Parallelisation Techniques for Random Number Generators

Thomas Bradley* Jacques du Toit† Mike Giles‡
Robert Tong† Paul Woodhams†

In this chapter, we discuss the parallelisation of three very popular random number generators. In each case, the random number sequence which is generated is identical to that produced on a CPU by the standard sequential algorithm. The key to the parallelisation is that each CUDA thread block generates a particular block of numbers within the original sequence, and to do this it needs an efficient skip-ahead algorithm to jump to the start of its block.

Although the general approach is the same in the three cases, there are significant differences in the details of the implementation due to differences in the size of the state information required by each generator. This is perhaps the point of most general interest, the way in which consideration of the number of registers required, the details of data dependency in advancing the state, and the desire for memory coalescence in storing the output lead to different implementations in the three cases.

1 Introduction

Random number generation [3] is a key component of many forms of simulation, and fast parallel generation is particularly important for the naturally parallel Monte Carlo simulations which are used extensively in computational finance and many areas of computational science and engineering.

This article presents CUDA implementations of three of the most popular generators which appear in major commercial software libraries: L'Ecuyer's

*NVIDIA, 2/F 1310 Arlington Business Park, Theale, Berkshire RG7 4SA
†Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR
‡Oxford-Man Institute of Quantitative Finance, Eagle House, Walton Well Road, Oxford OX2 6ED

multiple recursive generator MRG32k3a, the Mersenne Twister MT19937 and the Sobol quasi-random generator. Although there is much in common in the underlying mathematical formulation of these three generators, there are also very significant differences and one of the aims of this article for a more general audience is to explain why these differences lead to quite different implementations.

In all three cases, there is a *state* Y_n , consisting of one or more variables, which can be advanced, step-by-step, by some algorithm

$$Y_{n+1} = f_1(Y_n)$$

from an initial value Y_0 . In addition, there is an output process

$$x_n = g(Y_n)$$

which generates an approximately uniformly distributed random number x_n .

The parallel implementations are made possible by the fact that each generator can efficiently "skip ahead" a given number of points using a state advance algorithm of the form

$$Y_{n+p} = f_p(Y_n),$$

with a cost which is $O(\log p)$ in general. The question then becomes how such a skip ahead should be used. Broadly speaking, there are three possible strategies:

- Simple skip ahead: each thread in each CUDA thread block performs a skip ahead to a specified point in the generator sequence, and then generates a contiguous segment of points. The skip aheads are chosen so that the segments are adjacent and do not overlap.
- Strided (or "leap frog") skip ahead: the n -th thread (out of N) generates points n , $n+N$, $n+2N$, etc.
- Hybrid: a large skip ahead is performed at the thread block level, and then within a block each thread does strided generation.

In outline, the approaches used in the three cases are as follows:

- MRG32k3a has a very small state and a very efficient skip ahead, so the simple skip ahead approach is used. However, care must be taken to achieve memory coalescence in writing the random outputs to device memory.

- The Sobol generator has a very small state and very efficient skip ahead. It could be implemented efficiently using the simple skip ahead approach, but it is slightly more efficient to use the hybrid approach to achieve simple memory coalescence when writing the output data.
- MT19937 has a very large state and a very slow skip ahead. However, special features of the state advance algorithm make it possible for the threads within a block to work together to advance a shared state. Hence the hybrid approach is adopted.

The next three sections examine each generator in detail and explain the CUDA implementation. We then present some benchmark figures comparing our implementations against the equivalent parallel generators in the Intel MKL/VSL libraries.

The software described in this chapter is available from The Numerical Algorithms Group (NAG); please see `www.nag.co.uk/numeric/gpus/` or contact `support@nag.co.uk`. In addition, the CUDA SDK example "SobolQRNG" has the source code for an implementation of the Sobol generator.

2 L'Ecuyer's Multiple Recursive Generator MRG32k3a

2.1 Formulation

L'Ecuyer studied Combined Multiple Recursive Generators (CMRG) in order to produce a generator that had good randomness properties with a long period, while at the same time being fairly straightforward to implement. The best known CMRG is MRG32k3a [4] which is defined by the set of equations

$$\begin{aligned} y_{1,n} &= \big(a_{12}\, y_{1,n-2} + a_{13}\, y_{1,n-3}\big) \mod m_1, \\ y_{2,n} &= \big(a_{21}\, y_{2,n-1} + a_{23}\, y_{2,n-3}\big) \mod m_2, \\ x_n &= \big(y_{1,n} + y_{2,n}\big) \mod m_1, \end{aligned} \tag{1}$$

for all $n \geq 3$ where

$$\begin{aligned} a_{12} &= 1403580 & a_{13} &= -810728 & m_1 &= 2^{32} - 209, \\ a_{21} &= 527612 & a_{23} &= -1370589 & m_2 &= 2^{32} - 22853\,. \end{aligned}$$

The sequence of integers $x_3, x_4, x_5 \ldots$ are the output of this generator. When divided by m_1 they give pseudo-random outputs with a uniform

distribution on the unit interval $[0, 1)$. These may then be transformed into various other distributions.

