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

Wei Wu, Izidor Gertner

Abstract

Monte Carlo simulation is one of the best methods to calculate Value-at-Risk (VaR) of portfolios. But this method is time consuming especially when the simulated samples and the number of portfolio assets are very large. In this paper, we demonstrate how the graphics processing units (GPU) accelerate of Monte Carlo simulation for VaR estimation. First, we design the parallel computing Monte Carlo Method, which contains parallel algorithms of sorting, matrix multiplication and singular value decomposition, and parallel algorithms to generate random numbers and to calculate mean and standard deviation. Second, we compare the performance of three different ways to calculate VaR: the improved parallel computing Monte Carlo Method running in GPU, non-parallel Monte Carlo simulation using C program running in CPU, and Matlab program using Matlab functions running in CPU. Finally, we compare the performance of GPU-based Monte Carlo Simulation with different generation of GPUs: NVIDIA Quadro FX 3700 and NVIDIA Geforce 9800 GTX/9800 GTX+. The result shows that the performance of GPU NVIDIA Geforce 9800 GTX/9800 GTX+ is 2 times faster than GPU NVIDIA Quadro FX 3700 for this simulation case. And the performance of GPU (NVIDIA Geforce 9800 GTX/9800 GTX+) computing is about 11 times faster than Matlab, and 110 times faster than C program in CPU, which demonstrates clear the advantage to use GPU in VaR estimation.

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

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 till need to know how much risk to assume in exchange for these returns. Risk is the rate of losing money, and VaR based on that common-sense fact, 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?

In 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] This is the first using VaR ideas to solve financial questions. In 1955, JP Morgan’s RiskMetrics system increased the profile of Value at Risk substantially, and because the importance of Value at Risk, 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.

There are three general methods to estimate VaR: historical method, variance-covariance method and Monte Carlo simulation. 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. Ashok Srinivasan and Ajay Shah (2000) [2] improve the performance of Monte Carlo method based on deriving bounds for different matrix norms. 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. SPAN or TIMS also uses. When a trade happens, the positions of two economic agents are updated, and two VaR computations are required. What are 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. If the trading happened unevenly, a peak requirement will be 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[17], said GPU programming is the new kids on Wall Street, because that is making inroads across nearly every type of HPC application. 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.

2. Value at Risk Methodologies

In financial mathematics and financial risk management, Value at Risk is a widely used risk measure of the risk of loss on a specific portfolio of financial assets. Jorion (1997) defines Value at Risk as: "the expected maximum loss (or worst loss) over a target horizon within a given confidence interval."[1] For a given portfolio, probability and time horizon, VaR is defined as a threshold value such that the probability that the mark-to-market loss on the portfolio over the given time horizon exceeds this value (assuming normal markets and no trading in the portfolio) is the given probability level.[24]

So we can see that the VaR has three elements: the result we want to get is a threshold value, an estimate of investment loss, which can be either in dollar or percentage term; a given confidence level, typically either 95% or 99%, also can be said as the probability level 5% or 1%; a given time horizon, always a day, a month or a year.

For example, if we want know the VaR of a portfolio at 95% confidential level over a day, Figure1 shows the result. The curve represents a hypothetical frequency profit-and-loss probability by a day. It has mean 0.0002 and standard deviation 0.02017, and is in normal distribution. The 95% confidential level (5% probability) VaR point is -3%, which means there is 5% probability that the portfolio value will fall by more than 3% loss rate over a day period; if the portfolio current value is $1,000.00, then the maximum loss value is $30.00, which means 5% probability that the portfolio value will fall by more than $30.00 over a day period.

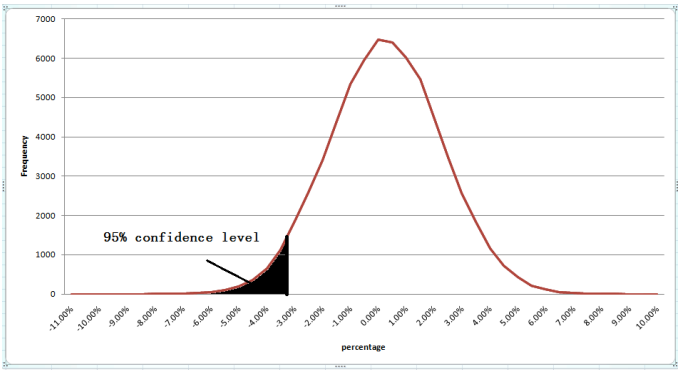


Figure1: One day 95% VaR of a hypothetical profit-and-loss probability frequency

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

2.1 Historical Method

Historical simulations represent the simplest way of estimating the Value at Risk for a portfolio. 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, this assumption make the Historical Method have big default, because future is not simply repeat according to history.

So the process of this method can be described in two steps:

Step1: we get the historical data in a time period and calculate the profit-and-loss rate.

Step2: we sort the profit-and-loss rate ascending, and then calculate VaR according to the confidential level. VaR(the maximum loss percent) is the n^{th} value in the ascending sorted profit-and-loss rate vector, where $n = \text{row numbers of profit} - \text{and} - \text{loss rate vector} \times (1 - \text{confidence level})$.

