2023 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
MODELING SIMULATION & SOFTWARE (MS2) TECHNICAL SESSION
AUGUST 15-17, 2023 - NOVI, MICHIGAN

EVALUATION OF HASH-SEEDED PSEUDO-RANDOM NUMBER GENERATORS IN PARALLEL ENVIRONMENTS

John Kaniarz[1], Mark Brudnak, PhD[1]

[1]US Army DEVCOM-GVSC, Warren, MI

ABSTRACT

A customized approach to Pseudo Random Number Generation (PRNG) is developed specifically for the highly parallelizable sensor models in the ground vehicle autonomy application domain. The work considers three desirable attributes (namely quality, efficiency and determinism). Furthermore, the application demands high fanout (1:1Million+) seeding of traditional PRNGs. An approach using hash functions to generate the seeds for the PRNGs, each of which generates a small (i.e. 20) run of numbers, to handle determinism is investigated. Quality and efficiency are evaluated for multiple combinations of hash functions and PRNGs and a pareto front is created. Quality assessments were performed using industry standard testing suites (TestU01 and PractRand) and efficiency of various hash, PRNG, and batch size combinations was benchmarked on Windows/x64, ARM and NVIDIA/CUDA architectures.

Citation: J. Kaniarz, M. Brudnak, "Evaluation of Hash-Seeded Pseudo-Random Number Generators in Parallel Environments," In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 15-17, 2023.

1. INTRODUCTION

Modeling and Simulation (M&S) is a precise discipline because the underlying technologies are precise, namely computers and programming languages. As such M&S practitioners can encode mathematical laws into implementations which easily compute using 15 significant digits of precision. Furthermore, the data which feeds these models are considered equally precise; planes are perfectly flat, joints contain no slack, bodies are perfectly rigid, terrain is composed of connected polygons, etc. These simplifying assumptions make computation and representation easier and more computable and very often provide a very good first order approximation to reality. In other cases, these assumptions oversimplify reality and make even a first order assumption in adequate. One aspect of modeling which is underserved by this precision is the representation and modeling of natural environments. In this case the system to be modeled is imprecise, complex, and stochastic. It is therefore imperative that

the modeling of any system which interacts with a natural environment either through contact (i.e. tire/track) or through sensing (i.e. camera, LIDAR, radar) in some way reflect this imprecision which we regard as uncertainty around simplified yet precise models. Typically, uncertainty is introduced in these cases by a source of randomness such a Pseudo Random Number Generator (PRNG).

In this work we are seeking to add randomness to sensor models which are used in the autonomous systems domain, specifically cameras and LIDARs. Simulations of these sensor systems are highly parallelizable in that each individual point in a raster or scan is independent of the others. This work was motivated by the need for a high-quality PRNG which could work with a highly parallelized LIDAR model. In this use case the following characteristics are essential:

1. High within sample quality
2. High between sample quality
3. High over-time quality
4. Determinism
5. Parallelism
6. Efficiency

With respect to quality, we intend the statistical quality of non-correlation. By determinism we mean that given the same global seed value, the actual random number are well-determined for each time step, each beam and each value within a beam. By parallelism, we mean that the PRNG does not depend on a shared state across parallel wok units (i.e. threads). Because a single LIDAR scan may have on the order of 1 million individual beams, we don't want to depend on a shared seed or serialize the generation of random numbers. By efficiency we mean computational cost.

Most programming languages and/or their standard libraries contain facilities for the generation of random numbers. These facilities typically implement one of the well-known algorithms such as Linear Congruential Generator (LCG) which takes the form of

$$x_{n+1} = (ax_n + c) \bmod m$$

where the prior value of the generator x_n is often referred to as the seed, and a, c, and m are chosen to provide long periods in the cyclic nature of modular arithmetic. These types of generators are fast to compute, being composed of a few integer arithmetic operations. As such, they are designed to provide a single sequence of random numbers due to the shared state as represented by the seed. So LCGs provide efficiency but are not well suited to parallel use because of the shared seed. To achieve parallelism and determinism, we sought a method which would allow each work item to maintain its own seed for efficiency, but make the seed deterministically depend on the time step and the work item.

1.1. Related work

Random number generators have been extensively researched and studied. Surveys of methods may be found on-line [1]. Press et al. [2] also provide a good discussion of the topic. Matsumoto, et al. [3] examine the problem of poor choice in the seeding of PRNG algorithms when employed in a parallel configuration. More recently Jarzynski, et al. [4] examined hash functions as suitable sources of PRNGs which they evaluated on GPU hardware for both efficiency and quality. Steele and Vigna [5] recently discuss the choice of multipliers in the linear congruential PRNGs. Manssen, et al. [6] discuss the practical implementation of popular PRNG on GPU hardware and present their own algorithm designed to be efficient on CUDA hardware.

