Multiplierless Algorithm for Multivariate Gaussian Random Number Generation in FPGAs

David B. Thomas, *Member, IEEE*, and Wayne Luk, *Fellow, IEEE*

***Abstract*— The multivariate Gaussian distribution is used to model random processes with distinct pair-wise correlations, such as stock prices that tend to rise and fall together. Multivariate Gaussian vectors with length n are usually produced by first generating a vector of n independent Gaussian samples, then multiplying with a correlation inducing matrix requiring $O(n^2)$ multiplications. This paper presents a method of generating vectors directly from the uniform distribution, removing the need for an expensive scalar Gaussian generator, and eliminating the need for any multipliers. The method relies only on small read-only memories and adders, and so can be implemented using only logic resources (lookup-tables and registers), saving multipliers, and block-memory resources for the numerical simulation that the multivariate generator is driving. The new method provides a ten times increase in performance (vectors/second) over the fastest existing field-programmable gate array generation method, and also provides a five times improvement in performance per resource over the most efficient existing method. Using this method, a single 400-MHz Virtex-5 FPGA can generate vectors ten times faster than an optimized implementation on a 1.2-GHz graphics processing unit, and a hundred times faster than vectorized software on a general purpose quad core 2.2-GHz processor.**

***Index Terms*— Field-programmable gate array (FPGA), Monte Carlo simulation, multivariate samples, random number generation.**

I. Introduction

THE multivariate Gaussian distribution is used to capture simple correlations between related stochastic processes, such as the stock prices of companies in similar business sectors, where the stock prices of companies in the sector tend to rise and fall together. To simulate the behavior of such processes, multivariate Gaussian random number generators (MVGRNGs) are used to generate random samples, which are then used to drive Monte Carlo simulations. Such simulations often require a huge number of independent runs in order to provide an accurate answer, such as the value-at-risk calculations performed by financial institutions, which are used to estimate how much money the institution might lose over the next day.

Such long-running simulations are an ideal target for field-programmable gate array (FPGA) acceleration, as they offer abundant parallelism, and are computationally bound [1]–[3]; however, they are reliant on a fast, resource-efficient, and accurate source of random samples from the multivariate Gaussian distribution. This paper improves on previous methods [4], [5] for FPGA-based MVGRNG, by developing a method that provides a large increase in performance, while limiting resource usage to standard bit-wise logic elements.

Our key contributions are:

1) an algorithm for generating samples from the multivariate Gaussian distribution using only uniform random bits, table-lookup, and addition;
2) a hardware architecture for implementing an MVGRNG using only lookup-tables (LUTs) and flip-flops (FFs), which allows a regular densely-packed placement strategy and achieves 500 MHz+ clock speeds;
3) correction methods for achieving the correct statistical properties, even when using small fixed-point tables;
4) an evaluation of the statistical properties of the table-based MVGRNG, demonstrating that the algorithm and correction methods produce high quality random vectors;
5) a comparison with two existing FPGA generation methods, showing more than ten times the performance of the fastest method, and five times the performance per resource of the most efficient method;
6) a comparison of FPGA generation performance with GPU and CPU implementations, showing the FPGA can provide ten times the performance of an optimized GPU generator, and a hundred times that of a quad-core SIMD-optimized CPU generator.

This paper was originally presented in a conference paper [6]; this paper provides a much better intuitive understanding of how the generator point-set operates (Section III), includes more detail about how to correct for finite-precision effects in tables and measures the run-time cost of creating them (Section V), and adds a resource efficient method for loading new matrices at run-time without requiring configuration bit-stream manipulation (Section IV).

II. Multivariate Gaussian Random Numbers

Generation of the univariate Gaussian distribution with a specific mean, μ, and variance, σ^2, is achieved by first

Manuscript received February 18, 2012; revised July 29, 2012; accepted October 18, 2012. Date of publication January 11, 2013; date of current version October 14, 2013. This work was supported in part by the U.K. Engineering and Physical Sciences Research Council under Grant EP/D062322/1 and Grant EP/C549481/1, and Alpha Data and Xilinx.

D. B. Thomas is with the Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, U.K. (e-mail: dt10@ic.ac.uk).

W. Luk is with the Department of Computing, Imperial College London, Imperial College London, London SW7 2AZ, U.K. (e-mail: wl@doc.ic.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2012.2228017

generating a standard Gaussian variate r with mean zero and variance one, then applying a linear transformation

$$x = \sigma r + \mu \tag{1}$$

where $r \sim N(0, 1)$. Note that the standard Gaussian variate is multiplied by the standard deviation (SD) σ, rather than the variance σ^2.

Generation of a multivariate Gaussian distribution is very similar, except in this case, each output sample is, a length n vector $\mathbf{x}$. The mean and variance also increase in dimension, so the mean is a length n vector $\mathbf{m}$, and the variance becomes an $n \times n$ covariance matrix Σ. The covariance matrix is a symmetric matrix, which captures the variance of each output component on the diagonal, and the correlations between each component on the off-diagonal elements.

Generation is similar to the univariate linear transform, except the starting point is a vector $\mathbf{r}$ of n independent identically distributed (IID) standard Gaussian random numbers

$$\mathbf{x} = \mathbf{A}\mathbf{r} + \mathbf{m}. \tag{2}$$

The matrix $\mathbf{A}$ is conceptually similar to the SD: just as the variance is the SD squared, so $\mathbf{A}\mathbf{A}^T = \Sigma$. However, in the multivariate case there is considerable freedom in the selection of $\mathbf{A}$, as there are many ways of decomposing Σ.

One method is to perform Cholesky decomposition of the correlation matrix, producing a lower-triangular matrix. This choice has computational and storage advantages: only $n(n+1)/2$ of the elements are nonzero and must be stored, so only $n(n+1)/2$ multiplications are required. A disadvantage is that the Cholesky decomposition only works with positive definite covariance matrices Σ-many matrices constructed from estimates may be singular or very close to singular [7].

An alternative method is to use the singular value decomposition (SVD) algorithm. This decomposes the matrix into an orthogonal matrix $\mathbf{U}$ and a diagonal matrix $\mathbf{S}$, such that $\Sigma = \mathbf{U}\mathbf{S}\mathbf{U}^T$. This decomposition allows the construction of the solution $\mathbf{A} = \mathbf{U}\sqrt{\mathbf{S}}$. The disadvantage of the SVD-based construction is that in general all the elements of the matrix are nonzero, resulting in an n^2 cost in both the number of stored elements, and in the number of multiply-accumulates per transformed vector. However, the SVD algorithm is able to handle a wider range of covariance matrices, such as ill-conditioned matrices that are very close to singular, and reduced rank matrices, where the output vector depends on fewer than n random factors. Such difficult covariance matrices frequently occur in practice [7], so this paper focuses on the use of a dense SVD-style decomposition.

III. Generation Using LUTs and Adders

The standard generation method uses direct matrix multiplication, forming each output element from a linear combination of $\mathbf{r}$ (the vector of n IID Gaussian samples)

$$x_i = m_i + \sum_{j=1}^{n} a_{i,j} r_j, \quad i \in 1, \ldots, n. \tag{3}$$

If there is no advance knowledge about the covariance matrix and the SVD decomposition is used, this requires n^2 multiply-accumulations. In addition, this method also requires the generation of the n elements of $\mathbf{r}$, which are IID standard Gaussian samples. Both the generation of $\mathbf{r}$ and the multiplication with $\mathbf{A}$ require significant resources (i.e., DSPs and block-RAMs in an FPGA), so simplifying the process and reducing the resource cost is highly desirable.

The method proposed in this paper is that, instead of generating expensive independent Gaussian samples and then inducing the desired covariance structure with n^2 multiplications, cheap uniform samples will be converted to correlated Gaussian samples using n^2 table-lookups. Each table contains a pre-calculated discretized Gaussian distribution with the correct SD, so the only operations required are table-lookups and additions.

The following text frequently refers to tables, which in this context means an array of read-only elements (a ROM) which will be implemented in the FPGA using LUTs. Unless otherwise specified, each table contains k elements, and is indexed using the syntax $L[i]$ to access table elements $L[1], \ldots, L[k]$. Where arrays of tables are used, sub-scripts identify a table within the array, which can then be indexed as for a standalone table, e.g., $L_{2,3}$ [4]. Tables can also be interchangeably treated as discrete random number generators, where the discrete PDF of each table is given by assigning each element of the table an equal probability of $1/k$. For example, if u is an IID uniform sample between 0 and 1, a random sample x from table L is generated as

$$x = L\left[\lceil uk \rceil\right] \tag{4}$$

where $u \sim U(0, 1)$. The central idea of this method is to construct an $n \times n$ array of tables $\mathbf{G}$, such that the discrete distribution of each table $\mathbf{G}_{i,j}$ approximates a Gaussian distribution with SD $\mathbf{A}_{i,j}$

$$\mathbf{G}_{i,j} \sim N(0, \mathbf{A}_{i,j}). \tag{5}$$

Now instead of starting from a Gaussian vector $\mathbf{r}$, the input is an IID uniform vector $\mathbf{u}$. Generation of each output element uses each element of $\mathbf{u}$ as a random index into the table, then sums the elements selected from each table

$$x_i = m_i + \sum_{j=1}^{n} L_{i,j}\left[\ \lceil u_j k \rceil\ \right], \quad i \in 1, \ldots, n. \tag{6}$$

In practice k will be selected to be a power of 2, so each element of $\mathbf{u}$ is actually a uniform integer constructed from the concatenation of $\log_2(k)$ random bits.

