City University of New York (CUNY)
CUNY Academic Works

Dissertations and Theses
City College of New York

2010

Acceleration of Monte Carlo Value at Risk Estimation Using Graphics Processing Unit (GPU)

Wei Wu
CUNY City College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/cc_etds_theses/10
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Acceleration of Monte Carlo Value at Risk Estimation Using Graphics Processing Unit (GPU)

THESIS

Submitted in partial fulfillment of the requirement for the degree

Master of Engineering (Computer Science)

At

The City College

of the

City University of New York

by
Wei Wu
December 2010

Approved:

____
Professor Izidor Gertner, Thesis Advisor

____
Professor Douglas Troeger, Chairman

Department of Computer Science

Abstract

Value at Risk (VaR) is one of the most popular tools used to estimate the exposure to market risks, and it measures the worst expected loss at a given confidence level. Monte Carlo simulation is one of the best methods to calculate VaR and it is widely used in financial industry. Unfortunately, it is time consuming especially when the simulated samples and the number of assets in a portfolio are very large. The graphics processing unit (GPU) is a specialized multiprocessor which has highly parallel structure supporting more effective than general-purpose CPUs for a range of complex algorithms. In this paper, we will investigate the acceleration of Monte Carlo simulation by using GPU. Firstly, we will introduce the VaR conception and three basic method to estimate VaR. Then we will describe GPU computation and performance using matrix multiplication. At last, we will focus on the parallel algorithm of estimation VaR using Monte Carlo method, and implementation of VaR calculation using CUDA on GPU. Extensive experiments will be performed to show that GPU can achieve a much faster speed than Matlab, which demonstrates clear the advantage to use GPU in VaR estimation.

Keywords: Value at Risk, Monte Carlo Method, CUDA, GPU

Contents

1. **Introduction** 1
2. **Value at Risk Methodologies** 4
   - 2.1 Historical Method 4
   - 2.2 Variance-Covariance method 6
   - 2.3 Monte Carlo Simulation 8
3. **Graphics Processing Unit (GPU) Computing** 10
   - 3.1 GPU and CUDA 10
   - 3.2 Matlab 13
   - 3.3 Performance Comparison of Matrix Multiplication using C and Matlab in CPU, and CUDA in GPU 13
4. **Monte Carlo Simulation to Estimate Value at Risk** 15
   - 4.1 Calculate Profit-Loss-Rate 16
   - 4.2 Multivariate Normal Distribution 16
   - 4.3 Calculate Portfolio Value 23
   - 4.4 Merge sort 24
5. **Experiments** 26
6. **Conclusion** 30
7. **Reference** 31
8. **Appendix** 33

List of Figures

Figure 1: Historical method 4
Figure 2: Historical method disadvantage analysis 6
Figure 3: Variance-covariance method 7
Figure 4: Variance-covariance method disadvantage analysis 8
Figure 5: Monte Carlo method 9
Figure 6: CUDA Program Model 11
Figure 7: Kernel Structure 12
Figure 8: Memory Hierarchy 12
Figure 9: Performance of Matrix Multiplication 14
Figure 10: Profit-Loss-Rate parallel algorithm 16
Figure 11: Mean parallel algorithm 18
Figure 12: Covariance parallel algorithm 18
Figure 13: Uniform Distribution parallel algorithm 21
Figure 14: Box–Muller parallel algorithm 22
Figure 15: Matrixes Multiplication parallel algorithm 23
Figure 16: Calculate portfolio parallel algorithm 24
Figure 17: Merge Sort algorithm example 24
Figure 18: Parallel algorithm with $n = 8$ 25
Figure 19: Frequency Distribution of Monte Carlo Simulation and Historical Method 28
Figure 20: Running Time of Monte Carlo Simulation 29

List of Tables

Table 1: Algorithm for VaR estimation using Monte Carlo 15

Table 2: Some Suggested Linear and Multiplicative Random Number Generators 21

Table 3 :NVIDIA Quadro FX 3700 Performance 26

1. Introduction

In the financial world nowadays, Value-at-Risk has become one of the most important and the most used measures of risk. Investors like to focus on the promise of high returns, but they should also ask how much risk they must assume in exchange for these returns. Risk is about the odds of losing money, and VaR is based on that common-sense fact. By assuming investors care about the odds of a really big loss, VaR answers the question, "What is the most I can - with a 95% or 99% level of confidence - expect to lose in dollars over the next month?", or "What is the maximum percentage I can - with 95% or 99% confidence - expect to lose over the next day? So we can see that the "VAR question" has three elements: a relatively high level of confidence (typically either 95% or 99%), a time period (a day, a month or a year) and an estimate of investment loss (expressed either in dollar or percentage terms). Jorion (1997) defines Value at Risk as: "the expected maximum loss (or worst loss) over a target horizon within a given confidence interval."[1]

The first using VaR ideas can date to the late 1970s and early 1980s, the Chicago Mercantile Exchange used "Standard Portfolio Analysis" (SPAN) system and the Chicago Board Options Exchange (CBOE) used "Theoretical Intermarket Margining System" (TIMS) to do margin calculations. [2] JP Morgan's RiskMetrics system in 1995 increased the profile of Value at Risk substantially, and as the importance of Value at Risk has increased, so has the volume of academic literature developing, supporting or criticizing this risk measure. [3]

Theoretical research that relied on the Value-at-Risk as a risk measurement was initiated by Jorion (1997)[1], Dowd (1998)[4], and Saunders (1999)[5], who applied the Value-at-Risk approach based on risk management emerging as the industry standard by choice or by regulation.

The existing VaR related academic literature focuses mainly on measuring VaR from different estimation methods to various calculation models. Cabedo and Moya (2003)[20], Estimating oil price Value at Risk using the historical simulation, and develop the variance-covariance method based on ARCH models forecasts. Duffie and Pan (1997)[6], Cardenas (1999) [7], Rouvinez (1997) [8], Jamshidian and Zhu (1997) [9] do research to improve Monte Carlo method used to estimate VaR. Embrechts, Kluppelberg, and Mikosch (2003)[11], Lucas and Klaassen (1998)[12] focus on the tail behavior of the returns. Bollerslev, Engle, and Nelson (1994)[13] discuss the GARCH-type models. Andrey Rogachev(2002) [14] introduce dynamic

Value-at-Risk. Dean Fantazzini(2009) [15]use dynamic Copula theory to model VaR, copula functions allow to construct flexible multivariate distribution with different margins and different dependence structure, without the constraints of the traditional joint normal distribution.

All these researches mentioned above are based on improvement the algorithm or models. In reality, however, computational constraints are one of important factors in explaining the simplifications which have been into systems such as SPAN or TIMS. Every time a trade takes place, the positions of two economic agents are updated, and two VaR computations are required. The most active futures exchanges in the world today experience roughly 1,000,000 trades in around 20,000 seconds. This requires 100 VaR computations per second, on average. Given the unevenness of trading intensity in the day, this easily maps to a peak requirement of 500 VaR computations per second, or a VaR computation in two milliseconds. [2] So how to improve the performance of VaR estimation becomes important practical issue in current financial industry.

With the development of new hardware and improvement of processor speed, parallel computing has been broadly used in the finance area. One of the representations is the Graphic Processor Unit (GPU). GPUs are originally designed to very efficiently at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. The term of GPU was defined proposed and popularized by NVIDIA in 1999, who marketed the GeForce 256 as "the world's first 'GPU', or Graphics Processing Unit, a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."

Thanks to GPU's highly parallel structure that makes them more effective than general-purpose CPUs for a range of complex algorithms. Nowadays, GPU is widely used in financial computing, such as VaR estimating, option pricing, etc. Lots of general methods used in finance can be greatly accelerate by GPU, such as Finite Differences, Random number generation, Monte Carlo test case, dynamic programming, etc. Michael Feldman, an HPCwire editor, said that one of the new kids on Wall Street is GPU computing, a technology that is making inroads across nearly every type of HPC application. [17]Greg N. Gregoriou described GPU computing of VaR in his book that GPU approach is ten or even hundreds of times cheaper than other tow supercomputing approaches (mainframs and grid computing).[18] And, Matthew Dixon(2009) [19]compares NVIDIA GeForce GTX280 graphics processing unit (GPU) and a quadcore Intel

Core2 Q9300 central processing unit (CPU) to simulate VaR based delta-gamma method. GPU is hundreds times faster than the CPU. All of these researches show GPU have great potential to do complex computation in financial industry with a much faster speed than general CPU and a much lower cost than Supercomputers.

In this paper, we will investigate how to use GPU to calculate VaR based on Monte Carlo method.

The remaning part of this paper is organized as follows: section 2 will describe and compare three basic methods to calculate VaR: historical, Monte-Carlo and variance-covariance methods and point out the advantage and disadvantage of using these methods. Section 3 will introduce GPU and CUDA computing, and then compare the performance using C and Matlab in CPU, and CUDA in GPU to do matrix multiplication. Section 4 will describe the parallel algorithm to calculate VaR using Monte Carlo simulation. Section 5 will show the experiments and performance results. At last, secection 6 will conclude the whole paper.

2. Value at Risk Methodologies

All the methods used to estimate VaR can be separately in three categories. We simply explain these three methods as following:

2.1 Historical Method

Historical simulations represent the simplest way of estimating the Value at Risk for many portfolios. In this approach, the VaR for a portfolio is estimated by creating a hypothetical time series of returns on that portfolio, obtained by running the portfolio through actual historical data, putting returns from worst to best, and computing the changes that would have occurred in each period. Historical method assumes that history will repeat itself, from a risk perspective.

Cabedo and Moya provide a simple example of the application of historical simulation to measure the Value at Risk in oil prices. [20] Using historical data from 1992 to 1998, they obtained the daily prices in Brent Crude Oil and then calculate the VaR. Another example from this website: http://www.investopedia.com/articles/04/092904.asp, explains the historical method very clearly, see Figure 1.

