Highly optimized simulations on single- and multi-GPU systems of 3D Ising spin glass

M. Lulli[1],[*] M. Bernaschi[2], and G. Parisi[1,3]

[1] *Dipartimento di Fisica di "Sapienza",*

Università di Roma, 00185 Roma, Italy

[2] *Istituto per le Applicazioni del Calcolo, IAC-CNR, Rome, Italy and*

[3] *INFN, Sezione di Roma I, 00185 Roma, Italy*

Abstract

We present a highly optimized implementation of a Monte Carlo (MC) simulator for the three-dimensional Ising spin-glass model with bimodal disorder, *i.e.*, the 3D Edwards-Anderson model running on CUDA enabled GPUs. Multi-GPU systems exchange data by means of the Message Passing Interface (MPI). The chosen MC dynamics is the classic Metropolis one, which is purely dissipative, since the aim was the study of the critical off-equilibrium relaxation of the system. We focused on the following issues: *i*) the implementation of efficient access patterns for nearest neighbours in a cubic stencil and for lagged-Fibonacci-like pseudo-Random Numbers Generators (PRNGs); *ii*) a novel implementation of the asynchronous multispin-coding Metropolis MC step allowing to store one spin per bit and *iii*) a multi-GPU version based on a combination of MPI and CUDA streams. We highlight how cubic stencils and PRNGs are two subjects of very general interest because of their widespread use in many simulation codes. Our code best performances ~ 3 and ~ 5 psFlip on a GTX Titan with our implementations of the MINSTD and MT19937 respectively.

PACS numbers:

---
[*]Electronic address: matteo.lulli@gmail.com

I. INTRODUCTION

The three-dimensional Ising spin glass is a statistical mechanics model defined on a cubic lattice by the Hamiltonian

$$H = -\sum_{\langle ik \rangle} J_{ik}\, \sigma_i \sigma_k, \tag{1}$$

where the $\sigma_i \in \{-1, +1\}$ are the so-called spin variables, the $J_{ik} \in \{-1, +1\}$ are the coupling constants (representing *quenched* disorder) which are randomly drawn according to a given probability distribution $P(J_{ik})$ and the sum $\sum_{\langle ik \rangle}$ is restricted to nearest neighbouring spins. Since we deal with bimodal disorder the probability distribution reads

$$P(J_{ik}) = \frac{1}{2} \left[\delta_K(J_{ik} - 1) + \delta_K(J_{ik} + 1) \right], \tag{2}$$

where $\delta_K(a - b) = \delta_{ab}$ stands for the Kroneker delta. Such a model describes, in three dimensions, a disordered and frustrated magnetic system showing a glassy dynamics below a finite critical temperature $T_c = 1.1019(29)$ [1]. Because of a very high dynamic critical exponent $z = 6.86(16)$ [2] for the MC Metropolis dynamics, this model has represented a long standing challenge for numerical simulations. For almost thirty years special purpose machines have been employed [3–6] in order to get equilibrium measures for always larger systems. Large systems are needed because of the severe finite-size corrections to scaling [1, 7]. Moreover, new kinds of dynamics have been developed in order to reach equilibrium as fast as possible: Parallel Tempering has proved to be the best choice so far [8].

However, the hegemony of special purpose hardware for this class of problems might be about to end because of the steadily increasing use of (multi) GPU devices enabling us to reach computational horsepower exceeding by orders of magnitude those of common CPUs.

Since the very early use of CUDA, and even before [9], it has been understood that MC simulations of Ising spin systems would have enjoyed benefits from the use of GPUs. This is not surprising: for even cubic lattice sizes $L = 2n$ the system can be simply partitioned according to a checkerboard scheme into two coloured subsystems, which we will refer to as *reds* and *blues*, which can be updated separately since nearest neighbours of one colour belong to the other, *i.e.*, a red spin has blue nearest neighbours only. Hence, the problem exposes an intrinsic parallelism which perfectly suits the GPU architecture: the update of each spin of a given color does not require any coordination with the update process of other

spins of the same color so that the large amount of computing threads needed for the best use of the GPU can be programmed to update concurrently independent spins of the system.

Using the Metropolis dynamics, the update of the entire system is performed via two separate kernels, one for each colour. This is an easy way to enforce the independence of the update of the two subsets, a necessary condition for a correct implementation of the Markov chain. Of course, this kind of update does not ensure *detailed balance* but *balance* holds nonetheless, which is enough to properly carry on a MC simulation of such systems where the system probability distribution converges to the equilibrium Gibbs distribution [10, 11]

$$P(\{\sigma_i\}, \beta) = \frac{e^{-\beta H[\{\sigma_i\}]}}{\sum_{\{\sigma_i\}} e^{-\beta H[\{\sigma_i\}]}}. \tag{3}$$

Now, we will review the previous works on spin systems for GPUs, taking into account also different models and dimensionalities, analyzing them according to:

- memory allocation strategies and spins-threads mappings;

- the kind and implementation of PRNGs;

- multi-GPU implementation techniques.

We can classify the preceding works on spin systems according to the allocation and the memory access strategies starting from the lowest level (Global Memory) up to the highest level (Shared Memory and registers) of the memory hierarchy:

1. **Global Memory allocation**. Two different strategies are mainly used for the memory allocation of spins in the GPU Global Memory: a first one keeping a mixed scheme where one array is allocated containing both colours in a cubic lattice topology [12–17]; a second one allocating two separate buffers for the two colours breaking the cubic lattice topology [9, 18–21].

   Though the first strategy seems to be more natural one has to take into account that memory transactions are served from the L2 cache in blocks of 128 bytes so that for each transaction one loads both the to-be-changed spins and the nearest neighbours which stay unchanged during the kernel execution. Different threads will update different spins sharing some neighbours thus rendering the access pattern highly non-trivial. As a matter of fact, many of the works adopting the first strategy use the

Shared Memory to improve the locality of such mixed access to the coloured spins [13, 14, 16, 17].

In the second case there are several benefits but also drawbacks. On one hand it is possible to achieve good memory loading performances since many threads would look for the same neighbouring spin allowing for a higher second hit probability in L1/L2 caches. However, one has to deal with an algebraically demanding access pattern due to the loss of the cubic topology: it is necessary to take into account the *parity* of the lattice site in order to correctly determine the right and left neighbours. Usually this strategy does not require the use of Shared Memory as reported in [18–21].

Other considerations are in order. Being the two colours allocated in two different arrays it is possible to bind each buffer to a texture in order to load the neighbouring spins through the dedicated texture unit of each Streaming Multi-processor (SM) with a separated cache different from the L1/Shared. This choice offers a two-fold advantage: on one side the slowing down due to occasional non-coalescence of loads for nearest neighbours is softened because texture fetches work on a memory locality principle, on the other side the dedicated texture hardware is in charge of the computation of physical memory addresses rather than CUDA cores.

2. **Spins arrangement in Global Memory**. For both allocation strategies it is still possible to choose several spins arrangements. Such a choice aims at maximizing the loading efficiency from the Global Memory, *i.e.* reducing the number of non-coalesced loads.

   In case of a single buffer for both colours two different strategies have been proposed in [16] and [17]. In [16] the authors divide the cubic lattice in sublattices linearly organized in memory in a "snake" fashion so that every block taking charge of one sublattice could load more efficiently the spins to the Shared Memory. As for [17] the authors studied a so-called *shuffled* scheme where spins coming from different replicas where mixed saving memory transactions. However they found that such a strategy performs worse than the so-called *unified* one where each array contains spins belonging to one replica.

   In the case of a separate allocation of colours, the authors of [18] proposed, for the two-dimensional Potts model, a coordinate transformation of the lattice that leads to

have three of the four nearest neighbours lying sequentially in the array.

3. **Spins-threads correspondence**. The mapping between spins and threads can also be done in different ways.

Some of the works adopting the unified allocation scheme resort to (per-block) Shared Memory [13, 14, 17]. In one of the most commonly used mappings each thread-block evaluates the MC move on a sublattice [13–17] which usually is a square in two dimensions or a cubic sublattice in three. Clearly, with this strategy one needs to look for neighbours in the boundaries which will also be retrieved by neighbouring blocks thus leading to a duplication of data served by memory transactions. The most frequently used technique, in this case, is loading in Shared Memory the neighbours and, if needed, the couplings. However, this might represent a serious limitation since the number of thread blocks running on a single Streaming Multiprocessor (SM) depends on the amount of Shared Memory needed by each of them. In the case of models with complex degrees of freedom only few blocks can be run concurrently thus leading to underuse the SM.

A rather different, but apparently less effective, approach has been tried in [12] where thread-blocks were associated to stripes of the cubic lattice of size $L \times 2 \times 2$. A thread-block is associated to each region and each thread updates 4 spins in the three-dimensional case.

As for the separated scheme there are no particular restrictions on the dimensionality of the thread blocks which can also be taken as one-dimensional. Hence, no specific correspondence between blocks and lattice portions has to be considered resulting in a more tunable and flexible scheme [19–21]. Indeed, such a choice allows to decrease memory transfers redundancy. Moreover, as shown in [19] and verified in the present work, it turns out that nearest neighbours values are loaded in a more efficient way directly from global memory making use of texture fetches, *i.e.* texture cache and hardware.

We continue now with the choice of the PRNG which is one of the most important aspects of a MC simulation. The reliability of the estimates depends on the quality of the sequence generated by the chosen PRNG. As an example, it is well known that the use of one

single Linear Congruential PRNG with a period $p = 2^k$ in equilibrium MC for the 2D Ising model leads to systematic discrepancies on lattice sizes $L = 2^\ell$ due to resonance phenomena between the size of the system and the systematic long-range correlations which affect Congruential PRNGs. It is then important to provide fast implementations for reliable PRNGs. Nonetheless, many of the previous works on MC simulations of spin systems on GPU used such PRNGs [9, 12, 14, 15, 19–21], mainly for benchmarking reasons. The main reason is that the state of these PRNGs is limited to one integer value making them the best choice in terms of speed but a questionable choice as for the quality of the produced numbers. However, it is interesting to notice that in [14] compatible results with theoretical predictions for the two-dimensional Ising model were reported. In the most straightforward implementation of Linear Congruential PRNGs, each thread accesses its own memory location storing the one-valued state of the generator so that one deals with a battery of generators rather than a single one used to update the entire lattice. Little is known about the behaviour of such parallelized implementations of Linear Congruential PRNGs and it would be interesting to study them carefully.

In [16, 18] the authors used the so-called multiply-with-carry PRNG which consists in a modified Linear Congruential PRNG where the result of the module operation is used for the successive update hence requiring two integers to store the state.

In [17] the cuRand implementation of the XORWOW has been used. This PRNG consists in a XOR-Shift summed to a Weyl generator.

There are a few other works using the so-called lagged-Fibonacci PRNG [22, 23]: these generators use a state of a certain length from which two 'lagged' entries are read and combined, usually summed, giving the random number and updating at the same time the state. Usually, this kind of generators have very long periods, much longer with respect to Linear Congruential PRNGs. As we are mainly interested in the memory access scheme for the GPU implementation we can label as lagged-Fibonacci-like all those PRNGs sharing a scattered read pattern of the state. Since there are lags between the reads of the state, a certain amount of random numbers can be produced in parallel by different threads [24, 25] using Shared Memory to store the state which will be used in a thread block. One of the most popular generators of this kind is the Mersenne Twister MT19937 [26] which has a very long period $p = 2^{19937} - 1$. However, since its state needs at least 624 entries, a Shared Memory implementation would be too memory-consuming thus strongly limiting the SM

occupancy. Hence in [27] the authors chose to use the so called Warp generator [24] which has been written along the same lines of MT19937.

To our knowledge there is only one work [23] using the so-called Parisi-Rapuano generator [28] which basically consists in a lagged-Fibonacci PRNG. However no implementation details are given. Indeed, in [25] it has been observed that such a PRNG is not well suited for a Shared Memory-based GPU implementation because of the lags values.

Finally, as for multi-GPU implementations exposing strong scaling, we are only aware of [13, 20, 21]. Other works present weak scaling [17, 23] although in [23] communication between different nodes is needed because of the adopted Parallel Tempering implementation. However, the requirement of strong scaling depends on the physical features of the simulated systems one is interested in.

