Multivariate Gaussian Random Number Generation Targeting Reconfigurable Hardware

DAVID B. THOMAS and WAYNE LUK
Imperial College London

The multivariate Gaussian distribution is often used to model correlations between stochastic time-series, and can be used to explore the effect of these correlations across N time-series in Monte-Carlo simulations. However, generating random correlated vectors is an $O(N^2)$ process, and quickly becomes a computational bottleneck in software simulations. This article presents an efficient method for generating vectors in parallel hardware, using N parallel pipelined components to generate a new vector every N cycles. This method maps well to the embedded block RAMs and multipliers in contemporary FPGAs, particularly as extensive testing shows that the limited bit-width arithmetic does not reduce the statistical quality of the generated vectors. An implementation of the architecture in the Virtex-4 architecture achieves a 500MHz clock-rate, and can support vector lengths up to 512 in the largest devices. The combination of a high clock-rate and parallelism provides a significant performance advantage over conventional processors, with an xc4vsx55 device at 500MHz providing a 200 times speedup over an Opteron 2.6GHz using an AMD optimised BLAS package. In a case study in Delta-Gamma Value-at Risk, an RC2000 accelerator card using an xc4vsx55 at 400MHz is 26 times faster than a quad Opteron 2.6GHz SMP.

Categories and Subject Descriptors: B.2.1 [**Arithmetic And Logic Structures**]: Design Styles

General Terms: Algorithms, Design, Economics

Additional Key Words and Phrases: Random numbers, multivariate Gaussian distribution, FPGA

ACM Reference Format:
Thomas, D. B. and Luk, W. 2008. Multivariate Gaussian random number generation targeting reconfigurable hardware. ACM Trans. Reconfig. Techn. Syst. 1, 2, Article 12 (June 2008), 29 pages. DOI = 10.1145/1371579.1371584. http://doi.acm.org/10.1145/1371579.1371584.

The support of UK Engineering and Physical Sciences Research Council (Grant reference EP/D062322/1 and EP/C540481/1) is gratefully acknowledged.
Author's address: D. B. Thomas; email: Dt10@doc.ic.ac.uk.

1. INTRODUCTION

An important element of many simulations is the multivariate Gaussian distribution, which captures correlations between different random factors. The multivariate Gaussian distribution can be used to model, for instance, the correlation between changes in different financial market indices, or the correlation between temperature and demand for ice cream. The capability of capturing complex correlations between many random factors, such as those within a complex portfolio containing tens or hundreds of assets, is a key benefit.

Generating random samples from a multivariate Gaussian distribution is a computationally demanding process, since it is based on matrix-vector multiplication to capture the correlations. The complexity of this process grows quadratically with vector size. Large clusters, as seen in large compute farms that banks use to calculate overnight Value-at-Risk, have been used to address this complexity. This article offers an alternative solution: adopting reconfigurable hardware for accelerating generation of multivariate Gaussian random vectors. This solution enables complete simulations to be built using reconfigurable hardware for many applications.

To describe our approach, this article presents:

—An analysis of hardware performance for Gaussian vector generation. This establishes limits on the size of vectors that can be generated in hardware in terms of available multiplier and RAM resources, and examines methods for storing coefficients in fixed-point.

—An abstract architecture for implementing vector generation in FPGAs, designed to serially generate vectors of size N over N cycles. In the Virtex-4 xc4vsx55 this architecture can operate at 500MHz, and provides performance over 200 times that of a single 2.6GHz Opteron.

—An architecture for implementing a Delta-Gamma Value-at-Risk simulation, using an xc4vsx55 part on the RC2000 platform, demonstrating a 26 times speed-up over parallelised software on a quad Opteron 2.6GHz, reducing simulation time from 30 to 1.1 seconds when simulating a 448 asset portfolio.

—An examination of the statistical quality of generated vectors, showing that the Virtex-4 fixed-point vector generator provides both high quality marginal Gaussian distributions, and imposes the correct correlation structure on the generated vectors.

2. ALGORITHM AND ANALYSIS

The univariate Gaussian distribution $X \sim \mathrm{Norm}(\mu, \sigma^2)$ is described through its Probability Density Function (PDF):

$$P(X = x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right). \tag{1}$$

Extended to the multivariate case, the distribution $X_N \sim \text{Norm}(\mathbf{m}, \mathbf{S})$ describes the PDF of a vector of length N:

$$P(X_N = \mathbf{x}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T\mathbf{S}^{-1}(\mathbf{x}-\mathbf{m})\right)}{(2\pi)^{N/2}\sqrt{\det(\mathbf{S})}}. \tag{2}$$

The vector $\mathbf{m}$ contains the mean of each component in the vector, that is, $E(X_N)$. The matrix $\mathbf{S}$ describes the covariance matrix between components. The diagonal elements $\mathbf{S}_{i,i}$ describe the marginal variances of the vector components (the variance of the component when treated as a univariate PDF), while the off-diagonal elements describe the degree of correlation between components. Large values of $\mathbf{S}_{i,j}$ indicate high levels of correlation, so if component i increases then it is likely that component j will also increase, while negative values make it likely that if one decreases the other will increase (and vice versa). For example, one might expect the correlation between the stock returns of Microsoft and Oracle to be significant as they are in similar markets, but the correlation of Microsoft and British Petroleum stock returns would probably be lower.

2.1 Generating Multivariate Gaussian Samples

To generate random samples from X_N it is necessary to form a vector $\mathbf{r}$ of independent univariate Gaussian samples from some infinite source $r_1, r_2, \ldots$. The desired correlation structure is then applied to $\mathbf{r}$ by multiplication with a lower triangular matrix $\mathbf{A}$, where $\mathbf{S} = \mathbf{A}\mathbf{A}^T$. The means of the components are then adjusted by adding the vector $\mathbf{m}$. Thus the generation of the k-th vector $\mathbf{x}_k$ is calculated as follows [Barr and Slezak 1972]:

$$\mathbf{r}_k = (r_{kN+1}, r_{kN+2}, \ldots, r_{kN+N})^T \tag{3}$$

$$\mathbf{x}_k = \mathbf{A}\mathbf{r}_k + \mathbf{m}. \tag{4}$$

If Equation (4) is expanded the structure of the computation becomes clearer:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} a_{1,1} & 0 & \ldots & 0 \\ a_{2,1} & a_{2,2} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \ldots & a_{N,N} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} + \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_N \end{bmatrix}, \tag{5}$$

which can be grouped into independent equations:

$$x_1 = a_{1,1}r_1 + m_1 \tag{6}$$

$$x_2 = a_{2,1}r_1 + a_{2,2}r_2 + m_2 \tag{7}$$

$$x_3 = a_{3,1}r_1 + a_{3,2}r_2 + a_{3,3}r_3 + m_3 \tag{8}$$

$$x_N = a_{N,1}r_1 + a_{N,2}r_2 + a_{N,3}r_3 + \ldots + a_{N,N}r_N + m_N. \tag{9}$$

Table I. FPGA Resource Usage for Parallel and Serial Vector Generation

	1/cycle	N^{-1} / cycle
Multiplies/cycle	$N(N+1)/2$	$(N+1)/2$
Total coefficients	$N(N+3)/2$	$N(N+3)/2$
Coefficients/cycle	$N(N+1)/2$	$(N+1)/2$
Adds/cycle	$N(N+1)/2$	$(N+1)/2$
Registers (bits)	$(w_c+w_s)N(N+1)/2$	$w_s(N+1)/2$
Block RAMs (storage)	-	$w_cN(N+1)/2/l_r/w_r$
Block RAMs (bandwidth)	-	$w_c(N+1)/2/p_r/w_r$

We will now explore the minimal cost of generating Gaussian samples by looking at the minimum number of resources needed. Four types of resources are considered: multipliers, adders, coefficient storage, and coefficient bandwidth. We concentrate on the dominating $O(N^2)$ costs, and ignore most linear and constant costs.

Equation (4) is essentially a matrix-vector multiply and add, where **A** is lower-triangular. A lower-triangular matrix has at most $N(N+1)/2$ nonzero coefficients, so in total **A** and **m** require the storage of $N(N+3)/2$ coefficients. During the generation of each random vector all (lower-diagonal) coefficients of **A** are accessed once, then each coefficient is used as input to one multiplication. Adding together the terms (including **m**) requires the same number of additions, so $N(N+1)/2$ multiplies, adds, and coefficient accesses are needed per generated vector.

If one vector is to be generated per cycle, then the only practical way to store **A** and **m** is in registers, as each coefficient is accessed once per cycle. Under this schedule the performance requirements grow rapidly, making it practical only for small N. However, if vectors are generated serially such that N cycles are used to generate each vector, it is possible to store coefficients in RAM.

The top half of Table I shows the equations for abstract computational resource usage, then the bottom half relates them to implementation, in terms of registers for the parallel version, and memories for the serial version. The constants w_c, w_s, and w_r denote the width of coefficients, adders, and RAM elements respectively, while l_r and p_r are the length and number of ports of each block RAM.

Figure 1 shows these estimates applied to the Virtex-4 xc4vsx55, using $w_c = 18$, $w_s = w_r = 36$, $l_r = 1024$, and $p_r = 2$. It shows that the largest vector that can be generated in the parallel case is around 30, but the RAM based serial generator can support up to 512.

