

PauliEngine: high-performant symbolic arithmetic for quantum operations

Leon Müller, Adelina Bärligea, Alexander Knapp, Jakob S. Kottmann

Angaben zur Veröffentlichung / Publication details:

Müller, Leon, Adelina Bärligea, Alexander Knapp, and Jakob S. Kottmann. 2026. "PauliEngine: high-performant symbolic arithmetic for quantum operations." In *Software Engineering 2026 (SE2026): companion proceedings; Fachtagung des GI-Fachbereichs Softwaretechnik*, 23.–27. Februar 2026, Bern, Schweiz, edited by Eduard Kamburjan, Manuel Wimmer, Manuel Ohrndorf, Pooja Rani, Thomas Thüm, Jonas Spieler, and Timo Kehler, 217–31. Bonn: Gesellschaft für Informatik e.V. https://doi.org/10.18420/se2026-ws_20.

Nutzungsbedingungen / Terms of use:

CC BY-SA 4.0



PauliEngine: High-Performant Symbolic Arithmetic for Quantum Operations

Leon Müller ¹, Adelina Bärligea ¹, Alexander Knapp ¹, and Jakob S. Kottmann ^{1,2}

Abstract: Quantum computation is inherently hybrid and fast classical manipulation of qubit operators is necessary to ensure scalability in quantum software. We introduce PauliEngine, a high-performance C++ framework that provides efficient primitives for Pauli string multiplication, commutators, symbolic phase tracking, and structural transformations. Built on a binary symplectic representation and optimized bitwise operations, PauliEngine supports both numerical and symbolic coefficients and is accessible through a Python interface. Runtime benchmarks demonstrate substantial speedups over state-of-the-art implementations. PauliEngine provides a scalable backend for operator-based quantum-software tools and simulations.

Keywords: Quantum Software, Quantum Computing, Quantum Algorithms, Operator Algebra, Scientific Computing

1 Introduction

Classical computation is and will remain a central part of quantum algorithmics. An obvious aspect is the simulation of quantum computers [Xu25], both for understanding quantum algorithms and for validating near-term hardware, but an often overlooked aspect is the classical framework responsible for assembling the classical description – *e.g.* in the form of gates and measurements – of the quantum computational protocol. In the context of simulation, there has recently been renewed interest in simulation methods that operate directly in the Pauli basis, such as Pauli propagation methods [LZ25; Ra19; Ru25] or other simulators [Go25]. Their effectiveness, however, critically depends on the availability of high-performance primitives for manipulating Pauli strings at large scale: multiplying, commuting, and transforming millions of strings with minimal runtime and memory overhead.

This paper introduces PauliEngine [PE25], a compact C++ backend designed to provide fast, memory-efficient Pauli string arithmetic for quantum-software tools. PauliEngine implements a binary symplectic representation together with optimized bit-level operations for multiplication, commutator evaluation and testing, and symbolic phase tracking. The

¹ Institute for Computer Science, University of Augsburg, 86159 Augsburg, Germany,
leon.mueller@uni-a.de,  <https://orcid.org/0009-0002-6754-9324>;
adelina.baerligea@uni-a.de,  <https://orcid.org/0009-0008-5497-1941>;
alexander.knapp@uni-a.de,  <https://orcid.org/0000-0002-4050-3249>;
jakob.kottmann@uni-a.de,  <https://orcid.org/0000-0002-4156-2048>

² Center for Advanced Analytics and Predictive Sciences, University of Augsburg, Germany,
jakob.kottmann@uni-a.de,  <https://orcid.org/0000-0002-4156-2048>

framework supports both numeric and symbolic coefficients (via SymEngine [De25]) and is provided through a lightweight Python interface suitable for integration into quantum-software workflows.

2 Preliminaries and Notation

Pauli matrices form a fundamental operator basis for describing quantum computations. Besides the unit operation I , the single-qubit matrices are

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \quad (1)$$

An N -qubit Pauli string is a tensor product of single-qubit Paulis,

$$P_{\mathbf{k}} = \bigotimes_{i=1}^N \sigma_{k_i}, \quad (2)$$

where the multi-index $\mathbf{k} = (k_1, k_2, k_3, \dots, k_N)$ takes values each in $k_i \in \{0, 1, 2, 3\}$ corresponding to the four matrices I, X, Y, Z . In the following, we will usually drop the boldface $\mathbf{k}$ when the meaning is clear and, where convenient, write $k_i \in \{I, X, Y, Z\}$ directly.

A weighted Pauli string is a pair (c, P) with complex coefficient $c \in \mathbb{C}$. Linear combinations of weighted Pauli strings are sufficient to represent all operators relevant for quantum computation, in particular: Hermitian and unitary operators.

Hermitian operators A linear combination of weighted Pauli strings,

$$H = \sum_{k=1}^M c_k P_k \quad (3)$$

is Hermitian, precisely when all coefficients c_k are real. In the same manner, if the coefficients are purely imaginary, the operator is anti-Hermitian. Because the N -qubit Paulis form an orthonormal basis for $\mathfrak{su}(2^N)$, every N -qubit observable and every generator of a quantum evolution can be written as such a combination.

Unitary operators An N -qubit quantum operation is represented by a unitary $2^N \times 2^N$ matrix. Any unitary $U \in U(2^N)$ can be written in exponential form through its generator G

$$U = e^{iG}, \quad (4)$$

where G is Hermitian and may be expanded in the Pauli basis. In practice, however, it is more convenient to work with elementary quantum gates generated by single Pauli strings.

