Mersenne Twister Implementation on a GPU

By Rohit Bal & Sagar Keer
CSE 704 Spring 2011
Instructor: Dr. Russ Miller

Introduction

What is the Mersenne Twister?

- A pseudo-random number generator
- Developed by Matsumoto & Nishimura in 1997
- For a k-bit word length, produces uniform distribution in the range $[0,2^k-1]$

Introduction

Properties:

- Long Period
- Efficient use of memory
- Good distribution properties
- High Performance

Introduction

- Mersenne Twister focuses on having an almost perfectly uniform distribution
- Designed for statistical simulations like the Monte-Carlo simulations where uniformity plays a key role
- In its canonical form, it is not suitable for cryptographic applications as future values can be predicted from a limited set of outputs

Rationale

Parameters:

- w - word size
- n, m - degree of recursion, middle term
- Also n>m>1
- 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

Rationale

- Bit vectors are given by the recurrence relation:

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

where:

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

Rationale

- Matrix A is a $w*w$ matrix of the form

$$
\begin{vmatrix}
0 & 1 & 0 & 0 & & & \\
0 & 0 & 1 & 0 & & \cdots & \\
0 & 0 & 0 & 1 & & & \\
 & & & & 1 & & \\
 & & \cdots & & & \cdots & \\
 & & & & & & 1 \\
a_{w-1} & a_{w-2} & & & & a_1 & a_0
\end{vmatrix}
$$

Rationale

- For better distribution, transformation is applied using the tempering vector T to each bit vector, with the operations:

  $z = x$

  $z \mathrel{\hat{}}= (z >> u)$

  $z \mathrel{\hat{}}= (z << s)\ \&\ b$

  $z \mathrel{\hat{}}= (z << t)\ \&\ c$

  $z \mathrel{\hat{}}= (z >> l)$

  where *b, c, l, u, s, t* are the tempering components as defined earlier

Rationale

- For a position $k >= n$, x_k is the function of three preceding sequence elements ie:

  $$x_k = f(x_{k-n}, x_{k-n+1}, x_{k-n+m})$$

- To sum up, we get an almost perfectly uniform distribution using n initial seeds

Implementation

- C with CUDA
- Used MTGP libraries
- MAGIC system
- Worked with 32-bit word size
- Produced integral values

Implementation

Algorithm maps well to CUDA because:

- Uses bitwise arithmetic
- Arbitrary amount of memory writes

Implementation

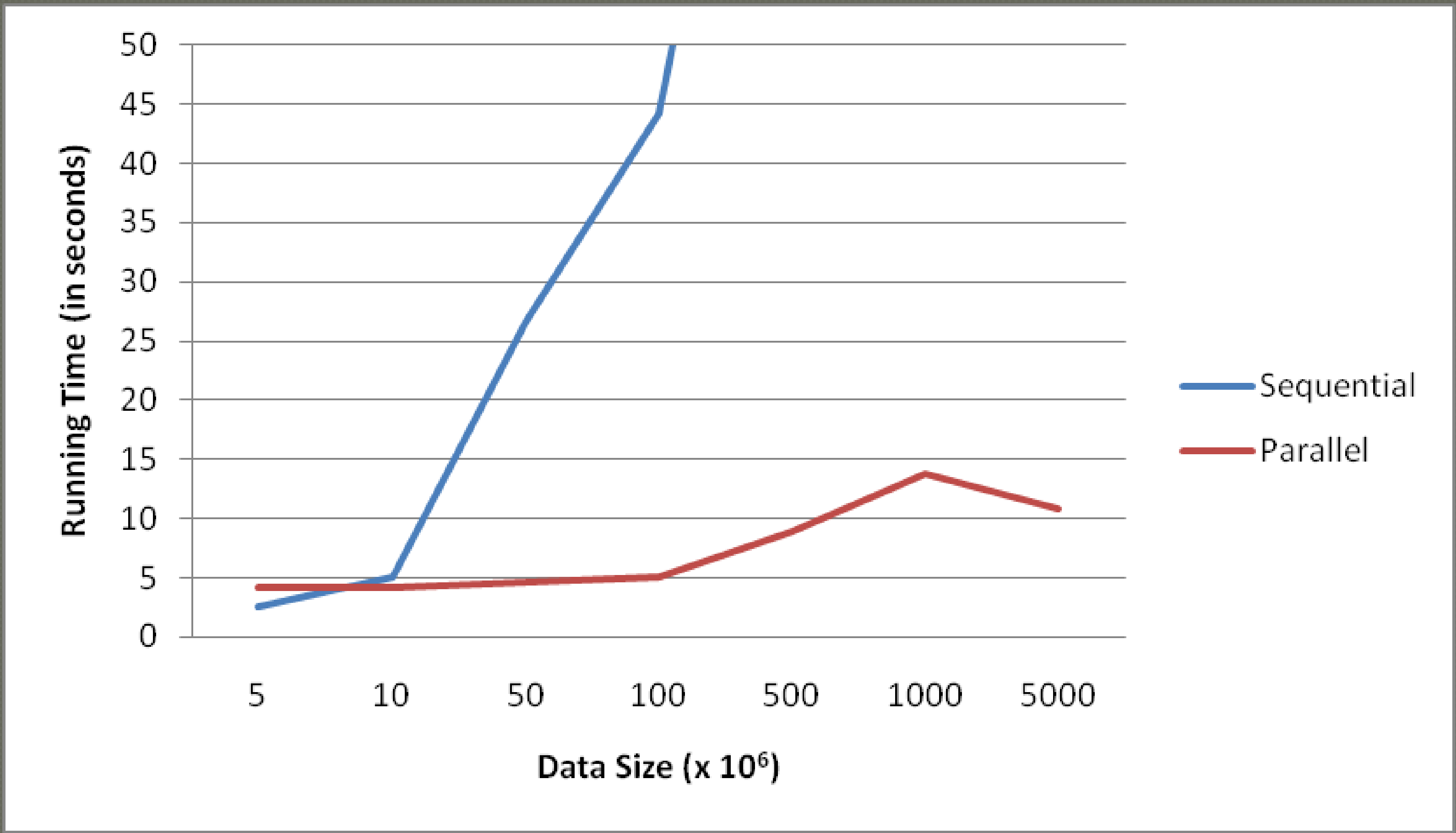
- Parameter 'sets' determine period
- Total number of parameter sets is 128
- In other words, 128 pseudorandom sequences can be generated for each period.
- Maximum period is $2^{44497} - 1$!!

- However, we used only one period : $2^{23209} - 1$

Implementation

- Each thread uses its thread_id as a parameter (like a seed)
- This parameter is used to calculate parameters defined by the MT algorithm
- This guarantees randomization at thread level

Results



Running Time (in seconds)
0
5
10
15
20
25
30
35
40
45
50
5
10
50
100
500
1000
5000
Data Size (x 10^6)
Sequential
Parallel

References

- MTGP -http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/index.html
- Makoto Matsumoto, Keio University/Max-Planck_Institut fur Mathematik; TakujiNishimura, Keio University.
- NVIDIA Sample Code for MT

Thank you!