THE CURAND LIBRARY VERSION 4.2

Will Landau, Prof. Jarad Niemi

OUTLINE

- Using the host API
- Using the device API
- Rejection sampling on the GPU

Featured examples:

- host_api.cu
- device_api.cu
- Dr. Niemi's rejection sampling code at https://github.com/jarad/gpuRejectionSampling.

CURAND: A CUDA C library for generating pseudorandom and quasi-random numbers.

Pseudorandom sequence: a sequence of numbers that is generated by a deterministic algorithm but that exhibits 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 to appear random and to fill a region of n-dimensional space evenly.

THE 2 PIECES OF CURAND

Host API

- Include the header, `curand.h`, and link with the `-lcurand` flag at compilation.
- Calls to number generators happen on host.
- With each call, a predetermined number of random draws is generated and then stored for later use in a kernel call or a copy statement.
- Supports 3 pseudorandom generators and 4 quasirandom 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 other device functions.
- Random numbers are generated and immediately consumed in real time on an as-need basis.
- Supports few generator algorithms.

USING THE HOST API

1. Create a new generator of the desired type with `curandCreateGenerator()`.
2. Set the generator options. For example, use `curandSetPseudoRandomGeneratorSeed()` to set the seed.
3. Allocate memory on the device with `cudaMalloc()`.
4. Generate random numbers with `curandGenerate()` or another generation function.
5. Use the results.
6. If desired, generate more random numbers with more calls to `curandGenerate()`.
7. Clean up the generator with `curandDestroyGenerator()`.
8. Clean up other objects with `free()` and `cudaFree()`.

GENERATOR TYPES: `curandCreateGenerator()`

Pseudorandom Number Generators:

- **CURAND_RNG_PSEUDO_DEFAULT:** currently XORWOW algorithm
- **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, 64-bit sequences
- **CURAND_RNG_QUASI_SCRAMBLED_SOBOL32:** Scrambled Sobol, 32-bit sequences
- **CURAND_RNG_QUASI_SCRAMBLED_SOBOL64:** Scrambled Sobol, 64-bit sequences

GENERATOR OPTIONS

Seed: a 64-bit integer that initializes the starting state of a pseudorandom number generator

Offset: a parameter used to skip ahead in the sequence. If offset = 100, the first random number generated will be the 100th in the sequence. Not available for CURAND_RNG_PSEUDO_MTGP32.

Order: a parameter specifying how the results are ordered in global memory.

Pseudorandom sequence order options

CURAND_ORDERING_PSEUDO_DEFAULT

CURAND_ORDERING_PSEUDO_BEST currently implemented the same as the default

CURAND_ORDERING_PSEUDO_SEEDED

Quasi-random sequence order options

CURAND_ORDERING_QUASI_DEFAULT

GENERATOR FUNCTIONS

Random bits:

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

Random Unif(0,1):

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

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

Random 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)
```

Random Log-normal:

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

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

HOST API EXAMPLE: host_api.cu

```
/*
* This program uses the host CURAND API to generate 10
  pseudorandom floats.
*/

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

int main(int argc, char *argv[]){
  size_t n = 10;
  size_t i;
  curandGenerator_t gen;
  float *devData , *hostData;

  /* Allocate n floats on host */
  hostData = (float *) calloc(n, sizeof(float));

```

```
/* Allocate n floats on device */
cudaMalloc((void **) &devData, n*sizeof(float));

/* Create a Mersenne Twister pseudorandom number generator */
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_MTGP32);

/* Set seed */
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);

/* Generate n floats on device */
curandGenerateUniform(gen, devData, n);

/* Copy device memory to host */
cudaMemcpy(hostData , devData , n * sizeof(float),
  cudaMemcpyDeviceToHost);

/* Show result */
printf("Random Unif(0, 1) draws:\n");
for(i = 0; i < n; i++) {
  printf("  %1.4f\n", hostData[i]);
}
printf("\n");

