Simulating Quantum Information Scrambling: A Comprehensive Guide to OTOC Calculation with PyQrack and Tensor Network Methods

Section 1: Introduction to Quantum Chaos and High-Performance Simulation

The Challenge of Simulating Quantum Many-Body Dynamics

The simulation of quantum many-body systems on classical computers represents one of the most significant challenges in computational science. The difficulty originates from the fundamental structure of quantum mechanics: the state of a system of N quantum bits, or qubits, is described by a vector in a Hilbert space whose dimension scales exponentially with N. For a system of N qubits, a complete description of its state vector requires storing and manipulating 2^N complex amplitudes. This exponential scaling, often referred to as the "exponential wall," renders the direct, or "brute-force," simulation of even moderately sized quantum systems—those with more than approximately 50-60 qubits—intractable for even the most powerful supercomputers available today.[1]

This computational barrier severely limits the ability to theoretically investigate and predict the behavior of a vast range of important physical systems, from novel materials and complex molecules in quantum chemistry to the dynamics of black holes in high-energy physics. Consequently, the development of advanced simulation techniques that can circumvent or mitigate this exponential scaling is a primary focus of modern research. These methods aim to exploit specific physical properties of the systems under study, such as limited entanglement or underlying symmetries, to find more compact and computationally efficient representations

of their quantum states and dynamics. Two prominent and powerful paradigms in this pursuit are high-performance hybrid quantum simulators and tensor network methods. The former, exemplified by the Qrack ecosystem, leverage sophisticated algorithmic optimizations and hardware acceleration to push the boundaries of direct simulation, while the latter provide a completely different framework for representing quantum states that can fundamentally break the exponential scaling for a large and physically relevant class of problems.

Information Scrambling, the Butterfly Effect, and Out-of-Time-Order Correlators

Within the complex dynamics of quantum many-body systems, the concept of quantum chaos describes how systems with complex interactions evolve in a manner that is highly sensitive to initial conditions. A key process associated with quantum chaos is information scrambling, which describes how initially localized quantum information spreads throughout the many degrees of freedom of a system, becoming hidden in complex, non-local correlations.[1] This process is a quantum analog of the classical "butterfly effect," where a small, local perturbation can have a dramatic, system-wide effect over time. Understanding and quantifying information scrambling is crucial for fields ranging from condensed matter physics, where it relates to thermalization, to quantum gravity, where it is connected to the information paradox of black holes.[3]

The primary tool for quantitatively diagnosing information scrambling is the Out-of-Time-Order Correlator (OTOC). The OTOC measures the degree to which two quantum operators, which are initially commuting and act on different parts of a system, fail to commute after one of them has evolved in time.[3] Formally, for two operators V and W, the OTOC is often defined as $F(t) = \langle W^\dagger(t) V^\dagger W(t) V \rangle$, where $W(t)$ is the operator W evolved in the Heisenberg picture. A non-trivial value of the OTOC indicates that the time evolution has spread the influence of operator W such that it now affects the part of the system where V acts. The rate at which the OTOC grows in chaotic systems provides a measure of how quickly information scrambles.[5] However, the very structure of the OTOC, involving both forward and backward time evolution, makes its direct calculation on classical computers exceptionally resource-intensive, often even more so than simulating simple forward dynamics.[1] This computational expense serves as the central motivation for developing the advanced simulation workflow detailed in this report.

Bridging Specialized Ecosystems: The Synergy of PyQrack and Tensor

Networks

This report details a powerful, modern methodology for the high-fidelity simulation of OTOCs by strategically combining the distinct strengths of two specialized quantum computing software ecosystems: the PyQrack high-performance simulator and dedicated tensor network libraries. This approach exemplifies a significant trend in computational physics, moving away from monolithic, one-size-fits-all simulation packages toward modular, interoperable workflows where the best tool is selected for each specific part of a complex problem.

PyQrack is the Python wrapper for the Qrack library, a C++-based quantum simulator designed for maximum performance on a wide range of classical hardware, including CPUs and GPUs via OpenCL and CUDA.[6] Qrack is not a simple state-vector simulator; it incorporates a suite of novel, hybrid optimization techniques, such as Schmidt decomposition and stabilizer tableau methods, to dramatically accelerate simulations of circuits with exploitable structural properties.[9] Its design philosophy emphasizes performance, portability, and interoperability, evidenced by its integration with major quantum frameworks like Qiskit, Cirq, and PennyLane.[6]

Tensor Networks (TNs), on the other hand, offer a fundamentally different approach to simulation. They provide a framework for representing quantum states and operators in a compressed form that is particularly efficient for systems with limited entanglement, a characteristic of many physically relevant ground states and short-time dynamics.[13] By representing the quantum state as a network of interconnected, smaller-rank tensors, TN methods can scale to hundreds or even thousands of qubits for certain problems, far beyond the reach of state-vector simulation.[15]

The synergy arises from a crucial integration point: PyQrack's QrackCircuit class, while primarily designed for its internal simulation engines, can be converted into a tensor network representation.[16] This allows a researcher to leverage PyQrack's user-friendly and highly optimized environment for circuit construction, pre-simulation, and analysis, while offloading the most computationally demanding task—the contraction of the large tensor network representing the OTOC dynamics—to a specialized, scalable backend. This report presents this hybrid workflow not merely as a technical tutorial, but as a case study in a new paradigm of high-performance quantum simulation, where frameworks are designed not as isolated islands but as interoperable components in a modular and powerful computational pipeline.

Section 2: Theoretical Foundations of Out-of-Time-Order Correlators

The OTOC Formalism and its Physical Interpretation

The Out-of-Time-Order Correlator is a multi-point correlation function that quantifies the growth of quantum operators in the Heisenberg picture and, by extension, the spread of quantum information. For a quantum system governed by a Hamiltonian H, the time evolution of an operator W is given by $W(t) = U^\dagger(t) W U(t)$, where $U(t) = e^{-iHt/\hbar}$ is the time evolution unitary. The most commonly studied form of the OTOC involves two generic, often local and initially commuting, Hermitian operators V and W. It is defined as the expectation value [3]:

$$F(t) = \langle W^\dagger(t) V^\dagger W(t) V \rangle_\beta$$
where the expectation value $\langle \cdot \rangle_\beta = \text{Tr}(e^{-\beta H} \cdot) / \text{Tr}(e^{-\beta H})$ is typically taken with respect to a thermal Gibbs state at inverse temperature β. For simplicity in simulation, this is often replaced by an expectation value in a pure state $|\psi_0\rangle$, such as the ground state or a simple product state, yielding $F(t) = \langle \psi_0 | W^\dagger(t) V^\dagger W(t) V | \psi_0 \rangle$.

