Acceleration of Market Value-at-Risk Estimation

Matthew Dixon
Department of Computer Science
3060 Kemper Hall
UC Davis, CA 95616
mfdixon@ucdavis.edu

Jike Chong
Department of Electrical Engineering and Computer Science / Center for Innovative Financial Technologies
573 Soda Hall
UC Berkeley, CA 94720-1770
jike@berkeley.edu

Kurt Keutzer
Department of Electrical Engineering and Computer Science / Center for Innovative Financial Technologies
576 Soda Hall
UC Berkeley, CA 94720-1770
keutzer@eecs.berkeley.edu

ABSTRACT

The proliferation of algorithmic trading, derivative usage and highly leveraged hedge funds necessitates the acceleration of market Value-at-Risk (VaR) estimation to measure the severity of portfolios losses. This paper demonstrates how solely relying on advances in computer hardware to accelerate market VaR estimation overlooks significant opportunities for acceleration.

We use a simulation based delta-gamma Value-at-Risk (VaR) estimate and compute the loss function using basic linear algebra subroutines (BLAS). Our NVIDIA GeForce GTX280 graphics processing unit (GPU) based baseline implementation is a straight-forward port from the CPU implementation and only had a 8.21x speed advantage over a quadcore Intel Core2 Q9300 central processing unit (CPU) based implementation.

We demonstrate three approaches to gain additional speedup over the baseline GPU implemention. Firstly, we reformulate the loss function to reduce the amount of necessary computation and achieved a 60.3x speedup. Secondly, we selected functionally equivalent distribution conversion modules to give the best convergence rate - providing an additional 2x speedup. Thirdly, we merged data-parallel computational kernels to remove redundant load store operations leading to an additional 1.85x speedup. Overall, we have achieved a speedup of 148x against the baseline GPU implementation, reducing the time of a VaR estimation with a standard error of 0.1% from minutes to less than one second.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: [Parallel programming]; G.3 [**Probability and Statistics**]: [Probabilistic algorithms (including Monte Carlo)]; J.4 [**Social and Behavioral Sciences**]: [Economics]

WHPCF'09, November 15, 2009 Portland, Oregon, USA.

1. INTRODUCTION

Accelerated market VaR estimation gives financial institutions a competitive edge in algorithmic trading. Recent advances in computation technologies have enabled the extensive use of high frequency algorithmic trading. Financial institutions use market VaR to measure the risk of severe losses on their portfolios due to market events [11]. In stressed market conditions, VaR constrained trading strategies could result in lost trading opportunities or under-capitalized positions if the VaR estimation is out-of-step with intra-day market movements.

New regulations on VaR methodology imply more computation. The latest financial market dislocations exposed the deficiencies of Basel II market risk management regulations. This has prompted revisions to the regulations which include the requirement for financial institutions to measure VaR under stressed market scenarios and adjust the Tier I capital in leveraged funds. Stressed VaR estimation requires more pervasive systematic stress testing of financial pricing models and subsequent increased computation - especially for derivative products.

Fast VaR estimation improves the responsiveness of risk management systems. There are currently two broad usage models for VaR estimation in large financial institutions. Daily VaR reports summarize the vulnerabilities to market movements in the positions financial business unit take. They are the central tenet of institutions' market risk management operations. These reports are drawn from a succession of stress testing and sensitivity analysis which measure the sensitivity of the VaR to adverse market conditions and business unit trading strategies.

Another usage model involves VaR estimations on an ad-hoc basis by business analysts. Usage cases may include the assessment of the impact of new traded products on their business unit profit and loss account, or re-estimating the VaR with adjusted positions in the assets. In each usage model, fast VaR estimation reduces the delay from having to re-estimate the VaR in the event of a risk management system failure.

Efficient integration of pricing models into VaR engines leads to more robust financial management systems. Financial model developers often face challenges in integrating new pricing models into the VaR engines of risk management systems. Typically financial modelers are domain experts with limited access to the high performance computing development and testing environment for the live risk management system. Prototype model migration from a desktop

development environment to the deployment environment is often hampered by disjoint domain expertise. The lag between specification of new models and their deployment could result in unaccountable exposure to risks. So the need to speedup VaR estimation and improve its deployment is paramount to responsive and robust financial risk management.

There are several approaches to estimating the VaR of portfolios surveyed by the literature [8][10][11]. The *asset based* approaches consider the loss of each individual asset over a time horizon and *aggregate* them to determine the portfolio loss. Monte-Carlo methods are typically used to then estimate the VaR because the aggregation is a non-linear mapping of the assumed joint distribution of the underlying asset returns. *Portfolio* based approaches, on the other hand, only consider the aggregated portfolio loss and use Monte-Carlo methods to estimate the portfolio loss distribution.

Although the asset based approach gives far more credible estimates than the portfolio based approach, it typically relies on Monte Carlo simulation of each asset. The approach becomes computationally prohibitive when the portfolio is large and many of the assets are themselves contingent claims on the other underlying assets, such as options and futures. This motivated the emergence of various polynomial approximations of the portfolio loss function which provide a compromise between cost and accuracy.