/* Cleanup */
```

```
    curandDestroyGenerator(gen);
    cudaFree(devData);
    free(hostData);
}
```

OUTPUT

```
[landau@impact1 host_api]$ make
nvcc host_api.cu -lcurand -o host_api
[landau@impact1 host_api]$ ./host_api
Random Unif(0, 1) draws:
  0.5823
  0.4636
  0.6156
  0.9964
  0.1182
  0.2672
  0.9241
  0.7161
  0.2309
  0.4075

[landau@impact1 host_api]$
```

USING THE DEVICE API

1. Within a kernel, call `curand_init()` to initialize the “state” of the random number generator.
2. Within a 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 SUPPORTED

Pseudorandom:

- XORWOW

Quasi-random:

- 32-bit Sobol
- 32-bit scrambled Sobol

Notes:

- MRG32k3a (combined multiple recursive PRNG) is ostensibly available, but there is no documentation on how to access it.
- MTGP32 (Mersenne Twister PRNG) is ostensibly available, but the associated functions mentioned in the documentation are undefined in the library.

XORWOW

Initialize RNG with:

```
__device__ void curand_init (unsigned long long seed,
                             unsigned long long sequence,
                             unsigned long long offset,
                             curandState_t *state)
```

Then, output pseudorandom numbers with any of the following:

```
__device__ unsigned int
curand (curandState_t *state) // RANDOM BITS

__device__ float
curand_uniform (curandState_t *state) // U(0,1)

__device__ double
curand_uniform_double (curandState_t *state) // U(0,1)

__device__ float
curand_normal (curandState_t *state) // N(0,1)
```

```
__device__ double
curand_normal_double (curandState_t *state) // N(0,1)

__device__ float2
curand_normal2 (curandState_t *state) // 2 N(0,1) draws

__device__ float2
curand_log_normal2 (curandState_t *state) // 2 N(0,1) draws

__device__ float
curand_log_normal (curandState_t *state, float mean, float stddev
  )

__device__ double
curand_log_normal_double (curandState_t *state, double mean,
  double stddev)

__device__ double2
curand_normal2_double (curandState_t *state) // 2 draws

__device__ double2
curand_log_normal2_double (curandState_t *state) // 2 draws
```

SOBOL

Initialize the QRNG with one of the following:

```
__device__ void
curand_init (
    unsigned int *direction_vectors,
    unsigned int offset,
    curandStateSobol32_t *state) // Sobol

__device__ void
curand_init (
    unsigned int *direction_vectors,
    unsigned int scramble_c,
    unsigned int offset,
    curandStateScrambledSobol32_t *state) // Scrambled Sobol
```

Then, generate quasi-random numbers with any of the following:

```
__device__ unsigned int
curand (curandStateSobol32_t *state)

__device__ float
```

```
curand_uniform (curandStateSobol32_t *state)

__device__ float
curand_normal (curandStateSobol32_t *state)

__device__ float
curand_log_normal (
    curandStateSobol32_t *state,
    float mean,
    float stddev)

__device__ double
curand_uniform_double (curandStateSobol32_t *state)

__device__ double
curand_normal_double (curandStateSobol32_t *state)

__device__ double
curand_log_normal_double (
    curandStateSobol32_t *state,
    double mean,
    double stddev)
```

EXAMPLE: device_api.cu

```
/*
 * This program uses the device CURAND API to calculate what
 * proportion of pseudo-random ints are odd.
 */

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

__global__ void setup_kernel(curandState *state){
  int id = threadIdx.x + blockIdx.x * 64;

  /* Each thread gets same seed, a different sequence number , no offset */
  curand_init(1234, id, 0, &state[id]);
}

__global__ void generate_kernel(curandState *state, int *result){
  int id = threadIdx.x + blockIdx.x * 64; int count = 0;
  unsigned int x;

