2011

Parallel computing with improved techniques for Monte Carlo simulation in VaR

Ping Hung Wu
CUNY City College

Parallel computing with improved techniques for Monte Carlo simulation in VaR

Thesis

Submitted in partial fulfillment of the requirement for the degree

Master of Engineering (Computer Science)

at

The City College of New York

of the

City University of New York

by

Ping Hung Wu

August 2011

Approved:

Professor Izidor Gertner, Thesis Advisor

Professor Douglas Troeger, Chairman

Department of Computer Science

ACKNOWLEDGEMENTS

Thank you to all of my professors who helped to prepare me in my studies during my master's degree in the City College of New York. I would also like to give thanks to those people who supported me during my studies for the master's degree of computer science and to finish this thesis: my parents, my family and my classmates in CCNY. I would like to especially thank Professor Izidor Gertner who introduced me to the concepts and actual implementation in parallel computing using CUDA. With your generous help in providing knowledge and granting the use of the machines in the lab, I have been able to experience the power of parallel computing. A special thanks to Professor Stephen Lucci who checked and corrected my English. Thank you, once again to all of you.

ABSTRACT

Value at Risk (VaR) is a widely used tool for the assessment of one's investments. VaR is used to evaluate the risk of loss on a financial portfolio. This metric can be computed in several ways. In the historical approach, past trends of the appropriate combination of stocks is used to estimate current portfolio fluctuations. The variance – covariance method, meanwhile, seeks to discover relationships in price fluctuations for one's stocks. Finally, Monte Carlo simulation capitalizes upon the stochastic nature of stock prices to predict future value. This latter approach, however, relies heavily on the multiplication of vectors with matrices and is therefore time consuming.

An investor always has a predetermined limit in mind of how much they can afford to lose – this quantity is called collateral. This limit is often agreed upon via consultation with a risk manager. The VaR decision problem seeks to answer this question: Does VaR exceed an investor's collateral?

The loss matrix of a portfolio contains valuable information that may be gleaned via the use of norms. This thesis focuses on computing upper bounds for VaR which rely upon information obtained from these norms; we thereby can discern when it is possible to answer the VaR decision problem without resorting to time-consuming Monte Carlo methods.

We use a parallel computing architecture – CUDA in this thesis. Several CUDA kernels are implemented and executed on a Nvidia graphics card thereby increasing computational performance.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Statement of the problem

In risk management we want to answer the following question: Do our losses exceed what we can afford during the life of an investment. More precisely, we want to know how much we can lose during a specific period of time, and the probability of such a loss. This leads to the problem of estimating Value at Risk (VaR).

Computing VaR using Monte Carlo simulation is a time consuming job. The matrix-vector multiplication processes in order to simulate profit for a portfolio always take time complexity O(NM). It will save time if we can estimate the magnitude of VaR and check if it exceeds collateral without using Monte Carlo simulation. This is called “The VaR decision problem”:

Given the loss matrix L, the collateral value V_c, and the weight vector ω, is $V_R > V_c$, where V_R is the p = P * N^{th} largest component of L * ω?

This thesis will focus on solving the VaR decision problem. Several algorithms are used to compute the upper bound of VaR based on L_1, L_2 and L_∞, the norm of a matrix and reduce the chance of VaR estimation requiring Monte Carlo simulation. We will apply the parallel computation architecture - CUDA in this thesis. CUDA kernels are implemented in the program for parallel computation to increase the performance of the code, compared to conventional C/ C++ code.

1.2 Requirements and specifications

In this thesis we focus on simulating a large number of VaR decision problems that need to be solved for different portfolios based on Monte Carlo simulation. The number of portfolios is set to 10,000 in our simulation; there are M elements in each of the portfolio vector ω. Each asset in a portfolio vector is sampled N = 10,000 times based on a normal distribution model to generate its loss and gain; a loss matrix L whose size is $N \times M$ is thereby generated.

The $N \times M$ loss matrix L is fixed during the day so that the information provided by the loss matrix L can be used for every portfolio vector ω. The conventional VaR computation procedure computes b = L * ω; a positive value indicates a loss, whereas a negative value indicates a profit. We then sort the vector b. The value at risk is determined as the p = P * N largest component in b, where P stands for our level of confidence. For a 99% confidence level, P is set in the program to be 0.01. This is done using CUDA in the C programming language. The graphic card used in this simulation is the NIVDIA Quadro FX 3700, with an Intel Xeon(R) CPU E5410 running at 2.33 GHz.

1.3 Roadmap

The flow of our program to solve the VaR decision problem is given in figure 1:

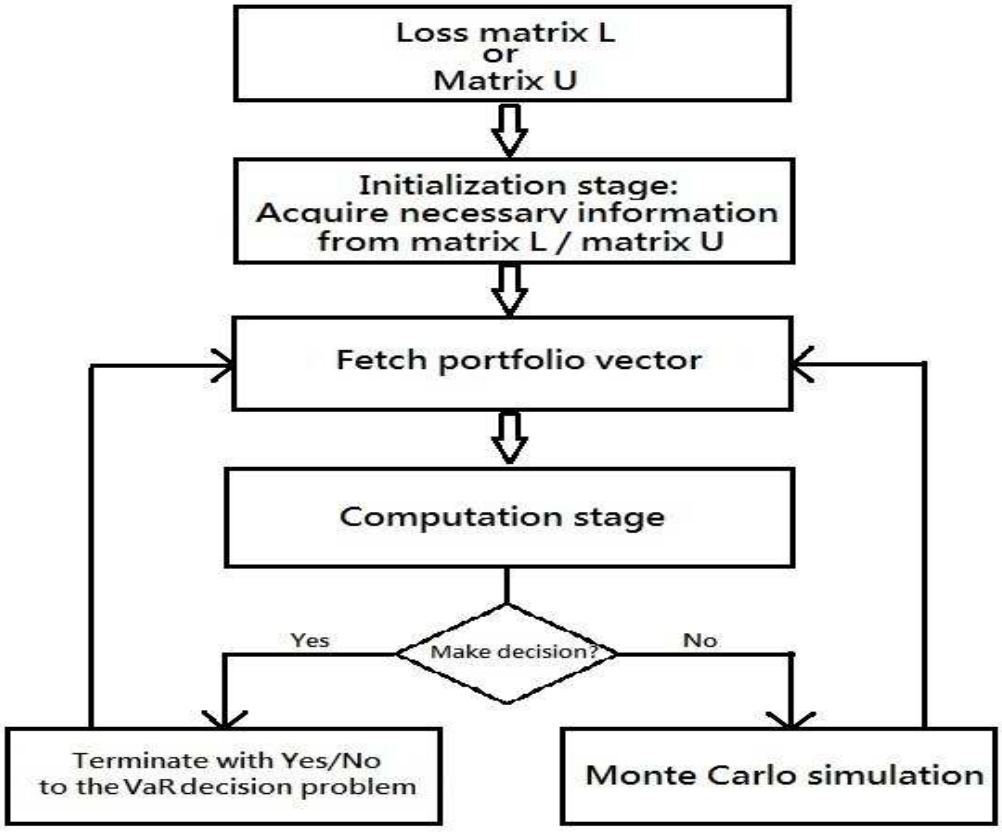

Figure 1: Road map

Loss matrix L, or its U matrix computed by singular value decomposition, is used to compute the information needed during the initialization stage. In the computation stage we start to fetch portfolio data. Several algorithms are used to compute an upper bound for VaR based on different norms for the portfolio. Some heuristic methods are also used to reduce the computation during the computation stage. If the VaR decision problem is solved with a yes or no answer, then the VaR decision problem for this portfolio vector

terminates with that answer. Otherwise Monte Carlo simulation must be used to compute the value at risk for this portfolio.

2. The problem of Value at risk (VaR)

Value at risk is defined as the threshold value such that the loss for a portfolio over a time period exceeds this value at the given probability level. Three main factors in the problem of value at risk are: A fixed period of time to the investment, the level of confidence for the loss (usually 95% or 99%) and a specified level of loss in value.

There are three ways to compute value at risk:

1) Historical method
2) The variance – covariance method
3) Monte Carlo simulation

2.1 Historical method

The historical method for estimating Value at Risk is completely based on the historical data of the portfolio. Value at Risk is computed by running the portfolio using past data to estimate its return. To run a historical simulation, we begin by gathering historical data for the portfolio. When computing Value at Risk, we simply re-compute the change for the portfolio in a specific time period based on its historical data. The Value at Risk is determined by actual price movement. There is no distribution assumption attached to the data.

2.2 The variance – covariance method

In the variance – covariance method, the potential value of Value at Risk is computed by some distribution model. To estimate the Value at Risk of a single asset using the variance – covariance method with a normal distribution model, for example, with a known mean value and standard deviation extracted from the historical data, there is a 95% confidence for an asset value drop between two standard deviations above and below the mean. Computing VaR for a portfolio of multiple assets is more complicated since we need to take the relationship between variables into consideration. A variance/covariance matrix needs to be computed based on the historical data to present the relationship between two factors; positive values indicates that the two factors move in the same direction, while negative values indicate that they move in opposite directions.

2.3 Monte Carlo simulation

In Monte Carlo simulation, a distribution model is used to simulate the change in market factors. Unlike the historical method, which completely depends on historical data, an adequate distribution model can be applied to reflect the movement for the market factor. Once the distribution model is set, thousands to tens of thousands of data items are generated randomly. This data is then used to compute lose and profit for the current portfolio. The Value at Risk is then determined based on the confidence level. Monte Carlo simulation is used in this thesis as the final step to compute Value at risk.

3. Parallel computing using CUDA

3.1 CUDA - Instructions

CUDA stands for **C**ompute **U**nified **D**evice **A**rchitecture. It is a parallel computing architecture developed by Nvidia. Programmers can access memory or design kernel code which runs on GPU by using a programming language such as C with a CUDA library. Several steps are needed to work with CUDA:

1. Design the CUDA kernel code: A function begins with **__global__** qualifier is declared as a kernel function running on a device (graphic care).
2. Linear memory space on a device must be allocated: CUDA function cudaMalloc( ) is used in order to allocate memory space on device to store data used by the kernel code.
3. Define threads and blocks: The number of threads in a block and the number of blocks for device memory are defined using function dim3( ).
4. Data transfers between host and device memory: Since the kernel code works only on device, we must copy the data needed to device memory. The result located in device memory must then be copied back to host memory for further use. These two steps are done by calling the CUDA function cudaMemcpy( ) function.
5. Invoke kernel function followed by **<<<…>>>** execution configuration syntax to specify the number of threads executed by the kernel.
6. Memory space declared by function cudaMalloc( ) can be freed by calling CUDA function cudaFree( ) .

3.2 CUDA kernel for matrix-matrix multiplication

In matrix multiplication we multiply two N x N matrices Md and Nd, the result is then stored in matrix Pd:

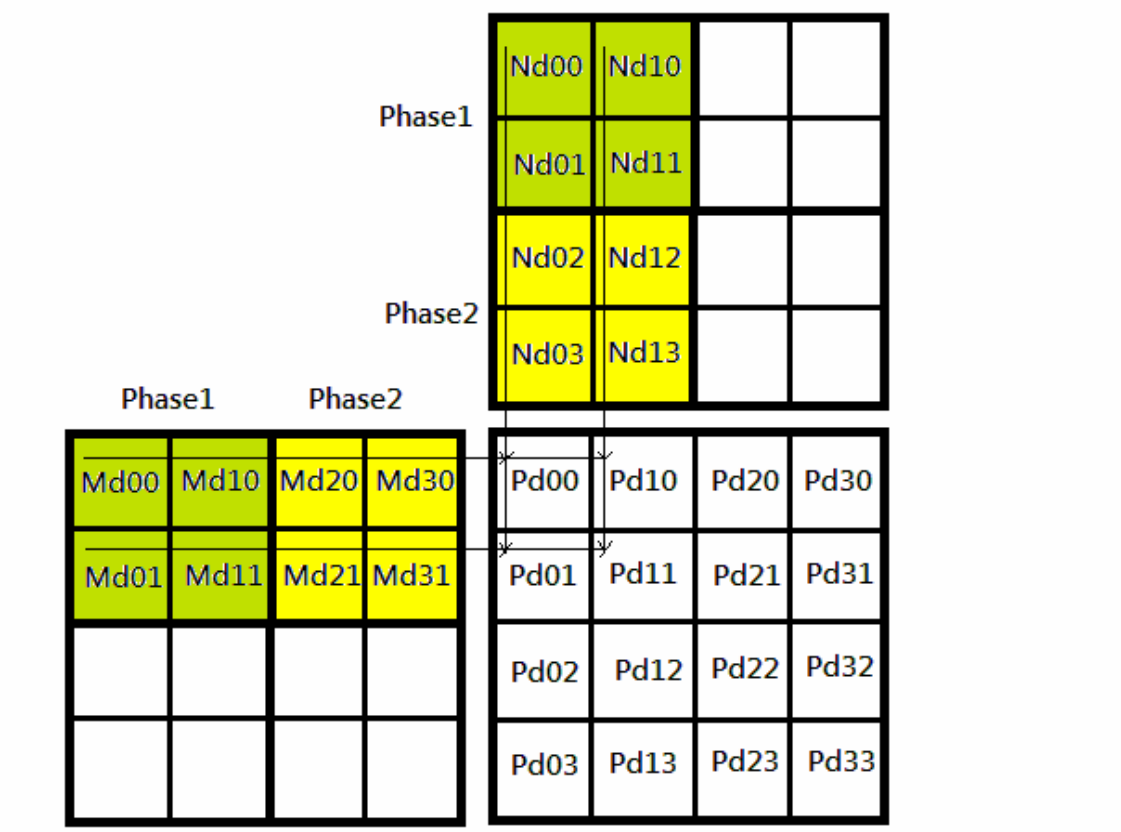

Figure 2: Matrix multiplication

Figure 2 demonstrates the computation involved in matrix multiplication. Each element in Pd is computed by the summation of the corresponding row elements in Md which multiplies the corresponding column elements in Nd as shown below:

Pd00 = Md00 * Nd00 + Md10 * Nd01 + Md20 * Nd02 + Md30 * Nd03
Pd10 = Md00 * Nd10 + Md10 * Nd11 + Md20 * Nd12 + Md30 * Nd13
Pd01 = Md01 * Nd00 + Md11 * Nd01 + Md21 * Nd02 + Md31 * Nd03

Pd11 = Md01 * Nd10 + Md11 * Nd11 + Md21 * Nd12 + Md31 * Nd13

N

In designing the CUDA kernel for matrix multiplication we use shared memory in GPU to store the data. Shared memory is expected to be faster than global memory and is allocated using the **__shared__** qualifier. The following code is the CUDA kernel used in the program. It is tested under settings of 256 threads per block (TILE_WIDTHx=16, TILE_WIDTHy=16) and 16 threads per block (TILE_WIDTHx=4, TILE_WIDTHy=4) with correct result.

```
1. //(CUDA) kernel for matrix-matrix multiplication
2. __global__ void MatrixMulKernelv3(float *md , float *nd, float *pd,
3. int m_row , int m_col)
4. {
5.   //shared memory for sub-matirx Mds
6.   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
7.   //shared memory for sub-matrix Nds
8.   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];
9.   //Block index
10.  int bx = blockIdx.x;
11.  int by = blockIdx.y;
12.  //Thread index
13.  int tx = threadIdx.x;
14.  int ty = threadIdx.y;
15.  //Identify the position for matrix(row/column) in global memory
16.  int Row = by * TILE_WIDTHy + ty;
17.  int Col = bx * TILE_WIDTHx + tx;
18.
19.  float pvalue =0;
20.  //divide into m_col/TILE_WIDTHx phases
21.  for(int i=0 ; i< m_col/TILE_WIDTHx ; i++)
22.  {
23.      //Load the data from global memory into shared memory
24.      Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
25.      Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
26.      //synchronize to make sure matrices are loaded
27.      __syncthreads();
28.
29.      for(int j=0 ; j<TILE_WIDTHx ; j++)
30.      {
31.            pvalue += Mds[ty][j] * Nds[j][tx];
32.            //sychronize to make sure the computation is done before
33.            //loading the data for next phase
34.            __syncthreads();
35.      }
36.      //write the computation result to global memory
```

```
37.       *(pd + Row * m_col + Col) = pvalue;
38.   }
39.
40. }
```

Table 1: CUDA kernel for matrix-matrix multiplication.

In line 2 we declare this fucntion as a CUDA kernel function using the **__global__** qualifier. It takes five arguments: Float pointer *md and *nd point to the space for target matrices that will do matrix-matrix multiplication. Float pointer *pd points to the space for the result matrix. The last two arguments specify the row and column number of the matirx.

Line 6 and line 8 declare two shared memory arrays in each thread. Two constants TILE_WIDTHy and TILE_WIDTHx are declared at the beginning of the code as row and column numbers in a block. For example, if we have TILE_WIDTHx=4 and TILE_WIDTH=4, there are 4 * 4 = 16 threads in a block. The limitation on the number of threads in a block is 512 for CUDA 3.0.

Lines 10 to 14 store the value of threadId and blockId. Lines 16 and 17 compute the corresoponding 2D matrix index used by matrix pd in global memory.

Lines 21 through 38 is a loop based on the size of the target matrix divided by the size of a block. This loop divides the algorithm into M/TILE_WIDTHx phases. In each phase we load the data needed to compute the value of each element in matrix Pd.

Lines 23 and 24 load the sub matrix needed from global memory to shared memory for each phase. The synchronization function **__syncthreads( )** is

used in line 27 to make sure all threads in a block reach this line and ensure that the data needed is loaded correctly before proceeding to multiplication.

Lines 29 through 35 perform the actual multiplication for the corresponding row and column elements in the shared memory matrices. The result is accumulated and stored in the variable pvalue. The final result that goes through all phases will be placed into the corresponding position in global memory using line 37.

3.3 CUDA kernel for matrix-vector multiplication

The CUDA code for matrix-vector multiplication is similar to the code for matrix-matrix multiplication discussed above; the only difference is that the second argument is a vector instead. The space for shared memory array Nds is adjusted to fit its size.

```
1. //(CUDA) core for matrix-vector multiplication
2. __global__ void MatrixVectorMulKernel(float *md , float *nd, float
3. *pd , int m_row, int m_col , clock_t* time)
4. {
5.    //Declare shared memory for use
6.    __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
7.    __shared__ float Nds[TILE_WIDTHx2];
8.
9.    //Acquire blockId and threadId for use
10.   //int bx = blockIdx.x;
11.   int by = blockIdx.y;
12.   int tx = threadIdx.x;
13.   int ty = threadIdx.y;
14.
15.   int Row = by * TILE_WIDTHy2 + ty;
16.   //int Col = bx * TILE_WIDTHx2 + tx;
17.
18.   float pvalue=0;
19.
20.   if(ty==0) time[by]=clock();
21.
22.   for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
23.   {
24.     //Load the data need from global memory to shared memory
```

```
25.     Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
26.     Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
27.     //synchronization
28.     __syncthreads();
29.     //compute the result in each thread
30.     for(int j=0 ; j < TILE_WIDTHx2 ; j++)
31.     {
32.           pvalue += Mds[ty][j] * Nds[j];
33.     }
34.
35.     *(pd + Row) = pvalue;
36.   }
37.
38.   if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2] = clock();
39.
40. }
```

Table 2: CUDA kernel for matrix-vector multiplication

3.4 CUDA kernel for computing column sum of the matrix.

The following CUDA kernel code computes the column sum for each column in the matrix.

```
1. //(CUDA)Core for computing the column sum of each culumn in the
2. //matrix
3. __global__ void CUDA_matrix_col_sum(float *a, float *b, int m_row ,
4. int m_col)
5. {
6.    //Declared shared memory array partialsum
7.    __shared__ float partialsum[TILE_WIDTHy][TILE_WIDTHx];
8.    //Acquire blockId and threadId
9.    int bx = blockIdx.x;
10.   int by = blockIdx.y;
11.   int tx = threadIdx.x;
12.   int ty = threadIdx.y;
13.   //Identify the position in global memory
14.   int Row = by * TILE_WIDTHy + ty;
15.   int Col = bx * TILE_WIDTHx + tx;
16.
17.   //float column_sum=0;
18.   //Load the data needed from global memory to shared memory
19.   partialsum[ty][tx] = *(a+Row*m_col+Col);
20.   __syncthreads(); //
21.   //Compute the column sum for each column
22.   for(int i=1 ; i < TILE_WIDTHy ; i *= 2)
23.   {
24.         __syncthreads();
25.         if( (ty % (2*i) ==0) )
26.         {
27.               //Make sure matrix index not exceed it's maximum!!!
28.               if(ty+i < TILE_WIDTHy)
```

```
29.                 {
30.                    partialsum[ty][tx] += partialsum[ty+i][tx];
31.                 }
32.                 else
33.                 {
34.                 }
35.                 __syncthreads();
36.           }
37.    }
38.    __syncthreads();
39.    //The result should be the first elememt for each column.
40.    //Copy the result to global memory
41.    *(b + by * m_col + Col) = partialsum[0][tx]; //
42. }
```

Table 3: CUDA kernel for computing column sum in the matrix

An underlying idea in this code is to use threads with an odd number threadId.y to gather the results in the array shown in Figure 3

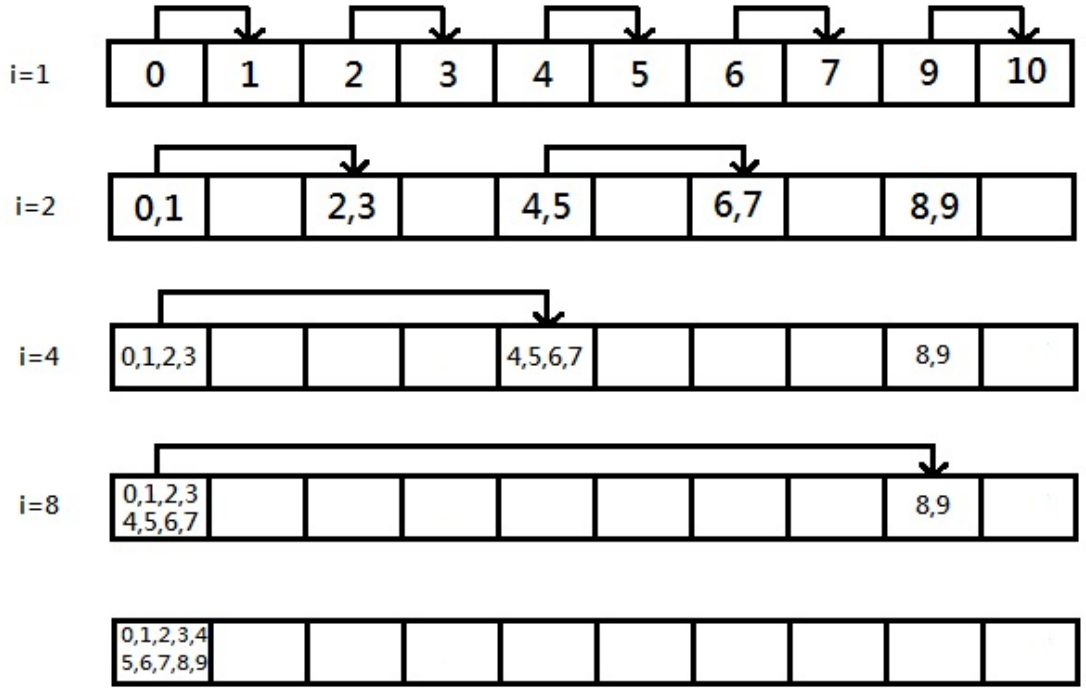

Figure 3: Underlying idea for computing column sum using CUDA

Lines 22 through 37 in the code compute the summation for each column in the matrix. Each iteration i increases by 2. In the first iteration only those threads with an odd threadyId.y number will compute the summation with their adjacent element. Those elements with threaded.y equals to 0, 2, 4…will add the adjacent element 1, 3, 5…etc. We then increase i by 2 so that the thread with threadId.y = 0, 4, 8…etc. will be added with the element with threadId.y = 2, 6, 10… This process will continue until variable i is greater than the dimension of the block. The final result will then be located in the element with threadId.y = 0 and will be placed into global memory for later use.

3.5 CUDA kernel for finding out minimum/Maximum value in the vector

The following CUDA kernel reads in a vector and returns the minimum and maximum value in each block. The kernel first loads the data for each block and then takes the first argument as its maximum and minimum value in lines 14 through 16. The comparison process for each block is done in lines 18 through 35. The corresponding minimum value for each block is placed in the first part of the vector and the maximum value in the latter part of the vector.

```
1. //(CUDA) code to find out the minimum/maximum value in ca
2. //this CUDA function works only when matrix elememt number is the
3. //power of 2!!
4. __global__ void CUDA_find_minmax(float *ca , float *ans_min_max)
5. {
6.      __shared__ float ma[TILE_WIDTHx];
7.
8.      int bx = blockIdx.x;
9.      int tx = threadIdx.x;
10.     int m_index = bx * TILE_WIDTHx + tx;
```

```
11.     float min_v;
12.     float max_v;
13.
14.     ma[tx] = *(ca + m_index);
15.     min_v = ma[0];
16.     max_v = ma[0];
17.
18.     for(int i=0; i<TILE_WIDTHx ; i++)
19.     {
20.        if(min_v > ma[i])
21.        {
22.            min_v = ma[i];
23.        }
24.        else
25.        {
26.        }
27.        if(max_v < ma[i])
28.        {
29.            max_v = ma[i];
30.        }
31.        else
32.        {
33.        }
34.        __syncthreads();
35.     }
36.
37.     *(ans_min_max + bx ) = min_v;
38.     *(ans_min_max + DRAW/TILE_WIDTHx + bx) = max_v;
39. }
```

Table 4: CUDA kernel for finding out minimum/Maximum value in the vector

3.6 CUDA kernel for computing the row L_2 norm of a matrix

The following CUDA kernel computes the L_2 norm for each row in matrix ca. The data needed for each phase is loaded into shared memory in line 15. Lines 17 through 20 compute the summation of the square of each element in a row, the result is then stored in variable m_total. We next compute the square root of m_total and the result will be the L_2 norm for this row of the matrix. This value is placed into global memory in line 24.

```
1. __global__ void CUDA_matrix_row_l2norm( float *ca, float *cb , int
m_row , int m_col)
2. {
3.     //Acquire blockId and threadId
4.     int tx = threadIdx.x;
5.     int ty = threadIdx.y;
6.     //int bx = blockIdx.x;
7.     int by = blockIdx.y;
8.     int ROW = by * blockDim.y + ty;  //use blockDim.y
9.     float m_total=0;
10.    //Declare shared memory
11.    __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
12.    //Compute the summation of square of each element in a row
13.    for(int i=0; i< (m_col/TILE_WIDTHx); i++)
14.    {
15.        Mds[ty][tx] = *(ca + ROW*m_col + (i*TILE_WIDTHx)+tx);
16.
17.        for(int j=0 ; j<TILE_WIDTHx ; j++)
18.        {
19.             m_total += Mds[ty][j] * Mds[ty][j];
20.        }
21.
22.    }
23.
24.    *(cb + ROW) = sqrt(m_total);
25. }
```

Table 5: CUDA kernel for computing row L_2 norm of the matrix

4. Improved Algorithm for Monte Carlo simulation

The algorithms below try to answer the VaR decision problem before using Monte Carlo simulation, since the matrix-vector multiplication process dominates the computational cost of the problem. These algorithms compute the bounds of VaR based on the different matrix norms. Others answer the question by some heuristic process to reduce the computation of matrix-vector multiplications.

4.1 Algorithm 1: Conventional technique for determining if the VaR exceeds collateral

Input: N by M loss matrix **L**, M by 1 portfolio vector ω and **Vc** > 0.
Output: Answer to the decision problem: **Yes** or **No**
Procedure:
1. Compute b = L * ω
2. Determine V_R , the p^{th} largest component of b in O(N) time using the linear time algorithm.
3. If $V_R > V_c$ then the answer is **Yes**. Otherwise the answer is **No**.

Algorithm 1 is the conventional way to compute the value at risk. Inputs for this algorithm are the N by M loss matrix L and the M by 1 portfolio vector ω in step 1. Matrix-vector multiplication is used to compute b, the profit of this portfolio. We then sort by using quicksort. The value at risk is determined as the p^{th} largest component in b in step 2. The answer to the VaR decision problem is determined based on the value at risk computed above.

4.2 Algorithm 2: Computation of $\sum b_i$ in O(M) time.

Input: N by M loss matrix **L** and M by 1 portfolio vector ω

Output: S = $\sum_{i=1}^{N} b_i$ where b = L * ω

Procedure:

1. Note: $\sum b_i = \sum_i \sum_j L_{ij}\, \omega_j = \sum_j \omega_j \sum_i L_{ij}$
2. Initialization: We pre-compute $S_j = \sum_i L_{ij}$ (i.e. the summation for each column of L) for each 1 ≤ j ≤ M once and store these values.
3. Computation Given a vector ω, we compute S = $\sum_{j=1}^{M} \omega_j S_j$ in O(M) time.

This process computes the summation of b, labeled as S, when portfolio vector ω is given. The summation for each column of L can be pre-computed since the loss matrix L is fixed during the day. S is the summation of portfolio element ω_i multiplies the corresponding column sum S_i.

4.3 Algorithm 3: Computation of the L_2 norm of b, or its square, in O(M^2) time.

Input: Loss matrix **L** and portfolio vector ω.

Output: W = $b^T * b = \sum_{i=1}^{N} b_i^2$, where b = L * ω

Procedure:

- Initialization:
  1. Compute the singular value decomposition of L = UDV
  2. Pre-compute DV, and store the M non-zero rows in the M x M matrix D_v
- Computation:
  1. Compute the matrix-vector product $\omega' = D_v * \omega$ in O(M^2) time.
  2. Compute W = $\sum_{i=1}^{M} \omega_i'^2$, the square of the L_2 norm, in O(M) time.
- The L_2 norm is computed as $\sqrt{W}$, but we really need only W in our bounds.
- W is also the square of the L_2 norm of b = L * ω, since U is orthogonal.

This process computes the L_2 norm of b. Instead of computing b using matrix-vector multiplication L * ω, singular value decomposition is used to decompose the N by M loss matrix L into three matrices U, D and V. Matrix D contains M nonnegative singular values on its diagonal, the rest of the rows contain zeros. Matrix V will be an M by M matrix. In the initialization stage, we compute matrix multiplication D * V, only M rows of D are needed for computation. The result will be an M by M matrix D_v.

In the computation stage, $\omega' = D_v * \omega$ is computed when portfolio vector ω is given. This takes O(M^2) time. W can be computed by first taking the square of all elements in ω' and then computing its sum in O(M) time.

4.4 Algorithm 4: Computation of an upper bound on the p^{th} largest component of L * ω using the L_2 norm.

Input: N by M loss matrix **L** and M by 1 portfolio vector ω.

Output: An upper bound $\hat{V}_R$ on the p^{th} largest component of L * ω.

1. Compute S using Algorithm 2.
2. Compute W using Algorithm 3.
3. Compute $\hat{V}_R$ = (1/N) * (S + $\sqrt{\frac{(N-p)}{p} * (N*W - S^2)}$) in constant time.

Note: If $\hat{V}_R \leq V_c$, than the answer to the decision problem is No. Otherwise, the answer is not determined by this bound. This procedure takes O(M^2) time. If it fails to give an answer, we fall back upon the conventional procedure of Algorithm 1.

In Algorithm 4 we need to first compute S, the summation of *b*, and W, the square of *b*'s L_2 norm, using Algorithm 2 and Algorithm 3. The p^{th} largest component of *b*, labeled as V_R, is bounded by the following function:

$$V_R \leq (1/N) * \left(S + \sqrt{\frac{(N-p)}{p} * (N * W - S^2)} \right)$$

Where $S = \sum_{i=1}^{N} b_i$ and $W = b^T * b = \sum_{i=1}^{N} b_i^2$

4.5 Algorithm 5: Computation of an upper bound on the p^{th} largest component of L * ω using the L_1 norm.

Input: Loss matrix **L** and portfolio vector **ω**.

Output: An upper bound $\hat{V}_R$ on the p^{th} largest component of L * ω.

Procedure:

- Initialization: These steps are only carried out once.
  1. Compute L_{lp}^j = sum of the p largest components of column j of L, for each j, 1 ≤ j ≤ M.
  2. Compute L_{sp}^j = sum of the N – p smallest components of column j of L, for each j, 1 ≤ j ≤ M.
  3. Compute L_{lm}^j = sum of the p smallest components of column j of L, for each j, 1 ≤ j ≤ M.
  4. Compute L_{sm}^j = sum of the N – p largest components of column j of L, for each j, 1 ≤ j ≤ M.
- The following steps are carried out for the portfolio vector ω that is given.
  1. Compute $S = \sum_i b_i$ in O(M) using Algorithm 2.
  2. Compute $\tilde{\omega}$ in O(M) time.
  3. Compute $\hat{\omega} = \tilde{\omega}$ - ω in O(M) time.
  4. Compute $\hat{V}_R = 1/(2 * p) * \left[S + \sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i) - \sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i) \right]$ in O(M) time.

5. If $\hat{V}_R \leq V_c$, then the answer to the decision problem is No. Otherwise, the answer is not determined by this bound. This procedure takes O(M) time. If it fails to give an answer, then we try the conventional technique.

In Algorithm 5, four variables L_{lp}^j, L_{sp}^j, L_{lm}^j and L_{sm}^j are needed to estimate the upper bound of V_R. This is done in the program by first reading in a column of L into a vector and then sorting using quicksort. The corresponding L_{lp}^j, L_{sp}^j, L_{lm}^j and L_{sm}^j are then computed as defined above for each column of L.

The definition of $\tilde{\omega}$ and $\hat{\omega}$ used in steps 3 and 4 are defined as:

$\tilde{\omega}_i = \omega_i, \ \omega_i \geq 0; \ \tilde{\omega}_i = 0$, otherwise

$\hat{\omega} = \tilde{\omega} - \omega$

The upper bound of VaR is estimated by the function:

$$V_R \leq \hat{V}_R = 1/(2 * p) * \left[S + \sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i) - \sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i) \right]$$

Where $S = \sum_{i=1}^{N} b_i$

4.6 Algorithm 6: Computation of an upper bound on the p^{th} largest component of L * ω using the L_∞ norm.

Input: Loss matrix **L** and portfolio vector **ω**.

Output: An upper bound $\hat{V}_R$ on the p^{th} largest component of L * ω.

Procedure:

- Initialize: The following steps are carried out only once.
    1. Compute $L_p^j = \max_{i=1}^{N} L_{ij}$, for each j, $1 \leq j \leq M$.

2. Compute $L_m^j = \min_{i=1}^{N} L_{ij}$, for each j, 1 ≤ j ≤ M.

- Computation: The following steps are carried out for the portfolio vector ω that is given.
  1. Compute $\tilde{\omega}$.
  2. Compute $\hat{\omega} = \tilde{\omega} - \omega$ in O(M) time.
  3. Compute $\hat{V}_R = \sum_{i=1}^{M} L_p^i * \tilde{\omega}_i - \sum_{i=1}^{M} L_m^i * \hat{\omega}_i$ in O(M) time.
  4. If $\hat{V}_R \leq V_c$, then the answer to the decision problem is No. Otherwise, the answer is not determined by this bound. This procedure takes O(M) time. If it fails to give an answer, then we fall back upon the conventional technique.

