Overcoming Barriers to Scalability in Variational Quantum Monte Carlo

Tianchen Zhao
Department of Mathematics
University of Michigan
Ann Arbor, MI, USA
ericolon@umich.edu

Saibal De
Department of Mathematics
University of Michigan
Ann Arbor, MI, USA
saibalde@umich.edu

Brian Chen
Department of Mathematics
University of Michigan
Ann Arbor, MI, USA
chenbri@umich.edu

James Stokes
Flatiron Institute
Simons Foundation
New York, NY, USA
jstokes@flatironinstitute.org

Shravan Veerapaneni
Department of Mathematics
University of Michigan
Ann Arbor, MI, USA
shravan@umich.edu

ABSTRACT

The variational quantum Monte Carlo (VQMC) method received significant attention in the recent past because of its ability to overcome the curse of dimensionality inherent in many-body quantum systems. Close parallels exist between VQMC and the emerging hybrid quantum-classical computational paradigm of variational quantum algorithms. VQMC overcomes the curse of dimensionality by performing alternating steps of Monte Carlo sampling from a parametrized quantum state followed by gradient-based optimization. While VQMC has been applied to solve high-dimensional problems, it is known to be difficult to parallelize, primarily owing to the Markov Chain Monte Carlo (MCMC) sampling step. In this work, we explore the scalability of VQMC when autoregressive models, with exact sampling, are used in place of MCMC. This approach can exploit distributed-memory, shared-memory and/or GPU parallelism in the sampling task without any bottlenecks. In particular, we demonstrate GPU-scalability of VQMC for solving up to ten-thousand dimensional combinatorial optimization problems.

CCS CONCEPTS

• **Applied computing** → **Physics**; • **Computing methodologies** → **Distributed computing methodologies**; **Neural networks**; **Quantum mechanic simulation**.

KEYWORDS

variational inference, density estimation, normalizing flows, generative models, neural networks, GPU parallelization

ACM Reference Format:
Tianchen Zhao, Saibal De, Brian Chen, James Stokes, and Shravan Veerapaneni. 2021. Overcoming Barriers to Scalability in Variational Quantum Monte Carlo. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3458817.3476219

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC '21, November 14–19, 2021, St. Louis, MO
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476219

1 INTRODUCTION

The fact that the state space of a quantum system scales exponentially with the number of its constituents leads to an inevitable curse-of-dimensionality facing the exact simulation of generic quantum many-body systems.

In practice, approximate solutions are sufficient for most purposes and a number of successful variational methods based on the Rayleigh-Ritz principle have been developed, which, given a local Hamiltonian H, produce an estimate for the minimal eigenvalue $\lambda_{\min}(H)$ and a description of an associated eigenvector. Nevertheless, complexity-theoretic arguments suggest that the curse-of-dimensionality is ultimately unavoidable [1] and the investigation of scalable variational algorithms is an active field of research. A particularly promising variational algorithm from the viewpoint of scalability is the variational quantum Monte Carlo (VQMC) [24].

VQMC targets the ground eigenstate by performing alternating steps of Monte Carlo sampling from a high-dimensional quantum state followed by gradient-based optimization. By exploiting neural networks as trial wavefunctions, Carleo and Troyer [9] showed that VQMC can achieve state-of-the-art results for the ground state energies of physically interesting magnetic spin models. Unfortunately, the increased flexibility afforded by neural networks comes at the cost of rendering exact Monte Carlo sampling intractable, which necessitates the use of a Markov Chain Monte Carlo (MCMC) sampling strategy.

However, MCMC sampling limits the scalability of VQMC in two ways: (1) the burn-in process is an inherently sequential task; (2) sampling precise and uncorrelated samples become increasingly difficult for large input dimension. Autoregressive models, in contrast, provide efficient and exact computations for both sampling and density evaluation that are GPU-supported. Recently, autoregressive neural quantum states have been introduced [26], which has allowed the VQMC to enjoy the advantages that autoregressive models have previously provided in machine learning. Inspired by the ability of autoregressive models to eliminate the reliance of

the VQMC on the MCMC, we undertake a parallelization study of autoregressive neural quantum states, thereby improving the time-efficiency and scalability of VQMC.

2 BACKGROUND

In this section, we briefly explain the basics of VQMC, MCMC, and autoregressive models, and state the high-dimensional problems considered.

2.1 Variational Quantum Monte Carlo

We consider the problem of determining a minimal eigenpair of a large and sparse random real-symmetric matrix H admitting an efficient description in a sense to be made precise later. Moreover, we assume that all off-diagonal entries of H are non-positive so that the ground eigenvector can be chosen to be entry-wise non-negative real vector as a consequence of the Perron-Frobenius theorem. The sparsity assumption is summarized by the following requirement

Definition 2.1. A real-symmetric matrix $H \in \mathbb{R}^{N\times N}$ is row-s sparse and efficiently row computable if for each row index $x \in [N]$, the list of non-zero entries $\{(y, H_{xy}) : H_{xy} \neq 0\}$ is computable in time $O(s)$.

The specific matrices we will consider are motivated by many-body quantum Hamiltonians. The size of these matrices is a power of 2, that is, $N = 2^n$, and they have sparsity parameter $s = \text{poly}(n)$ with $n = O(\log N)$. These include as a special case quadratic unconstrained binary optimization (QUBO) problems such as Max-Cut [7].

Given a matrix H satisfying Definition 2.1, together with differentiable family of trial vectors indexed by $\theta \in \mathbb{R}^d$ described via a function $\psi_\theta : [N] \to \mathbb{R}$ which outputs components of the vector relative to the standard basis $\psi_\theta(x) = \langle e_x, \psi_\theta\rangle$, we define the VQMC learning problem as the following continuous stochastic optimization task,

$$\min_{\theta\in\mathbb{R}^d} L(\theta) \ , \quad L(\theta) := \frac{\langle\psi_\theta, H\psi_\theta\rangle}{\langle\psi_\theta,\psi_\theta\rangle} = \mathop{\mathbb{E}}_{x\sim\pi_\theta}\left[\frac{(H\psi_\theta)(x)}{\psi_\theta(x)}\right] \ , \tag{1}$$

where the expectation value is over the probability distribution

$$\pi_\theta(x) := \frac{\psi_\theta(x)^2}{\langle\psi_\theta,\psi_\theta\rangle} \ . \tag{2}$$

The population objective function (1) satisfies the variational inequality $L(\theta) \geq \lambda_{\min}(H)$ and can be concisely expressed as the expectation value of a function $l_\theta(x)$ (called the local energy for historical reasons),

$$L(\theta) = \mathop{\mathbb{E}}_{x\sim\pi_\theta}[l_\theta(x)] \ , \quad l_\theta(x) := \frac{(H\psi_\theta)(x)}{\psi_\theta(x)} \ . \tag{3}$$

It follows from Definition 2.1 that each entry of the matrix vector product $H\psi_\theta$ is computable in time $O(s)$ and thus $l_\theta(x)$ is also computable in time $O(s)$ given our sparsity assumption $s = \text{poly}(O(\log N))$. The variance of the stochastic objective under π_θ satisfies the identity.,

$$\begin{aligned}\mathop{\text{var}}_{x\sim\pi_\theta}\big(l_\theta(x)\big) &:= \mathop{\mathbb{E}}_{x\sim\pi_\theta}\left[\big(l_\theta(x) - L(\theta)\big)^2\right] \\ &= \frac{\langle\psi_\theta, H^2\psi_\theta\rangle}{\langle\psi_\theta,\psi_\theta\rangle} - \left[\frac{\langle\psi_\theta, H\psi_\theta\rangle}{\langle\psi_\theta,\psi_\theta\rangle}\right]^2 \ .\end{aligned} \tag{4}$$