The simplest method of generating a table-based approximation to the Gaussian distribution is direct cumulative distribution function (CDF) inversion. To generate a table L with SD σ, table elements are chosen according to

$$L[i] = \sigma \Phi^{-1}(i/(k+1)), \quad i \in 1, \ldots, k \tag{7}$$

where $\Phi^{-1}(.)$ is the Gaussian inverse CDF. The table $\mathbf{G}$ corresponding to a given target matrix $\mathbf{A}$ can then be specified as

$$\mathbf{G}_{i,j}[z] = \mathbf{A}_{i,j}\Phi^{-1}(z/(k+1)), \quad i, j \in 1, \ldots, n, \quad z \in 1, \ldots, k. \tag{8}$$

Construction of $\mathbf{G}$ allows the direct transformation of uniform samples (random bits) into multivariate Gaussian samples using 6, requiring only table-lookups and addition.

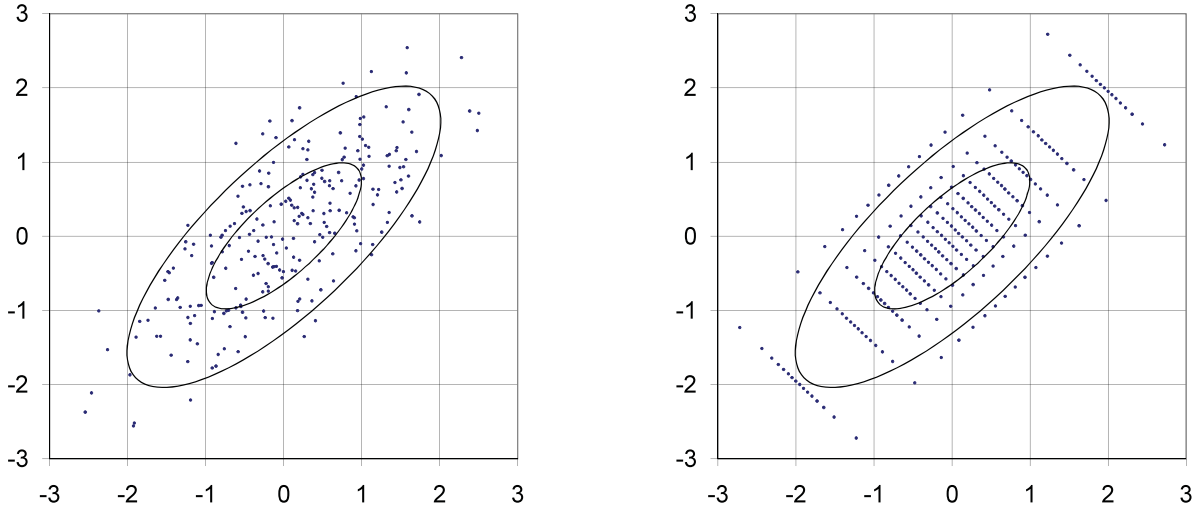

Fig. 1. Comparison of random correlated points from a traditional random number generator to the fixed lattice of a table-based bivariate generator with 16 elements per table.

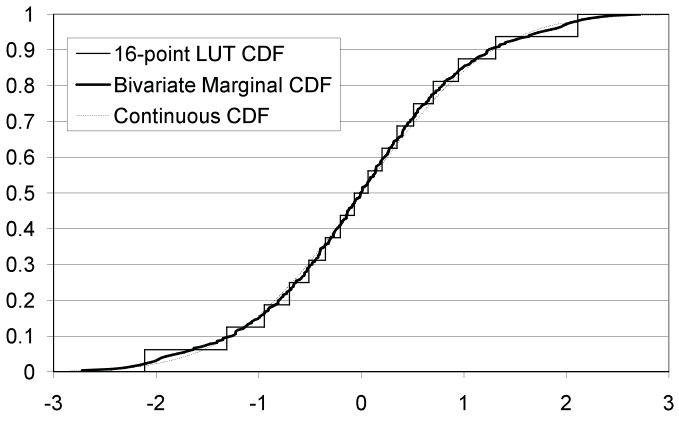

Fig. 2. Comparison of the CDF of a univariate 16-element table against the marginal CDF of a bivariate generator using two 16-element tables, showing convergence to the continuous CDF.

Replacing continuous samples with discrete tables means that the output distribution range is no longer continuous; instead, each sample is drawn from within a discrete multidimensional lattice. Fig. 1 shows an example of this process. On the left is a set of samples drawn from a standard bivariate random number generator using 2, showing that there is no structure to the set of points.

The right side shows the point-set for a bivariate table using two tables with $k = 16$, and the contrast with the continuous version is clear. Each sample corresponds to randomly selecting one of the discrete points in the lattice and returning the x and y co-ordinates as the random vector. The point-set seems alarmingly regular, but it is important to remember that the application consuming the random samples actually observes the marginal distribution, i.e., the projection onto each axis.

Fig. 2 shows the effect of this projection process, by graphing the CDF of the marginal distribution. The stepped line shows the raw CDF of the univariate 16-point table, showing clear discontinuities. However, the marginal distribution of the bivariate table is much closer to the continuous version, and appears (visually, at least) much more continuous.

This smoothing process increases with both the number of dimensions and the size of each table, as the number of points in the lattice grows as k^n. As will be described in the next section, modern architectures will efficiently support $k = 128$, so even for a bivariate generator, the point set can contain 2^{14} points. For higher dimension sets with $n \geq 2$ the point-set grows rapidly, and in Section VI a more rigorous analysis is applied to show that the achieved quality is good enough for use in Monte Carlo simulations.

IV. Hardware Architecture

The central idea in this paper, of replacing Gaussian samples and multipliers with uniform samples and tables, allows for many types of possible implementations. For example, the tables can be implemented using LUTs or block-RAMs, and the generator can vary in throughput from 1 to n cycles per generated vector. This paper focuses on the highest performance mode of the generator, to provide the maximum contrast with previous implementations, while still providing good efficiency and quality.

The specific choices made are as follows.

1) *Logic Resources Only:* tables are implemented using LUTs, so the only resources used are LUTs and FFs (no DSPs or block-RAMs).
2) *Parallel Generation:* the generator operates in a fully parallel mode, providing one new n-element vector per cycle, unlike previous approaches which generated one vector for every n cycles.
3) *Maximum Clock Rate:* the generator operates at the maximum realistic clock-rate for the target FPGA. For the Virtex-5 this is effectively 550 MHz, as this is the maximum clock rate of the DSP and RAM blocks that will be used in the simulation that the generator is driving.
4) *Regular Architecture:* simulations typically consume almost all resources in the FPGA (due to replication of simulation cores), and a regular, explicitly placed,

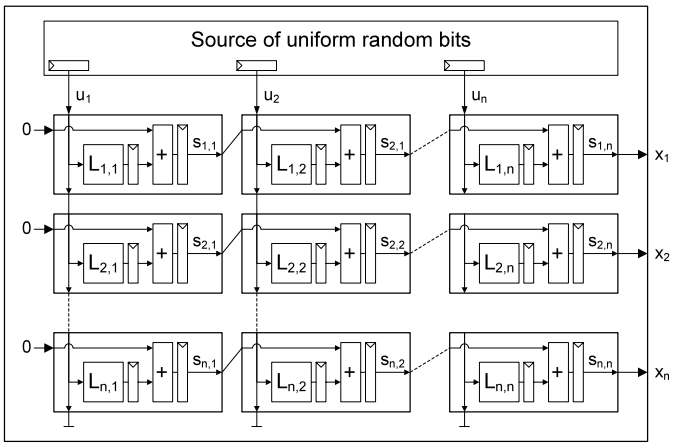

Fig. 3. Tiled hardware architecture for table-based generation.

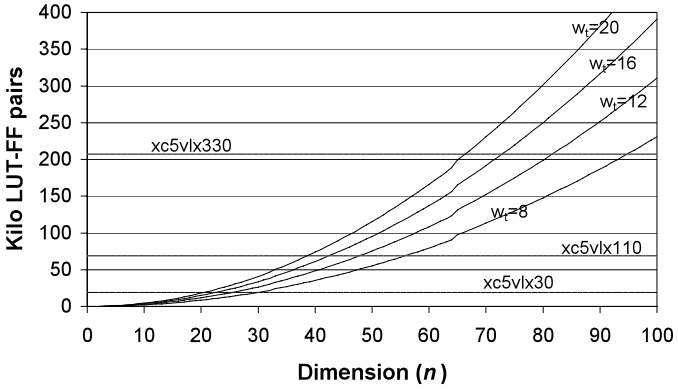

Fig. 4. Resource usage for different vector lengths and table widths.

highly routeable, and generator allows fast place-and-route while still achieving high overall clock-rate.

5) *No Matrix Specialization:* the generator is not optimized for a specific correlation matrix, and should support the loading of any correlation structure without structural modification.

