Journal of Computational Information Systems 10: 2 (2014) 763–771
Available at http://www.Jofcis.com

Parallel Algorithm for BSDEs Based High Dimensional American Option Pricing on the GPU [⋆]

Ying PENG [1], Hui LIU [1], Shuzhen YANG [2], Bin GONG [1,*]

[1]*School of Computer Science and Technology, Shandong University, Jinan 250101, China*

[2]*School of Mathematics, Shandong University, Jinan 250100, China*

Abstract

In this paper, we explore the opportunity for solving high dimensional Backward Stochastic Differential Equations (BSDEs) on the GPU with application in high dimensional American option pricing. A Least Square Monte Carlo method based numerical algorithm for solving the BSDEs is studied and summarized in four phases. For the parallel GPU algorithms of different phases, multiple factors which affect the performance, such as the task allocation and data store/access strategies and the thread synchronization features, are taken into consideration. Experimental results for different dimensionality show a maximum speedup of 75 for the backward calculation phase and that of 29 for the global algorithm compared with the CPU serial version.

Keywords: Parallel Algorithm; Backward Stochastic Differential Equations (BSDEs); Option Pricing; GPU

1 Introduction

In the field of financial engineering, the pricing of high dimensional American options emerges as a challenging issue. Although different numerical algorithms are proposed to solve this intractable problem in polynomial time, large amount of computation is still required. As any delay of time or information can lead to huge economic loss in financial markets, parallel computing is applied to this computational challenging area. Under different parallel environments, parallel algorithms with different numerical models have been studied and proposed. Wan [1] proposed two parallel approaches with the low discrepancy mesh method. Experimental tests were performed on a shared memory machine of 8 processors and demonstrated near optimal speedup for 3 dimensional options. On the heterogeneous supercomputer Deep Comp 7000, Chang [2] performed large scale parallel simulations based on the stochastic mesh method for the options with a maximum of 7 dimensions. Abbas-Turki [3] implemented parallel algorithms on the GPU by utilizing the Least Square Monte-Carlo (LSM) method. However, the experiments were performed only for 1 and 4

[⋆]Project supported by the Provincial Science and Technology Development Project (No. 2009GG10001001).
[*]Corresponding author.
Email address: gb@sdu.edu.cn (Bin GONG).

DOI: 10.12733/jcis9620
January 15, 2014

dimensional American option pricing, and no comparison was made between the CPU and the GPU versions for the 4 dimensional cases. In this paper, we develop a parallel algorithm based on solving high dimensional Backward Stochastic Differential Equations (BSDEs in short). The BSDEs are proofed to be very useful theoretical tools to deal with many financial problems ranging from option pricing to risk management. However, in our knowledge, among the parallelization works based on solving BSDEs for high dimensional American option pricing, only Labart [4] proposed a parallel algorithm which was efficiently used in cluster environment. As in our previous work [5], the GPU-based parallel algorithms demonstrated good performance for the computation of one dimensional BSDEs, we explore the opportunity for solving high dimensional BSDEs on the GPU with application in high dimensional American option pricing. The parallel algorithm is implemented with C and CUDA.

The reminder of this paper is organized as follows: Section 2 describes the numerical algorithm for solving high dimensional BSDEs with application in option pricing. Section 3 presents the parallel algorithm on the GPU. The experimental results and analysis are shown in Section 4 and Section 5 gives the conclusion.