In Algorithm 6 the upper bound of VaR is estimated using L_∞ norm. The maximum value L_p^j and the minimum value L_m^j for each column are found using linear search.

The VaR upper bound $\hat{V}_R$ is estimated by the function:

$$\hat{V}_R = \sum_{i=1}^{M} L_p^i * \tilde{\omega}_i - \sum_{i=1}^{M} L_m^i * \hat{\omega}_i$$

where $\tilde{\omega}_i$ and $\hat{\omega}$ are defined as:

$\tilde{\omega}_i = \omega_i$, $\omega_i \geq 0$; $\tilde{\omega}_i = 0$, otherwise

$\hat{\omega} = \tilde{\omega} - \omega$

4.7 Algorithm 7: Computation of an upper bound on the p^{th} largest component of L * ω using the p^{th} largest row norm.

Input: N by M loss matrix **L** and M by 1 portfolio vector **ω.**

Output: An upper bound $\hat{V}_R$ on the p^{th} largest component of L * ω.

Procedure:

- **Initialize:** Compute the L_2 norm of each row of L, and let W_L be the p^{th} largest one.
- **Computation:**
  1. Compute $\|\omega\|_2$, the L_2 norm of ω in O(M) time.
  2. Compute $\hat{V}_R = W_L * \|\omega\|_2$ in constant time.

If $\hat{V}_R \leq V_c$, then the answer to the decision problem is No. Otherwise, the answer is not determined by this bound. This procedure takes O(M) time. If it fails to give an answer, then we fall back upon the conventional technique.

In Algorithm 7 the p^{th} largest row norm is used to estimate the upper bound of V_R. This is done by first computing the row norm for each row in the matrix L, then quicksort is used to sort the data in order and the p^{th} largest row norm, labeled as W_L , is chosen.

The upper bound of VaR is estimated by multiplying W_L with $\|\omega\|_2$, where $\|\omega\|_2$ stands for the L_2 norm of the portfolio vector ω when it is given.

4.8 Algorithm 8: Eliminating rows with "small" norms.

Input: N by M loss matrix **L**, M by 1 portfolio vector **ω** and collateral V_c**.**

Output: An answer to the decision problem: **Yes** or **No.**

Procedure:

- Initialize:
  1. Compute the L_2 norm of each row of L.

2. Sort the rows of L in descending order of their L_2 norms and store them in a matrix $\tilde{L}$. If $\tilde{L}_i$ denotes the i^{th} row of $\tilde{L}$, then i < j implies $\left\|\tilde{L}_i\right\| \geq \left\|\tilde{L}_j\right\|$ since the rows are sorted.
3. Store the corresponding norms of the rows in a sorted array $\tilde{l}$.

- Computation:
  1. Compute $\|\omega\|_2$ in O (M) time.
  2. Determine the largest index s such that $\tilde{l}_s > |V_c|/\|\omega\|$. If there is no such component, then set s = 0. This index can be determined in O(log N) time using binary search.
  3. If s < *p*, then there are fewer than p rows with a potential for their inner product exceeding V_c, and therefore the answer to the decision problem is No. Otherwise compute inner produce $\tilde{L}_i * \omega$ for each i ≤ s and call the set of inner products computed $\hat{b}$. This takes O(sM) time.
  4. If the number of elements in $\hat{b}$ greater than V_c is fewer than p, then the answer to the decision problem is No. Otherwise, it is Yes. This can be done in O(s) time.

Therefore the total time taken is O(sM + log N). In the worst case, it can be O(N M). However, if s is small, then there is the possibility for a significant reduction in time.

In Algorithm 8, we try to find out how many rows there are with the possibility of their L_2 norm being greater than $|V_c|/\|\omega\|$. Where $\|\omega\|$ stands for the L_2 norm of the portfolio vector. If this number s is smaller than *p* we answer the VaR decision problem with No. Otherwise we check those rows with large L_2 norms.

4.9 Algorithm 9: Heuristic for potential reduction in time when VaR is exceeded, by first computing the inner product for rows with large norms.

Input: N by M loss matrix **L**, M by 1 portfolio vector **ω**, and collateral V_c.
Output: An answer: **Yes.** to the decision problem, or **Undecided.**
Procedure:

- Initialize: Compute the L_2 norm of each row of L, and let $\hat{L}$ be the matrix consisting of the h rows with the largest L_2 norms, where h is a fixed constant greater than *p*.
- Computation:
  1. Compute $\hat{L} * \omega$ in O(hM) time.
  2. Count the number q of components of $\hat{L} * \omega$ that are greater than V_c in O(h) time.
  3. If q ≥ *p*, then the answer is Yes. Otherwise, the answer could not be determined by this procedure, and so return Undecided.

This procedure takes O(hM) time, where we can choose the constant h such that $p < h << N$. If it fails to give an answer, then we compute the inner product with the rest of the rows, and proceed as usual. This procedure can use U, DV **ω** instead of L, **ω** too, but there will be an additional O(M^2) time, due to the cost of computing DV **ω.**

In Algorithm 9, we try to find those rows with large L_2 norm and compute their profit or loss *b* = L * ω. These h rows of data with a greater chance of their loss exceeding collateral V_c compared to those rows with small norm values. This algorithm focuses on computing the result for these h rows in order to answer the VaR decision problem before running a Monte Carlo simulation. If there are more than *p* rows with their loss over V_c, there will definitely be more then *p* rows with losses over V_c after sorting the results so we can answer Yes to the VaR decision problem. Otherwise, the results for the rest of the rows are checked and the total number of rows with their loss exceeding V_c is determined. If the total number is greater than *p*, answer Yes to the VaR decision problem. Otherwise, answer No.

4.10 Algorithm 10: A composite algorithm for the VaR decision problem.

Input: N by M loss matrix L, M by 1 portfolio vector **ω,** and collateral V_c.

Output: An answer to the decision problem: **Yes** or **No**.

Procedure:

- Step 1: Determine an upper bound $\hat{V}_R$ on the p^{th} largest component of L * ω, using bounds given in Algorithm 4, 5, 6, 7.
- Step 2: If $\hat{V}_R \leq V_c$, then terminate this algorithm with answer No.
- Step 3: Otherwise, apply Algorithm 9.
- Step 4: If the answer from he above step is Yes, then terminate this algorithm with answer Yes.
- Step 5: Apply Algorithm 1 and return the answer from this algorithm.

The algorithms based on bounds are used to give the answer No when VaR does not exceed V_c, while the method used in Algorithm 9 gives the answer when VaR exceeds V_c. Algorithm 10 gives a strategy of combining Algorithm 4, 5, 6, 7 with Algorithm 9 to handle both cases in general and reduce the uses of Monte Carlo simulation.

5. Tests

The following tests are implemented based on algorithms mentioned in section 4. Each algorithm will be tested with 10,000 portfolios with 10,000 draws in each portfolio. The number of assets in a portfolio is set to be M = 5, 16 and 32. The level of confidence is set to 99%. All programs are composed using C for CUDA and compiled using Microsoft Visual Studio 2008 on Microsoft Windows 7.

Algorithm tested:

1. Algorithm 1: The conventional technique.
2. The algorithm using the L_2 bound in Algorithm 4.
3. The algorithm using the L_1 bound in Algorithm 5.
4. The algorithm using the L_1 bound in Algorithm 5, with U, DV ω replacing L, ω.
5. The algorithm using the L_∞ bound in Algorithm 6.
6. The algorithm using the L_∞ bound in Algorithm 6, with U, DV ω replacing L, ω.
7. Algorithm 7.
8. Algorithm 7, with U, DV ω replacing L, ω.
9. Algorithm 8.
10. Algorithm 8, with U, DV ω replacing L, ω.
11. Algorithm 9, with h = 0.1 * N.
12. Algorithm 9, with U, DV ω replacing L, ω and h = 0.1 * N.
13. Algorithm 10 where the L_1 of Algorithm 5 is used in step 1 (with U, DV ω replacing L, ω), and the algorithm in Algorithm 1 is used instead in step 5.
14. Algorithm 10 where the L_1 of Algorithm 5 is used in step 1 (with U, DV ω replacing L, ω), and the algorithm in Algorithm 1 is used in step 5. Step 3 uses U, DV ω instead of L, ω and h = 0.1 * N.

Test 1: Algorithm 1: The conventional technique.

In the conventional technique to compute VaR, we rely on matrix-vector multiplication to compute b. That is, the loss matrix L multiplied by the weight vector ω. The time complexity for this procedure is O(NM) if we have loss matrix L with N by M and weight vector ω with M by 1. We then sort the results in ascending order using quicksort. The p^{th} largest component will be the VaR estimation for this portfolio.

```
1.  //This code multiplies matrix md with vector nd and put the result into pd
2.  __global__ void MatrixVectorMulKernel(float *md , float *nd, float
3.  *pd , int m_row, int m_col , clock_t* time)
4.  {
5.       //shared memory space for md
6.       __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
7.       //shared memory space for nd
8.       __shared__ float Nds[TILE_WIDTHx];
9.
10.      //int bx = blockIdx.x;
11.      int by = blockIdx.y;
12.      int tx = threadIdx.x;
13.      int ty = threadIdx.y;
14.
15.      int Row = by * TILE_WIDTHy + ty;
16.
17.      float pvalue=0;
18.
19.      if(ty==0) time[by]=clock();
20.
21.      //Load the data needed based on the size of the shared mamory
22.      //and compute the result
23.      for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
24.      {
25.         Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
26.         Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
27.         __syncthreads();
28.
29.         for(int j=0 ; j < TILE_WIDTHx ; j++)
30.         {
31.            pvalue += Mds[ty][j] * Nds[j];
32.         }
33.
34.         *(pd + Row) = pvalue;
35.      }
36.
37.      if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy] = clock();
38.}
```

Table 6: CUDA kernel for doing matrix times a vector

The CUDA kernel above performs the matrix-vector multiplications in parallel. It takes seven arguments. Float pointer *md, *nd and *pd points to the space declared in GPU using cudaMalloc( ) function. Argument m_row and m_col specify the row and column number of matrix md.

The size of shared memory is defined by two constants: TILE_WIDTHx and TILE_WIDTHy. The space for shared memory should be no greater than 512 on the Nvidia Quadro FX 3700. We use the following code to define the number of threads in a block and the number of blocks:

```
dim3 threadsperBlock( TILE_WIDTHx , TILE_WIDTHy );
dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy );
```

Code to define thread number and block number

In the case of M = 16 for the weight vector, we define TILE_WIDTHx = 16 and TILE_WIDTHy = 20 for each block and we divide our memory space into 1 * 500 = 500 blocks in the program.

Memory space used in GPU is allocated using the function cudaMalloc( ) and is freed using function cudaFree( ):

```
cudaMalloc((void**) &md, size_mdgpuspace);
cudaMalloc((void**) &nd, size_ndgpuspace);
cudaMalloc((void**) &pd, size_pdgpuspace);
```

CUDA function cudaMalloc( ) to allocate memory space in device

Data transfer between the host (CPU) memory and the device (GPU) memory uses the function cudaMemcpy( ):

```
cudaMemcpy( md , loss_m , size_mdgpuspace , cudaMemcpyHostToDevice);
```

CUDA function cudaMemcpy( ) to move data between host and device

The code above moves our loss matrix L into GPU memory for computation.

The CUDA kernel is then invoked by calling our CUDA function:

```
MatrixVectorMulKernel<<< numBlocks , threadsperBlock >>>( md , nd , pd,
TEMP_AREA , M, CUDA_time );
```

Invoke CUDA kernel function to compute matrix-vector multiplication

Once we finish the matrix-vector computations, the function cudaMemcpy( ) is used to copy the result back to CPU memory for further use:

```
cudaMemcpy( b_temp , pd , size_pdgpuspace , cudaMemcpyDeviceToHost);
```

CUDA function cudaMemcpy( ) to move data between host and device

After we finish the matrix-vector multiplication, the result in b_temp should be the profit for these 10,000 draws. Quicksort is used to sort the result into ascending order.

```
1. //quicksort in ascending order
2. void quicksort(float* target, int left , int right)
3. {
4.    int i=left , j=right;
5.    float temp;
6.    float pivot = target[(left+right)/2];
7.
8.    while(i<=j)
9.    {
10.         while((target[i]<pivot)&&(i<right))
11.         {
12.               i++;
13.         }
14.
```

```
15.          while((target[j]>pivot)&&(j>left))
16.          {
17.                j--;
18.          }
19.
20.          if(i<=j)
21.          {
22.                temp=target[i];
23.                target[i] = target[j];
24.                target[j] = temp;
25.                i++;
26.                j--;
27.          }
28.
29.    }
30.
31.    if(left < j)
32.    {
33.          quicksort(target , left , j);
34.    }
35.    if(i < right)
36.    {
37.          quicksort(target , i , right);
38.    }
39.
40. }
```

Table 7: Code for quicksort in ascending order

The result VaR will be the p^{th} largest element after we sort the results in ascending order.

If the value of VaR exceeds V_c, the COLLATERAL value defined on the top of the program, the answer to the decision problem is Yes and the counter not_to_buy is increased by 1, otherwise the answer to the decision problem is No and the counter to_buy is increased by 1.

Function QueryPerformanceCounter( ) is used to estimate the execution time for the computation.

```
QueryPerformanceCounter(&t1);
…
QueryPerformanceCounter(&t2);
```

Function to compute the execution time needed for the code

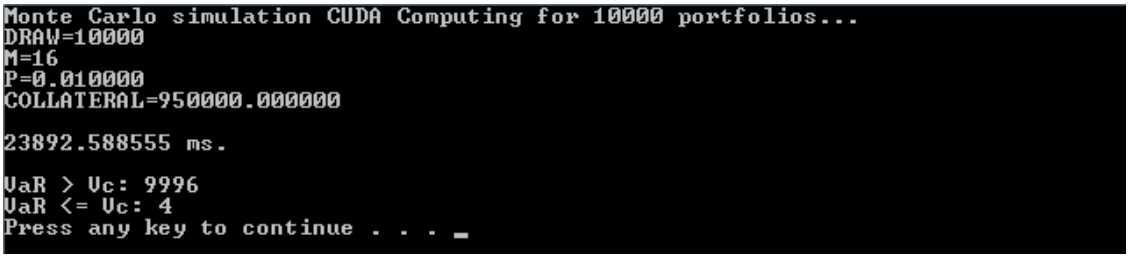

Figure 4: Output for cudaprog1main.cu

Complete code for Algorithm 1 is in cudaprog1main.cu.

Test 2: The algorithm using the L_2 bound in Algorithm 4.

In test 2 we need to compute S = $\sum_{j=1}^{M} \omega_j S_j$ using Algorithm 2 and W = $\sum_{i=1}^{N} b_i^2$ using Algorithm 3, the upper bound of VaR will be estimated by the formula:

$$\hat{V_R} = (1/N) * \left(S + \sqrt{\frac{(N-p)}{p} * (N*W - S^2)}\right)$$

For computing S and W, we can pre-compute the summation of each column of L. The matrix multiplication D * V can also be pre-computed before we start the simulation.

The following CUDA code computes the summation for each column of matrix:

```
1.  //(CUDA) Computing the column sum of each column in the matrix
2.  __global__ void CUDA_matrix_col_sum(float *a, float *b, int m_row ,
3.  int m_col)
4.  {
5.        //shared memory space for matrix a
6.      __shared__ float partialsum[TILE_WIDTHy3][TILE_WIDTHx];
7.
8.      int bx = blockIdx.x;
9.      int by = blockIdx.y;
10.
11.     int tx = threadIdx.x;
12.     int ty = threadIdx.y;
13.
```

```
14.   int Row = by * TILE_WIDTHy3 + ty;
15.   int Col = bx * TILE_WIDTHx + tx;
16.
17.   partialsum[ty][tx] = *(a+Row*m_col+Col);
18.   __syncthreads(); //
19.
20.   //
21.   for(int i=1 ; i < TILE_WIDTHy3 ; i *= 2)
22.   {
23.         __syncthreads();
24.
25.         if( (ty % (2*i) ==0) )
26.         {
27.               //Make sure matrix index not exceed it's maximum!!
28.               if(ty+i < TILE_WIDTHy3)
29.               {
30.                  partialsum[ty][tx] += partialsum[ty+i][tx];
31.               }
32.               else
33.               {
34.               }
35.               __syncthreads();
36.         }
37.
38.     }
39.     __syncthreads();
40.
41.   *(b + by * m_col + Col) = partialsum[0][tx]; //
42. }
```

Table 8: CUDA kernel for computing the sum of each column of matrix a

The summation of each column of L will be computed using our CUDA function CUDA_matrix_col_sum:

```
CUDA_matrix_col_sum<<< numBlocks ,threadsperblock >>>( CUDA_mloss ,nd ,
TEMP_AREA , M );
```

Invoke CUDA kernel CUDA_matrix_col_sum to comptute column sum of L

The matrix multiplication D * V is computed using our CUDA function MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks2 , threadsperblock2 >>>(CUDA_md ,
CUDA_mv , CUDA_dv , M , M);
```

Compute D * V using CUDA Kernel function MatrixMulKernelv3

Once the pre-computation is finished, we start the simulation by fetching a row of portfolio data using function protf_data_cpy:

```
portf_data_cpy(m_portf , m_portfolio, PORTF_NUM , M , k);
```

Copy portfolio data from matrix m_portf into m_portfolio

This function will take one row of data in matrix m_portfo and store it into vector m_portfolio for later use.

A matrix-vector multiplication for computing D * V * ω is performed by using the function matrix_multi( ):

```
matrix_multi( dv_temp , m_portfolio , m_dvw , M, M, M,1, M,1);
```

Compute D * V * ω using function matrix_multi( )

This function takes the result of D * V stored in dv_temp and multiplies it with the vector m_portfolio. The result is then stored in m_dvw.

S used in Algorithm 2 is computed by multiplying each element in m_portfolio with the summation of each column in L and then taking its summation. It is done by using two functions.

```
vector_ele_multi( matrix_col_sum , m_portfolio, m_temp1, M );
ls = vector_sum( m_temp1, M);
```

Compute S used in Algorithm 2

W used in Algorithm 3 will be computed using the function vector_power2_sum:

```
lw = vector_power2_sum(m_dvw , M);
```

Compute W used in Algorithm 3

This function computes the summation of the square of each element in m_dvw.

The result W = $b^T * b$ is the summation of the square of each element of b = L * ω. By using singular value decomposition it can be computed using D * V * ω instead of actually computing L * ω. This will save time since for matrix D, the singular value only exists on its diagonal so we only need M rows of them to compute D * V * ω. Its size is much smaller compared to L * ω.

The upper bound for the VaR is estimated by the equation:

$$\hat{V}_R = (1/N) * \left(S + \sqrt{\frac{(N-p)}{p} * (N * W - S^2)}\right)$$

```
vr_upperbound = (1/(float)DRAW) * (ls + sqrt( ((DRAW-sp)/sp) * (DRAW *
lw - pow(ls,2)) ));
```

Compute the upper bound of VaR in Algorithm 4

If the upper bound for VaR is smaller than V_c, answer No to the VaR decision problem. Otherwise compute the actual value at risk using Monte Carlo simulation.

```
CUDA computing for VaR with 10000 portfolios.

CUDA_Program2:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000

23061.028650 ms.

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 5: Output for cudaprog2main.cu

For the complete code of test 2, see cudaprog2main.cu in section 9.

Test 3: The algorithm using the L_1 bound in Algorithm 5.

In test 3 we pre-compute L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} during the initialization stage. This is done by the following code:

```
1. for(int i=0; i<M ; i++)
2. {
3.       matrix_col_cpy( m_loss, m_pv, DRAW, M , i);
4.       quicksort(m_pv, 0, DRAW-1);  //quicksort into ascending order
5.
6.       //sum of the p largest components of column j of L
7.       ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
8.       //sum of the N-p smallest components of column j of L
9.       ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
10.      //sum of the p smallest components of column j of L
11.      ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
12.      //sum of the N-p largest components of column j of L
13.      ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
14. }
```

Table 9: Code for computing L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} for each column of L

In line 3 we first copy one column of data in our loss matrix L into a vector m_pv using the function matrix_col_cpy( ). Quicksort is then applied to this vector to sort it into ascending order. The sum of the p largest components L_{lp}^{j}, the sum of the N-p smallest components L_{sp}^{j}, the sum of the p smallest components L_{lm}^{j} and the sum of the N-p largest components L_{sm}^{j} for column *j* of the loss matrix are then computed using function vector_sum2( ).

The summation of each column in loss matrix L is also computed during the initialization stage using the CUDA kernel function CUDA_matrix_col_sum:

```
//compute column sum for each column in loss matrix
CUDA_matrix_col_sum<<< numBlocks , threadsperblock >>>( CUDA_ca ,
CUDA_cb , TEMP_AREA , M );
```

Computing column sum of L using CUDA kernel function CUDA_matrix _col_sum

The computation starts by first fetching a row of data from matrix m_portf into vector m_protfolio using the function portf_data_cpy( ):

```
portf_data_cpy(m_portf , m_portfolio , PORTF_NUM , M , k);
```

Copy portfolio data from matrix m_portf into m_portfolio

The S used in Algorithm 2 is then computed as the summation of column sum for each column multiplied by its corresponding portfolio value:

```
vector_ele_multi( matrix_col_sum , m_portfolio , m_temp1 , M);
ls = vector_sum( m_temp1 ,M );
```

Compute S used in Algorithm 2

$\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$ and $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ are computed by the following code:

```
1. for(int i=0; i<M ; i++)
2. {
3.       //compute w_lomda
4.       if(*(m_portfolio+i) >=0)
5.       {
6.           w_londa = *(m_portfolio+i);
7.       }
8.       else
9.       {
10.          w_londa = 0.0;
11.      }
12.      //compute w_arrow
13.      w_arrow = w_londa - *(m_portfolio+i);
14.
15.      a1 += w_londa * ( ljlp[i] - ljsp[i] );
16.
17.      b1 += w_arrow * ( ljlm[i] - ljsm[i] );
18.  }
```

Table 10: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$ and $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ in algorithm 5

$\tilde{\omega}_i$ is equal to the element in m_portfolio if it is greater than zero, or set to zero otherwise in line 6. $\hat{\omega}_i$ is equal to $\tilde{\omega}_i$ minus the element in m_portfolio in line 13. These two values always have non-negative values.

Variable a1 = $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$ and b1 = $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ in line 15 and 17.

The upper bound of VaR used in this test is then estimated by the following function:

$\hat{V}_R$ = 1/(2 * p) * $\left[S + \sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i) - \sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i) \right]$

```
vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
```

Compute the upper bound of VaR in Algorithm 5

If the upper bound of VaR is smaller than V_c, answer No to the VaR decision problem. Otherwise use the conventional Monte Carlo simulation to compute the actual value at risk.

```
Loading matrix Loss...
Loading portfolio data.

CUDA_Program3:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000

23426.760282 ms.
Total clocks: 23431

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 6: Output for cudaprog3main.cu

For complete code of test 3, see cudaprog3main.cu in section 9.

Test 4:

The algorithm using the L_1 bound in Algorithm 5, with U, DV ω replacing L, ω.

The steps used in test4 are basically the same as test3 except we replace the loss matrix L and portfolio ω with matrix U and vector D * V * ω.

In the initialization stage we compute only the first M columns of data in U. The summation and four variables L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} for each column are computed using the following code:

```
1. for(int i=0; i<M ; i++)
2. {
3.      matrix_col_cpy( m_u, m_pv, DRAW, M , i);
4.      quicksort(m_pv, 0, DRAW-1);  //quicksort
5.      //compute the sum of each column of U
6.      mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);
7.
8.      //sum of the p largest components of column j of L
9.      ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
10.     //sum of the N-p smallest components of column j of L
11.     ljsp[i] = vector_sum2( m_pv , 0 , (DRAW – (int)sp)-1 );
12.     //sum of the p smallest components of column j of L
13.     ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
14.     //sum of the N-p largest components of column j of L
15.     ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
16. }
```

Table 11: Code for computing L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} for each column of U

Lines 3 and 4 read in the columns in matrix U and sort the values into ascending order using quicksort. Line 6 stores the corresponding column sum in array mu_col_sum.

Lines 8 through 15 compute the corresponding L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} for each column.

The computation of D * V is done during the initialization stage by calling the CUDA kernel function MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb ,
CUDA_cc, M, M);
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The first M columns of matrix U is also loaded into GPU memory so that we do not have to do it later during the computation stage.

```
cudaMemcpy(CUDA_utemp , m_utemp , size_CUDA_utemp , cudaMemcpyHostToDevice);
```

Copy M columns of U to GPU memory

In the computation stage we start by loading a row of portfolio data:

```
portf_data_cpy(m_portf , m_portfolio , PORTF_NUM , M, k);
```

Copy portfolio data from matrix m_portf into m_portfolio

The vector for D * V * ω is then computed using the function matrix_multi( ):

```
matrix_multi( dv_temp , m_portfolio , m_dvw, M, M, M, 1, M, 1 );
```

Compute D * V * ω using function matrix_multi( )

S, which is used in Algorithm 2 is then computed by multiplying the summation of each column with the corresponding element in vector D * V * ω:

```
vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);
ls = vector_sum2(m_stemp , 0 , M-1);
```

Compute S used in Algorithm 2

$\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$ and $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ are computed by the following code:

```
1. for(int i=0; i<M ; i++)
2. {
3.       //compute w_lomda
4.       if(*(m_dvw+i) >=0)
5.       {
6.            w_londa = *(m_dvw+i);
7.       }
8.       else
9.       {
10.           w_londa = 0.0;
11.      }
12.      //compute w_arrow
13.      w_arrow = w_londa - *(m_dvw+i);
14.
15.      a1 += w_londa * ( ljlp[i] – ljsp[i] );
16.
17.      b1 += w_arrow * ( ljlm[i] – ljsm[i] );
18. }
```

Table 12: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$ and $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ in algorithm 5

Notice that in line 4 we are using D * V * ω instead of ω which was used in test 3. This is because U, D * V * ω are used instead of **L,** ω in test 4.

The upper bound on the value at risk is then estimated using the same function as used in test 3:

```
vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 – b1)  );
```

Compute the upper bound of VaR in Algorithm 5

If our upper bound of VaR is smaller than V_c , answer No to the VaR decision problem. Otherwise use Monte Carlo simulation to estimate the actual value at risk.

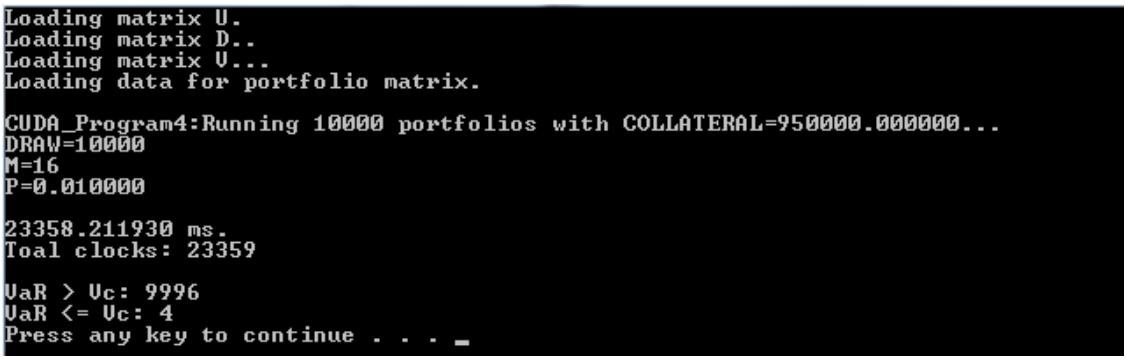

Figure 7: Output for cudaprog4main.cu

For the complete code of test 4, see cudaprog4main.cu in section 9.

Test 5

Computation of an upper bound on the p^{th} largest component of L * ω using the L_∞ norm.

In test 5 we estimate the VaR upper bound using L_∞ norm. The maximum L_p^j and the minimum value L_m^j in each column will be found during the initialization stage. During the computation stage the upper bound of VaR is estimated and a decision is made.

The following code finds out the minimum and maximum in each column of the loss matrix L using the CUDA kernel CUDA_find_minmax:

```
1. for(int i=0; i<M ; i++)
2. {
3.    matrix_col_cpy(m_loss , col_temp , DRAW , M , i);  //
4.    //move data from col_temp (Host) to ca (Device)
5.    cudaMemcpy(ca , col_temp , size_CUDA_ca ,
cudaMemcpyHostToDevice);
6.    //find out the minimum/maximum value in the column using cuda
core
7.    //CUDA_find_minmax
8.    CUDA_find_minmax<<< numBlock , threadsPerBlock >>>( ca, cb );
9.
10.   cudaMemcpy( ans_temp, cb , size_CUDA_cb, cudaMemcpyDeviceToHost);
```

```
11.   vect_min_max( ans_temp , col_min_max, DRAW/TILE_WIDTHx);
12.
13.   *(column_min+i) = *(col_min_max+0);
14.   *(column_max+i) = *(col_min_max+1);
15. }
```

Table 13: Code to find the minimum/maximum value in each column of L

The result minimum value L_m^i and maximum value L_p^i for each column will be stored in vector column_min and column_max in lines 13 and 14.

The loss matrix L will be loaded into the GPU space for the preparation of Monte Carlo simulation.

```
cudaMemcpy(CUDA_utemp , m_loss , size_CUDA_utemp , cudaMemcpyHostToDevice);
```

Copy data of L into GPU memory using CUDA function cudaMemcpy( )

In the computation stage we start by fetching a row of portfolio data into the vector portfolio by using the portf_data_cpy function:

```
portf_data_cpy( portf_m , portfolio , PORTF_NUM , M , k);
```

Copy portfolio data from matrix m_portf into m_portfolio

The corresponding $\tilde{\omega}_i$, $\hat{\omega}_i$ for each element in the portfolio and $\sum_{i=1}^{M} L_p^i * \tilde{\omega}_i$, $\sum_{i=1}^{M} L_m^i * \hat{\omega}_i$ are computed by the following code:

```
1. for(int i=0; i<M ; i++)
2. {
3.     if(*(portfolio+i) >= 0.0)
4.     {
5.           w_londa = *(portfolio+i);
6.     }
7.     else
8.     {
9.           w_londa = 0.0;
10.    }
11.    //compute w_arrow
12.    w_arrow = w_londa - *(portfolio+i);
13.
14.    a1 +=  *(column_max+i) * w_londa;
```

```
15.     b1 +=  *(column_min+i) * w_arrow;
16.
17. }
```

Table 14: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} L_p^i * \tilde{\omega}_i$ and $\sum_{i=1}^{M} L_m^i * \hat{\omega}_i$ in Algorithm 6

The upper bound of VaR using L_∞ norm is then estimated by:

$$\hat{V}_R = \sum_{i=1}^{M} L_p^i * \tilde{\omega}_i - \sum_{i=1}^{M} L_m^i * \hat{\omega}_i$$

```
vr_upperbound = a1 - b1;
```

Compute the upper bound of VaR in Algorithm 6

If our upper bound is smaller than V_c, answer the VaR decision problem with No, Otherwise answer the problem by using Monte Carlo simulation.

```
Loading data:

CUDA_Program5:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000

23014.683251 ms.
Total clocks:23015

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 8: Output for cudaprog5main.cu

For the complete code of test 5, see cudaprog5main.cu in section 9.

Test 6

The algorithm using the L_∞ bound in Algorithm 6, with U, D*V*ω replacing L, ω.

In test 6 we compute the upper bound of VaR using the L_∞ bound, but replacing L, ω in test 5 with U and D * V * ω.

In the initialization stage the minimum value L_m^j and the maximum value L_p^i in the first M columns of U are computed using the code below:

```
1. for(int i=0 ; i<M ; i++)
2. {
3.       matrix_col_cpy(m_u , col_temp, DRAW , M , i);
4.
5.       matrix_find_min_max( m_u , p_minmax , DRAW, M , i);
6.
7.       lp[i] = *(p_minmax+0);//Max value for each column
8.       lm[i] = *(p_minmax+1);//min value for each column
9. }
```

Table 15: Code to find the minimum/maximum value in each column of U

Only the first M columns of U will be processed in the loop. Line 3 copies a column of U into col_temp and uses the matrix_find_min_max function to find its minimum and maximum. The result will be placed into matrix lp and matrix lm.

D * V also is computed during the initialization stage by calling the CUDA kernel MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb ,
CUDA_cc, M, M);
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The computation stage starts by first fetching a row of portfolio data:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);
```

Copy portfolio data from matrix m_portf into m_portfolio

The vector D * V * ω is then computed using matrix_multi function:

```
matrix_multi( dv_temp , portfolio , m_dvw, M, M, M, 1, M, 1 );
```

Compute D * V * ω using function matrix_multi( )

The corresponding $\tilde{\omega}_i$, $\hat{\omega}_i$ for each element in portfolio and $\sum_{i=1}^{M} L_p^i * \tilde{\omega}_i$, $\sum_{i=1}^{M} L_m^i * \hat{\omega}_i$ are computed using the code below:

```
1. for(int i=0 ; i<M ; i++)
2. {
3.       if(*(m_dvw + i) >= 0.0)
4.       {
5.           w_londa = *(m_dvw + i);
6.       }
7.       else
8.       {
9.           w_londa = 0.0;
10.      }
11.      w_arrow = w_londa - *(m_dvw + i);
12.
13.      a1 +=  lp[i] * w_londa;
14.      b1 +=  lm[i] * w_arrow;
15. }
```

Table 16: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} L_p^i * \tilde{\omega}_i$ and $\sum_{i=1}^{M} L_m^i * \hat{\omega}_i$ in Algorithm 6

The upper bound of VaR estimation using L_∞ norm is estimated by the following equation:

$$\hat{V}_R = \sum_{i=1}^{M} L_p^i * \tilde{\omega}_i - \sum_{i=1}^{M} L_m^i * \hat{\omega}_i$$

```
vr_upperbound = a1 - b1;
```

Compute the upper bound of VaR in Algorithm 6

If the upper bound of VaR is smaller than V_c , answer the VaR decision problem with No. If it fails to give an answer, then run Monte Carlo simulation and compute the actual value at risk.

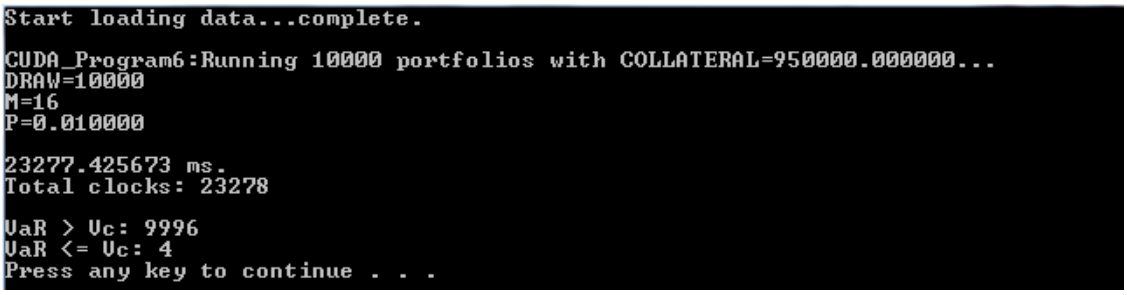

Figure 9: Output for cudaprog6main.cu

For complete code of test 6, see cudaprog6main.cu in section 9.

Test 7

Algorithm 7: Computation of an upper bound on the p^{th} largest component of L *ω using the p^{th} largest row norm

In test 7 we estimate the upper bound of VaR using the information provided by the p^{th} largest row norm of the loss matrix L. The L_2 norm for each row of matrix L is computed and the p^{th} largest norm is found using quicksort during the Initialization stage.

The following code computes the L_2 norm for each row of L by using the CUDA kernel CUDA_matrix_row_l2norm to compute the summation of the square of each element in a row:

```
1. for(int i=0 ;i< (DRAW/TEMP_AREA) ; i++)
2. {
3.     cudaMemcpy(CUDA_ca , m_loss , size_CUDA_ca ,
cudaMemcpyHostToDevice);
4.
5.     //Invoke CUDA kernel function for computing the L2 norm for each
6.     //row in matrix CUDA_ca
7.     CUDA_matrix_row_l2norm<<< numBlocks , threadsperblock
>>>( CUDA_ca , CUDA_cb , DRAW ,M);
8.
9.     cudaMemcpy( m_pl2norm , CUDA_cb , size_CUDA_cb ,
cudaMemcpyDeviceToHost );
10.
11. }
```

Table 17: Code to compute the square of each row in L using CUDA kernel function CUDA_matrix_row_l2norm

The resulting vector m_pl2norm is then processed by function vector_squareroot to compute the L_2 norm for each row.