2.2 Variance-Covariance method

Since Value at Risk measures the threshold value that the value of an asset or portfolio will drop below a specified value in a particular time period in particular probability, 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 Table 1. But whether all the distributions of samples are exactly normal distribution, the assumption is the disadvantage of this method.

So the process also can be described in two steps.

Step1: we get the historical data in a time period and calculate the profit-and-loss rate.

Step2: we assume the loss is normal distribution, and then calculate the standard deviations.

Step3: we calculate VaR according to the formula listed in Table 1.

Table 1: Variance-covariance method

Confidence Level	VaR
95%	$-1.65\times\sigma$
99%	$-2.33\times\sigma$

* σ - Standard Deviation

2.3 Monte Carlo Simulation

Monte Carlo simulation relies on repeated random sampling to compute the results. It refers to any method that randomly generates trials. Monte Carlo method used to do VaR estimation are similar with historical method and variance-covariance method we mentioned before. The steps of Monte Carlo method are bellow.

Step1: we get the historical data in a time series and calculate the profit-and-loss rate.

Step2: we find the possibility distribution and its parameters according to the historical data. For example, if the profit-and-loss rate is in normal distribution, then we calculate its mean and standard deviation.

Step3: we generate the specific distribution random numbers based on parameters get in step2.

Step4: we sort these random numbers ascending and get VaR, which is the n^{th} in the sorted profit-and-loss rate, where $n = \text{row numbers of random number} \times (1 - \text{confidence level})$.

The strengths of Monte Carlo simulations can be seen when compared to the other two approaches for computing Value at Risk. Monte Carlo is 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 normal distribution in returns and use the unchanged formula. In contrast to the historical simulation approach, we begin with historical data but not simply assume to the future is repeat of history. 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 faster way to do Monte Carlo calculation.

3. Monte Carlo Simulation to Estimate Value at Risk

3.1 Algorithm of Monte Carlo Simulation of Single Asset

The algorithm of single asset Monte Carlo simulation is the same as we introduced in section 2.

The following example is to calculate single asset VaR using Monte Carlo Simulation. We generate 10000 hypothetical trials of daily returns for the 'GE' stock based on mean -0.0032 and standard deviation 0.0298 which are calculated from historical stock price profit-loss rate from 6/2/2008 to 6/30/2010. In Figure 2, 103 outcomes were less than and equal to -7%; 565 outcomes were less than and equal to -5%, and 272 outcomes were between -6% and -5%. This means the worst 1% probability maximum loss is less than and equal to 7%, and the worst 5% probability maximum loss is less than and equal to 5%. If we sort these 10000 tails simulated profit-and-loss rate results ascending, we also can get the worst 1% probability maximum loss is less than 7.04%, and the worst 5% probability maximum loss is less than 5.20%. So, the Monte Carlo simulation therefore leads to the following VAR-type conclusion: with 99% confidence, we do not expect to lose more than 7% during any given day; with 95% confidence, we do not expect to lose more than 5% during any given day.

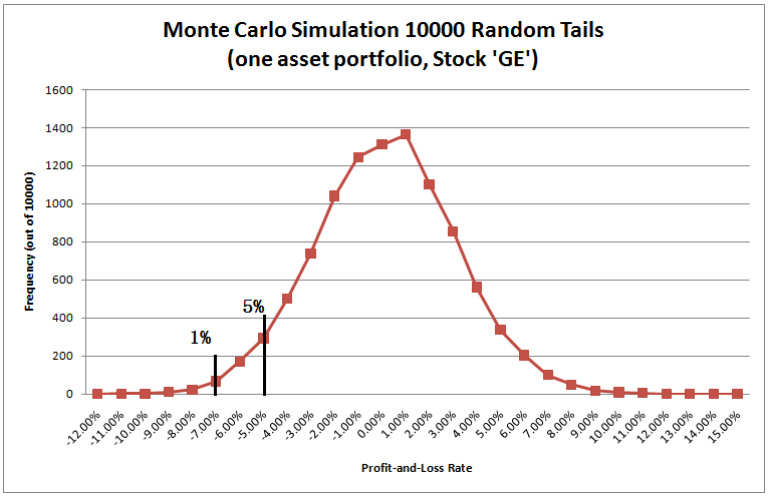


Figure 2: Monte Carlo method of VaR estimation for one asset portfolio

3.2 Parallel Algorithm of Monte Carlo Simulation of Multi Assets

In this section, we will describe the detail of VaR estimation in multi-assets 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 2, and the workflow is in Figure 3. We assume there are n stocks, and every stock has

m business day historical data. The input is the historical data of portfolio stocks, which includes open price and close price in the time horizon, the close price of the previous day v_n^t and $share_n$ of each asset in the portfolio. Based on the open and close historical data, we can get historical profit-and-loss rate ($r_{m,n}$), which will be used for further simulation. The second step is to calculate mean and covariance according to historical profit-and-loss rate $r_{m,n}$. Then, we use Monte Carlo simulation to generate ms tails of result - the simulated profit-and-loss rate $R_{ms,n}$ in multivariate normal distribution. Based on last asset price v_n^t and simulated profit-and-loss rate $R_{ms,n}$, we will then calculate $t+1$ portfolio price V_{ms}^{t+1}. Finally, we will sort V_{ms}^{t+1} and output the VaR of the given confidence level. The VaR result is in dollar; if VaR in dollar divide by $v_n^t \times share_n$, we can get VaR in percent.