2 Numerical Algorithm

The pricing of a multi-dimensional American option could be boiled down to solving a reflected BSDE, as shown in Eq. (1):

$$\begin{cases} X_t = X_0 + \int\limits_0^t \mu(s, X_s)ds + \int\limits_0^t \sigma(s, X_s)dW_s; \\ Y_t = g(X_T) + \int\limits_t^T f(s, X_s, Y_s, Z_s)ds - \int\limits_t^T Z_s dW_s + K_T - K_t; \\ Y_t \geq h(t, X_t), \int\limits_0^T [Y_t - h(t, X_t)]dK_t = 0. \end{cases} \tag{1}$$

Where $X_0 = (X_0^1, \ldots, X_0^d)^T$ represents the initial asset price, $X_t = (X_t^1, \ldots, X_t^d)^T$ represents the d-dimensional asset price on time t, $W_t = (W_t^1, \ldots, W_t^d)^T$ is the Brownian motion, $Z_t = (Z_t^1, \ldots, Z_t^d)$ is the price of a d-dimensional portfolio which helps to hedge the risk, Y_t represents the option price. (X_t, Y_t, Z_t) is the solution of BSDE (1). When X_t obeys the geometric Brownian motion, we get $\mu(t, X_t) = \mu X_t$ and $\sigma(t, X_t) = \sigma X_t$, where μ is a d-dimensional column vector, σ is a $d * d$ volatility matrix. $h(t, X_t)$ represents the payoff when the option is executed on time t, K is the time value. $f(\cdot)$ is the generation function of the BSDE, here we consider the form of Eq. (2) where R and r represents respectively the lending rate and the deposit rate.

$$f(t, X_t, Y_t, Z_t) = -rY_t - \sum_{i=1}^{d} (\mu^i - r)\Big((\sigma^{-1})^T Z_t\Big)^i + \left(Y_t - \sum_{i=1}^{d} \Big((\sigma^{-1})^T Z_t\Big)^i\right)^- (R - r) \tag{2}$$

In order to solve the BSDE (1), we adopt the numerical algorithm of Zhang [6] which is based on the Least Square Monte-Carlo (LSM) method. The time interval $[0, T]$ is evenly divided into

N time steps. Then the discrete form of BSDE (1) could be derived by:

$$\begin{cases} X_{t_{i+1}}^N = X_{t_i}^N + \mu(t_i, X_{t_i}^N)h + \sigma(t_i, X_{t_i}^N)\Delta W_{t_{i+1}}; \\ Z_{t_i}^N = \frac{1}{h}E_{t_i}\left[Y_{t_{i+1}}^N \Delta W_{t_{i+1}}\right]; \\ \tilde{Y}_{t_i}^N = E_{t_i}\left[Y_{t_{i+1}}^N\right] + f(t_i, X_{t_i}^N, Y_{t_{i+1}}^N, Z_{t_i}^N)h; \\ Y_{t_i}^N = \max(\tilde{Y}_{t_i}^N, h(t_i, X_{t_i}^N)); \\ K_{t_0}^N \triangleq 0; K_{ti}^N = \sum_{j=0}^{i-1}\left[Y_{t_i}^N - \tilde{Y}_{t_i}^N\right]. \end{cases} \tag{3}$$

Where $X_{t_0}^N \triangleq X_0$, $h \triangleq T/N$, $t_i = i * h$, $\Delta W_{t_{i+1}} \triangleq W_{t_{i+1}} - W_{t_i}$, $E_{t_i}[X]$ is the conditional expectation of X. For better and easier description, we use respectively $Z_{i,j}$, $Y_{i,j}$, $X_{i,j}$ instead of $Z_{t_i}^N$, $Y_{t_i}^N$, $X_{t_i}^N$, where i is the time step, j represents the simulation path of the Monte-Carlo method. The forward process X is simulated by the Monte-Carlo simulations with Eq. (4), which is equivalent to the discrete form for solving X in Eq. (3):

$$X_{i,j}^m = X_0^m \exp\{[\mu_m - \sum_{k=1}^{D}\frac{(\sigma_{mk})^2}{2}]t_i + \sum_{k=1}^{D}\sigma_{mk}W_{i,j}^k\}. \tag{4}$$

