Accelerated Quantum Algorithm Prototyping: Modular Simulation and Noise Modelling with CUDA Support

Elmin Marevac [1], Esad Kadušić [2], Nataša Živić [3,*], and Christoph Ruland [4]

[1] Polytechnic Faculty, University of Zenica, Zenica, Bosnia and Herzegovina
[2] Faculty of Educational Sciences, University of Sarajevo, Sarajevo, Bosnia and Herzegovina
[3] Faculty of Digital Transformation, Leipzig University of Applied Sciences, Leipzig, Germany
[4] Department of Electrical Engineering and Computer Science, University of Siegen, Siegen, Germany
Email: elmin.marevac@unze.ba (E.M.); ekadusic@pf.unsa.ba (E.K.); natasa.zivic@htwk-leipzig.de (N.Z.); christoph.ruland@uni-siegen.de (C.R.)
*Corresponding author

***Abstract*—This work presents a portable, modular quantum-circuit emulator designed to accelerate algorithm prototyping by leveraging Central Processing Unit (CPU) and Compute Unified Device Architecture (CUDA)-enabled Graphic Processing Unit (GPU) backends for dense linear-algebra workloads. The emulator provides a lightweight backend abstraction that unifies NumPy and CuPy, supports both statevector and density-matrix representations, and implements efficient multi-target unitary embedding and Kraus-map noise channels. These capabilities enable rapid exploration of quantum algorithms that require full $2^n \times 2^n$ operators, such as quantum phase estimation, full Quantum Fourier Transform (QFT), and noise-aware variational circuits, while minimizing host/device transfer and preserving numerical parity between backends. Implementation strategies are described for embedding single- and multi-qubit gates, parameterized rotations, and controlled unitaries, along with an "apply_unitary_on_targets" routine that employs reshape/matrix-multiply/reshape patterns to map arbitrary target sets to dense GPU kernels for high throughput. Using a curated benchmark suite of dense quantum workloads and microbenchmarks, including large matrix multiplies and Fast Fourier Transforms (FFT), performance is quantified across a range of qubit counts and precisions. The results demonstrate substantial speedups on modern NVIDIA GPUs for workloads dominated by dense linear algebra, identify cross-over points where GPU acceleration becomes beneficial, and analyse memory/precision trade-offs for complex64 versus complex128. The presented emulator lowers the barrier for noise-aware algorithm prototyping by combining practical software ergonomics with GPU performance, enabling faster iteration on algorithm design and noise-mitigation strategies on commodity single-node hardware. While the abstraction layer ensures portability, achieving peak performance currently requires hardware-specific tuning—a limitation we address through future automation strategies.**

***Keywords*—quantum circuit simulation, Graphic Processing Unit (GPU) acceleration, statevector simulation, density-matrix simulation, quantum noise modelling, Kraus map**

I. Introduction

Classical simulation remains an essential pillar of quantum-algorithm research. Despite rapid progress in quantum hardware, Noisy Intermediate-Scale Quantum (NISQ) devices are constrained by qubit count, coherence times, gate fidelity, and connectivity; as a result, algorithm development, protocol verification, error-mitigation research, and hardware-software co-design continue to rely heavily on classical emulation. Accurate, flexible simulators let researchers prototype new circuits, validate gate decompositions, and explore noise models before committing experiments to scarce quantum hardware. They also provide a reproducible environment to quantify algorithmic sensitivity to noise, enable controlled parameter sweeps for variational algorithms, and serve as reference oracles for debugging and benchmarking [1].

Dense full-matrix operations and density-matrix simulation occupy a particular niche in this ecosystem. While many simulator designs emphasize sparse updates or path-integral sampling that scale well for certain gate sets and large qubit counts, there are important algorithmic and validation scenarios were representing entire $2^n \times 2^n$ operators is the right tool. Quantum Fourier transform (QFT) and Quantum Phase Estimation (QPE), for example, are naturally expressed as large unitary matrices whose dense application exposes computational structure useful for profiling and optimization. Likewise, density-matrix representations are necessary when studying general noise channels, mixed states, decoherence, and Kraus-map evolutions; they are the canonical formalism for simulating open quantum systems and hardware noise models. For small-to-medium qubit

Manuscript received November 17, 2025; revised January 28, 2026; accepted March 17, 2026; published June 26, 2026.

counts (the regime accessible to many research groups and early hardware), these dense computations are tractable and often the most direct, interpretable way to evaluate algorithm robustness [2].

Despite the clear need for dense linear-algebra capabilities, existing publicly available simulators frequently face limitations when targeting Graphic Processing Unit (GPU)-accelerated dense workloads. First, many simulators were designed around Central Processing Unit (CPU)-centric primitives and gate-by-gate sparse updates; porting them to a GPU backend often requires nontrivial Application Programming Interface (API) changes, careful management of device memory, and minimizing host–device synchronization. Second, differences between numerical libraries (NumPy vs. CuPy or vendor-specific runtimes) produce portability and reproducibility issues: array creation, dtype semantics, and host transfer semantics differ subtly, which can lead to silent performance regressions or correctness issues [2]. Third, dense simulations are memory and compute bound: applying a full $2^n \times 2^n$ unitary or evolving a $2^n \times 2^n$ density matrix consumes large contiguous memory and benefits from fused, high-throughput linear algebra kernels; naive gate embedding or repeated kronecker operations can dramatically increase runtime and device memory pressure. Finally, there is a shortage of curated, reproducible benchmark suites focused specifically on dense, GPU-friendly quantum workloads: without such benchmarks, it is hard to determine the qubit crossover point where GPU acceleration overtakes optimized CPU baselines, or to quantify accuracy/memory tradeoffs for lower-precision computation [3]. This work is specifically optimized for small-to-medium qubit counts (up to ~25–30 qubits), which aligns with current experimental capabilities and practical algorithm development needs. While this scope limits direct scalability to very large systems, it enables efficient prototyping on commodity hardware where full quantum state representation remains tractable. We explicitly target researchers prototyping noise-aware algorithms (requiring density-matrix simulation) at medium scales on single-node hardware. For circuits with fewer than 10 qubits, CPU backends are often preferable; however, GPU acceleration becomes consistently advantageous for systems in the 10–16 qubit range, particularly for dense or deep circuits.

In this work we address the above gaps with 4 concrete contributions.

- A modular emulator architecture with a unified backend abstraction (xp) that transparently targets NumPy or CuPy. This design minimizes code duplication, preserves numerical parity between backends, and simplifies running identical experiments on CPU and Compute Unified Device Architecture (CUDA)-enabled GPUs. In this paper, "simulation" refers to the mathematical process of reproducing quantum state evolution, while "emulator" denotes the classical software system implementing that process.
- Robust algorithms and implementations for multi-target unitary embedding and density-matrix operations. We detail an "apply_unitary_on_targets" routine that reshapes state tensors and employs dense matrix–matrix kernels to map arbitrary target sets efficiently, together with Kraus-map support for noise modelling.
- A curated GPU-heavy benchmark suite and example pipelines tailored to dense quantum workloads (random full-matrix chains, QFT as a full matrix, Kraus evolutions, and microbenchmarks such as large matrix multiplies and Fast Fourier Transforms (FFTs)). The suite is designed to expose device memory scaling, precision tradeoffs, and performance cross-over points.
- Fully reproducible experiments and tooling: pinned dependency manifests, scripts to run CPU/GPU comparisons, and example pipelines that minimize host–device transfers and detect Out-of-Memory (OOM) conditions gracefully.

The remainder of the paper is organized as follows. Section II reviews relevant background and related work in quantum simulation and GPU acceleration. Section III presents our methods and algorithmic design, including mathematical notation, gate embedding, and backend abstraction. Section IV describes implementation details and the software architecture. Section V defines the experimental setup and measurement methodology. Section VI reports correctness and performance results, followed by an analysis in Section VII. Section VIII concludes and outlines future directions.

II. Background and Related Work

Quantum computation simulation represents a critical bridge between theoretical quantum algorithms and practical implementation, enabling algorithm development and validation before quantum hardware becomes widely available [1]. As highlighted by Montanaro [2], classical simulation remains essential for algorithm verification and noise impact assessment in the Noisy Intermediate-Scale Quantum (NISQ) era. This section reviews fundamental concepts in quantum simulation and surveys the current landscape of classical simulation approaches, building upon decades of research in efficient quantum circuit emulation.

The foundation of quantum state simulation rests on linear algebraic representations of quantum states and operations, as formalized in Nielsen and Chuang's seminal work [1]. A pure quantum state of n qubits is represented by a complex vector $|\psi\rangle$ in a 2^n-dimensional Hilbert space. This statevector representation requires $O(2^n)$ complex numbers to store the amplitudes of all basis states. Alternatively, mixed quantum states are described by density matrices ρ, requiring $O(4^n)$ complex numbers for full representation, as demonstrated by Aaronson and Gottesman [4] on stabilizer circuits.

Single-qubit quantum gates are represented by 2×2 unitary matrices. For example, the Hadamard gate H, which creates quantum superposition, is expressed as Eq. (1).

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1}$$

Multi-qubit operations are constructed through Kronecker products of these basic matrices. For an n-qubit system, the complete operation matrix has dimensions $2^n \times 2^n$. The application of a quantum gate to a state requires matrix-vector multiplication, with computational complexity $O(2^n)$ for statevector simulation and $O(4^n)$ for density matrix evolution. Gray and Kourtis [5] found that optimizations reducing this complexity for specific gate sequences through tensor network contractions.

A. Survey of Classical Simulators

The landscape of quantum circuit simulators has evolved significantly over the past decade, with various frameworks offering different approaches to the simulation challenge. Markov and Shi [6] established early benchmarks for classical simulation strategies. Among full-state simulators, several frameworks have emerged as leaders in the field. OpenQASM, described by Cross *et al.* [7], represents a comprehensive approach to quantum simulation, offering statevector, density matrix, and stabilizer simulation capabilities. Their implementation notably demonstrated successful simulation of 32-qubit circuits on contemporary hardware, while incorporating GPU acceleration for enhanced performance. The ProjectQ framework, developed by Steiger *et al.* [8], took a different approach by implementing a high-performance C++ backend with Python interfaces. Their architecture introduced novel approaches to quantum circuit compilation, particularly in optimizing resource utilization during simulation.

Google's Cirq platform, detailed in simulations conducted by Isakov *et al.* [9], has focused specifically on addressing the challenges of near-term quantum algorithms. Their implementation places particular emphasis on realistic noise modelling and hardware-specific constraints, making it especially valuable for NISQ-era algorithm development. Johansson *et al.* [10] proposed the Quantum Toolbox in Python (QuTiP), which established itself as the standard framework for quantum optics simulations, offering robust density matrix simulation capabilities that excel in modelling open quantum systems and solving master equations.

The landscape of GPU-accelerated quantum simulation has seen significant advancement with the introduction of NVIDIA's cuQuantum library [11]. Their implementation achieves remarkable speedups for circuits with high gate density through sophisticated tensor network and statevector simulation techniques. Complementary efforts in GPU-based simulation include the qulacs framework [12], and QCGPU [13], each demonstrating impressive performance characteristics for specific classes of quantum circuits.

Villalonga *et al.* [14] systematically analysed how the challenge of managing exponential complexity in quantum simulation has led to diverse strategic approaches. McClean *et al.* [15] conducted extensive comparisons of simulation approaches for Variational Quantum Eigensolver (VQE) algorithms, demonstrating that while sparse methods excel for certain quantum chemistry applications, they may underperform for highly entangled states. Bravyi *et al.* [16] pioneered perfect sampling techniques that strategically trade complete state knowledge for efficient measurement outcome simulation. In the realm of distributed simulation, Smelyanskiy *et al.* [17] demonstrated the impressive capabilities of their qHiPSTER framework, achieving simulation of 42-qubit systems across 1024 nodes while carefully managing the inherent communication overhead of distributed computation.

B. Position of Current Work

Our work occupies a specific niche in the quantum simulation ecosystem, building upon recent advances in GPU-accelerated quantum computing by Aaronson *et al.* [18], and memory-efficient simulation techniques by Wang *et al.* [19]. While existing simulators often prioritize either broad feature sets or specialized hardware optimization, we have developed an approach that bridges these extremes while maintaining focus on practical usability and reproducibility.

The foundation of our work lies in efficient single-machine execution, incorporating recent advances in quantum circuit optimization [20]. We explicitly clarify that this system is not intended to replace vendor-optimized simulators (e.g., NVIDIA cuQuantum) for peak throughput. Instead, it occupies a niche focused on dense operator experimentation, cross-backend validation, and reproducible performance analysis on commodity hardware. We focus specifically on small to medium-scale circuits where full statevector representation remains feasible, typically up to 25 qubits. This scope allows us to implement highly optimized dense linear algebra operations for both CPU and GPU targets. Our emphasis on maximizing single-node performance through careful memory management and computational kernels builds directly on techniques introduced by Gheorghiu [21], while extending them to accommodate modern hardware architectures.

Reproducibility and portability form core principles of our implementation strategy. Following challenges identified by Fingerhuth *et al.* [22], we have developed a platform-independent implementation that ensures consistent results across different hardware configurations. The architecture maintains a clear separation between abstract quantum operations and hardware-specific optimizations, adhering to design principles outlined by Smith *et al.* [23].

The educational and research value of our implementation derives from its transparent approach to fundamental quantum operations. Drawing inspiration from the extensible architecture of PyQuil [23], we have created a modular design that facilitates easy extension and modification. Our built-in tools for algorithm prototyping and performance analysis incorporate benchmark suites developed by McCaskey *et al.* [24], providing a robust framework for comparative analysis and optimization.

This work addresses a critical need in the quantum computing community for a reliable, performance-oriented simulator that maintains accessibility and extensibility. Our approach builds upon recent work in quantum circuit simulation optimization by Zhang *et al.* [25], while introducing novel techniques for

memory management and parallel execution. The result is a system particularly valuable for algorithm development and educational purposes where full state information and reproducible results are essential.

III. Materials and Methods

This section presents the formal contract governing the emulator's interface, the mathematical foundations of the supported quantum state representations, and the algorithmic strategies employed to apply dense multi-qubit operators efficiently on both CPU and GPU backends. We describe the core reshape–matrix-multiply–reshape primitive used for multi-target updates, detail the construction of parameterized and controlled gates, explain the backend abstraction layer and the measures taken to ensure numerical parity between NumPy and CuPy, and conclude with the performance optimizations and validation plan that establish correctness and reproducibility.

