J. KSIAM Vol.20, No.4, 277–293, 2016 http://dx.doi.org/10.12941/jksiam.2016.20.277

A PRICING METHOD OF HYBRID DLS WITH GPGPU

YEOCHANG YOON[1], YONSIK KIM[2], AND HYEONG-OHK BAE[3†]

[1]SHINHAN INVESTMENT CORP.
E-mail address: derivatives86@gmail.com

[2]FINANCIAL RESERACH CENTER, FNPRICING INC.
E-mail address: yskim@fnpricing.com

[3]DEPARTMENT OF FINANCIAL ENGINEERING, AJOU UNIVERSITY, REPUBLIC OF KOREA
E-mail address: hobae@ajou.ac.kr

ABSTRACT. We develop an efficient numerical method for pricing the Derivative Linked Securities (DLS). The payoff structure of the hybrid DLS consists with a standard 2-Star step-down type ELS and the range accrual product which depends on the number of days in the coupon period that the index stay within the pre-determined range. We assume that the 2-dimensional Geometric Brownian Motion (GBM) as the model of two equities and a no-arbitrage interest model (One-factor Hull and White interest rate model) as a model for the interest rate. In this study, we employ the Monte Carlo simulation method with the Compute Unified Device Architecture (CUDA) parallel computing as the General Purpose computing on Graphic Processing Unit (GPGPU) technology for fast and efficient numerical valuation of DLS. Comparing the Monte Carlo method with single CPU computation or MPI implementation, the result of Monte Carlo simulation with CUDA parallel computing produces higher performance.

1. INTRODUCTION

In recent years, the computation speed plays a vital role in the financial world. It is important to predict risk factors, to hedge other financial products and to compute financial derivatives with complicated structures. As the trading volume of derivatives has been increased steadily, the computation speed has been more seriously concerned. In this sense, Fei [1] has studied pricing of the simple stock option using Monte Carlo simulation by using CUDA. And in [2], they have parallelized the Monte Carlo method on GPUs for the calibration of the parameters in SABR stochastic volatility modes.

In this article, we compute CD-Equity Hybrid Derivative Linked Securities (DLS), which is issued in [3]. This is a hybrid combination between the Equity Linked Securities (ELS) with

Received by the editors November 11 2016; Revised November 30 2016; Accepted in revised form November 30 2016; Published online December 2 2016.

2010 *Mathematics Subject Classification.* 97M30, 91G80, 62P05, 65C05.

Key words and phrases. hybrid DLS, GPGPU, geometric Brownian motion, CUDA, Monte Carlo, MPI, ELS.

† Corresponding author.

two underlying stock price indices and the certificate of deposit (CD) (91 days) rates single-range accrual DLS. This is why it is called the Hybrid DLS. In order to evaluate this financial product, we assume that stock prices follow 2-dimensional Geometric Brownian Motion respectively, and that CD (91days) rates follow the One-factor Hull-White short-rate model. We use the Monte Carlo method to simulate the stock price indices model, and the Euler scheme for the interest-rate model.

To show the performance of parallel computation methods, we attempt the parallel computing using the MPI library and the General Purpose computing on Graphics Processing Units (GPGPU). The recent advent of usage of the Compute Unified Device Architecture (CUDA), which is one of the GPGPU technologies, provides a high efficiency in computing. CUDA, introduced by NVidia in 2007, is suitable and natural for implementing independent paths simulation using thousands of threads. As a popular numerical tool for evaluation of financial derivatives, the Monte Carlo method has been widely used in computational financial engineering to solve problems of which closed-form solution is not feasible or is impossible to obtain. However, this method is computationally too costly due to a large number of trials.

The crucial key of the Monte-Carlo method is the random number generators (RNGs) that provide to each trial independently. The random number generation is usually the most time-consuming procedure. To overcome this drawback, we need to consider massive parallel computation essentially when we generate random numbers. Apart from the standard libraries, CUDA also provides several useful libraries, which can improve the programming efficiency significantly. We especially use the `cuRAND` library, which provides facilities that efficient generation of high-quality pseudo-random or quasi-random numbers. We focus on the pseudo-random numbers. `cuRAND` consists of libraries on the host side (CPU) and a device side (GPU). We use the device Application Programming Interface (API), which includes functions for various random sequences.

We finally compare the elapsed time of serial computation and two of parallel computing methods. Our final goal is to get good performance results when using GPU with CUDA. As the results, we may conclude CUDA computing is more efficient than parallel computing by MPI library. In a number of respects, CUDA enable us to compute for complicated options and Over-the-Counter derivative products in a just few seconds. This computational speedup allows us to run more simulation paths that increase the confidence.

In Section 2, hybrid DLS as an example of financial product is explained in detail. In Section 3, we introduce the GPGPU computing with CUDA. In Section 4, we provide numerical examples for the verification of the CUDA performance. Finally, in Section 5, we conclude that the GPGPU computing with CUDA is an effective and promising substitute for the high performance computing in the financial industry.

2. HYBRID DERIVATIVE LINKED SECURITIES

In this section, we first introduce the concept of the hybrid DLS, and then explain a detailed products structure, payoff conditions and underlying assets mathematical models.

2.1. **Explanation of Hybrid DLS.** ELS payoff depends on movements of underlying assets such as a single stock or index, a multiple stocks or indices. Similarly, DLS payoff on investments is determined by movements of underlying assets such as interest rates, FX rates, oil prices, typically West Texas Intermediate (WTI), credit rating and actual assets, and so on. DLS is available to be made in a variety of products payoff conditions. As an example of hybrid DLS, we compute the price of CD-Equity Duet 387th DLS issued by KDB Daewoo securities in 2011 [3]. This is a hybrid combination between ELS linked with two underlying stock price indices (KOSPI200 and HSCEI) and CD (Certificate of Deposits) 91 days rates range accrual DLS. This is why it is called the hybrid DLS. It has been given the right of using during 3-month exclusively by Korea Financial Investment Association.