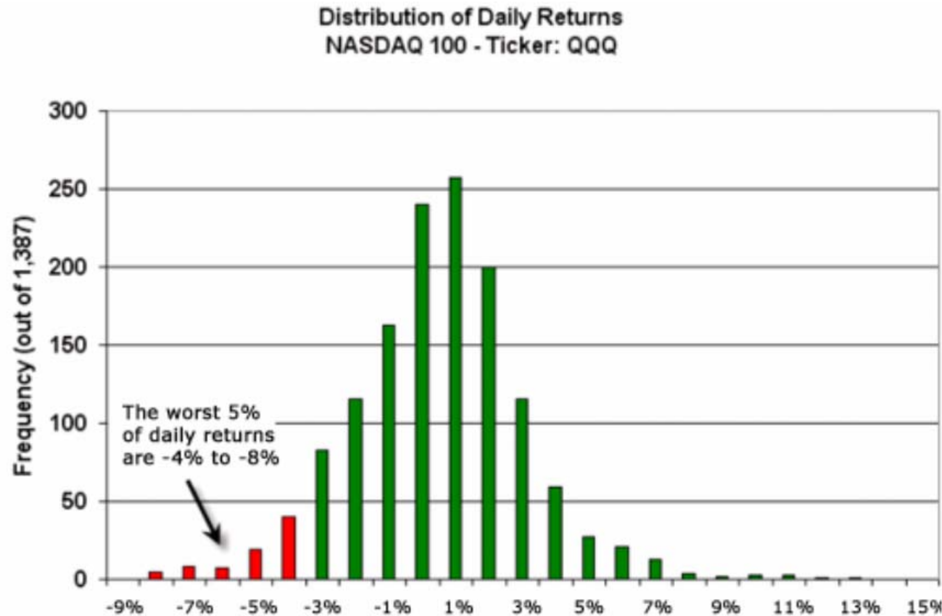


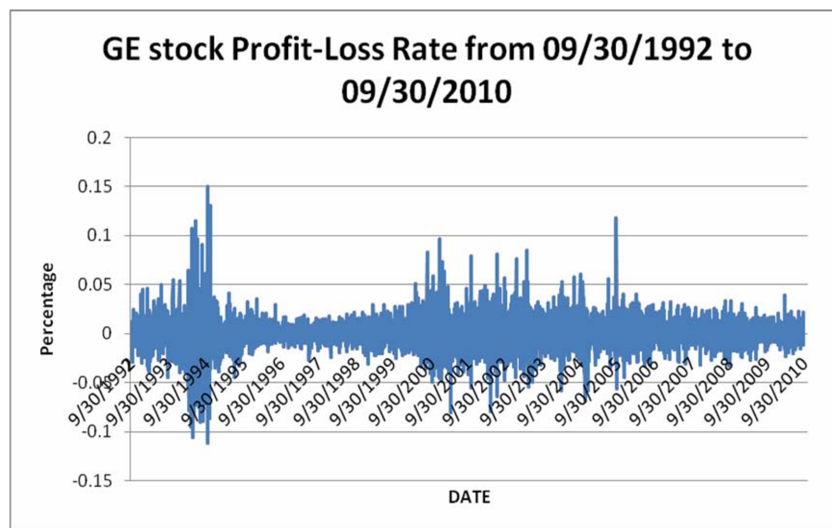
Figure 1: Historical method

The QQQ started trading in Mar 1999. Historical method will calculate each daily return about 1400 points, and put them in a histogram that compares the frequency of return "buckets". The returns are ordered from left to right, then we can get that with 95% confidence the worst daily loss will not exceed 4%. If we invest $100, we are 95% confident that our worst daily loss will not exceed $4.

Historical method is the simplest and fastest method to calculate VaR, but the underlying assumptions, that the near future will be like the recent past and that we can reasonably used the data from the past to estimate risks over the near future, give rise to its weaknesses.

While all three approaches estimating VaR use historical data, historical simulations are much more reliant on historical data than the other two as the Value at Risk is computed entirely from historical price changes. There is little room to overlay distributional assumptions (as we do with the Variance-covariance approach) or to bring in subjective information (as we can with Monte Carlo simulations). In Figure 2 (a), it shows an example of GE stock price change in the period from 09/30/1992 to 09/30/2010. From 1992 to 2003, stock price increased gently, but in the period 2003-2005, 2005-2009 and 2009-2010, stock price changed periodically and increased dramatically. And in Figure 2 (b), the Profit-Loss-Rate in the period 1993-1995 and 2000-2006 changed intensively than period 1996-1998 and 2006-2010. We compute VaR, using historical data, where all data points are weighted equally. In other words, the price changes from trading days in 1994 or 2001 affect the VaR in exactly the same proportion as price changes from trading days in 1997 or 2009. But the trend of changing in volatility is different in different historical time period, so, based upon 1993-1996 and 2000-2006 data, we would have been exposed to much larger losses than expected over the 1996-1998 and 2006-2010 period. We will under estimate or over estimate the VaR.

(a) GE stock price change from 09/30/1992 to 09/30/2010



(b) GE profit-loss rate change from 09/30/1992 to 09/30/2010

Figure 2: Historical method disadvantage analysis

2.2 Variance-Covariance method

Since Value at Risk measures the probability that the value of an asset or portfolio will drop below a specified value in a particular time period, it should be relatively simple to compute if we can derive a probability distribution of potential values. So the idea behind the variance-covariance is similar to the ideas behind the historical method - except that we use the familiar curve instead of actual data. The advantage of the normal curve is that we automatically know

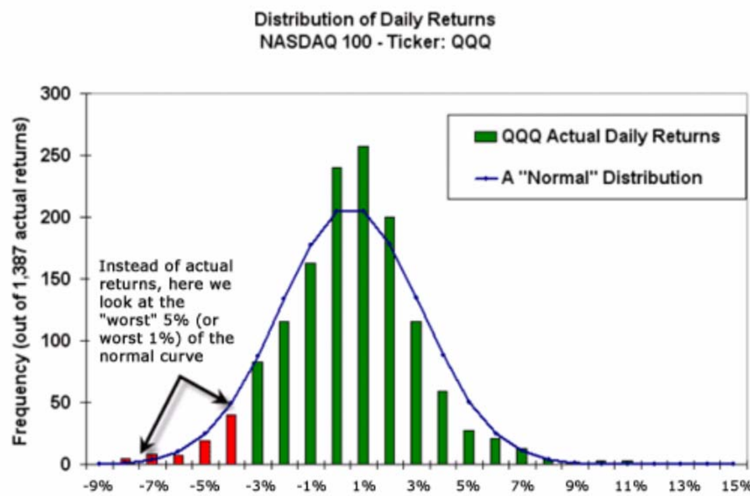
where the worst 5% and 1% lie on the curve. They are a function of our desired confidence and the standard deviation (σ), see

Figure 3 (b).

Use the example from this website: http://www.investopedia.com/articles/04/092904.asp, to explain variance-covariance method. The curve above is based on the actual daily standard deviation of the QQQ, which is 2.64%. So we can very easily get VaR, which is 4.36% when confidence level is 95%, and which is 6.16% when confidence level is 99% (

Figure 3 (a) and

Figure 3 (c)).



(a)

Confidence	# of Standard Deviations (σ)
95% (high)	- 1.65 × σ
99% (really high)	- 2.33 × σ

(b)

Confidence	# of σ	Calculation	Equals:
95% (high)	- 1.65 × σ	- 1.65 × (2.64%) =	- 4.36%
99% (really high)	- 2.33 × σ	- 2.33× (2.64%) =	- 6.16%

(c)

Figure 3: Variance-covariance method

That is basically what we do in the variance-covariance method, an approach that has the benefit of simplicity but is limited by the difficulties associated with deriving probability distributions. The most convenient assumption both from a computational standpoint and in terms of estimating probabilities is normality and it should come as no surprise that many VaR measures are based upon some variant of that assumption. If, for instance, we assume that each market risk factor has normally distributed returns, we ensure that that the returns on any

portfolio that is exposed to multiple market risk factors will also have a normal distribution. But if conditional returns are not normally distributed, the computed VaR will understate the true VaR. Moreover, as showed in Figure 4, the mean and covariance across assets change over time, that means standard deviations can be changed over time.

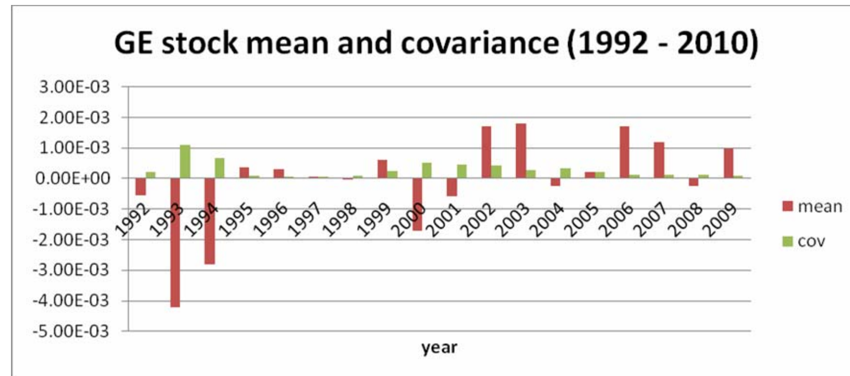


Figure 4: Variance-covariance method disadvantage analysis

2.3 Monte Carlo Simulation

A Monte Carlo simulation refers to any method that randomly generates trials, but by itself does not tell us anything about the underlying methodology. As Variance-covariance method, Monte Carlo method has first to calculate mean and covariance. Rather than calculate VaR using σ , it simulates route, specify probability distributions using random number.

Use the example from this website: http://www.investopedia.com/articles/04/092904.asp, the result shows in Figure 5. Run 100 hypothetical trials of monthly returns for the QQQ. Among them, two outcomes were between -15% and -20%; and three were between -20% and 25%. That means the worst five outcomes (that is, the worst 5%) were less than -15%. The Monte Carlo simulation therefore leads to the following VAR-type conclusion: with 95% confidence, we do not expect to lose more than 15% during any given month (Figure 5).

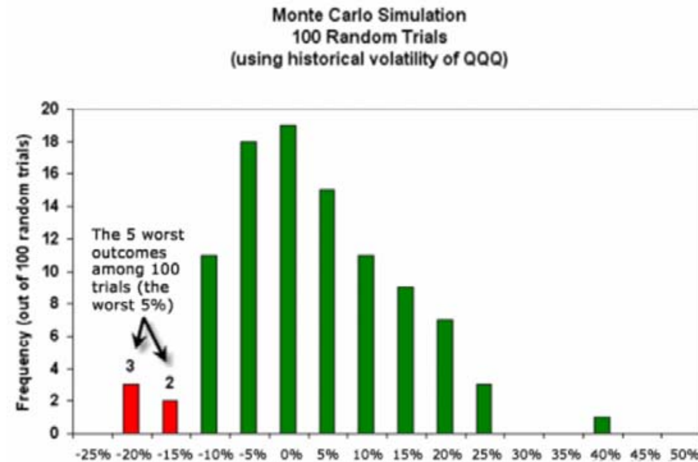


Figure 5: Monte Carlo method

The strengths of Monte Carlo simulations can be seen when compared to the other two approaches for computing Value at Risk. Monte Carlo is by far the most flexible, since it allows considering arbitrarily complex models and/or portfolio instruments. Unlike the variance-covariance approach, we do not have to make unrealistic assumptions about normality in returns. In contrast to the historical simulation approach, we begin with historical data but are free to bring in both subjective judgments and other information to improve forecasted probability distributions. All of these changes make Monte Carlo a better method to calculate VaR in reality. However, Monte Carlo method is extremely computationally intensive because it is based on the iteration of a particular, generally simple, procedure. [18]When the number of portfolio assets or the samples of simulation is large, Monte Carlo method is very slow. This limitation triggers us to investigate more fast way to do Monte Carlo calculation. Next, we will introduce GPU computing, which is a good way to conduct Monte Carlo calculation to estimate the VaR.