The outlook of the paper is the following. In section 2 we describe a new cubic-stencil access pattern which uses a separated allocation scheme while keeping a cubic lattice topology and avoiding the use of Shared Memory. This result is obtained via a suitable spin arrangement which follows from a coordinate transformation akin to [18]. Texture fetches are used to load both nearest neighbours and couplings. Different spins-threads mappings have been tested and results are compared to those obtained with an access pattern proposed in [19] handling only $L = 2^\ell$ sizes and its generalization to $L = 2n$. In section 3 we present a new implementation of the Parisi-Rapuano PRNG which avoids the Shared Memory. In order to show the advantages of this approach we will propose a GPU implementation of a per-thread version of the MT19937, the first one in our knowledge, comparing its performances with the device API cuRand MTPG32 (a GPU-tailored variant of the Mersenne Twister) and a very recent version of the host API cuRand MT19937. At the same time we propose a simple benchmarking test for the device API whereas for the host API we comply with the benchmark proposed in [29]. In section 4 we describe a new asynchronous multispin-coding technique and its implementation. We then show the results obtained on a variety of different GPUs and analyze them. In section 5 we present the strongly scaling multi-GPU version of the code outlining its features showing the results obtained on the Piz Daint [30] supercomputer. In section 6 we draw conclusions.
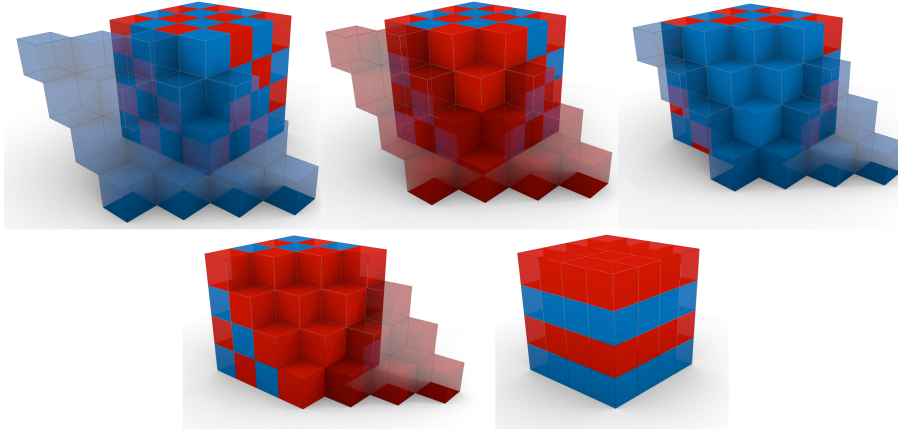
FIG. 1: A depiction of the slicing procedure. Lighter cells are the periodic ones.

II.   CUBIC STENCIL

With "Cubic Stencil" we refer to a set comprising one vertex of a cubic lattice together with its six nearest neighbouring vertices and edges. It represents the set of data needed to perform the update of a spin. It is also the fundamental data element of many other algorithms based on the cubic lattice discretization, *e.g.* for the solution of partial differential equations.

We now discuss the features of a new cubic stencil access pattern which we will refer to as **sliced**. We store red and blue spins in two separate arrays of Global Memory bound to two different textures. The novelty of the approach is in the spin arrangement. Here, we analyze the three-dimensional case, however the approach naturally extends to lower, *i.e.* two-dimensional, and higher dimensional cases. In three dimensions, under the assumption of having periodic boundary conditions, there exist a way of separating the colours while keeping the cubic lattice topology: let us consider the cubic lattice starting from the origin of a Cartesian reference frame where the coordinates take on integer values, hence $\vec{x} \in \mathbb{Z}^3$; vertices belonging to planes orthogonal to the direction $\vec{n} = (1, -1, 1)$ are all one-coloured. That is the reason why we call this scheme *sliced*. With periodic boundary conditions, vertices belonging to one slice will have all nearest neighbours either in the upper or in the lower slice. Such a slicing procedure is depicted in Fig. 1, whereas in Fig. 2 the transformation from the classic spin arrangement is shown. Hence, one starts from a three-dimensional checkerboard and ends up with a cubic lattice where each horizontal plane is one-coloured.
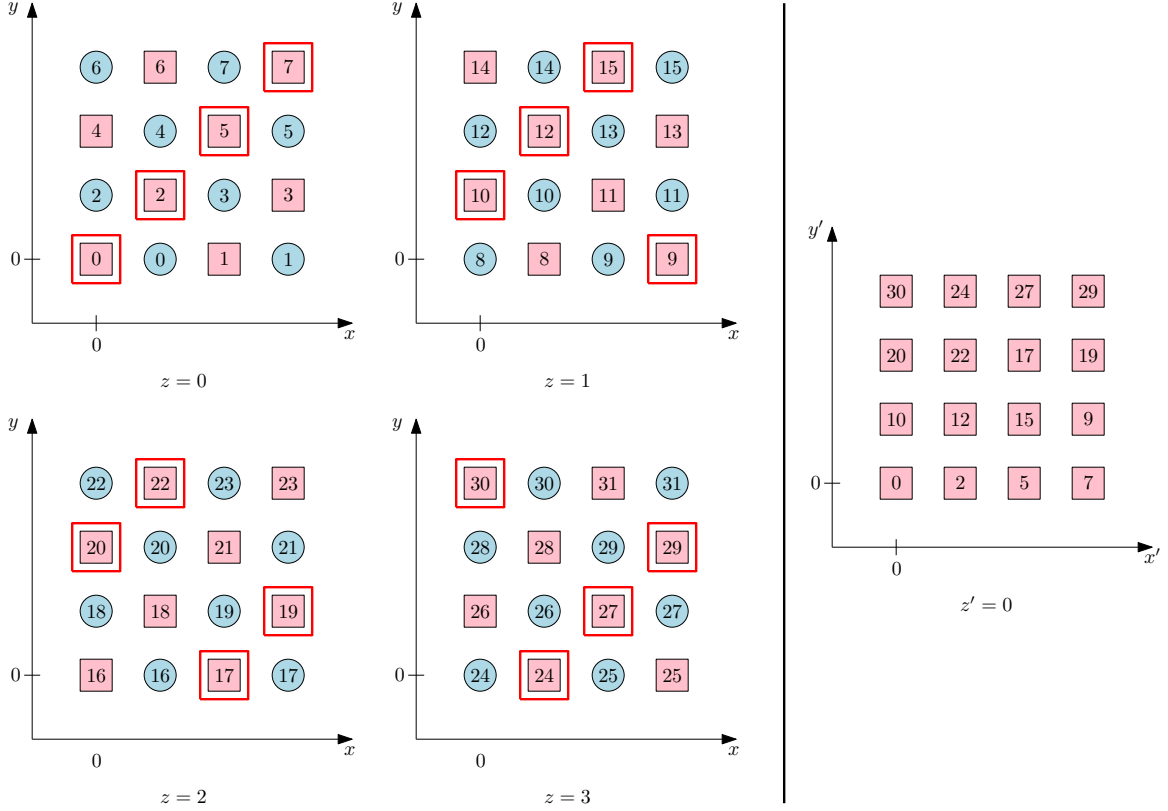
8

FIG. 2: Mapping from the separated allocation checkerboard scheme to the sliced one for a lattice with $L = 4$. All four planes of the real lattice are involved in the definition of the first plane. The procedure is iterated starting from the $z = 1$ plane taking the blue diagonal: a blue slice is obtained for $z' = 1$ and so on.

It is easy to write down the transformation and its inverse given that the vertices coordinates read $\vec{x} = (x, y, z)$ and the transformed coordinates read $\vec{x'} = (x', y', z')$

$$\begin{cases} x' = x \\ y' = y - x \\ z' = x - y + z \end{cases} \qquad \begin{cases} x = x' \\ y = y' + x' \\ z = z' + y' \end{cases} \qquad (4)$$

recognizing in the equation for z' the expression of a plane orthogonal to the direction $\vec{n} = (1, -1, 1)$. The generalization to any number of dimensions for the vector $\vec{n}$ is simply given by $n_i = (-1)^{i+1}$ so that in two dimensions one-coloured vertices lie on lines orthogonal to $\vec{n} = (1, -1)$, and in four dimensions they lie on hyperplanes orthogonal to $\vec{n} = (1, -1, 1, -1)$ and so on. Hence, as long as one deals with regular (hyper)cubic lattices the scheme is completely general.
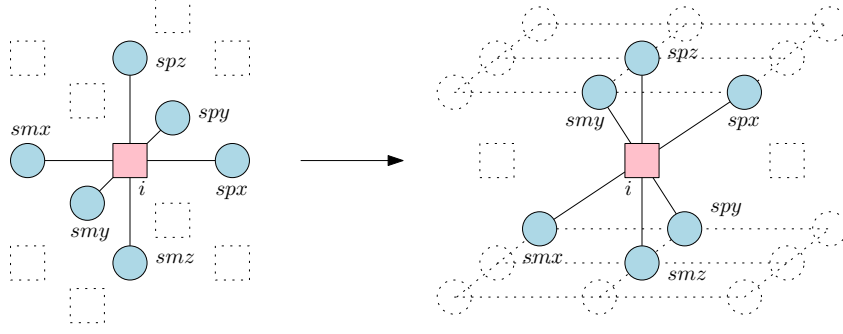
9

FIG. 3: Cubic stencil for the real lattice (left) and for the transformed lattice (right).

The transformed coordinates of nearest neighbours are:

$$\vec{x}_{spz} = (x, y, z+1) \quad \rightarrow \quad \vec{x'}_{spz} = (x', y', z'+1)$$

$$\vec{x}_{smy} = (x, y-1, z) \quad \rightarrow \quad \vec{x'}_{smy} = (x', y'-1, z'+1)$$

$$\vec{x}_{spx} = (x+1, y, z) \quad \rightarrow \quad \vec{x'}_{spx} = (x'+1, y'-1, z'+1)$$

(5)

$$\vec{x}_{smz} = (x, y, z-1) \quad \rightarrow \quad \vec{x'}_{smz} = (x', y', z'-1)$$

$$\vec{x}_{spy} = (x, y+1, z) \quad \rightarrow \quad \vec{x'}_{spy} = (x', y'+1, z'-1)$$

$$\vec{x}_{smx} = (x-1, y, z) \quad \rightarrow \quad \vec{x'}_{smx} = (x'-1, y'+1, z'-1)$$

where the labels $spz, \ldots$ are self-explaining. The order of the new coordinates makes apparent that for the calculations of, say, $\vec{x'}_{spz}$ it is possible to reuse $\vec{x'}_{smy}$ and $\vec{x'}_{spx}$:

$$\vec{x'}_{spx} = \hat{i}'(x'+1) + \vec{x'}_{smy} = \hat{i}'(x'+1) + \hat{j}'(y'-1) + \vec{x'}_{spz}, \tag{6}$$

where $\hat{i}'$, $\hat{j}'$ and $\hat{k}'$ are the unit basis vector for the x', y' and z' direction respectively. This feature gives some advantage in terms of calculations for the memory accesses, and it does not hold for the standard expressions. Thus, the cubic stencil layout in both set of coordinates looks as in Fig. 3, and periodic boundary conditions apply also to the new coordinates.

Memory transactions for bulk spins are coalesced with some per-warp redundancy for smy/spx and spy/smx which is completely handled by the memory hierarchy. The new layout naturally shows a two-dimensional locality of data. Indeed, such a remapping is in the same spirit as the one proposed in [18], with the difference that being this approach geometric it can be extended to other dimensionalities as shown above.

After having explained the memory arrangement we finally explain the spins-threads mappings. Some general remarks are in order:

- we simulate four replicas at once, systems with the same quenched disorder but different initial conditions and evolutions, which are stored in different arrays. Such a choice is a common practice [1]. However, the extension to an arbitrary number of replicas is trivial, requiring to handle a new stride in the kernels.

- colours and couplings are bound to textures in order to delegate addresses calculations to the texture hardware rather than to CUDA cores. Indeed one colour is constant while updating the other.

- for the sliced scheme, since the arrays are separated, one does not really need to consider z' as running from 0 to $L-1$, but rather from 0 to $L/2 - 1$. By assigning the same z' label to pairs of differently coloured $x' - y'$ planes and starting from the bottom with red vertices, the blue spz of a red vertex i has its same index, $i.e.$ $spz = i$, whereas the blue vertex j will have its bottom red spin at $smz = j$. This scheme further simplifies calculations.

