Benchmarking the TensorCircuit Library on a Quantum Machine Learning Task

Peter Wegmann
School of Computation, Information and Technology
Technical University of Munich
peter.wegmann@tum.de

***Abstract*—TensorCircuit [6] is a novel library for simulating quantum circuits. It is built upon state-of-the-art libraries, offering many performance-enhancing features. This allows for the simulation and execution of large circuits with up to ~ 600 qubits. Focus is given to high-performance and efficient implementations of the underlying libraries. In order to determine the performance of TensorCircuit, a Quantum Machine Learning (QML) task is executed and benchmarked. A comparison to other quantum and machine learning libraries is additionally given.**

***Index Terms*—quantum computing, tensor networks, benchmark, performance**

I. Introduction

TensorCircuit [6] is a quantum circuit simulator based on tensor network contractions [5] that is fully compatible with the key features of modern machine learning frameworks. The authors of TensorCircuit focus on general performance and speedup over existing libraries. In contrast to existing libraries, a tensor network contraction simulator [5, 4] is utilized over a state vector simulator.

TensorCircuit lists the combination of fast circuit simulation with the following key features as a major improvement over other libraries:

- **Automatic Differentiation** Easy utilization using e. g. the *JAX* backend
- **JIT Compilation** Easy utilization provided by the configurable backend
- **Batch processing** Easy utilization by the implemented *vvag* function
- **GPU support** Native support by the underlying *TensorNetwork* library and backends

Combining these elements in a user-friendly fashion allows TensorCircuit to reach better performances over competitor libraries.

This paper is structured as follows: Related work is discussed in Section II. Methods and performed benchmarks are described in Section III. Final results are visible in Section IV.

II. Related Work

The TensorCircuit [6] paper lays major focus on the comparison to Pennylane [1] and TensorFlow Quantum [2]. Less focus is given to the tensor network library in the background [5, 4], which is actually responsible for huge speedups, as well as huge circuit sizes of up to 600 qubits.

TensorCircuit combines and fully supports fast quantum circuit simulation with the key features presented in section I, making it faster than Pennylane or Qiskit (see Section IV-C). Pennylane also supports using tensor network simulation based on matrix product state (MPS) [3]. In contrast, the utilized TensorNetwork [5] library generally outperforms other simulator ansätze, as it is less likely to run into memory bottlenecks. According to the authors, TensorCircuit reaches orders of magnitude speedups compared to other libraries. Furthemore, many performance enhancing features can be used in a more user friendly fashion (e. g. *vvag* and *JIT*).

In general, it can be said that TensorCircuit offers significantly better performance than other libraries. This stems from the focus on selecting efficient underlying libraries, as well as the capability of utilizing multi-threading both on CPU and GPU (integration with JAX and its support for JIT compilation). Many other libraries (e.g., Pennylane) support machine learning libraries but do not fully integrate them into the software stack. TensorCircuit, on the other hand, natively implements state-of-the-art machine learning libraries and offers easier (and more performant) utilization of such.
The TensorCircuit authors performed different tasks utilizing these libraries and benchmarked their execution time. While substantial speedups are shown, it is not always clearly stated which features or libraries are responsible. Additionally, in terms of (device) memory utilization, it is not always clear where data structures reside and how these are synchronized with the host device (when utilizing GPUs for computation).

III. Methods

A QML task is considered for benchmarking. This consists of a classification task utilizing a standard supervised learning approach with a neural network. Additionally, the neural network is combined with a parametrized quantum circuit. A reduced MNIST dataset is utilized, where ~ 12000 data points are used during training.

Presented results of the performed benchmarks are grouped into two distinct types:

- **TensorCircuit** and **JAX** benchmarks on the QML task
- **PennyLane** and **PyTorch** implementation of the QML task

All results and implementations can be found in the following repository:
https://github.com/Wegii/TensorCircuit-Benchmark.

