Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 571–578

ISBN 978-83-60810-51-4

Parallel GPU-accelerated Recursion-based Generators of Pseudorandom Numbers

Przemysław Stpiczyński
Maria Curie-Skłodowska University, Lublin, Poland
Institute of Theoretical and Applied Informatics of
the Polish Academy of Sciences, Gliwice, Poland
Email: przem@hektor.umcs.lublin.pl

Dominik Szałkowski, Joanna Potiopa
Institute of Mathematics, Maria Curie-Skłodowska University,
Pl. M. Curie-Skłodowskiej 1, Lublin, Poland
Email: dominisz@umcs.lublin.pl,
joannap@hektor.umcs.lublin.pl

***Abstract*—The aim of the paper is to show how to design fast parallel algorithms for linear congruential and lagged Fibonacci pseudorandom numbers generators. The new algorithms employ the *divide-and-conquer* approach for solving linear recurrence systems and can be easily implemented on GPU-accelerated hybrid systems using CUDA or OpenCL. Numerical experiments performed on a computer system with modern Fermi GPU show that they achieve good speedup in comparison to the standard CPU-based sequential algorithms.**

I. Introduction

PSEUDORANDOM numbers are very important in practice and pseudorandom number generators are often central parts of scientific applications such as simulations of physical systems using Monte Carlo methods. There are a lot of such generators with different properties [5]. Recursion-based generators such as linear congruential generators (LCG) and lagged Fibonacci (LFG) generators have good statistical properties and they are commonly used [2], [6] (especially LFG, which has better statistical properties than LCG). The problem of designing efficient parallel pseudorandom numbers is very important from the practical point of view [1], [4], [22], [23].

Graphical processing units (GPUs [18]) have recently been widely used for scientific computing due to their large number of parallel processors which can be exploited using the Compute Unified Device Architecture (CUDA) programming language [17]. In 2010 NVIDIA introduced a new GPU (CUDA) architecture with an internal name of Fermi [21]. Now programmers should consider the use of hybrid systems which comprise multicore CPUs and powerful manycore GPUs. Thus it is a good idea to develop algorithms for hybrid (heterogeneous) computer architectures where large parallelizable tasks are scheduled for execution on GPUs, while small non-parallelizable tasks should be run on CPUs [30].

The aim of the paper is to show how to design fast parallel and scalable LCG and LFG generators using the *divide-and-conquer* approach [27] for solving recurrence systems instead of using standard *cycle division* and *parametrization* techniques which produce pseudorandom number streams on different processes and such streams should be tested for possible correlations [25]. The new method can be easily implemented on modern computing architectures including GPU-accelerated systems [28], [29]. It should be noticed that the approach can be adopted for solving other recursion-based computational problems on modern hybrid systems.

II. Related works

Parallel pseudorandom generators have been studied by many authors [3], [4], [10], [14], [15], [23]. Parallel versions of LCG and LFG generators have been introduced in the `SPRNG` Library [11], [12]. Such libraries have been developed under the assumption that a parallel generator should produce a totally reproduced stream of pseudorandom numbers without any interprocessor communication [8] using cycle division or parameterizing techniques [9], [13]. A parallel generator should also be portable between serial and parallel platforms. and such generators should be tested for possible correlations and 'high quality' properties [25].

Our approach for developing parallel pseudorandom generators is quite different. We parallelize recurrence relations for LCG and LFG, thus statistical properties of our parallel generators are exactly the same as for corresponding sequential ones, thus there is no need to perform special statistical tests.

It should be noticed that the CURAND library [19] provides facilities that can be used for the simple and efficient generation of pseudorandom numbers for GPU computations using some other popular generators such as `XORWOW`, `MRG32K3A` and `MTGP32`. Unfortunately, LCG and LFG generators are not present in CURAND.

III. Recursion-based pseudorandom numbers generators

Let us consider the following two well-known sequential pseudorandom numbers generators:

1) **Linear congruential generator (LCG)**:

$$x_{i+1} \equiv (ax_i + c)(\mod m), \tag{1}$$

where x_i is a sequence of pseudorandom values, $m > 0$ is the *modulus*, a, $0 < a < m$ is the *multiplier*, c, $0 \leq c < m$ is the *increment*, x_0, $0 \leq x_0 < m$ is the *seed* or *start value*. For example:

a) $m = 2^{32}$, $a = 1664525$, $c = 1013904223$ [24],
b) $m = 2^{64}$, $a = 6364136223846793005$, $c = 1442695040888963407$, [6],
c) $m = 2^{31} - 1$, $a = 7^5$, $c = 0$ [7].

2) **Lagged Fibonacci generator (LFG)**:

$$x_i \equiv (x_{i-p_1} \diamond x_{i-p_2})(\mod m), \tag{2}$$

where $0 < p_1 < p_2$, $\diamond \in \{+, -, *, \texttt{xor}\}$. For example [2], it can be $\diamond \equiv +$ and

a) $p_1 = 7$, $p_2 = 10$,
b) $p_1 = 5$, $p_2 = 17$, which was the standard parallel generator in the Thinking Machines Connection Machine Scientific Subroutine Library.

