TENSORIZED PAULI COMPOSER

Hyunseong Kim
Department of Physics and Photon Science,
Gwangju Institute of Science and Technology
Gwangju
qwqwhsnote@gm.gist.ac.kr

ABSTRACT

In the paper, a new method was designed to construct a hamiltonian matrix which is given by a form of weighted summation of Pauli operators. A symplectic representation of Pauli terms allows us to express finite dimension Hilbert space as a matrix form. Therefore, Pauli basis representation of hamiltonian could be treated as one kind of symplectic representation. A transformation between two specific symplectic representations was designed. One is a common XZ code and the other is an index of Pauli basis matrix of TPD algorithm[1]. With inversed TPD algorithm, the transformation of the code yields a common matrix representation from weighted Pauli summation. The transformation between two codes consist of simple bitwise XOR operation. Consequently, the benefit of original benefit of TPD is preserved in iTPD. In addition to the algorithm, a term-chasing version was developed for sparse matrix in Pauli basis. It is roughly $O(16^n)$ time complexity but, in sparse case and < 5 qubit system, it showed 10 time faster than the prior algorithm. The algorithm has $O(n4^n)$ time complexity in the worst case, and comparing to the term-by-term construction method with $O(n16^n)$ complexity, it is computationally efficient for general case.

***K*eywords** Matrix composition · Pauli polynomial · Tensor product

1 Introduction

The research proposes a method to construct a specific Pauli basis matrix representation of weighted Pauli polynomial. The Pauli basis matrix is a representation of finite dimension Hilbert space in Heisenberg representation. In other word, the representation is faithful to every operator in Hilbert space. Moreover, the matrix is a result matrix of Tensorized Pauli Decomposition(`TPD`) algorithm which had been presented by Hantzko et al[1]. Since the decomposition process was sequential basis transformation, the inverse direction is also well-defined. Combining these two results yields the inversed TPD algorithm(`iTPD`), transforming a given weighted Pauli polynomial to its single matrix representation.

$$\sum_{i}^{n} \lambda_i P_i \rightarrow_{\texttt{iTPD}} H \tag{1}$$

Following sections demonstrate a brief review of TPD algorithm, especially index determination of single Pauli element. Consequently, a symplectic representation of Pauli element was introduced. Treating the representation as *p*-adic representation provides a connection between Pauli element, and matrix element. In the last section, the inverse TPD algorithm was compared with the other algorithms in complexity and real execution time in hardware.

2 Tensorized Pauli Decomposition Algorithm

In 2024, Hantzko et al proposed that general $\mathbf{M}_{2^n}(\mathbb{C})$ matrices could be efficiently decomposed into several Pauli terms with corresponding coefficients, using a tensorized block matrix manipulation[1].

$$\sum_{i=0}^{3} c_i\sigma_i = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \rightarrow_{TPD} \begin{bmatrix} c_0 & c_1 \\ c_2 & c_3 \end{bmatrix} \tag{2}$$

where,

$$\begin{aligned} A_{11} &= c_0 + c_3 \\ A_{12} &= c_1 - ic_2 \\ A_{21} &= c_1 + ic_2 \\ A_{22} &= c_0 - c_3. \end{aligned} \tag{3}$$

The basic idea is decoupling the coefficients of each tensor producted space, iteratively. See Figure 1. The decomposition process is non-linear in $2^n \times 2^n$ matrix space. How-

ever, it is a basis transformation in a higher dimension $\mathbb{C}^N$ space, which is isomorphic to the vector spaces with $N = 4^n$ dimension. For example, the 2×2 dimension matrix of a 1-qubit system, the process could be expressed as $\mathbf{v} \in \mathbb{C}^4$.

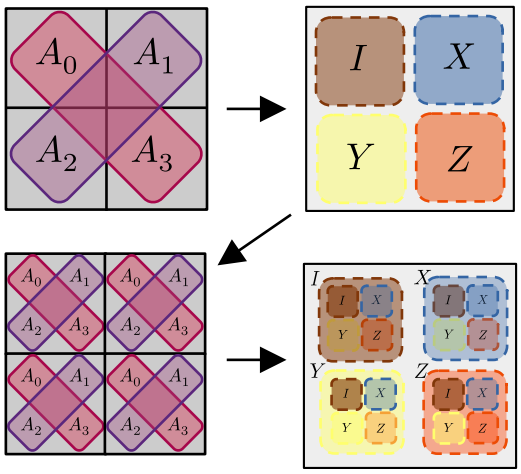


Figure 1: Iterative diagram of Tensorized Pauli Decomposition algorithm.

$$\frac{1}{2}\begin{bmatrix}1 & 0 & 0 & 1\\ 0 & 1 & 1 & 0\\ 0 & i & -i & 0\\ 1 & 0 & 0 & -1\end{bmatrix}_{TPD}\begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix} = \begin{bmatrix}c_0\\ c_1\\ c_2\\ c_3\end{bmatrix} \tag{4}$$

