An On-Demand Fast Parallel Pseudo Random Number Generator with Applications

Dip Sankar Banerjee, Aman Kumar Bahl and Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India – 500 032.
{dipsankar.banerjee@research.,aman.kumar@research.,kkishore@}iiit.ac.in

***Abstract*— The use of manycore architectures and accelerators, such as GPUs, with good programmability has allowed them to be deployed for vital computational work. The ability to use randomness in computation is known to help in several situations. For such computations to be made possible on a general purpose computer, a source of randomness, or in general a pseudo random generator (PRNG), is essential. However, most of the PRNGs currently available on GPUs suffer from some basic drawbacks that we highlight in this paper. It is of high interest therefore to develop a parallel, quality PRNG that also works in an on demand model.**

In this paper we investigate a CPU+GPU hybrid technique to create an efficient PRNG. The basic technique we apply is that of random walks on expander graphs. Unlike existing generators available in the GPU programming environment, our generator can produce random numbers on demand as opposed to a one-time generation. Our approach produces 0.07 GNumbers per second. The quality of our generator is tested with industry standard tests. We also demonstrate two applications of our PRNG. We apply our PRNG to design a list ranking algorithm which demonstrates the on-demand nature of the algorithm and a Monte Carlo simulation which shows the high quality of our generator.

Keywords: **PRNG, on-demand, list ranking, Monte Carlo, GPGPU.**

I. Introduction

Randomness is an essential computing resource for many computations [21], [16], [13]. Hence, investigations into sources of high quality (pseudo) random number generators (PRNGs) are important. In parallel computing, designing parallel random generators is a challenging problem. This problem becomes more significant, as we are witnessing a shift to multicore processors.

In recent years, accelerators such as the IBM Cell SPUs, FPGAs, GPUs, ASICs are studied because of the performance gains they offer in comparison to the CPUs. Amongst them GPUs standout to be highly popular, because of low cost and mass manufacture. GPUs today provide the highest amount of FLOPs per dollar, and the latest models have a theoretical peak of 1 TFLOP having hundreds of cores. It is therefore natural that GPUs have started to occupy a prominent place in parallel computing in recent years [26], [9], [30]. The GPU programming SDK also includes PRNGs such as the CURAND[24], and a Mersenne twister[25] based generator.

Most of the pseudo random number generators based on GPUs however suffer from several drawbacks. For instance, PRNGs on GPUs require the application to pre-generate and store a large batch of random numbers and then use them in the application. Apart from occupying a significant portion of the limited storage on GPUs, this is not a satisfactory solution since the randomness demand of every application cannot be known apriori. It is therefore important that an on-demand pseudo random number generator be available so that each thread running on a GPU can make an API call, such as the `rand()` function in ANSI C [14], to obtain a new pseudo random number as required. Such an on demand generator also does not require as much storage to store the random numbers in the GPU memory. Secondly, another limitation of present generators on the GPUs is that they are not resource efficient. While the generator is working on the GPU, the host to which the GPU is attached, typically a multicore CPU, is computationally idle. This is not a good practice as the computational power of multicore CPUs is also ever increasing.

In this paper, we address these limitations and demonstrate the design and implementation of a fast, efficient, on demand, and high quality PRNG on a platform consisting of CPUs and GPUs. This is evident from the performance of the list ranking and Monte Carlo simulation which we designed using the PRNG.

A. Motivation

From the above discussion, it is clear that existing PRNGs on GPUs suffer from several drawbacks. To motivate our work further, we present four properties that a parallel pseudo random number generator has to satisfy.

- **Scalability:** Scalability suggests that large quantities of random numbers can be generated without any limitations.
- **Quality:** Many cryptographic and security applications solely depend on good sources of random numbers [32], [22]. Hence, the quality of the generator is important.
- **On demand generation:** It must be possible to use the generator without a-priori knowing the quantity of random numbers required. Ideally, a simple API call should produce a new random number without a large overhead.
- **Performance:** The performance aspect suggests that the time spent on generating a random number is as small as possible. One common way to measure this is to study the number of random numbers that can be produced in a unit time.

TABLE I

COMPARISON OF PROPERTIES

PRNG	On-Demand Supply	Scalable	High Quality	Speed Rank
glibc `rand()`	×	√	×	5
CURAND	√	√	×	4
CUDPP	×	×	√	3
M.Twister	×	√	√	2
Hybrid PRNG	√	√	√	1

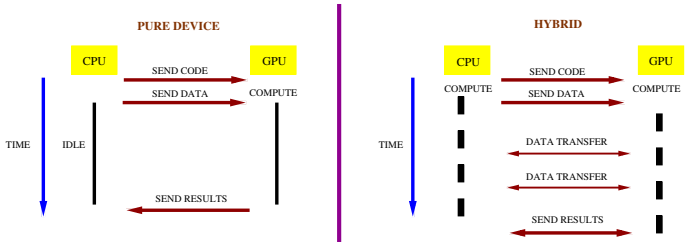

Fig. 1. The case for hybrid computing. The dashed lines of the right figure denotes the interleaving of computation and idle times.

