Parallel generation of Gaussian random numbers using the Table-Hadamard transform

David B. Thomas
Dept. of Electrical and Electronic Engineering
Imperial College London
Email: dt10@imperial.ac.uk

***Abstract*—Gaussian Random Number Generators (GRNGs) are an important component in parallel Monte-Carlo simulations using FPGAs, where tens or hundreds of high-quality Gaussian samples must be generated per cycle using very few logic resources. This paper describes the Table-Hadamard generator, which is a GRNG designed to generate multiple streams of random numbers in parallel. It uses discrete table distributions to generate pseudo-Gaussian base samples, then a parallel Hadamard transform to efficiently apply the central limit theorem. When generating 64 output samples the Table-Hadamard requires just 100 slices per generated sample, a quarter the resources of the next best technique, while providing higher statistical quality.**

***Keywords*-Gaussian; Normal; Random Number; FPGA**

I. Introduction

The Gaussian (or normal) distribution is one of the fundamental probability distributions used in stochastic models, either directly, or as a building block for more complicated distributions, so Monte-Carlo (MC) solvers for such models must also generate large numbers of Gaussian random samples. FPGAs are an attractive way of implementing many parallel MC instances, but to achieve maximum performance (i.e. maximum spatial replication) it is necessary for each instance to be very small, so the underlying Gaussian Random Number Generators (GRNGs) must also be small. Existing work has focussed on GRNGs which provide a single stream of samples, using block-RAMs or DSPs to implement the transforms, but such resources are precious, and often needed in the simulation that the GRNG is driving.

This paper presents a GRNG which produces large numbers of independent random samples each cycle, which can then be used to drive many independent MC instances. The GRNG uses Hadamard transforms to turn many low quality pseudo-Gaussian samples into the same number of high quality Gaussian samples using adders and subtractors. The low-quality samples are generated using a small table-lookup, which matches the statistical moments of the Gaussian using a sparse discrete distribution. The Table-Hadamard method has the unique advantage that the more Gaussian samples are needed per cycle, the higher the quality of the outputs becomes, while resource usage only grows with the logarithm of the outputs per cycle.

Our contributions are:

- A new approach to creating GRNGs, which aims to provide many independent streams of Gaussian samples, using the Hadamard transform to allow efficient application of the Central-Limit-Theorem.
- A concrete architecture called the Table-Hadamard GRNG, which uses table distributions to provide many pseudo-Gaussian inputs to the Hadamard transform.
- A method for correcting the table-distribution moments using quintic and heptic transfoms, resulting in an order of magnitude improvement in quality without any extra hardware resources.
- Analysis of the Table-Hadamard properties, showing the the distribution is accurate to high sigma, and can pass the χ^2 empirical test for samples sizes up to 2^{50}.

The Table-Hadamard generator is freely available as part of the open-source FloPoCo project, and includes all code needed to generate GRNGs, as well as implementations of the statistical and theoretical tests which were applied.

II. Background

This section will introduce the mathematical and practical characteristics of the Gaussian distribution, then describe existing techniques for building GRNGs in FPGAs.

A. The Gaussian Distribution

The Gaussian distribution can be defined in many ways, but three common definitions of the Gaussian distribution $G \sim N(\mu, \sigma^2)$ with mean μ and variance σ^2 are:

The Central-Limit Theorem (CLT) : If $B_1, B_2, ..., B_\infty$ is a set of IID variates with finite mean $E(B) = \mu$ and finite variance $E(B^2) = \sigma^2$, then:

$$G = \lim_{n \to \infty} \mu + \frac{1}{\sqrt{n}} \sum_{i=1}^{n} (B_i - \mu)$$

Central Moments : The d'th standardised central moment is $m^{(d)}(X) = E\left[(X - E(X))^d\right]$, and the Gaussian is uniquely defined by the mean $E(X) = \mu$ and the higher order ($d > 1$) moments:

$$m^{(d)}(G) = \begin{cases} \sigma^2, & \text{if } d = 2 \\ 0, & \text{if } d \text{ is odd} \\ (d-1)m^{(d-2)}(G)\sigma^2, & \text{otherwise} \end{cases} \quad (1)$$

Cumulative Distribution Function (CDF) :

$$\Pr[x < G] = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(\frac{v-\mu}{\sqrt{2}\sigma}\right)^2} dv = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

If a distribution satisfies any of these definitions then it is the Gaussian distribution, and so will satisfy all three.

The importance of the Gaussian distribution is largely due to the CLT, which says that the sum of many identically distributed random variables will tend towards the Gaussian distribution. For example, in physics the Brownian motion of particles is the result of many tiny random movements, while in computational finance the day to day changes in prices are the sum of huge numbers of small, random-seeming minute by minute changes. This means that many stochastic models use the Gaussian distribution to capture the aggregate of many tiny random changes within a particular time-span.

