SAMPLING FROM THE EXPONENTIAL DISTRIBUTION USING INDEPENDENT BERNOULLI VARIATES

David B. Thomas and Wayne Luk

Department of Computing
Imperial College
London, England
{dt10,wl}@doc.ic.ac.uk

ABSTRACT

The exponential distribution is a key distribution in many event-driven Monte-Carlo simulations, where it is used to model the time between random events in the system. This paper shows that each bit of a fixed-point exponential random variate is an independent Bernoulli variate, allowing the bits to be generated in parallel. This parallelism is of little interest in software, but is particularly well suited to FPGA generators, where huge numbers of independent uniform bits can be cheaply generated per cycle. Two generation architectures are developed using this approach, one using only logic elements to generate individual bits, and another using block-RAMs to group multiple bits together. The two methods are evaluated at three different quality-resource trade-offs, and when compared to existing methods have both higher performance and better resource utilisation. The method is particularly useful for very high performance applications, as extremely high-quality 36-bit exponential variates can be generated at 600MHz in the Virtex-4 architecture, using just 880 slices and no block-RAMs or embedded DSP blocks.

1. INTRODUCTION

The exponential distribution is one of the "big three" distributions, along with the uniform and normal distributions. All three distributions are maximum entropy distributions for a simple set of conditions, which means that their entropy is at least as great as any other distribution meeting those conditions. In the case of the exponential distribution, the two conditions are that the distribution is non-negative, i.e. random values lie in the range $[0, \infty)$, and has a fixed mean. [1] These conditions explain one of the most important uses of the distribution, which is to model the time between random events in a Monte-Carlo simulation: the waiting time till the next event must clearly be non-negative, and if the events are assumed to occur at some fixed average rate then the distribution of waiting times will be exponential.

Previous work on FPGA-accelerated Monte-Carlo simulations have typically modelled discrete time systems, and so require generators for the normal distribution. However, the increasing size and floating-point capabilities of FPGAs are allowing larger event-driven Monte-Carlo simulations, using the exponential distribution to model waiting times. One example of this is the modelling of biochemical systems, where chemical reactions within cells are simulated to estimate changes in concentration over time [1]. Another example is in finance, where the time between loan defaults is simulated, allowing the expected value of a portfolio of loans to be estimated at future points in time [2].

The loan-portfolio model provides a good example of the motivation for this paper, as it consumes huge numbers of exponential variates: in one design instance 40 independent high-quality variates are needed every cycle at 233MHz, for an aggregate rate of 9.3 GSamples/sec. The generation method presented here allows these quality and performance limits to be met, while also limiting the total amount of resources that must be dedicated to random number generation. The key contributions of this method are:

- An analysis of the exponential distribution, showing that the individual bits of a fixed-point exponential variate are completely independent, allowing the distribution to be seen as the concatenation of independent Bernoulli variates (random bits).
- A bit-wise generation architecture that generates Bernoulli bits independently, allowing extremely high generation speeds using only LUT and FF resources.
- A table-based generation architecture that uses block RAMs to generate groups of Bernoulli bits, trading off logic resources for RAM resources, while still using

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

[1] The corresponding conditions for the uniform and normal are respectively: distributions with a finite range; and distributions with fixed mean and variance.

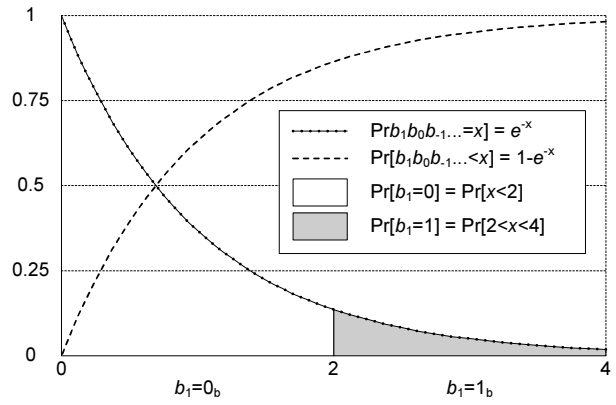

Fig. 1. PDF and CDF of the exponential distribution.

no multipliers.

- An evaluation of the two architectures when described using platform independent Handel-C code, showing that in the Virtex-4 family the bit-wise method can generate high-quality 36-bit fixed-point random numbers in excess of 600MHz, using just 881 slices, while the table-based method operates at 375MHz and uses 397 slices and four block RAMs.