The table-based generator maps naturally into a regular pipelined structure, shown in Fig. 3. At the top is a random bit source, which generates a new vector $\mathbf{u}$ for every cycle. The elements of $\mathbf{u}$ are broadcast vertically down through the cells, and used to select one element from each table. The selected elements are then accumulated horizontally from left to right by implementing the function $s_{i,j} = s_{i,j-1} + L_{i,j}[u_j]$, resulting in a new vector output at the right every cycle. Each node in the grid is very similar to a KCM constant coefficient multiplier, though due to the specialized table entries it performs a more complicated function.

This LUT-based architecture is very small, but there are still tradeoffs in terms of area versus quality, where quality can be interpreted as the number of table elements–if k is larger, then the marginal distributions will be closer to the Gaussian and appear less discrete. In Section IV-B, a low-area solution is examined, but first a higher quality method is examined, which aims to maximize the number of elements.

A. Quality Optimized Architecture

A useful optimization for increasing the effective number of elements per table is to take advantage of the symmetry of the tables. If the tables have a mean of zero, they have the property that $L[i] = -L[k - i + 1]$, so it is only necessary to store the elements $L[1], \ldots, L[k/2]$. The half-table is now indexed by all but the most-significant bit of the uniform index, while the most significant uniform bit is used to select whether the table value is added or subtracted from the accumulator. Note that the values stored in the table must be signed, so that it is possible to encode both positive and negative values from $\mathbf{A}$. This optimization doubles the effective table size of each LUT; for example, a Virtex-5 6-LUT can support a 128 element table, rather than just 64.

The exact resource utilization can be calculated from the following parameters and assumptions.

1) Table elements have width w_t.
2) Accumulators have a width $w_a = w_t + \lceil \log_2 n \rceil$.
3) Each LUT can implement a $1 \times k$ bit LUT (using the symmetry optimization).
4) The uniform generator is implemented using a LUT-optimized RNG [8], requiring one LUT-FF pair per uniform bit.

The resource usage of the generator then breaks down as:

1) uniform RNG: $n \log_2 k$;
2) tables: $n^2 w_t$;
3) accumulators: $n^2 w_a = n^2(w_t + \lceil \log_2 n \rceil)$.

Total resource utilization for the entire random number generator is

$$n(\log_2 k + n(2w_t + \lceil \log_2 n \rceil)). \tag{9}$$

This describes the number of LUTs, the number of FFs, and also the number of fully-occupied LUT-FF pairs, as all elements use a LUT connected to a FF.

Fig. 4 charts the increase in resource utilization as n increases, for table widths from 8 to 20. Also shown are the number of LUT-FF pairs in the smallest (xl5vlx30), intermediate (xc5vlx110), and largest (xc5vlx330) Virtex-5 parts.

In principle it is possible to reach dimensions up to around 100 in a large Virtex-5, such as the xc5vlx330, but it is important to remember that the generator has to drive something, and it probably has to be on the same FPGA. A generator with $n = 100$ and $w_a = 8$ running at 550 MHz will generate 55 Gb/s of data, so it would be very difficult to dedicate an entire FPGA to multivariate generation and ship the vectors elsewhere. So the practical maximum is around $n = 64$.

The regularity of the architecture makes it simple to explicitly place all components in the generator, reducing the load on the place-and-route tools, and making it much easier to achieve high clock-rates for the overall design. In this paper, the simple placement strategy shown in Fig. 6 is used, where the accumulator is simply stacked on top of the table.

B. Area-Optimized Architecture

Both the Virtex-5 and Stratix-6 FPGA architectures support LUTs, which can be fractured in some way. In Virtex-5, each

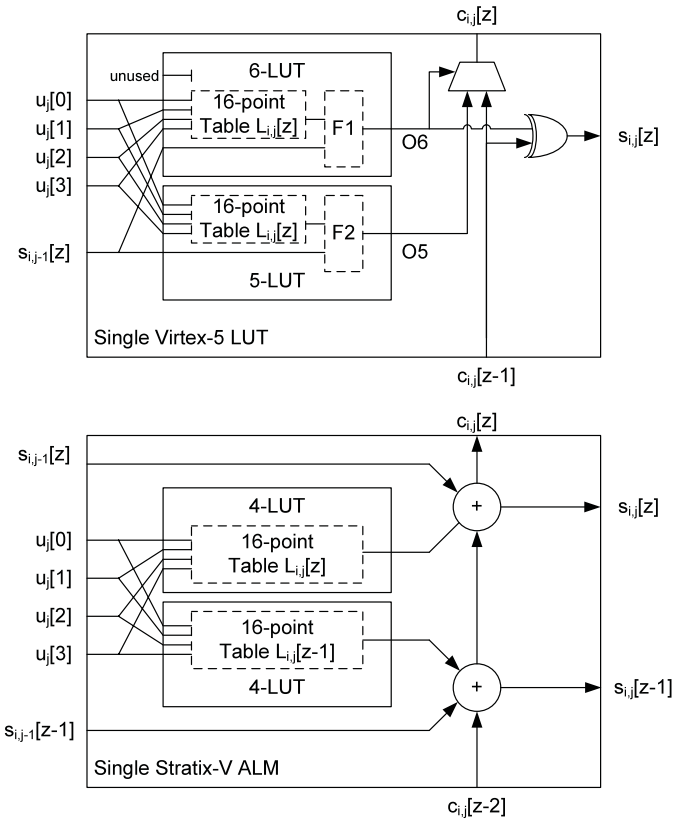

Fig. 5. Tightly packed RNG cell implementations for Virtex-5 and Stratix-6, combining both table and adders into one LUT.

6-LUT actually contains two outputs: one (O6) is a 6-LUT, while the other (O5) is a 5-LUT, which shares five of the O6 inputs. These two outputs are connected to a carry-mux and XOR-gate, allowing each dual-output LUT to add any two of its inputs. However, the extra three LUT inputs can also be used, so it is possible to perform a 16-element table lookup before performing the add, by feeding in the four-bit uniform index along with the accumulator bit to be accumulated. The top of Fig. 5 shows this approach, with z representing the bit index within a given accumulator.

The Stratix-V architecture uses a fracturable adaptive logic module (ALM), which can combine multiple smaller LUTs into two 6-LUTs with four common inputs, and also supports full-adders on the carry chain. Unfortunately the full-adder occurs before the components of the 6-LUT are re-combined, meaning that only the output of the earlier 4-LUTs is available. This means that a single ALM can implement two bits of an RNG cell with $k = 16$, shown in the bottom of Fig. 5, with z and $z - 1$ representing consecutive bits within an accumulator.

Both these packing methods can be implemented by the vendor's synthesis tools, providing an absolute saving of $\omega_t n^2$ over the resources given by 9. This approximately halves the relative resource cost, but comes at the cost of reducing table size from 128 to 16. Given the previous method is already so cheap in terms of area, the high-quality version is evaluated in the rest of this paper, as the area-optimized version is only appropriate when a designer is willing to check that the quality-area tradeoff is acceptable in their application.

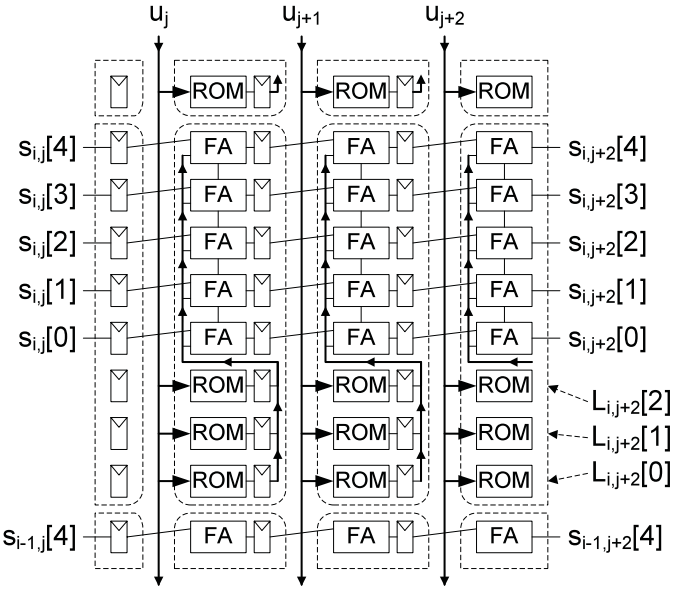

Fig. 6. Practical dense placement for quality-optimized hardware realization.

C. Implementation Results

The quality-optimized architecture has been described in VHDL, using Virtex-5 primitives with relationally placed macro (RPM) constraints. When mapped into hardware, the resource utilization exactly matches the predictions of 9. For all $n \leq 16$ and $\omega_t \leq 16$, this strategy provides 550-MHz operation in an xc5vlx330 device (post place-and-route timing).

As n grows larger, the fan-out of the uniform generator lines begins to reduce the clock rate, as each uniform bit must drive n ROM address bits spread over a tall column. The overall shape of the RPM'd grid may also fit poorly into a given device; for example, it may become too tall or wide, or specialized devices, such as DSP columns may intrude.

Fortunately, the regular data-flow in the architecture, combined with the IID property of the uniform random inputs, makes it simple to both insert buffering and to fragment the grid. An arbitrary number of registers can be inserted into the left to right path through the accumulators, as long they are inserted on vertical lines through the architecture. Similarly the top to bottom path from the uniform generators can be buffered with an arbitrary number of register levels, as long as the total delay from each uniform output bit to each ROM input is the same.