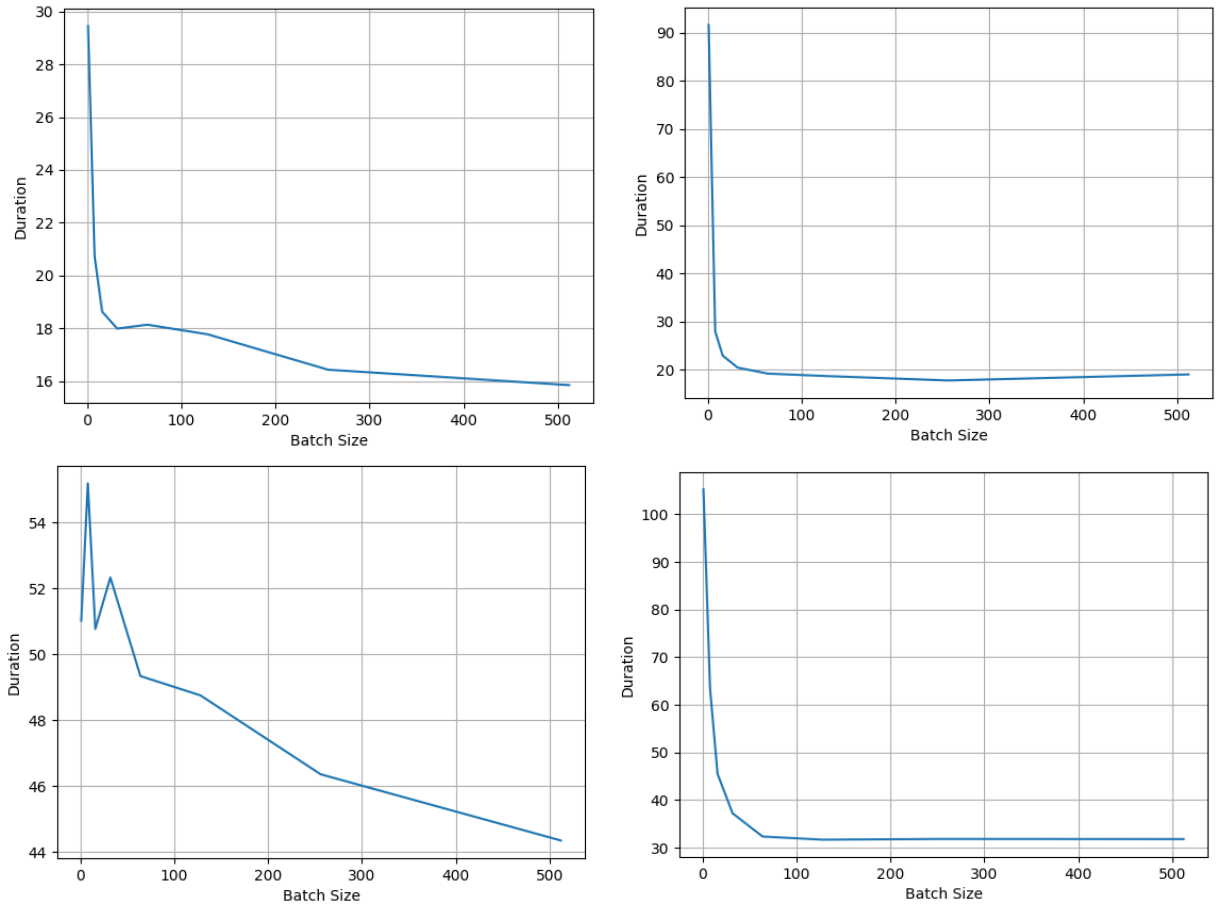

Fig. 1: Execution time of the full training loop for different batch sizes. Figures left show the execution time for the *Small Circuit* (top) and *Big Circuit* (bottom), executed on the CPU. The right column shows the same figures for GPU runs.

A. Testbed

The utilized testbed consists of a single CPU and four Nvidia GPUs. All runs utilized a single GPU only. The following information is provided in order to reproduce comparable results. Resulting performance and accuracy can heavily depend on the utilized hardware.

The utilized CPU and GPUs are as follows:

- AMD EPYC 7402 with 256GB RAM
- 4x NVIDIA RTX 3080 Turbo

IV. Results

This section presents the results of the different benchmarks and experiments performed. Section IV-A focuses on the training loop of the QML tasks and analyzes the execution times of each step performed. Additionally, execution time is evaluated for different batch sizes. In Section IV-B, only the TensorCircuit library is considered. Other steps, such as optimization or updates, are ignored. In Section IV-C achieved results are compared with an implementation of the QML task in Pennylane and PyTorch.

A. Training Loop

The major part of the QML task is the continuous execution of a training loop. This loop consists of three distinct steps:

- **Evaluation**: Evaluation of the Quantum Circuit and NN
- **Optimization**: Optimization of Quantum Circuit and NN weights
- **Update**: Updating of the utilized weights

Execution Time [ms]	CPU	GPU
Circuit	650 ± 200	750 ± 200
Optimize	20	60
Update	0.5	1

TABLE I: Execution times for the different steps executed during the training loop for CPU and GPU devices. A Batch size of 512 is selected.

Table I shows the execution time for the mentioned training loop steps. During execution, most time is spent evaluating the quantum circuit. Further investigation into the quantum circuit evaluation showed a major bottleneck at the configurable gates utilizing weights as their configuration. During the execution of the quantum circuit, 50% of the whole execution time is needed by the RX and RY gates utilizing a configurable theta

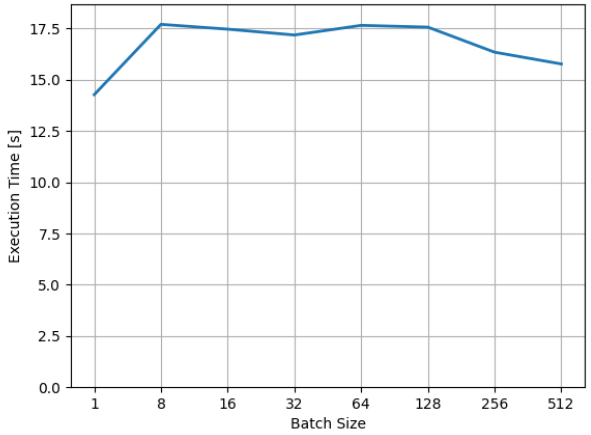

Fig. 2: Execution time for the quantum circuit evaluation using different batch sizes. The quantum circuit is executed on the CPU. Timing is performed on QML task training loop without the optimization and update step.