2. ALGORITHM

The exponential distribution E has the PDF (probability density function) and CDF (cumulative distribution function):

$$\Pr[x = E] = e^{-x} \qquad \Pr[x < E] = 1 - e^{-x} \tag{1}$$

Figure 1 shows the PDF and CDF in the range $[0, 4]$.

In this paper we generate numbers in a fixed point unsigned binary format, with a most-significant bit of 2^s and least-significant bit of 2^{-f}, for a total width of $w = s+f+1$:

$$x = b_s b_{s-1} ... b_1 b_0 . b_{-1} b_{-2} b_{-3} ... b_{-f} \tag{2}$$

In practice s must be finite, so we will be sampling from the truncated exponential distribution, limited to $[0, 2^{s+1})$. Truncating to this range removes some of the PDF's area, so one choice that must be made is how much area can be safely removed. For four integer bits, the range is $[0, 16)$, and the area lost is $\sim 2^{-23}$, so any application using more than 2^{23} random samples may see problems due to truncation. However, with five integer bits an area of only 2^{-46} is lost, which is sufficient for the vast majority of Monte-Carlo simulations, and for ultra-sensitive long-running simulations, six integer bits lose only 2^{-92}.

The exponential distribution has the interesting property that it is memoryless. Assume that we are waiting for some event, where the number of seconds until the next event follows an exponential distribution E. The memoryless property asserts that if we start waiting at time 0, and reach time k seconds without observing an event, then the expected waiting time still follows the exponential distribution. Put another way, the knowledge that we have already waited for k seconds has no impact on how much longer we still have to wait. This is expressed more formally by describing the conditional distribution:

$$\Pr[E > k + t | E > k] = \Pr[E > t] \tag{3}$$

The memoryless property was the original insight leading to our method for generating exponential variates, as it suggests that the individual bits of the fixed-point may be independent.

The distribution of the first bit b_s is clearly independent of all the following bits, and so must follow a Bernoulli distribution with probability:

$$\Pr[b_s = 0] = \Pr[E < 2^s | E < 2^{s+1}] \tag{4}$$

$$= \frac{\Pr[E < 2^{s+1} | E < 2^s] \Pr[E < 2^s]}{\Pr[E < 2^{s+1}]} \tag{5}$$

$$= 1 - 1/(1 + \exp(2^s)) \tag{6}$$

In Figure 1 the area of the grey segment shows this probability (the white and grey areas together have an area of 1). By itself this is not particularly interesting, as the most significant bit of *any* distribution is independent. However, for most distributions the MSB is the only independent bit, with the following bits having dependencies on all the preceding bits.

If we now wish to generate the second bit, we need to calculate the distributions conditional on the first bit. First, when b_s is zero:

$$\Pr[b_{s-1} = 0 | b_s = 0] = \Pr[E < 2^{s-1} | E < 2^s] \tag{7}$$

$$= 1 - 1/(1 + \exp(2^{s-1})) \tag{8}$$

So when the first bit is zero we see the same formula as we saw in Equation 6 for the first bit.

In the second case, where b_s is one, we have:

$$\Pr[b_{s-1} = 0 | b_s = 1] \tag{9}$$

$$= \Pr[E < 2^s + 2^{s-1} | 2^s < E < 2^{s+1}] \tag{10}$$

$$= \frac{\Pr[2^s < E < 2^{s+1} | E < 2^s + 2^{s-1}] \Pr[E < 2^s + 2^{s-1}]}{\Pr[2^s < E < 2^{s+1}]} \tag{11}$$

$$= 1 - 1/(1 + \exp(2^{s-1})) \tag{12}$$

The full working is elided for reasons of space, but can be easily if laboriously proved using basic probability transforms and Bayes' theorem.

We now have the situation that:

$$\Pr[b_{s-1} = 0 | b_s = 1] = \Pr[b_{s-1} = 0 | b_s = 0] \tag{13}$$

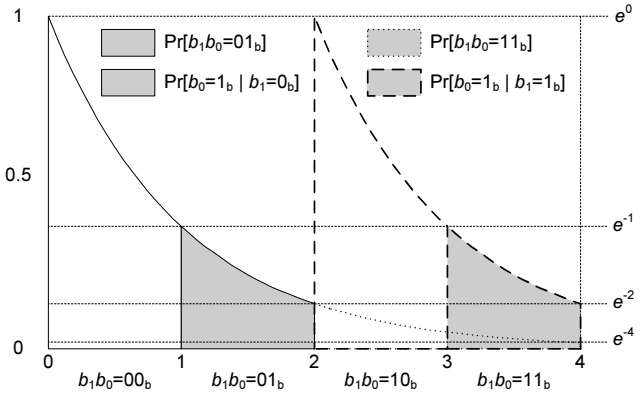

Fig. 2. Graphical demonstration of the independence of the most significant bit and the following bit.