Where $m = 1, \cdots, d$ represents the dimension label, $W_{i,j}^k = \sum_{n=1}^{i}\Delta W_{n,j}^k = \sum_{n=1}^{i}\xi_{n,j}^k\sqrt{\Delta t}$ ($\xi_{n,j}$ is a d dimensional random number obeying $N(0,1)$ distribution). For calculating Y_t and Z_t in the backward process, we can specify a sequence of L basis functions $\phi_1, \phi_2, \cdots, \phi_L$. Then on each time step, the estimations $\widehat{Z}_{i,j}$, $\widehat{Y}_{i,j}$ of $Z_{i,j}$ and $Y_{i,j}$ can be calculated with Eq. (5) and Eq. (6):

$$\widehat{Z}_{i,j} = \sum_{l=1}^{L}\alpha_{i,l}\phi_l(X_{i,j}), 1 \le i \le N-1. \tag{5}$$

Where $\{\alpha_{i,1}, \alpha_{i,2}, ..., \alpha_{i,L}\} = \inf_{\alpha}\frac{1}{M}\sum_{j=0}^{M-1}\left|\sum_{l=1}^{L}\alpha_{i,l}\phi_l(X_{i,j}) - \frac{1}{h}Y_{i+1,j}\Delta W_{i+1,j}\right|^2$.

$$\widehat{Y}_{i,j} = E_{t_i}[\widehat{Y}_{i+1,j}] + f(t_i, X_{i,j}, \widehat{Y}_{i+1,j}, \widehat{Z}_{i,j})h, 1 \le i \le N-1. \tag{6}$$

Where $E_{t_i}[\widehat{Y}_{i+1,j}] = \sum_{l=1}^{L}\beta_{i,l}\phi_l(X_{i,j})$, $\{\beta_{i,1}, \beta_{i,2}, ..., \beta_{i,L}\}$
$= \inf_{\beta}\frac{1}{M}\sum_{j=0}^{M-1}\left|\sum_{l=1}^{L}\beta_{i,l}\phi_l(X_{i,j}) - Y_{i+1,j}\right|^2$.

More specifically, consider the following matrix:

$$\Phi = \begin{bmatrix} (\phi_1, \phi_1) & ...... & (\phi_1, \phi_L) \\ ...... & ...... & ...... \\ (\phi_L, \phi_1) & ...... & (\phi_L, \phi_L) \end{bmatrix}, \tag{7}$$

Where $(\phi_a, \phi_b) = \sum_{j=0}^{M-1}\phi_a(X_{i,j})\phi_b(X_{i,j})$. Then the matrix α and β of the least square factors can be derived by Eq. (8) and Eq. (9):

$$\alpha = \begin{bmatrix} \alpha_{i,1} & \alpha_{i,2} & ... & \alpha_{i,L} \end{bmatrix}^T = \Phi^{-1}\begin{bmatrix} (\phi_1, Z) & (\phi_2, Z) & ... & (\phi_L, Z) \end{bmatrix}^T, \tag{8}$$

$$\beta=\begin{bmatrix} \beta_{i,1} & \beta_{i,2} & \dots & \beta_{i,L} \end{bmatrix}^T=\Phi^{-1}\begin{bmatrix} (\phi_1,Y) & (\phi_2,Y) & \dots & (\phi_L,Y) \end{bmatrix}^T. \tag{9}$$

Where $(\phi_a, Z) = \frac{1}{h}\sum_{j=0}^{M-1}\phi_a(X_{i,j})Y_{i+1,j}\Delta W_{i+1,j}$, $(\phi_a, Y) = \sum_{j=0}^{M-1}\phi_a(X_{i,j})Y_{i+1,j}$.
Finally, the values of Y_t and Z_t on time step 0 can be given by:

$$Z_{0,j} = \frac{1}{M}\sum_{j=0}^{M-1} Y_{1,j}\Delta W_{1,j}; \tag{10}$$

$$Y_{0,j} = \frac{1}{M}\left(\sum_{j=0}^{M-1} Y_{1,j} + f(0, X_{0,j}, \widehat{Y}_{1,j}, \widehat{Z}_{0,j})h\right). \tag{11}$$

