

Picasso: Memory-Efficient Graph Coloring Using Palettes With Applications in Quantum Computing

S M Ferdous*, Reece Neff[†]*, Bo Peng*, Salman Shuvo*, Marco Minutoli*, Sayak Mukherjee*, Karol Kowalski*, Michela Becchi*[†], Mahantesh Halappanavar*

*Pacific Northwest National Laboratory, Richland, WA [†]North Carolina State University, Raleigh, NC
*{FirstName.LastName}@pnl.gov [†]{rwneff, mbecchi}@ncsu.edu

Abstract—A coloring of a graph is an assignment of colors to vertices such that no two neighboring vertices have the same color. The need for memory-efficient coloring algorithms is motivated by their application in computing clique partitions of graphs arising in quantum computations where the objective is to map a large set of Pauli strings into a compact set of unitaries. We present Picasso, a randomized memory-efficient iterative parallel graph coloring algorithm with theoretical sublinear space guarantees under practical assumptions. The parameters of our algorithm provide a trade-off between coloring quality and resource consumption. To assist the user, we also propose a machine learning model to predict the coloring algorithm’s parameters considering these trade-offs. We provide a sequential and a parallel implementation of the proposed algorithm.

We perform an experimental evaluation on a 64-core AMD CPU equipped with 512 GB of memory and an Nvidia A100 GPU with 40GB of memory. For a small dataset where existing coloring algorithms can be executed within the 512 GB memory budget, we show up to 68× memory savings. On massive datasets we demonstrate that GPU-accelerated Picasso can process inputs with 49.5× more Pauli strings (vertex set in our graph) and 2,478× more edges than state-of-the-art parallel approaches.

Index Terms—Graph coloring, quantum computing, memory-efficient algorithms.

I. INTRODUCTION

Given a graph $G(V, E)$, the problem of graph coloring is to assign a color to each vertex such that no two adjacent vertices are assigned the same color, while minimizing the number of colors used. Graph coloring is one of the central problems in combinatorial optimization with applications in various scientific domains [1]. Many algorithmic techniques have been developed to solve coloring in sequential, parallel, and distributed settings for many variants of the coloring problem [2]. All of these algorithms are memory-demanding since they require loading the entire graph and several auxiliary data structures into the memory. For massive graphs, especially on limited-memory accelerators (GPUs), these algorithms easily exhaust the available memory, and the problem is exacerbated for dense graphs due to quadratic scaling of edges. Our work considers graphs that are $\approx 50\%$ dense ($|E| \approx |V|^2/2$) for which the current graph coloring approaches [3]–[6] quickly run out of memory when processing large instances (§ VII).

Our quest for memory-optimized coloring techniques stems from an application of quantum algorithms to computational chemistry. Quantum computers have the potential to offer new insights into chemical phenomena that are not feasible with

classical computers. To utilize quantum approaches effectively, it is indispensable to encode chemical Hamiltonians and chemical-inspired wave function ansätze (i.e., the assumptions of the wave function form) in a quantum-compatible representation, while adhering to the proposed theoretical framework. However, the direct encoding of chemical Hamiltonians and typical chemical-inspired ansätze, which usually grows as high-degree polynomials, is unscalable. This directly affects the efficiency and applicability of the corresponding quantum algorithms. For example, transforming the chemical Hamiltonian from the second quantization to spin operators yields *Pauli strings* (tensor products of 2×2 Pauli and identity matrices) that scale as $\mathcal{O}(N^4)$, where N is the number of basis functions spanning the Hamiltonian [7]–[9].

One near-term solution to improve the scaling of quantum algorithms is the *unitary partitioning* and its variants, which aim to find compact representations of a linear combination of Pauli strings (§ II). These strategies, predominantly based on the graph analysis of Pauli strings, typically yield a reduction ranging from 1/10 to 1/6 in the number of alternative unitaries for small test cases. The unitary partitioning problem reduces to a graph coloring problem, where the vertices of the graph represent the Pauli strings, and the edges represent whether these strings obey an anticommute relation (§ II). These graphs are large and dense. Therefore, the state-of-the-art approaches for the unitary partitioning problem can only solve small molecules with a few thousand Pauli strings, whereas the desired scale is on the order of $\mathcal{O}(10^6 \sim 10^{12})$ of Pauli strings.

Building on recent developments in sublinear algorithms for graph coloring [10], we introduce a novel parameterized coloring algorithm, Picasso, where the parameters of the algorithm provide a trade-off between the quality of the solution (i.e., the number of colors used) and the resource consumption of the algorithm. We theoretically prove that Picasso has a sublinear space requirement (§ IV), making it attractive to implement on GPUs. We empirically show that Picasso can solve *trillion edge graph problems* arising from molecules with more than 2 million Pauli strings in under fifteen minutes. Many of the results we report are the first-of-its-kind for the corresponding molecule and basis set. Since our parametric algorithm considers the target number of colors, we show that high-quality coloring can be achieved by an aggressive choice of parameters (§ VII-A1). We design a machine learning prediction model to determine the parameters configuration to

be used to achieve a given trade-off between coloring quality and resource requirements (§VI).

Although `Picasso` is designed to solve a specific problem in quantum computing, it can be used in a generalized graph setting where memory efficiency is needed. The main contributions of this work are:

- We introduce a first-of-its-kind graph coloring algorithm `Picasso` to address the unitary partitioning problem in quantum computing with demonstrations for dense inputs with up to two million vertices and over a *trillion edges*.
- We prove that, under a practical assumption `Picasso` has *sublinear memory* requirement with high probability.
- We propose a machine learning approach to predict the configuration of the algorithm’s parameters that allows achieving a given trade-off between coloring quality, runtime and memory requirements
- We demonstrate the practical efficiency of our approach. The GPU-accelerated `Picasso` enables to process inputs with $49.5\times$ more Pauli strings and $2,478\times$ more edges than existing state-of-the-art parallel approaches.

II. PROBLEM FORMULATION

Our work is driven by the challenge of identifying compact unitary representations of chemical Hamiltonians and strongly correlated wave functions to enable accurate and efficient quantum simulations. Typically, this challenge can be abstracted as determining how to solve a clique partitioning problem efficiently. In this section, we delve into the graph formulation of the problem.

A. Quantum computing problem

In quantum simulations aimed at elucidating physical and chemical phenomena, both the wave function and the system Hamiltonian play crucial roles. The wave function encodes the probabilistic state of a quantum system, facilitating the computation of various physical properties, whereas the system Hamiltonian governs the temporal evolution of the system’s states according to quantum mechanical principles and represents the total energy of the system. In this context, the wave function is responsible for state preparation, and the system Hamiltonian controls the evolution of the state.

In these simulations, the processes of state preparation and evolution *must* be conducted through *unitary operations*. These are operations that maintain the norm (magnitude) of the state vector in Hilbert space, ensuring that the total probability remains one. To accomplish this, both the system Hamiltonian and the wave function generator must be reformulated as either a unitary operator or a linear combination of unitary operators. This reformulation is crucial because unitary operations are reversible and preserve the quantum information within the system, making them indispensable for the coherent manipulation of quantum states in simulations. In small-scale demonstrations, one could employ, for instance, the Jordan-Wigner, Bravyi-Kitaev, or parity techniques [11], [12], which rewrite the Hamiltonian and the wave function generator as a combination of Pauli strings, each being a unitary. A Pauli

string represents a tensor product of a series of 2×2 Pauli matrices, $\sigma_x, \sigma_y, \sigma_z$, and the 2×2 identity matrix, I . However, these techniques encounter the curse of dimensionality in large-scale applications. For instance, in molecular applications, reformulating the molecular Hamiltonian represented in the N -spin-orbital basis set would require $\mathcal{O}(N^4)$ Pauli strings. The number of the Pauli strings needed to reformulate the highly correlated wave function ansätze, such as the non-unitary coupled-cluster ansatz that include single and double excitations, would be even higher, i.e. $\mathcal{O}(N^{7\sim 8})$. The curse of dimensionality makes the subsequent quantum simulation, especially quantum measurements, extremely challenging. Compact unitary representation for both the Hamiltonians and the wave function generators addresses the curse of dimensionality and the associated quantum measurement overhead.