```
Vector_squareroot( m_pl2norm , DRAW);
```

Compute the L_2 norm for each row of L

Quicksort is then used to sort the norm in ascending order and the p^{th} largest component W_L is determined as the 100th largest term in the array under 1% of the confidence level with 10,000 draws.

```
Quicksort( m_pl2norm , 0 ,DRAW-1);
wl = m_pl2norm[DRAW – sp];
```

Compute the p^{th} largest component W_L using Quicksort

In the computation stage, we start by first fetching a row of data from matrix m_portf and then computing its L_2 norm, this value is stored in variable wl2norm:

```
portf_data_cpy( m_portf, portfolio , PORTF_NUM , M, k);
wl2norm = matrix_col_l2norm( portfolio, M, 1,0 );
```

Compute the L_2 norm of the portfolio

The upper bound of VaR is estimated by multiplying the portfolio row norm with the p^{th} largest component row norm of L as defined in Algorithm 7:

```
vr_upperbound = wl2norm * wl;
```

Compute the upper bound of VaR in Algorithm 7

If the upper bound of VaR is smaller than V_c, answer the VaR decision problem with No. Otherwise the answer is not determined by this bound and we must perform Monte Carlo simulation.

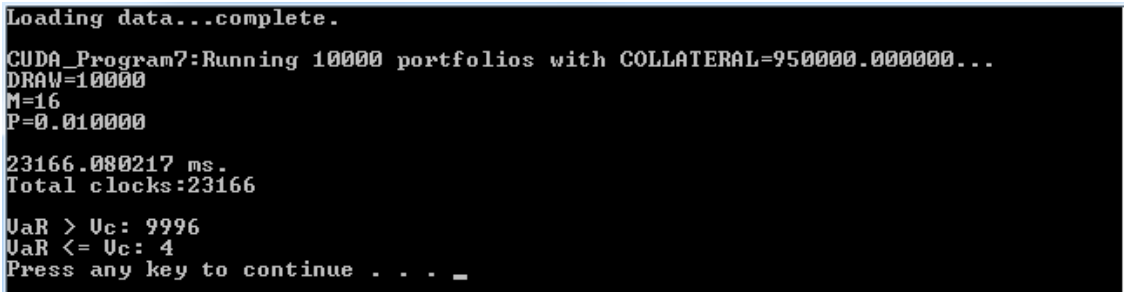

Figure 10: Output for cudaprog7main.cu

For complete code of test 7, see cudaprog7main.cu in section 9.

Test 8

Algorithm 7. With U, D * V *ω replacing L, ω.

In test 8 we estimate the upper bound of VaR using the same method used in test 7, but replacing L, ω with U and D * V *ω.

To compute the row norm of matrix U we compute only the first M elements in a row instead of the entire row.

```
1. for(int i=0 ; i<DRAW ; i++)
2. {
3.      //only compute L2 norm for the first M element in a row...
4.      m_pl2norm[i] = matrix_row_l2normv2( m_u , DRAW , M , 0, M-1 ,i);
5. }
```

Table 18: Code to compute L_2 norm for each row of U

Quicksort is used to sort the norm in ascending order:

```
quicksort(m_pl2norm , 0, DRAW-1);
```

Sort the norm using Quicksort

The p^{th} largest component W_L is determined as the 100th largest component under 1% of confidence level with 10,000 draws.

```
wl = m_pl2norm[DRAW - (int)sp];
```

Determine the p^{th} largest component W_L

The matrix multiplication D * V is done during the Initialization stage by calling the CUDA kernel function MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>( CUDA_ca ,
CUDA_cb , CUDA_cc , M ,M);
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The data in the first M columns of the matrix U is also loaded into GPU memory for Monte Carlo simulation:

```
cudaMemcpy( CUDA_utemp , m_utemp , size_CUDA_utemp , cudaMemcpyHostToDevice);
```

Copy data for M column of U into GPU memory using CUDA function cudaMemcpy( )

In the computation stage we start by fetching a row of portfolio data from matrix m_portf and then compute the M by 1 vector D * V * ω.

```
portf_data_cpy( m_portf , portfolio, PORTF_NUM , M , k);
matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1 , M ,1 );
```

Fetching a row of portfolio data and compute D * V * ω

The L_2 norm for this vector is then computed using function matrix_col_l2norm, the result is then stored in variable w_l2norm:

```
w_l2norm = matrix_col_l2norm( m_dvw, M,1,0);
```

Compute the L_2 norm of D * V * ω

The upper bound of VaR used in test 8 is estimated by multiplying the L_2 norm of D * V * ω with the p^{th} largest row norm of U:

```
vr_upperbound = wl * w_l2norm;
```

Compute the upper bound of VaR in Algorithm 7

If the upper bound of VaR is smaller than V_c, answer the VaR decision problem with No. Otherwise we must run Monte Carlo simulation.

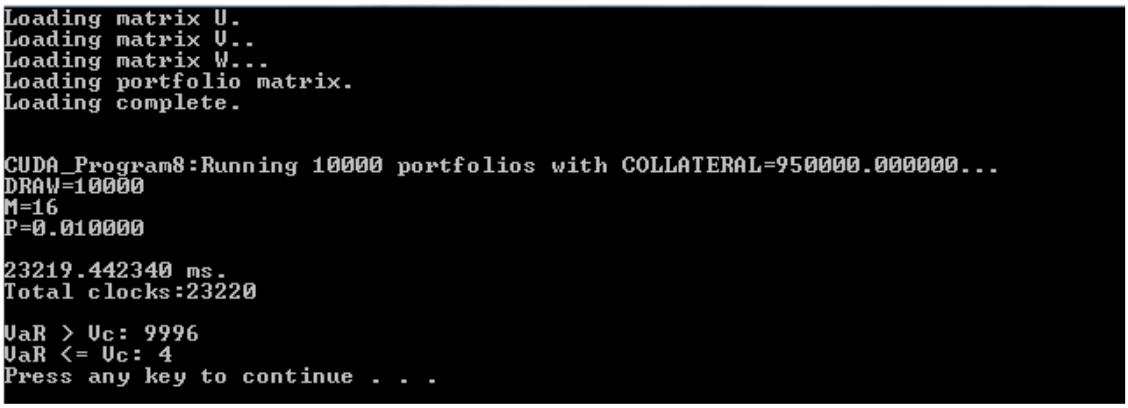

Figure 11: Output for cudaprog8main.cu

For the complete code of test 8, see cudaprog8main.cu in section 9.

Test 9

Algorithm 8: Eliminating rows with “small” norms.

In test 9 the L_2 norm of each row of L is computed and sorted using quicksort during the initialization stage:

```
1.  //1.compute the L2 norm of each row of L
2.  for(int i=0 ; i<DRAW ; i++)
3.  {
4.      //the first column indicate the order number of the element
5.      *(m_ll2norm+ 2*i )=(float)i;
6.      // the second column contain L2 norm of each row
7.      *(m_ll2norm+(2*i+1)) = matrix_row_l2norm(m_loss, DRAW,M, i);
8.  }
9.  //quicksort: sort it into descending order
10. quicksort2( m_ll2norm , 0, DRAW-1 );
```

Table 19: Code to compute the L_2 norm of each row of L and sorted in descending order using Quicksort

The first column of matrix m_ll2norm contains the row number of each row. The second column contains the L_2 norm for each row of L which is computed by function matrix_row_l2norm. This matrix m_ll2norm stands

for $\tilde{L}$ in Algorithm 8. If $\tilde{L}_i$ stands for the *i*th row of $\tilde{L}$, then $i < j$ implies $\left\|\tilde{L}_i\right\| \geq \left\|\tilde{L}_j\right\|$.

Function quicksort2 in line 10 is used to sort this matrix based on the second column into descending order. The result will be a sorted list of L_2, the norm for each row with its row number in the first column.

The data of the loss matrix L is copied, row by row, into matrix m_llonda based on the order of its row norm using the matrix_cpy2 function:

```
matrix_cpy2( m_loss , m_ll2norm , m_llonda , DRAW , M);
```

Copy row data of L into matrix m_llonda base on the order of its row norm

The loss matrix L is then loaded into GPU memory space for the preparation of Monte Carlo simulation during the initialization stage.

In the computation stage of test 9, we start by fetching a row of portfolio data from m_portf using the function protf_data_cpy:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);
```

Copy portfolio data form matrix m_portf into portfolio

The L_2 norm of this portfolio vector is then computed using the function matrix_row_l2norm:

```
w_l2norm = matrix_row_l2norm( portfolio , 1, M , 0);
```

Compute L_2 norm of the portfolio

The largest index s is determined by binary search such that $\tilde{L}_s > |V_c|/\|\omega\|$.
If there is no such component, set s to 0. This takes O(log N) time.

```
a1 = ((float)COLLATERAL)/w_l2norm;
index_s = binarysearch( m_ll2norm ,a1, 0 , DRAW-1);
```

Determine the index s in Algorithm 8 using Binary search

If the index s is smaller than *p*, answer the VaR decision problem with No. Otherwise run the conventional technique used in test 1 and perform Monte Carlo simulation.

```
Loading data for matrix L:

CUDA_Program9:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000

24193.162071 ms.
Total clocks: 24193

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 12: Output for cudaprog9main.cu

For the complete code of test 9, see cudaprog9main.cu in section 9.

Test 10

Algorithm 8 with U, D * V * ω replacing L, ω

In test 10 the same method is used as in test 9, except that we replace matrix L, ω with U and D * V * ω.

The first M columns of U are used in computing the row norm of U by the following code:

```
1. for(int i=0 ; i<DRAW ; i++)
2. {
3.     //the first column indicate the order number of the element
4.     *(m_ul2norm+ 2*i )=(float)i;
5.     //the second column contain L2 norm for the first M elements in
each row
6.     *(m_ul2norm+(2*i+1)) = matrix_row_l2normv2(m_u , DRAW, M, 0 , M-
1 , i);
7. }
```

Table 20: Code to compute the L_2 norm for each row of U

The first column of matrix m_ul2norm contains the row number for each row, the second column contains the L_2 norm for each row of U which is computed by the function matrix_row_l2normv2.

Quicksort is used to sort the matrix into descending order based on the L_2 norm of each column.

```
quicksort2( m_ul2norm , 0, DRAW-1 );
```

Sort the L_2 norm for each row of U into descending order using Quicksort

The matrix multiplication D * V is computed during the initialization stage using the CUDA kernel MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca ,
CUDA_cb , CUDA_cc , M ,M );
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The first M columns of matrix U is also copied into GPU memory space to prepare for Monte Carlo simulation:

In the computation stage we start by fetching a row of portfolio data from matrix m_portf:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);
```

Copy portfolio data from matrix m_portf into portfolio

Vector D * V * ω is then computed using the function matrix_multi:

```
matrix_multi( dv_temp , portfolio, m_dvw, M, M , M, 1 , M, 1);
```

Compute D * V * ω using function matrix_multi( )

The L_2 norm of vector D * V * ω is then computed using function matrix_col_l2norm:

```
dvw_l2norm = matrix_col_l2norm( m_dvw , M, 1 ,0);
```

Compute the L_2 norm of D * V * ω using function matrix_col_l2norm( )

The largest index s is determined such that $\tilde{L_s} > |V_c|/\|\omega\|$ using binary search:

```
a1 = ((float)COLLATERAL)/ dvw_l2norm;
index_s = binarysearch( m_ul2norm ,a1, 0 , DRAW-1);
```

Determine the index s in Algorithm 8 using Binary search

If s < *p*, then answer the VaR decision problem with No. Otherwise run the conventional technique used in test 1 and performs Monte Carlo simulation.

```
Loading data for matrix U:
Loading data for matrix V:
Loading data for matrix W:
Loading data for portfolio matrxi.


CUDA_Program10:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000

21408.855617 ms.
Total clocks:21409

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 13: Output for cudaprog10main.cu

For the complete code of test 10, see cudaprog10main.cu in section 9.

Test 11: Algorithm 9, with h = 0.1 * N.

In test 11 we use those steps provided in Algorithm 9 to find the h rows with large L_2 norm. These rows are expected to have larger magnitude of L*ω.

In this case h is set to have 0.1 * DRAW = 1000.

In the initialization stage we first determine those h rows with large norms. This is done by the following code in the program:

```
1.  for(int i=0 ; i<DRAW ; i++)
2.  {
3.      //The first column is the order number of the row
4.      *(m_ll2norm + 2*i) = (float)i;
5.      //The second column is the L2 norm of the row
6.      *(m_ll2norm + (2*i+1)) = matrix_row_l2norm(m_loss , DRAW, M ,
i);
7.  }
8.  //quicksort in descending order
9.  quicksort2(m_ll2norm , 0, DRAW-1);
```

```
10. //copy h rows of loss matrix into m_larrow based on its L2 norm
11. matrix_cpy2(m_loss , m_ll2norm, m_larrow , h , M);
```

Table 21: Code to determine h rows of L with large L_2 norm

Lines 1 through 7 compute the L_2 norm of each row in matrix L. Quicksort is used in line 9 to sort the matrix in descending order based on norm values. Those h rows with large norm values will be copied to a matrix m_larrow using the matrix_cpy2 function in line 11. This matrix and the original loss matrix are now ready to move into GPU memory space for computation.

The following code copies matrix m_larrow and m_loss into GPU memory space using the CUDA cudaMemcpy function:

```
1. //copy data m_larrow from CPU to GPU
2. cudaMemcpy( CUDA_ca , m_larrow , size_CUDA_ca , cudaMemcpyHostToDevice );
3. //move m_loss from host(CPU) to device(GPU)
5. cudaMemcpy( CUDA_ltemp , m_loss , size_CUDA_ltemp ,cudaMemcpyHostToDevice );
```

Copy h rows of L and entire matrix L into GPU memory

In the computation stage we start by fetching a row of portfolio data from matrix m_portf using the function portf_data_cpy:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);
```

Copy protfolio data from matrix m_portf into portfolio

The results for these h rows of matrix L will be computed using the CUDA matrix-vector multiplication kernel MatrixVectorMulKernel2:

```
1.  //comput L_arrow * w in O(hM) time
2.  for(int i=0 ; i< (h/TEMP_AREA) ; i++)
3.  {
4.      //Invoke CUDA core MatrixVectorMulKernel2 to do matrix-vector multiplication
5.      MatrixVectorMulKernel2<<< numBlock , threadsperblock >>>( CUDA_ca, CUDA_cb ,
CUDA_cc , TEMP_AREA , M);
6.      //copy the result from GPU to CPU
7.      cudaMemcpy( m_b_temp , CUDA_cc , size_CUDA_cc , cudaMemcpyDeviceToHost );
8.      //compute the number of elements in L_arrow * w that are greater than
COLLATERAL
9.      for(int i=0 ; i<TEMP_AREA ; i++)
10.     {
11.         if( *(m_b_temp+i) > COLLATERAL )
12.         {
```

```
13.             lq++;
14.         }
15.         else
16.         {
17.         }
18.     }
19. }
```

Table 22: Matrix-vector multiplication for h rows of L using CUDA kernel function MatrixVectorMulKernel2

Let q be those components with values greater than V_c, if q ≥ *p*, this means that there are more than *p* rows with its product greater than the COLLATERAL value, the answer to the VaR decision problem is Yes. Otherwise the answer is not determined by this procedure. This takes O(hM) time.

If this code fails to give an answer, we use Algorithm 1 to actually compute the value at risk.

```
Loading data for loss matrix, please wait...

CUDA_Program11:Running 10000 portfolios with COLLATERAL=950000.000000...
DRAW=10000
M=16
P=0.010000
h=1000

18285.595500 ms.
Total clocks:18286

VaR > Vc: 9996
VaR <= Vc: 4
Press any key to continue . . .
```

Figure 14: Output for cudaprog11main.cu

For the complete code of test 11, see cudaprog11main.cu in section 9.

Test 12: Algorithm 9, with U, D * V * ω replacing L, ω.

In test 12 the process is the same as in test 11 except the input is replaced with U, D * V * ω.

The ordered matrix U by its row L_2 norm is computed by the following code:

```
1.  //comput each row's L2 norm of U
2.  for(int i=0 ; i<DRAW; i++)
3.  {
4.      //The first column is the oreder number of the row
5.      *(u_l2norm+ 2*i) = (float)i;
6.      //only compute L2 norm for the first M elements of row in U
7.      *(u_l2norm+ (2*i+1)) = matrix_row_l2normv2( m_u , DRAW , M ,0,
M-1,i );
8.  }
9.
10. //quicksort in descending order
11. quicksort2(u_l2norm , 0 , DRAW-1);
12.
13. //copy data of m_u into u_larrow based on its L2 norm
14. //only copy the first M column of m_u into u_larrow
15. matrix_cpy2v2( m_u , u_l2norm , u_larrow , DRAW, M, 0, DRAW-1 , 0,
M-1 );
```

Table 23: Code to find h rows of U with large L_2 norm

Lines 1 through 8 compute the L_2 norm for each row of matrix U. Only the first M elements in a row are included.

Quicksort is used in line 11 to sort the matrix u_l2norm into descending order based on the value of its row norm. The function matrix_cpy2v2 is then used to copy the first M columns of U into a matrix u_larrow based on the order of its row norm in line 15.

The h rows with large L_2 norm are then selected from matrix u_larrow and copied into the GPU memory space CUDA_utemp1:

```
//move first h row (0 ~ (h-1))of U into m_utemp1
matrix_move( u_larrow , m_utemp1 , h , M, 0);
cudaMemcpy( CUDA_utemp1 , m_utemp1 , size_CUDA_utemp1 ,
cudaMemcpyHostToDevice );
```

Copy h rows of U into GPU memory using CUDA function cudaMemcpy( )

The rest of the rows are copied to another array in the GPU named CUDA_utemp2:

```
//move the rest of the row (h~(DRAW-1))to m_utemp2
matrix_move( u_larrow , m_utemp2 , DRAW-h , M, h);
cudaMemcpy( CUDA_utemp2 , m_utemp2 , size_CUDA_utemp2 ,
cudaMemcpyHostToDevice );
```

Copy the rest of the rows into GPU memory using cudaMemcpy( )

The computation stage starts by fetching a row of data from matrix m_portf and computing the vector D * V * ω:

```
portf_data_cpy(m_portf , portfolio, PORTF_NUM , M , k);
matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1, M, 1 );
```

Fetching portfolio data and compute D * V * ω

The inner products for these h rows are computed using the CUDA kernel function MatrixVectorMulKernel:

```
MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2 >>>
( CUDA_utemp1 , CUDA_dvw , CUDA_btemp1 , h , M);
```

Compute U * D * V * ω for h rows of U using CUDA kernel function MatrixVectorMulKernel

The result is a vector of profits computed by h rows of U multiplied by the vector D * V * ω, these elements are tested to see if they exceed V_c. If there are q elements that exceed the COLLATERAL value and q ≥ *p*, answer the VaR decision problem with Yes. Otherwise compute the inner product with the remaining rows using the CUDA kernel MatrixVectorMulKernel function to see if q ≥ *p*. If q ≥ *p*, there are more than *p* elements greater than V_c, the answer to the VaR decision problem is Yes. Otherwise, the answer is No.

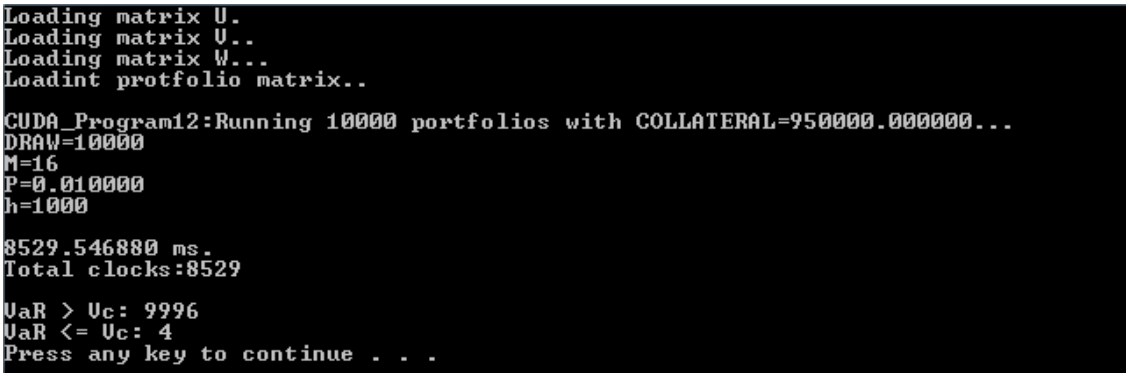

Figure 15: Output for cudaprog12main.cu

For the complete code for test 12, see cudaprog12main.cu in section 9.

Test 13
Algorithm 10: A composite algorithm for the VaR decision problem.

In test 13 the combination of Algorithm 1, Algorithm 5 and Algorithm 9 are used in each step. In step 1, Algorithm 5 is used to find out those cases with VaR not exceeding V_c. In step 3, Algorithm 9 is used to detect those cases with VaR exceed V_c, h is set to be 1000. If those two steps fail to answer the VaR decision problem, the conventional technique of Algorithm 1 is used to compute the actual value of value at risk.

In the initialization stage, four variables L_{lp}^{j} , L_{sp}^{j} , L_{lm}^{j} , L_{sm}^{j} and the summation for each column of U used in Algorithm 5 are computed by the following code:

```
1.  for(int i=0; i<M ; i++)
2.  {
3.      matrix_col_cpy( m_u, m_pv, DRAW, M , i);
4.      //quicksort in ascending order
5.      quicksort(m_pv, 0, DRAW-1);
6.      //compute the sum of each column of U
7.      mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);
8.      //sum of the p largest components of column j of L
```

```
9.      ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
10.     //sum of the N-p smallest components of column j of L
11.     ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
12.     //sum of the p smallest components of column j of L
13.     ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
14.     //sum of the N-p largest components of column j of L
15.     ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
16.
17. }
```

Table 24: Code for computing L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j}, L_{sm}^{j} and summation for each column of U in test 13

Line 3 takes one column of U into vctor m_pv and sorts it into ascending order using quicksort in line 5. The summation of the columns is computed using the function vector_sum2 in line 7. The corresponding L_{lp}^{j}, L_{sp}^{j}, L_{lm}^{j} and L_{sm}^{j} for each column of L are computed in lines 9, 11, 13 and 15.

The matrix multiplication D * V is computed using the CUDA kernel function MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb ,
CUDA_cc , M , M);
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The h rows with the largest row norm of L used in Algorithm 9 are selected by first computing the L_2 norm for each row and are sorted using quicksort:

```
1. for(int i=0 ; i<DRAW ; i++)
2. {
3.      *(m_ll2norm + 2*i) = (float)i;
4.      *(m_ll2norm + (2*i+1)) = matrix_row_l2norm(m_loss , DRAW, M ,
i);
5. }
6. quicksort2(m_ll2norm , 0, DRAW-1);
```

Table 25: Code to compute L_2 norm for each row of L

Those h rows of data are then selected from the loss matrix L, based on the order of their norm value. The result is then placed in matrix m_larrow:

```
1. //copy h rows of m_loss into m_larrow based on the order of its ROW L2 norm
2. for(int i=0; i<h ; i++)
3. {
4.     matrix_cpy2(m_loss , m_ll2norm, m_larrow , i , M);
5. }
```

Table 26: Code for selecting h rows of L with large norm

The matrix m_larrow used in step 3 and the loss matrix m_loss used in step 5 are then copied from host memory into the GPU memory using the CUDA function cudaMemcpy:

```
cudaMemcpy( CUDA_mtemp , m_larrow , size_CUDA_mtemp , cudaMemcpyHostToDevice );
cudaMemcpy( CUDA_m , m_loss , size_CUDA_m , cudaMemcpyHostToDevice );
```

Copy data from host memory to device memory

The computation stage starts by first fetching a row of portfolio data form matrix m_portf:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);
```

Fetching portfolio data from matrix m_portf to portfolio

In step 1 we use the upper bound of VaR estimated by Algorithm 5. The M by 1 vector D * V * ω is computed by multiplying D * V with portfolio vector ω:

```
matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1, M, 1 );
```

Compute D * V * ω using function matrix_multi( )

S is used in Algorithm 5 and is then computed by multiplying the column sum of each column with the corresponding element in vector D * V * ω:

```
vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);
ls = vector_sum2(m_stemp , 0 , M-1);
```

Computing S for Algorithm 5 in step 1

$\tilde{\omega}_i$, $\hat{\omega}_i$ and their corresponding $\sum_{i=1}^{M} \tilde{\omega}_i (L^i_{lp} - L^i_{sp})$, $\sum_{i=1}^{M} \hat{\omega}_i (L^i_{lm} - L^i_{sm})$ are computed using the following code:

```
1.  for(int i=0; i<M ; i++)
2.  {
```

```
3.      //compute w_lomda
4.      if(*(m_dvw+i) >=0)
5.      {
5.          w_londa = *(m_dvw+i);
6.      }
7.      else
8.      {
9.          w_londa = 0.0;
10.     }
11.
12.     w_arrow = w_londa - *(m_dvw+i);
13.
14.     a1 += w_londa * ( ljlp[i] - ljsp[i] );
15.
15.     b1 += w_arrow * ( ljlm[i] - ljsm[i] );
16. }
```

Table 27: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$, $\sum_{i=1}^{M} \tilde{\omega}_i (L^i_{lp} - L^i_{sp})$ and $\sum_{i=1}^{M} \hat{\omega}_i (L^i_{lm} - L^i_{sm})$ in step 1 of Algorithm 10

The upper bound of VaR is then estimated using the following function in Algorithm 5:

$$\hat{V}_R = 1/(2 * p) * \left[S + \sum_{i=1}^{M} \tilde{\omega}_i (L^i_{lp} - L^i_{sp}) - \sum_{i=1}^{M} \hat{\omega}_i (L^i_{lm} - L^i_{sm}) \right]$$

```
vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
```

Compute upper bound of VaR using Algorithm 5

If the upper bound of VaR is smaller than the collateral value V_c, terminate this test and answer No to the VaR decision problem. Otherwise proceed to step3.

In step3 Algorithm 9 is used to compute b = L * ω with large row L_2 norm. Matrix m_larrow contains those h rows of the loss matrix L and is copied into CUDA_mtemp located in GPU memory for later use. The matrix-vector multiplication is done using the CUDA kernel function MatrixVectorMulKernel:

```
MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2 >>>( CUDA_mtemp,
CUDA_portf , CUDA_btemp , TEMP_ROW1 , M);
```

Matrix-vector multiplication is done using CUDA kernel function MatrixVectorMulKernel

The result is a vector with h elements of L * ω. We then compute the number of elements q in the vector that are greater than the collateral value V_c. If q ≥ *p,* there are more than *p* elements greater than V_c and the answer to the VaR decision problem is Yes.

If steps 1 and 3 fail to give an answer to the VaR decision problem, Monte Carlo simulation is applied.

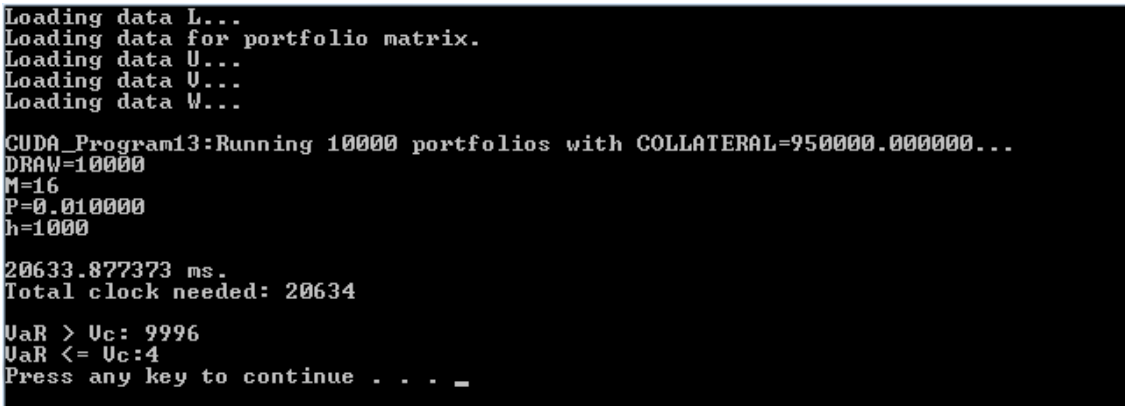

Figure 16: Output for cudaprog13main.cu

For the complete code of test 13, see cudaprog13main.cu in section 9.

Test 14
Algorithm 10: A composite algorithm for the VaR decision problem (2).

In test 14 all of the steps are the same as in test 13, except that L, ω used in Algorithm 9 are replaced with U, D * V *ω in step 3. Parameter h used in step 3 is set to be 1000.

In the initialization stage four variables L_{lp}^{j} , L_{sp}^{j} , L_{lm}^{j} , L_{sm}^{j} and the summation for each column of U are computed:

```
1.  for(int i=0; i<M ; i++)
2.  {
3.      //copy each column of U into m_pv, we only need the first M
column of U
4.      matrix_col_cpy( m_u, m_pv, DRAW, M , i);
5.      quicksort(m_pv, 0, DRAW-1);  //quicksort
6.      //compute the sum of each column of U
7.      mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);
8.
9.      //sum of the p largest components of column j of L
10.     ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
11.     //sum of the N-p smallest components of column j of L
12.     ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
13.     //sum of the p smallest components of column j of L
14.     ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
15.     //sum of the N-p largest components of column j of L
16.     ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
17. }
```

Table 28: Code for computing L_{lp}^{j} , L_{sp}^{j} , L_{lm}^{j} , L_{sm}^{j} and summation for each column of U in Test 14

Matrix multiplication D * V is done by calling the CUDA kernel function MatrixMulKernelv3:

```
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb ,
CUDA_cc , M , M);
```

Compute D * V using CUDA kernel function MatrixMulKernelv3

The h rows of U with large row L_2 norm are selected and moved to GPU memory space using the CUDA cudaMemcpy function:

```
1.  for(int i=0 ; i<DRAW ; i++)
2.  {
3.      //First column contains the order number of each row
4.      *(m_ul2norm + 2*i) = (float)i;
5.      //The second column containsthe L2 norm of each row
6.      //only compute L2 norm for the first M elements in a row
7.      *(m_ul2norm + (2*i+1)) = matrix_row_l2normv2(m_u , DRAW, M ,
0 , M-1 , i);
8.  }
9.
10. //quicksort in descendign order
11. quicksort2(m_ul2norm , 0, DRAW-1);
12.
13. //copy the first h rows of U based on its L2 norm
14. matrix_cpy2v2( m_u , m_ul2norm , m_uarrow, DRAW, M, 0, h-1 , 0, M-
```

```
1 );
15.
16. //move needed data from CPU to GPU
17. cudaMemcpy( CUDA_uarrow_temp , m_uarrow , size_CUDA_uarrow_temp ,
cudaMemcpyHostToDevice );
```

Table 29: Code for selecting h rows of U with large L_2 norm in Test 14

The loss matrix L is also loaded into GPU memory for Monte Carlo simulation during the initialization stage:

```
cudaMemcpy( CUDA_m , m_loss , size_CUDA_m , cudaMemcpyHostToDevice );
```

Copy data from CPU memory to GPU memory

In the computation stage, we start by fetching a row of portfolio data from matrix m_portf using protf_data_cpy function:

```
portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);
```

Fetching portfolio data from matrix m_portf into portfolio

Vector D * V * ω and the corresponding S for this portfolio is computed using the same process used in test 13:

```
//compute D*V*w
matrix_multi( dv_temp , portfolio , m_dvw , M, M, M, 1, M, 1 );
//compute S:
vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);
ls = vector_sum2(m_stemp , 0 , M-1);
```

Compute D * V *ω and S needed for Algorithm 5

$\tilde{\omega}_i$, $\hat{\omega}_i$ and the corresponding $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$, $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ are computed using the following code:

```
1.  //compute w_londa
2.  for(int i=0; i<M ; i++)
3.  {
4.      //compute w_lomda
5.      if(*(m_dvw+i) >=0)
6.      {
7.          w_londa = *(m_dvw+i);
8.      }
9.      else
10.     {
11.         w_londa = 0.0;
12.     }
```

```
13.
14.     //compute w_arrow
15.     w_arrow = w_londa - *(m_dvw+i);
16.     a1 += w_londa * ( ljlp[i] - ljsp[i] );
17.     b1 += w_arrow * ( ljlm[i] - ljsm[i] );
18. }
```

Table 30: Code for computing $\tilde{\omega}_i$, $\hat{\omega}_i$ and the corresponding $\sum_{i=1}^{M} \tilde{\omega}_i (L_{lp}^i - L_{sp}^i)$, $\sum_{i=1}^{M} \hat{\omega}_i (L_{lm}^i - L_{sm}^i)$ used in step 1 of Algorithm 10

The upper bound of VaR is computed using the same function as in test 13:

```
vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
```

Compute upper bound of VaR using Algorithm 5

If the upper bound of VaR is smaller than the collateral value V_c, terminate this test with an answer of No to the VaR decision problem. Otherwise proceed to step 3.

In step 3 we compute b = U * D * V * ω for those h rows of matrix U with large row L_2 norms. Matrix m_uarrow contains the necessary data and is copied into CUDA_uarrow_temp located in GPU memory during the initialization stage. The matrix-vector multiplication is done using the CUDA kernel function MatrixVectorMulKernel:

