GPU Optimised Uniform Random Number Generation

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

Abstract

The recent trend for software random number generators has been to develop extremely long period generators, trading off increased memory per generator against lowered computational cost per generated number. However, emerging fine-grain architectures such as GPUs have a different balance of resources, offering much cheaper computation, at the expense of smaller and comparatively slower memories. In particular, it is not feasible to dedicate large amounts of generator state to each thread, as each register or portion of fast local memory used for generator state has a direct and negative impact on the process being driven by the random numbers. This paper presents a random number generation approach designed for such architectures, which takes advantage of the fine-grain SIMD parallelism, by using multiple threads to co-operatively advance a shared random number generator from which each thread can then consume it's own stream of numbers. Each thread contributes a small portion of state to the overall generator, but the overall period is derived from the combined state size. The proposed generator the idea of binary linear recurrences used in many popular generator such as the Mersenne Twister, but rather than constructing a relatively sparse recurrence matrix optimised for FIFOs, the fine-grain parallelism is used to construct a dense matrix, where each bit of state is transformed on every pass.

1. Introduction

2. GPU Architecture

In this section we introduce the basic concepts and architectures in use by current GPUs. Because each GPU architecture has a number of differences we focus on the broad concepts rather than low-level details, presenting an abstract conceptual model. We use the nomenclature of CUDA, as at the time of writing this appears to be the only well-defined language/API, but the concepts broadly apply to both NVidia and AMD GPUs (at least as far as they are relevant to the RNG technique applied here).

Current GPU architectures rely on three main concepts to achieve high computational performance:

Thread Batching: Multiple threads are grouped together to create *warps*. All threads within a warp are scheduled and executed together, so if an instruction must be applied to one thread in a warp, then the instruction must be executed for all threads in the warp. This is essentially a SIMD (Single-Instruction Multiple-Data) approach, with the standard SIMD benefits of increasing the number of operations performed per instruction, while decreasing the scheduling overhead per operation. In addition, GPUs are able to selectively control whether the result of each operation is actually written back to the thread state, allowing each thread in a warp to follow a different path through the program.

High Memory Throughput: High memory throughput is achieved by having as many memory banks as there are threads in a warp. This allows all threads in a warp to perform independent random memory accesses, and as long as each thread addresses memory elements in a different bank, then all memory operations for the warp are executed in one pass. If two or more threads attempt to access addresses located in the same bank then a bank conflict occurs, which is automatically resolved by servicing each thread using multiple passes. This approach is employed at two-levels: each processor has a small dedicated local memory, providing shared storage for all threads executing on that processor; in addition, a much larger global memory is shared by all processors, allowing all threads within the GPU to access the same storage.

Thread-level parallelism: Although the ALUs and memory systems provide high throughput, they still have significant latencies. This varies from tens of cycles for ALU operations and local memory accesses, up to hundreds of cycles for accesses to the global memory shared by all processors. To avoid processor stalls, without resorting to complicated hazard detection or speculative execution, the processor supports large numbers of warps (groups of threads). When one warp

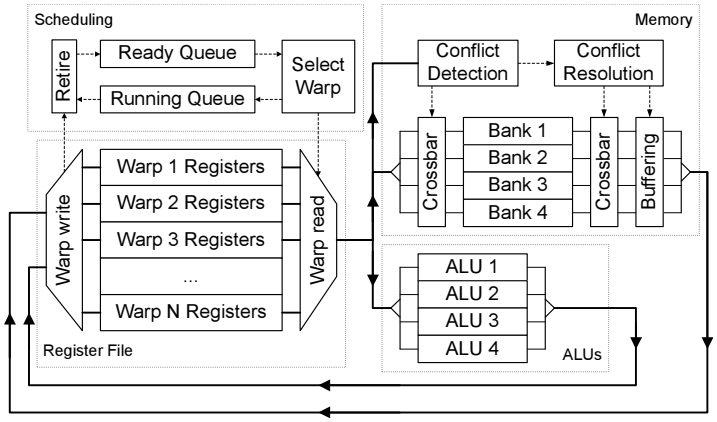

Figure 1. Overview of GPU architecture, using four threads per warp.

stalls, e.g. due to a read-after-write hazard or a memory access, the processor can schedule instructions from another warp. As long as the processor is given a large enough number of warps to execute, then even the very long latency of global memory accesses can be hidden.

Figure 1 gives an overview of the GPU architecture, showing the three concepts at work. For simplicity this model uses four threads per group, but in practical architectures this would be higher, e.g. 32 for the NVidia GT200 series. The bottom left shows the register file, which contains the SIMD register set for each warp (or equivalently, contains a set of scalar registers for the threads contained within the warp). The number of warp contexts may be fixed, as in Larabee, where there are four warp contexts, or variable, as in NVidia devices, where there are a fixed number of registers which can be partitioned into a variable number of warp contexts.

