arXiv:2603.14641v2 [quant-ph] 23 Sep 2026

GPU-Accelerated Quantum Simulation of Stabilizer Circuits

Muhammad Osama, Dimitrios Thanos, and Alfons Laarman

Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands

We introduce new parallel algorithms for efficiently simulating stabilizer (Clifford) circuits on GPUs, with a focus on data-parallel tableau evolution and scalable handling of projective measurements. Our approach reformulates key bottlenecks in stabilizer simulation—such as Gaussian elimination and measurement updates—into GPU-tailored primitives that eliminate sequential dependencies and maximize memory coalescing. We implement these techniques in QuaSARQ, a GPU-accelerated stabilizer simulator designed for large qubit counts and many-shot sampling. Across a broad benchmark suite reaching 180,000 qubits and depth 1,000 (roughly 130M gates), QuaSARQ shows substantial runtime improvements, with up to 105× speedup, and over 80% energy reduction on demanding instances. Moreover, QuaSARQ consistently outperforms Stim, a state-of-the-art CPU-optimized stabilizer simulator, as well as Qiskit-Aer (CPU/GPU), Qibo, Cirq, and PennyLane. Finally, QuaSARQ exhibits a significant advantage in many-shot sampling on large workloads. These results demonstrate that our parallel algorithms can significantly advance the scalability of stabilizer-circuit simulation, particularly for workloads involving extensive measurements and sampling.

1 Introduction

Classical simulation of quantum circuits plays an instrumental role in the development of quantum computing. Simulators support algorithm design, verification, and optimization, enabling tasks such as equivalence checking [1], circuit compilation and synthesis [2, 3], and state tomography [4, 5]. They also operate within hybrid quantum-classical workflows, where they are frequently used for pre-optimization routines [6], adaptive measurement protocols in measurement-based quantum computing [7], and variational algorithms [8].

Within this landscape, *stabilizer* (*Clifford*) circuits form a foundational class. Their rich algebraic structure allows efficient classical simulation via the *stabilizer-tableau formalism* [9], yielding an $\mathcal{O}(n^2)$ update cost and enabling large-scale applications in quantum error correction, communication, and simulation. Recent GPU-accelerated simulators such as IBM's Qiskit-Aer [10] have demonstrated that GPUs can deliver substantial speedups. Meanwhile, Stim – a state-of-the-art, CPU-optimized stabilizer simulator [11] – introduced a series of algorithmic innovations for measurement and sampling. However, these methods

Muhammad Osama: m.o.mahmoud@liacs.leidenuniv.nl
Dimitrios Thanos: d.thanos@liacs.leidenuniv.nl
Alfons Laarman: a.w.laarman@liacs.leidenuniv.nl

have not yet been fully adapted to exploit modern GPU architectures. Stim's paper itself reports a GPU experiment that did not achieve the desired results.[1]

In prior work, we showed that GPUs can dramatically accelerate certain stabilizer-circuit tasks, most notably with a GPU-based equivalence checker [12]. Yet equivalence checking does not require full support for projective measurements, nor does it handle extensive sampling workloads. Measurements are pivotal to a wide range of quantum protocols, including error correction [13, 14], randomized benchmarking [15], and hybrid quantum-classical algorithms [16]. Efficiently supporting them at scale on GPUs remains a significant challenge. In particular, naive implementations of measurement-induced tableau updates, such as Gaussian elimination [9, 11], introduce sequential dependencies and irregular memory access patterns that limit parallelism.

Contributions. In this work, we address these challenges by introducing new parallel algorithms for tableau evolution and measurement operations on GPUs, designed to align with GPU memory hierarchies and minimize control divergence.

Our main contributions are as follows:

- ⋆ A host-side parallel-gate and measurement scheduler to maximize GPU parallelism. Mutually independent gates are arranged into maximal parallel windows, enabling large batches of Clifford gates to be applied simultaneously (section 4.2).
- ⋆ A GPU-oriented data-structure with a dual tableau layout. A column-major layout enables fast gate application, while an in-place word/bit transpose provides a row-major layout optimized for measurement operations, ensuring coalesced memory access throughout tableau evolution (section 4.3).
- ⋆ A tree-based tableau evolution method for applying Clifford gates. We reduce atomic contention and improve occupancy by aggregating sign updates in shared memory via a binary-tree reduction scheme (section 4.4).
- ⋆ Dense stream compaction [17] of tableau pivots (indices of anti-commuting Pauli strings). By packing pivots into contiguous memory, we eliminate warp divergence and ensure coalesced memory access during Gaussian elimination (section 4.7).
- ⋆ A three-pass prefix-XOR formulation, Gaussian elimination. We transform CX injections [11] into fully parallel prefix operations, eliminating sequential dependencies in the measurement process (section 4.8).

We have implemented these algorithms in QuaSARQ, a GPU-accelerated stabilizer simulator designed for large-scale Clifford circuits and many-shot sampling. QuaSARQ completes 177 heavy-suite circuits within 72 hours, compared to Stim's 125 circuits in 132 hours, reduces energy consumption by more than 80% on demanding instances, and supports thousands of projective measurements at 180,000 qubits. Across a broad benchmark suite, QuaSARQ consistently outperforms Stim, Qiskit-Aer (CPU/GPU), Qibo, Cirq, and PennyLane. In addition, for many-shot sampling, QuaSARQ uses a GPU-based sampler which shows a clear advantage over Stim on our 1,024-shot benchmarks. These results show that GPU-oriented algorithms can significantly advance the scalability of stabilizer simulation, particularly for workloads involving extensive measurements and sampling. The remainder of this paper describes our parallel algorithms, implementation design, and benchmarking in detail.

[1] See "GPU Experiment Failure" in [11, p. 14].

Related Work. Classical simulation of quantum circuits spans a large body of research, ranging from generic state-vector simulators to specialized algorithms for Clifford circuits, tensor-network contractions, and hybrid approaches. GPU-based simulators have gained increasing attention as quantum workloads grow in scale, but most existing systems do not fully exploit the algebraic structure of stabilizer circuits.

State-vector simulators. General-purpose simulators such as Cirq [18], PennyLane [19], and Qiskit-Aer [10], and Qibo [20] provide GPU backends capable of accelerating dense state-vector operations. More recently, NVIDIA's cuQuantum SDK [21] has introduced high-performance GPU primitives for simulation and tensor-network contraction, enabling speedups for circuits that fit within GPU memory. These simulators support arbitrary quantum circuits but exhibit exponential memory scaling, limiting simulations to approximately 30 qubits on high-end hardware. Although their GPU kernels are optimized for linear-algebra operations, they do not exploit tableau structure. As a result, they are typically far slower than specialized stabilizer simulators, even for modest qubit counts.

Decision Binary Diagram (BDD)-based simulators. An alternative line of work uses graph-based or decision-diagram representations to compress quantum states. Tools such as BQSim [22] and earlier BDD-based simulators exploit structural regularities and repeated sub-patterns to avoid explicit state-vector storage. These techniques can achieve dramatic compression on circuits with high redundancy, but their performance can degrade significantly when applied to dense Clifford circuits.

Weighted Model Counting and Symbolic approaches. Another line of work is simulation through symbolic encodings. Weighted Model Counting (WMC) methods, such as Quokka [23], translate Clifford+T circuits into conjunctive normal form and use SAT or #SAT/WMC solvers to answer decision or probability queries. However, they suffer from rapid formula growth as circuit size increases. ZX-calculus simulators, such as QuiZX [24], apply rewrite rules to represent and simplify circuits heuristically. This approach is powerful for circuits with abundant algebraic structure but can become unpredictable for large or less structured Clifford circuits.

Stabilizer-tableau simulators. Classical simulation of Clifford circuits traces back to the Gottesman–Knill theorem [25], with practical tableau-based implementations developed in CHP [9] and later improved in Qiskit. The most recent advance is Stim [11], a high-performance CPU-based stabilizer simulator that introduced algorithmic optimizations for measurements, Gaussian elimination, and fast sampling. Stim represents the current state of the art in CPU efficiency. However, its techniques are optimized for vectorized instruction sets and branch prediction on CPUs and have not been adapted to fully exploit GPU parallelism. The single GPU experiment reported in the Stim paper achieved very limited speedup due to irregular memory access patterns and sequential dependencies in measurement and tableau updates.

GPU-based stabilizer simulation. Despite the natural parallelism in tableau operations, few stabilizer simulators target GPUs. The GPU backend of Qiskit-Aer supports Clifford gates but does not restructure measurement operations to avoid sequential operations, leading to limited scalability. Earlier GPU work has focused on specific subtasks, such as our own GPU equivalence checker [12], that do not require full support for projective measurements or high-volume sampling. To our knowledge, there is no existing GPU simulator that addresses the key challenges of tableau evolution, projective measurements, sampling fully in parallel.

In a different direction, PACOX [26] investigates FPGA acceleration of bit-level Pauli-string computations.

2 Preliminaries

2.1 Quantum States and Gates

A qubit is described by a linear combination of the *basis states* $|0\rangle$ and $|1\rangle$ that can be identified with the 2-vectors $[1 \quad 0]^T$ and $[0 \quad 1]^T$, respectively. These states are analogous to the classical bits 0 and 1, but in the quantum case, a single qubit can take on the value of the linear combination $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ where α_0, α_1 are complex numbers and $|\psi\rangle$, like any quantum state, must be normalized to be a unit vector (i.e., $|\alpha_0|^2 + |\alpha_1|^2 = 1$). The linear combination of basis states is also known as *superposition*.

The operation for combining n single-qubit states $|\psi\rangle_1, \cdots, |\psi\rangle_n$ into an n-qubit state $|\psi\rangle_1 \otimes \cdots \otimes |\psi\rangle_n$ is known as the *tensor product*. It is denoted by $\otimes$ and is typically known in linear algebra as the Kronecker product. The n-qubit states have the form $\sum_{\ell \in \{0,1\}^n} \alpha_\ell |\ell\rangle$ where $|\ell\rangle$ is the conventional way of denoting a string of multiple tensored single-qubit states, i.e., we suppress the $\otimes$ symbol and say $|1011\rangle$ instead of $|1\rangle \otimes |0\rangle \otimes |1\rangle \otimes |1\rangle$. If a quantum state cannot be factored into a tensor product of its individual subsystems, then we say it is an *entangled* state. For example, the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled, while the state $|1001\rangle$ is not.

In general, a quantum state $|\psi\rangle$ of n qubits can be written as a vector of 2^n complex numbers: $|\psi\rangle = [\alpha_0, \alpha_1, \cdots, \alpha_{2^n}]^T$. For every state, there is an *adjoint* defined as $(|\psi\rangle^*)^T$ and denoted by $\langle\psi|$. The "$*$" operator is a complex conjugation, meaning that we replace every entry with its complex conjugate, then transpose the vector (i.e., transforming it into a row vector). The normalization requirement, can be now reformulated into $\langle\psi| \cdot |\psi\rangle = 1$. This product is also known as *inner product* and can be performed between distinct states.

Quantum states can be manipulated by two types of operations: quantum gates and quantum measurements. Quantum gates are *unitary* matrices, representing reversible linear transformations denoted as $U \in \mathbb{C}^{2^n \times 2^n}$. These gates preserve both the norm and the inner product of quantum states. A key property of unitary gates is that $U^{-1} = U^\dagger$, where $U^\dagger = (U^*)^T$ represent the conjugate transpose of U. The effect of a gate U acting on a state $|\psi\rangle$, can be calculated using matrix-vector multiplication $U \cdot |\psi\rangle$.

An important gate set is the Clifford gate set (CX is also called $CNOT$):

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

It is (quantum) non-universal and classically simulatable as explained in the following sections. Extending the Clifford gate set with a $T = \left[\begin{smallmatrix} 1 & 0 \\ 0 & \omega \end{smallmatrix}\right]$ gate, where $\omega = \sqrt{i} = \frac{1+i}{\sqrt{2}}$, yields a universal gate set [27].

To classically simulate circuits consisting of Clifford gates, we will use the following Pauli gate set to represent symmetries of the encountered states.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

2.2 Circuit Simulation

Classical simulation of quantum circuits is typically categorized into two distinct tasks: *strong* and *weak* simulation. In this work, we focus on weak simulation, which is essential for systems involving extensive measurements.

Weak simulation, also refereed to as *sampling*, more closely mimics the behavior of quantum hardware. Rather than calculating exact probabilities as in strong simulation,

the goal is to produce a random bitstring x derived from the probability distribution defined by the state $|\psi\rangle$.

Definition 1 (Weak Simulation)**.** *Given a quantum circuit $\mathcal{C}$ and an initial state $|0\rangle^{\otimes n}$, weak simulation produces an output $x \in \{0,1\}^n$ such that the probability of obtaining x is exactly $p(x) = |\langle x|\mathcal{C}|0\rangle^{\otimes n}|^2$.*

In our simulator QuaSARQ, we implement weak simulation through two different GPU optimized algorithms:

- Single-shot simulation: For scenarios where the state tableau must be updated following a projective measurement, we design and implement a parallel algorithm for performing Gaussian Elimination (GE).
- Many-shot sampling: for applications requiring a high volume of samples or shots, our simulator implements *Pauli frames* [28, 29, 11] to avoid the overhead of GE being repeatedly performed for every sample. Pauli frames allow the simulator to generate multiple samples efficiently by tracking Pauli errors and updates without full tableau restructuring for every shot.

Next, we discuss simulation algorithms for the specific case of stabilizer circuits, a non-universal class of quantum circuits that are efficiently simulatable by classical algorithms.

2.3 Stabilizer Circuits

Stabilizer circuits, also known as *Clifford circuits*, are quantum circuits composed of Clifford gates. The quantum states generated by applying these circuits to the computational basis $|0\rangle^{\otimes n}$ are called *stabilizer states*. Interestingly, stabilizer circuits can be simulated efficiently by a classical algorithm [30, 9].

The *Pauli group* on n qubits is defined as the set of operators of the form $\mathcal{P}_n \triangleq i^k P_1 \otimes \cdots \otimes P_n$ with $P_i \in \{I, X, Y, Z\}$ and $k \in \{0,1,2,3\}$, i.e., the phase of *Pauli string* is one of $\{\pm 1, \pm i\}$.For simplicity, the tensor product is often suppressed, and n-Pauli strings are written in shorthand. For example, $X \otimes I \otimes Y \otimes X \otimes Z$ can be compactly written as $XIYXZ$. The *Clifford group* on n qubits consists of the unitary operators V that normalize the Pauli group, that is, $\mathcal{C}_n = \{V \in \mathbb{C}^{2n \times 2n} \mid \forall P \in \mathcal{P}_n \colon VPV^\dagger \in \mathcal{P}_n\}$.The Clifford group is generated by the Clifford gate set $\{H, P, CX\}$ [31]. Each stabilizer state, generated by a stabilizer circuit, can be uniquely encoded by a subgroup of the Clifford group, requiring exactly n independent generators, where each generator is an n-Pauli string.

Stabilizer states are called this way because they are *stabilized* by certain operators. A unitary operator U stabilizes a state iff $U|\psi\rangle = |\psi\rangle$. For example, the Pauli string $-IZ$ stabilizes the state $|0\rangle \otimes |1\rangle = |01\rangle$. A state can be stabilized by multiple Pauli strings; for instance, $|01\rangle$ is also stabilized by II, ZI and $-ZZ$. These Pauli strings generate the maximal commutative subgroup $\mathcal{H}_{\max} = \{II, -IZ, ZI, -ZZ\}$, which is the stabilizer group of $|01\rangle$. Each maximal commutative subgroup of the Pauli group stabilizes exactly one stabilizer state [31].

Although the stabilizer group for an n-qubit stabilizer state has 2^n elements, only n Pauli strings are needed to generate it. For example, $\mathcal{H}_{\max}$ can be generated by the two generators in multiple ways, expressed formally by: $\mathcal{H}_{\max} = \langle -IZ, -ZZ\rangle = \langle -IZ, ZI\rangle$. This property allows stabilizer states to be efficiently and succinctly represented by a data structure such as the tableau formalism.

The density matrix of the state $|\mathcal{T}\rangle$ represented by a tableau $\mathcal{T}$ consisting of generators $\mathrm{G}_1, \ldots, \mathrm{G}_n$, can be established directly as:

$$|\mathcal{T}\rangle\langle\mathcal{T}| \quad = \quad \prod_{i\in[n]} \frac{I^{\otimes n} + \mathrm{G}_i}{2} \quad = \quad \frac{1}{2^n} \sum_{P\in\langle \mathrm{G}_1,\ldots,\mathrm{G}_n\rangle} P. \tag{1}$$

2.4 Tableau Formalism

Throughout this paper, we refer to the conventional forward-evolving stabilizer tableau as the *state tableau*. We use the term *tableau evolution* to denote the application of gate-level update rules to the stabilizer tableau as the circuit executes.