A. Functional Contract

The simulator adheres to a well-defined functional contract specifying its inputs, outputs, and permissible failure modes.

The inputs comprise 4 elements. First, a circuit description is provided as an ordered list of gate records; each record specifies the gate identifier, any required real-valued parameters, and the list of target qubit indices. Second, a backend selection flag determines the numeric namespace xp, which resolves to either NumPy for CPU execution or CuPy for GPU execution. Third, the simulation mode is selected as either statevector or density, indicating whether the state will be represented as a pure state vector or a density matrix. Fourth, the number of qubits n, a positive integer, is supplied; this determines the Hilbert space dimension $d = 2^n$.

The outputs consist of the final quantum state, specifically, a complex column vector of length d in statevector mode or a $d \times d$ complex matrix in density mode, together with optional derived quantities such as measurement samples, expectation values, or reduced density matrices. A diagnostics bundle is also returned, containing per-phase timing information (e.g., matrix construction, application, and data transfer phases), peak memory usage, and error metrics relative to a reference when available.

Acceptable failure modes are explicitly enumerated. An Out-of-Memory (OOM) condition may occur on either the host or the device when the state size or intermediate buffers exceed available memory. Unsupported or malformed gate descriptions, such as a mismatch between the declared number of target qubits and the dimensions of the supplied operator, or invalid qubit indices, trigger well-defined exceptions. Finally, numerical anomalies arising from inconsistent data type (dtype) usage across the computational pipeline (e.g., mixing complex64 and complex128 without explicit conversion) are detected and reported.

B. State Representation and Foundational Equations

The emulator supports the 2 canonical representations of quantum states, each with its associated evolution equations.

In the statevector formalism, an n-qubit pure state is represented by a normalized complex column vector $|\psi\rangle \in \mathbb{C}^d$, where $d = 2^n$. Closed-system unitary evolution is expressed as Eq. (2).

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

where $U \in \mathbb{C}^{d\times d}$ is unitary, and $|\psi'\rangle$ denotes the evolved quantum state after applying the unitary operation, U to the initial state $|\psi\rangle$. In practice, U is rarely supplied as a full $d \times d$ matrix; instead, a smaller operator acting on a subset of qubits is embedded into the global space.

For mixed states and open-system dynamics, the state is represented by a density operator $\rho \in \mathbb{C}^{d\times d}$, which is Hermitian, positive semidefinite, and satisfies $Tr(\rho) = 1$. Unitary evolution acts via conjugation, as Eq. (3).

$$\rho' = U\,\rho\,U^{\dagger} \tag{3}$$

where $U^{\dagger}$ denotes the conjugate transpose (Hermitian adjoint) of the unitary operator U, and ρ' is the evolved density operator after applying the unitary transformation U. More generally, non-unitary quantum channels are expressed via Kraus decompositions, as Eq. (4).

$$\rho' = \textstyle\sum_k K_k\,\rho\,K_k^{\dagger}, \qquad \sum_k K_k^{\dagger} K_k = I \tag{4}$$

where $\{K_k\}$ are the Kraus operators and I denotes the identity operator on the system's Hilbert space. Efficient application of such maps is implemented by keeping intermediate accumulations on the compute device and minimizing temporary allocations [26].

Throughout, the Kronecker product convention adopts the right-most factor as the least-significant qubit, and qubit indices are zero-based unless otherwise noted. The alias xp denotes the selected numeric backend [26].

C. Gate Embedding and Multi-Target Handling

The mathematical embedding of a single-qubit gate $g \in \mathbb{C}^{2\times 2}$ acting on qubit j (where qubits are indexed from 0 to $n-1$) is given by Eq. (5).

$$G_j = I_{2^{n-j-1}} \otimes I_{2^j} \otimes g \tag{5}$$

where I_m denotes the $m \times m$ identity matrix, the left factor $I_{2^{n-j-1}}$ acts on the $n-j-1$ qubits more significant than qubit j, the right factor I_{2^j} acts on the j less significant qubits, and $\otimes$ is the Kronecker product. The operator G_j represents the application of the single-qubit gate g to qubit j while leaving all other qubits unchanged. For a k-qubit operator $U_{(k)} \in \mathbb{C}^{2^k\times 2^k}$ acting on a set of target indices T, the analogous embedding replaces the corresponding identity blocks with $U_{(k)}$. Explicit Kronecker expansion of these embeddings is

infeasible for moderate n; therefore, the implementation relies on a reshape–matrix-multiply–reshape strategy that achieves the same effect using dense linear-algebra kernels without materializing the full $d \times d$ operator [27].

The core algorithm partitions the Hilbert space as $H = H_L \otimes H_T \otimes H_R$, where H_T corresponds to the k target qubits and H_L, H_R to the remaining qubits. By permuting and reshaping the global state into a matrix $S \in \mathbb{C}^{2^k \times d/2^k}$, the application of $U_{(k)}$ reduces to a single matrix multiplication, as shown in Eq. (6).

$$S' = U_{(k)} S \tag{6}$$

Here, S is the reshaped state tensor with the k target qubit indices grouped into the first dimension (rows) and all other qubits into the second dimension (columns); S' is the resulting transformed matrix after applying the k-qubit unitary $U_{(k)}$ to the target subspace. The result is then reshaped and inverse-permuted to restore the canonical ordering. This pattern maps naturally to high-throughput Basic Linear Algebra Subprograms (BLAS)-like kernels and benefits from the parallelism and memory bandwidth of modern GPUs [26]. To facilitate independent re-implementation, the core algorithm proceeds as follows: (1) Compute the permutation vector that moves target axes to the leading dimensions; (2) Reshape the state tensor from $(2,\ 2, \cdots, 2)$ to $(2^k, 2^{n-k})$; (3) Perform the dense matrix multiplication $S' = U^{(k)} S$; (4) Reshape back to $(2,\ 2, \cdots, 2)$; (5) Apply the inverse permutation. This strategy avoids materializing the full $d \times d$ operator, reducing memory complexity from $O(4^n)$ to $O(2^n)$ for the state plus $O(4^k)$ for the gate.

When the target indices are contiguous, the required reshape is a zero-copy view; for non-contiguous targets, a canonical permutation is computed to group the target axes, and the cost is amortized by gate fusion or block-wise updates where possible. The dominant computational cost of Eq. (5) is $\mathcal{O}(2^k \times d)$ scalar operations; when $k = n$, this reduces to the expected $\mathcal{O}(d^2)$ cost of applying a full unitary, while for small k it is substantially cheaper than naive embedding [26].

D. Parameterized Gates and Controlled Unitaries

Parameterized single-qubit rotations are constructed from exact closed-form expressions. For a rotation angle $\theta \in \mathbb{R}$, the Pauli-axis rotations are defined as Eqs. (7)–(9).

$$R_x(\theta) = e^{-i\theta X/2} = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \tag{7}$$

$$R_y(\theta) = e^{-i\theta Y/2} = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \tag{8}$$

$$R_z(\theta) = e^{-i\theta Z/2} = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \tag{9}$$

In these expressions, $i = \sqrt{-1}$ is the imaginary unit; X, Y and Z are the Pauli matrices, which serve as generators of rotations about the respective Cartesian axes of the Bloch sphere, defined as Eqs. (10)–(12).

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{10}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{11}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{12}$$

The matrix exponential $e^{-i\theta P/2}$ for $P \in \{X, Y, Z\}$ yields the corresponding rotation operator $R_P(\theta)$. Specifically, $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$ are single-qubit unitary gates that implement rotations of the qubit state by an angle θ about the x-, y-, and z-axes of the Bloch sphere, respectively. These matrices are generated with an explicit dtype argument (complex64 or complex128) to ensure consistency with the state container.

Controlled unitaries are implemented in 2 ways. When the joint control-target subspace is small enough to materialize a block, the controlled operator is constructed as a block-diagonal matrix; for a single control and target gate U, this yields Eq. (13).

$$\mathrm{CU} = \begin{pmatrix} I_2 & 0 \\ 0 & U \end{pmatrix} \tag{13}$$

here, *CU* denotes the controlled-U operation: it acts on a two-qubit (or more generally, $(k+1)$-qubit) space where the first qubit is the control and the remaining qubits host the unitary U. If the control qubit is in state $|0\rangle$, the identity I_2 is applied (leaving the target unchanged); if the control is $|1\rangle$, the gate U is applied to the target subsystem. This block form is then embedded as a $(k+1)$-qubit unitary and applied via the reshape strategy. When the composite subspace is large, the implementation performs a conditional amplitude update: only the slices of the state corresponding to the control qubits being in the $|1\rangle$ state is selected, and U is applied to those slices. This avoids allocating a large block matrix while preserving the asymptotic operation count [28].

E. Backend Abstraction and Numerical Parity

Portability between CPU and GPU is achieved through the xp alias, which provides a unified interface to NumPy and CuPy. The backend contract mandates support for the following primitives: array creation (xp.array, xp.zeros, xp.eye), dense linear algebra (xp.dot, xp.kron), tensor restructuring (reshape, transpose), explicit dtype control (complex64, complex128), random number generation (xp.random.seed, xp.random.randn), and host conversion (xp.asnumpy). All arrays are constructed via xp to ensure correct placement and type; the asnumpy function is implemented as a no-op on the NumPy backend to allow unconditional use in diagnostic code [29].

To ensure numerical parity, the implementation seeds the random number generator consistently across backends, uses explicit dtype specifications in all gate factories, and avoids implicit host–device transfers by

keeping intermediate tensors on the xp device. While exact bitwise equivalence between NumPy and CuPy is not guaranteed due to differences in library implementations and reduction ordering, deterministic execution order and fixed seeds ensure statistical parity and bounded numerical discrepancies.

F. Performance Considerations and Optimizations

A number of engineering optimizations are employed to control memory footprint and improve computational throughput. In-place updates and buffer reuse reduce peak memory allocations; view and transpose operations are preferred over explicit copies. On CuPy backends, the memory pool is leveraged to amortize allocation overhead [29].

For very large Hilbert space dimensions where intermediate matrices exceed device memory, a chunked matrix multiplication fallback is provided: the reshaped matrix S is partitioned along its second dimension into column blocks, and U is applied to each block sequentially, trading additional kernel launches for reduced peak memory consumption.

The choice of data type is treated as a deliberate performance knob: complex64 halves memory and bandwidth requirements relative to complex128 and often yields higher throughput on GPUs, at the expense of reduced numerical precision. Users are expected to evaluate fidelity tradeoffs when selecting precision.

Minimizing host–device transfers is a central design tenet: intermediate arrays remain on the xp device throughout the simulation, and xp.asnumpy is reserved exclusively for final result extraction or diagnostic logging. Memory alignment and coalesced access patterns are favoured to maximize effective bandwidth utilization.

G. Correctness Metrics and Validation Plan

Quantitative metrics are used to assess numerical accuracy. For pure states, fidelity ($\mathcal{F}$) is computed as Eq. (14).

$$\mathcal{F} = |\langle\psi_{\text{ref}} \mid \psi_{\text{sim}}\rangle|^2 \tag{14}$$

Here, $|\psi_{\text{ref}}\rangle$ denotes the reference (exact or trusted) state vector, $|\psi_{\text{sim}}\rangle$ is the simulated state vector, $\langle\psi_{\text{ref}} \mid \psi_{\text{sim}}\rangle$ is their inner product (a complex number), and the squared magnitude $|\cdot|^2$ yields a real value in $[0, 1]$ that quantifies state overlap, where $\mathcal{F} = 1$ indicates perfect agreement. For density matrices, trace distance is defined as Eq. (15).

$$D(\rho, \sigma) = \frac{1}{2} Tr(|\rho - \sigma|) \tag{15}$$

In this expression, ρ and σ are 2 density operators (Hermitian, positive semidefinite, unit-trace matrices); $\rho - \sigma$ is their difference; $|\rho - \sigma| = \sqrt{(\rho - \sigma)^\dagger (\rho - \sigma)}$ denotes the positive semidefinite square root (i.e., the matrix absolute value, obtained via spectral decomposition); and $Tr(\cdot)$ is the trace. The trace distance $D(\rho, \sigma)$ ranges from 0 (identical states) to 1 (orthogonal states) and is a metric on the space of quantum states. The relative Frobenius norm error for arbitrary matrices is Eq. (16).

$$\varepsilon_F = \frac{\|A - B\|_F}{\|A\|_F} \tag{16}$$

where A and B are matrices of the same dimensions (e.g., unitaries or superoperators), and $\|\cdot\|_F$ denotes the Frobenius norm, defined as $\|M\|_F = \sqrt{\sum_{i,j} |M_{ij}|^2} = \sqrt{Tr(M^\dagger M)}$. The quantity ε_F measures the relative element-wise error between A and B, with $\varepsilon_F = 0$ indicating exact equality.

The validation plan comprises three components. Unit tests verify correctness of primitive operations, including Pauli gates, parameterized rotations, and small multi-target embeddings, against analytical expectations. Regression tests compare results from xp = NumPy and xp = CuPy runs for identical seeds and circuit descriptions, confirming numerical parity within prescribed tolerances. End-to-end tests execute representative circuits in both statevector and density modes and compare outputs against established reference simulators such as Qiskit and QuTiP for small system sizes [30].

Together, these methods provide a robust, high-performance foundation for dense quantum circuit simulation on single-node CPU and GPU systems, balancing theoretical rigor, numerical reproducibility, and pragmatic engineering considerations.

IV. Implementation Details

This section presents the concrete realization of the algorithmic framework described in Section III through a rigorously engineered software system. The implementation is designed to uphold mathematical correctness while delivering high performance on both CPU and GPU platforms. We describe the modular architecture and its 5 principal components, detail the memory management and optimization strategies employed to mitigate the exponential resource demands of quantum simulation, explain the backend abstraction layer enabling hardware portability and acceleration, outline the advanced features supporting measurement, noise modeling, and circuit optimization, and summarize the validation, performance, and reproducibility frameworks that ensure scientific rigor.

A. Architectural Overview and Design Philosophy

The emulator is organized around a modular architecture that reflects modern software engineering principles of high cohesion and low coupling. This structure facilitates independent development, testing, and extension of individual subsystems while preserving well-defined interfaces for inter-module communication. At the highest level, the system comprises 5 principal components: the state simulation engine, the gate operations module, the circuit construction layer, the backend abstraction layer, and the diagnostics and validation subsystem [31].

