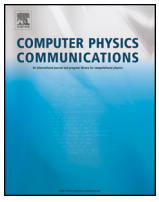
Computer Physics Communications 294 (2024) 108941

Contents lists available at ScienceDirect

Computer Physics Communications

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



Computational Physics

GPU-acceleration of tensor renormalization with PyTorch using CUDA ☆

Raghav G. Jha [a,*], Abhishek Samlodia [b]

[a] Thomas Jefferson National Accelerator Facility, Newport News, VA 23606, USA
[b] Department of Physics, Syracuse University, Syracuse NY 13244, USA

ARTICLE INFO

Article history:
Received 6 June 2023
Received in revised form 15 September 2023
Accepted 18 September 2023
Available online 21 September 2023

Keywords:
Tensor
Lattice field theory
Spin models
CUDA
Python
Speedup

ABSTRACT

We show that numerical computations based on tensor renormalization group (TRG) methods can be significantly accelerated with PyTorch on graphics processing units (GPUs) by leveraging NVIDIA's Compute Unified Device Architecture (CUDA). We find improvement in the runtime (for a given accuracy) and its scaling with bond dimension for two-dimensional systems. Our results establish that utilization of GPU resources is essential for future precision computations with TRG.

Published by Elsevier B.V.

1. Introduction

The state-of-the-art classical method to efficiently study classical/quantum spin systems in lower dimensions is undoubtedly the tensor network method. This started with the realization that the ground state of a one-dimensional system with local Hamiltonian can be written efficiently in terms of matrix product states (MPS) which is then optimized using well-known algorithms. This idea and some of its higher dimensional generalizations are now routinely used for simulating quantum systems with low entanglement [1]. There has been an alternate effort [2,3], more natural to lattice field theory based on the Lagrangian or the partition function, known as the tensor renormalization group (TRG). This enables us to perform a version of the numerical approximation of the exact renormalization group equations to compute the Euclidean partition function by blocking the tensor network. If this blocking (coarse-graining) is applied recursively, one generates a description of the theory at increasing length scales accompanied by a corresponding flow in the effective couplings. In addition to the application of TRG to discrete spin models, where it was first introduced, it has also been used to study spin models with continuous symmetry and gauge theories in two and higher dimensions [4–7]. We refer the interested reader to the review article [8] to start a reference trail.

The prospect of carrying out high-precision TRG calculations as an alternative to the standard Monte Carlo based lattice gauge computations has several motivations. The most important is the ability to study complex-action systems in the presence of finite chemical potential or topological θ-term. Since the TRG algorithm does not make use of sampling techniques, they do not suffer from the sign problem [5,9,10]. However, the trade-off seems to be the fact that truncation of TRG computations (which cannot be avoided) does not always yield the correct behavior of the underlying continuum field theory.

A major fraction of the computation time is the contraction of the tensors during successive iterations. An efficient way of doing this can lead to substantial improvements which becomes crucial when studying higher-dimensional systems. The explorations in four dimensions using ATRG [11] and HOTRG [3] have made use of parallel CPU computing to speed up the computations and have obtained good results [12].

The unreasonable effectiveness of tensors is not just restricted to de ibing the physical systems. In machine learning applications, tensors are widely used to store the higher-dimensional classical data and train the models. Due to such widespread implications of this field, several end-to-end software packages have been developed and one has now access to various scalable packages such as TensorFlow and PyTorch which can be also be used for Physics computations. PyTorch [13] is a Python package that provides

☆ The review of this paper was arranged by Prof. Z. Was.

* Corresponding author.
E-mail addresses: raghav.govind.jha@gmail.com (R.G. Jha), asamlodia@gmail.com (A. Samlodia).

https://doi.org/10.1016/j.cpc.2023.108941
0010-4655/Published by Elsevier B.V.