3. Graphics Processing Unit (GPU) Computing

3.1 GPU and CUDA

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth. The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The GPU has evolved over the years to have teraflops of floating point performance. NVIDIA revolutionized the GPU and accelerated computing world in 2006-2007 by introducing its new massively parallel architecture called “CUDA”.

CUDA is a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. The CUDA architecture consists of 100s of processor cores that operate together to crunch through the data set in the application. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages or application programming interfaces are supported, such as CUDA FORTRAN, OpenCL, and Direct Compute. [16]

CUDA programming model is showed in the following Figure 6. CUDA is a serial program with parallel kernels using C code. When a program is running, general C code executes in the host CPU, and parallel kernel C code executes in many device threads (GPU threads) across multiple processing elements. One kernel is executed at a time on the device, and it has many threads execute parallel.

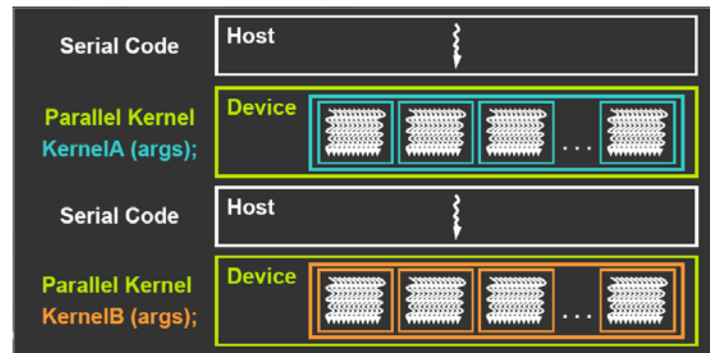


Figure 6: CUDA Program Model

The structure of a kernel is showed in Figure 7. A kernel also called a grid in the device that includes several blocks, and each block includes several threads. So, the number of total threads is equal to the number of threads per block times the number of blocks. Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write scalable code. So each thread executes the same code but processes different data based on its threadID.

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 8. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. The speed of a thread access these three different memories are increasing.

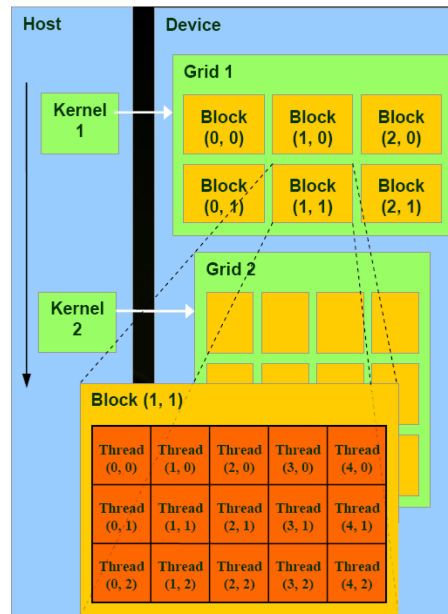


Figure 7: Kernel Structure

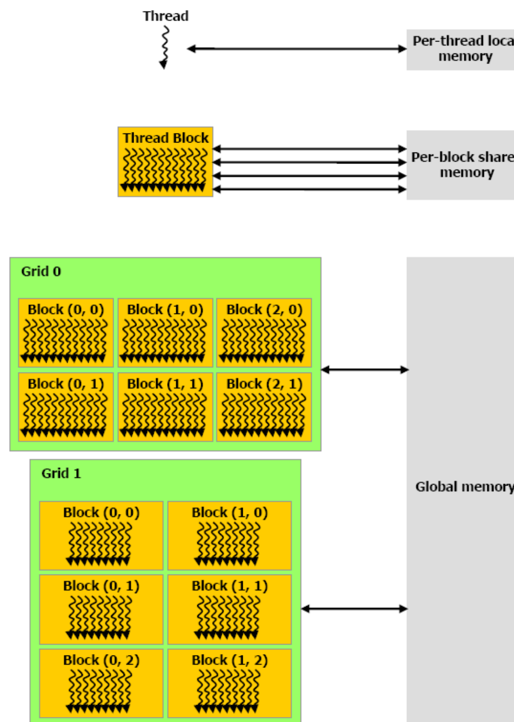


Figure 8: Memory Hierarchy

3.2 Matlab

MATLAB is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and FORTRAN. MATLAB allows for easy numerical calculation and visualization of the results without the need for advanced and time consuming programming. The disadvantage is that it can be slow, especially when bad programming practices are applied.

3.3 Performance Comparison of Matrix Multiplication using C and Matlab in CPU, and CUDA in GPU

In order to show the advantage of CUDA in GPU, we use matrix multiplication to test the performance of C programming and Matlab programming in CPU and C programming in CUDA in GPU. We set up a simple test scenario, two matrixes multiplication, with each of matrix is a $n \times n$ dimension matrix. The result shows in the following Figure 9. We can find it clearly shows that the trend of time consumption using C program is exponentially increased with the matrix size increasing. On the other hand, the speed of CUDA program is thousands of times faster than C program when matrix size is large. For example, C program in CPU takes 9 minutes when the matrix size is 2048×2048 , while CUDA in GPU takes 0.3 seconds. Matrix multiplication is the best example to show the advantage using of CUDA than using C program in CPU. The result also shows the property of GPU. The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processor in the system. When we use GPU sufficiently, which means the parallel threads used are almost maximum threads the device allowed, the performance of GPU is better. And reading and writing data with global memory is much more time consuming in GPU than reading and writing data in CPU. That's why CUDA programming is slower then C programming in CPU when matrix size is small. When matrix size is 16×16, the running time of C programming in CPU is almost 0, but the running time of CUDA GPU is 1.7 milliseconds.

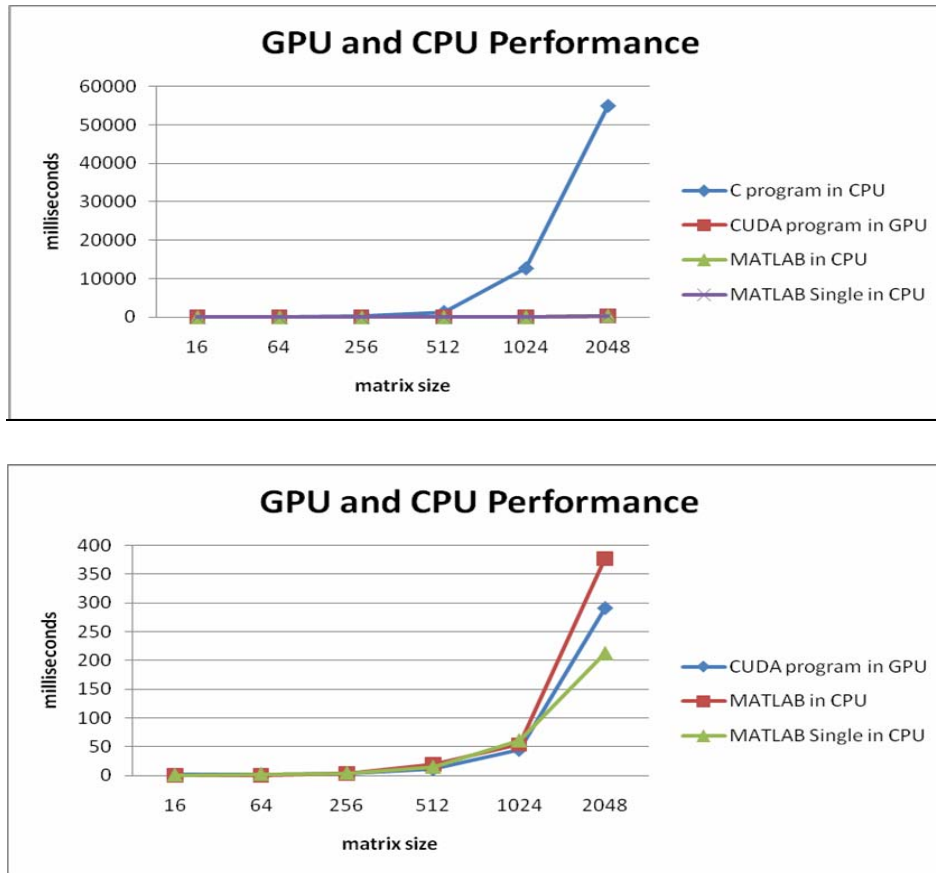


Figure 9: Performance of Matrix Multiplication

At the same time, we can also find that, Matlab is also very fast to do simple matrix computing. The Matlab programming using original format (double) is a little slower than CUDA. But as we known, CUDA just support float point computation. When we use single value to do matrix multiplication, Matlab is faster than CUDA (see Figure 9). So this example also shows that Matlab is very efficient in the simple numerical calculation.

4. Monte Carlo Simulation to Estimate Value at Risk

In this section, we will describe the detail of VaR estimation using Monte Carlo and how to implement it in CUDA. The algorithm of Monte Carlo Simulation to estimate VaR is showed in the following Table 1. The input is portfolio data, which includes historical data and the close price of the previous day V_t on each asset in the portfolio. Based on the historical data, we can get profit-loss rate, which is used for further simulation. The second step is to calculate profit-loss rate in $t+1$ time using the multivariate normal distribution, and where $t+1$ profit-loss rate has ms samples. Based on V_t and $t+1$ profit-loss rate, we will then calculate $t+1$ portfolio price V_{t+1}. Finally, we will sort V_{t+1} and output the VaR of the confidence level.

Table 1: Algorithm for VaR estimation using Monte Carlo

Algorithm of VaR estimation using Monte Carlo
Input: Portfolio $w \in \mathbb{R}^N$ Output: VaR of portfolio w. Procedure: 1. Let V_t be the value of the portfolio at the close of the previous day; let ${r^t}_{m,n}$ be the historical profit-loss rate. 2. Simulate $M_{samples}$ draws ${r^{t+1}}_{ms,n}$, from the multivariate normal distribution of returns on the underlying. 3. At each draw ${r^{t+1}}_{ms,n}$, apply theoretical valuation formulas to obtain V_{t+1}, the value of the portfolio w if prices changed by $r_{ms,t+1}$. 4. Sort the $M_{samples}$ values for V_{t+1}, and read of the percentile value, which is the desired VaR.

In the following part, we will present the detail implementation to use parallel computing to do Mente Carlo VaR estimation. We assume there are n stocks; every stock has m business day historical data; set the stock vector $X_1, X_{2,} \cdots, X_n$, each vector is m dimension.

4.1 Calculate Profit-Loss-Rate

According to the algorithm listed above, the first step is to get the historical data for n stocks, the opening and the close stock price of a business day. And then calculate the profit-loss rate ($plRate$).

$$plRate = (close - open) / open \quad (4.1)$$

The parallel calculation is implemented as following Figure 10, there are total $m \times n$ threads in parallel, the block i and the thread j calculates $plRate_{i,j} = (close_{i,j} - open_{i,j}) / open_{i,j} \quad i = 1, \cdots, n; j = 1, \cdots, m$.