For a weighted string (θ, P) , the associated Pauli rotation is

$$U_P(\theta) = e^{-i\frac{\theta}{2}P} = \cos\left(\frac{\theta}{2}\right)I - i\sin\left(\frac{\theta}{2}\right)P \quad (5)$$

where the factor $\frac{1}{2}$ follows a common convention in quantum-information theory. A general unitary can be expressed as a product of such gates

$$U = \prod_k U_{P_k}(\theta_k), \quad (6)$$

i.e., a quantum circuit. Pauli rotations form a universal gate set, which can be seen by Trotterizing Eq. (4) or via more specialized decompositions. For common quantum gates, the corresponding Pauli generators are listed in Eq. (1). Furthermore, given a generator G for an operation U , a controlled version on qubit m can be realized by replacing G with $\frac{1}{2}(1 - Z_m)G$.

Basic arithmetic rules The arithmetic of the data types introduced above follows directly from the algebra of Pauli matrices. At the single-qubit level,

$$\sigma_k \sigma_l = i\epsilon_{klm}\sigma_m, \quad (7)$$

with ϵ the Levi-Civita tensor. In particular, $XY = iZ$ and $XZ = iY$ and Pauli matrices anticommute whenever they differ and are non-identity.

The effect of basic arithmetic operations on the various data types is summarized in Tab. 1

Operation	P	(c, P)	Hermitian	Unitary
Multiplication	(c, P)	closed	operator	closed
Scalar Multiplication	(c, P)	closed	operator*	operator**
Addition	operator	closed	closed	operator
Conjugation	invariant	closed	invariant	closed
Unitary Transformation	Hermitian	Hermitian	closed	closed

* closed for real scalars, ** closed for unit roots

Fig. 1: Overview over datatypes and operations.

where we illustrated the transformation of the datatypes into each other via the given operations. Here, closed means the datatype stays the same, while invariant means that the datatype and content remain the same.

3 Fast Arithmetics

3.1 Efficient Data Structure for Pauli strings

Pauli operators admit several convenient data structures for symbolic computation. If a Pauli string P is given in the form of Eq. (2), then any qubit operator that is a linear combination of such strings, as in Eq. (3), can be naturally represented as a dictionary mapping the multi-index $\mathbf{k}$ to its corresponding complex coefficients. This is, for example, the native representation used by the `QubitOperator` class of `OpenFermion`.

An alternative, widely adopted format in various implementations [Ci25; DBB24; Gi21; HG25; Ja24; JBB+19; LPS25] is the binary symplectic representation. Here, an N -qubit Pauli string $P_{\mathbf{k}}$ is encoded by two N -bit vectors $\mathbf{x}$ and $\mathbf{y}$,

$$P_{\mathbf{k}} \rightarrow (\mathbf{x}, \mathbf{y}) = (x_1 x_2 x_3 \dots x_N, y_1, y_2, \dots, y_N), \quad (8)$$

where x_i indicates the presence of X - or Z -operators on qubit i , and y_i similarly tracks Y - or Z -operators. Concretely,

$$x_i = \begin{cases} 1 & \text{if } k_i \in \{X, Z\} \\ 0 & \text{else} \end{cases} \quad \text{and} \quad y_i = \begin{cases} 1 & \text{if } k_i \in \{Y, Z\} \\ 0 & \text{else.} \end{cases} \quad (9)$$

This mapping is invertible:

$$k_i = \begin{cases} I & \text{if } (x_i, y_i) = (0, 0) \\ X & \text{if } (x_i, y_i) = (1, 0) \\ Y & \text{if } (x_i, y_i) = (0, 1) \\ Z & \text{if } (x_i, y_i) = (1, 1). \end{cases} \quad (10)$$

For illustration:

$$X(2)Y(3)X(5)Z(7)Z(8) \rightarrow 01001011|00100011 \quad (11)$$

$$Z(3)X(4)Z(5) \rightarrow 00111|00101 \quad (12)$$

$$Z(2)Y(5)X(6) \rightarrow 0100001|010010, \quad (13)$$

where we used the “|” to visually separate $\mathbf{x}$ from $\mathbf{y}$.

3.2 Bitwise Multiplication and Phase Reconstruction

A key advantage of the symplectic encoding is that multiplication of Pauli strings becomes a cheap bitwise XOR-operation, denoted in the following as $\oplus$. For two strings

$$P = (\mathbf{x}, \mathbf{y}), \quad P' = (\mathbf{x}', \mathbf{y}'), \quad (14)$$

their product is another Pauli string $P'' = (\mathbf{x}'', \mathbf{y}'')$ with

$$\mathbf{x}'' = \mathbf{x} \oplus \mathbf{x}', \quad \mathbf{y}'' = \mathbf{y} \oplus \mathbf{y}'. \quad (15)$$

This reproduces exactly the multiplication table of single-qubit Pauli matrices, apart from the complex phase $\pm i$ that must be handled separately:

$*$	I	X	Y	Z
I	1	X	Y	Z
X	X	1	iZ	$-iY$
Y	Y	$-iZ$	1	iX
Z	Z	iY	$-iX$	1

(16)

