Efficient random number generation and application using CUDA

MRINAL PALIWAL, 2016164
SHUBHANG SATI, 2016198

1 LITERATURE SURVEY

Random number generators (RNG) are widely being used in number of applications, particularly simulation and cryptography. Monte carlo simulations are one such area where RNG are used. Monte carlo methods can be used to get approximate solutions for problems unsolvable or very difficult to solve otherwise (e.g., estimation of π). Law of large numbers is followed here more the number of trials, better the approximation will be. These independent trials are inherently parallelizable. The quality of simulation is dependent on the quality of underlying RNG used. We perform efficient random number generation and application using CUDA.
To generate uniform distributions, we use Mersenne Twister, Combined Tausworthe Generator, and Hybrid Tausworthe Generator. Makoto Matsumoto and Takuji Nishimura [2] provide the implementation details of the Mersenne Twister PRG. Its name derives from the fact that its period length is chosen to be a Mersenne prime. Tausworthe algorithm is also an iterative algorithm. However, Tausworthe algorithm, with its restrictions, has bad statistical properties on its own. A modified approach. which uses exclusive-or to combine the results of two or more independent binary matrix derived streams, providing a stream of longer period and much better quality, was given by [1] as combined Tausworthe generators. Hybrid Tausworthe generators use a combination of LCG-based generators and combined Tausworthe generators given by [3].

2 ALGORITHM ANALYSIS

2.1 Mersenne Twister

Mersenner Twister (MT) name is derived from the fact that its period length is chosen to be a Mersenne prime. For a particular choice of parameters, the algorithm provides a super astronomical period of $2^{199372} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy, while using a working area of only 624 words.
MT generates the bit vectors of fixed word size by the recurrence:

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

- x_k is sequence of bit vectors with fixed width w
- ${x_k}^{upper} | {x_{k+1}}^{lower}$ concatenation of r most significant bits of X_k and w - r least significant bits of $X_k - 1$
- Matrix A (w × w) is chosen for simplicity of computations
- In order to improve the distribution properties, each generated word is multiplied by a w × w invertible transformation matrix (T) from the right.

For every step, the state of the generator - the n-word array given by (state[0], state[1], ... , state[n-1]) is stored. This state of the words is then updated, given by:

```
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]);
```

Authors' addresses: Mrinal Paliwal, 2016164, mrinal16164@iiitd.ac.in; Shubhang Sati, 2016198, shubhang16198@iiitd.ac.in.

```
}

- tempering_transformation is the transformation given by transformation matrix (T)
```

However, as most of pseudorandom generators, is iterative in nature. As a result MT is not inherently easily parallelizable. An optimal stratergy would be to run multiple MT's simultaneously in parallel. However, even 'very different' (by any definition) initial state values for multiple MT launches do not prevent the emission of correlated sequences by each generator sharing identical parameters.
To overcome this problem, we use the seeds generated by an the offline library by Makoto Matsumoto and Takuji Nishimura [2] that enables efficient implementation of Mersenne twister on parallel architectures. These seeds are then given to different thread ids that individually generate random numbers from MT. This ensures that that every thread can update the twister independently, while still retaining good randomness of the final output.

2.2 Combined Tausworthe Generator

The mersenne twister algorithm uses extremely large sparse matrix and large vectors to transform one set of bits into a new set of bits. The combined tausworthe generator has a much smaller state space as compared to the mersenne twister algorithm. It uses XOR operations to combine two or more independent streams of random bits to achieve a stream of longer period and better quality.
A single step in the tausworthe generation is as described below:

```c
// Tausworthe step
// A, B, C and M are predefined and fixed

typedef unsigned long long llu;

llu gen(llu *s) {
    llu b;
    b = (((*s << A) ^ *s) >> B);
    *s = (((*s & M) << C) ^ b);
    return *s;
}
```

The above code simply takes a pointer to an unsigned long long seed value and performs some bitwise operations to obtain the next seed value or the generated random value.

Using combined tausworthe alone for generating a high number of random numbers is not very efficient because there is a significant correlation as suggested by statistical tests [3]. This correlation exists even for a small sample size. To tackle this a hybrid approach is suggested which is discussed in the next section.

2.3 Hybrid Tausworthe Generator

The combined tausworthe generator alone shows significant correlation between generated values and is not very suitable for a large sample space. So a hybrid approach is used which combines three steps of the combined tausworthe generator. The state space increases here but the randomness of the generated random numbers is also increased. The results of the three tausworthe steps are combined by taking their XOR. This resulting value is again combined along with a linear congruential generator by taking the XOR of the output of the LCG with the previously obtained result. The result is a PRNG with a large period and better random stream of bits.

The resulting generator period will be the LCM of the periods of the individual steps and if all periods are co-prime then the resulting period will be the product of individual periods. [3]

A structure for the state of the generator is defined as follows. The four states are for the three tausworthe steps and the fourth is for the LCG step.

```
typedef struct {
    llu s1, s2, s3, s4;
} tauswortheState;
```

The LCG step simply does the following:

```
llu lcg(llu *s, llu a, llu b) {
    *s = a * *s + b;
    return *s;
}
```

The modulo step of the LCG can be avoided since *unsigned long long* has an inherent limit of 2^{64} bits which takes care of the modulo step. The combination chosen here is a combination of the combined tausworthe "tauss88" from [1] and a similar LCG generator as in [5].

For running this hybrid tausworthe generator on multiple threads, it has to be ensured that each thread gets a unique set of four state values. This values can be obtained using a PRNG on the CPU.

3 FUTURE WORK

- Analysis of Mersenne Twister, Combined Tausworthe Generator, and Hybrid Tausworthe Generator's throughput.
- Implementation of Wallace Generator for Gaussian Distributions
- Implementation of Ziggurat plus Mersenne Twister, and Hybrid Tausworthe plus Box-Muller
- Analysis of implemented algorithms on Asian method and Lookback Option
- Estimating the value of pi using monte carlo simulation

REFERENCES

[1] L'ecuyer, P., 1996. Maximally equidistributed combined Tausworthe generators. Mathematics of Computation of the American Mathematical Society, 65(213), pp.203-213.

[2] Matsumoto, M. and Nishimura, T., 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), 8(1), pp.3-30.

[3] Howes, L. and Thomas, D., Efficient Random Number Generation and Application Using CUDA. GPU Gems Chapter 37

[4] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 1992. Numerical Recipes in C. Second edition. Cambridge University Press.