Many stochastic models cannot be evaluated analytically, so it is necessary to use Monte-Carlo simulation: the model is simulated millions or billions of times, using different random realisations of the random variates each time, and the solution is approximated using statistical measures over all the different simulations. Monte-Carlo is simple to apply, but the big drawback is execution time, as usually accuracy is proportional to the square root of the number of simulations. Random number generation is often also a bottleneck, particularly when large numbers of samples from non-uniform distributions are needed.

FPGAs are an attractive platform for implementing Monte-Carlo, as due to the embarrassingly parallel nature of the simulations, it is possible to put multiple independent simulators on an FPGA. Exploiting spatial parallelism in this way means that performance, measured in simulations per second, is inversely proportional to area - if each simulator instance can be halved in size, then two times as many simulators can be instantiated, and performance will double. In many simulations GRNGs takes a significant proportion of simulator area, so it is very important to minimise GRNG size, while at the same time ensuring they have sufficient statistical quality over billions of random samples.

B. Existing GRNG Methods

The most direct way of generating Gaussian samples is through the inversion method - if the Gaussian distribution function Φ transforms Gaussian samples into a uniform probability between 0 and 1, then the inverse Gaussian CDF must transform a uniform random number between 0 and 1 into a Gaussian sample:

$$u = \Phi\left(\frac{x-\mu}{\sigma}\right) \leftrightarrow x = \Phi^{-1}(u)\sigma + \mu$$

The function Φ^{-1} is difficult to approximate close to 0 and 1 (the tails of the distribution), but efficient FPGA implementations can use non-uniform segmentation and piecewise linear polynomials to retain accuracy into the tails [1].

Another popular technique for hardware implementation is the Box-Muller transform, which takes two IID samples u_1 and u_2, and turns them into two IID Gaussian samples g_1 and g_2 using the transform:

$$g_1 = \sqrt{-2\ln u_1}\cos(2\pi u_2) \quad g_2 = \sqrt{-2\ln u_1}\sin(2\pi u_2)$$

From an FPGA point of view the focus has been on implementing the various transforms efficiently, attempting to minimise bit-widths resource usage, while maintaining accuracy and statistical quality [2].

The Central-Limit-Theorem has also been used as the basis for GRNGs in FPGAs, with an early approach of simply adding together 128 uniform random bits [3], using the binomial approximation to get a Gaussian distribution reaching out to 6σ. One of the most recent ideas is the CLT+corrector method [4], which adds together n uniform random numbers to get the CLT-n distribution. By itself, this is not close enough to the Gaussian distribution, but it does have the right sort of shape, with long distribution tails. By applying a simple degree-1 polynomial corrector to the CLT-n distribution, the distribution can be stretched into something much closer to Gaussian.

III. The Table-Hadamard RNG

The CLT+corrector is a very attractive method for implementing a GRNG in hardware, but has two disadvantages:

1) A CLT-n generator requires $O(n)$ resources, as n uniform random samples must be added together using $n-1$ adders. The length of the distribution tails and the smoothness of the distribution both require large n, so many resources are needed per GRNG.
2) The correction from CLT to Gaussian distribution requires a complicated non-linear transform. It can be approximated using degree-1 polynomials and a single multiplier, but will only match the Gaussian CDF at a few points [4].

There is also a complicated resource-quality tradeoff between these two factors, as for a given target distribution quality we can trade-off a smaller CLT against a more complicated transform, and vice-versa.

The method proposed here takes the opposite approach, as rather than summing up extremely non-Gaussian base distributions then correcting them, we start with base distributions which are already close enough to the Gaussian that after the CLT step they need no further correction. In order to achieve this the base distributions must be extremely cheap in terms of hardware resources, so we propose the use of simple look-up table generators, requiring no DSP blocks.

The second idea is that instead of using multiple base samples to generate one Gaussian sample, we can actually

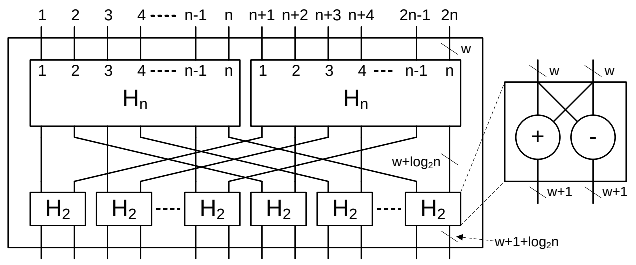

Figure 1. Butterfly architecture for fast Hadamard transform H_{2n}.

use multiple base samples to produce the same number of Gaussian samples. As long as the base transforms are IID, we can apply an orthogonal Hadamard transform which will result in IID output samples. This means that the number of CLT steps can be greatly increased, as the resource cost per generated sample increases as $O(\log n)$, rather than the $O(n)$ growth of the standard CLT.

So where the CLT+corrector method takes an n-vector of uniform samples and produces one Gaussian sample g

$$g = \text{Polynomial}(b), \qquad b = \sum_{i=1}^{n} - 1^i \vec{u}_i$$

the Table-Hadamard method takes an n-vector of uniform samples and produces a vector $\vec{g}$ of n IID Gaussian samples

$$\vec{g} = \text{FastHadamard}_n(\vec{b}), \qquad \vec{b}_i = \text{Table}[\vec{u}_i]$$

It is the relative cheapness of the table transform, combined with the $\text{O}(\log n)$ cost of the Hadamard transform which leads to the efficiency of this method.

