1 Méthode de Longstaff–Schwartz (LSMC)

L'objectif est d'estimer le prix d'une option américaine, c'est-à-dire un produit dérivé exerçable à n'importe quel instant avant l'échéance. Contrairement aux options européennes, il n'existe pas de formule fermée générale dans le modèle de Black–Scholes. La méthode de Longstaff–Schwartz (2001) permet d'approcher la stratégie d'exercice optimale en combinant :

- la simulation Monte Carlo du sous-jacent ;
- une régression polynomiale sur les trajectoires pour estimer la valeur de continuation ;
- un raisonnement backward (remontée du temps).

Dans notre projet, l'implémentation est optimisée :

- accumulation directe des équations normales $A^\top A$ et $A^\top y$;
- solveur 3×3 minimaliste ;
- parallélisation OpenMP pour les phases adaptées ;
- réduction des accès mémoire pour limiter les coûts CPU.

1.1 Modèle d'évolution : mouvement brownien géométrique (GBM)

Le prix du sous-jacent suit une équation différentielle stochastique :

$$dS_t = rS_t\,dt + \sigma S_t\,dW_t.$$

Sa solution discrétisée sur N_{steps} pas est :

$$S_{t+\Delta t} = S_t \exp\Big(\big(r - \tfrac{1}{2}\sigma^2\big)\Delta t + \sigma\sqrt{\Delta t}\,Z\Big),$$

où $Z \sim \mathcal{N}(0,1)$.

Dans notre code, la simulation du GBM est effectuée par la classe `GBM`, et dans le fichier `lsmc.cpp`, les trajectoires sont générées en parallèle grâce à OpenMP :

```
#pragma omp parallel
{
    RNG rng;
    #pragma omp for schedule(static)
    for (int i = 0; i < N_paths; ++i)
        paths[i] = gbm.simulate(rng);
}
```

Chaque thread possède son générateur aléatoire, ce qui évite toute contention.

1.2 Payoff : option de type put américain

Le payoff à chaque instant est :

$$\text{payoff}(S_t) = \max(K - S_t,\, 0).$$

Dans le code, les payoffs sont pré-calculés en parallèle :

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N_paths; ++i)
    for (int t = 0; t <= N_steps; ++t)
        payoff[i][t] = max(K - paths[i][t], 0.0);
```

1.3 Principe du LSMC : exercice optimal

Pour une option américaine, à chaque date t on compare :

$$\text{exercice immédiat} = \max(K - S_t, 0)$$

et

$$\text{valeur de continuation} = \mathbb{E}[\,\text{cashflow futur} \mid S_t\,].$$

Comme cette espérance est inconnue, on l'approxime par une régression polynomiale sur les trajectoires **in-the-money** (ITM). Dans notre implémentation, nous utilisons la base :

$$\phi(S_t) = (1,\ S_t,\ S_t^2).$$

Ceci construit localement un modèle :

$$\widehat{C}(S_t) = \beta_0 + \beta_1 S_t + \beta_2 S_t^2.$$

1.4 Accumulation des équations normales

La régression polynomiale par moindres carrés nécessite de former :

$$A^\top A = \begin{pmatrix} \sum \phi_0\phi_0 & \sum \phi_0\phi_1 & \sum \phi_0\phi_2 \\ \sum \phi_1\phi_0 & \sum \phi_1\phi_1 & \sum \phi_1\phi_2 \\ \sum \phi_2\phi_0 & \sum \phi_2\phi_1 & \sum \phi_2\phi_2 \end{pmatrix}, \qquad A^\top y = \begin{pmatrix} \sum \phi_0 Y \\ \sum \phi_1 Y \\ \sum \phi_2 Y \end{pmatrix}.$$

Dans le fichier `lsmc.cpp`, cette accumulation est entièrement parallélisée :

```
#pragma omp parallel for reduction(+:a00,a01,a02,a11,a12,a22,b0,b1,b2)
for (int i = 0; i < N_paths; i++)
{
    if (payoff[i][t] > 0.0) {
        double S = paths[i][t];
        double Y = cashflows[i] * discount;
        double phi0 = 1.0, phi1 = S, phi2 = S*S;

        a00 += phi0*phi0;
```

```
        a01 += phi0*phi1;
        a02 += phi0*phi2;
        a11 += phi1*phi1;
        a12 += phi1*phi2;
        a22 += phi2*phi2;

