The Journal of Supercomputing (2019) 75:3866–3881
https://doi.org/10.1007/s11227-019-02756-2

An OpenCL library for parallel random number generators

Tadej Ciglarič[1] · Rok Češnovar[1] · Erik Štrumbelj[1]

Published online: 17 January 2019

Abstract
We present a library of 22 pseudo-random number generators on the GPU. The library is implemented in OpenCL and all generators are tested using the TestU01 and PractRand libraries. We evaluated the efficiency of all generators on five different computing devices. Among the generators that pass all tests, Tyche-i was the fastest on most devices and on average. Tyche-i and several other generators from our library can be used to generate random numbers several times faster than generators from existing libraries.

Keywords Pseudo-random number generation · Parallelization · GPU · OpenCL · TestU01 · PractRand

1 Introduction

Parallelization is an effective option for reducing the running time of computationally intensive algorithms. However, to effectively parallelize stochastic computationally intensive algorithms, such as Monte Carlo methods, genetic algorithms, or simulations of stochastic processes, we need to be able to generate random numbers in parallel. Consequently we need a parallel implementation of a random number generator (RNG).

Most programming languages already implement an efficient and sufficiently random RNG in their standard libraries. However, these implementations are sequential. Libraries with parallel implementations exist, but only a few can be run on a graphics processing unit (GPU) [2, 10, 12, 18, 20, 24] [https://developer.nvidia.com/curand] and each library implements at most a few RNGs. A user who requires a parallel RNG in his algorithm has to, in most cases, implement that RNG or is forced to include an entire RNG library that implements it.

✉ Tadej Ciglarič
tc2922@student.uni-lj.si

1 Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, Ljubljana, Slovenia

It is also not clear which parallel RNG is the most efficient or which RNGs are too flawed for practical use. Quality and performance of sequential RNGs has been extensively evaluated [8]; however, the only evaluation of GPU implementations we have found compares a small number of RNGs on a single GPU [12] and most of those RNGs have known flaws [8]. We are not aware of any comparison of RNGs across different GPUs and CPUs.

The main goal of our research was to prepare a library based on a general approach to parallelizing RNGs and a set of parallel implementations of different RNGs, which users can easily include in their algorithms. Additionally, we performed several experiments that provide insights into the effectiveness of the RNGs and the practical usefulness of the sequences of numbers that they generate.

1.1 GPUs and OpenCL

GPUs are powerful parallel processing units. If an algorithm can be effectively parallelized, it will usually run significantly faster on a GPU compared to a CPU, especially if the algorithm is computationally complex.

Our implementation is made with OpenCL which allows the use of the same application on a multi-core CPU as well as on a many-core GPU [25, 26]. In OpenCL, functions can be run on hundreds or thousands of threads in parallel on a GPU. These functions are termed kernels. The host (CPU) runs the host program, these can be written in C, C++, Python, etc. This host program initializes the compute device, copies data to its memory (if needed) and sets parameters of execution. The most important parameters are the number of created threads and their organization in work groups.

2 RandomCL library

We implemented a library that contains 22 RNGs. The library is header-only and can be used on any operating system that supports OpenCL. The RNGs from the library can be executed on any OpenCL-enabled CPU or GPU, regardless of device vendor. The library is available at https://github.com/bstatcomp/RandomCL under the BSD-3 license.

All the RNGs can generate random numbers in the following formats: unsigned 32-bit integers, unsigned 64-bit integers, 32-bit floating-point numbers, or 64-bit double precision floating-point numbers. Integers are generated between 0 and a generator-dependent upper bound. Floating-point RNGs generate numbers between 0 and 1.

The typical use of the library consists of the following steps. First, random seeds are generated for each thread using a sequential generator from a standard library. Seeds are then copied to the computational device's global memory. Next, the OpenCL kernel is run on a device. It first initializes one generator for each thread using the previously generated seeds. Finally, the stochastic application's kernel calls the RNG function when random values are needed. This way random numbers are generated during the execution of the stochastic algorithm/application.

The library also supports generating random numbers in batches beforehand. In this case, a simple kernel is first run to generate random numbers and save them to the device's global memory. The algorithm that requires the random numbers is run afterward in a separate kernel and can access the pre-generated random number from global memory. However, if the algorithm requires many random numbers, we expect this option to be slower since generating random numbers can be significantly faster than loading them from the slow global memory on most devices.