Mathematically, given a set of n Pauli strings, $P = \{P_1, P_2, \dots, P_n\}$ each of which is of length N , along with their coefficients $\{p_1, p_2, \dots, p_n\}$, we aim to compute a smaller set of unitaries $\{U_1, U_2, \dots, U_c\}$ with the corresponding coefficients $\{u_1, u_2, \dots, u_c\}$, such that:

$$\sum_{i=1}^c u_i U_i = \sum_{j=1}^n p_j P_j, \quad c < n. \quad (1)$$

B. Connection to Clique partitioning and Graph coloring

A straightforward way for satisfying Eq. (1) is enabled through the following condition for m Pauli strings [13]:

$$\sum_{i,j=1, i \neq j}^m p_i^* p_j P_i P_j = 0, \quad m \geq 2, \quad (2)$$

for which a necessary condition is enabled by the *anticommutate property* between any two Pauli strings in the given set, i.e.,

$$\{p_i P_i, p_j P_j\} = p_i^* p_j P_i P_j + p_j^* p_i P_j P_i = 0, \quad i \neq j. \quad (3)$$

The above relationships can be translated to a *clique partitioning* and *coloring* problem in a graph as follows. Given a set of Pauli strings, P , we generate a graph $G(P, E)$, where E is the set of all possible (unique) pairs of Pauli strings in P that anticommute, i.e., satisfy the Eq. (3). A *clique* (or a complete subgraph) of a graph is a set of vertices such that every (unique) pair of vertices in that set are connected with an edge in G . A clique in G thus corresponds to satisfying Eq. (2). Our goal is to generate a set of cliques, as small as possible, in G , which forms a partition of P . We formally define the problem as follows.

Definition 1 (Clique partitioning): Given a set of Pauli strings, $P = \{P_1, P_2, \dots, P_n\}$, and a graph $G(P, E)$ generated from P , the clique partitioning problem is to compute a collection of cliques, $U = \{U_1, \dots, U_c\}$, where $\bigcup_i U_i = P$, and $U_i \cap U_j = \emptyset$, for $i \neq j$ that minimizes the number of cliques, i.e., c .

The clique partitioning problem is NP-Complete [14], which makes any optimal polynomial time algorithm unlikely. This problem is equivalent to the well-known *graph coloring* problem, which requires finding the smallest set of colors such that each vertex of the graph is assigned exactly one color from the

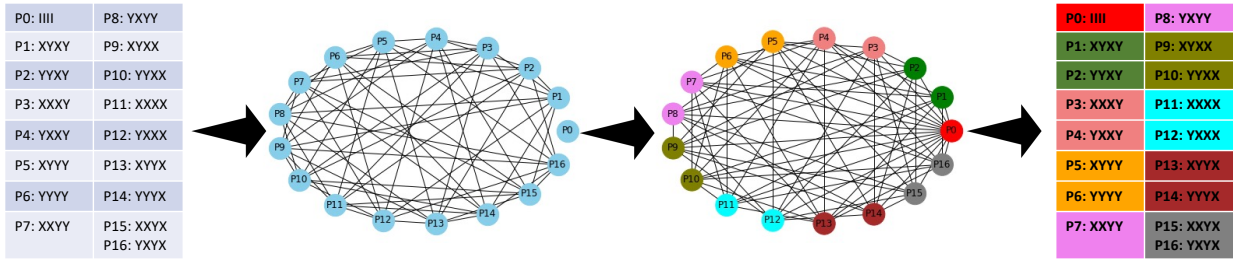


Fig. 1: An overview of the mapping problem solved as clique partition using graph coloring of the conflict graph for H2 molecule with sto-3g basis function.

set, and no two endpoints of an edge have the same color. The clique partitioning of a graph G is equivalent to the coloring of the complement graph G' [15]. Let $G'(V, E')$ be the complement of a graph $G(V, E)$, where the vertex set remains the same but the edge set, $E' = \{\{V \times V\} \setminus E\} \setminus \{\{v, v\} : v \in V\}$. Since the vertices of G' with the same color must represent a clique in G , it is easy to verify that any proper coloring of G' provides a feasible solution to the clique partitioning of G . We further note that when $|E| \geq |E'|$, computing clique partitioning via coloring of the complement graph is more efficient.

In this paper, we solve the unitary partitioning problem by formulating it as a clique partitioning problem, which in turn is solved using graph coloring on the complement graph. An illustration of the process is shown in Fig.1 using the clique partitioning to generate a compact representation of the H2 molecule with sto-3g basis function, with $N = 4$, as an example. We begin with a set of Pauli strings as vertices and compute the edges of the graph using Eq.3. We then construct the complement graph and color it. Finally, we output the partition according to the color classes. In the example, 17 Pauli strings are shrunk to a set of 9 unitaries.

III. RELATED WORK

Mapping Pauli Strings: To facilitate efficient term-by-term measurement schemes in NISQ devices, efforts to minimize the number of terms representing the Hamiltonian and wave function generators have been essential. Previous developments in hybrid quantum-classical algorithms have primarily focused on classically grouping Pauli strings to reduce quantum measurements. Drawing inspiration from the concept of Mutually Unbiased Bases (MUB) in quantum information theory [16], [17]—which is associated with maximizing the information gained from a single measurement—early endeavors have leveraged the commutativity of Pauli strings. This includes qubit-wise commutativity [18], general commutativity [9], [19], and unitary partitioning [20], to define the edges when translating the grouping problem to a clique partitioning and coloring problem in a graph composed of these Pauli strings.

For general molecular cases explored in quantum simulations, these strategies have the potential to group $(4^{N_q} - 1) N_q$ -qubit Pauli strings (excluding the identity string) into no more than 3^{N_q} groups, with further reductions to $\mathcal{O}(N_q^{2 \sim 3})$

possible, albeit at the cost of introducing additional one/multi-qubit unitary transformations before measurement. However, the polynomial scaling inherent in these grouping schemes renders script-based large-scale applications unscalable. This necessitates the development of high-performance computing libraries specialized in graph analysis for this purpose.

Graph Coloring: Graph coloring has been studied extensively in literature. The application of graph coloring in automatic (or algorithmic) differentiation has led to the study of different types of graph coloring problems [2], and a software library of serial implementations called ColPack [3]. Sequential coloring algorithms are based on *greedy* methods, which, given an ordering of the vertices, employ the smallest feasible color for each vertex. In the worst case and for general graphs, all these ordering-based methods require $\Delta + 1$ colors, where Δ is the maximum degree of the graph. The ordering methods include Largest Degree First (LF), Smallest Degree Last (SL), Dynamic Largest Degree First (DLF), and Incidence Degree (ID). While greedy coloring techniques provide reasonable quality in sequential settings, these algorithms have little to no concurrency in practice. For parallel settings, there are two primary algorithmic techniques: *i*) heuristics building on the idea of finding maximal independent sets, introduced in the pioneering work of Luby [21] and extended by Jones and Plassmann [22] (JP), *ii*) heuristics leveraging *speculation*, where parallel threads speculatively color vertices using the least available color. Conflicts resulting from concurrent execution are then corrected in an iterative manner [23]. Similar approaches have been extended to manycore or GPU implementations [5], [24], [25], and distributed-memory algorithms and implementations [26], [27]. In § VII we compare Picasso for quality (against [3]) and performance (against [5], [25]).

The existing parallel graph coloring algorithms, especially the single-node GPU solutions, fail to solve large graph problems due to memory limitations. The graphs need to be loaded into the GPU memory along with auxiliary data structures such as an array of “forbidden colors”, which can easily exhaust the available memory. The graphs generated by our target applications are large and dense. Thus, there exists a need for coloring algorithms that are memory efficient.

Graph Coloring in sublinear space: The recent seminal work of Assadi, Chen and Khanna [10] (ACK, henceforth), studied the $(\Delta + 1)$ -coloring problem with sublinear space

and time constraints. They developed the *Palette Sparsification Theorem*, which reduces the $\Delta+1$ -coloring of a graph $G(V, E)$ to a list-coloring problem in a subgraph of G with only $\mathcal{O}(|V| \log^2 |V|)$ edges. Applications of this theorem is shown by designing algorithms in dynamic semi-streaming in a single pass, sublinear query and MPC models. In terms of semi-streaming algorithm, the only known $(\Delta + 1)$ -coloring before ACK’s algorithm [10] was the $\mathcal{O}(\log |V|)$ pass distributed algorithm of Luby [21], [28], simulated in streaming setup. In semi-streaming model, ACK uses palette size as $\Delta + 1$, list size of colors as $\mathcal{O}(\log |V|)$ for each vertex, and a special post-processing step. In this paper, we put ACK’s algorithm in practice by non-trivial modifications listed as follows.

