# GPU Accelerated Scalable Parallel Random Number Generators

Shuang Gao, Gregory D. Peterson
Department of Electrical Engineering and Computer Science, University of Tennessee
Knoxville, Tennessee, USA
[sgao3, gdp]@utk.edu

***Abstract*— SPRNG (Scalable Parallel Random Number Generators) is widely used in computational science applications, particularly on parallel systems. The LFG and LCG are two frequently used random number generators in this library. In this paper, LFG and LCG are implemented on GPUs in CUDA. As a library for providing random number to GPU scientific applications, GASPRNG is designed to have one generator per thread and could generate thousands of random numbers simultaneously. The performance of this implementation is measured on Tesla s1060 and Fermi GTX480 GPUs, which provide 20x ~ 100x speedup compared with the CPU version, and 10x ~ 80x speed up compared with the FPGA implementation. An application of Pi calculation using Monte Carlo methods is also presented, with results showing that this code can integrate to computational science applications.**

***Keywords-SPRNG, GPGPU, random number generator***

## I. INTRODUCTION

Random number generators are widely used in computational science applications. High quality random numbers are necessary for these applications to achieve a high quality solution. SPRNG (Scalable Parallel Random Number Generators) is a one of the best libraries for generating random numbers for parallel applications; it provides support with fast, scalable and parallel random number generation. SPRNG generators provide good statistical properties and boast very long periods with support for large numbers of parallel, independent streams [1,6]. This makes SPRNG convenient and popular with computational scientists. In recent years, GPGPUs provide increasingly powerful computing resource for high performance computing; with more and more applications ported to GPU, random number generators for GPUs are needed. Some GPU random number generators have been proposed [3,4,5]. Demchik implemented basic uniform pseudo-random number generators on ATI GPUs [3]; Langdon implemented the Park-Miller PRNG [4]. The CUDA SDK includes the Marsenne Twister as well. The purpose of this paper is to enable GPU acceleration for SPRNG (GASPRNG), we implemented a CUDA version of the Lagged Fibonacci Generator (LFG) and Linear Congruential Generator (LCG) random number generators; as a pure device library, it has a simple user interface and serves GPU applications with fast and highly parallel random number generation.

## II. GASPRNG IMPLEMENTATION

### A. Overall concerns

Scalable GPU applications require random number generators implemented on the GPUs. It is impractical to use a CPU random number generator for GPU programs, because compared with the speed of consuming random numbers, the PCIE (PCI Express) bandwidth is low and can become the performance bottleneck. It is not efficient to use a GPU library with its own global kernel and host functions, for in such circumstance, the random numbers generated have to be written and read at least once between on chip storage and global memory, causing a performance decrease. For the proposed framework, the random number generated could be consumed with no data transfer. The result could be saved in registers or memory and be used immediately.

Assigning one random generator per thread is the most efficient way for providing random number to GPU applications. Compared with a random number generator that requires cooperation of multiple threads, this strategy eliminates the communication among threads; to generate the same amount of random numbers, it provides much better performance.

Our goals for GASPRNG include designing a “pure” device function library, efficient data storage, and good, scalable performance. This can be summarized as:

- Identical random number streams as with SPRNG
- Fast execution,
- Efficient memory usage,
- Simple application programming interface, and
- Flexible support for large numbers of streams.

### B. LFG Implementaion

The LFG generator is widely used because of its excellent random number distribution. The LFG random number generator uses:

$$Z(n) = Z(n - k) * Z(n - l) \pmod{2^{64}} \tag{1}$$

where $k$ and $l$ are lags. To get the current random number, previous random numbers with indices $(n - k)$ and $(n - l)$ are both used. SPRNG supports 10 sets of lags.

According to (1), the implementation requires a big data structure with several arrays as data members. Depending on the value of the lags, the length of these arrays could reach the maximum of 1279 in SPRNG. Such a large array of data per thread cannot fit in shared memory; it requires data reduction and an efficient way to distribute data among GPU memories.