For comparison purposes we also estimate the performance of software, again considering only the dominating $O(N^2)$ costs. As software is almost unlimited in the size of matrix that can be stored in memory, the performance when operating out of different memory hierarchies is considered. Each level i can hold l_i coefficients, and provides a peak bandwidth of b_i coefficients per second. For simplicity it is assumed that when the coefficients cannot be held totally in one level then they are all held in the next level. The CPU is assumed

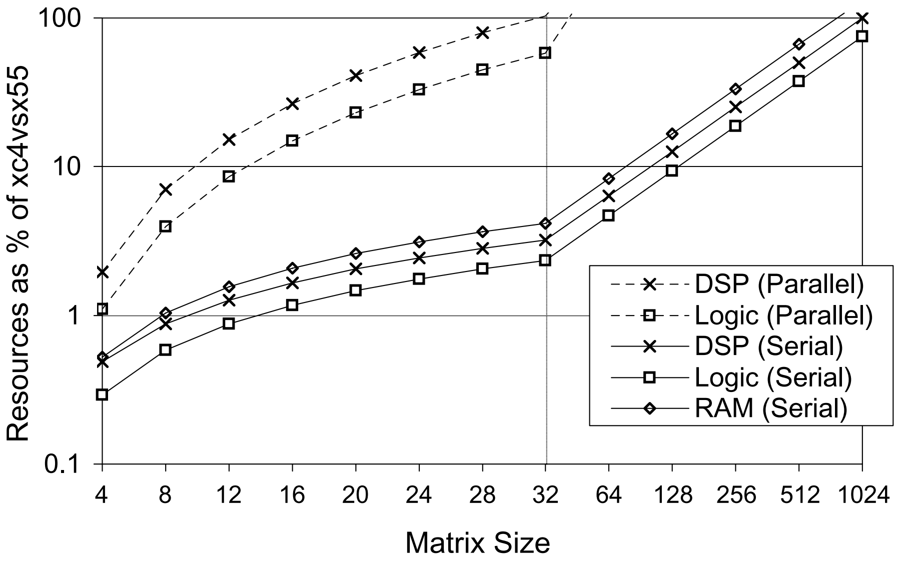


Fig. 1. Estimate resource usage for parallel and serial vector generation.

to be capable of m multiply-accumulates per second. The performance of the CPU for N length vectors per second is then estimated as:

$$i = \min_{i} \ : \ N(N+3)/2 \leq l_i \tag{10}$$

$$V_N = \min\left(\frac{2b_i}{N(N+3)}, \frac{2m}{N(N+1)}\right). \tag{11}$$

When implementing the vector generation in an FPGA, it is possible to replicate instances if each generator takes up less than half the space. So the total expected FPGA performance in vectors per second is:

$$V_N = \lfloor 1/p_N \rfloor f/N, \tag{12}$$

where f is the clock frequency, and p_N is the proportion of the FPGA taken up by a single vector generator of size N.

Figure 2 shows the expected performance for an Opteron at 2.6GHz and an xc4vsx55 at 500MHz. The predicted software performance uses the following estimated figures for the cache and floating-point performance: $m = 1.04 \times 10^{10}$ (four vectorized single-precision operations per cycle); $l_1 = 2^{14}$; $b_1 = 2.08 \times 10^{10}$; $l_2 = 2^{20}$; $b_2 = 5.75 \times 10^9$; $l_3 = \infty$; $b_3 = 3.2 \times 10^9$. Both memory and operation bounds are shown for the Opteron, with the memory forming the lower bound on performance. For small matrix sizes the software should operate mainly within the first-level cache, suggesting a hardware speedup of only about 20 times, while for larger sizes the coefficients must be stored in main memory, leading to a potential 100 times speedup.

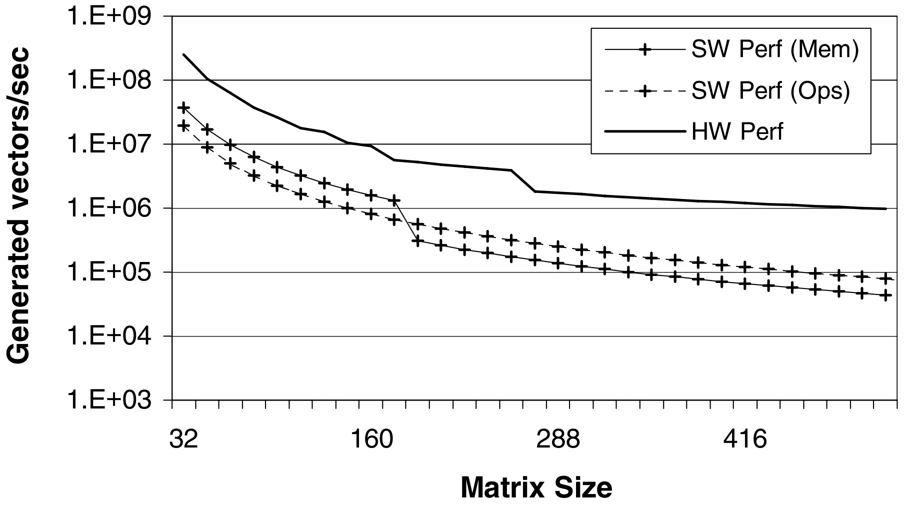


Fig. 2. Comparison of expected performance of xc4vsx55 and Opteron 2.6GHz.

Note that these figures are approximations, as both platforms are unlikely to achieve this performance in practice. The software version is extremely unlikely to achieve perfect utilisation of memory and bandwidth and function units, while the hardware figures for smaller vectors assume that the Input/Output constraints in outputting larger numbers of vectors at once are not a problem. However, in both cases the estimates for larger matrix sizes might be expected to become asymptotically accurate, particularly in the FPGA case.

2.2 Number Representation

The basic generation algorithm described in Equation (4) is exact, but only under the assumption of infinite precision maths. A double-precision software implementation provides a reasonable approximation to this ideal, at least for matrices smaller than 1024×1024, but using large numbers of double-precision units is still infeasible in contemporary FPGAs. Even single-precision floating-point carries a significant performance and area penalty, so only a fixed-point solution is feasible for large matrices. For maximum efficiency (both in area and speed) the fixed-point representation must be dictated by the available DSP blocks, as assumed in the previous section. In this section we examine the question of number representation and matrix coefficient ranges. The effect of the resulting numeric precision on the generator is then examined in more detail in Section 5.

The probability of a random univariate Gaussian sample lying outside the range $[-8, +8]$ is approximately 10^{-15}, so a signed fixed-point representation with 3 integer bits is sufficient in almost all situations. A counterexample is in Bit-Error-Rate testing [Lee et al. 2004], where extremely long simulations are used to explore events occurring in the range $[-8.2, 8.2]$, but in most simulations these extremely unlikely values are not important. In the case of the 18-bit multiplier inputs on the Virtex-4, this means that the elements of the input uni-variate Gaussian vector **r** have 14 bits of fractional precision. This leads to a minimum probability separation of values at zero (the mode of the Gaussian distribution) of $\Phi(2^{-14}) - \Phi(0) = 2.43 \times 10^{-5}$.

When looking at coefficient representations, we make two reasonable simplifying assumptions:

—The accumulation of matrix partial-products is exact.

—The output of the generator can be scaled by a binary power (i.e. shifted).

The first assumption is true in practise due to the 48-bit addition chain provided by the DSP48 blocks in the Virtex-4 (see Section 3.4), but is also possible with conventional adder trees, by increasing the bit-width of adders as the partial-products accumulate. The second assumption is reasonable as the bit-width of the output is far too large for most applications: the first thing they will do is select some reduced precision section of the output. For example, in the case study presented in Section 4, the first stage is a shifter to select an 18-bit segment from the vector generator output. Even if the application does not already have such a section, the logic to implement such a shifter is small compared to the rest of the vector generator.

The range of the coefficients of **A** depends on the marginal standard-deviations of the vector components. The standard deviation of each vector component is defined by the diagonals of the covariance matrix **S**, or equivalently, the rows of **A**:

$$\sigma_i = \sqrt{S_{i,i}} = \sqrt{\sum_{k=1}^{i} a_{i,k}^2}. \tag{13}$$

The largest standard deviation in the covariance matrix determines an upper bound on all coefficients in **A**. However, the largest coefficient might actually be much smaller than the largest standard deviation. Instead it makes sense to scale based on the largest coefficient. This leads to the *global* coefficient scaling scheme.

In the global coefficient scheme a single fixed-point w-bit data-representation is used for all coefficients in the matrix. First the maximum coefficient $\overline{a}$ in **A** is found. Then an integer shift s is found, such that $2^{w-2} \leq 2^s \times \overline{a} < 2^{w-1}$. All coefficients of **A** can then be multiplied by 2^s and rounded to w bits, enforcing a single representation on all coefficients. The shift s then determines the scaling to be applied to the overall vector output.

The problem with using a single global scaling factor is that one component with a very large standard deviation may cause the coefficients of smaller deviation components to be severely truncated. In the worst case this may mean that all coefficients of a much smaller standard deviation element are reduced to zero, reducing that component to a constant zero output. The solution suggested here is the *per-row* scaling method: instead of scaling coefficients to support the maximum coefficient in the matrix, each row has it’s own scaling factor, determined using the maximum coefficient in that row.

The advantage of this scheme is that, even if the marginal standard deviations are of vastly different magnitudes, the largest coefficients for each row will be stored with maximum accuracy. It still may be the case that very small coefficients within a row are reduced to zero, but arguably these coefficients are not significant anyway. Using 18-bit coefficients we can guarantee that the largest coefficient in each row is stored with 17 fractional bits, and even a coefficient a thousand times smaller is still represented using 7 bits. The disadvantage of this scheme is that now a separate scaling factor must be stored per vector component, but this is easily accommodated in a small N element RAM (e.g., a single Block-RAM).

3. ARCHITECTURE

In this section the mapping of a vector generator into reconfigurable hardware is presented. Following the analysis in the preceding section the architecture implements the serial model, generating a length N vector every N cycles. The architecture design is heavily optimised to take advantage of Virtex-4 specific resources, in order to achieve the maximum possible clock rate.

The calculations for implementing Equation 4 in hardware can be broken into four stages:

Univariate Gaussian Vector Generation. Every N cycles a new vector of independent univariate Gaussian samples must be generated.

Coefficient Management. The coefficients of matrix $\mathbf{A}$ must be extracted from RAM in the correct order, and a means of loading new matrices must be provided.

Multiplication. The univariate Gaussian components and matrix coefficients are multiplied together.

Summation. The products are summed to provide the components of the output vector in successive cycles.

3.1 Univariate Gaussian Vector Generation

To produce each output vector, N independent Gaussian samples must be generated. In a naive implementation this could be achieved using N separate generators, with each generator producing a new random sample at the start of each vector and holding it constant for the remaining $N-1$ cycles. This concept is shown on the left-hand side of Figure 3. Although simple, this is very wasteful, as Gaussian random number generators are relatively large and expensive, typically requiring a number of block-RAMs and often a number of multipliers.