Usually, $m = 2^M$, and $M = 32$ or 64. It allows the modulus operation to be computed by merely truncating all but the rightmost 32 or 64 bits, respectively. Thus, when we use `unsigned int` or `unsigned long` data types, we can neglect "$(\mod m)$". Note that the integers x_k are between 0 and $m-1$. Thay can be converted to real values $r_k \in [0, 1)$ by $r_k = x_k/m$ [1].

IV. New parallel algorithms

Equations (1) and (2) can be considered as special cases of the following linear recurrence system

$$x_k = \begin{cases} 0 & \text{for } k < 0 \\ c_k + \sum_{j=1}^{p} a_j x_{k-j} & \text{for } 0 \le k \le n-1, \end{cases} \tag{3}$$

where we have to compute n number x_k, $k = 0, \ldots, n-1$, and all coefficients c_k, a_k are given. In [28], [29] we showed that the problem (3) can be efficiently solved on heterogeneous computer architectures based on CPU and CUDA-enabled GPU processors using the *divide and conquer* method [27]. Now we will show how to use this approach for finding sequences of preudorandom numbers. Let

$$\mathbb{Z}_m = \{0, 1, \ldots, m-1\}.$$

Now we will consider parallel versions of LCG and LFG.

A. Parallel LCG

Let us consider the following (simple, sequential) algorithm for finding (1):

$$\begin{cases} x_0 = d \\ x_{i+1} = ax_i + c, \quad i = 0, \ldots, n-2, \end{cases} \tag{4}$$

The recurrence relation (4) can be rewritten as the following system of linear equations:

$$\begin{bmatrix} 1 & & & \\ -a & 1 & & \\ & \ddots & \ddots & \\ & & -a & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} d \\ c \\ \vdots \\ c \end{bmatrix}, \tag{5}$$

Let us assume that $n = rs$, where $r, s > 1$. Then (5) can be rewritten in the following block form:

$$\begin{bmatrix} A & & & \\ B & A & & \\ & \ddots & \ddots & \\ & & B & A \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f} \\ \vdots \\ \mathbf{f} \end{bmatrix} \tag{6}$$

where $\mathbf{x}_i = (x_{is}, \ldots, x_{(i+1)s-1})^T \in \mathbb{Z}_m^s$ and $\mathbf{f}_0 = (d, c, \ldots, c)^T \in \mathbb{Z}_m^s$, and $\mathbf{f} = (c, \ldots, c)^T \in \mathbb{Z}_m^s$, and the matrices A and B are defined as follows

$$A = \begin{bmatrix} 1 & & & \\ -a & 1 & & \\ & \ddots & \ddots & \\ & & -a & 1 \end{bmatrix} \in \mathbb{Z}_m^{s\times s},$$

$$B = \begin{bmatrix} 0 & \cdots & 0 & -a \\ \vdots & \ddots & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} \in \mathbb{Z}_m^{s\times s}.$$

From (6) we have

$$\begin{cases} A\mathbf{x}_0 = \mathbf{f}_0 \\ B\mathbf{x}_{i-1} + A\mathbf{x}_i = \mathbf{f}, \quad i = 1, \ldots, r-1, \end{cases}$$

and then

$$\begin{cases} \mathbf{x}_0 = A^{-1}\mathbf{f} \\ \mathbf{x}_i = A^{-1}\mathbf{f} - A^{-1}B\mathbf{x}_{i-1}, \quad i = 1, \ldots, r-1, \end{cases}$$

where A^{-1} denotes the matrix such that $A^{-1}A = I$. Note that

$$-A^{-1}B\mathbf{x}_{i-1} = A^{-1}\begin{bmatrix} ax_{is-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} = x_{is-1}A^{-1}\begin{bmatrix} a \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

When we set $\mathbf{t} = A^{-1}\mathbf{f}$ and $\mathbf{y} = A^{-1}(a\mathbf{e}_0)$, where $\mathbf{e}_0 = (1, 0, \ldots, 0)^T \in \mathbb{Z}_m^s$, then we get

$$\begin{cases} \mathbf{x}_0 = A^{-1}\mathbf{f}_0 \\ \mathbf{x}_i = \mathbf{t} + x_{is-1}\mathbf{y}, \quad i = 1, \ldots, r-1. \end{cases} \tag{7}$$

The equation (7) has a lot of potential parallelism. The algorithm comprises the following steps:

1) Find $\mathbf{x}_0$, $\mathbf{y}$, $\mathbf{t}$ (using three separate tasks).
2) Find the last entry of each vector $\mathbf{x}_i$, $i = 1, \ldots, r-1$ (sequentially, using (7)).
3) Find $s-1$ entries of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{r-1}$ (in parallel, using (7)).