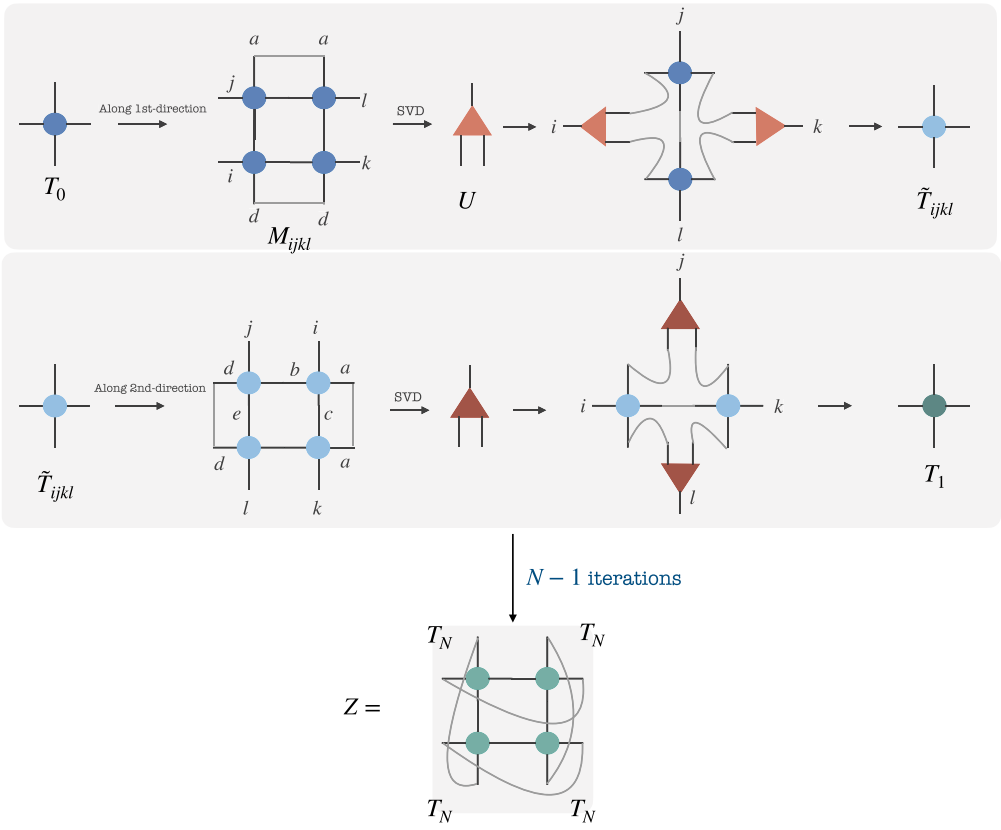


Fig. 1. Schematic representation of the higher-order TRG implemented in this work. The diagram should be viewed from top to bottom, left to right with the first two panels denoting the coarse-graining along two directions. The first step combines four initial tensors (denoted T_0) as shown to construct M_{ijkl}. Then we combine the left and right indices to construct a matrix and take the SVD of that matrix to obtain the projector U. We then combine the tensors along a particular direction to obtain a coarse-grained tensor $\tilde{T}$. Then with this tensor, we perform similar steps along the orthogonal direction to obtain T_1. This constitutes one step of coarse-graining. Doing this $N-1$ times more and then contracting the indices gives us an approximation to Z with periodic boundary conditions.

some high-level features such as tensor contractions with strong GPU acceleration and deep neural networks built on a reverse-mode automatic differentiation system which is an important step used in backpropagation, a crucial ingredient of machine learning algorithms. Though there have been some explorations of MPS tensor network implementations using CUDA (a parallel computing platform that allows programmers to use NVIDIA GPUs for general-purpose computing) [14], it is not widely appreciated or explored in the real-space TRG community to our knowledge. CUDA provides libraries such as cuBLAS and cuDNN that can leverage tensor cores and specialized hardware units that perform fast contractions with tensors.

In this paper, we demonstrate that a simple modification of the code using PyTorch with CUDA and `opt_einsum` [15] improves the runtime by a factor of ~ 12x with $D = 89$ for the generalized XY model (described in Sec. 3). We also present results for the Ising model and the 3-state Potts model as references for the interested reader and how one can obtain state-of-the-art results in less computer time. We refer to the use of PyTorch for TRG computations with CUDA as TorchTrg and the code used to produce the results in this paper can be obtained from Ref. [16].

2. Algorithm and TorchTrg discussion

We use the higher-order TRG algorithm based on higher-order singular value decomposition (HOSVD) of tensors. This algorithm has been thoroughly investigated in the last decade and we refer the reader to the recent review article [8] for details. The goal of this algorithm is to effectively carry out the coarse-graining of the tensor network with controlled truncation by specifying a local bond dimension D which is kept constant during the entire algorithm. We show one full iteration of coarse-graining using higher-order TRG algorithm in Fig. 1. The first step is to combine four initial tensors (denoted T_0) as shown to construct M_{ijkl}. Then we combine the left and right indices to construct a matrix and take the SVD of that matrix to obtain the projector U. We then combine the tensors along a particular direction to obtain a coarse-grained tensor $\tilde{T}$. Then we take this tensor and perform similar steps along the orthogonal direction to obtain T_1. This constitutes one step of coarse-graining. Doing this $N-1$ more times and then contracting the indices with periodic boundary conditions gives us the approximation to Z. We show the algorithm in Fig. 1 for the reader. The computational complexity for the higher-order TRG algorithm scales as $O(D^{4d-1})$ for d-dimensional Euclidean systems. The most expensive part of HOTRG computations (especially for higher dimensions) is the contraction of tensors with some fixed truncation D. This is needed to keep the growing size fixed to a reasonable value depending on the resources. The ratio of time complexity between tensor contraction and SVD is $O(D^{4d-7})$. In an earlier work by one of the authors [17], to perform the tensor contractions, the `ncon` Python library was used. There is an equivalent way of doing these contractions which has been extensively used in machine learning and is known as `opt_einsum` [18] which was used for standard CPU computations in [19]. In this work, we make use of additional capabilities of `opt_einsum` by performing these contractions on a GPU architecture without explicitly copying any tensor to GPU device. For this purpose, a more performant backend is required which requires converting back and forth between array types. The `opt_einsum` software can handle this automatically for a wide range of options

such as TensorFlow, Theano, JAX, and PyTorch. In this work, we use PyTorch on NVIDIA GeForce RTX 2080 Ti. The use of packages developed primarily for machine learning like PyTorch and TensorFlow to problems in many-body Physics is not new. TensorFlow was used to study spin chains using tree tensor networks [20] based on the software package developed in Ref. [21]. However, we are not aware of any real-space tensor renormalization group algorithms which have made use of GPU acceleration with these ML/AI-based Python packages and carried out systematic study showing the improvements. Another advantage of using PyTorch is the ability to carry out the automatic differentiation using: `torch.tensor(T, requires_grad = True)` useful in computing the derivatives similar to that in Ref. [22]. The availability of additional GPUs also accelerates the program substantially as we will discuss in the next section. The main steps involving the conversion to the desired backend (if CUDA is available) and performing the coarse-graining step are summarized below:

1. Start with initializing all the tensors in the program as `torch` CPU tensors.
2. For tensor contractions, we use the library - `opt_einsum_torch` which utilizes GPU cores for contractions and returns a `torch` CPU tensor [15].
3. We use the linear algebra library available within `torch` i.e., `torch.linalg` for performing SVD and other basic operations.

Since the tensor contractions are carried out on GPU, some fraction of the memory load on the CPU is reduced, and hence the program becomes more efficient. Furthermore, we have observed that as the architecture of the GPU improves, the computational cost improves further. We used `opt_einsum` since it can significantly cut down the overall execution time of tensor network contractions by optimizing the order to the best possible time complexity and dispatching many operations to canonical BLAS or cuBLAS which provides GPU-accelerated implementation of the basic linear algebra subroutines (BLAS) [18,23]. The order of contracting tensors is an important consideration to make in any quantum many-body computations with tensors. We revisit this issue in Appendix A and show how they significantly differ in computation times. We show some code snippets with explanations below for the interested reader. The program requires three major libraries: `numpy,scipy,torch` which we import at the start. We also check whether we can make use of GPU i.e., whether CUDA is available. If it is available, `use_cuda == True` is set for the entire computation.

```
import numpy as np # NumPy version 1.21.6
import scipy as sp # SciPy version 1.7.1
import torch # Torch version 1.10.1+cu102

# Import PyTorch. pip install torch usually works.
use_cuda = torch.cuda.is_available()
# Check whether CUDA is available. If not, we do the standard CPU computation
```