        b0 += phi0*Y;
        b1 += phi1*Y;
        b2 += phi2*Y;
    }
}
```

Cette approche évite la création de grands vecteurs `X` et `Y`, ce qui réduit fortement le coût mémoire et améliore les performances CPU.

1.5 Résolution du système 3×3

Une fois $A^\top A$ et $A^\top y$ obtenus, on résout :

$$(A^\top A)\,\beta = A^\top y.$$

Le solveur `solve3x3` implémente une élimination de Gauss spécialisée et robuste aux petits pivots. Il est extrêmement léger (quelques dizaines de cycles processeur).

1.6 Règle d'exercice et mise à jour des cashflows

Pour chaque trajectoire, on compare :

$$\text{immediate} = \max(K - S_t, 0)$$

à

$$\text{continuation} = \beta_0 + \beta_1 S_t + \beta_2 S_t^2.$$

Dans le code :

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N_paths; ++i)
{
    double immediate = payoff[i][t];

    if (immediate > 0.0) {
        double S = paths[i][t];
        double continuation =
            beta0 + beta1*S + beta2*S*S;

        if (immediate > continuation)
            cashflows[i] = immediate;
```

```
        else
            cashflows[i] *= discount;
    }
    else {
        cashflows[i] *= discount;
    }
}
```

1.7 Backward induction

On répète l'étape précédente pour les dates :

$$t = N_{\text{steps}} - 1,\ N_{\text{steps}} - 2,\ \ldots,\ 1.$$

À maturité :

$$\text{cashflows}[i] = \text{payoff}_i(T).$$

Puis à chaque date, on actualise ou on exerce selon la règle optimale.

1.8 Retour du prix de l'option

La valeur de l'option est la moyenne des cashflows actualisés :

$$\widehat{P} = \frac{1}{N_{\text{paths}}} \sum_{i=1}^{N_{\text{paths}}} \text{cashflows}[i].$$

Dans le code :

```
double mean = 0.0;
#pragma omp parallel for reduction(+:mean)
for (int i = 0; i < N_paths; ++i)
    mean += cashflows[i];

return mean / N_paths;
```

1.9 Résumé du rôle du fichier `lsmc.cpp`

- Simulation parallèle des trajectoires GBM.
- Calcul parallèle des payoffs.
- Backward induction avec :
  - accumulation parallèle des équations normales ;
  - résolution locale 3×3 ;
  - mise à jour parallèle des cashflows.
- Calcul parallèle de la moyenne finale.

Ce code implémente une version optimisée du LSMC CPU, avec une parallélisation adaptée à chaque phase et une gestion de la mémoire permettant d'obtenir des performances fiables, quoique limitées par la bande passante RAM (propriété typique des méthodes Monte Carlo).

2 Architecture CPU, exécution séquentielle et parallélisation OpenMP

Cette section présente le fonctionnement interne d'un processeur moderne, puis montre comment OpenMP exploite les ressources matérielles pour accélérer un code C++ tel que notre implémentation LSMC.

2.1 Architecture d'un processeur moderne

Un processeur (CPU) est organisé en plusieurs étages micro-architecturaux :

- **Coeurs (cores)** : unités d'exécution indépendantes.
- **Unités SIMD** : opérations vectorielles (AVX, SSE).
- **Pipeline** : décodage, planification, exécution, permettant d'exécuter plusieurs instructions en parallèle logique.
- **Hiérarchie de caches** :
  - L1 : très petit mais très rapide (1 ns),
  - L2 : plus grand mais plus lent,
  - L3 : partagé entre les coeurs, centaine de nanosecondes,
  - RAM : très grande, mais lent accès (100 ns).

Chaque coeur dispose de ses propres unités de calcul, mais la mémoire est partagée entre tous les coeurs. Cela entraîne des conflits d'accès (*memory contention*), limitant le parallélisme réel.

2.2 Exécution native : un modèle fondamentalement séquentiel

Par défaut, un programme C/C++ est exécuté par **un seul coeur**, selon le modèle de von Neumann :

$$\text{instruction}_1 \longrightarrow \text{instruction}_2 \longrightarrow \cdots$$

Toutes les itérations d'une boucle sont exécutées dans l'ordre. Exemple :

```
for (int i = 0; i < N; i++)
    f(i);
```

Même si la machine possède 8 ou 16 coeurs, le compilateur n'exploite qu'un seul coeur, sauf indication explicite.

C'est ce que résout OpenMP.

2.3 Comment OpenMP transforme un programme séquentiel en programme parallèle

OpenMP (Open Multi-Processing) est une API de parallélisation basée sur des **directives de compilation**. Le développeur indique quelles boucles peuvent être distribuées entre plusieurs threads.