1.1 Overview

We tested a standard implementation of quadratic VaR estimation, described in the next Section, on a large portfolio and achieved a 8.21x speedup on the GPU compared to an algorithmically equivalent (but unoptimized) multi-core CPU implementation. We then identified some of the key bottlenecks and found significant opportunities for optimization by reformulating the problem as described in Section 4.1. This led to a 99.5x speedup in the loss estimation on the GPU. Selecting the modules described in Section 5 which gave the fastest conversion rate led to a further 2x speedup. Finally merging some of the kernels, which we dub 'implementation styling' and are described in Section 6, led to an additional 1.85x speedup. Section 7 shows that this combination of improvements enabled a total speedup of 148x against the baseline GPU implementation.

The paper demonstrates that solely relying on the platform for performance optimizations overlooks significant optimization opportunities elsewhere. By simultaneously considering multiple levels of optimization, we demonstrate a more complete approach for optimizing quadratic VaR estimation. This approach provides clear interfaces for collaboration and facilitates potentially faster and smoother integration across domains of expertise.

2. QUADRATIC VALUE-AT-RISK

Quadratic VaR approximations, often referred to as *delta-gamma* VaR approximations compute the first (*delta*) and second (*gamma*) terms in the Taylor expansion of the loss function. This type of approach avoids the much more computationally demanding task of contingent claim pricing by only simulating the risk factor returns. For this reason, these approximations are better suited to large portfolios whose values are a smooth function of the underlying asset returns, such as portfolios of non-exotic instruments such as stocks, bonds and vanilla derivatives.

The loss function is linear in the risk factors for certain asset classes held in the portfolio, such as stocks, because their prices are directly represented by equity index risk factors. In this case, only a linear (or *delta*) approximation is required. Bond prices on the other hand, depend non-linearly on quoted interest rates and so the loss function is non-linear in interest rate risk factors. Equally, other asset classes require higher order loss function approximations such as commodity and fixed income derivatives, e.g. swaptions and spread options. Quadratic VaR estimation is not appropriate when such asset classes are included in the portfolio.

Large portfolios may contain hundreds of thousands of tradable assets that are exposed to movements in various currency, commodity, rates, credit and equity markets. Risk analysts capture these market dynamics by selecting a smaller number of market risk factors which are linear combinations of one or more quoted tickers.

Consider a portfolio, whose value $P(R_1, R_2, \ldots, R_N)$ is a non-linear function in N correlated market risk factors $R_i(t)$ at time t. Under a *delta-gamma* approximation, the portfolio incurs a change of value on the profit and loss (P&L) account due to market movements of the form

$$dP = \sum_{i,j} \underbrace{\Delta_i dR_i}_{delta} + \underbrace{\frac{1}{2} dR_i \Gamma_{ij} dR_j}_{gamma}, \quad (1)$$

where the first and second derivatives in the portfolio value with respect to the risk factors are denoted $\Delta_i = \frac{\partial P}{\partial R_i}$ and $\Gamma_{ij} = \frac{\partial^2 P}{\partial R_i \partial R_j}$ - i is the index for each risk factor whose change in value dR_i over a chosen time period is a log-normal random variable. For linear portfolios, comprised of linear instruments such as stocks, the second (gamma) term may be neglected. Exclusion of the gamma term when the portfolio loss $-dP$ is non-linear in the risk factors, such as when bonds or contingent claims are held, leads to errors which grow with the risk horizon.

For the simplest case when all asset positions are long[1] VaR is estimated at the c confidence limit as the $1-c$ most negative value $\text{VaR}[-dP] = F^{-1}(1-c)$, where $F(dR_1..dR_N)$ is the cumulative loss distribution over the risk factor returns. c is typically between 95% and 99% corresponding to the respective 1 in 20 up to the 1 in 100 chance of encountering a severe loss.

By far the simplest model assumption is that portfolio losses are described by a joint normal distribution. But normal loss distributions do not exhibit the "fatter tails" of the loss distribution implied from historical data, where extreme events occur more frequently than the normal distribution describes [7]. Consequently normal loss distributions tend to severely underestimate potential losses. Risk analysts prefer to relax this mathematically convenient assumption in favor of more realistic but analytically intractable representations of the inverse cumulative distribution function. The normal distribution belongs to the class of elliptic distribution functions which contain more empirically consistent functions such as the student-t distribution [8].

[1]When both long and short trade positions are held in the portfolio, VaR must be estimated from both tail regions of the loss distribution.

2.1 Elliptic distributions

Under the quadratic VaR approximation, the inverse cumulative distribution of the portfolio losses is not analytically tractable and so the VaR must be estimated or approximated. Elliptic distribution functions, however, exhibit the affine transformation property (see Appendix A) through which the moments of the loss distribution can be conveniently expressed in terms of the moments of the univariate distributions for each of the risk factor returns. This property implies that the loss distribution[2] is represented solely by the mean μ and covariance matrix Σ. Because the moments of the loss distribution are given analytically, the standard error can be used as a convenient measure of accuracy of any unbiased estimator.

Monte Carlo methods are typically used to estimate the VaR when the inverse cumulative distribution function of the portfolio losses can not be determined analytically. These methods can be prohibitively computationally expensive [14] and performance optimizations are critical.

3. SOLUTION APPROACH

We demonstrate the following three level solution approach to speeding up MC-VaR estimation. This approach is able to draw together a number of optimizations:

1. *Problem reformulation*: Transform the mathematical formulation to minimize the number of floating point operations;
2. *Module selection*: Select the normal and student-t quasi-random number sequences which are mathematically proven to yield faster convergence; and
3. *Implementation styling*: Study the computation in the algorithm to best leverage the capabilities of the underlying platform and scope the data set to reduce memory sub-system access contention.