If CUDA is available, we print the number of devices, names, and memory and import the planner for Einstein's summation (tensor contractions). Note that the planner from Ref. [15] implements a memory-efficient `einsum` function using PyTorch as backend and uses the `opt_einsum` package to optimize the contraction path to achieve the minimal FLOPS. If `use_cuda == False`, then we just import the basic version of the `opt_einsum` package as `contract`. The notation for `contract` and CUDA based `ee.einsum` is similar. To compute $A_{ijkl}B_{pjql} \rightarrow C_{ipkq}$, we do:
`C = ee.einsum('ijkl,pjql->ipkq', A, B)` with
`use_cuda == True` or
`C = contract('ijkl,pjql->ipkq', A, B)` otherwise.

```
if use_cuda:
    print('__CUDNN VERSION:', torch.backends.cudnn.version())
    print('__Number CUDA Devices:', torch.cuda.device_count())
    print('__CUDA Device Name:', torch.cuda.get_device_name(0))
    print('__CUDA Device TotalMemory[GB]:', torch.cuda.get_device_properties(0).total_memory/1e9)

    # __CUDNN VERSION: 7605
    # __Number CUDA Devices: 1
    # __CUDA Device Name: NVIDIA GeForce RTX 2080 Ti
    # __CUDA Device Total Memory [GB]: 11.554717696

    from opt_einsum_torch import EinsumPlanner
    # To install use: pip install opt-einsum-torch
    ee = EinsumPlanner(torch.device('cuda:0'), cuda_mem_limit = 0.8)

else:
    from opt_einsum import contract
    # To install use: pip install opt-einsum
```

One thing to note is that we have to specify the CUDA memory limit for the planner. This parameter can be tuned (if needed) but we have found that a value between 0.7 and 0.85 usually works well. Note that this can sometimes limit the maximum D one can employ in TRG computations. So, it should be selected appropriately if CUDA runs out of memory. The choice of this parameter and the available memory can result in errors. A representative example is:

```
RuntimeError: CUDA out of memory.
Tried to allocate 2.25 GiB (GPU 0; 10.76 GiB
total capacity; 5.17 GiB already allocated;
2.20 GiB free; 7.40 GiB reserved in
total by PyTorch) If reserved memory is >>
allocated memory try setting
max_split_size_mb to avoid fragmentation.
```

We show code snippet to address this error below.

```
# Sometimes to tackle the error above, doing the below works.
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:<size here>"
```

```
# Tuning the cuda_mem_limit also helps.
ee =
    EinsumPlanner(torch.device('cuda:0'),
    cuda_mem_limit = 0.7)
```

The user can also completely skip the memory allocation and run without specifying. It should not affect the performance significantly. In implementing TorchTrg, we explored four models that can be selected at run time by the user. The choices are:

```
models_allowed = ['Ising',
    'Potts','XY', 'GXY']
```

There are four command-line arguments: temperature $1/\beta$, bond dimension D, number of iterations, and the model. An example of execution is: `python 2dTRG.py 2.27 64 20 Ising`. Based on the model and the parameters, it constructs the initial tensor for the coarse-graining iterations to commence. It is straightforward to add other models or observables and make use of the basic CUDA setup presented here. In TorchTrg, we have simplified the code for a non-expert to the extent that a single coarse-graining step which takes in a tensor and outputs transformed tensor and normalization factor is `23` lines long and can accommodate different architectures. We wrap all commands which can potentially make use of CUDA acceleration i.e., contractions etc. inside `use_cuda` conditional statement. We show this part of the code below:

```
def SVD(t, left_indices, right_indices,
    D):
    T = torch.permute(t,
    tuple(left_indices + right_indices))
    if use_cuda else np.transpose(t,
    left_indices + right_indices)
    left_index_sizes = [T.shape[i] for
    i in range(len(left_indices))]
    right_index_sizes = [T.shape[i] for
    i in range(len(left_indices),
    len(left_indices) +
    len(right_indices))]
    xsize, ysize =
    np.prod(left_index_sizes),
    np.prod(right_index_sizes)
    T = torch.reshape(T, (xsize,
    ysize)) if use_cuda else
    np.reshape(T, (xsize, ysize))
    U, _, _ = torch.linalg.svd(T,
    full_matrices=False) if use_cuda
    else sp.linalg.svd(T,
    full_matrices=False)
    size = np.shape(U)[1]
    D = min(size, D)
    U = U[:, :D]
    U = torch.reshape(U,
    tuple(left_index_sizes + [D])) if
    use_cuda else np.reshape(U,
    left_index_sizes + [D])
    return U

def coarse_graining(t):
    Tfour =
    ee.einsum('jabe,iecd,labf,kfcd->ijkl',
    t, t, t, t) if use_cuda else
    contract('jabe,iecd,labf,kfcd->ijkl',
    t, t, t, t)
    U = SVD(Tfour,[0,1],[2,3],D_cut)
    Tx =
    ee.einsum('abi,bjdc,acel,edk->ijkl',
    U, t, t, U) if use_cuda else
    contract('abi,bjdc,acel,edk->ijkl',
    U, t, t, U)
    Tfour =
    ee.einsum('aibc,bjde,akfc,flde->ijkl',
    Tx,Tx,Tx,Tx) if use_cuda else
    contract('aibc,bjde,akfc,flde->ijkl',
    Tx,Tx,Tx,Tx)
    U = SVD(Tfour,[0,1],[2,3],D_cut)
    Txy =
    ee.einsum('abj,iacd,cbke,del->ijkl',
    U, Tx, Tx, U) if use_cuda else
    contract('abj,iacd,cbke,del->ijkl',
    U, Tx, Tx, U)
    norm = torch.max(Txy) if use_cuda
    else np.max(Txy)
    Txy /= norm
    return Txy, norm
```

