Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Ciglarič

Vzporedno generiranje naključnih števil

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Erik Štrumbelj
Somentor: as. Rok Češnovar, univ. dipl. inž. rač. in inf.

Ljubljana, 2018

University of Ljubljana
Faculty of Computer and Information Science

Tadej Ciglarič

Parallel Random Number Generation

master's thesis

second-cycle study programme
computer and information science

Supervisor: Assoc. Prof. Dr. Erik Štrumbelj
Co-supervisor: Asst. Rok Češnovar, B. Sc.

Ljubljana, 2018

Povzetek

Predstavljamo knjižnico devetnajstih generatorjev psevdonaključnih števil. Generatorji so implementirani v programskem jeziku OpenCL in so namenjeni uporabi na grafičnih procesnih enotah. Večina implementiranih generatorjev prestane statistične teste kvalitete naključnosti generiranih števil iz knjižnice TestU01. Hitrost generiranja števil smo ovrednotili na petih različnih računskih napravah. Skupno najboljše rezultate dosega generator Tyche-i, vendar so za nekatere izmed naprav drugi generatorji boljši.

Abstract

We present a library of 19 pseudo-random number generators, implemented for graphical processing units. The library is implemented in the OpenCL framework and empirically evaluated using the TestU01 library. Most of the presented generators pass the tests. The generators' performance is evaluated on five different devices. The Tyche-i generator is the best choice overall, while on some specific devices other generators are better.

I would like thank to supervisor Erik Štrumbelj and co-supervisor Rok Češnovar for their insights, patience and guidance during the creation of this thesis. They have also lent me the hardware I needed for running the tests.

I would also like to thank my family and friends for their support during my studies.

Contents

List of Used Acronyms and Translations

acronym	English	slovensko
CPU	central processing unit	centralna procesna enota
GPU	graphics processing unit	grafična procesna enota
ISAAC	indirection, shift, accumulate, add and count (random number generator)	posrednost, pomik, kopičenje, seštevanje in štetje (generator naključnih števil)
KISS	keep it simple, stupid (random number generator)	pusti preprosto, neumnež (generator naključnih števil)
LCG	linear congruential generator	linearni kongruenčni generator
MRG	multiple recursive generator	večkratni rekurzivni generator
MT	Mersenne Twister (random number generator)	Mersenne Twister (generator naključnih števil)
OpenCL	open computing language (framework for programming various compute devices, including graphics processing units)	odprt računski jezik (ogrodje za programiranje različnih računskih naprav, med drugim grafičnih kartic)
PCG	permutated congruential generator	permutiran kongruenčni generator
Phylox	product high low xorshift (random number generator)	produkt zgornji spodnji xorshift (generator naključnih števil)

acronym	**English**	**slovensko**
RNG	random number generator	generator naključnih števil
WELL	well-equidistributed, long-period, linear (random number generator)	dobro enakomerno porazdeljen, linearen z dolgo periodo (generator naključnih števil)
/	kernel (function that is executed on compute device)	ščepec (funkcija, ki se izvaja na računski napravi)
/	compute device	računska naprava
/	compute unit	računska enota
/	host (computer that controls execution of kernels on a compute device)	gostitelj (računalnik, ki nadzira izvajanje ščepcev na računski napravi)
/	lagged Fibonacci generator	zamaknjen Fibonaccijev generator
/	tiny Mersenne Twister (random number generator)	majhni Mersenne Twister (generator naključnih števil)
/	middle square Weyl sequence (random number generator)	metoda sredine kvadrata z Weylovim zaporedjem (generator naključnih števil)

Razširjeni povzetek

Uvod

Če želimo učinkovito vzporedno implementirati stohastični algoritem, potrebujemo tudi vzporedno implementacijo generatorja naključnih števil. Primeri takih algoritmov so metode Monte Carlo, genetski algoritmi in simulacije stohastičnih procesov.

Žal ni veliko vzporednih implementacij generatorjev naključnih števil, še manj pa jih omogoča izvajanje na grafičnih procesnih enotah [1, 2, 3, 4] . Primerjava učinkovitosti generatorjev naključnih števil pa je bila v preteklosti izvedena le na manjšem številu generatorjev in samo eni računski napravi [2].

Uporabnik, ki v svojem vzporednem algoritmu potrebuje naključna števila, mora v večini primerov generator implementirati sam ali pa je prisiljen, da v svojo rešitev vključi celotno knjižnico, ki generator implementira. Še več, ni niti jasno, kateri vzporedni generator je najbolj smiselna izbira z vidika učinkovitosti in kateri generatorji so preslabi za praktično rabo.

Glavni cilj te magistrske naloge je bil pripraviti knjižnico z naborom različnih vzporednih implementacij generatorjev, ki jih lahko uporabniki enostavno vključijo v svoj algoritem. Obenem smo želeli s poskusi na različnih napravah ponuditi vpogled v njihovo učinkovitost in praktično uporabnost nizov naključnih števil, ki jih generirajo.

Knjižnica RandomCL

Knjižnico generatorjev naključnih števil, ki jo imenujemo RandomCL, smo implementirali v ogrodju OpenCL. Ta omogoča, da vzporedni del programa, t.i. ščepec

(angl. *kernel*), napišemo v programskem jeziku, ki je podoben jeziku C. Ščepec se lahko vzporedno izvaja na katerikoli računski napravi, ki podpira ogrodje OpenCL, kar vključuje večino procesorjev in sodobnih grafičnih kartic.

Tipična uporaba knjižnice RandomCL je sestavljena iz več korakov. Najprej je potrebno generirati naključna semena (angl. *random seed*) s sekvenčnim generatorjem. Ta se kopira v pomnilnik računske naprave, na kateri se nato požene ščepec. Slednji uporabi semena za inicializacijo generatorjev – običajno za vsako nit svoj generator. Ščepec izvaja stohastični algoritem. Ko ta potrebuje naključna števila, kliče ustrezno funkcijo iz knjižnice RandomCL. Primer ščepca, ki generira naključna števila in jih shrani v pomnilnik računske naprave, je na izpisu kode 2.3.

V knjižnici RandomCL so implementirani generatorji iz naslednjih družin:

- linearni kongruenčni generatorji (angl. *Linear Congruential Generators, LCG*) [5]: 64-bitni `lcg6432` in 128-bitni `lcg12864`
- permutirani kongruenčni generatorji (angl. *Permutated Congruential Generators, PCG*) [6]: `pcg6432`
- večkratni rekurzivni generatorji (angl. *Multiple Recurential Generators, MRG*) [7, 8]: `mrg63k3a` in `mrg31k3p`
- zamaknjeni Fibonaccijevi generatorji (angl. *Lagged Fibonacci Generators*) [9]: `lfib`
- xorshift [10]: `xorshift1024`
- xorshift* [11]: `xorshift6432star`
- Mersenne Twister (MT) [12]: `mt19937`
- majhni Mersenne Twister [3]: `tinymt32` in `tinymt64`
- dobro enakomerno porazdeljen, linearen z dolgo periodo (angl. *Well-Equidistributed Long-period Linear, WELL*) [13]: `well512`
- metoda sredine kvadrata z Weylovim zaporedjem (angl. *Middle Square Weyl Sequence*) [14]: `msws`
- Philox (angl. *Product HIgh LOw Xorshift*) [15]: `philox2x32_10`

- Tyche [16]: `tyche` in `tyche_i`
- ISAAC (angl. *Indirection, Shift, Accumulate, Add, and Count*) [17]: `isaac`
- KISS (angl. *Keep It Simple, Stupid*) [18, 19]: `kiss99` in `kiss09`

Obstaja več načinov, kako z enim algoritmom vzporedno generirati več zaporedij naključnih števil. V knjižnici RandomCL uporabljamo naključno inicializacijo. Vse niti uporabljajo isti algoritem za generiranje naključnih števil. Vsaka nit ga na začetku izvajanja programa inicializira z naključno začetno vrednostjo. Ker imajo vsi implementirani generatorji periodo vsaj 2^{64}, je verjetnost, da bi več niti generiralo prekrivajoče se nize naključnih števil, majhna. Ta pristop smo izbrali, ker ga je mogoče učinkovito implementirati za poljuben generator.

Testiranje

Pri uporabi generatorjev v vzporednem algoritmu se lahko naključna števila porabljajo v drugačnem zaporedju, kot bi se v zaporednem. To je enako, kot da bi generirana števila premešali z določeno permutacijo, kar bi lahko vplivalo na kvaliteto generiranega zaporedja števil. Zato je potrebno testiranje kvalitete vzporednih implementacij generatorjev. Dober generator bi moral prestati teste tako v zaporedni, kot vzporedni implementaciji.

