J Supercomput (2018) 74:936–952
https://doi.org/10.1007/s11227-017-2172-x

Vectorized algorithm for multidimensional Monte Carlo integration on modern GPU, CPU and MIC architectures

Przemysław Stpiczyński[1]

Published online: 31 October 2017

Abstract The aim of this paper is to show that the multidimensional Monte Carlo integration can be efficiently implemented on computers with modern multicore CPUs and manycore accelerators including Intel MIC and GPU architectures using a new vectorized version of LCG pseudorandom number generator which requires limited amount of memory. We introduce two new implementations of the algorithm based on directive-based parallel programming standards OpenMP and OpenACC and consider their performance using Hockney–Jesshope theoretical model of vector computations. We also present and discuss the results of experiments performed on dual-processor Intel Xeon E5-2670 computers with Intel Xeon Phi 7120P and NVIDIA K40m.

Keywords Multidimensional Monte Carlo integration · Vectorized linear congruential generator · Performance analysis · GPU · Intel MIC · OpenMP · OpenACC

1 Introduction

Many problems in physics, chemistry and financial analysis involve computing of multidimensional integrals of the form

$$V = \int_{I^d} f(\mathbf{v}) d\mathbf{v}, \tag{1}$$

where $I^d = [0, 1]^d$ and $d \geq 1$. Very often such problems have to be solved numerically because their analytical solutions are not known. Monte Carlo methods are a broad

✉ Przemysław Stpiczyński
przem@hektor.umcs.lublin.pl

[1] Institute of Mathematics, Maria Curie–Skłodowska University, Pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin, Poland

class of computational algorithms that rely on repeated random sampling to obtain numerical results approximating exact analytical solutions. Using parallel computers can accelerate the performance of programs that use such methods in science, technology or financial applications [3,20,22,24]. Monte Carlo methods are also very attractive for solving multidimensional integration problems [1]. However, to ensure appropriately high accuracy of the results, the number of random samples should be really huge [15]; thus, the use of parallel processing is necessary. Unfortunately, high-quality algorithms for generation pseudorandom points have rather sequential nature. For example, the routine `D01GBF` available in NAG Numerical Library computes the approximation of the multidimensional integral of a function using Monte Carlo algorithm described in [12] and does not utilize multiple cores of target architectures and its performance is really poor [29].

In [28,29] we have shown that the multidimensional Monte Carlo integration can be efficiently implemented on various distributed memory parallel computers and clusters of multicore GPU-accelerated nodes using recently developed parallel versions of LCG and LFG pseudorandom number generators [26]. Unfortunately, those implementations use all available memory to generate in parallel a huge number of random points. In such a case it is necessary to increase the number of nodes within a cluster when higher accuracy of results is desired.

The aim of this paper is to introduce a new simplified algorithm for multidimensional Monte Carlo integration based on a new vectorized version of LCG pseudorandom number generator which requires limited amount of memory. We consider its performance using Hockney–Jesshope model of vector computations [4,6]. This model can also be applied to find the values of important parameters of the algorithm to minimize its execution time. Our new portable implementations are based on two directive-based parallel programming standards OpenMP and OpenACC; thus, the algorithm can be run on computers with modern multicore CPUs and manycore accelerators including Intel MIC and GPU architectures. We also present and discuss the results of experiments performed on dual-processor Intel Xeon E5-2670 computers with Intel Xeon Phi 7120P and NVIDIA K40m.

2 Multidimensional Monte Carlo integration

The idea of Monte Carlo methods for multidimensional integration can be found in [15]. For a given $d \geq 1$ let $I^d = [0, 1]^d$ be the d-dimensional unit cube and let $f(\mathbf{v})$ be a bounded Lebesgue-integrable function defined on I^d. The approximation for the integral of f over I^d can be found using

$$\int_{I^d} f(\mathbf{v}) d\mathbf{v} \approx \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{v}_i), \tag{2}$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_N$ are random points from I^d. The strong law of large numbers guarantees that such numerical approximation of the integral converges almost surely and from the central limit theorem it follows that the expected error is $O(N^{-1/2})$ [15]. The idea of Monte Carlo algorithm for multidimensional integration is quite simple

[29]. We calculate the sum of the values of $f(\mathbf{v}_i)$, where all $\mathbf{v}_i \in I^d$, $i = 1, \ldots, N$, are constructed using a sequence of $N \cdot d$ pseudorandom real numbers from [0, 1).

Linear congruential generator (LCG) is a well-known pseudorandom number generator. It is based on the recurrence relation

$$x_{i+1} \equiv (ax_i + c)(\mod m), \tag{3}$$

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*, $x_0 \equiv \sigma$, $0 \leq \sigma < m$ is the *seed* or *start value*. Usually, $m = 2^M$, and $M = 32$ or $M = 64$; thus, the generators produce numbers from $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. It allows the modulus operations to be computed by merely truncating all but the rightmost 32 or 64 bits, respectively. Thus, we can neglect "$(\mod m)$". The integers x_k can be converted to real values $v_k \in [0, 1)$ by $v_k = x_k/m$.

Properties of the sequence generated using (3) heavily depend on properties of a, c and m. The sequence has the maximum possible period, when the following conditions are preserved [10]:

1. c is relatively prime to m,
2. every prime factor of m also divides $a - 1$,
3. if 4 divides m, then 4 divides $a - 1$.

It is clear that m should be large enough. For $m = 2^{64}$, LCG can be used to generate sequences of double-precision real numbers. Then $a = 6364136223846793005$ and $c = 1442695040888963407$ is a good choice [11]. When single precision is sufficient, one can chose $m = 2^{32}$, $a = 1664525$, $c = 1013904223$ [19].

