Parallel Tensor Network Contraction for Efficient Quantum Circuit Simulation on Multicore CPUs and GPUs

Alfred M. Pastor[1][a], Maribel Castillo[2][b] and Jose M. Badia[2][c]

[1]*Dpt. of Computer Science, Universitat de València, Avinguda de la Universitat, s/n, 46100 Burjassot, Spain*

[2]*Dpt. of Computer Science and Engineering, Universitat Jaume I, Av. Vicent Sos Baynat, s/n, 12071 Castellón de la Plana, Spain*

Keywords: Quantum Circuit Simulation, Tensor Network Contraction, Parallel Computing, Multicore CPUs, GPUs.

Abstract: Quantum computing has the potential to transform fields such as cryptography, optimisation and materials science. However, the limited scalability and high error rates of current and near-term quantum hardware require efficient classical simulation of quantum circuits for validation and benchmarking. One of the most effective approaches to this problem is to represent quantum circuits as tensor networks, where simulation is equivalent to contracting the network. Given the computational cost of tensor network contraction, exploiting parallelism on modern high performance computing architectures is key to accelerating these simulations. In this work, we evaluate the performance of first-level parallelism in contracting individual tensor pairs during tensor network contraction on both multi-core CPUs and many-core GPUs. We compare the efficiency of three Julia packages, two optimised for CPU-based execution and the other for GPU acceleration. Our experiments, conducted with two parallel contraction strategies on highly entangled quantum circuits such as Quantum Fourier Transform (QFT) and Random Quantum Circuits (RQC), demonstrate the benefits of exploiting this level of parallelism on large circuits, in particular the superior performance gains achieved on GPUs.

1 INTRODUCTION

Quantum computing has the potential to solve problems that are infeasible for classical computers (Nielsen and Chuang, 2010). However, current quantum hardware remains limited by low qubit counts and high error rates, leading to considerable interest in simulating quantum circuits on classical systems. Such simulations provide critical insight into algorithm performance and offer a practical means of testing and benchmarking quantum algorithms.

This article explores advanced methods for simulating quantum circuits, with an emphasis on tensor networks and parallelism to improve scalability and efficiency. Tensor networks represent quantum states and operations as interconnected tensors, allowing significant compression of the state space and reducing the computational overhead of simulations. By exploiting the structural properties of quantum circuits, tensor networks can simulate larger and more complex systems (Markov and Shi, 2008).

Recent advances in tensor network simulations include optimisation techniques for GPUs, such as transforming Einstein summation operations into GEMM operations and using mixed precision to balance speed and accuracy. Improved algorithms for determining optimal tensor contraction paths have further reduced computational times, demonstrating significant gains in performance and accuracy (Gray and Kourtis, 2021). Parallelism complements these advances by distributing computational workloads across multiple processors, enabling the simulation of larger circuits than would be feasible on a single processor. By integrating tensor networks and parallelism, classical simulations can approach the practical limits of quantum circuit emulation.

Parallel simulation algorithms can be classified by the levels of parallelism they exploit and the types of parallel architectures they target. At a fine-grained level, parallelism is applied to pairwise tensor contractions, often reformulated as matrix multiplications for compatibility with highly optimised linear algebra libraries. A higher level of parallelism involves the simultaneous contraction of groups of tensors, de-

[a] https://orcid.org/0000-0002-7740-6354
[b] https://orcid.org/0000-0002-2826-3086
[c] https://orcid.org/0000-0002-5927-0449

Pastor, A. M., Castillo, M. and Badia, J. M.
Parallel Tensor Network Contraction for Efficient Quantum Circuit Simulation on Multicore CPUs and GPUs.
DOI: 10.5220/0013551400004525
In *Proceedings of the 1st International Conference on Quantum Software (IQSOFT 2025)*, pages 120-127
ISBN: 978-989-758-761-0

termined by techniques such as community detection in tensor network graphs or hypergraph partitioning. Slicing, which splits indices to create subcircuits, further increases parallelism by dividing the original tensor network into smaller, independently contractible components (Huang et al., 2021). Finally, multiple output state amplitudes can be computed in parallel by contracting the same tensor network multiple times while keeping both the input and output indices closed. This is particularly useful for tasks such as random circuit sampling, which was used by Google in 2019 in an attempt to demonstrate quantum supremacy (Arute et al., 2019).