At any point in the sequence the state can be represented by the pair of vectors

$$Y_{i,n} = \begin{pmatrix} y_{i,n} \\ y_{i,n-1} \\ y_{i,n-2} \end{pmatrix}$$

for $i = 1, 2$. It follows that the two recurrences in (1) above can be represented as

$$Y_{i,n+1} = A_i \, Y_{i,n} \mod m_i$$

for $i = 1, 2$ where each A_i is a 3×3 matrix, and therefore

$$Y_{i,n+p} = A_i^p \, Y_{i,n} \mod m_i \tag{2}$$

for any $p \geq 0$.

2.2 Parallelisation

The parallel MRG generator has a small state (only six 32-bit integers) and requires few registers. The simple skip ahead strategy whereby each thread has its own copy of state and produces a contiguous segment of the MRG32k3a sequence (as specified in [4]) independently of all other threads works well, provided we have a very efficient way to skip ahead an arbitrary number of points. To efficiently compute A_i^p for large values of p, one can use the classic "divide and conquer" strategy of iteratively squaring the matrix A_i [8, 5]. It begins by writing

$$p = \sum_{j=0}^{k} g_j \, 2^j,$$

where $g_j \in \{0, 1\}$ and then computing the sequence

$$A_i, A_i^2, A_i^4, A_i^8, A_i^{16}, \ldots, A_i^{2^k}, \qquad \mod m_i$$

by successively squaring the previous term found. It is then a simple matter to compute

$$A_i^p \; Y_i = \prod_{j=0}^{k} A^{g_j \, 2^j} \; Y_i \qquad \mod m_i$$

for $i = 1, 2$ and the entire process can be completed in approximately $O(\log_2 p)$ steps.

We can improve the speed of this procedure at the cost of more memory by expanding the exponent p in a base higher than 2 so that

$$p = \sum_{j=0}^{k} g_j \, b^j,$$

for some $b > 2$ and $g_j \in \{0, 1, \ldots, b-1\}$. This improves the "granularity" of the expansion. To illustrate, suppose $b = 10$ and $p = 7$. Then computing $A^p Y$ requires only a single lookup from memory, namely the pre-computed value A^7, and a single matrix-vector product. However if $b=2$ then $A^7 Y = A^4(A^2(A\,Y))$ requiring three lookups and three matrix-vector products.

2.3 Implementation

We take $b = 8$ and pre-compute the set of matrices $A_i^{g_j b^k}$ for $g_j = 1, \ldots, 7$ and $k = 0, \ldots, 20$, namely

$$\begin{array}{cccc} A_i & A_i^2 & \cdots & A_i^7 \\ A_i^8 & A_i^{2\cdot 8} & \cdots & A_i^{7\cdot 8} \\ A_i^{8^2} & A_i^{2\cdot 8^2} & \cdots & A_i^{7\cdot 8^2} \\ \vdots & \vdots & \cdots & \vdots \\ A_i^{8^{20}} & A_i^{2\cdot 8^{20}} & \cdots & A_i^{7\cdot 8^{20}} \quad A_i^{8^{21}} \end{array}$$

for $i = 1, 2$ on the host. Since the state is so small only 10,656 bytes of memory are used and both sets of matrix powers are copied to constant memory on the GPU. Selecting from this large store of pre-computed matrix powers allows us to compute $A_i^p Y_i$ roughly three times faster than using $b=2$.

To generate a total of N random numbers, a kernel can be launched with any configuration of threads and blocks. Using T threads in total, the i-th thread for $1 \le i \le T$ will advance its state by $(i-1)N/T$ and will then generate N/T numbers.

The most efficient way to use the numbers generated in this manner is to consume them as they are produced, without writing them to global memory. Since any configuration of threads and blocks can be used, and since the MRG generator is so light on resources, it can be embedded directly in an application's kernel. If this is not desired, output can be stored in

global memory for subsequent use. Note that since each thread generates a contiguous segment of random numbers, writes to global memory will *not* be coalesced if they are stored to correspond to the sequence produced by the serial algorithm described in [4], in other words if the j-th value (counting from zero) produced by a given thread is stored at

```
storage[j+p*threadIdx.x+p*blockIdx.x*blockDim.x]
```

where $p = N/T$ and blocks and grids are one dimensional. Coalesced access can be regained either by re-ordering output through shared memory, or by simply writing the j-th number from each thread to

```
storage[threadIdx.x+j*blockDim.x+p*blockIdx.x*blockDim.x].
```

This will result in a sequence in global memory which has a different ordering to the MRG32k3a sequence described in [4].

3 Sobol Generator

Sobol [12] proposed his sequence as an alternative method of performing numerical integration in a unit hypercube. The idea is to construct a sequence which fills the cube in a regular manner. The integral is then approximated by a simple average of the function values at these points. This approach is very successful in higher dimensions where classical quadrature techniques are very expensive.

Sobol's method for constructing his sequence was improved by Antonov and Saleev [1]. Using Gray code, they showed that if the order of the points in the sequence is permuted, a recurrence relation can be found whereby the i+1-th point can be generated directly from the i-th point in a simple manner. Using this technique, Bratley and Fox [2] give an efficient C algorithm for generating Sobol sequences, and this algorithm was used as the starting point for our GPU implementation.