To implement the algorithm efficiently, let us assume that we have to find the following matrix

$$Z = \begin{bmatrix} z_{00} & \cdots & z_{0,r-1} \\ \vdots & \ddots & \vdots \\ z_{s-1,0} & \cdots & z_{s-1,r-1} \end{bmatrix} \in \mathbb{Z}_m^{s\times r}$$

such that

$$\begin{bmatrix} z_{0i} \\ \vdots \\ z_{s-1,i} \end{bmatrix} = \mathbf{x}_i, \; i = 0, \ldots, r-1.$$

The steps of the parallel algorithm can be described as follows:

Step 1:

$$\begin{cases} y_0 = a \\ y_{i+1} = a y_i, \quad i = 0, \ldots, s-2 \end{cases}$$

$$\begin{cases} z_{00} = d \\ z_{i+1,0} = a z_{i,0} + c, \quad i = 0, \ldots, s-2 \end{cases}$$

$$\begin{cases} t_0 = c \\ t_{i+1} = a t_i + c, \quad i = 0, \ldots, s-2 \end{cases}$$

Step 2:

$$z_{s-1,i} = z_{s-1,i-1} y_{s-1} + t_{s-1},\ i = 1, \ldots, r-1$$

Step 3:

$$\begin{bmatrix} z_{0i} \\ \vdots \\ z_{s-2,i} \end{bmatrix} = \begin{bmatrix} t_0 \\ \vdots \\ t_{s-2} \end{bmatrix} + z_{s-1,i-1} \begin{bmatrix} y_0 \\ \vdots \\ y_{s-2} \end{bmatrix}, \quad i = 1, \ldots, r-1.$$

B. Parallel LFG

Let us consider the following simple algorithm for finding a pseudorandom sequence using (2):

$$\begin{cases} x_i = d_i & i = 0, \ldots, p_2 - 1 \\ x_i = x_{i-p_1} + x_{i-p_2}, & i = p_2, \ldots, n-1. \end{cases}$$

Let $n = rs$, $r, s > p_2$. To find a sequence $x_0, \ldots, x_{n-1}$, we have to solve the following system of linear equations

$$\begin{bmatrix} A_0 & & & \\ B & A & & \\ & \ddots & \ddots & \\ & & B & A \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{8}$$

where the matrices A_0, A and B are of the following forms (see also Figure 1):

$$A_0 = \begin{bmatrix} 1 & & & & & & \\ \vdots & \ddots & & & & & \\ 0 & \ldots & 1 & & & & \\ \vdots & \ddots & & \ddots & & & \\ -1 & & -1 & & 1 & & \\ & \ddots & & \ddots & & \ddots & \\ & & -1 & & -1 & & 1 \end{bmatrix} \in \mathbb{Z}_m^{s \times s},$$

$$A = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ -1 & & 1 & & & & \\ & \ddots & & \ddots & & & \\ -1 & & -1 & & 1 & & \\ & \ddots & & \ddots & & \ddots & \\ & & -1 & & -1 & & 1 \end{bmatrix} \in \mathbb{Z}_m^{s \times s},$$

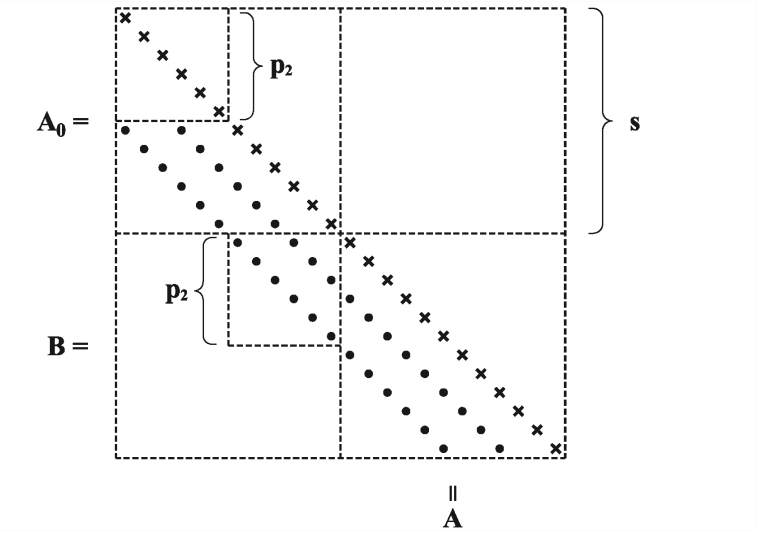

Fig. 1. The matrix of the system (8) for $n = 24$, $s = 12$, $r = 2$, $p_1 = 3$, $p_2 = 6$

$$B = \begin{bmatrix} 0 & & -1 & & -1 & 0 & \ldots & 0 \\ & \ddots & & \ddots & & \ddots & \ddots & \vdots \\ & & \ddots & & \ddots & & \ddots & 0 \\ & & & \ddots & & \ddots & & -1 \\ & & & & \ddots & & \ddots & \\ & & & & & \ddots & & -1 \\ & & & & & & \ddots & \\ & & & & & & & 0 \end{bmatrix} \in \mathbb{Z}_m^{s \times s}.$$