- i) The palette size in ACK’s algorithm is $\Delta+1$, which limits the practicality of the algorithm on problems operating on large dense graphs, like the one we considered. Our graphs (original and complement) are dense ($\Delta > |V|/2$). One would require a fraction of $|V|$ to color all the vertices, as shown in Table III in § VII-A1 for our dataset ($\leq 16\%$ of $|V|$). Our algorithm allows users to specify a variable palette size. Modifying the analysis of [10], we proved that if the ratio Δ/\mathcal{P} is bounded (by $\log |V|$), a sublinear space is guaranteed. This assumption holds for our graphs since this ratio is a constant for our use cases.
- ii) The conflict graph coloring algorithm of [10] decomposes the graph and then applies a greedy coloring, an almost clique coloring, and a maximum matching-based coloring on the decomposition. We provide an efficient implementation of list-greedy-based coloring that dynamically colors the vertices based on their color list size.
- iii) ACK’s streaming algorithm is single-pass. For valid coloring, the single iteration algorithm would require a large palette size, degrading the solution quality. We address this issue with an iterative approach, where in each iteration, we attempt to color the uncolored vertices from the previous iteration. Our proof for sublinear space holds for each iteration.

IV. OUR ALGORITHM

TABLE I: Notation used in the paper.

| Symbol | Description |
|----------------------|---|
| \mathcal{P} | Set of Pauli strings |
| N | Length of each Pauli string |
| ℓ | Iteration counter |
| $G_\ell = \{V, E\}$ | Complement graph from $V \subseteq \mathcal{P}$ at it. ℓ |
| n | number of vertices, $n := V $ |
| $G_c = \{V_c, E_c\}$ | Conflict graph, $V_c \subseteq V$ |
| $\delta(v)$ | Degree of vertex v , with neighbor set: $adj(v)$ |
| \mathcal{P} | Palette size, Palette= $\{(\ell - 1)\mathcal{P}, 1, \dots, \ell\mathcal{P} - 1\}$ |
| \mathcal{C} | Final number of colors, $\mathcal{C} \leq \mathcal{P}$ |
| α | Multiplicative factor for List size |
| \mathcal{L} | List Size, set to $\alpha \log V $ |
| β | Weighting factor for bi-objective optimization |

Picasso, listed in Algorithm 1, attempts to color the graph by initially assigning to each vertex a list of candidate colors chosen uniformly at random from a palette of \mathcal{P} colors

Algorithm 1 Picasso: Palette-based graph coloring

Input: A graph $G = (V, E)$
Output: A *color* array, with a valid coloring of G

- 1: Initialize the *color* array
- 2: $\ell = 1$ \triangleright The iteration number
- 3: $G_\ell \leftarrow G$
- 4: **while** V is not empty **do**
- 5: $(\mathcal{P}_\ell, \mathcal{L}_\ell) \leftarrow$ Initialize palette and list size for G_ℓ
- 6: $colList \leftarrow$ assign_rand_list_colors($\mathcal{P}_\ell, \mathcal{L}_\ell, G_\ell$)
- 7: $G_c \leftarrow$ construct_conflict_graph($colList, G_\ell$)
- 8: color_unconflicted_vertices($V \setminus V_c, colList, colors$)
- 9: $V_u \leftarrow$ color_conflict_graph($G_c, colList, colors$)
- 10: $\ell = \ell + 1$
- 11: $G_\ell \leftarrow$ subgraph induced by V_u in G
- 12: $V \leftarrow V_u$

$\{0, 1, \dots, \mathcal{P} - 1\}$, and progressively assigning to each vertex a color from its color list without violating the graph coloring constraint. It takes a graph G as input and computes a valid coloring of G in the *color* array. The algorithm starts with the original graph G , and iteratively computes coloring for a subgraph. In each iteration (ℓ), it estimates the palette size (\mathcal{P}_ℓ) and list size (\mathcal{L}_ℓ) for the current subgraph G_ℓ (Line 5). Next, it assigns to each vertex a list of candidate colors chosen uniformly at random from the palette (Line 6). For each vertex, we record the list of colors in the *colList* data structure, where $colList(u)$ refers to the list of colors assigned to vertex u . We build the conflict subgraph, which consists of the edges that share a common color in their corresponding lists (Line 7) using the *colList* array. We then color the unconflicted vertices using an arbitrary color from their color list (Line 8), and attempt to color the conflict graph (Line 9). The vertices uncolored in the current iteration are denoted as V_u . We then compute the subgraph induced by V_u and continue if V_u is non-empty. We note that the algorithm in every iteration starts with a new palette of colors and attempts to color the subgraph with these new colors. The colors of an iteration are not reused in the subsequent iterations. We can ensure that by defining the palette set as $\{(\ell - 1)\mathcal{P}, \dots, \ell\mathcal{P}\}$ at iteration ℓ .

We now describe the construction and coloring of the conflict graph at iteration ℓ . For ease of presentation, we omit the subscript ℓ .

A. Conflict Graph Construction

An edge (u, v) in G is conflicted if its two endpoints share a color, i.e., $colList(u) \cap colList(v) \neq \emptyset$. If the color values in *colList* are sorted, we need $O(\mathcal{L})$ time to check for a conflict edge. In our application, we are not provided with the graph. Instead, we are given a set of Pauli strings \mathcal{P} that defines the vertex set of G . We use a bit encoding scheme to dynamically derive the edges from Pauli strings in a memory-efficient way. Given two Pauli strings P_i and P_j , whether an edge exists between them (in our case, a non-edge) can be checked using

Eq. 3. Here, a Pauli string consists of N characters, where each character corresponds to a 2×2 Pauli matrix:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \quad (4)$$

When directly implementing the anti-commute condition given by Eq. 3, we must carry out $N - 1$ tensor products for each string, followed by two matrix multiplications for matrices P_i and P_j . However, the inherent properties of Pauli matrices allow us to check this condition efficiently. Specifically, any pair of Pauli matrices will either commute or anti-commute. Importantly, two distinct Pauli matrices will anti-commute

$$\{\sigma_i, \sigma_j\} = 0 \text{ if } \sigma_i \neq \sigma_j \neq I, \quad (5)$$

which is simplified to an element-wise character comparison.

The anti-commutation property can be extended to Pauli strings. This is done by counting the element-wise character comparisons corresponding to the anti-commute relation. It is worth noting that the anti-commutation relationship between two Pauli strings is determined by phase $\pm i$. Only an odd number of element-wise anti-commuting character comparisons will result in a non-vanishing phase.

Further reduction in the number of comparisons can be achieved using bit representations. Specifically, we make an observation: as there are only four possible matrices $(\sigma_x, \sigma_y, \sigma_z, I)$, we can encode an 8-bit character datatype to a smaller representation using bits. If we encode each Pauli matrix into a 2-bit value, we would still need to perform N comparisons to count the number of mismatches. To determine a complement edge, we only check whether the number of mismatches is even or odd, which is determined by observing the least significant bit of the count value. With this, we need an encoding that causes a bit flip only when there is a mismatch between σ_x, σ_y , or σ_z .

To accomplish this, we implement an encoding scheme similar to an inverse one-hot encoding, where matrices σ_x, σ_y , and σ_z are assigned 110, 101, and 011, and I is assigned 000, respectively. We perform a bitwise AND operation between the encoded Pauli strings and perform a `popcount` operation to count the number of ‘1’ bits. Due to this encoding scheme, the only time the least significant bit is flipped in the `popcount` is during a mismatch, allowing us to track whether there is an odd or an even number of mismatches. Speedups from the encoded implementation on CPU range from 1.4 to 2.0 \times , including the encoding overheads.

B. Coloring the Conflict Graph

Once we compute the conflict graph, G_c we are required to color G_c using the list of colors assigned to each vertex. Here, we discuss two possible approaches to achieve that.

Static order schemes: Given an ordered set of vertices, we iterate through the set and attempt to color each vertex with the first available color in its list that does not conflict with the colors already assigned to its neighbors. We may use

popular [2] vertex order such as the Natural, Largest Degree First, Smallest Degree Last, or Random ordering.

Dynamic vertex order scheme: A second approach is to color the conflict graph using the lists in a dynamic order [29]. Here, we describe a dynamic greedy algorithm from [29] for the list coloring and discuss an efficient implementation of it. The algorithm attempts to color the vertices that are most constrained, i.e., it colors a vertex according to the size of the list. When a vertex is assigned a color from its list, the assigned color is removed from the list of all the neighbors of the vertex, rendering a dynamic order on the vertices. A naïve implementation of the dynamic list coloring algorithm would require $\mathcal{O}(|V_c|^2 + |E_c|\mathcal{L})$ time. We have at most $|V_c|$ iterations, and in each iteration, we need to find the vertex with the smallest current list size in $\mathcal{O}(|V_c|)$ time, then color the vertex and mark this color removed from all its neighbors in $\mathcal{O}(\delta(v)\mathcal{L})$ time, where $\delta(v)$ is the degree of vertex v . Summing over $|V|$ gives us the total time. We can reduce it to $\mathcal{O}((|V_c| + |E_c|\mathcal{L}) \log |V_c|)$ using a minimum heap data structure with logarithmic `update_key` operation. Next, we present an efficient implementation in Algorithm 2 that eliminates the $\log |V_c|$ factor by using the bucketing technique.