A standard 2-dimensional step-down structured ELS is dependent upon the minimum rate of return between two underlying assets by checking every 6-month. But we use an average rate of return, not a minimum rate of return. In here, the rate of return means ratio of underlying asset to initial price.

And as another underlying asset, CD Range Accrual as a type of non-vanilla derivative product will be applied to decide final payoff. This product is structured that a coupon whose value accrues with the number of days a reference index stays within the pre-determined range. To enhance the marketability and to reduce the burden of new products development, many securities companies began issuing a variety of structured patented ELSs. However, we guess that except 2in1 ELS and Airbag ELS, most patented ELSs was not successful. Maybe it is because they have complex structures, which are difficult for investors to understand. CD-Duet hybrid DLS is not popular these days, also it is not traded any longer. This can be structured with or without an embedded call option. Also, the issuer will either pay the coupon or nothing. In other words, it looks a digital option. When the reference index trades outside of the pre-determined range, the coupon is not accrued. This has been once very popular among investors who hope that the reference index will stay within a pre-determined range. After early and terminal redemption conditions are decided, CD Range Accrual plays a role in final return. In explaining range accrual without mathematical formula, we have to observe the index value and memorize the number of observations within the given range.

Finally, based on ELS payoff, the more indices stay within the range, the higher we will get return. The payoff diagram of this hybrid DLS is shown in Figure 1.

2.2. **Mathematical Models.** We assume that underlying assets for 2-dimensional ELS follow GBM, and that the CD rate follows the One-factor Hull, and White interest-rate model as the no-arbitrage model and the mean-reverting short-rate model.

2.2.1. *The 2-dimensional Geometric Brownian Motion.* The 2-dimensional Monte-Carlo asset paths for ELS are generated by the following scheme:

$$S_1(t_{n+1}) = S_1(t_n)e^{(r-q-\frac{1}{2}\sigma_1^2)\Delta t+\sigma_1\sqrt{\Delta t}\Delta W_1}, \tag{2.1}$$

$$S_2(t_{n+1}) = S_2(t_n)e^{(r-q-\frac{1}{2}\sigma_2^2)\Delta t+\sigma_2\sqrt{\Delta t}(\rho_{12}\Delta W_1+\sqrt{1-\rho_{12}^2}\Delta W_2)}. \tag{2.2}$$

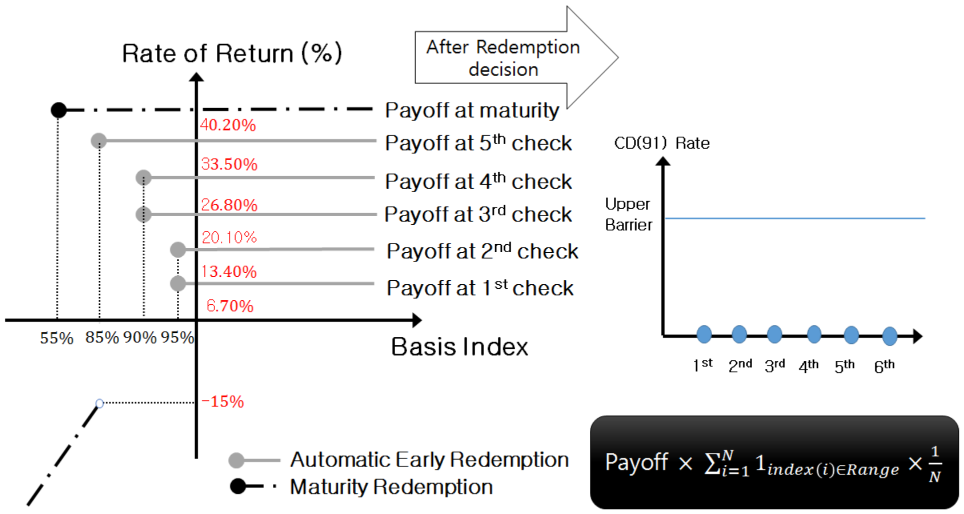

FIGURE 1. CD-Equity DLS Payoff Structure [3, 4]

In (2.1) and (2.2), stock indices stand for KOSPI 200 and HSCEI, respectively. We use constant risk-free interest rates, correlation between two stock indices, volatilities and zero dividend rates.

2.2.2. *The One-factor Hull and White Interest-rate Model.* In [5], an extended-Vaicek model has been introduced so that it fits both an initial term-structure of interest rates and interest-rate volatilities. The model is a no-arbitrage yield curve model. This means that it can reproduce the initial yield curve implied by bond prices. The Hull-White model is widely used in derivative pricing as well as in risk management filed. A main advantage of the Hull-White model is easy to get the distribution of future time interest rates if the initial yield curve and other parameters are all given. The instantaneous short-rate $r(t)$ (over a very short period of time) under this model is governed by the following dynamics under the risk-neutral measure. Note that this short-rate is not equal to interest rate in (2.1) and (2.2):

