Random Number Generation in the Pim-Architecture

Research Project Report

Lukas Geis

Contents

1 Introduction

A *Pseudo*-RandomNumberGenerator (Rng) is the basis of all random functionality in practice. We can generate a series of (*pseudo*) random numbers to simulate *true* randomness to a certain degree. This way, we can create random models to create diverse data sets or approximate complex problems efficiently.

The Upmem-Pim[1] architecture is the first publicly-available real-world processing-in-memory (Pim) architecture. It combines traditional DRAM memory arrays with general-purpose in-order cores, called DRAM Processing Units (DPUs), integrated in the same chip.

This report describes the implementation and benchmarks of a handful of widely-used Rng's as well as several algorithms for uniformly sampling integers in a given range (Sampler) in the Upmem-Pim toolchain using the C programming language. The code can be found here[2]. Section 2 provides algorithm details of the implemented Rng's. Section 3 provides algorithm details of the implemented Sampler's. Section 4 benchmarks the implementations and analyzes the results.

2 Random Number Generators

In this section, we provide a short overview of the implemented Rng's. In total, 6 different Rng's were implemented and benchmarked. As the Upmem-Pim only has efficient support for 32-bit numbers, every Rng were implemented in its 32-bit variant. A short overview of the comparison of all Rng's except the Subtract-With-Carry Rng can be seen here[3].

2.1 XorShift

The first and most trivial Rng implemented was the XorShift (XS) Rng [4]. In spirit of its name, it only consists of three BitShifts followed by a Xor:

```
uint32_t gen_xs32(struct xs32 *rng) {
    uint32_t x = rng->x;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return rng->x = x;
}
```

While its simplicity makes it very predictable, it should also be the fastest one as it only incorporates bit operations which all can be efficiently executed on the DPU.

[1] https://www.upmem.com/

[2] https://github.com/lukasgeis/upmem-rng

[3] https://www.pcg-random.org/

2.2 MersenneTwister

The second Rng implemented is the MersenneTwister (MS) Rng [6]. The key idea is to define a series x_i through a simple recurrence relation, and then output numbers of the form x_i^T where T is an invertible $\mathbb{F}_2$-matrix called a tempering matrix.[4]. While the complete Rng is too complex to put here[5], we will provide a quick overwie over its parameters:

```
1  #define MT_N 624            // Parameter ‘N‘
2  #define MT_M 397            // Parameter ‘M‘
3  #define MT_A 0x9908b0dfU    // Constant Vector ‘A‘
4  #define MT_U 0x80000000U    // Mask for most significant ‘W − R‘ Bits
5  #define MT_L 0x7fffffffU    // Mask for least significant ‘R‘ Bits
6  
7  struct mt32 {
8      uint32_t mt[MT_N];
9      int mti;
10 };
```

As seen in line 8, the Rng needs at least a working area of 624 words, making it quite large in comparison to other Rng's. This leads to the requirement of using the compiler-flag

-DSTACK_SIZE_DEFAULT=2800

in the `dpu-upmem-dpurte-clang` compiler used to compile the DPU source code.

2.3 SubtractWithCarry

The third Rng implemented is the SubtractWithCarry (SC) Rng [5]. This Rng is defined by the following recurrence relation:

$$x(i) = (x(i-S) - x(i-R) - cy(i-1)) \mod M$$

$$\text{where} \quad cy(i) = \begin{cases} 1, & \text{if } x(i-S) - x(i-R) - cy(i-1) < 0 \\ 0, & \text{otherwise} \end{cases}$$

with $0 < S < R$ and $M = 2^W$ where W is the word size. In our case, we used $S = 8, R = 20, M = 2^{32}$.

The last three Rng's act as a form of a baseline since they are all Rng's that are either complex or very efficient if we have access to fast multiplication which is not the case in the Upmem-Pim architecture.

[4] https://en.wikipedia.org/wiki/Mersenne_Twister

[5] See the full code here: https://github.com/lukasgeis/upmem-rng/blob/main/includes/rng.c

2.4 Lehmer

The fourth RNG implemented is the Lehmer (LM) RNG [9]. The general formula is

$$X_{k+1} = \alpha \cdot X_k \mod m$$

where the modulus m is a prime number or power of a prime number, the multiplier α is an element of high multiplicative order modulo m. We used the MINSTD parameters of $m = 2^{31} - 1 = 2147483647$ and $\alpha = 7^5 = 16807$ [8].

