EPJ Web of Conferences **295**, 11010 (2024) https://doi.org/10.1051/epjconf/202429511010
CHEP 2023

Fast, high-quality pseudo random number generators for heterogeneous computing

Marco Barbone[1,*], *Georgi* Gaydadjiev[3], *Alexander* Howard[1], *Wayne* Luk[1], *George* Savvidy[2], *Konstantin* Savvidy[2], *Andrew* Rose[1], and *Alexander* Tapper[1]

[1]Imperial College London, South Kensington, London, United Kingdom
[2]Institute of Nuclear and Particle Physics, Demokritos National Research Center, Ag. Paraskevi, GR15342, Athens, Greece
[3]Bernoulli Institute, University of Groningen, Nijenborgh 9, Groningen 9747 AG, The Netherlands

Abstract. Random number generation is key to many applications in a wide variety of disciplines. Depending on the application, the quality of the random numbers from a particular generator can directly impact both computational performance and critically the outcome of the calculation. High-energy physics applications use Monte Carlo simulations and machine learning widely, which both require high-quality random numbers. In recent years, to meet increasing performance requirements, many high-energy physics workloads leverage GPU acceleration. While on a CPU, there exist a wide variety of generators with different performance and quality characteristics, the same cannot be stated for GPU and FPGA accelerators. On GPUs, the most common implementation is provided by cuRAND - an NVIDIA library that is not open source or peer reviewed by the scientific community. The highest-quality generator implemented in cuRAND is a version of the Mersenne Twister. Given the availability of better and faster random number generators, high-energy physics moved away from Mersenne Twister several years ago and nowadays MIXMAX is the standard generator in Geant4 via CLHEP. The MIXMAX original design supports parallel streams with a seeding algorithm that makes it especially suited for GPU and FPGA where extreme parallelism is a key factor. In this study we implement the MIXMAX generator on both architectures and analyze its suitability and applicability for accelerator implementations. We evaluated the results against "Mersenne Twister for a Graphic Processor" (MTGP32) on GPUs which resulted in 5, 13 and 14 times higher throughput when a 240, 17 and 8 sized vector space was used respectively. The MIXMAX generator coded in VHDL and implemented on Xilinx Ultrascale+ FPGAs, requires 50% fewer total Look Up Tables (LUTs) compared to a 32-bit Mersenne Twister (MT-19337), or 75% fewer LUTs per output bit. In summary, the state-of-the art MIXMAX pseudo random number generator has been implemented on GPU and FPGA platforms and the performance benchmarked.

1 Introduction

In the context of the two general-purpose experiments at the LHC, ATLAS and CMS, the central production teams generate ~ 10 billion events per year of LHC data collection and

*e-mail: m.barbone19@imperial.ac.uk

then subject them to full detector simulation and event reconstruction. These simulated event samples are generally around three times larger in size than the total number of collected data events during the corresponding timeframe. The execution of these extensive event generation campaigns incurs a significant computational expense [1].

Looking ahead, ATLAS forecasts that by 2028, around 75% of its computational resources will be dedicated to simulating various aspects of particle generation in collisions, encompassing event generation, modelling interactions with detectors, converting signals to digital data, and finally reconstructing this data for analysis [2]. Similar projections have been put forth by other experiments operating at the LHC. This surge in computational demand is further underlined by the ongoing development and deployment of Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) for Monte Carlo (MC) simulations, as a response to the escalating data requirements presented by the HL-LHC era [3]. Recent results, show that MC simulation can be 110x faster on FPGA and 10x faster on GPU compared to multicore CPUs [4].

However, MC simulations are heavily reliant on random numbers. These random numbers are generated using Pseudo Random Number Generators (PRNGs) [1]. The preference for PRNGs over True RNGs (TRNGs) stems from practical considerations. To replicate a specific run using PRNGs, it suffices to employ the same parameters and seed value. Conversely, with TRNGs, all the generated random numbers must be meticulously stored and reintroduced into the simulation, which can be quite cumbersome and resource-intensive. Currently, there are limited RNG implementations available for these platforms and most of them do not meet either performance or quality requirements for high-energy physics. Hence, the goal of this study is to propose an RNG implementation that is suitable for MC on CPU, GPU and FPGA to simplify porting of MC simulations to these platforms.

