Efficient Pseudo-Random Number Generation for Monte-Carlo Simulations Using GPU

Siddhant Mohanty,Ajit Mohanty and Federico Carminati

Random Number Generation

The future of high power computing is evolving towards the efficient use of highly parallel computing environment. The class of devices that have been designed having parallelism features in mind is the Graphics Processing Unit (GPU) which are highly parallel, multithreaded 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.

We use NVIDIA Graphic card with compute capability 1.3 or above which supports floating point calculation and implement a random number generator scheme which is fast and efficient for GPU implementation. As a case study, the method is applied for a correlated 2D Gaussian Sampling

Device: NVIDIA GeForce GTX 480

- Compute Capability: 2.0(suitable for double precision)
- Total Global Memory: 1609760768
- Total Constant Memory: 65536
- Multiprocessor count: 15
- Shared memory per mp: 49152
- Registers per mp: 32768
- Threads in warp: 32
- Maximum threads per block: 1024
- Maximum number of blocks: 65535

Two important aspects of this presentation

- Fast and efficient random number generation on GPU.
- Multivariate correlated sampling using both CPU and GPU.

For GPU Programming, the random number generator should have:

- Good statistical properties.
- High computational speed.
- Low memory use.
- Large period.

Mersenne Twister

- It is one of the most respected methods(used in ROOT as class TRandom3).
- Has a large period of $2^{19,937}$.
- Very good statistical properties.

Limitations of Mersenne Twister

However it is 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.

Hybrid Approach: Tausworthe and LCG

LCG(Linear Congruential Algorithm)

$$X_{n+1} = (aX_n + C) \bmod m$$

where,

m = 2^{32}

a= 1664525

c=1013904223UL

Here the mod operation is not explicitly required due to the unsigned overflow.

Combined Tausworthe Generator

There are a number of related generators that use much smaller vectors, of the order of two to four words, and a correspondingly denser matrix. An example of this kind of generator is the combined Tausworthe generator, 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.

Tausworthe Generator

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

Where s1,s2,s3,M are all constants.

Combined Generator

```
 float HybridTaus()
{
  //combined period is LCM(p1,p2,p3,p4)
  return 2.3283064364387e-10 * (                       //Periods
    TausStep(seed1,13,19,12,4294967294UL) ^   //p1=2^31-1
    TausStep(seed2,2,25,4,4294967288UL) ^      //p2=2^30-1
    TausStep(seed3,3,11,17,4294967280UL) ^     //p3=2^28-1
     LCGStep(seed4,1664525,1013904223UL)      //p4=2^32
    );
 }
```

Hence, combined period = 2^121.