The terminal condition $Y_{N,j}$ is given in Eq. (12):

$$f(x) = \begin{cases} \max(\max(X^1_{N,j}, \cdots, X^d_{N,j}) - K, 0) & \text{for call options;} \\ \max(K - \max(X^1_{N,j}, \cdots, X^d_{N,j}), 0) & \text{for put options.} \end{cases} \tag{12}$$

The sequence of the basis functions is chosen according to [7] as follows:
$\left\{1, X^1_{i,j}, X^2_{i,j}, ..., X^d_{i,j}, (X^1_{i,j})^2, (X^2_{i,j})^2, ..., (X^d_{i,j})^2, (\max(X^1_{i,j}, X^2_{i,j}, ...X^d_{i,j}) - K)^+\right\}$
Then we get the number of basis functions $L = 2 * d + 2$.

According to the whole calculation process, we summarize the serial algorithm in 4 phases:

1. Path generation: Generate the random numbers on the Monte-Carlo simulation paths for all the time steps.

2. Terminal condition calculation: For each path j of the Monte-Carlo simulation, calculate the terminal conditions $Y_{N,j}$ on time step N. Calculate $\frac{1}{h}Y_{N,j}\Delta W_{N,j}$ and $X_{N-1,j}$ in advance for time step $N-1$.

3. Backward calculation: From time step $i = N-1$ to $i = 1$, estimate $\widehat{Y}_{i,j}$ and $\widehat{Z}_{i,j}$ on each simulation path j with the LSM method. The calculation on each time step can be further divided into three sub-parts:

   (1). Matrix Φ calculation: Calculate matrix Φ with Eq. (7).

   (2). Coefficient calculation: Calculate the LSM coefficients α and β with Eq. (8), (9).

   (3). Solution estimation: Calculate $\widehat{Z}_{i,j}$ and $\widehat{Y}_{i,j}$ with Eq. (5), (6), get the value of $Y_{i,j}$ with $Y_{i,j} = \max(\widehat{Y}_{i,j}, h(t_i, X_{i,j}))$, then calculate the values of $\frac{1}{h}Y_{i,j}\Delta W_{i,j}$ and $X_{i-1,j}$.

4. Final solution calculation: Calculate the final solution on time step 0 with Eq. (10), (11).

For the implementation of the serial algorithm, we use one dimensional arrays X, Y, Z to store respectively the values of $X_{i,j}$, $\widehat{Y}_{i,j}$, and $\frac{1}{h}Y_{i,j}\Delta W_{i,j}$ on the current time step, and the corresponding array length is $M * d$, M, $M * d$ respectively. Then the array elements are updated with the proceeding of the backward calculation, in this way the storage space can be saved.

3 Parallel Algorithm on the GPU

We take the example of $M = 131072$, $N = 300$ and measure the time distribution of different phases in the serial algorithm when vary the dimension d with $d = 3, 5, 7, 9$. The running time results show that the backward calculation phase occupies more than 95% of the serial time, while the final solution calculation phase only occupies less than 0.002%. Thus, in the parallel algorithm, we consider to perform the calculation of the first three phases with GPU, and then upload the results to CPU to perform the final solution calculation phase.

3.1 Backward calculation phase

As the most time-consuming part in the serial algorithm is the backward calculation phase, we emphasize on the parallelization of this phase. As described in Section 2, the backward calculation phase is composed of $N-1$ iterations. Assuming the time step that the current iteration locates on to be i, then the GPU acceleration strategies of different sub-parts are designed and implemented for an arbitrary time step i $(1 \leq i \leq N-1)$.

1. Matrix Φ calculation

The size of matrix Φ is $L * L$, where L is the number of basis functions. In our option pricing application, $L = 2 * d + 2$. Thus the matrix size is relatively small $(O(d^2))$. As known from Eq. (7), for the calculation of an element $(\phi_a, \phi_b)(1 \leq a \leq L, 1 \leq b \leq L)$ in Φ, we need to accumulate the values of $\phi_a(X_{i,j})\phi_b(X_{i,j})$ (j is the path label) on all the M paths, while the calculation on each path is independent. Furthermore, M should be enough large to keep the level of precision. Therefore, we focus on the data partition of each element $(\phi_a, \phi_b) = \sum_{j=0}^{M-1} \phi_a(X_{i,j})\phi_b(X_{i,j})$ instead of the whole matrix, i.e. the GPU kernel takes charge of the calculation of an arbitrary element (ϕ_a, ϕ_b) in Φ.