A stabilizer state's generators can be described by a binary table known as a *stabilizer tableau* [30]. The tableau encodes a Pauli operator $P \in \{I, X, Y, Z\}$ with two bits x, z such that $P = Z^z X^x$ up to global phase (i.e., $Y = iZX \equiv ZX$). The (n-qubit) generators are then encoded with bits $x_1, \ldots, x_n, z_1, \ldots, z_n, s$, where s is a bit recording the phase as $(-1)^s$.[2] The n generators then form the tableau as follows.

$$\mathcal{T} = \left(\begin{array}{ccc|ccc|c} x_{11} & \cdots & x_{1n} & z_{11} & \cdots & z_{1n} & s_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & \cdots & x_{nn} & z_{n1} & \cdots & z_{nn} & s_n \end{array}\right) \tag{2}$$

For example, the tableau below encodes the generators $\{ZII, \text{-}IZI, \text{-}IXY\}$.

$$\left(\begin{array}{ccc|ccc|c} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}\right) \cong \left\langle \begin{array}{c} ZII \\ -YZI \\ -IXY \end{array} \right\rangle \tag{3}$$

X part | Z part | Signs

To update $\mathcal{T}$ after applying a Clifford gate on a single qubit q or on a pair (c, t), the following rules are applied to every row index $i \in \{1, \ldots, n\}$ [30]:

$$\begin{aligned} H_q: &\quad (x_{iq},\ z_{iq},\ s_i) \leftarrow (z_{iq}, & x_{iq}, & & s_i \oplus (x_{iq} \wedge z_{iq})), \\ P_q: &\quad (x_{iq},\ z_{iq},\ s_i) \leftarrow (x_{iq}, & z_{iq} \oplus x_{iq}, & & s_i \oplus (x_{iq} \wedge z_{iq})), \\ CX_{c,t}: &\quad (x_{it},\ z_{ic},\ s_i) \leftarrow (x_{it} \oplus x_{ic}, & z_{ic} \oplus z_{it}, & & s_i \oplus (x_{ic} \wedge z_{it} \wedge \neg(x_{it} \oplus z_{ic}))). \end{aligned} \tag{4}$$

Equation (4) is expressed in the standard state-tableau formulation, where Clifford gates are indexed by qubits and applied column-wise across all generators. Stim uses the operator-tableau representation, which, while very similar to the state-tableau, goes beyond encoding states. The operator-tableau is the state-tableau extended by the *destabilizer* generators [9]. This doubles the number of rows and yields a structure isomorphic to the Pauli group modulo global phase.

The column update rules remain identical to those of the state tableau, except that they apply to twice as many generators. This isomorphism originates in [30], and is made explicit in later works such as [9]. Both representations are symplectic encodings, whose structure is rooted in symplectic geometry [32].

The following example shows how gates can be applied in parallel to $\mathcal{T}$ when acting on independent qubits, hence affecting independent columns.

[2] Since all stabilizers in the group mutually commute, the phase $\pm i$ does not occur, because $(\pm iP)^2 = -I^{\otimes n}$ (which only commutes with itself).

Example 1. *Given a 3-qubit system initialized to the* $|000\rangle$ *state, assume that the gates* H_1 *and* P_3 *are applied in order, to the qubits 1 and 3 respectively. The affected columns, according to eq. (4), are updated and highlighted by colors:*

$$\left[\begin{array}{ccc|c|ccc|c} 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right] \xRightarrow{H_1} \left[\begin{array}{ccc|c|ccc|c} 1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right] \xRightarrow{P_3} \left[\begin{array}{ccc|c|ccc|c} 1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right] \tag{5}$$

2.5 Measurements in the Stabilizer Formalism

A quantum measurement projects a qubit onto a chosen basis and yields a classical outcome determined probabilistically, irreversibly collapsing superpositions and leaving the system in an eigenstate of the measured observable. In this work, we restrict attention to *projective Pauli measurements*, and in particular to computational-basis (Z-basis) measurements of individual qubits. Measurements of multiple qubits are performed by repeating the same single-qubit procedure for each measured qubit. More general measurements, such as POVMs [31], are out of scope.

A projective measurement is specified by a set of projectors $\{P_m\}$ corresponding to the possible outcomes. For a computational-basis measurement, these projectors are $P_m = |m\rangle\langle m|$ (with $m \in \{0,1\}$). If the system is in state $|\phi\rangle$, the probability of observing outcome m is given by the Born rule,

$$p(m) = \langle\phi| P_m |\phi\rangle = \langle\phi| \left(|m\rangle\langle m|\right) |\phi\rangle = \left|\langle m \mid \phi\rangle\right|^2.$$

The computational-basis measurement is equivalently a measurement of the Pauli-Z operator, whose eigenstates are $|0\rangle$ and $|1\rangle$, satisfying $Z|0\rangle = +1|0\rangle$, $Z|1\rangle = -1|1\rangle$, and $Z = |0\rangle\langle 0| - |1\rangle\langle 1|$. Measuring qubit q in the computational basis thus corresponds to measuring the Pauli observable

$$Z_q \equiv I^{\otimes q} \otimes Z \otimes I^{\otimes(n-q)}.$$

The two measurement outcomes correspond to the eigenvalues $+1$ and -1, which we encode as a classical bit

$$m \in \{0,1\} \quad \text{with eigenvalue} \quad (-1)^m.$$

Equivalently, writing the pre-measurement state as a density matrix $\rho = |\phi\rangle\langle\phi|$, the outcome probabilities can be expressed as $p(m) = \mathrm{Tr}(\rho P_m)$. Since all Pauli operators are traceless except the identity, this trace extracts precisely the coefficient of the projector $|m\rangle\langle m|$ in ρ. For a computational-basis measurement on qubit q, this gives

$$p(m=0) = \langle\phi| \left(|0\rangle\langle 0|_q\right) |\phi\rangle, \qquad p(m=1) = \langle\phi| \left(|1\rangle\langle 1|_q\right) |\phi\rangle.$$

Equivalently, in terms of the expectation value of Z_q,

$$p(m) = \frac{1 + (-1)^m \langle\phi| Z_q |\phi\rangle}{2}.$$

For a more elaborate introduction to quantum measurements, see [31].

Deciding whether the measurement is deterministic. Let $|\mathcal{T}\rangle$ be the stabilizer state represented by a tableau $\mathcal{T}$ with stabilizer generators $\mathrm{S}_1, \ldots, \mathrm{S}_n$, and let $\rho = |\mathcal{T}\rangle\langle\mathcal{T}|$. The quantity $\langle Z_q\rangle = \mathrm{tr}(Z_q\rho)$ fully determines whether the outcome is deterministic or probabilistic. For stabilizer states $\langle Z_q\rangle \in \{+1, -1, 0\}$, it can be characterized purely by the stabilizer group $\langle \mathrm{S}_1, \ldots, \mathrm{S}_n\rangle$:

$$\mathrm{tr}(Z_q\rho) = \begin{cases} +1 & \text{if } +Z_q \in \langle \mathrm{S}_1, \ldots, \mathrm{S}_n \rangle, \\ -1 & \text{if } -Z_q \in \langle \mathrm{S}_1, \ldots, \mathrm{S}_n \rangle, \\ 0 & \text{if } Z_q \text{ anti-commutes with at least one generator.} \end{cases} \tag{6}$$

Interpreting this in terms of the classical measurement outcome m:

- If $\langle Z_q \rangle = +1$, then $p(m = 0) = 1$ (deterministic outcome $m = 0$).
- If $\langle Z_q \rangle = -1$, then $p(m = 1) = 1$ (deterministic outcome $m = 1$).
- If $\langle Z_q \rangle = 0$, then $p(m = 0) = p(m = 1) = \frac{1}{2}$ (uniformly random).

At the tableau level, testing commutation with Z_q is straightforward. A stabilizer generator has an X (or Y) on qubit q exactly when its x-bit at column q is 1, and this is precisely the condition under which it anti-commutes with Z_q. Hence the measurement of q is probabilistic iff *at least one stabilizer row has* $x_{iq} = 1$.

Updating the tableau after measurement. If the measurement is *deterministic*, the post-measurement quantum state is unchanged , since it is already an eigenstate of Z_q.

If the measurement is probabilistic, the stabilizer description must be updated. Let p (the pivot) be any index such that S_p anti-commutes with Z_q (i.e., S_p has $x_{pq} = 1$). A standard stabilizer-update rule proceeds as follows:

1. For every $i \neq p$ such that S_i anti-commutes with Z_q, replace $\mathrm{S}_i \leftarrow \mathrm{S}_i\mathrm{S}_p$ so that S_i now commutes with Z_q.
2. Sample a fresh random bit $m \in \{0, 1\}$.
3. Replace the anti-commuting generator (S_p) by the measured observable with the sampled sign:
$$\mathrm{S}_p \leftarrow (-1)^m Z_q.$$

After this update, all stabilizer generators commute with Z_q, and the resulting tableau represents the post-measurement stabilizer state conditioned on the sampled outcome m.

In [9], it is shown that GE can be performed more efficiently by introducing *destabilizer* generators, which are Pauli operators that each anti-commute with exactly one stabilizer generator. The tool CHP implements the improved algorithm, which requires time $\mathcal{O}(n^2)$. Next, we explain the CHP algorithm in detail.

The CHP algorithm [9]. The CHP algorithm operates on the forward-evolving *state tableau*, explicitly maintaining both stabilizer and destabilizer generators. We denote destabilizer and stabilizer matrices by $X_\mathrm{D}, Z_\mathrm{D}$ and $X_\mathrm{S}, Z_\mathrm{S}$, respectively, where the subscripts D and S are formatted in Roman font to avoid ambiguity with mathematical indices. Further, destabilizer bits are marked with a prime (e.g., x'_{ij}, z'_{ij}) to distinguish them from stabilizer bits. The extended tableau representation is

$$\mathcal{T} = \left(\begin{array}{c|c|c} X_\mathrm{D} & Z_\mathrm{D} & S_\mathrm{D} \\ \hline X_\mathrm{S} & Z_\mathrm{S} & S_\mathrm{S} \end{array}\right) = \left(\begin{array}{ccc|ccc|c} x'_{11} & \cdots & x'_{1n} & z'_{11} & \cdots & z'_{1n} & s'_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x'_{n1} & \cdots & x'_{nn} & z'_{n1} & \cdots & z'_{nn} & s'_n \\ \hline x_{11} & \cdots & x_{1n} & z_{11} & \cdots & z_{1n} & s_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & \cdots & x_{nn} & z_{n1} & \cdots & z_{nn} & s_n \end{array}\right) \tag{7}$$

If the initial state is a product state, for example $|{+}01\rangle$, one valid initialization of the tableau is shown in eq. (8). In this example, the X-part of the stabilizer rows coincides with the Z-part of the destabilizer rows, and vice versa, reflecting the standard CHP initialization.

$$\begin{array}{r} \text{Destabilizers} \\ \\ \\ \text{Stabilizers} \\ \\ \\ \end{array} \left\{ \begin{array}{ccc|ccc|c} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right) \cong \left\langle \begin{array}{c} ZII \\ IXI \\ -IIX \end{array} \right\rangle \left\langle \begin{array}{c} XII \\ IZI \\ -IIZ \end{array} \right\rangle \tag{8}$$

$$\underbrace{}_{X\text{ part}} \quad \underbrace{}_{Z\text{ part}} \quad \underbrace{}_{\text{Signs}}$$

We now describe the CHP procedure for measuring qubit q in the computational basis, i.e., the observable Z_q. According to eq. (6), there are two possible cases: (1) the outcome is deterministic (the state does not change), or (2) the outcome is probabilistic (the state collapses). To determine whether the measurement is deterministic, we check whether any stabilizer generator anti-commutes with Z_q. As discussed above, this occurs iff the q-th column of X_{S} contains a 1. If no such row exists, the outcome is deterministic; otherwise it is probabilistic.

Probabilistic case. There is at least one $p \in [n]$ such that S_p anti-commutes with Z_q. We call $\tilde{p}$, the smallest such index. We update the state as follows:

1. For the indices $i \in [n] \setminus \{\tilde{p}\}$ with S_i anti-commuting with Z_q, update $\mathrm{S}_i \leftarrow \mathrm{S}_i\mathrm{S}_{\tilde{p}}$ and $\mathrm{D}_i \leftarrow \mathrm{D}_i\mathrm{D}_{\tilde{p}}$.
2. Set $\mathrm{D}_{\tilde{p}}$ to $\mathrm{S}_{\tilde{p}}$.
3. Flip an unbiased coin $m \in \{0, 1\}$. Set $\mathrm{S}_{\tilde{p}}$ to $(-1)^m Z_q$. Return outcome m.

In CHP, enforcing commutation requires explicit multiplication of Pauli generators, implemented as bitwise XOR over the full tableau rows. Since each such operation touches $\mathcal{O}(n)$ bits and may be applied to up to $\mathcal{O}(n)$ rows, this leads to an overall quadratic cost.

Deterministic case. For all $p \in [n]$, we have that S_p commutes with Z_q. The tableau requires no update in absence of the collapse of a superposition. We may avoid the GE by using the destabilizers to check membership of $\pm Z_q$ in the full stabilizer group $\langle \mathrm{S}_1, \ldots, \mathrm{S}_n \rangle$ as required by eq. (6) in this case. As explained in [9], we can do this by observing that GE yields $\pm Z_q = \prod_{i \in [n]} \mathrm{S}_i^{c_i}$ for some assignments to $c_i \in \{0, 1\}$. The destabilizers can identify the assignments directly since $c_i = \prod_{j \in [n]} (\mathrm{D}_i\mathrm{S}_j)^{c_j} = \mathrm{D}_i \prod_{j \in [n]} \mathrm{S}_j^{c_j} = \mathrm{D}_i \pm Z_q$.

1. Let $A \subseteq [n]$ be the largest index set marking anti-commuting destabilizers, i.e., D_i and Z_q anti-commute iff $i \in A$.
2. Let $(-1)^m Z_q = \prod_{i \in A} \mathrm{S}_i$. Return outcome m.

Gottesman and Aarsonson [9] describe in detail how this CHP algorithm functions on the level of the bits of the tableau.

Innovations introduced by Stim simulator [11]. The state-of-the-art tool for simulating measurements in stabilizer circuits is Stim [11] simulator. Stim resolves measurements using the procedure detailed in algorithm 1. The algorithm builds and extends on the ideas from CHP [9]. We briefly outline the innovations of Stim that are most relevant to our GPU adaptation.

Instead of explicitly multiplying Pauli generators as in CHP, Stim replaces these row multiplications with conditional CX injections, which implement the same row-addition effect in the tableau during GE. These injections are applied only for probabilistic measurements and significantly reduce computational cost. QuaSARQ adopts the same measurement approach while operating on a data structure tailored for GPU architectures (see section 4.3).

The key steps of algorithm 1 are referenced by their line numbers (e.g., CX injection in line 18). We use bracket notation to access individual generator rows of the tableau; for example, $Z_S[i]$ denotes the i-th stabilizer Z-row, and similarly $X_S[i]$, $X_D[i]$, and $Z_D[i]$.

Algorithm 1: Conceptual CPU-based probabilistic measurement in Stim.

Input: $\mathcal{T} := (X_D, X_S, Z_D, Z_S, S_D, S_S)$, q, n // $\mathcal{T}$ (Tableau), q (qubit index), n (Number of qubits).
Output: m // Meaurement outcome

1 ***function*** ProbabilisticMeasurement($\mathcal{T}, q, n$)**:**
2 $p = \min\big(\{\, i \in [n] \mid X_S[i,q] = 1\} \cup \{-1\}\big)$ // Find a stabilizer p that anti-commutes with $Z[q]$
3 **if** $p \neq -1$ **then** // Collapsing into Z is needed.
4 GaussianEliminationByCX($\mathcal{T}, p, q, n$)
5 SwapAntiCommuting($\mathcal{T}, p, q, n$)
// If random measurement differs from observable's sign, invert signs by injecting an X gate.
6 **if** $S_S[q] \neq$ random(0, 1) **then**
7 XGate(S_D, $Z_D[p]$), XGate(S_S, $X_D[p]$)
8 **return** $m \leftarrow S_S[q]$
/* Perform GE over stabilizer generators that anti-commute with q. */
9 ***function*** GaussianEliminationByCX($\mathcal{T}, p, q, n$)**:**
10 **for** $t = p+1$ **to** $n-1$ **do** // Targets must follow a strict order in $[p+1, n-1]$
11 **if** $X_S[t,q] = 1$ **then**
12 CXGate(S_D, $Z_S[p]$, $Z_D[p]$, $Z_S[t]$, $Z_D[t]$), CXGate(S_S, $X_S[p]$, $X_D[p]$, $X_S[t]$, $X_D[t]$)
/* Swap anti-commuting stabilizer generator for one that commutes with q. */
13 ***function*** SwapAntiCommuting($\mathcal{T}, p, q, n$)**:**
14 **if** $X_D[p,q] = 1$ **then** // Y case.
15 PInvGate(S_D, $Z_S[p]$, $Z_D[p]$), PInvGate(S_S, $X_S[p]$, $X_D[p]$)
16 **else** // X case.
17 HGate(S_D, $Z_S[p]$, $Z_D[p]$), HGate(S_S, $X_S[p]$, $X_D[p]$)
/* x: stabilizer row, z: destabilizer row, s: sign vector */
18 ***function*** CXGate(s, x_c, z_c, x_t, z_t): $\big(s,\ x_t,\ z_c\big) \leftarrow \big(s \oplus \big(x_c \wedge z_t \wedge \neg(z_c \oplus x_t)\big),\ x_t \oplus x_c,\ z_c \oplus z_t\big)$
19 ***function*** PInvGate(s, x, z): $\big(s,\ x,\ z\big) \leftarrow \big(s \oplus (x \wedge \neg z),\ x \oplus z,\ z\big)$
20 ***function*** HGate(s, x, z): $\big(s,\ x,\ z\big) \leftarrow \big(s \oplus (x \wedge z),\ z,\ x\big)$
21 ***function*** XGate(s, z): $\big(s,\ z\big) \leftarrow \big(s \oplus z,\ z\big)$

Stim introduces the following implementation-level optimizations over CHP [9].

