# Variants of Mersenne Twister Suitable for Graphic Processors

MUTSUO SAITO
Hiroshima University

MAKOTO MATSUMOTO
The University of Tokyo

March 21, 2012

**Abstract**

This paper proposes a type of pseudorandom number generator, *Mersenne Twister for Graphic Processor* (MTGP), for efficient generation on graphic processessing units (GPUs). MTGP supports large state sizes such as 11213 bits, and uses the high parallelism of GPUs in computing many steps of the recursion in parallel. The second proposal is a parameter-set generator for MTGP, named *MTGP Dynamic Creator* (MTGPDC). MTGPDC creates up to $2^{32}$ distinct parameter sets which generate sequences with high-dimensional uniformity. This facility is suitable for a large grid of GPUs where each GPU requires separate random number streams.

MTGP is based on linear recursion over the two-element field, and has better high-dimensional equidistribution than the Mersenne Twister pseudorandom number generator.

keywords: Pseudo Random Number Generator, Mersenne Twister, General-Purpose Computing on Graphics Processing Units, Dynamic Creator

## 1 Random number generation for GPU

A Graphic Processing Unit (GPU) is a highly parallel processor designed for computer graphics. GPUs are now widely used in both personal computers and game machines, and are cheap despite their high computational power. As a consequence, a trend in parallel computation is *General-Purpose computing on Graphics Processing Units (GPGPU)* [15], namely, to utilize the high parallelism of GPUs for solving computing problems outside the graphics domain. A number of recent super-computers actually consist of a large grid of GPUs, controlled by one or several CPUs.

Pseudorandom number generators are often necessary in GPGPU, for example, in Monte Carlo simulations, so it is useful to design pseudorandom number generators taking advantage of the parallelism of GPUs.

We propose a class of pseudorandom number generators, *Mersenne Twister for Graphic Processors* (MTGP). The algorithm is similar to that of Mersenne Twister (MT) [17], but refined and adjusted to the hierarchical parallelism of GPUs. The parameter sets for generators are selected by their high-dimensional equidistribution properties. We prepared 128 different parameter sets for each of the periods $2^{11213}-1$, $2^{23209}-1$ and $2^{44497}-1$. The gap between the theoretical upper bound and the realized dimensions of equidistribution of MTGPs is smaller than those of MT, so the MTGPs provide better statistical quality for the same amount of storage.

We also designed a parameter-set generator for MTGP, named MTGP Dynamic Creator (MTGPDC). Analogously to Dynamic Creator [18] for MT, MTGPDC finds good parameter-sets for MTGP with high dimensions of equidistribution. Users specify a set of generator IDs and the desired Mersenne prime period, then the ID is embedded in the recursion parameters of each generator, so that generators with distinct IDs will yield independent streams. This is useful for large grids of GPUs, where each GPU needs one or more independent random number streams, as each stream can be generated from generator recursion parameters with a distinct ID.

The design policies of MTGP and MTGPDC are as follows.

1. Many parallel threads operate on the state space of one large pseudorandom number generator (PRNG). The large state allows a long period and high-dimensional equidistribution properties. This is in contrast to the usual approach to PRNG parallelism, namely, one generator per thread. In the latter case, the increase in the number of threads implies that each generator has a small state space, since the size of fast memory in a GPU is limited.

2. GPUs have a hierarchy in memory: some memory is fast but its access is limited to a group of threads (called a *block*), and some memory is fast but read-only. MTGP takes into account the characteristics of each class of memory for efficient parallel generation.

## 2 GPGPU and CUDA

For GPGPU, a typical hardware setting is as follows. One CPU, called *the host CPU*, is connected to one GPU (often to a grid of GPUs, but for simplicity of explanation we choose the one-GPU case). Consider a computation $C$ which one wants to do in GPGPU. As usual, $C$ is divided into several parts, and some of the parts can be executed in parallel. To use the GPU effectively, one needs to analyze which part of the program can run in parallel, in the many processing units in the GPU. Then, one writes a program for the CPU, called *the host program*, which sends input data and GPU codes (called *the kernel program*) to the GPU for parallel execution. The GPU does the given computation, and returns the output data to the CPU. Usually the output data of GPU computation is a large array, so the data is written in a memory called *global memory* which is accessible from the CPU.

Under this setting, the computer program must be equipped with facilities for scheduling execution of GPU kernels and communication between the CPU and the GPU. One solution is the *Compute Unified Device Architecture* (CUDA) [20],[22], which is a widely used software-development environment for GPGPU on NVIDIA's GPUs. CUDA has a C-like language which supports these facilities. A CUDA program consists of one host program and several kernel programs. Some more concrete explanation is in § 8.

MTGP is written in and executed in the CUDA-environment. Initialization of MTGP is done in the host program (i.e. by the CPU), and the host program calls a kernel program (i.e. invokes the GPU) which generates a specified number of pseudorandom numbers in the global memory (by the GPU). MTGPDC is written in C and executed in the usual CPU (i.e. without GPU).

Another programming environment for GPGPU is the OpenCL [8]. At present, there is no OpenCL version of MTGP/MTGPDC. We use CUDA's terminology, but put comments on OpenCL's corresponding terminology when they differ.

## Hierarchical Structure: software and memory

Since the hardware architecture of a GPU is rather complicated, we explain mainly its software-level hierarchy. When we mention concrete values as examples, we use a middle-range GPU GTX260.

In a kernel program, (namely, a program executed in the GPU), the minimum execution unit is called a *thread* (*work item* in OpenCL), which is one lane in a CUDA execution grid. A *block* (*work group* in OpenCL) consists of many threads, with some upper bound on their number (e.g. 512 under the present CUDA). We may think of these threads in one block as running in parallel. A GPU can run several blocks in parallel.

To realize the high parallelism, threads and blocks have strong constraints in accessing memory and getting instructions. There is a corresponding hierarchy in memory. Each thread has its own set of *registers*, which is inaccessible from other threads. A block has its own *shared memory* (*local memory* in OpenCL) of size 16Kbyte (for most CUDA-enabled GPUs at present), inside the GPU chip. Any thread in one block can access the shared memory of the block, but can not access those of other blocks. A block has also its own read-only cache of the *texture memory* (*image object* in OpenCL), intended for storing texture information in computer graphics.

The tightest restriction is that any thread in a block gets the same instruction sequence. Each thread has its own thread-ID number (consecutive) and can refer to that. Consequently, each thread acts differently according to its ID-number.

Global memory is stored in external memory chips, located outside the GPU chip. Although our ideas on MTGP apply more generally, for the readers who may want to grasp the size and speed of GPU, we give illustrative examples: the size of the memory, in the case of GTX260, is typically 896Mbyte. Data-transfer speed between global memory and the GPU is 112Gbyte/sec, which is faster than typical CPU-memory transfer speeds (eg. 26Gbyte/sec.) Still, the global memory is slower to access than the shared memory inside the GPU. Unlike the shared memory, all threads in the GPU, regardless of which block they belong to, can access the global memory. Different blocks can exchange information only via the global memory. The shared memory is grouped into 16 banks, according to the address (as memory of 32-bit words) modulo 16. The threads in one block are grouped in *warps.* A warp consists of 32 threads in the present CUDA-enabled GPUs. A half warp (16 threads in a warp) may simultaneously access the shared memory, only if each thread accesses to a mutually distinct bank. If two or more threads in a half warp access the same bank (namely, the memory addresses coincide modulo 16), then they can not access in parallel. This phenomenon is called a *bank conflict.*

The cache of the texture memory can be read by many threads simultaneously, differently from the shared memory where bank-conflicts may occur.

# 3 Mersenne Twister for GP (MTGP)

## 3.1 Pseudorandom number generators by recursion

Let $W$ be the set of $w$-bit ($w$: word size) integers, and $x_i \in W$ $(i = 0, 1, 2, \ldots)$ be a sequence of $w$-bit integers.

