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

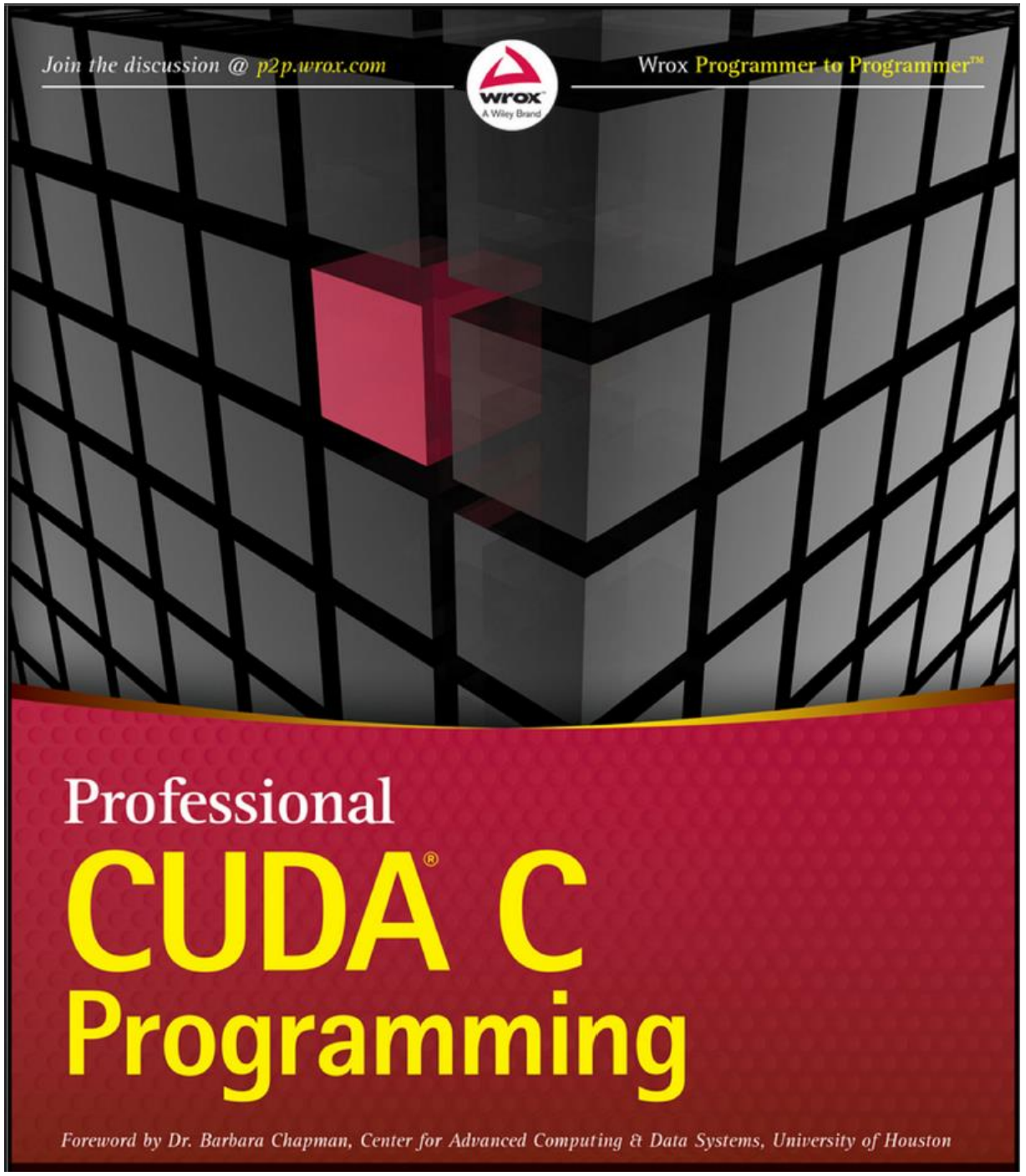
School of Electrical Engineering
and Computer Science

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 2 : Modèle de programmation CUDA

Plan

- Écriture d'un programme CUDA
- Exécution d'une fonction noyau
- Organiser les threads avec des grilles et des blocs
- Mesurer les performances du GPU

Présentation du modèle de programmation CUDA

- Les modèles de programmation présentent une abstraction des architectures informatiques qui sert de pont entre une application et sa mise en œuvre sur le matériel disponible.

- La figure 2-1 illustre les couches d'abstraction importantes qui se situent entre le programme et l'implémentation du modèle de programmation.

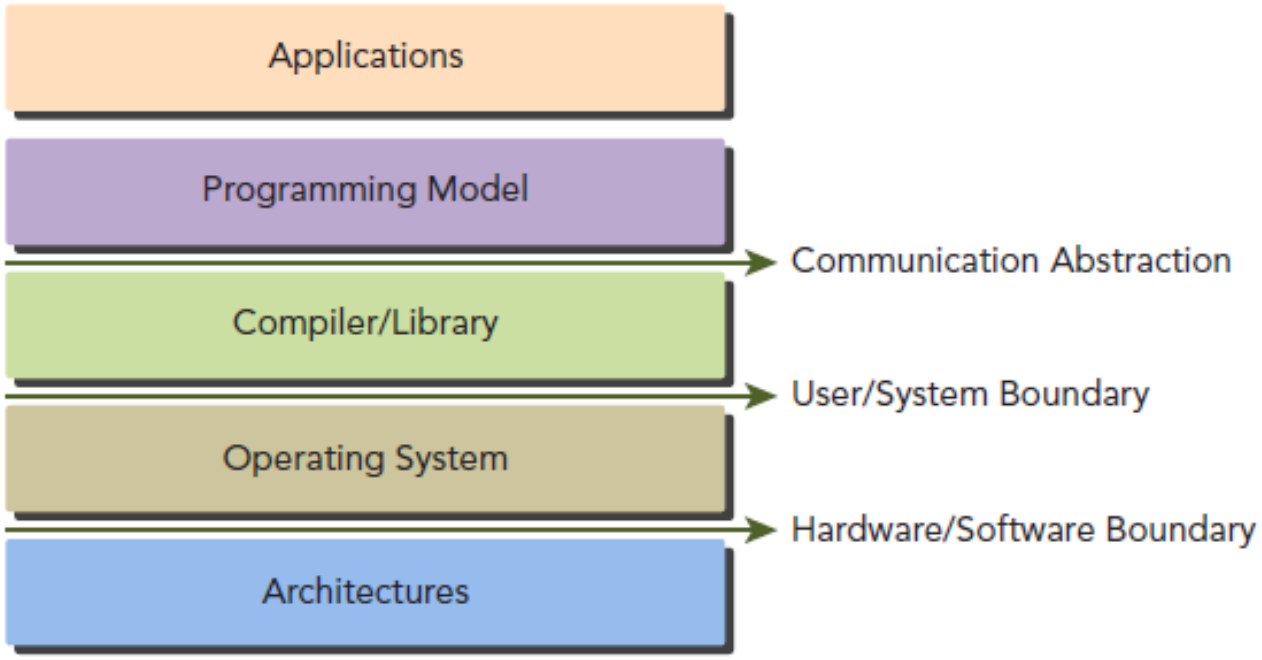


FIGURE 2-1

CUDA Programming Structure

- Un environnement hétérogène se compose de CPU complétés par des GPU, chacun disposant de sa propre mémoire séparée par un bus PCI-Express.
  - Hôte : le CPU et sa mémoire (mémoire hôte)
  - Dispositif : le GPU et sa mémoire (mémoire du dispositif)

- Comme le montre la figure 2-2, le code série (ainsi que le code parallèle à la tâche) est exécuté sur l'hôte, tandis que le code parallèle est exécuté sur l'unité GPU.

