Computer Physics Communications 260 (2021) 107728

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Brute-forcing spin-glass problems with CUDA ☆

Konrad Jałowiecki [b,*], Marek M. Rams [c], Bartłomiej Gardas [a,c]

[a] Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Bałtycka 5, 44-100 Gliwice, Poland
[b] Institute of Physics, University of Silesia, Uniwersytecka 4, 40-007 Katowice, Poland
[c] Jagiellonian University, Marian Smoluchowski Institute of Physics, Łojasiewicza 11, 30-348 Kraków, Poland

ARTICLE INFO

Article history:
Received 9 April 2019
Received in revised form 14 September 2020
Accepted 10 November 2020
Available online 20 November 2020

Keywords:
CUDA Fortran
Ising spin-glass
Quantum annealers
Titan V GPU

ABSTRACT

We demonstrate how to compute the low energy spectrum for small ($N \leq 50$), but otherwise arbitrary, spin-glass instances using modern Graphics Processing Units or similar heterogeneous architecture. Our algorithm performs an exhaustive (i.e., brute-force) search of all possible configurations to select $S \ll 2^N$ lowest ones together with their corresponding energies. We mainly focus on the Ising model defined on an arbitrary graph. An open-source implementation based on CUDA Fortran and a suitable Python wrapper are provided. As opposed to heuristic approaches, ours is exact and thus can serve as a references point to benchmark other algorithms and hardware, including quantum and digital annealers. Our implementation offers unprecedented speed and efficiency already visible on commodity hardware. At the same time, it can be easily launched on professional, high-end graphics cards virtually at no extra effort. As a practical application, we employ it to demonstrate that the recent Matrix Product State based algorithm – despite its one-dimensional nature – can still accurately approximate the low energy spectrum of fully connected graphs of size N approaching 50.

© 2020 Published by Elsevier B.V.

1. Introduction

With increasing complexity and interconnectivity in the modern world, the ability to solve optimization problems becomes indispensable. Notwithstanding, these problems are fundamentally hard to resolve as they often require seeking over *enormous* spaces of possible solutions [1]. A notable example is the famous spin-glass problem encoded via the Ising model [2], where the low energy spectrum (the ground state in particular) is sought after. The importance of this system is reflected in the fact that many NP-complete [3] optimization problems (i.e. Karp's 21 problems [4]) can be mapped onto its Hamiltonian [5]. Furthermore, there is growing hardware support for many spin-glass based models [6–8]. These cutting edge technologies, when combined with classical neural networks [9], lead to quantum artificial intelligence [10]. A type of artificial intelligence believed to be powerful enough to simulate many-body quantum systems *efficiently*, which is a holy grail of modern physics [11].

The most promising ideas to overcome mathematical difficulties concerning classical optimization could rely on quantum computers [12]. In particular, on quantum annealers such as the D-Wave 2000Q chip [13]. In principle, such machines could solve variate of (hard) optimization problems (almost) "naturally" by finding low energy eigenstates [14]. However, current quantum annealers are extremely noisy and thus not powerful enough to tackle large scale optimization challenges [15,16]. In contrast, heuristic approaches, often offering superior performance, can *not* typically certify that the solution that has been found is, in fact, optimal [17,18]. Most heuristic solvers rely on strategies ranging from famous simulated annealing [19], branch and bound approaches [20] their chordal extension [18], various Monte Carlo methods [21] throughout dynamical systems simulations [22] to tensor network analysis [23].

In this work, we focus on yet another class of solvers, namely those that perform exact brute-force search [24]. The idea is to search the entire Hilbert space exhaustively to find configurations with the lowest energies. Such a search can be performed either in the probability or energy space [23]. For all classical Hamiltonians, where all their terms commute, this is essentially equivalent to the exact diagonalization. However, in contrast to the quantum case, the eigenvalue problem for classical models can be executed truly in parallel. An efficient implementation nonetheless is *not* trivial.

Although practical applications of such solvers are limited to small problem sizes ($N \leq 50$), they can solve the Ising model that is defined on an arbitrary graph. Moreover, with the exhaustive search, one can easily certify the output. All of these features are crucial for testing, benchmarking, and validating new methods [25], strategies, and paradigms (e.g., memcomputing [22]) for solving classical optimization problems [26]. It is

☆ The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author.
E-mail address: konrad.jalowiecki@smcebi.edu.pl (K. Jałowiecki).

https://doi.org/10.1016/j.cpc.2020.107728
0010-4655/© 2020 Published by Elsevier B.V.

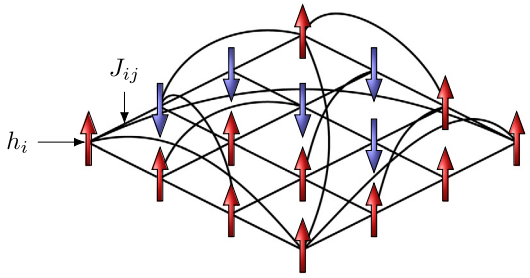


Fig. 1. An example of the Ising spin glass model (1). Here, J_{ij} correspond to weights of the edges of the graph and h_i are biases associated with the graph's nodes ($N = 16$). Physically, J_{ij} describe the interaction between spins s_i, s_j and h_i is the external magnetic field imposed on spin s_i. The picture also demonstrates possible spin encoding, with red arrows indicating assignment of $s_i = +1$ and blue ones indicating assignment of $s_i = -1$ [or $q_i = 0$ if QUBO (2) is used]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

