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

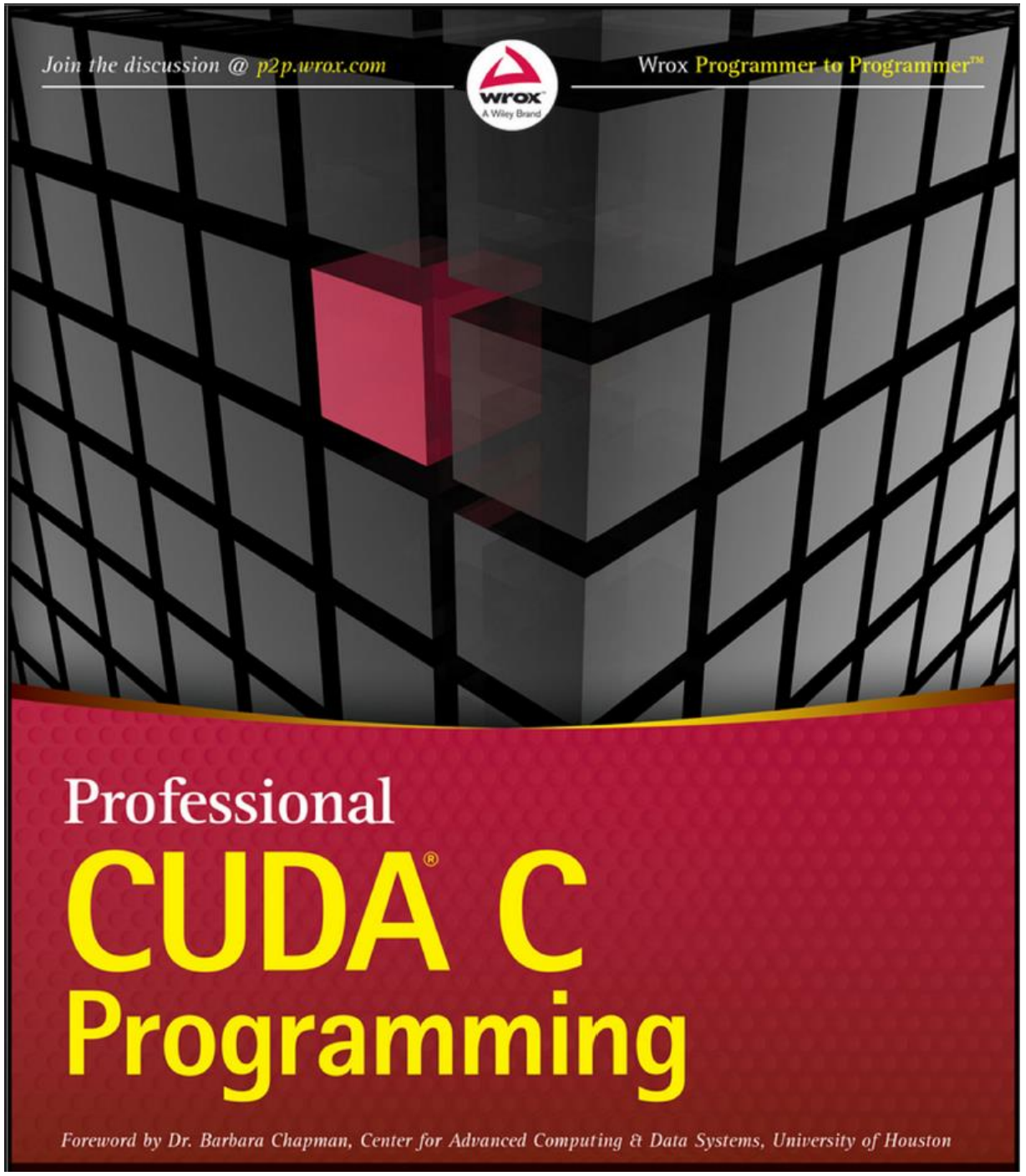
School of Electrical Engineering
and Computer Science

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 1 : Calcul parallèle hétérogène avec CUDA

Plan

- Introduction : Importance de l'informatique parallèle hétérogène.
- Calcul parallèle : Programmation séquentielle et programmation parallèle.
- Architecture informatique : Taxonomie de Flynn ; multicœur et multicœur.
- Calcul hétérogène : CPU et GPU dans les systèmes modernes.
- Architecture des GPU : Conception des GPU et mesures des performances.
- Plate-forme CUDA : CUDA en tant que plateforme de calcul parallèle.
- Hello World depuis le GPU : Exemple de programme CUDA de base.
- Résumé : Récapitulation des points clés.

Architecture ordinateur (aspect matériel)

La plupart des processeurs modernes mettent en œuvre l'architecture Harvard, comme le montre la figure 1-1, qui se compose de trois interfaces principales:

- Mémoire (mémoire d'instructions et mémoire de données)
- Unité centrale de traitement (unité de commande et unité arithmétique et logique)
- Interfaces d'entrée/sortie