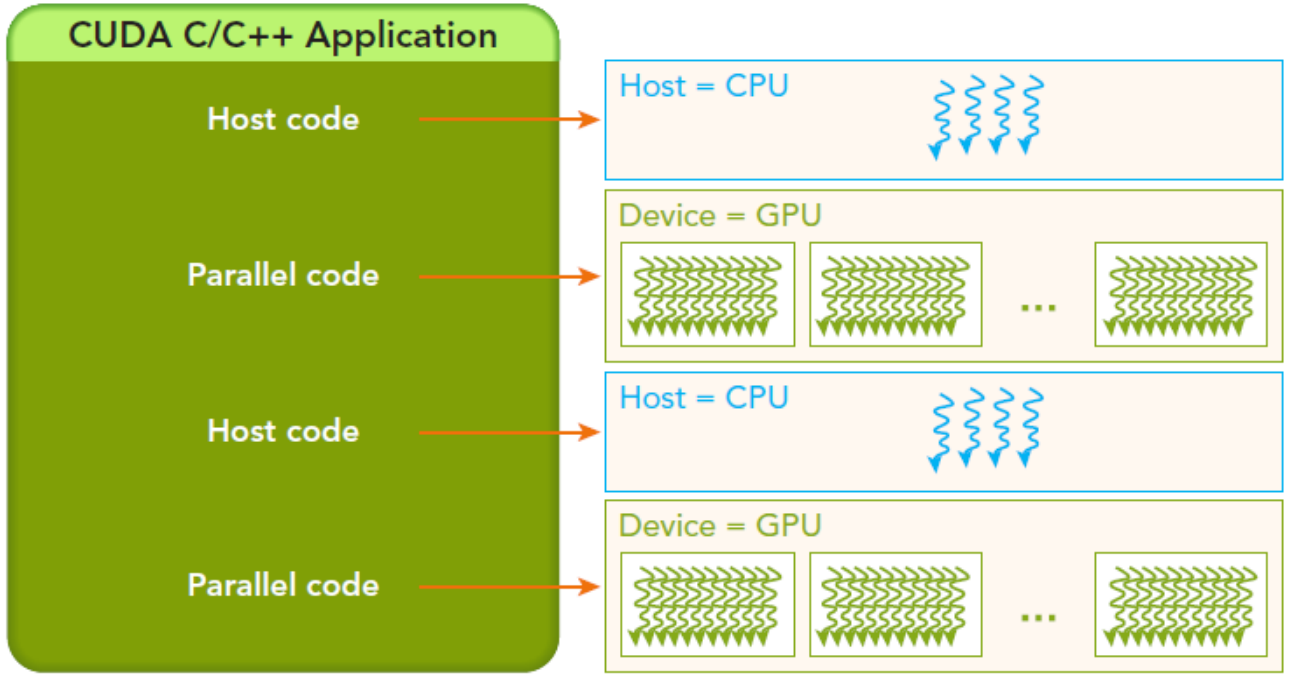


FIGURE 2-2

Gestion de la mémoire

- Le modèle de programmation CUDA suppose un système composé d'un hôte et d'un périphérique, chacun disposant de sa propre mémoire.

- Pour vous permettre d'avoir un contrôle total et d'obtenir les meilleures performances, le runtime CUDA fournit des fonctions pour allouer la mémoire des périphériques, libérer la mémoire des périphériques et transférer des données entre la mémoire de l'hôte et la mémoire des périphériques.

- Le Tableau 2-1 répertorie les fonctions C standard et les fonctions CUDA C correspondantes pour les opérations de mémoire.

TABLE 2-1: Host and Device Memory Functions

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

Structure de la mémoire du GPU

La figure 2-3 illustre la structure simplifiée de la mémoire du GPU, qui comprend deux éléments principaux : la mémoire globale et la mémoire partagée.