The approach to parallel independent streams of PRNG has been addressed by several researchers. Mascagni and Srivivasan developed an algorithm for generating parameterized PRNGs in the

context of Monte Carlo simulation. Steele, et al. [7] present an algorithm which they call "SplitMix" which allows a deterministic split of the stream of random number generators. In their approach, this splitting behavior can be arbitrarily deep, but it couples the generation and splitting functions. It is based on the DotMix algorithm of Leiserson, et al. [8] which introduces the idea of pedigrees to handle the branching of random number generator seeds. Subsequently Steele and Vigna improve on the SplitMix algorithm with the LXM family or PRNG algorithms. This family improves the PRNG with longer periods (up to 2^{128}) along with a better mixing function. Salmon, et al. [9] approach the problem of splitting using a simple counter and introduce a new PRNG which they call Philox. Phillips, et al. [10] address the problem of GPU based PRNGs with small batches over thousands of threads in the context of simulating Brownian Dynamics. L'Ecuyer, et al. [11], [12] also examine PRNGs in parallel on GPU hardware and provide an excellent survey of prior work on the topic.

Finally, regarding the testing of PRNGs with respect to statistical performance several options are available. Srinivasan, et al. [13] give a good overview of the qualities which make a good PRNG and outline some testing approaches. An early test suite called DIEHARD was developed by Marsaglia [14] which was subsequently improved by Brown, et al. [15] as Dieharder. L'Ecuyer and Simard [16] developed a test suite called TestU01 which used provides several statistical tests for a PRNG stream. The test suite provides different batteries of tests named "SmallCrush", "Crush" and "BigCrush" with 15, 144, and 160 metrics respectively. Finally, the PractRand suite developed by Cook [17] provides a software suite designed to interface with a user-provided PRNG.

1.2. Summary

The remainder of this paper presents our approach to seeding PRNGs for each work item. Based on our approach, we then seek to find high quality hash functions and PRNGs which work well in our high fan-out, low evaluation use case. We describe the contenders and our inclusion/exclusion criteria. We then present our evaluation methods. Results of the analysis are presented and the paper finished with our conclusions and lessons learned.

2. OUR APPROACH

Our approach (see Figure 1) is a hybrid of DotMix and SplitRand's approach with inspiration from Philox. DotMix creates independent sequential PRNG for each task initialized with a hash of a unique identifier. This unique identifier, called a pedigree, is a list of counters. The pedigree of a task is the pedigree of its parent concatenated with N, where N is the N^{th} subtask spawned.

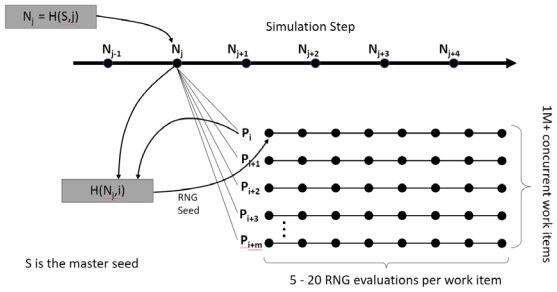

Figure 1. Topology of the Hash-Seeded PRNG approach for high-fanout, low batch application.

Understanding the difference between DotMix and our approach first requires a change in terminology. Instead of tracking task dependencies and threads, we track *processing steps* and *work items* respectively. A work-item's identifier (e.g., its pixel number) is constant irrespective of what core or in what order the work is performed. So, our pseudo-pedigree in a multi-camera simulation might look like this: [frame#, camera#, pixel#].

If we stopped at this stage, we could use the DotMix algorithm. But as Steele showed in

the SplintRand algorithm [7], there are much more efficient ways of hashing. Alternately we could take these numbers, plus a counter, as the 4 parameters to Philox4x32. However, in practical implementation, the pedigree would have more than 4 items, as shown in Figure 2, and would not map so easily. Also, as we show later, there are faster methods than using Philox.

1. Program Startup
   1.1. Loading
2. Program Execution
   2.1. Simulation Initialization
   2.2. Simulation Run
      2.2.1.Timestep 1
         2.2.1.1. Simulation Global Update
         2.2.1.2. Sensor Updates
            2.2.1.2.1. Camera1 Update
               2.2.1.2.1.1. Pixel1 update
               2.2.1.2.1.2. Pixel2 update
               2.2.1.2.1.3. …
            2.2.1.2.2. Camera2 Update
               2.2.1.2.2.1. …
            2.2.1.2.3. Lidar Update
               2.2.1.2.3.1. Laser 1 update
               2.2.1.2.3.2. Laser2 update
               2.2.1.2.3.3. …
      2.2.2.Timestep 2
         2.2.2.1. …
   2.3. Simulation Cleanup
3. Program Shutdown

Figure 2. Pedigrees (i.e. numbers of the form X.Y.Z…) shown for the levels of nesting in this application.

What Steele noticed about hashing the pedigree is we don't need to hash the whole pedigree once it's completely known. Instead, SplitRand maintains a running hash state, shared across all items at the same level. Then for each work item, it finishes the hash independently. Furthermore, it doesn't use a hashing function capable of variable length inputs, it instead creates a hash chain using by hashing a mix of the parent tasks hash and the sub-tasks counter. For our algorithm we do the same but use a 2→1 reducing hashing function a counter based PRNG (CBPRNG). The formula is:

$$\text{state}_n = H(\text{state}_{n-1}, \text{counter})$$

where state_0 is the global seed for the program and n is the n^{th} item in the pedigree. On a leaf node of our dependency tree the PRNG is seeded with state_n.

