

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*

raghav.govind.jha@gmail.com, asamlodia@gmail.com

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 and its scaling with bond dimension for two-dimensional systems. Our results establish that the utilization of GPU resources is essential for future precision computations with TRG.

Contents

1	Introduction	1
2	Algorithm and TORCHTRG discussion	2
3	Models and Results	7
3.1	Classical GXY model	7
3.2	Ising model	8
3.3	Three-state Potts model	10
4	Summary	10
A	Contraction of network - Different methods	12

SECTION 1

Introduction

Tensor networks are undoubtedly the state-of-the-art classical approach to efficiently study classical/quantum spin systems in lower dimensions. This has a long history but the fundamental realization is 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-defined 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], 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 [3–6]. We refer the interested reader to the review article [7] 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 among them 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 [4]. 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 and often has problem reproducing the correct fixed-point tensor.

A major part of the computation time is 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 and HOTRG have made extensive use of parallel CPU computing to speed up the computations and have obtained some good results.

The unreasonable effectiveness of tensors is not just restricted to describing 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 [8] is a Python package that provides 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), 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` [9] improves the runtime by a factor of $\sim 12\times$ with $D = 89$ for the generalized XY model (described in Sec. 3) and lowers the cost of TRG from $\mathcal{O}(D^7)$ down to $\mathcal{O}(D^5)$. We also present results for the Ising model and the 3-state Potts model as reference for the interested reader and how one can obtain results with same precision in less computer time. We refer to the use of PYTORCH for TRG computations with CUDA as TORCHTRG¹ in this article.

SECTION 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 [7] for details. The goal of this algorithm is

¹The code used in this paper can be obtained from the authors.

to effectively carry out the coarse-graining of the tensor network with controlled truncations by specifying a local bond dimension D which is kept constant during the entire algorithm. The computer time for the higher-order TRG algorithm scales as $O(D^{2d-1})$ for d -dimensional Euclidean systems. We show the algorithm in Fig. 1 for the reader. The main bottleneck in these computations (especially for higher dimensions) is SVD and the contraction of tensors to keep the growing size fixed to a reasonable value depending on resources. In an earlier work by one of the authors [10], to perform the most expensive part of the computations - 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` [11] which was used for standard CPU computations in [12]. 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 [13] based on the software package developed in Ref. [14]. 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. [15]. The availability of additional GPUs also improves the runtime. 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 [9].
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

