2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00081

GPU-Accelerated Error-Bounded Compression Framework for Quantum Circuit Simulations

Milan Shah
North Carolina State University
Argonne National Laboratory
Raleigh, NC, USA
mkshah5@ncsu.edu

Xiaodong Yu
Argonne National Laboratory
Lemont, IL, USA
xyu@anl.gov

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi1@anl.gov

Danylo Lykov
Argonne National Laboratory
University of Chicago
Lemont, IL, USA
dlykov@anl.gov

Yuri Alexeev
Argonne National Laboratory
Lemont, IL, USA
yuri@alcf.anl.gov

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

***Abstract*—Quantum circuit simulations enable researchers to develop quantum algorithms without the need for a physical quantum computer. Quantum computing simulators, however, all suffer from significant memory footprint requirements, which prevents large circuits from being simulated on classical supercomputers. In this paper, we explore different lossy compression strategies to substantially shrink quantum circuit tensors in the QTensor package (a state-of-the-art tensor network quantum circuit simulator) while ensuring the reconstructed data satisfy the user-needed fidelity.**

Our contribution is fourfold. (1) We propose a series of optimized pre- and post-processing steps to boost the compression ratio of tensors with a very limited performance overhead. (2) We characterize the impact of lossy decompressed data on quantum circuit simulation results, and leverage the analysis to ensure the fidelity of reconstructed data. (3) We propose a configurable compression framework for GPU based on cuSZ and cuSZx, two state-of-the-art GPU-accelerated lossy compressors, to address different use-cases: either prioritizing compression ratios or prioritizing compression speed. (4) We perform a comprehensive evaluation by running 9 state-of-the-art compressors on an NVIDIA A100 GPU based on QTensor-generated tensors of varying sizes. When prioritizing compression ratio, our results show that our strategies can increase the compression ratio nearly 10 times compared to using only cuSZ. When prioritizing throughput, we can perform compression at the comparable speed as cuSZx while achieving 3-4$\times$ higher compression ratios. Decompressed tensors can be used in QTensor circuit simulation to yield a final energy result within 1-5% of the true energy value.

***Index Terms*—compression, quantum computing, GPU**

I. Introduction

Quantum circuit simulators enable researchers to perform a series of non-trivial research tasks [1], [2], including verification of quantum advantage and supremacy claims, verification of quantum devices, co-design quantum computers, development of new quantum algorithms, and finding optimal parameters for variational quantum algorithms. One of the most advanced methods to simulate quantum circuits is using tensor networks [3], because it provides a fairly efficient way to manage and store the data. In this model, quantum circuits are formed as tensor networks, where nodes and edges represent tensors and the indices between tensors, respectively.

In this work, we adopt the QTensor [4] package – an open-source tensor network simulator, which involves two main steps in its simulation: finding an optimal tensor contraction sequence and actual contraction (multiplication) of tensors. The finding of the contraction sequence is done using various algorithms that treat the tensor network as a graph. The contraction sequence is a sequence of sets of tensor indices that are contracted at each contraction step. The actual tensor network contraction is typically done using a vendor-provided tensor library like cuTensor from NVIDIA [5].

While the QTensor package improves performance over previous approaches of tensor network circuit simulation [6], [7], memory can quickly become a bottleneck in the simulation of large circuits. QTensor uses the bucket elimination algorithm [8], which groups tensors into buckets and then contracts buckets one by one. Each bucket contains tensors that are indexed by the corresponding tensor index. As the bucket elimination algorithm advances, tensors can grow too large to fit in memory. The tensors we examine range from 0.5 GB to 4 GB, with simulations requiring many such tensors. Larger circuits require even larger tensors which can strain memory resources, especially for GPU-based simulations where GPU memory is on the scale of tens of gigabytes for a single GPU. For extreme-scale simulations of more complex quantum circuits, hundreds to thousands of GPUs are required, with tensor memory requirements becoming exponentially more demanding.

A straightforward solution to resolve this memory space issue is using data compression to shrink the memory footprint, which, however, faces several key challenges. First of all, lossless compressors [9]–[12] can guarantee the lossless nature of the reconstructed data but suffer from very low compression ratios [13]. In comparison, lossy compressors can significantly reduce the data size, while the data distortion introduced during compression may significantly affect the analysis results. Second, quantum simulation datasets are fairly high-dimensional (e.g., 26+ dimensions in a dataset) and the degree of data similarity is very low along each dimension. As such, quantum computing simulation datasets are treated as 1D data arrays, while the existing lossy compressors (such as SZ [14], ZFP [15]) are designed and optimized for multi-dimensional datasets so that they suffer very low compression ratios on quantum computing simulation data compression. For example, cuSZ [16] and cuSZx [17] with a relative error of 0.005 can only obtain compression ratios of 30 and 15-20 in the compression of quantum tensor datasets, respectively. On the other hand, since lossy compressors typically consist of multiple processing stages, there is an opportunity to tailor the compression pipeline to 1D datasets. Moreover, compression ratio (the ratio of raw data size to compressed data size) and compression/decompression speed cannot be both optimized together, which forces the definition of specific performance/compression tradeoffs for different use cases. Tensor compression can address several important use cases including: 1) reducing the footprint of large intermediate tensors during bucket elimination to free space for other tensors' contractions, 2) allow for tensor slices to occupy less memory during sliced contraction, and 3) alleviate storage costs of tensors generated during a partial contraction.

In this paper, we propose a novel GPU-based lossy compression framework that can compress floating-point data stored in quantum circuit tensors with optimized speed, while keeping the simulation result within a reasonable error bound after decompression. The compressed data can be decompressed when the tensors are needed during the computation. The key contributions are summarized as follows.

- We develop a novel configurable compression framework based on the characteristics of quantum circuit tensor datasets generated by QTensor [4]. Each compression pipeline of the framework is composed of three critical stages: data pre-processing, data compression, and data post-processing. The compression framework can adapt to different use cases in practice: either prioritizing the compression ratio or maximizing the compression/decompression throughput.
- We develop a variety of implementation choices for GPU that compose the lossy compression framework.
- We provide an in-depth analysis of the effects of lossy compression on quantum circuit simulation results, which is then leveraged to ensure the reconstructed data fidelity.
- We perform a comprehensive evaluation using an NVIDIA A100 GPU, which is fed with tensors of varying size and composition of values, based on up to 9 different compression strategies and analyze the data compression ratio, throughput, and resulting errors. Experiments show that our high-compression ratio compression pipeline with user-satisfied error-bound settings can yield nearly a 10× increase in compression ratio over two state-of-the-arts GPU-accelerated compression frameworks: cuSZ [16] and cuSZx [17]. When the throughput is prioritized, our high-speed compression pipeline exhibits comparable performance while still yielding 3-4× improvement in compression ratio over cuSZx.

II. Background

A. Quantum Circuit Tensors

Quantum circuit simulation usually computes either probability amplitudes for some quantum state of the system in question or some integral value that characterizes this state. The most common value in use is the energy of the system [4]. The key idea of tensor network-based simulators is to represent the simulation result as a tensor network. A quantum gate or state is represented as a tensor, and indices in the tensor network refer to which index of a bitstring a gate operates on. It is often useful to view a tensor network as a graph where tensors are nodes, and tensor indices are edges. If two tensors have the same index, they are connected by an edge. If an index is shared by more than two tensors, the graph becomes a hypergraph. QTensor prefers another notation where nodes and edges switch places: each node is a tensor index, and each (hyper-) edge is a tensor, as shown in Fig. 1. To perform the contraction, QTensor creates a list of buckets where each bucket has tensors indexed by a common index, i.e. edges of a particular node. Each contraction step contracts a bucket and produces a single tensor. A contraction step essentially is a series of tensor multiplications followed by a summation over a fixed index. As the bucket elimination algorithm advances, tensors tend to grow in size, straining the memory resources of the simulator. Fig. 1 shows how QTensor converts a quantum circuit to a tensor network and where tensors reside in the network. In QTensor's notation, nodes refer to unique indices, and edges denote a tensor.