2.5 PCG

The fifth RNG implemented was the PCG32 (PCG) RNG [7]. While we generate 32-bit numbers, the state consists of two 64-bit numbers:

```c
struct pcg32 {
    uint64_t state; // RNG state.  All values are possible.
    uint64_t inc;   // Always odd. Controls which RNG sequence is selected
};
```

While the generation of the next state is short, it not only uses 64-bit numbers but also multiplication of two 64-bit numbers which is very inefficient on the DPU:

```c
uint32_t gen_pcg32(struct pcg32 *rng) {
    uint64_t oldstate = rng->state;
    rng->state = oldstate * 6364136223846793005ULL + rng->inc;
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}
```

2.6 ChaCha

The last RNG implemented is the ChaCha (CHA) RNG [1] which is a variation of the Salsa20[6] RNG. Its internal state consists of a $4x4$-Matrix of 32-bit words and also has a buffer of the same size. The last 4 words of the states simulate a 128-bit counter and are incremented in each iteration. The main part of the algorithm consists of the following schema:

```c
#define CHACHA_ROTL32(x, n) (((x) << (n)) | ((x) >> (32 - (n))))

#define CHACHA_QUARTERROUND(x, a, b, c, d) \
    x[a] = x[a] + x[b]; x[d] ^= x[a]; x[d] = CHACHA_ROTL32(x[d], 16); \
    x[c] = x[c] + x[d]; x[b] ^= x[c]; x[b] = CHACHA_ROTL32(x[b], 12); \
    x[a] = x[a] + x[b]; x[d] ^= x[a]; x[d] = CHACHA_ROTL32(x[d],  8); \
    x[c] = x[c] + x[d]; x[b] ^= x[c]; x[b] = CHACHA_ROTL32(x[b],  7)

#define ROUNDS 20
```

[6] https://en.wikipedia.org/wiki/Salsa20

and the following round-system:

```c
for (unsigned int i = 0; i < ROUNDS; i += 2) {
    // Column Round
    CHACHA_QUARTERROUND(rng->buffer, 0, 4,  8, 12);
    CHACHA_QUARTERROUND(rng->buffer, 1, 5,  9, 13);
    CHACHA_QUARTERROUND(rng->buffer, 2, 6, 10, 14);
    CHACHA_QUARTERROUND(rng->buffer, 3, 7, 11, 15);
    // Diagonal Round
    CHACHA_QUARTERROUND(rng->buffer, 0, 5, 10, 15);
    CHACHA_QUARTERROUND(rng->buffer, 1, 6, 11, 12);
    CHACHA_QUARTERROUND(rng->buffer, 2, 7,  8, 13);
    CHACHA_QUARTERROUND(rng->buffer, 3, 4,  9, 14);
}
```

3 Uniform Integer Sampling

We now can sample random unsigned integers with 32-bits. This alone, however, is often useless as we might only need to generate a random number between 1 and 10 for example. Hence, we somehow need to convert our random number in the interval $[0, 2^{32})$ uniformly to a number in the interval $[0, s)$ for some $s \in \mathbb{N}$. Since simple solutions like the classic C-line:

```
rand() % s
```

lack uniformity, we need a better approach. For simplicity, we assume we can generate a *uniform* random number in the interval $[0, 2^L)$ for $L = 32$ using the function `rng()`. Furthermore, we assume s is given as a parameter for every following algorithm.

Finally, we will skip extensive explanations of the following Sampler's. For the first three, please refer to [2] for better explanations.

3.1 OpenBSD

The OpenBSD Sampler is an algorithm used in the *C* standard library in OpenBSD and macOS. The Go language also has adopted this algorithm with minor implementation differences. The algorithm is given by the following pseudocode:

Algorithm 1: OpenBSD

```
$t \leftarrow (2^L - s) \mod s$
$x \leftarrow$ rng()
while $x < t$ do
    $x \leftarrow$ rng()
return $x \mod s$
```

The key idea is to apply simple rejection sampling to ensure uniformity. This algorithm uses 2 division operations which are not very efficient on the DPU.

3.2 Java

The Java Sampler is an algorithm used in the Java `Random class` and aims to only use one division operation in expectation. The algorithm is given by the following pseudocode:

Algorithm 2: Java

```
1  $x \leftarrow$ rng()
2  $r \leftarrow x \mod s$
3  while $x - r > 2^L - s$ do
4      $x \leftarrow$ rng()
5      $r \leftarrow x \mod s$
6  return $r$
```

While in theory, we could have an unbounded number of division operations, in expectation, we only have $\frac{1}{p}$ for $p = 1 - \frac{2^L \mod s}{2^L} \geq \frac{1}{2}$ and hence for small s no more than one division operation in expectation.

