# Mersenne Twister Random Number Generation on FPGA, CPU and GPU

Xiang Tian and Khaled Benkrid
The University of Edinburgh, School of Engineering
Edinburgh, UK
(x.tian, k.benkrid)@ed.ac.uk

***Abstract*—Random number generation is a very important operation in computational science e.g. in Monte Carlo simulations methods. It is also a computationally intensive operation especially for high quality random number generation. In this paper, we present the design and implementation of a parallel implementation of one of the most widely used random number generators, namely the Mersenne Twister. The latter is very widely used in high performance computing applications such as financial computing. Implementations of our parallel Mersenne Twister number generator core on Xilinx Virtex4 FPGAs achieve a throughput of 26.13 billion random samples per second. The paper also reports equivalent parallel software implementations running on an Intel Core 2 Quad Q9300 CPU with 8 GB RAM, using multi-threading technology and the Intel® Math Kernel Library (MKL), as well as on an NVIDIA 8800 GTX GPU. Comparative results show that our FPGA-based implementation outperforms equivalent CPU and GPU implementations by ~25x and ~9x respectively. Moreover, when using the same amount of energy, the FPGA can generate 37x and 35x more Mersenne Twister random samples than the CPU and the GPU, respectively.**

***Keywords-Mersenne Twister; FPGA; random number generator***

## I INTRODUCTION

A pseudorandom number generator (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's state. Pseudo-random numbers are important in practice for simulations (e.g., of physical systems using the Monte Carlo method), and are central in the practice of cryptography. Most pseudo-random generator algorithms produce sequences which are uniformly distributed by any of several tests. Common classes of these algorithms are linear congruential generators, lagged Fibonacci generators, linear feedback shift registers and generalised feedback shift registers. Recent instances of pseudo-random algorithms include Blum Blum Shub, Fortuna, and the MersenneTwister. The latter is the subject of this paper study.

The 1997 invention of the Mersenne twister algorithm, by Makoto Matsumoto and Takuji Nishimura [1], avoids many of the problems with earlier random number generators. Indeed, Mersenne twister random numbers have the colossal period of $2^{19937}$-1 iterations (>43×$10^{6000}$), are proven to be equidistributed in (up to) 623 dimensions (for 32-bit values), and can be generated faster than other statistically reasonable generators. This has made the Mersenne twister increasingly the random number generator of choice for statistical simulations and generative modeling.

Hardware acceleration based on reconfigurable hardware in the form of Field Programmable Gate Array (FPGA) has attracted a great deal of interest in the past 20 years as they offer the high performance of a dedicated hardware solution with the programmability feature. More recently, general purpose computation on Graphic Processing Units (GPUs) has been gaining great popularity as a new field of study (coined GPGPU) which aims at harnessing the parallelism available in GPUs for more general purpose computing applications as opposed to the original purpose these devices have been designed for i.e. graphics processing. With the increasing programmability of commodity GPUs, these relatively cheap and open devices are now being used in scientific computing applications as high performance computing platforms.

This paper presents the design and implementation of a parallel Mersenne Twister PRNG on FPGAs. It also compares the FPGA implementation to equivalent software implementations running on a multi-core CPU, together with a GPU. The remainder of this paper is as follows. Section 2 gives the mathematical background of Mersenne Twister PRNG, and the principles behind its parallel implementation. The following section will then report previous hardware and software implementations of the Mersenne Twister PRNG. The architecture of our FPGA design is then described in detail in section 4. We compare the results of our FPGA implementation with existing hardware designs as well our own CPU and GPU-based implementations in section 5. Finally, conclusions are drawn.

## II MATHEMATICAL BACKGROUND

The Mersenne Twister algorithm generates a sequence of word vectors, which are considered to be uniform pseudorandom integers between 0 and $2^w - 1$. Dividing by $2^w - 1$, each word vector can be a real number in [0, 1].