- Currently, physically interesting behaviour of the EA3D model can be studied only for relatively small sizes, hence we should try to saturate the GPU resources also for small-size lattices. The easiest way to achieve this goal is to simulate different coded disorder realizations numbered by k. Thus, the stride separating in the spins arrays different coded samples is $L^3/2 = V/2$.

  As far as we know only in [17, 23] such a technique has been adopted and indeed it is possible to sustain almost stable performances while varying the linear size L. We reserve the y, z block grid dimension as disorder index, $i.e.$ different realizations of the couplings, which seems a reasonable choice since `gridDim.y,z` < 65536.

- couplings are indexed as if they were red vertices and they are allocated in six different arrays `Jpx, Jpy, Jpz, Jmx, Jmy, Jmz`. This choice introduces an asymmetry in the kernels which can easily be fixed by allocating a copy of the couplings suitably transformed in order to be indexed as blue vertices. However, we do not show the results since the difference in the performances of the two updating kernels is negligible.

11

We implemented two different ways of mapping the vertices index to the threads index:

1. a one-dimensional mapping that associates s vertices to a single thread. The value of s can be tuned in order to find the best performance. Though, one needs to compute two divisions and two modulus operations in order to calculate nearest neighbours indices.

2. a multi-dimensional mapping exploiting the grid algebra provided by the GPU which allows to avoid divisions and modulus operations at the price of a more rigid choice for the total number of threads. The corresponding Kernels are tagged as *Grid*.

We tested the new access scheme comparing it with an implementation of the classic checkerboard spins arrangement, which we will refer to as **standard**, and with another scheme [19] using mainly bitwise operations, which however works only in the case $L = 2^\ell$. We will refer to this last implementation as **bitwise**.

We also wrote the Grid version of the standard scheme so that we end up with five different kinds of kernels: bitwise, standard, standard-Grid, sliced and sliced-Grid.

III.   PSEUDO-RANDOM NUMBERS GENERATORS

We chose to implement as a baseline the so-called Lehmer-Park-Miller MINSTD Linear Congruential PRNG which is defined as

$$R_{n+1} = (16807 \, R_n) \mathrm{mod}(2^{31} - 1), \tag{7}$$

Its period is a prime number, more precisely a Mersenne prime $M_{31} = 2^{31} - 1$. This generator can be used for the coupling values J_{ik} and it is also a reasonable choice for the critical off-equilibrium relaxation dynamics under the hypothesis of a number of instances no larger than the period.

One difficulty comes with the implementation of the module which cannot be carried out by hardware truncation, so that we need to directly handle the overflow due to the multiplication and then take the module. This can be done by means of a swap 64-bit variable or by means of 32-bit variables only as proposed by Carta in [31, 32]. The latter solution does not require the module operation. We followed the implementation proposed

in [32] but we substituted the conditional statements with bitwise operations to avoid warp branchings.

However, since we plan to extend this MC implementation to the equilibrium regime we also developed a GPU version of the Parisi-Rapuano PRNG [28] which is mostly used in the spin-glass community. It is a lagged-Fibonacci-like PRNG with a minimal state of 62 words. One instance of the generator reads

```
ira[i] = ira[i - 24] + ira[i - 55];
R = ira[i]^ira[i - 61];
```

where `ira` denotes the state array and `R` is the new random number. A common approach [25, 33] consists in exploiting the lags and let the threads in a block share one or more states which can be concurrently updated storing them in Shared Memory. However lags as those of the Parisi-Rapuano PRNG are not well-suited for this scheme [25].

Hence, we propose a new simple alternative: allocate an array of $N_{threads} \times N_{state}$ entries and let each thread access it with its own global grid index and load the lagged entries just using a stride, *i.e.* the number of threads. Thus, defining `d_threads` as the number of threads and `globalId` as the thread global grid index, a sketch of the kernel implementation simply reads

```
swap = ira[(i - 24)*d_threads + globalId]
         + ira[(i - 55)*d_threads + globalId];
  R = swap^ira[(i - 61)*d_threads + blobalId];
          ira[i*d_threads + globalId] = swap;
```

although in an actual implementation one has to take into account the periodic conditions for the access to the state[46].

In order to show the validity of the new scheme we chose to implement the widely known Mersenne Twister MT19937 and compare its performance to that of the cuRand MTGP which is a modified version of the Mersenne Twister. For the host API comparison we use the criterion proposed in [29]. Nonetheless, we propose as standard benchmark for a PRNG its kernel version counting the fraction of odd numbers (just as the example reported in the cuRand manual [34]). Such a benchmark should be more suitable for kernel-use PRNGs.

Results are reported in Table I for the PRAND test, and in Table II for the device API test. Tests were run on GTX 680, GTX Titan, Tesla M2090 and Tesla K20x GPUs. Looking

	Tesla M2090			Tesla K20X			GTX Titan		
PRNG	t_{INIT}(s)	t_{GEN}(s)	t_{TOT}(s)	t_{INIT}(s)	t_{GEN}(s)	t_{TOT}(s)	t_{INIT}(s)	t_{GEN}(s)	t_{TOT}(s)
cuRand MTGP32	0.09	12.34	12.43	0.12	13.46	13.58	0.21	10.11	10.32
cuRand XORWOW	0.01	2.91	2.92	0.01	2.90	2.91	0.01	2.31	2.32
cuRand MT19937	0	0	0	0	0	0	0.01	3.23	3.24
MT19937	3.86	6.40	10.26	4.63	6.12	10.75	3.92	4.66	8.58
Parisi-Rapuano	0.40	8.17	8.57	0.45	5.87	6.32	0.41	4.18	4.59
MINSTD	0.01	1.72	1.73	0.01	1.34	1.35	0.01	1.13	1.14

TABLE I: PRAND benchmark [29] results using cuRand host API. The task consists in filling an array of 2^{29} single-precision floating point variables. In the upper half of the Table, cuRand library results are reported whereas in the lower half those of our implementations. Two different measures are reported: t_{INIT} is the time needed to initialize the PRNG; t_{GEN} is the generation time. For the M2090 ECC is off, while for K20x ECC is on.

at Table I, where we report the execution times for filling an array of 2^{29} single-precision floating point variables, we see that our implementation of MT19937 runs roughly twice as fast as the cuRAND MTGP32 implementation. We could only test the most recent host API cuRAND implementation of the MT19937 on the GTX Titan and not on the K20x (the M2090 is ruled out being too old), and our implementation performs 44% slower than cuRAND. The large t_{INIT} values for our implementation are due to the fact that the seed are read from the system random pool, slowing down the process. It is clearly possible to reduce those times implementing some initialization algorithms as those proposed in [35]. In Table II the metric is changed to the number of PRNG instances per second. The trends are qualitatively the same although our implementation of the MT19937 on the GTX Titan runs almost three times faster than the cuRAND MTGP32.

However, such benchmarks only give an indication of the speed of different PRNGs. As we will see one should always compare different PRNGs in a real-life application.

PRNG	M2090	K20X	GTX Titan	GTX 680
cuRand MTGP32	$4.5 \cdot 10^9$	$3.9 \cdot 10^9$	$5.2 \cdot 10^9$	$5.1 \cdot 10^9$
cuRand XORWOW	$2.9 \cdot 10^{10}$	$7.6 \cdot 10^{10}$	$10.7 \cdot 10^{10}$	$6.1 \cdot 10^{10}$
MT19937	$10.1 \cdot 10^9$	$10.7 \cdot 10^9$	$14.1 \cdot 10^9$	$9.6 \cdot 10^9$
Parisi-Rapuano	$9.4 \cdot 10^9$	$12.5 \cdot 10^9$	$16.8 \cdot 10^9$	$8.3 \cdot 10^9$
MINSTD	$4.1 \cdot 10^{10}$	$7.6 \cdot 10^{10}$	$8.9 \cdot 10^{10}$	$6.7 \cdot 10^{10}$

TABLE II: Device API test. Number of instances per second. The launch configuration is the following: 64 blocks of 256 threads, each thread producing 2^{15} instances, repeated 10 times. For the M2090 ECC is off, while for K20x ECC is on.

IV. ASYNCHRONOUS MULTISPIN-CODING

Multispin-coding techniques are rooted in lattice gauge theory simulations [36]. They have been employed later in Ising models simulations [37–40]. The search for a close packing of data was motivated by the limited memory resources of that time, and by the intrinsic bit-level parallelism which can be obtained through bitwise operations. Indeed, since the quantities involved in the simulations are two-valued, *i.e.* $\sigma_i \in \{-1, +1\}$ and $J_{ik} \in \{-1, +1\}$, the optimal solution is to store couplings and spins in single bits rather than use a single byte, *e.g.* using a `char`.

Multispin-coding comes in two different flavours:

- synchronous multispin-coding (SMSC) consisting in storing in one word spins belonging to one single system, usually aligned along one specific direction. This allows to get faster simulations in terms of wall-clock time compared to a simple one-variable-one-spin setting. Indeed, such a technique is used in the Janus supercomputer [5] for reaching thermal equilibrium. Clearly, the update of each bit-spin requires one instance of the PRNG;

- asynchronous multispin-coding (AMSC) consists in storing spins belonging to different systems, located at the same vertex, in the same word. The total wall-clock time does not decrease, but it is possible to update all spins contained in a word with only one instance of the PRNG at the cost of the introduction of a certain amount of correlation, which can be taken care of easily.

We chose to implement the AMSC because we were interested in the off-equilibrium critical relaxation regime, hence being able to simulate a large number of samples is preferable over obtaining a long simulation time. The AMSC for spin systems was clearly explained in [40] where each system was considered to be at a different temperature. We are aware of AMSC implementations on GPU: [13] for the 2D Ising model and [17] for the EA3D model with external field. In particular in [17] the proposed AMSC techinque stores in one word spins of the same system, *i.e.* with the same couplings, which are evolved at different temperatures. This scheme has been adopted for implementing the PT dynamics. Transition probabilities are stored in a look-up table indexed by the energy difference ΔE of the proposed flip and the spin direction (with respect to an external magnetic field). Hence, the swap of two temperature-replicas simply requires to swap two lines in the look-up table. However, in order to speed up the access to the look-up table, some space in the spin words is reserved so that not all bits of a word codify for a spin. We will see that for non-PT dynamics this represents a bottle-neck for memory use efficiency. Again, each spin update is served by one PRNG instance.

As we anticipated, we associate to each spin a different disorder realization thus only one PRNG instance is needed for all spins contained in a word. Considering the contribution to the Hamiltonian due to a single cubic stencil, it is clear that the possible energy differences after a proposed spin flip on σ_a are

$$\Delta E = H[\{\sigma_{i \neq a}, -\sigma_a\}] - H[\{\sigma_{i \neq a}, \sigma_a\}] = -12, -8, -4, 0, 4, 8, 12. \tag{8}$$

The Metropolis dynamics is defined by the acceptance probability

$$P_{\text{flip}}(\Delta E) = \begin{cases} 1, & \Delta E \leq 0 \\ e^{-\beta \Delta E}, & \Delta E > 0 \end{cases} \tag{9}$$

where $\beta = T^{-1}$ is the inverse temperature. The value of $P_{\text{flip}}(\Delta E > 0)$ has to be compared to a flat-distributed random number $r \in [0, 1]$ so that if $r < P_{\text{flip}}(\Delta E > 0)$ the proposed flip is accepted otherwise it is rejected. However, since PRNGs are defined for integers, one does not really need to use a normalized r. The most direct way is to multiply the transition probability for the value of the biggest random number R_{max} and compare it with the PRNG instance R, *i.e.* $R \lessgtr R_{max} \exp(-\beta \Delta E)$. We label the non-trivial normalized transition probabilities as $R_{max} \exp(-\beta \Delta E) = \texttt{EXP12}, \texttt{EXP8}, \texttt{EXP4}$. We employ the following

mapping of spins and couplings to bits

$$J_{ik} = -1 \;\rightarrow\; \mathtt{J}_{ik}\mathtt{=1}, \qquad \sigma_i = -1 \;\rightarrow\; \mathtt{s}_i\mathtt{=0},$$
$$J_{ik} = +1 \;\rightarrow\; \mathtt{J}_{ik}\mathtt{=0}, \qquad \sigma_i = +1 \;\rightarrow\; \mathtt{s}_i\mathtt{=1}. \tag{10}$$

