

Rapport de Projet P1RV

Optimisation et Parallélisation du Pricing d'Options Américaines

Méthode de Monte Carlo Longstaff-Schwartz sur CPU et GPU

Auteurs :

Florian BARBE
Narjisse EL MANSSOURI

École Centrale de Nantes
Année Universitaire 2025 – 2026

Table des matières

1	Introduction	3
2	Cadre de Modélisation	3
2.1	Problème d'arrêt optimal et dynamique	3
2.2	Algorithme de Longstaff–Schwartz (LSMC)	3
3	Travail réalisé : implémentation et méthodologie	5
3.1	Environnement de développement	5
3.2	Implémentation CPU séquentielle	5
3.3	Parallélisation CPU avec OpenMP	5
3.4	Accélération GPU avec CUDA	6
3.5	Méthodes de Différences Finies (FDM) pour comparaison	6
3.6	Structure du code et organisation des fichiers	7
4	Résultats et analyse des performances	7
4.1	Configuration matérielle	7
4.2	Première approche : comparaison des méthodes de parallélisme	7
4.3	Pour aller plus loin : analyse de l'impact de la base de régression	8
4.4	Validation numérique	8
4.5	Performances CPU séquentiel	9
4.6	Accélération par parallélisation CPU avec OpenMP	9
4.7	Accélération GPU avec CUDA	9
4.8	Comparaison globale des architectures	10
4.9	Discussion sur les limites du parallélisme	10
5	Analyse Critique et Limitations	10
5.1	Organisation du projet et évolution des dépôts	11
5.2	Échec de l'intégration CUDA dans Visual Studio	11
5.3	Complexité de l'exercice anticipé	11
5.4	Choix, stabilité et extensions de la régression	11
5.5	Limites de la parallélisation CPU	13
5.6	Spécificités et contraintes de l'implémentation GPU	13
5.7	Compromis précision–performance	13
6	Perspectives et améliorations possibles	13
6.1	Améliorations algorithmiques	14
6.2	Extensions du modèle financier	14
6.3	Optimisations CPU avancées	14
6.4	Optimisation et généralisation de l'implémentation GPU	14
6.5	Perspectives méthodologiques	15
7	Organisation du travail	15
7.1	Découpage du projet	15
7.2	Organisation du développement et itérations	16
7.3	Répartition des tâches et binôme	16
7.4	Outils et environnement collaboratif	17
7.5	Gestion du temps et avancement	17
8	Conclusion	17
A	Résolution de l'EDS du GBM	19
B	Compléments de Performance OpenMP	19

C Rappels de Probabilités et propriétés du Mouvement Brownien	20
C.1 Définition et propriétés du Mouvement Brownien	20
C.2 Propriétés des incréments	20
C.3 Loi des Grands Nombres et Monte Carlo	20
D Notations et Conventions	21

1 Introduction

Ce rapport synthétise les travaux d'implémentation et d'optimisation de l'algorithme de Longstaff–Schwartz (LSMC) pour la valorisation d'options américaines. Ce projet explore l'impact des architectures de calcul sur la performance d'une méthode de Monte Carlo avec régression, intrinsèquement coûteuse en ressources.

L'objectif est d'évaluer les gains de performance offerts par le parallélisme sur trois architectures distinctes :

- **CPU Séquentiel** : référence fonctionnelle.
- **CPU Multi-cœurs (OpenMP)** : exploitation du parallélisme à mémoire partagée.
- **GPU (CUDA)** : exploitation du parallélisme massif de données.

Ce document détaille la formulation mathématique, les choix d'implémentation algorithmique, et fournit une analyse critique des performances comparées (speedup, efficacité) et des limitations structurelles identifiées.

2 Cadre de Modélisation

2.1 Problème d'arrêt optimal et dynamique

La valeur d'une option américaine de vente (Put) de maturité T et de strike K est donnée, sous la mesure risque-neutre \mathbb{Q} , par le problème d'arrêt optimal :

$$V_0 = \sup_{\tau \in \mathcal{T}_{0,T}} \mathbb{E}^{\mathbb{Q}} [e^{-r\tau} (K - S_{\tau})^+]$$

où le sous-jacent $(S_t)_{t \geq 0}$ suit un Mouvement Brownien Géométrique (GBM) régi par l'EDS :

$$dS_t = rS_t dt + \sigma S_t dW_t^{\mathbb{Q}}$$

La solution explicite et sa discréétisation exacte sur une grille temporelle $\{t_0, \dots, t_N\}$ sont données par (voir Annexe A pour la dérivation) :

$$S_{t_{n+1}} = S_{t_n} \exp \left(\left(r - \frac{\sigma^2}{2} \right) \Delta t + \sigma \sqrt{\Delta t} Z_n \right), \quad Z_n \sim \mathcal{N}(0, 1)$$

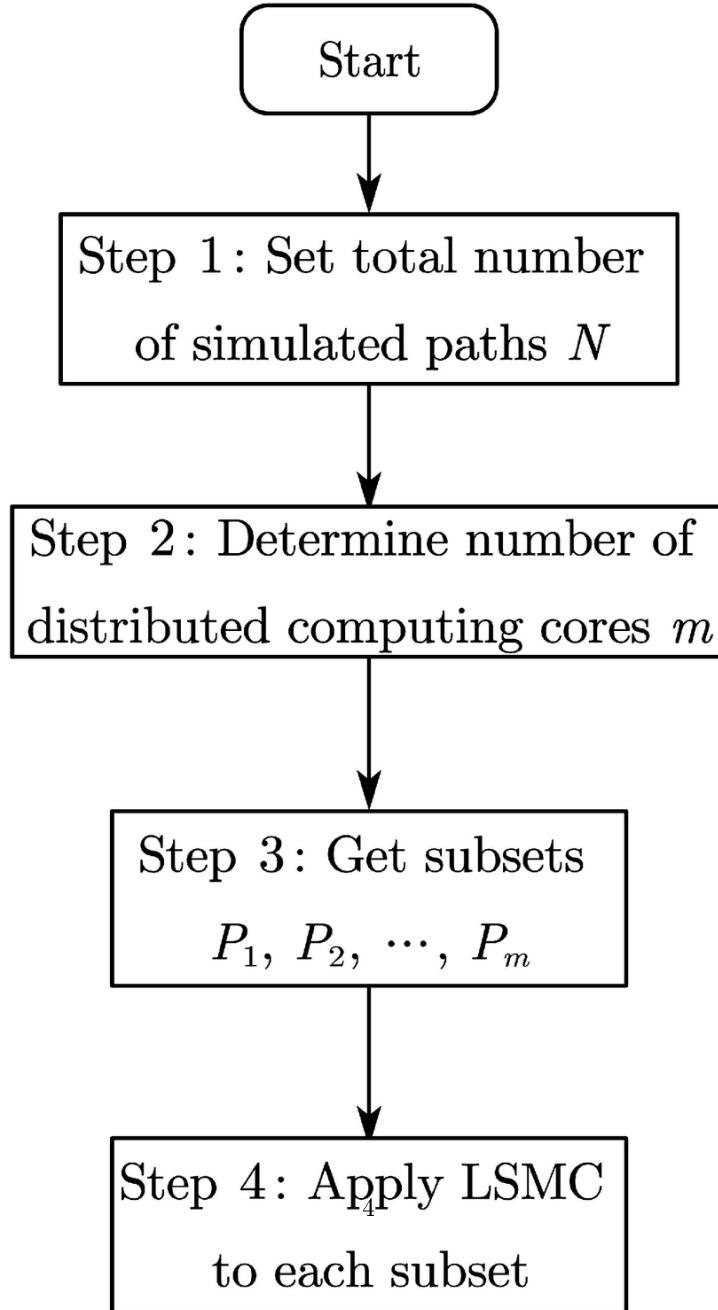
2.2 Algorithme de Longstaff–Schwartz (LSMC)

L'algorithme LSMC [1] résout ce problème par *backward induction*, en approximant l'espérance conditionnelle de continuation par une régression sur une base de fonctions.

