User guide for CUDA RNG library

Mike Giles

February 26, 2009

Abstract

This document explains how to use my CUDA RNG library. A separate document describes the implementation of the library.

Within the library the user has a choice of two random number generators:

- l'Ecuyer's multiple recursive generator mrg32k3a;
- Sobol's quasi-random number generator;

three output distributions:

- uniform;
- exponential;
- unit Normal;

and two modes of operation:

- high-level (all calls from user's main C/C++ host code);
- low-level (some calls from user's CUDA device code).

1 Initialisation

Initialisation is performed by a call from the user's main C/C++ code to specify the generator which is to be used.

void cuda_mrg32k3a_init(*v1, *v2, offset)

- **v1** (input)
  an array of three 64-bit unsigned long integers to be used as seeds
- **v2** (input)
  an array of three 64-bit unsigned long integers to be used as seeds
- **offset** (input)
  a 32-bit integer defining an offset from the beginning of the sequence

void cuda_sobol_init(maxdim, *vector, offset)

- **maxdim** (input)
  the maximum dimension of the points to be generated;
- **vector** (input)
  an array of size **maxdim** containing unsigned 32-bit integers to be used for digital scrambling
- **offset** (input)
  a 32-bit integer defining an offset from the beginning of the sequence

Comments:

1. Other generators (e.g. MCG59, rank-1 lattice rule) may be added in the future;
2. Digital scrambling is useful for randomised QMC methods to give a confidence interval;
3. The offset capability is useful if random numbers need to be generated in groups, for example due to limited GPU memory availability. There is also literature which suggests that it is best to skip the first 2^n points when one wants to generate 2^n Sobol points.

2 High-level generation

These routines can be called from the user's main C/C++ code:

void cuda_mrg32k3a_uniform(nb, nt, np, *x)
void cuda_mrg32k3a_exponential(nb, nt, np, *x)
void cuda_mrg32k3a_normal(nb, nt, np, *x)

void cuda_sobol_uniform(dim, nb, nt, np, *x)
void cuda_sobol_exponential(dim, nb, nt, np, *x)
void cuda_sobol_normal(dim, nb, nt, np, *x)

- **dim** (input)
  32-bit integer giving dimension for Sobol generator;
- **nb** (input)
  32-bit integer giving number of CUDA "blocks";
- **nt** (input)
  32-bit integer giving number of CUDA threads per block;
- **np** (input)
  32-bit integer giving number of random points to be generated by each thread;
- **x** (output)
  a device array of 32-bit real variables for the output

Comments:

1. Each random point has dimension 1 for the mrg32k3a generator, and dimension **dim** for the Sobol generator;
2. For high performance, **nt** should be a multiple of 64;
3. For the mrg32k3a generator, $x_{n,t,b}$, the n^{th} element generated by the thread t in block b, is stored at location $t+nt*n+nt*nn*b$ whereas the standard sequential storage location would be $n+nn*t+nt*nn*b$. This is due to the requirements of memory coalescence to achieve the maximum device memory bandwidth when storing the data in these routines, and when it is read back in by the user's application.

   For the Sobol generator, there is an extra index d corresponding to the dimension, and $x_{d,n,t,b}$ is stored at location $t+nt*d+nt*dim*n+nt*dim*nn*b$.

3 Low-level generation

These routines are called from the user's device code written in CUDA. First there is a stream initialisation routine which is called once by each thread.

__device__ void cuda_mrg32k3a_stream_init(*v1, *v2, np)

- **v1** (output)
  an array of three 64-bit unsigned long integers which are initialised by this routine
- **v2** (output)
  an array of three 64-bit unsigned long integers which are initialised by this routine
- **np** (input)
  32-bit integer giving number of random points to be generated by each thread

__device__ void cuda_sobol_stream_init(dim, np, *n, *v)

- **dim** (input)
  32-bit integer giving dimension for Sobol generator
- **np** (input)
  32-bit integer giving number of random points to be generated by each thread
- **n** (output)
  a 32-bit integer counter which is initialised by this routine and must be passed unaltered to subsequent routines
- **v** (output)
  an array of size **dim** of unsigned 32-bit integers which are initialised by this routine and must be passed unaltered to subsequent routines

Comments:

1. Each random point will have dimension 1 for the mrg32k3a generator, and dimension **dim** for the Sobol generator;
2. The routines determines the values of **nb** and **nt** as defined in the high-level interface from the CUDA built-in parameters **gridDim** and **blockDim**.

The following routines then generate random values each time they are called.

__device__ void cuda_mrg32k3a_next_uniform(*v1, *v2, *x)
__device__ void cuda_mrg32k3a_next_exponential(*v1, *v2, *x)
__device__ void cuda_mrg32k3a_next_normal(*v1, *v2, *x, *x2)

- **v1** (input/output)
  an array of three 64-bit unsigned long integers which are updated by this routine
- **v2** (input/output)
  an array of three 64-bit unsigned long integers which are updated by this routine
- **x** (output)
  a local real variable into which the output will go
- **x2** (input/output)
  a local real variable for temporary storage; this should be initialised to `100.0f` before this routine is called for the first time, and never touched subsequently

__device__ void cuda_sobol_next_uniform(dim, *n, *v, *x)
__device__ void cuda_sobol_next_exponential(dim, *n, *v, *x)
__device__ void cuda_sobol_next_normal(dim, *n, *v, *x)

- **dim** (input)
  32-bit integer giving dimension for Sobol generator
- **n** (input/output)
  a 32-bit integer which was initialised by **cuda_sobol_stream_init** and is updated by this routine
- **v** (input/output)
  an array of size **dim** of unsigned 32-bit integers which were initialised by **cuda_sobol_stream_init** and are updated by this routine
- **x** (output)
  pointer to a local real array of size **dim** into which the output will go

Comments:

1. These routines are automatically inlined for efficiency. The low-level routines enable random numbers to be generated as they are needed, avoiding the bandwidth cost of storing them after generation and then loading them back in to be used.
2. The normal random numbers are generated in pairs using the Box-Muller method. **x2** is used as temporary storage for the second variable which is why it must be correctly initialised and then never touched.