Based on previous observation, the GPU version of LFG is implemented as layers, shown in Fig. 1. The data structure of the SPRNG LFG generator is split into several parts, and put into different GPU memories: registers, global memory, and shared memory. The duplicated data across generators are merged to save storage space.

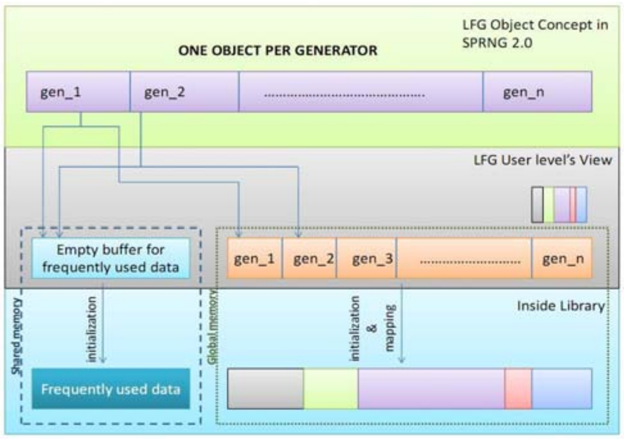

Figure 1. Structure of LFG on GPU

Global memory holds the long arrays required for storing previous results and other temporary data. Library users allocate the generator buffer. From the callers' point of view, it is an array of objects of generators; however, inside the library, the data is interleaved. A simple memory mapping is setup to map the logical location of each data to addresses in global memory. Although involving more operations, the global memory performance improvement outweighs the extra work.

Shared memory is used as a constant buffer and caches. (1) It stores all members used as constant. Since it has efficient broadcast to satisfy read operations, the performance is improved in this way. (2) Frequently used values of small size are saved in shared memory to improve read and write efficiency, such as hptr.

The function of generating each random number is optimized; global memory operations are eliminated as much as possible.

## C. *LCG Implementaion*

The LCG is based on:

$$Z(n) = a \times Z(n - 1) + p \text{ (Mod M)} \qquad (2)$$

where $a$ is a multiplier and $p$ is a prime number. To generate the $n$th random number, the result of $n$-1 is referenced. Since the lag here is only 1, the space for saving previous results is much less than for the LFG.

GASPRNG eliminates the duplicated data and fits it into different kinds of GPU memories as shown in Fig.2. Since the LCG requires little space for saving previous results, it is possible to keep everything inside shared memory. The shared memory is used as a cache and constant buffer here, just like the LFG; however, unlike LFG, there is only one object of the data structure for each thread block: it includes arrays of seeds indexed by threadIdx as well as constants used in generating random numbers. Compared with LFG, the speed of LCG is improved because the latter avoids global memory operations. In addition, the interface is cleaner since we do not need to allocate global buffers for the library.

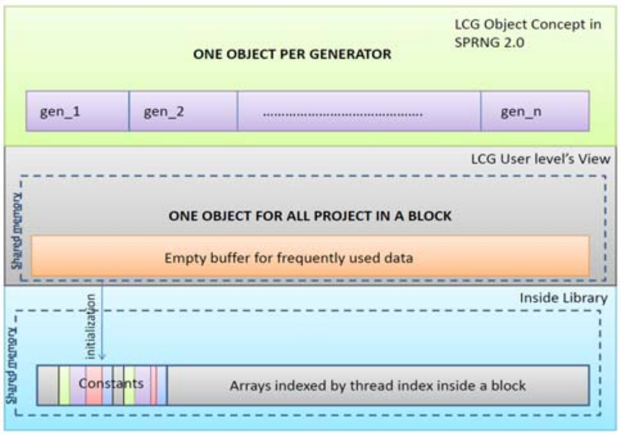
LCG Object Concept in SPRNG 2.0
ONE OBJECT PER GENERATOR
gen_1
gen_2
gen_n
LCG User level's View
Shared memory
ONE OBJECT FOR ALL PROJECT IN A BLOCK
Empty buffer for frequently used data
Inside Library
initialization
Shared memory
Constants
Arrays indexed by thread index inside a block

Figure 2. Structure of LCG on GPU

## III. GASPRNG PERFORMANCE

The random number sequences generated by GASPRNG are identical to the results from SPRNG. Experiments are made to test the performance of the GASPRNG LFG and LCG, with results is compared with the CPU version (generated with SPRNG2.0 [1]) and the FPGA version [2]. Finally, a Monte Carlo example for calculating Pi is presented, which consumes the random numbers generated by the GASPRNG LFG and displays good speedup.