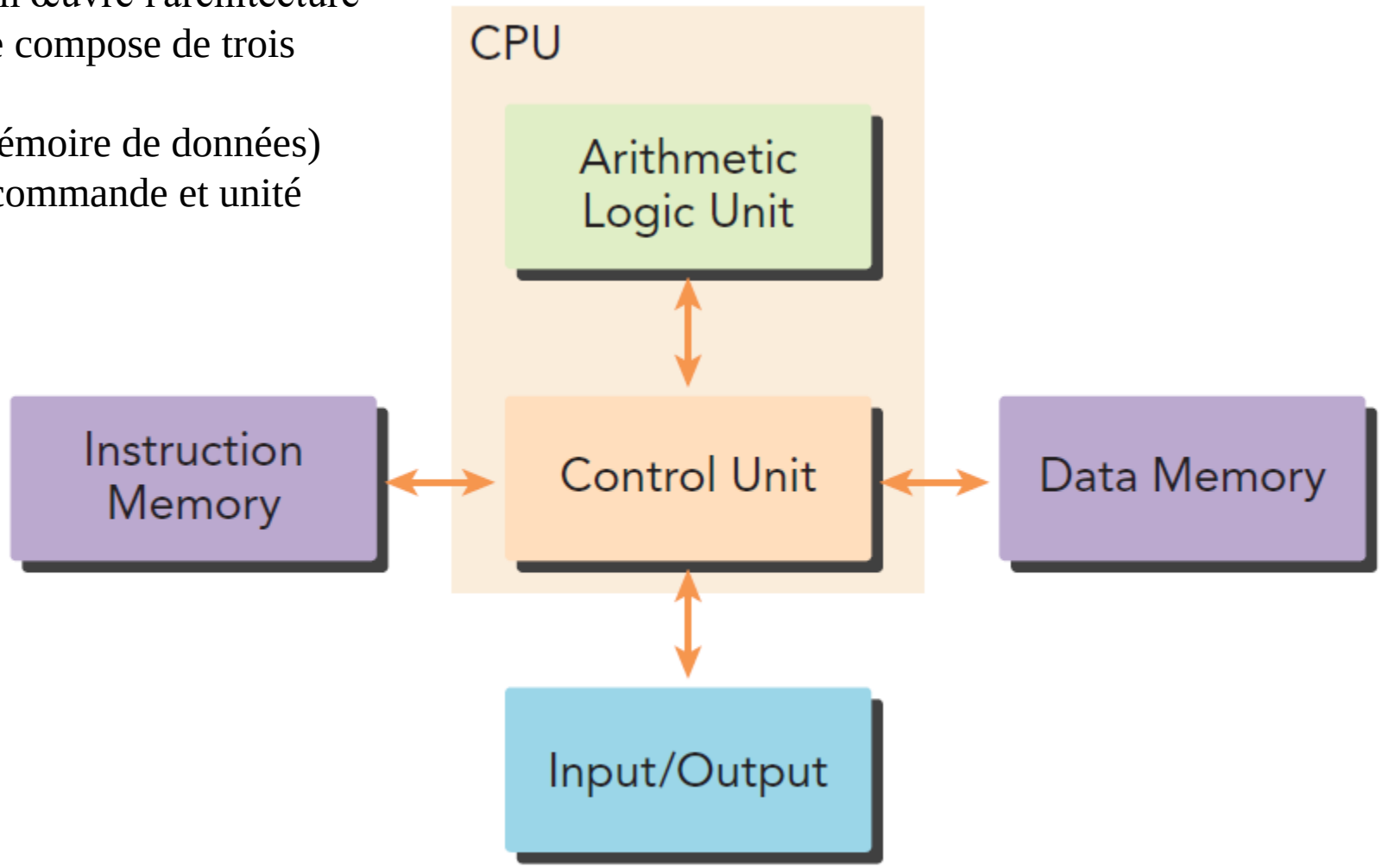
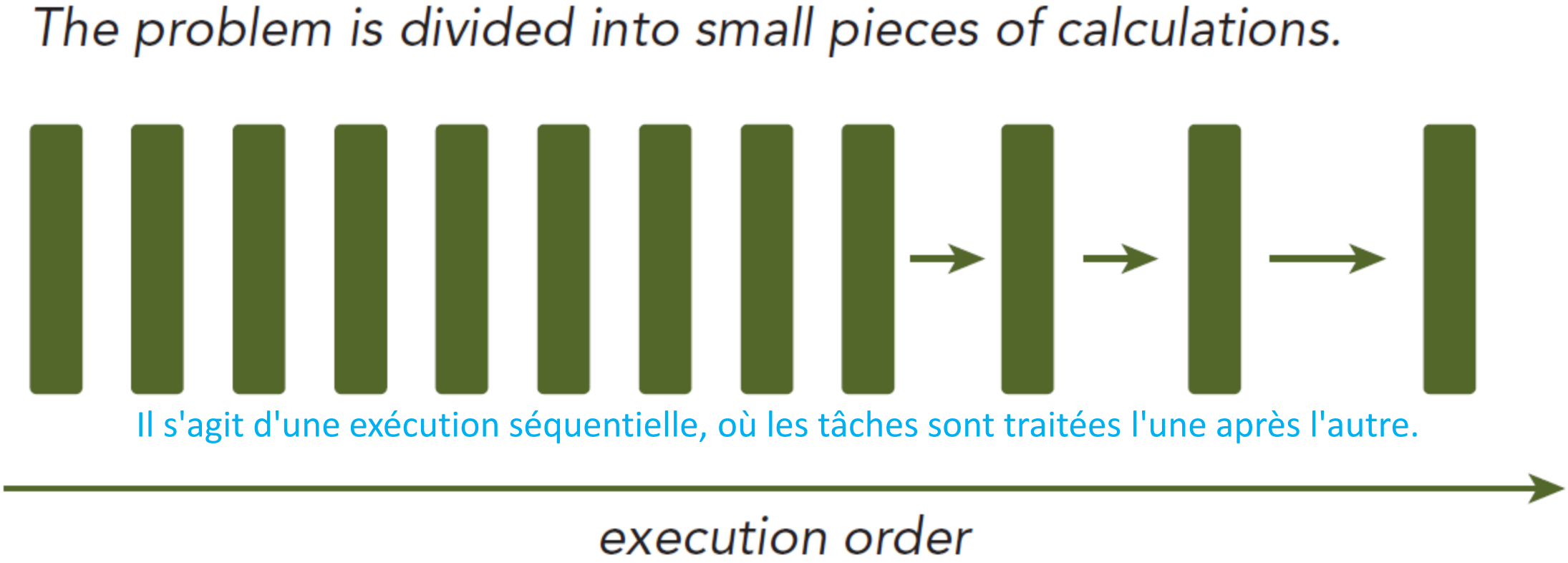


FIGURE 1-1

Programmation séquentielle et parallèle



Il s'agit d'une exécution séquentielle, où les tâches sont traitées l'une après l'autre.