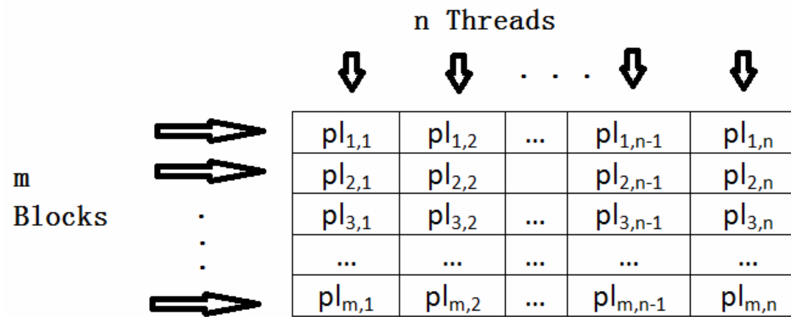


Figure 10: Profit-Loss-Rate parallel algorithm

4.2 Multivariate Normal Distribution

After obtaining the profit-loss rate for m business day, the next is to assume X_n stocks fit to multivariate normal distribution, and generate ms samples $r^{t+1}{}_{ms,n}$.

The multivariate normal distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions. A random vector is said to be multivariate normally distributed if every linear combination of its components has a univariate normal distribution.

If we have a p random vector X that is distributed according to a multivariate normal distribution with population mean vector μ and population variance-covariance matrix Σ , then this random vector, X , will have the joint density function as shown in the expression below:

$$\varphi(x) = (\frac{1}{2\pi\sigma^2})^{p/2} |\Sigma|^{-1/2} \exp\{-\frac{1}{2}(X-\mu)'\Sigma^{-1}(X-\mu)\} \tag{4.2}$$

And the distribution is $X \sim N(\mu, \Sigma)$.

A widely used method for drawing a vector X from the n-dimensional multivariate normal distribution with mean vector μ and covariance matrix Σ (required to be symmetric and positive-definite) works as follows:

(1) Find any matrix A such that $AA^T = \Sigma$. Often this is a Cholesky decomposition, but a square root of Σ would also suffice; here we use Singular Value Decomposition instead.

(2) Let $Z = (Z_1, \cdots Z_n)$ be a vector whose components are n independent standard normal variates (which can be generated by using the Box-Muller transform).

(3) Let $mvd = u + A * Z$. This has the desired distribution due to the affine transformation property.

According to this theory, generate a multivariate normal distribution matrix, which has ms rows, can be achieved in the following steps.

4.2.1 Mean (μ)

The mean is the arithmetic average of a set of values.

According to Eq. (4.3), the easiest way is to set n threads. But it wastes lots of resource, which not sufficiently use all the blocks and threads. So I set n blocks, and every block has $m/32+1$ threads, each threads calculate $pl_{i,j} + pl_{i+1,j} + \cdots + pl_{i+31,j}$. (Figure 11)

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} X_{ij} \qquad i = 1, 2, \cdots n \tag{4.3}$$

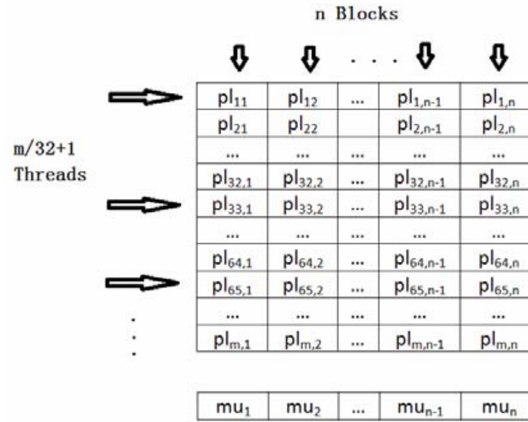


Figure 11: Mean parallel algorithm

4.2.2 Covariance(Σ)

Covariance is a measure of how much two variables change together as defined in Eq.(4.4).

$$\Sigma = \operatorname{cov}(X_i, X_j) = \frac{1}{n}\sum_{i=1}^{n}(X_i - u_i)(X_j - u_j) \qquad i, j = 1, 2, \cdots n \tag{4.4}$$

Using the same idea as mean to do parallel computing of covariance, there are $n \times n$ blocks, and every block has $m/32+1$ threads, each threads calculate $(pl_{t,i} - \mu_i)(pl_{t,j} - \mu_j) + (pl_{t+1,i} - \mu_i)(pl_{t+1,j} - \mu_j) + \cdots + (pl_{t+31,i} - \mu_i)(pl_{t+31,j} - \mu_j)$. (Figure 12)

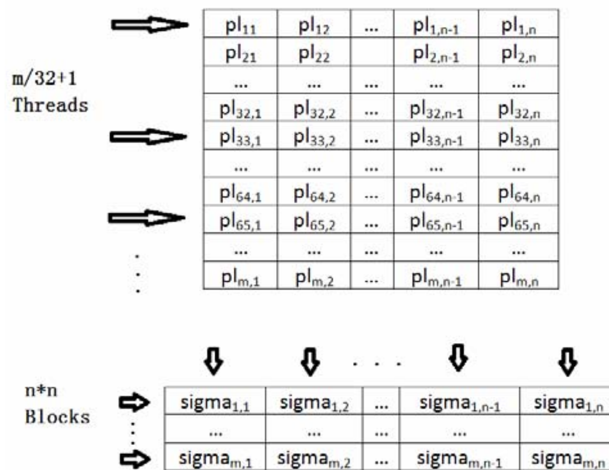


Figure 12: Covariance parallel algorithm

4.2.3 Singular Value Decomposition (Two-sided Jacobi Scheme[21])

In order to get a matrix A, where $AA^T = \Sigma$, the easiest way is Cholesky decomposition, but, unfortunately, the numbers can become negative because of round-off errors, in which case the square-root algorithm cannot continue. So in this situation, using singular value decomposition is a good choice.

We consider the standard eigenvalue problem

$$Bx = \lambda x \tag{4.5}$$

where B is a real $n*n$-dense symmetric matrix, which is covariance Σ in this project. One of the best known methods for determining all the eigenpairs of Eq. (4.6) was developed by the 19th century mathematician, Jacobi. We recall that Jacobi's sequential method reduces the matrix B to the diagonal form by an infinite sequence of plane rotations

$$B_{k+1} = V_k B_k V_k^T, \qquad k = 1, 2, \cdots \tag{4.6}$$

$$V = \begin{bmatrix} 1 & & & & & & & & 0 \\ & \ddots & & & & & & & \\ & & \cos\theta & 0 & \cdots & 0 & \sin\theta & & \\ & & 0 & 1 & \cdots & 0 & 0 & & \\ & & \vdots & \vdots & \ddots & \vdots & \vdots & & \\ & & 0 & 0 & \cdots & 1 & 0 & & \\ & & -\sin\theta & 0 & \cdots & 0 & \cos\theta & & \\ & & & & & & & \ddots & \\ & & & & & & & & \ddots \\ 0 & & & & & & & & 1 \end{bmatrix} \tag{4.7}$$

where $B_1 \equiv B$, and $V_k = V_k(i, j, \theta_{ij}^k)$ is a rotation of the $(i, j) - plane$ where $v_{ii}^k = v_{jj}^k = c_k = \cos\theta_{ij}^k$ and $v_{ij}^k = -v_{ji}^k = s_k = \sin\theta_{ij}^k$. The angle θ_{ij}^k is determined so that $b_{ij}^{k+1} = b_{ji}^{k+1} = 0$, or $\tan 2\theta_{ij}^k = \frac{2b_{ij}^k}{b_{ii}^k - b_{jj}^k}$, where $\left|\theta_{ij}^k\right| \le \frac{1}{4}\pi$.

For numerical stability, we determine the plane rotation by $c_k = \frac{1}{\sqrt{1+t_k^2}}$ and $s_k = c_k t_k$, where $a_k = \cot 2\theta_{ij}^k$ and $t_k = \frac{\text{sign} a_k}{|a_k| + \sqrt{1+a_k^2}}$. Each B_{k+1} remains symmetric and differs from B_k only in rows and columns i and j, where the modified elements are given by

$$b_{ii}^{k+1} = b_{ii}^k + t_k b_{ij}^k$$

$$b_{jj}^{k+1} = b_{jj}^k - t_k b_{ij}^k$$

$$b_{ir}^{k+1} = c_k b_{ir}^k + s_k b_{jr}^k \qquad r \neq i, j$$

$$b_{jr}^{k+1} = -s_k b_{ir}^k + c_k b_{jr}^k \qquad r \neq i, j$$

$$b_{ij}^{k+1} = b_{ji}^{k+1} = 0$$

So, matrix B can be decomposed into $B = V\lambda V^T = (V\lambda^{1/2})(\lambda^{1/2}V)^T$, where λ diagonal matrix, $\lambda \approx V_n \cdots V_k \cdots V_1 B$, $V \approx V_n \cdots V_k \cdots V_1 E$, and E is unit matrix. We can get $A = V\lambda^{1/2}$ and $AA^T = \Sigma$.

Multiplicative congruential Jacobi rotation is matrix multiplication. So we can do parallel computing as matrix multiplication.

4.2.4 Uniform Distribution (u)

Multiplicative congruential algorithm is the basis for many of the random number generators in use today. It involves three integer parameters, a, c, and m, and an initial value, x_0, called the seed. A sequence of integers is defined by

$$x_{k+1} = a * x_k + c \mod \text{m} \qquad (4.8)$$

Some of the linear or multiplicative generators which have been suggested are the following Table 2:

Table 2: Some Suggested Linear and Multiplicative Random Number Generators [22]

m	a	c	
2^{31} - 1	7^5 = 16807	0	Lewis, Goodman, Miller (1969)IBM
2^{31} - 1	6303600016	0	Fishman (Simscript II)
2^{31} - 1	742938285	0	Fishman and Moore
2^{31}	65539	0	RANDU
2^{32}	69069	1	Super-Duper (Marsaglia)
2^{32}	3934873077	0	Fishman and Moore
2^{32}	3141592653	1	DERIVE
2^{32}	663608941	0	Ahrens (C-RAND)
2^{32}	134775813	1	Turbo-Pascal, Version 7 (period= 2^{32})
2^{35}	5^{13}	0	APPLE
10^{12} - 11	427419669081	0	MAPLE
2^{59}	13^{13}	0	NAG
2^{61} - 1	$2^{20} - 2^{19}$	0	Wu (1997)

In the 1960s, the Scientific Subroutine Package (SSP) on IBM mainframe computers included a random number generator named RANDU. It has parameters a = 65539, c = 0, and m = 2^{31}. After test all the method list in the table, RANDU is one of the best methods to generate uniform random number. So in this project, I use RANDU to generate pseudo random number. The parallel computing algorithm shows in Figure 13 . There are $ms/1000+1$blocks and every block has n threads, and each thread generate random number separately.

