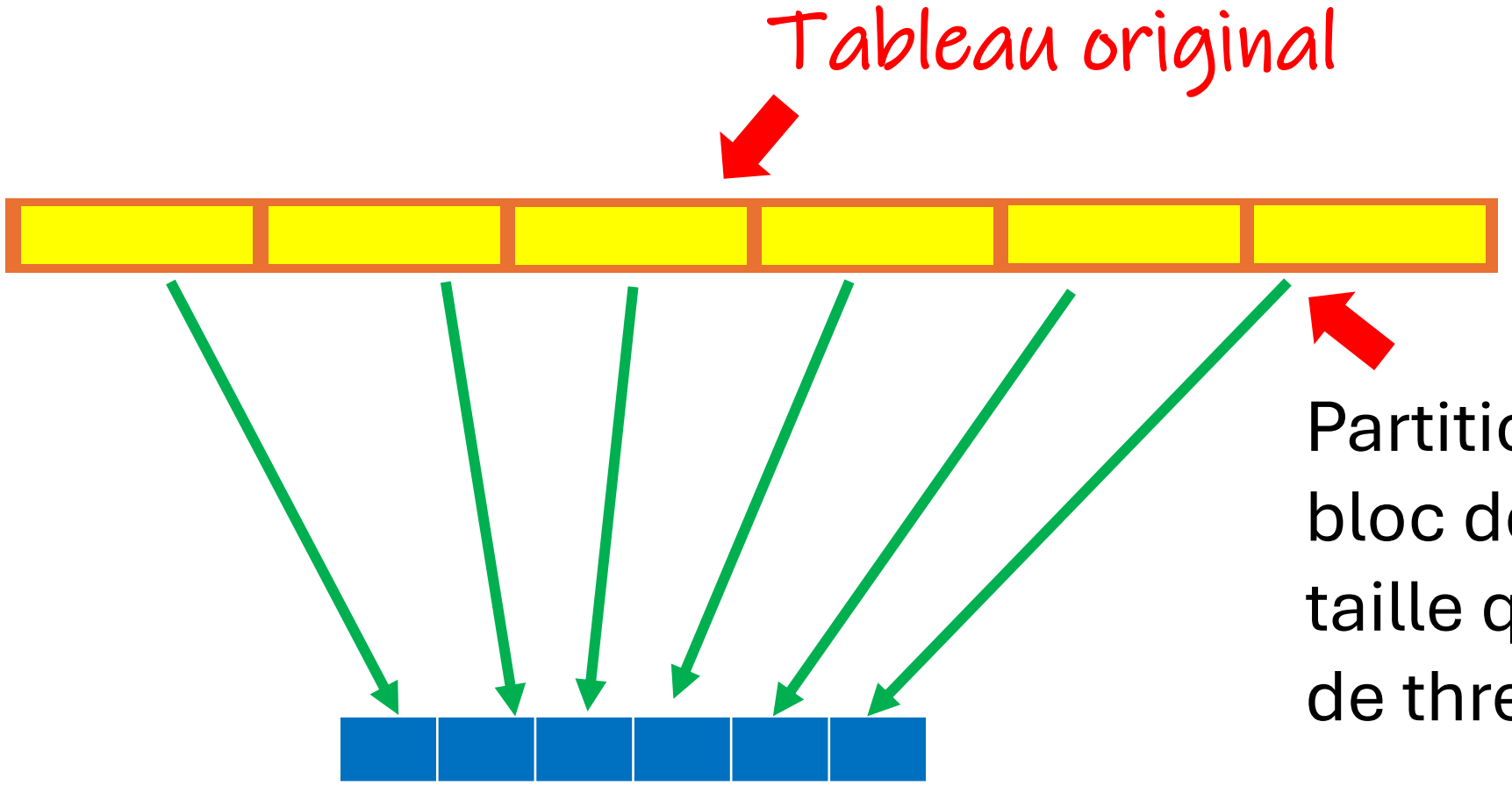
Le problème général de l'exécution d'opérations commutatives et associatives sur un vecteur est connu sous le nom de problème de réduction.

Réduction séquentielle

```
int sum =0 ;
Pour (int I =0 ; I < taille ; I ++)
{
        sum + = array[i] ;
}
```

Notre approche

- Partitionner le vecteur d'entrée en petits morceaux.
- Et chaque morceau sera additionné séparément.
- additionner les résultats partiels de chaque morceau pour obtenir une somme finale



Partitionné en un bloc de la même taille que le bloc de threads

La somme de chaque bloc sera stockée dans un tableau de somme partielle.

