UNIVERSITÉ DE RENNES 1

Présentation projet final CPU/ GPU

Matthieu PETIT
Djemsay MORVAN
Ulysse CAROMEL

January 12, 2024

Table des matières

Méthodes d'intégration

1. Les différentes méthodes d'intégration
   - Méthode de Simpson
   - Méthode de Gauss 2D
   - Méthode de Runge-Kutta
   - Méthode de Monte-Carlo
   - Comparaison des méthodes

2. Outils de parallélisation
   - Open MP
   - MPI
   - CUDA

3. Résultats
   - OpenMP
   - MPI
   - CUDA

Méthode de Simpson

- Division de l'intervalle $[a, b]$ en sous-intervalles de largeur égale $h = \frac{b-a}{n}$, n étant pair.
- Approximation sur chaque sous-intervalle $[x_i, x_{i+2}]$:

$$\int_{x_i}^{x_{i+2}} f(x)\,dx \approx \frac{h}{3}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right]$$

- Estimation finale de l'intégrale :

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\Big[f(a) + 4f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(b)\Big]$$

Quadrature Gaussienne 2D

- Estimation numérique de l'intégrale d'une fonction $f(x, y)$ sur une région bidimensionnelle définie par $a \leq x \leq b$ et $c \leq y \leq d$.
- Utilise un ensemble de points et de poids associés pour approximer l'intégrale.
- Formule générale :

$$\iint_R f(x, y)\, dx\, dy \approx \sum_{i=1}^{n} \sum_{j=1}^{m} w_{ij} \cdot f(x_i, y_j)$$

- Formule spécifique pour un quadrilatère :

$$\iint_R f(x, y)\, dx\, dy \approx \sum_{i=1}^{n} \sum_{j=1}^{m} w_{ij} \cdot f\left(\frac{1}{2}(1 + \xi_i)x + \frac{1}{2}(1 - \xi_i)y, \frac{1}{2}(1 + \eta_j)x + \frac{1}{2}(1 - \eta_j)y \right)$$

- Offre une précision supérieure à la quadrature de Gauss unidimensionnelle.

Méthode de Runge-Kutta (RK4) pour les EDO

- Technique numérique pour résoudre des équations différentielles ordinaires (EDO).
- Méthode de Runge-Kutta d'ordre 4 (RK4) : équilibre entre précision et complexité.
- Forme générale d'une EDO du premier ordre :

$$\frac{dy}{dt} = f(t, y)$$

- Étapes de la méthode RK4 :

$$\begin{aligned} k_1 &= h \cdot f(t_n, y_n) \\ k_2 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= h \cdot f(t_n + h, y_n + k_3) \end{aligned}$$

- Mise à jour de la solution à chaque pas :

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- Offre une meilleure précision que des méthodes de pas fixe plus simples, largement utilisée pour sa robustesse et polyvalence.

Méthode de Monte Carlo pour l'intégration

- Utilisée pour estimer numériquement des intégrales complexes, notamment dans des espaces multidimensionnels.
- Fondée sur des principes probabilistes, associant le hasard à des calculs numériques.
- Estimation d'une valeur en utilisant des échantillons aléatoires dans un domaine donné.
- Génération de points aléatoires dans le domaine D.
- Évaluation de la fonction $f(x, y)$ pour chaque point généré.
- Calcul de la moyenne pondérée des valeurs de $f(x, y)$ pour les points générés, multipliée par la mesure de D.

Formule d'estimation de Monte Carlo pour l'intégrale :

$$I \approx A \cdot \frac{1}{N} \sum_{i=1}^{N} f(x_i, y_i)$$

Avantages : Flexibilité, capacité à traiter des problèmes complexes en dimensions élevées. Précision dépend du nombre de points aléatoires générés.

Comparaison des méthodes numériques

Runge-Kutta (RK4)

+ Précision élevée.
+ Adaptée aux EDO.
- Plus de calculs.

Gaussienne 2D

+ Haute précision.
+ Adaptée aux intégrales bidimensionnelles.
- Plus complexe.

Simpson

+ Simple à mettre en œuvre.
+ Bonne précision.
- Limité aux formes simples.

