Efficient pseudo-random number generation for monte-carlo simulations using graphic processors

Siddhant Mohanty[1,2], A K Mohanty[1,3] and F Carminati[1]

[1] CERN, CH-1211, Geneve 23, Switzerland
[2] Department of Computer Science, VNIT, Nagpur, India 440010
[3] Nuclear Physics Division, Bhabha Atomic Research Centre, Mumbai, India 400085

E-mail: siddhant9892@gmail.com, ajit@cern.ch, Federico.Carminati@cern.ch

Abstract. A hybrid approach based on the combination of three Tausworthe generators and one linear congruential generator for pseudo random number generation for GPU programing as suggested in NVIDIA-CUDA library has been used for MONTE-CARLO sampling. On each GPU thread, a random seed is generated on fly in a simple way using the quick and dirty algorithm where mod operation is not performed explicitly due to unsigned integer overflow. Using this hybrid generator, multivariate correlated sampling based on alias technique has been carried out using both CUDA and OpenCL languages.

1. Introduction

The future of high power computing is evolving towards the efficient use of highly parallel computing environment. The class of devices that has been designed having parallelism features in mind is the Graphics Processing Units (GPU) which are highly parallel, multi threaded computing devices. One application where the use of massive parallelism comes instinctively is Monte-Carlo simulations where a large number of independent events have to be simulated. At the core of the Monte-Carlo simulation lies the random number generators. For GPU programming, the random number generator should have (a) good statistical properties (b) high computational speed (c) low memory use, and (d) a large period . The most commonly used Mersenne Twister generator [1] has very good statistical properties with a long period of 2^{19937}, but not suitable for implementation in the GPU as it has a large state that must be updated serially. Each GPU thread must have an individual state in global RAM and requires multiple access per generator. The relatively large number of computation per generated number makes the generator too slow for GPU programming except in cases where the ultimate in quality is needed. In this paper, we have used a hybrid approach as used in NVIDIA-CUDA library [2]. The suggestion is to use a combination of three Tausworthe generators along with a simple Linear Congruential Generator (LCG) where the mod operation is not performed explicitly. The period of these combinations is quite high 2^{121} and has good statistical properties as the defects of one generator should hide those of the other. This hybrid generator requires four random seeds which can either be supplied using a CPU-side random number generator or through a simple quick and dirty type LCG generator which can be implemented on each GPU thread on fly. We have carried out alias Monte-Carlo sampling [3] from a two dimensional correlated distribution using this hybrid generator where each GPU thread is used to generate random variable in parallel.

This would mean each thread needs to be provided a random seed independently. In the present work, we have implemented alias sampling with NVIDIA GeForce GTX 480 GPU card using both CUDA and OpenCL kernels. It is noticed that the kernel execution in both cases is about 1000 times faster as compared to the CPU whereas the total code execution is only 10 times faster. This is due to the fact that memory copy from host to device or vice-versa is very slow. Therefore we try to minimize memory access time and implement a simple scheme to generate random seed per thread on fly from the formula $seed = 1099087573 * id$ where id is the thread index. This is known as quick and dirty LCG which has a period of 2^{32} and mod operation is not explicitly needed due to overflow of unsigned integer. It is shown that this hybrid generator which takes seed on fly is quite fast, reproduces the statistical properties reasonably well and can easily be implemented on each thread of GPU as well as CPU in an efficient way.

2. Random number generation:A review

A true random number generator provides truly unpredictable numbers which is difficult to achieve through computational implementation. The most common type of random number generators used are pseudo random in character as the implementation uses a deterministic software. However it is desirable to have the output of the generator practically indistinguishable and should not exhibit any correlations or patterns. Every deterministic generator must eventually loop but the goal is to make the loop period as long as possible. Poor generator quality can ruin the results of Monte-Carlo applications. In the following, we briefly mention various uniform PRNGS(Pseudo Random Number Generators) that are available for implementation [4].

It may be mentioned here that ROOT package [5] provides various classes for random number generators based on different algorithms. The TRandom1 class is based on the Ranlux engine originally developed by M. Lusher and implemented in Fortran by F. James [7]. The TRandom2 is based on the maximally equidistributed combined Tausworthe generator by L'Ecuyer [6] while the TRandom3 uses the Mersenne Twister algorithm [1].