Table 2: Algorithm for VaR estimation using Monte Carlo

Algorithm of VaR estimation using Monte Carlo
Input: Portfolio w (this portfolio w includes n assets) Output: VaR of portfolio w . Procedure: 1. Let v_n^t be the close value each asset in the portfolio at previous day; let $r_{m,n}$ be the historical profit-and-loss rate ($r_{m,n}$, m is the number of the historical data, n is the number of assets). 2. Calculate mean and standard covariance according to $r_{m,n}$. 3. Generate ms number of tails $R_{ms,n}$ using multivariate normal distribution method according to the mean and covariance we get at step 2. 4. Calculate the value of portfolio V_{ms}^{t+1} at each tail using $V_{ms}^{t+1} = (1+R_{ms,n}) \times v_n^t \times share_n$. 5. Sort the portfolio price V_{ms}^{t+1} ascending, and VaR is the i^{th} in the sorted vector, where $i = ms \times (1 - \text{confidence level})$.

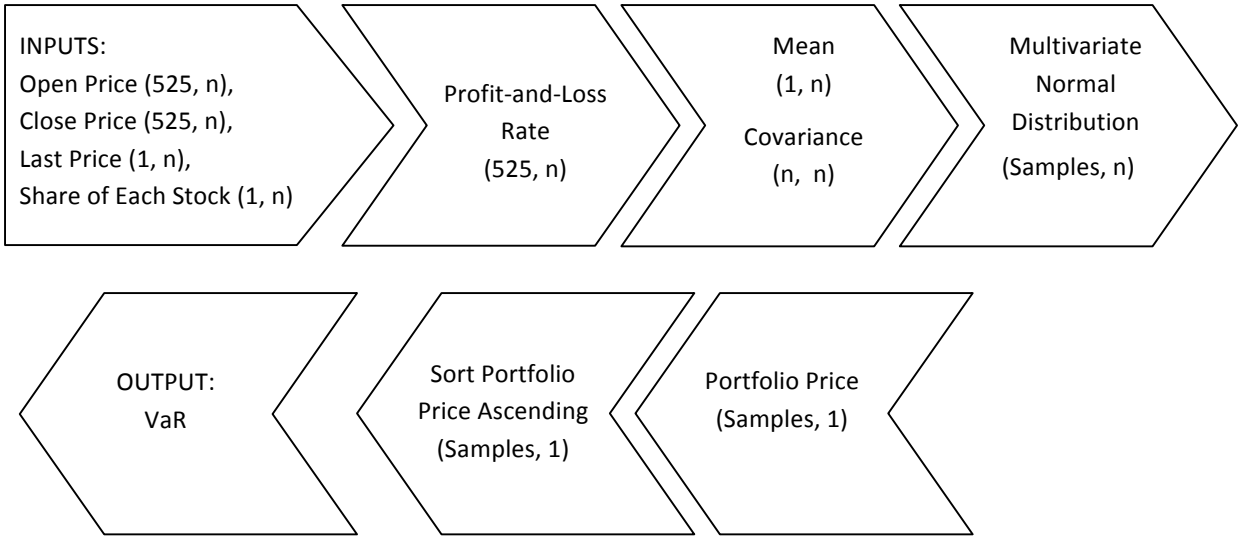


Figure 3: Workflow of VaR estimation

In the following part, we will present the detail implementation to use parallel computing to do Mente Carlo VaR estimation.

3.2.1 Calculate Profit-and-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 each business day. And then calculate the profit-and-loss rate ($plRate$) using Eq. (3.1).

$$plRate = (close - open) / open \tag{3.1}$$

The parallel calculation is implemented as following Figure , there are total $m \times n$ threads computing in parallel, each thread (i, j) do same operation: $plRate_{i,j} = (close_{i,j} - open_{i,j}) / open_{i,j}$ $i = 1, \mathrm{L}\ , n; j = 1, \mathrm{L}\ , m$. And we define block dimension $\dim Block(16,16)$ and grid dimension $\dim Grid(m/16, n/16)$ in CUDA, which means there are 256 threads in each block, and $(m/16) \times (n/16)$ blocks in the grid.