2.1 Implemented random number generators

A pseudo-RNG is an algorithm that outputs a sequence of numbers that appears random. While deterministic, good RNGs generate a sequence of seemingly unpredictable numbers that can be used to simulate a random process.

In general, a random number generator consists of a state x, a state transition function f and an output function g. To generate a random number y, the state of generator is advanced $x_n = f(x_{n-1})$ first, before outputting $y_n = g(x_n)$.

In practice, the output function is usually simple, sometimes even the identity. For generators with a large state it often returns just a part of the generator state.

When choosing which RNGs to implement, we opted for some well-known RNGs, such as the linear congruential generator and Mersenne Twister. Other RNGs were chosen because they pass the tests from TestU01 library [8] and are relatively efficient. We implemented the following RNGs:

- *ISAAC (Indirection, Shift, Accumulate, Add, and Count)* [5] is a RNG intended for cryptographic purposes. We implemented *isaac*, but it does not work on graphics cards, because it requires unaligned memory access.
- *KISS (Keep It Simple, Stupid)* [13, 15] is a common name for three compound RNGs by the same author. We implemented the second, *kiss99*, proposed in 1999, and third—*kiss09*, proposed in 2009. Their components are LCG, xorshift and MWC generators. KIS99 is 32-bit RNG, while KISS09 is 64-bit RNG. Both pass BigCrush, even though none of their components do.
- *Lagged Fibonacci Generator* [16], defined by lags r, p and binary operation $*$ generates numbers according to equation $x_n = x_{n-r} * x_{n-p}$. If $*$ is addition, subtraction or exclusive-or, resulting generators are known to have poor quality [8]. We implemented a lagged Fibonacci generator *lfib* using multiplication.
- *Linear Congruential Generator (LCG)* [7] generates random numbers according to the equation $x_n = (x_{n-1} * a + b) \bmod m$, where a, b and m are parameters. If m is a power of 2, implementation is very simple and fast. LCGs are known as poor generators, especially if m is a power of 2, but they can still pass the BigCrush battery (a test suite, described in chapter 3.1) if only a part of state is returned [21]. We have implemented 128-bit LCG that returns the upper 64 bits (*lcg12864*) and 64-bit LCG and that returns the upper 32 bits (*lcg6432*). *Lcg6432* does not pass BigCrush [21].

- *Mersenne Twister* [17] is one of most popular RNGs. It is based on a large linear feedback shift register (LFSR) and a linear output function. However, it does not pass BigCrush. We implemented the Mersenne Twister *mt19937*.
- *Middle Square Weyl Sequence* [29] generates the next number by squaring the previous one before swapping the lower and upper half of its bits. Lastly, a number generated by a Weyl sequence [14] is added. Weyl sequence produces the next number by adding a constant to the previous one. We implemented the 64-bit middle square Weyl sequence *msws*.
- *Multiplicative Recursive Generator (MRG)* [6] of order k generates random numbers according to equation $x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m$, where a_i and m are parameters. It is one of the most commonly used parallel generators. We implemented three versions: *mrg31k3p* [9], *mrg32k3a* [6], and *mrg63k3a* [6].
- *Multiply With Carry (MWC)* [4] with state (x, c) and parameters a and m generates random numbers according to equation $x_n = (x_{n-1} * a + c_{n-1}) \bmod m$. In each step, it also updates c according to equation $c_n = \lfloor (x_{n-1} * a + c_{n-1})/m \rfloor$. A modification of MWC exists that instead of x returns $x + c$ in an attempt to improve the quality of the generator [27]. We implemented the modified MWC generator *mwc64x*.
- *Permutated Congruential Generator (PCG)* [21] combines a LCG and a non-trivial output function. Multiple versions with different output functions exist. We implemented a 64-bit generator that returns 32-bit numbers *pcg6432*. To generate a random number LCG is advanced, the state is shifted and xor-ed with the unshifted state. Then the uppermost four bits of the result determine which 32 bits are returned.
- *Philox (Product HIgh LOw Xorshift)* [24] is a counter-based RNG. That means that its state transition function is just an increment, while the output function is more complex. It can be even used without storing a state, just by applying its output function to some other variable in the algorithm it is used in, such as a loop counter. It is based on ideas of cryptographic block cyphers—using multiple rounds of a bit-scrambling operation. We implemented a 10-round Phylox RNG that works on two 32-bit numbers *philox2x32-10*.
- *Ran2* [23] combines two 32-bit LCGs. A table is used to save some results of the first LCG. The result of the second LCG is combined with a randomly chosen previous result of the first LCG. We implemented *ran2*.
- *Tiny Mersenne Twister* [18] is a smaller version intended for situations where not much memory can be used for storing generator state, for example, on GPUs. The original implementation is already compatible with OpenCL. We only modified the interface and initialization to match other generators in RandomCL. There is 32-bit version *tinymt32* and 64-bit version *tinymt64*.
- *Tyche* [19] is a random number generator based on a quarter round function of the ChaCha cypher. Tyche-i uses state transition function that is the inverse of Tyche's. This allows it to exploit instruction-level parallelism of modern processors to be slightly faster. We implemented both *tyche* and *tyche_i*.
- *WELL (Well-Equidistributed Long-period Linear)* [22] were created as an improvement to Mersenne Twister. While it has some nice theoretical properties, it still fails

