The CURAND library

Will Landau

Iowa State University

November 4, 2013

Outline

CURAND

- **CURAND**: a CUDA C library for quickly generating pseudorandom and quasi-random numbers.
- **Pseudorandom sequence**: a sequence of numbers, generated by a deterministic algorithm, that has most of the properties of a truly random sequence.
- **Quasi-random (low-discrepancy) sequence**: a sequence of n-dimensional points, generated by a deterministic sequence, that appear random and appear to fill a region of n-dimensional space evenly.

Host and device APIs

- Host API
  - Include the header, `curand.h`, and link with the `-lcurand` flag at compilation.
  - Calls to number generators happen on the host.
  - With each call, a predetermined number of random draws is generated and then stored for alter use in a kernel call or a copy statement.
  - Supports 3 pseudorandom generators and 4 quasi-random generators.
- Device API
  - Include the header, `curand_kernel.h`, and link with the `-lcurand` flag at compilation.
  - Calls to number generators happen within kernels and device functions.
  - Random numbers are generated and immediately used in real time on an as-need basis.
  - For CUDA version 4.2, supports fewer generator algorithms then the host API.

Outline

Using the host API

1. Create a new generator with `curandCreateGenerator()`.
2. Set the generator options. For example, use `curandSetPseudoRandomGeneratorSeed()` to set the seed.
3. Allocate memory for the random numbers with `cudaMalloc()`.
4. Generate random numbers with on or more calls to `curandGenerate()` or another generation function.
5. Clean up the generator with `curandDestroyGenerator()`.
6. Clean up everything else with `free()` and `cudaFree()`.

Generator types for curandCreateGenerator()

- Pseudorandom number generators
  - CURAND_RNG_PSEUDO_DEFAULT: XORWOW for the version currently on impact1
  - CURAND_RNG_PSEUDO_XORWOW: XORWOW algorithm
  - CURAND_RNG_PSEUDO_MRG32K3A: combined multiple recursive family
  - CURAND_RNG_PSEUDO_MTGP32: Mersenne Twister family
- Quasi-random number generators
  - CURAND_RNG_QUASI_DEFAULT: currently Sobol, 32-bit sequences
  - CURAND_RNG_QUASI_SOBOL32: Sobol, 32-bit sequences
  - CURAND_RNG_QUASI_SOBOL64: Sobol, 62-bit sequences
  - CURAND_RNG_QUASI_SCRAMBLED_SOBOL32: scrambled Sobol, 32-bit sequences
  - CURAND_RNG_QUASI_SCRAMBLED_SOBOL64: scrambled Sobol, 62-bit sequences

Generator options

- **Seed**: a 64-bit integer that initializes the starting state of a pseudorandom number generator.
- **Offset**: a parameter used sot skip ahead in the sequence. Set offset $= 100$ to return the 100th number in the sequence first. Not available for the Mersenne Twister.
- **Order**: a parameter specifying how results are ordered in global memory.

Generator functions

- Random bits:

```
curandStatus_t curandGenerate(curandGenerator_t generator,
    unsigned int *outputPtr, size_t num)
```

- Uniform(0, 1):

```
curandStatus_t curandGenerateUniform(curandGenerator_t generator
    , float *outputPtr, size_t num)

curandStatus_t curandGenerateUniformDouble(curandGenerator_t
    generator, double *outputPtr, size_t num)
```

- Normal:

```
curandStatus_t curandGenerateNormal(curandGenerator_t generator,
    float *outputPtr, size_t n, float mean, float stddev)

curandStatus_t curandGenerateNormalDouble(curandGenerator_t
    generator, double *outputPtr, size_t n, double mean,
    double stddev)
```

Generator functions

- Log-normal:

```
1 curandStatus_t curandGenerateLogNormal(curandGenerator_t
      generator, float *outputPtr, size_t n, float mean, float
      stddev)
2
3 curandStatus_t curandGenerateLogNormalDouble(curandGenerator_t
      generator, double *outputPtr, size_t n, double mean,
      double stddev)
```

Example: host_api.cu

```
1  /*
2  * This program uses the host CURAND API to generate 10 pseudorandom
       floats.
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <cuda.h>
8  #include <curand.h>
9
10 int main(int argc, char *argv[]){
11   size_t n = 10;
12   size_t i;
13   curandGenerator_t gen;
14   float *devData , *hostData;
15
16  /* Allocate n floats on host */
17   hostData = (float *) calloc(n, sizeof(float));
18
19   /* Allocate n floats on device */
20   cudaMalloc((void **) &devData, n*sizeof(float));
21
22   /* Create a Mersenne Twister pseudorandom number generator */
23   curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_MTGP32);
24
25   /* Set seed */
26   curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
27
28   /* Generate n floats on device */
29   curandGenerateUniform(gen, devData, n);
```