Seed generation

- For the initial seed we use

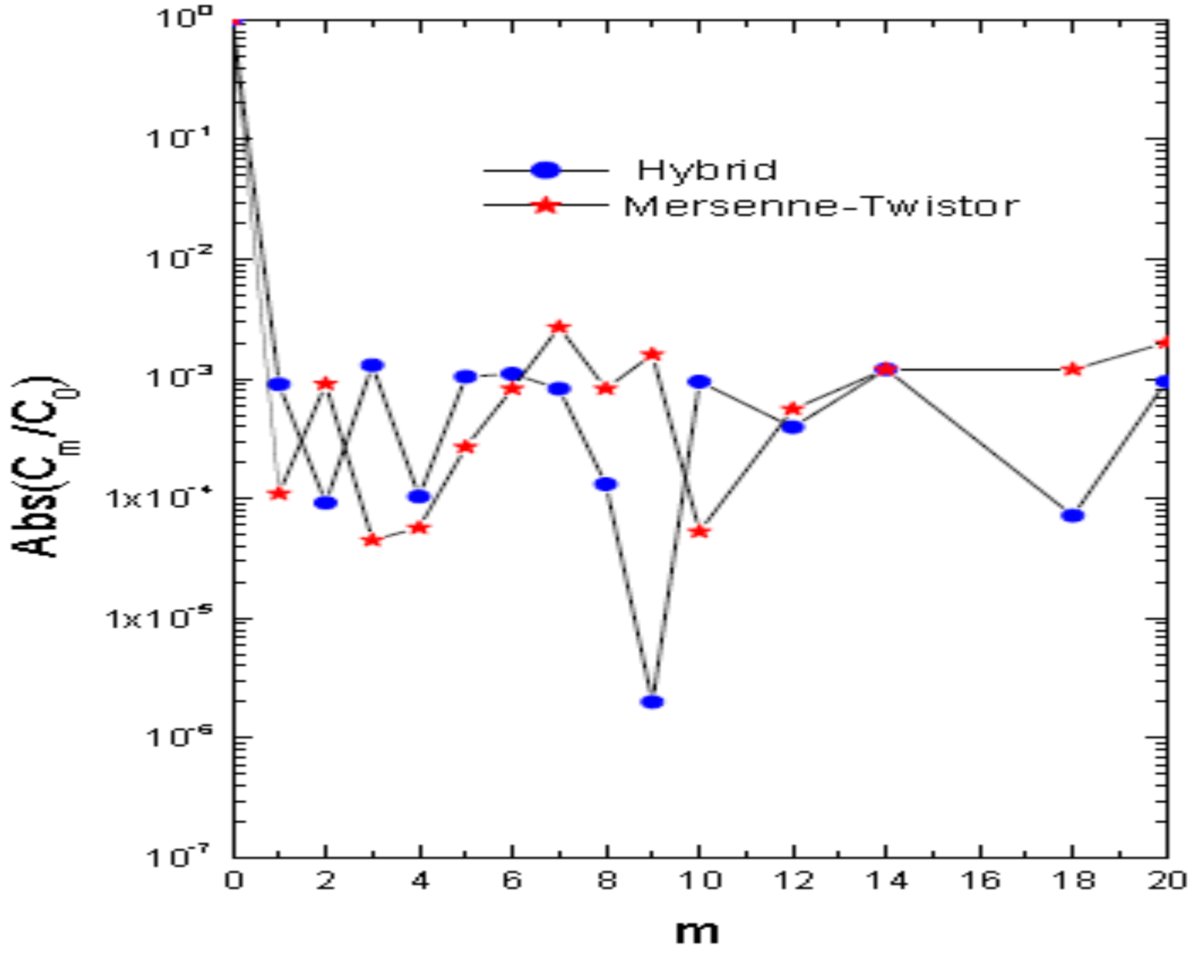
```
unsigned seed(int idx)
 {
   return idx*1099087573UL;
  }
```

Where idx is the thread index.

It is known as the quick and dirty LCG and the sequence is random enough for the seed generation.

Auto-Correlation

$$C_m = (x[i] - x_{av})(x[i+m] - x_{av})$$



Application: 2D Sampling

It is carried out :

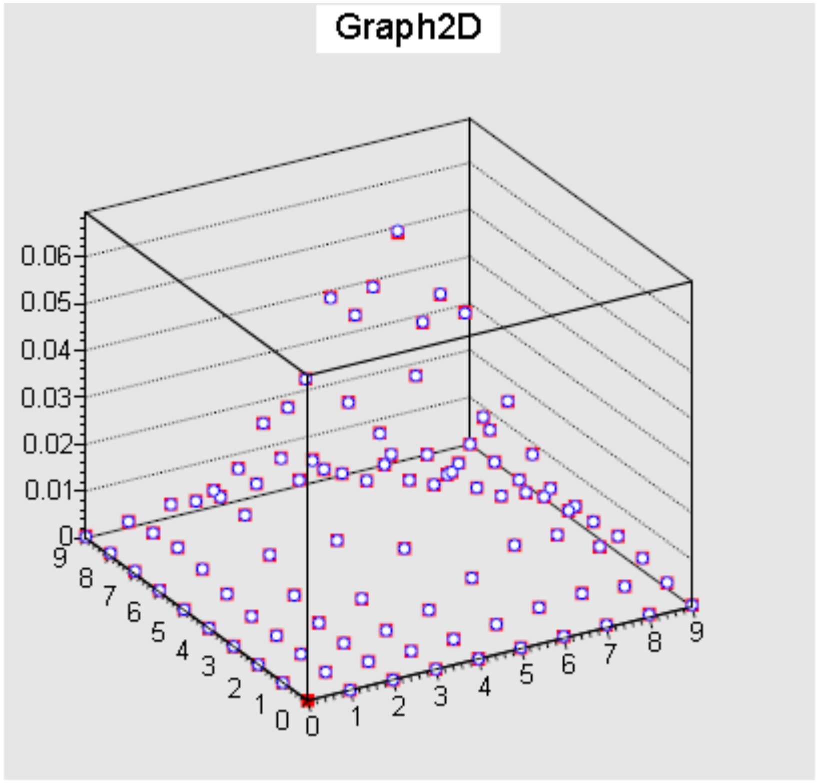
- Using CPU
- Using GPU(both CUDA and OpenCL)

