# qblaze: An Efficient and Scalable Sparse Quantum Simulator

HRISTO VENEV, INSAIT, Sofia University “St. Kliment Ohridski”, Bulgaria
THIEN UDOMSRIRUNGRUANG, University of Oxford, United Kingdom and INSAIT, Sofia University “St. Kliment Ohridski”, Bulgaria
DIMITAR DIMITROV, INSAIT, Sofia University “St. Kliment Ohridski”, Bulgaria
TIMON GEHR, ETH Zurich, Switzerland
MARTIN VECHEV, ETH Zurich, Switzerland and INSAIT, Sofia University “St. Kliment Ohridski”, Bulgaria

Classical simulation of quantum circuits is critical for the development of implementations of quantum algorithms: it does not require access to specialized hardware, facilitates debugging by allowing direct access to the quantum state, and is the only way to test on inputs that are too big for current NISQ computers.

Many quantum algorithms rely on invariants that result in sparsity in the state vector. A sparse state vector simulator only computes with non-zero amplitudes. For important classes of algorithms, this results in an asymptotic improvement in simulation time. While promising prior work has investigated ways to exploit sparsity, it is still unclear what is the best way to scale sparse simulation to modern multi-core architectures.

In this work, we address this challenge and present `qblaze`, a highly optimized sparse state vector simulator based on (i) a compact sorted array representation, and (ii) new, easily parallelizable and highly-scalable algorithms for all quantum operations. Our extensive experimental evaluation shows that `qblaze` is often orders-of-magnitude more efficient than prior sparse state vector simulators even on a single thread, and also that `qblaze` scales well to a large number of CPU cores.

Overall, our work enables testing quantum algorithms on input sizes that were previously out of reach.

CCS Concepts: • **Computing methodologies** → **Quantum mechanic simulation**; **Shared memory algorithms**.

Additional Key Words and Phrases: quantum circuit simulation, sparse state vector, parallel algorithms

**ACM Reference Format:**
Hristo Venev, Thien Udomsrirungruang, Dimitar Dimitrov, Timon Gehr, and Martin Vechev. 2025. `qblaze`: An Efficient and Scalable Sparse Quantum Simulator. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 288 (October 2025), 27 pages. https://doi.org/10.1145/3763066

## 1 Introduction

Quantum algorithms exploit quantum mechanical phenomena to significantly accelerate certain computational tasks. One of the earliest examples is prime factorization—a problem for which classical approaches require exponential time, but which Shor’s quantum algorithm [57] can solve

Authors’ Contact Information: Hristo Venev, INSAIT, Sofia University “St. Kliment Ohridski”, Sofia, Bulgaria, hristo.venev@insait.ai; Thien Udomsrirungruang, University of Oxford, Oxford, United Kingdom, thien.udomsrirungruang@keble.ox.ac.uk and INSAIT, Sofia University “St. Kliment Ohridski”, Sofia, Bulgaria; Dimitar Dimitrov, INSAIT, Sofia University “St. Kliment Ohridski”, Sofia, Bulgaria, dimitar.dimitrov@insait.ai; Timon Gehr, ETH Zurich, Zurich, Switzerland, timon.gehr@inf.ethz.ch; Martin Vechev, ETH Zurich, Zurich, Switzerland and INSAIT, Sofia University “St. Kliment Ohridski”, Sofia, Bulgaria, martin.vechev@inf.ethz.ch.

ACM 2475-1421/2025/10-ART288
https://doi.org/10.1145/3763066

in polynomial time. Research has since led to polynomial[1] and superpolynomial[2] speedups on a range of important problems. These advances have the potential to transform entire fields, including chemistry [16], biology [23, 24, 52], drug design [14, 55], and cryptography [10, 11].

However, despite substantial progress [15], at the moment quantum computers are too unreliable and limited in scale for practical applications. For instance, Shor's algorithm has so far been demonstrated on quantum devices only for the numbers 15 and 21 [6, 48].[3] On the other hand, classical computing resources are much cheaper and much more abundant, and that is one of the main reasons why currently quantum computation is mostly *simulated* on classical hardware. Simulations are invaluable for validating hardware designs, and as it turns out, they are also essential for validating software designs, i.e., algorithms. First, they give the ability to inspect intermediate quantum states, which is impossible on quantum hardware. Second, quantum algorithms are usually probabilistic, which complicates testing; a classical simulator gives the option to execute a quantum computation deterministically in order to debug specific program paths.

A classical simulator of quantum computation manipulates a mathematical description of the quantum state. When measured in terms of classical resources, such a description can have an exponential size compared to the size of the quantum system. Because of the high complexity, it is important to design simulation methods that are able to harness the specific structures found in quantum computations. A notable example are stabilizer states, which can be represented efficiently thanks to the Gottesman-Knill theorem; circuits producing only stabilizer states can actually be simulated in polynomial time [2, 27, 61]. For general but less structured quantum computations, the most popular methods are based on binary decision diagrams [3, 26, 31, 33, 41, 47, 51, 58, 59, 65, 67, 69, 72, 73] and tensor networks [1, 28, 29, 45, 49, 56, 64, 66, 68, 71].

Jaques and Häner [36] recently introduced a simulation technique that applies a basic form of compression to the state vector representation of quantum states. With this method, which we shall call *sparse encoding*, only the non-zero amplitudes of the state vector are recorded. The sparse encoding leads to substantial speedups when simulating certain quantum algorithms used for factoring, discrete logarithms, and quantum chemistry (cf. [36] for more details). For example, in one instance of Shor's factoring algorithm [35], a $k$-bit number is processed using $n = 2k + 2$ qubits, yet the state vector has at most $O(2^k)$ non-zero amplitudes at all times; a sparse encoding tracks only these amplitudes, which is a *quadratic improvement* over tracking all of the $2^n$ amplitudes of the state vector, since $O(2^k) = O(2^{\frac{n}{2}})$. As a rule, such sparsity arises from classical invariants of the quantum computation. For example, if the computation involves the quantum registers $x$ and $y$, and moreover, the invariant $y = f(x)$ is guaranteed whenever the registers are measured, then the potential measurement outcomes for which $y \neq f(x)$ necessarily receive zero amplitudes.

Unfortunately, experimental results indicate that Jaques and Häner [36]'s hash table based simulator does not scale well to multiple cores, most likely due to high contention and synchronization overhead; additionally, hash tables also exhibit poor spatial locality. Concurrently with our work, Westrick et al. [70] addressed the scalability issue by using a lock-free hash table. However, this design still suffers from poor spatial locality, and when the qubit count exceeds the machine word size, extra indirection is used for storing keys so that they can be inserted atomically.

These limitations highlight the need to develop new approaches for efficient sparse simulation of quantum computations. In this work, we address this challenge and introduce `qblaze`, a scalable and highly optimized sparse state quantum circuit simulator, making the following contributions:

[1] Plain search [32], gradient estimation [37], convex optimization [7, 18], network flows and graph matching [5], etc.

[2] Number theory [9, 57], physical & chemical simulations [4, 12, 38, 42], statistical learning [19, 43, 44, 54], solving linear systems [34], etc.

[3] Larger numbers have been reported, but these approaches bake knowledge about the factors into the implementation [25, 60].

(a) Gate application cost, one thread.

| | 1 single-qubit gate | | $k$ phase/permutation gates | |
|---|---|---|---|---|
| | Time | Cache | Time | Cache |
| J&H [36] | $O(n)$ | $O(n)$ | $O(nk)$ | $O(n)$ |
| **This work** | $O(n)$ | $\mathbf{O(\frac{n}{B})}$ | $O(nk)$ | $\mathbf{O(\frac{n\min(k,\log n)}{B})}$ |

(b) Gate application cost, $p$ threads.

| | 1 single-qubit gate | | $k$ phase/permutation gates | |
|---|---|---|---|---|
| | Time | Cache | Time | Cache |
| J&H [36] | $O(n)$ | $O(n)$ | $O(\frac{nk}{p} + n)$ | $O(n)$ |
| **This work** | $\mathbf{O(\frac{n}{p} + \log n)}$ | $\mathbf{O(\frac{n}{Bp} + \log n)}$ | $\mathbf{O(\frac{nk}{p})}$ | $\mathbf{O(\frac{n\min(k,\log n)}{Bp})}$ |

Fig. 1. Asymptotic cost for the two kinds of state vector updates in qblaze, where $n$ is the number of nonzero entries and $p$ is the number of processors. The "Time" column shows the asymptotic running time, and the "Cache" column shows the asymptotic number of memory transfers assuming cache line size $B$. If possible, phase/permutation gates are queued and applied in groups. Every such group application (a queue flush) rebuilds the state vector once, which requires sorting. If the group size $k$ is smaller than $\log n$, it may be better to avoid sorting by applying the gates separately. The asymptotic analysis may be found throughout Section 4. We ignore the cost of reading the sequence of phase/permutation gates to apply because it is often negligible.

- A compact sparse encoding suitable for parallel simulation. It is based on *sorted arrays* of index-amplitude pairs, with a dynamically changing order. This ensures that threads access disjoint and contiguous blocks, eliminating contention and requiring minimal synchronization (Section 4.1).
- New, easily parallelizable, highly scalable, and cache-friendly algorithms for all quantum operations. Figure 1 summarizes the asymptotic running times in the single and multi-threaded settings, and compares them with the hash table based representation of [36] (Sections 4.2 and 4.3).
- An open-source implementation of qblaze of high quality. The implementation can be easily integrated into SDKs via highly interoperable Python and C interfaces (Section 6).
- An experimental evaluation against different types of simulators on the QASMBench benchmark suite [40], demonstrating that qblaze is competitive with the state of the art. Moreover, it is the only simulator that handles the binary welded tree benchmark for 37 qubits (Section 6.1).
- We experimentally compare with the state-of-the-art sparse simulator of Jaques and Häner on Shor's algorithm, with two different types of adder circuits.[4] Even on a *single core*, qblaze is sometimes 120× faster, on challenging problem instances (Section 6.2). At the same time, we demonstrate that our simulator can *scale linearly* up to the 180 processor cores (Section 6.3).

## 2 Overview

In this section, we give an intuitive overview of qblaze, showing its operation on a simple example. Readers unfamiliar with quantum notation can consult Section 3 first.

[4] Our numbers for Jaques and Häner's simulator are not comparable to what they originally reported due to variations in the experimental setup.

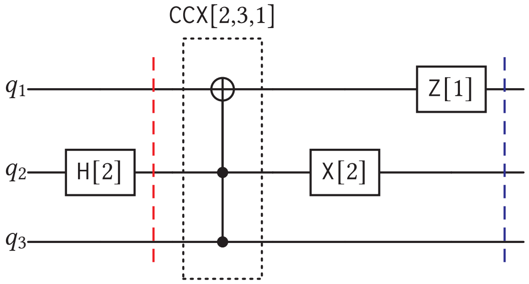

Fig. 2. A quantum circuit operating on three qubits $q_1$, $q_2$, $q_3$ via a sequence of gates. Gate operations are applied from left to right. We indicate the qubits on which each gate operates in square brackets. From the gates, three operate on a single-qubit: the Hadamard H, the bit flip X, and the conditional sign flip Z. The multi-qubit gate CCX applies a bit flip to $q_1$ (the "target") conditional on $q_2$, and $q_3$ (the "controls") being set. The gates are applied in groups as indicated by the vertical bars.

## 2.1 State Vector Simulation

As typical for state vector simulators, qblaze is implemented as an abstract data type that represents the current state vector, with operations to apply quantum gates to the state vector, and to obtain results of measurements. This way, the simulator can be treated as a quantum coprocessor whose operation is driven by a classical program, following the QRAM model [39]. This permits programs to freely interleave classical and quantum computation, a feature used in many quantum algorithms, and supported by languages like Quipper [30], QWIRE [53], Q# [46], and Silq [13].

*Supported operations.* Our simulator natively supports three kinds of operations:

- Applying an arbitrary *single-qubit gate* (e.g., $\mathsf{R}_x$ $\mathsf{R}_y$, $\mathsf{R}_z$, $X$, H). These gates transform the amplitudes in pairs whose indices differ only in the target qubit.
- Applying a *phase/permutation gate* (e.g., CCX, SWAP, $\mathsf{R}_z$, $X$). Such a gate first rotates each amplitude by an angle that depends on the index; and then permutes the indices/amplitudes; the gate can be implemented elementwise on the index-amplitude pairs.
- *Measurement* of a single qubit in the computational basis. The simulator samples a classical value, and then it collapses the state vector to the basis state with that value.

Single-qubit and phase/permutation gates form a *complete gate set*, meaning that together with measurement they are sufficient to express any quantum computation.

