Template Library for Multi-GPU Pseudorandom Number Recursion-based Generators

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

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

***Abstract*—The aim of the paper is to show how to design and implement fast parallel algorithms for Linear Congruential, Lagged Fibonacci and Wichmann-Hill pseudorandom number generators. The new algorithms employ the *divide-and-conquer* approach for solving linear recurrence systems. They are implemented on multi GPU-accelerated systems using CUDA. Numerical experiments performed on a computer system with two Fermi GPU cards show that our software achieve good performance in comparison to the widely used NVIDIA CURAND Library.**

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. They are used by Monte Carlo methods, especially in case of multidimensional numerical integration [1], [4], [9]. In [8] we showed the general techniques for implementing recursion-based generators of pseudorandom numbers on GPU-accelerated systems which are much more efficient than their sequential counterparts.

NVIDIA CURAND Library [5] provides routines for simple and efficient generation of high-quality random numbers. It comprises two types of generators:

- XORWOW, MRG32K3A and MTGP32 are *pseudorandom number generators* which means that a sequence of random numbers which they produce satisfy most of desired statistical properties of a truly random sequence and they work on 32-bit numbers,
- SOBOL32, SCRAMBLED_SOBOL32, SOBOL64, SCRAMBLED_SOBOL64 are *quasirandom number generators*, n-dimensional points obtained from these fill n-dimensional space evenly, SOBOL32, SCRAMBLED_SOBOL32 use 32-bit arithmetic and SOBOL64, SCRAMBLED_SOBOL64 use 64-bit arithmetic.

Unfortunately, these generators utilize only a single GPU device, thus if we want to perform computations using multiple GPUs, we should apply some parametrization techniques for parallel generation of pseudorandom numbers [3] what can lead to possible unwanted correlations between numbers resulting in their poor statistical properties [6]. It should be noticed that only one generator from CURAND produces fully 64-bit results.

In this paper we show how to design fast parallel algorithms for Linear Congruential, Lagged Fibonacci [3] and Wichmann-Hill [10] pseudorandom number generators which employ the *divide-and-conquer* approach for solving linear recurrence systems [7] and can be easily used in computations on multi-GPU systems. Our generators have exactly the same statistical properties as their sequential counterparts.

Numerical experiments performed on a computer system with two Fermi GPU cards show that they achieve good speedup in comparison to the standard CPU-based sequential algorithms [8] and implementations provided by NVIDIA CURAND Library. Our implementation is freely available as the C++ template library which requires only CUDA Toolkit. It can be downloaded from `http://dominisz.umcs.lublin.pl/gpu-rand`.

II. Parallel Pseudorandom Number Generators

We consider the following three pseudorandom number generators:

1) **Linear Congruential Generator (LCG)**: $x_{i+1} \equiv (ax_i + c)(\mod m)$, where x_i is a sequence of pseudorandom values, $m > 0$ is the *modulus*, a, $0 < a < m$ is the *multiplier*, c, $0 \le c < m$ is the *increment*, x_0, $0 \le x_0 < m$ is the *seed* or *start value*,
2) **Lagged Fibonacci Generator (LFG)**: $x_i \equiv (x_{i-p_1} + x_{i-p_2})(\mod m)$, where $0 < p_1 < p_2$,
3) **Wichmann-Hill Generator (WHG, [10])**:

$$\begin{aligned}
x_i &\equiv 11600x_{i-1}(\mod 2147483579)\\
y_i &\equiv 47003y_{i-1}(\mod 2147483543)\\
z_i &\equiv 23000z_{i-1}(\mod 2147483423)\\
t_i &\equiv 33000t_{i-1}(\mod 2147483123)\\
W &\equiv x_i/2147483579.0 + y_i/2147483543.0\\
&\quad + z_i/2147483423.0 + t_i/2147483123.0\\
W_i &\equiv W - \lfloor W \rfloor .
\end{aligned} \tag{1}$$

It should be noted that, in fact, WHG combines four LCG generators, each with the increment 0 (such generator is also called Multiplicative Congruential Generator, MCG). This generator has much better statistical properties than LCG. Its period is about 2^{121}. It passes Big Crush test from TestU01 Library [2].