some tests in BigCrush. We implemented the smallest version of the generator with 512-bit state *well512*.

- *Xorshift* [14] generates a random number from the previous number by shifting it and xor-ing it with the unshifted version three times, using a different shift each time. Xorshift has been shown to be mathematically equivalent to a linear feedback shift register (LFSR) generator [1]. The 64-bit xorshift does not pass BigCrush on its own [8]. We implemented a 1024-bit xorshift generator *xorshift1024* [12]. Its state is advanced jointly by 32 threads.
- *Xorshift** [28] is an xorshift generator with a non-trivial output function—a multiplication with an constant. We implemented a 64-bit generator that returns 32 bits of its state *xorshift_star*. That makes it pass BigCrush [21].

2.2 Parallelization

There are several different approaches to generating random numbers in a parallel algorithm. We use random initialization—each thread has its own instance of a generator, initialized to a random state. While this is efficient and applicable to any generator, it is possible that the generated streams overlap. However, for generators with sufficiently long periods, the probability of overlap is negligible [11]. All generators we implemented have a period of at least 2^{64}, so the probability of overlap is small. It has been shown for LCGs that random initialization gives better quality of generated numbers compared to equally spaced substreams [12].

There are several possible alternatives to our approach. A trivial alternative would be to generate a sequence of random numbers sequentially—possibly in advance. However, this approach is slow as it does not scale with the number of threads.

Next, we could use a different generator for each thread. Same algorithm with a different parameter set for each thread would suffice. However, parameter sets that produce streams of good quality exist only for a few RNGs. Even if the quality of all streams is good, they must be tested for independence [11].

If we have T threads, each with a single instance of a generator, we can initialize generators with T sequential states. Before using the output function to generate a number, the generator is advanced not for 1, but for T states. However, for most generators jumping ahead by multiple states is significantly slower than just advancing the state by one.

We could also split the stream of numbers into T substreams of (almost) equal length and initialize each generator to the first state in different substreams. This is realized by initializing all generators to the same state before advancing them for an appropriate number of steps. However, efficient jumping for many steps is only possible for a few RNGs and advancing one step at a time would be too time-consuming to be practically feasible.

2.3 An example of how to use a RNG

Listing 1 shows how to fill an array in an OpenCL kernel with random numbers using a RNG from the RandomCL library. It uses the *tyche_i* RNG to generate 32-bit unsigned integers.

Line 1 includes the header file with the implementation of the RNG.

Lines 3–5 contain the kernel function header. This is the function that can be called from the host and executes in parallel on the device. It accepts three arguments. The first argument *num* sets the number of random values to generate. The second argument *seed* is a pointer to the array in global memory that contains seeds for initialization of generators. Since this example uses one generator per thread, the seed array must contain (at least) as many seeds. The third and final argument *res* is a pointer to an array in global memory, where the generated numbers will be stored.

Lines 6 and 7 determine the execution parameters: total number of threads *gsize* and index of the thread *gid*. Line 8 declares the variable *state* that stores the state of the RNG. Line 9 initializes the RNG of each thread with one of the seeds. Lines 10-12 generate random numbers and save them in the *res* array.

Listing 1 An example of how to use a RandomCL RNG

```
 1  #include <tyche_i.cl>
 2
 3  kernel void array(uint num,
 4                    global ulong* seed,
 5                    global uint* res){
 6      uint gid = get_global_id(0);
 7      uint gsize = get_global_size(0);
 8      tyche_i_state state;
 9      tyche_i_seed(&state, seed[gid]);
10      for(uint i = gid; i < num; i += gsize){
11          res[i] = tyche_i_uint(state);
12      }
13  }
```

3 Empirical evaluation

3.1 Testing quality: TestU01

TestU01 [8] is the most commonly used library for empirically testing the quality of RNGs and supersedes other popular libraries, such as Diehard, Dieharder and the NIST statistical test suite.

