TP – Initiation à la programmation GPU (CUDA)

---

1. Objectifs du TP

Ce TP a pour but de vous faire découvrir les bases de la programmation GPU avec CUDA :

- Apprendre à connaitre les outils de base proposés par Nvidia pour monitorer le GPU de votre machine.
- Écrire une première application d'addition de vecteurs sur CPU puis sur GPU [web:3][web:17]
- Mesurer et comparer les performances CPU/GPU [web:7][web:13]
- Implémenter un calcul d'histogramme sur CPU puis sur GPU en utilisant des opérations atomiques [web:6][web:18]

---

2. Organisation générale

Vous travaillerez en C/C++ avec CUDA,. La compilation se fera avec `nvcc` (le compilateur CUDA) en ligne de commande dans un terminal ou via une MakeFile.

Exemple indicatif de compilation :

```
nvcc -O2 -o prog prog.cu
```

3. Découverte des outils NVIDIA

But : Comprendre l'environnement GPU de la machine et ses capacités avant de programmer.

3.1 Commande `nvidia-smi`

Objectif : Obtenir des informations sur le GPU et les processus qui l'utilisent.

1. **Exécutez** `nvidia-smi` dans un terminal et observez les informations affichées :

```
nvidia-smi
```

2. **Identifiez et notez** dans le tableau ci-dessous :

Information demandée	Valeur observée
Nom du GPU	
Version CUDA supportée	
Mémoire totale GPU (GB)	
Utilisation mémoire actuelle	
Driver version	
Nombre de GPU(s)	

3. **Observez** : des processus utilisent-ils le GPU, Combien, lequels?

3.2 Commandes avancées `nvidia-smi`

1. **Affichage continu** (toutes les 2 secondes) :

```
nvidia-smi -l 2
```

(Ctrl+C pour arrêter)

2. **Historique des performances** (5 dernières minutes) :

```
nvidia-smi dmon -s pucv -c 10
```

3. **Nettoyage des processus GPU** (si nécessaire) :

```
nvidia-smi --gpu-reset
```

3.3 Vérification de l'environnement CUDA

1. **Version du compilateur CUDA** :

```
nvcc --version
```

2. **Test de compilation CUDA** :

```
nvcc -o test_version -lcudart version.cu
```

Avec le fichier `version.cu` suivant :

```
#include <cuda_runtime.h>
#include <stdio.h>

int main() {
  int deviceCount;
  cudaGetDeviceCount(&deviceCount);
  printf("Nombre de GPU CUDA détectés : %d\n", deviceCount);
  printf("Version CUDA runtime : %d\n", CUDART_VERSION);
  return 0;
}
```

3. **Exécutez** `./test_version` et notez le résultat.
4. En utilisant la documentation de CUDA, quelles autres informations peut-on obtenir. Ecrivez un programme qui permet d'afficher vos recherches.

3.4 Questions

1. Quelle est la capacité mémoire de votre GPU ? Commentez.
2. Quelle version CUDA est installée ? Est-elle compatible avec `nvcc` ?
3. Si plusieurs GPU sont détectés, comment spécifier quel GPU utiliser dans un programme CUDA ?
4. Que se passerait-il si vous lancez un programme qui consomme toute la mémoire GPU ?

---

4. Exercice 1 – Addition de vecteurs sur CPU

But : Mettre en place une version séquentielle de référence.

1. Écrire un programme C/C++ qui :
    - Définit ou lit une taille de vecteur `N` (par exemple `N = 10^7`)
    - Alloue dynamiquement trois tableaux de `float` ou `double` : `A`, `B`, `C`
    - Initialise les tableaux d'entrée, par exemple :
        - `A[i] = i`
        - `B[i] = 2*i`
    - Calcule `C[i] = A[i] + B[i]` dans une boucle séquentielle classique
2. **Vérifier la correction** :
    - Contrôler quelques valeurs (par exemple `C[0]`, `C[1]`, `C[N-1]`)
    - Optionnel : calculer l'erreur maximale par rapport à une valeur théorique attendue
3. **Mesurer le temps** d'exécution de la boucle d'addition uniquement (sans compter l'allocation et l'initialisation) avec une fonction de mesure de temps (par exemple `clock_gettime` ou `std::chrono`)
4. Noter le temps obtenu pour au moins une valeur de `N` (par exemple `10^7`)

---

5. Exercice 2 – Addition de vecteurs sur GPU (CUDA)

But : Porter l'addition de vecteurs sur GPU avec un kernel CUDA.

