2025 Design, Automation & Test in Europe Conference (DATE 2025)

# Optimizing Qubit Assignment in Modular Quantum Systems via Attention-Based Deep Reinforcement Learning

Enrico Russo
*University of Catania*
Catania, Italy
enrico.russo@phd.unict.it

Maurizio Palesi
*University of Catania*
Catania, Italy
maurizio.palesi@unict.it

Davide Patti
*University of Catania*
Catania, Italy
davide.patti@unict.it

Giuseppe Ascia
*University of Catania*
Catania, Italy
giuseppe.ascia@unict.it

Vincenzo Catania
*University of Catania*
Catania, Italy
vincenzo.catania@unict.it

***Abstract*—Modular, distributed, and multi-core architectures are considered a promising solution for scaling quantum computing systems. Optimising communication is crucial to preserve quantum coherence. The compilation and mapping of quantum circuits should minimise state transfers while adhering to architectural constraints. To address this problem efficiently, we propose a novel approach using Reinforcement Learning (RL) to learn heuristics for a specific multi-core architecture. Our RL agent uses a Transformer encoder and Graph Neural Networks, encoding quantum circuits with self-attention and producing outputs via an attention-based pointer mechanism to match logical qubits with physical cores efficiently. Experimental results show our method outperform the baseline reducing by $28\%$ inter-core communications for random circuits while minimising time-to-solution.**

***Index Terms*—quantum computing, qubit, routing, allocation, attention, transformer, gnn, reinforcement learning, multi-core**

## I. Introduction

Quantum computing promises exponential speedups for certain tasks compared to classical systems [32]. However, real-world applications require a large number of qubits [19], [30], [44], making scalability a crucial challenge. Scaling NISQ (Noisy Intermediate Scale Quantum) systems is difficult, particularly for superconducting implementations, due to low qubit fidelity, cryogenic requirements, dense control electronics, and crosstalk [2], [9], [42]. A monolithic quantum processor with many qubits results in dense wiring, higher crosstalk, and lower yield [46]. A more feasible solution, akin to classical computing, is dividing the processor into smaller cores [22]. This multi-core setup necessitates quantum state transfers between cores, which are costly in terms of fidelity and delays [41].

Quantum circuits consist of gates involving logical qubits, with operations restricted to connected physical qubits. Due to sparse connectivity in multi-core systems, logical qubits must be allocated within the same core during gate execution [34]. Circuit transformations are required to align with hardware constraints, a process called quantum compilation or transpiling [31]. The compilation problem is NP-complete even in single-core systems [7], [45]. Several works have addressed the qubit allocation problem for single-core architectures with optimal and heuristic approaches [21], [26], [27], [29], [31], while multi-core and modular architectures have recently gained attention [3], [4], [12], [14], [35], [37], [52].

This work focuses on qubit allocation in multi-core architectures. We propose a novel attention-based Deep Reinforcement Learning (DRL) technique for combinatorial problems and sequential decision making, which has shown promise in various domains [10], [25]. Our method addresses the multi-core qubit allocation problem using Graph Neural Networks for state representation learning and action masking for feasible solution construction. We demonstrate its effectiveness in mapping random quantum circuits on a 10-core grid interconnection system, comparing it to black-box optimization baselines.

Our contributions include: (a) Insights into the design of autoregressive DRL agents for multi-core quantum compilation. (b) A novel attention-based agent that outputs feasible solutions with deterministic execution time. (c) Problem formulation for comparison against derivative-free optimization algorithms, and a performance comparison of the learned agent against these baselines.

## II. Preliminaries

### A. Quantum Circuits

In the gate model architecture of quantum computers, quantum programs are expressed as a sequence of reversible gates $G$. Quantum programs are also known as quantum circuits and Fig. 1a shows an example. Typically, quantum hardware implements 1-qubit and 2-qubit gates, with the most common 2-qubit gate being the Controlled Not (CNOT) as depicted in Fig. 1a, where CNOT$_{q_1,q_2}$ negates $q_2$ if $q_1$ is in state $|1\rangle$.

To execute a quantum circuit, logical qubits must be mapped onto physical qubits (two-state quantum systems) after transforming the circuit into hardware-compatible gates. The location of the qubits is crucial for 2-qubit gates, as a physical connection between the qubits is required. At the execution step, the logical qubits involved in the same 2-qubit gate must be allocated to two connected physical qubits. In sparsely connected, single-core quantum architectures (Fig.1b), this is achieved by swapping quantum states between connected physical qubits (Fig.1c). However, these additional SWAP gates reduce fidelity and introduce noise.