For multi-qubit Pauli strings, the same logic applies qubit-wise. For example,

$$\begin{array}{llll} XYZXYZ & \cdot & YZXZXY & = & ZXYYZX \\ 101101 \, 011011 & \cdot & 011110 \, 110101 & = & 110011 \, 101110. \end{array}$$

However, the XOR-based multiplication alone does not track the global phase, which arises from the noncommutativity of single-qubit Paulis. To recover this phase using fast bit operations, we precompute truth tables indicating when a local product contributes a factor of $+i$ or $-i$ (see Fig. 2).

	x	y	x'	y'	i		x	y	x'	y'	$-i$
II	0	0	0	0	0		II	0	0	0	0
IY	0	0	1	0	0		IY	0	0	1	0
IX	0	1	0	1	0		IX	0	1	0	1
IZ	0	1	1	0	0		IZ	0	1	1	0
YI	0	1	0	0	0		YI	0	1	0	0
YY	0	1	0	1	0		YY	0	1	0	1
YX	0	1	1	0	0		YX	0	1	1	0
YZ	0	1	1	1	1		YZ	0	1	1	1
XI	1	0	0	0	0		XI	1	0	0	0
XY	1	0	0	1	1		XY	1	0	0	1
XX	1	0	1	0	0		XX	1	0	1	0
XZ	1	0	1	1	0		XZ	1	0	1	1
ZI	1	1	0	0	0		ZI	1	1	0	0
ZY	1	1	0	1	0		ZY	1	1	0	1
ZX	1	1	1	0	1		ZX	1	1	1	0
ZZ	1	1	1	1	0		ZZ	1	1	1	1

Fig. 2: Coefficient Determination Tables to determine the phase factors arising in the multiplication of two single-qubit Pauli operators in binary symplectic form. Each row corresponds to one pair of local Paulis with bit representation (x, y) and (x', y') . The left table marks the cases that contribute a factor of $+i$, while the right table marks the cases contributing $-i$.

From these tables, we obtain Boolean expressions for the bitstrings F_+ and F_- :

$$\begin{aligned} F_+ &= (\neg \mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \neg \mathbf{x}' \wedge \neg \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \neg \mathbf{y}'), \\ F_- &= (\neg \mathbf{x} \wedge \mathbf{x}' \wedge \mathbf{y} \wedge \neg \mathbf{y}') \oplus (\mathbf{x} \wedge \neg \mathbf{x}' \wedge \mathbf{y} \wedge \mathbf{y}') \oplus (\mathbf{x} \wedge \mathbf{x}' \wedge \neg \mathbf{y} \wedge \mathbf{y}'), \end{aligned} \quad (17)$$

where $|F_{\pm}|$ denotes the Hamming weight (the number of set bits). The total phase factor of the multiplication is then

$$c = i^{(|F_+| - |F_-|) \bmod 4}. \quad (18)$$

This new formula involves only one subtraction and two popcount operations. Combined with the XOR rule from before, it enables fast multiplication of arbitrary Pauli strings using purely bitwise operations.

3.3 Symbolic Extensions and Parametrizable Structures

An extension of the framework is support for *symbolic* coefficients of Pauli strings. Instead of allowing only complex numbers as coefficients, also variables are possible. This enables symbolic differentiation, analytic manipulation of parametrized operators, and delayed substitution of parameter values. For this purpose, we integrate SYMENGINE [De25], a library for high-performance symbolic manipulation.

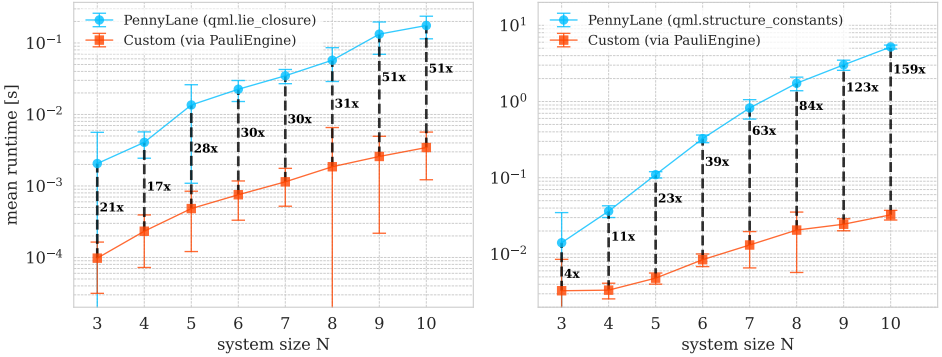


Fig. 3: Runtime benchmark of DLA computations using PauliEngine arithmetic versus PennyLane. Left: Mean runtime over 1000 runs for computing the Lie closure of DLAs isomorphic to $\mathfrak{so}(2N)$. Right: Mean runtime over 100 runs for computing the corresponding structure constants. Both axes are logarithmic. For each N , the dashed vertical markers indicate the speedup factor (PauliEngine relative to PennyLane). All benchmarks were performed on an Apple M4 processor with 24 GB RAM.

4 Initial Applications

Fast Commutators The commutator $[A, B] = AB - BA$ between two operators A and B is a fundamental concept in quantum mechanics, which determines whether two observables

can be measured simultaneously. In principle, the commutator can be evaluated directly by forming both products AB and BA . In the Pauli basis, however, computing these explicitly is unnecessary; instead, we can infer the commutation relation between two Pauli strings directly from their binary symplectic representation.

