Acceleration of Option Pricing Technique on Graphics Processing Units

Bowen Zhang* Cornelis. W. Oosterlee†

April 23, 2010

Abstract

The acceleration of an option pricing technique based on Fourier cosine expansions on the Graphics Processing Unit (GPU) is reported. European options, in particular with multiple strikes, and Bermudan options will be discussed. The influence of the number of terms in the Fourier cosine series expansion, the number of strikes, as well as the number of exercise dates for Bermudan options, are explored. We also give details about the different ways of implementing on a GPU. Numerical examples include asset price processes based on a Lévy process of infinite activity and the stochastic volatility Heston model. Furthermore, we discuss the issue of precision on the present GPU systems.

Keywords: Option pricing, Fourier Cosine expansions, Graphics Processing Units implementation, Options with multiple strikes, Riccati ODEs, Jump and stochastic volatility processes.

1 Introduction

In this paper we deal with a topic from Computational Finance, i.e., the efficient pricing of options on stocks or other assets. Several methods for pricing these contracts exist. The Feynman-Kac theorem relates the conditional expectation of the value of an option contract payoff function under the risk-neutral measure to the solution of a partial differential equation. Various pricing techniques can therefore be developed, like partial-(integro) differential equation (PIDE) finite-difference solvers, monte Carlo simulations or numerical integration methods. Option pricing techniques need to be accurate, robust and fast. The latter feature is particularly necessary

*Delft University of Technology, Delft Institute of Applied Mathematics, Delft, the Netherlands, email: bowen.zhang@tudelft.nl

†CWI – Centrum Wiskunde & Informatica, Amsterdam, the Netherlands, email: c.w.oosterlee@cwi.nl, and Delft University of Technology, Delft Institute of Applied Mathematics.

when the mathematical asset price models are calibrated to real market data. Option values, with many different parameter values for the underlying asset price process, are then computed thousands of times in order to fit the mathematical model. In this paper, we focus on this setting and show that it may make sense to perform this task on a Graphics Processing Unit-based computer.

A highly efficient pricing method is the COS method [1, 2], based on Fourier cosine series expansions. In this method, based on the conditional expectation formula, the conditional density function of the underlying is approximated by a series expansion which is connected to the characteristic function. The COS method is applicable if the characteristic function of the stochastic asset price process (i.e. the Fourier transform of the conditional density function) is available. This is certainly the case for state-of-the-art asset price models, like the Lévy jump processes and the Heston stochastic volatility process, which we discuss in the present paper. However, also more involved *hybrid* stochastic processes, for example for interest rates and equity can be considered, as long as we can get to a characteristic function. For such asset models with stochastic volatility *and* stochastic interest rate, like the Heston-Hull-White or the Heston–Gaussian two-factor model, the analytic characteristic function is typically not available. However, after some appropriate reformulations of the SDE system (see, for example [7]) the coefficients of the characteristic function can be found as the solution of a Riccati system of ordinary differential equations (ODEs), as described in [8]. These ODE systems can be solved numerically by means of an explicit Runge–Kutta method or by other ODE solvers. We will show that this task can also be performed efficiently on a GPU.

In practice, the option values obtained from a mathematical model should be consistent with market option prices. Usually, options with many different strike prices are needed for calibration. In the COS method, European option prices *for a vector of strikes* can be computed in one computation, which accelerates the calibration procedure significantly.

To further accelerate the calibration procedure, two approaches directly come into mind. The easiest is to purchase a faster CPU! As an example, Table 1 compares error convergence and cpu times between the CPU used in [1] (CPU 1: Intel(R) Pentium(R) 4 CPU, 2.80GHz with cache size 1MB) and a faster CPU (CPU 2: Intel(R) Core(TM)2 Duo CPU, E6550 @ 2.33GHz Cache size 4MB) for European calls under a Geometric Brownian Motion asset process. Time is in milli-seconds[1].

The faster CPU gives a satisfactory acceleration, but one needs to wait (sometimes up to two years) for an acceleration by a factor two.

Another possibility to accelerate the pricing engine is to run the program, or parts of it, on the popular Graphics Processing Unit, which supports

[1] 1 milli-second=10^{-3} second

parallel computation. Executing a code on a GPU is worthwhile if:

1. A program can be divided into several independent parts;
2. A program does not contain many sequential parts;
3. A program does not require much memory transfer from host to device or vice versa.

Previous work in the direction of option pricing, with an integration-based method [6], concluded that the GPU option pricing code outperformed a corresponding CPU code for pricing American and so-called path-dependent options, but not for European options. A large number of space and time points was needed to show the advantages of the GPU. In our paper, we will demonstrate a significant performance improvement on the GPU, due to parallelization, when pricing European options with *multiple strikes* but with not-more-than-necessary terms in the Fourier cosine expansion. The GPU may therefore be used for the important task of calibration, which is traditionally done with European options. For certain modern option products it also makes sense to calibrate to barrier options, or other, liquid financial products. The pricing of barrier options with the COS method is closely connected to the example of pricing Bermudan options in the present paper.