For generating pseudorandom numbers, it is common to use a recursion: Let $N$ be a positive integer (called the *degree*), $f : W^N \to W$ be a function, and $x_0, x_1, \ldots, x_{N-1}$ be elements of $W$ (called the initial values). Then, the following recursion generates a sequence of elements in $W$:

$$x_{N+i} := f(x_{N-1+i}, x_{N-2+i}, \ldots, x_i) \quad (i = 0, 1, \ldots). \tag{1}$$

For high-speed generation, it is better to choose an $f$ depending only on few variables, but if the variables are too few, the generated sequence tends to show non-randomness. As a trade-off, we consider the following type of recursion:

$$x_{N+i} := f(x_{M+i}, x_{1+i}, x_i) \quad (i = 0, 1, \ldots). \tag{2}$$

We call the positive integer $M$ $(1 < M < N)$ the *middle position*.

Such a recursion can be efficiently computable (under an appropriate choice of $f$) using an array $X[0..L-1]$ of words of length $L$ with $L \geq N$ (see [9, Algorithm A, p.28]), as follows.

1. Store the initial values $x_0, \ldots, x_{N-1}$ to $X[0], X[1], \ldots, X[N-1]$.
2. Set an integer variable $i$ to 0.
3. Set

   $$X[(N+i) \bmod L] \leftarrow f(X[(M+i) \bmod L], X[(1+i) \bmod L], X[i \bmod L]).$$

   This computes $x_{N+i}$.
4. Increment $i \leftarrow i + 1$. If $i \geq L$, then $i \leftarrow i \bmod L$.
5. Return to step 3.

## 3.2 Parallel computation of a single recursion

Suppose that $L$ is sufficiently large, namely $L \geq 2N - M$. Then, one may compute

$$X[N+i] \leftarrow f(X[M+i], X[1+i], X[i])$$

for $0 \leq i \leq N - M - 1$ in parallel. When $i = N - M$, the computation of $X[N+i]$ requires the value of $X[M+i] = X[N]$, which can be obtained only after the $i = 0$-th step has been done, giving the above upper bound $N - M$ for the number of parallel threads.

A basic strategy in our proposed MTGP is to use this parallelism for threads in one block, as pictured in Figure 1. One block works on a single recursion using up to $N - M$ threads in parallel. Suppose that one block has $n$ threads $(n \leq N - M)$. Then, the $i$-th thread $(1 \leq i \leq n)$ works as follows:

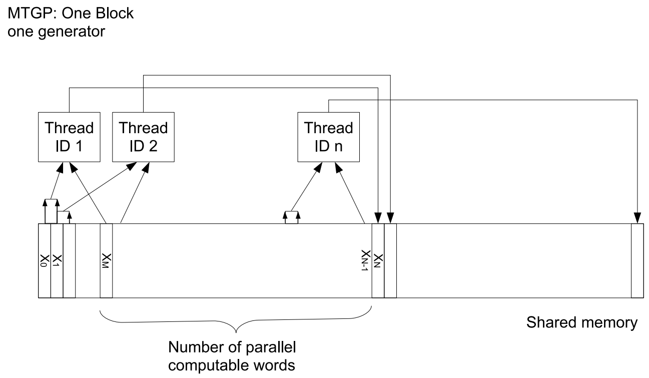


Figure 1: Strategy of MTGP: one block works on a single recursion using up to $N-M$ threads. At Step 1, the threads read from memory via arrows numbered 1 in parallel, then at Step 2 (respectively 3) via arrows numbered 2 (respectively 3). At Step 4, the threads write the generated numbers via arrows numbered 4.

**Step 1** reads $X[i-1]$ $(1 \leq i \leq n)$.

**Step 2** reads $X[i]$ $(1 \leq i \leq n)$.

**Step 3** reads $X[M+i-1]$ $(1 \leq i \leq n)$.

**Step 4** computes $f(X[M+i-1], X[i], X[i-1])$ and writes the result to $X[N+i]$ $(1 \leq i \leq n)$.

This idea of *one block for one generator* is in contrast to a simpler idea: *one thread for one generator*, adopted in CUDA SDK Mersenne Twister sample, which will be explained in §4.1.

## 3.3 $\mathbb{F}_2$-linear generators and Mersenne Twister

We briefly recall the notion of $\mathbb{F}_2$-linear generators, in particular Mersenne Twister (MT) [17] generator, since MTGP is its variant. See [13] for a general theory on $\mathbb{F}_2$-linear generators.

In this article, we identify the set of bits $\{0,1\}$ with the two-element field $\mathbb{F}_2$. A $w$-bit integer is identified with a horizontal vector in $\mathbb{F}_2^w$, and $\oplus$ denotes the sum as vectors (i.e., bit-wise exor). Thus, the set of word-size integers is an $\mathbb{F}_2$-linear space, as well as the set of states of a memory array, etc. We mean by an $\mathbb{F}_2$-linear generator a pseudorandom number generator with $\mathbb{F}_2$-linear vector state space, $\mathbb{F}_2$-linear transition function, and $\mathbb{F}_2$-linear output function.

### Notation

The following notations on bit-operations on words are used. The bitwise-and is denoted by &. Let $r$ be an integer with $1 \leq r \leq w$ and $\mathbf{x}$, $\mathbf{y}$ be two $w$-bit words. The $(w-r)$-bit integer consisting of the $(w-r)$ most significant bits (MSBs) of $\mathbf{x}$ is denoted by $\overline{\mathbf{x}}^{w-r}$. The $r$-bit integer consisting of the $r$ least significant bits (LSBs) of $\mathbf{y}$ is denoted by $\underline{\mathbf{y}}^{r}$. The $w$-bit integer obtained by concatenating

$\overline{\mathbf{x}}^{w-r}$ and $\underline{\mathbf{y}}^{r}$ in this order is denoted by $(\mathbf{x}^{w-r}|\mathbf{y}^{r})$. The bold $\mathbf{0}$ means the zero word. Thus, $(\mathbf{x}^{w-r}|\mathbf{0}^{r})$ is a word whose most significant $w-r$ bits coincide with those of $\mathbf{x}$ and the other $r$ bits being 0. The logical left shift of $\mathbf{x}$ by $r$ bits is denoted by $(\mathbf{x} \ll r)$, and the right shift by $(\mathbf{x} \gg r)$. A hexadecimal number is denoted with `0x` at the head, so for example `0xf` denotes the integer 15 whose binary representation is 1111. Matrices are denoted by bold letters such as $\mathbf{A}$ and $\mathbf{R}$, and multiplication to a row vector $\mathbf{x}$ is denoted by $\mathbf{xA}$, and every component is computed modulo 2, i.e., in $\mathbb{F}_2$.

Choose a Mersenne prime, i.e., a prime number of the form of $2^p - 1$; the integer $p$ is called a Mersenne exponent (MEXP). A basic strategy of MT is to realize the Mersenne prime period. For this purpose, put $N = \lceil p/w \rceil$, $r = wN - p$. Thus, $N$ is the least length of array of $w$-bit integers that accommodates $p$ bits. MT generates a sequence of elements in $\mathbb{F}_2^w$ by a recursion

$$\begin{aligned} \mathbf{x}_{N+i} &= f(\mathbf{x}_{M+i}, \mathbf{x}_{1+i}, \overline{\mathbf{x}}_i^{w-r}) && (3) \\ &= \mathbf{x}_{M+i} \oplus (\mathbf{x}_i^{w-r}|\mathbf{x}_{1+i}^{r})\mathbf{A}, && (4) \end{aligned}$$

where $\mathbf{A}$ is a $(w \times w)$-matrix such that $\mathbf{xA}$ is computable by

$$\mathbf{xA} = \begin{cases} (\mathbf{x} \gg 1) & (\text{if } \underline{\mathbf{x}}^1 = 0) \\ (\mathbf{x} \gg 1) \oplus \mathbf{a} & (\text{if } \underline{\mathbf{x}}^1 = 1), \end{cases} \qquad (5)$$

where $\mathbf{a}$ is a constant $w$-dimensional row vector.

The function

$$(\mathbf{x}_{N+i-1}, \mathbf{x}_{N+i-2}, \ldots, \mathbf{x}_{i+1}, \overline{\mathbf{x}}_i^{w-r}) \mapsto (\mathbf{x}_{N+i}, \mathbf{x}_{N+i-1}, \ldots, \mathbf{x}_{i+2}, \overline{\mathbf{x}}_{i+1}^{w-r})$$

is a fixed $\mathbb{F}_2$-linear transformation $F$, and the recursion is simply the iteration of $F$. Thus, Mersenne Twister is considered as an automaton with state transition function $F : \mathbb{F}_2^p \to \mathbb{F}_2^p$, where $p = wN - r$. The period of $F$ attains the maximum $2^p - 1$ if and only if the characteristic polynomial of $F$ is primitive — there is an efficient algorithm to check this when $p$ is a Mersenne exponent [17].