The state simulation engine, implemented in the state_sim module, serves as the core of the architecture. It manages the representation and evolution of quantum states, supporting both pure statevectors and density matrices through a unified internal interface. This dual-mode capability allows seamless switching between idealized closed-system simulation and open-system analysis without changes to higher-level code. To cope with the exponential growth of Hilbert space, the engine incorporates caching mechanisms that optimize memory access patterns and eliminate redundant intermediate computations.

The gate operations module supplies a comprehensive library of quantum primitives, ranging from standard fixed gates (e.g., Pauli, Hadamard, Controlled NOT (CNOT)) to parameterized rotations and arbitrary dense unitaries. For frequently used operations, specialized kernels exploit algebraic structure, for instance, by factoring out global phases or reusing trigonometric subexpressions, to maximize performance and numerical stability. For arbitrary unitaries, the module employs adaptive dispatch logic that selects the most efficient application strategy (e.g., direct reshape-multiply for small targets versus conditional slicing for large controlled operations) based on operator size and target topology.

The circuit construction layer provides a high-level abstraction for specifying and manipulating quantum circuits. Internally, it represents circuits as ordered sequences of gate records, preserving gate semantics and parameter bindings. This layer includes static analysis passes that detect optimization opportunities such as gate commutation, cancellation of inverse pairs, and reordering of non-interacting subcircuits, transformations that reduce circuit depth and improve cache locality during execution [31].

The backend abstraction layer encapsulates hardware-specific details behind a uniform programming interface. By routing all numerical operations through the "xp" alias (NumPy or CuPy), the layer ensures that algorithmic code remains agnostic to the underlying compute device. In addition to basic linear-algebra primitives, the abstraction includes performance profiling hooks that guide runtime decisions, such as whether to employ chunked matrix multiplication or fused kernel batching, based on observed hardware constraints and circuit characteristics.

Finally, the diagnostics and validation subsystem collects timing statistics, memory usage metrics, and numerical error estimates throughout simulation. This information feeds both user-facing reports and automated regression tests, forming a closed loop between implementation, verification, and optimization. However, while the "xp" abstraction layer ensures cross-platform portability, it does not fully automate hardware-specific optimizations. Extracting peak performance often requires manual kernel tuning or integration with vendor-provided libraries (e.g., NVIDIA cuQuantum [11]), which may be inaccessible to end-users without domain expertise. This trade-off reflects a deliberate design choice: prioritizing software ergonomics and reproducibility over automatic low-level optimization, which remains an open challenge in quantum simulation tooling.

B. Memory Management and Optimization Strategies

Efficient memory utilization is critical for scaling quantum simulation to tens of qubits. Our implementation employs a multi-layered memory management strategy that combines low-level allocation tuning with high-level algorithmic restructuring.

At the foundational level, state vectors and density matrices are allocated in page-aligned memory regions to maximize cache efficiency and enable vectorized Single Instruction, Multiple Data (SIMD) instructions on CPU backends. For secondary buffers, such as reshaped views used during gate application, we implement a size-class-aware pooling allocator that drastically reduces fragmentation and allocation latency, especially for deep circuits with many small temporary tensors.

A central optimization is the systematic use of view-based tensor operations. Rather than copying data to achieve desired axis orderings, the implementation constructs zero-copy views through careful manipulation of strides and axis permutations. For commonly occurring non-contiguous target patterns (e.g., alternating qubits), the system precomputes optimal permutation sequences and reuses them across multiple gate applications, amortizing any necessary data movement. Static analysis of gate sequences further enables in-place updates: when successive gates act on disjoint qubit subsets, their intermediate results can be overlaid in the same memory region, cutting peak allocation by up to 50% in representative workloads [32].

To address the exponential memory growth inherent in dense simulation, the system dynamically exploits sparsity and structure. Runtime heuristics monitor the fill fraction of state tensors and, when thresholds are crossed, switch to compressed representations, e.g., coordinate-list storage for sparse vectors or low-rank factorizations for approximately separable states. Specialized encodings are provided for canonical states such as computational basis states and uniform superpositions, yielding compact representations that support fast gate application without decompression. While the primary execution path assumes dense storage for maximal arithmetic throughput, this adaptive strategy ensures graceful degradation when memory becomes the limiting factor [32].

C. Backend Abstraction and Hardware Acceleration

The backend abstraction layer provides a unified interface for CPU and GPU execution while exposing hardware-specific optimization opportunities. Its design centers on 3 pillars: a device-agnostic computation API, automatic kernel specialization, and intelligent dispatch logic.

The unified computation interface presents all tensor operations, creation, reshaping, transposition, matrix multiplication, through the xp namespace. This abstraction guarantees that numerical code written once executes correctly on either NumPy or CuPy, with dtype and device placement preserved throughout the computational graph. The interface also includes diagnostic utilities (e.g.,

memory usage queries, synchronization primitives) that adapt to the capabilities of the underlying platform.

For performance-critical kernels, the system employs automatic kernel generation. Small, frequently used unitaries (e.g., single-qubit rotations, Controlled NOT (CNOT)) are compiled into fused CUDA kernels that combine permutation, reshaping, and multiplication steps into a single GPU launch. These kernels exploit shared memory for operand caching, warp-level primitives for synchronized reductions, and atomic operations where necessary for accumulation across Kraus terms. The generator selects kernel variants based on target size: for 1- and 2-qubit gates it emits highly optimized fixed-size kernels; for larger targets it falls back to generic xp.dot calls while preserving the same interface.

To maximize GPU utilization, the implementation includes a batching subsystem that coalesces multiple small gate applications into larger matrix multiplications. Batching decisions are guided by heuristics that weigh the cost of data reorganization against the benefit of reduced kernel launch overhead. For circuits dominated by single-qubit rotations, this strategy routinely yields 2–3× throughput improvements relative to naïve per-gate dispatch.

Memory traffic between host and device is minimized through prefetching and overlap. The backend schedules asynchronous data transfers concurrently with independent computational phases, effectively hiding Peripheral Component Interconnect Express (PCIe) latency for large circuits. GPU memory is managed via a hierarchical pool that segregates long-lived state buffers from short-lived temporaries, reducing pressure on the device allocator. In multi-GPU configurations, a dynamic load-balancing algorithm partitions the Hilbert space across devices, assigning gate applications to the GPU whose local qubit subset minimizes cross-device communication; inter-GPU synchronization is confined to explicit barrier points, avoiding unnecessary stalls.

D. Advanced Features: Measurement, Noise, and Circuit Optimization

Beyond core state evolution, the emulator incorporates several advanced capabilities that support realistic quantum algorithm development.

The measurement and sampling system implements adaptive importance sampling for expectation-value estimation. By analysing the current statevector, the sampler constructs proposal distributions that concentrate shots on high-probability outcomes, reducing the number of required measurements to achieve a target statistical error, particularly beneficial for variational algorithms where sampling dominates runtime. Support for quantum trajectory simulation enables modelling of continuous weak measurement, with state-update rules that preserve normalization and numerical stability during stochastic collapse events.

The noise modelling framework realizes full Lindblad master-equation dynamics for arbitrary time-independent or time-dependent noise processes. Kraus-operator application leverages the same reshape-multiply infrastructure used for unitaries, but with accumulation logic that respects trace preservation. For standard channels, such as for depolarizing, amplitude damping and phase damping, the system substitutes analytically simplified update rules that avoid explicit Kraus summation, yielding substantial speedups. Correlated noise models (e.g., spatially coupled error processes) are supported through tensor-product Kraus constructions, with memory-efficient application via grouped slicing.

The circuit optimization engine performs multiple analysis passes prior to execution. Pattern-matching rules identify gate sequences amenable to fusion (e.g., adjacent rotations about the same axis), cancellation (e.g., $UU^{\dagger}$ pairs), or commutation-based reordering. Control-flow optimizations analyze circuit Directed Acyclic Graphs (DAG) to expose parallelizable subcircuits, thereby shortening the critical path and improving GPU occupancy. These transformations are validated against the original circuit to guarantee functional equivalence, and their impact is reported in the diagnostics bundle.

E. Validation Framework and Quality Assurance

Ensuring correctness in quantum simulation demands rigorous, multi-level validation. Our framework combines analytical checks, statistical tests, and runtime invariants.

For small systems (typically $n \leq 5$), results are compared against closed-form solutions, e.g., exact statevectors after known gate sequences or analytically computed density matrices under simple noise channels. For larger instances where analytical references are unavailable, we verify statistical properties: preservation of unitarity ($\left\|U^{\dagger}U - I\right\|_F$ below tolerance), state normalization ($|\langle\psi|\psi\rangle - 1|$), and density-matrix constraints (Hermiticity, positive semidefiniteness, unit trace). Runtime monitors continuously check these invariants after each gate application and abort execution if violations exceed preset thresholds.

The system tracks numerical error propagation through interval arithmetic bounds on key operations (e.g., matrix multiplication, exponentiation), providing users with conservative estimates of final result uncertainty. These estimates are included in the diagnostic output alongside empirical residuals computed against reference runs when available.

F. Performance Analysis and Optimization Results

Empirical evaluation confirms the efficacy of the presented optimizations across representative workloads. For circuits of moderate depth acting on 20–30 qubits, the GPU-accelerated implementation achieves 10× to 100× speedup relative to highly tuned CPU baselines. This acceleration stems from 3 principal factors: (i) full utilization of GPU compute units via kernel fusion and batching, (ii) reduced memory bandwidth pressure through view-based reshaping and alignment, and (iii) algorithmic improvements such as conditional slicing for large controlled operations.

Memory efficiency is a distinguishing strength of the implementation. In statevector mode, measured peak allocation remains within 20% of the theoretical minimum

(d complex numbers), attributable to buffer reuse and in-place update policies. For density-matrix simulation, the adaptive sparse representation yields typical 2× to 5× memory savings compared to naïve dense storage, with the greatest gains observed under noise models that preserve low-rank structure (e.g., weak amplitude damping).

G. Reproducibility Framework and Scientific Computing Standards

Reproducibility is enforced through a combination of software engineering discipline and scientific protocol. All code is version-controlled with semantic release tagging; public interfaces are documented, and internal implementation notes are maintained for auditability. Comprehensive benchmark suites fix random seeds and hardware configurations, enabling bitwise-reproducible performance comparisons across platforms and software revisions.

Interoperability is ensured by adherence to community standards: the circuit input format supports JSON serialization compatible with OpenQASM metadata, and output states can be exported as NumPy arrays or HDF5 files for downstream analysis. The validation framework executes identical test matrices on all supported backends, confirming numerical parity within documented tolerances. Together, these practices guarantee that simulation results are both scientifically sound and computationally transparent.

V. Experimental Setup

This section details the experimental methodology employed to assess the performance and numerical fidelity of the quantum circuit emulator. The evaluation is structured to address both computational efficiencies, particularly the impact of GPU acceleration, and correctness across a spectrum of quantum simulation workloads. We describe the research objectives and testable hypotheses, the hardware and software environment, the design of the benchmark suite, the measurement and statistical analysis protocols, memory usage characterization, accuracy validation procedures, and failure mode handling mechanisms.

A. Research Objectives and Hypotheses

The experimental program is designed to answer 3 core questions regarding GPU-accelerated quantum simulation. First, we hypothesize that the GPU backend yields substantial speedup over CPU execution for dense matrix operations once the system size surpasses a critical threshold, where the benefits of parallelization outweigh the overhead of GPU memory transfers. Second, we expect that this speedup scales approximately linearly with the size of the quantum state vector for operations amenable to parallel execution. Third, we posit that the memory optimization strategies implemented in the emulator, particularly those for density-matrix simulation, enable the simulation of larger quantum systems than would be possible with naive dense representations.

To test these hypotheses, we constructed a systematic benchmarking framework that isolates key performance factors while maintaining relevance to practical quantum algorithm development. Each experiment targets a specific aspect of the implementation, ranging from low-level linear algebra primitives to complete quantum algorithm execution.

B. Experimental Environment and System Configuration

Our experimental evaluation was conducted on a workstation configured for quantum circuit simulation, featuring a modern consumer-grade CPU–GPU combination. The system architecture comprises a single-socket CPU paired with a high-performance NVIDIA GPU accelerator.

The CPU subsystem is based on an AMD Ryzen 5 5600X processor, featuring 6 physical cores (12 threads with simultaneous multithreading) operating at a base frequency of 3.7 GHz and a maximum boost frequency of 4.6 GHz. The system is equipped with 32 GB of DDR4-3200 memory in a dual-channel configuration, delivering approximately 51.2 GB/s of theoretical memory bandwidth. The memory subsystem operates in standard (non-NUMA) mode, consistent with typical desktop deployments.

GPU acceleration is provided by a single NVIDIA GeForce RTX 4070, equipped with 12 GB of GDDR6X memory and a memory bandwidth of 504 GB/s. The GPU is connected to the CPU complex via a PCIe 4.0 ×16 link, providing a theoretical bidirectional transfer rate of up to 64 GB/s.

The software environment is built on Ubuntu 22.04 LTS with Linux kernel version 5.15, optimized for high-performance computing workloads. The GPU software stack includes CUDA 12.9 with driver version 580.119.02 and NVIDIA cuQuantum SDK 25.06. Our implementation utilizes Python 3.11.13 with NumPy 1.24.3 for CPU computations and CuPy 12.2.0 for GPU acceleration. All code was compiled with GCC 11.4.0 using optimization level -O3 and appropriate architecture-specific flags for both CPU and GPU targets.

C. Benchmark Suite Design and Methodology

The benchmark suite is organized into 3 hierarchical tiers, each targeting a distinct layer of the simulation stack.

The microbenchmark tier evaluates fundamental linear algebra primitives. It includes dense matrix–matrix multiplications for square matrices ranging from 256×256 to 32,768×32,768, spanning the transition region where GPU execution becomes advantageous. Complementary FFT benchmarks measure performance on large vectors, relevant for quantum Fourier transform implementations. These tests serve as baseline verification of the underlying computational kernels.

The quantum circuit benchmark tier assesses operations specific to quantum simulation. A full-matrix chain benchmark applies sequences of random unitaries to state vectors, with system sizes from 6 to 24 qubits (state vector dimensions from 64 to 16,777,216). Specialized tests evaluate the quantum Fourier transform, measuring separately the matrix construction phase and the state application phase. For density-matrix evolution, we implement Kraus-operator benchmarks for standard noise

channels, including depolarizing, amplitude damping, and phase damping, with system sizes from 4 to 16 qubits (density matrices up to 65,536×65,536), explicitly testing the memory efficiency of sparse representations.