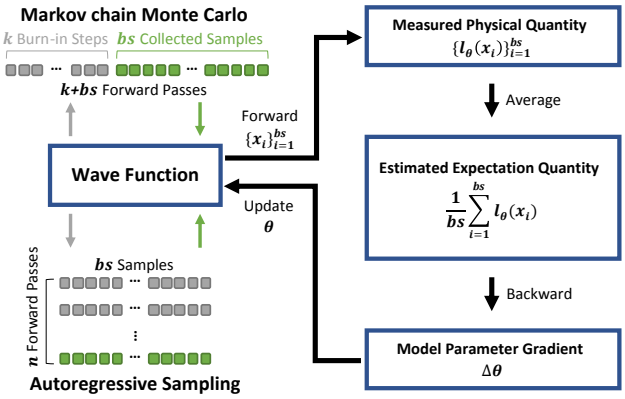

Figure 1: Overview of our algorithms, with illustrations of the comparison between Markov chain Monte Carlo sampling (MCMC) and autoregressive sampling (AUTO) on the left, and the VQMC optimization procedure on the right. MCMC sampling involves $k + bs/c$ forward passes, where k is the number of burn-in samples, c is the number of sampling chains ($c = 1$ in the figure) and bs is the batch size; AUTO only requires n forward passes to sample *exactly* from the distribution of interest.

Using the Rayleigh-Ritz principle it can be seen that the variance is vanishing if ψ_θ approaches any eigenvector of H. In practice, the objective function is optimized using stochastic natural gradient descent, also called stochastic reconfiguration (SR) [28], where the estimators for the gradient and the Fisher information matrix follow from the following population forms,

$$\begin{aligned}\nabla L(\theta) &= 2\mathop{\mathbb{E}}_{x\sim\pi_\theta}\left[\big(l_\theta(x) - L(\theta)\big)\nabla_\theta \log|\psi_\theta(x)|\right] \ , \\ I(\theta) &= \mathop{\mathbb{E}}_{x\sim\pi_\theta}\left[\nabla_\theta \log\pi_\theta(x) \otimes \nabla_\theta \log\pi_\theta(x)\right] \ .\end{aligned} \tag{5}$$

Typically the normalizing constant $\langle\psi_\theta,\psi_\theta\rangle$ of the probability distribution π_θ is unknown, so the above expectation values are to be approximated using MCMC sampling.

2.2 Markov Chain Monte Carlo Sampling

MCMC methods have been developed for sampling from a probability distribution π_θ that is difficult to directly draw *i.i.d.* samples from. The canonical Metropolis-Hastings algorithm [17] and its numerous variations, *e.g.*, Gibbs sampling [12], Reversible Jump MCMC [16] and Hamiltonian Monte Carlo [11, 19], achieve this by carefully constructing a transition kernel $p(x_{t+1}|x_t)$ for an ergodic Markov chain whose state distribution limits to the target distribution. Using samples from this Markov chain, we can then compute estimates for the expected values required in VQMC framework

$$\mathop{\mathbb{E}}_{x\sim\pi_\theta}[\phi(x)] \approx \overline{\phi}_T = \frac{1}{T}\sum_{t=1}^{T}\phi(x_t) \ , \tag{6}$$

where ϕ represents some deterministic function. Furthermore, these estimates are guaranteed to be asymptotically unbiased by the ergodic theorem.

Algorithm 1 Autoregressive Sampling [13] (Batch Size 1)

Input: Randomly initialized state x^0 of size n
Output: Sampled state x^* of size n
for i-th out of n iterations **do**
  Compute $p(x_i|x_{1:i-1}^{i-1})$ with a forward pass
  Sample $y_i \in \{\pm 1\}$ with $p(\cdot|x_{1:i-1}^{i-1})$
  Get x^i by updating x_i^{i-1} with y_i
end for
Set $x^* = x^n$

2.3 Autoregressive Models

Now we discuss the modeling assumptions which enforce normaliztion of the differentiable trial function $\psi_\theta : [N] \to \mathbb{R}$, and thus eliminate the need for MCMC sampling. An elegant method to impose normalization is to make use of an autoregressive assumption, which has recently been generalized to neural network quantum states in [18, 26]. Since we are targeting a ground eigenvector, which is known to be non-negative, we may assume without loss of generality that $\psi_\theta(x) = \sqrt{\pi_\theta(x)}$, thereby shifting the modeling assumption into the choice of a normalized distribution π_θ satisfying the following condition,

$$\pi_\theta(x) = \prod_{i=1}^{n} \pi_i(x_i|x_{i-1}, \ldots, x_1) \ . \tag{7}$$

Many proposals for neural networks satisfying the autoregressive assumption have been put forth. In this work we follow Germain et al. [13], who proposed the masked autoencoder for distribution estimation (MADE) which computes all conditionals in one forward pass using a single network with appropriate masks. Recall that a single hidden layer autoencoder is described by the following composition of functions,

$$g_1(x) = \max\{0, W_1 x + b_1\} \tag{8}$$
$$g_2(x) = \sigma(W_2 g_1(x) + b_2) \ , \tag{9}$$

and where the rectification and sigmoid functions are applied elementwise. MADE achieves the desired autoregressive assumption by appropriate application of binary masks M_1 and M_2 to the weight matrices defining the autoencoder, resulting in a MADE layer of the form

$$\begin{aligned} g_1(x) &= \max\{0, (M_1 \odot W_1)x + b_1\} \\ g_2(x) &= \sigma\left((M_2 \odot W_2) g_1(x) + b_2\right) \ , \end{aligned} \tag{10}$$

where $\odot$ denotes elementwise multiplication.

In Figure 1, we compare the sampling procedures between MCMC and AUTO (as described in Algorithm 1). MCMC involves $k + bs/c$ forward passes, where c is the number of sampling chains and bs is the batch size. Although the number of forward passes can be reduced by increasing the number of chains, the number of burn-in iterations k required for convergence is undetermined and cannot be parallelized. On the other hand, AUTO only requires n forward passes to sample exactly from the distribution of interest.

2.4 Quantum Hamiltonians and QUBO Problems

In this paper, we consider a family of matrices motivated by quantum physics, which are parametrized by $O(\text{poly}(n))$ real parameters $\alpha_i, \beta_i, \beta_{ij} \in \mathbb{R}$ as follows,

$$H = -\sum_{1 \le i \le n} \left(\alpha_i X_i + \beta_i Z_i\right) - \sum_{1 \le i < j \le n} \beta_{ij} Z_i Z_j \ , \tag{11}$$

where $X_i := I^{\otimes(i-1)} \otimes X \otimes I^{\otimes(n-i)}$ and $Z_i := I^{\otimes(i-1)} \otimes Z \otimes I^{\otimes(n-i)}$ are $2^n \times 2^n$ matrices defined in terms of the following elementary 2×2 matrices,

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \ , \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \ , \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \ . \tag{12}$$

It is easily verified that H meets the conditions of definition 2.1 with sparsity parameter $s = n$. In terms of the binary representation of the row index $x = 2^{n-1}x_1 \cdots 2^0 x_n$ and the column index $y = 2^{n-1}y_1 \cdots 2^0 y_n$, the matrix entries of H are given by

$$\begin{aligned} H_{xy} = &-\sum_{1 \le i \le n} \left(\alpha_i \delta_{x_1 y_1} \cdots \delta_{\neg x_i y_i} \cdots \delta_{x_n y_n} + \beta_i (1 - 2x_i)\right) \\ &- \delta_{xy} \sum_{1 \le i \le j \le n} \beta_{ij} (1 - 2x_i)(1 - 2x_j) \end{aligned} \tag{13}$$

and $\neg x_i$ denotes logical negation of $x_i \in \{0, 1\}$. For simplicity we imposed $\alpha_i \ge 0$ to ensure that the ground eigenvector can be chosen to be a non-negative vector as a consequence of the Perron-Frobenius theorem.

