Enhancing Quantum Circuit Compilation with Modular Floorplanning

Giacomo Lancellotti
Politecnico di Milano-DEIB
Milano, Italy
giacomo.lancellotti@polimi.it

Giovanni Agosta
Politecnico di Milano-DEIB
Milano, Italy
giovanni.agosta@polimi.it

Alessandro Barenghi
Politecnico di Milano-DEIB
Milano, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano-DEIB
Milano, Italy
gerardo.pelosi@polimi.it

***Abstract*—Quantum circuit compilation is a multi-stage process that translates a high-level quantum circuit representation into a sequence of quantum gates that can be directly executed by a quantum machine. One of the core steps in quantum compilation is mapping, a procedure that transforms a quantum circuit into a semantically equivalent circuit while inserting additional gates to comply with the architectural constraints of the underlying quantum hardware. Current mapping techniques exhibit significant performance degradation—in terms of circuit depth and gate count—when applied to large-scale circuits containing hundreds of thousands of gates or more. In this work, we propose a novel methodology to enhance quantum compiler capabilities by adapting general principles from classical Electronic Design Automation. We model the quantum hardware as being composed of standard cells, onto which portions of the circuit are mapped using a floorplanning-based technique. This contrasts with current state-of-the-art solutions, which perform mapping at the gate level without considering any form of hierarchical abstraction. We use the ChaCha20 stream cipher as a running example to validate and explain our strategy. We evaluate our methodology using two well-known, industrial-grade quantum compilers: IBM Qiskit and Quantinuum TKET. Compared to these compilers, our approach achieves an average circuit depth reduction of 59.78% and 68.85%, respectively, and a gate count reduction of 4.07% and 6.20%, respectively. These results are obtained across a large set of randomly generated quantum circuit descriptions in the range of millions of gates, with varying input and output connectivity for each block in the circuit.**

***Index Terms*—Quantum compilation, mapping, electronic design automation, floorplanning.**

I. Introduction

Quantum computing has attracted growing interest from both academic and industrial communities due to its potential to solve problems that are otherwise intractable for classical computers. Recent advancements in hardware manufacturing [1] and error correction [2], [3] have significantly accelerated progress toward realizing fault-tolerant quantum machines with computation accuracy comparable to modern classical hardware. However, designing quantum algorithms still requires working at the level of individual qubits, applying one quantum gate at a time. This lack of high-level programming abstractions and standardization—from intermediate representations to classical hardware interfaces–remains a major obstacle to scaling quantum systems to a utility level, where they can effectively solve real-world problem instances [4]. A major bottleneck in the quantum computing pipeline lies in the middleware software stack. Quantum algorithms—typically expressed as high-level circuits—cannot be executed directly on hardware, as each quantum architecture imposes unique constraints that must be addressed during compilation. A compiler—whether classical or quantum—is a software tool that takes a high-level program as input and produces a semantically equivalent version optimized for execution on a specific hardware platform. One of the most critical steps in quantum compilation is *mapping*, which ensures that a quantum algorithm adheres to the physical connectivity constraints of the target hardware.

Even though the mapping step is essential across different quantum technologies—such as spin qubits [5], trapped ions [6], and neutral atoms [7]—the specific methods used to implement it can vary slightly depending on the platform [8]. In this work, the focus is on the mapping step for superconducting hardware [9]. In superconducting Quantum Processing Units (QPUs), physical qubits are typically arranged on a two-dimensional planar surface and connected in structured topologies. To execute a quantum circuit, its virtual qubits must first be mapped onto the physical qubits. Each interaction between virtual qubits in the circuit must also occur between two directly connected physical qubits. Since this connectivity constraint is often not satisfied by the initial mapping, the qubit assignment must be dynamically adjusted during algorithm execution—typically by inserting additional SWAP gates—which leads to a corresponding degradation in overall performance. It has been shown that the quantum mapping problem is NP-complete [10], implying that finding an optimal mapping cannot be achieved efficiently in polynomial time.

Related Works. A wide range of approaches has been developed to tackle the quantum mapping problem. Due to its combinatorial nature, most of these methods rely on suboptimal techniques that aim to balance solution quality and computational performance. Some of the most notable works include modeling the mapping as a temporal planning problem [11], or relying on greedy heuristics guided by dedicated cost functions, such as the A^* algorithm [12] and the SABRE algorithm [13]. Other strategies use token swapping in combination with approximate subgraph isomorphism [14] or Dijkstra-style search [15]. Alternative formulations use Quadratic Unconstrained Binary Optimization (QUBO) for-

mulation [16], or Satisfiability Modulo Theory (SMT) [17]. Additionally, learning-based methods such as reinforcement learning [18] and AI-assisted techniques [19] have also been explored. Notably, all these works address the mapping problem at the level of individual gates, without considering any higher-level gate abstractions in the circuit.

This work. Inspired by Electronic Design Automation (EDA) tools—which can manage billions of transistors in chiplet design through the use of multiple abstractions and simplifications—we investigate how similar techniques and best practices can be adapted to quantum circuit mapping. In the following, we adopt a higher-level perspective by employing block-level mapping, where the circuit is represented as a set of interconnected blocks, each larger than an individual gate. We employ a *standard cell* partitioning of the QPU, assigning to each block a target architecture with the same connectivity as the QPU. Each block is then compiled independently for its target architecture using existing state-of-the-art quantum compilers such as IBM Qiskit and Quantinuum TKET. Finally, all compiled blocks are mapped onto the QPU using a strategy inspired by *floorplanning* in classical chip design, which determines the placement of the blocks in a way that minimizes the overall depth of the final circuit.

Contribution. We propose a scalable and modular approach to quantum circuit mapping, which can be seamlessly integrated into existing quantum compiler toolchains with minimal computational overhead, while outperforming traditional monolithic compilation methods. Our method introduces the concepts of standard cells and floorplanning, widely adopted in EDA, for quantum circuit mapping, demonstrating how these classical design techniques can enhance current mapping strategies. To validate our approach, we compile the ChaCha20 stream cipher onto a grid-based layout suitable for superconducting quantum technologies. We benchmark our floorplanning-based mapping strategy on randomly generated circuits containing up to tens of millions of gates, comparing it against IBM Qiskit and Quantinuum TKET compilers. Our method achieves a circuit depth reduction of 59.19% and a gate count reduction of 4.11% compared to the results obtained using only the IBM Qiskit compilation stack. When compared to TKET, our approach yields a 67.96% reduction in circuit depth and a 6.25% reduction in gate count.

II. Background

In this section, we present the key stages of quantum circuit compilation. We then define the floorplanning in classical EDA and how it can be adapted to quantum circuit mapping.

A. Quantum circuit compilation

A quantum algorithm is typically described using a circuit-level formalism, where virtual qubits are depicted as wires and computation is carried out by applying sequences of quantum gates. This representation abstracts away the complexity of executing the algorithm on actual quantum hardware, as it does not account for factors such as the specific gate set supported by the device or its physical qubit connectivity.

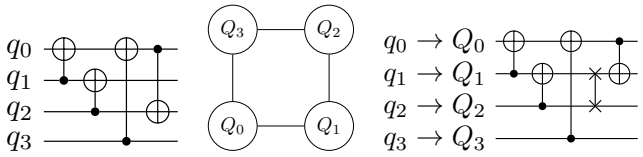

Fig. 1: From left: a quantum circuit where q_i identifies a virtual qubit; in the center, a 4-qubit coupling graph representing the physical QPU connectivity; on the right, the compiled circuit, where a SWAP gate is inserted between qubits q_2 and q_3 to enable the execution of the final CNOT gate.

Indeed, before executing a quantum algorithm on a QPU, the circuit must undergo several transformations. While some compilation steps—such as gate decomposition—can be performed in a hardware-agnostic manner and are therefore common across different quantum architectures, our discussion of lower-level, machine-dependent steps focuses specifically on superconducting quantum architectures. In superconducting QPUs, physical qubits—typically implemented using Josephson junctions—are placed on a two-dimensional surface and interact via resonators. These qubits are often arranged in regular patterns, such as lattices, hexagonal grids, or rectangular grids [20]. The topology of a QPU is typically represented by a coupling graph, where nodes correspond to physical qubits and edges indicate physical connections between them as shown in Fig.1.