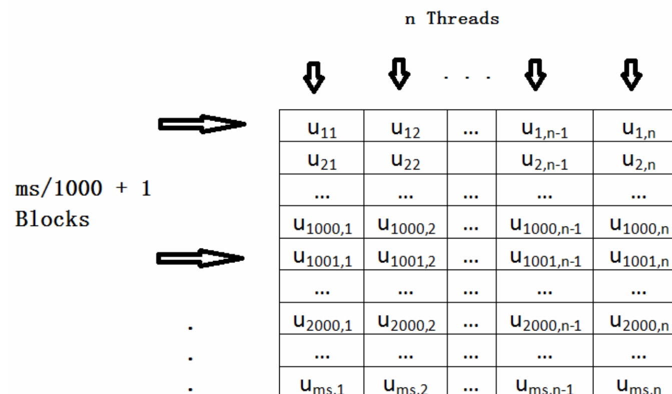


Figure 13: Uniform Distribution parallel algorithm

4.2.5 Box-Muller

A Box–Muller transform (by George Edward Pelham Box and Mervin Edgar Muller 1958[23]) is a method of generating pairs of independent standard normally distributed (zero expectation, unit variance) random numbers, given a source of uniformly distributed random numbers.

Suppose U_1 and U_2 are independent random variables that are uniformly distributed in the interval (0, 1]. Let

$$Z_0 = R * \cos\theta = \sqrt{-2\ln U_1}\cos(2\pi U_2) \quad (4.9)$$

and

$$Z_1 = R * \sin\theta = \sqrt{-2\ln U_1}\sin(2\pi U_2) \quad (2.8)$$

Then Z_0 and Z_1 are independent random variables with a normal distribution of standard deviation 1. The parallel computing algorithm shows in Figure 14. There are ms blocks and every block has $n/2$ threads. We don't use n threads in this algorithm, because reading data from memory is time consuming.

n/2 Threads

ms Blocks

$u_{1,1}$	$u_{1,2}$	...	$u_{1,n-1}$	$u_{1,n}$
$u_{2,1}$	$u_{2,2}$		$u_{2,n-1}$	$u_{2,n}$
$u_{3,1}$	$u_{3,2}$	...	$u_{3,n-1}$	$u_{3,n}$
...	...	...	...	...
$u_{ms,1}$	$u_{ms,2}$	...	$u_{ms,n-1}$	$u_{ms,n}$

Figure 14: Box–Muller parallel algorithm

4.2.6 Matrix Multiplication($mvd = u + A * Z$)

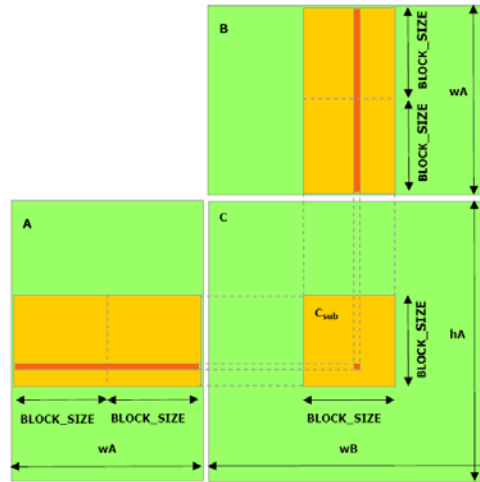


Figure 15: Matrixes Multiplication parallel algorithm

As illustrated in Figure 15, C_{sub} is equal to the product of two rectangular matrices: the sub-matrix of A of dimension (wA, block_size) that has the same line indices as C_{sub}, and the sub-matrix of B of dimension (block_size, wA) that has the same column indices as C_{sub}. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension block_size as necessary and C_{sub} is computed as the sum of the products of these square matrices. [16]

In this project, $mvd = u + Z \times A$, Z is $ms \times n$, A is $n \times n$, μ is $1 \times n$.

Thread Number = BLOCK_SIZE * BLOCK_SIZE and Block Number = ms / Block_SIZE

The value of mvd is $r^{t+1}_{ms,n}$, we mentioned in the algorithm.

4.3 Calculate Portfolio Value

$$portofolio_i = \sum_{j=1}^{n} (1 + mvd_{i,j}) \times lastprice_j \times shares_j \qquad i = 1, \cdots, ms; \quad j = 1, \cdots, n$$

$portofolio$ is $ms \times n$, mvd is $ms \times n$, $lastprice$ is $1 \times n$, $shares$ is $1 \times n$.

Because this is a matrix multiply two vectors, the fastest and easiest way to do parallel is to set ms threads, each of which calculate $portofolio_i = \sum_{j=1}^{n} (1 + mvd_{i,j}) \times lastprice_j \times shares_j$ (see Figure 16).

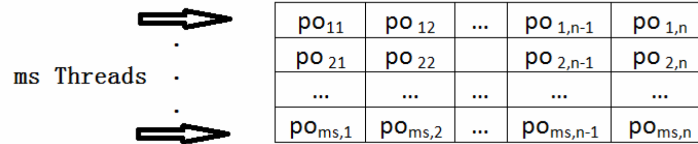


Figure 16: Calculate portfolio parallel algorithm

4.4 Merge sort

In computer science, merge sort is a sorting algorithm for rearranging lists into a specified order. It can be seen as a good example of the divide and conquer algorithmic paradigm.

Conceptually, merge sort works as follows steps, and a simple example is showed in Figure 17:

- Divide the unsorted list into two sublists of about half the size.
- Sort each of the two sublists.
- Merge the two sorted sublists back into one sorted list.

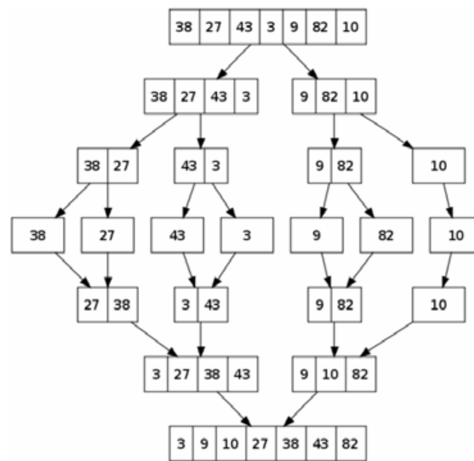


Figure 17: Merge Sort algorithm example

If the number of sorting elements is power of 2, merge sort is the fastest algorithm in all sorting algorithms. The time complexity of parallel computing is ($O(1+2+\cdots+\lg(n)) = O(\lg(n)^2)$). For example, when $n = 8$, the parallel loop steps are 6, while odd-even needs parallel loop step are 8 (Figure 18). And when $n = 64$, the parallel loop steps for merge sort are 21.

So in this project, we use merger sort to do sorting parallel computing.

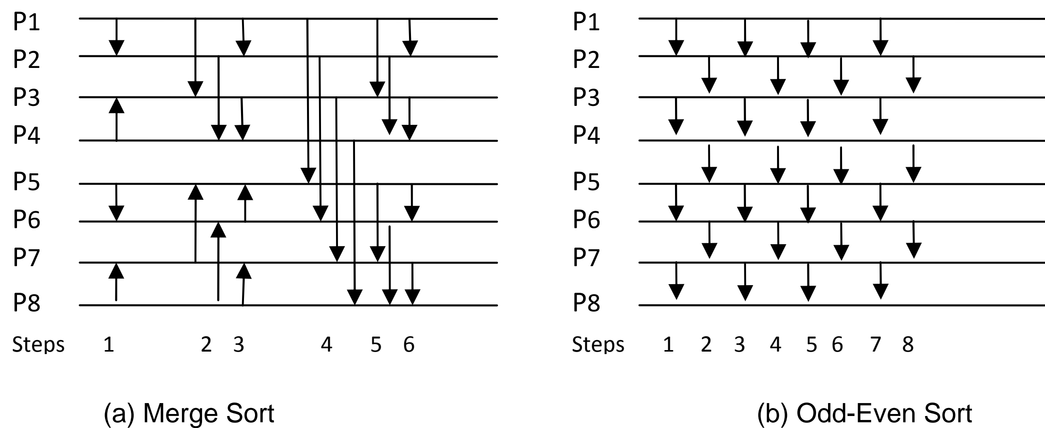


Figure 18: Parallel algorithm with $n = 8$

5. Experiments

This project is to test the performance of Monte Carlo simulation using GPU and CPU. We don't use very complex finance model; and we assume that all the assets in the portfolio are stocks, no future, option, and any other derivatives. And this model is used to estimate the Value at Risk in a day.

The computer we used to do simulation has the following properties. CPU processor is Intel Xeon, E5410 @2.33GHz (2 processors), installed memory (RAM0 is 8.00 GB (3.25 GB usable), system type is 32-bit operating system. GPU is NVIDIA Quadro FX 3700, and its main performance shows in Table 3. NVIDIA Quadro FX 3700 has 128 parallel processor cores, so it can run 128 blocks in parallel. And the maximum number of threads per block is 512, so the maximum number of threads in a grid is 65536.

Table 3 :NVIDIA Quadro FX 3700 Performance

Mobile Platform Generation	CUDA Parallel Processor Cores	Memory Size	Memory Type	Memory Interface	Memory Bandwidth
Centrino 2	128	1 GB	GDDR3	256-bit	51.2 GB

We use CUDA and Matlab to build the Monte Carlo Model separately. The reason why we not use C program is that: first, we can very easily get current and historical stock price with Matlab package function. Using following codes in Matlab, we can get the last day close price of a stock, and the history open price and close price in a time period. The following Matlab codes use 'fetch' to get 'GE' stock information from Yahoo Finance. Open and Close historical data of 'GE' are from 06/01/2008 to 06/30/2010.

```
y = yahoo;
last = fetch(y, 'GE','Last');
open = fetch(y, 'GE', 'Open', '06/01/2008', '06/30/2010');
close = fetch(y, 'GE', 'close', '06/01/2008', '06/30/2010');
```

Second reason is that we assumed the simplest condition, so Matlab program is very easy, and if we can get the result that CUDA is faster than Matlab in this case, it much faster

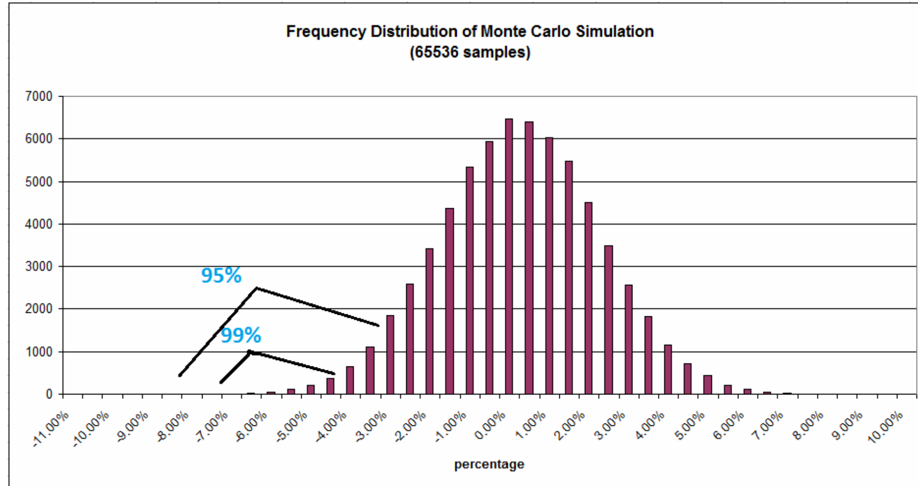
than C program absolutely, according to the conclusion we get from the matrix multiplication example.