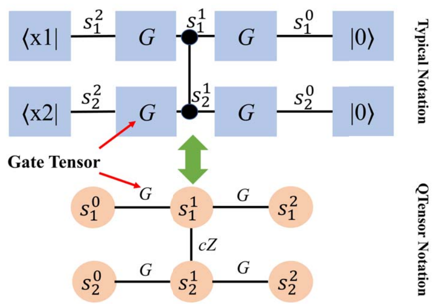


Fig. 1: Tensor networks of quantum circuits, typical notation above and QTensor notation below. s_i^t denotes the state of qubit i at cycle t. cZ is the tensor corresponding to the cross-connection of s_1^1 & s_2^1.

For QTensor and other simulators, tensors are composed of complex numbers with floating point values. Tensors with dimensionality d contain 2^d complex numbers, many of which can be values close to zero. Tensors are often high-dimensional but can be flattened to one dimension since there is low data similarity across a dimension. Casting tensors to other dimensions, such as 2-D or 3-D, does not lead to the increased similarity between adjacent data points. Fig. 2 illustrates the data of an example tensor with $d = 22$ flattened to one dimension. As seen in the right plot of Fig. 2, there can exist a periodicity of data points as well as relative sparsity. Most values are relatively small in magnitude compared to the maximum and minimum tensor values. Tensor data also lacks "smoothness" in that adjacent data points can be drastically different in value from each other.

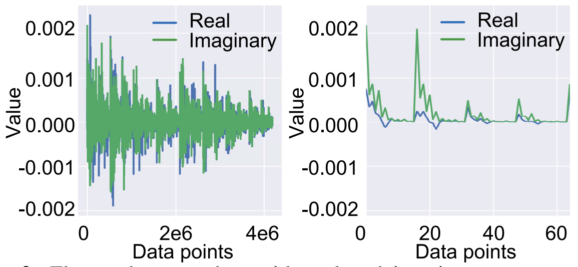


Fig. 2: Flattened tensor data with real and imaginary components plotted, tensor of dimension 22. The left plot shows all data points, and the right plot shows the first 128 data points.

B. Lossy Compression with (cu)SZ and (cu)SZx

In this work, we adapt two state-of-the-art compressors – *(cu)SZ* [18] [19] [14] [16] and *(cu)SZx* [17] for quantum tensor data due to their respective high compression ratio and high GPU speed capabilities. We do not use lossless compressors because their compression ratios are very low for scientific datasets. By contrast, *lossy* compressors incorporate point-wise error control that guarantees decompressed values are within the user-defined error-bounded range centered on their corresponding uncompressed value. Thus, lossy compressors can achieve higher compression ratios for scientific floating point data compared to their lossless counterparts. Since cuSZ and cuSZx are the GPU versions of SZ and SZx, respectively, we mainly describe SZ and SZx in the following text.

SZ adopts a prediction-based compression model, which includes three critical stages: (1) data prediction, (2) linear-scale quantization based on the user-specified error bound, (3) encoding the quantization code by lossless compression techniques such as Huffman encoding and dictionary encoding. The core step of SZ is the data prediction because higher data prediction accuracy can generate a sharper distribution of quantization codes, leading to higher efficiency in Huffman/dictionary encoding accordingly. As such, the key difference between various SZ versions is mainly in the distinct prediction methods. However, the most recent prediction methods [14] in SZ, such as dynamic spline interpolation, are heavily dependent on the high smoothness of the data, which generally works for multi-dimensional datasets instead of 1D data arrays. SZ still adopts a very primitive prediction method – 1D Lorenzo predictor to deal with the 1D data compression, as illustrated in Fig. 3.

Compared with SZ, SZx is designed for the purpose of extremely high throughput, so it has no data prediction step and Huffman/dictionary encoding step, which are the main bottleneck in SZ. Instead, SZx is composed of two critical steps: (1) dividing the dataset into equal-sized data chunks (a.k.a., blocks) and checking whether the data values in each chunk can be represented by one data value (called *constant chunk* if yes) or not (named *non-constant chunk*), as demonstrated in Fig. 4. (2) compressing the non-constant data chunks by counting the number of identical leading bytes (a.k.a., XOR leading-zero method) and truncating the insignificant bitplanes based on user-specified error bound. Since all the involved operations are only bitwise operations, addition, and subtraction, (cu)SZx is extremely fast in both compression and decompression: ∼4× as fast as SZ/ZFP on CPU and ∼10× as fast as cuSZ/cuZFP on GPU.

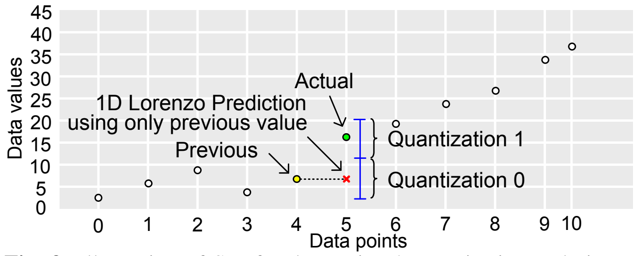


Fig. 3: Illustration of SZ: for data point 5, quantization code is set to 1 based on its prediction and quantization interval (2ϵ)

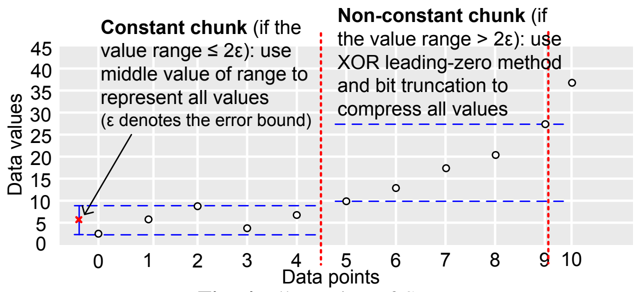


Fig. 4: Illustration of SZx

III. Related Work

There have been many existing generic error-bounded lossy compressors [15], [17], [18], [20], [21] proposed for the compression of scientific datasets. ZFP [15] is an orthogonal transform-based lossy compressor supporting two types of error controls: absolute error bound and precision mode. Compared with ZFP, SZ adopts the prediction-based lossy compression framework, which can better adapt to users' requirements and datasets as it allows to customize of a particular data predictor based on the characteristics of the datasets. FPZIP [20] is another prediction-based lossy compressor that adopts a different prediction method and a different encoding strategy. MGARD [21] is a Quantity of Interest (QoI) oriented lossy compressor, allowing users to specify a linear function to preserve during the compression. SZx [17] is an error-bounded lossy compressor designed to address the requirement of extremely high compression/decompression speed, so it achieves much lower compression ratios than other compressors such as SZ and ZFP. According to recent studies [17], [18], [22], SZ is arguably the most efficient error-bounded lossy compressor, especially for 1D datasets. Thus we mainly compare our proposed approach with SZ and SZx in our experiments without loss of generality.