FIGURE 1-2

Programmation séquentielle et parallèle

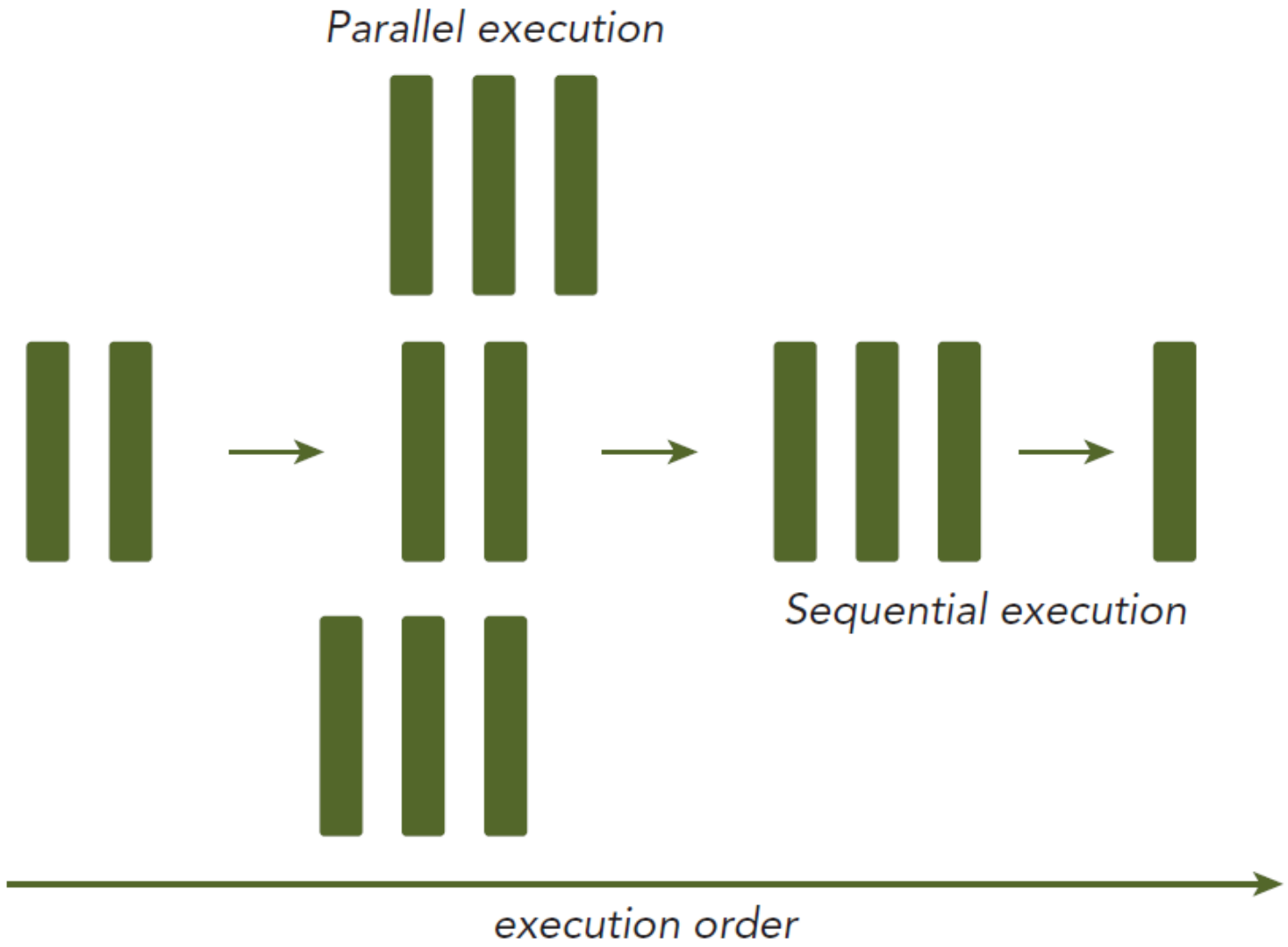


FIGURE 1-3

Parallélisme

Il existe deux types fondamentaux de parallélisme dans les applications:

- ➤ Parallélisme des taches
- ➤ Parallélisme des données

- Parallélisme des taches se produit lorsqu'il existe de nombreuses tâches ou fonctions qui peuvent être exécutées indépendamment et en grande partie en parallèle.
- Parallélisme des tâches se concentre sur la distribution des fonctions sur plusieurs cœurs.

- Parallélisme des données se produit lorsqu'il y a de nombreux éléments de données qui peuvent être traités en même temps.
- Parallélisme des données se concentre sur la distribution des données sur plusieurs cœurs.

Figure 1-4

- La figure 1-4 présente deux exemples simples de partitionnement de données 1D.
- Dans la partition par blocs, chaque thread ne traite qu'une partie des données, et dans la partition cyclique, chaque thread traite plus d'une partie des données..

Block partition: each thread takes one data block

Cyclic partition: each thread takes two data blocks

FIGURE 1-4

Figure 1-5

La figure 1-5 présente trois exemples simples de partitionnement des données en 2D :

- partitionnement en bloc le long de la dimension y, partitionnement en bloc sur les deux dimensions et partitionnement cyclique le long de la dimension x.
- Les modèles restants - partitionnement par blocs le long de la dimension x, partitionnement cyclique sur les deux dimensions et partitionnement cyclique le long de la dimension y - sont laissés à l'état d'exercice.

Block partition on one dimension

Block partition on both dimensions

Cyclic partition on one dimension

FIGURE 1-5

Architecture de l'ordinateur

Un système de classification largement utilisé est la taxonomie de Flynn, qui classe les architectures en quatre types différents en fonction de la manière dont les instructions et les données circulent à travers les cœurs (voir figure 1-6) :

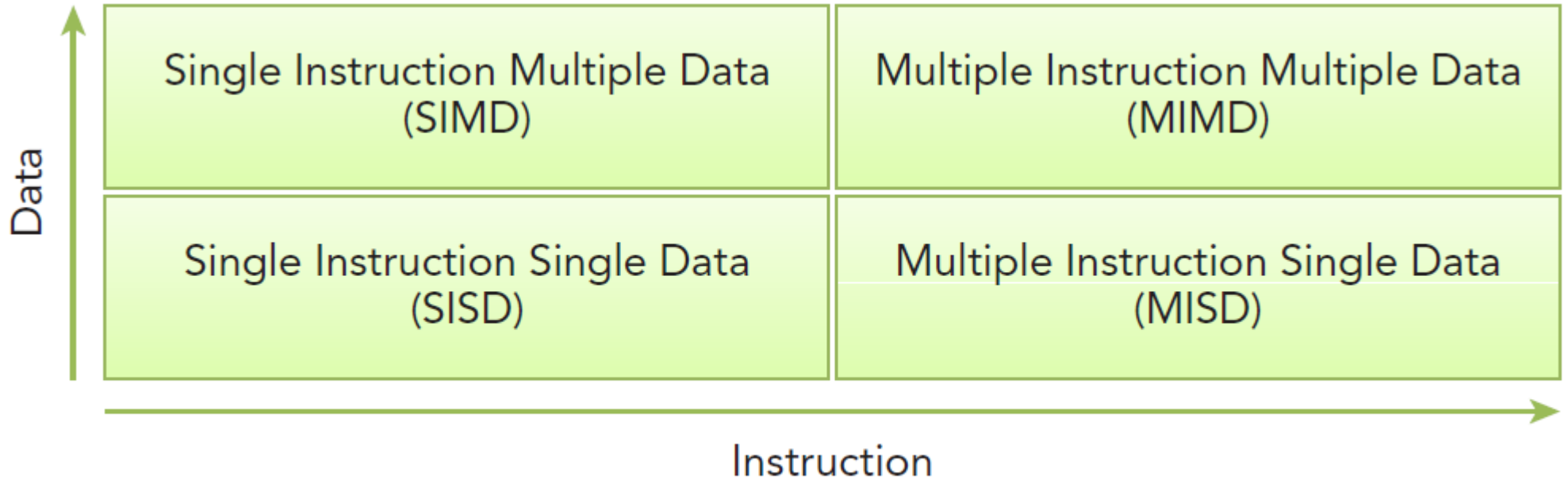


FIGURE 1-6

Taxonomie de Flynn

- L'expression « Single Instruction Single Data » fait référence à l'ordinateur traditionnel : une architecture en série.
- Il n'y a qu'un seul cœur dans l'ordinateur.
- À tout moment, un seul flux d'instructions est exécuté et les opérations sont effectuées sur un seul flux de données.

- L'expression Single Instruction Multiple Data (instruction unique, données multiples) désigne un type d'architecture parallèle.
- L'ordinateur comporte plusieurs cœurs.
- Tous les cœurs exécutent le même flux d'instructions à tout moment, chacun opérant sur des flux de données différents.

- L'expression « Multiple Instruction Single Data » fait référence à une architecture peu courante, dans laquelle chaque cœur fonctionne sur le même flux de données via des flux d'instructions distincts.

- L'expression «*Multiple Instruction Multiple Data*» désigne un type d'architecture parallèle dans laquelle plusieurs cœurs fonctionnent sur des flux de données multiples, chacun exécutant des instructions indépendantes.

- De nombreuses architectures MIMD comprennent également des sous-composants d'exécution SIMD.

- Au niveau de l'architecture, de nombreuses avancées ont été réalisées pour atteindre les objectifs suivants :
  - ➤ Diminuer la latence
  - ➤ Augmenter la bande passante
  - ➤ Augmenter le débit

Latence, bande passante et débit en informatique

La latence est le temps nécessaire au démarrage et à l'achèvement d'une opération et est généralement exprimé en microsecondes.

La bande passante est la quantité de données pouvant être traitées par unité de temps, communément exprimée en mégaoctets/seconde ou en gigaoctets/seconde.

Le débit est la quantité d'opérations pouvant être traitées par unité de temps, communément exprimée en gflops (milliards d'opérations en virgule flottante par seconde), en particulier dans les domaines du calcul scientifique qui font un usage intensif des calculs en virgule flottante. La latence mesure le temps nécessaire pour effectuer une opération, tandis que le débit mesure le nombre d'opérations traitées dans une unité de temps donnée.