The name "out-of-time-order" arises from the sequence of operators in the correlator: $t, 0, t, 0$. This is distinct from standard time-ordered correlation functions, like $\langle W(t) V(0) \rangle$, which are typically used to probe linear response and spectral properties. The OTOC can be related to the squared norm of the commutator $$ by considering the real part of $F(t)$. For unitary operators V and W, the OTOC is directly related to the squared commutator through the expression $C(t) = \langle ||^2 \rangle_\beta = 2(1 - \text{Re}[F(t)])$.[3] This connection provides a clear physical interpretation: the growth of $C(t)$ from its initial value of zero (since $ = 0$ for local, separated operators) directly measures how the initially local operator W spreads under time evolution to the extent that it no longer commutes with the local operator V. This process is the essence of information scrambling.

Furthermore, the OTOC can be interpreted as a measure of state fidelity. Consider two states: $|\psi_1\rangle = W(t) V |\psi_0\rangle$ and $|\psi_2\rangle = V W(t) |\psi_0\rangle$. The OTOC is simply the overlap of these two states, $F(t) = \langle \psi_2 | \psi_1 \rangle$. The state $|\psi_1\rangle$ corresponds to first applying a perturbation V and then evolving the operator W, while $|\psi_2\rangle$ corresponds to evolving W and then applying V. The deviation of $F(t)$ from unity quantifies how sensitive the system's state is to the order of these operations, providing another lens through which to view the chaotic dynamics.[3]

Probing Chaos with the OTOC

The connection between OTOCs and chaos is most sharply defined in the semiclassical limit. For classically chaotic systems, the squared commutator is expected to grow exponentially at early times, mirroring the exponential divergence of nearby trajectories characterized by the classical Lyapunov exponent λ_L. This behavior is captured by the OTOC as [3]:

$$C(t) \approx \epsilon e^{2\lambda_L t}$$
where ϵ is a small parameter related to the effective "size" of the operators, and the factor of 2 arises from the squared commutator. This exponential growth is a hallmark of scrambling and is often used to define a *quantum* Lyapunov exponent. This behavior was famously conjectured to be bounded in any quantum system, with $\lambda_L \le 2\pi k_B T / \hbar$, a result known as the Maldacena-Shenker-Stanford (MSS) bound on chaos, which is saturated by black holes.[3]

While this exponential growth is a strong indicator of chaos, further research has revealed a more nuanced relationship. It is now understood that exponential growth of the OTOC is a necessary but not sufficient condition for quantum chaos.[1] Some integrable or non-chaotic systems can exhibit transient exponential growth. A more complete diagnosis of chaos requires examining other signatures, such as spectral statistics conforming to random matrix theory. Nevertheless, the OTOC remains an invaluable tool. After the initial period of exponential growth, the OTOC typically saturates to a value consistent with thermal equilibrium, a process linked to the system's thermalization.[4] The spatiotemporal profile of the OTOC, $C(x, t)$, where the operators V and W are separated by a distance x, can reveal a "light cone" effect, showing a characteristic butterfly velocity v_B that describes the speed at which information propagates through the system.[1]

Mapping OTOCs to Quantum Circuits: The Forward-Backward Evolution Protocol

A key advantage of the OTOC formalism for the field of quantum computation is that it can be directly mapped to a specific quantum circuit protocol that is, in principle, measurable on a quantum computer and, more relevantly for this report, simulable on a classical one. This protocol, sometimes referred to as a "quantum echo," directly implements the operator sequence in the OTOC definition on a quantum state.[5]

The standard circuit for measuring the real part of the OTOC, $\text{Re}[F(t)]$, often requires an ancillary qubit and controlled operations. However, a more direct simulation approach

focuses on calculating the full expectation value $\langle \psi_0 | W^\dagger(t) V^\dagger W(t) V | \psi_0 \rangle$. This can be achieved by constructing a tensor network that represents this entire process. The structure of this process is as follows:

1. **State Preparation:** The simulation begins with a representation of the initial state, $|\psi_0\rangle$. In a tensor network, this is typically a simple product state, represented as a chain of rank-1 tensors.
2. **Forward Evolution (V):** The operator V is applied to the initial state. In the circuit model, this corresponds to a layer of quantum gates.
3. **Forward Time Evolution ($W(t)$ part 1):** The state is evolved forward in time under the Hamiltonian H for a duration t. This is represented by applying the unitary operator $U(t) = e^{-iHt}$.
4. **Perturbation (W):** The operator W is applied.
5. **Backward Time Evolution ($W(t)$ part 2):** The state is evolved backward in time, represented by applying the unitary $U^\dagger(t) = e^{iHt}$. This "reverses" the dynamics.
6. **Final Operator ($V^\dagger W^\dagger$):** The remaining operators from the OTOC expression are applied to the bra state $\langle \psi_0 |$. In the full tensor network contraction, this corresponds to applying the conjugate transpose of the operators to the conjugate of the initial state network.

The resulting computational task is to contract the entire tensor network formed by these sequential operations. A crucial feature of this protocol is the presence of both the forward evolution $U(t)$ and the backward evolution $U^\dagger(t)$. In a tensor network representation, where gates are individual tensors, the network for $U^\dagger(t)$ is composed of the Hermitian conjugates of the tensors for $U(t)$. When these two sub-networks are connected, there is significant potential for internal simplification and contraction before the full, computationally expensive contraction of the entire network is performed. This inherent symmetry in the OTOC protocol is not just a theoretical curiosity; it is a structural feature that can be exploited by sophisticated tensor network contraction algorithms to significantly reduce the computational cost of the simulation. This goes beyond simply treating the circuit as a generic sequence of gates and instead leverages the underlying physics of the OTOC to optimize its simulation.

Section 3: The Tensor Network Computational Paradigm

Beyond the Exponential Wall: Representing Quantum States with Tensor Networks

Tensor networks provide a powerful mathematical and computational framework for representing and manipulating the high-dimensional tensors that arise in quantum many-body physics, effectively mitigating the "exponential wall" for a large class of physically relevant states.[13] A tensor is a multi-dimensional array of numbers, generalizing scalars (rank-0), vectors (rank-1), and matrices (rank-2) to an arbitrary number of indices.[19] A tensor network is a collection of such tensors whose indices are interconnected according to a specific graph structure. In this graphical language, tensors are represented by nodes, and their indices are represented by edges or "legs" extending from the nodes. An edge connecting two nodes signifies a tensor contraction—a summation over the shared index—which is a generalization of matrix multiplication.[19]

The core idea behind using tensor networks for quantum simulation is to decompose the single, exponentially large tensor of coefficients of a quantum state vector into a network of many smaller, interconnected tensors. The complexity of this representation is governed not by the total number of qubits, but by the dimensions of the internal indices that connect the tensors. These dimensions are known as "bond dimensions" (often denoted by χ). The maximum bond dimension required to exactly represent a given quantum state is related to the amount of entanglement across any bipartition of the system, as quantified by the Schmidt rank. The celebrated "area law" of entanglement entropy, which states that the entanglement of a gapped ground state scales with the area of the boundary of a subregion rather than its volume, implies that such states can be efficiently represented by tensor networks with a small, constant bond dimension.[14] This property allows tensor network methods to accurately capture the physics of many ground states and short-time dynamical evolutions with computational resources that scale polynomially, rather than exponentially, with the system size.

