PRNG Random Numbers on GPU

W. B. Langdon

Mathematical and Biological Sciences, University of Essex,
Colchester CO4 3SQ, UK

Technical Report CES-477
ISSN: 1744-8050
29 November 2007

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
John von Neumann

Abstract

Limited numerical precision of nVidia GeForce 8800 GTX and other GPUs requires careful implementation of PRNGs. The Park-Miller PRNG is programmed using G80's native Value4f floating point in RapidMind C++. Speed up is more than 40. Code is available via ftp ftp://cs.ucl.ac.uk/genetic/gp-code/random-numbers/gpu_park-miller.tar.gz

1 Introduction

Monte Carlo computation, evolutionary algorithms, artificial neural networks and many other computational intelligence techniques require cheap randomisation. In many cases true randomness is both hard to obtain and not necessary. Instead a pseudo random number generator is used. Typically these are fast computer algorithms which dispense a sequence of seemingly unrelated numbers. These numbers are drawn from a distribution, such a uniform in the interval 0..1, Poisson, Gaussian, exponential and Cauchy. However the numbers are not truly random since the next number is chosen deterministically by the PRNG. (Deterministic behaviour has practical advantages for regression testing and debugging.)

However the history of pseudo random numbers on digital computers is full of poor implementations. Randomness is tricky. For example, in the 1960s IBM's mainframes were supplied with the infamous [3, page 1194] library function RANDU, which Knuth described as "really horrible" [1, page 1973]. Similarly even IBM describes the "randomness" of its AIX `rand` subroutine as "somewhat limited" [2]. Despite many reports of poor PRNG practise [3, 4], implementation problems continue to dog PRNGs. For example Peter Ross reports [5] limited randomness in `random.c` (as sold by Sun in Solaris 2.6).

In parallel implementations of random generators, we also need to consider now to ensure processes running in one computational stream appear random with respect to the same computation occurring on another processing node [6]. Its often sufficient to seed the initial state of the PRNG with different values on each processing stream.

The absence of high precision integer arithmetic in current generation GPUs makes implementing PRNGs tricky (e.g. [7, 8]). Indeed it has been widely regarded as impossible. [9, Sect 3.4] suggests using a "linear congruential generator" but no code is available and no test or performance results are given. The four functions (`ran1 ran2 ran3 ran4`) given in Numerical Recipes in C [10] require `long` integer data types and so are not suitable for use on current GPUs. Evolutionary computation is famously tolerant of bugs and may yield good results despite them.

As Knuth says "The moral of this story is that random numbers should not be generated by a method chosen at random" [1, page 5]. Therefore we chose not to implement a new random number generate but instead to implement Park and Miller's minimal standard PRNG [3].

```
int intrnd (int& seed) // 1<=seed<m
{
int const a    = 16807;      //ie 7**5
int const m    = 2147483647; //ie 2**31-1
seed = (long(seed * a))%m;
return seed;
}
```

Figure 1: park-miller.cc long int implementation. Multiplication and modulus are used to return a randomised version of the input. By careful choice of `a` and `m` Park and Miller produce an apparently random sequence of integers which uniformly samples the first $2^{31}-2$ integers without repeating any [3].

2 A C/C++ Park-Miller Implementation

Park and Miller included both a Pascal implementation and validation test results in [3]. In 1994 we implemented Park-Miller in C/C++ `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/random-numbers` (see Figure 1).

Figure 1 shows the algorithm is quite simple and consequently fast. (A modern Linux PC can generate 27 million random numbers per second.) However the `long int` in Figure 1 is where the GPU implementation difficulties arise. A minimum of 46 bits of precision are needed. (If one were to implement Matsumoto's Mersenne Twister at least 19 968 bits would be needed.) Today's generation of GPUs do not even have true integer arithmetic, instead all operations are actually performed in single precision floating point numbers. 64 bit operation and a wider range of data types may well be included in the next generation of GPUs. However current GPUs do provide very rapid floating point operations on short vectors of floating point values. This is built into GPUs for manipulating up to four floating point values. (Vectors may be used for red, green, blue and alpha components of colours. Short vectors can also be used to hold positions and directions in three dimensional simulations). RapidMind provides Value4f and Value3f C++ types.