For two Pauli strings $P = (\mathbf{x}, \mathbf{y})$ and $P' = (\mathbf{x}', \mathbf{y}')$, their product produces phase factors $\pm i$ whenever both act nontrivially or anticommute on a qubit. Using the bit masks derived in Eq. 17, the multiplication routine counts, for each qubit, how often the product contributes a factor of $+i$ and $-i$. Let $F_+(P, P')$ and $F_-(P, P')$ denote these counts. Their difference $\tau = F_+ - F_-$ then determines the global phase generated by the product.

Our fast commutator implementation uses this single multiplication to decide whether the commutator vanishes:

- If τ is even, then $PP' = P'P$, and thus $[P, P'] = 0$.
- If τ is odd, then $PP' = -P'P$, and $[P, P'] = 2PP'$ up to the accumulated phase from the bitwise multiplication. No second multiplication is therefore needed.

Algorithmically, the routine performs only a single XOR to determine the resulting Pauli string PP' , two bit-summations (population counts) to compute F_+ and F_- , and finally a parity check on $\tau \bmod 2$. This yields an $O(N)$ commutator evaluation for N -qubit Pauli strings, without ever explicitly having to form both the products PP' and $P'P$.

Below, we discuss scenarios where this fast computation and check of commutators drastically reduces runtime.

Dynamical Lie Algebras Recent work has revealed a strong connection between barren plateaus (exponentially vanishing loss and gradient variance) in variational quantum algorithms (VQAs) [La24] and the dimension of the dynamical Lie algebra (DLA) generated by the ansatz [Fo24; La22; Ra24]. DLAs have since become a central tool for characterizing the expressivity and trainability of parametrized quantum circuits. Furthermore, several results indicate that ansätze provably free of barren plateaus often admit efficient classical simulation [Ce25], which is understood to arise when the associated DLA grows only polynomially with the number of qubits and therefore remains tractable to manipulate and simulate [Go25].

A VQA is defined by a parameterized quantum circuit, which can often be written in this layer-wise form,

$$U(\theta) = \prod_{l=1}^L \exp(-i\theta_l H_l), \quad (19)$$

acting on an initial state ρ_{in} and measured with an observable O . Many properties of

the ansatz depend entirely on its generator set $\mathcal{G} = \{H_l\}_{l=1}^L$. The corresponding DLA is obtained as the Lie closure,

$$\mathfrak{g} = \langle i\mathcal{G} \rangle_{\text{Lie}} = \{h_k\}_{k=1}^{\dim(\mathfrak{g})} \subseteq \mathfrak{su}(2^N), \quad (20)$$

i.e., the smallest Lie algebra containing all generators and all (non-zero) nested commutators among them. The scaling of $\dim(\mathfrak{g})$ with system size is directly linked to the scaling of the VQA loss function variance [Ra24], motivating efficient methods for explicitly computing DLAs of concrete ansätze.

Whenever $\dim(\mathfrak{g})$ scales polynomially, expectation values can be simulated efficiently through the adjoint representation of the DLA [Go25]. Such simulators evolve expectation values entirely within the linear span of $\mathfrak{g}$, with a runtime polynomial in the DLA dimension. The main computational bottleneck is the construction of the structure constants,

$$[h_\alpha, h_\beta] = \sum_\gamma f_{\alpha\beta}^\gamma h_\gamma, \quad f_{\alpha\beta}^\gamma = \frac{\langle h_\gamma, [h_\alpha, h_\beta] \rangle}{\langle h_\gamma, h_\gamma \rangle}, \quad (21)$$

with the Hilbert-Schmidt inner product $\langle A, B \rangle = \text{tr}[A^\dagger B]$. The resulting tensor f defines the full adjoint representation.

For many physically relevant ansätze, each h_k in the DLA basis is itself a single Pauli string. In this case, dense-matrix methods are unnecessarily expensive: matrix-level commutators require $O((2^N)^3)$ operations for each pair of basis elements, and identifying basis closure necessitates repeated linear-independence checks on $2^N \times 2^N$ matrices. Using Pauli representations, this exponential bottleneck can be avoided. With the techniques of the last section, the entire process becomes purely symbolic and can be accelerated significantly. Both the Lie-closure and the structure constants reduce to: (i) fast commutator evaluation between Pauli strings via bitwise operations, and (ii) constant-time dictionary lookup to detect whether the resulting Pauli string is already in the basis. No large matrices or dense linear algebra are required.

To empirically demonstrate the performance advantage, we benchmarked the DLA and adjoint-representation routines using PauliEngine against state-of-the-art PennyLane implementations [Be18] (see also the blogpost [binz]). Figure 3 shows the mean runtimes for DLAs isomorphic to $\mathfrak{so}(2N)$. Across all system sizes tested, our custom functions based on PauliEngine reduce runtime by one to two orders of magnitude. This improvement is especially pronounced for the structure constants computation, where the speedup grows from roughly a factor of 4 at $N = 3$ to nearly 160 at $N = 10$ (right panel). These gains will make DLA-based analysis and simulation practical for system sizes previously out of reach.

