Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

Pseudorandom Number Generation on the GPU

M. Sussman[1], W. Crutchfield[1], and M. Papakipos[†1]

[1]PeakStream, Inc., Redwood City, CA, USA

Abstract

Statistical algorithms such as Monte Carlo integration are good candidates to run on graphics processing units. The heart of these algorithms is random number generation, which generally has been done on the CPU. In this paper we present GPU implementations of three random number generators. We show how to overcome limitations of GPU hardware that affect the feasibility and efficiency of employing a GPU-based RNG. We also present a data flow model for managing and updating substream state for each of the parallel substreams of random numbers. We show that GPU random number generators will greatly benefit from having more outputs from each thread. We discuss other hardware modifications that will be beneficial to the implementation of GPU-RNG, and we present performance measurements of our implementations.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications G.3 [Probability and Statistics]: Randum Number Generation G.4 [Mathematical Software]: Parallel and vector implementations

1. Introduction

GPU performance is improving rapidly and already offers a significant performance advantage in floating point computations compared to the CPU [OLG*05]. Statistical algorithms such as Monte Carlo integration are particularly good candidates for using the computing power of the GPU, as these algorithms are often referred to as "naturally parallel" [MCS99]. An important part of Monte Carlo integration techniques, as well as numerical simulations of stochastic systems [SPM05], is the generation of pseudorandom numbers. For certain random number generators, the performance advantage of the GPU should make it significantly faster at generating random numbers than the CPU. Additionally, using a GPU-based RNG for statistical simulations on the GPU avoids the data transfer cost of using a CPU-based RNG. In this paper we present techniques for mapping pseudorandom number generation algorithms to the GPU, in order to use the GPU as a parallel compute engine for statistical simulations.

There are many pseudorandom number generation algorithms (also referred to as random number generators), which provide sequences of numbers that *approximate* the statistical properties of true random numbers. In this paper we are concerned with *uniform* random numbers u_n in the interval $[0,1)$:

$$\{u_0, u_1, u_2, \ldots\}$$

By far the most popular random number generators are *linear* generators, whose basic operations consist only of multiplication, addition, and modulus [Knu97]. For example, the linear congruential generator (LCG) is written as:

$$\begin{aligned} x_n &= (ax_{n-1} + c) \bmod m \\ u_n &= \frac{x_n}{m} \end{aligned}$$

where a is the multiplier, m is the modulus, and c is the increment. As with all pseudorandom number generators, the LCG has a period, or cycle length over which the sequence repeats. With appropriate choices of a, c, m and x_0 (referred to as the seed), the period is m [Knu97]. In contrast to the linear random number generators of various types are *nonlinear* random number generators, such as inversive congruential generators. These generators are less popular because they require more operations.

In computer graphics, *noise functions* such as the *Perlin noise* function are used to add randomness to other-

† Founder and Chief Technology Officer, PeakStream, Inc.

wise sharp computer generated images. They have controllable frequency bands and are designed such that animations do not change unexpectedly from frame to frame. Some noise functions have recently been implemented on the GPU [Ola05]. These appear to be related to a class of explicit nonlinear pseudorandom number generators. However, the requirements for noise functions and pseudorandom number sequences for Monte Carlo simulations are different, so we do not further discuss noise functions here.

Modern GPU hardware (Direct3D 9 pixel shader version ps_3_0 [Micb]) provides a programmable parallel processor with the following important attributes [LKM01] [Eng04]:

- floating point arithmetic, single precision only
- no hardware support for integer arithmetic
- up to 16 floating point outputs for each thread (4 render targets of type float4)

We will show how these hardware attributes play an important role in determining which random number generation algorithms may be efficiently implemented on a GPU.

We present implementations of three different random number generators for the GPU. The first is a nonlinear generator that may be implemented using ps_3_0. The second and third are linear generators that require changes to the GPU hardware and programming model to enable efficient implementations. We also propose a model for any generator, where parallel substream states are initialized, maintained and updated on the CPU, while the random numbers are generated on the GPU.