The value of the interaction energy with one of the nearest neighbours is then converted for each bit as

$$-J_{ik}\,\sigma_i\,\sigma_k = -1 \;\rightarrow\; \mathtt{e}_{ik} = \mathtt{J}_{ik}\mathtt{\hat{}s}_i\mathtt{\hat{}s}_k = 0,$$
$$-J_{ik}\,\sigma_i\,\sigma_k = +1 \;\rightarrow\; \mathtt{e}_{ik} = \mathtt{J}_{ik}\mathtt{\hat{}s}_i\mathtt{\hat{}s}_k = 1. \tag{11}$$

If we sum the six energy variables per stencil $\mathtt{e}_{ik}$ we obtain a three-bit result

$$\sum_k \mathtt{e}_{ik} = (\mathtt{sum2,\ sum1,\ sum0}) = 2^2 \times \mathtt{sum2} + 2 \times \mathtt{sum1} + \mathtt{sum0}, \tag{12}$$

which directly maps to the seven possible values of ΔE, since flipping the spin leads to flip the partial values $\mathtt{e}_{ik}$.

$$
\begin{array}{llll}
(\mathtt{0,\ 0,\ 0}) = 0 & \rightarrow & \Delta E = -12 \qquad & (\mathtt{1,\ 0,\ 0}) = 4 \;\;\rightarrow\;\; \Delta E = 4 \\
(\mathtt{0,\ 0,\ 1}) = 1 & \rightarrow & \Delta E = -8 & (\mathtt{1,\ 0,\ 1}) = 5 \;\;\rightarrow\;\; \Delta E = 8 \\
(\mathtt{0,\ 1,\ 0}) = 2 & \rightarrow & \Delta E = -4 & (\mathtt{1,\ 1,\ 0}) = 6 \;\;\rightarrow\;\; \Delta E = 12 \\
(\mathtt{0,\ 1,\ 1}) = 3 & \rightarrow & \Delta E = 0 &
\end{array}
\tag{13}
$$

Now, the aim is to define a mask in order to flip the right spins with a XOR operation

$$\mathtt{spin = spin\hat{}mask;} \tag{14}$$

As a first step we compare the random number $\mathtt{R}$ with the non-trivial transition probabilities defining the variables

$$
\begin{aligned}
&\mathtt{cond12 = -(R < EXP12);} \\
&\mathtt{cond8 = -(R < EXP8);} \\
&\mathtt{cond4 = -(R < EXP4);}
\end{aligned}
\tag{15}
$$

i.e. if $\mathtt{R} < \mathtt{EXP4}$ then $\mathtt{cond4 = 0xffffffff}$ (all bits set equal to one), whereas if $\mathtt{R} > \mathtt{EXP4}$ then $\mathtt{cond4 = 0x00000000}$ (all bits equal to zero). Clearly, if $\mathtt{cond12 = 0xffffffff}$, *i.e.* the most improbable flip can be accepted, then all spins must be flipped. Also all spins with $\mathtt{sum2 = 0}$ must be flipped so that we can write

$$\mathtt{mask = cond12 \mid (\tilde{}sum2);} \tag{16}$$

17

where the | stands for the bitwise OR operator. We still need to handle the two remaining non-trivial cases corresponding to $\Delta E = 4,\ 8$. A first selection is obtained by using sum2 as a mask, although we must discard the case $\Delta E = 12$, hence we write sum2 & (sum2 ^ sum1). The flipping condition for the cases $\Delta E = 4, 8$, when R < EXP8, simply reads (sum2 & (sum2 ^ sum1)) & cond8. The last step is to consider $\Delta E = 4$ when R < EXP4 which leads to (sum2 & (sum2 ^ sum1)) & (cond8 | (cond4 & (~sum0)). All in all the mask reads

$$
\begin{aligned}
\texttt{mask = cond12 | (\~sum2)} \\
\texttt{| ((sum2 \& (sum2\^{}sum1)) \& (cond8 | (cond4 \& (\~sum0))));}
\end{aligned}
\tag{17}
$$

This expression has the same number of bitwise operations of the natural extension of [40].

Results

We present now the results concerning the performances of the different GPU implementations which are labeled as **sliced**, **standard** and **bitwise**. The **sliced** one uses the sliced checkerboard scheme we propose in this work, whereas the **standard** and **bitwise** implementations are based on the usual checkerboard scheme with the difference that the last one only works for linear sizes which are powers of two, $L = 2^\ell$ and the calculations are implemented mainly through bitwise operations. We checked that all these schemes give the same bit-to-bit results so that they are completely equivalent[47].

Before discussing the results let us define the principal metric we will use in order to measure performances: the pico-second-spin-flip $\mathrm{psFlip}_{n,\mathrm{X}}$ that is how many pico-seconds are needed in order to reject or accept a proposed spin-flip. Here n stands for the number of GPUs and 'x' for the used PRNG. The mathematical definition is the following

$$
\mathrm{psFlip}_{n,\mathrm{X}}(L,k) = t_{\mathrm{SW}} \cdot n \cdot \left(32 \cdot k \cdot 4 \cdot L^3\right)^{-1},
\tag{18}
$$

where $32 \cdot k$ is the number of different disorder realizations (32 multispin-coded times k different codings), 4 is the number of simulated replicas, and t_{SW} is the wall-clock time needed to perform one sweep, *i.e.* update red and blue spins, for *all* disorder realizations: t_{SW} is always measured on a single node. Data were taken for four different GPUs: GTX 680, GTX Titan, Tesla M2090 and Tesla K20x.
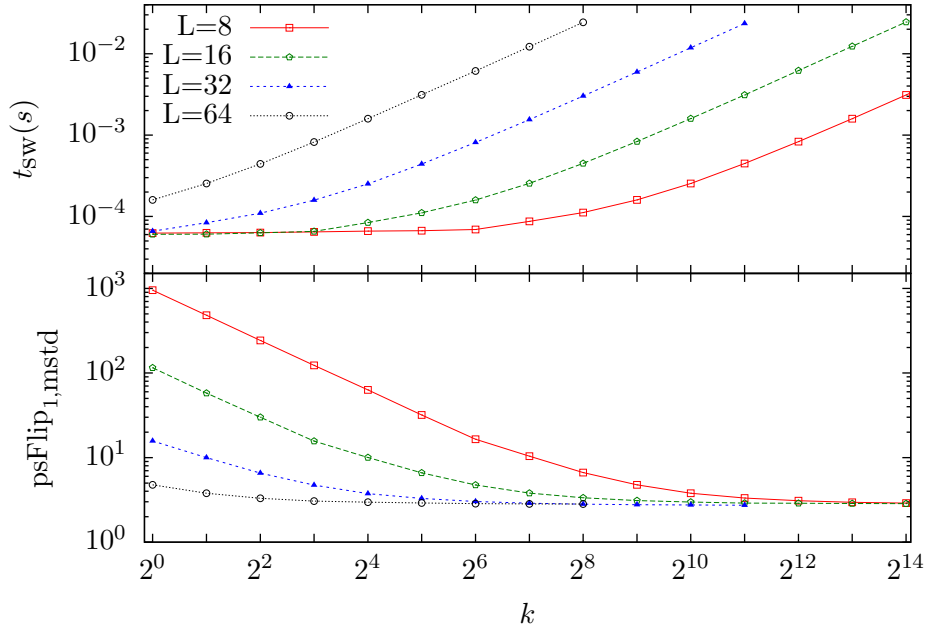
FIG. 4: Values for t_{sw} and $\text{psFlip}_{1,\text{mstd}}$, *i.e.* the number of pico-seconds needed to accept or reject a flip proposal with the MINSTD PRNG, as a function of the number of coded systems k for a GTX Titan. Data refer to the best performances at varying grid launch parameters for a given value of k for different sizes L.

Let us begin by studying a problem that has been hardly explored in the past: how to saturate the GPU resources for small lattices. The solution we adopted, as others did [17, 23], is to allocate at the same time different systems. As it is shown in Fig.4, in the case of $L = 8$, t_{sw} is almost constant up to $k = 64$ which means that simulating one or 64 coded systems has the same cost for the GPU. This means that a factor 64 can be gained for free. Indeed, this observation is important since the accessible physics for the EA3D is still confined to relatively small lattices, hence obtaining the best result also for $L \leq 32$ is crucial. We notice that even though, for $k > 64$, t_{sw} starts to increase, a linear regime is attained only for $k \geq 4096$ in the case $L = 8$. Indeed, the case $L = 32$ saturates the GPU almost at the beginning and the metric $\text{psFlip}_{1,\text{mstd}}$ only evolves from 4 psFlip to 3 psFlip, which however is a $\sim 25\%$ gain.

Now, in order to make a fair comparison with Janus FPGA hardware [6, 41], it is important to stress that those machines sustain comparable performances in terms of pico-seconds-spin-flip (16 psFlip for Janus and 3-5 psFlip for Janus II) for a **single** sample also

19

for small lattice sizes. In our case we need to simulate several samples to saturate the GPU resources. Hence, Janus and Janus II are the fastest solution in terms of wall-clock time to bring a single sample to equilibrium and for small lattice sizes GPUs are still far away. A direct comparison with Janus supercomputers can be only performed when a single system is large enough to saturate the GPU resources. However, the game is subtle since saturation is attained only for large sizes which might be out of the domain of physical interest, at least for equilibrium simulations.

In Figures 5(a), 5(b), 6(a) and 6(b) we report benchmarks results for different GPUs and different algorithms on all even lattice size in the range $8 \leq L \leq 256$. They all share the MINSTD as PRNG. Benchmarks were performed measuring the sweep wall-clock time t_{sw} while varying L, k and the grid configuration for the kernel, in order to find the best configuration for each lattice size, *i.e.* only the best configurations times are reported. There are some qualitative features which are shared by the different GPUs

- the best performances are obtained in the first range of lattice sizes $L < L_{thr}$, where the threshold L_{thr} varies according to the GPU and the algorithm, assuming larger values for latest GPUs; L_{thr} is defined as the first value of L for which psFlip$_{1,\mathrm{mstd}}$ begins to grow significantly;

- the sliced scheme performances get worse always before those of the standard scheme do;

- the sliced scheme gives always the best performance for $L > L_{thr}$;

- we split the data in two different branches defined by two subsequences of the lattice size $L_0 = 4m$ (faster) and $L_1 = 2(2m + 1)$ (slower), which converge for $L > L_{thr}$, and this splitting is most evident for small lattice sizes.