The outline of this paper is as follows. Section 2 gives an introduction in the COS option pricing method. For European options different ways of implementation of the method are described in Section 3. Section 4 presents the speed-up for multi-strike European options on the GPU. The acceleration of an explicit Runge–Kutta method for numerically solving systems of Riccati ODEs to approximate a characteristic function (if it is not available in closed form) is presented in Section 5. Section 6 gives pricing results for Bermudan options, that may be exercised early (before the maturity date). Here the influence of the number of terms in the Fourier cosine expansion as well as the number of early exercise dates on the speed-up factor on the GPU are discussed.

The GPU we work on is an NVIDIA GeForce 9800 GX2, which has two graphics processing units (GPUs) and 1 GB of memory (512 MB for each GPU); the CPU on the same computer, needed for data transfer etc., is an AMD Athlon(tm)64 X2 Dual Core Processor 4600+ (cache size 512 KB, 2412.364MHz).

The results obtained are compared with timings on a CPU from an Intel(R) Core(TM)2 Duo CPU E6550 (@ 2.33GHz Cache size 4MB).

2 COS Pricing Method

Starting from the risk-neutral valuation formula

$$v(x,t_0) = e^{-r\Delta t}\int_{-\infty}^{\infty} v(y,T)f(y|x)dy,$$

where $v(x,t)$ is the option value, and x, y can be any increasing functions of the underlying at t_0 and T, respectively, we truncate the integration range, so that

$$v(x,t_0) \approx e^{-r\Delta t}\int_{a}^{b} v(y,T)f(y|x)dy. \tag{1}$$

with $|\int_{\mathbb{R}} f(y|x)dy - \int_a^b f(y|x)dy| < TOL$. Error analysis of the various approximations is given in [1, 2].

The conditional density function of the underlying is then approximated by means of the characteristic function via a truncated Fourier cosine expansion, as follows:

$$f(y|x) \approx \frac{2}{b-a}{\sum_{k=0}^{N-1}}' Re(\phi(\frac{k\pi}{b-a};x)\exp{(-i\frac{ak\pi}{b-a})})\cos{(k\pi\frac{y-a}{b-a})}, \tag{2}$$

where Re means taking the real part of the expression in brackets, and $\phi(\omega;x)$ is the characteristic function of $f(y|x)$ defined as:

$$\phi(\omega;x) = \mathbb{E}(e^{i\omega y}|x). \tag{3}$$

The prime at the sum symbol in (2) indicates that the first term in the expansion is multiplied by one-half. Replacing $f(y|x)$ by its approximation (2) in (1) and interchanging integration and summation, gives us the COS algorithm to approximate the value of a European option:

$$v(x,t_0) = e^{-r\Delta t}{\sum_{k=0}^{N-1}}' Re(\phi(\frac{k\pi}{b-a};x)e^{-ik\pi\frac{a}{b-a}})V_k, \tag{4}$$

where

$$V_k = \frac{2}{b-a}\int_a^b v(y,T)\cos{(k\pi\frac{y-a}{b-a})}dy \tag{5}$$

is the Fourier cosine coefficient of $v(y,T)$, which is available in closed form for several European option payoff functions.

Formula (4) can be directly applied to calculate the value of a European option, and it also forms the basis for pricing Bermudan options.

The COS algorithm exhibits an exponential convergence rate for all processes whose conditional density $f(y|x) \in C^{\infty}((a,b) \subset \mathbb{R})$. The size of the integration interval $[a,b]$ can be determined with help of the cumulants [1].

2.1 Pricing of European Options with Multi–Strike Features

With $X_t = \log(S_t/K)$, the solution for Equation (5) can be written as

$$V_k = U_k K, \tag{6}$$

where

$$U_k = \{ \begin{array}{ll} \frac{2}{b-a}(\chi_k(0,b) - \psi_k(0,b)), & \text{for a call,} \\ \frac{2}{b-a}(\psi(a,0) - \chi(a,0)), & \text{for a put,} \end{array} \tag{7}$$

with

$$\chi_k(x_1,x_2) := \int_{x_1}^{x_2} e^x \cos\left(k\pi\frac{x-a}{b-a}\right) dx, \tag{8}$$

$$\psi_k(x_1,x_2) := \int_{x_1}^{x_2} \cos\left(k\pi\frac{x-a}{b-a}\right) dx. \tag{9}$$

Now, the pricing formula (4) reads:

$$v(x,t_0) = Ke^{-r\Delta t} Re({\sum_{k=0}^{N-1}}' \phi(\frac{k\pi}{b-a};x) \cdot e^{-ik\pi\frac{a}{b-a}} U_k) \tag{10}$$

Let's assume that we deal with a vector of strikes, $K = [K(1), \cdots, K(P)]^T$, where P is the number of strikes. Recall from (1) that $[a,b]$ is the truncation range for the log–asset price x. As x is a function of K, i.e. $x(j) = \log(S/K(j))$, $j = 1, \cdots, P$, the parameters a and b also depend on K; therefore, they are also vectors with length P. For a vectorised version of the COS method, enabling an efficient computation of multi-strike options, we define the following matrices:

- Φ is a $(P \times N)$-matrix with elements

  $$\Phi(j,k+1) = \phi(\frac{k\pi}{b(j)-a(j)};x(j))e^{-ik\pi\frac{a(j)}{b(j)-a(j)}},$$

  $j = 1, \cdots, P, k = 0, \cdots, N-1$.