The choice of parallel architecture also plays a key role in simulation performance. Distributed memory systems are ideal for large-scale simulations due to their large memory capacity, enabling the simulation of high-qubit circuits where tensor sizes grow exponentially. Multicore processors, commonly found in modern servers, facilitate shared memory communication and support multiple levels of parallelism. GPUs excel at tensor contraction tasks due to their massive data parallelism capabilities, although their limited memory can limit the size of tensors they can handle. Hybrid approaches that combine these architectures, such as using GPUs for matrix multiplication within a distributed memory framework, can mitigate individual limitations and improve overall performance.

In this study, we evaluate the first level of parallelism using multicore processors and GPUs within two different algorithms: one that applies this parallelism throughout the contraction process, and another that combines both levels of parallelism in a multi-stage algorithm, applying the first level in the final stage. We have evaluated and compared the parallel performance of three Julia packages, two that perform pairwise tensor contractions on multicore CPUs, and one that performs them on the GPU. We used two types of complex and highly entangled circuits as test beds: Quantum Fourier Transform (QFT) and Random Quantum Circuits (RQC). This work is carried out in the QXTools environment, a Julia-based framework for simulating quantum circuits via tensor networks (Brennan et al., 2022).

The main contributions of this work can be summarised as follows:

- We evaluate the impact of first-level parallelism in tensor network contraction for quantum circuit simulation on high-performance architectures.
- We compare the efficiency of two CPU-based and one GPU-based Julia packages and analyse their performance on different quantum circuit types.
- We evaluate the influence of circuit structure and contraction strategy on parallel performance, distinguishing between full-network and community-based tensor contraction.
- Our experimental results provide a quantitative analysis of the scalability of multicore CPUs and GPUs to contracts tensor networks in parallel.

The paper is structured as follows: Section 2 reviews related work, while Section 3 provides background on tensor networks. Section 4 outlines tensor contraction strategies, and Section 5 details the evaluation methodology. Section 6 presents experimental results, followed by conclusions in Section 7.

2 RELATED WORK

A variety of libraries, frameworks and simulators have been developed for tensor network contraction to address the high computational and memory cost of this process (Quantiki, 2023). Many of these tools implement different levels of parallelism to improve the efficiency of contraction and to enable the simulation of larger quantum circuits.

One of the most notable frameworks for tensor network-based quantum circuit simulation on CPU is `qFlex` (Villalonga et al., 2019). This framework, as many simulators, uses slicing techniques to partition tensor networks across nodes in a cluster, while using a multithreaded BLAS library to perform pairwise tensor contractions on each node's CPU. `TAL-SH` was designed for GPU-based simulations and later evolved into the `qExaTN` simulator (Lyakh et al., 2022), which integrates the first and second levels of parallelism.

Other approaches optimise tensor network contraction using task-based execution models. The `Jet` simulator combines slicing techniques and asynchronous tasks to minimise redundant computation and improve execution time on shared memory multiprocessors and GPUs (Vincent et al., 2022). The `AC-QDP` simulator[1] has been applied to RQC circuits, employing tensor network contraction optimisations that have also been incorporated into the widely used `CoTenGra` library[2]. Some of the latest implementations exploit GPU-based parallelism by optimising tensor index reordering, adjusting data precision, and using the cuTensor library to improve contraction efficiency (Pan et al., 2024).

NVIDIA's cuQuantum SDK provides tools for exact and approximate tensor contraction on GPUs (Bayraktar et al., 2023). The framework

[1] https://github.com/alibaba/acqdp

[2] https://github.com/jcmgray/cotengra

provides the `cuTensorNet` library, which computes contraction plans on the CPU and implements pairwise tensor contractions using single or multi-GPU configurations. These methods combine the first and second levels of parallelism using both intra-node and inter-node optimisations.

3 BACKGROUND

Tensors naturally generalise the concepts of vectors and matrices to higher dimensions. A rank-r tensor is an element of the space $\mathbb{C}^{d_1 \times \cdots \times d_r}$, where $d_1, \ldots, d_r$ denote its dimensions. For example, a vector with d complex components belongs to the space $\mathbb{C}^d$ and is classified as a rank-1 tensor ($v \in \mathbb{C}^d$), while a matrix with dimensions $n \times m$ resides in the space $\mathbb{C}^{n \times m}$ and is classified as a rank-2 tensor ($M \in \mathbb{C}^{n \times m}$).

