Accelerating the Ising Model with Numba

Advanced Scientific Programming in Python, MATH-661, EPFL

Sokratis Anagnostopoulos

Abstract

The Ising model is one of the most simulated phenomena in physics, but also poses computational challenges, mainly due to the spatio-temporal spin interactions which increase the algorithmic complexity proportionally to the problem dimensionality. This work aims to explore the capabilities of Numba in accelerating such simulations, and compare it to a vectorization approach. The benchmarks show that the standard Metropolis algorithm can be accelerated by more than 2 orders of magnitude as the computational lattice scales to larger sizes. Although Numba is the best performing model, the vectorization approach via the checkerboard scheme provides adequate performance, suggesting that vectorization is a good option for cases where Numba may introduce algorithmic difficulties due to its low-level implementation requirements.

1. Introduction

One of the most fundamental analytical models in statistical physics is the Ising model, which was originally introduced to describe ferromagnetic systems [1]. In its original form, the model expresses the interactions between neighboring spin variables $s_i \in \{-1, +1\}$, which lie on the vertices of a lattice. The energy of this system is given by the Hamiltonian which includes the spin interactions and an external magnetic field term which may act on the system. Despite its simplicity, the Ising model is able to capture essential properties of ferromagnets, phase transitions and critical behaviors like symmetry breaking and universality.

The first exact solution for the 2D Ising zero-field model on square lattices was obtained by Onsager [2], which revealed a continuous phase transition at a positive critical temperature. However, as the dimensionality increases or more effects are introduced (external fields, complex geometries) the model becomes intractable. Hence, numerical simulations based on Markov Chain Monte Carlo (MCMC) methods like the Metropolis algorithm [3] become especially useful when it comes to studying state equilibria and critical system behaviors.

A major computational challenge arises in simulating lattice systems of increasing size or dimensions, where Monte Carlo sweeps become increasingly expensive. This computational bottleneck is the main limiting factor for long-time integration and studying the dynamics at larger finite scales. The present work demonstrates how a naive Python 2D implementation of the Metropolis algorithm can be optimized using just-in-time (JIT) complication using Numba. A vectorized version of the original algorithm is also implemented as an additional point of reference. The two optimized versions yield substantial performance improvements without compromising algorithmic clarity, enabling the study of larger lattices under time and computational cost constraints.

2. The Ising Model

In its original form, the Ising model consists of a lattice of discrete binary variables (up or down spins), which interact with their nearest neighbors and may be influenced by an external magnetic field. More specifically, let $s_{i,j} \in \{-1, +1\}$ denote the spin at a lattice site (i, j) on a two-dimensional square lattice of size $N \times N$. The total energy of a spin configuration $\{s_{i,j}\}$ is given by the Hamiltonian:

$$H(\{s_{i,j}\}) = -J \sum_{\langle i,j;k,l \rangle} s_{i,j} s_{k,l} - h \sum_{i,j} s_{i,j}, \tag{1}$$

where J is the coupling constant ($J > 0$ for ferromagnetic interactions), h is the strength of an external magnetic field acting on the lattice, and $\langle i,j;k,l \rangle$ is the summation over the four adjacent neighbor pairs. The aim of the model is to simulate configurations which follow a Boltzmann distribution:

$$\mathbb{P}(\{s_{i,j}\}) \propto \exp\left(-\beta H(\{s_{i,j}\})\right), \tag{2}$$

where $\beta = \frac{1}{k_B T}$ is the inverse temperature, with k_B the Boltzmann constant (often set to 1).

3. Monte Carlo Simulation: the Metropolis Algorithm

In order to achieve sampling from the Boltzmann distribution, a Markov chain is constructed via the Metropolis-Hastings algorithm, which satisfies detailed balance. Namely, for any spin flip $s_{i,j} \mapsto -s_{i,j}$, the energy change is given by:

$$\Delta E = E_{\text{after}} - E_{\text{before}} = 2 s_{i,j} (J \sum_{(k,l) \in \mathcal{N}} s_{k,l} + h), \tag{3}$$

where $\mathcal{N}(i,j)$ denotes the set of the four adjacent neighbors of site (i,j). To avoid getting stuck in local minima where energy dissipates completely, an acceptance rule must be implemented which can randomly accept energy increasing states with the Boltzmann distribution as:

$$\text{Accept with probability } \begin{cases} 1 & \text{if } \Delta E \leq 0, \\ \exp(-\beta \Delta E) & \text{if } \Delta E > 0. \end{cases} \tag{4}$$

Accepting random higher-energy states approximates the correct distribution at equilibrium, ensures ergodicity (exploring different states) and detailed balance (transitions between states satisfy Eq. 3). Indicative snapshots of the Ising animation are shown in Figure 1.