With the restriction that $2^{nw-r} - 1$ is a Mersenne prime. For a word $x$ with $w$ bit width, it is expressed as the recurrence relation:

$$x_{k+n} = x_{k+m} \oplus (x_k^u \mid x_{k+1}^l)A \quad k = 0,1,... \quad (2.1)$$

With | as the bitwise or and ⊕ as the bitwise exclusive or (XOR), $x_u$, $x_l$ being $x$ with upper and lower bitmasks applied. We choose a form of the matrix $A$ so that multiplication by $A$ is very fast:

978-0-7695-3714-6/09 $25.00 © 2009 IEEE
DOI 10.1109/AHS.2009.11

IEEE computer society

$$A = R = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots a_0) \end{pmatrix} \qquad (2.2)$$

With $I_{n-1}$ as the $(n-1) \times (n-1)$ identity matrix (and in contrast to normal matrix multiplication, bitwise XOR replaces addition). The rational normal form has the benefit that it can be efficiently expressed as:

$$xA = \begin{cases} shiftright(x) & if\ x_0 = 0 \\ shiftright(x) \oplus a & if\ x_0 = 1 \end{cases} \qquad (2.3)$$

Where

$$x = (x_k^u \mid x_{k+1}^l) \qquad k = 0,1,\ldots \qquad (2.4)$$

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

To improve $k$-distribution to $v$-bit accuracy, we multiply each generated word by a suitable $w \times w$ invertible maxtrix $T$ from the righ. For the tempering matrix $x \rightarrow z = xT$, we chose the following successive transformations:

$$y = x \oplus (x >> u) \qquad (2.5)$$

$$y = x \oplus (y << s) \,\&\, b \qquad (2.6)$$

$$y = x \oplus (y << t) \,\&\, c \qquad (2.7)$$

$$y = x \oplus (x >> l) \qquad (2.8)$$

With <<, >> as the bitwise left and right shifts, and & as the bitwise and. The first and last transforms are added in order to improve lower bit equidistribution. From the property of TGFSR, $s+t \geq \lfloor w/2 \rfloor - 1$ is required to reach the upper bound of equidistribution for the upper bits.

The coefficients for MT19937 are:

$(w, n, m, r) = (32, 624, 397, 31)$
$a = 9908B0DF_{16}$
$u = 11$
$(s, b) = (7, 9D2C5680_{16})$
$(t, c) = (15, EFC60000_{16})$
$l = 18$

Figure 1 gives the pseudocode of MT19937. Mersenne Twister implementations cannot be parallelized across parallel computing cores simply through changing the initial seed for each core as this does not provide uncorrelated sequences on each generator sharing identical parameters. To solve this problem and enable Mersenne Twister parallel implementations, the authors of MT19937 developed a library for the dynamic creation of Mersenne Twister parameters [2]. This library receives user's specification such as word length, period, size of working area, and a process ID, so that ID number is encoded in the characteristic polynomial of Mersenne Twister.

## III RELATED WORK

Various software implementation of Mersenne Twister were reported in the literature using different high level languages, such as C/C++, JAVA, and C#. One of the most optimized versions can be found in the Intel® Math Kernel Library (MKL) [3]. The MKL provides a vector statistical library [3][4] including both MT19937 and a parallel version of Mersenne Twister (MT2203). Moreover, ss part of NVIDIA Corporation's integrated developing environment, called Compute Unified Device Architecture (CUDA) [5], there is a Mersenne Twister random number generator implementation targeted at NVIDIA's GPUs [6]. There are also few published reports on FPGA

```
// Create a length 624 array to store the state of the generator
 int[0..623] MT
 int index = 0

 // Initialize the generator from a seed
 function initializeGenerator(int seed) {
     MT[0] := seed
     for i from 1 to 623 { // loop over each other element
         MT[i] := last 32 bits of(1812433253 * (MT[i-1] xor (right shift by 30 bits(MT[i-1]))) + i) // 0x6c078965
     }
 }

 /* Extract a tempered pseudorandom number based on the index-th value, calling generateNumbers() every 624 numbers */
 function extractNumber() {
     if index == 0 {
         generateNumbers()
     }

     int y := MT[index]
     y := y xor (right shift by 11 bits(y))
     y := y xor (left shift by 7 bits(y) and (2636928640)) // 0x9d2c5680
     y := y xor (left shift by 15 bits(y) and (4022730752)) // 0xefc60000
     y := y xor (right shift by 18 bits(y))

     index := (index + 1) mod 624
     return y
 }

 // Generate an array of 624 untempered numbers
 function generateNumbers() {
     for i from 0 to 623 {
         int y := 32nd bit of(MT[i]) + last 31 bits of(MT[(i+1) mod 624])
         MT[i] := MT[(i + 397) mod 624] xor (right shift by 1 bit(y))
         if (y mod 2) == 1 { // y is odd
             MT[i] := MT[i] xor (2567483615) // 0x9908b0df
         }
     }
 }
```