Quantum circuits can naturally be represented as tensor networks, where the gates are represented as tensors and the indices correspond to the qubits connecting these gates. In this representation, all indices have a dimension of 2, reflecting the two possible base states of a qubit, $|0\rangle$ and $|1\rangle$, which form the computational basis. The basic operation used with tensor matrices is contraction, which corresponds to a tensor product of two tensors followed by a trace over the indices they share.

The use of tensor networks for the simulation of quantum circuits was introduced in (Markov and Shi, 2008). The approach involves contracting tensors in pairs until only a single tensor remains. However, finding the optimal tensor contraction order, which significantly affects the time and space cost, is NP-hard, leading to the development of various heuristics (Gray and Kourtis, 2021).

Partitioning the tensor network into smaller subnetworks is a widely used strategy to reduce the complexity of the contraction process and enable parallel execution. A particularly promising approach to achieve this is community detection, a method from graph theory that identifies highly connected regions within a network. In our previous work, we proposed a multi-stage parallel algorithm that uses community detection to partition tensor networks (Pastor et al., 2025). We employed the Girvan-Newman (GN) algorithm that uses edge betweenness centrality to iteratively remove high-centrality edges, isolating communities within the network (Girvan and Newman, 2002). In tensor network contraction, it minimizes inter-community connections, reducing tensor ranks in later stages.

4 STRATEGIES FOR TENSOR CONTRACTION

Due to its high computational and memory requirements, several strategies have been developed to optimise the execution of tensor contractions on modern hardware, especially on multicore CPUs and GPUs. The three main approaches to tensor contraction are: (1) direct execution as matrix multiplication, know as GEMM (GEneral Matrix-Matrix), (2) explicit memory reordering to enable GEMM execution, known as TTGT (Transpose-Transpose-GEMM-Transpose), and (3) memory access optimisations that avoid explicit transpositions. This section discusses these strategies, highlighting their advantages, limitations, and implementations in widely used libraries.

The first approach to tensor contraction is direct matrix multiplication using optimized GEMM routines like multithreaded BLAS (CPUs) or cuBLAS (GPUs) (NVIDIA, 2023). This method maximizes efficiency but is only feasible when tensor indices can be grouped without transposition, which is uncommon in general tensor network contractions for quantum circuits.

When a tensor contraction does not map directly to a matrix multiplication, a common alternative is to explicitly reorder the tensor indices in memory to transform the contraction into a GEMM operation. This strategy is known as the TTGT approach, and is widely used in GPU-based tensor contraction libraries such as `cuTensor` (NVIDIA, 2024), which performs optimised transpositions using CUDA kernels before performing the GEMM operation.

Other advanced techniques have been developed to optimise tensor contractions while avoiding data reordering in memory. These include the GEMM-like Tensor-Tensor Contraction (GETT) strategy or the Block-Scatter-Matrix Tensor Contraction (BSMTC). The GETT approach improves memory access efficiency by using cache-aware partitioning and hierarchical tensor contraction loops, reducing the need for explicit reordering (Springer and Bientinesi, 2018). BSMTC uses block-scatter vector layouts to dynamically compute memory addresses, allowing data to be accessed in its natural order without transposition. It was introduced in the TBLIS library (Matthews, 2018).

The first level of parallelism can be applied in the three strategies just described. Optimised versions of BLAS such as Intel MKL or OpenBLAS can be used in multi-core CPUs and cuBLAS in GPUs to perform parallel matrix multiplications. Data transpositions can be performed in parallel using specialised libraries. Finally, optimised multithreading

can also be applied in the BSMTC method by parallelizing multiple loops around a microkernel, optimising memory access and minimising synchronisation overhead (Matthews, 2018).

5 TENSOR NETWORK CONTRACTION

5.1 QXTools Tensor Network Contraction Framework

Our work involves the development and evaluation of parallel algorithms within the QXTools simulation framework (Brennan et al., 2021), a core component of the QuantEx project. Implemented in Julia, QXTools uses several external packages to manage tensor operations, optimise contraction sequences, determine efficient slicing strategies, and run large-scale simulations on GPUs or high performance computing clusters [3]. The framework supports all three levels of parallelism described earlier.

For the first level of parallelism, the framework relies on QXContext, which relies on `CUDA.jl` to perform contractions on GPU, while CPU-based contractions are performed using the `OMEinsum` and `ITensors` packages [4].