Le compilateur transforme ensuite :

$$\text{programme séquentiel} \longrightarrow \text{programme multi-threads}$$

De façon transparente pour l'utilisateur.

OpenMP applique le modèle *fork-join* :

1. le thread principal crée un **pool de threads** (2, 4, 8, 16...)
2. le travail est partagé entre les threads
3. les threads synchronisent leurs résultats
4. le thread principal continue le programme

La directive la plus importante est :

```
#pragma omp parallel for
```

Elle indique que les itérations d'une boucle peuvent être exécutées indépendamment, sans dépendance séquentielle.

2.4 Exemple illustratif : avant et après OpenMP

Version séquentielle

```
for (int i = 0; i < N_paths; i++)
    paths[i] = simulate(i);
```

Version OpenMP

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N_paths; i++)
    paths[i] = simulate(i);
```

Le compilateur génère automatiquement du code :

- créant plusieurs threads,
- assignant un sous-ensemble d'itérations à chaque thread,
- garantissant que la boucle est exécutée sans interférence.

2.5 Répartition de la charge : rôle de `schedule`

OpenMP propose plusieurs stratégies :

- **static** : les itérations sont réparties par blocs contigus (**meilleur cache** et **charge homogène**) ;
- **dynamic** : les threads demandent du travail au fur et à mesure (**meilleur équilibre** si les tâches sont déséquilibrées) ;
- **guided** : paquets décroissants (mixte entre static et dynamic).

Dans le LSMC, la charge par trajectoire est homogène, donc :

```
schedule(static)
```

est optimal.

2.6 Gestion de la mémoire : variables partagées et privées

Dans une région parallèle, il est crucial de distinguer :

- **shared** : accessible par tous les threads
- **private** : dupliquée pour chaque thread

Exemple typique dans notre code :

```
#pragma omp parallel for private(rng) shared(paths)
for (int i = 0; i < N_paths; i++)
    paths[i] = gbm.simulate(rng);
```

Chaque thread possède :

- son **RNG privé**, pour éviter les conflits,
- un accès partagé au tableau **paths**.

2.7 Parallélisation des réductions

Dans LSMC, on calcule :

$$a_{00}, a_{01}, a_{02}, a_{11}, a_{12}, a_{22}, \qquad b_0, b_1, b_2.$$

Ces quantités doivent être sommées en parallèle. OpenMP propose une syntaxe efficace :

```
#pragma omp parallel for reduction(+:a00,a01,a02,a11,a12,a22,b0,b1,b2)
```

Le compilateur :

- crée une copie locale de chaque accumulateur,
- les combine en fin de boucle,
- garantit l'absence de race condition.

2.8 Limites physiques du CPU et impact sur les performances

Même avec 16 coeurs, les gains ne sont pas linéaires. Les limitations principales sont :

- **bande passante mémoire limitée** (40–70 GB/s) ;
- **goulot d'étranglement du cache** ;
- **latence élevée** pour accéder à la RAM ;
- **nombre réduit de coeurs physiques** (souvent 4–16).

Ainsi :

$$\text{Speedup}(N_{\text{threads}}) \ll N_{\text{threads}}.$$

Dans nos tests, l'accélération CPU réelle est d'environ 3 à 7× selon la taille du problème.

2.9 Résumé

- Le CPU est une architecture généraliste, optimisée pour la **latence** et le **contrôle logique**.
- Un code C++ est naturellement **séquentiel**.
- OpenMP permet une parallélisation simple, portable et efficace.
- Le LSMC utilise intensivement OpenMP dans toutes les phases où les trajectoires sont indépendantes.
- Les performances restent limitées par la bande passante RAM, phénomène classique des algorithmes Monte Carlo.

3 Introduction à CUDA et calcul parallèle sur GPU

L'exécution GPU représente un changement majeur par rapport à l'exécution traditionnelle sur CPU. CUDA (Compute Unified Device Architecture) est la plateforme de calcul parallèle de NVIDIA permettant d'exécuter du code massivement parallèle sur les cartes graphiques.

Un GPU n'est pas un processeur généraliste comme un CPU : il est conçu pour exécuter simultanément des milliers de threads, chacun réalisant une quantité très faible de travail, mais de manière parfaitement parallèle. Le GPU est donc particulièrement adapté aux algorithmes présentant une structure *SIMD* (Single Instruction, Multiple Data), comme la simulation Monte Carlo.

3.1 Architecture matérielle d'un GPU NVIDIA

Un GPU est structuré en plusieurs blocs fonctionnels appelés **Streaming Multiprocessors** (SM). Chaque SM contient :

- des centaines de **CUDA cores** (unités de calcul scalaires) ;
- une mémoire rapide par bloc (**Shared Memory**) ;
- des unités spécialisées (multiplicateurs, SFU, Tensor cores éventuels) ;
- un planificateur matériel de threads.

L'organisation hiérarchique est la suivante :

GPU ⇒ SM (Streaming Multiprocessors) ⇒ Warps (32 threads) ⇒ Threads.

Chaque **warp** est un groupe de 32 threads exécutés en vrai parallèle dans un SM. Tous les threads d'un warp exécutent la *même instruction* au même instant. Dès qu'un thread diverge (conditionnelle), le warp doit sérialiser la branche, ce qui pénalise énormément la performance.

3.2 Hiérarchie d'exécution CUDA : threads, warps et blocs

Lorsqu'un kernel CUDA est lancé (via la syntaxe `<<< >>>`), le programmeur ne choisit pas explicitement où les threads s'exécutent. CUDA crée :

- des **threads**, unités de calcul élémentaires ;
- regroupés en **blocs de threads** (thread blocks) ;
- eux-mêmes organisés en une **grille** (grid).

Un lancement typique est :