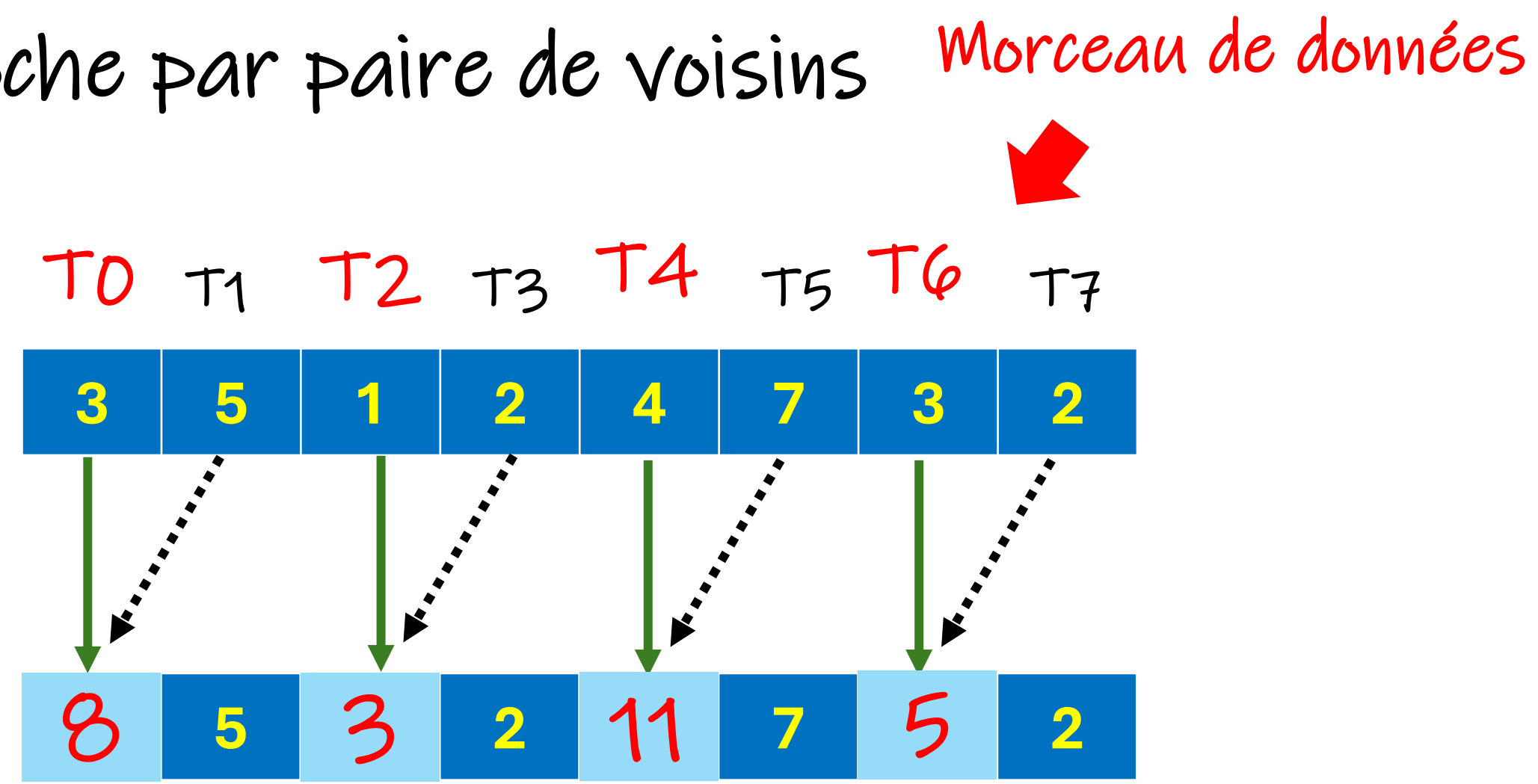
Approche par paire de voisins

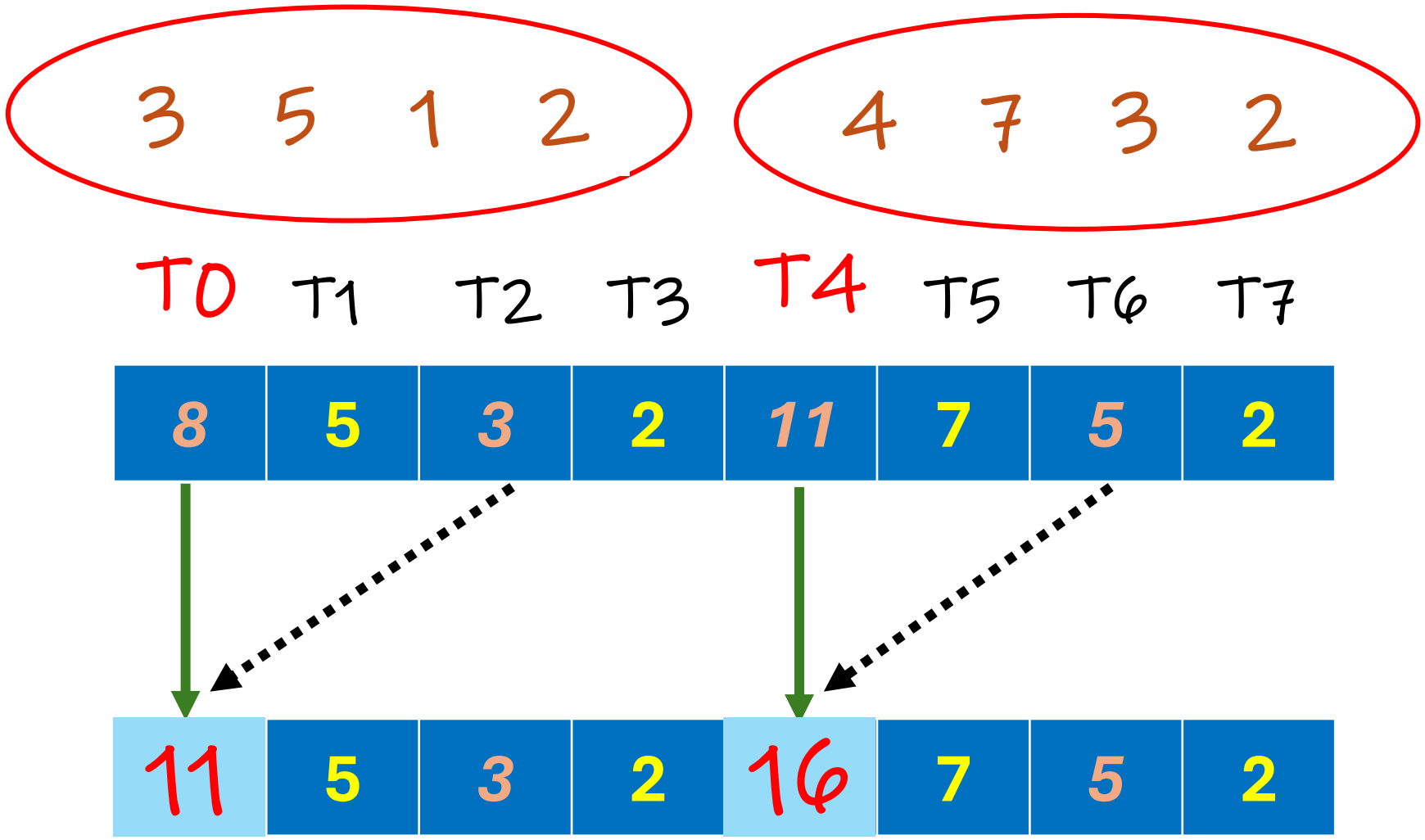
- Nous allons calculer la somme du bloc de manière itérative et, à chaque itération, les éléments sélectionnés sont associés à leurs voisins à partir d'un décalage donné.

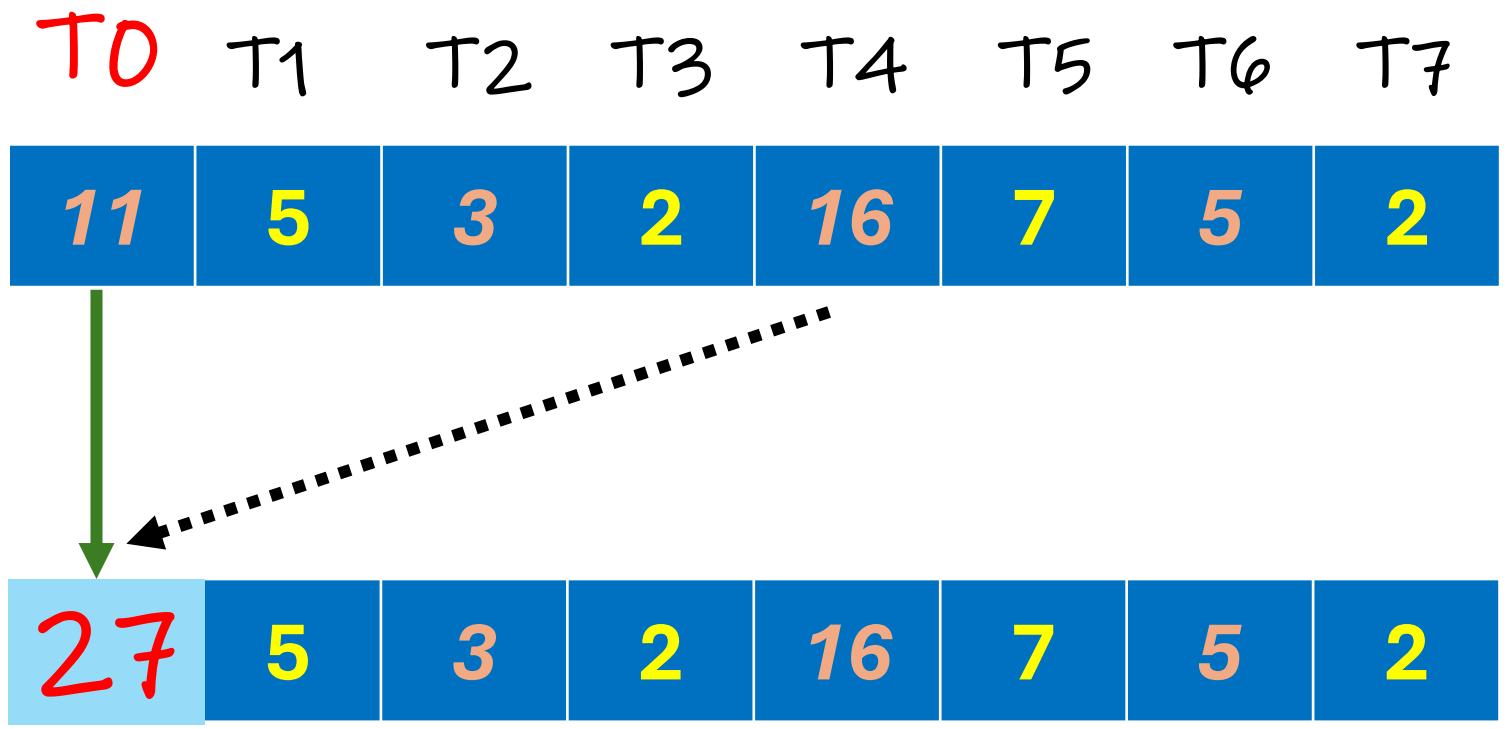
- Pour la première itération, nous allons définir 1 comme décalage et, à chaque itération, ce décalage sera multiplié par deux.

- Le nombre de threads qui effectueront un travail efficace sera divisé par cette valeur de décalage.