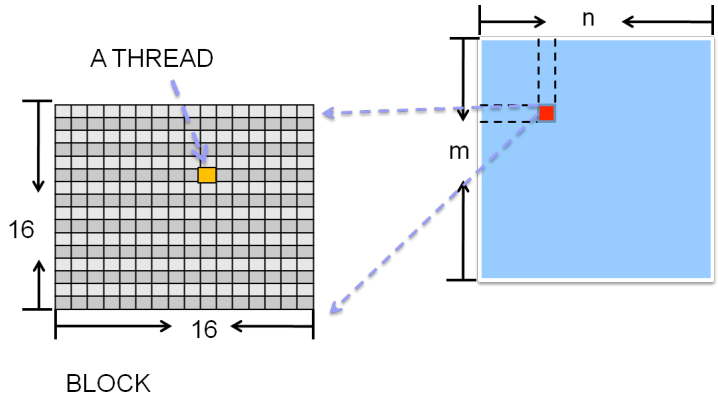


Figure 3: CUDA parallel programming to calculate profit-and-loss Rate

3.2.2 Multivariate Normal Distribution

After obtaining the profit-and-loss rate for m business day, we can find the distribution of each stock vector is normal distribution. So, we use multivariate normal distribution to generate ms random 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:

$$\varphi(x) = (\frac{1}{2\pi\sigma^2})^{p/2} |\Sigma|^{-1/2} \exp\{-\frac{1}{2}(X-\mu)'\Sigma^{-1}(X-\mu)\} \tag{3.2}$$

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:

First, 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.

Second, let $Z = (Z_1, \mathrm{L}\ Z_n)$ be a vector whose components are n independent standard normal distribution random.

Finally, let $mvd = u + A * Z$, where mvd is the multivariate normal distribution result.

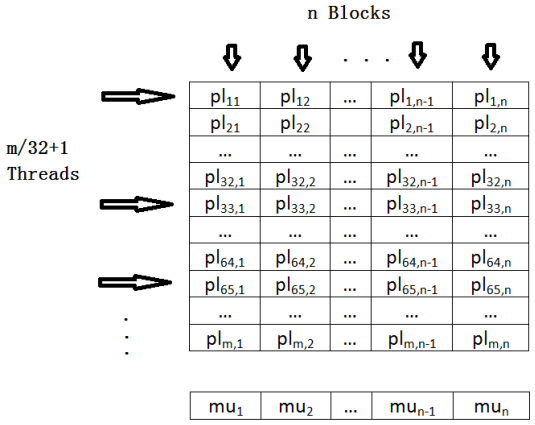
According to this theory, we generate a multivariate normal distribution matrix, which has ms rows and n columns, can be achieved in the following steps.

(1) Mean (u)

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

According to Eq. (3.3), the easiest way is to set n threads. But it wastes lots of resource, which not sufficiently use all the blocks and threads. So, in CUDA programming we set n blocks, and every block has $m/32+1$ threads, each threads calculate $plRate_{i,j} + plRate_{i+1,j} + \mathrm{L} + plRate_{i+31,j}$. (Figure 4)

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} X_{ij} \qquad i = 1, 2, \mathrm{L}\ n \tag{3.3}$$



*note: pl means $plRate$, mu means u

Figure 4: Parallel algorithm to calculate mean

(2) Covariance(Σ)

Covariance is a measure of how much two variables change together as defined in Eq.(3.4).

$$\Sigma = \mathrm{cov}(X_i, X_j) = \frac{1}{n}\sum_{i=1}^{n}(X_i - u_i)(X_j - u_j) \qquad i, j = 1,2,\mathrm{L}\ \ n \tag{3.4}$$

Using the same idea as mean to do parallel computing of covariance, in CUDA programming there are total $n \times n$ blocks, and every block has $m/32+1$ threads, each threads calculate $(pl_{t,i} - \mu_i)(pl_{t,j} - \mu_j) + (pl_{t+1,i} - \mu_i)(pl_{t+1,j} - \mu_j) + \mathrm{L}\ + (pl_{t+31,i} - \mu_i)(pl_{t+31,j} - \mu_j)$. (Figure 5)

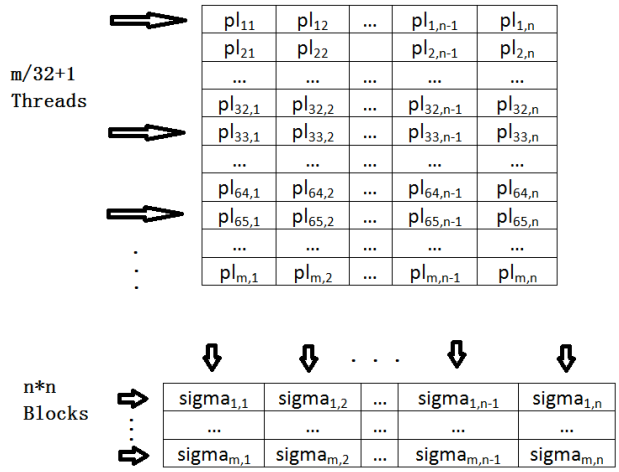


Figure 5: Parallel computing to calculate covariance

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

We want to get get a matrix A, where $AA^T = \Sigma$. Cholesky decomposition is good choice but the numbers can become negative numbers, in which case the square-root algorithm cannot continue. So in this situation, we use singular value decomposition.