While statistical testing cannot prove a generator is good, it can be used to search for particular deficiencies. TestU01 defines three test batteries that determine the tests and their parameters. From fastest to most discriminative, they are SmallCrush, Crush, and BigCrush. Sequential implementations of most generators we implemented are known to pass BigCrush (exceptions are *lcg6432*, *mt19937*, *tinymt32*, *tinymt64*, and *well512*).

Depending on how they are used, random numbers generated in parallel might or might not be consumed in the same order as they are generated. If they are, the quality of the RNG is exactly the same as the quality of the sequential implementation of the same RNG. If they are not, this is effectively the same as permuting the order in which the numbers are generated. If each thread works on an independent part of the problem, numbers are consumed in the same order as generated, resulting in no permutation. However, if threads work jointly on the same part of problem, one number from each thread is consumed before the next number from first thread is consumed.

For example, we can take a simple case of generating random numbers and saving them in an array in memory. If this task is done with a sequential program, there is only one obvious way of ordering numbers. The i-th generated number is saved to the i-th place in the array. In parallel, however, there are two reasonable options. If we have T threads, each generating N numbers (for a total of NT numbers), the i-th number generated by the thread t can be saved at the index $Nt + i$ or $Ti + t$. The first option is similar to sequential generation of numbers. Each thread stores numbers generated in sequence in a contiguous part of array. In the second option, the consecutive numbers of the resulting array are generated by different threads.

These permutations could affect the quality of the generated stream of numbers. This is why we have tested the quality of parallel implementations which return permuted sequences. It is impossible to test permutations for all possible numbers of threads. We have selected 1024 as a representative number of threads and executed tests on that many.

TestU01 can only test 32-bit numbers. So we have tested 64-bit generators three times: the lower 32 bits of each number, the upper 32 bits and both the lower and the upper 32 bits as two consecutive 32-bit numbers.

3.2 Testing quality: PractRand

PractRand (Practically Random) is a C++ RNG library [3]. It includes a battery of statistical tests in the tradition of Diehard and TestU01, some of which detect statistical flaws that are not covered by TestU01. Two other important advantages of PractRand testing are multi-core computation support and that tests can easily be performed on relatively long sequences.

PractRand runs all tests on all the generated data. This is in contrast with the TestU01 test batteries SmallCrush, Crush, and BigCrush, where each test is performed on an independently generated data set and data size varies from test to test. To reduce computation times, PractRand tests start with a small data set (256 kB) and increase in increments of factor 2 to the maximum data size (in our case 2TB) or until a test fails.

Note that some RNGs generate only 31/63 bits (*lfib*, *mrg31k3p*, *mrg63k3a*, *ran2*). This does not pose a problem for TestU01, because it ignores the lowest bit of every 32-bit number. We modified testing with PractRand for these 4 generators so that the missing bit was ignored.

3.3 Testing speed

We tested the speed of the implemented RNGs on several different devices: AMD Radeon R7 260X (2013 mid-range gaming GPU), AMD Ryzen Threadripper 1950X (2017 high-end CPU), Intel Core i5-4690 (2014 mid-range CPU), Intel HD Graphics 4000 (2012 low-end integrated GPU) and NVIDIA GeForce GTX 1070 (2016 high-end gaming GPU).

The performance of a particular generator on a particular device can vary greatly with the number of threads used and how they are divided into work groups. We have made no attempt at finding optimal configurations. Instead, we used a simple heuristic to determine the number of threads that worked relatively well for all generators and devices. We set a number of threads per work group to 256 and number of work groups to 4 times the number of compute units on the device. In practice, RNGs are usually part of a larger program and it makes no sense to expect the number of threads to be optimized for performance of the RNG.

Some RNGs generate 32-bit numbers and some generate 64-bit numbers. To avoid the overhead of converting all numbers to either 64 or 32 bits, we tested 32- and 64-bit generators separately and report measured speed in gigabytes per second.