Above the register file is the instruction scheduler, which is responsible for scheduling one of the warps for execution in each issue slot. The scheduler maintains two sets of warps: a ready queue, containing those warps which are ready to execute; and a running queue, which lists warps that are currently stalled due to an executing instruction that has not yet completed (the scheduler may track true dependencies between instructions, or may just assume that any instruction in flight stalls the warp). Once an instruction completes and is written back to the warp context, then the warp can be moved from the running queue back to the ready queue.

When the instruction scheduler selects a warp for execution, the correct set of (SIMD) input registers are selected from the warp's context in the register file, then dispatched to either the memory or ALUs. Both memory and ALU operations are applied to each independent lane (thread) within the SIMD inputs, then the result is written back to the appropriate register within the warp context. When only a subset of threads within the warp are active, e.g. some, but not all, threads in a warp to a branch, then the processor will only write-back the results for those threads that are currently active.

The ALUs are likely to be pure pipelines, with a fixed throughput and latency, but the memories will have a variable latency. The first reason for variations is that when threads access the shared global memory they are competing for access with all other processors in the GPU (and potentially with other warps in the same processor). If the ratio of global memory accesses to computation is high across the GPU, then the global memory may become a bottleneck, as the majority of the time warps will be queueing to share access.

The second reason for variable memory latency is due to bank conflicts. As mentioned earlier, high memory throughput is achieved by splitting memory into banks, allowing all threads within a warp to perform memory accesses in a single pass *as long as each thread's access maps to a different bank*. If multiple threads attempt to access the same bank, then a bank conflict occurs, and the threads must access the conflicted bank in a number of serial passes. In the worst case, where all threads access the same bank, this takes as many passes as there are threads in the warp. Bank conflicts can occur for both local and global memory, and can severely reduce the throughput of both memory systems.

From this architecture we can draw a number of guiding principles that can guide us in developing RNGs:

1. Global memory accesses are a large potential performance bottleneck, limiting scalability across processors, and should be avoided (particularly in library components like RNGs).

2. The only way of sharing data between threads is via memory, and because we are avoiding global memory, the sharing must occur via local memory.

3. To achieve high-performance all memory accesses must have no bank conflicts.

4. Minimising the number of dedicated registers per thread for RNGs reduces register pressure for the rest of the application.

5. Minimising the number of dedicated registers per thread for RNGs may also allows a larger number of warps to execute in parallel (in NVidia architectures, for example), which allows more thread-level parallelism and so minimises stalls in the application.

6. Minimising the amount of local memory required per thread for RNGs will free up more local memory space for the rest of the application, and may also allow more warps to execute on the processor (see previous point).

7. During memory accesses, each thread within the warp will complete it's access before any other memory accesses are scheduled, so barriers are not needed to ensure correct ordering of memory accesses between threads within a warp (however, barriers are needed to ensure memory access ordering between threads in different warps).

From these we can draw a number of conclusions about how we can create RNGs in a GPU:

- We cannot use registers to hold a large RNG state for each thread, as it increases register pressure on the rest of the application, and reduces the potential for warp-level parallelism.

- We cannot use local memory to hold a large RNG state for each thread, as it would require multiple words of shared storage per thread, and limit the amount of shared memory available to the rest of the application.

- There is no lack of storage in global memory, but storing RNG state there would cause a potential bottleneck, because each thread would have to perform at least one read and one write to global memory per generated number.

- The only feasible way of retaining performance when using a large RNG state is to spread the state across multiple threads.

- The only way for threads to share data is via memory, and for performance reasons, this must be local memory.

- For data sharing without barriers, the RNG state must only be shared between threads in the same warp.

3. Generator Framework

There are a number of ways to fulfil these criteria, but we have chosen the most direct form, where each warp of k threads shares one RNG, using k words of state stored in local memory (i.e. one word per thread). On each update the threads read one or more words from the shared state, transform these words to produce a new word, then write this word back into the shared RNG state. The output of the RNG is the set of words produced after each update, resulting in k streams of random numbers, one for each thread.

The state of the generator consists of k words with w bits each, resulting in an $n = wk$ bit RNG state shared by the whole warp. The generator starts in some initial state $\mathbf{s}_0 \in \mathbb{F}_2^n$, and after each update the state is modified, leading to a sequence of states $\mathbf{s}_1, \mathbf{s}_2, \ldots$. Each update is performed by a deterministic transition function f:

$$\mathbf{s}_{i+1} = f(\mathbf{s}_i), \quad \text{where } i \geq 0 \tag{1}$$

Because the state is finite, the sequence of states must eventually repeat, with the period $p > 0$ defined by:

$$\min_p [\exists i : \mathbf{s}_{i+p} = \mathbf{s}_i] \tag{2}$$

Although at the abstract level the generator uses an n bit state with a single transition function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2^n$, the actual implementation uses k threads applying k different transition functions $f_1..f_k$ to k state words $x_1..x_k$:

$$\mathbf{x}_{[i,j]} = \mathbf{R}_j \mathbf{s}_i, \quad \text{where } i \geq 0,\ 1 \leq j \leq k \tag{3}$$

$$\mathbf{x}_{[i+1,j]} = f_j(\mathbf{s}_i) \tag{4}$$

where $\mathbf{R}_j$ (with $1 \leq j \leq k$) is a $w \times n$ matrix that selects the j^{th} contiguous w-bit word from within a vector of n bits:

$$\mathbf{R}_j = \begin{bmatrix} \mathbf{R}_{[j,1]} & \mathbf{R}_{[j,2]} & \ldots & \mathbf{R}_{[j,k]} \end{bmatrix}$$
$$\text{where } \mathbf{R}_{[j,i]} = \begin{cases} \mathbf{I}_w, & \text{if } j = i \\ \mathbf{0}_w, & \text{otherwise} \end{cases} \tag{5}$$

At this point we have significant freedom in how we implement $f_1..f_k$; all the function have to do is read one or more words from the current state, and combine them to create one new word for the next state. In this

w	$\mathbb{N}$	Number of bits in each word (e.g. 32 or 64).
k	$\mathbb{N}$	Number of threads in a warp (and no. of w-bit words in shared memory).
g	$\mathbb{N}$	Number of banks in each RAM ($k \mod g = 1$).
n	$\mathbb{N}$	Number of bits in the RNG state ($n = wk$).
h	$\mathbb{N}$	Number of read-shift stages used in the RNG.
$\mathbf{s}_i$	$\mathbb{F}_2^n$	State of the RNG after the i'th update ($s_{i+j} = \mathbf{A}^j s_i$).
$\mathbf{x}_{i,j}$	$\mathbb{F}_2^w$	Output word from stream j after the i'th update ($\mathbf{x}_{i,j} = \mathbf{R}(j)\mathbf{s}_i$).
$\mathbf{A}$	$\mathbb{F}_2^{n,n}$ matrix	Binary transition matrix for the RNG.
$\mathbf{R}_j$	$\mathbb{F}_2^{w,n}$ matrix	Reader matrix: Extracts word j from RNG state vector.
$\mathbf{S}_j$	$\mathbb{F}_2^{w,w}$ matrix	Shift matrix: Shift left for negative j, right for positive.
$\mathbf{W}_j$	$\mathbb{F}_2^{w,n}$ matrix	Write matrix: Determines the new value for word j in the next RNG state ($x_{i,j} = W_j s_{i-1}$).
$\mathbf{q}$	$\mathbb{N}^{h,k}$ matrix	Entry $\mathbf{q}[i,j]$ is the word to be read at stage i while calculating word j.
$\mathbf{m}$	$\mathbb{F}_2^{h,k,w}$ matrix	Entry $\mathbf{m}[i,j] \in \mathbb{F}_2^w$ is the mask applied at stage i while calculating word j.
$\mathbf{z}$	$\mathbb{N}^{h,k}$ matrix	Entry $\mathbf{z}[i,j]$ is the shift to be applied at stage i while calculating word j.

work we have made the choice to use binary linear operations. This is because these are both efficient to implement in GPUs, and because the theory of binary linear RNGs is well established and has seen significant practical use. The two primitive operations for constructing binary linear functions in a word-based architecture are bit shifting and bit-wise masking (i.e. bit-wise and with a constant).

We can represent shifts left or right with a set of $w \times w$ matrices $\mathbf{S}_{-w}..\mathbf{S}_w$, where the subscript indicates the size of the shifts. Negative shifts correspond to a left shift (i.e. the most significant bits are lost), while positive shifts are unsigned right shifts. [1] We can define the shift matrices in terms $\mathbf{S}_r$ and $\mathbf{S}_l$, the primitive 1-bit right and left shift operations:

$$\mathbf{S}_i = \begin{cases} \mathbf{S}_l^{-i}, & \text{if } i < 0 \\ \mathbf{S}_r^{i}, & \text{otherwise} \end{cases} \tag{6}$$

where $\mathbf{S}_r$ ($\mathbf{S}_l$) has ones on its sub-diagonal (super-diagonal), and zeros elsewhere:

$$\mathbf{S}_r = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \quad \mathbf{S}_l = \begin{bmatrix} 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{7}$$

The mask operation is defined by the w-bit mask $\mathbf{m} \in \mathbb{F}_2^w$ to be applied, and results in a $w \times w$ matrix $\mathbf{M}_\mathbf{m}$ that only allows through the selected bits:

$$\mathbf{M}_\mathbf{m} = \begin{bmatrix} \mathbf{m}[1] & 0 & \dots & 0 & 0 \\ 0 & \mathbf{m}[2] & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \mathbf{m}[w-1] & 0 \\ 0 & 0 & \dots & 0 & \mathbf{m}[w] \end{bmatrix} \tag{8}$$