Since loading data using Matlab is very slow when the data is very large, so we saved all the data in EXCEL, and the program read from EXCEL when computing.

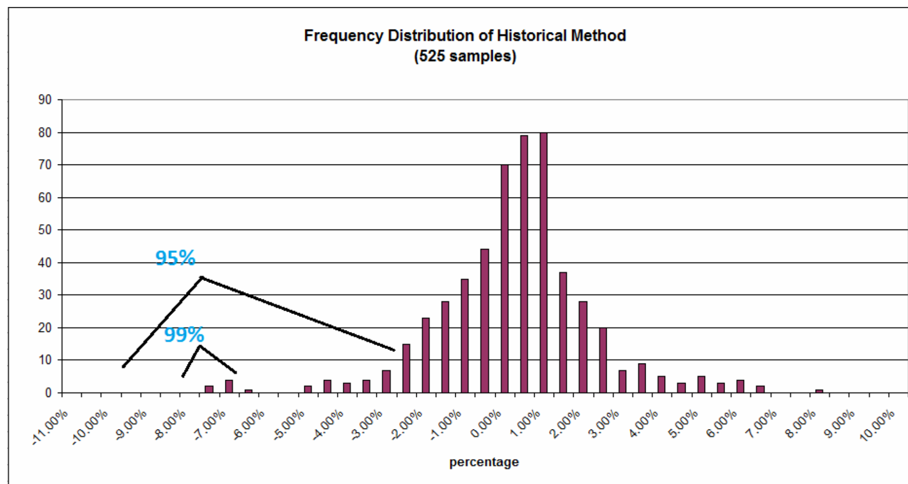
We conduct two different sets of experiment to do Monte Carlo Simulation. In the first experiment, we use $n = 16$ stocks, and in the second experiment, we use $n = 192$ stocks. All the history data of assets are from 06/01/2008 to 06/30/2010. The $open$ matrix is the open price of the assets in the business day from 06/01/2008 to 06/30/2010, which is $525 \times n$ matrix; and the $close$ matrix is the open price of the assets in the business day from 06/01/2008 to 06/30/2010, which is also $525 \times n$ matrix. The $lastprice$ matrix is the previous business day price of the assets, assuming current day is 10/30/2010, $lastprice$ is also $1 \times n$ matrix. The $shares$ matrix is the shares of every asset, which is also $1 \times n$ matrix. And we assume the confidence level is 95%. So the output is the maximum loss in the 95% confidence level on current day 10/30/2010.

Result and Discussion

First we discuss the result of Monte Carlo Simulation and Historical Method. Figure 19 (a) is the frequency of Monte Carlo simulated profit-loss rate of the portfolio. In the 99% confidence level in a day, the loss rate is 4.5%, and in the 95% confidence level, the loss rate is between 3.0% and 3.5%. Figure 19 (b) is the frequency of historical method profit-loss rate of the portfolio. In the 99% confidence level in a day, the loss rate is 7%, and in the 95% confidence level, the loss rate is 3.0%.



(a)

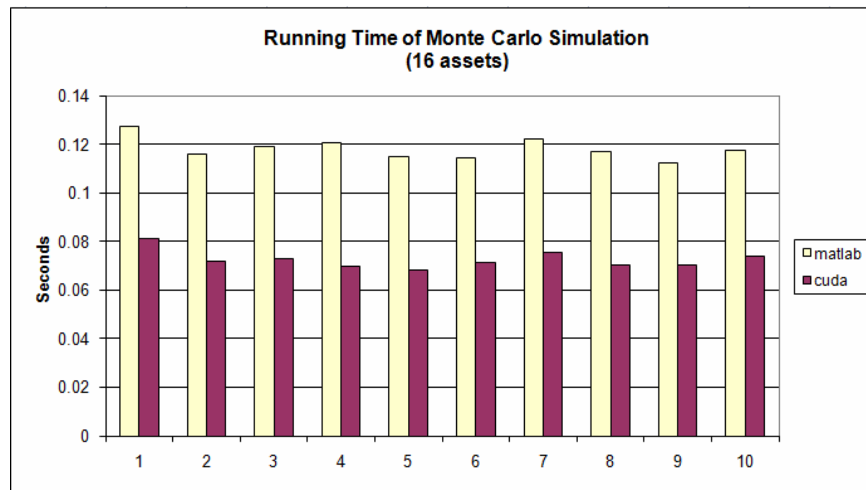


(b)

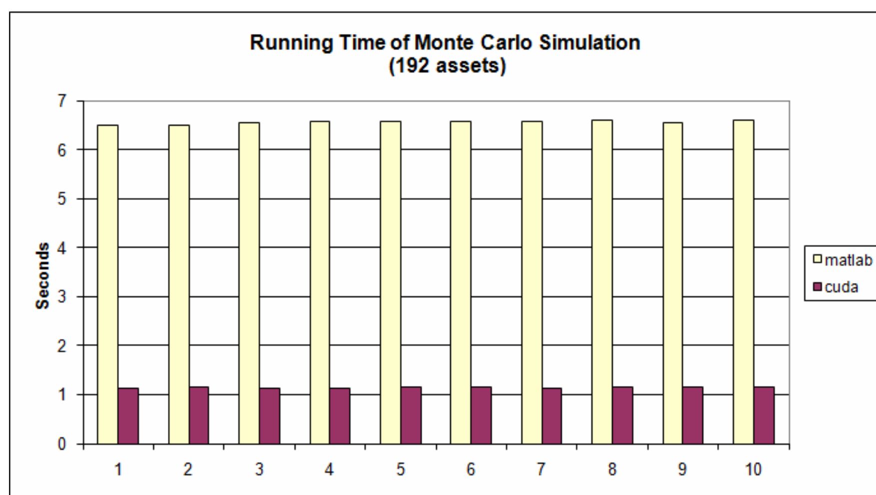
Figure 19: Frequency Distribution of Monte Carlo Simulation and Historical Method

Then we discuss about the running time of Monte Carlo Simulation in GPU using CUDA programming and in CPU using Matlab. Figure 20 (a) is the running time when assets number is 16, Figure 20 (b) is the running time when assets number is 192. We can find that when assets number is 16, CUDA is two times faster than Matlab, and when assets number is 192, CUDA is five times faster then Matlab. This result shows GPU is much faster than CPU. According to the experiment about matrix multiplication, we know that Matlab is very fast in simple matrix computing. Because we assume very simple condition to do Monte Carlo Simulation, Matlab just needs ten lines of programming to make this model, while in CUDA, we need hundreds of lines. We don't do the C programming, because CUDA will be thousands of times faster than C

absolutely. There are lots of matrix operations in the programming, and the largest matrix size is 65536×192. In this case, C programming will be very slowly.



(a)



(b)

Figure 20: Running Time of Monte Carlo Simulation

6. Conclusion

In this paper, we have implemented the Monte Carlo Simulation based VaR estimation using CUDA. This paper describes the detailed computing algorithm by leveraging the parallel computation capability of CUDA. We run two different experiments to compare CUDA results and Matlab based results, with one portfolio having 16 assets and the other having 192 assets. The results show that using CUDA in GPU can greatly improve the performance of Monte Carlo Simulation.

In the future, we will consider to achieve the real-time VaR estimation, and to build more complicated financial model to support most of VaR cases.

7. Reference

[1] Jorion, P. (1997): “Value at Risk”, The New Benchmark for Controlling Derivatives Risk, McGraw Hill, New York.

[2] Ashok Srinivasan, Ajay Shah (2001): “Improved techniques for using Monte Carlo in VaR estimation”, http://www.cs.fsu.edu/~asriniva/papers/nsefinal.pdf

[3] Stephen Lawrence (2000): “Value at Risk Incorporating Dynamic Portfolio Management”, No 147, Computing in Economics and Finance 2000 from Society for Computational Economics.

[4] DOWD, K. (1998): “Beyond Value-at-Risk: The New Science of Risk Management”, John Wiley & Sons, London.

[5] SAUNDERS, A. (1999): “Financial Institutions Management: A modern Perspective (3rd ed.)”, Irwin Series in Finance, McGraw-Hill, New York.

[6] DUFFIE, D. and J. PAN (1997): “An Overview of Value-at-Risk”, Journal of Derivatives, Vol. 4, No. 3, 7-49.

[7] CARDENAS, J., E. FRUCHARD, J.-F. PICRON, C. REYES, K. WALTERS, W. YANG (1999): “Monte-Carlo within a Day: Calculating Intra-Day VAR Using Monte-Carlo”, Risk, Vol. 12, No. 2, 55-60.

[8] ROUVINEZ, C. (1997): “Going Greek with VAR”, Risk, Vol. 10, No. 2, 57-65.

[9] JAMSHIDIAN, F. and Y. ZHU (1997): “Scenario Simulation: Theory and Methodology”, Finance and Stochastics, Vol. 1, No. 1, 43-67.

[10] ABKEN, P. (2000): “An Empirical Evaluation of Value-at-Risk by Scenario Simulation”, Journal of Derivatives, Vol. 7, No. 4, 12-29.

[11] Embrechts, P. Klüppelberg, C. and Mikosch, T. (2003): “Modelling Extremal Events for Insurance and Finance” Springer-Verlag, 648 pages, corr. 4th printing, 1st ed.

[12] Lucas, A. and P. Klaassen (1998): “Extreme Returns, Downside Risk, and Optimal Asset Allocation”. Journal of Portfolio Management, Fall, 71-79.

[13] Bollerslev, T., R.F. Engle and D.B. Nelson (1994), "ARCH Models," in R.F. Engle and D. McFadden (eds.), Handbook of Econometrics, Volume IV, 2959-3038. Amsterdam: North-Holland.

[14] Andrey Rogachev, (2002): "Dynamic Value-at-Risk", http://www.fmpm.org/docs/6th/Papers_6/Papers_Netz/SGF658b.pdf

[15] Dean Fantazzini (2009): "Value at Risk for High-Dimensional Portfolios: A Dynamic Grouped-T Copula Approach", The VAR IMPLEMENTATION HANDBOOK, McGraw-Hill, pp. 253-282, 2009

[16] "CUDA programming guide", version 3.0, 2/20/2010

[17] Michael Feldman (2008): "GPUs Finding A New Role on Wall Street", http://www.hpcwire.com/specialfeatures/hpws08/features/GPUs_Finding_A_New_Role_on_Wall_Street.html

[18] Greg N. Gregoriou (2009): "The VaR implementation handbook" McGraw-Hill; 1 edition

[19] Matthew Dixon, Jike Chong, Kurt Keutzer (2009): "Acceleration of market value-at-risk estimation", Proceeding WHPCF '09 Proceedings of the 2nd Workshop on High Performance Computational Finance.