For comparison, we also include generators from other libraries that use OpenCL. Note that we did not include generators that are known to fail tests from the TestU01 library:

- The *Random123* library [24] implements various counter-based RNGs. The Phylox family of generators is designed for use on GPUs. We tested the generator *phylox2x32_10(random123)*, which is also implemented in our library. Other Phylox generators generate more than 64 random bits at once, which is impractical for most use-cases.
- The *clRNG* library [10] implements 4 generators and we tested three of them—*mrg32k3a(clrng)*, *mrg31k3p(clrng)*, and *phylox4x32_10(clrng)*. Both MRG generators are implemented in the library, while Phylox is just a wrapper around the implementation from Random123. It also implements *lfsr113*, which we did not test as it is known to fail some tests from the TestU01 library [8].
- The *RANLUXCL* library [20] implements *ranlux(ranluxcl)* generator, which we included in our tests.
- The *PRNGCL* library [2] implements 7 generators. We tested the two generators that pass testing with TestU01 library—*ranlux(prngcl)* and *mrg32k3a(prngcl)*. All other implemented generators are known to fail some of the tests. Compared to all other mentioned libraries (ours included), the generators from PRNGCL are not intended to be included in the user's kernel program. Instead, a separate kernel is run before random numbers are required. It generates random numbers and saves them in memory. This is how we used it when testing efficiency.
- The original *MWC64X* generator [27] is also implemented in OpenCL. Except for different initialization, its implementations are practically the same as ours so it produces numbers at same speed. That is why we did not list it separately in the table with results.

4 Results

From Table 1, we can see that parallel implementations of generators *isaac* and *mt19937* fail at least one of the tests from TestU01. That makes them unsuitable for general-purpose parallel RNGs. *kiss09*, *msws* and *lfib* also fail some, but could still be used. We can see that lower 32 bits of *kiss09* and *msws* and upper 32 bits of *lfib* still pass all tests so we could modify them to only return half of their state. However, that would effectively halve the speed at which they generate numbers.

Table 2 shows results of testing the generators with the PractRand library. Most of the generators that fail testing with the TestU01 library also fail with the PractRand. It also identifies problems with some generators that pass the testing with the TestU01 library. Only generators *isaac*, *kiss09*, *xorshift_star*, *tyche*, *tyche_i* and all three MRGs pass the testing.

We also tested the original implementation of MWC64X [27] that uses stream splitting and it failed 4 tests on both Crush and BigCrush while passing

Table 1 TestU01 quality tests results

Output	Generator	SmallCrush			Crush			BigCrush		
32-bit	kiss99			0			0			0
	lcg6432			0			0			0
	mrg31k3p			0			0			0
	mrg32k3a			0			0			0
	pcg6432			0			0			0
	tinymt32			0			0			0
	well512			0			0			0
	xorshift1024			0			0			0
	xorshift_star			0			0			0
	mwc64x			0			0			0
	isaac			0			1			0
	mt19937			1			1			0
64-bit	lcg12864	0	0	0	0	0	0	0	0	0
	mrg63k3a	0	0	0	0	0	0	0	0	0
	philox2x32_10	0	0	0	0	0	0	0	0	0
	ran2	0	0	0	0	0	0	0	0	0
	tinymt64	0	0	0	0	0	0	0	0	0
	tyche	0	0	0	0	0	0	0	0	0
	tyche_i	0	0	0	0	0	0	0	0	0
	kiss09	0	0	0	0	1	0	0	0	0
	lfib	8	0	6	70	0	53	52	0	40
	msws	0	0	0	0	10	4	0	26	10

For each generator, we report the number of failures on each test battery. 64-bit generators have three results for every battery—for lower 32 bits, upper 32 bits and both as two 32-bit numbers

Table 2 PractRand quality tests results

Generator	Sequential	Parallel
isaac	Pass	Pass
kiss09	Pass	Pass
mrg31k3p	Pass	Pass
mrg32k3a	Pass	Pass
mrg63k3a	Pass	Pass
tyche	Pass	Pass
tyche_i	Pass*	Pass*
xorshift_star	Pass	Pass
kiss99	Pass	16 MB
lfib	512 MB	2 MB
lcg12864	128 GB	Pass
lcg6432	32 MB	32 GB
mt19937	512 GB	512 kB
msws	2 MB	128 MB
mwc64x	Pass*	1 TB*
pcg6432	Pass	256 kB
philox2x32_10	Pass	2 GB
ran2	16 GB	1 TB
tinymt32	16 MB	128 MB
tinymt64	32 MB	256 MB
well512	16 MB	1 GB
xorshift1024	128 MB	8 GB

We tested each generator several times, for output sizes ranging from 256 kB to 2TB with increments of factor 2. If the generator failed, we report the lowest data size at which it failed. (*) Due to computational constraints, we tested only the two most efficient generators *tyche_i* and *mwc64x* up to the PractRand default output size of 32TB

the testing with the PractRand library. Therefore, our approach to initialization improves some statistical properties of this generator and degrades others.