### 3.4 $k$-distribution and tempering

The quality of a pseudorandom number generator is often measured by the period, together with its high-dimensional equidistribution property, defined below (cf. [2][12]).

**Definition 3.1** *A sequence of $v$-bit integers with period $P = 2^p - 1$*

$$\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{P-1}, \mathbf{x}_P = \mathbf{x}_0, \ldots$$

*is said to be* $k$-dimensionally equidistributed *if the consecutive $k$-tuples*

$$(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+k-1}), \quad i = 0, \ldots, P-1$$

*are uniformly distributed over all possible $kv$-bit patterns except the all-0 pattern, which occurs once less often, i.e., each distinct $k$-tuple of $v$-bit words appears the same number of times in the sequence (with one exception of $k$-tuple of zeroes).*

**Definition 3.2** *A periodic sequence of $w$-bit integers is* $k$-dimensionally equidistributed to $v$-bit accuracy *if the most significant $v$-bit integer sequence is $k$-dimensionally equidistributed.*

*The* dimension of equidistribution to $v$-bit accuracy $k(v)$ *is the maximum value of $k$ such that the sequence is $k$-dimensionally equidistributed to $v$-bit accuracy. Larger values of $k(v)$ show higher dimensional equidistribution for $v$-bit accuracy.*

*For $P = 2^p - 1$, there is a bound $k(v) \le \lfloor p/v \rfloor$. The* dimension defect $d(v)$ *at $v$ is the difference* $d(v) := \lfloor p/v \rfloor - k(v) \ge 0$, the total dimension defect $\Delta$ *is their sum over $v$: $\Delta := \sum_{v=1}^{w} d(v)$. Smaller $\Delta$ value shows that the generator is closer to the optimum from the view point of $k(v)$ $(v = 1, \ldots, w)$.*

A possible alternative to $\Delta$ is *the maximum dimension defect* $\max\{d(v) | v = 1, 2, \ldots, w\}$. In this article we use only $\Delta$.

To obtain a better equidistribution property, MT chooses a $(w \times w)$ matrix $\mathbf{T}$ (called the tempering matrix), and outputs $\mathbf{x}_i\mathbf{T}, \mathbf{x}_{i+1}\mathbf{T}, \mathbf{x}_{i+2}\mathbf{T}, \ldots$. The tempering matrix $\mathbf{T}$ is chosen so that each $k(v)$ (in particular for small $v$) is close to its upper bound mentioned above; namely, so that the dimension defect $d(v)$ and their sum $\Delta$ is small.

## 3.5 Design of MTGP : using threads

In designing MTGP, we utilize the parallelism explained in §3.1, namely $N - M$ threads can work in parallel on the state space. We chose the number of threads for degree-$N$ MTGP generators as the largest power of 2 no more than $N - 2$ (here $M \ge 2$ implies $N - M \le N - 2$), which we denote by $\lfloor N - 2 \rfloor_2 = 2^{\lfloor \log_2(N-2) \rfloor}$. We choose a small $M$ (but $M \ge 2$), so that the gap $N - M \ge \lfloor N - 2 \rfloor_2$ holds. We released MTGP for three MEXPs $p = 11213, 23209, 44497$. These choices correspond to the number of threads being 256, 512, and 1024, respectively.

## 3.6 Description of MTGP:recursion

Let us fix an MEXP $p$. Similarly to MT (see §3.3), we compute $N = \lceil p/w \rceil$ and $r = wN - p$.

MTGP generates a sequence of elements in $\mathbb{F}_2^w$ by a recursion

$$\mathbf{x}_{N+i} \quad := \quad g(\mathbf{x}_{M+i}, \mathbf{x}_{1+i}, \overline{\mathbf{x}}_i^{w-r}), \tag{6}$$

where $M$ is an integer satisfying the condition in the previous section. The $\mathbb{F}_2$-linear recurring function $g$ is determined by six integer parameters $N, M, w, r, sh1, sh2$ and a $(4 \times w)$ matrix $\mathbf{R}$.

**Definition 3.3** *The MTGP recursion (6) is defined as follows, using temporary variables $\mathbf{t}$ and $\mathbf{u}$ of $w$-bit integer (identified with row vectors in $\mathbb{F}_2^w$):*

$$\begin{aligned}
\mathbf{t} &\leftarrow \mathbf{x}_{1+i} \oplus (\mathbf{x}_i^{w-r} | \mathbf{0}^r) \\
\mathbf{t} &\leftarrow \mathbf{t} \oplus (\mathbf{t} \ll sh1) \\
\mathbf{u} &\leftarrow \mathbf{t} \oplus (\mathbf{x}_{M+i} \gg sh2) \\
\mathbf{x}_{N+i} &\leftarrow \mathbf{u} \oplus (\underline{\mathbf{u}}^4 \mathbf{R}),
\end{aligned} \tag{7}$$

*with the computation in this order. In the last line, $\underline{\mathbf{u}}^4$ denotes the four-dimensional row vector over $\mathbb{F}_2$ consisting of the four LSBs of $\mathbf{u}$, and hence the multiplication $\underline{\mathbf{u}}^4\mathbf{R}$ with $(4 \times w)$-matrix $\mathbf{R}$ yields a $w$-dimensional vector (see notations in §3.3).*

*The parameters $(N, M, w, r, sh1, sh2, \mathbf{R})$ are called the* recursion parameters *of the MTGP, and chosen to realize the period $2^{wN-r} - 1$.*

For the speed, the multiplication of $\mathbf{R}$ is implemented as a look-up table. Since $\underline{\mathbf{u}}^4$ may assume only 16 values $0000, 0001, \ldots, 1111$ in the binary form, we may precompute 16 $w$-dimensional vectors $(0000)\mathbf{R}$, $(0001)\mathbf{R}$, ..., $(1111)\mathbf{R}$ and store them in an array of words, say, in `rectbl[0..15]`. Then, $\underline{\mathbf{u}}^4\mathbf{R}$ is `rectbl`[$\underline{\mathbf{u}}^4$].

An equivalent description to the recursion (7) in C-language is as follows. We assume that the content of the array `rectbl` is precomputed as above. Note that the symbol ∧ denotes the bitwise exor in C.

$$\begin{aligned} \texttt{t} &= \texttt{X[1+i]} \wedge (\texttt{X[i]}\ \&\ \texttt{BITMASK(w,r)}); \\ \texttt{t} &= \texttt{t} \wedge (\texttt{t<<sh1}); \\ \texttt{u} &= \texttt{t} \wedge (\texttt{X[M+i]>>sh2}); \\ \texttt{X[N+i]} &= \texttt{u} \wedge \texttt{rectbl}[\texttt{u}\ \&\ \texttt{0xf}]; \end{aligned} \tag{8}$$

Here, `BITMASK(w,r)` is the bitmask of a $w$-bit integer with $(w - r)$ MSBs being 1 and $r$ LSBs being 0.

The cache of the texture memory of each block is suitable for such a look-up table, because it is free from bank-conflicts. Note that because the values of $\underline{\mathbf{u}}^4$ in distinct threads (even in a warp) may often coincide, the execution of (8) may often cause a bank-conflict if the array `rectbl` is kept in the shared memory. If we keep them in registers, the number of available registers for each thread decreases, hence the number of concurrently executable threads decreases since there is a limitation in the total number of registers in one block.

## 3.7 Description of MTGP : tempering

Analogous to MT, MTGP transforms the sequence $\mathbf{x}_i$ by a fixed linear transformation to obtain better $k(v)$ (called *tempering*): the $(N + i)$-th output $\mathbf{o}_{N+i}$ is given by

$$\mathbf{o}_{N+i} := \mathbf{x}_{N+i} \oplus h(\mathbf{x}_{M-1+i}) \tag{9}$$

for a suitably chosen linear transformation $h$ so that the output sequence has a high value of $k(v)$ $(v = 1, \ldots, w)$.