3.1 Formulation

We will briefly discuss how to generate Sobol sequences in the unit cube. For more details and further discussion we refer to [2]. A D-dimensional Sobol sequence is composed of D different one dimensional Sobol sequences. We therefore examine a one dimensional sequence.

When generating at most 2^{32} points, a Sobol sequence is defined by a set of 32-bit integers $m_i,\ 1 \leq i \leq 32$ known as *direction numbers*. Using

these direction numbers the sequence $y_1, y_2, \ldots$ is computed from

$$\begin{aligned} y_n &= g_1 m_1 \oplus g_2 m_2 \oplus g_3 m_3 \oplus \cdots \\ &= y_{n-1} \oplus m_{f(n-1)} \end{aligned} \tag{3}$$

starting from $y_0 = 0$. Here $\oplus$ denotes the binary exclusive-or operator, and the g_i's are the bits in the binary expansion of the Gray code representation of n. The Gray code of n is given by $n \oplus (n/2)$, and so $n \oplus (n/2) = \ldots g_3 g_2 g_1$. The function $f(n)$ above returns the index of the rightmost zero bit in the binary expansion of n. Finally, to obtain our Sobol sequence $x_1, x_2, \ldots$ we set $x_n = 2^{-32} y_n$.

To obtain multidimensional Sobol sequences, different direction numbers are used for each dimension. Care must be taken in choosing these, since poor choices can easily destroy the multidimensional uniformity properties of the sequence. For more details we refer to [10].

3.2 Parallelisation

The first expression in (3) gives a formula for directly computing y_n whereas the second expression gives an efficient algorithm for computing y_n from the value of y_{n-1}. The first formula therefore allows us to skip ahead to the point y_n. This skip ahead is quite fast as it requires a loop with at most 32 iterations, and each iteration performs a bit shift and (possibly) an xor. We could therefore have parallelized this generator along the same lines as the MRG32k3a generator above, with threads performing a skip ahead and then generating a block of points. However, there is another option.

Recall the second expression in (3) above, fix $n \geq 1$ and consider what happens as n increases to $n+8$. If $n = \ldots b_3 b_2 b_1$ denotes the bit pattern of n, clearly the last three bits $b_3 b_2 b_1$ remain unchanged when we add 8 to n: adding 1 to n eight times results in flipping b_1 eight times, flipping b_2 four times and flipping b_3 twice. It follows that $b_3 b_2 b_1$ will enumerate all permutations of 3 bits as n increases to $n+8$. Consider now what happens to $f(n+i)$ for $1 \leq i \leq 8$: since we are enumerating all permutations of $b_3 b_2 b_1$ we will have $f(n+i) = 1$ four times, $f(n+i) = 2$ twice, $f(n+i) = 3$ once, and $f(n+i) > 3$ once. Returning to (3) and recalling that two exclusive-ors cancel (i.e. $y_n \oplus m_i \oplus m_i = y_n$), we see that

$$\begin{aligned} y_{n+8} &= y_n \oplus \overbrace{m_1 \cdots m_1}^{\text{four times}} \oplus \overbrace{m_2 \cdots m_2}^{\text{twice}} \oplus m_3 \oplus m_{q_n} \\ &= y_n \oplus m_3 \oplus m_{q_n} \end{aligned}$$

for some $q_n > 3$. This analysis can be repeated for any power of 2: in general,

$$y_{n+2^p} = y_n \oplus m_p \oplus m_{q_n} \tag{4}$$

for some $q_n > p$ given by $q_n = f(n|(2^p - 1))$ where $|$ denotes the bitwise or operator. This gives an extremely efficient algorithm for strided (or "leap frog") generation, which in turn is good for memory coalescing (see below).

3.3 Implementation

Our algorithm works as follows. The m_i values are precomputed on the host and copied to the device. In a 32 bit Sobol sequence, each dimension requires at most 32 of the m_i values. Since individual dimensions of the D dimensional Sobol sequence are independent, it makes sense to use one block to compute the points of each dimension (more than one block can be used, but suppose for now there is only one). For each Sobol dimension then, a block is launched with 2^p threads for some $p \geq 6$ and the 32 m_i values for that dimension are copied to shared memory. Within the block, the i-th thread skips ahead to the value y_i using the first expression in (3) above. Since p is typically small (around 6 or 7), the skip ahead loop will have few iterations (around 6 or 7) since the bit pattern of $i \oplus i/2$ will contain mostly zeros. The thread then iteratively generates points $y_i, y_{i+2^p}, y_{i+2\cdot 2^p}, \ldots$ using (4). Note that m_p is fixed throughout this iteration: all that is needed at each step is the previous y value and the new value of q_n. Writes to global memory are easily coalesced since successive threads in a warp generate successive values in the Sobol sequence. In global memory we store all the numbers for the first Sobol dimension first, then all the numbers for the second Sobol dimension, and so on. Therefore if N points were generated from a D dimensional Sobol sequence and stored in an array `x`, the i-th value of the d-th dimension would be located at `x[d*N+i]` where $0 \leq i < N$ and $0 \leq d < D$.

