An Efficient Implementation of Bailey and Borwein's Algorithm for Parallel Random Number Generation on Graphics Processing Units

Gleb Beliakov*, Michael Johnstone**, Doug Creighton**, Tim Wilkin*
*School of Information Technology,
Deakin University,
221 Burwood Hwy, Burwood 3125, Australia
gleb@deakin.edu.au, tim.wilkin@deakin.edu.au

**Centre for Intelligent Systems Research,
Deakin University,
Pigdons Rd, Geelong 3217, Australia

Abstract

Pseudorandom number generators are required for many computational tasks, such as stochastic modelling and simulation. This paper investigates the serial and parallel implementation of a Linear Congruential Generator for Graphics Processing Units (GPU) based on the binary representation of the normal number $\alpha_{2,3}$. We adapted two methods of modular reduction which allowed us to perform most operations in 64-bit integer arithmetic, improving on the original implementation based on 106-bit double-double operations, which resulted in four-fold increase in efficiency. We found that our implementation is faster than existing methods in literature, and our generation rate is close to the limiting rate imposed by the efficiency of writing to a GPU's global memory.

Keywords GPU, random number generation, normal numbers.

1 Introduction

Pseudorandom number generators are required for many computational tasks, such as simulating stochastic models, numerical integration and cryptography [14]. We focus on generation of random variates for numerical simulations, and hence do not consider generation of cryptographically strong random numbers, which are treated elsewhere [8, 18]. For numerical simulations it is sufficient that the sequence of generated variates imitates a random

sequence, even if the future and past elements can be predicted using the current element of the sequence. Typically a generator that produces uniform pseudorandom variates on $(0, 1)$ (or random integers from $\{0, 1, \ldots, m\}$) is the base generator used to generate random variates from more complicated distributions [2, 7, 11].

Linear Congruential Generators (LCG) and Mersenne Twisters (MT) are two of the most important families of random generators. MTs offer longer periods than LCGs and do not suffer from various correlations between the elements of the sequence as LCGs, especially when an LCG is poorly designed [17]. However MTs require a bigger state space and their implementation is more complex than that of LCGs. Various statistical tests, such as SmallCrush and BigCrush from the TestU01 library [16] are used to evaluate the quality of the generated sequences.

General purpose Graphics Processing Units (GPU) have recently become a powerful alternative to traditional high performance computing. Certain calculations can be offloaded to the graphics card (referred to as GPU *device*), on which thousands of threads are executed concurrently. GPUs offer hundreds of processor cores ($p = 448$ in NVIDIA's Tesla C2050) at very modest prices, and are marketed as "desktop supercomputers". GPUs are well suited for many numerical simulations, where algorithms offer a high degree of parallelism. Hence a significant interest to parallel pseudorandom number generation, suitable for GPUs [12, 23, 24]. Important issues here are the speed of generation, the quality of the sequence generated, and the ability to produce the same sequence using different numbers of threads, number of hosts and GPU characteristics.

Recently Bailey and Borwein [4] proposed a pseudorandom generator based on binary representation of normal numbers. For an integer $b \geq 2$ we say that a real number K is b-normal if, for all $m > 0$, every m-long string of digits in the base-b expansion of K appears, in the limit, with frequency b^{-m}. That is, with exactly the frequency one would expect if the digits appeared completely "at random". The authors presented a thorough theoretical overview of the topic, and designed a special class of constants $\alpha_{b,c}$, where $b, c > 0$ and coprime. They proved that such constants are b-normal, and therefore their base-b expansion can serve to generate pseudorandom sequences.

A useful feature of the numbers $\alpha_{b,c}$ is that their $n-$th base$-b$ digit can be calculated directly, without needing to compute any of the first $n-1$ digits. It implies that any element of the generated sequence can be computed without computing the preceding elements, and therefore offers the desirable skip ahead feature of a random number generator. It is especially valuable for parallel random number generators, as consecutive subsequences of random variates can be generated concurrently on different processors or even different hosts, and the entire sequence is identical to the one generated in a single thread. This is a notable advantage of LCGs, shared by Bailey-Borwein generator, which is much harder to achieve in other schemes.

Bailey and Borwein [4] present a pseudorandom number generator based on the binary representation of $\alpha_{2,3}$, and outline its implementation as a version of LCG, with the period $P = 2 \cdot 3^{32} \approx 3.7 \cdot 10^{15}$. The seed of this generator is the index of the starting element, which is calculated directly through a simple formula. They suggest its use in parallel computations,

by splitting the sequence of variates into m subsequences (m is the number of threads), each subsequence i starting from the $1 + \frac{(i-1)n}{m}$-th element, where n is the desired number of random variates.

In this work we follow the path suggested by Bailey and Borwein and investigate a parallel implementation of their pseudorandom number generator suitable for GPUs. Firstly we confirm the quality of the random sequence using the comprehensive suite "BigCrush" from the TestU01 library [16]. Secondly we port the authors' implementation [3, 4] to GPU, and benchmark its speed. Initial investigations found that speedwise the proposed generator was comparable to but less efficient than the newest GPU-based MT generator `mtgp` [24]. Therefore we investigate improvements to Bailey's implementation based on either 106 bit double-double, or 128-bit integer arithmetic. We found several alternatives that use modular arithmetic, and improved the generation rate four-fold. Lastly we investigate the impact of GPU architecture on the generation rate, and determine an optimal configuration concerning the number of threads and thread blocks, and therefore work assigned to each thread. Our implementation delivers generation rate of 11 Gnumbers/sec on GPU, which is 25% faster than state of the art `mtgp`, and approaches the maximal rate of 12.4 Gnumbers/sec imposed by the speed of writing to global memory.