We assume the number of blocks and the number of threads in each block to be respectively $PhiBlocks$ and $PhiThreads$. Then each block is in charge of the summation of $M/PhiBlocks$ paths, and each thread is in charge of $M/(PhiBlocks * PhiThreads)$ paths. Firstly each thread accumulate the value of (ϕ_a, ϕ_b) on all the paths it takes charge of, then the sum reduction is performed on each block to get the partial summation of (ϕ_a, ϕ_b). Finally, the reduction results on each block is uploaded to the array $PartialSum$ on the CPU, and we get the value of (ϕ_a, ϕ_b) by summing up all the elements values in array $PartialSum$. During the process of sum reduction on every block, the shared memory is utilized to store the interim results. Fig. 1 shows an example of the sum reduction process on each block when $PhiThreads = 8$. Besides, the loop unrolling technique is used to further enhance the performance.

2. Coefficient calculation

The main work of the coefficient calculation sub-part is to calculate ϕ^{-1} and the matrices α, β. As the size of the matrices are small and the calculation does not take much time, we perform this sub-part on the CPU and download the values of α, β to the GPU to perform the solution estimation sub-part.

3. Solution estimation

The solution estimation is the most time-consuming sub-part in the backward calculation phase. For the M simulation paths on time step i, the values of $\widehat{Y}_{i,j}$ and $\widehat{Z}_{i,j}$ are estimated and utilized to get the solution $Y_{i,j}$, afterward the values of $\frac{1}{h}Y_{i,j}\Delta W_{i,j}$ and $X_{i-1,j}$ are calculated. As there

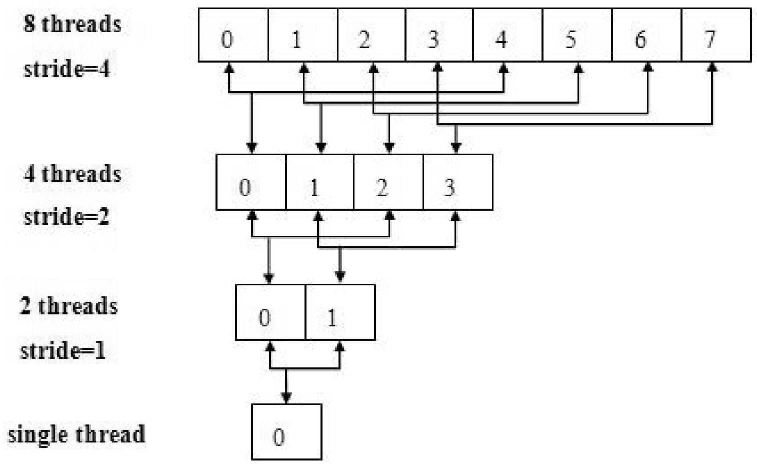

Fig. 1: Parallel sum reduction process in the matrix Φ calculation sub-part

is no dependency between the calculations of different paths, a naive data distribution strategy is to map a thread with multiple paths in a cyclic allocation way. However, that will cause the discontinuity of data access to the arrays Z and X within a half warp. Thus the memory coalescing requirement can not be satisfied, and that will greatly affect the performance of the GPU kernel. Therefore, we utilize 2-dimensional thread blocks, and each thread is responsible for the calculation of a single dimension of the arrays Z and X under the corresponding paths. Fig. 2 shows the data mapping strategy which satisfies the memory coalescing requirement.