With respect to the second level of parallelism, QXTools enables slicing to divide the tensor network into multiple smaller subtensor networks, facilitating the contraction of larger networks. In addition, QXTools supports the use of MPI processes to perform parallel contractions of the subtensor networks generated by slicing, both in shared memory multiprocessor servers and in distributed memory clusters.

Regarding the third level of parallelism, QXTools allows the distribution of the output state amplitude sampling across multiple MPI processes running in parallel.

We used QXTools to implement and evaluate a novel multi-stage parallel algorithm for tensor network contraction and compared it with other parallelization strategies (Pastor et al., 2025). This algorithm integrates the first two levels of parallelism. Specifically, it uses the Girvan-Newman community detection algorithm to partition the tensor network. Then, multiple threads are used to contract the tensor network associated with each detected community in parallel. Finally, the first level of parallelism is used to contract each pair of tensors in the resulting network on a GPU.

[3]https://juliaqx.github.io/QXTools.jl

[4]https://github.com/under-Peter/OMEinsum.jl

5.2 Parallel Tensor Network Contraction Algorithms

In this paper we compare the performance of three Julia packages that exploit the first level of parallelism described in the introduction to contract tensor networks. Two of them run on multi-core CPUs, while the third runs on GPUs. The tools used are `OMEinsum (OME)`: a Julia package used in QXTools to perform pairwise tensor contractions within the network on a CPU; `BliContractor (BLI)`: a Julia package that implements a wrapper for the TBLIS library to contract tensor pairs on a CPU [5]; and `CUDA.jl`: a Julia package used by QXTools to perform pairwise tensor contractions on NVIDIA GPUs.

Our experiments analyse the performance of these three packages using two different algorithms: `single-stage` to carry out the contractions of all the tensor pairs during the execution of a tensor network contraction plan, and `multi-stage` to contract the tensor pairs in the final stage of the parallel multi-stage method introduced in (Pastor et al., 2025).

The first level of parallelism behaves differently in the two algorithms. The first contracts hundreds or thousands of tensors with varying ranks, from small single-qubit tensors to high-rank ones, involving single or multiple indices. The second contracts fewer but high-rank tensors, often sharing many qubits. When both input and output are closed, the final contraction reduces to a scalar.

`OMEinsum.jl` performs tensor contractions using Einstein summation notation, providing an efficient and flexible approach to tensor operations. If a tensor contraction can be reformulated as a matrix multiplication, `OMEinsum.jl` internally calls BLAS routines (such as `BLAS.gemm!`), allowing multi-threaded execution and optimised performance.

The Julia package `BliContractor.jl` is based in the TBLIS library, which optimizes tensor contractions using the BSMTC technique and parallelized its execution on multi-core arquitectures.

To test the performance of `BliContractor` we modified the method executed by `QXTools` to perform pairwise tensor contractions. Specifically, we replaced the call to `EinCode` from `OMEInsum` with a call to `contract` from `BliContractor`.

Finally, `QXTools` uses `QXContexts` for tensor contraction on distributed machines, including GPUs. `QXContexts` relies on `CUDA.jl`, which provides a high-level interface to NVIDIA's CUDA ecosystem, enabling seamless execution of GPU kernels in Julia.

[5]https://github.com/xrq-phys/BliContractor.jl

6 EXPERIMENTAL RESULTS AND DISCUSSION

6.1 Experimental Environment

The experiments were conducted on a high performance computing server equipped with two AMD EPYC 7282 processors, each with 16 cores and running at a base frequency of 2.8GHz. The system is configured with a total of 256 GiB of DDR4 RAM and benefits from 64 MiB of L3 cache. For GPU acceleration, the server is equipped with an NVIDIA RTX A6000 graphics card based on the Ampere architecture (Compute Capability 8.6). This GPU has 10,752 CUDA cores and 336 Tensor cores and is manufactured using 8nm process technology. It also includes 48 GiB of GDDR6 memory.

6.2 Pairwise Tensor Contraction

This section presents a comparative analysis of two Julia packages, `OMEinsum` and `BliContractor`, for sequential and parallel contraction of a pair of tensors using a multi-core CPU. The experiments were performed by contracting two tensors of different ranks (10 and 22) that share a subset of their indices.

First, we evaluate the sequential performance of both packages when the two tensors share only the contracted index. We then analyse the impact on performance of varying this index.