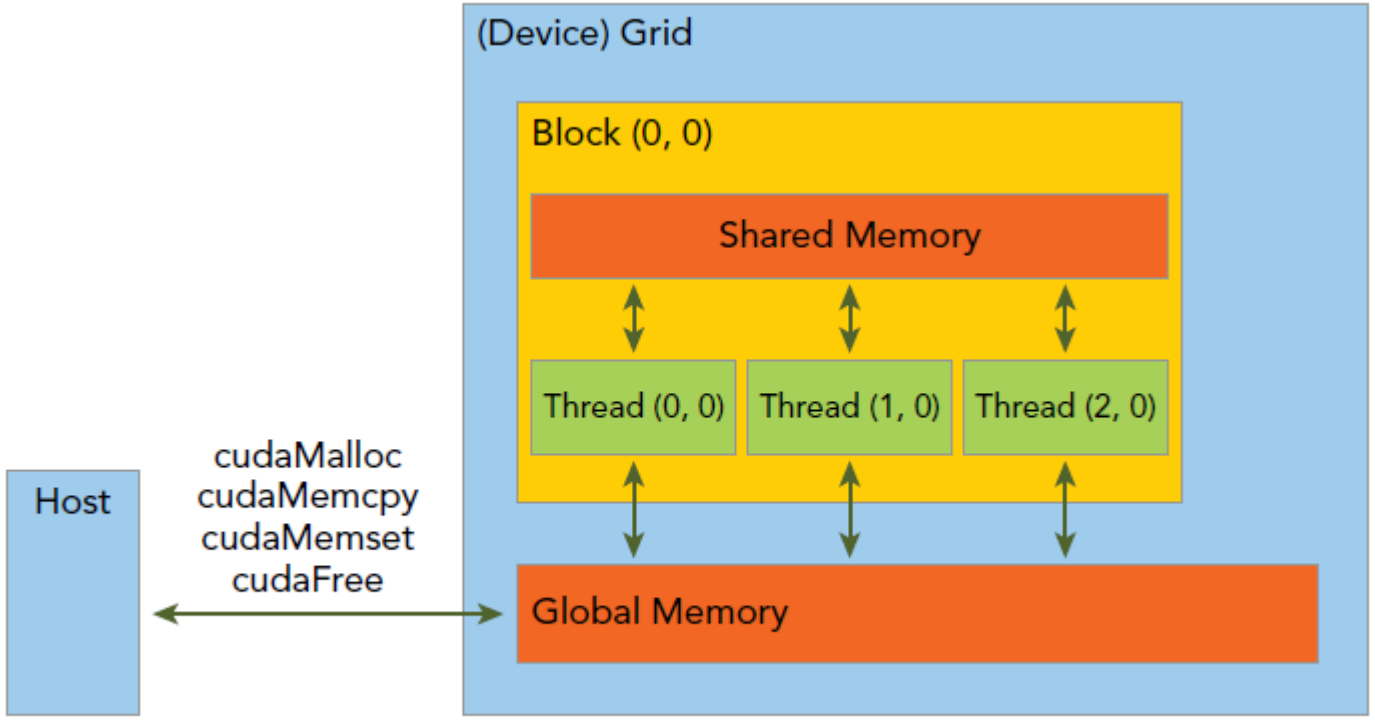


FIGURE 2-3

Addition de deux tableaux

- Comme l'illustre la figure 2-4, le premier élément du tableau a est ajouté au premier élément du tableau b, et le résultat est affecté au premier élément du tableau c.

- Ce calcul est répété pour tous les éléments successifs du tableau.

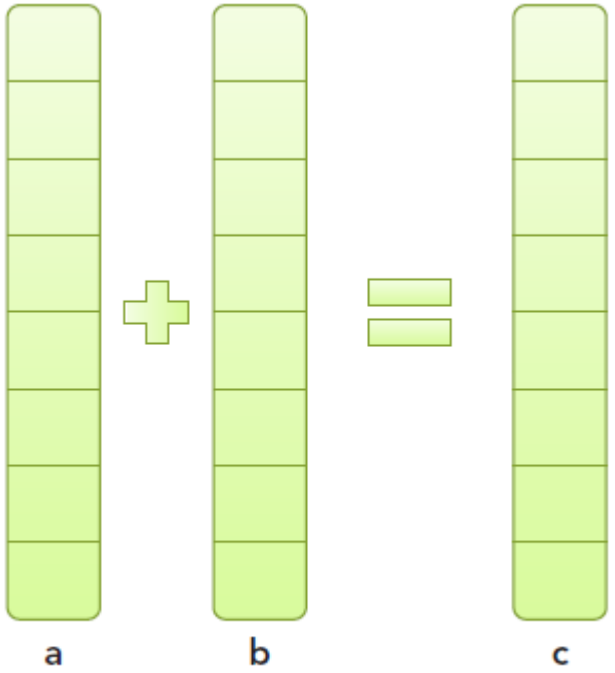


FIGURE 2-4

Liste 2-1 : Somme de tableaux basée sur l'hôte (1/3)

```
#include <stdlib.h>
#include <time.h>

/*
 * This example demonstrates a simple vector sum on the host. sumArraysOnHost
 * sequentially iterates through vector elements on the host.
 */

void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
   for (int idx = 0; idx < N; idx++)
   {
      C[idx] = A[idx] + B[idx];
   }
}
```

Liste 2-1 : Somme de tableaux basée sur l'hôte (2/3)

```
void initialData(float *ip, int size)
{
   // generate different seed for random number
   time_t t;
   srand((unsigned) time(&t));

   for (int i = 0; i < size; i++)
   {
      ip[i] = (float)(rand() & 0xFF) / 10.0f;
   }
   return;
}
```

Liste 2-1 : Somme de tableaux basée sur l'hôte (3/3)

```
int main(int argc, char **argv)
{
   int nElem = 1024;
   size_t nBytes = nElem * sizeof(float);

   float *h_A, *h_B, *h_C;
   h_A = (float *)malloc(nBytes);
   h_B = (float *)malloc(nBytes);
   h_C = (float *)malloc(nBytes);

   initialData(h_A, nElem);
   initialData(h_B, nElem);

   sumArraysOnHost(h_A, h_B, h_C, nElem);

   free(h_A);
   free(h_B);
   free(h_C);

   return(0);
}
```

Organisation des threads

- Savoir organiser les threads est un élément essentiel de la programmation CUDA.
- CUDA expose une abstraction de hiérarchie de threads pour vous permettre d'organiser vos threads.
- Il s'agit d'une hiérarchie de threads à deux niveaux, décomposée en blocs de threads et en grilles de blocs, comme le montre la figure 2-5.