- $\bar{U}$ is an $(N \times 1)$-vector with elements $\bar{U}(k+1,1) = U_k,\ k = 0, \cdots, N-1$.

- Λ is a $(P \times P)$ diagonal matrix with $\Lambda(j,j) = K(j),\ j = 1, \cdots, P$.

With these matrices, the formula for pricing options with multi–strike features reads:

$$v(\mathbf{x},t_0) = e^{-r\Delta t}\ \Lambda\ Re(\Phi\ U), \tag{11}$$

so that option values for many strikes can be computed simultaneously.

2.2 COS Method for Bermudan Options

The pricing formula for a Bermudan option with M exercise dates, with $m = M, M-1, \ldots, 2$, is divided into a stage in which a *continuation value* is computed, and a stage where this value is compared to the pay-off $g(x, t_{m-1}) \equiv v(x, T)$. These stages are given by:

$$\begin{cases} c(x, t_{m-1}) = e^{-r\Delta t} \int_{\mathbb{R}} v(y, t_m) f(y|x) dy, \\ v(x, t_{m-1}) = \max\left(g(x, t_{m-1}), c(x, t_{m-1})\right), \end{cases} \tag{12}$$

followed by the final computation,

$$v(x, t_0) = e^{-r\Delta t} \int_{\mathbb{R}} v(y, t_1) f(y|x) dy. \tag{13}$$

In this description, we have $x := \ln\left(S(t_{m-1})/K\right) \quad y := \ln\left(S(t_m)/K\right)$, and $v(x, t), c(x, t)$ are the option value, and the continuation value at time t, respectively. For vanilla options $g(x, t) \equiv (\alpha K(\exp(x) - 1))^+$ with $\alpha = 1$ for a call and $\alpha = -1$ for a put.

Practically, for each time step we first determine an early-exercise point, x^*_m, for which $c(x^*_m, t_m) = g(x^*_m, t_m)$ by means of the Newton method. If x^*_m lies outside interval $[a, b]$ we set x^*_m equal to the nearest boundary point. At each time step, t_m, we then can split the Fourier cosine coefficients $V_k(t_m)$ into two parts:

$$V_k(t_m) = C_k(a, x^*_m, t_m) + G_k(x^*_m, b), \qquad \text{for a call,} \tag{14}$$

$$V_k(t_m) = G_k(a, x^*_m) + C_k(x^*_m, b, t_m), \qquad \text{for a put.} \tag{15}$$

for $m = M-1, M-2, \ldots, 1$, and

$$V_k(t_M) = G_k(0, b) \qquad \text{for a call,}$$

$$V_k(t_M) = G_k(a, 0) \qquad \text{for a put.}$$

Here,

$$G_k(x_1, x_2) = \frac{2}{b-a} \int_{x_1}^{x_2} g(x, t_m) cos(k\pi \frac{x-a}{b-a}) dx, \tag{16}$$

$$C_k(x_1, x_2, t_m) = \frac{2}{b-a} \int_{x_1}^{x_2} \hat{c}(x, t_m) cos(k\pi \frac{x-a}{b-a}) dx, \tag{17}$$

with

$$c(x, t_m) = e^{-r\Delta t} {\sum_{k=0}^{N-1}}' Re(\phi(\frac{k\pi}{b-a}; x) e^{-ik\pi \frac{a}{b-a}}) V_k(t_{m+1}),$$

from the Fourier cosine expansion.

$G_k(x_1, x_2)$ is known analytically, like for European options, and for Lévy processes,

$$C(x_1, x_2, t_m) \equiv C_k(x_1, x_2, t_m))_{j=0}^{N-1}$$

can be written as

$$C(x_1, x_2, t_m) = e^{-r\Delta t} Im(M_s u + M_c u)/\pi$$

where Im means taking the imaginary part of the expression in brackets. $M_s u$ represents the first N elements of $D^{-1}(D(m_s) \cdot D(u_s))$ [2], and $M_c u$ denotes the computation of the first N elements of $D^{-1}(D(m_c) \cdot sgn \cdot D(u_s))$, in reversed order, see [2].

In this description, we have

$$sgn = [1, -1, 1, -1, \dots]^T, \; m_s = [m_0, m_{-1}, \cdots, m_{1-N}, 0, m_{N-1}, \cdots, m_1]^T,$$

$$m_c = [m_{2N-1}, m_{2N-2}, \cdots, m_1, m_0]^T, \; u_s = [u_0, u_1, \cdots, u_{N-1}, 0, \cdots, 0]^T,$$

with elements

$$m_j = \frac{(x_2 - x_1)}{b - a}\pi i, \quad \text{if} \quad j = 0,$$

$$m_j = \frac{\exp(ij\frac{(x_2-a)\pi}{b-a}) - \exp(ij\frac{(x_1-a)\pi}{b-a})}{j}, \quad \text{if} \quad j \neq 0.$$

Finally, $u_j = \phi(j\pi/(b-a))V_j(t_{m+1})$ and $u_0 = \frac{1}{2}\phi(0)V_0(t_{m+1})$.