3.1 Previous related work

There is a wide range of research effort to solve high dimensional computational finance problems on parallel architectures. This started with initial efforts on low-cost Beowulf clusters [5][15] and more recently includes novel parallel architectures [1][14][16]. The main focus in these prior works has been on the software architecture and platform dependent optimizations and, to a lesser extent, the application-specific considerations such as module selection and problem reformulation.

By integrating the problem formulation into our solution approach, we are able to use more realistic modeling assumptions such as fatter tails and non-linearity in the portfolio and hence extend the class of models currently considered by the high performance computing community. By integrating module selection into our solution approach, we can select the minimum necessary computation to achieve a target margin of error. Using more realistic modeling assumptions in our problem formulation also introduces new computational challenges, some of which we expose in this paper.

There is also some effort in evaluating performance of computational finance related kernels on GPUs. Thomas et al. [16] compared the performance of random number generators on multicore CPUs and GPUs by measuring the number of samples generated per second and the quality of the random sequences. We too demonstrate through module selection how the quality of the random sequences has a significant effect on the loss estimation error convergence rate but consider more empirically consistent statistical distributions together with other model approximations and parameterizations.

Glasserman et al. [7] presents a Monte Carlo method for estimating VaR using a delta-gamma approximation of a student-t distributed loss function. Their approach uses the characteristic function of the distribution to efficiently approximate the VaR and subsequently derive a novel hybrid Monte Carlo method. This approach is demonstrated on portfolios of up to 100 instruments. Our interest is in the estimation of portfolios with at least ten times as many instruments, where the portfolios are typically subjected to several thousand risk factors. We make similar modeling assumptions but instead choose to use readily available off-the-shelf random number generators designed for high dimensional problems.

In the next Section we show how the loss function evaluation is typically implemented and show how this function can be reformulated to reduce it's computational cost. We will then discuss further optimizations in the proceeding Sections followed by their effect on performance benchmarks.

4. PROBLEM FORMULATION

It is standard convention to discretize the stochastic evolution of an asset with an Euler scheme which draws i.i.d. random vectors $\epsilon_{\mathbf{k}}$ for each time step Δt and realization k

$$\frac{\Delta S_{ik}(t)}{S_{ik}(t)} = \mu_i \Delta t + \sigma_i \sqrt{\Delta t}\epsilon_{ik},\ \epsilon_{ik} \sim N(0,1), \qquad (2)$$

whose parameters μ_i and σ_i respectively represent the drift and standard deviation of the asset returns. The k^{th} *scenario* is a vector $Y_k,\ k \in 1, 2, \ldots, M$ of correlated risk factor returns $Y_{ik} := \frac{\Delta R_{ik}(t)}{R_{ik}(t)}$.

Through the affine transformation property $Y = \mu + \hat{Q}X$ of the assumed joint distribution of the risk factor returns, each scenario is expressed in terms of the uncorrelated risk return vector X_k with elements X_{ik}, where $\hat{Q}$ is a lower triangular Cholesky factor of the s.p.d. covariance matrix $\Sigma = \hat{Q}\hat{Q}'$. Following [8], we diagonalize the symmetric matrix $\hat{Q}^T \Gamma \hat{Q} = U \Lambda U^T$, where U is orthogonal and Λ is diagonal with elements λ_i so that equation 1 becomes

$$\Delta P_k = \sum_{i,j} R_i \Delta_i Q_{ij} X_{jk} + \hat{\lambda}_i X_{ik}^2, \qquad (3)$$

where $Q = \hat{Q}U$ and $\hat{\lambda}_i = 2R_i^2\lambda_i$.

The first two moments of the loss distribution are typically estimated in order to assess the asymptotic convergence rate to the analytic moments and thereby terminate the simulation. These estimated moments are respectively given by $\hat{\mu}_p = \frac{1}{M}\sum_k dP_k$ and $\hat{\sigma}_p^2 = \frac{1}{M-1}\sum_k (dP_k - \mu_p)^2$ (recall that the analytic moments are derived in the Appendix). For sufficiently large M, the loss distribution converges to the assumed distribution and the VaR is then estimated from the (1-c)% most severe loss.

[2]For a student-t loss distribution, the covariance matrix is given by $\frac{v}{v-2}\Sigma$, where v is a parameter defining the number of degrees of freedom of the distribution.

Requirements.

The estimated mean and standard deviation of the simulated portfolio losses must fall within an acceptable tolerance before the VaR can be estimated.

Standard parallel implementation.

In a standard implementation, the generation of independent random numbers is vectorized. This ensures that the pipelines of the multicore processor are kept busy. A standard parallel implementation (see for example [14]) will then vectorize the generation of correlated random vectors by multiplying a random matrix with the Cholesky factor of the covariance matrix before then evaluating the losses. The moments of these losses are estimated to establish convergence criteria and, if met, requires the array of $M >> N$ portfolio losses to be sorted in ascending order before reporting the (1-c)% array index. Because the matrix-matrix multiply is the bottleneck, the loss function is often overlooked for sources of optimization.