## 2.2 Sparse State Vectors

Consider the quantum circuit in Fig. 2. The circuit operates on $n = 3$ qubits $q_1, q_2, q_3$. An initial state vector $|\psi\rangle$ is a linear combination of $2^n$ basis vectors $|i_1 i_2 i_3\rangle$ and complex amplitudes $\psi_{i_1 i_2 i_3} \in \mathbb{C}$:

$$\begin{aligned}|\psi\rangle = \psi_{000} \cdot |000\rangle + \psi_{001} \cdot |001\rangle + \psi_{010} \cdot |010\rangle + \psi_{011} \cdot |011\rangle \\ + \psi_{100} \cdot |100\rangle + \psi_{101} \cdot |101\rangle + \psi_{110} \cdot |110\rangle + \psi_{111} \cdot |111\rangle.\end{aligned}$$

Assume that we know that $\psi_{010} = \psi_{110} = 0$. By leaving out the terms of zero amplitude, we obtain a *sparse* representation of the state vector:

$$\begin{aligned}|\psi\rangle = \psi_{000} \cdot |000\rangle + \psi_{001} \cdot |001\rangle \qquad\qquad + \psi_{011} \cdot |011\rangle \\ + \psi_{100} \cdot |100\rangle + \psi_{101} \cdot |101\rangle \qquad\qquad + \psi_{111} \cdot |111\rangle.\end{aligned}$$

Conceptually, we store a sorted array of pairs that consist of an *index* $i_1 i_2 i_3$ and the corresponding non-zero amplitude $\psi_{i_1 i_2 i_3}$. (In practice, we use two sorted arrays of pairs, cf. Section 4.) For large

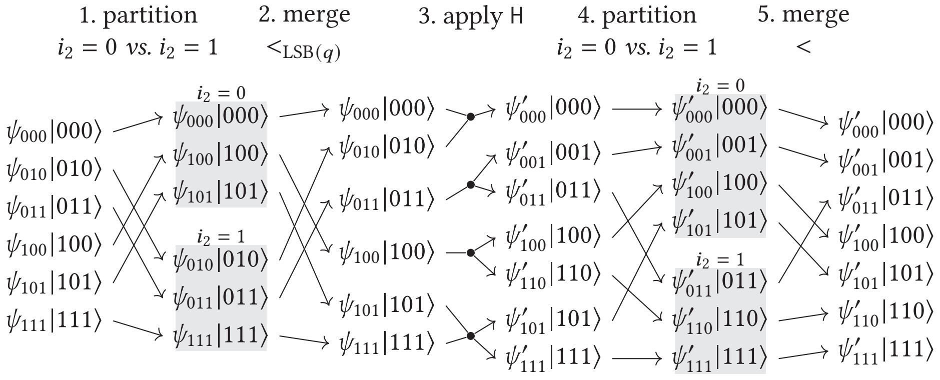

Fig. 3. Example of Hadamard gate application. The gate is applied to the second qubit of a three-qubit system. Shaded regions correspond to partitions of the state vector. In the example, two *non-zero* amplitudes $\psi_{i_1 i_2 i_3}$, and $\psi_{j_1 j_2 j_3}$ may interact only if $i_1 = j_1$, and $i_3 = j_3$. The possible interactions are illustrated by the arrows in the middle of the figure: We illustrate the possible interactions between non-zero amplitudes with the converging and diverging arrows in the center of the figure: two amplitudes turn into one, one amplitude turns into two, and two amplitudes turn into two. The important feature of the sorted state vector reprsentation is that interacting amplitudes are always situated next to each other in memory.

superpositions of $n$ qubits with many zero amplitudes, the sparse representation can be significantly more space-efficient than the naive dense encoding that stores all $2^n$ complex amplitudes.

*Advantages.* Compared to the hash-table data structure used by Jaques and Häner [36], our sorted sparse encoding has several advantages:

(1) The representation is compact and simple; there are no pointers, hashes and rehashing, or large amounts of memory allocator metadata.
(2) For most operations, the array can be split into chunks that can be processed independently in parallel; there is no need for complex and expensive synchronization.
(3) Large groups of amplitudes can be processed sequentially, which has low overhead and is cache-friendly, compared to hash tables where memory accesses are effectively random.

Next, we show how gates are applied to our example state vector. We consider the sequence of quantum gates of the circuit pictured in Fig. 2: $\mathtt{H}[2]\ ;\ \mathtt{CCX}[2,3,1]\ ;\ \mathtt{X}[2]\ ;\ \mathtt{Z}[1]$.

## 2.3 Applying Single-Qubit Gates

Let us consider how to apply the Hadamard gate $\mathtt{H}[2]$, which transforms the second qubit of our 3 qubit system. The key insight is that we can group the amplitudes into pairs that transform together. More specifically, for all $i_1$ and $i_3$, the pair $\{\psi_{i_1 0 i_2}, \psi_{i_1 1 i_2}\}$ is sent to the pair $\{\psi'_{i_1 0 i_3}, \psi'_{i_1 1 i_3}\}$:

$$\psi'_{i_1 0 i_3} = \tfrac{1}{\sqrt{2}}\psi_{i_1 0 i_3} + \tfrac{1}{\sqrt{2}}\psi_{i_1 1 i_3} \qquad\qquad \psi'_{i_1 1 i_3} = \tfrac{1}{\sqrt{2}}\psi_{i_1 0 i_3} - \tfrac{1}{\sqrt{2}}\psi_{i_1 1 i_3}.$$

We think of this transformation as an *interaction* between $\psi_{i_1 0 i_3}$ and $\psi_{i_1 1 i_3}$.

The input state vector is encoded as an array of index-amplitude pairs that are sorted according to the lexicographic ordering $\le$ of the indices. We picture this array as the first column of amplitudes and basis states (indices) on the left in Fig. 3. The figure shows the five steps of applying the gate:

(1) The first step *partitions* the array into two: one part with all pairs having $i_2 = 0$, followed by another part with all pairs having $i_2 = 1$. Importantly, the partition operation is *stable*, meaning that elements with equal $i_2$ preserve their relative ordering. The result is that both

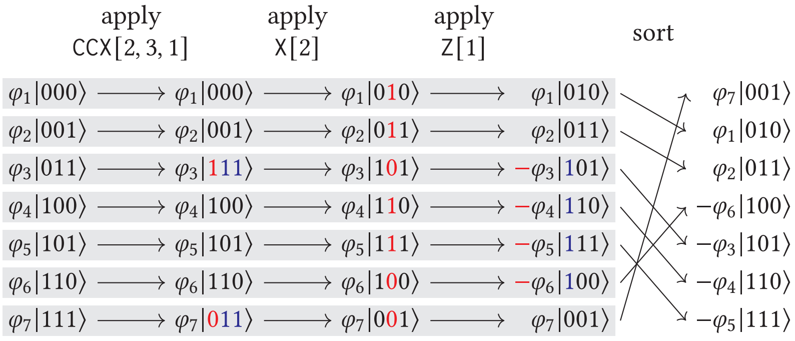
apply CCX[2, 3, 1]
apply X[2]
apply Z[1]
sort
φ1|000⟩ → φ1|000⟩ → φ1|010⟩ → φ1|010⟩
φ2|001⟩ → φ2|001⟩ → φ2|011⟩ → φ2|011⟩
φ3|011⟩ → φ3|111⟩ → φ3|101⟩ → −φ3|101⟩
φ4|100⟩ → φ4|100⟩ → φ4|110⟩ → −φ4|110⟩
φ5|101⟩ → φ5|101⟩ → φ5|111⟩ → −φ5|111⟩
φ6|110⟩ → φ6|110⟩ → φ6|100⟩ → −φ6|100⟩
φ7|111⟩ → φ7|011⟩ → φ7|001⟩ → φ7|001⟩
φ7|001⟩
φ1|010⟩
φ2|011⟩
−φ6|100⟩
−φ3|101⟩
−φ4|110⟩
−φ5|111⟩

Fig. 4. Example of applying a sequence of phase/permutation gates. The gates are applied to each element independently, as indicated by the shaded regions. The X$[m]$ gate toggles qubit $q_m$; CCX$[k, l, m]$ toggles $q_m$ provided that $q_k$, and $q_l$ are set; Z$[m]$ inverts the sign of the amplitude when $q_m$ is set. A final sorting step restores the correct order of amplitudes for further processing.

parts are sorted not only lexicographically, but also according to the modified lexicographic ordering $\preceq_{\text{LSB}(2)}$, where the second bit of $i_1 i_2 i_3$ is considered the *least significant*:

$$i_1 i_2 i_3 \preceq_{\text{LSB}(2)} j_1 j_2 j_3 \triangleq i_1 i_3 i_2 \preceq j_1 j_3 j_2.$$

(2) The second step *merges* the two parts into a single array that is sorted according to $\preceq_{\text{LSB}(2)}$. According to this ordering, the interacting amplitudes would be adjacent in the sorted array.
(3) The third step *applies* the gate by going over all interacting amplitudes and transforming them accordingly. Because we ensured that interacting amplitudes are adjacent, we not only achieve excellent spatial locality, but we also ease parallelization: We divide the array into chunks of interacting amplitudes, and then we transform the chunks in parallel.
(4) The fourth step is again a partition step, but this time of the transformed array. The partition guarantees that each of the parts is sorted according to the original lexicographic ordering $\preceq$.
(5) The fifth step merges the two parts into a single array sorted according to $\preceq$. The point is to restore the original ordering, which prepares the state for applying more gates.

The full algorithm (Section 4) actually contains a further optimization that *fuses* consecutive merge-apply-partition steps into a single pass over the array. The fusion reduces the number of passes per gate from 5 to 2. However, the state vector between applications now has to be stored as a *pair of sorted arrays* (the shaded regions in Fig. 3), instead in a single one.

## 2.4 Applying Phase/Permutation Gates

A phase/permutation gate sends each basis vector $|\bar{i}\rangle$ to the state vector $e^{i\theta(\bar{i})}|f(\bar{i})\rangle$, where the angle $\theta(\bar{i})$ is a function of the index, and $f$ is a permutation of the indices, Accordingly, each index-amplitude pair in the sorted array must be transformed as follows:

$$(\bar{i}, \psi_{\bar{i}}) \mapsto (f(\bar{i}), e^{i\theta(\bar{i})}\psi_{\bar{i}}).$$

This transformation can be done elementwise in parallel for an arbitrary sequence of phase/permutation gates (kept in a queue), as illustrated in Fig. 4. Parallelization is easy because there is absolutely no interaction between the amplitudes, as also observed by Jaques and Häner [36].

However, applying the sequence of phase/permutation gates destroys the sorting of the amplitudes. Thus, before applying the next single-qubit gate, we need to restore the sorting. In general we do that with a parallel quicksort algorithm, but sometimes a sort can be avoided altogether. For example, if only phase gates are present, then the indices are not permuted and nothing further needs to be done. Another case is when all the phase/permutation gates in the sequence update the *same* qubit $q$. In this situation, we can restore the sorting order as we did in the single-qubit case: One partition followed by one merge sorts the vector according to $\preceq_{\mathrm{LSB}(q)}$, and then another partition followed by another merge restores the original lexicographic order.

While this approach based on sorted arrays has a worse asymptotic performance than the hash-table based method of Jaques and Häner [36], it is easier to parallelize and has much better spatial locality, leading to better cache utilization and better performance overall.

### 2.5 Simulating Measurements

We also support measurement, with a stochastic or fixed outcome. Measurement is performed in stages: We first compute the probabilities of each outcome, then we sample the outcome, before discarding the elements that do not match the sampled result and renormalizing the state vector.

## 3 State Vectors, Quantum Gates, and Sparse Encodings (Preliminaries)

We next review some basics of quantum computing, and sparse state vectors encodings, focusing on formulas for gate application. The reader may consult Nielsen and Chuang [50] for more on quantum computing. We assume familiarity with classical parallel algorithms, for which we direct the reader to the 4th edition of CLRS [20].

### 3.1 State Vectors

The state of a $n$-qubit system is represented by $2^n$ complex *amplitudes* $\psi_{\bar{i}} \in \mathbb{C}$, one per bit string $\bar{i} = i_1 \dots i_n \in \{0,1\}^n$, satisfying the identity $\sum_{\bar{i}} |\psi_{\bar{i}}|^2 = 1$. For example, a state of a system that consists of 3 qubits can be described by 8 bit-string indexed amplitudes:

$$\psi_{000}, \quad \psi_{001}, \quad \psi_{010}, \quad \psi_{011}, \quad \psi_{100}, \quad \psi_{101}, \quad \psi_{110}, \quad \psi_{111}.$$

The amplitudes $\psi_{\bar{i}}$ encode the probabilities $|\psi_{\bar{i}}|^2$ for registering every possible outcome $\bar{i}$ in case the system is *measured*. When this happens, the system state *collapses* to a *basis* state $|\bar{i}\rangle$, i.e., a state left intact if the measurement is repeated. It follows that the system state is in fact represented by a unit vector $|\psi\rangle$ that is a linear combination, or a *superposition*, of the possible basis states:

$$|\psi\rangle = \sum_{\bar{i} \in \{0,1\}^n} \psi_{\bar{i}} |\bar{i}\rangle. \tag{1}$$

Below are some examples of 1-qubit and 2-qubit basis states and superpositions of basis states:

$$|0\rangle,\ |1\rangle,\ \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle \qquad\qquad |00\rangle,\ |11\rangle,\ \tfrac{1}{\sqrt{2}}|00\rangle + \tfrac{1}{\sqrt{2}}|11\rangle.$$

Besides by measurements, a systems can also be transformed by invertible linear operators $U$ that map state vectors to state vectors $|\psi'\rangle = U|\psi\rangle$. Such operators are called *unitary*. A unitary transformation is naturally represented by an $n \times n$ unitary matrix with complex entries:

$$\psi'_{\bar{i}} = \sum_{\bar{j} \in \{0,1\}^n} U_{\bar{i},\bar{j}} \psi_{\bar{j}} \qquad\qquad UU^\dagger = I \qquad\qquad U^\dagger U = I. \tag{2}$$

An example of a unitary operator is the Hadamard operator H, whose matrix is $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

Not all state vectors represent distinct physical states. More specifically, the state vectors $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ represent the same physical state, e.g., $-|0\rangle$ and $|0\rangle$. The reason is that the two cannot be discerned by an arbitrary unitary transformation followed by a measurement. Indeed, the state vectors $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ always remain related by the *phase factor* $e^{i\theta}$ after a unitary transformation, and the phase factor does not affect the measurement probabilities in any way.