In the special case where $\alpha_i = \beta_i = 0$ and $\beta_{ij} = \frac{1}{4} L_{ij}$ where L is the adjacency matrix of an undirected graph $G = (V, E)$ of size $|V| = n$, the ground state problem coincides with the Max-Cut problem, and thus VQMC can be employed as a heuristic for approximate combinatorial optimization [15, 31], which is equivalent to natural evolution strategies [31].

3 RELATED WORK

The idea of utilizing neural network quantum states to overcome the curse of dimensionality in high-dimensional VQMC simulations was first introduced by Carleo and Troyer [9], who concentrated on restricted Boltzmann machines (RBMs) applied to two-dimensional quantum spin models. Sharir *et al.* [26, 27] introduced neural network quantum states based on the autoregressive assumption inspired by PixelCNN [29] and demonstrated significant improvement in performance compared to RBMs. The autoregressive assumption was subsequently explored in VQMC using recurrent neural wavefunctions [18]. Autoregressive models have also been used to solve statistical mechanics models in [30]. Since our focus is on the scalability of VQMC, particularly in situations where MCMC is expected to struggle, unlike [9, 18, 26] we consider non-geometrically local Hamiltonians without an underlying lattice structure. This also contrasts with the work of [25], who considered parallelization of VQMC using MCMC sampling but assuming geometric locality. It was recently shown [15, 31] that techniques from quantum VQMC literature [9] can be adapted for approximately solving combinatorial optimization problems.

Larochelle and Murray [23] proposed neural autoregressive distribution estimator (NADE) as feed-forward architectures. MADE [13] improves the efficiency of models with minor additional cost for

simple masking operations. For probabilistic generative models, unnormalized models such as RBM rely on approximate sampling procedures like MCMC, whose convergence time remains undetermined, which often results in the generation of highly correlated samples and deterioration in performance. Such sampling approximations can be avoided by using autoregressive models [5] that estimate the joint distribution by decomposing it into a product of conditionals by the probability chain rule, making both the density estimation and generation process tractable. Kingma et al. [21] used autoregressive models as a form of normalizing flow [22].

4 ALGORITHM PARALLELIZATION

Unlike standard Monte Carlo methods, MCMC cannot be parallelized easily. The fundamental limitation is easily seen: to generate a sample x_{t+1} from a Markov chain, we need to sample the transition kernel $p(\cdot|x_t)$, which requires knowledge of the immediate past state x_t. This sequential nature of the sampling immediately precludes any direct attempt at parallelizing the sampling process.

We could attempt to initialize multiple independent sampling chains; indeed, this is one of the standard approaches often implemented in Bayesian inference frameworks. But when sampling a high-dimensional distribution using random walk Metropolis-Hastings, it typically takes a very long time for the random walk to explore the parameter space. This significantly slows down the convergence of the estimates (6) to the true expectation value; furthermore, it is very difficult to determine *a priori* how many samples will be required for this convergence within a specified tolerance. In practice, MCMC first discards a pre-determined number of samples in each of the independent chains to avoid the transient Markov transitions (a.k.a. burn-in) and down-samples the remainder by selecting samples at regular intervals to reduce correlations (a.k.a. thinning). Any expectations are then computed based on this smaller set of selected samples. Improper choice of these parameters can severely degrade the quality of the generated estimates. Furthermore, they also reduce the parallel efficiency; suppose k samples are discarded as burn-in and every j-th samples are selected during thinning. Then constructing n samples on each of L independent computing units will lead to a parallel efficiency of

$$\frac{k+(nL-1)j+1}{k+(n-1)j+1} = 1 + \frac{nj}{k+(n-1)j+1}(L-1) = a + bL \qquad (14)$$

for some a and b depending on k, j and n. Note that this calculation is solely focused on the sampling task, and therefore does not take into account any communications that might be necessary between the computing units for obtaining the final result. Even then, as the number of burn-in samples k is increased, the slope b decays from 1 towards 0 ($b = 1$ is indicative of optimal scaling).

On the other hand, an autoregressive model (AUTO) can generate exact samples from the target distribution. Although the implementation of AUTO has a sequential nature that scales linearly with the input dimension, it can generate independent samples from the target distribution by transforming *i.i.d.* samples from a simple distribution (*e.g.* Gaussian). This step is easily parallelized: as long as we have identical copies of the autoregressive model in a number of computing units (*e.g.* GPUs), we can construct independent samples in parallel. Communication between the computing units is necessary only when we need to update the parameters of the neural network, *e.g.* during a stochastic gradient descent update.

Our model consists of fully connected weight matrices; therefore as we scale up the problem size, the bottleneck for our algorithm is the memory usage. For example, assuming a GPU can only store models with up to 10M parameters, we can set the size of the hidden layer to 500 at maximum when solving a problem with 10K input dimensions. This limitation can be addressed along with two complementary but independent avenues:

(1) **Model Parallelization:** Distribute the model parameters across computing units, so that each unit needs to store and update a small part of the model.
(2) **Sampling Parallelization:** Use identical copies of the model across the computing units to generate only a few samples per unit, and combine the independent samples from all these units to construct an accurate expectation estimate.

The communication pattern between the computing units in model parallelization is intimately linked with the choice of the autoregressive neural network while the sampling parallelization is model agnostic.

In this work, we restrict our attention to only parallelizing the sampling step. Consider a quantum Hamiltonian of size $N = 2^n$ and an autoregressive model with two hidden layers of size h. Given a total number of L computing units/GPUs and a mini-batch size of mbs samples to be drawn on each GPU, we end up with an effective batch size of $bs = L \times mbs$. Locally, each process first generates mbs samples, then computes the physical measurements with the samples, and finally uses backpropagation to get the gradient of the model parameters. These local gradient vectors have length $d = 2hn + h + n$, which are averaged over the GPUs using a parallel reduction. Each GPU then updates its own model parameters locally.

The computation complexity can be estimated as follows: during the local sampling process on each GPU, the algorithm involves n forward passes for sampling, and a fixed number of forward passes for physical quantity measurements. The dominant cost of each forward pass is multiplication by $h \times n$ and $n \times h$ matrices, both $O(hn)$; this leads to a total computational cost of $O(hn^2 \times mbs)$ flops per GPU. Computing the average gradient over GPUs using parallel reduction costs further $O(hn)$ flops, and involves communication of $O(hn)$ floating point numbers. Clearly, the parallel efficiency is given by

$$\frac{O(hn^2 \times bs)}{O(hn^2 \times mbs) + O(hn)} = \frac{O(hn^2 \times L \times mbs)}{O(hn^2 \times mbs) + O(hn)} \qquad (15)$$

Since the constants in the $O(hn^2 \times L \times mbs)$ and the $O(hn^2 \times mbs)$ are the same, this ratio is approximately L when n or mbs are large.

5 EXPERIMENTAL RESULTS

This section contains an extensive evaluation of our approach. We first compare AUTO sampling and MCMC sampling in Section 5.2, where the advantage of AUTO in terms of computational efficiency becomes clear for problems of higher dimensions. The convergence performance is shown in Section 5.3. Our algorithm is competitive against the state-of-the-art SDP solvers for small/medium scale Max-Cut problems. In Section 5.4, we demonstrate the scalability of our technology by solving large-scale problems up to 10K dimensions.

Table 1: Training time (measured in seconds) comparison on TIM for 300 training iterations with one GPU. Our MCMC settings are introduced in Section 5.1. The running time of MADE&AUTO scales roughly linearly with respect to the number of dimensions, due to the sequential nature of its sampling procedure, but significantly outperforms RBM&MCMC in practice.