Algorithm 2 Greedy list-coloring of conflict graph, G_c

Input: Conflict graph G_c , color list $colList(v), \forall v \in V_c$.
Output: A coloring of G_c using the lists, and the uncolored vertex set, V_u .

- 1: $B \leftarrow$ An array of bucket lists
- 2: \triangleright *Creating the initial buckets*
- 3: **for** $v \in V$ **do**
- 4: \lfloor Insert v to $B[colList(v).size()]$
- 5: Mark all vertices of G_c as unprocessed
- 6: $V_u \leftarrow \emptyset$
- 7: **while** \exists an unprocessed vertex **do**
- 8: Pick a vertex, v from the lowest bucket
- 9: Color v from $colList(v)$ chosen uniformly at random, say c
- 10: Remove v from its bucket and mark it as processed
- 11: **for** $u \in adj(v)$ **do**
- 12: \triangleright *Neighbors of v in G_c*
- 13: **if** u is uncolored and $c \in colList(u)$ **then**
- 14: Remove c from $colList(u)$
- 15: **if** $colList(u)$ is empty **then**
- 16: Mark u as processed
- 17: $V_u = V_u \cup u$
- 18: \lfloor Continue
- 19: Remove u from its current bucket
- 20: \lfloor Insert u to $B[colList(u).size()]$

Algorithm 2 stores the vertices of G_c in an array of buckets B , according to the size of their color lists. The bucket at $B[i]$ holds the vertices $v \in V$, whose size of the $colList(v)$ is i . We define the *lowest bucket* from the array as the non-empty bucket with the smallest index. The algorithm marks all the vertices as unprocessed and continue until all the vertices are processed. In each iteration, the algorithm finds the lowest

bucket and chooses a vertex uniformly at random from it. This vertex is colored with an arbitrary color from its list and marked as processed. We say the color is c . Then, the algorithm scans all neighbors of this vertex in G_c and removes c if it exists in their color list. If any neighbor's list becomes empty, we mark this neighboring vertex as processed. The runtime of Algorithm 2 is $\mathcal{O}((|V_c|+|E_c|)\mathcal{L})$, since there might be at most $|V_c|$ iterations and in each iteration we require $\mathcal{O}(\mathcal{L})$ time to find the lowest bucket. The removals of a vertex from a bucket (Lines 10 and 19) can be implemented in constant time by an auxiliary array that stores the location of the vertex in the bucket, while the removal of a color from $colList$ (Line 14) takes $\mathcal{O}(\mathcal{L})$. Thus, processing a chosen vertex requires $\mathcal{O}(\delta(v)\mathcal{L})$ time, and summation over all vertices gives the total time.

C. Analysis of the Algorithm

Building on the analysis in [10], we now show that, under reasonable assumptions on palette size, with a high probability our algorithm constructs in each iteration a conflict graph that is sublinear in the graph size. Following the previous section, we will omit the subscript ℓ since the results hold for any iteration. Recall that G_c is the conflict subgraph computed from G and $n := |V|$. We will require the following concentration results.

Lemma 1 (Chernoff-Hoeffding bound [30]): Let X_1, \dots, X_m be m independent binary random variables such that $Pr(X_i) = p_i$. Let $X = \sum_{i=1}^m X_i$ and $\mu = \mathbb{E}[X]$. Then the following holds for $0 < \gamma \leq 1$:

$$Pr[X \geq (1 + \gamma)\mu] \leq e^{-\frac{\mu\gamma^2}{3}} \quad (6)$$

Lemma 2: At iteration, ℓ , let $G(V, E)$ be the graph to be colored with Palette size \mathcal{P} , and the color list size for each vertex $\mathcal{O}(\log n)$. Let the average and maximum degree of G be \bar{d} and Δ , respectively. Then, the total number of colors found by Algorithm 1 is $\sum_{\ell} \mathcal{P}_{\ell}$. At iteration ℓ of the Algorithm 1, the following relations hold:

- 2.1) Expected degree of vertex $v \in G_c$ is $\mathcal{O}(\frac{\delta(v)}{\mathcal{P}} \log^2 n)$.
- 2.2) Assuming $\frac{\Delta}{\mathcal{P}} = \mathcal{O}(\log n)$, the maximum degree and the maximum number of edges in G_c is $\mathcal{O}(\log^3 n)$ and $\mathcal{O}(n \log^3 n)$ respectively with high probability.
- 2.3) Assuming $\frac{\bar{d}}{\mathcal{P}} = \mathcal{O}(\log n)$, the expected number of edges in G_c is $\mathcal{O}(n \log^3 n)$.

Proof: Due to the design of the algorithm, the total number of colors used is: $\sum_{\ell} \mathcal{P}_{\ell}$.

1) Let us consider vertex v , and let T be the size of the color list for v . Let $X_{v,u}$ be a binary random variable that takes 1 if the edge (v, u) is in the conflict graph G_c , and 0 otherwise. Also, let $X_v = \sum_{u \in adj(v)} X_{v,u}$. Let us fix the colors in the list of v as c_1, c_2, \dots, c_T . The probability that at least one of these colors is shared with a neighbor, u , i.e., $Pr(X_{v,u} = 1)$ is $\mathcal{O}(\frac{T}{\mathcal{P}})$. So the expected degree of v is $\mathbb{E}[X_v] = \sum_{u \in adj(v)} \mathcal{O}(\frac{T}{\mathcal{P}}) = \mathcal{O}(\frac{\delta(v)}{\mathcal{P}} \log^2 n)$, since $T = \mathcal{O}(\log^2 n)$.

2) Since the maximum degree is Δ , the maximum expected degree in G is $\mathcal{O}(\frac{\Delta}{\mathcal{P}} \log^2 n) = \mathcal{O}(\log^3 n)$ according to Lemma (2.1) and our assumption on the ratio $\frac{\delta(v)}{\mathcal{P}}$. We will now show the high probability concentration result. We note that all $X_{v,u}$ s are independent of each other. So, using the Chernoff-Hoeffding bound (Lemma 1) and setting $\gamma = 1$, the probability that $Pr[X_v \geq 2(\log^3 n)] \leq e^{-(\log^3 n)/3} \leq \mathcal{O}(n^{-\log^2 n})$. So with high probability the maximum degree of v is $\mathcal{O}(\log^3 n)$ assuming the ratio $\frac{\Delta}{\mathcal{P}} = \mathcal{O}(\log n)$.

3) We can further improve the bound by using the average degree (\bar{d}) of G rather than the maximum degree on our assumption. Let Y be the random variable representing the sums of degrees of the graph at level ℓ . From Lemma (2.1), the expected sums of degrees,

$$\begin{aligned} \mathbb{E}[Y] &= \sum_{v \in n} X_v = \sum_{v \in V} \mathcal{O}\left(\frac{\delta(v)}{\mathcal{P}} \log^2 n\right) \\ &= \sum_{v \in V} \mathcal{O}\left(\frac{\bar{d}}{\mathcal{P}} \log^2 n\right) = \mathcal{O}(n \log^3 n). \end{aligned}$$

This relationship follows since the sums of degrees of a graph can be replaced with sums of average degrees. The final equality follows from our assumption, $\frac{\bar{d}}{\mathcal{P}} = \mathcal{O}(\log n)$. ■

V. PARALLEL GPU IMPLEMENTATION

Algorithm 3 GPU Conflict Graph Construction

Input: Pauli strings, V and their list of colors $colList$
Output: Conflict graph G_c in CSR format

- 1: AvailMem = min(2|V|(|V| - 1), MaxAvailGPUMem)
- 2: Allocate AvailMem on the GPU
- 3: $V_{edgecount}, E_{coo} \leftarrow \text{build_unordered_coo}(colList, V)$
- 4: $V_{offsets} \leftarrow \text{exclusive_sum}(V_{edgecount})$
- 5: **if** $|E_{coo}| \leq \text{AvailMem}/2$ **then**
- 6: $G_c \leftarrow \text{generate_csr_gpu}(V_{offsets}, E_{coo})$
- 7: **else**
- 8: $G_c \leftarrow \text{generate_csr_cpu}(V_{offsets}, E_{coo})$