In addition to the generic compressors, there are also some specific compression algorithms tailored for particular applications. PaSTRI is a lossy compressor designed for quantum chemistry electron repulsion integrals [22]. PaSTRI divides input data into blocks based on the pre-observed scaled patterns in the datasets. Unlike quantum chemistry electron repulsion integrals, however, the QTensor quantum computing simulation dataset does not have scaled patterns, so PaSTRI cannot be applied in the QTensor data compression. Shangnan proposed a lossless compression scheme for a quantum source and leveraged cross entropy to carry out compression [23]. As noted previously, lossless compression for scientific floating-point data typically suffers from very limited compression ratios, while a fairly high compression ratio is needed in the QTensor simulation data compression. Wu et al. [24] evaluated four candidate compression methods for quantum circuit simulation and identified the most effective compression method is combining XOR leading-zero data reduction and Zstd lossless compressor [9]. It cannot be applied in our situation because of the following two factors. On the one hand, Zstd does not support GPU devices, while we target the modern quantum computing simulation model on the GPU environment. On the other, that work targets state vector simulation, which can require much more memory compared to tensor network simulation. Tensor networks use at most the same memory as state vector simulation, but can provide a significant advantage for favorably structured circuits.

IV. Problem Formulation

We formulate the research problem as follows. Given a tensor dataset T, whose data points' original values are denoted as x, our objective is to develop an efficient error-bounded lossy compressor for the tensor data T, which can lead to both a high compression ratio and high throughput with little impact to the quantum circuit simulation results. Compression ratio (denoted by ***CR***) is defined as the ratio of the original raw tensor data size to the compressed data size: $\frac{|T|}{|T'|}$, where T' denotes the decompressed tensor data; $|T|$ and $|T'|$ represent the raw data size and compressed data size of the tensor, respectively. The compression throughput is defined as $\frac{|T|}{\tau_C(T)}$, where $\tau_C(T)$ refers to the compression time of the raw dataset (in seconds). Similarly, the decompression throughput is defined as $\frac{|T|}{\tau_D(T')}$, where $\tau_D(T')$ denotes the decompression time in getting the reconstructed data T'.

Specifically, our error-bounded compressor should address two important use cases: either maximizing the compression ratio or maximizing throughput on GPU, which are formulated as Formula (1) and Formula (2), respectively.

$$\begin{aligned}&\max\left(\frac{|T|}{|T'|}\right)\\ &s.t.\ \ |x_i - x_i'| \le \epsilon, \forall x_i \in T\\ &\qquad |P(T) - P(T')| \le 5\%, \text{WITH ERROR BOUND } \epsilon\end{aligned} \tag{1}$$

$$\begin{aligned}&\max\left(\frac{|T|}{\tau_C(T)}\right)\ and\ \max\left(\frac{|T|}{\tau_D(T)}\right)\\ &s.t.\ \ |x_i - x_i'| \le \epsilon, \forall x_i \in T\\ &\qquad |P(T) - P(T')| \le 5\%, \text{WITH ERROR BOUND } \epsilon\end{aligned} \tag{2}$$

where $P()$ stands for the post hoc analysis function. According to the QTensor developers, the P function should be the network contraction yielding energy, and the acceptable tolerance is 5% as suggested by QTensor developers. 5% error is a "good approximation" number that can be further tuned to suit users' needs.

V. Algorithmic Design

In this section, we present the key design of our solution.

A. Compression Pipeline

To boost the compressibility of quantum tensor data while preserving a contraction energy result in a reasonable error range, we propose a lossy compression framework composed of varying data pre-processing and post-processing stages for tensors. These stages leverage tensor sparsity to process data as well as compensate for the effect on energy value when modifying tensor data. Fig. 5 outlines the data pipeline integrating QTensor, SZ/SZx, and our pre-/post-processing stages. Green boxes indicate QTensor stages, and the orange region includes compression and decompression. Results from compression can be stored in memory or on disk. First, a quantum circuit is formatted to the QTensor tensor network model and fed to the start of the bucket elimination algorithm. The algorithm is stopped after n steps, equivalent to eliminating n buckets. At this point, tensors will have grown to be large and would need to be compressed to avoid straining memory resources. The pre-processing stage applies data transformations to tensors before forwarding the transformed tensors to cuSZ or cuSZx. Compressed tensors can be stored for later use or decompressed when needed for the bucket elimination algorithm. After decompression, a post-processing stage prepares tensors into a format ready for use in the remaining steps of the bucket elimination algorithm. Lastly, an energy result is calculated and can be used for circuit analysis. The effect of data distortions from this pipeline on the energy value and tensor fidelity is studied in the experimental evaluation.

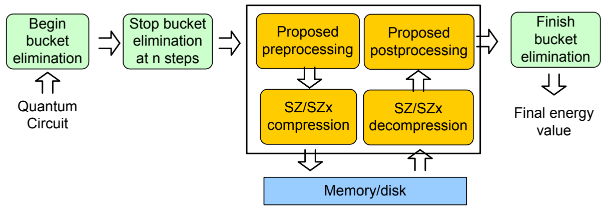


Fig. 5: Compression/Decompression Pipeline. In this work, we propose a lossy compression framework for QTensor data.

We design two strategies for pre-processing: a *threshold* method and a *threshold+grouping* method. A post-processing stage is required only for the threshold+grouping method. Stages work with a one-dimensional flattened tensor for increased data smoothness and to leverage 1-D data sparsity. The tensor is decomposed into real and imaginary components, and each component enters the compression stages separately. From these algorithmic strategies, many platform-specific choices compose a framework that can adjust throughput and compression ratio to the user's needs.

B. *Threshold Method*

Our fundamental idea is to increase the similarity/smoothness between data points by leveraging the intrinsic nature of quantum circuit tensors without significant loss of the reconstructed data fidelity. As discussed in Section II, the lossy compressors (such as cuSZ and cuSZx) strongly depend on the high smoothness of data, while the quantum circuit datasets are very spiky (as shown in Fig. 2). In fact, many tensor values are close to zero, but they are rarely exactly zero when represented in binary floating-point format. Across ten tensors ranging in size from 2^{24} to 2^{29} data points, 20-95% of values are less than 0.1% of the value range. If we increase the percentage to 1%, 60-90% of values are less than 1% of the value range for several MaxCut QAOA circuits, and 40-80% of values are less than 1% of the value range for several rectangle and Bristlecone lattice random quantum circuits. Compressors can exploit this sparse-like behavior of tensors if values are truly zero since the similarity between data points increases substantially. Another important aspect is that the contraction process involves many matrix multiplications. Thus, multiplying tensor values close to zero yield values that are even closer to zero. Converting a small value to zero will have little impact on the final energy value. This expectation forms the basis of our proposed methods and is quantified in Section VII.

The threshold method applies a filter to all data values of the input tensor. A threshold value signifies the boundary between data points that remain the same value in the tensor and those that are set to zero. If the absolute value of a data point is less than the threshold value, the data point is set to zero. Otherwise, the data point remains the same in the tensor. Formally, the filter $F(x)$ with threshold t is:

$$F(x) = \begin{cases} 0 & \text{if } |x| \leq t \\ x & \text{else} \end{cases} \tag{3}$$

Applying $F(x)$ to all data points $x \in T$ where T is the tensor can be easily parallelized since the filter is independently applied to each data point. Fig. 6 illustrates the threshold method implementation, showing data points from a sample tensor with the range of threshold values highlighted as "Threshold Width". $F(T)$ is the tensor result of the threshold method and has a length equal to the original tensor. x_i corresponds to the data point value at index i. After applying the threshold, SZ or SZx compresses $F(T)$ with no need for post-processing after decompression. Note that all reads and writes can be coalesced. Thus the threshold-only approach provides a low-cost pre-processing stage that can greatly increase the similarity between data points. For cuSZ and cuSZx, this similarity translates to more similar quantization codes and more blocks compressed as their mean only.