Model	Optimizer	Sampler	# of Dimensions				
			20	50	100	200	500
RBM	ADAM	MCMC	135.64	154.25	189.91	249.40	456.68
MADE	ADAM	AUTO	2.85	5.74	10.63	20.45	49.62

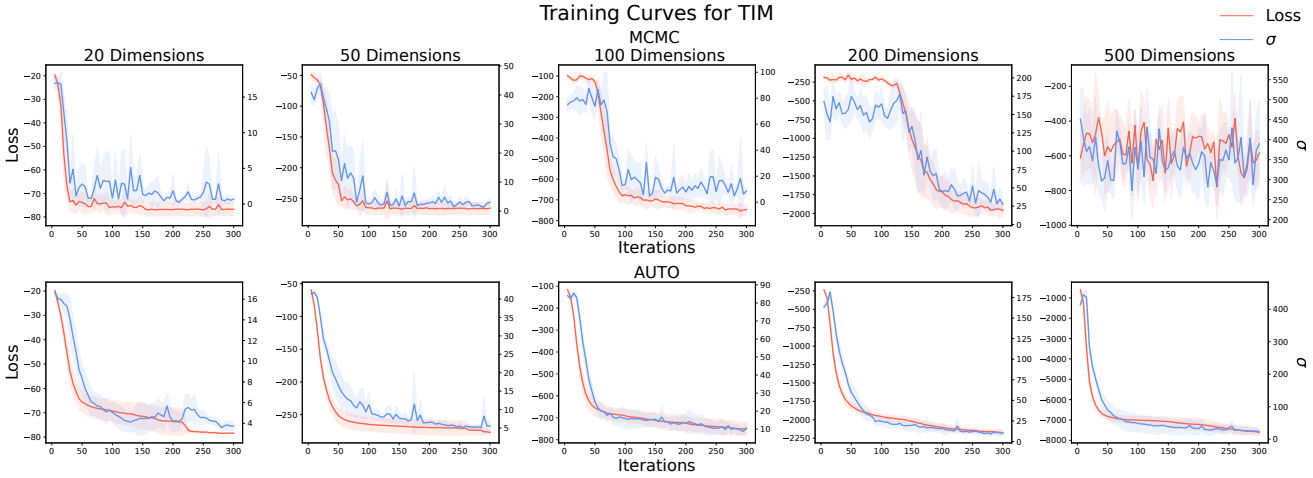

Figure 2: Training curves for TIM, where the red curves refer to the training loss/energy, and the blue curves refer to the standard deviation of the stochastic objective, which should be zero when the wave function converges to the exact ground-state. By fixing the learning rate and the total number of training iterations, it becomes more difficult for RBM&MCMC to converge as the problem size grows, due to the inaccurate estimation of the population energy by the low-quality MCMC samples. The training of MADE&AUTO is stable across all problems.

We achieved near-optimal weak scaling, and the convergence of our model improves as we increase the effective training batch size.

5.1 Experimental Setup

In this paper, we evaluate VQMC using two non-geometrically local Hamiltonians: the Max-Cut and the transverse field Ising model (TIM) model. In the case of Max-Cut, the adjacency matrix was chosen by forming the $n \times n$ matrix $(B + B^T)/2$ with $B_{ij} \sim$ Bernoulli(0.5) sampled once and fixed, followed by rounding and setting diagonal entries to zero. The second example is a disordered quantum system referred to as transverse field Ising model model, whose Hamiltonian is of the form (13) with $\beta_i, \beta_{ij} \sim U(-1, 1)$ and $\alpha_i \sim U(0, 1)$ sampled once and fixed.

For Max-Cut, we compare our approach against VQMC with MCMC sampling [15, 31], as well as the semidefinite programming (SDP) relaxation approximation algorithms including Goemans-Williamson Algorithm [14] and the Burer-Monteiro reformulation with the Riemannian Trust-Region method [2]. As an additional baseline, each model is also trained using the SR method. We benchmark the running time and converged energy of our model on TIM in our scalability experiments.

Model architecture

Network architecture is chosen to be MADE and is compared against RBM, proposed by Carleo and Troyer [9], taking the one-dimensional state as input and outputs the logarithmic probability amplitude.

The structure of MADE is as follows

$$\textit{Input} \xrightarrow{[bs,n]} \texttt{MaskedFC1} \xrightarrow{[bs,h]} \texttt{ReLU} \xrightarrow{[bs,h]} \texttt{MaskedFC2} \xrightarrow{[bs,n]} \texttt{Sigmoid} \xrightarrow{[bs,n]} \textit{Output},$$

and the structure of RBM is

$$\textit{Input} \xrightarrow{[bs,n]} \texttt{FC}_{n,h} \xrightarrow{[bs,h]} \texttt{Lncoshsum} \xrightarrow{[bs]} \textit{Output1} \xrightarrow{[bs,n]} \texttt{FC}_{n,1} \xrightarrow{[bs]} \texttt{Add } \textit{Output1} \xrightarrow{[bs]} \textit{Output}.$$

Here bs is the batch size and n is the number of dimensions. $\texttt{FC}_{a,b}$ is a fully connected layer with input size a and output size b; and `MaskedFC` is the masked version of `FC`, to remove the connections in the computational path of MADE. `Lncoshsum` refers to a series of linear and non-linear operations involving: 1) taking natural logarithm for each entry of the input tensor; 2) taking hyperbolic cosine for each entry of the input tensor; 3) summation over the last

dimension of the input tensor. The size of the tensor being passed to the next operator is indicated above the right arrows.

For large-scale problems with high dimensional input size n, we need to choose a proper latent size h to balance between the memory usage and the capacity of the model. In our experiments, we set $h = 5(\log n)^2$ as the hidden layer size for MADE and $h = n$ as the number of hidden units for RBM.

Training

All models are trained for 300 iterations. In our single-GPU experiments, at each iteration, the model is updated with a batch of 1024 training samples. For evaluation, we draw a batch of 1024 testing samples from trained model, and report their mean energy. Two base optimizers are considered: stochastic gradient descent (SGD) with learning rate 0.1 or ADAM with learning rate 0.01, where the latter is our default optimizer. In addition, we provide additional results on models trained using the SR [28] method for performance comparison. The SR optimization was performed using a regularization parameter $\lambda = 0.001$ and a learning rate 0.1. No learning rate scheduler is applied. For scalability experiments, each GPU is distributed with a constant mini-batch size mbs, and the effective batch size is $mbs \times L$, where L is the total number of GPUs available.

Our MCMC sampler is the random walk Metropolis–Hastings algorithm, running with two chains. We expect that it takes more effort for MCMC to converge for large-scale problems. Therefore, for each chain, we set heuristically the burn-in iterations k to scale linearly with respect to the input dimension n, *i.e.*, $k = 3n + 100$.

Throughout the experiments, the timing benchmarks are performed on NVIDIA Tesla V100 GPUs, with 32GB of memory for each.

5.2 MCMC vs. AUTO: Runtime

Despite the sequential nature of both MCMC and AUTO sampling, in practice, AUTO sampling can be operated with GPU in a straightforward fashion and exhibit superior running time efficiency. Our results on the running time comparison is shown in Table 1.