Example: host_api.cu

```
31   /* Copy device memory to host */
32   cudaMemcpy(hostData , devData , n * sizeof(float),
           cudaMemcpyDeviceToHost);
33
34   /* Show result */
35   printf("Random Unif(0, 1) draws:\n");
36   for(i = 0; i < n; i++) {
37     printf("   %1.4f\n", hostData[i]);
38   }
39   printf("\n");
40
41   /* Cleanup */
42   curandDestroyGenerator(gen);
43   cudaFree(devData);
44   free(hostData);
45 }
```

```
1  > nvcc host_api.cu -lcurand -o host_api
2  > ./host_api
3  Random Unif(0, 1) draws:
4     0.5823
5     0.4636
6     0.6156
7     0.9964
8     0.1182
9     0.2672
10    0.9241
11    0.7161
12    0.2309
13    0.4075
```

Outline

Using the Device API

1. Within a kernel, call `curand_init()` to initialize the "state" of the random number generator.
2. Within a (possibly separate) kernel, call `curand()` or one of its wrapper functions (such as `curand_uniform()` or `curand_normal()` to generate pseudorandom or quasi random numbers as needed.

- RNG types available
  - Pseudorandom
    - XORWOW
  - Quasi-random
    - 32-bit Sobol
    - 32-bit scrambled Sobol

Device API functions: XORWOW

```
1  __device__ void curand_init (unsigned long long seed,
2                               unsigned long long sequence,
3                               unsigned long long offset,
4                               curandState_t *state)
5
6  __device__ unsigned int
7  curand (curandState_t *state) // RANDOM BITS
8
9  __device__ float
10 curand_uniform (curandState_t *state) // U(0,1)
11
12 __device__ double
13 curand_uniform_double (curandState_t *state) // U(0,1)
14
15 __device__ float
16 curand_normal (curandState_t *state) // N(0,1)
17
18 __device__ double
19 curand_normal_double (curandState_t *state) // N(0,1)
```

Device API functions: XORWOW

```
1  __device__ float2
2  curand_normal2 (curandState_t *state) // 2 N(0,1) draws
3
4  __device__ float2
5  curand_log_normal2 (curandState_t *state) // 2 N(0,1) draws
6
7  __device__ float
8  curand_log_normal (curandState_t *state, float mean, float stddev)
9
10 __device__ double
11 curand_log_normal_double (curandState_t *state, double mean, double
       stddev)
12
13 __device__ double2
14 curand_normal2_double (curandState_t *state) // 2 draws
15
16 __device__ double2
17 curand_log_normal2_double (curandState_t *state) // 2 draws
```

Device API functions: Sobol

```
1  __device__ void
2  curand_init (
3      unsigned int *direction_vectors,
4      unsigned int offset,
5      curandStateSobol32_t *state) // Sobol
6
7  __device__ void
8  curand_init (
9      unsigned int *direction_vectors,
10     unsigned int scramble_c,
11     unsigned int offset,
12     curandStateScrambledSobol32_t *state) // Scrambled Sobol
13
14 __device__ unsigned int
15 curand (curandStateSobol32_t *state)
16
17 __device__ float
18 curand_uniform (curandStateSobol32_t *state)
```

Device API functions: Sobol

```
1  __device__ float
2  curand_normal (curandStateSobol32_t *state)
3
4  __device__ float
5  curand_log_normal (
6      curandStateSobol32_t *state,
7      float mean,
8      float stddev)
9
10 __device__ double
11 curand_uniform_double (curandStateSobol32_t *state)
12
13 __device__ double
14 curand_normal_double (curandStateSobol32_t *state)
15
16 __device__ double
17 curand_log_normal_double (
18     curandStateSobol32_t *state,
19     double mean,
20     double stddev)
```