[20] J.D. Cabedo and I. Moya (2003): "Estimating oil price Value at Risk using the historical simulation Approach", Energy Economics, v25, 239-253.

[21] Erricos John Kontoghiorghes (2005): "Handbook of Parallel Computing and Statistics", Chapman and Hall/CRC; 1 edition

[22] Don L. McLeish (2005): "Monte Carlo Simulation and Finance", Wiley; 1 edition

[23] G. E. P. Box and Mervin E. Muller (1958): "A Note on the Generation of Random Normal Deviates", The Annals of Mathematical Statistics, Vol. 29, No. 2 pp. 610–611(wiki)

8. Appendix

(1) Value at Risk . m

```
function [current VaRmc VaRnor] = Stock192_VaR(p)
nvmex -f nvmexopts.bat Value_at_Risk_Stock192.cu -IC:\cuda\include -LC:\cuda\lib -lcufft -lcudart;

so = xlsread('C:\Users\CUDA_admin\Desktop\ww\VaR\VaR program\stockprice','SO','A2:CR526');
sc = xlsread('C:\Users\CUDA_admin\Desktop\ww\VaR\VaR program\stockprice','SC','A3:CR527');
last = xlsread('C:\Users\CUDA_admin\Desktop\ww\VaR\VaR program\stockprice','SC','A530:CR530');
so = single([so so]);
sc = single([sc sc]);
last = single([last last]);
share = [10; 20; 30; 10; 15; 5; 25; 60; 25; 20; 18; 7; 9; 13; 34; 26];
share = [share; share; share; share; share; share; share; share; share; share; share; share];
share = single(share);
current = sum(last .* share');

% CUDA-GPU
tic;
VaRmc= Value_at_Risk_Stock192(so, sc, last, share, single(p));
toc;

% Matlab-CPU
tic;
level = floor(65536*(1-p));
plRate = (sc - so) ./ so;
mu = mean(plRate,1);
sigma = cov(plRate);
mvd = single(mvnrnd(mu, sigma, 65536));
for i = 1 : 192
   stockPrice(:, i) = last(i) * (1 + mvd(:, i) );
end
portfolioPrice = stockPrice * share;
s= sort(portfolioPrice);
VaRnor = s(level);
toc;
```