  /* Copy state to local memory for efficiency */
  curandState localState = state[id];

  /* Generate pseudo -random unsigned ints */
  for(int n = 0; n < 100000; n++){
    x = curand(&localState);

    /* Check if odd */
    if(x & 1){
      count ++;
    }
  }

  /* Copy state back to global memory */
  state[id] = localState;

  /* Store results */
  result[id] += count;
}

int main(int argc, char *argv[]){
  int i, total;

 int *devResults, *hostResults;
  curandState *devStates;

  /* Allocate space for results on host */
  hostResults = (int *) calloc(64 * 64, sizeof(int));

  /* Allocate space for results on device */
  cudaMalloc((void **)&devResults , 64 * 64 *sizeof(int));

  /* Set results to 0 */
  cudaMemset(devResults , 0, 64 * 64 * sizeof(int));

  /* Allocate space for prng states on device */
  cudaMalloc((void **)&devStates , 64 * 64 * sizeof(curandState));

  /* Setup prng states */
  setup_kernel<<<64, 64>>>(devStates);

  /* Generate and use pseudorandom numbers*/
  for(i = 0; i < 10; i++){
    generate_kernel<<<64, 64>>>(devStates, devResults);
```

```
  }

  /* Copy device memory to host */
  cudaMemcpy(hostResults, devResults , 64 * 64 * sizeof(int),
    cudaMemcpyDeviceToHost);

  /* Show result */
  total = 0;
  for(i = 0; i < 64 * 64; i++) {
    total += hostResults[i];
  }
  printf(”Fraction odd was %10.13f\n”, (float) total / (64.0f *
    64.0f * 100000.0f * 10.0f));

  /* Cleanup */
  cudaFree(devStates);
  cudaFree(devResults);
  free(hostResults);

  return EXIT_SUCCESS;
}
```

OUTPUT

```
[landau@impact1 device_api]$ make
nvcc device_api.cu -lcurand -o device_api
ptxas /tmp/tmpxft_000020d0_00000000-2_device_api.ptx, line 501;
  warning : Double is not supported. Demoting to float
[landau@impact1 device_api]$ ./device_api
Fraction odd was 0.4999966323376
[landau@impact1 device_api]$
```

EXAMPLE: REJECTION SAMPLING

Dr. Niemi's rejection sampling code is available at https://github.com/jarad/gpuRejectionSampling.

Rejection sampling:

1. Draw a pseudorandom number, x.
2. If x is too big, throw out x and return to step 1.
3. Return x if x is small enough.

cpu_runif.c

```
#include <Rmath.h>
//#include <stdlib.h>


int cpu_runif(int n, double ub, int ni, int nd, double *u, int *
   count)
{
    int i, j, a;
    double b;
    GetRNGstate();
    for (i=0;i<n;i++)
    {
        count[i] = -1;
        u[i] = ub+1;
        while ( u[i]>ub  )
        {
            count[i]++;
            //u[i] = rand()/((double)RAND_MAX + 1);
            u[i] = runif(0,1);
            // Computational overhead
            a=0; for (j=0; j<ni; j++) a += 1;
            b=1; for (j=0; j<nd; j++) b *= 1.00001;
        }
    }
    PutRNGstate();
}

void cpu_runif_wrap(int *n, double *ub, int *ni, int *nd, double
  *u, int *count)
{
    cpu_runif(*n, *ub, *ni, *nd, u, count);
}
```

gpu_runif.cu

```
#include <curand_kernel.h>
#include "cutil_inline.h"

#define THREADS_PER_BLOCK 256

__global__ void setup_prng(unsigned long long seed, curandState *
  state)
{
    int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
    curand_init(seed, id, 0, &state[id]);
}

__global__ void runif_kernel(curandState *state, double ub, int
  ni, int nd,
                                   double *uniforms, int *counts)
{
    int i, a, count, id = threadIdx.x + blockIdx.x *
      THREADS_PER_BLOCK;
    double b, u;
    // Copy state to local memory for efficiency */
    curandState localState = state[id];