In 2×2 matrix with computational basis, the next matrix is a *Pauli basis matrix* of 1 qubit system.

$$\begin{bmatrix}c_0 & c_1\\ c_2 & c_3\end{bmatrix} \tag{5}$$

In the vectorized representation, the intermediate step of TPD algorithm could be represented with the next notation.

$$\begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}_1 \otimes \begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}_2 \otimes \cdots \otimes \begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}_{n-1} \otimes \begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}$$
$$\Downarrow_{i\text{steps},i>2} \tag{6}$$
$$\begin{bmatrix}c_0\\ c_1\\ c_2\\ c_3\end{bmatrix}_1 \otimes \begin{bmatrix}c_0\\ c_1\\ c_2\\ c_3\end{bmatrix}_2 \otimes \cdots \otimes \begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}_{n-1} \otimes \begin{bmatrix}A_{11}\\ A_{12}\\ A_{21}\\ A_{22}\end{bmatrix}_n$$

For the n-fold matrix, the researcher can choose the basis transformation, freely. The transformation even permits the different basis in each product and each sub-matrix operations. In original implementation, each Pauli term was chased during the process. Therefore, they mapped the coefficients by adding a character to each string variables in each step of the iteration. The result Pauli basis matrix has identical coefficients without considering indexing. By choosing appropriate basis transformation, the decomposition yields the Latin matrix whose element indexes are XZ symplectic representation of Pauli terms in Reggio et al[2]. In below sections, the index of the Pauli basis matrix as *ij-index*.

3 Symplectic representation of Pauli element

A generalized Pauli matrix is defined with a sequential tensor product of 2×2 Pauli matrices.

$$P = (i)^m \otimes_j^n \sigma_j \tag{7}$$

From Pauli matrices, $\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, replacing σ_2' into σ_2, where $\sigma_2 = i\sigma_2'$, and separating phase to the outside yield a tensor product representation of n-fold Pauli element has a next form.

$$P_g = \{\sigma_0, \sigma_1, \sigma_2', \sigma_3\} \tag{8}$$

$$P = (i)^m \otimes_j^n \sigma_j \tag{9}$$

where, m is a number of occurrence of σ_2' in the product. Since $\sigma_2' = \sigma_1\sigma_3$ holds true, we can decompose the given n-fold Pauli term as next two parts; elements of families, X, Z. The example families are z-family and x-family referred by Reggio et al[2].

$$\otimes_j^n \sigma_j = \left(\otimes_{j\in\{0,1\}}^n \sigma_j\right)\left(\otimes_{k\in\{0,3\}}^n \sigma_k\right) \tag{10}$$

Eq(10) yields a unique binary vector tuple representation of length 2, replacing $I \to 0, X, Z \to= 1$ in each meneber.

$$P = (\vec{z}, \vec{x}),\ \vec{z}, \vec{x} \in \{0,1\}^N \tag{11}$$

where $\vec{z}, \vec{x}$ are binary vector representation of $\otimes_{j\in\{0,1\}}^n \sigma_j, \otimes_{j\in\{0,3\}}^n \sigma_j$ indicates $I = 0, X = 1, Z = 1$.

A 2-adic representation of binary vector permit them to be treated with a single integer tuple, (n_z, n_x), ignoring phase factor.

$$\ldots 110 \leftrightarrow \cdots + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \tag{12}$$
$$P \leftrightarrow (n_z, n_x) \tag{13}$$

For example, $(6, 5)$ of 3-qubit system is a symplectic representation of YXI.

$$\begin{aligned} 6 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = X \otimes X \otimes I \\ 5 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = Z \otimes I \otimes Z \end{aligned} \tag{14}$$

Comprehensive example is, $IXXZYY$, Pauli element could be transformed to integer tuple.

$$IXXZYY = (-i)^f(IIIZZZ) \cdot (IXXIXX) \quad (15)$$
$$\rightarrow (-i)^f([0,0,0,1,1,1],[0,1,1,0,1,1]) \quad (16)$$
$$\rightarrow (f, 7, 27) \quad (17)$$

This is a symplectic tuple representation of Pauli element. If the phase term, f, was ignored, every Pauli elements are corresponding to each element of $2^n \times 2^n$ dimension matrix.

$$H = \lambda_0 IXX + \lambda_1 YXZ + \lambda_2 ZIX \quad (18)$$
$$H = [H]_{n_z,n_x} \quad (19)$$
$$= \begin{bmatrix} 0 & 0 & 0 & \lambda_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \lambda_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (20)$$

A binary vector representation is now commonly adopted in quantum computing frameworks such as IBM Qiskit and Pennylane written by Xanadu, because the representation has a significant benefit in execution time. IBM implemented the above symplectic vector representation for their *Pauli* class in python library, Qiskit, to use the above binary implementation[3]. However, in the paper, the order of the symplectic representation is reversed order of the IBM implementation. There is no difference in algebra implementation and commutation conversion routine, however, the conversion to index of coefficient matrix is more direct in the reversed order than the IBM order.