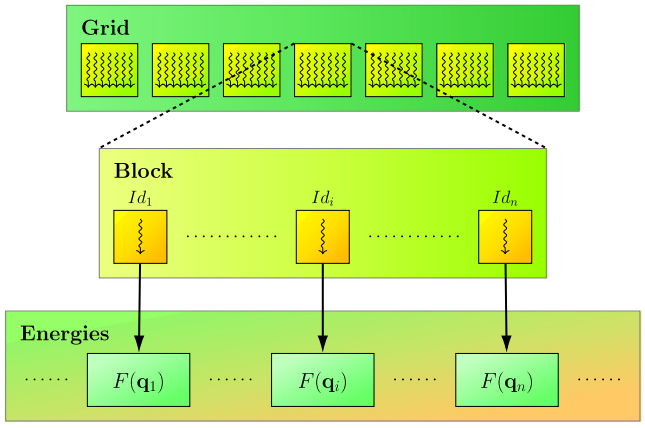


Fig. 2. Scheduling of the energy computation on the GPU. A CUDA program is executed by threads that are organized by blocks. Both the grid and blocks can form one, two or three dimensional structures. Our implementation uses a one dimensional grid structure, where the global thread index, Id_i, is converted into a state $\boldsymbol{q}$ with mapping $Id_i = (\boldsymbol{q})_2$, cf. Eq. (4). Next, each thread in each block computes its own energy, $F(\boldsymbol{q})$, according to Eq. (3). To fit into, often limited, GPU memory the computation is executed in carefully tailored chunks, cf. Eq. (5).

worth mentioning that brute-force approaches however limited can still serve as a reference point for today's quantum supremacy experiments [27,28].

Our implementation offers excellent flexibility and portability, as well as the significant efficiency and speed. Our solver can be executed on either CPU (Central Processing Unit) or GPU (Graphics Processing Unit) using Nvidia's CUDA (Compute Unified Device Architecture). The latter architecture is of particular importance due to its massive parallel capabilities [29,30]. Moreover, we provide a simple Python wrapper that allows users to access both architectures effortlessly [31].

Finally, we employ our solver to test the applicability of a particular tensor network ansatz – based on the Matrix Product State (MPS) – to optimization purposes of a fully connected graph of growing size. For a detailed description of this algorithm, we refer the reader to look at Supplementary Information in Ref. [23]. We have verified that indeed, such an ansatz, despite its inherited one-dimensional structure, can still successfully capture the low energy spectrum for tested graphs up to $N = 50$. This indicates that the MPS ansatz should still perform well also for much larger systems having a dominant quasi-one-dimensional nature. At the same time, sparse connections at long–range do *not* necessary exclude the applicability of the MPS approach.

2. Spin-glass problems

In this work, we mainly focus on the Ising Hamiltonian. However, our approach can easily be extended to include other *classical* spin-glass models [32,33]. To begin with, consider a simple undirected graph with N nodes (i.e., vertices) as the one drawn in Fig. 1. We assign a unique spin variable, $s_i \pm 1$ (blue and red arrows), to each node. Adjacent nodes labeled as i, j are coupled via interaction strength J_{ij}, which may be viewed as a weight of the edge connecting those two nodes. Additionally, for every spin, we associate a local magnetic field (bias) h_i interacting with it. Then the energy of such a system of spins is defined as

$$H(\boldsymbol{s}) = -\sum_{\langle i,j\rangle} J_{ij} s_i s_j - \sum_{i=1}^{N} h_i s_i, \tag{1}$$

where $\boldsymbol{s} := (s_1, \ldots, s_L)$. The first sum runs over all adjacent sites, which we denote here as $\langle i, j\rangle$.

In many practical applications, one is typically interested in finding a particular spin configuration, say $\boldsymbol{s}_0$, for which $H(\boldsymbol{s}_0)$ in Eq. (1) admits its minimum value. Such configuration is called the *ground* state. Naturally, states with energies above the ground state energy are called *excited* states. Finding the low energy spectrum (consisting of the ground state energy and a number $S \ll 2^N$ of excited states) of the Ising model (1) can also be formulated as a Quadratic Unconstrained Optimization Problem (QUBO). Namely,

$$F(\boldsymbol{q}) = -\sum_{\langle i,j\rangle} a_{ij} q_i q_j - \sum_{i=1}^{N} b_i q_i, \tag{2}$$

where $\boldsymbol{q} = (\boldsymbol{s} + 1)/2$ are *binary* variables whereas

$$a_{ij} = 4J_{ij}, \quad b_i = 2h_i - 2\sum_{\langle i,j\rangle} J_{ij}. \tag{3}$$

The energy offset reads $H(\boldsymbol{s}) - F(\boldsymbol{q}) = \sum_{i=1}^{N} h_i - \sum_{\langle i,j\rangle} J_{ij}$. Note, if a given q_i vanishes so does any product $q_i q_j$. Therefore, QUBO formulation (2) *effectively* reduces the number of multiplications almost by half in comparison to Eq. (1).

Despite its straightforward formulation, the problem of solving spin-glass instances can *not* be easily tackled using a brute force approach even for a modest number of spin variables. This is since the number of possible spin assignments grows exponentially with the number of nodes in the graph. For instance, when $N = 40$, the number of possible states is greater than the number of bits in a 32 GB memory chip. Already when $N = 64$, the size of the search space is greater than the estimated age of the Universe in seconds [34]. In fact, the problem of finding the ground state of the Ising model defined on an arbitrary graph is long known to be NP-hard [35]. This means, in particular, that even verifying if a given configuration minimizes the cost function (1) is difficult.

