

Qandle: Accelerating State Vector Simulation Using Gate-Matrix Caching and Circuit Splitting

Gerhard Stenzel^a, Sebastian Zielinski, Michael Kölle, Philipp Altmann, Jonas Nüßlein
and Thomas Gabor
LMU Munich, Munich, Germany
fi

Keywords: Quantum Computing, Quantum Machine Learning, State Vector Simulation, Hybrid Machine Learning, Quantum-Classical Machine Learning, PyTorch.

Abstract: To address the computational complexity associated with state-vector simulation for quantum circuits, we propose a combination of advanced techniques to accelerate circuit execution. Quantum gate matrix caching reduces the overhead of repeated applications of the Kronecker product when applying a gate matrix to the state vector by storing decomposed partial matrices for each gate. Circuit splitting divides the circuit into sub-circuits with fewer gates by constructing a dependency graph, enabling parallel or sequential execution on disjoint subsets of the state vector. These techniques are implemented using the PyTorch machine learning framework. We demonstrate the performance of our approach by comparing it to other PyTorch-compatible quantum state-vector simulators. Our implementation, named *Qandle*, is designed to seamlessly integrate with existing machine learning workflows, providing a user-friendly API and compatibility with the OpenQASM format. Qandle is an open-source project hosted on GitHub and PyPI.

1 INTRODUCTION


Quantum machine learning (QML) is a rapidly expanding field that aims to combine the computational power of quantum computing with the flexibility and scalability of classical machine learning algorithms (Nielsen and Chuang, 2001; Kölle et al., 2024; Stamatopoulos et al., 2020; Zoufal et al., 2019). In recent years, machine learning has gained significant popularity and has been widely applied in various domains, including image and speech recognition, natural language processing, and recommendation systems. These applications often rely on deep learning models, which are trained on large datasets using substantial computational resources (Cerezo et al., 2021; Farhi et al., 2022; Rebentrost et al., 2014; Schuld and Petruccione, 2021; Bauckhage et al., 2022; Nielsen and Chuang, 2001).

Quantum machine learning seeks to harness the potential of quantum computing to solve complex optimization problems currently intractable for classical computers. By doing so, it offers a novel approach to addressing intricate challenges in machine learning and other disciplines (Nielsen and Chuang, 2001).

However, existing quantum hardware still faces several limitations, such as hardware noise without sufficient error mitigation and correction (Preskill, 2018), limited qubit connectivity (Wang et al., 2022), and a restricted number of qubits. These limitations impact the real-world performance of quantum machine learning algorithms and models.

To overcome these challenges, hybrid quantum-classical machine learning models have been developed. These models consist of classical and quantum layers, enabling training on either real hardware (with reduced noise impact due to their smaller scale) or simulators (Schuld and Petruccione, 2021). These simulators, which run on classical hardware such as CPUs or GPUs, are used to mimic the behavior of quantum circuits. They facilitate the rapid development and training of quantum machine learning models (Preskill, 2018).

The classical simulation of quantum circuits plays a crucial role in the development and testing of quantum machine learning models. Although the ultimate goal is to utilize quantum hardware to exploit quantum mechanical advantages, the current limitations of quantum computers make classical simulation an indispensable tool. It allows researchers to design, debug, and optimize quantum circuits in a con-

^a  <https://orcid.org/0009-0009-0280-4911>

trolled environment. Moreover, simulators facilitate the integration of quantum layers into classical machine learning models, enabling hybrid approaches that can be experimentally explored even when the use of quantum hardware is not accessible or practical. Thus, classical simulation serves as a valuable means to advance research in quantum machine learning despite the challenges of implementing it on real quantum computers.

As the computational complexity of quantum circuits increases exponentially with the number of qubits, the efficient performance of simulators plays a crucial role in advancing quantum machine learning. This paper introduces two novel methods, namely quantum gate matrix caching and circuit splitting, to accelerate the execution of quantum circuit simulation. We implement these methods in Qandle, a state-vector simulator we specifically designed for hybrid quantum-classical machine learning applications in conjunction with the widely adopted PyTorch library. Through a comparative analysis with existing PyTorch-compatible quantum state-vector simulators, Qandle demonstrates superior performance in terms of execution time and memory usage.

Our contributions are

1. the introduction of two novel methods, namely gate matrix caching and circuit splitting,
2. the implementation of these methods in a new simulator and
3. a performance comparison to existing approaches.

This paper is structured as follows: in Section 2, we introduce the required symbols and background. In Section 3, we analyze related work and elaborate our contribution. We then present our proposed performance enhancing techniques of gate matrix caching and circuit splitting in Section 4 and evaluate their implementation in Section 5. Our conclusion can be found in Section 6.