- It uses a Cache-friendly, bit-packed layout with 256-bit SIMD instructions where the tableau is stored in a bit-packed format that maximizes contiguous memory access. Tight loops apply 256-bit AVX2 instructions to entire rows or columns in one go, reducing the per-qubit Clifford-gate overhead.

- It exploits *Pauli frames*, a classical record of bit-flip and phase-flip corrections for each qubit. Pauli frames can be used for both the simulation of errors and the randomness caused by measurements. Instead of re-updating the tableau as a whole for each error event or probabilistic occurrence, the tableau is calculated once (reference shot) and the appropriate bit- and phase-flips are propagated by Clifford conjugation.

2.6 GPU Programming

CUDA [33] is a programming interface that enables general-purpose programming for a GPU. It has been developed and continues to be maintained by NVIDIA since 2007. In this work, we use CUDA with C++. Therefore, we use CUDA terminology when we refer to thread and memory hierarchies.

Figure 1 gives an overview of a GPU architecture. A GPU consists of a finite number of *streaming multiprocessors* (SM), each containing hundreds of *cores*. For instance, the RTX 4090, which we used for this work, has 128 SMs containing 16,384 cores in total. A GPU computation can be launched in a program by the *host* (CPU side of a program), which calls a GPU function called a *kernel*, executed by the *device* (GPU side of a program).

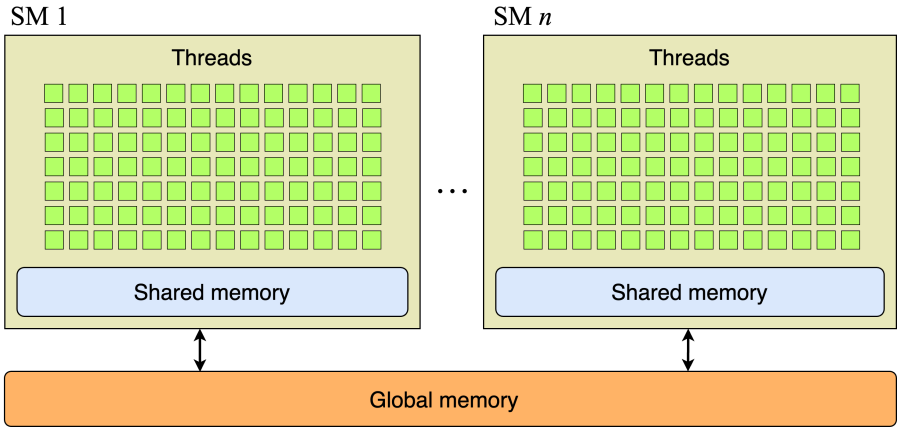


Figure 1: GPU architecture [34, 35].

When a kernel is called, the number of threads needed to execute it is specified. According to the CUDA paradigm [33], these threads are partitioned into thread *blocks*, i.e., 3-dimensional (3D) vectors grouping threads up to 1,024. Each thread block is assigned to an SM. All threads together form a 3D *grid* where threads and blocks can be indexed by a 1D, 2D, or 3D unique identifier (ID) accessible within the kernel. This ID works similarly as the ($\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$) coordinates in 3D space. With this ID, different threads in the same block can process multidimensional data (e.g., X and Z matrices).

Within each block, threads are further grouped into fixed-size units of 32 threads called *warps*, which are the fundamental units of scheduling and execution on the GPU. All threads in a warp execute the same instruction in lockstep. If threads within a warp follow different control-flow paths due to conditional branches (*warp divergence*), execution is serialized across the different paths, causing stalls and degrading overall kernel performance. Avoiding warp divergence is therefore critical for achieving high throughput on GPUs.

Constant memory Shared memory is on-chip SRAM visible only to threads within the same block. Each SM provides a limited amount (tens of kilobytes) of shared memory; it has much lower latency than global memory and is ideal for inter-thread communication and tile-based algorithms. Registers are the fastest storage, private to each thread and used for local variables. Overuse of registers can limit occupancy, so we balance register usage against shared-memory needs.

Memory Hierarchy. Concerning the memory hierarchy, a GPU has multiple types of memory:

- *Global memory* is the large, high-bandwidth DRAM that both CPU and GPU threads can access. It has relatively high latency, so good performance requires coalesced,

aligned accesses. Memory accesses are coalesced when consecutive threads access consecutive memory addresses, allowing the hardware to combine them into fewer transactions.

- *Constant memory* is a small, read-only cache for values that do not change during a kernel. It offers lower latency when all threads read the same address simultaneously.
- *Shared memory* is on-chip SRAM visible only to threads within the same block. Each SM provides a limited amount (tens of kilobytes) of shared memory; it has much lower latency than global memory and is ideal for inter-thread communication and tile-based algorithms.
- *Registers* are the fastest storage, private to each thread and used for local variables. Overuse of registers can limit occupancy, therefore we balance register usage against shared-memory needs.

Regarding atomicity, a GPU is capable of executing *atomic* operations on both global and shared memory. A GPU *atomic* function typically performs a *read-modify-write* memory operation on one 32-bit or 64-bit word. For example, AtomicXOR(*addr*, *val*) reads the old value at *addr*, XORes with *val*, then writes back the new value at *addr*.

2.7 Algorithmic Conventions

Throughout this article, we follow a consistent notation. Constants appear in all-caps, for example `MASKS`. Arrays and lists are written in lower-case, such as `pivots` or `prefixes`. Kernel and host-side helper routines use the notation Function, while local variables inside algorithm descriptions are typeset in an italic font via *variable*.

Functions highlighted in violet are kernels executed on device. Memory qualifiers indicate the scope and latency of storage: data in on-chip shared memory is marked with **shared**, and read-only constants in the GPU's constant cache with **constant**. When we refer to thread dimensions, we use bold italic symbols $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. In particular, the thread's block-local index in the $\mathbf{x}$ dimension is written as $t_{\mathbf{x}}$.

All parallel loops over a range of indices employ the grid-stride pattern. For example, we express grid-stride in $\mathbf{x}$-dimension as '**for all** $tid_{\mathbf{x}} \in [0, N-1]$ **do in parallel**,' meaning that each thread begins with $tid_{\mathbf{x}} = t_{\mathbf{x}}$, processes that element, then increments $tid_{\mathbf{x}}$ by the total number of threads and repeats until $tid_{\mathbf{x}} \geq N$. This ensures that any range larger than the number of available threads is still fully covered.

3 Key Challenges for Parallel Measurement and Approach

Our objective is to efficiently parallelize Stim's measurement algorithm to realize full simulation of stabilizer circuits on GPUs. To achieve this, we first analyze the access patterns in Stim's algorithm to identify parallelization opportunities. Before presenting our GPU-based measurement algorithm, we outline the key challenges and describe a data layout designed to maximize coalesced memory access and fully exploit modern GPU architectures.

3.1 Identifying Probabilistic Measurements

For each measurement operator in the input circuit, we must first identify a stabilizer generator that anti-commutes with the measurement (see line 2 in algorithm 1). Although

probabilistic measurements lead to different post measurement states, we can precompute, independently of the measurement outcomes, which observable qubits will be probabilistic.

Prior to executing measurements, we launch a two-dimensional GPU kernel in which one dimension iterates over measurement observables and the other searches for at least one anti-commuting stabilizer generator, whose index we refer to as the pivot p. This precomputation allows us to skip deterministic measurements, as in Stim, while still correctly handling measurements that become probabilistic after earlier state updates.

Once a probabilistic measurement is performed and the quantum state is modified, the pivot must be recomputed with respect to the updated tableau. We address this efficiently using stream compaction [17], as described in the following subsections.

3.2 Gaussian Elimination by Injecting CX Gates

One might ask why we parallelize the collapse step of Stim instead of the CHP update in section 2.5. In the CHP rule $\mathrm{S}_i \leftarrow \mathrm{S}_i\mathrm{S}_{\tilde{p}}$ and $\mathrm{D}_i \leftarrow \mathrm{D}_i\mathrm{D}_{\tilde{p}}$, the pivot generator $\mathrm{S}_{\tilde{p}}$ serves only as a fixed multiplier and is not overwritten until all anti-commuting rows have been updated. The updates for different indices i are thus independent, so CHP can in principle be parallelized directly. Doing so is wasteful, however, because each update multiplies full Pauli generators and recomputes their signs bit by bit. Stim avoids this cost by replacing the row multiplications with word-level conditional CX injections (algorithm 1), which it applies only for probabilistic measurements. This same optimization is what makes parallelization harder. Instead of a fixed multiplier, the injections now share a single control generator, the pivot, whose Z-destabilizer word z_c is updated in place as the targets are processed. We therefore parallelize the more efficient Stim-style algorithm and show how to recover the parallelism that the CHP formulation would otherwise provide.

The main obstacle to parallelizing the Gaussian elimination (GE) step in Stim (Function GaussianEliminationByCX in algorithm 1) is a loop-carried dependency across target generators. In the target loop at line 10, every CX injection shares the same control generator indexed by p, but each invocation of CXGate updates the control's Z-destabilizer word z_c in place (line 18). As a result, the sign update applied to a given target depends on the value of z_c produced by all preceding targets.

This dependency enforces the strict execution order $t = p+1, \ldots, n-1$ in Stim and makes naive parallelization incorrect. Even with atomic operations, concurrent threads would observe inconsistent values of z_c, breaking the required read-after-write dependence and producing incorrect sign histories. The challenge is therefore not merely avoiding write conflicts, but preserving the sequential evolution of z_c that subsequent sign updates depend upon.

To expose parallelism, we unfold the sequential updates in CXGate (line 18 in algorithm 1). Let s^k denote the sign state after processing the k-th target and z_c^k the corresponding control Z-destabilizer word. For $k = 1, 2, \ldots, n-p-1$, the updates satisfy

$$s^k \leftarrow s^{k-1} \oplus \left(x_c^{k-1} \wedge z_t^{k-1} \wedge \neg(z_c^{k-1} \oplus x_t^{k-1})\right), \tag{9}$$

$$z_c^k \leftarrow z_c^{k-1} \oplus z_t^{k-1}. \tag{10}$$

Quantities highlighted in blue denote loop-carried state that evolves across successive targets and enforces the sequential dependency in line 10. Unfolding the recurrence for z_c^k yields

$$z_c^k = z_c^0 \oplus \bigotimes_{l=0}^{k-1} z_t^l, \tag{11}$$

which is a classic prefix-XOR (scan) over the sequence $\{z_t^l\}$. Intuitively, each target contributes its z_t word to the evolving control state, and the value of z_c observed by target k is exactly the XOR of all prior contributions.

Once all z_c^k values are computed in parallel via a prefix scan, each target's sign contribution can be evaluated independently. The accumulated sign updates can then be combined using an XOR reduction, yielding the same final sign state as the sequential target loop. This reformulation eliminates the sequential bottleneck at line 10 and enables correct, fully parallel execution of CX injections on the GPU.

3.3 Handling Sparse X-Stabilizers

In Stim, a CX injection is required only for stabilizer generators that anti-commute with the measurement observable, as determined by the condition at line 11. In practice, the corresponding X-stabilizer rows are often sparse. Naively scanning the full range $[p+1, n-1]$ for each measurement therefore leads to unnecessary work, irregular memory accesses, and warp divergence on GPUs.

To address this, we first compact all target indices t such that $X_S[t, q] = 1$ (i.e., generators that anti-commute with the observable) into a contiguous array before performing GE. This preprocessing step yields several benefits:

- ✓ **Reduced warp divergence**: Only targets that actually require a CX injection are processed, avoiding conditional branches over non-pivots.
- ✓ **Coalesced memory access**: Consecutive threads operate on consecutive target indices, improving memory locality during tableau updates.
- ✓ **Lower overhead**: The GE kernels bypass scanning the full $[p+1, n-1]$ range, which is especially beneficial when stabilizers are sparse.

By combining prefix-XOR reformulation of CX injections (eq. (9)–eq. (11)) with compact indexing of sparse anti-commuting stabilizers, our algorithm preserves the semantics of Stim 's sequential GE while achieving high GPU utilization and substantially improved memory throughput.

4 Parallel Algorithm for Simulation

In this section, we present the first algorithm for performing stabilizer-based single-shot simulation in parallel on GPUs. Our parallel algorithm for Many-shot sampling is described separately in section 5. We begin with a high-level overview—from the input circuit all the way to the recorded measurement outcomes—then explain in detail each key component in the subsections that follow.

4.1 Overview of Our Simulator

Algorithm 2 gives a high-level view of our GPU-based stabilizer simulator. It accepts as input a quantum circuit $\mathcal{C}$, the number of qubits n, word size w (i.e., the machine word size), and an initial basis state $\texttt{initstate} \in \{0,1\}^n$. It produces as output the final state tableau $\mathcal{T}$ and, if any measurements occur, a bit-string $\texttt{measured} \in \{0,1\}^n$ recording the measured outcomes.

At line 1, We first decompose the input circuit into a sequence of windows $\mathcal{R}$. Each window contains either a single measurement or a group of parallel unitary gates acting

Algorithm 2: Top-level overview of parallel simulation on GPUs. Functions highlighted in violet denote GPU kernels.

```
   Input: 𝒞, n, w, initstate ∈ {0,1}^n
   Output: 𝒯, measured ∈ {0,1}^n
   /* Main procedure for simulation.                                                    */
1  ℛ ← ScheduleWindows(𝒞)                          // Schedule parallel gates on host into layers.
2  𝒯, k ← AllocateTableau(2n(2n+1), w)              // Allocate 2n(2n+1) bits in k = ⌈n/w⌉ words.
3  𝒯 ← InitializeTableau(initstate)                 // Initialize tableau on device.
4  forall 𝒲 ∈ ℛ do                                                          // For all layers.
5      𝒲_d ← CopyWindowToDevice(𝒲)                                  // Copy gate layer to device.
6      if HasMeasurement(𝒲) then                     // Does current window have measurement?
7          Measure(𝒯, 𝒲_d, measured, n, w, k)                  // Dispatch measurement procedure.
8      else
9          ApplyGates(𝒯, 𝒲_d, k)                                        // Apply gate rules on device.
   /* Main procedure for measurement.                                                   */
10 function Measure(𝒯, 𝒲_d, measured, n, w, k):
11     𝒯^T ← InlineTranspose(𝒯, n, w, k)                          // In-place row-major transpose.
12     𝒬_d ← MaxNondeterministic(𝒯^T, 𝒲_d, n, w, k)                // Find all probabilistic observables.
13     𝒬 ← CopyListToHost(𝒬_d)                                    // Copy 𝒬_d to host container 𝒬.
14     forall q ∈ 𝒬 ∧ q ≠ −1 do                          // For all (potential) probabilistic cases.
15         pivots ← FindAndCompactPivots(𝒯^T, q, n, w, k)          // Filter anti-commuting pivots.
16         p ← CopyFirstPivotToHost(pivots)                          // Bring first pivot to host.
17         if p ≠ −1 then                                          // Collapsing into Z is needed.
18             ParaGaussianEliminationByCX(𝒯^T, pivots, k)                  // Mutates 𝒯^T in place.
19             ParaSwapAntiCommuting(𝒯^T, p, q, n, k, w)
20             s_q ← CopySignBitToHost(q)                                // Copy q's sign bit to host.
21             if s_q ≠ Random(0,1) then
22                 ParaXGate(𝒯^T, p, n, k)                                  // Mutates the sign words.
23     measured ← CopyStabilizerSigns(S)                 // Unpack stabilizer signs in S[k → 2k] to measured.
24     𝒯 ← InlineTranspose(𝒯^T, n, w, k)          // In-place column-major transpose (restores column-major).
```

on disjoint qubits, allowing them to be applied simultaneously. The construction of these windows is described in section 4.2. At line 2, we allocate space for the $2n \times (2n+1)$ tableau, packed into w-bit words and with $k = \lceil n/w \rceil$ words per column. Section 4.3 describes this layout in detail. Then, at line 3, the tableau is initialized to the basis state `initstate` on the device.

At line 5, each scheduled window $\mathcal{W} \in \mathcal{R}$ is copied to device memory as $\mathcal{W}_d$ for use by subsequent GPU kernels. We then check whether the current window contains a measurement. If so, we invoke the host-driven Measure procedure (line 7); otherwise, we launch the ApplyGates kernel to apply all Clifford gates in parallel (lines 5–9). Section 4.4 describes the implementation of ApplyGates in detail.

When a window contains a measurement, the Measure procedure (lines 10–24) is executed. First, the tableau is transposed in place from column-major to row-major order using the InlineTranspose kernel (line 11), so that each generator's qubit bits are contiguous in memory. Next, we identify qubits that will collapse non-deterministically by launching the MaxNondeterministic kernel (line 12), which collects all observables whose generators anti-commute. The resulting device array $\mathcal{Q}_d$ is copied back to the host into $\mathcal{Q}$ (line 13), allowing the CPU to iterate over each candidate qubit sequentially.

For each nondeterministic observable qubit $q \in \mathcal{Q}$, the GPU kernel FindAndCompactPivots filters and compacts the indices of anti-commuting generators into a dense array `pivots` $\in [-1, n-1]^n$. The host then reads the first pivot $p =$ `pivots`$[0]$. If $p \neq -1$, the observable remains nondeterministic after prior tableau updates, and the GPU executes ParaGaussianEliminationByCX at line 18. This kernel implements the three-pass prefix-XOR elimination described in section 4.8, zeroing out all anti-commuting generators. Afterward, ParaSwapAntiCommuting is invoked at line 19 to swap in a commuting generator at index p.