The vectors are defined as $\mathbf{f} = (d_0, \ldots, d_{p_2-1}, 0, \ldots, 0)^T \in \mathbb{Z}_m^s$ and $\mathbf{x}_i = (x_{is}, \ldots, x_{(i+1)s-1})^T \in \mathbb{Z}_m^s$. Thus we have

$$\begin{cases} A_0 \mathbf{x}_0 = \mathbf{f} \\ B \mathbf{x}_{i-1} + A \mathbf{x}_i = 0, \quad i = 1, \ldots, r-1. \end{cases} \tag{9}$$

Let

$$\mathbf{e}_k = (\underbrace{0, \ldots, 0}_{k}, 1, 0, \ldots, 0)^T \in \mathbb{Z}_m^s.$$

Then

$$\begin{aligned} \mathbf{x}_i &= -A^{-1} B \mathbf{x}_{i-1} = \\ &= A^{-1} \left(\sum_{k=0}^{p_2-1} \mathbf{e}_k \mathbf{e}_{s-p_2+k}^T + \sum_{k=0}^{p_1-1} \mathbf{e}_k \mathbf{e}_{s-p_1+k}^T \right) \mathbf{x}_{i-1} = \\ &= \sum_{k=0}^{p_2-1} A^{-1}(x_{is-p_2+k} \mathbf{e}_k) + \sum_{k=0}^{p_1-1} A^{-1}(x_{is-p_1+k} \mathbf{e}_k). \end{aligned}$$

Let $\mathbf{y}_k$ satisfies the system of linear equations $A \mathbf{y}_k = \mathbf{e}_k$, $k = 0, \ldots, p_2 - 1$, and $\mathbf{y} \equiv \mathbf{y}_0 = (1, y_1, \ldots, y_{s-1})^T$. It is

easy to verify that

$$\mathbf{y}_k = (\underbrace{0,\ldots,0}_{k},1,y_1,\ldots,y_{s-1-k})^T,$$

Thus it is sufficient to find only the vector $\mathbf{y}$. Finally we get

$$\begin{cases} \mathbf{x}_0 = A_0^{-1}\mathbf{f} \\ \mathbf{x}_i = \sum_{k=0}^{p_2-1} x_{is-p_2+k}\mathbf{y}_k + \sum_{k=0}^{p_1-1} x_{is-p_1+k}\mathbf{y}_k, \\ \qquad\qquad i=1,\ldots,r-1. \end{cases} \tag{10}$$

Note that (10) is the generalization of (7). Thus one can develop a similar parallel *divide-and-conquer* algorithm. During the first step we have to find the vectors $\mathbf{x}_0$ and $\mathbf{y}_0$, then (Step 2) using (10) we find p_2 last entries of $\mathbf{x}_1,\ldots,\mathbf{x}_{r-1}$. Finally (Step 3) we use (10) to find $s-p_2$ first entries of these vectors in parallel. Note that Step 2 requires interprocessor communication.

V. GPU-ACCELERATED IMPLEMENTATIONS

The detailed description of NVIDIA CUDA and Fermi architectures can be found in [21] and [20]. A *computing device* (usually GPU) comprises a number of streaming multiprocessors (SM). Each SM consists of 32 (8 for older CUDA devices) scalar cores (streaming processors, SP). SMs are responsible for executing blocks of threads. Threads within a block are grouped into so-called *warps*, each consisting of 32 threads managed and executed together.

A device has its own memory system including the *global memory* (large but low latency), *constant* and *texture* read-only memories providing reduction of memory latency. Each SM has also a 48 kB (16 kB for older CUDA devices) of fast *shared memory* that can be used for sharing data among threads within a block. The global memory access can be improved by coalesced access of all threads in a half-warp. Threads must access either 4-byte words in one 64-byte memory transaction, or 8-byte words in one 128-byte memory transaction. All 16 words must lie in the same memory segment [20].

Fermi (CUDA) programs consist of a number of C functions called *kernels* that are to be executed on devices as threads. Kernels are called from programs executed on CPUs. Host programs are also responsible for allocation of variables in the device global memory. There are also some CUDA API-functions used to copy data between host and device global memories.