As a final tuning of the algorithm, additional blocks can be launched for each dimension as long as the number of blocks cooperating on a given dimension is a power of 2. In this case if 2^b blocks cooperate, the i-th thread in a block simply generates the points $y_i, y_{i+2^{p+b}}, y_{i+2\cdot 2^{p+b}}, \ldots$

4 Mersenne Twister MT19937

The Mersenne Twister (MT19937) is a pseudo-random number generator proposed by Matsumoto and Nishumira [11]. It is a twisted generalized feedback shift register (TGFSR) generator featuring state bit reflection and tempering. The generator has a very long period of $2^{19937} - 1$ and good multidimensional uniformity and statistical properties. Since the generator is also relatively fast compared to similar quality algorithms, it is widely used in simulations where huge quantities of high quality random numbers are required.

We start by discussing the rather complex-looking mathematical formulation. We then present the relatively simple sequential implementation, which some readers may prefer to take as the specification of the algorithm, before proceeding to the parallel implementation.

4.1 Formulation

A twisted generalized feedback shift register (TGFSR) generator is based on the linear recurrence

$$X_{k+N} = X_{k+M} \oplus X_k D$$

for all $k \geq 0$ where $M < N \in \mathbb{N}$ are given and fixed. Each value X_i has a word length of w, i.e. is represented by w 0-1 bits. The value D is a $w \times w$ matrix with 0-1 entries, the matrix multiplication in the last term is performed modulo 2, and $\oplus$ is again bitwise exclusive-or which corresponds to bitwise addition modulo 2. These types of generators have several advantages: they are easy to initialise (note that we need N seed values), they have very long periods, and they have good statistical properties.

The Mersenne Twister defines a family of TGFSR generators with a separate output function for converting state elements into random numbers. The output function applies a *tempering transform* to each generated value X_k before returning it (see [11] for further details) where the transform is chosen to improve the statistical properties of the generator. The family of generators is based on the recurrence

$$X_{k+N} = X_{k+M} \oplus (X_k^u | X_{k+1}^\ell) D \tag{5}$$

for all $k \geq 0$ where $M < N \in \mathbb{N}$ are fixed values and each X_i has a word length of w. The expression $(X_k^u | X_{k+1}^\ell)$ denotes the concatenation of the

$w-r$ most significant bits of X_k and the r least significant bits of X_{k+1} for some $0 \le r \le w$, and the $w \times w$ bit-matrix D is given by

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \\ d_{w-1} & d_{w-2} & d_{w-3} & d_{w-4} & \cdots & d_1 & d_0 \end{pmatrix}$$

where all the entries d_i are either zero or one. The matrix multiplication in (5) is performed bitwise modulo 2.

The popular Mersenne Twister MT19937 [11] is based on this scheme with $w = 32$, $N = 624$, $M = 397$, $r = 31$, $d_{31}d_{30}\ldots d_1d_0 = 2567483615$, and has a period equal to the Mersenne prime $2^{19937}-1$. The state vector of MT19937 consists of 19,937 *bits* — 623 unsigned 32-bit words plus one bit — and is stored as 624 32-bit words. We denote this state vector by

$$Y_k = \begin{pmatrix} X_{N-1+k} \\ X_{N-2+k} \\ \vdots \\ X_k \end{pmatrix}$$

for all $k \ge 0$ where Y_0 denotes the initial seed. Reading from top to bottom the first 19,937 bits are used and the bottom 31 bits (the 31 least significant bits of X_k) are ignored. If the generator is considered as an operation on individual *bits*, it can be recast in the form

$$Y_{k+1} = A\,Y_k \quad \text{and} \quad Y_k = A^k\,Y_0 \tag{6}$$

where A is a matrix of dimension 19,937 with elements having value 0 or 1 and the multiplication is performed mod 2. For the explicit form of this matrix, see [11].

Matsumoto and Nishimura give a simple C implementation of their algorithm which is shown in Listing 1. Their implementation updates all 624 elements of state at once so that the `state` variable contains $A^{624n}\,Y_0$ for $n \ge 0$. The subsequent 624 function calls each produce a single random number without updating any state, and the 625-th call will again update the state.

```c
#define N 624
#define M 397
#define UPPER_MASK 0x80000000UL  /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL  /* least significant r bits */
static unsigned long state[N];   /* the array for the state vector  */
static int stateIdx = N+1;     /* stateIdx==N+1 means state[N] uninitialized */

/* Generates a random number in [0, 0xffffffff] */
unsigned int genrand_int32(void) {
    static unsigned long constants[2]={0x0UL, 0x9908b0dfUL};
    unsigned long y;
    /* UPDATING STATE */
    if (stateIdx >= N) { /* generate N words at one time */
        int k;
        for (k=0; k<N-M; k++) {
            y = (state[k]&UPPER_MASK) | (state[k+1]&LOWER_MASK);
            state[k] = state[k+M] ^ (y >> 1) ^ constants[y&0x1UL];
        }
        for (; k<N-1; k++) {
            y = (state[k]&UPPER_MASK) | (state[k+1]&LOWER_MASK);
            state[k] = state[k+(M-N)] ^ (y >> 1) ^ constants[y&0x1UL];
        }
        y = (state[N-1]&UPPER_MASK) | (state[0]&LOWER_MASK);
        state[N-1] = state[M-1] ^ (y >> 1) ^ constants[y&0x1UL];
        stateIdx = 0;
    }
    /* GENERATING */
    y = state[stateIdx++];
    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7)  & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);
    return y;
}
```

Listing 1: Serial implementation of MT19937.