```
MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2
>>>( CUDA_uarrow_temp , CUDA_dvw, CUDA_btemp , TEMP_AREA , M);
```

Compute U * D * V * ω using CUDA kernel function MatrixVectorMulKernel

We then search for the number of elements in b that exceed our collateral V_c. If this number q is greater than *p,* terminate this test and answer Yes to the VaR decision problem.

If steps 1 and 3 fail to give an answer to the VaR decision problem, Algorithm 1 Monte Carlo simulation is applied.

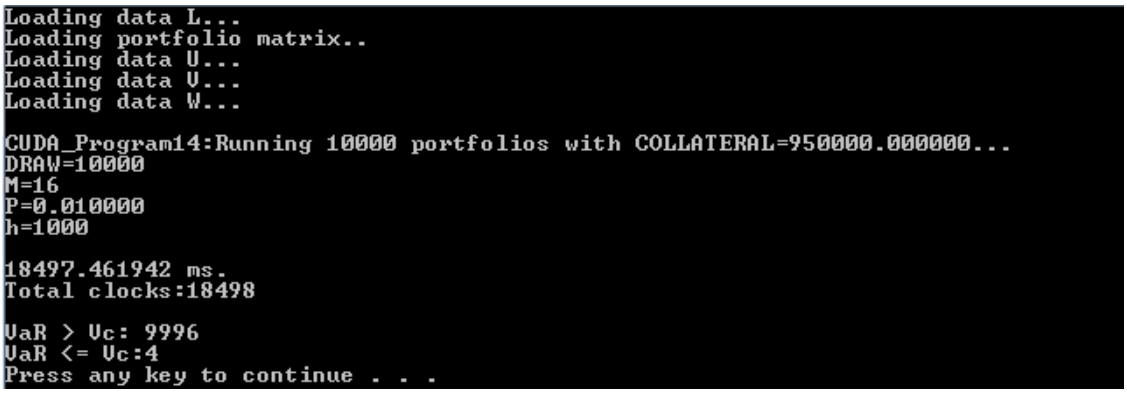

Figure 17: Output for cudaprog14main.cu

For the complete code of test 14, see cudaprog14main.cu in section 9.

6. Conclusions

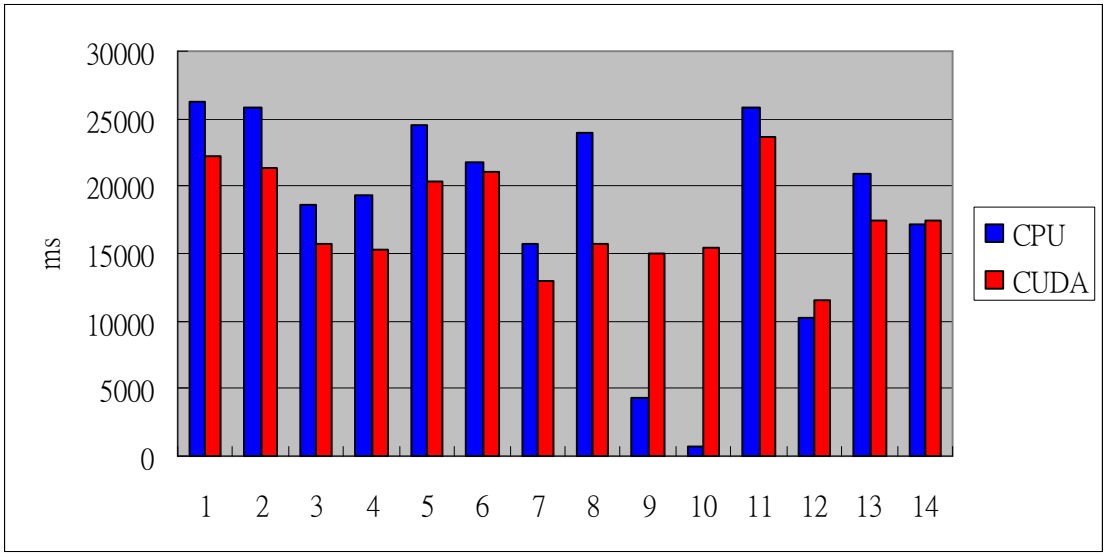

Figure 18: Performance comparison for different tests with a loss matrix of size 10,000 x 5. The collateral is set to be 200,000 in each test

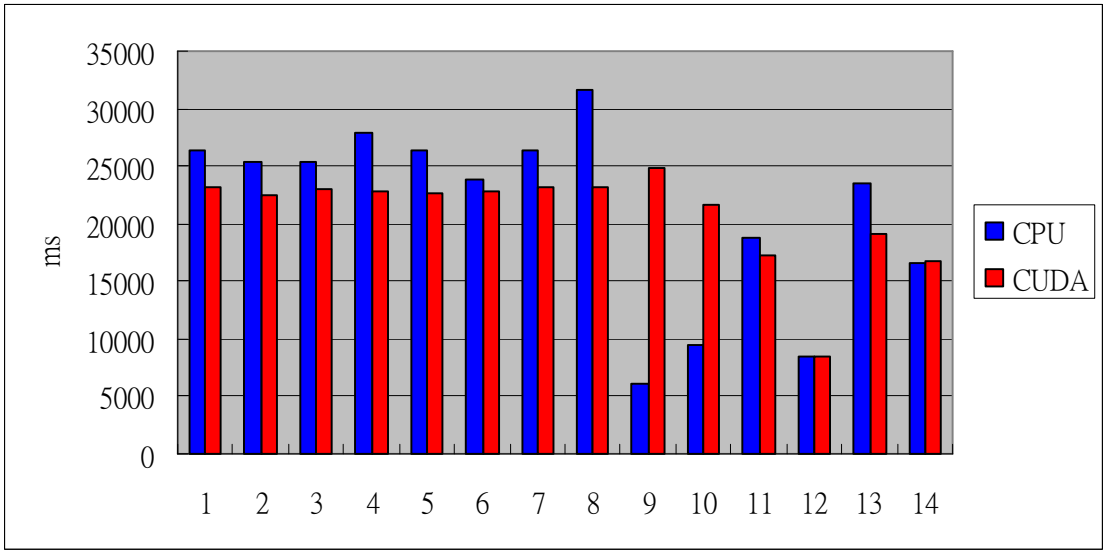

Figure 19: Performance comparison for different tests with a loss matrix of size 10,000 x 5. The collateral is set to be 60,000 in each test

Figures 18 and 19 demonstrate the performance for each test in milliseconds with a loss matrix size of 10,000 x 5 and a portfolio vector of size 5 x 1. The level of confidence is set to 99%. The variable h used in Algorithm 9 is set to be 1000. Collateral V_c is set to be 200,000 in Figure 18 and 60,000 in Figure 19.

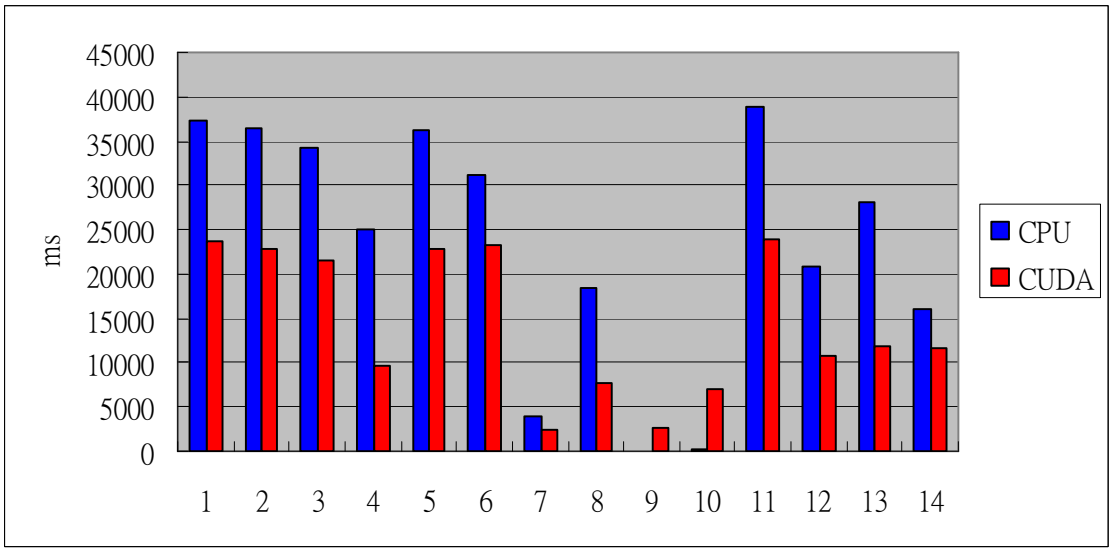

Figure 20: Performance comparison for different tests with a loss matrix of size 10,000 x 16. The collateral is set to be 5500,000 in each test

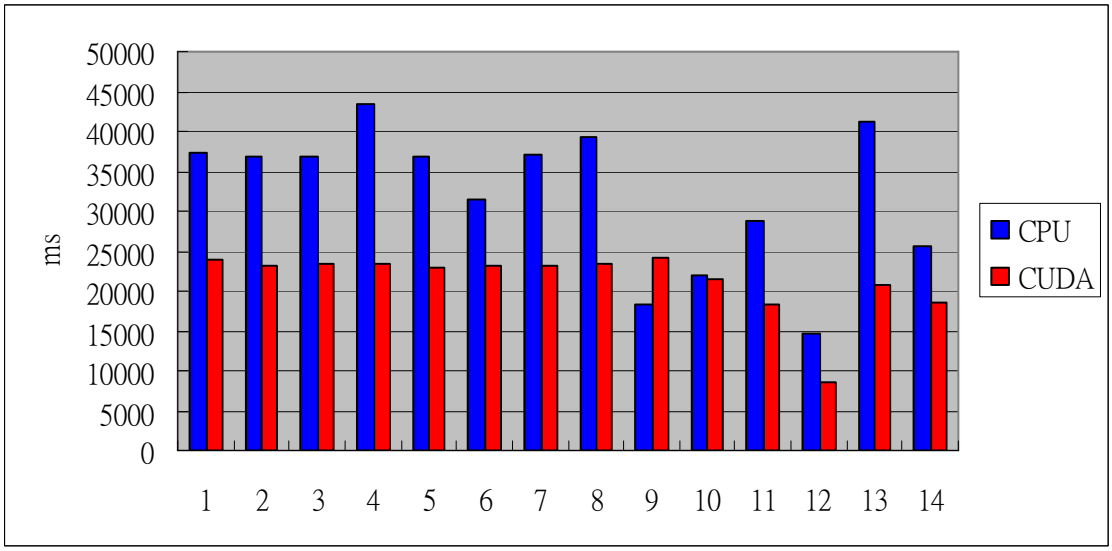

Figure 21: Performance comparison for different tests with a loss matrix of size 10,000 x 16. The collateral is set to be 950,000 in each test

Figures 20 and 21 demonstrate the performance for each test in milliseconds with a loss matrix size of 10,000 x 16 and a portfolio vector of size 16 x 1. Collateral V_c is set to be 5500,000 in Figure 20 and 950,000 in Figure 21.

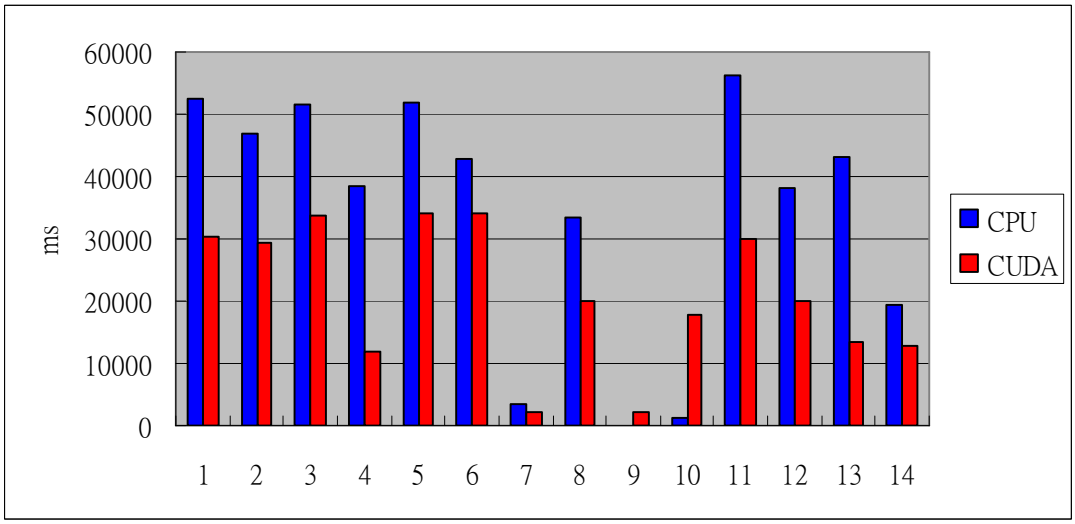

Figure 22: Performance comparison for different tests with a loss matrix of size 10,000 x 32. The collateral is set to be 10,000,000 in each test

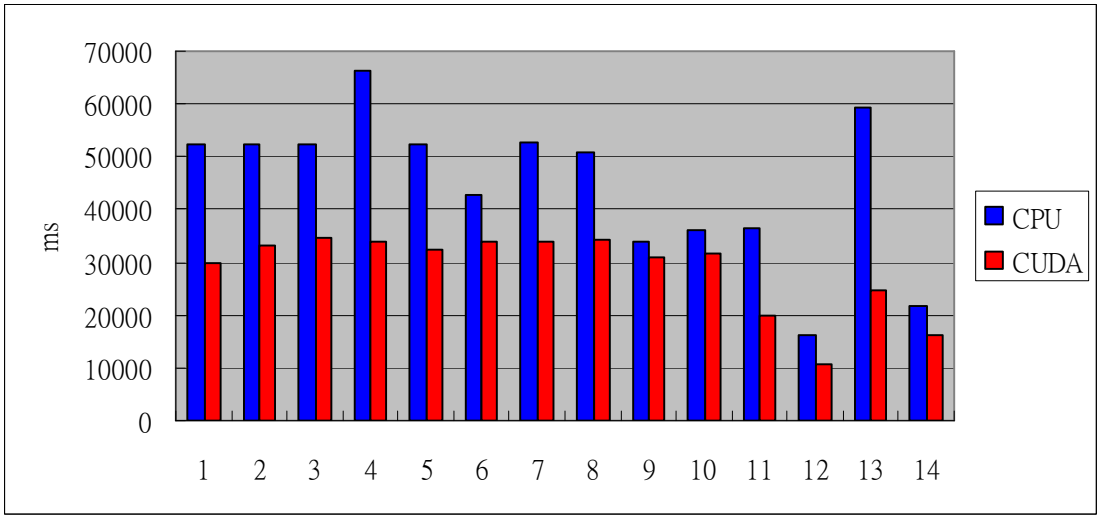

Figure 23: Performance comparison for different tests with a loss matrix of size 10,000 x 32. The collateral is set to be 850,000 in each test

Figures 22 and 23 demonstrate the performance for each test in milliseconds with a loss matrix size of 10,000 x 32 and a portfolio vector of size 32 x 1. Collateral V_c is set to be 10,000,000 in Figure 22 and 850,000 in Figure 23. The computation time needed for matrix-vector multiplication in Monte Carlo simulation grows as the number of elements in the portfolio grows. Since more column data related to the asset in the portfolio will be added into the loss matrix L, the size of L grows as the number of assets in the portfolio increases.

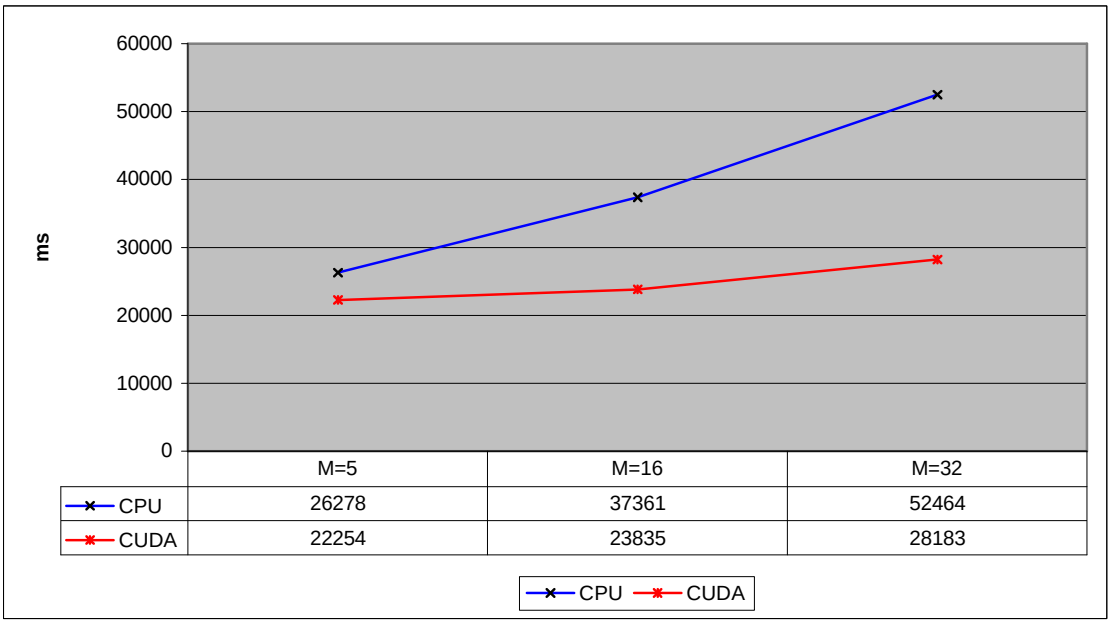

Figure 24: Performance comparison for different size of loss matrix. The column number M is set to 5, 16 and 32

Compare the performance of test 1, which relies only on Monte Carlo simulation, as an example. The computation time grows roughly 40% when the number of elements in the portfolio M increases from 5 to 16 and when M increases to 32, the computation time almost doubles**.** The computation time needed using CUDA increases only by 7% when M increases to 16, and by 26% when M increases to 32, as shown in figure 24. With the help of CUDA, we can take more advantage as the number of assets in a portfolio increases with demand.

7. Bibliography

Reference

[1] *Improved techniques for using Monte Carlo in VaR estimation*, available at http://www.cs.fsu.edu/~asriniva/papers/nsefinal.pdf

[2] Thomas J. Linsmeier and Neil D. Person *, *Risk management: An Introduction to Value at risk*, University of Illinois at Urbana-champaign, 1996, available at: http://www.exinfm.com/training/pdfiles/valueatrisk.pdf

[3] David Kirk and Wen-mei Hwu, *Programming Massively Parallel Processors*, 2010

[4] Nvidia CUDA C Programming Guide available at Nvidia official website at: http://developer.nvidia.com/category/zone/cuda-zone

8. Appendix

8.1 L_1, L_2 and L_∞ norms for matrix

In linear algebra, a norm is a function that assigns a positive length or size to all vectors in a vector space. A matrix norm is an extension of the notion of a vector norm to matrices. The following are the definitions of L_1, L_2 and L_∞ norm used in our algorithm:

For a vector x:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ M \\ x_n \end{bmatrix}$$

The L_1 norm is defined as the summation of the vector and we take its absolute value:

$$|x|_1 = \sum_{i=1}^{n} |x_i|$$

The L_2 norm is defined as the length of a vector. We compute the summation of the square of each element and take its square root.

$$|x|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

So for a vector x = $[x_1 \quad x_2 \quad x_3]$, the L_2 norm of vector x will be:

$$|x|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

The L_∞ norm is defined as the largest element in a vector.

$$|x|_\infty = \max_i x_i$$

8.2 Singular value decomposition (SVD)

Singular value decomposition will decompose an N x M matrix into three matrices:

M = U * D * V^T

Where U is an N x N unitary matrix, D is an N x M diagonal matrix with nonnegative real number on the diagonal and V^T (the conjugate of the transpose of V) is an M x M unitary matrix.

The following code opencvsvdmain.c uses the OpenCV library and its function cvSVD( ) to compute singular value decomposition of a loss matrix. This program is able to read in an N x M loss matrix (in this example: loss16.txt) and generate the corresponding U, D and V^T matrices (matrixw16.txt, matrixv16.txt and matrixv16.txt).

opencvsvdmain.c

```
1.  //opcvtest1.cpp :This code use the library of OpenCV to do SVD
2.  //Input: Loss matrx
3.  //Output: Matrix U, matrix D and V'
4.  #include "stdafx.h"
5.
6.  #include <cv.h> //OpenCV header
7.  #include <cxcore.h> //OpenCV header
8.  #include <highgui.h> //OpenCV header
9.  #include <stdio.h>
10. //Define the row number of Loss matrix
11. #ifndef DRAW
12. #define DRAW 10000
13. #endif
14. //Define the column number of Loss matrix
15. #ifndef M
16. #define M 16
17. #endif
18. //function to display matrix
19. void PrintMatrix(CvMat *Matrix,int Rows,int Cols);
20. //function to output matrix into file
21. void write_matrix(CvMat *Matrix, int row , int col , FILE* fp);
```

```
22.
23. double Array1[DRAW][M];
24. int main()
25. {
26.     //cvCreateMat allocate a space for matrix
27.     CvMat *Matrix1=cvCreateMat( DRAW, M,CV_64FC1);
28.     CvMat *W=cvCreateMat( DRAW, M,CV_64FC1);
29.     CvMat *V=cvCreateMat( M, M,CV_64FC1);
30.     CvMat *U=cvCreateMat( DRAW, DRAW,CV_64FC1);
31.
32.     CvMat *V_T=cvCreateMat( M , M,CV_64FC1);
33.     CvMat *ResultMatrix=cvCreateMat( DRAW, M,CV_64FC1);
34.
35.     float elen;
36.     int i,j;
37.     FILE* fp;
38.     FILE* fp_w;
39.     FILE* fp_v;
40.     FILE* fp_u;
41.     FILE* fp_ch;
42.     //File pointer points to loss matrix
43.     fp = fopen("G:\\loss16.txt", "r" );
44.     fp_w = fopen("G:\\matrixw16.txt", "w");
45.     fp_v = fopen("G:\\matrixv16.txt", "w");
46.     fp_u = fopen("G:\\matrixu16.txt", "w");
47.     fp_ch = fopen("d:\\matrix_loss.txt", "w");
48.
49.     if(fp == NULL || fp_w == NULL || fp_v == NULL || fp_u == NULL)
50.     {
51.         printf("Warning! Reading file error!!\n");
52.     }
53.     //Readin loss matrix data from file
54.     printf("Reading data into Array!!\n\n");
55.     for(i=0 ; i<DRAW; i++)
56.     {
57.         for(j=0; j<M; j++)
58.         {
59.            fscanf(fp, "%f ", &elen);
60.            Array1[i][j] = elen;
61.         }
62.     }
63.     printf("Data loading complete!\n\n");
64.
65.     //
66.     cvSetData(Matrix1, Array1 ,Matrix1->step);  //
67.
68.     //singular value decomposition, this may takes several minutes
69.     printf("Computing SVD, please wait...\n\n");
70.     cvSVD( Matrix1, W , U, V , 1);
71.
72.     printf("\nW:(DRAW x M )\n");
73.     //output matrix W to file
74.     write_matrix(W, W->rows , W->cols, fp_w);
75.     fclose(fp_w);
76.
77.     printf("\nU (DRAW x DRAW)\n");
78.     //output matrix U to file
```

```
79.     write_matrix(U, U->rows, U->cols, fp_u);
80.     fclose(fp_u);
81.
82.     printf("\nV_T (M x M)\n");
83.     //conjugate transpose of V
84.     cvTranspose(V,V_T);
85.     //output matrix V'
86.     write_matrix( V_T, V->rows, V->cols, fp_v);
87.     fclose(fp_v);
88.
89.     system("pause");
90.
91. }
92. //function to display matrix
93. void PrintMatrix(CvMat *Matrix,int Rows,int Cols)
94. {
95.    int i,j;
96.    for( i=0;i<Rows;i++)
97.    {
98.        for( j=0;j<Cols;j++)
99.        {
100.            //cvGet2D() get the value for Matrix
101.            printf("%.2f ",cvGet2D(Matrix,i,j).val[0]);
102.        }
103.        printf("\n");
104.    }
105. }
106. //function to output matrix into file
107. void write_matrix(CvMat *Matrix, int row , int col , FILE* fp)
108. {
109.    int i,j;
110.    for( i=0; i<row; i++)
111.    {
112.        for( j=0; j<col; j++)
113.        {
114.            fprintf(fp , "%f ", cvGet2D(Matrix,i,j).val[0]);
115.        }
116.        fprintf(fp, "\n");
117.    }
118. }
```

9. Source code

cudaprog1main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <time.h>
#include <math.h>
#include <windows.h>
#include "matrix_ope.h"

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

extern void ini_fmatrix(float* m, int t_row, int t_col);
extern void loaddata(FILE* fp, float* target , int t_row, int t_col);
extern void show_2d_matrix(float* m, int row , int col);

void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
void vector_move(float *a, float *b , int ele_num , int start_e);
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) Kernel for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;
```

```
	int Row = by * TILE_WIDTHy + ty;
	//int Col = bx * TILE_WIDTHx + tx; //size change

	float pvalue=0;

	if(ty==0) time[by]=clock();

	for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
	{
	  Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
	  Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
	  __syncthreads();

	  for(int j=0 ; j < TILE_WIDTHx ; j++)
	  {
		pvalue += Mds[ty][j] * Nds[j];
	  }

	  *(pd + Row) = pvalue;
	}

	if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy] = clock();

}

int main()
{
	FILE *fp_loss;  //file pointer points loss matrix
	FILE *fp_portf_m;

	float *loss_m;
	float *portfolio;
	float *portf_m;
	float *b;
	float *data_temp;
	float *b_temp;

	float *md; //pointer for GPU space
	float *nd; //pointer for GPU space
	float *pd; //pointer for GPU space

	float var;
	float ls=0.0;
	float lw=0.0;

	int vr_index;
	//md stores loss matrix data in GPU
	int size_mdgpuspace = TEMP_AREA * M * sizeof(float);
	//nd stores weight vector data in GPU
	int size_ndgpuspace = M * sizeof(float);
	//pd stores the result for matrix-vector multiplication
	int size_pdgpuspace = TEMP_AREA * sizeof(float);
	int size_CUDA_time = sizeof(clock_t) * (TEMP_AREA/TILE_WIDTHy)*2;
	//size of the array depending on how many blocks

	int time;
	int to_buy=0;
```

```
      int not_to_buy=0;

      LARGE_INTEGER frequency;
      LARGE_INTEGER t1, t2;
      double elapsedTime;

      clock_t clock_start;
      clock_t *CUDA_time;  //used for clock computation in CUDA Kernel
      clock_t *time_used;
      clock_t avg_clk=0;

      fp_loss = fopen("d:\\loss16.txt", "r");
      fp_portf_m = fopen("d:\\portfoliom16.txt" , "r");

      //
      loss_m = new float [DRAW*M];
      portfolio = new float [M];
      b = new float [DRAW];
      data_temp = new float [TEMP_AREA*M];
      b_temp = new float [TEMP_AREA];
      portf_m = new float [DRAW * M];
      time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy) * 2];

      vr_index = DRAW - (int)((float)DRAW * P);

      //load loss matrix and weight vector data from file
      loaddata(fp_loss , loss_m, DRAW ,M);
      loaddata(fp_portf_m , portf_m , PORTF_NUM , M);

      //define threads per block
      dim3 threadsperBlock( TILE_WIDTHx , TILE_WIDTHy );
      //define how many blocks are used
      dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy );

      //declare space in GPU
      cudaMalloc((void**) &md, size_mdgpuspace);
      cudaMalloc((void**) &nd, size_ndgpuspace);
      cudaMalloc((void**) &pd, size_pdgpuspace);
      //this space is used in CUDA Kernel to record the clocks needed
      //to finish computation
      cudaMalloc((void**) &CUDA_time , size_CUDA_time);

     //copy loss matrix data from CPU to GPU
      cudaMemcpy( md , loss_m ,size_mdgpuspace ,
cudaMemcpyHostToDevice);

      printf("Monte Carlo simulation CUDA Computing for %d
      portfolios...\n", PORTF_NUM);
      printf("DRAW=%d\n", DRAW);
      printf("M=%d\n", M);
      printf("P=%f\n", P);
      printf("COLLATERAL=%f\n", COLLATERAL);

      QueryPerformanceFrequency(&frequency);

      clock_start = clock();
      QueryPerformanceCounter(&t1);
```

```
      for(int k=0 ; k<PORTF_NUM ; k++)
      {
            portf_data_cpy( portf_m , portfolio , PORTF_NUM , M , k);
            //copy weitht vector data from CPU to GPU
            cudaMemcpy(nd , portfolio , size_ndgpuspace ,
cudaMemcpyHostToDevice );

            for(int i=0; i< DRAW/TEMP_AREA ; i++)
            {
               //Invoke CUDA Kernel function to compute matrix-vector
               //multiplication
               MatrixVectorMulKernel<<< numBlocks , threadsperBlock
>>>( md , nd , pd , TEMP_AREA , M , CUDA_time);

               //Copy the result form GPU to CPU
               cudaMemcpy( b_temp , pd , size_pdgpuspace ,
cudaMemcpyDeviceToHost);
               cudaMemcpy( time_used , CUDA_time , size_CUDA_time,
cudaMemcpyDeviceToHost);

               vector_move( b_temp , b , TEMP_AREA , i* TEMP_AREA);
            }

             //quicksort in ascending order
             quicksort(b , 0, DRAW-1);

             var = b[vr_index];

             //printf("vr_index: %d\n", vr_index);
             //printf("var: %f\n", var);
             if(var > COLLATERAL)
             {
                   //The answer to the decision problem is YES!
                   not_to_buy++;
             }
             else
             {
                   //The answer to the decision problem is NO.
                   to_buy++;
             }


      }
      QueryPerformanceCounter(&t2);
      time = clock() - clock_start;

      elapsedTime = (t2.QuadPart -
t1.QuadPart)*1000.0/frequency.QuadPart;

      printf("\n%f ms.\n\n" , elapsedTime);


      printf("VaR > Vc: %d\n", not_to_buy);
      printf("VaR <= Vc: %d\n", to_buy);
```

```
	//free space
	delete[] loss_m;
	delete[] portfolio;
	delete [] portf_m;
	delete[] b;

	delete[] data_temp;

	delete[] b_temp;

	delete [] time_used;

	cudaFree(md);
	cudaFree(nd);
	cudaFree(pd);
	cudaFree(CUDA_time);

	return 0;
}

//copy matrix from matrix a to matrix b
void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
	int i,j;
	for(i=0 ; i< m_row ; i++)
	{
		for(j=0 ; j< m_col ; j++)
		{
		    *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
		}
	}

}

//copy vector from vector a to vector b
void vector_move(float *a, float *b , int ele_num , int start_e)
{
	int i;
	for(i=0; i<ele_num ; i++)
	{
		*(b+start_e+i) = *(a+i);
	}
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
	int i;

	for(i=0 ; i<a_col ; i++)
	{
	   *(b + i) = *(a + a_row2 * a_col + i);
	}

}
```

cudaprog2main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <time.h>
#include <windows.h>
#include "matrix_ope.h"
#include <math.h>

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

//if modify this also check kernel MatrixVectorMulKernel2!!
#ifndef TILE_WIDTHx3
#define TILE_WIDTHx3 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif
//if modify this also check kernel MatrixVectorMulKernel2!!
#ifndef TILE_WIDTHy3
#define TILE_WIDTHy3 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float* m, int t_row, int t_col);
extern void loaddata(FILE* fp, float* target , int t_row, int t_col);
extern void show_2d_matrix(float* m, int row , int col);
extern void show_vect(float* v , int ele_n);
extern float vector_sum(float* a, int ele_a);
extern void vector_ele_multi(float* a, float* b, float* ans, int
ele_a);
extern void matrix_sum_of_col(float* target, float* s , int
target_row , int target_col);
extern void matrix_multi(float* m , float* n , float* ans, int m_row ,
int m_col , int n_row , int n_col ,int ans_row , int ans_col);

void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
void vector_move(float *a, float *b , int ele_num , int start_e);
void vector_ele_multiv2(float* a, float* b, int ele_a , int ele_b);
float vector_power2_sum( float *a, int ele_a);
```

```
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) Kernel for matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int
m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< m_col/TILE_WIDTHx ; i++)
   {
         //loading data into shared memory
         Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
         Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
         __syncthreads();

         for(int j=0 ; j<TILE_WIDTHx ; j++)
         {
               pvalue += Mds[ty][j] * Nds[j][tx];
               __syncthreads();
         }

         *(pd + Row * m_col + Col) = pvalue;
   }

}

//(CUDA) Kernel for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
      //int Col = bx * TILE_WIDTHx + tx; //size change

      float pvalue=0;

      if(tx==0) time[by]=clock();
```

```
	for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
	{
	  Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
	  Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
	  __syncthreads();

	  for(int j=0 ; j < TILE_WIDTHx ; j++)
	  {
		pvalue += Mds[ty][j] * Nds[j];
	  }

	  *(pd + Row) = pvalue;
	}

	if(tx==0) time[by + TEMP_AREA/TILE_WIDTHy] = clock();

}

//(CUDA) Kernel for matrix-vector multiplication (different shared
//memory size)
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
	__shared__ float Mds[TILE_WIDTHy3][TILE_WIDTHx3];
	__shared__ float Nds[TILE_WIDTHx3];

	//int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;

	int Row = by * TILE_WIDTHy3 + ty;
	//int Col = bx * TILE_WIDTHx + tx;
	if(ty==0) time[by]=clock();

	float pvalue=0;

	for(int i=0 ; i< (m_col/TILE_WIDTHx3) ; i++)
	{
	  Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx3 + tx));
	  Nds[tx] = *(nd + (i* TILE_WIDTHx3 + tx));
	  __syncthreads();

	  for(int j=0 ; j < TILE_WIDTHx3 ; j++)
	  {
		pvalue += Mds[ty][j] * Nds[j];
	  }

	  *(pd + Row) = pvalue;
	}

	if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy3]=clock();

}

//(CUDA) Kernel for computing the column sum of each column in the
//matrix
```

```
__global__ void CUDA_matrix_col_sum(float *a, float *b, int m_row , int
m_col)
{
	__shared__ float partialsum[TILE_WIDTHy3][TILE_WIDTHx];


	int bx = blockIdx.x;
	int by = blockIdx.y;

	int tx = threadIdx.x;
	int ty = threadIdx.y;

	int Row = by * TILE_WIDTHy3 + ty;
	int Col = bx * TILE_WIDTHx + tx;

	partialsum[ty][tx] = *(a+Row*m_col+Col);
	__syncthreads(); //

	for(int i=1 ; i < TILE_WIDTHy3 ; i *= 2)
	{
		__syncthreads();

		if( (ty % (2*i) ==0) )
		{
			//must make sure matrix index not exceed it's
			//maximum!!!
			if(ty+i < TILE_WIDTHy3)
			{
			 partialsum[ty][tx] += partialsum[ty+i][tx];
			}
			else
			{
			}
			__syncthreads();
		}

	}
	__syncthreads();

	*(b + by * m_col + Col) = partialsum[0][tx]; //

}
int main()
{
	FILE *fp_in;
	FILE *fp_md;
	FILE *fp_mv;
	FILE *fp_mportf;

	float *m_loss;
	float *m_d;
	float *m_v;
	//this pointer points to the space of data for portfolio matrix
	float *m_portf;
	//this pointer points to the space of data for single portfolio
	float *m_portfolio;
	float *data_temp;
```