1. Conserver la partie CPU de l'exercice 1 (allocation et initialisation de `A` et `B`)
2. **Sur le GPU** :
    - Allouer trois tableaux `A_d`, `B_d`, `C_d` en mémoire globale avec `cudaMalloc`
    - Copier `A` et `B` du CPU vers le GPU avec `cudaMemcpy`
3. **Écrire un kernel CUDA** de forme générale :

```
global void vectorAdd(const float* A, const float* B, float* C, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}
```

4. **Choisir une configuration de lancement** :

- Taille de bloc typique : `blockDim.x = 256`
- Nombre de blocs : `gridDim.x` calculé pour couvrir au moins `N` éléments

5. Lancer le kernel, synchroniser le GPU (`cudaDeviceSynchronize`), puis recopier `C_d` vers `C` sur le CPU
6. **Vérifier** que la version GPU produit les mêmes résultats que la version CPU :

- Calculer l'erreur maximale entre les deux versions et vérifier qu'elle est inférieure à un petit seuil (par exemple `1e-5`)

---

6. Exercice 3 – Comparaison de performances CPU vs GPU

But : Observer l'intérêt du GPU en fonction de la taille du problème.

1. Choisir plusieurs tailles de vecteurs, par exemple :

- `N = 10^5`, `10^6`, `10^7`, `10^8`

1. **Pour chaque valeur de `N`** :

- Mesurer le temps CPU de l'addition (boucle seule)
- Mesurer le temps GPU, avec deux approches possibles :
    - Temps « global » : transferts hôte → device, lancement du kernel, transfert device → hôte
    - Temps « kernel seul » : utiliser des événements CUDA (`cudaEvent_t`)

1. **Présenter les mesures** sous forme de tableau :

N	Temps CPU (ms)	Temps GPU global (ms)	Temps kernel GPU (ms)
1e5			
1e6			
1e7			
1e8			

4. **Commenter brièvement** (quelques lignes) :

- À partir de quelle taille de problème le GPU devient-il plus rapide que le CPU ?
- Quel est l'impact des transferts de données sur le temps global ?

---

7. Exercice 4 – Histogramme sur CPU puis sur GPU

But : Mettre en œuvre une opération plus complexe (histogramme) et introduire les opérations atomiques

7.1 Version CPU

1. Considérer un tableau `data` de `N` éléments de type `unsigned char` (valeurs de 0 à 255)
2. Initialiser `data` avec des valeurs aléatoires ou un motif déterministe
3. Allouer un tableau `hist[256]` d'entiers, initialisé à zéro
4. **Calculer l'histogramme** sur CPU :

- Pour chaque `i` de `0` à `N-1`, incrémenter `hist[data[i]]`

5. **Vérifier** que la somme des 256 cases de `hist` est égale à `N`

7.2 Version GPU avec `atomicAdd`

1. Allouer `data_d` (copie GPU de `data`) et `hist_d` (256 entiers) en mémoire globale GPU
2. Copier `data` vers `data_d`, initialiser `hist_d` à zéro (`cudaMemset`)
3. **Écrire un kernel CUDA** qui :

- Calcule un indice global `i` et un pas `stride`
- Parcourt les indices `i, i+stride, i+2*stride, ...` tant que `< N`
- Pour chaque élément lu, effectue `atomicAdd(&hist_d[val], 1)` où `val` est `data_d[i]`

4. Après l'exécution du kernel, recopier `hist_d` vers un tableau `hist_gpu[256]` sur le CPU
5. **Comparer** `hist_gpu` et `hist` (version CPU) : toutes les cases doivent être identiques
6. Mesurer et comparer les temps CPU/GPU.