Approche par paire de voisins




3 5 1 2
4 7 3 2
T0 T1 T2 T3 T4 T5 T6 T7
8 5 3 2 11 7 5 2
11 5 3 2 16 7 5 2


T0
T1
T2
T3
T4
T5
T6
T7
11
5
3
2
16
7
5
2
27
5
3
2
16
7
5
2

Segment de code

```
For(int offset =1 ; offset < blockdim.x ; offset *=2)
{
        if( tid % (2* offset)==0 )
        {
                input[ tid ] += input[ tid + offset ] ;
        }
        __syncthreads()
}
```

Attention..... Attention.....

- Soyez attentif lorsque vous utilisez l'appel à la fonction __syncthreads() à l'intérieur de la vérification des conditions.

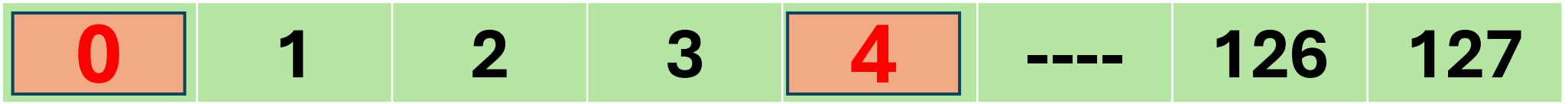
Paradoxe

Divergence dans l'algorithme de réduction

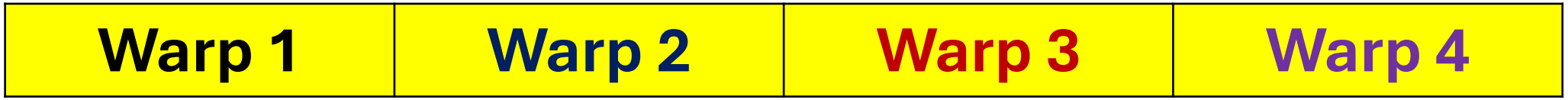

0
1
2
3

126
127


Warp 1
Warp 2
Warp 3
Warp 4


0
1
2
3
4

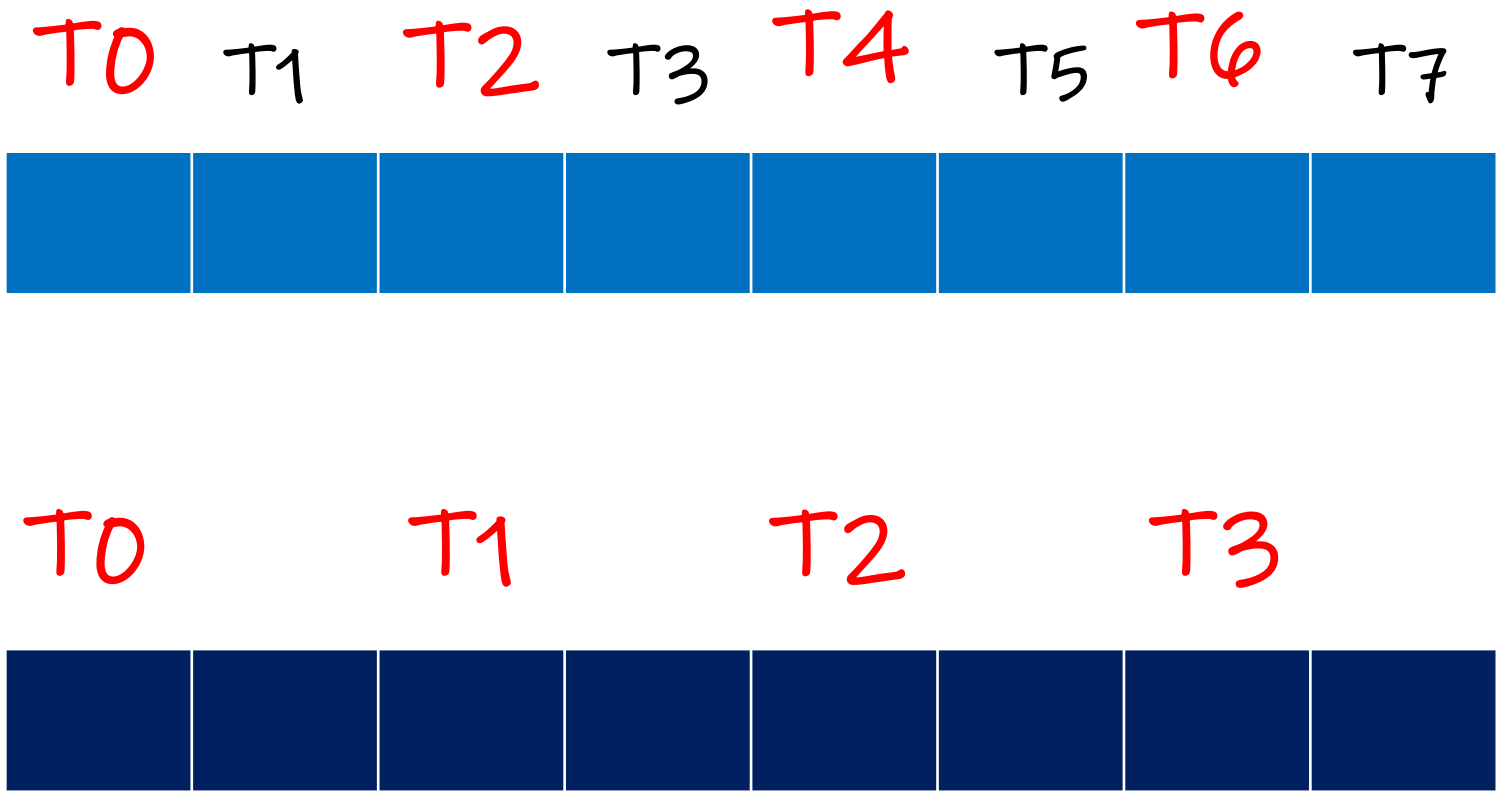
126
127


Warp 1
Warp 2
Warp 3
Warp 4

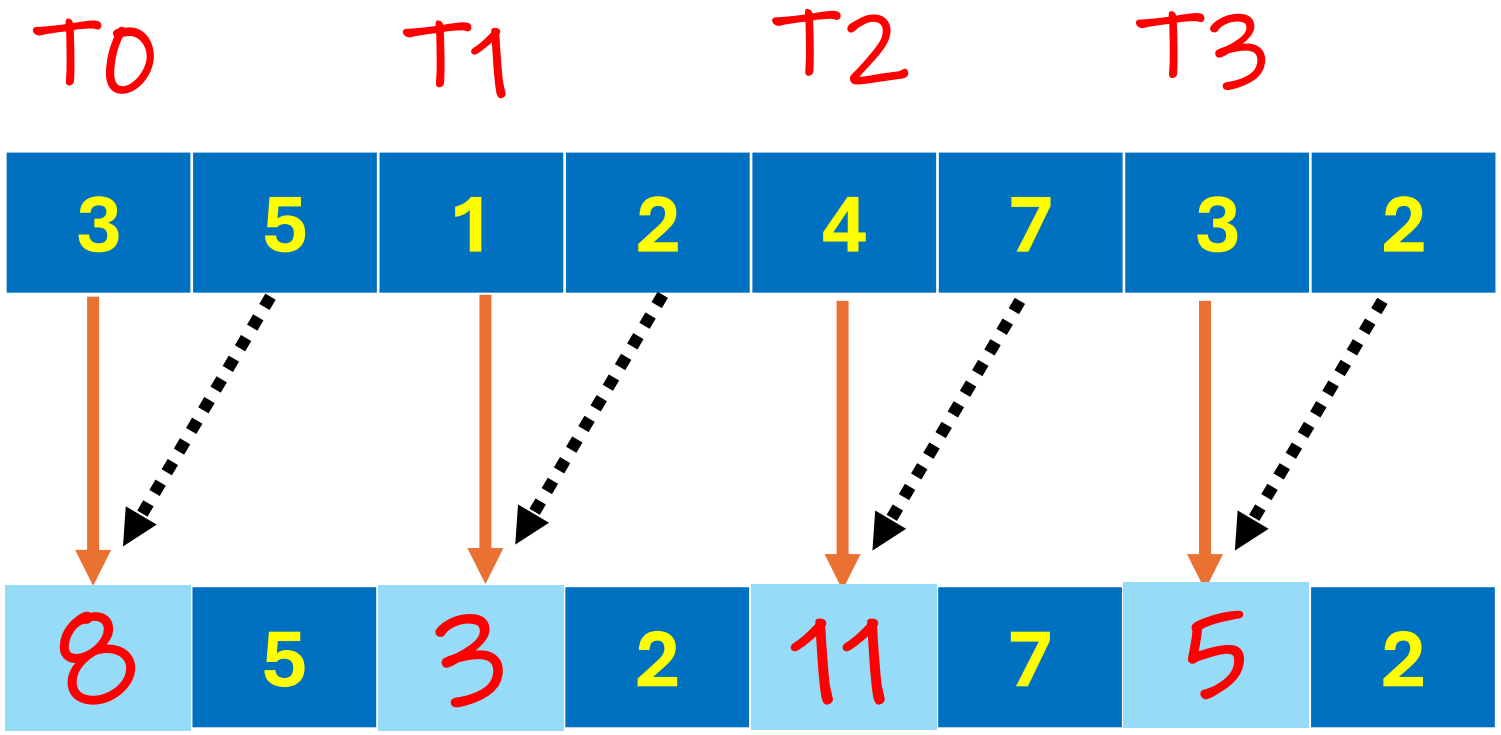
1. Forcer les threads voisins à effectuer la sommation