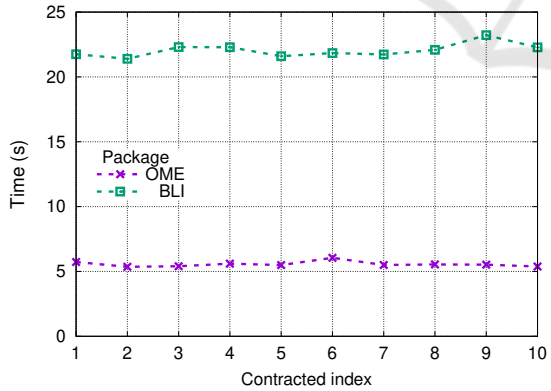


Figure 1: Comparison of the sequential contraction time of `OMEinsum` and `BliContractor` to contract one index. Tensors with ranks 10 and 22.

Figure 1 shows that `OMEinsum` consistently outperforms `BliContractor` in sequential execution, achieving approximately four times the speedup across all indices tested.

Next, we examine the parallel performance of both packages when contracting tensors over a single index.

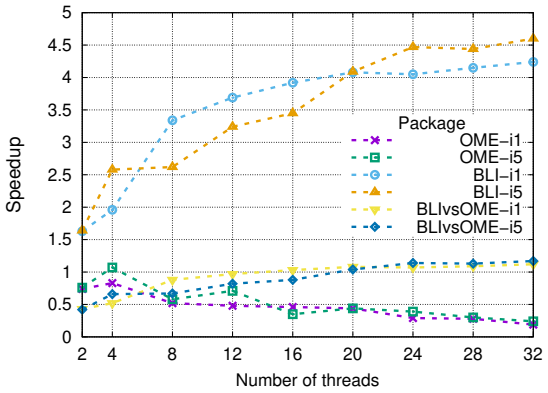


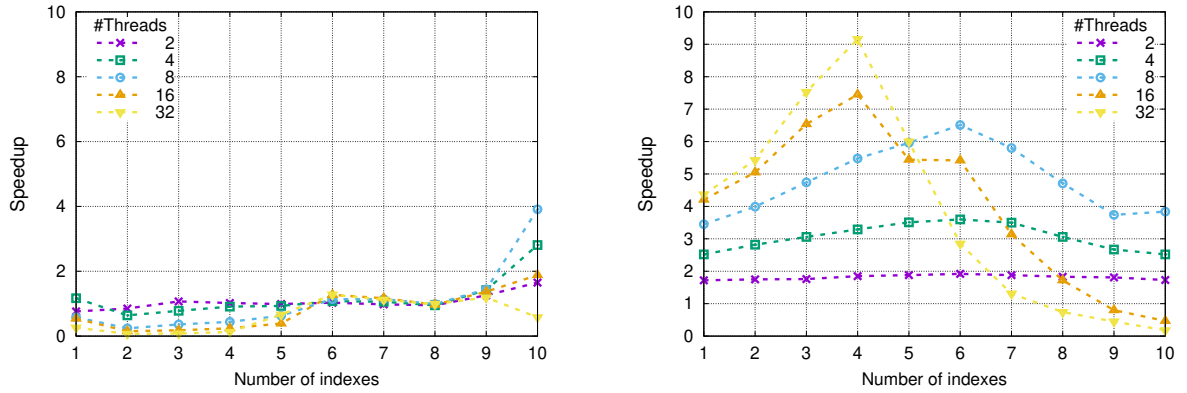
Figure 2: Comparison of the speedups of `OMEinsum` and `BliContractor` to contract indices 1 and 5. `BLIvsOME` lines show the speedup of `BLIContractor` with respect to the fastest sequential version (`OMEinsum`). Tensors with ranks 10 and 22.

Figure 2 highlights a fundamental difference between the two packages: `BliContractor` allows parallel contraction over a single index (1 and 5), whereas `OMEinsum` does not. The conclusion from these experiments is that `OMEinsum` was not designed to take advantage of the parallelism offered by modern multi-core processors, but rather to efficiently perform sequential pairwise tensor contractions. `OMEinsum` only uses multithreaded implementations of matrix multiplication in very few cases. Conversely, `BliContractor` uses the C-based TBLIS package, which is designed not only to minimise data movement in memory, but also to efficiently perform parallel pairwise tensor contractions.