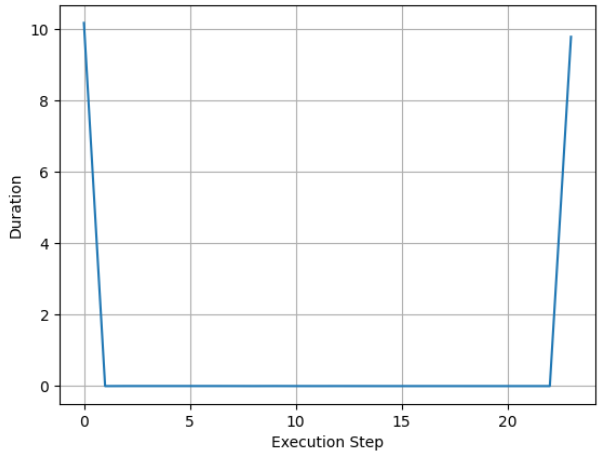

Fig. 3: Execution time of the quantum circuit over multiple steps during the training loop with a batch size of 512 and 25 iterations. The quantum circuit is executed on the GPU. Similar behavior was visible for CPU runs. Timing is performed on the QML tasks training loop without the optimization and update step.

value. This most likely results from accessing the weights. Performing the expectation evaluation needed another 35% of the whole execution time.

Since the batch size and number of utilized qubits present important hyperparameters, the QML task is performed using different batch sizes varying from 1 up to 512. Furthermore, the quantum circuit is configured to use 9 qubits (*Circuit Small*) and 12 qubits (*Big Circuit*). Figure 1 visualizes the resulting time measurements.

For the *Small Circuit* the GPU performs comparable to the CPU runs. For smaller batch sizes, the CPU is able to outperform the GPU runs. This results from the increased memory transfers to and from the GPU after the quantum circuit execution (further memory profiling utilizing *nsys* is necessary). Optimization and weight updates are performed on the CPU. For the *Big Circuit* configuration, the GPU outperforms the CPU runs significantly. In general, a non-continuous GPU utilization of $\sim$ 80% is reached. For even bigger circuits (increased number of qubits), the GPU is able to achieve even higher speedups and becomes of paramount importance.

B. Quantum Circuit

During the execution of the QML task, most time is spent during the execution of the quantum circuit. This section focuses on the benchmarking of the quantum circuit execution alone. Thus, the optimization and update steps are not executed. Executing the quantum circuit on different batch sizes resulted in equal execution times. This can be seen in Figure 2. This results from the efficient task distribution provided by JAX.

Since the circuit is executed many times, Figure 3 shows the execution time over multiple time steps. A huge initial and final peak (9s and 7s, respectively) is observed. Both CPU and GPU runs resulted in the same behavior, whereas the GPU needed much lower execution times between the initial and final peak. This initial peak most likely stems from the primary JIT compilation phase, whereas the last peak needs further investigation.

C. Pennylane and PyTorch

To evaluate the performance of the TensorCircuit and JAX implementation, the QML task was additionally implemented in PennyLane (utilizing Cirq as a simulator) and PyTorch. During benchmarking, only the CPU was utilized. Like the results in Section IV-A, most time is spent in the quantum circuit evaluation ($\sim$ 95% of whole execution time). As it was not possible to utilize the batching capabilities (e. g. pytorch vmap), only one CPU thread was utilized. Due to a version conflict, JIT compilation was not possible using PennyLane Catalyst.

In general, speedups of 100x - 1000x were achieved using TensorCircuit.

References

[1] Ville Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2022. arXiv: 1811.04968 [quant-ph]. URL: https://arxiv.org/abs/1811.04968.

[2] Michael Broughton et al. *TensorFlow Quantum: A Software Framework for Quantum Machine Learning*. 2021. arXiv: 2003.02989 [quant-ph]. URL: https://arxiv.org/abs/2003.02989.

[3] Johnnie Gray. "quimb: A python package for quantum information and many-body calculations". In: *Journal of Open Source Software* 3.29 (2018), p. 819. DOI: 10.21105/joss.00819. URL: https://doi.org/10.21105/joss.00819.

[4] Johnnie Gray and Stefanos Kourtis. “Hyper-optimized tensor network contraction”. In: *Quantum* 5 (Mar. 2021), p. 410. ISSN: 2521-327X. DOI: 10.22331/q-2021-03-15-410. URL: https://doi.org/10.22331/q-2021-03-15-410.

[5] Chase Roberts et al. *TensorNetwork: A Library for Physics and Machine Learning*. 2019. arXiv: 1905.01330 [physics.comp-ph].

[6] Shi-Xin Zhang et al. “TensorCircuit: a Quantum Software Framework for the NISQ Era”. In: *Quantum* 7 (Feb. 2023). DOI: 10.22331/q-2023-02-02-912.