3. Description of the algorithm

A general idea underlying this work is to perform an exhaustive search over the whole state space, taking advantage of massive parallel capabilities of modern GPUs. This requires an efficient strategy to encoding all states, $\boldsymbol{q} = (q_1, q_2, \ldots, q_L)$, on a GPU. A naive approach would require storing an array of N integers, $q_i = 0, 1$, for each state $\boldsymbol{q}$. However, this would also lead to excessive use of memory and render this approach inefficient. As an optimal strategy, one should try to reuse information already

stored in the GPU memory. Therefore, in our algorithm, we take advantage of the following correspondence

$$\text{GPU thread index} = (\boldsymbol{q})_2, \tag{4}$$

where $k = (\boldsymbol{q})_2$ denotes the binary representation of an integer k (see Fig. 2). For instance, when there is $N = 8$ spins, one may associate

$$\text{thread index \#13} = (00001101)_2, \tag{5}$$

Theoretically, this strategy allows one to store $M = 2^{64} \sim 10^{19}$ states with no extra cost, limiting the system size to $N = 64$ spins. Nonetheless, this is more than the current architecture, based on the von Neumann paradigm of computation, can process in a reasonable time [36]. Indeed, we estimated that optimal search among 2^{64} states to extract the low energy spectrum consisting of $S = 10^2$ of them would take 821 *years* on an efficient Titan V GPU [37]. In comparison, systems of sizes $N = 32$, 49, 50 can be solved within 3.5 s, 7 and 14 days, respectively. A detailed benchmark is presented in Section 5.

One should stress that the fastest (as of 2018) supercomputer in the world—Summit—is equipped with $27\,648 > 2^{14}$ Nvidia Tesla V100 GPUs [38]. Therefore, "only" 2^{14} of them (processing chunks of size 2^{50} each, simultaneously) should be able to reduce the number of 821 years (for a single GPU tackling $N = 64$) substantially. Perhaps, maybe even down to a couple of months. Nevertheless, *a priori*, it is hard to estimate the exact numbers due to various communication bottlenecks. This interesting open problem is, however, beyond the scope of the current work.

In theory, one could first compute all $M = 2^N$ energies in parallel and only then select $S \ll 2^N$ lowest ones (and the corresponding states if needed). However, even with an efficient storage strategy, this approach quickly becomes impractical for large systems. It requires an exponentially increasing storage space to encode possible solutions. To overcome this problem, one could iterate over the solution space in manageable chunks, each time extracting the desired number of states [e.g., with the bucket select algorithm [39], for which the mean execution time scales linearly with the size of the input vector]. Sorting the energies is executed only in the final step. Since GPU threads and blocks are labeled in the same way for every chunk, an offset is required to correctly enumerate all states, i.e.,

$$\text{GPU thread index} + \text{offset} = (\boldsymbol{q})_2, \tag{6}$$

Note, the energy calculations are independent and thus can be performed in parallel. The overall parallel speedup is limited by the serial part (Amdahl's law [40]) consisting of the lowest energy states extraction and merging all local information into the global record. Also note that this idealized description assumes using number of threads equal to the processed chunk size. This restriction can be lifted using strided loops, see Section 4.2 for the details. Algorithm 1 in the below listing summarizes the underlying structure of our solver.

Algorithm 1 Searching $S \ll 2^N$ configurations (i.e. states) with the lowest energies defined in Eq. (1). The adjacency matrix, J_{ij}, and local magnetic fields, h_i, are provided.

```
k ← chunk_size_exponent
for i = 1 to 2^(N−k) do
  for j = 1 to 2^k do
    state_code ← j + (i − 1) · 2^k
    energies[j] ← energy(graph, state_code)
    states[j] ← state_code
  end for
  select_lowest(energies, states, num_st)
  if i == 1 then
    low_en[1: num_st] ←energies[1: num_st]
    low_st[1: num_st] ←states[1: num_st]
  else
    low_en[num_st + 1: 2 · num_st]
        ←energies[1: num_st]
    low_st[num_st + 1: 2 · num_st] ←states[1: num_st]
    select_lowest(low_en, low_st, num_st)
  end if
end for
sort_by_key(low_en, low_st, num_st)
```

4. Implementation details

4.1. Languages and technologies employed

The core components of our implementation has been written in modern Fortran 95/2003 [41], which we have chosen for its flexibility [42], extensive support for linear algebra [43], performance [44] and native support for CUDA technology [45]. To make our code easier to use, we have wrapped it in a Python package using the f2py [46] utility and numpy's fork of distutils package [47]. Whereas Fortran is widely used mostly for numerical simulations [48], Python is one of the most popular general-purpose programming language [49].

We have also incorporated the fast k–selection algorithm for GPU [39], and the Thrust library [50] into our solver for its parallel implementation of many standard methods such as finding the minimum and maximum of an array or partitioning thereof. The Thrust library is utilized both for the GPU implementation and for the pure CPU implementation with the OMP backend. In order to take advantage of various Thrust's functions, we have written several small C++ modules to bind them into the Fortran code. The source code of the entire package, together with the comprehensive documentation, can be found on GitHub [31].

The Python wrapper allows one to execute the algorithm both on the CPU and GPU. It was designed with simplicity in mind, and as such, its primary usage does *not* require any specialized knowledge. A basic understanding of the underlying optimization problem is enough, cf. the listing below. In particular, only the system definition (i.e., the graph or adjacency matrix) and the desired number of states needs to be provided by the user. Nevertheless, other parameters, including the chunk sizes, can also be passed to the wrapper (see Fig. 4).