The conflict subgraph construction on Line 7 of Algorithm 1 significantly dominates the execution time for our application. Because of this, we implemented a parallel GPU version to reduce the bottleneck. We note that, despite the original graph being dense, in almost all the cases the conflict graph is expected to be significantly sparse (details in §VII-A1). Although the pairwise comparisons of vertices are independent of each other, due to the unknown number of conflicting edges at runtime, we designed an implementation capable of generating the conflict graph in Compressed Sparse Row (CSR) format that is not only efficient in memory usage during construction but also enables contiguous access of memory for processing the conflict graph. We present this implementation in Algorithm 3. As a preprocessing (now shown in the Algorithm 3), we copy the input data on the GPU with a size of $N\mathcal{L}|V|/10$ 4-byte values for the encoded Pauli strings and

their list of colors, and initialize $2|V|$ edge offset counters. We use 8 bytes for the counter if $|V|^2 \geq 2^{32}$; otherwise, the offset counters are 4-bytes. All remaining available memory on the GPU, or the worst-case edgelist size of $2|V|(|V| - 1)$, whichever is smaller, is then allocated to store the unordered edgelist, and the conflict graph generation kernel is launched on the GPU. For this kernel, each thread processes one of the $|V|(|V| - 1)/2$ possible edges and, and inspect whether the edge is both a complement and conflicting. If so, the edge is written in the output and the respective edge offsets are incremented. After the kernel execution, we are left with an unordered edge list of size $|E_c|$. The edges of the complement graph are determined independently during conflict graph construction, and the complement graph does not need to be stored on the GPU memory. Since for CSR representation each edge is stored twice, if $|E_c|$ used less than half of the available GPU memory, then we generate the CSR output on the GPU. Otherwise, we read the unordered edge list and convert it to CSR on the host CPU instead. We attempted warp-level reduction on the offset values to condense the number of global atomic operations, but the number of conflicting edges was much smaller than the total number of possible edges ($\leq 5\%$ in most cases), so the overhead outweighed the benefits. To get an estimation for $|E_c|$ to preallocate memory on the GPU, we developed a machine learning based predictor, which we describe next (§VI).

VI. PREDICTION OF PALETTE SIZE

We employ a machine learning (ML) based methodology to predict the palette size \mathcal{P} and α values to simultaneously minimize the number of final colors \mathcal{C} and the number of conflicting edges $|E_c|$. As these two objectives are conflicting, we introduce β as the weighting factor to determine the balance between minimizing \mathcal{C} and $|E_c|$. The goal for this prediction is to find the optimal combination of (\mathcal{P}, α) to minimize $(\beta \cdot \mathcal{C} + (1 - \beta) \cdot |E_c|)$, i.e.,

$$\min_{(\mathcal{P}, \alpha)} (\beta \cdot \mathcal{C} + (1 - \beta) \cdot |E_c|). \quad (7)$$

We generate a dataset by varying values for percentile palette size, $\mathcal{P}' = \frac{\mathcal{P}}{|V|} \times 100$ (as a percentage of the number of vertices $|V|$ in the complement graph G') and α to perform a grid search. We capture the (\mathcal{P}', α) combinations that minimizes Eq. (7) for different values of β . Finally, this dataset trains regression models to predict the (\mathcal{P}', α) for a given graph G and β . We train the regressor with several molecules and test its accuracy on a new set of molecules. The methodology can be summarized as follows:

- *Step 1:* For a given graph $G(V, E)$, perform sweeps on (\mathcal{P}', α) and compute (\mathcal{C}, E_c) .
- *Step 2:* For a particular value of β , compute objective in Eq. (7), and select the optimal choice of $(\mathcal{P}', \alpha)_{opt}$.
- *Step 3:* Run Step 2 for different values of β , and collect corresponding values of $(\mathcal{P}', \alpha)_{opt}$ to construct the data-set for the graph G .

- *Step 4:* Run Steps 1-3 for the graphs correspond to different molecules to construct the complete training set.
- *Step 5:* Train the regressor model with (\mathcal{P}', α) as outputs of the model for a given graph G and β .
- *Step 6:* After the model is trained, we provide a new graph and a particular choice of β , for which, the model predicts an optimal choice of (\mathcal{P}', α) that optimizes for Eq. (7).

Model Training and Results: We generated a dataset for the molecules provided in Table II for percentile palette sizes $\mathcal{P}' \in \{1\%, 2.5\%, 5\%, \dots, 20\%\}$ and $\alpha \in \{0.5, 1.0, \dots, 4.5\}$. We capture the (\mathcal{P}', α) combinations that minimized Eq. (7) for $\beta \in \{0.1, 0.2, \dots, 0.9\}$. From the dataset, we used the first five molecules for training and the last two for testing the regression analysis. We experimented with several linear (ridge, lasso) and nonlinear predictors (svm-kernel-rbf, decision trees, random forests) [31]. For given input of: (β, V, E) , the nonlinear regression models performed better in predicting the ideal (\mathcal{P}', α) combination. In particular, the *random forest regressor* provided the best performance with multiple iterations, with a mean absolute percentage error (MAPE) of 0.19 and an R-squared value of 0.88 over 100 iteration runs. We selected the number of trees (estimators) to be 100 and the maximum tree depth of 20. This methodology provides `Picasso` a means to predict optimal choice of palette and list sizes for an input with values for (β, V, E) , where β provides the trade-off. We note that while our model is trained specifically for the dense input graphs used in this work, it can be extended to any family of inputs that can be well characterized with data as described above.

VII. EXPERIMENTAL EVALUATION

We evaluate the performance and quality of the solution obtained using `Picasso` to the current state-of-the-art approaches. For our evaluation, we used a machine equipped with a 64-core AMD EPYC 7742 CPU, 512GB host DDR memory, and an NVIDIA A100 GPU with 40GB of HBM memory. Table II lists the datasets we selected for our evaluation. These systems are chosen with a deliberate intent to

TABLE II: Molecule Dataset in our experiment

| Molecule Name | # of qubits | # of Pauli terms | # of edges |
|---------------|-------------|------------------|-------------------|
| H6_3D_sto3g | 12 | 8,721 | 19,178,632 |
| H6_2D_sto3g | 12 | 18,137 | 82,641,188 |
| H6_1D_sto3g | 12 | 19025 | 90,853,544 |
| H4_2D_631g | 16 | 22529 | 127,024,320 |
| H4_3D_631g | 16 | 34481 | 297,303,496 |
| H4_1D_631g | 16 | 42449 | 450,624,984 |
| H4_2D_6311g | 24 | 154641 | 5,979,614,600 |
| H4_3D_6311g | 24 | 245089 | 15,017,722,736 |
| H8_2D_sto3g | 16 | 271,489 | 18,513,622,112 |
| H8_1D_sto3g | 16 | 274,625 | 18,944,162,720 |
| H4_1D_6311g | 24 | 312817 | 24,464,823,272 |
| H8_3D_sto3g | 16 | 419,457 | 44,149,092,736 |
| H6_3D_631g | 24 | 554,713 | 77,027,619,060 |
| H10_3D_sto3g | 20 | 1,274,073 | 410,446,230,804 |
| H6_2D_631g | 24 | 2,027,273 | 1,028,164,570,684 |
| H6_1D_631g | 24 | 2,066,489 | 1,068,358,440,628 |
| H10_2D_sto3g | 20 | 2,093,345 | 1,108,417,973,696 |
| H10_1D_sto3g | 20 | 2,101,361 | 1,116,895,244,280 |

encompass a broad spectrum of quantum scenarios. By picking H_n molecular systems with varying values of $n = 4, 6, 8, 10$, we aim to ensure diversity in system size, ranging from simple to complex structures, enabling us to gauge the algorithm’s scalability and performance across different magnitudes. The incorporation of three spatial configurations for each H_n molecular system, namely 1D, 2D, and 3D, introduces dimensional variability, shedding light on the tool’s capability to manage problems with different geometric complexities and symmetries. As the size of the H_n system increases, so does the intricacy of electron-electron interactions and correlations. By integrating systems of different sizes, the goal is to critically assess the tool’s proficiency in addressing varying degrees of electron correlation, a pivotal aspect in quantum calculations. Moreover, this diverse selection, spanning across multiple system sizes and dimensions, serves as an effective stress test for `Picasso`. It not only offers insights into its performance benchmarks but also aids in pinpointing potential areas of enhancement, ensuring a comprehensive evaluation of its reliability and robustness. We classify our datasets into three categories: *i*) Small (≤ 10 Billion edges); *ii*) Medium (≤ 1 Trillion edges); and *iii*) Large (> 1 Trillion edges). We evaluate our implementation by considering:

- **Performance:** We present a detailed performance study of the execution time of `Picasso` and compare it against two of the state-of-the-art GPU implementations of the distance-1 coloring: *i*) Kokkos-EB: edge-based coloring included as part of the kokkos-kernels [4], [25], [32], and *ii*) ECL-GC-R: shortcutting and reduction based heuristics for JP-LDF [5], [33].
- **Quality:** We assess the quality of coloring obtained by `Picasso` with respect to sequential greedy coloring implementations in ColPack [2], [3], and GPU implementations of Kokkos-EB, and ECL-GC-R. We also study the quality versus memory trade-offs in these implementations.
- **Parameter Sensitivity:** We study the impact of palette size (\mathcal{P}) and color list size (α, \mathcal{L}) on the final coloring, the number of conflicting edges, and the runtime of `Picasso`.