Using shifts and masks we can define binary linear generators by using multiple stages, where each stage reads one word from the current state, which is then masked and shifted. The results of all these stages are reduced using exclusive-or to produce one word, building up a single $w \times n$ matrix $\mathbf{W}_j$, which describes the next value of word j in terms of the current state:

$$\mathbf{W}_j = \bigoplus_{i=1}^{h} \mathbf{S}_{\mathbf{z}[i,j]} \mathbf{M}_{\mathbf{m}[i,j]} \mathbf{R}_{\mathbf{q}[i,j]} \mod 2 \tag{9}$$

$$f_j(\mathbf{s}) = \mathbf{W}_j \mathbf{s}, \quad \text{where } f_j : \mathbb{F}_2^n \mapsto : \mathbb{F}_2^w \tag{10}$$

$$= \bigoplus_{i=1}^{h} \mathbf{S}_{\mathbf{z}[i,j]} \mathbf{R}_{\mathbf{q}[i,j]} \mathbf{s} \mod 2 \tag{11}$$

The equivalent C-code for the word update function f_j is shown in Listing 1 (adjusted from 1 to 0 based indices).

Each generator within this framework is described by the tuple $(w, k, h, \mathbf{q}, \mathbf{m}, \mathbf{h})$. The first three parameters are integer scalars defining the overall "shape" of the generator, with w and k describing the "width" of the update function (the degree of parallelism), and h defining the "depth" (the number of stages or iterations per update). The other three parameters are $wk \times h$ tables of constants, describing what operations to apply at each of the h updates for the k words. From this tuple we can derive the set of word update matrices $\mathbf{W}_1..\mathbf{W}_k$, and so

[1] In this work we only consider logical right shifts that introduce zeros, but arithmetic shifts that shift in the sign could also be used. We avoided arithmetic shifts because they result in the most-significant bits from the previous state having a much larger effect on the next state, which is arguably the opposite of what we want.

Listing 1. Word Update Function

```
const unsigned Q[H][K] = {...};
const unsigned M[H][K] = {...};
const int Z[H][K] = {...};

unsigned f(unsigned j, const unsigned *s)
{
  unsigned acc=0;
  for(unsigned i=0;i<H;i++){
    unsigned tmp=s[ Q[i][j]] & M[i][j];
    if(Z[i][j]<0)
      tmp=tmp << -Z[i][j];
    else
      tmp=tmp >> Z[i][j];
    acc = acc ^ tmp;
  }
  return acc;
}
```

determine the transition matrix $\mathbf{A}$ for entire generator:

$$\mathbf{A} = \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \vdots \\ \mathbf{W}_k \end{bmatrix} \tag{12}$$

Within this framework it is possible to describe any desired recurrence matrix $\mathbf{A}$ - each bit of each output word can be constructed in at most n steps, by selecting the (at most n) source bits one-by-one, then shifting them into the correct position. To construct the whole output word this must be repeated w times, so up to $h = wn = w^2k$ stages might be required. In practise the number of stages would probably be much lower, but the idea is not to pick a matrix $\mathbf{A}$, then find the best generator that produces it; instead, we define a family of generators, then try to find one that produces the best matrix $\mathbf{A}$.

4. Generator Parametrisation

There are three competing concerns when defining the parameter space. First, we wish to maximise the possibility of finding maximum period generators - if we define a very small search space then it may not contain any such generators, while if we define an enormous search space the probability of finding them may be very low. Second, we want to maximise the statistical quality, by reducing the risks of intra- and inter-stream correlations. Finally, we need to maximise performance, by reducing the number of operations required per transformation. In this section we explain how we have attempted to balance these competing concerns, by defining a search space that strikes an acceptable balance between all three.

There are a number of necessary (but not sufficient) conditions for a finding maximal period generators. The simplest is that every bit of the current state must be used at least once in the construction of the next state (in the simplest scenario, one bit in the next state is a direct copy of a bit from the previous state). In terms of the matrix $\mathbf{A}$, this means that each column of $\mathbf{A}$ cannot be all zero. Another trivial requirement is that every bit in the next state must be formed from some combination of bits in the previous state: no row of $\mathbf{A}$ can be all zero. A more complex requirement is that each bit in the current state must eventually effect all bits in some future state: informally it must be possible to "get to" each bit from every other bit, or more formally:

$$\forall j, j' \in 1..k : \exists i > 1 : \mathbf{A}^i[j, j'] = 1 \tag{13}$$

We can meet these conditions by ensuring that we restrict the search space according to the following conditions:

$$\text{sort}(\mathbf{q}[i]) = 1..k \tag{14}$$
$$\mathbf{q}[i, j] \neq \mathbf{q}[i', j] \tag{15}$$
$$\mathbf{m}[1, j] = \mathbf{m}[2, j] = [11...11]^{\mathrm{T}} \tag{16}$$
$$\lceil -w/2 \rceil < \mathbf{z}[1, j] < 0 \tag{17}$$
$$0 < \mathbf{z}[2, j] < \lfloor w/2 \rfloor \tag{18}$$
$$i \neq i' \rightarrow (\mathbf{q}[i, j] = \mathbf{q}[i', j] \leftrightarrow \mathbf{z}[i, j] \neq \mathbf{z}[i', j]) \tag{19}$$

where $1 \leq i, i' \leq h$ and $1 \leq j \leq k$.

The first two constraints ensure that each state word is read exactly once in each stage (i.e. $\mathbf{q}[1]$ and $\mathbf{q}[2]$ are permutations of the integers $1..k$), and that no thread reads the same word in multiple stages. The next constraint specifies that no masking occurs in the first two rounds, so the words are passed straight through. Constraints four and five ensure that all threads apply a left shift in the first stage, and a right shift in the second stage, with the total shift less than half the word width in both cases.

The first five constraints ensure that each new word is constructed from the top and bottom bits of two different words, and that every bit from the previous state is introduced into the new state by the first two stages. In addition, it guarantees that each new word contains an overlap, so there is at least minimal mixing of bits from previous states. The final constraint ensures that this mixing cannot be undone by future stages, by stating that the pairs of sources and shifts used in the first two stages are never used again.

These conditions provide a search space that is reasonably likely to contain maximal period generators, but we still have a number of free parameters. We now define three different classes of generator, which vary

in the degree of expected statistical quality, and in the performance characteristics.

5. Evaluating Generators

The RNG template suggested in the previous section has a large number of free parameters, all of which must be chosen in a way that both provides maximum period, and produces a high-quality stream. Finding maximum period generators is actually rather easy: randomly selecting generators from a search space results in a maximum period generator in minutes (for $n \leq 2048$), so a large number of potential RNGs can be quickly generated by using multiple CPUs in parallel.

However, selecting the "best" generators from within this set of maximum period generators is more difficult. For typical architectures (e.g. $k \geq 16$, $w \geq 32$, $n > 512$) the generators are effectively equivalent from the viewpoint of empirical batteries of tests such as Crush and Big-Crush - all tests are passed convincingly, except for the Matrix Rank and Linear Complexity tests which all linear generators fail. Increasing the number of samples tested is also not an option, as running Big-Crush already takes hundreds of times longer than finding a maximum-period generator.

If we want some means of comparing generators we have to turn to theoretical measures of quality, which provide some kind of quality metric based on theory, rather than by looking at streams of random numbers. The standard metric for binary linear generators is *equidistribution*, which measures the "even-ness" of the generator's entire output sequence, when successive tuples are treated as multi-dimensional points. However, equidistribution usually assumes only one sample per update step, so it needs to be modified to support the parallel output streams of a GPU based generator.

5.1. Equidistribution for Multi-Stream RNGs

Equidistribution is a theoretical measure of random number generator quality, which considers the properties of small sub-sequences of the generator's *entire* output stream: even though the generator may have a period of $2^{1024}-1$, we can still use equidistribution to say something meaningful about the quality of all outputs without having to generate and examine the entire stream (which is clearly impractical). It is important to bear in mind that equidistribution can not guarantee quality at a local level within the stream, but in general better equidistribution at a global level tends to result in better quality at a local level.

An intuitive explanation of (t,l) equidistribution is to consider a concrete case: consider a generator which contains $n = 512$ state bits, has a maximal period of $2^5 12-1$, and produces an output stream of $w = 32$ bits. If we determine that this generator is is distributed to full resolution (i.e. $l = 32$) over 20 dimensions ($t = 20$), then the generator is able to produce every possible subsequence of 20 words, and every possible 20 word subsequence occurs exactly the same number of times during the entire An alternate view is that if an observer sees 19 consecutive outputs of the generator, they are completely unable (by any means, no matter how complicated) to predict anything about the next output word. However, a sequence of 20 numbers *does* allow the observer to say something about the next (although not necessarily to predict it exactly). Similarly, if the generator is equidistributed to a resolution of $l = 2$ with dimension $t = 250$, then the two most significant bits of each 250 word sub-sequence will take on every possible pattern over the entire sequence of the generator.

The standard approach is to find generators where $\Delta_\infty = \Delta_1 = 0$, which are called maximally equidistributed (ME) RNGs. However it is not always possible to find ME RNGs, either because none exist within some exhaustively searched parameter space, or because ME parametrisations are so infrequent that it is impossible to find one by random search. In these cases the informally observed standard has been to search for generators with the lowest values of Δ_∞, then to choose from within that set the generator(s) with the lowest Δ_1 (i.e. sort first by worst-case gap, then sort by sum of gaps).

