Projet d'Architectures et Modèles de Calcul

1 Multiplication de matrices en C avec AVX-512

L'objectif de cette partie du projet est d'implémenter une multiplication de matrices $C = A \times B$ en langage C en utilisant les intrinsics AVX-512. Vous devrez :

- écrire une version naïve en C ;
- écrire une version optimisée utilisant les intrinsics AVX-512 ;
- exécuter et tester le code sur une machine ne disposant pas d'AVX-512 grâce à l'émulateur Intel SDE.

L'accent sera mis sur la manipulation directe des registres vectoriels et de fonctions `_mm512_*()` pour exploiter le parallélisme SIMD à grande largeur.

1.1 Intrinsics AVX-512 utiles

L'architecture AVX-512 permet d'effectuer des opérations vectorielles sur 512 bits, soit :

- 16 flottants simple précision (32 bits),
- ou 8 flottants double précision (64 bits),
- ou des vecteurs entiers de taille équivalente.

Vous utiliserez exclusivement la famille d'intrinsics `_mm512_*`. Voici les principales instructions à utiliser pour réaliser une multiplication de matrices optimisée :

- `_mm512_loadu_ps` : charge un vecteur de 16 floats.
- `_mm512_storeu_ps` : stocke un vecteur de 16 floats.
- `_mm512_set1_ps` : copie un flottant dans les 16 cases du vecteur.
- `_mm512_mul_ps` : multiplication de deux vecteurs.
- `_mm512_add_ps` : addition de deux vecteurs.
- `_mm512_fmadd_ps` : Fused Multiply-Add (a*b + c).

Exemple de code :

```c
#include <stdio.h>
#include <immintrin.h>   // Nécessaire pour AVX-512

int main() {
    // 16 floats = 512 bits
    float A[16], B[16], C[16];

    // Initialisation des données
    for (int i = 0; i < 16; i++) {
        A[i] = i * 1.0f;
        B[i] = 2.0f;
        C[i] = 1.0f;
    }

    // Chargement en registres AVX-512
    __m512 vA = _mm512_loadu_ps(A);
    __m512 vB = _mm512_loadu_ps(B);
    __m512 vC = _mm512_loadu_ps(C);

    // Calcul vectorisé : C = A * B + C
    __m512 result = _mm512_fmadd_ps(vA, vB, vC);

    // Stockage du résultat
    _mm512_storeu_ps(C, result);

    // Affichage
    printf("Resultat:\n");
    for (int i = 0; i < 16; i++) {
        printf("%f ", C[i]);
    }
    printf("\n");

    return 0;
}
```

Ce code se compile par la commande `gcc -O3 -mavx512f test.c -o test` et s'exécute avec l'émulateur Intel SDE par `./sde64 -icl -- ./test`

1.2 Travail demandé

Vous devez :

1. Écrire une fonction naïve de multiplication matricielle utiliosant le type `float`.
2. Écrire une version vectorisée AVX-512 :
   - chargement des colonnes/lignes dans des registres de 512 bits ;
   - utilisation de `_mm512_fmadd_ps` pour accumuler les produits.
3. Exécuter votre programme sur Intel SDE si votre machine ne supporte pas AVX-512.

1.3 Utilisation de l'émulateur Intel SDE

Intel SDE (Software Development Emulator) permet d'exécuter sur n'importe quelle machine un programme contenant des instructions AVX-512.

- Téléchargez SDE depuis : `https://www.intel.fr/content/www/fr/fr/download/684897/intel-software-development-emulator.html`
- Décompressez l'archive :

  ```
  tar -xvf sde-external-*.tar.gz
  ```

- Pour exécuter votre programme :

  ```
  ./sde64 -icl -- ./matmul_avx512
  ```

2 Calcul de π par méthode stochastique

On cherche à calculer π avec un algorithme de type Monte-Carlo.

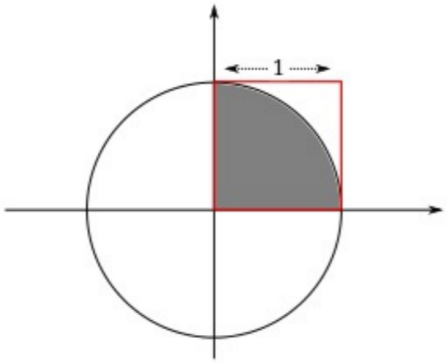

Figure 1: La surface grise vaut $\pi/4$.

L'aire du cercle de rayon unité est π, donc son aire sur $(x, y) \in [0, 1] \times [0, 1]$ est $\frac{\pi}{4}$. La fonction tracée dans le carré rouge de la figure 1 est $y = \sqrt{1 - x^2}$ donc

$$\int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4} \tag{1}$$

On peu ainsi utiliser l'algorithme suivant :

1. $N_{in} = 0$; $N_{total} = 0$
2. Tirer x aléatoirement uniformément dans $[0, 1]$
3. Tirer y alématoirement uniformément dans $[0, 1]$
4. Si $x^2 + y^2 < 1$ alors (x, y) est dans le cercle et on incrémente N_{in}
5. Incrémenter N_{total}
6. Go to 2

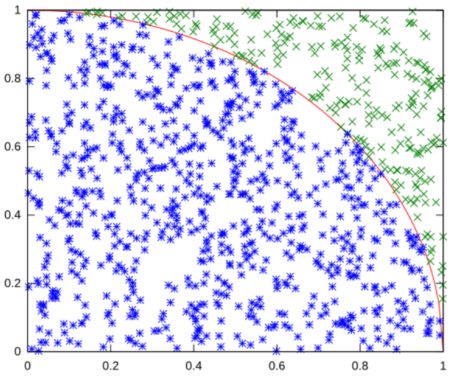

Figure 2: La valeur de $\pi/4$ est égale au rapport du nombre de points bleus sur le nombre total de points.