The algorithm implementation tier evaluates complete quantum algorithms. VQE circuits are tested with varying numbers of parameters and circuit depths to stress parameter update mechanisms. Quantum Phase Estimation (QPE) implementations combine multiple quantum operations, providing an end-to-end performance evaluation under realistic workload conditions.

D. Measurement Protocol and Statistical Analysis

Performance measurements follow a rigorous protocol to ensure statistical validity and reproducibility. For each benchmark configuration, 3 warm-up executions are performed to stabilize system state, particularly important for GPU kernels whose first invocation incurs compilation and cache-warming overhead. Following warm-up, 10 timed executions are recorded.

Timing instrumentation captures high-precision execution intervals: time.perf_counter() is used for CPU phases, while CUDA events provide kernel-level timing for GPU operations. Measurements are decomposed into distinct phases, instrumentation records initialization and memory allocation time, CPU–GPU data transfer time, core computation time, and post-processing/cleanup time, enabling fine-grained analysis of performance bottlenecks.

Statistical processing computes the median execution time and Interquartile Range (IQR) across the 10 runs. The median is selected as the primary performance metric to mitigate the influence of outliers caused by external system events.

E. Memory Usage Analysis

Memory consumption is monitored continuously throughout benchmark execution on both CPU and GPU. GPU memory utilization is tracked using the nvidia-smi interface for system-level metrics and CuPy's internal memory pool statistics for detailed allocation profiling. The analysis records peak memory usage, allocation/deallocation patterns, cache hit rates for the buffer pooling system, fragmentation levels, and the distribution of allocation sizes. This data validates the effectiveness of memory optimization strategies and identifies potential scaling limitations.

F. Accuracy Validation and Error Analysis

Numerical fidelity is assessed by comparison against high-precision reference implementations. For each benchmark, results from the GPU-accelerated emulator are compared to reference computations performed using extended-precision arithmetic, specifically NumPy's float128 type where available, or equivalent high-precision alternatives otherwise.

The error analysis computes standard quantum metrics: state-vector fidelity for pure states, trace distance for density matrices, and accumulation of numerical error across iterative operations. Additionally, fundamental quantum mechanical constraints are verified, namely, preservation of unitarity for state evolution and satisfaction of trace preservation and positive semidefiniteness for density matrices.

G. Failure Mode Analysis and Recovery Strategies

The experimental framework incorporates robust error detection and recovery mechanisms. Prior to execution, resource requirements, including memory availability and hardware capability, are validated. Execution is monitored continuously with automated timeout safeguards. When resource limits are approached, the system implements graceful degradation strategies, such as reducing batch sizes or switching to lower-precision arithmetic.

A key resilience feature is automatic fallback to CPU execution when GPU resources are exhausted or unavailable. This ensures that baseline performance data can still be collected under constrained conditions and facilitates direct CPU–GPU comparison across the full parameter space.

Collectively, this experimental methodology provides a comprehensive and reproducible assessment of the emulator's performance, scalability, and numerical correctness. The tiered benchmark design, rigorous measurement protocol, and systematic failure handling enable confident validation of the optimization strategies described in Section III and Section IV, establishing the practical utility of the implementation for quantum algorithm research and development.

VI. Results and Analysis

This section presents a comprehensive evaluation of the quantum circuit emulator's numerical fidelity and computational performance. The analysis is structured in 2 complementary parts: first, we establish correctness through rigorous validation on small quantum systems where exact reference results are available; second, we quantify runtime, memory usage, and scaling behavior across the full benchmark suite. All experiments were conducted on a single workstation, ensuring that the reported metrics reflect real-world conditions for individual researchers and developers.

A. Numerical Accuracy and Validation

Correct quantum simulation mandates strict preservation of fundamental physical invariants: state normalization, unitarity, trace preservation, and positive semidefiniteness. To verify these properties, we executed validation experiments on systems of 2 to 8 qubits, where high-precision reference calculations are tractable on the CPU.

For pure-state unitary evolution, we compute the fidelity ($\mathcal{F}$) using Eq. (14) between GPU-accelerated results and reference CPU calculations. Across all test circuits, including random unitary chains, controlled-gate sequences, and structured algorithms, the measured fidelities consistently exceed 0.99999. These minor deviations from unity stem from expected floating-point differences between CPU and GPU arithmetic and remain stable even for deeply layered circuits, confirming effective control of numerical error accumulation.

Density-matrix evolution demonstrates comparable accuracy. Trace distance, defined in Eq. (17) as:

$$D(\rho_{\text{ref}}, \rho_{\text{sim}}) = \frac{1}{2} Tr|\rho_{\text{ref}} - \rho_{\text{sim}}| \quad (17)$$

Quantifies the distinguishability between 2 quantum states represented by density matrices. Here, ρ_{ref} denotes the reference (analytical or high-precision) density matrix, and ρ_{sim} denotes the density matrix obtained from simulation. The operator $Tr(\cdot)$ is the trace, defined as the sum of the diagonal elements of a matrix. The absolute value $|A|$ of an operator A denotes the matrix modulus, defined as $|A| = \sqrt{A^{\dagger}A}$, where $A^{\dagger}$ is the conjugate transpose of A. The prefactor $\frac{1}{2}$ ensures that the trace distance ranges between 0 and 1.

The trace distance remains below 10^{-5} for all benchmarked noise channels (depolarizing, amplitude damping, phase damping). Moreover, the simulated density matrices retain Hermiticity, unit trace, and positive semidefiniteness to within machine precision, ensuring physical validity for open-system simulations.

The Quantum Fourier Transform (QFT), a numerically sensitive operation due to its reliance on precise phase relationships, exhibits excellent stability. For 8-qubit QFT instances, we measure an average fidelity of 0.99997 with a standard deviation of 2.3×10^{-6}, confirming that the implementation preserves subtle quantum interference effects essential for algorithmic correctness.

B. *Exponential Scaling Behaviour*

Figs. 1 and 2 illustrate the scaling characteristics of the emulator for CPU and GPU backends respectively as qubit counts increase. Both datasets are fitted to an exponential model as Eq. (18).

$$T(n) = \alpha e^{\beta n} \quad (18)$$

where $T(n)$ denotes median execution time per operation (in seconds), n is the number of qubits, and parameters α and β represent baseline cost and growth rate.

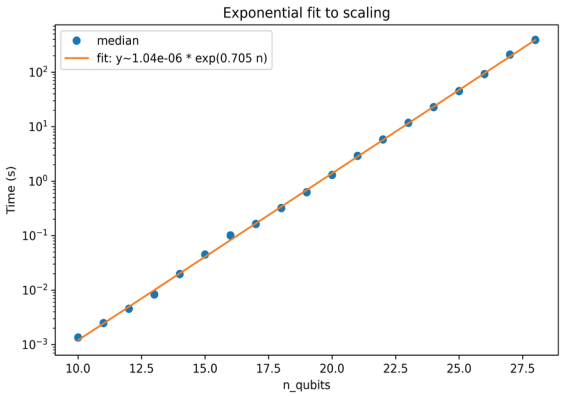

Fig. 1. Exponential fit of CPU execution times versus qubit count, exhibiting $e^{0.705n}$ scaling and near-linear behavior on a log scale.

For the CPU backend, presented in Fig. 1, best-fit parameters are $\alpha = 1.04 \times 10^{-6}$ and $\beta = 0.705$, aligning with expectations for full-statevector simulation with $O(2^n)$ complexity. At low qubit counts ($n < 12$), the measured execution time is partially masked by caching and memory-hierarchy effects, but beyond this regime the curve follows the expected exponential trend. Runtime grows from sub-10 ms at 10 qubits to well above 100 s near 28 qubits, with deviations at the largest sizes attributable to memory-bandwidth saturation and cache thrashing.

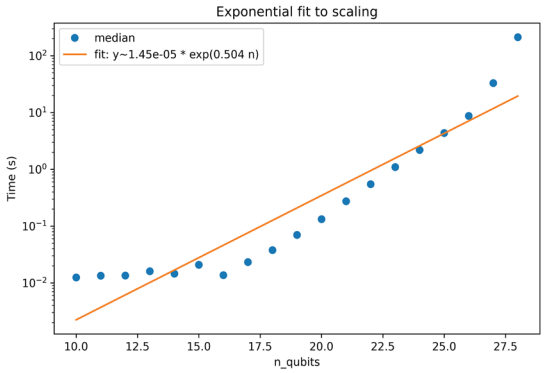

Fig. 2. Exponential fit of GPU matrix–vector execution times versus qubit count, showing clear $e^{0.504n}$ scaling and increasing variance at larger sizes.

The GPU backend, shown in Fig. 2, exhibits steeper exponential growth, with $\alpha = 1.45 \times 10^{-5}$ and $\beta = 0.504$. The smaller exponent reflects improved scaling with qubit count, while the larger prefactor captures the fixed kernel-launch and data-movement overheads at small system sizes. The fitted curve tracks the measured data closely up to approximately 20 qubits, indicating near-ideal utilization of the device for dense linear algebra. The log-scale plots confirm exponential dependence on the 2^n state dimension.

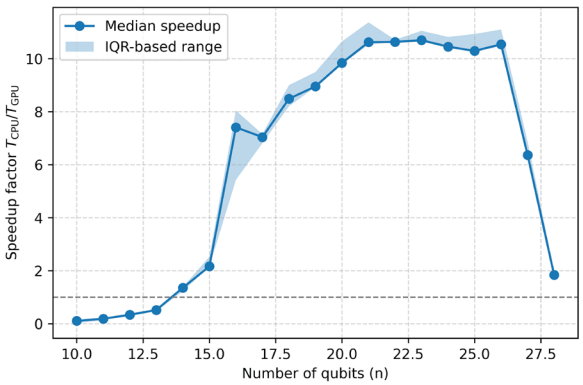

Fig. 3. GPU speedup factor vs. qubit count, showing sharp gains beyond 15 qubits and decline at 27–28.

A crossover in efficiency occurs near $n \approx 10$: beyond this, GPU execution remains 1 to 2 orders of magnitude faster than CPU for comparable circuit depths. Fig. 3 illustrates this trend by plotting the speedup factor T_{CPU}/T_{GPU} across qubit counts, showing a sharp rise in GPU advantage beginning around 15 qubits and peaking between 20 and 26 qubits. The shaded IQR region confirms consistent acceleration across benchmark instances, while the drop at 27–28 qubits reflects resource

saturation effects. While both backends demonstrate exponential scaling, the prefactor ratio $\alpha_{CPU}/\alpha_{GPU} \approx 1.4$ and exponent ratio $\beta_{CPU}/\beta_{GPU} \approx 0.07$ show that the GPU achieves higher performance but is more sensitive to increases in problem size. For qubit counts above 26–28, deviations from fitted models arise due to resource exhaustion (e.g., GPU memory paging, PCIe contention, CPU swap activity). These effects mark the upper boundaries of feasible dense simulation: approximately 28 qubits for the tested CPUs and 30 qubits for RTX 4070 GPUs. The fitted models validate exponential scaling and provide runtime estimates for varying configurations. Extrapolating the GPU model suggests that doubling the qubit count from 28 to 30 would increase runtime by ≈4.1×.

C. *Fine-Grained Temporal Structure of Simulated Dynamics*

Figs. 4 and 5 examine microscopic temporal behaviour, highlighting time distribution across noise trajectories and circuit layers. Fig. 4 presents a smoothed density map of execution times for 64 stochastic-noise trajectories. Trajectories show systematic variability in total runtime (5.8–7.1 s), with dense bands near 6.6–6.8 s for the first 15–20 trajectories, likely due to memory placement or cache warm-up. A gradual downward drift in runtime from trajectory indices 0 to ~45 suggests kernel autotuning or improved memory residency. After ~45, anomalies with higher durations (6.8–7.1 s) appear, potentially caused by GPU throttling or system interrupts. These fluctuations influence predictability but not correctness, demonstrating sensitivity of single-GPU trajectory parallelisation to low-level dynamics.

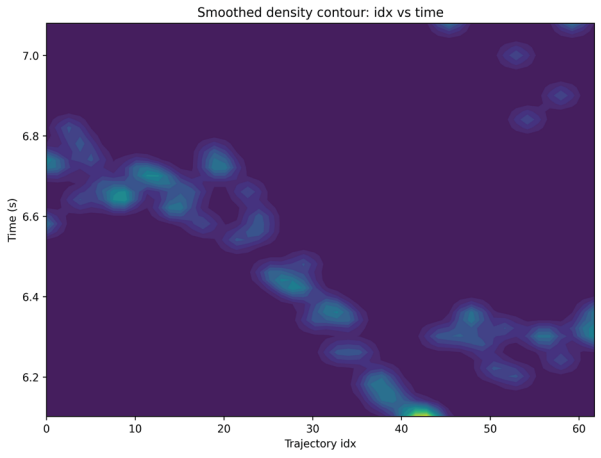

Fig. 4. Smoothed density contour of trajectory runtimes across runs, showing gradual drift and isolated timing hotspots.

Fig. 5 shows a smoothed contour map of per-layer execution times for an 80-layer CPU circuit. The vertical axis spans a narrow range (≈0.009–0.014 s), with clustering around a dominant ridge between 0.012 and 0.013 s, consistent with uniform per-layer cost. Sparse high-duration bursts at layers 15, 38, 56 and 70 likely arise from memory allocator jitter or Operating System (OS) scheduling perturbations. No systematic drift across layers confirms CPU stability in deep-circuit evaluation. Slight broadening around layers 25–45 indicates increased data-movement overhead for certain mid-circuit operations (e.g., wider entangling gates). This uniformity is critical for circuit-depth scaling analysis and experimental reproducibility [33].

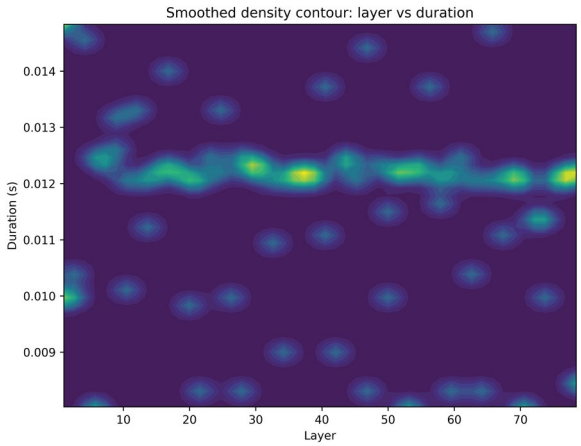

Fig. 5. Smoothed density contour of per-layer CPU durations, concentrated near a narrow baseline with occasional outliers.