Monte Carlo

+ Grande flexibilité.
+ Adaptée aux problèmes complexes.
- Précision dépend du nombre d'échantillons.

Outils de paralélisation

1. Les différentes méthodes d'intégration
   - Méthode de Simpson
   - Méthode de Gauss 2D
   - Méthode de Runge-Kutta
   - Méthode de Monte-Carlo
   - Comparaison des méthodes

2. Outils de parallélisation
   - Open MP
   - MPI
   - CUDA

3. Résultats
   - OpenMP
   - MPI
   - CUDA

Open MP

- **Intégration de Simpson**
  - Boucles `for` parallélisées avec `#pragma omp parallel for reduction(+:integral)` pour les deux versions.
  - Pour `compositeSimpsons_3_8`, utilisation d'un pas de 3 pour les indices de boucle, améliorant l'efficacité de la méthode.
- **Intégration de Gauss 2D**
  - Parallélisation de la génération de points et de poids avec `#pragma omp parallel for collapse(2)`.
  - L'utilisation de `collapse(2)` exploite le parallélisme dans les deux boucles imbriquées, améliorant l'efficacité.
- **Runge-Kutta**
  - Boucle temporelle parallélisée avec `#pragma omp parallel for`.
  - Copie temporaire des données (`std::vector<double> tempU(u)`) utilisée pour chaque thread, évitant les dépendances de données et optimisant les calculs.
- **Intégration de Monte Carlo**
  - La boucle de génération de points aléatoires est parallélisée avec `#pragma omp parallel for reduction(+:total)`.
  - Chaque thread utilise une graine différente (`rd() + thread_id`) pour éviter les séquences aléatoires identiques.

MPI

- **Intégration Simpson**
  - Division équitable des points entre les processus.
  - Utilisation de MPI_Reduce pour obtenir le résultat global.
- **Intégration Gauss 2D**
  - Parallélisation de la génération de points et de poids.
  - Utilisation de MPI_Sendrecv pour l'échange des bords entre les processus.
- **Runge-Kutta**
  - Division des données spatiales entre les processus.
  - Utilisation de MPI_Gather pour collecter les résultats.
- **Intégration Monte Carlo**
  - Division équitable des points entre les processus.
  - Utilisation de MPI_Reduce pour obtenir le résultat global.

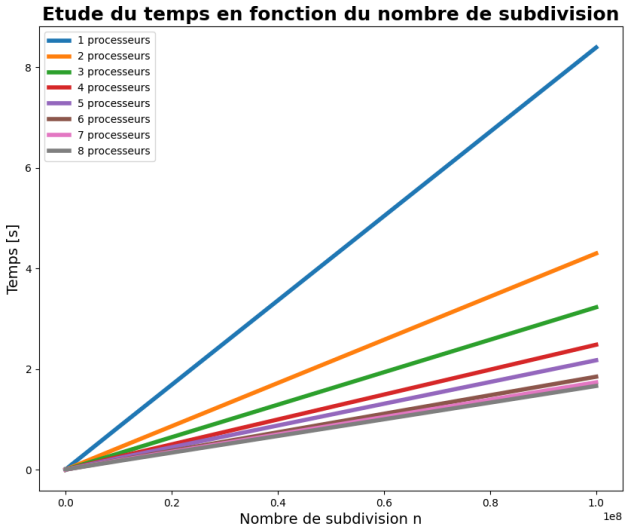
CUDA

Dans chaque code la méthode à son propre kernel CUDA, les autres fonctions sont utilisés sur le device et pour gérer la mémoire on utilise cudaMalloc et cudamemcpy.