It is much more efficient to use just one univariate Gaussian generator, observing that, if the generator produces one random sample per cycle, then while one vector is being generated and output, the next univariate input vector can be generated in parallel. It then becomes a problem of distributing the univariate vector elements to the appropriate multipliers. The solution used here is to use a long shift-register, with its input connected to

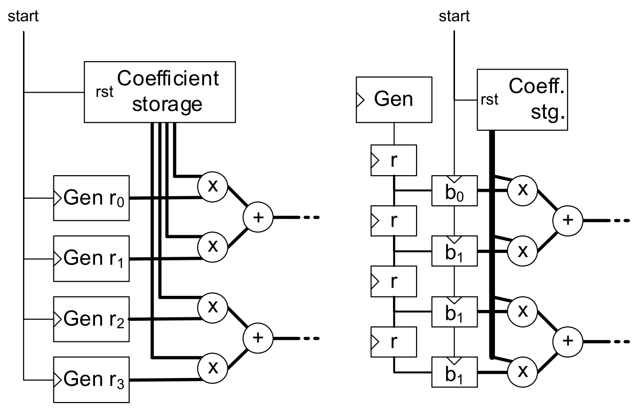


Fig. 3. Generation and distribution of univariate Gaussian samples.

the univariate generator, and having each register stage close to a multiplier site. This arrangement is shown in the right side of Figure 3. During the N cycles of multivariate generation the shift-chain is serially loaded with N fresh univariate samples. When calculation of the next multivariate vector begins, the contents of the shift-register are transferred in parallel to registers located by each multiplier.

As well as requiring only one Gaussian generator, this arrangement is also very appropriate for high-speed designs, as only local routing is required between registers in the shift-register. The shift-chain can be placed so that alternating columns in the chain shift up and down, as it doesn't matter what order the samples reach the multipliers, as long as each sample is only used by a single multiplier. Short horizontal connections at the top and bottom of the columns can then be used to create a single global shift-chain.

The two requirements on the univariate generator are that it should be fast enough to meet the clock-constraint, and that it should provide a high quality distribution. The resource usage is less important, as only one generator is needed for each vector generator.

There are many architectures for generating Gaussian samples in an FPGA, such as the Box-Muller [Lee et al. 2006; Xilinx, Inc. 2002], Wallace [Lee et al. 2005], and Ziggurat [Zhang et al. 2005] methods. However, these all use multipliers, and so would require some of the device's block-multipliers to be diverted from the matrix-vector multiplication stage. They are also relatively complex internally, and would require significant effort to reach the desired 500MHz target speed.

The generator used in this article employs piecewise-linear approximations [Thomas and Luk 2006b], as this method does not require any multipliers, and is very easy to pipeline for high-speed operation. The output of a piecewise-linear generator can only ever be an approximation to the PDF, so the outputs of four separate generators are additively combined. Due to the Central-Limit Theorem the combined output is significantly closer to the Gaussian distribution, and passes all the statistical tests we applied (see Section 5).

3.2 Coefficient Management

Coefficient management consists of two tasks: extracting elements of **A** in the correct order during calculations, and providing a means of loading in new elements when the matrix **A** changes. During the N cycles taken to generate each vector, each multiplier requires at most N different coefficients. Depending on the aspect ratio of available RAMs, this may require a memory port per multiplier, or with wide RAMs it may be possible to pack multiple coefficients into a single word. The entries in the RAM can be arbitrarily re-ordered to allow efficient indexing schemes, so it is not necessary for the coefficient layout in RAM to reflect the logical structure of **A**.

In this work, each Virtex-4 block RAM supplies two coefficients to two multipliers, with the RAM organized as a 1024 by 18 RAM. The two sets of coefficients are packed into the top and bottom halves of the RAM, and addressed using a single counter, with the most-significant bit set to 0 for one port, and to 1 for the other port. During the first cycle of vector generation the counter is reset to zero, then during successive cycles it is incremented by one.

The task of loading new coefficients should be a relatively infrequent operation compared to the actual generation of vectors, and so does not need to be heavily optimized. A very convenient and efficient method of updating coefficients is to reuse the shift-chain used to distribute univariate Gaussian numbers. During coefficient update, coefficients are serially loaded onto the shift-chain, until all N coefficients for a row are ready. The controller then starts the standard vector generation process, causing the address counters for all RAMs to start incrementing. At the appropriate cycle the controller then asserts a write strobe, causing the value of the shift-chain to be written into the memory location associated with that cycle.

Loading in this way means that entire rows can be written at once, with each row requiring N cycles to fill the shift-chain, and between 1 and N cycles to reach the correct point in vector generation for the row to be written. Each row can thus be written in $2N$ cycles, so the entire matrix can be loaded in $2N^2$ cycles. Assuming the xc4vsx55 parameters of $N = 512$ and a 500 MHz clock, this means that a new matrix can be loaded in just over one millisecond.

3.3 Multiplication

The multiplication stage multiplies together the coefficients and random numbers, producing the terms that will be summed together in the next stage. If the multiplier units provide no additional functionality (for example, Virtex-II block-multipliers [Xilinx, Inc. 2000]) then no further processing is performed at this stage. However, modern FPGA architectures provide more complex DSP blocks, fusing multipliers and wide adders together, such as the Stratix-II DSP [Altera Corporation 2005] and Virtex-4 DSP48 [Xilinx, Inc. 2005] blocks. These blocks allow some of the terms to be summed at the same time as they are produced, reducing the number of adders that must be implemented in general logic.

In the case of the Stratix-II, the DSPs support local addition of the four 36-bit values produced by four 18 by 18-bit multipliers in the DSP. This pushes

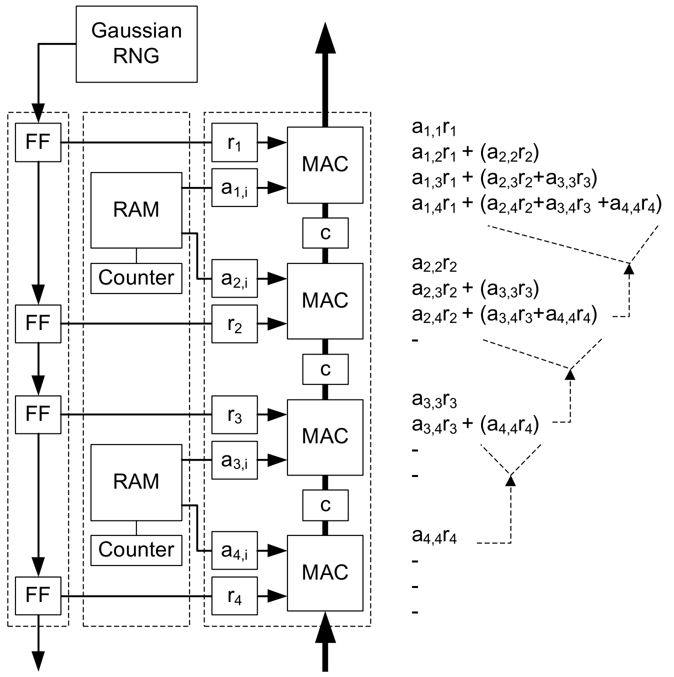

Gaussian RNG
FF
FF
FF
FF
RAM
Counter
RAM
Counter
r_1
$a_{1,i}$
$a_{2,i}$
r_2
r_3
$a_{3,i}$
$a_{4,i}$
r_4
MAC
MAC
MAC
MAC
c
c
c
$a_{1,1}r_1$
$a_{1,2}r_1 + (a_{2,2}r_2)$
$a_{1,3}r_1 + (a_{2,3}r_2+a_{3,3}r_3)$
$a_{1,4}r_1 + (a_{2,4}r_2+a_{3,4}r_3 +a_{4,4}r_4)$
$a_{2,2}r_2$
$a_{2,3}r_2 + (a_{3,3}r_3)$
$a_{2,4}r_2 + (a_{3,4}r_3+a_{4,4}r_4)$
$a_{3,3}r_3$
$a_{3,4}r_3 + (a_{4,4}r_4)$
$a_{4,4}r_4$

Fig. 4. Architecture of MAC-based multivariate Gaussian generator.

two levels of addition into the DSP block, so if the output of n multipliers must be combined this means that only $(n/4) - 1$ of the required $n - 1$ adders must be implemented in general logic.

However, the Virtex-4 DSP48 supports a 48-bit wide carry path that runs the entire height of DSP columns, allowing each DSP to perform an 18x18-bit multiply, and combine the 36-bit result with the 48-bit result of the DSP block below. This means that in a device with w DSP columns each containing h DSP devices, potentially $(h - 1)w$ additions can be implemented in dedicated addition hardware, so if all hw DSPs are used in vector generation only $w - 1$ additions need to be implemented in general logic. However, the pipelining of the DSP carry path means that the data processing must be rescheduled to allow the DSP adders to be used.

Fortunately the fact that matrix **A** is triangular allows the calculations to be rescheduled to take advantage of this dedicated carry path. Consider Equation (6) (ignoring **m** as this can be processed in the final stage): only one of the elements (x_N) requires N multiplications and uses all N elements of the random vector **r**. If this calculation is started in the first cycle of vector generation, then over the following $N - 1$ cycles the remaining terms can be accumulated, and will eventually be output as the last element of the vector. In contrast there is one element that only contains one multiplication, which can be executed in the first cycle and output as the first element. In general the calculations can be organised so that in cycle k of the vector generation process, the element derived from k multiplications and $k - 1$ additions can be output.

This way of scheduling the calculation takes advantage of the fact that **A** is lower triangular, and hence contains a large number of zeroes.

Figure 4 shows how this schedule maps to the DSPs. On the left-hand side is the univariate Gaussian distribution chain, which will be captured (in parallel) into $r_1..r_4$ at the beginning of vector generation. In the middle are the coefficient RAMs, and counters for selecting the coefficients in the correct order. Finally the DSP column contains the buffers $r_1..r_4$, which are held constant during each generated vector, the MAC units which perform $r_i a_i + c$, and the registers on the data carry path. The right hand side shows the calculations performed in each MAC in each cycle. In the first cycle the terms $a_{i,i} r_i$ are all calculated, and $x_{1,1}$ can be output. In following cycles the elements with more and more terms reach the top of the column, until finally the last n-term element reaches the top.