The sliced scheme worsen before the standard does probably because the latter deals with boundary conditions on the y-axis only after $L^2/2$ elements have been processed whereas for the sliced scheme the boundary conditions on the y'-axis are treated after L^2 elements. Hence, a cache hit for the standard scheme is more likely. It appears that the behaviour of the GPU memory is somehow correlated to the number of memory requests for the periodic boundaries. As a matter of fact, the following scaling relation $L_{thr}^{standard} \sim \sqrt{2}\, L_{thr}^{sliced}$ roughly holds.
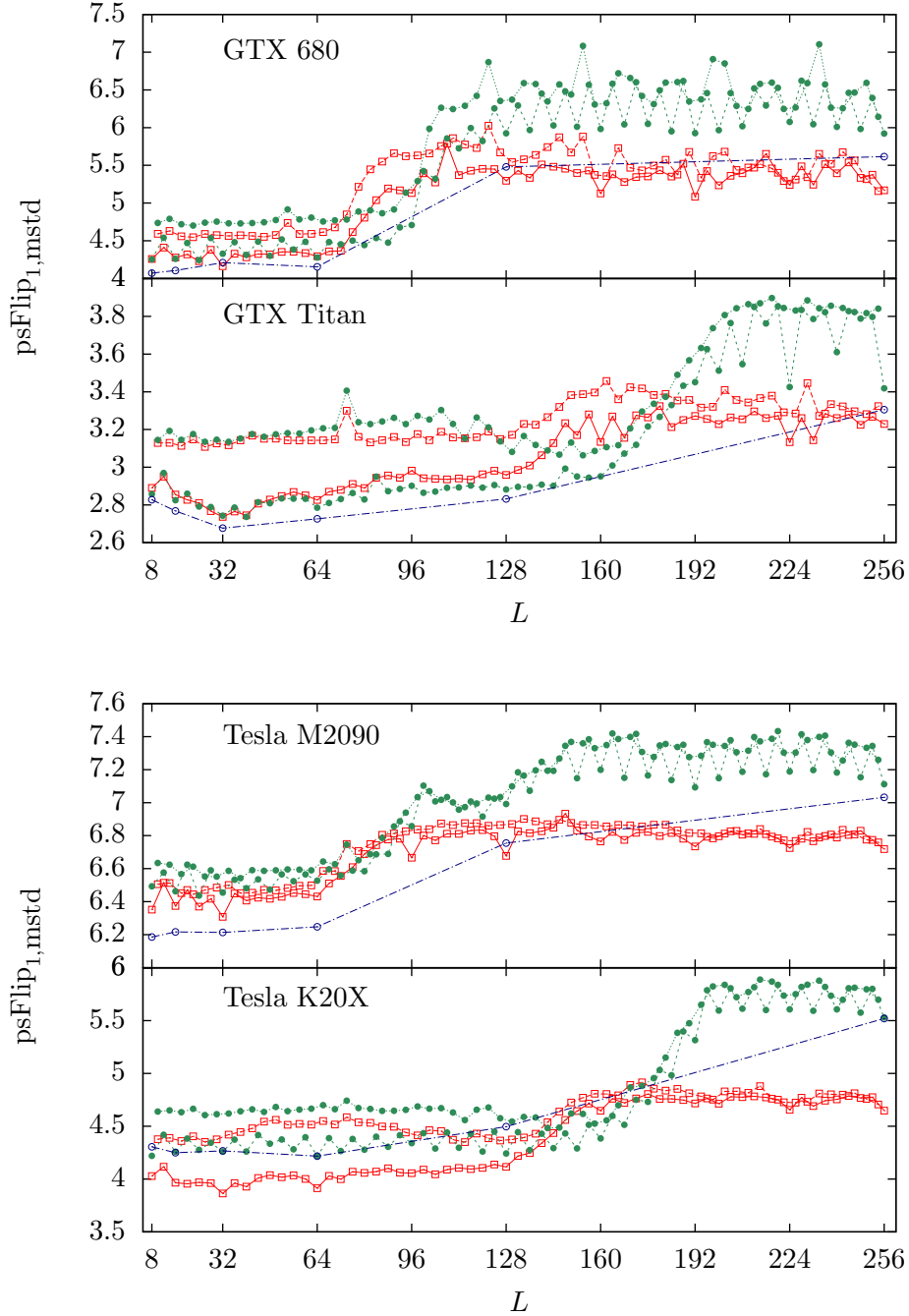
FIG. 5: Best performances for psFlip$_{1,mstd}$. The red empty squares refer to the **sliced** implementation, the light-green filled circles refer to the **standard** implementation whereas the blue empty circles to the **bitwise** one. The value of L_{thr} is larger for the GTX Titan and the Tesla K20x.

Data related to the standard-Grid and sliced-Grid implementations are clearly less stable.We notice that for $L \gtrsim 8$ the standard-Grid implementation performs much worse than
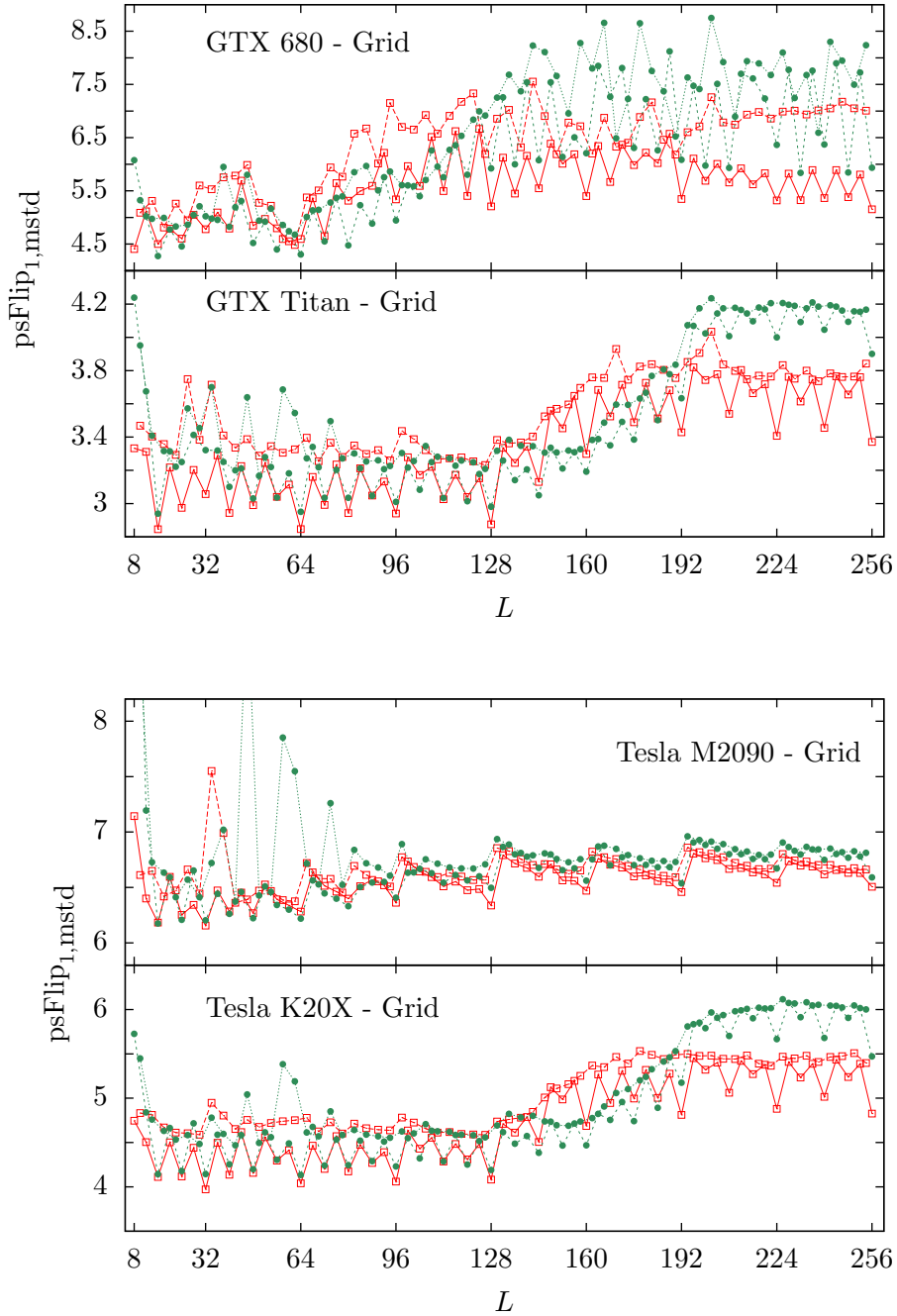
FIG. 6: Best performances for $\mathrm{psFlip}_{1,\mathrm{mstd}}$ for Grid implementations. The red empty squares refer to the **sliced-Grid** implementation and the light-green filled circles refer to the **standard-Grid** implemention.

the sliced-Grid. This should be related to the fact that the block size is fixed to the number of one-coloured spins in a z slice, which means for the standard-Grid scheme $L^2/2$, starting

from 32 threads, and for the sliced-Grid L^2, starting from 64 threads. Clearly, having blocks which coincide with a warp is not an optimal choice for the GPU. Looking at the data for the Tesla M2090 in 6(b) there is a modulation as a function of the lattice size with a period $\Delta L = 32$. We notice that for such values of L the blocks are always multiple of a warp.

The 'Grid' algorithms perform slower than the others in the examined range so that we can safely discard this implementation choice which relies on the inherent algebra of the thread-grid indices. Hence, we will focus hereafter mainly on the non-grid implementations.

FIG. 7: Upper panel: best values of k for the GTX 680. The red empty squares refer to the **sliced** implementation, the light-green filled circles refer to the **standard** implementation whereas the blue empty circles to the **bitwise** one. Lower panel: number of branched warps divided by k: for the sliced and the standard implementations the subsequence $L_1 = 2(2m+1)$ has a divergent warp for each system.

In Fig. 7 we report the number of different coded systems k as a function of L and the number of branching warps normalized to k: such a ratio gets only two values marking a distinction between the two lattice size subsequences. More details can be found in Appendix B. To complete the analysis for the best performances, we report in Fig. 8 the results for the bandwidth measures for the best launch configurations. Except for the Tesla M2090,
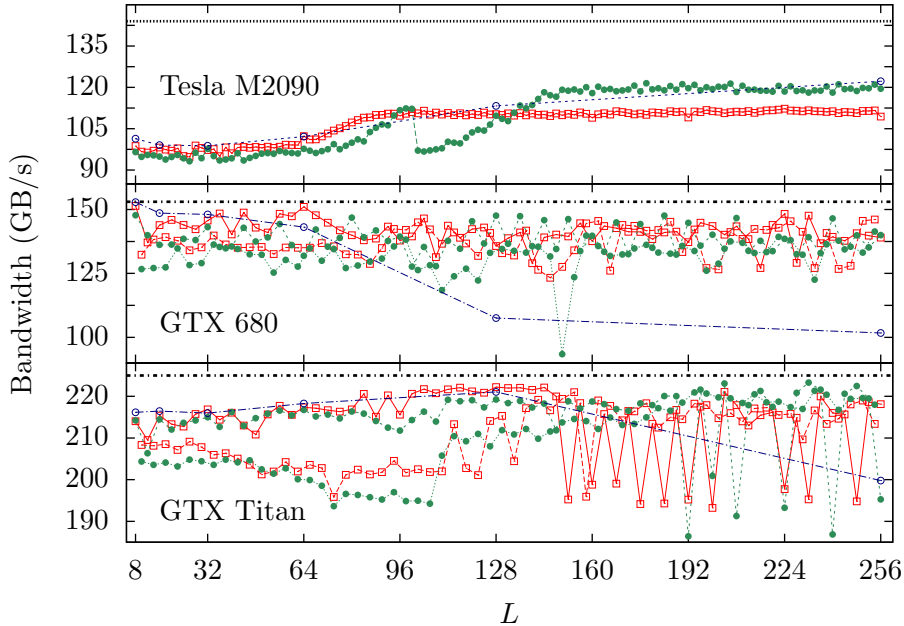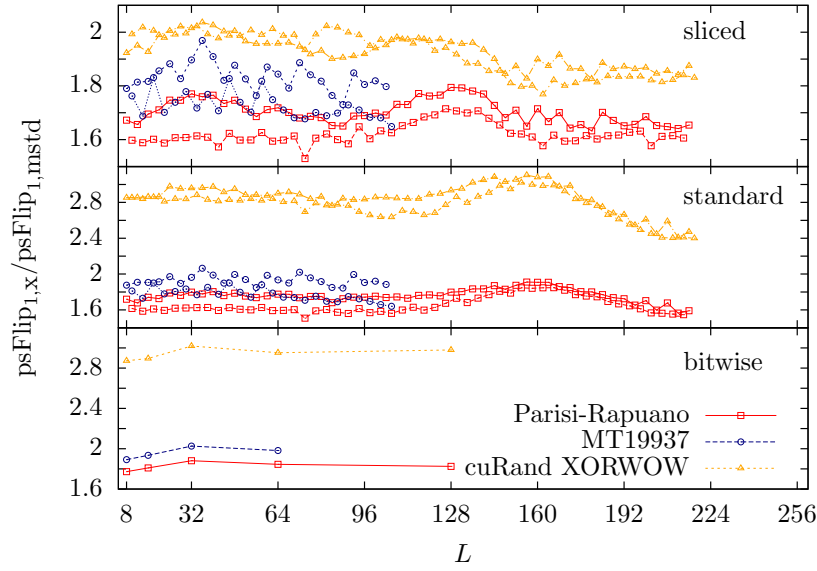
FIG. 8: Data for the GTX 680 and GTX Titan GPUs. The red empty squares refer to the **sliced** implementation, the light-green filled circles refer to the **standard** implementation whereas the blue empty circles to the **bitwise** one. The horizontal lines are the peak bandwidths as measured with CUDA SDK code.

the sliced and the standard algorithms saturate the available bandwidth in the entire range with some fluctuations.
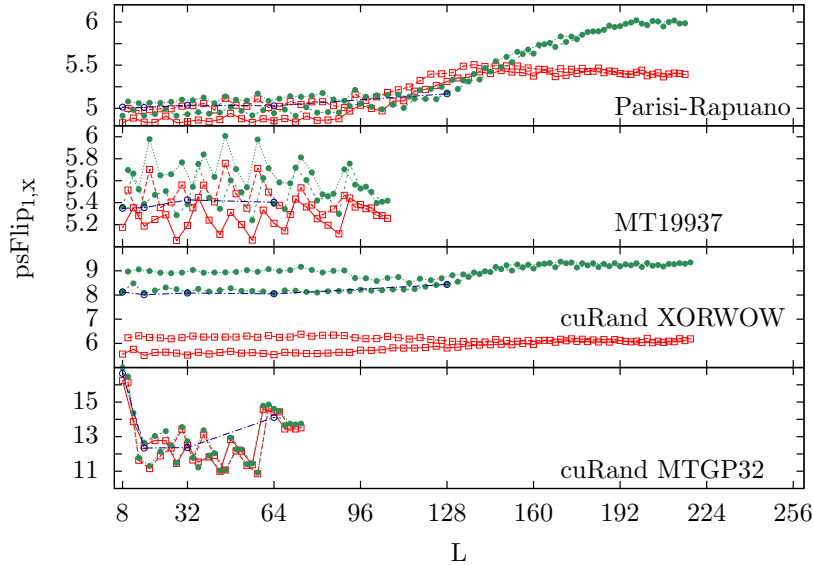
Let us now examine the results for different PRNGs: our implementations of the Parisi-Rapuano and the usual Mersenne Twister MT19937, together with the cuRand XORWOW (which is the standard cuRand PRNG) and MTGP32 which is a reduced version of the MT19937. While for the first three PRNGs we could perform full benchmarks with the only limitation of the memory usage, for the MTGP32 we could use a maximum number of 200 blocks and a maximum blocks size of 256 threads. As for the number of blocks, this is a limitation of the standard usage which, however, can be by-passed with some effort as reported in the cuRand documentation [42].

