Design and Implementation of a Quantum Computing Simulator with Multiple State Representations

Duarte Rebelo de São José
ddduarte@sapo.pt

Jens Verherstraeten
jensver@kth.se

GitHub Repository: GitHub-QuantumSimulator

Expected grade: *We expect to receive a grade of "A" as we developed a quantum computing simulator featuring multiple state representations and optimizations, and conducted extensive experiments to evaluate design choices.*

Work-division statement: *The work was evenly divided. Both Duarte and Jens worked on building the framework as well as testing and experimentation.*

***Abstract*—Classical quantum simulators play a critical role in developing and testing quantum algorithms when access to physical quantum hardware remains limited. This work presents a Python-based quantum computing simulator capable of representing quantum states using four distinct state representations: dense Cartesian, dense polar, sparse dictionary, and vectorized tensor representations. We evaluate the performance trade-offs of each approach in terms of memory consumption and computational efficiency, and benchmark our results against the Qiskit framework using the Quantum Fourier Transform (QFT) as a representative test case. Our findings demonstrate that careful data representation and vectorization techniques can yield substantial performance improvements, with our vectorized implementation outperforming Qiskit for small systems while maintaining competitive scaling for larger circuits.**

I. Introduction

Quantum computing exploits quantum mechanical principles such as superposition and entanglement to perform computations fundamentally different from classical computing. However, physical quantum hardware remains limited in availability, scale, and reliability, making classical quantum simulators essential tools for algorithm development and testing.

This paper presents the design, implementation, and evaluation of a quantum computing simulator that explores the impact of different state representation strategies on simulation performance. By implementing four distinct approaches (dense Cartesian, dense polar, sparse dictionary, and vectorized tensor representations) we systematically investigate the trade-offs between memory consumption, computational efficiency, and implementation complexity.

Our primary contributions are:

- Four functionally equivalent quantum simulator implementations with different internal representations
- Comprehensive performance analysis comparing memory usage and execution time
- Benchmarking against the industry-standard Qiskit framework
- Insights into design decisions that affect quantum simulation scalability

We focus our evaluation on the Quantum Fourier Transform (QFT), a standard benchmark that extensively performs single- and two-qubit operations while exhibiting gate complexity $O(n^2)$.

II. Background

A. Quantum State Representation

The qubit is the fundamental unit of quantum information. Unlike a classical bit, which must be either 0 or 1, a qubit can exist in a superposition of both states simultaneously. Mathematically, the state of a single qubit $|\psi\rangle$ is represented as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where α and β are complex probability amplitudes satisfying $|\alpha|^2 + |\beta|^2 = 1$.

A quantum system of n qubits is represented by 2^n complex amplitudes describing its complete state. Each additional qubit doubles the number of possible states, leading to exponential growth in the state space. This exponential scaling is both the source of quantum computing's potential power and the primary challenge in classical simulation.

B. Quantum Gate Operations

Quantum gates are modeled as unitary operators acting on the state vector:

$$|\psi'\rangle = U\,|\psi\rangle \tag{2}$$

where U is the gate's unitary matrix and $|\psi'\rangle$ is the resulting state. For an n-qubit system, a general gate would be represented as a $2^n \times 2^n$ matrix, making naive matrix multiplication computationally intractable.

However, most quantum gates act on only one or two qubits at a time. A single-qubit gate can be represented as a 2×2

matrix, while a two-qubit gate requires a 4×4 matrix. Efficient simulation exploits this structure to avoid constructing and applying exponentially large matrices.

C. Computational Challenges

The main challenge in simulating quantum systems is exponential resource growth:

- **Memory:** A dense representation requires $O(2^n)$ complex numbers. For 30 qubits, this already exceeds 16GB in double-precision format (16 bytes per complex number).
- **Computation:** Applying a single-qubit gate requires updating 2^{n-1} amplitude pairs, resulting in $O(2^n)$ operations. Two-qubit gates require $O(2^n)$ operations over quadruples.
- **Scalability:** These exponential costs limit classical simulation to approximately 40-50 qubits on high-end hardware.