2 PRELIMINARIES

2.1 Symbols

In this paper, we adopt the most significant bit first (MSb 0) notation for representing quantum states. Under this notation, the state $|0000\rangle$ corresponds to all qubits being in the state 0, while the state $|0001\rangle$ represents all qubits being in the state 0 except for the last qubit, which is in the state 1. This notation allows for a consistent and unambiguous representation of quantum states throughout our analysis. Other symbols used include S for the state vector of $|\phi\rangle$, W

the total number of qubits, w is the current qubit. R_d is the (matrix representation of the) gate for a rotation around axis d , and \mathcal{R}_d the matrix representation of R_d on W qubits.

2.2 State Vector Simulation

The quantum state $|\phi\rangle$ of a system with W qubits can be represented as a vector of size 2^W . This vector contains the complex probability amplitudes of each of the 2^W possible states, ranging from $|00\dots 0\rangle$ to $|11\dots 1\rangle$. Thus, it fully describes the system's state at any given time.

Quantum gates, represented by unitary matrices, are applied to the quantum state to transform it. On real quantum hardware, the state vector is not directly accessible. Instead, it can be inferred from the probabilistic measurement results of the quantum system. However, these measurements only provide an approximation of the state vector due to the inherent noisiness of the hardware in the NISQ era (Preskill, 2018; Nielsen and Chuang, 2001).

In contrast, simulators that work with the full state vector can provide the exact state of the system at any given time. However, these simulators face a challenge when dealing with large circuits due to the exponential growth of the state vector with the number of qubits. Due to their deterministic nature, simulators excel in building, debugging, and training variational quantum circuits.

2.3 Hybrid Machine Learning

In the context of quantum machine learning, hybrid machine learning refers to integrating classical and quantum machine learning algorithms. This integration can be achieved by incorporating trainable quantum circuits into larger machine learning models or by applying classical machine learning techniques to optimize quantum circuits.

Typically, quantum models in this context take the form of quantum variational circuits, which consist of several groups of gates:

1. Embedding layers, which encode classical data into the quantum state of the circuit. Different embedding methods offer varying trade-offs between the expressiveness of the quantum state and the number of required qubits. Some circuit architectures employ "data re-uploading" techniques to enhance the expressiveness of the quantum state by embedding the same data points at multiple locations within the circuit, effectively reinforcing the circuit's memory of the input data.

2. Trainable layers, which are parameterized gates whose parameters serve as the trainable weights of the quantum model. These parameters, often represented as angles of rotational gates, can be optimized using classical optimization algorithms such as gradient descent or its variants.
3. Measurement layers, which extract relevant information encoded in the quantum state and map it to a classical output. This output can then be further processed or optimized. While simulators allow measurements at any point in the circuit, real quantum hardware typically only permits measurements as the final operation on a qubit due to its destructive nature.

These quantum models can be treated as black boxes, enabling seamless integration into existing machine learning workflows. They can be applied to a wide range of tasks, including classification, regression, clustering, and generative modeling.

During training, the weights of the quantum models are optimized using methods such as the parameter-shift rule or classical backpropagation. The parameter-shift rule enables the calculation of the gradient of the loss function without requiring knowledge of the internal workings of the quantum circuit, making it suitable for both real quantum hardware and simulators. It approximates the gradient using the finite difference method. On the other hand, classical backpropagation, which can be efficiently deployed on state-vector simulators, treats the quantum and classical parts of the machine learning model separately and allows for different optimization algorithms and learning rates, while allowing the use of classical optimization algorithms on the quantum weights, too.

2.4 Concept of Shapes

The concept of shapes is employed in accordance with the notion of shape in PyTorch (Paszke et al., 2019) tensors. A tensor is a potentially high-dimensional matrix, where the shape specifies the number of elements or sub-tensors in each dimension. For instance, a tensor with shape $(2, 3, 4)$ consists of two sub-matrices, each with three rows and four columns, resulting in a total of $2 \cdot 3 \cdot 4$ elements. In the context of quantum circuits, the quantum state S of a system with W qubits can be represented as a tensor of shape 2^W , containing the complex probability amplitudes of each of the 2^W possible states $|00\dots 0\rangle$ to $|11\dots 1\rangle$. This can be formulated as a complex vector $S \in \mathbb{C}^{2^W}$. By employing isomorphic transformations, we can reshape the tensor to a shape of (d_1, d_2, \dots, d_W) , where all d_i are equal to two and W

is the number of qubits. This changes the representation of the state from $S \in \mathbb{C}^{2^W}$ to $S \in \mathbb{C}^{2 \times 2 \times \dots \times 2}$. Intuitively, each dimension of this tensor represents a qubit of the quantum circuit. For example, the probability amplitude of the state $|010\rangle$ is stored in the tensor at position $(0, 1, 0)$, which corresponds to the first element of the first dimension, the second element of the second dimension, and the first element of the third dimension.