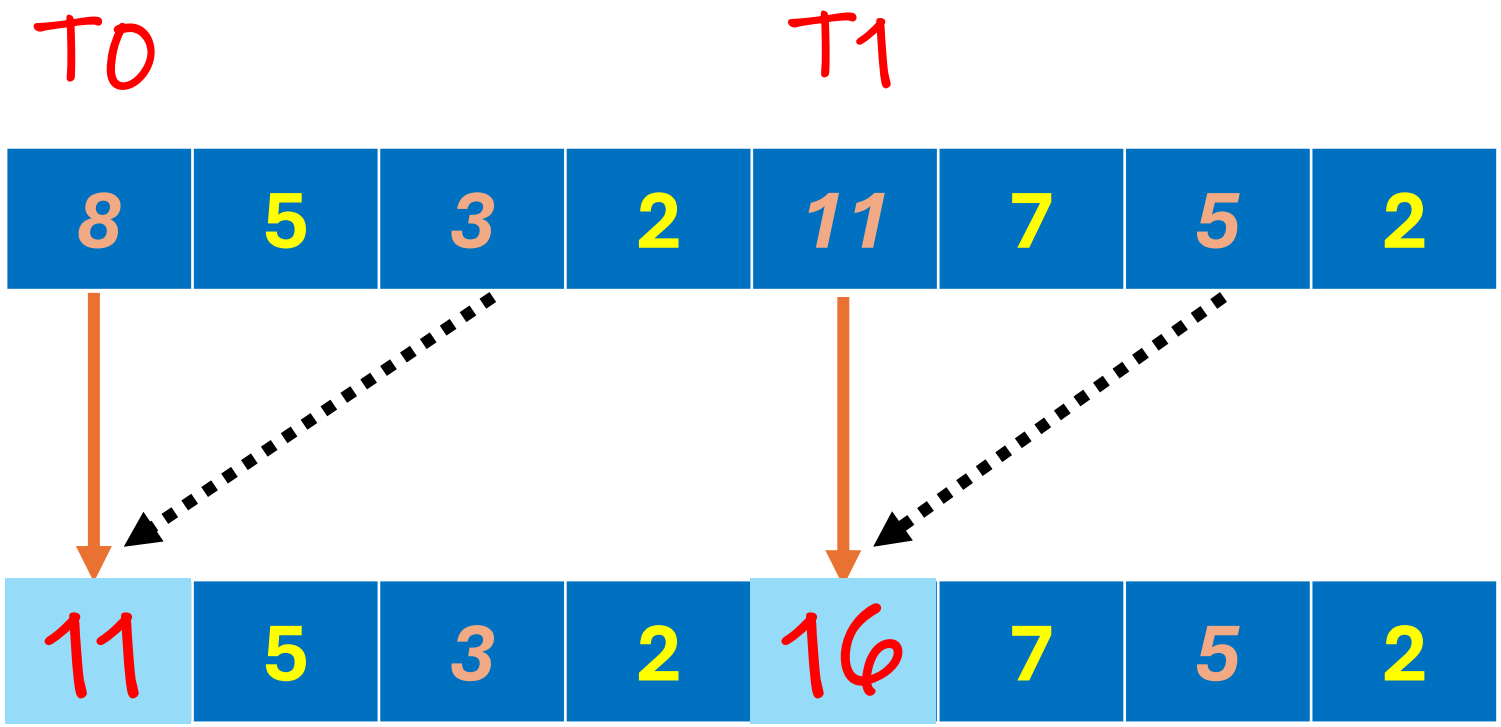
2. Paires entrelacées

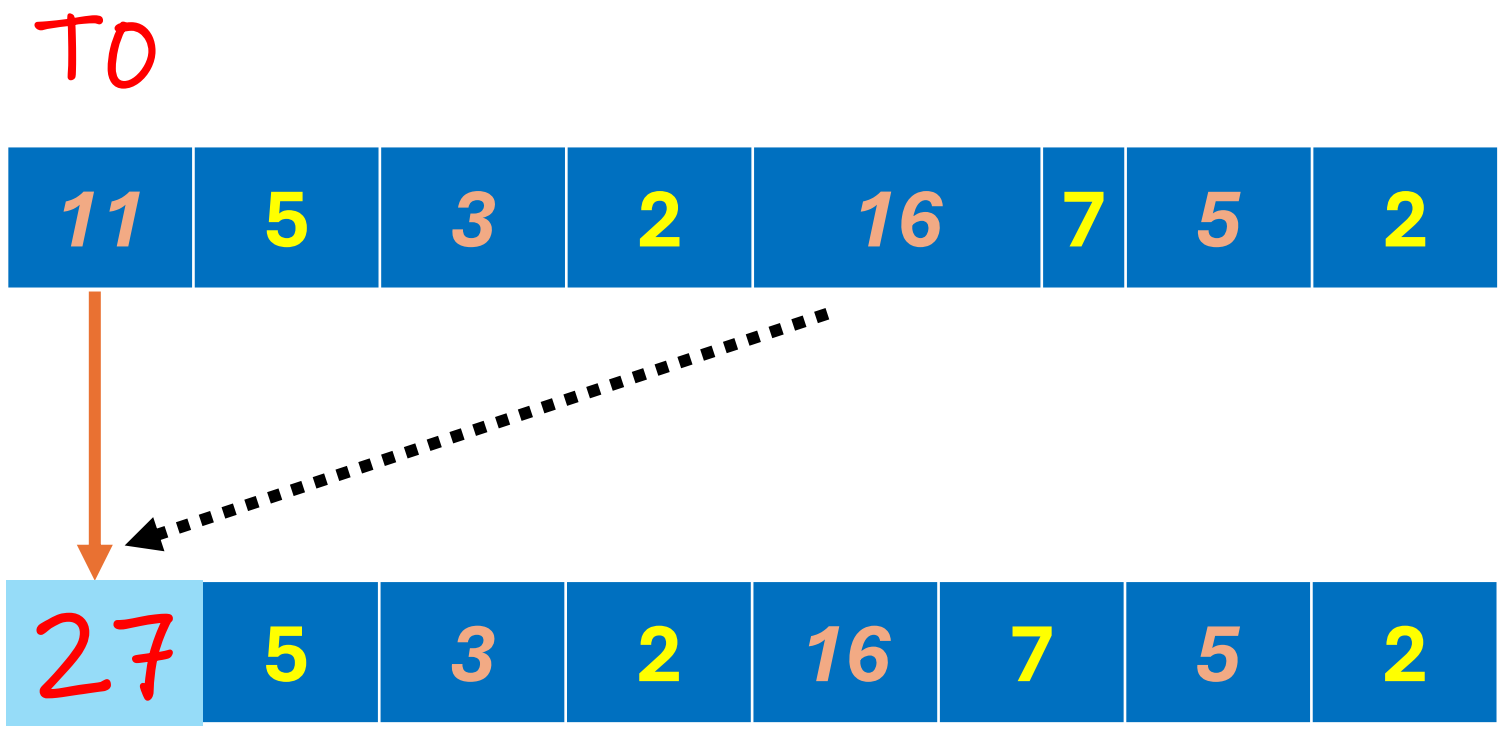
Réorganisation de l'index des fils

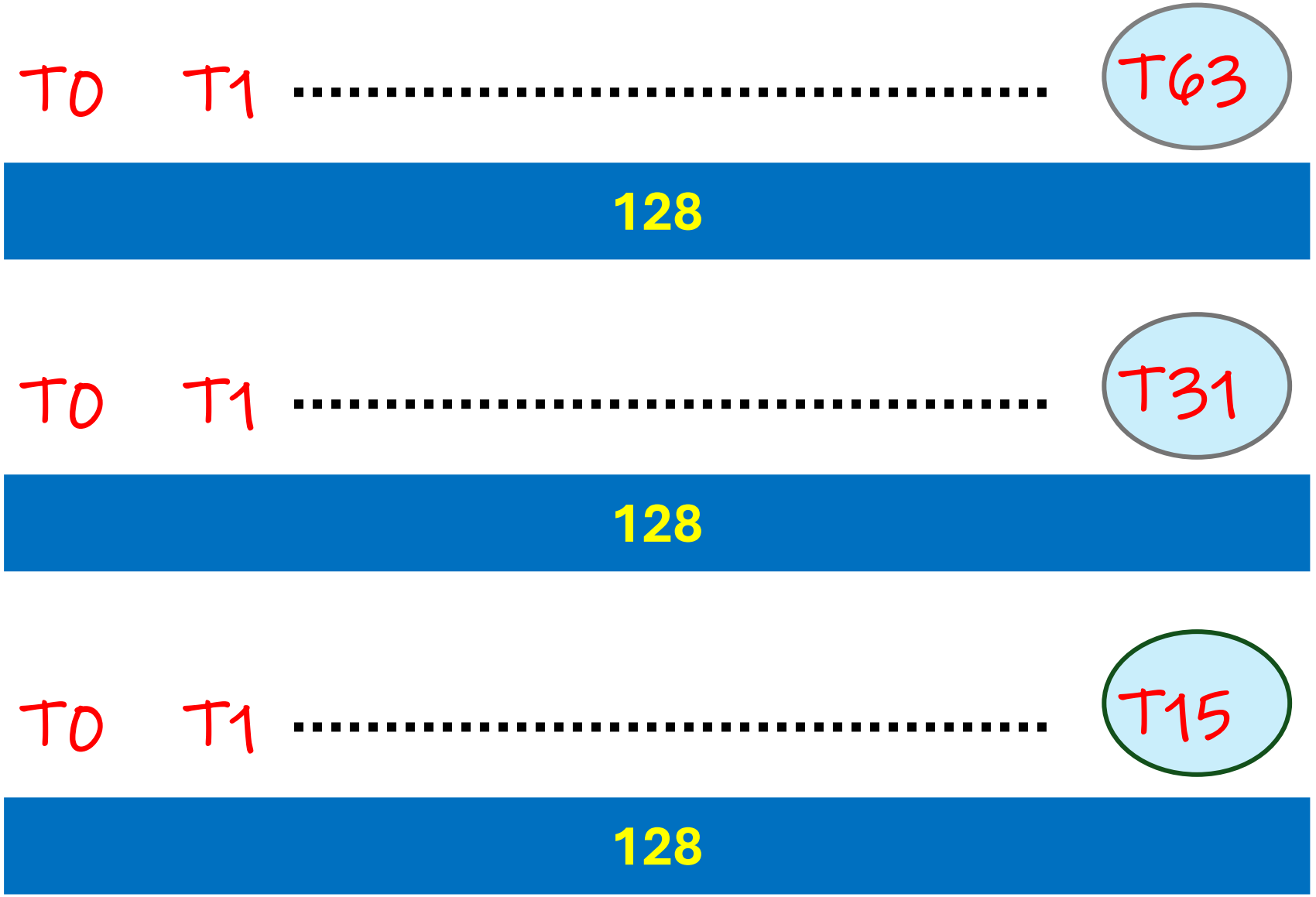


Nouvelle approche




T0
T1
8
5
3
2
11
7
5
2
11
5
3
2
16
7
5
2


To
11
5
3
2
16
7
5
2
27
5
3
2
16
7
5
2

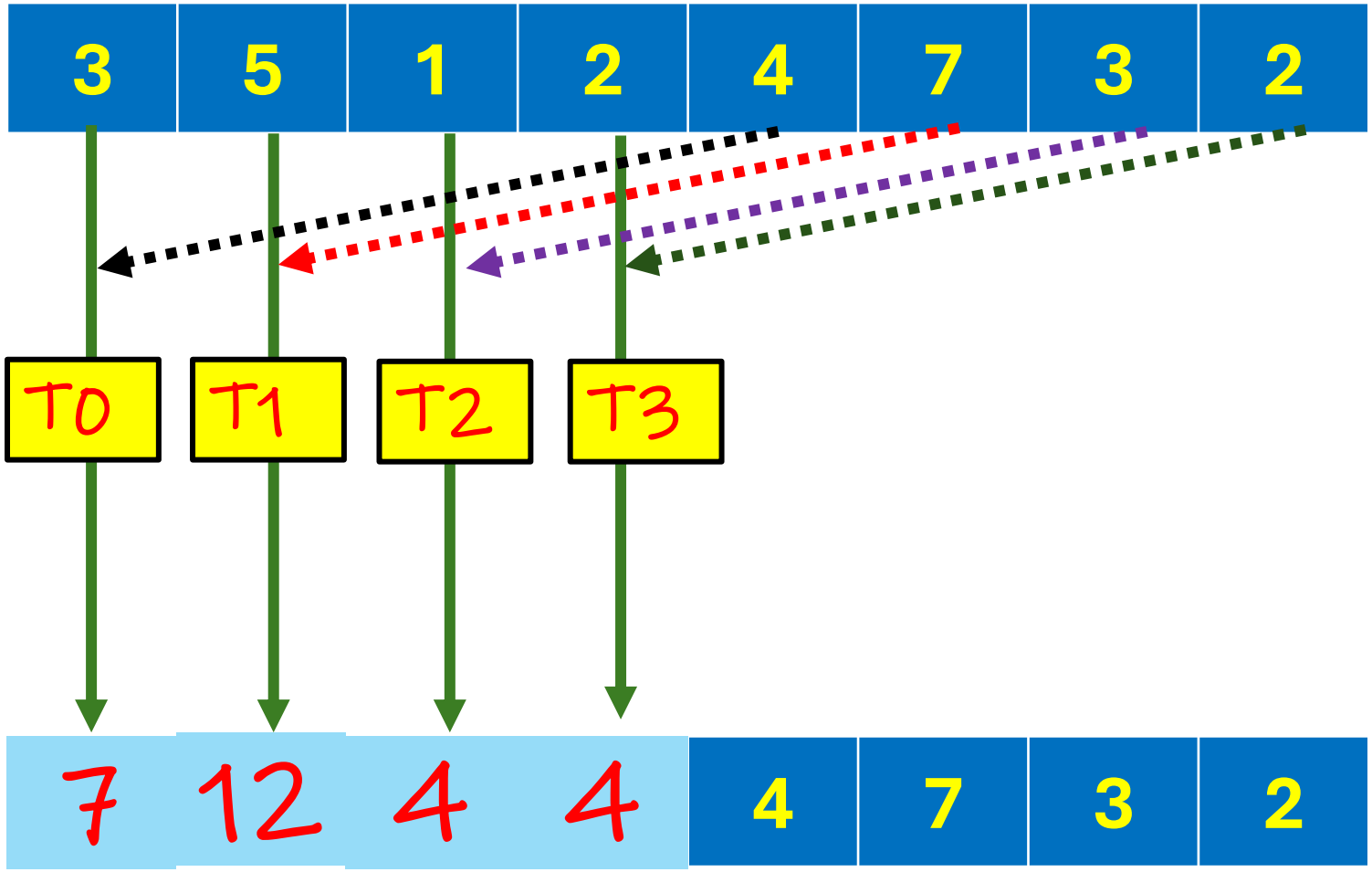

T0
T1
T63
128
T0
T1
T31
128
T0
T1
T15
128

code

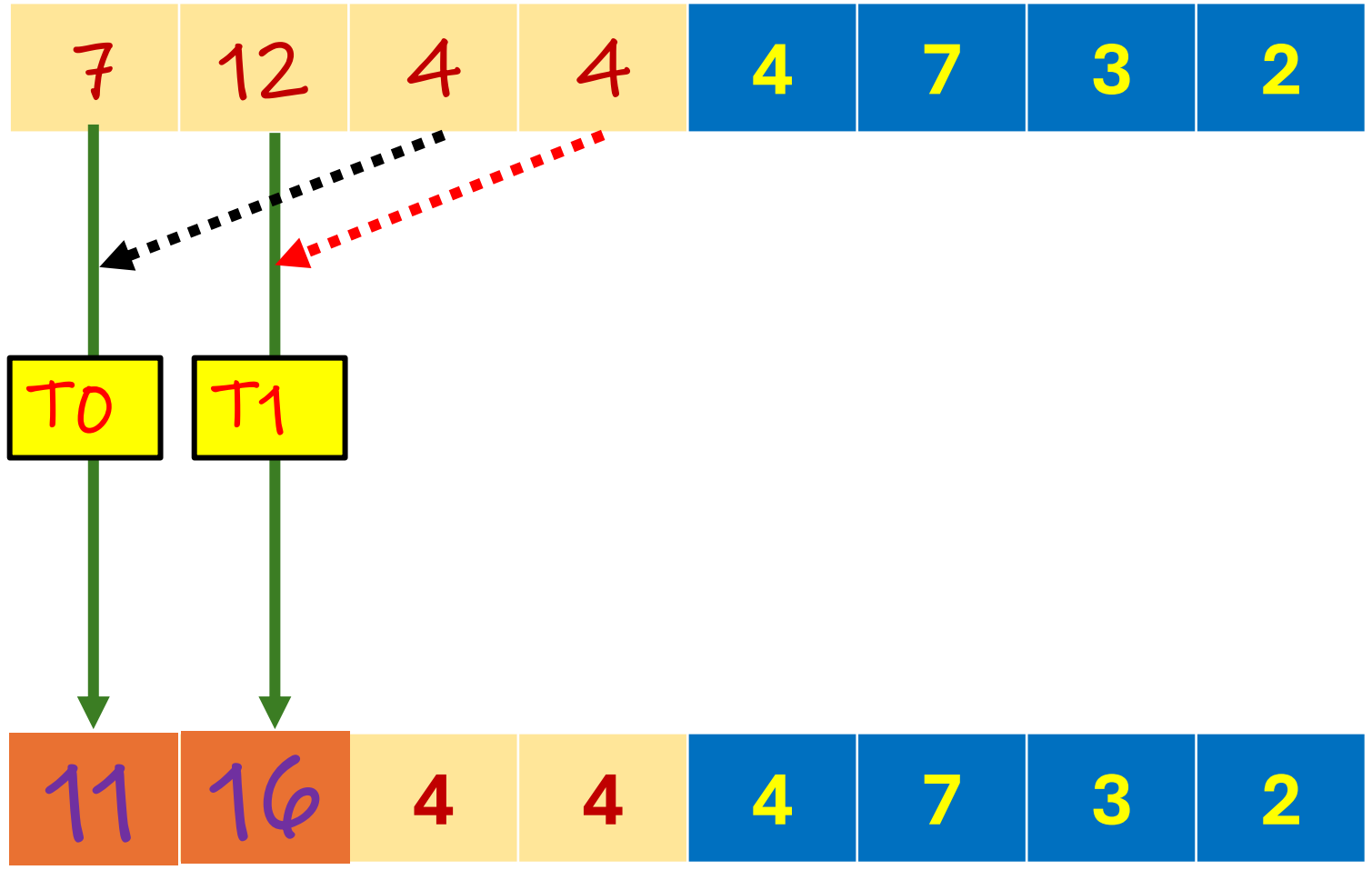
