ENSIMAG

Calcul parallèle avancé

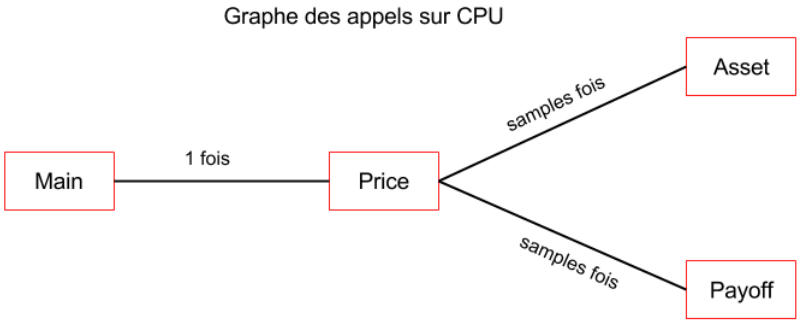
Parallélisation du pricer

Etudiant :
Guillaume Fuchs

2 mai 2014

L'objectif de ce rapport est de montrer les résultats de l'implémentation parallèle sur GPU du pricer Monte-Carlo réalisé durant le cours Modélisation et programmation de 2A. La première partie du document présente la démarche analytique mise en place pour la parallélisation. Enfin, la deuxième partie présentera et étudiera les résultats de performance obtenues avec cette implémentation.

1 Conception du pricer sur GPU



Ce que l'on constate sur le graphe des appels sur CPU c'est que la procédure `price` appel un très grand nombre de fois en série les procédures `asset` et `payoff`. Il semble donc important de pouvoir paralléliser ces deux procédures afin de réaliser les différents appels.

Le pricer sur GPU alloue des blocs de dimension *1*512* et une grille dont le nombre de blocs dépend du nombre de samples. On subdivise le nombre de samples en un nombre optimal de samples (*OptimalSamples*) selon la taille et le nombre de pas de temps de l'option. Il va correspondre au nombre de thread total qui calculeront une estimation du prix de l'option. On défini ce nombre de samples par :

$$OptimalSamples = \frac{10240}{N * size}$$

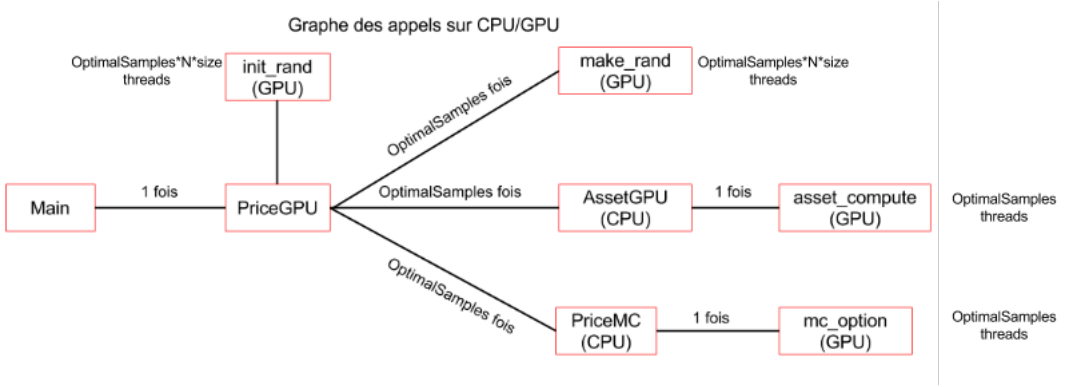
Le nombre 10240 est déterminé par des tests de performance sur différentes options que nous verrons plus tard et dépend aussi de la machine sur laquelle les tests sont effectués.

Le pricer va posséder une boucle principal qui va calculer le prix et l'intervalle de confiance d'une option possédant un nombre de samples égal à *OptimalSamples*. La somme de ces prix et de ces intervalles sur la boucle complète va nous donner le prix et l'intervalle de confiance demandé (avec le nombre de samples total).
Avant d'entrer dans la boucle, la fonction appel `init_rand` (fonction sur GPU) qui va permettre d'initialiser la graine du générateur de nombre aléatoire. Le nombre de thread total qui vont l'appeler sur un tour de boucle est de $OptimalSamples * N * size$.
Sur un tour de boucle les appels sont :

— `make_rand` (fonction sur GPU) qui va permettre d'associer à chaque thread un nombre aléatoire issue de la loi normale centrée réduite. Le nombre de thread total qui vont l'appeler est de $OptimalSamples * N * size$.