Figure 2 shows a single precision floating point implementation of Park-Miller. This can be run on a normal computer but is intended as a stepping stone to a true GPU implementation. On a Linux PC, the more complicated code is approximately 12 times slower than that shown in Figure 1. Figures 2 and 3 are given to explain the algorithm used to avoid errors due to floating point lack of precision. If double precision is available Park-Miller can be implemented directly, cf. `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/random-numbers`. Figures 2 and 3 may also be useful to others who do not use RapidMind but instead wish to implement random numbers on their graphics hardware, e.g. using Cg, Brook or CUDA.

The first part of Park-Miller is unchanged, `exactmul()` multiplies the input by `a` (to give `temp`). The next step is to reduce this product modulo `m`. However to avoid integer division inherent in the C modulus operator `%` we replace it by approximate floating point division. This is used as a first guess for the largest integer multiple of `m` which does not exceed `temp`. Since `approxdiv` is the result of a floating point calculation it may be inaccurate. The `do` loop is used to refine the estimate using exact multiplication. Since `approxdiv` may also overestimate, the following `while` loop is used to reduce the multiple of `m` until the first exact multiple of `m` less than `temp` is reached. In fact the last call of `exactsub()` actually calculates the required remainder. Finally the 4 bytes of the answer are combined into a 31 bit integer value.

`exactmul()` multiplies a 31 bit positive integer by a non-negative integer no bigger than 16807. The smaller number can be represented accurately by a single floating point number (needing only 15 bits). However the 31 bit number is split into 4 bytes. The multiplication is carried out in four steps. Each requires no more than 23 bits of precision. At each step the carry is moved to the next float up. The carry on the last step will not exceed 15 bits and so can be reliably stored in a fifth single precision float.

`comp()` is used to compare both the 4 byte and 4 byte+1 float numbers. Since each float actually stores an integer value `>` and `<` can be safely used.

`exactsub()` is also used with both 4 and 5 element numbers. For simplicity `intrnd()` is arranged so that `exactsub()` only deals with non-negative numbers. This makes it easy to borrow (carry down) when subtracting a byte holding a bigger value from one holding a smaller one. Also `intrnd()` ensures the result will fit into 4 bytes.

```
int intrnd (int& seed) // 1<=seed<m
{
  float in[4];
  int t=seed;
  for(int i=0;i<4;i++) {in[i]=t&255; t=int(floor(t/256));}

  float const a    = 16807;      //ie 7**5
  float const m    = 2147483647; //ie 2**31-1
  const float M[4] = {255,255,255,127};
  float Seed[4];
  float temp[5];
  float prod[5];
  exactmul(16807,in,temp); //exact multiply seed*a;
  float approxdiv  = floor(float(seed)*a/m);
  approxdiv++;
  do {
    approxdiv--;
    exactmul(approxdiv,M,prod); //prod = exact multiply approxdiv*m;
  } while(comp(5,temp,prod)<0); //decrease prod until prod <= temp

  exactsub(5,temp,prod,Seed); //seed = temp - prod; (cannot be negative)
  while(comp(4,Seed,M)>=0) {  //decrease until 0<=seed<m
    exactsub(4,Seed,M,Seed);  //seed=seed-m;
  }
  seed=int(Seed[0]);float p=256;
  for(int i=1;i<4;i++) {seed += int(Seed[i]*p); p *= 256;}
  return seed;
}
```

Figure 2: park-miller.cc (single precision implementation). `a` and `m` have the same meaning as before, cf. Figure 1. `M` is `m` represented in 4 byte format. The first step of Park-Miller is unchanged, we multiply the input by `a`. To ensure this can be done in single precision arithmetic we split the input into its four constituent bytes. Thus `exactmul()` must be able to represent $16807 + 16807 \times 255 = 4302592$ exactly. The modulus operation is replaced by repeated multiplication or subtraction until `comp()` reports the remainder has been reached. See text.

```
inline void exactmul(const float f,const float in[4],float out[5]) {
  assert(f<=16807);
  out[0]=0;
  for(int i=0;i<4;i++) {
    const float t=in[i]*f;
    out[i]  += t;
    out[i+1] = floor(out[i]/256);
    out[i]   = int(out[i])%256;
  }
}
inline float comp(const int len,const float a[], const float b[]) {
  for(int i=(len-1);i>=0;i--) {
    if(a[i]>b[i]) return +1;
    if(a[i]<b[i]) return -1;
  }
  return 0;
}
inline void exactsub(const int len,const float a[], const float b[], float out[4]) {
  //nb a>=b
  float A[5];memcpy(A,a,len*sizeof(float));
  float B[5];memcpy(B,b,len*sizeof(float));
  for(int i=0;i<len;i++) {
    if(a[i]<B[i]) {assert(i<len);A[i]+=256; B[i+1]++;} //borrow if need be
    const float t = A[i]-B[i];                         //a-b
    if(i<4) { assert(0<=t && t<256); out[i] = t; }
    else    { assert(t==0); }
  }
}
```