4.1 Optimization

The key optimization reformulates the portfolio loss function to use a precomputed, deterministic part of the delta term. This precomputation enables the bottleneck matrix-matrix computation to be replaced with a matrix-vector operation, reducing computation by a factor of $O(N)$, where N is the number of risk factors.

For large portfolios, this optimization gives tractable VaR estimations provided that the eigenvalues of $\hat{Q}^T \Gamma \hat{Q}$ are estimated on a less frequent basis than the VaR. This optimization is specific to the use of the transformed form of the delta-gamma approximation described in [8]. A parallel implementation of this form of the approximation does not appear to have been addressed by the literature and the observation that the delta term can be reformulated to reduce computation is novel. There is no merit to this reformulation if the gamma term is not expressed in diagonalized form since the matrix-matrix multiplication $\hat{Q}X$ would still be required to evaluate the gamma term.

Recall that in the standard form, a random matrix of $N \times M$ correlated random iterates is drawn in $2N \times N \times M$ FLOPs and used to evaluate a vector of portfolio losses. The delta term in the loss function given by equation 3 requires a further $2N \times M$ FLOPs.

In the optimized form, the total number of FLOPs is significantly reduced with pre-computation of the loss function

$$q_j := \sum_i R_i \Delta_i (c_i + Q_{ij}). \tag{4}$$

The portfolio loss can then be simulated using the optimized form

$$\Delta P_k(R_1, \ldots, R_N, t) = \sum_{i,j} q_j X_{jk} + \hat{\lambda}_i X_{ik}^2. \tag{5}$$

The deterministic component of delta, q, computes in $N \times (2N + 1)$ FLOPs and the delta component of the loss function computes in just $2N \times M$ FLOPs (excluding terms independent of M). This reformulation achieves a factor of $N+1$ reduction in computation resulting from replacing the BLAS single precision general matrix-matrix multiplication kernel `Sgemm` with the matrix-vector multiplication kernel `Sgemv`.

	Standard	Reformulated	Speedup
Δ	$2N \times (N+1) \times M$	$2N \times M$	$N+1$
$\Delta - \Gamma$	$N \times (2N+5) \times M$	$5N \times M$	$\frac{(2N+5)}{5}$

Table 1: A comparison of the upper bound on the number of FLOPs (excluding terms independent of M) required to compute the standard and reformulated delta and delta-gamma loss approximations.

The quadratic gamma term is evaluated by first scaling each row of X_{ik} by $\hat{\lambda}_i$ using N_Γ `Saxpy` evaluations totaling $N_\Gamma \times M$ FLOPs, where $N_\Gamma < N$ is the dimension of Γ. This step is followed by M BLAS single precision vector-vector products `Sgemv` evaluations, totaling $2N_\Gamma \times M$ FLOPs. So when N_Γ approaches N, the evaluation of the gamma term is at most $3N \times M$ FLOPs. Table 1 summarizes the upper bound on the FLOP count for delta and delta-gamma loss function approximations with and without optimization.

5. MODULE SELECTION

The number of simulations M required for the estimated standard deviation to converge to the analytic depends on the quality of the random sequences.

5.1 Quasi-random Number Generation

Low discrepancy sequence generators, or quasi-random number generators as they are frequently referred to as, are widely used by financial practitioners for (quasi-)Monte Carlo methods on account of their often superior convergence rates which do not significantly deteriorate with the number of dimensions d. Moskowitz et al. [12] find empirically that for large dimensions, the discrepancy of the quasirandom sequence of length N is $1/\sqrt{N}$ for $N < \exp(d)$. This observation is not consistent with the theory, however, which defines the convergence rate $c(d)\frac{ln(N)^d}{N}$ from the measure of discrepancy and thus should cease to be asymptotically faster than the $1/\sqrt{N}$ rate of pseudorandom sequences in high dimensions.

The same authors [12] introduced the concept of *effective dimensionality* to explain how smoothness in the integrand effectively reduces the dimensionality of the problem, motivating examples in the pricing of mortgage backed securities [4]. This work was followed by a decade of research activity demonstrating the application of quasi-Monte Carlo methods to a wide class of high dimensional problems in finance and continues to be an active research area (see for example [6][17]).

The aforementioned theoretical convergence rate estimate is based on the discrepancy of the sequence alone and is not the only measure of uniformity. Sobol' introduced an additional uniformity property, referred to as *property A*, by dividing the unit hypercube $[0,1]^d$ into 2^d equally sized subcubes and partitioning any sequence of points belonging to the unit hypercube into 2^d consecutive blocks. If each one of the points in any block belongs to a different equally sized subcube, then the sequence exhibits property A. Whilst the claim can not be made that QRNGs tractably give low discrepancy sequences in high dimensions, Sobol' sequences have been shown to satisfy an alternative measure of uniformity in dimensionality sufficiently high for the

most demanding financial applications. Bratley and Fox [3] developed an efficient Sobol' QRNG generator which satisfies property A in up to 64 dimensions. Joe and Kuo [9] originally extended Bratley and Fox's Sobol' QNRG [3] to satisfy this property in upto 1111 and have since reported an upperbound of 16900.