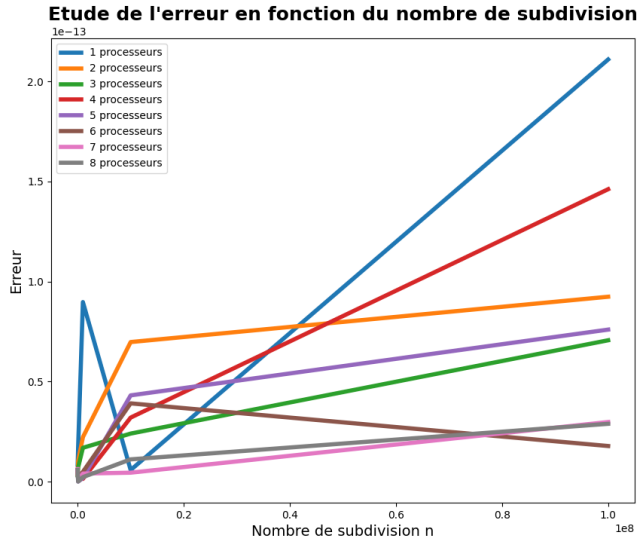
- **Intégration Simpson**
  - Division équitable des points entre les blocks.
  - Nombre de block maximal disponible sur mon GPU : 8584 et 64 threads par block.
- **Intégration Gauss 2D**
  - Parallélisation de la génération de points et de poids.
  - Nombre de block en fonction du nombre de points et des poids avec le nombre maximal de threads par block : 1024.
- **Runge-Kutta**
  - Division des données spatiales entre les blocks.
  - Nombre de block en fonction de la discrétisation et 64 threads par block.
- **Intégration Monte Carlo**
  - Division équitable des points entre les blocks.
  - Nombre de block maximal disponible sur mon GPU : 8584 et 64 threads par block.