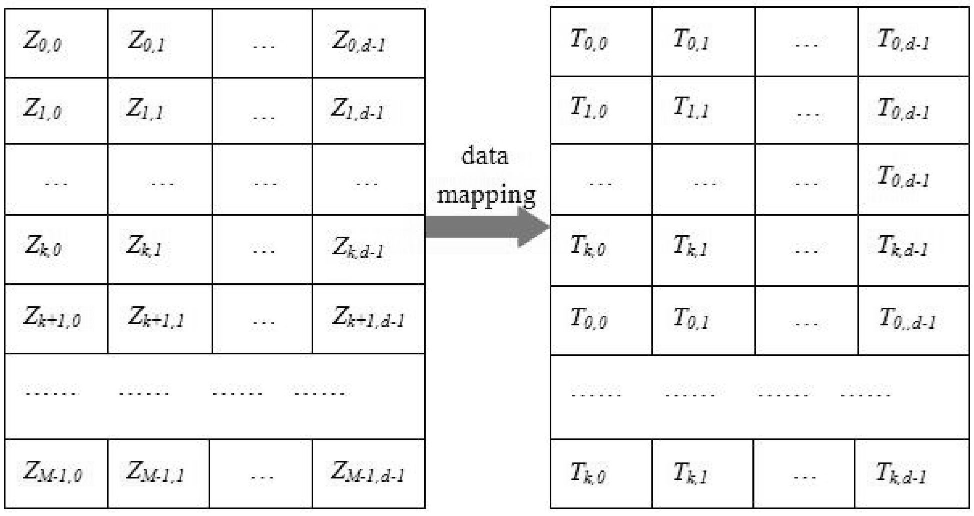
$Z_{0,0}$ $Z_{0,1}$ … $Z_{0,d-1}$
$Z_{1,0}$ $Z_{1,1}$ … $Z_{1,d-1}$
$Z_{k,0}$ $Z_{k,1}$ … $Z_{k,d-1}$
$Z_{k+1,0}$ $Z_{k+1,1}$ … $Z_{k+1,d-1}$
$Z_{M-1,0}$ $Z_{M-1,1}$ … $Z_{M-1,d-1}$
data
mapping
$T_{0,0}$ $T_{0,1}$ … $T_{0,d-1}$
$T_{1,0}$ $T_{1,1}$ … $T_{0,d-1}$
$T_{0,d-1}$
$T_{k,0}$ $T_{k,1}$ … $T_{k,d-1}$
$T_{0,0}$ $T_{0,1}$ … $T_{0,,d-1}$
$T_{k,0}$ $T_{k,1}$ … $T_{k,d-1}$

Fig. 2: Data mapping strategy in the solution estimation sub-part

3.2 Other phases

1. Path generation phase

The main work of this phase is to generate random numbers required by all the simulation paths with $N(0,1)$ distribution. Here we generate random numbers on the GPU with the Mersenne Twister (MT) pseudo random number generator [8]. The MT method can efficiently generate random numbers with a long period ($2^{19937}-1$) and with good dimensional uniformity. Moreover, it is easy to be parallelized. The generated random numbers are stored in the global memory on the GPU, so that they can be conveniently utilized during the terminal condition calculation and the backward calculation phases.

2. Terminal condition calculation phase

In this phase, for each simulation path, the values of $X_{N,j}$, $Y_{N,j}$, $\frac{1}{h}Y_{i,j}\Delta W_{i,j}$ and $X_{N-1,j}$ are calculated. As the calculation of each path is independent to the others, and the calculation process is similar with the solution estimation sub-part in the backward calculation phase, a similar data mapping strategy to that of the solution estimation sub-part is used.

4 Experimental Results

We implement the CPU serial algorithm and the GPU algorithm with C and CUDA. Running time tests are performed on a LANGCHAO NF5588 GPU server, with the CPU type: Xeon E5620 (2.4GHz), memory: 24GB DDR3, and the GPU type: NVIDIA Tesla C2050. The 32 bit single precision floating point numbers are used in the experimental tests. The parameters of the GPU kernels of different phases (sub-parts) are shown in Table 1, where *blocksize* is the number of thread blocks, *threadsize* is the number of threads in each block, d is the dimensionality of the BSDE.