Example: device_api.cu

```
1  /*
2   * This program uses the device CURAND API to calculate what
3   * proportion of pseudo-random ints are odd.
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <cuda.h>
9  #include <curand_kernel.h>
10
11 __global__ void setup_kernel(curandState *state){
12   int id = threadIdx.x + blockIdx.x * 64;
13
14   /* Each thread gets same seed, a different sequence number, no
         offset */
15   curand_init(1234, id, 0, &state[id]);
16 }
17
18 __global__ void generate_kernel(curandState *state, int *result){
19   int id = threadIdx.x + blockIdx.x * 64; int count = 0;
20   unsigned int x;
21
22   /* Copy state to local memory for efficiency */
23   curandState localState = state[id];
24
25   /* Generate pseudo-random unsigned ints */
26   for(int n = 0; n < 100000; n++){
27     x = curand(&localState);
```

Example: device_api.cu

```
28      /* Check if odd */
29      if(x & 1){
30        count ++;
31      }
32    }
33
34    /* Copy state back to global memory */
35    state[id] = localState;
36
37    /* Store results */
38    result[id] += count;
39  }
40
41  int main(int argc, char *argv[]){
42    int i, total;
43
44    int *devResults, *hostResults;
45    curandState *devStates;
46
47    /* Allocate space for results on host */
48    hostResults = (int *) calloc(64 * 64, sizeof(int));
49
50    /* Allocate space for results on device */
51    cudaMalloc((void **)&devResults, 64 * 64 *sizeof(int));
52
53    /* Set results to 0 */
54    cudaMemset(devResults, 0, 64 * 64 * sizeof(int));
55
56    /* Allocate space for prng states on device */
57    cudaMalloc((void **)&devStates, 64 * 64 * sizeof(curandState));
```

Example: device_api.cu

```
59    /* Setup prng states */
60    setup_kernel<<<64, 64>>>(devStates);
61
62    /* Generate and use pseudorandom numbers*/
63    for(i = 0; i < 10; i++){
64      generate_kernel<<<64, 64>>>(devStates, devResults);
65    }
66
67    /* Copy device memory to host */
68    cudaMemcpy(hostResults, devResults , 64 * 64 * sizeof(int),
          cudaMemcpyDeviceToHost);
69
70    /* Show result */
71    total = 0;
72    for(i = 0; i < 64 * 64; i++) {
73      total += hostResults[i];
74    }
75    printf("Fraction odd was %10.13f\n", (float) total / (64.0f * 64.0f
          * 100000.0f * 10.0f));
76
77    /* Cleanup */
78    cudaFree(devStates);
79    cudaFree(devResults);
80    free(hostResults);
81
82    return EXIT_SUCCESS;
83  }
```

Example: device_api.cu

```
1 > nvcc device_api.cu -lcurand -o device_api
2 ptxas /tmp/tmpxft_000020d0_00000000-2_device_api.pts, line 501;
3   warning : Double is not supported. Demoting to float.
4 >
5 > ./device_api
6 Fraction odd was 0.4999966323376
```

Outline

RNG types supported

- Dr. Jarad Niemi wrote example rejection sampling code available at https://github.com/jarad/gpuRejectionSampling.
- Idea
  1. Draw a pseudorandom number, x.
  2. If x is too big, throw out x and return to step 1.
  3. Return x.

cpu_runif.c

```
1  #include <Rmath.h>
2  //#include <stdlib.h>
3
4
5  int cpu_runif(int n, double ub, int ni, int nd, double *u, int *count
       )
6  {
7      int i, j, a;
8      double b;
9      GetRNGstate();
10     for (i=0;i<n;i++) {
11         count[i] = -1;
12         u[i] = ub+1;
13
14         while ( u[i]>ub  ) {
15             count[i]++;
16             //u[i] = rand()/((double)RAND_MAX + 1);
17             u[i] = runif(0,1);
18
19             // Computational overhead
20             a=0; for (j=0; j<ni; j++) a += 1;
21             b=1; for (j=0; j<nd; j++) b *= 1.00001;
22         }
23     }
24     PutRNGstate();
25 }
26
27 void cpu_runif_wrap(int *n, double *ub, int *ni, int *nd, double *u,
       int *count){
28     cpu_runif(*n, *ub, *ni, *nd, u, count);
29 }
```

gpu_runif.cu

```
1  #include <curand_kernel.h>
2  #include "cutil_inline.h"
3
4  #define THREADS_PER_BLOCK 256
5
6  __global__ void setup_prng(unsigned long long seed, curandState *
       state)
7  {
8      int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
9      curand_init(seed, id, 0, &state[id]);
10 }
11
12 __global__ void runif_kernel(curandState *state, double ub, int ni,
       int nd,
13                              double *uniforms, int *counts)
14 {
15     int i, a, count, id = threadIdx.x + blockIdx.x *
           THREADS_PER_BLOCK;
16     double b, u;
17
18     // Copy state to local memory for efficiency */
19     curandState localState = state[id];
20
21     // Find random uniform below the upper bound
22     count  = -1;
23     u = ub+1;
```

