Simulated Quantum Annealing With Path-Integral Quantum Monte Carlo

Hadayat Seddiqi [*]

April 14, 2015

Abstract

In these notes I will describe how to implement a simulation of quantum annealing using the path-integral quantum Monte Carlo algorithm. I also include a small reference for an open-source code at the end.

1 Introduction

Quantum annealing (QA) is described as a heuristic version of an adiabatic quantum computing (AQC) algorithm with the beginning and final Hamiltonians described by the Ising spin-glass Hamiltonian in a transverse magnetic field. The Hamiltonian is given by

$$H_{ising}(t) = -\sum_{i,j}^{N} J_{ij}\sigma_z^i\sigma_z^j - \Gamma(t)\sum_{i}^{N}\sigma_x^i \tag{1}$$

where σ_x^i and σ_z^i are the ith Pauli operators in the x and z spin bases, respectively. N is the total number of spins, J includes the coupling and local field strengths in an adjacency matrix, and the sum can be over an arbitrary coupling graph. The annealing process is implemented by starting with a large Γ at $t = 0$ (where large essentially means greater than $|J|$, i.e., the order of magnitude of the couplings) and slowly decreasing it until the corresponding σ_x term is negligible.

There are several methods for simulating this process, but in these notes we focus on using the path-integral quantum Monte Carlo (PIQMC) approach. This involves mapping the d-dimensional quantum system into a $(d+1)$-dimensional classical system, where we reference the extra degree of freedom as the "Trotter dimension". Eq. (2) can then be transformed to

$$H_{d+1} = -\sum_{k=1}^{P}\left(\sum_{i,j}^{N} J_{ij} z_i^k z_j^k + J_\perp \sum_{i}^{N} z_i^k z_i^{k+1}\right) \tag{2}$$

where we use z_i to denote a classical spin, P for the number of replicas of the original Ising system, and $J_\perp$ for coupling the spins of each replica in a 1-dimensional Ising chain (which is typically taken to have periodic boundaries). Fig. (4.4) shows a nice visualization. $J_\perp$ is given by

$$J_\perp = -\frac{PT}{2}\ln\tanh\frac{\Gamma}{PT} \tag{3}$$

[*]Contact: hadsed@gmail.com

where T is the ambient temperature. These expressions are derived in [4].

2 Implementation

We'll start the implementation discussion with the initialization procedure. An appropriate method is to do a short preannealing using simulated annealing (SA) on a random spin configuration and feed that into the PIQMC program. We copy the preannealed configuration over all P Trotter slices. This fulfills the first boundary condition of the quantum path integral, i.e., that all paths start from the same intial point. Martonak et al [4] assume this is practical only if PT is not too small so that the system can attain thermal equilibrium with temperature $PT \geq |J|$. Then, for each decrease in Γ, we do one Monte Carlo update step on each of the PN spins in the system until Γ is sufficiently small (but still non-zero).

It is worth noting that in the beginning, when Γ is large, the different Trotter replicas are highly decoupled from each other and become increasingly coupled to each other as the annealing process steps forward in time. This is an attempt to fulfill the second boundary condition, namely that all paths should end at the same point in configuration space.

Each spin update step follows the Metropolis rule, given as

$$P(z, z') = \min\{1, e^{(E(z)-E(z'))/T}\} \tag{4}$$

for an attempted spin flip $z \to z'$. We rewrite the above implementation description in terms of pseudocode in Algorithm (1). Note that the Metropolis update for the quantum piece does depend on the magnetic field strength through the energy function, but we neglect to make this explicit in the function arguments for the update function for conciseness.

3 Optimizations

3.1 Non-interacting Domains

In Algorithm 1, the Metropolis update rule requires two energy calculations of the entire system, i.e., summing over the entire (2+1)-dimensional system at each attempted spin update. This can be simplified by noticing that the energy difference for a particular spin only depends on the subsystem including the spin to be updated and its nearest neighbors. In mathematical terms, we have the spin flip probability,

$$P(z, z') = \exp\frac{E(z) - E(z')}{T} = \exp\frac{\big(E_{sub}(z) + E_{rest}(z)\big) - \big(E_{sub}(z') + E_{rest}(z')\big)}{T} \tag{5}$$

where E_{sub} and E_{rest} are the subsystem and rest of the system energy contributions. Clearly $E_{rest}(z)$ and $E_{rest}(z')$ are the same and will cancel, leaving only the difference of energies in the subsystems. Picking a particular spin ω to flip, we can write

$$\exp\frac{E_{sub}(z) - E_{sub}(z')}{T} = \exp\left(-\frac{1}{T}\sum_{j\in N(\omega)} J_{\omega j} z_j (z_\omega - z'_\omega)\right) \tag{6}$$