**Definition 3.4** *The tempering (9) of MTGP is defined as follows, using a temporary variable $\mathbf{t}$ of $w$-bit integer:*

$$\begin{aligned} \mathbf{t} &\leftarrow \mathbf{x}_{M-1+i} \oplus (\mathbf{x}_{M-1+i} \gg 16) \\ \mathbf{t} &\leftarrow \mathbf{t} \oplus (\mathbf{t} \gg 8) \\ \mathbf{o}_{N+i} &\leftarrow \mathbf{x}_{N+i} \oplus \underline{\mathbf{t}}^4\mathbf{T} \end{aligned} \tag{10}$$

*with the computation in this order. Here, $\mathbf{T}$ is a $(4 \times w)$-matrix over $\mathbb{F}_2$. The defining parameter is $\mathbf{T}$, called the* tempering parameter. *The linear transformation $h(\mathbf{x}_{M-1+i})$ appearing in (9) is defined as $\underline{\mathbf{t}}^4\mathbf{T}$ appearing in (10).*

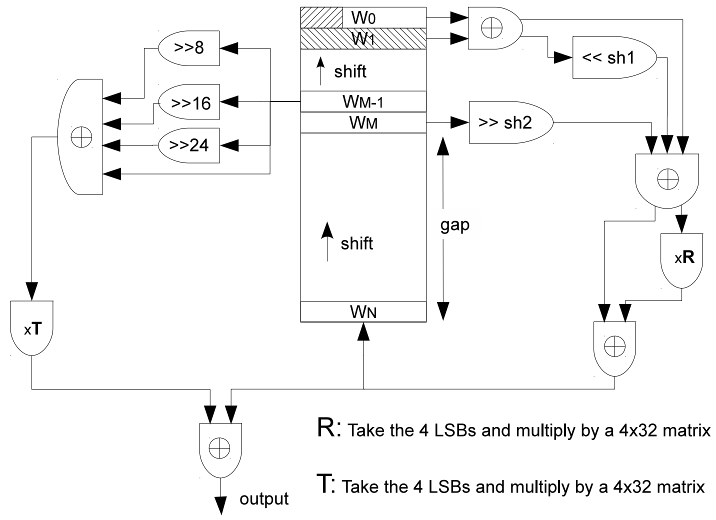

W0
W1
shift
WM-1
WM
shift
WN
gap
>>8
>>16
>>24
xT
<< sh1
>> sh2
xR
output
R: Take the 4 LSBs and multiply by a 4x32 matrix
T: Take the 4 LSBs and multiply by a 4x32 matrix

Figure 2: Circuit-like description of MTGP: the right circuit is for recursion, the left circuit for the tempering.

This type of tempering using another word $\mathbf{x}_{M-1+i}$ in the state array came from [5]. Use of the multiplication by a $(4 \times w)$-matrix was obtained through trial-and-error to obtain a fast tempering with a large value of $k(v)$ $(1 \le v \le 32)$.

An equivalent description to the tempering (10) in C-language is as follows:

$$\begin{aligned} \texttt{t} &= \texttt{X[M-1+i]} \wedge (\texttt{X[M-1+i] >> 16}); \\ \texttt{t} &= \texttt{t} \wedge (\texttt{t >> 8}); \\ \texttt{return}\ \ &(\texttt{X[N+i]} \wedge \texttt{tmptbl}[\texttt{t \& 0xf}]); \end{aligned} \tag{11}$$

Similarly to the look-up table `rectbl` in the recursion (8), `tmptbl` is an array of length 16 storing the values $(0000)\mathbf{T}$, $(0001)\mathbf{T}$, ..., $(1111)\mathbf{T}$.

Altogether, one MTGP has the recursion parameters as in Definition 3.3 and the tempering parameter as in Definition 3.4. A circuit-like description of MTGP is shown in Figure 2.

## 3.8 Dimensions of equidistribution of MTGP

MTGP supports three different periods $2^{11213}-1$, $2^{23209}-1$ and $2^{44497}-1$. For each period, we searched for 128 different parameter sets with good equidistribution properties, sorted in terms of the total defect $\Delta$. So users can generate 128 distinct streams by MTGPs of different parameter sets.

Table 1 lists $k(v)$ and $d(v)$ of the MTGP23209 of period $2^{23209}-1$ for the first parameter set among the obtained 128 sets. In addition, the defect ratio $100 \times d(v)/\lfloor p/v \rfloor$ in % is listed. For $v = 1, \ldots, 16$, $d(v)$ is 0 or 1, which means that the equidistribution up to 16-bit accuracy is close to the optimal. For comparison, Mersenne Twister MT19937 has $d(1) = d(2) = d(4) = d(8) = 0$, but $d(3) = 405, d(5) = 249, d(6) = 207$ and $d(7) = 355$ for $1 \le v \le 8$. The

Table 1: $k(v)$, $d(v)$ and the defect ratio $r(v) := 100 \times d(v)/(k(v) + d(v))$ (in %) of MTGP23209 with first parameter set

| $v$ | $k(v)$ | $d(v)$ | $r(v)\%$ | $v$ | $k(v)$ | $d(v)$ | $r(v)\%$ |
|---|---|---|---|---|---|---|---|
| 1 | 23209 | 0 | 0 | 17 | 1362 | 3 | 0.22 |
| 2 | 11604 | 0 | 0 | 18 | 1266 | 23 | 1.78 |
| 3 | 7736 | 0 | 0 | 19 | 1181 | 40 | 3.28 |
| 4 | 5802 | 0 | 0 | 20 | 1137 | 23 | 1.98 |
| 5 | 4641 | 0 | 0 | 21 | 1043 | 62 | 5.61 |
| 6 | 3868 | 0 | 0 | 22 | 931 | 123 | 11.67 |
| 7 | 3315 | 0 | 0 | 23 | 930 | 79 | 7.83 |
| 8 | 2900 | 1 | 0.03 | 24 | 930 | 37 | 3.83 |
| 9 | 2578 | 0 | 0 | 25 | 727 | 201 | 21.66 |
| 10 | 2320 | 0 | 0 | 26 | 726 | 166 | 18.61 |
| 11 | 2109 | 0 | 0 | 27 | 725 | 134 | 15.60 |
| 12 | 1934 | 0 | 0 | 28 | 725 | 103 | 12.44 |
| 13 | 1785 | 0 | 0 | 29 | 725 | 75 | 9.38 |
| 14 | 1657 | 0 | 0 | 30 | 725 | 48 | 6.21 |
| 15 | 1547 | 0 | 0 | 31 | 725 | 23 | 3.07 |
| 16 | 1450 | 0 | 0 | 32 | 725 | 0 | 0 |

Total dimension defect $\Delta$ is 1141.

total dimension defect $\Delta = 1141$ of MTGP23209 is better than $\Delta = 6750$ of MT19937.

The WELL [23] generator has optimal $\Delta = 0$, but it seems difficult to run WELL on GPUs efficiently because of the heavy dependencies among the partial computations in the recursion.

The last (and worst) among the 128 parameter sets for MTGP23209 has $\Delta = 2100$, still much better than MT19937.

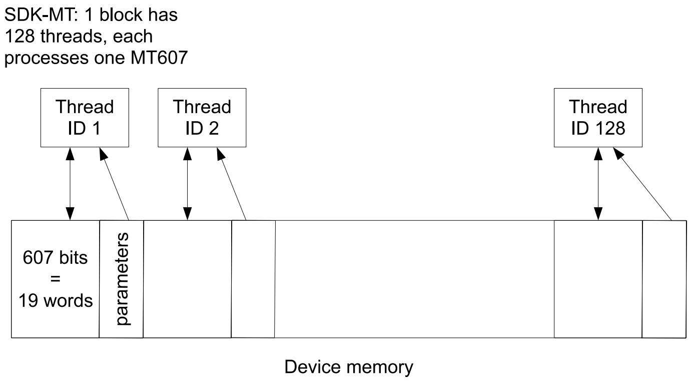
# 4 Comparison with other generators on GPU

## 4.1 SDK-MT: a naive approach

As mentioned earlier in §3.1, a more naive idea for PRNG for GPU is to use one PRNG for one thread, unlike for one block as in MTGP. An example is SDK Mersenne-Twister (SDK-MT) in CUDA-SDK [21] sample programs from NVIDIA, which is a set of 4096 (small) Mersenne Twister implementations on CUDA. SDK-MT uses 32 blocks, each block consists of 128 threads, and each thread runs a Mersenne Twister with 607-bit state space (MT607). Each of the $32 \times 128 = 4096$ threads uses a distinct set of parameters, produced by Dynamic Creator for MT [18] to support the independence of those streams.

