Sampling from the Multivariate Gaussian Distribution using Reconfigurable Hardware

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

Abstract

The multivariate Gaussian distribution models random processes as vectors of Gaussian samples with a fixed correlation matrix. Such distributions are useful for modelling real-world multivariate time-series such as equity returns, where the returns for businesses in the same sector are likely to be correlated. Generating random samples from such a distribution presents a computational challenge due to the dense matrix-vector multiplication needed to introduce the required correlations. This paper proposes a hardware architecture for generating random vectors, utilising the embedded block RAMs and multipliers found in contemporary FPGAs. The approach generates a new n dimensional random vector every n clock cycles, and has a raw generation rate over 200 times that of a single Opteron 2.2GHz using an optimised BLAS package for linear algebra computation. The generation architecture is an ideal source for both software simulations connected via high bandwidth connection, and for completely FPGA based simulations. Practical performance is explored in a case study in Delta-Gamma Value-at-Risk, where a standalone Virtex-4 xc4vsx55 solution at 400 MHz is 33 times faster than a quad Opteron 2.2GHz SMP. The FPGA solution also scales well for larger problem sizes, allowing larger portfolios to be simulation.

1. Introduction

The multivariate Gaussian distribution is a key component of many simulations, as it allows correlations between different random factors to be captured. For example, a multivariate Gaussian distribution can be used to model the correlation between changes in the FTSE and NASDAQ indices, or to model relationships between outside temperature and demand for frozen food. A particular benefit is the ability to capture complex sets of correlations between large numbers of random factors, for example those within a complex portfolio containing tens or hundreds of assets.

However, generating random samples from the multivariate Gaussian distribution is computationally demanding, as it relies on a matrix-vector multiply to capture the correlations. Traditionally this $O(n^2)$ growth with vector size has been addressed using large clusters, as seen in the large compute farms used in banks to calculate overnight Value-at-Risk. This paper offers an alternative, showing that reconfigurable hardware can be used as an effective means of accelerating multivariate Gaussian random vectors, and that complete FPGA based simulations can be built up around this core component.

The contributions of this paper are:

- 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.
- 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 2.2GHz Opteron.
- A case study implementing a Delta-Gamma Value-at-Risk simulation in an xc4vsx55 part on the RC2000 platform, demonstrating a 132 times speed-up over a single Athlon 2.2GHz, and reducing simulation time from 37 to 1.1 seconds when simulating a 448 asset portfolio.

2. Algorithm and Analysis

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

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

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:

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

The vector **m** contains the mean of each component in the vector, i.e. $E(X_N)$. The matrix **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 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 have a significant correlation as they are in similar markets, but the correlation of Microsoft and British Petroleum stock returns would be lower.

To generate random samples from X_N it is necessary to form a vector **r** of independent univariate Gaussian samples from some infinite source $r_1, r_2, \ldots$. The desired correlation structure is then applied to **r** by multiplication with a lower triangular matrix **A**, where $\mathbf{S} = \mathbf{A}\mathbf{A}^T$. The means of the components are then adjusted by adding the vector **m**. Thus the generation of the k-th vector $\mathbf{x}_k$ is calculated as follows [3]:

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

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

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:

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

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

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

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

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.

For the purposes of the analysis it will be assumed that the input width of individual multipliers provides sufficient precision. In practice the 18-bit multipliers in modern FPGA architectures provide ample precision, but we do not offer rigorous justification of this assumption in this paper.

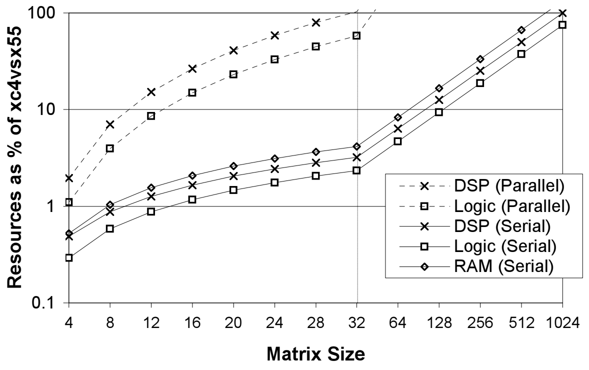


Figure 1. Estimate resource usage for parallel and serial vector generation.