When applying a single qubit gate, represented by a $G \in \mathbb{C}^{2 \times 2}$ matrix, to the w -th qubit, we can reshape the shape of the quantum state from (2^W) to $(d_0 \times d_1 \times \dots \times d_{w-1} \times d_w \times d_{w+1} \times \dots \times d_W)$ (with all dimensions being 2), and then further rearrange the elements to $((d_0 \times d_1 \times \dots \times d_{w-1} \times d_{w+1} \times \dots \times d_W), d_w)$. This results in a tensor shape of $(2^{W-1}, 2)$, which can be multiplied with the gate matrix G and then reshaped back to the original $S \in \mathbb{C}^{2^W}$.

In the context of machine learning, the tensor is typically extended by an additional dimension representing the batch size of the data, expanding the shape to $(B, 2^W)$ or $(B, 2, 2, \dots, 2)$ (with B being the batch size, e.g., 16). This allows for processing multiple data points simultaneously during the same forward and backward passes.

3 RELATED WORK

3.1 PennyLane

PennyLane is a Python 3 software framework for differentiable programming of quantum computers (Bergholm et al., 2022). It provides support for a wide range of quantum hardware and simulators, and seamlessly integrates with machine learning libraries such as PyTorch (Paszke et al., 2019) and Tensorflow (Abadi et al., 2015), as well as other quantum software platforms including Qiskit (Qiskit contributors, 2023) (see also Section 3.2) and Cirq (Cirq developers, 2023). PennyLane distinguishes between quantum nodes and classical nodes, where quantum nodes represent the parts of the execution graph that run on a quantum device or simulator. The framework offers an extensive collection of quantum operations, encompassing single- and multi-qubit gates, measurements, and non-unitary operations such as the Reset operation. Furthermore, PennyLane provides built-in support for quantum chemistry simulations.

The performance of PennyLane is primarily dependent on the underlying quantum simulators, with different backend implementations offering varying trade-offs between computational speed and supported operations. Some simulators even support ex-

ecution on NVIDIA GPUs to further enhance performance. To expedite the execution of the same quantum circuit with different parameters, PennyLane employs caching techniques. These caches are however only effective for executing the same circuit with the same parameters multiple times. Additionally, most gates and simulators support batching (albeit not all), a common technique in machine learning. PennyLane's circuit cutting allows for executing parts of a circuit independently, allowing to run big circuits on smaller hardware. However, this process comes at huge overhead in simulation time and memory usage. PennyLane also offers circuit visualization methods and supports importing and exporting circuits in the OpenQASM 2.0 format (Cross et al., 2017; Bergholm et al., 2022).

3.2 Qiskit

Qiskit is a comprehensive framework for quantum computing developed by IBM (Qiskit contributors, 2023; Wille et al., 2019). It offers a wide range of quantum operations, including single- and multi-qubit gates, measurements, and non-unitary operations. Qiskit provides access to real quantum hardware through the IBMQ Experience, allowing users to run their quantum circuits on IBM's quantum computers or simulators in the cloud. Local simulators are also available without the need for registration.

To optimize quantum circuits for specific quantum devices, Qiskit offers a transpiler. The transpiler adapts the circuit to hardware-specific coupling constraints, which determine the allowed combinations of qubits for CNOT gates and their directions. It also handles gate restrictions by decomposing unsupported gates into the supported set of gates for the target hardware. Additionally, gate-fusing and gate-cancellation techniques are employed to reduce the total number of gates, resulting in improved execution time and mitigating hardware noise and errors.

It is important to note that Qiskit uses the least significant bit as the first bit (LSb 0), while most other frameworks use the most significant bit as the first bit (MSb 0). This distinction can lead to confusion when using multiple frameworks simultaneously.

Qiskit's integration with the IBMQ Experience provides researchers and developers with valuable resources for exploring and experimenting with quantum computing. The combination of its extensive quantum operations, transpiler capabilities, and access to real quantum hardware makes Qiskit a powerful tool for quantum algorithm development and execution.

3.3 TorchQuantum

TorchQuantum (Wang et al., 2022) is a recently developed framework based on PyTorch, with a focus on execution speed and parallelization. It offers seamless integration with IBM's Qiskit, allowing for easy conversion of its models to Qiskit circuits. These circuits can then be executed on real quantum hardware using IBMQ or exported to the OpenQASM format.

