A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation

David B. Thomas
Imperial College London
dt10@doc.ic.ac.uk

Lee Howes
Imperial College London
lwh01@doc.ic.ac.uk

Wayne Luk
Imperial College London
wl@doc.ic.ac.uk

ABSTRACT

The future of high-performance computing is likely to rely on the ability to efficiently exploit huge amounts of parallelism. One way of taking advantage of this parallelism is to formulate problems as "embarrassingly parallel" Monte-Carlo simulations, which allow applications to achieve a linear speedup over multiple computational nodes, without requiring a super-linear increase in inter-node communication. However, such applications are reliant on a cheap supply of high quality random numbers, particularly for the three main maximum entropy distributions: uniform, used as a general source of randomness; Gaussian, for discrete-time simulations; and exponential, for discrete-event simulations. In this paper we look at four different types of platform: conventional multi-core CPUs (Intel Core2); GPUs (NVidia GTX 200); FPGAs (Xilinx Virtex-5); and Massively Parallel Processor Arrays (Ambric AM2000). For each platform we determine the most appropriate algorithm for generating each type of number, then calculate the peak generation rate and estimated power efficiency for each device.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*

General Terms

Algorithms, Design, Performance

Keywords

Monte-Carlo, Random Numbers, FPGA, GPU, MPPA

1. INTRODUCTION

The power of conventional super-scalar CPUs has steadily increased for many decades, but we have now reached the point where processor performance does not scale geometrically with passing generations. However, the performance offered by parallel processing systems *is* still increasing with each generation, through the introduction of multi-core versions of conventional CPUs, and through "unconventional" computing platforms, such as FPGAs (Field Programmable Gate Arrays), GPUs (Graphics Processor Units), and MPPAs (Massively Parallel Processor Arrays).

One application domain where it is often possible to both easily describe applications, and to take advantage of parallel computational power, is Monte-Carlo simulation. This is one of a class of "embarrassingly parallel" application domains, where as the number of computational elements increases, the communication required between elements does not increase super-linearly. We know that Monte-Carlo applications are able to scale (approximately) linearly across parallel architectures, but this doesn't necessarily mean that the new generation of parallel architectures will be ideal for Monte-Carlo; we also need to know that they can efficiently support the types of computation found in such applications. This includes standard building-blocks such as addition, multiplication, and division, but also requires Random Number Generators (RNGs).

The importance of efficient RNGs is demonstrated by the enormous effort applied by researchers to traditional software techniques, both to improve the efficiency of generation techniques, and to modify simulations to reduce the number of random numbers consumed per run. Such is the cost of random numbers in software, that in the Gibson method for simulating discrete-time biochemistry it is preferable to perform four floating-point operations (including a divide) just so that a previous random number can be re-used, rather than the simpler approach of generating a new random number. However, in an FPGA accelerated simulation the economics change: it is actually cheaper to generate a new random number, with the added benefit that the simulation code is easier to write and understand.

To understand how new parallel architectures can and should be used to accelerate Monte-Carlo, we need to properly understand the costs of RNG primitives. In this paper we consider how best to implement RNGs across four platforms, and then calculate the peak generation rate and power efficiency of each device. Our contributions are:

- An examination of random number generation on four different architectures: CPU, GPU, MPPA, and FPGA.
- Selection of RNGs for the uniform, Gaussian, and exponential distribution for each platform.
- A comparison of the four platform's potential for Monte-Carlo applications, both in terms of raw generation

FPGA'09, February 22–24, 2009, Monterey, California, USA.

speed, and estimated power efficiency.

2. REQUIREMENTS FOR RNGS

Highly parallel architectures present challenges for RNG designers due to the enormous quantities of random numbers generated and consumed during each application run. To ensure that we compare solutions that meet a common standard, we use the following requirements, which all the generators benchmarked in this paper adhere to.

Deterministic and Pseudo-Random: In this paper we only consider deterministic generators. This class of generator uses a deterministic state transition function, operating on a finite-size state, and hence will produce a pseudo-random sequence that repeats with a fixed period. The repetition property is critical in many applications, as it allows applications to be repeatedly executed using the same sequence of numbers, allowing "surprising" simulation runs to be re-examined, or to use the same random stimulus to drive different models. We also require that generators are pseudo-random, i.e. they should be designed to produce a sequence that looks as random as possible (see later requirements on statistical quality). These requirements rules out both True Random Number Generators (TRNG) which use some source of physical randomness; and Quasi-Random Number Generators (QRNG) which produce a sequence that covers some multi-dimensional space "evenly", rather than pseudo-randomly.

Period: The period of each random sequence must be at least 2^{160}, which allows the overall sequence to be split up into 2^{64} sub-streams of length 2^{64}, with an extra "safety-factor" of 2^{32}. Larger periods are of course desirable, and are achievable in software due to the high-ratio of memory to processing elements, but in highly-parallel devices it is likely to be difficult to allocate the large amounts of memory needed to hold the state of each generator per element. However, parallel devices should be able to substitute increased processing per output sample for a large state, and so maintain statistical quality.

Stream-splitting: When executing Monte-Carlo applications in parallel it is critical that each node has an independent stream of random numbers, completely independent of the numbers on all other nodes. To ensure this independence we require that it must be possible to partition the stream into (at least) 2^{64} non-overlapping sub-sequences of length (at least) 2^{64}. Alternatively, the generator must have a period so large that if 2^{64} random sub-sequences of length 2^{64} are chosen, then the probability of any pair overlapping is less than 2^{-64}.

Empirical statistical quality: It is difficult to prove anything about the theoretical randomness of generators, particularly when comparing different types of generator. Instead we rely on batteries of empirical tests, which looks for signs of non-randomness in sub-sequences of each generator. The Big-Crush test battery from TestU01 [12] is the most stringent available, applying 106 tests to 2^{38} samples from each generator, and each generator must pass all the tests. For uniform generators we require that the generator must supply 32-bits, while for non-uniform generators we require that, after applying the CDF to transform to the uniform distribution, the most significant 16 bits pass the test.