The running time of RBM&MCMC scales with the total number of iterations in each chain, which includes a fixed number of burn-in iterations that cannot be parallelized. In our setting, we set the number of chains to be 2, and burn-in iterations k that grows linearly with respect to the input dimension n. In principle, the running time of MCMC can be reduced further by increasing the number of chains or choosing a smaller k. However, a more severe problem of MCMC lies in the fact that the distribution of the samples generated by MCMC only converges to the distribution of interest asymptotically. As the input dimension increases, it becomes more difficult for the random walk Metropolis–Hastings algorithm to converge, which can potentially affect the quality of generated samples if k is not properly chosen. The running time of MADE&AUTO is dominated by the sampling time that scales linearly with respect to the input dimension n, which significantly outperforms its RBM&MCMC counterpart. More importantly, for AUTO, we know exactly the computational complexity needed to get correct samples from the distribution of interest, as opposed to MCMC that requires undetermined number of iterations to converge.

The corresponding training curves are shown in Figure 2, where the red curves refer to the training loss/energy, and the blue curves refer to the standard deviation of the stochastic objective, which approaches zero as the wave function converges to the exact ground-state, as discussed in Eq. 4. RBM&MCMC converges reasonably well on small-scale problems, but has more difficulty to converge as the problem scales up. On the other hand, our model converges rapidly and stably to low energy across problems of different scales. This observation motivates us to attempt to solve problems of even higher dimensions.

5.3 MCMC vs. AUTO: Convergence Study

The convergence result of our model on the Max-Cut problems is shown in Table 2, where we compare MADE&AUTO against the state-of-the-art SDP relaxation approximation algorithms developed in the past decades, as well as VQMC with RBM&MCMC.

Random Cut algorithm is a simple randomized 0.5-approximation algorithm that randomly assigns each node to a partition. Goemans and Williamson [14] improved the performance ratio from 0.5 to at least 0.87856, by making use of the semidefinite programming (SDP) relaxation of the original integer quadratic program. Burer and Monteiro [8] reformulated the SDP for Max-Cut into a non-convex problem, with the benefit of having a lower dimension and no conic constraint. The implementation of Goemans-Williamson Algorithm used the `CVXPY` [3, 10] package and the Burer-Monteiro reformulation with the Riemannian Trust-Region method [2] used `Manopt` toolbox [6], which essentially implements the optimization algorithm proposed by [20].

For evaluation, we constructed a problem instance for each Hamiltonian size $n \in \{20, 50, 100, 200, 500\}$ by randomly generating parameters defined in Eq. 11. For each problem instance, each algorithm was executed 5 times using 5 random seeds. In Table 2, we report the averaged result over problem instances of different sizes.

In general, MADE&AUTO slightly outperforms RBM&MCMC on small-scale problems, and the latter fails to converge for problems of input dimension 500, due to our constraint on the number of training iterations.

The natural gradient descent [4, 28] proved essential for converging to a good local optimum. We apply the SR to both VQMC methods and observe similar improvements: optimizers equipped with SR are consistently improved over all architectures. On the other hand, the performance of our algorithm with SR is competitive against the state-of-the-art SDP solvers on Max-Cut problems.

5.4 AUTO: Multi-GPU Scalability

By distributing the sampling task across multiple GPUs, our method can extend to large-scale problems (with input dimensions up to 10K) by reducing the mini-batch size mbs distributed to each GPU. The effective batch size depends on both mbs and the number of GPUs L available for training.

In Figure 3, we plot the normalized execution times for the 1K, 5K and 10K dimensional TIM problems as we vary the number of GPUs and the GPU distribution across nodes. We choose the minibatch sizes assigned to each GPU depending on the dimensionality of the problem so that the GPU memory is saturated. Note that for both

Table 2: Optimized objective (maximize cut number for Max-Cut, minimize ground state energy for TIM) values for different problem sizes and different optimizers, averaged over 5 runs with different random seeds. The first three rows in the Max-Cut section consist of results from running classical algorithms and serve as benchmarks. For the rest of the rows in the table, the batch size is fixed to be 1024. We note that MADE&AUTO achieves satisfactory performance in the sense that it's directly comparable with the SDP solvers on Max-Cut. On the other hand, RBM&MCMC takes longer to converge as the problem size grows, whereas the convergence of MADE&AUTO remains stable.

Problem	Model	Sampler	Optimizer	# of Dimensions				
				20	50	100	200	500
Max-Cut	Classical: Random			27.2 ± 2.2	150.4 ± 5.8	610.4 ± 11.6	2495.8 ± 42.8	15696.0 ± 16.8
	Classical: Goemans-Williamson			41.4 ± 2.0	194.2 ± 2.3	741.0 ± 11.1	2881.6 ± 14.4	17242.4 ± 37.3
	Classical: Burer–Monteiro			**43.0** ± 0.0	**200.0** ± 0.0	754.0 ± 3.0	**2928.0** ± 3.7	**17416.0** ± 23.13
	RBM	MCMC	SGD	41.4 ± 1.5	192.0 ± 3.3	733.8 ± 13.0	2825.6 ± 5.5	15945.6 ± 44.2
			ADAM	40.6 ± 1.6	190.2± 2.7	719.8 ± 6.6	2777.6 ± 14.2	16576.0 ± 30.9
			SGD+SR	**43.0** ± 0.0	198.8 ± 1.5	758.0 ± 1.1	2898.0 ± 22.0	15956.8 ± 29.9
	MADE	AUTO	SGD	42.6 ± 0.4	192.0 ± 2.4	742.2 ± 5.9	2846.0 ± 4.8	16880.0 ± 73.6
			ADAM	42.4 ± 0.8	193.8 ± 3.1	733.8 ± 9.1	2847.8 ± 12.1	17006.6 ± 23.0
			SGD+SR	**43.0** ± 0.0	**200.0** ± 1.5	**758.4** ± 6.5	2909.2 ± 3.1	17176.6 ± 30.5
TIM	MCMC	RBM	SGD	-80.22 ± 2.79	-270.65 ± 9.64	-762.11 ± 28.58	-1981.17 ± 72.19	-976.25 ± 119.43
			ADAM	-80.38 ± 2.42	-265.47 ± 8.21	-756.33 ± 16.73	-2216.45 ± 31.95	-924.53 ± 121.10
			SGD+SR	-80.70 ± 2.10	**-282.02** ± 8.37	-764.74 ± 14.67	-2234.23 ± 36.72	-1046.40 ± 334.50
	MADE	AUTO	SGD	-80.30 ± 0.01	-281.18 ± 5.51	-767.88 ± 13.45	-1872.16 ± 41.89	-6773.97 ± 233.19
			ADAM	-80.48 ± 0.18	-277.11 ± 4.48	-771.11 ± 17.06	-2181.31 ± 33.39	-7597.37 ± 171.25
			SGD+SR	**-81.25** ± 0.07	-277.23 ± 9.96	**-812.33** ± 12.55	**-2252.12** ± 84.00	**-8673.27** ± 304.45

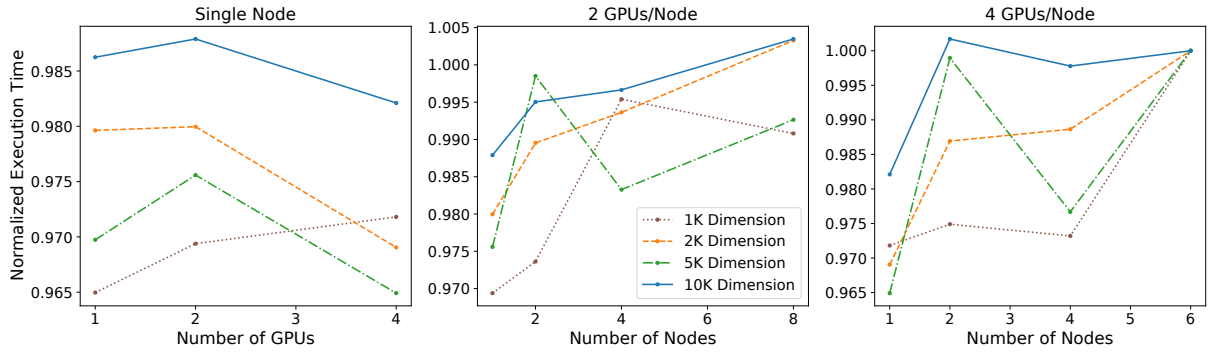

Figure 3: Sampling times for the TIM problem in 1K, 2K, 5K and 10K dimensions with mini-batch sizes *mbs* = 512, 128, 16 and 4 samples per GPU, respectively. The minibatch sizes were chosen to saturate GPU memory per problem dimension. All times are normalized by the execution time of the largest GPU configuration (6×4) for each dimension. Note that the normalized executions times are all close to 1, indicative of near-optimal weak scaling.

intra-node and inter-node distributed sampling schemes, the execution times remain nearly constant as long as the number of samples per GPU is kept fixed. This is indicative of near-optimal weak scaling: consider a problem so large that we are able to generate only a few samples using a single GPU due to memory constraints. In this scenario, by using a large number of GPUs to generate independent sets of samples, we should be able to drive the stochastic optimization problem to convergence.

