Corrigée Type de Contrôle de Calcul Parallèle

Questions du Cours : (04 points)

Q-1) Pourquoi les réseaux d'interconnexion multi-étages (MINs) sont-ils utilisés à la place des commutateurs crossbar à un seul étage ?

R-1) Les **réseaux d'interconnexion multi-étages** (MINs) sont utilisés à la place des **commutateurs crossbar à un seul étage** principalement pour des raisons de **coût**, de **complexité** et de **scalabilité**, surtout quand le nombre de ports devient élevé.

Q-2) Donnez un exemple de scénario de blocage dans un réseau Omega ?

R-2) Considérons un réseau Omega 8×8 (N = 8, donc $\log_2 8 = 3$ étages), composé de commutateurs 2×2.

Connexions demandées simultanément

- **Entrée 0 → Sortie 3 (011)**
- **Entrée 1 → Sortie 2 (010)**

Les sorties **2 (010)** et **3 (011)** ont les **deux premiers bits identiques**.

Q-3) Quelles sont les limites de CUDA par rapport à la programmation parallèle basée sur le CPU ?

R-3) CUDA offre un fort parallélisme massif grâce au GPU, mais il présente aussi plusieurs limites par rapport à la programmation parallèle basée sur le CPU. Ces limites sont à la fois architecturales, programmatiques et applicatives.

1. Latence et coût des transferts mémoire CPU ↔ GPU
2. Portabilité réduite
3. Performances faibles pour tâches séquentielles ou peu parallèles
4. Moins de flexibilité que le CPU
5. Modèle mémoire plus complexe
6. Synchronisation limitée
7. Mémoire limitée par rapport au CPU

Q-4) Quel est le rôle de l'hôte (host) et du Device dans CUDA ?

R-4) Le **host** est le programme qui s'exécute sur le **CPU**.

Le **device** correspond au **GPU**, où s'exécutent les kernels **CUDA**.

Exercice N° 01: (04 points)

Q-1) Appliquer la matrice de routage suivante sur un réseau OMEGA de 8x8:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 5 & 6 & 0 & 1 & 2 & 4 \end{pmatrix}$$

M=

Exercice N° 02 : (06 points)

Le tableau ci-dessous représente les temps d'exécution en secondes de 11 programmes différents.

Programmes	1	2	3	4	5	6	7	8	9	10	Moyen
Séquentiel	4.8211	8.2812	4.3985	3.2503	4.5570	3.2118	3.2208	3.3760	4.4168	3.2425	4.2776
Threads 2	3.3232	4.4650	3.5050	3.3009	3.3092	4.5649	3.3175	3.3447	4.1107	3.8410	3.7082
Threads 3	3.3464	3.3776	4.6079	3.4077	4.1804	4.5042	3.5958	3.4012	3.3858	4.6997	3.8507
Threads 4	3.3990	3.3879	4.5811	3.4650	3.3830	3.3713	4.5743	3.3740	3.3755	4.5096	3.7421
Threads 6	3.6643	3.5352	3.9529	4.5329	3.5873	3.5526	4.9193	3.5718	3.6162	4.8659	3.9798
Threads 12	3.7532	3.6965	4.7812	3.9506	3.6968	4.1479	4.6828	3.6765	3.6707	5.0324	4.1089
MPI 2	2.5700	2.5600	2.5800	2.5400	2.5600	2.5700	2.5600	2.5600	2.5600	2.5600	2.5655
MPI 3	2.6300	2.6200	2.6400	2.6500	2.6300	2.6400	2.6200	2.6400	2.6300	2.6400	2.6340
MPI 4	2.6000	2.5900	2.6100	2.5900	2.6000	2.5900	2.6100	2.5900	2.5900	2.6000	2.5999
GPU 16x16	0.0459	0.0459	0.0324	0.0236	0.0236	0.0236	0.0194	0.0196	0.0195	0.0188	0.0272
GPU 32x32	0.0192	0.0196	0.0193	0.0191	0.0205	0.0186	0.0202	0.0195	0.0193	0.0203	0.0196