Algorithm 1: Sequential multidimensional Monte Carlo integration based on LCG defined by the values of m,σ,a,c

Data: $f(\cdot)$ – given function defined on I^d, N – the number of points
Result: the approximation of $\int_{I^d} f(\mathbf{v})d\mathbf{v}$ calculated using (2)

$x_0 \leftarrow \sigma$
$v_0 = x_0/m$
for $j = 1, \ldots, d-1$ **do**
  $x_j \leftarrow ax_{j-1} + c$
  $v_j = x_j/m$
end
$sum \leftarrow f(v_0, \ldots, v_{d-1})$
for $i = 1, \ldots, N-1$ **do**
  $x_0 \leftarrow ax_{d-1} + c$
  $v_j = x_j/m$
  for $j = 1, \ldots, d-1$ **do**
    $x_j \leftarrow ax_{j-1} + c$
    $v_j = x_j/m$
  end
  $sum \leftarrow sum + f(v_0, \ldots, v_{d-1})$
end
return sum/N

Algorithm 1 performs multidimensional Monte Carlo integration based on LCG. Note that we need N points from I^d; thus, we apply (3) to generate $N \cdot d$ points. It is clear that LCG is a special case of linear recurrence systems [25] that can be defined as follows

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

Algorithms strictly based on (4) cannot fully utilize the underlying hardware of modern computers, i.e., vector extensions and multiple cores, what is essential in case of modern multicore and manycore computer architectures.

To introduce a method that would be suitable for modern computer architectures, let us assume that there exist two positive integers r and s such that $n = rs$. This assumption can be easily omitted. If we choose $n < N$, where N is the desired number of points, then we can use the following method to find n random points from I^d. Remaining $N - n$ points can be found using (3). Equation (4) can be rewritten in the following block form [26,29]:

$$\begin{bmatrix} A & & & \\ B & A & & \\ & \ddots & \ddots & \\ & & B & A \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f} \\ \vdots \\ \mathbf{f} \end{bmatrix}, \tag{5}$$

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

$$A = \begin{bmatrix} 1 & & & \\ -a & 1 & & \\ & \ddots & \ddots & \\ & & -a & 1 \end{bmatrix} \in \mathbb{Z}_m^{s \times s}, \qquad B = \begin{bmatrix} 0 & \cdots & 0 & -a \\ \vdots & \ddots & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} \in \mathbb{Z}_m^{s \times s}.$$

From (5) we have

$$\begin{cases} A\mathbf{x}_0 = \mathbf{f}_0 \\ B\mathbf{x}_{i-1} + A\mathbf{x}_i = \mathbf{f}, \quad i = 1, \ldots, r-1. \end{cases} \tag{6}$$

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 the main formula for the vectorized version of LCG:

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

Algorithm 2 shows how to apply (7) to perform multidimensional Monte Carlo integration. First we have to find vectors $\mathbf{x}_0$, $\mathbf{y}$, $\mathbf{t}$ using three separate tasks that can be executed in parallel. Note that because of data dependencies, the loops 2–4, 6–8, 10–12 are strictly sequential. Then we convert all entries of $\mathbf{x}_0$ to real values (line

Algorithm 2: Vectorized multidimensional Monte Carlo integration based on LCG defined by the values of m,σ,a,c

Data: $f(\cdot)$ – given function defined on I^d, n – the number of points, r, s – parameters of the method such that $r \cdot s = n$
Result: the approximation of $\int_{I^d} f(\mathbf{v})d\mathbf{v}$ calculated using (2)

```
1  x_0 ← σ                                          ▷ lines 1–4: x ← A^{-1} f_0
2  for j = 1, ..., sd − 1 do
3  |  x_j ← a x_{j−1} + c
4  end
5  y_0 ← a                                          ▷ lines 5–8: y ← A^{-1}(a e_0)
6  for j = 1, ..., sd − 1 do
7  |  y_j ← a y_{j−1}
8  end
9  t_0 ← c                                          ▷ lines 9–12: t ← A^{-1} f
10 for j = 1, ..., sd − 1 do
11 |  t_j ← a t_{j−1} + c
12 end
13 v = x/m                                          ▷ v_j ∈ [0, 1)
14 sum ← 0
15 parfor j = 0, ..., s − 1 do
16 |  sum ← sum + f(v_{jd}, ..., v_{(j+1)d−1})     ▷ atomic update of the variable sum
17 end
18 for i = 1, ..., r − 1 do
19 |  α ← x_{sd−1}
20 |  x ← t + αy
21 |  v = x/m                                       ▷ v_j ∈ [0, 1)
22 |  parfor j = 0, ..., s − 1 do
23 |  |  sum ← sum + f(v_{jd}, ..., v_{(j+1)d−1})  ▷ atomic update of the variable sum
24 |  end
25 end
26 return sum/N
```

13), using fully vectorized operation and find the sum of $f(\mathbf{v}_j)$ for $j = 1, \ldots, s-1$ (lines 15–17, parallel loop). In the second part of the algorithm, we find all vectors $\mathbf{x}_i$, $i = 1, \ldots, r-1$, convert their entries to real values using fully vectorized operations (lines 20 and 21, respectively) and find (lines 22–24, parallel loop) the sum of $f(\mathbf{v}_j)$. It should be noticed that although the operations from lines 13, 20, 21 are written in a vector form, they can be treated as loops without data dependencies.

3 Performance analysis

It is clear that the performance of Algorithm 2 depends on chosen values of r and s. Our experiments show that the right choice of these parameters can improve the performance significantly. Moreover, the right choice remains appropriate for various functions f (see Sect. 5). Thus, for the sake of simplicity, we will use the theoretical model of vector computations introduced by Hockney and Jesshope [4,6] to analyze Algorithm 2 reduced to the problem of finding $r \cdot s$ vectors from I^d. It will also help

us to predict the right choice of r and s that can minimize the execution time of the algorithm.