We extend Mike Giles' implementation of the single-precision parallel Sobol' quasi-random number generator provided in the CUDA SDK version 2.2. This Sobol' QRNG code is efficiently implemented in p-way parallel by jumping from element x_n in a sequence to element x_{n+p} in $\mathcal{O}(\log\ p)$ operations. This CUDA implementation assumes that the entire quasi-random number sequence is generated in one contiguous memory block, which in high dimensions, may exceed the global memory available on the graphics card. Our modification of this implementation allows for a random sequence to be generated in N_b equally sized contiguous memory blocks, each of which are loaded sequentially into global memory.

Requirements.

Uniform quasi-random sequences should satisfy property A to a dimensionality at least equal to the number of risk factors and be amenable to efficient parallel implementation.

5.2 Transformation of uniform variates

Various methods exist for transforming quasi-random numbers to normal or student-t variates. The Moro method [13] of interpolation of the inverse cumulative normal distribution gives a sequence of independent normal random variates. This approach is frequently used by practitioners using normal MC-VaR.

Alternatively the Box-Muller method, which is based on a polar transformation, uses two of the coordinates of each random point in the unit N hypercube to generate pairs of independent standard normal variates. Bailey [2] extended the Box-Muller method to transform independent uniform random numbers to student-t distributed random variables.

The effect of the transformations on the uniformity properties of the random sequence does not appear to be addressed from a theoretical perspective in the literature. For this reason, we simply compare the relative standard error of the portfolio loss followed by the relative error in the VaR estimate using a single-precision implementation of Moro's interpolation method and the Box-Muller method applied to a Sobol' sequence.

The results are presented in Figure 1. For 4096 risk factors, approximately 1.5×10^6 or 7.5×10^5 scenarios is sufficient to estimate the standard error of the loss distribution to within 0.1% when using Moro's interpolation method or the Box-Muller method respectively. This tolerance corresponds to an error in the delta-VaR of 0.1%. The study of the comparative effect of using single versus double precision arithmetic on the convergence rate is an important extension of this work, but is beyond the scope of this paper.

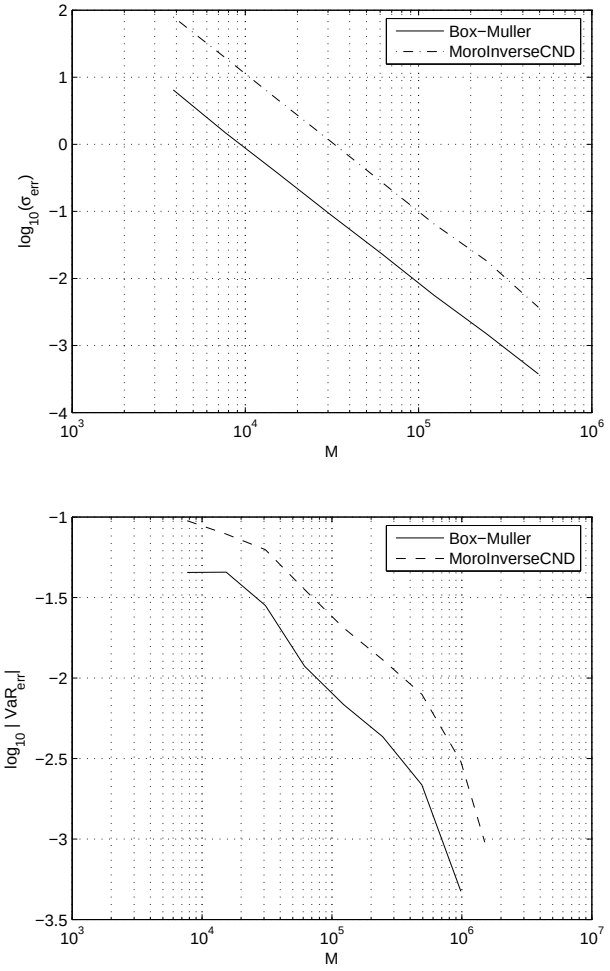

Figure 1: (top) A comparison of the standard error (%) in the portfolio loss distribution using Moro's interpolation method and the Box-Muller method applied to Sobol' sequences. (Bottom) The corresponding error (%) in the simulated 1 day portfolio delta VaR (c=95%) monotonically converges to the analytic delta VaR (9.87%) with the number of scenarios. In single-precision arithmetic, approximately 1.5×10^6 or 7.5×10^5 scenarios is sufficient to estimate the delta-VaR to within 0.1% when using Moro's interpolation method or the Box-Muller method respectively.

6. IMPLEMENTATION STYLING

6.1 Standard Implementation

Our MC-VaR implementation consists of three performance critical steps: Uniform random sequence generation, parameter distribution conversion, and portfolio loss calculation (including risk factor cross-correlation). Each step is typically optimized independently with intermediate results passed between the steps. A simulation of just a few thousand scenarios with thousands of risk factors can produce multiple megabytes of intermediate results, creating a large amount of simulation states that must be maintained during the execution of the algorithm.

This type of implementation is favored by developers because of its functional modularity, where input and output of isolated steps can be designed, maintained, and tested separately. However [14] shows that this pipeline maps poorly on to the available GPU hardware resources. The large amount

of simulation states does not fit in on-chip caches and must be stored to and retrieved from off-chip memory between steps. At the same time, each step usually involves a trivial amount of computation and the performance bottleneck is the memory throughput accessing off-chip memory. After appropriately blocking the computation so that constants common to a step are only loaded once per thread block, we estimate that the algorithm spends 90% in communication and 10% in computation.