We now give a comparison of the qualities that are possessed by the currently available PRNGs in Table I. The speed ranking in Table I (rank of 1 is fastest), shows the relative time taken by each PRNG to generate a fixed quantity of random numbers.

We now turn our attention to the aspect of resource efficiency of PRNGs on GPUs. We observe that in most GPGPU based computing, the CPU is practically idle in the computation process as illustrated in the left portion of Figure 1. This leads to inefficient resource usage, more so as the computational power of present generation multicore CPUs is on the rise. Hence, to improve performance, we use such a hybrid CPU and GPU system and target full resource utilization as shown in the right portion of Figure 1. Hybrid multicore computing is gaining tremendous research attention of late given that issues such as power and performance dominate parallel computing. Recent works in this direction include [28], [27], [3], to name a few. Arriving at an efficient hybrid PRNG that meets the requirements listed in Table I is a step in that direction.

B. Our Methodology and Results

Our main result of this work is to design a high quality, fast, scalable, and on-demand random number generator. We achieve this by employing random walks on expander graphs. Each thread performing the walk is essentially executing independent of other threads. Therefore, our generator is thread-safe.

To improve the performance of our generator, we employ a hybrid computing platform consisting of a multicore CPU and a GPU. Our generator produces 0.07 GNumbers per second. The results of our generator has been put through rigorous quality testing using test suites such as the DIEHARD battery of tests [18] and the TestU01 [17] suite. Our generator passes most of these tests as reported in Section IV-B.

We also show how to use our PRNG in two applications: list ranking, and a Monte Carlo based photon migration. These applications demonstrate the speed of generation and the quality of the hybrid PRNG respectively. In both these applications, using our PRNG leads to reduced runtime, and improvements in quality.

II. Our Hybrid Platform

Our hybrid platform, see also Figure 2, is a combination of two devices, an Intel i7 980 and an Nvidia Tesla C1060 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU.

The GPU is viewed as a massively multi-threaded architecture containing hundreds of processing elements (*cores*). All of the cores are composed of four stage pipelines. The cores which are also known as *Symmetric Processors* (SPs). Each of the SPs are grouped in an SIMD fashion into a *Symmetric Multiprocessor* (SM). So, all of the SMs execute the same instruction at a particular point of time. The Tesla C1060 has 30 such SMs, which makes for a total of 240 processing cores. All of these SMs are caplable of running millions of threads which are scheduled on each of the cores in groups of 32 threads. These groups of 32 threads are called *warps*. To program the GPU we use the CUDA API Version 3.2 [23].

For the CPU we use a Intel i7 processor which is a multicore processor of the Sandy Bridge family of microporcessors. This processor is the fastest from the in its family with each core running at 3.4 GHz and having a highest throughput of 109 GFLOPS. For programming the CPU, we use OpenMP specification 3.0 and ANSI C [15].

Asynchronous concurrent execution model is offered by all the NVidia GPUs with a compute capability of 1.1 or above. This allows the GPU kernel calls to be non-blocking and allows for host execution to overlap with the GPU kernel computations. *Streams* support asynchronous data transfer while the kernel is executing. Hence, not only computation but also data transfer can be overlapped between the device and the host.

III. Our Random Number Generation Technique

The main idea behind the development of our generator is to use parallel random walks on an expander graph. As each of the random walks on the graphs is entirely independent of each other, any thread of the GPU can make a request for random number(s) at any point of time. The only operation involved is to select a neighbor from the expander graph uniformly at random, perform a walk, and return the destination node as a random number. This random selection of a neighbor can be made by using a few random bits. In the following, we explain our approach in more detail. Implementation details are presented in Section III-B.

A. Expander Graphs

We now define an expander graph and also describe the expander graph we use in our construction. Let $G(V, E)$ be an undirected regular graph of degree d. For a subset of vertices

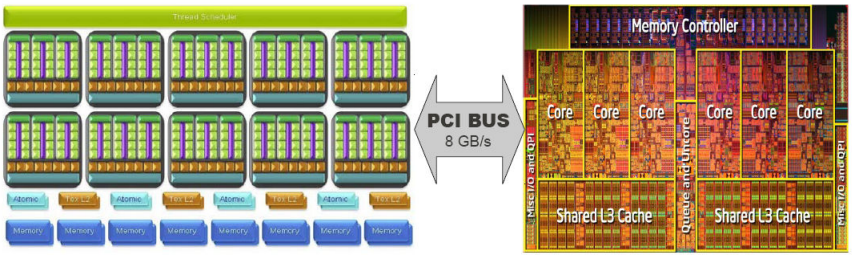

Fig. 2. The GPU-CPU hybrid platform. The pictures are obtained from $http://www.hothardware.com$.