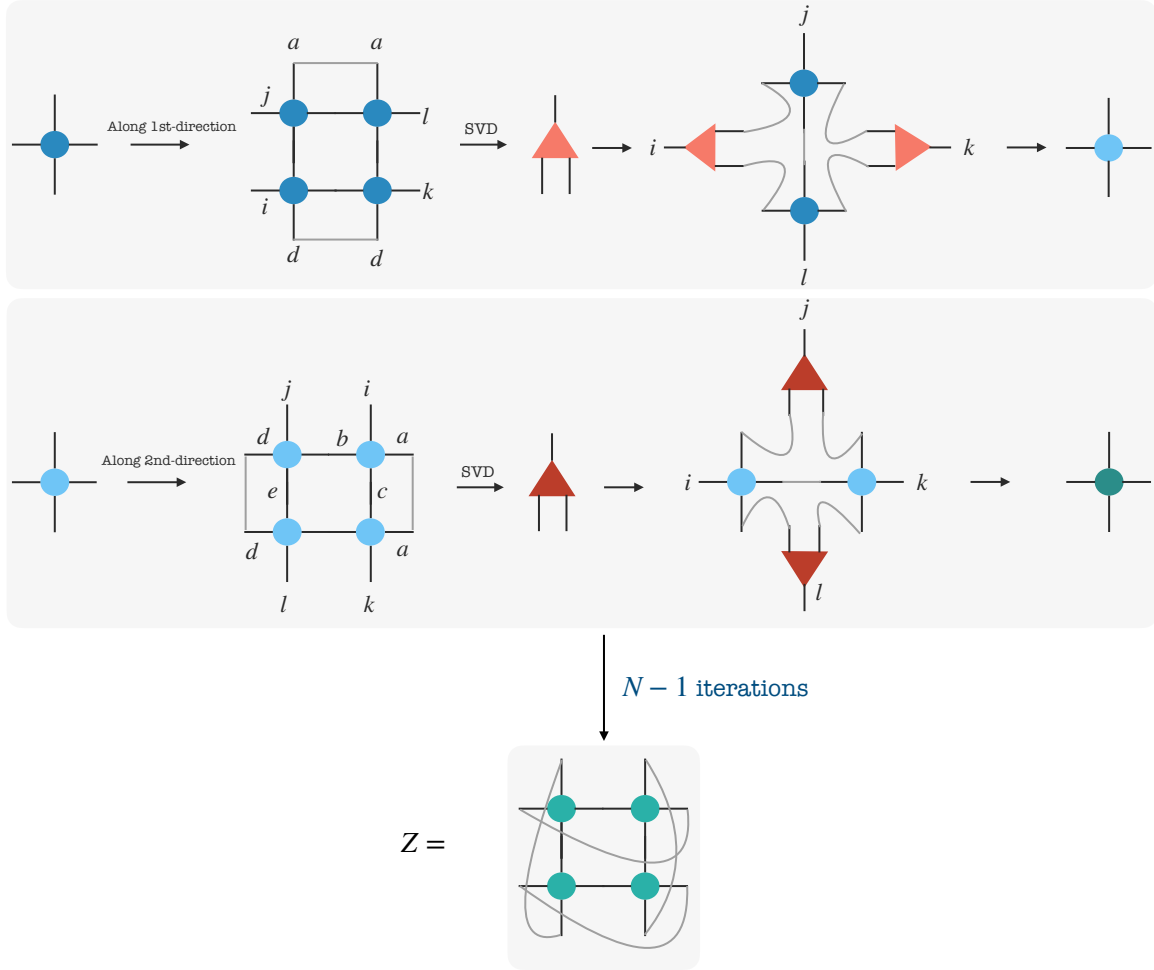


Figure 1. Schematic representation of the TRG implemented in this work. The diagram is to be seen from top to bottom, with the first two panels denoting the coarse graining along two directions. Then repeating this $N - 1$ times, we obtain the resulting tensor for a $2^N \times 2^N$ lattice which we then contract as shown to get the partition function Z with periodic boundary conditions.

implementation of the basic linear algebra subroutines (BLAS) [11, 16]. 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 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.

```
1 import numpy as np # NumPy version 1.21.6
```

```

2 import scipy as sp # SciPy version 1.7.1
3 import torch # Torch version 1.10.1+cu102
4
5 # Import PyTorch. pip install torch usually works.
6 use_cuda = torch.cuda.is_available()
7 # 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. [9] 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 $A_{ijkl}B_{pqkl} \rightarrow C_{ipkq}$, we do: `C = ee.einsum('ijkl,pjql->ipkq', A, B)` with `use_cuda == True` or `C = contract('ijkl,pjql->ipkq', A, B)` otherwise.

```

1 if use_cuda:
2     print('__CUDNN VERSION:', torch.backends.cudnn.version())
3     print('__Number CUDA Devices:', torch.cuda.device_count())
4     print('__CUDA Device Name:', torch.cuda.get_device_name(0))
5     print('__CUDA Device
6         TotalMemory[GB]:', torch.cuda.get_device_properties(0).total_memory/1e9)
7
8     # __CUDNN VERSION: 7605
9     # __Number CUDA Devices: 1
10    # __CUDA Device Name: NVIDIA GeForce RTX 2080 Ti
11    # __CUDA Device Total Memory [GB]: 11.554717696
12
13    from opt_einsum_torch import EinsumPlanner
14    # To install use: pip install opt-einsum-torch
15    ee = EinsumPlanner(torch.device('cuda:0'), cuda_mem_limit = 0.8)
16
17 else:
18     from opt_einsum import contract
19     # 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.

```
1 # Sometimes to tackle the error above, doing the below works.
2 os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:<size here>"
3
4 # Tuning the cuda_mem_limit also helps.
5 ee = EinsumPlanner(torch.device('cuda:0'), cuda_mem_limit = 0.7)
```

In implementing the CUDA for TORCHTRG, we explored four models that can be selected at run time by the user. The choices are:

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

This is part of the four command line arguments: temperature $1/\beta$, bond dimension D , number of iterations, and the model name. 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 accomodate 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:

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

```

10     U = U[:, :D]
11     U = torch.reshape(U, tuple(left_index_sizes + [D])) if use_cuda else
np.reshape(U, left_index_sizes + [D])
12     return U
13
14     def coarse_graining(t):
15         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)
16         U = SVD(Tfour, [0,1], [2,3], D_cut)
17         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)
18         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)
19         U = SVD(Tfour, [0,1], [2,3], D_cut)
20         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)
21         norm = torch.max(Txy) if use_cuda else np.max(Txy)
22         Txy /= norm
23         return Txy, norm

```

SECTION 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 and accurately determine the transition temperature corresponding to the continuous phase transition.

3.1 Classical GXY model

The generalized XY (GXY) model is a nematic deformation of the standard XY model [17]. The Hamiltonian (with finite external field) is given by:

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

where we follow the standard notation $\langle ij \rangle$ to denote nearest neighbours 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

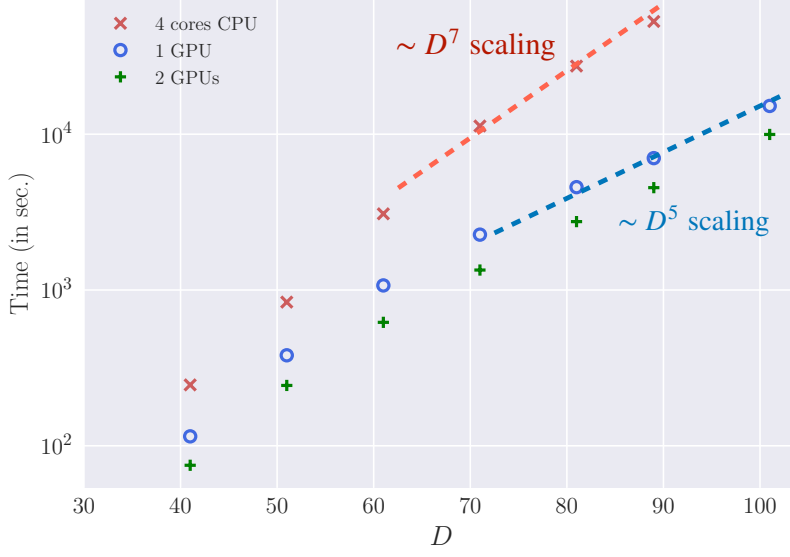


Figure 2. The runtime in seconds for the GXY model with different D on lattice of size $2^{30} \times 2^{30}$ with CPU version and TORCHTRG. We used maximum $D = 101$ with TORCHTRG and $D = 89$ with the standard CPU version to compare the timings.

separate work [18]. The reason we consider this model to compare the run time on CPU and with CUDA is that the effect of truncation (i.e., finiteness of D) is injected into the TRG from the beginning even before we start to coarse-grain the system because of the continuous global $O(2)$ symmetry. We will only consider $h = 0$ in this work for which this symmetry is preserved. 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. 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 RTX² and found a further speedup of $\sim 1.3x$ (not shown in the figure) over two CUDA RTX 2080 Ti for range of D . It is clear that with better GPU architectures in the future, TRG computations will benefit significantly from moving over completely to GPU-based computations.

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

²We thank Nobuo Sato for access to the computing facility.