Equation 4 is essentially a matrix-vector multiply and add, where **A** is lower-triangular. A lower triangular matrix has at most $N(N+1)/2$ non-zero 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 1 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

	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)	-	$w_c(N+1)/2/p_r/w_r$

Table 1. FPGA resource usage for parallel and serial vector generation.

held totally in one level then they are all held in the next level. The CPU is assumed 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 and an xc4vsx55. The predicted software performance uses the following figures for the cache and floating-point performance: $l_1 = 2^{14}$; $b_1 = 3.52 \times 10^{10}$; $m = 8.8 \times 10^9$; $l_2 = 2^{18}$; $b_2 = 5.75 \times 10^9$; $l_3 = \infty$; $b_3 = 3.2 \times 10^9$. 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.

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.

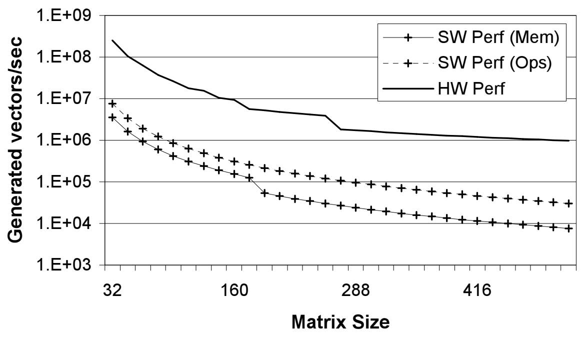


Figure 2. Comparison of expected performance of xc4vsx55 and Opteron 2.2GHz.

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. A platform-independent architecture is presented, but particular attention is paid to the Virtex-4 architecture, and how 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 **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

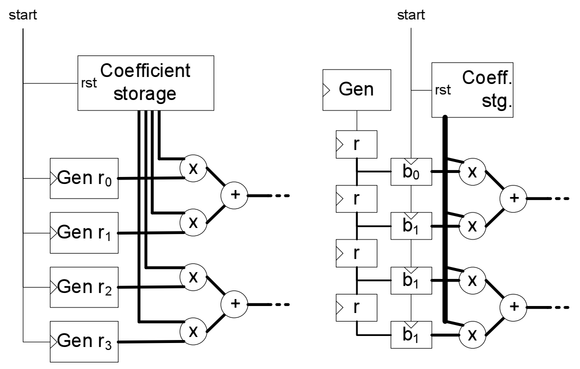


Figure 3. Generation and distribution of univariate Gaussian samples.

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 [9, 7, 10].

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 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 only requiring 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 organised 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 they are only used at a single multiplier. Short horizontal connections at the top and bottom can then be used to create a single global shift-chain.

The exact method used to generate the univariate samples is relatively unimportant, at least in terms of resource usage, as only one instance is needed. The generator used in this paper employs piecewise linear approximations [14], as this method does not require any multipliers and is very easy to pipeline.

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.

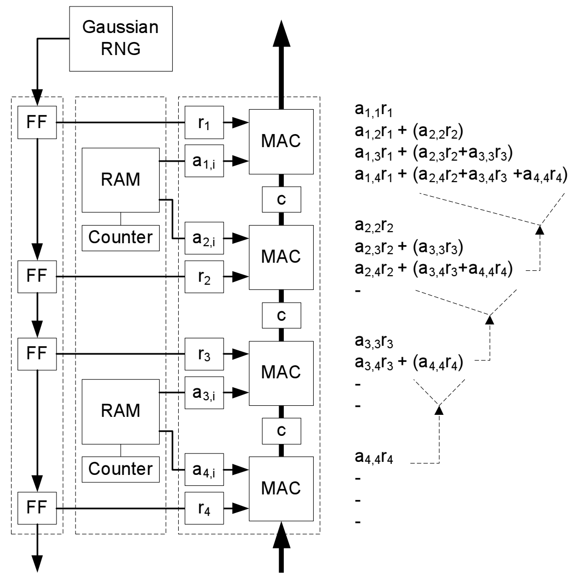
In this work each Virtex-4 block RAM supplies two coefficients to two multipliers, with the RAM organised as a 1024 by 18 RAM. The two sets of coefficients are packed into the top and bottom halves of the RAM, and addressed using a single counter, with the most-significant bit set to 0 for one port, 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 optimised. A very convenient and efficient method of updating coefficients is to re-use 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 [16]) then no further processing is per-