In case of LCG and LFG, $m = 2^M$, where $M = 32$ or $M = 64$, thus these generators produce numbers from $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. 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 int` data types, we can neglect "(mod m)". In case of WHG, we have moduli given explicitly. Note that the integers x_k are between 0 and $m-1$. They can be converted to real values $r_k \in [0, 1)$ by $r_k = x_k/m$.

It is clear that LCG, LFG, WHG generators can be considered as special cases of linear recurrence systems [7]. Indeed, LCG can be defined as

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

and similarly for LFG we have

$$\begin{cases} x_i = d_i & i = 0, \ldots, p_2 - 1 \\ x_i = x_{i-p_1} + x_{i-p_2}, & i = p_2, \ldots, n-1. \end{cases} \tag{3}$$

The details of our single-GPU implementations of LCG and LFG generators can be found in [8]. Here we only recall the most important formulas. The parallel version LCG can be expressed as follows

$$\begin{cases} \mathbf{x}_0 = A^{-1}\mathbf{f}_0 \\ \mathbf{x}_i = \mathbf{t} + x_{is-1}\mathbf{y}, \quad i = 1, \ldots, r-1, \end{cases} \tag{4}$$

where $\mathbf{x}_i = (x_{is}, \ldots, x_{(i+1)s-1})^T \in \mathbb{Z}_m^s$, $\mathbf{f}_0 = (d, c, \ldots, c)^T \in \mathbb{Z}_m^s$, $\mathbf{f} = (c, \ldots, c)^T \in \mathbb{Z}_m^s$, and

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

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

Similarly, for LFG we have

$$\begin{cases} \mathbf{x}_0 = A_0^{-1}\mathbf{f} \\ \mathbf{x}_i = \sum_{k=0}^{p_2-1} x_{is-p_2+k}\mathbf{y}_k + \sum_{k=0}^{p_1-1} x_{is-p_1+k}\mathbf{y}_k, \\ \qquad\qquad i = 1, \ldots, r-1, \end{cases} \tag{5}$$

where matrix A_0 and vectors $\mathbf{f}$, $\mathbf{y}_k$ are defined analogously as for LCG case (see [8] for details). Note that (5) is the generalization of (4).

III. Multi-GPU implementation

To implement the parallel algorithms efficiently on GPU, we will form the following matrix

$$Z = [\mathbf{x}_0, \ldots, \mathbf{x}_{r-1}] \in \mathbb{Z}_m^{s \times r}, \tag{6}$$

where all vectors $\mathbf{x}_i$ are defined by (4) or (5). This allows to use fast coalesced memory access and makes possible the use of shared memory.

The equation (4) has a lot of potential parallelism. The algorithm comprises the following steps. First (Step 1) we have to find $\mathbf{y}$, $\mathbf{t}$. Then (Step 2) we find the last entry of each vector $\mathbf{x}_i$, $i = 1, \ldots, r-1$. Finally (Step 3), we find $s-1$ entries of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{r-1}$ in parallel. In case of multi-GPU implementation vectors $\mathbf{y}$, $\mathbf{t}$ are computed by CPU and then sent to all GPU devices. The generator seed required to compute Step 2 is received from the previous GPU device and sent to the next one after Step 2 is completed locally (Figure 1). Then all GPUs perform Step 3 independently.

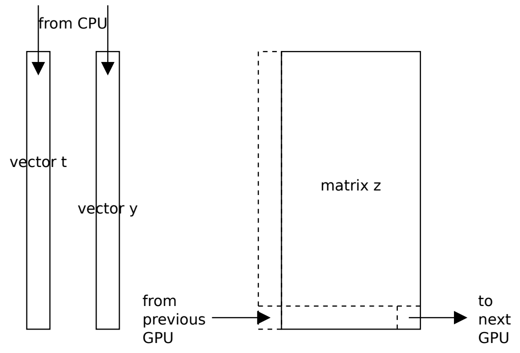

Fig. 1. LCG: data structures on a GPU device and communication scheme

We can develop a similar parallel algorithm for LFG. During the first step we have to find vector $\mathbf{y}_0$. This vector is computed by CPU and sent to all GPUs. It is easy to verify that