Despite `BliContractor`'s ability to exploit parallelism, its sequential performance remains significantly lower than `OMEinsum`. As a result, when we calculate the speedups of `BliContractor` with respect to the fastest sequential algorithm, `OMEinsum`, we only get small speedups when using more than 20 threads.

Finally, we compare the parallel performance of both packages when the tensors share several indices and are contracted over all of them. The speedup evolution is analysed when contracting between 1 and the 10 indices of the lower-rank tensor. To eliminate potential bias due to index selection, the indices were chosen randomly and the reported results are the average of five different index sets.

The figures 3a and 3b show different behaviour of both packages. `OMEinsum` only achieves speedup when the 10 indices of the lower-rank tensor are contracted, and even then the acceleration is modest (below 4). Furthermore, using more than eight



(a) `OMEinsum` (b) `BliContractor`

Figure 3: Comparison of the speedups of `OMEinsum` and `BliContractor` to contract between 1 and 10 indices. Tensors with ranks 10 and 22.

threads reduces the observed speedup. In contrast, `BliContractor` shows a more regular speedup pattern. For two threads, it achieves a near-optimal speedup for any number of contracted indices. For 4 and 8 threads, the speedup increases progressively for up to 6 indices and then decreases. Using more than 8 threads reinforces this trend, with a rapid increase up to 4 contracted indices, followed by an even steeper decline beyond that point.

Additional experiments confirm that the highest parallel efficiency in both packages occurs when contracting all indices of two tensors of equal rank, and this efficiency tends to increase with rank. For example, contracting all indices of two tensors of rank 30 gives a speedup of 18.3 using 28 threads with `BliContractor` and 6.9 using 20 threads with `OMEinsum`. Both speedups are with respect to the sequential time using the same package.

The general conclusion from the experiments presented in this section is that `OMEinsum` is significantly more efficient for sequential pairwise tensor contractions. However, `BliContractor` takes much better advantage of the multi-core architecture of modern processors, which can speed up contractions for both single and multiple indices.

6.3 Parallel Tensor Network Contraction

This section evaluates the performance of the three packages introduced in section 5.2 for exploiting first-level parallelism in tensor network contraction on both multicore CPUs and manycore GPUs. To quantify the efficiency of the parallel algorithms, we use two well-established types of quantum circuits: RQCs and QFT. In both cases, contraction is performed by summing over all input and output indices of the circuit, effectively computing the probability amplitude of a given quantum state.

We first analyse the scalability of `BliContractor` in parallel execution as the contracted circuit size increases. To this end, we evaluate QFT circuits of different qubit sizes using the single-stage contraction algorithm, where `BliContractor` is used to contract all tensor pairs defined by a contraction plan derived from the Girvan-Newman algorithm. Figure 4 illustrates the speedup achieved by the single-stage method when using 2 to 32 threads for QFT circuits of various sizes. The speedup is calculated relative to the sequential execution time obtained with the same package.

The results in Figure 4 show that the benefits of parallelism become more pronounced as the circuit size increases, especially as the number of qubits increases. This trend occurs because the computational cost and tensor dimensions scale significantly with the number of qubits, resulting in a higher workload that allows more efficient thread utilisation in the contraction process.

For the smallest circuit (QFT25), parallel execution on multiple CPU threads does not speed up the contraction; instead, it slightly increases the execution time compared to the sequential approach. For the QFT29 circuit, only modest speedups (< 1.5) are observed, with little improvement as more threads are added. However, for the QFT33 circuit and beyond, speedups become significant, peaking at $15\times$ for the QFT35 circuit. It is expected that even greater speedups can be achieved for larger circuits, provided that sufficient memory is available.

As discussed in the previous section, the efficiency of parallel execution in these contractions depends heavily on the tensor ranks and the indices involved. Higher-ranked tensors and a greater number of con-

tracted indices allow better exploitation of first-level parallelism in `BliContractor`. This explains the increasing benefits of parallelism with increasing circuit size.

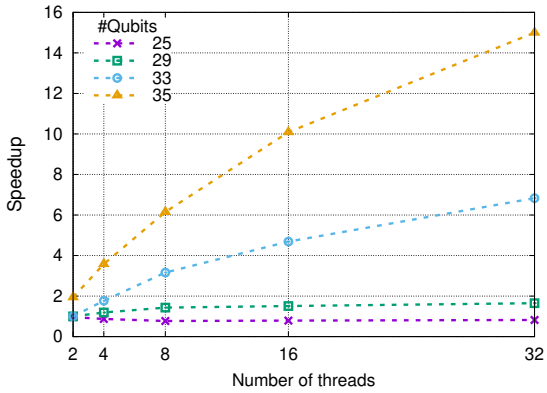


Figure 4: Speedup evolution of the parallel algorithm using `BliContractor` for QFT circuits of different sizes (#Qubits).