### B. Modular Quantum Systems

Due to challenges like qubit crosstalk, control electronics, cryogenic devices, and yield requirements in densely integrated

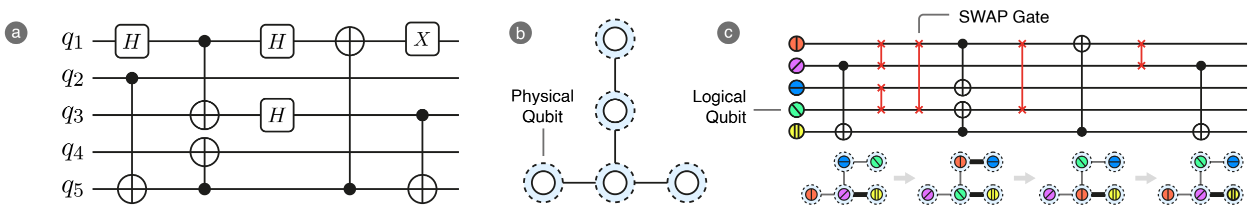

Fig. 1: From left to right: a quantum circuit, a quantum processor (IBM Vigo) physical qubit interconnection graph and the resulting circuit with additional SWAP gates after compilation respecting architectural constraint.

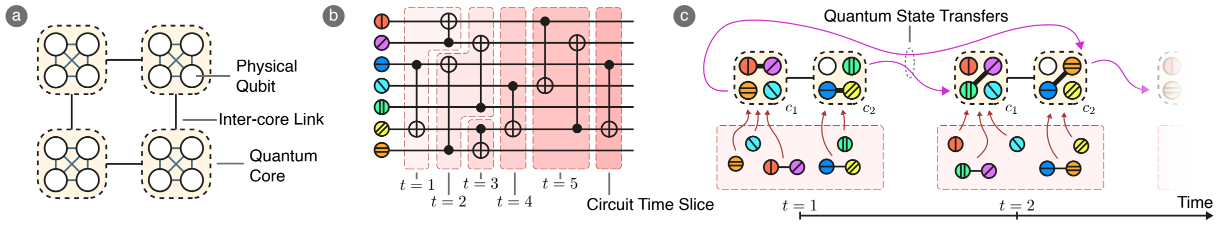

Fig. 2: A Multi-core quantum architecture, a sliced quantum circuit and qubit allocation for the first two circuit slices.

quantum processors, combining multiple quantum processing units into a modular multi-core system is a promising approach to scaling physical qubits for real-world applications [46]. As shown in Fig. 2a, modular quantum systems transfer qubit states between cores via a quantum link that includes both classical and quantum communication [41]. Quantum state teleportation, based on entanglement, or remote gates can be used to transfer states across cores [12]. This work focuses on minimizing inter-core state transfers using teleportation. In this scenario, for a quantum gate to execute, the logical qubits involved must be in the same quantum core. We assume all-to-all connectivity within each core, making specific physical qubit allocation irrelevant [4], and do not assume specific core interconnectivity (Fig. 2a).

### C. Circuit Slicing and Qubit Allocation

A common initial step in qubit allocation optimization, both in single-core [31] and multi-core architectures [3], [4], [15], is to divide the circuit into *slices* that contain gates which can be executed in parallel, meaning the gates do not share any logical qubits, as shown in Fig. 2b. To obtain the partitioning of a quantum circuit in time slices the iterative algorithm in [4] can be considered. In modular architectures, the next step involves allocating logical qubits to quantum cores for each slice, ensuring that: (a) *friends* qubits (those involved in the same gate) are in the same core; (b) the number of logical qubits in each core does not exceed the core's physical qubit capacity[1]. During this process, the goal is to minimize inter-core communications needed to move logical qubits between cores from one slice to the next. In sparsely connected multi-core systems, state transfer costs can vary depending on the hop distance.

[1]This refers to physical *computation* qubits; some qubits may be reserved for state transfer [41], [52].