L'algorithme 1 formalise cette procédure.

Algorithm 1 Algorithme de Longstaff–Schwartz (LSMC)

```
1: Simuler  $N_{\text{paths}}$  trajectoires  $(S_{t_n}^{(i)})_n$ .  
2: Initialiser  $V_{t_N}^{(i)} = \Phi(S_{t_N}^{(i)})$ .  
3: for  $n = N - 1$  down to 1 do  
4:    $Y^{(i)} = e^{-r\Delta t_n} V_{t_{n+1}}^{(i)}$ .  
5:   Régression  $Y^{(i)} \approx \hat{C}(t_n, S_{t_n}^{(i)})$ .  
6:   for chaque trajectoire  $i$  do  
7:     if  $\Phi(S^{(i)}) \geq \hat{C}(S^{(i)})$  then  
8:        $V^{(i)} \leftarrow \Phi(S^{(i)})$   
9:     else  
10:       $V^{(i)} \leftarrow Y^{(i)}$   
11:    end if  
12:   end for  
13: end for  
14:  $V_0 = \text{mean}(e^{-rt_1} V_{t_1}^{(i)})$ .
```



Synthèse : La structure du LSMC est intrinsèquement hybride. Si la phase de simulation est un candidat idéal pour le parallélisme massif ("embarrassingly parallel"), la phase de backward induction impose une synchronisation globale à chaque pas de temps, limitant le potentiel d'accélération sur GPU (loi d'Amdahl).

3 Travail réalisé : implémentation et méthodologie

Cette section décrit les choix techniques et méthodologiques retenus pour l'implémentation de l'algorithme de Longstaff-Schwartz, ainsi que les stratégies de parallélisation mises en œuvre sur CPU et GPU. L'objectif est double : assurer la validité numérique des résultats tout en évaluant l'impact des architectures de calcul sur les performances.

3.1 Environnement de développement

L'ensemble du projet a été développé en C++, avec une attention particulière portée aux performances et à la gestion mémoire. Les principaux outils et technologies utilisés sont :

- compilateur g++ compatible C++17 ;
- bibliothèque OpenMP pour la parallélisation CPU ;
- CUDA C++ pour l'implémentation GPU ;
- générateurs pseudo-aléatoires indépendants par thread ;
- système de build basé sur CMake.

Les calculs ont été réalisés sur une machine équipée d'un processeur multi-cœurs et d'un GPU NVIDIA compatible CUDA. Les expériences ont été conduites en privilégiant la reproductibilité et la stabilité numérique.

3.2 Implémentation CPU séquentielle

Une première version séquentielle de l'algorithme LSMC a été implémentée afin de servir de référence fonctionnelle et de point de comparaison en termes de performance.

Cette version suit strictement les étapes théoriques :

- simulation des trajectoires du sous-jacent selon un mouvement brownien géométrique ;
- calcul des payoffs à chaque date ;
- application de la backward induction ;
- estimation de la valeur de continuation par régression polynomiale ;
- calcul de la moyenne finale des cashflows actualisés.

L'implémentation séquentielle permet de valider la cohérence des résultats numériques et de mesurer le coût de calcul intrinsèque de l'algorithme avant toute parallélisation.

3.3 Parallélisation CPU avec OpenMP

La version parallèle CPU repose sur l'observation que la majorité des calculs dans l'algorithme LSMC sont indépendants par trajectoire. Cette propriété est exploitée à l'aide d'OpenMP.

Les sections parallélisées sont notamment :

- la génération des trajectoires du sous-jacent ;
- le calcul des payoffs ;
- l'accumulation des termes des équations normales pour la régression ;

- la mise à jour des cashflows lors de la backward induction ;
- le calcul de la moyenne finale.

Les réductions OpenMP sont utilisées pour agréger efficacement les contributions des différentes trajectoires, tout en garantissant l'absence de conditions de course¹. Le schéma `schedule(static)` est privilégié afin d'assurer une répartition homogène de la charge de travail et un bon comportement mémoire.

Cette parallélisation permet d'exploiter efficacement les coeurs du processeur, mais reste limitée par la bande passante mémoire et la nature statistique de l'algorithme.

3.4 Accélération GPU avec CUDA

Une version accélérée sur GPU a été développée afin d'exploiter le parallélisme massif offert par les architectures CUDA, une approche explorée dans des travaux similaires [4, 5]. Le GPU est particulièrement adapté à la simulation Monte Carlo, chaque trajectoire pouvant être associée à un thread indépendant.

Les principales étapes déportées sur le GPU sont :

- la simulation des trajectoires du mouvement brownien géométrique ;
- le calcul des payoffs ;
- certaines phases de réduction nécessaires à la régression.

La backward induction impose cependant une dépendance temporelle forte entre les dates successives, ce qui limite la parallélisation complète de l'algorithme. L'approche retenue consiste donc à paralléliser intensivement les calculs spatiaux (trajectoires) tout en conservant une synchronisation globale entre les dates.

Une attention particulière est portée à l'organisation mémoire des données afin de garantir des accès coalescents² et de limiter les transferts entre l'hôte (CPU) et le périphérique (GPU).

Cette implémentation permet d'évaluer concrètement les gains de performance apportés par le GPU et de mettre en évidence les limites structurelles du LSMC dans un contexte massivement parallèle.

3.5 Méthodes de Différences Finies (FDM) pour comparaison

Afin de valider nos résultats Monte Carlo et de disposer d'un point de comparaison déterministe performant, nous avons également implémenté des solveurs basés sur les différences finies (FDM) pour l'équation de Black-Scholes-Merton (PDE) [2]. Bien que ces méthodes soient limitées en dimension (difficiles à étendre au-delà de 2 ou 3 sous-jacents), elles sont extrêmement efficaces pour les options vanilles et américaines sur un seul sous-jacent.

Trois schémas numériques ont été implémentés dans le fichier `fdm.cpp` :

- Le schéma de **Runge-Kutta 4 (RK4)** a été choisi comme méthode de référence ("baseline") pour sa haute précision ($O(\Delta t^4)$). Bien que plus coûteux en calcul que les méthodes d'Euler, il fournit une valeur quasi-exacte pour valider les résultats Monte Carlo.
- **Euler Implicit** : Schéma inconditionnellement stable, nécessitant la résolution d'un système linéaire tridiagonal à chaque pas de temps (algorithme de Thomas³).
- **Euler Explicit** : Schéma simple et rapide, mais conditionnellement stable (nécessite un pas de temps suffisamment petit pour éviter les instabilités numériques).