Since a quantum circuit can employ gates of arbitrary complexity, the first stage of compilation is *gate decomposition*, where all abstract or unsupported gates are decomposed into a set of native quantum gates supported by the target hardware. These native gates typically belong to a universal gate set—which can approximate any unitary operation to arbitrary precision—such as the Clifford+T gate set [21]. Once decomposition is complete, the compilation process proceeds to *circuit mapping*, which involeves of two key steps: *qubit assignment* and *qubit routing*. In qubit assignment, each virtual qubit is mapped to a physical qubit on the QPU. This mapping can follow several strategies, ranging from simple indexing—assigning qubits based on their order in the circuit and on the hardware—to more compelx approaches that take into account circuit interactions and hardware topology. For instance, qubits that frequently interact in the circuit should ideally be assigned to physically adjacent qubits on the QPU to minimize routing overhead [22]. In the routing stage, two-qubit gates must be executed between physically connected qubits on the QPU. However, due to hardware connectivity constraints, the initial assignment often does not ensure that all interacting qubits are directly connected, except in architectures with all-to-all connectivity. To satisfy these connectivity constraints, SWAP gates—which exchange the quantum states of two qubits—are inserted to bring interacting qubits closer together. An example comparing the input circuit and the compiled circuit is shown in Fig. 1.

Finding an optimal mapping—i.e., minimizing the number of SWAP gates while keeping circuit depth as low as

possible—is an NP-complete problem [10], making optimal solutions impractical for circuits with more than a hundred qubits and gates. Indeed, non-optimal mapping strategies significantly increase the *post-compilation overhead*, which in turn adversely affects the compiled circuit performance in terms of execution time and fidelity. As quantum circuits scale to millions of qubits and gates, new challenges and optimization opportunities arise; addressing these scalability issues is the primary focus of this work.

B. Quantum circuit mapping

A quantum program, or algorithm, in this work, the terms *quantum algorithm*, *program*, and *circuit* are used interchangeably, is typically described using a circuit-level formalism. In this formalism, virtual qubits are represented as wires, and computation is performed through the application of quantum single- and two-qubit gates (such as X, H, S, or CNOT), or more complex multi-controlled gates that involve groups of qubits. Although gates of arbitrary complexity can be expressed using a universal set of quantum gates, with an acceptable approximation error, such as the *universal* Clifford+T gate set [21], the circuit representation abstracts away the complexity of executing the program on actual quantum hardware. This abstraction is similar to how high-level programming languages abstract away low-level assembly instructions. Indeed, before execution on a quantum processing unit (QPU), a quantum program must undergo several transformations. While we focus on superconducting quantum architectures, the methodologies discussed here can be applied to other quantum computing architectures as well. In a superconducting QPU, physical qubits—typically implemented using Josephson junctions—are placed on a two-dimensional surface and interact through resonators. Qubits are often arranged in regular patterns, such as lattices, hexagonal grids, or rectangular grids. The QPU topology is represented through the *coupling graph*, where nodes represent physical qubits and edges represent physical connections. The first stage of compilation is *gate decomposition*, where all abstract or unsupported gates in the circuit are decomposed into a set of native quantum gates supported by the target hardware. Different quantum hardware platforms may have different native gate sets. Once gate decomposition is complete, the compilation process proceeds to *circuit mapping*, which consists of two key steps: *qubit Assignment* and *qubit Routing*. In qubit assignment, each virtual qubit in the circuit is mapped to a physical qubit on the QPU. This assignment can be done in several ways, ranging from simple indexing (assigning qubits based on their order in the circuit and on the hardware) to more sophisticated strategies that consider circuit interactions or hardware topology. For example, qubits that frequently interact in the circuit should ideally be mapped to physically adjacent qubits on the QPU to minimize routing overhead [22]. The initial assignment significantly impacts the efficiency of the subsequent routing stage. After assignment, the process moves to qubit routing. In this stage, two-qubit operations must be executed between physically connected qubits on the QPU.

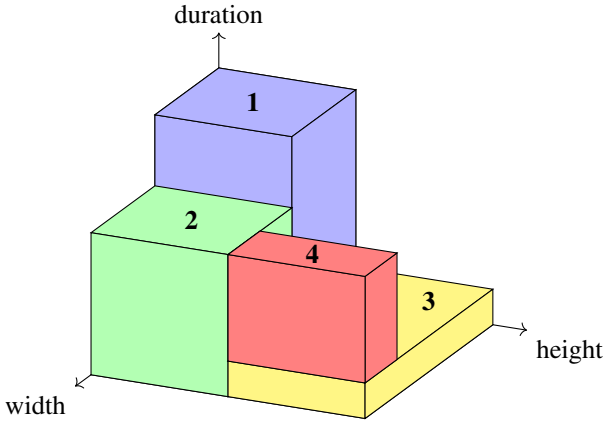

Fig. 2: Temporal floorplanning solution with four modules of varying height, width, and duration. The three-dimensional placement determines both the allocation of blocks on the chip area and the execution schedule: while modules 1,2,3 can be scheduled concurrently, module 4 must execute only after module 3 has completed.

However, due to hardware connectivity constraints, the initial assignment often does not ensure that all interacting qubits are directly connected, except in architectures with all-to-all connectivity. To satisfy these connectivity constraints, SWAP gates—which exchange the quantum states of two qubits—are inserted to bring interacting qubits closer together. An example with the input circuit and the compiled one is shown in Fig.1. Finding an optimal mapping, i.e., minimizing the number of SWAP gates while keeping circuit depth as low as possible, is an NP-complete problem [10]. As demonstrated in various studies, finding an exact solution is impractical for circuits with more than a few dozen qubits and gates. A non-optimal mapping strategy significantly increases the *post-compilation overhead*, as inserting additional gates directly increases both the gate count and circuit depth. This, in turn, negatively impacts the compiled circuit performance in terms of execution time and fidelity. As quantum circuits scale to hundreds of thousand (and more) qubits and gates, new challenges and optimization opportunities emerge, addressing these scalability issues is the primary focus of this work.

C. Floorplanning

In EDA, *floorplanning* is a fundamental step in hierarchical physical design that determines the placement of functional units—such as arithmetic units, memory blocks, and interconnects—on a silicon die. Typically, these units are modeled as geometric shapes with predefined aspect ratios, most commonly as rectangles [23]. Given a limited-area die, which can be modeled by a coordinate set, and a set of modules $M = \{m_1, \ldots, m_n\}$, where each module m_i has width w_i and height h_i, and each module is allowed to rotate freely, the goal is to minimize the total area used for placing all the modules i.e., $min(\sum_{i=1}^{n} (x_i + w_i) \times (y_i + h_i))$, where (x_i, y_i) represents the coordinates of the bottom-left corner

of the rectangle. During placement, additional objective functions can be considered, such as minimizing the wirelength of interconnections between modules. It is well established that floorplanning can be formulated as a *rectangle packing* problem, which is known to be NP-complete [24].

To solve it efficiently, several algorithmic techniques and dedicated data structures have been developed. These approaches allow for effective exploration of the search space while ensuring good solution quality. Some widely adopted representations include the *sequence pair* representation [24], *polish expression*, and B^*-tree [25]. These representations capture the geometric relationships between modules, such as whether a module m_i is to the left or right of another module m_j, and they enable efficient exploration of the solution space, as modifications to their structure typically incur low memory and time complexity. The objective function models cost metrics such as area and wirelength, while constraints enforce geometric and dimensional restrictions, such as non-overlapping conditions. These cost functions are typically embedded in heuristic approaches—such as genetic algorithms [26], simulated annealing [27], ant colony optimization [28], and reinforcement learning [29]—to refine placement solutions. Other methods can apply mathematical programming techniques, such as mixed-integer linear programming (ILP) [30].

Of particular interest for our scope is the so-called *3D* or *temporal floorplanning* [31]. This model extends traditional floorplanning by introducing a temporal dimension, making it suitable for dynamic synthesis scenarios, such as those encountered in Field-Programmable Gate Arrays (FPGAs). FPGAs can be reprogrammed at runtime, allowing the same physical location to host different logic modules at different stages of execution. In this model, circuit units are represented as three-dimensional parallelepipeds, where the additional dimension encodes their active execution interval t_i, i.e., their duration. The placement must ensure that the scheduling of modules respects data dependencies. For example, if a module m_i depends on the output of module m_j, then m_i must be scheduled after m_j. An illustration of temporal floorplanning solution is provided in Fig. 2. If the data flow of an application is known in advance, this information can be integrated into the floorplanning solver to further optimize both placement and scheduling. In our work, we extend this temporal floorplanning model to the mapping of quantum circuits. ¿¿¿¿¿¿¿ 64cb7c444d66db1d9ffbee3c7bf3f74cfc2f5a19

III. Floorplanning Based Mapping

In this section, we clarify our definitions of logic blocks and standard cells in the context of quantum circuits. We then present our floorplanning based mapping algorithm, using the quantum implementation of the ChaCha20 stream cipher as a demonstrative case study.