The devices we used in the experiments include:

- Tesla and Fermi: two GPU chips: Tesla s1060 and Fermi GTX480;
- CPU of Intel Xeon X5570 @ 2.93Ghz,
- Xilinx V2P70 FPGA on Cray XD1

The default thread number of each block is 128; using 64 threads per block shows similar performance. It indicates that GASPRNG requires little storage resources and users could have more space for applications.

The LFG performance is shown in Table 1; the lags used are (1279, 861). On Fermi, the GPU LFG gains a 28x speedup over the CPU and 21x over the FPGA. On Tesla, the speedup is 13x over the CPU. Table 2 shows the GASPRNG LCG performance, with speedup up to 98x over the CPU and 82x over the FPGA version [2]. The results show that there is trivial difference in performance when using 90 or 120 blocks, which indicates that the library allows more choice for library users during their design of GPU programs.

TABLE I. LFG PERFORMANCE OF GASPRNG

| | Number of Blocks | Number of Generators | Number of RNs | Time (seconds) | Rate (Millions of RNs/second) |
|---|---|---|---|---|---|
| Fermi | 90 | 11520 | $2.7\text{x}10^{11}$ | 86.36 | 3183MRNs |
| | 120 | 15360 | $2.7\text{x}10^{11}$ | 84.94 | 3236MRNs |
| Tesla | 90 | 11520 | $2.7\text{x}10^{11}$ | 187.54 | 1466MRNs |
| | 120 | 15360 | $2.7\text{x}10^{11}$ | 162.10 | 1696MRNs |
| CPU | - | 1 | $1.0\text{x}10^{9}$ | 8.81 | 114MRNs |
| FPGA | - | 1 | $1.0\text{x}10^{9}$ | 6.67 | 150MRNs |

TABLE II. LCG PERFORMANCE OF GASPRNG

| | Number of Blocks | Number of Generators | Number of RNs | Time (seconds) | Rate (Millions of RNs/second |
|---|---|---|---|---|---|
| Fermi | 90 | 11520 | $2.7\text{x}10^{11}$ | 9.72 | 28280MRNs |
| | 120 | 15360 | $2.7\text{x}10^{11}$ | 8.46 | 32509MRNs |
| Tesla | 90 | 11520 | $2.7\text{x}10^{11}$ | 40.9 | 6721MRNs |
| | 120 | 15360 | $2.7\text{x}10^{11}$ | 41.1 | 6688MRNs |
| CPU | - | 1 | $1.0\text{x}10^{9}$ | 3.02 | 331MRNs |
| FPGA | - | 1 | $1.0\text{x}10^{9}$ | 2.51 | 398MRNs |

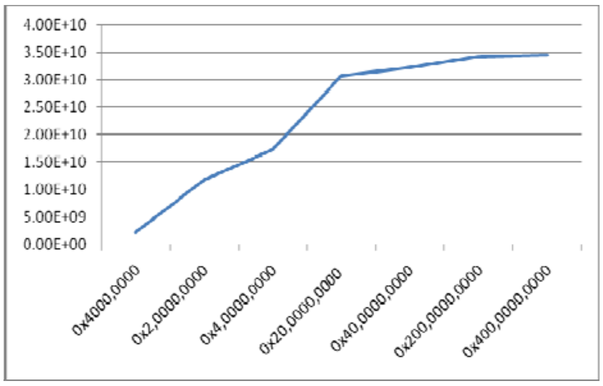

Figure 3. GASPRNG LCG Performance (number of RNs generated versus generation rate)