Algorithm 1 Sketch of a PIQMC implementation for simulating QA

```
 1: $N \leftarrow$ number of spins in $H_{ising}$
 2: $P \leftarrow$ number of Trotter slices
 3: $T \leftarrow$ ambient temperature (at end of preannealing)
 4: $T_{pre}, T_{step} \leftarrow$ preannealing starting temperature, stepsize to define preannealing schedule
 5: $\Gamma_{start}, \Gamma_{end} \leftarrow$ starting and final magnetic field strengths
 6: $\Gamma_{step} \leftarrow$ amount to decrease magnetic field during QA
 7: $\mathbf{z} \leftarrow \{z_0, z_1, \ldots, z_{N-1}\}$, random initial configuration
 8:
 9: function METROPOLIS($\mathbf{s}, j, \tau$)                      ▷ define spin update rule
10:     $\mathbf{s}' \leftarrow$ configuration after the spin $s_j$ is flipped
11:     $\Delta \leftarrow E(\mathbf{s}) - E(\mathbf{s}')$                  ▷ quantum energy includes field term
12:     if $\Delta$ is positive or $e^{\Delta/\tau} > U_{random}$ then
13:         return $\mathbf{s}'$
14:     else
15:         return $\mathbf{s}$
16:
17: for $t$ in $(T_{pre} : T : T_{step})$ do                          ▷ preannealing
18:     for $z_i$ in $\mathbf{z}$ do
19:         $\mathbf{z} \leftarrow$ METROPOLIS($\mathbf{Z}, i, t$)
20:
21: $\mathbf{Z} \leftarrow \{\mathbf{z}_0, \ldots, \mathbf{z}_k, \ldots, \mathbf{z}_{P-1}\}$                ▷ store $P$ copies of $\mathbf{z}$
22:
23: for $\gamma$ in $(\Gamma_{start} : \Gamma_{end} : \Gamma_{step})$ do        ▷ simulate quantum annealing
24:     for $Z_i^k$ in $\mathbf{Z}$ do
25:         $\mathbf{Z} \leftarrow$ METROPOLIS($\mathbf{Z}, i + k, T$)
26:
27: $\mathbf{Z}_{solution} \leftarrow$ lowest energy replica amongst the $P$ Trotter slices
```

where $N(\omega)$ denotes the neighboring spins of z_ω. Substituting $z_\omega - z'_\omega$ for $2z_\omega$,

$$P(z, z') = \exp\left(-\frac{2}{T}\sum_{j\in N(\omega)} J_{\omega j} z_j z_\omega\right). \tag{7}$$

This means that for calculating a spin flip in H_{d+1} we only need to calculate six terms in the sum to obtain the Boltzmann probability instead of considering the entire system. This will be useful for doing parallel spin updates as we will see in the following sections.

3.2 Parallel Spin Updates

Since we can calculate relative energies by only looking at the subsystem local to the particular spin to be updated, we can update many spins in parallel at once. This violates the detailed balance condition but preserves global balance. (In the case of detailed balance, the transition probability from state a to b is the same as from b to a, but in global balance this is only true for the total probability *flux*, see [1] at the end of section 1.) For these parallel updates, we want to pick spins that are decoupled from each other, so a checkerboard-style scheme is used where we skip every other spin in each dimension to update and then shift once in each dimension for subsequent updates. See Figs. (4.4, 4.4) for a visualization (taken from [5]). For more complicated graphs, we must choose a more complicated update graph but the only requirement is that parallel updates happen only with spins that are not directly coupled.

4 PIQMC Software

I have written a PIQMC software package in Python (with C extensions) which can be found at `https://github.com/hadsed/pathintegral-qmc`. In this section I will detail the current API (as it is on April 14th, 2015). There are two main parts to the code, one for a classical simulated annealing algorithm and another for PIQMC. The two relevant files are `piqmc/sa.pyx` and `piqmc/qmc.pyx`, respectively. `piqmc/tools.pyx` includes helper functions for setting up problems and aiding in post-processing. We will describe the contents of the SA and QMC files with some reference to the helper functions in the following sections, but first a brief tutorial on installation.