3.1 TPD for symplectic representation

Eq (20) seems a one candidate of Pauli basis matrix for `iTPD`. Unfortunately, it is not, but still it is a good starting point to combine with `TPD` algorithm, `TPD` could be modified to generate a Pauli basis matrix whose index is a symplectic tuple of Pauli elements whose element value is a weight of the term. See modified version code in [**?**][1].

Unfortunately, the modified-`TPD` has less practical than original `TPD`. During the calculation, in k-th step, additional $2^k \times 2^k$ size memory is required to swap the partition of the matrix, and it yields a great inefficient in spatial and time complexity. Therefore, a transformation is required preserving a computational benefit of TPD algorithm The transformation converts a symplectic tuple representation to Pauli basis.

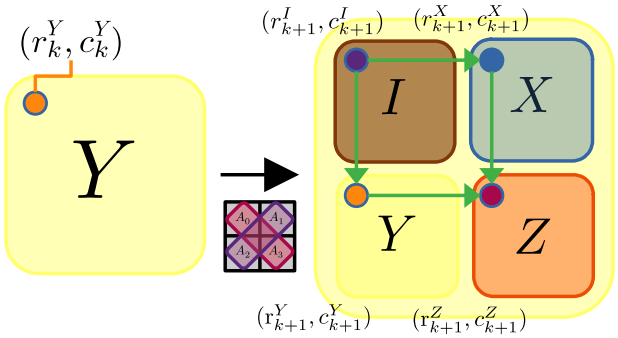


Figure 2: Index diffusion diagram by TPD decomposition process

3.2 Symplectic tuple to Pauli basis matrix

Theorem 1. *For a given symplectic representation, (n_x, n_z) of the given Paili term, P, their index, (i, j), in Pauli basis matrix is determined by next formula*

$$(i, j) = (n_z, n_x \hat{} n_z) \quad (21)$$

where, $\hat{}$ is a XOR bitwise operator.

Proof. From k-th iteration of the TPD algorithm of $2^n \times 2^n$ dim square matrix, the unit sub-matrix dimension is 2^{n-k} and there are 4 block matrices, see Figure 1. With (n_z, n_x) symplectic tuple representation, each k-th binary values of 2-adic representation of n_z, n_x determine a $k+1$ index movement in Fig (2).

$$\begin{bmatrix} I & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} 0_z \cdot 0_x & 0_z \cdot 1_x \\ 1_z \cdot 1_x & 1_z \cdot 0_x \end{bmatrix} \quad (22)$$

where, $0_z, 1_z, 0_x, 1_x$ are k-th binary value of (n_z, n_x) of Pauli element. Let them as nz_k, nx_k.

For row index, $nz_k \in \{0, 1\}$ the Z-binary determine the row index movement, if $nz_k = 1$, the row location is changed by $+2^{n-k}$ else it is not. For column index, the column index changed by $+0$ if $(1_x, 0_z)$ or $(0_x, 1_z)$, else $+2^{n-k}$ if $(0_x, 0_z)$ or $(1_x, 1_z)$. It is a simple XOR binary operator, thereby $2^{n-k} * nz_k \hat{} nx_k$, $nz_k \in \{0_z, 1_z\}, nx_k \in \{0_x, 1_x\}$

Thus, we have (i, j) coefficient index of XZ representation by iteration from 1-th to n-th. In 2-adic representation of integer, they are identity and bitwise XOR operator of (nz, nx).

$$\begin{array}{rcl} i = & \sum_{k=0}^{n-1} 2^k nz_k & = nz \\ j = & \sum_{k=0}^{n-1} 2^k nz_k \hat{} nx_k & = nz \hat{} nx \end{array} \quad (23)$$

□

Using Eq (21), weighted Pauli polynomial could be transformed to a Pauli basis matrix directly applied for `iTPD`.

[1] In *tp* directory.

Theorem 1 provides 1-1 correspondence between Pauli basis matrix element location and each Pauli element. The symplectic tuple representation is restored by next relationship.

$$(nz, nx) = (i, i^{\wedge} j) \tag{24}$$

4 Minor improvement in TPD

In original TPD, Pauli elements were chased in each iteration, by constructing each k-th Pauli characters of all elements. However, with Eq (21), the weights in Pauli basis matrix are directly determine corresponding Pauli elements. Therefore, the character storing routine could be eliminated from the original TPD algorithm.