Organisation de la mémoire

Les architectures des ordinateurs peuvent également être subdivisées en fonction de l'organisation de la mémoire, qui est généralement classée selon les deux types suivants :

- ➤ Multi-nœuds avec mémoire distribuée
- ➤ Multiprocesseur avec mémoire partagée

Multi-nœuds avec mémoire distribuée

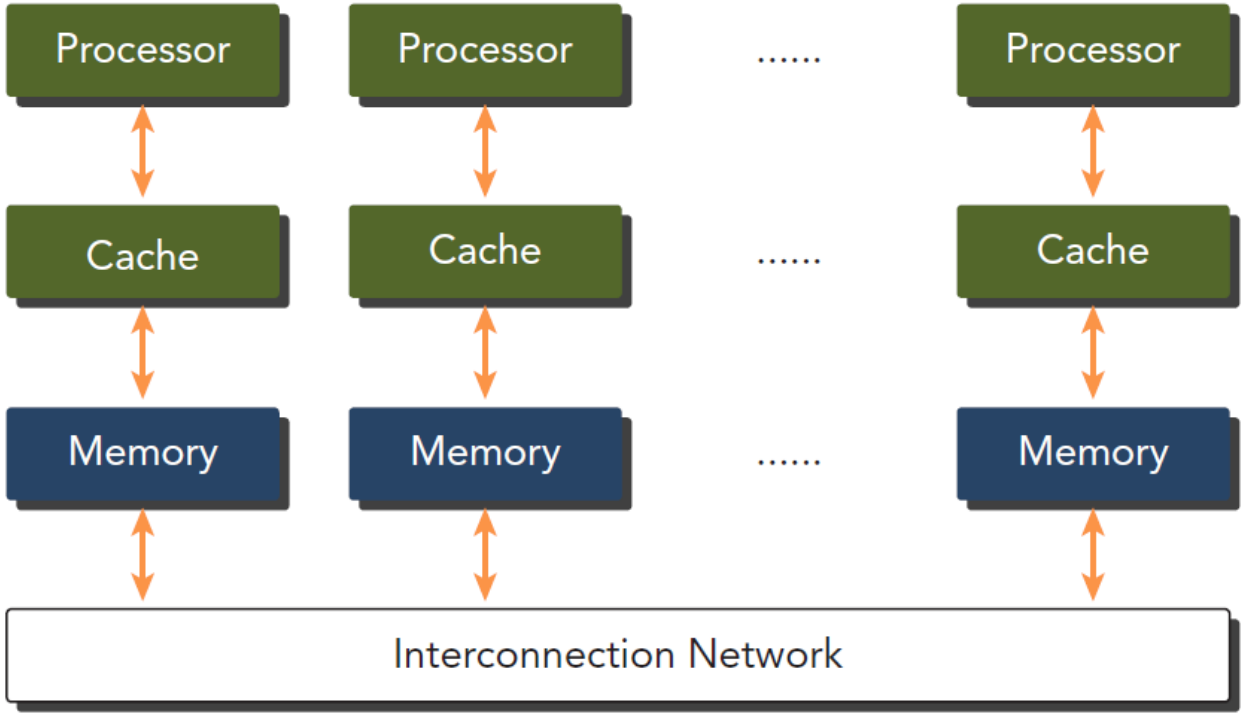


FIGURE 1-7

Multiprocesseur avec mémoire partagée

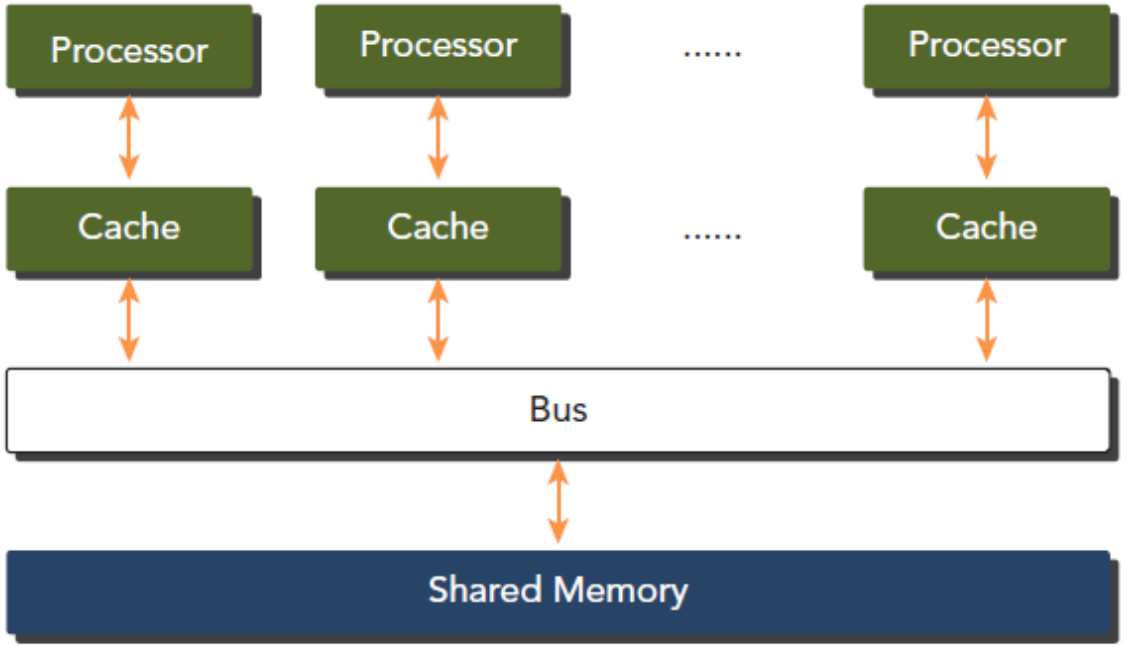


FIGURE 1-8

- Les GPU représentent une architecture à plusieurs cœurs et disposent de pratiquement tous les types de parallélisme décrits précédemment : multithreading, MIMD, SIMD et parallélisme au niveau des instructions.
- NVIDIA a inventé l'expression Single Instruction, Multiple Thread (SIMT) pour désigner ce type d'architecture.

Cœur de GPU contre cœur de CPU

- Bien que les termes "many-core" et "multicore" soient utilisés pour décrire les architectures de GPU et de CPU, un cœur de GPU est très différent d'un cœur de CPU.

- Un cœur de CPU, relativement lourd, est conçu pour une logique de contrôle très complexe, cherchant à optimiser l'exécution de programmes séquentiels.

- Un noyau GPU, relativement léger, est optimisé pour les tâches parallèles aux données avec une logique de contrôle plus simple, en se concentrant sur le débit des programmes parallèles.

Architecture hétérogène

- Un nœud de calcul hétérogène typique se compose aujourd'hui de deux sockets de CPU multicœur et de deux ou plusieurs GPU multicœurs.

- Actuellement, un GPU n'est pas une plateforme autonome mais un coprocesseur d'un CPU.

- Par conséquent, les GPU doivent fonctionner en conjonction avec un hôte basé sur le CPU par le biais d'un bus PCI-Express, comme le montre la Figure 1-9. C'est pourquoi, en termes d'informatique GPU, le CPU est appelé l'hôte et le GPU est appelé le périphérique.