4.1 Installation

Installing the software is very straightforward. Simply clone the GIT repository, navigate to its root directory, and run `sudo python setup.py develop`. You may also write `install` instead of `develop`, but with the latter you can make changes to the source and run the same command again to recompile. The code has only been tested with Python 2.7. Required packages include `NumPy`, `SciPy`, `Cython`, and `setuptools`.

Once you have installed the code, you should run the tests by typing in `nosetests` into the command line while in the root directory.

You can navigate to `examples/` and run any of the example scripts. All of them run fairly quickly. Some require extra packages, like `mpi4py`, `Matplotlib`, and `PyCUDA`.

4.2 Simulated Annealing

`piqmc/sa.pyx` currently includes a classical Ising energy function useful for evaluating the energy of a given configuration as well as six different annealing functions: `Anneal()`, `Anneal_dense()`, `Anneal_parallel()`, `Anneal_multispin()`, `Anneal_bipartite()`, and `Anneal_cuda()`. We will describe a code example before explaining each annealing method in detail.

We will base our examples on the 8-qubit diamond graph problem given by Boixo et al [2]. The problem specification is given in `examples/ising_instances/boixo.txt` as a 3-column text file where the first two columns denote spin indices and the last column denotes the coupling value.

First, we must import some required packages:

```python
import numpy as np
import scipy.sparse as sps
import piqmc.sa as sa
import piqmc.tools as tools
```

The first two lines are required external packages for the library (NumPy and SciPy), and the last two lines include the relevant modules with the functions we want to use. We should define some variables for our example simulation:

```python
# total number of spins
nspins = 8
# number of steps in annealing schedule
tannealingsteps = 5
# number of sweeps for each annealing step
tsweeps = 1
# initial temperature
tempstart = 3.0
# final temperature
tempend = 0.01
# random number generator, can be seeded with an integer argument
rng = np.random.RandomState()
```

Now we read in the text file that defines the Ising problem:

```python
# filename
fname = 'ising_instances/boixo.txt'
# load in data into a numpy array
loaded = np.loadtxt(fname)
# initialize a ``dictionary of keys'' sparse matrix type
isingJ = sps.dok_matrix((nspins,nspins))
# add to this matrix row by row
for (i,j,val) in loaded:
    isingJ[i-1,j-1] = val
```

The convention assumed above is that the text file starts spin labels from one, but in numpy arrays we start with zero. Any local field bias terms should be placed on the diagonal of this matrix and will be extracted internally by the library functions. To save memory, the matrix should be upper-triangular. The method above does not need to be followed exactly. As long as we end up with a coupling matrix in the form of a scipy sparse matrix, everything else will work properly. For example, the tools module includes a helper function for generating a random square 2D Ising problem:

```
isingJ = tools.Generate2DIsingInstance(4, rng)
```

which will give back a DOK sparse matrix. Other functions are also included for different graphs.

Now we'll define the annealing schedule. It doesn't matter how we do this as long as we end up with a one-dimensional numpy array that lists a series of temperature values.

```
tschedule = np.linspace(tempstart, tempend, tannealingsteps)
```

We also want to initialize a random starting state:

```
spinvec = np.array([ 2*rng.randint(2)-1 for k in range(nspins) ],
                   dtype=np.float)
```

Please note the data types. Since the annealing functions are written as C extensions, they expect arrays of a particular type (that is, numpy floats). Python lists cannot be passed through since they are not equivalent to C arrays, hence why we wrap our lists with `np.array()` or `np.asarray()` functions.

Now we begin our discussion of the different annealing methods. First, we will show how to use `Anneal()`.

```
maxdegree = 4
neighbors = tools.GenerateNeighbors(nspins, isingJ, maxdegree)
sa.Anneal(tschedule, tsweeps, spinvector, neighbors, rng)
```

There's a fair bit to explain here. The first line takes the number of spins, the coupling matrix, and the maximum degree of any spin in the Ising graph as arguments and produces a special data structure that makes spin update calculations faster, which we call the "neighbor array". It is a 3D matrix whose first dimension is equal in size to the number of spins, its second dimension indexing all of its neighbors (which will be equal to the maximum degree argument supplied above), and its third dimension having of length two, the first for the actual index of the neighboring spin and the second for its coupling value.