Ces méthodes nous ont servi de "vérité terrain" pour vérifier la convergence de nos prix LSMC.

¹Une *race condition* (condition de course) survient lorsque le résultat d'un programme dépend de l'ordre d'exécution imprévisible de threads accédant simultanément à des données partagées en écriture.

²L'accès coalescent désigne le regroupement par le matériel de plusieurs requêtes mémoire provenant de threads voisins en une seule transaction physique, optimisant ainsi l'utilisation de la bande passante.

³L'algorithme de Thomas, également appelé TDMA (TriDiagonal Matrix Algorithm), est une forme simplifiée de l'élimination de Gauss optimisée pour les systèmes tridiagonaux. Sa complexité est $(n)\text{contre}(n^3)$ pour l'élimination de Gauss générale.

3.6 Structure du code et organisation des fichiers

Le projet est organisé de manière modulaire pour séparer les responsabilités (modélisation, calcul, utilitaires). Voici une description de l’arborescence :

- `src/` (dossier racine P1RV_CUDA/) :
 - `lsmc.cu` / `lsmc.cpp` : Cœur de l’algorithme Longstaff-Schwartz. La version `.cu` contient les kernels CUDA pour le GPU.
 - `gbm.cu` : Simulation du Mouvement Brownien Géométrique.
 - `fdm.cpp` : Implémentation des méthodes de différences finies (solvers PDE).
 - `main.cu` : Point d’entrée, gestion des arguments et lancement des benchmarks.
- `Utils/` :
 - `csv_writer.hpp` : Utilitaires pour l’export des résultats.
 - Scripts Python : Pour l’interface utilisateur et la visualisation.

4 Résultats et analyse des performances

4.1 Configuration matérielle

Tous les benchmarks présentés dans cette section ont été réalisés sur une machine équipée d’un processeur **Intel® Core™ i7-13620H** (13ème génération) et d’une carte graphique **NVIDIA GeForce RTX 4060**. Cette dernière, basée sur l’architecture Ada Lovelace, dispose de **3072 cœurs CUDA** et de **8 Go de mémoire GDDR6**, offrant une puissance de calcul théorique adaptée aux simulations massivement parallèles.

Cette section présente les résultats obtenus lors de l’exécution de l’algorithme de Longstaff–Schwartz selon les différentes architectures étudiées : CPU séquentiel, CPU parallélisé avec OpenMP et GPU via CUDA. L’analyse porte à la fois sur la validité numérique des résultats et sur les performances de calcul observées.

4.2 Première approche : comparaison des méthodes de parallélisme

Les performances ont été évaluées sur un ensemble de simulations Monte Carlo pour une option de type put américain. Nous avons comparé les temps d’exécution et la précision des prix obtenus par nos trois implémentations (CPU Séquentiel, OpenMP, GPU CUDA) ainsi que par les méthodes de différences finies.

Les résultats ci-dessous ont été obtenus sur une machine équipée d’un GPU NVIDIA.

Mode	Pas (N)	Trajectoires (M)	Prix (€)	Temps (ms)	Écart / FDM
FDM Implicit	1000	-	6.067	0.67	Ref
FDM Explicit	1000	-	6.079	60.18	+0.012
FDM RK4	1000	-	6.079	231.88	+0.012
CPU Séquentiel	50	100,000	6.057	559.91	-0.010
OpenMP	50	100,000	6.057	540.23	-0.010
GPU CUDA	50	100,000	6.070	41.96	+0.003
CPU Séquentiel	50	1,000,000	6.059	6914.19	-0.008
OpenMP	50	1,000,000	6.059	6562.99	-0.008
GPU CUDA	50	1,000,000	6.047	455.63	-0.020
CPU Séquentiel	50	5,000,000	6.057	35352.25	-0.010

Table 1: Comparaison des temps de calcul et précision pour un Put Américain ($S_0 = 100, K = 100, r = 0.05, \sigma = 0.2, T = 1$). (Matériel : i7-13620H + RTX 4060)

Analyse des résultats :

- **Précision** : Tous les modes LSMC convergent vers un prix très proche de la référence FDM (6.067). Les écarts observés sont de l'ordre du centime, ce qui est acceptable pour une méthode de Monte Carlo avec ces paramètres.
- **Performance GPU** : Le GPU démontre une accélération spectaculaire. Pour 1 million de trajectoires, le calcul prend environ **455 ms** sur GPU contre près de **7 secondes** (6914 ms) sur CPU séquentiel, soit un facteur d'accélération (speedup) d'environ $\times 15$.
- **Comparaison FDM** : Les méthodes de différences finies (surtout l'implicite) sont extrêmement rapides ($< 1\text{ms}$) pour ce problème 1D. Cela confirme que pour des options simples, les PDE restent supérieures. Cependant, l'intérêt du LSMC (et donc de notre implémentation GPU) réside dans sa capacité à traiter des problèmes de plus haute dimension où les méthodes de grille échouent.

4.3 Pour aller plus loin : analyse de l'impact de la base de régression

Les benchmarks réalisés ("boosted", voir tableau 2) révèlent que l'augmentation du degré de la base est bénéfique. La base Cubique permet d'atteindre un prix de ≈ 6.06 , nettement plus proche de la référence théorique (≈ 6.08) que les bases quadratiques/monomiales (≈ 5.58 dans cette configuration CPU non-optimisée).

Cette amélioration de précision justifie le léger surcoût calculatoire lié à la résolution de systèmes linéaires 4×4 , désormais gérée par notre solveur de Gauss générique sur GPU.

Base	Architecture	Trajettoires	Prix (€)	Temps (ms)	Throughput (ops/s)
N = 100,000					
Monomiale	CPU Séquentiel	100k	5.586	261.96	19.1 M
Monomiale	CPU OpenMP	100k	5.586	274.97	18.2 M
Monomiale	GPU	100k	6.070	45.46	109.9 M
Hermite	CPU Séquentiel	100k	5.586	265.94	18.8 M
Hermite	CPU OpenMP	100k	5.586	266.81	18.7 M
Hermite	GPU	100k	6.070	43.40	115.2 M
Laguerre	CPU Séquentiel	100k	5.586	268.81	18.6 M
Laguerre	CPU OpenMP	100k	5.586	265.83	18.8 M
Laguerre	GPU	100k	6.070	35.44	141.1 M
Chebyshев	CPU Séquentiel	100k	5.586	263.63	18.9 M
Chebyshев	CPU OpenMP	100k	5.586	265.64	18.8 M
Chebyshev	GPU	100k	6.070	41.66	120.0 M
N = 1,000,000					
Monomiale	CPU Séquentiel	1M	5.565	2653.88	18.8 M
Monomiale	CPU OpenMP	1M	5.565	2688.26	18.6 M
Monomiale	GPU	1M	6.047	334.53	149.5 M
Hermite	CPU Séquentiel	1M	5.565	2672.32	18.7 M
Hermite	CPU OpenMP	1M	5.565	2704.67	18.5 M
Hermite	GPU	1M	6.047	594.25	84.1 M
Laguerre	CPU Séquentiel	1M	5.565	2710.66	18.4 M
Laguerre	CPU OpenMP	1M	5.565	2841.14	17.6 M
Laguerre	GPU	1M	6.047	649.80	76.9 M
Chebyshév	CPU Séquentiel	1M	5.565	2633.49	18.9 M
Chebyshév	CPU OpenMP	1M	5.565	2638.32	18.9 M
Chebyshév	GPU	1M	6.047	626.32	79.8 M
Cubique	CPU Séquentiel	1M	5.565	2677.14	18.7 M
Cubique	CPU OpenMP	1M	5.565	2628.52	19.0 M
Cubique	GPU	1M	6.063	687.69	72.7 M

Table 2: Benchmarks "Boosted" complets : Impact du choix de la base et de l'architecture. (Matériel : i7-13620H + RTX 4060)

Les temps mesurés confirment l'efficacité de l'approche massivement parallèle pour la phase de simulation. Le goulot d'étranglement restant sur GPU est la régression séquentielle à chaque pas de temps.