6.2 Optimization

Maintenance of a large intermediate result working set is not required but is a consequence of implementation decisions for function modularity. To optimize the implementation for execution on GPUs, one must minimize the number of data movements, which can dominate the execution time of the implementations.

We experiment with merging the uniform random sequence generation and the distribution conversion steps, so that the conversion takes place as soon as the uniform random values are generated. By converting the uniform random values while they are still in the GPU's register files on-chip, we saved the execution time associated with a set of round trip memory transfers compared to writing out the results to device memory and reading them back. This optimization, however, can be highly sensitive to the various resource bottlenecks on the GPU. It is found to be effective for the Box-Muller method, but not as effective for Bailey's method, even though their functional form is very similar. The result is provided in Section 7.4.

7. RESULTS

We present the performance of the quadratic VaR estimation engine and demonstrate speedups in three areas: 1) problem reformulation, 2) module selection, and 3) implementation styling. The results are provided for implementations on both the CPU and GPU where applicable.

The benchmark results for the CPU implementation are measured on an Intel Core2 Q9300 Quadcore CPU, with 3MB L2 cache and 2.5GHz clock frequency. The estimation engine is compiled with Intel ICC version 11.1 (056). We use the multi-threaded BLAS kernels implemented in the Intel Math Kernel Library (MKL) version 10.2 (update 2).

The benchmark results for the GPU implementation are measured on an NVIDIA GeForce GTX 280 GPU with 1GB of global memory. The GTX280 has 30 multiprocessor cores running at 1.3GHz clock frequency, each with an 8-element wide single instruction multiple data (SIMD) vector unit. The CUDA programs are compiled with NVCC version 2.2 and use BLAS routines available in the cuBLAS library version 1.0.

We have chosen algorithmically equivalent baseline CPU and GPU implementations which provide negligible differences in intermediate results and VaR estimates. In other words, the baseline implementations only provide a transparent reference point from which to trace back any differences in output between the CPU and GPU after subsequent code modifications. They are not optimized for performance.

The baseline CPU implementation first generates Sobol' sequences using our own OpenMP parallel implementation of the Sobol' QRNG adapted from a publicly available c++ code written by Joe and Kuo [9]. This implementation conveniently provided precomputed direction vectors to dimensions beyond our requirements. Intel's MKL provides an optimized implementation of the Sobol' QRNG (with merged distribution transformations) based on Bratley and Fox's [3] algorithm 659. This implementation, however, is only preconfigured to generate quasi-random numbers in dimensions of up to 40, although it does allow for precomputed direction vectors in higher dimensions (e.g. from [9]) to be externally referenced.

Sobol' sequences are transformed into normal random variables using a c implementation of Moro's Inverse Cumulative Distribution Function (ICDF) provided, for comparative reasons, in the NVIDIA CUDA SDK 2.2. The loss function is then evaluated with calls to the MKL cBLAS kernels `cblasSgemm`, `cblasSgemv` and `cblasSaxpy`. We observed that the baseline CPU implementation spends approximately half the time generating the random matrices (QRNG + distribution conversion), and half the time evaluating the loss function. Finally, the moments of the loss distribution are estimated by sorting the loss vector of length M using `qsort` from `cstdlib` in $\mathcal{O}(MlogM)$ operations on average.

7.1 Overall results

We present the overall speedup from our three-stage optimizations of the CUDA implementation of the $\Delta - \Gamma$ VaR model in Table 2. The columns in the Table represent the three steps of execution in the VaR estimation and the Table content specifies the absolute and proportional timings of the steps.

The baseline GPU implementation first generates uniform quasi-random numbers with the "embarrassingly parallel" Sobol' QRNG provided in the NVIDIA SDK 2.2. The sequence is transformed using our own optimized version of Moro's ICDF which makes extensive use of computation blocking and shared constants in the on-chip shared memory. The loss function is then evaluated using `cublasSgemm`, `cublasSgemv` and `cublasSaxpy` kernels.

The baseline GPU implementation is able to exploit the absence of cross thread communication in the Sobol' QRNG and Moro's ICDF by leveraging the faster native transcendental functions to significant effect. There is a 22.5× speedup for the Sobol quasi-random number generation and a 988× speedup for Moro's ICDF. With a 4.99× speedup, the loss function evaluation step also benefits from being mapped to the GPU, although it becomes the performance bottleneck in the baseline GPU implementation.

We optimize the VaR computation on the GPU in three areas: in problem reformulation, module selection, and implementation styling. We briefly illustrate the effect of the three areas of optimization here and explain them in detail in the following subsections:

- *Problem reformulation* By reformulating the algorithm as described in Section 4.1, we are able to obtain a 99.5× speedup in the loss function evaluation - the bottleneck in the baseline implementation.

- *Module selection* By choosing the Box-Muller method over Moro's ICDF, we selected the module that gives the faster numerical convergence rate in the standard error. We were able to reach the same VaR estimation accuracy with nearly half the number of simulations. This provides another 2× speedup.

Timing (s)	QRNG	Distribution Conversion	Loss Evaluation
Baseline GPU	0.441 (0.32%)	0.470 (0.34%)	137 (99.3%)
Problem Formulation (GPU)	- (19.2%)	- (20.5%)	1.38 (60.3%)
Module Selection (GPU)	0.220 (19.2%)	0.235 (20.5%)	0.692 (60.3%)
Implementation Styling (GPU)	0.246 (26.2%)		- (73.8%)
Speedup	3.70x		199x
Total Speedup	148x		

Table 2: GPU speedup for the delta-gamma approximation with 4096 risk factors, simulated to achieve a standard error in the normal loss distribution within 0.1%.