3.4 Final Summation

Single-cycle logic-based adders are performance limited by the critical path through the carry-chain. If an adder of length w is implemented using a carry-chain with a delay per full-adder of d_c, then the maximum clock-rate that can be supported is $1/(wd_c)$. In Virtex-4 $d_c = 0.07$ ns, so the maximum performance at $w = 45$ is 317MHz, even before considering factors such as LUT delay and routing inputs to the LUTs. Clearly to achieve 500MHz performance the adder carry-chains must be pipelined, but current synthesis tools do not automatically pipeline adders, even when given tight timing constraints and ample registers for retiming.

The solution used in this work is to use relationally placed adders with explicitly pipelined carry chains. Each w-bit adder is broken into segments of length s, each of which adds together two s-bit numbers and supports a carry-in and carry-out. Each adder is implemented using $k = \lceil w/s \rceil$ segments, and the carry-out of each segment is registered before being routed in to the carry-in of the next segment. The pipelined carry-chain means that segments must be skewed in time, with the least-significant segments arriving first. This skew is achieved using a triangle of registers attached to the adder inputs, in this case the output of DSP48 blocks. The final sum is de-skewed using another triangle of registers.

Figure 5 shows the skewed-adder system for $w = 9$ and $s = 3$. Note that each adder-stage includes an additional stage of registers placed right next to the addition logic; this is included as it allows the adder inputs to have the absolute minimum routing delay, allowing each segment to be slightly longer. The cost of each triangle of registers is $sk(k+1)/2$ FFs (Flip-Flops), and for an n input adder tree a total of $n + 1$ triangles are required. Assuming n is a binary power, $n - 1$ skewed adder stages are required, each of which uses $(3s + 1)k$ LUT-FF pairs (including FFs for input buffering). Thus the total cost of an adder tree in LUT-FF pairs is:

$$\frac{1}{2}(s(n(k + 7) + k - 5) + 2n - 2)k. \tag{14}$$

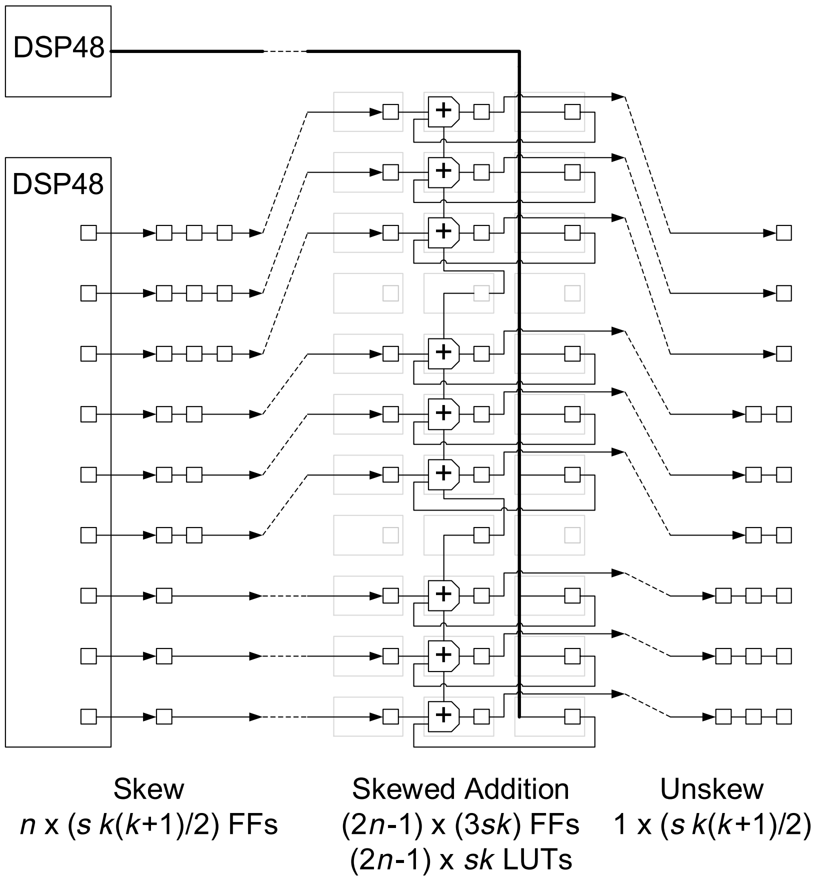


Fig. 5. Pipelined adder tree.

In the xc4vsx55 architecture the value $s = 9$ is chosen, requiring $k = 5$ segments. When all DSP48 columns are used for vector generation $n = 8$, a total of 2195 LUT-FF pairs ($\sim$ 4% of total resources) are consumed in the summation tree. Note that the area could be decreased by removing the extra buffering stages and using longer addition segments, but this would make meeting 500MHz timing constraints much more difficult. Another solution would be to dedicate the top DSP48 of each column to addition, and to stagger the initiation of new vector calculations in successive columns. This would remove almost all logic needed for addition, but would also reduce the maximum possible vector size to 504.

3.5 Implementation and Evaluation

The generator we have described is implemented in Virtex-4 specific VHDL, designed to give maximum performance. The design uses two main building blocks, one to describe the multiply and accumulate element shown in Figure 4, and another to describe the pipelined adder components in Figure 5. Both

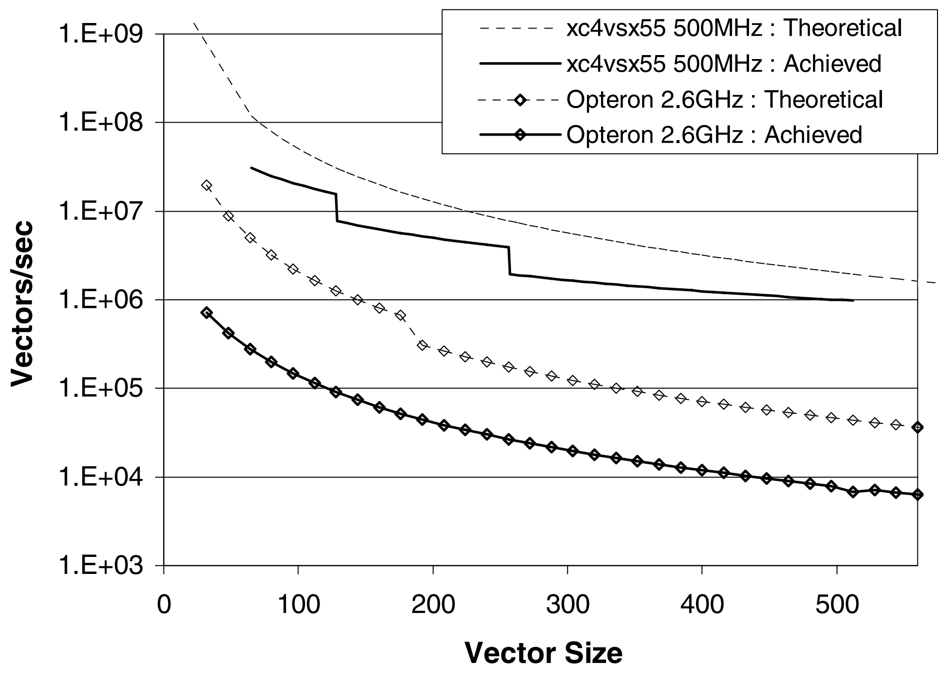


Fig. 6. Performance comparison between software and hardware vector generators, both according to the performance model and realised performance.

are individually tuned and relatively placed to ensure 500MHz performance could be achieved. The blocks are then instantiated and connected using parametrised container components, creating columns of multiply-accumulate elements, and trees of adders. The containers apply absolute placement constraints to the blocks, and are individually synthesised to EDIF blocks. The set of EDIF multiply-accumulate columns and adder tree are then combined in a top-level design, which is placed and routed for the xc4vsx55-12. All inputs and outputs are routed to pins, with an intermediate layer of buffering registers clocked at the same rate as the vector generator. Four different top-level designs are created, using four, two and one replicated instances of the vector generator, capable of handling vectors up to length 128, 256, and 512 respectively.

The Xilinx 8.1 toolchain is used throughout, using XST for synthesis, and Xilinx primitives to instantiate almost all registers and other components. Shift-register inference and optimization of primitives is disabled during synthesis, as XST attempted to optimize registers into SRL16s, without realizing that they were deliberately introduced for timing reasons. Timing driven placement is employed, but other advanced map options such as retiming are not needed. Place and route at the default effort level is found to be sufficient.

The maximum clock rate of the design is reported by Xilinx tools as 500MHz, which is the maximum supported by the block RAMs and DSPs, so the practical performance ceiling has been achieved. Figure 6 shows the performance of the generator as vector size is increased. The two steps in performance are due to the switch between 4, 2, and 1 replicated instances as the matrix size (and the size of each instance) gets larger. The theoretical maximum performance derived in the preceding section is also shown above the practical performance. The difference between theoretical and achieved performance changes most significantly at the steps between the boundary sizes for replicated instances, with a best-case realized performance of 1/2 the theoretical maximum when the vector size equals the maximum vector size, and a worst-case of 1/4 for the next larger size.

Underneath the performance curves for hardware are the predicted and realized curves for an Opteron 2.6GHz (model 2218). The realized software performance is measured using the ACML BLAS [Advanced Micro Devices 2006], a set of vector and matrix libraries specifically optimized for AMD processors, using features such as SIMD and cache management instructions. Single-precision floating-point is used to allow maximum software processing speed. No equivalent fixed-point software version is shown, as SIMD floating-point is faster than integer-based fixed-point, so there would be no performance advantage. Performance is measured on a Linux based quad-core Opteron server, with no other computational tasks running, and ensuring each run is measured over at least 10 seconds of wall-clock time.

The maximum speedup over software of 205 is achieved when four instances operate on 128 element vectors, and the minimum speedup of 91 with vectors of size 257 (at the point where it is necessary to move from a two instance design to a one instance design). For the largest supported vector size of 512 the speedup is 174 times.

These results demonstrate performance when an entire device can be dedicated to vector generation. In the next section an application is developed that incorporates the vector generation architectures into a real design, allowing practical speedup to be measured.