Finally, the host reads back the sign bit corresponding to qubit q, compares it with a freshly generated random bit, and if they differ, triggers the ParaXGate kernel to inject an X gate and flip the measurement outcome.

Once all nondeterministic qubits have been collapsed, the GPU unpacks the final stabilizer sign bits into the output array `measured` $\in \{0,1\}^n$ and applies InlineTranspose again to restore the column-major layout of $\mathcal{T}$. The procedure returns the updated tableau and the measured bit string, after which the simulation loop at line 4 continues with the remaining windows.

4.2 Parallel Gate and Measurement Scheduling on CPU

We first partition the input circuit into *maximal windows* on the host, where each window is the largest possible set of mutually independent gates (cf. definitions 2–4). By doing so, we expose fine-grained parallelism before any GPU work begins. Further, we treat measurement operations as scheduling barriers: all unitary gates are advanced as early as possible into maximal windows, while measurements are delayed and executed only after the completion of their preceding window (see figure 2).

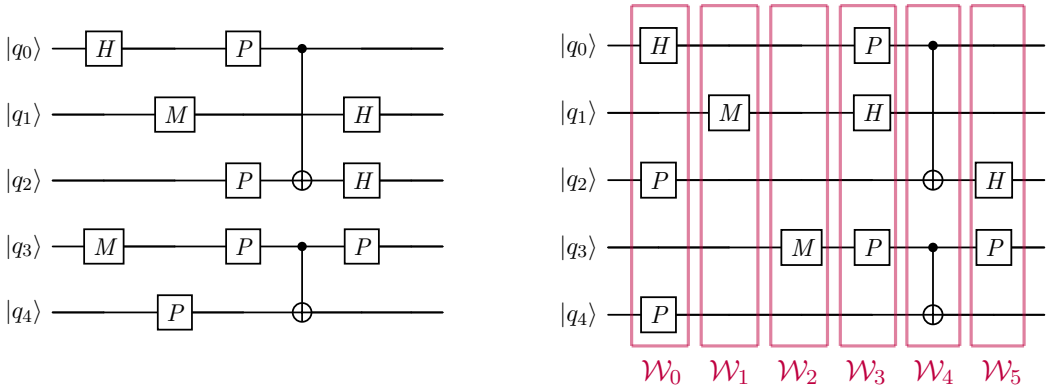


Figure 2: Input circuit on the left. Scheduled circuit on the right.

Definition 2 (Gate Dependency Relation)**.** *Given a circuit $\mathcal{C}$, let $g, g' \in \mathcal{C}$ be two gates. We say $g \preceq g'$ iff g' is causally dependent on g. This relation is a partial order over $\mathcal{C}$.*

Definition 3 (Parallel Gates)**.** *A subset $W \subseteq \mathcal{C}$ is a set of* parallel gates *(a* window*) if*

$$\forall\, g, g' \in W, \quad \neg(g \preceq g') \ \wedge \ \neg(g' \preceq g).$$

Definition 4 (Maximal Window)**.** *A window $W \subseteq \mathcal{C}$ is* maximal *if there exists no strictly larger window $W' \subseteq \mathcal{C}$ and W' contains up to one measurement operator.*

Henceforth, *window* will always mean a maximal window, denoted $\mathcal{W}$.

The above concepts underpin the ScheduleWindows routine in algorithm 2. Starting from the input-circuit queue (see left of figure 2), we iteratively:

1. Advance gates from the queue while preserving dependencies, until either n independent gates are collected or no further independent gates can be advanced. Measurement operations are treated as scheduling barriers and are therefore delayed and appended to the next available window.
2. Group the collected gates into the next maximal window $\mathcal{W}$.
3. Repeat until the queue is empty.

This produces the scheduled list $\mathcal{R}$ (right of figure 2), whose length scales with the original circuit depth. Each window can then be dispatched as a single batch to the GPU, maximizing parallel gate execution. Finally, measurements act as scheduling barriers: they cannot be included in a unitary window and are therefore deferred and appended to the next available window after all unitary gates in their causal past have been scheduled.

Remark on Sampling Mode. The above scheduling applies to single-shot simulation, where measurements act as barriers because the tableau must collapse after each non-deterministic outcome.

In contrast, for many-shot sampling (see section 5), we do not collapse the tableau into the Z-basis. Instead, we use Pauli frames to track measurement randomness. As a result, all measurement operators within a layer can be merged into a single measurement window and processed fully in parallel across shots. This removes the sequential barrier present in single-shot simulation and enables constant-time sampling per qubit.

4.3 Tableau Layout and Indexing

Upon designing the data structure for storing and updating the extended tableau on a GPU, several key considerations were made to ensure high efficiency of both the GPU memory and execution performance. GPUs operate on aligned memory, meaning data are stored and accessed in *words* (e.g., 8, 32, or 64 bits). This allows multiple bits to be processed simultaneously with a single instruction (e.g. XOR), which is crucial for fast tableau evolution.

Let w be the GPU word size in bits (here $w = 64$). We encode the extended tableau $\mathcal{T}$ as a 2D array of $\lceil k = n/w \rceil$ rows by $(2n + 1)$ columns of words:

$$\mathcal{T} \in \{0,1\}^{2n\times(2n+1)} \;\cong\; [0, 2^w - 1]^{2k\times(2n+1)} \quad \text{(packed } w\text{-bit words)}.$$

where each entry is a packed w-word. The first $2n$ columns encode the X and Z components of destabilizer and stabilizer generators, while the final column stores the sign vector S, which encodes the global phase of each generator in column-major layout. Each word encodes w bits of a single tableau column across w Pauli generators.

Transposing $\mathcal{T}$ for measurement kernels switches to a row-major layout of $2k$ rows by $2n + 1$ columns of words, so that each word packs w qubit bits across $2n$ generators. In contrast to the column-major layout, where words pack generator bits for a single qubit, the row-major layout packs qubit bits for a single generator. The size of the sign vector S remains unchanged and consists of $2k$ words; however, in the transposed layout, each word encodes the sign bits of w qubits. After performing a measurement on an observable q, the corresponding sign bit encodes the measurement outcome for that qubit.

While algorithm 1 presents the tableau in terms of explicit stabilizer (X_S, Z_S) and destabilizer (X_D, Z_D) blocks for clarity, the GPU implementation operates on a single packed 2D array of words. In this representation, both stabilizer and destabilizer rows are stored contiguously and accessed via linear word indices. Offsets are used to distinguish between the destabilizer and stabilizer halves of the tableau. This linear indexing enables coalesced memory access and simplifies parallel kernel design while remaining equivalent to the block-based formulation.

Memory Alignment. We pack bits into 64-bit words—the maximum GPU's native width—to maximize bitwise throughput. In our benchmarks, 64-bit words outperformed smaller sizes by updating more generators per instruction and reducing total memory traffic.

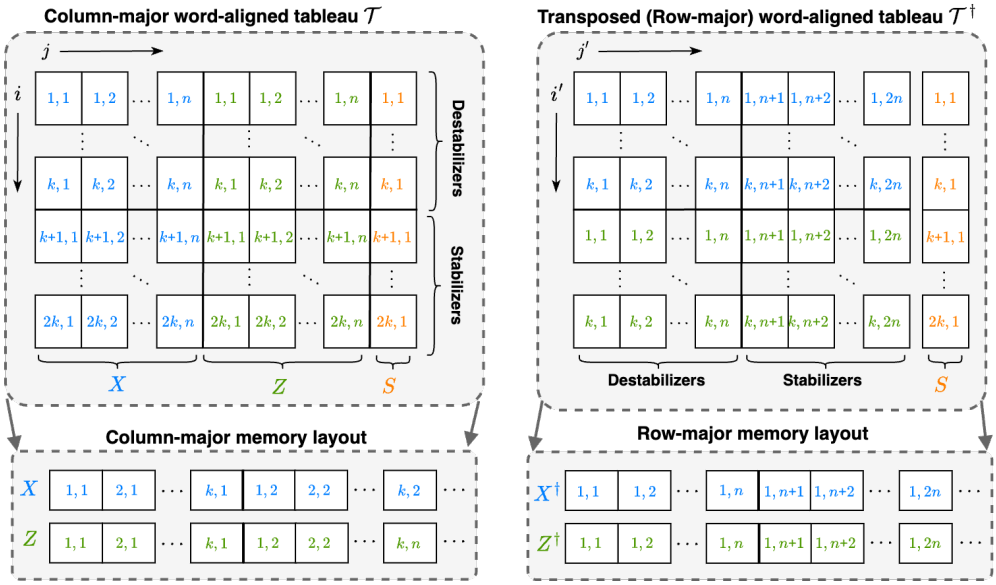


Figure 3: Logical and physical formats of a word-aligned tableau.

Memory Layout. Given the characteristics of the tableau evolution when applying Clifford gates, it is logical to store the X and Z in a column-major format (figure 3, left). This ensures each column (flattening destabilizers and stabilizers) can be accessed contiguously in memory, improving locality during tableau evolution. Measurement routines perform pivot searches and Gaussian elimination on different generators, per measured qubit, therefore we use a row-major layout (figure 3, right) to coalesce reads across generators. The sign vector S is stored separately and remains word-aligned; it is updated consistently with both layouts and is not affected by the transpose.

Tableau Density. As circuit depth increases, the tableau densifies and probabilistic measurements become more frequent. Our dual-layout strategy ensures both gate application and measurement kernels maintain coalesced access even under high tableau density.

4.4 Tableau Evolution under Clifford Gates

The kernel in algorithm 3 implements tableau evolution by applying parallel gates to the relevant tableau columns. Within this process, threads in the **y**-dimension (line 2) are responsible for fetching new words per column, whereas threads in the **x**-dimension (line 3) handle the retrieval of parallel gates. The variables $tid_{\mathbf{y}}$ and $tid_{\mathbf{x}}$ hold the global thread ID in **y**- and **x**-dimension, respectively. To branch over all supported gates by our algorithm, a switch statement is used at line 4. Here, we only show the H gate, but many more Clifford gates are directly supported without decomposition. At lines 5–9, we implement the update rules executed for each gate modifying both the words encoding the Pauli strings and their signs. Recall that $\mathcal{T}$ is stored in a column-major array when applying Clifford gates; thus, given a qubit q, and the column length $2k$, a column in the X or Z matrix can be accessed in position $(q \cdot 2k)$. To access words per column in parallel, the thread ID ($tid_{\mathbf{y}}$) is used to offset the column index. Note that in this work, the tableau has been extended to include destabilizers; the algorithms below treat tableau evolution for both stabilizers and destabilizers, unlike [12] where only stabilizers were evolved.

The subscript q in the gate's type refers to the connected wire or qubit. Qubit values are assumed to start from 0 to $n-1$ to align with the array index. At line 7, the signs

Algorithm 3: Tableau Evolution on GPU.

Input: $\mathcal{T} := (X, Z, S)$, $\mathcal{W}_d$, n, w, k
Output: $\mathcal{T}$

```
1  device kernel ApplyGates(T := (X, Z, S), W_d, k):
2      for all tid_y ∈ [ 0, 2k − 1 ] do in parallel                    // tid_y fetches a new word per column.
3          for all tid_x ∈ [ 0, |W_d| − 1 ] do in parallel              // tid_x fetches a new gate.
4              switch Type(W_d[tid_x]) do                               // Switch over gate type.
5                  case H_q do                                          // q ∈ {0, ··· , n − 1}.
6                      r ← q · 2k + tid_y                               // Linear word index in column-major X, Z.
7                      shared sm[t_x] ← X[r] ∧ Z[r]                     // Load local signs into shared memory.
8                      Swap(X[r], Z[r])
9                  case ... do ...                                      // Other Clifford gates like CX, CY, etc.
10         SynchronizeBlock( )                                          // Synchronize shared memory.
11         CollapseSigns(S[tid_y], sm)                                  // Collapse signs in shared memory.
   /* A function to collapse all thread-local signs using tree structure.                          */
12 device function CollapseSigns(s, sm):
       /* Notation. B_x = threads per block along x; t_x ∈ [0, B_x − 1].                           */
13     for b := B_x/2, B_x/4, ..., 1 do                                 // Loop over log(B_x) steps.
14         if t_x < b then sm[t_x] ← sm[t_x] ⊕ sm[t_x + b]
15         SynchronizeBlock( )
16     if t_x = 0 ∧ sm[0] = 1 then                                      // Collapse root sm[0] with old signs-word s.
17         AtomicXOR(s, sm[0])
```

are calculated and temporarily stored in shared memory (denoted as `sm`). The variable $\texttt{sm}[t_{\mathbf{x}}]$ holds the thread-local sign per thread block in the **x**-dimension. To ensure each thread block has written its signs to `sm` before proceeding, we need to synchronize at line 10. Once synchronized, these signs need to be collapsed into a final result, which will be used to update the global sign in the tableau. This collapsing process is handled by a specialized procedure, CollapseSigns, which we will describe in more detail next. The main tableau evolution to the tableau occurs at line 8, where the words in X and Z (representing the Pauli strings) are swapped in global memory.

Signs Collapsing. In our initial attempt to parallelize the signs collapsing process, we employed two kernels. The first kernel handled the tableau evolution and the local sign calculations, storing the results in global memory. The second kernel read these local signs from global memory and performed the collapsing using the atomic operation AtomicXOR. Unfortunately, this trial proved to be considerably slow due to the excessive global memory accesses (approximately $2k \cdot n$) and the high usage of atomic instructions ($2k \cdot |\mathcal{W}_d|$).

To remedy the global memory access issue, we made a second attempt. In this trial, we eliminated the temporary storage of local signs in global memory. Instead, we collapsed all local signs atomically on-the-fly during the tableau evolution within the whole window. While this approach saved roughly $(2k \cdot n)$ memory accesses compared to the initial attempt, the number of atomic operations remained a bottleneck. This is where our tree-like approach comes into play. The CollapseSigns routine in algorithm 3 effectively reduces the number of atomics by collapsing signs per thread block using a bottom-up binary tree. The downside is the excessive usage of global memory during the tree build-up, which we mitigate by utilizing the fast on-chip shared memory.

The loop at line 13 performs the collapsing using the XOR operator in shared memory by sweeping the binary tree in a bottom-up manner. Initially, b is set to $B_x/2$, where $B_x = 2^l$ and $l \in \{1, 2, \ldots, 10\}$. At line 14, we XOR in parallel the first half in $\texttt{sm}[t_{\mathbf{x}}]$ with the second half $\texttt{sm}[t_{\mathbf{x}}+b]$. Afterward, threads are synchronized and b is halved until only one element remains at `sm`[0] (resembles the root of the tree), which holds the collapsed block-local sign. Finally, at line 17, we collapse the block-local signs stored at `sm`[0] atomically

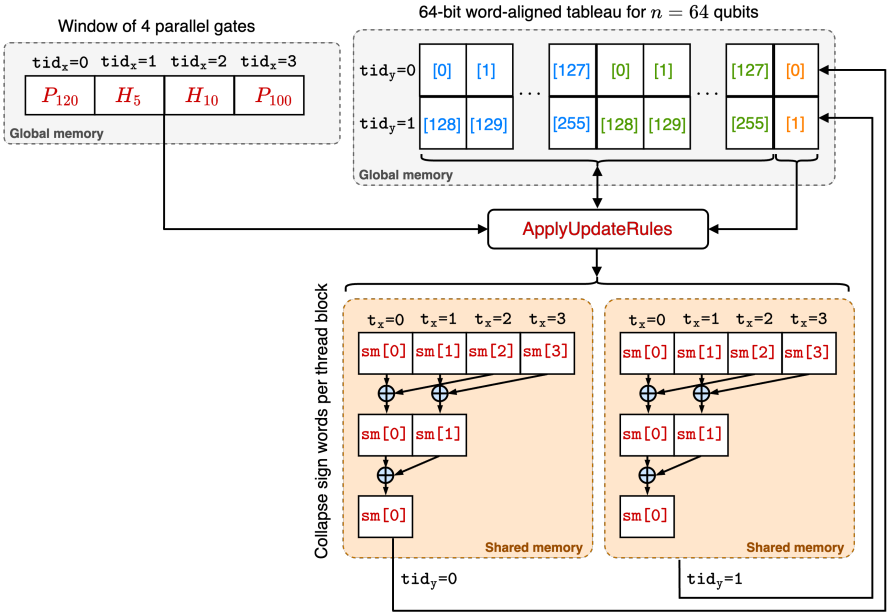


Figure 4: Running example of algorithm 3 on 128-qubit system.

with the old global signs at $S[tid_{\mathbf{y}}]$. This approach introduces an additional $log(B_x)$ steps to jump between tree levels (line 13). Albeit, the number of atomic operations is reduced from $2k \cdot |\mathcal{W}_d|$ to $2k \cdot (|\mathcal{W}_d|/B_x)$.

Figure 4 depicts a running example of algorithm 3 executed on the window $\mathcal{W} = \{P_{120}, H_5, H_{10}, P_{100}\}$ with $(n = 64)$. For that, given a word size of 64 bits, we require a tableau of dimensions $(k = 64/64 = 1,\ 2k \cdot (2n + 1) = 2 \cdot (128 + 1) = 258$ words). As a result, the number of threads in the **y**-dimension is 2, and 4 in the **x**-dimension. In that case, one thread block for both dimensions is sufficient (e.g. $B_x = 4$, $B_y = 2$). Once the collapsing process is complete and threads are synchronized per block, the root of the tree, denoted as `sm`[0], holds the final collapsed word. If multiple thread blocks are involved in the **x**-dimension, their results are combined using an AtomicXOR operation (as shown in line 17). In this specific example, as we only have one thread block in the **x**-dimension, `sm`[0] can be directly written to $S[tid_{\mathbf{y}}]$ non-atomically.