### D. RL for Combinatorial Optimization

At its essence, RL involves an agent, responsible for taking actions within a given context (state), and an environment, which responds by providing rewards or penalties based on these actions. The primary goal of the agent is to acquire an optimal policy, a strategic mapping of states to actions that maximizes cumulative rewards over time. This human-like trial-and-error learning process has proven remarkably successful in diverse applications [18], [28], [47]. Recent works have demonstrated promising results of RL applied to combinatorial optimization problems [5], [6]. As the solution space of combinatorial grows exponentially, it quickly become infeasible for a neural network to output a feasible solution in one shot, due to the gargantuan action space. For the problem considered in this work, the number of possible decision variable assignments (including non-feasible solutions) is $2^{TQC}$, *e.g.*, about $10^{9030}$ when mapping a $T$ = 30 slices circuit of $Q$ = 100 qubits on a $C$ = 10 cores architecture. To address these issues, similarly to language models [48], autoregressive methods that generates feasible solution step-by-step taking into account constraints have been recently proposed [6], [25].

## III. Methodology

In this section we describe the proposed methodology for RL-based heuristic for the multi-core qubit allocation problem. Initially, we furnish an outline of encoding and decoding procedures in the trained policy. Subsequently, we introduce the components essential to represent and encode the input circuit and the environment state. Finally, we go deeper describing each component and we describe how we make sure that the output solution is a valid solution.

### A. Autoregressive RL for Qubit Allocation

Autoregressive RL has been successfully applied to routing problems like the Travelling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP) [25], using an attention-based model. In TSP, the input is a set of coordinates, and the output is a permutation indicating the traversal order. Pointer Networks [50] help generate valid solutions by selecting the next city while masking visited nodes. Recently, these methods have been extended to domains with specific challenges, such as machine scheduling [11], user-server allocation [8], and hardware design [23]. Applying autoregressive RL to qubit allocation presents its own unique challenges.

**Algorithm 1:** Policy workflow

```
Input  : $X = [G_1, \dots, G_T]$ ;          # seq. of slices
Output : $A = [A_1, \dots, A_T]$ ;          # alloc. ∀ slice
$A \leftarrow []$;
$\mathbf{H}^{(I)} \leftarrow$ InitEmbedding($X$);
$\mathbf{H}^{(S)} \leftarrow$ EncoderBlocks($\mathbf{H}^{(I)}$);
$\mathbf{H}^{(X)} \leftarrow$ AveragePooling($\mathbf{H}^{(S)}$);
for $t \leftarrow 1$ to $T$ do                # for each circuit slice
    $A_t \leftarrow []$;
    $\mathbf{H}_t^{(C)} \leftarrow$ SnapshotEncoder($A_{t-1}$);
    for $q \leftarrow 1$ to $Q$ do            # for each logical qubit
        context $\leftarrow$ concat($\mathbf{H}^{(X)}$, $\mathbf{H}_t^{(S)}$, $\mathbf{E}_q^{(Q)}$);
        $\mathbf{f} \leftarrow$ current free capacities;
        $\mathbf{d} \leftarrow$ distance from $q$'s core in $t-1$;
        $\mathbf{G}_{t,q}^{(C)} \leftarrow$ DynamicEmbedding($\mathbf{H}^{(C)}$, $\mathbf{f}$, $\mathbf{d}$);
        $a \leftarrow$ MaskedPointer(context, $\mathbf{G}^{(C)}$);
        $A_t$.append($a$);
    $A$.append($A_t$);
```

The workflow of the trained policy, which takes an input quantum circuit and outputs a valid solution, is summarized in Algorithm 1. The policy receives circuit slices as input, and each slice $t$ is encoded by `InitEmbedding` to generate embeddings $\mathbf{H}_t^{(I)} \in \mathbb{R}^{d_E}$, where $d_E$ is the embedding size. Similar to [25], transformer `EncoderBlocks` capture relations between all slices in the circuit. The decoding process follows, where the policy sequentially assigns cores for each logical qubit in each slice, resulting in $T \cdot Q$ decoding steps (number of slices × logical qubits). The context for decisions includes three elements: the circuit representation $\mathbf{H}^{(X)}$, slice representation $\mathbf{H}_t^{(S)}$, and qubit representation $\mathbf{E}_q^{(Q)}$. The circuit representation remains constant, the slice representation stays the same for $Q$ steps, and the qubit representation changes at each decoding step.