However, looking just at the gaps ignores the relative importance of gaps at different resolutions. If we consider the case where $k = 32, w = 32$, then the maximum possible dimension at $l = 2$ and $l = 32$ are $t_2^* = 512$ and $t_{32}^* = 32$. If we are now told that a generator has $\Delta_\infty = 16$, we really need to know at which resolution this gap occurs: if the worst case occurs for $l = 2$ then the generator is still equidistributed over 496 dimensions, or 97% of the maximum possible, while if the worst case occurs for $l = 32$ then the generator only achieves 50% of the maximum possible. Clearly it is in some sense more optimal if the same size gap appears at lower resolutions.

The parameter space we have defined for GPU generators does not produce ME RNGs, even for very large searches, but it does produce many generators with $\Delta_\infty = 1$ and small values of Δ_1. To select the "best" generator from among these generators we use a metric which provides a single score, taking into account both the size of the dimension gaps, and the resolution

at which they occur:

$$G = -\sum_{l=1}^{l=w} \ln\left(t_l / t_l^*\right) \tag{20}$$

This score behaves intuitively, providing a non-negative score for all generators, with a minimum score of zero iff the generator is ME. In addition, it distinguishes between generators with identical values of Δ_∞, assigning lower scores when gaps occur at lower resolutions.

Another complication when considering the equidistribution of multiple streams from an RNG is that not all streams have the same equidistribution - one stream may be ME, but another might have a large dimension gap. For this reason we actually calculate the equidistribution across each stream, then use the dimension gap of the worst stream at each resolution. This means that when we quote the equidistribution for GPU generators, we are stating that each stream has that level of equidistribution *or better*.

The standard equidistribution methodology works well for existing software generators, because only one w-bit word from each successive generator state is returned, so only the distribution of those w bits is important. The equidistribution of most GPU generators also appears to be intrinsically quite good, with typical values of Δ_∞ from 2 to 5 for randomly selected generators with $k = 32, w = 32$ ($n = 1024$). However, in the generators suggested here, all n bits of the generator are consumed after every update, which presents a more worrying prospect than correlations within the stream. Consuming all k state words from the recurrence is key to the performance of the GPU generator, but also its biggest potential weakness, so we need to minimise the risk of inter-word correlations.

5.2. Mutual Equidistribution as a Measure of Stream Correlation

The conventional measure of equidistribution looks at the distribution of the l most significant bits of one word over t successive outputs, but we can extend this to the multi-word case by also considering the distribution across all k words. This adds an extra dimension to the existing t dimensional partition, with k (rather than l) possible values, so the number of partitions increases from 2^{tl} to $k2^{tl}$. If each cell in this extended partition contains the same number of tuples, then we consider the generator to be $(t,l)_k$ distributed, or mutually equidistributed.

An alternate way of looking at this is to take the l most significant bits from each of the GPU RNG's k output streams (state words), and to concatenate them to create a new stream of lk-bit words. If (and only if) this new stream is (t,lk)-distributed, then the generator's k output streams are also $(t,l)_k$-distributed.

Listing 2. Correlated RNG

```
unsigned CorrelatedRng(unsigned j, unsigned *s)
{
  sync();
  unsigned acc=(s[Q[0][j]]<< Z0)^(s[Q[1][j]] >> Z1[j]);
  sync();
  return (s[j]=acc);
}
```

For a given resolution l the maximum possible dimension t for which an unbuffered generator can be $(t,l)_k$ distributed is:

$$t_{l,k}^* = \lfloor n/lk \rfloor = \lfloor w/l \rfloor \tag{21}$$

This means that $t_{1,k}^* = w$, so (all other things being equal) architectures using 64-bit words have an advantage over those using 32-bit words, at least in terms of mutual equidistribution.

In a buffered generator (where state is stored in registers as well as shared memory), the maximum possible mutual equidistribution is increased. If r is the number of state registers per generator, then the maximum possible dimension increases to:

$$t_{l,k}^* = \lfloor n/lk \rfloor = wk(1+r)/lk\lfloor w(1+r)/l \rfloor \tag{22}$$

This means that adding one state register doubles the maximum possible dimension, while adding three registers quadruples it.

6. Generator Families

It is now possible to define a number of generator families, which provide different performance and quality characteristics, and within which it is reasonably likely to be able to find generators through random search.

6.1. Correlated RNG

The first class of generator simply fixes $h = 2$, so only includes the two stages necessary to achieve maximal period. This represents the absolute minimal generator possible, both in terms of memory operations and bit-wise operations: two reads, two shifts, an exclusive-or, and one write. The big drawback of this generator is that the most and least significant bits of each new word are a verbatim copy of bits from a word in the previous state.