3.3 Lemire

Lemire's Sampler aims to completely minimize the number of costly division operations by reducing the number of expected divisions to $\frac{s}{2^L}$ and capping the number of division operations at 1, meaning that for small s, we have no divison operation. The algorithm is given by the following pseudocode:

Algorithm 3: Lemire

```
1  $x \leftarrow$ rng()
2  $m \leftarrow x \times s$
3  $l \leftarrow m \mod 2^L$
4  if $l < s$ then
5      $t \leftarrow (2^L - s) \mod s$
6      while $l < t$ do
7          $x \leftarrow$ rng()
8          $m \leftarrow x \times s$
9          $l \leftarrow m \mod 2^L$
10 return $m \div 2^L$
```

Here, operations $\mod 2^L$ and $\div 2^L$ can be implemented using a simple Bit-And and Bit-Shift. While we minimize division operations, we also introduce a multiplication operation of two 64-bit numbers in line 3 which is very costly on a DPU.

3.4 Flips

The Flips Sampler uses a different approach by building up the final number bit-by-bit and resetting some of its progress if we exceed the threshold s [3]. Let `flip()` be a function that returns a single uniform random bit (can be implemented by `rng() & 1`). The algorithm is given by the following pseudocode:

Algorithm 4: Flips

```
$v \leftarrow 1$
$d \leftarrow 0$
while true do
    $d = 2 \cdot d +$ flip()
    $v = 2 \cdot v$
    if $v \geq s$ then
        if $d < s$ then
            return $d$
        $v \leftarrow v - s$
        $d \leftarrow d - s$
```

Note that we can only draw 32 bits at a time, we store the unused 31 in a buffer and use them up until we draw a new random number.

3.5 RoundReject (Variations)

The main idea of the RoundReject (RR) Sampler is to only look at the bits of the random number that are relevant to s, namely the last $\lfloor \log_2(s) \rfloor + 1$ bits - and then use rejection sampling to generate a random number in the interval $[0, s)$. The algorithm is given by the following pseudocode:

Algorithm 5: RoundReject

```
/* Number of leading zeros in the binary representation of $s$     */
$l \leftarrow s$.leading_zeros()
$m \leftarrow (1 << (32 - m)) - 1$
$x \leftarrow$ rng() & $m$
while $x \geq s$ do
    $x \leftarrow$ rng() & $m$
return $x$
```

s`.leading_zeros()` can be achieved using the compiler-builtin function `__builtin_clz(`s`)`. Since for small s, it might be wasteful to generate a new random number in each iteration and hence throw the $31 - \lfloor \log_2(s) \rfloor$ away, we create the following variations of the RR Sampler.

RoundRejectBuffered (RRB) Simply store all bits generated by `rng()` and use them in the next round instead of drawing a new random number every time. If the remaining unused bits are not enough to fill up the required bits, draw a new number an concatenate the required bits.

RoundRejectFlips (RRF) This is a combination of the RR and the Flips Sampler. Since the longest part of Flips for bigger s is building up the first batch of required bits, we skip that step by substituting the „first round“ of Flips by one round of RR and then proceed with the original Flips algorithm.

4 Benchmarks

In this section, we discuss the most important benchmarks for these implementations. The benchmark-code can also be found in the same GitHub-repository. Note that we ran each benchmark only on 1 DPU at a time, since every experiment indicated that there was no time loss if multiple DPUs were running at the same time (except for the initial load-overhead which was not part of the benchmark hence excluded). Thus, every result can be achieved on all DPUs at the same time (which in our case are about 2500). Every DPU has a capacity to have up to 22 tasklets run parallel at the same time - hence every result in this section will be presented as a function of number of tasklets.