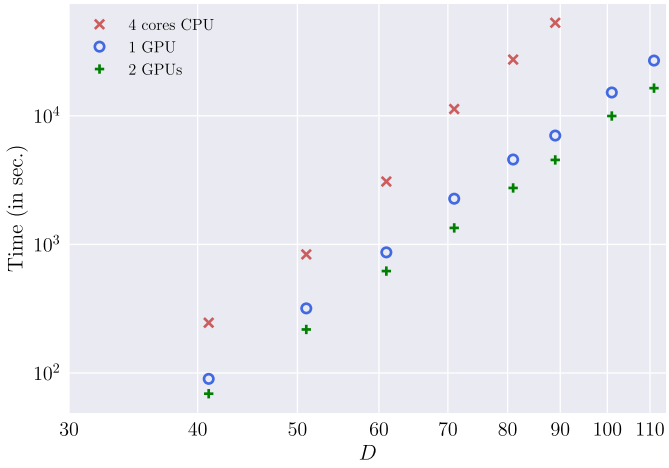


Fig. 2. The runtime in seconds for the GXY model with different D on lattice of size $2^{30} \times 2^{30}$ with CPU version and TorchTrg. We used maximum $D = 111$ with TorchTrg and $D = 89$ with the standard CPU version to compare the timings. Note that these timings are just for the computation of Z without any impure tensor computations. If we insert impure tensors (say to compute magnetization) then for $D = 101$, the run time on a single GPU increases to $\sim 19,000$ seconds compared to $15,200$ seconds for a pure network.

3. Models and results

In this section, we show the results obtained using TorchTrg. We first show the run time comparison on CPU and CUDA architectures for the generalized XY model, which is a deformation of the standard XY model. Then, we discuss the TRG method as applied to the classical Ising model and discuss how we converge to a desired accuracy faster. In the last part of this section, we discuss the q-state Potts model with $q = 3$ and accurately determine the transition temperature corresponding to the continuous phase transition.

3.1. GXY model

The generalized XY (GXY) model is a spin nematic deformation of the standard XY model [24]. The Hamiltonian (with finite external field) is given by:

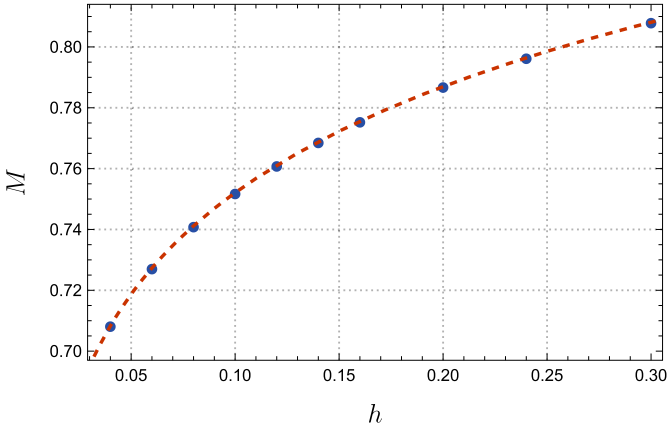


Fig. 3. The magnetization of the XY model at various finite magnetic fields h at the critical temperature on lattice size of $2^{36} \times 2^{36}$. The fit is $0.877(3)h^{1/15.11(17)}$.

$$\mathcal{H} = -\Delta \sum_{\langle ij \rangle} \cos(\theta_i - \theta_j) - (1-\Delta) \times \sum_{\langle ij \rangle} \cos(2(\theta_i - \theta_j)) - h \sum_i \cos\theta_i, \quad (3.1)$$

where we follow the standard notation $\langle ij \rangle$ to denote nearest neighbors and $\theta_i \in [0, 2\pi)$. Two limits are clear cut: $\Delta = 0$ corresponds to a pure spin-nematic phase and $\Delta = 1$ is the standard XY model. We will report on the ongoing tensor formulation of this model in a separate work [25].

To test our GPU-acceleration, we have focused on two models. One with discrete symmetry (Ising model) and the other with continuous global symmetry (GXY). Since the indices for the initial tensor in GXY model takes infinite values, a suitable truncation is needed from the first step of coarse-graining while for the Ising model, the first few iterations can be carried out exactly due to the size of the initial tensor.

For the GXY model in this work, we will only consider $h = 0$. We performed tensor computations for a fixed value of $\Delta = 0.5$ and for different D. The computation time for this model scaled like $\sim D^{5.4(3)}$ with TorchTrg while the CPU timings were close to $\sim D^7$ which is consistent with the expectation of higher-order TRG scaling in two dimensions. Note that since the cost of SVD scales like $\sim D^6$, the observed computation time complexity also signals the fact that even SVD computations are accelerated by GPU, though they are sub-dominant compared to the tensor contractions.

We show the comparison between the run times showing the CUDA acceleration with TorchTrg in Fig. 2. We used one and two CUDA devices available with NVIDIA GeForce RTX 2080 Ti. We found that the latter is a factor of about `1.5x` faster. In addition, we also tested our program on 4 CUDA devices with NVIDIA TITAN RTX[1] and found a further speedup of $\sim$ `1.3x` (not shown in the figure) over two CUDA RTX 2080 Ti for range of D. Therefore, it is clear that with better GPU architectures in the future, TRG computations will benefit significantly from moving over completely to GPU-based computations.

Using our GPU-enhanced code, we can access larger D and there is possibility of computing the critical exponents accurately. We fixed $\Delta = 1$ in Eq. (3.1) and computed the critical exponent δ defined as $M \sim h^{1/\delta}$ at $T = T_c = 0.89290$ [17] and obtained $\delta = 15.11(17)$ compared to the exact result of 15. The results are shown in Fig. 3 with $D = 111$. To ensure convergence in the bond dimension, we also took $D = 131$ and found that difference in magnetization between the two D is $\sim 10^{-8}$.