Next, we compare the performance of the three Julia packages when contracting QFT-type circuits of different sizes using the single-stage algorithm. To do this, we compute the speedup achieved by `OMEinsum`, `BliContractor` and `CUDA` (for GPU-accelerated contraction) relative to the sequential execution time of the fastest method provided by `OMEinsum`. Figure 5 shows the speedup evolution for the three packages as the circuit size increases. The results for the CPU-based packages correspond to the fastest execution using 32 threads.

Results show that GPU-based contraction with `CUDA` consistently outperforms CPU-based methods, while the two CPU-based packages achieve similar performance regardless of circuit size. In particular, for the smallest circuits (25 and 29 qubits), all three packages yield modest speedups. In fact, parallel `BliContractor` shows longer execution times than the sequential version of `OMEinsum`. As the circuit size increases, all three methods improve, although CPU-based packages show only marginal gains, while the GPU-based package exploits its many-core architecture to achieve significant and growing speedups, reaching up to 4.75× for the QFT35 circuit. These results suggest that even greater improvements can be achieved for larger circuits, provided sufficient memory is available.

Finally, we compare the performance of the three packages for two contraction algorithms: single-stage and multi-stage. Figure 6 presents the speedups achieved by the three packages relative to the best sequential algorithm for two circuit types: a QFT35 circuit and an RQC_12_12_14 circuit. The latter consists

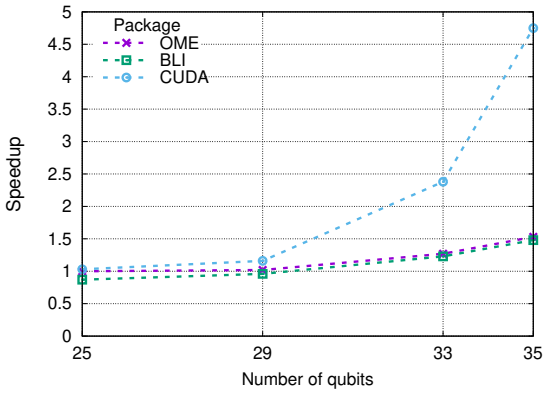


Figure 5: Speedup evolution using different packages for QFT circuits of varying sizes for single-state algorithm.

of a 12×12 qubit array with 14 layers. The reported speedups reflect only the portions of contraction that leverage first-level parallelism. Specifically, for the single-stage method, speedups account for total contraction time, while for the multi-stage method, they correspond to speedups achieved in the third stage.

These results confirm that GPU execution consistently delivers superior performance, while the two CPU-based packages show similar behaviour. In addition, the performance of all three packages is comparable for both circuit types in the single-stage approach. However, the multi-stage algorithm achieves significantly better performance for QFT circuits compared to RQC circuits. These results highlight the impact of circuit structure on parallelism benefits. In particular, there is a distinction between contracting all tensor pairs (single-stage) and contracting only a much smaller set of tensors with high ranks and numerous indices (multi-stage).

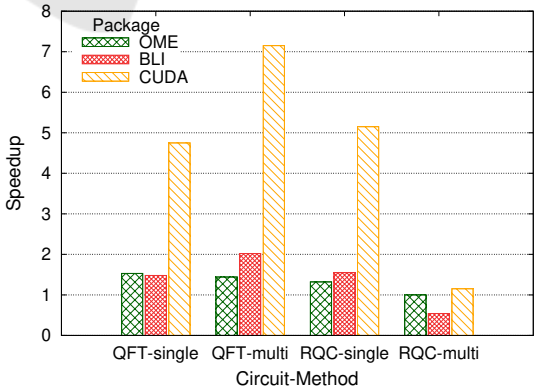


Figure 6: Parallel performance of the three Julia packages for the single- and multi-stage methods. Results for QFT35 and RQC_12_12_14 circuits.