The action space at each step is the set of possible cores for each qubit. Unlike single-encoder models in routing problems, we use a `SnapshotEncoder` to capture qubit allocations in previous slices. For each core $c$, an embedding $\mathbf{H}_c^{(C)}$ is generated, which includes information about the qubits allocated and core relationships. Additional dynamic data, such as core capacity and transfer costs from previous slices, are incorporated at each step. A masked attention-based pointer mechanism selects the core for each qubit, ensuring valid allocations and feasible solutions for the next steps. After decoding, the solution assigns cores for all qubits in each slice while satisfying problem constraints.

## *B. Circuit Slice Encoder*

The first step of the autoregressive policy is obtaining a learned representation of the input circuit. In traditional RL

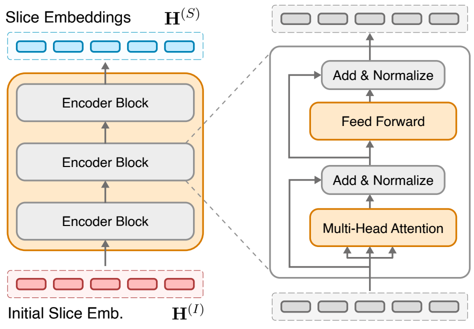

Fig. 3: Attention-based circuit slice encoder.

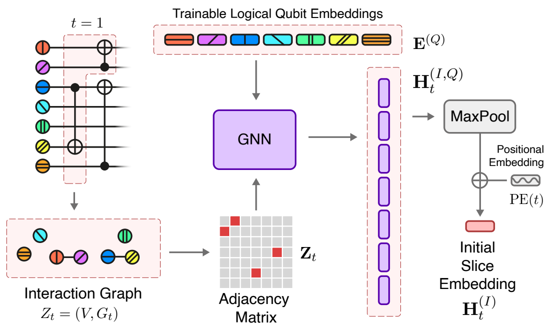

Fig. 4: Initial embedding of the time slice $t = 1$ through a GNN layer.

problems modeled as a Markov Decision Process (MDP), the agent makes decisions based on the current state, derived from past states and actions. In autoregressive combinatorial optimization, however, a global representation of the input is built first, followed by sequential decisions focusing on specific contexts. Our encoder constructs a slice representation incorporating global circuit information. Fig. 3 shows the circuit slice encoder, using attention-based encoder blocks from [48].

*1) `InitEmbedding`:* In this regard, the challenge specific to the qubit allocation problem is that we first need to represent the input of the transformer encoder as a set of initial embeddings $\mathbf{H}_t^{(I)} \in \mathbb{R}^{d_E}$ for each slice $t$ where $d_E$ is the size of such embedding. The output of the slicing algorithm [4], described previously, is the sequence of the $T$ slices we need to embed. Each slice $G_t$ consist in a set of logical qubit pairs representing the gates contained in the slice. We can consider $G_t$ as the edge set of a undirected disconnected graph $Z_t = (V, G_t)$ representing the interaction between logical qubits $V$ given by gates [4]. Because of this, we decide to embed each slice employing a Graph Neural Network (GNN), as shown in Fig. 4. In particular we employ the convolutional graph operator introduced in [24]. In our case, for each slice $t$, we obtain a qubit-wise representation $\mathbf{H}_t^{(I,Q)} \in \mathbb{R}^{Q \times d_E}$ as follows:

$$\mathbf{H}_t^{(I,Q)} = \tilde{\mathbf{D}}_t^{-\frac{1}{2}} \tilde{\mathbf{Z}}_t \tilde{\mathbf{D}}_t^{-\frac{1}{2}} \mathbf{E}^{(Q)} \mathbf{W}^{(I)}, \tag{1}$$

where $\mathbf{W}^{(I)}$ is a trainable weight matrix, $\mathbf{E}^{(Q)} \in \mathbb{R}^{Q \times d_E}$ are trainable logical qubit embeddings, $\tilde{\mathbf{Z}}_t \in \{0,1\}^{Q \times Q}$ is the adjacency matrix of the interaction graph $Z_t$ with added self-

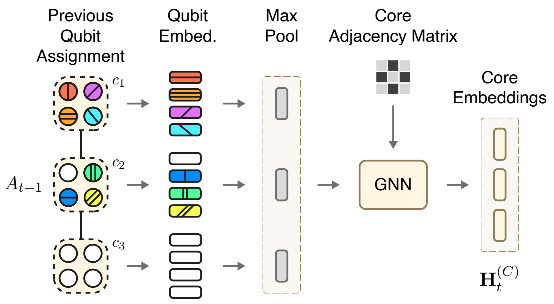

Fig. 5: Previous slice qubit assignment snapshot encoder.

