Monte-Carlo Option Pricing

Victor Podlozhnyuk
vpodlozhnyuk@nvidia.com

June 2007

Document Change History

Version	Date	Responsible	Reason for Change
1.0	03/20/2007	vpodlozhnyuk	Initial release

Abstract

The pricing of options is a very important problem encountered in financial engineering since the creation of organized option trading in 1973. As more computation has been applied to finance-related problems, finding efficient ways to implement option pricing models on modern architectures has become more important. This sample shows an implementation of the Monte-Carlo approach to the problem of option pricing in CUDA.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Introduction

The most common definition of an *option* is an agreement between two parties, the *option seller* and the *option buyer*, whereby the option buyer is granted a right (but not an obligation), secured by the option seller, to carry out some operation (or *exercise* the option) at some moment in the future. [1]

Options come in several varieties: *A call option* grants its holder the right to buy some *underlying asset* (stock, real estate, or any other good with inherent value) at a fixed predetermined price at some moment in the future. The predetermined price is referred to as *strike price*, and future date is called *expiration date*. Similarly, a *put option* gives its holder the right to *sell* the underlying asset at a *strike price* on the *expiration date*.

For a call option, the profit made at expiration date – assuming a same-day sale transaction – is the difference between the price of the asset at expiration date and the strike price, minus the option price. For a put option, the profit made at expiration date is the difference between the strike price and the price of the asset at expiration date, minus the option price.

The price of the asset at expiration date and the strike price therefore strongly influence how much one would be willing to pay for an option.

Other factors are:

The time to the expiration date, T: Longer periods imply wider range of possible values for the underlying asset on the expiration date, and thus more uncertainty about the value of the option.

The riskless rate of return, R, which is the annual interest rate of bonds or other "risk-free" investment: any amount P of dollars is guaranteed to be worth $P \bullet e^{rT}$ dollars T years from now if placed today in one of theses investments or in other words, if an asset is worth P dollars T years from now, it is worth $P \bullet e^{-rT}$ today, which must be taken in account when evaluating the value of the option today.

Exercise restrictions: So far only so-called European options, which can be exercised only on the expiration date, have been discussed. But options with different types of exercise restriction also exist. For example, American-style options are more flexible as they may be exercised at any time up to and including expiration date and as such, they are generally priced at least as high as corresponding European options.

The Monte-Carlo Method in Finance

The price of the underlying asset S_t follows a geometric Brownian motion with constant drift μ and volatility v: $dS_t = \mu S_t dt + v S_t dW_t$ (where W_t is Wiener random process: $X = W_T - W_0 \sim N(0,T)$).

The solution of this equation is:

$$\frac{dS_t}{S_t} = \mu dt + v dW_t \Rightarrow S_T = S_0 \bullet \exp(\mu T + v(W_T - W_0)) = S_0 \bullet \exp(\mu T + vN(0,T)) =$$

$$S_0 \bullet \exp(\mu T + N(0, v^2 T)) = S_0 \bullet \exp(\mu T + v\sqrt{T} N(0,1))$$

The expected future value is:

$$E(S_T) = S_0 \bullet \exp(\mu T) \bullet E(\exp(N(0, v^2 T)) = S_0 \bullet \exp(\mu T) \bullet \exp(0.5 v^2 T) =$$

$$= S_0 \bullet \exp((\mu + 0.5 v^2) T)$$

By definition, $E(S_T) = S_0 \bullet \exp(rT) \Rightarrow \mu = r - 0.5 v^2$, so

$S_T = S_0 \bullet \exp((r - 0.5 v^2) T + v\sqrt{T} N(0,1))$ - the possible stock end price, depending on the random sample N(0, 1), "describing" how exactly the stock price was moving.

The possible prices of derivatives at the period end are derived from the possible underlying asset's price:

$V_{call}(S,T) = \max(S_T - X, 0)$ (If the market stock price at the exercise date is greater than the strike price, a call option makes its holder a profit of $S_T - X$ dollars, zero otherwise.)

$V_{put}(S,T) = \max(X - S_T, 0)$ (If the strike price at the exercise date is greater than the market stock price, a put option makes its holder profit of $X - S_T$, zero otherwise).

One of the possible solutions to estimate the mathematical expectations of $V_{call}(S,T)$ and $V_{put}(S,T)$ is to take some amount of N(0, 1) random samples, calculate the derivative end-period prices corresponding to each of the samples, and average the generated prices:

$$V_{mean}(S,T) = \frac{\sum_{i=1}^{N} V_i(S,T)}{N}$$

This is the core of the Monte-Carlo approach to option pricing.

Discounting the approximation of future price by discount factor of $e^{-r \cdot T}$ we get an approximation of the present-day fair derivative price: $V_{fair}(S,0) = V_{mean}(S,T) \bullet e^{-r \cdot T}$

Though in our particular problem closed-form expressions for $E(V_{call}(S,T))$ and $E(V_{put}(S,T))$ are known from the Black-Scholes formula, (which is used to compute reference analytical values for comparison against Monte-Carlo simulation results), in most applications of the Monte-Carlo approach closed-form expressions are unknown.