In the remainder of this paper we discuss well-known methods of parallelizing random number generators. We give examples of parallel random number generators that have been implemented on the CPU, and we discuss the limitations and difficulties of mapping such algorithms to the GPU. We then show how many of these limitations may be overcome in the implementations of our three random number generators. We present performance measurements (or performance estimates) for our GPU-based random number generators and compare their performance to reference implementations on the CPU. Finally, we discuss future work, and review the changes in GPU hardware that will be most beneficial for GPU-based random number generation.

1.1. Parallel Random Number Generators

It is well-known that correlations occur in linear random number generators [Knu97]. To construct a generator that minimizes correlations, tests such as the spectral test are used to find good choices of modulus and multiplier. If such searches for modulus and multiplier are not done carefully, it is easy to construct a method that has poor randomness properties according to one or more tests. Examples of software containing a "bad" generator can be found in the literature [Ent97]. The problem of correlations also affects any parallel linear random number generator.

Much work has already been devoted to devising random number generation algorithms for parallel computation [EUW98] [MCS99] [LSCK02]. The key problem solved in previous work has been to create parallel substreams of random numbers that are statistically independent, both within a substream and between substreams. Defining parallel substreams:

$$\begin{gathered}\{x_n, x_{n+1}, x_{n+2}, \ldots\} \\ \{y_n, y_{n+1}, y_{n+2}, \ldots\} \\ \vdots \\ \{w_n, w_{n+1}, w_{n+2}, \ldots\}\end{gathered}$$

Any of the substreams may be tested for intra-stream correlations, using one or more tests. Any sequence constructed by selecting elements of the substreams at the same n

$$\{x_n, y_n, \ldots, w_n\}$$

may be tested for inter-stream correlations. The cycle length of the substreams, over which the pattern of numbers repeats, must be sufficiently long to avoid repeating a cycle within one computation [SPM05]. Testing the statistical properties of random number generators can be done using well-known test suites such as Diehard [Mar95] and TestU01 [L'E05].

There are two primary methods of parallelizing random number generators, referred to as *cycle splitting* and *parameterization* [SPM05]. In *parameterization*, a random number stream has input parameters that have different values in each substream. The parameters may be seeds, which means that each substream is a different stream emitted from the same generating function, or the parameters may be used in the underlying generating function at each iteration n, for example parameterizing the multiplier and modulus of an LCG. In the latter case, the method of generating random numbers is different in each substream. An example of this kind of parameterization is the Wichmann-Hill generator [Mac89], a family of combined linear congruential generators (CLCG) that offers 273 possible substreams with different multipliers and moduli.

In *cycle splitting*, a single random number stream u_n of length N is broken into P separate substreams of length B. If *blocking* is being used, the i^{th} substream generates the sequence $\{u_{iB}, u_{iB+1}, \ldots, u_{iB+B-1}\}$. If *leapfrog* is being used, the i^{th} substream generates the sequence $\{u_i, u_{i+P}, u_{i+2P}, \ldots\}$. Figure 1 is a schematic diagram showing these two methods of cycle splitting.

Long range correlations that occur in linear random number generators are further exacerbated by cycle splitting. With blocking, long range correlations in a random number stream become inter-substream correlations, and with leapfrog, long range correlations become short-range intra-substream correlations [SPM05]. Searches are performed to

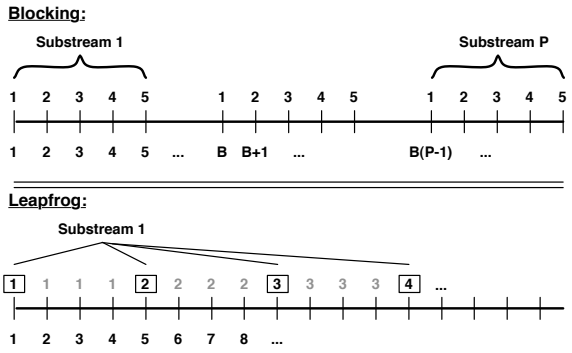

Figure 1: *Diagrams of blocking and leapfrog*

find good values of block size for a given linear generator in order to minimize these correlations [LSCK02].