A. Modular Standard Cell architecture

¡¡¡¡¡¡¡ HEAD In this section, we first describe the preprocessing stage of our algorithm, clarifying our definitions of logic blocks and standard cells. We then present our

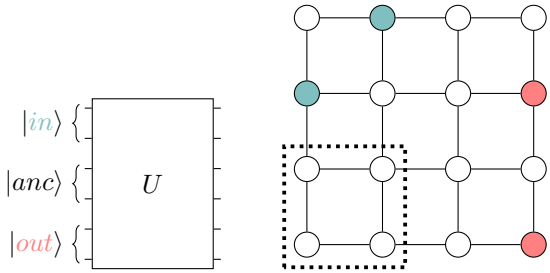

Fig. 3: A generic quantum circuit U with an input register $|in\rangle$ and output register $|out\rangle$, each of size 2, along with an ancillary register of size 12. On the left, the standard cell approximation of the target architecture is shown using a 2×2 standard cell (highlighted by a dotted line).

floorplanning-mapping algorithm along with its pseudocode. Finally, we illustrate the key concepts of our approach using the quantum implementation of the (widely deployed) ChaCha20 stream cipher as a relevant use case of a circuit composed by clear arithmetic sub-block decomposition.

B. Logic Block and Standard cells

A quantum program —like a classical one— can be viewed at different levels of abstraction. From a high-level perspective, it can be represented through its *call graph*, where each node corresponds to a function and each edge represents a caller-callee relationship. Alternatively, it can be analyzed via its *control flow graph (CFG)*, where each node represents a basic block, which is a sequence of instructions with a single entry and a single exit point, while edges represent control flow transitions between blocks, determined by conditional and unconditional branches in the program; CFG are widely used in compiler optimization and static analysis. A similar hierarchy of abstraction exists also in classical hardware design. For example, a digital circuit can be described at the *Register Transfer Level (RTL)* or using more compact representations such as logic modules (e.g., adders and multipliers). In this work, we adopt a similar abstraction to the concept of logic blocks used in hardware design, such as in FPGA architectures, where a logic block corresponds to a high-level function, such as an adder between two quantum registers, and has connector pins for input and output signals. In our approach, logic blocks are the nodes forming the *Data Dependency Graph (DDG)*. The edges represent the input-output relationship between the blocks. The DDG is inherently directed and acyclic. A logic block thus represents the atomic unit of computation in our algorithm. Once the blocks are derived, we can determine the number of physical qubits they will occupy on the QPU during execution For this section and the remainder of this work, we assume a QPU with a lattice (grid) topology consisting of r rows and c columns. Each physical qubit has up to four connections with its neighbors—except for edge qubits, which have three, and corner qubits, which have two. This assumption closely reflects real quantum hardware topologies, such as Google

Willow architecture, and is well-suited for integrating error correction codes [3]. We approximate the qubit usage of each block using a set of standard cells. A standard cell represents the minimum hardware qubit allocation required for a specific function, and its granularity can vary—from a single qubit to the entire QPU, a similar concept have been used for neutral atoms quantum architectures [32]. For clarity, we assume that a single standard cell consists of four qubits arranged in a square lattice. However, the shape and aspect ratio of the standard cell are tunable parameters in our algorithm, allowing flexibility in mapping strategies. Naturally, the standard cell topology is constrained by the underlying QPU architecture. For each logic block, we allocate a sufficient number of standard cells to ensure that enough qubits are available to accommodate the entire block. This may result in unused pre-allocated qubits if the number of qubits required by the logic block is smaller than the total number of qubits provided by the standard cells composing it. This arises from the fact that standard cells are atomic and must be allocated in discrete multiples. For this work, we assume that each block is mapped using standard cells with a balanced rectangular aspect ratio between width and height. An example of standard cell approximation is depicted in Fig. 3. Once the required standard cells are allocated, we can reserve qubits for specific functions, such as the inputs, outputs, or ancillary qubits that are released after the block completion. To keep track of the function assigned to each qubit, they are allocated to a block and numbered in a row-wise fashion, starting from the bottom-left corner. We conclude the preprocessing phase by compiling each block independently, a process that can be executed in parallel with the other blocks. Any existing quantum compiler can be used for this task, compiling the block using its standard cell approximation as the physical architecture to target. ======= In this work, we consider quantum algorithms in the form of circuits. Since memory elements are not present in this type of representation—and the feasibility of *Quantum Random-Access Memory* (QRAM) remains an open question—these circuits can, without loss of generality, be regarded as combinatorial networks. In classical EDA, combinatorial networks can be interpreted at various levels of abstraction. For instance, they can be described using a netlist, where each logic port is explicitly represented, or through more compact structural representations with functional modules (e.g., adders or multipliers). In our approach, we adopt a structural, block-level description, where blocks are defined as groups of gates within the input quantum circuit. From this representation, we derive the *Data Dependency Graph* (DDG), where nodes correspond to logic blocks and edges represent the input-output dependencies between them. The DDG is inherently a *Directed Acyclic Graph* (DAG). We do not focus here on how blocks are derived—such as via quantum circuit partitioning techniques—but instead adopt a high-level abstraction inspired by classical hardware design. In this view, a logic block may correspond to a specific function, such as an adder operating on two quantum registers.

For this section and throughout the remainder of this work, we assume a QPU model based on a lattice topology with r rows and c columns. Each physical qubit is connected to up to four neighboring qubits, except for edge qubits, which have at most three connections, and corner qubits, which have at most two. This architecture closely mirrors the topology of real quantum hardware, such as Google Willow processor, and is particularly well-suited for the integration of quantum error correction codes [3].

Once the logic blocks are derived, we assign to each a target architecture defined through a standard cell approximation. A *standard cell* represents the minimal physical qubit allocation on the QPU. Similar concepts have already been explored for neutral atom quantum architectures [32]. In our approach, both the shape and aspect ratio of the standard cell are tunable parameters, providing flexibility in the mapping strategy. Naturally, the topology of the standard cell is constrained by the underlying QPU architecture. To this end, we consider a standard cell composed of four qubits arranged in a 2×2 square lattice, an example of which is shown in Fig. 3. This may result in unused, pre-allocated qubits when the number of qubits required by a logic block is smaller than the total number of qubits provided by the standard cells composing it. This stems from the fact that standard cells are atomic units and must be allocated in discrete multiples. However, having more qubits than strictly necessary can also be beneficial for block internal routing [33]. We assume that each block is mapped using standard cells with a balanced rectangular aspect ratio between width and height. An example of a standard cell approximation is shown in Fig. 3. Additionally, qubits can be reserved for specific roles, such as inputs, outputs, or ancillary qubits that are released once the execution of the block is complete.

We conclude the preprocessing phase by compiling each block independently, a process that can be executed in parallel across all blocks. Any existing quantum compiler can be employed for this task, using the standard cell approximation as the target physical architecture for compilation. ¿¿¿¿¿¿¿ 64cb7c444d66db1d9ffbee3c7bf3f74cfc2f5a19

C. Floorplanning Mapping algorithm

Once each logic block is compiled, we proceed to schedule the execution of all blocks. We define a *layer* as a group of blocks that can be executed concurrently—that is, they do not share any data dependency path in the DDG. Any scheduling policy can be adopted; here, we employ an *As Soon As Possible* (ASAP) scheduler to partition the entire circuit into layers. The execution time of each block is associated with the depth of its compiled circuit. Each layer then defines the set of blocks to be mapped onto the QPU through floorplanning. To ensure that data dependencies between blocks are respected, each qubit storing the output of a block must be connected to the qubits where a dependent block (i.e., a successor node in the DDG) expects its inputs. To achieve this, we employ a series of SWAP gates—referred to as SWAP *chains*—to connect output and input qubits in a process known as *data routing*. Data routing is performed after each layer has been placed to ensure

that output values are correctly positioned for the subsequent layer. Our algorithm is designed to minimize the impact on the circuit depth introduced by the data routing.