Empirical distribution: In addition to the tests for statistical randomness provided by TestU01, we also require that the empirical distribution is extremely accurate. To test the marginal distribution, we require that the generator is able to pass a χ^2 test with 2^{16} equal probability buckets, for at least 2^{36} samples.

No specific requirements are made on data-types: if a generator can pass the empirical quality tests, then whatever data-type (i.e. fixed-point or floating-point) it produces is assumed to be acceptable.

3. OVERVIEW OF RNG METHODS

There are a huge variety of methods for generating random numbers, both uniform and non-uniform, thanks to a stream of theoretical advances and practical improvements over the last 60 years. Devroye's book [6] on the subject provides detailed coverage of methods up to the mid 1980s; more recent developments in uniform generation are surveyed by L'Ecuyer [11], and Gaussian generators by Thomas [25]. There appear to be no recent surveys of exponential RNGs, but many of the best techniques for Gaussian generation can also be applied to exponential numbers.

3.1 Uniform Generation

The purpose of uniform RNGs is to produce a sequence that appears as random as possible. Each generator has a finite set of states $\mathbf{S}$, a deterministic stateless transition function t from state to state, and a mapping m from each state to an output value in the continuous range $[0, 1]$.

$$t : \mathbf{S} \mapsto \mathbf{S} \qquad\qquad m : \mathbf{S} \mapsto [0, 1] \tag{1}$$

On execution the generator is started in some state $s_0 \in \mathbf{S}$. Repeated application of t produces an infinite sequence of successor states $s_1, s_2, ...$, and also a sequence of pseudo-random samples $x_1, x_2, ...$:

$$s_i = t(s_{i-1}) \qquad\qquad x_i = m(s_i) \tag{2}$$

As t is deterministic and $\mathbf{S}$ is finite, the output sequence must eventually repeat itself. The period p of the generator is defined as the shortest cycle within the sequence:

$$\min_p : \forall i : s_{i+p} = s_i \qquad\qquad 1 \leq p \leq |\mathbf{S}| \tag{3}$$

One of the measures of efficiency of an RNG is how close p is to $\mathbf{S}$. In practise $\mathbf{S}$ is a vector of w storage bits, so a period close to 2^w is desirable.

The difficulty when creating RNGs is to balance the competing concerns. On the one hand we want extremely long periods, and sequences that are statistically random looking (even over long sub-sequences); but on the other, we want the computational cost per generated number to be very low, and the size of the RNG state should not be excessive. The history of random number generation has been the gradual improvement of these metrics: for example, longer periods with the same computational cost, or higher statistical quality with the same period and cost.

Each uniform generator is based on some underlying theory, which allows us to guarantee that each generator has a specific period, and often allows some kind of theoretical measure of statistical quality to be made over the entire output sequence. The classic generator is the Linear Congruential Generator (LCG), which uses integers in the range $[0..m)$ as its state space, and has a transition function

$t(s) = sc + a \mod m$, where c, a, and m are specially chosen integers. The maximum period of such generators is m, which means that even in a 64-bit machine $p \leq m \leq 2^{64}$, so these generators have fallen from favour.

Multiple Recursive Generators (MRG) are an extension that treats the state as a fixed-length FIFO of k integers, with a new value formed from a linear combination of the previous values (modulo m), then pushed into the FIFO to form the next state. Lagged Fibonacci generators are a popular subset of the MRG, which forms the next value through summation only (all multipliers are one). Such generators are still in use, but they are becoming less popular, due to their relatively high computational complexity, low period, or poor quality, when compared to modern generators. We considered the whole spectrum of generators when selecting for each platform, but these older methods were not competitive on (one or more of) period, quality or speed, so we do not mention them further here.

Currently the most popular type of generator (and the one selected for use in all the platforms we considered) is the binary linear generator. These operate in GF_2^w, i.e. they perform binary linear operations (logical conjunction or exclusive disjunction) on vectors of individual bits. Such operations are implemented in CPUs using bit-wise operations on words, such as bit-wise masks ("and" with a constant), bit-wise xor, and shifting. The central idea is to choose a fixed-size state, for example using k 32-bit words to form a $32 \times k$ bit state, then to apply binary linear transformations to this state, such as shifts and exclusive-ors. Each transformation instruction corresponds to a matrix; for example, if we have a 4-bit state x, the 1-bit shift left and right operators are:

$$\text{shl}(x) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} x \quad \text{shr}(x) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} x \qquad (4)$$

The combination of the state shifted left and right is then:

$$\text{shl}(x) \;\oplus\; \text{shr}(x) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} x \qquad (5)$$

After a number of instructions have been applied then a combined matrix $\mathbf{A}$ is built up. This matrix maps the current random number generator state to the next one: $x_{i+1} = \mathbf{A}x_i$. If the matrix $\mathbf{A}$ has a primitive characteristic polynomial [16], then the period of the sequence will be $2^w - 1$, where w is the number of bits in the state.

This basic framework has been used to create a large number of different RNGs, which vary in how they transform the state words, and so have different types of recurrence matrix. Some, such as the Combined Tausworthe [10], use combinations of shifts and masking to define low-period generators with relatively prime periods, then combine them to produce one overall generator. The XorShift [16], operates using only exclusive-or and shifting, using a small number of words ($k = 3..8$) that are all transformed in each pass.

Much larger period generators can be formed by treating the state as a tapped FIFO (similar to the MRG), forming a new word by reading a few words from the FIFO and performing binary linear operations on them, then pushing the new word into the FIFO. This approach led to the Mersenne Twister, a popular generator with the period of $2^{19937} - 1$ which has seen wide-spread use. More recent generators improve on these ideas, such as WELL [18], which increases the quality of the generated sequence, and SFMT [20], which uses 128-bit SIMD operations to achieve a significant speed advantage over generators based on 32-bit words.