2.1. Linear congruential generator

A Linear Congruential Generator (LCG) represents one of the oldest pseudo random number generator algorithms. It is a classical generator which uses a transition function of the form

$$X_{n+1} = (a\ X_n + c)\ mod\ m \tag{1}$$

The maximum period of the generator is m, which means that in 32-bit integer, the period can be at most 2^{32}. It is expected that c is relatively prime to m and $(a-1)$ is a multiple of p for every prime factor p of m. If $c = 0$, the generator is called multiplicative LCG. LCGs are fast and require minimal memory (typically 32 or 64 bit) to retain state. This makes them valuable for simulating multiple independent streams. Due to their statistical flaws, LCGs should not be used for applications where high-quality randomness is critical. For example it is not suitable for a Monte-Carlo simulation because of the serial correlation. A further problem of LCG is that the lower order bits of the generated sequence have a far shorter period than the sequence as a whole if m is set to a power of 2.

2.2. Multiplicative Recursive Generator

A derivative of the LCG is the multiple recursive generator (MRG), which additively combines two or more generators. If n generators with periods $m_1, m_2, ..., m_n$ are combined, then the resulting period is their LCM, thus the period can be at most $m_1\ m_2\ .....m_n$. These generators provide both good statistical quality and long periods, but the relatively prime moduli require complex algorithms using 32-bit multiplications and divisions, so they are not suitable for current GPU programing.

2.3. Mersenne Twister Generator
Mersenne Twister is one of the most respected methods for random number generation in software and has a period of 2^{19937} and provides a very high quality pseudo random numbers [1]. Even though it is good at generating high quality random numbers, it is not very elegant and is very complex to implement. This generator is not very sensitive to initialization and can take a long time to recover from a zero-excess initial state. It is not convenient for GPU implementation due to its large state that must be updated serially. Thus each GPU thread must have an individual state in global RAM and has to make multiple accesses per generator. In combination with the relatively large amount of computation per generated number, this requirement makes the generator too slow, except in cases where the ultimate in quality is needed.

2.4. Combined Tausworthe Generator
Linear Feedback Shift Register (LFSR) or the combined Tausworthe generator provides a fast implementation and make use of only exclusive-OR and shift operations. The following code shows the example of an one component combined Tausworthe generator,

```
-------------------------------------------------------------------------------
 // S1, S2, S3, and M are all constants, and z is part of the
 // private per-thread generator state.
-------------------------------------------------------------------------------
   unsigned TausStep(unsigned &z, int S1, int S2, int S3, unsigned M)
{
  unsigned b=(((z << S1) ^ z) >> S2);
  return z = (((z & M) << S3) ^ b);
}
-------------------------------------------------------------------------------
```

A popular version of this procedure is the four component LFSR113 generator from L'Ecuyer 1999 which produces random stream with a period of approximately 2^{113} [6]. However, statistical tests show that even the four-component LFSR113 produces significant correlations across 5-tuples and 6-tuples for relatively small sample sizes. Also the periods are not sufficiently large enough. But in terms of speed, these are highly competitive to those which are currently available from software libraries.

3. A Hybrid generator for GPU programing
To increase the period length and improve the statistical behavior of a generator, it is always good to combine generators of different periods. If the periods of the individual generators are co-prime, then their product will be the period of the combined generators and also the combined generators will have less statistical flaws as compared to their individual counter parts. In the present work, we follow an approach suggested by NVIDIA [2] where three Tausworthe generator are combined with a LCG generators as given by,

```
-------------------------------------------------------------------------------
unsigned z1, z2, z3, z4;
float HybridTaus()
{
   // Combined period is lcm(p1,p2,p3,p4)~ 2^121
    return 2.3283064365387e-10 * (              // Periods
    TausStep(z1, 13, 19, 12, 4294967294UL) ^    // p1=2^31-1
    TausStep(z2, 2, 25, 4, 4294967288UL)   ^    // p2=2^30-1
    TausStep(z3, 3, 11, 17, 4294967280UL)  ^    // p3=2^28-1
    LCGStep(z4, 1664525, 1013904223UL)          // p4=2^32
   );
}
-------------------------------------------------------------------------------
```

The above combination provides an overall period of around 2^{121}. The LCGStep is a simple LCG generator of the type,

```
-------------------------------------------------------------------------------
 // A and C are constants
   unsigned LCGStep(unsigned &z, unsigned A, unsigned C)
{
  return z=(A*z+C);
}
-------------------------------------------------------------------------------
```

In the LCG, the mod operation is not explicitly required due to unsigned overflow and has a period of 2^{32} due to truncation in 32-bit arithematic. Since 2^{32} is also relatively prime to the periods of other three Tausworthe generators, the fourth combination with the LCG increases the period by a factor of 2^{32} without any heavy computation.