¡¡¡¡¡¡¡ HEAD Each layer defines the space for floorplanning. The goal is to map each block in the layers onto the QPU while respecting data dependencies between blocks. Each qubit storing the output of a block must be connected to the qubits where a dependent block (a descendant node in the DDG) expects its inputs. To achieve this, we use SWAP gate chains to connect input-output qubits, a process we refer to as *data routing*. Data routing must be performed after a layer has been placed to ensure that output values are correctly positioned for the subsequent layers. Since the routing step introduces overhead in the final compiled circuit, our algorithm aims to minimize the length of the SWAP gate chains and their impact on the overall circuit depth. We discuss the ideas behind the main procedures, referring to the pseudocode shown in Alg. 1.

The full circuit is processed sequentially, starting from the first layer, by the main procedure FORWARDMAPPING. This procedure takes as input details about the QPU topology $\mathcal{Q}$, including the number of qubits, its width, and height, along with the stack of layers $\mathcal{L}$. Each layer is a set of logic block objects, denoted as $l_i = \{b_0, \ldots, b_n\}$. Each block object contains attributes describing its dimensions, accessible via $width(b_i)$ and $height(b_i)$. Additionally, each block specifies the locations of the qubits where it expects its inputs and where it computes its output values. We refer to the placement of a block b_i using the tuple $\langle b_i, \langle x_o, y_o\rangle, W\rangle$, where $\langle x_o, y_o\rangle$ represents the coordinates of a physical qubit of the QPU where the bottom-left qubit of the block is mapped. The QPU is indexed starting from the bottom-left qubit, which has coordinates $\langle 0, 0\rangle$. We refer to $\langle x_o, y_o\rangle$ as the *origin coordinate*, and W is a set of coordinate paths. Each path connects a qubit storing one of the output values of the block to a qubit where the corresponding input of its descendant block is mapped. We also use a supporting list data structure, bk, which contains the set of coordinates of quibts that cannot be chosen for assignment in the layer l_i. This is necessary because some qubits retain values across multiple layers. For example, consider a block that writes its output value to qubit q_j in layer l_0, and its value is later used by a descendant in layer l_i. In this case, q_j remains unavailable for all blocks in the intermediate layers $\{l_1, \ldots, l_{i-1}\}$. ======= The full circuit is processed sequentially, starting from the first layer, by the main procedure FORWARDMAPPING, whose pseudocode and main supporting procedures are presented in Alg. 1. Each layer is a set of logic blocks b_i, each characterized by its height $height(b)$, width $width(b)$, and the location of its input and output qubits on the target architecture. We denote a layer as $l_i = \{b_0, \ldots, b_n\}$, and maintain the list of layers in a FIFO stack $\mathcal{L}$. We represent the placement of a block with the tuple $\langle b_i, \langle x_o, y_o\rangle, W\rangle$, where $\langle x_o, y_o\rangle$ indicates the coordinates of the physical qubit on the QPU corresponding to the bottom-left qubit of the block. We refer to this as the *origin coordinate*. The component W represents a set of coordinate paths used for SWAP chains associated with the block. The QPU grid is indexed starting from the bottom-left qubit, which has coordinates $\langle 0, 0\rangle$. Since with ASAP scheduling some qubits may retain values across multiple layers—thus becoming *blocked coordinates* for a given layer l_i—we keep track of these using a supporting data structure denoted as bk. ¿¿¿¿¿¿¿ 64cb7c444d66db1d9ffbee3c7bf3f74cfc2f5a19 The set bk can be efficiently determined using live-variable analysis, given the schedule of the blocks.

We initialize the mapping for the first layer using a dedicated strategy, implemented by the function INITMAPPING. Our approach follows a compact placement policy, commonly employed in floorplanning applications, where blocks are placed in descending order of height, starting from the bottom-left qubit of the QPU and proceeding in a row-wise manner. If a row—whose height is determined by the first block placed—cannot accommodate additional blocks, the placement continues on the next row above. The complexity of the initial mapping procedure is $\mathcal{O}(n)$, where n denotes the number of blocks in the first layer.

The blocks within a layer are processed in order of decreasing constraint, where constraint is defined as the number of descendant relations—equivalently, the out-degree of the block in the DDG—and are mapped using the procedure PLACEBLOCK. To minimize the length of the SWAP chains, we use the concept of the geometric *centroid*, computed by the procedure COMPUTECENTROID. For discrete points on a 2D surface, the centroid is defined as the average of all point coordinates: $\langle x_c, y_c\rangle = \left\langle \frac{1}{n}\sum_{i=1}^{n} x_i, \frac{1}{n}\sum_{i=1}^{n} y_i \right\rangle$, where the summation is taken over the coordinates of the qubits to which the outputs of the predecessor blocks are mapped. The resulting coordinates $\langle x_c, y_c\rangle$ correspond to the geometric center of the block to be placed, ensuring that we identify a location that minimizes the cumulative path lengths from the block input qubits to the outputs of its predecessors. The origin coordinates for the placement are then computed as: $\langle x_o, y_o\rangle = \langle x_c - width(b)/2, y_c - height(b)/2\rangle$.

ISVALIDPOSITION checks whether the $\langle x_o, y_o\rangle$ is a valid mapping for the block. This involves three main checks. The first check ensures that the origin coordinates of b reside within the QPU borders, i.e., $(0 \leq x_o \wedge x_o + width(b) \leq width(\mathcal{Q})) \wedge (0 \leq y_o \wedge y_o + height(b) \leq height(\mathcal{Q}))$. The second check ensures that the current block does not overlap with any other block in the same layer. Specifically, given the origin coordinates $\langle x_p, y_p\rangle$ of a previously placed block b_p, the condition is: $(x_o + width(b) \leq x_p \vee x_o \geq x_p + width(b_p)) \vee (y_o + height(b) \leq y_p \vee y_o \geq y_p + height(b_p))$. Finally, the third check ensures that the block does not overlap with blocked coordinates: $\neg((x_o \leq x_{bk} \leq x_o + width(b)) \wedge (y_o \leq y_{bk} \leq y_o + height(b)))$. We need to perform $n(n-1)/2$ overlap checks between already placed blocks in the same layer, resulting in an overall complexity of $\mathcal{O}(n^2)$.

If $\langle x_o, y_o\rangle$ results in an invalid position, we employ a greedy strategy that explores neighboring positions around the initially computed centroid. The procedure GREEDYPOSITIONSEARCH, whose pseudocode is reported in Alg. 2, be-

Algorithm 1: Floorplanning Mapping

```
// map the full quantum circuit
Procedure ForwardMapping
    Input: L: layers stack l_i={b_0, ..., b_n}
           Q: QPU topology information
    Output: P: sequence of placement ⟨b, ⟨x_o, y_o⟩, W⟩, where b
            is a block, ⟨x_o, y_o⟩ are origin qubit coordinates, and
            W is a set of paths connecting the outputs to
            descendant blocks w_i = {{⟨x_0, y_0⟩, ..., ⟨x_n, y_n⟩}}
    Data: bk: stack of sets of blocked coordinates for each layer
          bk_i = {⟨x_0, y_0⟩, ..., ⟨x_n, y_n⟩}
    l ← dequeue(L) // first layer of the circuit
    P ← InitMapping(l, Q)
    while (top(L) ≠ ⊥) do
        l ← dequeue(L)
        bk ← dequeue(bk) // blocked coordinate
        foreach b ∈ l do
            ⟨b, ⟨x_o, y_o⟩, W⟩ ← PlaceBlock(b, Q, P, bk)
            P ← P ∪ ⟨b, ⟨x_o, y_o⟩, W⟩
    return P // all blocks placed
// find the best placement for a block
Procedure PlaceBlock
    Input: b: current block to place
           Q, P, bk
    Output: ⟨b, ⟨x_o, y_o⟩, W⟩: block placement
    A ← FindAncestors(b, P) // coordinates set of
      ancestors outputs
    ⟨x_c, y_c⟩ ← ComputeCentroid(A)
    ⟨x_o, y_o⟩ ← ⟨x_c − width(b)/2, y_c − height(b)/2⟩
    if !IsValidPosition(b, ⟨x_o, y_o⟩, P, bk, Q) then
        C ← GreedyPositionSearch(b, ⟨x_c, y_c⟩, Q)
        foreach ⟨x_c, y_c⟩ ∈ C do
            ⟨x_o, y_o⟩ ← ⟨x_c − width(b)/2, y_c − height(b)/2⟩
            if IsValidPosition(b, ⟨x_o, y_o⟩, P, bk, Q) then
                break
    W ← ComputeConnection(b, ⟨x_o, y_o⟩, A)
    return ⟨b, ⟨x_o, y_o⟩, W⟩
// check if the position of the block is valid
Procedure IsValidPosition
    Input: b, ⟨x_o, y_o⟩: current block to place
           Q, P, bk
    Output: 1: if the position of the block is valid, otherwise 0
    // placement inside QPU
    if !((0 ≤ x_o ∧ x_o + width(b) ≤ width(Q)) ∧ (0 ≤
      y_o ∧ y_o + height(b) ≤ height(Q))) then
        return 0
    // overlap with current layer blocks
    pb ← SameLayerPlaced(b, P)
    foreach ⟨b_p, ⟨x_p, y_p⟩, W_p⟩ ∈ bp do
        if !((x_o + width(b) ≤ x_p ∨ x_o ≥ x_p + width(b_p)) ∨
          (y_o + height(b) ≤ y_p ∨ y_o ≥ y_p + height(b_p))) then
            return 0
    // overlap with blocked coordinates
    foreach ⟨x_bk, y_bk⟩ ∈ bk do
        if (x_o ≤ x_bk ≤ x_o + width(b)) ∧ (y_o ≤ y_bk ≤
          y_o + height(b)) then
            return 0
    return 1
```