The paper is structured as follows: Section 2 reviews the LCG proposed by Bailey and Borwein. Section 3 describes improvements that can be made to the original implementation using modular arithmetic. Numerical experiments are described and results presented in Section 4. Finally we draw conclusions in Section 5.

2 Random number generator based on normal numbers

Bailey and Borwein [4] investigated the following class of constants, which was introduced previously in [5],

$$\alpha_{b,c}(r) = \sum_{k=1}^{\infty} \frac{1}{c^k b^{c^k + r_k}}$$

where integers $b, c > 1$ and co-prime and $r \in [0, 1]$, and where r_k denotes the k-th binary digit of r.

Bailey and Crandall [5] proved the following:

Theorem 1 *Every number $\alpha_{b,c}(r)$ is b-normal.*

Bailey and Borwein focus on the number

$$\alpha = \alpha_{2,3}(0) = \sum_{k=1}^{\infty} \frac{1}{3^k 2^{3^k}}$$

This number is 2-normal, but is not 6-normal, as Bailey and Borwein show. Further they show that all constants $\alpha_{b,c}$ are not $bc-$normal for co-prime integers $b, c > 1$.

The normality of α suggests that its binary expansion can be used to generate a sequence of pseudorandom numbers. Let $x_n = \{2^n\alpha\}$ be the binary expression of α starting from position $n+1$, where $\{\cdot\}$ denotes the fractional part of the argument. Bailey and Borwein show that when n is not the power of three

$$x_n = \frac{(2^{n-3^m}\lfloor 3^m/2 \rfloor) \mod 3^m}{3^m} + \varepsilon, \tag{1}$$

and the tail of the series $\varepsilon < 10^{-30}$ if n is not within 100 of any power of three.

Then the authors of [4] construct the following algorithm to calculate pseudorandom 64-bit real numbers in $(0,1)$, which would contain in their mantissas 53-bit segments of the binary expansion of α

Algorithm BB

1. Select starting index a in the range $3^{33} + 100$ to 2^{53}, referred to as the seed of the generator.
2. Calculate
$$z_0 = 2^{a-3^{33}} \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33} \tag{2}$$
3. Generate iterates
$$z_k = 2^{53} z_{k-1} \mod 3^{33} \tag{3}$$
and return $z_k 3^{-33}$, which are 64 floating point random variates in $(0,1)$.

Bailey and Borwein note that several operations need to be done with accuracy 106 mantissa bits, i.e., in 106-bit floating point arithmetic. They described and implemented their algorithm in double-double arithmetic [3,4]. Note that double-double arithmetic allows exactly the representations of integers up to 2^{106}.

The algorithm BB is a version of the LCG, and therefore it's period can be established from the theory of LCGs and results in $P = 2 \cdot 3^{32} \approx 3.7 \cdot 10^{15}$. The algorithm can be checked by calculating z_k-th iterate recursively, or directly by using formula (2). The binary digits of α within a range of indices spanned by successive powers of three are given by an LCG with a modulus that is a large power of three. Of course, by using a modulus, as large as 3^{40} one can get larger periods, but this will involve a higher number of bits in integer arithmetic, i.e. 128-bit integer arithmetic.

This approach has an advantage over LCGs that use a power of two value as the modulus, where arrays of pseudorandom data, of size matching a power of two, are accessed by row and column and therefore have the potential for a reduced period [4]. A modulus as that proposed by Bailey and Borwein removes this issue.

The direct formula (2) produces the notable skip ahead property: the ability to calculate the k-th iterate without iterating through the previous steps. This property is very valuable for parallel random number generators, as every parallel thread can calculate its starting iterate directly from (2), and it is harder to achieve in the MT scheme.

3 Parallelization and improvements to the algorithm

Using Bailey and Borwein's (BB) implementation [3] as the starting point, we translated it from FORTRAN to C, and implemented as a parallel random number generator on GPU platform using CUDA [19]. Our initial tests confirmed that the pseudorandom sequence produced was identical to the one produced by using the original implementation in a single thread. Our generation rate was 0.6 Gnumbers per second on Tesla C2050 card.

However, when compared to the state of the art MT generator developed by Saito and Matsumoto in 2010 [24], which has the rate of over 9 Gnumbers/sec for doubles in $(0, 1)$ on the same card, the BB's implementation was not competitive. We note that the MT generator `mtgp` we benchmarked against has the periods from $2^{11213} - 1$ to $2^{44497} - 1$, and is free from the usual problems of LCG such as small periods and some statistical flaws.

An alternative implementation of the BB algorithm is to use 128 bit arithmetic. NVIDIA have released such a library for CUDA [21], available to registered developers. This library implements in software 128 bit arithmetic, as GPU hardware support for 128 bit arithmetic is not currently available. The generation rate was similar to that of the BB implementation, still uncompetitive with `mtgp`.

Nevertheless we were still interested in the BB's LCG for two reasons. First, LCG's implementation is much simpler than that of MTs, and in particular the well optimized implementation from [13]. Simpler implementations are more portable and less tied to hardware characteristics than complex implementations such as `mtgp`. Second, BB's LCG offers an acceptable period, suitable for many simulation applications, which can also be easily extended using a larger modulus and therefore making the range between consecutive powers of three larger. Third, MT's state is larger than that of LCG, and Saito and Matsumoto's implementation involves cooperative generation of the sequence by several threads keeping the state in the shared memory. This could be inconvenient when using this generator in complex simulations on GPU, as pointed out by [10], which have their own demands on shared memory. In contrast, BB's LCG offers fully independent generation of the sequence by different threads and only a 64-bit state. Therefore we investigated possible improvements to the LCG.