Figure 1. Pseudocode of MT19937

implementations of the Mersenne Twister algorithm. One of them [7] reported a random number generation throughout of 38.5 million samples per second. Another hardware implementation was published in [8] provides 20 million random samples per second. Both of the above hardware implementations targeted a single FPGA chip. Another FPGA implementation of the Mersenne Twister was reported in [9] and targeted a Xilinx Virtex-5 LX50 device. The design was captured in Impulse C and used ten parallel processes that act as independent Mersenne Twister generators. This resulted in a throughout of 740 million random samples per second, using 20 computing cores, excluding the host I/O time.

## IV HARDWARE IMPLEMENTATION

According to the MT2203, the process of generating Mersenne Twister numbers can be separated into the following 4 steps:

- Generating the tempering matrix for each computing core based on the given word length, size of working area, and process ID;
- Initializing the generator based on the given seed number;
- Generating the untempered numbers;
- Tempering.

After an initial feasibility analysis, we noticed that the first two steps consume the most of hardware resources, but the least computing time: both of these two steps only run once at the beginning of the generation, and the time does not increase no matter how long the random sequence is. Moreover, the following two steps only require shift and XOR operations which consume relatively few logic resources on FPGA. The output of the first step is the 12 parameters needed by step 3 and 4, and the output of the second step is 624 initialized numbers. In our hardware implementation, step 1 and 2 are performed in software while step 3 and 4 are performed in hardware. The architecture of a Mersenne Twister random number generator core is given in Figure 2. However, existing designs[7][8] are all based on generating the initial number on FPGA, which makes the logic more complicated, and the clock speed is hence much lower, as more logic is consumed in extra arithmetic units, e.g. multipliers. We can see that in [7], the clock frequency, the number of occupied slices and LUTs are 24.234MHz, 330, and 539, respectively. And in [8], the number of slices is 420. However, these parameters, which can be seen in detail in next section, are significantly optimized in our design.

Based on (2.3) and (2.4), we have to read two numbers from the RAM at step 2 each clock cycle. To pipeline the random number generator, which means one number can be generated each clock cycle, two block RAMs are needed: one is for $x^l$, and the other is for $x^u$. Note that we can pre-compute the parameters and hardwire them to each Mersenne Twister core. If the tempering numbers are fixed, the independence of the sequence does not depend on the initial numbers, hence we can use the same initial numbers on each core. Therefore, we also initialized the block RAMs with a set of pre-computed initial numbers, which can be used as a default option if the user wants to skip step 2.

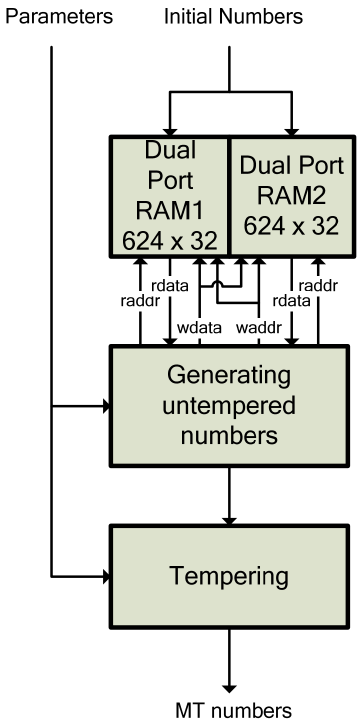

Figure 2. Architecture of Mersenne Twister random number generator core

## V RESULTS AND COMPARISON WITH GPGPU AND MULTI-CORE CPU IMPLEMENTATIONS

We implemented our Mersenne Twister random number generator using Verilog-HDL on a Xilinx Virtex4 VFX100 FPGA (xc4vfx100-10ff1152) [10], using Xilinx ISE 9.2 software for hardware synthesis. The synthesis report shows that 128 slices and 4 BlockRAMs are used by a single Mersenne Twister core (32 bit word length), and the peak clock frequency can reach ~265MHz. The detailed resource utilization is shown in Table 1. To maximize logic utilization, we have also experimented with the use of distributed RAM for the dual port RAM implementation. When synthesizing the Mersenne Twister core using distributed RAM, 5815 slices were used for a single Mersenne Twister core, but the maximum clock frequency fell by 28%, to 191MHz. Table 2 gives the performance of our Mersenne Twister core using both BlockRAMs (BRAMs) and distributed RAM, and using just BlockRAMs, on the Virtex 4 FPGA.