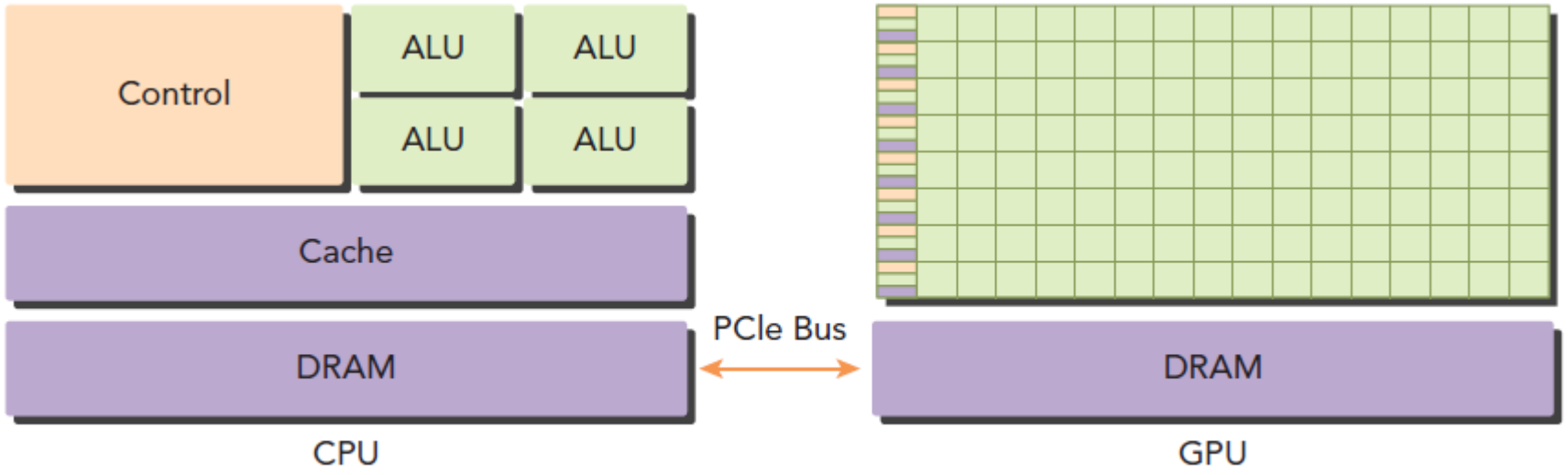


FIGURE 1-9

Architecture hétérogène

Une application hétérogène se compose de deux parties :

- ➤ Host code
- ➤ Device code

- Le code hôte s'exécute sur les CPU et le code de dispositif s'exécute sur les GPU.
- Une application exécutée sur une plateforme hétérogène est généralement initialisée par le processeur.
- Le code de l'unité centrale est responsable de la gestion de l'environnement, du code et des données du dispositif avant de charger les tâches à forte intensité de calcul sur le dispositif.

Paradigme de l'informatique hétérogène

Comme le montre la figure 1-10, deux dimensions différencient le champ d'application des CPU et des GPU :

- Niveau de parallélisme
- Taille des données

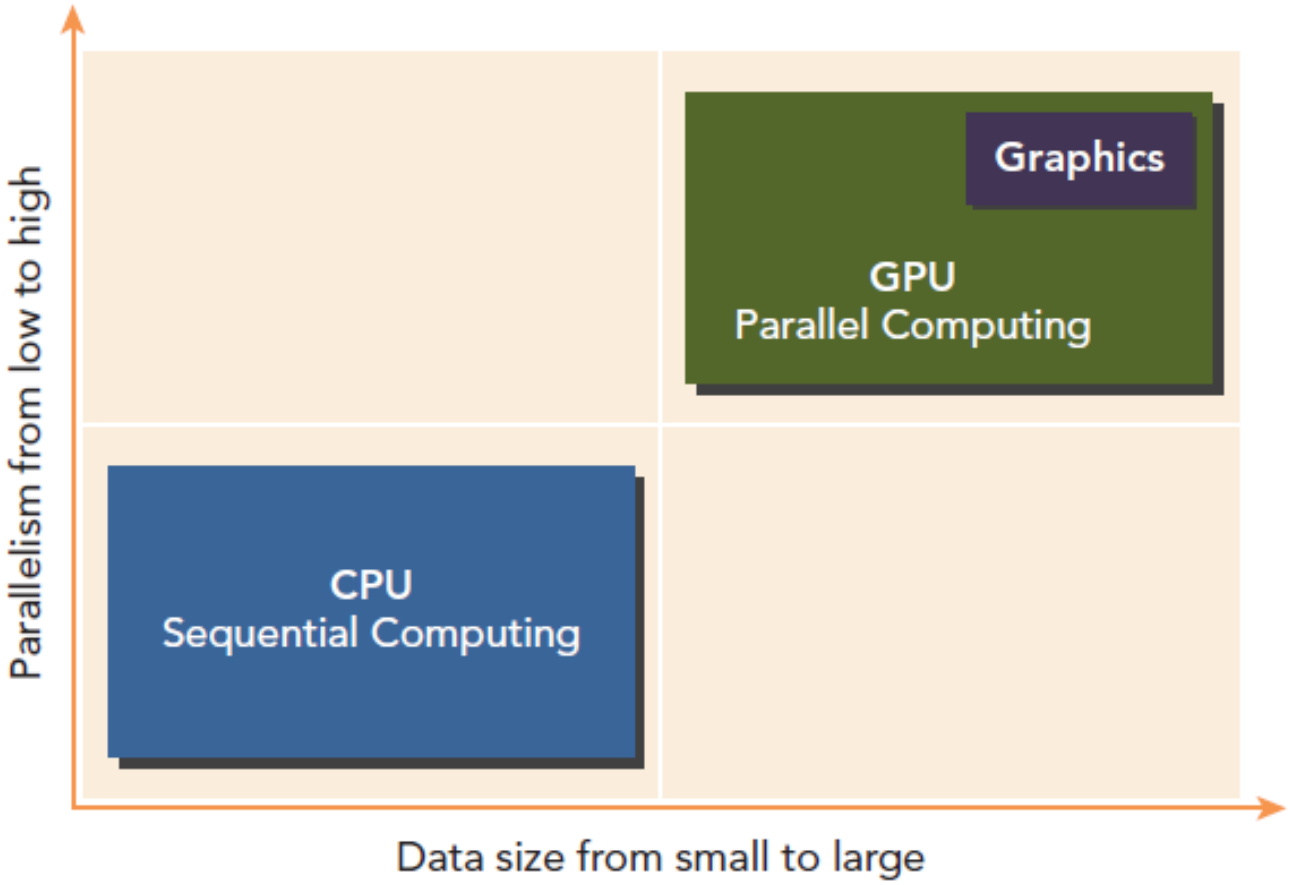


FIGURE 1-10

Paradigme de l'informatique hétérogène

- Les architectures informatiques parallèles hétérogènes (CPU + GPU) ont évolué parce que le CPU et le GPU ont des attributs complémentaires qui permettent aux applications d'être plus performantes en utilisant les deux types de processeurs.

- Par conséquent, pour obtenir des performances optimales, vous devrez peut-être utiliser à la fois le CPU et le GPU pour votre application, en exécutant les parties séquentielles ou les parties parallèles aux tâches sur le CPU et les parties parallèles aux données intensives sur le GPU, comme le montre la figure 1-11.