Table 1: Parameters valuation in each GPU kernel

GPU kernel	*blocksize*	*threadsize*
Matrix Φ calculation	64	128
Solution estimation	256	32*d
Terminal condition Estimation	256	32*d

Fig. 3 shows the CPU serial time with the changes of simulation paths M and time steps N when $d = 3, 5, 7, 9$. The horizontal axis represents the value of M_N, for example, 65536_50 represents the case when $M = 65536$, $N = 50$. In Fig. 3, when the number of simulation paths and time steps increase, the CPU serial time rise up steadily. Moreover, the increase of dimension d can also lead to the augmentation of the serial execution time. When M_N = 131072_300, the executation time with $d = 3$ is only 290 seconds, while that of $d = 9$ is nearly 2500 seconds. Actually, when the dimension d increases, the number of simulation paths M and time steps N should increase correspondingly to keep the level of precision.

Fig. 4 shows the speedup profiling of the GPU parallel algorithm. We can observe that with the increase of M, N and d, the speedup of the parallel algorithm rise up steadily, and achieves 29 when $d = 7$, $M = 131072$, $N = 300$.

As the influence of the backward calculation phase to the global algorithm is the most important, and the solution estimation sub-part occupies most the CPU serial time of this phase, we show in Fig. 5 the speedups of the solution estimation sub-part. We can see that with the increase of M and N, the speedup for the solution estimation sub-part rise up very slowly. The possible reason is that the amount of computation tasks allocated to each thread is saturated. On the other hand, with the increase of d, the speedup rises up significantly. When M_N = 131072_300,

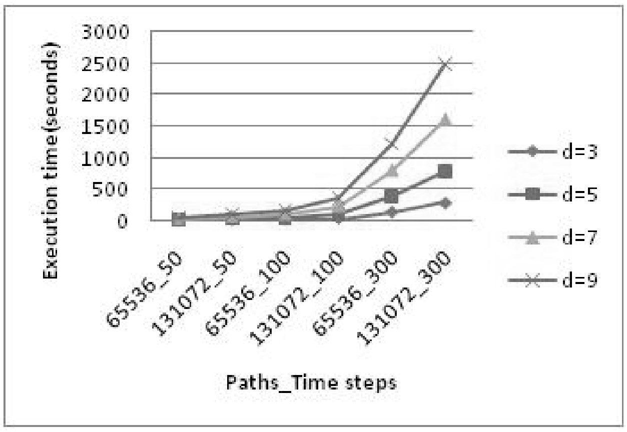

Fig. 3: CPU serial time

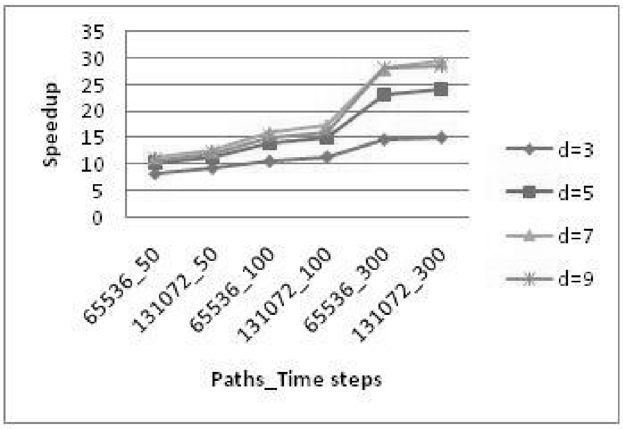

Fig. 4: Speedups with d=3, 5, 7, 9

the speedup for $d = 3$ is only 25.5, while that of $d = 9$ achieves 75.3. That is because the total number of threads increases when increasing d (each thread block owns $32 * d$ threads).