2. Examples of Parallel RNGs

2.1. Combined Multiple Recursive Generator

The multiple recursive generator (MRG) is a linear generator with k stages defined by:

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k}) \bmod m \quad (1)$$

$$u_n = \frac{x_n}{m} \quad (2)$$

The output of the generator is a floating point number u_n which is in the half-open interval $[0,1)$, while the state of the generator is represented by k integers $x_{n-1},\ldots,x_{n-k}$. To create a generator with a longer period, the combined multiple recursive generator (CMRG) [L'E96] combines more than one MRG according to:

$$u_n = \left(\frac{x_{n1}}{m_1} + \cdots + \frac{x_{nj}}{m_j}\right) \bmod 1 \quad (3)$$

One example of a CMRG that has good properties is the MRG32k3a generator of L'Ecuyer [LAZ*99]. It has a period approximately 2^{191} and can be divided into many parallel substreams [LSCK02]. It is incorporated in the MKL and ACML math libraries from Intel and AMD, respectively [MKL] [ACM].

2.2. Combined Explicit Inverse Congruential Generator

An example of a non-linear random number generator is the Explicit Inverse Congruential Generator (EICG) [EUW98] of the form:

$$x_n = \overline{a(n+n_0)+c} \bmod m \quad (4)$$

$$u_n = \frac{x_n}{m} \quad (5)$$

where the horizontal bar indicates inversion modulo m with special handling of zero argument:

$$\begin{array}{ll} \overline{n}n = 1 \,(\bmod m)\,, & n \neq 0 \\ \overline{n} = 0, & n = 0 \end{array} \quad (6)$$

Equation 6 may be solved using the extended Euclid algorithm [Knu97] and by the binary extended gcd algorithm [MvOV96]. The EICG is *explicit* because the n^{th} random number u_n only depends on n and the "seed" n_0, and not on the previous state. The period of the generator is m and m is a prime number. Combining more than one EICG, each with different modulus, according to Equation 3, results in a CEICG. The period of CEICG is equal to the product of moduli [EUW98]:

$$\text{period} = \prod_{i=1}^{j} m_i \quad (7)$$

Note that computing modular inverse involves integer division and remainder, which means that intermediate values computed in the modular inverse algorithm never exceed the number of bits of m.

CEICG have good randomness properties [Lee95], for example they are completely free of lattice structure observed with linear generators [Knu97], and therefore require no special tuning of parameters to split into substreams for use in parallel computing [EUW98]. However, their use is limited by the fact that modular inversion is an expensive operation where cost scales $O(\log m)$, using the extended Euclid algorithm. As we discuss below, CEICG are of interest on the GPU, due to the limits of the hardware.

3. Mapping to the GPU : Limitations and Difficulties

3.1. Limitation: 16 outputs per thread

In a pixel shader program, each pixel position (x,y) may be thought of as being a separate thread of execution. Although the actual number of executing SIMD threads is much smaller than the maximum number of pixels, the order in which pixel positions will be visited by the shader program is unknown. Framebuffers have maximum size of 4096 by 4096, meaning that a GPU-RNG may process up to 2^{24} independent requests to generate random numbers. If the GPU does not allow more than 16 outputs per thread, the choice of the method of parallelizing the generator is restricted. In a pixel shader program, 16 random numbers may be generated for each pixel position. In this case, there would be a need for 2^{20} parallel substreams to fill the largest possible buffer with random numbers ($16 \cdot 2^{20} = 2^{24}$). Using the method of substream parameterization, either a large amount of seed data would be needed, or the family of random number generators would have to offer a large choice of substream parameters (far exceeding, for example, the available options for the Wichmann-Hill type). For most generators a large amount of memory would be required to store the

states of all of the 2^{20} substreams. For example, MRG32k3a would require 48 MB to store substream state.

Without more than 16 outputs per thread, it would be prohibitive to use the CPU to update the state of each substream because of the number of substreams required, both in CPU cost of the update function and in memory transfer cost. Alternatively, updating the state of each substream on the GPU would require most of the output render targets be used for outputting *substream state* rather than generated random numbers. For example, the substream state of MRG32k3a is represented by six 32-bit integers. As discussed below, this requires 12 single precision floating point numbers to represent on the GPU. Each pixel position would therefore be required to write 12 floating point values to update the state of its substream, leaving only four slots to write outputs from the RNG. That increases the required number of substreams by another factor of four.