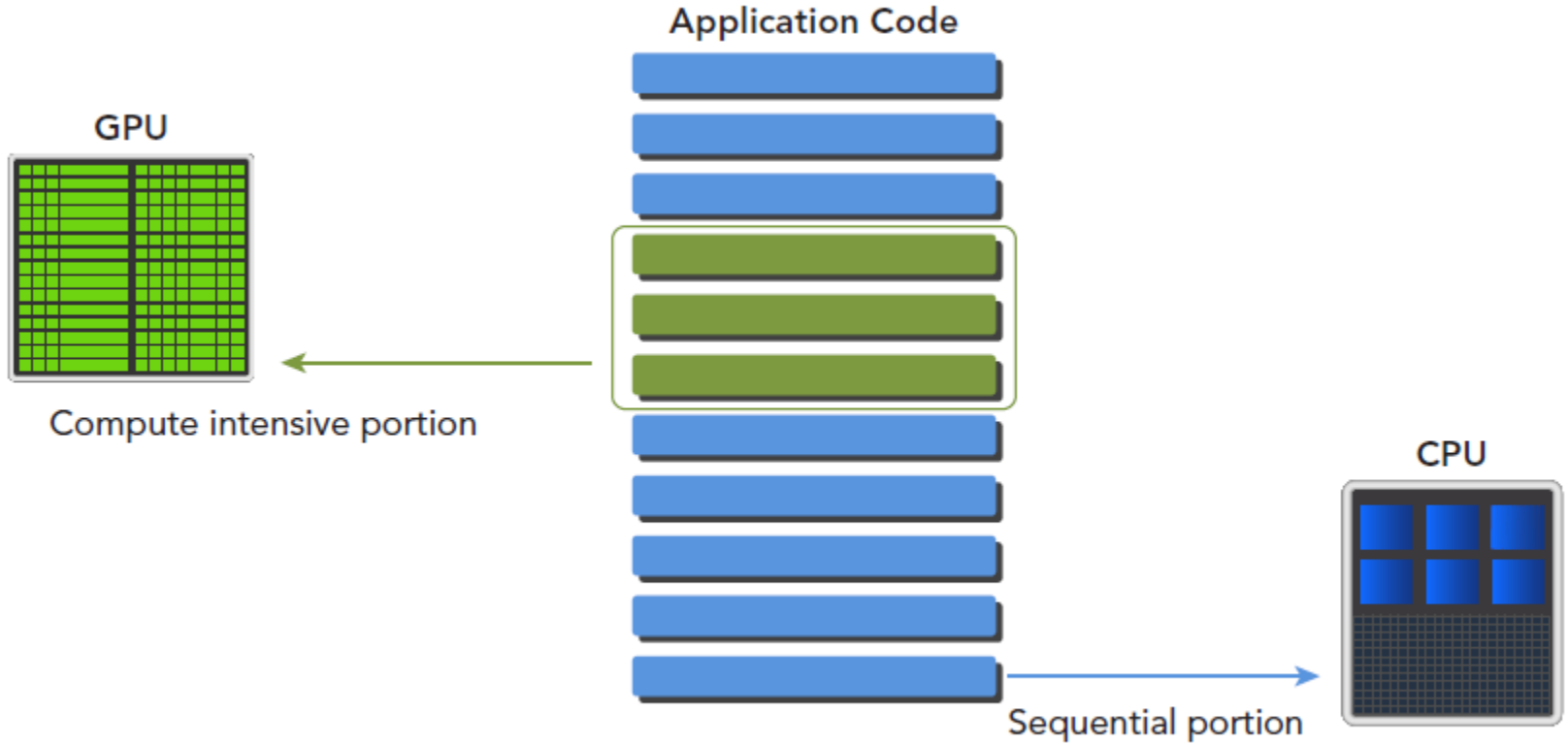


FIGURE 1-11

Thread CPU versus thread GPU

- Les threads sur une unité centrale sont généralement des entités lourdes.
- Le système d'exploitation doit permuter les threads sur les canaux d'exécution de l'unité centrale et hors de ceux-ci afin de fournir une capacité de multithreading.
- Les changements de contexte sont lents et coûteux.
- Les threads sur les GPU sont extrêmement légers.
- Dans un système typique, des milliers de threads sont en file d'attente.
- Si le GPU doit attendre sur un groupe de threads, il commence simplement à exécuter le travail sur un autre groupe.
- Les cœurs des processeurs sont conçus pour minimiser la latence pour un ou deux threads à la fois, tandis que les cœurs des GPU sont conçus pour gérer un grand nombre de threads légers et simultanés afin de maximiser le débit.
- Aujourd'hui, un CPU doté de quatre processeurs quadricœurs ne peut exécuter que 16 threads simultanément, ou 32 si les CPU prennent en charge l'hyperthreading.
- Les GPU NVIDIA modernes peuvent prendre en charge jusqu'à 1 536 threads actifs simultanément par multiprocesseur.
- Sur les GPU dotés de 16 multiprocesseurs, cela représente plus de 24 000 threads actifs simultanément.

CUDA : Une plate-forme pour le calcul hétérogène

- La plateforme CUDA est accessible via des bibliothèques accélérées par CUDA, des directives de compilation, des interfaces de programmation d'applications et des extensions pour les langages de programmation standard, notamment C, C++, Fortran et Python (comme l'illustre la figure 1-12).

- Ce livre se concentre sur la programmation CUDA C.