```
	float *matrix_col_sum;
	float *block_col_sum;
	float *m_dvw;
	float *dv_temp;
	float *m_b;
	float *m_temp1;

	float sp=0;

	float *CUDA_mloss; //GPU space for loss matrix
	float *nd; //GPU space used by CUDA_matrix_col_sum
	float *CUDA_md;  //GPU space for D matrix
	float *CUDA_mv;  //GPU space for V matrix
	float *CUDA_dv;  //GPU space for D*V used by MatrixMulKernelv3
	float *CUDA_portf; //GPU space for weight vector in
//MatrixVectorMulKernel
	float *CUDA_dvw;  //GPU space for DVw used by
//MatrixVectorMulKernel
	float *CUDA_mb;

	float vr_upperbound=0;
	float ls=0;
	float lw=0;
	float var;

	//
	int size_CUDA_mloss =  TEMP_AREA * M * sizeof(float);
	int size_ndgpuspace =  TEMP_AREA/TILE_WIDTHy3 * M *
sizeof(float);
	int size_CUDA_md_space = M * M * sizeof(float);
	int size_CUDA_mv_space = M * M * sizeof(float);
	int size_CUDA_dv_space = M * M * sizeof(float);
	int size_CUDA_portf_space = M * sizeof(float);
	int size_CUDA_mb = TEMP_AREA * sizeof(float);
	int size_CUDA_time=sizeof(clock_t)*(TEMP_AREA/TILE_WIDTHy3) * 2;

	int to_buy=0;
	int not_to_buy=0;
	int mt_count=0;

	LARGE_INTEGER frequency;
	LARGE_INTEGER t1, t2;
	double elapsedTime;

	clock_t s_time;
	clock_t total_time;
	clock_t* CUDA_time;
	clock_t* time_used;
	time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy3) * 2];
	clock_t avg_clk=0;

	//declare space
	m_loss = new float [DRAW * M];
	m_d = new float [M * M];
	m_v = new float [M * M];
	m_portf = new float [PORTF_NUM * M];
	m_portfolio = new float [M];
```

```
      data_temp = new float [TEMP_AREA * M];
      matrix_col_sum = new float [M];
      ini_fmatrix( matrix_col_sum , 1,M);
      block_col_sum = new float [ TEMP_AREA/TILE_WIDTHy * M ];
      m_dvw = new float [M];
      dv_temp = new float [M*M];
      m_b = new float [TEMP_AREA];
      m_temp1 = new float [M];

      //
      fp_in = fopen("d:\\loss16.txt", "r");
      fp_md = fopen("d:\\matrixw16.txt" , "r");
      fp_mv = fopen("d:\\matrixv16.txt" , "r");
      fp_mportf = fopen("d:\\portfoliom16.txt" , "r");

      //loading data
      loaddata(fp_in , m_loss , DRAW, M);
      loaddata(fp_md , m_d , M , M);
      loaddata(fp_mv , m_v , M , M);
      loaddata(fp_mportf , m_portf , PORTF_NUM , M);
      //show_2d_matrix(m_loss , 10, M);

      dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy3 );
      dim3 threadsperblock2( TILE_WIDTHx , TILE_WIDTHy );   //compute
//D*V
      dim3 threadsperblock3( TILE_WIDTHx3 , TILE_WIDTHy3 );  //for
//Monte Carlo simulation

      dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy3 );
      dim3 numBlocks2( M/TILE_WIDTHx , M/TILE_WIDTHy );  //compute D*V
      dim3 numBlocks3( M/TILE_WIDTHx3 , DRAW/TILE_WIDTHy3 ); //for
//Monte Carlo simulation

      sp = (float)DRAW * P;
      printf("CUDA computing for VaR with %d portfolios.\n\n",
PORTF_NUM);
      //===========Initialization====================
      //declare space for GPU
      cudaMalloc((void**) &CUDA_mloss , size_CUDA_mloss);
      cudaMalloc((void**) &nd , size_ndgpuspace);
      cudaMalloc((void**) &CUDA_mb , size_CUDA_mb);
      cudaMalloc((void**) &CUDA_time, size_CUDA_time);

      //compute column sum for each column of loss matrix
      for(int i=0 ; i< DRAW/TEMP_AREA ; i++)
      {
            matrix_move(m_loss , data_temp, TEMP_AREA , M ,
i*TEMP_AREA );
            //copy loss matrix data from CPU to GPU
            cudaMemcpy(CUDA_mloss , data_temp , size_CUDA_mloss ,
cudaMemcpyHostToDevice);
            //Invoke CUDA Kernel
            CUDA_matrix_col_sum<<< numBlocks , threadsperblock
>>>( CUDA_mloss , nd , TEMP_AREA , M );
            //copy blocks of result from GPU to CPU
            cudaMemcpy( block_col_sum , nd , size_ndgpuspace ,
cudaMemcpyDeviceToHost);
```

```
            //compute the result from the blocks and store in
            //matrix_col_sum
            matrix_sum_of_col( block_col_sum , matrix_col_sum ,
TEMP_AREA/TILE_WIDTHy3, M );
      }

      //release gpu space
      cudaFree(nd);

      //precompute D*V
      cudaMalloc((void**) &CUDA_md , size_CUDA_md_space);
      cudaMalloc((void**) &CUDA_mv , size_CUDA_mv_space);
      cudaMalloc((void**) &CUDA_dv , size_CUDA_dv_space);

      //copy D and V matrix data to GPU
      cudaMemcpy( CUDA_md , m_d , size_CUDA_md_space ,
cudaMemcpyHostToDevice );
      cudaMemcpy( CUDA_mv , m_v , size_CUDA_mv_space ,
cudaMemcpyHostToDevice );

      //compute D*V using CUDA core MatrixMulKernelv3
      MatrixMulKernelv3<<< numBlocks2 , threadsperblock2 >>>(CUDA_md ,
CUDA_mv , CUDA_dv , M , M);
      cudaMemcpy( dv_temp , CUDA_dv , size_CUDA_dv_space ,
cudaMemcpyDeviceToHost);

      //=========================================================
      cudaMalloc((void**) &CUDA_portf , size_CUDA_portf_space);
      cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf_space);

      cudaMemcpy( CUDA_dv , dv_temp , size_CUDA_dv_space ,
cudaMemcpyHostToDevice);

      printf("CUDA_Program2:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);

      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0 ; k<PORTF_NUM ; k++)
      {
          vr_upperbound = 0;

          portf_data_cpy(m_portf , m_portfolio, PORTF_NUM , M , k);
          //compute D*V*w
          matrix_multi( dv_temp , m_portfolio , m_dvw , M, M, M,1,
M,1);
          //compute W
          lw = vector_power2_sum(m_dvw , M);

          //compte S
```

```
          vector_ele_multi( matrix_col_sum , m_portfolio, m_temp1, M );

          ls = vector_sum( m_temp1, M);

          //compute upper bound for VR
          vr_upperbound = (1/(float)DRAW) * (ls + sqrt( ((DRAW-sp)/sp)
* (DRAW * lw - pow(ls,2)) ));
          //printf("vr_upperbound: %f\n", vr_upperbound);

          //if vr_upperbound <= Vc then the answer to the decision
          //problem is No...
            if( vr_upperbound <= COLLATERAL )
            {

                    to_buy++;
            }
            else
            {
                    mt_count++;
                    //printf("\nRun Monte Carlo simulation!\n");
                    cudaMemcpy( CUDA_portf , m_portfolio ,
size_CUDA_portf_space , cudaMemcpyHostToDevice );
                    //Invoke CUDA core for matrix-vector multiplication
                    MatrixVectorMulKernel2<<< numBlocks3 ,
threadsperblock3 >>>( CUDA_mloss , CUDA_portf , CUDA_mb , TEMP_AREA ,
M , CUDA_time);

                    cudaMemcpy( m_b, CUDA_mb, size_CUDA_mb ,
cudaMemcpyDeviceToHost );
                    cudaMemcpy( time_used , CUDA_time, size_CUDA_time ,
cudaMemcpyDeviceToHost);

                     quicksort( m_b , 0 , DRAW-1);

                     var = m_b[DRAW - (int)sp];

                     if(var > COLLATERAL)
                     {
                          not_to_buy++;
                     }
                     else
                     {
                          to_buy++;
                     }

             }
             if(k==0)
             {
                    for(int i=0 ; i< TEMP_AREA/TILE_WIDTHy3 ; i++)
                    {
                   avg_clk += *(time_used + i + TEMP_AREA/TILE_WIDTHy3)
- *(time_used + i);
                    }
             }

      }
      QueryPerformanceCounter(&t2);
```

```
	total_time = clock() - s_time;

	elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;


	printf("\n%f ms.\n\n", elapsedTime);

	printf("VaR > Vc: %d\n", not_to_buy);
	printf("VaR <= Vc: %d\n", to_buy);


	//free space
	delete [] m_loss;
	delete [] m_d;
	delete [] m_v;
	delete [] m_portf;
	delete [] m_portfolio;
	delete [] data_temp;
	delete [] matrix_col_sum;
	delete [] block_col_sum;
	delete [] m_dvw;
	delete [] dv_temp;
	delete [] m_temp1;
	delete [] time_used;

	cudaFree(CUDA_md);
	cudaFree(CUDA_mv);
	cudaFree(CUDA_dv);
	cudaFree(CUDA_dvw);

	cudaFree(CUDA_mloss);
	cudaFree(CUDA_portf);
	cudaFree(CUDA_mb);
	cudaFree(CUDA_time);


	return 0;
}

void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
	int i,j;
	for(i=0 ; i< m_row ; i++)
	{
		for(j=0 ; j< m_col ; j++)
		{
		    *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
		}
	}

}
void vector_move(float *a, float *b , int ele_num , int start_e)
{
	int i;
	for(i=0; i<ele_num ; i++)
```

```
      {
            *(b+start_e+i) = *(a+i);
      }
}
void vector_ele_multiv2(float* a, float* b, int ele_a , int ele_b)
{
     int i;
     for(i = 0; i< ele_a; i++)
     {
          *(a+i) = *(a+i) * *(b+ (int)(i % ele_b));
     }
}

float vector_power2_sum( float *a, int ele_a)
{
     float total=0.0;

     for(int i=0; i<ele_a ; i++)
     {
         total += pow( *(a+i) , 2);
     }

     return total;
}
//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
      int i;

      for(i=0 ; i<a_col ; i++)
      {
         *(b + i) = *(a + a_row2 * a_col + i);
      }

}
```

cudaprog3main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <windows.h>
#include "matrix_ope.h"

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

//this is used in MatrixVectorMulKernel
#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 20
#endif

//this is used in MatrixVectorMulKernel
#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
extern void matrix_col_cpy(float* target, float* vect, int row_t, int
col_t, int col_i);
extern void matrix_sum_of_col(float* target, float* s , int
target_row , int target_col);
extern float vector_sum2(float* a, int start, int end);
extern void vector_ele_multi(float* a, float* b, float* ans, int
ele_a);
```

```
extern void vector_ele_multiv2(float* a, float* b, int ele_a , int
ele_b);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
      __shared__ float Nds[TILE_WIDTHx2];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy2 + ty;
      //int Col = bx * TILE_WIDTHx2 + tx;

      float pvalue=0;

      if(ty==0) time[by]=clock();

      for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx2 ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

      if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2] = clock();

}

//(CUDA)Core for computing the column sum of each culumn in the matrix
__global__ void CUDA_matrix_col_sum(float *a, float *b, int m_row , int
m_col)
{
      __shared__ float partialsum[TILE_WIDTHy][TILE_WIDTHx];


      int bx = blockIdx.x;
      int by = blockIdx.y;

      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
```

```
	int Col = bx * TILE_WIDTHx + tx;

	//float column_sum=0;

	partialsum[ty][tx] = *(a+Row*m_col+Col);
	__syncthreads(); //

	for(int i=1 ; i < TILE_WIDTHy ; i *= 2)
	{
		__syncthreads();

		if( (ty % (2*i) ==0) )
		{
			//must make sure matrix index not exceed it's
			//maximum!!!
			if(ty+i < TILE_WIDTHy)
			{
			    partialsum[ty][tx] += partialsum[ty+i][tx];
			}
			else
			{
			}
			__syncthreads();
		}

	}
	__syncthreads();

	*(b + by * m_col + Col) = partialsum[0][tx]; //
}

int main()
{
	FILE *fp_in;
	FILE *fp_mportf;

	float *m_loss;
	//this pointer points to the space for data of potrfolio matrix
	float *m_portf;
	//this pointer points to the sapce for single portfolio
	float *m_portfolio;
	float *m_pv;
	float *data_temp;
	float *matrix_col_sum;
	float *block_col_sum;
	float *m_b;
	float *m_temp1;

	float *CUDA_ca; //GPU space pointer for loss matrix used by
//CUDA_matrix_col_sum
	float *CUDA_cb; //GPU space pointer for CUDA_matrix_col_sum
	float *CUDA_portf; //GPU space pointer for weight vector used by
//MatrixVectorMulKernel
	float *CUDA_b;  //GPU space pointer for b used by
//MatrixVectorMulKernel

	float ljlp[M];
```

```
float ljsp[M];
float ljlm[M];
float ljsm[M];
float ls=0;
float w_londa=0;
float w_arrow=0;
float a1=0;
float b1=0;
float vr_upperbound=0;
float var;

float sp = (float)DRAW * P;

int size_CUDA_ca = TEMP_AREA * M * sizeof(float);
int size_CUDA_cb = TEMP_AREA/TILE_WIDTHy * M * sizeof(float);
int size_CUDA_portf = M * sizeof(float);
int size_CUDA_b = TEMP_AREA * sizeof(float);
int size_CUDA_time=sizeof(clock_t)*(TEMP_AREA/TILE_WIDTHy2) * 2;

int to_buy=0;
int not_to_buy=0;

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;
clock_t* CUDA_time;
clock_t* time_used;
clock_t avg_clk=0;

m_loss = new float [DRAW*M];
m_portf = new float [PORTF_NUM * M];
m_portfolio = new float [M];
m_pv = new float [DRAW];
data_temp = new float [TEMP_AREA * M];
matrix_col_sum = new float [M];
ini_fmatrix(matrix_col_sum , 1,M);
block_col_sum = new float [TEMP_AREA/TILE_WIDTHy * M];
m_b = new float [TEMP_AREA];
m_temp1 = new float [M];
time_used = new clock_t [ (TEMP_AREA/TILE_WIDTHy2) * 2 ];

fp_in = fopen("d:\\loss16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
//loading data
printf("Loading matrix Loss...\n");
loaddata(fp_in , m_loss , DRAW, M);
printf("Loading portfolio data.\n");
loaddata(fp_mportf , m_portf , PORTF_NUM , M);

//===Initialization: these steps are only carried out
//once================================================
for(int i=0; i<M ; i++)
{
      matrix_col_cpy( m_loss, m_pv, DRAW, M , i);
```

```
            quicksort(m_pv, 0, DRAW-1);  //quicksort

            //sum of the p largest components of column j of L
            ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
            //sum of the N-p smallest components of column j of L
            ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
            //sum of the p smallest components of column j of L
            ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
            //sum of the N-p largest components of column j of L
            ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
      }

      //declare threadsperblock and blocks for CUDA_matrix_col_sum
      dim3 threadsperblock( TILE_WIDTHx, TILE_WIDTHy );
      dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy);
      //declare threadsperblock and blocks for MatrixVectorMulKernel
      dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
      dim3 numBlocks2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

      cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
      cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);

      for(int i=0 ; i< DRAW/TEMP_AREA ; i++)
      {
          //
          matrix_move(m_loss , data_temp, TEMP_AREA , M ,
i*TEMP_AREA );

          //copy loss matrix from CPU to GPU
          cudaMemcpy(CUDA_ca , data_temp , size_CUDA_ca ,
cudaMemcpyHostToDevice);
          //compute column sum for each column in loss matrix
          CUDA_matrix_col_sum<<< numBlocks , threadsperblock
>>>( CUDA_ca , CUDA_cb , TEMP_AREA , M );
          //copy answer block from GPU to CPU
          cudaMemcpy( block_col_sum , CUDA_cb , size_CUDA_cb ,
cudaMemcpyDeviceToHost);

          //compute sum for each column
          matrix_sum_of_col( block_col_sum , matrix_col_sum ,
TEMP_AREA/TILE_WIDTHy , M);
      }
      //================================================================

      cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
      cudaMalloc((void**) &CUDA_b , size_CUDA_b);
      cudaMalloc((void**) &CUDA_time , size_CUDA_time);

      printf("CUDA_Program3:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);
```

```
      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0; k<PORTF_NUM ; k++)
      {
           a1=0;
           b1=0;
           vr_upperbound = 0;
           //compute S
           portf_data_cpy(m_portf , m_portfolio , PORTF_NUM , M , k);

         vector_ele_multi( matrix_col_sum , m_portfolio , m_temp1 , M);

           ls = vector_sum( m_temp1 ,M );
           //printf("S:%f \n" , ls);

           //compute w_londa as defined by equation 3 in O(M) time
           for(int i=0; i<M ; i++)
           {
             //compute w_lomda
             if(*(m_portfolio+i) >=0)
             {
               w_londa = *(m_portfolio+i);
             }
             else
             {
               w_londa = 0.0;
             }

             //compute w_arrow
             w_arrow = w_londa - *(m_portfolio+i);

             a1 += w_londa * ( ljlp[i] - ljsp[i] );

             b1 += w_arrow * ( ljlm[i] - ljsm[i] );

           }

           vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
           //printf("VR_upperbound: %f \n", vr_upperbound);

           //if vr_upperbound <= Vc, then the answer to the decision is
           //No.
           //Otherwise, the answer is not determined by this bound
           if( vr_upperbound <= COLLATERAL )
           {
                   to_buy++;
           }
           else
           {
                 //Do Monte Carlo simulation
                 cudaMemcpy( CUDA_portf , m_portfolio ,
size_CUDA_portf , cudaMemcpyHostToDevice );
                 //Invoke core for matrix-vector multiplication
                 MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2
>>>(CUDA_ca , CUDA_portf , CUDA_b , TEMP_AREA ,M, CUDA_time);
                 //copy result from GPU to CPU
```

```
                cudaMemcpy( m_b ,CUDA_b , size_CUDA_b ,
cudaMemcpyDeviceToHost );
                 cudaMemcpy( time_used , CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost);

                //quicksort in ascending order
                quicksort(m_b , 0 , DRAW-1);

                var = m_b[ DRAW - (int)sp ];

                //printf("\nVaR: %f\n", var);
                if(var < COLLATERAL)
                {
                        to_buy++;
                }
                else
                {
                        not_to_buy++;
                }

          }

          if(k==0)
          {
                 for(int i=0 ; i<TEMP_AREA/TILE_WIDTHy2 ; i++)
                 {
                 avg_clk += *(time_used + i + TEMP_AREA/TILE_WIDTHy2)
- *(time_used + i);
                 }
          }
     }
     QueryPerformanceCounter(&t2);
     total_time = clock() - s_time;

     elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;

     printf("\n%f ms.\n" , elapsedTime);
     printf("Total clocks: %d\n\n" , total_time);
     printf("VaR > Vc: %d\n", not_to_buy);
     printf("VaR <= Vc: %d\n", to_buy);


     delete [] m_loss;
     delete [] m_portf;
     delete [] m_portfolio;
     delete [] m_pv;
     delete [] data_temp;
     delete [] matrix_col_sum;
     delete [] block_col_sum;
     delete [] m_b;
     delete [] m_temp1;
     delete [] time_used;

     cudaFree(CUDA_ca);
     cudaFree(CUDA_cb);
     cudaFree(CUDA_portf);
```

```
	cudaFree(CUDA_b);
	cudaFree(CUDA_time);

	return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
	int i;

	for(i=0 ; i<a_col ; i++)
	{
	    *(b + i) = *(a + a_row2 * a_col + i);
	}

}
```

cudaprog4main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <windows.h>
#include "matrix_ope.h"

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16  //this is used by MatrixMulKernelv3
#endif

#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16  //this is used by MatrixVectorMulKernel2
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16 //this is used by MatrixMulKernelv3
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20  //this is used by MatrixVectorMulKernel2
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
```

```
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
extern void matrix_col_cpy(float* target, float* vect, int row_t, int
col_t, int col_i);
extern float vector_sum2(float* a, int start, int end);
extern void vector_ele_multi(float* a, float* b, float* ans, int
ele_a);
extern void matrix_cpy3(float* a, float* c, int a_row, int a_col, int
start_row , int end_row, int start_col , int end_col);
extern void matrix_multi(float* m , float* n , float* ans, int m_row ,
int m_col , int n_row , int n_col ,int ans_row , int ans_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int
m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx]; // initial value = 0
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx]; // initial value = 0

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
       //loading data into shared memory
       //ty -> row , tx -> col
       Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
       //ty -> tow, tx -> col
       Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
       __syncthreads();

       for(int j=0 ; j<TILE_WIDTHx ; j++)
       {
             pvalue += Mds[ty][j] * Nds[j][tx];
             __syncthreads();
       }
       *(pd + Row * m_col + Col) = pvalue;
   }
```

```
}
//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
      //int Col = bx * TILE_WIDTHx + tx;

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

}

//(CUDA)core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
      __shared__ float Nds[TILE_WIDTHx2];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy2 + ty;
      //int Col = bx * TILE_WIDTHx + tx;
      if(ty==0) time[by]=clock();

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
```

```
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx2 ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

      if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2]=clock();

}

int main()
{
      FILE *fp_mu;
      FILE *fp_md;
      FILE *fp_mv;
      FILE *fp_mportf;
      FILE *fp_out;

      float *m_u;
      float *m_d;
      float *m_v;
      float *m_portf;
      float *m_portfolio;
      float *m_pv;
      float *fp_ljlp;
      float *fp_ljsp;
      float *fp_ljlm;
      float *fp_ljsm;
      float *dv_temp;
      float *m_dvw;
      float *mu_col_sum;
      float *m_stemp;
      float *m_b;
      float *m_utemp; //for monte carlo simulation
      float *m_btemp; //for monte carlo simulation

      float *vr_upp;  //this array is used to check vr_upperbound

      //declare pointer for CUDA space allocation
      float *CUDA_ca;
      float *CUDA_cb;
      float *CUDA_cc;
      float *CUDA_portfolio;
      float *CUDA_dvw;
      float *CUDA_utemp;  //monte carlo simulation
      float *CUDA_btemp;  //monte carlo simulation

      float ljlp[M];
      float ljsp[M];
      float ljlm[M];
      float ljsm[M];
      float ls=0;
      float w_londa=0;
```

```
	float w_arrow=0;
	float a1=0;
	float b1=0;
	float vr_upperbound=0;
	float var;

	float sp = (float)DRAW * P;

	int size_CUDA_ca = M*M*sizeof(float);
	int size_CUDA_cb = M*M*sizeof(float);
	int size_CUDA_cc = M*M*sizeof(float);
	int size_CUDA_portfolio = M*sizeof(float);
	int size_CUDA_utemp = TEMP_AREA * M * sizeof(float);  //for monte
//carlo simulation
	int size_CUDA_btemp = TEMP_AREA * sizeof(float);      //for monte
//carlo simulation
	int var_index;
	int mt_count=0;
	int to_buy=0;
	int not_to_buy=0;
	int size_CUDA_time = sizeof(clock_t)*(TEMP_AREA/TILE_WIDTHy2) *2;

	LARGE_INTEGER frequency;
	LARGE_INTEGER t1, t2;
	double elapsedTime;

	clock_t s_time;
	clock_t total_time;
	clock_t* CUDA_time;
	clock_t* time_used;
	clock_t avg_clk=0;
	time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy2) * 2];

	var_index = DRAW - (int)sp;

	m_u = new float [DRAW*M];
	m_d = new float [M*M];
	m_v = new float [M*M];
	m_portf = new float [PORTF_NUM * M];
	m_portfolio = new float [M];
	m_pv = new float [DRAW];
	dv_temp = new float [M*M];
	m_dvw = new float [M];
	mu_col_sum = new float [M];
	m_stemp = new float [M];
	m_b = new float [DRAW];
	m_utemp = new float [TEMP_AREA * M];
	m_btemp = new float [TEMP_AREA];

	vr_upp = new float [PORTF_NUM];

	fp_mu = fopen("d:\\u_extract16.txt", "r");
	fp_md = fopen("d:\\matrixw16.txt", "r");
	fp_mv = fopen("d:\\matrixv16.txt", "r");
	fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
	//loading data
	printf("Loading matrix U.\n");
```

```
      loaddata(fp_mu , m_u , DRAW, M);
      printf("Loading matrix D..\n");
      loaddata(fp_md , m_d , M , M);
      printf("Loading matrix V...\n");
      loaddata(fp_mv , m_v , M , M);
      printf("Loading data for portfolio matrix.\n");
      loaddata(fp_mportf , m_portf, PORTF_NUM , M);

      //=Initialization: these steps are only carried out
      //once=========================================================
      fp_ljlp = &ljlp[0];
      fp_ljsp = &ljsp[0];
      fp_ljlm = &ljlm[0];
      fp_ljsm = &ljsm[0];
      ini_fmatrix( fp_ljlp , 1 , M);
      ini_fmatrix( fp_ljsp , 1 , M);
      ini_fmatrix( fp_ljlm , 1 , M);
      ini_fmatrix( fp_ljsm , 1 , M);

      for(int i=0; i<M ; i++)
      {
        matrix_col_cpy( m_u, m_pv, DRAW, M , i);
        quicksort(m_pv, 0, DRAW-1);  //quicksort
        //compute the sum of each column of U
        mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);

        //sum of the p largest components of column j of L
        ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
        //sum of the N-p smallest components of column j of L
        ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
        //sum of the p smallest components of column j of L
        ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
        //sum of the N-p largest components of column j of L
        ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
      }

      dim3 threadsperblock( TILE_WIDTHx, TILE_WIDTHy );
      dim3 numBlocks( M/TILE_WIDTHx , M/TILE_WIDTHy);

      cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
      cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);
      cudaMalloc((void**) &CUDA_cc , size_CUDA_cc);

      cudaMemcpy(CUDA_ca , m_d , size_CUDA_ca ,
cudaMemcpyHostToDevice);
      cudaMemcpy(CUDA_cb , m_v , size_CUDA_cb ,
cudaMemcpyHostToDevice);
      //Invoke compute D * V
      MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca,
CUDA_cb , CUDA_cc, M, M);

      cudaMemcpy(dv_temp , CUDA_cc , size_CUDA_cc ,
cudaMemcpyDeviceToHost);

      cudaMalloc((void**) &CUDA_utemp , size_CUDA_utemp);  //for monte
//carlo simulation
```

```
      cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp);  //for monte
//carlo simulation
      cudaMalloc((void**) &CUDA_time , size_CUDA_time);

      dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );  //for
//monte carlo simulation
      dim3 numBlock2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );  //for
//monte carlo simulation

      matrix_cpy3( m_u , m_utemp , DRAW, M , 0 , DRAW-1 , 0, M-1);
//only need first M column of U
      cudaMemcpy(CUDA_utemp , m_utemp , size_CUDA_utemp ,
cudaMemcpyHostToDevice);
      //=================================================================

      cudaMalloc((void**) &CUDA_portfolio , size_CUDA_portfolio);
      cudaMalloc((void**) &CUDA_dvw , size_CUDA_portfolio);

      printf("CUDA_Program4:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);

      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0; k<PORTF_NUM ; k++)
      {
           a1=0;
           b1=0;
           ls=0;
           vr_upperbound = 0;
           portf_data_cpy(m_portf , m_portfolio , PORTF_NUM , M, k);

           //compute D*V*w
           matrix_multi(dv_temp, m_portfolio, m_dvw, M, M, M, 1, M,
1 );


           //compute S:
           vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);

           ls = vector_sum2(m_stemp , 0 , M-1);
           //printf("S: %f \n", ls);

           //compute w_londa as defined by equation 3 in O(M) time
           for(int i=0; i<M ; i++)
           {
              //compute w_lomda
              if(*(m_dvw+i) >=0)
              {
                 w_londa = *(m_dvw+i);
              }
              else
              {
```

```
                w_londa = 0.0;
            }

            //compute w_arrow
            w_arrow = w_londa - *(m_dvw+i);

            a1 += w_londa * ( ljlp[i] - ljsp[i] );

            b1 += w_arrow * ( ljlm[i] - ljsm[i] );
         }

         vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
         //printf("VR_upperbound: %f \n", vr_upperbound);

         //*(vr_upp+k) = vr_upperbound;

         //if vr_upperbound <= Vc, then the answer to the decision is
//No. Otherwise, the answer is not determined by this bound
         if( vr_upperbound <= COLLATERAL )
         {
                to_buy++;
         }
         else
         {
               //printf("Run Monte Carlo simulation\n");
               mt_count++;
               cudaMemcpy( CUDA_dvw, m_dvw, size_CUDA_portfolio ,
cudaMemcpyHostToDevice );
               for(int i=0; i < DRAW/TEMP_AREA ; i++)
               {
                     //Invoke CUDA core for matrix-vector
                     //multiplication
                     MatrixVectorMulKernel2<<< numBlock2 ,
threadsperblock2 >>>( CUDA_utemp , CUDA_dvw , CUDA_btemp , TEMP_AREA ,
M , CUDA_time);
                     //copy the result from GPU to CPU
                     cudaMemcpy( m_btemp , CUDA_btemp ,
size_CUDA_btemp , cudaMemcpyDeviceToHost );
               }

                //quicksort in ascneding order
                quicksort( m_btemp , 0, DRAW-1);

                var = m_btemp[var_index];
                //printf("VaR: %f \n", var);

                if( var > COLLATERAL )
                {
                     not_to_buy++;
                }
                else
                {
                     to_buy++;
                }

                cudaMemcpy( time_used, CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost);
```

```
                  if(k==0)
                  {
                        for( int i=0; i< TEMP_AREA/TILE_WIDTHy2 ; i++)
                        {
                           avg_clk += *(time_used + i +
TEMP_AREA/TILE_WIDTHy2) - *(time_used + i);
                        }
                  }
            }

      }

      QueryPerformanceCounter(&t2);
      total_time = clock() - s_time;

      elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;


      printf("\n%f ms.\n", elapsedTime);
      printf("Toal clocks: %d \n\n", total_time);
      printf("VaR > Vc: %d\n", not_to_buy);
      printf("VaR <= Vc: %d\n", to_buy);


      delete [] m_u;
      delete [] m_d;
      delete [] m_v;
      delete [] m_portfolio;
      delete [] m_pv;
      delete [] dv_temp;
      delete [] m_dvw;
      delete [] mu_col_sum;
      delete [] m_stemp;
      delete [] m_b;
      delete [] m_utemp;
      delete [] m_btemp;
      delete [] vr_upp;

      cudaFree(CUDA_ca);
      cudaFree(CUDA_cb);
      cudaFree(CUDA_cc);
      cudaFree(CUDA_portfolio);
      cudaFree(CUDA_dvw);

      cudaFree(CUDA_utemp);
      cudaFree(CUDA_btemp);


      return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
      int i;
```

```
      for(i=0 ; i<a_col ; i++)
      {
          *(b + i) = *(a + a_row2 * a_col + i);
      }