$$dr(t) = \Big(\theta(t) - \alpha(t)r(t)\Big)dt + \sigma(t)dW(t)^Q.$$

Unlike the equilibrium model, in the no-arbitrage model, the drift depends on time. This is why the shape of the initial zero curves govern the average path taken by the short-rate in the future. The parameters $\theta(t)$, $\alpha(t)$ and $\sigma(t)$ are deterministic functions of time. The function $\theta(t)$ is chosen so as to exactly fit an initial term-structure of interest rates observed in the market. The other two time-varying functions $\alpha(t)$ and $\sigma(t)$ enable the above model to satisfy the volatility structure at time zero. However, in [3] the same authors have proposed that the result from volatility term structure could be non-stationary and must be dealt with carefully. It is because some market sectors are less liquid. The future volatility structures implied by the above dynamics are likely to be unreliable and uninformative. We therefore focus on the following extension of the Vasicek model in [6] as a special case with positive constant parameters α and σ. As mentioned before, the function $\theta(t)$ is chosen so as to exactly fit to current term-structure of interest rates being observed in the market:

$$dr(t) = \Big(\theta(t) - \alpha r(t)\Big)dt + \sigma dW(t)^Q. \tag{2.3}$$

It can be rewritten as the following:

$$dr(t) = \alpha\Big(\frac{\theta(t)}{\alpha} - r(t)\Big)dt + \sigma dW(t)^Q.$$

Where, constant parameter α means the short-rate mean reversion speed, and if this value become greater, then the short-rate reverts faster. The other constant parameter σ means the short-rate standard deviation. And W is a Brownian motion under the risk-neutral measure. At time t, the short-rate reverts to $\frac{\theta(t)}{\alpha}$ with speed α. In this Hull and White One-factor model could generate negative interest. This model may not suitable in a low interest rate environment. As an alternative to avoid generating negative interest rate, the Black-Karasinski model could be a suitable candidate. We denote the market instantaneous forward rate at time 0 for the maturity T by $F^M(0,T)$:

$$F^M(0,T) = -\frac{\partial \ln P^M(0,T)}{\partial T}.$$

Let us assume that the instantaneous forward rate follows from the interest rates which are observed current market and that the zero-coupon bonds which are observed current market is denoted by $P^M(0,T)$. With $P^M(0,T)$ from the market discount factor at time 0 for the maturity T, in order to fit the model to the current interest rate term structure, we must have the function $\theta(t)$ which is analytically calculated by

$$\theta(t) = F_t^M(0,t) + \alpha F^M(0,t) + \frac{\sigma^2}{2\alpha}\Big(1 - e^{-2\alpha t}\Big). \tag{2.4}$$

Since the last term is quiet small, if we ignore this, the equation implies that the drift of the process at time t. The short-rate follows the slope of the initial instantaneous forward rate curve. In (2.3), the short-rates are represented in terms of a stochastic differential equation. So as to obtain an explicit expression for short-rates, we need to integrate (2.3) using a bit of trick. Under the Hull-White model, the short-rate at a future time follows a normal distribution. If the yield curve at time s is given, short-rates stochastic process is mathematically integrated at any time $s < t$:

$$r(t) = r(s)e^{-\alpha(t-s)} + \int_s^t e^{-\alpha(t-u)}\theta(u)du + \sigma\int_s^t e^{-\alpha(t-u)}dW(u)^Q. \tag{2.5}$$

Equation (2.5) can be rewritten:

$$r(t) = r(s)e^{-\alpha(t-s)} + A(t) - A(s)e^{-\alpha(t-s)} + \sigma\int_s^t e^{-\alpha(t-u)}dW(u)^Q,$$

where

$$A(t) = F^M(0,t) + \frac{\sigma^2}{2\alpha^2}\Big(1 - e^{-\alpha t}\Big)^2.$$

The short-rate $r(t)$ conditional on $\mathcal{F}_s$ has mean and variance, respectively, as follows:

$$E[r(t)|\mathcal{F}_s] = r(s)e^{-\alpha(t-s)} + A(t) - A(s)e^{-\alpha(t-s)},$$
$$Var[r(t)|\mathcal{F}_s] = \frac{\sigma^2}{2\alpha}\Big(1 - e^{-2\alpha(t-s)}\Big).$$

Generally, the processes of stock prices or interest rate are assumed a continuous-time stochastic process. However, when we simulate any SDEs for evaluation, it is important to discretize a continuous-time process into a discretized time version. Hence we adopt the explicit Euler scheme for a time discretization. This scheme is equivalent to approximating the integral terms using the left-point rule. We want to discretize the continuous version short-rate process by approximating the integrals

$$\delta r(t) = \Big(\theta(t) - \alpha r(t)\Big)\delta t + \sigma Z\sqrt{\delta t}, \tag{2.6}$$

where Z follows the standard normal distribution and r_0 from IRS(Interest Rate Swap) curve is defined as an initial value of interest rate scenario. And (2.6) can be rewritten as follows:

$$r(t+\delta t) = r(t) + \Big(\theta(t) - \alpha r(t)\Big)\delta t + \sigma Z\sqrt{\delta t}, \qquad r(0) = r_0.$$

We approximate $\theta(t)\delta t$ by (2.4):

$$\theta(t)\delta t = F_t^M(0,t+\delta t) - F_t^M(0,t) + \alpha F^M(0,t)\delta t + \frac{\sigma^2}{2\alpha}\Big(1 - e^{-2\alpha t}\Big)\delta t.$$