5 iTPD algorithm

5.1 Naive version

The previous section showed that TPD algorithm is a sequential applying of unitary transformation for each vectors in a product representation. By the property of unitary, the inverse transformation is well-defined in 4^n dimension, and `iTPD` is defined with a $2^n \times 2^n$ space representation with sub-block matrix additions. The $\{c_i\}_{i=0}^{3}$ coefficients in Eq (3) is restored as

$$\begin{aligned} A_{11} &= c_0 + c_3 \\ A_{12} &= c_1 + ic_2 \\ A_{21} &= c_1 - ic_2 \\ A_{22} &= c_0 - c_3 \end{aligned} \tag{25}$$

.

The total process is achieved by iteratively applying the Eq (25) in reverse order of `TPD` algorithm. The inverse process is written in Algorithm 2. With symplectic code conversion(`SCC`) algorithm 1, it yields *Tensorized Pauli composer algorithm*(`TPC`).

Algorithm 1 Symplectic code conversion

Require: $N \leftarrow$ qubit number, $P \leftarrow \sum_i \lambda_i P_i$ (weighted Pauli summation).
  $M \leftarrow$ Zero matrix of dim (N, N)
  for $(\lambda_i, n_z^i, n_x^i)$ **in** P **do**
    $row \leftarrow n_z^i$
    $col \leftarrow n_z^{i\wedge} n_x^i$
    $M[row, col] \leftarrow \lambda_i$
  end forreturn M

The algorithm is a combination of `iTPD` and the above symplectic construction method. If Pauli elements are stored in string type, the conversion requires some computational resources, but if a framework store them as symplectic code, it is very efficient to convert the representation. This approach is already adopted in some frameworks[4][5]. In addition, as has mentioned in [1], it does not need further instant matrix to save the terms, so that spatial complexity of the algorithm is also practical in large system.

Algorithm 2 Naive `iTPD`

Require: $M \leftarrow$ Pauli basis matrix of $(2^n, 2^n)$
  matdim $\leftarrow 2^n$
  steps $\leftarrow$ n
  unit_size $\leftarrow 1$
  for step **in** steps **do**
    step1 $\leftarrow$ step+1
    mat_size $\leftarrow 2 *$ unit_size
    indexes $\leftarrow$ [matdim/2^{step1}]
    indexes_ij $\leftarrow$ mat_size*indexes
    for i **in** indexes_ij **do**
      for i **in** indexes_ij **do**
        $r_{1s} \leftarrow$ i
        $r_{1f2s} \leftarrow r_{1s}$ + unit_size
        $c_{1s} \leftarrow$ j
        $c_{1f2s} \leftarrow c_{1s}$ + unit_size
        $r_{2f} \leftarrow r_{1f2s}$ + unit_size
        $c_{2f} \leftarrow c_{1f2s}$ + unit_size
        coef $\leftarrow 1$
        $M[r_{1s}{:}\ r_{1f2s}, c_{1s}{:}c_{1f2s}]$ += coef*$M[r_{1f2s}{:}\ r_{2f}, c_{1f2s}{:}c_{2f}]$
        $M[r_{1f2s}{:}\ r_{2f}, c_{1f2s}{:}c_{2f}] = M[r_{1s}{:}\ r_{1f2s}, c_{1s}{:}c_{1f2s}]$ -2*coef *$M[r_{1f2s}{:}\ r_{2f}, c_{1f2s}{:}c_{2f}]$
        coef $\leftarrow -\sqrt{-1}$
        $M[r_{1f2s}{:}\ r_{2f}, c_{1s}{:}c_{1f2s}]$ += coef*$M[r_{1s}{:}\ r_{1f2s}, c_{1f2s}{:}c_{2f}]$
        $M[r_{1s}{:}\ r_{1f2s}, c_{1f2s}{:}c_{2f}] = M[r_{1f2s}{:}\ r_{2f}, c_{1s}{:}c_{1f2s}]$ -2*coef *$M[r_{1s}{:}\ r_{1f2s}, c_{1f2s}{:}c_{2f}]$
      end for
    end for
    unit_size $\leftarrow$ 2*unit_size
  end forreturn M

Algorithm 3 TPC

Require: $N \leftarrow$ qubit number, $P \leftarrow \sum_i \lambda_i P_i$
  $M \leftarrow$ `SCC`(N, P)
  $H \leftarrow$ `iTPD`(M)**return** H

5.2 Effective term chasing

During the calculation, if two partial matrices were zero matrix, then the calculation is meaningless. This case arises in sparse Pauli basis matrix; Hamiltonian has few Pauli terms. In sparse case, chasing non-zero term provides some benefit in calculation time.