We limit our relative comparisons only to the small dataset due to limitations imposed by specific implementations. In fact, ColPack and Kokkos-EB run out of memory beyond the small datasets (Kokkos-EB also runs out of memory for the last instance of the small dataset). ECL-GC-R does not support a graph size larger than what a 32-bit integer type can represent (2 to 4B). All results presented here are averaged over five runs. Five different seeds for pseudo-random number generation are used for `Picasso` runs. For `Picasso`, we present the results using only the greedy list coloring of Algorithm 2 to color the conflict graph since it provided better coloring relative to the static ordering algorithms.

We note that `Picasso` does not require loading the entire graph into memory to color it. Instead, it computes the conflicting subgraph on-the-fly at every iteration, thus

providing the memory improvement. This is fundamentally different from the previous state-of-the-art algorithms. In fact, ColPack, Kokkos-EB, and ECL-GC-R require loading of the entire graph into memory before coloring it. Therefore, we decided to be conservative in comparing the performance of the different implementations, and we *include* the conflict subgraph construction time for `Picasso` while we *exclude* the graph construction time for all the other approaches. Under these conservative settings, we show that `Picasso` is better or comparable to ECL-GC-R and within a factor $2\times$ slower than Kokkos-EB. We also note that the explicit construction of a complement graph is expensive for large instances.

The two main parameters of `Picasso` are the size of palettes (\mathcal{P}) and the color list (\mathcal{L}). At any iteration of the algorithm with V as the vertices considered, in our experiments, \mathcal{P} represents the percentage of vertices, and $\mathcal{L} = \alpha \log |V|$, where α is the coefficient. We omit the subscript since, in each iteration, the same percentage value and α are used. We report the number of colors, running time in seconds, and maximum resident set size in GB for memory. Apart from these, we also define a few other metrics as follows.

- **Color percentage:** percentage ratio between the number of colors to the number of vertices of the input, i.e., $\frac{c}{|V|} * 100$. It represents the percentage of shrinkage of the Pauli strings to the unitaries (impact on the application).
- **Maximum Conflicting Edge percentage:** The percentage ratio between the maximum number of conflicting edges (across all iterations of `Picasso`) to the number of complement edges of the graph, i.e., $\frac{|E_c|}{|E|} * 100$.

A. Quality and Memory Comparisons

TABLE III: Quality comparisons of the algorithms. Results in bold are the best coloring. Norm.: $\mathcal{P} = 12.5\%$, $\alpha = 2$; Aggr.: $\mathcal{P} = 3\%$, $\alpha = 30$.

| Problem | ColPack | | | | Picasso | | Kokkos-EB | ECL-GC |
|-------------|---------|------|-------------|------|---------|---------------|-----------|--------|
| | LF | SL | DLF | ID | Norm. | Aggr. | | |
| H6_3D_sto3g | 2479 | 902 | 901 | 952 | 1425.4 | †880.6 | 1040.17 | 943 |
| H6_2D_sto3g | 5389 | 1598 | 1580 | 1634 | 2901.5 | †1587.4 | 1749.17 | 1596 |
| H6_1D_sto3g | 5771 | 1672 | 1601 | 1689 | 3036.1 | †1650.4 | 1815.83 | 1642 |
| H4_2D_631g | 10049 | 1922 | 1694 | 1917 | 3579.8 | 1784.2 | 1772.67 | 1860 |
| H4_3D_631g | 15883 | 2729 | 2633 | 2668 | 5431.4 | †2606 | 2478.50 | 2596 |
| H4_1D_631g | 19412 | 3241 | 2943 | 3233 | 6538 | 3212.8 | 3426.17 | 3098 |
| H4_2D_6311g | 72493 | 8615 | 6944 | 8628 | 22463.8 | 8917.4 | NA | NA |

1) *Small Dataset:* Tables III and IV show the number of colors achieved and the memory requirements of the compared algorithms for the small dataset, respectively. We experimented with four ordering heuristics for sequential greedy coloring that are commonly considered in the literature: Largest First Degree (LF), Smallest Degree Last (SL), Dynamic Largest Degree First (DLF), and Incidence Degree (ID). The previous study on unitary partitions [13], [20] only considered LF greedy coloring algorithm. We refer you to the excellent survey of Gebremedhin *et al.* [2] for details on vertex orderings.

Table III reports the average number of colors over five runs for all the algorithms. We show results from two different configurations of `Picasso`: *i*) Normal: $\mathcal{P} = 12.5\%$, $\alpha = 2$ and *ii*) Aggressive: $\mathcal{P} = 3\%$, $\alpha = 30$. We observe that

Picasso always produces fewer colors than the LF heuristic using Picasso’s normal mode. We find that the DLF heuristics provide the best coloring. However, the aggressive configuration of Picasso provides coloring that is within **5%** in 4/7-th of the inputs (marked with † in Table III), and within **10%** in all cases except the largest of the small dataset. Finally, we observe that ECL-GC-R provides better coloring than Kokkos-EB on 4/6-th of the inputs for which they could compute a solution. In both cases, the coloring provided by Picasso’s aggressive configuration is better or within 5% of what is obtained through ECL-GC-R and Kokkos-EB.

TABLE IV: Memory comparison of the algorithms: Maximum resident memory in GB

| Problem | ColPack | Picasso | | Kokkos-EB | ECL-GC-R |
|-------------|---------|-------------|-------|-----------|----------|
| | | Norm. | Aggr. | | |
| H6_3D_sto3g | 0.38 | 0.08 | 0.23 | 1.19 | 0.30 |
| H6_2D_sto3g | 1.52 | 0.16 | 0.90 | 4.50 | 0.77 |
| H6_1D_sto3g | 1.68 | 0.17 | 0.97 | 4.93 | 0.83 |
| H4_2D_631g | 2.72 | 0.20 | 1.24 | 6.81 | 1.10 |
| H4_3D_631g | 5.66 | 0.31 | 3.10 | 15.69 | 2.37 |
| H4_1D_631g | 10.77 | 0.38 | 4.31 | 23.69 | 3.51 |
| H4_2D_6311g | 140.23 | 2.06 | 57.12 | NA | NA |

Table IV shows the maximum resident memory in GB of the reference implementations for the small dataset. We find that the normal configuration of Picasso is the most memory efficient. In particular, it requires **68×** lesser memory than the ColPack for H4_2D_6311g while Kokkos-EB and ECL-GC-R run out of memory and couldn’t compute a solution for the same instance. We observe between 14× and 60× lower memory utilization when comparing Picasso’s normal mode to Kokkos-EB, and a reduction in memory usage of $\approx 5\times$ when considering Picasso’s aggressive mode. ECL-GC-R shows to be more memory efficient than Kokkos-EB and is comparable to Picasso’s aggressive mode. However, ECL-GC-R memory optimizations come with runtime penalties that we will detail in our performance evaluation (§ VII-B).

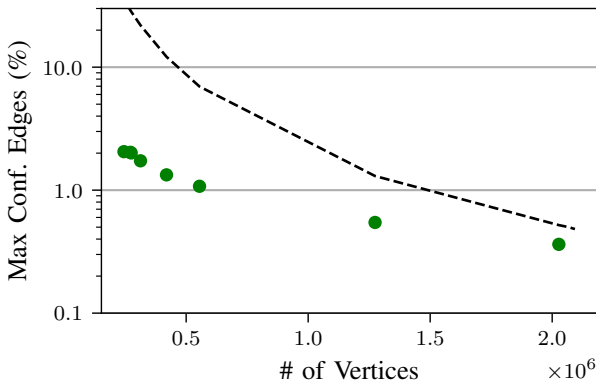


Fig. 2: Input dataset scaling on the iterative GPU implementation up to 2 million vertices. $\alpha = 2$, $\mathcal{P} = 12.5\%$. The black dashed line in the top plot denotes the maximum conflicting edge ratio supported by a 40GB NVIDIA A100.

2) *Medium and Large Dataset*: As mentioned earlier, none of the compared algorithms could generate coloring for these inputs; we will only discuss results obtained from Picasso. Using $\mathcal{P} = 12.5\%$ and $\alpha = 2$, we could color all the medium inputs. We even observe a slight quality improvement compared to the small dataset. For the chosen parameters, the color percentage is **14–15%** (as a percent of $|V|$), whereas for the same parameter setting, Picasso achieved 14–16% for the smaller dataset (see Table III). For the four large inputs (with over 1 Trillion edges), we set $\mathcal{P} = 12.5\%$ as before, but changed $\alpha = 1$. We were then able to generate coloring for all the large instances with this parameter except the largest one, which ran out of GPU memory. For the first three inputs of the large dataset, Picasso achieved a color percentage of **16.2–16.4%**. These results suggest that Picasso can achieve reasonably high-quality coloring even for larger datasets, for which none of the current state-of-the-art GPU implementations included in our study were able to color within the 40 GB of available GPU memory of our system. Due to the quadratic scaling of the complement edges relative to the number of Pauli strings, an increasingly smaller conflicting-edge ratio is needed to satisfy memory requirements for larger problems. This can be observed in Fig. 2 where the black dashed line traces the limits of the maximum fraction of conflict edges (%) that can fit the A100 for each input. We address the memory issue by choosing more conservative parameters (\mathcal{P} and α) as demonstrated for the largest inputs in our dataset.