$U \subseteq V$, let us denote by $(U, \overline{U})$, with $\overline{U} = V \setminus U$, the set of edges that have exactly one endpoint in U and another endpoint in $\overline{U}$. The edge expansion of G, denoted $\alpha(G)$, is defined as:

$$\alpha(G) = \min_{\substack{U \subseteq V; \\ |U| \leq |V|/2}} \frac{|(U, \overline{U})|}{|U|}.$$

A family of graphs $\mathcal{G} = \{G_1, G_2, \cdots\}$ is called an edge expander family if there exists a constant c so that for every $G \in \mathcal{G}$, $\alpha(G) \geq c$. In our construction of a PRNG, we have made use of explicit definitions of an expander graph due to Gabber-Galil [8]. Here we consider the graph G as a bipartite graph with two independent set of nodes X and Y. For a given integer m, with $n = 2m^2$, we can assign unique labels of the form $(a, b) \in \mathbb{Z}_m \times \mathbb{Z}_m$ to each of the vertices in X and Y. A Gabber-Galil expander on n vertices is defined as follows [8]. The vertices of the graph are tuples of the from (x, y) for $x, y \in \mathbb{Z}_m$. The neighbors of a vertex (x, y) in X can be found in Y by these labels : $(x, y), (x, 2x + y), (x, 2x + y + 1), (x, 2x+y+2), (x+2y, y), (x+2y+1, y)$, and $(x+2y+2, y)$. All the above calculations are done modulo m. The expansion of the graph is shown to be $\alpha(G) = (2 - \sqrt{3})/2$ (cf. [8]).

It has been shown [11] that random walks on expander graphs have a rapid mixing property so that the position of the walk after a certain steps is close to the stationary distribution of the underlying Markov process.

B. Implementation Details

We initialize a 7-regular Gabber-Galil expander graph G of $n = 2^{65}$ nodes. With $n = 2^{65}$, each vertex of the Gabber-Galil expander graph of the form (x, y) can be represented using 64 bits with x and y being 32 bit each. The random numbers generated by our construction are the 64 bit vertex ids of the expander graph G.

We initialize our generator by having each thread start at a random vertex of G and performing an initial walk of length 64. To select the starting position, we need 64 random bits for each thread. In our implementation, we make use of the CPU for this as follows. These random bits are generated on the CPU and are supplied to the GPU. As current GPUs support asynchronous memory transfers, we can pipeline the execution and transfer. The details of this process are explained in Algorithm 1.

Algorithm 1 InitializeGenerator(G, l, bin)

Input: A $7-regular$ expander graph G of n nodes, the length of walk l and some random bits bin
Output: An initialized G

1: `CPU` : : Generate a random binary stream bin
2: `CPU`→ `GPU` : : Transfer bin asynchronously
3: `GPU` : : **for** each node u in G do in parallel
4: `GPU` : : **for** i = 0 to l
5: $b(u)$ = $(int)(bin(t)$ & $(111 << (i * 3))$
{t is the Thread ID}
6: $v = f(u, b(u))$ { $f(u, b(u))$ gives the $b(u)^{th}$
7: neighbor of u}
8: $u := v$
9: **endfor**
10: **endfor**

In Algorithm 1, we use labels such as `CPU`, `GPU`, and `CPU`→ `GPU`. These represents the executions that are happening at the individual processors at any point of time. We employ these labels in order to distinguish between the parallel computations which are being carried out in each of these devices. The label `CPU`→ `GPU` represents the asynchronous data transfer from the CPU to the GPU. The function $f(u, k)$ which is used in line 7 returns the k^{th} neighbor of u according to the definition of the Gabber-Galil expander graph.

In Algorithm 1, we first initialize a graph such that each of the node can represent a unique 64 bit number. For the initialization phase, we generate some random numbers using the CPU rand() utility which in turn calls the LCG present in the `glibc` library. As each of the vertices have 7 neighbors to identify from randomly, it requires only 3 bits to do so. The CPU streams random bits to the GPU as long as the kernel is executing, and the GPU has a constant supply of random bits to perform the walk. The overlap between the CPU and GPU computation will be explained in Section IV.

Once each thread completes a random walk of length 64, each thread is now ready to generate random numbers. To generate each random number, each thread has to essentially perform another random walk. As these walks are completely independent, our approach allows for massive parallelism. This can be done on demand also, unlike other GPU-based generators such as Mersenne Twister [19].

Let a walk be presently at vertex v. To continue the walk, we need to select a neighbor of v in G uniformly at random.

This therefore requires few random bits. As earlier, we use the CPU to generate these random bits and supply them to the GPU in an asynchronous manner. At the end of the walk, each thread outputs a 64 bit random number. The details of this approach are explained in Algorithm 2. The asynchronous transfer in Line 1 is same as that of Algorithm 1.

Algorithm 2 GetNextRand(G, bin)

Input: The initialized graph G and some random bits bin
Output: A new random number R

1: CPU$\rightarrow$ GPU :: Generate and transfer bin asynchronously
2: GPU :: **for** i = 0 to l
3: $b(u) = (int)(bin(t)\ \&\ (111 << (i * 3))$
{here t is the Thread ID}
4: $v = f(u, b(u))$ $\{f(u, b(u))$ gives the $b(u)^{th}$
5: neighbor of $u\}$
6: $u := v$
7: **endfor**
8: Return v

Threads in an application that requires randomness can call the *GetNextRand()* routine to obtain random numbers. The *GetNextRand()* routine is described in Algorithm 2. The application does not require to pre-specify the number of bits before executing its kernels. The application has to only initialize the random number generator as described in Algorithm 1 before using *GetNextRand()*.

