Challenge NumPy vs uFuncs

Cours d'Optimisation Numérique

Concepts Clés

Que sont les uFuncs ?

Les uFuncs sont des fonctions NumPy qui opèrent élément par élément sur des tableaux entiers, en utilisant du code compilé pour des performances optimales.

Avantages des uFuncs

— **Vectorisation** : Opérations parallèles sur tous les éléments
— **Réduction mémoire** : Évite les tableaux temporaires
— **Performance C** : Exécution en code natif compilé
— **Broadcasting** : Gestion automatique des dimensions

Challenge Principal

Fonction Mathématique à Implémenter

$$f(x) = \sin(x^2) + \cos(2x) \times \exp\left(-\frac{x}{10}\right) + \frac{\log(|x| + 1)}{x^2 + 1}$$

Partie 1 : Implémentation NumPy Standard

Code de Référence

```python
import numpy as np
import time

def calcul_numpy_standard(arr):
    """Version standard avec NumPy"""
    terme1 = np.sin(arr**2)
    terme2 = np.cos(2 * arr)
    terme3 = np.exp(-arr / 10)
    terme4 = np.log(np.abs(arr) + 1)
    terme5 = arr**2 + 1

    resultat = terme1 + terme2 * terme3 + terme4 / terme5
    return resultat

# Benchmark
def benchmark_standard():
    sizes = [10**5, 10**6, 10**7, 5*10**7]
    temps_standard = []

    for size in sizes:
        arr = np.random.randn(size)

        start = time.time()
        result = calcul_numpy_standard(arr)
        temps_exec = time.time() - start

        temps_standard.append(temps_exec)
        print(f"Taille {size}: {temps_exec:.4f} secondes")

    return temps_standard
```

Partie 2 : uFunc Personnalisée avec Numba

Optimisation avec uFunc

```
from numba import vectorize, float64
import math

@vectorize([float64(float64)], nopython=True, target='parallel')
def ufunc_optimisee(x):
    """uFunc personnalisée optimisée"""
    x_carre = x * x
    sin_term = math.sin(x_carre)
    cos_term = math.cos(2 * x)
    exp_term = math.exp(-x / 10)
    log_term = math.log(abs(x) + 1)
    denom = x_carre + 1

    return sin_term + cos_term * exp_term + log_term / denom

def benchmark_ufunc():
    sizes = [10**5, 10**6, 10**7, 5*10**7]
    temps_ufunc = []

    for size in sizes:
        arr = np.random.randn(size)

        start = time.time()
        result = ufunc_optimisee(arr)
        temps_exec = time.time() - start

        temps_ufunc.append(temps_exec)
        print(f"Taille {size}: {temps_exec:.4f} secondes")

    return temps_ufunc
```

Questions du Challenge

Question 1 : Analyse de Performance

1. Implémentez les deux versions et comparez leurs temps d'exécution
2. Calculez le **speedup** : speedup $= \frac{T_{\text{standard}}}{T_{\text{ufunc}}}$
3. Tracez un graphique comparatif des performances

Question 2 : Analyse Mémoire

1. Utilisez `memory_profile` pour mesurer l'utilisation mémoire
2. Comptez le nombre de tableaux temporaires créés dans chaque version
3. Expliquez pourquoi les uFuncs réduisent l'empreinte mémoire

Question 3 : uFunc Avancée

Créez une uFunc pour le calcul de gradient numérique :

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

```
@vectorize([float64(float64, float64)], nopython=True)
def gradient_numerique(f_x_plus, f_x_minus, h=1e-7):
    return (f_x_plus - f_x_minus) / (2 * h)
```

Question 4 : Application Réelle

Appliquez vos uFuncs à un problème de machine learning :

```
# Fonction de coût pour régression linéaire
@vectorize([float64(float64, float64)], nopython=True)
def mse_loss(y_true, y_pred):
    return (y_true - y_pred) ** 2

# Dérivée de la fonction sigmoïde
@vectorize([float64(float64)], nopython=True)
def sigmoid_derivative(x):
    sig = 1 / (1 + math.exp(-x))
    return sig * (1 - sig)
```

Benchmark Complet

Code de Test Intégral

Benchmark Complet

```python
def benchmark_complet():
    sizes = [10**5, 10**6, 10**7]

    print("=== BENCHMARK NumPy vs uFuncs ===")
    print(f"{'Taille':<10} {'NumPy (s)':<12} {'uFunc (s)':<12} {'Speedup':<10}")
    print("-" * 50)

    for size in sizes:
        arr = np.random.randn(size)

        # Version NumPy standard
        start = time.time()
        result_std = calcul_numpy_standard(arr)
        time_std = time.time() - start

        # Version uFunc
        start = time.time()
        result_ufunc = ufunc_optimisee(arr)
        time_ufunc = time.time() - start

        # Vérification de l'exactitude
        diff_max = np.max(np.abs(result_std - result_ufunc))
        speedup = time_std / time_ufunc

        print(f"{size:<10} {time_std:<12.4f} {time_ufunc:<12.4f} {speedup:<10.2f}x"
        print(f"Erreur max: {diff_max:.2e}")

    return time_std, time_ufunc, speedup
```

Critères d'Évaluation

— **Exactitude (30%)** : Résultats numériquement corrects
— **Performance (40%)** : Gain de vitesse mesurable
— **Analyse (20%)** : Qualité de l'analyse technique
— **Innovation (10%)** : Créativité dans les optimisations

Extensions Avancées

uFunc Multi-paramètres

```python
@vectorize([float64(float64, float64, float64)], nopython=True)
def ufunc_multiparam(x, y, z):
```

```
    return math.sin(x*y) + math.cos(y*z) * math.exp(-x/z)
```

Avec Gestion d'Erreurs

```
@vectorize([float64(float64)], nopython=True)
def ufunc_securisee(x):
    if x == 0:
        return 0.0  # Évite la division par zéro
    return math.log(abs(x)) / x
```