4.1 Rng Benchmarks

In this section, we will shortly review the execution time of each Rng on the DPU as well as on the CPU in comparison. The results on the DPU can be seen in Figure 1. Clearly, XS yields the best results, needing less than 3 seconds in total for $N = 10^7$ generated random numbers. There is a slight increase in execution time starting from 11 tasklets up to the end, but it stays under 4 seconds nonetheless. The next best Rng is SC, followed by MT, then CHA, LM, and finally PCG, which needs more than 40 seconds at 1 tasklet and more than 80 seconds at 22 tasklets. This translates to our hypothesis, that Rng's like XS, SC, and MT are better on the DPU because they do not require costly multiplications or 64-bit numbers. Every Rng seems to achieve a constant time regardless of number of tasklets up to 11 tasklets before taking on linear growth in exeuction time. This trend however is better observed in the right graphic which depicts the total number of bytes generated per second by all tasklets combined for each Rng. The number is given by the conversion

$$\underbrace{4}_{\text{Bytes in }\texttt{uint32\_t}} \cdot \underbrace{10^7}_{N} \cdot \underbrace{n_T}_{\text{Number of Tasklets}} \quad / \quad \underbrace{t_T}_{\text{Execution time in seconds}}$$

Here, we can clearly see that for more than 11 tasklets, there is neither an additional benefit nor an additional cost for using more tasklets. Using XorShift, we can achieve up to roughly 200MB generated per second which translates to about 500GB per second when using all 2500 DPU modules.

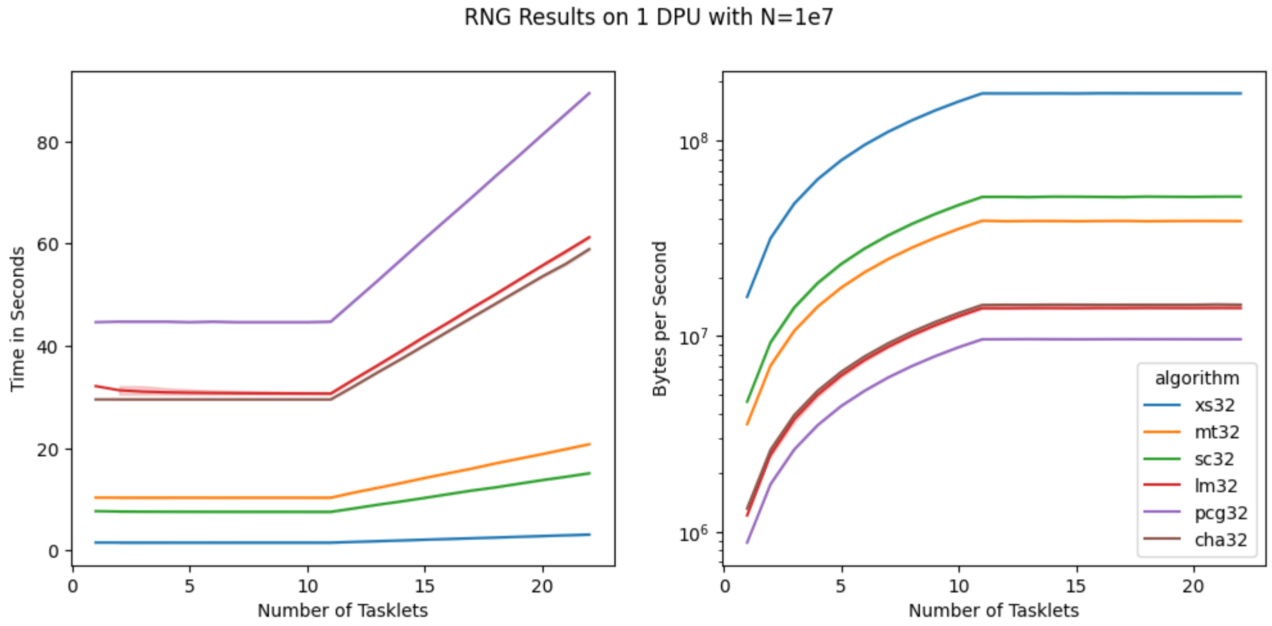

Figure 1: Measured time and generated bytes per second taken by one DPU for generating $N = 10^7$ numbers with the respective RNG's as a function of number of tasklets.