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

Then (Step 2) using (5) we find p_2 last entries of $\mathbf{x}_1, \ldots, \mathbf{x}_{r-1}$. Finally (Step 3) we use (5) to find $s - p_2$ first entries of these vectors in parallel. Note that Step 2 requires communication (sending and receiving the seed consisting of p_2 numbers) between GPU devices (Figure 2).

The parallel algorithm for WHG is a simple extensions of the parallel LCG. Instead of vectors $\mathbf{t}$ and $\mathbf{y}$, we have four instances of $\mathbf{y}$, each for one MCG. We also have four separate "last rows" of matrix Z, corresponding to appropriate MCG, which are required during Step 2 (Figure 3).

IV. 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. 5.0 provided by the vendor. The results of experiments are presented in Figures 4-6. We can conclude the following:

- Parallel LCG is the fastest among the considered generators. It produces $\approx 32 \cdot 10^6$ `unsigned int` pseudorandom numbers per second, while the fastest CURAND pseudorandom generator achieves the speed of $\approx 12 \cdot 10^6$.

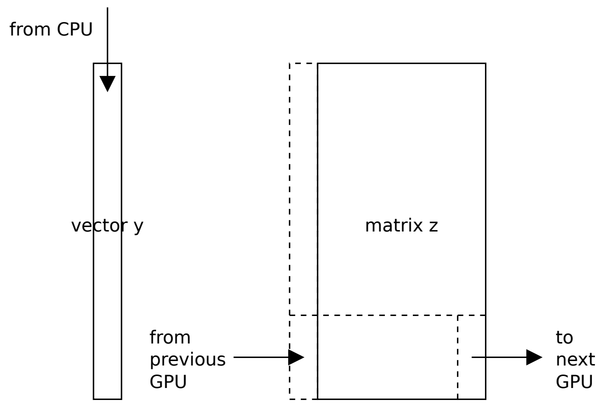

Fig. 2. LFG: data structures on a GPU device and communication scheme

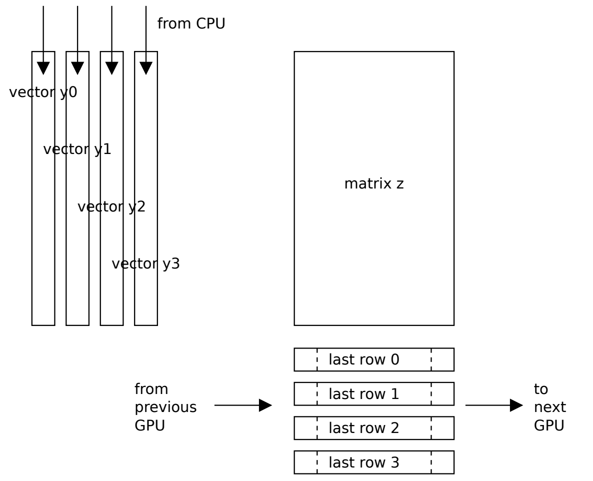

Fig. 3. WHG: data structures on a GPU device and communication scheme

- Parallel WHG is about 6 times slower than LCG. However it uses more computations and communications in comparison to LCG. It also has better statistical properties, so it should be use instead of LCG, when the performance is not so important.
- The performance of parallel LFG depends on the values of p_1 and p_2.
- Our template library provides both 32-bit and 64-bit versions of all generators. CURAND does not support 32-bit or 64-bit arithmetic in all cases (hence missing bars in Figure 4).
- The use of two GPUs accelerates the overall time of computations (Figure 6). In case of LCG we obtain almost linear speedup. The scalability of WHG is worse because of longer lasting Step 2. Unfortunately, the scalability of LFG is poor for large values of p_1, p_2.