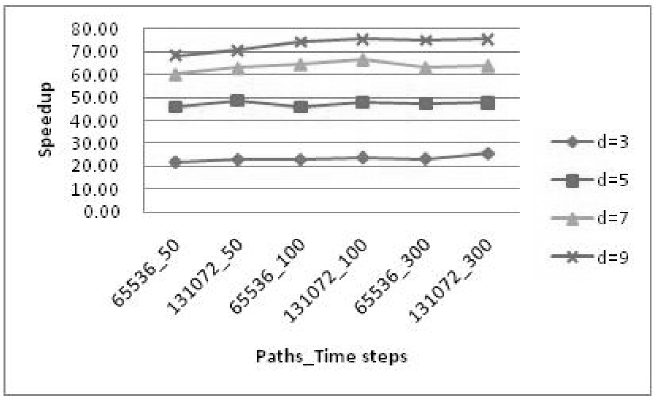

Fig. 5: Speedups for the solution estimation sub-part

Moreover, when increasing the number of GPU threads, the GPU time decreases greatly, thus the execution time of the coefficient calculation sub-part which is performed on the CPU become more significant. In order to further enhance the performance, the parallel strategy for the coefficient calculation sub-part, especially the parallel matrix inversion method [9, 10] will be further studied in the future.

5 Conclusion

In this paper, in order to solve the parallelization problem for high dimensional American option pricing, we propose a parallel algorithm based on solving the high-dimensional BSDEs on the GPU. According to the computation time and characteristics of different phases of the numerical algorithm, we design and implement the GPU-based acceleration algorithms respectively for the path generation phase, the terminal condition calculation phase and the backward calculation phase, and perform the final solution phase on the CPU. For the GPU acceleration algorithms in different phases, multiple factors which affect the parallel performance, such as the task allocation strategy, the thread synchronization feature and the data store/access strategy, etc. are taken into consideration, and the general performance is greatly enhanced.

References

[1] Justin W. L. Wan, Kevin Lai, A Parallel Quasi-Monte Carlo Approach to Pricing American Options on Multiple Assets, in: International Journal of High Performance Computing and Networking 4, pp. 321-330, 2006.

[2] Chang Hong-xu, LU Zhong-hua, and CHI Xue-bin, Large-scale Parallel Simulation of High-dimensional American Option Pricing, in: Journal of Algorithms & Computational Technology, Vol. 6, No. 1, pp. 1-16, 2012.

[3] L. A. Abbas-Turki and B. Lapeyre, American Options Pricing on Multi-Core Graphic Cards, in: International Conference on Business Intelligence and Financial Engineering, BIFE 2009, pp. 307-311, 2009.

[4] Celine Labart, Jerome Lelong, A Parallel Algorithm for solving BSDEs, in: Application to the pricing and hedging of American options. ArXiv: math. PR/1102. 4666, preprint, http://arxiv.org/abs /1102. 4666, 2011.

[5] Ying Peng, Bin Gong, Hui Liu, Bin Dai, Option Pricing on the GPU with Backward Stochastic Differential Equation, in: 4th International Symposium on Parallel Architectures, Algorithms and Programming, pp. 19-23, PAAP 2011.

[6] Jianfeng Zhang, A Numerical Scheme for BSDEs, in: The Annals of Applied Probability, Vol. 14, No. 1, pp. 459-488, 2004.

[7] Jose M. Villalobos, Monte-Carlo Methods for Forward Backward Stochastic Differential Equations in High Dimensions, in: Doctoral Dissertation, University of Southern Califonia, 2007.

[8] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator, in: ACM Transactions on Modeling and Computer Simulation, 8 (1), pp. 3-30, 1998.

[9] Kaiqi Yang, Yubai Li, Yijia Xia: A Parallel Method for Matrix Inversion Based on Gauss-jordan Algorithm, in: Journal of Computational Information Systems 9 (14), pp. 5561-5567, 2013.

[10] Pablo Ezzatti, Enrique S. Quintana-Orti, Alfredo Remon: High Performance Matrix Inversion on a Multi-core Platform with Several GPUs, in: Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 87-93, 2009.