```
int * i_data = int_array + blockDim.x * blockIdx.x ;

for (int offset = 1 ; offset < blockDim.x ; offset *= 2)
{
        int index = 2 * offset * tid ;

        si (index < blockDim.x)
        {
                i_data[index] += i_data[index + offset] ;
        }
        __syncthreads() ;
}
```

Approche par paires entrelacées

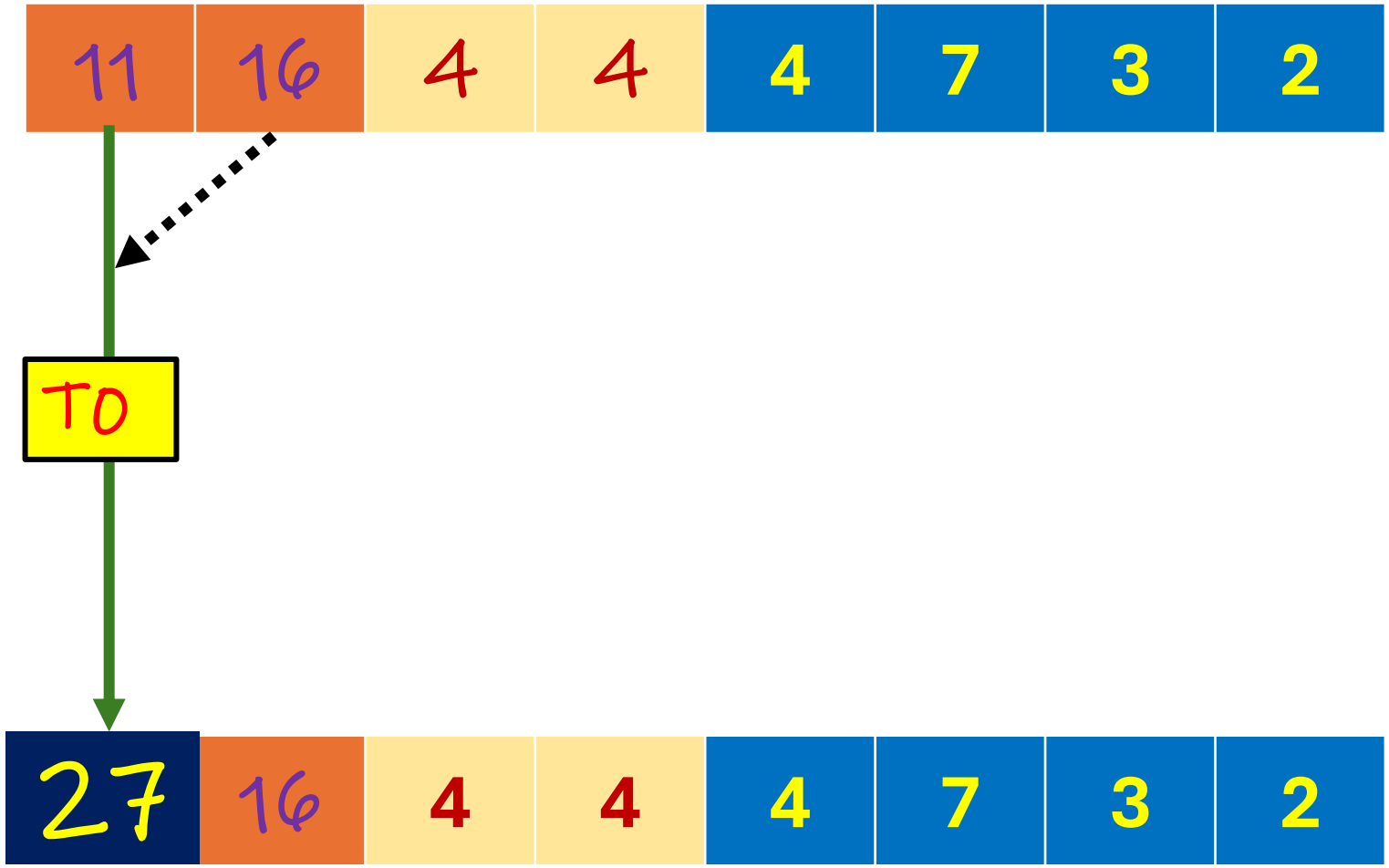
Approche par paires entrelacées



Approche par paires entrelacées



Approche par paires entrelacées



code

```
int * i_data = int_array + blockDim.x * blockIdx.x ;

for (int offset = blockDim.x / 2 ; offset > 0 ; offset /= 2)
{
        if (tid < offset)
        {
                i_data[index] += i_data[index + offset] ;
        }
        __syncthreads() ;
}
```

Déroulement

Qu'est-ce que l'enroulement de boucle ?