The performance of computers involved in scientific calculations is usually expressed in terms of the number of floating point operations completed per second. The basic measure can be expressed as

$$r_{\text{avg}} = \frac{N}{10^9 \cdot t} \text{ Gflops}, \tag{8}$$

where N represents the number of floating point operations executed in t seconds. Obviously, when N floating point operations is executed with the average performance of r_{avg}, the execution time of a given algorithm is

$$t = \frac{N}{10^9 \cdot r_{\text{avg}}} \text{ sec}. \tag{9}$$

The performance r_N of a loop of length N can be expressed in terms of two parameters r_∞ and $n_{1/2}$, which are specific for a given type of a loop and computer architectures [4,6]. The first parameter represents the performance in Gflops for a very long loop, while the second the loop length for which the performance of about $r_\infty/2$ is achieved. Then

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Gflops}. \tag{10}$$

Let m denotes the number of floating point operations repeated during each iteration of a given loop. Applying (9) and (10) we get the execution time of such a loop of length N

$$t_N = \frac{mN}{10^9 r_N} = \frac{m \cdot 10^{-9}}{r_\infty}(n_{1/2} + N) \text{ sec}. \tag{11}$$

The total execution time of Algorithm 2 reduced to the problem of finding $r \cdot s$ vectors from I^d (i.e., without lines 14–17 and 22–24) depends on the values of parameters r, s and it can be expressed as $T(r, s) = \sum_{i=1}^{3} t_i$, where t_1 is the execution time of the first three parallel tasks (the loops 2–4, 6–8, 10–12, respectively), t_2 is the time needed to convert nd integers to real values (lines 13 and 21), t_3 is the time needed for $r - 1$ executions of the line 20. Using (11) we get the following (in seconds):

$$t_1 = \frac{2 \cdot 10^{-9}}{r^A_\infty}(n^A_{1/2} + sd - 1),$$
$$t_2 = r \cdot \frac{10^{-9}}{r^B_\infty}(n^B_{1/2} + sd) = \frac{2 \cdot 10^{-9}}{r^B_\infty}(0.5 n^B_{1/2} r + 0.5nd),$$
$$t_3 = (r-1) \cdot \frac{2 \cdot 10^{-9}}{r^C_\infty}(n^C_{1/2} + sd) = \frac{2 \cdot 10^{-9}}{r^C_\infty}(n^C_{1/2} r - sd + nd - n^C_{1/2}),$$

where pairs $(r_\infty^A, n_{1/2}^A)$, $(r_\infty^B, n_{1/2}^B)$ and $(r_\infty^C, n_{1/2}^C)$ characterize the execution of the relative loops. This yields

$$T(r,s) = 2 \cdot 10^{-9} \left(\frac{d}{r_\infty^A} s + \frac{n_{1/2}^B}{2r_\infty^B} r + \frac{n_{1/2}^C}{r_\infty^B} r - \frac{d}{r_\infty^C} s \right) + D, \tag{12}$$

where D denotes the execution time that does not depend on r and s. We assume that $rs = n$; thus, (12) reduces to the following

$$T(s) = 2 \cdot 10^{-9} \left(\frac{d(r_\infty^C - r_\infty^A)}{r_\infty^C r_\infty^A} s + \frac{n(0.5 n_{1/2}^B r_\infty^C + n_{1/2}^C r_\infty^B)}{r_\infty^B r_\infty^C} \cdot \frac{1}{s} \right) + D. \tag{13}$$

It can be easily verified that $T(s)$ defined by (13) reaches its minimum at the point

$$s_{\text{opt}} = \sqrt{\frac{\beta}{d}} n, \qquad \beta = \frac{0.5 n_{1/2}^B r_\infty^C + n_{1/2}^C r_\infty^B}{r_\infty^B (r_\infty^C - r_\infty^A)}. \tag{14}$$

The optimal choice of s depends on the problem size n, the dimension d and the value of β, which is machine-dependent. It should also be noticed that $r_\infty^C > r_\infty^A$, and thus $\beta > 0$. The numbers s and r should be integers; thus, one can choose

$$s^\star = \lfloor s_{\text{opt}} \rfloor, \qquad r^\star = \lfloor n/s^\star \rfloor. \tag{15}$$

It is clear that we need to know the values of $r_\infty^A, r_\infty^B, r_\infty^C$ and $n_{1/2}^B, n_{1/2}^C$ to find β and then $s^\star$ [7]. However, it is also possible to estimate $s^\star$ after some numerical experiments performed for several various values of n and d. The value of the parameter β can be found using

$$\beta = d \cdot s_{\text{opt}}^2 / n. \tag{16}$$

Then one can use (15) to find $s^\star$ and tune the performance of the algorithm for given values of n and d (i.e., to minimize the execution time of the algorithm).

It should be noticed that the theoretical model of performance based on (10) gives us some information about possible *scalability* of vectorized algorithms. When loops are split over available p processors or cores, the values of parameters $r_\infty, n_{1/2}$ grow by a factor of p. However, due to synchronization, r_∞ grows slower, but $n_{1/2}$ faster [4]. Therefore, when the number of processors grows, our method can achieve better performance for bigger problem sizes (i.e., when parallelized loops are sufficiently large).