```
myKernel<<< gridSize, blockSize >>>(args...);
```

Les définitions importantes sont :

- **thread** : unité de base, exécute du code séquentiel ;
- **warp** : groupe matériel de 32 threads ;
- **block** : ensemble de threads partageant la *shared memory* ;
- **grid** : ensemble de blocs exécutant le même kernel ;
- **kernel** : fonction marquée `__global__` exécutée sur le GPU ;
- **device** : carte graphique ;
- **host** : CPU.

Le développeur choisit la taille des blocs et la grille, ce qui influence la *occupancy* du GPU (pourcentage de ressources effectivement utilisées).

3.3 Hiérarchie mémoire CUDA

L'efficacité d'un programme GPU dépend fortement de la manière dont il utilise les différents types de mémoire :

- **Global memory** : grande capacité, lente, accessible par tous ;
- **Shared memory** : stockage très rapide partagé par un bloc ;
- **Registers** : stockage le plus rapide, privé à chaque thread ;
- **Constant / texture memory** : caches spécialisés ;
- **L2 cache** : cache global permettant de réduire les accès DRAM.

Les règles essentielles :

- privilégier les lectures coalescentes (accès séquentiels par warps) ;
- éviter les divergences de threads dans un warp ;
- minimiser les allers-retours Host $\leftrightarrow$ Device ;
- utiliser la shared memory lorsque plusieurs threads manipulent des données proches.

4 CUDA dans le cadre du LSMC

La méthode LSMC se prête parfaitement au calcul GPU :

- la simulation des trajectoires GBM est **indépendante** : parfaite pour un lancement de milliers de threads ;
- le calcul des payoffs est également SIMD ;
- la mise à jour backward nécessite des réductions (somme globale), très efficaces via kernels spécialisés ou *thrust*.

Toutefois, la partie la plus délicate est la régression (**accumulation de** $A^\top A$ **et** $A^\top y$). Elle nécessite des **reductions parallèles**, une tâche pour laquelle CUDA excelle grâce :

- aux **warp reductions** ;
- aux **kernel de réduction hiérarchique** ;
- ou à la librairie `cub::DeviceReduce`.

4.1 Exemple de kernel CUDA pour la simulation GBM

Voici un kernel typique permettant d'assigner un thread par trajectoire :

```
__global__ void simulate_gbm_paths(
    double* paths,
    double S0, double r, double sigma, double dt,
    int N_steps, int N_paths,
    curandState* states)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N_paths) return;

    double S = S0;
    paths[i*(N_steps+1)] = S;

    curandState localState = states[i];

    for (int t = 1; t <= N_steps; t++)
    {
        double Z = curand_normal_double(&localState);
        S *= exp((r - 0.5*sigma*sigma)*dt + sigma*sqrt(dt)*Z);
        paths[i*(N_steps+1) + t] = S;
    }

    states[i] = localState;
}
```

Chaque thread génère sa propre trajectoire.
Le lancement se fait typiquement via :

```
int blockSize = 256;
int gridSize  = (N_paths + blockSize - 1) / blockSize;

simulate_gbm_paths<<< gridSize, blockSize >>>(
    d_paths, S0, r, sigma, dt, N_steps, N_paths, d_states
);
```

4.2 Kernel de calcul du payoff