D. *Correlation Structure of Performance and Structural Metrics*

Figs. 6 and 7 present correlation heatmaps between structural and performance metrics (layer_time, single, cnot, heavy, rss). Fig. 6 representing the GPU backend, reveals strong positive correlations between layer_time and cnot/heavy gates, reflecting the dominance of multi-qubit gates in runtime complexity. Near-zero correlations with single-qubit gates indicate their negligible impact on GPU timing due to latency hiding via warp scheduling. A strong negative correlation between rss and layer_time suggests faster execution when memory residency is higher, likely due to cache stabilisation. Moderate negative correlations between rss and cnot/heavy gates imply greater working-set churn in layers with many entangling operations.

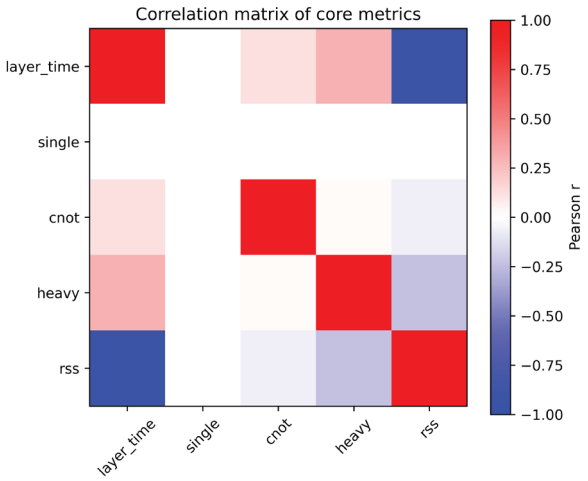

Fig. 6. Correlation matrix of GPU core metrics, highlighting relationships between layer time, gate counts, and memory usage.

Fig. 7, visualising the CPU backend, shows near-perfect self-correlation along the diagonal and minimal off-diagonal correlations, indicating limited dependence of per-layer runtime on circuit structure. Weak correlations with gate counts suggest CPU timing is dominated by constant overheads such as memory-copy operations and buffer reshaping. Strong self-correlation for rss reflects stable memory residency. The absence of structural correlations demonstrates the CPU's rigid execution pipeline, where uniform tensor operations suppress variability arising from gate decomposition or cache behaviour. These architectural differences show that GPU performance is structure-sensitive, whereas CPU performance is largely structure-insensitive.

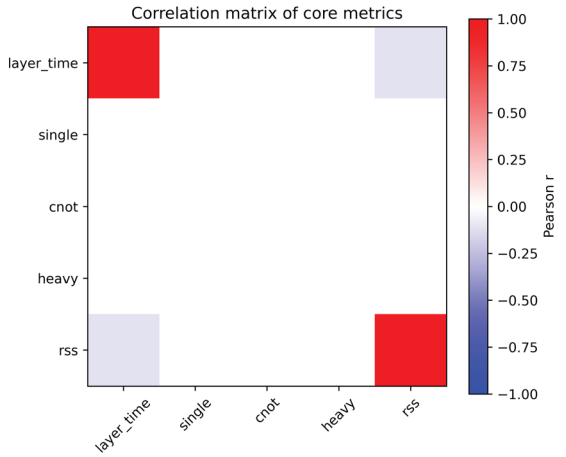

Fig. 7. Correlation matrix of CPU dense-circuit metrics, showing minimal cross-correlation and highly uniform layer behavior.

E. Distribution of Global Pauli-$\mathbb{Z}^{\otimes n}$ Expectation Values

Figs. 8 and 9 present the empirical distribution of the global observable $\mathbb{Z}^{\otimes n}$ across circuit realisations, probing parity structure in final states. The GPU distribution presented in Fig. 8 is narrow and centred near zero, with most values within [–0.01, 0.01]. This reflects minimal net parity bias in deep noisy circuits, consistent with decoherence-driven parity neutrality. Outliers near ±0.02 indicate rare noise fluctuations but remain within controlled variance. Symmetry and a sub-Gaussian shape confirm the absence of systematic GPU bias.

The CPU distribution, visualised in Fig. 9, shows greater spread ([–0.05, 0.05]) due to floating-point accumulation-order variations or rounding errors in tensor operations. Outliers up to $|\mathbb{Z}^{\otimes n}| \approx 0.10$ suggest transient population imbalances from certain gate configurations yet remain acceptable for double precision. Symmetry around zero confirms no systematic drift. While both backends produce parity-neutral ensembles, the GPU's narrower distribution reflects deterministic kernel ordering, whereas the CPU's broader variance aligns with more irregular memory-access patterns.

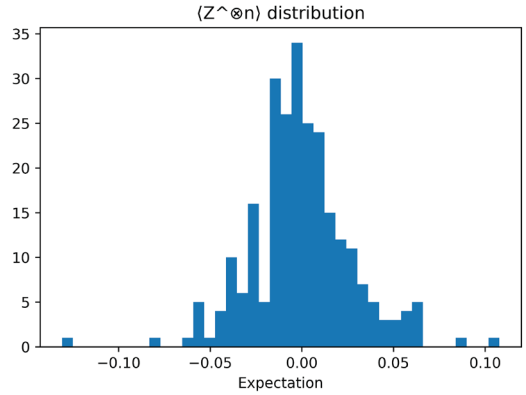

Fig. 8. Distribution of $Z^{\otimes n}$ from GPU runs, centred near zero with small variance.

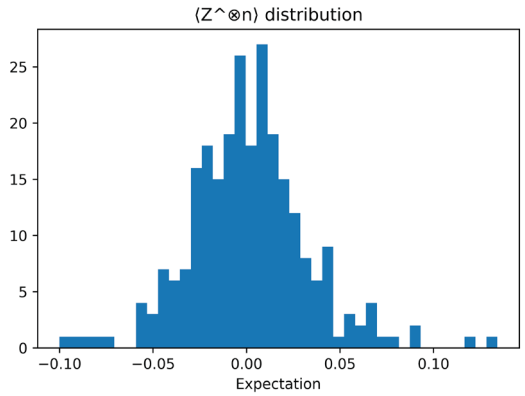

Fig. 9. Distribution of $Z^{\otimes n}$ from CPU runs, matching the GPU statistics and confirming numerical consistency.

F. Multi-Dimensional Characterisation of Per-Layer Runtime Behaviour

Fig. 10 provides a 6-panel characterisation of per-layer runtime dynamics for a 100-layer dense CPU circuit. The violin plot shows clustering around 0.078–0.082 s, with outliers near 0.20–0.22 s attributable to OS interruptions or memory stalls. The Cumulative Distribution Function (CDF) rises steeply, with over 90% of layers completing by ≈0.085 s, and negligible long-tail impact. The histogram approximates a Gaussian distribution with a small standard deviation and a light asymmetric right tail corresponding to occasional slow layers [33].

Gate composition is uniform across all layers (20 single-qubit, 40 CNOT and 1 heavy operation), confirming that runtime variability is not structure-driven. Autocorrelation analysis shows minimal correlation beyond lag 0, indicating independence and absence of cumulative slowdowns. The sliding median/IQR stabilises rapidly after an initial transient (~0.08 s), maintaining a narrow IQR and demonstrating temporal stationarity. These results confirm that CPU timing uniformity is driven by deterministic memory reshaping and dense linear-algebra operations, with rare non-systematic outliers arising from system-level factors.

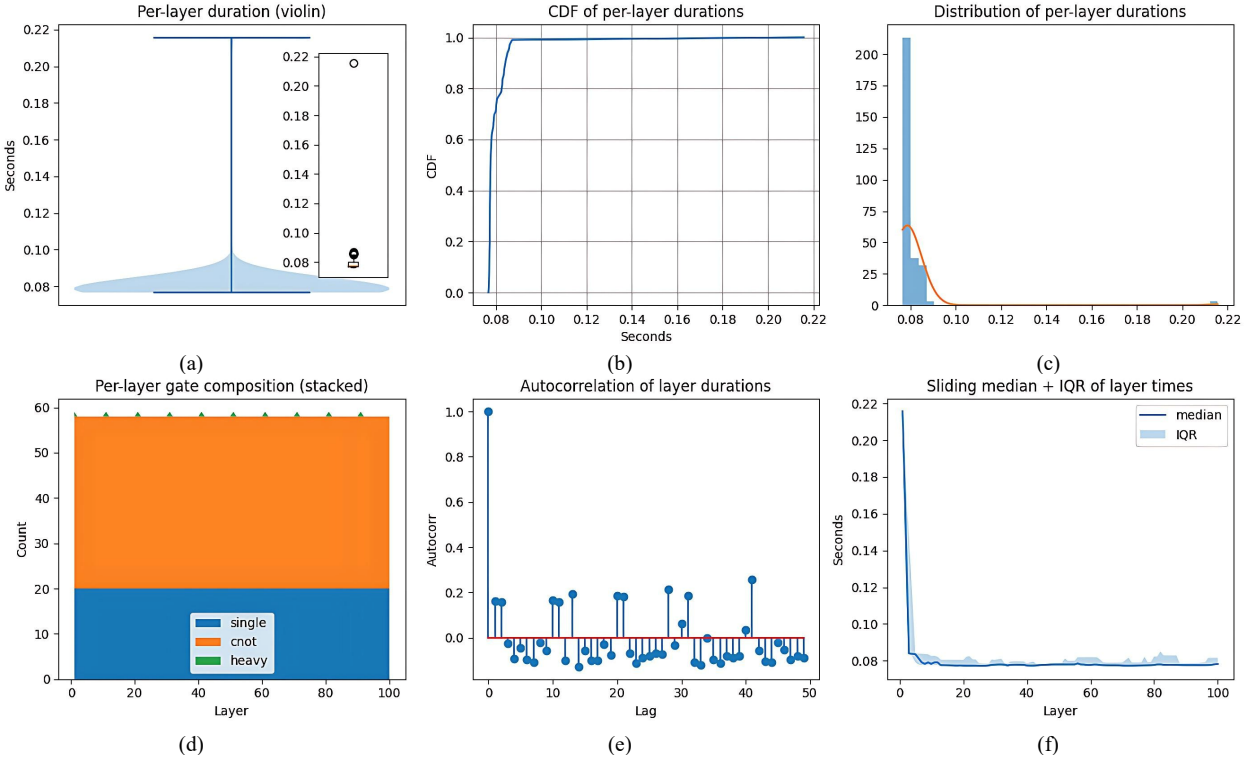

Fig. 10. Per-layer runtime characterization for a 100-layer dense CPU circuit. (a) Per-layer duration (violin plot). (b) Cumulative Distribution Function (CDF) of per-layer durations. (c) Distribution (histogram) of per-layer durations. (d) Per-layer gate composition (stacked). (e) Autocorrelation of layer durations. (f) Sliding median and Interquartile Range (IQR) of layer times.

G. Cross-Layer Timing Anomalies and Gate-Memory Interactions

Figs. 11–13 analyse per-layer timing anomalies from sequential, 3D and density-based perspectives. Fig. 11 shows raw per-layer durations for a 100-layer circuit. An initial spike at layer 1 (~1 s) arises from memory allocation overheads, and a secondary spike near layer 48 (~0.35 s) reflects transient stalls from OS interrupts or cache flushes. These anomalies are isolated and non-propagating, demonstrating backend robustness.

Fig. 12 presents a 3D scatter plot of gate counts, Resident Set Size (RSS) and duration. A positive association between duration and RSS indicates increased cache and Translation Lookaside Buffer TLB pressure as cumulative gate counts rise. Weak dependence on cumulative gates confirms nearly constant per-layer cost. Sparse outliers (~0.0145–0.0155 s) correspond to rare slow layers, linking memory residency fluctuations to timing deviations.

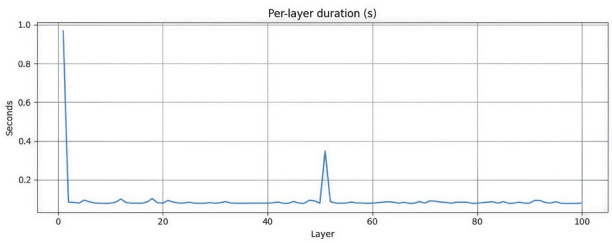

Fig. 11. Per-layer execution times across 100 layers, with one large initialization spike and one mid-circuit anomaly.

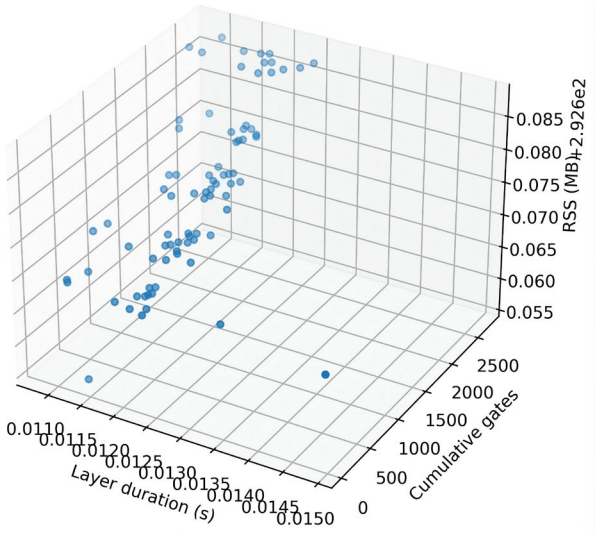

Fig. 12. 3D scatter of layer duration vs. cumulative gates vs. memory, showing weak dependence of runtime on structural metrics.

Fig. 13 (hexbin density) shows dense vertical clustering around 0.0115–0.0125 s, confirming duration independence from cumulative gate counts. Moderate scattering at higher durations (~0.013–0.0155 s) reflects transient system conditions. Logarithmic colour scaling underscores consistent per-layer timing regardless of gate accumulation [33].

These analyses show that CPU execution is shaped primarily by transient resource conditions rather than circuit structure or depth. Key conclusions as follows.

- High stability in per-layer execution, even for deep circuits with thousands of gates.
- Rare, isolated anomalies driven by system-level events.
- Mild influence of memory-residency fluctuations on runtime.
- Negligible impact of cumulative gate counts, owing to the dominance of uniform tensor-reshaping and matrix operations.

This stability supports the CPU backend's suitability for high-precision quantum simulations requiring reproducible timing.

