A Hardware Architecture for Direct Generation of Multi-Variate Gaussian Random Numbers

David B. Thomas and Wayne Luk
{dt10,wl}@doc.ic.ac.uk
Imperial College London

***Abstract*—The multi-variate Gaussian distribution is used to model random processes with distinct pair-wise correlations, such as stock prices that tend to rise and fall together. Multi-variate 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 ROMs and adders, and so can be implemented using just logic resources (LUTs and FFs), saving DSP and block-RAM resources for the numerical simulation that the multi-variate generator is driving. The new method provides a ten times increase in raw performance over the fastest existing FPGA generation method, and also provides a five times improvement in performance per resource over the most efficient existing method. Using this method a single 400MHz Virtex-5 FPGA can generate vectors ten times faster than an optimised CUDA implementation on a 1.2GHz GPU, and a hundred times faster than SIMD optimised software on a quad core 2.2GHz CPU.**

I. Introduction

The multi-variate 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 behaviour of such processes, Multi-Variate 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 estimate how much money the institution might lose over the next day.

Such long-running simulations are an ideal target for FPGA acceleration, as they offer abundant parallelism, and are computationally bound; however they are reliant on a fast, resource-efficient, and accurate source of random samples from the multi-variate Gaussian distribution. This paper improves on previous methods [1], [2] for FPGA-based MVGRNG, by developing a method that provides a large increase in performance, while limiting resource usage to standard logic elements (LUTs and FFs). Our key contributions are:

The support of UK Engineering and Physical Sciences Research Council (Grant references EP/D062322/1 and EP/C549481/1), Alpha Data and Xilinx is gratefully acknowledged.

- An algorithm for generating samples from the multi-variate Gaussian distribution using only uniform random bits, table-lookup, and addition.
- A hardware architecture for implementing an MVGRNG using only LUTs and FFs, which allows a regular densely-packed placement strategy, and provides operating speeds of 550MHz in Virtex-5 FPGAs.
- Correction methods for producing correct statistical properties even when using small fixed-point tables.
- An evaluation of the statistical properties of the table-based MVGRNG, demonstrating that the algorithm and correction methods produces high quality random vectors.
- A comparison with two existing FPGA generation methods, showing more than 10 times the performance of the fastest method, and five times the performance per resource of the most efficient method.
- A comparison of FPGA generation performance with GPU and CPU implementations, showing the FPGA can provide ten times the performance of an optimised GPU generator, and a hundred times that of a quad-core SIMD-optimised CPU generator.

A sketch of the idea of using tables instead of multipliers has previously been presented in a short paper [3] - this paper provides a much more detailed analysis, demonstrating how the hardware architecture can be used in practice, and providing a detailed analysis of the generator quality and performance.

II. Multi-variate Gaussian Random Numbers

Generation of the uni-variate Gaussian distribution with a specific mean, μ, and variance, σ^2, is achieved by first generating a standard Gaussian variate r with mean zero and variance one, then applying a linear transformation:

$$x = \sigma r + \mu, \quad \text{where} \quad r \sim N(0,1) \tag{1}$$

Note that the standard Gaussian variate is multiplied by the Standard Deviation (SD) σ, rather than the variance σ^2.

Generation of a multi-variate Gaussian distribution is very similar, except each output sample is now a length n vector $\mathbf{x}$. The mean and variance also increase in dimension, so the mean is a length n vector of $\mathbf{m}$, and the variance becomes an $n \times n$ covariance matrix $\mathbf{\Sigma}$. 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 uni-variate 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 $\mathbf{A}\mathbf{A}^T = \mathbf{\Sigma}$. However, in the multi-variate case there is considerable freedom in the selection of $\mathbf{A}$, as there are many ways of decomposing $\mathbf{\Sigma}$.

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 non-zero and must be stored, so only $n(n+1)/2$ multiplications are required.

An alternative method is to use the Singular Value Decomposition (SVD) algorithm. This decomposes the matrix into an orthonormal matrix $\mathbf{U}$ and a diagonal matrix $\mathbf{S}$, such that $\mathbf{\Sigma} = \mathbf{U}\mathbf{S}\mathbf{U}^T$. 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 non-zero, 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 practise, so we focus 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):

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

If we have no advance knowledge about the covariance matrix and use the SVD decomposition, 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 RAMs), so we would like to eliminate them if possible.