Q-1) Calculer les valeurs de l'**accélération** et de l'**efficacité** dans chaque cas, ensuite dessiner des courbes ? **(02 points)**

R-1)

1) **Threads :**

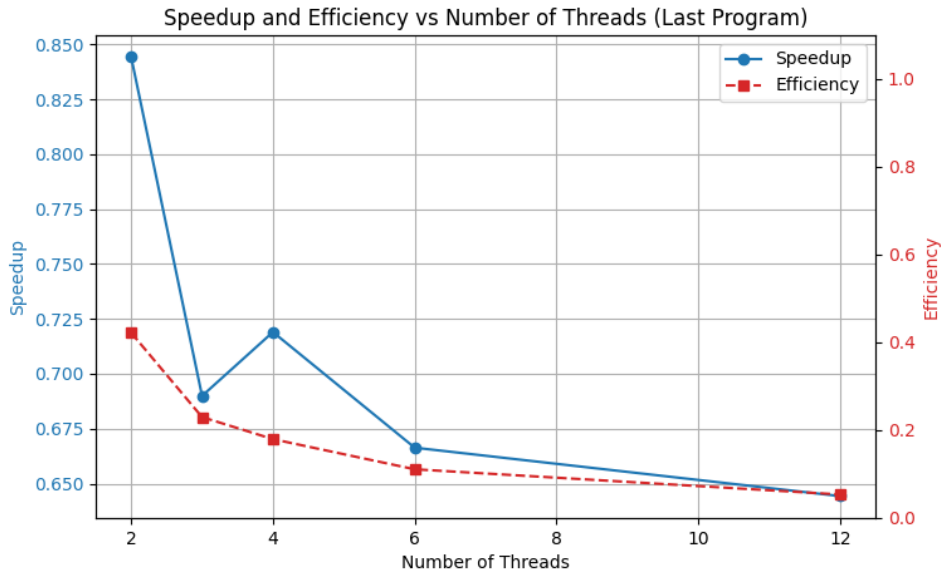
p= 2 speedup = 0.8441812028117678 efficiency = 0.4220906014058839