In fact, we need a couple of calibrated constant parameters σ, α so as to using the above scheme. These parameters are calibrated to fit this model as closely as possible to market data. To perform this process, usually main diagonal data in KRW Swaption quoted implied volatilities surface from the Black 76 model are used. With the volatility matrix, to fit the model we have to calculate the implied volatilities surface from the Hull and White swaption value. In practice, we fix the parameter α. Then we have to find the constant parameter of short rate standard deviation. It is decided to minimize the sum of squared errors. These errors mean that the simple difference between the KRW Swaption quoted implied volatilities surface from Black76 model and the swaption value which is calculated by the short-rate model. To solve the calibrated parameter, the following formula can be used:

$$\sigma = \min\sum_{i=1}^{n}\big(\sigma^{\text{Hull-White}}_{\text{Swaption}_i} - \sigma^{\text{Quoted}}_{\text{Swaption}_i}\big)^2.$$

The end value of the above summation, n is the total number of main diagonal data in Swaption matrix. However, we assumed the constant σ, not calibrated from market data. Finally, if we know Interest Rate Swap (IRS) curve as an initial term structure, then we know $F(0,t)$. If parameters are constant, then $\theta(t)$ can be calculated. With fix quiet small discretization interval, we generate short-rates inductively.

3. GPGPU Computing with CUDA

3.1. **Basic CUDA.** We use GPGPU with CUDA for parallel computing of the financial derivative. The parallel computing platform CUDA is one of the GPGPU technologies which can use such as C, C++ and FORTRAN codes, and is aimed to make more similar GPU programming. Due to these advantages of CUDA, it is widely used in various science fields. Before using CUDA, we need to equip with GPU that support CUDA. In CUDA terminology, the GPU is called a device, whereas the CPU is called host. And a function executed by many threads in parallel on the device is called a kernel. As the C programming language extension, CUDA adds function type qualifiers __global__, __device__ and __host__ to specify execution on host or device and variable type qualifiers __device__, __constant__ and __shared__ to specify memory location on the device. The function __global__ with global qualifier is executed on the device, but this is callable from the host only. The function __device__ with device qualifier is executed on the device, and is callable from the device only. The function __host__ with host qualifier is executed on the host and callable from the host only. For more details, refer to [7]. The variable __device__ resides on the global memory in device and has the lifetime until program is over, so this is accessible form all the threads and host using API function. The variable __constant__ resides on the constant memory in device and has the lifetime until program is over; this is also accessible from all threads and host using API function. The last variable __shared__ resides on the shared memory in a thread block, so it has the lifetime of a block. It is also accessible from all the threads within the block. And there are four build-in variables gridDim, blockDim, blockIdx and threadIdx. The built-in variables gridDim and blockDim are type dim3 and refer to dimensions of the grid, block respectively. Other built-in variables blockIdx and threadIdx are type unit3 and refer to the block indices in the grid and the thread indices in the block respectively.

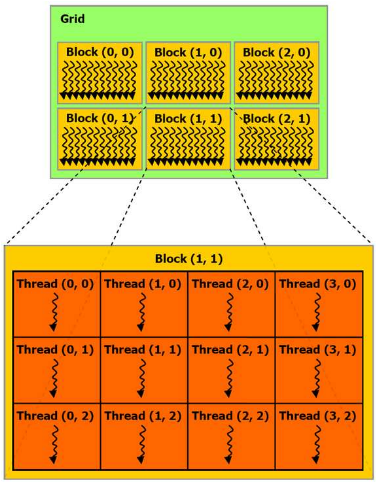

FIGURE 2. A Grid of Blocks, Blocks of threads [8]

As shown in Figure 2, the grid consists of blocks, and the block consists of threads. When we set up the kernel execution configuration parameters, it is important to select the dimension of the grid, i.e. the number of blocks in each grid, and the dimension of blocks, i.e. the number of threads in each block.

In other words, to maximize the utilization, the assignment of the number of threads per block and the number of blocks per grid should be selected carefully. Once the number of blocks and threads in kernel function is defined by programmer, the number of total threads is finally decided. Multi-thread model is highly suitable for multi-threaded Monte Carlo simulation for financial derivatives.

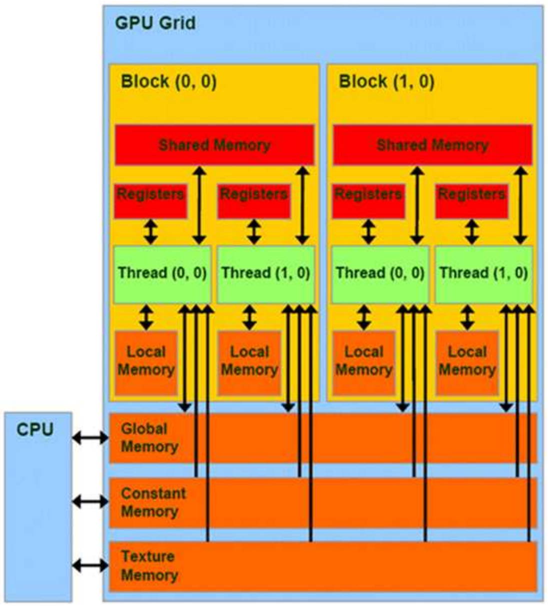

FIGURE 3. CUDA Memory Structure [10]

Unlike CPU with unique memory space, CUDA device has a variety of memories (See Figure 3) for getting high execution speed. First of all, the most representative memory is global memory as an off-chip GDDR RAM. The global memory can be read and written by all assigned threads and the host. As the second memory, shared memories as an on-chip GDDR are allocated to each block. Through the shared memory, GPU can communicate with all threads in a block. The shared memory which is used for optimization is much faster than the global memory since it is on-chip. As the third memory, the fastest registers are allocated to individual threads; each thread can communicate with only its own registers. In order not to exceed the limited memory sizes depending on compute capability, it is important for us to be aware of the device specifications we have. Once the kernel function runs on the device, device memory should be allocated in advance and data must be copied from host to device memory.