4 Implementation issues

Both Algorithms 1 and 2 have been implemented in C. The implementation of Algorithm 1 is straightforward; however, one can suppose that a modern compiler can apply some optimization techniques that can improve the overall performance of the algorithm on modern CPUs. For example, Intel C/C++ compiler applies the loop fission

technique in which each of the loops 3–6 and 11–14 is broken into two loops. The first one is strictly sequential and performs $x_j \leftarrow ax_{j-1}+c$, while the second one performs $v_j \leftarrow x_j/m$ and can be vectorized using new SIMD extensions like AVX and AVX-512 which are available in modern multicore and manycore processors [9,27]. It can be enforced by placing the pragma `simd` before each loop [27]. To optimize memory access the array, **v** should be allocated using the `_mm_malloc()` intrinsic. It works just like the `malloc` function and additionally allows data alignment [27]. This loop has limited length (i.e., d); thus, the use of multiple threads cannot be profitable.

Algorithm 2 can easily be parallelized using OpenMP [2]. Lines 1–25 can be treated as a parallel region defined by the `parallel` construct. Lines 1–4, 5–8, and 9–12, respectively, can be treated as three independent sections that can be run in parallel. Lines 13, 20, 21 can be rewritten as loops annotated with `pragma omp for` to be executed in parallel. Such loops can also be vectorized by the compiler using SIMD extensions. The loops from lines 15–17 and 22–24 can also be executed in parallel; however, the variable *result* should be updated and is shown in Fig. 1.

The OpenMP implementation of Algorithm 2 can be used on various multicore CPUs and Intel Xeon Phi working in native mode [8,21]. To develop portable implementation for GPUs, we have used OpenACC [18]. This standard offers compiler directives for offloading C/C++ and Fortran programs from host to attached accelerator devices. Such simple directives allow to mark regions of source code for automatic acceleration in a vendor-independent manner [23]. OpenACC provides the `parallel` construct that launches gangs of workers that will execute in parallel. Gangs may support multiple workers that execute in vector or SIMD mode [18] available in GPUs. The standard also provides several constructs that can be used to specify the scope of data in accelerated parallel regions.

Our OpenACC implementation of Algorithm 2 assumes that the most compute-intensive part of the algorithm, namely the lines 18–25, will be offloaded to an accelerator (see Fig. 2). We use the OpenACC `data` construct to specify the scope of data in the accelerated region. The construct `parallel loop` is used to vectorize the internal loops 20, 21, 22–24. Note that the variable `result` resides in host memory and it is updated using the value of `temp`. We also use the OpenACC construct `update host` to guarantee that the actual value of the last entry of `x` resides in host memory.

It should be noticed that PGI compiler which supports OpenACC has one disadvantage, namely it does not support indirect calls to external functions within accelerated regions. Instead one should consider the use of inlined functions supported by the compiler. Unfortunately, in such a case the source code of the implementation should be recompiled for each function.

5 Results of experiments

Algorithm 2 for the multidimensional integration using the vectorized version of LCG has been tested on three different target architectures which are modern accelerated systems allowing OpenMP and OpenMP+OpenACC programming models:

```
ss=s*d;
#pragma omp parallel
{
  double temp;
  #pragma omp sections
  {
       #pragma omp section
       { y[0]=a;                        // solve A y = ae0
         for(int i=0;i<ss-1;i++)
           y[i+1]=a*y[i];
       }
       #pragma omp section
       { x[0]=sigma;                    // solve A x0 = f0
         for(int i=0;i<ss-1;i++)
           x[i+1]=a*x[i]+c;
         alpha=x[ss-1];
       }
       #pragma omp section
       { t[0]=c;                        // solve A t = f
         for(int i=0;i<ss-1;i++)
           t[i+1]=a*t[i]+c;
       }
  }
  #pragma omp for simd
  for(int i=0;i<ss;i++)
    v[i]=x[i]/two64;  // two64==2^64

  temp=0.0;
  #pragma omp for
  for(int i=0;i<s;i++)
    temp+=fun(d,&v[i*d]);

  #pragma omp atomic
  result+=temp;

 for(int nr=1;nr<r;nr++)
 {
   #pragma omp for simd
   for(int i=0;i<ss;i++)
     x[i]=t[i]+alpha*y[i];

   #pragma omp for simd
   for(int i=0;i<ss;i++)
     v[i]=x[i]/two64;

   temp=0.0;
   #pragma omp for
   for(int i=0;i<s;i++)
     temp+=fun(d,&v[i*d]);

   #pragma omp atomic
   result+=temp;

   #pragma omp single
   {
     alpha=x[s-1];
   }
 }
} // parallel
```

Fig. 1 OpenMP implementation of Algorithm 2

```
ss=s*d;

#pragma omp parallel
{
  double temp;
  #pragma omp sections
  {
    // ...
  }

  // ... the same as in our OpenMP implementation

} // OpenMP parallel region ends here!

#pragma acc data copyin(x[0:ss],t[0:ss],y[0:ss]) create(v[0:ss])
{
 for(int nr=1;nr<r;nr++)
 {
   #pragma acc update host(x[ss-1:1])
   alpha=x[ss-1];

   #pragma acc parallel loop present(t[0:ss],y[0:ss])
   for(int i=0;i<ss;i++)
     x[i]=t[i]+alpha*y[i];

   #pragma acc parallel loop present(x[0:ss],v[0:ss])
   for(int i=0;i<ss;i++)
     v[i]=x[i]/two64;

   double temp=0.0;
   #pragma acc parallel loop reduction(+:temp) present(v[0:ss])
   for(int i=0;i<s;i++)
     temp+=fun(d,&v[i*d]);

   result+=temp;

 } // for

 #pragma acc update host(x[ss-1:1])
} // data
```

Fig. 2 OpenACC implementation of the most compute-intensive part Algorithm 2

E5-2670: a server with two Intel Xeon E5-2670 v3 (totally 24 cores with hyper-threading, 2.3 GHz), 128GB RAM, running under CentOS 6.5 with Intel Parallel Studio ver. 2016,

Xeon Phi: like E5-2670, additionally with Intel Xeon Phi Coprocessor 7120P (61 cores with multithreading, 1.238 GHz, 16GB RAM), all experiments have been carried out on Xeon Phi working in native mode [8,21],

K40m: like E5-2670, additionally with NVIDIA Tesla K40m GPU [13] (2880 cores, 12GB RAM), CUDA 7.0 and Portland Group compilers and tools with OpenMP and OpenACC support.