3.2. Ising model

In the previous subsection, we compared the run time on the GXY model, however, we also want to test the algorithm with `opt_einsum` and GPU acceleration on a model with a known solution. In this regard, we considered the Ising model on a square lattice which admits an exact solution. This makes it a good candidate for the sanity check of the algorithm and the convergence properties. We will check the accuracy of the higher-order TRG method by computing the free energy which is the fundamental quantity accessible in TRG computations. It can be obtained directly in the thermodynamic limit from the canonical partition function Z as $-T \ln Z$. The exact result for the free energy of the Ising model is given by:

$$f_{\mathrm{E}} = -\frac{1}{\beta}\left(\ln(2\cosh(2\beta)) - \kappa^2\, {}_4F_3\left[\begin{matrix}1 & 1 & \frac{3}{2} & \frac{3}{2} \\ & 2 & 2 & 2\end{matrix}; 16\kappa^2\right]\right), \quad (3.2)$$

where $\kappa = \sinh(2\beta)/(2\cosh^2 2\beta)$ and ${}_pF_q$ is the generalized hypergeometric function and β is the inverse temperature. We define the error in TRG computation of the free energy as:

$$\left|\frac{\delta f}{f}\right| = \left|\frac{f_{\mathrm{TRG}} - f_{\mathrm{E}}}{f_{\mathrm{E}}}\right|. \quad (3.3)$$

We show the results for this observable for various T in the left panel of Fig. 4 and at fixed $T = T_c$ for various D (run time) in the right panel of Fig. 4. Each data point in the left panel of Fig. 4 took about 2000 seconds on 4 cores of Intel(R) Xeon(R) Gold 6148. The largest deviation we observed (as expected) was at $T = T_c \sim 2.269$ where $|(\delta f)/f|$ was 1.91×10^{-9}. We could not find any other algorithm with such accuracy for the same execution time. Note that we did not even use the CUDA acceleration for this comparison. We used a bond dimension of $D = 64$ and computed the free energy on a square lattice of size $2^{20} \times 2^{20}$. In order to ensure that the result has converged properly, we also studied lattice size $2^{25} \times 2^{25}$ and obtained the same deviation from the exact result.

Another useful quantity to compute is the coefficient α defined as $|(\delta f)/f| \propto 1/D^{\alpha}$. Different TRG algorithms have different α and a higher value represents faster convergence with the bond dimension D. We show the error as a function of D at $T = T_c$ for Ising model in the right panel of Fig. 4. Doing a fit of the numerical fit of the data for $D \in [48, 114]$ gave $\alpha = 4.12(2)$. This exponent has close relation to the scaling of entanglement entropy of the critical $c = 1/2$ Ising model which in turn is related to the finite D scaling of correlation length given by $\xi_D \sim D^{\kappa}$ [26]. It is argued that the error in free energy scales like $\sim \xi^{-d}$ for $d = 2$ dimensional Ising model with maximum error at $T = T_c$ where the correlation length diverges. This implies that $|(\delta f)/f| \propto D^{-2\kappa}$. Using this we get $\kappa \sim 2.06$ which is very close to the expected[2] exponent $\kappa \sim 2.03425$. For this model, we also compared our numerical results with two other recent works. The triad second renormalization group introduced in [27] can only get to an accuracy of 10^{-9} at $T = T_c$ with about 10^5 seconds of CPU time which roughly translates to our CPU code being about 30 times faster to get the same accuracy for the Ising model. The ∂TRG method of Ref. [22] does not have an accuracy of 10^{-9} at the critical temperature even with $D = 64, 128$ though admittedly it behaves much better away from critical temperatures. We compared the CPU and GPU code for the Ising model and the results are summarized in Table 1. We

[1] We thank Nobuo Sato for access to the computing facility.

[2] We thank the referee for this suggestion.

Table 1
Timings (in seconds) for the HOTRG algorithm for Ising model for $2^{20} \times 2^{20}$ lattice at $T = T_c$ using CPU and GPU.

D	$\left\|\frac{\delta f}{f}\right\|$	A100	RTX 2080	4 CPUS
84	6.6×10^{-10}	6004	9171	11714
94	4.4×10^{-10}	11960	19305	29376
104	2.9×10^{-10}	21376	36159	58715
109	2.4×10^{-10}	28942	46350	80578

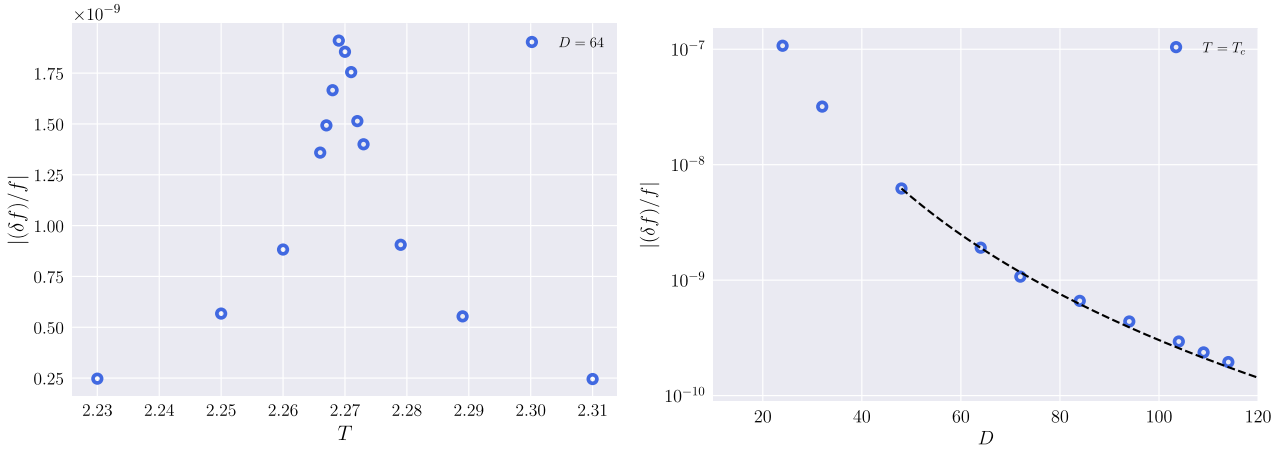


Fig. 4. Left: The deviation of the TRG results from the exact result (3.2). **Right**: The dependence of the error on D and therefore on the execution time at $T = T_c$.

found that using NVIDIA A100[3] had much better performance than RTX 2080.