We consider the standard eigenvalue problem

$$Bx = \lambda x \tag{3.5}$$

where B is a real $n*n$-dense symmetric matrix, which is covariance Σ in this project. Use Eq. (3.6) to determine all the Eigen pairs. We recall that Jacobi's sequential method reduces the matrix B to the diagonal form by an infinite sequence of plane rotations

$$B_{k+1} = V_k B_k V_k^T, \qquad k = 1,2,\mathrm{L} \tag{3.6}$$

$$V = \begin{bmatrix} 1 & & & & & & & & 0 \\ & \ddots & & & & & & & \\ & & \cos\theta & 0 & \cdots & 0 & \sin\theta & & \\ & & 0 & 1 & \cdots & 0 & 0 & & \\ & & \vdots & \vdots & \ddots & \vdots & \vdots & & \\ & & 0 & 0 & \cdots & 1 & 0 & & \\ & & -\sin\theta & 0 & \cdots & 0 & \cos\theta & & \\ & & & & & & & \ddots & \\ 0 & & & & & & & & 1 \end{bmatrix} \tag{3.7}$$

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 $\tan 2\theta_{ij}^k = \dfrac{2b_{ij}^k}{b_{ii}^k - b_{jj}^k}$, where $\left|\theta_{ij}^k\right| \le \dfrac{1}{4}\pi$.

For numerical stability, we determine the plane rotation by $c_k = \dfrac{1}{\sqrt{1 + t_k^{\,2}}}$ and $s_k = c_k t_k$, where $a_k = \cot 2\theta_{ij}^k$ and $t_k = \dfrac{\operatorname{sign} a_k}{|a_k| + \sqrt{1 + a_k^{\,2}}}$. 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, $\lambda \approx V_n \mathrm{L}\ V_k \mathrm{L}\ V_1 B$, $V \approx V_n \mathrm{L}\ V_k \mathrm{L}\ V_1 E$, 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) 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

$$x_{k+1} = a * x_k + c \mod \mathrm{m} \tag{3.8}$$

We use parameters: $m = 2^{31}$, $a = 65539$ and $c = 0$, this method named RANDU. [22]

In the 1960s, the Scientific Subroutine Package (SSP) on IBM mainframe computers included a random number generator named RANDU. There are lots of different choices of parameters. 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 . There are $ms/1000+1$ blocks and every block has n threads, and each thread generate random number separately.

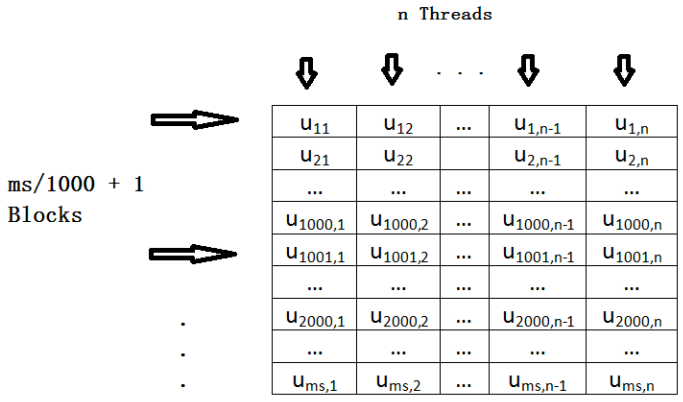


Figure 6: Parallel computing of RANDU method to generate uniform distribution random numbers

(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

$$Z_0 = R * \cos\theta = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{3.9}$$

and

$$Z_1 = R * \sin\theta = \sqrt{-2\ln U_1}\sin(2\pi U_2) \tag{3.10}$$

Then Z_0 and Z_1 are independent random variables with standard normal distribution. The parallel computing algorithm shows in Figure 7. 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.

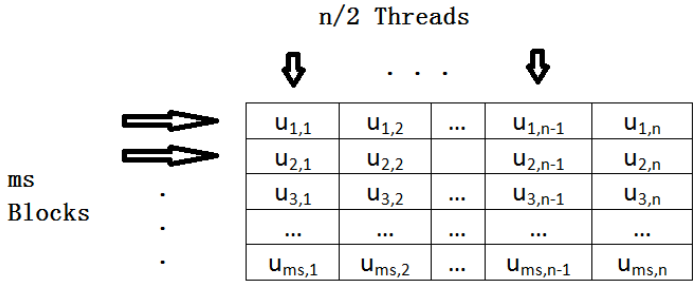


Figure 7: Parallel computing of Box–Muller method to generate standard normal distribution random numbers

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

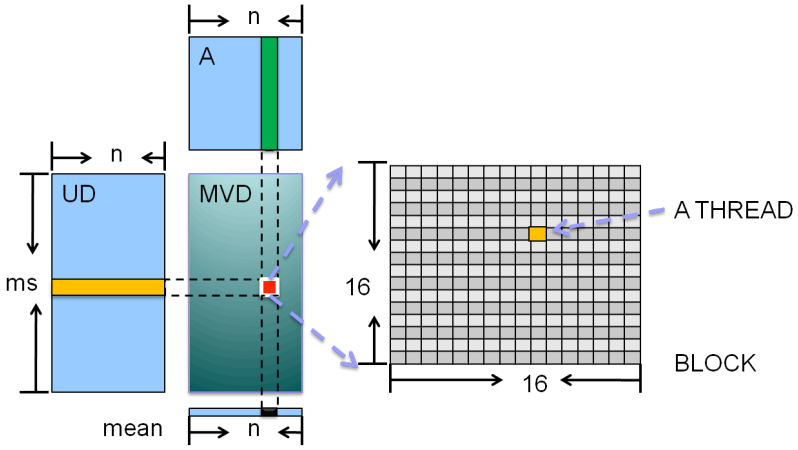


Figure 8: Parallel computing of matrixes multiplication and addition

In this project, $mvd = u + Z \times A$, the value of mvd is $r^{t+1}_{ms,n}$, which we mentioned in the algorithm, Z is $ms \times n$, A is $n \times n$, μ is $1 \times n$. This is a matrix multiplication and addition operation. As illustrated in Figure 8, there are total $ms \times n$ threads computing in parallel, each thread (i, j) do same operation:

$$mvd(i,j) = Z(i,1) \times A(1,j) + \mathrm{L} \ + Z(i,k) \times A(k,j) + \mathrm{L} \ + Z(i,n) \times A(n,j)$$

$$i = 1, \mathrm{L} \ , ms; \ j = 1, \mathrm{L} \ , n; k = 1, \mathrm{L} \ , n.$$

And we define block dimension $\dim Block(16,16)$ and grid dimension $\dim Grid(m/16, n/16)$ in CUDA, which means there are 256 threads in each block, and $(m/16) \times (n/16)$ blocks in the grid.

3.2.3 Calculate Portfolio Value

$$portofolio_i = \sum_{j=1}^{n} (1 + mvd_{i,j}) \times lastprice_j \times shares_j \qquad i = 1, \mathrm{L} \ , ms; \quad j = 1, \mathrm{L} \ , n \qquad 3.11$$

$portofolio$ 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 9).

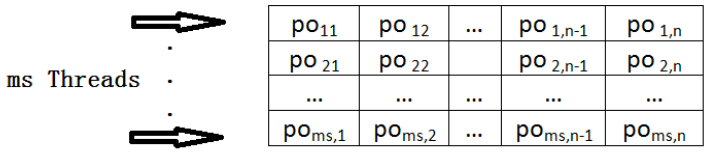


Figure 9: Parallel computing to calculate portfolio price

3.2.4 Merge sort

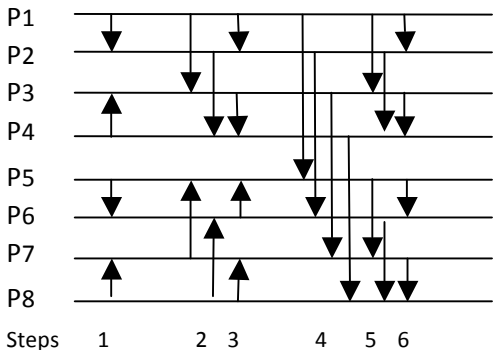
In computer science, merge sort is a sorting algorithm for rearranging lists into a specified order. It can be seen as a good example of the divide and conquer algorithmic paradigm.

Conceptually, merge sort works as follows steps. First, divide the unsorted list into two sublists of about half the size and sort each of the two sublists. Second, merge the two-sorted sublists back into one sorted list.

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 ($O(1+2+\mathrm{L}\ +\lg(n))=O(\lg(n)^2)$). For example, when $n=8$, the parallel loop steps are 6 (Figure 1). 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. When $n=8$, the number of parallel threads is 4, and when $n=64$, the number of parallel threads is 32.

In parallel programming in CUDA, there are total $ms/2$ threads running at the same time, each thread do comparison.



(P1…P8 – eight different elements in random order;
Down arrow – if lower position value smaller than higher position value, swap two value;
Up arrow - if lower position value larger than higher position value, swap two values)
Figure 1: Parallel algorithm of merge sort with 8 random numbers to ascending order

4. 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. GPUs are NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+, which have 128 parallel processor cores, so they 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.

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.

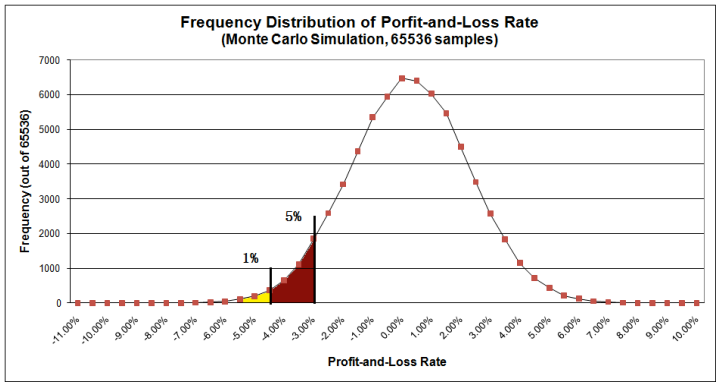
Since loading data using Matlab is very slow when the data is very large, so we saved all the data in EXCEL, and the program read from EXCEL when computing.