Results are reported in Fig. 9(a) and 9(b). In Fig. 9(a) we show the values of $psFlip_{1,x}$ normalized to the MINSTD performances $psFlip_{1,mstd}$, which we use as a baseline, for the three different algorithm implementations. In $psFlip_{1,x}$, 'x' labels three different PRNGs: Parisi-Rapuano, MT19937 and XORWOW. All data refer to the GTX Titan GPU. It is clear

(a)

(b)

FIG. 9: In figure (a) the $psFlip_{1,x}$ performances normalized to the MINSTD are shown. In figure (b) the abolute performances are reported where the red squares represent the sliced scheme data, the filled green circles and the empty blue circles those of the standard and bitwise schemes respectively. All data refer to the GTX Titan.

that the lowest ratio for the XORWOW is obtained for the sliced implementation for which it is ~ 2 whereas for the standard and bitwise versions the ratio is ~ 3. The Parisi-Rapuano

and the MT19937 have roughly the same ratio for the three different algorithms.

In Fig. 9(b) we report the absolute values for $psFlip_{1,X}$. It is possible to see that the performances for our implementations of the Parisi-Rapuano and MT19937 and of cuRand MTGP32 weakly depend on the chosen algorithm, while there is a considerable difference for the XORWOW for which $psFlip_{1,xor} \sim 6ps$ for the sliced scheme while $psFlip_{1,xor} \sim 9ps$ for the standard and bitwise implementations. There are two main results emerging from the data:

- the standard cuRAND XORWOW performs slower than our best-quality PRNG, the MT19937;

- the sliced scheme is more robust with respect to a change in the memory bandwidth load.

The first point can be easily understood by considering that the data structure of the cuRand XORWOW PRNG has a size of 48 bytes: each 128 byte transaction, which is served from the L2 cache, only loads the data needed by two threads, so that we need roughly 16 memory transactions for a warp to be ready, whereas in our approach we only need $O(1)$ memory transactions, *e.g.* 3 for the Parisi-Rapuano and the MT19937. Indeed, also the MTGP32 follows a similar pattern because every thread in a warp loads in the shared memory one entry of the state. The strategy used for the XORWOW implementation is not adequate for intense memory usage algorithms.

As for the second point this should be a proof that the memory alignment given by the sliced scheme is better suited for second hits in the caches: indeed the amount of needed data transfers is the same for the three schemes in the XORWOW case but for the sliced scheme there is a $\sim 33\%$ gain with respect to the standard and bitwise schemes.

As a final remark, the MTGP32 performs from 3 to 5 times worse than the MINSTD implementation. We stress that this result is strongly influenced by some limitations of the cuRand implementation which, however, can be softened with some further work.

V. MULTI-GPU IMPLEMENTATION

As far as we know, there are just a few works showing strong scaling results for spin systems [13, 20, 21]. We chose to adopt the same technique proposed in [20, 21] where the

partitioning is performed along the z'-axis of the system. All communications among nodes are handled by MPI and the overlap between calculations and communications is achieved by using CUDA streams. We keep the single-GPU version flexibility for a customary number of spins per thread and coded systems k. A priori, it should not be taken for granted that the bulk update, executed on one CUDA stream, can mask the boundary update and data copy/transfer, executed on the other stream, since the algebraic intensity of the algorithm is rather low.
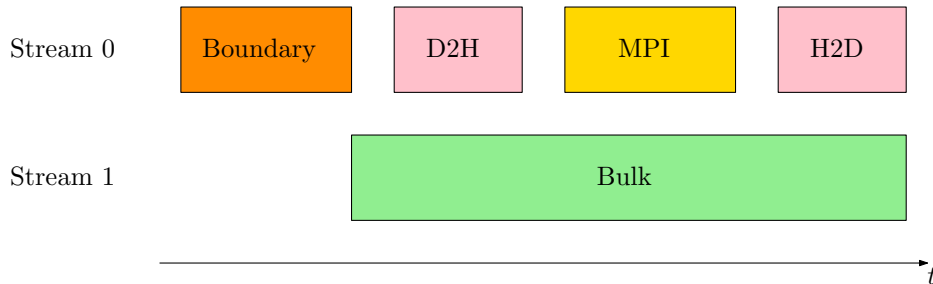

FIG. 10: Scheme representing the multi-GPU strategy leveraging CUDA streams. Here 'Boundary' and 'Bulk' represent two kernels launched on the same GPU. After the boundary update on the stream 0 an asynchronous 'D2H' device-to-host copy of the only one-coloured boundary is performed, then 'MPI' handles the one-directional boundary exchange between nodes and an asynchronous 'H2D' host-to-device memory copy updates the boundary spins.

The multi-GPU version of the sliced Kernel is rather different from the one using the standard checkerboard scheme [20, 21] since the disposition of colours in the cubic lattice is different. At fixed z' value spins are one-coloured so that for every partition of the system the lowermost plane is always red whereas the highermost one is always blue. This means that when updating red spins the only boundary coincides with the lowermost red plane or with the highermost blue one when updating the blue spins. Hence, the communication between the nodes goes in the downward direction for red spins and in the upward one for the blue spins: there is no need for all nodes to communicate with all nearest neighbours after a colour update. To-be-sent boundary spins are stored in the bulk array and copied to an auxiliary buffer by the same kernel that performs the update. To-be-recieved boundary spins are stored in a separate array, bound to a texture, which is just read when updating the spins of the other colour. This scheme automatically handles the z'-axis periodic boundary conditions and reduces the number of intra-node communications. In Fig. 11 we report a

depiction of the multi-GPU sliced scheme.

This is an interesting property which might be of use in cases where the amount of data to transfer is low and the latency time is comparable to the data-exchange time. Then, one would expect to have a significant speed-up in the communication.
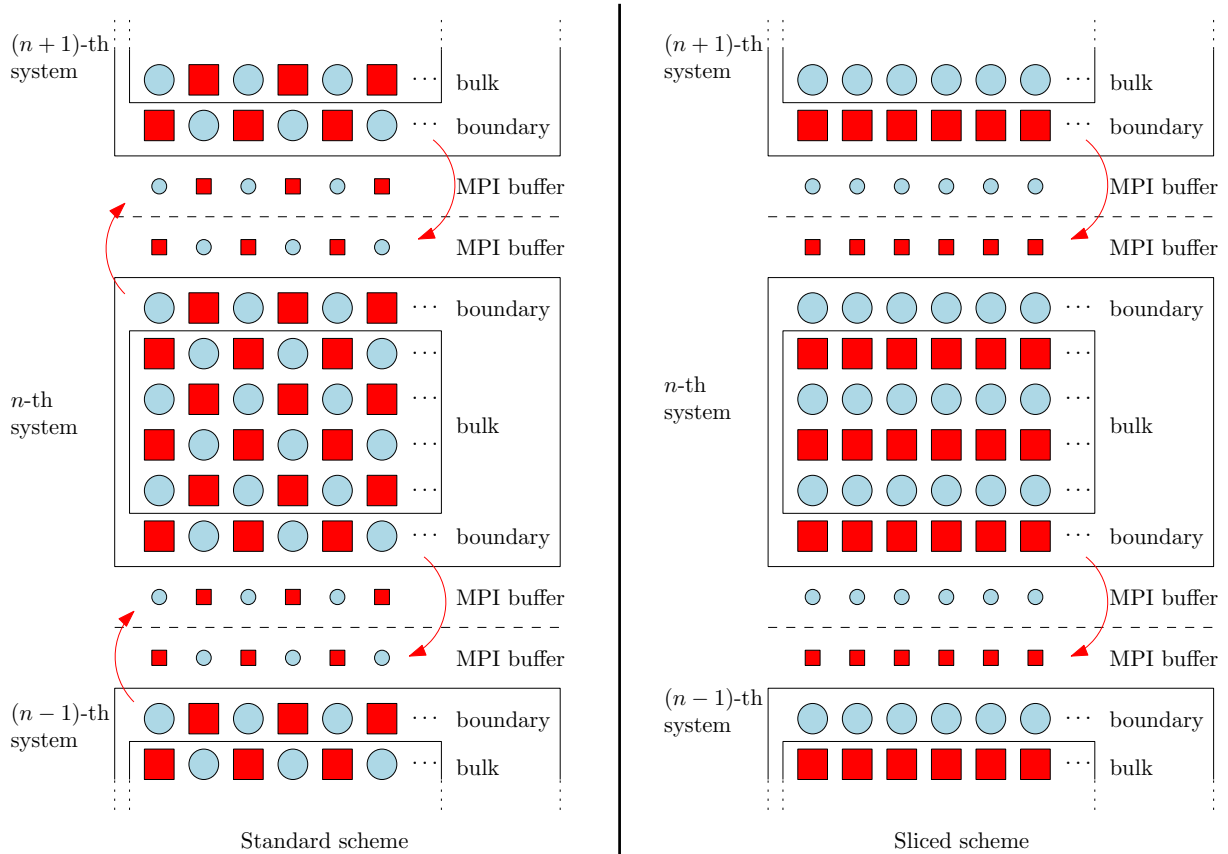

FIG. 11: A depiction of the standard (on the left) and the sliced (on the right) checkerboard schemes for the multi-GPU version. For the update of red spins one needs to update the bulk and the boundaries of each system partition. The standard scheme has two-coloured boundaries while for the sliced scheme these are one-colured: communication (red arrows) must be two-ways for the standard implementation whereas it is only one-directional for the sliced scheme. Clearly, in both cases the same amount of data is transferred.

Results

Let us now discuss the results we obtained for the multi-GPU implementation of the three-dimensional Edwards-Anderson model. Given the definition (18) of $\mathrm{psFlip}_{N,\mathrm{X}}$, it clearly

appears that for $N > 1$ we consider the time spent by a single GPU on its own system partition rather than the wall-clock time spent by the N GPUs as a whole. However, the strong-scaling efficiency η_{sc} is directly defined as

$$\eta_{SC} = \frac{\text{psFlip}_{1,\text{X}}}{\text{psFlip}_{N,\text{X}}}, \tag{19}$$

and thus the performance referred to the multi-GPU system as a whole is defined as

$$\text{psFlip}_{\text{multi,x}} = \text{psFlip}_{N,\text{X}}/N, \tag{20}$$

allowing to recover the usual strong-scaling efficiency definition.

$$\eta_{SC} = \frac{\text{psFlip}_{1,\text{X}}}{\text{psFlip}_{N,\text{X}}} = \frac{\text{psFlip}_{1,\text{X}}}{N \cdot \text{psFlip}_{\text{multi,x}}}. \tag{21}$$

All data have been gathered on the Piz Daint Supercomputer which uses Tesla K20x GPUs [30]. In Fig. 12(a) we report the strong scaling efficiency up to 8 GPUs. Indeed, the saturation efficiency is remarkable, $\eta_{sc} \gtrsim 0.9$, although the more the GPUs the further in terms of lattice size L one needs to go to reach a stable regime. Nonetheless up to 8 GPUs the algorithm practically scales linearly with the number of GPUs.