Algorithm 1 has been tested only on E5-2670. However, due to its sequential nature, only one core has been utilized.

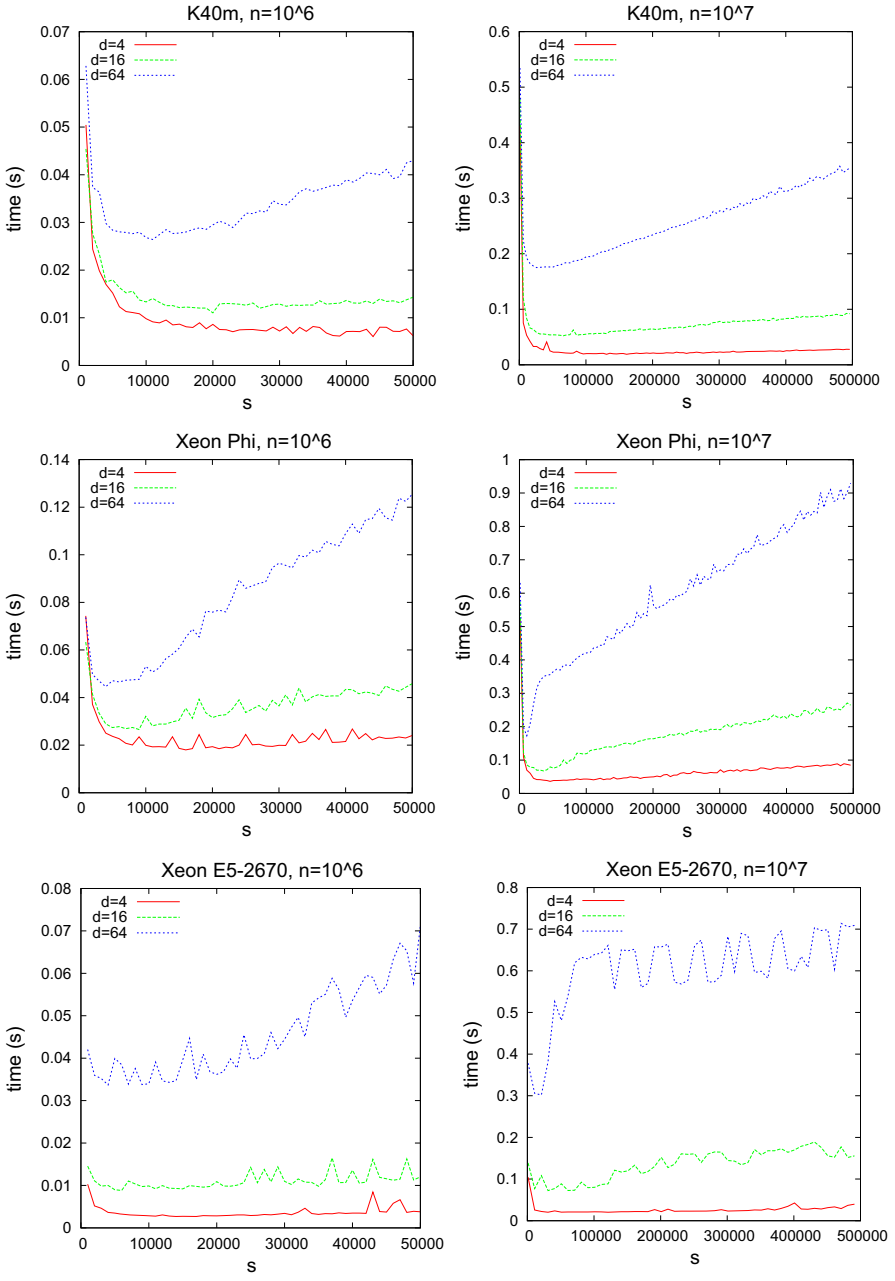

Fig. 3 Execution time of Algorithm 2 reduced to the problem of finding $r \cdot s$ vectors from I^d

Table 1 Estimated β and exemplary values of $s^\star$ for K40m, Xeon Phi and two E5-2670

n	d	Optimal values of β and s		
		K40m, $\beta = 7744$	Xeon Phi, $\beta = 1024$	E5-2670, $\beta = 2822$
$1e+06$	4	44000	16000	26561
	16	22000	8000	13281
	64	11000	4000	6640
$1e+07$	4	139140	50596	83994
	16	69570	25298	41997
	64	34785	12649	20999

First we have tested the performance of the Algorithms 1 and 2 for the test functions proposed in [5] using various values of r and s. We have observed that the right choice of these parameters can significantly improve the performance of Algorithm 2. Moreover, the optimal values of these parameters minimize the execution time for various test functions.

Figure 3 shows the performance of Algorithm 2 reduced to the problem of finding $r \cdot s$ vectors from I^d. We can observe that the optimal value of s depends on the problem size, namely n and d. It is also different for each architecture. After these experiments we have evaluated the approximation of the parameter β using (16). Then applying (14) and (15) we have obtained theoretical approximation of optimal values of s for various n and d (see Table 1).