Listing 1: Ziggurat method pseudo-code

```
float Ziggurat()
{
  const float W[N]={...};
  const int K[N]={...};

  do{
    // sample in range [-2^32..+2^32)
    int u=uniform();
    // scale to floating-point
    float x=u*W[u%N];
    if(u<K[u%N])  // fast path ...
      return x;   // ... taken ~99% of time
    if(SlowCheck(u)) // slow path
      return x;
  }while(1);
}
```

3.2 Non-uniform Generation

As with uniform generation, there are a number of options for generating non-uniform random samples, but there is much more of a difference between the algorithms. We break these methods down into four approaches [25]: inversion, transformation, rejection, and recursion.

Inversion: Each non-uniform distribution is defined by the Cumulative Distribution Function (CDF), which takes a value from the RNG's output range, and determines the probability of seeing a lower value.

$$F_G(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \qquad F_E(x) = 1 - e^{-x} \qquad (6)$$

Because the CDF maps from the non-uniform distribution to the uniform, if we apply the Inverse CDF (ICDF) to a uniform sample u, then we produce a non-uniform distribution sample x. This is simple for the exponential distribution:

$$x = F_E^{-1}(u) = -\ln(1-u) \qquad (7)$$

but requires a potentially expensive logarithm for every single sample. In the case of the Gaussian, there is no closed-form solution for the ICDF. Approximations can be made using rational polynomials, but due to the behaviour of the ICDF near 0 and 1 it is necessary to use a number of polynomial segments of high degree.

Transformation: A fixed number of uniform samples are transformed into a fixed number of non-uniform samples, with no looping or branching. The canonical example is the Box-Muller transform [3], which takes a pair of independent uniform samples u_1, u_2, and transforms them to a pair of independent Gaussian samples g_1, g_2 using the transform:

$$g_1 = \sqrt{-2\ln u_1}\sin(2\pi u_2), \quad g_2 = \sqrt{-2\ln u_1}\cos(2\pi u_2) \quad (8)$$

Although simple, this method relies on transcendental functions which have historically been very expensive.

Rejection: Both inversion and transformation consume a known number of uniform inputs and require a fixed amount of computation per output sample. The idea of rejection methods is to use a cheap method to generate candidate samples, but with the drawback of occasionally having to discard the candidates and start again. This requires the generator to contain a loop, which will keep generating and testing candidates until one can be accepted, with the number of iterations following a geometric distribution. Examples of this class of generator include the Polar method, GRAND, the Monty-Python method, and SA.

The Ziggurat method is the most recent and efficient rejection method, which is currently the fastest software generator for the Gaussian and exponential distributions (see Listing 1). It uses an extremely fast candidate generator and acceptance check, and in 99% of iterations requires just one uniform input, a table lookup, a multiply, and a comparison. In the remaining 1% of cases the code is much more complicated, requiring transcendental functions to determine acceptance or rejection, but this code is taken so infrequently that the generator remains extremely fast.

Recursive: The final non-uniform method is different to the other three, in that it doesn't consume uniform input samples, but produces samples directly. The only generator in this class is the Wallace method [26], which can produce samples from the Gaussian or exponential distribution, and can be extremely efficient as it requires no transcendental functions and can be vectorised. However, this method has problems with correlations between samples, which are difficult to fix while retaining the simplicity (and performance) of the method.

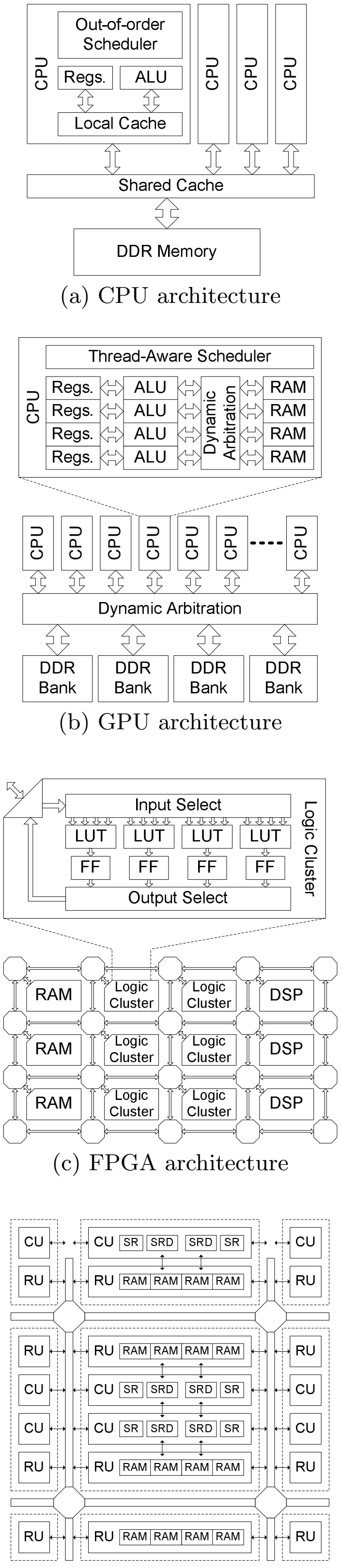
4. OVERVIEW OF PLATFORMS

We consider four different parallel platforms in this paper, so we now give a brief overview of each architecture. The main features of each are also shown in Figure 1.

CPU: Most current Multi-core CPUs operate in the same way as single-processor CPUs, using the shared memory paradigm for communication, with synchronisation achieved via a shared cache (or core-to-core cache coherency protocol). Each core hosts one thread at a time, with a set of registers containing thread state, an ALU dedicated to the current thread (containing a number of functional units), and a large unit devoted to management and scheduling tasks, such as branch prediction, instruction ordering, speculative execution, and so-on.