Core Architectures for Dynamic Simulations: Matrix Product States (MPS) and Matrix Product Operators (MPO)

For one-dimensional quantum systems, which are a primary focus for many foundational studies of quantum dynamics, the most successful and widely used tensor network architecture is the Matrix Product State (MPS).[13]

Matrix Product State (MPS): An MPS represents the state vector $|\psi\rangle$ of an

L-qubit system by decomposing its 2^L amplitudes, $C_{s_1 s_2 \dots s_L}$, into a product of L tensors, one for each physical site. Each tensor $A_i^{s_i}$ has one physical index s_i (of dimension d, which is 2 for a qubit) and two virtual or bond indices, α_{i-1} and α_i, that connect it to its neighbors in a chain-like structure. The amplitude for a specific computational basis state $|s_1 s_2 \dots s_L\rangle$ is given by the matrix product [15]:

$$C_{s_1 s_2 \dots s_L} = \text{Tr}[A_1^{s_1} A_2^{s_2} \cdots A_L^{s_L}]$$
The tensors at the ends of the chain are vectors, ensuring the final product is a scalar. The dimension of the bond indices, χ, is the bond dimension of the MPS. The memory required to store an MPS scales as $O(Ld\chi^2)$, a polynomial scaling in system size L that contrasts sharply with the exponential $O(d^L)$ scaling of a full state vector. Any quantum state can be exactly represented as an MPS, but the required bond dimension χ may become exponentially large for highly entangled states. The power of the MPS representation lies in its ability to approximate states by truncating χ, keeping only the most significant contributions to the entanglement spectrum.

Matrix Product Operator (MPO): The MPS concept can be naturally extended from states (vectors) to operators (matrices). A Matrix Product Operator (MPO) represents a large operator acting on the L-qubit Hilbert space as a chain of four-index tensors.[13] Each tensor $W_i^{s_i, s'_i}$ has two physical indices, an input s'_i and an output s_i, and two bond indices connecting it to its neighbors. MPOs are particularly useful for representing local Hamiltonians, as the sum of local terms can be efficiently constructed as an MPO with a very small bond dimension. They are also central to time-evolution algorithms, where the time-evolution operator $U(t)$ can be approximated as an MPO.[15]

The Language of Contraction: Simulating Circuit Evolution and Expectation Values

When simulating a quantum circuit using tensor networks, the methodology differs from the MPO-based time evolution common in condensed matter physics (like the Time-Evolving Block Decimation, TEBD, algorithm). Instead of approximating the propagator e^{-iHt}, the circuit is translated directly into a two-dimensional tensor network where each quantum gate becomes an individual tensor.[19]

In this representation:

- The initial state of the qubits (e.g., $|00\dots0\rangle$) is a set of rank-1 tensors.
- Each single-qubit gate is a rank-2 tensor (a matrix).
- Each two-qubit gate is a rank-4 tensor.

- The "wires" of the quantum circuit correspond to the indices of these tensors. Applying a gate to a set of qubits is equivalent to contracting the corresponding gate tensor with the open indices of the tensors representing the state of those qubits at that point in time.

The simulation of the entire quantum circuit up to a certain point yields a large, complex tensor network. To calculate a specific quantity, such as the probability amplitude of a final basis state or the expectation value of an operator, one must contract this entire network down to a single scalar. The order in which these pairwise tensor contractions are performed has a dramatic impact on the computational cost (both time and memory). Finding the optimal contraction path is an NP-hard problem. Therefore, a crucial component of any tensor network simulation library is a "contraction path finder," an algorithm that uses heuristics (often based on graph partitioning) to find a near-optimal sequence of contractions.[23] The final, most intensive step is the execution of this contraction path, which is where the exponential slowdown for simulating generic, highly entangling quantum circuits ultimately manifests. However, for circuits with limited entanglement or special structures, this approach can be vastly more efficient than state-vector simulation. The workflow developed in this report leverages exactly this paradigm: it uses pyqrack to define the gate sequence for the OTOC protocol and then translates this sequence into a tensor network for subsequent contraction by a specialized backend.

Section 4: Architectural Deep Dive: The PyQrack Simulation Engine

Core Components: The C++ Qrack Library and the PyQrack Python Wrapper

The PyQrack ecosystem is built upon a two-layer architecture designed to maximize performance while providing a high-level, user-friendly interface. At its core is the **Qrack library**, a comprehensive framework for quantum computer simulation written in C++11.[6] Qrack is engineered for speed, portability, and minimal dependencies. Its C++ foundation allows for low-level memory management and direct access to hardware acceleration features. It supports multi-device GPU acceleration through OpenCL, making it agnostic to specific GPU vendors, and can also be compiled for CUDA for NVIDIA-specific hardware or in a CPU-only configuration for maximum portability.[8]

The second layer is **PyQrack**, which serves as the official Python wrapper for the Qrack library.[8] Unlike many wrappers that rely on complex binding generators like pybind11 or SWIG, PyQrack is a "pure Python" wrapper that uses the built-in ctypes library. This design choice has several significant advantages: it introduces zero additional Python package dependencies, ensures broad compatibility across Python versions (theoretically back to Python 2.3), and provides a lightweight, direct interface to the compiled C++ shared library.[6] This architecture allows Python users to harness the full performance of the underlying C++ engine with the ease and flexibility of the Python programming language.

Hybrid Simulation Strategies: Native Optimizations

Qrack's high performance stems from its implementation of a suite of advanced, often hybrid, simulation strategies that go far beyond basic state-vector evolution. The simulator can transparently switch between or combine these methods to best suit the structure of the quantum circuit being executed.[10]

- **Schmidt Decomposition (QUnit):** This is one of Qrack's foundational optimizations, implemented in a layer referred to as QUnit.[7] Schmidt decomposition (or Singular Value Decomposition) is used to identify and exploit separability in the quantum state. If a subset of qubits is not entangled with the rest of the system, Qrack can simulate it as a separate, smaller state vector, significantly reducing the memory footprint and computational cost of subsequent operations on that subset. This optimization is applied dynamically and transparently, making Qrack particularly efficient for circuits with evolving entanglement structures or those that contain many operations on unentangled qubits.
- **Hybrid Stabilizer Simulation (QStabilizerHybrid):** For circuits dominated by Clifford gates (Hadamard, Phase, CNOT, and Pauli gates), Qrack employs a powerful hybrid simulation technique based on the Gottesman-Knill theorem.[9] The QStabilizerHybrid engine tracks the quantum state using a stabilizer tableau, a classical data structure that can be updated in polynomial time under the action of Clifford gates. When a non-Clifford gate (such as a T gate or an arbitrary rotation) is encountered, Qrack does not abandon the stabilizer formalism entirely. Instead, it can represent the state as a superposition of stabilizer states or fall back to a state-vector representation for only the necessary components of the system, while maintaining the stabilizer description for the rest.[10] This makes Qrack exceptionally fast for "near-Clifford" circuits, which are common in quantum error correction and certain algorithms. For the OTOC simulation, this feature can be particularly potent for simulating the Trotterized time evolution of certain Hamiltonians, like the Ising model, which can be constructed from a large number of Clifford gates.
- **Other Techniques:** Qrack also offers alternative simulation backends that can be

explicitly selected by the user. One such method is based on **Quantum Binary Decision Diagrams (QBDDs)**, a graph-based data structure that can represent quantum states compactly if they possess certain types of structural regularity, often corresponding to low entanglement.[10] While not a GPU-accelerated method, QBDD simulation can offer significant memory advantages for specific classes of circuits on CPU-based systems.