}
```

cudaprog5main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

//Here TILE_WIDTHx = the number of threads in a block!  this is used
//in kernel CUDA_find_minmax
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 250
#endif

#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
```

```
extern void matrix_col_cpy(float* target, float* vect, int row_t, int
col_t, int col_i);
extern void vect_min_max(float* target ,float *ans, int ele_m);
extern void matrix_cpy3(float* a, float* c, int a_row, int a_col, int
start_row , int end_row, int start_col , int end_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) code to find out the minimum value in ca
//this CUDA function works only when matrix elememt number is the power
//of 2!!
__global__ void CUDA_find_min(float *ca , float *ans_min)
{
   __shared__ float min[TILE_WIDTHx];

   int bx = blockIdx.x;
   int tx = threadIdx.x;
   int m_index = bx * blockDim.x + tx;
   int nTotalThreads = blockDim.x;

   min[tx] = *(ca + m_index);

   while( nTotalThreads > 1)
   {
      int halfPoint = nTotalThreads/2;

        if(threadIdx.x < halfPoint)
        {
               float temp = min[threadIdx.x + halfPoint];

               if( temp < min[threadIdx.x])
               {
                     min[threadIdx.x] = temp;
               }
        }
        __syncthreads();

        nTotalThreads = nTotalThreads/2;
   }

   *(ans_min + bx) = min[0];  //the smallest element will be stored in
   //the first element of the array

}

//(CUDA) code to find out the minimum/maximum value in ca
//this CUDA function works only when matrix elememt number is the power
//of 2!!
__global__ void CUDA_find_minmax(float *ca , float *ans_min_max)
{
   __shared__ float ma[TILE_WIDTHx];

   int bx = blockIdx.x;
   int tx = threadIdx.x;
   int m_index = bx * TILE_WIDTHx + tx;
   float min_v;
```

```
    float max_v;

    ma[tx] = *(ca + m_index);
    min_v = ma[0];
    max_v = ma[0];

    for(int i=0; i<TILE_WIDTHx ; i++)
    {
          if(min_v > ma[i])
          {
                min_v = ma[i];
          }
          else
          {
          }
          if(max_v < ma[i])
          {
                max_v = ma[i];
          }
          else
          {
          }
          __syncthreads();
    }

    *(ans_min_max + bx ) = min_v;
    *(ans_min_max + DRAW/TILE_WIDTHx + bx) = max_v;
}

//(CUDA) code for computing matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
      __shared__ float Nds[TILE_WIDTHx2];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy2 + ty;
      //int Col = bx * TILE_WIDTHx + tx;
      if(ty==0) time[by]=clock();

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx2 ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }
```

```
        *(pd + Row) = pvalue;
      }

      if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2]=clock();
}

int main()
{
      FILE* fp_mloss;
      FILE *fp_portf_m;
      FILE* fp_out;

      float *m_loss;
      float *portf_m;
      float *portfolio;
      float *col_temp;
      float *ans_temp;
      float *col_min_max;
      float *column_min;
      float *column_max;
      float *m_b;
      float *m_utemp;
      float *m_btemp;

      float *ca; //GPU space pointer for CUDA_find_minmax
      float *cb; //GPU space pointer for CUDA_find_minmax
      //GPU space pointer for Monte Carlo simulation used by
      //MatrixVectorMulKernel2
      float *CUDA_utemp;
      //GPU space pointer for MatrixVectorMulKernel2
      float *CUDA_btemp;
      //GPU space pointer for MatrixVectorMulKernel2
      float *CUDA_portf;

      float a1=0;
      float b1=0;
      float w_londa;
      float w_arrow;
      float vr_upperbound=0;
      float var;

      int sp = (int)((float)DRAW * P);
      int var_index = DRAW - sp;

      int size_CUDA_ca = DRAW * sizeof(float);
      int size_CUDA_cb = 2 * DRAW/TILE_WIDTHx * sizeof(float);

      int size_CUDA_utemp = TEMP_AREA * M * sizeof(float);
      int size_CUDA_btemp = TEMP_AREA * sizeof(float);
      int size_CUDA_portf = M * sizeof(float);
      int size_CUDA_time=sizeof(clock_t)*(TEMP_AREA/TILE_WIDTHy2) * 2;

      int to_buy=0;
      int not_to_buy=0;

      LARGE_INTEGER frequency;
```

```
      LARGE_INTEGER t1, t2;
      double elapsedTime;

      clock_t s_time;
      clock_t total_time;
      clock_t* CUDA_time;
      clock_t* time_used;
      time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy2)*2];
      clock_t avg_clk=0;

      fp_mloss = fopen("d:\\loss16.txt","r");
      fp_portf_m = fopen("d:\\portfoliom16.txt", "r");

      m_loss = new float [DRAW*M];  //loss matrix (DRAW by M)
      portf_m = new float [PORTF_NUM * M];
      portfolio = new float [M];    //portfolio matrix (M)
      col_temp = new float [DRAW];  //
      ans_temp = new float [2 * DRAW/TILE_WIDTHx];
      col_min_max = new float [2];
      column_min = new float [M];
      column_max = new float [M];
      m_b = new float [DRAW];
      m_utemp = new float [TEMP_AREA * M];
      m_btemp = new float [TEMP_AREA];

      printf("Loading data:\n");
      loaddata(fp_mloss, m_loss , DRAW, M);
      loaddata(fp_portf_m, portf_m, PORTF_NUM , M);

      dim3 threadsPerBlock( TILE_WIDTHx , 1 );
      dim3 numBlock( DRAW/TILE_WIDTHx , 1);

      cudaMalloc((void**) &ca , size_CUDA_ca);
      cudaMalloc((void**) &cb , size_CUDA_cb);

      cudaMalloc((void**) &CUDA_utemp , size_CUDA_utemp);
      cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp);
      cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
      cudaMalloc((void**) &CUDA_time, size_CUDA_time);

      //====Initialize: The following steps are carried out only
      //once=============================================
      for(int i=0; i<M ; i++)
      {
            matrix_col_cpy(m_loss , col_temp , DRAW , M , i);  //
            //move data from col_temp (Host) to ca (Device)
            cudaMemcpy(ca , col_temp , size_CUDA_ca ,
cudaMemcpyHostToDevice);
            //find out the minimum/maximum value in the column using
            //cuda core CUDA_find_minmax
            CUDA_find_minmax<<< numBlock , threadsPerBlock >>>( ca,
cb );
            cudaMemcpy( ans_temp, cb , size_CUDA_cb,
cudaMemcpyDeviceToHost);

            vect_min_max( ans_temp , col_min_max, DRAW/TILE_WIDTHx);
```

```
            *(column_min+i) = *(col_min_max+0);
            *(column_max+i) = *(col_min_max+1);
      }

      //Declare threads per block and block number for monte carlo
      //simulation
      dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
      dim3 numBlock2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

      //copy loss matrix data from CPU to GPU
      cudaMemcpy(CUDA_utemp , m_loss , size_CUDA_utemp ,
cudaMemcpyHostToDevice);
      //=============================================================
      //Computation: The following steps are carried out for the w that
      //is given
      printf("CUDA_Program5:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);

      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0 ; k< PORTF_NUM ; k++)
      {
            a1=0;
            b1=0;
            portf_data_cpy( portf_m , portfolio , PORTF_NUM , M , k);

            for(int i=0; i<M ; i++)
            {
                if(*(portfolio+i) >= 0.0)
            {
                w_londa = *(portfolio+i);
            }
            else
            {
                w_londa = 0.0;
            }

                //compute w_arrow
                w_arrow = w_londa - *(portfolio+i);

                a1 +=  *(column_max+i) * w_londa;
                b1 +=  *(column_min+i) * w_arrow;

            }

          vr_upperbound = a1 - b1;
          //printf("VR_upperbound: %f\n" , vr_upperbound);

          //If vr_upperbound <= Vc then the answer to the decision
          //problem is No. Otherwise, the answer is not determined by
          //this bound
```

```
            //If it fails to give an answer, then we fall back upon the
            //conventional technique.
            if( vr_upperbound <= COLLATERAL )
            {
                  to_buy++;
            }
            else
            {
                  //printf("Run Monte Carlo simulation!\n");
                  //copy portfolio data from CPU to GPU
                  cudaMemcpy(CUDA_portf , portfolio , size_CUDA_portf ,
cudaMemcpyHostToDevice);

                  for(int i=0; i < DRAW/TEMP_AREA ; i++)
                  {
                        //Invoke CUDA core MatrixVectorMulKernel2 to do
                        //matrix-vector multiplication
                        MatrixVectorMulKernel2<<< numBlock2 ,
threadsperblock2 >>>( CUDA_utemp , CUDA_portf , CUDA_btemp ,
TEMP_AREA , M , CUDA_time);

                        cudaMemcpy( m_btemp , CUDA_btemp ,
size_CUDA_btemp , cudaMemcpyDeviceToHost );
                  }

                   //quicksort in ascending order
                   quicksort( m_btemp , 0, DRAW-1);

                   var = m_btemp[var_index];
                   //printf("VaR: %f \n", var);
                   if(var > COLLATERAL)
                   {
                         not_to_buy++;
                   }
                   else
                   {
                         to_buy++;
                   }

                   cudaMemcpy( time_used , CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost );

              }
              if(k==0)
              {
                    for(int i=0 ; i< TEMP_AREA/TILE_WIDTHy2 ; i++ )
                    {
                          avg_clk += *(time_used + i +
TEMP_AREA/TILE_WIDTHy2) - *(time_used + i);
                    }
              }
      }
      QueryPerformanceCounter(&t2);
      total_time = clock() - s_time;

      elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;
```

```
	printf("\n%f ms.\n", elapsedTime);
	printf("Total clocks:%d \n\n", total_time);
	printf("VaR > Vc: %d\n", not_to_buy);
	printf("VaR <= Vc: %d\n", to_buy);


	//free space
	delete [] m_b;
	delete [] m_utemp;
	delete [] m_btemp;

	cudaFree(CUDA_utemp);
	cudaFree(CUDA_btemp);
	cudaFree(CUDA_portf);

	delete [] m_loss;
	delete [] portf_m;
	delete [] portfolio;
	delete [] col_temp;
	delete [] ans_temp;
	delete [] col_min_max;
	delete [] column_min;
	delete [] column_max;

	cudaFree(ca);
	cudaFree(cb);

	return 0;
}

void show_vect(float *a , int ele_a)
{
   for(int i=0 ; i<ele_a ; i++)
   {
	printf("%f ", *(a+i));
   }
   printf("\n\n");
}

void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
	int i,j;
	for(i=0 ; i< m_row ; i++)
	{
		for(j=0 ; j< m_col ; j++)
		{
		    *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
		}
	}

}

void vector_move(float *a, float *b , int ele_num , int start_e)
{
	int i;
```

```
	for(i=0; i<ele_num ; i++)
	{
		*(b+start_e+i) = *(a+i);
	}
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
	int i;

	for(i=0 ; i<a_col ; i++)
	{
	    *(b + i) = *(a + a_row2 * a_col + i);
	}

}
```

cudaprog6main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

//define how many rows for our loss matrix
#ifndef DRAW
#define DRAW 10000
#endif
//define the column of loss matrix
#ifndef M
#define M 16
#endif
//confidence level 99%
#ifndef P
#define P 0.01
#endif

//this is used in kernal MatrixMulKernelv3
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

//this is used in MatrixVectorMulKernel2
#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

//this is used in kernal MatrixMulKernelv3
```

```
#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif

//this is used in MatrixVectorMulKernel2
#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif
//Collateral
#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif
//define how many portfolios to simulate
#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void matrix_find_min_max(float* target, float* min_max, int
row_t, int col_t, int col_i);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
extern void matrix_col_cpy(float* target, float* vect, int row_t, int
col_t, int col_i);
extern void matrix_cpy3(float* a, float* c, int a_row, int a_col, int
start_row , int end_row, int start_col , int end_col);
extern void matrix_multi(float* m , float* n , float* ans, int m_row ,
int m_col , int n_row , int n_col ,int ans_row , int ans_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) core for finding minimum/maximum value in vector ca
__global__ void CUDA_find_minmax(float *ca , float *ans_min_max)
{
   __shared__ float ma[TILE_WIDTHx];

   int bx = blockIdx.x;
   int tx = threadIdx.x;
   int m_index = bx * TILE_WIDTHx + tx;
   float min_v;
   float max_v;

   ma[tx] = *(ca + m_index);
   min_v = ma[0];
   max_v = ma[0];
```

```
    for(int i=0; i<TILE_WIDTHx ; i++)
    {
          if(min_v > ma[i])
          {
                min_v = ma[i];
          }
          else
          {
          }
          if(max_v < ma[i])
          {
                max_v = ma[i];
          }
          else
          {
          }
          __syncthreads();
    }

    *(ans_min_max + bx ) = min_v;
    *(ans_min_max + DRAW/TILE_WIDTHx + bx) = max_v;
}

//(CUDA) core for matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int
m_row , int m_col)
{
    __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
    __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int Row = by * TILE_WIDTHy + ty;
    int Col = bx * TILE_WIDTHx + tx;

    float pvalue =0;


    for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
    {
          //loading data into shared memory
          Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
          Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
          __syncthreads();

          //
          for(int j=0 ; j<TILE_WIDTHx ; j++)
          {
                pvalue += Mds[ty][j] * Nds[j][tx];
                __syncthreads();
          }
          //

          *(pd + Row * m_col + Col) = pvalue;
```

```
   }

}

//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
      //int Col = bx * TILE_WIDTHx + tx;

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx)+1 ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

}

//(CUDA) this function is the same as above except the size of shared
//memory
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
      __shared__ float Nds[TILE_WIDTHx2];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy2 + ty; //size change
      //int Col = bx * TILE_WIDTHx + tx; //size change
      if(ty==0) time[by]=clock();

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
```

```
	{
	  Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
	  Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
	  __syncthreads();

	  for(int j=0 ; j < TILE_WIDTHx2 ; j++)
	  {
		pvalue += Mds[ty][j] * Nds[j];
	  }

	  *(pd + Row) = pvalue;
	}

	if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2]=clock();

}

int main()
{
	FILE* fp_mu;
	FILE* fp_mv;
	FILE* fp_mw;
	FILE* fp_mportf;
	FILE* fp_out;

	float *m_u;
	float *m_v;
	float *m_w;
	float *m_portf;
	float *portfolio;
	float *col_temp;
	float *p_minmax;
	float *dv_temp;
	float *m_dvw;
	float *m_b;
	float *m_utemp;  //for monte carlo simulation
	float *m_btemp;  //for monte carlo simulation

	float *CUDA_ca;
	float *CUDA_cb;
	float *CUDA_cc;
	float *CUDA_portfolio;
	float *CUDA_dvw;
	float *CUDA_utemp; //for monte carlo simulation
	float *CUDA_btemp; //for monte carlo simulation

	float lp[M];
	float lm[M];
	float w_londa;
	float w_arrow;
	float a1=0;
	float b1=0;
	float vr_upperbound=0;
	float var;

	int sp = (int)((float)DRAW * P);
	int var_index = DRAW - sp;
```

```
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = M * M * sizeof(float);
int size_CUDA_cb = M * M * sizeof(float);
int size_CUDA_cc = M * M * sizeof(float);
int size_CUDA_portfolio = M * sizeof(float);
//for monte carlo simulation
int size_CUDA_utemp = TEMP_AREA * M * sizeof(float);
//for monte carlo simulation
int size_CUDA_btemp = TEMP_AREA * sizeof(float);
int size_CUDA_time=sizeof(clock_t)*(TEMP_AREA/TILE_WIDTHy2) * 2;

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;
clock_t* CUDA_time;
clock_t* time_used;
clock_t avg_clk=0;

m_u = new float [DRAW*M];
m_v = new float [M*M];
m_w = new float [M*M];  //Identity matrix we only need non zero
                        //rows
m_portf = new float [PORTF_NUM * M];
portfolio = new float [M];
col_temp = new float [DRAW];
p_minmax = new float [2];
dv_temp = new float [M*M];
m_dvw = new float [M];
m_b = new float [DRAW];
m_utemp = new float [TEMP_AREA * M];  //monte carlo simulation
m_btemp = new float [TEMP_AREA];      //monte carlo simulation
time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy2)*2];

fp_mu = fopen("d:\\u_extract16.txt", "r");
fp_mv = fopen("d:\\matrixv16.txt", "r");
fp_mw = fopen("d:\\matrixw16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt", "r");

printf("Start loading data...");
loaddata(fp_mu, m_u , DRAW , M);
loaddata(fp_mv, m_v , M ,M);
loaddata(fp_mw, m_w , M ,M);
loaddata(fp_mportf , m_portf , PORTF_NUM , M);
printf("complete.\n\n");


//==The following steps are carried out only
//once=============================================
//find out Max and min value for each column
for(int i=0 ; i<M ; i++)
{
   matrix_col_cpy(m_u , col_temp, DRAW , M , i);
```

```
         matrix_find_min_max( m_u , p_minmax , DRAW, M , i);

         lp[i] = *(p_minmax+0); //Max value for each column
         lm[i] = *(p_minmax+1); //min value for each column
      }

      //for computing D * V
      dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy);
      dim3 numBlocks( M/TILE_WIDTHx ,  M/TILE_WIDTHy);

      //for monte carlo simulation
      dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
      dim3 numBlock2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

      cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
      cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);
      cudaMalloc((void**) &CUDA_cc , size_CUDA_cc);

      cudaMalloc((void**) &CUDA_utemp , size_CUDA_utemp); //for monte
      //carlo simulation
      cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp); //for monte
      //carlo simulation
      cudaMalloc((void**) &CUDA_time , size_CUDA_time);
      //copy matrix D and V from CPU to GPU
      cudaMemcpy(CUDA_ca , m_w , size_CUDA_ca ,
cudaMemcpyHostToDevice);
      cudaMemcpy(CUDA_cb , m_v , size_CUDA_cb ,
cudaMemcpyHostToDevice);
      //matrix-matrix multiplication
      MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca,
CUDA_cb , CUDA_cc, M, M);
      //copy the result from GPU to CPU
      cudaMemcpy(dv_temp , CUDA_cc, size_CUDA_cc,
cudaMemcpyDeviceToHost);

      cudaMemcpy(CUDA_cc , dv_temp, size_CUDA_cc ,
cudaMemcpyHostToDevice);

      //only need first M column of U
      matrix_cpy3( m_u , m_utemp , DRAW, M , 0 , DRAW-1 , 0, M-1);
      //move U data from CPU to GPU
      cudaMemcpy(CUDA_utemp , m_utemp , size_CUDA_utemp ,
cudaMemcpyHostToDevice);

//=====================================================================
      cudaMalloc((void**) &CUDA_portfolio , size_CUDA_portfolio);
      cudaMalloc((void**) &CUDA_dvw , size_CUDA_portfolio);

      printf("CUDA_Program6:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);
```

```
      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0 ; k<PORTF_NUM ; k++)
      {
           a1=0;
           b1=0;
           vr_upperbound = 0;

           portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);

           //compute D * V * w
           matrix_multi( dv_temp, portfolio, m_dvw, M, M, M, 1, M, 1 );

           //compute w_londa and w_arrow in O(M) time
           for(int i=0 ; i<M ; i++)
           {
             if(*(m_dvw + i) >= 0.0)
             {
                  w_londa = *(m_dvw + i);
             }
             else
             {
                  w_londa = 0.0;
             }

                  w_arrow = w_londa - *(m_dvw + i);

                  a1 +=  lp[i] * w_londa;
                  b1 +=  lm[i] * w_arrow;

           }

           //compute VR upperbound
           vr_upperbound = a1 - b1;
           //printf("VR_Upperbound: %f\n", vr_upperbound);


           //If vr_upperbound <= Vc then the answer to the decision
           //problem is No.
           //Otherwise, the answer is not determined by this bound
           //If it fails to give an answer, then we fall back upon the
           //conventional technique.
           if(vr_upperbound <= COLLATERAL)
           {
                 to_buy++;
           }
           else
           {
                 //printf("Run Monte Carlo simulation!\n");
                 for(int i=0; i < DRAW/TEMP_AREA ; i++)
                 {
                       //copy data of D*V*w from CPU to GPU
                       cudaMemcpy( CUDA_dvw , m_dvw,
size_CUDA_portfolio , cudaMemcpyHostToDevice );
                       //Invoke CUDA core function
                       //MatrixVectorMulKernel2 to do matrix-vector
```

```
                        //multiplication
                        MatrixVectorMulKernel2<<< numBlock2 ,
threadsperblock2 >>>( CUDA_utemp , CUDA_dvw , CUDA_btemp , TEMP_AREA ,
M , CUDA_time);
                        //copy the result from GPU to CPU
                        cudaMemcpy( m_btemp , CUDA_btemp ,
size_CUDA_btemp , cudaMemcpyDeviceToHost );
                }
                //quicksort in ascending order
                quicksort( m_btemp , 0, DRAW-1);

                var = m_btemp[var_index];
                //printf("VaR: %f \n", var);

                if(var > COLLATERAL)
                {
                        not_to_buy++;
                }
                else
                {
                        to_buy++;
                }

                cudaMemcpy(time_used , CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost);
                  if(k==0)
                  {
                        for(int i=0 ; i< TEMP_AREA/TILE_WIDTHy2; i++ )
                        {
                            avg_clk += *(time_used + i +
TEMP_AREA/TILE_WIDTHy2) - *(time_used+i);
                        }
                  }
            }
        }
        QueryPerformanceCounter(&t2);
        total_time = clock() - s_time;

        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;

        printf("\n%f ms.\n", elapsedTime);
        printf("Total clocks: %d \n\n" , total_time);
        printf("VaR > Vc: %d\n", not_to_buy);
        printf("VaR <= Vc: %d\n", to_buy);


        //Free space
        delete [] m_u;
        delete [] m_v;
        delete [] m_w;
        delete [] m_portf;
        delete [] portfolio;
        delete [] col_temp;
        delete [] p_minmax;
        delete [] dv_temp;
        delete [] m_dvw;
```

```
        delete [] m_b;
        delete [] m_utemp;
        delete [] m_btemp;
        delete [] time_used;

        cudaFree(CUDA_ca);
        cudaFree(CUDA_cb);
        cudaFree(CUDA_cc);
        cudaFree(CUDA_portfolio);
        cudaFree(CUDA_dvw);

        cudaFree(CUDA_utemp);
        cudaFree(CUDA_btemp);
        cudaFree(CUDA_time);
        return 0;
}

void show_vect(float *a , int ele_a)
{
   for(int i=0 ; i<ele_a ; i++)
   {
         printf("%f ", *(a+i));
   }
   printf("\n\n");
}

void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
      int i,j;
      for(i=0 ; i< m_row ; i++)
      {
            for(j=0 ; j< m_col ; j++)
            {
                *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
            }
      }

}

void vector_move(float *a, float *b , int ele_num , int start_e)
{
      int i;
      for(i=0; i<ele_num ; i++)
      {
            *(b+start_e+i) = *(a+i);
      }
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
      int i;

      for(i=0 ; i<a_col ; i++)
      {
```

```
            *(b + i) = *(a + a_row2 * a_col + i);
      }

}
```

cudaprog7main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

//this is used in kernel CUDA_matrix_row_l2norm and
MatrixVectorMulKernel
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

//this is used in kernel CUDA_matrix_row_l2norm and
MatrixVectorMulKernel
#ifndef TILE_WIDTHy
#define TILE_WIDTHy 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern float matrix_row_l2norm( float* m, int row , int col, int
row_2);
extern float matrix_col_l2norm(float* m, int row , int col , int
col_2 );
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
void vector_squareroot(float *a , int ele_a);
```

```
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) this function compute the L2 norm for each row in ca and store
//the result in cb
__global__ void CUDA_matrix_row_l2norm( float *ca, float *cb , int
m_row , int m_col)
{
      int tx = threadIdx.x;
      int ty = threadIdx.y;
      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int ROW = by * blockDim.y + ty;  //use blockDim.y
      float m_total=0;

      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];

      for(int i=0; i< (m_col/TILE_WIDTHx); i++)
      {
         Mds[ty][tx] = *(ca + ROW*m_col + (i*TILE_WIDTHx)+tx);

         for(int j=0 ; j<TILE_WIDTHx ; j++)
         {
           m_total += Mds[ty][j] * Mds[ty][j];
         }

      }

      *(cb + ROW) = sqrt(m_total);
}

//(CUDA) core for computing matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
      //int Col = bx * TILE_WIDTHx + tx;

      float pvalue=0;

      if(ty==0) time[by]=clock();

      for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
      {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
          __syncthreads();
```

```
            for(int j=0 ; j < TILE_WIDTHx ; j++)
            {
                pvalue += Mds[ty][j] * Nds[j];
            }

            *(pd + Row) = pvalue;
        }

        if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy] = clock();

}
int main()
{
        //Input: L: Row by M and portfolio: M
        FILE* fp_ml;
        FILE* fp_mportf;
        FILE* fp_out;

        float *m_loss;
        float *m_portf;
        float *portfolio;
        float *m_pl2norm;
        float *m_temp;
        float *row_l2norm_temp;


        float *CUDA_ca;
        float *CUDA_cb;
        float *CUDA_portf;

        float wl;
        float wl2norm;
        float vr_upperbound;
        float var;

        int sp = (int)((float)DRAW * P);
        int to_buy=0;
        int not_to_buy=0;

        int size_CUDA_ca = TEMP_AREA * M * sizeof(float);
        int size_CUDA_cb = TEMP_AREA * sizeof(float);
        int size_CUDA_portf = M * sizeof(float);
        int size_CUDA_time=sizeof(clock_t) * (TEMP_AREA/TILE_WIDTHy) * 2;

        LARGE_INTEGER frequency;
        LARGE_INTEGER t1, t2;
        double elapsedTime;

        clock_t s_time;
        clock_t total_time;
        clock_t* CUDA_time;
        clock_t* time_used;
        time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy) * 2];
        clock_t avg_clk=0;

        m_loss = new float [DRAW*M];
        m_portf = new float [PORTF_NUM * M];
```

```
	portfolio = new float [M];
	m_pl2norm = new float [DRAW];  //store the L2 norm in each Row of
                                       //loss matrix L
	m_temp = new float [TEMP_AREA*M];
	row_l2norm_temp = new float [TEMP_AREA];


	fp_ml = fopen("d:\\loss16.txt", "r");
	fp_mportf = fopen("d:\\portfoliom16.txt", "r");

	printf("Loading data...");
	loaddata(fp_ml , m_loss , DRAW, M);
	loaddata(fp_mportf , m_portf , PORTF_NUM , M);
	printf("complete.\n");

	//=Initialize:Compute the L2 norm of each row of L=====
	dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
	dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy );

	cudaMalloc((void**) &CUDA_ca , size_CUDA_ca );
	cudaMalloc((void**) &CUDA_cb , size_CUDA_cb );
	cudaMalloc((void**) &CUDA_portf , size_CUDA_portf );
	cudaMalloc((void**) &CUDA_time , size_CUDA_time);

	for(int i=0 ;i< (DRAW/TEMP_AREA) ; i++)
	{
		cudaMemcpy(CUDA_ca , m_loss , size_CUDA_ca ,
cudaMemcpyHostToDevice);

		//Invoke CUDA core for computing the sum of power of 2 fo
		//each row element in matrix CUDA_ca
		CUDA_matrix_row_l2norm<<< numBlocks , threadsperblock
>>>( CUDA_ca , CUDA_cb , DRAW ,M);

		cudaMemcpy( m_pl2norm , CUDA_cb , size_CUDA_cb ,
cudaMemcpyDeviceToHost );

	}

	//compute the square root of each element
	vector_squareroot( m_pl2norm , DRAW);

	//quicksort
	quicksort( m_pl2norm , 0 ,DRAW-1);


	//Let WL be the pth largest one.
	wl = m_pl2norm[DRAW - sp];
	//printf("This is WL: %f\n\n" , wl);
	//=============================================================
	//Computation:
	printf("CUDA_Program7:");
	printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
	printf("DRAW=%d\n", DRAW );
	printf("M=%d\n", M);
	printf("P=%f\n", P);
```

```
	QueryPerformanceFrequency(&frequency);

	s_time = clock();
	QueryPerformanceCounter(&t1);
	for(int k=0 ; k<PORTF_NUM ; k++)
	{
		portf_data_cpy( m_portf, portfolio , PORTF_NUM , M, k);

		//compute L2 norm of portfolio in O(M) time
		wl2norm = matrix_col_l2norm( portfolio, M, 1,0 );
		//printf("\nL2 norm of portfolio:%f \n", wl2norm);

		vr_upperbound = wl2norm * wl;
		//printf("VR_upperbound: %f\n\n", vr_upperbound);

		//if VR_upperbound <= Vc the answer to the problem is No,
		//otherwise the answer is not determined by this bound
		if(vr_upperbound <= COLLATERAL)
		{

			to_buy++;
		}
		else
		{
			//printf("Run Monte Carlo simulation.\n");
			cudaMemcpy( CUDA_portf, portfolio ,
size_CUDA_portf , cudaMemcpyHostToDevice );

			for(int i=0 ; i<(DRAW/TEMP_AREA) ; i++)
			{
			  //Invoke CUDA core MatrixVectorMulKernel for
			  //matrix-vector multiplication
			  MatrixVectorMulKernel<<< numBlocks ,
threadsperblock >>>(CUDA_ca , CUDA_portf , CUDA_cb , TEMP_AREA , M ,
CUDA_time);

			  cudaMemcpy( m_pl2norm , CUDA_cb , size_CUDA_cb ,
cudaMemcpyDeviceToHost);//reuse the space of m_pl2norm
			}

			//quicksort
			quicksort( m_pl2norm , 0, DRAW-1);

			var = m_pl2norm[DRAW -sp];
			//printf("VaR: %f\n", var);
			if(var > COLLATERAL)
			{
			    not_to_buy++;
			}
			else
			{
			    to_buy++;
			}

		       cudaMemcpy( time_used , CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost );
```

```
                  if(k==0)
                  {
                     for(int i=0 ; i< TEMP_AREA/TILE_WIDTHy ; i++)
                     {
                           avg_clk += *(time_used + i +
TEMP_AREA/TILE_WIDTHy) - *(time_used + i);
                     }
                  }
            }
      }
      QueryPerformanceCounter(&t2);
      total_time = clock() - s_time;

      elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;

      printf("\n%f ms.\n", elapsedTime);
      printf("Total clocks:%d \n\n", total_time);
      printf("VaR > Vc: %d\n", not_to_buy);
      printf("VaR <= Vc: %d\n", to_buy);

      //Free space
      delete [] m_loss;
      delete [] m_portf;
      delete [] portfolio;
      delete [] m_pl2norm;
      delete [] m_temp;
      delete [] row_l2norm_temp;
      delete [] time_used;

      cudaFree(CUDA_ca);
      cudaFree(CUDA_cb);
      cudaFree(CUDA_portf);
      cudaFree(CUDA_time);

      return 0;
}

// column number of a must be the same as the column number of b
void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
      int i,j;
      for(i=0 ; i< m_row ; i++)
      {
            for(j=0 ; j< m_col ; j++)
            {
              *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
            }
      }
}
void vector_move(float *a, float *b , int ele_num , int start_e)
{
      int i;
      for(i=0; i<ele_num ; i++)
      {
```

```
            *(b+start_e+i) = *(a+i);
      }
}
//this function compute the square root of each element in vector a
void vector_squareroot(float *a , int ele_a)
{
      int i;
      float org;
      for(i=0; i<ele_a ; i++)
      {
         org = *(a+i);
         *(a+i) = sqrt(org);
      }
}
//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
      int i;
      for(i=0 ; i<a_col ; i++)
      {
            *(b + i) = *(a + a_row2 * a_col + i);
      }
}
```

cudaprog8main.cu

```
//(CUDA) Algorithm 7 , with U, DVw replacing L, w
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef DRAW
#define DRAW 10000
#endif

#ifndef M
#define M 16
#endif

#ifndef P
#define P 0.01
#endif

//this is used in kernel MatrixMulKernelv3
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHx2
```

```
#define TILE_WIDTHx2 16
#endif

//this is used in kernel MatrixMulKernelv3
#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern float matrix_row_l2norm( float* m, int row , int col, int
row_2);
extern float matrix_row_l2normv2( float* m, int m_row , int m_col, int
s_col , int e_col ,int row_2);
extern float matrix_col_l2norm(float* m, int row , int col , int
col_2);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
extern void matrix_cpy3(float* a, float* c, int a_row, int a_col, int
start_row , int end_row, int start_col , int end_col);
extern void matrix_multi(float* m , float* n , float* ans, int m_row ,
int m_col , int n_row , int n_col ,int ans_row , int ans_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) core for matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int
m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
```

```
    int ty = threadIdx.y;

    int Row = by * TILE_WIDTHy + ty;
    int Col = bx * TILE_WIDTHx + tx;

    float pvalue =0;

    for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
    {
          //loading data into shared memory
          Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
          Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
          __syncthreads();

          for(int j=0 ; j<TILE_WIDTHx ; j++)
          {
                pvalue += Mds[ty][j] * Nds[j][tx];
                __syncthreads();
          }

          *(pd + Row * m_col + Col) = pvalue;
    }

}
//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float
*pd , int m_row, int m_col , clock_t* time)
{
      __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
      __shared__ float Nds[TILE_WIDTHx2];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy2 + ty;
      //int Col = bx * TILE_WIDTHx + tx;
      if(ty==0) time[by]=clock();

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
        __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx2 ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

      if(ty==0) time[by + TEMP_AREA/TILE_WIDTHy2]=clock();
```

```
}
int main()
{
	FILE* fp_mu;
	FILE* fp_mv;
	FILE* fp_mw;

	FILE* fp_out;
	FILE* fp_mportf;

	float *m_u;
	float *m_v;
	float *m_w;
	float *m_portf;
	float *portfolio;
	float *m_pl2norm;
	float *dv_temp;
	float *m_dvw;
	float *m_utemp;
	float *m_btemp;
	float *CUDA_ca;
	float *CUDA_cb;
	float *CUDA_cc;
	float *CUDA_portf;
	float *CUDA_dvw;
	float *CUDA_utemp;
	float *CUDA_btemp;

	float sp = (float)DRAW * P;
	float wl;
	float w_l2norm;
	float vr_upperbound;
	float var;

	int to_buy=0;
	int not_to_buy=0;
	int mt_count=0;

	int size_CUDA_ca = M * M * sizeof(float);
	int size_CUDA_cb = M * M * sizeof(float);
	int size_CUDA_cc = M * M * sizeof(float);
	int size_CUDA_portf = M * sizeof(float);
	int size_CUDA_utemp = TEMP_AREA * M * sizeof(float);
	int size_CUDA_btemp = TEMP_AREA * sizeof(float);
	int size_CUDA_time = sizeof(clock_t) * (TEMP_AREA)/TILE_WIDTHy2 *
2;

	LARGE_INTEGER frequency;
	LARGE_INTEGER t1, t2;
	double elapsedTime;

	clock_t s_time;
	clock_t total_time;
	clock_t* CUDA_time;
	clock_t* time_used;
	time_used = new clock_t [(TEMP_AREA/TILE_WIDTHy2) * 2];
```

```
clock_t avg_clk=0;

m_u = new float [DRAW*M];
m_v = new float [M*M];
m_w = new float [M*M]; //Identity matrix
m_portf = new float [PORTF_NUM * M];
portfolio = new float [M];
m_pl2norm = new float [DRAW];
dv_temp = new float [M*M];
m_dvw = new float [M];
m_utemp = new float [TEMP_AREA * M];
m_btemp = new float [TEMP_AREA];

fp_mu = fopen("d:\\u_extract16.txt", "r");
fp_mv = fopen("d:\\matrixv16.txt", "r");
fp_mw = fopen("d:\\matrixw16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
fp_out = fopen("d:\\l2norm2.txt", "w");

//Readin data from the file
printf("Loading matrix U.\n");
loaddata(fp_mu , m_u , DRAW , M);
printf("Loading matrix V..\n");
loaddata(fp_mv , m_v , M,M);
printf("Loading matrix W...\n");
loaddata(fp_mw , m_w , M,M);
printf("Loading portfolio matrix.\n");
loaddata(fp_mportf , m_portf, PORTF_NUM , M);
printf("Loading complete.\n\n");

//===Initialize: Compute the L2 norm of each row of U, and let WL
//be the pth largest one.=======================
for(int i=0 ; i<DRAW ; i++)
{
 //only compute L2 norm for the first M element in a row...
 m_pl2norm[i] = matrix_row_l2normv2( m_u , DRAW , M , 0, M-1 ,i);
}

//quicksort
quicksort(m_pl2norm , 0, DRAW-1);

wl = m_pl2norm[DRAW - (int)sp];
//printf("%d %f\n", DRAW-(int)sp , wl);

//define threads per block and number of blocks
dim3 threadsperblock(TILE_WIDTHx , TILE_WIDTHy);
dim3 numBlocks(M/TILE_WIDTHx , M/TILE_WIDTHy);
//this is for monte carlo simulation
dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
 dim3 numBlocks2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

//compute D * V
cudaMalloc((void**) &CUDA_ca, size_CUDA_ca);
cudaMalloc((void**) &CUDA_cb, size_CUDA_cb);
cudaMalloc((void**) &CUDA_cc, size_CUDA_cc);

cudaMemcpy(CUDA_ca , m_w , size_CUDA_ca ,
```

```
cudaMemcpyHostToDevice);
	cudaMemcpy(CUDA_cb , m_v , size_CUDA_cb ,
cudaMemcpyHostToDevice);

	//Invoke CUDA core MatrixMulKernelv3 for matrix-matrix
	//multiplication
	MatrixMulKernelv3<<< numBlocks , threadsperblock >>>( CUDA_ca ,
CUDA_cb , CUDA_cc , M ,M);

	cudaMemcpy(dv_temp , CUDA_cc , size_CUDA_cc ,
cudaMemcpyDeviceToHost);

	//load first M column of U into m_utemp
	matrix_cpy3( m_u , m_utemp , DRAW, M, 0, DRAW-1 , 0 , M-1);

	//allocate resource for Monte Carlo simulation
	cudaMalloc((void**) &CUDA_utemp , size_CUDA_utemp);
	cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp);
	cudaMalloc((void**) &CUDA_time , size_CUDA_time);
	cudaMemcpy( CUDA_utemp , m_utemp , size_CUDA_utemp ,
cudaMemcpyHostToDevice);

	//=================================================================
	//Computation:
	cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
	cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf);

	printf("CUDA_Program8:");
	printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
	printf("DRAW=%d\n", DRAW );
	printf("M=%d\n", M);
	printf("P=%f\n", P);

	QueryPerformanceFrequency(&frequency);

	s_time = clock();
	QueryPerformanceCounter(&t1);
	for(int k=0 ; k<PORTF_NUM; k++)
	{
		portf_data_cpy( m_portf , portfolio, PORTF_NUM , M , k);

		matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1 ,
M ,1 );

		//compute L2 norm of D*V*w in O(M) time
		w_l2norm = matrix_col_l2norm( m_dvw, M,1,0);
		//printf("w_l2norm: %f\n", w_l2norm);


		//comput VR_upperbound
		vr_upperbound = wl * w_l2norm;
		//printf("VR_Upperbound: %f\n", vr_upperbound);

		//if VR_upperbound <= Vc then the answer to the decision
		//problem is No.
		//Otherwise the answer is not determined by this bound
```

```
            if( vr_upperbound <= COLLATERAL)
            {
                  to_buy++;
            }
            else
            {
                    //printf("Run Monte Carlo simulation\n");
                    for(int i=0; i< (DRAW/TEMP_AREA) ; i++)
                    {
                       cudaMemcpy( CUDA_dvw , m_dvw, size_CUDA_portf ,
cudaMemcpyHostToDevice );
                       //Invoke CUDA core MatrixVecotrMulKernel2 for
                       //matrix-vector multiplication
                       MatrixVectorMulKernel2<<< numBlocks2 ,
threadsperblock2 >>>(CUDA_utemp , CUDA_dvw , CUDA_btemp , TEMP_AREA ,
M , CUDA_time);

                       cudaMemcpy(m_pl2norm , CUDA_btemp ,
size_CUDA_btemp , cudaMemcpyDeviceToHost );

                    }

                    //quicksort
                    quicksort( m_pl2norm , 0 , DRAW-1);

                    var = m_pl2norm[DRAW - (int)sp];
                    //printf("VaR: %f \n", var);
                    if(var > COLLATERAL)
                    {
                        not_to_buy++;
                    }
                    else
                    {
                        to_buy++;
                    }

                    cudaMemcpy(time_used , CUDA_time , size_CUDA_time ,
cudaMemcpyDeviceToHost);

                    if(k==0)
                    {
                       for(int i=0 ; i < TEMP_AREA/TILE_WIDTHy2 ; i++)
                       {
                           avg_clk += *(time_used + i +
TEMP_AREA/TILE_WIDTHy2) - *(time_used + i);
                       }
                    }
                    mt_count++;
            }

      }
      QueryPerformanceCounter(&t2);
      total_time = clock() - s_time;

      elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;
```

```
	printf("\n%f ms.\n", elapsedTime);
	printf("Total clocks:%d \n\n", total_time);
	printf("VaR > Vc: %d\n", not_to_buy);
	printf("VaR <= Vc: %d\n", to_buy);


	//Free space
	delete [] m_u;
	delete [] m_v;
	delete [] m_w;
	delete [] m_portf;
	delete [] portfolio;
	delete [] m_pl2norm;
	delete [] dv_temp;
	delete [] m_dvw;
	delete [] m_utemp;
	delete [] m_btemp;
	delete [] time_used;

	cudaFree(CUDA_ca);
	cudaFree(CUDA_cb);
	cudaFree(CUDA_cc);
	cudaFree(CUDA_portf);
	cudaFree(CUDA_dvw);
	cudaFree(CUDA_utemp);
	cudaFree(CUDA_time);

	return 0;
}
//This function shows a vector on the screen
void show_vect(float *a , int ele_a)
{
   for(int i=0 ; i<ele_a ; i++)
   {
         printf("%f ", *(a+i));
   }
   printf("\n\n");
}
//Copy data from matrix a to matrix b
void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start)
{
	int i,j;
	for(i=0 ; i< m_row ; i++)
	{
		for(j=0 ; j< m_col ; j++)
		{
		    *(b + i*m_col + j) = *(a + (row_start+i)*m_col + j);
		}
	}

}

void vector_move(float *a, float *b , int ele_num , int start_e)
{
	int i;
	for(i=0; i<ele_num ; i++)
```

```
	{
		*(b+start_e+i) = *(a+i);
	}
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
	int i;

	for(i=0 ; i<a_col ; i++)
	{
	    *(b + i) = *(a + a_row2 * a_col + i);
	}

}
```