Résultats

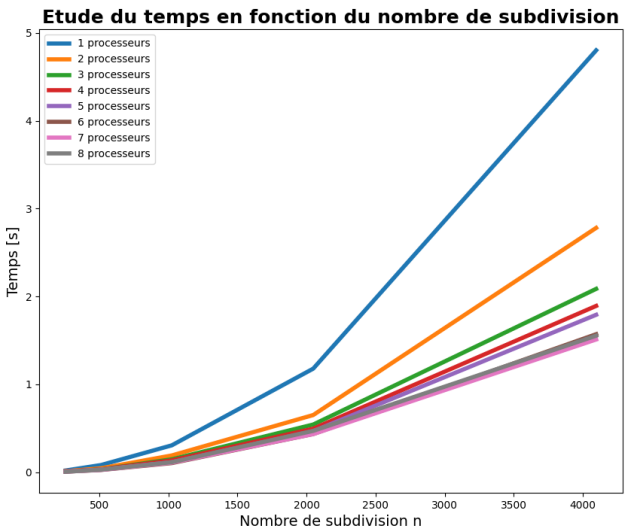
Résultats pour OpenMP



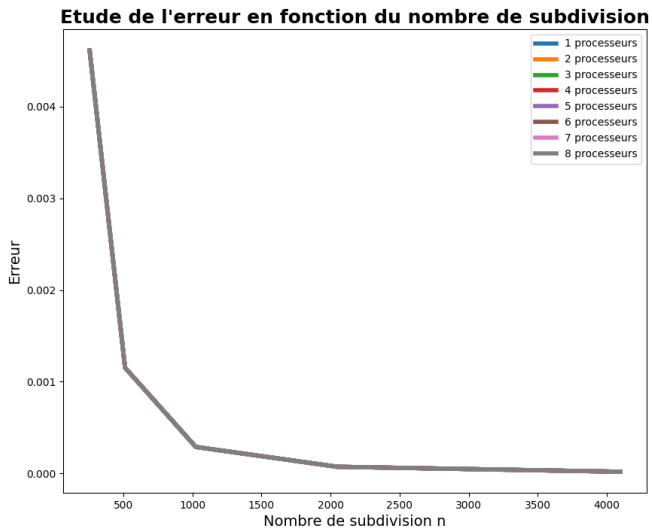
Simpson Temps



Simpson Erreur

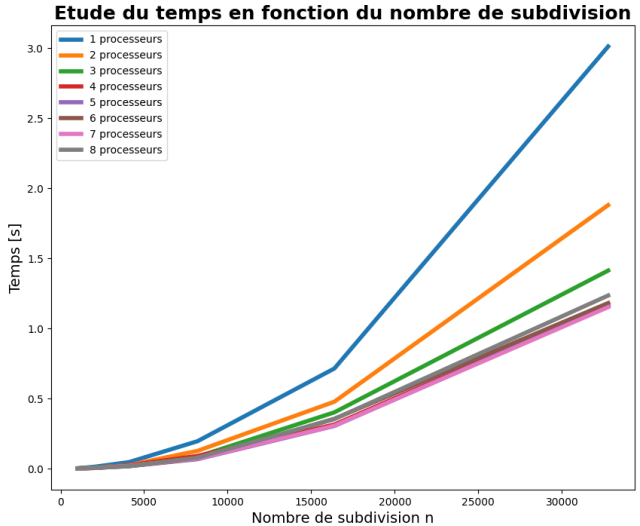


Gauss 2D Temps

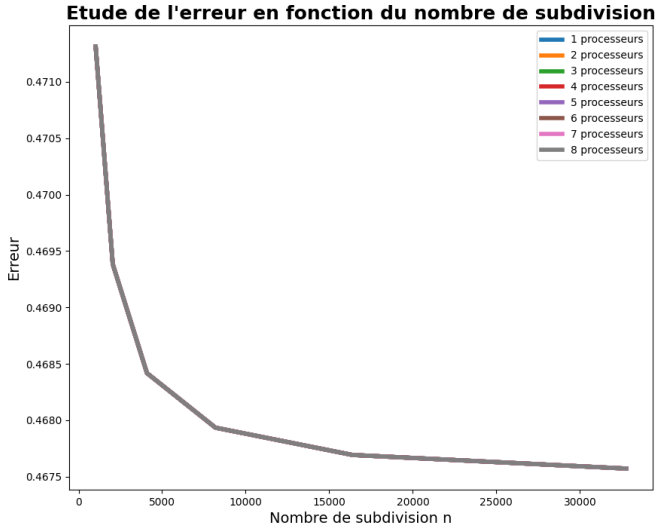


Gauss 2D Erreur