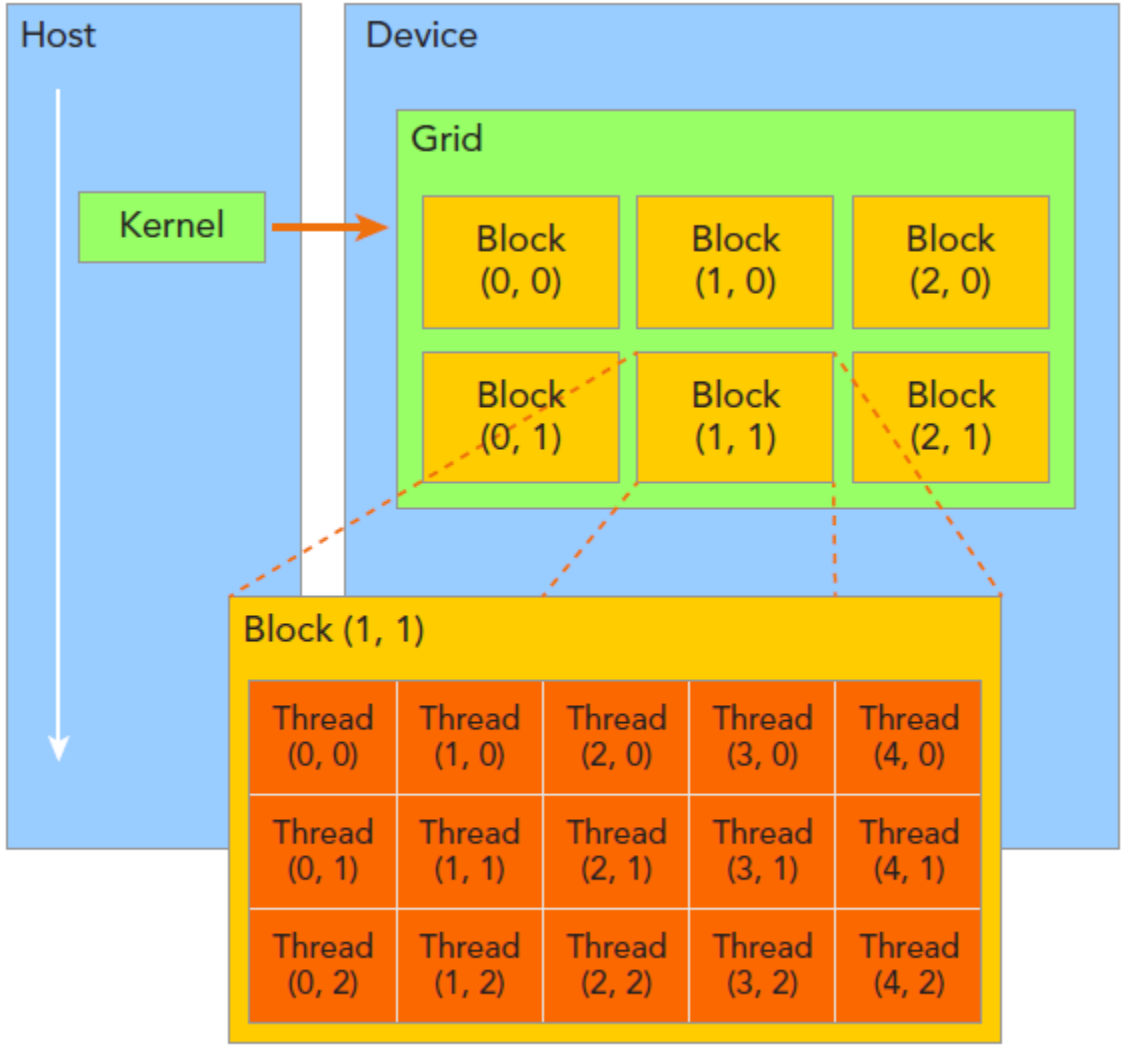


FIGURE 2-5

Organisation des threads

- Tous les threads générés par un seul lancement du noyau sont collectivement appelés une grille.
- Tous les threads d'une grille partagent le même espace mémoire global.
- Une grille est composée de plusieurs blocs de threads.
- Un bloc de threads est un groupe de threads qui peuvent coopérer les uns avec les autres en utilisant :
  - Synchronisation bloc-local
  - Mémoire partagée bloc-local
- Les threads de discussion de blocs différents ne peuvent pas coopérer.
- Les threads s'appuient sur les deux coordonnées uniques suivantes pour se distinguer les uns des autres :
  - blockIdx (index du bloc dans une grille)
  - threadIdx (index du thread dans un bloc)

Organisation des threads

- La variable de coordonnées est de type uint3, un type de vecteur intégré à CUDA, dérivé du type entier de base.

- Il s'agit d'une structure contenant trois entiers non signés, dont les 1ère, 2ème et 3ème composante sont accessibles via les champs x, y et z respectivement.
  - blockIdx.x
  - blockIdx.y
  - blockIdx.z

  - threadIdx.x
  - threadIdx.y
  - threadIdx.z

Organisation des threads

- CUDA organise les grilles et les blocs en trois dimensions.
- La figure 2-5 montre un exemple de structure hiérarchique de threads avec une grille 2D contenant des blocs 2D.
- Les dimensions d'une grille et d'un bloc sont spécifiées par les deux variables intégrées suivantes :
  - blockDim (dimension du bloc, mesurée en threads)
  - gridDim (dimension de la grille, mesurée en blocs)

- CUDA organise les grilles et les blocs en trois dimensions.
- La figure 2-5 montre un exemple de structure hiérarchique de threads avec une grille 2D contenant des blocs 2D.
- Les dimensions d'une grille et d'un bloc sont spécifiées par les deux variables intégrées suivantes :
  - blockDim (dimension du bloc, mesurée en threads)
  - gridDim (dimension de la grille, mesurée en blocs)

Organisation des threads

- Ces variables sont de type dim3, un type de vecteur d'entiers basé sur uint3 qui est utilisé pour spécifier les dimensions.

- Lors de la définition d'une variable de type dim3, toute composante non spécifiée est initialisée à 1.

- Chaque composante d'une variable de type dim3 est accessible par ses champs x, y et z, respectivement, comme le montre l'exemple suivant :
  - blockDim.x
  - blockDim.y
  - blockDim.z

Dimensions des grilles et des blocs

- En général, une grille est organisée comme un tableau 2D de blocs, et un bloc est organisé comme un tableau 3D de fils.
- Les grilles et les blocs utilisent le type dim3 avec trois champs entiers non signés.
- Les champs inutilisés sont initialisés à 1 et ignorés.

Lancement d'un noyau CUDA

- Vous connaissez la syntaxe d'appel de fonction C suivante :

  function_name (argument list);

- Un appel de noyau CUDA est une extension directe de la syntaxe des fonctions C qui ajoute la configuration d'exécution d'un noyau à l'intérieur de parenthèses à triple angle :

  kernel_name <<<grid, block>>>(argument list);

- La première valeur de la configuration d'exécution est la dimension de la grille, le nombre de blocs à lancer. La deuxième valeur est la dimension du bloc, c'est-à-dire le nombre de threads dans chaque bloc. En spécifiant les dimensions de la grille et du bloc, vous configurez :
  - Le nombre total de threads pour un noyau
  - La disposition des threads que vous souhaitez employer pour un noyau

- Pour un problème donné, vous pouvez utiliser une grille et une disposition de bloc différentes pour organiser vos threads.
- Par exemple, supposons que vous disposiez de 32 éléments de données pour un calcul.
- Vous pouvez regrouper 8 éléments dans chaque bloc et lancer quatre blocs comme suit :

  kernel_name<<<4, 8>>>(argument list);

Disposition des threads dans une configuration donnée

La figure 2-6 illustre la disposition des fils dans la configuration ci-dessus

kernel_name<<<4, 8>>> (argument list);

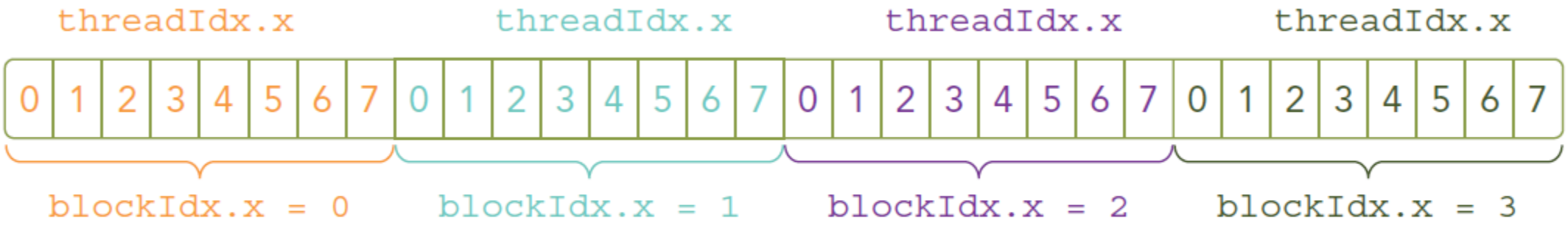
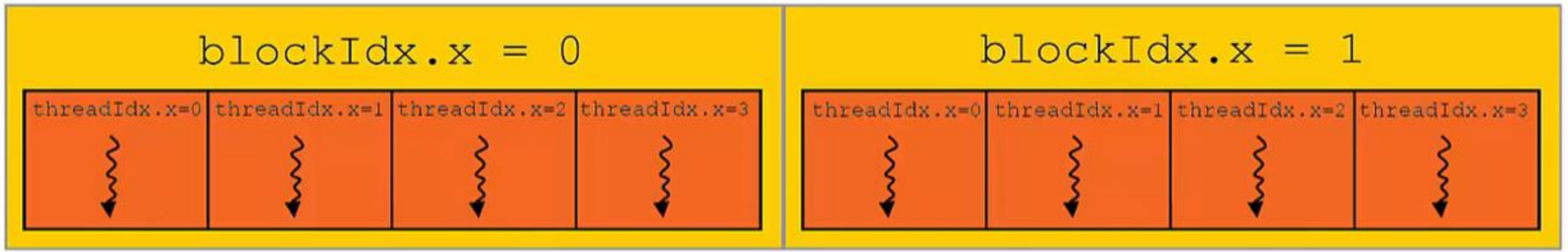