TorchQuantum leverages distributed GPU computing to handle large-scale circuits and batch sizes, resulting in significant performance improvements compared to PennyLane. In fact, TorchQuantum has been reported to achieve execution time improvements of up to 1000 times (Wang et al., 2022). The framework inherits the support for backpropagation and batching from the PyTorch library, enabling efficient scaling with the number of qubits and batch size.

One notable feature of TorchQuantum is its design as a tool for running QuantumNAS, a noise-adaptive search for robust quantum circuits (Wang et al., 2022). This is achieved by dividing circuits into smaller sub-circuits and optimizing them independently. The sub-circuits are then combined using an evolutionary algorithm. This approach minimizes the impact of hardware noise and therefore maximizes performance on real quantum hardware, making it highly beneficial for quantum machine learning applications.

3.4 Contribution

Our contribution lies in the proposal and combination of advanced techniques aimed at accelerating the execution of quantum circuits. As a result, we have developed a high-performance state-vector simulator called Qandle, which offers seamless integration into PyTorch-based machine learning workflows. Qandle demonstrates significant improvements in execution times and memory usage compared to existing frameworks such as PennyLane, Qiskit, and TorchQuantum. Notably, both of our methods are matrix-based, making them highly compatible with PyTorch's `torch.compile` function, thereby further enhancing performance.

It is important to emphasize that our simulator does not aim to replace PennyLane or Qiskit. Instead, it serves as a valuable tool for quantum machine learning applications within the PyTorch ecosystem, similarly to TorchQuantum. Our simulator prioritizes efficient execution of quantum circuits on both CPU and GPU platforms, focusing on performance rather than providing advanced visualization tools or direct ac-

cess to quantum hardware, unlike more mature frameworks such as PennyLane, Qiskit and TorchQuantum.

By leveraging the presented techniques gate matrix caching (Section 4.1) and partial matrix decomposition (Section 4.2), our simulator optimizes the execution of gate operations on the state vector. This results in reduced computation (Section 5.2) and memory requirements (Section 5.3) during the forward pass of the quantum circuit.

The integration of our simulator with PyTorch enables seamless incorporation of quantum circuits into machine learning models. This allows researchers and practitioners to explore the potential of quantum computing in various domains, such as quantum chemistry simulations, optimization problems, and generative modeling. Furthermore, our simulator's compatibility with the OpenQASM format facilitates interoperability with other quantum software platforms, enabling easy integration with existing quantum algorithms and libraries, thanks to its user-friendly yet powerful API (Section 5.1).

In summary, we combine our presented methods of gate matrix caching and circuit splitting in our presented high-performance state-vector simulator Qandle, with reduced memory usage and increased execution speed and support for just-in-time compilation, making it an attractive choice for researchers and practitioners seeking to leverage the power of quantum computing in their machine learning workflows.

4 PERFORMANCE ENHANCING TECHNIQUES

4.1 Gate Matrix Caching

To improve execution times, we employ a technique we call gate matrix caching, which involves storing partial matrices of the gates. These partial matrices are decompositions of the gate matrices into two matrices with the same shape but higher sparsity. For instance, we can decompose the $R_x(\theta)$ gate into two matrices, R_{xa} and R_{xb} , both of shape $(2, 2)$, but with only two non-zero elements each.

The decomposition of the gate matrix is achieved as follows:

$$\begin{aligned} R_x(\theta) &= \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \cos(\theta/2) + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot -i\sin(\theta/2) \\ &= R_{xa} \cdot \cos(\theta/2) + R_{xb} \cdot -i\sin(\theta/2) \end{aligned} \quad (1)$$

The advantage of using these partial matrices is that they require fewer operations during the forward pass. Instead of allocating and filling the full gate matrix, we can simply multiply the parameters with their respective partial matrices, add the results together, and then multiply with the state vector. This reduces the computational complexity and improves the overall efficiency of the circuit.