If we know an index set of Pauli terms, where their coefficients are not zero, we could avoid the operation for zero sub matrices terms in the intermediate steps of the composition. The non-zero terms are denoted with *effective* terms. Considering I-Z and X-Y calculation, when k number of Pauli terms were given, there is a k_{eff} number of effective terms where

$$k = k_{eff} + d,\ d \geq 0 \tag{26}$$

, d is a number of the duplicated terms.

From the i-th effective index set, the effective index set for the next step is calculated by quotient of 2, such as

$$\begin{array}{rcl} row_{i+1} & = & r_i \\ col_{i+1} & = & c_i \end{array} \tag{27}$$

where, $row_i = 2 * m_i + r_i$ and $col_i = 2 * n_i + c_i$. The k_i number is same with $(k_{eff})_{i-1}$ number.

For example, in $2^4 \times 2^4$ Pauli basis matrix, if we have (1, 14), (2, 13), (3, 1), (6, 4), (7, 4), (7, 5), (13, 9), (14, 10) elements were non-empty Pauli terms. We can chase the non-empty unit indexes with Eq (27)

$$\begin{array}{cccccc} & (1,14) & (0,7) & (0,3) & (0,1) & (0,0) \\ & (2,13) & (1,6) & (0,0) & (0,0) & \\ & (3,1) & (1,0) & (1,1) & (1,1) & \\ & (6,4) & (3,2) & (3,2) & - & \\ & (7,4) & (6,4) & - & - & \\ & (7,5) & (7,5) & - & - & \\ & (13,9) & - & - & - & \\ & (14,10) & - & - & - & \\ k & 8 & 6 & 4 & 3 & 1 \\ k_{eff} & 6 & 4 & 3 & 2 & 1 \end{array} \tag{28}$$

.

See 4 for further details.

6 Benchmarks

6.1 Complexity analysis

6.1.1 Term-by-term methods

The general Pauli-composition methods focus on term-by-term matrix implementation. That is, with the given k-term Pauli polynomial, the methods generate k matrices corresponding to each tern and sum the matrices.

For $2^n \times 2^n$ matrices, if we denote the complexity of an algorithm for constructing a single Pauli matrix, $f(n)$, then the total composition complexity is estimated as,

$$k * f(n) + (k-1)4^n \tag{29}$$

where, the 4^n term represents element wise addition complexity. Since k ranges from 1 to 4^n, the maximum complexity is

$$16^n + 4^n(f(n) - 1) \tag{30}$$

Therefore, the term-by-term algorithms are fast with 16^n time-complexity in worst case. Spatial complexity of term-by-term methods is $2 \cdot 4^n$ by preparing zero matrix and iteratively adding each terms to the zero matrix.

Algorithm 4 Effective term chasing `iTPD`

Require: poly = $\{((i,j))\}_{l=1}^{k}$ ▷ ij converted Pauli terms
Require: $M \leftarrow$ Pauli basis matrix of $(2^n, 2^n)$
  matdim $\leftarrow 2^n$
  steps $\leftarrow$ n
  unit_size $\leftarrow 1$
  for step **in** steps **do**
    pstep $\leftarrow$ [] ▷ Empty list
    dup $\leftarrow$ []
    for (i, j) **in** poly **do**
      if (i, j) **in** dup **then** continue
      end if
      n, o $\leftarrow$ i%2, j%2 ▷ IZ, XY determination
      l, m $\leftarrow$ (i+1-2*(n), j+ 1-(2*(o))) ▷ Get a corresponding location
      dup.insert((l,m) , (i,j))
      if n == 1 **then**
        pair $\leftarrow$ ((l, m), (i, j))
      else
        pair $\leftarrow$ ((i, j), (l, m))
      end if
      if (i+j)%2 == 1 **then**
        coef $\leftarrow -\sqrt{-1}$
      else
        coef $\leftarrow 1$
      end if
      $r_{1s} \leftarrow$ unit_size * pair[0][0]
      $r_{1f} \leftarrow r_{1s}$ + unit_size
      $c_{1s} \leftarrow$ unit_size * pair[0][1]
      $c_{1f} \leftarrow c_{1s}$ + unit_size
      $r_{2s} \leftarrow$ unit_size * pair[1][0]
      $r_{2f} \leftarrow r_{2s}$ + unit_size
      $c_{2s} \leftarrow$ unit_size * pair[1][1]
      $c_{2f} \leftarrow c_{2s}$ + unit_size
      $M[r_{1s}:\ r_{1f},\ c_{1s}{:}c_{1f}]$ += coef*$M[r_{2s}:\ r_{2f},\ c_{2s}{:}c_{2f}]$
      $M[r_{2s}:\ r_{2f},\ c_{2s}{:}c_{2f}] = M[r_{1s}:\ r_{1f},\ c_{1s}{:}c_{1f}]$ - 2*coef *$M[r_{2s}:\ r_{2f},\ c_{2s}{:}c_{2f}]$
      i »=1 ▷ Bit shift operation
      j »=1
      if (i, j) **in** pstep **then** continue
      else: pstep.insert((i,j))
      end if
    end for
    poly $\leftarrow$ pstep
    unit_size $\leftarrow$ 2*unit_size
  end forreturn M