p= 3 speedup = 0.6899376555950381 efficiency = 0.22997921853167935

p= 4 speedup = 0.7190216427177577 efficiency = 0.17975541067943943

p= 6 speedup = 0.6663720997143386 efficiency = 0.11106201661905644

p= 12 speedup = 0.6443247754550513 efficiency = 0.05369373128792094

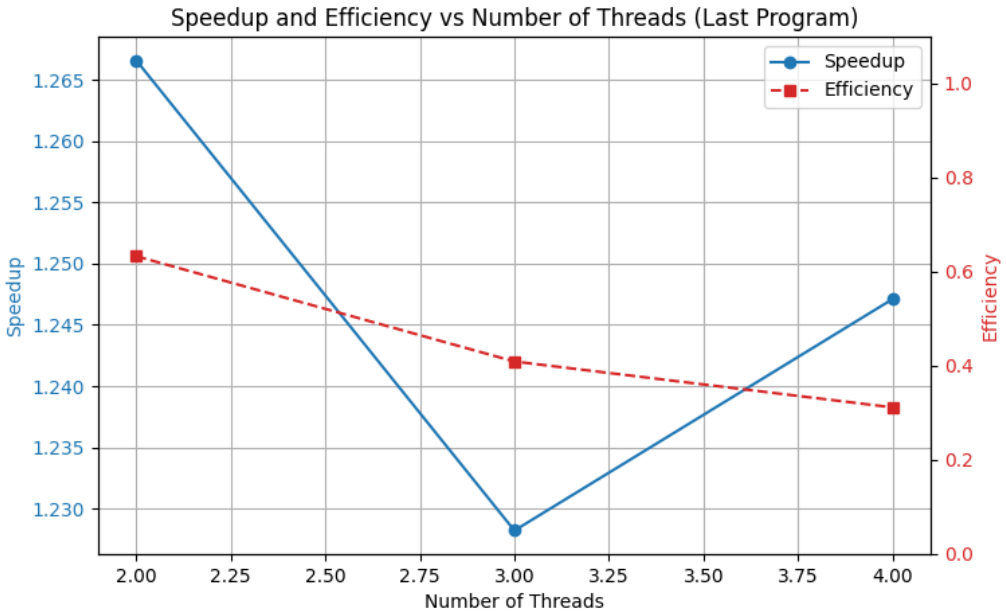


2) MPI :

p= 2 speedup = 1.2666015625 efficiency = 0.63330078125

p= 3 speedup = 1.228219696969697 efficiency = 0.4094065656565657

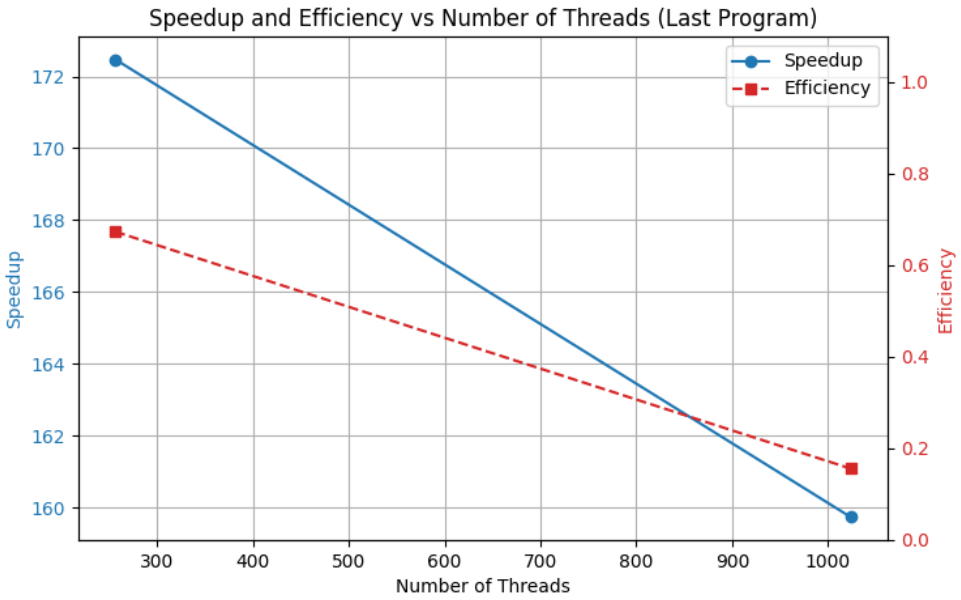
p= 4 speedup = 1.2471153846153846 efficiency = 0.31177884615384616



3) GPU :

p= 256 speedup = 172.47340425531914 efficiency = 0.6737242353723404

p= 1024 speedup = 159.72906403940888 efficiency = 0.15598541410098524



Q-2) Donnez vos remarques sur les différentes courbes ? **(02 points)**

R-2) Remarques :

1) Threads :

- **L’accélération augmente très faiblement** avec le nombre de threads.
- Au-delà de 4 threads, les performances se dégradent.
- **L’efficacité** chute fortement, atteignant moins de **10 %** avec 12 threads.