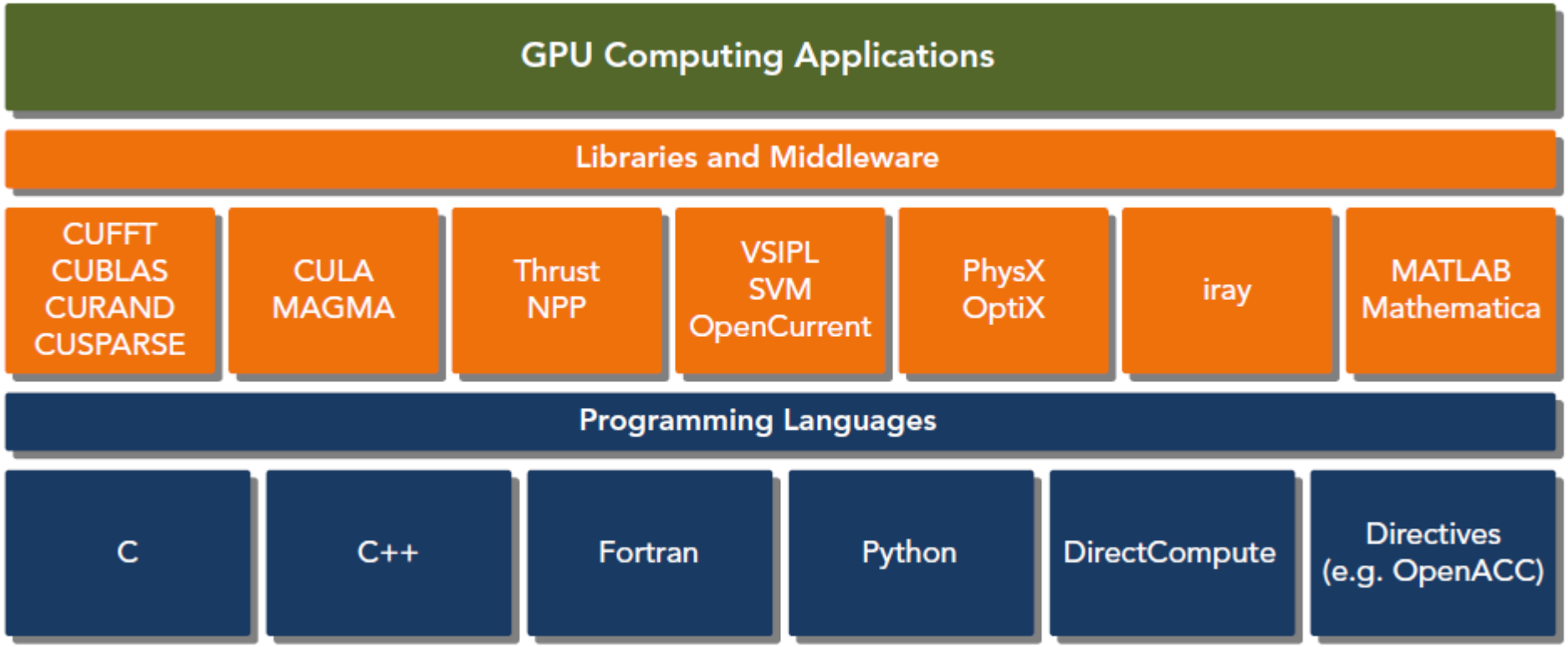


FIGURE 1-12

Architecture de la pile logicielle CUDA

CUDA fournit deux niveaux d'API pour gérer le périphérique GPU et organiser les threads, comme le montre la figure 1-13.

- ➤ API de pilote CUDA
- ➤ API d'exécution CUDA

- L'API du pilote est une API de bas niveau et est relativement difficile à programmer, mais elle permet de mieux contrôler la manière dont le périphérique GPU est utilisé.

- L'API d'exécution est une API de niveau supérieur mise en œuvre au-dessus de l'API du pilote.

- Chaque fonction de l'API d'exécution est décomposée en opérations plus élémentaires transmises à l'API du pilote.

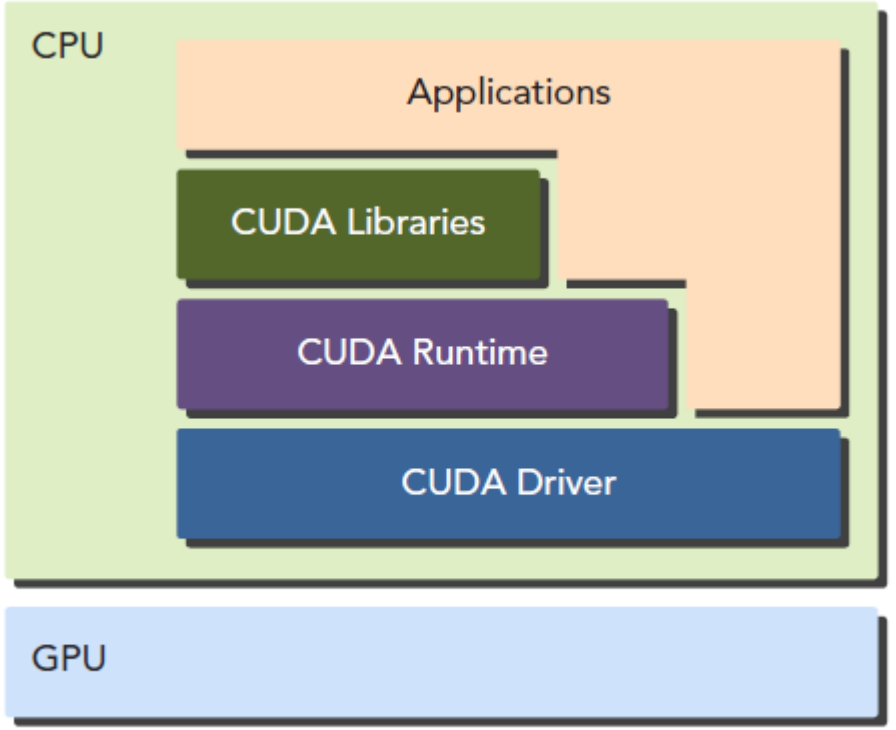


FIGURE 1-13

API d'exécution et API de pilote

- Il n'y a pas de différence de performance notable entre les API du moteur d'exécution et du pilote.
- La façon dont vos noyaux utilisent la mémoire et la façon dont vous organisez vos threads sur le périphérique ont un effet beaucoup plus prononcé.
- Ces deux API s'excluent mutuellement.
- Vous devez utiliser l'une ou l'autre, mais il n'est pas possible de mélanger les appels de fonction des deux.
- Tous les exemples présentés dans ce livre utilisent l'API runtime.

Un programme CUDA est constitué d'un mélange des deux parties suivantes :

- ➤ Le code hôte s'exécute sur l'unité centrale.
- ➤ Le code du dispositif s'exécute sur le GPU.

CUDA de NVIDIA

- Le compilateur CUDA nvcc de NVIDIA sépare le code du dispositif du code hôte au cours du processus de compilation.
- Comme le montre la figure 1-14, le code hôte est un code C standard qui est ensuite compilé avec des compilateurs C.
- Le code du dispositif est écrit en CUDA C étendu avec des mots-clés pour étiqueter les fonctions parallèles de données, appelées noyaux.
- Le code du dispositif est écrit à l'aide de CUDA C étendu avec des mots-clés pour étiqueter les fonctions de données parallèles, appelées noyaux.
- Le code du dispositif est ensuite compilé par nvcc.
- Au cours de la phase de liaison, des bibliothèques d'exécution CUDA sont ajoutées pour les appels de procédure du noyau et la manipulation explicite des dispositifs GPU.