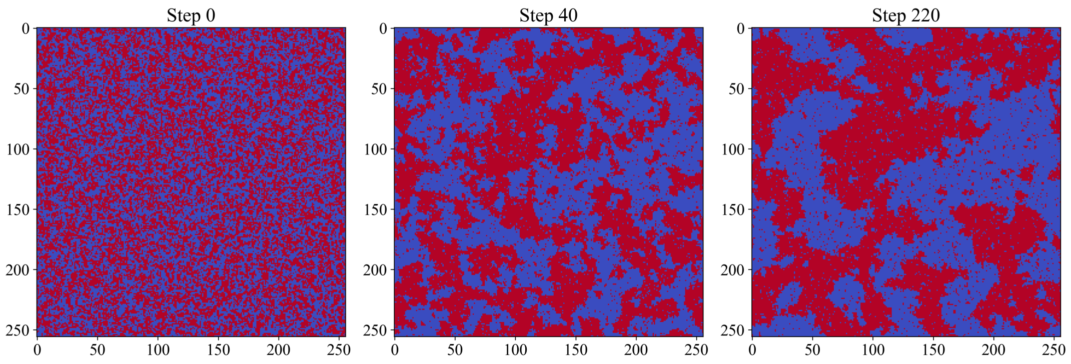

Figure 1. Indicative Ising Model states of a 256 by 256 lattice without an acting magnetic field. The random initialization develops into a segregation field of opposing spins. The settings used for this simulation were $J = 1, h = 0, T = 2.15$.

4. Implementation Details

The Metropolis algorithm is implemented in three different versions: initially we start with a naive approach using NumPy and nested "for" loops, then we use the Numba decorator to speed up the functions, and finally a vectorized version of the initial NumPy algorithm is also tested. Generally, all algorithms have $\mathcal{O}(T \cdot N^2)$ complexity, where T is the number of steps. However, as we will see in the benchmarks the performance can have great variations based on the implementation. The original NumPy algorithm applies equations 3 and 4 for each site of a randomly initialized lattice, containing N^2 randomly picked coordinates from the original square lattice. For fairness, the initialization is implemented with NumPy's *random.randint* function, as *random.shuffle* is not supported by Numba. Each Metropolis step is then performed by iterating over the sequence of all lattice sites.

The functions can be accelerated simply by using Numba's `@njit` decorator. However, note that the coding style should be generally kept at low-level, as most of the high-level functions provided by NumPy or other libraries will not be compiled properly. This means that nested "for" loops are usually preferred, even though they are not considered a "pythonic" approach. Another important note is the overhead time associated with Numba's low level virtual machine (LLVM) complication for any given specific function. For that reason, a single warm-up call is made for a small grid (16 x 16), which is then excluded from the performance timing of the first resolution.

Finally, a vectorized form of the original NumPy algorithm is also tested for its performance, along the previous two versions. One way to implement this vectorization without violating the detailed balance for independent sublattices is to apply the checkerboard approach. If we assume that the full lattice forms a checkerboard, one may notice that black and white spins are not conditionally independent of each other, while all spins of the same color are. If both colors are updated simultaneously, then a black spin is affected by white spins that are themselves changing at the same time, which introduces a cyclic dependency during the update step. Thus, an efficient speedup is to apply the Metropolis algorithm in two sweeps, first for the black and then for the white spins, which almost vectorizes the inner "for" loop completely (from N^2 iterations we now need only 2).