For all time steps, $m = M - 1, \cdots, 1$, approximation of $V_k(t_m)$ is recovered from (14) or (15). Option value $v(x, t_0)$ is obtained by inserting $V_k(t_1)$ into (13), and then, applying (4) with T replaced by t_1.

2.3 Underlying Asset Processes

In this paper we discuss two different underlying asset processes, the CGMY process, a Lévy jump process and the Heston stochastic volatility process. For our purposes, the characteristic functions related to these processes are needed.

The CGMY process, as defined in [9], is a generalisation of the Variance Gamma process with the following characteristic function:

$$\phi_{CGMY}(\omega, t) = \exp(tC\Gamma(-Y)[(M - i\omega)^Y - M^Y + (G + i\omega)^Y - G^Y]). \quad (18)$$

Four parameters need to be calibrated to market data: Parameter $Y : Y < 2$ controls whether the CGMY process has finite or infinite activity. Parameter $C : C > 0$ controls the kurtosis of the distribution and non-negative parameters G, M give control over the rate of exponential decay on the right and left tails of the density, respectively.

In the Heston stochastic volatility model, the underlying and the volatility are modeled by the following stochastic differential equations,

$$\begin{aligned} dx_t &= (r - \frac{1}{2}\mu_t)dt + \sqrt{\mu_t}dW_{1,t}, \\ d\mu_t &= \lambda(\bar{\mu} - \mu_t)dt + \eta\sqrt{\mu_t}dW_{2,t}, \end{aligned} \quad (19)$$

[2]Here, $D(vector)$ denotes the discrete Fourier transform, whereas D^{-1} stands for the inverse discrete Fourier transform.

where x_t and μ_t denote the log–asset price process and the variance of the asset price process, respectively. Parameters $\lambda, \bar{\mu}, \eta$ represent the speed of mean–recursion, the mean value of variance and the volatility of volatility. Moreover, $W_{1,t}$ and $W_{2,t}$ are Brownian motions, correlated with correlation coefficient ρ.

For the log-asset price in the Heston model an analytic characteristic function can be found, which reads:

$$\begin{aligned}\phi(\omega, \Delta t, \mu_0) = \quad & \exp \quad (i\omega r\Delta t + \frac{\mu_0}{\eta^2}(\frac{1-e^{-D\Delta t}}{1-Ge^{-D\Delta t}})(\lambda - i\rho\eta\omega - D)) \cdot \\ & \exp \quad (\frac{\lambda\bar{\mu}}{\eta^2}(\Delta t(\lambda - i\rho\eta\omega - D) - 2\log(\frac{1-Ge^{-D\Delta t}}{1-G})))\end{aligned}$$

with $D = \sqrt{(\lambda - i\eta\rho\omega)^2 + (\omega^2 + i\omega)\eta^2}$ and $G = \lambda - i\eta\rho\omega - D/\lambda - i\eta\rho\omega + D$.

As for the value of D, we take the square root whose real part is non-negative.

Remark 2.1. *[Advantage of COS method on GPU] From [10] we know that, compared to the execution on a CPU, the GPU is favorable for many of the time-consuming operations in the COS method. Moreover, the elements of the sum in (4) are independent of each other and can be computed simultaneously. Therefore, the GPU is expected to outperform the CPU when executing the COS algorithm, in particular when many computations are necessary, as for European options with multi-strike features and for Bermudan options.*

3 European Options

A European option can be viewed as a special case of a Bermudan option with only one possible exercise date (the expiry time). For European options, the Fourier and inverse Fourier transform operations are not needed.

3.1 Different Ways of GPU Implementation

In this section, we discuss different ways of implementation on the GPU. Consider a simple case where we need to price one vanilla option.

From (4) the COS algorithm can be decomposed into two steps, i.e., computations on each element of a vector, $Re(\exp(-ik\pi\frac{a}{b-a})\phi(\frac{k\pi}{b-a}; x)V_k)$, which can be parallelized; and the summation of vector elements.

We consider three ways of GPU implementation:

1. Directly run the whole code on the GPU, referred to as GPU1;
2. All operations related to each vector element are parallelized on the GPU, whereas the summation is performed on the CPU. This hybrid GPU/CPU way of implementation is referred to as GPU2;

3. When summing up the elements of a vector, we can split the vector in two vectors, each of size $N/2$, and sum up these two on the GPU. The procedure is repeated until $N = 1$. The summation of pairs of elements can be parallelized on the GPU this way. The number of operations for the summation can be reduced from N to $\log_2(N)$, referred to as GPU3.

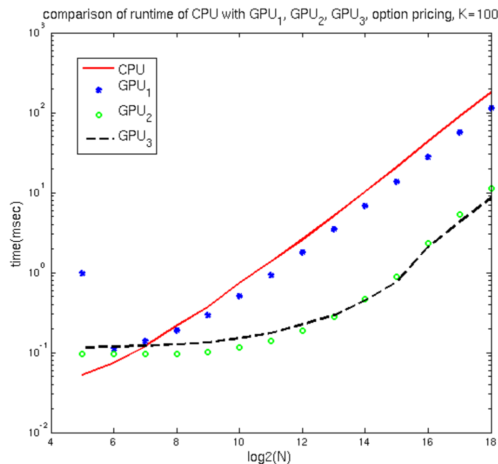

Figure 1: Comparison of different GPU implementations.