It is clear that without more than 16 outputs for each thread, it will be difficult to create an efficient implementation of many types of parallel RNGs. One exception to this concern is CEICG, which does not require storage of substream states, since it is an explicit method. This allows computation of independent substreams dependent on pixel position and not previous state.

3.2. Difficulty: 32-bit (and higher) integer arithmetic

Most modern parallel RNGs require 32-bit integer constants, with greater than 32-bit arithmetic. For example, MRG32k3a uses formulas of the form:

$$z_n = (ax - by) \bmod m \tag{8}$$

In this case the modulus m is 32 bits, as are the previous values of the state x and y. The coefficients a and b are up to 21 bits. The computation may be carried out using either double precision floating point arithmetic, which can represent integers up to 53 bits, or it may be carried out using 64-bit integer arithmetic. Otherwise, simulated higher-range integers are required to perform the steps of the computation. In ps_3_0, arithmetic operations are limited to single precision floating point arithmetic, which can exactly represent integers up to 24 bits.

In general it is a complex task to find the parameters for a good linear random number generator that will fit into a certain number of bits. For this reason, implementing linear generators on the GPU will most often require simulating higher range integers, because many of the published methods that are known to work well use greater than 24-bit arithmetic. An exception to this rule is the Wichmann-Hill generator included in MKL and ACML.

Another exception to this concern is CEICG, because it is nonlinear. A nonlinear generator may be readily constructed in which all of the operations require only 24-bit floating point arithmetic, as will be shown in a later section.

3.3. Difficulty: inexact integer division

Although the GPU supports IEEE-754 *format* single precision floating point numbers, certain mathematical operations do not necessarily conform to the results expected from the CPU. Among those operations are integer division and modulus, which are the basic operations needed by many random number generators. On the GPU there may be rounding errors in integer division results in the least significant bit, which causes errors in fmod. A workaround for fmod is Algorithm 1. The workaround for integer division $\lfloor a/b \rfloor$ is similar to fmod, instead returning div at the end of Algorithm 1, or if both the integer division and fmod result are needed, they may be returned simultaneously from Algorithm 1.

Algorithm 1 Calculate fmod(a,b) for non-negative integers

```
div ← floor(a/b)
rem ← a − b·div
if rem < 0 then
    div ← div − 1
    rem ← rem + b
else if rem >= b then
    div ← div + 1
    rem ← rem − b
end if
return rem
```

4. Implementation of CEICG on the GPU

Because of the difficulties discussed in the previous sections, it is easier to implement CEICG on the GPU than it is to implement a high quality *linear* random generator. Specifically, single precision floating point arithmetic is sufficient, and there is no need for more than 16 outputs per thread. We may have a large number of substreams because there is no need to save substream state.

Since CEICG is explicit and we are limited in the number of outputs, we would like to have as many substreams as there are pixel positions. Thus we require there to be 2^{24} substreams. To avoid exceeding 24-bit arithmetic, we choose prime numbers slightly less than 2^{24} as the prime modulus m for each EICG. Since the period of the generator is m, we will require a combination of three EICG to give a period $\approx 2^{72}$ (recall Equation 7, the period is the product of 3 moduli $\approx 2^{24}$). The sequence can be divided into 2^{24} substreams of length B not to exceed $\approx 2^{48}$. If we were to only combine two EICG, each substream would have less than 17 million numbers before repeating, which is too short a period for practical use. We divide the long stream into substreams of length B using each pixel position (x,y), so that each EICG is:

$$x_n = \overline{a\,[n + n_0 + (x + 4096y)\,B]} + c \bmod m \tag{9}$$

The values of the multipliers and moduli for the three combined EICG are given in Table 1.

Table 1: *Parameters used in CEICG*

k	m	a	c	$B \bmod m$
1	16777213	7	0	24
2	16777199	11	0	1753
3	16777183	13	0	3969

$B = 140739392569023 \approx 2^{47}$

The only tuning that has been done is to select a value of B such that $B \bmod m$ is less than $\sqrt{m}$. We can then apply Algorithm 2, a factorization for modular multiplication that permits computing $ax \bmod m$ without computing any numbers that exceed m in absolute value [Knu97]. Since m is less than 2^{24} that means we do not require more than single precision floating point arithmetic.