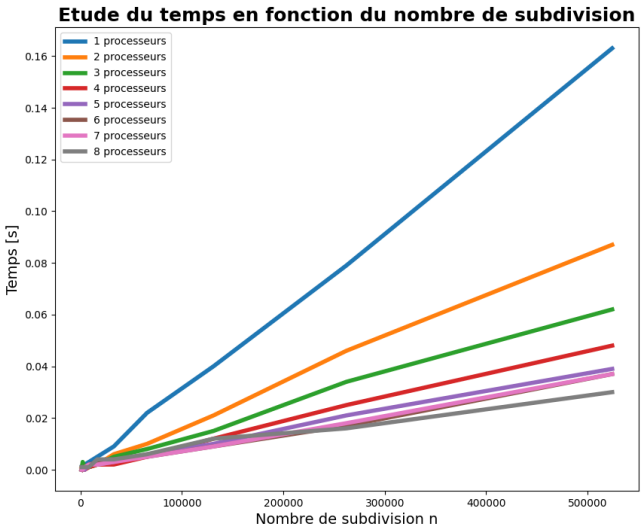
OpenMP



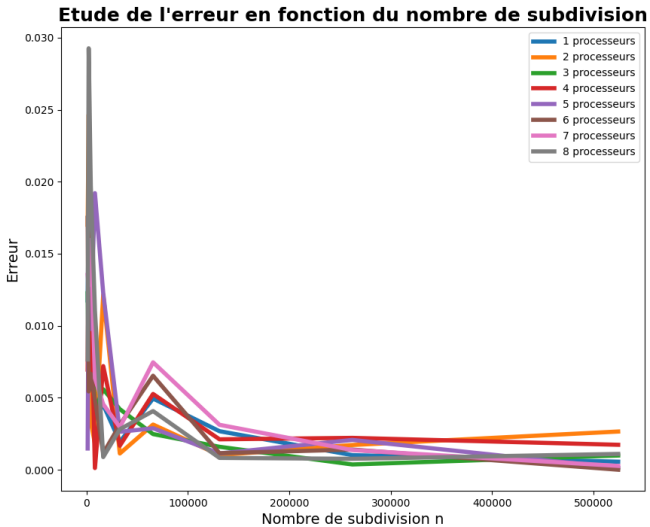
Runge Kutta Temps



Runge Kutta Erreur

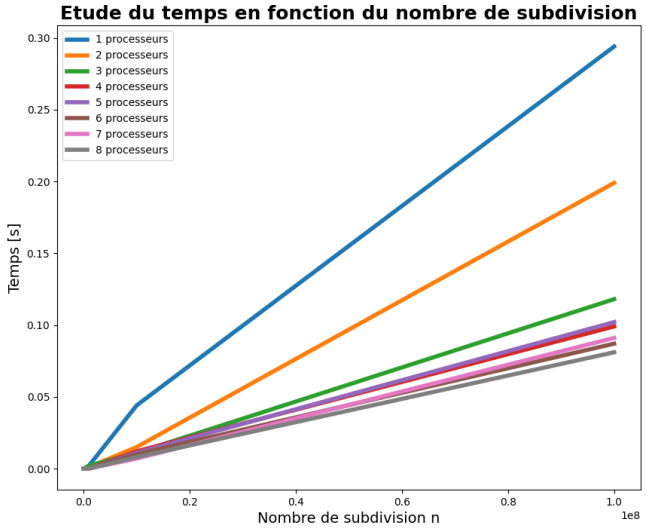


Monte Carlo Temps

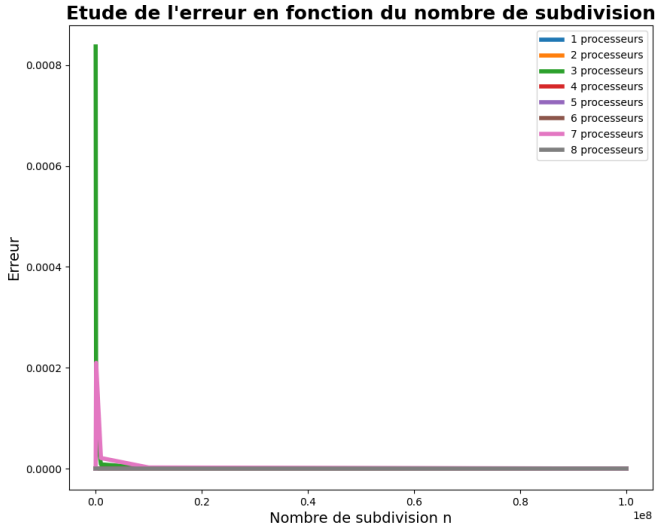


Monte Carlo Erreur

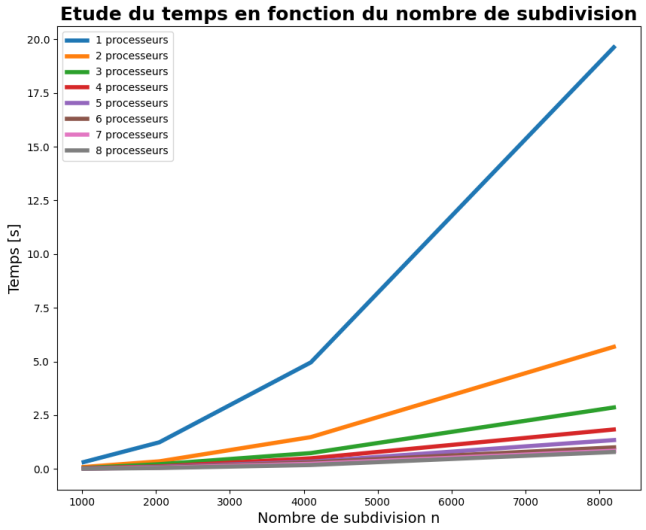