Za empirično testiranje kvalitete generatorjev smo uporabili knjižnico TestU01 [20]. Ta uporablja statistične teste, s katerimi išče vzorce v zaporedju generiranih števil. Če vzorcev ni mogoče zaznati, generator test prestane. Knjižnica TestU01 vsebuje 3 skupine testov: *SmallCrush*, *Crush* in *BigCrush*. Prva je najhitrejša, zadnja pa sposobna odkrivati tudi manj izrazite vzorce. Večina implementiranih generatorjev v zaporedni različici teste prestane (izjeme so generatorji `lcg6432`, `mt19937`, `tinymt32`, `tinymt64` in `well`).

Hitrost generatorjev smo testirali na 5 računskih napravah – dveh centralnih procesorjih in treh grafičnih karticah.

Rezultati

V tabeli 3.1 so rezultati testiranja kvalitete generatorjev. Generatorji `isaac`, `mt19937`, `kiss09`, `msws` in `lfib` padejo na vsaj enem izmed testov. Prva dva zato nista primerna za splošno uporabo. Generatorja `kiss09` in `msws` bi lahko prilagodili tako, da vračata le spodnjih 32 bitov in generator `lfib` zgornjih 32. Tako prirejeni generatorji prestanejo teste, a generirajo naključna števila s polovično učinkovitostjo.

Da lahko primerjamo generatorje prek različnih naprav, definiramo *relativno hitrost* generatorja (enačba (3.1)), *povprečno relativno hitrost* (enačba (3.2)) in *najslabšo relativno hitrost* (enačba (3.3)). V formulah je D število testiranih naprav, s_{gd} hitrost generatorja g na napravi d, sr_{gd} *relativna hitrost*, $\overline{s_d}$ *povprečna relativna hitrost* in min s_d *najslabša relativna hitrost* generatorja g.

V tabelah 3.2 in 3.3 so rezultati testiranja hitrosti generatorjev. Vsebujeta povprečno hitrost in standardni odklon hitrosti generiranja, ki sta izračunana iz 100 ponovitev meritve, ter *povprečno relativno hitrost* in *najslabšo relativno hitrost* vsakega generatorja. Generatorji so urejeni padajoče po *povprečni relativni hitrosti*. Za vsak stolpec je najboljši rezultat generatorjev, ki prestanejo teste, napisan krepko.

Razprava in zaključek

Generatorji `lcg6432`, `mt19937`, `tinymt32`, `tinymt64`, `well`, `isaac`, `kiss09`, `lfib` in `msws` niso splošno uporabni, ker padejo na vsaj enem testu v zaporedni ali vzporedni implementaciji.

Generator `tyche_i` je v povprečju najhitrejši. Le na Intelovem procesorju je generator `pcg6432` občutno hitrejši. Najbolj robusten je `msws`. To pomeni, da dosega največjo *najslabšo relativno hitrost* – na vseh testiranih napravah deluje relativno hitro. To velja tudi, če uporabljamo le spodnjih 32 bitov izhoda generatorja, ki teste kvalitete prestanejo.

Generator naključnih števil je običajno del večjega algoritma. Skupna hitrost algoritma je lahko odvisna od generatorja na netrivialen način. Zato je pri optimizaciji algoritma smiselno testirati več izmed hitrih generatorjev. Na ta način lahko ugotovimo, kateri deluje najhitreje znotraj konkretnega algoritma. Možno je

uporabiti tudi generator, ki ne prestane vseh testov, a je v tem primeru potrebno preveriti, da ne vpliva na pravilnost algoritma.

V nadaljnjem delu je možno implementirati in testirati še druge generatorje. Samo testiranje kvalitete generatorjev bi bilo veliko hitrejše, če bi obstajale vzporedne implementacije statističnih testov iz knjižnice TestU01.

Chapter 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 [21] methods, genetic algorithms [22] 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).

A RNG of poor quality can affect the performance of such an algorithm or even cause it to produce incorrect results. Most programming languages already implement an efficient and sufficiently good RNG in their standard libraries. However, these implementations are sequential. Some libraries with parallel implementations exist, but only a few can be run on a graphics processing unit (GPU) [1, 2, 3, 4] . Furthermore, each library implements at most a few RNGs.

Quality and performance of sequential RNGs has been extensively evaluated [20]. The only evaluation of GPU implementations we have found compares a small number of RNGs on a single GPU [2]. Most of those RNGs have known flaws [20]. We are not aware of any comparison of RNGs across different GPUs and CPUs.

Therefore, a user that 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. To make things worse, it is also not clear which parallel RNG is the most efficient or which RNGs are too flawed for practical use.

The main goal of this thesis was to prepare a library with a set of different parallel RNG implementations, which users can easily include in their algorithms.

Additionally, we performed several experiments that provide insight into the effectiveness of the RNGs and the practical usefulness of the sequences of numbers that they generate.

1.1 GPUs and OpenCL Framework

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.

The OpenCL framework allows the use of the same application on a multi-core CPU as well as on a many-core GPU. It allows to implement functions that 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. A work group is a group of threads that is executed on a single compute unit. For a CPU a compute unit is the same as a core. The threads of a work group can execute mathematical operation at the same time in a vector unit.

GPUs are organized in a similar way. However for GPUs the term core is not used for compute units.

GPUs commonly have three levels of memory. Global memory is the slowest but largest level of GPU memory. It is equivalent to what RAM is to the CPU. Local memory is faster but much smaller. It has the same function as cache on CPU. Private memory is the fastest but also the smallest. It consists of registers. Compared to CPU cache hierarchy all levels of GPU memory are directly addressable by code.

1.2 Random Number Generators

Random is the opposite of deterministic. An event is random if it can not be predicted with certainty. In computers, random numbers are typically generated as independent samples from a uniform distribution. Computers are inherently

deterministic. That means they can not algorithmically generate true random numbers. For generating true random numbers, an external source of randomness is required. As we do not deal with true random number generators, we use the term RNG or generator to describe pseudo-random number generators.

In stochastic algorithms, pseudo-random numbers are typically used. A pseudo-random number generator is an algorithm that outputs a sequence of numbers that appears random. However, for all pseudo-random number generators this sequence is periodic. While being deterministic, good pseudo-RNGs generate a sequence of seemingly unpredictable numbers that has a long period and can be used to simulate a random process.

This definition is quite vague. In fact, good definitions of randomness and RNGs are either vague, such as "*Conceptually, these RNGs are designed to produce sequences of real numbers that behave approximately as the realizations of independent random variables uniformly distributed over the interval (0, 1), or i.i.d.* $U(0,1)$" [23, p.3] or even claimed not to exist: "*... there is no satisfactory definition of randomness feasible for PRNG.*" [24, p.2]. Some authors even make the definition of randomness depend on the tests applied to the RNG: "*... using a (pseudo-)random number generator, one can deterministically generate a sequence of numbers that looks just like a truly random sequence in terms of specified statistical tests.*" [25, p.1].

In this work, we adopt the last definition - we consider a RNG to be random (that is, good) enough, if it passes all the statistical tests that are aimed at identifying discrepancies between the generated sequences and what we would expected from a theoretical uniform random variable (or, if the RNG is used to generate another distribution, discrepancies from that distribution).

In general, a RNG consists of a state x_n, a state transition function f and an output function g. To generate the n-th random number y_n, the state of generator is first advanced according to $x_n = f(x_{n-1})$, before outputting a number $y_n = g(x_n)$. In practice, the output function is usually simple, sometimes even the identity. In generators with a large state, it often returns just a part of the generator state.

If the state size of a RNG is b bits, it is a b-bit RNG. A b-bit RNG can be in at most 2^b different states, which is also an upper bound on its period.

Before generating any numbers, state x_0 is initialized using a random seed. If

generating a repeatable sequence of numbers is desired, a predetermined value can be used as the seed. Otherwise, a true random number or the current time is commonly used.

Compared to true random numbers, generating pseudo-random numbers is often faster. Any algorithm involving RNGs can be executed with repeatable results as many times as is required by using the same random seed. This is useful for debugging and reproducing the results of scientific experiments.

RNGs have a large number of uses. Trivial use cases include playing shuffled music and randomizing events in computer games. When recording audio with a computer a random dithering [26] is used to reduce audibility of the quantization noise. Randomness is needed if one wants to simulate any system with some random inputs, such as weather or Brownian motion [27]. In fact it is needed to generate random samples from any statistical distribution [28]. In cryptography [29], good random numbers are very important for guarantees of security, for example when generating cryptographic keys. A large field that uses RNGs are randomized algorithms [30], including Monte Carlo Simulations [21] and genetic algorithms [22].

In this thesis the focus is on RNGs for use in algorithms. This is the field where the speedup by parallelization is the most important. RNGs, used in algorithms must be fast and produce numbers of reasonable quality. Implemented RNGs could also be used for any other mentioned field, except cryptography.