To compare speeds of a generators across devices, we define rs_{gd}—the *relative speed* of a particular generator g on a particular device d—as the quotient between the speed of the generator on that device s_{gd} and the speed of the fastest generator implementation on the same device

$$rs_{gd} = \frac{s_{gd}}{\max_i s_{id}}.$$

Time measurement results are shown in Table 3. For each generator, we also report its *average relative speed* and its *worst relative speed* across all devices. These two summarize the generators expected average and worst-case performance, relative to the best generator, respectively. The worst relative speed represents the generator's robustness. A generator can achieve a high average relative speed by being

Table 3 *Performance tests results*

Generator	Speed average relative	Speed worst relative	AMD Radeon R7 260X	AMD Ryzen Thread-ripper 1950X	Intel Core i5-4690	Intel HD Graphics 4000	NVIDIA GeForce GTX 1070
msws	0.863	0.583	327.32	141.07	152.22	29.32	1626.22
			± 0.09	± 1.94	± 3.16	± 0.62	± 11.89
tyche_i	**0.722**	0.157	545.85	**112.69**	23.97	**22.59**	**1922.77**
			± 0.10	± 1.95	± 0.76	± 0.51	± 59.95
lcg6432	0.635	0.312	175.44	111.57	114.62	33.16	613.61
			± 0.04	± 2.77	± 3.27	± 0.92	± 4.26
tyche	0.569	0.110	**561.44**	70.04	16.68	20.46	1194.78
			± 0.12	± 0.75	± 0.37	± 0.54	± 8.95
mwc64x	0.565	0.504	283.08	75.65	90.86	19.94	1128.12
			± 0.05	± 1.51	± 2.32	± 0.48	± 8.29
kiss09	0.519	0.056	353.18	97.76	127.88	1.87	726.84
			± 0.07	± 1.33	± 4.10	± 0.10	± 4.73
xorshift_star	0.442	**0.246**	278.86	61.62	**75.61**	17.64	472.72
			± 0.05	± 1.13	± 1.64	± 0.54	± 3.44
tinymt64	0.437	0.331	234.76	53.97	51.77	23.60	637.21
			± 0.07	± 0.81	± 1.73	± 2.10	± 4.59
pcg6432	0.403	0.165	92.91	63.01	101.36	17.43	404.06
			± 0.02	± 0.66	± 2.80	± 0.39	± 2.80
tinymt32	0.290	0.176	152.40	26.19	26.85	18.96	467.01
			± 0.03	± 0.48	± 0.87	± 0.79	± 3.27
kiss99	0.288	0.070	238.28	31.84	54.04	2.32	703.51
			± 0.04	± 0.26	± 1.74	± 0.11	± 5.02

Table 3 (continued)

Generator	Speed average relative	Speed worst relative	AMD Radeon R7 260X	AMD Ryzen Threadripper 1950X	Intel Core i5-4690	Intel HD Graphics 4000	NVIDIA GeForce GTX 1070
lcg12864	0.281	0.137	78.64	72.02	60.74	7.19	263.51
			± 0.02	± 1.72	± 1.76	± 0.20	± 5.02
philox2x32_10	0.192	0.089	81.92	18.40	26.78	13.95	170.88
			± 0.02	± 0.21	± 0.83	± 0.70	± 1.26
mrg31k3p	0.174	0.103	109.15	14.55	19.95	7.00	442.77
(clrng)			± 0.03	± 0.10	± 0.40	± 0.19	± 3.15
philox2x32_10	0.167	0.107	82.17	15.07	28.68	9.00	236.48
(random123)			± 0.02	± 0.26	± 0.88	± 0.25	± 1.94
mrg31k3p	0.144	0.051	109.38	14.96	21.17	1.69	442.58
			± 0.14	± 0.14	± 0.65	± 0.07	± 18.51
lfib	0.140	0.034	28.85	65.12	5.21	2.95	120.42
			± 0.02	± 1.16	± 0.10	± 0.10	± 0.33
xorshift1024	0.089	0.021	112.42	3.53	3.15	3.64	167.54
			± 0.03	± 0.05	± 0.08	± 0.12	± 2.03
isaac	0.078	0.027	/	18.18	4.14	/	/
				± 0.17	± 0.16		
well512	0.065	0.028	40.87	16.48	4.20	2.14	84.52
			± 0.01	± 0.33	± 0.13	± 0.07	± 0.52
mrg63k3a	0.060	0.003	14.84	13.43	22.70	0.11	49.14
			± 0.09	± 0.31	± 0.74	± 0.00	± 0.43
mrg32k3a	0.056	0.006	14.48	13.38	18.94	0.20	51.43
			± 0.03	± 0.26	± 0.48	± 0.00	± 0.44