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

Figure 4. Architecture of MAC based multivariate Gaussian generator.

formed at this stage. However, modern FPGA architectures provide more complex DSP blocks, fusing multipliers and wide adders together, such as the Stratix-II DSP [2] and Virtex-4 DSP48 [17] 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 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.

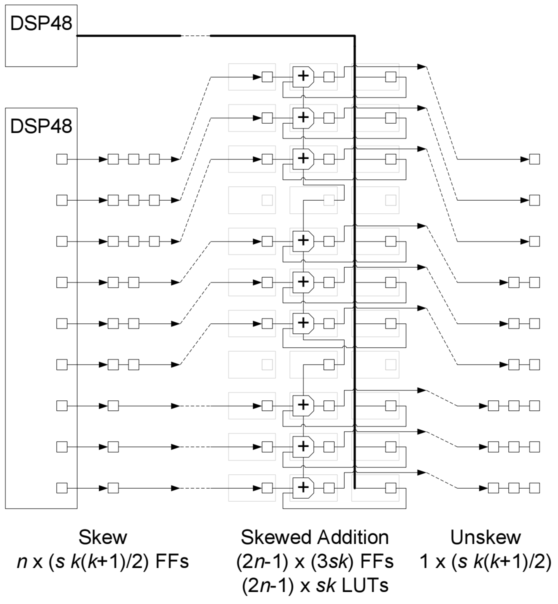


Figure 5. Pipelined adder tree.

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 $n+1$ trees 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:

$$\frac{1}{2}(s(n(k+7)+k-5)+2n-2)k \qquad (13)$$

In the xc4vsx55 architecture the value $s = 9$ is chosen, requiring $k = 5$ segments. When all DSP48 columns are used for vector generation $n = 8$, so 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.

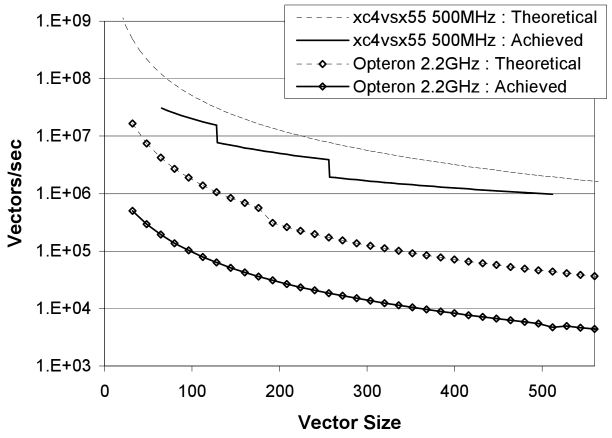


Figure 6. Performance comparison between software and hardware vector generators, both according to the performance model and realised performance.

3.5. Implementation and Evaluation

The generator described above 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 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 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 optimisation of primitives is disabled during synthesis, as XST attempted to optimise registers into SRL16s, without realising 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 realised 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 realised curves for an Opteron 2.2GHz. The realised software performance is measured using the ACML BLAS [1], a set of vector and matrix libraries specifically optimised for AMD processors, using features such as SIMD and cache management instructions. 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 speed-up over software of 246 is achieved when four instances operate on 128 element vectors, and the minimum speed-up of 110 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 speed-up is 208 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 speed-up 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 modelled as having correlated Gaussian returns. A portfolio over these underlying assets can then be modelled, where the portfolio incorporates both direct positions, as well as options on the underlyings. Profit and loss due to direct positions clearly vary in proportion to changes in the underlying, but profit and loss on options depend on factors such as the strike price and time to expiry of the option. 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 [4].