In the second set of our experiments we have compared the performance of Algorithm 2 for several test functions. For each target architecture we have used the estimated value of β. Equations (15) have been applied to find the optimal values of s and r for given problem sizes n and d. In [28] we used the set of the test functions proposed in [5] but we observed that it is sufficient to test only four of them:

1. Continuous: $f_1(v_1, v_2) = \exp\left(-c_1(v_1 - w_1) - c_2(v_2 - w_2)\right)$
2. NAG test: $f_2(v_1, \ldots, v_4) = \frac{4v_1v_3^2 \exp(2v_1v_3)}{(1+v_2+v_4)^2}$
3. Corner peak: $f_3(v_1, \ldots, v_d) = (1 + \sum_{i=1}^{d} c_i v_i)^{-(d+1)}$
4. Product peak: $f_4(v_1, \ldots, v_d) = \prod_{i=1}^{d} \frac{1}{(v_i - w_i)^2 + c_i^{-2}}$

where $\mathbf{c}, \mathbf{w} \in \mathbb{R}^d$ are fixed coefficients, because remaining ones can be characterized similarly.

Table 2 shows the execution times of Algorithm 1 (only on E5-2670) and Algorithm 2 for *Continuous* and *NAG test* functions. Similarly, Table 3 shows the performance of the algorithms for *Corner peak* and *Product peak* functions. Figures 4 and 5 show the speedup of Algorithm 2 over Algorithm 1 for all tested functions. It should be noticed that Algorithm 1 is completely useless on manycore architectures like GPUs and Intel MIC because it can utilize only a small fraction of their theoretical peak performance and its execution time would be much longer then on E5-2670. Figures 4 and 5 help us to realize the advantages of the use of manycore architectures and Algorithm 2 which is much more sophisticated than Algorithm 1.

Table 2 Execution time of Algorithm 1 (only on E5-2670) and Algorithm 2 for *Continuous* and *NAG test* functions

n	Continuous				NAG test			
	Alg.1 (E5)	E5	Phi	K40m	Alg.1 (E5)	E5	Phi	K40m
$1e+05$	0.002	0.002	0.023	0.004	0.001	0.002	0.026	0.004
$1e+06$	0.017	0.005	0.030	0.008	0.012	0.008	0.035	0.012
$1e+07$	0.162	0.025	0.078	0.032	0.118	0.035	0.097	0.044
$1e+08$	1.623	0.190	0.268	0.138	1.157	0.297	0.355	0.209
$1e+09$	16.921	2.483	2.031	0.765	11.060	6.136	3.798	1.601

Table 3 Execution time of Algorithm 1 (only on E5-2670) and Algorithm 2 for *Corner peak* and *Product peak* functions

d	n	Corner peak				Product peak			
		Alg.1 (E5)	E5	Phi	K40m	Alg.1 (E5)	E5	Phi	K40m
4	$1e+05$	0.008	0.013	0.034	0.006	0.005	0.003	0.024	0.006
	$1e+06$	0.075	0.024	0.040	0.023	0.047	0.007	0.033	0.023
	$1e+07$	0.746	0.076	0.111	0.102	0.475	0.036	0.071	0.125
	$1e+08$	7.459	0.634	0.479	0.580	5.086	0.418	0.244	0.889
	$1e+09$	74.586	6.417	4.752	4.966	46.596	6.454	2.929	8.203
16	$1e+05$	0.015	0.025	0.032	0.008	0.013	0.014	0.027	0.009
	$1e+06$	0.145	0.060	0.050	0.030	0.125	0.030	0.034	0.039
	$1e+07$	1.448	0.163	0.164	0.163	1.245	0.141	0.105	0.246
	$1e+08$	14.483	1.840	1.163	1.135	12.472	1.380	0.759	1.895
	$1e+09$	144.811	13.626	10.260	9.823	124.566	12.860	8.037	17.239
64	$1e+05$	0.043	0.034	0.042	0.016	0.041	0.034	0.036	0.019
	$1e+06$	0.428	0.084	0.089	0.075	0.404	0.089	0.056	0.092
	$1e+07$	4.274	0.479	0.345	0.597	4.036	0.439	0.230	0.729
	$1e+08$	42.729	5.340	3.529	4.452	40.356	5.395	3.019	6.620
	$1e+09$	427.951	50.657	30.761	38.499	404.294	70.072	27.361	65.353

We can observe that Algorithm 2 outperforms Algorithm 1 significantly for all architectures and test functions. However, the use of Algorithm 2 is much more profitable for bigger problem sizes and more complicated functions. In such cases, manycore architectures, namely Xeon Phi and K40m, outperform E5-2670. For *Corner peak* and *Product peak* functions Xeon Phi outperforms K40m for $n > 1e+06$ and the advantage is greater for $d \geq 16$. K40m works fine for smaller values of d when coalesced memory access can take place, i.e., when multiple memory accesses into a single transaction. [16,17]. Similarly, GPUs outperform Xeon Phi when integrand functions have calls to transcendental functions. In such a case plenty of cores can be utilized. Table 3 shows that the form of integrand functions has a great influence on the performance of the algorithm on individual architectures. In case of Xeon Phi the performance

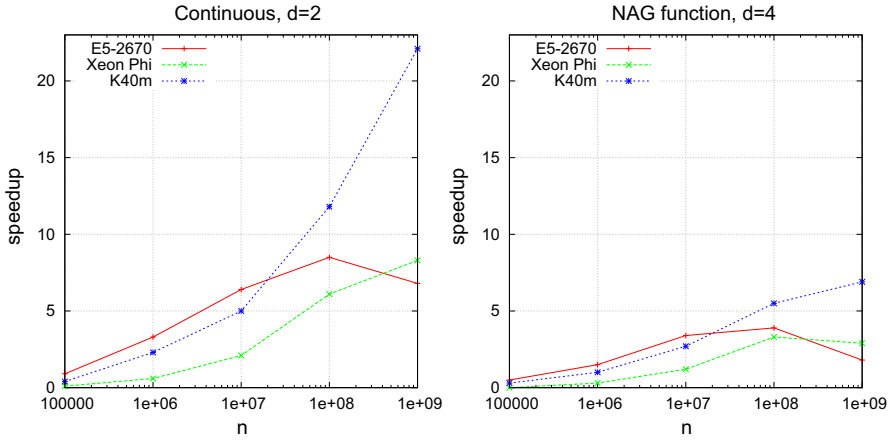

Fig. 4 Speedup of Algorithm 2 over Algorithm 1 for *Continuous* and *NAG test* functions