In our implementation large parallelizable tasks are scheduled for execution on GPUs, while small non-parallelizable tasks should be run on CPUs. **Step 1** is executed on CPU and then data are copied to the global memory of GPU. Next, **Step 2** and **Step 3** are executed on GPU (see Figures 2 and 3 for details) using one-dimensional blocks of threads. Each thread is responsible for computing one row of the matrix Z. The implementation of LFG is rather similar. The first two steps are also computed on CPU, data are copied to the global memory of GPU and then the appropriate kernel is executed on GPU

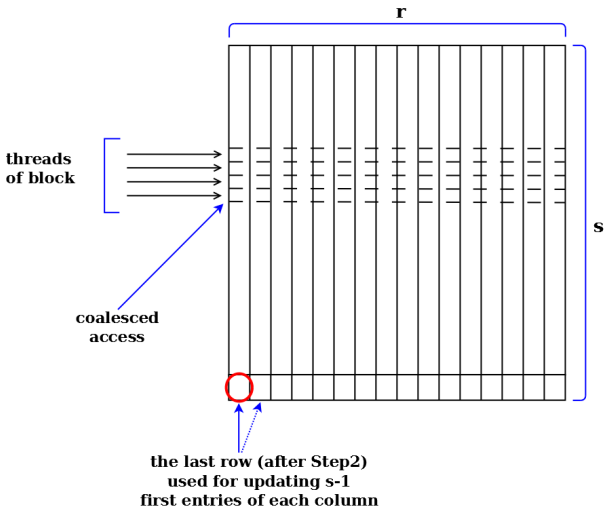

Fig. 2. Parallel LCG – Step 3 executed on GPU

(see Figure 4). The kernel is more sophisticated. To reduce the number of global memory references, the vector $\mathbf{y}$ is "manually" cached in the shared memory. It should be noticed that our implementation is fully portable, thus the kernels will be run efficiently on the new NVIDIA GPU architecture with an internal name of Kepler. The algorithms can also be implemented using OpenCL [16].

VI. RESULTS OF EXPERIMENTS

The considered algorithms have been tested on a computer with Intel Xeon X5650 (2.67 GHz, 48GB RAM) and NVIDIA Tesla M2050 (448 cores, 3GB GDDR5 RAM with ECC off), running under Linux with `gcc` and NVIDIA `nvcc` compilers and CURAND Library ver. 4.0 provided by the vendor. Our algorithms are faster than CURAND generators (Figure 9), however they implement different generators, so they cannot be compared directly. The results of experiments are presented in Figures 5 and 6 for LCG, and in Figures 7 and 8 for LFG. We can conclude the following:

- For both parallel generators, GPU achieves the best performance for `blockDim.x==256`.
- The best performance results are achieved for when r and s are $O(\sqrt{n})$, and $s \approx 10r$ (Figures 5 and 7).
- Parallel LCG is up to $80\times$ faster than simple sequential LCG.
- The performance of Parallel LFG depends on the sequence length (the value of n), and the values of p_1 and p_2.
- For smaller values of p_1 and p_2, the performance of Parallel LFG is reasonable (up to $30\times$). Unfortunately, when $p_2 > 60$, then the performance of Parallel LFG is similar to the performance of LFG executed on CPU. It is obvious, because the parallel algorithm requires much more computations for bigger values p_1 and p_2 (see Eq. (10)).

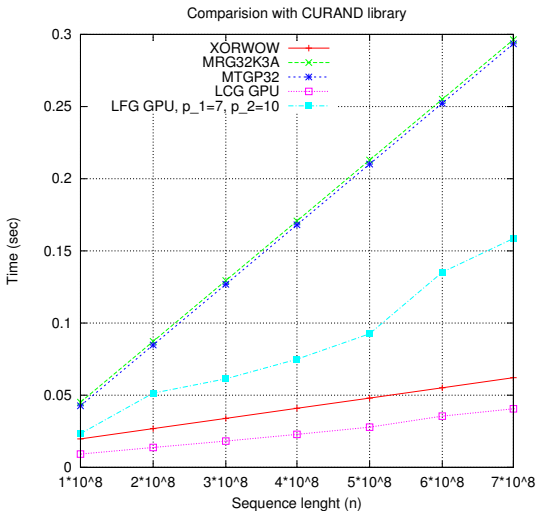

Fig. 9. CURAND and the new algorithms

VII. Conclusions and Future Work

We have showed how to design and implement fast Parallel LCG and Parallel LFG pseudorandom number generators using the *divide-and-conquer* approach for solving linear recurrences on modern hybrid systems. The methods can be easily implemented on multiple GPUs and clusters of hybrid CPU+GPU nodes using the method introduced in [26]. We are planning to reimplement out routines using CURAND-like implementation style and make them freely available for the community.

Acknowledgements

The authors would like to express their gratitude to the Anonymous Referees for their helpful comments on the paper.

The work has been prepared using the supercomputer resources provided by the Institute of Mathematics of the Maria Curie-Skłodowska University in Lublin.

References

[1] R. H. Bisseling, *Parallel scientific computation. A structured approach using BSP and MPI*. Oxford: Oxford University Press, 2004.

[2] R. P. Brent, "Uniform random number generators for supercomputers," in *Proc. Fifth Australian Supercomputer Conference*, 1992, pp. 95–104.

[3] S. A. Cuccaro, M. Mascagni, and D. V. Pryor, "Techniques for testing the quality of parallel pseudorandom number generators," in *PPSC*, 1995, pp. 279–284.