- Dans le déroulement de boucle, au lieu d'écrire le corps d'une boucle une seule fois et d'utiliser une boucle pour l'exécuter à plusieurs reprises, le corps est écrit dans le code plusieurs fois.

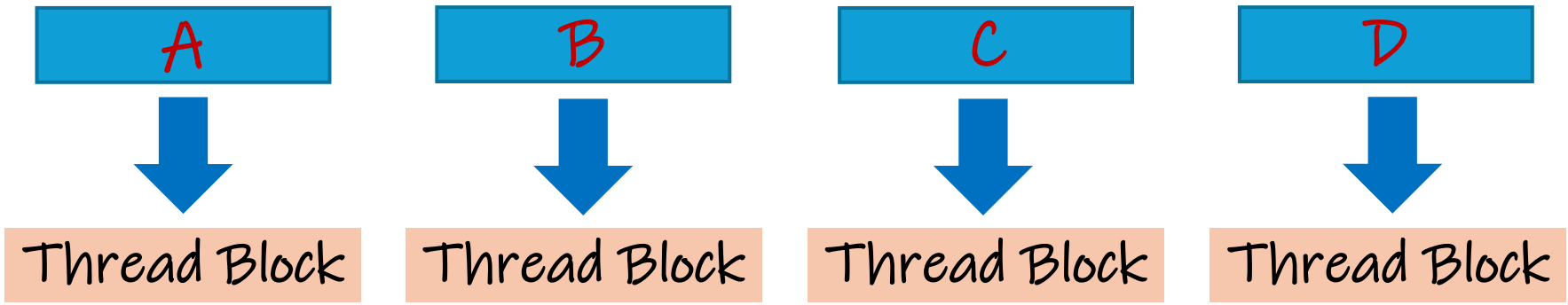
- Le nombre de copies du corps de la boucle est appelé facteur de déroulement de la boucle.

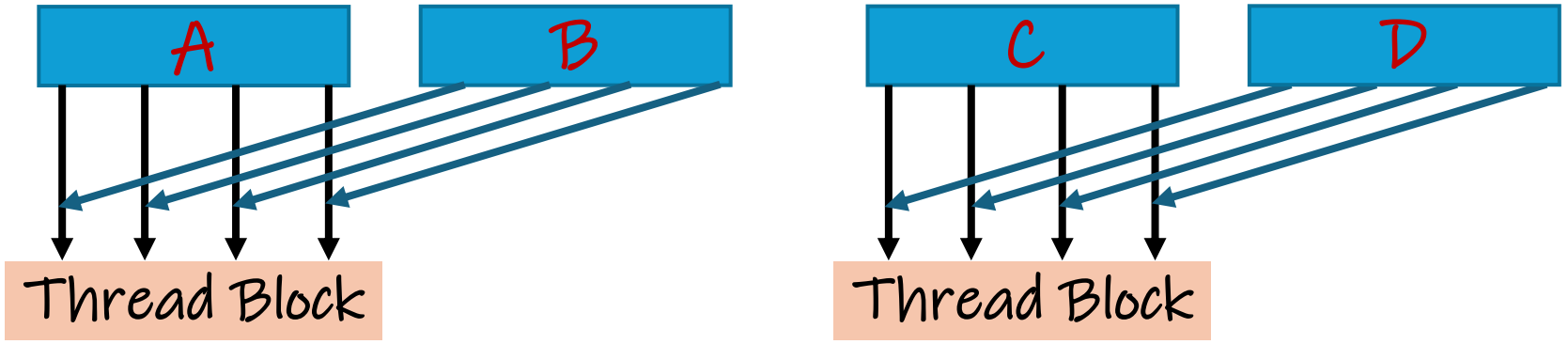
```
for ( int i = 0 ; i < 100 ; i++ )
{
        sum += a[i] ;
}
```

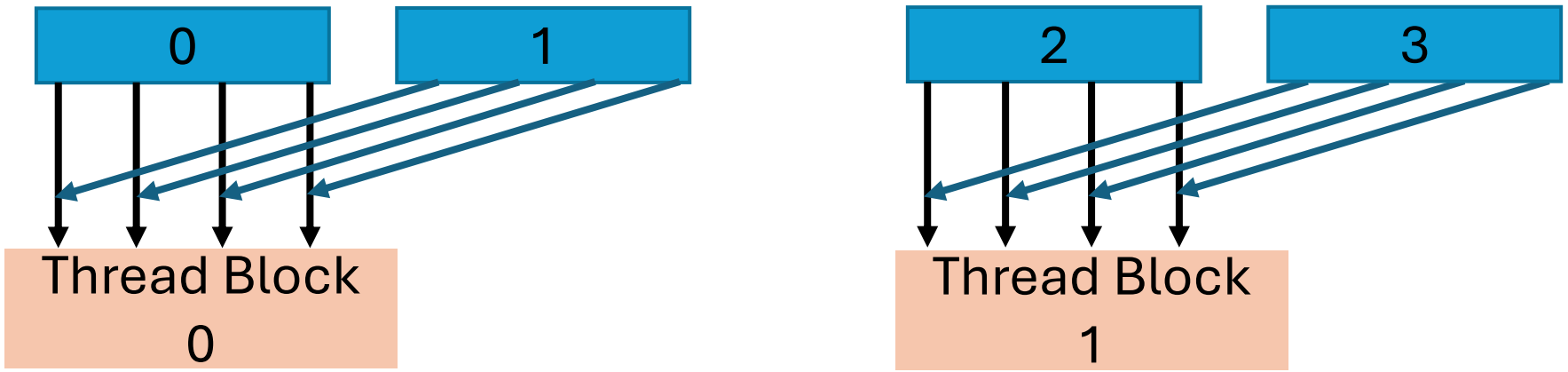
```
for ( int i = 0 ; i < 100 ; i += 2 )
{
        sum + = input[i] ;
        sum + = input[i+1] ;
}
```

Déroulement des blocs de fil

- Le déroulement que nous allons appliquer ici est quelque peu différent de notre exemple de déroulement de boucle dans un code séquentiel.
- Il s'agit du déroulement du bloc de fil.
- Mais l'objectif principal reste le même, à savoir la réduction des frais généraux d'instruction.


A
B
C
D
Thread Block
Thread Block
Thread Block
Thread Block


A
B
C
D
Thread Block
Thread Block


0
1
Thread Block
0
2
3
Thread Block
1

Parallélisme dynamique dans CUDA (1)

- **Approche précédente** : Tous les noyaux GPU étaient lancés à partir de l'hôte (CPU), la charge de travail étant entièrement gérée par le CPU.

- **Parallélisme dynamique** :
  - Permet aux noyaux de GPU de lancer et de synchroniser de nouveaux noyaux directement sur le GPU.
  - Permet d'ajouter du parallélisme de manière dynamique au sein du noyau lui-même à différentes étapes, améliorant ainsi la flexibilité.

Parallélisme dynamique dans CUDA (2)

- **Avantages** :
  - Prise en charge de la conception d'algorithmes hiérarchiques avec plusieurs niveaux de concurrence dans un noyau GPU.
  - Simplifie la mise en œuvre des algorithmes récursifs, en les rendant plus faciles à exprimer et à comprendre.
  - Les décisions relatives à la configuration de la grille et des blocs peuvent être reportées jusqu'au moment de l'exécution, ce qui permet d'optimiser l'utilisation des ressources sur la base de conditions déterminées par les données.
  - Réduit le besoin de contrôles fréquents et de transferts de données entre l'hôte (CPU) et le dispositif (GPU), améliorant ainsi l'efficacité en laissant le GPU gérer directement le travail.