Furthermore, the benefits of gate matrix caching are even more pronounced when working with circuits that involve multiple qubits. To expand the gate matrix R_x to the full size of the state vector, we compute the Kronecker product (\otimes) of the partial matrices R_{xa} and R_{xb} with identity matrices, resulting in $\mathcal{R}_x \in \mathbb{C}^{2^W \times 2^W}$.

$$\begin{aligned} \mathcal{R}_x(\theta) &= I_{2^w} \otimes R_x(\theta) \otimes I_{2^{(W-w)}} \\ &= I_{2^w} \otimes R_{xa} \otimes I_{2^{(W-w)}} \cdot \cos(\theta/2) \\ &\quad + I_{2^w} \otimes R_{xb} \otimes I_{2^{(W-w)}} \cdot -i\sin(\theta/2) \\ &= \mathcal{R}_{xa} \cdot \cos(\theta/2) + \mathcal{R}_{xb} \cdot -i\sin(\theta/2) \end{aligned} \quad (2)$$

It is crucial to note the correct execution order of the Kronecker product concerning the number of states 2^w for the qubits before the gate and $2^{(W-w)}$ for the qubits after the gate. This order is essential for the proper reshaping of the state vector after the gate application. By utilizing the cached partial matrices \mathcal{R}_{xa} and \mathcal{R}_{xb} , the application of the expanded matrices \mathcal{R}_x is faster than computing the full gate matrix for each forward pass, which would necessitate repeated applications of the Kronecker product.

Gate matrix caching is not limited to single-qubit gates but also extends to multi-qubit gates such as CNOT and composed gate structures like the rotational gates for angle embedding layers. For these, each rotational gate is decomposed into two partial matrices. For ease of access and better hardware-level caching, the two groups of partial matrices \mathcal{R}_a and \mathcal{R}_b are stacked into tensors of shape $(W, 2^W, 2^W)$. During embedding, the partial embedding functions (e.g., $f_{ax}(\theta) = \cos(\theta/2)$ and $f_{bx}(\theta) = -i\sin(\theta/2)$ for the R_x gate) are computed for all inputs, resulting in two vectors of shape (W) . These vectors are then multiplied with the partial matrices \mathcal{R}_a and \mathcal{R}_b , respectively, along the first axis. The resulting matrices are added together, forming the full sequence of gate matrices for the embedding layer, which can now be matrix multiplied with the state vector.

The computationally expensive parts of the embedding operation, such as the repeated application of the Kronecker product, are executed only once during circuit initialization and cached for future use. Although the cache is computationally fast, it becomes memory-intensive as the number of qubits increases.

To mitigate this, we employ circuit splitting (see Section 4.2).

While these matrices consist mostly of zeros (the matrix for W qubits has $2^{2W} - 2^W$ zeros), it would be advantageous to use sparse matrix representations, which are faster to multiply with another. However, our preliminary tests have shown that due to the constant multiplications with the quantum state (which is a very dense vector) and the consequent required type conversions, the overhead greatly outweighs the benefits of sparsity. Therefore, Qandle does not utilize sparse matrices for the gate matrices.

PennyLane, on the other hand, employs an aggressive caching approach, where the circuit structure, inputs, and outputs are saved in cache, with structure and input acting as keys. This caching strategy enables fast execution times for repeated executions of the same circuit, particularly when the number of gates and qubits is low. However, as the number of qubits increases, the cache becomes less effective. In many quantum machine learning applications, the input data changes with each forward pass, resulting in frequent cache misses. This further diminishes the benefits of PennyLane’s caching mechanism. The impact of PennyLane’s caching can be observed in the execution speed comparison presented in Figure 1.

4.2 Circuit Splitting

One of the major challenges faced by state vector simulators is the exponential growth of the state vector and the corresponding gate matrix size with the number of qubits. As the number of qubits, denoted by W , increases, a circuit’s state vector size becomes 2^W , and the gate matrices involved in the computations become $2^W \times 2^W$. Consequently, implementations of quantum circuits that rely on naive state vector and gate matrix multiplications struggle to handle larger circuits efficiently.

To address this computational complexity, we propose a technique called circuit splitting. The idea behind circuit splitting is to divide the circuit into smaller sub-circuits, thereby reducing the matrix sizes and the memory and computation time required. This splitting can be performed during circuit creation, eliminating the need to make a trade-off between splitting quality and execution time. The split circuits, which are essentially groups of quantum gates, can then be executed sequentially, operating only on a subset of the full state vector at a time.

To generate these groups, we interpret the circuit as a dependency graph, where each CNOT gate represents a node, ignoring other gates. In this graph, two CNOT gates are connected by an edge if they share

either a control or a target qubit and are successive in the circuit. Currently, our implementation utilizes a simple greedy algorithm. It iterates over all subtrees of the dependency graph and introduces a new group whenever the current group would exceed the given maximum number of qubits (typically between three and six). In the final step, the previously ignored single-qubit gates are added to the nearest group of CNOT gates on the same qubit.