6.1.2 Algorithm complexity

In the naive algorithm 2, the time complexity of each step is 4^n, and there are n number of steps. Therefore, the total time complexity is $O(n4^n)$

For the effective term algorithm, it is very complicated for estimating time-complexity. As mentioned in TPD algorithm, tensorized method can choose basis transformation separately[1] and effective term method determine meaningful transformation. Therefore, the complexity strongly depend on problem specification. However, by taking the worst case of effective term, we can estimate its upper bound.

With initial $k = k_{eff} + d$ number non zero-terms, assuming that the worst case that $d = 0$, and we naively calculate all tensorized transformation, we have

$$k_{eff,0} \in [\frac{1}{2}4^n], k_{eff,i} = \frac{1}{2}k_{eff,i-1} \tag{31}$$

with duplication search step of $O(k_{eff,i})$ complexity, the total time complexity consists of

$$2nk_{eff}^2 + 34(2^{n+1} - 1)k_{eff} \tag{32}$$

The maximum complexity is not different in the naive version, but it is still lower than any term-by-term algorithm. About the effective term algorithm, at some range of k_{eff} value it is the most efficient but at very small or worst case, the naive version is more efficient. The worst complexity of the effective algorithm arises when $k = \frac{1}{2}4^n$ so that,

$$\frac{n}{2}16^n + 17(2 * 8^n - 4^n) \tag{33}$$

.

From Eq (32), the benefit region to use the effective term algorithm is

$$k < \frac{1}{2n}\left(\sqrt{289(2^{n+1})^2 + n8^n} - 17(2^{n+1} - 1)\right) \tag{34}$$

The efficient is achieved when the non-zero terms are under 0.5% of 4^n number of terms.

6.2 Benchmarks with the other frameworks

The belows are brief reviews of current quantum frameworks. List of the fraameworks and methods are provides Pauli composition routine.

- PauliComposer[6].
- Qiskit `Pauli`, to_matrix method [3].
- Pennylane, `Pauli` to matrix [7].
- Cirq: unitary matrix transform [8].

Pennylane and Cirq's matrix conversion are just a simple kronecker product of each matrix terms. In the recent version of Pennylane, they provide *PauliSetence* class and matrix routine. However, the current implementation is not stable for test the general matrix composition.

Meanwhile, Qiskit routine is based on X, Z simplectic representation of Pauli term and they implemented the composition routine with PauliComposer method which uses row wise mapping. They provides *PauliList* class as like in Pennylane. The *to_matrix* routine generates rank-3 matrices for the given Pauli terms. However, the class does not support coefficient supports. In Qiskit version > 1.0 all backend routines were replaced with Rust from numpy based in <=0.43 version. To compare the algorithm complexity precisely, in the benchmark, 0.43 version was used. The composition needs alternative summation with coefficient multiplication.

The main conversion were conducted with 4 implementations the inverse tensorized algorithm with using efficient term chasing routine or not, or pure python-numpy routine and numba acceleration. Therefore, the comparsion with naive tensor product method and PauliComposer method is enough to see Pennylane and Qiskit frameworks. About the comparsion, each routine are prepared as their intended data. For example, Pennylane prepares the single Pauli term as `Pauli` object class , and Qiskit requires them to be in the symplectic representation.

The estimation was conducted for n-qubit system random matrices from $n = 1$ to $n = 9$ with 20%, 40%, 60%, 80%, 100% terms of 4^n degree polynomial. In the previous section we already showed that the effective term algorithm is efficient when only 0.5% terms are non-zero in the space, however, for the practical application, we started from 20% non-zero terms. The system specification and libraries are denoted on Table 2

6.3 Results

In Fig (3), we tested the Pauli-composition methods in various quantum frameworks. There was no method considering Pauli-polynomials for matrix conversion, so that all the methods are term-by-term methods. PauliComposer has a $f(n) = 2^n$ time complexity, therefore, the total composition complexity is 8^n and standard tensor product method has $f(n) = 4^n$ time complexity, so that the total complexity is $2 \cdot 8^n - 4^n$.

The naive algorithm showed most efficient time costs for higher $n > 6$ system for all cases. It took 10^1 or 10^3 times faster than term-by-term methods. Even in the smaller system $n < 5$, the time costs were compatible with the term-by-term methods. The effective term algorithm showed better time costs in $n < 5$ and the non-zero terms accounted for the matrix below 60% percentage of the whole system. however, comparing to the naive algorithm there was no significant time benefit for the term chasing, it can be adopted to further applications but in the current stage, the improvement was not noticiable. Moreover, in

Table 1: Summary of complexity of the algorithms with big-O notation.