Figure 2 illustrates this problem: because we require both a left and a right shift to create a full-period

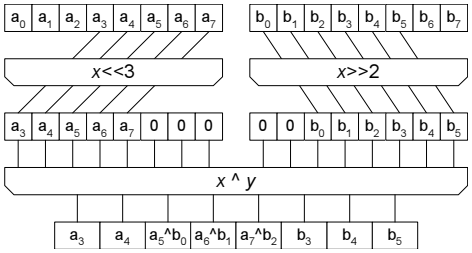

Figure 2. When forming a new word from just two shifted words, the top and bottom bits are direct copies of bits from the previous word.

generator, only the central bits of the new word are "new". Both the most significant and least significant bits will already have been consumed by another thread, so there is a significant risk that an unwanted correlation will be introduced between supposedly independent threads. For example, if treated as random bits, the hamming weight of the new word will be strongly correlated with the hamming weights of the two source words. More worryingly, if the words are treated as integers, then the new integer will be very strongly correlated with the first (left shifted) source word modulo some binary power, i.e. $x_{i+1,j}$ and $x_{i,\mathbf{q}[1,j]} \mod 2^{w-\mathbf{z}[1,j]}$ will be very correlated.

Whether this correlation between streams is important is completely dependent on the consuming application. In cases where the threads are very unsynchronised, for example if each thread is independently executing multiple independent simulation runs with a variable number of time-steps, then the correlation may have no detectable effect. In these cases it may be worth using the correlated generator, as it is extremely fast, and the quality of each individual stream is very good. However, if the threads are working together on one simulation, or are starting and executing independent simulations in lock-step (for example, generating fixed-length time-series), then the correlation could completely ruin the results in an obvious way (the best case), or subtly corrupt the results in a difficult to detect way (the more worrying case).

The probability of finding a full period generator seems to be related to the hamming weight of the recurrence matrix **A**: more ones in the matrix increases the probability of a primitive characteristic polynomial. This presents a problem when looking for correlated generators with large left and right shifts, as the resulting matrix is very sparse, so only a very small fraction of candidate generators are actually full period. One solution is to restrict the shifts to small values (between 1 and 3), but this has a serious effect on mutual equidistribution - it is not uncommon to find generators that have close to the worst possible mutual equidistribution.

Listing 3. Three Input RNG

```
unsigned ThreeInputRng(unsigned j, unsigned *s)
{
  sync();
  unsigned acc=(s[Q[0][j]] << Z0) ^ (s[Q[1][j]] >> Z1[j]);
  acc ^= s[Q[2][j]];
  // Alternate: Could also shift extra input
  // acc ^= s[Q[2][j]] << Z2;
  // Optional: Could extend to four input, etc.
  // acc ^= s[Q[3][j]];
  sync();
  return (s[j]=acc);
}
```

Listing 4. Slow Memory RNG

```
unsigned SlowMemoryRng(unsigned j, unsigned *s)
{
  sync();
  unsigned t0=s[Q[0][j]], t1=s[Q[1][j]];
  unsigned acc=(t0 << Z0[j]) ^ (t1 >> Z1);
  acc ^= t0 ^ t1;
  // Alternate: shift each word in opposite direction
  // acc ^= (t0 >> Z2) ^ (t1 << Z3);
  // Alternate: second stage using masking
  // acc ^= (t0 & MASK) | (t1 & ~MASK);
  sync();
  return (s[j]=acc);
}
```

6.2. Three Input RNG

One relatively cheap way to improve the correlation problem is to introduce a third input to the function. Because the first two samples have already introduced the shifting required for maximum period, we can choose not to shift this third input. The result is that both the most and least significant bits are a mixture of bits from two different streams, while the middle bits are a mixture from three streams.

This approach can be generalised to four or more input RNGs, simply by adding more and more unshifted stages to the generator. The cost of each stage is one memory lookup, one exclusive-or, and one constant register to hold the source index, so this solution is ideal for architectures where memory reads are cheap (e.g. NVidia architectures).

6.3. Slow Memory Rng

The multi-input solution to the correlation problem is only efficient when memory reads are cheap, which may not be the case in some architectures. When

Listing 5. Iterated RNG

```
unsigned IteratedRng(unsigned j, unsigned *s)
{

  CorrelatedRng(j,s);
  return CorrelatedRng(j,s);
}
```

logic operations are much cheaper than memory accesses, it may make more sense to read just two inputs from memory, but then to re-use each input in multiple stages. For example, after the initial left and right shift of the first and second inputs (required for maximum period), the two inputs can then be xor'd directly into the result without any shifting.

A more expensive alternative, would be to apply shifts in the second stage, moving the words in the opposite direction to the first stage. This would mean that bits from each input end up further from each other in the final value. Another alternate would be to use a bit-mask to interleave bits from the two words without shifting, although it is not clear why this would be a better option, as it has the same cost but results in less mixing.