The previously large circuit has been decomposed into smaller sub-circuits, which can be treated as unitary gates acting on multiple qubits. During circuit execution, the state vector is reshaped to match the dimensions of the sub-circuit. After applying the sub-circuit, the state vector is reshaped back to its original dimensions. In the reshaping process, the qubits involved in the sub-circuit are stored in a separate dimension. For example, if the circuit has five qubits labeled 0, 1, 2, 3, 4, and the sub-circuit acts on qubits 1 and 2, the reshaping would transform the state vector from (2^5) dimensions to $(d_0 \times d_3 \times d_4, d_1 \times d_2)$ dimensions. This allows for matrix multiplication between the sub-circuit (with a gate matrix $G \in \mathbb{C}^{2^2 \times 2^2}$) and the states over the last dimension. In the case of batched execution, the additional batch dimension of the state vector is merged during reshaping, while storing the original batch size b for reshaping back. This results in a reshaped state vector of dimensions $(b \times d_0 \times d_3 \times d_4, d_1 \times d_2)$ for batched execution. The overhead introduced by this reshaping process has a negligible impact on execution speed compared to the computational load of matrix multiplications. Additionally, hardware caching remains unaffected as the batches are processed independently.

4.3 Additional Optimizations

To enhance the quality of the machine learning process, we employ quantum weight remapping techniques (Kölle et al., 2023a; Kölle et al., 2023b). During the remapping process, all quantum weights are transformed to a new range, such as $[-\pi, \pi]$, using smooth functions like the hyperbolic tangent (\tanh). The additional computational overhead incurred by the remapping step is negligible compared to the numerous other operations performed during each forward pass. However, it yields noticeable improvements in the training process, including faster convergence and a more stable loss curve (Kölle et al., 2023a; Kölle et al., 2023b).

In addition, we encourage using PyTorch’s `torch.compile` function to further optimize the execution of our simulator. Since our implementation relies exclusively on PyTorch’s tensor operations, it can

be compiled into a single execution graph. This compilation process enables faster execution on both CPU and GPU by optimizing the execution graph. This optimization includes reordering the execution order of parallelizable operations to improve hardware cache layout and fusing consecutive reshaping operations into a single reshaping operation. By reducing the number of calls to system memory and CPU cycles, the compilation process can significantly enhance the overall performance of our simulator (Paszke et al., 2019).

5 IMPLEMENTATION AND EVALUATION

5.1 API

We showcase our proposed techniques by implementing a PyTorch-compatible state-vector simulator. It is designed to ensure compatibility with other quantum software platforms, facilitating easy exporting to the OpenQASM format. In addition, we provide a simple API that closely resembles the standard PyTorch API. This design choice allows for seamless integration of our circuits as `torch.nn.Modules` into existing machine learning workflows. Similar to conventional PyTorch modules such as convolutional layers, we store the quantum weights as parameters, eliminating the need for manual handling of the quantum weights and their gradients, as required in PennyLane. If users still desire to manually access or modify the weights, they can do so using the `parameters` method of the module.

5.2 Execution Time

The execution time of our proposed methods in our simulator is evaluated by comparing it to the execution times of PennyLane, Qiskit, and TorchQuantum. For PennyLane and Qiskit, which offer multiple backends each, the fastest available backend is chosen for each (determined through pretesting).

To ensure accurate measurements, warm-up runs are performed to allow on-demand/just-in-time compiling of modules, which are then stored in system memory. Random input data is sampled to simulate the execution of a larger dataset which exceed the capacity of CPU caches and system memory. The weights of the quantum circuit are modified using a classical optimizer. To minimize the influence of other components, a trivial loss function and the well-tested Adam optimizer (Kingma and Ba, 2017) are employed.

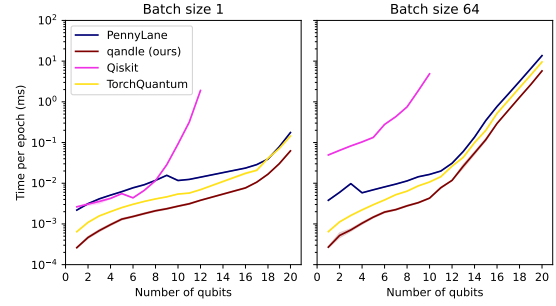


Figure 1: Simulation results for the network.

The evaluation of execution times (mean of 15 runs, other statistics shown in Figure 3) in Figure 1 demonstrates Qandle’s superior performance compared to other simulators. Qandle consistently outperforms TorchQuantum, which is specifically designed for high execution speed.

The speed curve reveals the impact of PennyLane’s caching mechanism. As the number of qubits increases, the execution times grow until a certain point, determined by the batch size, where the caching feature is disabled. At this point, the execution times briefly decrease before inevitably rising again. This behavior is a result of our experiment setup, which uses different inputs (sampled randomly) and weights (modified by the optimizer) for each forward pass, leading to cache misses. In scenarios where the same circuit is repeatedly executed without changes to the input or weights (e.g., for datasets that fit within the batch size or during inference), PennyLane’s caching mechanism would provide better performance than observed in this evaluation. We however argue that this is not a realistic scenario for training a quantum machine learning model.