Complexity. The total work performed by algorithm 3 is dominated by the nested loops over k words and $|\mathcal{W}_d|$ gates, giving $\mathcal{O}(k \times |\mathcal{W}_d|)$ operations. In the worst case, both k and $|\mathcal{W}_d|$ grow as $\mathcal{O}(n)$, so the total work is $\mathcal{O}(n^2)$. These operations are spread across up to N hardware threads (the product of grid dimensions in **x** and **y**), yielding an ideal parallel time of $\mathcal{O}(\frac{n^2}{N})$. In addition, each block executes the CollapseSigns routine (lines 13–15), which is a binary-tree XOR over B_x threads. That loop takes $O(\log B_x)$ sequential steps within each block. Because B_x is capped by the hardware's maximum threads per block rather than by n [33], the overall worst-case parallel span becomes $\mathcal{O}(\frac{n^2}{N} + \log B_x)$.

4.5 In-Place Transpose

Before performing any measurement-related operations, we transpose the tableau layout from column-major to row-major order. Measurement procedures, such as collapsing a qubit into the Z-basis, require applying CX gates to anti-commuting generators *across all qubits*. To ensure that each generator's words remain contiguous and coalesced in memory during these operations, we store the tableau in row-major order and therefore

Algorithm 4: Tile-wise In-Place Transpose on GPU

Input: $\mathcal{T} := (X, Z, S)$, n, w, k
Output: $\mathcal{T}^T$

```
1  function InlineTranspose(T, n, w, k) → T^T:
2      ShuffleTiles(X, Z, n, w, k) SwapTiles(X, Z, k)
3      return T^T                          // now row-major in memory (denoted T^T); S unchanged.
4  device kernel ShuffleTiles(X, Z, n, w, k):
5      M ← (tid_z = 0) ? X : Z             // tid_z ∈ {0,1} selects X vs Z.
       /* Lookup table of masks in constant memory (listed at end of algorithm).   */
6      constant MASKS ← { bit-permutation masks }
7      for all tid_y ∈ [ 0, k − 1 ] do in parallel                    // Tile-row index.
8          for all tid_x ∈ [ 0, 2k − 1 ] do in parallel               // Tile-column index.
9              tileIdx ← (tid_y × w) × 2k + tid_x                     // Calculate linear tile index.
10             shared sm[t_x] ← M[tileIdx]                           // Load w × w-bit words (tile) into sm.
11             SynchronizeBlock( )                                   // Synchronize threads loading a tile.
12             for l ← 0 to log(w) − 1 do                            // For every pair of 2^l-bit groups.
13                 mask ← MASKS[l], offset ← 2^l
                   /* Shuffle (new) pairs of 2^l-bit groups.   */
14                 if (t_x mod (2 · offset)) = 0 then
15                     x ← sm[t_x], y ← sm[t_x + offset]
16                     a ← x ∧ mask, b ← x ∧ ¬mask
17                     c ← y ∧ mask, d ← y ∧ ¬mask
18                     sm[t_x] ← a ∨ ShiftLeft(c, offset)
19                     sm[t_x + offset] ← ShiftRight(b, offset) ∨ d
20                 SynchronizeBlock( )                               // Synchronize block threads shuffling a pair.
21             M[tileIdx] ← sm[t_x]                                  // Write new shuffled word.
22 device kernel SwapTiles(X, Z, k):
23     M ← (tid_z = 0) ? X : Z
24     for all tid_y ∈ [ 0, k − 1 ] do in parallel                    // Word-row index.
25         for all tid_x ∈ [ 0, k − 1 ] do in parallel                // Word-column index.
26             if tid_x > tid_y then                                  // If tid_x is above the diagonal.
27                 i_1 ← tid_y × 2k + tid_x                           // Word index above the diagonal.
28                 i_2 ← tid_x × 2k + tid_y                           // Word index below the diagonal.
29                 Swap(M[i_1], M[i_2])                               // Swap destabilizers off the diagonal.
30                 Swap(M[i_1 + k], M[i_2 + k])                       // Swap stabilizers off the diagonal.
31 /* Bit-permutation masks: {0x5555555555555555, 0x3333333333333333, 0x0F0F0F0F0F0F0F0F,
     0x00FF00FF00FF00FF, 0x0000FFFF0000FFFF, 0x00000000FFFFFFFF}                          */
```

must transpose it from its natural column-major layout.

We implement a two-stage, in-place GPU algorithm to transpose the tableau while concurrently converting from word-packed generators to bit-packed qubits. Because multiple threads may swap the same memory locations, we coordinate shared-memory shuffling and per-word swaps carefully to avoid data races.

In-Register Bit Shuffle. As shown in the ShuffleTiles kernel (lines 4–21), we launch a 3D grid with two threads in **z** so that $tid_{\mathbf{z}}$ selects between the X and Z matrices. Recall matrices are stored in column-major format as 1D arrays; therefore, an index of the form $i \times 2k + j$ expresses the location of the i-th qubit's j-th generator word. We then partition each matrix into $w \times w$-word tiles, flattening the triple (tile-row, tile-col, word-in-tile) into a single offset in the word array (see line 9).

Within each tile, we first load all w words into shared memory (line 10) and synchronize (line 11). Then we run the bit-shuffling [36, 37] loop at line 12 which, for each $l = 0, \ldots, 5$, swaps bit-groups of width 2^l using the precomputed w-bit masks in `MASKS`, inserting a barrier after each iteration to avoid races. Finally, we write back the now-transposed words to global memory (line 21).

Off-Diagonal Word Swaps. After bit-transposing each tile of $w \times w$-bit words in place, the entire tableau still needs its off-diagonal tiles exchanged. We sweep the $k \times k$ word grid

and, for each $tid_{\mathbf{x}} > tid_{\mathbf{y}}$, we swap locations i_1 and i_2 once in the destabilizer half (line 29) and once (offset by k) in the stabilizer half (line 30). Because each pair (i_1, i_2) is visited exactly once, no further synchronization is needed.

Complexity. This two-stage approach achieves a fully in-place transpose in $\mathcal{O}(\frac{n^2}{N})$ work, with perfect coalescing and no extra output buffers.

4.6 Finding Maximal Nondeterministic Measurements

Once the tableau has been transposed and prepared for measurement, we must determine which qubits require a probabilistic measurement, i.e., collapsing into the Z-basis. Scanning each qubit sequentially on the CPU would be prohibitively slow; instead, we launch a single 2D GPU kernel over the transposed tableau that tests all qubits in parallel and records only those whose stabilizer generators anti-commute with the measured observable.

Algorithm 5: Finding Maximal Nondeterministic Measurements on GPU

Input: $\mathcal{T}^T$, $\mathcal{W}_d$, n, w, k
Output: $\mathcal{Q}_d \equiv (q_i)_{i=0}^{|\mathcal{W}_d|-1}$, $q_i \in \{0, \ldots, n-1\} \cup \{-1\}$

1 ***device kernel*** MaxNondeterministic($\mathcal{T}^T, \mathcal{W}_d, n, w, k$) $\to \mathcal{Q}_d$:
2 **for all** $tid_y \in [\,0,\ |\mathcal{W}_d| - 1\,]$ **do in parallel** // Gate index.
3 $q \leftarrow$ Qubit($\mathcal{W}_d[tid_y]$) // Get observable $q \in \{0, \cdots, n-1\}$.
4 $\mathcal{Q}_d[tid_y] \leftarrow -1$ // Initialize device list.
5 **for all** $tid_x \in [\,0,\ n-1\,]$ **do in parallel** // Generator index.
6 $r \leftarrow \lfloor q/w \rfloor \times 2n + tid_x + n$ // Stabilizer index for observable q.
7 **if** GetBit($X^T[r], q) = 1 \wedge \mathcal{Q}_d[tid_y] = -1$ **then**
8 $\mathcal{Q}_d[tid_y] \leftarrow q$

Algorithm 5 launches a 2D grid in which the $\mathbf{y}$–dimension checks every measurement in the current window, and the $\mathbf{x}$–dimension scans through all n stabilizer generators. For each $(tid_{\mathbf{x}}, tid_{\mathbf{y}})$ pair, we compute the word index r that holds the corresponding stabilizer bit in X^T, where $\lfloor q/w \rfloor$ is the observable's word index and n is an offset used to access the stabilizer half of the row-major layout. At line 7, we test whether the stabilizer anti-commutes with the observable via the condition GetBit$(X^T[r], q) = 1$. As soon as one anti-commuting generator is found, we write q into $\mathcal{Q}_d[tid_{\mathbf{y}}]$ and prevent further overwrites by checking that $\mathcal{Q}_d[tid_{\mathbf{y}}] = -1$ beforehand.

In practice, multiple threads may attempt to write the observable index q into $\mathcal{Q}_d[tid_{\mathbf{y}}]$ if more than one anti-commuting generator exists. This benign race condition is harmless, since the algorithm only requires the existence of at least one anti-commuting generator, not its specific index. If none is found, the observable q is deterministic and $\mathcal{Q}_d[tid_{\mathbf{y}}]$ remains -1.

Complexity. Each of the $|\mathcal{W}_d| \times n$ bit tests is a constant-time operation; therefore, the total work is $\mathcal{O}(n^2)$. Since these tests are distributed across up to N hardware threads (the product of the grid's $\mathbf{x}$– and $\mathbf{y}$–dimensions), the parallel time is $\mathcal{O}\left(\frac{n^2}{N}\right)$.

4.7 Anti-Commuting Pivots Compaction

In order to eliminate warp divergence and maximize coalesced memory access, we compact all anti-commuting pivots into a one-dimensional array called `pivots`. Algorithm 6 performs this compaction in two steps. First, we launch a one-dimensional grid (line 2) that scans all generators and scatters the indices of anti-commuting ones into `pivots`, writing -1 for non pivots and the generator index (i.e., thread ID $tid_{\mathbf{x}}$) for pivots.

Algorithm 6: Pivot Stream Compaction on GPU

Input: $\mathcal{T}^T$, q, n, w, k
Output: `pivots` $\in [-1, \cdots, n-1]^n$ // Device array, compacted in place.

1 ***function*** FindAndCompactPivots($\mathcal{T}^T, q, n, w, k$) $\rightarrow$ `pivots`:
2 `pivots` $\leftarrow$ Scatter(X^T, q, n, w, k)
3 DeviceSelect::If(`pivots`, n)
4 ***device kernel*** Scatter(X^T, q, n, w, k):
5 **for all** $tid_x \in [\ 0,\ n-1\ ]$ **do in parallel** // Generator index.
6 $r \leftarrow \lfloor q/w \rfloor \times 2n + tid_x + n$ // Stabilizer index for observable q.
7 **if** GetBit($X^T[r], q$) $= 1$ **then** `pivots`$[tid_x] \leftarrow tid_x$
8 **else** `pivots`$[tid_x] \leftarrow -1$

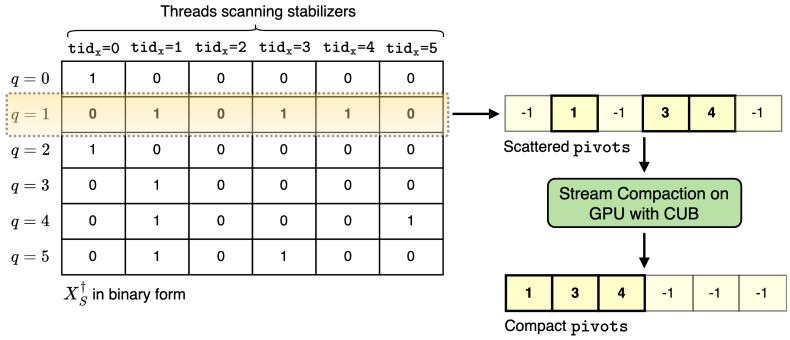


Figure 5: Compacting pivots on a 6-qubit tableau.

As in algorithm 5, a generator $tid_{\mathbf{x}}$ is considered a pivot for the observable q if and only if GetBit($X^T[r], q$) $= 1$. Each thread therefore checks the X-stabilizer entry corresponding to the observable q: if it anti-commutes, it writes `pivots`$[tid_{\mathbf{x}}] = tid_{\mathbf{x}}$; otherwise, it writes a negative value `pivots`$[tid_{\mathbf{x}}] = -1$, which lies outside the valid generator index range $[0, n)$ (lines 7-8).

Next, at line 3, we invoke CUB's DeviceSelect::If to perform *stream compaction* on `pivots` in place, moving all entries ≥ 0 to the front of the array while preserving their relative order. We refer to the CUB documentation[3] for details on DeviceSelect::If and stream compaction [17]. This produces a dense list of pivot indices $\{p_0, p_1, \dots\}$ followed by -1 fillers, allowing subsequent kernels to iterate only over valid pivots without gaps.

While stream compaction itself is a standard GPU primitive, its application to anti-commuting pivot selection in stabilizer-tableau simulation is, to our knowledge, new. In contrast to the Stim approach, our compaction removes warp divergence and ensures fully coalesced memory access during Gaussian elimination.

Figure 5 illustrates the process for a 6-qubit system: first scatter into {-1, 1, -1, 3, 4, -1}, then compact to {1, 3, 4, -1, -1, -1}.

Complexity. The Scatter kernel in algorithm 6 launches a grid that scans n generators using up to N threads, resulting in a parallel span of $\mathcal{O}(\frac{n}{N})$. The stream compaction step implemented by CUB is based on multi-level prefix sums [17] and has a worst-case parallel time of $\mathcal{O}(\frac{n}{N} + \log n)$. Therefore, the overall worst-case parallel time of algorithm 6 is $\mathcal{O}(\frac{n}{N} + \log n)$.

[3] https://nvidia.github.io/cccl/unstable/cub/device_wide.html#device-module

4.8 Three-Pass Prefix-XOR Elimination

Once the pivot indices have been compacted, we can perform Gaussian elimination without any further search for anti-commuting generators. Algorithm 7 summarizes our three-pass, prefix-XOR approach to injecting CX gates in parallel for each pivot in the compacted array `pivots`. The inputs are the transposed tableau $\mathcal{T}^T$ (obtained by algorithm 4), the compacted pivot list `pivots`, and the number of qubit words per row k.

Because the number of pivots can significantly exceed a single thread-block's capacity, we split the computation into multiple CUDA kernel launches. To carry intermediate results between passes, we allocate two arrays on the device, called `prefixes` and `blocksums`. Each measured qubit can have up to n anti-commuting pivots, therefore we provision these arrays with dimensions $n \times k$. Although only the first $|\texttt{pivots}|$ rows are populated in any given measurement, allocating $n \times k$ guarantees sufficient storage. Each entry is a $2w$-bit cell that interleaves the Z-prefix and X-prefix words; by packing both w-bit values into one cell, every thread's global-memory access fetches exactly the paired prefixes it needs, yielding fully coalesced loads and stores when scanning Z^T and X^T. Figure 6 illustrates this layout.

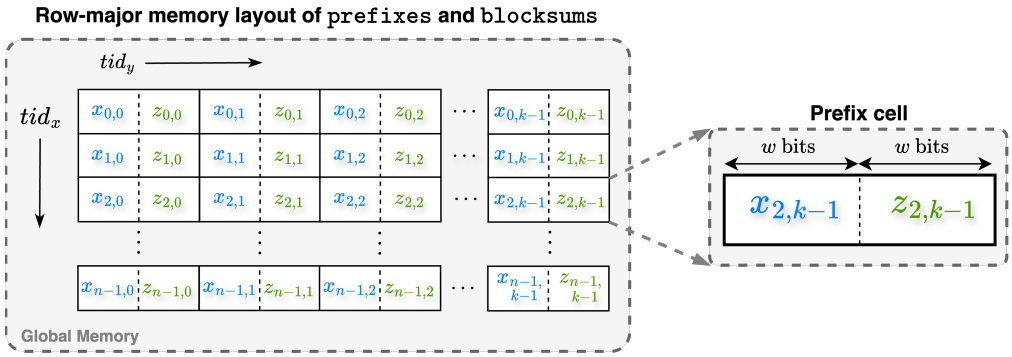


Figure 6: Memory layout of `prefixes` and `blocksums`.

The outer driver ParaGaussianEliminationByCX (lines 1–4) invokes three kernels in succession:

1. At line line 2, the kernel PrefixXORPass1 is launched. Inside PrefixXORPass1, each thread block handles one contiguous block of pivots for a fixed qubit-word. For each pivot in the block, the kernel loads the corresponding target-destabilizer bit z_t into shared memory. It then invokes CUB's ExclusivePrefix routine, which implements Blelloch's up/down sweep, to compute two values per block: the exclusive prefix (stored as $prefix_z$) and the total block-sum (stored as $bsum_z$) of those z_t words. The same process runs in parallel on the x_t words to produce $prefix_x$ and $bsum_x$,.

2. At line 3, PrefixXORPass2 performs a device-wide exclusive prefix-XOR using the CUB's routine DeviceScan::ExclusiveSum over the accumulated `blocksums` using PrefixXORPass2, producing global block-prefix corrections that will be applied to every pivot's per-block prefixes.

3. At line 4, PrefixXORPass3 merges the two levels of prefix data–combining per-block prefixes with their block-prefix corrections–then, for each pivot and qubit-word, computes the final CX-injection update on the tableau via the device function InjectCXByPrefix (implementing equations eq. (9) and eq. (11)).