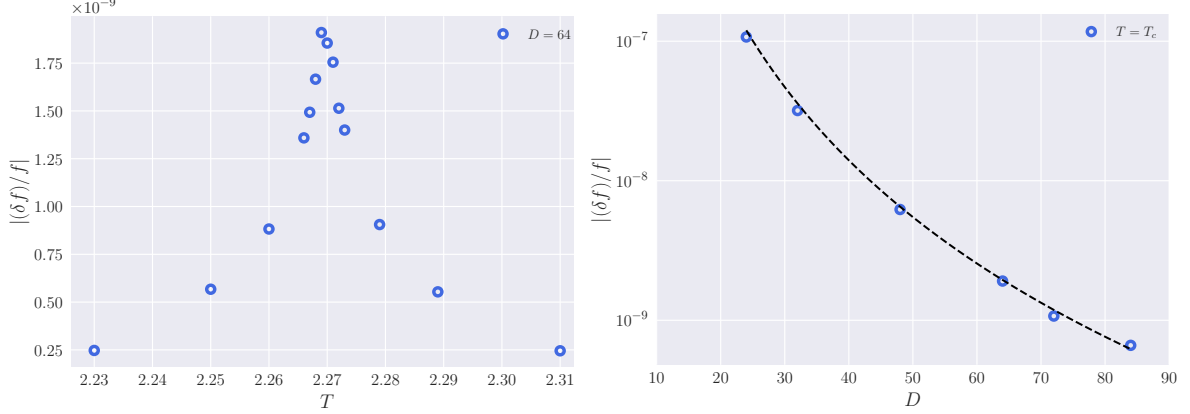


Figure 3. **Left:** The deviation of the TRG results from the exact result (3.2). **Right:** The dependence of the error on D and therefore on the execution time at $T = T_c$. The largest D took about 10,000 seconds on 4 CPU cores.

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:

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

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:

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

We show the results for this observable for various T in the left panel of Fig. 3 and at fixed $T = T_c$ for various D (run time) in the right panel of Fig. 3. Each data point in the left panel of Fig. 3 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. 3. A simple fit gives $\alpha = 4.20(3)$. For this model, we compared

our numerical results with two other recent works. The triad second renormalization group introduced in [19] 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. [15] 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.

3.3 Three-state Potts model

As a generalization of the Ising model, we can also consider the classical spins to take values from $1, 2, \dots, q$. 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 $q < 4$ [20]. The exact analytical result for T_c on square lattice is:

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

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. [12, 21]. The initial tensor can be written down by considering the $q \times q$ Boltzmann nearest-neighbor weight matrix as:

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

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. For zero magnetic field, this model has a phase transition at $T_c \approx 0.995$ and we check this using TORCHTRG. The results obtained are shown in Fig. 4.

SECTION 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 $\sim 8\times$ speedup for $D = 89$ for the generalized XY model on $2^{30} \times 2^{30}$ lattice using a single GPU which increases to $\sim 12\times$ using two GPUs. For a larger bond dimension of $D = 105$, there is an estimated $\sim 15\times$ speedup. The scaling of computation time scales like $\sim O(D^5)$ with GPU acceleration which is to be

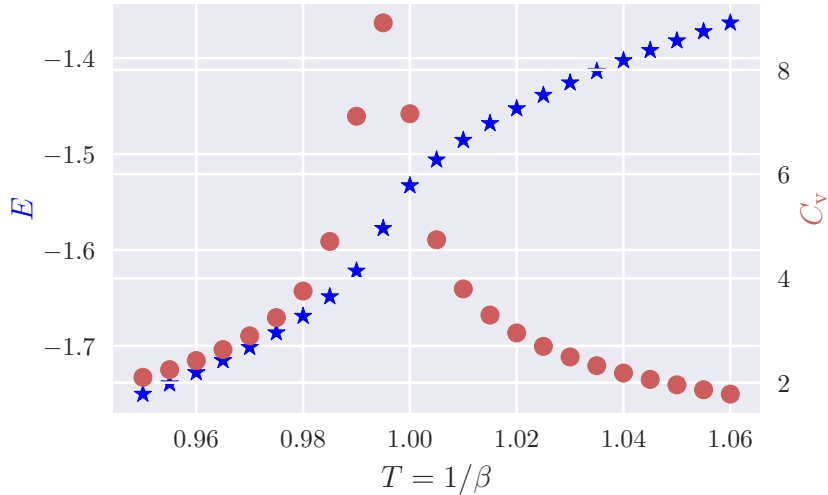


Figure 4. The internal energy (E) and specific heat (C_v) for the $q = 3$ Potts model with $D = 64$ on a lattice of size $2^{20} \times 2^{20}$. The continuous transition from the peak of specific heat is consistent with the exact analytical result. Each data point in the plot took about 1300 seconds using TORCHTRG on 2 CUDA devices.

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. 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.

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 its hospitality during the completion of this work. The numerical computations were done on Symmetry which is Perimeter Institute’s HPC system.