GPU: The idea behind GPUs is to dedicate as much silicon area as possible to ALUs, by removing all the scheduling logic and caches required to exploit instruction-level parallelism and reduce memory latency in CPUs. Instead, thread-level parallelism is used to hide latency, with each CPU executing up to 1024 threads at once. The threads execute in batches of 32 threads called warps, providing SIMD style parallelism, but with the ability to independently enable and disable each thread within a warp, allowing each thread to execute different parts of the program. However, this batching comes at a cost: the fewer threads within a warp that are active, the less parallel operations are executed per cycle. It is critical to minimise thread divergence, by making sure that all threads take the same branch of conditional statements, and execute loops the same number of times.

MPPA: Massively Parallel Processor Arrays introduce parallelism by using hundreds of very simple in-order-RISC CPUs. The CPUs are instantiated in a regular grid, with



(a) CPU architecture

(b) GPU architecture

(c) FPGA architecture

(d) MPPA architecture

Figure 1: Comparison of the internal architectures of the four platforms used in this paper.

	SR	SRD
Instr. width	16	32
Registers	8	20
Local RAM (bytes)	256	1024
Shifter	1-bit	multi-bit
ALUs	1	3
Multiply-accumulate	no	yes

Table 1: Properties of Ambric SR and SRD CPUs.

2D communication channels between them, and small local memories. Such small CPUs provide excellent efficiency in terms of peak performance per mm^2 or power efficiency, but present significant problems when partitioning applications. As this is the newest type of architecture, the next section explores the Ambric architecture in greater detail.

FPGA: Unlike the three previous architectures, FPGAs do not have any fixed instruction-set architecture. Instead they provide a fine-grain grid of bit-wise functional units, which can be composed to create any desired circuit or processor. Much of the FPGA area is actually dedicated to the routing infrastructure, which allows functional units to be connected together at run-time. Modern FPGAs also contain a number of dedicated functional units, such as DSP blocks containing multipliers, and RAM blocks.

We will now examine each architecture in turn, and attempt to match the different architectural features to the available RNG algorithms, attempting to maximise the RNG performance for each device.

5. MPPA: AMBRIC AM2045

One way of taking advantage of the large area now available to chip designers is to design relatively simple processors, then to instantiate a large number of them in a 2D mesh. In this paper we examine the Ambric AM2045 [2], as this is a very unconventional architecture: as well as having to partition applications over 336 processors, it is also necessary to map to two different types of processors. The processors are also very different to conventional CPUs, with only a few kilo-bytes of RAM per processor for instructions and data, and only basic integer addition and multiplication.

The two types of processor used in the AM2045 are the SR and SRD, summarised in Table 1. Both processors are very simple in-order 32-bit RISC processors, with the SR designed for extremely simple operations such as generating address streams and routing data around the device, while the SRD is more complex, with a large register set and an integer multiply-accumulate unit. Both processors execute most instructions with a throughput and latency of 1 cycle, with no stalls due to standard register usage. Stalls can occur due to conditional jumps, or when using the integer multiply-accumulator, but if non-conflicting instructions can be appropriately scheduled then the execution speed is one instruction per cycle.

Ambric CPUs communicate with each other over channels, which are self-synchronising uni-directional FIFOs, allowing producers and consumers to operate at different clock-rates in a GALS (globally asynchronous, locally synchronous) fashion. Each CPU has a number of input and output channels, and in many cases can use channels instead of registers as instruction input and outputs. If a channel is used as the input (output) of an instruction, but the channel is currently empty (full), then the processor automatically stalls until the channel is ready.

Listing 2: SA (exponential) pseudo-code

```
// Need 11 entries for 32-bit
const float Q[]={...};

float SA(){
  // Choose random segment of distribution
  float a=0;
  float u=2*UnifReal();
  while(u>1){
    u=(u-1)*2;
    a=a+log(2);
  }
  // Use rejection method within segment
  int i=2;
  float umin=UnifReal(),ustar;
  while(1){
    umin=min(umin,UnifReal());
    if(u<=Q[i++])
      return a+umin*log(2);
  }
}
```

The Ambric architecture makes use of clusters of SRs and SRDs, pairing two SRs and two SRDs with a set of four 2KB RAMs, shown in Figure 1(d). A dynamically arbitrating interconnect allows channel based communication between the processors and RAMs in the clusters, allowing CPUs to stream data between themselves, to read and write to the RU RAMs, and to connect to a chip-wide 2D channel network. This allows CPUs to transfer data to and from CPUs and RAMs elsewhere in the device, and to access external devices such as DDR RAMs, PCI Express connections, and general purpose IO.

From the point of view of random number generation, the Ambric duality of SR and SRD processor suggests a natural way of splitting things up. Uniform random number generation can be implemented using binary linear operations such as shifts and bit-wise ands which are supported on the SR. However, non-uniform generation typically requires some sort of multiplication, which can only be performed on the SRD. Because the Ambric architecture lets us efficiently connect processors together over local channels, this lets us map a uniform random number generator onto an SR processor, then stream these uniform numbers over a local channel to an SRD processor, where they can be transformed into a non-uniform distribution.

5.1 Uniform generation

Implementing a uniform RNG on an SR processor presents a number of challenges, mainly because of the extremely limited instruction set. Because there are no multipliers, any sort of Linear Congruential Generator or Multiply Recursive Generator is impossible, and the 256 byte memory (which must also contain the instructions) makes a classic Lagged Fibonacci generator impossible. However, the SR does contain a basic set of binary linear operations, such as shifts, exclusive-or, and bit-wise and.

There are many existing binary linear generators designed for CPUs, but these assume very different costs per operation. In particular, the assumption underlying the most popular small-state generators, such as the Combined Tausworthe [10], XorShift [16], or WELL [18] generators, is that multi-bit shifts are cheap. However, the SR instruction set only includes a 1-bit left or right shift, so multi-bit shifts must be executed over multiple cycles. Even if performance were not an issue, this approach is not feasible in the SR, as the code for the XorShift and Combine Tausworthe generator does not fit in the 128-word instruction memory.