gpu_runif.cu

```
24   while ( u>ub )
25      {
26           count++;
27           u = curand_uniform_double(&localState);
28
29           // Computational overhead
30           a=0; for (i=0; i<ni; i++) a += 1;
31           b=1; for (i=0; i<nd; i++) b *= 1.00001;
32      }
33
34      // Copy state back to global memory */
35      state[id] = localState ;
36
37      // Store results */
38      uniforms[id] = u;
39      counts[id] = count;
40  }
41
42  //CURAND_RNG_PSEUDO_MTGP32
43
44  extern "C" {
45
46  void gpu_runif(int *n, double *ub, int *ni, int *nd, double *seed,
          double *u, int *c)
47  {
48      int nBlocks = *n/THREADS_PER_BLOCK, *d_c;
49      size_t u_size = *n *sizeof(double), c_size = *n *sizeof(int);
50      double *d_u;
51
52      cutilSafeCall( cudaMalloc((void**)&d_u,  u_size) );
53      cutilSafeCall( cudaMalloc((void**)&d_c,  c_size) );
```

gpu_runif.cu

```
54      // Setup prng states
55      curandState *d_states;
56      cutilSafeCall( cudaMalloc((void**)&d_states, nBlocks*
            THREADS_PER_BLOCK*sizeof(curandState)) );
57      setup_prng<<<nBlocks,THREADS_PER_BLOCK>>>(*seed, d_states);
58
59      runif_kernel<<<nBlocks,THREADS_PER_BLOCK>>>(d_states, *ub, *ni, *
            nd, d_u, d_c);
60
61      cutilSafeCall( cudaMemcpy(u,   d_u,   u_size,
            cudaMemcpyDeviceToHost) );
62      cutilSafeCall( cudaMemcpy(c,   d_c,   c_size,
            cudaMemcpyDeviceToHost) );
63
64      cutilSafeCall( cudaFree(d_u)       );
65      cutilSafeCall( cudaFree(d_c)       );
66      cutilSafeCall( cudaFree(d_states) );
67  }
68
69  } // end of extern "C"
```

my.unif.r

```
my.runif = function(n, ub, ni=1, nd=1,
                        engine="R", seed=1)
{
    engine = pmatch(engine, c("R","C","GPU"))

    switch(engine,
    {
        # R implementation
        u = rep(Inf,n)
        count = rep(0,n)
        set.seed(seed)

        for (i in 1:n) while( (u[i] <- runif(1))>ub )
        {
            count[i] = count[i]+1
            a = 0
            b = 1
            for (j in 1:ni) a = a + 1
            for (j in 1:nd) b = b * 1.00001
        }
        return(list(u=u,count=count))
    },
```

my.unif.r

```
23      {
24          # C implementation
25          set.seed(seed)
26          out = .C("cpu_runif_wrap",
27                       as.integer(n),
28                       as.double(ub),
29                       as.integer(ni),
30                       as.integer(nd),
31                       u=double(n),
32                       count=integer(n))
33          return(list(u=out$u,count=out$count))
34      },
35      {
36          # GPU implementation
37          out = .C("gpu_runif", as.integer(n), as.double(ub),
38                                as.integer(ni), as.integer(nd),
39                                as.double(seed),
40                                u=double(n), count=integer(n))
41          return(list(u=out$u,count=out$count))
42      })
43  }
```