Commuting Cliques A major computational bottleneck in variational quantum algorithms is the amount of individual measurements necessary to determine the expectation values of interest – in electronic structure applications, this requires $O(N^4)$ individual runs to

determine a single energy value. [Go22; Pa25]. Despite recent heuristics [BK25] and randomized approaches [El23; Na23], the dominant flavor of mitigation is to decompose the Hamiltonian into commuting cliques [GK25; VYI20; YGI23; YVI20] where fast arithmetic accelerates the involved algorithms and fast commutators allow fast verification of the detected cliques.

Generator Gradients: Many applications in quantum algorithms are subjected to parametrized expectation values

$$\langle \psi(\theta) | H | \psi(\theta) \rangle = \langle 0 | U^\dagger(\theta) H U(\theta) | 0 \rangle \equiv \langle H \rangle_{U(\theta)}, \quad (22)$$

where a parametrized quantum state $\psi(\theta)$ is generated through a circuit $U(\theta)$. In the case of adaptive circuit construction, new gates $V(\varphi)$ are screened with respect to their gradient

$$\frac{\partial}{\partial \varphi} \langle H \rangle_{U(\theta)+V(\varphi)} \quad (23)$$

The gradient can either be evaluated via parameter-shift rules [Sc19] or, in the case where $\varphi = 0$, via the commutator of the operator H and the generator G or the gate V

$$\frac{\partial}{\partial \varphi} \langle H \rangle_{U(\theta)+V(\varphi)} = \frac{1}{4} \langle [H, G] \rangle_{U(\theta)}. \quad (24)$$

If the gradient at a different point is sought, and if the gate of interest is not a trailing gate in the circuit, a similar approach can be used to speed up classical computation. [JBB+19; JG20] Here, one additionally needs to consider operator folding techniques (see next section).

Operator Folding A technique that benefits significantly in practice, when fast Pauli arithmetic is available, is operator folding, typically carried out for expectation values

$$\begin{aligned} \langle H \rangle_U &= \langle 0 | U^\dagger H U | 0 \rangle = \langle 0 | U_1^\dagger U_2^\dagger U_3^\dagger U_4^\dagger H U_4 U_3 U_2 U_1 | 0 \rangle \\ &= \langle 0 | U_1^\dagger U_2^\dagger \left(U_3^\dagger U_4^\dagger H U_4 U_3 \right) U_2 U_1 | 0 \rangle = \langle \tilde{H} \rangle_{\tilde{U}} \end{aligned} \quad (25)$$

in this example $\tilde{H} = U_3^\dagger U_4^\dagger H U_4 U_3$ and $\tilde{U} = U_2 U_1$. Typically, and especially in a variational setting [An22; BH21; Ce21], the individual gates are parametrized, so we get a parametrized transformation $H \rightarrow \tilde{H}(\theta)$. Being able to perform these operations quickly is often beneficial for analysis and testing purposes. Examples are so-called Heisenberg-picture techniques, developed in various flavors [Ry20; Sh23; Zh22], circuits with Clifford tails which can, for expectation values, be folded into the operator without changing the number of Pauli strings [AB25; SCL24; SCZ25] or highly-specialized methods, like the transcorrelated Hamiltonian in electronic structure [Do24; Ku22; So23]. There are also scenarios for

non-symmetric folding, *e.g.* in transition amplitudes $\langle\psi|H|\phi\rangle$ where the different states are constructed via different circuits. A typical example in practice are non-orthogonal VQEs [Hu20; KS24; SHE20] and downfolding techniques [BPK23], where partial folding can become a useful tool for analysis, numerical benchmarks, or as an algorithmic component.

5 Benchmarks

A natural stress test for symbolic Pauli arithmetic with the `PauliEngine` framework is the multiplication of two Hamiltonians,

$$H_1 = \sum_i (c_i, P_i), \quad (26)$$

$$H_2 = \sum_j (d_j, Q_j), \quad (27)$$

$$H_1 H_2 = \sum_i \sum_j (c_i d_j) (P_i Q_j), \quad (28)$$

which involves evaluating all pairwise products of Pauli strings and accumulating their coefficients. This task directly probes the efficiency of the underlying data structures, the Pauli-multiplication routine, and memory usage.

We benchmark the `PauliEngine` implementation against the de facto community standard in the form of `OpenFermion` [Mc20], and the recently developed `PauliArray` [DBB24], a highly optimized bit-parallel approach. For each benchmark instance, random Hamiltonians are generated by sampling Pauli strings of a fixed length with uniformly random local Paulis (including the identity). Two complementary benchmarks are considered:

- Scaling with Hamiltonian size (i.e., the number of Pauli terms), while keeping the Pauli string length fixed at 500.
- Scaling with Paul string length, while keeping the Hamiltonian size fixed at 500 terms.

All computations were performed on an Intel i9-11900KF CPU with 32 GB RAM. The results of these benchmarks are visualized in Figures 4 and 5.

Figure 4 shows that both `PauliEngine` and `PauliArray` outperform `OpenFermion` by one to two orders of magnitude for all tested system sizes with considerably better scaling behavior. Because `OpenFermion` becomes prohibitively slow as soon as the Hamiltonian has more than a few hundred terms, subsequent tests focus on a direct comparison between `PauliEngine` and `PauliArray`.