Not any hashing function is suitable for this purpose. A hash collision will result in two PRNGs being initialized with the same seed which will cause two pixels to have the exact same sequence of random numbers which will result in visible artifacts.

To avoid this, we need the hash function to be a bijection but, by definition, a reducing function cannot be a bijection. As a practical workaround, we require the hashing function to be bijective in each argument. More specifically, we need a function of multiple arguments and, with respect to each argument, the function is a bijection when all other arguments are held constant. In the case of a two-argument function, you can think of this as a collection of bijections on one parameter selected by the other parameter.

With this requirement we can show that there can be no meaningful collisions within the domain of camera simulation. In the case of individual pixels: each PRNG is initialized with $H(\text{camera\_seed}, \text{pixel\#})$ since camera seed is constant across the camera/frame, each pixel is guaranteed a unique seed. And when viewed temporally, pixel# is constant across frames and therefore each pixel will always get a new seed for each frame. In the case of stereo camera pairs, the same holds for the same pixel in different cameras in the same frame as they differ only by camera#.

At one level up, each camera will have a unique seed as calculated by $H(\text{frame\_seed}, \text{camera\#})$ where frame_seed is constant at a given time step. It should be noted that different pixels at different timesteps, or different pixels in

different cameras may still collide as two or more of the parameters in the hashing chain have changed. However, with 2^{64} possible seeds, collisions will be extremely rare and with the seeds being used for unrelated PRNGs, the collisions should be unnoticeable in the simulation output.

As an additional optimization, a different, faster hash function may be used at the leaf nodes. Users may want to do this because the final hash function in the chain is used many orders of magnitude more often than the hash functions leading up to it. The high entropy output of the preceding hash functions mitigates the low-entropy input weakness of lower quality hash functions. Alternately, a PRNG that offers separate streams may be used with no final hash at the leaf node. The PRNG's seeding function assumes the duties of the hash.

3. SELECTION OF HASH FUNCTIONS AND PRNGS.

As demonstrated by the wealth of options as collected by Jarzynski [4], creating new PRNG and hashing functions is a popular pastime among scientists and hobbyists alike. Also shown by Jarzynski is that most don't hold up to scrutiny.

Because we are testing hash/PRNG combinations, the number of tests we will ultimately have to run grows geometrically. Therefore, we limit our consideration to functions that are either: popular, included in popular software, or well-regarded in the literature. From this list we further eliminate functions that are cryptographic (slow), have large state (>128bits), are floating-point based, or are ill-suited for GPU compute.

3.1. Hash Functions

The list of hash functions considered in this work are contained in Table 1.

For the 1→1 hashes (denoted by h()) in this list, we convert them into 2→1 hashes (denoted by H(x, y)) with the methods collected by Jarzynski and described in the following sections. Note: The caret (^) symbol denotes a bitwise exclusive-or operation as defined in the C family of computer languages.

Nesting Method

This hash is implemented as:

$$H(x, y) = h(x \wedge h(y))$$

In Jarzynski's study, this improves the quality of poor hashes at the expense of doubling the computational effort as it requires two evaluations of h().

Dot Product Method

This hash is implemented as:

$$H(x, y) = h(x * C_1 + y * C_2)$$

Table 1 Hash functions evaluated.

Hash Name	Description
DotMix	The N→1 hash used in the Pedigrees architecture. [8] We only use it as a 2→1 hash.
Iqint1	A 1→1 hash function recommended by Jarzynski [4]
Lea64	A variant of the Murmur hash finalizer. A 1→1 mixing function optimized for 32-bit architectures. Used in SplitRand [7]
MoreMur	A variant of the Murmur hash finalizer. A 1→1 mixing function with constants optimized for higher-order avalanche. The best of the known variants. [18]
Nasam	A 1→1 mixing function similar in form to the Murmur hash finalizer but with more operations. [19]
Xxhash32	A hash recommended by Jarzynski [4]
Various PRNG [20]	Any full-period PRNG of the form $\text{state}_n = f(\text{state}_{n-1})$ is also a bijection and suitable for use as a 1→1 hash.

where C_1 and C_2 are co-prime. In our implementation we used the constants in the "dot" portion of the DotMix hash as they are shown to be collision resistant [8].

Linear Method:

This hash is implemented as:

$$H(x, y) = h(y * A + x)$$

which is a simplification of the dot product method and the fastest combination method, but Jarzynski found that it weakens the strength of the hash. In our implementation we make some minor tweaks to improve its avalanche properties (i.e. the probability of a given bit to flip on the output, given a single bit flip on the input). First, because the form is identical to LCG, we use a spectrally good A from Steele [5] instead of any old prime. In addition, because the counter (y) has less entropy than the seed, we multiply the counter by A to whiten it instead of the seed.

Xor-encrypt-xor (XEX) Method:

This hash is implemented as:

$$H(x, y) = x \wedge h(x \wedge y)$$

This method was not included in Jarzynski but it is a common mode of operation for block encryption cyphers and simple enough for inclusion. We later noticed that XEX is a bijection with respect to only one parameter and this inadequacy shows up in our test results. This method is incompatible with our architecture but is included here for completeness.