FIGURE 2-6

Lecture2_example_4

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
__global__ void kernel_name() {
    // Each thread can identify itself by its block and thread index
    int block_id = blockIdx.x;
    int thread_id = threadIdx.x;

    // Compute the global thread ID
    int global_thread_id = block_id * blockDim.x + thread_id;

    // Print out thread and block information
    printf(_Format: "Block %d, Thread %d, Global Thread ID %d\n", block_id, thread_id, global_thread_id);
}
int main() {
    // Define grid and block sizes
    int num_blocks = 4;
    int threads_per_block = 8;

    // Launch kernel with <<<4, 8>>> configuration
    kernel_name << <num_blocks, threads_per_block >> > ();

    // Wait for all threads to complete
    cudaDeviceSynchronize();

    return 0;
}
```

Indexation à l'intérieur de la grille




```
i = threadIdx.x + blockIdx.x * blockDim.x;
```

i	threadIdx.x	blockIdx.x * blockDim.x
0	0	0 * 4 = 0
1	1	0 * 4 = 0
2	2	0 * 4 = 0
3	3	0 * 4 = 0
4	0	1 * 4 = 4
5	1	1 * 4 = 4
6	2	1 * 4 = 4
7	3	1 * 4 = 4

Indexation à l'intérieur de la grille

Examples

```
// Launch Kernel
kernel<<< 3, 4 >>>(a);
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockDim.x;
}
```

```
a: 4 4 4 4 4 4 4 4 4 4 4 4
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = threadIdx.x;
}
```

```
a: 0 1 2 3 0 1 2 3 0 1 2 3
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockIdx.x;
}
```

```
a: 0 0 0 0 1 1 1 1 2 2 2 2
```

```
__global__ void kernel( int* a ) {
      int i = threadIdx.x + blockIdx.x * blockDim.x;
      a[i] = i;
}
```

```
a: 0 1 2 3 4 5 6 7 8 9 10 11
```

Lecture2_example_3 : Indexation dans la grille

Disposition des threads dans une configuration donnée

Les données étant stockées linéairement dans la mémoire globale, vous pouvez utiliser les variables intégrées blockIdx.x et threadIdx.x pour :

- Identifier un fil unique dans la grille.
- Établir une correspondance entre les threads et les éléments de données.

Si vous regroupez les 32 éléments en un seul bloc, vous n'aurez qu'un seul bloc comme suit :

kernel_name<<<1, 32>>>(argument list);

Si chaque bloc n'a qu'un seul élément, on obtient 32 blocs comme suit :

kernel_name<<<32, 1>>>(argument list);

Comportements asynchrones

- Unlike a C function call, all CUDA kernel launches are asynchronous.
- Control returns to the CPU immediately after the CUDA kernel is invoked.

Résumé des qualificatifs de type de fonction utilisés dans la programmation CUDA C

TABLE 2-2: Function Type Qualifiers

QUALIFIERS	EXECUTION	CALLABLE	NOTES
`__global__`	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a `void` return type
`__device__`	Executed on the device	Callable from the device only	
`__host__`	Executed on the host	Callable from the host only	Can be omitted

Lecture2_Example_6: Kernel Function

```cpp
__global__ void addKernel(int* c, const int* a, const int* b) {
    int index = threadIdx.x;
    c[index] = a[index] + b[index];
}

int main() {
    const int arraySize = 5;
    int a[arraySize] = { 1, 2, 3, 4, 5 };
    int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    int* dev_a, * dev_b, * dev_c;

    // Allocate GPU memory
    cudaMalloc(devPtr: (void**)&dev_a, arraySize * sizeof(int));
    cudaMalloc(devPtr: (void**)&dev_b, arraySize * sizeof(int));
    cudaMalloc(devPtr: (void**)&dev_c, arraySize * sizeof(int));

    // Copy data to GPU
    cudaMemcpy(dst: dev_a, src: a, count: arraySize * sizeof(int), kind: cudaMemcpyHostToDevice);
    cudaMemcpy(dst: dev_b, src: b, count: arraySize * sizeof(int), kind: cudaMemcpyHostToDevice);

    // Launch kernel with 1 block and 5 threads
    addKernel << <1, arraySize >> > (dev_c, dev_a, dev_b);

    // Copy result back to host
    cudaMemcpy(dst: c, src: dev_c, count: arraySize * sizeof(int), kind: cudaMemcpyDeviceToHost);

    printf(_Format: "Result: ");
    for (int i = 0; i < arraySize; i++) {
        printf(_Format: "%d ", c[i]);
    }
```

Lecture2_Example_7: __device__ Function

```cpp
// Device function, callable from the GPU only
__device__ int multiplyByTwo(int x) {
    return x * 2;
}

// Kernel function, callable from the host
__global__ void multiplyArray(int* arr) {
    int index = threadIdx.x;
    arr[index] = multiplyByTwo(arr[index]);
}

int main() {
    const int arraySize = 5;
    int arr[arraySize] = { 1, 2, 3, 4, 5 };
    int* dev_arr;

    // Allocate memory on the GPU
    cudaMalloc(devPtr: (void**)&dev_arr, arraySize * sizeof(int));

    // Copy data to GPU
    cudaMemcpy(dst: dev_arr, src: arr, count: arraySize * sizeof(int), kind: cudaMemcpyHostToDevice);

    // Launch kernel with 1 block and 5 threads
    multiplyArray << <1, arraySize >> > (dev_arr);

    // Copy result back to host
    cudaMemcpy(dst: arr, src: dev_arr, count: arraySize * sizeof(int), kind: cudaMemcpyDeviceToHost);

    printf(_Format: "Result: ");
    for (int i = 0; i < arraySize; i++) {
        printf(_Format: "%d ", arr[i]);
    }