2 Random number generators

The application of RNGs in MC simulations necessitates meeting two primary criteria: **quality** and **speed**. In modern MC simulations, which model intricate physical systems with numerous degrees of freedom, the *quality* of RNGs holds significant importance. RNGs that fail to fulfil quality requirements can lead to subtle yet *significant* systematic errors [5]. Furthermore, the quality of the RNG can impact the rate of convergence [6]. Although lower quality generators might not directly compromise the simulation results, in the most adverse cases, they can escalate the number of simulated histories required for convergence by an order of magnitude.

Knuth summarises the lesson learned by using and finding flaws in many RNGs [7]:

- The period of an RNG should exceed the length of any sequence that will be used in a single computation, but this is not enough to guarantee the absence of flaws.
- Empirical testing can reveal flaws in an RNG (if it fails a test), but no amount of empirical testing can ever confirm the flawlessness of an RNG.
- Increasing the complexity of an algorithm (especially, combining two or more methods in the same algorithm) may improve an RNG, but it can also degrade it significantly if the component methods are not statistically independent.
- It is preferable to use an RNG that has been studied extensively, whose flaws are known and understood, rather than one that appears good but whose flaws are unknown.

[1] Throughout this article, all RNGs are assumed to be PRNGs.

- There is no universal method to determine how good an RNG must be for a specific MC application. The best way to ensure that an RNG is adequate for a given application, is to use one that is designed to be adequate for all applications.

In addition, as both GPUs and FPGAs are parallel processors, RNGs that are not suitable for parallel computing cannot be utilised. Having each thread choosing a different seed **does not guarantee** that there are no correlations between streams that can cause systematic errors [8]. Hence, it is important that the RNG supports parallel streams. Given the constraints mentioned above, there are very few RNGs that are suitable for MC simulations. One is Mersenne Twister (MT) [9] which has been the default choice for more than 20 years.

In a series of publications [6, 10, 11] the authors developed new principles for the construction of high-quality RNGs. This alternative approach is based on the Ergodic theory and the authors suggested a construction of RNGs by using maximally chaotic dynamical systems (MCDS) developed by Anosov and Kolmogorov. These systems T are characterised by the *Kolmogorov-Sinai entropy* $h(T)$ *that allows to quantitatively compare, construct and select the high-quality RNGs*. The analysis performed in [10] made it clear that MT [9] and RANLUX [12] generators have low entropies of order $h(MT) \simeq 4.8$ and $h(RANLUX) \simeq 7.81$. The MIXMAX generator of dimension N developed in [6, 10, 11] has the entropy $h(MIXMAX) \propto \ln(m)N$, where $m = 2^{36}$ and $N = 8, 17, 240$. Thus the MIXMAX generators have very high entropies $h(T)$ compared with the other RNGs and therefore they have very short decorrelation time $\tau_0 = 1/2Nh(T)$ and relaxation time τ, as well as comfortably large periods q. These basic parameters are presented in the Table below:

Dimension N	Entropy $h(T)$	Decorrelation $\tau_0 = \frac{1}{h(T)2N}$	Iteration t	Relaxation $\tau = \frac{1}{h(T)\ln\frac{1}{\delta v_0}}$	Period q $\log_{10}(q)$
8	220	0,00028	1	1,54	129
17	374	0,000079	1	1,92	294
240	8679	0,00000024	1	1,17	4389

Table 1: The MIXMAX parameters. $\tau_0 < t < \tau$. The iteration time t is normalised to 1. The MIXMAX is a *genuine 61 bit generator* on Galois field GF[p], with Mersenne prime number $p = 2^{61} - 1$.

In this study, we focus our attention on MIXMAX as it achieves higher performance than RANLUX [10]. Moreover for a long time Geant4 (via CLHEP) has used MIXMAX as its default generator. Mersenne Twister is used as a baseline as it is available both on GPUs and FPGAs and it was the de-facto standard RNG for many years. Other generators, such as Philox [13], xorwow [14] and MRG32k3a [15] are also included in our GPU comparison as although they are not recommended for MC simulations [12], they are available on GPU as part of the cuRAND [16] toolkit from NVIDIA and can give a better idea of the performance of MIXMAX.

MIXMAX[N] is a nomenclature used to denote an instance of MIXMAX characterised by a state size of $N \times 64$ bits. Specifically, MIXMAX8, MIXMAX17, and MIXMAX240 possess state sizes comprising 8, 17, and 240 elements, each element consisting of 64 bits.

3 GPU implementation

The GPU implementation of MIXMAX aims to achieve high performance whilst maintaining the high-quality properties of the original CPU version. The characteristics of the original implementation are:

- Small data transfer;
- Parallel seeding;
- Multiple streams;
- Individual state.