4.2 Parallelisation

The state size of the Mersenne Twister is too big for each thread to have its own copy. Therefore the per-thread parallelisation strategy used for the MRG32k3a is ruled out, as is the strided generation strategy used for the Sobol generator. Instead, threads within a block have to cooperate to update state and generate numbers, and the level to which this can be achieved determines the performance of the generator. Note from (5) that the process of advancing state is quite cheap, involving 3 state elements and 7 bit operations.

We follow a hybrid strategy. Each block will skip the state ahead to a given offset, and the threads will then generate a contiguous segment of points from the MT19937 sequence by striding (or leap-frogging). There are three main procedures: skipping ahead to a given point, advancing the state, and generating points from the state. We will examine how to parallelise the latter two procedures first, and then return to the question of skipping ahead.

4.3 Updating State and Generating Points

Generating X_{k+N} for any $k \geq 0$ requires the values of X_k, X_{k+1}, X_{k+M} where $N = 624$ and $M = 397$. In particular, $X_{(N-M)+N}$ requires $X_{(N-M)+M} = X_N$. If there are T threads in a block and the i-th thread generates $X_{N+i}, X_{N+i+T}, X_{N+i+2T}, \ldots$ for $0 \leq i < T$, then we see that we must have $T \leq N - M = 227$ since otherwise thread $N - M$ would require X_N, a value which will be generated by thread 0. To avoid dependence between threads, we are limited to fewer than 227 threads per block. We will use 1D blocks with 224 threads, since this is a multiple of 32.

We implement state as a circular buffer in shared memory of length $N + 224$, and we update 224 elements at a time. We begin by generating 224 random numbers from the initial seed, and then updating 224 elements of state and storing them at locations `state[N...N+223]`. This process repeats as often as needed, with the writing indices wrapping around the buffer. All indices except writes to global memory are computed modulo $N+224$. The code is illustrated in Listing 2. As was the case with the Sobol generator, writes to global memory are easily coalesced since the threads cooperate in a leap-frog manner.

```
#define N2   (N+224)
    /* ... kernel function signature, etc ... */
    __shared__ unsigned int state[N2];
    /* ... copy values into state from global memory ... */
    output_start = ... // determine where block starts
        generating points

    int k1 = threadIdx.x;
    int k2 = k1 + 1;
    int k3 = k1 + M;
    int k4 = k1 + N;
    int k5 = output_start + k1;
    int num_loops = ... // Number of 224-updates of state
    for(; num_loops>0; num_loops--) {
        /* GENERATING */
        y = state[k1];
        y ^= (y >> 11);
        y ^= (y << 7) & 0x9d2c5680UL;
        y ^= (y << 15) & 0xefc60000UL;
        y ^= (y >> 18);
        global_mem_storage[k5] = y; // Modify to change
            output distributions

        /* UPDATING STATE */
        y = (state[k1]&UPPER_MASK) | (state[k2]&LOWER_MASK);
        state[k4] = state[k3] ^ (y >> 1) ^ constants[y&0x1UL
            ];

        k1 += 224;  k2 += 224;  k3 += 224;  k4 += 224;  k5
            += 224;
        if (k1>=N2) k1 -= N2;
        if (k2>=N2) k2 -= N2;
        if (k3>=N2) k3 -= N2;
        if (k4>=N2) k4 -= N2;
        __syncthreads();
    }
    // Tidy up the last few points ...
```

Listing 2: CUDA code for generating points and updating state for MT19937. The code follows the general notation of Listing 1.

4.4 Skipping Ahead

We now consider how blocks can skip ahead to the correct point in the sequence. Given a certain number of Mersenne points to generate, we wish to determine how many points each block should produce, and then skip that block ahead by the number of points all the preceding blocks will generate. For this we need a method to skip a single block ahead by a given number of points in the Mersenne sequence.

Such a skip ahead algorithm was presented by Haramoto et al. [9]. Recall that $Y_n = A^n Y_0$ for $n \geq 0$ where A is a $19{,}937 \times 19{,}937$ matrix. However computing A^n even through a repeated squaring (or "divide-and-conquer") strategy is prohibitively expensive and would require a lot of memory. Instead, a different approach is followed in [9] based on polynomials in the field $\mathbb{F}_2$ (the field with elements $\{0, 1\}$ and where all operations are performed modulo 2). Briefly, they show that for any $v \in \mathbb{N}$ we have

$$\begin{aligned} A^v\, Y_0 &= g_v(A) Y_0 \\ &= \Big(a_k A^{k-1} + a_{k-1} A_{k-2} + \cdots + a_2 A + a_1 I \Big) Y_0 \\ &= a_k Y_{k-1} + a_{k-1} Y_{k-2} + \cdots + a_2 Y_1 + a_1\, Y_0 \end{aligned} \tag{7}$$

where $k = 19937$ and $g_v(x) = a_k x^{k-1} + \ldots + a_2 x + a_1$ is a polynomial over $\mathbb{F}_2$ which depends on v. A formula is given for determining g_v for any $v \geq 1$ given and fixed. Note that each of the coefficients a_i of g_v are either zero or one.