Algorithm 7: Three-Pass Prefix-XOR Gaussian Elimination

Input: $\mathcal{T}^T$, pivots, n, w, k
Output: $\mathcal{T}^T$

1 ***function*** ParaGaussianEliminationByCX($\mathcal{T}^T$, pivots, k):
2 prefixes, blocksums ← PrefixXORPass1($\mathcal{T}^T$, pivots, k)
3 blocksums ← PrefixXORPass2(blocksums) // Calls a CUB's device-wide routine.
4 $\mathcal{T}^T$ ← PrefixXORPass3(pivots, prefixes, blocksums, $\mathcal{T}^T$, k)
5 ***device kernel*** PrefixXORPass1($\mathcal{T}^T$, pivots, k) → prefixes, blocksums:
6 **for all** $tid_y \in [\,0,\ k-1\,]$ **do in parallel** // Qubits-word index.
7 **for all** $tid_x \in [\,0,\ |\text{pivots}|-2\,]$ **do in parallel** // pivots index.
8 $c \leftarrow \text{pivots}[0],\ t \leftarrow \text{pivots}[tid_x+1]$ // Fetch control and target pivots.
9 $r_t \leftarrow tid_y \times 2n + t$ // Target-word index.
10 **shared** $\text{sm}_z[t_x] \leftarrow Z^T[r_t],\ \text{sm}_x[t_x] \leftarrow X^T[r_t]$ // Load destabilizer words.
11 SynchronizeBlock( ) // Synchronize threads loading into sm.
 /* Compute prefixes $\bigotimes_{t_x=0}^{B_x-1} Z^T[r_t]^{t_x}$ and $\bigotimes_{t_x=0}^{B_x-1} X^T[r_t]^{t_x}$ per thread block. */
12 $(prefix_z,\ bsum_z) \leftarrow$ ExclusivePrefix($\text{sm}_z[t_x]$)
13 $(prefix_x,\ bsum_x) \leftarrow$ ExclusivePrefix($\text{sm}_x[t_x]$)
14 $r_c \leftarrow tid_y \times 2n + c$ // Control-word index.
15 $\text{prefixes}[tid_x \times k + tid_y] \leftarrow (prefix_x \oplus X^T[r_c],\ prefix_z \oplus Z^T[r_c])$
 /* Last thread per block stores $bsum$ and updates z_c w.r.t. eq. (11). */
16 **if** $t_x = B_x - 1$ **then**
17 $\text{blocksums}[\lfloor tid_x/B_x \rfloor \times k + tid_y] \leftarrow (bsum_x,\ bsum_z)$
18 AtomicXOR($X^T[r_c],\ bsum_x$), AtomicXOR($Z^T[r_c],\ bsum_z$)
 /* Compute the final prefixes and finalize CX injections. */
19 ***device kernel*** PrefixXORPass3(pivots, prefixes, blocksums, $\mathcal{T}^T$, k):
20 **for all** $tid_y \in [\,0,\ k-1\,]$ **do in parallel** // Qubits-word index.
21 **for all** $tid_x \in [\,0,\ |\text{pivots}|-2\,]$ **do in parallel** // pivots index.
 /* Finalize prefixes by applying the exclusive scan results of block-sums. */
22 $(prefix_x,\ prefix_z) \leftarrow \text{prefixes}[tid_x \times k + tid_y] \oplus \text{blocksums}[\lfloor tid_x/B_x \rfloor \times k + tid_y]$
 /* Update (de)stabilizers signs using computed prefixes $(prefix_x,\ prefix_z)$. */
23 $c \leftarrow \text{pivots}[0],\ t \leftarrow \text{pivots}[tid_x+1]$
24 $r_c \leftarrow tid_y \times 2n + c,\ r_t \leftarrow tid_y \times 2n + t$
25 InjectCXByPrefix($S[tid_y],\ Z^T[r_c+n],\ prefix_z,\ Z^T[r_t+n],\ Z^T[r_t],\ sm$)
26 InjectCXByPrefix($S[tid_y+k],\ X^T[r_c+n],\ prefix_x,\ X^T[r_t+n],\ X^T[r_t],\ sm$)
 /* Update (de)stabilizers and signs. x_c: c-stabilizer, $prefix$: c-destabilizer prefix, */
 /* x_t: t-stabilizer, and z_t: t-destabilizer */
27 ***device function*** InjectCXByPrefix($s, x_c, prefix, x_t, z_t$, sm):
28 **shared** $\text{sm}[t_x] \leftarrow x_c \wedge z_t \wedge \neg(prefix \oplus x_t)$
29 CollapseSigns(s, sm) // Collapse signs in sm with a binary tree.
30 $x_t \leftarrow x_t \oplus x_c$

Pass One: Block-Local Prefix-XOR. In PrefixXORPass1, we launch a 2D grid of thread-blocks sized $B_x \times B_y$ in $(\mathbf{x}, \mathbf{y})$-dimensions, where $\mathbf{y}$ covers qubit-word indices $tid_\mathbf{y} \in [0, k-1]$, and $\mathbf{x}$ covers pivot indices $tid_\mathbf{x} \in [0, |\texttt{pivots}|-2]$. Each thread loads the target's destabilizer words $Z^T[tid_\mathbf{y} \times 2n + t]$ and $X^T[tid_\mathbf{y} \times 2n + t]$ into the shared memory array `sm` at line 10, then synchronizes.

Next, we invoke CUB's ExclusivePrefix routine (lines 12–13), which performs an up-sweep to build a full binary tree over the B_x threads and captures the block-sum at the root, then a down-sweep to compute every thread's exclusive prefix. Thread $t_\mathbf{x}$ (per block) thus obtains $prefix_z$ and $bsum_z$, and analogously $prefix_x$ and $bsum_x$. We record the combined words $(prefix_x \oplus X^T[r_c], prefix_z \oplus Z^T[r_c])$ in the global `prefixes` array at line 15, where $X^T[r_c], Z^T[r_c]$ are the control-pivot's destabilizer words stored in X^T and Z^T, respectively.

Finally, the last thread in each block (when $t_\mathbf{x} = B_x - 1$) writes the block-sums pair $(bsum_x, bsum_z)$ to the global array `blocksums` at position $\lfloor tid_\mathbf{x}/B_x \rfloor \times k + tid_\mathbf{y}$, where $\lfloor tid_\mathbf{x}/B_x \rfloor$ is the block index (i.e. which block of pivots this thread belongs to). Immediately afterward, at line 18, the same thread applies two atomic XORs to globally update the c-destabilizers $X^T[r_c]$ and $Z^T[r_c]$. Thereby, accumulating the block-sums into the global tableau entries according to eq. (11).

Pass Two: Block-Sum Prefix-XOR. In the second pass, we compute an exclusive prefix over the block-sum pairs stored in `blocksums` using a device-wide scan implemented in PrefixXORPass2. This operation combines the partial block-sums produced in PrefixXORPass1 into global block-prefix corrections, without imposing any restrictions over the block size.

The resulting scanned corrections are stored back in `blocksums` in place and are subsequently consumed by PrefixXORPass3 to finalize the prefix values for each pivot.

Pass Three: Final CX Injection. Finally, in the third kernel, PrefixXORPass3 (line 19), we launch a 2D grid with size similar to the one in PrefixXORPass1. Each thread first reads its per-block prefix from the global array `prefixes` at position $(tid_{\mathbf{x}}, tid_{\mathbf{y}})$ and then reads the corresponding block-prefix correction from `blocksums` at position $(\lfloor tid_{\mathbf{x}}/B_x \rfloor, tid_{\mathbf{y}})$. It computes the final exclusive prefix as

$$prefix \;=\; \texttt{prefixes}[tid_{\mathbf{x}}, tid_{\mathbf{y}}] \;\oplus\; \texttt{blocksums}[\lfloor tid_{\mathbf{x}}/B_x \rfloor, tid_{\mathbf{y}}].$$

With this prefix in hand, the thread calls the device function InjectCXByPrefix. Inside that function it loads four 64-bit words: the c-stabilizer x_c, the t-destabilizer z_t, the t-stabilizer x_t, and the computed prefix word. It then calculates the sign history $x_c \wedge z_t \wedge \neg(prefix \oplus x_t)$ and stores it into shared memory at index $t_{\mathbf{x}}$. Next, all threads in the block cooperate in a binary-tree reduction to collapse those history values into a single 64-bit result at the root `sm`[0]. That result is atomically XORed into the global S array at the entry corresponding to the current X-signs word index $tid_{\mathbf{y}}$ (or $tid_{\mathbf{y}} + k$ for the Z-signs), finalizing the sign update for this CX injection. Afterward, InjectCXByPrefix finishes the CX injection by merging the c-stabilizer word into the t-stabilizer entry. This completes the in-place injection of the CX gate for each pivot, with correct sign handling and coalesced memory access throughout.

Complexity. Overall, this three-pass scheme performs $|\texttt{pivots}| \times k$ injections of the CX gate in parallel. Noting that $|\texttt{pivots}|$ and k in the worst case can grow to $\mathcal{O}(n)$, therefore, the total work is $\mathcal{O}(n^2)$. These $\mathcal{O}(n^2)$ operations are distributed across up to N hardware threads (the product of the grid's $\mathbf{x}$– and $\mathbf{y}$–dimensions), giving an ideal parallel time of $\mathcal{O}(\frac{n^2}{N})$. Passes One and Three include an intra-block exclusive scan over B_x threads (e.g. lines 12 and 13), which costs $\mathcal{O}(\log B_x)$ steps. Since B_x is limited by the hardware's maximum threads per block rather than by n [33], the overall worst-case parallel time is $\mathcal{O}(\frac{n^2}{N} + \log B_x)$.

The algorithm thus achieves a fully in-place, lock-free, coalesced elimination that scales to large pivot sets on GPUs.

4.9 Final Measurement Assembly

Swapping the Anti-commuting Generator. After collapsing non-deterministic measurements into the Z-basis (via algorithm 7), we must replace the first anti-commuting generator at index p with one that commutes with the measured observable q. The kernel SwapAntiCommuting (algorithm 8, lines 1–12) launches a one-dimensional grid of k threads, each responsible for a single w-bit word of the pivot row.

Thread 0 first extracts the observable's Pauli type at the pivot position $\lfloor q/w \rfloor \times 2n + p$ into the flag $isYPauli$ (line 3). A grid-wide barrier SynchronizeGrid (line 5) ensures that all threads observe the same value.

Algorithm 8: Final Measurement Assembly on GPU

Input: $\mathcal{T}^T$, p, q, n, w, k
Output: $\mathcal{T}^T$

1 ***device kernel*** ParaSwapAntiCommuting($\mathcal{T}^T, p, q, n, k, w$):
2 **for all** $tid_x \in [\,0,\ k-1\,]$ **do in parallel** // For all qubit-word indices.
3 **if** $tid_x = 0$ **then**
4 $isYPauli \leftarrow$ GetBit$\left(X^T[\lfloor q/w \rfloor \times 2n + p], q\right)$ // Observable's bit.
5 SynchronizeGrid( ) // Ensure *isYPauli* is visible to all.
6 $r \leftarrow tid_x \times 2n + p$
7 **if** $isYPauli = 1$ **then** // If Y-Pauli, inject $P^\dagger$.
8 PInvGate($Z^T[r], Z^T[r+n], S[tid_x]$)
9 PInvGate($X^T[r], X^T[r+n], S[tid_x + n]$)
10 **else** // If X-Pauli, inject H.
11 HGate($Z^T[r], Z^T[r+n], S[tid_x]$)
12 HGate($X^T[r], X^T[r+n], S[tid_x + n]$)
13 ***device kernel*** ParaXGate($\mathcal{T}^T, p, k, n$):
14 **for all** $tid_x \in [\,0,\ k-1\,]$ **do in parallel** // For all qubit-word indices.
15 $r \leftarrow tid_x \times 2n + p$
16 $S[tid_x] \leftarrow S[tid_x] \oplus Z^T[r]$
17 $S[tid_x + k] \leftarrow S[tid_x + k] \oplus X^T[r]$

Randomizing the Measurement. Finally, if the coin flip disagrees with the observable's sign (line 21 in algorithm 2), we inject an X gate via the kernel XGate (lines 13–17). This one-dimensional grid XORs each word of the pivot's Z- and X stabilizer rows into the corresponding entries of the sign array, completing the measurement update.

Complexity. Both ParaSwapAntiCommuting and ParaXGate launch k parallel threads, each performing a constant number of word-level bitwise operations. Since $k = \mathcal{O}(n)$, the total work is $\mathcal{O}(n)$, yielding an ideal parallel time of $\mathcal{O}(n/N)$. The single grid synchronization in ParaSwapAntiCommuting incurs only constant overhead.

5 Sampling

One often needs to run many *shots* to evaluate the probability distribution of the state generated by the circuit, i.e., independent simulations of the same circuit, to estimate output probabilities [5]. Naively, this would require reapplying all gates, including probabilistic measurements, once per shot, which can be prohibitively expensive. Stim instead uses the *Pauli-frame* trick to amortize this cost. Each qubit maintains a bit-string that tracks phase flips, and measurements simply update that frame instead of re-collapsing the full tableau state.

Regarding scheduling in sampling mode, the window structure $\mathcal{R}$ constructed in section 4.2 is reused for sampling. However, measurement windows are interpreted differently. In single-shot simulation, measurements act as barriers because the tableau must collapse after each probabilistic outcome. In contrast, during sampling we do not collapse the tableau. Instead, Pauli frames absorb the randomness, allowing all measurement operators within a window to be processed fully in parallel across all shots.

Compared to state tableau in eq. (2), frames replace Pauli generators and their signs. Concretely, for f shots per frame (column), the stabilizer tableau $\tilde{\mathcal{T}} = (\tilde{X}, \tilde{Z})$ is represented as two bit-matrices of size f (rows) $\times\ n$ (columns), but packs each row of n into words of size w. Recall $k = \lceil n/w \rceil$ be the number of machine words needed per row. Then we store $\tilde{X}, \tilde{Z} \in \{0,1\}^{f \times n} \equiv \{0, \ldots, 2^w - 1\}^{f \times k}$ so that each word holds w independent shots for a single qubit.

During sampling, every Clifford gate on shot-bundle words simply updates these packed

rows with bitwise operations, and each measurement step toggles bits in the Pauli frame randomly rather than collapsing the full tableau.

Algorithm 9 exploits our frame simulation algorithm pulling out the randomness into those w-bit flips, allowing us to process w shots in parallel for all measured qubits simultaneously, achieving a dramatic speedup over conventional methods. We first allocate two packed matrices, $\tilde{X}$ and $\tilde{Z}$, of size $f \times k$ (algorithm 9), plus an output buffer `record` of the same shape (algorithm 9). On the device, at line 3, we reset $\tilde{X}$ and fill $\tilde{Z}$ with independent Philox-generated random words via cuRAND [38], ensuring each word encodes w fresh random bits. The Philox family is well suited for parallel platforms as it can generate independent streams of random numbers without the need of synchronization [39].

Algorithm 9: Sampling on GPU

```
   Input: R, n, w, f                                          // f (number of shots).
   Output: record                                             // record ∈ {0,...,2^w − 1}^{f×k}.
   /* Main procedure for sampling.                                                        */
1  T̃, k ← AllocateTableau(f × 2n, w)                          // Allocate f × 2n bits in k = ⌈n/w⌉ words.
2  record ← AllocateTableau(f × n, w)                         // Allocate f × n bits in k words.
3  X̃ ← {0}, Z̃ ← Randomize()                                   // Initialize tableau on device.
4  forall W ∈ R do                                            // For all windows.
5      W_d ← CopyWindowToDevice(W)                            // Copy gate layer to device.
6      if HasMeasurement(W) then                              // Is W a measuring window?
7          record ← MeasureSample(T̃, W_d, k, n)
8      else
9          ApplyGates(T̃, W_d, k)                              // Apply gate rules on device (algorithm 3).
10 device kernel MeasureSample(T̃, W, k, n) → record:
11     for all tid_y ∈ [ 0, |W_d| − 1 ] do in parallel        // Measuring-operator index.
12         q ← Qubit(W_d[tid_y])                              // Get measurable qubit.
13         for all tid_x ∈ [ 0, k − 1 ] do in parallel        // Word-shot index.
14             r ← q × k + tid_x
15             record[r] ← X̃[r]                               // Record q's frame from X̃.
16             Z̃[r] ← RandomWord(r)                           // Randomize q's frame in Z̃.
```

Next, we simulate each scheduled window in turn. Non-measurement windows invoke our standard ApplyGates kernel to update $\tilde{X}, \tilde{Z}$ under Clifford gates (line 9). Measurement windows run the MeasureSample (line 7) kernel. This kernel launches a 2D grid: for every measured qubit and each frame-word index, we copy the current $\tilde{X}$ word (outcome of w shots) into `record`, then re-randomize the corresponding $\tilde{Z}$ word for subsequent measurements.

Complexity. For each window in $\mathcal{R}$, sampling performs either (1) a call to ApplyGates (Algorithm 3) for Clifford gates, or (2) a call to MeasureSample (Algorithm 9) for measurement windows.

Recall from section 4.4 that the cost of ApplyGates is $\mathcal{O}(n^2)$ total work per window, yielding an ideal parallel time of $\mathcal{O}(\frac{n^2}{N} + \log B_x)$ due to the binary-tree sign aggregation.

For measurement windows, the MeasureSample kernel performs $\mathcal{O}(k \times |\mathcal{W}_d|)$ work. In the worst case, both scale as $\mathcal{O}(n)$, giving $\mathcal{O}(n^2)$ total work and an ideal parallel time of $\mathcal{O}(\frac{n^2}{N})$. Unlike gate application, no logarithmic synchronization is required.