IV. Experimental Results

Our experimental platform is described in Section II. In Section IV-A we study the speed and quality of our generator compared to existing generators. In Section IV-B, we study the quality of the random numbers produced by our generator. In all cases, the experiments are conducted over repeated trails and the average values are reported.

A. Performance Analysis

We now compare the performance of our generator to presently available GPU based generators. Some of the fastest generators on the GPU are the Mersenne Twister [19], and the CURAND utility [24]. We have therefore compared against these generators. In this experiment, we study the time taken to produce a random stream of N numbers for a given N ranging from 5 M to 1000 M. The resulting run times are plotted in Figure 3. In Figure 3, the label "Hybrid Timing" refers to the timings obtained by our generator. The label "Mersenne Twister" refers to the generator based on [20]. These timings were obtained by running the example code which is provided by NVidia with the CUDA toolkit. The label "CURAND" refers to the generator based on [24] from the CUDA library. CURAND is also an on-demand variant which can generate numbers on the fly as requested by an application when it is used in its Device API mode. We have considered the Device API for comparison since an on demand supply of random numbers is supported only on this mode. As can be seen from Figure 3, the hybrid generator outperforms both the Mersenne Twister based generator and the CURAND generator by a factor of 2 in most cases.

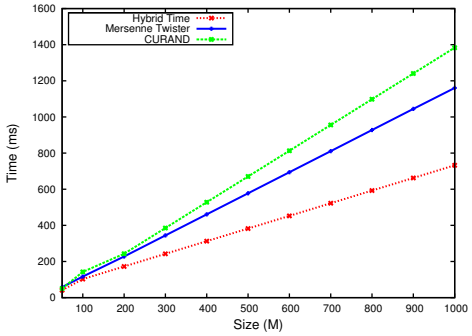

Fig. 3. Timings across several list sizes

Another aspect of a hybrid algorithm to study is the overall resource utilization. In our program, the main work units are (i) an initial source of random bits, FEED, (ii) the transfer time required to transfer these initial source of random bits, TRANSFER, and (iii) the generation of random numbers using random walks on an expander, GENERATE. We map the FEED work unit onto the CPU and the GENERATE work unit on the GPU. This is a natural mapping as GENERATE is massively parallel and hence can be done on the GPU. With this mapping, TRANSFER corresponds to transferring data between the CPU and the GPU using the PCI Express link. In Figure 4, the time taken for each of the above work units is indicated on the arrows. The arrow with label 6.2 ns corresponds to the TRANSFER work unit. As can be seen, the CPU is almost never idle, and the GPU is idle for about 20% during each iteration. The timings shown are for a batch size of 100 (see also Figure 5) where batch size, S, is defined as the number of random numbers each thread is generating. For instance, if N random numbers are to be generated, then with a block size of S, each of the N/S threads generate S random numbers each.

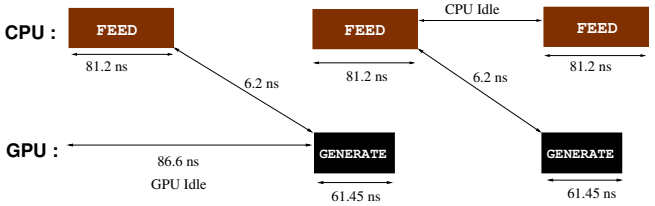

Fig. 4. The overlapped execution of the work units.

In Figure 5, we study the variation of the timing with the block size. As we see, the timing is minimum at a work load of around 100 numbers per thread. This suggests that when the GPU threads are high but the work load per thread is low then the utilization of the system is low and the CPU stays idle for most of the time. Beyond 100 numbers per thread, the utilization is high but the CPU gets overloaded and the GPU starts to wait for CPU to transfer random bits. So, the time taken increases.

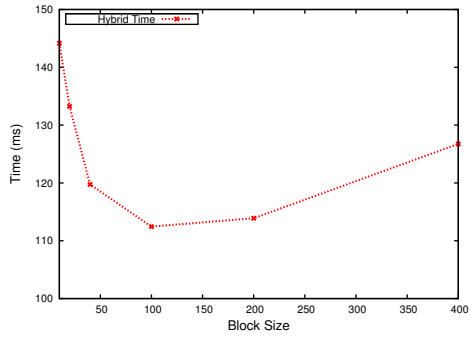

Fig. 5. Variation of timing with block size.

Comparison with `rand()`: Our hybrid generator can also work on other multicore architectures with minor programmatic changes. This is showcased by developing our generator for a multicore CPU alone. In this case, each core of the CPU runs threads which perform random walks on the implicitly defined expander graph.

On the CPU described in Section II, we implemented our generator using the OpenMP specification 3.0 library. We then compare the time taken by our generator to that of the standard `glibc rand()` which is provided by the Fedora 14 Linux distribution. The result of this experiment is shown in Figure 6. The label "CPU Rand Time" refers to the time taken by `rand()` to generate the required quantity of random numbers. As can be seen, our generator scales up well compared to `rand()`. Further, our algorithm is thread safe.