Between the host and the device, CUDA API functions(such as `cudaMemCpy`) are utilized for the memory block transfer. Figure 4 shows typical procedure of the data processing in CUDA, and it is summarized as follows:

Step 1: Allocate memory on the host and the device separately.
And copy data from the host to the device using cudaMemCpy function.
Step 2: Host instructs the process to the device.
Step 3: Kernel function executes parallel on each core.
Step 4: Copy the final results back from the device to the host using cudaMemCpy function.

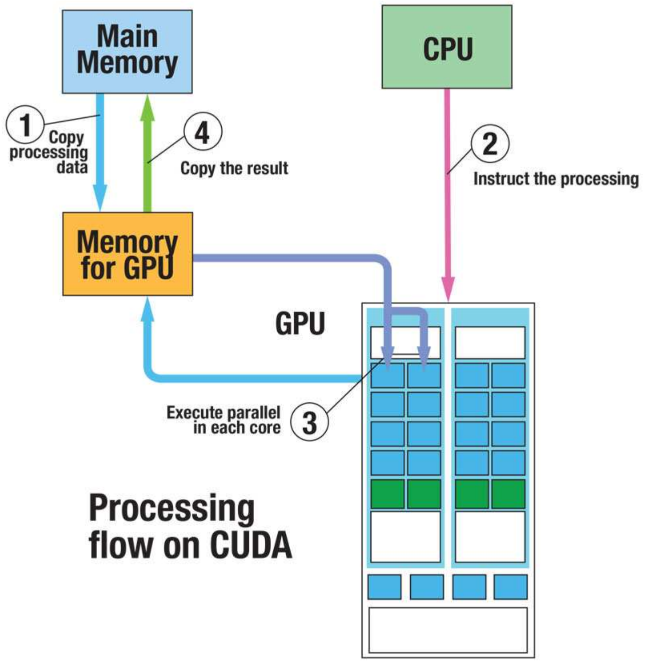

FIGURE 4. CUDA Data Processing Flow [11]

3.2. **Random Number Generation Using `cuRAND` Library.** Here, we refer to [9] for explanation of `cuRAND` Library. As mentioned before, the crucial key for simulating of Monte-Carlo method is to generate quickly random numbers. The standard random number generating functions in stdlib.h library was usually used to generate trivial pseudo-random numbers. However, in the CUDA world, because kernel function cannot call general embedded host function, we do not have to use it anymore. So we need to utilize `cuRAND` library. Here, `cuRAND` means NVIDIA CUDA random number generation (RNG) library for high-quality pseudo-random or quasi-random (low-discrepancy) numbers. `cuRAND` also allow us to generate random numbers in bulk from host code running on CPU or from CUDA functions/kernels running on GPU, and `cuRAND` provides a variety of RNG algorithms (MRG32k3a, Merseinne Twister, XORWOW for pseudo-RNG and Sobol for quasi-RNG) and distributions (uniform, normal, log-normal, Poisson, single-precision and double-precision) for our needs. In order to initialize the state of the RNG, using the device Application Programming Interface (API) that allows developers to create applications that interact with device hardware, within a kernel, we can call `cuRAND_init` function that sets up an initial state allocated by given seed, sequence number and offset. Each GPU thread must have an individual state in global memory. Then wrapper functions such as `cuRAND_normal` function can be used as needed. The implementation `cuRAND_normal` function mentioned above uses a Box_Muller transformation to generate normally distributed values. In general, when initializing the random generator state, we need more registers and local memory than RNG. It is important to call `cuRAND_init` function and `cuRAND_normal` function separately in different kernels to optimize performance. The following is the thread control algorithm we use on Single Program Multiple Data (SPMD). By doing like this, all threads execute the same program. The goal is to reduce the computa-

```
__global__ void Derivatives (parameters) {
        Thread Id = blockDim.x * blockIdx.x + threadIdx.x
        Thread N = blockDim.x * gridDim.x
        while(Thread Id<NSim) {
                for(j=0; j<T/Δt; j++) {
                        Derivative Valuation
                }
                Thread Id+=Thread N
        }
}
```

FIGURE 5. Approximate Code for Thread Control [1]

tion time by using parallel computing. The Monte-Carlo method is good candidate in parallel computing using MPI library or CUDA, since all paths are independent to each other, and we need to run massive number of path in order to get convergence. As the comparison target, parallel computing using MPI library will be used. MPI is a communication system which has been designed by a group of researchers to supply programmers with a standard for distributed-memory parallel programming. Since MPI provides a tool to enable communication between different CPUs, it does not depend on shared-memory architectures. The status of MPI as a distributed-memory system implies that multiple processes are started from the beginning and run, usually on different CPUs to completion. MPI programs have general structures:

Step 1: Include MPI header file (mpi.h).
Step 2: Initialize the MPI environment and communication.
Step 3: MPI calls and functions for parallel computation.
Step 4: Finalize the MPI communication.

In order to achieve good performance using MPI, we first divide simulation paths for valuation into different CPUs, then we combine those results as the final result of Monte Carlo simulation. Let us suppose that we have computer with four CPU cores to simulate Monte Carlo with 1M paths for any derivatives. In this situation, if we use MPI library, then each core works just only for 0.25M paths simultaneously. By dividing works, we can save costs and theoretically it will be exactly four times faster than single-CPU computation.