### 3.2 Quantum Gates

Unitary operators can also be applied to subsystems of the whole system. To describe how this affects the whole system, suppose that there are $m + n$ qubits in total, and that we would like to transform $n$ of them using $U$. We partition the qubit identifiers $1, 2, \ldots, m + n$ into two lists,

$$\tilde{Q} = [\tilde{q}_1 < \cdots < \tilde{q}_m] \qquad\qquad Q = [q_1, \ldots, q_n]. \tag{3}$$

namely, the qubits $\tilde{q}_i \in \tilde{Q}$ outside the subsystem, and the qubits $q_i \in Q$ inside the subsystem. The partition determines the binary operation $\bar{r} *_Q \bar{i}$ that *interleaves* an $m$-bit string with an $n$-bit string:

$$i_1 i_2 \ldots i_{m+n} = \left(i_{\tilde{q}_1} \ldots i_{\tilde{q}_m}\right) *_Q \left(i_{q_1} \ldots i_{q_n}\right). \tag{4}$$

The transformation of the whole system is then given by the equation

$$\psi'_{\bar{r} *_Q \bar{i}} = \sum_{\bar{j} \in \{0,1\}^n} U_{\bar{i},\bar{j}} \psi_{\bar{r} *_Q \bar{j}}. \tag{5}$$

We call the unitary operator $U[q_1, \ldots, q_n]$ that captures this transformation a *gate*, following the convention in quantum computing. An important instance is when $n = 1$, i.e., when $U$ operates on a single qubit. In this case $(i_{\tilde{q}_1} \ldots i_{\tilde{q}_m}) *_{\{q\}} i_q = \bar{r} i_q \bar{s}$, and the gate $U[q]$ transforms the system as

$$\psi'_{\bar{r} i \bar{s}} = \sum_{j \in \{0,1\}} U_{i,j} \psi_{\bar{r} j \bar{s}}. \tag{6}$$

For example, the single-qubit Hadamard gate $\mathtt{H}[q]$ expresses the transformation

$$\psi'_{\bar{r}0\bar{s}} = \tfrac{1}{\sqrt{2}} \psi_{\bar{r}0\bar{s}} + \tfrac{1}{\sqrt{2}} \psi_{\bar{r}1\bar{s}} \qquad\qquad \psi'_{\bar{r}1\bar{s}} = \tfrac{1}{\sqrt{2}} \psi_{\bar{r}0\bar{s}} - \tfrac{1}{\sqrt{2}} \psi_{\bar{r}1\bar{s}}.$$

A subsystem $Q$ can also be transformed by $U$ upon a condition on a group of qubits $c_1, \ldots, c_l \notin Q$, called the *controls*; typically, the condition is "being in the 1 state":

$$\psi'_{\bar{r} *_Q \bar{i}} = \begin{cases} \sum_{\bar{j} \in \{0,1\}^n} U_{\bar{i},\bar{j}} \psi_{\bar{r} *_Q \bar{j}} & \text{if } r_{c_1} \ldots r_{c_l} = 1 \ldots 1, \\ \psi_{\bar{r} *_Q \bar{i}} & \text{otherwise.} \end{cases} \tag{7}$$

This transformation is captured by the *controlled* gate $\mathtt{C}^l U[c_1, \ldots, c_l, q_1, \ldots, q_n]$. For example, the operator $\mathtt{X}$, where $\mathtt{X}|0\rangle = |1\rangle$ and $\mathtt{X}|1\rangle = |0\rangle$, can be turned into the singly controlled gate:

$$|1 i_2 0\rangle \xmapsto{\mathtt{CX}[1,3]} |1 i_2 1\rangle \qquad |1 i_2 1\rangle \xmapsto{\mathtt{CX}[1,3]} |1 i_2 0\rangle \qquad |0 i_2 i_3\rangle \xmapsto{\mathtt{CX}[1,3]} |0 i_2 i_3\rangle.$$

### 3.3 Sparse Encodings and Divide-Apply-Unite

To simulate gate application (5) we need a suitable data structure that encodes the state vector in computer memory. A *sparse encoding* of the state vector implements a basic form of compression that records only the non-zero amplitudes. Conceptually, we can think of it as a data structure that encodes an *indexed set* of amplitudes, that is, a mapping from indices to amplitudes:

$$\Psi = \{\psi_{\bar{i}}\}_{\bar{i} \in I}. \tag{8}$$

Naturally, an indexed set can be implemented in many ways, for example:

- A contiguous array that records all index-amplitude pairs in some undetermined order:

$$\boxed{(\bar{i}, \psi_{\bar{i}}) \quad (\bar{j}, \psi_{\bar{j}}) \quad (\bar{k}, \psi_{\bar{k}}) \quad \cdots} \tag{9}$$

- A hash table, i.e., an array, possibly with empty cells, that keeps each pair $(\bar{i}, \psi_{\bar{i}})$ at one of several possible positions determined by hash functions $h_1$, $h_2$, $h_3$, … of the indices:

$$\begin{array}{|ccccccc|} \hline \cdots & (\bar{j}, \psi_{\bar{j}}) & \cdots & (\bar{i}, \psi_{\bar{i}}) & \cdots & (\bar{k}, \psi_{\bar{k}}) & \cdots \\ \hline \multicolumn{1}{c}{} & \uparrow & & \uparrow & & \uparrow & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & h_1(\bar{j}) & & h_1(\bar{i}) & & h_3(\bar{k}) & \multicolumn{1}{c}{} \end{array} \tag{10}$$

The question is how to support the efficient computation of (5). For that, we need to decompose the computation into simpler steps; depending on how we do that, different data structures will be more or less suitable. We shall work with the following decomposition:

**Divide** the given indexed set $\Psi$ into groups of amplitudes that are combined together in (5):

$$E_{\bar{r}} \triangleq \left\{ \psi_{\bar{r} *_Q \bar{i}} \in \Psi \right\}_{\bar{r} *_Q \bar{i} \in I}. \tag{11}$$

**Apply** the unitary $U$ to combine the grouped amplitudes, producing transformed amplitudes (5):

$$U[Q] E_{\bar{r}} = E'_{\bar{r}} \triangleq \left\{ \psi'_{\bar{r} *_Q \bar{i}} : \psi'_{\bar{r} *_Q \bar{i}} \neq 0 \right\}. \tag{12}$$

**Unite** the groups of transformed amplitudes to form the new indexed set:

$$\Psi' = \bigcup_{\bar{r}} E'_{\bar{r}}. \tag{13}$$

In the next section we shall present our main contribution, a particular sparse encoding that is designed to support these stages efficiently on current multi-core (classical) computers.

## 4 A Sparse Encoding, Algorithms, and Parallelization

When designing a sparse encoding we had two main goals in mind: (i) the encoding should possess good spatial locality, and (ii) scalable parallelization should be possible with a very simple and lightweight synchronization. We adopted these design goals to address the shortcomings of hash-table based encodings (10), e.g., used by Jaques and Häner [36], as hash-tables possess bad spatial locality, and require complicated synchronization to scale well.

The starting point for our design is the scenario where a gate $U[q]$ transforms the "last" qubit $q$. That is, in (3) $n = 1$ and $q = m + 1$. We can easily carry out this application if the indexed-set $\Psi$ (8) is encoded as a contiguous array of index-amplitude pairs (9) sorted lexicographically by the indices. Each group (11) will consist of at most two elements, who will lie next to each other in the array. Therefore, if we distribute the groups into $p$ subarrays, then the apply stage (12) can be carried out by $p$ parallel tasks. This meets both of our design goals because the tasks access contiguous memory, and require only barriers to synchronize between stages.

In Section 2.3 we discussed how this scheme extends to the case when $q$ is not the last qubit. We proceeded along the same lines, except that we temporarily sort the array in the order $\preceq_{\mathrm{LSB}(q)}$ where $q$ is considered last. We saw that this can be done in five passes over the array: a partition, a merge, the gate application stage, and then again a partition, and a merge.

While this approach is efficient compared to employing a complete sorting algorithm, we shall achieve the same effect with only two passes. The idea is that consecutive merge-apply-partition operations can be fused into a single pass over the memory. We shall use this fused operation as the primitive to express gate application; the result is the following sequence of passes:

$$\Big[\text{partition, } \underbrace{\text{merge, } \textit{apply}\text{, partition,}}_{\text{single pass}} \underbrace{\text{merge}\Big]\;\Big[\text{partition,}}_{\text{single pass}} \underbrace{\text{merge, } \textit{apply}\text{, partition,}}_{\text{single pass}} \text{merge}\Big] \tag{14}$$

The pass in the middle is a merge-apply-partition operation that applies the identity gate. This does not change the state vector, but merely reencodes it in a form suitable for the next transformation, which actually applies the gate. This way a gate application requires only 2 memory passes.

### 4.1 Data Structure

Using merge-apply-partition as a primitive means that we encode $\Psi$ as the pair of sorted arrays produced during partitioning and consumed during merging. However, in order to identify the precise invariants of this data structure, it is both more convenient and more illuminating to go beyond the single-qubit case, and describe a data structure suitable for arbitrary gates $U[Q]$. We do this merely as a presentation device; in practice, we only use the special case where $N = 1$ below.

*Organization.* Given a fixed subsystem $Q = [q_1, \dots, q_n]$ and $N \geq n$, the data structure consists of an array $S$ that references $2^N$ arrays of index-amplitude pairs, the *parts* of $S$, through the pointers

$$S[\bar{k}] \qquad\qquad \bar{k} \in \{0,1\}^N. \tag{15}$$

Each of the pointers $S[\bar{k}]$ is "fat" in the sense that it also contains two additional attributes

$$S[\bar{k}].\,head \qquad\qquad S[\bar{k}].\,tail, \tag{16}$$

that indicate respectively the first and the last position in the referenced array that contain data encoded by $S$. The data structure $S$ also has an extra attribute that indicates the subsystem $Q$:

$$S.\,subsystem = Q. \tag{17}$$

An instance $S$ may share the underlying arrays with other instances. This capability will serve us various purposes: to represent a slice of a larger collection of index-amplitude pairs; to represent a queue of index-amplitude pairs; to make cheap shared copies of an instance.

*Invariants.* A pair may appear only in one $S[\bar{k}]$, so the non-empty parts of $S$ partition the set

$$\operatorname{set}(S) = \{(\bar{i}, \psi) : \exists \bar{k} : (\bar{i}, \psi) \in S[\bar{k}]\}. \tag{18}$$

The pairs in each $S[k]$ must be sorted according to the following ordering of the indices:

$$\left(\bar{r} *_Q \bar{i}\right) \leq_Q \left(\bar{s} *_Q \bar{j}\right) \triangleq \bar{r} \leq \bar{s}. \tag{19}$$

We actually require the stronger property that for all $S[\bar{k}][a] = (\bar{r} *_Q \bar{i}, \psi)$, and $S[\bar{k}][b] = (\bar{s} *_Q \bar{j}, \psi)$

$$a \leq b \iff \bar{r} \leq \bar{s}. \tag{20}$$

So two pairs with $\bar{r} = \bar{s}$ cannot appear at distinct positions in the same part. Thus, if furthermore

$$(\bar{r} *_Q \bar{i}, \psi) \in S[\bar{k}] \implies \bar{i} = \bar{k}, \tag{21}$$

for all $\bar{k}$, then all the arrays $S[\bar{k}]$ must actually be $\leq_{Q'}$-sorted for any $Q' \subseteq Q$.

### 4.2 Applying Single-Qubit Gates

We shall present both sequential and parallel algorithms for the merge-apply-partition operation, respectively implemented as the Transform and the P-Transform procedures. The algorithms have the same interface, except for the fact that the parallel one works only for single-qubit gates. Depending on which of the two procedures we invoke, we obtain either a parallel or a sequential algorithm for applying gates. Sequential gate application looks as follows:

```
APPLY-GATE($S, U, Q$)
// assume (21) w.r.t. $S.subsystem$
if $Q \neq S.subsystem$
    $S :=$ TRANSFORM($S, \mathtt{I}, \varnothing, Q$)
return TRANSFORM($S, U, Q, Q$)
```

The three arguments are: the system state $S$ to be transformed, the unitary $U$ to be applied, and the subsystem $Q$ to which the unitary is applied. Apply-Gate requires $S$ to satisfy (21), which ensures that the parts of $S$ are sorted according to $\preceq_{\varnothing}$, that is, lexicographically. The stricter ordering is needed by the first (conditional) invocation of Transform, which reencodes the system state by applying the identity unitary $\mathtt{I}$ to the empty subsystem. The fourth argument of the call indicates the subsystem according to which the reencoded state should be partitioned. We set this to be the subsystem $Q$ to which we are applying $U$. After this has been ensured, we can proceed with the second call to Transform, which performs the application.

*4.2.1 Sequential Transform.* The left side of Fig. 5 shows the pseudocode for Transform. The procedure first creates a copy of the state $S$. We let the pointers in the copy refer to the original arrays, which is safe because we will only modify the *head* attributes, and not the arrays themselves, thus using the copy as a queue. The procedure then makes the data structure $S'$ that will hold the transformed state partitioned according to $Q'$. After that, it executes a loop for the divide-apply-unite pattern (11–13) until the queue has been exhausted. The call to Merge-Group dequeues a group $E$ of index-amplitude pairs from $S$ (divide); then, the group is transformed by $U$ (apply); finally, the transformed group $E'$ is enqueued to $S'$ by a call to the Partition-Group (unite).