4.4 Validation numérique

Avant toute analyse de performance, la cohérence numérique des différentes implantations a été vérifiée. Les prix obtenus avec :

- l'implémentation CPU séquentielle,
- l'implémentation CPU OpenMP,
- l'implémentation GPU CUDA,

sont compatibles entre eux à l'intérieur des intervalles d'erreur statistique attendus pour une méthode de Monte Carlo.

Lorsque le nombre de trajectoires augmente, la variance de l'estimateur décroît conformément au taux théorique $\mathcal{O}(1/\sqrt{N_{\text{paths}}})$, ce qui confirme la bonne implémentation de l'algorithme LSMC sur les trois architectures.

4.5 Performances CPU séquentiel

L'implémentation séquentielle sert de référence de performance. Le temps d'exécution croît linéairement avec le nombre de trajectoires simulées, ce qui est cohérent avec la complexité algorithmique du LSMC :

$$\mathcal{O}(N_{\text{paths}} \times N_{\text{steps}}).$$

Cette version met en évidence le caractère fortement coûteux des méthodes de Monte Carlo lorsque la précision recherchée impose un grand nombre de trajectoires. Elle justifie pleinement le recours à des techniques de parallélisation.

4.6 Accélération par parallélisation CPU avec OpenMP

La version OpenMP exploite le parallélisme multi-cœurs du processeur. Les gains observés sont significatifs pour les phases suivantes :

- simulation des trajectoires du sous-jacent ;
- calcul des payoffs ;
- accumulation des équations normales pour la régression ;
- mise à jour des cashflows.

Le facteur d'accélération obtenu est inférieur au nombre de cœurs disponibles, ce qui s'explique par :

- la bande passante mémoire limitée ;
- les accès concurrents aux structures de données partagées ;
- la présence de phases séquentielles incompressibles (notamment la progression temporelle de la backward induction).

Néanmoins, OpenMP permet une réduction notable du temps de calcul, tout en conservant une implémentation relativement simple et portable.

Synthèse : Le parallélisme OpenMP est efficace pour les phases *compute-bound* (simulations) mais satire rapidement la bande passante mémoire lors des opérations vectorielles massives, plafonnant l'accélération bien en deçà du nombre de cœurs physiques.

4.7 Accélération GPU avec CUDA

L'implémentation CUDA montre des gains de performance particulièrement importants pour les parties massivement parallèles de l'algorithme :

- simulation des trajectoires GBM ;
- calcul des payoffs ;
- certaines réductions statistiques.

Le GPU tire parti du parallélisme massif en assignant un thread par trajectoire, ce qui permet de traiter simultanément plusieurs centaines de milliers, voire millions de chemins.

Cependant, la backward induction impose une dépendance temporelle forte entre les dates successives. Cette contrainte limite la parallélisation complète de l'algorithme et réduit le gain théorique maximal.

- du nombre de trajectoires simulées ;
- de l'occupation effective du GPU ;
- de l'efficacité des réductions parallèles ;
- de la limitation des transferts mémoire CPU-GPU.

Synthèse : Le GPU excelle sur le parallélisme de données (trajectoires), offrant des speedups d'un ordre de grandeur, mais l'accélération globale reste structurellement bornée par la nature séquentielle temporelle de l'algorithme (loi d'Amdahl appliquée à la backward induction).

4.8 Comparaison globale des architectures

De manière synthétique :

- le CPU séquentiel fournit une référence simple mais peu performante ;
- OpenMP permet un gain modéré, limité par la bande passante mémoire ;
- CUDA offre les meilleures performances pour les grandes tailles de problèmes, en particulier lorsque le nombre de trajectoires est très élevé.

Le GPU s'avère donc particulièrement adapté aux méthodes de Monte Carlo à grande échelle, tandis que le CPU reste pertinent pour des tailles de problèmes plus modérées ou lorsque la simplicité d'implémentation est prioritaire.

4.9 Discussion sur les limites du parallélisme

Les résultats confirment que l'algorithme LSMC est naturellement bien adapté au parallélisme spatial, mais intrinsèquement limité par sa structure séquentielle dans le temps. En effet, l'algorithme repose sur une **induction arrière** (backward induction) : le calcul des valeurs à l'étape t dépend mathématiquement des résultats de l'étape $t + 1$ (nécessaires pour construire la variable cible de la régression).

Il est donc impossible de paralléliser le traitement des différents pas de temps pour une même option. Le GPU doit impérativement attendre la fin du calcul de l'étape $t + 1$ avant de commencer l'étape t , ce qui crée une barrière de synchronisation inévitable. L'accélération maximale est ainsi plafonnée par cette contrainte séquentielle, même avec un nombre infini de coeurs. La seule manière d'accroître davantage le parallélisme serait de traiter simultanément plusieurs options distinctes (batching).

Ces observations expliquent pourquoi les gains GPU, bien que très importants, ne peuvent pas être parfaitement linéaires avec la puissance de calcul.

Ces résultats mettent en évidence l'intérêt d'architectures hybrides CPU-GPU, ainsi que l'importance de choix d'implémentation fins (organisation mémoire, réductions efficaces, limitation des synchronisations) pour exploiter pleinement les capacités du matériel moderne.

Synthèse Globale : Si le GPU déverrouille la scalabilité spatiale du stock de trajectoires, la dimension temporelle reste séquentielle. L'optimum de performance réside dans un équilibre : assez de trajectoires pour saturer le GPU, mais pas au-delà des besoins de précision, car le coût de régression devient alors prépondérant.

5 Analyse Critique et Limitations

La réalisation de ce projet a mis en évidence plusieurs défis techniques et méthodologiques, dont l'analyse permet de mieux cerner les contraintes du calcul haute performance en finance.

5.1 Organisation du projet et évolution des dépôts

Le développement du projet s'est articulé autour de **deux dépôts Git distincts**, reflétant une évolution technique majeure :

1. **lsmc-openmp** (Octobre 2025) : Le premier dépôt a été consacré au développement de la logique métier de l'algorithme LSMC en C++ avec parallélisation OpenMP. Le système de build reposait sur des solutions **Visual Studio** (.sln, .vcxproj). Ce dépôt inclut également une interface graphique en Python (Streamlit/Matplotlib) pour la visualisation des trajectoires.
2. **CUDA-Implementation** (Janvier 2026) : Face aux difficultés d'intégration CUDA dans le premier dépôt, un second dépôt a été créé avec une architecture repensée autour de **CMake**. C'est dans ce dépôt que les fonctionnalités avancées (5 bases de régression, solveur générique, kernels optimisés) ont été développées et validées.