3.3. Three-state Potts model

As a generalization of the Ising model, we can also consider the classical spins to take values from $1, 2, \cdots, q$. This is the q-state Potts model which is another widely studied statistical system. In particular, we consider the case $q = 3$ as an example. On a square lattice, this model has a critical temperature that is exactly known for all q. The transition, however, changes order at some q and the nature of the transition is continuous for $q < 4$ [28]. The exact analytical result for T_c on square lattice is:

$$T_c = \frac{1}{\ln(1+\sqrt{q})}. \tag{3.4}$$

If we restrict to $q = 2$, we reproduce the Ising result (up to a factor of 2). The q-state Potts model has been previously considered using TRG methods both in two and three dimensions in Refs. [19,29,30]. The initial tensor can be written down by considering the $q \times q$ Boltzmann nearest-neighbor weight matrix as:

$$\mathbb{W}_{ij} = \begin{cases} e^{\beta}; & \text{if} i = j \\ 1; & \text{otherwise}, \end{cases} \tag{3.5}$$

and then splitting the $\mathbb{W}$ tensor using Cholesky factorization, i.e., $\mathbb{W} = LL^T$ and combining four L's to make T_{ijkl} as $T_{ijkl} = L_{ia}L_{ib}L_{ic}L_{id}$. Note that this tensor can be suitably modified to admit finite magnetic fields. We first study the convergence of density of $\ln(Z)$ with inverse bond dimension. The result is shown in Fig. 5. In the absence of external magnetic field, this model has a phase transition at $T_c \approx 0.99497$ and we check this using TorchTrg. The results obtained are shown in Fig. 6.

[3] This is currently the state-of-the-art GPU used extensively to train large language models (LLM) like GPT-4.

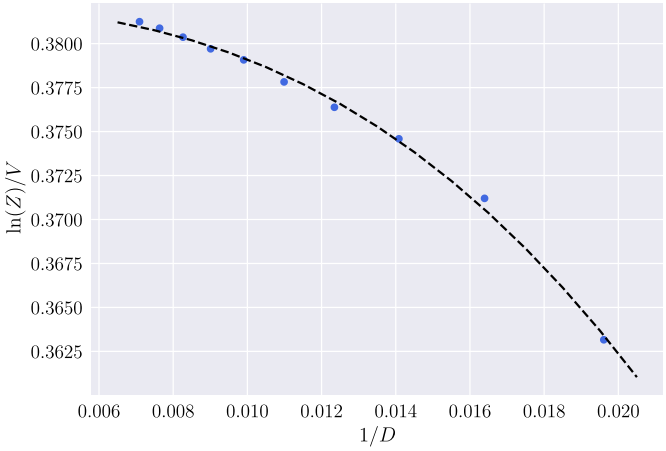


Fig. 5. The convergence of the density of $\ln(Z)$ with inverse of bond dimensions at T_c for the three-state Potts model on lattice of size $2^{30} \times 2^{30}$. The quadratic fit gives $\ln(Z)/V = 0.3822(2)$ in the $D \to \infty$ limit.

As mentioned before, a prime motivation of our work was to access large bond dimensions to accurately determine the critical exponents. We computed these exponents for the XY model in Sec. 3.1 and now we determine critical exponent α for the three-state Potts model. Taking the zero-field limit, we look at the behavior of the specific heat as $T \to T_c$. Defining $\tau = (T - T_c)/T_c$, we fit the specific heat (as $\tau \to 0$) using the standard form:

$$\ln(C_v) = \beta - \alpha \ln|\tau| \implies C_v \propto |\tau|^{-\alpha}. \tag{3.6}$$

To compute the specific heat, we compute the second derivative of the partition function using the standard finite-difference method. We fit the numerical data with (3.6) to obtain the critical exponent α taking 10% error in specific heat. Using TorchTrg, we get $\alpha = 0.38(5)$ and $0.34(7)$ respectively as shown in Fig. 7 by fitting the data for $T < T_c$ and $T > T_c$ respectively. Our result is consistent with the known result of $\alpha = 1/3$ [31].

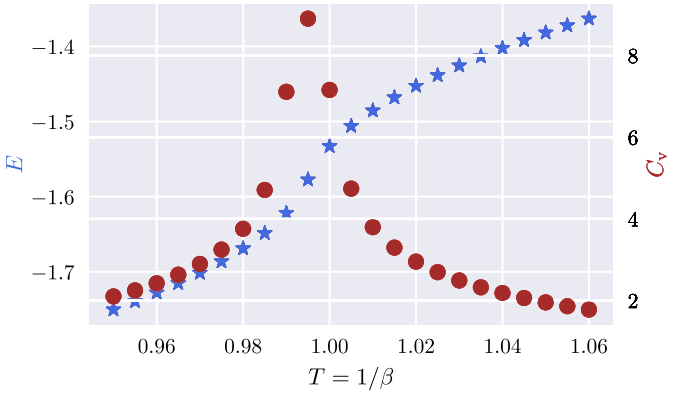


Fig. 6. The internal energy (E) and specific heat (C_v) for the $q = 3$ Potts model with $D = 64$ on a lattice of size $2^{20} \times 2^{20}$. The continuous transition from the peak of specific heat is consistent with the exact analytical result. Each data point in the plot took about 1300 seconds using TorchTrg on 2 CUDA devices.

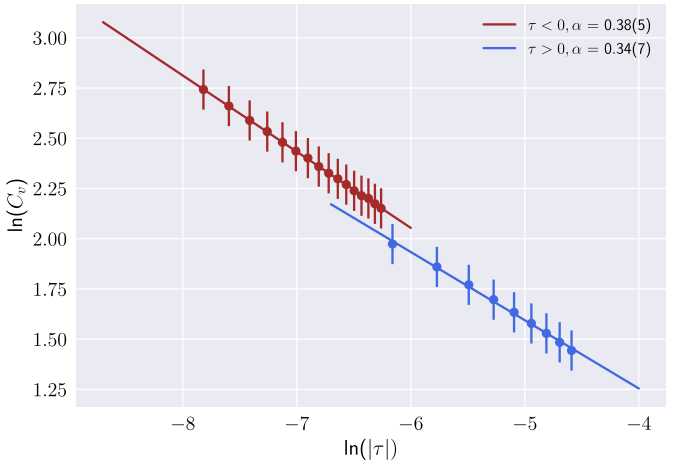


Fig. 7. The determination of the critical exponent α for lattice of size $2^{30} \times 2^{30}$ with $D = 111$.