Figure 3 pictures one block (128 threads) for SDK-MT. These parameters (e.g., the coefficients in the recursion) are kept in the global memory.

Figure 3: SDK-MT: PRNGs for GPU, one thread for one generator.



## 4.2 Merits of MTGP over SDK-MT

The parallelization in MTGP, namely using the parallelism naturally appearing in the recursion in the array, has been known since the 1980s (see for example [1, §5.2.1]). Its merits compared to SDK-MT are:

- SDK-MT's consumption of memory counted in bits is (607+parameter size)× the number of threads.

- MTGP's consumption is 32× the number of threads.

- As an illustrative example, assume that the size of the shared memory is 16Kbyte and that the state spaces of SDK-MT are allocated in the shared memory. Then, the number of parallel threads is small: namely (16Kbyte)/(size of working space for MT607)<400, losing its gain in parallelism.

- The period of generated sequence: SDK-MT has period $2^{607} - 1$, while MTGP has period $2^{11213} - 1$ and higher dimensional equidistribution property.

## 4.3 Other GPU-based PRNGs

We compare MTGP with some other generators implemented for GPUs. We do not treat classical Linear Congruential Generators with modulus less than or equal to $2^{32}$, such as the Park-Miller generator implemented for GPUs [11], since they are not recommendable for massive use (its period is $\leq 2^{32}$, and the sequence is rejected by several simple statistical tests called SmallCrush, see [14] and [28]). We consider the following generators.

- The MTGP generator with period of $2^{11213} - 1$ described above. We use 108 blocks to run 108 MTGPs whose characteristic polynomials are mutually distinct.

- The SDK-MT generator with period of $2^{607} - 1$ described in Section 4.1.

Table 2: Consumed time for $5 \times 10^7$ numbers by five PRNGs, their periods, and the results of BigCrush statistical tests: "pass*" means passed all the tests, "pass" means passed the tests except those on $\mathbb{F}_2$-linearity, "reject" means rejected systematically by at least one of the tests not based on $\mathbb{F}_2$-linearity.

| | SDK-MT | MTGP11213 | | | HybridTaus | Warp | CURAND |
|---|---|---|---|---|---|---|---|
| | float[0,1) | 32bit int | float[1,2) | float[0,1) | float[0,1) | float[0,1) | float[0,1) |
| GT 120 | 50.2ms | 38.2ms | 38.9ms | 39.8ms | 35.4 ms | 13.8ms | 16.5ms |
| GTX 260 | 18.6ms | 4.7ms | 4.8ms | 5.0ms | 9.3ms | 3.2ms | 3.0ms |
| Period | $2^{607} - 1$ | $2^{11213} - 1$ | | | $\sim 2^{121}$ | $2^{1024} - 1$ | $\sim 2^{192}$ |
| BigCrush | reject | pass | | | pass | pass* | reject |

- The HybridTaus generator [10, Example 37-4] in GPU Gems, with period of around $2^{121}$. This generator outputs the exor of two 32-bit integer streams, one generated by a combined Tausworthe generator taus88 [12] and the other by a linear congruential generator "Quick and Dirty."

- Warp Generator [28]. This generator is based on a sparse $\mathbb{F}_2$-linear transition matrix $M : \mathbb{F}_2^{32\times 32} \to \mathbb{F}_2^{32\times 32}$. Its period is $2^{1024} - 1$. The space $\mathbb{F}_2^{32\times 32}$ is identified with an array of 32-bit integers of length 32, so that a warp computes the multiplication of $M$. The content of the array is considered as 32 of 32-bit random integers. The raw output (WarpCorrelated) does not pass the BigCrush test suite, but by combining with a non $\mathbb{F}_2$-linear output function (sum of two 32-bit integers modulo $2^{32}$), the output (WarpStandard) passes the BigCrush test suite. In this article, the Warp Generator means WarpStandard.

- The CURAND generator [19]. This is the newest in the NVIDIA SDK. This is an implementation of the xorwow generator, which is the combination of an $\mathbb{F}_2$-linear generator xorshift and a Weyl generator (i.e., a linear congruential generator with multiplier 1) by addition modulo $2^{32}$, proposed in [16]. Its period is $2^{192} - 2^{32}$.

## 4.4 Generation Speed

### Comparison of speed

We measured the consumed time in milliseconds, for generating $5 \times 10^7$ single floating-point numbers in the range $[0, 1)$ for MTGP11213, SDK-MT, HybridTaus, WarpStandard, and CURAND, on the GeForce GT120 GPU (4 cores) and the GeForce GTX260 GPU (27 cores). For MTGP11213, we also measured the time for generating unsigned integers and floating-point numbers in $[1, 2)$ (see §6 for the reason). Table 2 shows the results. On both GT120 and GTX260, SDK-MT is the slowest, and WarpStandard and CURAND are faster than MTGP. On GT120, CURAND is 2.4 times faster than MTGP, while on GTX260 the factor decreases to 1.7.

Since CPU's and GPU's have inherently different purposes, there may be no fair way to compare their performance, but for a reference, we show that MTGP is faster than a PRNG on high-spec CPUs. SFMT [25], which is one of the fastest generators on SIMD machines, takes 25ms to generate the same $5\times 10^7$ number of pseudorandom 32-bit integers on an Intel Xeon X5570 2.93GHz 4 core 2 CPU using 1 process (i.e. one core in one CPU), while MTGP takes 4.6ms as shown in the table. We also tried to use 4 cores of Xeon, but it turns out to be slower than using one core. This seems due to the overhead time for thread generation and synchronization (we use pthread in POSIX to generate four threads in the Xeon).

We mention that there are studies on hardware implementations of Mersenne Twisters. For example, [29] implemented MT19937 on both a FPGA hardware and on a GPU. They used the 8800 GTX GPU which is a little slower than the GTX260. Their implementation in the GPU takes 16.9ms to generate the same number of 32-bit integers as above, hence slower than MTGP on the GTX260. On the other hand, their FPGA implementation is 35 times faster than their GPU implementation and hence is much faster than MTGP on GTX260.

## 4.5 Statistical tests

We conducted TestU01 statistical test suits [14] to the MTGPs and the four other generators in the previous section.

The Crush library in TestU01 contains 96 tests and the BigCrush library contains 106 tests. See the user's guide for details on the tests. The results of the tests show that MTGP, HybridTaus and Warp are not rejected, but SDK-MT and CURAND are systematically rejected, as explained below.

**MTGP** Among 160 tests in the BigCrush library, four tests based on $\mathbb{F}_2$-linearity reject MTGPs. Two LinearComp tests (test number 80 and 81), which measure linear complexity of the sequence, reject all the MTGPs. Two MatrixRank tests (number 70 and 71), which measure the $\mathbb{F}_2$-linear rank of $5000\times 5000$ matrices generated from the output bit stream, reject the MTGPs with a state size less than 5000 bits. This rejection is common among $\mathbb{F}_2$-linear generators such as Mersenne Twister (see [14]) and WELL generators [23]. Because these statistics are based on $\mathbb{F}_2$-linear relations of each bit of the outputs, the observed deviation would hardly cause problems when the sequence is used as real numbers or integers in a simulation. Other tests in the BigCrush library show no systematic deviation of MTGPs (including those generated by MTGPDC explained in the next section).

**HybridTaus** Similarly to MTGPs, HybridTaus passed the tests except MatrixRank tests and LinearComp tests in the BigCrush.

**SDK-MT** SDK-MT is rejected by RandomWalk tests (number 76) in addition to the above mentioned $\mathbb{F}_2$-linearity tests in the BigCrush. It passes the other tests.

**CURAND** CURAND is systematically rejected by three tests in the BigCrush: CollisionOver (number 7), SimpPoker (number 27) and LinearComp (number 81). The $p$-values are outside the interval $[10^{-15}, 1-10^{-15}]$. This phenomenon has been reported by [3]. Since the first test is on the random numbers in the interval $[0,1)$, the observed deviation may cause some erroneous results in a serious simulation.