The approach used here is to scale generators up using a two-level structure, where the overall generator is formed from a grid of smaller sub-generators. Each sub-generator uses the relatively placed design shown in Fig. 6, with the maximum path being FF-LUT-FF. Each sub-generator grid is then packaged as a single component, with registers on all the inputs. Fig. 7 shows this architecture for $n = 9$ with 3×3 sub-generators.

This two-level decomposition has three key advantages when generating large vectors. First, it allows the majority of the logic to be relatively placed, while still providing freedom to the placer to adapt to the specific device. Second, all the

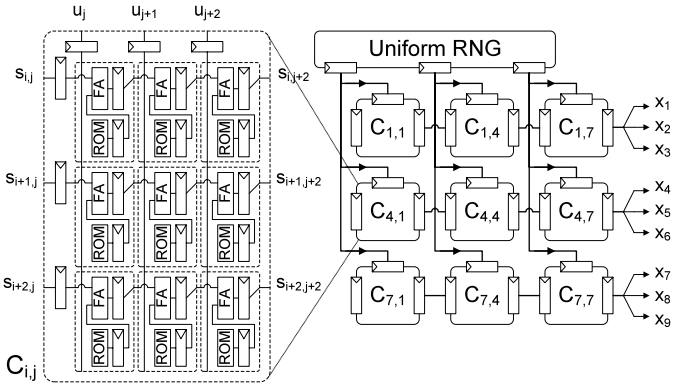

Fig. 7. Two-level pipelined decomposition of generator for large n.

connections between sub-generators are direct FF-to-FF paths, providing more slack to the place-and-route tools. Finally, the fan-out of the uniform generator is reduced to the number of sub-generators, rather than the number of elements.

Using this approach, any generator with $w_t = 16$ and $n \leq 64$ that occupies 90% or less of a Virtex-5 device's logic resources provides 550-MHz performance (from post place-and-route timing analysis, using highest speed grade part). This holds true for all device sizes tried, up to and including the xc5vlx330 device.

In the largest devices, the limiting factor is the 550-MHz global clock limits, not the generator itself; for smaller devices which support 710-MHz global clocks, the generator logic becomes the limit. In the xc5vlx85 observed reported clock rates exceeded 600 MHz when using a 550-MHz target constraint. However, 550 MHz is the useful maximum clock rate, as it is extremely unlikely that any circuit which is receiving the random vectors will be able to operate above 550 MHz.

D. Run-Time Loading of Tables

The proposed structure uses RAMs to store the tables, but so far contains no explicit means of modifying the tables at run-time. In a Monte Carlo simulation, each distinct correlation matrix will be used for many millions or billions of cycles, so it is important to make sure that very few resources are dedicated just to changing matrices.

One method with zero resource overhead is to use bit-stream manipulation to modify the contents of the tables directly, simplifying the circuit, and reducing resources. As the covariance matrix is entirely specified by the table contents, the only part that needs to be modified is the contents of the LUTs containing the tables, with no changes to routing or other more complex parts of the architecture. Modifying LUT values is simply a case of directly changing bits within the bit-stream before FPGA configuration, and is very fast, much quicker than the actual reconfiguration.

When bit-stream modification is unacceptable, there are two simple possibilities for modifying the covariance matrix using on-chip circuitry. One option in the Virtex-5 family is to configure the table LUTs as SRL32 primitives. This allows the table data to be shifted in serially with no extra logic per cell, but has the disadvantage of reducing the effective table width to $k = 64$ (as the SRL32 can only act as a 32 entry table). If data is fed into the table one table entry (i.e., w-bits) per cycle, then it requires at least n^2k cycles. For example, with $n = 16$, $k = 64$, and a clock rate of 550 MHz, this would require 0.03 ms.

A better option, which is appropriate for Altera architectures and also allows full use of all 64 elements of the LUT-RAMs in Xilinx architectures, is to treat the LUTs as RAMs, and explicitly write to each address in turn. However, great care must be taken to ensure that the resource cost of the $O(n^2)$ part of the grid remains unchanged, by re-using existing resources within each grid cell.

Fig. 8 demonstrates one technique for achieving this without requiring any extra resources per cell. Part (a) shows the modified architecture - the only additions to the basic cells are:

1) a data-path from the output of the accumulator to the input of the RAM;
2) a new synchronous clear signal for the output register of the RAM, shared among all cells;
3) a new write enable signal for the RAM, shared among all cells.

These new signal paths utilize existing inputs of the logic, and the shared control signals can be controlled with a single state machine shared for the whole array. The control signals must fan-out to n^2k locations, but can be pipelined with a single one-bit register per island of Fig. 7 for very little overhead, or in modern architectures can often use the spare registers present in each CLB.

Given these additional paths, the matrix can be loaded over k phases, with each phase loading one table entry across the entire grid. Within each phase $1 \leq z \leq n$, the steps are:

1) assert the reset signal of the RAM output register, forcing it to zero [Fig. 8(b)]. Now any data passed along the accumulator chain will remain unchanged, forming a shift-register through the accumulators;
2) the tables entries $G_{i,n}[z], G_{i,n-1}[z], \ldots, G_{i,1}$ are fed into the chain on successive cycles [Fig. 8(c)].
3) once all entries are in the correct accumulator register, the address z is fed in through the u_i input, and all table entries for the z index are written [Fig. 8(d)]. This exploits the fact that LUT-opt generators already contain a shift register used for state initialization, and so can be used to present any pattern $u_1, \ldots, u_n$ at the RAM address inputs.

After all entries have been loaded, the two control signals are both set to zero, and each cell reverts to the standard generation mode [Fig. 8(e)].

At a minimum each phase requires n cycles, with a bandwidth of wn bits per cycle, for a total loading time of nk cycles. However, the table entries must be sourced from external IO or an off-chip RAM, so it is more practical to stream the data one w-bit word per cycle, resulting in a minimum configuration time of n^2k cycles, the same as the serial loading technique. In the implementation developed for this paper, there is a small over-head between phases, resulting in a minimum loading time of $(n^2 + 3)k$ cycles, and an overhead of approximately 25 LUTs to implement the state machine.

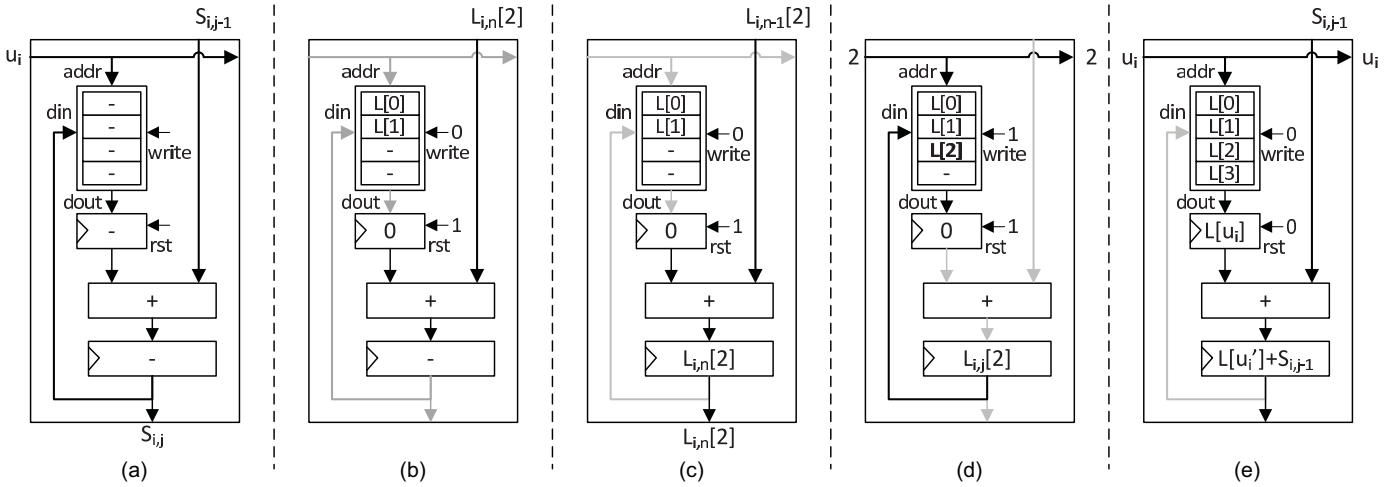

Fig. 8. Direct loading matrices of matrices into LUT-RAMs. (a) Uninitialised. (b) Begin loading entry 2. (c) Finish loading entry 2. (d) Write entry 2. (e) Generate numbers.

V. Finite-Precision Effects

The generation algorithm described in Section III is asymptotically correct, but the architecture described in Section IV introduces limits due to the constraints of real hardware. The first problem is that, in practice, tables must be relatively small ($k \leq 2^{16}$), which affects the accuracy of the Gaussian approximation provided by 7. The second problem is that the tables must be stored in a finite-width fixed-point format, which will also affect the accuracy of the approximation. This section shows how to control the effects of these practical limitations by modifying the table contents–no changes to the hardware architecture are required.