Table 2 PRNG functions evaluated.

PRNG Name	Description
LCG variants	For the 64-bit generator we use Knuth's constants. For the 32-bit generator we use the constants from Numerical Recipes.
PCG Variants	A family of PRNG functions [21]
Philox Variants	A family of CBPRNG functions. Optimized for GPU Compute. [9]
Xorshift Variants	A subset of linear-feedback shift registers which are efficient to implement on modern processors. [20]

3.2. PRNGs

The PRNG functions which we included in this analysis are shown in Table 2. We also have two customized variants: lcg64top32 and philox2x32x10rev. The first is the same as our 64-bit LCG except the low 32 bits are discarded as suggested by O'Neill as a cheap improvement. [21] The second is Philox2x32x10 with the inputs reversed to make the key 64 bits and the counter 32 bits.

4. TESTING APPROACH

TestU01 [16] and PractRand [17] are two test suites that we used to measure the quality of our system. TestU01 is the gold standard in the scientific community as its tests are based on common measurements and processes. Its weakness is that many of its tests are floating-point-based which makes it less sensitive to weaknesses in the low order bits. PractRand, in comparison, treats the input as a stream of bytes. It's primarily looking for patterns in the bits or low variance.

These test batteries are designed for sequential PRNG testing and not parallel testing. So, we converted our parallel output into a sequential stream by concatenating the output of multiple generators until the test was satisfied. Because different applications will use different quantities of numbers per generator, and different generators produce data at different bit sizes, we normalized the output to 32-bit words and retested each hash/PRNG pair at 4, 8, 16, 32, 64, 128, and 256 random words generated per seed.

TestU01's "BigCrush" battery reports 160 metrics which we report as a pass/fail percentage. PractRand runs until it finds a

statistical flaw on any metric. At which point it reports the number of bytes analyzed.

Running BigCrush requires significant computing resources. Roughly 20 core-hours per test. To reduce our compute effort, we first ran the algorithms against “SmallCrush” and then “Crush”. Only the algorithms that passed were promoted to the next level of testing. After seeing consistency of the results across all batch sizes on Crush, we skipped the 8, 32, and 128 words per generator analyses on BigCrush as they are unlikely to yield any new information.

While PractRand is also computationally expensive, it can fail fast. So for it, we only eliminated algorithms that failed SmallCrush.

Philox, which is suitable as a total system replacement, was not paired with any hash functions as it would reduce performance for no benefit.

5. RESULTS

Figure 3 shows the results of our testing using PractRand on x86-64 architecture with 16 numbers generated per work-item. Many algorithm pairs pass the statistical tests and are suitable for use with our architecture. They are lined up at the top of the graph. Amongst these options, the faster algorithms are on the right. The BigCrush results, Arm-64 results, and results for N = 4, 16, 256 paint a similar picture and are available in the Appendix.

Of note, PCG RXS M XS 64 is consistently the fastest or near fastest and it passes without any additional hashing (labeled as “identity” hash). Its separate streams feature provides the differentiation required.

The plots for N = 8, 32, 64, and 128 do not contain any surprises and are omitted for brevity. As a series, the performance of the pair is close to that of the hash for small N, and smoothly transitions toward the