The method we propose in this paper is that, instead of generating independent Gaussian samples then scaling them to the desired SD (Standard Deviation) with n^2 multiplications, we will generate uniform samples and convert them directly to Gaussian samples with the appropriate SD via n^2 table-lookups. Each table contains a pre-calculated discretised Gaussian distribution with the correct SD, so the only operations required are table-lookups and additions.

In the following text we frequently refer to tables, which in this context means an array of read-only elements (a ROM) which will be implemented in the FPGA using LUTs or block RAM. Unless otherwise specified, each table contains k elements, and is indexed using the syntax $L[i]$ to access table elements $L[1]..L[k]$. 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. $L_{2,3}[4]$. We will also treat tables interchangeably 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:

$$x = L[\ \lceil uk \rceil\ ], \quad \text{where} \quad u \sim U(0,1) \tag{4}$$

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 $\mathbf{r}$, we can use 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:

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

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 CDF inversion. To generate a table L with SD σ, we choose table elements:

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

where $\Phi^{-1}(.)$ is the Gaussian inverse CDF. We can now fully specify the table $\mathbf{G}$ given a target matrix $\mathbf{A}$:

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

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

IV. Hardware Architecture

The central idea of 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. In this paper we focus 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 we make are:

- **Logic resources only**: tables are implemented using LUTs, so the only resources used are LUTs and FFs (no DSPs or block-RAMs).
- **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 every n cycles.

- **Maximum clock rate**: the generator operates at the maximum realistic clock-rate for the target FPGA. For the Virtex-5 we choose 550MHz, 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.
- **Regular architecture**: simulations typically consume almost all resources in the FPGA (due to replication of simulation cores), and a regular, explicitly placed, highly routeable, generator allows fast place-and-route while still achieving high overall clock-rate.
- **No matrix specialisation**: the generator is not optimised for a specific correlation matrix, and should support any correlation structure without structural modification.
- **Configuration through bit-stream manipulation**: previous approaches have relied on either dedicated circuitry to load new correlation matrices, or used a full place-and-route cycle to customise the design. Here we modify the matrix by modification of LUT values in the bit-stream.

The table-based generator maps naturally into a regular pipelined structure, shown in Figure 1. At the top is a random bit source, which generates a new vector $\mathbf{u}$ every cycle. The elements of vector $\mathbf{u}$ are broadcast vertically through the cells, and used to select one element from each table. The selected elements are then accumulated horizontally from left to right, with one new vector output per cycle.

A useful optimisation for increasing the effective number of elements per table is to take advantage of the symmetry of the tables. Because the tables have a mean of zero, we have the property that $L[i] = -L[k-i+1]$, so it is only necessary to store the elements $L[1]..L[k/2]$. 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 optimisation 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.

To calculate resource usage, we can assume that: table elements have width w_t; accumulators have a width $w_a = w_t + \lceil \log_2 n \rceil$; each LUT can implement a $1 \times k$ bit LUT (using the previous optimisation); and the uniform generator is implemented using a LUT-Optimised RNG [4]. The resource usage of the generator can then be broken down as: Uniform RNG, $n \log_2 k$; tables, $n^2 w_t$; and accumulators, $n^2 w_a = n^2(w_t + \lceil \log_2 n \rceil)$. Total resource utilisation is then:

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

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.

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 550MHz will generate 55GB/s of data, so it would be very difficult to dedicate an entire FPGA to multi-variate generation and ship the vectors elsewhere – so the practical maximum is around $n = 64$.

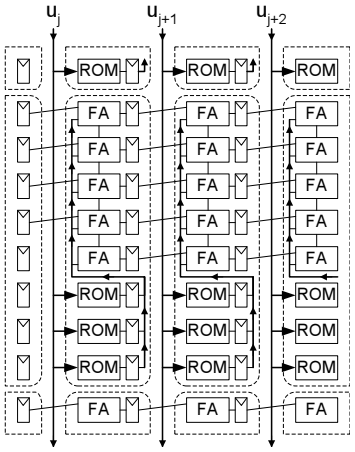

Fig. 1. Practical dense placement for hardware realisation.