The requirements for cryptographically secure RNGs are even more strict [31]. Knowing a sequence of numbers it should be practically impossible to predict any previous or next number numbers. Knowing the generator state it should also be practically impossible to predict any previously generated numbers. This is the reason cryptographic RNGs tend to be significantly slower.

Chapter 2

RandomCL Library

We implemented a library named RandomCL that contains 19 random number generators. The library is header-only and can be used on any operating system that supports the OpenCL framework. The generators from 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 our generators 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 numbers are generated between 0 and 1.

By default the RNGs generate either 32- or 64-bit unsigned integers. To generate a 32-bit number with a 64-bit output generator, we drop half of the RNG output. To generate a 64-bit number with a 32-bit output generator, we use two consecutive numbers. Conversion to floating point numbers is done by multiplication of the integer output by a precomputed constant $c = \frac{1}{\text{max output number}}$. The value of this constant depends on the RNG.

A typical use of the library consists of multiple steps:

- First, random seeds are generated for each thread using a sequential generator - from a standard library.
- Seeds are then copied to the compute device's global memory.

- Next, the stochastic application's OpenCL kernel (implemented by the user of the RandomCL library) is run on the device. It first calls the seeding function of RNG in use, to initialize a generator for each thread using the previously generated seeds.
- Finally, the kernel calls the RNG function that generates the next random number, whenever 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 the steps are:

- First, random seeds are generated for each thread using a sequential generator - from a standard library.
- Seeds are then copied to the compute device's global memory.
- Next, the OpenCL kernel that is part of the library is run on the device. It first initializes a generator for each thread using the previously generated seeds.
- The kernel calls the RNG function to generate random numbers and saves them to the device's global memory.
- Finally, the stochastic application's kernel is run. As it needs random numbers it reads them from the device's 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 RNGs

When choosing which RNGs to implement, we first opted for some well-known ones, such as the linear congruential generator and the Mersenne Twister. Other RNGs were chosen because their sequential implementations are known to pass the tests from TestU01 library [20] and are relatively fast.

We implemented several variations of the RNGs with small differences that could affect performance, but not the quality of the generator. However, after preliminary testing, we determined that for most RNGs differences in speed were very small. Where it actually makes a difference, the RandomCL library contains the fastest variation. This is also the variation we report the results for.

We implemented the following RNGs:

- **Linear Congruential Generator (LCG)** [5] generates random numbers according to equation $x_n = (x_{n-1}a + b) \bmod m$, where a, b and m are parameters. If m is a power of 2, the implementation is very simple and fast. LCGs are known as poor generators, especially for m that is a power of 2, but they can still pass the BigCrush test battery (described in Chapter 3.1) if only a part of state is returned [6]. We have implemented 128-bit LCG, that returns the upper 64 bits (`lcg12864`) and 64-bit LCG, that returns the upper 32 bits (`lcg6432`). The `lcg6432` generator has the following parameters: $m = 2^{64}$, $a = 6364136223846793005$ and $b = 1572607049536067 0683$. The `lcg12864` generator has the following parameters: $m = 2^{128}$, $a = 47026247687942121848144207491837523525$ and $b = 117397592171526113268558934119004209487$. The `lcg6432` generator does not pass the BigCrush test battery [6].

- **Permutated Congruential Generator (PCG)** [6] combines a LCG and a non-trivial output function. Multiple versions with different output functions exist. We implemented 64-bit generator that returns 32-bit numbers `pcg6432`. It uses the same LCG parameters as generator `lcg6432`. 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.

- **Multiple Recursive Generator (MRG)** [7] of order k generates random numbers according to function $y_n = (a_1 y_{n-1} + ... + a_k y_{n-k}) \bmod m$, where a_i and m are parameters. The state of this RNG consists of the last k generated numbers. We implemented two MRGs, `mrg31k3p` [8] and `mrg63k3a` [7].

- **Lagged Fibonacci Generator** [9], defined by lags r, s, and a binary operation $*$ generates numbers according to equation $y_n = y_{n-r} * y_{n-s}$. Its state consists of the last $\max(r, s)$ generated numbers. If $*$ is addition, subtraction or exclusive-or, resulting generators are known to have poor quality [20]. We implemented lagged Fibonacci generator `lfib` using multiplication and $r = 17$, $s = 5$.

- **Xorshift** [10] 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 [32]. The 64-bit xorshift does not pass the BigCrush test battery on its own [20]. We implemented 1024-bit xorshift generator `xorshift1024` that uses the following shifts: 329 to the left, 347 to the right and 344 to the left [2]. Its state is advanced jointly by 32 threads.

- **Xorshift*** [11] is an xorshift generator with a non-trivial output function - a multiplication with an constant. We implemented 64-bit generator `xorshift6432star` that returns 32 bits of its state. That makes it pass BigCrush test battery [6]. It uses the multiplication constant of 2685821657736338717 and following shifts: 12 to the right, 25 to the left and 27 to the right.

- **Mersenne Twister** [12] 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 the BigCrush test battery. We implemented Mersenne Twister `mt19937`.

- **Tiny Mersenne Twister** [3] 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 the OpenCL programming language. We only modified the interface to make it similar to other generators in the RandomCL library. There is 32-bit version `tinymt32` and 64-bit version `tinymt64`.

- **WELL (Well-Equidistributed Long-period Linear)** [13] was created as an improvement to Mersenne Twister. While it has some nice theoretical properties it still fails some tests in the BigCrush test battery. We implemented the smallest, 512-bit version of the generator `well512`.

- **Middle Square Weyl Sequence** [14] generates the next number by squaring the previous one before swapping the lower and upper bits of the residue modulo 2^{64}. Lastly, a number generated by a Weyl sequence [10] is added. Weyl sequence produces the next number by adding a constant to the previous one and taking the residue modulo 2^{64}. We implemented 64-bit middle square Weyl sequence `msws` that uses an increment of 13091206342165455529 for the Weyl sequence part.

- **Philox (Product HIgh LOw Xorshift)** [15] is a counter-based RNG. That means 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 10-round Phylox RNG that works on two 32-bit numbers `philox2x32_10`.

- **Tyche** [16] 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`.

- **ISAAC (Indirection, Shift, Accumulate, Add, and Count)** [17] is a RNG originally 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)** [18, 19] 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 multiply-with-carry (MWC) generators. `Kiss99` uses 32-bit component RNGs, while `kiss09` uses 64-bit components. Both pass the BigCrush test battery, even though none of their components do.

Sequential pseudocode for all implemented generators is in the appendix A. They all contain the declaration of generator state and functions for seeding the RNG and generating a random number. Seeding a generator can be done in an almost arbitrary way. However, one must be careful to assign a valid value to its state. For simplicity the pseudocode shows generator implementations that save their states in a global variable so there can only be one instance of each generator at once.

2.2 Parallelization

We are using random initialization to generate random numbers in a parallel. Each thread has its own instance of a generator, initialized to a random state. Seeds for each thread must not be generated with the same generator. If they were, threads would output the same sequence of numbers, shifted by one number. While random initialization is efficient and doable for any generator, it is possible that the generated streams overlap. The probability of overlap can be reduced by using a generator with a longer period. If a generator has period p and we use T threads to generate ℓ random numbers per thread, the probability that any generated streams overlap can be calculated with equation (2.1) [23].

$$1 - \left(1 - \frac{T\ell}{p}\right)^{T-1} \tag{2.1}$$

Some example probabilities are calculated in table 2.1. All implemented generators have a period in order of magnitude of at least 2^{64}, so the probability of an overlap is small for all but the largest use cases. For generators with the period of 2^{128} or more the probability of overlap is negligible. Generators with the period of 2^{32} or less almost guarantee overlap even in relatively small use cases. That is one of the reasons why we did not implement any generators with such short period.

There are several possible alternatives to our approach [23]. A trivial alternative would be to generate a sequence of random numbers sequentially - possibly in

Table 2.1: **Probability of random stream overlap.** Listed are a few example thread numbers and typical devices, on which such number of threads would be used. All example cases assume 10^7 numbers are generated per thread.

example device	number of threads	probability of overlap for typical RNG periods 2^{32}	2^{64}	2^{128}
CPU	200	1	10^{-8}	10^{-27}
mid-range GPU	4000	1	10^{-6}	10^{-25}
high-end GPU	250000	1	0.03	10^{-21}
30 GPUs	$7.5 \cdot 10^6$	1	1	10^{-18}
1000 GPUs	$2.5 \cdot 10^8$	1	1	10^{-15}
10^6 GPUs	$2.5 \cdot 10^{11}$	1	1	10^{-9}

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 different parameter sets would suffice. However, many parameter sets that produce streams of good quality exist only for a few RNGs. Even if the quality of each stream is good, that does not imply that the numbers from different streams are independent. The independence must be tested [23].