```
__global__ void compute_payoff(
    const double* paths,
    double* payoff,
    double K, int N_steps, int N_paths)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N_paths) return;
```

```
    for (int t = 0; t <= N_steps; t++) {
        double S = paths[i*(N_steps+1) + t];
        payoff[i*(N_steps+1) + t] = fmax(K - S, 0.0);
    }
}
```

4.3 Réduction parallèle pour accumuler $A^\top A$ et $A^\top y$

La construction des équations normales repose sur des dizaines de milliers de petites contributions. Un kernel simple accumulerait mal en mémoire globale. On utilise donc une structure hiérarchique :

- chaque thread calcule ses contributions locales ;
- chaque bloc réalise une réduction dans la shared memory ;
- la réduction finale s'effectue sur le GPU ou CPU.

Structure typique :

```
__global__ void accumulate_normal_equations(
    const double* paths,
    const double* cashflow,
    const double* payoff,
    double discount,
    int N_steps, int N_paths,
    int t, double* partial_AT_A, double* partial_AT_y)
{
    extern __shared__ double buffer[];

    // buffer[0..8] = contributions locales du bloc
    // buffer initialisé à zéro par les threads du bloc
    ...
}
```

La réduction finale s'effectue avec :

```
cudaMemcpy(...)
```

ou via :

```
cub::DeviceReduce::Sum(...)
```

ce qui permet d'obtenir directement les coefficients $\beta_0, \beta_1, \beta_2$.

4.4 Synthèse : pourquoi CUDA accélère massivement le LSMC

- Chaque trajectoire GBM est indépendante : un thread = une trajectoire.
- Les payoffs peuvent être évalués en parallèle.
- Les réductions statistiques s'effectuent extrêmement bien sur GPU.
- Le backward LSMC est adapté car chaque date t est une réduction globale.

Les gains attendus par rapport au CPU vont de $\times 50$ à $\times 500$ selon :

- le nombre de trajectoires ;
- la taille du backward ;
- la structure mémoire ;
- l'occupation du GPU.

CUDA permet donc d'évaluer en quelques millisecondes des simulations nécessitant plusieurs secondes en CPU.

5 Conclusion générale sur l'intégration CUDA

L'intégration CUDA dans un projet LSMC transforme un algorithme fortement CPU-bound (limité par la bande passante mémoire) en un algorithme massivement parallèle exploitant la puissance brute du GPU.

L'essentiel repose sur :

- découper chaque phase en kernels indépendants ;
- exploiter la coalescence mémoire ;
- éviter les divergences dans les warps ;
- structurer les réductions avec shared memory ;
- limiter les transferts Host $\leftrightarrow$ Device.

Le LSMC étant un algorithme naturellement parallèle, le portage CUDA s'intègre parfaitement et ouvre la voie à des simulations de très grande dimension ($N_{\text{paths}} \sim 10^6$ et au-delà).

6 Accélération GPU CUDA : principes, limites et intégration

Dans cette section, nous détaillons l'utilisation potentielle du GPU pour accélérer la méthode de Longstaff–Schwartz (LSMC). Le GPU permet une exécution massivement parallèle, mais son efficacité dépend fortement de la structure des données et des dépendances temporelles de l'algorithme. Nous décrivons ici les points clés nécessaires à une implémentation CUDA robuste.

6.1 Motivations : pourquoi utiliser CUDA ?

Les simulations Monte Carlo sont souvent idéales pour les GPU car elles reposent sur un très grand nombre de chemins indépendants. De plus, les évolutions de type GBM sont locales : la mise à jour d'une trajectoire ne dépend que de son propre état précédent. Cela permet :

- d'attribuer un thread CUDA par trajectoire ;
- d'exploiter la hiérarchie GPU (threads/blocs/grilles) ;
- de simuler des millions de chemins simultanément.

Dans le cas présent, la simulation des trajectoires est la partie la plus facile à transférer sur GPU.

6.2 Contraintes : pourquoi le LSMC est difficile à paralléliser complètement ?

La difficulté principale provient de la phase de **backward induction** :

- Pour chaque date t on doit calculer les coefficients $(\beta_0, \beta_1, \beta_2)$ à partir de l'ensemble des trajectoires ITM.
- Cette étape nécessite une **réduction globale** sur l'ensemble des chemins (calcul de $A^\top A$ et $A^\top y$).
- Une fois ce calcul terminé, on peut enfin mettre à jour les cashflows.

Autrement dit :

On ne peut pas paralléliser dans le temps : $t \to t-1$ dépend de t.

Le GPU excelle dans le *parallélisme spatial*, mais pas dans les dépendances temporelles.

Ainsi, la stratégie optimale consiste à faire :

- simulation sur GPU ;
- calcul des payoffs sur GPU ;
- accumulation des régressions sur GPU (via kernels de réduction) ;
- backward date-par-date avec synchronisation CPU/GPU minimale.

6.3 Organisation mémoire CUDA

Pour exploiter au mieux le GPU, les données doivent être stockées en mémoire globale avec un layout contigu :