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

$$p = \sum_{i=1}^{N} s_i + \delta_i x_i + \frac{\gamma_i}{2} x_i^2 \qquad (14)$$

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:

$$d = \sum_{i=1}^{N} x_i (\delta_i + \frac{\gamma_i}{2} x_i) \qquad (15)$$

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 in static floating-point, with mantissa scaling pre-calculated per asset. 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.

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:

$$d = \sum_{i=1}^{N} 2^{-k_i} \left(\left[2^{-j_i} x_i \right] \left[\delta'_i + \gamma'_i \left(2^{-j_i} x_i \right) \right] \right) \qquad (16)$$

where **j** is a vector of integers that determine the initial scaling of **r**, **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 [15], 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

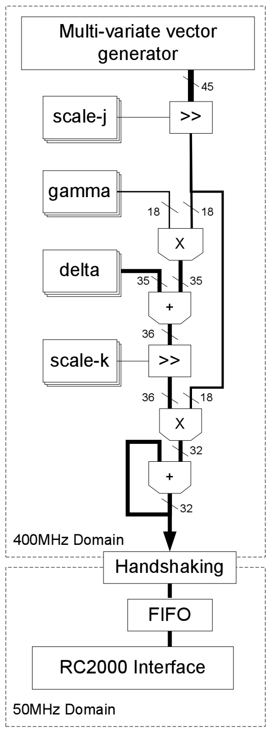


Figure 7. Delta-Gamma approximation pipeline.

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.

Table 2 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

	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 2. Resource usage for Delta-Gamma simulator, with percentage of xc4vsx55 resources shown in brackets.

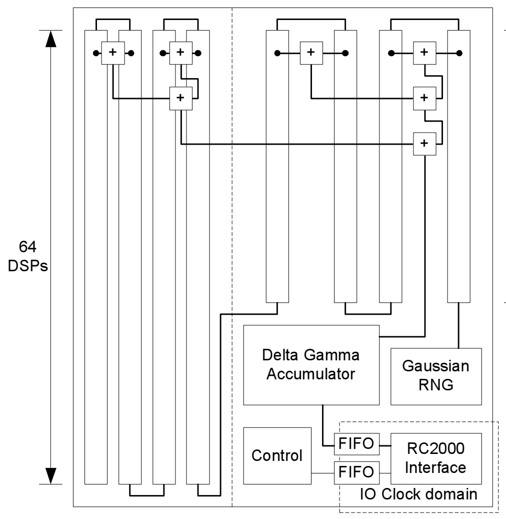


Figure 8. Layout of components in xc4vsx55 based RC2000.

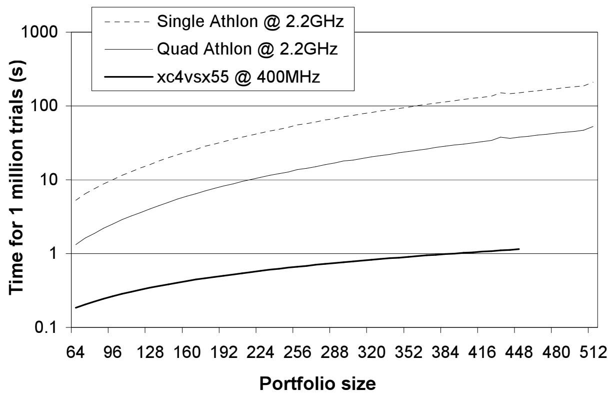


Figure 9. Time taken to perform 1 million portfolio evaluations.