We can see that the use of distributed RAM decreases the overall throughput even if the number of cores increases.

We calculated the initial numbers and the tempering matrix parameters using the dcmt library [11]. Since our Mersenne Twister core will be embedded into FPGA, we assume that the random samples will be used by another FPGA process and not sent back to the host. Hence, the computation times reported in this paper only include the initial parameters' calculation time, the input time and the core execution time, with no output time. Experiment shows that the initial parameters calculation time and the input time are both negligible compared to the core execution time.

We compare our results to the only multi-core parallel FPGA implementation of Mersenne Twister reported in the literature [9] in Table 3 below. This shows the superiority of our implementation.

TABLE 1. RESOURCE UTILIZATION

| | |
|---|---|
| Slices | 128 |
| FFs | 193 |
| LUTs | 213 |
| BRAMs | 4 |

TABLE 2. PERFORMANCE OF MULTI-CORE IMPLEMENTATION ON THE VIRTEX 4 FPGA

| | BRAM | BRAM & Distributed RAM |
|---|---|---|
| No. of Cores | 99 | 104 |
| Freq.(MHz) | 265 | 191 |
| Thru'put(billion) | 26 | 20 |

TABLE 3. COMPARISON OF EXISTING PARALLEL HARDWARE DESIGN

| Design | Proposed | [9] |
|---|---|---|
| Platform | Virtex4 VFX100 | Virtex5 LX50 |
| No. of Cores | 99 | 20 |
| Freq(MHz) | 265 | 111 |
| Throughput (billion/sec) | 26 | 0.74 |

We also implemented a Mersenne Twister random number generator on an NVIDIA 8800 GTX GPU, which has a core clock frequency of 575MHz, and 768 MB of 384-bit GDDR3 memory at 1.8 GHz, giving it a memory bandwidth of 86.4 GB/sec. The 128 GPU stream processors are clocked at 1.35GHz by default, and form the main source of parallelism in our implementation. When evaluating the GPU execution time, the initialization time is not included, as was the case in our FPGA implementation. The resulting GPU Mersenne Twister implementation throughout is 2.96 billion samples per second.

We also compared our FPGA implementation with a software implementation of the Mersenne Twister running on a workstation with an Intel Core 2 Quad Q9300 2.5GHz CPU. We used the Intel® Math Kernel Library (MKL) and compiled our C code using Intel® C++ Compiler. As the MT2203 function in MKL only needs a process ID to generate a set of parameters, and then provide an independent random sequence [12], we can easily realize a multi-threaded program that uses all of the cores in the CPU in parallel. To do this, we used the POSIX thread library [13] to implement a multi-threaded program. Four threads are generated in our C++ program for our Quad Core CPU. The experiment shows that 1.06 billion samples can be generated per second.

The throughputs of FPGA, CPU, and GPU are 1.06, 2.96, and 26.13 billion samples per second. The speed-up figures for FPGA, CPU and GPU implementations are shown in Figure 3 where the speed-up figures are normalized with the CPU implementation. This shows that our FPGA-based implementation can achieve ~25x speed-up compared to Intel Core 2 Quad Q9300 CPU implementation and a ~9x speed-up compared to NVIDIA 8800 GTX GPU implementation. It is worth mentioning at this stage that the data word lengths used for FPGA, CPU and GPU implementations were all 32 bits, which provides for a fair comparison.