A. Hadamard Transform

Given a vector $\vec{g}$ of IID samples, we can use an orthogonal matrix $\mathbf{A}$ to produce a transformed vector of random samples $\vec{x} = \mathbf{A}\vec{g}$. An orthogonal matrix has the property that $\mathbf{A}^T\mathbf{A} = \mathbf{I}$, meaning all rows and columns are mutually orthogonal and are unit vectors, so this means that if $\vec{g}$ contains IID zero-mean samples with variance σ^2, then $\vec{x}$ will also contain zero-mean samples with a variance of σ^2.

A simple orthogonal matrix is the 2x2 Hadamard matrix:

$$\mathbf{H}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & -1\\ 1 & 1\end{bmatrix}$$

Choosing $\vec{g} = \mathbf{H}_2\vec{b}$ results in the transform:

$$\vec{g}_1 = (\vec{b}_1 - \vec{b}_2)/\sqrt{2}$$
$$\vec{g}_2 = (\vec{b}_1 + \vec{b}_2)/\sqrt{2}$$

It is clear that the CLT will apply here, so $\vec{g}_1$ or $\vec{g}_2$ will be closer to the Gaussian than $\vec{b}_1$ or $\vec{b}_2$, but what is less obvious is that *both* samples converge on the Gaussian while still remaining independent samples. This is due to the orthogonality of the transform, so the elements of $\vec{g}$ will not be correlated with each other.

We can find larger orthogonal matrices of any size, but a particularly efficient implementation occurs if we recursively define Hadamard matrices of higher orders:

$$\mathbf{H}_{2n} = \begin{bmatrix}\mathbf{I}_n & -\mathbf{I}_n\\ \mathbf{I}_n & \mathbf{I}_n\end{bmatrix}\begin{bmatrix}\mathbf{H}_n & \mathbf{0}\\ \mathbf{0} & \mathbf{H}_n\end{bmatrix} = \begin{bmatrix}\mathbf{H}_n & -\mathbf{H}_n\\ \mathbf{H}_n & \mathbf{H}_n\end{bmatrix}$$

This recursive definition allows an FFT-like butterfly implementation, leading to the Fast Hadamard Transform (FHT) shown in Figure 1.

Assuming a fixed-point implementation, with the base distribution having w bits, then each $\mathbf{H}_2$ transform produces two $w+1$ bit outputs, requiring $2(w+1)$ LUTs for the adder and subtractor. The total cost in LUTs of an n output Hadamard transform is:

$$\begin{aligned}\text{L}(\mathbf{H}_2, w) &= 2(w+1)\\ \text{L}(\mathbf{H}_n, w) &= 2\,\text{L}(\mathbf{H}_{n/2}, w) + \frac{n}{2}\,\text{L}(2, w + \log_2 n - 1)\\ &= n\log_2 n(w + (\log_2 n + 1)/2)\end{aligned}$$

By comparison, the single output CLT-n transform costs:

$$\begin{aligned}\text{L}(CLT_2, w) &= (w+1)\\ \text{L}(CLT_n, w) &= 2\,\text{L}(CLT_{n/2}, w) + w + \log_2 n\\ &= (n-1)(w+2) - \log 2n\end{aligned}$$

The cost of the Hadamard is $O((w+\log n)n\log n)$ versus a cost of $O(wn)$ for the CLT-n, but the cost *per output* of the Hadamard is $O((w + \log_n)\log n)$ while the CLT-n is still $O(wn)$. This significantly reduced complexity means that we can support much higher values of n for the same number of resources. For example, CLT-8 with $w = 16$ requires 123 LUTs per output, but the Hadamard only requires 54 LUTs. Alternatively, in 117 LUTs per output, the Hadamard is able to provide $n = 64$.

The author is not the first to note the potential uses for the Hadamard transform in generating Gaussian random numbers. The 4x4 Hadamard is implicitly used in the Wallace method [5], but as a means of mutating an existing pool of Gaussian samples via the maximum entropy principle, rather than to exploit the effect of the CLT. Direct application of the Hadamard to transform uniform samples to the Gaussian distribution appears to have been considered in software [6], but only as a proof-of-concept. So it appears that, while it is well known that FPGAs can be used to efficiently perform Hadamard transforms, its use in generating large numbers of Gaussian samples in parallel has not been identified.

B. Table Generator

While the Hadamard transform allows significant numbers of samples to be aggregated, the convergence to the Gaussian is still quite slow if we start with uniform samples. Only loose bounds on the speed and nature of convergence are available, via the Berry-Esseen theorem. These bounds also only consider the maximum absolute CDF error, while the relative CDF error in the tails of the distribution may require many accumulations before it starts to converge. This is what necessitates the polynomial correction in the CLT+corrector approach, as most of the correction is applied in order to stretch out the tails.