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 we adopt the simple placement strategy shown in Figure 1, where the accumulator is simply stacked on top of the table.

This architecture has been described in VHDL, using Virtex-5 primitives with RPM (Relationally Placed Macro) constraints. When mapped into hardware, the resource utilisation exactly matches the predictions of Equation 9. For all $n \leq 16$ and $w_t \leq 16$ we found this strategy provides 550MHz operation in an xc5vlx330 device (post place-and-route timing).

As n grows larger, the fanout of the uniform generator lines begins to reduce 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 specialised devices such as DSP columns may intrude.

Fortunately the regular data-flow in the architecture, and 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 paths 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.

We propose that generators are scaled 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 Figure 1, 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. Figure 2

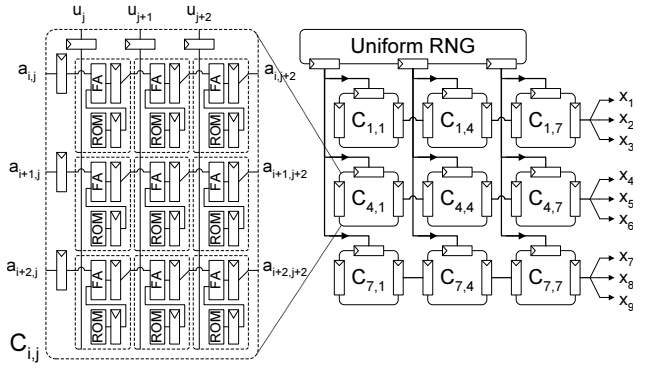

Fig. 2. Two-level decomposition of generator for large n.

shows this architecture for $n = 9$ with 3x3 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 connections between sub-generators are direct FF to FF paths, providing more slack to the place-and-route tools. Finally, the fanout of the uniform generator is reduced to the number of sub-generators, rather than the number of elements.

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

In the largest devices the limiting factor is the 550MHz global clock limits, not the generator itself; for smaller devices which support 710MHz global clocks, the generator logic becomes the limit. In the xc5vlx85 we observed reported clock rates over 600MHz when using a 550MHz target constraint. However, 550MHz 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 550MHz.

The circuits described here contain no support for direct modification or loading of new covariance matrices. Instead, we use direct 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.

If bit-stream modification is unacceptable then 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 almost no extra logic, but has the disadvantage of reducing the effective table width to $k = 64$. Another option, appropriate for Altera architectures, is to configure the table LUTs as RAMs, and to move data in via the accumulators. However, a small amount of extra control logic and signals will be needed to control the adders and RAM write-ports during this process.

V. Practical Considerations

The generation algorithm described in Section III is asymptotically correct, but the architecture described in Section IV introduces limits due to the reality of hardware. The first problem is that in practise tables must be relatively small ($k \leq 2^{16}$), which affects the accuracy of the Gaussian approximation provided by Equation 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. In this section we show how to control the effects of these practical limitations by modifying the table contents - no changes to the hardware architecture are required.

Equation 7 converges on the true Gaussian distribution as the table size (k) increases, but for table sizes that can be efficiently implemented in FGPAs the approximation is poor. In particular, the even central moments (SD and kurtosis) of the distribution can be very poor, leading to distortion of the moments of the output vectors. It is critical to have the correct SD, as this directly affects the marginal SD of the output vector elements. A simple linear transformation can transform the original table L into a new table L' with the correct SD:

$$L[i]' = c_1 L[i], \qquad \text{where } c_1 = \sigma \sqrt{\frac{k}{\sum_{i=1}^{k} L[i]^2}} \tag{10}$$

Correcting the kurtosis (fourth moment) is also 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. To correct both the SD and kurtosis we can apply a cubic polynomial stretch to the uncorrected entries, using only odd powers to preserve symmetry:

$$L'[i] = c_1 L[i] + c_3 \sigma^{-2} L[i]^3 \tag{11}$$

Finding the coefficients c_1 and c_3 requires the roots of a pair of multinomial equations, and requires the use of a numerical solver. For convenience, we list the coefficients for tables with length $k = 2^i$ in Table I. These coefficients all provide SD and kurtosis relative errors of less than 10^{-13}; the table also shows relative errors for the sixth and eighth moments, showing that as expected the error decreases with larger table sizes (due to the Central Limit Theorem).