of Algorithm 2 for *Corner peak* and *Product peak* is nearly the same, while K40m requires almost twice as much time for *Product peak* than for *Corner peak*. Computations on GPUs are much more effective when computations performed by cores are rather simple, but the form of *Product peak* is more complicated and requires more memory references. When the integrand function is really simple, K40m achieves much better performance than Xeon Phi and E5-2670 (see Fig. 4). The results presented in Figs. 4 and 5 also confirm theoretical considerations regarding scalability (see Sect. 3). Indeed, Xeon Phi and K40m (i.e., manycore architectures) achieve better speedup for bigger problem sizes and when the integrand function is rather simple. Too many memory references that may appear in more complex integrand functions can limit speedup of the method.

It should be pointed out that the performance of Algorithm 2 on GPUs could be improved by using CUDA [17] or OpenCL [14] programming interfaces. Unfortunately, it would require much more effort than using OpenACC. On the other hand, the implementation for Intel architectures can be optimized by using more sophisticated techniques like *programming with intrinsics* for Intel Advanced Vector Extensions [9].

6 Conclusions

We have showed that the multidimensional Monte Carlo integration based on a new vectorized version of *linear congruential generator* can be easily and efficiently implemented on modern CPU, GPU and Intel MIC architectures including Intel Xeon E5-2670, Xeon Phi 7120P and NVIDIA K40m using high-level directive-based standards like OpenMP and OpenACC. The new version of LCG requires a limited amount of memory; thus, the number of generated pseudorandom points can be really huge. We have also shown how to use Hockney–Jesshope theoretical model of vector computations to find values of algorithm's parameters to minimize its execution time.

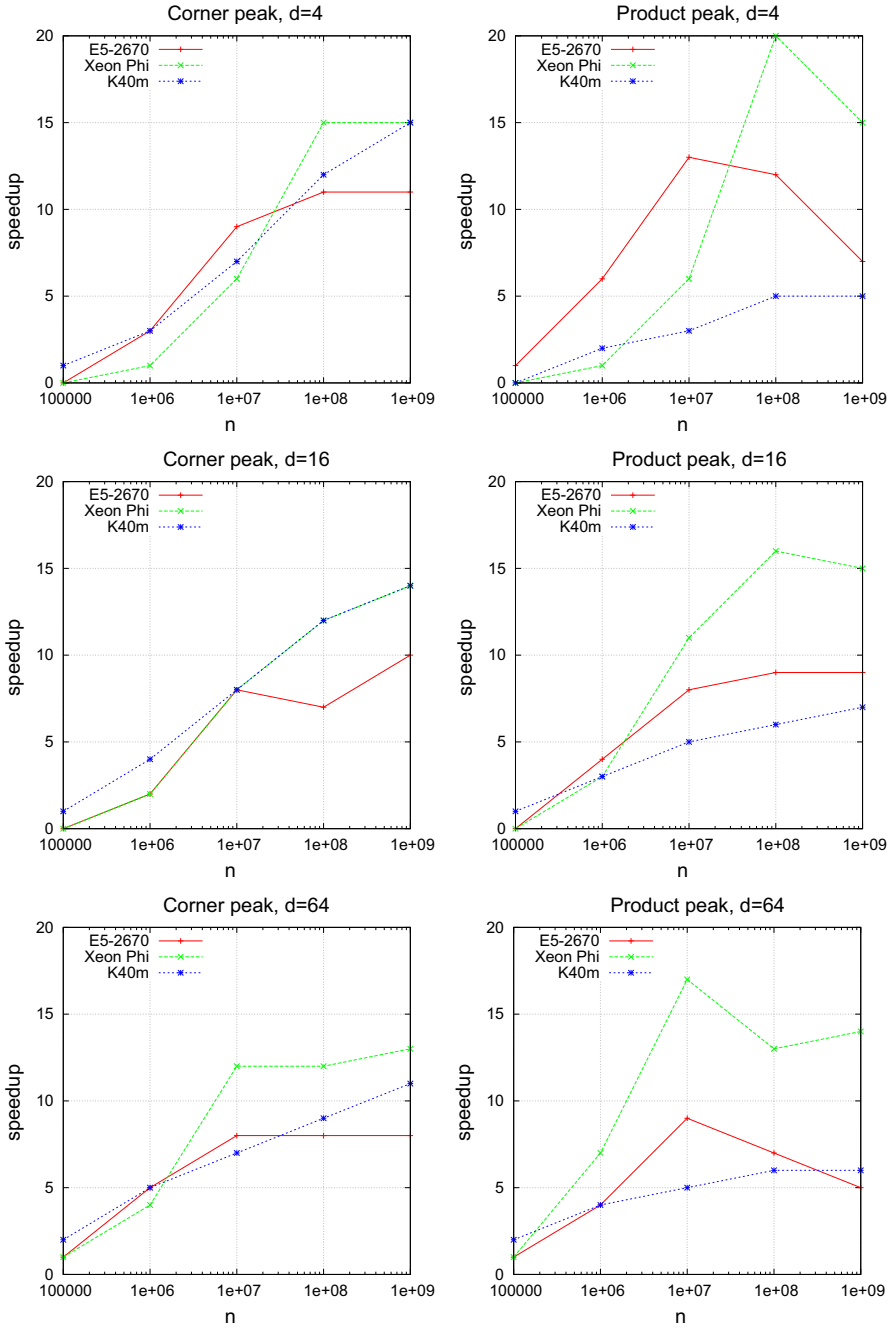

Fig. 5 Speedup of Algorithm 2 over Algorithm 1 for *Corner peak* and *Product peak* functions

Acknowledgements This work was partially supported by the National Centre for Research and Development under MICLAB Project POIG.02.03.00-24-093/13. The use of computer resources installed at Institute of Mathematics, Maria Curie-Skłodowska University, Lublin, is kindly acknowledged.