```
paths[i * (N_steps + 1) + t]
```

Ce format permet :

- un **coalesced memory access** optimal ;
- un parcours des trajectoires par thread ;
- une compatibilité directe avec notre code CPU optimisé.

6.4 Kernel CUDA pour la simulation GBM

```
// 1 thread = 1 trajectoire
__global__
void simulate_gbm_kernel(double* paths, int N_paths, int N_steps,
                         double S0, double r, double sigma, double dt,
                         curandState* states)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N_paths) return;

    curandState local = states[i];
    double S = S0;
    paths[i*(N_steps+1)] = S;

    for (int t = 1; t <= N_steps; t++) {
        double Z = curand_normal_double(&local);
        S = S * exp((r - 0.5*sigma*sigma)*dt + sigma*sqrt(dt)*Z);
        paths[i*(N_steps+1) + t] = S;
    }

    states[i] = local;
}
```

Chaque thread simule une trajectoire complète.

6.5 Kernel CUDA pour les payoffs

```
__global__
void payoff_kernel(double* payoff, const double* paths,
                   int N_paths, int N_steps, double K)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (i >= N_paths) return;

    for (int t = 0; t <= N_steps; t++) {
        double S = paths[i*(N_steps+1) + t];
        payoff[i*(N_steps+1) + t] = fmax(K - S, 0.0);
    }
}
```

6.6 Réduction parallèle pour les équations normales

Pour calculer les matrices :

$$A^\top A, \quad A^\top y,$$

il faut utiliser plusieurs kernels successifs :

1. kernel qui calcule localement les contributions ($\phi_0\phi_0$, $\phi_0\phi_1, \dots$) ;
2. réduction parallèle en utilisant un schéma hiérarchique :
   - réduction par bloc dans shared memory ;
   - écriture dans un buffer global ;
   - réduction finale par le CPU ou un dernier kernel.

6.7 Mise à jour des cashflows sur GPU

```
__global__
void update_cashflows(double* cash, const double* payoff,
                      const double* paths, int N_paths, int N_steps,
                      int t, double beta0, double beta1, double beta2,
                      double discount)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N_paths) return;

    double immediate = payoff[i*(N_steps+1) + t];
    if (immediate > 0.0) {
        double S = paths[i*(N_steps+1) + t];
        double continuation = beta0 + beta1*S + beta2*S*S;

        if (immediate > continuation)
            cash[i] = immediate;
        else
            cash[i] *= discount;
    }
    else {
        cash[i] *= discount;
```

```
    }
}
```

6.8 Comparaison CPU OpenMP vs GPU CUDA

- **Simulation** : GPU largement supérieur (jusqu'à $\times 100$).
- **Payoff** : accélération modérée (mémoire-bound).
- **Backward induction** : accélération faible car dépendante du temps.

Au final, le gain global dépend fortement de :

- la taille (N_paths) ;
- la bande passante mémoire ;
- la capacité du GPU à réduire efficacement les équations normales.

6.9 Plan d'intégration CUDA dans le projet

1. Ajouter un fichier `gbm_cuda.cu`.
2. Écrire les kernels :
   - `simulate_gbm_kernel`
   - `payoff_kernel`
   - kernels de réduction
   - `update_cashflows`
3. Utiliser un wrapper C++ pour lancer les kernels depuis `lsmc.cpp`.
4. Synchroniser date par date pour récupérer les coefficients β.
5. Comparer les perfs GPU vs CPU.

6.10 Conclusion

L'intégration CUDA permet :

- une accélération massive sur la simulation,
- une accélération limitée mais réelle sur le backward,
- un potentiel très élevé sur grands problèmes (plusieurs millions de chemins).

Cependant, l'algorithme reste limité par sa structure séquentielle dans le temps, ce qui fait que l'optimisation GPU ne peut pas être parfaite. Pour un GPU moderne, le gain final se situe typiquement entre $\times 3$ et $\times 20$ selon les tailles et la carte utilisée.