Parallel Mersenne Twister

Victor Podlozhnyuk
vpodlozhnyuk@nvidia.com

June 2007

Document Change History

Version	Date	Responsible	Reason for Change
1.0	03/20/2007	vpodlozhnyuk	Initial release

Abstract

The problem of random number generation is important for many different tasks, particularly the simulation of physical and mathematical systems using Monte-Carlo methods. The iterative nature of most random number generators doesn't map very well onto the traditional graphics GPGPU paradigm because of its limitations on the number and position of memory outputs. This sample demonstrates how the Mersenne Twister, one of the best available random number generators, can be implemented in parallel using the CUDA programming model.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Introduction

The Mersenne Twister (MT) is a pseudorandom number generator algorithm developed by Makoto Matsumoto and Takuji Nishimura [1]. It has many important properties:

- Long period.
- Efficient use of memory
- Good distribution properties.
- High performance.

MT generates the bit vectors of fixed word size by the recurrence:

$x_{k+n} = x_{k+m} + (x_k^{upper} \mid x_{k+1}^{lower}) \bullet A$ (1)

$n > m$ - fixed positive integers

x_k, $k = 0, 1, \ldots$ - sequence of bit vectors with fixed width w (which is 32 in our implementation)

$x_k^{upper} \mid x_{k+1}^{lower}$ - concatenation of r most significant bits of X_k and $w-r$ least significant bits of X_{k-1}

$(x_k^{upper} \mid x_{k+1}^{lower}) \bullet A$ - multiplication of the concatenated bit vector by bit matrix A

Matrix A $(w \times w)$ is chosen for simplicity of computations in the form of

$$x \bullet A = \{x_{w-1}, x_{w-2}, \ldots, x_0\} \bullet \begin{Vmatrix} 0 & 1 & 0 & 0 & & & \\ 0 & 0 & 1 & 0 & & \vdots & \\ 0 & 0 & 0 & 1 & & & \\ & & & & 1 & & \\ & & \cdots & & & \ddots & \\ & & & & & & 1 \\ a_{w-1} & a_{w-2} & & & & a_1 & a_0 \end{Vmatrix}$$

$x = \{x_{w-1}, x_{w-2}, \ldots, x_0\}$, $a = \{a_{w-1}, a_{w-2}, \ldots, a_0\}$

Bit values are added modulo 2, (i.e. using XOR operation).

For the special form of matrix A, assuming w is less than or equal to machine word size, $x \bullet A$ multiplication can be efficiently implemented on existing hardware:

(C-style pseudo code) `x = (x >> 1) ^ ((x & 1) ? a : 0);`

In order to improve the distribution properties, each generated word is multiplied by a special $w \times w$ invertible transformation matrix from the right: $x \to z : x \bullet T$. A tempering matrix T is chosen so that $x \bullet T$ multiplication, similarly to $x \bullet A$, can be efficiently implemented with bitwise operations:

```
z = x;
z ^= (z >> u);
z ^= (z << s) & b;
z ^= (z << t) & c;
z ^= (z >> l);
```

u, s, t, l are integer numbers, b and c - suitable bit masks of word size w.

As it follows from the formula, at some position $k >= n$ x_k is the function of three preceding sequence elements: $x_k = f(x_{k-n}, x_{k-n+1}, x_{k-n+m})$; The function $f(x_{k-n}, x_{k-n+1}, x_{k-n+m})$ was defined earlier, and first n elements $x_0, x_1, ..., x_{n-1}$ are given as seeds.

Speaking in terms of algorithms, it's enough to store the generator state, n-word array (state[0], state[1], …, state[n – 1]), initialized with some initial values $x_0, x_1, ..., x_{n-1}$ (seeds), and cyclically update the state (C-style pseudo code):

```
for(i = 0; i < randN; i++){
    k = i % n;
    state[k]  = f(state[k], state[(k + 1)%n], state[(k + m)%n]);
    result[i] = tempering_transformation(state[k]);
}
```

To sum it up, Mersenne twister has the following parameters:

w -word size

n, m - degree of recursion and middle term

r - separation point in $x_k^{upper} \mid x_{k+1}^{lower}$

a - bit vector, lower row of matrix A

l, u, s, t - tempering shift parameters

b, c - tempering masks, bit vectors

> **Note:** In its canonical form, Mersenne twister is not intended for cryptographic application, since n outputs of the generator is enough to predict all future values. One of the possible solutions is to apply hash function on every output.

Implementation details

The algorithm maps well onto the CUDA programming model since it can use bitwise arithmetic and an arbitrary amount of memory writes. On one hand, the Mersenne twister, as most of pseudorandom generators, is iterative, so it's hard to parallelize a single twister state update step among several execution threads. On the other hand the GPU has to have thousands of threads in the launch grid in order to be fully utilized. The short and simple solution is to have many simultaneous Mersenne twisters processed in parallel. But even "very different" (by any definition) initial state values do not prevent the emission of correlated sequences by each generator sharing identical parameters. To solve this problem and to enable efficient implementation of Mersenne twister on parallel architectures, dcmt, a special offline library for the dynamic creation of Mersenne Twisters parameters, was developed by Makoto Matsumoto and Takuji Nishimura [2].

The library accepts the 16-bit "thread id" as one of the inputs, and encodes this value into the Mersenne Twister parameters on a per-"thread" basis, so that every thread can update the twister independently, while still retaining good randomness of the final output.

> **Note:** Though the procedure of the Dynamic Creation is normally carried out only once for chosen thread count and twisters period, it can be very time-consuming.

In the supplied source, an offline utility spawnTwisters.c, using the dcmt0.3 library, is executed first, precomputing configurations for each thread index of CUDA computation grid, which are loaded by the parallel implementation of Mersenne Twisters in runtime. Though a Mersenne twister is completely defined by 11 parameters, only the bit-vector parameters vary on a per-thread basis for the same period and dcmt seed (which is the case): a - the lower row of matrix A; b, c - tempering masks of $x \bullet T$ transformation. The rest of the (integer) parameters are shared among all the threads, and can be simply inlined into the source. In runtime each CUDA thread stores the state in a local memory array, and since n and m are the same for all threads, each thread within a warp accesses the same state index, and state arrays reads/writes are always coalesced.

Uniformly distributed random number sequences, produced by Mersenne twisters or any other random number generators, can be transformed into normal distribution $N(0,1)$ with the Box-Muller transformation [3], which is implemented as a separate kernel.

Bibliography

1. Makoto Matsumoto, Keio University/Max-Planck_Institut fur Mathematik; Takuji Nishimura, Keio University.
   Mersenne Twister. A 623-dimensionally equidistributed uniform pseudorandom number generator.
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf
2. Makoto Matsumoto and Takaji Nishimura.
   Dynamic Creation of Pseudorandom Number Generators.
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf
3. Box-Muller Transformation
   http://mathworld.wolfram.com/Box-MullerTransformation.html

Notice