4. Numerical Results

In this section, we present parameters briefly, then compare the single CPU computation speed with parallel results. We compute two of financial products as samples. First one is Two-in-One ELS (hereinafter referred to 2in1 ELS), and the other one is CD-Equity Hybrid DLS (hereinafter referred to Hybrid DLS) as our main products. Since Hybrid DLS includes standard 2-dimensional ELS, it has much more simulation volume than 2in1 ELS. The reason why we include 2in1 ELS is to show that the more computational quantities we have to deal with, the better performance we will get. Table 1 is our experiment environments.

TABLE 1. Experiment Environments

OS	Linux Ubuntu 13.04 LTS
CPU	Intel Core(4) i7-2600 (3.40GHz)
Host Memory	DDR3 1600MHz (4GB)
GPU	NVIDIA GeForce GTX 770 Core(1536)
Device Memory	GDDR5 (2GB)
CUDA Version	CUDA 6.0
Compute Capability	3.0

4.1. **Two-in-One (2in1) ELS.** First of all, we see 2in1 ELS results as a preceding research. Unlike other standard ELSs, 2in1 ELS consider not the minimum rate of return but the average rate of return. By doing like this, it offers not only much higher automatic early-redemption chances, but much lower a risk of losing investment principal for investors.

TABLE 2. Parameters of 2in1 ELS

The number of Path	1,000,000	1^{st} Strike	90%
Initial Underlying Assets	(1.0, 1.0)	2^{nd} Strike	95%
Volatilities of Assets	(0.3, 0.3)	Knock-in	55%
Correlation	0.3	Discretized time	0.00125
Return per semi-annual	3.1%	Risk-free Rate	1%

The payoff diagram of the 2in1 ELS is shown in Figure 6. We executed the single-core 2in1 ELS code to measure the elapsed time using the ICPC as Intel C++ compiler on Linux. As shown in Table 3, the elapsed time is 54.75 seconds. We would like to see the reduced elapsed time in comparison with this result.

TABLE 3. Elapsed Time for Serial Codes (2in1 ELS)

2in1 ELS Price		Compiler / Elapsed Time(sec)
Sample mean	95% Confidence Interval	ICPC
0.981468	[0.980880, 0.982056]	54.75

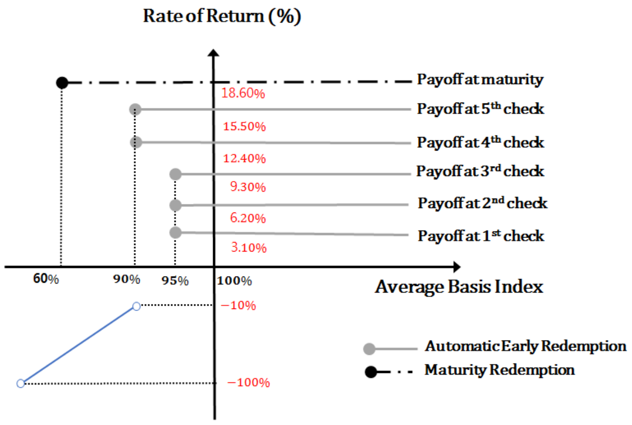

FIGURE 6. 2in1 ELS Payoff Structure [12]

We also execute the 2in1 ELS using MPI library, and use the MPICC compiler which links MPI programs written in C. If we insert into source code, data communication performs between processes. Each processor executes the same code simultaneously. It is obvious to see improved performance by adding cores to MPI program. When we use two and four cores, the elapsed time approximately is reduced by half and a quarter, respectively. The elapsed time speedup using MPI library with four cores shows approximately quadruple faster than serial code. And the elapsed time ratio ICPC/MPI was about 3.70.

TABLE 4. Elapsed time using MPI library (2in1 ELS)

2in1 ELS Price		Compiler / Elapsed Time(sec)	
Sample mean	95% Confidence Interval	MPICC (2-core)	MPICC (4-core)
0.981445	[0.980857, 0.982033]	28.07	14.81

We finally executed the product using the NVCC which is NVidia CUDA compiler. The NVCC looks for source code with the *.cu extension. So, all the CUDA code have the form of *.cu extension. Also this compiler plays a role in translating C codes to something that will run on the GPU. As a result when use NVCC, while MPI results show quadruple speedup than serial, CUDA shows approximately 26.32 times faster than serial, and 7.12 faster than MPI with quad cores. As we expected, the CUDA results show absolutely good performances.

In Table 5, $\langle\langle\langle 32, 512\rangle\rangle\rangle$ means that execution configuration parameters which are used during a call to a kernel function. That numbers are defined as the dimensions of the grid and the dimensions of each block, respectively.

TABLE 5. Elapsed time using CUDA (2in1 ELS)

2in1 ELS Price		Compiler / Elapsed Time(sec)
Sample mean	95% Confidence Interval	NVCC $\langle\langle\langle 32, 512\rangle\rangle\rangle$
0.981545	[0.980957, 0.982133]	2.08

In Figure 7, we depict the ratio of the execution time between MPI(4-cores) and CUDA, as well as the ratio of the execution time between single CPU and CUDA. According to the bar graph, the CUDA speed-up shows that if we use parallel computing using the CUDA, then CUDA result is 26.32 times faster than serial code using ICPC and is 7.12 times faster than MPI code using MPICC with four cores.