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

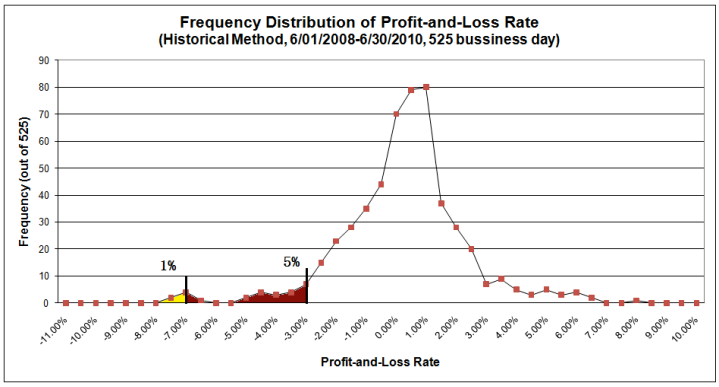
assume the confidence level is 95%. So the output is the maximum loss in the 95% confidence level on current day 10/30/2010.

Result and Discussion

First we discuss the result of Monte Carlo Simulation and Historical Method. Figure 2 (a) is the frequency of profit-and-loss rate of a portfolio over a day using Monte Carlo simulation. In the 99% confidence level, the loss rate is 4.5%, and in the 95% confidence level, the loss rate is between 3.0% and 3.5%. Figure 2 (b) is the frequency of profit-and-loss rate of a portfolio over a day using historical method. 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%. We can find that in the 99% confidence level, the results of two method is different, and in the 95% confidence level, the results of two method is similar.



(a) Frequency distribution of profit-and-loss rate using Monte Carlo Simulation



(b) Frequency distribution of profit-and-loss rate using Historical Method

Figure 2: Frequency Distribution of profit-and-loss rate using different method

Then we discuss about the running time of Monte Carlo Simulation in GPU using CUDA programming and in CPU using Matlab and C code. We use Monte Carlo simulation method to estimate VaR of a day at 95% confidence level. Figure 12 and Figure 13 are the running time of Monte Carlo Simulation using CUDA C in GPUs (NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+), and Matlab and C in CPU with a portfolio in various assets (16 stocks, 48 stocks, 96 stock and 192 stocks) and different samples (1024, 4096, 8192, 16384, 32768 and 65536) Monte Carlo method generated. The result shows that:

(1) The performance of NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+

When assets number in a portfolio is 16, the running time of Geforce 9800 GTX/ 9800 GTX+ and the running time of Quadro FX 3700 are close. But when the the assets number is 192 and simulation samples are very large (32768 samples or 65536 samples), the performance of Geforce 9800 GTX/ 9800 GTX+ is about 2 times faster than the performance of Quadro FX 3700.

(2) The performance of CUDA C program in NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+, and Matlab and C program in CPU

When we program C codes in CUDA, we have to copy data from host to device or from CPU to GPU before computing in GPU, and after computing finish, we have to copy data from device to host or from GPU to CPU. This memory copy time can't be ignored. And in Figure 14, we can see that with different asset number and sample size, the memory copy time is similar. That's why in Figure 12 and Figure 13, when asset size is 16, both GPUs are slower than Matlab. But when data size both asset size and Monte Carlo simulation sample size increasing, we can find the influence of memory copy time is smaller. The running time of Matlab and C codes in CPU grow exponentially. When asset size is 196 and the simulation samples are 65536, Geforce 9800 GTX/ 9800 GTX+ is 11 times faster than Matlab and 110 times faster than C program in CPU. The result shows that the when matrix size increases, GPU computing running much faster, which means GPU has big advantage in massive and large scale computing.

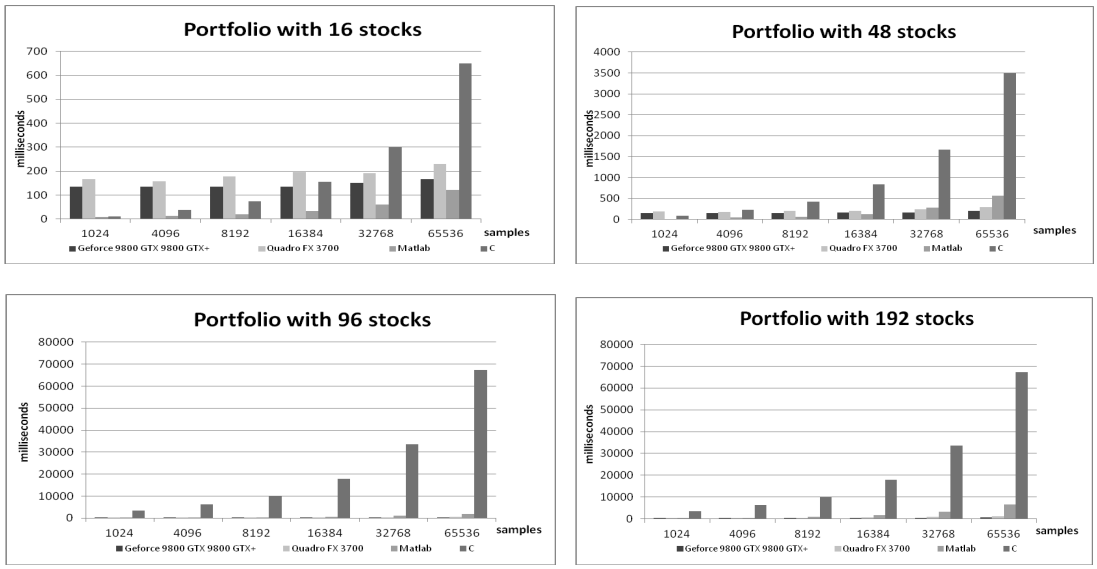


Figure 12: Running Time of Monte Carlo Simulation using CUDA C in GPUs (NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+), and Matlab and C in CPU