Résultats pour MPI



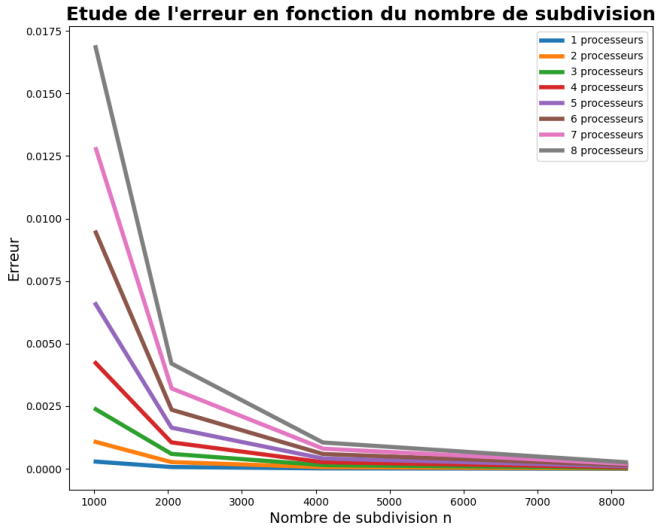
Simpson Temps



Simpson Erreur

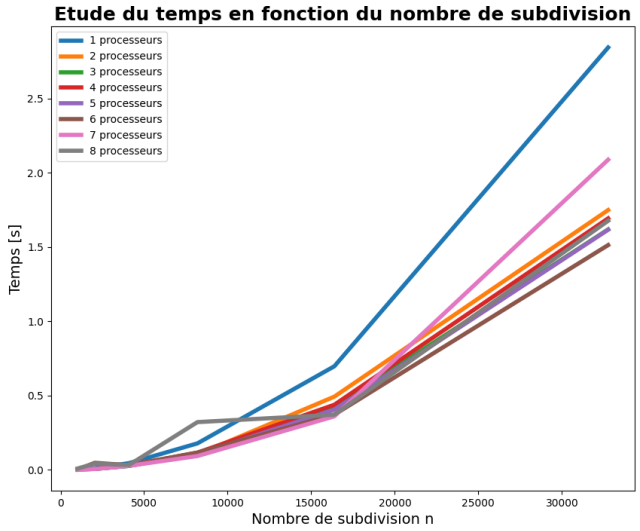


Gauss 2D Temps

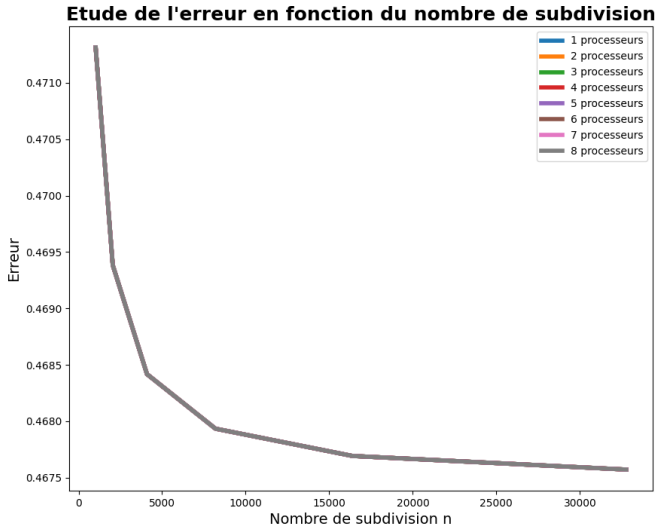


Gauss 2D Erreur

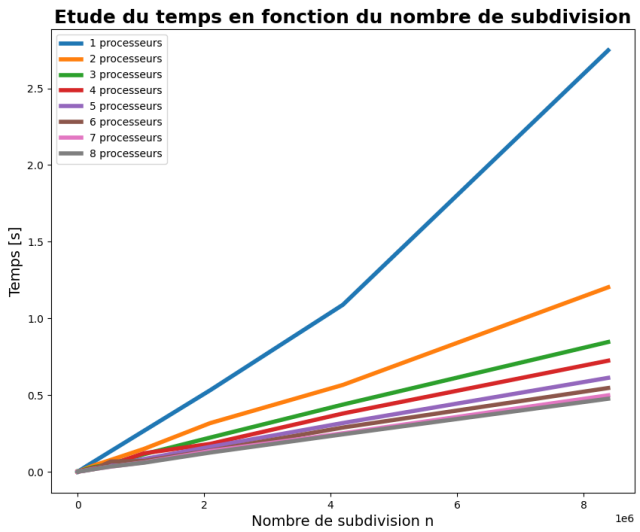
MPI



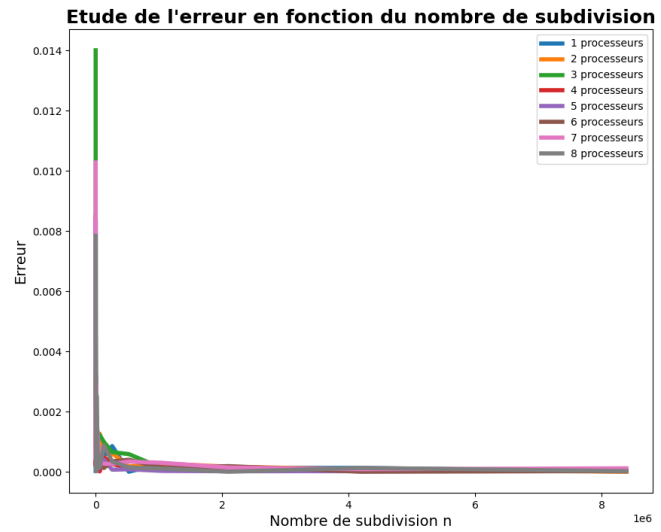