so the second bit follows a Bernoulli distribution and *is completely independent of the first bit*. This is shown graphically in Figure 2, where the fine dotted line shows the actual PDF, while the heavy dashed line shows the PDF conditional on the first bit being one. The conditional PDF is just vertically stretched (or a horizontally translated version of the conditional PDF on the left side, where the first bit is zero), so the ratio of grey to white is the same in both cases.

It is easy to see that this property actually extends down through all the bits, and *every bit of the fixed-point exponential distribution is a completely independent Bernoulli variate*. So we can construct a w-bit fixed-point sample from the (truncated) exponential distribution using w completely independent Bernoulli variates.

We cannot claim to be the first to discover this property of the exponential distribution, originally discovered by Chatterji [3], then rediscovered by Marsaglia [4]. In both cases, it was seen as a mathematical curiosity, stemming from the question "What continuous distributions arise from the concatenation of independent binary bits?", rather than an attempt to decompose the exponential distribution. However, we do appear to be the first to realise that this property has a practical application when generating exponential random numbers using hardware.

From a software point of view, generating independent random bits makes no sense as a basis for generating random numbers, due to the large number of Bernoulli variates required to construct just one exponential variate of useful width. Uniform random numbers are very expensive in software, so even a direct inversion method is more efficient, and more advanced methods such as the Ziggurat method can be much more efficient.

However, in an FPGA the cost of operations is very different to that of software. First, uniform bit generation becomes extremely cheap, while multiplication becomes more expensive (particularly for wide fixed-point numbers). Second, methods that trade off a fast and frequent execution path against a slower infrequent path are less attractive. In hardware both paths must be included in the circuit, and if one of the paths is under-utilised then efficiency decreases. Third, methods that do not guarantee a sample every cycle (such as the Ziggurat method) present far more difficulties in hardware than they do in software, requiring additional resources to deal with pipeline stalls, buffering, and/or multiple clock domains.

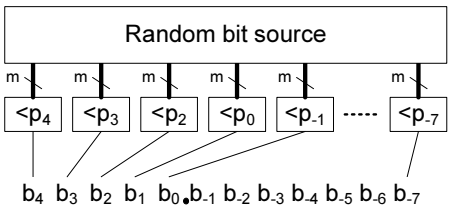

Fig. 3. Bitwise architecture

We now describe two ways in which the independent bits of the exponential distribution can be used as a basis for generating exponential random numbers. First a direct bit-wise architecture is considered, using LUTs and FFs to construct extremely fast pipelined generators. A less resource intensive table-based architecture is then proposed, which generates groups of Bernoulli variates. Finally, the performance and efficiency of these methods is considered, and comparisons made with existing random number generators.

3. BIT-WISE ARCHITECTURE

The most direct implementation is to construct each bit separately, using an independent Bernoulli generator. Each output bit requires a uniform variate and a constant comparator:

$$b_i = U_i < p_i, \quad s \geq i \geq f \tag{14}$$

$$p_i = 1/(1 + \exp(2^i)) \tag{15}$$

Each U_i must be independent, and the resolution of U_i determines how accurate the Bernoulli variate (and the overall exponential variate) is. If we assume U_i is a fixed point integer with a width of m_i bits, and choose a correspondingly truncated $\hat{p}_i$, then the error $|\hat{p}_i - p_i|$ will be at most 2^{-m_i}. The choice of m_i presents a quality-resource trade-off, as increasing m_i will increase the accuracy of the PDF, but will also increase the total area required. This trade-off is explored empirically in Section 5, but in this paper the quality issue is not approached analytically. We also assume that the same fractional precision m is used for all bits.

Figure 3 shows a high-level view of the generator, which concatenates together independent Bernoulli bits to create an exponential fixed-point random number. Generating each Bernoulli bit requires two stages. First, we must generate m random uniform bits, which can be cheaply and efficiently

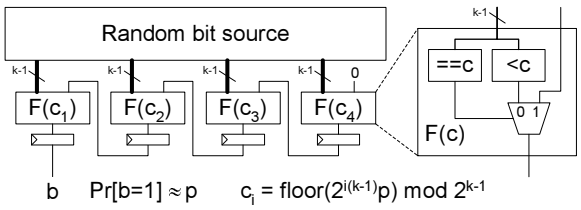

Fig. 4. Pipelined Bernoulli generator.