The first two characteristics are not generally necessary to achieve high-performance as they are part of the initialisation, hence the cost is paid only once, while a slower generator will impact the performance of the MC simulation workload. However, having large data transfers as a consequence of CPU seeding then state transfer makes the initialisation slower and the API cumbersome. That is the case of the NVIDIA cuRAND implementation of MTGP32 [16], that requires several API calls to first allocate memory, initialise the state and transfer it to the GPU. Moreover, this approach does not scale as GPU parallelism is increasing at a faster pace than the speed of the interconnect, hence increasing the number of parallel instances results in more time required to initialise the state and transfer it to the GPU. This may lead to the initialisation requiring more time than the simulation itself.

The third characteristic is crucial to achieve high performance on GPUs. First, one stream per instance is necessary to leverage the entire parallelism of the GPU, as it is possible to create one instance per parallel thread. This allows each thread to produce a unique stream of random numbers that, given MIXMAX's characteristics has no correlation with other streams as the initial vectors are independent and are located on the MIXMAX trajectory at least 10^{100} steps apart even if the generator is seeded using sequential seeds [10].

The last characteristic greatly contributes to achieving high performance. The reason for this is that GPUs have limited stack size and number of registers. Data stored either on the stack or in registers can be accessed quickly, however, if too many registers are used then "register spilling" occurs. Register spilling refers to the process of moving registers from the on-chip register file to global memory when there are not enough registers available. However, this process is slow. For example, the NVIDIA Ampere microarchitecture limits the number of 32-bit registers per thread to 255 [17]. It has to be noted that these registers can be used by the entire MC simulation, hence the larger the RNG register utilisation corresponds to lower register availability for the MC simulation. So, this approach is only possible for RNGs that have a reasonably small state. In particular, it is viable for MIXMAX8 and MIXMAX17 as they require 16 and 34 registers respectively, but for MIXMAX240 it is not viable as it requires 480 registers. For comparison, Mersenne Twister requires 623 registers [9]. Given the high register usage MTGP32, the GPU optimised implementation, mitigates register spills by sharing the state among all the threads in a block and storing it in shared memory.

4 FPGA implementation

Although the MIXMAX generator is analysed in matrix form, it was shown [10] that the algorithm could be simplified and expressed as a conditional iterative summation, making efficient implementation in an FPGA possible. A fully-pipelined RTL implementation was produced in VHDL, so as to be compatible across all FPGA vendors and device families, capable of producing a new 61-bit pseudo-random number on every clock-cycle. To achieve this, the algorithm was restructured to update one register on every clock-cycle, rather than

batch-updating all 16 registers on every 16th clock-cycle. To avoid the multiplexing and conditional assignment of registers, which is known to be difficult and inefficient in the routing of the design within the FPGA, the algorithm was implemented with the 16 registers in a continuously moving loop (vs. static), with a single copy of the update logic updating a single register, corresponding to a different matrix element on each clock-cycle. Since each element is only updated once in a 16-clock cycle, there are 15 clocks in a cycle in which it maintains its previous value: we leverage this fact to precalculate and register certain quantities in a pipelined fashion prior to the final update, to improve the timing performance, as shown in the flow diagram of Figure 1.

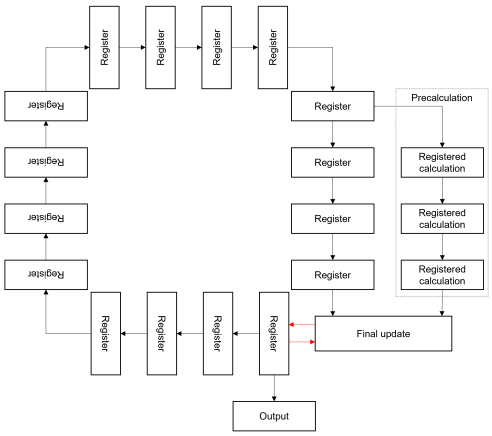


Figure 1: A fully pipelined implementation of the MIXMAX algorithm for FPGA demonstrating the 16 continuously moving registers and the pipelined precalculation of certain quantities to improve timing performance. The irreducible, non-trivial accumulator structure is highlighted in red.

The main challenge in a full duty-cycle implementation is the irreducible and non-trivial accumulator structure of the algorithm, whereby each register is of the form

```
next_value <= ( current_value + ... ) modulo ( 2^61 - 1 )
```

To avoid requiring a division in firmware, it is noted that the sum can never exceed twice the modulo value, and so is implemented with a subtraction and multiplexing; this, however, still results in a long combinatorial path, limiting the maximum clock speed.