Figure 1 presents a comparison of the time consumed by the above mentioned three ways of GPU implementation and also the CPU time. Clearly, GPU1 is faster than the CPU only when N, the number of terms in the Fourier cosine expansion, is large, whereas the implementations GPU2 and GPU3 are significantly faster than either the CPU implementation or GPU1. Moreover, as N increases, the speed-up of GPU2 and GPU3 also increases. GPU3 is slightly slower than GPU2 for small N, but when N is very large, GPU3 beats GPU2.

Moreover, unlike the CPU or GPU1, the time for GPU2 and GPU3 does not increase much as N increases, until $N \approx 2^{16}$.

For Bermudan options, implementation GPU3 is preferred, since the complete code then runs on the GPU and data transfer can be reduced. With GPU2, we would need to transfer data at each time step, which consumes time.

In this paper we will use implementation GPU3 for all numerical examples to follow.

3.2 Numerical Example

We take as an example the CGMY model with $Y = 1.5$. The other parameters are chosen as $S_0 = 100$; $K = 80$; $C = 1$; $M = 5$; $G = 5$. Table 2 compares time and accuracy of the CPU and the GPU results, with time measured in milli-seconds. Table 2 shows that with $N = 1024$ the GPU implementation is 1.5 times faster than running the code on the CPU. However, it is not necessary to take such a large value of N in the COS method in practice, as with $N = 256$ the option values already differ less than one basis point. Therefore, in the present setting (COS method, small values of N) the GPU is not advantageous. However, with multiple strikes, presented in Sections 4 and 5, many more computations are needed so that the GPU performance is expected to be more profound.

4 Multiple Strike Option Pricing

In this section we focus on pricing European options, but now with multiple strikes, on the GPU. The parameters used, next to $S_0 = 100$, in the CGMY process and for the Heston process are:

$$\begin{aligned} &\text{CGMY}: && r = 0.1; C = 1; G = 5; M = 5; Y = 1.5; T = 1; \\ &\text{Heston}: && \lambda = 1.577; \eta = 0.575; r = 0.040, \mu_0 = 0.018; \rho = -0.57; T = 10. \end{aligned}$$

We price European call options with different vectors of strikes, as shown in Table 3.

To efficiently implement (11) on a GPU, we first divide the P–axis and the N–axis in different blocks and threads, as shown in Figure 2.

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
```

Figure 2: Blocks and threads.

Then each element of a $(P \times N)$-matrix can be calculated *simultaneously* as illustrated in Figure 3.

When performing the summation on each row of matrix v, as the final step in (11), we divide the $(P \times N)$-matrix into smaller sub-matrices. For instance, with $N = 128$, and 21 strikes, the corresponding 21×128-matrix can be subdivided into fifty–six 3×16-matrices. The values of these smaller sub-matrices are copied to shared memory as shown in Figure 4:

Here *aBegin* is the first location of As, i.e. the blocks with shared memory, and tx, ty are the thread indices of As. Then as we run the program,

```
v[i*Ndim+j]=exp(-r*T)*K[j]*real(cf[i*Ndim+j]
            *exp((x[i]-a[i])*omega[i*Ndim+j])*U[i]);
```

Figure 3: Parallelization of cos method for options with multi–strike.