C. *Threshold+Grouping Method*

Using only the threshold method can affect the performance of cuSZ and cuSZx since many zero values must be loaded and processed even though these values are insignificant. The

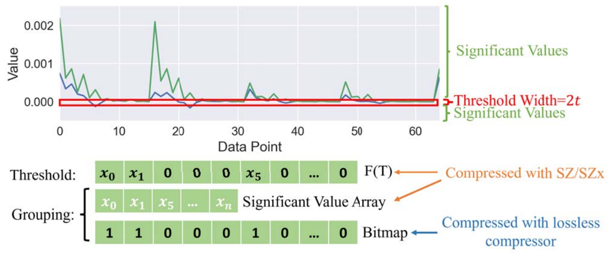


Fig. 6: Threshold and Grouping Methods.

metadata for zero values generated from cuSZ and cuSZx can negatively impact their compression ratios, thus removing zero values from the data sent to compression eliminates the need for compression metadata other than a bitmap (described next). cuSZx specifically is forced to perform bit representation compression if only one significant value outside the error bound exists in a data block of mostly zeros, reducing both the compression ratio and throughput.

To overcome these limitations, we propose the threshold+grouping method. The threshold+grouping method extends the threshold method, first applying the filter in Eq. 3 and then storing nonzero data points into a new array, called the *significant value* array. A bitmap is introduced to indicate whether a data point is a significant value or a zero value. The significant value array is compressed (and later decompressed) using a lossy compressor. For post-processing, the decompressed significant value array returns significant values to their original location in the tensor and assigns a zero value to all other points.

The threshold+grouping method is illustrated in Fig. 6, with the two resultant arrays noted at the bottom of the figure. As with $F(T)$ from the threshold method, the significant value array is compressed using the SZ or SZx framework. The threshold+grouping is carried out as follows: First, the threshold filter $F(x)$ is applied to each data point $x \in T$ while simultaneously storing in a bitmap b if the value is zero or nonzero. $b[i]$ is assigned 0 if data point x_i with index i was filtered to zero and 1 if x_i is nonzero, or significant. The resultant tensor $F(T)$ is fed through a parallel stream compaction stage where significant values are moved to the significant value array. Parallel stream compaction is the task of reducing an input array to an output array containing only values that satisfy some condition. In this instance, compaction moves significant values to the output array from the input array $F(T)$. The bitmap b for a tensor with dimension d and number of nonzero values NNZ can occupy $\lceil \frac{2^d}{8} \rceil$ bytes compared to $NNZ \times 2^d \times 4$ bytes for a Compressed Sparse Row (CSR) format, assuming a 32-bit index. CSR would have a smaller footprint compared to the bitmap if 3.125% of values are nonzero. However, for a R2R threshold of 1%, 60-90% of tensor values are zero, thus we opt for a bitmap to store nonzero value location.

Unlike the threshold method, threshold+grouping requires a post-processing stage to reconstruct the output array from the compressed significant value array and bitmap. After de-compressing the significant value array using cuSZ or cuSZx, the bitmap must be decompressed since it is used to reassign significant values and zeros to their original position in the tensor. A parallel prefix scan is performed so that in the next stage, threads can appropriately index the bitmap and output array to assign values. Lastly, the re-ordering of decompressed values and zeros to the output tensor is conducted in parallel.

D. Selecting a Threshold Value

The threshold parameter, t in Eq. 3, is a key component of the proposed compression framework, and it balances limiting data distortion and boosting compressibility in conjunction with the compressor error bound. The significant values in a tensor must remain within a reasonable bound to determine an energy result from contraction that can still provide meaningful insight into the quantum circuit simulation. Increasing the threshold value leads to more zero values and, thus, higher compression ratios, while decreasing the threshold value reduces the error in the final energy result. A lower threshold value also increases the fidelity of the tensor, ensuring that a decompressed tensor remains similar to its original form. Fidelity is defined as follows [25]:

$$F(or, de) = |\langle \psi_{or} | \psi_{de} \rangle| \tag{4}$$

Eq. 4 shows that the inner product of two quantum state vectors can be used to quantify their similarity. ψ_{or} is the state vector of the original tensor, and ψ_{de} is the state vector of the compressed and then decompressed tensor. $F(or, de)$ can be in the range of $[0, 1]$ where values closer to 1 indicate greater similarity between states.

In the experimental evaluation, we explore the impact on energy value and fidelity of varying t as well as the error bound ϵ.

VI. Platform-Specific Optimization

Here, we discuss GPU-specific implementation and optimization strategies for threshold+grouping. Recall that the threshold+grouping method has three main phases: 1) apply the threshold to the input tensor 2) write the bitmap depending on if a value is zero or nonzero 3) use parallel stream compaction to move nonzero values. For phase 2, each thread handles 32 contiguous elements, writing four bytes of the bitmap. Shared memory is used as a buffer to enable coalesced memory accesses while simultaneously avoiding atomics.

A. Parallel Stream Compaction

We recall that the threshold+grouping method requires parallel stream compaction to populate the significant value array. We tested three existing compaction libraries: thrust [26], cuSPARSE [27], and CUB [28] available through the CUDA 11 Toolkit. CUB performed best with the highest throughput; thus, we integrated CUB into our final threshold+grouping implementation. After compaction, the threshold+grouping method performs lossless compression on the bitmap to reduce the metadata footprint.

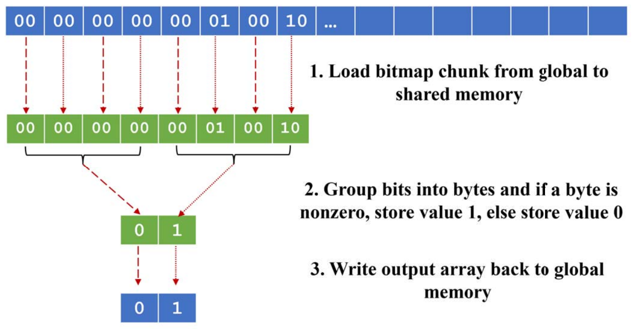

00 00 00 00 00 01 00 10 ...
1. Load bitmap chunk from global to shared memory
00 00 00 00 00 01 00 10
2. Group bits into bytes and if a byte is nonzero, store value 1, else store value 0
0 1
3. Write output array back to global memory
0 1

Fig. 7: Compressing the bitmap using a second-level bitmap. The first-level bitmap is sent through parallel stream compaction to group significant bytes.

B. Compressing the Bitmap

The bitmap required in the threshold+grouping method represents a storage overhead. Occupying $\lceil \frac{2^d}{8} \rceil$ bytes, the bitmap can limit the compression ratio achieved by the threshold+grouping method. One option is to compress the bitmap using existing GPU-accelerated compressors, such as LZ4 [29], a part of the NVCOMP library [30]. However, these lossless compressors, including LZ4, are designed to compress scientific data but yield sub-optimal performance when used to compress the bitmap (which stores *metadata*). In an effort to increase performance while still compressing the bitmap, we introduce a two-level bitmap compression kernel that leverages bitmap sparsity to achieve compression.

Fig. 7 illustrates the operation of our proposed two-level bitmap compression kernel. We note that, on tensor datasets, many bits of the raw bitmap are zero, resulting in many zero bytes. Using the same principle of the threshold+grouping method, we perform grouping on the bitmap's bytes to create a second bitmap level. Our implementation exploits memory coalescing and high shared memory bandwidth to reduce bitmap compression latency. Each thread handles 256 consecutive bits of the bitmap, writing four bytes to represent 32 groups of data points in the first level. Since each thread is known to handle 32 bytes of the first-level bitmap and four bytes of the second-level bitmap, no atomics are needed. The second-level bitmap directs a parallel stream compaction phase with CUB to group nonzero bytes into a new significant byte array.