These constraints motivate the exploration of alternative representations that may offer better performance for certain classes of quantum circuits.

III. Methodology

A. Design Goals

Our simulator was designed with four key objectives. First, we aimed to implement distinct state storage strategies to enable direct performance comparison. Second, we ensured functional equivalence across all implementations to enable verification and fair benchmarking. Third, we provided support for a universal set of quantum gates sufficient for practical circuits. Finally, we established a framework for comparative analysis, benchmarking our implementations against each other and against Qiskit to contextualize performance relative to an established framework.

B. State Representation Approaches

We implemented four distinct representations, each with different performance characteristics.

1) Dense Cartesian Representation: The `DenseCartesianSim` stores the complete state vector as a complex-valued NumPy array:

$$|0\rangle^{\otimes n} \longrightarrow [1+0j, 0+0j, \ldots, 0+0j] \in \mathbb{C}^{2^n} \tag{3}$$

Single-qubit gates are applied by iterating over all basis states and updating amplitude pairs whose indices differ only in the target qubit position. Bitwise operations efficiently compute these indices:

$$i_0 = i \;\&\; \sim(1 \ll t), \quad i_1 = i_0 \mid (1 \ll t) \tag{4}$$

where t denotes the target qubit index. Two-qubit gates similarly group amplitudes into quadruples.

This representation is conceptually transparent and straightforward to implement and debug, serving as our validation baseline. However, it exhibits $O(2^n)$ memory scaling and suffers from reduced computational efficiency due to explicit Python loops.

2) Dense Polar Representation: The `DensePolarSim` decomposes each amplitude into separate magnitude and phase arrays:

$$|0\rangle^{\otimes n} \longrightarrow \texttt{mag} = [1,0,0,\ldots] \in \mathbb{R}^{2^n}, \tag{5}$$

$$\texttt{phase} = [0,0,0,\ldots] \in \mathbb{R}^{2^n} \tag{6}$$

where each amplitude is reconstructed as $\psi_i = \texttt{mag}_i\, e^{i\,\texttt{phase}_i}$.

Gate operations require temporary conversion to complex form, followed by gate application and conversion back to polar form using $r = |\psi|$ and $\theta = \arg(\psi)$. This representation provides explicit access to magnitude and phase components and is potentially more efficient for phase-only operations. However, the conversion overhead between polar and complex forms introduces computational costs, while memory scaling remains $O(2^n)$.

3) Sparse Dictionary Representation: The `SparseDictSim` stores only non-zero amplitudes using a Python dictionary with the structure `state = { index : complex amplitude }`:

$$|0\rangle^{\otimes n} \longrightarrow \{\, 0 : 1+0j \,\} \tag{7}$$

For each non-zero amplitude, all related indices affected by the target qubit(s) are computed and updated during gate application. After each operation, amplitudes below a threshold ϵ are pruned to maintain sparsity.

This approach offers memory scaling proportional to the number of non-zero amplitudes rather than 2^n, making it highly efficient for sparse states. The primary limitations are dictionary manipulation overhead and performance degradation as states become denser, along with increased complexity in gate application logic.

4) Vectorized Tensor Representation: The `VectorizedSim` leverages NumPy's tensor operations to eliminate explicit Python loops. The 1D state vector is reshaped into an n-dimensional tensor:

$$\psi[i_0, i_1, \ldots, i_{n-1}] \;\leftrightarrow\; \psi_{i_0 i_1 \ldots i_{n-1}} \tag{8}$$

where each axis corresponds to one qubit.

For a single-qubit gate on qubit t, the target axis is permuted to the front, the gate is applied via batched matrix multiplication, and axes are restored. This approach avoids explicit Kronecker products while achieving the same mathematical operation:

$$\psi' = (U \otimes I^{\otimes(n-1)})\psi \tag{9}$$

Two-qubit gates are handled similarly by moving both target axes to the front, reshaping to $4 \times 2^{n-2}$, multiplying by the gate matrix, and reshaping back.