However, the Central Limit Theorem also states that the closer the base samples are to the Gaussian, the quicker they will converge to the Gaussian. In particular, if we are able to match a large number of the higher order moments in the base distribution, then the CLT steps will be much more effective. The higher order moments mainly describe what happens in the tails of a distribution, so if the base distribution already has somewhat accurate tails, the Central Limit Theorem can quickly polish them.

There are a number of ways of generating medium-quality Gaussian samples in a small number of resources, for example, low-precision Box-Muller, or a Piecewise-Linear approximation. These all require a small number of block-RAMs and/or DSPs in order to transform uniform random bits into Gaussian samples. However, as the goal of this work is to use minimum resources per generated sample, we focus on methods which can be implemented using only logic resources.

One of the simplest ways of generating a non-uniform distribution is through tabulation. Given a target distribution X with CDF $F_X(x)$, and a table $L[1]..L[k]$ with k entries, we can quantise X into a discrete approximation:

$$L[i] = F_X^{-1}\left(i/(k+1)\right)$$

If k is a power of two, we can sample from this distribution using a $\log_2 k$-bit uniform random number u:

$$b = L[u]$$

We shall use L to denote both the table, and the discrete random variable generated by uniformly sampling the table. The resource requirements for a table generator are simply the ROM resources required to store the table, and k uniform random bits to provide a random index.

As $k \to \infty$ then $\max : |F_L(x) - F_X(x)| \to 0$, but as with the central limit theorem this convergence is usually quite slow, and applies only to absolute error. The basic table construction will not even result in the correct standard-deviation, but this is critically important if we are to apply the Hadamard transform, as the output standard-deviation is simply $\sqrt{n}$ times the input standard-deviation.

The table distribution has previously been used in the context of multivariate GRNGs, where a method for cubic correction of tables was suggested [7]. A correction for second and fourth order moments (standard-deviation and kurtosis) was presented, but higher-order corrections could not be found. Here we are able to find robust corrections for higher-order moments, using polynomial correction up to degree-7 in order to correct moments up to order 8.

We will build the tables to approximate the zero-mean unit-variance Gaussian distribution, as then any other standard-deviation can be achieved simply by linear scaling of all the table entries. Our starting point is the uncorrected table L:

$$L[i] = \Phi^{-1}(i/(k+1)), \qquad 1 \leq i \leq k$$

The central moments of the table are then given by:

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

In order to provide a good starter distribution we would like our table to match as many moments given in Equation 1 as possible. The uncorrected table is symmetric by definition, and so $m^{(d)}(L) = 0$ for all odd d, but for even d we require:

$$m^{(2)}(L) = 1 \qquad m^{(d)}(L) = (d-2)m^{(d-2)}(L)$$

This leads to the familiar definition that the kurtosis ($m^{(4)}$) of a Gaussian distribution is three, but we would also like to match more of the low order moments:

$$m^{(6)}(G) = 15 \quad m^{(8)}(G) = 105 \quad m^{(10)}(G) = 945$$

We will correct the table L using a polynomial correction of degree r, using only odd coefficients to preserve the symmetry of the table:

$$L'[i] = \sum_{j=1}^{r} a_j L[i]^j$$

The moments of the corrected table will be:

$$\begin{aligned} m^{(d)}(L') &= \frac{1}{k}\sum_{i=1}^{k} L'[i]^d \\ &= \frac{1}{k}\left[a_1\sum_{i=1}^{k} L[i] + a_3\sum_{i=1}^{k} L[i]^3 + \ldots + a_r\sum_{i=1}^{k} L[i]^r\right]^d \end{aligned}$$

The expansion of $m^{(d)}(L')$ results in a multinomial with coefficients defined in terms of $\sum_{i=1}^{k} L[i]^j$ for $1 \leq j \leq r$. For a cubic correction it is possible to solve for the first two moments directly, as a system of polynomial equations, but as noted in [7] this is not feasible for higher order corrections, due to the growth of the multinomial coefficients.

Here we take a different approach to the solution, using numerical optimisation to minimise the euclidean distance from the Gaussian moments, using the objective function:

$$\sqrt{\sum_{i=1}^{d}\left[m^{(i)}(L') - m^{(i)}(G)\right]^2} \tag{2}$$

Because the odd moments are already correct due to the symmetry of the table, only the even moments need to be matched. This means that to correct up to moment d there are $d/2$ polynomial coefficients, and $d/2$ roots to find.

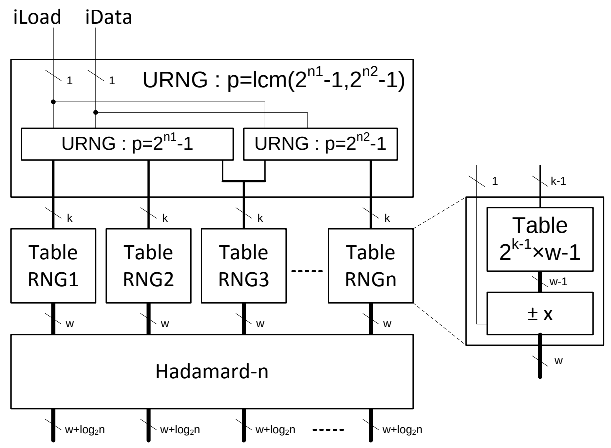

Figure 2. Overview of complete Table-Hadamard architecture.