loops and $\tilde{\mathbf{D}}_t^{-\frac{1}{2}}$ the element-wise reciprocal square root of its diagonal degree matrix. On the obtained qubit-wise embeddings $\mathbf{H}_t^{(I,Q)}$ we apply a max pooling across the qubit dimension $Q$. Then, sinusoidal positional encoding is applied finally obtaining the embedding $\mathbf{H}_t^{(I)} \in \mathbb{R}^{d_E}$ for each circuit time slice $t$.

*2)* `EncoderBlocks`*:* The initial slice embeddings $\mathbf{H}^{(I)} \in \mathbb{R}^{T \times d_E}$ are processed by $b$ transformer encoder blocks [8], [25], [48], which incorporate information from other circuit slices. This allows the policy to consider future slices when selecting cores for qubits, improving decision-making. The transformer model outperforms conventional neural transduction models like recurrent and convolutional networks by capturing long-range dependencies, making it ideal for large-scale problems and parallel execution. For more on the encoder block and on the definition of the scaled-dot-product attention function, see [48]. The output of the $b$-th (last) encoder block represents the ultimate slice embeddings $\mathbf{H}^{(S)} \in \mathbb{R}^{T \times d_H}$, where $T$ is the number of circuit slices and $d_H$ is the dimensionality of the embeddings, considered in the decoding phase.

## C. *Core Snapshot Encoder*

The task of the core `SnapshotEncoder` is to provide an encoding of the qubit allocations in the previous circuit time slice. During the decoding process, in fact, we also want the policy to take into account the previous qubit allocation, possibly trying to make a compromise between the state transfer distances from the previous slices and the qubit allocation for the next slices still to be decided.

During the decoding, when processing the $t$-th time slice, the output action for the previous slice $t-1$ is a vector $A_{t-1} \in \{1, \ldots, C\}^Q$, *i.e.*, it consists in the mapping between each logical qubit and the core index. As shown in Fig. 5, the Core Snapshot Encoder transform the allocation it represents in the embeddings $\mathbf{H}_t^{(C)} \in \mathbb{R}^{C \times d_H}$. This is obtained employing another GNN similarly to the `InitEmbedding` component. This time the input adjacency matrix is fixed and represents the connectivity of the multi-core hardware architecture. Regarding the node input features instead, these are obtained for each core by performing the max pooling of the embeddings of the logical qubits allocated on it. Logical qubit embeddings are calculated for each physical qubit available in the core. If a physical qubit is not associated to logical qubit, a padding embedding is considered.

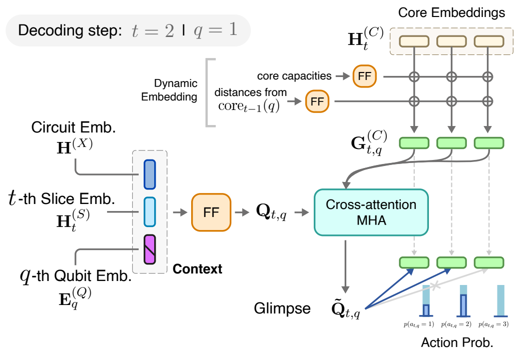
Decoding step: $t = 2$ | $q = 1$
Core Embeddings
$\mathbf{H}_t^{(C)}$
Dynamic Embedding
core capacities
distances from $\mathrm{core}_{t-1}(q)$
FF
FF
$\mathbf{G}_{t,q}^{(C)}$
Circuit Emb. $\mathbf{H}^{(X)}$
$t$-th Slice Emb. $\mathbf{H}_t^{(S)}$
$q$-th Qubit Emb. $\mathbf{E}_q^{(Q)}$
Context
FF
$\mathbf{Q}_{t,q}$
Cross-attention MHA
Glimpse
$\tilde{\mathbf{Q}}_{t,q}$
$p(a_{t,q}=1)$
$p(a_{t,q}=2)$
$p(a_{t,q}=3)$
Action Prob.

Fig. 6: Action probability calculation at each decoding step.