-	Case	Time complexity	Spatial complexity
Term by term	Common	$O(k(f(n)+4^n))$	$O(4^n)$
	Worst	$O(16^n)$	
Naive	Common	$O(n4^n)$	$O(4^n)$
	Worst		
Effective term	Common	$O(nk_{eff}^2)$	
	Worst	$O(n16^n)$	

Table 2: System specification for simulation.

Processor	AMD Ryzen 5 1600, Six-Core Processor, 3.20 GHz
RAM	32.0 GB
OS	Windows 10 Home, 64bit, 22H2
Python	3.11.8
Numpy[9]	1.26.4
Scipy[10]	1.13.0
Qiskit[3]	0.43
Pennylane[7]	0.35.1
PauliComposer[6]	Original paper version.

the higher dimension system it overwhelms the term-by-term methods.

7 Conclusion

In the paper, tensorized Pauli composition(TPC) algorithm was designed using a proper Pauli basis matrix and inverse Tensorizerd Pauli decomposition(TPD) algorithm[1]. TPD was a sequential basis transformation of the given matrix. The common XZ symplectic representation is one type of the transformation. A simple conversion between the symplectic representation and an index of the Pauli basis matrix of TPD, was investigated and the inverse composition algorithms were designed. Furthermore, a modified algorithm of TPC was designed by adding an effective term chasing routine. The chasing routine eliminates unnecessary terms in the naive TPC process, so for sparse Pauli summation case, the algorithm is expected to show better result than naive version.

The algorithms are designed to compose the multiple terms simultaneously, thereby achieving better computational complexity in Pauli polynomial composition, in time and spatial both domains. The naive composition algorithm comprises a basis transformation mapping between the Pauli basis matrix and the original matrix representation.

Comparing to the previous term-by-term methods which have $O(16^n + 4^n(f(n)-1))$ complexity in the worst case, the naive algorithm is at least, twice faster than the common term-by-term methods with $O(n4^n)$ complexity. It means that we can construct the matrix with computational basis corresponding to the given Pauli-polynomial at process of the algorithm. The inverse algorithm could chase effective terms during the composition process. However, the chasing routine incurs significant computational costs and is not as efficient as the naive version, and even comparable with the term-by-term methods with $O(n16^n)$ complexity in the worst case.

In addition, the composition speed benchmark between the current quantum computing frameworks, Qiskit, Pennylane, and naive tensor product routines. The inverse composition algorithm showed better speed for all cases, single, multi, worst terms for from $n = 2$ to $n = 9$ qubit cases. The naive algorithm was 10 or 1000 times faster than term-by-term methods. Practically, even though the k is small, the naive version is comparable with the term-by-term methods. We assume that it is caused from the spatial complexity effect. The effective term algorithm could chase the effective terms, however in the current stage the time-cost benefit was not noticible in the implementation.

Acknowledgments

The research was funded by the Quantum Sapiens Human Resources Center.

Data and code available

The research was conducted for implementing a submodule of OptTrot python package for fast manipulation and optimization routine for Hamiltonian. OptTrot is a quantum computing frameworks for optimizing Trotter circuit on gate model computer. The benchmarked code and data are on Github repository[11].

References

[1] Lukas Hantzko, Lennart Binkowski, and Sabhyata Gupta. Tensorized pauli decomposition algorithm. *Physica Scripta*, 99(8):085128, jul 2024.

[2] Ben Reggio, Nouman Butt, Andrew Lytle, and Patrick Draper. Fast Partitioning of Pauli Strings into Commuting Families for Optimal Expectation Value Measurements of Dense Operators, June 2023. arXiv:2305.11847 [hep-lat, physics:hep-ph, physics:quant-ph].

[3] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[4] Maxime Dion, Tania Belabbas, and Nolan Bastien. Efficiently manipulating pauli strings with pauliarray, 2024.

[5] Hyunseong Kim. OptTrot: Optimization framework of Trotter circuit., October 2024.

[6] Sebastián Vidal Romero and Juan Santos-Suárez. PauliComposer: compute tensor products of Pauli matrices efficiently. *Quantum Information Processing*, 22(12):449, December 2023.

[7] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B Akash-Narayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[8] Cirq Developers. Cirq, December 2023.