Interpretation:

- Surcharge (création de threads, synchronisation, contention mémoire)
- Parallélisme limité (Amdahl’s Law)
- Charge de travail insuffisante pour justifier l’utilisation de nombreux threads

2) **MPI:**

- MPI montre une meilleure accélération que les threads
- Meilleur résultat avec 2 processus MPI
- L’efficacité reste nettement supérieure à celle des threads

Interpretation:

- Meilleure répartition de la charge de travail
- **Moins de contention sur la mémoire partagée**
- Le coût de communication commence à limiter le passage à l’échelle au-delà de 2–3 processus
- MPI est plus efficace que les threads pour cette charge de travail

3) **GPU:**

- L’accélération est deux ordres de grandeur plus élevée
- GPU 32×32 est plus rapide que 16×16
- L’efficacité diminue avec l’augmentation du nombre de threads

Important note:

- L’efficacité du GPU n’est pas directement comparable à celle du CPU
- Les threads GPU sont légers
- Des milliers de threads sont nécessaires pour masquer la latence mémoire
- Une faible efficacité ne signifie pas de mauvaises performances

Le GPU surpasse clairement les approches CPU

Q-3) Déterminer le point de saturation : à partir de combien de processus l’accélération n’augmente plus significativement ? **(01 points)**

R-3) Points de saturation :

1) **Threads:**

- Le meilleur speedup est atteint avec 2–4 threads
- À partir de 4 threads, l’accélération stagne puis diminue
- Les gains supplémentaires deviennent négligeables (< 3%)

Alors, le point de saturation ≈ 4 threads

2) **MPI :**

- Le speedup maximal est déjà atteint avec 2 processus
- Les variations entre 2, 3 et 4 processus sont très faibles
- L'augmentation du nombre de processus n'apporte plus de gain réel

Alors, le point de saturation ≈ 2 processus MPI

Q-4) Comparer les résultats avec le nombre idéal de processus : pourquoi certaines versions parallèles (ex : 12 processus) ont un temps moyen supérieur à 6 processus ? **(01 points)**

R-4) Le nombre idéal de processus est atteint bien avant le maximum disponible.

Au-delà de ce point (ici ≈ 4–6 processus), l'augmentation du nombre de processus entraîne une surcharge (synchronisation, contention mémoire, partie séquentielle du code) supérieure au gain apporté par le parallélisme. C'est pourquoi certaines versions parallèles, comme celle à 12 processus, présentent un temps moyen supérieur à celle à 6 processus.

Exercice N° 03 : (06 points)

Q-1) Écrire un programme Python utilisant la bibliothèque MPI pour calculer le produit de deux matrices **R**=**A** × **B**.

Données :

- **A** : matrice de taille **L** × **K**
- **B**: matrice de taille **K** × **C**
- **R**: matrice résultat de taille **L** × **C**

R-1)

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Dimensions des matrices
L = 8   # lignes de A
K = 6   # colonnes de A = lignes de B
C = 5   # colonnes de B

rows_per_proc = L // size

# ====================
# Processus 0 (HOST)
# ====================
if rank == 0:
    A = np.random.rand(L, K)
    B = np.random.rand(K, C)
    R = np.zeros((L, C))

    # Envoi des données aux autres processus
    for p in range(1, size):
```

```
            start = p * rows_per_proc
            end = start + rows_per_proc
            comm.Send(A[start:end, :], dest=p, tag=0)
            comm.Send(B, dest=p, tag=1)

        # Calcul local (partie du processus 0)
        A_local = A[0:rows_per_proc, :]
        R[0:rows_per_proc, :] = np.dot(A_local, B)

        # Réception des résultats
        for p in range(1, size):
            start = p * rows_per_proc
            end = start + rows_per_proc
            comm.Recv(R[start:end, :], source=p, tag=2)

        print("Matrice A :\n", A)
        print("\nMatrice B :\n", B)
        print("\nMatrice R = A x B :\n", R)

    # ====================
    # Autres processus
    # ====================
    else:
        A_local = np.zeros((rows_per_proc, K))
        B = np.zeros((K, C))
        R_local = np.zeros((rows_per_proc, C))

        # Réception des données
        comm.Recv(A_local, source=0, tag=0)
        comm.Recv(B, source=0, tag=1)

        # Calcul local
        R_local = np.dot(A_local, B)

        # Envoi du résultat
        comm.Send(R_local, dest=0, tag=2)
```

Bon Courage **Bonne Chance**