The QrackSimulator and QrackCircuit Classes: An API Perspective

Interaction with the Qrack engine from Python is primarily managed through two high-level classes:

- **QrackSimulator:** This class represents an instance of the quantum simulator itself. When instantiated, it can be configured with a variety of parameters that control the underlying simulation methods and hardware usage. For example, a user can specify the number of qubits, and toggle flags such as isStabilizerHybrid=True, isTensorNetwork=False (note: this refers to an internal, novel TN-like technique, not the external offloading discussed in this report), or isOpenCL=True to customize the simulator's behavior for a specific task.[29] The simulator object holds the quantum state and exposes methods for applying gates, performing measurements, and retrieving state information.
- **QrackCircuit:** This class provides a mechanism for defining a quantum circuit as a sequence of operations before it is executed on a simulator.[29] A key feature of QrackCircuit is its interoperability. It is not a closed, proprietary format; rather, it is designed to integrate with the broader quantum software ecosystem. For instance, a QrackCircuit can be directly instantiated from a Qiskit QuantumCircuit object using the class method QrackCircuit.in_from_qiskit_circuit(qiskit_circ).[29] This capability is central to the workflow of this report, as it allows a user to construct their OTOC circuit using familiar tools like Qiskit and then seamlessly bring it into the PyQrack environment for either native simulation or, crucially, for conversion to a tensor network.

Performance Profile: Identifying Use Cases for Native vs. Offloaded Computation

The architectural features of PyQrack define a clear performance profile that informs the strategic decision of when to use its native simulation capabilities versus when to offload the computation to an external tensor network backend.

Native PyQrack simulation is the optimal choice when:

- The number of qubits is relatively small (typically $N < 30$), where state-vector simulation on a GPU is extremely fast and the overhead of setting up a tensor network contraction is not justified.
- The circuit is predominantly Clifford or "near-Clifford." In this regime, the QStabilizerHybrid engine's polynomial scaling will vastly outperform any other simulation method, including tensor networks.
- The quantum state throughout the evolution is known to have low entanglement, allowing the QUnit Schmidt decomposition optimizations to be highly effective.

Offloading to a Tensor Network backend is necessary or advantageous when:

- The number of qubits is large ($N > 40-50$), making the 2^N memory requirement of state-vector simulation prohibitive.
- The circuit generates a high degree of entanglement, but the entanglement structure is expected to adhere to an area law (e.g., in 1D systems), allowing for an efficient MPS representation with a manageable bond dimension χ.
- The primary goal is to calculate a small number of amplitudes or local expectation values from a very large system, a task for which contracting a targeted portion of a tensor network is particularly well-suited.

The OTOC simulation of a chaotic spin chain for long times represents a canonical example of the second case. As time progresses, entanglement spreads throughout the chain, making state-vector methods unfeasible, while the one-dimensional nature of the system makes it a perfect candidate for an MPS-based tensor network approach. The following table provides a structured comparison of these simulation paradigms.

Simulation Method	**Memory Complexity**	**Time Complexity (Gate)**	**Strengths**	**Weaknesses/ Limitations**
PyQrack State-Vector	$O(2^N)$	$O(2^N)$	Exact; extremely fast for small N (<30) on GPU; simple to implement.	Exponential scaling in memory and time; intractable for $N > 50-60$.
PyQrack Hybrid Stabilizer	$O(N^2 + k 2^N)$	$O(\text{poly}(N))$ for Clifford gates	Polynomial time for Clifford circuits;	Scales exponentially with the number of

			exceptionally fast for near-Clifford circuits.	non-Clifford gates (k).
External Tensor Network (MPS)	$O(Ld\chi^2)$	$O(Ld\chi^3)$	Scales polynomially with system size L; can handle hundreds of qubits if entanglement (and thus χ) is bounded.	Scales exponentially with entanglement (bond dimension χ); can be slow for highly entangled states; finding optimal contraction path is hard.

Section 5: The Integration Workflow: Connecting PyQrack to Tensor Network Libraries

The QrackCircuit as a Universal Circuit Description

The cornerstone of the hybrid simulation workflow is the QrackCircuit class, which serves as a versatile and interoperable representation of a quantum circuit. Its role extends beyond being a simple data structure for the native QrackSimulator. Due to its ability to be constructed from other major quantum software development kits, it functions as a powerful intermediary. A user can design and build their quantum circuit using the familiar syntax and tools of a preferred framework, such as Qiskit or Cirq, and then import it seamlessly into the PyQrack ecosystem.[12]

This capability is particularly valuable for complex tasks like constructing the OTOC circuit. The required Trotter-Suzuki decomposition of the time-evolution operator can be intricate to

build manually. Libraries like Qiskit provide high-level tools to generate such structures automatically from a Hamiltonian definition. The workflow can thus begin in Qiskit, leveraging its advanced circuit synthesis features, and then transition to PyQrack via a single method call (QrackCircuit.in_from_qiskit_circuit). Once inside the PyQrack environment, the QrackCircuit object holds a complete, backend-agnostic description of the gate sequence, ready for the next crucial step: translation into the language of tensor networks.

The CircuitToEinsum Conversion: Translating Quantum Gates into a Tensor Network

The critical link enabling the offloading of computation from PyQrack to a tensor network library is a conversion utility that translates the gate-based description of a QrackCircuit into a format suitable for tensor contraction. While the C++ Qrack library has contemplated a native ConvertToTensorNetwork() method to interface directly with NVIDIA's cuTensorNet, the currently available and functional implementation for Python users exists as a utility function within the pyqrack-jupyter examples repository.[18] This function, which can be conceptualized as CircuitToEinsum, provides the exact bridge required for this workflow. Its existence, primarily as a demonstration in an examples repository, highlights the expert-level nature of this technique, as it is not yet part of the formally documented core PyQrack API.

The conversion process implemented by this function is as follows:

1. **Initialization:** It begins by creating tensors representing the initial state of each qubit, typically the $|0\rangle$ state, which is the vector $$.
2. **Gate Iteration:** The function iterates through every gate in the input QrackCircuit object.
3. **Tensor Generation:** For each gate, it generates the corresponding tensor. A single-qubit gate is a 2×2 matrix (a rank-2 tensor). A two-qubit gate, like a CNOT, is a $2 \times 2 \times 2 \times 2$ array (a rank-4 tensor).
4. **Index Management:** It systematically assigns indices to each leg of every tensor. A convention is used where each "wire" in the circuit at each "time step" (between gates) is given a unique index. When a gate is applied, its input indices match the output indices of the previous layer, and it produces new output indices.
5. **Output Generation:** The function returns two primary outputs:
    - An **einsum-style string:** This is a string that specifies the entire tensor network contraction in the notation used by NumPy's einsum function. It lists the indices for each tensor and specifies which final indices should remain uncontracted. For an expectation value calculation, all indices are contracted, resulting in a scalar.
    - A **list of tensors:** This is a Python list containing the actual numerical tensor objects (e.g., as NumPy arrays) in the order corresponding to the einsum string.