The effective batch size increases as we scale up the number of GPUs. This improves the convergence performance of our method. We benchmark the result in Figure 4, where we train our models across different numbers of GPUs, on TIM problems of different sizes. The improvement saturates for smaller problems as the effective batch size increases but remains significant for larger problems. This implies that our model requires a larger batch size to achieve optimal performance for problems of a larger scale. Intuitively, batch size quantifies the exploration capability in the state space: the algorithm has a better chance to discover the ground state if it is allowed to explore more.

Table 3: Ablation study on the latent size. We train the models with ADAM on Max-Cut problems. n is the graph size. Optimal performance is obtained under a proper choice of latent size h; MADE falls off if we push GPU to its computational limits.

Model	n	Latent size h											
		Cut table							Time table				
		$(\log n)^2$	$3(\log n)^2$	$5(\log n)^2$	n	n	$5n$	n^2	$(\log n)^2$	$3(\log n)^2$	n	$5n$	n^2
MADE	50	191.	192.8	193.8	-	195.	194.6	195.	7.22	7.19	7.24	7.42	7.41
	100	735.8	737.2	733.8	-	734.2	731.2	726.2	13.43	13.49	13.48	13.90	13.96
	200	2832.8	2846.4	2847.8	-	2848.6	2821.4	2779.	26.49	25.78	26.07	26.85	57.19
	500	16905.4	17039.6	17006.6	-	16973.8	16872.8	16311.4	64.81	66.48	67.79	105.97	1426.92
RBM	50	193.	194.8	-	190.2	192.	192.2	191.4	151.07	151.49	150.72	150.71	152.68
	100	721.	734.2	-	719.8	730.2	711.	705.2	181.11	180.30	180.47	182.15	183.62
	200	2786.2	2810.8	-	2777.6	2779.6	2765.6	2747.4	242.95	241.05	243.24	243.91	246.05
	500	16568.8	16530.	-	16576.0	16652.6	16577.2	16543.	427.23	429.07	432.39	428.17	510.02

Table 4: Ablation study on the MCMC sampling scheme. We train the RBM with ADAM on Max-Cut problems. n is the graph size; $\{n, 10n\}$ and $\{\times 2, \times 5, \times 10\}$ are from Scheme 1 and Scheme 2, respectively.

Model	n	Sampling scheme										
		Cut table						Time table				
		n	$3n$+100	$10n$	$\times 2$	$\times 5$	$\times 10$	n	$10n$	$\times 2$	$\times 5$	$\times 10$
MCMC	50	190.8	190.2	193.8	191.6	192.6	192.8	110.44	197.02	199.64	500.02	1004.96
	100	700.2	719.8	733.	706.8	720.	729.8	124.01	296.83	201.52	507.65	1011.51
	200	2674.8	2777.6	2795.4	2670.4	2720.6	2736.8	143.76	492.31	206.91	514.80	1023.43
	500	16205.	16576.0	16626.6	16022.2	16066.6	16156.6	212.86	1103.18	207.43	508.43	1021.21

The raw data of our experiments in this section is provided in the appendix.

6 CASE STUDIES

In this section, we conduct experiments on several aspects of our settings in more detail, to support our conclusions that MADE+AUTO significantly outperforms RBM+MCMC in terms of the convergence rates for large-scale problems. Throughout this section, we train our models for Max-Cut problems with ADAM optimizer on a single GPU. All results are averaged over 5 runs with different random seeds.

6.1 Ablation Study: Latent Size

We conduct ablation studies on the choice of latent size for our models. Latent size refers to the number of hidden units for RBM and the hidden layer size for MADE.

In Table 3, we train both MADE and RBM on Max-Cut problems with graph sizes $n \in \{50, 100, 200, 500\}$ under different choices of latent size $h \in \{(\log n)^2, 3(\log n)^2, n, 5n, n^2\}$. We also cite the numbers from Table 2 for direct comparison, where we adopt $h = 5(\log n)^2, n$ for MADE and RBM, respectively. We measure the training time of each model for 300 iterations in seconds and present the numbers on the right side of the table. The results are averaged over 5 runs with different random seeds.

Several observations can be made. First, optimal performance is obtained under a reasonable choice of h, between $3(\log n)^2$ and n; models with a latent size that is either too large or too small do not perform well. Second, the time complexity usually does not scale with the model size when running on GPU. However, MADE falls off if we push GPU to its computational limits, *e.g.*, AUTO sampling $bs = 1024$ samples from MADE with $O(n^3)$ parameters. This is in practice not a serious concern for MADE with latent size $h = O((\log n)^2)$ as it will always face its memory bottlenecks first by storing the batch of high dimensional inputs as the problem size increases. Third, we re-did the experiments on RBM with n hidden units and obtain slightly different results in Table 2, due to different choices of random seeds and machines that the model is trained on.

6.2 Ablation Study: MCMC Sampling Scheme

We conduct ablation studies on the choice of MCMC sampling schemes. In particular, we consider:

- Scheme 1: the sampler discard the first $\{n, 10n\}$ samples in the chain and keep the next bs samples.
- Scheme 2: the sampler takes every $\{2, 5, 10\}$th sample in the chain until bs samples are collected in total.

In Table 4, we train RBM on Max-Cut problems with graph sizes $n \in \{50, 100, 200, 500\}$ under different choices of MCMC sampling schemes $\{n, 10n, \times 2, \times 5, \times 10\}$. We also cite the numbers from Table 2 for direct comparison, where we discard the first k=3n+100 samples in the MCMC chain. We measure the training time of each model for 300 iterations in seconds and present the numbers on the right side of the table. The results are averaged over 5 runs with different random seeds.

Several observations can be made. First, schemes $10n$ or $\times 10$ with longer MCMC chains result in better performance, at the cost of longer running time. Second, when running with GPU, the time

Table 5: Time elapsed to reach the target performance, measured in seconds. We train the RBM with ADAM on Max-Cut problems. At every iteration, after the training updates, we sample another batch of samples for evaluation; the algorithm terminates if the evaluation score surpasses the target score. Evaluation time is not taken into account.