A. Correction of Table Moments

The inversion method for generating table-based Gaussian approximations (7) is simple, but in practice is not very good. As $k \to \infty$ this approximation converges on the normal distribution, but for practical values of k, the approximation is rather bad. In particular, the second and fourth central moments (variance and kurtosis) are too small, as the approximation does not extend very far into the tails.

The SD (i.e., $\sqrt{\text{variance}}$) in particular is very important, as the marginal SD of each element of the multivariate output is derived from the sum of the SDs of the components used to build it. It is important to realize that although the central limit theorem (CLT) states that the sum of increasing IID variates converges to some Gaussian distribution, it does not necessarily converge on the specific parametrization of the Gaussian hoped for by the user. To ensure the marginal SD of each element is correct, the marginal SD of each table element must be as precise as possible.

The raw central moments of a table can be calculated as

$$\mu_1(L) = \frac{1}{k}\sum_{i=1}^{k} L[i] \tag{10}$$

$$\mu_d(L) = \frac{1}{k}\sum_{i=1}^{k} (L[i] - \mu_1(L))^d\,. \tag{11}$$

If the table is symmetric and k is divisible by two, as in the optimization suggested in Section IV-A, then $L[i] = -L[n+1-i]$. This means that all the odd moments, from mean, skewness, and up, reduce to zero. With a mean of zero, the even moments reduce to

$$\mu_d(L) = \frac{1}{k}\sum_{i=1}^{k} L[i]^d. \tag{12}$$

Transforming the table to give a SD of 1 can be achieved by scaling all table elements by some linear factor c_1 to produce a new table L'

$$1 = \mu_2(L') = \frac{1}{k}\sum_{i=1}^{k}(c_1 L[i])^2 = \frac{c_1^2}{k}\sum_{i=1}^{k} L[i]^2. \tag{13}$$

If the sum of powers of the original table is defined as a constant scalar value

$$L^{(d)} = \sum_{i=1}^{k} L[i]^d \tag{14}$$

this results in

$$k = c_1^2 L^{(2)} \quad c_1 = \sqrt{\frac{k}{L^{(2)}}}. \tag{15}$$

This gives a simple linear correction for correcting the SD of a given table.

However, it is better to transform the table into a new table L', such that the new table matches both the variance and kurtosis of the standard (unit SD) normal

$$\mu_1(L') = 0 \quad \mu_2(L') = 1 \quad \mu_3(L') = 0 \quad \mu_4(L') = 3. \tag{16}$$

Correcting the kurtosis is desirable because many properties of convergence in simulations rely on the accuracy of the moments. In addition, correcting the kurtosis of the tables provides a measurable improvement to the accuracy of the marginal PDFs at the outputs, as shown later in Section VI.

The table can be transformed by applying a cubic polynomial stretch to the entries, using only odd powers to preserve symmetry

$$L'[i] = c_1 L[i] + c_3 L[i]^3. \tag{17}$$

To match the moments of the transformed table, the following two constraints must be satisfied:

$$1 = \mu_2(L') = \frac{1}{k}\sum_{i=1}^{k}\left(c_1 L[i] + c_3 L[i]^3\right)^2 \tag{18}$$

$$3 = \mu_4(L') = \frac{1}{k}\sum_{i=1}^{k}\left(c_1 L[i] + c_3 L[i]^3\right)^4. \tag{19}$$

Expanding and rearranging 18 gives

$$k = \sum_{i=1}^{k}\left(c_1^2 L[i]^2 + 2c_1 c_3 L[i]^4 + c_3 L[i]^6\right)^2 \tag{20}$$

$$= c_1^2\sum_{i=1}^{k} L[i]^2 + 2c_1 c_3\sum_{i=1}^{k} L[i]^4 + c_3^2\sum_{i=1}^{k} L[i]^6. \tag{21}$$

Equation 19 can be similarly decomposed into terms containing independent sums of powers.

The solution (c_1, c_3) can then be found as the roots of a polynomial system with two unknowns

$$k = c_1^2 L^{(2)} + 2c_1 c_3 L^{(4)} + c_3^2 L^{(6)} \tag{22}$$

$$3k = c_1^4 L^{(4)} + 4c_1^3 c_3 L^{(6)} + 6c_1^2 c_3^2 L^{(8)} + 4c_1 c_3^3 L^{(10)} + c_3^4 L^{(12)}. \tag{23}$$

Such systems can be solved with automated root finders, for example the algsys function supplied with Maxima [9]. If the chosen root finder can only achieve a close approximation (for example, for this problem, the algsys method only achieves a relative error of around 10^{-8}), then the problem is relatively well-behaved near the roots, so Newton–Raphson polishing can reduce the error level to any degree desired.

Table I shows the cubic corrections for binary-power sized tables from eight up to 65 536, with the uncorrected entries calculated according to 7. The full procedure for calculating $\mathbf{G}'$, the cubic corrected tables, is:

1) identify the relevant (c_1, c_3) from Table I;
2) form the uncorrected set of tables $\mathbf{G}$, using 8;
3) create $\mathbf{G}'$ using the cubic transform

$$\mathbf{G}'_{i,j}[z] = c_1\mathbf{G}_{i,j}[z] + c_3\mathbf{G}_{i,j}[z]^3, \quad i, j \in 1, \ldots, n,\ z \in 1, \ldots, k.$$

The given corrections all result in a relative error for both SD and kurtosis of less than $10^{-13.7}$, assuming double precision calculations. The typical error is closer to 10^{-15}, but values around $k = 8192$ caused Maxima's Newton–Raphson polisher to fail. A rational approximation was used to interpolate between accurate values for c_1 and c_3 on either side, but resulted in slightly lower accuracy for this point.

Although it is possible to create tables with nonbinary power sized tables, there is no real need, as it is too expensive to generate random indices to index such tables. For table sizes below eight, the suggested correction method does not converge, but arguably a table size of eight is the minimum one might use–the vast majority of FPGAs use four-LUTs or higher, so 16 is likely to be the smallest table size used in practice.

TABLE I
CONSTANTS FOR CORRECTING VARIANCE AND KURTOSIS OF GAUSSIAN LUTS WITH DIFFERENT SIZES

k	c_1	c_3	Err(m_6)	Err(m_8)
8	0.5537484093	2.777255135e−1	$10^{-1.1234}$	$10^{-0.4005}$
16	0.8554643151	8.028744579e−2	$10^{-1.5606}$	$10^{-0.6602}$
32	0.9348314060	3.311529112e−2	$10^{-1.9499}$	$10^{-0.9244}$
64	0.9669892318	1.567406031e−2	$10^{-2.3248}$	$10^{-1.1981}$
128	0.9823454399	7.954369226e−3	$10^{-2.7023}$	$10^{-1.4875}$
256	0.9903017451	4.193257348e−3	$10^{-3.0909}$	$10^{-1.7969}$
512	0.9946065355	2.256665408e−3	$10^{-3.4954}$	$10^{-2.1286}$
1024	0.9969885562	1.227048219e−3	$10^{-3.9180}$	$10^{-2.4835}$
2048	0.9983200415	6.698532817e−4	$10^{-4.3593}$	$10^{-2.8618}$
4096	0.9990662611	3.657104498e−4	$10^{-4.8193}$	$10^{-3.2625}$
8192	0.9994836866	1.992237068e−4	$10^{-5.2970}$	$10^{-3.6844}$
16 384	0.9997161525	1.081550890e−4	$10^{-5.7911}$	$10^{-4.1260}$
32 768	0.9998448719	5.847915319e−5	$10^{-6.3002}$	$10^{-4.5853}$
65 536	0.9999157029	3.148687468e−5	$10^{-6.8229}$	$10^{-5.0608}$

It would be better if higher even moments above kurtosis were correct, but we were unable to find a simple and computationally tractable method of solving for a higher order correction that could also correct the sixth moment. However, the behavior of the higher even moments should have a lesser effect on most simulations, as the sampling distributions of the higher moments have very high variance.

B. Conversion to Fixed-Point

The cubic table correction can be used to match the first four moments of the Gaussian distribution to very high precision, but only if the elements of the table are also stored with high precision. In principle, the tables could be stored in hardware in double-precision, but in practice this would be wildly inefficient, requiring n^2 double precision adders. For efficiencies' sake, the tables must be held in fixed-point, both to reduce storage requirements and to allow efficient addition.

A straightforward approach is to simply round each table element to the nearest fixed-point number, but this could distort the moments of the table: if a large majority of the elements happen to be rounded up to the next representable value then both the SD and kurtosis will become too large. The table-based multivariate RNG is particularly sensitive to the SD of each component table, as this directly affects the quality of the resulting correlation matrix, so a more effective rounding method is needed.

This section proposes a simple and direct approach to rounding, which gives good results while taking time linear in the number of table elements. The process starts with the naively rounded table, then visits each table element from largest to smallest, flipping the rounding choice whenever it will reduce the error in the SD. Note that the symmetry of the table must be preserved to keep the mean of the table at zero and avoid skewness.