Our focus was the generation step Eq. (3). Eq. (2) needs to be applied only once at the start of the process, and contributes to the total time only marginally, as long as the majority of the sequence is generated by Eq. (3).

3.1 Modular arithmetic

A modular reduction is simply the computation of the remainder of an integer division. However, division is a much more expensive operation than multiplication. There are a number of special methods for performing modular reduction using multiplication, summation and subtraction, and single precision division.

3.1.1 L'Ecuyer method

The first approach to modular reduction in LCG was adapted from P. L'Ecuyer [13]. This approach follows the earlier work [9], and is based on modular multiplication $az \mod m$, where $m = aq + r$, $r < a$ and $a^2 < m$. One has

$$az \mod m = (a(z \mod q) - \left\lfloor \frac{z}{q} \right\rfloor r) \mod m$$

The algorithm is stated as follows.

Algorithm L'Ecuyer
Input: z, a, m, $q = \lfloor \frac{m}{a} \rfloor$, $r = m \mod a$
Output: $s = (a \cdot z) \mod m$

1. $t = z \; div \; q$
2. $s = a(s - t \cdot q) - t \cdot r$
3. if$(s < 0)$ then $s = s + m$
4. return s

The quantities $q = \lfloor \frac{m}{a} \rfloor$ and $r = m \mod a$ are pre-computed. Every intermediate step remains strictly between $-m$ and m as shown in [13], and hence only one `if` statement is needed to perform the modulus operation.

However we cannot use L'Ecuyer's algorithm directly for calculating $2^{53} z_{k-1} \mod m$ because $a = 2^{53} > 3^{33} = m$, and the algorithm requires $a^2 < m$. Instead we apply this reduction algorithm twice using $a = 2^{25}$, and expanding

$$2^{53} z \mod m = 2^{25} \cdot 2(2^{25} \cdot 4z \mod m) \mod m.$$

Hence we call L'Ecuyer's algorithm in the following sequence of calls
$z = LEcuyer(4z_{k-1}, a, m, q, r)$
$z_k = LEcuyer(2z, a, m, q, r)$

Here all intermediate operations are less than 64-bits. Let us show that s stays in $[-m, m]$. An essential condition guaranteeing that the intermediate steps are between $-m$ and m, and hence correctness of $\mod m$ in step 3, is $z < m$ [13]. But in our case $z = 4z_{k-1} < 4m$, so z can be larger than m.

We modify the argument from [13] as follows.

$$\left\lfloor \frac{z}{q} \right\rfloor r < 4 \left\lfloor \frac{aq + r}{q} \right\rfloor r \leq 4ar \leq 4a^2 = 4 \cdot 2^{50} = 2^{52} < 3^{33} = m.$$

Therefore s at step 2 of L'Ecuyer's algorithm is between $-m$ and m for our specific values of m and a, and hence the algorithm works correctly when $z \leq 4m$.

We also note that computations are faster if instead of integer division in step 1 we multiply z by the pre-computed reciprocal of q (a 64-bit floating point number), which provides sufficient (53-bits) accuracy for our purposes, because its result is between $-m$ and m. The actual C source code is presented in Figure 1.

```
#define lcn(s, m, q, r, qinv)\
    T1 = (s)*qinv;\
    s = ((s-T1*q)<<25)-T1*r;\
    if (s < 0) s+=m;

#define lcn4(s, m, q, r, qinv)\
    T1 = (s)*qinv;\
    s =  (((s)-T1*q)<<25 )- T1*r;\
    s+=m;


...
for(int i = 0; i < WorkPerThread; i++)
{
  s<<=2;
  lcn4(s, m, q, r, qinv);
  s<<=1;
  lcn(s, m, q, r, qinv);
  output[startindex + i] = r3i  * s;
}
```

Figure 1: A fragment of our implementation of the modified L'Ecuyer's method in C (as a macro).

3.1.2 Barrett's reduction

Next, we looked at Barrett's reduction [6]. Barrett introduced the idea of estimating the quotient $\lfloor \frac{x}{m} \rfloor$ with operations that are either less expensive in time than a multi-precision division by m or can be done as a pre-calculation for a given m. Barrett uses the approximation

$$\left\lfloor \frac{x}{m} \right\rfloor - 2 \leq \left\lfloor \frac{\left\lfloor \frac{x}{2^{k-1}} \right\rfloor \cdot \left\lfloor \frac{2^{k-1}2^{k+1}}{m} \right\rfloor}{2^{k+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{x}{2^{k-1}} \right\rfloor \cdot \left\lfloor \frac{2^{2k}}{m} \right\rfloor}{2^{k+1}} \right\rfloor \leq \left\lfloor \frac{x}{m} \right\rfloor$$

and the equation $x \mod m = x - m \left\lfloor \frac{x}{m} \right\rfloor$. The term $\lfloor \frac{2^{2k}}{m} \rfloor$ depends only on m and can be pre-computed. The other divisions are efficient binary shift operations. The value of k here is the number of binary digits of m. The algorithm reads as follows

Algorithm Barrett

Input: $x = (x_{2k-1} \ldots x_1 x_0)_2$,$m = (m_{k-1} \ldots, m_1 m_0)_2$, $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$, $x, m > 0$, $m_{k-1} \neq 0$

Output: $r = x \mod m$

1. $q_1 = \left\lfloor \frac{x}{2^{k-1}} \right\rfloor$, $q_2 = \mu \cdot q_1$, $q_3 = \left\lfloor \frac{q_2}{2^{k+1}} \right\rfloor$

2. $r_1 = x \mod 2^{k+1}$, $r_2 = q_3 \cdot m \mod 2^{k+1}$, $r = r_1 - r_2$

3. if $r < 0$ then $r = r + 2^{k+1}$

4. while $r \geq m$ do $r = r - m$

5. return r.