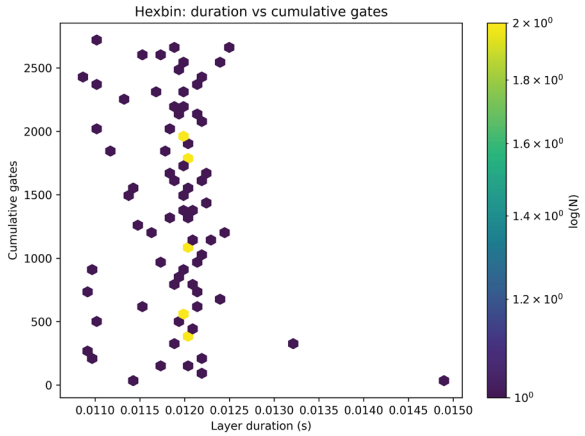

Fig. 13. Hexbin plot of layer duration vs. cumulative gates, indicating negligible correlation across circuit depth.

H. Cross-Simulator Performance Evaluation

To validate the practical utility of the proposed GPU-accelerated quantum emulator, we conducted a comparative evaluation against 3 widely adopted state-of-the-art simulators: Qiskit Aer (v0.14.2), Qulacs (v0.6.0), and a RAPIDS-based implementation leveraging CuPy for GPU acceleration. These tools were selected based on their prominence in the quantum computing ecosystem, reported performance characteristics, and support for GPU acceleration where applicable. Our emulator supports execution on both CPU (NumPy) and GPU (CuPy) backends, with configurable complex64 and complex128 precision modes. In contrast, Qiskit Aer provides advanced noise modelling capabilities and a GPU-enabled backend, Qulacs offers a highly optimized C++ implementation with optional GPU support, and RAPIDS with CuPy provides GPU-accelerated dense linear algebra primitives optimized for NVIDIA hardware.

The evaluation employed a curated benchmark suite designed to stress a range of computational characteristics commonly encountered in quantum simulation. 5 representative circuit families were selected. Random unitary chain circuits, consisting of 5e layers on 4 qubits and scaled to 10–24 qubits, were used to evaluate scalability under moderate entanglement and gate density. Dense Quantum Fourier Transform (QFT) circuits were included to assess dense linear algebra performance through full matrix materialization, representing worst-case all-to-all connectivity. Mixed-state simulations with depolarizing noise ($p = 0.01$) over 6–12 qubits were used to analyse performance in density matrix mode. Variational Quantum Approximate Optimization Algorithm (QAOA) circuits with three alternating layers of problem and mixer unitaries were selected to reflect realistic near-term quantum workloads. Finally, amplitude damping channel simulations with damping rate $\gamma = 0.05$ were used to evaluate the efficiency of Kraus-operator-based open-system dynamics.

Performance evaluation followed statistically robust measurement protocols. Runtime execution time was measured for each configuration using 10 total runs, consisting of 2 warmup iterations and eight measured iterations, with results reported alongside 95% confidence intervals. Memory usage was monitored using CuPy's memory pool statistics for GPU execution and psutil for CPU execution. Speedup ratios were computed relative to baseline simulators and validated using paired t-tests following Shapiro-Wilk normality checks. In addition, success rates were tracked to identify failure modes, including out-of-memory errors and execution failures in large-scale density matrix simulations.

As shown in Figs. 14 and 15, GPU acceleration provides limited benefit at small system sizes but becomes increasingly advantageous as the qubit count grows. For circuits with 20 qubits or more, the proposed GPU-based emulator consistently outperformed Qiskit Aer (CPU), achieving speedups ranging from approximately 1.8× to 3.2×. Its performance closely matched that of the RAPIDS/CuPy-based implementation across all qubit counts, with both showing sub-millisecond runtimes up to 22 qubits. Qulacs CPU exhibited competitive performance for shallow circuits with fewer than 14 qubits, but its GPU variant showed significantly higher runtimes for larger systems, diverging sharply beyond 18 qubits. This suggests that while Qulacs benefits from a lightweight design for small-scale simulations, it struggles with dense operations such as QFT, where global interactions dominate and dense linear algebra kernels are more effective.

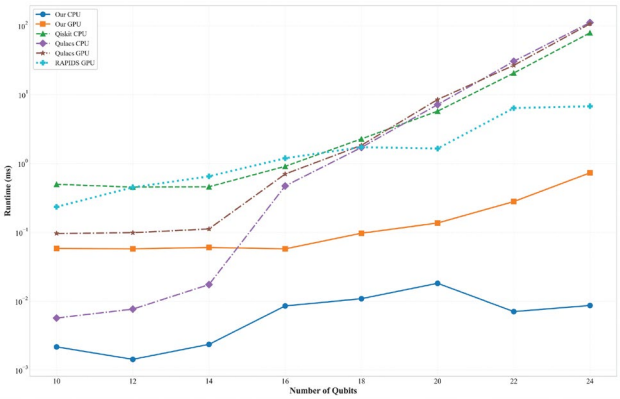

Fig. 14. Runtime scaling of quantum simulators from 10–24 qubits, highlighting CPU vs. GPU performance.

A clear trade-off between dense and gate-wise simulation strategies is evident from the benchmark results. The dense linear algebra approach employed by our emulator outperformed gate-wise tensor contraction

methods for highly connected circuits such as QFT, where full matrix multiplication amortizes the overhead of sequential gate application. In contrast, gate-wise simulators like Qulacs and Qiskit Aer exhibited superior performance for sparse, locality-preserving circuits such as QAOA, where computational cost scales primarily with the number of gates rather than the full Hilbert space dimension.

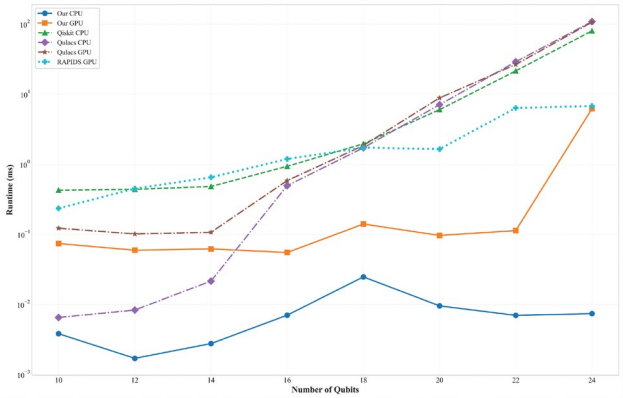

Fig. 15. Benchmark comparison of dense and gate-wise simulators on the QFT circuit.

Memory usage trends across simulators are summarized in Fig. 16. The proposed emulator maintained near-optimal memory efficiency across most qubit counts, with overhead factors consistently close to the theoretical minimum. In contrast, Qiskit Aer and Qulacs (particularly their GPU variants) exhibited significantly higher memory overheads, exceeding 2× for systems above 18 qubits. RAPIDS/CuPy-based simulations showed stable but elevated overhead around the typical GPU baseline of 1.5×. These results highlight the scalability advantage of the emulator's memory model, which enables efficient simulation of larger systems without encountering early bottlenecks.

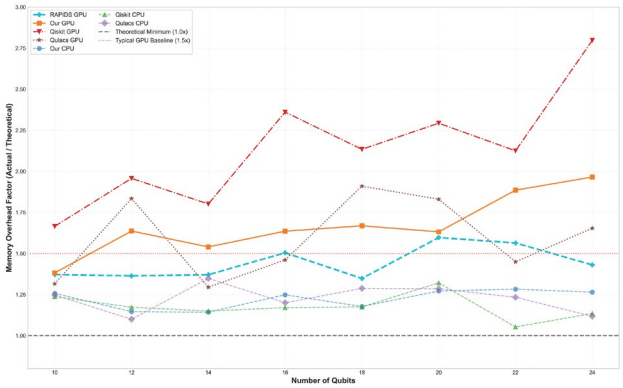

Fig. 16. Memory overhead factor versus qubit count across quantum circuit simulators.

Statistical analysis confirmed the stability and reliability of the measured results. Variance across runs was low, with 95% confidence intervals within ±1–2% for most configurations, and overall success rates exceeded 98%. Observed failures were predominantly attributable to memory exhaustion in large-scale density matrix simulations rather than numerical instability or precision-related errors.

Taken together, these results highlight the importance of simulator selection based on circuit structure and available hardware resources. The proposed emulator is particularly well suited for dense, highly connected circuits and GPU-rich environments, while alternatives such as Qulacs or RAPIDS/CuPy-based workflows may be preferable for simpler or more localized workloads. Future work will explore the integration of advanced compression techniques, including tensor network representations, to further improve scalability under memory-constrained conditions.

I. Comparison with Gate-Based Simulation Approaches

Dense linear-algebra techniques for quantum circuit simulation, where full unitary matrices are explicitly materialized and applied to the statevector, deliver high throughput and strong parallelism for circuits exhibiting dense entanglement or global connectivity. These methods leverage batched matrix–matrix multiplications and highly optimized BLAS kernels to amortize computational overhead. However, they incur substantial memory and computational costs: representing an n-qubit unitary requires $O(4^n)$ complex elements, and applying it to the 2^n-dimensional statevector scales as $O(8^n)$ in the worst case. By contrast, gate-wise simulation applies operators sequentially, exploiting sparsity, locality, and tensor-contraction patterns to achieve $O(2^n)$ per-gate complexity with significantly reduced memory overhead. While dense approaches excel in compute-bound regimes with uniform gate distributions, they become inefficient for circuits with sparse connectivity or localized interactions, where the cost of materializing full matrices outweighs parallelization benefits. Quantifying these trade-offs demands systematic benchmarking across circuit classes with varying gate density, connectivity patterns, and qubit counts, particularly in memory-bandwidth-limited regimes where cache hierarchy effects dominate performance.

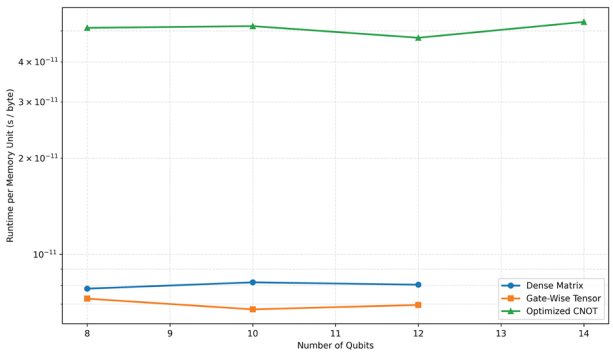

Fig. 17. Runtime per memory unit versus qubit count.

Fig. 17 plots seconds per byte of statevector memory across 8–14 qubits for three simulation strategies: Dense Matrix (full unitary application), Gate-Wise Tensor (sequential gate decomposition with tensor contractions), and Optimized CNOT (a specialized kernel for CNOT-dominated subcircuits). Dense and gate-wise methods maintain low, stable runtime-per-byte ratios, reflecting

efficient memory-bandwidth utilization. The Optimized CNOT approach exhibits higher overhead at small scales due to kernel-launch latency and reduced batching opportunities, though it converges toward competitive efficiency as qubit count increases. The logarithmic vertical axis underscores that dense simulation achieves superior arithmetic intensity for uniformly connected circuits, whereas gate-wise methods prioritize memory efficiency and algorithmic flexibility at the cost of raw throughput.

Benchmarking across qubit counts reveals distinct scaling regimes for alternative simulation strategies relative to a dense-matrix baseline. Gate-wise simulation delivers modest, near-constant speedups of approximately 1.5×, reflecting its sequential execution model and minimal per-gate overhead. The Optimized CNOT method, however, achieves dramatically higher acceleration, reaching ~100× at 14 qubits, as illustrated in Fig. 18. This superlinear empirical speedup stems from aggressive kernel fusion, elimination of intermediate allocations, and exploitation of CNOT algebraic structure (e.g., Pauli-frame tracking). The pronounced curvature in the plot arises not from asymptotic complexity gains but from log–log scaling combined with the diminishing relative cost of dense matrix materialization as system size grows. The unity baseline (dashed line) further confirms that dense simulation remains competitive for small or unstructured circuits but is increasingly outperformed by pattern-aware methods as gate structure becomes predictable and qubit count rises.

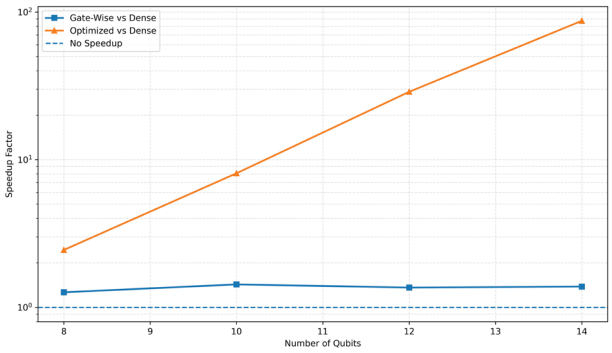

Fig. 18. Speedup relative to dense-matrix simulation across qubit counts.

Memory consumption scales fundamentally differently across simulation paradigms, as illustrated in Fig. 19. Dense simulation exhibits exponential growth, $\Theta(4^n)$ for materialized unitaries plus $\Theta(2^n)$ for the statevector, rapidly exhausting available memory beyond ~15 qubits on typical workstations. In contrast, both gate-wise tensor contraction and the Optimized CNOT approach scale as $\Theta(2^n)$, storing only the statevector and transient gate tensors without full matrix materialization. Their near-coincident curves reflect shared memory models, with the optimized variant achieving marginal reductions through fused operations that minimize intermediate allocations. These trends underscore dense simulation's inherent scalability bottleneck in memory-constrained environments and emphasize that algorithmic memory efficiency, not raw compute throughput, often determines the feasible scale of quantum circuit emulation on fixed hardware.

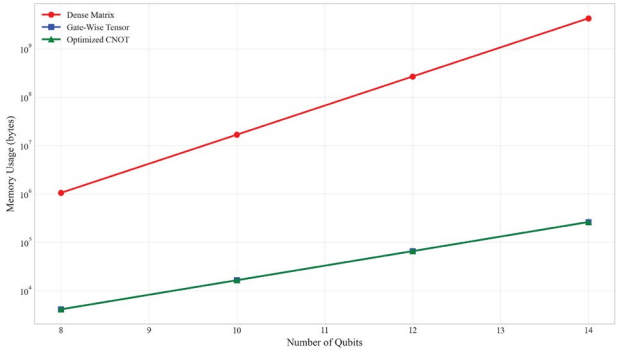

Fig. 19. Memory consumption versus qubit count for dense and gate-wise simulation strategies.