4. CASE STUDY

In this section the Gaussian vector generator is used in a real-world application, a Delta-Gamma asset simulator for Value-at-Risk. The simulator is implemented using an xc4vsx55-10 in a Celoxica RC2000 platform, so the maximum clock rate reduces to 400MHz when compared with the 500MHz that could be achieved using the xc4vsx55-12 in the previous section.

In many financial simulations a large collection of underlying assets is modeled as having correlated Gaussian returns. An underlying asset is something like a stock, a bond, or a physical commodity, which can be bought or sold directly. A portfolio over these underlying assets can then be modeled, where the portfolio incorporates both direct positions, as well as options and other derivatives on the underlying assets. A direct position is where the portfolio directly contains one of the underlying assets: for example, the portfolio contains 100 Microsoft shares. The value of the direct position in the portfolio then varies linearly with the price of Microsoft shares in the market.

Derivates are assets whose value is derived from changes in value of some underlying asset. A simple example is an option, which is a contract providing the option (but not the obligation) to buy some underlying asset A at time T for a fixed price S. The value of this option varies with the value of A, but the relationship is complex, and also depends on T and S. Accurately pricing the option may require significant calculations, for example the commonly used Black-Scholes pricing operator requires evaluations of the Gaussian CDF and many other functions [Black and Scholes 1973].

A simple way of approximating changes in portfolio value over short time periods is to consider only the first and second derivatives of portfolio assets with respect to changes in the underlying. Thus if **s** is the vector of original positions, and **x** is a random vector of correlated changes in the underlying assets, then the new price is

$$p = \sum_{i=1}^{N} s_i + \delta_i x_i + \frac{\gamma_i}{2} x_i^2, \tag{15}$$

where δ is the vector of first derivatives; γ is the vector of second derivatives. Rearranging this to extract just the change in price gives:

$$d = \sum_{i=1}^{N} x_i(\delta_i + \frac{\gamma_i}{2} x_i), \tag{16}$$

By simulating millions of different random **x** vectors the probability distribution of d can be estimated, and used to evaluate the portfolio. For example, picking the 5th percentile amongst all values of d gives the 5% Value-at-Risk, which is the amount of money that would be lost in the 5 worst days out of every 100. The computationally intensive process is in the calculation of d, so the remainder can be implemented in software.

To allow full-speed operation in hardware these calculations are performed using fixed-point, but with a different scaling (binary-point position) for each vector element. The scaling factor is chosen in advance in software, to maximise the accuracy of each calculation while ensuring that overflow does not occur. This is possible as the range of **r** is already known (via the marginal standard deviation and mean of each component), establishing an upper bound on the magnitude of each element. Thus the 45 bit values produced by the correlated vector generator can be reduced with shifters down to 18 bit, even when the standard-deviations of asset returns have very different magnitudes. This stage can also transparently incorporate the per-row scaling coefficient described in Section 2.2.

A second scaling is applied prior to the summation of price changes, allowing assets with very different sensitivities to be accommodated. The final Delta-Gamma calculation is then:

$$d = \sum_{i=1}^{N} 2^{-k_i} \left(\left[2^{-j_i} x_i \right] \left[\delta_i' + \gamma_i' \left(2^{-j_i} x_i \right) \right] \right), \tag{17}$$

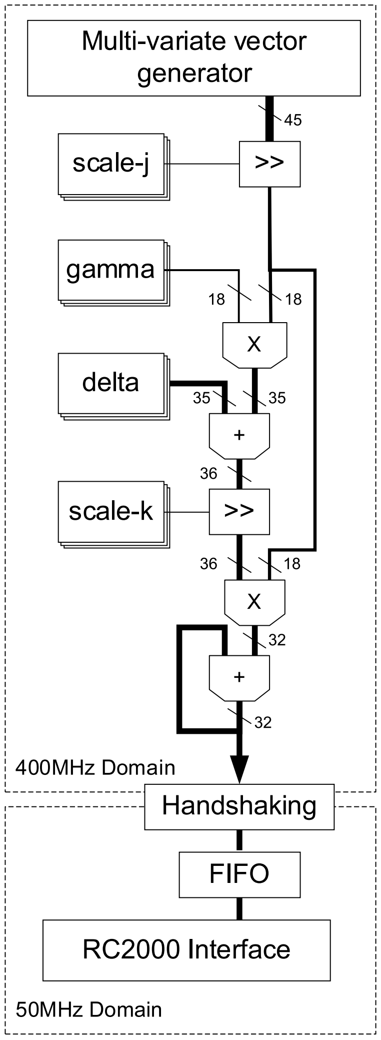


Fig. 7. Delta-Gamma approximation pipeline.

where $\mathbf{j}$ is a vector of integers that determine the initial scaling of $\mathbf{r}$, $\mathbf{k}$ is a vector of integers that determine scaling before summation, and δ' and γ' are the portfolio sensitivities after applying the scaling.

Figure 7 shows the pipeline used to implement the Delta-Gamma pricing operator, including the widths of data-paths. DSP48s are used to implement all the multipliers and adders, while pipelined LUT-based multiplexors are used to implement the shifters. The final sum is performed using a DSP48 in accumulation mode, with the accumulator reset to zero at the beginning of each new vector. All calculation components are placed relative to each other to provide a relatively compact pipeline, while the registers used for pipelining and synchronising datapaths are unplaced.

Each vector of random asset returns produces one total pricing change, and this is the result that needs to be passed back to software for further processing. If a portfolio of size N is generated at frequency f, then simulated price changes are generated at a rate f/N, so in large portfolios (which present a significant computation challenge in software) the frequency of results that are transferred back to software will be significantly lower than f. The results can thus be transferred into a slower clock-domain without data-loss.

This implementation targets the RC2000 platform, so the slower clock-domain needs to support both the RC2000 local-bus interface, and the logic that implements the protocol used to communicate with host software. This section is implemented in Handel-C and a clock rate of 50MHz is chosen, with the faster clock domain operating at 400MHz via two levels of DCMs.

Ideally the clock-domain bridges would have been implemented using the Virtex-4 built-in FIFO16, but due to the LUT-based workaround needed for correct operation [Xilinx, Inc. 2006], the faster clock domain could not reach maximum speed. Instead a simple register-based protocol is used to transfer single words, where the faster clock domain holds all data changes constant for at least three cycles. The slower clock domain registers the output of the faster domain's transfer register every cycle, and detects new data using a single validity bit within transferred words. Incoming data are captured the cycle after the validity bit is asserted, ensuring that all bits of the transferred word have been retrieved correctly. This system requires six cycles in the slower clock domain per transferred word, but the data-transfer rate between the clock-domains is slow enough that this is not a bottleneck. With the 50MHz clock domain this allows a minimum vector size of $N = 48$.

Figure 8 shows the high-level layout of the simulator in the xc4vsx55. The left half of the device is dedicated to full-height columns of matrix-multiply components, containing 256 DSPs. The right half of the device uses three-quarter height columns of matrix-multiply components, containing another 192 DSPs, allowing for generation of vectors up to length 448. The columns are synthesised to eight EDIF components from a single VHDL description, parametrised for column height, absolute placement, and whether the shift-chain goes up or down the column. The adder tree is synthesised from another VHDL description, with absolute placement for the adders.

The bottom right corner is reserved for the Delta-Gamma pipeline, and the Gaussian random number generator, both of which are implemented as relatively placed components, and are then absolutely placed within the device at synthesis time. The figure also shows the control logic and RC2000 interface in the same area, but this is more conceptual. Actual placement is left to the tools, and as a result the logic for these components is distributed throughout the entire device.

The synthesised components are all gathered together in a top-level design and placed and routed as a single design. No top-level placement or area constraints are used, as all timing-critical components are already absolutely placed. A 400MHz clock-constraint is placed on the main processing clock, and is met.

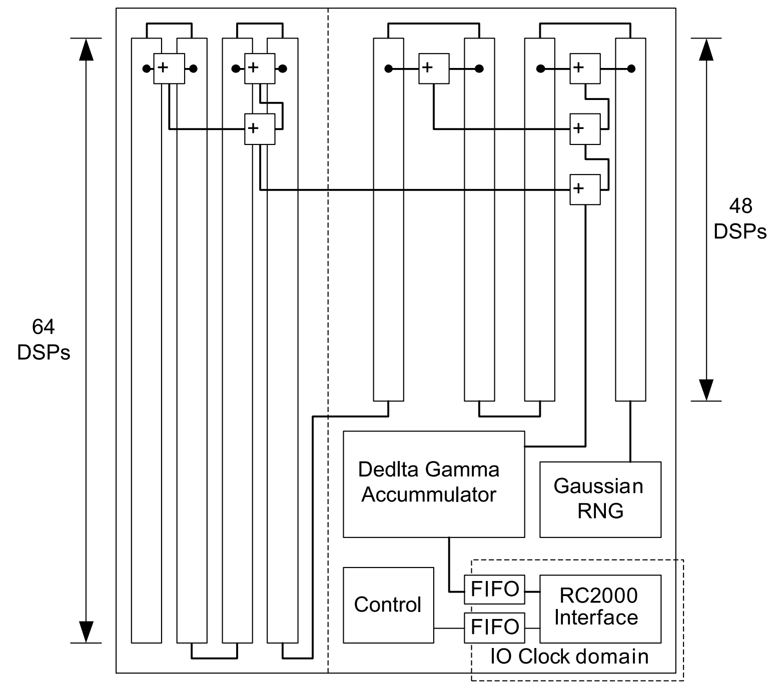


Fig. 8. Layout of components in xc4vsx55 based RC2000.