Figure 3: park-miller.cc (single precision implementation support routines). `exactmul()`, `comp()` and `exactsub()` simulate long integer arithmetic using single precision floats. Each assumes its array inputs represent a positive integer. The four least significant bytes of which are stored in the last four floats of the array. `comp()` and `exactsub()` may act either on 4 bytes or 5 component arrays. The maximum value of the fifth component is 16741. The routines assume they are only used as part of `intrnd()` cf. Figure 2, and for speed do not comprehensively validate their inputs.

```
inline void Park_Miller(Value4f& Seed) {
#define nul comp=comp
  float const a    = 16807;      //ie 7**5
  float const m    = 2147483647; //ie 2**31-1
  const Value4f M(255,255,255,127);
  Value<5,float> temp;
  Value<5,float> prod;
  const Value1f seed = Seed(0) +
                       Seed(1)*Value1f(256) +
                       Seed(2)*Value1f(256*256) +
                       Seed(3)*Value1f(256*256*256);

  exactmul(a,Seed,temp); //exact multiply seed*a;
  Value1i approxdiv  = floor(seed*a/m);
  Value1i comp = -1; //loop at least once
  FOR(nul,comp<0,nul) {
    exactmul(Value1f(approxdiv),M,prod); //prod = exact multiply approxdiv*m;
    comp=comp5(temp,prod);
    approxdiv--;
  }ENDFOR
  exactsub5(temp,prod,Seed); //seed = temp - prod;
  FOR(nul,comp4(Seed,M)>=0,nul) {
    exactsub4(Seed,M,Seed); //seed=seed-m;
  }ENDFOR
#undef nul
}
```

Figure 4: RapidMind C++ glsl implementation of Park-Miller. `a`, `m`, `M`, `temp`, `prod` and `approxdiv` have the same meaning as before, cf. Figure 2. Since `int` is not available on the GPU but `Value4f` is, the precise four byte format is used both when calling the random number generator and for its return value. Therefore the single precision representation of the input, `seed`, must be calculated before `approxdiv` is needed. The RapidMind `FOR` loops are equivalent to the `do` and `while` loops shown in Figure 2 but are executed by the GPU's stream processors. On the GPU the single `comp()` and `exactsub()` which were used with two different types of inputs are each replaced by two equivalent routines which deal with only `Value4f` or `Value<5,float>` types, cf. Figure 5.

3 A GPU Random Number Implementation

The GPU code, cf. Figures 4 and 5, follows directly from the CPU single precision floating point Park-Miller C/C++ implementation described in the previous section. A large measure of device independence is obtained by appearing to treat the GPU as if it had one processor per element of the user's data. The GPU itself breaks up the work into `NP` (cf. Figure 6) separate threads (or tasks) and schedules them across its stream processors. The programmer need not know if the GPU has 4, 16, 128, or more stream processors. Note each element of the vector (which may itself be a composite object, such as Value4f) is serviced by its own thread. So the number of simultaneous threads is equal to the number of seeds transfered as a unit.

The intention is that the random numbers will be both generated and used on the GPU, however Figure 6 sketches the test harness used to call the GPU code and return its results to the host CPU. The values returned are compared with the anticipated results.