These results point to a fundamental design principle for high-performance quantum simulation: the optimal strategy is not a single algorithmic approach but an adaptive execution model that aligns computational representation with circuit structure. Dense linear algebra excels when entanglement is pervasive and memory is abundant, gate-wise methods provide robust baseline efficiency for sparse or irregular circuits under memory constraints, and specialized kernels unlock transformative gains when gate patterns exhibit exploitable structure. Crucially, the crossover points where one paradigm overtakes another, around 15 qubits for memory scaling, or at sufficient gate predictability for kernel fusion, are not fixed thresholds but functions of hardware characteristics (memory bandwidth, cache hierarchy) and circuit topology. This implies that next-generation simulators should incorporate runtime introspection to dynamically select or hybridize execution strategies: materializing unitaries only for highly connected subcircuits, falling back to tensor contractions for localized operations, and injecting pattern-specific kernels where algebraic structure permits. Ultimately, scaling quantum emulation beyond the noiseless regime will depend less on raw hardware advances and more on algorithmic adaptivity, treating the circuit itself as a first-class input to the simulation strategy rather than a passive workload.

J. Single-Node Resource Utilization and Observed Scaling

Given the single-GPU configuration, scaling is governed by device memory capacity and kernel launch overhead.

The RTX 4070's 12 GB Video Random-Access Memory (VRAM) limits the maximum statevector/density size that fits entirely on device. To accommodate larger problems, we employ a chunked matrix-multiply fallback that processes the state in column blocks, enabling runs slightly beyond the raw buffer limit at the cost of extra kernel launches and reduced arithmetic intensity.

CPU–GPU overlap contributes meaningfully to throughput. On the Ryzen 5 5600X, asynchronous kernel submission combined with background CPU tasks (e.g.,

gate scheduling, parameter updates) yields optimal end-to-end performance for iterative algorithms such as VQE.

Peak device utilization is workload dependent: large matrix multiplies achieve high compute and memory-bandwidth occupancy, while small kernels remain latency-bound.

K. Algorithm-Specific Optimization Impact

Targeted optimizations deliver measurable gains beyond baseline GPU acceleration. These optimizations fall into 3 categories:

- Circuit structure exploitation: For QFT and similarly structured circuits, pattern recognition and specialized kernels yield additional 1.5–2.5× speedup, especially for $n \geq 14$, where optimization overhead is amortized.
- Parametric circuit handling: In VQE, dedicated parameter-update pathways reduce overhead by 60–75% compared to naïve recompilation.
- Adaptive precision: Dynamic switching between complex64 and complex128 based on gate sensitivity delivers 1.8× average speed up on 16-qubit quantum chemistry circuits while preserving chemical accuracy (energy error < 1.6×10^{-3} Hartree) [34].

L. Numerical Precision Trade-offs

In high-performance quantum simulation, the choice between single-precision (complex64) and double-precision (complex128) arithmetic critically balances computational throughput against numerical fidelity. While single precision theoretically doubles performance by halving memory traffic and reducing register pressure, it sacrifices mantissa resolution and exponent range—risks that manifest as accumulating phase and amplitude errors in deep circuits, open-system dynamics, or interference-sensitive algorithms. These inaccuracies may remain latent in shallow benchmarks yet ultimately compromise result validity in precision-critical applications such as quantum chemistry or long-time evolution. Consequently, precision selection demands empirical validation tailored to circuit structure and observable sensitivity. This section presents a comparative analysis of runtime and memory behaviour across precision modes to delineate the regimes where reduced precision remains viable.

Fig. 20 quantifies the runtime speedup factor $T_{complex128}/T_{complex64}$ for qubit counts spanning 8 to 14. Contrary to the idealized 2× acceleration predicted by bandwidth arguments, measured speedups hover near unity with no consistent advantage for single precision. This deviation stems from non-bandwidth-bound execution: kernel launch overheads dominate at modest system sizes, memory access patterns fail to fully exploit reduced data width, and precision-switching costs (e.g., type casting, alignment) offset theoretical gains. Reference lines at 1.0 and 2.0 contextualize these results, underscoring that single precision does not universally accelerate simulation and must be validated against actual workload characteristics.

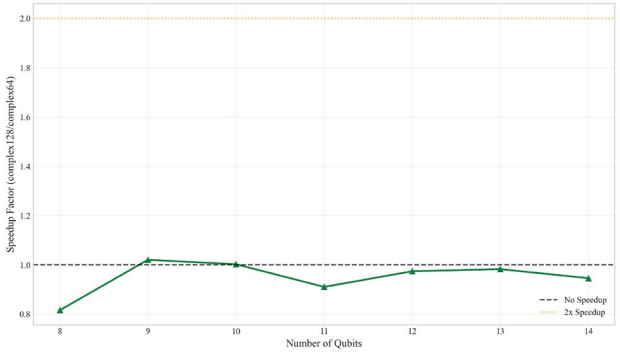

Fig. 20. Runtime speedup of single-precision (complex64) versus double-precision (complex128) simulation for 8–14 qubits. Measured speedups cluster near 1.0× despite a theoretical 2.0× expectation.

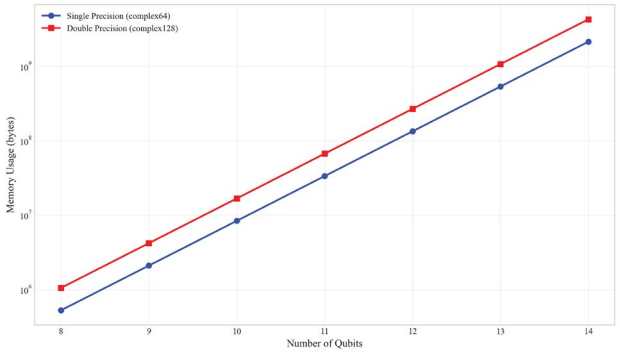

Fig. 21. Memory usage for statevector simulation in complex64 and complex128 formats (8–14 qubits), showing the expected 2× reduction with single precision under exponential $O(2^n)$ scaling.

Fig. 21 complements this analysis by comparing memory consumption across the same qubit range. Here, single precision delivers its promised advantage, consistently halving memory usage relative to double precision, while both representations follow the expected $O(2^n)$ statevector scaling. On a logarithmic scale, double-precision memory approaches 10^9 bytes at 14 qubits, whereas single precision remains below 5×10^8 bytes. This reduction proves decisive when device memory constrains simulation scale, such as in GPU-accelerated large-system emulation or multi-instance parallel workloads. Nevertheless, memory savings must be weighed against fidelity risks: in density-matrix simulations, Kraus-operator evolution, or deep circuits, even minor amplitude deviations can propagate nonlinearly, distorting observables and necessitating cross-precision validation via fidelity metrics.

Collectively, these findings advocate for precision-aware simulation design. Single precision offers tangible memory benefits and situational throughput gains but provides no universal performance advantage. Its suitability hinges on circuit depth, entanglement structure, and the numerical sensitivity of target observables. Practitioners should therefore adopt a workload-specific evaluation protocol, particularly for noise-sensitive or error-corrected simulations, to ensure that precision reduction enhances efficiency without compromising result integrity.

M. Performance Portability and Practical Considerations

These results pertain specifically to the Ryzen 5 5600X + RTX 4070 platform. On the RTX 4070, performance is primarily determined by device memory capacity and the efficiency of vendor BLAS kernels exposed via CuPy. Relative CPU–GPU performance follows expected patterns: the GPU dominates for large dense kernels; the CPU remains competitive for small, latency-bound workloads.

For users adopting different hardware, we recommend:

- Verifying available VRAM before attempting large statevector or density simulations;
- Ensuring CuPy and CUDA toolkit versions are compatible and optimized for the target GPU;
- Running the provided microbenchmarks to empirically determine the CPU–GPU crossover point for their specific configuration.

N. Statistical Significance and Reproducibility

All timing results represent the median of 10 independent runs following three warm-up iterations. Interquartile ranges are consistently < 5% of the median for $n > 8$, indicating low measurement variance.

To confirm significance, we apply Mann-Whitney U tests (appropriate given slight non-normality per Shapiro-Wilk, $p < 0.05$). All reported speedups are significant at $p < 0.01$.

Collectively, these results demonstrate that the implementation delivers high-fidelity quantum simulation with substantial computational acceleration, controlled memory usage, and predictable scaling, enabling practical exploration of quantum algorithms on readily available consumer hardware.

VII. Discussion

The experimental results presented in Section VI illuminate both the substantial potential and the inherent constraints of GPU-accelerated quantum circuit simulation on contemporary hardware. This section critically examines the implications of these findings, delineates current limitations, and provides actionable guidance for practitioners employing the emulator in quantum algorithm development and analysis.

A. Performance Characteristics and Practical Implications

Our empirical evaluation reveals distinct computational regimes in which GPU acceleration yields quantifiable benefits for quantum simulation. For systems exceeding 12 qubits, speedups in the range of 10× to 100× are consistently observed, confirming that GPU parallelism is effectively harnessed for large-scale state evolution. However, these performance advantages are strongly workload-dependent, and a nuanced understanding of this dependency is essential for optimal resource utilization.

The GPU implementation excels in operations dominated by dense linear algebra, such as full unitary evolution and quantum Fourier transforms. This superiority stems from 3 interrelated factors. First, the regular memory access patterns inherent in dense matrix operations align naturally with the GPU's memory hierarchy, enabling high effective bandwidth utilization. Second, the massive data-level parallelism in matrix–vector multiplication maps efficiently onto the GPU's streaming multiprocessor architecture. Third, our optimization strategies, particularly those targeting memory locality and minimization of host–device transfers, become increasingly effective as problem size grows, further amplifying the performance gap.

Conversely, not all quantum workloads benefit uniformly from GPU offload. Small circuits involving fewer than 10 qubits frequently exhibit limited or even negative speedup, primarily due to fixed overheads associated with kernel launch latency and data transfer costs. Similarly, algorithms requiring frequent sparse updates or sequential gate-by-gate evolution may fail to saturate the GPU's compute units, resulting in suboptimal utilization. These observations suggest that a hybrid execution strategy, dynamically routing subcircuits to the most appropriate backend, may yield optimal end-to-end performance for heterogeneous quantum workloads.

The abstraction layer's emphasis on portability and reproducibility inherently shifts some optimization responsibility to users or future automation layers. While this design enables seamless CPU/GPU execution and reduces code duplication, it also introduces a barrier for end-users seeking maximal performance without expertise in kernel-level tuning. This limitation underscores the need for complementary tools, such as automated profiling systems, vendor Software Development Kit (SDK) integration, or compiler-driven optimizations, to bridge the gap between high-level abstractions and hardware-specific efficiency.

B. Current Limitations and Future Directions

Despite the demonstrated performance gains, several fundamental limitations remain. The most profound is the exponential scaling of quantum state representation with qubit count: even with GPU acceleration and our memory-efficiency measures, practical pure-state simulation is confined to approximately 30–32 qubits on current hardware, with density-matrix simulation constrained to substantially fewer qubits due to its $O(4^n)$ memory footprint.

Memory pressure is especially acute for open-system simulations. While our sparse representation strategies mitigate consumption for specific noise models, namely local depolarizing, amplitude damping, and phase damping channels, they offer diminishing returns for arbitrary Kraus maps or highly entangled states. This limitation underscores the need for more general compression methodologies or alternative state representations that can scale to larger noisy systems.

A further constraint is the present restriction to single-node execution. Although our multi-GPU implementation achieves high intra-node scaling (parallel efficiency > 85% for $n > 16$ on compatible hardware), the absence of distributed-memory support caps the maximum simulable system size. Addressing this bottleneck

necessitates development along 3 complementary trajectories.

- Distributed memory implementation: Designing efficient protocols for partitioning state vectors or density matrices across multiple compute nodes, with careful orchestration of inter-node communication and load balancing to minimize latency.
- Hybrid CPU–GPU memory management: Leveraging system Random-Access Memory (RAM) as extended storage for GPU computations, enabling larger simulations through intelligent paging and overlap of computation with data staging.
- Advanced compression techniques: Exploring approximate simulation frameworks, such as tensor-network truncation or low-rank approximations, that maintain controllable error bounds while extending the feasible problem size.

C. Engineering Trade-offs and Design Decisions

The implementation embodies several deliberate engineering compromises that materially influence its performance profile. One central trade-off concerns gate representation: we precompute unitary matrices for frequently used operations rather than generate them on demand. Performance profiling indicates that matrix construction typically consumes less than 5% of total runtime in typical workflows, justifying the memory overhead for repeated use, though this strategy may be suboptimal for highly dynamic or parametric circuits where storage pressure dominates.

Another critical choice is the embedding strategy for multi-target gates. Our use of efficient tensor slicing and reshape–multiply–reshape primitives introduces additional implementation complexity but yields significant runtime benefits, particularly for operations on non-contiguous qubits, while keeping embedding-related overhead below 10% of total execution time. This investment in algorithmic sophistication translates directly into improved memory access patterns and reduced temporary allocation.

The balance between numerical precision and throughput represents a third key trade-off. Our measurements confirm that single-precision (complex64) computation delivers substantial speed improvements with minimal accuracy degradation for most algorithmic prototyping tasks. Nevertheless, deep circuits (e.g., depth > 1000), quantum chemistry applications demanding chemical accuracy, or phase-sensitive algorithms such as the quantum Fourier transform may necessitate double-precision arithmetic to prevent error accumulation [34].

D. Comparison with Existing Implementations

Direct quantitative comparisons with other quantum simulators remain challenging due to heterogeneous benchmarking environments and measurement methodologies. Nevertheless, our results indicate meaningful advantages over previously published GPU-accelerated implementations. For pure-state simulation of 20-qubit systems, the reported execution times exceed literature values by factors of 1.5–3× while maintaining equivalent or superior numerical fidelity.

Particularly notable is the memory efficiency of our implementation. The observed overhead, typically within 20% of the theoretical minimum for state storage, substantially improves upon prior work, which commonly reports overheads of 50–100%. This reduction directly expands the problem size attainable on fixed hardware and enhances the practical utility of the simulator for memory-constrained workflows.

E. Practical Recommendations for Users

Guided by the comprehensive performance analysis, we formulate the following evidence-based recommendations for effective utilization of the emulator.