Table II. Resource Usage for Delta-Gamma Simulator, with Percentage of xc4vsx55 Resources Shown in Brackets

	Vector Generator		Gaussian Generator		Delta Gamma		Control		Total	
DSP	448	(87.5)	-	-	4	(0.7)	-	-	452	(88.2)
RAM	226	(58.9)	4	(1.0)	4	(1.0)	19	(3.7)	253	(65.9)
LUT	9006	(18.3)	955	(1.9)	822	(1.6)	504	(1.0)	11287	(23.0)
FF	18771	(38.1)	1278	(2.6)	1935	(3.9)	803	(1.6)	22787	(46.4)
Slice	12673	(51.6)	947	(3.8)	1523	(6.1)	1210	(4.9)	16353	(66.5)

Table II shows the resources used in the design, broken down by component. The LUT and FF resource usage of all components is higher than strictly necessary, particularly in the Delta-Gamma pipeline, since the focus is on achieving the maximum possible clock rate without requiring large amounts of design time. The majority of the resources are used in the vector generator, as even though the amount of logic per DSP is very small (approximately 19 LUTs and 34 FFs per cell), it is replicated at each location. Many resources could be saved by sharing local control logic and address counters across multiple DSP units, but this would make achieving timing closure much more difficult.

Figure 9 compares software and hardware performance, by measuring the time taken to simulate 10^6 portfolio returns. Monte-Carlo simulations can be trivially parallelised across multiple CPUs, simply by starting up as many simulation processes as there are CPUs, and in this case the speedup was exactly

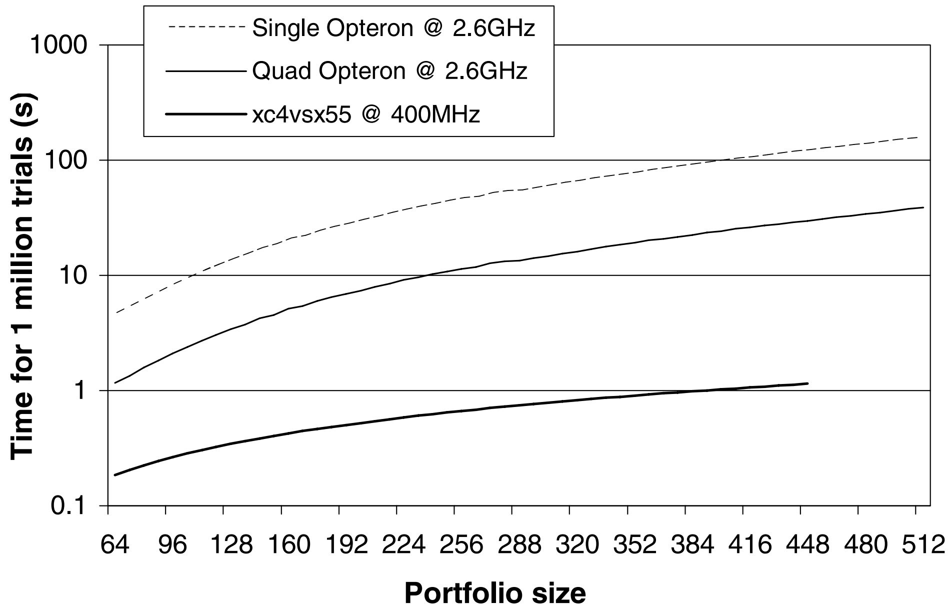


Fig. 9. Time taken to perform 1 million portfolio evaluations.

four times (to within the bounds of measurement error). Note that, in principle, a multiple FPGA system could potentially achieve a linear speedup in exactly the same way, by running independent simulations on different FPGAs.

The hardware execution time is measured as the wall-clock time taken to complete and receive the results for all 10^6 portfolio returns into a software buffer (the same as for the software). This includes the time taken to load the correlation matrix over PCI (24ms), but does not include the time taken to configure the FPGA. Note that there is no other data-transfer overhead, as successive portfolio values are sent from the FPGA to the CPU as they are generated, during the simulation runs. The required data-rate is well within the bandwidth capabilities of PCI, varying from 33MB/s for $N = 48$ down to 3.5MB/s for N=448.

For smaller portfolios the hardware is less than 10 times faster than the Quad Opteron, as very few of the hardware's multipliers can be applied to the problem. However, as the portfolio size increases the relative performance of the hardware increases, achieving a maximum speed-up for the maximum supported portfolio of 448 assets. This is 26 times the speed of a quad Opteron, or 104 times faster than a single Opteron, at less than a fifth of the clock rate.

Only one delta-gamma generator design was created, supporting a maximum of $N = 448$. This means that the performance curve remains smooth as the vector size increases, as compared with the jumps in performance seen in Figure 6, which are caused when the number of vector generators instantiated within the design changes. The bottleneck in the Delta-Gamma generator is the vector generation, but because the Delta-Gamma generator supports only one vector size, the maximum performance advantage of software occurs at this largest vector size, rather than at $N = 128$ as seen in the standalone generator tests.

This case study provides a useful, if somewhat simple, application that could be used in the real world. The maximum vector size of $N = 448$ allows very large number of underling assets to be modelled, and is sufficient for most portfolio models. Higher dimension models are used, but these typically involve sparse correlation matrices, rather than the dense empirically derived matrices used here. In practical terms, the main limitation of the case study is the simplicity of the delta-gamma pipeline, rather than limitations on the maximum vector length.

5. GENERATOR ACCURACY

A key requirement for a multivariate Gaussian generator is that the marginal distributions of the vector elements should be Gaussian, and that the correlation structure of the vectors is correct. In theory the generation method guarantees both, but in practice the real-world limitations of limited precision number representations and imperfect random number generators may cause the generator to fail. This is of particular concern in the hardware version presented here, due to the use of fixed-precision coefficients.

In this section we describe a number of empirical tests that have been applied to the generator architecture. These look for potential statistical defects in the generated vectors, as both the matrix size and the coefficient representation are varied. The first set of tests verifies that the marginal distributions are Gaussian distributed, followed by tests that examine the correlation structure for flaws. The section then concludes with experiments applied to the Delta-Gamma application from the case study.

Many of the empirical tests in this section produce p-values, which are real values between 0 and 1. Under the hypothesis that the generator passes a specific test, repeated applications of that test should produce p-values that are Independent Identically Distributed (IID) uniform random numbers: thus p-values very close to 0 or 1 are unlikely, and are grounds for rejecting the hypothesis. For example, if a test was executed three times, and produced the values 0.11, 0.78, 0.45, we might conclude that the test is probably being passed. However, if the values 0.05, 0.12, 0.01 were observed we might become suspicious, as all the values are close to zero. However, interpreting p-values is difficult, as they are inherently statistical: even very unlikely looking sequences should occur occasionally.

One approach is to reject the hypothesis that a generator passes a test if the p-value is outside [0.01..0.99], but this ignores the fact that in a set of 100 tests one would expect to see two values outside this range by chance. The approach taken in the following tests is that if a p-value outside [0.01..0.99] is observed the test is repeated, and the two p-values are combined into one p-value using Fishers method [Birnbaum 1954]. This is repeated until the combined p-value lies within the passing range [0.01..0.99], or until the p-value lies outside the range $[10^{-6}..1 - 10^{-6}]$, signifying a definite fail (none of which were found in the tests).

5.1 Marginal Distributions

The first requirement is that the source uni-variate generator must provide a sequence of IID Gaussian variates. The generator used in this article uses the Piecewise Linear method [Thomas and Luk 2006b], which approximates the Gaussian PDF using equal-length piecewise-linear segments. Because a generator only has a finite number of segments, a single generator does not provide a good approximation to a continuous distribution. To overcome this problem the source generator actually consists of four individual piecewise generators, which are additively combined. The output of the composite source generator is significantly closer to the Gaussian distribution than that of the individual generators, due to the Central Limit Theorem.

This generator combination was not tested in Thomas and Luk [2006b], so we applied a number of standard statistical uni-variate tests to this composite uni-variate generator. Note that the uni-variate generator actually outputs a 24-bit sample, even though only 18-bits are used in the vector generator, as otherwise there is insufficient resolution to reliably apply many of the tests. The tests applied to the generator are:

χ^2 Test. A χ^2 test [Pearson 1900; Marsaglia and Marsaglia 2004] over 2^{36} random samples, using 4096 equal probability buckets. This tests the overall shape of the distribution over very large numbers of samples, and is sensitive to defects in the asymptotic PDF of the distribution.

Anderson-Darling Test. The Anderson-Darling test [Anderson and Darling 1954] is applied to 256 successive batches of 2^{20} random numbers, then an Anderson-Darling test over the 256 test results is performed. The Anderson-Darling test is sensitive to defects in the tails of the distribution, and is not limited by the fixed bucket structure of the χ^2 test. Repeating the test detects any overall bias in the p-values of the individual tests.

Crush. The Crush test battery from TestU01 [L'Ecuyer and Simard 2007] is applied to samples that have been transformed to uniform samples using the Gaussian Inverse CDF [Marsaglia 2004]. The Crush battery comprises 96 separate tests for uniform randomness, consuming approximately 2^{35} samples. Note that the 24-bit Gaussian samples do not have enough resolution to provide 32 bit random samples after transformation, so the 16 least significant bits are taken from KISS, a known good generator [Marsaglia 1999].

All the tests were successfully passed, providing a high degree of confidence in the quality of the source generator. It is particularly important that the Crush tests are all passed, as many of these look for any existing correlation defects, which would interfere with the correlation we wish to impose.

5.2 Correlation Structure

The most obvious source of potential inaccuracy in the generator is the need to store the matrix coefficients in relatively small fixed-points values. The fixed-point representation is largely dictated by the properties of the target platform,

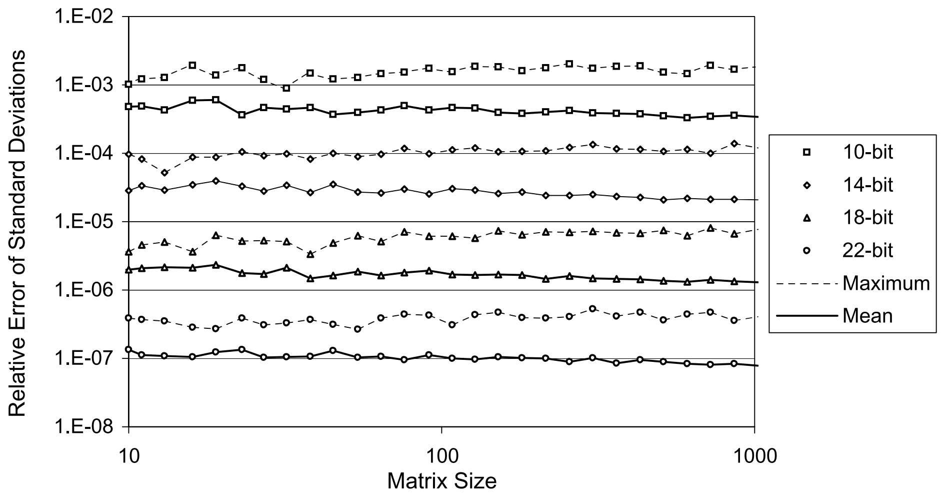


Fig. 10. Maximum and mean relative-error of marginal standard-deviations for increasing matrix size using per-row scaling.

and cannot be easily extended without wasting large numbers of resources. For example, the Virtex-4 target architecture used in this article requires 18-bit coefficients, as the DSP48 component only supports 18-bit inputs; supporting wider coefficients using extra LUT-based logic would mean the dedicated 48-bit carry chain could not be used (or at least not as efficiently). Given this externally imposed bit-width, it is important to check that it does not significantly affect the correlation structure of the output.