Cette organisation permet de conserver une **version CPU de référence** (premier dépôt) tout en disposant d'une **version HPC/GPU complète** (second dépôt).

5.2 Échec de l'intégration CUDA dans Visual Studio

L'historique des commits du premier dépôt témoigne des tentatives infructueuses d'intégration CUDA :

- "*Début intégration CUDA*" : ajout initial des fichiers .cu et configuration NVCC.
- "*On continue de debug CUDA car rien ne marche*" : erreurs de compilation persistantes.
- "*Suppression complète de tout le code lié à CUDA/GPU*" : abandon de l'approche.

Cause identifiée : Visual Studio peinait à gérer correctement la cohabitation entre le compilateur C++ standard (MSVC) et le compilateur CUDA (nvcc). Les conflits de flags de compilation, les incompatibilités d'architectures cibles et la gestion des dépendances rendaient la configuration instable.

Solution adoptée : La migration vers **CMake** a résolu ces problèmes. CMake intègre nativement le support CUDA via `enable_language(CUDA)` et permet une séparation propre entre le code hôte (.cpp) et le code device (.cu). Cette architecture a permis de finaliser l'implémentation GPU avec succès.

5.3 Complexité de l'exercice anticipé

La difficulté théorique majeure provient de la possibilité d'un exercice anticipé pour les options américaines. Cette caractéristique transforme le problème de valorisation en un problème d'arrêt optimal, nécessitant une estimation fiable de la valeur de continuation à chaque date. Une mauvaise compréhension de cette quantité conduit rapidement à des erreurs de logique dans la backward induction. Un soin particulier a donc été apporté à la séparation claire entre valeur d'exercice immédiat et valeur de continuation estimée.

5.4 Choix, stabilité et extensions de la régression

L'estimation de la valeur de continuation par régression constitue un point sensible de l'algorithme. Le choix des fonctions de base représente un compromis entre précision et stabilité numérique. Des bases trop simples introduisent un biais, tandis que des bases trop riches peuvent engendrer des instabilités ou un surcoût de calcul.

Étude initiale : Bases quadratiques. Dans un premier temps, une étude comparative a été menée entre la base canonique $(1, S, S^2)$ et la base de **polynômes d'Hermite de degré 2** $(1, S, S^2 - 1)$. Bien que les benchmarks montrent peu de différence en termes de précision pour ce problème spécifique, la base d'Hermite a été retenue par défaut pour ses meilleures propriétés théoriques de conditionnement (matrice $A^T A$).

Extensions et bases avancées. Afin d'affiner la capture de la valeur de continuation, nous avons étendu l'implémentation à trois autres familles de bases, souvent citées dans la littérature spécialisée [2] :

1. **Laguerre** : Polynômes orthogonaux sur $[0, \infty]$, historiquement suggérés par Longstaff et Schwartz [1] pour leur adaptation naturelle aux prix d'actifs positifs (pondération par exponentielle décroissante).
2. **Tchebychev** : Polynômes minimisant l'erreur d'interpolation sur $[-1, 1]$. Cette base a nécessité l'implémentation d'un kernel de réduction Min-Max sur GPU pour normaliser les prix à chaque pas de temps.
3. **Cubique** : Base monômiale enrichie au degré 3 ($1, S, S^2, S^3$).

La figure 2 illustre l'impact du choix de la base de régression sur la précision de l'estimation.

Figure 4 Accuracy comparison of LSMC methods using different basis functions: vanilla put options.

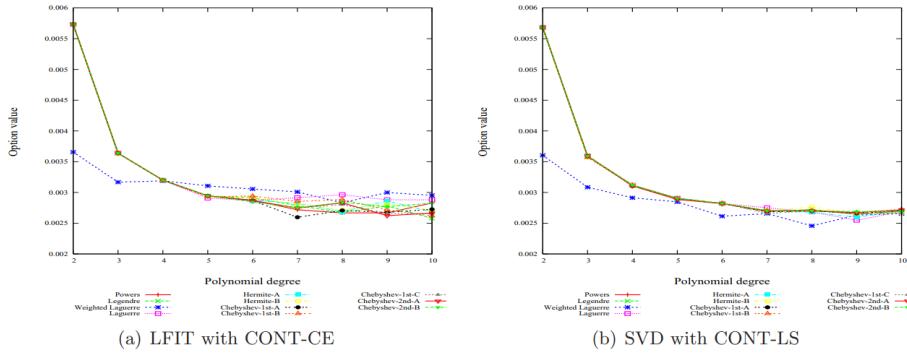


Figure 2: Comparaison de la précision selon les fonctions de base utilisées pour la régression LSMC (Source: [9]).

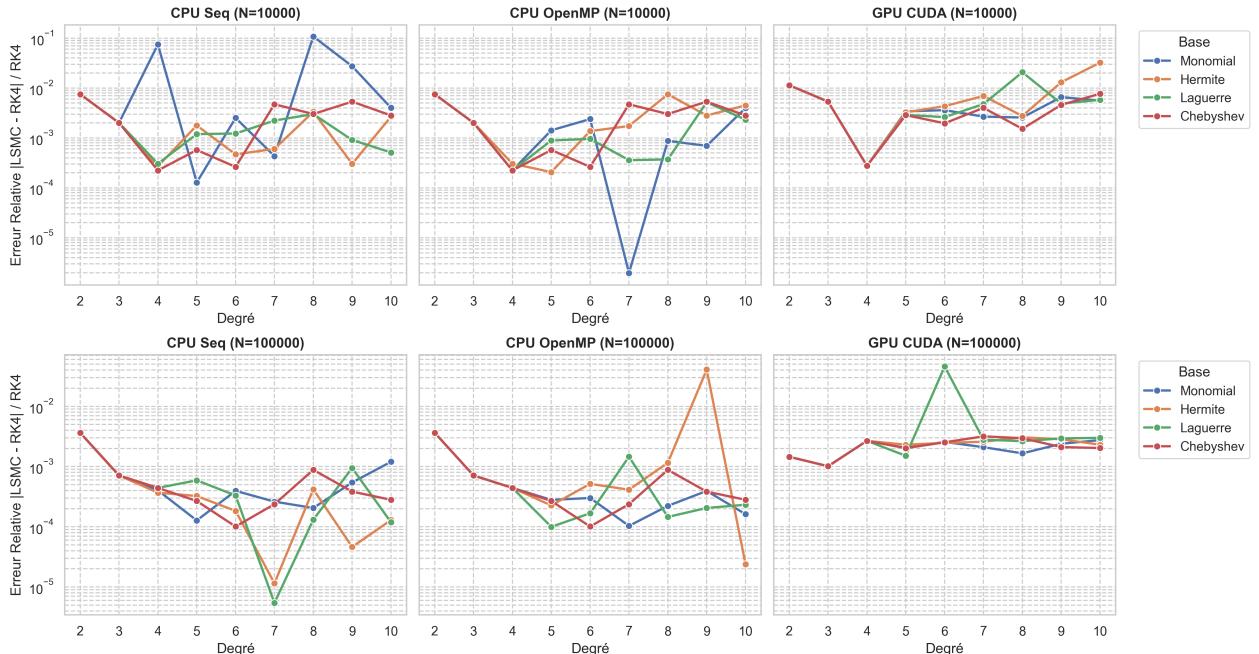


Figure 3: Comparaison de la convergence de l'erreur relative selon le nombre de trajectoires ($N = 10^4, 10^5$) et l'architecture.