Listing 1: Simple example of how to use the `ising` module

```
# import package
from ising import search

# adjacency matrix (problem definition)
graph = {(1,1):-1, (1,2):-0.2}
# solve the Ising model
solution = search(graph, num_states=4)

# shows the states and energies found
print(solution.energies)
print(list(solution.states))
```

On virtually all Linux platforms, it is possible to install the very basic version (i.e., with no GPU support) of our solver directly from the Python Package Index, by issuing

```
pip install ising
```
(7)

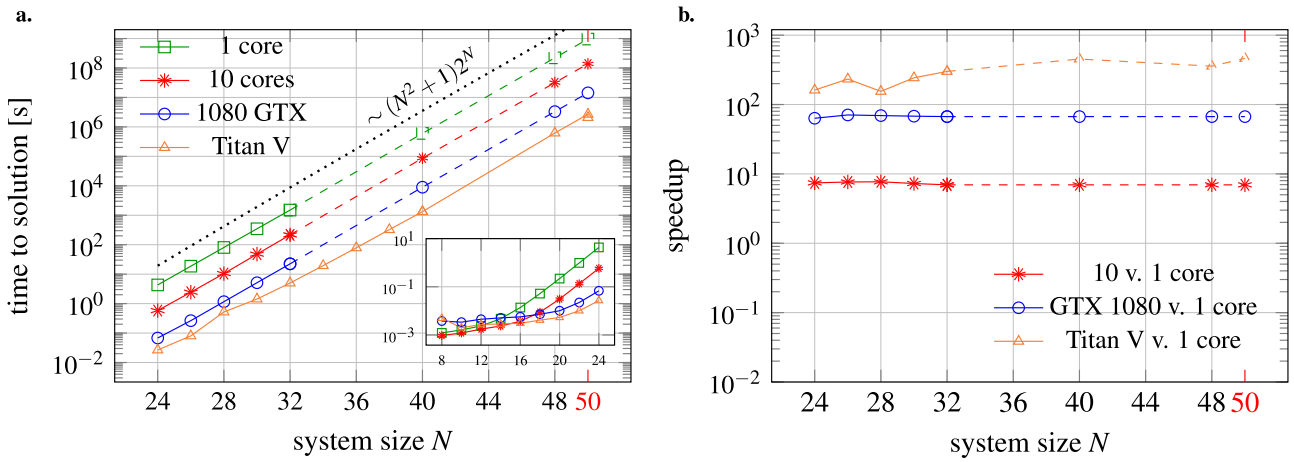


Fig. 3. a. Time to solution obtained by our solver for various system sizes N (the inset captures small systems). b. Calculated speedup in comparison to a single CPU core. The speedup is obtained as a ratio of respective execution times. The algorithm computes $S \ll 2^N$ low energy states in a single run (here $S = 10^2$). The problem instances (J_{ij}, h_i), on a fully connected graph, were randomly generated. The solid lines show actual measurements: 100 repetitions for each N except for $N = 48$ and $N = 50$ for which time to solution was calculated only once. The dashed lines represent *experimentally* estimated values for larger system sizes. The estimate is based on the time necessary to process a single chunk of data [of size $M = 2^{29}$ (CPU), and $M = 2^{27}$ (GTX 1080)]. This estimation is consistent with the scaling (9), which is depicted as a black dotted line. For $N \geq 24$ the overhead of parallel computations starts playing less important role and the execution time becomes linear in the state space size.

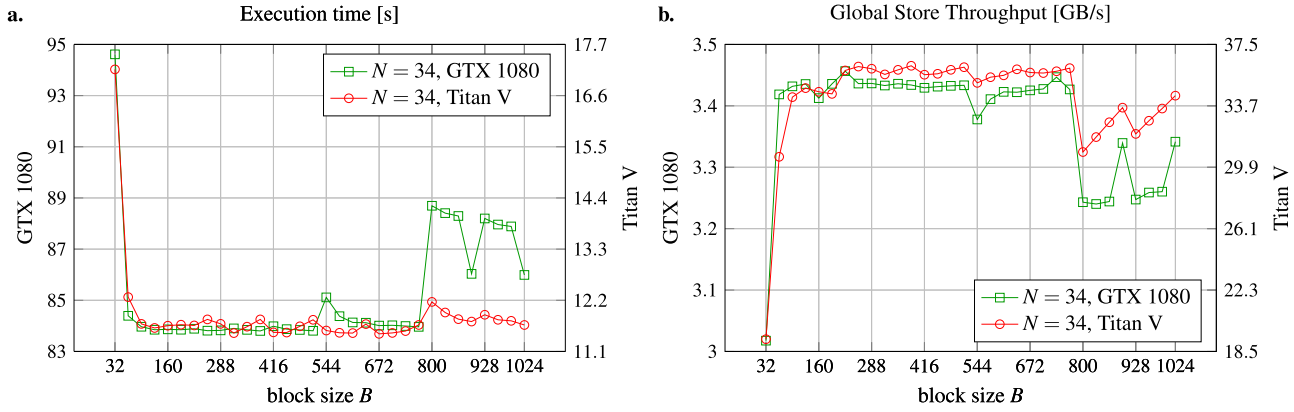


Fig. 4. a. The execution time (in seconds) of the proposed algorithm versus the CUDA block size, B, for a given system size, here $N = 34$. b. Global Store Throughput of the energy computing kernel. Here, like in the rest of this work, the number of blocks was equal to 2^{15}.

where `ising` is the name of the package. However, to assure full compatibility with modern GPUs, CUDA requires a custom build from source which can be initiated via

```
python install.py --usecuda
```
(8)

from the package source directory. For more details regarding custom installation, including CUDA and various Fortran compilers, we refer the reader to documentation [51]. Note, only PGI and IBM XLF support CUDA Fortran. Our package has only been tested using the former.

4.2. GPU execution scheduling and kernel implementation

Programming GPUs often poses a nontrivial endeavor. The core element of our implementation is the `compute_energies` kernel presented in Fig. 5. This implementation exhibited the best performance amongst all of the variants we tested. There are several factors that affect the performance of this kernel and might be easily overlooked. Perhaps the most important detail is using shared memory for caching the coefficient matrix, and using integers of size 1 for `bits` array.