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.

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	$10^{-5.2970}$	$10^{-3.6844}$
16384	0.9997161525	1.081550890e-4	$10^{-5.7911}$	$10^{-4.1260}$
32768	0.9998448719	5.847915319e-5	$10^{-6.3002}$	$10^{-4.5853}$
65536	0.9999157029	3.148687468e-5	$10^{-6.8229}$	$10^{-5.0608}$

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

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 the majority of the elements are rounded up in magnitude to the next representable value, then both the SD and kurtosis will become too large. For our purposes we are particularly sensitive to the SD, as this directly affects the quality of the resulting correlation matrix, so we need a more effective rounding method.

We propose 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 we *must* preserve the symmetry of the table 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 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.

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:

1) How accurate are the moments of the fixed-point cubic-corrected Gaussian LUT for limited precision tables?

Algorithm 1 Round table to fixed-point.

$s \leftarrow 0, \quad A \leftarrow \mathbf{0}_k$
for $i = 1..k$ **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 $i = 1..k/2$ **do** {Loop over one half from big to small}
  $s' \leftarrow s - 2L[i]^2 + 2A[i]^2$ {Sum of sqr. if we flip elt. i}
  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

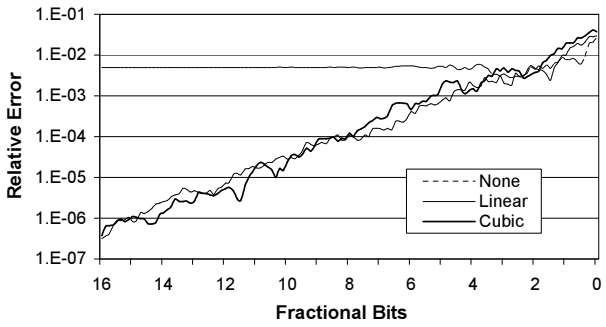

Fig. 3. Relative error of SD as number of fractional bits is reduced.

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?

In this section we investigate 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. To test this, we consider a table with 2 integer bits, and 16 fractional bits. This allows values in the range [0,3) to be represented, and so can accommodate a unit-SD Gaussian table for $k \leq 2^{14}$.

To test the effect of different numbers of fractional bits, we start with an SD of one, then map 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 us information both about the change in moment accuracy for different SDs, but also about the change in accuracy for reduced precision tables.

Figure 3 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 (Eqn. 7), linear (Eqn. 10), and cubic (Eqn. 11). 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 can

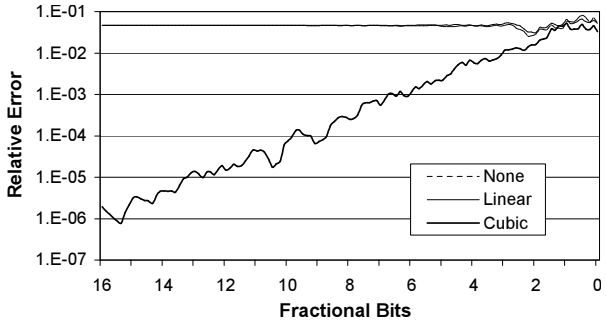

Fig. 4. Relative error of kurtosis as number of fractional bits is reduced.

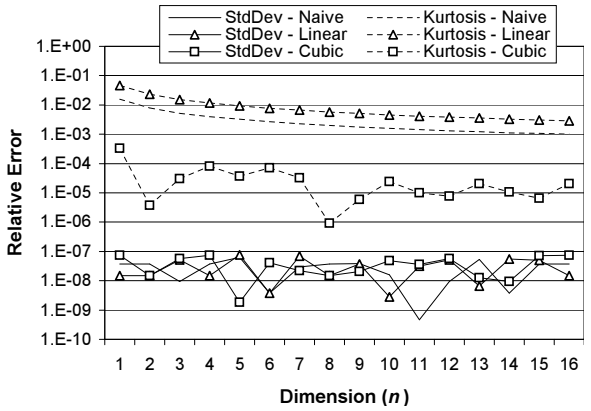

Fig. 5. Relative error of SD and kurtosis for differing vector dimension and table correction methods.