5.3 Memory Usage

To evaluate memory usage, we executed the same circuits on different simulators and measured their peak memory usage. We employed a realistic training scenario, performing multiple backward passes with a simple loss function and varying input data to avoid caching effects. We measured the maximum resident set size (RSS) of the Python process, including the loaded simulator libraries and the PyTorch library, using the GNU `time` command. Each measurement was repeated 15 times, with negligible variance caused by swapping and other system processes. All tests were conducted on workstations with 64 GB of RAM and Intel Core i9-9900 CPUs. Simulators offering multiple backends, such as PennyLane and Qiskit, were executed with their fastest backend variants, `default.qubit.torch` and

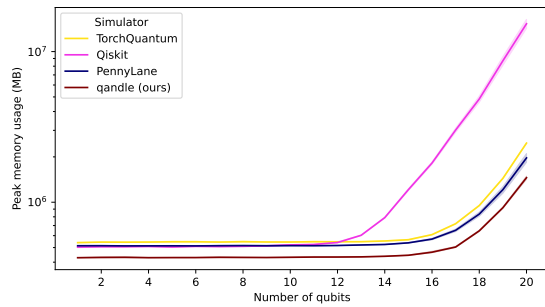


Figure 2: Memory usage for a hardware-efficient SU(2) circuit with varying numbers of qubits. Qandle exhibits lower memory usage compared to other simulators.

statevector_simulator on the Aer simulator, respectively (Bergholm et al., 2022; Qiskit contributors, 2023). As PennyLane’s caching mechanism is not effective in an activate training scenario, and circuit cutting is focused for execution on real hardware, neither of these features were enabled for the evaluation (pretesting showed a significant negative impact on execution time).

The memory scaling behavior exhibits similar characteristics to other simulators: even with optimizations, memory usage grows exponentially with the number of qubits. This is due to the large size of the state vector, which consists of 2^W complex numbers, and the associated memory overhead of matrix multiplications. Over the tested quantum circuits with up to 20 qubits (see Figure 2 for an implementation of a hardware-efficient SU(2) circuit over all qubits), our simulator demonstrates lower memory usage compared to other simulators, although it still scales exponentially with the number of qubits. TorchQuantum and PennyLane perform similarly (with a slight advantage for TorchQuantum), while Qiskit utilizes the most memory, potentially making it unsuitable for very large circuits.

6 CONCLUSION

This paper presents advanced techniques, namely quantum gate matrix caching and circuit splitting, to accelerate the execution of quantum circuits. The showcase implementation, Qandle, is a high-performance state-vector simulator that seamlessly integrates with PyTorch-based machine learning workflows. Qandle demonstrates significant improvements in execution times and memory usage compared to existing frameworks such as PennyLane, Qiskit, and TorchQuantum, validating the effectiveness of the proposed methods. Moreover, Qandle’s compatibility with PyTorch’s `torch.compile` func-

tion further enhances its performance. The user-friendly API of Qandle enables easy integration, even for users with limited experience in quantum machine learning and quantum computing, thereby expanding the accessibility of quantum machine learning to a wider audience.

Based on the promising performance of the proposed methods, we recommend incorporating them into other existing simulators.

As part of future work, we plan to expand the range of supported quantum gates, particularly multi-qubit gates like the Toffoli gate. This expansion will enable the simulation of more complex circuits that are currently not supported by our implementation. Additionally, we aim to develop a more sophisticated splitting algorithm based on graph algorithms, leveraging the circuit’s dependency graph. This algorithm will determine the optimal split, reducing the number of sub-circuits and minimizing the overhead of reshaping the state vector, while ensuring efficient execution. We propose exploring graph coloring techniques or split decomposition algorithms for this purpose.

ACKNOWLEDGEMENTS

This paper was partially funded by the German Federal Ministry of Education and Research through the funding program “quantum technologies — from basic research to market” (contract number: 13N16196).