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 possible only for a few RNGs, while advancing one step at a time would be too time consuming to be practically feasible. It has been shown at least for LCGs that

Listing 2.1: Using a LCG to generate 1000 random numbers sequentially

```
1  uint64 a = 6364136223846793005
2  uint64 b = 1572607049536067683
3  uint64 seed = 12345
4  uint32 result[1000]
5
6  uint64 state = seed
7
8  for(uint32 i = 0; i < 1000; i++){
9      state = a * state + b
10     result[i] = (uint32)state
11 }
```

random initialization gives better quality of the generated numbers than equally spaced substreams [2].

2.2.1 An Example of Parallelization

We can take for example linear congruential generator `lcg6432` (described in section 2.1). Listing 2.1 shows the pseudo-code of how we generate 1000 random numbers sequentially. The first two lines define parameters of the generator. The third line declares the variable holding the initial seed of generator. The fourth line declares an array holding the results. Next the generator is initialized using the provided seed. Finally the numbers are generated in a loop and saved in the array.

Alternatively, we can generate random numbers in parallel using the same generator. Listing 2.2 shows pseudo-code how to do that. The first four lines are the same as in the sequential example. The fifth line declares a variable that holds the number of threads that will be run in parallel. The sixth line declares an array of generator states - one state per thread. First a sequential RNG is initialized using provided `seed`. It can be any RNG except the one used for parallel generation. It is used to generate one random seed for each thread. Finally, `T` threads are started to generate numbers. Each thread queries for its index `id` and then generates every

`T`-th number in the `result` array starting at index `id`.

2.3 An Example of Using a RandomCL RNG

Listing 2.3 shows how to fill an array in OpenCL kernel with random numbers using a RandomCL RNG. It uses the `tyche_i` RNG to generate 32-bit unsigned integers. Other RandomCL RNGs could be used in similar way.

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. First argument `num` sets the number of random values to generate. 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. Last 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 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 2.2: Using LCG to generate 1000 random numbers in parallel using random initialization

```
1  uint64 a = 6364136223846793005
2  uint64 b = 1572607049536067683
3  uint64 seed = 12345
4  uint32 result[1000]
5  int32 T = 128
6  uint64 state[T]
7
8  uint64 seq_state = seed
9  for(int i = 0; i < T; i++){
10     seq_state = sequential_RNG(seq_state)
11     state[i] = seq_state
12 }
13
14 execute in parallel using T threads{
15     int32 id = get_index_of_current_thread()
16     for(int32 i = id; i < 1000; i += T){
17         state[id] = a * state[id] + b
18         result[i] = (uint32)state[id]
19     }
20 }
```