The gradient and Jacobian of the objective function are both immediately available, allowing the use of standard function minimisation algorithms. However, the heptic (degree 7) expansions of the objective require addition of very high powers, so the calculations must be performed in extended precision. The moment correction method is available as part of the FloPoCo framework, and is automatically calculated when instantiating Table-Hadamard generators.

As will be shown in Section V, the effect of higher order table corrections can be significant, with the new quintic and heptic corrections increasing the quality of the generator both in terms of CDF error and empirical statistics.

IV. Practical Implementation

The practical architecture of the Table-Hadamard GRNG is shown in Figure 2. At the top is a URNG producing $n \log_2 k$ uniform random bits per cycle, where n is the number of Gaussian output samples, and k is the number of points in each table generator. These uniform samples are transformed by n identical and parallel table generators, producing signed IID pseudo-Gaussian samples of width w. Because the tables are symmetric, only the positive half of the table is stored, and a random sign is attached using a conditional negator. Finally, these n samples are transformed in parallel using the H_n Hadamard transform, producing n IID Gaussian samples in signed fixed-point with $w + \log_2 n$ bits.

Note that, with the exception of the URNG, the entire structure contains no feedback paths, and so can be completely pipelined. There are also no multipliers in the structure, as the only operations are: exclusive-or, in the URNG; table-lookup, in the table generator; and addition, in the Hadamard and the table generator.

The Table-Hadamard is designed to generate large numbers of samples, and to do that it consumes $n \log_2 k$ input bits. For a LUT-based generator with $n = 128, k = 2^7$, this requires 896 uniform bits per cycle, while for a highly-parallel high-quality BRAM-based generator with $n = 256, k = 2^{11}$, we need 2816 bits per cycle. To maintain statistical properties we also want every bit pattern to be equally likely, so that all outputs from the Hadamard are equally likely. As there are $n \log_2 k$ uniform bits, that means we need an overall period of $2^{n \log_2 k}$.

Two common approaches for generating uniform bits are to use many parallel 1-bit output LFSRs, or to use a smaller number of parallel word-level generators, such as 32-bit output Mersenne Twisters (MT). Parallel LFSRs have poor resource efficiency, as multiple LUTs are needed per generated bit, but the biggest problem is that the overall period of the output bits will be limited to that of the LFSR, typically around 2^{128}, and far smaller than the desired period. Large generators such as the MT provide a period 2^{19937}, but to do this need significant RAM resources - to generate 896 bits we would need 28 parallel MT instances, each of which require a block-RAM and must be given a unique random seed at initialisation.

Hardware optimised generators such as the LUT-SR [8] generator can provide large numbers of bits per cycle, though dwork less well with more than 512 bits due to congestion. They also provide periods well in excess of what is needed, but to achieve this they require two logic resources per output bit. The most appropriate generator is the LUT-OPT generator [9], as this requires exactly one LUT per output bit, and an r-bit generator has a period of exactly $2^r - 1$.

For $64 \leq n \log_2 k \leq 256$ the LUT-OPT generator is an ideal solution, but for larger generators we face two problems:

1) The LUT-OPT generator uses randomised connections between the LUTs in the generator. The maximum fan-out of any net is limited to the number of LUT inputs in the architecture, but for very large generators this can cause timing degradation.
2) In order to create a generator with exactly r bits, it is necessary to have the complete factorisation of $2^r - 1$, but for large r the factorisation may not be known.

One solution is simply to choose a generator with period r which is an approximate factor of $n \log_2 k$, then instantiate parallel copies of it. For example, for $n \log_2 k = 2816$ bits we could choose $r = 256$ and create 11 parallel instances, but the overall generator will have a period of just $2^{256} - 1$, which is much smaller than desired. The parallel URNG instances will also be identical, so unless they are given unique seeds at initialisation, they will all generate exactly the same sequence, introducing catastrophic correlations.

To fix this problem, the approach which has been used for the Table-Hadamard, and is now included as a general purpose solution for generating uniform random bits in FloPoCo, is to decompose the generator into LUT-OPT generators with distinct periods. In general, if we have m RNGs with periods $p_1 = 2^{r_1} - 1, p_2 = 2^{r_2} - 1, ..., p_m =$

$2^{r_m}-1$, then the period of the overall bit-stream is the Lowest Common Multiple (LCM) of the periods: $p = \mathrm{lcm}(p_1, p_2, ..., p_m)$. Note that this is a statement about the entire output of the generator, as the period of any individual bit is simply the period of the sub-generator it came from. However, due to the way the Hadamard transform works, any bit of the input uniform RNG eventually affects *all* of the output Gaussian samples, so the period of the Table-Hadamard will also be p.