Method	# of Dimensions (Targeted cut number)				
	20(41)	50(190)	100(730)	200(2800)	500(16800)
MADE+AUTO	3.14	3.61	20.08	3.25	6.27
RBM+MCMC	126.84	154.09	247.91	612.76	1096.08

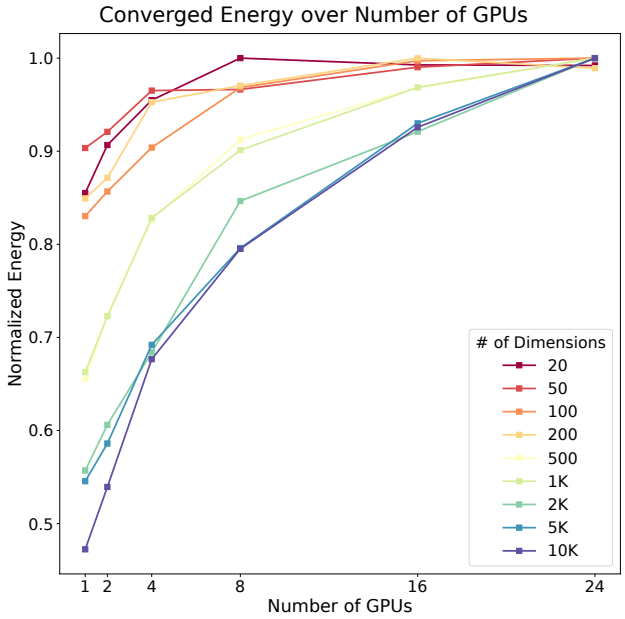

Figure 4: Normalized converged energy for TIM problems of different sizes. Each GPU is distributed a batch size of 4; the total effective batch size equals 4 times the total number of GPUs used. The energy is normalized for each problem size (the values from each curve are divided by the one with the largest magnitude among them). The converged energy improves as the total number of GPUs (effective batch size) increases. The improvement saturates for smaller problems, which also implies that a larger batch size is required for larger problems.

complexity only scales with the length of the MCMC chain, but not the $O(n^2)$ model size.

6.3 Comparison of Hitting Time

In addition to showing the running time with a fixed number of iterations in Table 1, we demonstrate that MADE+AUTO also significantly out-performs RBM+MCMC in the sense that the former reach a target performance faster.

In Table 5, we train MADE and RBM on Max-Cut problems with graph sizes $n \in \{50, 100, 200, 500\}$ with target performance $\{41, 190, 730, 2800, 16800\}$ that are heuristically chosen based on the results in Table 2. The performance is measured in seconds and the results are averaged over 5 runs with different random seeds. RBM+MCMC requires a significantly longer time to converge to a target performance for large-scale problems.

7 CONCLUSIONS

In this work, motivated by recent developments in VQMC made possible by autoregressive sampling, we implemented a distributed variant of VQMC and applied it to solving large-scale quantum systems for which standard random-walk Markov chain Monte Carlo sampling fails to converge. The main advantage of AUTO compared to MCMC lies in its ability to sample exactly from the distribution of interest, unlike MCMC for which the quality of the generated samples is plagued by unknown convergence time, which becomes a severe problem as the dimension of the problem increases. Empirically, we demonstrated that AUTO significantly outperforms MCMC in terms of the convergence rates for large-scale problems. Training of AUTO is also more stable than that of MCMC, finding converged solutions that are competitive against the state-of-the-art baselines for Max-Cut. The above findings motivated us to explore large-scale problems up to 10K dimensions. For that purpose, we built large models and chose a batch size to exhaust the memory usage of each GPU to be distributed. The optimality of our results is only limited by the computational resources available at hand: while the convergence performance quickly saturates for small-scale problems, it continues to improve for larger-scale problems as we scale up the number of GPUs.

ACKNOWLEDGEMENTS

Authors gratefully acknowledge support from NSF under grant DMS-2038030.

REFERENCES

[1] Scott Aaronson. 2009. Why quantum chemistry is hard. *Nature Physics* 5, 10 (2009), 707–708.
[2] P.-A. Absil, C. G. Baker, and K. A. Gallivan. 2007. Trust-region methods on Riemannian manifolds. *Foundations of Computational Mathematics* 7, 3 (2007), 303–330.
[3] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. 2018. A Rewriting System for Convex Optimization Problems. *Journal of Control and Decision* 5, 1 (2018), 42–60.
[4] Shun-Ichi Amari. 1998. Natural gradient works efficiently in learning. *Neural computation* 10, 2 (1998), 251–276.
[5] Yoshua Bengio and Samy Bengio. 2000. Modeling high-dimensional discrete data with multi-layer neural networks. *Advances in Neural Information Processing Systems* 12 (2000), 400–406.
[6] Nicolas Boumal, Bamdev Mishra, P.-A. Absil, and Rodolphe Sepulchre. 2014. Manopt, a Matlab Toolbox for Optimization on Manifolds. *J. Mach. Learn. Res.* 15, 1 (2014).
[7] Sergey Bravyi, David Gosset, Robert König, and Kristan Temme. 2019. Approximation algorithms for quantum many-body problems. *J. Math. Phys.* 60, 3 (2019), 032203.

[8] Samuel Burer and Renato D.C. Monteiro. 2001. A Nonlinear Programming Algorithm for Solving Semidefinite Programs via Low-rank Factorization. *Mathematical Programming (series B* 95 (2001), 2003.
[9] Giuseppe Carleo and Matthias Troyer. 2017. Solving the quantum many-body problem with artificial neural networks. *Science* 355, 6325 (2017), 602–606.
[10] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.
[11] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. 1987. Hybrid monte carlo. *Physics letters B* 195, 2 (1987), 216–222.
[12] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence* 6, 6 (1984), 721–741.
[13] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*. 881–889.
[14] Michel X Goemans and David P Williamson. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)* 42, 6 (1995), 1115–1145.
[15] Joseph Gomes, Keri A McKiernan, Peter Eastman, and Vijay S Pande. 2019. Classical quantum optimization with neural network quantum states. *arXiv preprint arXiv:1910.10675* (2019).
[16] Peter J Green. 1995. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika* 82, 4 (1995), 711–732.
[17] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (1970), 97–109.
[18] Mohamed Hibat-Allah, Martin Ganahl, Lauren E Hayward, Roger G Melko, and Juan Carrasquilla. 2020. Recurrent neural network wave functions. *Physical Review Research* 2, 2 (2020), 023358.
[19] Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.
[20] M. Journée, F. Bach, P.-A. Absil, and R. Sepulchre. 2010. Low-Rank Optimization on the Cone of Positive Semidefinite Matrices. *SIAM J. on Optimization* 20, 5 (May 2010), 2327–2351.
[21] Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. 2016. Improving variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems*.
[22] Ivan Kobyzev, Simon Prince, and Marcus Brubaker. 2020. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).
[23] Hugo Larochelle and Iain Murray. 2011. The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 29–37.
[24] W. L. McMillan. 1965. Ground State of Liquid He^4. *Phys. Rev.* 138 (1965), A442–A451. Issue 2A.
[25] Takahiro Misawa, Satoshi Morita, Kazuyoshi Yoshimi, Mitsuaki Kawamura, Yuichi Motoyama, Kota Ido, Takahiro Ohgoe, Masatoshi Imada, and Takeo Kato. 2019. mVMC—Open-source software for many-variable variational Monte Carlo method. *Computer Physics Communications* 235 (2019), 447–462.
[26] Or Sharir, Yoav Levine, Noam Wies, Giuseppe Carleo, and Amnon Shashua. 2020. Deep autoregressive models for the efficient variational simulation of many-body quantum systems. *Physical review letters* 124, 2 (2020), 020503.
[27] Or Sharir, Yoav Levine, Noam Wies, Giuseppe Carleo, and Amnon Shashua. 2020. FlowKet: an open-source library based on Tensorflow for running Variational Monte-Carlo simulations on GPUs. https://github.com/HUJI-Deep/FlowKet.
[28] Sandro Sorella. 1998. Green Function Monte Carlo with Stochastic Reconfiguration. *Physical Review Letters* 80, 20 (1998), 4558–4561.
[29] Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, koray kavukcuoglu, Oriol Vinyals, and Alex Graves. 2016. Conditional Image Generation with PixelCNN Decoders. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc.
[30] Dian Wu, Lei Wang, and Pan Zhang. 2019. Solving statistical mechanics using variational autoregressive networks. *Physical review letters* 122, 8 (2019), 080602.
[31] Tianchen Zhao, Giuseppe Carleo, James Stokes, and Shravan Veerapaneni. 2020. Natural evolution strategies and variational Monte Carlo. *Machine Learning: Science and Technology* 2, 2 (2020), 02LT01.