Listing 2.3: 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 }
```

Chapter 3

Empirical Evaluation

Since pseudo-random number generators produce numbers in a deterministic way, it is clear that the generated numbers are not equivalent to theoretical observations of random variables. Deciding which list of properties a good RNG should have is hard, as even the definition of what random in pseudo-RNG means is not agreed upon.

Some deficiencies and desirable properties of RNGs can be proven theoretically. Further evaluation of RNGs can be done by empirical testing. While it can not prove a generator is good, statistical tests can be used to check a RNG for common deficiencies. A sequence of numbers produced by a good RNG should be difficult to distinguish from true random numbers.

A good RNG should also be efficient (that is, fast). This is even more important for parallel implementations, as the purpose of parallelization is reduction of the running time.

In cryptography, it is common to test correctness of implementations for algorithms, such as cryptographic RNGs using test vectors. That means the original author of the RNG publishes a seed and a sequence of numbers that is generated for that seed (a test vector). This way any other implementation can be checked that it produces the same numbers while initialized with the same seed. The testing using test vectors can only prove two implementations of the RNG are different. It can not show which one, if any, is the correct one.

However, publishing test vectors for the non-cryptographic RNGs is not common. We have not found test vectors for any of the implemented RNGs. This might be

because the cryptographic RNGs tend to be more complex, which makes mistakes in the implementation more likely.

3.1 Testing Quality

The TestU01 library [20, 33] is the most commonly used suite for empirically testing the quality of RNGs. It extends the DIEHARD suite with more tests. TestU01 defines three batteries that specify the tests and their parameters. From fastest to most discriminative they are SmallCrush, Crush and BigCrush. SmallCrush includes 10 tests, requiring approximately 51 million 32-bit random numbers to be generated. Crush includes 96 tests, requiring approximately 34 billion 32-bit numbers. BigCrush includes 106 tests, requiring approximately 274 billion 32-bit numbers.

Tests work in the following way. First, the random numbers are generated using the tested RNG. Optionally, they are transformed in some way to result in samples from another distribution. Lastly, these samples are compared to what would be expected of a theoretically determined distribution using a statistical test. This entire process is based on the fact that a true RNG can be used, with appropriate transformations, to generate samples from other distributions. If it fails to do so, the RNG is flawed.

The test fails if the probability (p-value) that the expected distribution generates the numbers obtained using tested RNG is too small or too large. Too small a p-value means that a RNG exhibits noticeable patterns, which would not occur in a true random sequence. Too large a p-value means the RNG produces results that are too regular. A relatively small threshold on p-value (significance) must be chosen to reduce probability of test failures by chance as many tests are used on each generator. We used the default TestU01 significance value of 0.0001. That means a test fails if it produces a p-value smaller than 0.0001 or larger than 0.9999.

A mathematical description of all tests and their parameters, as they are used in test batteries from TestU01 library, is out of scope of this thesis, as there are simply too many of them. The complete list of tests, their parameters, brief descriptions and citations of further mathematical background can be found in TestU01 User's Guide [33]. To give the reader an idea of how exactly the tests work, we describe

two of them: the *WeightDistrib* test and the *HammingWeight* test.

The *WeightDistrib* test with parameters n, k, α and β uses the tested RNG to generate n groups of k floating-point numbers between 0 and 1. In each group the numbers that are between α and β are counted. Counts C should be distributed according to binomial distribution $C \sim \text{Binom}(k, \beta - \alpha)$. The actual distribution is compared to the expected one with a chi-square test.

The *HammingWeight* test groups numbers into L groups containing n bits. L and n are parameters of the test. In each block the number of bits equal to 1 is counted. Counts C are expected to be distributed according to binomial distribution $C \sim \text{Binom}(L, 0.5)$. A chi-square test is used to compare expected and actual distribution.

Among others, the following properties of RNGs are tested:

- Multidimensional equidistribution [34].
- Entropy [35].
- Pairwise distances between generated points in multidimensional space.
- Number of permutations required to sort generated numbers.
- Fraction of generated numbers that are within specified interval.
- Count of distinct numbers in groups of generated small integers.
- Count of small integers to generate before all distinct values are generated.
- Lengths of increasing and decreasing runs in a generated sequence of numbers.
- Distribution of maximal numbers from groups of generated numbers.
- Linear independence [36] of generated bits.
- Autocorrelation [37] of generated sequences.
- Distribution of mean values and products of generated sequences.
- Spectra obtained by Fourier transform of generated sequences.
- Lengths of runs of ones and zeros in generated sequence of bits.

- Distribution of Hamming weights [38] in generated numbers.
- Distances between consecutive sorted numbers.

Sequential implementations of most generators we implemented are known to pass the BigCrush test battery (the exceptions are `lcg6432`, `mt19937`, `tinymt32`, `tinymt64`, and `well`).

Depending on how they are used, random numbers generated in parallel may or may not be consumed in the same order as they have been 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 the problem, one number from each thread is consumed before the next number from first thread is consumed. Which of those options is used in a parallel algorithm depends on the way the algorithm is parallelized.

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 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 of generators, which effectively return permuted sequences. It is impossible to test permutations for all possible numbers of threads. We wanted to choose a number that would be representative of a typical usage of a GPU. So it must be a multiple of 64 as GPUs run threads either in multiples of 32 or 64. Nowadays a mid-range GPU has around 1000 cores, so we selected 1024 as a representative number and executed the tests on as many threads.

The TestU01 library can only test 32-bit numbers. So we have tested 64-bit output 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 Speed

We tested the speed of the implemented RNGs on several different devices. We used one CPU and one GPU from each major vendor. This way we obtained the following list: AMD Radeon R7 260X (2013 mid-end, gaming GPU), AMD Ryzen Threadripper 1950X (2017 high-end CPU), Intel Core i5-4690 (2014 mid-end CPU), Intel HD Graphics 4000 (2012 low-end, integrated GPU) and NVIDIA GeForce GTX 1070 (2016 high-end, gaming GPU). Tests on all the devices were carried out on computers with Windows 10 operating system.

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 the 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.

To time an OpenCL kernel function, we simply measured wall-clock time as a difference between start time and end time. We did not want to use device-specific measurement methods and the OpenCL framework only supports measurement of wall-clock time.

The amount of numbers to generate with each generator on each device was automatically selected as a power of 2 (for efficiency of the generation) that resulted in running time between 0.1 and 0.2 seconds. This way tests ran for long enough that the latency of starting the kernel is negligible. If we used a fixed amount of

generated numbers, the running times on slower devices would be considerably larger. Performance testing of every generator was run 100 times on each device. This number of repetitions was selected because it is large enough to provide stable results and small enough for a practical test duration.

The amount of numbers to generate was determined dynamically, just before the start of testing on each device. First, one number is generated and execution is timed. The amount of numbers is than repeatedly doubled until an appropriate running time is reached. This also serves to "warm-up" the device, which makes sure the device is not in a power-saving or idle state.

3.3 Results

From Table 3.1 we can see that the parallel implementations of generators `isaac` and `mt19937` fail at least one of the tests. That makes them unsuitable for general purpose parallel RNGs. Generators `kiss09`, `msws` and `lfib` also fail some, but could still be used. We can see that the lower 32 bits of generators `kiss09` and `msws` and upper 32 bits of generator `lfib` pass all tests, so we could modify those RNGs to only return half of their current output. However, that would effectively halve the speed at which they generate numbers.

To compare speeds of a generator across devices we define *relative speed* of a particular generator on a particular device as quotient of the speed of the generator on that device and the speed of the fastest generator implementation on the same device. Let s_{gd} be the speed of generator g on device d. Its *relative speed* sr_{gd} is

$$sr_{gd} = \frac{s_{gd}}{\max\limits_{i}(s_{gi})}. \tag{3.1}$$

Using the *relative speed* we can report the average and the worst speed across devices. Let s_{gd} be the speed of generator g on device d and D the number of devices. We can define *average relative speed* $\overline{s_d}$ and *worst relative speed* $\min\ s_d$:

$$\overline{s_d} = \frac{1}{D}\sum_{d=1}^{D}\frac{s_{gd}}{\max\limits_{i}(s_{gi})}. \tag{3.2}$$

$$\min\ s_d = \min_{d}\frac{s_{gd}}{\max\limits_{i}(s_{gi})}. \tag{3.3}$$

Table 3.1: **Quality Tests Results.** For each generator, we report the number of failures on each test battery. 64-bit generators have three results for every battery of tests - for lower 32 bits, upper 32 bits and both as two 32-bit numbers.

output	generator	SmallCrush			Crush			BigCrush		
32-bit	isaac			0			1			0
	kiss99			0			0			0
	lcg6432			0			0			0
	mrg31k3p			0			0			0
	mt19937			1			1			0
	pcg6432			0			0			0
	tinymt32			0			0			0
	well512			0			0			0
	xorshift1024			0			0			0
	xorshift6432star			0			0			0
64-bit	kiss09	0	0	0	0	1	0	0	0	0
	lcg12864	0	0	0	0	0	0	0	0	0
	lfib	8	0	6	70	0	53	52	0	40
	mrg63k3a	0	0	0	0	0	0	0	0	0
	msws	0	0	0	0	10	4	0	26	10
	philox2x32_10	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

The *average relative speed* is the average performance of a generator across tested devices. It can also be understood as expected speed on a random or unknown device.

The *worst relative speed* represents the robustness of a generator. A generator can achieve a high *average relative speed* by being particularly fast on one or a few devices and slow on the others. To have a high *worst relative speed* a generator must perform reasonably well on all tested devices.

Tables 3.2 and 3.3 contain measurements of speed as the average value and standard deviation for each pair of generator and device. They also contain the *average* and the *worst relative speed* across devices for each generator.

Table 3.2: **Performance Tests Results 1/2.** Generators are ordered by their *average relative speed*. Absolute measurements are presented as the average value and the standard deviation in gigabytes per second. For each column the speed of the fastest generator that passes the BigCrush test battery is in bold.

generator	average relative speed	worst relative speed	AMD Radeon R7 260X	AMD Ryzen Threadripper 1950X	Intel Core i5-4690	Intel HD Graphics 4000	NVIDIA GeForce GTX 1070
msws	0.857	**0.583**	327.37	140.44	128.23	29.51	1621.94
			±0.06	±4.22	±8.26	±0.44	±13.92
tyche_i	**0.722**	0.158	545.87	**112.20**	20.39	**22.81**	**1967.44**
			±0.33	±3.81	±1.50	±0.44	±16.95
lcg6432	0.636	0.312	175.46	111.24	99.27	33.16	617.70
			±0.03	±4.73	±5.65	±0.92	±5.26
tyche	0.567	0.109	**561.48**	69.70	14.09	20.69	1196.88
			±0.31	±2.58	±0.94	±0.41	±32.91
kiss09	0.516	0.054	353.20	96.80	108.90	1.81	717.65
			±0.20	±2.78	±7.08	±0.02	±8.40
xorshift-6432star	0.440	0.240	278.89	61.37	63.81	17.77	471.88
			±0.05	±1.82	±4.08	±0.38	±3.99
tinymt64	0.438	0.323	234.80	53.99	44.56	24.01	635.35
			±0.06	±1.52	±3.27	±0.45	±5.25
pcg6432	0.402	0.165	92.93	62.66	**86.18**	17.51	402.79
			±0.02	±2.05	±6.75	±0.41	±3.30
tinymt32	0.291	0.179	152.42	26.72	23.05	19.15	474.96
			±0.03	±0.76	±1.60	±0.51	±4.01
lcg12864	0.290	0.132	78.65	78.32	51.52	7.30	259.94
			±0.01	±2.19	±3.51	±0.12	±2.30

Table 3.3: **Performance Tests Results 2/2.** Generators are ordered by their *average relative speed*. Absolute measurements are presented as the average value and the standard deviation in gigabytes per second. Our implementation of the `isaac` generator does not work on GPUs as it requires unaligned memory access.

generator	average relative speed	worst relative speed	AMD Radeon R7 260X	AMD Ryzen Thread-ripper 1950X	Intel Core i5-4690	Intel HD Graphics 4000	NVIDIA GeForce GTX 1070
`kiss99`	0.289	0.066	238.29	33.35	46.78	2.20	701.94
			±0.04	±0.94	±3.38	±0.03	±5.95
`lfib`	0.164	0.034	72.29	77.64	4.42	1.69	104.27
			±0.05	±2.57	±0.32	±0.03	±1.21
`isaac`	0.139	0.033	/	34.43	4.20	/	/
				±1.15	±0.31		
`philox2-x32_10`	0.116	0.067	82.83	14.21	8.63	5.90	173.76
			±0.13	±0.40	±0.42	±0.23	±1.42
`xorshift-1024`	0.083	0.002	112.44	0.21	2.69	3.67	164.05
			±0.03	±0.01	±0.18	±0.07	±1.38
`mrg31k3p`	0.079	0.033	18.68	13.98	17.17	1.66	154.37
			±0.00	±0.37	±1.09	±0.03	±1.31
`well512`	0.066	0.028	40.88	16.23	3.65	2.17	97.93
			±0.01	±0.63	±0.23	±0.05	±0.74
`mrg63k3a`	0.060	0.003	14.81	13.48	19.40	0.11	49.23
			±0.10	±0.30	±1.25	±0.00	±0.51
`mt19937`	0.033	0.005	16.68	13.22	2.02	0.18	44.01
			±0.07	±0.43	±0.13	±0.01	±1.64

Chapter 4

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 Section 3.1. Generators `lcg6432`, `mt19937`, `tinymt32`, `tinymt64`, `well`, `isaac`, `kiss09`, `lfib` and `msws` fail the testing either when run sequentially or in parallel. Among the remaining RNGs, we are most interested in the ones that can generate numbers quickly on a variety of devices.

We can see that in general different generators produce numbers at very diverse speeds. The generator `tyche_i` is on average the fastest among the ones that pass the BigCrush test battery (the generator `msws` does not, unless we only use half of its output, which would make it slower on average than the generator `tyche_i`). It is also the fastest on the Intel and the NVIDIA GPU, the AMD CPU and a close second on the AMD GPU (we again disregard generators that fail the BigCrush test battery). However, its *worst relative speed* is quite low due to poor performance on the Intel CPU. The best generator on the AMD GPU is `tyche`. On the Intel CPU generator `pcg6432` is the fastest among those that pass the tests. The best *worst relative speed* is achieved by generator `msws`. While it does not pass the BigCrush test battery, we can use only the lower half of its output, which does pass. This way we obtain half of its *worst relative speed*, which is still more than any other generator that passes the BigCrush test battery.

As a general purpose RNG that can run very fast on almost any device, generator `tyche_i` is a good choice. If we target only a specific device, we can instead select the generator with the best performance on the most similar device. If it is really

important that the generator does not run slowly on any device, the lower 32 bits of generator `msws` should be used.

Usually, however, the 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 in order to find which one works best for a particular case.

One could also try the generators that fail some of the tests, as the algorithm might not be sensitive to deficiencies of a particular generator. In that case, however, the algorithm should be tested for correctness while using each of the candidate generators. Such testing may also be beneficial in general, as the algorithm could be sensitive to a deficiency that is not tested for in the TestU01 library.

As part of future work, one could extend the list of tested generators. It would be interesting to add implementations that can be split into substreams of equal length or that use different parameter sets for each thread. It might also be possible to tweak the Middle Square Weyl Sequence RNG to make its whole state pass the BigCrush test battery without affecting its speed, which would make by far the most robust generator. Finally, for the research of parallel RNGs, it would be convenient to have a parallel implementation of the computationally intensive statistical tests of generator quality.

Bibliography

[1] P. L'Ecuyer, D. Munger, and N. Kemerchou, "clRNG: A random number API with multiple streams for OpenCL," *report,*, 2015. [Online] Available: `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf` [Accessed: 20 September 2018].

[2] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on GPU," *The European Physical Journal-Special Topics*, vol. 210, no. 1, pp. 53–71, 2012.

[3] M. Matsumoto and T. Nishimura, "Tiny Mersenne twister," 2011. [Online] Available: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html` [Accessed: 20 September 2018].

[4] NVIDIA technical staff, "cuRAND," 2010. [Online] Available: `https://developer.nvidia.com/curand` [Accessed: 20 September 2018].

[5] P. L'ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation of the American Mathematical Society*, vol. 68, no. 225, pp. 249–260, 1999.

[6] M. E. O'Neill, "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation," *ACM Transactions on Mathematical Software*, 2014.

[7] P. L'ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[8] P. L'Ecuyer and R. Touzin, "Fast combined multiple recursive generators with multipliers of the form a=±2 q±2 r," in *Proceedings of the 32nd conference on Winter simulation* (J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.), pp. 683–689, Society for Computer Simulation International, 2000.

[9] G. Marsaglia and L.-H. Tsay, "Matrices and the structure of random number sequences," *Linear algebra and its applications*, vol. 67, pp. 147–156, 1985.

[10] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[11] S. Vigna, "An experimental exploration of Marsaglia's xorshift generators, scrambled," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, p. 30, 2016.

[12] M. Matsumoto and T. Nishimura, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[13] F. Panneton, P. L'ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 1, pp. 1–16, 2006.

[14] B. Widynski, "Middle square Weyl sequence RNG," *arXiv preprint arXiv:1704.00358*, 2017.

[15] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for* (S. Lathrop, J. Costa, and W. Kramer, eds.), pp. 1–12, IEEE, 2011.

[16] S. Neves and F. Araujo, "Fast and small nonlinear pseudorandom number generators for computer simulation," in *International Conference on Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, eds.), pp. 92–101, Springer, 2011.

[17] R. J. Jenkins, "ISAAC," in *International Workshop on Fast Software Encryption* (D. Gollmann, ed.), pp. 41–49, Springer, 1996.

[18] G. Marsaglia, "Random numbers for C: End, at last?," 1999. [Online] Available: `http://www.cse.yorku.ca/~oz/marsaglia-rng.html` [Accessed: 20 September 2018].

[19] G. Marsaglia, "64-bit KISS RNGs," 2009. [Online] Available: `https://www.thecodingforums.com/threads/64-bit-kiss-rngs.673657` [Accessed: 20 September 2018].

[20] P. L'Ecuyer and R. Simard, "TestU01: A C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[21] C. Z. Mooney, *Monte carlo simulation*, vol. 116. Sage Publications, 1997.

[22] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[23] P. L'Ecuyer, D. Munger, B. Oreshkin, and R. Simard, "Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs," *Mathematics and Computers in Simulation*, vol. 135, pp. 3–17, 2017.

[24] M. Matsumoto, "Randomness in computation," in *Japanese German Frontiers of Science Symposium* (H. Arimura and J. Rothe, eds.), p. 2, Japan Society for the Promotion of Science (JSPS), 2005.

[25] H. Arimura and J. Rothe, "Randomness in computation," in *Japanese German Frontiers of Science Symposium* (H. Arimura and J. Rothe, eds.), p. 1, Japan Society for the Promotion of Science (JSPS), 2005.

[26] S. P. Lipshitz, R. A. Wannamaker, and J. Vanderkooy, "Quantization and dither: A theoretical survey," *Journal of the audio engineering society*, vol. 40, no. 5, pp. 355–375, 1992.

[27] T. Hida, "Brownian motion," in *Brownian Motion*, pp. 44–113, Springer, 1980.

[28] C. Forbes, M. Evans, N. Hastings, and B. Peacock, *Statistical distributions*. John Wiley & Sons, 2011.

[29] K. Marton, A. Suciu, and I. Ignat, "Randomness in digital cryptography: A survey," *Romanian Journal of Information Science and Technology*, vol. 13, no. 3, pp. 219–240, 2010.

[30] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge university press, 1995.

[31] F. Özkaynak, "Cryptographically secure random number generator with chaotic additional input," *Nonlinear Dynamics*, vol. 78, no. 3, pp. 2015–2020, 2014.

[32] R. P. Brent, "Note on Marsaglia's xorshift random number generators," *Journal of Statistical Software*, vol. 11, no. 5, pp. 1–4, 2004.

[33] P. L'Ecuyer and R. Simard, "TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators, User's guide, compact version," 2013. [Online] Available: `http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf` [Accessed: 20 September 2018].

[34] R. P. Brent, "The myth of equidistribution for high-dimensional simulation," *arXiv preprint arXiv:1005.1320*, 2010.

[35] P. L'Écuyer, A. Compagner, and J.-f. Cordeau, *Entropy tests for random number generators*. École des hautes études commerciales, Groupe d'études et de recherche en analyse des décisions, 1996.

[36] "Linear independance." [Online] Available: `https://en.wikipedia.org/wiki/Linear_independence` [Accessed: 23 October 2018].

[37] "Autocorrelation." [Online] Available: `https://en.wikipedia.org/wiki/Autocorrelation` [Accessed: 23 October 2018].

[38] "Hamming weight." [Online] Available: `https://en.wikipedia.org/wiki/Hamming_weight` [Accessed: 23 October 2018].

Appendix A

Pseudocode of the Implemented Generators

Listing 1: `lcg6432` pseudocode