Algorithm 1 provides pseudo-code for the process. The inputs to the algorithm are L, a symmetric table of k elements, and the target SD σ. The rounding of individual elements is

Algorithm 1 Round Table to Fixed-Point

$s \leftarrow 0, \quad A \leftarrow \mathbf{0}_k$
for $i = 1, \ldots, k$ **do**
  $(L[i], A[i]) \leftarrow \text{round}(L[i])$ {Closest is $L[i]$, alt. is $A[i]$}
  $s \leftarrow s + L[i]^2$ {Update sum of squares}
end for
for $i = 1, \ldots, k/2$ **do** {Loop over one half from big to small}
  $s' \leftarrow s - 2L[i]^2 + 2A[i]^2$ {Sum of sqr. if elt is flipped}
  if $|s'/k - \sigma| < |s/k - \sigma|$ **then** {More accurate?}
    $L[i] \leftarrow A[i]$
    $L[k - i + 1] \leftarrow -A[i]$ {Ensure symmetry}
    $s \leftarrow s'$ {Update sum of squares}
  end if
end for

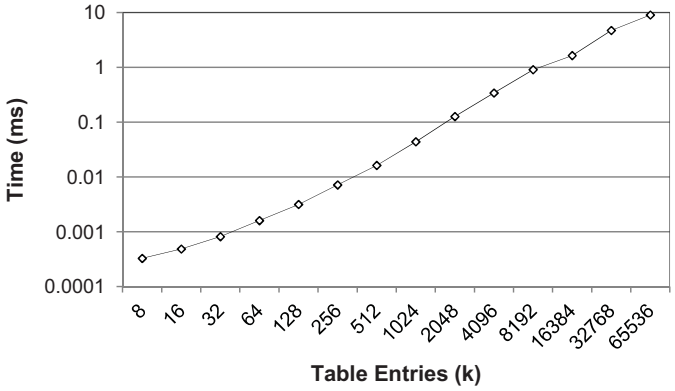

Fig. 9. Time to generate and correct one k-element table.

handled by the function round, which rounds elements to a pair: the first element is the closest representable value, and the second is the next closest value.

In the first loop, the algorithm rounds the table L in-place, while building up a table of alternates A. In the second loop, the algorithm examines the elements from largest to smallest magnitude. For each element, the choice between keeping L_i or swapping to A_i is examined: if swapping reduces the error in the SD then the element is changed (making sure to preserve symmetry), and the sum of squares is updated. By iterating from large to small elements, the algorithm has a chance to correct larger errors at the start, then polishes the SD with later smaller values.

Both the cubic correction and fixed-point correction times are $O(k)$, making the total time to generate tables $O(n^2k)$. There is little variation in execution time due to w, so Fig. 9 shows the change in LUT generation time for increasing k on a 3.4-GHz desktop PC. As expected, the time taken increases linearly with k, meaning that even very large tables are practical.

To provide an overall view of the time taken to change a matrix, Fig. 10 measures the time taken to generate and load a new matrix onto the FPGA for $k = 128$. It is clear that generating the tables is the dominant factor, so if necessary the table generation process could be parallelized,

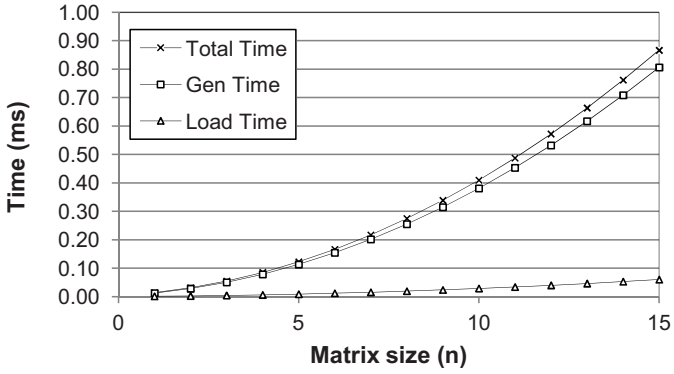

Fig. 10. Time taken to configure generator with a new correlation matrix, with break-down for table generation time and loading time.

as generated each table can be treated as an independent task. However, even with a nonparallel method, very large tables can still be generated and loaded in ten milliseconds or less.

VI. Statistical Evaluation

The use of small LUTs to approximate Gaussian distributions raises some important questions about the quality of the distribution that the generator actually produces. The three main questions are as follows.

1) How accurate are the moments of the fixed-point cubic-corrected Gaussian LUT for limited precision tables?
2) How close is the marginal distribution of each vector element to the Gaussian distribution?
3) Does the correlation structure of the generator match the original target correlation matrix?

This section investigates these questions, using analytical methods where possible and empirical methods if required.

The first question is whether the methods described in the previous section are actually effective for producing low-resolution tables that accurately match a given Gaussian distribution. This is tested using a table with two integer bits, and from zero to 16 fractional bits. This allows values in the range [0, 4), and so can accommodate a unit-SD Gaussian table for $k \leq 2^{14}$, as the largest sample in the table is $L[k] = \Phi^{-1}(k/(k+1))$, which after cubic correction is 3.847.

The effect of different numbers of fractional bits is tested by starting with an SD of one, then mapping Gaussian distributions with progressively smaller SDs into the same table. Each time the SD is reduced by half, it is equivalent to reducing the number of fractional bits by one, so this gives information about both the change in moment accuracy for different SDs, and the change in accuracy for reduced precision tables.

Fig. 11 shows the change in SD relative error as the target SD is reduced, using three different methods to produce a table with $k = 128$: none (7), linear (15), and cubic (17). The uncorrected table has an intrinsically inaccurate SD even before rounding, but both linear and cubic methods achieve a good relative error, degrading smoothly with decreasing number of fractional bits, so Algorithm 1 does a good job of preserving the target SD.

Fig. 12 shows the relative error in the kurtosis as the SD is varied. Now the limitations of the linear correction are clear, as

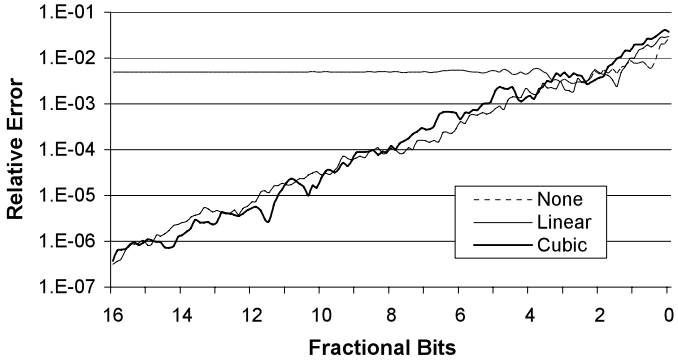

Fig. 11. Relative error of SD as number of fractional bits is reduced.

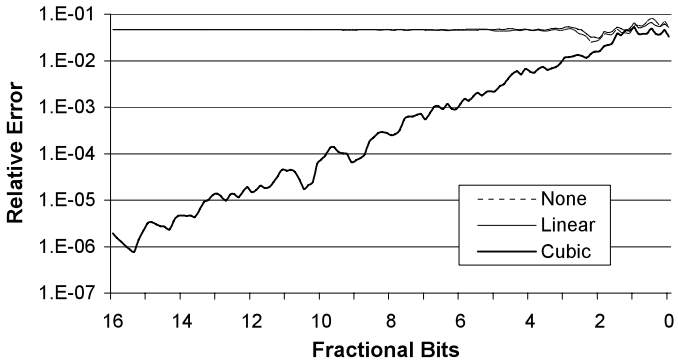

Fig. 12. Relative error of kurtosis as number of fractional bits is reduced.

it produces levels of error very similar to the uncorrected case. However, the cubic corrected table shows the same smooth increase in error as the number of effective fractional bits is reduced, allowing the relationship between table precision and the accuracy of the moments to be easily predicted.

The next question is whether the marginal distribution of the vector elements has the Gaussian distribution; the CLT guarantees that as more tables are accumulated (i.e., vector dimension increases), then the outputs will become ever closer to the true Gaussian distribution. However, the known theoretical bounds on convergence are extremely conservative, so it is necessary to determine what occurs in practice.

The marginal PDF of each vector element can be calculated through convolution: each table describes a PDF on a discrete range with k spikes of $1/k$ (assuming all table elements hold distinct values), so the PDF of the sum of two tables is determined by the convolution of the two table's PDFs. This convolution can be efficiently performed using a fast Fourier transform (FFT), so an exact FFT using the NTL arbitrary precision library [10] was developed, allowing the exact marginal PDF of each element to be determined analytically.

The tests used a table with $k = 128$ and a precision $w_t = 14$. More precision (i.e., $w_t > 14$) can only result in greater accuracy, but is not examined as the convolutions become extremely slow, as each extra bit of precision doubles the time taken to perform the convolutions. Vector sizes of $n = 1, \ldots, 16$ are considered, producing a unit SD marginal, with each input factor contributing equal weight. As before, tests are performed for the three different table correction methods.

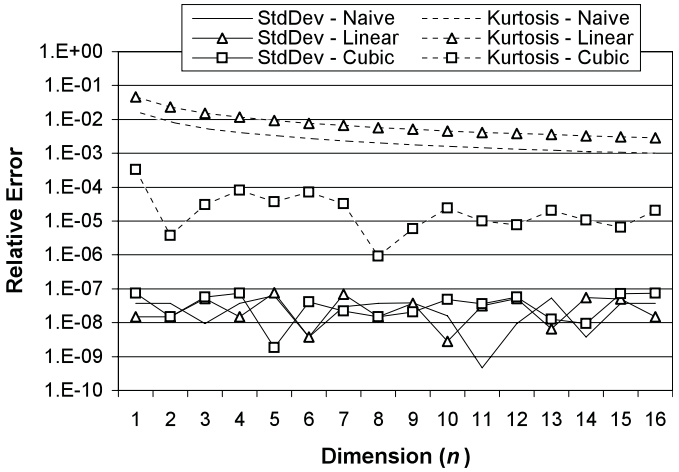

Fig. 13. Relative error of SD and kurtosis for differing vector dimension and table correction methods.