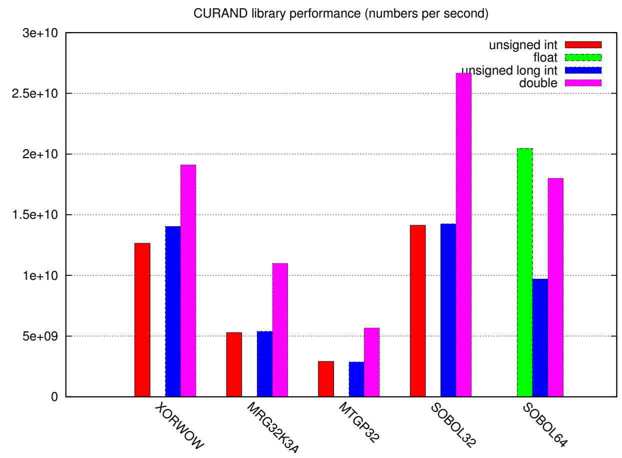

Fig. 4. CURAND Library performance: generation of random number using various generators

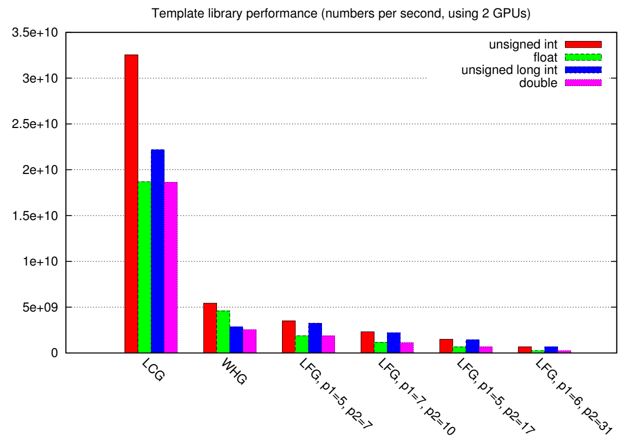

Fig. 5. Our template library performance: generation of random number using various generators

V. Using template library

Let us consider the use our template library in case of LCG. The following class template should be used (we only show `public:` part of it).

```
template <class T>
class LcgGpu {
  public:
    LcgGpu(T multiplier, T increment,
           T seed, size_t count);
    void generate();
    void generateFloat();
    void generateDouble();
    T* getNumbersFromDevice(int device);
    size_t getCountFromDevice(int device);
    int getDeviceCount();
    ...
}//class LcgGpu
```

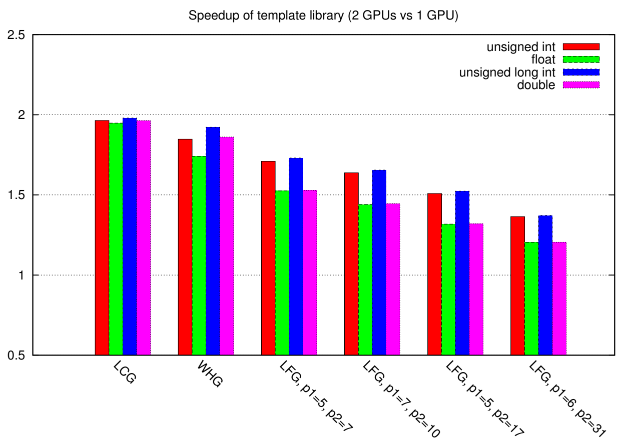

Fig. 6. Our template library: speedup 2 GPUs vs 1 GPU

In order to use the generator we should create an object providing desired parameters (multiplier, increment, seed of the generator and the number of pseudorandom numbers to generate).

```
unsigned int multiplier=1664525;
unsigned int increment=1013904223;
unsigned int seed=31;
size_t count=100000000;

LcgGpu<unsigned int> lcg
  =new LcgGpu<unsigned int>(multiplier,
                            increment,
                            seed, count);
```

Then we generate random numbers (e.g. uniformly distributed real numbers from interval $[0, 1)$).

```
lcg->generateFloat();
```