For this to work, the subsystem $Q$ must be a subset of $S.subsystem$, and the parts of the input state $S$ must be $\preceq_Q$-sorted. This ensures that Merge-Group dequeues exactly the groups $E$ to which the unitary $U$ has to be applied. Furthermore, the subsystem $Q$ must be a subset of $Q'$ so that the transformed groups $E'$ are compatible with $Q'$, and also that they are given to Partition-Group in the correct order to guarantee that $S'$ satisfies the invariant (20). In turn, the partition procedure ensures the property (21), which is required by the Apply-Gate procedure.

*Cost Analysis.* For the cost analysis, we assume that the sizes of $Q$, $Q'$, and $S.subsystem$ are upper bounded by constants, and similarly for the number of arrays in $S$. In the actual implementation, the lists $Q$, $Q'$, and $S.subsystem$ contain at most 1 qubit, and $S$ contains exactly 2 arrays.

Let us first take a look at Transform in Fig. 5. Let $n = \text{Size}(S)$, i.e., the total number of elements in all arrays of $S$. The total number of iterations is $O(n)$ because in each iteration Merge-Group dequeues at least one element. The calls to Merge-Group are $O(1)$, as each inspects one element per part of $S$. The calls to Partition-Group are also $O(1)$ because $|E'| \leq 2^{|Q|}$ is constant. Therefore, the total runtime is $O(n)$.

For the number of cache misses, we need to consider the access pattern of the arrays of $S$ and $S'$. Repeated invocation of Merge-Group reads $S[\bar{k}][S[\bar{k}].head]$ and advances $S[\bar{k}].head$ sequentially. Similarly, repeated invocation of Partition-Group appends elements at the tail of $S'[\bar{i}]$, thus writing to consecutive memory locations. A constant cache size is sufficient to ensure that each cache line located in the parts of $S$ and $S'$ is read and written at most once. Therefore, there are most $O(\frac{n}{B})$ cache misses, where $B$ equals the size of a cache line.

*4.2.2 Parallel Transform.* The right side of Fig. 5 shows the pseudocode for P-Transform. Most of the pseudocode is dedicated to splitting the input and output states into chunks, and it works only for input states that contain at most two parts. We shall consider only the case of $S$ having exactly two parts, $S[0]$, and $S[1]$, as the other case is trivial.

```
Transform($S, U, Q, Q'$)
// assume $Q \subseteq S.\mathit{subsystem}$ and $Q \subseteq Q'$
// assume $S$ is $\preceq_Q$-sorted
$S := \textsc{Shared-Copy-State}(S)$
$S' := \textsc{Make-State}(Q', |Q'|)$
while $(E := \textsc{Merge-Group}(!S, Q)) \neq \varnothing$
    $E' := U[Q]\ E$
    Partition-Group($!S', E'$)
return $S'$

Partition-Group($!S, E$)
$Q := S.\mathit{subsystem}$
// assume $\forall(\bar{s} *_Q \_, \_) \in E$ :
//     $\wedge\ \forall(\bar{r} *_Q \_, \_) \in E : \bar{r} = \bar{s}$
//     $\wedge\ \forall(\bar{r} *_Q \_, \_) \in \mathrm{set}(S) : \bar{r} \preceq \bar{s}$
for each pair $(\bar{r} *_Q \bar{i}, \psi) \in E$
    $S[\bar{i}].\mathit{tail} := S[\bar{i}].\mathit{tail} + 1$
    $S[\bar{i}][S[\bar{i}].\mathit{tail}] := (\bar{r} *_Q \bar{i}, \psi)$

Merge-Group($!S, Q$)
// assume $Q \subseteq S.\mathit{subsystem}$
// assume $S$ is $\preceq_Q$-sorted
$H := \Big\{(\bar{k}, S[\bar{k}][S[\bar{k}].\mathit{head}])$ :
          $S[\bar{k}].\mathit{head} \leq S[\bar{k}].\mathit{tail}\Big\}$
$M := \Big\{\big(\_, (\bar{r} *_Q \_, \_)\big) \in H$ :
          $\forall\big(\_, (\bar{s} *_Q \_, \_)\big) \in H : \bar{r} \preceq \bar{s}\Big\}$
for each pair $(\bar{k}, \_) \in M$
    $S[\bar{k}].\mathit{head} := S[\bar{k}].\mathit{head} + 1$
return $\{e : (\_, e) \in M\}$
```

```
P-Transform($S, U, Q, Q'$)
// assume $Q \subseteq S.\mathit{subsystem}$ and $Q \subseteq Q'$
// assume $S$ is $\preceq_Q$-sorted
// assume $S$ has at most two parts
$n := \textsc{Size}(S)$
// find split points for $S$
let $SP[0:p]$ be a new array
parallel for $\iota := 0$ to $p$
    $SP[\iota] := \textsc{Find-Strict-Split-Point}(S, \preceq_Q, \lfloor \iota \frac{n}{p} \rfloor)$
// split $S$ into chunks
let $CH[1:p]$ be a new array
parallel for $\iota := 1$ to $p$
    $CH[\iota] := \textsc{Slice-State}(S, SP[\iota - 1] + 1, SP[\iota])$
// find chunk sizes for $S'$
let $SZ'[1:p]$ be a new array
parallel for $\iota := 1$ to $p$
    $SZ'[\iota] := \textsc{Trial-Transform}(CH[\iota], U, Q, Q')$
// find split points for $S'$
let $SP'[0:p]$ be a new array
P-Prefix-Sum($SZ', !SP'$)
// make $S'$
$S' := \textsc{Make-State}(Q', |Q'|, SP'[p])$
// split $S'$ into chunks
let $CH'[1:p]$ be a new array
parallel for $\iota := 1$ to $p$
    $CH'[\iota] := \textsc{Slice-State}(S', SP'[\iota - 1] + 1, SP'[\iota])$
// transform each chunk in parallel
parallel for $\iota := 1$ to $p$
    Transform-To($CH[\iota], U, Q, !CH'[i]$)
return $S'$
```

Fig. 5. Algorithms for merge-apply-parition. Transform is the sequential algorithm, and P-Transform is the parallel one. We prefix invocation arguments with ! when the invocation may modify their contents. For brevity, the sequential version omits the dynamic resizing of $S'$, which is required to accomodate newly enqueued items. In the parallel version, the parameter $p$ indicates how many chunks are processed in parallel.

The key concept we need is that of a *split point sp*. It consists of a pair of positions $sp[0]$ in $S[0]$ and $sp[1]$ in $S[1]$ that respectively split the parts into a prefix and a suffix such that all elements in the prefixes are $\preceq_Q$ than those in the suffixes. Or more formally, for all $\bar{k}, \bar{l} \in \{0, 1\}$

$$\begin{pmatrix} \forall(\bar{r} *_Q \bar{i}, \psi) \in S\big[\bar{k}\big]\big[S[\bar{k}].\mathit{head} \ :\ sp[\bar{k}]\big] \\ \forall(\bar{s} *_Q \bar{j}, \varphi) \in S\big[\bar{l}\big]\ \big[sp[\bar{l}] + 1 : S[\bar{l}].\mathit{tail}\big] \end{pmatrix} \ :\ \bar{r} \preceq \bar{s}. \tag{22}$$

The *rank* of the split point is the combined length of the prefixes. There exists at least one split point of every rank $r$ from 0 up to the number $n$ of index-amplitude pairs in $S$. To obtain one, we can start merging the two parts, and count how many elements from each we have taken so far; when we reach $r$ elements in total, we have found our split point. But the much faster method, the one we employ, is by binary search in the sorted parts: If in (22) $\bar{r} \npreceq \bar{s}$, then we know that we have to increase $sp[\bar{l}]$ and decrease $sp[\bar{k}]$; to do that, we keep track of the feasible range of values for each component of $sp$, and we halve those ranges in the corresponding direction.

Now, to split $S$ into $p$ nearly equal chunks, we can find $p + 1$ split points $SP[\iota]_{\iota=0\dots p}$ each having respective rank $\lfloor \iota \frac{n}{p} \rfloor$. Then, the $\iota$th chunk $CH[\iota]$ of $S$ will consist of the parts

$$CH[\iota][0] = S[0][SP[\iota - 1][0] + 1 : SP[\iota][0]] \tag{23}$$

$$CH[\iota][1] = S[1][SP[\iota - 1][1] + 1 : SP[\iota][1]]. \tag{24}$$

Here, however, we hit a minor complication. It might happen that some of the groups (11) that we must transform with $U$ is split between two consecutive chunks. To prevent that, we actually search for *strict split points*, that is, split points for which $\bar{r} \prec \bar{s}$ in (22). We search for a strict split point by finding a possibly non-strict split point $sp$ first. Now, the invariant (20) guarantees that both parts $S[0]$ and $S[1]$ are sorted in a strictly increasing order. It follows that $sp$ may fail to be strict in only two possible ways, both of which are easily corrected by reducing the rank by 1:

$$S[0][sp[0]] = (\bar{r} *_Q \bar{i}, \psi) \text{ and } S[1][sp[1] + 1] = (\bar{r} *_Q \bar{j}, \varphi) \implies [sp[0] - 1, sp[1]] \text{ is strict} \tag{25}$$

$$S[1][sp[1]] = (\bar{r} *_Q \bar{i}, \psi) \text{ and } S[0][sp[0] + 1] = (\bar{r} *_Q \bar{j}, \varphi) \implies [sp[0], sp[1] - 1] \text{ is strict} \tag{26}$$

After these preparations, we return to P-Transform. First, $p + 1$ strict split points of $S$ are found. They are used to slice $S$ into $p$ input chunks $CH[\iota]_{\iota=1\dots p}$. The underlying arrays are shared to avoid any copying overhead. Next, P-Transform performs a trial transformation of every input chunk in order to find out how large would each part of the corresponding output chunk be. This is similar to Transform, but instead of recording the output index-amplitude pairs, a counter for the corresponding part is incremented. The resulting pairs of sizes are stored in $SZ'[\iota]_{\iota=1\dots p}$. A parallel prefix sum (see, e.g., [20]) over the sizes yields the $p + 1$ split points $SP'[\iota]_{\iota=0\dots p}$ of the output $S'$. Then, the output $S'$ is allocated and sliced into output chunks. Finally, $U$ is applied to each input chunk by calls to Transform-To. This procedure is the same as Transform except that it does not allocate its own output data structure, but it receives the data structure as an argument.

*Cost Analysis.* For the cost analysis of P-Transform, we investigate the individual steps.

(1) The $p$ applications of Find-Strict-Split-Points. Each runs in $O(\log n)$ time and performs at most $O(\log n)$ memory operations. They are performed in parallel, so the contribution to the total cost is $O(\log n)$ time and $O(\log n)$ cache.
(2) Slice-State and Make-State are $O(1)$.
(3) The $p$ applications of Trial-Transform. Each is given a slice of at most $\left\lceil \frac{n}{p} + 1 \right\rceil < \frac{n}{p} + 2$ elements, and all $p$ are performed in parallel, so the total cost is $O(\frac{n}{p})$ time and $O(\frac{n}{Bp})$ cache.
(4) The call to P-Prefix-Sum. It runs in $O(\log n)$ time and cache.
(5) The $p$ calls to Transform-To have the same cost as Trial-Transform.

Thus, the total running time is $O(\frac{n}{p} + \log n)$ with $O(\frac{n}{Bp} + \log n)$ cache misses.

### 4.3 Applying Sequences of Phase/Permutation Gates

For phase/permutation gates the formula (5) simplifies dramatically, allowing for a specialized gate application algorithm that can efficiently handle such gates on an arbitrary number of qubits. Recall that a phase/permutation gate $U[Q]$ is characterized by an angle-valued function $\theta$, together with a permutation $f$ such that any basis state $|\bar{j}\rangle$ maps to $U[Q]|\bar{j}\rangle = e^{i\theta(\bar{j})}|f(\bar{j})\rangle$. Equivalently, the gate's matrix has exactly one non-zero unit number in every row and in every column.

```
P-PHASE/PERMUTE-PART(!$P$, $G$)
$n := P.tail - P.head + 1$
// number of chunks per processor
// $M$ is the cache size
$b := \left\lceil \frac{n}{pM} \right\rceil$
// calculate chunk boundaries
let $BD[0:p]$ be a new array
parallel for $\iota := 0$ to $bp$
    $BD[\iota] := P.head + \lfloor \iota \frac{n}{bp} \rfloor$ - 1
// transform each chunk in parallel
parallel for $\iota := 1$ to $bp$
    for each gate $U[Q] \in G$
        for $l := BD[\iota - 1] + 1$ to $BD[\iota]$
            $(\overline{j}, \psi) := P[l]$
            let $U[Q]_{\overline{i},\overline{j}} \neq 0$
            $P[l] := (\overline{i}, U[Q]_{\overline{i},\overline{j}} \psi)$
```

```
P-PHASE/PERMUTE(!$S$, $G$)
// assume (21)
// assume $S$ has two parts
$Q := \{q :$ some gate in $G$ modifies $q\}$
// prepare
if $|Q| = 1$ and $S.subsystem \neq Q$
    $S :=$ P-TRANSFORM($S$, I, $\varnothing$, $Q$)
// transform
parallel for each $\overline{k} \in \{0, 1\}$
    P-PHASE/PERMUTE-PART(!$S[\overline{k}]$, $G$)
// recover
if $|Q| = 1$
    $S :=$ P-TRANSFORM($S$, I, $Q$, $Q$)
if $|Q| > 1$
    $S.subsystem := S.subsystem \setminus Q$
    parallel for each $\overline{k} \in \{0, 1\}$
        $S[\overline{k}] :=$ P-SORT($S[\overline{k}]$, $\preceq$)
return $S$
```

Fig. 6. Algorithm for applying phase/permutation gates.