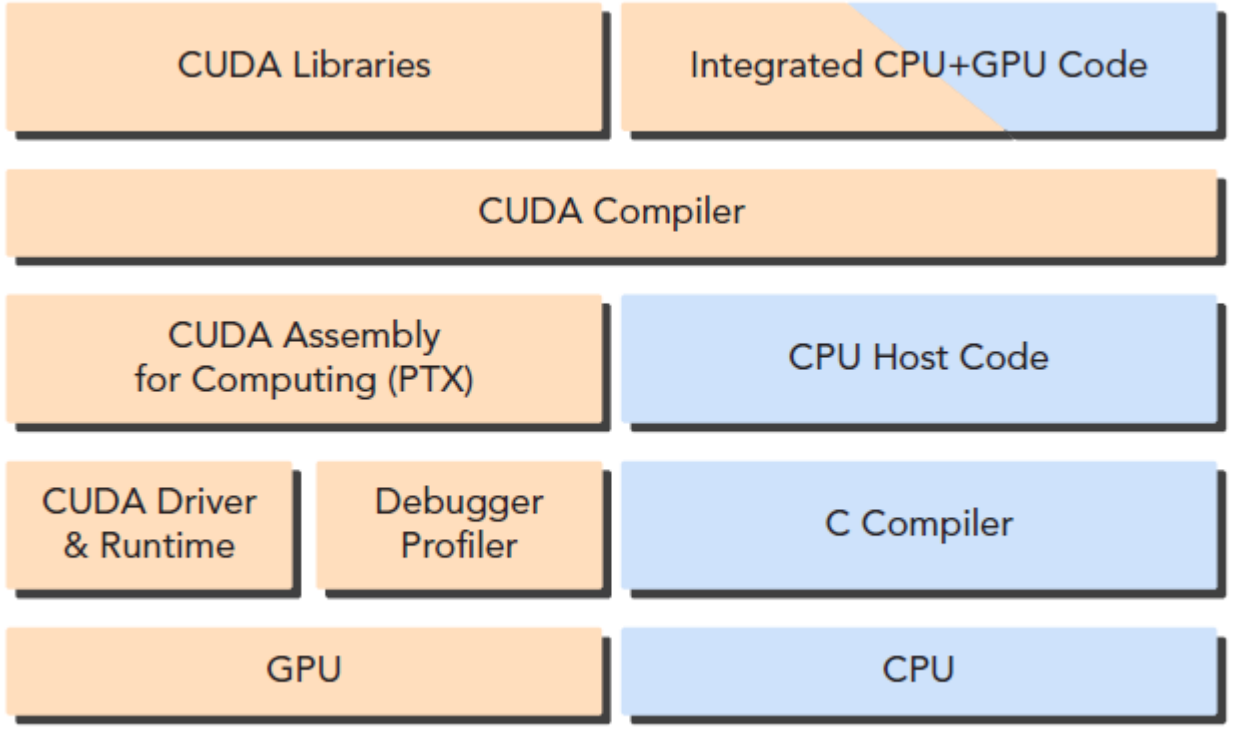


FIGURE 1-14

Compilateur CUDA nvcc

- Le compilateur CUDA nvcc est basé sur l'infrastructure de compilateur open-source LLVM largement utilisée.

- Vous pouvez créer ou étendre des langages de programmation prenant en charge l'accélération GPU à l'aide du SDK du compilateur CUDA, comme le montre la Figure 1-15.

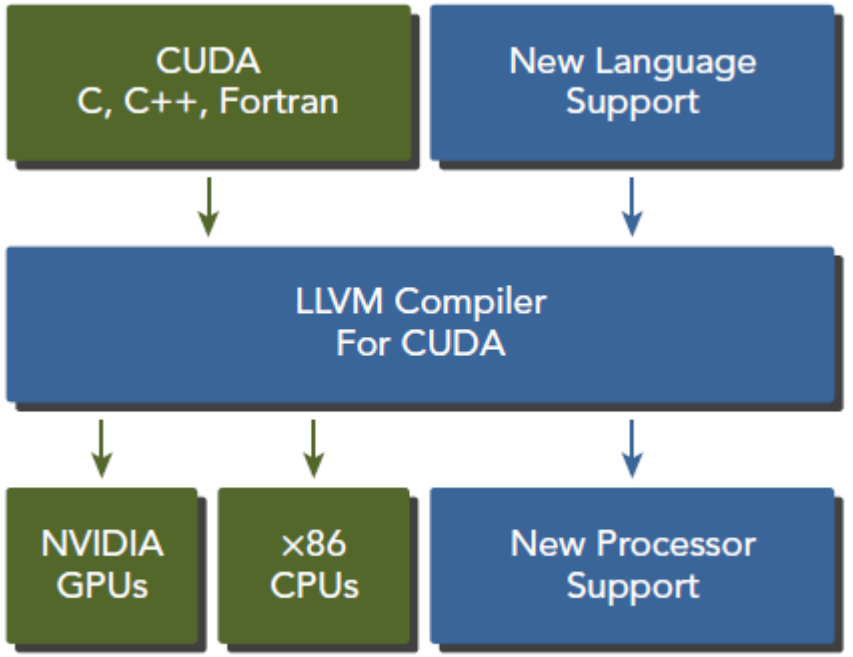


FIGURE 1-15

Plateforme CUDA

- La plateforme CUDA est également une base qui soutient un écosystème de calcul parallèle diversifié, comme le montre la Figure 1-16.

- Aujourd'hui, l'écosystème CUDA se développe rapidement car de plus en plus d'entreprises fournissent des outils, des services et des solutions de classe mondiale.

- Si vous souhaitez développer vos applications sur des GPU, le moyen le plus simple d'exploiter les performances des GPU est d'utiliser le kit d'outils CUDA, qui fournit un environnement de développement complet pour les développeurs C et C++.

- Le kit d'outils CUDA comprend un compilateur, des bibliothèques mathématiques et des outils de débogage et d'optimisation des performances de vos applications.

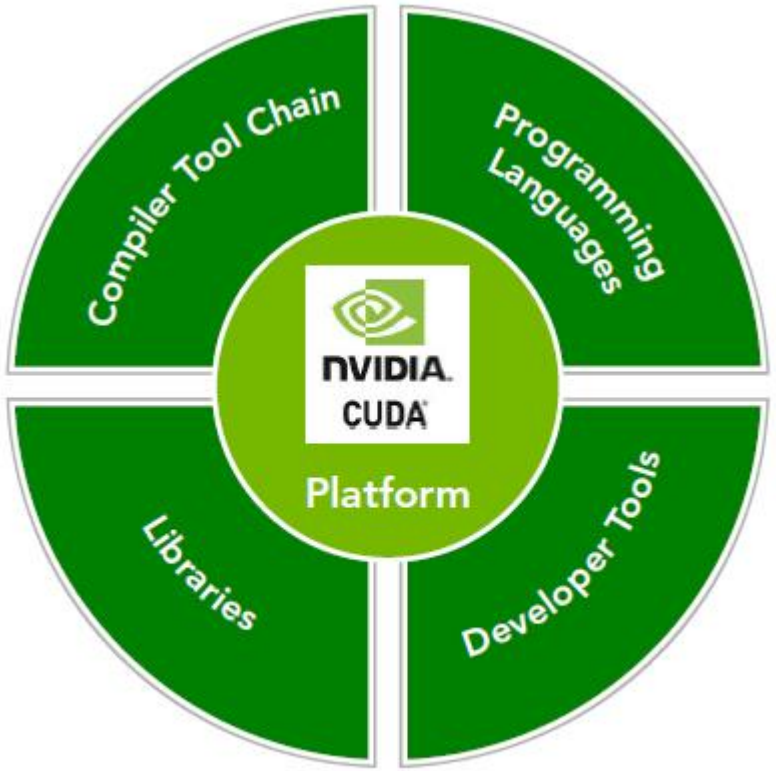


FIGURE 1-16

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

#include <stdio.h>
// Kernel function to be executed on the GPU
__global__ void helloFromGPU() {
    printf("Hello World from GPU!\n");
}

int main() {
    helloFromGPU << <1, 10 >> > ();

    // Synchronize the device to ensure the printf completes
    cudaDeviceSynchronize();

    // Print from the CPU
    printf("\nHello World from CPU!\n");

    return 0;
}
```

Résumé : Récapitulation des points clés

- L'**informatique parallèle hétérogène** est essentielle, car elle oppose la programmation séquentielle à la programmation parallèle.
- La **taxonomie de Flynn** classe les architectures en SISD, SIMD, MISD et MIMD.
- **Parallélisme** : Le parallélisme des tâches répartit les tâches entre les cœurs et le parallélisme des données répartit les données.
- **Latence, bande passante, débit** : Mesures de performance importantes.
- **Organisation de la mémoire** : Les systèmes peuvent être multi-nœuds (mémoire distribuée) ou multiprocesseurs (mémoire partagée).
- Les **GPU** sont optimisés pour les tâches parallèles aux données, tandis que les CPU sont conçus pour les tâches séquentielles complexes.
- La **plateforme CUDA** permet de programmer les GPU via des API, le code hôte s'exécutant sur les CPU et le code périphérique sur les GPU.
- Un **programme CUDA** de base démontre l'exécution de « Hello World » sur le GPU.