This pair of outputs is a universal description of the tensor network contraction problem, completely decoupled from the PyQrack environment, and can be consumed by any standard tensor network library that supports an einsum-like interface.

Practical Interface Patterns for quimb and TensorNetwork

Once the einsum string and the list of tensors have been generated, they can be passed to various Python tensor network libraries for contraction.

- **tensornetwork Library:** Google's tensornetwork library provides a natural interface for this format through its ncon function, which is designed for contracting networks specified in a similar notation.[30] The einsum string and tensor list from CircuitToEinsum can be readily adapted for use with ncon to perform the contraction.
- **quimb Library:** The quimb library, another powerful tool for tensor network calculations, can also consume this output.[31] While it may not have a single function that directly parses an einsum string for an entire network, one can programmatically construct a quimb.TensorNetwork object. This involves iterating through the list of tensors, adding each one to the network, and then connecting their indices according to the relationships defined in the einsum string. quimb then offers its own sophisticated tools for simplifying and contracting the resulting network object.[22]

Accelerated Contraction with NVIDIA's cuTensorNet

For achieving maximum performance, especially for the large and complex tensor networks generated by OTOC circuits, the ideal backend is NVIDIA's cuTensorNet library, part of the cuQuantum SDK.[23] This library is specifically designed to accelerate tensor network computations on NVIDIA GPUs.

- **Setup:** Using cuTensorNet requires a compatible hardware and software environment, including an NVIDIA GPU (Compute Capability 7.0+), the CUDA toolkit, and the cuquantum-python package, which provides the Python bindings.[33]
- **Integration:** The integration is remarkably straightforward due to the design of the cuquantum-python API. The library provides a high-level function, cuquantum.contract, which is a direct, GPU-accelerated replacement for numpy.einsum or opt_einsum.contract.[24] It accepts the exact same inputs generated by our CircuitToEinsum function: the einsum string and the list of tensors (which can be NumPy

arrays, CuPy arrays, or PyTorch tensors).

When cuquantum.contract is called, it handles the entire high-performance contraction workflow on the GPU:

1. **Pathfinding:** It internally invokes a hyper-optimizer to find a near-optimal contraction path for the given network, a step that is critical for performance.[24]
2. **Execution:** It executes the sequence of pairwise tensor contractions on the GPU, leveraging highly optimized kernels for maximum throughput.
3. **Autotuning:** It may optionally autotune its kernels to find the most efficient implementation for a specific contraction.[24]

This direct, drop-in compatibility makes cuTensorNet the premier choice for the contraction step of our workflow, fulfilling the original motivation behind the development of the QCircuit class, which was to interface with high-performance tensor network backends.[18]

Table 2: Key QrackCircuit and CircuitToEinsum API Reference	
Component	**Description & Usage**
QrackCircuit Construction	
QrackCircuit.in_from_qiskit_circuit(qiskit_circ)	**Class Method.** Instantiates a QrackCircuit object from a pre-existing qiskit.QuantumCircuit object. This is the primary entry point for leveraging external circuit generation tools. [29]
qcirc.run(qsim)	**Instance Method.** Executes the defined circuit on a QrackSimulator instance. Used for native PyQrack simulation, not for the TN workflow. [29]
CircuitToEinsum Conversion	
CircuitToEinsum(qcirc)	**Hypothetical Function.** Takes a QrackCircuit object as input. This represents the utility found in the pyqrack-jupyter repository. [18]

Returns: (einsum_path, tensor_list)	**einsum_path (str):** An einsum-compatible string describing the full network contraction. E.g., 'ab,bc,cd->ad'.
	tensor_list (list): A list of NumPy arrays, where each array is a tensor corresponding to a term in the einsum_path string.

Section 6: End-to-End Implementation: OTOC of a Spin Chain

This section provides a complete, end-to-end implementation of the OTOC simulation workflow, from defining the physical system to analyzing the final results. The code presented here serves as a practical guide, demonstrating how the theoretical concepts and software components discussed in previous sections are integrated into a functional research script.

Defining the Physical System: The Transverse-Field Ising Model (TFIM)

The one-dimensional Transverse-Field Ising Model (TFIM) is chosen as the model system for this demonstration. It is a canonical model in condensed matter physics that exhibits a quantum phase transition and chaotic dynamics in certain parameter regimes, making it an excellent candidate for studying information scrambling. Its relevance within the Qrack ecosystem is further underscored by the existence of specialized tools like PyQrackIsing for related problems.[36] The Hamiltonian for the TFIM with nearest-neighbor interactions is given by:

$$H = -J \sum_{i=0}^{L-2} Z_i Z_{i+1} - h \sum_{i=0}^{L-1} X_i$$
where Z_i and X_i are the Pauli-Z and Pauli-X operators acting on qubit i, L is the number of spins (qubits) in the chain, J is the interaction strength, and h is the strength of the transverse magnetic field. For this simulation, we will consider a non-integrable regime (e.g., by adding a longitudinal field or next-nearest-neighbor interactions, or simply by choosing parameters that lead to chaotic behavior) to observe non-trivial scrambling dynamics.

Step 1: Building the OTOC Circuit Components in pyqrack

The first step is to construct the quantum circuit that corresponds to the OTOC measurement protocol. The time evolution unitary $U(t) = e^{-iHt}$ is approximated using a first-order Trotter-Suzuki decomposition. The Hamiltonian is split into two commuting parts, $H = H_{ZZ} + H_X$, where $H_{ZZ} = -J \sum_i Z_i Z_{i+1}$ and $H_X = -h \sum_i X_i$. A single Trotter step of duration δt is then approximated as:

$$U(\delta t) \approx e^{-iH_X \delta t} e^{-iH_{ZZ} \delta t} = \left(\prod_i e^{ihX_i \delta t} \right) \left(\prod_i e^{iJ Z_i Z_{i+1} \delta t} \right)$$
Each term in this product corresponds to a standard quantum gate: $e^{ihX_i \delta t}$ is an RX rotation, and $e^{iJ Z_i Z_{i+1} \delta t}$ can be decomposed into CNOT gates and an RZ rotation. The full evolution $U(t)$ is constructed by repeating this step $N_{steps} = t / \delta t$ times.

The operators for the OTOC measurement are chosen to be local Pauli operators, for instance, $V = X_0$ (Pauli-X on the first qubit) and $W = Z_{L/2}$ (Pauli-Z on the middle qubit). The complete OTOC circuit is then assembled in PyQrack by creating a QrackCircuit object and appending the gate sequences for V, $U(t)$, W, and $U^\dagger(t)$ in the correct order.