## D. *Decoding Process*

As described earlier and summarized in Algorithm 1, the decoding process spans $T \times Q$ steps, where each logical qubit in each slice is allocated sequentially. Each decoding step is identified by the indices $(t, q)$, representing the current slice and logical qubit, respectively. The context at each step is the concatenation of three embeddings: (a) the circuit embedding $\mathbf{H}^{(X)}$, calculated as the average of slice embeddings from the circuit slice encoder; (b) the slice embedding $\mathbf{H}_t^{(S)}$ for the current slice; (c) the trainable embedding $\mathbf{E}_q^{(Q)}$ of the current logical qubit.

At the beginning of each new slice $t$, the qubit allocation snapshot from the previous slice $t-1$ is encoded into embeddings $\mathbf{H}_t^{(C)} \in \mathbb{R}^{C \times d_H}$ by the `SnapshotEncoder`. For each logical qubit $q$, these embeddings are augmented via the `DynamicEmbedding` process. The remaining core capacities are projected into embeddings of dimensionality $d_H$ and summed with $\mathbf{H}_t^{(C)}$. Core capacities are reset at the start of each slice and decrease as qubits are allocated. Similarly, the distance of qubit $q$ from each core, based on its previous allocation and the architecture's core distance matrix, is also projected into embeddings and summed with the core embeddings. Finally, from this `DynamicEmbedding` procedure, we obtain core embeddings $\mathbf{G}_{t,q}^{(C)} \in \mathbb{R}^{C \times d_H}$ that incorporate information about: (a) qubit allocation in the previous slice; (b) current remaining capacity; (c) hypothetical state transfer cost for qubit being mapped;

As shown in Fig. 6, the final step of each decoding step is obtaining the probabilities of allocating the qubit $q$ of slice $t$ on each possible core $c$, starting from the current context and the core embeddings $\mathbf{G}_{t,q}^{(C)}$. This task is achieved with an attention mechanism [25] and it first involves projecting the context to obtain a query vector as follows:

$$\mathbf{Q}_{t,q} = \mathrm{concat}\left(\mathbf{H}^{(X)}, \mathbf{H}_t^{(S)}, \mathbf{E}_q^{(Q)}\right) \mathbf{W}^{(V,G)} \qquad (2)$$

where $\mathbf{W}^{(V,G)}$ is the trainable weight matrix for query projection from the context. Then, the cross attention is performed between the query and the core embeddings obtaining the attended query vector $\tilde{\mathbf{Q}}_{t,q}$ also known as *glimpse* [5], [49].

Subsequently, compatibilities are calculated between the attended query and the core embeddings. Compatibilities with cores on which mapping the current logical qubit $q$ would result in an invalid solution are masked, thus having:

$$u_{t,q,c} = \begin{cases} -\infty, & \text{if masked} \\ \frac{\tilde{\mathbf{Q}}_{t,q}\mathbf{K}_{t,q}^{\top}}{\sqrt{d_K}}, & \text{otherwise} \end{cases} \tag{3}$$

The resulting vector $\mathbf{U}_{t,q}$ can be interpreted as logarithm of probabilities (logits) and the final output action probability for the $(t,q)$ decoding timestep can be computed using the softmax function:

$$p(a_{t,q} = c) = \frac{e^{u_{t,q,c}}}{\sum_{j=1}^{C} e^{u_{t,q,j}}} \qquad \text{for } c = 1, 2, \ldots, C \tag{4}$$

Next section describes when a core (action) is masked and cannot be selected.

## E. Action Masking

Appropriately masking logits in Eq. (3) is crucial for building a valid solution, as it depends on the current partial solution from previous actions. The first masking mechanism addresses the core capacity constraint. A core's logit is masked if its remaining capacity is 0, preventing further qubit allocation in the current time slice. Another necessary masking applies to the *friendship* constraint. During decoding, qubits are allocated sequentially. If two qubits $q_1$ and $q_2$ are involved in the same gate and $q_1$ is allocated first, then when decoding $q_2$, only the core of $q_1$ is allowed. To ensure core capacity is not exhausted before allocating $q_2$, the core's capacity is reduced by 2 when allocating $q_1$, reserving space for $q_2$, but no capacity is reduced when $q_2$ is actually allocated.