The defect of CURAND becomes more serious when we examine the differences between the successive outputs: namely, from the output sequence $x_n \in [0,1)$, we make a new sequence $y_n := x_n - x_{n-1} \mod 1 \in [0,1)$. Among the BigCrush battery, three CollisionOver tests (number 7, 8, 10), one Gap test (number 36), one RandomWalk test (number 75), and one Run of Bits test (number 102) show $p$-values outside $[10^{-15}, 1-10^{-15}]$ for $y_n$.

Even among the Crush battery, two CollisionOver tests (number 7, 8), one ClosePairsNJumps test (number 20), and one Gap test (number 32) show $p$-values outside $[10^{-3}, 1-10^{-3}]$ systematically.

**Warp** WarpStandard generator passes all the tests in the BigCrush library.

# 5 MTGPDC

Dynamic Creation of pseudorandom number generators [18] is proposed for large scale parallel simulations in which a number of PRNGs with distinct parameters are desired. Dynamic Creator for Mersenne Twister (MTDC) is released online: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`.

MTDC receives a 16-bit integer called *ID*, and generates a parameter set for a Mersenne twister with specified (Mersenne prime) period, with the ID embedded in the recursion, so that distinct ID assures a distinct (hence relatively prime) irreducible characteristic polynomial of the transition function of the generator.

One problem of MTDC is that there are some IDs where MTDC can not find a parameter set with the specified period. This phenomenon is observed only when $w = 31$. The ID is embedded as the 16 MSBs of the parameter $\mathbf{a}$ of MT (see §3.3), and MTDC searches for the remaining bits of $\mathbf{a}$ that attain the maximal period, but sometimes no such bit-pattern exists. Another problem is that the 16-bit space for the ID is somewhat narrow for the recent trend of high parallelism.

Our proposed MTGPDC receives a 32-bit integer as an ID, which is embedded in the $(4 \times 32)$ matrix parameter $\mathbf{R}$ in Definition 3.3. Although there is no assurance that distinct IDs give distinct characteristic polynomials, there is a heuristic argument that with high probability they would do so, since the 32 bits are mapped to $p$ (say, 11213) bits of the coefficients of the characteristic polynomial in a complicated way. For the sake of completeness, MTGPDC calculates the SHA1 digest of the coefficients of the characteristic polynomial, so that users can check whether the characteristic polynomials are different (by checking that the SHA1 digests are different) if they want.

As pointed out by a referee, there is no theoretical assurance for the independence of the generated streams even if the characteristic polynomials are distinct. However as far as we know, there are no reports on the deviation of the correlations among two such $\mathbb{F}_2$-linear generators. We also tested a merged sequence from the outputs of two parameter sets of MTGPs of the same period using the BigCrush library, but no deviation is observed.

As explained in Definitions 3.3 and 3.4, MTGP has two kinds of parameter sets, namely recursion parameters and tempering parameters. Recursion parameters decide the recursion and hence the period of the generator. The

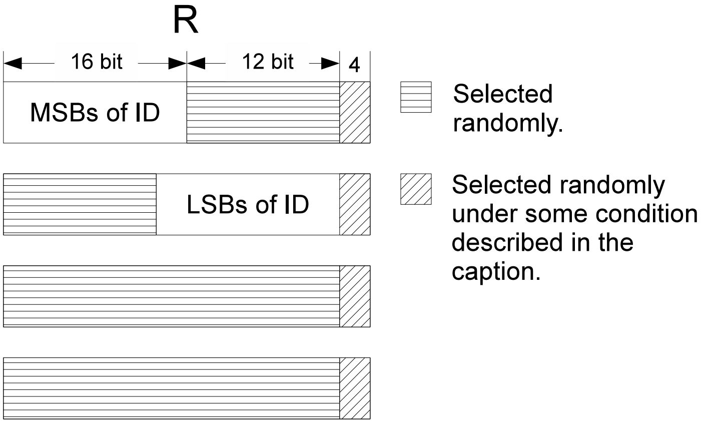

R
16 bit
12 bit
4
MSBs of ID
LSBs of ID
Selected randomly.
Selected randomly under some condition described in the caption.

Figure 4: A recursion parameter $R$. For the efficiency of the search, the four sets of 4 LSBs in $R$ are chosen so that the corresponding $4 \times 4$ matrix does not have an eigenvalue 1, which is equivalent to the invertibility of the transformation on $\mathbf{u}$ in the last line of (7). This is a necessary condition to have the maximal period.

tempering parameter $\mathbf{T}$ changes the output function and decide the dimensions $k(v)$ of equidistribution.

MTGPDC first searches for the recursion parameter set $(N, M, w, r, sh1, sh2, \mathbf{R})$ in Definition 3.3, and then the tempering parameter set (Definition 3.4). Once the Mersenne prime period $2^p - 1$ and the wordsize $w$ are fixed, then $N = \lceil p/w \rceil$ and $r = Nw - p$ are determined. MTGPDC embeds the received ID in the matrix $\mathbf{R}$, as follows.

Figure 4 describes the recursion parameter $\mathbf{R}$, which consists of four 32-dimensional row vectors over $\mathbb{F}_2$. These vectors are identified with 32-bit integers, with the MSB at the left-most component of the vector. The given 32-bit ID is separated into the 16 MSBs and the 16 LSBs. The former are embedded in the 16 MSBs of the first row. The latter are embedded in the 16 bits of the second row consisting of the 12th, 13th, ..., and 28th MSB. 4 LSBs of each row are selected randomly so that the corresponding $(4 \times 4)$ matrix plus the identity matrix is invertible (this is a necessary condition for the last line in (7) being invertible, which is necessary for the irreducibility of the characteristic polynomial). Other parts of matrix $R$ are randomly selected.

Shift parameter `sh1` and `sh2` are fixed to 13 and 4 respectively, and the middle position $M$ is randomly selected so that $2 < M < N - \lfloor N \rfloor_2$. Experimentally, we found that some shift parameters gave rather large defect $\Delta$ even after tempering, so we decided to fix these shift parameters which give good $\Delta$ empirically. For each selection of a parameter set, the characteristic polynomial is calculated using Berlekamp-Massey algorithm implemented in Number Theory Library [27]. If the polynomial is reducible, the next possible parameter set (with ID still embedded in $\mathbf{R}$) is randomly generated, until the irreducible polynomial is found. Since the degree of the irreducible polynomial is a Mersenne exponent, the polynomial is primitive and the maximal period $2^p - 1$ is attained.

Table 3: Time (sec) for recursion and tempering parameter search

| | $p$ | 3217 | 4423 | 11213 | 23209 | 44497 |
|---|---|---|---|---|---|---|
| | samples | 3000 | 3000 | 1500 | 1500 | 750 |
| recursion | min | 0 | 0 | 4 | 24 | 143 |
| | max | 90 | 191 | 3318 | 10146 | 49987 |
| | average | 11.2 | 25.0 | 338.1 | 1404.7 | 6529.4 |
| tempering | min | 10 | 15 | 76 | 379 | 946 |
| | max | 25 | 40 | 253 | 1040 | 3893 |
| | average | 21.7 | 34.1 | 213.7 | 910.0 | 3236.4 |

According to our experiments, probably due to the large degree of freedom in **R**, MTGPDC finds a parameter set for any given ID, unlike the case of MTDC which fails to find one for some ID when $w = 31$.

Once the recursion parameter set with maximal period is found, MTGPDC searches for a tempering parameter set. The tempering parameter is a $4 \times 32$ matrix (realized by a look-up table `tmptbl`) which consists of four 32-bit vectors (see (10)). MTGPDC searches for tempering parameters as follows:

```
• prepare an array tmp[0..3] of 32-bit integers.
• set the four components of tmp[0..3] to 0.
• search for the 23 MSBs of the parameters
for $i = 0$ to 3 do
  for $j = 0$ to 20 step 5 do
    • $e = \min(j + 5, 23)$
    • generate all bit patterns of $j$-th to $e$-th bits of tmp[$i$].
    • calculate $k(1), k(2), \dots k(e)$ for each bit pattern.
    • fix the bit pattern that attains the least sum of the dimension defects
    $d(1) + d(2) + \cdots + d(e)$.
  end for
end for
• modify the 9 LSBs of the parameters found above according to the distribution of 9 LSBs
for $i = 0$ to 3 do
  for $j = 0$ to 9 step 5 do
    • $e = \min(j + 5, 9)$
    • generate all bit patterns of 31-$j$-th to 31-$e$-th bits of tmp[$i$]. This does
    not change the 23 MSBs of the parameters.
    • calculate $k'(1)$ to $k'(e)$, where $k'(v)$ means the dimension of equidistribution for the $v$ LSBs.
    • fix the bit pattern that attains the least sum of the dimension defects
    $d'(1) + d'(2) + \cdots + d'(e)$, where $d'(v)$ means the dimension defect for the
    $v$ LSBs.
  end for
end for
```

Computing the dimension of equidistribution is a time-consuming part in MTGPDC. We used the SIS algorithm [7] to compute these dimensions, which reduces the computation time considerably. (Note that a faster method [6] has recently been developed.)