4. Summary

We have described an efficient way of performing tensor renormalization group calculations with PyTorch using CUDA. For the two-dimensional classical statistical systems we explored in this work, there was a substantial improvement in the scaling of computation time with the bond dimension. In particular, the results show that there is ~ 8x speedup for $D = 89$ for the generalized XY model on $2^{30} \times 2^{30}$ lattice using a single GPU which increases to ~ 12x using two GPUs. For a larger bond dimension of $D = 105$, there is an estimated ~ 15x speedup. The scaling of computation time scales like $\sim O(D^5)$ with GPU acceleration which is to be compared with the naive CPU scaling of $\sim O(D^7)$ in two dimensions. This speedup means that one can explore larger D using CUDA architecture which is often required for accurate determination of the critical exponents as we have demonstrated for the XY and Potts model. We envisage that the CUDA acceleration would also help TRG computations in higher dimensions in addition to the two (Euclidean) dimensions considered in this work. There have not been many explorations in this direction but we believe that in the future, we would see extensive use of the GPU resources. A potential bottleneck in the use of GPUs for TRG computations is the memory availability. This can often cause errors and severely limit the scope of the numerical computations. Partly due to this, we have not been able to significantly speed up any three-dimensional models yet though it appears to be possible. There are several other directions that can be pursued such as implementing a C/C++ version with `opt_einsum` to have better control of the available memory while utilizing the CUDA acceleration. We leave such questions for future work.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Raghav G. Jha reports financial support was provided by U.S. Department of Energy. Raghav G. Jha reports a relationship with Thomas Jefferson National Accelerator Facility that includes: employment.

Data availability

Data will be made available on request.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177. The research was also supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Co-design Center for Quantum Advantage under contract number DE-SC0012704. We thank the Institute for Nuclear Theory at the University of Washington for hospitality during the completion of this work. The numerical computations were done on Symmetry which is Perimeter Institute's HPC system and Syracuse University HTC Campus Grid supported by NSF award ACI-1341006.

Appendix A. Contraction of network - different methods

In this Appendix, we elaborate on the optimized sequence of contraction order when dealing with complicated tensor networks. In Fig. 8, we show two different contraction pattern that yields different time complexity. Let us start with two rank-three (each with D^3 elements) and one rank-two tensor (D^2 elements). Suppose we want to contract three pairs of indices and obtain a final tensor of rank-two as shown. If we follow the blue-marked regions in the order 1 and 2 as mentioned, the cost will be $O(D^4)$. However, if we rather choose to contract the bond starting with the pink blob, then this step would be $O(D^5)$ followed by $O(D^4)$ steps leading to overall time complexity of $O(D^5)$. Hence, choosing an optimum sequence is very important for practical purposes. Fortunately, this is something `opt_einsum` and `ncon` do fairly well. The efficient evaluation of tensor expressions that involve sum over multiple indices is a crucial aspect of research in several fields, such as quantum many-body physics, loop quantum gravity, quantum chemistry, and tensor network-based quantum computing methods [32].

The computational complexity can be significantly impacted by the sequence in which the intermediate index sums are performed as shown above. Notably, finding the optimal contraction sequence for a single tensor network is widely accepted as **NP**-hard problem. In view of this, `opt_einsum` relies on different heuristics to achieve near-optimal results and serves as a good approximation to the best order. This is even more important when we study tensor networks on non-regular graphs or on higher dimensional graphs. We show a small numerical demonstration below. We initialize a random matrix and set a contraction pattern option and monitor the timings. We find that all three: `tensordot`, `ncon`, `opt_einsum` perform rather similarly. The slowest is

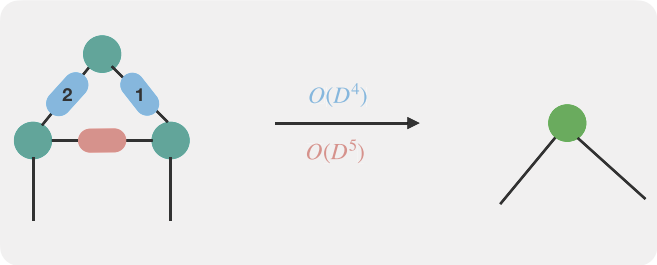


Fig. 8. Schematic representation of two ways of contracting a network. The cost is $O(D^4)$ if we follow the order of blue-shaded regions as numbered. However, if we start by contracting the pink link first, then the leading cost will be $O(D^5)$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

`np.einsum` when the optimization flag not set (i.e., false). However, since we are interested in GPU acceleration in this work, we use `opt_einsum` which has better support to our knowledge and is also backend independent. We also compare its performance for a specific contraction on CPU and with `torch` on CUDA.

```
import numpy as np
from opt_einsum import contract
from ncon import ncon

i, j, k, l = 80, 75, 120, 120
A = np.random.rand(i, j, k, l)
B = np.random.rand(j, i, k, l)

%timeit np.tensordot(A, B, axes=([1,0],[0,1]))
# 2.72 s \pm 38.3 ms per loop

%timeit np.einsum('ijkl,jiab->klab', A, B)
# WARNING: Never use this without optimization.
# Slower by factor of 500x or so! Not considered.
# We can turn the optimize flag as below.

%timeit np.einsum('ijkl,jiab->klab', A, B, optimize=True)
# 2.75 s \pm 40.2 ms per loop

%timeit contract('ijkl,jiab->klab', A, B)
# 2.69 s \pm 40.9 ms per loop

%timeit ncon((A, B),([1,2,-1,-2],[2,1,-3,-4]))
# 2.37 s \pm 20 ms per loop

i, j, k, l = 200, 100, 80, 80
A = np.random.rand(i, j, k, l)
B = np.random.rand(j, i, k, l)

import torch
from opt_einsum_torch import EinsumPlanner
ee = EinsumPlanner(torch.device('cuda:0'), cuda_mem_limit = 0.7)

%timeit contract('ijkl,jiab->klab', A, B)
# 6.57 s \pm 80.7 ms per loop [on CPU]

%timeit ee.einsum('ijkl,jiab->klab', A, B)
# 3.76 s \pm 16.9 ms per loop [on CUDA]
# For this single contraction, we see a factor of about 1.7!
```