Algorithm 2: Greedy-Search for valid positions

```
// search for new positions by incrementally
   expanding the Manhattan radius
Procedure GreedyPositionSearch
    Input: b: current block to place
           ⟨x_c, y_c⟩: coordinates of the centroid
           Q: QPU topology infomation
    Output: C: new set of coordinatea for the centroid
    x_max ← max(width(Q) − x_c − width(b)/2, x_c − width(b)/2)
    y_max ← max(height(Q) − y_c − height(b)/2, y_c − height(b)/2)
    r ← max(x_max, y_max) // maximum search radius
    foreach d_x ∈ (−r, r+1) do
        d_y+ ← r − |d_x| // Manhattan distance
        d_y− ← −(r − |d_x|)
        C ← C ∪ {⟨x_c + d_x, y_c + d_y+⟩, ⟨x_c + d_x, y_c + d_y−⟩}
    return C
```

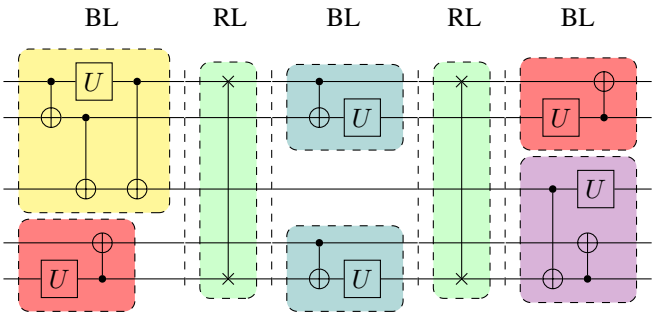

Fig. 4: Example of a compiled circuit with layer-based construction. The output circuit alternates between block layers (BL), and data-routing layers (RL).

gins by examining candidate positions along a circumference centered at $\langle x_c, y_c\rangle$, incrementing the exploration radius by 1 at each iteration. The radius is computed using the *Manhattan distance*, which, for two points $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$ on a 2D surface, is defined as:$d = |x_2 - x_1| + |y_2 - y_1|4$. The Manhattan distance restricts movement to horizontal and vertical directions, aligning better with hardware constraints. This avoids selecting positions far from the centroid that would result in a high cumulative SWAP chain length. In the worst-case, the algorithm may need to evaluate all possible available positions on the QPU, yielding a complexity of $\mathcal{O}(width(\mathcal{Q}) \times height(\mathcal{Q}))$.

Once a valid position for a block is found, we compute all the SWAP chains using the ComputeConnection procedure. To find the shortest chains, we use the Manhattan path between each pair of output-input qubits, which can be computed in constant time, as it only requires coordinate differences. It is possible that a SWAP chain may pass through blocked qubits for that layer or through qubits where other output values are stored. If any of these constraints are violated, we sanitize the chain by swapping the qubits back to their original positions. Since we need to compute a SWAP chain for each block in the layer, the complexity of the procedure is $\mathcal{O}(n)$. The final compiled circuit alternates between block layers and data routing layers whose example is shown in Fig. 4.

We derive the bounds on the complexity of our floorplanning-mapping algorithm. Let N be the total number of blocks, l the number of layers, and n the maximum number of blocks in a layer. In the best-case scenario, the circuit is fully sequential—i.e., each layer contains only one logic block—so $N = l$. In this case, we perform one validity check and one data routing step per layer. Therefore, the lower-

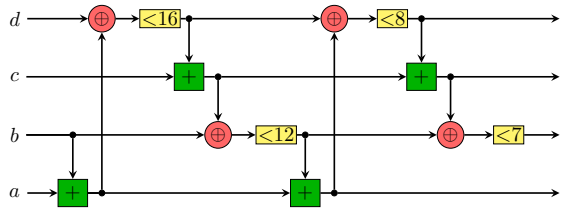

Fig. 5: Circuit diagram of the ChaCha20 quarterround function, where ADD (⊞), XOR (⊕), and SHIFT ($<n$) modules are highlighted in green, red, and yellow, respectively. The computation flows from left to right.

bound complexity is $\Omega(N)+\Omega(N)$. Conversely, in the worst-case scenario, the circuit is fully parallelizable, consisting of a single layer with N blocks. In this case, we need to perform N^2 validity checks (one for each pair of blocks) but no data routing. Thus, the upper-bound complexity is $\mathcal{O}(N^2)$. In the average case, assuming the blocks are uniformly distributed across the layers, we need to perform n^2 validity checks for each of the l layers and compute n input-output connections per layer. This results in an average-case complexity of $\Theta(nN)+\Theta(nl)$.

D. ChaCha20 mapping

We demonstrate our strategy on ChaCha20 cryptographic algorithm [34]. ChaCha20 processes a 512-bit input block, generating an output keystream while operating on 32-bit words. Its core operation, the *quarterround* function, is repeated for a fixed number of rounds, depending on the desired security level (20 rounds in the original implementation). The quarterround function, whose circuit is reported in Fig. 5,consists of four iterations of modular addition (⊞), bitwise XOR (⊕), and cyclic left shifts ($<n$) by 16, 12, 8, and 7 bits. For the quantum version of the quarterround function, we refer to the implementation proposed in [35].

In our example, we map the first round of ChaCha20, which consists of four parallel applications of the quarterround circuit, requiring a total of 16 registers of 32 qubits each. The mapping is executed on a 38×38-qubit QPU with lattice topology. The target architecture for the input registers is a 6×6 lattice layout, the same configuration is used for the SHIFT block. Meanwhile, the ADD and XOR blocks are mapped using 8×8 layout. Operations are scheduled using an ASAP policy, resulting in 13 layers (12 block layers plus one input layer). The QPU occupancy of each layer is shown in Fig. 6, where we display the placement of each block, along with blocked qubits represented as black dots.

In the first layer, the input blocks are mapped in a compact row-wise arrangement. In the second layer, four ADD blocks are placed as close as possible to their respective input blocks from the first layer, following the placement strategy described in Algorithm 1. The qubits blocked in this layer correspond to input values that will be reused in later layers—specifically for the XOR operation in the third layer, the second ADD in the fifth layer, and the second XOR in the sixth layer—as illustrated in Fig. 5. All subsequent layers are mapped using the same heuristic placement strategy, which aims to minimize the cumulative SWAP chain length required to connect block outputs to the inputs of their dependent blocks.