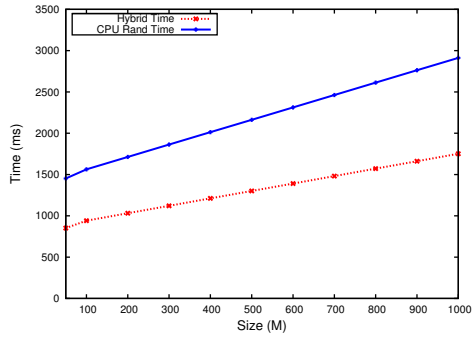

Fig. 6. The time comparison when the algorithm runs on the CPU vs CPU `rand()`

B. Quality

For studying the quality of our PRNG, we use several industry standard statistical tests. For instance, we have taken the *DIEHARD* battery of tests based on statistical testing methods as proposed in [6]. This suite consists of 15 different statistical tests. Marsaglia [18] implemented these tests so that these tests can be run for any PRNG easily. Each of the tests produces a p-value which is a measure of the uniformity in distribution of these numbers. Each of the p-values are further verified using Kolmogorov-Smirnov(KS) Test [6]. The KS test gives a measure of the uniformity of the numbers that are generated and how well they pass the *DIEHARD* tests.

TABLE II
QUALITY RESULTS OF DIFFERENT ALGORITHMS

Algorithm	*DIEHARD* Tests Passed	KS-Test D
Hybrid PRNG	15/15	0.167
CUDPP_RAND	15/15	0.202
M. Twister	15/15	0.166
CURAND	8/15	0.534
`glibc rand()`	6/15	0.621

TABLE III
TESTU01 TEST RESULTS

PRNG	Test Suite	Tests Passed
CURAND	SmallCrush	15/15
	Crush	14/15
	BigCrush	13/15
M.Twister	SmallCrush	15/15
	Crush	13/15
	BigCrush	13/15
Hybrid PRNG	SmallCrush	15/15
	Crush	14/15
	BigCrush	13/15

The *DIEHARD* tests state that if the values generated by a PRNG are truly random, then the values are indistinguishable from a set of uniformly distributed values in the range of $[0, 1)$. The test statistic p should lie between 0.01 and 0.99 to pass the test. After the *DIEHARD* tests are completed, we also looked at the KS test against a set of uniformly generated numbers. This test tries to measure the maximal deviation between two curves drawn on a cumulative distribution function. A low value indicates the lower deviation from a set of uniformly distributed values. The test statistic D gives a measure of how well the generator performs in comparison to the other generators. As we see from Table II, the KS test result of our algorithm is comparable to that of Mersenne Twister and better than that of CURAND.

Apart from the *DIEHARD* suite, we also use the *TestU01* suite. L'Ecuyer and Simard implemented the *TestU01* software library [17] which is a more advanced test than the *DIEHARD* tests. This suite contains three different batteries: SmallCrush, Crush and BigCrush. These three tests are all in the increasing order of quality. The results obtained from the tests are tabulated in table III.

As we see from Table III, our generator passes all the tests in the SmallCrush battery. It is able to pass only fourteen and thirteen tests in Crush and BigCrush tests respectively. This is comparable to other presently available PRNGs on GPUs such as CURAND and the Mersenne Twister based generator.

C. Discussion

One of the interesting aspects of our implementation is the use of CPU generated random numbers to aid the GPU based generator. This can be justified by the following arguments regarding quality and performance.

Our generator based on random walks on expander graphs, produces quality pseudo random numbers as can be seen from the results of Section IV-B. The quality of our generator is better than `glibc rand()`, and is comparable or better

compared to existing PRNGs on GPUs. Hence, our technique can be seen an improving the quality of a naive random number generator. Further, this increase in quality is obtained by using a little amount of initial randomness. Our technique has connections to other works on expander graphs such as probability amplification [21].

To perform a random walk, one has to select a neighbor of the current position of the walk uniformly at random. This requires some source of randomness. Using a GPU based generator for this purpose is not a good solution as it would still keep the CPU idle. Further, there are no fast, on-demand GPU based generators. Hence, we make of the CPU to provide us with these few random bits that are used by the GPU. This is also helping us improve the performance of our generator. Our generator is the fastest known PRNG on GPUs as can be seen from Figure 6.

In our framework, we notice that the GPU is idle for a small fraction of the time. We are presently using the ANSI C based `rand()` for this purpose. This is mainly due to the fact that there is no fast randomness generator on a multicore CPU. Our generator, working on a multicore CPU can be used in the place of `rand()`. As Figure 6 shows, this would help us in never keeping the GPU idle.

V. Application I : List Ranking

The problem of list ranking is to find the distance of any node in a list of n nodes from either end of the list. List ranking was first identified by Wyllie [31] as an important primitive of parallel computing. Considering the importance of the problem, there have been many recent solutions for list ranking on modern architectures such as the Cell BE [2], GPU [30], and a more recent CPU+GPU hybrid solution [3].

We follow the three step method from [3] where the first step is to reduce the size of the original list to a very small one. This is done by repeatedly identifying a large fractional independent set of nodes using techniques from fractional independent sets [12]. In the second step, the list of the remaining nodes is ranked using the algorithm of Hellman and JaJa [10], as is done in [3]. Finally, the nodes removed in step 1 are re-inserted to get the final ranks of all nodes in the list. A fractional independent set of a graph G is an independent subset U of low degree nodes of G such that $|U|$ is at least a constant fraction of $|V|$.