4 RapidMind C++ Random Number Validation and Performance

The original (1994) and both the new floating point and the GPU implementations were validated using the method suggested by Park and Miller [3]. That is, by detailed comparison of the sequence of results they produced against the values given by Park and Miller in [3]. Also they were each run more than 100 million times and their results confirmed against those at `http://www.firstpr.com.au/dsp/rand31/rand31-park-miller-carta.cc.txt`

```
inline void exactmul(const Value1f f, const Value4f in, Value<5,float>& out) {
  //RM_DEBUG_ASSERT(f<= Value1f(16807));
  out[0]=0;
  for(int i=0;i<4;i++) {
    out[i]  += round(in[i]*f);
    out[i+1] = floor(out[i]/Value1f(256));
    out[i]   = round(Value1i(out[i])%256);
  }
}
inline Value1i comp4(const Value4f a, const Value4f b) {
  return cond(a[3]>b[3],Value1i(+1),
         cond(a[3]<b[3],Value1i(-1),
         cond(a[2]>b[2],Value1i(+1),
         cond(a[2]<b[2],Value1i(-1),
         cond(a[1]>b[1],Value1i(+1),
         cond(a[1]<b[1],Value1i(-1),
         cond(a[0]>b[0],Value1i(+1),
         cond(a[0]<b[0],Value1i(-1),Value1i(0)))))))));
}
inline Value1i comp5(const Value<5,float> a, const Value<5,float> b) {
  return cond(a[4]>b[4],Value1i(+1),
         cond(a[4]<b[4],Value1i(-1),
         cond(a[3]>b[3],Value1i(+1),
         cond(a[3]<b[3],Value1i(-1),
         cond(a[2]>b[2],Value1i(+1),
         cond(a[2]<b[2],Value1i(-1),
         cond(a[1]>b[1],Value1i(+1),
         cond(a[1]<b[1],Value1i(-1),
         cond(a[0]>b[0],Value1i(+1),
         cond(a[0]<b[0],Value1i(-1),Value1i(0)))))))))));
}
inline void exactsub4(const Value4f a, const Value4f b, Value4f& out) {
  //nb a>=b
  Value<5,float> A;
  Value<5,float> B;
  for(int i=0;i<4;i++) {A[i]=a[i];B[i]=b[i];} A[4]=0;B[4]=0;
  exactsub5(A,B,out);
}
inline void exactsub5(const Value<5,float> a, const Value<5,float> b, Value4f& out) {
  //nb a>=b
  Value<5,float> A;
  Value<5,float> B;
  for(int i=0;i<5;i++) {A[i]=a[i];B[i]=b[i];}
  for(int i=0;i<4;i++) {
    B[i+1] = cond(a[i]<B[i],round(B[i+1]+Value1f(1)),B[i+1]);
    A[i]   = cond(a[i]<B[i],round(A[i]+Value1f(256)),A[i]  );
    out[i] = round(A[i]-B[i]);
  }
  //A[5]==B[5]
}
```

Figure 5: RapidMind C++ glsl implementation support routines are equivalent to those given in Figure 3. To ensure precise results explicit `round` operations are used. Although RapidMind provides an IF for operation on the GPU stream processors, the `cond()` operation is regarded as better. The `for()` loops are a coding convenience. RapidMind expands them before the compiled code is transfered to the GPU. In `exactsub5` the two `cond()` statements only change to `B[i+1]` and `A[i]` if their first argument is true. (If its false they effectively do nothing.) Again `cond()` is seen as better than `IF`. Care must be taken to ensure only the correct arguments are passed to the code since almost no validation is provided.

```
park_miller = RM_BEGIN {
  InOut<Value4f> Seed;
  Value1i I;
  FOR(I=0,I<TIMES,I++) {
  Value1i J;
  FOR(J=0,J<TIMES2,J++) {
  Value1i K;
  FOR(K=0,K<TIMES3,K++) {
    Park_Miller(Seed);
  } ENDFOR
  } ENDFOR
  } ENDFOR
} RM_END

⋮

// Access the internal arrays where the data is stored
Array<1,Value4f> Seed(NP);
float* in = Seed.write_data();
for (int i = 0; i<NP; i++ ) {
  int t = park_miller_seed[i];
  if(!(0<t && t<2147483647)) {
    cout<<"Bad park_miller_seed i="<<i<<" "<<park_miller_seed[i]<<endl;
    exit(1);}
  for(int j=0;j<4;j++) {in[i*4+j] = t%256; t /= 256;}
}
Array<1,Value4f> Result;
Result = gpu->park_miller(Seed);
const float* result = Result.read_data();
```

Figure 6: Parts of the RapidMind test harness. The lower code fragment passes a vector of `NP` initial seed values to the GPU. (`NP` may be up to 4 million). The GPU's multiple stream processors simulate simultaneous parallel operation of NP processors. The code between `RM_BEGIN` and `RM_END` is run on the GPU for each of the NP simulated processors. For test purposes it calls `Park_Miller()`, the random number generator, many times. When the GPU has finished the last random number created by each (simulated) processor is returned to the host CPU via a vector containing `NP Value4f` values. Despite the single precision code's complexity, the GPU can run more than 40 times faster than the CPU.