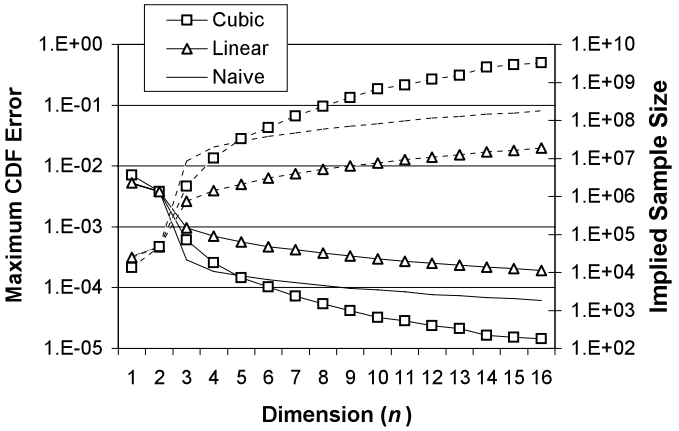

Fig. 14. Maximum CDF error and implied sample size before failure of Kolmogorov–Smirnov test for differing dimension and correction methods.

The exact SD and kurtosis can be extracted from the PDF of the marginal distribution, allowing the error to be calculated even for very high dimension generators. Fig. 13 shows the change in relative SD and kurtosis error as the vector dimension is increased. As before, the SD of the uncorrected method is poor, and remains poor as the dimension increases, while the linear and cubic corrected method maintain good accuracy independent of the dimension. The kurtosis of the linear and uncorrected methods starts off poorly, but gradually improves as the dimension increases, due to the CLT. The cubic method provides a much more accurate initial kurtosis, but then improves at a slower rate.

Another way of looking at the marginal distribution is to consider the worst CDF error. The exact CDF can be extracted from the PDF as a running sum, which defines the discrete CDF at each representable fixed-point value. The discrete CDF can then be compared with the target Gaussian CDF to find the worst discrepancy between the two. Fig. 14 shows the maximum CDF error as the dimension increases. The CLT predicts inverse quadratic convergence to the true Gaussian CDF as $n \to \infty$, but the observed convergence differs for each correction method. For very small n, the uncorrected method is

actually more accurate, but for $n > 3$ the cubic method quickly provides much better results, demonstrating the effectiveness of correcting the kurtosis.

The maximum CDF error also provides a conservative means of estimating the practical quality of the marginal PDF, through the Kolmogorov–Smirnov (KS) test. Usually the KS test is used to combine the worst error in the empirical CDF and the sample size to provide a p-value (significance level). However, this process can be inverted, using the known CDF error and a p-value of 0.5 to estimate the number of samples before KS test failure–this technique abuses the theory of the KS test, and only gives a rough guide to the order of sample-size that might cause failure; it does not predict a precise failure point. This predicted sample size is shown on the right axis of Fig. 14, but note that this is a pessimistic lower-bound on sample size.

The final question is whether the correlation structure of the generator matches the target correlation structure, or equivalently: does the covariance matrix of the generator match the target covariance matrix? In principle, the covariance matrix of the generator could be recovered simply by calculating the exact SD of each table in $\mathbf{G}$ after conversion to fixed-point. However, this would only hold if the tables were very close to Gaussian–in practice each individual table is a relatively poor approximation to the Gaussian distribution, so it is possible that the actual covariance matrix differs from that predicted by the marginal SDs of the tables.

One option for extracting the exact covariance matrix is Brute-force enumeration, but even for a moderate number of dimensions this is computationally infeasible, so empirical methods are required. Two metrics for empirical evaluation of the correlation matrix proposed in previous work can be used for the evaluation of multivariate Gaussian generators.

In [4], Fisher's Z transform is used to examine the hypothesis that each element of the empirical correlation matrix has the same correlation as the target matrix. This results in an $n \times n$ matrix of p-values–p-values should appear as uniformly distributed values between 0 and 1, with values very close to 0 or 1 suggesting failure. The matrix of p-values can then be reduced down to a single p-value using the Anderson–Darling method. The advantage of this method is that it has a direct statistical interpretation, and takes into account the number of samples examined. The disadvantage is that it cannot be used for comparison purposes, as it gives no absolute measure of quality or accuracy.

An alternative approach is used in [5], where the correlation matrix quality is measured using the mean squared error (MSE) between the empirical matrix and the target matrix. This metric has the advantage of simplicity, and provides an absolute metric which can be used to compare different generators. However, it has no statistical basis, and may assign a poor generator a good score–two generators might produce the same MSE, but one might spread the error over all matrix elements, while the other concentrates it in just one. From a statistical point of view, the former is much better than the latter, but the MSE is not able to distinguish between the two.

Fig. 15 shows the results of both approaches, using $n = 10$ and a randomly generated matrix (for easier comparison with the results in [5]), with sample sizes from 2^{10} vectors up to 2^{31}. The left axis shows the empirical MSE, with a steady decrease in MSE as the sample size is increased. Eventually it appears to stabilize at an error of about 10^{-9}, suggesting that the average correlation error is 3×10^{-4}.

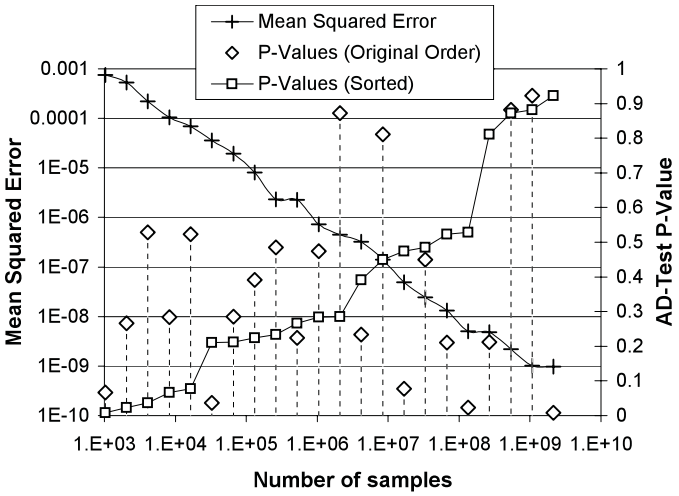

Fig. 15. Empirical measurement of correlation quality for a ten-element matrix as sample size is increased, showing mean square error of the empirical correlation matrix and p-values for Anderson–Darling test of empirical covariance matrix.

The points on vertical stalks show the p-values associated with the sample size at which they were calculated, so for easier interpretation the connected line shows the sorted p-values. If the p-values are truly uniformly distributed, they should form a roughly straight line when sorted. The results of this test do not show any clear evidence for the hypothesis that the empirical and target correlation matrices are different, but the final p-value of 0.99 for a sample size of 2^{31} is very close to significance. The fact that the MSE also levels out suggests that a critical sample size has been reached.

These tests suggest that inaccuracies in both the generator's correlation matrix and in its marginal PDFs become detectable at around 2^{31} samples. However, this does not mean that it cannot be used for long-running simulations using much larger numbers of vectors. These tests are designed specifically to be sensitive to flaws, and most Monte Carlo simulations will not be biased until a much larger sample size is reached.

VII. Comparison With Related Work

Including this paper, there are three main approaches to sampling from the multivariate Gaussian distribution in FPGAs.

1) DSP-based [4]: Use DSPs to perform the dot product of one column of $\mathbf{A}$ with r per cycle, requiring n cycles to form the entire output vector $\mathbf{x} = \mathbf{Ar}$.
2) Custom-logic [5]: The covariance matrix $\mathbf{A}$ is analyzed to create an architecture with optimized structure and precision, which can be placed-and-routed to give a circuit specialized on the matrix that can generate samples for every n cycles.
3) Table-based [This Paper]: Use lookup-tables and adders to create a general-purpose structure, which takes one

TABLE II
CHARACTERISTICS OF THREE ALTERNATIVE METHODS FOR GENERATING MULTIVARIATE GAUSSIAN SAMPLES IN FPGAS

	Part	Clock Rate	Cycles/ Vector	Config. Method	External RNG	LUTs	DSPs	MVec/ sec	KVec/ LUT/sec	Correlation MSE
DSP [4]	Virtex 5	550 MHz	n	Dedicated logic (1–10 ms)	Gaussian	717	10	55	76.7	1.0×10^{-4}
Custom [5]	Stratix 3	411 MHz	n	Place-and-route (10 mins+)	Gaussian	1250	0	41	32.9	2.5×10^{-4}
Table	Virtex 5	550 MHz	1	Bitstream manip (1–5 s)	none	3270	0	550	168.2	1.4×10^{-5}
				Direct loading (1–10 ms)		3302			166.5	

cycle to calculate $\mathbf{x} = \mathbf{A}\mathbf{r}$, and can support different matrices by modifying the table entries.