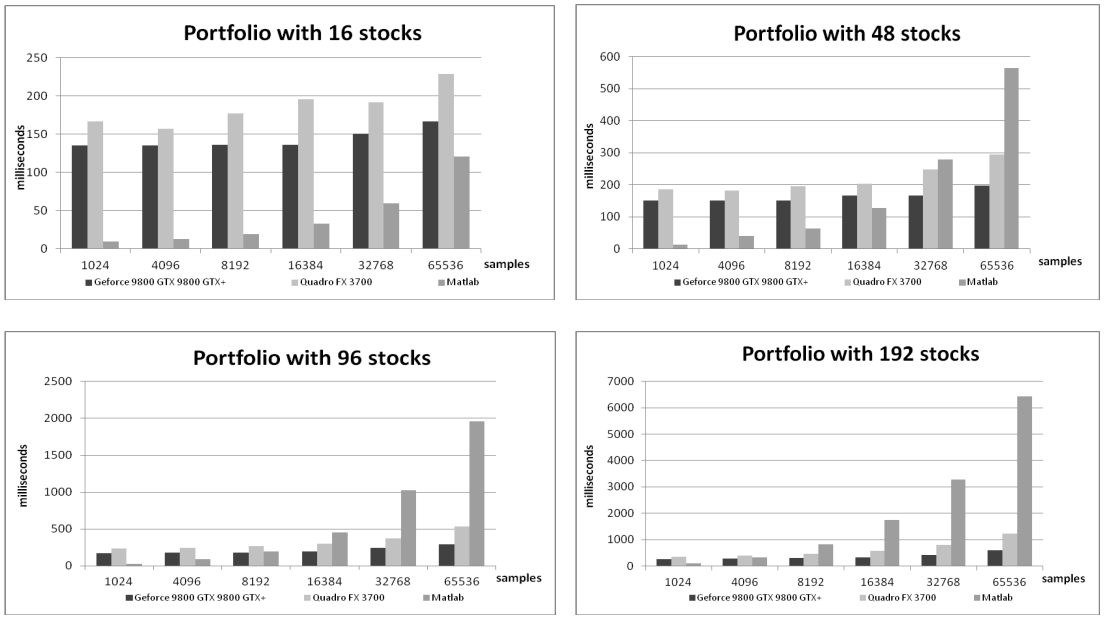


Figure 13: Running Time of Monte Carlo Simulation using CUDA C in GPUs (NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+), and Matlab in CPU

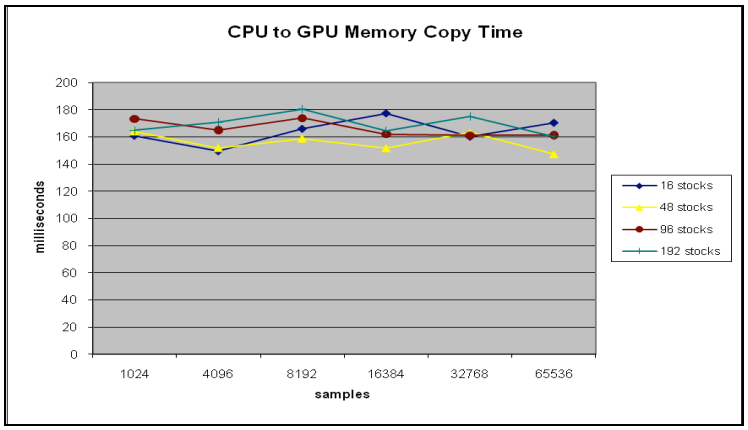


Figure 14: Runing Time of Memory Copy from CPU to GPU for NVIDIA Quadro FX 3700

5. Conclusion

In this paper, we have implemented the Monte Carlo Simulation based VaR estimation using CUDA C program run in GPUs (NVIDIA Quadro FX 3700 and Geforce 9800 GTX/ 9800 GTX+), and C and Matlab run in CPU. This paper describes the detailed computing algorithm by leveraging the parallel computation capability of CUDA. We run different experiments to compare the performance in CUDA C in different GPUs, C and Matlab in CPU in different conditions: one portfolio having 16, 48, 96 and 192 assets and Monte Carlo simulation generating 1024, 4096, 8192, 16384, 32768 and 65536 samples. The results show that NVIDIA Geforce 9800 GTX/ 9800 GTX+ is 2 times faster than Quadro FX 3700, and using CUDA C program in GPU can greatly improve the performance of Monte Carlo Simulation. When the data size larger, GPU computing has more advantage. So, CUDA or GPU computing is fit to large scale data computing not small data set. And GPU parallel computing greatly improve the performance of Monte Carlo simulation application in VaR estimation.

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)

[24] Jorion, Philippe (2006). Value at Risk: The New Benchmark for Managing Financial Risk (3rd ed.). McGraw-Hill. ISBN 978-0071464956.