Among other challenges, one has to design the grid on which kernels are launched [52]. Our kernel uses strided loops, i.e. a single thread computes several energies, each time increasing its index by the stride size equal to the total number of threads. Therefore, our implementation is not constrained to grids with total number of threads being equal to the current chunk size 2^g. We tested our kernel for various launch configurations and determined that for sufficiently large grids there is no clearly visible optimal one. For the purpose of benchmarks presented in this work, we used a grid of 2^{15} blocks with $B = 512$ threads. We would like to stress, however, that this configuration may need further (experimental) adjustment depending on the hardware, and the problem size.

4.3. Complexity analysis

Our algorithm performs an exhaustive search over the entire, exponentially large, state space in predefined chunks to find S lowest states (cf. Algorithm 1). Thus, unavoidably its time complexity has to be at least exponential in the system size N. Computing the energy (2) for a single state, $\boldsymbol{q}$, requires $\mathcal{O}(N^2)$ operations. The selection procedure executed on a data chunk of size 2^k, however, requires $\mathcal{O}(2^k)$ comparisons resulting in $\mathcal{O}[2^k(N^2 + 1)]$ operations. Finally, taking into account the total number of

```
attributes(global) subroutine compute_energies(Q, num_bits, sweep_size, energies, states, m)
  real(wp), intent(in), device :: Q(num_bits,num_bits)
  integer(ik), intent(in), value :: m, num_bits, sweep_size
  real(wp), intent(out), device :: energies(sweep_size)
  integer(ik), intent(out), device :: states(sweep_size)

  real(wp) :: en
  integer(1) :: i, j

  integer(ik) :: state_repr, idx
  real(wp), shared :: sQ(num_bits, num_bits)
  integer(1) :: bits(64)

  do idx = threadIdx%x, num_bits * num_bits, blockDim%x
     i = (idx -1) / num_bits + 1
     j = MOD(idx, num_bits) + 1
     sQ(j, i) = Q(j, i)
  end do

  call syncthreads()

  do idx=threadIdx%x + blockDim%x * (blockIdx%x-1), 2 ** sweep_size, blockDim%x * gridDim%x
     state_repr = idx + m
     states(idx) = state_repr
     do i = 0, num_bits - 1
        bits(num_bits-i) = IAND(state_repr, 1)
        state_repr = RSHIFT(state_repr, 1)
     end do
     en = 0.0_wp

     do i = 1, num_bits
        if(bits(i) == 1) then
           en = en - sQ(i, i)
           do j = 1+i,num_bits
              en = en - sQ(j, i) * bits(j)
           end do
        end if
     end do

     energies(idx) = en
  end do
end subroutine compute_energies
```

Fig. 5. Implementation of the energy computing kernel. Here, `Q` is the QUBO matrix (which is upper triangle), `energies` and `states` are output arrays. Finally, `m` is the offset that is common to all threads processing the current chunk of size $2^{\texttt{sweep\_size}}$. For the ith chunk, $\texttt{m} = (i-1)2^{\texttt{sweep\_size}} - 1$, so that each configuration in the state space is visited precisely once. The `wp` and `ik` denote the kind of the variables (both equal to 8 in all tests), which corresponds to double precision floating point numbers and 64-bit integers respectively. Typically for CUDA kernels (especially ones launched using one dimensional grid), `idx` refers to the global index of the current thread being executed within a block of size `blockDim%x` that is identified by its index `blockIdx%x`. Moreover, the local index `threadIdx%x` distinguishes threads in that block. The threads loop over the current chunk using stride of size `blockDim%x * gridDim%x`, which is the total number of threads in the launch configuration.

chunks, 2^{N-k}, and adding complexity of the final sorting procedure, $\mathcal{O}[S\log(S)]$, results in total complexity being

$$\mathcal{O}[2^N(N^2+1) + S\log(S)] = \mathcal{O}[2^N(N^2+1)]. \tag{9}$$

Therefore, essentially the solver's complexity behaves as $\mathcal{O}(2^N)$ which we also demonstrate experimentally in Section 5 (cf. Fig. 3). As one can see, the GPU implementation takes 20 seconds (GeForce 1080) and 3.5 s (Titan V) on average to solve the Ising problems with $N = 32$ spins. The same problem requires about 1500 seconds on average on a single CPU core. For GPU, the differences in solution times between single and double precision are close to 10% and are not reported on Fig. 3.

5. Benchmarks

5.1. GPU vs. CPU comparison

We have tested our algorithm on the following hardware:

- CPU: 10 Cores Intel® Core™ i7-6950X;
- GPU(1): Nvidia GeForce GTX 1080, 8 GB GDDR5 global memory, 2560 CUDA Cores;
- GPU(2): Nvidia Titan V, 12 GB HBM2 global memory, 5120 CUDA Cores.

For benchmarking purposes, we have executed our algorithm on a fully connected, $K = 100$ randomly generated (cf. Ref. [53, 54]), problem instances for systems up to $N = 50$ (on Titan V).