In our case $k = 53$, and μ is pre-computed in extended arithmetic but is a 64-bit integer itself. Integer q_2 needs extended precision, but q_1 and q_3 need only 64 bits using binary shift operations with high and low parts of q_2. Integer $q_3 \cdot m$ requires extended arithmetic. Finally r_1, r_2 and r are 64 bit integers. These observations allow one to reduce CPU time. At most two subtractions at step 4 are needed. The actual C code is presented in Figure 2.

```
#define barrett_step_simple(rlo)\
  xlo = t53 * rlo;\
  xhi = __umul64hi (t53, rlo);\
  qlo = (xhi << 12) | (xlo >> 52);\
  qhi = __umul64hi(qlo, mulo);\
  qlo = qlo * mulo;\
  qlo = (qhi << 10) | (qlo >> 54);\
  qhi = 0;\
  r1lo = xlo & 0x3FFFFFFFFFFFFFULL;\
  r2lo = qlo * m;\
  r2lo = r2lo & 0x3FFFFFFFFFFFFFULL;\
  rlo = r1lo - r2lo;\
  if (r1lo<r2lo) rlo += 0x40000000000000ULL;\
  while (rlo >= m) rlo -= m;
...
for(int i = 0; i < WorkPerThread; i++)
{
  barrett_step_simple(rlo);
  output[startindex + i] = r3i * rlo;
}
```

Figure 2: A fragment of our implementation of the Barrett's method in C (as a macro). Note that we use CUDA's operation `__umul64hi` which returns the most significant 64 bits of the product of two 64 bit integers.

3.1.3 Modified Barrett's reduction

We note that with our choice of $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$, at step 1 of Barrett's algorithm we have $q_1 = \left\lfloor \frac{x}{2^{53-1}} \right\rfloor = \left\lfloor \frac{2^{53}z}{2^{52}} \right\rfloor = 2z$, and consequently $q_2 = 2\mu z$. Similarly at Step 2 we have $r_1 =$

$2^{53}z \mod 2^{52} = z \mod 2$. We questioned whether these two operations can be eliminated altogether.

We rewrite Barrett's formula as follows

$$\left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor \cdot \left\lfloor \frac{2^k 2^k}{m} \right\rfloor}{2^k} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor \cdot \left\lfloor \frac{2^{2k}}{m} \right\rfloor}{2^k} \right\rfloor \leq \left\lfloor \frac{x}{m} \right\rfloor .$$

Then we can keep the same $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$, but the two mentioned operations simplify as follows:

1) at Step 1 we calculate $q_1 = \left\lfloor \frac{x}{2^{53}} \right\rfloor = \left\lfloor \frac{2^{53}z}{2^{53}} \right\rfloor = z$;

2) at Step 2 we have $r_1 = 2^{53}z \mod 2^{53} = 0$.

Therefore these two instructions become redundant, and hence we save on CPU time. Furthermore, since $r = 0 - r_2 < 0$, the `if` statement in Step 3 is redundant. Finally, because in our case $3^{33} = m < 2^k = 2^{53} < 2m$, and because $r < 2^{53}$, we have $r < 2m$ and therefore `while` in Step 4 can be replaced with a cheaper `if`. Our modified Barrett's algorithm becomes

Algorithm Modified Barrett

Input: $z = (z_{k-1} \dots z_1 z_0)_2$,$m = (m_{k-1} \dots, m_1 m_0)_2$, $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$, $x, m > 0$, $m_{k-1} \neq 0$

Output: $r = 2^k z \mod m$

1. $q_2 = \mu \cdot z$, $q_3 = \left\lfloor \frac{q_2}{2^k} \right\rfloor$
2. $r_2 = q_3 \cdot m \mod 2^k$
3. $r = 2^k - r_2$
4. if $r \geq m$ then $r = r - m$
5. return r.

Our C implementation of this algorithm is shown in Figure 3. Note that similar modular reduction techniques can be applied to the seeding step (2) of the BB algorithm. Here we need modular exponentiation, and repeated application of Barrett's (known as Barrett's exponentiation) will do the trick. We did not use this method in our implementation, as the seeding step is applied just once, and hence its efficiency makes only a marginal contribution.

The proposed three methods of reduction have a generation rate of 0.6 Gnumbers/sec, and still did not match the rate provided by `mtgp`. Therefore we also attempted to accelerate calculations using programming techniques. Our first suspicion was the speed of writing to GPU's global memory.