— `assetGPU` (fonction sur CPU) qui va permettre de générer l'ensemble des chemins de l'option en utilisant `asset_compute` (fonction sur GPU). A noter que le nombre de thread total qui vont appeler cette fonction est différent du nombre de thread total dans `make_rand` et `init_rand` car dans `assetGPU`, un thread va créer *size* chemins de N pas de temps ce qui correspond à une option. Il utilisera donc les $size * N - 1$ threads suivant pour obtenir une suite d'échantillon de loi normale centrée réduite indépendante et identiquement distribuée. Le nombre de thread total qui appellerons la fonction est de *OptimalSamples*.
— `price_mc` qui va appeler selon le type de l'option la fonction associée pour permettre de calculer le prix et l'intervalle de confiance sur un nombre de samples donné par une méthode de réduction par `mc_option` (fonction sur GPU). Le nombre de thread total qui appellerons la fonction est de *OptimalSamples*.

Le graphe des appels sur CPU/GPU est donc le suivant :



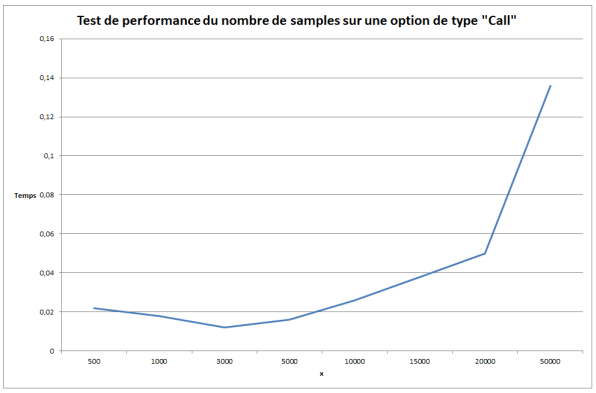
2 Analyse des performances

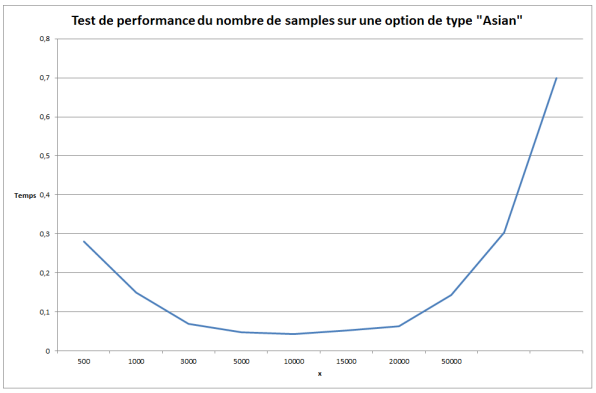
Les tests de performance ont été effectués sur une carte graphique NVIDIA GeForce GT 750M. Les informations obtenus via `cudaGetDevicePropertiesl` sont :

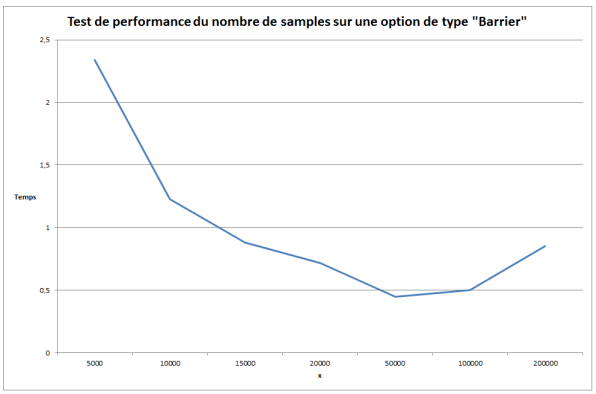
— Taille mémoire globale 4 294 967 295
— Taille mémoire constante : 65 536
— Taille mémoire partagée : 49 152
— Nombre de threads maximum par direction : 1024 x 1024 x 64
— Nombre maximum de thread par bloc : 1024
— Nombre de blocs par grille : 2 147 483 647 x 65 535 x 65 535

2.1 Détermination du nombre samples optimal

Pour déterminer le nombre de samples optimal qui va correspondre à calculer pour chaque tour de boucle une option possédant un nombre de samples égal à *OptimalSamples*, on détermine le temps d'exécution de différentes options (ici "call", "asian0", "barrier") en fonction du nombre x qui est présent dans $\frac{x}{N*size}$.