    // Free GPU memory
    cudaFree(devPtr: dev_arr);

    return 0;
}
```

Lecture2_example_5: Capacité de calcul

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>


int main() {
    cudaDeviceProp prop;
    int device;

    cudaGetDevice(&device);
    cudaGetDeviceProperties(&prop, device);

    printf(_Format: "Compute capability: %d.%d\n", prop.major, prop.minor);

    return 0;
}
```

Lecture2_Example_8: Appelable à partir de l'hôte et du dispositif

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

// Function callable from both the host and device
__host__ __device__ int square(int x) {
    return x * x;
}

// Kernel function to calculate squares on the GPU
__global__ void squareArray(int* arr) {
    int index = threadIdx.x;
    arr[index] = square(arr[index]);
}
```

```cpp
int main() {
    const int arraySize = 5;
    int arr[arraySize] = { 1, 2, 3, 4, 5 };
    int* dev_arr;

    // Allocate memory on the GPU
    cudaMalloc(devPtr: (void**)&dev_arr, arraySize * sizeof(int));

    // Copy data to GPU
    cudaMemcpy(dst: dev_arr, src: arr, count: arraySize * sizeof(int), kind: cudaMemcpyHostToDevice);

    // Launch kernel with 1 block and 5 threads
    squareArray << <1, arraySize >> > (dev_arr);

    // Copy result back to host
    cudaMemcpy(dst: arr, src: dev_arr, count: arraySize * sizeof(int), kind: cudaMemcpyDeviceToHost);

    printf(_Format: "GPU Result: ");
    for (int i = 0; i < arraySize; i++) {
        printf(_Format: "%d ", arr[i]);
    }
    printf(_Format: "\n");

    // Call square function on host
    printf(_Format: "Host Result: ");
    for (int i = 0; i < arraySize; i++) {
        printf(_Format: "%d ", square(x: arr[i]));
    }

    // Free GPU memory
    cudaFree(devPtr: dev_arr);

    return 0;
}
```

Les noyaux CUDA sont des fonctions avec des restrictions

Les restrictions suivantes s'appliquent à tous les noyaux :

- Accès à la mémoire du périphérique uniquement
- Doit avoir un type de retour *void*
- Pas de prise en charge d'un nombre variable d'arguments
- Pas de prise en charge des variables statiques
- Pas de prise en charge des pointeurs de fonction
- Comportement asynchrone

Pointeur de fonction

Demo: Lecture2_Function_pointer

Exemple simple d'addition de deux vecteurs A et B de taille N

Le code C pour l'addition de vecteurs sur l'hôte est donné ci-dessous :

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
        for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

Il s'agit d'un code séquentiel qui itère N fois.

En enlevant la boucle, on obtient la fonction noyau suivante :

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
```

Supposons un vecteur de 32 éléments, vous pouvez invoquer le noyau avec 32 threads de la manière suivante :

```
sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C)
```

Lecture2_Example_9: sum Arrays On GPU small case

Organisation des threads parallèles

Vous allez en apprendre davantage sur l'heuristique des grilles et des blocs en utilisant les schémas suivants pour l'addition de matrices :

- Grille 2D avec blocs 2D
- Grille 1D avec blocs 1D
- Grille 2D avec blocs 1D

Indexation de matrices avec des blocs et des threads

- En règle générale, une matrice est stockée linéairement dans la mémoire globale avec une approche majeure de ligne.
- La figure 2-9 illustre un petit cas pour une matrice 8 x 6.

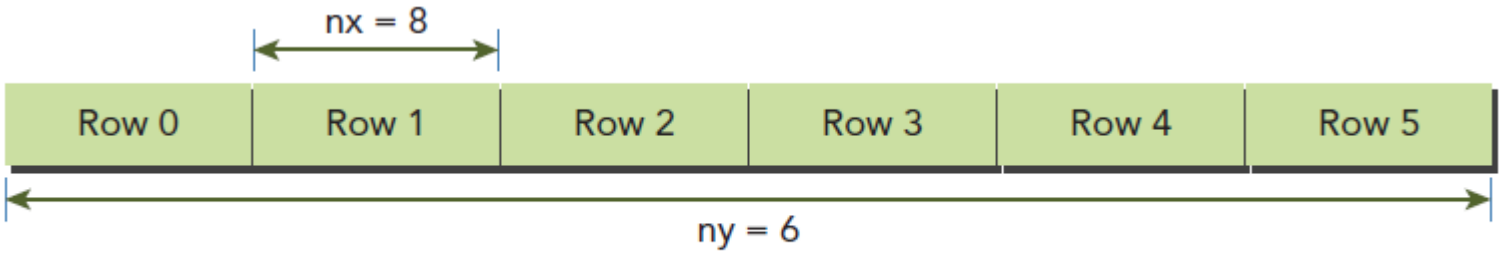


FIGURE 2-9

Indexation de matrices avec des blocs et des threads

- Dans un noyau d'addition de matrice, un thread se voit généralement attribuer un élément de données à traiter.
- Le premier problème à résoudre est l'accès aux données assignées à partir de la mémoire globale à l'aide de l'index des blocs et des threads.
- En règle générale, il existe trois types d'index pour un cas 2D que vous devez gérer :
  - Index de thread et de bloc
  - Coordonnées d'un point donné de la matrice
  - Offset dans la mémoire globale linéaire

Indexation de matrices avec des blocs et des threads

- Dans un premier temps, vous pouvez faire correspondre l'index des threads et des blocs aux coordonnées d'une matrice à l'aide de la formule suivante :
  - `ix = threadIdx.x + blockIdx.x * blockDim.x`
  - `iy = threadIdx.y + blockIdx.y * blockDim.y`

- Dans un deuxième temps, vous pouvez faire correspondre une coordonnée matricielle à un emplacement/index de la mémoire globale avec la formule suivante :
  - `idx = iy * nx + ix`

Block and thread indices, matrix coordinates, and linear global memory indices

Figure 2-10 illustrates the corresponding relationship among block and thread indices, matrix coordinates, and linear global memory indices.