A few small code snippets may clarify this further. Let's say we have a spin k and we want to print out all of its neighbors. We can do the following:

```
for nb in neighbors[k]:
    print("Coupling value between spin %i and %i: %f" % k, nb[0], nb[1])
```

Some spins in the graph may not have a neighbors count equal to the maximal degree. In that case, to keep with the structure of the array, we add a spin index for zero (or some other random spin) and assign its coupling value to zero. For highly irregular graphs, this will lead to extra wasted loop iterations in the annealing function.

The annealing function will update the spin vector array in-place. We can use some helper functions to help see what is happening by modifying the snippet to read

```
maxdegree = 4
neighbors = tools.GenerateNeighbors(nspins, isingJ, maxdegree)
print("Starting state: ", getbitstr(spinvector))
print("Starting energy: ", sa.ClassicalIsingEnergy(spinvector, isingJ))
sa.Anneal(tschedule, tsweeps, spinvector, neighbors, rng)
print("Final state: ", getbitstr(spinvector))
print("Final energy: ", sa.ClassicalIsingEnergy(spinvector, isingJ))
```

where we can define a function **getbitstr()**

```
def getbitstr(vec):
    """ Return bitstring from spin vector array. """
    return reduce(lambda x,y: x+y,
                  [ str(int(k)) for k in tools.spins2bits(vec) ])
```

This annealing function should not be used if the number of sweeps and annealing steps is small since generating the neighbor array can be somewhat time consuming. It is possible to generate the neighbor array and save it to a file, which will be much faster on subsequent runs since loading from a file will be more efficient for very large problems.

Here is the entire code, for reference:

```
def getbitstr(vec):
    """ Return bitstring from spin vector array. """
    return reduce(lambda x,y: x+y,
                  [ str(int(k)) for k in tools.spins2bits(vec) ])

# total number of spins
nspins = 8
# number of steps in annealing schedule
tannealingsteps = 5
# number of sweeps for each annealing step
tsweeps = 1
# initial temperature
tempstart = 3.0
# final temperature
tempend = 0.01
# random number generator, can be seeded with an integer argument
rng = np.random.RandomState()
# maximal degree of the Ising graph
maxdegree = 4
# filename
fname = 'ising_instances/boixo.txt'
# load in data into a numpy array
loaded = np.loadtxt(fname)
# initialize a ``dictionary of keys'' sparse matrix type
isingJ = sps.dok_matrix((nspins,nspins))
# add to this matrix row by row
for (i,j,val) in loaded:
    isingJ[i-1,j-1] = val
# annealing schedule
tschedule = np.linspace(tempstart, tempend, tannealingsteps)
# initial random state
spinvec = np.array([ 2*rng.randint(2)-1 for k in range(nspins) ],
                   dtype=np.float)
# generate neighbor array
neighbors = tools.GenerateNeighbors(nspins, isingJ, maxdegree)
# look at the initial state and its energy
print("Starting state: ", getbitstr(spinvector))
print("Starting energy: ", sa.ClassicalIsingEnergy(spinvector, isingJ))
# anneal in-place
sa.Anneal(tschedule, tsweeps, spinvector, neighbors, rng)
# look at the outputs
print("Final state: ", getbitstr(spinvector))
print("Final energy: ", sa.ClassicalIsingEnergy(spinvector, isingJ))
```

We can replace `Anneal()` with `Anneal_dense()` if we have limited memory (the neighbor array introduces larger memory requirements for a decrease in calculation time), if the coupling matrix is too dense for the optimization to make sense, or if the problem is small enough for it not to matter. In the dense annealing function, the inner loop for calculating the energy difference for each spin cycles through all other spins instead of just direct neighbors. In the summary code above for `Anneal()` we can replace lines 35 and 40 with a single call:

```
sa.Anneal_dense(tschedule, tsweeps, spinvector, isingJ.todense(), rng)
```

where we replace the reference to the neighbor array with the coupling matrix. Note that we have converted the sparse matrix to a dense one. This is required since the sparse matrix type cannot be expressed in terms of native C types.

The rest of the annealing functions are under development and shouldn't necessarily be used for any other purpose. I will give a brief explanation for each.