Runge Kutta Temps



Runge Kutta Erreur

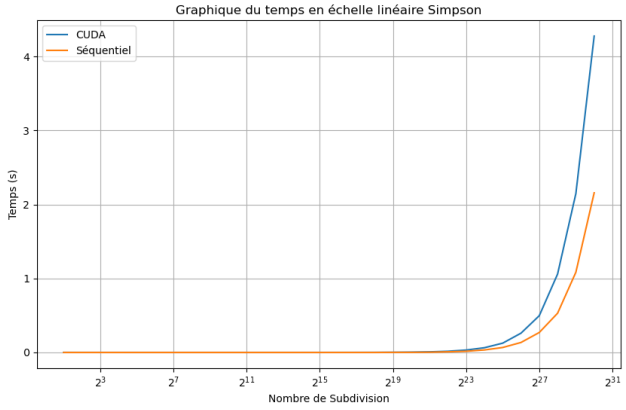


Monte Carlo Temps

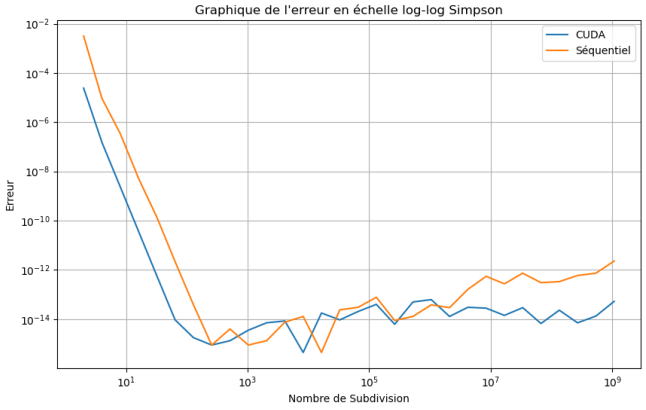


Monte Carlo Erreur

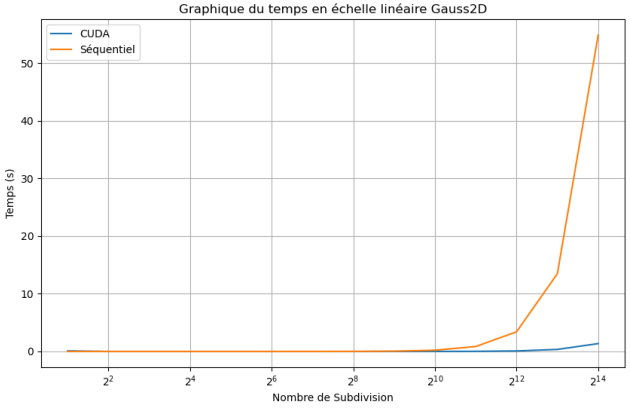
Résultats pour CUDA



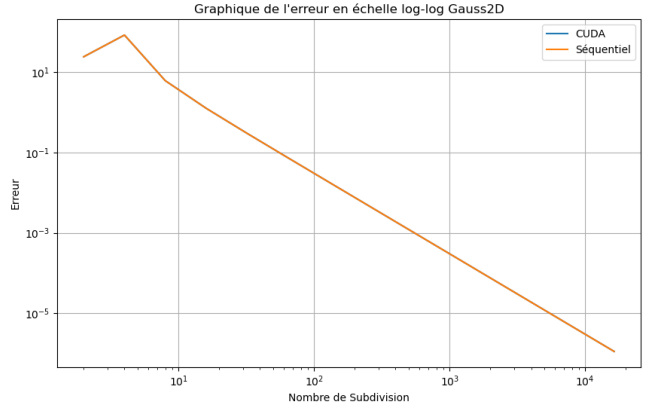
Simpson Temps