Python

```
import numpy as np
from pyqrack import QrackSimulator, QrackCircuit
# Assume existence of a utility to convert from Qiskit
# from qiskit.quantum_info import SparsePauliOp
# from qiskit.circuit.library import PauliEvolutionGate

def create_trotter_step_circuit(L, J, h, dt):
    """Creates a QrackCircuit for one Trotter step of the TFIM."""
    # This is a conceptual implementation. In practice, one would
    # use a library like Qiskit to build this and then convert.
    # For example, using qiskit.circuit.library.PauliEvolutionGate
    # and then QrackCircuit.in_from_qiskit_circuit(qiskit_circ).

    # Placeholder for a complex circuit construction
    # In a real scenario, this would involve creating a qiskit.QuantumCircuit
```

```
    # with the appropriate RZZ and RX gates, and then converting it.
    # qiskit_circ =...
    # qcirc = QrackCircuit.in_from_qiskit_circuit(qiskit_circ)
    # return qcirc
    pass # Returning None for demonstration purposes

def build_otoc_circuit(L, J, h, t, dt):
    """Builds the full OTOC circuit V, U(t), W, U_dagger(t)."""
    # Define operators
    V_op = {'gate': 'x', 'qubits': }
    W_op = {'gate': 'z', 'qubits': [L // 2]}

    # Build U(t) and U_dagger(t) from Trotter steps
    num_steps = int(t / dt)
    trotter_step = create_trotter_step_circuit(L, J, h, dt)
    # trotter_step_dag = trotter_step.inverse() # Conceptual

    # Assemble the full circuit
    # full_qcirc = QrackCircuit(L)
    # full_qcirc.append_gate(V_op)
    # for _ in range(num_steps):
    #     full_qcirc.append_circuit(trotter_step)
    # full_qcirc.append_gate(W_op)
    # for _ in range(num_steps):
    #     full_qcirc.append_circuit(trotter_step_dag)

    # This function would return a fully constructed QrackCircuit object.
    pass
```

Step 2: Exporting the Circuit to an einsum-based Tensor Network Representation

With the QrackCircuit object constructed, the next step is to convert it into the universal tensor network format. This is achieved by calling the CircuitToEinsum utility. This function takes the circuit object and processes its gate-by-gate description, outputting the einsum contraction path string and the corresponding list of tensors as NumPy arrays.

Python

```
# Assume CircuitToEinsum is available from a utility module
# from tn_utils import CircuitToEinsum

# L = 10
# J = 1.0
# h = 0.5
# t = 2.0
# dt = 0.1

# otoc_qcirc = build_otoc_circuit(L, J, h, t, dt)

# The core conversion step
# einsum_path, tensor_list = CircuitToEinsum(otoc_qcirc)

# print("Einsum Path String (example snippet):", einsum_path[:50], "...")
# print("Number of tensors in network:", len(tensor_list))
```

The einsum_path string will be a long, complex string encoding the connections between hundreds or thousands of tensors, while tensor_list will contain the numerical data for each gate and initial state in the circuit.

Step 3: Executing the Contraction with a Chosen TN Backend

This step involves feeding the output from Step 2 into a tensor network contraction engine. For optimal performance, NVIDIA's cuTensorNet is the preferred choice. The script will import the cuquantum library and use its contract function. For comparison, and to demonstrate the massive performance difference, the same contraction can be performed using a CPU-based tool like opt_einsum, which also finds a good contraction path before executing it with NumPy.

Python

```
import time
# import cuquantum
# from opt_einsum import contract as opt_einsum_contract
```

```
# --- GPU Accelerated Contraction ---
# try:
#     start_gpu = time.time()
#     # The tensors must be on the GPU, e.g., as CuPy arrays
#     # tensor_list_gpu = [cupy.asarray(t) for t in tensor_list]
#     # otoc_value_gpu = cuquantum.contract(einsum_path, *tensor_list_gpu)
#     end_gpu = time.time()
#     print(f"cuTensorNet contraction took: {end_gpu - start_gpu:.4f} seconds")
#     # print(f"OTOC Value (GPU): {otoc_value_gpu}")
# except ImportError:
#     print("cuQuantum not found. Skipping GPU contraction.")

# --- CPU-based Contraction for Comparison ---
# start_cpu = time.time()
# otoc_value_cpu = opt_einsum_contract(einsum_path, *tensor_list)
# end_cpu = time.time()
# print(f"opt_einsum (CPU) contraction took: {end_cpu - start_cpu:.4f} seconds")
# print(f"OTOC Value (CPU): {otoc_value_cpu}")
```

This comparison will typically show orders of magnitude speedup for the GPU-based contraction, especially for larger system sizes and deeper circuits, consistent with benchmarks showing significant performance gains over CPU-based libraries.[24]

Step 4: Data Analysis and Visualization of the Scrambling Dynamics

To study the dynamics of information scrambling, the entire process from Step 1 to Step 3 is repeated for a range of evolution times t. The real part of the calculated OTOC value, $\text{Re}[F(t)]$, is stored for each time point. The final step is to plot this data to visualize the behavior of the OTOC.

The resulting plot is expected to show characteristic features of scrambling dynamics:

- An initial value of $\text{Re}[F(t=0)] \approx 1$, indicating that the operators initially commute.
- A period of decay, which may be exponential in a chaotic system, signifying the onset of scrambling.
- Oscillations and eventual saturation to a small value close to zero, corresponding to the thermalization of the system where the information of the initial perturbation is spread throughout the system.

This plot provides a direct, quantitative visualization of the quantum butterfly effect in the

simulated spin chain, a result made computationally accessible through the hybrid PyQrack-TensorNetwork workflow. The generation of such data via tensor network methods is a standard approach in contemporary research on quantum chaos.[38]

Section 7: Advanced Techniques and Optimization Strategies

While the end-to-end workflow described provides a powerful method for exact OTOC calculation, its feasibility is ultimately limited by the growth of entanglement in the quantum system. As the simulation time increases, the bond dimension required to exactly represent the state grows, leading to an exponential increase in the cost of tensor contraction. This section explores advanced techniques to push the boundaries of simulation further, focusing on approximate methods and hybrid workflows that combine the strengths of different simulation paradigms.

Managing Computational Cost: Bond Dimension Truncation and Fidelity

For simulations of larger systems or longer evolution times where exact contraction becomes intractable, approximate tensor network methods are essential. The most common approximation technique is **bond dimension truncation**. This method is typically employed in algorithms that evolve a Matrix Product State (MPS) step-by-step, such as TEBD. After each application of a gate or a small time-evolution operator, the bond dimension of the MPS may increase. To keep the computation manageable, a Singular Value Decomposition (SVD) is performed across the bond, and only the χ largest singular values and their corresponding singular vectors are retained, where χ is a pre-defined maximum bond dimension.[30]

This truncation introduces a controlled error into the simulation. The magnitude of the discarded singular values provides a measure of the error at each step, allowing the trade-off between computational cost (determined by χ) and simulation fidelity to be systematically controlled. While the primary workflow of this report focuses on contracting the full circuit tensor network, which is conceptually simpler, the principles of approximation are crucial for scaling. Advanced libraries like cuTensorNet provide low-level APIs for performing key tensor decompositions like SVD and QR factorization on the GPU, which are

the building blocks for developing custom, high-performance approximate time-evolution algorithms.[23] A researcher could adapt the workflow to build the OTOC circuit as an MPO and apply it to an MPS, using these GPU-accelerated primitives to manage the bond dimension at each step.