[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

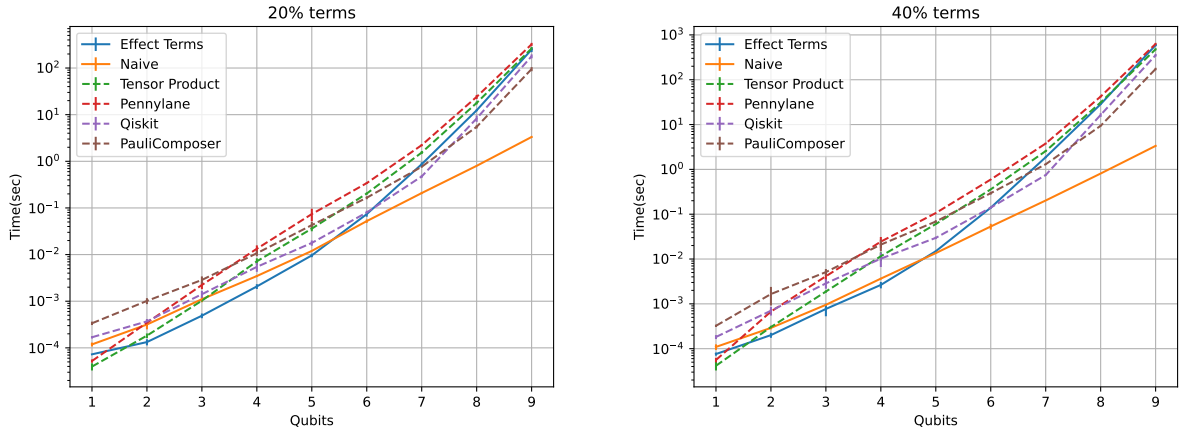
[10] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[11] Hyunseong Kim. Tensorized Pauli composer, October 2024.

A Computational aspects

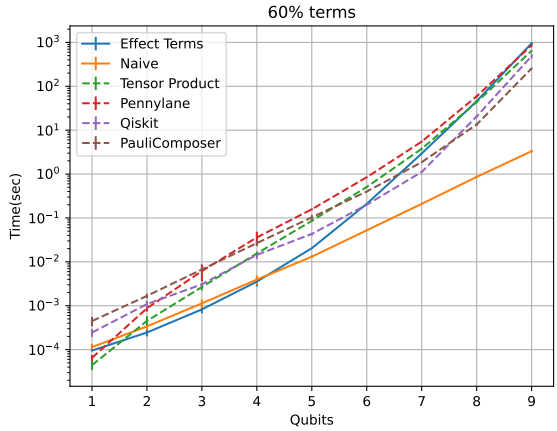
In real implementation, the chasing the efficient calculation term during the algorithm requires huge time complexity. The current implementation use bitwise operation, since, Python is not good for manipulate binary data, efficiently[2]. The above routines would be more appropriate for C/C++ or Rust like language implementation. We could observe that the binary compiled routines did not show difference whether the algorithm has a effective term chasing routine or not. In python or the other interpreter language, it is wise to use the naive algorithm. , and if the language environment naturally manipulate the bits, the effective term chasing version would be more appropriate.

[2]Bitwise operators are even slower than string manipulation in Python.

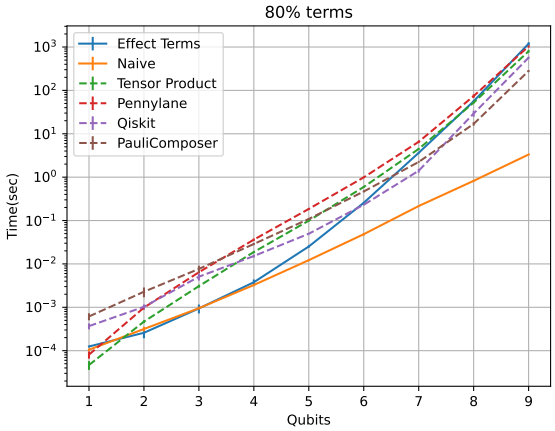


(a)

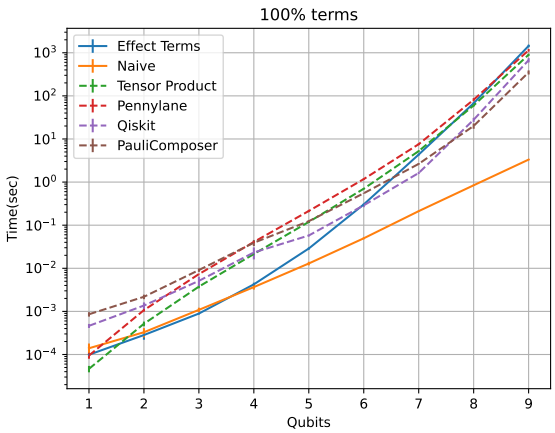
(b)



(c)



(d)



(e)

Figure 3: Benchmarks for matrix composition of Puali polynomials with the algorithm 2, 4 with Qiskit, Pennylane, PauliComposer, and standard tensor product methods, for $n = 1$ to $n = 9$. The percentages of the each case represents how many coefficients are non-empty in 4^n number of spaces.