For a linked list L of n nodes, to compute a fractional independent set (FIS) in parallel, we proceed as follows. Each node v picks a bit, $b(v) \in \{0, 1\}$, uniformly at random and independent of other nodes. Then, we say that a node v belongs to the FIS if $b(v) = 1$ and neither the predecessor nor the successor of v also chose 1. It can be seen from relatively simple arguments (cf. [12]) that with high probability, the FIS constructed above has at least n/c nodes for $c \geq 24$. Applying this repeatedly results in a list of $n/\log n$ nodes. Our complete algorithm to identify nodes to be removed so that only $n/\log n$ nodes remain is shown in Algorithm 3. The only difference between Algorithm 3 and the Phase I of the list ranking algorithm in [3] is in the generation of random numbers. However, as one does not know exactly how many nodes are removed in each iteration of Line 3 of Algorithm 3, a PRNG that can work with efficiently on demand is needed. This application therefore serves to illustrate the on demand property of our PRNG.

Algorithm 3 $ReduceList(\ L, G)$

Input: A list L of size n and a 7 regular graph G
Output: A sublist of $n/\log n$ nodes

1: ***Phase I*** : *Pre-processing*
2: CPU :: Initialize 7-regular graph G by Algorithm 1
3: **for** r iterations in parallel **do**
4: `CPU` **::** Generate and transfer asynchronously random binary stream bin
5: `GPU` **::** **for** each node u in the list do in parallel
6: $temp$=getNextRand(bin)
7: Let $b(u)$ be the bit choice of node u from $temp$
8: **if** $b(u) = 1$ and $b(pred(u)) = 0$ and
9: $b(succ(u)) = 0$ **then**
10: Remove node u with proper book-keeping
11: **endif**
12: **endfor**
13: **end for**

We now describe Algorithm 3 in more detail. We first initialize a 7-regular Gabber-Galil expander graph. This graph is then used to generate random numbers as required. As can be seen in Line 6 of Algorithm 3, each thread can call `getNextRand()` to obtain a new random number that will be used by this thread. Since this operation is done only for those nodes in L that are not removed in previous iterations, the number of such calls is not known apriori. One can only say that the number of nodes in L reduces by a constant factor in each iteration. Further, the GPU compute time is overlapped with an asynchronous transfer from CPU.

The ability to efficiently produce random numbers on demand in our current approach offers a big advantage to our implementation compared to the hybrid implementation of [3]. In [3], the CPU generates a quantity of random numbers that is predetermined to be an upper bound on the number of nodes remaining in the list at each iteration. We will show shortly in our results that an on demand generation reduces the runtime by 40%.

Once we have a list of size $n/\log n$, we continue with the approach of Phases II and III of the algorithm in [3].

A. Results

The experimental platform we use is described in Section II. We store the initial list in a array and pre-compute the predecessor and successor array before we start the experiments. We use random lists for experimentation as the random lists are the most difficult to rank due to their irregular nature of memory access patterns.

In our experiments we vary the list size to upto 128 million elements and compare the results with other relevant works [3]. The results of [3] are presently the fastest known solution for the problem under study. We also compare our results with two other techniques where the glibc random number generator

used in [3] is replaced with a generator based on Mersenne Twister. This is referred in Figure 7 as "Pure GPU MT" and is hence a pure GPU implementation without any involvement of the CPU. To summarize, our generator outperforms the fastest running algorithm by almost 40%.

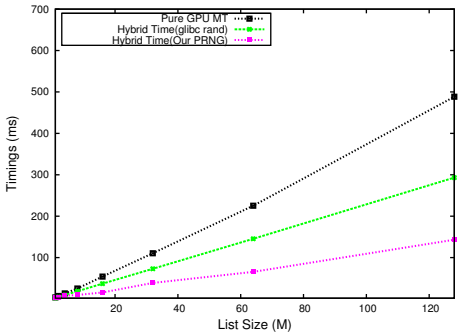

Fig. 7. The timing comparison with the other algorithms.

It can be seen from Figure 7, that the improvement in runtime for reducing the list to a size of $n/\log n$ is due to two factors. Firstly, using hybrid algorithms helps by bringing also the CPUs into the computation process. Secondly, an efficient on demand PRNG helps in reducing the runtime further as can be seen from the plots labeled Hybrid-glibc and Hyrbid-PRNG. Given that Phases II and III of list ranking take only 20% of the overall time, using Algorithm 3 for Phase I of list ranking as described in [3] would result in an improvement of 50% in the runtime of list ranking over various list sizes ranging up to 128 M nodes.

VI. Application II : Hybrid Monte Carlo

Monte Carlo methods are used in several areas of science to simulate complex processes, to validate simpler processes, and to evaluate data. In Monte Carlo (MC) methods, a stochastic model is constructed in which the expected value of a certain random variable is equivalent to the physical quantity to be determined. The expected value of this random variable is then determined by the average of many independent samples representing the random variable. These independent samples are constructed by the use of random numbers following the distribution of that random variable.