Algorithm 2 Calculate $ax \bmod m$ without exceeding m

Require: $a^2 < m$
Require: $x < m$
Require: $\lfloor m/a \rfloor$ and $m \bmod a$ {Precompute}
  $q \leftarrow \lfloor m/a \rfloor$
  $r \leftarrow a(x \bmod q) - (m \bmod a)\lfloor x/q \rfloor$ {use Algorithm 1}
  if $r < 0$ **then**
    $r \leftarrow r + m$
  end if
  return r

The pseudocode for computing one of the EICG is shown in Algorithm 3. The three EICG are combined according to Equation 3 to form the final random number output.

Algorithm 3 Compute EICG at pixel position (x,y)

Require: $x < 4096$
Require: $y < 4096$
Require: $n + n_0 < m$
  $s \leftarrow (x + 4096y) \bmod m$
  $b \leftarrow B \bmod m$
  $s \leftarrow bs \bmod m$ {use Algorithm 2}
  $s \leftarrow (s + n) \bmod m$
  $s \leftarrow as \bmod m$ {use Algorithm 2}
  $s \leftarrow (s + c) \bmod m$
  $r \leftarrow \bar{s} \bmod m$
  return r

The state of the generator is represented by only three floating point numbers n_1, n_2, n_3, one for each of the three EICG, where

$$0 \le n_k < m_k$$

After each call, the state is advanced according to

$$n_k = (n_k + 1) \bmod m_k.$$

The float4 version of this generator uses the leapfrog method, taking four consecutive values of n_k for the r, g, b, α components of the float4 buffer. In that case the state is advanced according to

$$n_k = (n_k + 4) \bmod m_k.$$

The initial value of n_k is set to a user-specified seed value n_{0k} with no loss of generality.

4.1. Test Results

This generator passes the Diehard test suite provided we assume that the resolution of the generator is 23 bits, and generate 32-bit inputs to Diehard from two consecutive outputs of the generator.

5. Implementation of MRG32k3a on the GPU

We assume that the hardware limitation of 16 outputs per thread will be eliminated in future GPU hardware. This assumption is a requirement for this section of the paper. One method of providing more outputs per thread is a hardware scatter operation, where a thread may write to multiple locations in an output target. In ps_3_0 hardware, pixel position (x,y) always writes to position (x,y) in one to four render targets. Scatter operations are desirable for algorithms that require random access to output array(s). Vertex shaders are already capable of scatter operations [OLG*05]. In the case of random number generators, the primary benefit of a scatter operation would be to allow more outputs per thread. Therefore, the requirement for this section of the paper would be satisfied by any hardware modification that permits a much larger number of outputs per thread.

Assuming that a much larger number of outputs are possible from a shader program, it is feasible to implement a linear generator such as MRG32k3a. In this case pixel position designates a particular random number substream, whose state may be read from input textures. As an example, one may partition the 2^{24} values in a large output target into 2^{12} substreams that each write 2^{12} output values. The amount of memory required to store the state of 2^{12} substreams is 192 KB, because each substream requires 12 floating point values (48 bytes). A detailed study is needed to determine the precise optimum number of substream states that should be maintained. It should be noted that such a study may also need to test for correlations between different positions in an output target, which is a function of the chosen number of substreams and the size of the arrays being filled.

The basic update step of MRG32k3a is [L'E96]

$$x_{1,n} = \left(ax_{1,n-2} - bx_{1,n-3}\right) \bmod m_1 \quad (10)$$

$$x_{2,n} = \left(cx_{2,n-1} - dx_{2,n-3}\right) \bmod m_2 \quad (11)$$

$$u_n = \frac{x_{1,n}}{m_1} + \frac{x_{2,n}}{m_2} \quad (12)$$

where the values of the constants are given in Table 2.