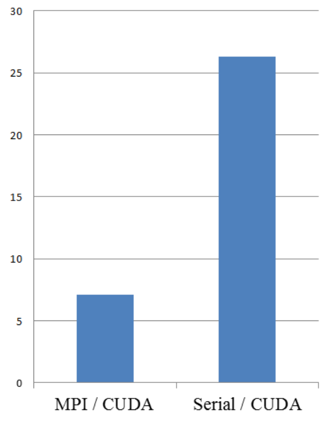

FIGURE 7. CUDA Speedup (2in1 ELS)

As Figures 8 and 9 show, when we execute the Monte Carlo simulation using CUDA, we set up the kernel execution configuration parameters. It is important to input the number of blocks per grid and threads per block in kernel. To find the optimal number of both of them is crucial point. But there have been some limits. In Figure 8, if we set the number of threads with 512, starting block number 32, the elapsed time does not reduce anymore.

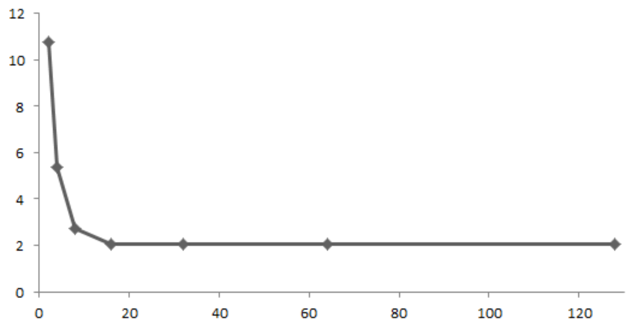

FIGURE 8. CUDA Elapsed time per blocks with fixed 512 threads (2in1 ELS)

Similarly, in Figure 9, if we set the number of blocks with 1024, starting threads number 32, like the preceding the elapsed time does not reduce anymore. As a result of the proper number of blocks and threads, we adopt that in the 2in1 ELS case, the number of blocks and threads is 32, 512, respectively, and in the hybrid DLS case, the number of blocks and threads is 32, 1024, respectively. It shows the fastest elapsed time in this problem.

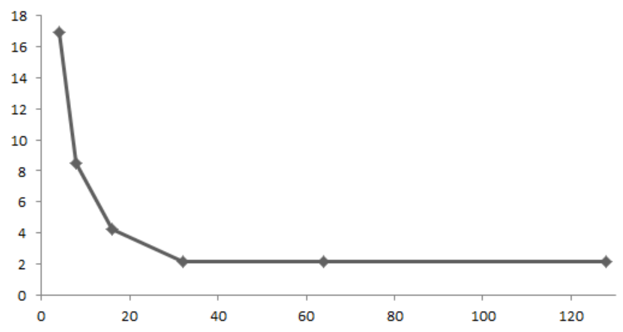

FIGURE 9. CUDA Elapsed time per threads with fixed 1024 blocks (2in1 ELS)

4.2. **CD-Equity Hybrid DLS.** Here, we refer to [3] for explanation of the product. This financial product is our main target to show the efficiency of the parallel computing performance. Even though this product is similar to any other 2-dimensional ELS, this additionally contains interest-rate model. So, it is more complicated and it needs much more computation works. For these reasons, CUDA in parallel computation has become indispensable. Table 6 includes the parameters we have used for Hybrid DLS. In order to compare with previous product, we set same the number of paths and discretized time interval. We assume that the mean-reverting speed, i.e. Hull and White alpha and short-rate volatility are constant value as shown the following table. And we have used the actual KRW IRS curve up to 3 years. In this curve, we assume that IRS is flat before 1year and IRS from 1 year to 2 year, 2 year to the maturity will be linear interpolated.

TABLE 6. Parameters of Hybrid DLS

The number of Paths	1,000,000	1^{st} Strike	95%
Initial Underlying Assets	(1.0, 1.0)	2^{nd} Strike	90%
Volatilities of Assets	(0.3, 0.3)	3^{rd} Strike	85%
Correlation	0.3	Knock-in	55%
Return per semi-annual	6.7%	Discretized time	1.0%
Hull-White volatility	1.0%	Hull-White alpha	0.1
IRS (to 1year)	2.175%	IRS (to 2year)	2.218%
IRS (to 3year)	2.280%		

As shown in Table 3, in the case of 2in1 ELS, serial code using ICPC elapsed time was approximately 55 seconds. However, Hybrid DLS result shows 186.22 seconds in Table 7. It

TABLE 7. Elapsed Time for Serial Codes (Hybrid DLS)

Hybrid DLS Price		Compiler / Elapsed Time(sec)
Sample mean	95% Confidence Interval	ICPC
0.920115	[0.919527, 0.920703]	186.22

takes a long time about 3.4 times than 2in1 ELS. This means that products computation works are much bulky than simple 2-dimensional ELS.

We also execute the Hybrid DLS using MPI library and use the MPICC compiler. As we mentioned in Table 4, when we use two and four cores, the elapsed time approximately is reduced by half and a quarter, respectively, in Table 8. The elapsed time ratio ICPC/MPI is about 3.81.

TABLE 8. Elapsed time using MPI library (Hybrid DLS)

Hybrid DLS Price		Compiler / Elapsed Time(sec)	
Sample mean	95% Confidence Interval	MPICC (2-core)	MPICC (4-core)
0.919963	[0.919375, 0.920551]	92.54	48.81

Table 9 is the final results for hybrid DLS pricing using CUDA.

TABLE 9. Elapsed time using CUDA (Hybrid DLS)

2in1 ELS Price		Compiler / Elapsed Time(sec)
Sample mean	95% Confidence Interval	NVCC $\langle\langle\langle 32, 1024\rangle\rangle\rangle$
0.920541	[0.919953, 0.921129]	5.05

Finally, we depict the ratio of the run time between MPI(4-cores) and CUDA, as well as the ratio of the run time between single CPU and CUDA in Figure 10. It shows that, the CUDA result shows 36.88 times faster than the result of using single CPU and it is 9.67 times faster than the result of using MPI(4-cores with NVCC compile). As in the previous example(2in1 ELS), the CUDA result show absolutely good performance.