However, the SR does include a number of instructions not usually found in processors, which are able to perform byte permutations. Our approach is to take one word from the state, then to shift it right a small number of times, performing an exclusive-or with a byte-permuted word from the state after each shift. If $s_1..s_k$ are the words of the current state, then the next state is given by $s'_1..s'_k$, calculated over k stages:

$$s'_i = (u >> i) \oplus \text{perm}_i(s_i) \oplus \text{transform}_i(s_{(i-1) \mod k}) \quad (9)$$

The choices within this framework are for the permutations and transformations applied at each stage (perm_i and transform_i), and for which of the state variables ($s_1..s_k$) is used to initialise the value u before each pass. The SR instruction set provides six byte permutations (including the identity permutation), and provides ten transformations (the six permutations plus four shift instructions). This provides a total of 60 possibilities per stage, plus the choice of which state element to use for u_0, leading to a total of $k60^k$ parameters sets for the whole generator.

We performed a brute-force search of this space for $k = 4$, and found the set of maximum period generators. Among the maximum period generators, we then selected from those with the best equidistribution, which is a measure of theoretical quality [10], The best generator found is actually maximally equidistributed, which means it has the best possible quality for this type of generator, and provides equivalent quality to generators such as the XorShift. However, all linear generators have a known statistical flaw, due to their fixed linear complexity [11]. The standard approach is to add some sort of post-processing stage, which can mask this linearity, while retaining the other desirable properties of the generator, such as its period. A popular approach is to combine the output with a Weyl generator [16], but this does not fix the least significant bit, so Crush is still failed.

We propose a simpler approach: output the running sum of the linear generators output. So given the sequence $x_1, x_2, ...$ from the linear generator, the generator output will be $c_1, c_2, ...$ where $c_i = c_{i-1} + x_i \mod 2^{32}$. The initial value c_0 of the running sum can be initialised to any value, and the overall period will be $2^{32}(2^{32k} - 1)$. This simple combiner allows all tests in Crush to be passed, while requiring only one more instruction per generated number, taking the total instructions (and cycles) per generated number to 14.

5.2 Non-Uniform Generation

In principle the Ziggurat method could be implemented on the SRD processor, as it works well in scalar oriented architectures, but in practise it is not a good match for the SRD's capabilities. The first problem is that the Ziggurat method relies on two tables, K and W, each of which contains N 32-bit values (in Ambric the floating-point table W would be converted to fixed-point). Typical values of N are 128 or 256, so a total of 2048 bytes are needed. These can be mapped into one of the RU RAMs, but this means that only three RAMs are left for the application logic that the random number generator is driving. RAM is a critical resource in the Ambric architecture, as it is needed for buffering between processors, for lookup tables, and to hold instruction streams that will not fit into the small per-processor local RAM, so this is a serious disadvantage.

The second problem is that the `SlowCheck` function is actually rather complicated. Internally it makes use of both the exponential and logarithm instructions, generates more uniform random numbers, and does more table-lookups. The transcendental functions present a particular problem, as an accurate fixed-point implementations requires either a large amount of instructions (both executed, and in the instruction stream), or a large lookup-table. Either option means that another RAM must be dedicated to random number generation, so each random number generator requires two of the four RAMs contained in each RU. This is unacceptable for real-world usage: it doesn't matter how fast we can generate random samples if there is not enough RAM left for the simulation we wish to drive.

What we require is a method with the following characteristics:

- No large tables of constants.
- No rational polynomials or transcendental functions.
- No reliance on floating-point.

Fortunately such a method exists, and is actually one of the earliest techniques, suggested by John von Neumann [8]. The central idea is to generate sequences of uniform random numbers, then either to accept or reject the candidate sample based on whether the first non-decreasing element in the sequence has an odd or an even index. This method has fallen from favour for general CPUs because it consumes a relatively large number of uniform input samples per non-uniform output sample, but because we are using the SR to generate uniform samples, we are not as sensitive to their cost. From our point of view the huge advantage is that the only operations are multiplies, additions, and table-lookups (no transcendentals), and the tables are small enough to fit in the SRD's local RAM.

Von Neumann's technique can sample from a large set of distributions, including the Gaussian and exponential distributions, but must be customised for each one. The most efficient version for the Gaussian distribution is the GRAND algorithm [4], and for the exponential distribution it is the SA algorithm [1]. Pseudo-code for the SA method is given in Listing 2 (with some optimisations omitted for brevity). The structure of the general method is: first, choose a random segment of the output range within the distribution; second, generate candidates within the chosen segment, accepting or rejecting based on the ordering of a sequence of random uniform variables. Although the method may look complicated, it is actually very simple to translate both GRAND and SA into fixed-point for the SRD, and the instructions and tables are able to fit in the SRD's local memory, so no storage from the RU is needed.

Table 2 summarises the characteristics of the three generators when implemented in the Ambric architecture. Initial versions of each were prototyped using the Java compiler provided by the Ambric tool-chain, but for performance reasons we hand-assembled the final versions. Due to the simple instruction timing and regular instruction set of the SR and SRD this was not a difficult task, and the effort is justified for such low-level building blocks. The first three columns

	Storage			Computation/Sample			SR+SRD	AM2045
Distn.	Instr.	Data	Bytes	Cycles	Uniform	Multiply	MSamples/s	GSamples/s
Uniform	19	0	38	14.00	0.00	0.00	50.00	8.40
Gaussian	68	64	1300	66.16	2.60	2.32	5.12	0.86
Exponential	48	13	0	41.27	2.53	0.44	7.66	1.29

Table 2: Summary of random number generator performance for the Ambric architecture.