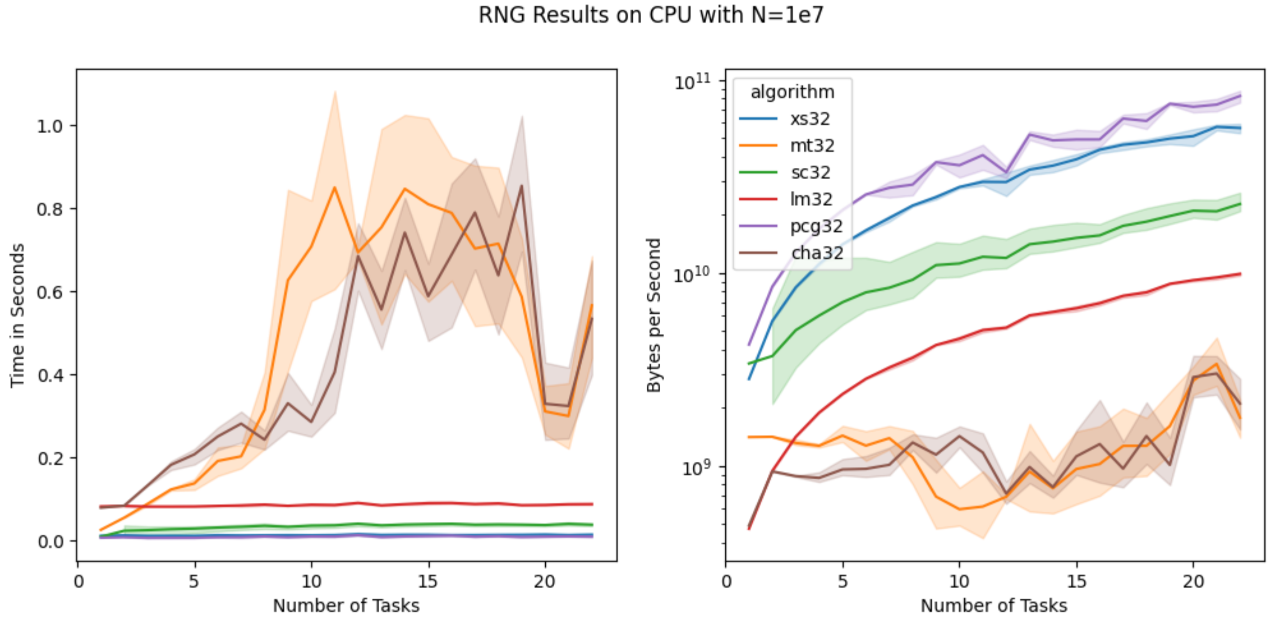

Figure 2: Measured time and generated bytes per second taken by CPU for generating $N = 10^7$ numbers with the respective RNG's as a function of number of tasks.

In comparison, we can see the results on the CPU in Figure 2. We will not go into too much detail about these results as they are only meant for a comparison with the DPU results - which are the main focus of this project. We can clearly see that PCG and XorShift are the fastest Rng's which is plausible considering the complexity of their generation code. While MT was the third fastest Rng on the DPU, it is the slowest among the Rng's on the CPU. SC is still better than LM, MT and CHA, only beaten by the previously worse PCG.

4.2 Sampler Benchmarks

In this section, we will shortly review the execution time (and bytes generated per second) of each Sampler on the DPU as well as on the CPu in comparison. We generated a random integer in the interval $[0, s)$ for every $2 \leq s \leq 10^9$ for every Sampler and added up the total computation time. We also did this experiment one time using the XorShift-Rng - the fastest Rng on the DPU - and the PCG-Rng - the slowest Rng on the DPU - to account for the impact of the running time of generating a random number.

We can see the results on the DPU using the XS-Rng in Figure 3. The fastest Sampler is RR, closely followed by RRF and RRB, then by Java, OpenBSD, Lemire, and finally Flips. This was to be expected since RR and its variants only use cheap and efficient operations like masking, addition and comparisons - with the exception of the one-time-use of `__builtin_clz()`. Java and OpenBSD are also quite fast since they have a bounded number (or expected bounded) of costly division operations but no further inefficient operations. While Lemire minimizes the number of divisions, it has a very costly multiplication operation of two 64-bit integers which can happen several times. Flips being last place also was to be expected since for bigger s, we will have to take $\log_2(s)$ steps to reach the first point where we can accept or reject a generated integer. On the right, we can see that RR achieves a throughput of roughly 70MB generated per second while Flips only achieves a throughput of about 6MB generated per second. We can also see the same phenomenon we saw earlier: there is neither an additional benefit nor an additional cost for using more tasklets.

In comparion, if we use the PCG-Rng instead (Figure 4), the results now differ at the top. While Flips, and then Lemire, is still the slowest Sampler, the fastest one is now RRF, followed by Java, then OpenBSD and finally RR and RRF. This makes sense since RRF minimizes the number of uses of rng() while RR enables rejecting many unused bits for smaller s in comparison. Here we achieve a maximal throughput of roughly 8MB generated per second using RRF and about 4MB generated per second using Flips. Hence, when using a worse Rng, the difference between the best and the worst Sampler narrows.