We also measured the power consumption of all three implementations using a power meter. The idle power of the host machine used for all three implementations is ~130 Watts. When running the Mersenne Twister implementation on GPU, the power rose to ~212 watts, whereas it rose to ~188 watts when running the C++ Mersenne Twister program on the CPU. The extra power consumed by the FPGA implementation is about ~20 watts. By deducting the idle power for the GPU and CPU implementations, and dividing it by the throughput, we get the throughput per Joule of energy consumed by each implementation (see Figure 4). From this figure, we can see that the energy efficiency of the Quad core CPU and the GPU are similar. However, FPGAs are much more energy efficient. Indeed, using the same amount of energy, FPGAs can generate 37x and 35x more Mersenne Twister random samples than the CPU and the GPU, respectively.

## VI CONCLUSION

A parallel Mersenne Twister random number generator engine has been presented in this paper. By exploiting deep pipelining, effectively partitioning the assignments for hardware and software, and fully utilizing the FPGA resources on the reconfigurable device, our FPGA implementation outperforms the more mainstream multi-core CPU and GPU platforms by ~25x and ~9x, respectively. Moreover, power measurements showed our FPGA implementation to be 37x and 35x more energy efficient than CPU and GPU respectively. These experiments clearly show the superiority of FPGAs to CPUs and GPUs on speed and energy consumption grounds.

However, other comparison criteria include hardware and software cost, development time, maintenance costs, and technology maturity, which might differ from one user/developer to another. Moreover, in large system implementations, the Mersenne Twister random number generator is likely to be a small component. Porting the same conclusions to a large scale system implementation is hence not guaranteed. Nonetheless, the experiment shown in this paper, and others, show that FPGAs can easily be viable in heterogeneous multi-processor systems.

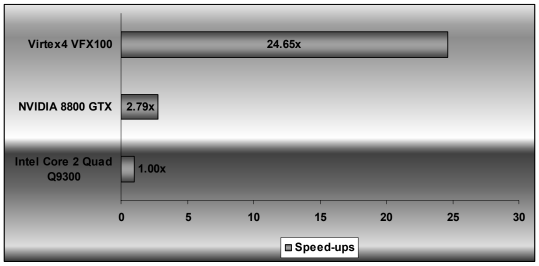

Figure 3. Speed-up comparison between FPGA, CPU and GPU

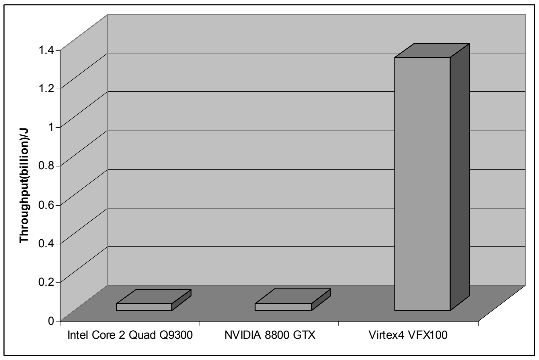

Figure 4. Throughput per Joule comparison

# REFERENCES

[1] Matsumoto, M. and Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*(1). 3-30.

[2] Matsumoto, M. and Nishimura, T. Dynamic Creation of Pseudorandom number generator. in Niederreiter, E.H. and Spanier, J. eds. *Monte Carlo and Quasi-Monte Carlo Methods*, Springer, 2000, 56-69.

[3] Intel® Math Kernel Library Reference Manual, Intel Corporation, 2007.

[4] Intel® Math Kernel Library Vector Statistical Library Notes, Intel Corporation, 2007.

[5] NVIDIA CUDA Compute Unified Device Architecture Reference Manual, NVIDIA Corporation, 2008.

[6] Pdlozhnyuk, V. Parallel Mersenne Twister, NVIDIA Corporation, 2008.

[7] Sriram, V. and Kearney, D., An area time efficient field programmable mersenne twister uniform random number generator. in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, (LasVegas, USA, 2006), CSREA Press.

[8] Chandrasekaran, S. and Amira, A., High Performance FPGA Implementation of the Mersenne Twister. in *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, (2008), 482-485.

[9] Pellerin, D., Trexel, E. and Xu, M. FPGA-based hardware accleration of C/C++ based applications, Impulse Accelerated Technologies, 2007.

[10] Virtex-4 Family Overview, Product Specification, Xilinx, 2007.

[11] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html.

[12] Intel® Math Kernel Library Vector Statistical Library Notes, Intel Corporation, 2007.

[13] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html.