Since we require greater than single precision floating

Table 2: *MRG32k3a parameters [L'E96]*

constant	value
m_1	4294967087
m_2	4294944443
a	1403580
b	810728
c	527612
d	1370589

point arithmetic, numbers are represented by partitioning them into 2 or 3 values, depending on whether they are 32-bit generator state or 53-bit intermediate quantities, respectively. An extended range integer is represented by

$$x = x_0 + x_1 2^{24} + x_2 2^{48} \tag{13}$$

if 53 bits are required, and $x_2 = 0$ if only 32 bits are required. The algorithm to add or subtract two such extended range integers is an extension of the paper and pencil method with carry and borrow. The algorithm to multiply two such numbers relies on further factoring the numbers according to

$$x = x_l + 4096 x_h$$

and again keeping track of the carry digits as in the paper and pencil method. An algorithm to compute modulus is called Barrett modular reduction [MvOV96], which relies on storing the numbers $\lfloor 2^{64}/m_k \rfloor$ and using multiplication and division by powers of 2. Finally the division by m_k necessary in Equation 12 is done by storing $1/m_k$ in the form

$$1/m_k = d_0 2^{-24} + d_1 2^{-36} + d_2 2^{-48}$$

that is sufficient to produce the required accuracy in the final floating point result.

5.1. Initializing and Maintaining Substream State on the CPU

In our proposed example, each thread designated by a pixel position (x,y) will compute 4096 output values that will be written to an output render target. At the end of that loop, either the GPU must update the substream state in preparation for the next call to the RNG, or this update must be done on the CPU. For the update to be done on the GPU, we must be able to follow the computation of the output values by writing 12 floating point values representing the updated state of the substream for pixel position (x,y).

Alternatively, the substream states may be updated on the CPU either after a call to the shader program that generates the random numbers, or while such a shader program is executing. This may also be advantageous in the instance that the generator requires periodic re-seeding from the CPU, because the substream state will be resident on the CPU. This update step when a substream is advanced by more than one iteration is called "jumping ahead" [LSCK02]. For our example implementation of MRG32k3a, the new state of each substream may be computed by jumping ahead by 4096. This involves multiplying the state vector $(x_{k,n-1}, x_{k,n-2}, x_{k,n-3})$ by a 3-by-3 matrix $A_k^{\nu} \bmod m$, which can be computed by a divide-and-conquer algorithm from the jump-ahead one step matrix A_k, with cost $O(\log \nu)$. The same procedure is used for each of 2 state vectors, i.e. $k = 1,2$. To do the jump-ahead computation on the GPU would be cumbersome, because intermediate numbers go up to 64-bits, but in theory it could be done using the same extended range integers that are used in the update step. This model of the GPU-RNG data flow is shown in Figure 2. A more detailed study is necessary to determine the optimum location to perform the substream state updates, and the optimum may be specific to the particular RNG and to the way it is being used.

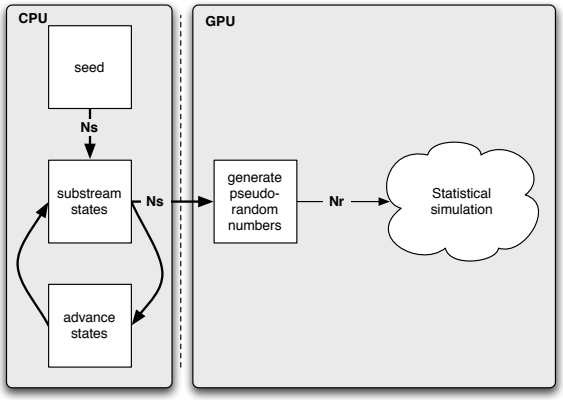

Figure 2: *Data flow for substream state and RNG. Note that $N_r \gg N_s$, where N_r is the amount of data generated by the RNG and N_s is the amount of substream state data.*

6. Implementation of Wichmann-Hill on the GPU

Assuming that a much larger number of outputs are possible from a shader program, the Wichmann-Hill generator may also be easily implemented on the GPU. This generator is one of the few linear generators with the *significant* advantage of requiring only single precision floating point arithmetic. The update step for Wichmann-Hill is [Mac89]

$$\begin{aligned} x_{1,n} &= (c_1 x_{1,n-1}) \bmod m_1 \\ x_{2,n} &= (c_2 x_{2,n-1}) \bmod m_2 \\ x_{3,n} &= (c_3 x_{3,n-1}) \bmod m_3 \\ x_{4,n} &= (c_4 x_{4,n-1}) \bmod m_4 \\ u_n &= \left(\frac{x_{1,n}}{m_1} + \frac{x_{2,n}}{m_2} + \frac{x_{3,n}}{m_3} + \frac{x_{4,n}}{m_4} \right) \bmod 1 \end{aligned}$$