The above hybrid generator requires four random seeds $z1, z2, z3, z4$. One way is to generate them in the host machine using a CPU based program and then copying into the device. It is noticed that the memory copy from host to device or vice versa is very time consuming. Therefore, we use another LCG type generator for seed generation which requires only a multiplication on each GPU thread and given by,

```
-------------------------------------------------------------------------------
//Function giving seed for each thread

unsigned seed(int id)
{
return seed=id*1099087573UL;
}
-------------------------------------------------------------------------------
```

where the index *id* varies with the thread number of the GPU. This algorithm is known as quick and dirt algorithm and gives a random number due to same unsigned integer overflow. It is seen that this hybrid generator is quite fast as compared to the traditional Mersenne Twister algorithm and having nearly similar statistical performance. As an example, we have estimated the auto-correlation function defined as

$$C_m \;=\; (x[i] - \bar{x})\;(x[i+m] - \bar{x}) \tag{2}$$

where $x[i]$ is an array of random numbers between 0 and 1, $\bar{x}$ is the average and m is the correlation distance. For $m = 0$, it represents the variance of the distribution. We have estimated C_m using both Mersenne Twister (TRandom3 of root package) and the present Hybrid algorithm. As shown in figure 1, the auto-correlation function (C_m/C_0) of TRandom3 and the present hybrid generator are comparable. It is one at $m = 0$ and decreases to zero quite fast as expected.

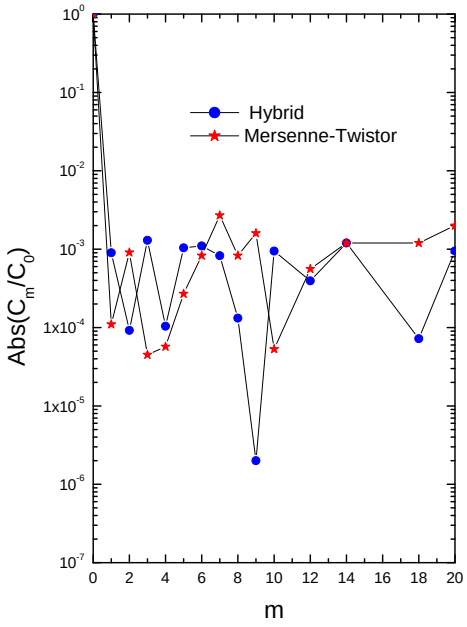

Figure 1. The comparison of auto-correlation function $Abs(C_m/C_0)$ as a function m for Mersenne-Twister (TRandom3) and Hybrid generator.

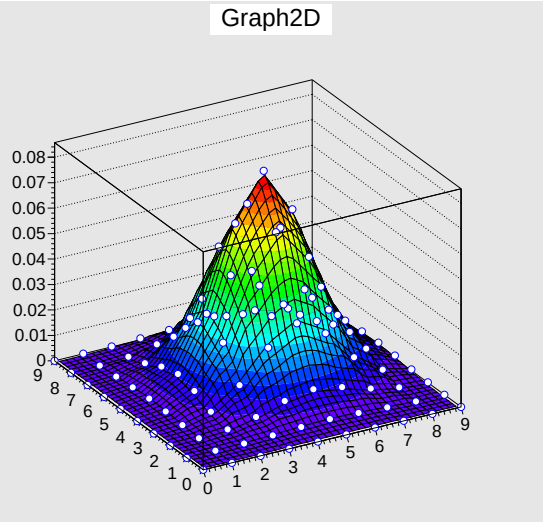

Figure 2. The comparison between actual 2D inputs (solid points) and generated points shown as continuous line obtained using alias technique.