Photon migration, i.e., light propagation in a random media, is an area where MC simulations are proven as a gold standard [4]. In this method, several photons are launched with their position and direction initialized to either zeros (for some pencil beam initialized at the origin) or some random numbers. A variance model is used for simulation, where the absorption of photons is simulated by reducing weights and not discrete termination. At every step a photon takes, a fraction of its weight is absorbed, and then photon packet is scattered. The new direction and weight of photon are updated. After several such steps if the remaining weight of a photon is below a certain threshold, the photon is terminated.

To summarize, the rules of photon migration can be expressed as step-size of photon movement between sites of photon tissue interaction, and the angles of deflection in the photons trajectory in case of a scattering event. The method is statistical in nature and we need to study the propagation of a large number of photons. Due to this, the method requires a large amount of computation time. An earlier work of photon migration on GPU [1] does this simulation across multiple layers of absorption. This work uses a multiply with carry (MWC) based RNG to initialize the weights of the photons. We use the same implementation to show that the simulation can happen in a much better way when it is plugged in with the hybrid PRNG.

A. Our solution

We try to solve the problem of photon migration using the hybrid PRNG which has been explained in the previous sections. The hybrid PRNG supply the random numbers which are used to initialize all the simulation kernels used. There are specifically three simulation kernels which are used to simulate the three different layers of the MC simulation. Our hybrid PRNG supplies the values which are required at each layer.

The PRNG is the heart of the multi-layer simulation of photons. Each of the initial weights of the photons must be set at a random value which should be generated independent of each other in order to minimize the number of weight clashes that might happen at the different layers. These clashes correspond to certain atomic operations. The better quality of random numbers ensure that there will be lesser clashes and hence lesser serialization will occur. The hybrid PRNG works completely in a data parallel way which ensures that whenever a call for the PRNG is made, it supplies a random number irrespective of that at the other thread calls. The algorithm proceeds in an iterative fashion where a fixed quantity of photon packets are processed in each iteration. This provides an ideal setting for the PRNG to work as the GPU kernel execution times can be used to supply the graph with fresh random bits which shall be used for random walks in subsequent iterations. Also, the high quality of the random numbers supplied allows for more number of photons to be simulated at each layer.

Algorithm 4 $MCPhotonMigration(P, LayerParams)$

Input: Number of photons P and parameters of layers
Output: Reflectance parameters and absorbed fraction

1: Initialize 7-regular graph G by Algorithm 1
2: `CPU` **::** Generate and transfer asynchronously random binary stream bin.
3: `GPU` **::** Launch a photon after initializing weights with getnextRand()
4: **While** the photon survives
5: `GPU` **::** Remove the absorbed weight
6: `GPU` **::** Scatter the photon
7: `CPU` **::** Generate and transfer new bin in an
8: overlapping manner
9: `GPU` **::** $noOfUsedPhotons+ = 1$
10: **If** $noOfUsedPhotons \leq maxNoOfPhotons$
11: Goto step 3.
12: **end while**

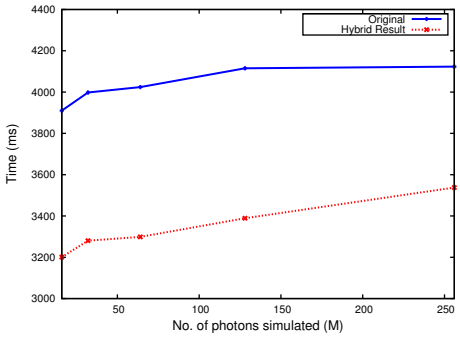

Fig. 8. Variation of timing with number of photons simulated

In Algorithm 4, we see the pseudo code of the algorithm applied. The PRNG is used in the algorithm by using an overlap between the CPU and the GPU for generating and transferring the required random bits. As the generation is not related to the GPU kernels, we have optimally made the CPU work towards re-populating the *bin* array while the GPU is busy in steps 5 and 6. The result of the experimentation can be observed from Figure 8. The number of photons is varied from 1 M to 256 M. The Y-axis shows the time taken by our method, labeled 'HybridResult', and the time taken by the implementation of [1], labeled 'Original'. We can attribute this result to the following reasons:

- **Reduced memory transaction overhead:** As the PRNG works in an hybrid fashion, the actual memory overhead of accessing the global memory for getting random numbers is minimized. This is due to the reason that our implementation does not use any extra space for storing the random numbers unlike [1]. In our model, random numbers are generated on the fly. Hence a certain speedup is obtained.
- **Lesser number of clashes in atomic operations:** The quality of the hybrid PRNG has been already discussed in Section IV-B. This is another advantage which the hybrid PRNG offers to the MC simulation. As all the threads independently supply high quality random numbers to initialize the weight of photons, the number of clashes happening is subsequent layers is reduced. By clashes, we mean the behavior of two photons as a single one due to having the same weight. As a higher amount of photons have unique weights, they are independently simulated. The weights of these photons quickly fall below the threshold and are terminated. As a result, if we are aiming to simulate a fixed number of photons, then all of them are simulated at a much lesser amount of time. This contributes towards an overall speedup of around 20%.

VII. Related Work

One of the early implementation of PRNGs for GPUs is the Mersenne Twister (MT) first proposed in [20] and later extended in [19]. Both these implementations however require that the quantity of random numbers to be pre-specified.