IV. EXPERIMENTAL EVALUATION

¡¡¡¡¡¡¡ HEAD In this section, we analyze and discuss the performance of our floorplanning mapping procedure. To evaluate the effectiveness of our algorithm at different scales, we generate test cases by progressively increasing the number of logic blocks in the program. To avoid biasing performance toward specific circuit structures, we create logic blocks with randomly selected 2-qubit gate interactions while varying both the depth and the number of qubits. One-qubit gates are not of particular interest, as they do not require routing and can always be applied directly. Beyond the internal structure of the logic modules—compiled using the selected mapping algorithm—we primarily focus on the data dependencies between blocks. To this end, we generate random data DDGs, varying key characteristics such as the average out-degree of the nodes. All tests are conducted on a QPU with a 40×40 lattice topology, consisting of 1600 qubits. For each block, we assign a layout with a width-to-height ratio as close to 1:1 as possible, build one standard cells with a 4×4 ratio, ensuring that each logic block has sufficient qubits to store its input and output, along with a random number of ancillary qubits. We assign randomly the indexes of qubits to store the blocks inputs and outputs. All code is implemented using Python 3.10.13, and experiments were conducted on a MacBook Pro with a Quad-Core Intel Core i5 @ 2.3 GHz and 8 GB of RAM, running macOS 14.6.1. To evaluate the effectiveness of our method, we benchmark it against two of the most recognized industry-grade quantum compilers: IBM Qiskit and Quantinuum TKET. Specifically, we used Qiskit version 1.3.2, with the *SabreLayout* as the initial mapping and the *SabreSwap* routing algorithm. For TKET, we used PyTKET version 1.40.0 with the *LexiLabellingMethod* and *LexiRouteRoutingMethod*.

We report the experimental results in Fig. 7. For the comparison, we first compile the circuits as a monolithic block using the two selected quantum compilers, referred to as SABRE and TKET in the figure. We then split up the circuit into blocks, compile them independently with the said compilers, and use our floorplanning algorithm to obtain their final map onto the QPU. These are labeled as SABRE+Floorplanning and TKET+Floorplanning in the figure. In the first-row graphs of Fig. 7, we report the circuit depth and gate count of the original circuit alongside the four compilation strategies. All values are plotted as a function of the number of logic blocks in the circuit. Intuitively, a higher number of logic blocks in the DDG increases circuit complexity—specifically, the number of gates and depth—and inter-block data dependencies. The number of block connections, represented by the average out-degree of the DDG, ranges from 2 for circuits with 10 blocks to 5 for circuits with 100 blocks. We observe that, for both depth and gate count, our floorplanning algorithm used in conjunction with existing compilers results in a lower depth

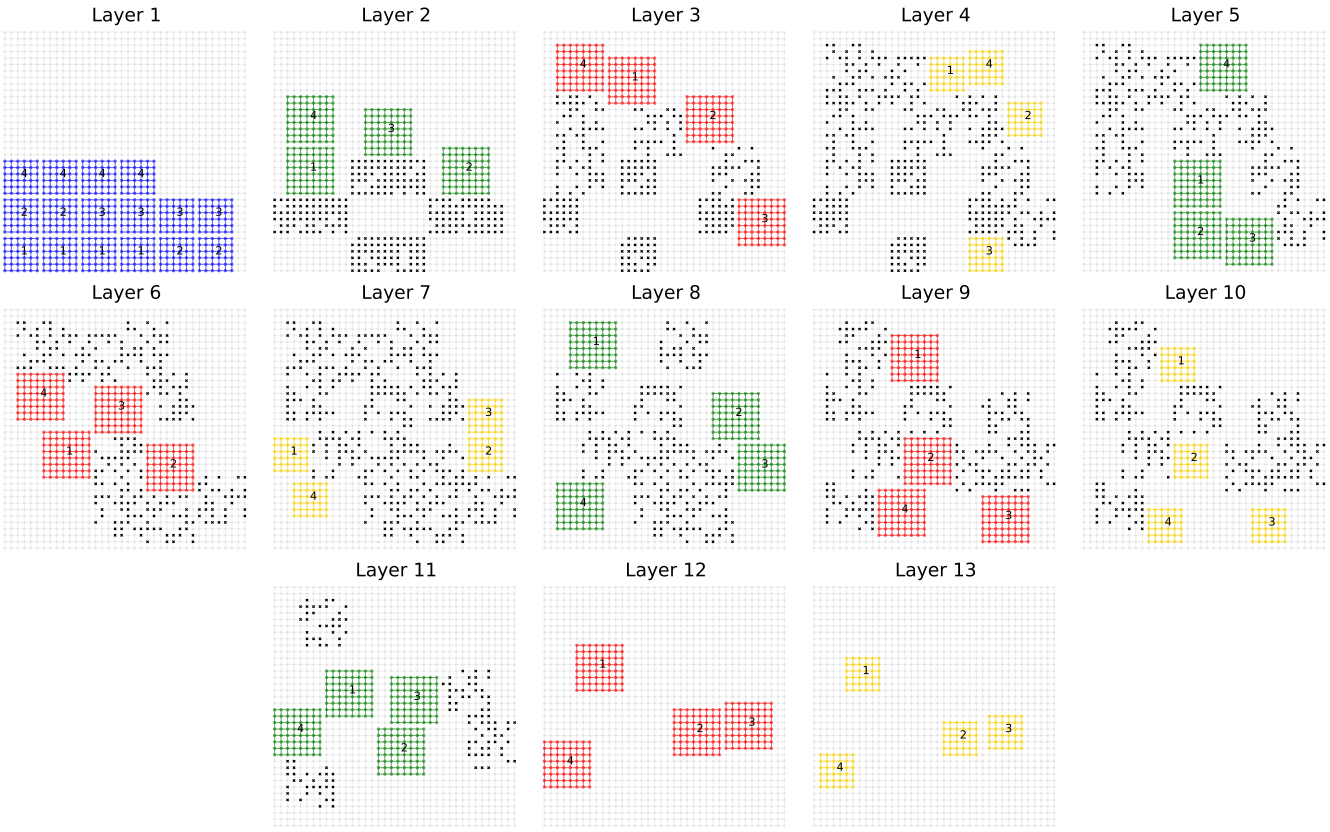

Fig. 6: Mapping of the first round of the ChaCha20 stream cipher onto a 38×38 qubit QPU with lattice topology. The input qubit registers (blue layout) and SHIFT operations (yellow layout) are arranged in 6×6 qubit blocks, while ADD (green layout) and XOR (red layout) operations are mapped onto 8×8 qubit blocks. Execution proceeds using ASAP scheduling, with each layer displayed in increasing order of execution. Each block is labeled from 1 to 4, corresponding to the quarterround function to which it belongs in the first round. Qubits blocked for each layer are indicated as black dots on the QPU. For clarity, SWAP chains used for data routing are not shown.

and gate overhead in the final circuit. For the depth, the different shapes of the curves for SABRE and TKET depend on how well their heuristic approaches handle the selection of the SWAP operations to insert during qubit routing. For SABRE+Floorplanning and TKET+Floorplanning, we observe a similar shape up to constant scaling in depth, meaning that our algorithm found the same mapping, but the internal structure of the blocks has different depths. This is reflected in the TKET curve, which shows a higher depth value than the SABRE curve. Similar considerations can be drawn also for the gate count figure of merit. In the second-row graphs of Fig. 7, we report the depth increment and the gate count increment as a percentage relative to the depth and gate count of the original circuit. In this case, the values are reported as a function of the product of the depth and gate count of the original circuit. In particular, we measure that SABRE introduces an average depth increase of 854.29% compared to the baseline, while TKET increases depth on average by 1324.38%. With our floorplanning approach, the average depth increase is reduced to 283.86% for SABRE+Floorplanning and 343.74% for TKET+Floorplanning. For the total gate count, the overheads are more uniform, with an average increase of 117.41% for SABRE, 159.42% for TKET, 108.57% for SABRE+Floorplanning, and 143.33% for TKET+Floorplanning. This means that our floorplanning algorithm, when used with the SABRE compiler, reduced the depth of the final circuit by 59.19% and the gate count of 4.11%compared to using SABRE alone, and by 67.96% and of 6.25% when used with TKET compared to using TKET alone. This demonstrates how our method, by leveraging knowledge of the data dependencies while abstracting away the internal structure of the blocks, effectively reduces the post-compilation overhead.