6.4. Iterated RNG

Another way of reducing correlations between streams is to perform two update passes per generated random number: in the simplest case this could just mean executing CorrelatedRng twice instead of once before returning a random number to the application. Because $\gcd(2, 2^n - 1) = 1$ the resulting random stream will still have period $2^n - 1$, but each output word will be constructed from four of the previous output words, and the most and least significant bits will be combined from two previous words.

A more sophisticated alternative is to perform multiple passes with different sources and shifts, but to treat the combination of the two passes as a single transition: if the first pass applies matrix $\mathbf{A}_{(0)}$ and the second applies $\mathbf{A}_{(1)}$, then we just need to check whether $\mathbf{A} = \mathbf{A}_{(0)}\mathbf{A}_{(1)}$ has a primitive characteristic polynomial.

The iterated method only really makes sense if both memory reads and writes are very fast: a standard four input rng is able to provide a similar amount of mixing, without needing to perform the extra intermediate write. This approach also requires twice the amount of synchronisation, so in architectures where this is not free there will be a significant overhead.

Listing 6. BufferedRNG

```
unsigned BufferedRng(unsigned j, unsigned *s)
{
  // stored in register between invocations
  static unsigned buff;

  sync();
  unsigned t0=s[Q[0][j]], t1=s[Q[1][j]];
  unsigned acc=buff;
  acc ^= (t0<<Z0[j]) ^ (t1>>Z1);
  // Save a different value for the next iteration
  buff = s[Q[2][j]];
  // Alternate: Less memory reads, more operations
  // buff=(t0>>Z2) ^ (t1<<Z3);
  sync();
  return (s[j]=acc);
}
```

6.5. Buffered RNG

So far we have stayed within the framework outlined in section (TODO): all state is stored in shared memory, and with each thread performing multiple reads and one or more writes to transform this state. In this approach the state size (i.e. period) is directly limited to the number of threads co-operating in each RNG, as there is one word per thread. In general RNGs with larger states are better, both intuitively, and as measured by equidistribution (see section TODO). In principle we could simply increase the number of threads per RNG, but this has two disadvantages. The first is that if the number of threads is greated than the architecture's natural warp size, then all the memory synchronisation between threads no longer comes for free; for example, on NVidia GPUs the synchronisation must now be performed explicitly using `__syncthreads()`, effectively adding two extra instructions to the execution time.

The second disadvantage is that while increasing the period of the generator increases the quality of each individual stream, it does nothing to reduce the correlations between the streams. In section (TODO) it is shown that the maximum achievable mutual-equidistribution (a measure of stream independence) does not increase as the number of threads increases. Because correlations are the key potential weakness of this type of generator, and the quality of the individual streams is already very good, it makes little sense to attempt to increase RNG period without also trying to decrease stream correlation.

One solution to this problem is to note that state can also be held in thread-local registers, which increases the overall state size and period length of the RNG, without requiring more shared memory or more threads

per RNG. The simplest approach is the BufferedRng approach shown in Listing 6, which uses one register per thread to double the RNG state size. The basic structure is similar to the Three Input RNG, where we take the minimal Correlated RNG and introduce a third input to fix the most and least significant bits. However, instead of taking the third input from the current state, it actually comes from the previous state.

There is significant freedom in what exactly is saved in the register, with the simplest and cheapest option being to just save one of the inputs verbatim. At the expense of one exclusive-or, the combination of both inputs can be saved, or a completely independent input could be read and saved.

6.6. Additional Possibilities

Introducing state registers to the RNG opens up many possibilities, for improving quality and increasing period. For example, multiple levels of registers can be introduced per thread to create short FIFOs, greatly increasing the RNG period. Using three registers per thread would quadruple both the period of the generator, and quadruple the maximum achievable mutual-equidistribution.

Using registers also addresses one of the problems of creating very long period RNGs: in order to verify that an RNG has maximum period, we must know the exact factorisation of $2^n - 1$. Unfortunately, for convenient values of $n > 2048$ we do not have complete factorisations. For example, in an NVidia architecture (where $g = 32$ and $w = 32$) adding three state registers per thread would result in a state size of $n = 4096$, but the complete factorisation of $2^{4096} - 1$ is currently unknown [2]

One way of approaching this is to only use a part of the values stored in the register FIFOs, either by shifting or masking some of the bits out. For example, a bit mask could be used so that only subset of the bits from the previous state are stored in the extra registers; the mask would have to contain ones in the most and least significant bits to remove direct copying of bits, but some of the middle bits could be zeros. This allows the effective size of the state to be reduced to a value of n for which the factorisation *is* available, such as 2^{3900}.

7. Generators

8. Appendix

[2] Unknown at the time this paper was written: better factorisation algorithms and increasing computational power allow ever larger Mersenne numbers to be factorised.