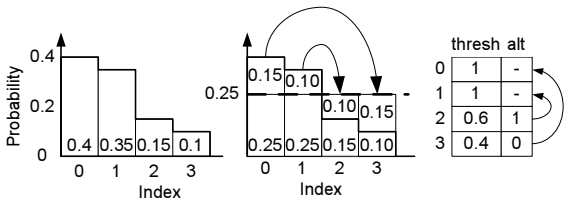

Fig. 5. Building an alias table for a discrete distribution.

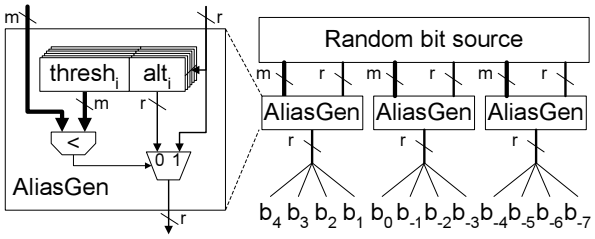

Fig. 6. Alias-table generation architecture.

generated in m LUTs using a LUT-optimised linear recurrence [5]. Second, we need to compare these uniform bits with the m bit constant $\hat{p}_i$. A standard carry-chain based implementation would require m LUTs for the compare, but this uses more resources than necessary, and for large values of m the carry propagation delay would seriously limit speed. Instead we can implement the comparison using $\lceil m/(k-1) \rceil$ k-LUTs, with a critical path of one LUT, by noting that the constant comparison can be completely pipelined (Figure 4).

4. ALIAS-TABLE ARCHITECTURE

The bit-wise implementation is simple, direct, and promises very high speed, but it does require a large number of logic-elements. However, in certain designs there may be very few spare logic-elements, for example if the target device is optimised for DSP applications and emphasises multipliers and RAMs over general logic. In this section we show how the Bernoulli variates can be grouped together, so that logic-elements can be swapped for RAMs.

The bit-wise implementation treats each bit as a single variate, using the discrete Bernoulli distribution. However, we can take a pair of consecutive bits, $b_{i+1}b_i$, and treat them as a four-valued discrete distribution over the integers $0..3$:

$$\Pr[b_{i+1}b_i = 00_b] = q_{i+1}q_i \quad \Pr[b_{i+1}b_i = 01_b] = q_{i+1}p_i$$
$$\Pr[b_{i+1}b_i = 10_b] = p_{i+1}q_i \quad \Pr[b_{i+1}b_i = 11_b] = p_{i+1}p_i$$

where $q_i = 1 - p_i$. This grouping can be extended to the general case of r bits, and if we can generate this 2^r valued discrete distribution, then the w bit wide output can be split into $\lceil w/r \rceil$ independent segments. Fortunately Walker's alias method [6] provides just such a mechanism for sampling from discrete distributions, which uses a table with 2^r entries to allow sample generation in constant time.

Figure 5 shows graphically how the alias method works. On the left is shown a discrete distribution with four output values, each of which has a different probability. We can't directly sample from this distribution, but we can select between four equal probability values, by using two uniform random bits as the index. As there are four values, an equal probability selection will assign each value a probability of 0.25, shown as the horizontal dashed line in the middle graph. The original histogram is sometimes above this line and sometimes below, so we take excess portions above the line, and place them in empty areas below the line. It is always possible to exactly pack the excess parts into the empty areas, and each empty area never needs to be packed with area from more than one column (though a large excess area may be split into multiple empty areas).

The alias sampling algorithm first generates a random uniform index, for example, 1. All of the area below the dashed line in column 1 actually belongs to column 1, so the generated variate will be 1. However, if the random index is 2, then some of the area below the dashed line belongs to column 2, but some is actually excess area from column 1. We need to randomly choose which section we are in, according to the relative probabilities. The lower area belonging to column 2 has probability 0.15, while the upper area belonging to column 1 has probability 0.1, so we should return value 2 with probability 0.6 (and 1 with probability 0.4).

The expanded form of the table is shown on the right, which represents the data needed at run-time. It should be noted that the alias method itself is exact, and it is *always* possible to construct a table for a discrete distribution. However, in practice the accuracy is limited by the precision with which the threshold is stored.