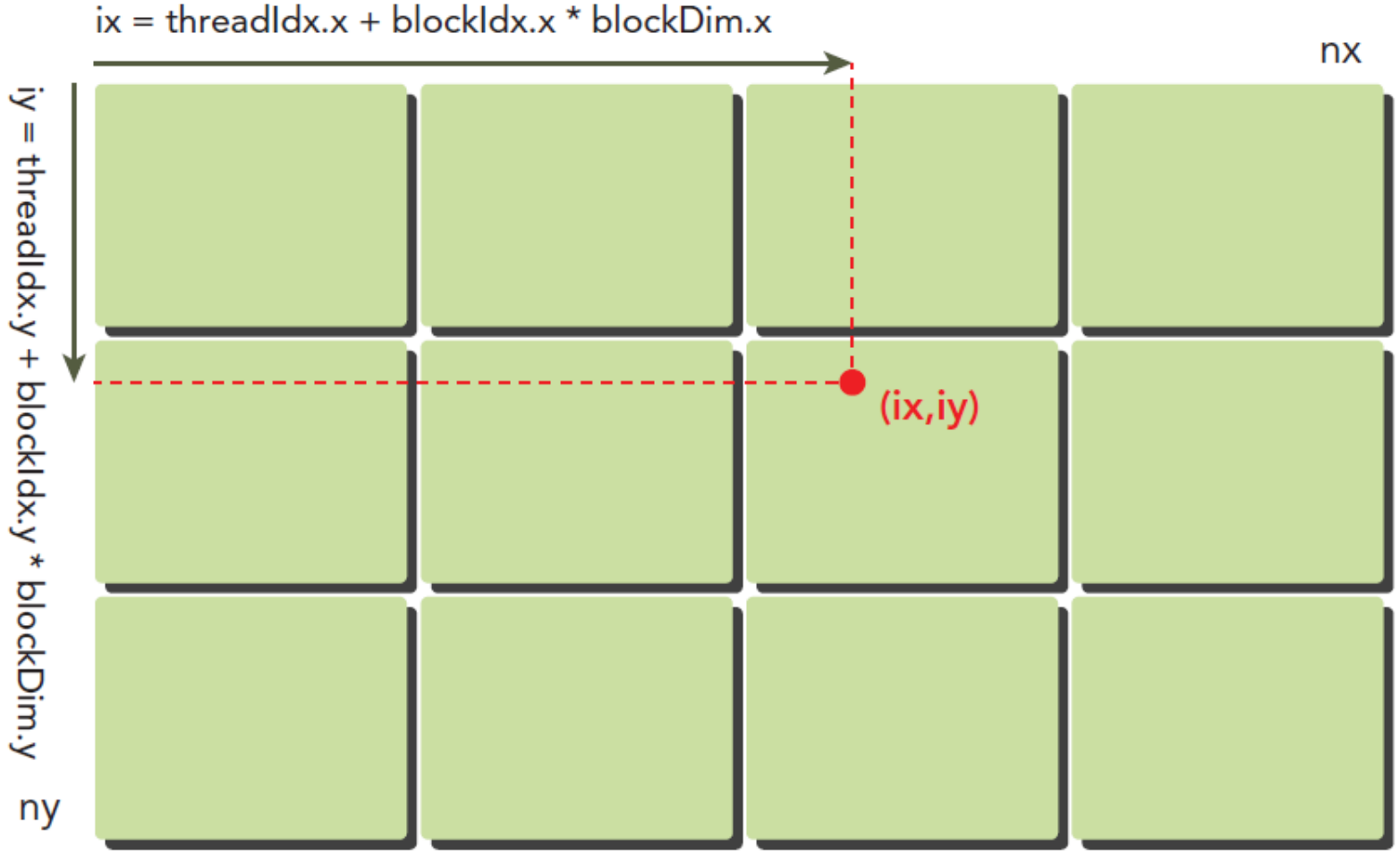


FIGURE 2-10

printThreadInfo

- Index des fils
- Index du bloc
- Coordonnées de la matrice
- Décalage de la mémoire linéaire globale
- Valeur des éléments correspondants

La figure 2-11 illustre la relation entre ces trois indices

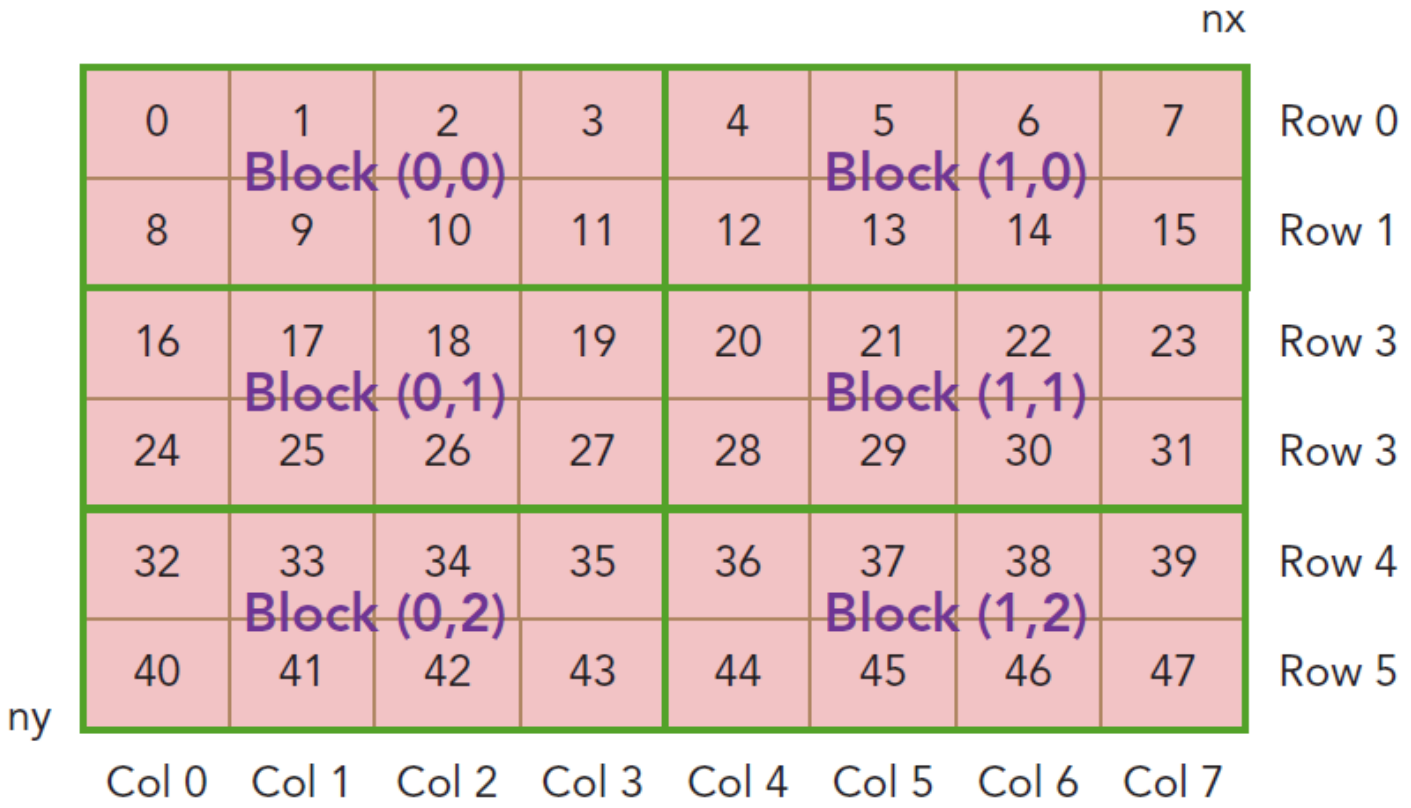


FIGURE 2-11

- block(4, 2): 4 threads dans la dimension X, 2 threads dans la dimension Y
- grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y)
- grid((8 + 4 - 1) / 4, (6 + 2 - 1) / 2)
- Grid(2, 3): 2 blocs dans la dimension X, 3 blocs dans la dimension Y
- printThreadIndex <<<grid, block>>>
- printThreadIndex <<<(2, 3), (4, 2)>>>