B. Performance Evaluation

1) *CPU-only Vs. GPU-assisted Implementation*: We report the speedup of our GPU implementation over the CPU-only implementation of Picasso in Table V. We show the average time over five runs for the conflict graph construction and the total time of the CPU-only implementation in the 2nd and 3rd columns, respectively. Here, the reported conflict graph construction time includes the cumulative time spent in building the conflict graphs during each iteration of the algorithm. The process of building conflict graph accounts for over 98% of execution time (geo. mean) for these problems. We accelerate the conflict graph build on GPUs (§ V). The last two columns of Table V report the speedup of our GPU implementation with respect to the conflict graph build step and the total runtime. We see that as the problem size increases, the speedup also increases, and we expect that trend to continue for even larger problems. We report results only for the small datasets because we used a cut-off time of 1 hour, and the CPU implementation was able to complete only the small dataset within that time budget. The geometric mean of the speed up for the conflict graph construction step is $\sim 60\times$, and that results into a $\sim 16\times$ speed up for the entire application (geo. mean). We note that our GPU implementation produces exactly the same coloring as the CPU-only one because the conflict graph construction is deterministic.

2) *Performance on Medium and Large Dataset*: Fig. 3 shows the running time (with a breakdown in components)

TABLE V: Runtime comparison for CPU only and GPU assisted implementation. $\mathcal{P} = 12.5\%$, $\alpha = 2$

| Problem | CPU only | | GPU assisted | |
|-------------|---------------------|---------------|---------------------|---------------|
| | Graph Build Time(s) | Total Time(s) | Graph Build Speedup | Total Speedup |
| H6_3D_sto3g | 3.14 | 3.26 | 24.37 | 2.21 |
| H6_2D_sto3g | 14.87 | 15.19 | 43.76 | 8.94 |
| H6_1D_sto3g | 16.35 | 16.69 | 45.36 | 9.75 |
| H4_2D_631g | 24.47 | 24.92 | 51.55 | 13.41 |
| H4_3D_631g | 57.67 | 58.40 | 73.21 | 26.30 |
| H4_1D_631g | 91.02 | 92.00 | 83.85 | 35.25 |
| H4_2D_6311g | 1,428.94 | 1,436.10 | 173.36 | 110.90 |
| Geo. Mean | | | 59.54 | 15.98 |

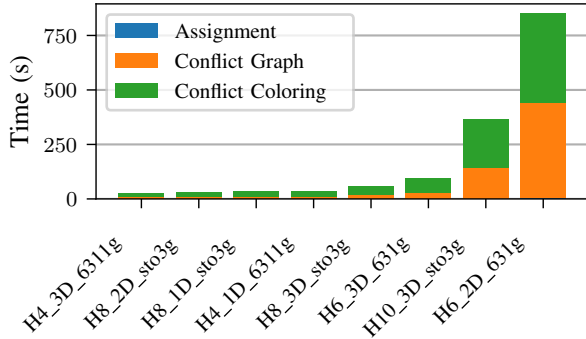


Fig. 3: Input dataset scaling on the iterative GPU implementation up to 2 million vertices. $\alpha = 2$, $\mathcal{P} = 12.5\%$.

for all the medium and one of the large datasets for our GPU implementation. The problems are sorted from left to right, with the smallest problem on the left. For all the inputs, we set $\mathcal{P} = 12.5\%$, and $\alpha = 2$. We see that the conflict coloring that happens in CPU dominates the runtime. Despite this, we were able to color the largest graph with over 1 Trillion edges within **800 seconds** with a color percentage ranging between 14–15%. For detailed quality results, see § VII-A2.

C. Performance Comparison with Kokkos-EB and ECL-GC-R

Fig. 4 compares Picasso to the current state-of-the-art GPU implementations of graph coloring: Kokkos-EB and ECL-GC-R. The study is limited to the small dataset due to memory constraints imposed by specific implementations. We study quality of solution in terms of number of colors produced by the implementations, memory usage, and execution time. The results in Fig. 4 are normalized with respect to ECL-GC-R. We fixed $\alpha = 4.5$ and we varied \mathcal{P} from 1% up to 15% in our Picasso runs. We observe that the quality of solution achieved by Picasso increases when reducing the palette size (\mathcal{P}). When $\mathcal{P} = 1\%$, Picasso matched the quality of solution of Kokkos-EB and ECL-GC-R (within 5%–15%).

Fig. 4 shows that ECL-GC-R produces the best quality results at the expense of a much longer execution time. In comparison, Picasso’s runs with $\mathcal{P} = 1\%$ completed between $0.44\times$ and $0.60\times$ the time used by ECL-GC-R. Kokkos-EB approach results in the fastest execution time:

between $0.06\times$ and $0.15\times$ the time of ECL-GC-R. However, Kokkos-EB uses between $5.83\times$ and $6.74\times$ the memory used by ECL-GC-R while Picasso had comparable or reduced memory usage ($0.99\times - 0.32\times$) with respect to the same baseline.

D. Parameter Sensitivity

The heatmap in Fig. 5 shows the impact of \mathcal{P} and α on the final colors, conflicting edges, and execution time for a representative input (H4_2D_6311g). The heatmaps show normalized quantities with respect to what is observed for the same input graph. Therefore, as a measure of quality we report the fraction of colors with respect to the number of vertices in the input (lower is better) while we report execution time and the normalized fraction of the total number of conflicting edges processed by the algorithm as measures of the work done by the algorithm (lower is better).

The general trend favors smaller \mathcal{P} and larger α to achieve a lower number of final colors at the cost of extra work. Conversely, larger \mathcal{P} and smaller α lead to lower conflicting edges, and therefore, lower memory requirements and lead to faster execution time. We observed similar trends on all our datasets. These observations led us to design the approach in § VI to jointly minimize work and the size of the coloring by controlling \mathcal{P} and α .

VIII. CONCLUSIONS AND FUTURE WORK

We demonstrated the memory efficiency of Picasso using a large set of inputs and compared its performance with state-of-the-art approaches for graph coloring. In the realm of quantum computing, the introduction of Picasso marks a notable advancement. It is, quite plausibly, the inaugural scalable graph algorithm and implementation tailored for Hamiltonian and wave function partitioning that surpasses the capabilities of contemporary state-of-the-art quantum emulators. What amplifies its significance is its versatile nature; the same tool can be adeptly employed in qubit tapering, thereby reducing the effective number of qubits required for a given problem. When synergized with other methodologies, such as single reference guidance [13], Picasso promises the ability to solve systems comprising 100 to 1000 spin orbitals (i.e. qubits). It is within these vast and complex systems that the much-vaunted quantum advantage is believed to manifest, positioning our tool at the forefront of a transformative computational frontier.

Our future work will focus on developing distributed multi-GPU parallel implementations along with new algorithms for predicting \mathcal{P} and α values to enable better trade-offs for quality and performance. We plan to further optimize our algorithm to address inputs from diverse applications with varying degrees of sparsity. We also plan to develop efficient algorithms for clique partitioning that explore applications beyond quantum computing. To the best of our knowledge, this is the first of its kind study for computing coloring of dense graphs on limited memory accelerator platforms and believe that it will enable several applications that critically depend on the computation of clique partitioning and graph coloring,