There are two semi-distinct features of the covariance matrix that must be reflected in the generated vectors: the marginal standard-deviations of the vector elements, the square-root of the matrix diagonals; and the pair-wise correlations, values in the range $(-1..1)$ derived from the lower triangle of the matrix. The marginal standard-deviations must have low relative error, as many applications are sensitive to small relative changes in variance. By comparison, applications are more likely to be sensitive to absolute errors in the correlation coefficients, as strong correlations (those near ± 1) matter much more than weak correlations (those near 0).

To test these features we apply a number of tests to randomly generated covariance matrices. These are constructed by first generating N vectors of 100 independent Gaussian samples, then randomly selecting pairs of vectors and adding a random fraction of one to the other. The covariance matrix between the vectors is then used as the target matrix, providing a matrix that is semi-positive definite and has a wide range of correlation coefficients and marginal standard-deviations. The target matrix is then quantised down to a fixed-point representation, either by using a global fixed-point scaling, or using per-row scaling (see Section 2.2).

Figure 10 explores the behavior of the marginal standard deviation's relative error for increasing matrix-size, using bit-widths from 10 to 22 and

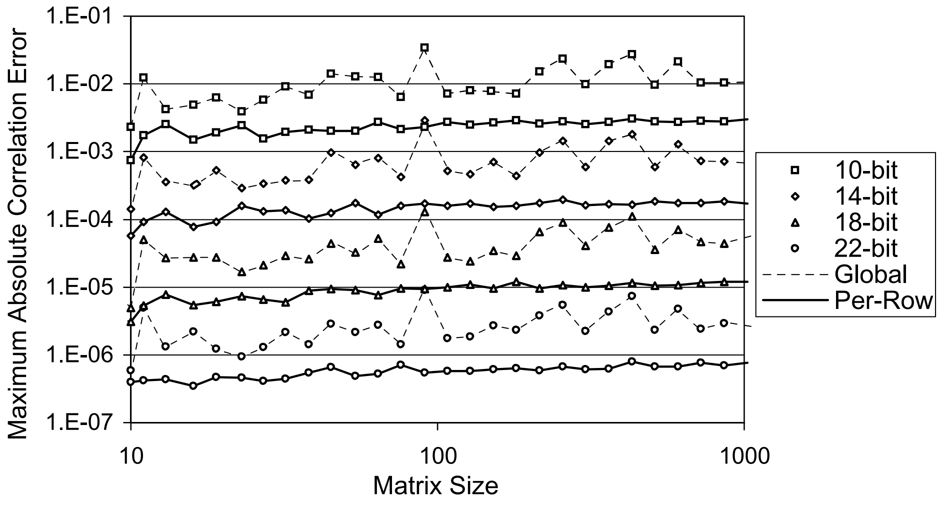


Fig. 11. Correlation coefficient errors for increasing matrix size, using different bit-widths for storage, and global or per-row scaling.

per-row scaling. The mean error remains relatively constant over the range of sizes, perhaps even decreasing a little for larger matrices, while the maximum error has a definite, but slow, increase. The F-Test [Press et al. 1997] uses the ratio between two variances to produce a p-value for the significance of the difference. By inverting the test, it is possible to roughly predict the number of samples at which the generator would fail the F-Test at the 5% level. Assuming the 18-bit generator has a worst case maximum of approximately 10^{-5}, this suggest that $\sim 2^{35.9}$ vectors can be generated before the largest (i.e., easiest to detect) error is noticeable.

Figure 11 shows the maximum absolute error of the generator's correlation coefficients for increasing matrix size. Results for four different bit-widths are shown, using both the global coefficient scaling method, and per-row scaling. Again, the maximum error slowly increases with matrix size, but it is noticeable that as well as providing a consistently lower error, the per-row scaling scheme is also much more predictable; in matrices with a large dynamic coefficient range, the global scaling method loses a significant amount of accuracy, causing the numerous spikes in error for all bit-widths.

As well as examining the asymptotic correlations, it is also necessary to investigate the empirical correlation structure, by looking at the actual generated vectors. The empirical behaviour is examined by generating 2^{14} vectors from a random 512×512 matrix, then calculating the empirical correlation matrix. Fisher's Z-transform [Press et al. 1997] is then used to test the hypothesis that each element of the empirical matrix is the same as that of the target correlation matrix, producing a matrix of p-values, one for each pairwise correlation coefficient. Although these p-values are not independent, the overall distribution can provide a strong indication of whether the generator's observed correlation matches the target correlation.

Figure 12 provides a visualization of the resulting p-values for three different bit-widths, and both the global and per-row scaling scheme. The upper triangle of each matrix shows the p-values converted directly to grayscale, and should be completely random, with no bias towards white or black. The lower triangle shows the same data (correlation matrices are symmetric), but singles out particularly suspicious p-values, showing those below below 10^{-4}, 10^{-3}, and 10^{-2} as black, dark gray, and light gray, respectively. Of the $\approx 2^{17}$ p-values in the lower-triangle, only about 1300 should be non-white, and about 13 should be black.

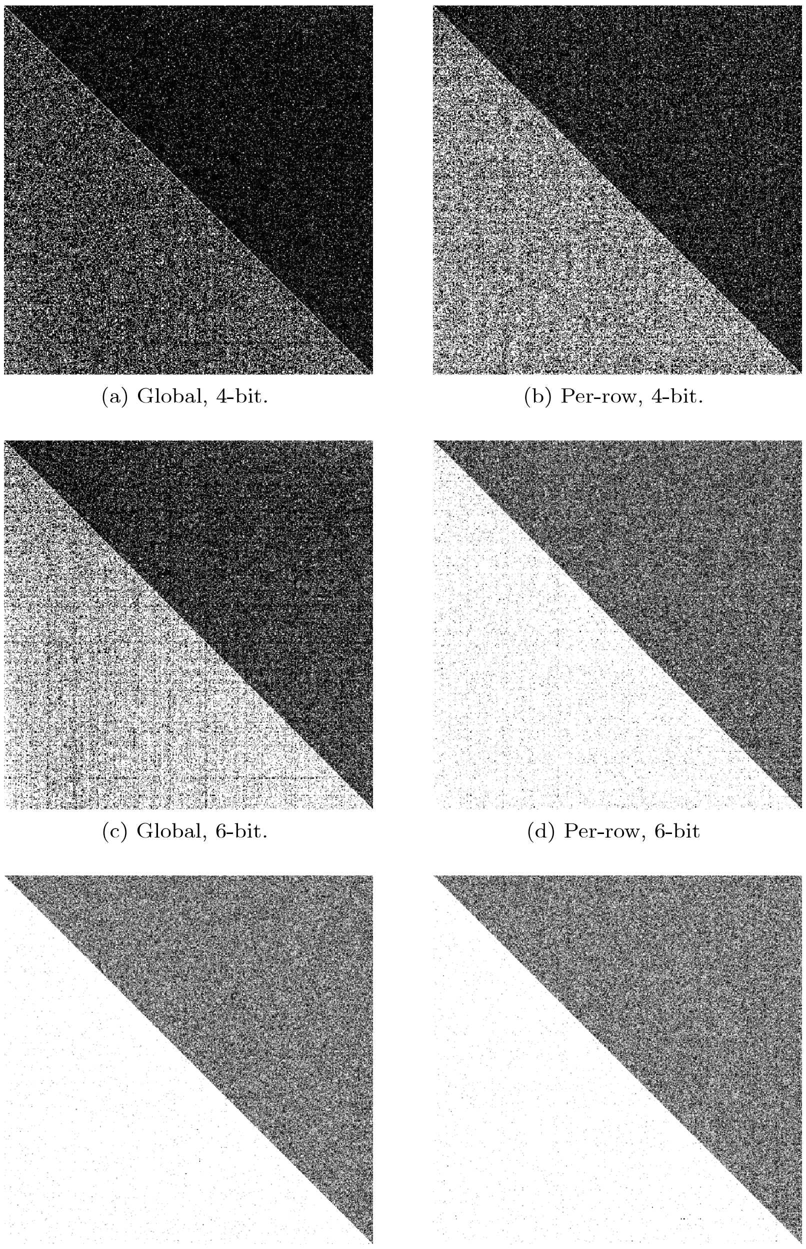
It is clear that 4-bit coefficients are causing significant failures, with huge numbers of p-values below 10^{-4}. At 6-bit the global scaling scheme is still very bad, but the per-row method begins to look respectable. In the 18-bit case there is no visual difference, and it is impossible to tell from inspection whether either is failing. The results are summarized in Figure 13, which shows the percentage of p-values that are below the 10^{-4}, 10^{-3}, and 10^{-2} levels. This suggests that the correlation accuracy is useful for bit-widths larger than 10, and certainly should be sufficient with 18-bit coefficients.

6. RELATED WORK

Previous work on random number generation in FPGAs has focused on the univariate building blocks, particularly the uniform distribution [George and Alfke 2001; Shackleford et al. 2002; Thomas and Luk 2005], which provides underlying randomness to applications, and the Gaussian distribution [Xilinx, Inc. 2002; Lee et al. 2006, 2005]. Recent work provides methods for generating arbitrary continuous univariate distributions [Thomas and Luk 2006a], and also provides efficient methods for generating the Gaussian distribution. This article builds on this work, motivated by the ease with which high speeds can be achieved with various fast methods for generating univariate distributions.