Figure 1 shows the time taken for a single block to perform the various tasks in the MT19937 algorithm: calculate g_v on the host; compute $g_v(A)Y_0$; perform v updates of state (not generating points from state); and generating v values with all updates of state. While evaluating $g_v(A)Y_0$ is fairly expensive (equivalent to generating about 2,000,000 points and advancing state about 2,300,000 times), computing the polynomial g_v is is much more so: one could generate almost 13,000,000 points in the same time, and advance state almost 20,000,000 times. Updating state and generating points scale linearly with v.

Clearly we would prefer not to calculate g_v on the fly. We would also prefer not to perform the calculation $g_v(A)Y_0$ in a separate kernel since the first block never has to skip ahead and this would prevent it from generating points until the calculation was finished. However there is the question of load balance: the first block can generate around 2,000,000 points in the time it takes the second block to skip ahead. Since the total runtime is equal to the runtime of the slowest block, if the number of blocks equals

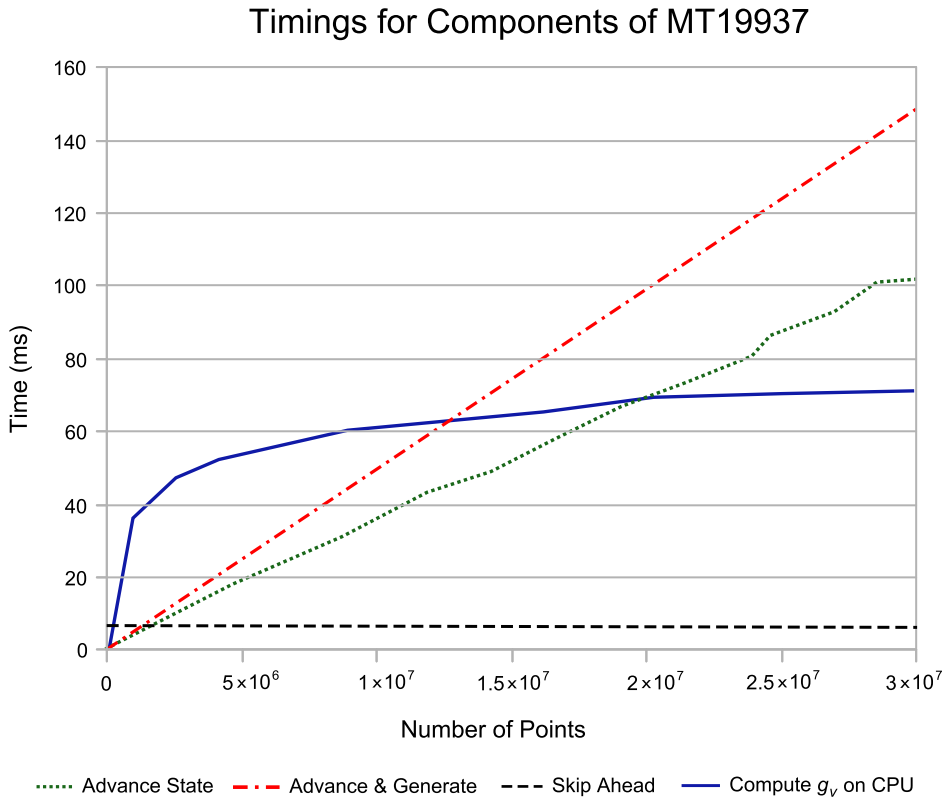

Figure 1: Time (in ms) for one block of 224 threads to: (a) advance state v times (without generating points); (b) advance state v times and generate points; (c) apply the skip ahead algorithm in equation (7); (d) compute the skip ahead polynomial $g_v(x)$ on the CPU (Xeon E5502). Number of points v ranges from 1000 to 30 million and GPU code was run on a Quadro FX5800.

the number of streaming multiprocessors in the GPU, it is clear that blocks should not all generate the same number of points. A mathematical analysis of the runtimes of each block, coupled with experimental measurements, can be done and this yields formulae for the optimal number of points each block should generate as well as how far the block should skip ahead. This depends on the sample size and so requires us to calculate g_v on the fly.

There are a number of options when choosing a skip ahead strategy for the Mersenne Twister, and it is not completely clear which approach is best:

- When a huge number of points are to be generated the only option may be to generate the necessary polynomials on the fly and perform

(7) on the device. In this case the generator will typically be embedded in a larger application so that the cost of computing the polynomials is small compared to the total runtime.

- When a small number of points are to be generated it may be more efficient to have a producer-consumer model where one (or a few) blocks advance state and write the values to global memory, while other blocks read these values in and generate points from the necessary distributions. This requires rather sophisticated inter-block communication using global memory locks and relies on the update of state (5) being much cheaper than converting state into a given distribution. It is not clear how useful this would be, and indeed for smaller samples it will probably be faster (and simpler) to compute the numbers on the host.