Simpson Erreur

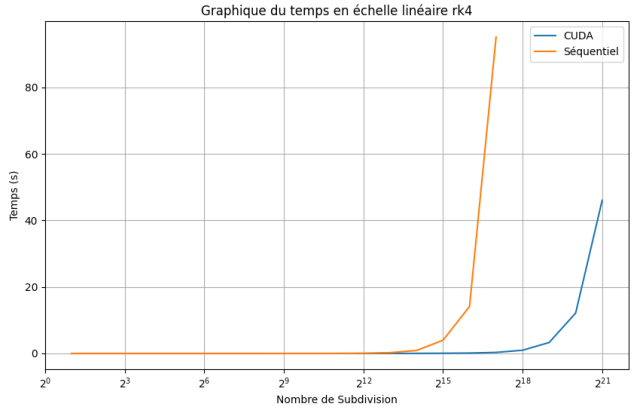


Gauss2D Temps

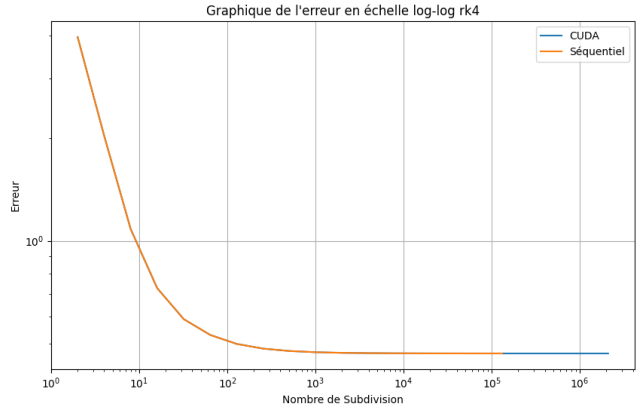


Gauss2D Erreur

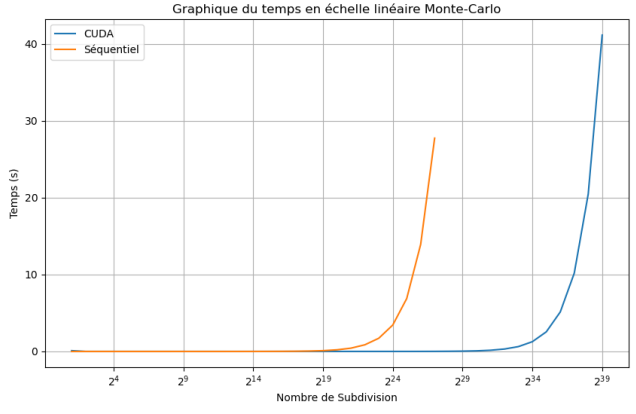
CUDA



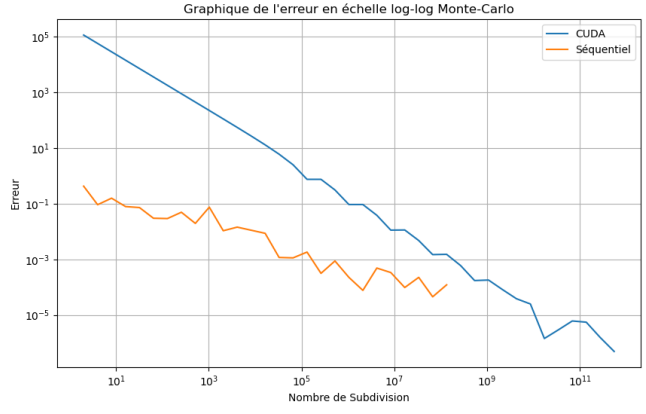
RK4 Temps



RK4 Erreur



Monte Carlo Temps



Monte Carlo Erreur