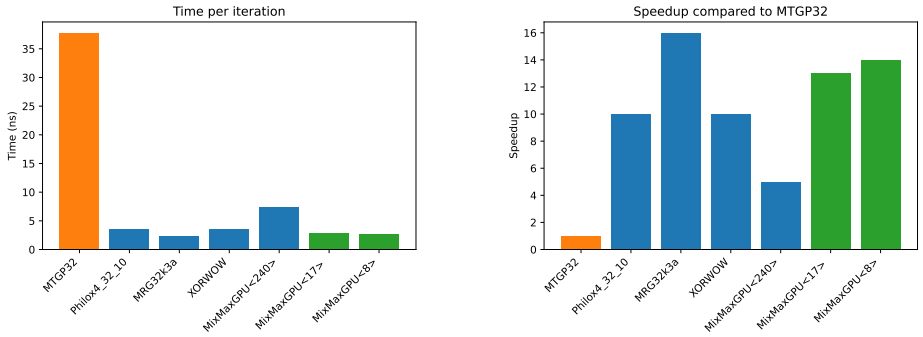
A testbench was also created for Siemens (Mentor) Modelsim[18] using the Foreign-Language Interface (FLI) to directly validate the VHDL implementation against the C++ implementation.

5 Results

5.1 GPU Results

The GPU version of MIXMAX has been implemented using nvcc 11.8, g++ 9.4.0 and executed on a NVIDIA RTX 3090Ti GPU using default compilation flags. To maximise the GPU utilisation, 32,768 parallel instances composed of 256 threads per 128 blocks are executed. We define an "iteration" as the time needed to generate 32,768 random numbers, that is the number to execute one iteration of all the parallel instances. We compared MIXMAX

against MTGP32 [19] Philox, xorwow and MRG32k3a. MTGP32 was used as the baseline as the Mersenne Twister CPU version has been used for MC simulation for many years. The other generators are included for performance comparison only as they are not recommended for MC simulation [12]. Figure 2 shows the comparison between the various generators.



(a) Time per iteration (ns)

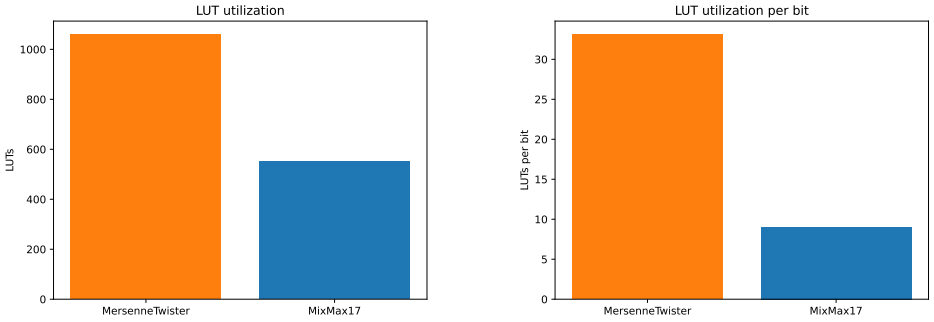
(b) Speedup compared to MTGP32

Figure 2: GPU experimental results

MRGP32 is the slowest due to the use of a shared state and the need for synchronisation between threads with an iteration time of 37 ns. The remaining generators perform significantly better as they require between 3 and 5 ns. With the only exception of MIXMAX240 that is hampered by register spills. The fastest overall is MRG32k3a which is 10% faster than MIXMAX8. Figure 2b shows the speedup compared to MTGP32, it has to be noted that all generators are an order of magnitude faster than MTGP32 with MRG32k3a, MIXMAX8 and MIXMAX17 being 16, 14 and 13 times faster respectively.

5.2 FPGA results

The implementation was synthesised and mapped into a Xilinx XCKU15P-1 Kintex Ultrascale+ FPGA using the Xilinx Vivado 2020.2 software with default settings. The implementation uses 550 Look Up Tables (LUTs) and meets timing at 300MHz. By comparison, the XCKU15P FPGA has 523,000 LUTs available, and the high-end XCVU19P Virtex Ultrascale+ FPGA has over 4M LUTs available [20]. At 330MHz, approximately 10% of endpoints fail to meet timing due to the accumulator structure. Figure 3a shows the comparison with a VHDL implementation of the 32-bit Mersenne Twister MT19937 [21] which uses 1060 LUTs, requiring nearly twice as many LUTs for a smaller output. Figure 3b shows the LUTs required per output bit. Mersenne Twister requires 33 LUTs per bit compared to the 9 required by MIXMAX. Thus, in a fair comparison MIXMAX requires 75% fewer LUTs compared to Mersenne Twister.