```
uint64 state

lcg6432_seed(uint64 seed){
    state = seed
}

uint32 lcg6432_next_int32(){
    state = state * 6364136223846793005 +
            1572607049536067683
    return (uint32) state
}
```

Listing 2: `lcg12864` pseudocode

```
uint64 state_low, state_high

lcg12864_seed(int64 seed){
    state_low = seed
    state_high = seed ^ 1572607049536067683
}
```

```
uint64 lcg12864_next_uint64(){
    state_high = state_high * 4865540595714422341 +
        state_low * 2549297995355413924 +
        mul_hi(state_low, 4865540595714422341)
    state_low = state_low * 4865540595714422341
    state_low += 1442695040888963407
    state_high += state_low < 1442695040888963407
    state_high += 6364136223846793005
    return state_high
}
```

Listing 3: pcg6432 pseudocode

```
uint64 state

pcg6432_seed(uint64 seed){
    state = seed
}

uint32 pcg6432_next_uint32(){
    uint64 oldstate = state
    state = oldstate * 6364136223846793005 +
        1572607049536067683
    uint32 xorshifted = ((oldstate >> 18) ^
        oldstate) >> 27
    uint32 rot = oldstate >> 59
    return (xorshifted >> rot) |
        (xorshifted << ((-rot) & 31))
}
```

Listing 4: mrg63k3a pseudocode

```
uint64 state10, state11, state12
uint64 state20, state21, state22
```

```
mrg63k3a_seed(uint64 seed){
    state10 = seed
    state11 = seed
    state12 = seed
    state20 = seed
    state21 = seed
    state22 = seed
    if(seed == 0){
        state10++
        state21++
    }
}

uint64 mrg63k3a_next_uint64(){
    uint64 h, p12, p13, p21, p23
    h = state10 / 2898513661
    p13 = 3182104042 * (state10 - h * 2898513661) -
        h * 394451401
    h = state11 / 5256471877
    p12 = 1754669720 * (state11 - h * 5256471877) -
        h * 251304723
    if (p13 < 0)
        p13 += 9223372036854769163
    if (p12 < 0)
        p12 += 9223372036854769163 - p13
    else
        p12 -= p13
    if (p12 < 0)
        p12 += 9223372036854769163
    state10 = state11
    state11 = state12
    state12 = p12
```