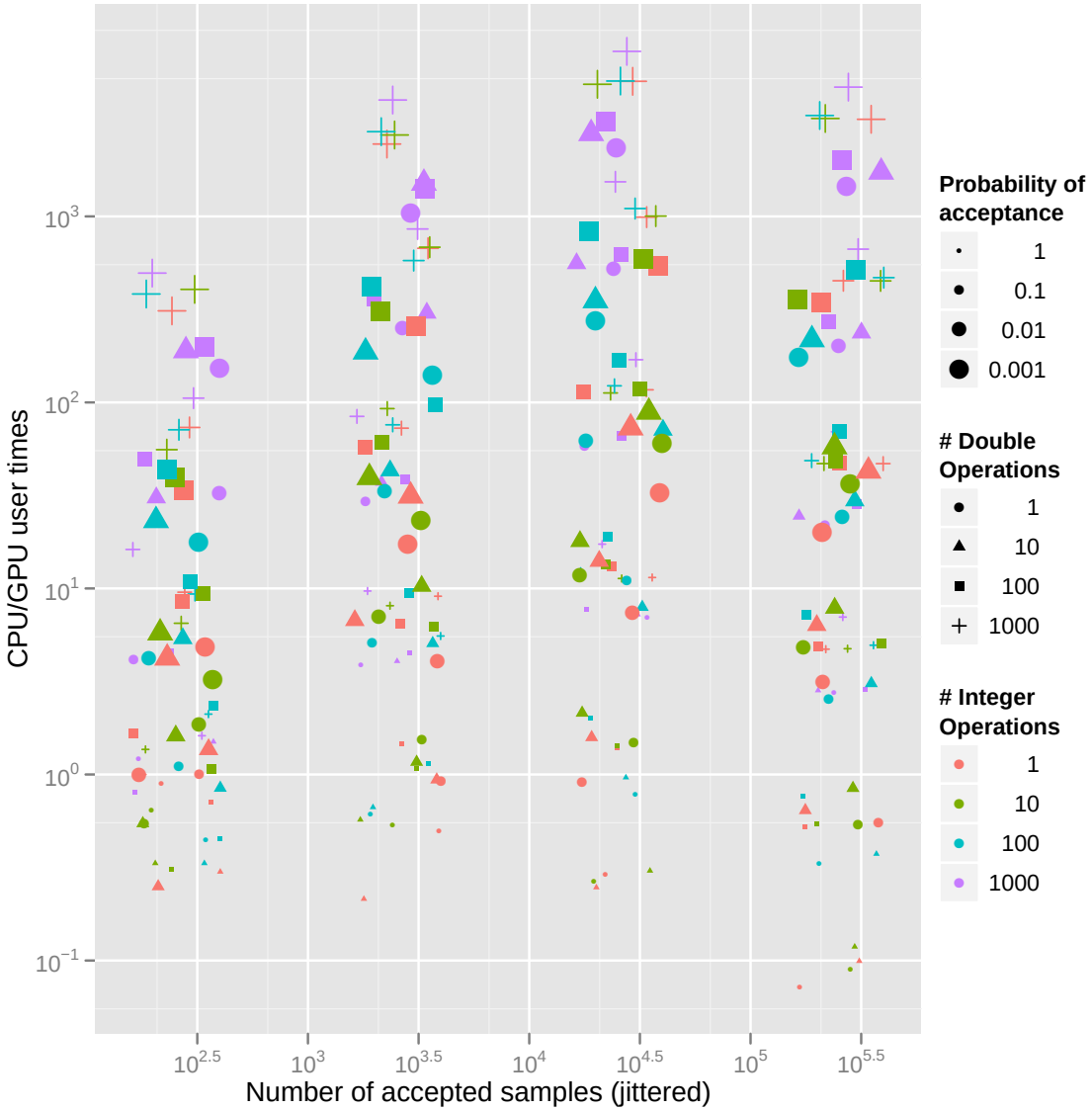
Running the example

- The files, `comparison.r` and `comparison-analysis.r`, compare the performances of the R, C, and GPU rejection samplers.

```
> ls
demo  inst  R  README.md  src
> cd src
> make
/usr/local/cuda/bin/nvcc -arch=sm_20  -c -I. -I/usr/local/include -I/
      usr/local/cuda/include -I/apps/lib64/R/include -I/usr/local/
      NVIDIA_GPU_Computing_SDK/C/common/inc -Xcompiler -fpic -DRPRINT
      -DNDEBUG cpu_runif.c -o cpu_runif.o
...

> cd ..
> ls
demo  inst  R  README.md  src
> cd demo
> ls
comparison.R      comparison-analysis.R      segfault.R
> R CMD BATCH comparison.R & # do this using screen: it takes a
      couple days unless you modify comparison.R
> R CMD BATCH comparison-analysis.R
> ls
comparison-analysis.R      comparison.csv  comparison.Rout  rejection.
      pdf  segfault.R
comparison-analysis.Rout  comparison.R    comparison.tex   Rplots.pdf
          sm.tex
```

Performance: ratios of CPU time to GPU time



Outline

Resources

- Guides:
  1. CURAND Guide
- Code from today:
  - host_api.cu
  - device_api.cu
  - Dr. Niemi's rejection sampling code

That's all for today.

- Series materials are available at http://will-landau.com/gpu.