7 CONCLUSIONS

In this work, we have evaluated the benefits of parallelizing pairwise tensor contractions on both multicore CPUs and GPUs. Our experimental results, obtained using three Julia packages, show that exploiting this level of parallelism can significantly accelerate the tensor network contraction process. In particular, we observed that the massive data parallelism provided by GPUs significantly outperforms the performance of multicore CPUs.

For our experiments we used the QXTools package, which relies on `OMEinsum` for pairwise tensor contraction on the CPU. Our results indicate that `OMEinsum` gets limited speedups from parallel execution in most cases. In contrast, `BliContractor` achieves a more effective use of first-level parallelism. However, since `BliContractor` is considerably slower in sequential contraction, its overall performance only slightly exceeds that of `OMEinsum` when using parallel execution.

Finally, we observed that parallel performance is strongly influenced by the structure of the circuit and by whether first-level parallelism is applied to the contraction of all tensor pairs or is restricted to the contraction of the reduced network formed by the communities detected in the initial tensor network. These results highlight the importance of choosing an appropriate contraction strategy based on the circuit properties to maximise computational efficiency.

ACKNOWLEDGEMENTS

This research was funded by the project PID2023-146569NB-C22 supported by MICIU/AEI/10.13039/501100011033 and ERDF/UE.

REFERENCES

Arute, F., Arya, K., Babbush, R., et al. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

Bayraktar, H., Charara, A., Clark, D., Cohen, S., Costa, T., Fang, Y.-L. L., Gao, Y., Guan, J., Gunnels, J., Haidar, A., et al. (2023). cuQuantum SDK: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 1050–1061. IEEE.

Brennan, J., Allalen, M., Brayford, D., Hanley, K., Iapichino, L., O'Riordan, L. J., Doyle, M., and Moran, N. (2021). Tensor network circuit simulation at exascale. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*, pages 20–26. IEEE.

Brennan, J., O'Riordan, L., Hanley, K., Doyle, M., Allalen, M., Brayford, D., Iapichino, L., and Moran, N. (2022). Qxtools: A julia framework for distributed quantum circuit simulation. *Journal of Open Source Software*, 7(70):3711.

Girvan, M. and Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826.

Gray, J. and Kourtis, S. (2021). Hyper-optimized tensor network contraction. *Quantum*, 5:410.

Huang, C., Zhang, F., Newman, M., Ni, X., Ding, D., Cai, J., Gao, X., Wang, T., Wu, F., Zhang, G., et al. (2021). Efficient parallelization of tensor network contraction for simulating quantum computation. *Nature Computational Science*, 1(9):578–587.

Lyakh, D. I., Nguyen, T., Claudino, D., Dumitrescu, E., and McCaskey, A. J. (2022). ExaTN: Scalable GPU-accelerated high-performance processing of general tensor networks at exascale. *Frontiers in Applied Mathematics and Statistics*, 8:838601.

Markov, I. L. and Shi, Y. (2008). Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981.

Matthews, D. A. (2018). High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24.

Nielsen, M. A. and Chuang, I. L. (2010). *Quantum computation and quantum information*. Cambridge university press, New York.

NVIDIA (2023). *cuBLAS Library User Guide*. https://docs.nvidia.com/cuda/cublas/index.html.

NVIDIA (2024). cuTENSOR: A High-Performance CUDA Library For Tensor Primitives. https://docs.nvidia.com/cuda/cutensor.

Pan, F., Gu, H., Kuang, L., Liu, B., and Zhang, P. (2024). Efficient quantum circuit simulation by tensor network methods on modern gpus. *ACM Transactions on Quantum Computing*, 5(4):1–26.

Pastor, A. M., Badia, J. M., and Castillo, M. (2025). A community detection-based parallel algorithm for quantum circuit simulation using tensor networks. *The Journal of Supercomputing*, 81. Art. no. 450.

Quantiki (2023). List of QC simulators. https://quantiki.org/wiki/list-qc-simulators.

Springer, P. and Bientinesi, P. (2018). Design of a high-performance GEMM-like tensor–tensor multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 44(3):1–29.

Villalonga, B., Boixo, S., Nelson, B., Henze, C., Rieffel, E., Biswas, R., and Mandrà, S. (2019). A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Information*, 5(1):86.

Vincent, T., O'Riordan, L. J., Andrenkov, M., Brown, J., Killoran, N., Qi, H., and Dhand, I. (2022). Jet: Fast quantum circuit simulations with parallel task-based tensor-network contraction. *Quantum*, 6:709.