Platform	Name	Process	Die Size	Transistors	Max Power	Clock	Parallelism
		(nm)	(mm^2)	(Millions)	(Watts)	(GHz)	(Threads)
CPU	Intel Core2 QX9650	45	214	820	170	3.00	4
GPU	NVidia GTX 280	65	574	1400	178	1.30	960
MPPA	Ambric AM2045	130	-	-	14	0.35	336
FPGA	Xilinx xc5vlx330	65	600	-	30	0.22	n/a

Table 3: Summary of target device characteristics

show the storage requirements per processor, both for instructions, constant data, and the total number of bytes.

The next three columns show the amount of computation required per generated sampled, measured as the number of cycles (including CPU stalls, but not channel stalls), the number of uniform numbers consumed, and the number of 32x32 multiplies performed. Because the GRAND and SA methods are probabilistic, the number of cycles is the average number: often the cycle count will be lower, but occasionally it will take 200 or more cycles to generate a sample.

The final two columns show the practical performance of the generators, both for one pair of SR and SRD processors, and for an entire Am2045 containing 336 processors, all executing at 350MHz. In the uniform case, both the SR and SRD can execute the same code, so each pair of processors generates two independent random streams. For the Gaussian and exponential distributions, the SR is generating uniform samples, then is passing them to the SRD over a channel for transformation to non-uniform. Because this channel has finite capacity, occasionally the SRD will stall, because it tries to read uniform numbers faster than the SR can fill the channel. However, the SRD code is designed to space uniform reads out as much as possible, so the loss of performance is minimal.

6. FPGA: XILINX VIRTEX-5

The Virtex-5 is typical of a contemporary high performance FPGA, containing a mixture of fine-grained Lookup-Tables (LUTs) and Flip-Flops (FFs), and larger specialised RAM and DSP (multiplier) units. However, it is still possible to use many of the same RNG algorithms that have been developed in software. An additional key requirement for FPGA based RNGs is that generators should produce one output sample per cycle, every cycle: if a generator only produces samples in 99% of cases then the consuming process must be able to handle pipeline bubbles. This introduces additional hardware, and significantly complicates the job of application developers, so we simply discard techniques that cannot produce one sample per cycle.

6.1 Uniform Generation

FPGAs have a natural advantage over CPUs when it comes to binary linear uniform RNGs, due to the availability of very fine-grain binary linear operations. In word-based instruction processors (i.e. CPUs, GPUs, MPPAs), the binary linear state transform must be constructed using the available word-level primitives: in particular, the only means of re-ordering bits is using word-level shifts, or, at best, byte level permutations. This means that consecutive bits in one state tend to affect another group of consecutive bits in the next state. For this reason it is usually not safe to extract more than one word from each multi-word generator state, as there may well be correlations between the two streams.

By comparison an FPGA allows a huge number of very fine-grain transformations to be constructed, as each LUT is capable of implementing a 3 to 6 bit exclusive-or, and the routing network allows extremely complex bit-level mixing to occur. This means that the state transform constructs each bit from a different set of bits from the previous state, so there are no correlations between consecutive bits; in fact, the very notion of consecutive bits does not make sense in an FPGA, as the output can be formed from any permutation of the generator's state.

These ideas led to the development of the LUT-Optimised generator [21], which forms a binary linear recurrence designed specifically for FPGA architectures. As with a word-based recurrence, the generator has a fixed-length binary state, but this is treated as a vector of w bits, rather than as a set of words. At each step of the transform each bit is updated with the exclusive-or of a set of bits from the previous state, so each generator bit maps into one LUT-FF pair. As w is increased, the matrix becomes increasingly sparse, and the quality of the generators increases, while the speed of the generator remains extremely high (limited by the FF-LUT-FF critical path). The practical limit is around $w = 1200$, where it becomes difficult to verify that a given matrix has the maximum period property.

As with all binary linear generators, this method suffers from low linear complexity, as any w bit linear recurrence has a linear complexity of w [11]. This is particularly important in LUT Optimised generators as w is small (compared to software methods such as SFMT [20]), so to meet our requirements we must post-process the bits. The method we use here is based on the idea of stateful non-linear combiners [19], or binary full-adders. We start with a w bit binary generator with state $s_0..s_{w-1}$, and then add an additional $w/2$ bit state vector $c_0..c_{w/2-1}$. The $w/2$ bit output of the

	Period	Inst.	Stms.	RAM	Slices (%)		LUT-FFs (%)		LUTs	FFs	MHz	GSamp./s
Uniform	1024	1	16	0	603	(1.2)	2049	1.0	2049	2048	550	8.80
Gaussian	512	1	1	4	774	(1.5)	2311	1.1	1625	2255	397	0.40
Exponential	288	1	2	4	238	(0.5)	661	0.3	426	593	363	0.73
Uniform	1024	64	1024	0	49102	(94.7)	154268	74.4	152725	151182	253	259.07
Gaussian	512	64	64	256	43537	(84.0)	149532	72.1	105512	146204	189	12.10
Exponential	288	64	128	256	16192	(31.2)	50496	24.4	35456	46144	210	26.88

Table 4: Characteristics of FPGA based random number generators, for a single instance, and for a whole xc5vlx330 device.

adapted generator ($r_0..r_{w/2-1}$) is formed as:

$$c'_i = (c_i \wedge s_i) \oplus (c_i \wedge s_{i+w/2}) \oplus (s_i \wedge s_{i+w/2}) \qquad (10)$$

$$r_i = c_i \oplus s_i \oplus s_{i+w/2} \qquad (11)$$

The resulting generator retains the FF-LUT-FF critical path, but requires $2w$ fully utilised LUT-FF pairs to generate $w/2$ random bits, so it has one quarter the area efficiency of the basic generator. However, this non-linear stage allows all the parallel generated streams to pass tests for linear complexity, meeting our requirements on empirical quality.

6.2 Gaussian Generation

As in software, there are a number of possible choices for Gaussian random number generation, including the Ziggurat method [27], CDF inversion [5], Wallace [15], and the Box-Muller method [13]. However, none of these methods are able to deliver the correct set of characteristics to meet our requirements. The Wallace and Ziggurat methods are rejected immediately due to poor statistical quality, and inability to produce one sample per cycle, respectively.