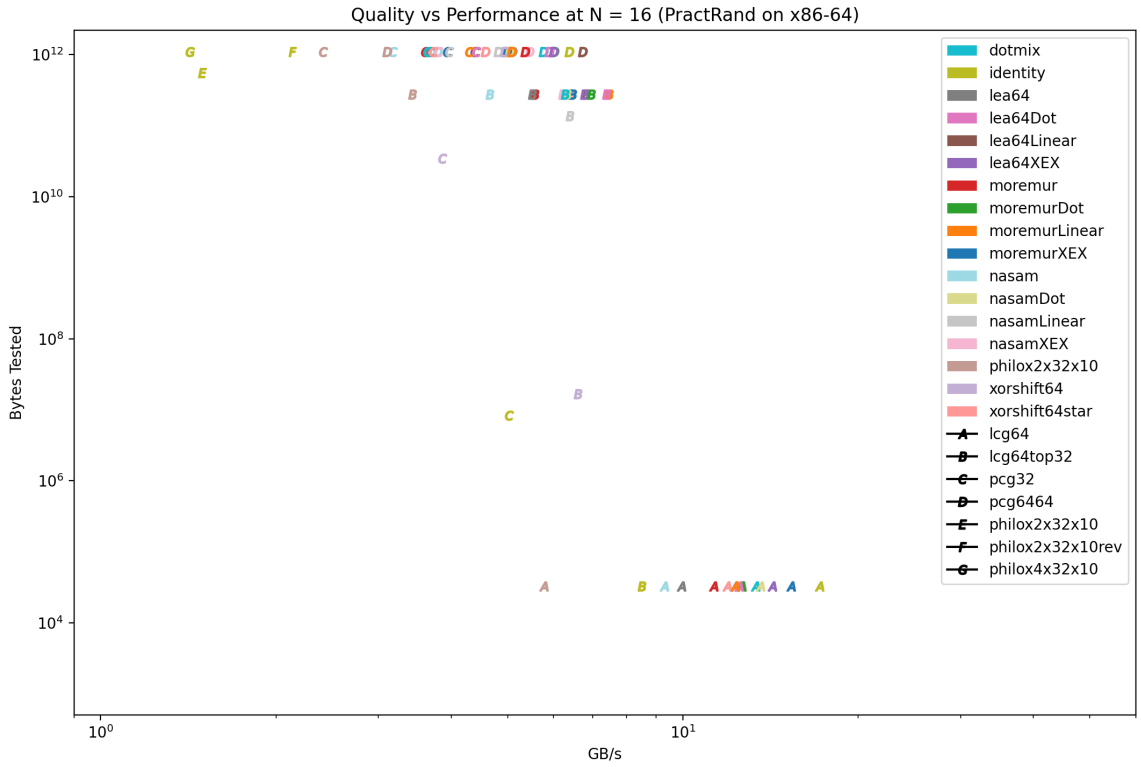

Figure 3 Quality vs performance of hash/PRNG pairs. The hash function is indicated by color; the PRNG by letter. Quality data measured by PractRand. Performance measured on x86-64 architecture.

performance of the PRNG for larger N. We did not find any catastrophic hash/PRNG pairings except for the broad case of mixing 32-bit and 64-bit algorithms.

When run on Arm64 architecture the relative rankings are consistent with x86-64. However, bit-shift heavy algorithms received a noticeable bump in performance. This is unsurprising as ARM machine instructions include “free” bit shifts.

When ran on NVIDIA GPU the results are again very similar with the notable exception being the Philox family of algorithms. Philox was designed to run well on GPU, and it shows.

It is important to note that all the generators with 32 bits of state were found inadequate. There are two reasons for this. The first is that BigCrush analyzes enough data to fail any generator with less than 36 bits of internal state [21]. Secondly, because we initialized 32-bit generators with 64-bit hashes, the reduction of the hash output to 32 bits to initialize the PRNG is not a bijection. Due to the birthday problem, a collision is likely to occur well before 2^{32} work items. 32-bit generators must be seeded with 32-bit hashes to keep our guarantee of limited collision resistance.

6. CONCLUSIONS

Overall, our testing shows that our architecture can provide deterministic parallel pseudo-random numbers when paired with appropriate hash and PRNG functions. It passes industry-standard tests of randomness with many different hash/PRNG combinations providing implementation flexibility. Of the algorithms we tested, PCG RXS M XS 64 is the clear choice for use within this architecture. It is as sound as any other generator to the limits of our testing and is the fastest passing PRNG. Its separate streams feature enables it to eliminate the final hashing step before initializing the generators which provides a small increase in performance.

7. REFERENCES

[1] "Pseudorandom number generator," Wikipedia, [Online]. Available: https:// en.wikipedia.org/ wiki/ Pseudorandom_ number_ generator. [Accessed 16 2 2023].

[2] H. W. Press, A. S. Teukolsky, T. W. Vetterling and P. B. Flannery, Numerical Recipies in C: The Art of Scientific Computing, 2nd ed., New York: Cambridge University Press, 1992.

[3] M. Matsumoto, I. Wada, A. Kuramoto and H. Ashihara, "Common Defects in Initialization of Pseudorandom Number Generators," *ACMTrans.Model. Comput. Simul.,* vol. 17, no. 4, 2007.

[4] M. Jarzynski and M. Olano, "Hash Functions for GPU Rendering," *Journal of Computer Graphics,* vol. 9, no. 3, 2020.

[5] G. L. Steele and S. Vigna, "Computationally easy, spectrally good multipliers for congruential pseudorandom number generators," *Journal of Software: Practice and Experience,* vol. 52, no. 2, pp. 443-458, 2022.

[6] M. W. M. &. H. A. Manssen, "Random number generators for massively parallel simulations on GPU," *The European Physical Journal Special Topics,* vol. 210, pp. 53-71, 2012.

[7] G. L. Steele, D. Lea and C. Flood, "Fast Splittable Pseudorandom Number Generators," in *OOPSLA*, Portland, OR, 2014.

[8] C. E. Leiserson, T. B. Schardl and J. Sukha, "Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms," in *PPoPP*, New Orleans, LA, 2012.

[9] J. K. Salmon, M. A. Moraes, R. O. Dror and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, 2011.

[10] C. L. Phillips, J. A. Anderson and S. C. Glotzer, "Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices," *Journal of Computational Physics,* vol. 230, pp. 7191-7201, 2011.

[11] 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 Simulations,* vol. 135, pp. 3-17, 2017.

[12] P. L'Ecuyer, O. Nadeau-Chamard, Y.-F. Chen and J. Lebar, "Multiple Streamswith Recurrence-Based, Counter-Based, and Splittable Random Number Generators," in *Proceedings of the 2021 Winter Simulation Conference*, Phoenix, AZ, 2021.

[13] A. Srinivasan, M. Mascagni and D. Ceperley, "Testing parallel random number generators," *Parallel Computing,* vol. 29, pp. 69-94, 2003.