SECTION 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. 5, 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 [22].

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 `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.

```

1  import numpy as np
2  from opt_einsum import contract
3  from ncon import ncon
4
5  i, j, k, l = 80, 75, 120, 120
6  A = np.random.rand(i, j, k, l)
7  B = np.random.rand(j, i, k, l)
8
9  %timeit np.tensordot(A, B, axes=([1,0], [0,1]))
10 # 2.72 s \pm 38.3 ms per loop
11
12 %timeit np.einsum('ijkl,jiab->klab', A, B)

```

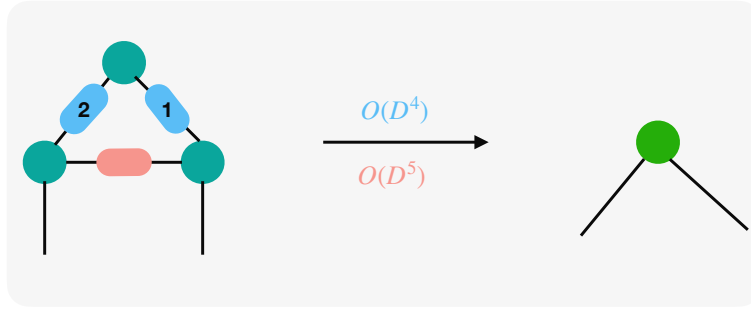


Figure 5. Schematic representation of two ways of contracting a network. The cost is $O(D^4)$ if we follow the order of blue-shaded regions as numbered. However, if we start by contracting the pink link first, then the leading cost will be $O(D^5)$.

```

13 # WARNING: Never use this without optimization.
14 # Slower by factor of 500x or so! Not considered.
15 # We can turn the optimize flag as below.
16
17 %timeit np.einsum('ijkl,jiab->klab', A, B, optimize=True)
18 # 2.75 s \pm 40.2 ms per loop
19
20 %timeit contract('ijkl,jiab->klab', A, B)
21 # 2.69 s \pm 40.9 ms per loop
22
23 %timeit ncon((A, B),([1,2,-1,-2], [2,1,-3,-4]))
24 # 2.37 s \pm 20 ms per loop
25
26 i, j, k, l = 200, 100, 80, 80
27 A = np.random.rand(i, j, k, l)
28 B = np.random.rand(j, i, k, l)
29
30 import torch
31 from opt_einsum_torch import EinsumPlanner
32 ee = EinsumPlanner(torch.device('cuda:0'), cuda_mem_limit = 0.7)
33
34 %timeit contract('ijkl,jiab->klab', A, B)
35 # 6.57 s \pm 80.7 ms per loop [on CPU]
36
37 %timeit ee.einsum('ijkl,jiab->klab', A, B)
38 # 3.76 s \pm 16.9 ms per loop [on CUDA]
39 # For this single contraction, we see a factor of about 1.7!