A RAW DATA FROM MULTI-GPU SCALABILITY EXPERIMENTS

We distribute the sampling task across multiple GPUs, our method can extend to large-scale problems with input dimensions up to 10K dimensions, by reducing the mini-batch size *mbs* distributed to each GPU. The effective batch size depends on both *mbs* and the number of GPUs L available for training. Here, we provide the raw data for our distributed computing experiments in Section 5.4.

In Table 6, we show the converged energy and running time for TIM problems of different dimensions. Each GPU is distributed with a batch size of 4; the total batch size equals to 4 times the total number of GPUs used. A number of different GPU configurations were used; $L_1 \times L_2$ indicates L_1 nodes with L_2 GPUs per node were utilized. The converged energy improves as the batch size (total number of GPUs) increases.

In Table 7, we show the running time (seconds) for TIM problems of different dimensions. Different from the experiments in Table 6, each GPU is distributed with the maximum number of batch size that can be accommodated on its memory. We note that for each dimension, the run times remain constant even as we increase the number of GPUs, increasing the effective batch size. This is indicative of near-optimal weak scaling.

Table 6: Converged energy and running time for TIM problems of different dimensions. Each GPU is distributed with a batch size of 4; the total batch size equals to 4 times the total number of GPUs used. Paralleling experiments are done across different GPU configurations, where $L_1 \times L_2$ refers to a total L_1 number of nodes with L_2 GPUs in each node, and the a total number of GPUs is $L = L_1 \times L_2$. The converged energy improves as the batch size (total number of GPUs) increases.

# GPUs	Metric	# of Dimensions								
		20	50	100	200	500	1000	2000	5000	10000
1×1	Energy	-69.64	-225.53	-656.91	-1511.22	-3862.86	-9642.54	-21962.55	-56337.84	-89733.83
	Time (s)	2.85	5.74	10.63	20.45	49.62	98.01	204.18	514.14	1067.56
1×2	Energy	-70.59	-260.91	-626.55	-1788.10	-4666.89	-12056.95	-24274.07	-73938.23	-142214.93
	Time (s)	3.06	6.00	10.81	20.36	49.47	97.29	200.32	512.39	1065.71
1×4	Energy	-82.79	-257.26	-702.94	-1778.35	-5587.58	-13797.55	-29219.47	-79650.12	-165364.75
	Time (s)	3.14	6.13	10.90	20.95	49.33	98.22	202.02	507.40	1066.03
2×2	Energy	-82.79	-257.26	-702.94	-1778.35	-5418.66	-13286.22	-28886.57	-74508.23	-159416.64
	Time (s)	3.29	6.16	10.81	20.63	49.59	98.01	204.90	512.80	1068.00
2×4	Energy	-81.49	-261.31	-766.29	-1984.61	-5886.93	-14826.83	-31665.81	-94311.98	-190800.37
	Time (s)	5.26	7.91	11.10	20.68	49.95	100.95	206.12	515.03	1085.33
4×2	Energy	-81.49	-261.31	-766.29	-1929.95	-5834.87	-14464.15	-33929.40	-93814.81	-200729.03
	Time (s)	3.55	6.22	10.92	20.60	49.86	97.98	202.73	513.87	1075.07
4×4	Energy	-81.70	-261.91	-776.00	-1892.16	-6348.56	-15636.99	-44506.68	-111165.27	-229567.37
	Time (s)	3.25	6.14	13.44	21.15	49.43	98.11	203.58	514.16	1068.51
8×2	Energy	-81.70	-261.89	-776.00	-1892.15	-5975.69	-15928.98	-46415.26	-120381.78	-224738.12
	Time (s)	3.30	6.18	10.88	20.77	49.97	98.29	203.80	520.13	1072.32
6×4	Energy	-80.99	-276.52	-769.72	-1950.40	-6672.37	-17105.77	-38496.40	-127652.29	-261517.21
	Time (s)	3.22	6.22	11.14	21.12	50.43	101.30	206.36	521.97	1067.83

Table 7: Running time (seconds) for TIM problems of different dimensions. Each GPU is distributed with the maximum number of batchsize that can be accommodated on its memory. A number of differnt GPU configurations were used; $L_1 \times L_2$ indicates L_1 nodes with L_2 GPUs per node were utilized. We note that for each dimension, the run times remain constant even as we increase the number of GPUs, increasing the effective batch size. This is indicative of near-optimal weak scaling.

# GPUs	# of Dimensions								
	20	50	100	200	500	1000	2000	5000	10000
	# of Samples per GPU								
	2^{19}	2^{17}	2^{15}	2^{13}	2^{11}	2^{9}	2^{7}	2^{4}	2^{2}
1×1	77.34	73.34	62.70	62.67	110.37	159.51	263.05	558.93	1058.85
1×2	76.30	73.74	62.88	62.24	110.93	160.24	263.14	562.30	1060.62
1×4	76.57	73.86	63.11	62.47	110.82	160.64	260.21	556.15	1054.41
2×2	76.24	73.82	63.02	62.56	111.20	160.94	265.71	575.51	1068.28
2×4	77.56	75.29	64.50	64.65	113.94	161.15	265.01	575.77	1075.45
4×2	76.32	73.86	63.03	62.35	111.31	164.54	266.81	566.73	1070.02
4×4	76.61	76.15	65.15	64.91	112.19	160.87	265.47	562.93	1071.24
8×2	77.01	75.13	64.59	65.27	112.46	163.78	269.40	572.13	1077.35
6×4	79.83	75.39	65.08	65.61	111.97	165.30	268.52	576.37	1073.62

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We implemented the VQMC algorithm discussed in the paper as an open source Python library. Our implementation supports parallel VQMC training across GPU nodes with various optimizers, samplers, and model architectures on different Hamiltonian types. In addition, for direct comparison, we provide codes to run the classical SOTA algorithms for Max-Cut, including Goemans-Williamson from CVX python toolbox and Burer-Monteiro from Manopt matlab toolbox. We provide .yml script to recover the anaconda environment we used to run our codes, and .sh scripts to reproduce both the experiments comparing MADE+AUTO with RBM+MCMC under the single-GPU setting, and the experiments running MADE+AUTO under multi-GPU setting. All experiments are done on the GPU nodes from Flatiron Institute Iron Cluster (Public:Instructions Iron Cluster - Simons Foundation), with 4 Nvidia Telsa V100 32 GB NVLinked GPUs and 40 CPUs per node.

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.4840621
Artifact name:
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Linux rusty2 5.4.83.1.fi #1 SMP Fri Dec 11 15:06:24 EST 2020 x86_64 x86_64 x86_64 GNU/Linux

Operating systems and versions: CentOS Linux 7 (Core), rhel fedora

Compilers and versions: : gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)

Libraries and versions: backpack-for-pytorch==1.2.0 cvxgraphalgs==0.1.2 cvxpy==1.1.11 matplotlib==3.4.1 networkx==2.5 numpy==1.20.1 opencv-python==4.5.1.48 pillow==8.1.2 pyyaml==5.4.1 tensorboardx==2.1 torch==1.8.0 torchvision==0.9 yacs==0.1.8

Key algorithms: Autoregressive sampling, MCMC, VMC

Input datasets and versions: There are no author-created data artifacts.