7. $\frac{N_{in}}{N_{total}} = \frac{\pi}{4}$

On utilisera une implémentation de type client/serveur. Le serveur est un processus qui collecte les résultats partiels calculés par les clients. Chaque client calcule un paquet de 10 millions de tirages et estime la valeur de π. Cette valeur (échantillon) est envoyée au serveur, et à la réception du message le serveur retourne une instruction au client lui donnant l'ordre de recalculer un nouvel échantillon ou de s'arrêter.

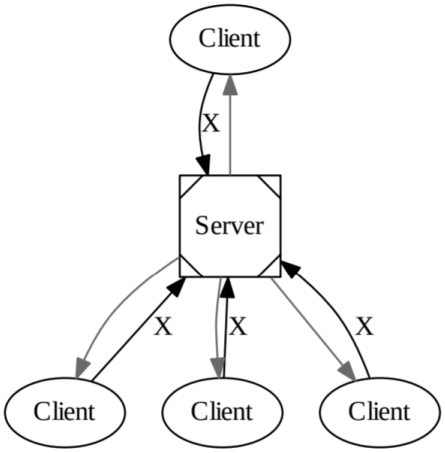

Figure 3: Modèele client/serveur.

Puisque tous les échantillons sont indépendants, ils ont une distribution gaussienne centrée sur la valeur estimée de π. La barre d'erreur est donc proportionnelle à $\frac{1}{\sqrt{N}}$ ou N est le nombre de tirages. Avec cette méthode, plus le nombre de tirages est grand, plus le résultat est précis. A temps de restitution constant, on utilisera la parallélisme pour améliorer la précision du calcul. Notons que cet algorithme est parfaitement scalable (embarrassingly parallel), et peut utiliser sans problème plusieurs dizaines de milliers de coeurs. On vérifiera que pour une durée de 10 secondes, le nombre de tirages augmente linéairement avec le nombre de clients.

Objectifs du projet

1. Implémenter en Python un programme MPI client/serveur suivant la description ci-dessus (utiliser `mpi4py`).

2. Mesurer la précision et la convergence de l'estimation de π en fonction du temps et du nombre de clients.

3. Vérifier que, pour un nombre fixe de tirages sur chaque processeur, la précision augmente avec le nombre de clients.

Consignes d'exécution

- Installer `mpi4py` et `numpy` : `pip install mpi4py numpy` (ou via votre gestionnaire de paquets).
- Lancer le programme avec `mpirun` / `mpiexec` : `mpirun -np 5 python3 pi.py` ici 1 serveur + 4 clients.

2.1 Description détaillée du protocole client/serveur

Messages échangés

- **client → serveur** : message contenant le couple $(N_{\text{in\_local}}, N_{\text{total\_local}})$ pour l'échantillon courant ;
- **serveur → client** : commande texte (par exemple : `"CONTINUE"` ou `"STOP"`) indiquant au client s'il doit calculer un autre paquet ou s'il doit s'arrêter.

Le serveur décide d'arrêter lorsqu'une précision atteinte (condition d'arrêt liée à l'erreur, lorsque la valeur de π ne varie plus).

2.2 Les instructions MPI en Python (mpi4py)

Vous utiliserez `mpi4py`, qui fournit une interface Python à MPI. voici les primitives les plus utiles pour ce projet :

- `from mpi4py import MPI`
  accès à l'instance MPI.
- `comm = MPI.COMM_WORLD`
  le communicateur global contenant tous les processus.
- `rank = comm.Get_rank()`
  rang du processus courant (0 = serveur).
- `size = comm.Get_size()`
  nombre total de processus (doit être ≥ 2).
- `comm.send(obj, dest=..., tag=...)` et `comm.recv(source=..., tag=...)`
  envoi/réception de messages *python objects* (sérialisés automatiquement). Simple et pratique pour petits messages (tuples, dicts).
- `MPI.ANY_SOURCE`, `MPI.ANY_TAG`
  utilisés par le serveur pour recevoir depuis n'importe quel client.
- `status = MPI.Status()` et `comm.recv(..., status=status)`
  pour récupérer des informations sur le message (source, tag, etc.) après réception.

Exemple de code :

```
from mpi4py import MPI

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

if size < 2:
    if rank == 0:
        print("Ce programme nécessite au moins 2 processus MPI.")
    exit()

if rank == 0:
    # Processus 0 envoie un message vers le processus 1
    message = "Bonjour du processus 0"
    print(f"[0] Envoi : {message}")
    comm.send(message, dest=1, tag=10)

    # Processus 0 attend la réponse
    reponse = comm.recv(source=1, tag=20)
    print(f"[0] Reçu : {reponse}")

elif rank == 1:
    # Processus 1 reçoit le message du processus 0
    msg = comm.recv(source=0, tag=10)
    print(f"[1] Message reçu de 0 : {msg}")

    # Processus 1 renvoie une réponse
    reponse = "Bien reçu, ici processus 1 !"
    comm.send(reponse, dest=0, tag=20)
    print(f"[1] Réponse envoyée à 0.")
```