This representation maximizes NumPy's C-level optimizations and eliminates Python loop overhead, achieving significantly faster performance than loop-based approaches. However, memory scaling remains $O(2^n)$, axis permutation introduces some overhead, and the implementation is more complex than the straightforward Cartesian approach.

C. Gate Set and Interface Consistency

All simulator implementations support a universal gate set comprising X, Y, Z, H, S, T, parameterized rotations R_X, R_Y, and R_Z, as well as two-qubit gates CX, CZ, CP, and SWAP. Additionally, projective measurement in the computational basis is supported across all representations.

To ensure fair comparison, all representations implement identical public interfaces and produce numerically equivalent results. Any performance differences arise solely from representational efficiency rather than algorithmic variations. This design choice enables direct benchmarking while allowing future implementations to explore representation-specific optimizations.

IV. Implementation

A. Software Architecture

The simulator was developed in Python using NumPy 1.24 as the numerical backend. The complete source code is available on GitHub[1]. Each simulator variant is implemented as a separate module in the `/src` directory: `Dense_Cartesian.py` provides the reference implementation with explicit loops, `Dense_Polar.py` implements magnitude-phase decomposition, `Sparse_Dict.py` uses dictionary-based sparse storage, and `Vectorized.py` employs tensor-based vectorized operations. A shared `gates.py` module defines all quantum gate matrices as NumPy arrays, ensuring consistency across implementations.

B. Testing and Validation

A comprehensive test suite using pytest validates correctness by verifying single-qubit gate behavior on basis states, testing two-qubit gates including controlled and swap operations, and comparing outputs against Qiskit. All implementations pass identical test cases, confirming functional equivalence.

C. Benchmarking Framework

Performance evaluation measures memory consumption during circuit execution, execution time for complete circuit simulation, and scaling behavior as qubit count increases. Each benchmark is averaged over 10 runs to account for system variability. Measurements were performed on a system with an Intel Core i7-11850H processor and 32GB RAM running Python 3.13.7.

V. Results

A. Test Case: Quantum Fourier Transform

The Quantum Fourier Transform (QFT) serves as our primary benchmark. The QFT circuit contains $O(n^2)$ gates, comprising Hadamard gates, controlled phase rotations, and swap operations, making it computationally representative while remaining tractable for validation. An example for 5 qubits is illustrated in Figure 1. Its dense entangling structure exercises both single- and two-qubit operations extensively, providing a realistic performance assessment.

[1]https://github.com/DuarteSJ/Quantum_Computing_KTH/tree/main/quantum-simulator

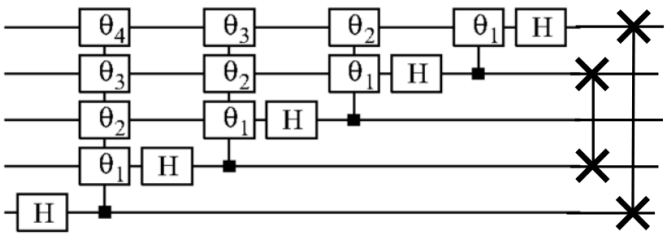

Fig. 1. The specific Quantum Fourier Transform circuit used in all tests and benchmarkings for $n = 5$ qubits.

Figure 2 illustrates memory usage as a function of the number of qubits. For small systems, all our implementations outperform Qiskit in terms of memory efficiency. However, as the number of qubits increases, Qiskit's lower-level optimizations allow it to scale more favorably. The dense representations (Cartesian, polar, and vectorized) follow the expected $O(2^n)$ scaling with almost identical memory footprints.

Interestingly, the sparse dictionary representation uses slightly more memory than the dense representations, which may seem counterintuitive given its design to reduce memory usage by storing only non-zero amplitudes. This phenomenon can be explained by the nature of the QFT circuit: while it does create some zero amplitudes through interference, this number is insufficient to outweigh the overhead introduced by dictionary storage, including keys, hash tables, and associated bookkeeping structures.