Table 3 (continued)

Generator	Speed average relative	Speed worst relative	AMD Radeon R7 260X	AMD Ryzen Threadripper 1950X	Intel Core i5-4690	Intel HD Graphics 4000	NVIDIA GeForce GTX 1070
ran2	0.053	0.017	29.65	13.28	2.53	1.68	98.76
			±0.00	±0.13	±0.04	±0.05	±0.50
mrg32k3a	0.047	0.003	15.08	11.62	14.75	0.11	48.34
(clrng)			±0.02	±0.17	±0.59	±0.00	±0.39
philox4x32_10	0.046	0.013	32.47	10.42	1.98	1.52	78.59
(clrng)			±0.14	±0.07	±0.07	±0.11	±0.64
ranlux	0.045	0.013	12.22	17.58	3.07	/	24.48
(ranluxcl)			±0.01	±6.61	±1.59		±13.35
mt19937	0.032	0.005	16.64	12.39	2.42	0.18	43.90
			±0.06	±0.18	±0.06	±0.01	±0.13
ranlux	0.026	0.010	9.61	1.63	1.49	/	126.45
(prngcl)			±0.07	±0.03	±0.05		±13.77
mrg32k3a	0.012	0.002	6.20	1.64	0.37	/	42.90
(prngcl)			±0.02	±0.00	±0.01		±5.96

Generators are ordered by average relative speed. Absolute measurements are presented as the average value and the standard deviation in GB of generated random numbers per second. Our implementation of *isaac* does not work on GPUs as it requires unaligned memory access. Names of generators from our library that pass all tests in bold. For each column, the speed of the fastest generator that passes all statistical tests is in bold. We did not test generators from other libraries for statistical deficiencies particularly fast on one or a few devices and slow on others. To have a high worst relative speed, a generator must perform reasonably well on all tested devices.

5 Discussion and conclusion

For a parallel RNG to be as general as possible, it should pass statistical tests both when run in a single thread and in parallel—as explained in Sect. 3.1. Generators *tyche*, *tyche_i*, *xorshift_star*, and all three versions of *mrg* pass all the tests, while all the remaining generators fail the testing either when run sequentially or in parallel.

Different generators produce numbers at very different speeds. If we disregard generators that do not pass all the tests, *tyche_i* is on average the best generator. It is also the best on Intel and NVIDIA GPUs, AMD CPU and practically as good as the best generator on the AMD GPU. Its worst relative speed is low only due to its poor performance on the Intel CPU. Therefore, as a general-purpose generator that can run very fast on almost any device, *tyche_i* is strongly recommended. If we target a specific device, we can instead select a generator that has the best performance on the most similar device.

Usually, however, RNG is a part of a larger algorithm. Its speed depends on many factors that are affected by both RNG and the rest of the algorithm in a non-trivial way. To achieve the best possible performance, the speed of the algorithm should be measured while using some of the fastest generators to find which one works best for the particular case.

In some cases, we might also be able to use generators that fail some of the tests, as the algorithm itself might not be sensitive to deficiencies of a particular generator. However in that case, the algorithm should be tested for correctness while using each of the candidate generators. Such testing may be beneficial in general, as the algorithm may be sensitive to a deficiency that is not tested for in TestU01. The best worst relative speed is achieved by *msws*. While it does not pass PractRand and only the lower 32 bits pass BigCrush, the lower 32 bits could still be used, resulting in a generator that is very robust in terms of speed across different devices.

The RandomCL library and the work presented in this paper provides users with a library of RNGs that the users can easily include in their algorithms. All the RNGs have been thoroughly tested, providing the user with information and guidance on both the statistical properties and the efficiency of each individual RNG. Furthermore, the best generators from the library generate random numbers 4 times faster on average, compared to the best generators from existing libraries.

As part of future work, we will test and implement other RNGs. Two particularly interesting avenues of research would be to (a) implementations that can be split into substreams of equal length or ones that use different parameter sets for each thread and (b) to modify the Middle Square Weyl Sequence RNG so that it passes all tests.

Acknowledgements Our research was partially supported by the Slovenian Research Agency (ARRS project grant L1-7542 and research core funding P5-0410).