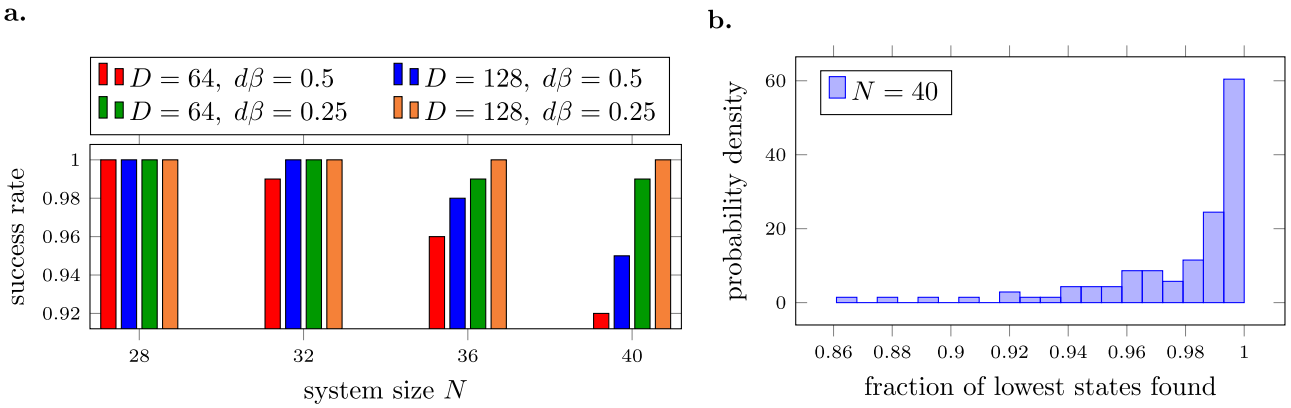


Fig. 6. Verification of the Matrix Product State (MPS) based algorithm introduced in Ref. [23]. a. Success rate is defined as the fraction of cases for which the MPS algorithm was able to find the ground state. All spin-glass instances were generated randomly on a fully connected graph of size N. Parameter D is the bond dimensions characterizing MPS tensors, and $d\beta$ denotes the increment of the inverse temperature, see the main text. b. Normalized histogram showing the percent of instances for which the MPS based algorithm was able to find a given number of configurations out of $S = 1000$ lowest ones. Here, we use $D = 128$, $d\beta = 0.25$ and in all panels $\beta = 1$.

For each instance, we have calculated the low energy spectrum consisting of $S = 10^2$ states in a single run. Typical results obtained with a high-end CPU (i7-6950X) and both a mid-class (GeForce 1080) and professional (Titan V) GPU are depicted in Fig. 3. We have also estimated time to solution *experimentally*, for larger systems (up to $N = 50$ spins) for which the low energy spectrum can be obtained in a reasonable time (i.e., one month) on Titan V. The estimate is based on the average time required to process a single chunk of data [of size $M = 2^{29}$ (CPU), and $M = 2^{27}$ (GTX 1080)]. Our measurements are consistent with the complexity analysis discussed in Section 4.3.

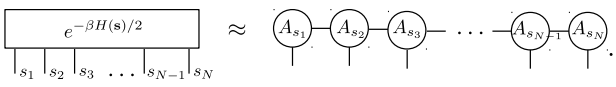
5.2. Validation of MPS algorithm

To demonstrate the capabilities of our solver, we employ it to benchmark a more sophisticated, *heuristic*, approach based on a Matrix Product States (MPS) technique (see Supplementary Material of Ref. [23] for details). Here, we are not interested in time to solution, but instead, we would like to investigate the accuracy of the latter. Heuristic algorithms can often solve large systems ($N \gg 50$). However, they cannot certify solutions.

With the MPS based algorithm one aims at approximating the Boltzmann distribution,

$$e^{-\beta H(\boldsymbol{s})/2} \approx A^{s_1} A^{s_2} \dots A^{s_L} = |\Psi(\beta)\rangle, \tag{10}$$

for a sufficiently large inverse temperature, β, where each A^{s_i} ($i = 1, 2 \dots, L$) is matrix of limited dimensions $\leq D$ (refereed to as the *bond dimensions*). The above approximation is usual depicted – using a network of tensors – as

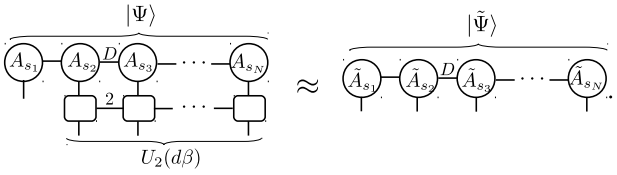


At each bond, one splits the system into two-halves. The *exact* decomposition would require the bond dimension D to grow exponentially with the number of spins in one half, interacting with spins in the second half (and arbitrary numerical precision). The limited bond dimension D reflects on the amount of entanglement/correlations (related to a given bipartition), which can be stored in a "quantum system" decomposed as MPS [55]—here we understand a "quantum system" as a superposition over all possible classical spin configurations. Having the approximation in Eq. (10) in the form of MPS, we can efficiently calculate any marginal and conditional probability (at the inverse temperature β) described by $|\Psi(\beta)\rangle$, and then systematically search for the most probable classical configurations (i.e., the ones with the smallest energies) using branch and bound strategy—building the most probable spin configurations one spin at the time.

Finally, to perform the search one needs to find $|\Psi(\beta)\rangle$, which is obtained by starting from $\beta = 0$ – for which the MPS decomposition $|\Psi(\beta = 0)\rangle$ is trivial – and then subsequently simulating the imaginary time evolution (i.e., the annealing). To that end, we apply the sequence of operators,

$$U_i(d\beta) = e^{-d\beta s_i(\sum_{j>i} J_{ij} s_j + h_i)/2}, \tag{11}$$

which amount to $\prod_{i=1}^{N} U_i(d\beta) = e^{-d\beta H(\boldsymbol{s})/2}$. Applying each gate (11) results in doubling of the affected bond dimensions. Moreover, applying all such operators would result in uncontrollable, exponential growth of the MPS matrices. However, the one-dimensional (and loop-free) structure of the MPS ansatz allows one to systematically, at each step, find its approximation, which effectively compresses the information and maintains the bond dimensions limited to D. The whole procedure can be graphically depicted as



While all the applied operators $U_i(d\beta)$ formally commute (independent of $d\beta$), due to the finite numerical precision and finite D, it is relevant to reach the final inverse temperature, β, gradually in a couple of consecutive steps, each with smaller $d\beta$. Otherwise, for larger $d\beta$, $U_i(d\beta)$ effectively act as projectors trapping the system in a local minima.