```
__shared__ float As[BLOCK_SIZE_K][BLOCK_SIZE_N];
As[tx][ty]=A[aBegin+N*tx+ty];
__syncthreads();
```

Figure 4: Data transfer from global memory to shared memory.

data transfer only happens within the shared memory, and not in the global memory, which saves us a lot of GPU time.

4.1 Convergence and Precision

Tables 4 and 5 present the convergence behaviour and the precision of option values with 5 strikes and 21 strikes, respectively, for the two underlying processes. Time is again measured in milli-seconds. Option values obtained with $N = 2^{16}$, and in double precision, are taken as the reference values. We calculate the maximum absolute error, for varying values of N, over the strike vectors.

Both the GPU and CPU results are extremely fast, as we need only $N = 64$ for the CGMY process and $N = 128$ for Heston's model, to obtain converged option values on the GPU and the CPU. However, the execution time on the GPU is significantly smaller than on the CPU. As we are in the milli-seconds range, one might question the relevance of this gain in speed. However, within a calibration setting option prices have to be computed several thousands of times, which immediately turns a small gain into a significant profit. The advantage of the use of the GPU becomes more pronounced when the values of N and K increase, since then more arithmetic operations are required. As shown in Tables 4 and 5, the acceleration on the GPU for Heston's model increases for 5 strikes from 12 to 21, as N increases from 128 to 256. For 21 strikes, the speed-up on the GPU is a factor 37

for $N = 128$ and 47 for $N = 256$. Since the evaluation of the characteristic function of the Heston model is more involved than the one of the CGMY model, the speed-up on the GPU for the Heston model is higher than for the CGMY model. However, due to the fact that the present GPUs give computed values in single precision only, round-off errors can build up easily during the computation of the characteristic function on the GPU. A larger maximum absolute error for the Heston model is therefore observed in the tables.

In the next section, we will deal with an explicit Runge-Kutta ODE solver to determine the characteristic function for Heston's model. In this numerical procedure operations like taking the square root of a complex number are not needed, in contrast to the evaluation of the analytic solution. For the numerical ODE solver the influence of single precision arithmetic is therefore less pronounced, and we obtain a higher accuracy than with the analytic characteristic function.

Figure 5 presents the speed-up obtained on the GPU for different numbers of strikes, with $N = 128, 512, 2048$. When N is relatively small ($N = 128$), we get an improved speed-up when the number of strikes increases. With N large ($N = 512, 2048$), however, more strikes may lead to a lower speed-up factor on the GPU due to the increased data transfer time between the CPU and the GPU, and due to the fact that a larger memory is needed. Figure 5 displays a speed-up of 30-40 for 21 strikes on the GPU with only $N = 128$, and a speed-up of 60-70 for 13 and 17 strikes with $N = 512$.

Figure 6 shows that, due to the parallelization, the GPU time hardly changes for $N \approx 2^9 - 2^{10}$. With $N > 2^{10}$, however, the GPU time increases as the influence of data transfer time comes into play.

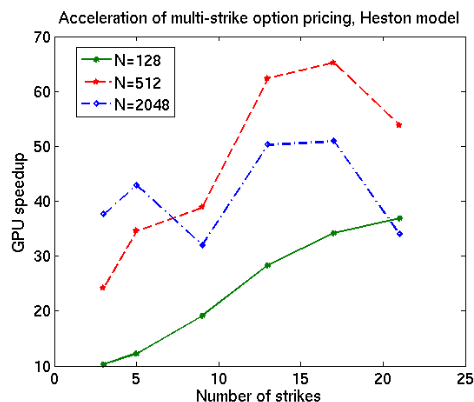

Figure 5: GPU speed-up for Heston model with different number of strikes, $N = 128, 512, 2048$.

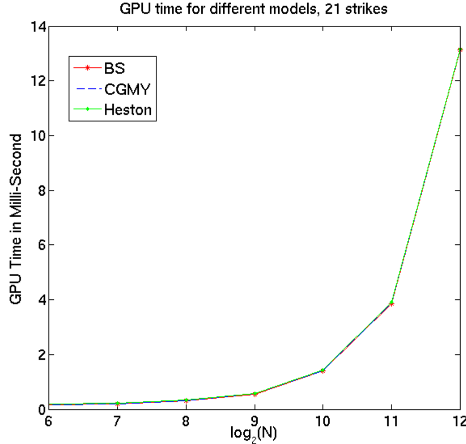

Figure 6: GPU time versus N for different stochastic models.

5 Riccati ODEs and Characteristic Function

The Riccati ODEs arise in the determination of a characteristic function. Often, as in the case of the Heston model, the characteristic function is known analytically. In the cases for which we cannot find an analytic expression, we may resort to a numerical solution of the Riccati ODEs. This is typically for systems of stochastic differential equations that are involved (meaning, including stochastic interest rate, by means of a two-factor model, and stochastic volatility, for example) we need the numerical approximation. Here, we focus again on the Heston model, and pretend that the characteristic function is not available. We aim at determining it by means of an explicit Runge-Kutta ODE solver. From [8] we know that the characteristic function for the Heston model (19) is of the following form:

$$\phi_{x,\mu}(\omega, t) = e^{A(\omega,t)+B_\mu(\omega,t)\mu_0+B_x(\omega,t)x_0}, \tag{20}$$

with the coefficients $A(\omega, t)$, $B_x(\omega, t)$ and $B_\mu(\omega, t)$ given by the following Riccati ODEs:

$$\begin{cases} \frac{\partial}{\partial t} B_x(\omega, t) = 0, & B_x(\omega, 0) = i\omega \\ \frac{\partial}{\partial t} B_\mu(\omega, t) = 0.5\eta^2\mu_t^2 - (\lambda - i\rho\eta\omega)\mu_t - 0.5i\omega - 0.5\omega^2, & B_\mu(\omega, 0) = 0 \\ \frac{\partial}{\partial t} A(\omega, t) = \lambda\bar{\mu}\mu_t + i(r - q)\omega, & A(\omega, 0) = 0. \end{cases} \tag{21}$$