This gives us the following method for applying $U[Q]$ to an indexed set $\Psi = \{\psi_{\overline{j}}\}_{\overline{j} \in J}$: For every amplitude $\psi_{\overline{j}} \in \Psi$, we determine the column $\overline{i}$ for which the matrix entry $U[Q]_{\overline{i},\overline{j}}$ is non-zero; we calculate the amplitude $\psi'_{\overline{i}} = U[Q]_{\overline{i},\overline{j}} \psi_{\overline{j}}$, which we then add to the output indexed set $\Psi'$.

The method is very easily parallelizable when the indexed set is encoded as a contiguous array of index-amplitude pairs (9): We simply divide the array into chunks among $bp$ parallel tasks, and let every task perform the above calculation in-place. The number of tasks per processor $b$ is chosen so that each task fits in cache, meaning that each chunk is loaded once, processed entirely in cache, and stored once. In our case, the state encoding $S$ consists of multiple such arrays, and we use this method as the procedure P-PHASE/PERMUTE-PART, with pseudocode shown on the left in Fig. 6. The procedure receives the pointer $P$ of the respective part of $S$, as well as a sequence $G$ of phase/permutation gates to be applied.

Applying a sequence of phase/permutation gates to the parts of $S$ might destroy the ordering invariant (20). That is why after the application we have to restore the invariant, for which we use different recovery strategies, depending on the qubits modified by the gates in $G$:

- If no gates are modified, then (20) is preserved, and no recovery is needed.
- If only a single qubit is modified, then we can recover with P-TRANSFORM.
- If more than one qubit is modified, then we need to perform a complete sort of the parts.

Pseudocode for applying phase/permutation gates is shown on the right in Fig. 6. The procedure P-PHASE/PERMUTE recives the state $S$ to be transformed, and the sequence $G$ of gates to be applied. The result might either be a newly allocated state, or it might coincide with its argument $S$ modified in-place, depending on the recovery strategy used. The procedure has three stages. The preparation stage checks whether only one qubit has been modified. If this is so, it will perform the second recovery stage later, for which it has to ensure that $S$ is partitioned according to the modified qubit, similarly to what happens in APPLY-GATE; that is why $S$ has been assumed to satisfy (21). The transformation stage simply invokes P-PHASE/PERMUTE-PART on each part of $S$. Then it is time for the recovery stage. If a single qubit $q$ has been modified, the recovery stage reencodes the state. Here, the procedure applies the identity gate to the subsystem $\{q\}$, and not to the empty subsystem $\emptyset$, as we did earlier in Apply-Gate. The reason is that the parts of $S$ need no longer be sorted according to $\preceq_\emptyset$, but they are still guaranteed to be sorted according to $\preceq_{\{q\}}$, because only $q$ has been modified. And, if more than one qubit has been modified, the procedure recovers with a parallel sort for each of the parts; importantly, any modified qubits are removed from $S.\mathit{subsystem}$, to reflect the new ordering invariant after the permutations. At the end, both (20) and (21) hold.

*Cost Analysis.* P-Phase/Permute-Part runs in $O(\frac{nk}{p})$ time, where $k$ is the number of gates in $G$: Each gate is applied to each element once and the elements are processed in parallel.

Each of the $p$ processors (processor cores) is about to process roughly $b$ chunks. We aim to set the number of chunks so that each fits in the cache of the processor. More specifically, if the part $P$ is $n$ items long, and that the cache of a processor can hold $M$ items, we set $b = \lceil \frac{n}{pM} \rceil$, and the maximum chunk size is $\lceil \frac{n}{bp} \rceil = \lceil M - \frac{bpM-n}{bp} \rceil \leq M$.

Consequently, a task can apply each gate in $G$ to the entire chunk; when the next gate is to be applied, the items are already in the cache; thus, each item is loaded and stored into memory exactly once. (At the same time, the task minimizes the number of times it has to switch between gates.) The number of cache misses per parallel task is $O(\frac{n}{Bbp})$, and the net total for all the tasks on one processor is $O(\frac{n}{Bp})$.

Now, P-Phase/Permute-Part is called by P-Phase/Permute once for every part of $S$, which costs $O(\frac{nk}{p})$ time and $O(\frac{n}{Bp})$ cache misses, where $k$ is the number of gates in $G$. These costs dominate the costs of calling P-Transform on lines 6 and 12 in P-Phase/Permute. Thus, it only remains to account for the recovery when more than one qubit is modified by the gates in $G$, i.e., recovery by sorting. Sorting needs $O(\frac{n \log n}{p})$ time, and $O(\frac{n \log n}{Bp})$ cache misses. In case $\log n > k$, we can avoid the logarithmic factor in the time cost. We consider two cases:

(1) $1 < |Q| < \log n$. We split $G$ into subsequences for which $|Q| = 1$ [5], Since there are at most $k$ subsequences, the total time is again $O(\frac{nk}{p})$, while there are $O(\frac{nk}{Bp})$ many cache misses.

(2) $\log n \leq |Q| \leq k$. Now we need full sorting. The time cost is unaffected $O(\frac{nk+n\log n}{p}) = O(\frac{nk}{p})$. The number of cache misses, however, becomes $O(\frac{n \log n}{Bp})$.

It follows that the total time cost for $k$ phase/permutation gates is $O(\frac{nk}{p})$ and the number of cache misses is $O(\frac{n \min(k, \log n)}{Bp})$.

## 5 Gate Queue

qblaze maintains the logical state $|\psi\rangle$ of the simulated computation as a combination of a raw state vector $|\psi\rangle_{\text{raw}}$ followed by a *gate queue* $Q$ that contains gates that are not yet applied:

$$|\psi\rangle = Q \cdot |\psi\rangle_{\text{raw}}.$$

The queue is a mechanism that lets us reorder the enqueued gates, while maintaining their overall effect on the state. The primary goals are to reduce the number of groups of phase/permutation gates applied to the state vector as well as the number of single-qubit gate applications.

The gate queue itself consists of two parts: a phase/permutation part $Q_p$, logically followed by a single-qubit gate part $Q_s$; therefore, $|\psi\rangle = Q_s \cdot Q_p \cdot |\psi_{\text{raw}}\rangle$. This structure means that the queue does not store arbitrary *sequences* of gates, but only those of the form described. Whenever a new gate arrives, qblaze first checks whether it is a single-qubit gate or it is a phase-permutation gate. A single-qubit gate can always be enqueued to $Q_s$, because $Q_s$ is applied after $Q_p$. On the other

[5] This requires decomposing each SWAP gate into three CX gates. This is a constant factor increase, preserving the asymptotics.

hand, if a phase-permutation gate arrives, then it might be impossible to directly enqueue it to $Q_p$ in case the new gate does not *commute* with $Q_s$. In such situations the queue needs to be *flushed*: The new raw state becomes the logical state, and the queue becomes empty.

*Phase/permutation gate queue $Q_p$.* The first part of the gate queue is a sequence of phase/permutation gates $Q_p$. Three kinds of gates are supported natively: X, SWAP, and the parametric phase gate $\mathsf{PHASE}_\theta$. Gates can have an arbitrary number of controls, where some may be negated. The phase/permutation queue is represented as an array of instructions, where each instruction consists of a gate, target qubit(s), and controls.

*Single-qubit gate queue $Q_s$.* The other part of the gate queue is a set of single-qubit gates $Q_s$. It is represented as a partial mapping from qubit indices to single-qubit gates in the OpenQASM gate representation $\mathsf{U}(\theta, \phi, \lambda)$, which can represent any single-qubit gate (up to global phase). It is closed under composition, so at most one gate per qubit is necessary. When adding a new gate, if a gate on the same qubit already exists, the two are combined. This means that single-qubit gates never trigger a flush.

This representation improves over Jaques and Häner [36], who explicitly keep up to three fixed gates per qubit: $\mathsf{H}; \mathsf{R}_x(\theta_x); \mathsf{R}_y(\theta_y)$. By keeping a single U gate, qblaze can represent all single-qubit gates, including arbitrary compositions of rotations. This has several advantages:

- There is less ambiguity in the representation of gates. For example, $\mathsf{R}_x(\pi); \mathsf{R}_y(-\frac{\pi}{2})$ is detected as equivalent to H, meaning reordering is possible in more cases.
- Simplifications such as $\mathsf{H}; \mathsf{R}_y(-\frac{\pi}{2}) = \mathsf{Z}$ also become easy. When the result is a phase-permutation gate such as Z, it can be moved to $Q_p$, “emptying” the qubit’s entry in $Q_s$.
- Finally, a single-qubit gate is always is flushed in one step instead of up to three.

## 5.1 Circuit Reordering Optimizations

The gate queue essentially reorders the gates of the input circuit in order to update qblaze’s representation more efficiently, e.g., to apply fewer (but larger) groups of phase-permutation gates.

When a phase/permutation gate $P$ arrives, an attempt is made to commute it with the single-qubit gates in $Q_s$ and append it to $Q_p$ directly. We implement several rules:

1. All gates commute with $\mathsf{R}_z$ on their controls.
2. A gate may be moved before an X on one of its controls by negating the control. Gates commute with $\mathsf{R}_z$ on their controls, so we check if the queued gate can be represented as $\mathsf{X}; \mathsf{R}_z(\lambda)$.
3. A (multi-)controlled X gate commutes with $\mathsf{R}_x(\theta)$ on its target for all $\theta$.
4. A (multi-)controlled X gate may be moved before a $\mathsf{H}; \mathsf{R}_x(\theta)$ gate on its target by turning it into a $\mathsf{PHASE}(\pi)$ gate (where the target becomes an additional control), because X commutes with $\mathsf{R}_x(\theta)$ and $\mathsf{H}; \mathsf{X} = \mathsf{Z}; \mathsf{H}$.
5. A (multi-)controlled PHASE gate may be moved before a $\mathsf{H}; \mathsf{R}_z(\lambda)$ gate on one of its controls by turning it into an X gate, where the control becomes the target.

If none of the rules can be applied, meaning that $P$ cannot be commuted before a single-qubit gate $U[q] \in Q_s$ on some qubit $q$, we consider the following cases:

1. If $U[q]$ can be represented as a phase/permutation gate, i.e., it is either $\mathsf{R}_z(\lambda)$ or $\mathsf{X}; \mathsf{R}_z(\lambda)$, it is moved to $Q_p$. Then $P$ can be added to $Q_p$ a well.
2. Otherwise, we need to flush $U[q]$, i.e., apply it to the raw state vector. To do that, we first check whether $U[q]$ commutes with $Q_p$. If it does (e.g., when $Q_p$ does not refer to $q$, or when $U = \mathsf{R}_x(\theta)$ for some $\theta$ and $Q_p$ only uses $q$ as the target of X gates), then $U[q]$ can be applied directly.

(3) Otherwise, we flush $Q_p$ first and then $U[q]$, because that is the order in which they are applied to get the logical state.

One notable difference to Jaques and Häner [36] is that in qblaze, all single-qubit phase/permutation gates go through the single-qubit gate queue, and are only moved to the phase/permutation queue when necessary. This means that, for example, sequences of the form X; $\mathsf{R}_z$; X are transformed into a single gate, reducing the number of operations.

### 5.2 Eager Flushing

There are scenarios where it can be beneficial to flush a single-qubit gate immediately. This is the case when the gate results in destructive interference, where the number of non-zero amplitudes decreases, thus leading to a smaller state representation.

Whenever a gate in the queue needs to be flushed, we check for each enqueued (and not previously checked) single-qubit gate whether flushing it is likely to reduce the state vector size. Given a qubit $q$ to check and its single-qubit gate $U[q]$ in the queue, this check only considers a small random sample of the state vector, and it proceeds in three steps:

(1) First, we sample a few non-zero entries from the state vector:

$$(\overline{i_1}, \psi_1), (\overline{i_2}, \psi_2), \ldots, (\overline{i_n}, \psi_n) \tag{27}$$

(2) Then, we use binary search to fetch the amplitudes that would interact with the sampled ones when $U[q]$ is applied (in the process we materialize any zero amplitudes):

$$(\overline{i_1 \oplus 2^q}, \psi'_1), (\overline{i_2 \oplus 2^q}, \psi'_2), \ldots, (\overline{i_n \oplus 2^q}, \psi'_n) \tag{28}$$

(3) Finally, we check how applying $U[q]$ would change the size of the result: for each pair $(\psi_k, \psi'_k)$, we compute the resulting pair $(\varphi_k, \varphi'_k)$ and then compute the expected ratio between the size of the old state vector $S$ and the new state vector $S'$:

$$\varphi_k|\overline{i_k}\rangle + \varphi'_k|\overline{i_k \oplus 2^q}\rangle = U[q]\left(\psi_k|\overline{i_k}\rangle + \psi'_k|\overline{i_k \oplus 2^q}\rangle\right) \tag{29}$$

$$\frac{\textsc{Size}(S')}{\textsc{Size}(S)} \approx \frac{1}{n} \sum_{1 \le k \le n} \frac{[\varphi_k \ne 0] + [\varphi'_k \ne 0]}{[\psi_k \ne 0] + [\psi'_k \ne 0]} \tag{30}$$

Note that even though this operation is nondeterministic, it does not affect the simulated logical state in any way, so from the perspective of the user, the data structure is deterministic.

