**Institut Supérieur des Sciences Appliquées et de Technologie de Sousse**

****

**Algorithmique& Architectures parallèles**

Examen TP CUDA

Prof : Tahar ALIMI

A.U : **2022/2023**

Section : **FIA2-GL**

Durée : **1H30**

Date : **Avril 2024**

***Important :***

* L’examen comporte 3 exercices répartis sur trois pages.
* Les réponses aux questions pratiques à remettre seront envoyées impérativement à la fin de l’épreuve à l’adresse : alimitahar2020@gmail.com

**Exercice 1 : (04 pts = 3 + 1)**

On a exécuté le code **CUDA** suivant sur le cloud (**Google Colab**) mais on n’a obtenu en affichage ni résultat ni erreur :

***#include <stdio.h>***

***#include <cuda.h>***

***#include <cuda\_runtime.h>***

***#include <device\_launch\_parameters.h>***

***\_\_global\_\_ void hello\_GPU()***

***{***

***printf("Hello world!\n");***

***}***

***int main()***

***{***

***hello\_GPU<<<1,1>>>();***

***}***

**Questions :**

1. Expliquez – avec détail – ce que s’est-il passé lors de l’exécution en précisant la raison de ne pas avoir d’affichage en sortie ?
2. Ajoutez l’instruction permettant de rendre ce code parfait ?

**Exercice 2 : (08 pts = 1 + 1 + 2 + 2 + 2)**

A l'intérieur du noyau (kernel), nous pouvons obtenir les paramètres de l'organisation à travers les variables automatiques associées à chaque thread. Ces variables sont: **threadIdx**, **blockIdx**, **blockDim**, **gridDim**.

Pour une organisation simple et unidimensionnelle, nous traitons l'index x fourni pour les threads et les blocs: **threadIdx.x**, **blockIdx.x**, **blockDim.x** et **gridDim.x**.

Veuillez-vous servir du code nommé "**premier\_exemple**" du fichier partagé à l'adresse URL:

**https://docs.google.com/document/d/1A5sm9ADi\_fC5lkYUbHA0qWAIlU5XNbwHCCCv0KOngwQ/edit**

**Questions :**

1. Dans le kernel du code **CUDA** ci-avant, on a déclaré le **i** comme suit :

***int i = threadIdx.x + blockIdx.x \* blockDim.x;***

Expliquez bien cette instruction. (Discutez tous les cas possibles)

1. Supposons que le kernel est appelé de cette façon : **addVect<<<ceil(n/256), 256>>>(Cv1, Cv2, Cres);**

Pour une taille des vecteurs supposée égale à **1000**, déterminez le nombre de blocs et de threads par bloc.

1. En gardant la déclaration du kernel comme noté dans le code ci-joint, et en prenant son appel défini dans la question précédente **(b)** avec une taille du vecteur égale à 1000, on aura un problème en rapport avec le résultat obtenu et/ou l'optimisation exigée du code? Expliquez en proposant une solution !
2. Distinguez clairement la différence entre ces deux déclarations possibles du kernel **addVect** :

* **Déclaration 1 :**

\_\_global\_\_ void addVect(int \*a, int \*b, int \*c)

{

int index = threadIdx.x + blockIdx.x \* blockDim.x;

if (index < N)

c[index] = a[index] + b[index];

}

* **Déclaration 2:**

\_\_global\_\_ void addVect(int \*a, int \*b, int \*c)

{

int index = threadIdx.x + blockIdx.x \* blockDim.x;

while (index < N)

{

c[index] = a[index] + b[index];

index += blockDim.x \* gridDim.x;

}

}

1. Comme vous le savez, le **GPU** est un coprocesseur graphique permettant également d’alléger le **CPU** d’une certaine quantité de travail. Expliquez le phénomène général de transfert des données entre la mémoire du **CPU** et celle du **GPU** lors d’une exécution parfaite d’un code **CUDA**.

**Exercice 3 : CUBE (8 pts = 5 + 3)**

Il s'agit ici de réaliser une opération **Cube** permettant d’enlever au cube les éléments d’un vecteur d’entiers.

Dans un programme classique (voir l’URL ci-dessous) avec un seul **CPU**, il faut utiliser une répétition et parcourir successivement tous les éléments du vecteur. Avec le **GPU**, on pourra traiter ça en parallèle (simultanément) en se servant de blocs et de threads en **CUDA**.

Pour cela, il faut :

1. Initialiser (charger) le vecteur de départ,
2. Le transférer dans la mémoire du **GPU**,
3. Effectuer le calcul sur le **GPU**,
4. Récupérer le résultat en mémoire centrale.

**N.B :**

Le code classique (nommé **Cube\_Seq**) et le squelette du code **CUDA** (nommé **Cube\_Cuda)** se trouvent dans le fichier partagé à l'adresse URL suivante :

**https://docs.google.com/document/d/1A5sm9ADi\_fC5lkYUbHA0qWAIlU5XNbwHCCCv0KOngwQ/edit**

**Questions :**

1. En vous basant sur le squelette situé à l’URL ci-dessus, écrivez le code ***le plus optimal*** effectuant le calcul du **Cube** d’un vecteur numérique (d’entiers) donné. Expliquez votre choix.
2. D’une façon générale, quelle est la différente – purement technique – entre la parallélisation (parallélisme) du code précédent (question a) et de celui permettant de chercher la somme des éléments d’un vecteur d’entiers donné dont une solution en **CUDA C/C++** se trouvant à l’URL ci-dessous ?

* Le squelette du code **CUDA** qui calcule la somme des éléments d’un vecteur d’entiers (nommé **sum\_elem\_cuda**) se trouve dans le fichier partagé à l'adresse URL suivante :

**https://docs.google.com/document/d/1A5sm9ADi\_fC5lkYUbHA0qWAIlU5XNbwHCCCv0KOngwQ/edit**

**Bon Courage**