(a) LUT utilisation compared to MT[21].

(b) LUT utilisation per bit compared to MT [21].

Figure 3: FPGA experimental results.

6 Conclusions

The state-of-the art MIXMAX pseudo random number generator has been implemented on GPU and FPGA platforms and performance analysis conducted. The code is available on GitHub [22, 23]. The findings indicate substantial performance enhancements when compared to MTGP32 on GPUs. Specifically, when employing vector spaces of sizes 240, 17, and 8, the MIXMAX generator achieves throughput improvements by factors of 5, 13, and 14, respectively. Additionally, the VHDL-coded MIXMAX generator, implemented on Xilinx Ultrascale+ FPGAs, exhibits a notable advantage by utilizing 50% fewer total LUTs in comparison to a 32-bit Mersenne Twister (MT-19337) or 75% fewer LUTs per output bit. The exceptional performance and platform-agnostic nature of the MIXMAX RNG make it a highly promising choice for Monte Carlo simulations, as it allows for the replication of accelerated results on CPUs for ease of debugging.

References

[1] A. Valassi, E. Yazgan, J. McFayden, S. Amoroso, J. Bendavid, A. Buckley, M. Cacciari, T. Childers, V. Ciulli, R. Frederix et al., Computing and Software for Big Science **5**, 12 (2021)

[2] The ATLAS Collaboration, Tech. rep., Geneva (2020), `https://cds.cern.ch/record/2729668`

[3] J. Catmore, Proceedings of Science **390**, 009 (2021)

[4] M. Barbone, A. Howard, A. Tapper, D. Chen, M. Novak, W. Luk, Journal of Physics: Conference Series **2438**, 012023 (2023)

[5] A.M. Ferrenberg, D.P. Landau, Y.J. Wong, Physical Review Letters **69**, 3382 (1992)

[6] G.K. Savvidy, N.G. Ter-Arutyunyan-Savvidy, Journal of Computational Physics **97**, 566 (1991)

[7] D.E. Knuth, *The art of computer programming. Vol. 2: Seminumerical algorithms.*, 3rd edn. (Bonn: Addison-Wesley, 1998), ISBN 0-201-89684-2

[8] P. Hellekalek, *Don't Trust Parallel Monte Carlo!*, in *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation* (IEEE Computer Society, USA, 1998), PADS '98, pp. 82–89, ISBN 0818684577

[9] M. Matsumoto, T. Nishimura, ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**, 3 (1998)
[10] K.G. Savvidy, Computer Physics Communications **196**, 161 (2015)
[11] K. Savvidy, G. Savvidy, Chaos Solitons Fractals **91**, 33 (2016), 1510.06274
[12] F. James, L. Moneta, Computing and Software for Big Science **4**, 1 (2020)
[13] J.K. Salmon, M.A. Moraes, R.O. Dror, D.E. Shaw, Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
[14] S.L. Jr., LeaDoug, F. H., ACM SIGPLAN Notices **49**, 453 (2014)
[15] P. L'Ecuyer, Operations Research **44**, 816 (1996)
[16] *cuRAND :: CUDA Toolkit Documentation*, https://docs.nvidia.com/cuda/curand/index.html
[17] Nvidia, *NVIDIA A100 Tensor Core GPU Architecture*, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
[18] Mentor Siemens, *Modelsim*, https://eda.sw.siemens.com/en-US/ic/modelsim/
[19] M. Saito, M. Matsumoto, ACM Transactions on Mathematical Software (TOMS) **39** (2013)
[20] AMD Xilinx, *UltraScale+ FPGAs Product Selection Guide (XMP103)*, https://docs.xilinx.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide
[21] J. van Rantwijk, *Pseudo Random Number Generator based on Mersenne Twister MT19937*, https://github.com/jorisvr/vhdl_prng/blob/master/rtl/rng_mt19937.vhdl
[22] Marco Barbone, *MIXMAX CUDA source code*, https://github.com/DiamonDinoia/mixmaxCUDA
[23] Andrew W. Rose, *MIXMAX VHDL source code*, https://github.com/Cefhalic/MixMax