Hybrid Workflows: Combining pyqrack's Near-Clifford Speed with TN Scaling

A more sophisticated optimization strategy involves creating a hybrid workflow that leverages the distinct advantages of PyQrack's native simulation engines and external tensor network backends within a single simulation. Many physical processes, including certain OTOC protocols, involve circuits that can be naturally partitioned into segments with vastly different entanglement characteristics. For example, a circuit might have a long initial phase (preamble) composed almost entirely of Clifford gates, followed by a shorter, highly-entangling non-Clifford section.

In such a scenario, a purely tensor-network-based approach might be inefficient for the preamble, while a purely state-vector approach would fail on the entangling section. A hybrid workflow offers a superior solution:

1. **Partition the Circuit:** Identify the point in the QrackCircuit where the dynamics transition from being efficiently simulable by native PyQrack methods to requiring a tensor network representation. This is typically where significant non-Clifford structure or entanglement generation begins.
2. **Native Simulation of Preamble:** Execute the first part of the circuit using a QrackSimulator configured for optimal performance—for instance, with isStabilizerHybrid=True.[29] This step is computationally cheap, even for a large number of qubits, as long as the circuit segment is near-Clifford.
3. **State Export:** After the preamble, the state of the QrackSimulator is exported. While direct export to an MPS is not a standard feature, PyQrack provides methods to output the state in other formats, such as a stabilizer tableau with non-Clifford corrections, which can then be converted into an MPS representation.[9]
4. **Tensor Network Simulation of Remainder:** The exported MPS becomes the initial state for a subsequent tensor network simulation of the remaining, highly-entangling part of the circuit. This second stage would be handled by a library like quimb or a custom code using cuTensorNet primitives.

This hybrid approach intelligently delegates each part of the computational task to the most suitable engine, maximizing efficiency and enabling the simulation of problems that would be intractable for any single method alone. It represents the frontier of high-performance

quantum simulation, requiring a deep understanding of both the physics of the problem and the architecture of the available software tools.

Benchmarking and Profiling for Optimal Performance

To effectively implement and optimize these advanced workflows, rigorous benchmarking and profiling are essential. A user should systematically measure the performance of each stage of the simulation pipeline to identify computational bottlenecks. Key metrics to record include:

- **Circuit Construction Time:** The time taken to build the QrackCircuit object. This is usually negligible but can be relevant for very deep circuits.
- **TN Conversion Time:** The time required for the CircuitToEinsum utility to process the circuit and generate the tensor network representation. As this is currently a Python-level process, it could become a bottleneck for extremely large circuits, motivating the future development of a C++ implementation.[18]
- **Contraction Pathfinding Time:** For large networks, the time spent by the backend (e.g., cuTensorNet or opt_einsum) to find an efficient contraction path can be significant. cuTensorNet is highly optimized for this task.[24]
- **Contraction Execution Time:** The time for the final, numerically intensive contraction. This is typically the dominant cost and the primary target for acceleration via GPUs.

By profiling these components across different system sizes (L), evolution times (t), and bond dimensions (χ), a researcher can make informed decisions about resource allocation, choice of simulation parameters, and the potential benefits of implementing more complex hybrid strategies.

Section 8: Conclusion and Future Directions

Recapitulation of the PyQrack-TN Method for OTOC Simulation

This report has detailed a comprehensive and powerful workflow for simulating Out-of-Time-Order Correlators by synergistically combining the PyQrack quantum simulation ecosystem with specialized tensor network libraries. The methodology addresses the

significant computational challenge posed by the exponential complexity of quantum many-body dynamics, particularly for probing chaotic phenomena like information scrambling. The core strategy involves leveraging PyQrack's high-performance, interoperable QrackCircuit class to construct the complex forward-backward evolution circuit required for OTOC calculation. This circuit is then translated, via a crucial CircuitToEinsum utility, into a universal tensor network representation. This representation can be efficiently contracted by dedicated backends, with NVIDIA's cuTensorNet providing state-of-the-art, GPU-accelerated performance.

The strategic advantage of this hybrid approach is its modularity. It allows a researcher to use the best tool for each stage of the problem: the high-level circuit design and synthesis capabilities of frameworks like Qiskit, the versatile and fast circuit management of PyQrack, and the scalable, entanglement-aware computational power of tensor networks. By offloading the most demanding part of the simulation—the contraction of a large, entangled tensor network—to specialized hardware and software, this method pushes the accessible boundaries of system size and evolution time far beyond what is possible with conventional state-vector techniques. Advanced extensions, such as hybrid workflows that combine PyQrack's native near-Clifford simulators with tensor network evolution, further enhance this capability, creating a flexible and powerful platform for cutting-edge research in quantum dynamics.

Outlook: Probing Larger Systems and More Complex Physics

The methodology presented herein is not only a solution for OTOC simulation but also a template for a broader class of advanced quantum simulations. The future development and application of this hybrid paradigm hold significant promise.

A key area for future improvement lies in the integration between PyQrack and tensor network backends. The current reliance on a Python-level CircuitToEinsum function, while effective, could be superseded by a native C++ implementation within the core Qrack library, as envisioned in the project's development roadmap.[18] Such an implementation would dramatically reduce the overhead of the conversion step, enabling the efficient handling of even larger and deeper circuits, and would allow for a more seamless, "transparent" integration where the switch to a tensor network backend could be handled automatically by the simulator based on the circuit's properties.

Furthermore, the workflow can be readily adapted to tackle other computationally demanding problems in quantum computing. The simulation of variational quantum algorithms, such as QAOA and VQE, on large numbers of qubits is one such area. These algorithms often involve circuits with specific structures that may be amenable to tensor network analysis.[40] Another

critical application is the simulation of quantum error correction codes, where tensor network methods have already proven to be a valuable tool for studying the performance of codes like the surface code under realistic noise models.[41] The ability to construct complex code circuits in a framework like PyQrack and then simulate their logical performance using a scalable tensor network backend would be a powerful tool for co-designing future fault-tolerant quantum computers.

Finally, this approach aligns with emerging paradigms in quantum computing like circuit knitting and cutting, where large quantum circuits are broken into smaller sub-circuits that can be executed on smaller quantum devices or classical simulators.[42] The outputs are then recombined using classical post-processing. Tensor networks provide the natural mathematical language for describing this recombination process. A framework like PyQrack, with its ability to manage and convert circuit representations, could serve as a powerful control plane for such hybrid quantum-classical computations. As the field progresses, the fusion of high-performance direct simulation, scalable tensor network methods, and hardware execution will be essential, and the principles outlined in this report provide a robust foundation for these future integrated systems.