In this case there are up 273 sets of constants, so we can have (for simplicity) 256 substreams that would each output

65536 values to fill up a render target of size (4096,4096). The constants vary in the range [Mac89]

$$16718909 \leq m_k \leq 16776971$$
$$112 \leq c_k \leq 127$$

Because the constants m_k are 24 bits, and c_k are all less than $\sqrt{m_k}$, we may use Algorithm 2 to compute modular multiplication, and that is what permits the computations to be done with single precision floating point arithmetic. The Wichmann-Hill generator may also be "skipped-ahead", meaning that substream state may be updated on the CPU while the GPU is computing random numbers, as shown in the data flow model in Figure 2. Since Wichmann-Hill is a linear generator that can be computed with single precision floating point arithmetic, it is expected to be among the fastest possible random number generators on the GPU.

7. Summary and performance

Table 3 has a summary of the three random number generators that we have implemented on the GPU, showing the number of substreams, the length of the sequence generated by each thread, and the total amount of substream state data that must be managed.

Table 3: *Summary of GPU-RNG substream properties*

name	number of substreams	sequence length	state data memory use
CEICG	2^{24}	$\approx 2^{47}$	12 B
MRG32k3a	$4096^{\dagger}$	$\approx 2^{115}$	192 KB
Wichmann-Hill	256	$\approx 2^{62}$	4 KB

†for example

We measured the performance of CEICG on an ATI X1900 series GPU with 500 MHz clock and 594 MHz memory clock. Peak speed was 44 million random numbers generated per second. In addition, we measured the speed of the update functions for MRG32k3a and Wichmann-Hill generators, while writing out only 16 values per pixel position. For MRG32k3a, we were unable to fit the entire update function in the limited instruction count for the pixel shader program, so we measured the performance of updating only one of the two component random numbers. That brings up another hardware limitation, which is limited number of programmable instructions (the complete generator would require two passes). From these measurements we estimate the peak speeds of MRG32k3a and Wichmann-Hill generators to be 111 million and 823 million random numbers per second, respectively. As anticipated, CEICG is the slowest generator because it is nonlinear. The huge difference in speed between MRG32k3a and Wichmann-Hill is due to the fact that for MRG32k3a we're required to perform extended range integer arithmetic beyond the range of single precision floating point, and this requires many instructions. Future generations of GPU hardware that offer 32-bit IEEE-compliant integer arithmetic, as described in the Direct3D 10 Technology Preview [Mica], are expected to yield significant improvements in the performance of MRG32k3a and other linear generators on the GPU.

Table 4: *Summary of RNG performance measurements*

name	machine	speed (million/sec)
CEICG	ATI X1900	44
CEICG	Xeon 3.6 GHz	$0.3^{\dagger}$
MRG32k3a	ATI X1900	111 (est.)
MRG32k3a	Xeon 3.6GHz	110
Wichmann-Hill	ATI X1900	823 (est.)
Wichmann-Hill	Xeon 3.6 GHz	79

†not using SSE

For comparison purposes, we also measured MKL implementations of MRG32k3a and Wichmann-Hill on an Intel Xeon using one 3.6 GHz CPU. We measured 110 million random numbers per second for MRG32k3a and 79 million random numbers per second for Wichmann-Hill. We also measured the performance of CEICG on an Intel Xeon using a reference C language implementation compiled with gcc, achieving a peak speed of 0.3 million random numbers per second. The measurements are summarized in Table 4. In the cases of CEICG and Wichmann-Hill, a significant advantage in performance is achieved using the GPU. It is expected that a significant advantage will be observed for CEICG even if compared to a highly tuned reference implementation on the CPU using SSE instructions.