The multivariate generation architecture used here has been described in earlier work [Thomas and Luk 2007], but this article incorporates many additional operational details. In addition, Section 5 demonstrates that a fixed point generator does not suffer from accuracy problems, and is able to produce both Gaussian marginal distributions and a correlation structure of high quality.

The core of the multivariate Gaussian generator is the dense matrix-vector multiply, and a key insight is the observation that when the matrix remains constant for large numbers of vectors, the parallel multipliers and RAMs make this very efficient. If the matrix changes frequently then the method becomes inefficient compared to software, as the bottleneck becomes the speed at which the coefficients of the matrix can be retrieved from an external source. However, if the matrices are sparse, the FPGA can again compete with software by using custom logic for decoding the matrix structure [deLorimier and DeHon 2005]. Dense matrix-matrix multiplications provide many opportunities for caching and reuse, so block RAMs can be used to increase effective memory bandwidth. For this reason previous work on linear algebra in FPGAs has



(a) Global, 4-bit. (b) Per-row, 4-bit.

(c) Global, 6-bit. (d) Per-row, 6-bit

(e) Global, 18-bit. (f) Per-row, 18-bit

Fig. 12. Visualization of p-values for significance of difference between empirical correlation observed over 2^{14} vectors, and the target correlation matrix. The lower triangle shows p-values below 10^{-4}, 10^{-3}, and 10^{-2} as black, dark gray, and light gray, while the upper triangle translates p-values directly to grayscale.

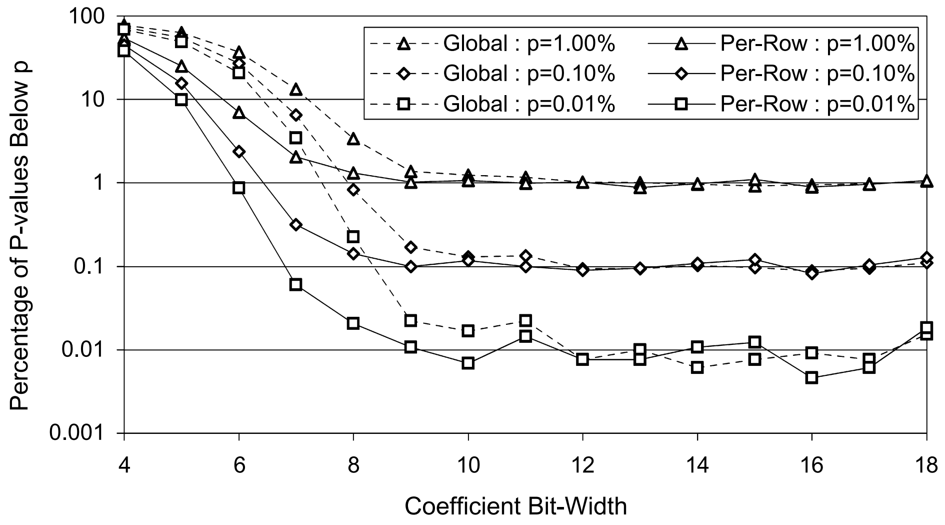


Fig. 13. Distribution of p-values for difference between target and empirical correlation coefficients for a 512 × 512 correlation matrix.

focused on dense matrix-matrix operations, often aiming to provide a drop-in accelerator for BLAS [Jang et al. 2005; Zhuo and Prasanna 2005].

7. CONCLUSION

This article describes an architecture for generating multivariate Gaussian random numbers, and shows how it can be implemented in advanced reconfigurable devices. Our main achievement is to organize the on-chip multipliers and local memories efficiently for dense matrix-vector multiplication, when the matrix remains constant for many different vectors. A particularly efficient mapping for Virtex-4 DSP48 blocks is then demonstrated, which can provide a speedup of 200 times over an equivalent software vector generator. Empirical and theoretical tests are applied to the fixed-precision architecture, showing that the quality and correlation structure of the generated vectors matches the target distribution.

This vector generator is then demonstrated in a case study for simulating Value-at-Risk, a common financial application. The application is implemented using an RC2000 device containing an xc4vsx55-10 part, and through manual placement a design capable of running at 400MHz with a maximum portfolio size of 448 assets is achieved. The hardware accelerated simulator provides a practical speed-up of 26 times over a quad Opteron workstation, reducing the time taken to simulate a 448 asset portfolio from 30 seconds down to 1.1 seconds.

The results of this article demonstrate two benefits of using reconfigurable logic for compute-intensive applications, such as computational finance. First, a single FPGA is able to replace the equivalent of 26 quad-core computers in a cluster, 104 CPU cores in total. This means that much of the heat, power, space, and capital outlay required for a 32U rack containing 26 blades could in principle be replaced with a single computer containing one FPGA. Obviously not all applications will see this level of improvement, but other financial simulations are likely to see similar improvements.

The second key benefit is the reduction in latency that an FPGA accelerated solution can provide. Figure 9 shows computational latency dropping from 30 seconds to 1 second when comparing a quad core computer to an FPGA. This is a great advantage in time-sensitive applications such as trader-support; a potential position may only be open for a few seconds, so it is important to reduce the latency observed by the user to the bare minimum.

It is of course possible to use a networked cluster of CPUs, but this introduces the problem of distributing and scheduling tasks across multiple nodes. Such clusters are also too big and expensive to dedicate one to each user; clusters are shared resources, and in periods of peak demand users will actually find latency increasing. An FPGA accelerator can be installed in every users computer, providing a dedicated computational resource with guaranteed latency and availability.

Current and future work involves integrating complex pricing operators such as Black-Scholes option valuation [Black and Scholes 1973], exploring more complicated types of correlations and probability distributions, and examining the intermediate levels of parallelism between fully parallel and fully serial generators.

REFERENCES

ADVANCED MICRO DEVICES 2006. *AMD Core Math Library (ACML)*. Advanced Micro Devices.

ALTERA CORPORATION 2005. *Stratix II Device Handbook, Volume 1*. Altera Corporation.

ANDERSON, T. W. AND DARLING, D. A. 1954. A test of goodness of fit. *J. Amer. Statist. Asso. 49,* 268, 765–769.

BARR, D. R. AND SLEZAK, N. L. 1972. A comparison of multivariate normal generators. *Comm. ACM 15,* 12, 1048–1049.

BIRNBAUM, A. 1954. Combining independent tests of significance. *J. Amer. Statist. Asso. 49,* 267, 559–574.

BLACK, F. AND SCHOLES, M. S. 1973. The pricing of options and corporate liabilities. *J. Polit. Econo. 81*, 637–659.

DELORIMIER, M. AND DEHON, A. 2005. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM Press, 75–85.

GEORGE, M. AND ALFKE, P. 2001. Linear feedback shift registers in Virtex devices. Tech. rep., Xilinx, Inc.

JANG, J.-W., CHOI, S. B., AND PRASANNA, V. K. 2005. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Trans. VLSI Syst. 13,* 11, 1305–1319.

L'ECUYER, P. AND SIMARD, R. 2007. TestU01 random number test suite. www.iro.umontreal.ca/ simardr/indexe.html.

LEE, D., VILLASENOR, J., LUK, W., AND LEONG, P. 2006. A hardware Gaussian noise generator using the box-muller method and its error analysis. *IEEE Trans. Comput. 55,* 6, 659–671.

LEE, D.-U., LUK, W., VILLASENOR, J. D., AND CHEUNG, P. Y. 2004. A Gaussian noise generator for hardware-based simulations. *IEEE Trans. Comput. 53,* 12, 1523–1534.

LEE, D.-U., LUK, W., VILLASENOR, J. D., ZHANG, G., AND LEONG, P. H. 2005. A hardware Gaussian noise generator using the wallace method. *IEEE Trans. VLSI Syst. 13,* 8, 911–920.

MARSAGLIA, G. 1999. Random numbers for C: The END? Posted to sci.crypt.random-numbers on 20th January 1999.

MARSAGLIA, G. 2004. Evaluating the normal distribution. *J. Statist. Softw. 11,* 4, 1–11.

MARSAGLIA, G. AND MARSAGLIA, J. 2004. Evaluating the Anderson-Darling distribution. *J. Statist. Softw. 9,* 2, 1–5.

PEARSON, K. 1900. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can reasonably supposed to have arisen from random sampling. *Philosoph. Maga., Series 5 50*, 157–175.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1997. *Numerical Recipes in C*, 2nd Ed. Cambridge University Press.

SHACKLEFORD, B., TANAKA, M., CARTER, R. J., AND SNIDER, G. 2002. FPGA implementation of neighborhood-of-four cellular automata random number generators. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM Press, New York, USA, 106–112.

THOMAS, D. B. AND LUK, W. 2005. High quality uniform random number generation through LUT optimised linear recurrences. In *Proceedings of the International Conference on Field-Programmable Technology*. IEEE Computer Society.

THOMAS, D. B. AND LUK, W. 2006a. Efficient hardware generation of random variates with arbitrary distributions. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 57–66.

THOMAS, D. B. AND LUK, W. 2006b. Non-uniform random number generation through piecewise linear approximations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–6.

THOMAS, D. B. AND LUK, W. 2007. Sampling from the multivariate Gaussian distribution using reconfigurable hardware. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 3–12.

XILINX, INC. 2000. Virtex-II platform FPGAs: Complete data sheet. Xilinx, Inc.

XILINX, INC. 2002. Additive white Gaussian noise (AWGN) core. Tech. rep., Xilinx, Inc.

XILINX, INC. 2005. XtremeDSP for Virtex-4 FPGAs User Guide. Xilinx, Inc.

XILINX, INC. 2006. Virtex-4—Why are the FIFO16 flags not working correctly? Tech. rep. AR22462.

ZHANG, G. L., LEONG, P. H., LEE, D.-U., VILLASENOR, J. D., CHEUNG, R. C., AND LUK, W. 2005. Ziggurat-based hardware Gaussian random number generator. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE Computer Society Press, 275–280.

ZHUO, L. AND PRASANNA, V. K. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM Press, 63–74.

Received August 2007; revised February 2008; accepted March 2008