Furthermore, our methodology introduces only a negligible compilation time overhead. In the experiments considered, our floorplanning-based mapping has an average runtime of 0.072 s, compared to 3.72 s required for full circuit compilation with SABRE and 165.21 s with TKET. Additionally, our approach enables independent and parallel compilation of each logic block, this allows us to allocate more compilation time to aggressively optimize individual modules and mapping them on the QPU using our algorithm. Finally, unlike other works

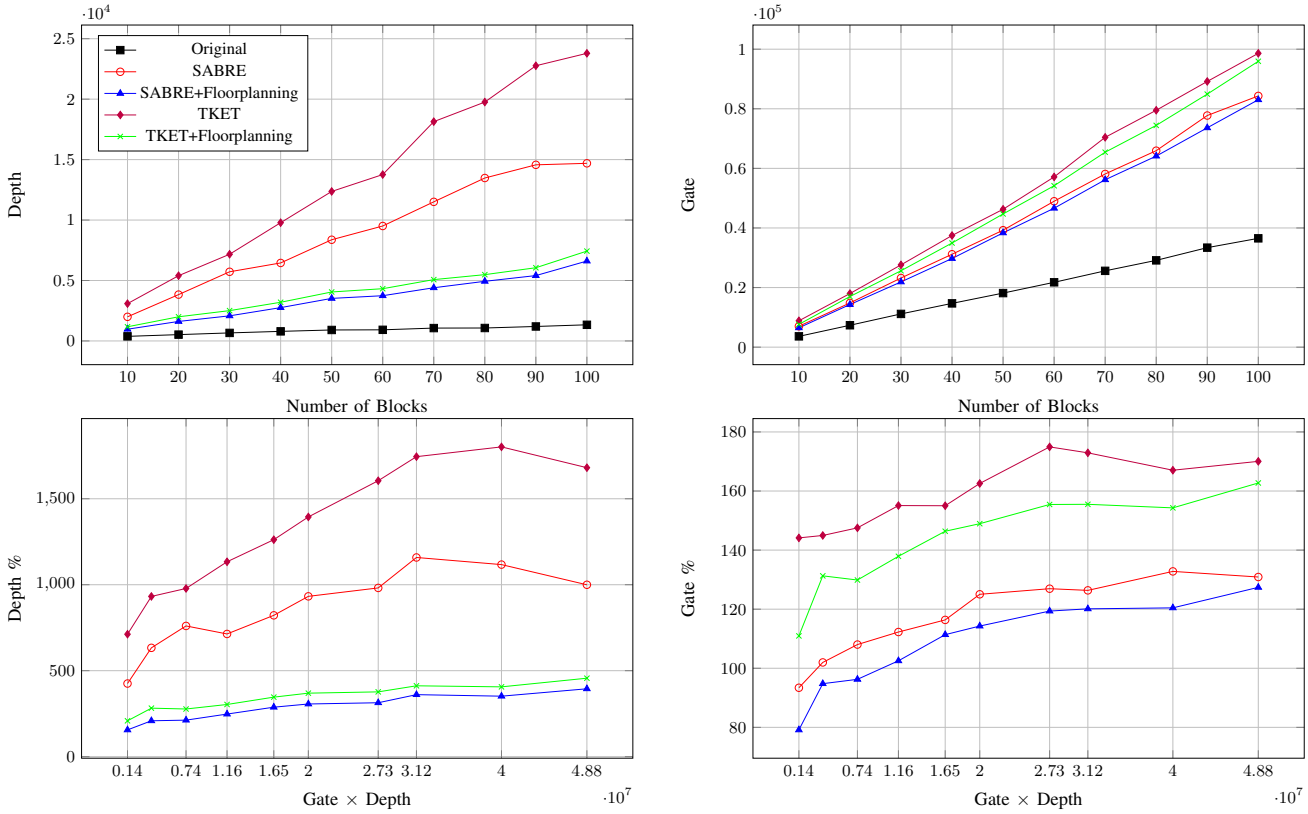

Fig. 7: Top row: circuit depth and gate count for the original circuit, the outputs of IBM Qiskit and Quantinuum TKET compilers, and their combinations with our floorplanning algorithm. All values are shown as a function of the number of logic blocks in the original circuit. Bottom row: percentage overhead in depth and gate count introduced by each compilation strategy, relative to the original circuit. These are plotted as a function of the original circuit gate count × depth.

on qubit allocation and routing algorithms, our solution can be seamlessly integrated into existing compilation toolchains. This offers a significant advantage, as it improves performance without requiring major changes to the current tools. ======== To evaluate the performance of our algorithm, we generate test cases by progressively increasing the number of blocks in the program. To avoid biasing the performance toward specific circuit structures, each block is constructed using randomly selected two-qubit gate interactions, with variations in both circuit depth and qubit count. Single-qubit gates are not of primary interest, as they do not require routing and can always be applied directly. Beyond the internal structure of each block, we primarily focus on their data dependencies. To this end, we generate random DDGs, varying key characteristics such as the average out-degree of the nodes. All experiments are conducted on a QPU featuring a 40×40 lattice topology. For each block, we assign a layout composed of 4×4 standard cells, with a width-to-height ratio as close to 1:1 as possible. The qubit indices used to store the block inputs and outputs are assigned randomly on the target architecture.

All code is implemented in Python 3.10.13 and executed on a MacBook Pro equipped with a quad-core Intel Core i5@ 2.3GHz processor and 8GB of RAM, running macOS 14.6.1. We benchmark our solution against two industry-grade quantum compilers: IBM Qiskit and Quantinuum TKET. Specifically, we use Qiskit version 1.3.2 with the *SabreLayout* for initial qubit mapping and the *SabreSwap* algorithm for routing. For TKET, we employ PyTKET version 1.40.0 using the *LexiLabellingMethod* and *LexiRouteRoutingMethod*. Our benchmarking results are presented in Fig. 7.

For the comparison, we first compile the circuits as a monolithic block using the two selected quantum compilers, referred to as *SABRE* and *TKET*. We then compile each block independently with the same compilers, followed by mapping using our floorplanning algorithm, denoted as *SABRE+Floorplanning* and *TKET+Floorplanning*. In the first-row graphs of Fig. 7, we report the circuit depth and gate count of the original circuit alongside the four compilation strategies, as a function of the number of blocks in the circuit. The average out-degree of DDGs ranges from 2 to 5. Intuitively, increasing the number of logic blocks in the DDG increase the circuit complexity—specifically, its gate count, depth, and inter-block data dependencies. We observe that, for both depth and gate count, our floorplanning-enhanced compilation results in reduced overhead compared to the baseline compilers. The differing shapes of the depth and gate count curves for *SABRE* and *TKET* are attributable to how each compiler heuristic manages the routing of intra-block opera-

tions. For *SABRE+Floorplanning* and *TKET+Floorplanning*, we observe similar trends up to a constant scaling factor in both depth and gate count, indicating that our algorithm find equivalent mappings across compilers.

In the second-row graphs of Fig. 7, we report both the depth and gate count increase as a percentage relative to the original circuit, plotted as a function of the product of depth and gate count of the original circuit. *SABRE* introduces an average depth increase of 854.29% over the baseline, while *TKET* results in an even higher average increase of 1324.38%. When incorporating our floorplanning approach, the average depth increase is significantly reduced to 283.86% for *SABRE+Floorplanning* and 343.74% for *TKET+Floorplanning*. For total gate count, the overheads are more comparable: 117.41% for *SABRE*, 159.42% for *TKET*, 108.57% for *SABRE+Floorplanning*, and 143.33% for *TKET+Floorplanning*. This corresponds to a depth reduction of 59.78% and a gate count reduction of 4.07% when using floorplanning with *SABRE*, and a reduction of 68.85% in depth and 6.20% in gate count when using it with *TKET*. These results demonstrate that our floorplanning algorithm, by leveraging data dependency information while abstracting away the internal structure of each block, effectively reduces post-compilation overhead in both circuit depth and gate count.

Furthermore, our approach introduces only a negligible overhead in compilation time. In the considered experiments, it has an average runtime of 0.072s, compared to 3.72s for full circuit compilation with *SABRE* and 165.21s with *TKET*. In addition, our method enables independent and parallel compilation of each block, further reducing the total compilation time to that of the largest individual block. Finally, our solution can be seamlessly integrated into existing compilation toolchains, as demonstrated by its compatibility with state-of-the-art compilers, without requiring major modifications to existing tools or workflows. ¿¿¿¿¿¿¿ 64cb7c444d66db1d9ffbee3c7bf3f74cfc2f5a19

V. Concluding Remarks

In this work, we proposed a quantum compilation procedure based on a modular standard-cell circuit construction combined with a floorplanning mapping algorithm. Our approach can be seamlessly integrated into existing quantum compiler toolchains, such as IBM Qiskit and Quantinuum TKET, and leads to a significant reduction in both circuit depth and gate count in the final compiled quantum circuit. Additionally, we provide experimental evidence that the improvements of our algorithm increase as the circuit to compile grows in terms of quantum gate depth and the number of qubits used. In future developments, we aim to extend our algorithm to support a broader range of QPU topologies and explore how additional compile-time information can further enhance the qubit mapping process.