`Anneal_parallel()` is a simple extension of `Anneal()` which uses OpenMP to create worker threads that parallelize over the loop over spins for each sweep. For small test problems, the overhead of threads actually outweighs any real benefit of parallelism, so this method should not be used.

`Anneal_multispin()` is an extension of `Anneal()` which uses bit-packing and bitwise operators to run 64 simultaneous SA runs. While this makes calculations very fast, it is unclear if it is truly beneficial since the internal packing and unpacking loops can consume a bit more time. It hasn't been tested extensively, but it is at least as fast as the default annealer if looped over 64 times.

`Anneal_bipartite()` is meant to be an efficient dense annealing routine for bipartite Ising graphs. Occasionally it may produce suboptimal states and it is unclear why this happens.

`Anneal_cuda()` is a first attempt at writing an annealing function with PyCUDA. It is not even close to being done and should be completely ignored.

4.3 Quantum Annealing

Using `piqmc/qmc.pyx` is not too unlike its classical counterpart. We will use the Boixo example from the previous section to illustrate how to use the QMC methods. Included in `piqmc/qmc.pyx` are the following functions: `QuantumAnneal` and `QuantumAnneal_parallel`, the latter being the OpenMP-parallelized version of the former. Like in the SA case, it does not outperform the default annealer and should not be used except for development purposes.

First, we must import the QMC module:

```
import numpy as np
import scipy.sparse as sps
import piqmc.qmc as qmc
import piqmc.tools as tools
```

We define some initial parameters which are slightly different from the SA case:

```
# total number of spins
nspins = 8
# number of steps in annealing schedule
annealingsteps = 5
# number of sweeps for each annealing step
```

```
sweeps = 1
# ambient temperature
temp = 0.01
# initial magnetic field
fieldstart = 1.5
# final magnetic field
fieldend = 1e-8
# number of Trotter slices
trotterslices = 20
# random number generator, can be seeded with an integer argument
rng = np.random.RandomState()
```

We used a linear schedule before, but perhaps an exponential schedule is more realistic or useful:

```
rexp = (fieldend/fieldstart)**(1./annealingsteps)
expsched = np.array([ fieldstart*rexp**k
                      for k in xrange(annealingsteps) ])
```

As before, we need to initialize a random starting state:

```
spinvec = np.array([ 2*rng.randint(2)-1 for k in range(nspins) ],
                   dtype=np.float)
```

We can use this state and do a small pre-annealing run by doing SA as done in the last section, but let's not complicate things. We need starting configurations for each Trotter slice:

```
configurations = np.tile(spinvector, (trotterslices, 1)).T
```

which creates a matrix whose columns correspond to the individual replicas (the final shape of this matrix is `(nspins, trotterslices)`). We will assume that the coupling matrix has been created already (see previous section for details). Now we create the neighbor array and run the annealer:

```
maxdegree = 4
neighbors = tools.GenerateNeighbors(nspins, isingJ, maxdegree)
qmc.QuantumAnneal(expsched, sweeps, trotterslices, temp,
                  nspins, configurations, neighbors, rng)
```

Notice that the neighbor array is no different for the quantum case. Viewing the individual energies and states of the replicas works similarly to the SA case, except that columns must be extracted individually (or looped over) before putting the states through the `ClassicalIsingEnergy()` function. A simple snippet to find the minimum energy configuration might look like:

```
minenergy, minconf = np.inf, []
for col in configurations.T:
    candidate = sa.ClassicalIsingEnergy(col, isingJ)
    if candidate < minenergy:
        minenergy = candidate
        minconf = col
print("PIQA energy:", minenergy)
```