Analyse de la convergence (Figure 3). Les résultats présentés en figure 3 montrent l'évolution de l'erreur relative par rapport au prix RK4 de référence. Cette dynamique peut être mise en perspective avec la Figure 2 (issue de [9]) qui illustre les attentes théoriques de convergence. Contrairement à l'intuition théorique qui suggère une amélioration monotone, les courbes observées ne présentent pas une décroissance régulière. Plusieurs facteurs peuvent expliquer ce comportement mitigé et "non concluant" sur ce cas test spécifique :

- **Phénomène de Runge :** L'utilisation de bases polynomiales de degré élevé sur des points non uniformément répartis (les trajectoires stochastiques) peut induire de fortes oscillations aux bords du domaine (valeurs extrêmes de S_t), dégradant la qualité globale de la régression [3].
- **Conditionnement de la matrice :** Pour des degrés élevés ($d > 5$), la matrice de Gram $A^T A$ devient mal conditionnée, en particulier pour la base Monomiale, ce qui amplifie les erreurs numériques lors de la résolution du système linéaire. Bien que les bases orthogonales (Laguerre, Hermite) soient censées atténuer ce problème, le bruit de Monte Carlo semble ici dominer les gains théoriques.
- **Compromis Biais-Variance :** Augmenter le degré réduit le biais de modélisation (capacité à fitter la "vraie" fonction de continuation) mais augmente la variance de l'estimateur, car le modèle capture le bruit statistique des trajectoires (sur-apprentissage).

Ces observations soulignent la difficulté pratique d'obtenir une convergence "parfaite" avec des méthodes de régression globale sur des degrés élevés sans un nombre de trajectoires extrêmement grand ($N \gg 10^5$).

5.5 Limites de la parallélisation CPU

La parallélisation sur CPU via OpenMP permet d'accélérer efficacement la simulation des trajectoires et le calcul des payoffs. Toutefois, les gains observés restent limités par la bande passante mémoire et la contention sur les caches partagés. En pratique, l'accélération n'est pas linéaire avec le nombre de coeurs, ce qui confirme le caractère fortement *memory-bound* de l'algorithme LSMC sur CPU.

5.6 Spécificités et contraintes de l'implémentation GPU

L'implémentation GPU avec CUDA a mis en évidence [7] un contraste marqué entre les différentes phases de l'algorithme. La simulation des trajectoires et le calcul des payoffs bénéficient pleinement du parallélisme massif, avec des accélérations importantes lorsque le nombre de trajectoires augmente.

Les résultats expérimentaux montrent un gain de performance croissant avec la taille du problème, atteignant des facteurs d'accélération supérieurs à 30 pour de grands nombres de trajectoires. En revanche, la backward induction introduit une dépendance temporelle forte entre les dates, ce qui limite la parallélisation complète et impose des synchronisations coûteuses.

Par ailleurs, une légère différence entre les prix CPU et GPU a été observée. Elle s'explique par la nature statistique de la méthode de Monte Carlo, les différences de générateurs pseudo-aléatoires et les effets d'arrondis en arithmétique flottante. Ces écarts restent toutefois compatibles avec la variance attendue de l'estimateur.

5.7 Compromis précision–performance

Enfin, ce projet a mis en évidence le compromis fondamental entre précision numérique et temps de calcul. L'augmentation du nombre de trajectoires améliore la convergence statistique, mais accroît fortement le coût de calcul, en particulier lors des phases de régression et de backward induction. Ce compromis a guidé le choix des paramètres expérimentaux et l'analyse comparative des architectures CPU et GPU.

6 Perspectives et améliorations possibles

Le travail réalisé dans le cadre de ce projet a permis de mettre en œuvre une version fonctionnelle et performante de l'algorithme de Longstaff–Schwartz sur différentes architectures de calcul. Plusieurs pistes

d'amélioration et d'extension peuvent toutefois être envisagées, tant sur le plan algorithmique que sur le plan des performances et des modèles financiers considérés.

6.1 Améliorations algorithmiques

Une première piste d'amélioration concerne le choix des fonctions de base utilisées pour l'approximation de la valeur de continuation. Dans ce projet, une base polynomiale simple de degré deux a été retenue pour des raisons de simplicité et de stabilité numérique. Il serait possible d'explorer :

- des bases polynomiales de degré plus élevé ;
- des polynômes orthogonaux (Laguerre, Hermite), souvent utilisés dans la littérature ;
- des bases adaptatives dépendant de la distribution du sous-jacent, comme suggéré dans des publications récentes [6].

Ces choix peuvent améliorer la précision de l'estimation de la valeur de continuation, au prix d'un coût de calcul plus élevé et d'un risque accru de sur-apprentissage.

Par ailleurs, l'utilisation de techniques de réduction de variance (antithetic variates, control variates, stratified sampling) pourrait significativement améliorer la convergence statistique de la méthode de Monte Carlo, en réduisant le nombre de trajectoires nécessaires pour atteindre une précision donnée.

6.2 Extensions du modèle financier

Le cadre retenu dans ce projet repose sur un mouvement brownien géométrique, modèle de référence mais relativement simplificateur. Plusieurs extensions naturelles peuvent être envisagées :

- introduction d'un taux de dividende continu ;
- prise en compte d'une volatilité locale ou stochastique (modèles de Heston, SABR) ;
- extension à des options multi-actifs ou dépendant de plusieurs sous-jacents ;
- valorisation de produits exotiques présentant des payoff path-dépendants.

L'algorithme de Longstaff–Schwartz conserve une grande flexibilité face à ces extensions, ce qui constitue l'un de ses principaux avantages par rapport aux méthodes analytiques.

6.3 Optimisations CPU avancées

Sur CPU, plusieurs optimisations supplémentaires pourraient être envisagées :

- vectorisation explicite via les instructions SIMD (AVX2, AVX-512) ;
- meilleure gestion de la localité mémoire pour réduire la pression sur la bande passante RAM ;
- utilisation de bibliothèques numériques optimisées pour les régressions linéaires.

Ces optimisations permettraient d'exploiter plus finement l'architecture matérielle, en particulier pour des tailles de problème intermédiaires où le GPU n'est pas toujours optimal.

6.4 Optimisation et généralisation de l'implémentation GPU

Du côté GPU, plusieurs améliorations sont envisageables :

- implémentation complète de la régression par moindres carrés directement sur GPU, sans synchronisation CPU ;
- utilisation de bibliothèques CUDA spécialisées (**cuBLAS**, **CUB**, **Thrust**) pour les réductions et opérations linéaires ;
- exploration de stratégies multi-GPU pour des simulations de très grande dimension.

Une telle approche permettrait de réduire encore les coûts de communication et d'atteindre des gains de performance plus proches du potentiel théorique du GPU.

6.5 Perspectives méthodologiques

Enfin, ce projet ouvre la voie à des comparaisons plus larges entre différentes approches numériques pour le pricing d'options américaines. Il serait intéressant de confronter les performances du LSMC à :

• .