In [29], authors have given a source of randomness necessary for graphics applications based on the MD5 algorithm proposed by Rivest [5]. However, one major drawback of this is that CUDPP Rand usually do not scale to very large requirements.

Monte Carlo simulations require a good source of random numbers and has been widely researched in the GPGPU community. In [7], the author gives an analysis of several random numbers generation techniques which are suitable for Monte Carlo simulations.

VIII. Conclusions

In this work, we have presented an efficient pseudo random generator for GPUs. Our generator satisfies all the conditions that are deemed important. Our generator also combines the computational abilities of multicore CPUs and GPUs in a clever way to improve resource utilization. In future, we wish to study cryptographic applications that often require high quality random numbers.

References

[1] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration." *Journal of Biomedical Optics*, vol. 13, no. 6, 2008.
[2] D. A. Bader, V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking," in *Proc. of IEEE IPDPS*, 2007, pp. 1–10.
[3] D. S. Banerjee and K. Kothapalli, "Hybrid multicore algorithms for list ranking and graph connected components," in *18th Annual International Conference on High Performance Computing (HiPC)*, 2011.
[4] D. Boas, J. Culver, J. Stott, and A. Dunn, "Three dimensional monte carlo code for photon migration through complex heterogeneous media including the adult human head," *Opt. Express*, vol. 10, pp. 159–170, Feb 2002.
[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.
[6] R. B. D'agostino and M. S. Stephen, *Goodness-of-Fit Techniques*, R. B. D'agostino and M. S. Stephen, Eds. Marcel Dekker, 1986.
[7] V. Demchik, "Pseudo-random number generators for monte carlo simulations on graphics processing units," http://arxiv.org/abs/1003.1898v1, 2010.
[8] O. Gabber and Z. Galil, "Explicit constructions of linear size superconcentrators," in *in IEEE FOCS*, 1979, pp. 364–370.
[9] N. Govindaraju and D. Manocha, "Cache-efficient Numerical Algorithms using Graphics Hardware," *Parallel Computing*, vol. 33, no. 10-11, pp. 663–684, 2007.
[10] D. R. Helman and J. JàJà, "Designing Practical Efficient Algorithms for Symmetric Multiprocessors," in *Proc. ALENEX*, 1999, pp. 37–56.
[11] S. Hoory, N. Linial, and A. Wigderson, "Expander graphs and their applications," *Bull. Amer. Math. Soc. (N.S*, vol. 43, pp. 439–561, 2006.
[12] J. Jaja, *An Introduction To Parallel Algorithms*. Addison-Wesley, 2004.
[13] H. J. Karloff and P. Raghavan, "Randomized algorithms and pseudorandom numbers," *J. ACM*, vol. 40, pp. 454–476, July 1993.
[14] B. Kernighan and D. Ritchie, *The C Programming Language*. Prentice Hall, 1978.
[15] B. W. Kernighan and D. M. Ritchie, *The C Programming Language Second Edition*. Prentice Hall, 1988.
[16] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*, 1998.
[17] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, 2007.
[18] G. Marsaglia, "The marsaglia random number cdrom including the diehard battery of tests of randomness." http://www.stat.fsu.edu/pub/diehard/, 1995.
[19] M. Matsumoto and T. Nishimura, "Dynamic Creation of Pseudorandom Number Generators," in *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, Jun. 1998, pp. 56–69.
[20] ——, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, January 1998.

[21] R. Motwani and P. Raghavan, *Randomized algorithms*. New York, NY, USA: Cambridge University Press, 1995.
[22] J. V. Neumann, "Various techniques used in connection with random digits," *Applied Mathematics Serires*, vol. 12, pp. 36–38, 1951.
[23] Nvidia Corporation, "Cuda: Compute Unified Device Architecture programming guide," Technical report, Nvidia, Tech. Rep., 2007.
[24] NVIDIA Corporation, "CUDA CURAND Library," NVIDIA Corporation, Santa Clara, CA, USA, Aug. 2010.
[25] V. Podlozhnyuk, "Parallel mersenne twister," Nvidia, Tech. Rep., 2007.
[26] M. S. Rehman, K. Kothapalli, and P. J. Narayanan, "Fast and Scalable List Ranking on the GPU," in *Proc. of ACM ICS*, 2009.
[27] S. Tomov, H. Ltaief, R. Nath, and J. Dongarra, "Faster, cheaper, better - a hybridization methodology to develop linear algebra software for gpus," in *GPU Computing Gems*, 2010.
[28] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense liner algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 12, pp. 10–16, Dec. 2009.
[29] S. Tzeng and L.-Y. Wei, "Parallel white noise generation on a gpu via cryptographic hash," in *Proc. of I3D*, 2008, pp. 79–87.
[30] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded gpus using cuda," in *Proc. of IPDPS*, April 2010.
[31] J. C. Wyllie, "The complexity of parallel computations," Ph.D. dissertation, Cornell University, Ithaca, NY, 1979.
[32] A. Young and M. Yung, *Malicious Cryptography: Exposing Cryptovirology*. John Wiley & Sons, 2004.