Inversion based methods can be small and efficient [5], but are unable to provide sufficient output resolution to pass the statistical tests without using a large number of DSP blocks: for the empirical tests for randomness we require 16 bits after conversion to the uniform distribution, which means the Gaussian distribution must have a resolution of *at least* 15 fractional bits, as $\log_2(\Phi(0.5 + 2^{-16}) - \Phi(0.5)) = -14.7$. Tail coverage out to $\pm 8\sigma$ is also required for a good quality generator [25], so the absolute minimum fixed-point output format is 19 bits with 15 fractional bits. Providing an accurate inversion method with this level of accuracy will require either high-degree polynomials, or very large tables.

The Box-Muller method is able to provide high-resolution output samples, but also requires a large number of DSPs. We also found that the most efficient implementation [13] fails tests for goodness of fit. Previous implementations have combined two output samples to make one higher quality output sample [14], which fixes this problem, but also halves the performance of the generator (although still provides one sample per cycle).

To meet all our requirements, including quality, DSP usage, and one sample per cycle, we created a new generator. The core of the generator is a moderately accurate Gaussian approximation, created using a piecewise linear generator [22]. A generator of this type has high resolution, but is not of a high enough quality to pass our statistical tests, so we run eight parallel instances, then add the independent samples together. Due to the Central Limit Theorem the quality of the combined sample is much higher than that of the individual samples, and passes all our tests easily. Because each of the basic generators produces the same distribution, we can share block RAMs between pairs, so only four block RAMs are required for the overall generator.

6.3 Exponential Distribution

The FPGA community has not shown as much interest in the exponential distribution as the Gaussian distribution, mainly because most FPGA-based Monte-Carlo simulations have been discrete-time based. However, as FPGAs become larger and more capable, complete discrete-event simulations in fields such as biochemistry [7] and credit-risk analysis [23] are being accelerated by moving the whole application into the FPGA. Such simulations need a fast high quality source of exponential random numbers, so new methods are beginning to be investigated.

Three Gaussian generation techniques, the Ziggurat, inversion, and Wallace methods, can also be used to generate exponential random numbers, although so far only the inversion method has been used in FPGAs [5]. However, the same arguments against using them also apply for the exponential distribution: lack of resolution, variable throughput, or excessive DSP usage.

To generate the exponential distribution we use the concept of concatenating independent Bernoulli bits [24]. This relies on the fact that each bit of a fixed-point exponential variate is actually an independent Bernoulli bit, i.e. a random bit that has a fixed probability of equalling one or zero. In a hardware generator this property can be used to generate the bits in parallel, allowing high-precision samples to be calculated in a small amount of RAM and logic.

7. GPU: NVIDIA GTX 280

Previous work on GPU based RNGs has found that the Combined Tausworthe with a 4 word state work well, combined with a 32-bit Linear Congruential Generator (LCG) to improve linear complexity [9]. However, after investigating a number of alternatives, we found that the XorShift [16] method was faster than the Combined Tausworthe, and that using an additive post-processing stage (as used in the Ambric uniform RNG) provided the same increase in quality as the LCG, but was faster, The period of the combined generator is $2^{32}(2^{128} - 1)$.

One key area where GPUs differ from CPUs is in the cost of branching, and this directly affects the efficiency of iterative rejection methods such as the Ziggurat method. In a GPU, if one thread takes a branch, then all threads in a warp that didn't take the branch must wait until the single thread is finished. This cripples methods such as the Ziggurat, where the fast path is taken in 99% of cases, and the slow path in the remaining 1%: if there are 32 threads in a warp, then the probability of all threads taking the fast path is $0.99^{32} = 0.72$. So in 30% of cases, the warp will end

	CPU	GPU	MPPA	FPGA
Uniform	**SFMT** [20] : 128-bit SIMD ops., and 2500 bytes of state.	**XorShift** [16] : 32-bit xor and multi-bit shifts, 5 state words, additive post-processing.	**Custom** : 1-bit shifts, byte permutations, 5 state words, additive post-processing.	**LUT-Opt.** [21, 19] : 1024-bit state, bit-wise linear ops, per-bit post-processing.
Gaussian	**Ziggurat** [17] : Scalar rejection method; fast path in 99% of cases, uses transcendental functions in slow 1% path.	**Box-Muller** [3] : Direct transform, calls (builtin) log, sqrt, and sin every time.	**GRAND** [4] : Scalar rejection, no transcendentals, small tables.	**Piecewise Linear** [22] : Table lookup, comparisons, additions; one sample per cycle.
Exponential		**Inversion** : Apply fast builtin log to uniform sample.	**SA** [1] : Scalar rejection, no transcendentals, small tables.	**Bernoulli Bits** [24] : Comparisons, uniform bits, tables.

Table 5: Comparison of generation methods used in different architectures.

	Performance (GSample/s)				Efficiency (MSample/joule)			
	CPU	GPU	MPPA	FPGA	CPU	GPU	MPPA	FPGA
Uniform	4.26	16.88	8.40	259.07	15.20	140.69	600.00	8635.73
Gaussian	0.89	12.90	0.86	12.10	3.17	107.52	61.48	403.20
Exponential	0.75	11.92	1.29	26.88	2.69	99.36	91.87	896.00
Geo Mean	1.42	13.75	2.10	43.84	5.07	114.55	150.21	1461.20

	Relative Mean Performance				Relative Mean Efficiency			
	CPU	GPU	MPPA	FPGA	CPU	GPU	MPPA	FPGA
CPU	1.00	9.69	1.48	30.91	1.00	9.26	18.00	175.14
GPU	0.10	1.00	0.15	3.19	0.11	1.00	1.95	18.92
MPPA	0.67	6.54	1.00	20.85	0.06	0.51	1.00	9.73
FPGA	0.03	0.31	0.05	1.00	0.006	0.05	0.10	1.00

Table 6: Comparison of absolute performance and efficiency of RNGs across platforms.

up waiting for one or more threads to execute the slow path, severely slowing down the overall processing rate.