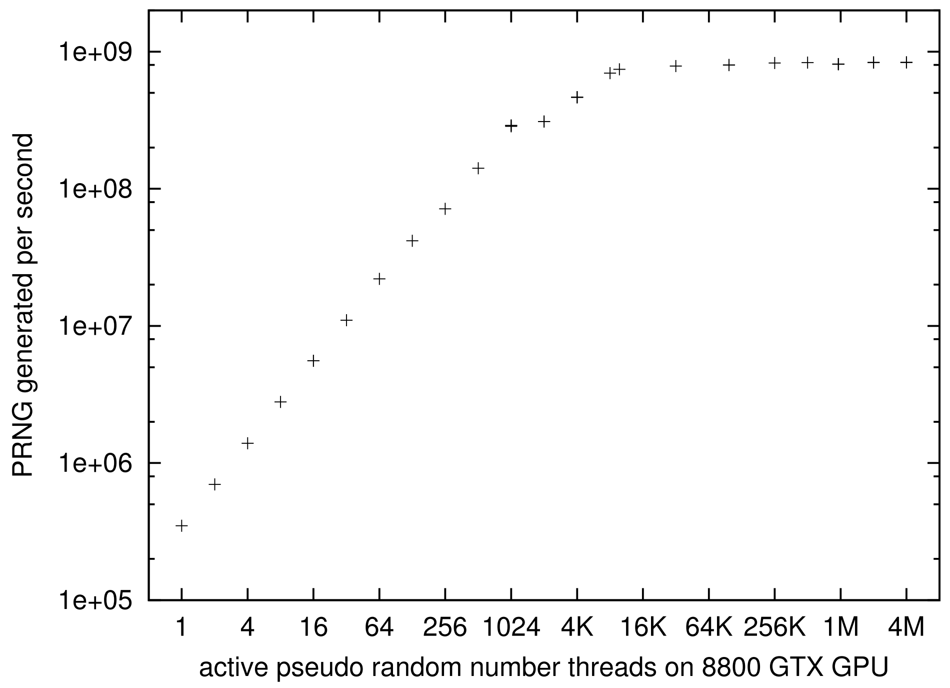

Figure 7: Park-Miller random numbers per second (excluding host-GPU transfer time) on nVidia 8800 GTX. In the test environment the rate depends upon how effectively the 128 parallel stream processors can be used. Only when there are more than 8192 separate threads do the 128 stream processors effectively saturate.

The following hardware and software were used for all timings. An unmodified high performance nVidia GeForce 8800 GTX GPU (VBIOS version 60.80.08.00.37) mounted inside a Linux 2.40 GHz Intel PC. The software versions were: RapidMind 2.01 OpenGL, nVidia driver 100.14.11 and GNU gcc 4.1.2 compiler. C++ code was compiled with `-O4` optimisation. RapidMind defaults were used. In particular the optimising GPU compiler was used at level 2.

With RapidMind at the maximum transfer size (4 million) the measured data transfer from the host to the GPU (via the PCI express 16X bus) was 798 MB/sec. As expected transfer back to the host is slower, and 277 MB/sec was measured.

For timing purposes the Park-Miller code was repeatedly called. From a practical point of view the GPU loop must not be made too long otherwise the operating system loses contact with the GPU requiring a reboot. In the current configuration (Ubuntu 4.1.2 KDE 3.5.6) mal-operation occurs after about 6 seconds, in Gentoo 3.4.6-r1 it was about 15 seconds. Hence the timing loops were adjusted to feed work to the GPU in about 3.5 second units.

The rate of generating random numbers was repeatedly measured at each of a number of data transfer sizes (from 1, 2, 4, ... 4 million) and the average value taken. As expected the peak rate is achieved with the largest data transfer size, cf. Figure 7.

When 4 million random numbers are returned to the host CPU for validation as a unit, an average of 73 billion numbers were created in 93 seconds. In testing mode data is transferred to/from the host. Using the data rates given above, the estimated time taken for data transfers is 4.6 seconds. For the GPU the data transfer costs should be excluded, yielding an average speed for the GPU of 833 million random numbers per second. In contrast, since we aim to run computational intelligence techniques on the GPU, the costs of transferring random numbers generated by the CPU onto the GPU must be include. Even so the CPU still generates and transfers on average 19 million random numbers per second. I.e. the GPU is 44 times faster. (If we compare with the same, single precision, algorithm running on the CPU, then the GPU is more than 400 times faster than the CPU. The fact that the GPU's power consumption is modest also suggests the host connection is also inhibiting the GPU and still higher rates might be achieved.)

We estimate in the region of 106 floating point operations are needed per random number. This suggests the GPU is delivering in the region of 90 GFLOPS. I.e. about 17% of the 518.4 Giga FLOPs claimed by the manufactures.