Figure 6 shows how the alias method is implemented in hardware, with the left hand side showing the internal structure of an alias table generator. The two inputs to each table are just uniform random bits, with the first group treated as an m bit wide fixed-point random number in the range $[0, 1)$, and the second as an r bit random index in the range $0..2^r - 1$. The total resource usage of each alias generator is: a $2^r \times (m + r)$ ROM containing pairs of thresholds and alternate indices; an m-bit wide comparator; and an r-bit wide multiplexor.

Parametrisation	High	Medium	Low
Output format	5.31	5.22	4.14
Threshold width	36	32	27
Output width	36	27	18
Output range	[0..32)	[0..32)	[0..16)

Table 1. Generator parametrisations used in the evaluation.

5. EVALUATION

The overall approach to exponential random number generation presented in this paper is very flexible, allowing per-bit accuracy, output range, and output resolution to be freely modified. However, for the purposes of evaluating quality, performance, and resource usage, we must pick specific parametrisations. The three variants shown in Table 1 attempt to capture three broad usage classes: "Low", where the distribution must only be "exponential-ish"; "Medium", appropriate for most applications; and "High", which can be used in the most demanding simulations.

We have developed a tool that automatically generates exponential generators, using either the bit or table methods, and allowing parameters such as threshold width (per-bit accuracy) and output width (fixed-point resolution) to be varied. The tool produces platform independent Handel-C descriptions, using only standard language features, and no FPGA-specific primitives. The Handel-C description can then be synthesised for any target architectures supported by the Handel-C compiler, either by using the built-in Handel-C synthesis route, or by compiling to VHDL or Verilog.

In this paper we used the DK5.1 (Handel-C) compiler, using the direct Handel-C to EDIF synthesis route for table-based generators, using all default settings with the addition of technology mapping ("-lutpack"); Unfortunately the Handel-C compiler was not able to detect that the pipelined Bernoulli stages could fit in a single LUT (see Figure 4), and used carry-chain based comparators, which wasted LUTs and reduced performance. Instead synthesis via Xilinx XST was used for bit-based generators, using all default flags with the addition of retiming. The designs were then placed and routed using Xilinx ISE 9.2, using all default settings. At no stage were timing constraints given to either the synthesis or place and route tools, so we rely on the "auto-constraint" mode of the Xilinx tools.

Table 2 shows the resulting area usage and performance of the three variants, in the Virtex-2 and Virtex-4 architectures (the same Handel-C source is used for both). The top group shows results for the bit-wise generators, where the most striking characteristic is the raw speed of the generators. Without any FPGA-specific optimisations, increased tool-chain effort levels, or timing constraints, speeds in excess of 600MHz are achieved in the Virtex-4 architectures (meaning 600M random numbers are generated per second, one per cycle). At present these very high speeds are not strictly necessary, as most processes that consume exponential variates would be limited by the 500MHz limit on the multiplier blocks, but could be of great interest in future architectures containing heavily pipelined hardened floating-point units.

	Virtex-2			Virtex-4		
	Low	Med.	High	Low	Med.	High
	Bit-wise generator					
Slices	455	836	1412	325	555	881
FFs	613	1083	1781	524	893	1422
LUTs	494	849	1327	506	866	1350
MHz	369	369	342	632	661	607
	Table generator					
RAMs	2	3	4	2	3	4
Slices	103	182	266	158	274	397
FFs	152	257	374	188	286	413
LUTs	157	269	394	302	250	370
MHz	264	249	242	404	394	375

Table 2. Resource and performance characteristics of exponential generators with different quality characteristics.

The drawback of the bit-wise generator is the large number of resources that must be used to achieve these speeds. At first glance the resource usage may appear excessive, but these generators use no block-RAMs or DSP blocks: the LUTs and FFs are the only resources consumed. This is ideal for simulations which require all the DSP and RAM resources for the "real" work, such as floating-point maths. The exponential generator can use spare logic resources, and due to it's intrinsically short critical path the placer can spread the generator into the free space left by more timing-sensitive parts of the application.

The resource usage of the table-based generators is much lower than that of the bit-wise generators, but this comes at the expense of both reduced performance, and the use of block-RAMs. However, although the generator's speeds are reduced, they all still match the 400MHz performance of the heavily optimised and Virtex-4 specific single-precision floating-point cores provided by Xilinx in ISE 9.2, *without* any optimisation or platform specific code. It is also often possible to share block RAMs between independent exponential generators, halving the effective RAM usage.