```
    h = state20 / 1487847900
    p23 = 6199136374 * (state20 - h * 1487847900) -
        h * 985240079
    h = state22 / 293855150
    p21 = 31387477935 * (state22 - h * 293855150) -
        h * 143639429
    if (p23 < 0)
        p23 += 9223372036854754679
    if (p21 < 0)
        p21 += 9223372036854754679 - p23
    else
        p21 -= p23
    if (p21 < 0)
        p21 += 9223372036854754679
    state20 = state21
    state21 = state22
    state22 = p21

    if (p12 > p21)
        return p12 - p21
    else
        return p12 - p21 + 9223372036854769163
}
```

Listing 5: `mrg31k3p` pseudocode

```
uint64 state10, state11, state12
uint64 state20, state21, state22

mrg31k3p_seed(uint64 seed){
    state10 = seed
    state11 = seed
    state12 = seed
```

```
    state20 = seed
    state21 = seed
    state22 = seed
    if(seed == 0){
        state10++
        state21++
    }
    if (state10 > 2147483647) state10 -= 2147483647
    if (state11 > 2147483647) state11 -= 2147483647
    if (state12 > 2147483647) state12 -= 2147483647
    if (state20 > 2147462579) state20 -= 2147462579
    if (state21 > 2147462579) state21 -= 2147462579
    if (state22 > 2147462579) state22 -= 2147462579
}

uint32 mrg31k3p_next_uint32(){
    ulong y1, y2

    y1 = (((state11 & 511) << 22) + (state11 >> 9)) +
        (((state12 & 16777215) << 7) + (state12 >> 24))
    if (y1 > 2147483647){
        y1 -= 2147483647
    }
    y1 += state12
    if (y1 > 2147483647){
        y1 -= 2147483647
    }
    state12 = state11
    state11 = state10
    state10 = y1

    y1 = ((state20 & 65535) << 15) + 21069 * (state20 >> 16)
    if (y1 > 2147462579){
```

```
        y1 -= 2147462579
    }
    y2 = ((state22 & 65535) << 15) + 21069 * (state22 >> 16)
    if (y2 > 2147462579){
        y2 -= 2147462579
    }
    y2 += state22
    if (y2 > 2147462579){
        y2 -= 2147462579
    }
    y2 += y1
    if (y2 > 2147462579){
        y2 -= 2147462579
    }
    state22 = state21
    state21 = state20
    state20 = y2

    if (state10 <= state20){
        return state10 - state20 + 2147483647
    }
    else{
        return state10 - state20
    }
}
```

Listing 6: `lfib` pseudocode

```
uint64 state[17]
uint32 state_p1, state_p2

void lfib_seed(uint64 seed){
    state_p1 = 17
    state_p2 = 5
```

```
    for (int i = 0; i < 17; i++){
        seed=6906969069 * seed + 1234567
        state[i] = seed | 1
    }
}

uint64 lfib_next_uint64(){
    state_p1--
    if(state_p1 < 0){
        state_p1=16
    }
    state_p2--
    if(state_p2 < 0){
        state_p2 = 16
    }
    state[state_p1] *= state[state_p2]
    return state[state_p1]
}
```

Listing 7: `xorshift1024` pseudocode

```
uint32 state[54]
uint32 state_i

xorshift1024_seed(uint64 seed){
    state_i = 0
    if (seed == 0) {
        seed++
    }
    for (int i = 0; i < 54; i++){
        state[i] = 0
        if (11 <= i && i < 43) {
            state[i] = seed
        }
```

```
    }
}

uint32 xorshift1024_next_uint32(){
    state_i++
    if(state_i < 43){
        return state[state_i]
    }
    state_i=11
    for (i = 11:43){
        state[i] ^= (state[i - 10] << 9) ^ state[i - 9] >> 23
    }
    for (i = 11:43){
        state[i] ^= (state[i + 10] << 5) ^ state[i + 9] >> 27
    }
    for (i = 11:43){
        state[i] ^= (state[i - 10] << 24) ^ state[i - 9] >> 8
    }
    return state[state_i]
}
```

Listing 8: `xorshift6432star` pseudocode

```
uint64 state

xorshift6432star_seed(uint64 seed){
    if(seed == 0){
        seed++
    }
    stat e =seed
}

uint32 xorshift6432star_next_uint32(){
    state ^= state >> 12
```

```
    state ^= state << 25
    state ^= state >> 27
    return (uint32)((state * 2685821657736338717) >> 32)
}
```

Listing 9: `mt19937` pseudocode

```
uint32 state[624]
uint32 state_i

void mt19937_seed(uint32 s){
    state[0] = s
    for (i = 1; i < 624; i++) {
        state[i] = 1812433253 *
            (state[i-1] ^ (state[i-1] >> 30)) + i
    }
    state_1 = i
}

uint32 mt19937_next_uint32(){
    uint32 y
    uint32 mag01[2] = {0, 2567483615}

    if(state_i < 624 - 397){
        y = (state[state_i] & 2147483648) |
            (state[state_i + 1] & 2147483647)
        state[state_i] = state[state_i + 397] ^ (y >> 1) ^
            mag01[y & 1]
    }
    else if(state_i < 624 - 1){
        y = (state[state_i] & 2147483648) |
            (state[state_i + 1] & 2147483647)
        state[state_i] = state[state_i + (397 - 624)] ^ (y >> 1)
            ^ mag01[y & 1]
```

```
    }
    else{
        y = (state[624 - 1] & 2147483648) |
            (state[0] & 2147483647)
        state[624 - 1] = state[397 - 1] ^ (y >> 1) ^
            mag01[y & 1]
    }
    y = state[state_i++]
    state_i %= 624

    y ^= (y >> 11)
    y ^= (y << 7) & 2636928640
    y ^= (y << 15) & 4022730752
    y ^= (y >> 18)

    return y
}
```

Listing 10: `tinymt64` pseudocode

```
uint64 state0, state1

tinymt64_seed(uint64 seed){
    uint64 status[2]
    status[0] = seed ^ ((uint64)4194639680 << 32)
    status[1] = 4291887092 ^ 6399667842752446396
    for (int i = 1; i < 8; i++) {
        status[i & 1] ^= i + 6364136223846793005 *
            (status[(i - 1) & 1] ^
            (status[(i - 1) & 1] >> 62))
    }
    state0 = status[0]
    state1 = status[1]
```

```
    if ((state0 & 9223372036854775807) == 0 &&
        state1 == 0) {
        state0 = 84
        state1 = 77
    }
}

uint64 tinymt64_next_uint64(){
    uint64 x
    state0 &= 9223372036854775807
    x = state0 ^ state1
    x ^= x << 12
    x ^= x >> 32
    x ^= x << 32
    x ^= x << 11
    state0 = state1
    state1 = x
    if (x & 1) {
        state0 ^= 4194639680
        state1 ^= 4291887092 << 32
    }

    x = state0 + state1
    x ^= state0 >> 8
    if (x & 1) {
        x ^= 6399667842752446396
    }
    return x
}
```

Listing 11: `tinymt32` pseudocode

```
uint32 state0, state1, state2, state3
```

```
tinymt32_seed(uint64 seed){
        uint32 status[4]
    status[0] = seed
    status[1] = 2406486510
    status[2] = 4235788063
    status[3] = 932445695
    for (int i = 1; i < 8; i++) {
        status[i & 3] ^= i + 1812433253
            * (status[(i - 1) & 3]
            ^ (status[(i - 1) & 3] >> 30))
    }
    state0 = status[0]
    state1 = status[1]
    state2 = status[2]
    state3 = status[3]
    if ((state0 & 2147483647) == 0 &&
        state1 == 0 &&
        state2 == 0 &&
        state3 == 0) {
        state0 = 84
        state1 = 73
        state2 = 78
        state3 = 89
    }
    for (int i = 0; i < 8; i++) {
        tinymt32_next_int()
    }
}

uint32 tinymt32_next_uint32(){
    uint32 x = (state0 & 2147483647) ^ state1 ^ state2
    uint32 y = state3
    uint32 t0, t1
```

```
    x ^= x << 1
    y ^= (y >> 1) ^ x
    state0 = state1
    state1 = state2
    state2 = x ^ (y << 10)
    state3 = y
    if (y & 1) {
        state1 ^= 2406486510
        state2 ^= 4235788063
    }
    t0 = state3
    t1 = state0 + (state2 >> 8)
    t0 ^= t1
    if (t1 & 1) {
        t0 ^= 932445695
    }
    return t0
}
```

Listing 12: `well512` pseudocode