It is easy to see from (21) that $B_x(\omega, t) = i\omega$. $A(\omega, t)$ and $B_\mu(\omega, t)$ are solved numerically by the explicit fourth order Runge–Kutta method (RK4),

and inserted in the general characteristic function (20), to employ the COS method.

Tables 6 and 7 show the timing results on the GPU and the CPU for 5 and 21 strikes, respectively, with the characteristic function determined by the RK4 method. Compared to the Tables 4 and 5, a higher speed-up can now be achieved on the GPU compared to the case in which the analytic characteristic function is used. For 5 strikes, with $N = 128$ and $N = 256$, the GPU timings are 45 and 65 times faster than the CPU results. For 21 strikes the acceleration on the GPU is a factor of 103 and 100. Note that for 21 strikes, due to increased data transfer, the speed-up factor reduces as N increases, but since the COS method exhibits an exponential convergence rate, the choice $N = 128$ is sufficient for converged option prices.

Figure 7 shows the speed-up factor for the GPU with $N = 128, 256, 512$ and a different number of strikes. The advantage of using the GPU for these computations is obvious. Of course, option pricing with an analytic characteristic function is still fastest, but the numerical solution of Riccati ODEs can be performed highly efficiently on these units.

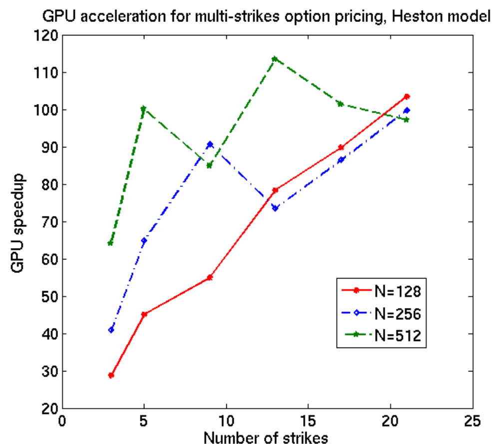

Figure 7: GPU acceleration for Heston model with different strike vectors and N, characteristic function obtained by RK4.

6 Bermudan Options

In this section we consider the pricing of Bermudan options with a discrete number of early exercise points. Also here we do not focus on large numbers of N or on many time points, M. In particular, by using the COS pricing method, with $N = 160$, we obtain the price of a Bermudan option already with an error of less than 10^{-9}. Moreover, Bermudan options with up to

64 time points ($M = 64$) are typically sufficient to get a very satisfactory approximation of the value of an American options (that can be exercised at any time before expiry) by a repeated Richardson extrapolation, see [3].

We will show in this section that by exploiting the parallelism on a GPU the COS algorithm executes faster than on the CPU, even with a "not-more-than-necessary" terms, N, in the Fourier cosine expansion and "not-more-than-necessary" time points, which is attractive from a practical point-of-view. However, the reduction of the total execution time is not as impressive as in the previous sections.

We use as a numerical example a Bermudan put option under CGMY. The model parameters are $S_0 = 100; K = 80; M = 10; C = 1; M = 5; G = 5; Y = 1.5$. Tables 8 and 9 compare time and accuracy between the GPU and CPU.

It is shown that, with N relatively small, the Bermudan option price converged well and that the error is small. However, as mentioned before, as N gets larger, probably for other types of options, the advantage of using the GPU will become more obvious.

6.1 Increasing Number of Exercise Dates

We increase the number of exercise dates, M, to 20, 40 and 80 and compare the GPU with the CPU time. The model parameters are $S_0 = 100; K = 80; N = 512; C = 1; M = 5; G = 5; Y = 1.5$. Results are listed in Tables 10 and 11.

With $N = 512$ the GPU is twice as fast as the CPU for different numbers of exercise dates. The number of exercise dates does not influence the speed-up factor of the GPU. This is because the algorithm can not be parallelized in time, as we use values at t_{m+1} to calculate values at t_m in the backward recursion procedure. This results in a recursion of $m = M - 1, \cdots, 1$ in the CUDA implementation.

Furthermore, as the number of early exercise dates increases, the GPU option values converge slower than the CPU, double precision, values. For instance, when $N = 1024$, the GPU option price for $M = 80$ is 28.932182, which resembles the reference value of 28.932234 closer than the value reported in Table 11.

7 Conclusions

The COS method is a highly efficient pricing method for both European and early–exercise options. It is a challenge to implement such an efficient method, which requires a small number of terms in the Fourier cosine ex-

pansion and a small number of exercise dates to approximate the value of an American option, efficiently on a GPU.

In this paper, we implemented COS algorithm on the GPU, and the GPU time and option values were compared to those obtained on the CPU. We optimized the GPU implementation, by splitting a vector and performing the summation in parallel to exploit the advantages of a GPU.

A highly satisfactory performance on the GPU is observed especially in the case of multiple strike European option computations. Then, we find speed-up factors ranging from 10 to 100, depending on the form of the characteristic function and on the number of strikes that is computed simultaneously.

Although computation on a GPU to date still exhibits several disadvantages, such as single precision arithmetic, a time-consuming memory transfer, and additional computations due to an unscaled inverse Fourier transform, it is a promising architecture for option pricing.