Exécution imbriquée avec parallélisme dynamique (1)

- **Grilles des parents et des enfants** :
  - Un **fil**, un **bloc** ou une **grille parent** peut lancer une **grille enfant**.
  - La **grille de l'enfant** doit être complétée avant que le parent ne soit considéré comme terminé.
  - La synchronisation entre les grilles des parents et des enfants est implicite, mais une barrière est souvent mise en place pour une synchronisation explicite.

Exécution imbriquée avec parallélisme dynamique (2)

- **Synchronisation des grilles et des threads** :
  - Les grilles enfants lancées par les threads d'un bloc doivent se terminer avant que le bloc ne se termine.
  - Les opérations de mémoire globale du parent sont visibles par la grille de l'enfant, et vice versa, une fois la synchronisation terminée.

- **Cohérence de la mémoire** :
  - Les grilles parents et enfants partagent l'accès à la **mémoire globale**, mais la **mémoire partagée** et la **mémoire locale** sont privées pour le thread ou le bloc de threads.
  - La cohérence de la mémoire est garantie lorsque la grille enfant démarre et se termine, ce qui assure la visibilité des opérations entre le parent et l'enfant.

Portée d'une grille parentale et d'une grille enfant

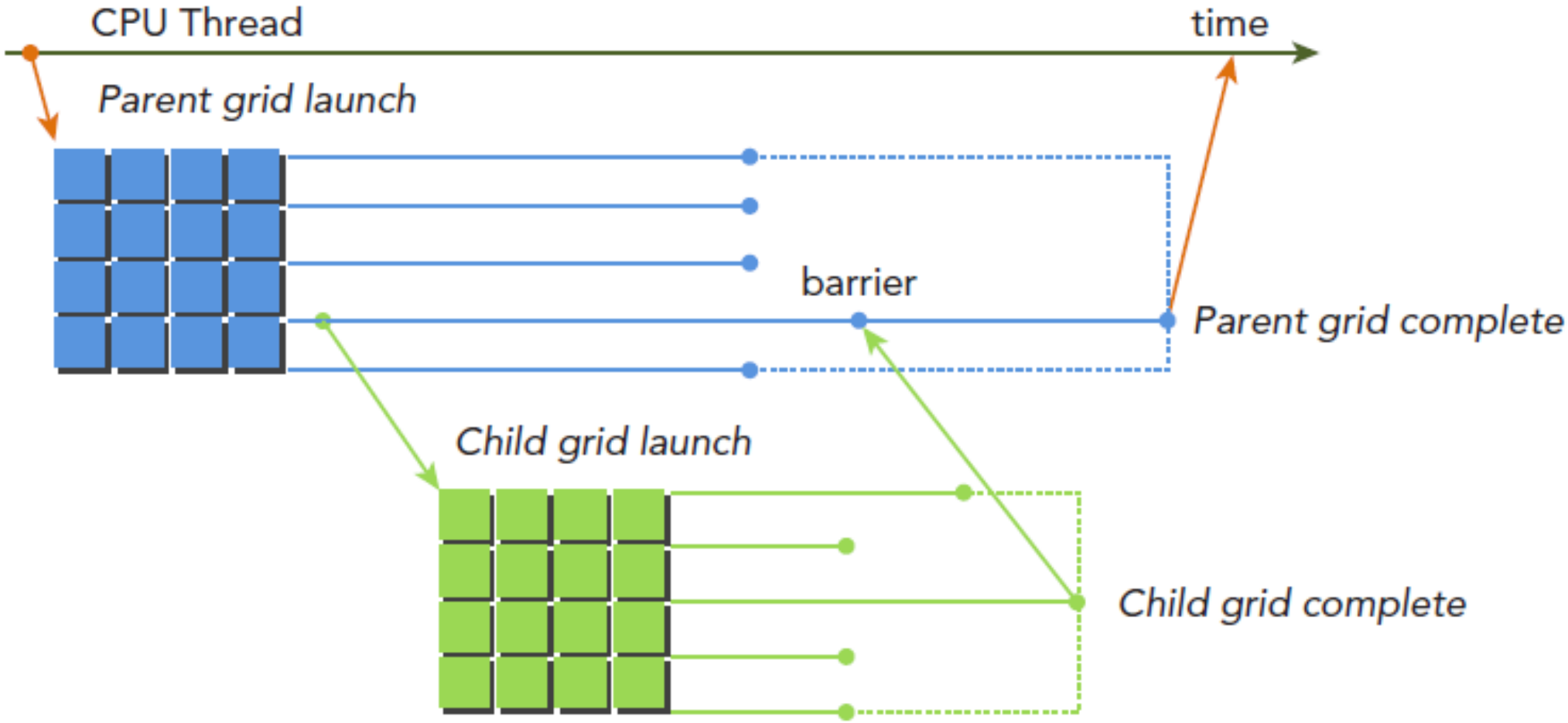


FIGURE 3-26

Décomposition du diagramme (1)

- **Lancement de la grille des parents (bleu)** :
  - La **grille mère** est lancée par le thread de l'unité centrale et commence à s'exécuter.
  - Chaque carré représente un fil de la grille mère, organisé en blocs.

- **Barrière** :
  - Une **barrière** est en place, qui garantit que la grille mère attend à ce stade que toutes les grilles enfants lancées aient terminé leurs tâches.
  - Il s'agit d'un point de synchronisation explicite, qui garantit que la grille mère ne s'achève pas tant que toutes les grilles enfants n'ont pas terminé leur travail.

Décomposition du diagramme (2)

- **Lancement de la grille enfant (vert)** :
  - La **grille enfant** est lancée par un thread de la grille parent (indiqué par la flèche verte).
  - La grille enfant fonctionne de manière indépendante mais doit être terminée avant que la grille parentale ne puisse franchir la barrière.

- **Remplissage de la grille des parents** :
  - Une fois que tous les threads de la grille mère, y compris les grilles enfants qu'ils ont lancées, ont terminé leur exécution, la grille mère peut enfin s'achever.

Décomposition du diagramme (3)

- **Concepts clés** :
  - **Exécution imbriquée** : La grille mère lance une grille enfant pendant son exécution. La grille mère s'arrête à la barrière jusqu'à ce que toutes les tâches de la grille enfant soient terminées.
  - **Synchronisation** : Une synchronisation implicite a lieu entre la grille mère et la grille enfant, garantissant que les modifications de la mémoire globale effectuées par une grille sont visibles par l'autre grille à des moments critiques (comme le lancement et l'achèvement de la grille).
  - **Parallélisme** : Les grilles parentales et enfantines peuvent être exécutées simultanément, ce qui permet une exécution hiérarchique plus complexe des tâches parallèles sur le GPU.

En résumé, le diagramme montre comment les tâches sont imbriquées dans un parallélisme dynamique, la grille mère contrôlant le lancement de la grille enfant et attendant son achèvement avant de poursuivre l'exécution.

RÉSUMÉ (1)

- **Modèle d'exécution CUDA sur les GPU** :
  - **SIMT (Single Instruction Multiple Threads**) : Les threads s'exécutent en chaîne.
  - **Partitionnement des ressources matérielles**** : Réparties en blocs et en threads.

- **Optimiser les performances** :
  - Contrôler le parallélisme et l'utilisation de la bande passante de la mémoire.
  - Tenir compte des limites du matériel GPU ; utiliser l'**heuristique des grilles** et des **blocs** pour améliorer les performances.

RÉSUMÉ (2)

- **Parallélisme dynamique** :
  - Permet de créer de nouvelles œuvres directement à partir du GPU.
  - Convient aux algorithmes récursifs ou dépendant des données.
  - Principales considérations :
  - Stratégie de lancement de la grille de l'enfant.
    - Synchronisation parents-enfants.
    - Profondeur des appels au noyau imbriqués.

- **Profilage du noyau avec `nvprof`** :
  - Indispensable pour identifier les goulets d'étranglement en matière de performances.
  - Approche axée sur les profils pour améliorer l'efficacité des noyaux.