The trends in Fig. 5 reveal a clear division of performance regimes. For small Pauli strings (roughly up to 250 qubits) or small Hamiltonians, `PauliArray` achieves lower runtimes due

to its extremely compact SIMD-based representation. However, as either the Hamiltonian size or the Pauli string length increases, a sharp crossover occurs: `PauliArray`'s memory usage grows significantly, eventually saturating available RAM and triggering a steep runtime increase. In contrast, `PauliEngine` maintains smooth scaling in both parameters. Its memory-efficient symbolic dictionary and bitwise arithmetic prevent the rapid blow-up that affects `PauliArray` at large sizes.

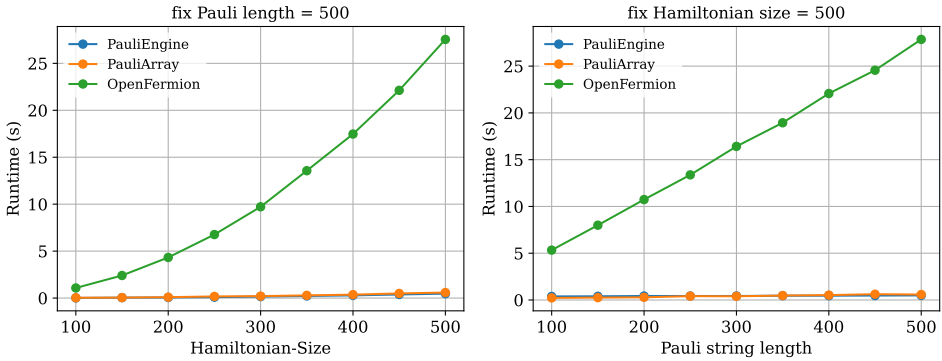


Fig. 4: Runtime comparison for Hamiltonian multiplication using `PauliEngine`, `PauliArray`, and `OpenFermion`. Left: Runtime vs. Hamiltonian size at fixed Pauli string length (500). Right: Runtime vs. Pauli string length at fixed Hamiltonian size (500). `PauliEngine` and `PauliArray` clearly outperform `OpenFermion` across all tested regimes; `OpenFermion` becomes a bottleneck already for moderate sizes. Performed on Intel i9-11900KF with 32GB RAM.

From a user perspective, it is comforting that applications relying on one of the three packages can easily switch between them. The API are deliberately designed to mimic the `QubitOperator` from `OpenFermion` to ensure this convenience. At the time, we would advise the usage of `PauliEngine` in two scenarios: 1. Large operators are needed. 2. Parametrized (and differentiable) operators are needed. In other scenarios, `PauliArray` and `OpenFermion` can offer more convenience as they are solely written in python.

6 Conclusion

This work introduces `PauliEngine`, a compact, high-performance C++ backend for symbolic Pauli string arithmetic. By combining a binary symplectic representation with optimized bitwise operations for multiplication, commutator evaluation, and phase tracking, `PauliEngine` provides fast and memory-efficient primitives for large-scale operator manipulation. The framework supports both numerical and symbolic coefficients and is accessible through a lightweight Python interface, making it suitable as a building block for quantum-software tools.

Across a wide range of benchmarks, `PauliEngine` consistently outperforms existing libraries such as `PennyLane`, `OpenFermion` that in our opinion define the state of the art, as well as

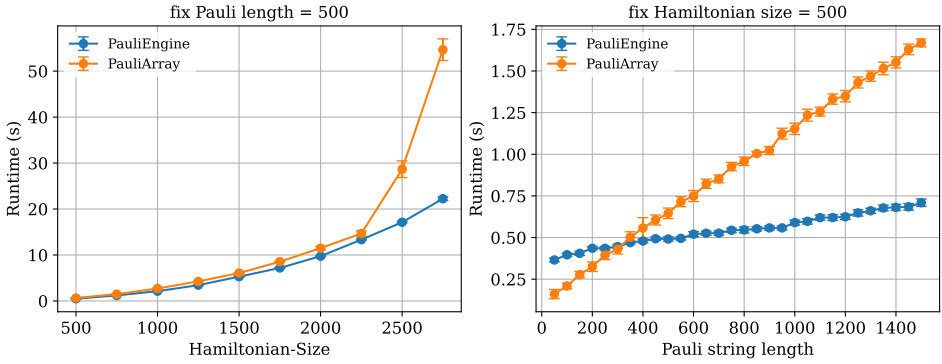


Fig. 5: Direct comparison between PauliEngine and PauliArray . Left: Mean runtime over 10 runs vs. Hamiltonian size at fixed Pauli string length (500). Right: Mean Runtime over 100 runs vs. Pauli string length at fixed Hamiltonian size (500). PauliArray is faster for small instances, but its runtime grows sharply once memory consumption becomes substantial, whereas PauliEngine maintains stable, with quadratic scaling in Hamiltonian size and near-linear scaling in Pauli string length. Performed on Intel i9-11900KF with 32GB RAM.

specialized libraries like PauliArray, with runtime gains of orders of magnitude and a clear scalability advantage for large Hamiltonians or long Pauli strings over all of them. As the PauliArray package was already benchmarked against qiskit [Ja24], we omitted it, in this article. We further demonstrated the utility of PauliEnding in practical applications, including accelerated computation of dynamical Lie algebras relevant for the analysis and simulation of variational circuits. We see the strongest potential within SDKs that support parametrized and differentiable structures, such as pennylane and tequila. In the latter, PauliEngine can be integrated almost seamlessly and we expect the same for the former.