Because windows are maximal, the number of scheduled windows is minimized. Independent Clifford gates are applied together via ApplyGates, and independent measurement operators are processed together via MeasureSample, each in a single kernel launch. Therefore, the overall sampling complexity scales with the circuit depth rather than with the total number of gates or measurements.

Comparison with Stim. In Stim, measurements are processed sequentially over qubits.

Each frame update touches $\mathcal{O}(n)$ bits, and over $\mathcal{O}(n)$ measured qubits this results in $\mathcal{O}(n^2)$ work per shot batch. Moreover, the qubit-wise iteration is not parallelized across shots, so increasing f increases total work proportionally.

In contrast, our MeasureSample kernel parallelizes both the qubit dimension and the shot-word dimension. Given sufficient hardware threads to cover $f \times k$ words, each measurement window has constant parallel span, independent of the number of shots.

6 Experiments and Discussion

Setup. We implemented algorithm 2 in a new CUDA C++ tool called QuaSARQ[4], compiled with CUDA 12.6 targeting compute capability 8.9. GPU experiments were run with a timeout of 4 hours on a machine running Ubuntu 22.04 and equipped with an RTX 4090 GPU (16,384 cores at 2.23 GHz and 24 GB of global memory).

We compare QuaSARQ against six leading simulators: Stim[5], Qiskit-Aer (both CPU and GPU modes)[6], Qibo[7], Cirq[8], and PennyLane[9], with their recommended Clifford-simulation settings. Note that PennyLane's Lightning-GPU plugin and Cirq's GPU alternative currently do not scale beyond tenths of qubits without exhausting device memory. While Stim is CPU-only, it exploits SIMD instruction sets (SSE/AVX) to process 128–512 bits in parallel, significantly accelerating tableau operations.

All CPU experiments were conducted separately on the DAS-6 cluster [40], whose nodes are equipped with AMD EPYC 7282 CPUs (2.8 GHz) and 256 GB of memory. Each circuit was executed in isolation on a dedicated node with a 4 hour timeout. Notably, the DAS-6 CPU clock is approximately 20% faster than the GPU clock used by QuaSARQ.

Benchmarks. Since the computational cost of stabilizer-circuit simulation is determined entirely by the circuit structure (qubit count, gate count, and measurement locations), performance can be characterized using randomly generated circuits without evaluating a broad range of quantum algorithms [9, 11, 10]. Accordingly, we constructed two benchmark suites, each comprising three sets of random OpenQASM circuits with depths $d \in \{100, 500, 1000\}$, generated using IBM's Qiskit.[10]

The *light* suite contains 100 circuits per depth, with qubit counts ranging from 100 to 10,000 in steps of 100. This suite targets simulators with modest resource requirements (e.g., PennyLane, Cirq, and Qiskit-Aer) while also evaluating QuaSARQ on smaller instances. The *heavy* suite spans 1,000 to 180,000 qubits in steps of 1,000, yielding 180 circuits per depth and stressing large-scale simulators such as Stim. Larger circuits would exceed the 24 GB GPU memory limit, and Stim also times out beyond this scale. Across both suites, circuit sizes range from 7K to 130M gates. By default, Qiskit generates uniformly distributed Clifford gates from the set $\{X, Y, Z, H, P, P^\dagger, CX, CY, CZ, SWAP, ISWAP\}$, all supported by the evaluated simulators and decomposable into the basic Clifford gates $\{X, Y, Z, H, P, CX\}$.

[4] https://github.com/System-Verification-Lab/QuaSARQ

[5] https://github.com/quantumlib/Stim

[6] https://github.com/Qiskit/qiskit-aer

[7] https://github.com/qiboteam/qibo

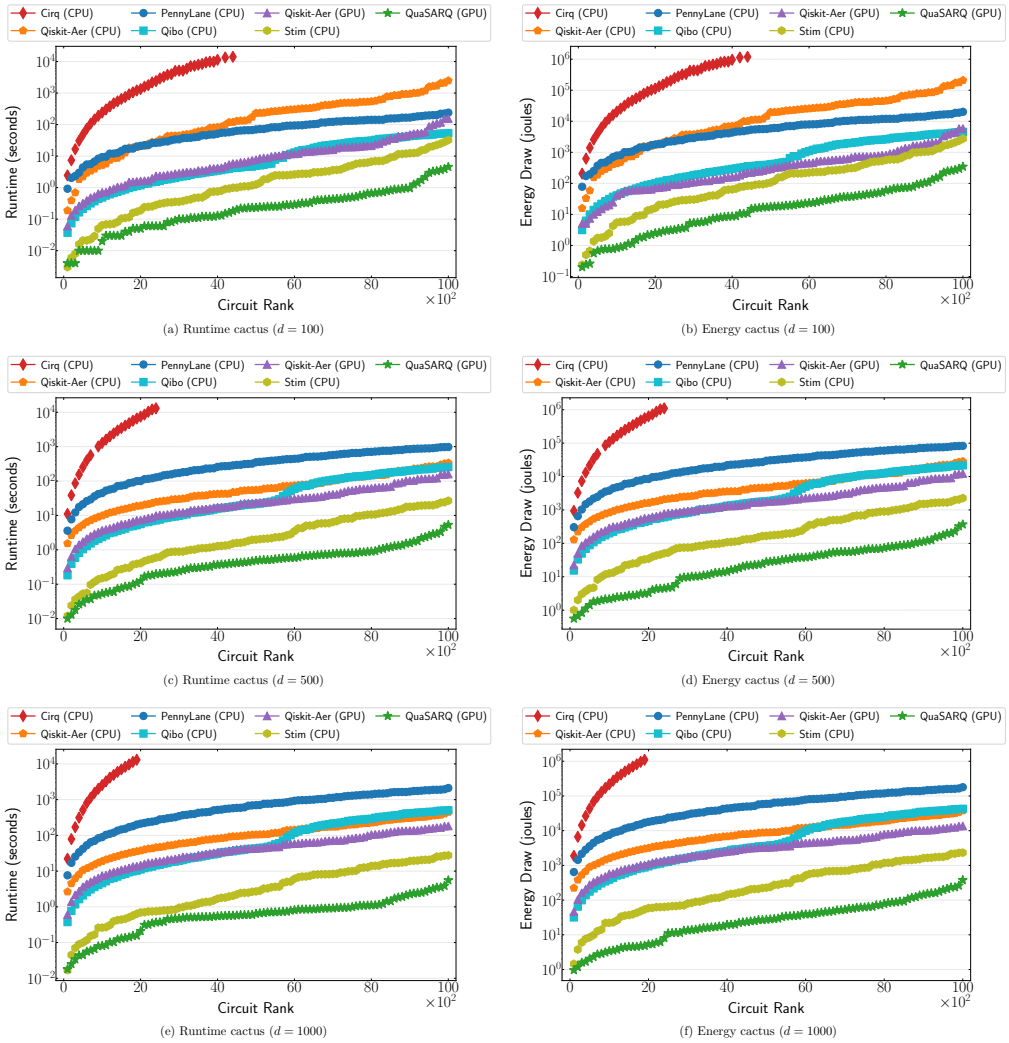
[8] https://github.com/quantumlib/Cirq

[9] https://github.com/PennyLaneAI/pennylane

[10] https://github.com/Qiskit/qiskit

To assess measurements performance (runtime and energy consumption) across all simulators, we augmented each circuit with a randomized set of measurement operators. We then executed a single simulation run for each circuit, which suffices to capture the impact of our GPU-based measurement kernels on overall performance without the need for multiple repetitions. For sampling performance, we compared Stim 's built-in high-performance sampler (configured for 1,024 shots) against our Pauli-frame sampler outlined in algorithm 9.

Experiments. Figure 7 presents cactus plots comparing runtime and energy consumption of our GPU-based QuaSARQ against six state-of-the-art simulators: PennyLane, Qibo, Stim, Qiskit-Aer (GPU), Qiskit-Aer (CPU), and Cirq on the light benchmark suite. In these plots, the x-axis shows the circuit rank, defined as the position of a circuit after sorting all benchmarks by increasing runtime or energy.



(a) Runtime cactus ($d = 100$)
(b) Energy cactus ($d = 100$)
(c) Runtime cactus ($d = 500$)
(d) Energy cactus ($d = 500$)
(e) Runtime cactus ($d = 1000$)
(f) Energy cactus ($d = 1000$)

Figure 7: Simulation performance of QuaSARQ vs. PennyLane, Qibo, Qiskit-Aer (both CPU and GPU versions), Cirq, and Stim on 100 circuits (qubits $\in [100, 10K]$, depth (d) $\in \{100, 500, 1000\}$).

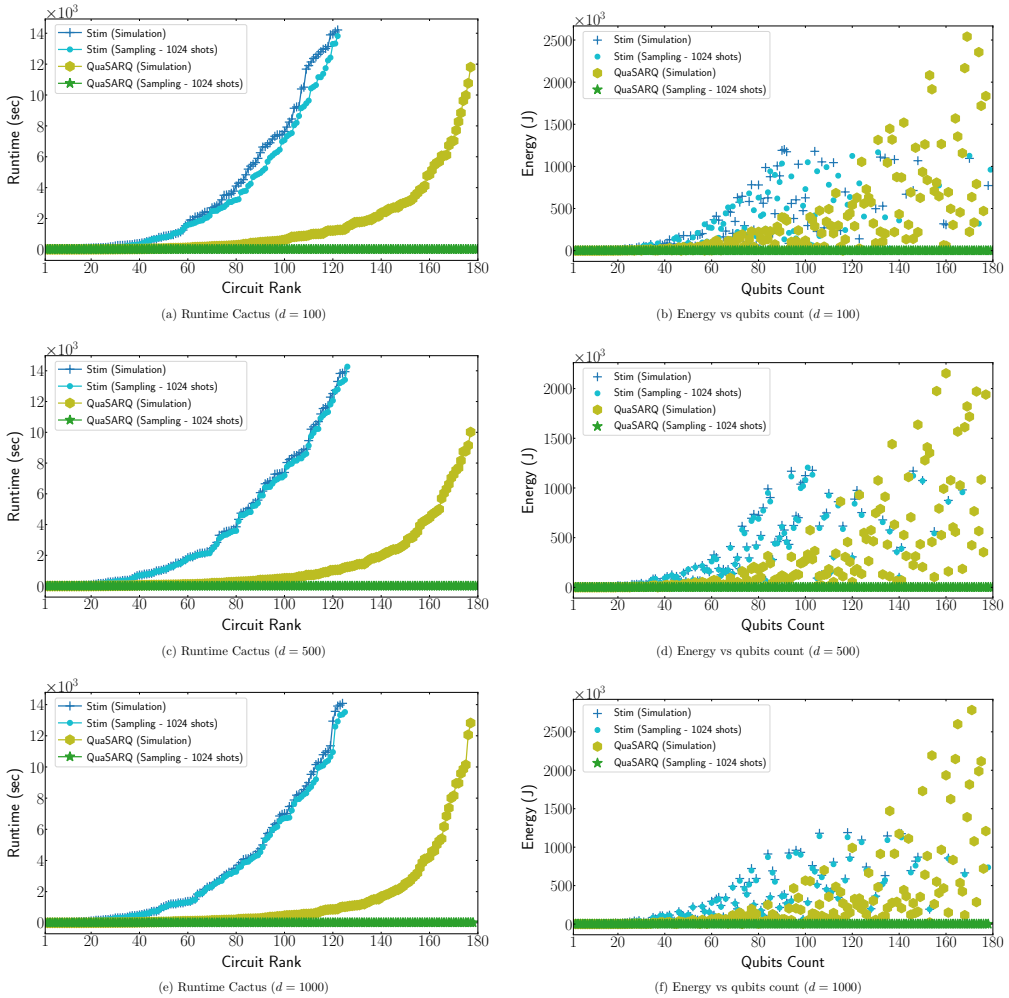
The runtime in QuaSARQ includes the initial time (memory allocations, parsing, etc.), the scheduling time to obtain $\mathcal{R}$, the transfer time to send $\mathcal{R}$ to the device, and the

simulation time of algorithm 2. To measure the energy draw, we multiplied the runtime to the average power consumed by the processing unit over the life cycle of running tools.

At depth 100, PennyLane outperforms both Qiskit-Aer (CPU) and Cirq on nearly all circuits. As qubit counts exceed roughly 4,000, Qiskit-Aer (GPU) begins to rival Qibo and Stim, occasionally surpassing them on the largest instances. Even so, QuaSARQ consistently outperforms all competitors by a substantial margin across the entire qubit range.

At depths 500 and 1,000, Qiskit-Aer (CPU) overtakes PennyLane on all circuits, while Qiskit-Aer (GPU) surpasses Qibo beyond approximately 6,000 qubits. In this regime, Stim regains an advantage over Qiskit-Aer (GPU). Nevertheless, QuaSARQ remains the fastest simulator at every tested instance, maintaining a clear lead that becomes increasingly pronounced beyond 5,000 qubits.

Regarding energy consumption, each point in the right-hand scatter plots reports total energy usage in joules (runtime multiplied by average power). QuaSARQ reduces energy consumption by approximately the same factors as its runtime speedups. Across all depths and qubit counts, it consistently consumes far less energy than both CPU- and GPU-based alternatives, reflecting not only its shorter runtimes but also the efficiency of executing measurement-intensive operations on the GPU.



(a) Runtime Cactus ($d = 100$)
(b) Energy vs qubits count ($d = 100$)
(c) Runtime Cactus ($d = 500$)
(d) Energy vs qubits count ($d = 500$)
(e) Runtime Cactus ($d = 1000$)
(f) Energy vs qubits count ($d = 1000$)

Figure 8: Simulation and sampling performance of QuaSARQ vs. Stim on 180 circuits (qubits $\in [1K, 180K]$, $d \in \{100, 500, 1000\}$).

Based on figure 7, we selected Stim as the strongest baseline for a detailed comparison with QuaSARQ on the heavy benchmark suite. Figure 8 compares the two simulators on this suite, reporting both single-shot simulation and many-shot sampling (1,024 shots) for circuits with 1K–180K qubits and depths up to 1,000. In the left-hand cactus plots, circuits are ordered by total runtime. QuaSARQ completes 177 circuits within 72 hours, whereas Stim completes only 125 circuits in 132 hours, demonstrating a substantial performance advantage at large scales.

This gap prevails for sampling: QuaSARQ 's Pauli-frame sampler exhibits flat runtime across circuit sizes, while Stim 's sampling cost increases steeply, resulting in speedups of up to several orders of magnitude. The right-hand scatter plots further show that these runtime improvements translate directly into energy savings, with QuaSARQ reducing energy consumption by over 80% on average and by more than 90% for sampling circuits.

Overall, these graphs demonstrate that our GPU-based algorithm delivers both faster runtime and dramatically lower energy consumption by over 90% savingsOn the heavy suite, QuaSARQ completes 42 more circuits within the same time window, extending the clear lead already observed on the light suite (100–10K qubits) over PennyLane, Cirq, and both CPU and GPU variants of Qiskit-Aer. Together, these findings highlight the superior scalability and efficiency of QuaSARQ across problem sizes ranging from a few hundred to nearly two hundred thousand qubits.

Figure 9 plots runtime versus qubit count (both in thousands), with color indicating the number of nondeterministic measurements per bin: darker (blue) regions correspond to fewer measurements, while warmer (red) regions indicate higher counts.

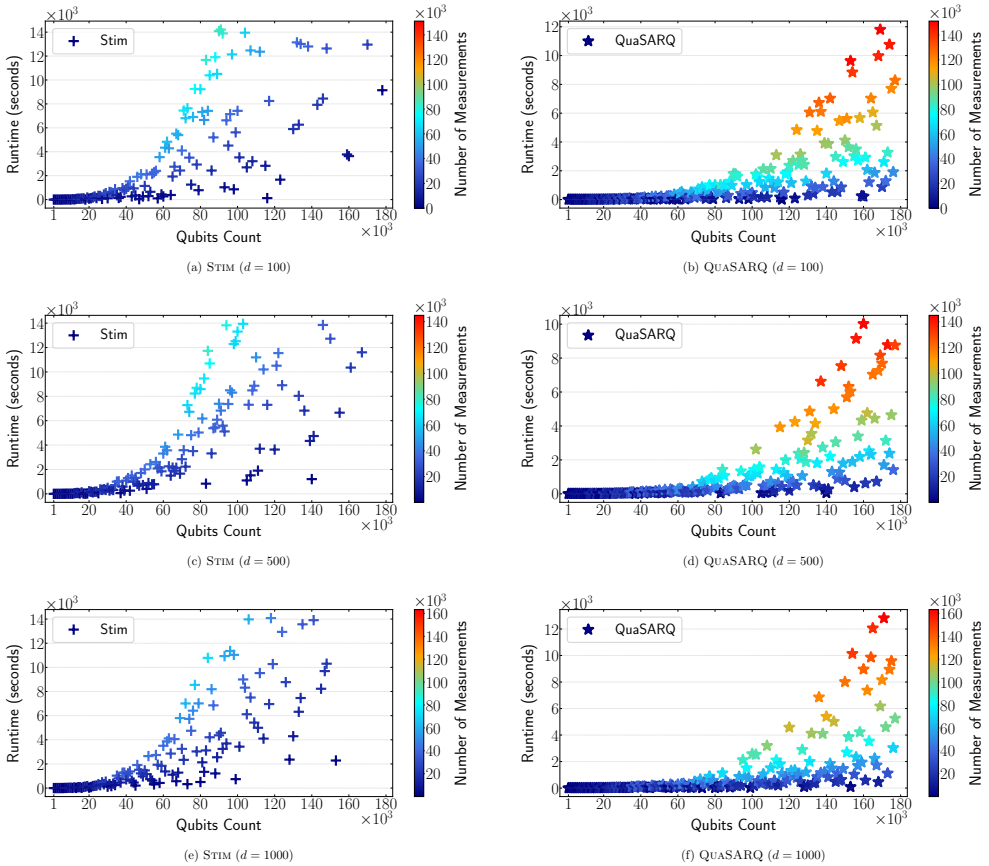


Figure 9: Distribution of successful measurements during simulation in QuaSARQ against Stim across 180 circuits (qubits $\in [1K, 180K]$, depth $(d) \in \{100, 500, 1000\}$).