- When a medium number of points are to be generated it is possible to pre-compute and store a selection of polynomials $\{g_v\}_{v\in\mathcal{V}}$ for some $\mathcal{V} \subset \mathbb{N}$. At runtime a suitable subset of these can be chosen and copied to the device where the skip ahead is then performed. This is the approach we have adopted. The difficulty here is deciding on $\mathcal{V}$ and on which subset to choose at runtime. Formulae can be developed to help with this, but it is still a rather tricky problem.

- Lastly we can always calculate g_{v_i} and $g_{v_i}(A)Y_0$ for all necessary skip points v_i on the host and then copy the advanced states to the device to generate random numbers. This is currently impractical, but the upcoming Westmere family of Intel CPUs contains a new instruction `PCLMULQDQ` – a bitwise carry-less multiply – which is ideally suited to this task; see [7] for further details.

5 Performance Benchmarks

We compared our CUDA implementations to the Intel MKL/VSL library's random number generators. Benchmarks were done in both single and double precision on a Tesla C1060 and a Tesla C2050 (which uses the new Fermi micro architecture). The test system configuration is detailed in Table 1 below.

The Intel random number generators are contained in the Vector Statistical Library (VSL). This library is not multithreaded, but is thread safe and contains all the necessary skip ahead functions to advance the generators' states. We used OpenMP to parallelise the VSL generators and obtained

Hardware	Intel Xeon E5410 2.33GHz with 8GB RAM for Tesla C1060 Intel Core i7 860 2.80GHz with 8GB RAM for Tesla C2050
Operating System	Windows XP 64bit SP2 for Tesla C1060 Windows 7 Professional for Tesla C2050
C++ Compiler	Intel C++ Compiler Pro v11.1
C++ Options	`/O2 /Og /Ot /Qipo /D "WIN32" /D "NDEBUG" /EHsc /MD /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /nologo /GS- /fp:strict /Qfp-speculation:off /W3 /Wp64 /Zi /Qopenmp /QxHost /Quse-intel-optimized-headers`
NVIDIA Toolkit	CUDA 2.3 for Tesla C1060 CUDA 3.0 for Tesla C2050
NVIDIA GPU	Tesla C1060 using NVIDIA Driver v190.38 Tesla C2050 using NVIDIA Driver v197.68
NVCC Options	`-O2 -D_CONSOLE -arch compute_13 -code sm_13 --host-compilation C++ -Xcompiler /MT -m 64`

Table 1: Test systems used for benchmarking. Tesla C2050 uses the new Fermi micro architecture.

figures when running 1, 2, 3 and 4 CPU threads. Timing of the CPU code was done using hardware high-resolution performance counters; the GPU code was timed using CUDA events and `cudaEventElapsedTime`. Ideally an application would consume the random numbers on the GPU, however if the random numbers were required by CPU code then there may be an additional cost to copy the data back from the GPU to the host. A fixed problem size of 2^{25} was chosen so that each generator produced $33,554,432$ floating point numbers. This corresponds to 134MB of data in single precision and 268MB in double precision. For the Sobol generators we chose $2^7 = 128$ dimensions and generated $2^{18} = 262,144$ multidimensional points. Note that the VSL does not have skip ahead functions for the MT19937 generator so that it was not possible to parallelise this generator.

We produced uniform, exponential and Normal random numbers. For the MRG the Normal random numbers were obtained through a Box-Muller transform, while for the Sobol and Mersenne generators the Normal random numbers were obtained by inverting the cumulative Normal distribution function using an efficient implementation of the inverse error function [6]. Exponential numbers were obtained by inverting the cumulative exponential distribution function. The results are given in Tables 2 and 3.

All benchmarks were performed after a clean reboot of the workstation, with only a command prompt open. There does seem to be some variability

Generators		Tesla GPU	Intel MKL on Xeon E5410			
		GPU pts/ms	1 Thread	2 Threads	3 Threads	4 Threads
MRG	Unif	3.6151E+06	41.161x	24.774x	19.639x	16.247x
		3.1202E+06	**45.528x**	**31.094x**	**30.887x**	**29.889x**
	Exp	2.8280E+06	39.545x	23.222x	17.882x	14.964x
		6.8651E+05	**12.329x**	**7.634x**	**6.724x**	**6.468x**
	Norm	2.6647E+06	47.043x	25.619x	18.498x	15.188x
		6.5853E+05	**18.012x**	**10.257x**	**7.463x**	**6.215x**
Sobol	Unif	1.5790E+07	100.51x	93.976x	94.179x	64.556x
		9.2006E+06	**97.591x**	**90.396x**	**84.130x**	**88.832x**
	Exp	6.8723E+06	52.591x	41.702x	33.784x	32.251x
		7.8709E+05	**10.622x**	**8.683x**	**7.398x**	**7.432x**
	Norm	7.4239E+06	57.079x	45.129x	35.896x	34.820x
		4.0799E+05	**5.516x**	**4.578x**	**3.915x**	**3.856x**
Mersenne	Unif	2.6721E+06	25.051x			
		2.5762E+06	**40.320x**			
	Exp	2.0741E+06	24.758x			
		5.9492E+05	**19.728x**			
	Norm	2.0657E+06	24.856x			
		3.1229E+05	**5.8127x**			

Table 2: Benchmark figures for Tesla C1060 vs. Intel Xeon E5410. Values in **bold type** are double precision, other values are single precision. Columns "1 Thread" through "4 Threads" show speedup of GPU vs. CPU, i.e. (GPU pts/ms) ÷ (CPU pts/ms). Generators produced 2^{25} points: Sobol generators produced 2^{18} points of 2^7 dimensions each. Test system is as detailed in Table 1.

in the figures across different runs, depending on system load, but this is small enough to be ignored. For the Mersenne Twister we pre-computed the skip ahead polynomials g_v for $v = 1 \times 10^6, 2 \times 10^6, \ldots, 32 \times 10^6$ and then used a combination of redundant state advance (without generating points) and runtime equations to find a good work load for each thread block.