This section considers the alternative methods in more detail, and compares them to the method proposed here.

The earliest and simplest approach is the multiplier-based approach used in [4]. This uses a set of n DSP blocks (multiply-accumulate units), and splits the generation into two stages. In the first stage, a scalar Gaussian generator is used to generate the vector $\mathbf{r}$ of independent variates over n cycles, shifting each generated sample down a shift register. In the second stage, the vector $\mathbf{r}$ is retrieved in parallel from the shift register, then over the next n cycles each element x_i of the output is calculated using 3.

The implementation can be optimized to take advantage of dedicated accumulation chains, such as those found in the Virtex-4 DSP48, and the matrix-vector multiplication takes very few resources. The original paper focused on large matrices, up to $n = 512$, but the implementation becomes much simpler when operating in the range discussed in this paper, where $n \leq 64$. In particular, the n block-RAMs required in the original can be placed in LUT-based RAMs, and all accumulation can be routed through the internal DSP48 adders, removing the need for expensive pipelined adders. For the results given here, a modified implementation has been developed using these optimizations for smaller vector sizes, and for Virtex-5 rather than the original Virtex-4 primitives. The design is only limited by the DSP components, resulting in a clock rate of 550 MHz.

The approach taken in [5] is to build a custom circuit for each covariance matrix $\mathbf{A}$. This allows the circuit to be heavily optimized, taking advantage of word-length optimization to reduce resource usage, while maintaining the accuracy of the generator's correlation matrix. Multiple possible implementations are also produced, by using a library of building blocks to create multiple candidate generators, then constructing a Pareto frontier of correlation accuracy versus resource usage. In more recent work, the same approach is used to generate a circuit, which is able to select between a set of fixed correlation matrices for each generated vector, exploiting any similarities between them [11], but that solves a different problem to the single correlation matrix problem addressed here.

Generating a custom circuit for each covariance matrix provides many optimization possibilities, but also has the disadvantage that each new matrix incurs a full synthesis and place-and-route cycle. The authors reported a total time of 11 min to generate, synthesize, and place-and-route a generator with $n = 50$, but this only considers a single generator, which does not occupy very much of the device. In practice, the random generator will be only one component of a larger simulation engine, which will then be replicated within the FPGA to use as many resources as possible, so these place-and-route times are likely to be a very optimistic lower bound. However, in applications where correlation matrices do not change frequently this compilation overhead can be tolerated.

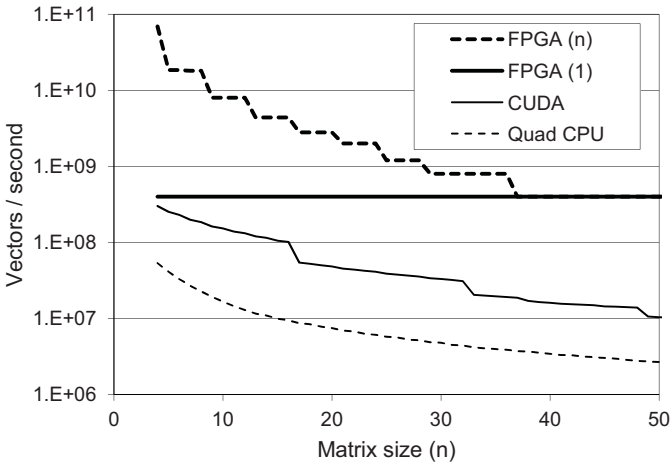

Fig. 16. Performance of different platforms for vector generation.

The two existing methods are compared with the table-based method in Table II, comparing both the performance and resource utilization for generators of ten-element vectors (as this was the size for the results from [5]). When comparing performance in generated vectors/second, the table-based method operates at a similar clock-rate to both existing methods, but because it generates a new vector for every cycle it actually offers ten times the performance. When resources are considered, the table-based method requires 2.6 times the resources of the customized generator, but due to the increased performance still offers 5 times the performance per resource.

The table-based method is also the only stand-alone method, as the uniform random number is included in the resource cost. The other methods require an external scalar Gaussian RNG, which increases the practical resource cost, and will require either DSPs, block-RAMs, or both.

Fig. 16 provides a performance comparison between a Virtex-5 FPGA, a Quad-core CPU, and a GPU, for $n = 4, \ldots, 50$. Because an MVGRNG does not do anything by itself, the test actually measures how many vectors can be generated and accumulated per second, to ensure that outputs are not optimized out by the compilers. All measurements are measured over at least ten seconds to ensure accurate times.

The FPGA used is an xc5vlx330, clocked at 400 MHz (a deliberately conservative target), using $w_t = 24$. This large output width was chosen in order to provide a fair comparison to the single-precision format of the CPU and GPU, but it should be noted that in many Monte Carlo simulations it is possible to significantly reduce the precision without impairing accuracy.

"FPGA(1)" shows the performance of a single generator instance, while "FPGA(n)" shows the performance of an entire FPGA when enough instances are replicated to fill the device to 90%. The GPU results are derived from a CUDA implementation, which is optimized to use memory coalescing, specialized functions, and shared memory, executed on an NVidia Tesla C1060 running at 1.25 GHz. The implementation uses a proprietary routine rather than the CUDA BLAS library, as it allowed merging of random number generation and matrix multiplication, and persistent storage of the matrix in shared memory, reducing memory traffic, and increasing performance when compared to using the CUDA BLAS. The CPU results are chosen from the fastest of a C++ implementation templatized on vector size and the MVGRNG supplied by the AMD SIMD-optimized ACML library (as for small vectors the plain C++ is faster), executed on all four cores of an AMD Phenom 9650 2.2-GHz quad-core processor. Note that the relatively small difference between CPU and GPU speeds is indicative of a CPU being used well, rather than the GPU being used poorly.

Devoting the whole FPGA to MVGRNG provides at least ten times the performance of the GPU, and one hundred times that of the CPU–even a single generator instance provides higher performance for vectors of size four and above.

VIII. Conclusion

This paper presented a method for multivariate Gaussian random number generation in FPGAs, which decomposes the generation into a series of table-lookups and additions. This method is ideal for use in numerical Monte Carlo simulations, as it only uses LUTs and FFs, and so all DSP and block-RAM resources can be used in the simulation core that the generator is driving.

When compared to previous methods for MVGRNG, the table-based method offers greatly improved performance, generating a new random vector for every cycle, rather than generating vectors serially over multiple cycles. When generating length ten vectors, the table-based method provided ten times the performance of the fastest DSP-based method. The performance per resource was also five times that of a generator constructed and compiled for a specific correlation matrix, while still supporting loading of arbitrary correlation matrices.

References

[1] D. B. Thomas and W. Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation," in *Proc. ACM Symp. FPGAs Custom Comput. Mach.*, 2008, pp. 229–238.

[2] N. Woods and T. VanCourt, "FPGA acceleration of Quasi-Monte Carlo in finance," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2008, pp. 335–340.

[3] A. Kaganov, P. Chow, and A. Lakhany, "FPGA acceleration of Monte-Carlo based credit derivative pricing," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2008, pp. 329–334.

[4] D. B. Thomas and W. Luk, "Multivariate Gaussian random number generation targeting reconfigurable hardware," *ACM Trans. Recon. Technol. Syst.*, vol. 1, no. 12, pp. 22–26, 2008.

[5] C. Saiprasert, C.-S. Bouganis, and G. A. Constantinides, "Word-length optimization and error analysis of a multivariate gaussian random number generator," in *Proc. Int. Conf. Appl. Reconf. Comput.*, 2009, pp. 231–242.

[6] D. B. Thomas and W. Luk, "An FPGA-specific algorithm for direct generation of multi-variate gaussian random numbers," in *Proc. IEEE Int. Conf. Appl. Specif. Syst., Archit. Process.*, Jun. 2010, pp. 208–215.

[7] N. J. Higham, "Computing the nearest correlation matrix—a problem from finance," *IMA J. Num. Anal.*, vol. 22, pp. 329–343, Oct. 2002.

[8] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," *J. Very Large Scale Integr. Signal Process.*, vol. 47, no. 1, pp. 77–92, 2007.

[9] *Maxima, a Computer Algebra System*. (2009) [Online]. Available: http://maxima.sourceforge.net/

[10] V. Shoup. (2005). "NTL: A library for doing number theory," [Online]. Available: http://www.shoup.net/ntl/

[11] C. Saiprasert, C.-S. Bouganis, and G. Constantinides, "Mapping multiple multivariate gaussian random number generators on an FPGA," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2010, pp. 89–94.

David B. Thomas (M'06) received the M.Eng. and Ph.D. degrees in computer science from Imperial College London, London, U.K., in 2001 and 2006, respectively.

He has been a Lecturer with the Electrical and Electronic Engineering Department, Imperial College London, since 2010. His current research interests include hardware-accelerated cluster computing, FPGA-based Monte Carlo simulation, algorithms and architectures for random number generation, and financial computing.

Wayne Luk (F'09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K.

He is currently a Professor of computer engineering with Imperial College London, U.K., and a Visiting Professor with Stanford University, Stanford, CA, and Queens University Belfast, Belfast, U.K. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.