(2) Value_at_Risk_Stock192.cu

```
#include "mex.h"
#include "math.h"
#include "cuda.h"
#include "cuda_runtime.h"

#define PI 3.14159265358979f
#define BLOCK_SIZE 16
#define MAX 32768


/***************************Profit and Loss Rate******************************/
__global__ void profitLossKernel(float *open, float *close, float *plRate, int mrows, int ncols)
{
   int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if( xIndex < mrows && yIndex < ncols  )
      plRate[yIndex * mrows + xIndex] = (close[yIndex * mrows + xIndex] - open[yIndex * mrows +
xIndex])/open[yIndex * mrows + xIndex];
}

/***************************mu******************************/
__global__ void meanParallelKernel(float *plRate, float *mu, int mrows, int k)
{
   int yIndex = blockIdx.x;
   int xIndex = threadIdx.x * k;
   int i = 0;

   extern __shared__ float shared[];
   shared[threadIdx.x] = 0;
   for(i = 0; i < k; i++)
   {
      if((xIndex + i) < mrows) // && yIndex < ncols
         shared[threadIdx.x] += plRate[yIndex * mrows + (xIndex + i)];
   }
```

```
    __syncthreads();
    mu[yIndex] = 0;
    for(i = 0; i < 32; i++)
        mu[yIndex] += shared[i];
    mu[yIndex] = mu[yIndex] / mrows;
}


/***************************sigma******************************/
__global__ void covParallelKernel(float *plRate, float *mu, float *sigma, int mrows, int ncols, int k)
{
    int xIndex = threadIdx.x * k;
    int i;

    extern __shared__ float shared[];
    shared[threadIdx.x] = 0;
    for(i = 0; i < k; i++)
    {
        if((xIndex + i) < mrows) // && yIndex < ncols
        {
            int x1 = blockIdx.x * mrows + (xIndex + i);
            int x2 = blockIdx.y * mrows + (xIndex + i);
            shared[threadIdx.x] += (plRate[x1] - mu[blockIdx.x]) * (plRate[x2] - mu[blockIdx.y]);
        }
    }

    __syncthreads();
    sigma[blockIdx.y * ncols + blockIdx.x] = 0;
    for(i = 0; i < 32; i++)
        sigma[blockIdx.y * ncols + blockIdx.x] += shared[i];
    sigma[blockIdx.y * ncols + blockIdx.x] = sigma[blockIdx.y * ncols + blockIdx.x] / mrows;
}

/*************************SVD*********************************/
__global__ void SingluarValueDecompositionKernel(float *B, float *F, int n)
{
    int j = threadIdx.y;
```

```
__shared__ float A[192][4];//p=0/1;q=2/3
__shared__ float E[192][2];//p=0;q=1
int p,q,l;
for(l = 0; l < 3; l ++)
{
   for(p = 0; p < n; p++)
   {
      for(q = p + 1; q < n; q++)
      {
            if(p == 0 && q == 1 && l == 0)
            {
               F[j * n + j] = 1;
            }
            __syncthreads();

            if(abs(B[q*n+p]) >0.00001)
             {
               if(threadIdx.x == 0)
               {
                  A[j][0] = B[j * n + p];
                  A[j][1] = B[p * n + j];
                  E[j][0] = F[p * n + j];
               }
               else if(threadIdx.x == 1)
               {
                  A[j][2] = B[j * n + q];
                  A[j][3] = B[q * n + j];
                  E[j][1] = F[q * n + j];
               }
            __syncthreads();

                    float w, t, cos, sin;
                    w = (B[p*n+p] - B[q*n+q]) / (2 * B[q*n+p]);
                    t = (w / abs(w)) / ((abs(w) + sqrt(1 + w * w)));
                    cos = 1 / sqrt(1 + t * t);
                    sin = t / sqrt(1 + t * t);
```

```
if(threadIdx.x == 0)  // i=p
{
   if(j == p)
   {
      B[p*n+p] = A[p][0] * cos * cos + A[q][2] * sin * sin + A[q][0] * 2 * sin * cos;
      F[p*n+p] = E[p][0] * cos + E[p][1] * sin;
   }
   else if(j == q)
   {
      B[q*n+p] = 0.5 * (A[q][2]  - A[p][0]) * 2 * sin * cos + A[q][0] * (2 * cos * cos - 1);
      F[q*n+p] =  -E[p][0] * sin + E[p][1] * cos;
   }
   else if(j != p && j != q)
   {
      B[j*n+p] = A[j][0] * cos + A[j][2] * sin;
      B[p*n+j] = A[j][1] * cos + A[j][3] * sin;//else if( j == p && i != p && i  != q)
      F[p*n+j] = E[j][0] * cos + E[j][1] * sin;
   }
}
else if(threadIdx.x == 1) // i=q
{
   if(j == q)
   {
      B[q*n+q] = A[p][0] * sin * sin + A[q][2] * cos * cos - A[q][0] * 2 * sin * cos;
      F[q*n+q] = -E[q][0] * sin + E[q][1] * cos;
   }
   else if(j == p)
   {
      B[p*n+q] = 0.5 * (A[q][2]  - A[p][0]) * 2 * sin * cos + A[q][0] * (2 * cos * cos - 1);
      F[p*n+q] =  E[q][0] * cos + E[q][1] * sin;
   }
   else if(j != p && j != q)
   {
      B[j*n+q] = - A[j][0] * sin + A[j][2] * cos;
      B[q*n+j] = - A[j][1] * sin + A[j][3] * cos;//else if( j == q && i != p && i != q)
      F[q*n+j] = - E[j][0] * sin + E[j][1] * cos;
   }
```

```
                }
            }// end-if(abs(A[q*n+p]) >0.000001)
            __syncthreads();
        }
    }
  }
  __syncthreads();
}

__global__ void GetSVD(float *SVD, float *F, float *B, int n)
{
  int j = threadIdx.x;
  for(int k = 0; k < 2; k++)
  {
    int i = gridDim.x * k + blockIdx.x;
    float x = sqrt(B[j*n+j]);
    if(x < 0.000001)
      x = x * 100000;
    else if(x < 0.00001)
      x = x * 10000;
    else if(x < 0.0001)
      x = x * 1000;
    else if(x < 0.001)
      x = x * 100;
    else if(x < 0.01)
      x = x * 10;
    SVD[j * n + i] = F[j * n + i] * x; //F[j * n + i];//
  }
}


/***************************Uniform Distribution******************************/
__global__ void UniformKernel1(float *ND, float *ud, int n, int m, float M)
{
  int j = threadIdx.x;
  int lambda, mu;
  lambda = 65539;
```

```
    mu = 0;
    int g = blockDim.x;
    for ( int i = 0; i < 1000; i++)
    {
        if( blockIdx.x * 1000 + i < m)
        {
            if(i == 0)
            {
                ND[j * m + (blockIdx.x * 1000 + i)] = ud[j + blockIdx.x * n];

            }
            else
            {
                int x = floor((lambda * ND[j * m + (blockIdx.x * 1000 + i) - 1] + mu) / M);
                ND[j * m + (blockIdx.x * 1000 + i)] = lambda * ND[j * m + (blockIdx.x * 1000 + i) - 1] + mu - x * M;
                if(ND[j * m + (blockIdx.x * 1000 + i)] == 0)
                    ND[j * m + (blockIdx.x * 1000 + i)] = 11111111;
            }
        }
    }
}

__global__ void UniformKernel2(float *ND, int n, int m, float M)
{
    int a = m / gridDim.x +1;
    for(int i = 0; i < a; i++)
    {
        int xIndex = i * gridDim.x + blockIdx.x;
        int yIndex = threadIdx.x;

        if(xIndex < m && yIndex < n)
        {
            ND[yIndex * m + xIndex] /= M;
        }
    }
}
```

```
/***************************Normal Distribution******************************/
__global__ void BoxMullerKernel(float *ND, int n, int m)
{
    int a = m / gridDim.x +1;
    for(int i = 0; i < a; i++)
    {
        int xIndex = i * gridDim.x + blockIdx.x;
        int yIndex = threadIdx.x;

        if(xIndex < m && (yIndex + n/2) < n)
        {
            float r = sqrt(-2.0f * logf(ND[yIndex * m + xIndex]));
            float phi = 2 * PI * ND[(yIndex + n/2) * m + xIndex];
            ND[yIndex * m + xIndex] = r * __cosf(phi);
            ND[(yIndex + n/2) * m + xIndex] = r * __sinf(phi);
        }
    }
}


/***************************Matrix Multiplication  C=A*B + mu ******************************/
__global__ void Muld(float* A, float* B, float *mu, int hA, int wA, float* C)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    for (int i = 0; i < hA / (BLOCK_SIZE * gridDim.x); i++)
    {
        int rx = i * gridDim.x * BLOCK_SIZE;
        int xIndex = rx + blockIdx.x * blockDim.x + threadIdx.x;
        int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
        if(xIndex < hA && yIndex < wA)
        {
            int aBegin = rx + BLOCK_SIZE * bx;
            int aEnd = hA * (gridDim.y - 1) * BLOCK_SIZE + aBegin;
```

```
            int aStep = hA * BLOCK_SIZE;
            int bBegin = wA * BLOCK_SIZE * by;
            int bStep = BLOCK_SIZE;
            float Csub = 0;

            for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
            {
                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
                As[tx][ty] = A[a + hA * ty + tx];
                Bs[tx][ty] = B[b + wA * ty + tx];
                __syncthreads();
                for (int k = 0; k < BLOCK_SIZE; ++k)
                    Csub += As[tx][k] * Bs[k][ty];
                __syncthreads();
            }
            C[yIndex * hA + xIndex] = Csub + mu[yIndex];
        }
        __syncthreads();
    }
}

/**************************Portfolio Price*****************************/
__global__ void portfolioPriceKernel(float *lastPrice, float *MVD, float * share, float *portfolioPrice, int
mrows, int ncols, int k)
{
       int rx = blockDim.x * blockIdx.x + threadIdx.x;
       int xIndex;

       for(int i = 0; i < k; i++)
        {
            xIndex = rx + i * gridDim.x * 16 * 16;
            if( xIndex < mrows )
            {
                float temp = 0;
                for(int j = 0; j < ncols; j++)
                {
```

```
                temp +=  lastPrice[j] * (1 + MVD[j * mrows + xIndex])  * share[j];
            }
            portfolioPrice[xIndex] = temp;
        }
    }
}

/**************************Merge Sort*****************************/
__global__ void mergeSortKernel(float *array, int i, int j)
{
   int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
   int multiple = (int) pow(2.0, i);
   int d = (int) pow(2.0, j-1);
   int step = multiple / 2 / d;
   int x1;
   float temp;

   if(step == 1)
   {
      x1 =  2 * xIndex;
   }
   else
   {
      if(xIndex < step)
         x1 = xIndex;
      else if(xIndex >= step && xIndex % step == 0)
         x1 = xIndex * 2;
      else
         x1 = xIndex * 2 - xIndex % step;
   }

   if((x1 / multiple) % 2 == 0) //x1/multiple is even, min up and max down
   {
      if(array[x1] > array[x1 + step])
      {
         temp = array[x1];
         array[x1] = array[x1 + step];
```

```
            array[x1 + step] = temp;
        }
    }
    else  //x1/multiple is odd, min down and max up
    {
        if(array[x1] < array[x1 + step])
        {
            temp = array[x1];
            array[x1] = array[x1 + step];
            array[x1 + step] = temp;
        }
    }
}


void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    float *open, *close, *lastPrice, *share, *level; //five inputs

    if (nrhs != 5)
        mexErrMsgTxt("Five input required!");
    if (nlhs > 3)
         mexErrMsgTxt("Too many output arguments!");
    if ( !mxIsSingle(prhs[0]) || !mxIsSingle(prhs[1]) || !mxIsSingle(prhs[2]) || !mxIsSingle(prhs[3])
|| !mxIsSingle(prhs[4]))
         mexErrMsgTxt("Input arry must be single precision!");
    int mrows = mxGetM(prhs[0]);
    int ncols = mxGetN(prhs[0]);


    /************************PART I - Caculate plRate, mu, and sigma******************************/
    float *mu, *sigma, *plRate;

   if( cudaMalloc((void**) &open, sizeof(float) * ncols * mrows) != cudaSuccess )
       mexErrMsgTxt("Memory allocating open failure on GPU!");
   if( cudaMalloc((void**) &close, sizeof(float) * ncols * mrows) != cudaSuccess )
       mexErrMsgTxt("Memory allocating close failure on GPU!");
```

```
    if( cudaMalloc((void**) &lastPrice, sizeof(float) * ncols * 1) != cudaSuccess )
        mexErrMsgTxt("Memory allocating lastPrice failure on GPU!");
    if( cudaMalloc((void**) &share, sizeof(float) * ncols * 1) != cudaSuccess )
        mexErrMsgTxt("Memory allocating share failure on GPU!");
    cudaMemcpy(open, (float*)mxGetData(prhs[0]), sizeof(float) * ncols * mrows,
cudaMemcpyHostToDevice);
    cudaMemcpy(close, (float*)mxGetData(prhs[1]), sizeof(float) * ncols * mrows,
cudaMemcpyHostToDevice);
    cudaMemcpy(lastPrice, (float*)mxGetData(prhs[2]), sizeof(float) * ncols, cudaMemcpyHostToDevice);
    cudaMemcpy(share, (float*)mxGetData(prhs[3]), sizeof(float) * ncols, cudaMemcpyHostToDevice);

    //plRate
    if( cudaMalloc((void**) &plRate, sizeof(float) * ncols * mrows) != cudaSuccess )
        mexErrMsgTxt("Memory allocating plRate failure on GPU!");
    int blocky = ncols/BLOCK_SIZE + 1;
    int blockx = mrows/BLOCK_SIZE + 1;
    dim3 dimBlock1(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid1(blockx, blocky);
    profitLossKernel <<<dimGrid1, dimBlock1>>> (open, close, plRate, mrows, ncols);
   //Caculate mu and sigma
    if( cudaMalloc((void**) &mu, sizeof(float) * ncols * 1) != cudaSuccess )
        mexErrMsgTxt("Memory allocating mu failure on GPU!");
    if( cudaMalloc((void**) &sigma, sizeof(float) * ncols * ncols) != cudaSuccess )
        mexErrMsgTxt("Memory allocating sigma failure on GPU!");

    int threadNum = 32;
    int blockNum = ncols;
    int k;
    if(mrows % 32 == 0)
        k = mrows / 32;
    else
        k = mrows / 32 + 1;
    //mu
    dim3 dimBlock2(threadNum);
    dim3 dimGrid2(blockNum);
    meanParallelKernel <<<dimGrid2, dimBlock2, threadNum * sizeof(float)>>> (plRate, mu, mrows, k);
    //sigma
```

```
    dim3 dimBlock3(threadNum);
    dim3 dimGrid3(ncols, ncols);
    covParallelKernel <<<dimGrid3, dimBlock3, threadNum * sizeof(float)>>> (plRate, mu, sigma, mrows,
ncols, k);

    cudaFree(open);
    cudaFree(close);
    cudaFree(plRate);

    /*****************************PART II: Multivariate Normal Distribution*******************************/
    //SVD
    float *F, *SVD;//B- sigma
    if( cudaMalloc((void**) &F, sizeof(float) * ncols * ncols) != cudaSuccess )
        mexErrMsgTxt("Memory allocating F failure on GPU!");
    if( cudaMalloc((void**) &SVD, sizeof(float) * ncols * ncols) != cudaSuccess )
        mexErrMsgTxt("Memory allocating SVD failure on GPU!");

    dim3 dimBlock4(2,192);
    SingluarValueDecompositionKernel <<<1, dimBlock4>>> (sigma, F, ncols);
    GetSVD<<<192/2, 192>>>(SVD, F, sigma, ncols);
    cudaThreadSynchronize();
    cudaFree(sigma);
    cudaFree(F);

    //Normal distribution
    int m = 65536;      //samples
    int g = m/1000 + 1;
    float *ND, *MVD, *ud, *ini;

    if( cudaMalloc((void**) &ND, sizeof(float) * ncols * m) != cudaSuccess )
        mexErrMsgTxt("Memory allocating ND failure on GPU!");
    if( cudaMalloc((void**) &MVD, sizeof(float) * ncols * m) != cudaSuccess )
        mexErrMsgTxt("Memory allocating MVD failure on GPU!");
    if( cudaMalloc((void**) &ud, sizeof(float) * ncols * g) != cudaSuccess )
        mexErrMsgTxt("Memory allocating ud failure on GPU!");

    srand(clock());
```

```
float M = pow(2.0,31);
ini = (float *)malloc(sizeof(float) * ncols * g);
for(int i = 0; i < ncols * g; i++)
{
    ini[i]=floor(float(111111111 + rand()));
}
cudaMemcpy(ud, ini, sizeof(float) * ncols * g, cudaMemcpyHostToDevice);

UniformKernel1 <<<g, ncols>>> (ND, ud, ncols, m, M);
cudaThreadSynchronize();
dim3 dimGrid5(m / ncols + 1);
UniformKernel2 <<<dimGrid5, ncols>>> (ND, ncols, m, M);
cudaThreadSynchronize();
dim3 dimGrid6(2 * m / ncols + 1);
BoxMullerKernel<<<dimGrid6, ncols/2>>>(ND, ncols, m);
cudaThreadSynchronize();

dim3 dimBlock7(BLOCK_SIZE, BLOCK_SIZE);
blockx = m / BLOCK_SIZE;
blocky = ncols / BLOCK_SIZE;
int maxBlockx = MAX / (BLOCK_SIZE * BLOCK_SIZE * blocky);
if(blockx > maxBlockx)
    blockx =  maxBlockx;
dim3 dimGrid7(blockx, blocky);
Muld<<<dimGrid7, dimBlock7>>>(ND, SVD, mu, m, ncols, MVD);


free(ini);
cudaFree(ud);
cudaFree(SVD);
cudaFree(mu);
cudaFree(ND);


/*****************************PART III: Stock Price, all var, sort******************************/
float *portfolioPrice;
if( cudaMalloc((void**) &portfolioPrice, sizeof(float) * m) != cudaSuccess )
```

```
    mexErrMsgTxt("Memory allocating portfolioPrice failure on GPU!");
  dim3 dimBlock8(BLOCK_SIZE * BLOCK_SIZE);
 int blockx8 = MAX / (BLOCK_SIZE * BLOCK_SIZE);
 k = m/MAX;
 dim3 dimGrid8(blockx8);
 portfolioPriceKernel <<<dimGrid8, dimBlock8>>> (lastPrice, MVD, share, portfolioPrice, m, ncols, k);


 int x = log(65536.0) / log(2.0);
 for(int i = 1; i <= x; i++)
 {
    for(int j = 1; j <= i; j++)
    {
            mergeSortKernel <<<128, 256>>> (portfolioPrice, i, j);
    }
 }

 cudaThreadSynchronize();

  level = (float*)mxGetData(prhs[4]);
  float *var;
  float b = *level;
  int a =floor((1-b)*m);
  var = &portfolioPrice[a];
  plhs[0] = mxCreateNumericMatrix(1, 1, mxSINGLE_CLASS, mxREAL);//output VaR
  cudaMemcpy((float*)mxGetData(plhs[0]),  var, sizeof(float), cudaMemcpyDeviceToHost);


  cudaFree(MVD);
  cudaFree(lastPrice);
  cudaFree(share);
  cudaFree(portfolioPrice);
}
```