The question then becomes how well the MPS ansatz, which by construction is one-dimensional, is able to encode the structure of low energy spectrum for fully-connected graphs. In general, the bigger the system, the higher D necessary to faithfully capture the structure of low energy spectrum. We observe that already moderate D of 128 is enough to find all ground states for 100 considered instances, see Fig. 6a. The inverse temperature $\beta = 1$ is large enough to sufficiently zoom-in on the low energy states. At the same time, the importance of small enough time-step (here $d\beta = 0.25$) is clearly visible. It is also enough to recover most of the 1000 configurations with lowest energies for those instances, see Fig. 6b for $N = 40$. Note that the *exact*

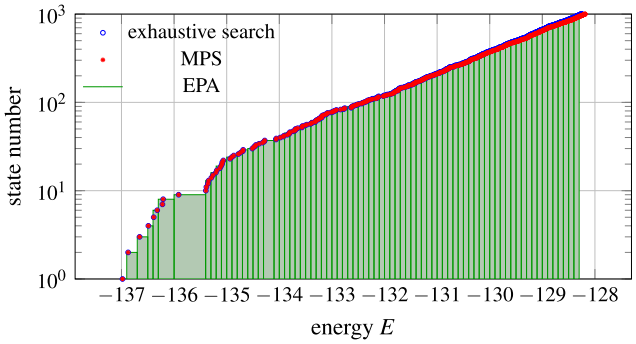


Fig. 7. Low energy spectrum of the Ising model (1) obtained with different algorithms and randomly generated instances of size $N = 50$. Here, the bond dimension for the MPS based algorithm $D = 128$ and the increment of the inverse temperature $d\beta = 0.125$ (cf. the main text or Ref. [23] for more details). We also depicted the approximated solutions obtained with the recent Monte Carlo based algorithm (Entropic Population Annealing), introduced in Ref. [56]. The data was provided by the authors of this paper.

MPS decomposition would require the bond dimension of $2^{N/2} = 2^{20}$. This demonstrate the magnitude of the compression of the relevant information encoded in MPS.

A typical lowest energy spectrum for $N = 50$ spins, and consisting of $S = 10^3$ states, is shown in Fig. 7. Therein, we have also incorporated an approximated low-energy spectrum obtained with the recent Monte Carlo based algorithm, introduced in [56], which can determine the density of states. The numerical data was provided to us by the authors of that paper.

6. Summary

We have demonstrated how to perform an exhaustive (brute-force) search in the solution space of the Ising spin-glass model [2] utilizing modern Graphics Processing Units [57]. Our algorithm can also be adapted for different heterogeneous architectures (e.g., Xeon Phi [58]). The Hamiltonian of this particular model encodes a variety of important optimization problems [5]. Moreover, this model has also been realized experimentally as a commercially available D-Wave quantum annealer [8].

Our implementation with CUDA Fortran [45] offers unprecedented speed and efficiency already visible on commodity hardware (e.g., GeForce 1080). Furthermore, it can be easily tuned for professional GPUs such as Titan V [37] virtually at no extra effort. To give an example, our algorithm, when tailored for the latter GPU, can extract the low energy spectrum (consisting of $N = 10^2$ states) in roughly 3.5 s for the spin system admitting $M = 2^{32} \sim 10^9$ different configurations. In comparison, a single CPU core takes (on average) 25 min to finish the same task (cf. Section 5 for detailed benchmark).

Admittedly, practical applications of brute-force algorithms are constrained to small problem sizes ($N \leq 50$). However, they can not only solve the spin-glass problems for arbitrary topologies and instances but also certify solutions [17,18,24]. These two features are crucial for developing and validating new methods and strategies for solving classical optimization problems [26]. We have explicitly exemplified this point by comparing our algorithm to a sophisticated recent Ising solver based on tensor network techniques [23]. In particular, we have demonstrated that despite its one-dimensional nature, the Matrix Product State ansatz is still able to approximate well the relevant part of the Boltzmann distribution for a fully connected graph of $N \leq 50$. Therefore, this suggests that the MPS algorithm should be superior for all problems having a dominant quasi-one-dimensional nature that allows for sparse connections to span the full problem.

Finally, to benefit the community, we have made our code publicly available as an open-source project [31]. Moreover, for those users who lack technical knowledge of Fortran or CUDA, we have provided an easy to install and use Python wrapper [31].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We appreciate fruitful discussions with Andrzej Ptok, Jerzy Dajka, and Piotr Gawron and Masoud Mohseni. We thank Paweł Wasiak for his valuable remarks regarding solver's documentation. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research. This work was supported by National Science Center (NCN, Poland) under projects 2015/19/B/ST2/02856 (KJ) 2016/20/S/ST2/00152 (BG) and NCN together with European Union through QuantERA ERA NET program 2017/25/Z/ST2/03028 (MMR). We acknowledge receiving Google Faculty Research Award 2017 (MMR) and 2018 (MMR and BG).