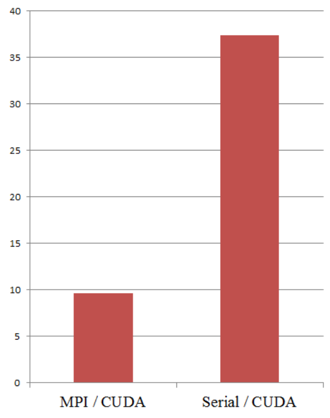

FIGURE 10. CUDA Speedup (Hybrid DLS)

The Hybrid DLS we have used is a combination of a standard 2-dimensional step-down ELS and CD range accrual DLS. So, this product has more complicated and massive problem. When we use the parallel computing with CUDA, unlike relatively less massive 2in1 ELS, hybrid DLS results show better performance.

5. Conclusion

In financial world, the Monte-Carlo method is one of the popular and dominant computational algorithms to compute value of financial derivatives with complicated structures, to predict risk factors. However, it is computationally too costly in terms of time. So, many researchers have pointed out this drawback. To overcome it, parallel computing as the best alternative way, has to be employed. In this work, we tested two of financial products using parallel computing such as MPI, CUDA.

As previous studies [1, 2, 7, 13], even though their fields of study are all different, they commonly concluded that Nvidia GPUs are well-suited for running large-scale Monte Carlo simulations. In the brochure [14], they say GPUs make the difference in finance: firstly, faster pricing enable us to get more revenue, secondly, more modeling is less risky, and lastly maximizing resources is to increase efficiency. Because of these advantages, NVidia GPU computing is used by many financial and computing software companies such as JP Morgan, MUREX, SUNGARD, Hanweck, Global Valuation, MathWorks, and so on. In order to agree if they are right or not, we have tested some of financial products. Then we finally concluded that the NVidias CUDA as GPGPU technology shows greater performance compare with others. In comparison with simple ELS, hybrid DLS which is more complicated product showed more good performance. The achieved speedup with respect to GPU computation is around 37 with single GPU. This is good news that when we have to compute more massive problems. Monte-Carlo simulation by CUDA can be used to accelerate the heavy computation and must be very suitable for implementing independent paths simulation in parallel. In order to meet the stringent requirements of the financial market participants, we are clearly sure that use CUDA must be considered in terms of low setup costs and computation speed. In terms of another positive side, we can increase number of Monte-Carlo paths to enhance accuracy.

Lastly, since we have yet to explore utilizing shared memories and others for memory optimization, we hope that future work will utilize a variety of CUDA memories. In fact, we should have utilized the shared memory which is hundreds of times faster than global memory. And threads can cooperate via the shared memory. If we take advantage of this memory, our performance will be maximized. Work in progress is to minimize the memory access time between device and host so as to can be made faster. After completing threads and memories optimization, personally in the near future, we would like to parallelize on multiple GPUs with MPI for pricing other financial derivatives by using the Operator Splitting Method.

Acknowledgements

Bae was supported by Ajou Research Fund, and also Bae (2015R1D1A1A01057976) and Kim (2016R1D1A1A09917026) by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology.

References

[1] D. Fei, *A study on the efficiency of CUDA in calculation of SOPM*, Master Thesis, Kyungpook National University (2009), `http://www.riss.kr/link?id=T11790495`

[2] J.L. Fernandez, *Static and dynamic SABR stochastic volatility models: Calibration and option pricing using GPUs*, Mathematics and Compuers in Simulation **94**, (2013), 55–75.

[3] KDB Daewoo Securities, *CD-EQUITY DUET 387th DLS Instruction* (2011).

[4] D. Brigo and F. Mercurio, Interest rate models theory and practice with smile, inflation and credit, 2nd edition, London: Springer (2006).

[5] J.C. Hull and A. White, *Pricing interest rate derivative securities*, Review of Financial Studies (1990).

[6] J.C. Hull and A. White, *Numerical procedures for implementing term structure models I: single factor models*, Journal of Derivatives (1994).

[7] T. Liu, X.G. Xu, and C.D. Carothers, *Comparison of two accelerators for Monte Carlo radiation transport calculations, Nvidia Tesla M2090 GPU and Intel Xeon Phi 5110p coprocessor, A case study for X ray CT imaging dose cacluation*, Joint International Conference on Supercomputing in Nuclear Applications, Monte Carlo **82**, (2015), 230–239.

[8] NVIDIA, *CUDA C Programming Guide,* `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (2016).

[9] NVIDIA, *CUDA CURAND Library,* `http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf`, (2016).

[10] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture,* `http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`, (2007).

[11] T. Williams, *Parallel Processing Platform Opens Bridge to High Performance Embedded Systems,* `http://rtcmagazine.com/articles/view/103718`, (2014).

[12] Korea Investment & Securities, *IMYOU 2IN1 ELS Instruction* (2013).

[13] J. A. Anderson, E. Jankowski, Th.L. Grubb, M. Engel, and S.C. Glotzer, *Massively parallel Monte Carlo for many particle simulations on GPUs,* Journal of Computational Physics **254**, (2013), 27–38.

[14] NVIDIA, *Introducing NVIDIA TESLA GPUs for Computational Finance,* `http://www.nvidia.com/content/tesla/pdf/finance_brochure_2014_fin2.pdf`, (2014).