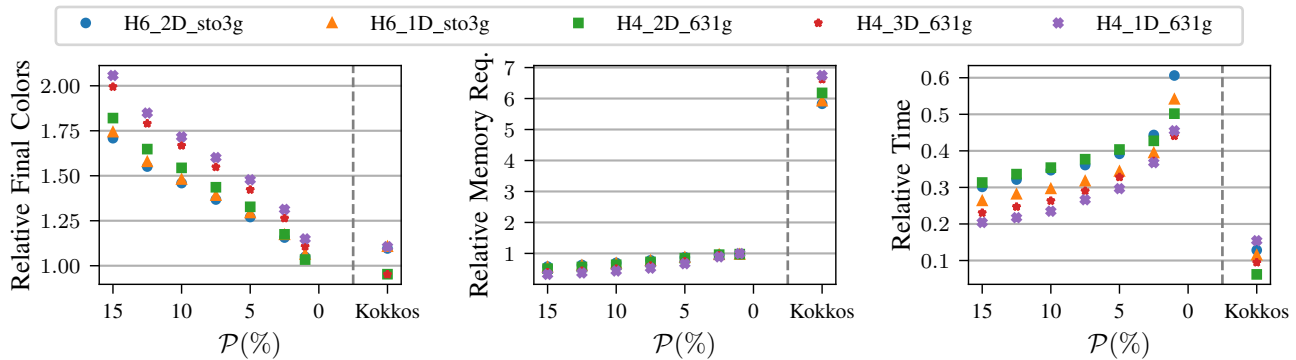


Fig. 4: Performance comparison of Picasso and Kokkos-EB on the small datasets relative to ECL-GC-R execution time. For Picasso runs, \mathcal{P} is varied and $\alpha = 4.5$

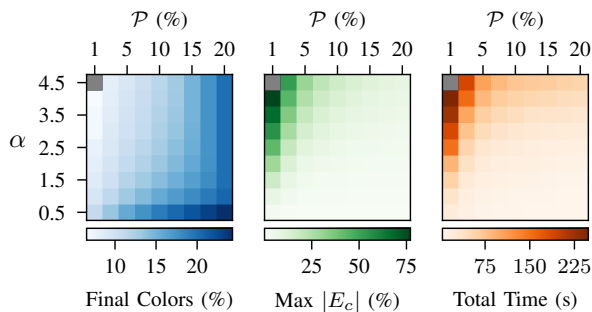


Fig. 5: Impact of \mathcal{P} and α using H4_2D_6311g on final colors, number of conflicting edges and runtime for different inputs (lighter color is better).

as well as enable the development of memory-efficient graph algorithms.

ACKNOWLEDGEMENT

The research is supported by the Laboratory Funded Research and Development at the Pacific Northwest National Laboratory (PNNL), the U.S. DOE Exascale Computing Project’s (ECP) (17-SC-20-SC) ExaGraph codesign center at PNNL, and NSF awards to North Carolina State University.

REFERENCES

- [1] R. R. Lewis, *A Guide to Graph Colouring: Algorithms and Applications*, 1st. Springer Publishing Company, Incorporated, 2015, ISBN: 3319257285.
- [2] A. H. Gebremedhin, F. Manne, and A. Pothen, “What color is your jacobian? graph coloring for computing derivatives,” *SIAM Review*, vol. 47, no. 4, pp. 629–705, 2005. DOI: 10.1137/S0036144504444711.
- [3] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, “ColPack: Software for graph coloring and related problems in scientific computing,” *ACM Trans. Math. Softw.*, vol. 40, no. 1, 2013, ISSN: 0098-3500. DOI: 10.1145/2513109.2513110. [Online]. Available: <https://doi.org/10.1145/2513109.2513110>.

- [4] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, “Parallel graph coloring for manycore architectures,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2016, pp. 892–901.
- [5] G. Alabandi and M. Burtscher, “Improving the speed and quality of parallel graph coloring,” *ACM Trans. Parallel Comput.*, vol. 9, no. 3, 2022, ISSN: 2329-4949. DOI: 10.1145/3543545. [Online]. Available: <https://doi.org/10.1145/3543545>.
- [6] S. Rajamanickam, S. Acer, L. Berger-Vergiat, *et al.*, “Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels,” *arXiv preprint arXiv:2103.11991*, 2021.
- [7] A. Kandala, A. Mezzacapo, K. Temme, *et al.*, “Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets,” *Nature*, vol. 549, pp. 242–246, 2017. DOI: 10.1038/nature23879.
- [8] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, “The theory of variational hybrid quantum-classical algorithms,” *New J. Phys.*, vol. 18, no. 2, p. 023023, 2016. DOI: 10.1088/1367-2630/18/2/023023.
- [9] T.-C. Yen, V. Verteletskyi, and A. F. Izmaylov, “Measuring all compatible operators in one series of single-qubit measurements using unitary transformations,” *J. Chem. Theory Comput.*, vol. 16, no. 4, pp. 2400–2409, 2020. DOI: 10.1021/acs.jctc.0c00008.
- [10] S. Assadi, Y. Chen, and S. Khanna, “Sublinear algorithms for $(\Delta + 1)$ vertex coloring,” in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, SIAM, 2019, pp. 767–786. DOI: 10.1137/1.9781611975482.48. [Online]. Available: <https://doi.org/10.1137/1.9781611975482.48>.
- [11] P. Jordan and E. Wigner, “Über das paulische Äquivalenzverbot,” *Z. Physik*, vol. 47, pp. 631–651, 1928. DOI: 10.1007/BF01331938.
- [12] S. B. Bravyi and A. Y. Kitaev, “Fermionic quantum computation,” *Ann. Phys.*, vol. 298, no. 1, pp. 210–226, 2002. DOI: 10.1006/aphy.2002.6254.

- [13] B. Peng and K. Kowalski, "Mapping renormalized coupled cluster methods to quantum computers through a compact unitary representation of nonunitary operators," *Physical Review Research*, vol. 4, no. 4, p. 043172, 2022.
- [14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979, ISBN: 0-7167-1044-7.
- [15] J. Bhasker and T. Samad, "The clique-partitioning problem," *Computers & Mathematics with Applications*, vol. 22, no. 6, pp. 1–11, 1991.
- [16] J. Schwinger, "Unitary operator bases," *Proc. Nat. Acad. Sci. USA*, vol. 46, no. 4, pp. 570–579, 1960. DOI: 10.1073/pnas.46.4.570.
- [17] A. Klappenecker and M. Rötteler, "Constructions of mutually unbiased bases," in *Finite Fields and Applications*, G. L. Mullen, A. Poli, and H. Stichtenoth, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 137–144.
- [18] J. B. Altepeter, D. F. V. James, and P. G. Kwiat, "4 qubit quantum state tomography," in *Quantum State Estimation*, M. Paris and J. Řeháček, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 113–145. DOI: 10.1007/978-3-540-44481-7_4.
- [19] P. Gokhale, O. Angiuli, Y. Ding, *et al.*, " $O(N^3)$ Measurement cost for variational quantum eigensolver on molecular hamiltonians," *IEEE Trans. Quantum Eng.*, vol. 1, pp. 1–24, 2020. DOI: 10.1109/TQE.2020.3035814.
- [20] A. F. Izmaylov, T.-C. Yen, R. A. Lang, and V. Verteletskyi, "Unitary partitioning approach to the measurement problem in the variational quantum eigensolver method," *Journal of chemical theory and computation*, vol. 16, no. 1, pp. 190–195, 2019.
- [21] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 1036–1053, 1986. DOI: 10.1137/0215074.
- [22] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993. DOI: 10.1137/0914041.
- [23] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 10, pp. 576–594, 2012, ISSN: 0167-8191. DOI: <https://doi.org/10.1016/j.parco.2012.07.001>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819112000592>.
- [24] N. Quang Anh Pham and R. Fan, "Efficient algorithms for graph coloring on GPU," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018, pp. 449–456. DOI: 10.1109/PADSW.2018.8644624.
- [25] I. Bogle, E. G. Boman, K. Devine, S. Rajamanickam, and G. M. Slota, "Distributed memory graph coloring algorithms for multiple GPUs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2020, pp. 54–62. DOI: 10.1109/IA351965.2020.00013.
- [26] Ü. V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1971–1980. DOI: 10.1109/IPDPS.2011.360.
- [27] D. Bozdağ, U. V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, "Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation," *SIAM Journal on Scientific Computing*, vol. 32, no. 4, pp. 2418–2446, 2010. DOI: 10.1137/080732158.
- [28] M. Luby, "Removing randomness in parallel computation without a processor penalty," *Journal of Computer and System Sciences*, vol. 47, no. 2, pp. 250–286, 1993.
- [29] D. Achlioptas and M. S. O. Molloy, "The analysis of a list-coloring algorithm on a random graph," in *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, IEEE Computer Society, 1997, pp. 204–212. DOI: 10.1109/SFCS.1997.646109. [Online]. Available: <https://doi.org/10.1109/SFCS.1997.646109>.
- [30] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005, ISBN: 978-0-521-83540-4. DOI: 10.1017/CBO9780511813603. [Online]. Available: <https://doi.org/10.1017/CBO9780511813603>.
- [31] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. USA: Prentice Hall Press, 2009, ISBN: 0136042597.
- [32] I. Bogle, G. M. Slota, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring algorithms for distributed GPU environments," *Parallel Computing*, vol. 110, p. 102896, 2022.
- [33] G. Alabandi, E. Powers, and M. Burtscher, "Increasing the parallelism of graph coloring via shortcutting," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 262–275.