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. The hardware execution time includes the time taken to load the correlation matrix (24ms). For smaller portfolios the hardware is less than 10 times faster than the Quad Athlon, 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 33 times the speed of a quad Athlon, or 132 times faster than a single Athlon, at less than a fifth of the clock rate.

5. Related Work

Previous work on random number generation in FPGAs has focussed on the univariate building blocks, particularly the uniform distribution [6, 11, 12], which provides underlying randomness to applications, and the Gaussian distribution [7, 9, 10]. Recent work provides methods for generating arbitrary continuous univariate distributions [13], and also provides efficient methods for generating the Gaussian distribution. This paper builds on this work, motivated by the ease with which high-speeds can be achieved with various fast methods for generating univariate distributions.

The core of the multivariate Gaussian generator is the dense matrix-vector multiply, and a key contribution of this paper 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 [5]. Dense matrix-matrix multiplications provide many opportunities for caching and re-use, so block RAMs can be used to increase effective memory bandwidth. For this reason previous work on linear algebra in FPGAs has focussed on dense matrix-matrix operations, often aiming to provide a drop-in accelerator for BLAS [8, 18].

6. Conclusion

This paper has presented an architecture for the generation of multivariate Gaussian random numbers in modern reconfigurable architectures. The key insight is in showing how the large number of multipliers and local memories can be organised to efficiently perform a dense matrix-vector multiply, as long as the matrix remains constant for many different vectors. A particularly efficient mapping for Virtex-4 DSP48 blocks is then demonstrated, which can provide a speed-up of 200 times over an equivalent software vector generator.

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 33 times over a quad Athlon workstation, reducing the time taken to simulate a 448 asset portfolio from 37 seconds down to 1.1 seconds.

The results of this paper 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 33 quad-core computers in a cluster, 132 CPU cores in total. This means that all the heat, power, space and capital outlay required for a 32U rack 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 37 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.

Future work will focus on integrating more complicated pricing operators such as Black-Scholes option valuation [4], exploring more complicated types of correlations and probability distributions, and examining the intermediate levels of parallelism between fully parallel and fully serial generators.

References

[1] Advanced Micro Devices. *AMD Core Math Library (ACML)*, 2006.

[2] Altera Corporation. *Stratix II Device Handbook, Volume 1*, 2005.

[3] D. R. Barr and N. L. Slezak. A comparison of multivariate normal generators. *Commun. ACM*, 15(12):1048–1049, 1972.

[4] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economics*, 81:637–659, 1973.

[5] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–85. ACM Press, 2005.

[6] M. George and P. Alfke. Linear feedback shift registers in Virtex devices. Technical report, Xilinx Inc., 2001.

[7] X. Inc. Additive white gaussian noise (AWGN) core. CoreGen documentation file, 2002.

[8] J.-W. Jang, S. B. Choi, and V. K. Prasanna. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Transactions on VLSI Systems*, 13(11):1305–1319, 2005.

[9] D.-U. Lee, W. Luk, J. D. Villasenor, and P. Y. Cheung. A gaussian noise generator for hardware-based simulations. *IEEE Transactions On Computers*, 53(12):1523–1534, december 2004.

[10] D.-U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. Leong. A hardware gaussian noise generator using the wallace method. *IEEE Transactions on VLSI Systems*, 13(8):911–920, 2005.

[11] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 106–112, New York, USA, 2002. ACM Press.

[12] D. B. Thomas and W. Luk. High quality uniform random number generation through lut optimised linear recurrences. In *International Conference on Field-Programmable Technology*. IEEE Computer Society, 2005.

[13] D. B. Thomas and W. Luk. Efficient hardware generation of random variates with arbitrary distributions. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 2006.

[14] D. B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. In *International Conference on Field Programmable Logic and Applications*, 2006.

[15] Virtex-4 - Why are the FIFO16 flags not working correctly? Technical Report AR22462, Xilinx Inc., 2006.

[16] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, 2000.

[17] Xilinx, Inc. *XtremeDSP for Virtex-4 FPGAs User Guide*, 2005.

[18] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–74. ACM Press, 2005.