REFERENCES

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
- Bauchhage, C., Bye, R., Knopf, C., Mustafic, M., Piatkowski, N., Reese, B., Stahl, R., and Sultanow, E. (2022). Quantum machine learning in the context of it security.
- Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M. S., Alonso-Linaje, G., Akash-Narayanan, B., Asadi, A., Arrazola, J. M., Azad, U., Banning, S., Blank, C., Bromley, T. R., Cordier,

- B. A., Ceroni, J., Delgado, A., Matteo, O. D., Dusko, A., Garg, T., Guala, D., Hayes, A., Hill, R., Ijaz, A., Isacsson, T., Ittah, D., Jahangiri, S., Jain, P., Jiang, E., Khandelwal, A., Kottmann, K., Lang, R. A., Lee, C., Loke, T., Lowe, A., McKiernan, K., Meyer, J. J., Montañez-Barrera, J. A., Moyard, R., Niu, Z., O’Riordan, L. J., Oud, S., Panigrahi, A., Park, C.-Y., Polatajko, D., Quesada, N., Roberts, C., Sá, N., Schoch, I., Shi, B., Shu, S., Sim, S., Singh, A., Strandberg, I., Soni, J., Száva, A., Thabet, S., Vargas-Hernández, R. A., Vincent, T., Vitucci, N., Weber, M., Wierichs, D., Wiersema, R., Willmann, M., Wong, V., Zhang, S., and Killoran, N. (2022). PennyLane: Automatic differentiation of hybrid quantum-classical computations.
- Cerezo, M., Arrasmith, A., Babbush, R., Benjamin, S. C., Endo, S., Fujii, K., McClean, J. R., Mitarai, K., Yuan, X., Cincio, L., and Coles, P. J. (2021). Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644.
- Cirq developers (2023). Cirq.
- Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). Open quantum assembly language.
- Farhi, E., Goldstone, J., Gutmann, S., and Zhou, L. (2022). The Quantum Approximate Optimization Algorithm and the Sherrington-Kirkpatrick Model at Infinite Size. *Quantum*, 6:759.
- Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.
- Kölle, M., Giovagnoli, A., Stein, J., Mansky, M. B., Hager, J., and Linnhoff-Popien, C. (2023a). Improving convergence for quantum variational classifiers using weight re-mapping.
- Kölle, M., Giovagnoli, A., Stein, J., Mansky, M. B., Hager, J., Rohe, T., Müller, R., and Linnhoff-Popien, C. (2023b). Weight re-mapping for variational quantum algorithms.
- Kölle, M., Stenzel, G., Stein, J., Zielinski, S., Ommer, B., and Linnhoff-Popien, C. (2024). Quantum denoising diffusion models.
- Nielsen, M. A. and Chuang, I. L. (2001). Quantum computation and quantum information. *Phys. Today*, 54(2):60.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.
- Preskill, J. (2018). Quantum computing in the NISQ era and beyond. *Quantum*, 2:79.
- Qiskit contributors (2023). Qiskit: An open-source framework for quantum computing.
- Rebentrost, P., Mohseni, M., and Lloyd, S. (2014). Quantum support vector machine for big data classification. *Physical review letters*, 113(13):130503.
- Schuld, M. and Petruccione, F. (2021). *Machine learning with quantum computers*. Springer.
- Stamatopoulos, N., Egger, D. J., Sun, Y., Zoufal, C., Iten, R., Shen, N., and Woerner, S. (2020). Option pricing using quantum computers. *Quantum*, 4:291.
- Wang, H., Ding, Y., Gu, J., Li, Z., Lin, Y., Pan, D. Z., Chong, F. T., and Han, S. (2022). Quantumnas: Noise-adaptive search for robust quantum circuits. In *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA-28)*.
- Wille, R., Van Meter, R., and Naveh, Y. (2019). IBM’s qiskit tool chain: Working with and developing for real quantum computers. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1234–1240.
- Zoufal, C., Lucchi, A., and Woerner, S. (2019). Quantum generative adversarial networks for learning and loading random distributions. *npj Quantum Information*, 5(1):103.

APPENDIX

Execution Time

In Figure 3, we show the minimum execution times of the simulators for the same circuits, employing a full forward and backward pass. The results are consistent with the mean execution times shown in Figure 1, showing Qandle as the fastest simulator, followed by TorchQuantum and PennyLane. Minimal execution times are more effected by other system processes and caching mechanisms, and are therefore less reliable to reproduce.

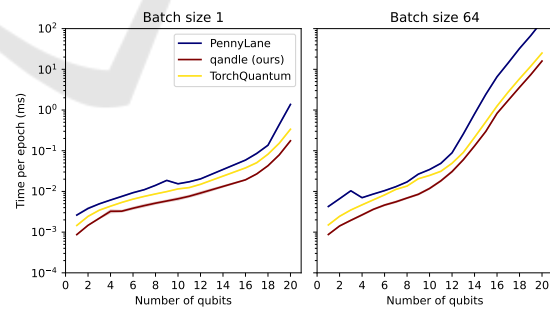


Figure 3: Simulation results for the network, showing only the fastest run.