8. Conclusions and summary of desirable GPU design changes

This paper presented three example implementations of random number generators on the GPU, showing the limitations of present GPU hardware that affect such algorithms, and showing how to overcome these limitations. We also presented a data flow model where the CPU is used for maintaining and updating the substream state for each of the parallel substreams on the GPU. Of the generators we examined, the nonlinear generator CEICG may be run on present hardware, and offers excellent randomness properties as well as efficient utilization of the GPU. Linear generators such as MRG32k3a and Wichmann-Hill will significantly benefit from being able to write many more output values from each thread, for example by using a scatter operation. GPU-based random number generators will also greatly benefit from hardware improvements such as:

- unlimited number of outputs per thread
- double precision arithmetic and/or integer arithmetic
- IEEE-754 exactly compliant fmod function

- more instructions in each shader program
- scatter operations

Finally, our performance measurements showed the promise of GPU implementations of random number generation, and by extension, of GPU-based statistical simulations.

References

[ACM] ACML: *AMD Core Math Library*. http://developer.amd.com.

[Eng04] ENGEL W.: *Programming Vertex and Pixel Shaders*. Charles River Media, Inc., 2004.

[Ent97] ENTACHER K.: *A collection of selected pseudorandom number generators with linear structures*. Tech. Rep. TR 97-1, ACPC, Austrian Center for Parallel Computation, 1997. http://random.mat.sbg.ac.at/results/karl/server.

[EUW98] ENTACHER K., UHL A., WEGENKITTL S.: Linear and inversive pseudorandom numbers for parallel and distributed simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation* (1998), IEEE Computer Society, pp. 90–97.

[Knu97] KNUTH D. E.: *The Art of Computer Programming*, 3 ed., vol. 2: Seminumerical Algorithms. Addison-Wesley, 1997.

[LAZ*99] L'ECUYER P., ARIELY D., ZAUBERMAN G., FISCHER W., CARMON Z.: Good parameters and implementations for combined multiple recursive random number generators. *Operations Research 47*, 1 (1999), 159–164.

[L'E96] L'ECUYER P.: Combined multiple recursive generators. *Operations Research 44*, 5 (1996), 816–822.

[L'E05] L'ECUYER P.: *TestU01: Empirical Testing of Random Number Generators*, 2005. http://www.iro.umontreal.ca/~simardr/testu01/tu01.html.

[Lee95] LEEB H.: *Random Numbers for Computer Simulation*. Master's thesis, University of Salzburg, 1995.

[LKM01] LINDHOLM E., KILGARD M. J., MORETON H.: A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH* (2001), pp. 149–158.

[LSCK02] L'ECUYER P., SIMARD R., CHEN E. J., KELTON W. D.: An object-oriented random-number package with many long streams and substreams. *Operations Research 50*, 6 (2002), 1073–1075.

[Mac89] MACLAREN N. M.: The generation of multiple independent sequences of pseudorandom numbers. *Applied Statistics 38* (1989), 351–359.

[Mar95] MARSAGLIA G.: *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*, 1995. http://www.stat.fsu.edu/pub/diehard.

[MCS99] MASCAGNI M., CEPERLEY D., SRINIVASAN A.: SPRNG: A scalable library for pseudorandom number generation. In *Recent Advances in Numerical Methods and Applications II, Proceeding of NMA '98* (1999), Iliev O., Kaschiev M., Sendov B., Vassilevski P. S., (Eds.).

[Mica] MICROSOFT: *Direct3D 10 Technology Preview*. http://msdn.microsoft.com/directx/sdk.

[Micb] MICROSOFT: *Direct3D 9*. http://msdn.microsoft.com/directx/sdk.

[MKL] MKL: *Intel Math Kernel Library*. http://www.intel.com.

[MM65] MACLAREN M., MARSAGLIA G.: Uniform random number generators. *JACM 12*, 1 (1965), 83–89.

[MN98] MATSUMOTO M., NISHIMURA T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulations 8*, 1 (1998), 3–30.

[MvOV96] MENEZES A. J., VAN OORSCHOT P. C., VANSTONE S. A.: *Handbook of Applied Cryptography*. CRC Press, 1996.

[Ola05] OLANO M.: Modified noise for evaluation on graphics hardware. In *Graphics Hardware* (2005), Meissner M., Schneider B.-O., (Eds.).

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (2005), pp. 21–51.

[SPM05] SCHOO M., PAWLIKOWSKI K., MCNICKLE D. C.: *A Survey and Empirical Comparison of Modern Pseudo-Random Number Generators for Distributed Stochastic Simulations*. Tech. Rep. TR-CSSE 03/05, University of Canterbury, New Zealand, 2005.