## 6 Experimental Evaluation

In this section we present an experimental evaluation of our simulator on a variety of circuits. We answer the following research questions:

**Q1** *Applicability*: How does qblaze generally compare to state-of-the-art CPU simulators and various other simulation approaches?
**Q2** *Efficiency*: Does qblaze outperform state-of-the-art *sparse* state vector simulators?
**Q3** *Scalability*: How well does qblaze scale to multiple threads?

*Implementation.* We implemented qblaze in Rust without relying on any third-party libraries[6]. We also provided Python bindings, through which we implemented Qiskit's Backend interface, which was used for the evaluation.

We validated the correctness of our implementation through extensive testing on random circuits. Each random circuit is given both to Qiskit Aer's statevector simulator and to qblaze, and the final state vector is compared for approximate equality up to global phase. Circuits with measurement

[6] The libc crate may optionally be used for more efficient synchronization and memory management on Linux.

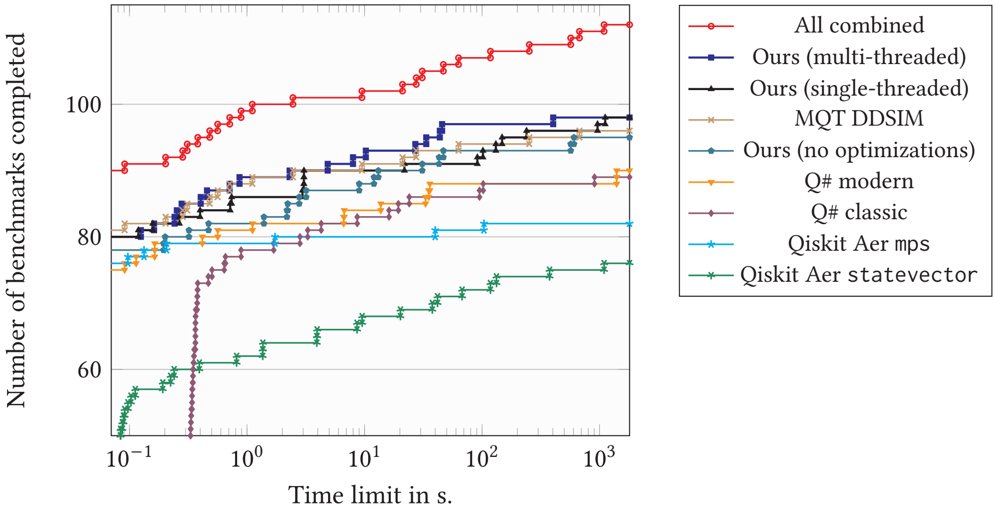

Fig. 7. Benchmark completion time (cumulative histogram). The X axis represents time in log-scale and the Y axis shows the number of benchmarks for which the runtime is at most X. Each line represents a simulator, and "All combined" uses the best time of all simulators shown.

were also tested, where in qblaze we ensured that the measurement results are the same as the ones returned by Qiskit Aer.

*Experimental setup.* We performed our evaluation on a c3d-standard-360 Google Compute Engine instance. The instance has two 90-core AMD EPYC 9B14 processors and 1440 GiB of DDR5 memory.

### 6.1 Q1: Applicability

First, we aim to demonstrate that qblaze can go head-to-head with the most performant existing CPU-based quantum simulators, on a benchmark that was not artificially selected for sparsity. Namely, we evaluate qblaze on QASMBench [40] in both single-threaded and multi-threaded mode, and compare its performance to other state-of-the art simulators:

- The Q# SparseSimulator in Microsoft's Classic QDK, the reference implementation of Jaques and Häner [36]. Note that experimental multi-threading support had been removed prior to the implementation being upstreamed.
- The Q# SparseSimulator in Microsoft's Modern QDK. Compared to the Classic QDK implementation, it uses a more efficient hash table, but lacks support for queueing phase/permutation gates.
- MQT DDSIM, a binary decision diagram simulator of high implementation quality [33].
- Qiskit Aer's statevector simulator. It is a heavily optimized dense state vector simulator. In this evaluation we used its default settings, where it uses multi-threading for circuits with 15 or more qubits. It is the only simulator in the comparison besides qblaze that supports multi-threading.
- Qiskit Aer's mps simulator. It uses a representation based on matrix product states.
- Ablation study (no optimizations): A variant of qblaze with all on-the-fly circuit optimizations disabled: no commutations are performed and single-qubit gates are applied immediately. (Note that the Q# simulators do perform such optimizations.)

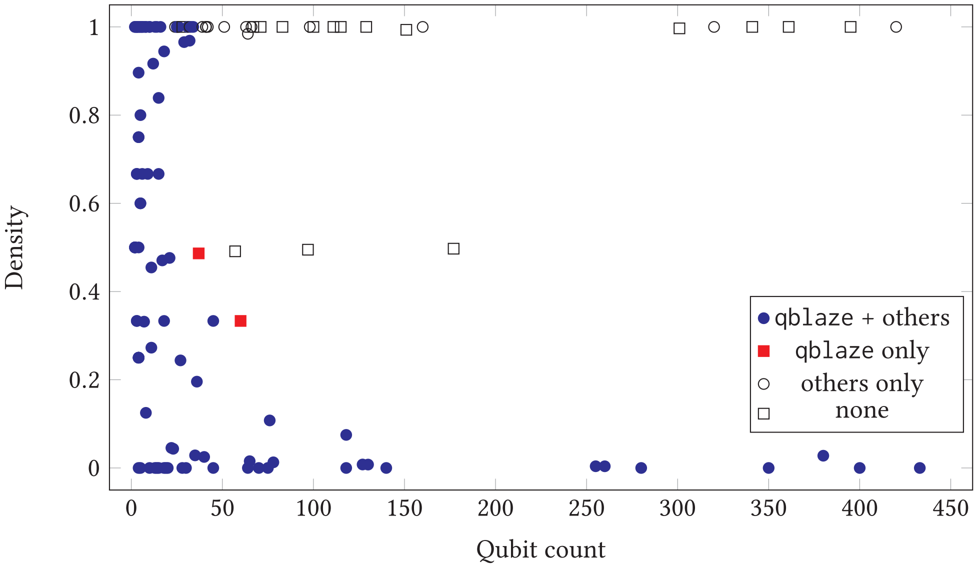

Fig. 8. Benchmark state vector sizes when run in qblaze. The X axis represents qubit count $q$ and the Y axis represents density $d$, where the maximum superposition size is $n = 2^{dq}$. For the benchmarks not completed by qblaze, we extrapolated the state vector size from smaller benchmark instances. Not visible on this graph are several benchmarks with over 1000 qubits that were not completed by any of the simulators.

For the Q# simulators, we translated all benchmarks from OpenQASM to Q# in a relatively straightforward manner. Several of the resulting OpenQASM benchmarks could however not be executed using the classic Q# QDK, as it is very slow at compiling large circuits.

*Results.* In Fig. 7 we show the number of benchmarks that each of the simulators can complete within each given time limit. We see that qblaze managed to complete the highest number of 98 benchmarks within the maximal time limit of 30 minutes. The multi-threaded version of qblaze achieves this result already after less than 3 minutes. For two of the benchmarks (bwt_n37 and square_root_n60), qblaze was the *only* tool that managed to complete them within the time limit. In Fig. 8 we show the benchmarks completed by qblaze by qubit count and state vector size.

*DDSIM.* qblaze solves a similar number of benchmarks as MQT DDSIM. However, the selections of benchmarks they manage to finish are somewhat different.

For some benchmarks (cc_*, ising_*, and qft_*), MQT DDSIM finished quickly even on very large instances, in contrast to qblaze. The states produced by these benchmarks are indeed not sparse but are efficiently representable as BDDs. For the QFT benchmark, however, this most likely happened due to the simplicity of the initial state $|0\ldots0\rangle$ – for a more complicated initial state, MQT DDSIM performs worse.

For other types of benchmarks, qblaze outperformed MQT DDSIM, completing larger benchmarks within the time limit. In contrast to MQT DDSIM, in addition to the previously-mentioned bwt_n37 and square_root_n60 benchmarks, it also finished knn_n31, dnn_n33, hhl_n14, swap_test_n25, and factor247_n15. On factor247_n15, it was furthermore the fastest simulator overall, beating the second fastest (Qiskit Aer MPS) by a factor of three.

Table 1. Simulation time of Shor's algorithm (CDKM adders) for different moduli. Multi-threaded tests were run with 90 threads. *Circuits use the *best qubit order* we found for DDSIM.

| Benchmark | qblaze | | | Hash table | Q# | | DDSIM* |
|---|---|---|---|---|---|---|---|
| | default | no opt. | single-thr. | | classic | modern | |
| $N = 35$ | 0.021s | 0.020s | 0.017s | **0.016s** | 0.367s | 0.064s | 0.057s |
| $N = 55$ | 0.020s | 0.019s | 0.016s | **0.016s** | 0.395s | 0.114s | 0.084s |
| $N = 247$ | 0.041s | 0.037s | 0.037s | **0.037s** | 0.557s | 0.114s | 0.445s |
| $N = 589$ | 0.085s | **0.078s** | 0.081s | 0.081s | 0.721s | 0.264s | 4.02s |
| $N = 667$ | 0.086s | **0.079s** | 0.082s | 0.082s | 0.842s | 0.339s | 6.01s |
| $N = 2021$ | 0.129s | 0.131s | **0.126s** | 0.129s | 1.20s | 0.815s | 36.0s |
| $N = 3599$ | 0.165s | 0.167s | **0.161s** | 0.166s | 1.42s | 1.02s | 56.8s |
| $N = 14351$ | 0.228s | **0.207s** | 0.226s | 0.229s | 1.12s | 0.614s | 43.4s |
| $N = 36089$ | **0.444s** | 0.461s | 1.69s | 1.90s | 14.5s | 27.1s | > 1800s |
| $N = 216067$ | 0.565s | **0.548s** | 1.25s | 1.38s | 10.4s | 16.6s | > 1800s |
| $N = 961307$ | **1.15s** | 1.33s | 14.8s | 16.4s | 125s | > 1800s | > 1800s |
| $N = 8276453$ | **13.1s** | 18.1s | 870s | 1092s | > 1800s | > 1800s | > 1800s |
| $N = 16130813$ | **28.6s** | 41.3s | > 1800s | > 1800s | > 1800s | > 1800s | > 1800s |
| $N = 29856637$ | **53.6s** | 77.8s | > 1800s | > 1800s | > 1800s | > 1800s | > 1800s |
| $N = 51446141$ | **106s** | 157s | > 1800s | > 1800s | > 1800s | > 1800s | > 1800s |
| $N = 124099307$ | **151s** | 227s | > 1800s | > 1800s | > 1800s | > 1800s | > 1800s |

*Ablation.* In qblaze with no circuit optimizations, simulating the three largest Bernstein-Vazirani circuits bv_70, bv_140, and bv_280 failed with an out-of-memory error. On those circuits the commutations performed by the gate queue (both our gate queue and the two Q# simulators) makes them trivial. On ising_n34, there was a significant difference in runtime: 44 seconds with a gate queue and 10 minutes without. The most likely cause is the last part of the circuit: a series of $\mathsf{H}[q]$; $\mathsf{R}_z(0)[q]$; $\mathsf{H}[q]$ on each qubit $q$. The gate queue is able to fully optimize this away.

The ablation study demonstrates the dominant role of our fast and scalable sparse algorithms in outperforming the Q# sparse simulators.

*Summary.* Finally, we observe that while no tool managed to complete more than 98 of the benchmarks, 112 were completed by at least one tool. We therefore conclude that there are multiple viable approaches to simulating quantum circuits. It strongly depends on the problem instance which of them performs best. In many cases, the best approach was indeed sparse state vector simulation, and qblaze was overall the most well-rounded approach on QASMBench.

### 6.2 Q2: Efficiency

Next, we compare qblaze to state-of-the-art *sparse* simulators on Shor's algorithm, an important algorithm that exhibits sparsity. This comparison includes the two sparse simulators for Q#.

As MQT DDSIM in particular benefits from sparsity (in addition to other structure that is easily captured with BDDs), we also include it in this comparison. Note that for DDSIM we observe more than an order-of-magnitude difference in performance depending on the order in which the quantum registers are defined in the OpenQASM circuit. The numbers in Tables 1 and 2 reflect the *best* register order we found. (For some register orders, running times were similar to Q# modern.)

To better isolate the reason for the improvements in runtime in comparison to the sparse Q# simulators, we compare against our ablation without circuit optimizations.

Table 2. Simulation time of Shor's algorithm (Draper adders) for different moduli. Multi-threaded tests were run with 90 threads. *Circuits use the *best qubit order* we found for DDSIM.

| Benchmark | qblaze | | | Hash table | Q# | | DDSIM* |
|---|---|---|---|---|---|---|---|
| | default | no opt. | single-thr. | | classic | modern | |
| $N = 35$ | 0.362s | 0.360s | 0.358s | 0.419s | 1.26s | 3.02s | **0.025s** |
| $N = 55$ | 0.367s | 0.365s | 0.363s | 0.466s | 1.66s | 4.77s | **0.034s** |
| $N = 247$ | 2.40s | 3.11s | 1.09s | 2.33s | 19.0s | 89.5s | **0.155s** |
| $N = 589$ | 10.8s | 11.8s | 6.76s | 44.1s | 637s | > 1800s | **5.61s** |
| $N = 667$ | 12.0s | 12.1s | 10.5s | 77.2s | 1198s | > 1800s | **8.15s** |
| $N = 2021$ | **27.6s** | 36.8s | 108s | > 1800s | > 1800s | > 1800s | 201s |
| $N = 3599$ | **49.3s** | 72.4s | 260s | > 1800s | > 1800s | > 1800s | 724s |
| $N = 14351$ | **65.1s** | 86.8s | 359s | > 1800s | > 1800s | > 1800s | > 1800s |

*Faster hash-table simulator.* Furthermore, we implemented our own (single-threaded) sparse simulator based on hash tables, to the best of our ability eliminating non-essential implementation inefficiencies that are present in the Q# simulators (which are also single-threaded and based on hash tables):

- We queue, rewrite, and flush gates using the implementation of qblaze. This is a bit more general than the two Q# implementations and avoids certain degenerate cases, as noted in Sections 5 and 5.2.
- Like the classic Q# implementation, it is statically compiled for several different qubit counts, avoiding slow dynamic memory allocations and indirections for bit sets.
- Like the modern Q# implementation, it uses a hash table based on Google's SwissTable, which avoids dynamic memory allocations for nodes.