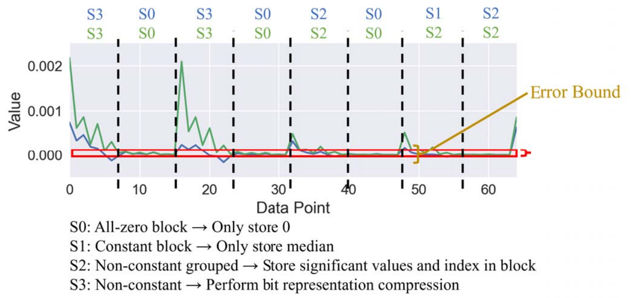


Fig. 8: High-level kernel design for integration of threshold+grouping with cuSZx. Each data block is mapped to a CUDA block. Blue and green states refer to real and imaginary components, respectively.

```
                         Integrated Kernel
   Compress<<<>>>(data, blk_size, threshold, error, out):
1     __shared__ float sig_values[blk_size]
2     __shared__ uint8_t indices[blk_size]
3     __shared__ int sig_ind = 0
4     int state;
5     float value = data[tid+bid*blksize]
6     if abs(value) > threshold:
7        sig_values[sig_ind] = atomicAdd(sig_ind, 1)
8        indices[sig_ind] = tid
9     // Determine if all points are within error range
10    __syncthreads()
11    if sig_ind == 0:              state = 0  // All zeros
12    else if isConstant:           state = 1  // Constant
13    else if sig_ind < blk_size/2: state = 2  // Non-constant group
14    else:                         state = 3  // Non-constant
15    __syncthreads()
16    if state == 2:
17       // Rest of grouping; Write sig_values,sig_ind,indices to out
18    if state == 3:
19       // Bit Representation Compression
                            Host Call
1  numBlks = data_size/blk_size
2  numThds= blk_size
3  Compress<<<numBlks,numThds>>>(data,blk_size,threshold,error,out)
```

Fig. 9: Pseudocode for threshold+grouping with cuSZx

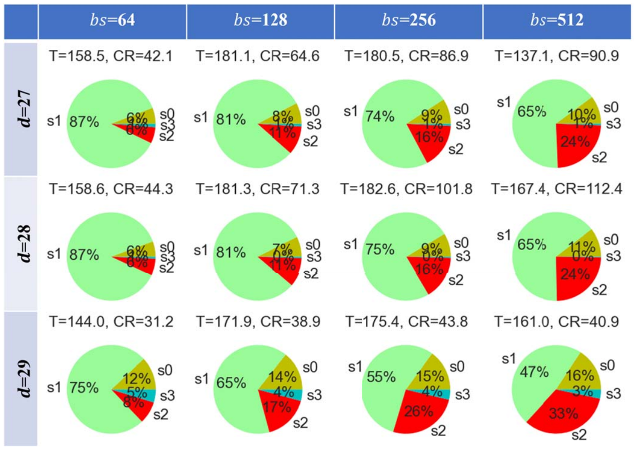


Fig. 10: Breakdown of a proportion of block states when varying tensor with dimension d and data block size bs. T indicates the compression throughput in GB/s, and CR indicates the compression ratio. States are as follows: $s0$=All-zero, $s1$=Constant, $s2$=Non-constant grouped, $s3$=Non-constant

C. Kernel Fusion Optimization for cuSZx

cuSZx categorizes blocks of data points as either constant or non-constant data blocks and maps data blocks to CUDA blocks. This framework naturally lends itself to integration with threshold+grouping since CUDA blocks can process grouping on a local level. Threshold+grouping as a separate pre-processing step can introduce additional kernel launch overhead and redundant memory accesses, limiting compression throughput. With integration, only a single kernel launch is needed for compression, and the tensor is loaded into GPU's global and shared memory only one time. When integrating threshold+grouping with cuSZx, we introduce two additional block states in addition to *constant* and *non-constant* data blocks: *all-zero* and *non-constant grouped*.

Fig. 8 shows how states are assigned to each data block and Fig. 9 shows the corresponding pseudocode for the integrated kernel with `state` representing the block state. The all-zero data block is a constant block where all data points are zero. The non-constant grouped block indicates a block with less than half of all values being nonzero and not bounded by the error bound. Each thread loads one value from the input tensor for processing (line 5). Since there are few nonzero values and the block size, `blk_size`, is small relative to the overall data size, `data_size`,(typically 64 to 256 data points per data block), nonzero values are moved to a new array, `sig_values`, and their index within the data block is stored in array `indices` (lines 6-8). At most, there are `blk_size` atomic operations. The index of a data point with block size b is $\log_2 b$ bits, and a local bitmap would be of size b bits. For s significant values, storing indices requires less space than a local bitmap when $s < \frac{b}{\log_2 b}$. After integration, the compressed tensor data is composed of 1. the median values of each block (written directly to `out`) 2. the state of each block 3. data point values (`sig_values` written to global memory `out`) and local indices (`indices` written to global memory `out`) of non-constant grouped blocks, and 4. bit representation compression (line 19) results for non-constant blocks (written directly to `out`). Integration requires modifying the existing cuSZx kernel to perform nonzero data point movement and index storage for CUDA blocks handling a non-constant grouped block, an operation that is relatively low in performance cost. Shared memory can be exploited within the kernel to enable coalesced reads and writes for non-constant grouped blocks.

Selecting a data block size can have a direct impact on performance since, in this integration, one thread is mapped to one data point within a block. Fig. 10 shows the effect of varying block size on tensor compression metrics. Larger data blocks involve more threads in the grouping and bit representation compression processes. With larger data blocks, workload distribution across thread blocks is more even since there is a greater likelihood that a data block is not an all-zero or constant block. Thus, the burden of non-constant block compression is more spread out across the streaming multiprocessors of the GPU. However, if the block size goes beyond 256, we note a performance decrease. This is because blocks that were previously state 0 or state 1 block are now state 2, and instead of even workload distribution, more blocks must perform the more time-consuming state 2 grouping process. Additionally, a block size of 256 allows for byte-aligned accesses since indices within a block are one complete byte. Other block sizes must store indices that are greater than or less than 8 bits which can affect memory access performance. In an effort to balance high throughput with a reasonable compression ratio, our experimental evaluation focuses on the integrated design using a block size of 256. A similar integration is performed for decompression, where data blocks with state 2 are expanded using `indices` and `sig_values`.

D. Overall Framework

The implementation choices of threshold and grouping compose a framework for tensor compression that can adapt to the user's needs. Our framework lets the user find the best compression ratio and throughput balance to suit their application. cuSZ and cuSZx can be substituted for each other and coupled with either threshold, grouping with LZ4, or grouping with the two-level bitmap. Additionally, the integration of grouping with cuSZx provides a higher compression ratio alternative to cuSZx with comparable throughput. In all, there are nine options on the spectrum of choices, with a high compression ratio on one end and high throughput on the other. In the experimental evaluation, we will determine where each of these choices resides on this spectrum (Fig. 15).