Whereas for the Heston model an analytic characteristic function is well-known, this is not the case for more complex hybrid stochastic models. Their characteristic functions need to be determined numerically by a Riccati ODE solver. Based on our results this is also a favorable exercise on a GPU. A GPU-based architecture may therefore serve very well as a calibration engine in option pricing.

References

[1] F. Fang and C.W.Oosterlee, A novel option pricing method for European options based on Fourier-cosine series expansions. *SIAM J. Sci. Comput.*, 31: 826-848, 2008.

[2] F. Fang and C.W.Oosterlee. Pricing Early-Exercise and Discrete Barrier Options by Fourier-Cosine Series Expansions. *Numerische Mathematik* 114: 27-62, 2009.

[3] R. Lord, F.Fang, F. Bervoets and C.W.Oosterlee. A fast and accurate FFT-based method for pricing early-exercise options under Lévy processes. *SIAM J. Sci. Comput.*, 30: 1678-1705, 2008.

[4] P.P. Carr and D.B.Madan. Option valuation using Fast Fourier Transformation. *J. Comp.Finance*, 2: 61-73,1999.

[5] NVIDIA Cuda Compute Unified Device Architecture, Programming Guide, Version 1.0, 2007.

[6] V.Surkov. Parallel Option Pricing with Fourier Space Time-stepping Method on Graphics Processing Units, 2008.

[7] L.A. Grzelak and C. W. Oosterlee, On The Heston Model with Stochastic Interest Rates *Delft University of Technology, Report 09-05.*, 2009.

[8] D. Duffie, J. Pan, and K. Singleton, Transform Analysis and Asset Pricing for Affine Jump-Diffusions. *Econometrica*, 68: 1343–1376, 2000.

[9] P. Carr, H. Geman, D.B. Madan, and M. Yor The fine structure of asset returns: An empirical investigation. *Journal of Business,* Vol. 75, no. 2, 2002.

[10] B.Zhang and C.W.Oosterlee, Option pricing with COS method on Graphics Processing Unit, *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium,* 25–29, May, 2009.

N	16	32	64
msec(cpu1)	0.337	0.388	0.506
msec(cpu2)	0.1032	0.1503	0.2270
max.abs.error	0.0059	9.1396e-08	1.4211e-14

Table 1: Comparison of cpu times between different CPUs,.

N	CPU(time)	GPU(time)	CPU(value)	GPU(value)
256	0.193. . .	0.182	27.974744	27.974733
1024	0.691. . .	0.433	27.974744	27.974733

Table 2: Comparison of time and precision for CPU and GPU implementation of the COS method for a single European option.

Number op strikes	value of K
3 strikes	$K = 80, 100, 120$
5 strikes	$K = 80, 90, 100, 110, 120$
9 strikes	$K = 80, 85, \cdots, 115, 120$
13 strikes	$K = 70, 75, \cdots, 125, 130$
17 strikes	$K = 60, 65, \cdots, 135, 140$
21 strikes	$K = 50, 55, \cdots, 145, 150$

Table 3: Vectors of strikes used in the numerical examples.

CGMY model				
	N	32	64	128
MATLAB	msec	0.413230	0.745590	1.388770
	max.abs.err	1.3409e-05	$< 10^{-14}$	$< 10^{-14}$
GPU	msec	0.141144	0.143051	0.152826
	max.abs.err	0.000027	0.000034	0.000034
Heston model				
	N	64	128	256
MATLAB	msec	1.206600	1.958680	3.873950
	max.abs.err	4.2839e-04	2.2218e-08	$< 10^{-14}$
GPU	msec	0.154972	0.159979	0.182867
	max.abs.err	0.000534	0.000104	0.000104

Table 4: Convergence and maximum absolute error when pricing a vector of 5 strikes.

CGMY model				
	N	32	64	128
MATLAB	msec	1.335130	2.690250	5.340340
	max.abs.err	1.3409e-05	$< 10^{-14}$	$< 10^{-14}$
GPU	msec	0.154018	0.169992	0.200987
	max.abs.err	0.000053	0.000053	0.000053
Heston model				
	N	64	128	256
MATLAB	msec	3.850890	7.703350	15.556240
	max.abs.err	6.0991e-04	2.7601e-08	$< 10^{-14}$
GPU	msec	0.177860	0.209093	0.333786
	max.abs.err	0.000534	0.000144	0.000144

Table 5: Convergence and maximum absolute error when pricing a vector of 21 strikes.

Heston model				
	N	64	128	256
MATLAB	msec	37.9491	50.5196	81.1083
	max.abs.err	4.2848e-04	8.4949e-08	1.0650e-07
GPU	msec	1.091957	1.121998	1.253843
	max.abs.err	0.000443	0.000013	0.000013

Table 6: Convergence and maximum absolute error for 5 strikes, characteristic function obtained by RK4.

Heston model				
	N	64	128	256
MATLAB	msec	84.9870	145.0005	268.2299
	max.abs.err	6.0979e-04	1.5607e-07	1.2847e-07
GPU	msec	1.228094	1.402855	2.689838
	max.abs.err	0.000611	0.000037	0.000037

Table 7: Convergence and maximum absolute error for 21 strikes, characteristic function obtained by RK4.

N	CPU(time)	GPU(time)
256	13.15...	11.02
512	24.69...	13.69
1024	46.65...	27.34

Table 8: Comparison between CPU and GPU time for the CGMY model, for a Bermudan option; different numbers of terms in the Fourier cosine expansion.

N	CPU(value)	GPU(value)
256	28.829781987399432	28.829739
512	28.829781987399425	28.829721
1024	28.829781987399404	28.829756

Table 9: Precision on the GPU, Bermudan put option, for different numbers of terms in the Fourier cosine expansion.

M	CPU(time)	GPU(time)
20	51.04...	26.09
40	104.00...	50.88
80	210.20...	100.54

Table 10: Comparison of CPU and GPU times for the CGMY model, Bermudan option with a different numbers of exercise dates.

M	CPU(value)	GPU(value)
20	28.888713582335640	28.888538
40	28.917953599279208	28.917654
80	28.932234254713762	28.931826

Table 11: Precision on the GPU, Bermudan put with different numbers of exercise dates.