Instead, it is more appropriate to use the built-in fast transcendental functions, which allow the Box-Muller method to be used for the Gaussian distribution, and the inversion method to be used for the exponential distribution. Importantly, neither method uses rejection, iteration, or branching, so there is no thread divergence within the warp.

8. CPU: INTEL CORE2 QX9650

Traditional CPUs present the least challenge when selecting generators, due to the intense research into software RNGs over the last 50 years. Until recently the clear leader in uniform RNGs was the Mersenne Twister, but this has been recently superseded by SFMT [20], which uses the same principles, but uses 128-bit wide SIMD instructions to achieve a higher generation rate.

There is also a clear leader in non-uniform generation, where the Ziggurat [17] method is used for both exponential and Gaussian distributions. Although the Ziggurat method is intrinsically scalar, and so cannot take advantage of SIMD instructions, it is designed specifically to match the operation costs of CPUs, and provides excellent performance.

9. RESULTS

Table 3 provides an overview of the key characteristics of each platform (in some cases there are no figures publicly available), while Table 5 summarises the different methods we found to be optimal for each architectures. One of the striking aspects is that no algorithm is used on more than one architecture, even when platforms are conceptually quite similar, such as CPUs and Ambric. One common feature across platforms is that binary linear recurrences were used for all uniform RNGs, but we actually used four different types of generator within this class. For non-uniform generation the differences are more marked. In the FPGA and GPU it is more efficient to use direct transforms, while in Ambric and CPU the cost of branching is lower, so it makes sense to use rejection. However, within these two classes another difference is found, as the CPU and GPU prefer methods that use transcendental functions, while in Ambric and FPGA it is necessary to avoid such functions.

The top of Table 6 summarises the absolute performance of each platform, both for each individual generator, and for the geometric mean across all three generators. We also include an estimated power efficiency, measured in millions of samples per joule. Due to the difficulty in measuring power for a given workload, we calculate efficiency using the peak power consumption of each device, ignoring supporting infrastructure such as RAM, hard disks, networks etc.

The bottom part of Table 6 presents the relative performance and power efficiency of the platforms for the geometric mean across all three generators. The FPGA provides the highest performance level, although it is only three times that of the GPU, and the cost of an xc5lv330 FPGA is many times that of a GTX 280 GPU. Perhaps the more surprising performance result is that the 350MHz MPPA Am2045 device provides better performance than a 3GHz Quad-Core CPU. The results are more conclusive when efficiency is compared, as the FPGA provides an order of magnitude more performance per joule than any other platform, and over 250 times that of the CPU. However, it should be remembered that these benchmarks only test one small aspect of a real

Monte-Carlo application; in practise the random numbers have to drive *something*, and it is entirely likely that CPUs and GPUs will make up ground in the non-RNG portion of the application.

10. CONCLUSION

This paper has looked at four different platforms for parallel computing: multi-core CPUs, GPUs, MPPAs, and FPGAs. For each platform we have attempted to identify the most appropriate RNG for generating the uniform, Gaussian and exponential distribution, taking into account the characteristics and architecture of each device. The surprising result is that each platform requires a different approach to random number generation, even amongst those platforms based on instruction processors: methods which are highly efficient in scalar oriented CPUs do not work well in wide issue GPUs, nor in memory limited MPPAs.

11. ACKNOWLEDGEMENTS

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.

12. REFERENCES

[1] J. H. Ahrens and U. Dieter. Computer methods for sampling from the exponential and normal distributions. *Commun. ACM*, 15(10):873–882, 1972.

[2] Ambric, Inc. *Am2000 Family Architecture Reference*, May 2008.

[3] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[4] R. P. Brent. Algorithm 488: A Gaussian pseudo-random number generator. *Commun. ACM*, 17(12):704–706, 1974.

[5] 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*, 15(8):952–962, 2007.

[6] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag New York, 1996.

[7] M. Y. et. al. FPGA implementation of a data-driven stochastic biochemical simulator with the next reaction method. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 254–259, 2007.

[8] G. E. Forsythe. Von neumann's comparison method for random sampling from the normal and other distributions. *Mathematics of Computation*, 26(120):817–826, 1972.

[9] L. Howes and D. Thomas. *GPU Gems 3 : Efficient Random Number Generation and Application Using CUDA*, chapter 37. Addison-Wesley, 2007.

[10] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.

[11] P. L'Ecuyer. *Elsevier Handbooks in Operations Research and Management Science: Simulation*, chapter 3 : Random Number Generation, pages 55–81. Elsevier Science, 2006.

[12] P. L'Ecuyer and R. Simard. TestU01 random number test suite. www.iro.umontreal.ca/~simardr/indexe.html, 2007.

[13] 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*, 55(6):659–671, 2006.

[14] 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.

[15] 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.

[16] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[17] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[18] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.

[19] R. A. Rueppel. Correlation immunity and the summation generator. In *CRYPTO 85*, pages 260–272, 1986.

[20] M. Saito and M. Matsumoto. SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte-Carlo and Quasi-Monte Carlo Methods*, pages 607–622, 2006.

[21] D. B. Thomas and W. Luk. High quality uniform random number generation using LUT optimised state-transition matrices. *Journal of VLSI Signal Processing*, 47(1), 2007.

[22] D. B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. *IET Computers and Digital Techniques*, 1(4):312–321, 2007.

[23] 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.

[24] D. B. Thomas and W. Luk. Sampling from the exponential distribution using independent bernoulli variates. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2008.

[25] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor. Gaussian random number generators. *ACM Computing Surveys*, 39(4):11, 2007.

[26] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software*, 22(1):119–127, 1996.

[27] 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*, pages 275–280. IEEE Computer Society Press, 2005.