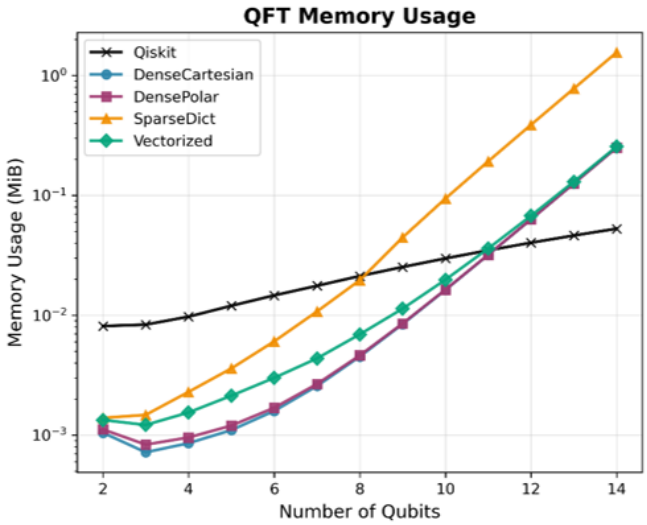

Fig. 2. Memory consumption for QFT simulation. Dense implementations (Cartesian, polar, vectorized) follow theoretical $O(2^n)$ scaling. Sparse representation shows limited memory savings in QFT due to the relatively small number of zero amplitudes, while Qiskit scales more efficiently for larger systems.

Figure 3 presents execution times across increasing qubit counts. Similar to memory performance, all our implementations outperform Qiskit for smaller systems, but Qiskit scales better for larger qubit counts due to its more optimized C++

backend.

The vectorized implementation achieves the best performance among our simulators, leveraging NumPy's highly optimized C-based tensor operations. In contrast, the dense Cartesian and dense polar implementations degrade quickly as system size increases, with the polar variant slightly slower due to conversion overhead between polar and complex representations.

Notably, the sparse dictionary consistently outperforms the dense Cartesian and dense polar implementations in execution time. This indicates that even a small number of zero amplitudes, caused by interference, can significantly improve performance. However, a higher proportion of zeros would be needed to achieve meaningful memory savings alongside these runtime improvements.

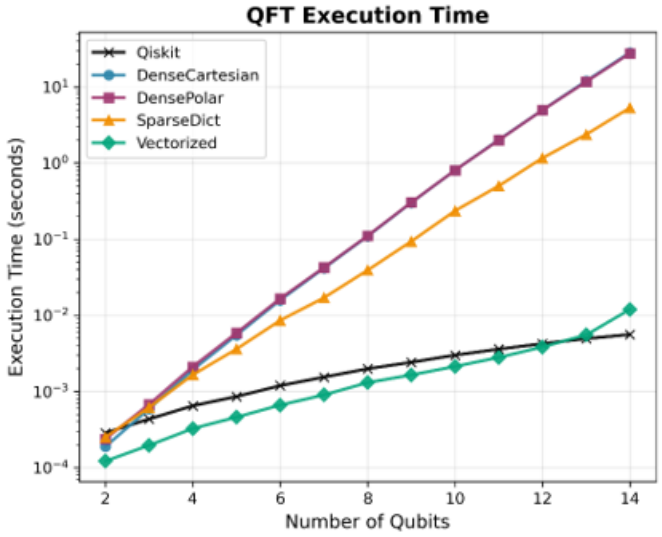

Fig. 3. Execution time for QFT simulation. The vectorized approach achieves the best performance among our implementations. Sparse dictionary reduces unnecessary operations, improving runtime despite limited memory savings. Qiskit's C++ backend provides superior scaling as system size increases.