- *Implementation styling* The structure of the random matrix generators can be further optimized to reduce redundant memory accesses by improving the style of implementation. Specifically, we merge the Sobol' and distribution conversion steps to remove a pair of redundant load and store operations from the computation of each distribution conversion. This reduces the QRNG generation and distribution conversion steps by 1.85×.

After these optimizations, we have enabled a 148× faster implementation compared to a GPU-based baseline solution, and 495× faster implementation compared to a CPU-based baseline solution. This illustrates that using a GPU-based implementation may provide some speedup over a CPU-based implementation, but relying on the platform advantage alone would overlook significant acceleration opportunities.

7.2 Problem reformulation

Table 3 compares the performance of CPU and GPU implementations of MC-VaR assuming a normal distribution for the joint risk factor returns. $N_b = 46$ blocks of size M_b were generated using a portfolio of $N = 4096$ risk factors. With a maximum block size of $M_b = 16384$ (for the available memory), this number of blocks ensures that approximately 7.5×10^5 (753664) scenarios are generated to achieve 0.1% accuracy in the standard error of the loss distribution. Table 3 shows the comparative effect of the optimized Monte Carlo algorithm on the time taken to evaluate delta-gamma VaR on the GPU and CPU. The numbers in parentheses are the loss estimating portion of execution times without the QRNG and distribution conversion steps.

Without problem reformulation, we see only a 8.21× speedup going from baseline CPU to baseline GPU implementation. With the problem reformulation, we see an additional 99.5× speedup for the loss estimation from baseline GPU implementation to reformulated GPU implementation. Reformulation enabled a 60.3× speedup for the delta-gamma VaR estimation problem. Overall, the reformulated algorithm performs 200× faster on the GPU than the reformulated approach on the CPU.

Timing (s)	Standard	Reformulated	Speedup
CPU	569 (344)	230 (4.188)	2.47× (82.0×)
GPU	69.3 (68.8)	1.15 (0.692)	60.3× (99.5×)
Speedup	8.21×(4.99×)	200×(6.05×)	**495**×(497×)

Table 3: The comparative timings(s) of the standard and reformulated Monte Carlo algorithm for evaluating delta-gamma VaR on the GPU and CPU using the Box-Muller method with 7.5×10^5 simulations. The parenthesized values represent the times and speedup factors in just the loss function evaluation step.

Timing (s)	Standard (Separate)	Optimized (Merged)	Speedup
Box-Muller (step1)	0.220	0.246	1.85x
(step2)	0.235		
Bailey (step1)	0.220	0.652	0.770x
(step2)	0.282		

Table 4: GPU Implementation cost in seconds for setting up 7.5×10^5 experiments each with 4096 risk factors.

7.3 Module selection

Our optimized CUDA implementation of the Box-Muller method transforms $4096 \times 1.5 \times 10^6$ uniform quasi-random numbers in $0.470s$ and accounts for 20.5% of the total time of the reformulated algorithm. As explained in Section 5.2, the primary criteria for choosing the Box-Muller method is that the standard error converges twice as fast in single precision as when using Moro's ICDF applied to the same sequence of uniform quasi-random numbers.

7.4 Implementation styling

We experimented with merging of the random number generation step and the parameter distribution conversion step. Table 4 illustrates the impact of implementation styling on the Box-Muller and Bailey's methods. The performance highlights the sensitivity of the kernel merging optimization. The Box-Muller method was sped up by 1.85× through merging the steps, whereas the Bailey's method sustained a slow down. The implementation of the Box-Muller method and the Bailey's method are almost identical, except for two extra parameters in the Bailey's method to produce a distribution with slightly higher weights for less likely events. These extra parameters caused the compiled kernel to maintain too many registers.

The GPU is very sensitive to the amount of context each parallel thread of execution must maintain. If the number of active program contexts exceeds the available resources on the GPU, one must either spill register values to off-chip memory or reduce the number of active program contexts. Both of these two solution approaches carry performance implications, and the merged Bailey's method resulted in a 0.770× slow down over two separate steps.

8. CONCLUSION

This paper describes three levels of optimization for accelerating market Value-at-Risk estimation comprised of problem reformulation, module selection and implementation styling.

We illustrate that leveraging platform differences alone overlooks significant speedup opportunities and demonstrate our optimizations beyond module-level translation from a CPU-based implementation to a GPU-based implementation on a portfolio of 4096 risk factors. We found that reformulating the loss evaluation delivers a 60.3x speedup in delta-gamma VaR estimation. Choosing the distribution conversion module that contributes to faster algorithm convergence provides a further 2x speedup. Merging the QRNG with the distribution conversion module provides up to 1.85x speedup over executing modules separately.

Overall, we have achieved a 148x speedup by asserting multi-disciplinary domain knowledge in computational finance and software architecture compared to a baseline GPU implementation. Our optimized GPU implementation takes a little under 1 second to estimate the delta-gamma VaR to within a standard error of 0.1%.