4. Sampling algorithm and GPU implementation

In this conference, some of us have presented a methodology to carry out Monte-Carlo sampling from a multivariate correlated data set using alias technique. The motivation for this sampling is to develop a ROOT based low energy neutron transport code using ENDF nuclear data library where evaluated nuclear data like energy and angle of neutron emissions are highly correlated. In this work, we try to implement the alias algorithm using GPU to make the execution faster. The detailed of the algorithm can be found in [3, 8]. The alias method recasts the original non-alias outcome i and probability p_i into an alias outcome Λ_i and alias probability Π_i. An non-alias outcome i is chosen randomly between 0 and N with uniform probability i.e. $i = Nu_1$ where u_1 is an uniform random number between 0 and 1. The method then compares a second random number u_2 against the alias probability Π_i. If $u_2 \leq \Pi_i$, then non alias outcome i is chosen, otherwise the alias outcome Λ_i is chosen. Thus instead of the original probability array p_i, the alias method requires to store two arrays one for Λ_i and other for Π_i. This is in case of an one dimensional sampling. In case of two dimensional sampling, four tables are required to be computed and stored. Since these computations are one time, we generate them using the CPU based program in the host and copy to the device memory. We use a NVIDIA GPU (GeForce

GTX 480) which has compute capability 2.0, suitable for double precision arithematic. Other specifications are: it has total global memory of 1609760768 bytes, total constant memory 65536 bytes, multiprocessor count 15, shared memory per mp 49152 bytes, registers per mp 32768, threads in warp 32, maximum threads per block 1024 and maximum number of blocks 65535. The essential feature of GPU programing is the difference between the host and the device. The host program (CPU) executes the device kernel on each GPU thread. Although, we have carried out sampling using both OpenCL and CUDA, in the following, explanation is given as per the CUDA implementation. The host program invokes the device kernel (source code for CUDA kernel is listed at the end)

```
__global__ void cuda_kernel(int*c1,int*c2,double*fA,double*fR,double*fA1,double*fR1)
```

which is executed on each thread in parallel with thread index id that varies from 0 to $N = blockno * blocksize - 1$. For each thread, the kernel invokes two device routines $seedperthread$ for random seed generation which is used in $TauStep$ in step 1. For step 2 to step 3, the previous random number provides seed to the subsequent one. Thus, each thread generates four random numbers which are used for alias sampling. Since each thread is executed in parallel, this is an efficient and fast method to generate random number on each thread independently as id varies depending on the thread index. The sampling is carried out on each thread using the alias table fA, fR, $fA1$ and $fR1$ which are common to all threads and are copied from the host to device using the CUDA $cudaMemcpyHostToDevice$ utility. Similarly, the results are copied back from device to host using $cudaMemcpyDeviceToHost$ utility. As shown in the ode, each thread requires four random numbers $u1$, $u2$, $u3$ and $u4$ which are generated invoking the hybrid generator four times. For the first time, it takes the seed from the seed generator and use the previous number for the subsequent steps.

5. Results and conclusion

In the following, we carry out two dimensional sampling from a correlated Gaussian given by,

$$f(x,y) = e^{-[(x-x_0)^2+(y-y_0)^2+\alpha(x-x_0)(y-y_0)]} \tag{3}$$

which becomes uncorrelated when $\alpha = 0$ and can be written as the product of two independent Gaussians. We generate 100 discrete data points (as 10 by 10 matrix) from the above distribution and assign as probability $P[i][j]$ which is normalized to unity. We also fix $x_0 = y_0 = 5$ and $\alpha = 2$. This corresponds to a situation when the co-variance is non zero, although the choice of parameters are arbitrary. The sampling is carried out as follows. First we select a row J using alias technique from the distribution R_j where $R_j = \sum_i P[i][j]$ [3]. Once the row J is chosen, the corresponding column is fixed again using the same alias technique as per the distribution $P[i][J]$. Figure 2 shows typical plot of a 2D correlated Gaussian sampling where the solid points represent the actual distribution where as continuous curve is obtained through alias sampling. Note that the present implementation of alias sampling generates only the discrete data points which are connected smoothly as continuous lines using ROOT inbuilt algorithms. Although not shown here in detail, the sampling results obtained from the GPU are as accurate as what we would have got implementing the same algorithm on a CPU based program. On the other hand, the execution on GPU is about 1000 times faster as compared to the CPU. Table I shows the time taken to sample 10^6 events both on CPU and GPU. For GPU, we have implemented the algorithm both using CUDA and OpenCL kernels. The CUDA execution is faster than OpenCL as it has more overheads. It may be mentioned here that although GPU performance is much faster than CPU, the overall performance is only about 10 times faster. This is due to the large memory access time between host and device and vice versa. Work is in progress to develop better sampling algorithm and also to minimize on the memory access time so that over

all execution can be made faster. Instead of transferring all the random numbers generated on GPU to host memory, it is possible to carry out Monte-Carlo simulation of interest on each GPU thread and finally transferring the desired results. This will make the overall code execution faster.