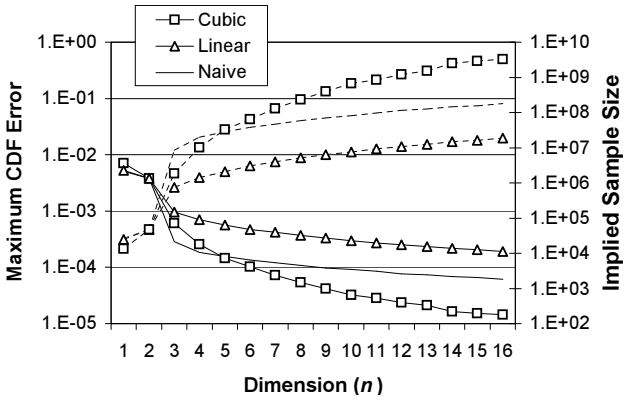

Fig. 6. Maximum CDF error and implied maximum sample size before failure of Kolmogorov-Smirnov test for differing vector dimension and table correction methods.

preserve the target SD.

Figure 4 shows the relative error in the kurtosis as the SD is varied. Now the limitations of the linear correction are clear, as 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 are 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 Central Limit Theorem (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.

We can calculate the marginal PDF of each vector element 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). We developed an exact FFT using the NTL arbitrary precision library [5], allowing the exact marginal PDF of each element to be determined analytically.

For testing purposes we use a table with $k = 128$, and a precision $w_t = 14$. [1] Vector sizes of $n = 1..16$ 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.

From the marginal PDF we can extract the SD and kurtosis of the marginal distribution. Figure 5 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. We can extract the exact CDF from the PDF as a running sum, which defines the discrete CDF at each representable fixed-point value. The discrete CDF can be compared with the target Gaussian CDF to find the worst discrepancy between the two. Figure 6 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, by inverting the Kolmogorov Smirnov (KS) test. This predicted sample size is shown on the right axis of Figure 6, but note that this is a very conservative lower-bound on sample size.

The final question is whether the covariance matrix of the generator matches the target covariance matrix. In principle we can recover the covariance matrix of the generator simply by

[1] More precision can only result in greater accuracy, but is not examined as the convolutions become extremely slow, and each extra bit of precision doubles the time taken to perform the convolutions.

	Part	Clock rate	Cycles/ Vector	Config. Method	External RNG	LUTs	DSPs	MVec/ sec	KVec/ LUT/sec	MSE
DSP [1]	Virtex 5	550MHz	n	Dedicated logic (1-10 ms)	Gaussian	717	10	55	76.7	1.0×10^{-4}
Custom [2]	Stratix 3	411MHz	n	Place-and-route (10 mins+)	Gaussian	1250	0	41	32.9	2.5×10^{-4}
Table	Virtex 5	550MHz	1	Bitstream manip (1-5 s)	none	3270	0	550	168.2	1.4×10^{-5}

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

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 we must turn to empirical methods. We examined two metrics for empirical evaluation of the correlation matrix proposed in previous work for the evaluation of multi-variate Gaussian generators.

In [1] 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 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 [2], 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.

Figure 7 shows the results of both approaches, using $n = 10$ and a randomly generated matrix (for easier comparison with the results in [2]), 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 stabilise at an error of about 10^{-9}, suggesting that the average correlation error is 3×10^{-4}.