9. ACKNOWLEDGMENTS

The authors would like to thank the referees for their comments and Prof. Claudio Albanese of Level3 finance and John-Anthony Murphy of Deutsche Bank Market Risk IT, London, for many illuminating discussions and helpful comments.

10. REFERENCES

[1] Agarwal V., Lurng-Kuo L. and Bader D.A., *Financial modeling on the cell broadband engine*, IEEE International Parallel and Distributed Processing Symposium, 2008, pp. 1–12, April 2008.

[2] Bailey R., *Polar generation of random variates with the t-distribution*, Mathematics of Computation 62, pp. 779–781, 1994.

[3] Bratley P. and Fox B.L., *Implementing Sobol's quasirandom sequence generator*, ACM Trans. on Math. Software 14(1), pp. 88–100, 1988.

[4] Caflisch R.E., Morokoff W. and Owen A., *Valuation of Mortgage backed securities using Brownian bridges to reduce effective dimension*, J. Comp. Finance 1, pp. 27–46, 1997.

[5] Dixon M. and Tan C.J.K., *Using Distributed Computers to Deterministically Approximate Higher Dimensional Convection-Diffusion Equations*, J. of Supercomputing 28(2), Kluwer, pp. 235–253, 2004.

[6] Giles M.B., Kuo F.Y., Sloan I.H. and Waterhouse B.J., *Quasi-Monte Carlo for finance applications*, ANZIAM Journal 50, pp. 308–323, 2008.

[7] Glasserman P., Heidelberger P. and Shahabuddin P., *Variance Reduction Techniques for Estimating Value-at-Risk*, Management Science 46(10), pp. 1349–1364, October 2000.

[8] Glasserman P., *Monte Carlo Methods in Financial Engineering*, Appl. of Math. 53, Springer, 2003.

[9] Joe S. and Kuo F., *Remark on algorithm 659: implementing Sobol's quasirandom sequence generator*, ACM Trans. on Math. Software 29(1), pp. 49–57, 2003.

[10] Jondeau E., Poon S. and Rockinger M., *Financial Modeling Under Non-Gaussian Distributions*, Springer Finance, 2007.

[11] Jorion P., *Value-at-Risk: the new benchmark for managing financial risk*, 3rd ed., New York, McGraw-Hill, 2007.

[12] Moskowitz B. and Caflisch R.E., *Smoothness and dimension reduction in quasi-Monte Carlo methods*, J. Math. Comp. Modeling 23, pp. 37–54, 1996.

[13] Moro B., *The Full Monte*, Risk Magazine 8(2), pp. 57–58, February 1995.

[14] Singla N., Hall M., Shands B. and Chamberlain R.D., *Financial Monte Carlo simulation on architecturally diverse systems*, Workshop on high performance computational finance, Supercomputing 08, pp. 1–7, 2008.

[15] Srinivasan A.,*Parallel and Distributed Computing Issues in Pricing Financial Derivatives through Quasi Monte Carlo*, Proceedings of the 16th international Parallel and Distributed Processing Symposium 252, IEEE, 2002.

[16] Thomas D.B. and Luk W., *Multivariate Gaussian Random Number Generation Targeting Reconfigurable Hardware*, ACM Trans. Reconfigurable Technol. Syst. 1(2), issn 1936–7406, pp. 1–29, ACM, 2008.

[17] Wang X. and Sloan I. H., *Low discrepancy sequences in high dimensions: How well are their projections distributed?*, J. Comput. Appl. Math. 213 (2), Mar. 2008.

APPENDIX

A. ANALYTIC MOMENTS OF THE PORTFOLIO DISTRIBUTION

Under the affine transformation $Y_t = c + QX_t$, the moments of the portfolio distribution can be expressed in terms of the moments of the distributions of the vector X_t of i.i.d. random variables. From equation 5, the expected portfolio loss conditional on the filtration $\mathfrak{F}_t$ of X_t is given by

$$\begin{aligned}\mathbb{E}_t[\Delta P_t] &= \int_{\mathbb{R}^N} f(x_1,\ldots,x_N)\Delta P_t \; dx_1 \ldots dx_N,\\ &= \sum_i \int_{\mathbb{R}^N} f(x_1,\ldots,x_N)(q_i x_i + \hat{\lambda}_i x_i^2) \; dx_1 \ldots dx_N,\\ &= \sum_i q_i \int_{\mathbb{R}} f(x_i) x_i \; dx_i + \hat{\lambda}_i \int_{\mathbb{R}} f(x_i) x_i^2 \; dx_i,\\ &= \sum_i q_i \mathbb{E}[x_i] + \hat{\lambda}_i \mathbb{E}[x_i^2],\end{aligned} \tag{6}$$

and the first moment is $\mathbb{E}_t[\Delta^2 P_t] - \mathbb{E}_t^2[\Delta P_t]$ where

$$\mathbb{E}_t[\Delta^2 P_t] = \sum_i q_i^2 \mathbb{E}[x_i^2] + q_i \hat{\lambda}_i \mathbb{E}[x_i^3] + \hat{\lambda}_i^2 \mathbb{E}[x_i^4]. \tag{7}$$

Line 3 of equation 6 has used the assumption that

$$f(x_1, x_2, \ldots, x_N) = \prod_i f(x_i) \tag{8}$$

is an $N-$variate elliptic density function of uncorrelated random variables (x_i).