Demo Lecture2_0002: checkThreadIndex

Sommation de matrices avec une grille 2D et des blocs 2D

La clé de ce noyau est l'étape qui consiste à faire correspondre à chaque thread son index de thread à l'index de la mémoire linéaire globale, comme illustré dans la figure 2-12 :

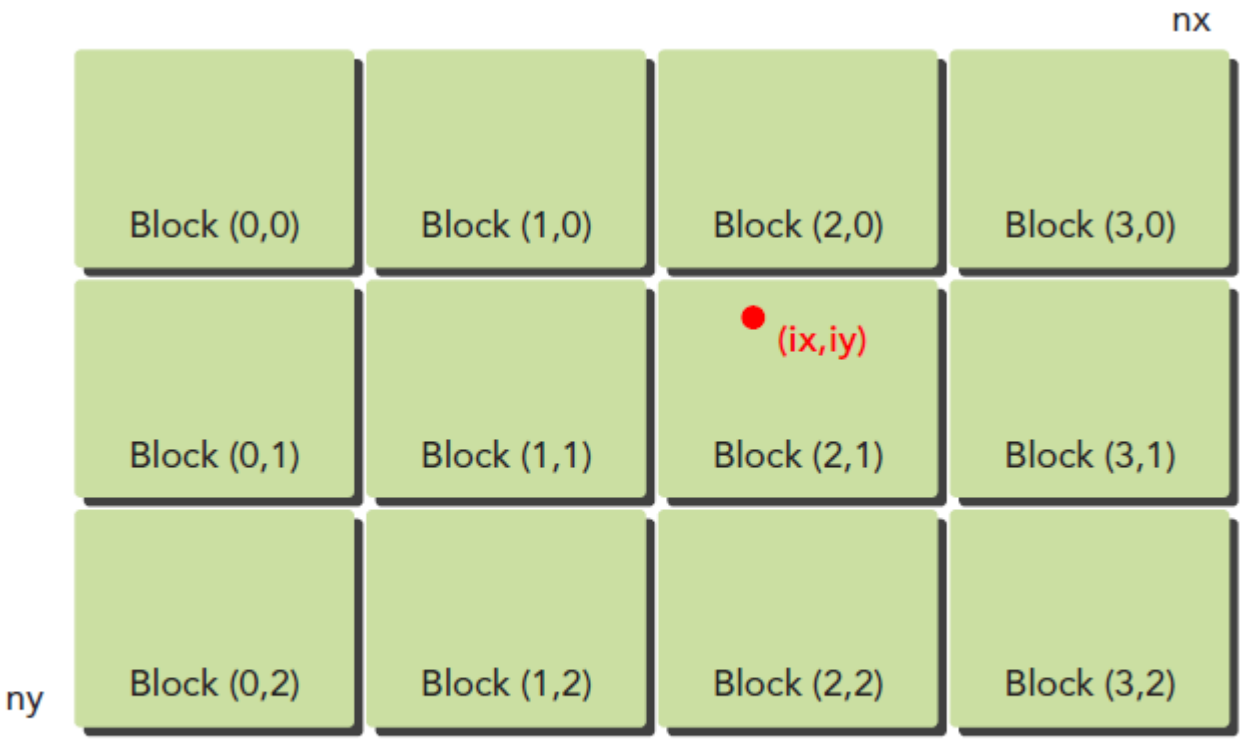


FIGURE 2-12

Configuration de l'exécution du noyau

La taille de la matrice peut être fixée à 16 384 éléments dans chaque dimension, comme suit :

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

La configuration de l'exécution du noyau peut être réglée pour utiliser une grille 2D et des blocs 2D comme suit :

```
int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);
```

- dim3 block(32, 32): 32 threads in X dimension, 32 threads in Y dimension
- dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y)
- dim3 grid((16 384 + 32 - 1) / 32, (16 384 + 32 - 1) / 32)
- dim3 grid(512, 512): 512 blocks in X dimension, 512 blocks in Y dimension
- sumMatrixOnGPU2D <<< grid, block >>>
- sumMatrixOnGPU2D <<<(512, 512), (32, 32)>>>

Demo Lecture2_Examples_12: sumMatrixOnGPU-2D-grid-2D-block

Résultats Comparaison des différentes implémentations du noyau

Kernel execution	Execution configure	Time elapsed
sumMatrixOnGPU2D	(512, 1024), (32, 16)	0.094000 sec
sumMatrixOnGPU2D	(128,16384), (128,1)	0.095000 sec
sumMatrixOnGPU2D	(64, 16384), (256, 1)	0.094000 sec

Demo Lecture2_Examples_12: sumMatrixOnGPU-2D-grid-2D-block

Utilisation de l'API d'exécution pour demander des informations sur le GPU

Demo Lecture2_0000: checkDeviceInfor

```
Device 0: "NVIDIA GeForce GTX 1050 Ti"
  CUDA Driver Version / Runtime Version          12.6 / 12.6
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 4.00 GBytes (4294836224 bytes)
  GPU Clock rate:                                1620 MHz (1.62 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Max Texture Dimension Size (x,y,z)             1D=(131072), 2D=(131072,65536), 3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers        1D=(32768) x 2048, 2D=(32768,32768) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
```

Résumé

- Par rapport à la programmation parallèle en C, la hiérarchie des threads dans la programmation CUDA est une caractéristique distinctive.
- En exposant une hiérarchie abstraite de threads à deux niveaux, CUDA vous permet de contrôler un environnement massivement parallèle.
- Les exemples de ce chapitre vous ont également montré que les dimensions de la grille et du bloc ont un impact important sur les performances du noyau.
- Pour un problème donné, vous disposez de plusieurs options pour implémenter le noyau et de différentes configurations pour l'exécuter.
- En général, l'implémentation naïve ne donne pas les meilleures performances.
- Par conséquent, l'apprentissage de l'organisation des threads est l'une des pratiques centrales de la programmation CUDA.
- La meilleure façon de comprendre l'heuristique des grilles et des blocs est d'écrire des programmes qui étendent vos compétences et vos connaissances par le biais d'essais et d'erreurs.
- Les grilles et les blocs représentent une vue logique de la disposition des threads pour l'exécution du noyau.
- Dans le chapitre 3, vous étudierez les mêmes questions d'un point de vue différent : la vue matérielle.