[4] I. Deák, "Uniform random number generators for parallel computers," *Parallel Computing*, vol. 15, no. 1-3, pp. 155–164, 1990.

[5] D. E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[6] ——, *MMIXware, A RISC Computer for the Third Millennium*, ser. Lecture Notes in Computer Science. Springer, 1999, vol. 1750.

[7] P. A. W. Lewis, A. S. Goodman, and J. M. Miller, "A pseudo-random number generator for the system/360," *IBM Systems Journal*, vol. 8, no. 2, pp. 360–146, 1969.

[8] M. Mascagni, "Parallel linear congruential generators with prime moduli," *Parallel Computing*, vol. 24, no. 5-6, pp. 923–936, 1998.

[9] M. Mascagni and H. Chi, "Parallel linear congruential generators with Sophie-Germain moduli," *Parallel Computing*, vol. 30, no. 11, pp. 1217–1231, 2004.

[10] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson, "Recent developments in parallel pseudorandom number generation," in *PPSC*, 1993, pp. 524–529.

[11] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: a scalable library for pseudorandom number generation," *ACM Trans. Math. Softw.*, vol. 26, no. 3, pp. 436–461, 2000.

[12] ——, "Corrigendum: Algorithm 806: Sprng: a scalable library for pseudorandom number generation," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 618–619, 2000.

[13] ——, "Parameterizing parallel multiplicative lagged-Fibonacci generators," *Parallel Computing*, vol. 30, no. 5-6, pp. 899–916, 2004.

[14] A. D. Matteis and S. Pagnutti, "A class of parallel random number generators," *Parallel Computing*, vol. 13, no. 2, pp. 193–198, 1990.

[15] ——, "Long-range correlations in linear and nonlinear random number generators," *Parallel Computing*, vol. 14, no. 2, pp. 207–210, 1990.

[16] A. Munshi, *The OpenCL Specification v. 1.0*. Khronos OpenCL Working Group, 2009.

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, 2008.

[18] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[19] NVIDIA, *CUDA Toolkit 4.1. CURAND Guide*. NVIDIA Corporation, 2012.

[20] NVIDIA Corporation, *CUDA Programming Guide*. NVIDIA Corporation, 2009, available at http://www.nvidia.com/.

[21] ——, "NVIDIA next generation CUDA compute architecture: Fermi," http://www.nvidia.com/, 2009.

[22] G. Ökten and M. Willyard, "Parameterization based on randomized quasi-Monte Carlo methods," *Parallel Computing*, vol. 36, no. 7, pp. 415–422, 2010.

[23] O. E. Percus and M. H. Kalos, "Random number generators for MIMD parallel processors," *J. Parallel Distrib. Comput.*, vol. 6, no. 3, pp. 477–497, 1989.

[24] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, 2nd Edition*. Cambridge University Press, 1992.

[25] A. Srinivasan, M. Mascagni, and D. Ceperley, "Testing parallel random number generators," *Parallel Computing*, vol. 29, no. 1, pp. 69–94, 2003.

[26] P. Stpiczyński, "Numerical evaluation of linear recurrences on high performance computers and clusters of workstations," in *2004 International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004), 7-10 September 2004, Dresden, Germany*. IEEE Computer Society, 2004, pp. 200–205. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/PCEE.2004.46

[27] ——, "Solving linear recurrence systems using level 2 and 3 BLAS routines," in *Parallel Processing and Applied Mathematics, 5th International Conference, PPAM 2003, Czestochowa, Poland, September 7-10, 2003. Revised Papers*, ser. Lecture Notes in Computer Science, vol. 3019. Springer, 2004, pp. 1059–1066.

[28] ——, "Solving linear recurrence systems on hybrid GPU accelerated manycore systems," in *Proceedings of the Federated Conference on Computer Science and Information Systems, September 18-21, 2011, Szczecin, Poland*. IEEE Computer Society Press, 2011, pp. 465–470.

[29] P. Stpiczyński and J. Potiopa, "Solving a kind of boundary-value problem for ordinary differential equations using Fermi - the next generation CUDA computing architecture," *J. Computational Applied Mathematics*, vol. 236, no. 3, pp. 384–393, 2011.