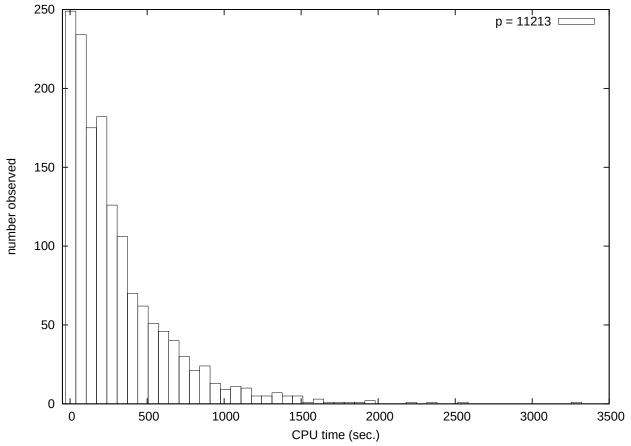


Figure 5: Distribution of time (sec) for recursion parameter search ($p = 11213$)

Figure 5 shows the distribution of the CPU time for recursion-parameter searches for $p = 11213$. The minimum and maximum $\Delta$ among 1500 parameter sets are 565 and 3542, respectively. Table 3 shows the minimum, maximum and average time (sec.) to find one MTGP-parameter set for various $p$. The upper three rows show CPU times to find recursion-parameter sets and the lower three rows show CPU times to find tempering-parameter sets. Times in Figure 5 and Table 3 are measured on an Intel Xeon 5500 2.26GHz 4 core × 2 CPU, running 15 MTGPDC processes simultaneously. These timings show that MTGPDC is too slow to use during a simulation. MTGPDC should be used to create a sufficient number of parameter sets before the simulation, and these sets should be reused.

MTGPDC records a set of recursion and tempering parameters in comma-separated values format in a file, together with the following additional data: SHA1 digest of the characteristic polynomial, number of non-zero terms of the characteristic polynomial, and the total dimension defect.

## Detectable deviations in LSBs in an older version of MTGPDC

[24] reported the following problem on the former version of MTGPDC. In MTGPDC version 0.2 (released on June 8th 2009) for $w = 32$, several (say seven) LSBs of 32-bit outputs are often rejected by some tests in the BigCrush library for some parameter sets. Among these tests, two notable tests are `sknuth_Gap` test for 5-bit sequences consisting of the 26th, 27th, 28th, 29th, and 30th bit of 32-bit outputs (Test 35) and `sstring_HammingIndep` test for the same bits (Test 100). Test 35 rejects 1/5 of the 10000 created parameter sets, while Test 100 rejects 1/10.

This older MTGPDC has only 23 MSBs for the tempering parameters. After this report, the remaining 9 LSBs in the tempering parameters are supplemented, as described in this section.

# 6 Floating point generation in IEEE745

If one generates random integer sequences and transforms them into uniform real numbers by multiplication or division, the conversion time is not negligible. For current NVIDIA CUDA-enabled GPUs, such conversion is not heavy (equivalent to 5 additions), but for present-day AMD GPUs, the integer-floating conversion is rather time-consuming.

These days, most computers use IEEE 754 format for both single and double precision floating-point numbers. Generating floating points in this format is faster than conversion, see [26].

The same idea is used in MTGP. IEEE 754 single-precision format uses 32 bits for representing a real number. It consists of a 1-bit sign part (the MSB), a 8-bit exponent part and a 23-bit fraction part. To generate single precision floating point numbers, the last line of the tempering (10) is changed to:

$$\mathbf{o}_i = (\mathbf{x}_i \gg 9) \wedge \texttt{sngltbl}[\texttt{t} \ \& \ \texttt{0x0f}]; \tag{C}$$

where `sngltbl` is essentially the same as `tmptbl`, just formatted to produce single floating point format in IEEE754, as explained below. Every component `sngltbl[i]` ($0 \leq i \leq 15$) has nine MSBs which are equal to 001111111, and its 27 LSBs are the 27 MSBs of `tmptbl[i]`. The above (C) amounts to producing a 32-bit integer sequence whose nine MSBs are 001111111 and 27 LSBs are the 27 MSBs of the 32-bit integer version of MTGP. The sign-bit and exponent-bit imply that the represented real number is in the range [1, 2), and the 27 uniformly random LSBs imply the uniformity in that range. By subtracting 1.0, we obtain uniform distribution in [0, 1). It is often the case that we can use [1, 2) directly, for example the Box-Muller transformation for Gaussian distribution (see [26, §2]).

# 7 Concatenating outputs of several generators

Occasionally practioners ask us whether it is possible to generate a stream of a single generator (say, of MT19937) using many processors. It would be possible by using jumping-ahead. Here, "jumping-ahead by $N$ steps" means to compute the state after generating $N$ outputs from the present state. If the state has $p$ bits, then jumping-ahead costs $O(p^2)$ operations. Even after some improvements, one jumping-ahead costs a few milliseconds for $p = 19937$ ([4]), which seems too costive.

They worry that concatenating outputs of several sequence generators may result in a poorly random sequence, since the assurance of the dimension of equidistribution $k(v)$ will be lost.

In our experience, concatenating the output sequences of good pseudorandom number generators does not cause trouble. We think the dimension of equidistribution $k(v)$ is often mis-interpreted. The $k(v)$ is the dimension of uniformity for the whole period, and it gives no assurance for subsequences. However, it gives assurance on nonexistence of a linear relation: among the consecutive $k$-tuples of $v$ bits, there is no $\mathbb{F}_2$-linear relation which holds forever. This property is inherited by the subsequences, and is preserved when concatenating outputs of several generators with the same property. In addition, in the cases of DC or MTGPDC, the generators have distinct irreducible characteristic polynomials. This property does not necessarily assure the absence of correlations among these generators, but the risk of correlations would be smaller than the case where generators share one same recursion and have different seeds.

# 8 Implementation using CUDA

To use (the present version of) MTGP in a GPGPU program, one needs to write the code in CUDA [22].

A user needs to write both a host program (processed in the CPU) and a kernel program (processed in the GPU). A flow chart is in Figure 6. The left hand side is the host program. User application program (User AP) means the code which the user wants to run on GPGPU, using the pseudorandom numbers generated by MTGP. In the host program, initialization (or set up) of the GPU for the User AP kernel program is executed. Then, the initialization of the GPU for MTGP (such as texture memory set up for look-up tables, initialization of the state array in the global memory) is executed. Then, the host program calls the MTGP-kernel program (processed in the GPU). In the kernel call, the host specifies (1) number of blocks and (2) number of threads in a block, and the following arguments are passed to the kernel program (or equivalently to each block) (3) the number of pseudorandom numbers to be generated for each thread (4) a pointer to global memory pointing to the state array (5) a pointer to global memory pointing to the area where the generated numbers are stored. When the GPU is called, each block reads the state from global memory to shared memory, then each thread generates pseudorandom numbers and writes them in the global memory. After writing a specified number of outputs, each block stores the present state back into the global memory, so that in the next MTGP kernel call the generation continues. To have a consecutive array of pseudorandom numbers, we need to wait for termination of all the blocks. This is done in the host program, using the *synchronization* function. Then, the User AP kernel program is called from the host. The User AP kernel program is executed on the GPU, using the array of generated pseudorandom numbers generated by MTGP.

One drawback of this scheme is that the user needs to specify the number of pseudorandom numbers to be generated, before calling the User AP kernel.