References

1. Bull JM, Freeman TL (1994) Parallel globally adaptive quadrature on the KSR-1. Adv Comput Math 2:357–373. https://doi.org/10.1007/BF02521604
2. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in OpenMP. Morgan Kaufmann Publishers, San Francisco
3. Chen C, Huang K, Lyuu Y (2015) Accelerating the least-square Monte Carlo method with parallel computing. The J Supercomput 71(9):3593–3608. https://doi.org/10.1007/s11227-015-1451-7
4. Dongarra J, Duff I, Sorensen D, Van der Vorst H (1991) Solving linear systems on vector and shared memory computers. SIAM, Philadelphia
5. Hahn T (2005) CUBA-a library for multidimensional integration. Comput Phys Commun 168:78–95
6. Hockney R, Jesshope C (1981) Parallel computers: architecture. Programming and algorithms. Adam Hilger Ltd., Bristol
7. RW Hockney (1985) $(r_\infty, n_{1/2}, s_{1/2})$ measurements on the 2-cpu CRAY X-MP. Parallel Comput 2(1):1–14. https://doi.org/10.1016/0167-8191(85)90014-6
8. Jeffers J, Reinders J (2013) Intel Xeon Phi coprocessor high-performance programming. Morgan Kaufman, Waltham
9. Jeffers J, Reinders J, Sodani A (2016) Intel Xeon Phi processor high-performance programming. Knights landing edition. Morgan Kaufman, Cambridge
10. Knuth DE (1981) The art of computer programming, volume II: seminumerical algorithms, 2nd edn. Addison-Wesley, Boston
11. Knuth DE (1999) MMIXware, a RISC computer for the third millennium, lecture notes in computer science, vol 1750. Springer, Berlin
12. Lautrup B (1971) An adaptive multi-dimensional integration procedure. In: Proceedings 2nd Colloquium Advanced Methods in Theoretical Physics, Marseille
13. Li Y, Schwiebert L, Hailat E, Mick JR, Potoff JJ (2016) Improving performance of GPU code using novel features of the NVIDIA Kepler architecture. Concurr Comput Pract Exp 28(13):3586–3605. https://doi.org/10.1002/cpe.3744
14. Munshi A (2009) The OpenCL Specification v. 1.0. Khronos OpenCL Working Group
15. Niederreiter H (1978) Quasi-Monte Carlo methods and pseudo-random numbers. Bull Am Math Soc 84:957–1041
16. NVIDIA (2015) CUDA C best practices guide. NVIDIA Corporation, available at http://www.nvidia.com/
17. NVIDIA Corporation (2015) CUDA programming guide. NVIDIA Corporation, available at http://www.nvidia.com/
18. OpenACC-standardorg (2015) The OpenACC application programming interface, v2.5. Tech. rep., OpenACC-Standard.org, http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf
19. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992) Numerical recipes in C, 2nd edn. Cambridge University Press, Cambridge
20. Pryor DV, Burns PJ (1989) Vectorized monte carlo molecular aerodynamics simulation of the rayleigh problem. The J Supercomput 3(4):305–330. https://doi.org/10.1007/BF00128168
21. Rahman R (2013) Intel Xeon Phi coprocessor architecture and tools: the guide for application developers. Apress, Berkely
22. Ripoll DR, Thomas SJ (1992) A parallel monte carlo search algorithm for the conformational analysis of polypeptides. The J Supercomput 6(2):163–185. https://doi.org/10.1007/BF00129777
23. Sabne A, Sakdhnagool P, Lee S, Vetter JS (2014) Evaluating performance portability of OpenACC. In: Languages and Compilers for Parallel Computing-27th International Workshop, LCPC 2014, Hillsboro,

OR, USA, September 15–17, 2014, Revised Selected Papers, pp 51–66, https://doi.org/10.1007/978-3-319-17473-0_4
24. Santos EE, Rickman JM, Muthukrishnan G, Feng S (2008) Efficient algorithms for parallelizing Monte Carlo simulations for 2d ising spin models. The J Supercomput 44(3):274–290. https://doi.org/10.1007/s11227-007-0163-z
25. Stpiczyński P (2011) 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, pp 465–470, https://fedcsis.org/proceedings/2011/pliks/148.pdf
26. Stpiczyński P, Szałkowski D, Potiopa J (2012) 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, pp 571–578, http://fedcsis.org/proceedings/2012/pliks/380.pdf
27. Supalov A, Semin A, Klemm M, Dahnken C (2014) Optimizing HPC applications with intel cluster tools. Apress, Berkely
28. Szałkowski D, Stpiczyński P (2014) Multidimensional Monte Carlo integration on clusters with hybrid GPU-accelerated nodes. In: Parallel Processing and Applied Mathematics, 10th International Conference, PPAM 2013, Warsaw, Poland, September 8–11, 2013, Revised Selected Papers, Part I, Springer, Lecture Notes in Computer Science, vol 8384, pp 603–612, https://doi.org/10.1007/978-3-642-55224-3_56
29. Szałkowski D, Stpiczyński P (2015) Using distributed memory parallel computers and GPU clusters for multidimensional Monte Carlo integration. Concurr Comput Pract Exp 27(4):923–936. https://doi.org/10.1002/cpe.3365