Overall, PauliEngine offers an efficient and scalable foundation for Pauli-based quantum-simulation methods, enabling classical studies and analysis tools at system sizes that were previously challenging to handle.

Acknowledgment

This work has been funded by the Hightech Agenda Bayern (JSK), the Munich Quantum Valley via the MQV Doctoral Fellowship (AB), and the German Federal Ministry of Research, Technology and Space (BMFTR) via Quantum Technologies:HoliQC2 (LM,AB). The authors thank Oliver Hüttenhofer for various fruitful discussions, Craig Gidney for improving in-place multiplication to reach similar performance to stim [Gi21], and Ludmila Botelho for extensions to the code repository. The project uses NanoBind [Ja22] to bind C++ and Python.

References

- [AB25] Anand, A.; Brown, K. R.: Stabilizer configuration interaction: Finding molecular subspaces with error detection properties. *Physical Review A* 112 (3), p. 032421, 2025.
- [An22] Anand, A. et al.: A quantum computing view on unitary coupled cluster theory. *Chemical Society Reviews* 51 (5), pp. 1659–1684, 2022.
- [Be18] Bergholm, V. et al.: PennyLane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018, <https://github.com/PennyLaneAI/pennylane>.
- [BH21] Bharti, K.; Haug, T.: Iterative quantum-assisted eigensolver. *Physical Review A* 104 (5), p. L050401, 2021.
- [BK25] Bincoletto, D.; Kottmann, J.: A Physics-Informed Measurement Protocol for Expectation Values of Fermionic Observables. *Digital Discovery*, DOI:10.1039/D5DD00251F, pp. -, 2025, <http://dx.doi.org/10.1039/D5DD00251F>.
- [BPK23] Bauman, N. P.; Peng, B.; Kowalski, K.: Coupled-cluster downfolding techniques: A review of existing applications in classical and quantum computing for chemical systems. *Advances in Quantum Chemistry* 87, pp. 141–166, 2023.
- [Ce21] Cerezo, M. et al.: Variational quantum algorithms. *Nature Reviews Physics* 3 (9), pp. 625–644, 2021.
- [Ce25] Cerezo, M. et al.: Does provable absence of barren plateaus imply classical simulability? *Nature Communications* 16 (1), 2025, <http://dx.doi.org/10.1038/s41467-025-63099-6>.
- [Ci25] Cirq Developers: Cirq, version 1.6.1, 2025, <https://github.com/quantumlib/Cirq>.
- [DBB24] Dion, M.; Belabbas, T.; Bastien, N.: Efficiently manipulating Pauli strings with PauliArray, 2024, *arXiv: 2405.19287 [quant-ph]*, <https://arxiv.org/abs/2405.19287>.
- [De25] Developers, S.: SymEngine: Fast symbolic manipulation library, 2025, <https://github.com/symengine/symengine>, accessed: 09/19/2025.
- [Do24] Dobrutz, W. et al.: Toward real chemical accuracy on current quantum hardware through the transcorrelated method. *Journal of Chemical Theory and Computation* 20 (10), pp. 4146–4160, 2024.
- [El23] Elben, A. et al.: The randomized measurement toolbox. *Nature Reviews Physics* 5 (1), pp. 9–24, 2023.
- [Fo24] Fontana, E. et al.: Characterizing Barren Plateaus in Quantum Ansätze with the Adjoint Representation. *Nature Communications* 15 (1), p. 7171, 2024.
- [Gi21] Gidney, C.: Stim: a fast stabilizer circuit simulator. *Quantum* 5, p. 497, 2021, <https://doi.org/10.22331/q-2021-07-06-497>.
- [GK25] Gresch, A.; Kliesch, M.: Guaranteed efficient energy estimation of quantum many-body Hamiltonians using ShadowGrouping. *Nature communications* 16 (1), p. 689, 2025.
- [Go22] Gonthier, J. F. et al.: Measurements as a roadblock to near-term practical quantum advantage in chemistry: Resource analysis. *Physical Review Research* 4 (3), p. 033154, 2022.
- [Go25] Goh, M. L. et al.: Lie-algebraic classical simulations for quantum computing. *Physical Review Research* 7 (3), 2025, <http://dx.doi.org/10.1103/3y65-f5w6>.
- [HG25] Higgott, O.; Gidney, C.: Sparse Blossom: correcting a million errors per core second with minimum-weight matching. *Quantum* 9, p. 1600, 2025, <https://doi.org/10.22331/q-2025-01-20-1600>.