Works cited

1. Learning out-of-time-ordered correlators with classical kernel methods | Phys. Rev. B, accessed October 25, 2025, https://link.aps.org/doi/10.1103/PhysRevB.111.144301
2. Efficient Quantum Circuit Simulation by Tensor Network Methods on Modern GPUs - arXiv, accessed October 25, 2025, https://arxiv.org/abs/2310.03978
3. Out-of-time-order correlations and quantum chaos - Scholarpedia, accessed October 25, 2025, http://www.scholarpedia.org/article/Out-of-time-order_correlations_and_quantum_chaos
4. [1703.09435] Out-of-time-order correlators in quantum mechanics - arXiv, accessed October 25, 2025, https://arxiv.org/abs/1703.09435
5. A verifiable quantum advantage - Google Research, accessed October 25, 2025, https://research.google/blog/a-verifiable-quantum-advantage/
6. PyQrack - unitaryhack, accessed October 25, 2025, https://2022.unitaryhack.dev/projects/pyqrack/
7. unitaryfoundation/qrack: Comprehensive, GPU accelerated framework for developing universal virtual quantum processors - GitHub, accessed October 25, 2025, https://github.com/vm6502q/qrack
8. unitaryfoundation/pyqrack: Pure Python bindings for the pure C++11/OpenCL Qrack quantum computer simulator library - GitHub, accessed October 25, 2025, https://github.com/unitaryfoundation/pyqrack
9. PyQrack — PyQrack v0.16.3 documentation, accessed October 25, 2025, https://pyqrack.readthedocs.io/
10. QJIT compilation with Qrack and Catalyst | PennyLane Demos, accessed October

25, 2025, https://pennylane.ai/qml/demos/qrack
11. pennylane-qrack - PyPI, accessed October 25, 2025, https://pypi.org/project/pennylane-qrack/
12. cirqqrack Changelog - Safety, accessed October 25, 2025, https://data.safetycli.com/packages/pypi/cirqqrack/changelog?page=3&
13. Tensor Network, accessed October 25, 2025, https://tensornetwork.org/
14. Quantum Simulation with Hybrid Tensor Networks | Phys. Rev. Lett., accessed October 25, 2025, https://link.aps.org/doi/10.1103/PhysRevLett.127.040501
15. arXiv:2202.07060v1 [quant-ph] 14 Feb 2022, accessed October 25, 2025, https://arxiv.org/abs/2202.07060
16. Simulating 54 qubits with Qrack – on a single GPU - Unitary ..., accessed October 25, 2025, https://unitary.foundation/posts/qrack_report/
17. Simulating 54 qubits with Qrack – on a single GPU - Unitary Foundation, accessed October 25, 2025, https://unitary.foundation/posts/qrack_report
18. Convert QCircuit to cuTensorNet input · Issue #989 · unitaryfoundation/qrack - GitHub, accessed October 25, 2025, https://github.com/unitaryfund/qrack/issues/989
19. Tensor-network quantum circuits | PennyLane Demos, accessed October 25, 2025, https://pennylane.ai/qml/demos/tutorial_tn_circuits/
20. Solutions to Tensor basics — TeNPy 1.0.6.dev34+b32c017 documentation, accessed October 25, 2025, https://tenpy.readthedocs.io/en/latest/toycodes/solution_1_basics.html
21. Simulating a Quantum Circuit - TensorNetwork reference documentation, accessed October 25, 2025, https://tensornetwork.readthedocs.io/en/latest/quantum_circuit.html
22. 7. Quantum Circuits - quimb 1.11.3.dev35+gfbf1f0c20 documentation, accessed October 25, 2025, https://quimb.readthedocs.io/en/latest/tensor-circuit.html
23. cuQuantum - Accelerate Quantum Computing Research - NVIDIA Developer, accessed October 25, 2025, https://developer.nvidia.com/cuquantum-sdk
24. Scaling Quantum Circuit Simulation with NVIDIA cuTensorNet | NVIDIA Technical Blog, accessed October 25, 2025, https://developer.nvidia.com/blog/scaling-quantum-circuit-simulation-with-cutensornet/
25. Getting Started — qrack documentation, accessed October 25, 2025, http://vm6502q.readthedocs.io/en/latest/start.html
26. PyQrack for quantum computing - Feature requests - OpenAI Developer Community, accessed October 25, 2025, https://community.openai.com/t/pyqrack-for-quantum-computing/995345
27. pyqrack - PyPI, accessed October 25, 2025, https://pypi.org/project/pyqrack/
28. Qrack and PyQrack join the Unitary Fund Organization, accessed October 25, 2025, https://unitary.foundation/posts/qrack_joins_uf
29. CDR with Qrack as Near-Clifford Simulator — Mitiq 0.46.0 documentation, accessed October 25, 2025, https://mitiq.readthedocs.io/en/stable/examples/cdr_qrack.html
30. tensornetwork - PyPI, accessed October 25, 2025,

https://pypi.org/project/tensornetwork/
31. quimb · PyPI, accessed October 25, 2025, https://pypi.org/project/quimb/1.4.2/
32. jcmgray/quimb: A python library for quantum information and many-body calculations including tensor networks. - GitHub, accessed October 25, 2025, https://github.com/jcmgray/quimb
33. pytket-cutensornet API documentation, accessed October 25, 2025, https://docs.quantinuum.com/tket/extensions/pytket-cutensornet/
34. pytket-cutensornet - PyPI, accessed October 25, 2025, https://pypi.org/project/pytket-cutensornet/
35. The Badger's Guide to Quantum Circuit Simulation on Campus - QUEST, accessed October 25, 2025, https://quest-lab.cs.wisc.edu/guides/index.html
36. PyQrackIsing - PyDigger, accessed October 25, 2025, https://pydigger.com/pypi/PyQrackIsing
37. cuTN-QSVM: cuTensorNet-accelerated Quantum Support Vector Machine with cuQuantum SDK **The first three authors contributed equally to this work. The order of these names is based on the alphabetical arrangement of the characters in their names. - arXiv, accessed October 25, 2025, https://arxiv.org/html/2405.02630v2
38. John-J-Tanner/ITensor-OTOC-Data: Welcome to the GitHub repository for the datasets produced and used in the paper titled "Learning out-of-time-ordered correlators with classical kernel methods". - GitHub, accessed October 25, 2025, https://github.com/John-J-Tanner/ITensor-OTOC-Data
39. ITensor Examples, accessed October 25, 2025, https://itensor.github.io/ITensors.jl/dev/examples/ITensor.html
40. [2505.23256] A New Scaling Function for QAOA Tensor Network Simulations - arXiv, accessed October 25, 2025, https://arxiv.org/abs/2505.23256
41. [1607.06460] Tensor-Network Simulations of the Surface Code under Realistic Noise - arXiv, accessed October 25, 2025, https://arxiv.org/abs/1607.06460
42. [2410.15080] Quantum-Classical Computing via Tensor Networks - arXiv, accessed October 25, 2025, https://arxiv.org/abs/2410.15080
43. SaashaJoshi/QML-circuit-cutting: Implementation of a Quantum Tensor Network with Circuit Cutting in Qiskit - A project for QHack 2024 - GitHub, accessed October 25, 2025, https://github.com/SaashaJoshi/QML-circuit-cutting