3.2 Writing to global memory

It is known that both global and shared memory access pattern affects the performance of GPU algorithms [22]. Strided access, when the values are read from or written to GPU

```
#define stepBarrett(z)\
  qhi =__umul64hi(z, mu);\   /* high part of $q_2$ */
  z *=  mu;\                 /* low part of $q_2$ */
  z = ( ((qhi << 11) | (z >> 53) )* m )&0x1FFFFFFFFFFFFFULL; \
  z= 0x20000000000000ULL - z; \  /* $2^53 - r_2$ */
  if (z >= m) z -= m;
...
for(int i = 0; i < WorkPerThread; i++)
{
  stepBarrett(z);
  output[startindex + i] = r3i * z;
}
```

Figure 3: A fragment of our implementation of the modified Barrett's method in C (as a macro). Note that x &0x1FFFFFFFFFFFFULL implements $x \mod 2^{53}$ operation and the shifts implement $q_2/2^{53}$. Operation `__umul64hi` which returns the most significant 64 bits of the product of two 64 bit integers

memory by each thread not consecutively but with certain steps, helps avoid bank conflicts and execute read/write operations more efficiently. While on many occasions the generated random numbers need not be stored in memory but rather used in a simulation routine on GPU, it is helpful to measure the generation rate when the random numbers are written into an array in the global memory. It allows one to benchmark the algorithm against the competitors in a fair way, and it also avoids the possibility that some instructions can be skipped by the optimizer if their results are not used. Furthermore, knowing the speed at which an algorithm writes arbitrary fixed values into global memory under the same memory access pattern, one can calculate the actual net generation speed.

For these reasons we also experimented with different steps at which threads write consecutively generated random numbers into global memory, as a function of the number of threads and thread blocks. As expected, letting each thread to write into consecutive positions in the output array was less efficient, because on GPUs read/writes to global memory are performed in blocks (of 128 bytes, as specified in CUDA Programmer Manual [22]). Therefore when p concurrent threads write to consecutive positions, they require just one block write for all p generated values (if they all fit one block), as opposed to p writes when they write to non-coalescent positions.

While the elements of the generated sequence will not be located in the global memory consecutively, this is not an issue, because the same access pattern can be applied when they are retrieved by a simulation algorithm (which is a more efficient pattern), therefore the sequence can be easily restored to its original order.

Our algorithm involves a parameter "step" (see Figure 4), calculated based on the number of threads and thread blocks. Our optimal configuration on Tesla C2050 involved 512 threads per block, and 4096 elements per thread, and therefore $\lceil \frac{n}{512 \cdot 4096} \rceil$ thread blocks. Thus *step*

was $512 \cdot blocks$.

Coalesced memory access improved the generation rate by a factor of more than ten.

3.3 Loop unrolling and inlining

Loop unrolling is a standard method for improving numerical performance. Loop unrolling allows for more computation to be performed without the overhead of managing loop counters and checking ending conditions, but comes at the expense of additional register usage. We unrolled the generation loop in each thread as in Figure 4. Unrolling the loop for Barrett's method improved generation rate by 4.9%, and the improvement to other methods was between 1 and 4%.

```
double r3i=1.0/5559060566555523.0;  // pow(3.0, -33.0);
unsigned long long mu = 0x33D9481681D79DULL; // pow(3.0, 33.0) div pow(2,53);
/* Calculate seed z using Eq. (2).*/
for(unsigned int i = 0; i < WorkPerThread; i+=8)
{
    stepBarrett(z);
    output[startindex + i*step] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*2] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*3] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*4] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*5] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*6] = r3i * z;
    stepBarrett(z);
    output[startindex + i*step+step*7] = r3i * z;
}
```

Figure 4: Partial loop unrolling for modified Barrett's method.

Another strategy is function inlining. It reduces computation time as no parameters need to go through stack. One can inline functions by using macros or inline directive. On GPU the device functions are automatically inlined, and we noticed only marginal benefit in using macro as opposed to inline function in our implementation of Barrett' algorithm.

4 Numerical experiments

The two aspects of the generator to consider are the statistical quality and the generation rate, presented in the following sections respectively.

4.1 Statistical testing

The testing suite BigCrush [16] was applied to the generator algorithm BB. Of the 106 statistical tests, all tests passed except for the Birthday Spacings tests #13-21 and the Close Pairs tests #22-24.

The Birthday Spacings tests assume the generation of a value of length 64bit. The BB algorithm generates a random value of length 53bits, which corresponds to the mantissa of a double value type. The restrictions placed on the parameters of the Birthday Spacings tests of BigCrush do not hold when considering only 53bits, therefore the failure of these tests is expected.

The Close Paris tests, as studied in [15], would expect an LCG to fail with period less than 2^{60}. Here we have a period of $P = 2 \cdot 3^{32}$, and a failed test. The rest of the tests (94 in total) were passed, and we conclude that the generated sequence is of a good statistical quality.

4.2 Numerical testing

Our numerical experiments were performed with an Intel Core i7-860 processor workstation with 4 GB RAM clocked at 2.8 GHz running Linux (Fedora 12), and with a Tesla C2050 GPU with 448 cores, 3 GB of global memory, and clocked at 1.15 GHz. We selected our version of Bailey and Borwein's algorithm [4] referred to as BCN, its 128-bit arithmetic implementation BCN128bit, the LCG adapted from L'Ecuyer [13] LCN, Barrett's reduction [6], and a modified version of Barrett's, refer to sections 2 and 3, for analysis. These methods were compared with a serial implementation of each, run on a CPU, and also compared with parallel generators, `mtgp` [24] and the CUDA-SDK Mersenne twister [20].

Each kernel was timed for 100 iterations for varying threads per block counts, ranging from 128 threads per block to 512 threads per block, with a step of 32 threads. The timing information for the best thread count was recorded. The need for different thread counts is attributed to the varying register demands of each kernel. The unrolled kernels in particular require more registers, affecting the occupancy and therefore can be tuned by varying the thread count.

The generation rate for the various methods, averaged over 100 runs, is given in Table 1 and illustrated in Figure 5. The table includes the rate at which numbers are generated with and without the time required to setup the generator, for example the time required to generate a starting seed for each thread in the BCN methods. The table shows the increase of the generation rate as techniques to speed up CUDA were applied, such as coalesced memory access and unrolling.