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

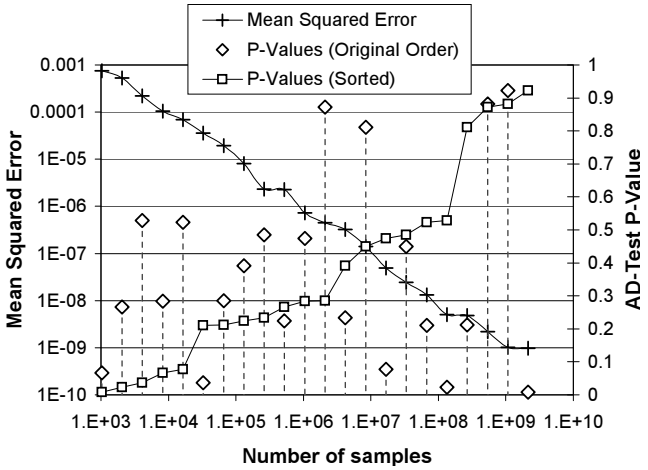

Fig. 7. Empirical results for 10-element matrix as sample size is increased, showing mean square error of the empirical correlation matrix, and p-values for Anderson-Darling test of empirical covariance matrix.

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.

VII. COMPARISON WITH RELATED WORK

There are two main existing approaches to the problem of sampling from the Multi-Variate Gaussian distribution. The earliest and simplest approach is the multiplier based approach used in [1]. 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 of independent variates $\mathbf{r}$ over n cycles, shifting each generated samples 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 Equation 3.

The implementation can be optimised 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. A new implementation has been developed using these optimisations

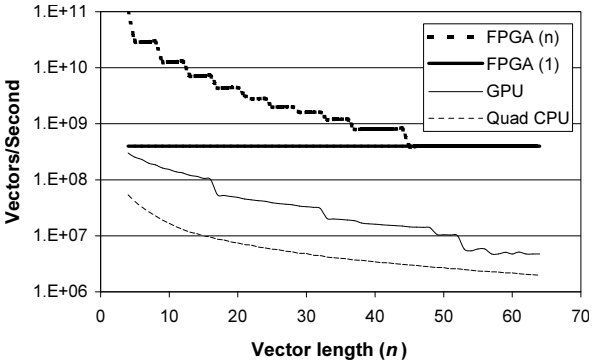

Fig. 8. Performance of different platforms for vector generation.

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 550MHz.

The approach taken in [2] is to build a custom circuit for each covariance matrix $\mathbf{A}$. This allows the circuit to be heavily optimised, taking advantage of word-length optimisation 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.

Generating a custom circuit for each covariance matrix provides many optimisation possibilities, but also has the disadvantage that each new matrix incurs a full synthesis and place-and-route cycle. The authors report a total time of 11 minutes to generate, synthesise, 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.

The two existing methods are compared with the table-based method in Table II, comparing both the performance and resource utilisation for generators of 10-element vectors. In terms of raw performance, the table-based method operates at a similar clock-rate to both existing methods, but because it generates a new vector every cycle it offers ten times the performance. When resources are considered, the table-based method requires 2.6 times the resources of the customised 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.

Figure 8 provides a performance comparison between an Virtex-5 FPGA, a Quad-core CPU, and a GPU, for $n = 4..64$. 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 optimised 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 400MHz (a deliberately conservative target), using $w_t = 14$. "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 optimised to use memory coalescing, specialised functions, and shared memory, executed on an NVidia Tesla C1060 running at 1.25GHz. The CPU results are chosen from the fastest of a C++ implementation templatised on vector size, and the MVGRNG supplied by the AMD SIMD-optimised ACML library, executed on all four cores of an AMD Phenom 9650 2.2GHz quad-core processor. [2]

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

VIII. Conclusion

This paper has presented a new method for multi-variate 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 every cycle, rather than generating vectors serially over multiple cycles. When generating length ten vectors, the table-based method provides ten times the performance of the fastest DSP-based method. The performance per resource is 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, "Multivariate Gaussian random number generation targeting reconfigurable hardware," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 12, 2008.

[2] 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. Applied Reconfigurable Computing*, 2009, pp. 231–242.

[3] D. B. Thomas and W. Luk, "Using FPGA resources for direct generation of multivariate gaussian random numbers," in *Proc. Int. Conf. on Field-Programmable Technology*, 2009, pp. 344–347.

[4] ——, "High quality uniform random number generation using LUT optimised state-transition matrices," *Journal of VLSI Signal Processing*, vol. 47, no. 1, pp. 77–92, 2007.

[5] V. Shoup, "NTL: A library for doing number theory," http://www.shoup.net/ntl/.

[2]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.