- [Hu20] Huggins, W. J. et al.: A non-orthogonal variational quantum eigensolver. *New Journal of Physics* 22 (7), p. 073009, 2020.
- [Ja22] Jakob, W.: nanobind: tiny and efficient C++/Python bindings, <https://github.com/wjakob/nanobind>, 2022.
- [Ja24] Javadi-Abhari, A. et al.: Quantum computing with Qiskit, 2024, arXiv: 2405.08810 [quant-ph].
- [JBB+19] Jones, T.; Brown, A.; Bush, I., et al.: QuEST and High Performance Simulation of Quantum Computers. *Scientific Reports* 9, p. 10736, 2019, <https://doi.org/10.1038/s41598-019-47174-9>.
- [JG20] Jones, T.; Gacon, J.: Efficient calculation of gradients in classical simulations of variational quantum algorithms. arXiv preprint arXiv:2009.02823, 2020.
- [KS24] Kottmann, J. S.; Scala, F.: Quantum Algorithmic Approach to Multiconfigurational Valence Bond Theory: Insights from Interpretable Circuit Design. *Journal of Chemical Theory and Computation* 20 (9), pp. 3514–3523, 2024.
- [Ku22] Kumar, A. et al.: Quantum simulation of molecular electronic states with a transcorrelated Hamiltonian: higher accuracy with fewer qubits. *Journal of chemical theory and computation* 18 (9), pp. 5312–5324, 2022.
- [La22] Larocca, M. et al.: Diagnosing Barren Plateaus with Tools from Quantum Optimal Control. *Quantum* 6, p. 824, 2022.
- [La24] Larocca, M. et al.: A review of barren plateaus in variational quantum computing. arXiv preprint arXiv:2405.00781, 2024.
- [LPS25] Loizeau, N.; Peacock, J. C.; Sels, D.: Quantum many-body simulations with PauliStrings.jl. *SciPost Phys. Codebases*, p. 54, 2025, <https://scipost.org/10.21468/SciPostPhysCodeb.54>.
- [LZ25] Li, Z.-L.; Zhang, S.-X.: The Dual Role of Low-Weight Pauli Propagation: A Flawed Simulator but a Powerful Initializer for Variational Quantum Algorithms, 2025, arXiv: 2508.06358, <https://arxiv.org/abs/2508.06358>.
- [Mc20] McClean, J. R. et al.: OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology* 5 (3), p. 034014, 2020.
- [Na23] Naldesi, P. et al.: Fermionic correlation functions from randomized measurements in programmable atomic quantum devices. *Physical Review Letters* 131 (6), p. 060601, 2023.
- [Pa25] Patel, S. et al.: Quantum Measurement for Quantum Chemistry on a Quantum Computer. *Chemical Reviews* 125 (16), pp. 7490–7524, 2025, <https://doi.org/10.1021acs.chemrev.5c00055>.
- [PE25] PE Devs.: PauliEngine, 2025, <https://github.com/tequilahub/pauliengine.git>.
- [Ra19] Rall, P. et al.: Simulation of qubit quantum circuits via Pauli propagation. *Physical Review A* 99 (6), 2019, <http://dx.doi.org/10.1103/PhysRevA.99.062337>.
- [Ra24] Ragone, M. et al.: A Lie Algebraic Theory of Barren Plateaus for Deep Parameterized Quantum Circuits. *Nature Communications* 15 (1), p. 7172, 2024.
- [Ru25] Rudolph, M. S. et al.: Pauli Propagation: A Computational Framework for Simulating Quantum Systems, 2025, <https://arxiv.org/abs/2505.21606>.
- [Ry20] Ryabinkin, I. G. et al.: Iterative Qubit Coupled Cluster Approach with Efficient Screening of Generators. *Journal of Chemical Theory and Computation* 16 (2), pp. 1055–1063, 2020, <https://doi.org/10.1021/acs.jctc.9b01084>.

- [Sc19] Schuld, M. et al.: Evaluating analytic gradients on quantum hardware. *Physical Review A* 99 (3), p. 032331, 2019.
- [SCL24] Sun, J.; Cheng, L.; Li, W.: Toward chemical accuracy with shallow quantum circuits: A Clifford-based Hamiltonian engineering approach. *Journal of Chemical Theory and Computation* 20 (2), pp. 695–707, 2024.
- [SCZ25] Sun, J.; Cheng, L.; Zhang, S.-X.: Stabilizer ground states for simulating quantum many-body physics: theory, algorithms, and applications. *Quantum* 9, p. 1782, 2025.
- [Sh23] Shang, Z.-X. et al.: Schrödinger-Heisenberg variational quantum algorithms. *Physical Review Letters* 131 (6), p. 060406, 2023.
- [SHE20] Stair, N. H.; Huang, R.; Evangelista, F. A.: A multireference quantum Krylov algorithm for strongly correlated electrons. *Journal of chemical theory and computation* 16 (4), pp. 2236–2245, 2020.
- [So23] Sokolov, I. O. et al.: Orders of magnitude increased accuracy for quantum many-body problems on quantum computers via an exact transcorrelated method. *Physical Review Research* 5 (2), p. 023174, 2023.
- [VYI20] Verteletskyi, V.; Yen, T.-C.; Izmaylov, A. F.: Measurement optimization in the variational quantum eigensolver using a minimum clique cover. *The Journal of chemical physics* 152 (12), 2020.
- [Xu25] Xu, X. et al.: A Herculean task: classical simulation of quantum computers. *Science Bulletin*, 2025, <http://dx.doi.org/10.1016/j.scib.2025.10.016>.
- [YGI23] Yen, T.-C.; Ganeshram, A.; Izmaylov, A. F.: Deterministic improvements of quantum measurements with grouping of compatible operators, non-local transformations, and covariance estimates. *npj Quantum Information* 9 (1), p. 14, 2023.
- [YVI20] Yen, T.-C.; Verteletskyi, V.; Izmaylov, A. F.: Measuring all compatible operators in one series of single-qubit measurements using unitary transformations. *Journal of chemical theory and computation* 16 (4), pp. 2400–2409, 2020.
- [Zh22] Zhang, Y. et al.: Variational quantum eigensolver with reduced circuit complexity. *npj Quantum Information* 8 (1), p. 96, 2022.