Table 1: The generation rate for the different methods proposed.

Method	Execute Rate GNum/sec	Including Setup GNum/sec
BCN	0.6706	0.6703
BCNCoalesced	9.1744	9.0347
BCNUnrolled	9.4756	9.3438
BCN128bit	0.6584	0.6582
BCN128bitCoalesced	3.8914	3.8690
BCN128bitUnrolled	3.9311	3.9140
LCN	0.6592	0.6590
LCNCoalesced	7.5974	7.5015
LCNUnrolled	8.1274	8.0422
Barrett	0.6660	0.6657
BarrettCoalesced	8.0413	7.9337
BarrettUnrolled	8.4334	8.3415
BarrettModified	11.4434	11.2268
Constant	0.6900	0.6897
ConstantCoalesced	12.4127	12.2891
ConstantUnrolled	12.6217	12.3587
mtgp	9.8455	9.5865
sdk MT	4.2408	4.0872
BCNSerial	-	0.024
LCNSerial	-	0.051
BarrettModifiedSerial	-	0.102
ConstantSerial	-	0.526

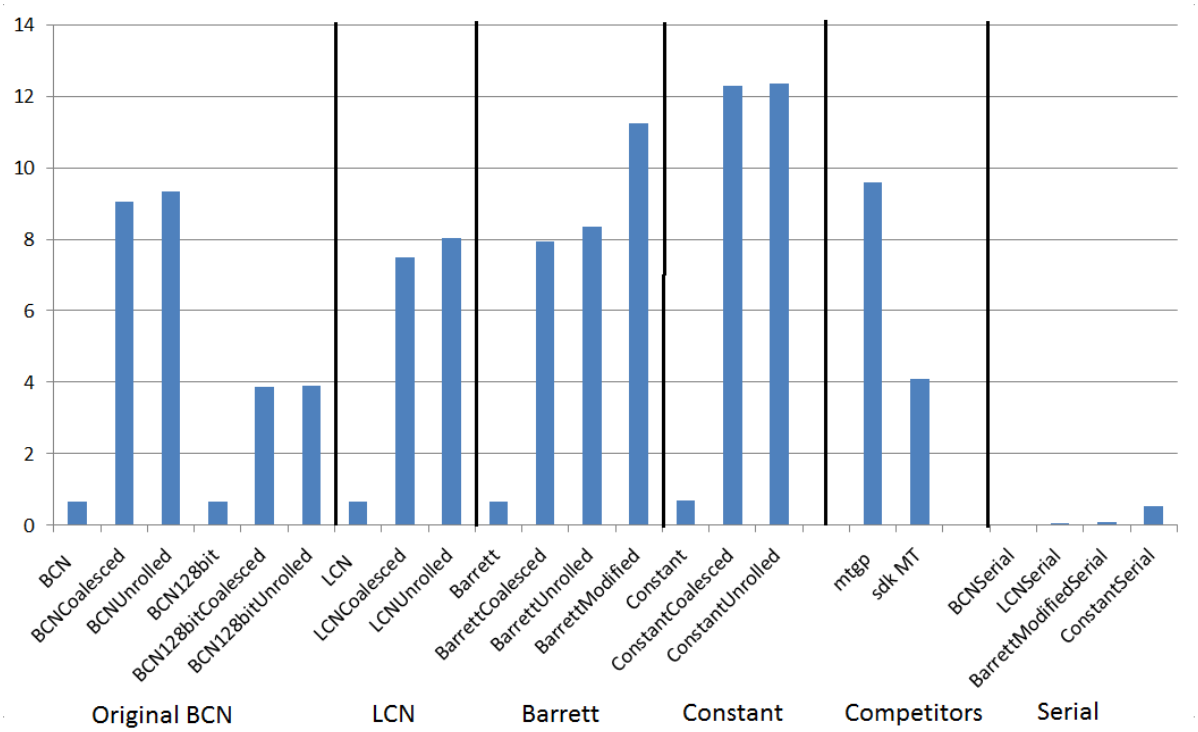

Figure 5: Generation rates of various methods in GNum/sec.

The Constant methods in the table refer to a simple CUDA implementation where a single constant value is written back to global memory. This method provides a maximum generation rate that we are able to obtain. Since the Constant methods use the same memory access as the random number generators, they describe the maximum rate achievable due to memory access bottlenecks.

In Table 1 it is evident that coalesced memory access increases the generation rate of the kernels, this is especially true in the case of memory bound algorithms such as the ones under analysis here. The improvements unrolling can make can also be seen, however this is a slight improvement when compared to the coalesced improvement.

From Table 1 it can be seen that the CPU-based serial methods are inferior to those of the generators running in parallel in CUDA. While global memory access is expensive in CUDA, on CPU it does not represent a bottleneck. Indeed, ConstantSerial method entry shows a very high memory write rate, which indicates that contribution due to memory access on CPU is negligible, and the serial algorithm is CPU bound. In contrast, the GPU algorithm is clearly memory bound.

The generation rate of the BarrettModified method can be seen to approach that of the Constant method. If we remove the memory access bottleneck from the timing results, as seen in Table 2, an appreciation of the net generation rate without expensive memory access can be made. Here the parallel optimized Barrett method exceeds the serial implementation by approximately 19%.

Table 2: The generation rate for various methods with memory access bottlenecks removed.