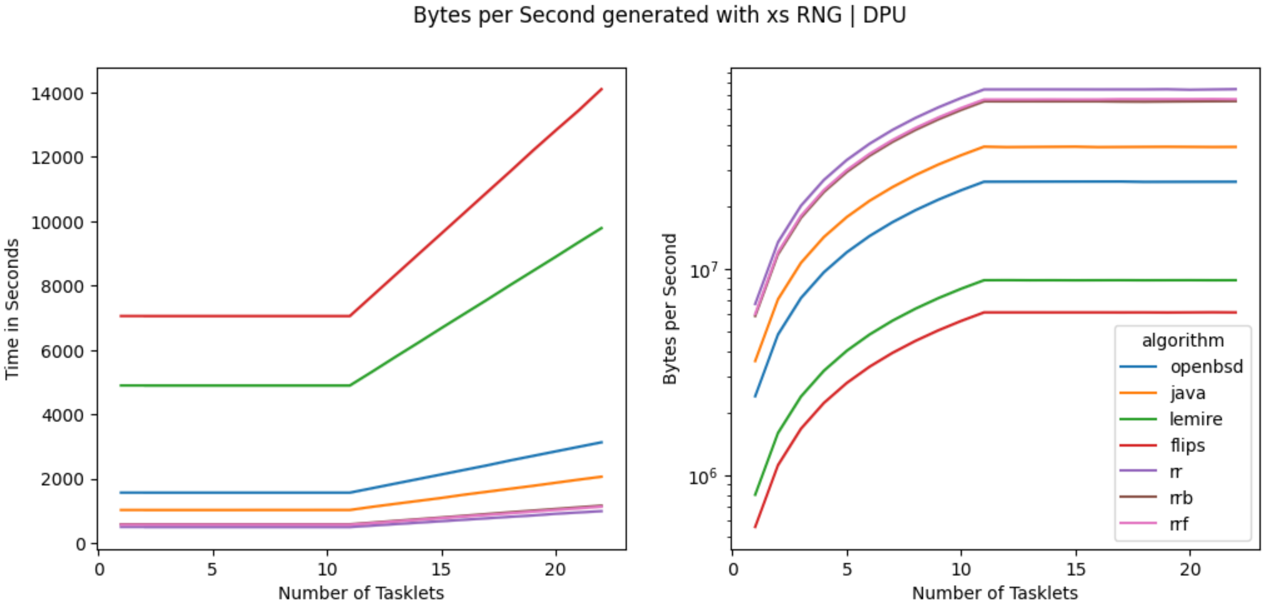

Figure 3: Measured time and generated bytes per second taken by one DPU for generating numbers in range $[0, s)$ for all $s = 2..10^9$ with the respective SAMPLER's as a function of number of tasklets. RNG used was XS.