5. Benchmarking and Observations

The computational time scaling for lattice sizes from 16 x 16 to 512 x 512 is reported in Figure 2, for all three versions discussed in the previous section, ran on a laptop Ryzen 9 CPU. The Numba implementation is the clear winner, offering a consistent performance gain of about 2.5 orders of magnitude, compared to the original NumPy implementation. Interestingly, the vectorized version of the NumPy algorithm is a decent competitor for Numba, as it is only about 2 times slower for all grid resolutions larger than 64 x 64. For smaller lattices the vectorization overhead dominates, hurting performance. In general, these results imply that the main bottleneck for this problem is the nested lattice "for" loop, as the interpreted Python loops cause a major overhead per iteration. The JIT complication circumvents this issue by translating the functions into optimized machine code at runtime via LLVM. This essentially removes Python bytecode interpretation, any overhead associated with function calls and inefficient bound checks. Therefore, for accelerating algorithms where vectorization is not trivial, the decision lies between easy JIT complication which requires a low-level implementation, or high-level implementation but vectorizing using creative approaches, like the checkerboard scheme. However, in cases where vectorization is not possible by the problem definition, Numba should be of high priority if the goal is computational speed.

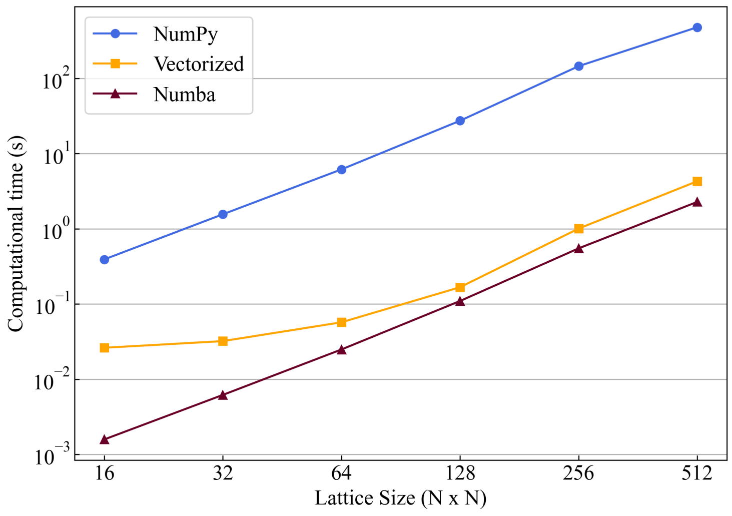

Figure 2. Scaling performance of the Metropolis algorithm. The computational time for the 512 x 512 lattice took 486 s, 3.9 s and 2 s for NumPy, Vectorized and Numba versions, respectively.

It should be highlighted that developing sub-optimal Numba code can happen very easily and is not always straightforward to debug. Initial implementations of the Numba version were actually slower than the NumPy code, which after some investigation, was narrowed down to two reasons. As previously mentioned *random.shuffle* is a NumPy routine which is not recognized by Numba. When it was used for the lattice initialization, and even worse, for the random index picking (instead of the correct *np.random.randint(0, N)*) the performance was hindered dramatically. The second problem was caused by an attempt to precompute the lattice indices and pass them as a Python list of tuples. This is a critical performance pitfall, as Numba has to fall back to object mode (slower, interpreted) since it doesn't know how to type Python objects like tuples in lists. As Numba supports a subset of NumPy and Python, any function that uses features outside this subset (like list comprehensions, or certain NumPy advanced indexing) will not be compiled efficiently.

A final note is that the function used for the animation had a very large overhead of 8s, while the vectorized code was able to run it in 0.5 seconds. This highlights an additional decision issue: Numba has to be used when function calls are expected to be made multiple times on the same machine, so that its cache-aware properties can take effect. For one-time calls (web-based apps) of functions that have a lot of "read/write" or "copy" operations (like the animation function), a vectorized version will probably be more suitable.

References

[1] E. Ising, Beitrag zur theorie des ferromagnetismus, Zeitschrift für Physik 31 (1925) 253–258.
[2] L. Onsager, Crystal statistics. i. a two-dimensional model with an order-disorder transition, Physical Review 65 (1944) 117–149.
[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, Equation of state calculations by fast computing machines, Journal of Chemical Physics 21 (6) (1953) 1087–1092.