5 Discussion

As Figure 7 shows even after the time taken to transfer initial seeds to the GPU and random numbers back to the host is excluded, performance varies strongly with the work unit size. When smaller units are used more time will be required to reschedule both the Unix user process on the CPU and work on the GPU. Previously we estimated this at about 300μS. This is negligible compared to the typical work unit of 3.5 seconds used here.

We suggest the variation of performance with number of threads (seen in Figure 7) may be mostly due to memory access latency in the GPU. In nVidia's G80 architecture, the GPU's main memory is housed in separate memory chips and is shared between all the stream processors. Although fast RAM, fast access buses and multiple caches are used, in the GeForce 8800 there is a delay of up to about 300 clock ticks between requesting data and being able to use it. Rather than letting the stream processor be idle, the GPU will schedule another thread. In effect multi threading is needed to conceal memory latency.

When Park-Miller is used as part of a GPU application the random number seed will only be used on the GPU and will probably be discarded when the application thread is finished. Hence the seed need never by either read or written to the GPU's RAM and will probably always reside in each stream processor's L1 cache. Thus in GPU applications random numbers may not hit RAM latency problems and so should run even faster than the 833 million per second reported in Section 4.

In most applications it will be necessary to seed each parallel thread separately [6]. In many cases it will be convenient and sufficient to use a single seed value for the whole application and derive individual thread seeds from it. This will mean pseudo random numbers used in one part of the program will in fact not be independent. How important this is will, of course, depend upon the application and how it uses its random numbers. However this problem is not new and must also be overcome in non-parallel implementations.

A simple way to create an individual seed per thread is by adding the thread number (e.g. derived from a RapidMind `grid Array`) to the master seed. However this means each thread's initial seed is only one different from its neighbours. Depending on how the pseudo random numbers are used, this may not be sufficient. However each succeeding random number becomes more distinct from the corresponding random number in the neighbouring thread. For most practical purposes, it should be sufficient to initialise using a master seed+thread number and simply discard the first 3 random numbers. (NB. legal values of Park-Miller seeds are between 1 and $2^{31}-2$. Also the seeding calculations must not exceed the GPU's floating point precision.)

6 Conclusions

We have described a fast GPU implementation of a pseudo random number generator, meeting Park and Miller's minimum recommendations [3]. It has been implemented in RapidMind's platform and demonstrated on a high end nVidia GPU. The code is available via anonymous ftp from `cs.ucl.ac.uk/genetic/gp-code/random-numbers/gpu_park-miller.tar.gz`

The algorithm should be suitable for implementation in other GPU languages such as Cg, Brook and CUDA. Bench marking the C++ code, shows operation on the GPU is at least 44 times faster than running Park-Miller on the host CPU and transferring pseudo random numbers to the GPU.

Acknowledgment

I would like to thank Tim Czyrnyj.

References

[1] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Addison-Wesley, 1981, vol. 2 Seminumerical Algorithms.

[2] IBM, `http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic= /com.ibm.aix.basetechref/doc/basetrf2/rand.htm`, 2007.

[3] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find," *Communications of the ACM*, vol. 32, no. 10, pp. 1192–1201, Oct 1988.

[4] K. Entacher, "A collection of selected pseudorandom number generators with linear structures," Dept. of Mathematics, University Salzburg, Austria, Tech. Rep., 13 January 1998.

[5] P. Ross, `http://www.dcs.napier.ac.uk/ peter/` 2007.

[6] "A brief on parallel random number generators," National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, USA, White paper, 15 Sep 2005.

[7] P. Warden, “Random numbers in fragment programs,” 10 May 2005, `http://petewarden.com/notes/archives/2005/05/random_numbers.html`, accessed 24 Nov 2007.

[8] T.-T. Wong, M.-L. Wong, and K.-L. Fok, “Why current GPU is no good for high-quality random numbers generation?” `http://www.cs.cuhk.edu.hk/ ttwong/software/ecgpu/ecgpu.html`, epgpu version 0.99. Accessed 21 Nov 2007.

[9] Q. Yu, C. Chen, and Z. Pan, “Parallel genetic algorithms on programmable graphics hardware,” in *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, ser. Lecture Notes in Computer Science, L. Wang, K. Chen, and Y.-S. Ong, Eds., vol. 3612. Changsha, China: Springer, Aug. 27-29 2005, pp. 1051–1059.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1992.