[14] G. Marsaglia, "DIEHARD: a battery of tests of randomness," Florida State University, 1995. [Online]. Available: https://en.wikipedia.org/wiki/Diehard_tests. [Accessed 21 02 2023].

[15] R. G. Brown, D. Eddelbuettel and D. Bauer, "Dieharder: A Random Number Test Suite," Duke University, [Online]. Available: https:/ /webhome. phy.duke.edu/ ~rgb/ General/ dieharder/ dieharder.abs. [Accessed 21 02 2023].

[16] P. L'Ecuyer and R. Simard, "TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators," University of Montreal, Montreal, 2013.

[17] J. D. Cook, "PractRand," John D. Cook Consulting, 5 2 2020. [Online]. Available: https://pracrand. sourceforge.net/. [Accessed 21 02 2023].

[18] P. Evensen, "Stronger, better, morer, Moremur; a better Murmur3-type mixer.," 16 12 2019. [Online]. Available: https://mostlymangling.blogspot.com/2019/12/stronger-better-morer-moremur-better.html.

[19] P. Evensen, "NASAM: Not Another Strange Acronym Mixer," 3 1 2020. [Online]. Available: https://mostlymangling.blogspot.com/2020/01/nasam-not-another-strange-acronym-mixer.html.

[20] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software,* vol. 8, pp. 1-6, 2003.

[21] M. E. O'Neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," Harvey Mudd College, Claremont, CA, 2014.

[22] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation," *ACM Transactions on Mathematical Software,* vol. 26, no. 3, pp. 436-461, 2000.

.

Appendix.

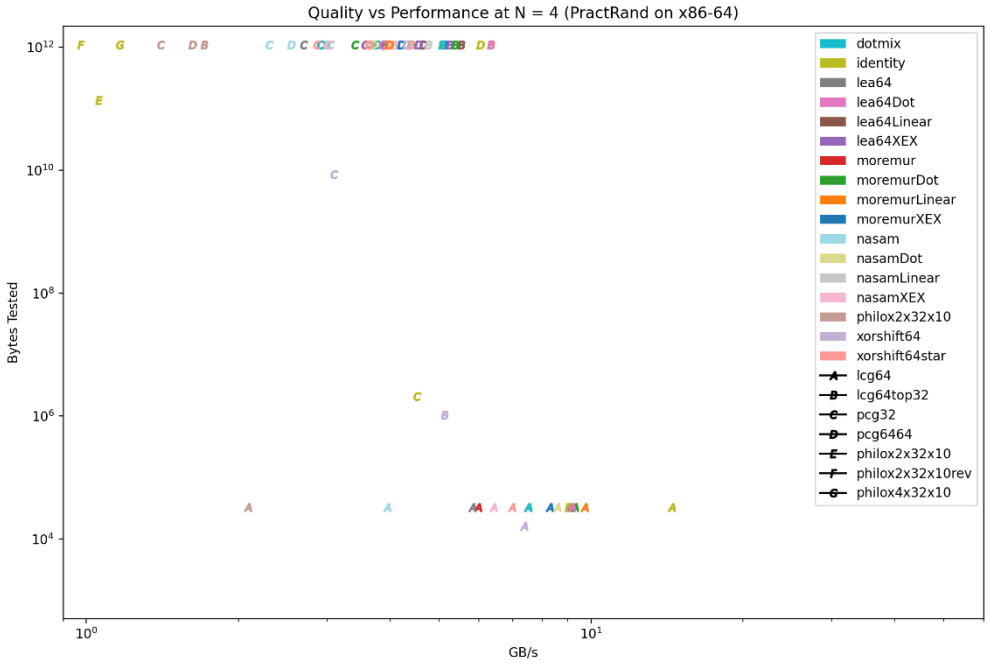

Figure 4 PractRand quality vs performance of hash/PRNG pairs in the small batch case (N=4) on x86-64 architecture.