# 9 Conclusions

We proposed MTGP pseudorandom number generators suitable for graphic processing units. The strategy to raise its efficiency is to apply many parallel threads to a single state space of a large pseudorandom number generator, to realize a large state space (and hence a long period and high-dimensional equidistribution) without loss of its generation speed. The state space is allocated in the shared memory in the GPU, and its parallelism of memory access is suitable for the design of the banks of the memory in the GPU. The generation speed of MTGP is comparable to some of the fastest generators for the GPU. For example, CURAND generator is faster than MTGP by a factor of 1.7, but statistical tests show some weaknesses of CURAND. On the other hand, WarpStandard is also faster than MTGP and passes the BigCrush statistical test suites. ¿From

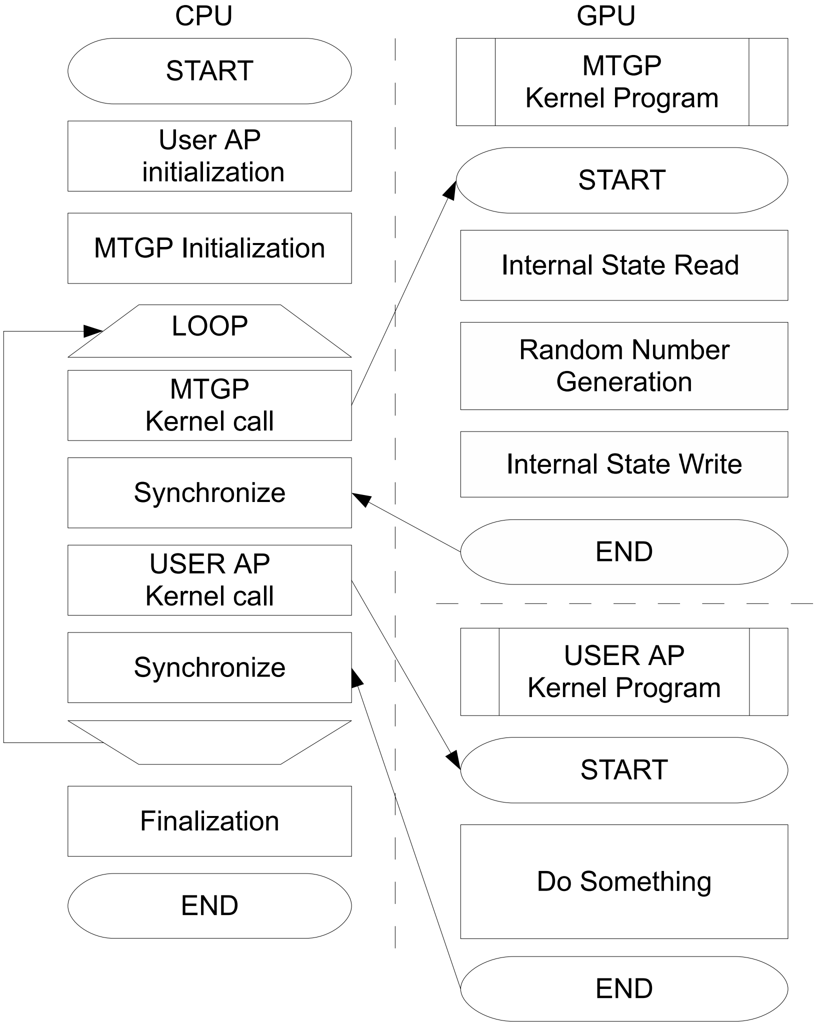


Figure 6: Flow chart for MTGP host, kernel programs and application program.

the viewpoint of the period and high-dimensional equidistribution, MTGPs have better assurance than other generators.

We also designed a parameter generator for MTGP, named MTGPDC. It runs on CPUs, receives the period and 32-bit ID, searches for a recursion parameter set with the ID embedded, and then searches for a tempering parameter set to attain high-dimensional equidistribution. This facility fits a GPGPU with many nodes. The source code of MTGP and MTGPDC, together with their 64-bit variants, are available from the url `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/`. MTGPs passed the statistical tests in the BigCrush library, except for those tests which measure $\mathbb{F}_2$-linear dependency of the output sequence. The failure in such tests is common to the Mersenne Twister and the WELL, which would not cause a problem in a usual stochastic simulation.

# Acknowledgments

This study is partially supported by JSPS/MEXT Grant-in-Aid for Scientific Research No.21654004, No.19204002, No. 23244002, No.21654017, and JSPS Core-to-Core Program No.18005. The authors are grateful for the anonymous referees for their invaluable comments.

# References

[1] S. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*, 32(2):221–251, June 1990.

[2] R. Couture, P. L'Ecuyer, and S. Tezuka. On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences. *Math. Comp.*, 60(202):749–761, 1993.

[3] Fabien. XORWOW L'Ecuyer TestU01 results, 2011. http://chasethedevil.blogspot.com/2011/01/xorwow-lecuyer-testu01-results.html.

[4] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for $\mathbb{F}_2$-linear random number generators. *Informs Journal on Computing*, 20(3):385–390, 2008. `doi:10.1287/ijoc.1070.0251`.

[5] S. Harase. Maximally equidistributed pseudorandom number generators via linear output transformations. *Math. Comput. Simul.*, 79(5):1512–1519, 2009.

[6] S. Harase. An efficient lattice reduction method for $\mathbf{F}_2$-linear pseudorandom number generators using Mulders and Storjohann algorithm. *Journal of Computational and Applied Mathematics*, 236(2):141–149, August 2011.

[7] S. Harase, M. Matsumoto, and M. Saito. Fast lattice reduction for $\mathbb{F}_2$-linear pseudorandom number generators. *Mathematics of Computation*, 80:395–407, January 2011.

[8] Khronos Group. OpenCL, 2012. `http://www.khronos.org/opencl/`.

[9] D. E. Knuth. *The Art of Computer Programming. Vol.2. Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[10] L. Howes and D. Thomas. Efficient random number generation and application using CUDA, 2009. `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch37.html`.

[11] W. B. Langdon. A fast high quality pseudo random number generator for nVidia CUDA. In *GECCO '09 Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, New York, 2009. ACM. `doi:10.1145/1570256.1570353`.

[12] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Math. Comp.*, 65(213):203–213, 1996.

[13] P. L'Ecuyer and F. Panneton. $\mathbb{F}_2$-linear random number generators. In *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 175–200, Heidelberg London New York, 2009. Springer-Verlag. C. Alexopoulos, D. Goldsman, and J. R. Wilson Eds.

[14] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):Article 22, 2007.

[15] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing, November 11-17*, New York, 2008. ACM. `doi:10.1145/1188455.1188672`.

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

[17] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[18] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer-Verlag, 2000. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`.

[19] NVIDIA. CURAND library, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf`.

[20] NVIDIA corp. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Ver 1.0.* NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050, 1.0 edition, 2007.

[21] NVIDIA corp. CUDA SDK code samples, 2009. `http://developer.nvidia.com/object/cuda_sdk_samples.html`.

[22] NVIDIA corp. *NVIDIA CUDA Programming Guide. Ver 2.3.1.* NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2.3.1 edition, 2009.

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

[24] J. Passerat-Palmbach, C. Mazel, A. Mahul, and D. Hill. Reliable initialization of GPU-enabled parallel stochastic simulations using mersenne twister for graphics processors. In *ESM 2010, Europ. Simul. Conf. Hasselt Univ. Belgique*, page 6 pages, 2010. `doi:10.1145/1570256.1570353`.

[25] M. Saito and M. Matsumoto. SIMD-oriented fast Mersenne twister : a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, 2008.

[26] M. Saito and M. Matsumoto. A prng specialized in double precision floating point number using an affine transition. In *Monte Carlo and Quasi-Monte Carlo Methods 2008*, pages 589–602. Springer-Verlag, Dec. 2009.

[27] Victor Shoup. NTL: A Library for doing Number Theory, 1990. `http://www.shoup.net/ntl/`.

[28] D. Thomas. Uniform random number generators for GPUs, 2010. `http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-uniform.html`.

[29] X. Tian and K. Benkrid. Mersenne twister random number generation on FPGA, CPU and GPU. In *Proceedings of the Adaptive Hardware and Systems Conference 2009*, pages 460–464, Los Alamitos, CA, July 2009. IEEE. `doi:10.1109/AHS.2009.11`.