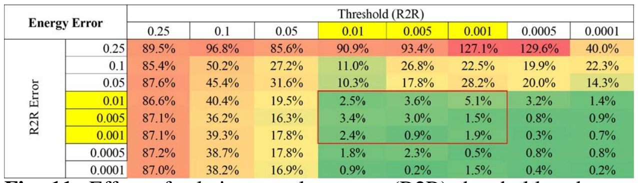
VII. Experimental Evaluation

We use the Quantum Approximate Optimization Algorithm (QAOA) for MaxCut [31] to generate circuits for QTensor simulation. Tensors in QTensor have a dimensionality of d, corresponding to 2^d data points. Table I reports the maximum size of double-precision tensors, as reported in [32], and the compression ratio CR needed to store tensors of MaxCut QAOA on the Nvidia A100 GPU 40 GB global memory. For the MaxCut input graph, the degree is fixed at 3, N is the number of vertices, p is the circuit depth, and the maximum tensor d is the average maximum tensor dimension of the corresponding N and p. Without compression, Table I indicates that we can only simulate MaxCut QAOA circuits with low N and p. If tensor multiplication were to take place in GPU memory, the required compression ratio would need to further increase in order to store multiple tensors in memory. While cuSZ and cuSZx may achieve sufficient compression ratio for low complexity circuits, such as for $N = 25, p = 5$, more complex circuits (e.g. $N = 34, p = 5$) require significantly higher compression ratios to compress tensors.

TABLE I: Circuit Parameter Effect on Tensor Size

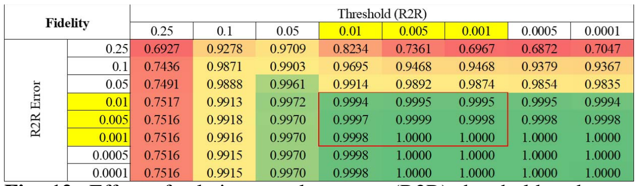
N	*p*	*Max Tensor d*	*Max Tensor Size (GB)*	*CR for one tensor*
20	4	24	0.25	0.00625
20	5	29	8	0.2
25	4	25	0.5	0.0125
25	5	36	1024	25.6
34	4	26	1	0.025
34	5	41	32768	819.2

To determine the effectiveness of our proposed methods as well as determine the relationship between error in tensors and the final energy result, we use tensors from the MaxCut QAOA circuit with nine varying combinations of $N \in [20, 35]$ and $p \in [3, 5]$. We quantify the "effectiveness" of each method using two metrics, the compression ratio, and the throughput, providing the end-user a guideline for the performance of the tested approaches. The quantum circuit enters the QTensor pipeline and we extract tensors after a varying q number of steps. Compression and decompression are performed on the tensor before resuming the QTensor contraction algorithm, completing the pipeline. Both threshold and threshold+grouping methods are implemented in CUDA 11 and are run on an NVIDIA A100 GPU, which features 64 FP32 cores per streaming multiprocessor (SM), 108 SMs, and a shared memory size of 192 KB per SM. Target tensors for compression vary in size from 2^{26} to 2^{29} values, or 512 MB to 4 GB, and are generated from QTensor. We collect the compression/decompression throughput and a compression ratio of both threshold and threshold+grouping methods when using cuSZ and cuSZx as base compressors.



Energy Error		Threshold (R2R)							
		0.25	0.1	0.05	0.01	0.005	0.001	0.0005	0.0001
R2R Error	0.25	89.5%	96.8%	85.6%	90.9%	93.4%	127.1%	129.6%	40.0%
	0.1	85.4%	50.2%	27.2%	11.0%	26.8%	22.5%	19.9%	22.3%
	0.05	87.6%	45.4%	31.6%	10.3%	17.8%	28.2%	20.0%	14.3%
	0.01	86.6%	40.4%	19.5%	2.5%	3.6%	5.1%	3.2%	1.4%
	0.005	87.1%	36.2%	16.3%	3.4%	3.0%	1.5%	0.8%	0.9%
	0.001	87.1%	39.3%	17.8%	2.4%	0.9%	1.9%	0.3%	0.7%
	0.0005	87.2%	38.7%	17.8%	1.8%	2.3%	0.5%	0.8%	0.8%
	0.0001	87.0%	38.2%	16.9%	0.9%	0.2%	1.5%	0.4%	0.2%

Fig. 11: Effect of relative-to-value-range (R2R) threshold and error on final energy result. Values are energy result error and it is averaged by applying distortion over all tensors of nine different circuits.



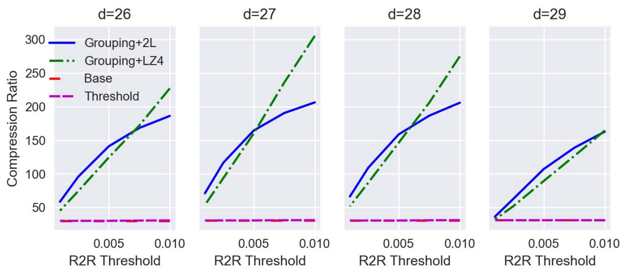
Fidelity		Threshold (R2R)							
		0.25	0.1	0.05	0.01	0.005	0.001	0.0005	0.0001
R2R Error	0.25	0.6927	0.9278	0.9709	0.8234	0.7361	0.6967	0.6872	0.7047
	0.1	0.7436	0.9871	0.9903	0.9695	0.9468	0.9468	0.9379	0.9367
	0.05	0.7491	0.9888	0.9961	0.9914	0.9892	0.9874	0.9854	0.9835
	0.01	0.7517	0.9913	0.9972	0.9994	0.9995	0.9995	0.9995	0.9994
	0.005	0.7516	0.9918	0.9970	0.9997	0.9999	0.9998	0.9998	0.9998
	0.001	0.7516	0.9916	0.9970	0.9998	1.0000	1.0000	1.0000	1.0000
	0.0005	0.7516	0.9915	0.9970	0.9998	1.0000	1.0000	1.0000	1.0000
	0.0001	0.7516	0.9915	0.9970	0.9998	1.0000	1.0000	1.0000	1.0000

Fig. 12: Effect of relative-to-value-range (R2R) threshold and error on tensor fidelity. Values are averaged by applying distortion to ten tensors of varying size and composition.

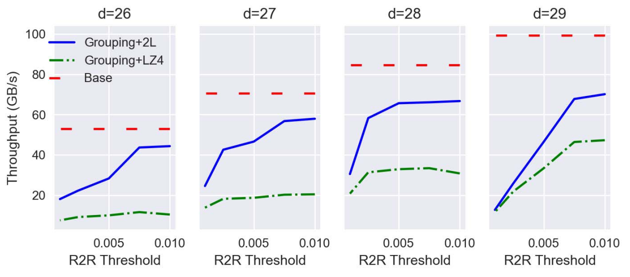
A. *Effect of Error Bound and Threshold on Energy Value*

Our distortion analysis seeks to quantify the effect of lossy compression on both the entire simulation and on a single tensor from a simulation, and the results for each are reported in Fig. 11 and Fig. 12, respectively. For both figures, green regions indicate less error while red regions indicate high error, and the red bounding box indicates the targeted region of values for the evaluation. Both the error bound and threshold are implemented as relative-to-value-range values: t=$r_t(\max T - \min T)$ and ϵ=$r_\epsilon(\max T - \min T)$ for tensor T. The inputs are thus r_t and r_ϵ for the compression pipeline. In Fig. 11, three MaxCut QAOA and six random quantum circuits with relative-to-value-range error and threshold applied to all tensors are used to generate energy values. They are compared against the energy value using the original circuit and the resulting percent error is reported. Since we aim to limit energy error to 5%, threshold and error are selected as $r_t \in [0.001, 0.01]$ and $r_\epsilon \in [0.001, 0.01]$ for compression performance analysis. Increasing r_t generally increases the compression ratio, thus the range $[0.001, 0.01]$ strikes a balance between high compression ratio and low result error. This range of r_t and r_ϵ is also suitable for achieving high fidelity decompressed tensors as seen in Fig. 12. Fidelity values are rounded to four decimal places and are collected from ten varying tensors from MaxCut QAOA and random circuits with r_t and r_ϵ applied using Eq. 4. Fidelity closer to 1 is beneficial for high accuracy circuit simulations [33] [34].