In Stim 's plots, the nearly uniform dark coloring shows that most successful runs involve very few nondeterministic measurements. Even at large qubit counts, Stim rarely reaches warmer regions, indicating that it collapses only a small number of qubits nondeterministically before timing out.

In contrast, QuaSARQ exhibits a clear shift from dark to warmer colors as runtime increases. Even at 100K qubits, it successfully handles thousands of nondeterministic measurements, with the highest counts appearing in the reddest regions. These results show that QuaSARQ not only matches Stim on low-measurement instances but also scales effectively to circuits with substantially heavier nondeterminism, validating the efficiency of our pivot compaction and three-pass prefix-XOR elimination algorithms.

Table 1 reports detailed per-algorithm runtimes (in milliseconds) for 30 representative circuits (the ten fastest at each depth), comparing Stim with our GPU-based simulator. For each circuit, we decompose the runtime into tableau evolution (TO), tableau transpose (T), and Gaussian elimination (GE) in Stim, and into TO, T, pivot compaction (CMP), and GE in QuaSARQ. The final three columns report speedup ratios (Stim over QuaSARQ) for TO, T, and the combined CMP+GE step, ensuring a fair comparison by folding compaction into elimination.

Table 1: Runtime of our GPU-based simulator (QuaSARQ) against Stim on a best-performing sample of 30 circuits. All times are reported in milliseconds. Runtimes are decomposed into tableau evolution (TO), transpose (T), pivot compaction (CMP), and Gaussian elimination (GE); for a fair comparison, CMP is included in our total GE time. Bold entries indicate circuits where QuaSARQ outperforms Stim by a significant margin. The *Speedup* columns report the ratio of Stim 's runtime to QuaSARQ 's runtime.

Qubits	Depth	Gates	Measures	Stim			QuaSARQ				Speedup		
				TO	T	GE	TO	T	CMP	GE	TO	T	CMP + GE
6,000	100	418,285	5,646	94	11	5,875	**3.7**	**0.4**	**83**	**225**	**26×**	**30×**	**20×**
7,000	100	486,959	5,422	267	18	7,684	**4.4**	**0.5**	**80**	**256**	**62×**	**37×**	**23×**
8,000	100	554,796	4,225	291	24	7,762	**5.1**	**0.6**	**62**	**228**	**58×**	**40×**	**27×**
9,000	100	624,149	5,087	300	27	11,366	**5.8**	**0.8**	**75**	**332**	**52×**	**35×**	**28×**
11,000	100	764,886	7,848	451	57	23,981	**6.8**	**1.2**	**117**	**842**	**67×**	**49×**	**26×**
12,000	100	828,593	3,831	843	49	13,545	**8.1**	**1.5**	**57**	**597**	**105×**	**34×**	**21×**
13,000	100	903,631	9,242	996	61	37,033	**14.0**	**2.0**	**138**	**1,766**	**74×**	**32×**	**20×**
14,000	100	975,315	12,705	1,172	72	57,466	**20.0**	**2.5**	**189**	**2,914**	**58×**	**29×**	**19×**
86,000	100	5,928,542	15,596	26,544	2,876	1,849,630	**641.0**	**106**	**441**	**111,136**	**42×**	**28×**	**17×**
116,000	100	7,975,271	70	48,092	6,799	14,479	**1,199**	**204**	**2.5**	**1,034**	**41×**	**34×**	**14×**
8,000	500	2,755,985	6,339	1,199	15	11,408	**25.0**	**0.6**	**93.3**	**342**	**49×**	**25×**	**27×**
10,000	500	3,440,969	2,961	1,510	24	7,731	**34.0**	**1.0**	**43.5**	**227**	**46×**	**25×**	**29×**
12,000	500	4,133,639	8,561	3,053	33	30,041	**39.0**	**1.5**	**127.6**	**1,336**	**78×**	**23×**	**21×**
13,000	500	4,481,534	11,941	3,528	48	47,996	**68.0**	**2.0**	**178.4**	**2,286**	**53×**	**25×**	**20×**
14,000	500	4,824,016	11,332	4,035	53	51,287	**101.0**	**2.5**	**169.6**	**2,602**	**40×**	**21×**	**19×**
83,000	500	28,535,171	4,403	124,012	3,475	486,459	**3,117**	**101**	**123**	**33,018**	**40×**	**35×**	**15×**
86,000	500	29,585,153	20,358	132,558	2,866	2,409,790	**3,201**	**106**	**575**	**148,758**	**42×**	**27×**	**17×**
105,000	500	36,098,518	3,506	198,221	4,294	613,795	**4,889**	**165**	**102**	**42,590**	**41×**	**26×**	**15×**
107,000	500	36,783,060	5,166	203,882	4,446	911,298	**4,981**	**165**	**149**	**61,264**	**41×**	**27×**	**15×**
140,000	500	48,126,478	1,672	348,276	7,762	482,996	**8,539**	**296**	**63**	**38,240**	**41×**	**27×**	**13×**
10,000	1000	6,881,923	6,733	2,564	24	17,875	**67.0**	**1.0**	**99**	**508**	**39×**	**25×**	**30×**
58,000	1000	39,878,531	3,065	123,420	933	167,891	**3,028**	**49**	**59**	**11,435**	**41×**	**20×**	**15×**
65,000	1000	44,694,333	9,390	153,727	1,718	672,873	**3,663**	**58**	**225**	**39,768**	**42×**	**30×**	**17×**
67,000	1000	46,063,395	3,030	162,051	2,113	230,627	**4,054**	**64**	**68**	**15,046**	**40×**	**34×**	**16×**
73,000	1000	50,186,126	35	191,483	2,069	3,077	**4,801**	**78**	**1.1**	**204**	**40×**	**27×**	**16×**
80,000	1000	55,000,355	871	229,682	2,709	90,947	**5,689**	**93**	**25**	**6,220**	**41×**	**30×**	**15×**
88,000	1000	60,501,128	4,722	277,716	3,812	591,435	**6,887**	**114**	**133**	**39,784**	**41×**	**34×**	**15×**
99,000	1000	68,063,957	742	350,677	3,830	113,640	**8,710**	**147**	**21**	**7,928**	**41×**	**27×**	**15×**
128,000	1000	88,000,855	4,433	583,717	6,382	1,089,120	**14,136**	**240**	**166**	**72,111**	**42×**	**27×**	**16×**
153,000	1000	105,195,653	2,023	827,808	9,259	703,303	**20,626**	**372**	**78**	**58,590**	**41×**	**25×**	**12×**

Across this sample, QuASARQ applies gates about 105× faster at 12K qubits and depth 100 (8 ms versus 843 ms in Stim), showing that our Algorithm 3 scales linearly with qubit count while Stim's routines incur super-linear overhead. The transpose kernel maintains 20-49× speedups across all instances, thanks to our tiled shared-memory bit-shuffling (algorithm 4). In the critical collapse step, the combined CMP+GE accelerations range from roughly 30× on smaller circuits to about 12× on the largest or deepest circuits. For example, at 10K qubits and depth 1,000, we reduce Stim's 17 seconds GE into under 1 second, achieving 30× improvement. These consistent gains derived from our efficient pivots compaction (algorithm 6) and three-pass prefix-XOR elimination (algorithm 7) designs.

Author Contributions

MO designed the parallel algorithms, implemented them in the QuASARQ simulator, performed numerical analysis, and wrote the initial draft. DT co-authored the manuscript, contributed to the stabilizer-formalism foundations, presented the measurement algorithms, and verified that the technical innovations are supported by theory. AL conceived the project and supervised the research. All authors discussed the results, reviewed the manuscript, and approved the final version, taking full responsibility for the work.

AI tools were used only to generate unit tests and to check the spelling of this document.

Acknowledgements

This work was supported by the Dutch National Growth Fund (NGF), as part of the Quantum Delta NL program.

References

[1] Dimitrios Thanos, Tim Coopmans, and Alfons Laarman. "Fast Equivalence Checking of Quantum Circuits of Clifford Gates". In Étienne André and Jun Sun, editors, Automated Technology for Verification and Analysis. Pages 199–216. Cham (2023). Springer Nature Switzerland. doi: 10.1007/978-3-031-45332-8_10.

[2] Dimitrios Thanos, Alejandro Villoria, Sebastiaan Brand, Arend-Jan Quist, Jingyi Mei, Tim Coopmans, and Alfons Laarman. "Automated reasoning in quantum circuit compilation". In Thomas Neele and Anton Wijs, editors, Model Checking Software. Pages 106–134. Cham (2025). Springer Nature Switzerland. doi: 10.1007/978-3-031-66149-5_6.

[3] Tom Peham, Nina Brandl, Richard Kueng, Robert Wille, and Lukas Burgholzer. "Depth-optimal synthesis of clifford circuits with sat solvers". In 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). Volume 01, pages 802–813. (2023). doi: 10.1109/QCE57702.2023.00095.

[4] Marcus Cramer, Martin B. Plenio, Steven T. Flammia, Rolando Somma, David Gross, Stephen D. Bartlett, Olivier Landon-Cardinal, David Poulin, and Yi-Kai Liu. "Efficient quantum state tomography". Nature Communications **1**, 149 (2010). doi: 10.1038/ncomms1147.

[5] Scott Aaronson. "Shadow tomography of quantum states". SIAM Journal on Computing **49**, STOC18–368–STOC18–394 (2020). doi: 10.1137/18M120275X.

[6] M. H. Cheng, K. E. Khosla, C. N. Self, M. Lin, B. X. Li, A. C. Medina, and M. S. Kim. "Clifford Circuit Initialization For Variational Quantum Algorithms". Phys. Rev. A **111**, 062413 (2025). doi: 10.1103/PhysRevA.111.062413.

[7] Richard Jozsa. "An Introduction to Measurement Based Quantum Computation" (2005). doi: 10.48550/arXiv.quant-ph/0508124. arXiv:quant-ph/0508124.

[8] Sergey Bravyi, Graeme Smith, and John A. Smolin. "Trading Classical and Quantum Computational Resources". Phys. Rev. X **6**, 021043 (2016). doi: 10.1103/PhysRevX.6.021043.

[9] Scott Aaronson and Daniel Gottesman. "Improved Simulation of Stabilizer Circuits". Physical Review A **70** (2004). doi: 10.1103/physreva.70.052328.

[10] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. "Quantum Computing with Qiskit" (2024). doi: 10.48550/arXiv.2405.08810. arXiv:2405.08810.

[11] Craig Gidney. "Stim: A Fast Stabilizer Circuit Simulator". Quantum **5**, 497 (2021). doi: 10.22331/q-2021-07-06-497.

[12] Muhammad Osama, Dimitrios Thanos, and Alfons Laarman. "Parallel Equivalence Checking of Stabilizer Quantum Circuits on GPUs". In Arie Gurfinkel and Marijn Heule, editors, TACAS. Pages 109–128. Cham (2025). LNCS. doi: 10.1007/978-3-031-90660-2_6.

[13] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. "Surface Codes: Towards Practical Large-Scale Quantum Computation". Phys. Rev. A **86**, 032324 (2012). doi: 10.1103/PhysRevA.86.032324.

[14] Daniel Litinski. "A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery". Quantum **3**, 128 (2019). doi: 10.22331/q-2019-03-05-128.

[15] Jonas Helsen, Xiao Xue, Lieven M. K. Vandersypen, and Stephanie Wehner. "A new class of efficient randomized benchmarking protocols". npj Quantum Information **5**, 71 (2019). doi: 10.1038/s41534-019-0182-7.

[16] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. "Variational quantum algorithms". Nature Reviews Physics **3**, 625–644 (2021). doi: 10.1038/s42254-021-00348-9.

[17] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Yu Chansu. "Efficient Algorithms for Stream Compaction on GPUs". International Journal of Networking and Computing **7**, 208–226 (2017). doi: 10.15803/ijnc.7.2_208.

[18] "Cirq: An Open-Source Framework for NISQ Circuits". (2025). doi: 10.5281/zenodo.4062499.

[19] Ville Bergholm and 67 others. "PennyLane: Automatic differentiation of hybrid quantum-classical computations" (2022). doi: 10.48550/arXiv.1811.04968. arXiv:1811.04968.

[20] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, Adrián Pérez-Salinas, Diego García-Martín, Artur Garcia-Saez, José Ignacio Latorre, and Stefano Carrazza. "Qibo: A Framework for Quantum Simulation with Hardware Acceleration". Quantum Science and Technology **7**, 015018 (2021). doi: 10.1088/2058-9565/ac39f5.

[21] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Höhnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. "cuquantum sdk: A high-performance library for accelerating quantum science". In Proceedings of the 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). Pages 1050–1061. IEEE (2023). doi: 10.1109/QCE57702.2023.00119.

[22] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. "BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram". In Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS. Pages 79–94. ACM (2025). doi: 10.1145/3676641.3715984.

[23] Jingyi Mei, Marcello M. Bonsangue, and Alfons Laarman. "Simulating quantum circuits by model counting". In Arie Gurfinkel and Vijay Ganesh, editors, Computer Aided Verification. Volume 14683 of Lecture Notes in Computer Science, pages 555–578. Springer (2024). doi: 10.1007/978-3-031-65633-0_25.

[24] Aleks Kissinger and John van de Wetering. "Simulating quantum circuits with zx-calculus reduced stabiliser decompositions". Quantum Science and Technology **7**, 044001 (2022). doi: 10.1088/2058-9565/ac5d20.

[25] Daniel Gottesman. "The heisenberg representation of quantum computers". In Proceedings of the XXII International Colloquium on Group Theoretical Methods in Physics. (1998). doi: 10.48550/arXiv.quant-ph/9807006. arXiv:quant-ph/9807006.

[26] Tran Xuan Hieu Le, Tuan Hai Vu, Vu Trung Duong Le, Hoai Luan Pham, and Yasuhiko Nakashima. "Pacox: A fpga-based pauli composer accelerator for pauli string computation" (2026). doi: 10.1109/coolchips68842.2026.11556991. arXiv:2601.04827.

[27] A Yu Kitaev. "Quantum Computations: Algorithms and Error Correction". Russian Mathematical Surveys **52**, 1191 (1997). doi: 10.1070/RM1997v052n06ABEH002155.

[28] Matthew Ware, Guilhem Ribeill, Diego Ristè, Colm A. Ryan, Blake Johnson, and Marcus P. da Silva. "Experimental pauli-frame randomization on a superconducting qubit". Phys. Rev. A **103**, 042604 (2021). doi: 10.1103/PhysRevA.103.042604.

[29] E. Knill. "Quantum Computing With Realistically Noisy Devices". Nature **434**, 39–44 (2005). doi: 10.1038/nature03350.

[30] Daniel Gottesman. "Stabilizer Codes and Quantum Error Correction" (1997). doi: 10.48550/arXiv.quant-ph/9705052.

[31] Michael A. Nielsen and Isaac L. Chuang. "Quantum Computation and Quantum Information". Cambridge University Press. Cambridge, UK (2010). 10th anniversary edition edition. doi: 10.1017/CBO9780511976667.

[32] Maurice de Gosson. "Symplectic spaces and lagrangian planes". Pages 3–26. Birkhäuser Basel. Basel (2006). doi: 10.1007/3-7643-7575-2_1.

[33] NVIDIA. "CUDA C Programming Guide" (2024).

[34] Muhammad Osama and Anton Wijs. "Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking". In Bernd Finkbeiner and Laura Kovács,

editors, TACAS. Volume 14571 of Lecture Notes in Computer Science, pages 23–43. Springer (2024). doi: 10.1007/978-3-031-57249-4_2.

[35] Anton Wijs and Muhammad Osama. "The Fast and The Capacious: Memory-Efficient Multi-GPU Accelerated Explicit State Space Exploration with GPUexplore 3.0". Frontiers in High Performance Computing **7** (2024). doi: 10.3389/fhpcp.2024.1285349.

[36] Henry S. Warren Jr. "Hacker's Delight". Addison–Wesley. (2013). 2 edition. url: https://dl.acm.org/doi/10.5555/2462741.

[37] Donald E. Knuth. "The Art of Computer Programming, Volume 1: Fundamental Algorithms". Addison–Wesley. (1997). 3 edition. url: https://dl.acm.org/doi/abs/10.5555/260999.

[38] NVIDIA Corporation. "curand library". NVIDIA. (2024). url: https://docs.nvidia.com/cuda/curand/.

[39] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. "Parallel random numbers: as easy as 1, 2, 3". In Scott Lathrop, Jim Costa, and William Kramer, editors, Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011. Pages 16:1–16:12. ACM (2011). doi: 10.1145/2063384.2063405.

[40] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for The Long Term". IEEE Computer **49**, 54–63 (2016). doi: 10.1109/MC.2016.127.