References

1. Brent RP (2004) Note on Marsaglia’s xorshift random number generators. J Stat Softw 11(5):1–4
2. Demchik V (2014) Pseudorandom numbers generation for Monte Carlo simulations on GPUs: OpenCL approach. In: Numerical computations with GPUs, Springer, pp 245–271
3. Doty-Humphrey C (2018) PractRand: C++ library of pseudo-random number generators and statistical tests for RNGs. http://pracrand.sourceforge.net/PractRand.txt. Accessed 20 Dec 2018
4. Goresky M, Klapper A (2003) Efficient multiply-with-carry random number generators with maximal period. ACM Trans Model Comput Simul 13(4):310–321
5. Jenkins RJ (1996) ISAAC. In: Gollmann D (ed) International workshop on fast software encryption. Springer, Berlin, pp 41–49
6. L’Ecuyer P (1999) Good parameters and implementations for combined multiple recursive random number generators. Oper Res 47(1):159–164
7. L’Ecuyer P (1999) Tables of linear congruential generators of different sizes and good lattice structure. Math Comput Am Math Soc 68(225):249–260
8. L’Ecuyer P, Simard R (2007) TestU01: A C library for empirical testing of random number generators. ACM Trans Math Softw 33(4):22
9. L’Ecuyer P, Touzin R (2000) Fast combined multiple recursive generators with multipliers of the form a=±2 q±2 r. In: Joines JA, Barton RR, Kang K, Fishwick PA (eds) Proceedings of the 32nd Conference on Winter Simulation, Society for Computer Simulation International, pp 683–689
10. L’Ecuyer P, Munger D, Kemerchou N (2015) clRNG: A random number API with multiple streams for OpenCL. Report. http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf. Accessed 20 Sept 2018
11. L’Ecuyer P, Munger D, Oreshkin B, Simard R (2017) Random numbers for parallel computers: requirements and methods, with emphasis on GPUs. Math Comput Simul 135:3–17
12. Manssen M, Weigel M, Hartmann AK (2012) Random number generators for massively parallel simulations on GPU. Eur Phys J Spec Top 210(1):53–71
13. Marsaglia G (1999) Random numbers for C: End, at last?. http://www.cse.yorku.ca/~oz/marsaglia-rng.html. Accessed 20 Sept 2018
14. Marsaglia G (2003) Xorshift RNGs. J Stat Softw 8(14):1–6
15. Marsaglia G (2009) 64-bit KISS RNGs. https://www.thecodingforums.com/threads/64-bit-kiss-rngs.673657. Accessed 20 Sept 2018
16. Marsaglia G, Tsay LH (1985) Matrices and the structure of random number sequences. Linear Algebra Appl 67:147–156
17. Matsumoto M, Nishimura T (1998) Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul 8(1):3–30
18. Matsumoto M, Nishimura T (2011) Tiny Mersenne twister. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html. Accessed 20 Sept 2018
19. Neves S, Araujo F (2011) Fast and small nonlinear pseudorandom number generators for computer simulation. In: Wyrzykowski R, Dongarra J, Karczewski K, Waśniewski J (eds) International Conference on Parallel Processing and Applied Mathematics, Springer, pp 92–101
20. Nikolaisen IU (2011) Bose-einstein condensation in trapped bosons: a quantum monte carlo analysis using OpenCL and GPU programming. Master’s thesis
21. O’Neill ME (2014) PCG: a family of simple fast space-efficient statistically good algorithms for random number generation. ACM Transactions on Mathematical Software
22. Panneton F, L’ecuyer P, Matsumoto M (2006) Improved long-period generators based on linear recurrences modulo 2. ACM Trans Math Softw 32(1):1–16
23. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992) Numerical recipes in C: Plee art of scientific computing. Cambridge
24. Salmon JK, Moraes MA, Dror RO, Shaw DE (2011) Parallel random numbers: as easy as 1, 2, 3. In: Lathrop S, Costa J, Kramer W (eds) 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), IEEE, pp 1–12
25. Scarpino M (2011) OpenCL in action. Manning Publications, Westampton
26. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66–73
27. Thomas DB (2014) The mwc64x random number generator. http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html. Accessed 23 Dec 2018

28. Vigna S (2016) An experimental exploration of Marsaglia's xorshift generators, scrambled. ACM Trans Math Softw 42(4):30
29. Widynski B (2017) Middle square Weyl sequence RNG. arXiv preprint arXiv:170400358