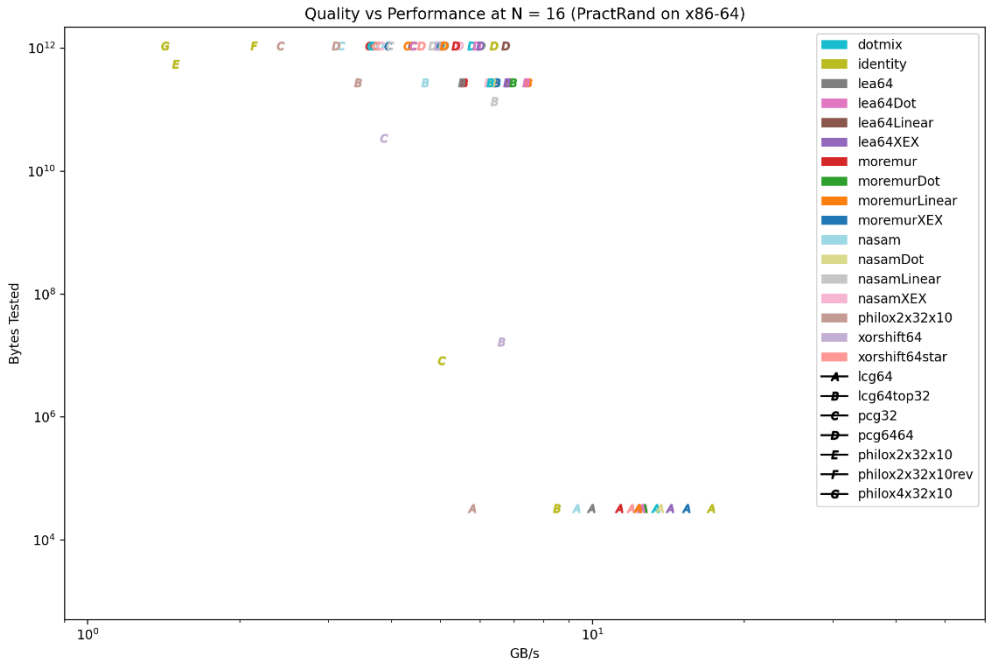

Figure 5 PractRand quality vs performance of hash/PRNG pairs in the medium batch case (N=16) on x86-64 architecture.

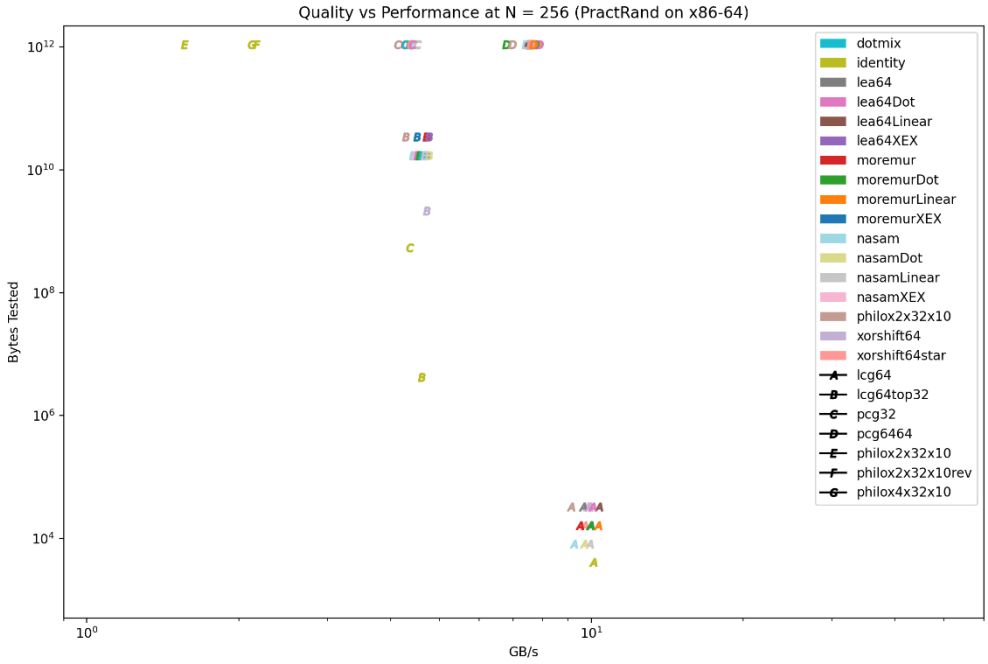

Figure 6 PractRand quality vs performance of hash/PRNG pairs in the Large batch case (N=256) on x86-64 architecture.

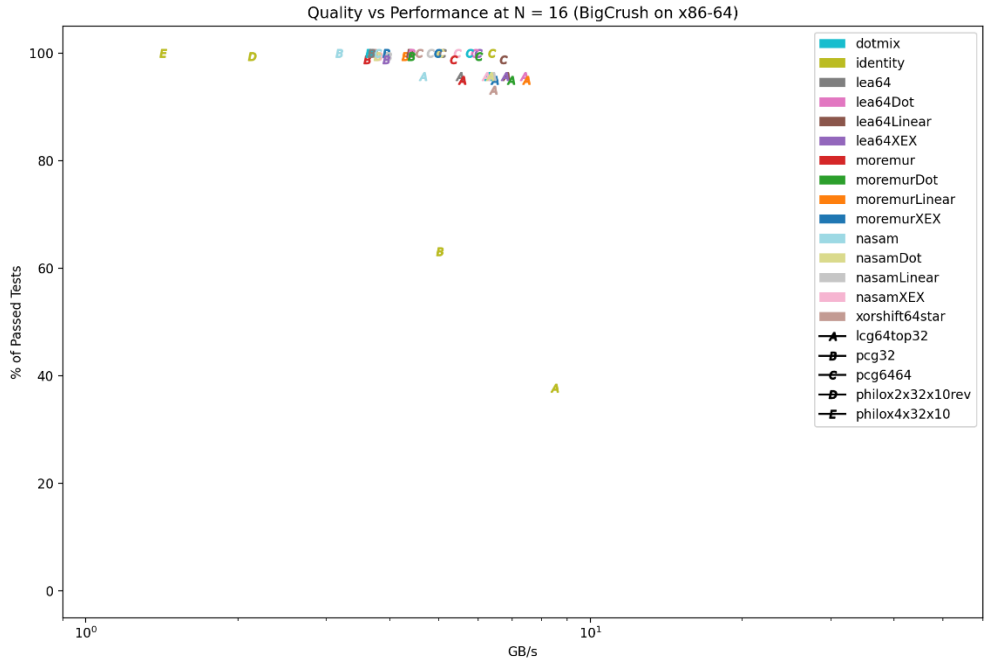

Figure 7 TestU01 (BigCrush) quality vs performance of hash/PRNG pairs in the medium batch case (N=16) on x86-64 architecture.