Implementation Details

The first stage of the computation is the generation of a normally distributed pseudo-random number sequence. For this sample we use a parallel version of the Mersenne Twister random number generator to generate a uniformly distributed [0, 1] sequence, and the Cartesian form of the Box-Muller transformation to transform the distribution into a normal one. For more details on the efficient CUDA implementation of the Mersenne Twister and Box-Muller transformation please refer to the "MersenneTwister" SDK sample.

Once we've generated the desired amount of N(0, 1) samples, the rest of the process of Monte-Carlo simulation maps very well onto the CUDA programming model:

```
const int     tid = blockDim.x * blockIdx.x + threadIdx.x;
const int threadN = blockDim.x * gridDim.x;
//...

for(int iAccum = tid; iAccum < accumN; iAccum += threadN){
    float sum = 0, sum2 = 0;

    for(int iPath = iAccum; iPath < pathN; iPath += accumN){
        float r = d_Random[iPath];
        //...
        sum  += endOptionPrice;
        sum2 += endOptionPrice * endOptionPrice;
    }

    d_Sum[iAccum] = sum;
    d_Sum2[iAccum] = sum2;
}
```

Listing 1. Main pass of Monte-Carlo simulation.

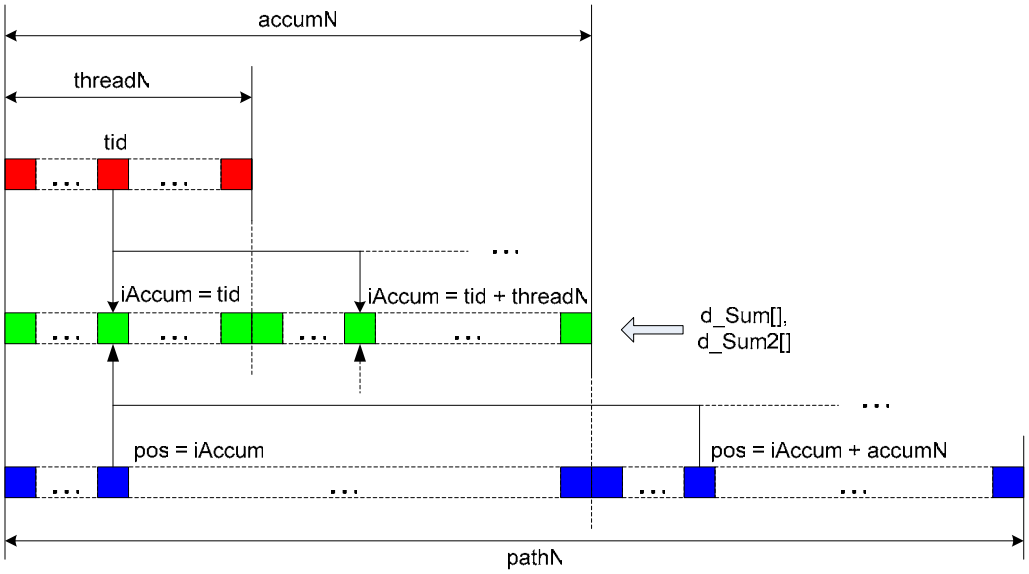

Figure 1. The structure of the reduction loop.

Each CUDA thread is responsible for the computation of several simulation paths and accumulation into several accumulators. Let's briefly describe the structure of the kernel:

In this sample the computation grid can be simply treated as an implicit for loop, encapsulating the code of the CUDA kernel:

for(tid = 0; tid < threadN; tid++) MonteCarloKernelGPU(tid, …);

The first level of nesting within the kernel is the accumulation loop; after the completion of *iAccum* iterations, *ds_Sum[iAccum]* and *ds_Sum2[iAccum]* contain partial sums and partial sums of squares. The innermost loop performs the actual "problem-related" computations for dedicated indices in the random samples array *d_Random[]* and accumulates the produced results into *iAccum*-th position of the accumulation arrays.

After the kernel execution is complete, all the *accumN* accumulators and all the *pathN* simulation paths *are* covered. *d_Sum* and *d_Sum2* now contain exactly *accumN* partial sums and partial sums of squares, which can be processed further.

As long as *accumN* and the number of threads in the computation grid are multiples of the warp size (32), memory accesses are perfectly coalesced. The situation where the thread block sizes are not multiples of the warp size is very rare though, since normally there is no sense in having under-populated warps and thus hardware resources idling.

The last stage of computation is the completion of the reduction and the computation of the present day fair option price value and the confidence width. Although this stage is carried out on the CPU and involves memory readback from the GPU, as long as *accumN* is small enough (we've chosen 16384), this stage takes only a fraction of the total processing time.

Performance

The time for processing a single option on an 80M sample domain is less than 11 ms on the GeForce8800 GTX

Conclusion

This sample demonstrates that CUDA-enabled GPUs are capable of efficient Monte-Carlo simulation, significantly outperforming the best available CPU implementations in the field of option pricing.

Bibliography

1. Simon Leger : "Monte Carlo techniques applied to finance",. http://homepages.nyu.edu/~sl1544/articles.html
2. Black, Fischer; Myron Scholes (1973). "The Pricing of Options and Corporate Liabilities". *Journal of Political Economy* **81** (3): 637-654.
3. Craig Kolb, Matt Pharr (2005). "Option pricing on the GPU". *GPU Gems 2*. Chapter 45.