Putting it all together:

```
import numpy as np
import scipy.sparse as sps
import piqmc.sa as sa
```

```python
import piqmc.qmc as qmc
import piqmc.tools as tools

# total number of spins
nspins = 8
# number of steps in annealing schedule
annealingsteps = 50
# number of sweeps for each annealing step
sweeps = 1
# ambient temperature
temp = 0.01
# initial magnetic field
fieldstart = 1.5
# final magnetic field
fieldend = 1e-8
# number of Trotter slices
trotterslices = 20
# random number generator, can be seeded with an integer argument
rng = np.random.RandomState()
# maximum degree of Ising graph
maxdegree = 4
# filename
fname = 'ising_instances/boixo.txt'
# load in data into a numpy array
loaded = np.loadtxt(fname)
# initialize a ``dictionary of keys'' sparse matrix type
isingJ = sps.dok_matrix((nspins,nspins))
# add to this matrix row by row
for (i,j,val) in loaded:
    isingJ[i-1,j-1] = val
# define annealing schedule
rexp = (fieldend/fieldstart)**(1./annealingsteps)
expsched = np.array([ fieldstart*rexp**k
                      for k in xrange(annealingsteps) ])
# initialize random state
spinvec = np.array([ 2*rng.randint(2)-1 for k in range(nspins) ],
                   dtype=np.float)
# copy over for each Trotter replica
configurations = np.tile(spinvector, (trotterslices, 1)).T
# get neighbor array
neighbors = tools.GenerateNeighbors(nspins, isingJ, maxdegree)
# anneal
qmc.QuantumAnneal(expsched, sweeps, trotterslices, temp,
                  nspins, configurations, neighbors, rng)
# output minimal energy replica
minenergy, minconf = np.inf, []
for col in configurations.T:
    candidate = sa.ClassicalIsingEnergy(col, isingJ)
    if candidate < minenergy:
        minenergy = candidate
        minconf = col
print("PIQA energy:", minenergy)
```

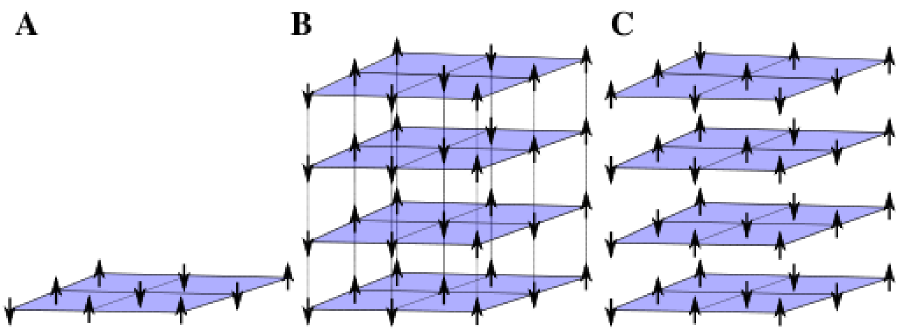

Figure 1: **(A)** shows the original quantum mechanical Ising model, **(B)** shows the Trotter discretization with P number of replicas in the vertical direction (spins are correlated with nearest neighbors on this 2D lattice and with their neighboring replica spins in 1D only), and **(C)** shows many decoupled replicas as might be used in simulated annealing. This would make for a fair comparison between QA and SA in terms of computational complexity and solution quality. Figure is taken from [3].

4.4 Further Examples

Many more examples can be found in the `examples/` directory. There are examples that test almost all of the functionality in the library (which the tests should be doing, but that hasn't caught up to the most recent development yet). A particularly nice example is `examples/hopfield8.py`, which runs an 8-neuron Hopfield network with 3 memories. If you have `Matplotlib` installed, you should see three plots that show the populations of both SA and QMC runs with the partition function graphed at the bottom. This is a useful tool for comparison of the two methods for smaller problem sizes. Another useful example is `examples/spinglass32_mpi.py` which simulates an exactly solved instance of a 1024-qubit 2D square lattice Ising model by executing MPI worker processes for each sample run. This simple wrapper script demonstrates how to use trivial parallelism to run many samples using `mpi4py`. Other examples showcase the OpenMP and multispin encoding routines, a basic library profiling script using `cProfile` and `pstats`, and more recent problems from the adiabatic quantum computing literature.

References

[1] B.J. Block and T. Preis. Computer simulations of the ising model on graphics processing units. *The European Physical Journal Special Topics*, 210(1):133–145, 2012.

[2] Sergio Boixo, Tameem Albash, Federico M. Spedalieri, Nicholas Chancellor, and Daniel A. Lidar. Experimental signature of programmable quantum annealing. *Nature Comm.*, 4:2067, 2013.

[3] B. Heim, T. F. Rønnow, S. V. Isakov, and M. Troyer. Quantum versus Classical Annealing of Ising Spin Glasses. *ArXiv e-prints*, November 2014.