Table 1. The comparison of execution time.

	CPU	GPU-CUDA	GPU-OpenCL
Kernel Exec.	9.73 Sec	53 μ sec	220 μ sec
Total time	9.93 sec	0.92 sec	1.12 sec

Appendix

```
--------------------------------------------------------------------------------
//Program in Cuda for 2D alias sampling
// Only the essential portions are given. This program
// performs a 2D correlated Gaussian sampling using alias technique
// for 100 data points given as 10 x 10 matrix.
--------------------------------------------------------------------------------
#include<stdio.h>
#include<cuda.h>
#include<math.h>
#include<sys/time.h>

--------------------------------------------------------------------------------
__device__ unsigned seed_per_thread(int id)
{
return id*1099087573UL;

}
--------------------------------------------------------------------------------


__device__ unsigned TauStep(unsigned z, int s1, int s2, int s3, unsigned M)
{

unsigned b=(((z << s1) ^ z) >> s2);
return z = (((z & M) << s3) ^ b);
}

--------------------------------------------------------------------------------



__global__ void cuda_kernel(int*c1,int*c2,double*fA,double*fR,double*fA1,double*fR1)
```

```
{
//fA, fR, fA1 and fA2 are alias tables which are computed in the host
//program and copied into the device using cuda memory copy utility.
//The final reasults which are stored in the device array c1 and c2 are also
//copied back to the host after the kernel execution is over.

int idx=blockIdx.x*blockDim.x+threadIdx.x;
float u1,u2,u3,u4;
int i1,i2,k1,k2;
unsigned b,z,z1,z2,z3,z4,M;
unsigned r0,r1,r2,r3;
unsigned seed;
//Generates four random numbers u1, u2, u3 and u4 in four steps.
//Only seed_per_thread is called in step 1 to privide the initial seed.

//STEP 1
seed=seed_per_thread(idx);
z1=TauStep(seed,13,19,12,429496729UL);
z2=TauStep(seed,2,25,4,4294967288UL);
z3=TauStep(seed,3,11,17,429496280UL);
z4=(1664525*seed+1013904223UL);
r0=(z1^z2^z3^z4);

//STEP 2
z1=TauStep(r0,13,19,12,429496729UL);
z2=TauStep(r0,2,25,4,4294967288UL);
z3=TauStep(r0,3,11,17,429496280UL);
z4=(1664525*r0+1013904223UL);
r1=(z1^z2^z3^z4);

//STEP 3
z1=TauStep(r1,13,19,12,429496729UL);
z2=TauStep(r1,2,25,4,4294967288UL);
z3=TauStep(r1,3,11,17,429496280UL);
z4=(1664525*r1+1013904223UL);
r2=(z1^z2^z3^z4);

//STEP 4
z1=TauStep(r2,13,19,12,429496729UL);
z2=TauStep(r2,2,25,4,4294967288UL);
z3=TauStep(r2,3,11,17,429496280UL);
z4=(1664525*r2+1013904223UL);
r3=(z1^z2^z3^z4);

------------------------------------------------------------------------------
// u1, u2, u3 and u4 varies between 0 and 1.0

u1=r0*2.3283064365387e-10;
u2=r1*2.3283064365387e-10;
u3=r2*2.3283064365387e-10;
```

```
u4=r3*2.3283064365387e-10;


i1=10*u1;
 if(u2<=fR1[i1]) k1=i1;
 else k1=fA1[i1];
i2=10*u3;
 if(u4<=fR[k1*10+i2]) k2=i2;
 else k2=fA[k1*10+i2];
c1[idx]=k1;
c2[idx]=k2;
//sampling results are stored in two arrays of c1 and c2 with thread index idx.

}
```

References

[1] Matsumoto M M and Nishimura T 1998 *ACM Transactions on Modeling and Computer Simulations* vol 8 p 3-20
[2] http://developer.nvidia.com/GPUGems3
[3] Walker A J 1977 *ACM Transactions on Mathematical Software* vol 3 p 253
[4] Knuth D 1969 *The Art of Computer Programming* (V-2, Addison-Wesly)
[5] http://root.cern.ch
[6] L'Ecuyer 1996 *Mathematics of Computations* vol 65 p 203-213
[7] James F 1994 *Computer Physics Communication* vol 79 p 111
[8] Popescu L M 2000 *Journal of Computational Physics* vol 160 p 612