References

[1] S. Aaronson, Quantum Computing Since Democritus, Cambridge University Press, 2013.
[2] R. Harris, et al., Science 361 (6398) (2018) 162–165, 30002250.
[3] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1979.
[4] R. Karp, Complexity of Computer Computations, Plenum Press, 1972, pp. 85–103.
[5] A. Lucas, Front. Phys. 2 (2014) 5.
[6] Y. Yamamoto, et al., npj Quantum Inf. 3 (1) (2017) 49.
[7] M. Aramon, et al., 2018, Preprint at arXiv:1806.08815.
[8] J. King, et al., 2015, Preprint at arXiv:1508.05087.
[9] A. Krizhevsky, I. Sutskever, G.E. Hinton, Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, in: NIPS'12, Curran Associates Inc., 2012, pp. 1097–1105.
[10] B. Gardas, M.M. Rams, J. Dziarmaga, Phys. Rev. B 98 (2018) 184304.
[11] T.A. Elsayed, K. Mølmer, L.B. Madsen, Sci. Rep. 8 (1) (2018) 12704.
[12] R.P. Feynman, Caltech Eng. Sci. 23 (1960) 22–36.
[13] T. Lanting, et al., Phys. Rev. X 4 (2014) 021041.
[14] R. Orus, Ann. Phys. 349 (2014) 117–158.
[15] B. Gardas, S. Deffner, Sci. Rep. 8 (1) (2018) 17191.
[16] B. Gardas, J. Dziarmaga, W.H. Zurek, M. Zwolak, Sci. Rep. 8 (1) (2018) 4539.
[17] J. Czartowski, K. Szymański, B. Gardas, Y. Fyodorov, K. Życzkowski, 2018, Preprint at arXiv:11812.09251.
[18] F. Baccari, C. Gogolin, P. Wittek, A. Acín, 2018, Preprint at arXiv:1808.01275.
[19] C. Cook, H. Zhao, T. Sato, M. Hiromoto, S.X.-D. Tan, 2018, arXiv:1807.10750.
[20] F. Rendl, G. Rinaldi, A. Wiegele, Math. Program. 121 (2) (2008) 307.
[21] I. Hen, Phys. Rev. E 96 (2017) 022105.
[22] F. Sheldon, F.L. Traversa, M.D. Ventra, 2018, Preprint at arXiv:1810.03712.
[23] M.M. Rams, M. Mohseni, B. Gardas, 2018, Preprint at arXiv:1811.06518.
[24] M.J.H. Heule, O. Kullmann, Commun. ACM 60 (8) (2017) 70–79.
[25] T. Leleu, Y. Yamamoto, P.L. McMahon, K. Aihara, Phys. Rev. Lett. 122 (2019) 040607.
[26] S. Mandra, H.G. Katzgraber, Quantum. Sci. Technol. 3 (4) (2018) 04LT01.
[27] F. Arute, et al., Nature 574 (7779) (2019) 505–510.
[28] E. Pednault, et al., 2019, arXiv:1910.09534.
[29] M. Januszewski, A. Ptok, D. Crivelli, B. Gardas, Comput. Phys. Commun. 192 (2015) 220–227.
[30] B. Gardas, A. Ptok, J. Comput. Phys. 366 (2018) 320–326.
[31] https://github.com/dexter2206/ising. 2019. accessed: 2019-01-27.
[32] F.Y. Wu, Rev. Modern Phys. 54 (1982) 235–268.
[33] Z. Liu, S.P. Rodrigues, W. Cai, 2018, Preprint at arXiv:1710.04987v1.
[34] P.A.R. Ade, et al., Astron. Astrophys. 594 (2016) A13.
[35] F. Barahona, J. Phys. A: Math. Gen. 15 (10) (1982) 3241.
[36] J. Backus, Commun. ACM 21 (8) (1978) 613–641.
[37] https://www.nvidia.com/en-gb/titan/titan-v/. 2018. accessed: 2019-01-27.
[38] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. 2019. accessed: 2019-09-24.

[39] T. Alabi, et al., J. Exp. Algorithm. 17 (2012) 4.2:4.1–4.2:4.29.
[40] M.D. Hill, M.R. Marty, Computer 41 (7) (2008) 33–38.
[41] S.J. Chapman, Fortran 95/2003 for Scientists & Engineers, McGraw-Hill, 2007.
[42] L. Rossi, J. Berzosa-Molina, D.M. Stam, Astron. Astrophys. 616 (2018) A147.
[43] E. Wang, et al., High-Performance Computing on the Intel® Xeon Phi ™, Springer, 2014, pp. 167–188.
[44] https://julialang.org/benchmarks/. 2019. accessed: 2019-01-27.
[45] M. Fatica, G. Ruetsch, CUDA Fortran for Scientists and Engineers, Elsevier, 2014.
[46] https://docs.scipy.org/doc/numpy-1.15.0/f2py/index.html, 2018.
[47] https://docs.scipy.org/doc/numpy-1.15.0/reference/distutils.html. 2018. accessed: 2018-10-23.
[48] M.L. Wall, L.D. Carr, New J. Phys. 14 (2012) 125015.
[49] https://insights.stackoverflow.com/survey/2018/. 2018. accessed: 2019-01-27.
[50] https://thrust.github.io/. 2015. accessed: 2019-01-27.
[51] https://ising.readthedocs.io/en/latest/. 2019. accessed: 2019-01-27.
[52] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, first ed., Addison-Wesley Professional, 2010.
[53] J. Marshall, V. Martin-Mayor, I. Hen, Phys. Rev. A 94 (2016) 012320.
[54] F. Hamze, et al., Phys. Rev. E 97 (2018) 043303.
[55] D. Jaschke, M.L. Wall, L.D. Carr, Comput. Phys. Comm. 225 (2018) 59–91.
[56] L. Barash, J. Marshall, M. Weigel, I. Hen, New J. Phys. 21 (7) (2019) 073065.
[57] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, first ed., Addison-Wesley Professional, 2005.
[58] I.A. Surmin, et al., Comput. Phys. Comm. 202 (2016) 204–210.