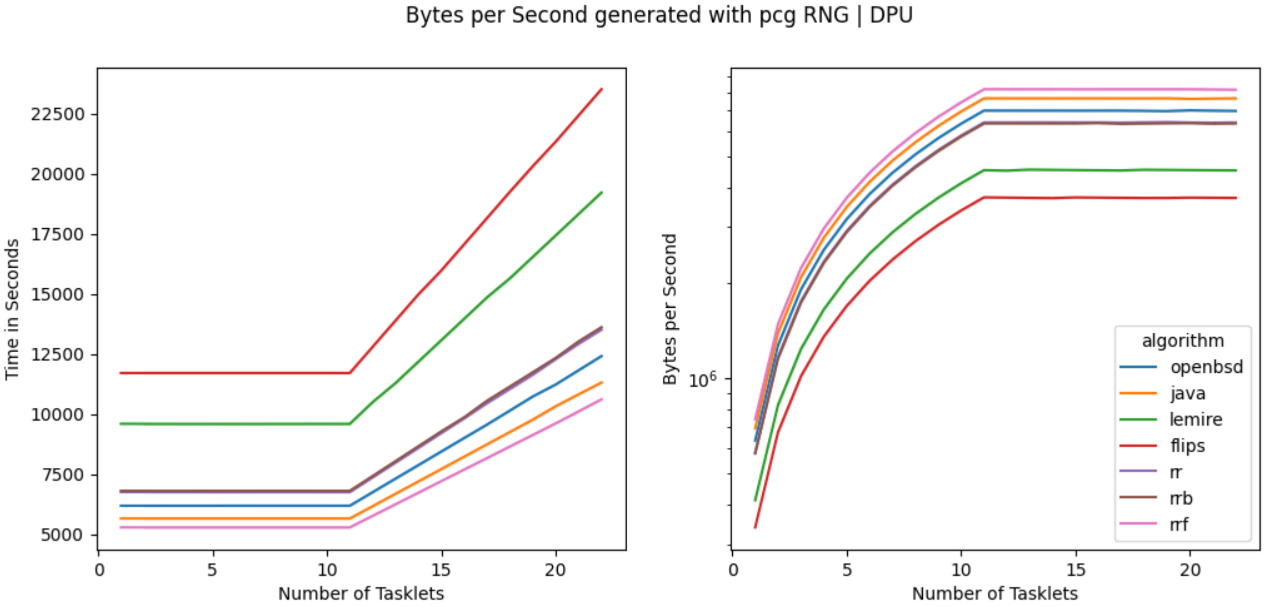

Figure 4: Measured time and generated bytes per second taken by one DPU for generating numbers in range $[0, s)$ for all $s = 2..10^9$ with the respective SAMPLER's as a function of number of tasklets. RNG used was PCG.

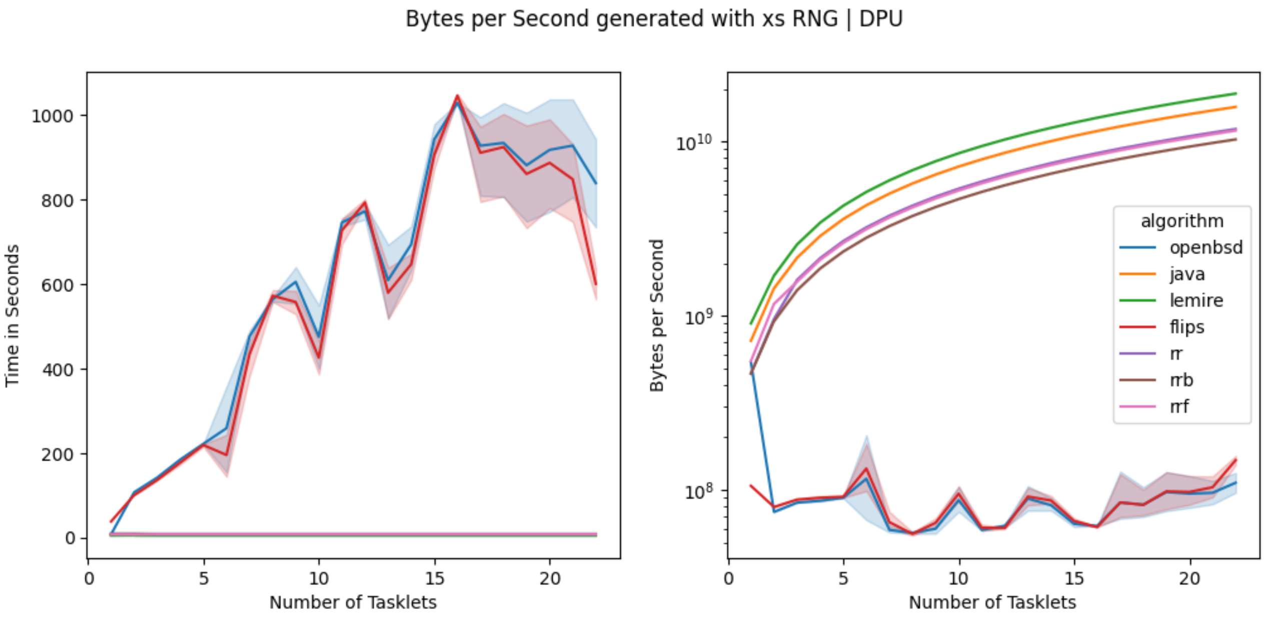

Figure 5: Measured time and generated bytes per second taken by CPU for generating numbers in range $[0, s)$ for all $s = 2..10^9$ with the respective SAMPLER's as a function of number of tasks. RNG used was XS.