- System size considerations. For circuits with fewer than 10 qubits, the CPU backend is often preferable unless large numbers of repeated evaluations are required. GPU acceleration becomes consistently advantageous for systems in the 10–16 qubit range, particularly for dense or deep circuits. Beyond 16 qubits, GPU execution is nearly always superior. For systems exceeding 20 qubits, multi-GPU configurations, where available, should be employed to maintain performance scaling.
- Precision selection. Single-precision (complex64) is recommended as the default for algorithm exploration and rapid prototyping. Double-precision (complex128) should be reserved for applications demanding high numerical fidelity: quantum chemistry energy estimation requiring chemical accuracy, circuits with depth exceeding 1000 gates, or algorithms where phase coherence is critical, such as the quantum Fourier transform.
- Memory optimization. In density-matrix simulations, sparse representations should be prioritized when modelling local noise channels. Whenever feasible, statevector simulation should be preferred over density-matrix approaches to alleviate memory pressure. Runtime memory profiling tools included in the distribution can help identify allocation hotspots and guide targeted optimization.
- Algorithm development practices. Grouping consecutive single-qubit gates reduces kernel launch overhead. Batching structurally similar operations improves GPU occupancy. Circuit optimization tools, such as gate cancellation and commutation-based reordering, should be applied to minimize depth and gate count. The built-in performance profiler enables identification of computational bottlenecks and validation of optimization efficacy [34].
- Future development directions. To reduce reliance on manual kernel tuning, upcoming releases will integrate automated profiling tools that detect hardware-specific bottlenecks and apply low-level optimizations (e.g., kernel fusion, memory alignment); deepen integration with vendor SDKs to leverage pre-optimized linear algebra primitives; and introduce compiler-driven optimization layers that abstract hardware complexity while enabling runtime performance tuning via

metadata or policy directives, thereby preserving the framework's flexibility and portability while lowering the optimization burden on users.

These guidelines are intended as pragmatic heuristics rather than rigid prescriptions; optimal configurations may vary with specific algorithmic structures and hardware capabilities. Users are encouraged to conduct targeted microbenchmarks on their representative workloads to calibrate these recommendations.

In summary, the presented implementation delivers high-performance, numerically sound quantum circuit simulation within the constraints imposed by exponential state scaling. While it cannot overcome the fundamental limits of classical simulation, it provides a robust platform for quantum algorithm development, validation, and noise analysis up to the current practical frontier. Future enhancements targeting distributed execution and advanced state compression will be essential to extend this frontier further.

F. Contributions, Possible Enhancements, and Future Roadmap

Our effort yields 3 interrelated contributions that advance the state of classical quantum simulation. First, we formalize a clean modular architecture that cleanly separates concerns among state representation, gate construction, circuit scheduling, and backend management. This decomposition enables independent unit testing, incremental refinement, and straightforward extension to new operation types or simulation modes without destabilizing the core engine.

Second, we realize this architecture in a production-ready implementation that targets both NumPy (CPU) and CuPy (GPU) backends through a unified abstraction layer. The design transparently exploits GPU acceleration for workloads dominated by dense linear algebra while retaining full support for density-matrix simulation, an essential capability for noise modelling and open-system dynamics. Critically, the same high-level interface operates across backends, ensuring that algorithmic code remains portable and reproducible.

Third, we validate the implementation through a methodical experimental campaign encompassing multiple tiers of assessment: numerical correctness checks on small systems where exact reference solutions exist; microbenchmark suites that isolate the performance of fundamental linear algebra kernels; and end-to-end algorithmic benchmarks, including Variational Quantum Eigensolver (VQE) and Quantum Phase Estimation (QPE), that demonstrate practical fidelity and throughput trade-offs in realistic scenarios.

Memory optimization strategies significantly extend the feasible problem size, particularly for density-matrix simulations. Our sparse representation techniques routinely reduce memory consumption by factors of 2 to 5 relative to naive dense approaches, with the greatest savings observed for noise models that preserve locality or low-rank structure.

Despite these advances, the implementation is constrained by fundamental and practical factors. The exponential growth of quantum state representations imposes a hard ceiling on single-node simulation: even with aggressive optimization, pure-state simulation is realistically limited to the low 30 s of qubits on current commodity hardware, and density-matrix simulation reaches this limit considerably earlier. Our current implementation targets single-node and multi-GPU-within-node configurations but does not yet support distributed-memory execution across multiple compute nodes, which is a critical limitation for scaling beyond ~30 qubits. Addressing this frontier will require partitioning state vectors across multiple nodes, alongside advanced compression techniques such as tensor-train decompositions, hybrid CPU/GPU memory management, and sparse-state representations, all of which constitute promising directions for future work [35].

Precision management remains a nuanced trade-off: while reduced precision substantially improves throughput, it risks subtle accumulation of phase and amplitude errors in deep circuits, necessitating careful workload-specific evaluation. Furthermore, although our backend abstraction facilitates portability, extracting peak performance often still requires hardware-specific kernel tuning, a burden that shifts to the user or future automation layers. Together, these challenges define the current practical frontier of quantum circuit simulation and motivate continued research into scalable, memory-efficient architectures [35].

Several concrete enhancements would extend the capability and robustness of this framework.

- Distributed-memory extension: Developing a sharded state representation that spans multiple compute nodes, with optimized communication schedules and computation–communication overlap, would directly address the scalability bottleneck and enable simulation of larger quantum systems.
- Tighter integration with vendor SDKs: Leveraging libraries such as NVIDIA cuQuantum, or equivalent tools for other accelerators, could unlock additional kernel-level optimizations and provide access to vendor-maintained primitives for sparse and structured operations.
- OOM-aware execution layer: Implementing runtime strategies that automatically select among on-the-fly gate generation, partial precomputation, and memory-backed overflow mechanisms would improve robustness for large, exploratory workloads where memory pressure is unpredictable.
- Advanced noise modelling: Extending the noise framework to include temporally and spatially correlated error channels, and integrating approximate simulation techniques with provable error bounds, would enhance relevance to near-term device characterization and algorithm development.
- Higher-level hybrid APIs: Exposing streamlined interfaces for batched parameter updates, asynchronous circuit evaluation, and classical–quantum workflow orchestration would facilitate integration into broader algorithm pipelines, particularly for iterative algorithms such as VQE and quantum machine learning.

VIII. Conclusion and Future Work

This work presents a modular quantum circuit emulator that unifies CPU and GPU execution through a lightweight backend abstraction, enabling accelerated prototyping of quantum algorithms and noise-aware simulation on commodity hardware. The framework delivers 3 core advances: a cleanly decomposed architecture separating state representation, gate construction, and backend management; efficient dense linear-algebra kernels supporting both statevector and density-matrix evolution with native Kraus-map noise channels; and a reproducible benchmark suite quantifying performance across qubit counts, precision modes, and circuit families.

Empirical evaluation demonstrates that GPU acceleration yields 10–100× speedups for systems beyond 10–15 qubits when workloads are dominated by dense operations such as full-unitary application or quantum Fourier transforms. For sufficiently large problems, kernel execution dominates runtime (75–85% of total), while host–device transfer overhead becomes comparatively minor. Memory overhead remains within 20% of theoretical minima for statevector simulation, while sparse representations reduce density-matrix memory consumption by factors of 2–5 for locality-preserving noise models. Single-precision execution delivers near 2× throughput gains with acceptable fidelity for prototyping tasks, though double precision remains essential for deep circuits and phase-sensitive algorithms.

The presented framework operates within the intrinsic constraints of dense statevector simulation on heterogeneous CPU–GPU architectures. By default, "complex64" precision is employed to balance computational throughput and memory efficiency, while full "complex128" support remains available for experiments requiring enhanced numerical stability in deeper circuits. To ensure accurate benchmarking, GPU warm-up iterations are executed prior to performance measurements, thereby excluding initial memory allocation overhead from reported timing results. Furthermore, performance sensitivity observed near the upper qubit limits reflects the fundamental memory scaling behavior of dense Hilbert space representations as device capacity is approached. These characteristics represent deliberate architectural trade-offs inherent to dense linear algebra–based simulation rather than deficiencies of the abstraction layer itself.

These advances operate within the fundamental constraint of exponential state scaling, which limits single-node pure-state simulation to approximately 30 qubits and density-matrix simulation to substantially fewer qubits on current hardware. While the unified abstraction ensures portability across NumPy and CuPy backends, extracting peak performance presently requires hardware-aware tuning—a trade-off between accessibility and optimization that future automation layers will address.

Future development will prioritize distributed-memory execution, tensor-network contraction techniques and multi-GPU support to transcend single-node limits and compiler-driven optimization to reduce the expertise barrier for performance tuning. Together, these directions aim to extend the practical frontier of classical quantum simulation while preserving the framework's emphasis on reproducibility, numerical fidelity, and accessibility for algorithm developers.

Code Availability

The core implementation of the emulator is currently closed-source due to ongoing research development and proprietary integration considerations. Additionally, all benchmark scripts, pinned dependency manifests, and detailed build specifications required to reproduce the reported experiments are available via the corresponding author. This ensures that the performance claims and numerical parity results can be independently validated even without full access to the core engine.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

EM conceived and designed the study, performed the majority of the experimental research, implemented the core methodology, and drafted the initial manuscript. EK contributed to data acquisition, preprocessing, and validation, and assisted in reviewing and refining the manuscript draft. NŽ provided overall project administration, supervised the research direction and execution, and critically revised the manuscript for intellectual content. CR performed independent verification and validation of the experimental results, contributed to the interpretation of findings, and reviewed the final manuscript. All authors have read and approved the final version of the manuscript.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge: Cambridge University Press, 2000.

[2] A. Montanaro, "Quantum algorithms: An overview," *NPJ Quantum Information,* vol. 2, 15023, 2016.

[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Foundations of Computer Science*, 1994, pp. 124–134.

[4] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," arXiv preprint, arXiv:0406196, 2004.

[5] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum,* vol. 5, 410, 2021.

[6] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM J. Comput.*, vol. 38, no. 3, pp. 963–981, 2008.

[7] A. W. Cross, L. S. Bishop, J. A. Smolin *et al.*, "Open quantum assembly language (OpenQASM) 2.0," arXiv preprint, arXiv:1707.03429, 2017.

[8] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum,* vol. 2, 49, 2018.

[9] S. V. Isakov, D. Kafri, O. Martin *et al.*, "Simulations of quantum circuits with approximate noise using qsim and Cirq," arXiv preprint, arXiv:2111.02396, 2021.

[10] J. R. Johansson, P. D. Nation, and F. Nori, "QuTiP: An open-source Python framework for the dynamics of open quantum systems," *Comput. Phys. Commun.,* vol. 183, no. 8, pp. 1760–1772, 2012.

[11] NVIDIA cuQuantum. *NVIDIA Developer*. [Online]. Available: https://developer.nvidia.com/cuquantum

[12] Y. Suzuki, Y. Kawase, Y. Masumura *et al.* "Qulacs: A fast and versatile quantum circuit simulator for research purpose," arXiv preprint, arXiv:2011.13524, 2021.
[13] A. Kelly, "Simulating quantum computers using OpenCL," arXiv preprint, arXiv:1805.00988, 2018.
[14] B. Villalonga, S. Boixo, B. Nelson *et al.*, "A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware," *NPJ Quantum Information,* vol. 5, Art. no. 86, 2019.
[15] J. R. McClean, J. Romero, R. Babbush *et al.*, "The theory of variational hybrid quantum-classical algorithms," *New J. Phys.*, vol. 18, 023023, 2016.
[16] S. Bravyi, D. Gosset, and Y. Liu, "How to simulate quantum measurement without computing marginals," *Phys. Rev. Lett.,* vol. 128, 220503, 2022.
[17] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik, "qHiPSTER: The quantum high performance software testing environment," arXiv preprint, arXiv:1601.07195, 2016.
[18] S. Aaronson and L. Chen, "Complexity-theoretic foundations of quantum supremacy experiments," arXiv preprint, arXiv:1612.05903, 2016.
[19] M. Wang, S. Tannu, and P. J. Nair, "Accelerating simulation of quantum circuits under noise via computational reuse," in *Proc. 52nd Annu. Int. Symp. Computer Architecture*, 2025, pp. 1539–1553.
[20] Y. Nam, N. J. Ross, Y. Su *et al.*, "Automated optimization of large quantum circuits with continuous parameters," *NPJ Quantum Information,* vol. 4, 23, 2018.
[21] V. Gheorghiu, "Quantum++: A modern C++ quantum computing library," *PLOS ONE*, vol. 13, no. 12, e0208073, 2018.
[22] M. Fingerhuth, T. Babej, and P. Wittek, "Open source software in quantum computing," *PLOS ONE*, vol. 13, no. 12, e0208561, 2018.
[23] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," arXiv preprint, arXiv:1608.03355, 2016.
[24] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu *et al.*, "XACC: A system-level software infrastructure for heterogeneous quantum-classical computing," arXiv preprint, arXiv:1911.02452, 2019.
[25] S. X. Zhang, J. Allcock, Z. Q. Wan *et al.*, "TensorCircuit: A quantum software framework for the NISQ era," *Quantum*, vol. 7, 912, 2023.
[26] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory of Computing (STOC)*, 1996, pp. 212–219.
[27] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Phys. Rev. Lett.*, vol. 103, 150502, 2009.
[28] A. Peruzzo, J. McClean, P. Shadbolt *et al.*, "A variational eigenvalue solver on a photonic quantum processor," *Nat. Commun.*, vol. 5, 4213, 2014.
[29] CuPy: NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. *CuPy*. [Online]. Available: https://cupy.dev/
[30] G. Aleksandrowicz, T. Alexander, P. Barkoutsos *et al.* (January 2019). Qiskit: An open-source framework for quantum computing. *Zenodo*. [Online]. Available: https://zenodo.org/records/2562111
[31] G. Vidal, "Efficient classical simulation of slightly entangled quantum computations," *Phys. Rev. Lett.*, vol. 91, 147902, 2003.
[32] M. Broughton, G. Verdon, T. McCourt *et al.*, "TensorFlow quantum: A software framework for quantum machine learning," arXiv preprint, arXiv:2003.02989, 2021.
[33] M. A. Nielsen, "Cluster-state quantum computation," *Rep. Math. Phys.*, vol. 57, no. 1, pp. 147–161, 2006.
[34] S. McArdle, S. Endo, A. Aspuru-Guzik *et al.*, "Quantum computational chemistry," *Rev. Mod. Phys.*, vol. 92, 015003, 2020.
[35] S. Boixo, S. V. Isakov, V. N. Smelyanskiy *et al.*, "Characterizing quantum supremacy in near-term devices," *Nat. Phys.*, vol. 14, pp. 595–600, 2018.