In Fig. 12(b) where we show the values of $\text{psFlip}_{\text{multi,mstd}}$, hence considering N GPUs as a single system, the linear scaling in N is clearly visible for any number of GPUs. We obtain very good results in absolute terms: for $N = 2$ in the range from $L = 64$ to $L = 128$ we have almost stable performances at $2\,\text{ps} < \text{psFlip}_{\text{multi,mstd}} < 3\,\text{ps}$ which is of interest for real scientific applications.
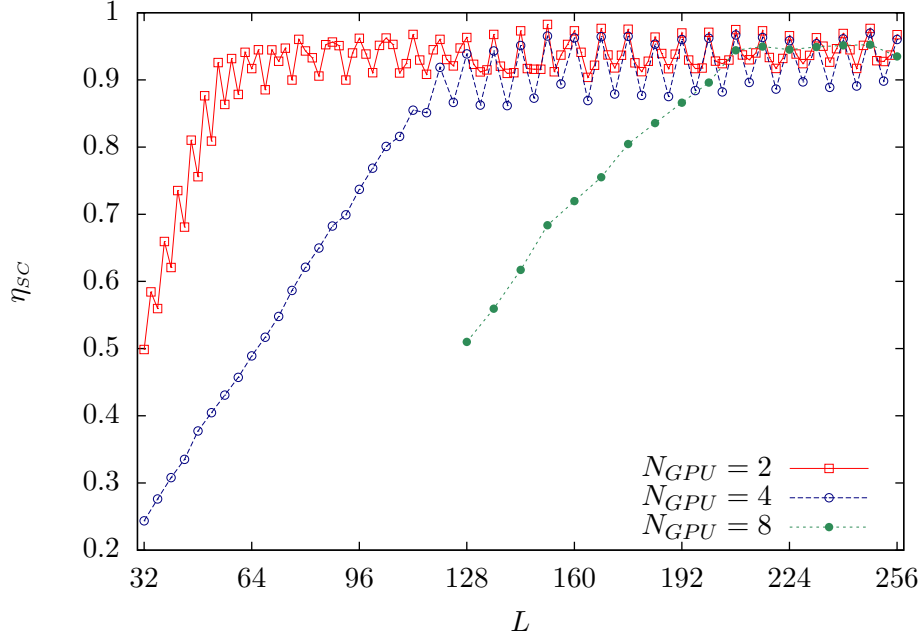
Lastly, we want to pay some attention to the power-law behaviour visible in 12(b). It is easy to determine that, roughly, performances scale as

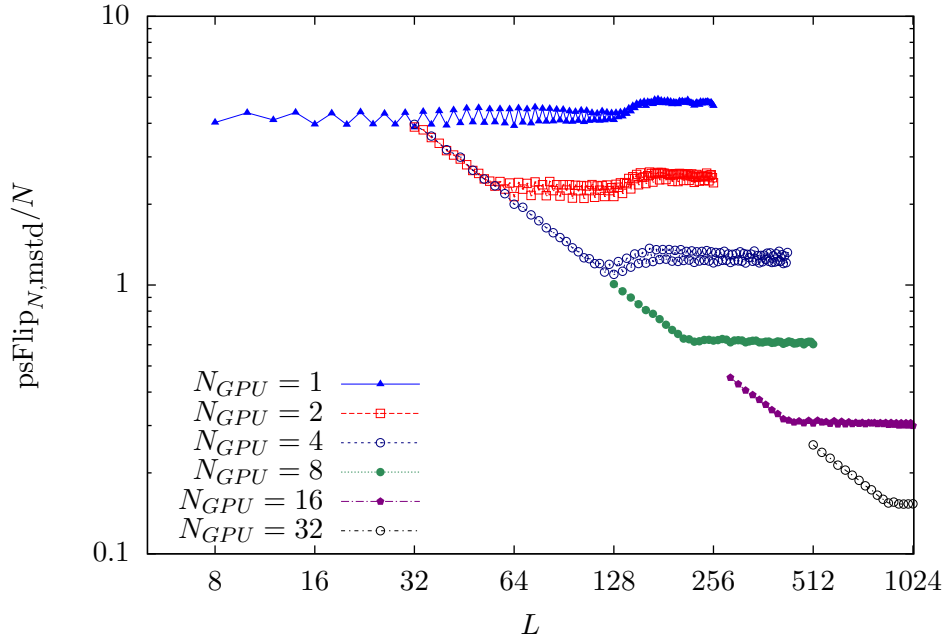$$\text{psFlip}_{\text{multi,mstd}} \sim L^{-1}. \tag{22}$$

Now, looking at the definition (18), it is easy to derive that

$$\frac{L}{N} \text{psFlip}_{N,\text{X}}(L, k) \propto t_{\text{sw}} \left(\frac{L}{N} \right)^{-2}, \tag{23}$$

hence, defining the rescaled variable $x = L/N$ we can plot $x\,\text{psFlip}_{N,\text{X}}$ as a function of x. The result is shown in Fig. 13. Indeed, we can see that data for every considered value of N collapse on the same curve, proving that x is a good scaling variable. From the plot two distinct regimes are visible: a first one where data lie on a horizontal line and a second

(a)

(b)

FIG. 12: In Fig. (a) the strong-scaling efficiency η_{sc} is reported for different numbers of GPUs. In Fig. (b) the multi-GPU system performances are shown. A power-law behaviour as $\mathrm{psFlip}_{\mathrm{multi,mstd}} \sim L^{-1}$ is noticeable.
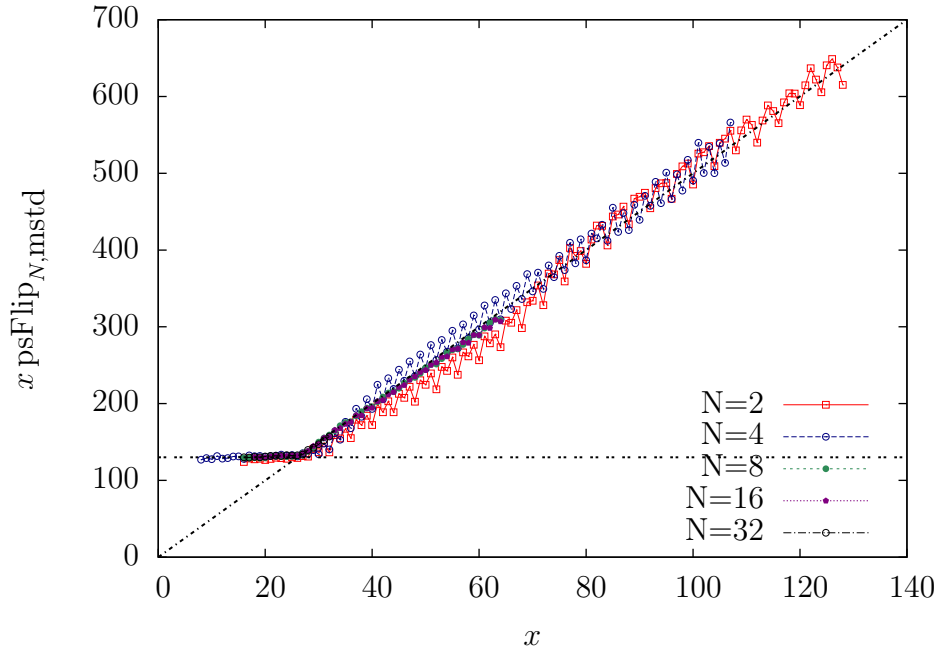
FIG. 13: Scaling plot for the performances of the multi-GPU system. The initial constant value indicates a scaling for the sweep wall-clock time as L^2, whereas for the linear growth in x the sweep time scales as L^3 signaling a cross-over from a communication dominated to a bulk calculations dominated regime. The data collapse is possible for the high quality of the inter-node communication.

one where they grow linearly in x. In the first regime the sweep wall clock time grows as $t_{SW} \sim L^2$, *i.e.* the boundary communication, which scales as the system area, dominates. In the second regime $t_{SW} \sim L^3$, which means that the wall-clock time is dominated by the bulk update task which is then able to mask the communication between the nodes.

Indeed, such a good data scaling and collapse shows the stability of the communication offered by the Piz Daint Supercomputer based on Aries routing, communications ASIC, and Dragonfly network topology [43, 44]. It is possible to use this kind of analysis in order to measure the communication infrastructure quality.

VI.   CONCLUSIONS

In this work we have studied different strategies for the implementation of the Metropolis dissipative dynamics for the three-dimensional Edwards-Anderson bimodal spin glass. We

proposed new access patterns for both the cubic stencil data structure and lagged-Fibonacci-like PRNGs. We showed, comparing different GPUs and different algorithm implementations that it is possible to obtain stable performances on a wide range of lattice sizes, $8 \leq L \leq 256$. For some GPUs our new **sliced** scheme performs slightly better than the other schemes, for the version which uses the MINSTD PRNG. However, the sliced scheme performs always better for large sizes L and when the data transfer load is increased using more complex PRNGs. In particular the sliced scheme gains roughly the 30% over standard implementation for the cuRand XORWOW. As for the comparison of different PRNGs we showed that our implementation of the full Mersenne-Twister MT19937 performs better than the standard cuRand XORWOW thus indicating a new implementation strategy for PRNGs which turns out to be very efficient for memory bandwidth demanding algorithms. Indeed, the MT19937 performs only 70% worse than the MINSTD congruential PRNG with our approach.

Of course at the basis of such results there is the possibility of using the asynchronous multispin-coding (AMSC) technique which allows us to store one spin of 32 different systems in a word. We explained how this technique is implemented in our case.

In terms of single GPU we showed that it is possible to obtain performances comparable to those of dedicated FPGA hardware [41] although one should be careful in this respect. However, single GPU performances are enough to obtain competitive results for critical parameters estimations using the out-of-equilibrium relaxation regime [45].

Furthermore, we explored the multi-GPU version of the sliced scheme which presents the intriguing feature of halving the number of MPI data transactions while, obviously, keeping the total amount of data transfer fixed. We showed that a very high strong-scaling efficiency can be reached leading to scientifically interesting performances in the range $64 \leq L \leq 128$.

Many of these result can be extended and reused outside the statistical mechanics domain since they involve cubic lattice discretization, along with their multi-GPU extension, and high quality random numbers PRNGs implementations.

VII.   ACKNOWLEDGEMENTS

Appendix A: Grid parameters and periodic boundary conditions

We now list some details for grid configurations of non-Grid and Grid kernels. The kernel launch parameters for the first case are defined as follows

```
dim3 block(blockSize,1,1);
int fitGrid = (V/2/s + blockSize - 1)/blockSize;
dim3 grid(fitGrid, k, 1);
```

where $s = s$ is the number of spins per thread and $blockSize = 32n$, *i.e.* a multiple of the warp size. For Grid kernels the launch parameters are:

```
dim3 blockG(L, l, 1);
dim3 gridG(A/(blockG.x*blockG.y), L/2, k);
```

where `A=L*L` and `l` is the number of lines of a single plane updated by a thread block. We highlight that for the latter case we use `threadIdx.x`, `threadIdx.y` and `blockIdx.y` as x, y and z indices respectively.

Let us now discuss some implementations details for periodic boundary conditions. The nearest neighbours indices are always calculated from the one-dimensional index of the spin that is updated. Here we report the calculation which are the same for the sliced and sliced-Grid kernels in order to be as clear as possible

```
int smz = i + (SM(z - 1, d_hL) - z)*d_A;
int spy = smz + (SP(y + 1, d_L) - y)*d_L;
int smy = i + (SM(y - 1, d_L) - y)*d_L;
int smx = spy - x + SM(x - 1, d_L);
int spx = smy - x + SP(x + 1, d_L);
```

where `i = kk + off`, with `kk` $< V/2$ and `off = blockId.y,z*d_hV` being a disorder offset (with `d_hV` $= V/2$). We have implicitly set `spz = i`. In order to avoid the modulus operation enforcing the periodic boundary conditions we defined the macros `SM` and `SP` which read

```
#define SP(a, m) (a&(~(-(a >= m))))
#define SM(a, m) (a+((-(a < 0))&m))
```

Let us briefly comment the definition of `SP`: if `a >= m` it evaluates to `1` then

```
(~(-(a >= m))) = 0x00000000
```

i.e. all bits set to zero, otherwise one has

```
(~(-(a >= m))) = 0xffffffff
```

i.e. all bits set to one. The macro `SM` is completely analogous. Hence we have reproduced the periodic boundary conditions since `SP(m + 1, m) = 0` and `SM(-1, m) = m - 1`.