    // Find random uniform below the upper bound
    count  = -1;
    u = ub+1;
    while ( u>ub )
    {
        count++;
        u = curand_uniform_double(&localState);

        // Computational overhead
        a=0; for (i=0; i<ni; i++) a += 1;
        b=1; for (i=0; i<nd; i++) b *= 1.00001;
    }

    // Copy state back to global memory */
    state[id] = localState ;

    // Store results */
    uniforms[id] = u;
    counts[id] = count;
}
```

```
//CURAND_RNG_PSEUDO_MTGP32

extern "C" {

void gpu_runif(int *n, double *ub, int *ni, int *nd, double *seed
   , double *u, int *c)
{
    int nBlocks = *n/THREADS_PER_BLOCK, *d_c;
    size_t u_size = *n *sizeof(double), c_size = *n *sizeof(int);
    double *d_u;

    cutilSafeCall( cudaMalloc((void**)&d_u,  u_size) );
    cutilSafeCall( cudaMalloc((void**)&d_c,  c_size) );

    // Setup prng states
    curandState *d_states;
    cutilSafeCall( cudaMalloc((void**)&d_states, nBlocks*
      THREADS_PER_BLOCK*sizeof(curandState)) );
    setup_prng<<<nBlocks,THREADS_PER_BLOCK>>>(*seed, d_states);
    runif_kernel<<<nBlocks,THREADS_PER_BLOCK>>>(d_states, *ub, *
      ni, *nd, d_u, d_c);

    cutilSafeCall( cudaMemcpy(u,   d_u,   u_size,
      cudaMemcpyDeviceToHost) );
    cutilSafeCall( cudaMemcpy(c,   d_c,   c_size,
      cudaMemcpyDeviceToHost) );

    cutilSafeCall( cudaFree(d_u)      );
    cutilSafeCall( cudaFree(d_c)      );
    cutilSafeCall( cudaFree(d_states) );
}