```

References

- [1] J. I. Cirac, D. Pérez-García, N. Schuch, and F. Verstraete, “Matrix product states and projected entangled pair states: Concepts, symmetries, theorems,” *Rev. Mod. Phys.* **93** (Dec, 2021) 045003. <https://link.aps.org/doi/10.1103/RevModPhys.93.045003>.
- [2] M. Levin and C. P. Nave, “Tensor renormalization group approach to two-dimensional classical lattice models,” *Phys. Rev. Lett.* **99** (Sep, 2007) 120601. <https://link.aps.org/doi/10.1103/PhysRevLett.99.120601>.
- [3] A. Bazavov, S. Catterall, R. G. Jha, and J. Unmuth-Yockey, “Tensor renormalization group study of the non-Abelian Higgs model in two dimensions,” *Phys. Rev. D* **99** no. 11, (2019) 114507, [arXiv:1901.11443](https://arxiv.org/abs/1901.11443) [hep-lat].
- [4] J. Bloch, R. G. Jha, R. Lohmayer, and M. Meister, “Tensor renormalization group study of the three-dimensional O(2) model,” *Phys. Rev. D* **104** no. 9, (2021) 094517, [arXiv:2105.08066](https://arxiv.org/abs/2105.08066) [hep-lat].
- [5] T. Kuwahara and A. Tsuchiya, “Toward tensor renormalization group study of three-dimensional non-Abelian gauge theory,” *PTEP* **2022** no. 9, (2022) 093B02, [arXiv:2205.08883](https://arxiv.org/abs/2205.08883) [hep-lat].
- [6] S. Akiyama and Y. Kuramashi, “Tensor renormalization group study of (3+1)-dimensional \mathbb{Z}_2 gauge-Higgs model at finite density,” *JHEP* **05** (2022) 102, [arXiv:2202.10051](https://arxiv.org/abs/2202.10051) [hep-lat].
- [7] Y. Meurice, R. Sakai, and J. Unmuth-Yockey, “Tensor lattice field theory for renormalization and quantum computing,” *Rev. Mod. Phys.* **94** no. 2, (2022) 025005, [arXiv:2010.06539](https://arxiv.org/abs/2010.06539) [hep-lat].
- [8] A. Paszke et al., “Pytorch: An imperative style, high-performance deep learning library,” 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.
- [9] H. Huo. <https://pypi.org/project/opt-einsum-torch/>.
- [10] R. G. Jha, “Critical analysis of two-dimensional classical XY model,” *J. Stat. Mech.* **2008** (2020) 083203, [arXiv:2004.06314](https://arxiv.org/abs/2004.06314) [hep-lat].
- [11] D. Smith and J. Gray, “opt_einsum - A Python package for optimizing contraction order for einsum-like expressions,” *Journal of Open Source Software* **3(26)** (2018) 753.
- [12] R. G. Jha, “Tensor renormalization of three-dimensional Potts model,” [arXiv:2201.01789](https://arxiv.org/abs/2201.01789) [hep-lat].
- [13] A. Milsted, M. Ganahl, S. Leichenauer, J. Hidary, and G. Vidal, “TensorNetwork on TensorFlow: A Spin Chain Application Using Tree Tensor Networks,” [arXiv:1905.01331](https://arxiv.org/abs/1905.01331) [cond-mat.str-el].
- [14] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, “TensorNetwork: A Library for Physics and Machine Learning,” [arXiv:1905.01330](https://arxiv.org/abs/1905.01330) [physics.comp-ph].

- [15] B.-B. Chen, Y. Gao, Y.-B. Guo, Y. Liu, H.-H. Zhao, H.-J. Liao, L. Wang, T. Xiang, W. Li, and Z. Y. Xie, “Automatic differentiation for second renormalization of tensor networks,” *Physical Review B* **101** no. 22, (Jun, 2020) . <https://doi.org/10.1103/PhysRevB.101.220409>.
- [16] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, “Tensor contractions with extended blas kernels on cpu and gpu,” [arXiv:1606.05696 \[cs.DC\]](https://arxiv.org/abs/1606.05696).
- [17] S. E. Korshunov, “Possible splitting of a phase transition in a 2d xy model,” *JETP Letters* **41** (Mar, 1985) 263–266.
- [18] R. G. Jha et al. *in preparation*, (2023) .
- [19] D. Kadoh, H. Oba, and S. Takeda, “Triad second renormalization group,” *JHEP* **04** (2022) 121, [arXiv:2107.08769 \[cond-mat.str-el\]](https://arxiv.org/abs/2107.08769).
- [20] F. Y. Wu, “The potts model,” *Rev. Mod. Phys.* **54** (Jan, 1982) 235–268. <https://link.aps.org/doi/10.1103/RevModPhys.54.235>.
- [21] H. H. Zhao, Z. Y. Xie, Q. N. Chen, Z. C. Wei, J. W. Cai, and T. Xiang, “Renormalization of tensor-network states,” *Phys. Rev. B* **81** no. 17, (May, 2010) 174411, [arXiv:1002.1405 \[cond-mat.str-el\]](https://arxiv.org/abs/1002.1405).
- [22] C. Ibrahim, D. Lykov, Z. He, Y. Alexeev, and I. Safro, “Constructing Optimal Contraction Trees for Tensor Network Quantum Circuit Simulation,” in *26th IEEE High Performance Extreme Computing*. 9, 2022. [arXiv:2209.02895 \[quant-ph\]](https://arxiv.org/abs/2209.02895).