We remark that we had to define another macro `SMM` for the Grid version in order to handle the fact that `threadIdx` and `blockIdx` variables are unsigned integers.

Appendix B: Even L subsequences and warp branchings

In Fig.7 we report the optimal values of the number of coded systems k as a function of the lattice size which we can see decreases roughly in a power-law fashion. In particular we also show the ratio between the number of branching warps and k: for lattices belonging to the subsequence $L_1 = 2(2m+1)$ this ratio is always equal to one, whereas for the subsequence $L_0 = 4m$ it is always equal to zero. This result is explained by the fact that half of the volume of a lattice $V_i/2 = L_i^3/2$, *i.e.* all the one-coloured spin, is always a multiple of the warp size for the even subsequence $V_0/2 \propto 32$ whereas it is not so for the odd subsequence $V_1/2$,

$$\frac{V_0}{2} = \frac{(4m)^3}{2} = 32m^3, \qquad \frac{V_1}{2} = \frac{[2(2m+1)]^3}{2} = 4(2m+1)^3. \tag{B1}$$

In order to prove this let us look if there exist a value of m for which $V_1/2$ is a multiple of the warp size

$$\frac{V_1}{2} = 4(2m+1)^3 = 32n, \qquad 2m+1 = 2n^{1/3}, \qquad m = n^{1/3} + \frac{1}{2}, \tag{B2}$$

which has integer solutions for m for non-integer n. Since we are looking for integer values of n, this proves the previous assertion. Hence, the subsequence of cubic lattices of linear size L_1 is intrinsically uncommensurate to the actual warp size which is characteristic of the CUDA framework. Thus, as long as the warp size is fixed to the actual value, there

will always be warp branchings for checkerboard algorithms updating one colour at the time. Indeed, this result is correlated to the fact that the two subsequences L_0 and L_1 have different performances, but does not provide a full explanation.

---

[1] M. Baity-Jesi, R. A. Baños, A. Cruz, L. A. Fernandez, J. M. Gil-Narvion, A. Gordillo-Guerrero, D. Iñiguez, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, J. Monforte-Garcia, A. M. n. Sudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, M. Pivanti, F. Ricci-Tersenghi, J. J. Ruiz-Lorenzo, S. F. Schifano, B. Seoane, A. Tarancon, R. Tripiccione, and D. Yllanes, "Critical parameters of the three-dimensional Ising spin glass," *Phys. Rev. B*, vol. 88, p. 224416, Dec 2013.

[2] F. Belletti, A. Cruz, L. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, J. Monforte, A. MuozSudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, J. Ruiz-Lorenzo, S. Schifano, D. Sciretti, A. Tarancon, R. Tripiccione, and D. Yllanes, "An In-Depth View of the Microscopic Dynamics of Ising Spin Glasses at Fixed Temperature," *Journal of Statistical Physics*, vol. 135, no. 5-6, pp. 1121–1158, 2009.

[3] J. H. Condon and A. T. Ogielski, "Fast special purpose computer for Monte Carlo simulations in statistical physics," *Review of Scientific Instruments*, vol. 56, no. 9, pp. 1691–1696, 1985.

[4] A. Cruz, J. Pech, A. Tarancn, P. Tllez, C. Ullod, and C. Ungil, "SUE: A special purpose computer for spin glass models," *Computer Physics Communications*, vol. 133, no. 23, pp. 165 − 176, 2001.

[5] M. Baity-Jesi, R. Baos, A. Cruz, L. Fernandez, J. Gil-Narvion, A. Gordillo-Guerrero, M. Guidetti, D. Iiguez, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, J. Monforte-Garcia, A. Muoz Sudupe, D. Navarro, G. Parisi, M. Pivanti, S. Perez-Gaviro, F. Ricci-Tersenghi, J. Ruiz-Lorenzo, S. Schifano, B. Seoane, A. Tarancon, P. Tellez, R. Tripiccione, and D. Yllanes, "Reconfigurable computing for Monte Carlo simulations: Results and prospects of the Janus project," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 33–51, 2012.

[6] M. Baity-Jesi, R. Baos, A. Cruz, L. Fernandez, J. Gil-Narvion, A. Gordillo-Guerrero, D. Iiguez, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, J. Monforte-Garcia, A. M. Sudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, M. Pivanti, F. Ricci-Tersenghi, J. Ruiz-Lorenzo,

S. Schifano, B. Seoane, A. Tarancon, R. Tripiccione, and D. Yllanes, "Janus II: A new generation application-driven computer for spin-system simulations," *Computer Physics Communications*, vol. 185, no. 2, pp. 550 – 559, 2014.

[7] M. Hasenbusch, A. Pelissetto, and E. Vicari, "Critical behavior of three-dimensional Ising spin glass models," *Phys. Rev. B*, vol. 78, p. 214205, Dec 2008.

[8] E. Marinari and G. Parisi, "Simulated Tempering: A New Monte Carlo Scheme," *EPL (Europhysics Letters)*, vol. 19, no. 6, p. 451, 1992.

[9] S. Tomov, M. McGuigan, R. Bennett, G. Smith, and J. Spiletic, "Benchmarking and implementation of probability-based simulations on programmable graphics cards," *Computers & Graphics*, vol. 29, no. 1, pp. 71 – 80, 2005.

[10] V. I. Manousiouthakis and M. W. Deem, "Strict detailed balance is unnecessary in Monte Carlo simulation," *The Journal of Chemical Physics*, vol. 110, no. 6, pp. 2753–2756, 1999.

[11] R. Ren and G. Orkoulas, "Acceleration of Markov chain Monte Carlo simulations through sequential updating," *The Journal of Chemical Physics*, vol. 124, no. 6, pp. –, 2006.

[12] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468 – 4477, 2009.

[13] B. Block, P. Virnau, and T. Preis, "Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model," *Computer Physics Communications*, vol. 181, no. 9, pp. 1549 – 1556, 2010.

[14] M. Weigel, "Simulating spin models on GPU," *Computer Physics Communications*, vol. 182, no. 9, pp. 1833 – 1836, 2011. Computer Physics Communications Special Edition for Conference on Computational Physics Trondheim, Norway, June 23-26, 2010.

[15] T. Levy, G. Cohen, and E. Rabani, "Simulating Lattice Spin Models on Graphics Processing Units," *Journal of Chemical Theory and Computation*, vol. 6, no. 11, pp. 3293–3301, 2010.

[16] T. Yavorskii and M. Weigel, "Optimized GPU simulation of continuous-spin glass models," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 159–173, 2012.

[17] Y. Fang, S. Feng, K.-M. Tam, Z. Yun, J. Moreno, J. Ramanujam, and M. Jarrell, "Parallel tempering simulation of the three-dimensional EdwardsAnderson model with compact asynchronous multispin coding on GPU," *Computer Physics Communications*, no. 0, pp. –, 2014.

[18] E. E. Ferrero, J. P. D. Francesco, and N. W. S. A. Cannas, "q-state Potts model metastability study using optimized GPU-based Monte Carlo algorithms," *Computer Physics Communications*, vol. 183, no. 8, pp. 1578 – 1587, 2012.

[19] M. Bernaschi, G. Parisi, and L. Parisi, "Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation," *Computer Physics Communications*, vol. 182, no. 6, pp. 1265 – 1271, 2011.

[20] M. Bernaschi, M. Fatica, G. Parisi, and L. Parisi, "Multi-GPU codes for spin systems simulations," *Computer Physics Communications*, vol. 183, no. 7, pp. 1416 – 1421, 2012.

[21] M. Bernaschi, M. Bisson, and D. Rossetti, "Benchmarking of communication techniques for GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 250 – 255, 2013.

[22] M. I. n. Berganza and L. Leuzzi, "Critical behavior of the XY model in complex topologies," *Phys. Rev. B*, vol. 88, p. 144104, Oct 2013.

[23] M. Baity-Jesi, L. A. Fernandez, V. Martin-Mayor, and J. M. Sanz, "Phase transition in three-dimensional Heisenberg spin glasses with strong random anisotropies through a multi-GPU parallelization," *Phys. Rev. B*, vol. 89, p. 014202, Jan 2014.

[24] D. B. Thomas, "Warp Generator." `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-warp_generator.html`.

[25] M. Manssen, M. Weigel, and A. Hartmann, "Random number generators for massively parallel simulations on GPU," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 53–71, 2012.

[26] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[27] Y.-D. Hsieh, Y.-J. Kao, and A. W. Sandvik, "Finite-size scaling method for the BerezinskiiKosterlitzThouless transition," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2013, no. 09, p. P09001, 2013.

[28] G. Parisi and F. Rapuano, "Effects of the random number generator on computer simulations," *Physics Letters B*, vol. 157, no. 4, pp. 301 – 302, 1985.

[29] L. Barash and L. Shchur, "PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern GPUs and CPUs," *Computer Physics Communications*, vol. 185, no. 4, pp. 1343 – 1353, 2014.

[30] "Cscs piz daint." http://www.cscs.ch/computers/piz_daint/index.html.

[31] D. F. Carta, "Two Fast Implementations of the &Ldquo;Minimal Standard&Rdquo; Random Number Generator," *Commun. ACM*, vol. 33, pp. 87–88, Jan. 1990.

[32] "Park-Miller-Carta Pseudo-Random Number Generator." http://www.firstpr.com.au/dsp/rand31/.

[33] M. Weigel, "Performance potential for simulating spin models on GPU," *Journal of Computational Physics*, vol. 231, no. 8, pp. 3064 – 3082, 2012.

[34] "cuRand Device API example." http://docs.nvidia.com/cuda/curand/device-api-overview.html#device-api-example.

[35] "MT19937 official site." http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

[36] M. Creutz, L. Jacobs, and C. Rebbi, "Experiments with a Gauge-Invariant Ising System," *Phys. Rev. Lett.*, vol. 42, pp. 1390–1393, May 1979.

[37] S. Wansleben, J. Zabolitzky, and C. Kalle, "Monte Carlo simulation of Ising models by multi-spin coding on a vector computer," *Journal of Statistical Physics*, vol. 37, no. 3-4, pp. 271–282, 1984.

[38] G. Bhanot, D. Duke, and R. Salvador, "Finite-size scaling and the three-dimensional Ising model," *Phys. Rev. B*, vol. 33, pp. 7841–7844, Jun 1986.

[39] M. Kikuchi and Y. Okabe, "Renormalization, self-similarity, and relaxation of order-parameter structure in critical phenomena," *Phys. Rev. B*, vol. 35, pp. 5382–5384, Apr 1987.

[40] N. Ito and Y. Kanada, "Monte Carlo Simulation of the Ising Model and Random Number Generation on the Vector Processor," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, (Los Alamitos, CA, USA), pp. 753–763, IEEE Computer Society Press, 1990.

[41] M. Baity-Jesi, R. Baos, A. Cruz, L. Fernandez, J. Gil-Narvion, A. Gordillo-Guerrero, M. Guidetti, D. Iiguez, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, J. Monforte-Garcia, A. Muoz Sudupe, D. Navarro, G. Parisi, M. Pivanti, S. Perez-Gaviro, F. Ricci-Tersenghi, J. Ruiz-Lorenzo, S. Schifano, B. Seoane, A. Tarancon, P. Tellez, R. Tripiccione, and D. Yllanes, "Reconfigurable computing for Monte Carlo simulations: Results and prospects of the Janus project," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 33–51, 2012.

[42] "cuRand MTGP32 Device API." http://docs.nvidia.com/cuda/curand/

device-api-overview.html#bit-generation-2.

[43] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 77–88, IEEE Computer Society, 2008.

[44] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[45] G. P. M. Lulli and A. Pelissetto, "Highly optimized simulations on single- and multi-GPU systems of 3D Ising spin glass," *in preparation*, 2014.

[46] More details can be found in the appendix

[47] Precisely, the bitwise check between the sliced and the other implementations requires to remap all random numbers after one of the two colours has been updated.