	Méthodes par arbre	Méthodes itératives	Méthodes basées sur des simulations de Monte Carlo	Méthodes par transformée de Fourier
Compréhension	++	+	+	-
Scalabilité	-	+	++	+
Consommation mémoire	-	+	++	+
Complexité calcul	-	+	+	+
Convergence	+	+	-	++

Figure 7 Tableau dressant les atouts et contraintes des différentes méthodes.

Figure 4: Tableau comparatif des méthodes de pricing : arbres, itératives, Monte Carlo et transformée de Fourier [9].

On pourrait ainsi comparer ;

- des approches par équations aux dérivées partielles résolues numériquement ;
- des méthodes récentes basées sur l'apprentissage automatique (réseaux de neurones pour l'approximation de la valeur de continuation).

Ces perspectives permettraient d'évaluer plus finement les compromis entre précision, coût de calcul et flexibilité des différentes méthodes, dans un contexte de finance quantitative moderne.

7 Organisation du travail

Le projet P1RV a été mené sur l'ensemble du premier semestre selon une organisation progressive, combinant approfondissement théorique, développement logiciel itératif et analyse des performances. Le travail s'est appuyé sur une démarche incrémentale, avec des phases successives de conception, d'implémentation, de refactorisation et d'optimisation, comme en témoigne l'historique détaillé du dépôt Git.

7.1 Découpage du projet

Le développement du projet a été structuré autour des grandes étapes suivantes :

- étude du cadre théorique : options américaines, Monte Carlo, backward induction et algorithme de Longstaff–Schwartz ;

- implémentation d'une version CPU séquentielle servant de référence ;
- restructuration complète du code afin de séparer clairement simulation, régression et logique LSMC ;
- parallélisation CPU avec OpenMP ;
- ajout progressif d'un backend GPU CUDA expérimental ;
- mise en place d'outils d'export des résultats (CSV) et de visualisation ;
- campagnes de tests, mesures de performances et nettoyage final du code ;
- rédaction et structuration du rapport.

Ce découpage a permis de valider progressivement chaque brique fonctionnelle avant d'aborder les aspects avancés de parallélisation et d'optimisation.

7.2 Organisation du développement et itérations

Le développement s'est appuyé sur un processus itératif, visible dans l'historique du dépôt Git :

- création initiale du projet et mise en place de la structure `src/include` ;
- premières implémentations du GBM, de la régression OLS et du LSMC ;
- phases de refonte complètes du code afin d'améliorer la lisibilité, la modularité et les performances ;
- intégration progressive d'OpenMP sur les boucles critiques ;
- ajout d'une interface de visualisation et d'export des résultats (scripts Python, Streamlit) ;
- implémentation d'un backend CUDA complet incluant la simulation GBM, le calcul des payoffs, la backward induction et les réductions nécessaires à la régression ;
- nettoyage approfondi du dépôt après des problèmes liés à des fichiers CSV volumineux et à l'historique Git.

Cette approche incrémentale a permis de maintenir un code fonctionnel à chaque étape tout en introduisant progressivement des optimisations plus complexes.

7.3 Répartition des tâches et binôme

Afin d'optimiser notre efficacité, nous nous sommes répartis les tâches selon nos affinités et compétences techniques :

- **Florian Barbe** : Responsable du "cœur de calcul" (*Core Engine*).
 - Modélisation mathématique et implémentation C++.
 - Développement de l'algorithme LSMC et des solveurs FDM.
 - Parallélisation CPU avec OpenMP.
 - Développement complet du kernel CUDA et de l'implémentation GPU.
 - Benchmarking et optimisation des performances bas niveau.
- **Narjisso** : Responsable de l'expérience utilisateur et de l'interface (*Frontend*).
 - Conception de l'interface graphique (GUI) pour rendre l'outil accessible.
 - Développement d'une application interactive (via Streamlit/Python) permettant de modifier les paramètres (S_0, K, r, σ) et de lancer les simulations sans toucher au code C++.
 - Visualisation des résultats (graphiques de convergence, trajectoires) et intégration avec l'exécutable C++.

Cette séparation claire (Back-end en C++/CUDA vs Front-end en Python) nous a permis d'avancer en parallèle : pendant que le moteur de calcul était optimisé, l'interface utilisateur était développée pour consommer les résultats produits.

7.4 Outils et environnement collaboratif

Le projet s'est appuyé sur les outils suivants :

- **Git / GitHub** pour le suivi du développement et l'historique des itérations ;
- **CMake** pour la configuration et la compilation du projet ;
- **OpenMP** pour la parallélisation CPU ;
- **CUDA C++** pour l'implémentation GPU ;
- **Python** et **Streamlit** pour l'analyse et la visualisation des résultats ;
- **Overleaf** pour la rédaction du rapport.

L'utilisation intensive de Git a joué un rôle central, notamment pour gérer les phases de refonte, corriger des erreurs structurelles (fichiers volumineux, historique trop lourd) et stabiliser le dépôt en fin de projet.

7.5 Gestion du temps et avancement

Le travail a été réparti sur l'ensemble du semestre avec une montée en complexité progressive :

- début de semestre : compréhension théorique, premières implémentations CPU séquentielles ;
- milieu de semestre : restructuration du code, parallélisation OpenMP, premières analyses de performance ;
- fin de semestre : implémentation CUDA complète, optimisation mémoire, nettoyage du dépôt Git, analyse comparative et rédaction finale.

Cette organisation a permis d'anticiper les difficultés techniques, notamment celles liées au calcul parallèle, à la gestion mémoire et aux limitations structurelles de l'algorithme LSMC, tout en assurant la livraison d'un projet fonctionnel, documenté et cohérent avec les objectifs pédagogiques du P1RV.

8 Conclusion

Ce projet a permis de concevoir, implémenter et analyser une version performante de l'algorithme de Longstaff–Schwartz pour le pricing d'options américaines, en combinant rigueur mathématique, validation numérique et étude approfondie des performances sur différentes architectures de calcul.

L'implémentation séquentielle sur CPU a servi de référence fonctionnelle et a permis de valider la cohérence de l'algorithme. La parallélisation via OpenMP a montré des gains mesurés mais limités, principalement contraints par la bande passante mémoire et la présence de phases séquentielles incompressibles. En revanche, l'implémentation GPU avec CUDA a démontré des accélérations significatives pour les phases massivement parallèles, en particulier la simulation Monte Carlo des trajectoires, avec des speedups pouvant atteindre un ordre de grandeur par rapport au CPU.

Les résultats obtenus mettent toutefois en évidence une limite structurelle fondamentale du LSMC : la backward induction impose une dépendance temporelle forte qui empêche toute parallélisation complète dans la dimension du temps. Le goulot d'étranglement réside donc dans l'estimation séquentielle de la valeur de continuation à chaque pas de temps, ce qui plafonne l'accélération théorique, même sur des architectures massivement parallèles.

La validation croisée avec des méthodes déterministes de différences finies a confirmé la cohérence numérique des prix obtenus, les écarts restant compatibles avec la variance statistique attendue d'une méthode de Monte Carlo. L'étude du choix des bases de régression a par ailleurs mis en évidence le compromis biais–variance inhérent à l'algorithme, ainsi que les limites pratiques des régressions polynomiales de degré élevé en présence de bruit stochastique.