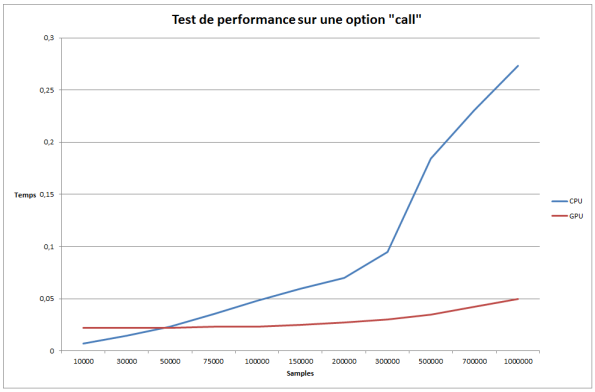


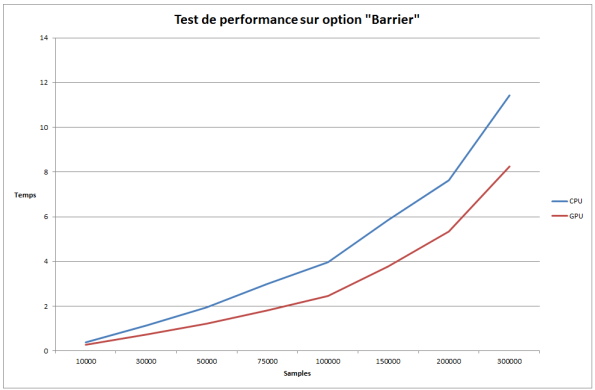
Ces tests de temps d'exécution montre que selon le type de l'option, le nombre de samples optimale est différent. Les options de type "Call" ne demandent pas de parcourir les N prix du sous-jacent contrairement aux options de type "Asian" et "Barrier". Chaque thread ne passe pas beaucoup de temps dans le calcul du premier type d'option tandis que dans le deuxième type d'option, les threads passent beaucoup de temps pour calculer un prix.

Dans le cas du pricer, nous avons fixé le nombre x à 10240 pour ne contraindre aucun type d'option sachant que les options de type "Call" ne demandent que très peu de temps de calcul.
Il serait intéressant de continuer cette étude sur le temps d'exécution par rapport au nombre de samples optimal afin pouvoir modifier le nombre optimal de samples selon le type d'option.

2.2 Test de performance

On s'intéresse ici à l'évolution du temps d'exécution du pricer sur deux type d'option : une option "call" et une option "barrier". De plus, on compare le temps d'exécution avec le pricer conçut en 2A qui s'exécute uniquement sur CPU.





On constate que pour une option de type "call", le temps d'exécution augmenter de façon linéaire sur CPU/GPU tandis qu'il augmente de façon exponentielle sur CPU. En effet à partir de 300 000 échantillons, le temps d'exécution du pricer sur CPU augmente de façon très forte. Cela peut s'expliqué par le fait que la mémoire utilisé pour le pricer sur CPU/GPU est constante (on utilise toujours une taille de mémoire qui est proportionnel à $OptimalSamples$ et qui est constante quelle que soit le nombre de samples total), tandis que sur CPU, on alloue des tableaux de taille $Samples$ qui correspond au nombre total d'échantillon. Si ce nombre augmente, la mémoire peut saturer et les calculs sont plus lent à effectuer.
De plus, pour un nombre d'échantillon faible (de l'ordre de 10 000), le temps d'exécution sur CPU est plus faible que le temps d'exécution sur CPU/GPU. Ce temps est du au fait que la mémoire à utiliser est très faible sur CPU et que les calculs se font très rapidement car la mémoire n'affecte pas le temps d'exécution. Sur CPU/GPU, la mémoire utilisée est constante et peut être plus élevée que sur CPU.

Sur une option de type "barrier", le temps d'exécution augmente de la même façon que l'on soit sur CPU ou sur CPU/GPU avec un temps d'exécution toujours plus faible pour le pricer sur CPU/GPU. L'option de type "barrier" demande lors du calcul du payoff de parcourir l'esnemble des prix du sous-jacent sur la durée de vie de l'option. Le fait de pouvoir parcourir de façon parallèle plusieurs chemins permet d'obtenir un gain de temps par rapport à parcourir de façon séquentielle les chemins.