B. Test Case: Sparse-Friendly Circuit

To further evaluate the efficiency of the sparse simulator, we utilized an idealized test case involving a quantum circuit designed to maintain a predominantly sparse state vector. In this experiment, gate operations were restricted to the first three qubits of an n-qubit register, leaving the remaining qubits in their initial state.

Consequently, the system evolves into a state with only eight non-zero amplitudes, corresponding to the 2^3 possible configurations of the active qubits, out of the total 2^n amplitudes in the full state space. Theoretically, this configuration enables the sparse simulator to achieve constant memory scaling, specifically $O(1)$ with respect to the total number of qubits, as only a fixed number of amplitudes are stored and updated regardless of system size. This theoretical prediction was validated by our experimental results, which are visualized in Figure 4.

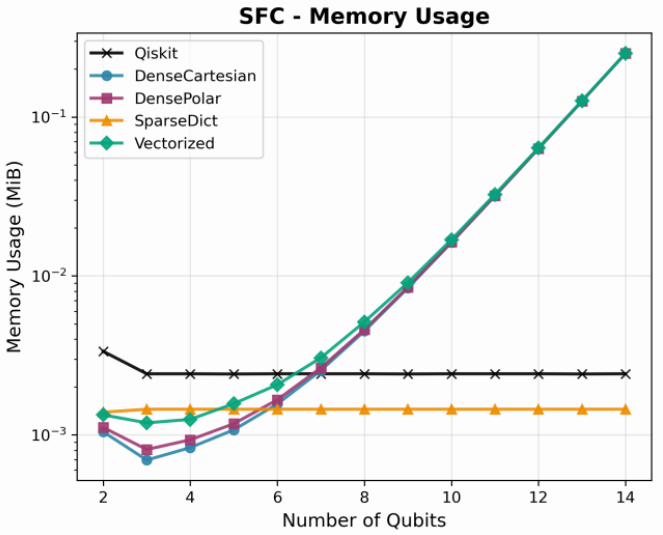

Fig. 4. Memory consumption for the SFC (Sparse-Friendly Circuit) simulation. Dense implementations follow the same theoretical scaling as before $O(2^n)$. For this instance, the sparse implementation and Qiskit both exhibit constant memory scaling in relation to the number of qubits.

Figure 5 illustrates the execution time of the different simulator implementations for the sparse-friendly circuit. As expected, the dense Cartesian and dense polar implementations exhibit exponential growth in execution time with respect to the number of qubits, as the sparsity of the state vector has no effect on their memory allocation or computational approach.

This circuit clearly demonstrates a viable use case for sparse representations. The sparse variant is now much quicker than the dense versions, highlighting its advantages when circuit structure maintains state sparsity. The sparse, vectorized, and Qiskit approaches all exhibit constant execution time regardless of the number of qubits, indicating that each simulator has mechanisms to apply gate operations selectively on non-zero states without wasting resources on zero-amplitude components.

The vectorized simulator remains efficient for small to medium systems, showing low execution times comparable to Qiskit up to around 12 qubits, before increasing more sharply at higher system sizes due to memory saturation effects. Overall, these results confirm that representation choice directly governs scalability. For circuits with limited entanglement and sparse activity, the sparse simulator provides orders-of-magnitude better scaling, with vectorized approaches trailing not far behind. Dense approaches are more suitable for fully entangled or dense quantum states.

VI. Discussion

A. Key Findings

Our results confirm theoretical expectations while revealing practical insights. First, vectorization matters significantly: the vectorized implementation demonstrates that high-level language performance can be substantially improved through careful use of optimized libraries. The 3-5× speedup over naive