[30] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, pp. 232–240, 2010.

```
typedef unsigned int uint;

__global__ void step2gpu(uint *t, uint *y, uint *z, uint r, uint s) {
        for (uint i=1; i<r; i++) {
                z[i*s+s-1]=z[(i-1)*s+s-1]*y[s-1]+t[s-1];
        }//for
}//__global__ voud step2gpu

__global__ void step3gpushared(uint *t, uint *y, uint *z, uint r, uint s) {
        __shared__ uint t_shr[256];
        __shared__ uint y_shr[256];
        uint j=blockIdx.x*blockDim.x+threadIdx.x;
        t_shr[threadIdx.x]=t[j];
        y_shr[threadIdx.x]=y[j];
        if (j<s-1) {
            for (uint i=1; i<r; i++) {
                z[i*s+j]=t_shr[threadIdx.x]+z[(i-1)*s+s-1]*y_shr[threadIdx.x];
            }//for
        }//if
}//__global__ void step3gpushared

//step 2: calculate last row of the matrix Z on device
    step2gpu<<<1, 1>>>(t_gpu, y_gpu, array, r, s);

//step 3: calculate rest of matrix on device
    step3gpushared<<<s/256, 256>>>(t_gpu, y_gpu, array, r, s);
```

Fig. 3. Kernels for **Step2** and **Step3** of (parallel) LCG

```
__global__ void compute_columns(uint *x, uint *y, uint r, uint s) {
        uint row=blockIdx.x*blockDim.x+threadIdx.x;
        uint i, k, tmp;
        __shared__ uint y_shared[512+p2];
        y_shared[threadIdx.x]=y[row];
        if (threadIdx.x<p2) {
                y_shared[512+threadIdx.x]=y[512+row];
        }//if
        __syncthreads();
        if (row<s-p2) {
          for (i=1; i<r; i++) {
            tmp=0;
            for (k=0; k<p2; k++) {
                tmp=tmp+x[i*s-p2+k]*y_shared[threadIdx.x+p2-k];
            }//for
            for (k=0; k<p1; k++) {
                tmp=tmp+x[i*s-p1+k]*y_shared[threadIdx.x+p2-k];
            }//for
            x[i*s+row]=tmp;
          }//for
        }//if
}//void compute_columns
```