*Evaluation circuits.* We compare the different approaches for sparse state vector simulations on two implementations of Shor's algorithm: one based on CDKM adders [21] and one based on Draper adders [8, 22]. Both of them exhibit sparsity, but to different degrees and with different characteristics.

*CDKM (high sparsity).* The results for the CDKM-adder implementation of Shor's algorithm are shown in Table 1. When factoring $N$, the maximum superposition size is $2\lambda(N) \in O(N)$, where $\lambda(N) < N$ is the Carmichael function [17]. Our circuit uses $3\lceil\log_2 N\rceil + 4$ qubits, which means we get a cubic advantage over dense state vector simulators. On larger benchmarks, the better data structure, optimizations and parallelization of our qblaze implementation have a significant effect and qblaze consistently dominates those benchmarks, outperforming the hash-table-based simulators as well as MQT DDSIM by orders of magnitude.

*Draper (variable sparsity).* The implementation of Shor's algorithm based on Draper adders (simulation results in Table 2) has a maximum superposition size of $2^{1+\lceil\log_2 N\rceil}\lambda(N) \in O(N^2)$. This maximum superposition size is achieved when the output register of the Draper adders is in the Fourier basis. When it is in the computational basis, the number of non-zero elements is $O(N)$. On average, across all gate applications, the number of non-zero elements is $O(\frac{N^2}{\log N})$. Our circuit uses $2\lceil\log_2 N\rceil + 3$ qubits, which means that on average the state is still quite sparse. In Fig. 9 we show the per-gate application times by state vector size.

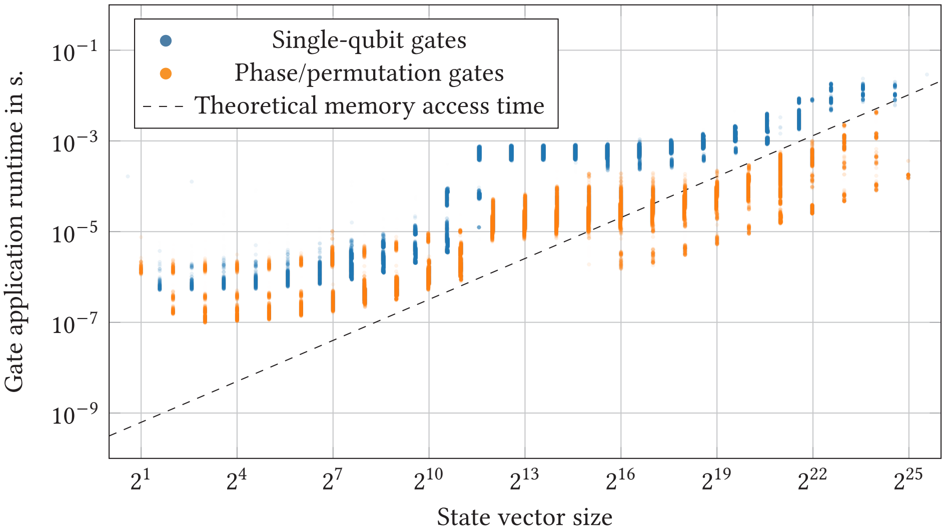

Fig. 9. Per-operation runtimes by state vector size when running Shor-Draper on $N = 14351$. For single-qubit gates the superposition size is considered to be the average of the one before and the one after applying the gate. For batches of phase/permutation gates the average per-gate application time is shown. Multi-threading (with 64 threads) is used when the superposition size is over $2^{11}$, which results in a significant increase in gate application time due to synchronization overhead. The dashed line shows the theoretical cost for six sequential passes through memory (matching the implementation of single-qubit gates), ignoring any caches.

`qblaze` simulates single-qubit gates very efficiently: it only performs a few sequential passes through memory, compared to hash tables, which require a full rebuild. This is visible in these results, where `qblaze` is significantly faster than the implementations based on hash tables, even when running in single-threaded mode. On our larger benchmarks, the single-threaded implementation even vastly outperforms the runner-up MQT DDSIM despite the qubit order being specifically selected to favor its state representation using BDDs. The higher performance of even our single-threaded implementation highlights the importance of designing state representations to be cache-friendly.

For large enough benchmarks, we often gain an additional order of magnitude in performance due to parallelization, investigated in more detail next.

### 6.3 Q3: Scalability

All operations performed by `qblaze` support multi-threading, so in the best case we can expect linear speedup. In Fig. 10 we evaluate the performance of the simulator on the two implementations of Shor's algorithm.

The evaluation was performed on a dual-socket system with 90 cores per socket and SMT support. In runs with $t \leq 90$ threads, the first $t$ physical cores of the first socket are used, and memory is allocated within the local NUMA node. For $t = 180$ and $t = 360$ threads, both sockets are used with memory interleaving. SMT is never explicitly disabled, but the scheduler of the operating system avoids it when $t \leq 180$.

*CDKM adders,* $N = 961{,}307$. In this benchmark we observe almost linear scaling up to 8 threads, which corresponds to one core complex (CCX) on our CPU. The maximum superposition size is

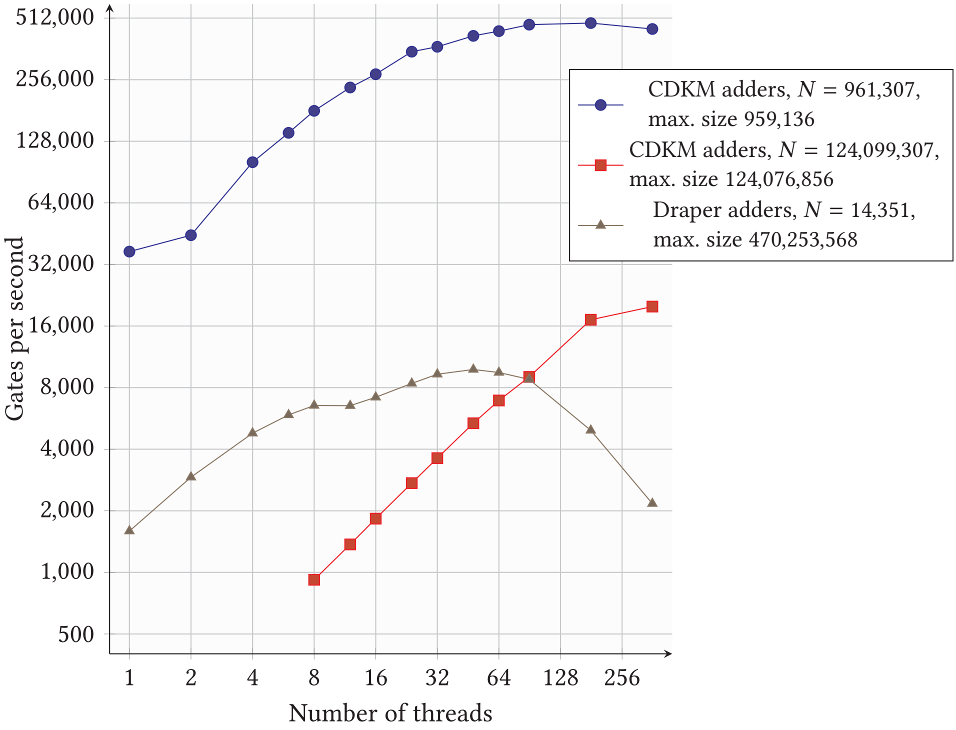

Fig. 10. Performance when simulating Shor's algorithm for different thread counts.

959,136, which at 32 bytes per element fits in the L3 cache. Above 8 threads, there is a performance penalty from using more than one CCX because the L3 cache is per-CCX.

*CDKM adders, N* = 124,099,307. Here, the maximum superposition size is roughly 130 times larger, hence gate application is more expensive. We only show results for 8 or more threads, because simulation did not finish within 30 minutes using fewer than 8 threads.

We observe approximately linear scaling up to 180 threads, with a slight improvement from enabling SMT when using 360 threads. This is possible because the superposition is large enough and most of the time is spent applying phase/permutation gates, which happens almost entirely within L2 cache. I.e., the benchmark is compute bound.

*Draper adders, N* = 14,351. This benchmark has a much higher number of non-phase/permutation gates, and a significant fraction of the time is spent applying single-qubit gates. Unlike the application of large batches of phase/permutation gates, which is mostly done in cache and is compute bound, applying single-qubit gates is done as linear passes over the entire state vector, and is bandwidth bound, i.e., it is limited by the bandwidth of system memory.

We observe almost no increase in performance past 32 threads and a decline past 48 threads. This is caused by several factors:

- Most gates are applied on superpositions that are significantly smaller than the largest ones encountered. The circuit mostly consists of Draper adders, which use a quantum Fourier transform to change between the (sparse) computational basis and the (dense) Fourier basis.

The additional synchronization overhead from more threads is comparatively more expensive on operations with smaller superposition size.

- For large superpositions, 32 threads are already enough to utilize over half of the system's theoretical memory bandwidth on large superpositions, so adding more cores suffers from diminishing returns.
- Using both CPU sockets at once has a negative effect on performance due to the increase in core-to-core latency.

Overall, our experiments demonstrate that `qblaze` is competitive with state-of-the art simulators on diverse workloads, vastly outperforms on sparse workloads previous hash-table-based sparse state vector simulators as well as MQT DDSIM, using a significantly simpler data structure, and is also able to scale to a large number of threads. In fact, `qblaze` is the first sparse state vector simulator that successfully scales to even just more than one core.

## 7 Conclusion and Future Work

We introduced `qblaze`, a highly efficient parallel classical simulator for quantum circuits. The simulator is based on a sparse state vector representation and utilizes novel efficient and parallel algorithms for all quantum operations. Our experimental evaluation of `qblaze` indicates that it is vastly more efficient than prior sparse vector simulators (often by multiple orders of magnitude). This is partly due to its ability to scale with a large numbers of cores on various classes of quantum circuits, in contrast to previous sparse simulators, which were not able to successfully scale beyond one core.

Our work opens up various interesting future research directions, including GPU implementations of our algorithms, which may significantly boost performance due to the higher memory bandwidth, as well as investigating new optimizations such as gate fusion, that reduce the number of passes through memory. Finally, sharding the state vector may permit distributed simulation.

## Data-Availability Statement

The initial version of `qblaze` and the benchmark scripts reviewed by the OOPSLA 2025 artifact evaluation committee are available at [62]. An updated version of the artifact is available at [63]. The latest version of `qblaze` is maintained in a repository at https://github.com/insait-institute/qblaze.

## Acknowledgment

This research was partially funded by the Ministry of Education and Science of Bulgaria (support for INSAIT, part of the Bulgarian National Roadmap for Research Infrastructure), as well as SNSF grant 207967. This project was supported with computational resources provided by Google Cloud Platform (GCP).

## References

[1] 2021. Efficient parallelization of tensor network contraction for simulating quantum computation. 1, 9 (2021), 578–587. doi:10.1038/s43588-021-00119-7 Publisher: Nature Publishing Group.

[2] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Physical Review A* 70, 5 (Nov. 2004), 052328. doi:10.1103/PhysRevA.70.052328

[3] A. Abdollahi and M. Pedram. 2006. Analysis and Synthesis of Quantum Circuits by Using Quantum Decision Diagrams. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 1. 1–6. doi:10.1109/DATE.2006.244176 ISSN: 1558-1101.

[4] Daniel S. Abrams and Seth Lloyd. 1999. Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors. *Physical Review Letters* 83, 24 (Dec. 1999), 5162–5165. doi:10.1103/PhysRevLett.83.5162

[5] Andris Ambainis and Robert Špalek. 2006. Quantum algorithms for matching and network flows. In *STACS 2006*, Bruno Durand and Wolfgang Thomas (Eds.). Springer, Berlin, Heidelberg, 172–183. doi:10.1007/11672142_13

[6] Mirko Amico, Zain H. Saleem, and Muir Kumph. 2019. Experimental study of Shor's factoring algorithm using the IBM Q Experience. *Physical Review A* 100, 1 (July 2019), 012305. doi:10.1103/PhysRevA.100.012305

[7] Joran van Apeldoorn, András Gilyén, Sander Gribling, and Ronald de Wolf. 2020. Convex optimization using quantum oracles. *Quantum* 4 (Jan. 2020), 220. doi:10.22331/q-2020-01-13-220

[8] Stephane Beauregard. 2003. Circuit for Shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.* 3, 2 (mar 2003), 175–185.

[9] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. 1996. Efficient networks for quantum factoring. *Physical Review A* 54, 2 (Aug. 1996), 1034–1063. doi:10.1103/PhysRevA.54.1034

[10] Charles H. Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* 560 (2014), 7–11. doi:10.1016/j.tcs.2014.05.025 Theoretical Aspects of Quantum Cryptography – celebrating 30 years of BB84.