cudaprog9main.cu

```
//(CUDA) Algorithm 8
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
```

```
extern void quicksort(float* target, int left , int right);
//sort in descending order based on the second column
extern void quicksort2(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int
row_start);
extern void vector_move(float *a, float *b , int ele_num , int
start_e);
extern float matrix_row_l2norm(float* m, int row , int col , int
row_2 );
extern int binarysearch(float* m ,float target,  int start , int end);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int
col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2);

//(CUDA) core for matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int
m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
         //loading data into shared memory
         Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
         Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
         __syncthreads();

         //
         for(int j=0 ; j<TILE_WIDTHx ; j++)
         {
               pvalue += Mds[ty][j] * Nds[j][tx];
               __syncthreads();
         }

         *(pd + Row * m_col + Col) = pvalue;
   }

}



//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float
```

```
*pd , int m_row, int m_col)
{
      __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
      __shared__ float Nds[TILE_WIDTHx];

      //int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int Row = by * TILE_WIDTHy + ty;
      //int Col = bx * TILE_WIDTHx + tx;

      float pvalue=0;

      for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
      {
        Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
        Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
          __syncthreads();

        for(int j=0 ; j < TILE_WIDTHx ; j++)
        {
              pvalue += Mds[ty][j] * Nds[j];
        }

        *(pd + Row) = pvalue;
      }

}

int main()
{
      FILE* fp_ml;
      FILE* fp_mportf;

      float *m_loss;
      float *m_portf;
      float *portfolio;
      float *m_ll2norm;
      float *m_llonda;
      float *llonda_temp;
      float *barrow_temp;
      float *m_barrow;

      float *CUDA_ca;
      float *CUDA_cb;
      float *CUDA_portf;

      float w_l2norm = 0;
      float a1;
      float var=0;

      int sp;
      sp = (int)((float)DRAW * P);
      int index_s;
      int mt_count=0;
```

```
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = TEMP_AREA * M * sizeof(float);
int size_CUDA_cb = TEMP_AREA * sizeof(float);
int size_CUDA_portf = M * sizeof(float);

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;

m_loss = new float [DRAW*M];
m_portf = new float [PORTF_NUM * M];
portfolio = new float [M];
m_ll2norm = new float [2*DRAW];
m_llonda = new float [DRAW*M];
llonda_temp = new float [TEMP_AREA*M];
barrow_temp = new float [TEMP_AREA];
m_barrow = new float [DRAW];

fp_ml = fopen("d:\\loss16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");

printf("Loading data for matrix L:\n");
  loaddata( fp_ml , m_loss, DRAW, M);
loaddata( fp_mportf , m_portf, PORTF_NUM , M);

dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
dim3 numBlocks( M/TILE_WIDTHx , TEMP_AREA/TILE_WIDTHy );

cudaMalloc((void**) &CUDA_ca , size_CUDA_ca );
cudaMalloc((void**) &CUDA_cb , size_CUDA_cb );
cudaMalloc((void**) &CUDA_portf , size_CUDA_portf );

//======================Initialize:============================
//1.compute the L2 norm of each row of L
for(int i=0 ; i<DRAW ; i++)
{
     //the first column indicate the order number of the
     //element
     *(m_ll2norm+ 2*i )=(float)i;
     //the second column contain L2 norm of each row
     *(m_ll2norm+(2*i+1)) = matrix_row_l2norm(m_loss, DRAW,M, i);
}

//quicksort: sort it into descending order
quicksort2( m_ll2norm , 0, DRAW-1 );

//copy data in m_loss into m_llonda based on the order of its L2
//norm
matrix_cpy2( m_loss , m_ll2norm , m_llonda , DRAW , M);

//move data into device
cudaMemcpy( CUDA_ca, m_loss, size_CUDA_ca,
```

```
cudaMemcpyHostToDevice);
      //============================================================
      printf("CUDA_Program9:");
      printf("Running %d portfolios with COLLATERAL=%f...\n",
PORTF_NUM, COLLATERAL);
      printf("DRAW=%d\n", DRAW );
      printf("M=%d\n", M);
      printf("P=%f\n", P);

      QueryPerformanceFrequency(&frequency);

      s_time = clock();
      QueryPerformanceCounter(&t1);
      for(int k=0 ; k<PORTF_NUM ; k++)
      {
           index_s = 0;
           portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);

           //compute L2 norm of w
           w_l2norm = matrix_row_l2norm( portfolio , 1, M , 0);
           //printf("%f \n", w_l2norm);

           a1 = ((float)COLLATERAL)/w_l2norm;
           //printf("a1: %f\n", a1);

           //determine the largest index s such that ls > |Vc|/(L2
           //norm of portfolio matrix)
           //if there is no such component, then set s = 0
           index_s = binarysearch( m_ll2norm ,a1, 0 , DRAW-1);
           //printf("\nindex_s: %d\n\n" , index_s);

           if(index_s < sp)
           {
                to_buy++;
           }
           else
           {
                //otherwise Run Monte Carlo simulation
                cudaMemcpy( CUDA_portf , portfolio , size_CUDA_portf ,
cudaMemcpyHostToDevice );
                //Invoke CUDA core MatrixVecotrMulKernel for matrix-
                //vector multiplication
                MatrixVectorMulKernel<<< numBlocks , threadsperblock
>>>( CUDA_ca , CUDA_portf , CUDA_cb , DRAW , M);

                cudaMemcpy( m_barrow , CUDA_cb , size_CUDA_cb ,
cudaMemcpyDeviceToHost);
                //quicksort in ascending order
                quicksort( m_barrow , 0 , DRAW-1);

                var = *(m_barrow + (DRAW-sp));
                //printf("VaR: %f\n", var);

                if(var > COLLATERAL)
                {
                     not_to_buy++;
                }
```

```
                else
                {
                     to_buy++;
                }

                mt_count++;
           }

      }
      QueryPerformanceCounter(&t2);
      total_time = clock() - s_time;

      elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;

      printf("\n%f ms.\n", elapsedTime);
      printf("Total clocks: %d \n\n", total_time);

      printf("VaR > Vc: %d\n", not_to_buy);
      printf("VaR <= Vc: %d\n", to_buy);

      //Free space
      delete [] m_loss;
      delete [] m_portf;
      delete [] portfolio;
      delete [] m_ll2norm;
      delete [] m_llonda;
      delete [] llonda_temp;
      delete [] barrow_temp;
      delete [] m_barrow;

      cudaFree(CUDA_ca);
      cudaFree(CUDA_portf);

      return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int
a_row2)
{
      int i;

      for(i=0 ; i<a_col ; i++)
      {
          *(b + i) = *(a + a_row2 * a_col + i);
      }

}
```

cudaprog10main.cu

```
//20110324 (CUDA) Algorithm 8 with U, DVw replacing L, w
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef DRAW
#define DRAW 10000
#endif

#ifndef M
#define M 16
#endif

#ifndef P
#define P 0.01
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif
//this is used by kernel MatrixMulKernelv3
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif
//this is used by kernel MatrixVectorMulKernel2
#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif
//this is used by kernel MatrixMulKernelv3
#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif
//this is used by kernel MatrixVectorMulKernel2
#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
```

```
extern void quicksort(float* target, int left , int right);
//sort in descending order based on the second column of target
extern void quicksort2(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float matrix_row_l2norm(float* m, int row , int col , int row_2 );
extern int binarysearch(float* m ,float target,  int start , int end);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int col);
extern void matrix_cpy2v2(float* a, float* b, float* c, int a_row, int a_col, int start_row ,
int end_row, int start_col , int end_col);
extern void matrix_cpy3(float* a, float* c, int a_row, int a_col, int start_row , int end_row,
int start_col , int end_col);
extern float matrix_col_l2norm(float* m, int row , int col , int col_2 );
extern float matrix_row_l2normv2( float* m, int m_row , int m_col, int s_col , int e_col ,int
row_2);
extern void matrix_multi(float* m , float* n , float* ans, int m_row , int m_col , int n_row ,
int n_col ,int ans_row , int ans_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2);

//(CUDA) core for matirx-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
           //loading data into shared memory
           Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
           Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
           __syncthreads();

           for(int j=0 ; j<TILE_WIDTHx ; j++)
           {
                   pvalue += Mds[ty][j] * Nds[j][tx];
                   __syncthreads();
           }

           *(pd + Row * m_col + Col) = pvalue;
   }

}
```

```
//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
        __shared__ float Nds[TILE_WIDTHx2];

        //int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx + tx;

        float pvalue=0;

        for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
        {
            Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
            Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));  //size change
            __syncthreads();

            for(int j=0 ; j < TILE_WIDTHx2 ; j++) //size change
            {
                    pvalue += Mds[ty][j] * Nds[j];
            }

            *(pd + Row) = pvalue;
        }
}

int main()
{

        FILE* fp_mu;
        FILE* fp_mv;
        FILE* fp_mw;
        FILE* fp_mportf;
        FILE* fp_out;

        float *m_u;
        float *m_v;
        float *m_w;
        float *m_portf;
        float *portfolio;
        float *m_ul2norm;
        float *dv_temp;
        float *m_dvw;
        float *m_utemp;

        float *barrow_temp;
        float *m_barrow;

        float *CUDA_ca;
        float *CUDA_cb;
```

```
float *CUDA_cc;
float *CUDA_portf;
float *CUDA_dvw;
float *CUDA_utemp;
float *CUDA_btemp;

float dvw_l2norm = 0;
float a1;
float var=0;

int sp;

sp = (int)((float)DRAW * P);

int index_s;
//int counter=0;
int mt_count=0;
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = M * M * sizeof(float);
int size_CUDA_cb = M * M * sizeof(float);
int size_CUDA_cc = M * M * sizeof(float);
int size_CUDA_portf = M * sizeof(float);
int size_CUDA_utemp = TEMP_AREA * M * sizeof(float);
int size_CUDA_btemp = TEMP_AREA * sizeof(float);

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;

m_u = new float [DRAW*M];
m_v = new float [M*M];
m_w = new float [M*M]; //identity matrix
m_portf = new float [ PORTF_NUM * M ];
portfolio = new float [M];
m_ul2norm = new float [2*DRAW];
dv_temp = new float [M * M];
m_dvw = new float [M];
m_utemp = new float [TEMP_AREA*M];

barrow_temp = new float [TEMP_AREA];
m_barrow = new float [DRAW];

fp_mu = fopen("d:\\u_extract16.txt", "r");
fp_mv = fopen("d:\\matrixv16.txt", "r");
fp_mw = fopen("d:\\matrixw16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");

printf("Loading data for matrix U:\n");
loaddata( fp_mu , m_u , DRAW, M);
printf("Loading data for matrix V:\n");
```

```
        loaddata( fp_mv , m_v , M , M);
        printf("Loading data for matrix W:\n");
        loaddata( fp_mw , m_w , M , M);
        printf("Loading data for portfolio matrxi.\n\n");
        loaddata( fp_mportf , m_portf, PORTF_NUM , M);

        //declare GPU space for use
        cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf );
        cudaMalloc((void**) &CUDA_portf , size_CUDA_portf );
        cudaMalloc((void**) &CUDA_utemp , size_CUDA_utemp );
        cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp );

        //=========================Initialize:================================
        //1.compute the L2 norm of each row of U
        for(int i=0 ; i<DRAW ; i++)
        {
               //the first column indicate the order number of the element
               *(m_ul2norm+ 2*i )=(float)i;
               //the second column contain L2 norm for the first M elements in each row
               *(m_ul2norm+(2*i+1)) = matrix_row_l2normv2(m_u , DRAW, M, 0 , M-1 , i);
        }

        //quicksort: sort it into descending order
        quicksort2( m_ul2norm , 0, DRAW-1 );

        dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
        dim3 numBlocks( M/TILE_WIDTHx , M/TILE_WIDTHy );

        //Compute D*V
        cudaMalloc((void**) &CUDA_ca , size_CUDA_ca );
        cudaMalloc((void**) &CUDA_cb , size_CUDA_cb );
        cudaMalloc((void**) &CUDA_cc , size_CUDA_cc );
        cudaMemcpy( CUDA_ca, m_w , size_CUDA_ca , cudaMemcpyHostToDevice);
        cudaMemcpy( CUDA_cb, m_v , size_CUDA_cb , cudaMemcpyHostToDevice);

        //Invoke MatrixMulKernelv3 for matrix-matrix multiplication
        MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca , CUDA_cb , CUDA_cc ,
M ,M );
        cudaMemcpy( dv_temp , CUDA_cc , size_CUDA_cc , cudaMemcpyDeviceToHost );

        //copy the first M column of U into m_utemp
        matrix_cpy3(m_u , m_utemp , DRAW, M , 0 , DRAW-1 , 0 , M-1);

        dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
        dim3 numBlocks2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

        cudaMemcpy( CUDA_utemp  , m_utemp , size_CUDA_utemp , cudaMemcpyHostToDevice);
        //=============================================================================
        printf("CUDA_Program10:");
        printf("Running %d portfolios with COLLATERAL=%f...\n", PORTF_NUM, COLLATERAL);
        printf("DRAW=%d\n", DRAW );
        printf("M=%d\n", M);
        printf("P=%f\n", P);

        QueryPerformanceFrequency(&frequency);
```

```
        s_time = clock();
        QueryPerformanceCounter(&t1);
        for(int k=0 ; k<PORTF_NUM ; k++)
        {
                 portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);

                 //compute D*V*w
                 matrix_multi( dv_temp , portfolio, m_dvw, M, M , M, 1 , M, 1);

                 //compute L2 norm of DVw
                 dvw_l2norm = matrix_col_l2norm( m_dvw , M, 1 ,0);
                 //printf("%f \n", dvw_l2norm);

                 a1 = ((float)COLLATERAL)/ dvw_l2norm;
                 //printf("a1: %f\n", a1);

                //determine the largest index s such that ls > |Vc|/(L2 norm of portfolio
                //matrix)
                //if there is no such component, then set s = 0
                 index_s = binarysearch( m_ul2norm ,a1, 0 , DRAW-1);
                 //printf("\nindex_s: %d\n" , index_s);

                 if(index_s < sp)
                 {
                      to_buy++;
                 }
                 else
                 {
                      //otherwise run Monte Carlo simulation
                      cudaMemcpy( CUDA_dvw, m_dvw , size_CUDA_portf , cudaMemcpyHostToDevice );
                      //Invoke CUDA core MatrixVectorMulKernel2 for matrix-vector multiplication
                      MatrixVectorMulKernel2<<< numBlocks2 , threadsperblock2 >>>(CUDA_utemp ,
CUDA_dvw , CUDA_btemp , DRAW , M);

                      cudaMemcpy( m_barrow , CUDA_btemp , size_CUDA_btemp ,
cudaMemcpyDeviceToHost);
                      //quicksort in ascending order
                      quicksort( m_barrow , 0 , DRAW-1 );

                      var = *(m_barrow + (DRAW-sp));
                      //printf("VaR: %f\n", var);

                      if(var > COLLATERAL)
                      {
                            not_to_buy++;
                      }
                      else
                      {
                            to_buy++;
                      }
                      mt_count++;

                 }

        }
```

```
        QueryPerformanceCounter(&t2);
        total_time = clock() - s_time;

        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;

        printf("\n%f ms.\n", elapsedTime);
        printf("Total clocks:%d\n\n", total_time);

        printf("VaR > Vc: %d\n", not_to_buy);
        printf("VaR <= Vc: %d\n", to_buy);

        //Free space
        delete [] m_u;
        delete [] m_v;
        delete [] m_w;
        delete [] m_portf;
        delete [] portfolio;
        delete [] m_ul2norm;
        delete [] dv_temp;
        delete [] m_dvw;
        delete [] m_utemp;

        delete [] barrow_temp;
        delete [] m_barrow;

        cudaFree(CUDA_ca);
        cudaFree(CUDA_cb);
        cudaFree(CUDA_cc);
        cudaFree(CUDA_dvw);
        cudaFree(CUDA_portf);
        cudaFree(CUDA_utemp);
        cudaFree(CUDA_btemp);

        return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2)
{
        int i;
        for(i=0 ; i<a_col ; i++)
        {
           *(b + i) = *(a + a_row2 * a_col + i);
        }
}
```

cudaprog11main.cu

```
//(CUDA) Algorithm 9 with h = 0.05 * DRAW
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 1000
#endif

#ifndef TEMP_AREA2
#define TEMP_AREA2 10000
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float matrix_row_l2norm(float* m, int row , int col , int row_2 );
extern void quicksort2(float* target, int left , int right);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2);

//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx];
        __shared__ float Nds[TILE_WIDTHx];
```

```
        int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx + tx;

        float pvalue=0;

        for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
        {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx + tx));
          __syncthreads();

          for(int j=0 ; j < TILE_WIDTHx ; j++)
          {
                  pvalue += Mds[ty][j] * Nds[j];
          }

          *(pd + Row) = pvalue;
        }

}


int main()
{
        FILE* fp_ml;
        FILE* fp_mportf;
        FILE* fp_out;

        float *m_loss;
        float *m_portf;
        float *portfolio;
        float *m_l12norm;
        float *m_larrow;

        float *m_b_temp;
        float *m_ltemp;
        float *m_b_temp2;
        float *m_b;


        float *CUDA_ca;
        float *CUDA_cb;
        float *CUDA_cc;
        float *CUDA_ltemp;
        float *CUDA_btemp;

        float sp = (float)DRAW * P;
        float var=0;
        //Define the size of h. Ex: 0.1*10000 = 1000
        int h = (int)(0.1 * (float)DRAW);
```

```
int lq=0;
int mt_count=0;
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = TEMP_AREA * M * sizeof(float);
int size_CUDA_cb = M * sizeof(float);
int size_CUDA_cc = TEMP_AREA * sizeof(float);
int size_CUDA_ltemp = TEMP_AREA2 * M *sizeof(float);
int size_CUDA_btemp = TEMP_AREA2 * sizeof(float);

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;

m_loss = new float [DRAW*M];
m_portf = new float [PORTF_NUM * M];
portfolio = new float [M];
m_l12norm = new float [2*DRAW];
//This contain the h rows of loss matrix with large L2 norm
m_larrow = new float [h*M];
m_b_temp = new float [TEMP_AREA];
m_ltemp = new float [TEMP_AREA2 * M];
m_b_temp2 = new float [TEMP_AREA2];
m_b = new float [DRAW];

fp_ml = fopen("d:\\loss16.txt", "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
//Readin data from the file
printf("Loading data for loss matrix, please wait...\n");
loaddata( fp_ml, m_loss , DRAW , M);
loaddata( fp_mportf, m_portf , PORTF_NUM , M);

//======================   Initialize:   ======================================
//Compute the L2 norm of each row of L, and let L_arrow be the matrix consisting of the
//h rows with the largest L2 norm, where h is a fixed constant greater than p
for(int i=0 ; i<DRAW ; i++)
{
        //The first column is the order number of the row
        *(m_l12norm + 2*i) = (float)i;
        //The second column is the L2 norm of the row
        *(m_l12norm + (2*i+1)) = matrix_row_l2norm(m_loss , DRAW, M , i);
}

//quicksort in descending order
quicksort2(m_l12norm , 0, DRAW-1);

//copy h rows of loss matrix into m_larrow based on its L2 norm
matrix_cpy2(m_loss , m_l12norm, m_larrow , h , M);

dim3 threadsperblock( TILE_WIDTHx, TILE_WIDTHy2 );
dim3 numBlock( M/TILE_WIDTHx , h/TILE_WIDTHy2 );
```

```
        dim3 threadsperblock2( TILE_WIDTHx , TILE_WIDTHy2 );
        dim3 numBlock2( M/TILE_WIDTHx , TEMP_AREA2/TILE_WIDTHy2 );

        cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
        cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);
        cudaMalloc((void**) &CUDA_cc , size_CUDA_cc);
        cudaMalloc((void**) &CUDA_ltemp , size_CUDA_ltemp);
        cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp);

        //copy data m_larrow from CPU to GPU
        cudaMemcpy( CUDA_ca , m_larrow , size_CUDA_ca , cudaMemcpyHostToDevice );

        //move m_loss from host(CPU) to device(GPU)
        cudaMemcpy( CUDA_ltemp , m_loss , size_CUDA_ltemp , cudaMemcpyHostToDevice );
        //=============================================================================

        printf("CUDA_Program11:");
        printf("Running %d portfolios with COLLATERAL=%f...\n", PORTF_NUM, COLLATERAL);
        printf("DRAW=%d\n", DRAW );
        printf("M=%d\n", M);
        printf("P=%f\n", P);
        printf("h=%d\n", h);

        QueryPerformanceFrequency(&frequency);

        s_time = clock();
        QueryPerformanceCounter(&t1);
        for(int k=0 ; k<PORTF_NUM ; k++)
        {
                //take on row of m_portf and store it in vector portfolio
                portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);
                //copy portfolio data from CPU to GPU
                cudaMemcpy( CUDA_cb , portfolio, size_CUDA_cb , cudaMemcpyHostToDevice );

                //comput L_arrow * w in O(hM) time
                for(int i=0 ; i< (h/TEMP_AREA) ; i++)
                {
                        //Invoke CUDA core MatrixVectorMulKernel2 to do matrix-vector
                        //multiplication
                        MatrixVectorMulKernel2<<< numBlock , threadsperblock >>>( CUDA_ca,
CUDA_cb , CUDA_cc , TEMP_AREA , M);
                        //copy the result from GPU to CPU
                        cudaMemcpy( m_b_temp , CUDA_cc , size_CUDA_cc ,
cudaMemcpyDeviceToHost );

                        //compute the number of elements in L_arrow * w that are greater than
                        //COLLATERAL
                        for(int i=0 ; i<TEMP_AREA ; i++)
                        {
                            if( *(m_b_temp+i) > COLLATERAL )
                            {
                                    lq++;
                            }
                            else
                            {
```

```
                        }
                }

        }

         //If the number of elements in L_arrow * w that are greater than COLLATERAL
         //exceed sp, answer the decision problem with Yes.
        if(lq >= sp)
        {
                not_to_buy++;
        }
        else
        {
                //Monte carlo simulation
                lq=0; //reset lq
                mt_count++;

                for(int i=0 ; i< (int)(DRAW/TEMP_AREA2) ; i++)
                {
                        //Invoke MatrixVectorMulKernel2 to do matrix-vector multiplication
                        MatrixVectorMulKernel2<<< numBlock2 , threadsperblock2
>>>(CUDA_ltemp , CUDA_cb , CUDA_btemp , TEMP_AREA2 , M);

                        //copy the result form GPU to CPU
                        cudaMemcpy(m_b , CUDA_btemp , size_CUDA_btemp ,
cudaMemcpyDeviceToHost );

                }

                //quicksort in ascending order
                quicksort( m_b , 0 , DRAW-1);

                var = m_b[ DRAW - (int)sp ];
                //printf("This is VaR: %f\n\n", var);

                if( var > COLLATERAL )
                {
                        //Answer Yes to the decision problem
                        not_to_buy++;
                }
                else
                {
                        //Amswer No to the decision problem
                        to_buy++;
                }

        }
        lq=0;

    }
    QueryPerformanceCounter(&t2);
    total_time = clock() - s_time;

    elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
```

```
        printf("\n%f ms.\n", elapsedTime);
        printf("Total clocks:%d \n\n", total_time);
        printf("VaR > Vc: %d\n", not_to_buy);
        printf("VaR <= Vc: %d\n", to_buy);

        //Free space
        delete [] m_loss;
        delete [] m_portf;
        delete [] portfolio;
        delete [] m_ll2norm;
        delete [] m_larrow;

        delete [] m_b_temp;
        delete [] m_ltemp;
        delete [] m_b_temp2;
        delete [] m_b;

        cudaFree(CUDA_ca);
        cudaFree(CUDA_cb);
        cudaFree(CUDA_cc);
        cudaFree(CUDA_ltemp);
        cudaFree(CUDA_btemp);

        return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2)
{
        int i;

        for(i=0 ; i<a_col ; i++)
        {
            *(b + i) = *(a + a_row2 * a_col + i);
        }
}
```

cudaprog12main.cu

```
//Algorithm 9 with U, DVw replacing L,w and h = 0.05N
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"
//define how many rows for loss matrix
#ifndef DRAW
#define DRAW 10000
#endif
```

```
//define how many columns for loss matrix
#ifndef M
#define M 16
#endif
//level of confidence
#ifndef P
#define P 0.01
#endif

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 10000
#endif
//define number of portfolios
#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void quicksort2(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float matrix_row_l2norm(float* m, int row , int col , int row_2 );
extern float matrix_row_l2normv2( float* m, int m_row , int m_col, int s_col , int e_col ,int
row_2);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int col);
extern void matrix_cpy2v2(float* a, float* b, float* c, int a_row, int a_col, int start_row ,
int end_row, int start_col , int end_col);
extern void matrix_multi(float* m , float* n , float* ans, int m_row , int m_col , int n_row ,
int n_col ,int ans_row , int ans_col);
extern float matrix_innerproduct( float *m, float *n , int m_row , int m_col , int n_ele , int
m_row2); //
```

```
extern void vector_move(float *a, float *b , int ele_num , int start_e);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2);

//(CUDA) core for doing matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx]; // initial value = 0
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx]; // initial value = 0

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
           //loading data into shared memory
           Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );

           Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
           __syncthreads();

           //
           for(int j=0 ; j<TILE_WIDTHx ; j++)
           {
                   pvalue += Mds[ty][j] * Nds[j][tx];
                   __syncthreads();
           }
           //

           *(pd + Row * m_col + Col) = pvalue;
   }

}
//(CUDA) core for doing matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
        __shared__ float Nds[TILE_WIDTHx2];

        //int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx2 + tx;
        float pvalue=0;
```

```
        for(int i=0 ; i< (m_col/TILE_WIDTHx2) ; i++)
        {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
          __syncthreads();

          for(int j=0 ; j < TILE_WIDTHx2 ; j++)
          {
                  pvalue += Mds[ty][j] * Nds[j];
          }

          *(pd + Row) = pvalue;
        }

}

int main()
{
        FILE* fp_mu;
        FILE* fp_mv;
        FILE* fp_mw;  //identity matrix
        FILE* fp_mportf;

        FILE* fp_out;

        float *m_u;
        float *m_v;
        float *m_w;
        float *m_portf;
        float *portfolio;
        float *u_l2norm;
        float *u_larrow;
        float *dv_temp;
        float *m_dvw;
        float *p_profit;
        float *m_utemp1;
        float *m_utemp2;
        float *m_btemp1;
        float *m_btemp2;
        float *m_b;

        float *CUDA_ca;
        float *CUDA_cb;
        float *CUDA_cc;
        float *CUDA_portf;
        float *CUDA_dvw;

        float *CUDA_utemp1;
        float *CUDA_btemp1;
        float *CUDA_utemp2;
        float *CUDA_btemp2;
        //define h. In this case h=1000.
        int h = (int)(0.1 * DRAW);
        int sp = (int)((float)DRAW * P);
        int to_buy=0;
```

```
int not_to_buy=0;
int lq = 0;
int mt_count=0;

int size_CUDA_ca = M * M * sizeof(float);
int size_CUDA_cb = M * M * sizeof(float);
int size_CUDA_cc = M * M * sizeof(float);
int size_CUDA_portf = M * sizeof(float);
int size_CUDA_utempl = h * M * sizeof(float);
int size_CUDA_btempl = h * sizeof(float);
int size_CUDA_utemp2 = (DRAW-h) * M *sizeof(float);
int size_CUDA_btemp2 = (DRAW-h) * sizeof(float);

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;

//U matrix. We need only the first M column.
m_u = new float [DRAW*M];
m_v = new float [M*M];
//Identity matrix
m_w = new float [M*M];
m_portf = new float [ PORTF_NUM * M];
portfolio = new float [M];
u_l2norm = new float [2*DRAW];
u_larrow = new float [TEMP_AREA * M];
dv_temp = new float [M*M];
m_dvw = new float [M];
p_profit = new float [h];
m_utempl = new float [ h * M ];
ini_fmatrix(m_utempl , h , M);
m_utemp2 = new float [(DRAW-h) * M];
ini_fmatrix(m_utemp2 , (DRAW-h), M);

m_btempl = new float [ h ];
m_btemp2 = new float [DRAW-h];
m_b = new float [DRAW];

fp_mu = fopen("d:\\u_extract16.txt" , "r");
fp_mv = fopen("d:\\matrixv16.txt" , "r");
fp_mw = fopen("d:\\matrixw16.txt" , "r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");

//loading data
printf("Loading matrix U.\n");
loaddata(fp_mu , m_u , DRAW , M);
printf("Loading matrix V..\n");
loaddata(fp_mv , m_v , M, M);
printf("Loading matrix W...\n");
loaddata(fp_mw , m_w , M, M);
printf("Loadint protfolio matrix..\n");
loaddata(fp_mportf , m_portf , PORTF_NUM , M);
```

```
        //================================Initialization:=================================
        //comput each row's L2 norm of U
        for(int i=0 ; i<DRAW; i++)
        {
                //Ths first column is the order number of the row
                *(u_l2norm+ 2*i) = (float)i;
                //only compute L2 norm for the first M elements of row in U
                *(u_l2norm+ (2*i+1)) = matrix_row_l2normv2( m_u , DRAW , M ,0, M-1, i );
        }

        //quicksort in descending order
        quicksort2(u_l2norm , 0 , DRAW-1);

        //copy data of m_u into u_larrow based on its L2 norm
        //only copy the first M column of m_u into u_larrow
        matrix_cpy2v2( m_u , u_l2norm , u_larrow , DRAW, M, 0, DRAW-1 , 0, M-1 );

        dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
        dim3 numBlocks( M/TILE_WIDTHx , M/TILE_WIDTHy );

        cudaMalloc((void**) &CUDA_ca, size_CUDA_ca);
        cudaMalloc((void**) &CUDA_cb, size_CUDA_cb);
        cudaMalloc((void**) &CUDA_cc, size_CUDA_cc);
        //compte D * V
        cudaMemcpy(CUDA_ca , m_w , size_CUDA_ca , cudaMemcpyHostToDevice);
        cudaMemcpy(CUDA_cb , m_v , size_CUDA_cb , cudaMemcpyHostToDevice);
        //Invoke MatrixMulKernelv3 for matrix-matrix multiplication
        MatrixMulKernelv3<<< numBlocks , threadsperblock >>>( CUDA_ca , CUDA_cb , CUDA_cc ,
M ,M);

        cudaMemcpy(dv_temp , CUDA_cc , size_CUDA_cc , cudaMemcpyDeviceToHost);
        cudaMemcpy(CUDA_cc , dv_temp , size_CUDA_cc , cudaMemcpyHostToDevice);

        //
        dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
        dim3 numBlocks2( M/TILE_WIDTHx2 , h/TILE_WIDTHy2 );

        dim3 threadsperblock3( TILE_WIDTHx2 , TILE_WIDTHy2 );
        dim3 numBlocks3( M/TILE_WIDTHx2 , (DRAW-h)/TILE_WIDTHy2 );

        cudaMalloc((void**) &CUDA_utemp1 , size_CUDA_utemp1);
        cudaMalloc((void**) &CUDA_btemp1 , size_CUDA_btemp1);

        cudaMalloc((void**) &CUDA_utemp2 , size_CUDA_utemp2);
        cudaMalloc((void**) &CUDA_btemp2 , size_CUDA_btemp2);

        //move first h row (0 ~ (h-1))of U into m_utemp1
        matrix_move( u_larrow , m_utemp1 , h , M, 0);
        cudaMemcpy( CUDA_utemp1 , m_utemp1 , size_CUDA_utemp1 , cudaMemcpyHostToDevice );

        //move the rest of the row (h~(DRAW-1))to m_utemp2
        matrix_move( u_larrow , m_utemp2 , DRAW-h , M, h);
        cudaMemcpy( CUDA_utemp2 , m_utemp2 , size_CUDA_utemp2 , cudaMemcpyHostToDevice );
```

```
        //=========================================================================

        //cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
        cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf);

        printf("CUDA_Program12:");
        printf("Running %d portfolios with COLLATERAL=%f...\n", PORTF_NUM, COLLATERAL);
        printf("DRAW=%d\n", DRAW );
        printf("M=%d\n", M);
        printf("P=%f\n", P);
        printf("h=%d\n", h);

        QueryPerformanceFrequency(&frequency);

        s_time = clock();
        QueryPerformanceCounter(&t1);
        for(int k=0 ; k<PORTF_NUM ; k++)
        {
            portf_data_cpy(m_portf , portfolio, PORTF_NUM , M , k);

            //compute D * V * w
            matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1, M, 1 );

            //compute the number of components of ^L * w that are greater than Vc in O(h) time
            for(int i=0 ; i<1 ; i++)
            {
                    cudaMemcpy( CUDA_dvw , m_dvw , size_CUDA_portf , cudaMemcpyHostToDevice );
                    //Invoke MatrixVectorMulKernel to do matrix-vector multiplication
                    MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2 >>>( CUDA_utemp1 ,
CUDA_dvw , CUDA_btemp1 , h , M);

                    cudaMemcpy( m_btemp1 , CUDA_btemp1 , size_CUDA_btemp1 ,
cudaMemcpyDeviceToHost );

                    for(int j=0 ; j<h ; j++)
                    {
                            if( *(m_btemp1+j) > COLLATERAL)
                            {
                                    lq++;
                            }

                    }

            }

            //if q >= p, then the answer is Yes.
            //Otherwise the answer could not be determined by this procedure, ans so return
            //Undecided
            if( lq >= sp)
            {
                    not_to_buy++;
            }
            else  //if it fails to give an answer, compute the rest of the (DRAW-h) rows
            {
                    mt_count++;
```

```
                        //Invoke MatrixVectorMulKernel to do matrix-vector multiplication
                        MatrixVectorMulKernel<<< numBlocks3 , threadsperblock3 >>>(CUDA_utemp2 ,
CUDA_dvw , CUDA_btemp2 , (DRAW-h), M );

                        cudaMemcpy( m_btemp2 , CUDA_btemp2, size_CUDA_btemp2 ,
cudaMemcpyDeviceToHost);

                        for(int i=0 ; i<(DRAW-h) ; i++)
                        {
                                if( *(m_btemp2+i) > COLLATERAL)
                                {
                                        lq++;
                                }
                        }

                        //If the number of elements in ^L * w is greater than sp, answer Yes.
                        //else, No.
                        if(lq >= sp)
                        {
                            not_to_buy++;
                        }
                        else
                        {
                            to_buy++;
                        }
            }

            //reset lq
            lq =0;
        }
        QueryPerformanceCounter(&t2);
        total_time = clock() - s_time;

        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;

        printf("\n%f ms.\n" , elapsedTime);
        printf("Total clocks:%d \n\n", total_time );

        printf("VaR > Vc: %d\n", not_to_buy);
        printf("VaR <= Vc: %d\n", to_buy);


        //Free space
        delete [] m_u;
        delete [] m_v;
        delete [] m_w;
        delete [] m_portf;
        delete [] portfolio;
        delete [] u_l2norm;
        delete [] u_larrow;
        delete [] dv_temp;
        delete [] m_dvw;
        delete [] p_profit;
        delete [] m_utemp1;
        delete [] m_utemp2;
```

```
        delete [] m_btemp1;
        delete [] m_btemp2;
        delete [] m_b;

        cudaFree(CUDA_ca);
        cudaFree(CUDA_cb);
        cudaFree(CUDA_cc);
        //cudaFree(CUDA_portf);
        cudaFree(CUDA_dvw);
        cudaFree(CUDA_utemp1);
        cudaFree(CUDA_btemp1);
        cudaFree(CUDA_utemp2);
        cudaFree(CUDA_btemp2);

        return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2)
{
        int i;

        for(i=0 ; i<a_col ; i++)
        {
            *(b + i) = *(a + a_row2 * a_col + i);
        }

}
```