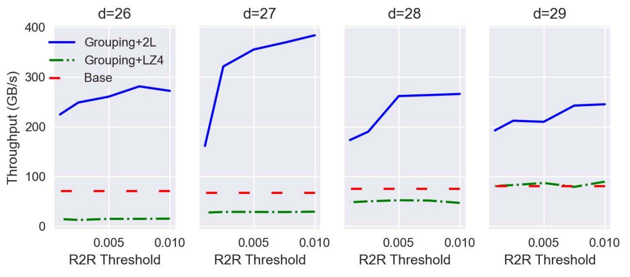
We found that there were no significant compression ratio or performance differences across all the methods when varying r_ϵ thus the evaluation focuses on varying r_t with $r_\epsilon = 0.005$.



(a) Compression Ratio



(b) Compression Throughput



(c) Decompression Throughput

Fig. 13: cuSZ-based results. The threshold only is omitted for throughput plots due to no compression ratio benefit.

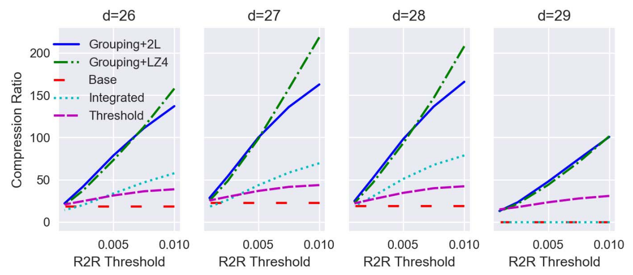
B. *Pre-processing and Post-processing with cuSZ*

Fig. 13a plots the compression ratio of cuSZ coupled with various strategies. “Grouping+LZ4” refers to using the grouping method with LZ4 as a compressor for the bitmap. ”Grouping+2L” refers to using the two-level bitmap design for bitmap compression. The “Base” compressor uses only cuSZ with no pre-processing, and “Threshold” applies only to the threshold pre-processing stage. The dimensionality d of each tensor is listed above each corresponding plot and the size of the tensor in bytes can be easily calculated as $2^d * 16$ where 16 is the number of bytes to store a double-precision floating-point complex number. Input tensors are cast to single-precision floating-point complex numbers for compatibility with cuSZ and cuSZx, thus we report compression ratio and throughput relative to single-precision tensors. LZ4 compression using the NVCOMP library performs the best for higher thresholds, scaling linearly with the value of the relative-to-value-range (R2R) threshold. Since LZ4 is a more refined general-purpose lossless compressor compared to the two-level bitmap compressor, it can scale well with an increasing number of zero values generated from applying the threshold. The two-level bitmap compression, however, can match or outperform LZ4’s compression ratio for lower R2R thresholds. Since lower thresholds lead to more significant values, bytes representing the bitmap value of eight adjacent data points can become more dissimilar from each other. This, in turn, can hurt LZ4’s compression performance while the two-level bitmap treats all nonzero bytes equally. Beyond implementation specifics, threshold+grouping generates much higher compression ratios compared to either the threshold-only or baseline compressor approaches. Grouping itself reduces the footprint of a tensor and coupled with cuSZ, significant values are further compressed leading to increases in compression ratio from ∼30 for baseline to over 300 for NVCOMP-based grouping.

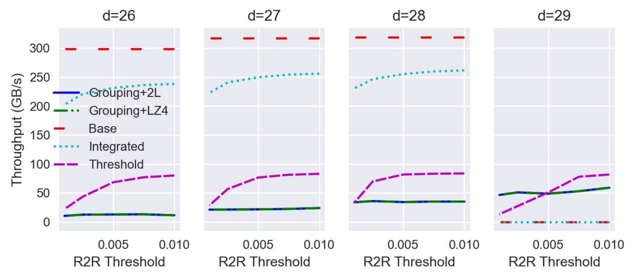
Fig. 13b plots the compression throughput of cuSZ using the same methods as Fig. 13a, omitting the threshold-only approach since there is no performance or compression ratio benefit across the varying r_t. Fig. 13c plots the decompression throughput of cuSZ coupled with other methods. Tensors are compressed and then decompressed, with the appropriate pre-processing and post-processing stages inserted before and after cuSZ compression/decompression. Plot conventions are the same as Fig. 13a. The lowest throughput is achieved by the grouping method with the NVCOMP backend. We recall that NVCOMP compressors are tailored to scientific data, not metadata. The threshold method introduces a small overhead for applying the threshold and does not lead to a significant increase in compression ratio over baseline cuSZ. Performing the best relative to baseline cuSZ, the grouping method with the two-level bitmap can achieve higher decompression throughput than cuSZ alone. This is for two main reasons: 1. low pre-processing and post-processing overhead for GPU and 2. smaller data input for cuSZ. Unlike LZ4, the two-level bitmap compressor performs only relatively fast GPU operations such as stream compaction and bit comparison. As such, the performance overhead of the two-level bitmap is low. Since cuSZ is only run on the significant values when using grouping, cuSZ completes decompression faster and performs closest to baseline for compression. On the A100 GPU, the throughput of the two-level bitmap with cuSZ approach can reach nearly 70 GB/s for compression and as high as 400 GB/s for decompression. We explore even faster approaches next with cuSZx.

C. *Pre-processing and Post-processing with cuSZx*

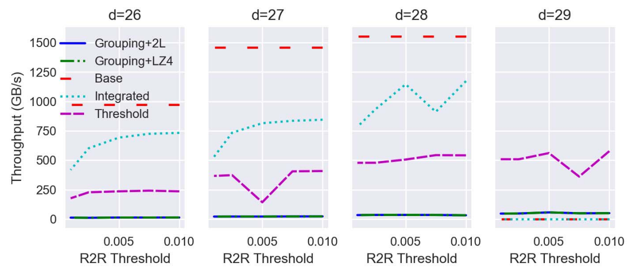
Fig. 14a reports the compression ratio of five different cuSZx-based approaches. “Grouping” refers to the threshold+grouping method, with “+2L” using the two-level bitmap and “+LZ4” using LZ4 to compress the bitmap. “Integrate” refers to the integrated approach described in Section VI-C. “Base” and “Threshold” are the same as Fig. 13a with cuSZx instead of cuSZ. Grouping methods generate the highest compression ratios of all the methods, reaching as high as ∼220 compared to ∼20 baseline for tensor with $d = 27$. This is in line cuSZ’s results since higher R2R thresholds yield more zero values, thus decreasing the number of significant



(a) Compression Ratio



(b) Compression Throughput



(c) Decompression Throughput

Fig. 14: cuSZx-based results. Note that tensor $d = 29$ did not run for baseline and integrated due to memory requirements. Both Grouping lines closely overlap for throughput plots.

values sent through cuSZx. The integrated approach can also boost the compression ratio over baseline, though with less significant gains compared to grouping. The plots additionally indicate that the threshold-only method can also increase the compression ratio since more constant blocks are generated. The threshold and grouping methods, when not integrated, also allow cuSZx to handle larger tensors without exhausting GPU memory since there are fewer values requiring metadata. Baseline cuSZx and the integrated method require many intermediate data structures for large tensors, such as for $d = 29$, and thus exhaust GPU memory.