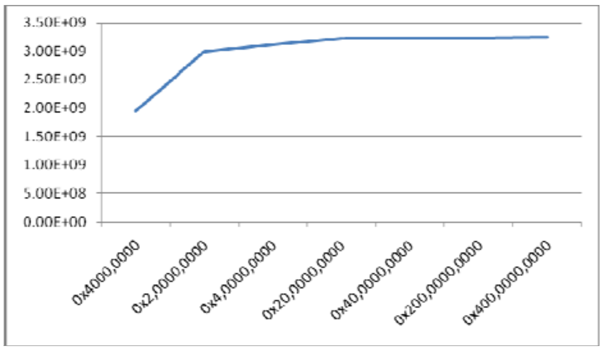

Figure 4. GASPRNG LFG Performance (number of RNs generated versus generation rate)

Fig. 3 and Fig. 4 present the GASPRNG LFG and LCG convergence rate. The data is generated on Fermi with 120 blocks, each block has 128 threads. The x-coordinate is the number of random numbers generated, the y-coordinate is the generation rate (RN/sec).

Table 3 shows the runtime of a Monte-Carlo method for calculating Pi. The GPU implementation generates and consumes the random numbers on the GPU. The CPU implementation produces random numbers using SPRNG and consumes them on the CPU. In this experiment, the GPU device is the Fermi 480 GTX. For a number of sample points above $8\times10^{10}$, the results indicates that the LFG on GPU has 85x speedup comparing with a single CPU core.

TABLE III. PERFORMANCE OF MONTE-CARLO PI APPLICATION

| Number of Random Samples N | CPU-LFG $\pi$-estimator (second) | GPU-LFG $\pi$-estimator (second) | GPU-LFG $\pi$-estimator results |
|---|---|---|---|
| $8\times10^{8}$ | $2.4\times10$ | 0.5 | 3.141678 |
| $8\times10^{9}$ | $2.4\times10^{2}$ | 3.0 | 3.141577 |
| $8\times10^{10}$ | $2.4\times10^{3}$ | $2.8\times10$ | 3.141588 |
| $8\times10^{11}$ | $2.4\times10^{4}$ | $2.8\times10^{2}$ | 3.141591 |

## IV. CONCLUSION

In this paper, we implemented the GPU version of LFG and LCG random number generators. Based on SPRNG2.0 library, this implementation aims at providing efficient random number tools for GPU programs. It is composed of pure device functions with simple interface for each GPU threads. The framework is layered to separate users' logical understanding from the exact memory usage inside the library; it improves the library's performance. By distributing the data properly to different levels of GPU storage resources, the performance is further improved. Experiments results show that both these two implementations provide obvious speedup compared to CPU and FPGA.

Future work includes implementing the other 4 types of random number generators in SPRNG 2.0. In addition, the interface will be unified to make the library easier to use. This library is also going to be tested and improved to fit into computationally intensive simulations on large GPU clusters.

## REFERENCES

[1] Scalable Parallel Pseudo Random Number Generators Library, http://sprng.fsu.edu

[2] J. Lee, Y. Bi, G. D. Peterson, R. J. Hinde, and R. J. Harrison, "HASPRNG: hardware accelerated scalable parallel random number generators," Computer Physics Communications, vol. 180, no. 12, pp. 2574-2581, 2009

[3] Vadim Demchik, "Pseudo-random number generators for Monte Carlo simulations on Graphics Processing Units", http://arxiv.org/pdf/1003.1898v1

[4] W. B. Langdon, "A Fast High Quality Pseudo Random Number Generator for nVidia CUDA", Genetic And Evolutinary Conputation Confernece, Proceedings of the 11th Annual Conference Companion on Genetic and Evolutinary Computation Conference, pp 2511-2514, 2009

[5] M. Sussman, W. Crutchfield, M. Papakipos, "Pseudorandom number generation on GPU", SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware, Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 87-94, 2006

[6] M. Mascagni, D. Ceperley, A. Srinivasan, "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation," *ACM Transactions on Mathematical Software* 26 (2000) 436