[11] Daniel J. Bernstein. 2009. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen (Eds.). Springer, 1–14. doi:10.1007/978-3-540-88702-7_1

[12] Dominic W. Berry, Graeme Ahokas, Richard Cleve, and Barry C. Sanders. 2007. Efficient quantum algorithms for simulating sparse hamiltonians. *Communications in Mathematical Physics* 270, 2 (March 2007), 359–371. doi:10.1007/s00220-006-0150-x

[13] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, London UK, 286–300. doi:10.1145/3385412.3386007

[14] Nick S. Blunt, Joan Camps, Ophelia Crawford, Róbert Izsák, Sebastian Leontica, Arjun Mirani, Alexandra E. Moylett, Sam A. Scivier, Christoph Sünderhauf, Patrick Schopf, Jacob M. Taylor, and Nicole Holzmann. 2022. Perspective on the Current State-of-the-Art of Quantum Computing for Drug Discovery Applications. *Journal of Chemical Theory and Computation* 18, 12 (Dec. 2022), 7001–7023. doi:10.1021/acs.jctc.2c00574

[15] Gregory T. Byrd and Yongshan Ding. 2023. Quantum computing: Progress and innovation. *Computer* 56, 1 (Jan. 2023), 20–29. doi:10.1109/MC.2022.3217021

[16] Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. 2019. Quantum chemistry in the age of quantum computing. *Chemical Reviews* 119, 19 (Oct. 2019), 10856–10915. doi:10.1021/acs.chemrev.8b00803

[17] R. D. Carmichael. 1910. Note on a new number theory function. *Bull. Amer. Math. Soc.* 16, 5 (1910), 232–238. doi:10.1090/s0002-9904-1910-01892-9

[18] Shouvanik Chakrabarti, Andrew M. Childs, Tongyang Li, and Xiaodi Wu. 2020. Quantum algorithms and lower bounds for convex optimization. *Quantum* 4 (Jan. 2020), 221. doi:10.22331/q-2020-01-13-221

[19] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. 2019. Quantum convolutional neural networks. *Nature Physics* 15, 12 (Dec. 2019), 1273–1278. doi:10.1038/s41567-019-0648-8

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to algorithms* (4 ed.). The MIT Press.

[21] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. 2004. A new quantum ripple-carry addition circuit. arXiv:quant-ph/0410184 [quant-ph]

[22] Thomas G. Draper. 2000. Addition on a quantum computer. arXiv:quant-ph/0008033 [quant-ph]

[23] Prashant S. Emani, Jonathan Warrell, Alan Anticevic, Stefan Bekiranov, Michael Gandal, Michael J. McConnell, Guillermo Sapiro, Alán Aspuru-Guzik, Justin T. Baker, Matteo Bastiani, John D. Murray, Stamatios N. Sotiropoulos, Jacob Taylor, Geetha Senthil, Thomas Lehner, Mark B. Gerstein, and Aram W. Harrow. 2021. Quantum computing at the frontiers of biological sciences. *Nature Methods* 18, 7 (July 2021), 701–709. doi:10.1038/s41592-020-01004-3

[24] A. K. Fedorov and M. S. Gelfand. 2021. Towards practical applications in quantum computational biology. *Nature Computational Science* 1, 2 (Feb. 2021), 114–119. doi:10.1038/s43588-021-00024-z

[25] Michael R. Geller and Zhongyuan Zhou. 2013. Factoring 51 and 85 with 8 qubits. *Scientific Reports* 3, 1 (Oct. 2013), 3023. doi:10.1038/srep03023

[26] David Goodman, Mitchell A Thornton, David Y Feinstein, and D Michael Miller. 2007. Quantum logic circuit simulation based on the QMDD data structure. In *International Reed-Muller Workshop* (Oslo, Norway).

[27] Daniel Gottesman. 1998. Theory of fault-tolerant quantum computation. *Physical Review A* 57, 1 (Jan. 1998), 127–137. doi:10.1103/PhysRevA.57.127

[28] Johnnie Gray and Garnet Kin-Lic Chan. 2024. Hyperoptimized approximate contraction of tensor networks with arbitrary geometry. 14, 1 (2024), 011009. doi:10.1103/PhysRevX.14.011009

[29] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. 5 (2021), 410. doi:10.22331/q-2021-03-15-410

[30] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. doi:10.1145/2491956.2462177

[31] David Greve. 1999. QDD: A quantum computer emulation library. https://web.archive.org/web/20021210200357/http://home.plutonium.net/~dagreve/qdd.html

[32] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. arXiv:quant-ph/9605043 [quant-ph]

[33] Thomas Grurl, Jürgen Fuß, and Robert Wille. 2023. Noise-aware quantum circuit simulation with decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2023), 860–873. doi:10.1109/TCAD.2022.3182628

[34] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum algorithm for linear systems of equations. *Physical Review Letters* 103, 15 (Oct. 2009), 150502. doi:10.1103/PhysRevLett.103.150502

[35] Thomas Häner, Martin Roetteler, and Krysta M. Svore. 2017. Factoring using 2n+2 qubits with Toffoli based modular multiplication. arXiv:1611.07995 [quant-ph] https://arxiv.org/abs/1611.07995

[36] Samuel Jaques and Thomas Häner. 2022. Leveraging state sparsity for more efficient quantum simulations. *ACM Transactions on Quantum Computing* 3, 3 (June 2022), 15:1–15:17. doi:10.1145/3491248

[37] Stephen P. Jordan. 2005. Fast quantum algorithm for numerical gradient estimation. *Physical Review Letters* 95, 5 (July 2005), 050501. doi:10.1103/PhysRevLett.95.050501

[38] Ivan Kassal, Stephen P. Jordan, Peter J. Love, Masoud Mohseni, and Alán Aspuru-Guzik. 2008. Polynomial-time quantum algorithm for the simulation of chemical dynamics. *Proceedings of the National Academy of Sciences* 105, 48 (Dec. 2008), 18681–18686. doi:10.1073/pnas.0808245105

[39] E Knill. 1996. Conventions for quantum pseudocode. doi:10.2172/366453

[40] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing* 4, 2, Article 10 (feb 2023), 26 pages. doi:10.1145/3550488

[41] Shaowen Li, Yusuke Kimura, Hiroyuki Sato, and Masahiro Fujita. 2024. Parallelizing quantum simulation with decision diagrams. 5 (2024), 1–12. doi:10.1109/TQE.2024.3364546 Conference Name: IEEE Transactions on Quantum Engineering.

[42] Seth Lloyd. 1996. Universal quantum simulators. *Science* 273, 5278 (Aug. 1996), 1073–1078. doi:10.1126/science.273.5278.1073

[43] Seth Lloyd, Silvano Garnerone, and Paolo Zanardi. 2015. Quantum algorithms for topological and geometric analysis of big data. arXiv:arXiv:1408.3106 [quant-ph] doi:10.48550/arXiv.1408.3106

[44] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2013. Quantum algorithms for supervised and unsupervised machine learning. arXiv:arXiv:1307.0411 [quant-ph] doi:10.48550/arXiv.1307.0411

[45] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (Jan. 2008), 963–981. doi:10.1137/050644756

[46] Microsoft. 2020. *Q# Language Specification.* Retrieved Jul 7, 2023 from https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language

[47] D.M. Miller and M.A. Thornton. 2006. QMDD: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. 30–30. doi:10.1109/ISMVL.2006.35 ISSN: 2378-2226.

[48] Thomas Monz, Daniel Nigg, Esteban A. Martinez, Matthias F. Brandl, Philipp Schindler, Richard Rines, Shannon X. Wang, Isaac L. Chuang, and Rainer Blatt. 2016. Realization of a scalable Shor algorithm. *Science* 351, 6277 (March 2016), 1068–1070. doi:10.1126/science.aad9480

[49] Thien Nguyen, Dmitry Lyakh, Eugene Dumitrescu, David Clark, Jeff Larkin, and Alexander McCaskey. 2022. Tensor network quantum virtual machine for simulating quantum circuits at exascale. 4, 1 (2022), 6:1–6:21. doi:10.1145/3547334

[50] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press.

[51] Philipp Niemann, Robert Wille, and Rolf Drechsler. 2013. On the "Q" in QMDDs: Efficient representation of quantum functionality in the QMDD data-structure. In *Reversible Computation*, Gerhard W. Dueck and D. Michael Miller (Eds.). Springer, Berlin, Heidelberg, 125–140. doi:10.1007/978-3-642-38986-3_11

[52] Carlos Outeiral, Martin Strahm, Jiye Shi, Garrett M. Morris, Simon C. Benjamin, and Charlotte M. Deane. 2021. The prospects of quantum computing in computational molecular biology. *WIREs Computational Molecular Science* 11, 1 (2021), e1481. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1481 doi:10.1002/wcms.1481

[53] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 846–858. doi:10.1145/3009837.3009894

[54] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. 2014. Quantum support vector machine for big data classification. *Physical Review Letters* 113, 13 (Sept. 2014), 130503. doi:10.1103/PhysRevLett.113.130503

[55] Raffaele Santagati, Alan Aspuru-Guzik, Ryan Babbush, Matthias Degroote, Leticia González, Elica Kyoseva, Nikolaj Moll, Markus Oppel, Robert M. Parrish, Nicholas C. Rubin, Michael Streif, Christofer S. Tautermann, Horst Weiss, Nathan Wiebe, and Clemens Utschig-Utschig. 2024. Drug design on quantum computers. *Nature Physics* 20, 4 (April 2024), 549–557. doi:10.1038/s41567-024-02411-5

[56] Y.-Y. Shi, L.-M. Duan, and G. Vidal. 2006. Classical simulation of quantum many-body systems with a tree tensor network. *Physical Review A* 74, 2 (Aug. 2006), 022320. doi:10.1103/PhysRevA.74.022320

[57] Peter W. Shor. 1997. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1484–1509. doi:10.1137/s0097539795293172

[58] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. 2024. Weighted context-free-language ordered binary decision diagrams. *Weighted CFLOBDDs* 8, OOPSLA2 (Oct. 2024), 320:1390–320:1419. doi:10.1145/3689760

[59] Meghana Aparna Sistla, Swarat Chaudhuri, and Thomas Reps. 2024. CFLOBDDs: ContextfFree-language ordered binary decision diagrams. *ACM Trans. Program. Lang. Syst.* 46, 2 (May 2024), 7:1–7:82. doi:10.1145/3651157

[60] John A. Smolin, Graeme Smith, and Alexander Vargo. 2013. Oversimplifying quantum factoring. *Nature* 499, 7457 (July 2013), 163–165. doi:10.1038/nature12290

[61] Maarten Van Den Nes. 2010. Classical simulation of quantum computation, the Gottesman-Knill theorem, and slightly beyond. 10, 3 (2010), 258–271.

[62] Hristo Venev, Thien Udomsrirungruang, Dimitar Dimitrov, Timon Gehr, and Martin Vechev. 2025. *Artifact for "qblaze: An Efficient and Scalable Sparse Quantum Simulator"*. doi:10.5281/zenodo.16929865

[63] Hristo Venev, Thien Udomsrirungruang, Dimitar Dimitrov, Timon Gehr, and Martin Vechev. 2025. *Artifact for "qblaze: An Efficient and Scalable Sparse Quantum Simulator"*. doi:10.5281/zenodo.16925511

[64] F. Verstraete and J. I. Cirac. 2004. Renormalization algorithms for quantum-many body systems in two and higher dimensions. doi:10.48550/arXiv.cond-mat/0407066 arXiv:cond-mat/0407066.

[65] George F. Viamontes, Igor L. Markov, and John P. Hayes. 2003. Improving gate-level simulation of quantum circuits. *Quantum Information Processing* 2, 5 (Oct. 2003), 347–380. doi:10.1023/B:QINP.0000022725.70000.4a

[66] Guifré Vidal. 2003. Efficient classical simulation of slightly entangled quantum computations. 91, 14 (2003), 147902. doi:10.1103/PhysRevLett.91.147902

[67] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. 2023. LIMDD: A decision diagram for simulation of quantum computing including stabilizer states. *Quantum* 7 (Sept. 2023), 1108. doi:10.22331/q-2023-09-11-1108

[68] Thorsten B. Wahl and Sergii Strelchuk. 2023. Simulating quantum circuits using efficient tensor network contraction algorithms with subexponential upper bound. 131, 18 (2023), 180601. doi:10.1103/PhysRevLett.131.180601

[69] Shiou-An Wang, Chin-Yung Lu, I-Ming Tsai, and Sy-Yen Kuo. 2008. An XQDD-Based Verification Method for Quantum Circuits. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E91.A, 2 (2008), 584–594. doi:10.1093/ietfec/e91-a.2.584

[70] Sam Westrick, Pengyu Liu, Byeongjee Kang, Colin McDonald, Mike Rainey, Mingkuan Xu, Jatin Arora, Yongshan Ding, and Umut A. Acar. 2024. GraFeyn: Efficient Parallel Sparse Simulation of Quantum Circuits. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 01. 1132–1142. doi:10.1109/QCE60285.2024.00132

[71] Nadav Yoran and Anthony J. Short. 2006. Classical Simulation of Limited-Width Cluster-State Quantum Computation. *Physical Review Letters* 96, 17 (May 2006), 170503. doi:10.1103/PhysRevLett.96.170503

[72] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019-11). 1–7. doi:10.1109/ICCAD45719.2019.8942057 ISSN: 1558-2434.

[73] Alwin Zulehner and Robert Wille. 2019. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 848–859. doi:10.1109/TCAD.2018.2834427 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

Received 2025-03-25; accepted 2025-08-12