References

[1] J.I. Cirac, D. Pérez-García, N. Schuch, F. Verstraete, Rev. Mod. Phys. 93 (Dec 2021) 045003, https://link.aps.org/doi/10.1103/RevModPhys.93.045003.
[2] M. Levin, C.P. Nave, Phys. Rev. Lett. 99 (Sep 2007) 120601, https://link.aps.org/doi/10.1103/PhysRevLett.99.120601.
[3] Z.Y. Xie, J. Chen, M.P. Qin, J.W. Zhu, L.P. Yang, T. Xiang, Phys. Rev. B 86 (Jul 2012) 045139, https://link.aps.org/doi/10.1103/PhysRevB.86.045139.
[4] A. Bazavov, S. Catterall, R.G. Jha, J. Unmuth-Yockey, Phys. Rev. D 99 (11) (2019) 114507, arXiv:1901.11443 [hep-lat].
[5] J. Bloch, R.G. Jha, R. Lohmayer, M. Meister, Phys. Rev. D 104 (9) (2021) 094517, arXiv:2105.08066 [hep-lat].
[6] T. Kuwahara, A. Tsuchiya, PTEP 2022 (9) (2022) 093B02, arXiv:2205.08883 [hep-lat].
[7] S. Akiyama, Y. Kuramashi, J. High Energy Phys. 05 (2022) 102, arXiv:2202.10051 [hep-lat].
[8] Y. Meurice, R. Sakai, J. Unmuth-Yockey, Rev. Mod. Phys. 94 (2) (2022) 025005, arXiv:2010.06539 [hep-lat].
[9] H. Kawauchi, S. Takeda, Phys. Rev. D 93 (11) (2016) 114503, arXiv:1603.09455 [hep-lat].
[10] D. Kadoh, Y. Kuramashi, Y. Nakamura, R. Sakai, S. Takeda, Y. Yoshimura, J. High Energy Phys. 02 (2020) 161, arXiv:1912.13092 [hep-lat].
[11] D. Adachi, T. Okubo, S. Todo, Phys. Rev. B 102 (Aug 2020) 054432, https://link.aps.org/doi/10.1103/PhysRevB.102.054432.
[12] S. Akiyama, Y. Kuramashi, T. Yamashita, Y. Yoshimura, Phys. Rev. D 100 (5) (2019) 054510, arXiv:1906.06060 [hep-lat].
[13] A. Paszke, et al., in: Advances in Neural Information Processing Systems, vol. 32, Curran Associates, Inc., 2019, https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.
[14] D.I. Lyakh, T. Nguyen, D. Claudino, E. Dumitrescu, A.J. McCaskey, Front. Appl. Math. Stat. 8 (2022), https://www.frontiersin.org/articles/10.3389/fams.2022.838601.
[15] H. Huo, https://pypi.org/project/opt-einsum-torch/.
[16] R. Jha, A. Samlodia, Zenodo (Sep 2023), https://zenodo.org/record/8190788.
[17] R.G. Jha, J. Stat. Mech. 2008 (2020) 083203, arXiv:2004.06314 [hep-lat].
[18] D. Smith, J. Gray, J. Open Sour. Softw. 3 (26) (2018) 753.
[19] R.G. Jha, Tensor renormalization of three-dimensional Potts model, arXiv:2201.01789 [hep-lat].
[20] A. Milsted, M. Ganahl, S. Leichenauer, J. Hidary, G. Vidal, TensorNetwork on TensorFlow: a spin chain application using tree tensor networks, arXiv:1905.01331 [cond-mat.str-el].
[21] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, S. Leichenauer, TensorNetwork: a library for physics and machine learning, arXiv:1905.01330 [physics.comp-ph].
[22] B.-B. Chen, Y. Gao, Y.-B. Guo, Y. Liu, H.-H. Zhao, H.-J. Liao, L. Wang, T. Xiang, W. Li, Z.Y. Xie, Phys. Rev. B 101 (22) (Jun 2020), https://doi.org/10.1103/physrevb.101.220409.
[23] Y. Shi, U.N. Niranjan, A. Anandkumar, C. Cecka, Tensor contractions with extended blas kernels on cpu and gpu, arXiv:1606.05696 [cs.DC].
[24] S.E. Korshunov, JETP Lett. 41 (Mar 1985) 263–266.
[25] R. G. Jha et al., 2023, in preparation.
[26] L. Tagliacozzo, T.R. de Oliveira, S. Iblisdir, J.I. Latorre, Phys. Rev. B 78 (Jul 2008) 024410, https://link.aps.org/doi/10.1103/PhysRevB.78.024410.
[27] D. Kadoh, H. Oba, S. Takeda, J. High Energy Phys. 04 (2022) 121, arXiv:2107.08769 [cond-mat.str-el].
[28] F.Y. Wu, Rev. Mod. Phys. 54 (Jan 1982) 235–268, https://link.aps.org/doi/10.1103/RevModPhys.54.235.
[29] H.H. Zhao, Z.Y. Xie, Q.N. Chen, Z.C. Wei, J.W. Cai, T. Xiang, Phys. Rev. B 81 (17) (May 2010) 174411, arXiv:1002.1405 [cond-mat.str-el].
[30] S. Wang, Z.-Y. Xie, J. Chen, N. Bruce, T. Xiang, Chin. Phys. Lett. 31 (7) (July 2014) 070503, arXiv:1405.1179 [cond-mat.stat-mech].
[31] M.T. Batchelor, J. Phys. A, Math. Theor. 50 (13) (Mar 2017) 131002, https://doi.org/10.1088/1751-8121/aa5fdc.
[32] C. Ibrahim, D. Lykov, Z. He, Y. Alexeev, I. Safro, in: 26th IEEE High Performance Extreme Computing, 9, 2022, arXiv:2209.02895 [quant-ph].