We defer detailed quality testing to a further paper, but have performed some basic validation. In particular, the χ^2 test with 256 equal-probability mass buckets has been applied to all generators, using sample sizes from 2^{10} to 2^{36} Both the Medium and High generators pass the test for the maximum sample size, but the Low generator fails at 2^{23}. This appears to be a consequence of the restricted output range, but more detailed analysis is required.

	Width	Slices	RAM	DSP	MHz
Wallace [7]	24	895	7	4	155
Ziggurat [8]	35	868	4	2	170
Inversion [9]	16	548	2	2	232
Med,Bit	27	836	-	-	369
Med,Table	27	182	3	-	249

Table 3. Comparison between existing methods and bit-wise and alias-table methods.

6. COMPARISON WITH RELATED WORK

The majority of recent work on FPGA-based non-uniform random number generation has focused on the normal distribution, as this is the most important distribution for the stochastic applications first mapped to FPGAs. However, both the Wallace [7] and Ziggurat [8] methods used for the normal distribution can also generate the exponential distribution. In both cases the architecture must be modified to change the output distribution, but the change in computational load and structure is relatively minor. For that reason we make the broad assumption that the performance and resource usage would be similar after the conversion, so they can be used for comparison purposes.

Another research theme has been the approximation of arbitrary distributions, using architectures compiled specifically for the distribution [9]. Methods for the normal, exponential, and log-normal distributions are all considered, but detailed results are only given for the normal distribution. However, the methods used for both distributions are very similar, so again we assume that the resource usage and performance can be used for comparison purposes.

Table 3 compares the resource usage and performance of these existing methods to our approach. The bit-wise method is by far the fastest, and is also even competitive on resource-usage, using less slices than the Ziggurat and Wallace methods, and while using 50% more resources than Inversion, it does not use any RAM or DSP resources.

For applications where resource usage is more important than speed, the table-based method is the best available choice. However, it is still the second fastest method, even though it was not optimised for the Virtex-2 family nor given a speed constraint; by comparison, the Inversion architecture (the third fastest generator), was iteratively optimised for the Virtex-2 family, by adjusting the degree of pipelining to achieve the maximum possible speed.

7. CONCLUSION

This paper has shown how the fixed-point exponential distribution can be generated as the concatenation of independent Bernoulli variates, allowing the calculation to proceed in parallel, using no multipliers or adders. This generation method is ideal for FPGAs, as they can both take advantage of the intrinsic parallelism, and supply huge numbers of high-quality uniform random bits per cycle.

Using a direct bit-wise implementation of the method allows very high-quality and high-resolution random numbers to be generated at 600MHz in the Virtex-4 architectures, while using just 880 slices and no RAM or DSP resources. An alternate implementation using block RAMs is also developed, which generates groups of Bernoulli bits using alias tables, and which reduces the resource logic resource requirements, while still achieving speeds of 375MHz without any platform specific optimisations.

Future work will focus on the question of accuracy: given a target CDF accuracy, what threshold width is needed for each Bernoulli bit? Optimisation of the generators will also be investigated, both in trying to achieve better accuracy with the same number of resources, and in trying to improve performance and resource usage at the same quality level.

8. REFERENCES

[1] M. Yoshimi, Y. Iwaoka, Y. Nishikawa, T. Kojima, Y. Osana, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Yamada, H. Kitano, and H. Amano, "FPGA implementation of a data-driven stochastic biochemical simulator with the next reaction method," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2007, pp. 254–259.

[2] D. B. Thomas and W. Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, 2008.

[3] S. D. Chatterji, "Certain induced measures and the fractional dimensions of their "supports"," *Probability Theory and Related Fields*, vol. 3, no. 3, pp. 184–192, 1964.

[4] G. Marsaglia, "Random variables with independent binary digits," *The Annals of Mathematical Statistics*, vol. 42, no. 6, pp. 1922–1929, 1971.

[5] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," *Journal of VLSI Signal Processing*, vol. 47, no. 1, 2007.

[6] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Trans. Math. Software*, vol. 3, pp. 253–256, 1977.

[7] 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*, vol. 13, no. 8, pp. 911–920, 2005.

[8] G. L. Zhang, P. H. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk, "Ziggurat-based hardware Gaussian random number generator," in *Proc. Int. Conf. on Field Programmable Logic and Applications*. IEEE Computer Society Press, 2005, pp. 275–280.

[9] R. Cheung, D. Lee, W. Luk, and J. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inverson method," *IEEE Transactions on VLSI*, vol. 15, no. 8, pp. 952–962, 2007.