Ideally we would choose $n \log_2 k = r = \sum_{i=1}^{m} r_i$, with $\forall 1 \leq i < j \leq m : \gcd(p_i, p_j) = 1$, which would ensure that $p = \Pi_{i=1}^{m} p_i$, but for a given table of LUT-OPT generators and a target r, we may not be able to find a solution where all periods are relatively prime. However, if we use the simpler constraints $\forall 1 \leq i < j \leq m : \gcd(r_i, r_j) \leq \tau$, then for LUT-OPT generators we find that $p \geq \frac{1}{\tau^m} \Pi_{i=1}^{m} p_i$, meaning the period is at worst a factor of τ^m smaller than the ideal case. For all cases considered for the results in this paper, we were able to choose $\tau = 5$, meaning the period of the ensemble generator is close to the maximum.

Ensuring that $\gcd(r_i, r_j) \leq \tau$ also requires that the periods are distinct, which makes seeding trivial. A LUT-OPT generator is seeded using a 1-bit wide random seed, loaded over multiple cycles. If we had $r_1 = r_2 = ...r_n$, we would need m different seeds to ensure that the generators each have different states. However, our conditions on GCD ensure that the component periods are unique, so even if all the generators are given the same seed, they will still produce different output streams. This allows us to seed all URNGs in the Table-Hadamard from just one single stream, meaning the seeding interface is independent of the GRNG parameters.

V. Evaluation

The parameters of the Table-Hadamard generator are:

- n : The size of the Hadamard transform and number of output samples.
- k : The number of entries in each Table generator.
- f : Fractional precision of the output distribution.
- d : The degree of the polynomial correction applied.

For the evaluation we mainly focus on $k = 7$ and $k = 11$, which correspond to the use of a 6-LUT as a 64 element ROM, and the use of a Virtex-6 BRAM in 2048x18 mode. However, the open source implementation available in FloPoCo can support any combination of parameters.

Each of these parameters will change the characteristics of the generator, with the most important factors being:

- Resource usage : How many LUTs, FFs, and BRAMs are used, both per generated sample, and in total.
- Performance : What clock rates can the RNG support, as large generators may becomes congested.
- Theoretical quality : What is the analytical difference between the target Gaussian distribution and the achieved distribution.
- Empirical quality : What statistical flaws are detectable in finite-size samples generated by the RNG.

All the resource and performance figures are derived from the publicly available code, and the statistical and theoretical tests are also included in the FloPoCo.

Theoretical Evaluation : When comparing two distributions with known CDFs, one can choose many points of comparison, such as the worst PDF error or the average absolute CDF error. Measures of PDF accuracy are difficult to interpret, as a small but consistent PDF error over a large contiguous part of the range can lead to large deviations in the CDF. Average measures are also suspect, as they may hide a few relatively large CDF errors, which are precisely the kind that may cause a χ^2 test or Monte-Carlo application to fail. So for this analysis we focus on maximum error, as this represents the most difficult metric, and choose the relative CDF error, as this puts more emphasis on the difficult to match tails. The relative CDF error typically starts low, then increases towards the tails, so we define a metric $R_\phi(X)$, which is the maximum relative CDF error within the range $-\phi..0$:

$$R_\phi(X) = \max_{x \in [\phi..0]} \left| \frac{\Pr[x < X] - \Pr[x < G]}{\Pr[x < G]} \right| \qquad (3)$$

In order to apply this metric, we need to recover the exact CDF of a given Table-Hadamard generator, which is performed using FFT-based convolution, with all calculations using MPFR floating-point with 128-bit mantissa.

Empirical Evaluation : The classic test for empirical goodness-of-fit is the χ^2 test, which bins a given random sample into buckets, then checks the number of samples in each bucket against the number expected from the target distribution. Deciding where the buckets should be placed within the output range affects the power of the test, and the types of flaws which can be detected. One approach is to choose buckets which have equal probability and cover different amounts of the output range, which tends to detect flaws in the centre of the distribution. Another approach uses buckets covering equal amounts of the range, which is better able to detect flaws in the tails. We use the latter approach, creating 2048 equal range buckets covering the range -16..+16. Many of these probabilities are very small, so adjacent buckets are coalesced as necessary, until the expected samples per bucket exceeds 10.

Figure 3 shows the change in maximum relative CDF error for three different Hadamard sizes of $n = 2^2, 2^6, 2^{10}$, and for $k = 7$ (LUT Table) and $k = 11$ (BRAM Table). For $n = 2^2$ the distribution is rather poor, with the error increasing to one at about the -6σ point, as the CDF of the generator has dropped to zero. However, for $n = 2^6$ the error has improved significantly, with the relative error less than 1% up to the -8σ range, and around 10^{-5} in the -4σ range. The BRAM generator has an advantage of around 5 times on average.