```
uint32 state[16]
uint32 state_i

well512_seed(uint64 seed){
    state_i = 0
    for (int i = 0; i < 16; i+=2){
        seed = 6906969069 * seed + 1234567
        state[i] = seed
        state[i + 1] = seed >> 32
    }
}

uint32 well512_next_uint32(){
```

```
    uint32 z0, z1, z2
    z0 = state[(state_i+15) & 15]
    z1 = state[state_i] ^ (state[state_i] << 16) ^
        state[(state_i+13) & 15] ^
        (state[(state_i+13) & 15] << 15)
    z2 = state[(state_i + 9) & 15] ^
        (state[(state_i + 9) & 15] >> 11)
    state[state_i] = z1 ^ z2
    state[(state_i + 15) & 15] = z0 ^ (z0 << 2) ^ z1 ^
        (z1 << 18) ^ (z2 << 28) ^ state[state_i] ^
        ((state[state_i] << 5) & 3661901092)
    state_i = (state_i + 15) & 15
    return state[state_i]
}
```

Listing 13: `msws` pseudocode

```
uint64 state_x, state_w

msws_seed(uint64 seed){
    state_x = seed
    state_w = seed
}

uint32 msws_next_uint32(){
    state_x *= state_x
    state_w += 13091206342165455529
    state_x += state_w
    return state_x = (state_x >> 32) | (state_x << 32)
}
```

Listing 14: `philox2x32_10` pseudocode

```
uint32 state_l, state_r
```

```
philox2x32_10_seed(uint64 seed){
    state_l = seed
    state_r = seed >> 32
}

uint32 philox2x32_10_next_uint32(){
    state_r++
    if(state_r == 0){
        state_l++
    }
    uint32 l=state_l
    uint32 r=state_r
    uint32 key = 12345
    for(uint32 i = 0; i < 10; i++){
        uint32 tmp = r * 3528905107
        r = mul_hi(r, 3528905107) ^ l ^ key
        l = tmp
        key += 2654435769
    }
    return ((uint64)l) << 32 | r
}
```

Listing 15: `tyche` pseudocode

```
uint64 state

tyche_seed(uint64 seed){
    state_a = seed >> 32
    state_b = seed
    state_c = 2654435769
    state_d = 1367130551
    for(uint32 i = 0; i < 20; i++){
        tyche_next_uint64()
```

```
    }
}

uint64 tyche_next_uint64(){
    state_a += state_b
    uint32 tmp
    tmp = state_d ^ state_a
    state_d = tmp << 16 | tmp >> 16
    state_c += state_d
    tmp = state_b ^ state_c
    state_b = tmp << 12 | tmp >> 20
    state_a += state_b
    tmp = state_d ^ state_a
    state_d = tmp << 8 | tmp >> 24
    state_c += state_d
    tmp = state_d ^ state_a
    state_b = tmp << 7 | tmp >> 25
    return ((uint64)state_a) << 32 | state_b
}
```

Listing 16: `tyche_i` pseudocode

```
uint64 state

tyche_i_seed(uint64 seed){
    state_a = seed >> 32
    state_b = seed
    state_c = 2654435769
    state_d = 1367130551
    for(uint32 i = 0; i < 20; i++){
        tyche_i_next_uint64()
    }
}
```

```
uint64 tyche_i_next_uint64(){
    state_b = (state_b << 7 | state_b >> 25) ^ state_c
    state_c -= state_d
    state_d = (state_b << 8 | state_b >> 24) ^ state_a
    state_a -= state_b
    state_b = (state_b << 12 | state_b >> 20) ^ state_c
    state_c -= state_d
    state_d = (state_b << 16 | state_b >> 16) ^ state_a
    state_a -= state_b
    return ((uint64)state_a) << 32 | state_b
}
```

Listing 17: `isaac` pseudocode

```
uint32 state_rr[256]
uint32 state_mm[256]
uint32 state_aa
uint32 state_bb
uint32 state_cc
uint32 state_i

isaac_seed(uint64 seed){
    state_aa = seed
    state_bb = seed ^ 123456789
    state_cc = seed + 123456789
    state_i = 256
    for(int i = 0; i < 256; i++){
        seed = 6906969069 * seed + 1234567
        state_mm[i] = seed
    }
}

uint32 isaac_next_uint32(){
    if(state_i == 256){
```

```
uint32 a, b, x, y, *m, *m2, *r, *mend
m = state_mm
r = state_rr
a = state_aa
b = state_bb + (++state_cc)
for (m = state_mm, mend = m2 = m+128; m < mend; ){
    x = *m
    a = (a ^ (a << 13)) + *(m2++)
    *(m++) = y = *(uint32 *)((uint8 *)state_mm +
        (x & 1020)) + a + b
    *(r++) = b = *(uint32 *)((uint8 *)state_mm +
        ((y >> 8) & 1020)) + x
    x = *m
    a = (a ^ (a >> 6)) + *(m2++)
    *(m++) = y = *(uint32 *)((uint8 *)state_mm +
        (x & 1020)) + a + b
    *(r++) = b = *(uint32 *)((uint8 *)state_mm +
        ((y >> 8) & 1020)) + x
    x = *m
    a = (a ^ (a << 2)) + *(m2++)
    *(m++) = y = *(uint32 *)((uint8 *)state_mm +
        (x & 1020)) + a + b
    *(r++) = b = *(uint32 *)((uint8 *)state_mm +
        ((y >> 8) & 1020)) + x
    x = *m
    a = (a ^ (a >> 16)) + *(m2++)
    *(m++) = y = *(uint32 *)((uint8 *)state_mm +
        (x & 1020)) + a + b
    *(r++) = b = *(uint32 *)((uint8 *)state_mm +
        ((y >> 8) & 1020)) + x
}
for (m2 = state_mm; m2 < mend; ){
    x = *m
```

```
            a = (a ^ (a << 13)) + *(m2++)
            *(m++) = y = *(uint32 *)((uint8 *)state_mm +
                (x & 1020)) + a + b
            *(r++) = b = *(uint32 *)((uint8 *)state_mm +
                ((y >> 8) & 1020)) + x
            x = *m
            a = (a ^ (a >> 6)) + *(m2++)
            *(m++) = y = *(uint32 *)((uint8 *)state_mm +
                (x & 1020)) + a + b
            *(r++) = b = *(uint32 *)((uint8 *)state_mm +
                ((y >> 8) & 1020)) + x
            x = *m
            a = (a ^ (a << 2)) + *(m2++)
            *(m++) = y = *(uint32 *)((uint8 *)state_mm +
                (x & 1020)) + a + b
            *(r++) = b = *(uint32 *)((uint8 *)state_mm +
                ((y >> 8) & 1020)) + x
            x = *m
            a = (a ^ (a >> 16)) + *(m2++)
            *(m++) = y = *(uint32 *)((uint8 *)state_mm +
                (x & 1020)) + a + b
            *(r++) = b = *(uint32 *)((uint8 *)state_mm +
                ((y >> 8) & 1020)) + x
        }
        state_bb = b
        state_aa = a

        state_i = 0
    }
    return state_rr[state_i++]
}
```

Listing 18: `kiss99` pseudocode

```
uint32 state_z, state_w, state_jsr, state_jcong

kiss99_seed(uint64 seed){
    state_z = 362436069 ^ seed
    if(state_z == 0){
        state_z = 1
    }
    state_w = 521288629 ^ (seed >> 32)
    if(state_w == 0){
        state_w = 1
    }
    state_jsr = 123456789 ^ seed
    if(state_jsr == 0){
        state_jsr = 1
    }
    state_jcong = 380116160 ^ (seed >> 32)
}

uint32 kiss99_next_uint32(){
    state_z = 36969 * (state_z & 65535) + (state_z >> 16)
    state_w = 18000 * (state_w & 65535) + (state_w >> 16)

    state_jsr ^= state_jsr << 17
    state_jsr ^= state_jsr >> 13
    state_jsr ^= state_jsr << 5

    state_jcong = 69069 * state_jcong + 1234567

    return (((state_z << 16) + state_w) ^ state_jcong) +
        state_jsr
}
```

Listing 19: `kiss09` pseudocode

```
uint64 state_x, state_c, state_y, state_z

kiss09_seed(uint64 seed){
    state_x = 1234567890987654321 ^ j
    state_c = 123456123456123456 ^ j
    state_y = 362436362436362436 ^ j
    if(state_y == 0){
        state_y = 1
    }
    state_z = 1066149217761810 ^ j
}

uint64 kiss09_next_uint64(){
    uint64 t = (state_x << 58) + state_c
    state_c = state_ x >> 6
    state_x += t
    state_c += state_x < t

    state_y ^= state_y << 13
    state_y ^= state_y >> 17
    state_y ^= state_y << 43

    state_z = 6906969069 * state_z + 1234567
    return state_x + state_y + z
}
```