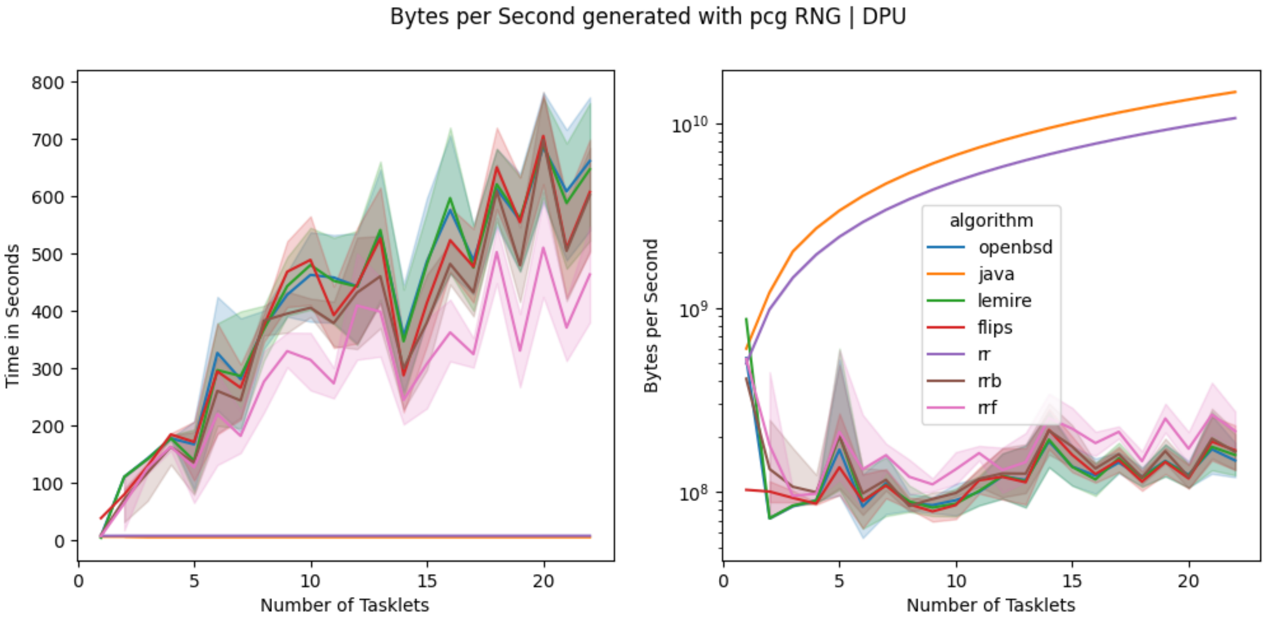

Figure 6: Measured time and generated bytes per second taken by CPU for generating numbers in range $[0, s)$ for all $s = 2..10^9$ with the respective SAMPLER's as a function of number of tasks. RNG used was PCG.

If we look at the results of the same experiment on the CPU in comparison (Figure 5 and Figure 6), we can clearly see that almost all Sampler - with the exception of Flips and OpenBSD - achieve an execution time of under 10 seconds when using the XS-Rng. Here, Lemire performs the best, followed by Java, RR, RRF, and RRB. When using the PCG-Rng, only Java and RR perform quite well while the rest is way worse (and shaky). This might be due to some external issues (like Power-Management, etc.), but it was not studied and should be ignore instead as it is of no significant importance in this research project.

References

[1] Daniel J. Bernstein. *ChaCha, a variant of Salsa20.* Tech. rep. The University of Illinois at Chicago, 2008.

[2] Daniel Lemire. "Fast Random Integer Generation in an Interval". In: *ACM Transactions on Modeling and Computer Simulation* 29.1 (Jan. 2019), pp. 1–12. issn: 1558-1195. doi: 10.1145/3230636. url: http://dx.doi.org/10.1145/3230636.

[3] Jérémie Lumbroso. *Optimal Discrete Uniform Generation from Coin Flips, and Applications.* 2013. arXiv: 1304.1916 [cs.DS].

[4] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6. doi: 10.18637/jss.v008.i14. url: https://www.jstatsoft.org/index.php/jss/article/view/v008i14.

[5] George Marsaglia and Arif Zaman. "A New Class of Random Number Generators". In: *The Annals of Applied Probability* 1.3 (1991), pp. 462–480. doi: 10.1214/aoap/1177005878. url: https://doi.org/10.1214/aoap/1177005878.

[6] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. issn: 1049-3301. doi: 10.1145/272991.272995. url: https://doi.org/10.1145/272991.272995.

[7] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.* Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[8] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find". In: *Commun. ACM* 31.10 (Oct. 1988), pp. 1192–1201. issn: 0001-0782. doi: 10.1145/63039.63042. url: https://doi.org/10.1145/63039.63042.

[9] W. H. Payne, J. R. Rabung, and T. P. Bogyo. "Coding the Lehmer Pseudo-Random Number Generator". In: *Commun. ACM* 12.2 (Feb. 1969), pp. 85–86. issn: 0001-0782. doi: 10.1145/362848.362860. url: https://doi.org/10.1145/362848.362860.