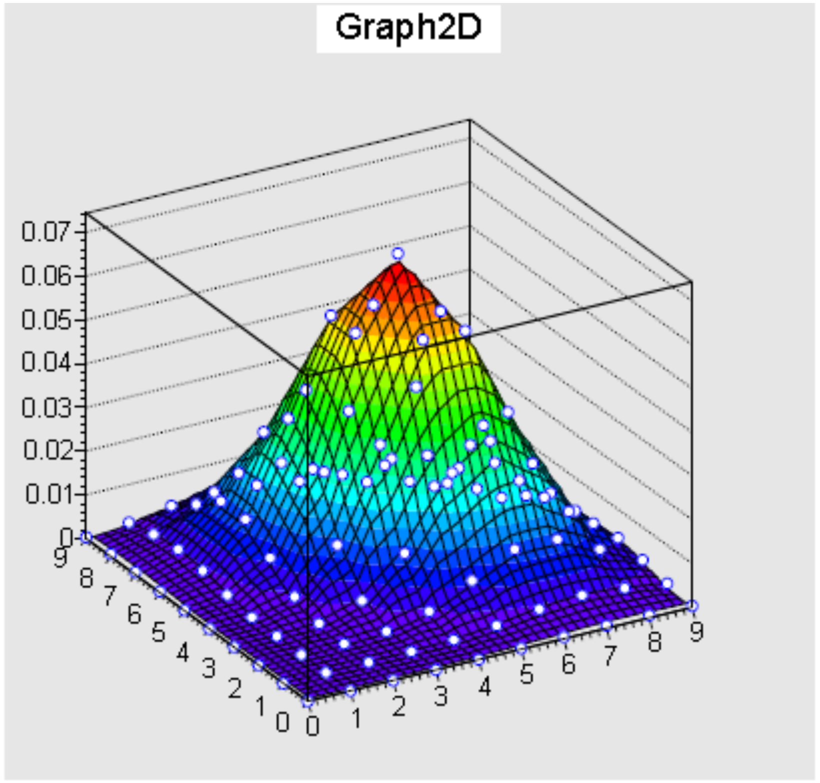
Example :

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

We Sample using walker's alias techniques which requires 4 random numbers per sample. The detail of the alias implementation is beyond the scope of this presentation.

Plots for original and generated distribution





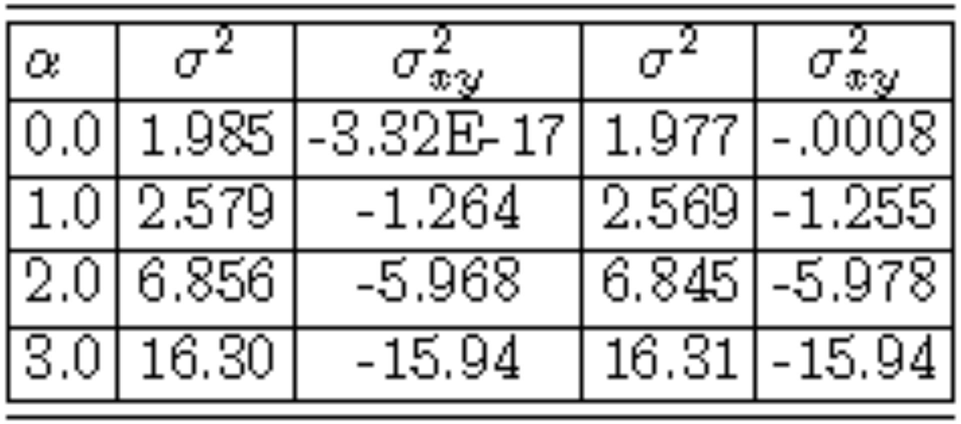
Left: Circles represent original points and squares are the generated points.
Right: Circles represent original points and continuous curve is for generated points