cudaprog13main.cu

```
//(CUDA) Algorithm 10, where the L1 of algorithm 5 is used in step 1 (with U,DVw replacing L,w)
//and the algorithm in algorithm 1 is used insteead in step 5
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef PORTF_NUM
#define PORTF_NUM 10000
#endif
#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
```

```
#endif
//this is used by kernel MatrixVectorMulKernel,  MatrixVectorMulKernel2
#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif
//this is used by kernel MatrixVectorMulKernel, MatrixVectorMulKernel2
#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif TILE_WDITHy2

#ifndef TEMP_ROW1
#define TEMP_ROW1 1000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void quicksort2(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_col_cpy(float* target, float* vect, int row_t, int col_t, int col_i);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float vector_sum2(float* a, int start, int end);
extern void vector_ele_multi(float* a, float* b, float* ans, int ele_a);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int col);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2);
//(CUDA) kernel for doing matrix-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
           //loading data into shared memory
           Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
           Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
           __syncthreads();
```

```
            //
            for(int j=0 ; j<TILE_WIDTHx ; j++)
            {
                    pvalue += Mds[ty][j] * Nds[j][tx];
                    __syncthreads();
            }
            //
            *(pd + Row * m_col + Col) = pvalue;
    }

}

//(CUDA) kernel for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
        __shared__ float Nds[TILE_WIDTHx2];

        //int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx + tx;

        float pvalue=0;

        for(int i=0 ; i< (m_col/TILE_WIDTHx2)+1 ; i++)
        {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
          __syncthreads();

          for(int j=0 ; j < TILE_WIDTHx2 ; j++)
          {
                  pvalue += Mds[ty][j] * Nds[j];
          }

          *(pd + Row) = pvalue;
        }

}

//(CUDA) core for matrix-vector multiplication
__global__ void MatrixVectorMulKernel2(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
        __shared__ float Nds[TILE_WIDTHx2];

        //int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;
```

```
        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx + tx;

        float pvalue=0;

        for(int i=0 ; i< (m_col/TILE_WIDTHx2)+1 ; i++)
        {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
          __syncthreads();

          for(int j=0 ; j < TILE_WIDTHx2 ; j++)
          {
                  pvalue += Mds[ty][j] * Nds[j];
          }

          *(pd + Row) = pvalue;
        }

}

int main()
{
        FILE* fp_ml;
        FILE* fp_mu;
        FILE* fp_mv;
        FILE* fp_mw;
        FILE* fp_mportf;

        FILE* fp_out;

        //loss matrix
        float *m_loss;
        //this matrix store the data for all portfolio
        float *m_portf;
        float *m_u;
        float *m_v;
        float *m_w;
        //store data for single portfolio
        float *portfolio;
        float *dv_temp;
        float *m_dvw;
        float *m_pv;
        float *mu_col_sum;
        float *m_stemp;
        float *fp_ljlp;  //algorithm 5
        float *fp_ljsp;  //algorithm 5
        float *fp_ljlm;  //algorithm 5
        float *fp_ljsm;  //algorithm 5
        float *m_l12norm; //algorithm 9
        float *m_larrow;  //algorithm 9
        float *m_ltemp;   //algorithm 9
        float *m_b_temp;  //algorithm 9
        float *m_b;
```

```
float *CUDA_ca;
float *CUDA_cb;
float *CUDA_cc;
float *CUDA_portf;
float *CUDA_dvw;
float *CUDA_mtemp;
float *CUDA_btemp;

float *CUDA_m;
float *CUDA_b;

float ljlp[M]; //algorithm 5
float ljsp[M]; //algorithm 5
float ljlm[M]; //algorithm 5
float ljsm[M]; //algorithm 5
float a1=0;    //algorithm 5
float b1=0;    //algorithm 5
float w_londa;
float w_arrow;
float ls;
float vr_upperbound=0;
float var;

int sp = (int)((float)DRAW*P);
//Define h, in this example h = 1000
int h = (int)(0.1 * (float)DRAW);
int lq = 0;
int mt_count=0;
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = M * M * sizeof(float);
int size_CUDA_cb = M * M * sizeof(float);
int size_CUDA_cc = M * M * sizeof(float);
int size_CUDA_portf = M * sizeof(float);
int size_CUDA_mtemp = TEMP_ROW1 * M * sizeof(float);
int size_CUDA_btemp = TEMP_ROW1 * sizeof(float);
int size_CUDA_m = DRAW * M * sizeof(float);
int size_CUDA_b = DRAW * sizeof(float);

LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
//clock_t end_time;
clock_t total_time;

m_loss = new float [DRAW*M];
m_portf = new float [PORTF_NUM * M];
m_u = new float [DRAW*M];
m_v = new float [M*M];
m_w = new float [M*M];
portfolio = new float [M];
dv_temp = new float [M * M];
```

```
m_dvw = new float [M];
m_pv = new float [DRAW];
mu_col_sum = new float [M];
m_stemp = new float [M];
m_l12norm = new float [2*DRAW];
m_larrow = new float [h*M];
m_ltemp = new float [TEMP_ROW1*M];
m_b_temp = new float [TEMP_ROW1];
m_b = new float [DRAW];

fp_ljlp = &ljlp[0];
fp_ljsp = &ljsp[0];
fp_ljlm = &ljlm[0];
fp_ljsm = &ljsm[0];

fp_ml = fopen("d:\\loss16.txt","r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
fp_mu = fopen("d:\\u_extract16.txt", "r");
fp_mv = fopen("d:\\matrixv16.txt", "r");
fp_mw = fopen("d:\\matrixw16.txt", "r");

printf("Loading data L...\n");
loaddata(fp_ml , m_loss , DRAW, M);
printf("Loading data for portfolio matrix.\n");
loaddata(fp_mportf , m_portf , PORTF_NUM , M);
printf("Loading data U...\n");
loaddata(fp_mu , m_u , DRAW , M);
printf("Loading data V...\n");
loaddata(fp_mv , m_v , M , M);
printf("Loading data W...\n");
loaddata(fp_mw , m_w , M , M);

cudaMalloc((void**) &CUDA_mtemp , size_CUDA_mtemp);
cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp);
cudaMalloc((void**) &CUDA_m , size_CUDA_m);
cudaMalloc((void**) &CUDA_b , size_CUDA_b);

//==========Initialization ============================
ini_fmatrix( fp_ljlp , 1 , M);
ini_fmatrix( fp_ljsp , 1 , M);
ini_fmatrix( fp_ljlm , 1 , M);
ini_fmatrix( fp_ljsm , 1 , M);

for(int i=0; i<M ; i++)
{
        matrix_col_cpy( m_u, m_pv, DRAW, M , i);
        //quicksort in ascending order
        quicksort(m_pv, 0, DRAW-1);
        //compute the sum of each column of U
        mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);

        //sum of the p largest components of column j of L
        ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
        //sum of the N-p smallest components of column j of L
        ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
```

```
        //sum of the p smallest components of column j of L
        ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
        //sum of the N-p largest components of column j of L
        ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);
}


dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
dim3 numBlocks( M/TILE_WIDTHx , M/TILE_WIDTHy );

cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);
cudaMalloc((void**) &CUDA_cc , size_CUDA_cc);
//comput D*V
cudaMemcpy( CUDA_ca , m_w , size_CUDA_ca , cudaMemcpyHostToDevice);
cudaMemcpy( CUDA_cb , m_v , size_CUDA_cb , cudaMemcpyHostToDevice);
//Invoke MatrixMulKernelv3 for matrix-matrix multiplication D * V
MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb, CUDA_cc , M , M);

cudaMemcpy( dv_temp , CUDA_cc , size_CUDA_cc , cudaMemcpyDeviceToHost );
//Initialize for algorithm 9: Compute the L2 norm of each row of L, and let L_arrow be
//the matrix consisting of the h rows with the largest L2 norm
//, where h is a fixed constant greater than p
for(int i=0 ; i<DRAW ; i++)
{
        *(m_l12norm + 2*i) = (float)i;
        *(m_l12norm + (2*i+1)) = matrix_row_l2norm(m_loss , DRAW, M , i);
}

//quicksort in descending order
quicksort2(m_l12norm , 0, DRAW-1);

//copy h rows of m_loss into m_larrow based on the order of its ROW L2 norm
for(int i=0; i<h ; i++)
{
        matrix_cpy2(m_loss , m_l12norm, m_larrow , i , M);
}

//copy m_larrow data from CPU to GPU
cudaMemcpy( CUDA_mtemp , m_larrow , size_CUDA_mtemp , cudaMemcpyHostToDevice );

//Load data in Loss matrix from CPU to GPU
cudaMemcpy( CUDA_m , m_loss , size_CUDA_m , cudaMemcpyHostToDevice );

//=================================================================

cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf);

printf("CUDA_Program13:");
printf("Running %d portfolios with COLLATERAL=%f...\n", PORTF_NUM, COLLATERAL);
printf("DRAW=%d\n", DRAW );
printf("M=%d\n", M);
printf("P=%f\n", P);
printf("h=%d\n", h);
```

```
QueryPerformanceFrequency(&frequency);

s_time = clock();
QueryPerformanceCounter(&t1);
for(int k=0 ; k<PORTF_NUM ; k++)
{

     a1=0; //reset a1
     b1=0; //reset b1
     lq = 0; //reset lq

     //take one row of m_portf
     portf_data_cpy( m_portf , portfolio , PORTF_NUM , M, k);
     //copy data portfolio from CPU to GPU
     cudaMemcpy( CUDA_portf , portfolio , size_CUDA_portf , cudaMemcpyHostToDevice);

     //compute D* V *w
     matrix_multi( dv_temp , portfolio, m_dvw , M, M, M, 1, M, 1 );
     //==========step1 Algorithm 5============================
     //compute S:
     vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);

     ls = vector_sum2(m_stemp , 0 , M-1);
     //printf("S: %f \n", ls);

     //compute w_londa as defined by equation 3 in O(M) time
     for(int i=0; i<M ; i++)
     {
          //compute w_lomda
          if(*(m_dvw+i) >=0)
          {
               w_londa = *(m_dvw+i);
          }
          else
          {
               w_londa = 0.0;
          }

          w_arrow = w_londa - *(m_dvw+i);

          a1 += w_londa * ( ljlp[i] - ljsp[i] );

          b1 += w_arrow * ( ljlm[i] - ljsm[i] );

      }

 vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
 //printf("\nVR_upperbound: %f \n", vr_upperbound);

 //Step2: if vr_upperbound <= Vc then terminate this algorithm with answer "No"
 if(vr_upperbound <= COLLATERAL)
 {
         to_buy++;
         continue;  //go to the next portfolio
```

```
        }
        else
        {
                //printf("step3\n");
        }

        //==============Step3: Otherwise apply Algorithm 9=======================
        dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2);
        dim3 numBlocks2( M/TILE_WIDTHx2 , TEMP_ROW1/TILE_WIDTHy2);

        //comput L_arrow * w in O(hM) time
        for(int i=0 ; i< (h/TEMP_ROW1) ; i++)
        {
                    //compute L_arrow * portfolio
                   //Invoke CUDA kernel MatrixVectorMulKernel to do matrix-vector multiplication
                    MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2 >>>( CUDA_mtemp,
CUDA_portf , CUDA_btemp , TEMP_ROW1 , M);

                    cudaMemcpy( m_b_temp , CUDA_btemp , size_CUDA_btemp ,
cudaMemcpyDeviceToHost );

                    //compute the number of elements in m_b_temp that are greater than
                    //COLLATERAL
                    for(int j=0 ; j<TEMP_ROW1 ; j++)
                    {
                            if( *(m_b_temp+j) > COLLATERAL )
                            {
                                    lq++;
                            }
                            else
                            {
                            }
                     }
         }

          //If the number is greager than sp answer the problem with Yes.
          if(lq >= sp)
          {
              not_to_buy++;
              continue;
          }
          else
          {
          }

          //Step 5: If those two steps can't fine the answer -> Apply algorithm 1: Monte Carlo
          //simulation
          dim3 threadsperblock3( TILE_WIDTHx2 , TILE_WIDTHy2 );
          dim3 numBlock3(M/TILE_WIDTHx2 , DRAW/TILE_WIDTHy2 );

          //Invoke CUDA kernel MatrixVectorKernel2 for doing matrix-vector multiplication
          MatrixVectorMulKernel2<<< numBlock3 , threadsperblock3 >>>( CUDA_m , CUDA_portf ,
CUDA_b , DRAW, M );
          cudaMemcpy( m_b , CUDA_b , size_CUDA_b , cudaMemcpyDeviceToHost );
```

```
        //quicksorq
        quicksort( m_b , 0 , DRAW-1);

        var = m_b[DRAW - sp];
        //printf("This is VaR: %f\n", var);

        if(var > COLLATERAL)
        {
              not_to_buy++;
        }
        else
        {
              to_buy++;
        }
        mt_count++;

    }
    QueryPerformanceCounter(&t2);
    total_time = clock() - s_time;

    elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;

    printf("\n%f ms.\n", elapsedTime);
    printf("Total clock needed: %d\n\n", total_time );
    printf("VaR > Vc: %d\n", not_to_buy);
    printf("VaR <= Vc:%d\n", to_buy);

    //Free space
    delete [] m_loss;
    delete [] m_portf;
    delete [] m_u;
    delete [] m_v;
    delete [] m_w;
    delete [] portfolio;
    delete [] dv_temp;
    delete [] m_dvw;
    delete [] m_pv;
    delete [] mu_col_sum;
    delete [] m_stemp;
    delete [] m_l12norm;
    delete [] m_ltemp;
    delete [] m_b_temp;
    delete [] m_b;

    cudaFree(CUDA_ca);
    cudaFree(CUDA_cb);
    cudaFree(CUDA_cc);
    cudaFree(CUDA_dvw);
    cudaFree(CUDA_mtemp);
    cudaFree(CUDA_btemp);
    cudaFree(CUDA_portf);

    cudaFree(CUDA_m);
    cudaFree(CUDA_b);
```

```
        return 0;
}

//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2)
{
        int i;

        for(i=0 ; i<a_col ; i++)
        {
            *(b + i) = *(a + a_row2 * a_col + i);
        }

}
```

cudaprog14main.cu

```
//(CUDA) Algorithm 10, where the L1 of algorithm 5 is used in step 1 (with U,DVw replacing L,w)
//and the algorithm in algorithm 2 is used insteead in step 5
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "matrix_ope.h"

#ifndef COLLATERAL
#define COLLATERAL 950000.0
#endif

#ifndef TILE_WIDTHx
#define TILE_WIDTHx 16
#endif

#ifndef TILE_WIDTHx2
#define TILE_WIDTHx2 16
#endif

#ifndef TILE_WIDTHy
#define TILE_WIDTHy 16
#endif

#ifndef TILE_WIDTHy2
#define TILE_WIDTHy2 20
#endif

#ifndef TEMP_AREA
#define TEMP_AREA 1000
#endif

#ifndef PORTF_NUM
```

```
#define PORTF_NUM 10000
#endif

extern void ini_fmatrix(float *target , int t_row , int t_col);
extern void show_2d_matrix(float* m , int row, int col);
extern void loaddata(FILE* fp, float* target, int t_row, int t_col);
extern void quicksort(float* target, int left , int right);
extern void quicksort2(float* target, int left , int right);
extern void show_vect(float* v , int ele_n);
extern void matrix_col_cpy(float* target, float* vect, int row_t, int col_t, int col_i);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float vector_sum2(float* a, int start, int end);
extern void vector_ele_multi(float* a, float* b, float* ans, int ele_a);
extern void matrix_cpy2(float* a, float* b, float* c, int row, int col);
extern void matrix_cpy2v2(float* a, float* b, float* c, int a_row, int a_col, int start_row ,
int end_row, int start_col , int end_col);
extern void matrix_move(float *a, float *b , int m_row, int m_col  ,int row_start);
extern void vector_move(float *a, float *b , int ele_num , int start_e);
extern float matrix_row_l2normv2( float* m, int m_row , int m_col, int s_col , int e_col ,int
row_2);
extern void matrix_multi(float* m , float* n , float* ans, int m_row , int m_col , int n_row ,
int n_col ,int ans_row , int ans_col);

void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2);

//(CUDA) kernel for matirx-matrix multiplication
__global__ void MatrixMulKernelv3(float *md , float *nd, float *pd, int m_row , int m_col)
{
   __shared__ float Mds[TILE_WIDTHy][TILE_WIDTHx];
   __shared__ float Nds[TILE_WIDTHy][TILE_WIDTHx];

   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int Row = by * TILE_WIDTHy + ty;
   int Col = bx * TILE_WIDTHx + tx;

   float pvalue =0;

   for(int i=0 ; i< (m_col/TILE_WIDTHx) ; i++)
   {
           //loading data into shared memory
           Mds[ty][tx] = *(md+ Row * m_col + (i*TILE_WIDTHx+tx) );
           Nds[ty][tx] = *(nd+ (i*TILE_WIDTHy+ty)*m_col + Col );
           __syncthreads();

           //
           for(int j=0 ; j<TILE_WIDTHx ; j++)
           {
                   pvalue += Mds[ty][j] * Nds[j][tx];
                   __syncthreads();
           }
```

```
            //
            *(pd + Row * m_col + Col) = pvalue;
    }

}

//(CUDA) kernel for matrix-vector multiplication
__global__ void MatrixVectorMulKernel(float *md , float *nd, float *pd , int m_row, int m_col)
{
        __shared__ float Mds[TILE_WIDTHy2][TILE_WIDTHx2];
        __shared__ float Nds[TILE_WIDTHx2];

        //int bx = blockIdx.x;
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by * TILE_WIDTHy2 + ty;
        //int Col = bx * TILE_WIDTHx2 + tx;

        float pvalue=0;

        for(int i=0 ; i< (m_col/TILE_WIDTHx2)+1 ; i++)
        {
          Mds[ty][tx] = *(md + Row * m_col + (i* TILE_WIDTHx2 + tx));
          Nds[tx] = *(nd + (i* TILE_WIDTHx2 + tx));
          __syncthreads();

          for(int j=0 ; j < TILE_WIDTHx2 ; j++)
          {
                  pvalue += Mds[ty][j] * Nds[j];
          }

          *(pd + Row) = pvalue;
        }

}

int main()
{
        FILE* fp_ml;
        FILE* fp_mportf;
        FILE* fp_mu;
        FILE* fp_mv;
        FILE* fp_mw;

        FILE* fp_out;

        float *m_loss;
        float *m_portf;
        float *m_u;
        float *m_v;
        float *m_w;
        float *portfolio;
        float *dv_temp;
```

```
float *m_dvw;
float *m_pv;
float *mu_col_sum;
float *m_stemp;
float *fp_ljlp;  //algorithm 5
float *fp_ljsp;  //algorithm 5
float *fp_ljlm;  //algorithm 5
float *fp_ljsm;  //algorithm 5
float *m_ul2norm; //algorithm 9
float *m_uarrow;  //algorithm 9
float *m_utemp;   //algorithm 9
float *m_b_temp;  //algorithm 9
float *m_barrow;  //algorithm 9
float *m_b;

float *CUDA_ca;
float *CUDA_cb;
float *CUDA_cc;
float *CUDA_portf;
float *CUDA_dvw;
float *CUDA_uarrow_temp;
float *CUDA_btemp;

float *CUDA_m;
float *CUDA_b;

float ljlp[M]; //algorithm 5
float ljsp[M]; //algorithm 5
float ljlm[M]; //algorithm 5
float ljsm[M]; //algorithm 5
float a1=0;    //algorithm 5
float b1=0;    //algorithm 5
float w_londa;
float w_arrow;
float ls;
float vr_upperbound;
float var;


int sp = (int)((float)DRAW*P);
int h = (int)(0.1 * (float)DRAW);
int lq = 0;
int mt_count=0;
int to_buy=0;
int not_to_buy=0;

int size_CUDA_ca = M * M * sizeof(float);
int size_CUDA_cb = M * M * sizeof(float);
int size_CUDA_cc = M * M * sizeof(float);
int size_CUDA_portf = M * sizeof(float);
int size_CUDA_uarrow_temp = TEMP_AREA * M * sizeof(float);
int size_CUDA_btemp = TEMP_AREA * sizeof(float);

int size_CUDA_m = DRAW * M * sizeof(float);
int size_CUDA_b = DRAW * sizeof(float);
```

```
LARGE_INTEGER frequency;
LARGE_INTEGER t1, t2;
double elapsedTime;

clock_t s_time;
clock_t total_time;

m_loss = new float [DRAW*M];
m_portf = new float [PORTF_NUM * M];
m_u = new float [DRAW*M];
m_v = new float [M*M];
m_w = new float [M*M];
portfolio = new float [M];
dv_temp = new float [M * M];
m_dvw = new float [M];
m_pv = new float [DRAW];
mu_col_sum = new float [M];
m_stemp = new float [M];
m_ul2norm = new float [2*DRAW];
m_uarrow = new float [h * M];
m_utemp = new float [TEMP_AREA *M];
m_b_temp = new float [TEMP_AREA];
m_barrow = new float [h];
m_b = new float [DRAW];

fp_ljlp = &ljlp[0];
fp_ljsp = &ljsp[0];
fp_ljlm = &ljlm[0];
fp_ljsm = &ljsm[0];

fp_ml = fopen("d:\\loss16.txt","r");
fp_mportf = fopen("d:\\portfoliom16.txt" , "r");
fp_mu = fopen("d:\\u_extract16.txt", "r");
fp_mv = fopen("d:\\matrixv16.txt", "r");
fp_mw = fopen("d:\\matrixw16.txt", "r");

printf("Loading data L...\n");
loaddata(fp_ml , m_loss , DRAW, M);
printf("Loading portfolio matrix..\n");
loaddata( fp_mportf, m_portf , PORTF_NUM , M);
printf("Loading data U...\n");
loaddata(fp_mu , m_u , DRAW , M);
printf("Loading data V...\n");
loaddata(fp_mv , m_v , M , M);
printf("Loading data W...\n\n");
loaddata(fp_mw , m_w , M , M);

cudaMalloc((void**) &CUDA_m , size_CUDA_m);
cudaMalloc((void**) &CUDA_b , size_CUDA_b);

cudaMalloc((void**) &CUDA_uarrow_temp , size_CUDA_uarrow_temp );
cudaMalloc((void**) &CUDA_btemp , size_CUDA_btemp );

//====================Initialization for algorithm 5==============================
```

```
        ini_fmatrix( fp_ljlp , 1 , M);
        ini_fmatrix( fp_ljsp , 1 , M);
        ini_fmatrix( fp_ljlm , 1 , M);
        ini_fmatrix( fp_ljsm , 1 , M);

        for(int i=0; i<M ; i++)
        {
                //copy each column of U into m_pv, we only need the first M column of U
                matrix_col_cpy( m_u, m_pv, DRAW, M , i);
                quicksort(m_pv, 0, DRAW-1);  //quicksort
                //compute the sum of each column of U
                mu_col_sum[i] = vector_sum2(m_pv , 0,DRAW-1);

                //sum of the p largest components of column j of L
                ljlp[i] = vector_sum2(m_pv , DRAW-(int)sp, DRAW-1);
                //sum of the N-p smallest components of column j of L
                ljsp[i] = vector_sum2( m_pv , 0 , (DRAW - (int)sp)-1 );
                //sum of the p smallest components of column j of L
                ljlm[i] = vector_sum2( m_pv , 0 , ((int)sp)-1);
                //sum of the N-p largest components of column j of L
                ljsm[i] = vector_sum2( m_pv , (int)sp , DRAW-1);

        }

        dim3 threadsperblock( TILE_WIDTHx , TILE_WIDTHy );
        dim3 numBlocks( M/TILE_WIDTHx , M/TILE_WIDTHy );

        cudaMalloc((void**) &CUDA_ca , size_CUDA_ca);
        cudaMalloc((void**) &CUDA_cb , size_CUDA_cb);
        cudaMalloc((void**) &CUDA_cc , size_CUDA_cc);
        //compute D * V
        cudaMemcpy( CUDA_ca , m_w , size_CUDA_ca , cudaMemcpyHostToDevice);
        cudaMemcpy( CUDA_cb , m_v , size_CUDA_cb , cudaMemcpyHostToDevice);
        //Invoke CUDA core MatrixMulKernelv3 for matrix-matrix multiplication
        MatrixMulKernelv3<<< numBlocks , threadsperblock >>>(CUDA_ca, CUDA_cb , CUDA_cc , M ,
M);

        cudaMemcpy( dv_temp , CUDA_cc , size_CUDA_cc , cudaMemcpyDeviceToHost );
        //================Initialization for algorithm 9==============================
        for(int i=0 ; i<DRAW ; i++)
        {
                //First column contains the order number of each row
                *(m_ul2norm + 2*i) = (float)i;
                //The second column containsthe L2 norm of each row
                //only compute L2 norm for the first M elements in a row
                *(m_ul2norm + (2*i+1)) = matrix_row_l2normv2(m_u , DRAW, M , 0 , M-1 , i);
        }

        //quicksort in descendign order
        quicksort2(m_ul2norm , 0, DRAW-1);

        //copy the first h rows of U based on its L2 norm
        matrix_cpy2v2( m_u , m_ul2norm , m_uarrow, DRAW, M, 0, h-1 , 0, M-1 );

        //move needed data from CPU to GPU
```

```
        cudaMemcpy( CUDA_uarrow_temp , m_uarrow , size_CUDA_uarrow_temp ,
cudaMemcpyHostToDevice );


        dim3 threadsperblock2( TILE_WIDTHx2 , TILE_WIDTHy2 );
        dim3 numBlocks2( M/TILE_WIDTHx2 , TEMP_AREA/TILE_WIDTHy2 );

        dim3 threadsperblock3( TILE_WIDTHx2 , TILE_WIDTHy2 );
        dim3 numBlock3( M/TILE_WIDTHx2 , DRAW/TILE_WIDTHy2 );
        //Load loss matrix data from CPU to GPU
        cudaMemcpy( CUDA_m , m_loss , size_CUDA_m , cudaMemcpyHostToDevice );
        //=============================================================================

        cudaMalloc((void**) &CUDA_portf , size_CUDA_portf);
        cudaMalloc((void**) &CUDA_dvw , size_CUDA_portf);

        printf("CUDA_Program14:");
        printf("Running %d portfolios with COLLATERAL=%f...\n", PORTF_NUM, COLLATERAL);
        printf("DRAW=%d\n", DRAW );
        printf("M=%d\n", M);
        printf("P=%f\n", P);
        printf("h=%d\n", h);

        QueryPerformanceFrequency(&frequency);

        s_time = clock();
        QueryPerformanceCounter(&t1);
        for(int k=0 ; k<PORTF_NUM ; k++)
        {

            a1=0; //reset a1
            b1=0; //reset b1
            lq=0;  //reset lq

            portf_data_cpy( m_portf , portfolio , PORTF_NUM , M , k);
            //compute D*V*w
            matrix_multi( dv_temp , portfolio , m_dvw , M, M, M, 1, M, 1 );

            //==================step1 Algorithm 5 (see cudaprg4main)=====================
            //compute S:
            vector_ele_multi(mu_col_sum , m_dvw , m_stemp , M);

            ls = vector_sum2(m_stemp , 0 , M-1);
            //printf("\nS: %f \n", ls);

            //compute w_londa
            for(int i=0; i<M ; i++)
            {
                 //compute w_lomda
                 if(*(m_dvw+i) >=0)
                 {
                      w_londa = *(m_dvw+i);
                 }
                 else
                 {
```

```
                    w_londa = 0.0;
                }

                //compute w_arrow
                w_arrow = w_londa - *(m_dvw+i);

                a1 += w_londa * ( ljlp[i] - ljsp[i] );

                b1 += w_arrow * ( ljlm[i] - ljsm[i] );
            }

            vr_upperbound = ( (1.0/(2.0*sp))*( ls + a1 - b1)  );
            //printf("\nVR_upperbound: %f \n", vr_upperbound);

            //Step2: if vr_upperbound <= Vc then terminate this algorithm with answer "No"
            if(vr_upperbound <= COLLATERAL)
            {
                    to_buy++;
                    continue;
            }
            else
            {
            }
            //=================Step3: Otherwise apply Algorithm 9 with U, D*V*w=============
            cudaMemcpy( CUDA_dvw , m_dvw , size_CUDA_portf , cudaMemcpyHostToDevice );
            //compute U*D*V*w with large L2 norms
            for(int i=0 ; i< (h/TEMP_AREA) ; i++)
            {
                     //Invoke CUDA core MatrixVectorMulKernel for matrix-vector multiplication
                     MatrixVectorMulKernel<<< numBlocks2 , threadsperblock2
>>>( CUDA_uarrow_temp , CUDA_dvw, CUDA_btemp , TEMP_AREA , M);

                     cudaMemcpy( m_b_temp , CUDA_btemp , size_CUDA_btemp ,
cudaMemcpyDeviceToHost );

                     for(int j=0 ; j<TEMP_AREA ; j++)
                     {
                             if( *(m_b_temp+j) > COLLATERAL )
                             {
                                     lq++;
                             }
                     }
            }

            //If the counter is greater than sp then the answer for decision problem is Yes.
            if( lq >= sp )
            {
                //printf("The answer to the decision problem is YES!\n\n");
                not_to_buy++;
                continue;
            }
            else
            {
                       //IF lq > p then answer YES to the decision problem, else Monte Carlo
                       //simulation.
```

```
            }


          //==========Step 5: If those two steps can't fine the answer -> Apply algorithm 1====
            mt_count++;
            cudaMemcpy( CUDA_portf , portfolio , size_CUDA_portf , cudaMemcpyHostToDevice);
            //Invoke CUDA core MatrixVectorMulKernel for matrix-vector multiplication
            MatrixVectorMulKernel<<< numBlock3 , threadsperblock3 >>>( CUDA_m , CUDA_portf ,
CUDA_b , DRAW , M);

            cudaMemcpy( m_b , CUDA_b, size_CUDA_b , cudaMemcpyDeviceToHost );
            //quicksort in ascending order
            quicksort( m_b , 0 , DRAW-1);

            var = m_b[DRAW - sp];
            //printf("This is VaR: %f\n\n\n", var);

              if(var > COLLATERAL)
              {
                        not_to_buy++;
              }
              else
              {
                        to_buy++;
              }

         }
         QueryPerformanceCounter(&t2);
         total_time = clock() - s_time;

         elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;

         printf("\n%f ms.\n", elapsedTime);
         printf("Total clocks:%d \n\n", total_time);

         printf("VaR > Vc: %d\n", not_to_buy);
         printf("VaR <= Vc:%d\n", to_buy);

         //Free space
         delete [] m_loss;
         delete [] m_portf;
         delete [] m_u;
         delete [] m_v;
         delete [] m_w;
         delete [] portfolio;
         delete [] dv_temp;
         delete [] m_dvw;
         delete [] m_pv;
         delete [] mu_col_sum;
         delete [] m_stemp;
         delete [] m_ul2norm;
         delete [] m_uarrow;
         delete [] m_utemp;
         delete [] m_b_temp;
         delete [] m_barrow;
```

```
        delete [] m_b;

        cudaFree(CUDA_ca);
        cudaFree(CUDA_cb);
        cudaFree(CUDA_cc);
        cudaFree(CUDA_portf);
        cudaFree(CUDA_dvw);
        cudaFree(CUDA_uarrow_temp);
        cudaFree(CUDA_btemp);

        cudaFree(CUDA_m);
        cudaFree(CUDA_b);

        return 0;
}


//this function move a row of data from matrix a to vector b
void portf_data_cpy(float *a , float *b , int a_row, int a_col , int a_row2)
{
        int i;

        for(i=0 ; i<a_col ; i++)
        {
            *(b + i) = *(a + a_row2 * a_col + i);
        }

}
```