Fig. 14b plots the compression throughput of the approaches used in Fig. 14a with varying R2R threshold. Fig. 14c plots the decompression throughput. Baseline cuSZx has a compression throughput of 300 to 320 GB/s and a decompression throughput of 1000 to 1600 GB/s. The best-performing approach relative to baseline cuSZx is the integrated approach with a compression throughput of 200 to 260 GB/s and a decompression throughput of 420 to 1200 GB/s. The integrated approach is implemented with the fewest kernel launches, reducing kernel launch overhead, and performs grouping only at a data block level which leads to few atomic operations. Additionally, the time-intensive bit representation compression for non-constant blocks can be offloaded into local grouping if the block is sufficiently sparse. The threshold-only method introduces an additional kernel launch which involves reading all data points and writing all threshold-applied points back to the global array. This process can lead to lower performance relative to the integrated approach, which applies the threshold within the cuSZx kernel. Still, grouping with LZ4 has the lowest performance due to bitmap compression overhead.

D. *Compression Ratio versus Throughput*

The proposed pipelines involve a trade-off between compression ratio and throughput, as illustrated in Fig. 15. We omit the threshold-only approaches since there is no significant compression ratio or performance benefit relative to other approaches. On one end, cuSZ coupled with the threshold+grouping method can yield very high compression ratios with lower throughput than cuSZx. cuSZx integrated with threshold+grouping is limited in its ability to boost tensor compression ratio over baseline but can perform close to the speed of baseline cuSZx. These two designs provide users with the flexibility to choose their priority, utilizing either the high throughput of the integrated cuSZx pipeline or the high compression ratio of the threshold+grouping cuSZ pipeline, with additional options between them. When compression is needed as a post-processing step to store tensors to disk, cuSZ with threshold+grouping can provide the lowest compressed data footprint. If intermediate tensors need to be compressed during simulation, compression can be overlapped with contraction if the tensors being compressed are not contracted. In this case, a higher throughput solution, such as cuSZx integrated with threshold+grouping, can prove to be more valuable.

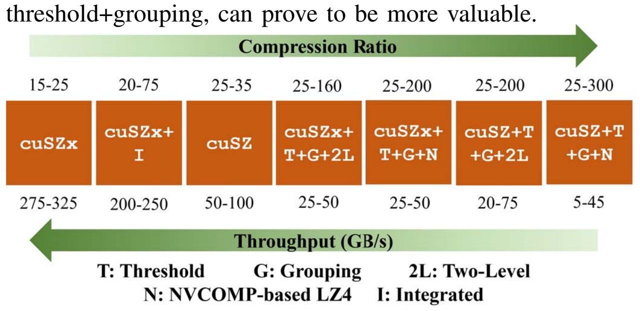


Fig. 15: Spectrum of Throughput versus Compression Ratio with specific implementation placed according to relative performance

VIII. Conclusion and Future Work

We design a lossy compression framework for the compression of quantum simulation tensors on GPU. The key insights based on our experiments with NVIDIA's A100 GPU include: (1) The nine compression pipelines (including our proposed designs) vary in their abilities. (2) Grouping method can increase the compression ratio of cuSZ from ∼30 to ∼300. (3) The grouping method increases cuSZx's compression ratio from ∼20 to ∼200 while the integrated kernel implementation can boost the compression ratio to 150+ with up to 250 GB/s for compression and 1200 GB/s for decompression, close to cuSZx's throughput. (4) The contraction error is limited to $\leq 5\%$ and ensures that tensor fidelity remains high after decompression. With our current design, multi-GPU compression can be achieved by chunking the tensor and compressing each chunk on a different GPU. In the future, we will extend this framework to a more tailored multi-GPU environment as well as tailor it to other quantum computing simulators that produce different metrics.

Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant OAC-2003709 and OAC-2104023. We acknowledge the computing resources provided on ThetaGPU and JLSE (operated by Argonne Leadership Computing Facility). At NC State, the project was supported through subcontract DE-AC02-06CH11357 and National Science Foundation's award CNS-1812727.

References

[1] J. Preskill, "Quantum computing and the entanglement frontier," 2012. [Online]. Available: https://arxiv.org/abs/1203.5813
[2] A. W. Harrow and A. Montanaro, "Quantum computational supremacy," *Nature*, vol. 549, no. 7671, pp. 203–209, sep 2017.
[3] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM Journal on Computing*, vol. 38, no. 3, pp. 963–981, jan 2008.
[4] R. Schutski, D. Lykov, and I. Oseledets, "Adaptive algorithm for quantum circuit simulation," *Phys. Rev. A*, vol. 101, p. 042335, Apr 2020.
[5] D. Lykov, A. Chen, H. Chen, K. Keipert, Z. Zhang, T. Gibbs, and Y. Alexeev, "Performance evaluation and acceleration of the qtensor quantum circuit simulator on gpus," in *2021 IEEE/ACM 2nd QCS Workshop*, 2021, pp. 27–34.
[6] D. Lykov and Y. Alexeev, "Importance of diagonal gates in tensor network simulations," 2021. [Online]. Available: https://arxiv.org/abs/2106.15740
[7] D. Lykov and et al., "Tensor network quantum simulator with step-dependent parallelization," 2020.
[8] R. Dechter, "Bucket elimination: A unifying framework for reasoning," *Artificial Intelligence*, vol. 113, no. 1-2, pp. 41–85, Sep. 1999.
[9] Y. Collet, "Zstandard – real-time data compression algorithm," *http://facebook.github.io/zstd/*, 2015.
[10] Zlib, https://www.zlib.net/, online.
[11] BlosC compressor, http://blosc.org/, online.
[12] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan 2009.
[13] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1643–1654.
[14] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739.
[15] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.
[16] J. Tian and et al, "cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3–15.
[17] X. Yu and et al., "Ultrafast error-bounded lossy compression for scientific datasets," ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 159–171.
[18] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *IEEE Big Data*, 2018, pp. 438–447.
[19] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *IEEE IPDPS*, 2017, pp. 1129–1139.
[20] P. G. Lindstrom *et al.*, "Fpzip," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2017.
[21] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," *Computing and Visualization in Science*, vol. 19, no. 5, pp. 65–76, 2018.
[22] A. M. Gok, S. Di, Y. Alexeev, D. Tao, V. Mironov, X. Liang, and F. Cappello, "Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 1–11.
[23] Z. Shangnan, "Quantum data compression and quantum cross entropy," 2021. [Online]. Available: https://arxiv.org/abs/2106.13823
[24] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," ser. ACM SC '19, 2019.
[25] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th ed. USA: Cambridge University Press, 2011.
[26] Thrust, https://docs.nvidia.com/cuda/thrust/index.html, online.
[27] cuSPARSE, https://docs.nvidia.com/cuda/cusparse/index.html, online.
[28] CUB, https://docs.nvidia.com/cuda/cub/index.html, online.
[29] LZ4, https://lz4.github.io/lz4/, online.
[30] NVCOMP, https://developer.nvidia.com/nvcomp, online.
[31] E. Farhi and A. W. Harrow, "Quantum supremacy through the quantum approximate optimization algorithm," 2016. [Online]. Available: https://arxiv.org/abs/1602.07674
[32] A. Galda, X. Liu, D. Lykov, Y. Alexeev, and I. Safro, "Transferability of optimal qaoa parameters between random graphs," 2021. [Online]. Available: https://arxiv.org/abs/2106.07531
[33] I. L. Markov, A. Fatima, S. V. Isakov, and S. Boixo, "Quantum supremacy is both closer and farther than it appears," 2018. [Online]. Available: https://arxiv.org/abs/1807.10749
[34] B. Villalonga, D. Lyakh, S. Boixo, H. Neven, T. S. Humble, R. Biswas, E. G. Rieffel, A. Ho, and S. Mandrà , "Establishing the quantum supremacy frontier with a 281 pflop/s simulation," *Quantum Science and Technology*, vol. 5, no. 3, p. 034003, apr 2020.