Acknowledgement

The activity was partially funded by the *Italian Centro Nazionale di Ricerca in HPC, Big Data e Quantum computing - SPOKE 10.*

References

[1] IBM Institute for Business Value, "The Quantum Decade," 2023.
[2] S. Bravyi, A. W. Cross, J. M. Gambetta, D. Maslov, P. Rall, and T. J. Yoder, "High-threshold and low-overhead fault-tolerant quantum memory," *Nature*, vol. 627, no. 8005, pp. 778–782, Mar. 2024.
[3] Google Quantum AI and Collaborators, "Quantum error correction below the surface code threshold," *Nature*, Dec. 2024.
[4] M. Mohseni, A. Scherer, and K. G. Johnson et al., "How to build a quantum supercomputer: Scaling challenges and opportunities," 2024. [Online]. Available: https://arxiv.org/abs/2411.10406
[5] M. De Michielis, E. Ferraro, E. Prati, L. Hutin, B. Bertrand, E. Charbon, D. J. Ibberson, and M. Fernando Gonzalez-Zalba, "Silicon spin qubits from laboratory to industry," *Journal of Physics D: Applied Physics*, vol. 56, no. 36, p. 363001, jun 2023. [Online]. Available: https://dx.doi.org/10.1088/1361-6463/acd8c7
[6] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, "Trapped-ion quantum computing: Progress and challenges," *Applied Physics Reviews*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:102351008
[7] K. Wintersperger, F. Dommert, T. Ehmer, A. Hoursanov, J. Klepsch, W. Mauerer, G. S. Reuber, T. Strohm, M.-Y. Yin, and S. Luber, "Neutral atom quantum computing hardware: performance and end-user perspective," *EPJ Quantum Technology*, vol. 10, pp. 1–26, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258352683
[8] C. Zhu, X. Wu, Z. Yang, J. Wang, A. Wu, S. Zheng, and X. Wang, "Quantum compiler design for qubit mapping and routing: A cross-architectural survey of superconducting, trapped-ion, and neutral atom systems," 2025. [Online]. Available: https://arxiv.org/abs/2505.16891
[9] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the IBM QX architectures," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 38, no. 7, pp. 1226–1236, 2019. [Online]. Available: https://doi.org/10.1109/TCAD.2018.2846658
[10] M. Y. Siraichi, V. F. dos Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, J. Knoop, M. Schordan, T. Johnson, and M. F. P. O'Boyle, Eds. ACM, 2018, pp. 113–125. [Online]. Available: https://doi.org/10.1145/3168822
[11] D. Venturelli, M. Do, E. Rieffel, and J. Frank, "Compiling quantum circuits to realistic hardware architectures using temporal planners," *2018 Quantum Science and Technology*, vol. 3, 2018. [Online]. Available: https://iopscience.iop.org/article/10.1088/2058-9565/aaa331
[12] A. Zulehner and R. Wille, "Compiling SU(4) quantum circuits to IBM QX architectures," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*. ACM, 2019, pp. 185–190. [Online]. Available: https://doi.org/10.1145/3287624.3287704
[13] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for nisq-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 1001–1014. [Online]. Available: https://doi.org/10.1145/3297858.3304023
[14] M. Y. Siraichi, V. F. dos Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation as a combination of subgraph isomorphism and token swapping," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 120:1–120:29, 2019. [Online]. Available: https://doi.org/10.1145/3360546
[15] C.-Y. Cheng, C.-Y. Yang, Y.-H. Kuo, R.-C. Wang, H.-C. Cheng, and C.-Y. R. Huang, "Robust qubit mapping algorithm via double-source optimal routing on large quantum circuits," *ACM Transactions on Quantum Computing*, vol. 5, no. 3, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3680291
[16] M. Bandic et al., "Mapping quantum circuits to modular architectures with QUBO," in *IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023*. IEEE, 2023, pp. 790–801. [Online]. Available: https://doi.org/10.1109/QCE57702.2023.00094
[17] B. Tan and J. Cong, "Optimal layout synthesis for quantum computing," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415620

[18] C.-Y. Huang, C.-H. Lien, and W.-K. Mak, "Reinforcement learning and dear framework for solving the qubit mapping problem," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549472

[19] M. Baioletti, R. Rasconi, and A. Oddi, "A novel ant colony optimization strategy for the quantum circuit compilation problem," in *Evolutionary Computation in Combinatorial Optimization: 21st European Conference, EvoCOP 2021, Held as Part of EvoStar 2021, Virtual Event, April 7–9, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 1–16. [Online]. Available: https://doi.org/10.1007/978-3-030-72904-2_1

[20] H.-L. Huang, D. Wu, D. Fan, and X. Zhu, "Superconducting quantum computing: a review," *Science China Information Sciences*, vol. 63, no. 8, p. 180501, Jul 2020. [Online]. Available: https://doi.org/10.1007/s11432-020-2881-9

[21] V. Kliuchnikov, D. Maslov, and M. Mosca, "Practical approximation of single-qubit unitaries by single-qubit quantum clifford and T circuits," *IEEE Trans. Computers*, vol. 65, no. 1, pp. 161–172, 2016. [Online]. Available: https://doi.org/10.1109/TC.2015.2409842

[22] G. Nannicini, L. S. Bishop, O. Günlük, and P. Jurcevic, "Optimal qubit assignment and routing via integer programming," *ACM Transactions on Quantum Computing*, vol. 4, no. 1, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3544563

[23] T.-C. Chen and Y.-W. Chang, "Chapter 10 - floorplanning," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds., Boston, 2009, pp. 575–634. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123743640500175

[24] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1518–1524, 1996. [Online]. Available: https://doi.org/10.1109/43.552084

[25] Y. Chang, Y. Chang, G. Wu, and S. Wu, "B*-trees: a new representation for non-slicing floorplans," in *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*. ACM, 2000, pp. 458–463. [Online]. Available: https://doi.org/10.1145/337292.337541

[26] J. Chen and W. Zhu, "A hybrid genetic algorithm for vlsi floorplanning," in *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, vol. 2, 2010, pp. 128–132.

[27] J. Chen, W. Zhu, and M. M. Ali, "A hybrid simulated annealing algorithm for nonslicing vlsi floorplanning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 4, pp. 544–553, 2011.

[28] C. Hoo, K. Jeevan, V. Ganapathy, and H. Ramiah, "Variable-Order Ant System for VLSI multiobjective floorplanning," *Appl. Soft Comput.*, vol. 13, no. 7, pp. 3285–3297, 2013. [Online]. Available: https://doi.org/10.1016/j.asoc.2013.02.011

[29] A. Mirhoseini, A. Goldie, and M. Yazgan et al., "A graph placement methodology for fast chip design," *Nat.*, vol. 594, no. 7862, pp. 207–212, 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-03544-w

[30] S. Sutanthavibul, E. Shragowitz, and J. B. Rosen, "An analytical approach to floorplan design and optimization," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 10, no. 6, pp. 761–769, 1991. [Online]. Available: https://doi.org/10.1109/43.137505

[31] P. Yuh, C. Yang, and Y. Chang, "T-trees: A tree-based representation for temporal and three-dimensional floorplanning," *ACM Trans. Design Autom. Electr. Syst.*, vol. 14, no. 4, pp. 51:1–51:28, 2009. [Online]. Available: https://doi.org/10.1145/1562514.1562519

[32] E. Dobbs, J. Friedman, and A. Paler, "Efficient quantum circuit design with a standard cell approach, with an application to neutral atom quantum computers," *ACM Transactions on Quantum Computing*, vol. 6, no. 1, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3670417

[33] T. Peham, L. Burgholzer, and R. Wille, "On optimal subarchitectures for quantum circuit mapping," *CoRR*, vol. abs/2210.09321, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.09321

[34] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, no. 1. Citeseer, 2008, pp. 3–5.

[35] B. N. Bathe, R. Anand, and S. Dutta, "Evaluation of grover's algorithm toward quantum cryptanalysis on chacha," *Quantum Inf. Process.*, vol. 20, no. 12, p. 394, 2021. [Online]. Available: https://doi.org/10.1007/s11128-021-03322-7