$$\chi^2 = \frac{1}{N}\sum\left[\frac{p[i][j]-gx[i][j]}{p[i][j]}\right]^2$$

TABLE II: The table for χ^2 test(σ^2=10.0)

α	χ^2/N $n = 10^2K$	χ^2/N $n = 10^3K$	χ^2/N $n = 10^4K$
0.0	2.56E-3	1.78E-4	2.92E-5
1.0	3.77E-3	1.53E-3	1.45E-4
2.0	2.42E-2	5.02E-3	4.51E-4
3.0	3.25E-1	4.24E-2	2.42E-2

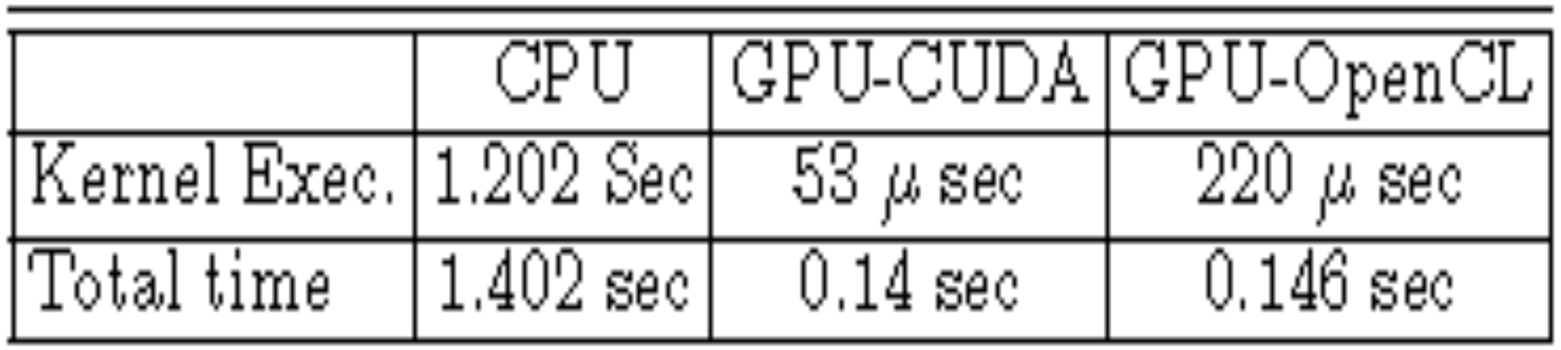
TABLE III: The table for co-variance test , N= 100K



α	σ^2	σ_{xy}^2	σ^2	σ_{xy}^2
0.0	1.985	-3.32E-17	1.977	-.0008
1.0	2.579	-1.264	2.569	-1.255
2.0	6.856	-5.968	6.845	-5.978
3.0	16.30	-15.94	16.31	-15.94

Time comparison for 10^6 events

TABLE IV: The comparison of execution time



	CPU	GPU-CUDA	GPU-OpenCL
Kernel Exec.	1.202 Sec	53 μ sec	220 μ sec
Total time	1.402 sec	0.14 sec	0.146 sec

Conclusions

- We have implemented a hybrid generator which is quite fast and efficient for GPU programming.
- Using this hybrid generator we have carried out multivariate correlated Monte-Carlo sampling.
- The kernel execution in GPU is about 1000 times faster as compared to CPU.
- The overall execution is only 10 times faster as the memory copy operation from device to host or vice-versa is extremely slow.