} // end of extern "C"
```

my.runif.r

```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))
    }, {
        # C implementation
        set.seed(seed)
        out <- .C("cpu_runif_wrap", as.integer(n),
            as.double(ub), as.integer(ni), as.integer(nd),
            u = double(n), count = integer(n))
        return(list(u = out$u, count = out$count))
    }, {
        # GPU implementation
        out <- .C("gpu_runif", as.integer(n), as.double(ub),
            as.integer(ni), as.integer(nd), as.double(seed),
            u = double(n), count = integer(n))
        return(list(u = out$u, count = out$count))
    })
}
```

HOW TO RUN THE EXAMPLE

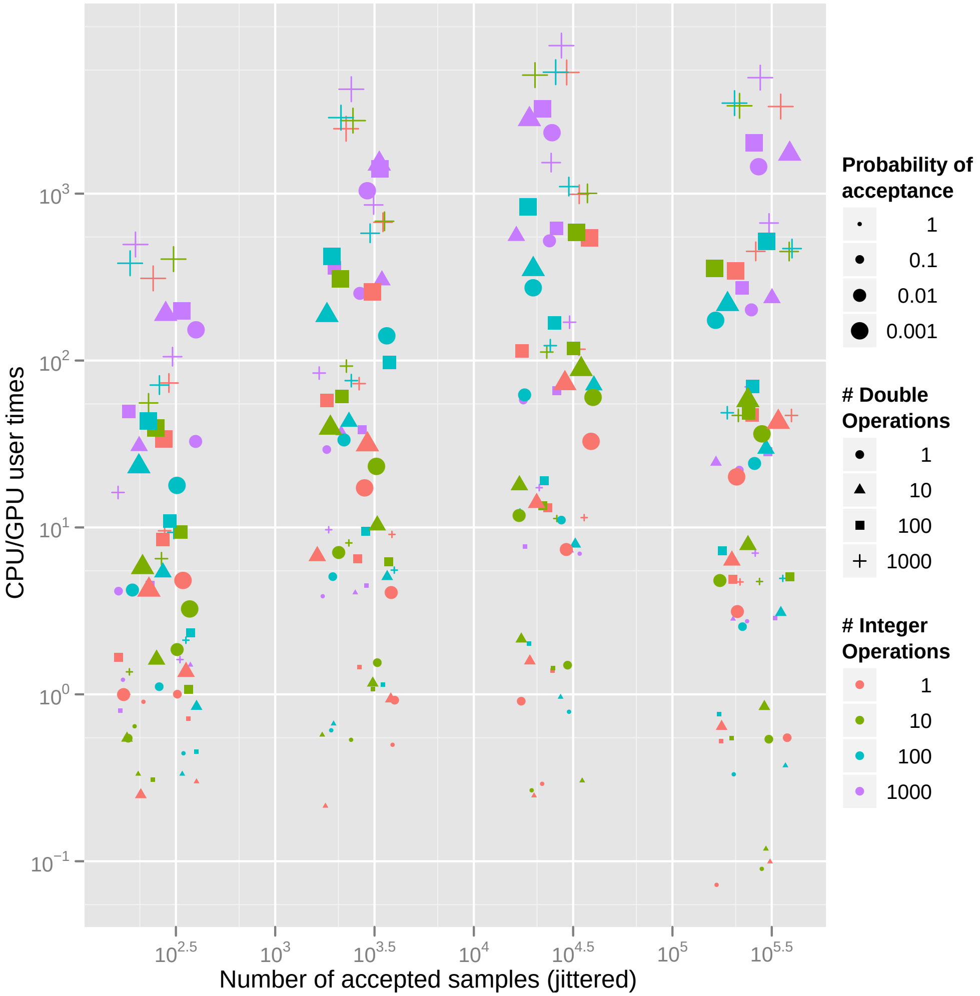
The files, `comparison.r` and `comparison-analysis.r`, compare the performances of the R, C, and GPU rejection samplers.

Here is the workflow:

```
[landau@impact1 gpuRejectionSampling]$ ls
demo  inst  R  README.md  src
[landau@impact1 gpuRejectionSampling]$ cd src
[landau@impact1 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
/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 gpu_runif.cu -o gpu_runif.o
/usr/local/cuda/bin/nvcc -arch=sm_20  -shared -Xlinker -L/usr/
   local/lib64  -L/usr/local/cuda/lib64  -lcublas -lRlapack -L/
   apps/lib64/R/lib -lRblas -lgfortran -lm -L/usr/local/
```

```
  NVIDIA_GPU_Computing_SDK/C/common/lib/linux   cpu_runif.o
  gpu_runif.o  -o runif.so
[landau@impact1 src]$ cd ..
[landau@impact1 gpuRejectionSampling]$ ls
demo  inst  R  README.md  src
[landau@impact1 gpuRejectionSampling]$ cd demo
[landau@impact1 demo]$ ls
comparison.R    comparison-analysis.R    segfault.R
[landau@impact1 demo]$ R CMD BATCH comparison.R & # do this using
   screen: it takes a couple days unless you modify comparison.R
[landau@impact1 demo]$ R CMD BATCH comparison-analysis.R
[landau@impact1 demo]$ 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

- Using the host API
- Using the device API
- Rejection sampling on the GPU

Featured examples:

- host_api.cu
- device_api.cu
- Dr. Niemi's rejection sampling code at https://github.com/jarad/gpuRejectionSampling.

GPU SERIES MATERIALS

These slides, a tentative syllabus for the whole series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the materials.

REFERENCES

"CUDA Toolkit 4.2 CURAND Guide". NVIDIA. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf

Niemi, Jarad. "gpuRejectionSampling". https://github.com/jarad/gpuRejectionSampling