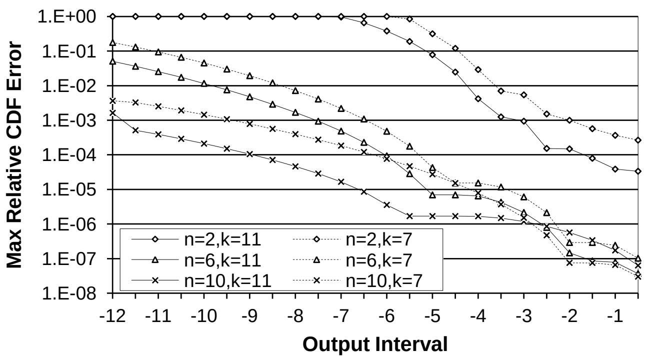

Figure 3. Maximum relative CDF error within an interval, for different numbers of Hadamard levels and table sizes.

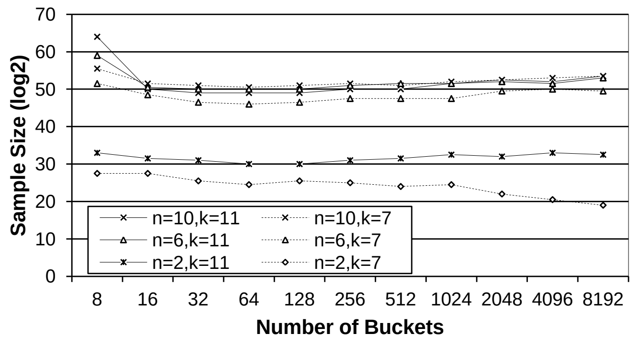

Figure 4. Last good χ^2 sample size for varying numbers of buckets, for different numbers of Hadamard levels and table sizes.

For $n = 2^{10}$ the relative error is very low, and the BRAM generator in particular can provide a maximum error less than 10^{-4} even down to -9σ. However, this would require 512 BRAMs, and produced 1024 samples per cycle, so it is not clear how useful it would be in practise.

The χ^2 results for the same generators are shown in Figure 4, showing that the change in relative CDF error is reflected in the point at which the χ^2 test fails. Both the $n = 2^2$ generators fail at quite small sample sizes of around 2^{30}, though the BRAM generator has a slight advantage due to its more accurate starting distribution. However, for $n = 2^6$ all the generators are greatly improved, requiring sample sizes of 2^{50} before failure.

The effect of the different corrections is shown in Figure 5 and Figure 6 for $n = 2^4$. Looking at the relative CDF error, the advantage of the new higher order correction is clear, as going from cubic to quintic provides an order of magnitude improvement in error for the LUT-based generator, and close to two orders of magnitude when going from cubic to heptic for the BRAM generator. Translated to empirical quality, this results in the failing sample size for the LUT-based generator improving by a factor of around 500, while the BRAM-based generator improves by around 2000 times. In both cases the hardware used is exactly the same, the only change is in the correction applied to the tables.

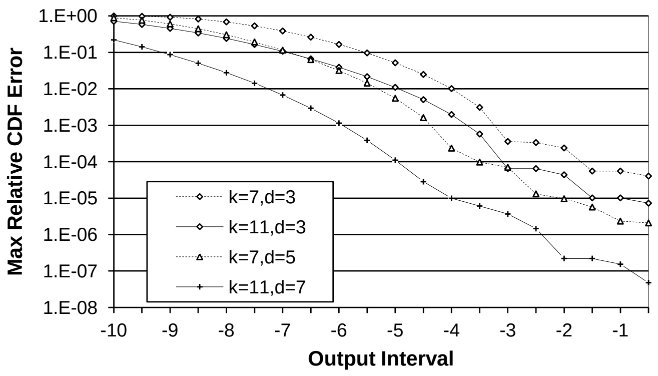

Figure 5. Maximum relative CDF error within an interval, for $n = 2^4$ and different degrees of polynomial correction.

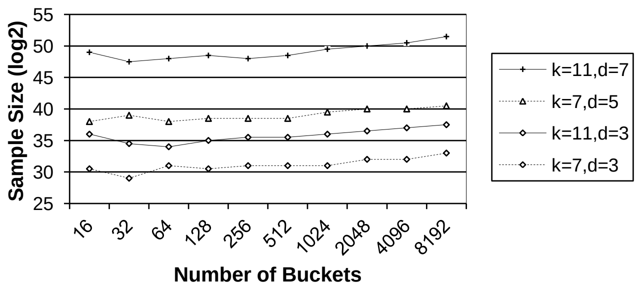

Figure 6. Last good χ^2 sample size for varying numbers of buckets, for $n = 2^4$ and different degrees of polynomial correction.

Figure 7 shows the post-place and route resource usage (measured in slices) and clock rate achieved for the generators with $k = 7$ on a Virtex-6 xc6vcx130t-2. The GRNG produces too many outputs to be routed to pins, so all the outputs are fed into a pipelined xor chain, then routed to a single pin. The results were placed and routed with out-of-the-box settings for Xilinx XST and ISE, with the only addition being a 400MHz timing constraint.