Generated numbers are stored in global memories of all GPU devices. We use the following routines to obtain the number of GPU devices, which produce numbers, the number of pseudorandom numbers generated by a given device and the address of memory block containing the numbers:

```
int getDeviceCount();
size_t getCountFromDevice(int device);
T* getNumbersFromDevice(int device);
```

Generated numbers can be used directly by each GPU or can be transferred to CPU memory using loop for accessing all devices.

```
for (i=0; i<lcg->getCountFromDevice(); i++) {
        cudaMemcpy(cpuNumbers+offset,
          lcg->getNumbersFromDevice(i),
          lcg->getCountFromDevice(i)
            *sizeof(unsigned int),
          cudaMemcpyDeviceToHost);
    offset=offset
      +lcg->getCountFromDevice(i);
}//for
```

Generation of numbers can be repeated as many times as desired to obtain very long sequence of pseudorandom numbers. Finally we can delete the object.

```
delete lcg;
```

Using LFG generator is quite similar. The only difference is when the object is created. For example, we can use the following code.

```
unsigned int p1=24;
unsigned int p2=55;
unsigned int seed[]={...}; //array of
                           //length p2
size_t count=100000000;

LfgGpu<unsigned int> lfg=
  new LfgGpu<unsigned int>(p1, p2
                           seed, count);
```

Analogously for WHG we use the following.

```
unsigned int seedX=389933028;
unsigned int seedY=148667295;
unsigned int seedZ=146045161;
unsigned int seedT=767880647;

WhgGpu<unsigned int> whg=
  new WhgGpu<unsigned int>(seedX, seedY,
                           seedZ, seedT,
                           count);
```

When we parametrize template with `unsigned int` type then we can use the following routines to generate 32-bit pseudorandom numbers (integer or real numbers).

```
void generate();
void generateFloat();
```

When we need 64-bit precision we use `unsigned long int` type to parametrize template and the following routines.

```
void generate();
void generateDouble();
```

VI. Conclusions

We have showed how to implement fast parallel LCG, LFG and WHG pseudorandom number generators using the *divide-and-conquer* approach on contemporary multi-GPU systems. Numerical experiments performed on a computer system with modern Fermi GPU cards showed that our routines achieve good performance in comparison to the widely used NVIDIA CURAND Library. Our template library is easy to use and it is freely available for the community.

ACKNOWLEDGEMENTS

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] J. M. Bull and T. L. Freeman, "Parallel globally adaptive quadrature on the KSR-1," *Adv. Comput. Math.*, vol. 2, pp. 357–373, 1994. [Online]. Available: http://dx.doi.org/10.1007/BF02521604

[2] P. L'Ecuyer and R. J. Simard, "TestU01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, 2007. [Online]. Available: http://doi.acm.org/10.1145/1268776.1268777

[3] 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.

[4] H. Niederreiter, "Quasi-Monte Carlo methods and pseudo-random numbers," *Bull. Am. Math. Soc.*, vol. 84, pp. 957–1041, 1978.

[5] NVIDIA, *CUDA Toolkit 5.0. CURAND Guide*. NVIDIA Corporation, 2012.

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

[7] P. Stpiczyński, "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. [Online]. Available: http://fedcsis.eucip.pl/proceedings/pliks/148.pdf

[8] P. Stpiczyński, D. Szałkowski, and J. Potiopa, "Parallel GPU-accelerated recursion-based generators of pseudorandom numbers," in *Proceedings of the Federated Conference on Computer Science and Information Systems, September 9-12, 2012, Wroclaw, Poland.* IEEE Computer Society Press, 2012, pp. 571–578. [Online]. Available: http://fedcsis.org/proceedings/fedcsis2012/pliks/380.pdf

[9] D. Szałkowski and P. Stpiczyński, "Multidimensional monte carlo integration on clusters with hybrid gpu-accelerated nodes," 2013, submitted to PPAM2013.

[10] B. A. Wichmann and I. D. Hill, "Generating good pseudo-random numbers," *Comput. Stat. Data Anal.*, vol. 51, no. 3, pp. 1614–1622, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.csda.2006.05.019