Method	RunTime (ms)	Generation Time (ms)	Generation Rate (GNum/sec)
BCNUnrolled	28.7293	7.0049	38.3206
BCN128bitUnrolled	68.5847	46.8604	5.7284
LCNUnrolled	33.3808	11.6565	23.0287
BarrettUnrolled	32.1831	10.4588	25.6658
BarrettOpt	23.9146	2.1902	122.5574
ConstantUnrolled	21.7243	-	12.3564

4.3 Serial methods under different compilers

Our GPU parallel implementation of the Barrett's method involves the function `__umul64hi` as part of one 128-bit multiplication, see Figure 3. This is CUDA's compiler `nvcc` intrinsic function, which is supported only on the GPU device. We experimented with several alternative implementations for CPU, which depend on both the compiler and hardware.

There is a generic implementation of this function available from `http://opencl-usu-2009.googlecode.com/svn/trunk/inc/dynlink/device_functions_dynlink.h`, presented in Figure 6.

The Microsoft Visual C++ compiler offers the intrinsic function `_umulh` which performs multiplication of two 64-bit unsigned integers and returns the high 64 bits of the 128bit result. The low 64 bits of the result can be obtained from the native arithmetic multiplication operator. This compiler also offers its `_umul128` intrinsic, which returns both hi and low quad-words of the signed multiplication.

The Gnu-C compiler on Linux provides native support for a 128 bit result of signed and unsigned 64 bit multiplications (when compiled with `-m64` and executed on x86_64 systems). This operation translates into just three assembler instructions on Intel 64-bit processors, and is therefore extremely efficient. The usage of these functions is also shown in Figure 6.

Therefore we expect that the efficiency of the generation on CPU will depend on the hardware and compiler being used, and also the compilation parameters. In Table 3 we present generation rates on different machines under different compilers. Cross-comparison should consider that the target chipsets were different for the three machines and thus direct comparison of numbers is not advised. The important part is the difference in generation rates of the alternative methods on the same architecture and compiler.

Given the differences in implementation of 128-bit multiplication, it is expected that performance of the algorithms presented in this paper will vary depending on CPU architecture and compiler, along with compilation parameters and the optimizations each compiler is capable of. Agner has shown that the Gnu compilers provide better optimization than Microsoft's Visual C++ compiler [1]. To evaluate these differences an optimal implementation of each algorithm (BCN, LCN and Barrett) was written for each compiler. Each algorithm was timed to produce 10^8 numbers starting with the same seed, and the results were averaged over ten runs. The standard library `rand()` function was also evaluated on the same task as

a baseline for comparison. Efforts were made to reduce timing complications due to cache flushes and process interruptions due to CPU scheduling. Unlike computations on GPU, the speed of memory write on CPUs did not contribute substantially to the overall performance (less than 10%), therefore we report the net generation rates only.

What is most noticeable is the performance of the modified Barrett method, which greatly exceeds the performance of the other algorithms across all compilers. In particular, the combination of this algorithm and a modern chipset (the i7-2760QM) provides exceptionally good performance compared with the other algorithms across the various architectures. Of note is the performance difference between the MSVC and LLVM-GCC compilers on the modified Barrett algorithm. The Windows version of the algorithm was run on an i7-2720QM processor, yet it achieved only 0.189 Gnum/sec on a 2.2 GHz clock cycle, whereas the Mac OSX version achieved 0.240 Gnum/sec (a factor of 1.27 better performance) on a 2.4 GHz clock cycle. It is likely that the bulk of the performance difference is due to the higher number of machine instructions needed by the MSVC compiler to perform the shift operation handling separately the hi and low quad-words of the 128 bit result of integer multiplications within this algorithm.

It is also interesting that the modified L'Ecuyer's method performed at least twice as well as the original BCN method, across all compilers. This method produced the highest generation rate in the debug mode (i.e., no compiler optimizations), but it lags behind Barrett in the release mode.

The speedup offered by the GPU is noticeable from Tables 2-3, and when excluding access to GPU global memory, is more than 1000-fold.

Table 3: The generation rate for various methods on CPU using different compilers and systems (Gnum/sec). We measured the rates on Intel i7-860 @ 2.8 GHz (Fedora), Intel Core i7-2720QM @ 2.2 GHz (Windows 7) and MacBook Pro Intel Core i7-2760QM @ 2.40 GHz (Mac OSX 10.7.3) respectively.

Compiler and architecture	Method			
	BCN	LCN	Barrett	rand
gcc 4.4 on Linux (Fedora 12) 32 bit implementation	0.023	0.020	0.031	0.062
gcc 4.4 on Linux (Fedora 12) 64 bit implementation	0.026	0.057	0.113	0.10
MSVC 10 on Windows 7 64 bit implementation	0.032	0.080	0.189	0.065
gcc 4.5 on Mac OSX 64 bit implementation	0.035	0.077	0.24	0.14

5 Conclusion

We presented a pseudorandom number generation algorithm based on Bailey and Borwein's work on normal numbers and their original implementation as a version of LCG. We confirmed that the generated sequence is statistically suitable for simulation purposes by running a comprehensive BigCrush suite of tests. We improved Bailey and Borwein's implementation by implementing a special modular reduction algorithm, which gave speedup of a factor of 4, in some cases 7, on a CPU. Our generator is twice as fast as the standard C function `rand()`.

Further, we implemented a parallel version of the proposed algorithm suitable for GPUs and benchmarked it against the state-of-the-art Mersenne Twister parallel generator `mtgp`. We found that our implementation is faster than `mtgp`, and our generation rate is close to the limiting rate imposed by the efficiency of writing to GPU's global memory. While MT's period is much longer than that of LCG, our LCG's advantage over `mtgp` is the simplicity and portability of implementation, the desirable skip-ahead property, independent generation of subsequences and less demand on GPU's scarce shared memory. We achieved speedup of 1000 times over the performance of random variate generator on CPU. Our implementation is available from `http://www.deakin.edu.au/~gleb/bcn_random.html`.