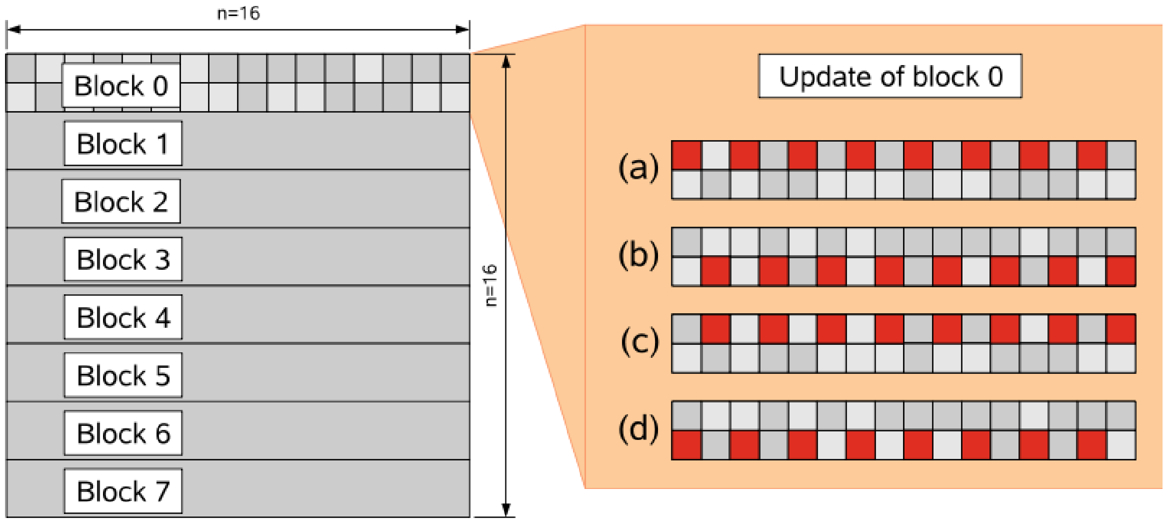

Figure 2: This figure, taken from [5], shows the assignment of checkerboard-style spin indices to GPU blocks, where the letters denote threads in the block. Only the indexing scheme (red blocks) is important.

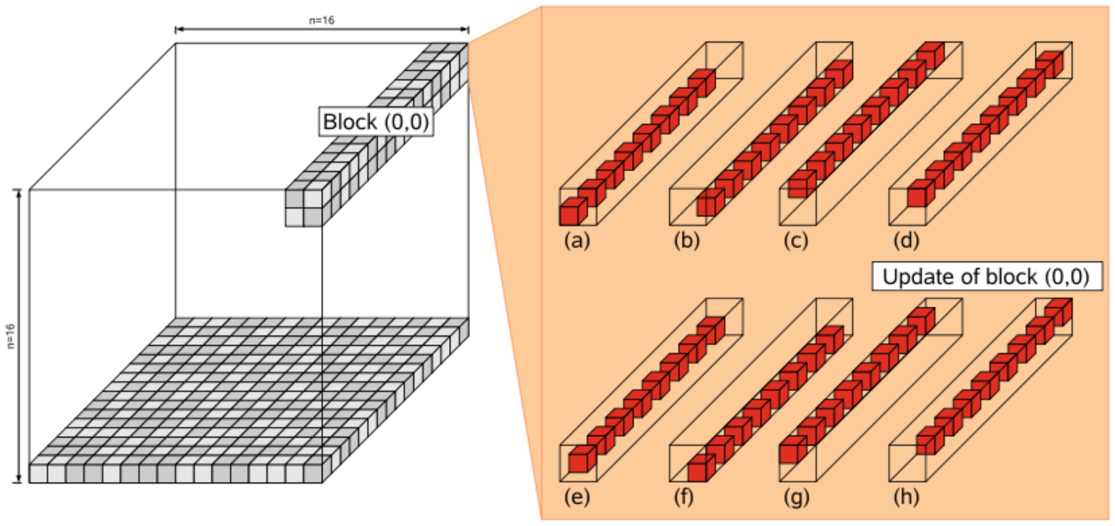

Figure 3: This figure, taken from [5], shows the assignment of checkerboard-style spin indices to GPU blocks, where the letters denote threads in the block. Only the indexing scheme (red blocks) is important.

[4] Roman Martoňák, Giuseppe Santoro, and Erio Tosatti. Quantum annealing by the path-integral monte carlo method: The two-dimensional random ising model. *Phys. Rev. B*, 66:094203, Sep 2002.

[5] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *J. Comput. Phys.*, 228(12):4468–4477, July 2009.