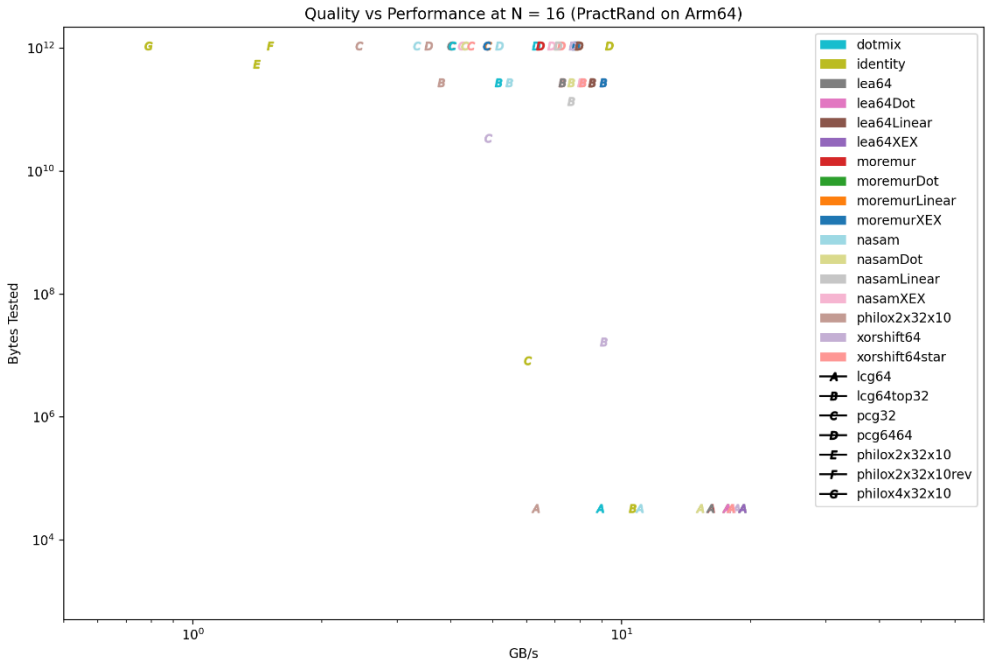

Figure 8 PractRand quality vs performance of hash/PRNG pairs in the medium batch case (N=16) on Arm64 architecture.

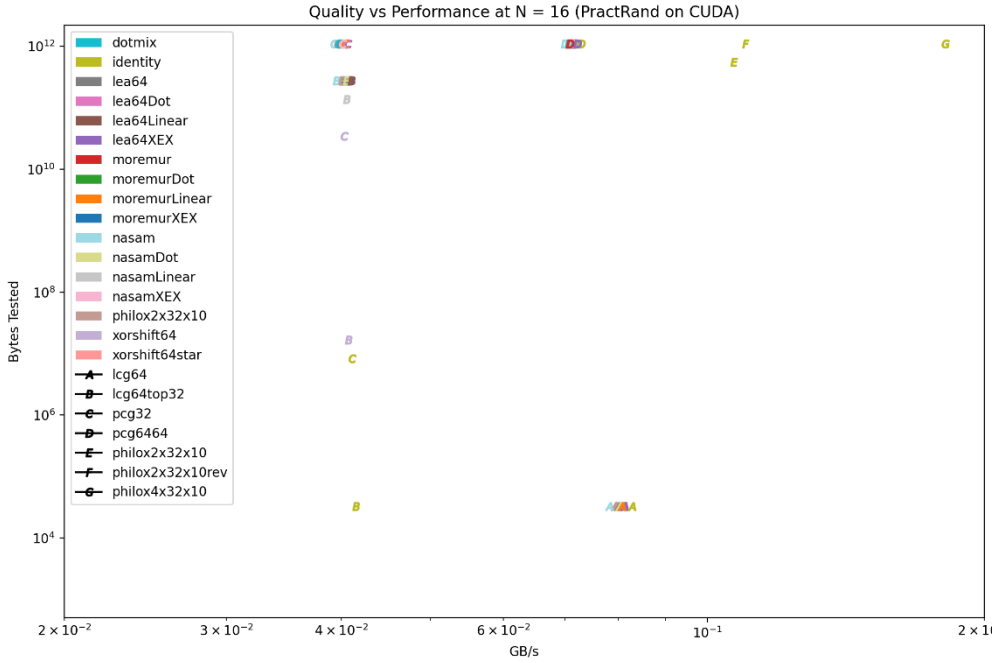

Figure 9 PractRand quality vs performance of hash/PRNG pairs in the medium batch case (N=16) on NVIDIA/CUDA architecture.