References

[1] F. Agner. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.* Copenhagen University College of Engineering. Available online at: http://www.agner.org/optimize/optimizing_cpp.pdf, Copenhagen, 2004-2012.

[2] A. C. Atkinson and M. C. Pearce. The computer generation of beta, gamma and normal random variables. *Journal of the Royal Statistical Society. Series A (General)*, 139:431–461, 1976.

[3] D. Bailey. High-precision software directory http://crd-legacy.lbl.gov/~dhbailey/mpdist/, accessed 1 September 2012.

[4] D. Bailey and J. Borwein. Normal numbers and pseudorandom generators. In *Proceedings of the Workshop on Computational and Analytical Mathematics in Honour of Jonathan Borwein's 60th Birthday, Springer, available online at: http://crd.lbl.gov/~dhbailey/dhbpapers/normal-pseudo.pdf*, 2011.

[5] D. Bailey and R. Crandall. Random generators and normal numbers. *Experimental Mathematics*, 11:527–546, 2000.

[6] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 311–323, London, UK, UK, 1987. Springer-Verlag.

[7] G. Beliakov. Class library ranlip for multivariate nonuniform random variate generation. *Computer Physics Communications*, 170:93–108, 2005.

[8] M Blum. How to generate cryptographically strong sequences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science*, pages 112–117, 1982.

[9] P. Bratley, F. Bennet, and L. Schrage. *A Guide to Simulation*. Springer-Verlag, 2nd ed. edition, 1987.

[10] D. Gladkov, J. Tapia, S. Alberts, and R. D'Souza. Graphics processing unit based direct simulation Monte Carlo. *Simulation*, 88:680–693, 2012.

[11] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer, Berlin; New York, 2004.

[12] L. Howes and D. Thomas. Efficient random number generation and application using CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 37. Addison Wesley, August 2007.

[13] P. L'Ecuyer. Efficient and portable combined random number generators. *Commun. ACM*, 31:742–751, 1988.

[14] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[15] P. L'Ecuyer, J. Cordeau, and R. Simard. Close-point spatial tests and their application to random number generators. *Operations Research*, 48:308–317, 2000.

[16] P. L'Ecuyer and R. Simard. Testu01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33, article 22, 2007.

[17] P. L'Ecuyer, R. Simard, J. Chen, and D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002.

[18] A.J. Menezes, P.C. van Oorschot, and S. A. Vanstone, editors. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1996.

[19] NVIDIA. CUDA Zone, http://www.nvidia.com/object/cuda_home.html, last accessed 1 June, 2012, 2012.

[20] NVIDIA. CUDA Zone SDK Samples, http://developer.nvidia.com/cuda-cc-sdk-code-samples, last accessed 1 June, 2012, 2012.

[21] NVIDIA. Double-double precision arithmetic, http://developer.nvidia.com/rdp/assets/double-double-precision-arithmetic, last accessed 4 June, 2012, 2012.

[22] NVIDIA. NVIDIA CUDA Programming Guide, http://http://developer.nvidia.com/nvidia-gpu-computing-documentation, last accessed 1 June, 2012, 2012.

[23] J. Passerat-Palmbach, C. Mazel, and D. Hill. Pseudo-random number generation on GP-GPU. In *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–8, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[24] M. Saito and M. Matsumoto. Variants of Mersenne twister suitable for graphic processors. *ArXiv e-prints 1005.4973, available online at: http://arxiv.org/abs/1005.4973*, May 2010.

```
/* Generic multiplication of two 64 bit integers */
typedef ULLong unsigned long long int;
inline unsigned int __umulhi(unsigned int a, unsigned int b)
{
  ULLong c = (ULLong)a * (ULLong)b;
  return (unsigned int)(c >> 32);
}

inline ULLong __umul64hi(ULLong a, ULLong b)
{
  unsigned int           a_lo = (unsigned int)a;
  ULLong a_hi = a >> 32;
  unsigned int           b_lo = (unsigned int)b;
  ULLong b_hi = b >> 32;
  ULLong m1 = a_lo * b_hi;
  ULLong m2 = a_hi * b_lo;
  unsigned int           carry;
  carry = (0ULL + __umulhi(a_lo, b_lo) + (unsigned int)m1 + (unsigned int)m2) >> 32;
  return a_hi * b_hi + (m1 >> 32) + (m2 >> 32) + carry;
}

/* 128 bit multiplication intrinsic for MS Visual C++ */
#include <intrin.h>
#pragma intrinsic(_umul128)
static inline _UINT128 mul128(UINT64 const & a, UINT64 const & b)
{
   _UINT128 ret;
   ret.ui64[LO] = _umul128(a,b,&ret.ui64[HI]);
   return ret;
}

double Barrett( UINT64 & z )
{
   register UINT64 qt;

#if defined (COMPILER_VS)
   _UINT128  q = mul128(z,mu);
   qt = ( ((q.ui64[HI] << 11 | q.ui64[LO] >> 53) * m ) & ULL(0x1FFFFFFFFFFFFF) );

#elif defined (COMPILER_GCC)
   qt = ( (mul128(z,mu) >> 53) * m ) & ULL(0x1FFFFFFFFFFFFF);

#else
   #error UNSUPPORTED COMPILER
#endif

   z = ULL(0x20000000000000) - qt;
   if (z >= m) z -= m;
   return r3i * z;
}
```