En conclusion, ce travail montre que le GPU constitue une solution particulièrement adaptée pour les méthodes de Monte Carlo à grande échelle, tout en soulignant que les gains de performance sont intrinsèquement bornés par la structure algorithmique du LSMC. Ces résultats ouvrent la voie à des approches hybrides

(batching d'options, parallélisme asynchrone) visant à contourner la barrière séquentielle temporelle, seule restriction à l'essor du LSMC sur les architectures exascale.

A Résolution de l'EDS du GBM

On considère le processus $(S_t)_{t \geq 0}$ solution, sous la mesure risque-neutre \mathbb{Q} , de l'équation différentielle stochastique

$$dS_t = rS_t dt + \sigma S_t dW_t^{\mathbb{Q}},$$

[...Dérivation...]

B Compléments de Performance OpenMP

Les figures suivantes illustrent la scalabilité de l'implémentation OpenMP.

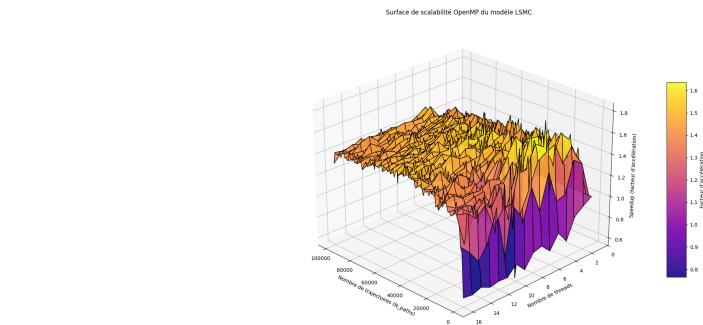


Figure 5: Surface de scalabilité OpenMP : temps d'exécution en fonction du nombre de trajectoires et de threads.

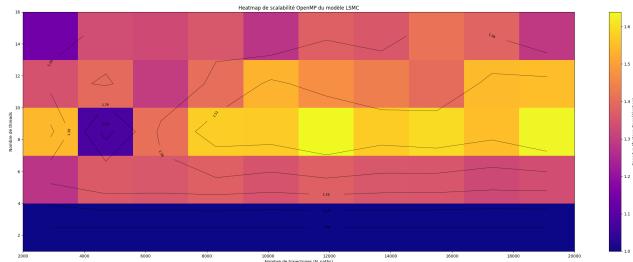


Figure 6: Heatmap des performances OpenMP.

Les figures suivantes illustrent la scalabilité de l'implémentation OpenMP en fonction du nombre de trajectoires et de threads.

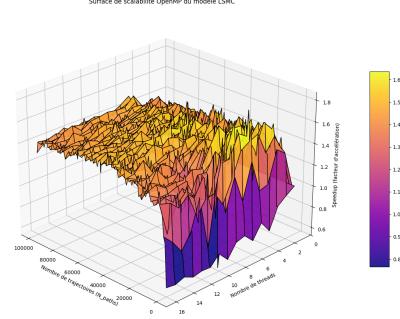


Figure 7: Surface de scalabilité OpenMP : temps d'exécution en fonction du nombre de trajectoires et de threads.

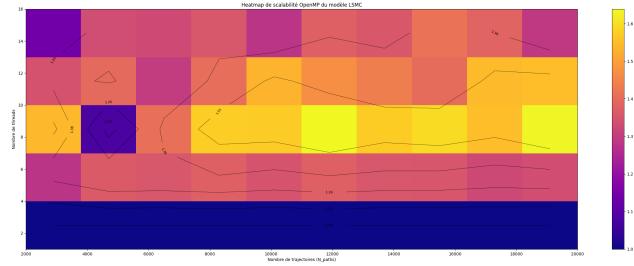


Figure 8: Heatmap des performances OpenMP.

C Rappels de Probabilités et propriétés du Mouvement Brownien

C.1 Définition et propriétés du Mouvement Brownien

Un mouvement brownien standard $(W_t)_{t \geq 0}$ est un processus stochastique caractérisé par :

1. $W_0 = 0$ presque sûrement.
2. Ses trajectoires $t \mapsto W_t$ sont continues.
3. Ses accroissements sont indépendants et stationnaires (loi normale $\mathcal{N}(0, t - s)$ pour $W_t - W_s$).

C.2 Propriétés des incrément

Pour la simulation numérique, nous utilisons la propriété d'indépendance des accroissements sur des intervalles disjoints. Si $0 \leq t_1 < t_2 < \dots < t_n$, les variables aléatoires $(W_{t_{i+1}} - W_{t_i})$ sont mutuellement indépendantes.

C.3 Loi des Grands Nombres et Monte Carlo

La convergence des méthodes de Monte Carlo repose sur la Loi Forte des Grands Nombres (LFGN) :

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(X_i) = \mathbb{E}[f(X)] \quad \text{p.s.}$$

L'erreur d'approximation décroît en $\mathcal{O}(1/\sqrt{N})$, conformément au Théorème Central Limite.

D Notations et Conventions

Cadre probabiliste

- $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t \geq 0}, \mathbb{P})$: espace probabilisé filtré usuel.
- \mathbb{Q} : mesure risque-neutre.
- $W_t^{\mathbb{Q}}$: mouvement brownien standard sous \mathbb{Q} .

Modèle de marché

- S_t : prix du sous-jacent à t .
- r, σ : taux sans risque et volatilité (constants).
- T, K : maturité et strike de l'option.
- $\Phi(S)$: payoff ($\max(K - S, 0)$ pour un Put).

Paramètres numériques

- $N_{\text{steps}}, \Delta t$: nombre de pas de temps et pas de temps.
- N_{paths} : nombre de simulations Monte Carlo.
- V_{t_n} : valeur de l'option à la date discrète t_n .
- $\widehat{C}(t, S)$: estimateur de la valeur de continuation par régression.
- $\psi_k(S)$: fonctions de base pour la régression LSMC.

References

- [1] Longstaff, F. A., & Schwartz, E. S. (2001). *Valuing American Options by Simulation: A Simple Least-Squares Approach*. The Review of Financial Studies, 14(1), 113-147.
- [2] Hull, J. C. *Options, Futures, and Other Derivatives*. Pearson Education.
- [3] Glasserman, P. (2004). *Monte Carlo Methods in Financial Engineering*. Springer.
- [4] Oger, G. *Mémoire de Magistère : Valorisation d'options américaines sur GPU*.
- [5] Croain, D., & Poulette. *Projet GPU : Pricing d'options américaines*.
- [6] Risks Journal (2023). *Recent Advances in American Option Pricing*. risks-11-00145.
- [7] Benguigui, M. (2015). *Valorisation d'options américaines et Value At Risk sur cluster GPU/CPU hétérogène*. Thèse de doctorat, Université Nice Sophia Antipolis.
- [8] Reesor, R. M., Stentoft, L., & Zhu, X. (2024). *A Critical Analysis of the Weighted Least Squares Monte Carlo Method for Pricing American Options*. Finance Research Letters.
- [9] Areal, N., Rodrigues, A., & Armada, M. J. R. *Improvements to the Least Squares Monte Carlo Option Valuation Method*. Source file: [Areal_Parallel_Methods.pdf](#).
- [10] Detra (2023). *Risk Management with Local Least Squares Monte Carlo*. Technical Note.