Fig. 4. Kernel for **Step3** of (parallel) LFG

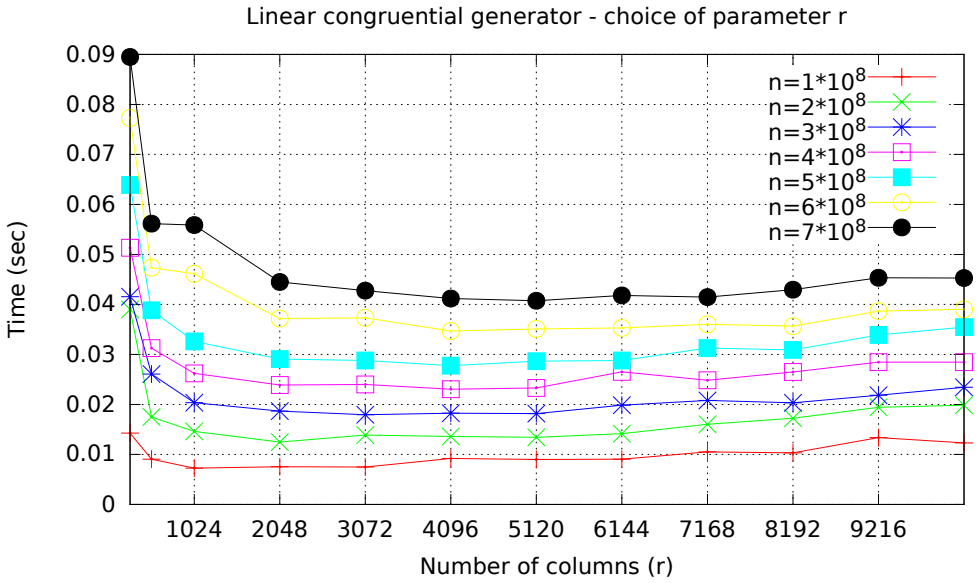

Fig. 5. Speedup of the hybrid CPU+GPU implementation over the CPU implementation

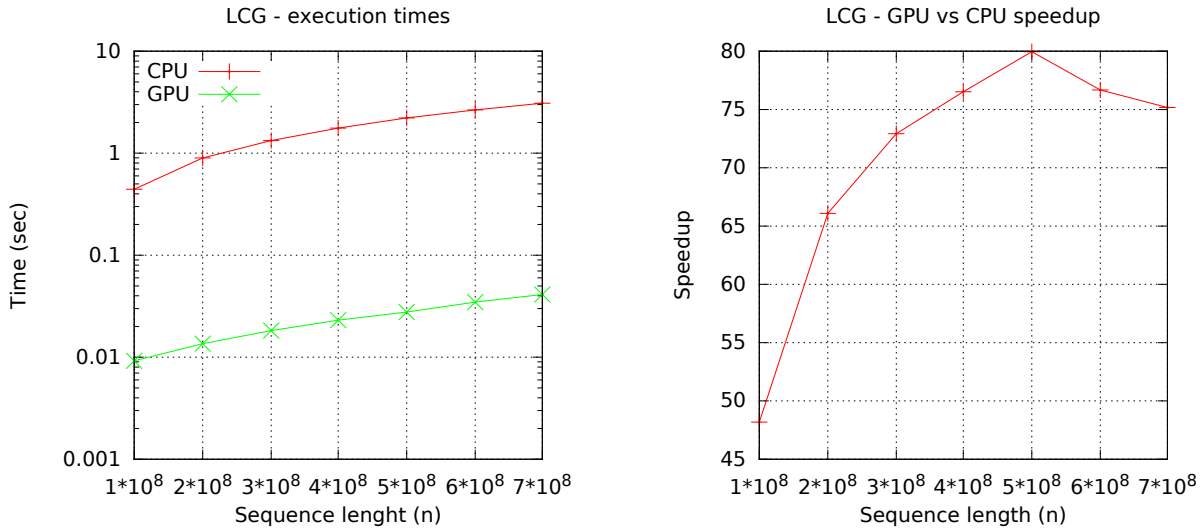

Fig. 6. Linear congruential generator: execution times (left, logarithmic scale) and CPU+GPU vs CPU speedup (right)

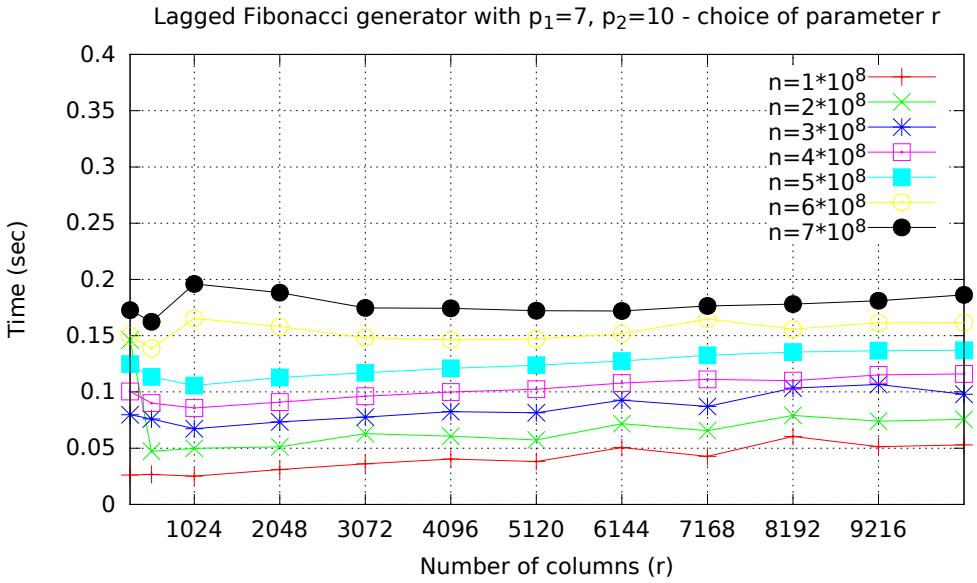

Fig. 7. Speedup of the hybrid CPU+GPU implementation over the CPU implementation

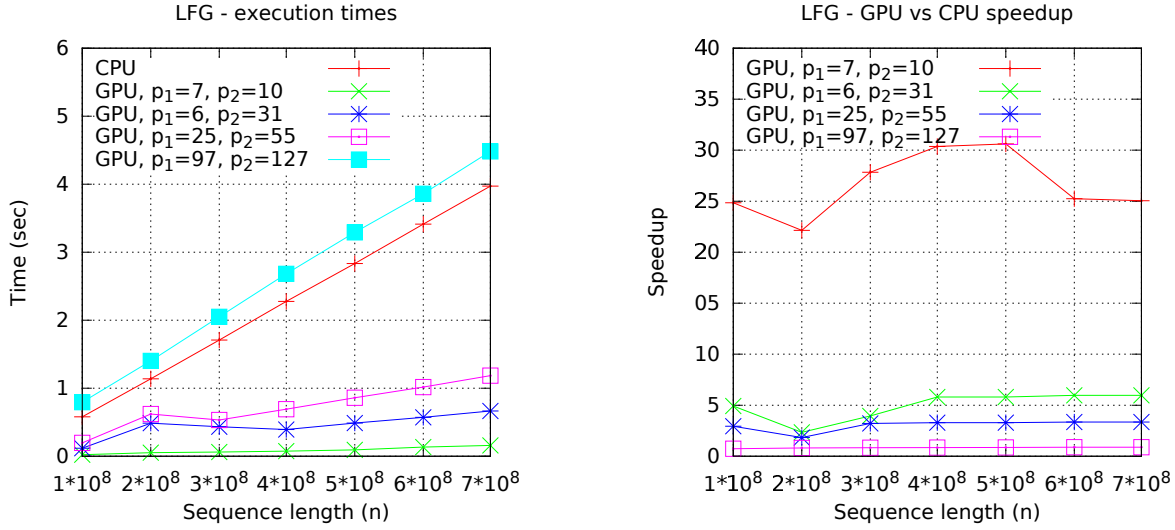

Fig. 8. Linear Fibonacci generator: execution times (left) and CPU+GPU vs CPU speedup (right)