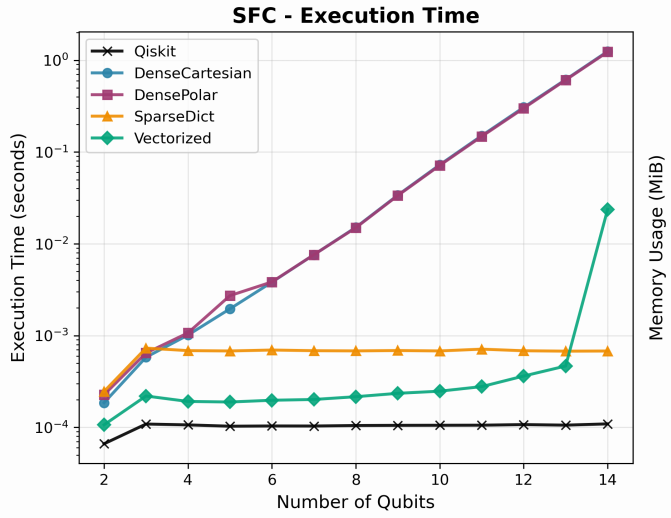

Fig. 5. Execution time for SFC (Sparse-Friendly Circuit) simulation. The Sparse and Qiskit implementations have a constant scaling $O(1)$ in relation to the number of qubits in the circuit. This graph demonstrates that circuits like SFC are practical use cases for sparse representations.

loops highlights the importance of leveraging NumPy's C-level operations.

Second, representation choice depends critically on circuit structure. Sparse representations excel for circuits maintaining low entanglement but degrade as states become dense, suggesting that circuit-specific analysis is necessary to select optimal representations. Third, Python can be competitive for small systems: our vectorized implementation outperforms Qiskit for systems below 8 qubits, indicating that framework overhead matters more than raw computational power at small scales.

B. Opportunities for Further Optimization

Several avenues for improvement were identified during this work. A hybrid sparse-dense switching approach could dynamically integrate multiple state representations to leverage the advantages of both methods depending on runtime state characteristics. Gate-specific optimizations could be implemented, particularly for diagonal gates such as Z, S, and T, which could be applied more efficiently in polar form without full conversion overhead. Parallel execution could be exploited by recognizing that gate operations on independent qubits are embarrassingly parallel. Finally, compiled implementations using Numba or Cython could further reduce overhead while maintaining Python's expressiveness and rapid prototyping advantages.

VII. Conclusion

This work demonstrates that the choice of state representation significantly impacts quantum simulation performance. Our implementation of four distinct representations (dense Cartesian, dense polar, sparse dictionary, and vectorized tensor) reveals that careful data structure design and vectorization can yield substantial performance improvements even in high-level languages like Python.

The vectorized implementation achieved speed-ups of 3-5× speedup over naive approaches while maintaining code clarity and correctness. For small systems (less than 8 qubits), it outperformed the industry-standard Qiskit framework, although Qiskit's lower-level optimizations provide superior scaling for larger circuits.

The sparse representation demonstrated the importance of matching representation to circuit structure, performing well for low-entanglement cases but degrading as states densify. This suggests that adaptive representations switching between sparse and dense storage based on runtime state analysis could provide optimal performance across diverse circuit types.

Beyond specific performance metrics, this project provides a flexible, well-tested framework for exploring quantum simulation strategies. Consistent interfaces and comprehensive validation enable future work to investigate advanced techniques such as hybrid representations, gate-specific optimizations, and parallel execution strategies.

The fundamental exponential scaling of quantum simulation ensures that classical simulators will remain limited to moderate qubit counts. However, within these constraints, thoughtful implementation choices can substantially improve efficiency. This is a critical consideration as quantum algorithms continue to be developed and refined on classical hardware.

Acknowledgments

The source code and complete benchmark results are available at https://github.com/DuarteSJ/Quantum_Computing_KTH/tree/main/quantum-simulator.

References

[1] IBM, "Qiskit Documentation," IBM Quantum, Accessed: Oct. 19, 2025. [Online]. Available: https://docs.quantum.ibm.com/

[2] J. M. Romero, E. Montoya-González, G. Cruz, and R. C. Romero, "A novel quantum circuit for the quantum Fourier transform," arXiv preprint arXiv:2507.08699, 2025. [Online]. Available: https://arxiv.org/abs/2507.08699