As would be expected, the total resource consumption is very predictable, increasing in proportion to n, with the logarithmic term not visible over this range. This means that

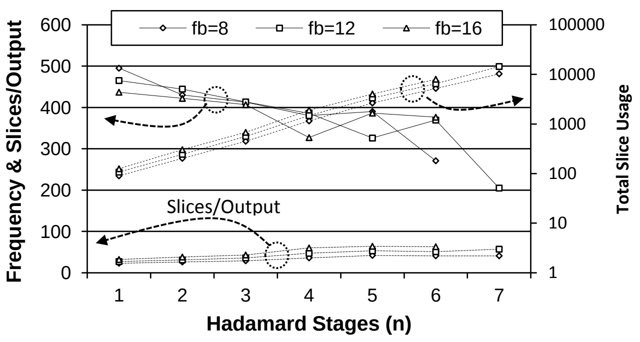

Figure 7. Comparison of post-place and route resource usage and performance in Virtex-6.

Design	[10]	[2]	[5]	[11]	[4]	n=64;k=7	n=64;k=11
Method	BM	BM	Wlc	Zgrt	CLT-Corr.	Table-Hadamard	
Slices	534	1528	770	891	420	102	98
BRAMs	2	3	6	2	0	0	1
DSPs	3	12	4	2	1	0	0
Bitwidth	16	16	24	32	16	18	18
Tail Accuracy	6.6σ	8σ	7σ	N/A	6σ	8σ	9σ
MSamp/sec	440	468	155	168	220	351	301

Table I
COMPARISON WITH PREVIOUS WORK, FOLLOWING THE STRUCTURE OF [4]. NOTE THAT THE TABLE-HADAMARD LISTS RESOURCES PER OUTPUT.

across the board, the resource consumption is well below 100 slices per output sample, and is typically around 50. The clock rate does not behave so well, as though it meets the timing constraint for $n \leq 2^3$, for higher n the tools have difficulty placing and routing the design, and for $n \geq 2^6$ the clock rate drops significantly. Informal floor-planning experiments have shown that it is relatively simple to get the clock rate back up to 400MHz with a few coarse area-groups, even for large n, but here we only report completely automated results. However, $n = 2^6$ already provides high quality Gaussian samples and is very routable.

Comparison with other GRNGs is difficult, as each author uses different metrics, and the source code to the GRNGs is usually not available. Rather than cherry pick statistics, this paper will follow the comparison table from [4], as it is the most recent Gaussian random number paper in the literature. The results are summarised in Table I.

VI. CONCLUSION

The Table-Hadamard method exploits the parallel nature of FPGAs to generate large numbers of Gaussian samples in parallel. Because there are usually many parallel consumers of random numbers, we can use one GRNG which generates multiple samples per cycle, which are then routed to multiple simulation units. The Table-Hadamard achieves this by using a large fully parallel Hadamard transform, using the Central-Limit Theorem to improve the distribution of each sample, while incurring only an $O(\log n)$ resource cost with the number of input samples. By creating initial table generators which are carefully corrected, the output distribution is of high quality, and requires a quarter the resources of the next most efficient generator, while also providing a much better output distribution.

The Table-Hadamard works well in situations requiring many samples, but one weakness is that if only one or two samples are needed per cycle then there are not enough Hadamard stages to correct the distribution. For very large numbers of outputs there are also congestion problems, though these can be handled manually with floor-planning. So while the Table-Hadamard is not appropriate in all situations, it is highly recommended for embarrassingly parallel applications such as Bit-Error-Rate testing and Monte-Carlo simulations.

ACKNOWLEDGMENT

The author would like to thank Florent de Dinechin for his help in integrating this work into FloPoCo.

REFERENCES

[1] 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.

[2] D. Lee, J. Villasenor, W. Luk, and P. Leong, “A hardware Gaussian noise generator using the box-muller method and its error analysis,” *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659–671, 2006.

[3] R. Andraka and R. Phelps, “An FPGA based processor yields a real time high fidelity radar environment simulator,” in *Conference on Military and Aerospace Applications of Programmable Devices and Technologies*, 1998.

[4] J. S. Malik, J. N. Malik, A. Hemani, and N. D. Gohar, “Generating high tail accuracy gaussian random numbers in hardware using central limit theorem,” in *Int. Conf. VLSI and SoC*, 2011, pp. 60–65.

[5] 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.

[6] S. O’Connor, “The hadamard transform fact sheet,” http://www.dsprelated.com/showmessage/65499/1.php, 2006.

[7] D. B. Thomas and W. Luk, “An FPGA-specific algorithm for direct generation of multi-variate gaussian random numbers,” in *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors*, 2010, pp. 208–215.

[8] ——, “FPGA-optimised uniform random number generators using luts and shift registers,” in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2010, pp. 77–82.

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

[10] A. Alimohammad, S. F. Fard, B. F. Cockburn, and C. Schlegel, “A compact and accurate gaussian variate generator,” *Trans. VLSI*, vol. 16, no. 5, pp. 517–527, 2008.

[11] 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.