The Fermi card is roughly twice as fast as the Tesla in single precision, and roughly four times as fast in double precision. The exception to this is the Sobol single precision figures, which are very similar between the two cards. It may be that the Sobol generator is bandwidth limited since Sobol values (in single precision) are so cheap to compute.

Generators		Fermi GPU	Intel MKL on Xeon E5410			
		GPU pts/ms	1 Thread	2 Threads	3 Threads	4 Threads
MRG	Unif	7.7127E+06	88.108x	52.854x	41.900x	34.622x
		7.4453E+06	**108.64x**	**74.197x**	**73.703x**	**71.321x**
	Exp	5.4368E+06	76.024x	44.643x	34.378x	28.767x
		2.6696E+06	**47.935x**	**29.682x**	**26.143x**	**25.148x**
	Norm	4.6129E+06	81.436x	44.348x	32.022x	26.291x
		2.4418E+06	**66.789x**	**38.034x**	**27.673x**	**23.044x**
Sobol	Unif	1.7434E+07	110.97x	103.76x	103.98x	71.724x
		1.3452E+07	**142.68x**	**132.16x**	**123.00x**	**129.88x**
	Exp	7.9361E+06	60.732x	48.157x	39.014x	37.243x
		3.2094E+06	**43.312x**	**35.404x**	**30.168x**	**30.304x**
	Norm	8.6020E+06	66.137x	52.291x	41.593x	40.346x
		1.6202E+06	**21.904x**	**18.179x**	**15.547x**	**15.314x**
Mersenne	Unif	2.9077E+06	27.260x			
		2.8728E+06	**44.961x**			
	Exp	2.2352E+06	26.680x			
		1.2465E+06	**23.097x**			
	Norm	2.1965E+06	26.430x			
		8.8145E+05	**16.407x**			

Table 3: Benchmark figures for Tesla C2050 vs. Intel Xeon E5410. Values in **bold type** are double precision, other values are single precision. Columns "1 Thread" through "4 Threads" show speedup of GPU vs. CPU, i.e. (GPU pts/ms) ÷ (CPU pts/ms). Generators produced 2^{25} points: Sobol generators produced 2^{18} points of 2^7 dimensions each. Test system is as detailed in Table 1.

Acknowledgements

Jacques du Toit thanks the UK Technology Strategy Board for funding his KTP Associate position with the Smith Institute, and Mike Giles thanks the Oxford-Man Institute of Quantitative Finance for their support.

References

[1] Antonov, I. A. *and* Saleev, V. M. (1979). An economic method of computing LP_τ sequences. *USSR Journal of Computational Mathematics and Mathematical Physics,* 19 (252-256)

[2] Bratley, P. *and* Fox, B. (1988). Algorithm 659: implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mod-*

eling and Computer Simulation, 14:1 (88-100)

[3] L'Ecuyer, P. (2006). Uniform Random Number Generation. In: Henderson, S.G, Nelson, B.L. (eds.) *Simulation Handbooks in Oper. Res. and Manag. Sci.*, pp. 55–81. Elsevier, Amsterdam.

[4] L'Ecuyer, P. (1999). Good parameter sets for combined multiple recursive random number generators. *Operations Research,* 47:1 (159-164)

[5] L'Ecuyer, P, Simar, R, Chen, E. J, *and* Kelton, W. D. (2002). An object oriented random number package with many long streams and substreams. *Operations Research,* 50:6 (1073-1075)

[6] Giles, M (2010). Approximating the `erfinv` function. *GPU Gems 4, volume 2*

[7] Gueron, S. *and* Kounavis, M. E. (2010). Intel carry-less multiplication instruction and its usage for computing the GCM mode. Intel White Paper, available at http://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode/.

[8] Knuth, D. (1997). *The Art of Computer Programming, Volume 2, 3rd Edition.* Addison-Wesley Professional.

[9] Haromoto, H, Matsumoto, M, Nishumira, T, Panneton, F. *and* L'Ecuyer, P (2008). Efficient jump ahead for $\mathbb{F}_2$ -linear random number generators. *INFORMS Journal on Computing,* 20:3 (385-390)

[10] Joe, S. *and* Kuo, F. Y. (2008). Constructing Sobol sequences with better two dimensional projects. *SIAM Journal of Scientific Computing,* 30 (2635-2654)

[11] Matsumoto, M. *and* Nishumira, T. (1998). Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modelling and Computer Simulation,* 8:1 (3-30)

[12] Sobol', I. M. (1967). On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Journal of Computational Mathematics and Mathematical Physics,* 16 (1332-1337)