One more issue arises for instance in a scenario with 2 cores, each having 2 physical qubits, and a circuit with four qubits $q_1, q_2, q_3$, and $q_4$, where $q_3 \bullet\!\!-\!\!\bullet\, q_4$ are involved in the same gate while $q_1$ and $q_2$ are non-interacting. If $q_1$ and $q_2$ are allocated to different cores before $q_3 \bullet\!\!-\!\!\bullet\, q_4$, both cores may have insufficient capacity to allocate the gate. To prevent this, when allocating a qubit, the number of remaining interacting qubit pairs $g$ is considered, and only cores that would keep sufficient capacity to allocate pairs (i.e., $\sum_{c=1}^{C} \lfloor \text{capacity}_c / 2 \rfloor \geq g$) are not masked.

## F. Reward and Training

After building an overall solution sampling the probability distributions at each decoding step, the corresponding reward is calculated as the total number of inter-core communications needed to bring a qubit in its assigned core from the core where it was allocated in the previous circuit slice. Formally, the reward is defined as:

$$R(A) = \sum_{t=2}^{T} \sum_{q=1}^{Q} D_{A_{t-1,q}, A_{t,q}} \tag{5}$$

where $D$ is the distance matrix and $A_{t,q}$ is the core (action) selected for $q$-th qubit at $t$-th slice. The training loss is then defined as the expected value of $R(A)$ and optimized by gradient descent using the REINFORCE [51] gradient estimator with rollout baseline as in [25].

## IV. Evalutation

In this section we provide implementation details and evaluation results of the proposed methodology.

### A. Experimental Setup

We implemented the proposed method using PyTorch [36], PyTorch Geometric [16], and RL4CO [6] libraries. Our models were trained on two architectures, each with 10 cores, where each core contains 10 physical qubits. One architecture features all-to-all core connectivity (*A2A*), and the other has a $2 \times 5$ mesh grid topology (*Grid*). We assume all-to-all connectivity within cores, as inter-core communication is costlier. We trained models on randomly generated quantum circuits with 30 time slices and 50 or 100 logical qubits, resulting in four trained models: *A2A-50*, *A2A-100*, *Grid-50*, and *Grid-100*.

Each model was trained for 100 epochs using 10,240 randomly generated circuits per epoch. Batch sizes were 128 for 100 qubits and 256 for 50 qubits due to memory limitations. Embedding dimensionality $d_E$ and latent space size $d_H$ were set to 256, with 8 attention heads in all Multi-Head Attention (MHA) layers, and 3 encoder blocks ($b$). The learning rate was $10^{-4}$, optimized with Adam. Training on a NVIDIA RTX 3060 12GB GPU took approximately 5 hours for 50 qubits and 10 hours for 100 qubits.

### B. Iterative black-box optimization approaches

This section presents the results of comparing qubit allocation solutions from iterative derivative-free optimization methods with those from the proposed trained policies. For solving the qubit allocation problem using algorithms like genetic algorithms, we use a suitable solution encoding for algorithm operators (e.g., crossover, mutation) to generate improved solutions based on a fitness function. We avoid encodings that lead to infeasible solutions to maintain sampling efficiency. Instead, we adopt a priority-based encoding, commonly used for problems like the Travelling Salesman Problem [17], [33].

The solution is encoded using a a real-valued array $\mathbf{x} \in \mathbb{R}^{T \times P}$, where $T$ is the number of slices and $P$ the total number of physical qubits. For each slice, a priority array $\mathbf{x}_t \in \mathbb{R}^{P}$ is used, and the first $C$ qubits with the highest priority are allocated to the first core, the next $C$ to the second, and so on. To ensure validity, the qubit with the highest priority in a gate pair determines the core for its *friend* qubit.

The solution fitness is evaluated using the same reward function as in Eq. 5. This encoding allows testing various gradient-free optimization algorithms: Random search, Differential Evolution (DE), Particle Swarm Optimization (PSO), Covariance Matrix Adaptation (CMA), Compact Genetic Algorithm (cGA) [20], One Plus One (1+1) [13], and NGOpt, an adaptive optimization algorithm from Nevergrad [40]. Results comparing these methods with the proposed approach are shown in Fig. 7. We tested two random 30-slice circuits with 50 and 100 qubits on both *Grid* and *A2A* topologies using the respective trained policy. The sampling budget for iterative algorithms was set at $10^6$ solutions, using default parameters. Search times for iterative methods ranged from

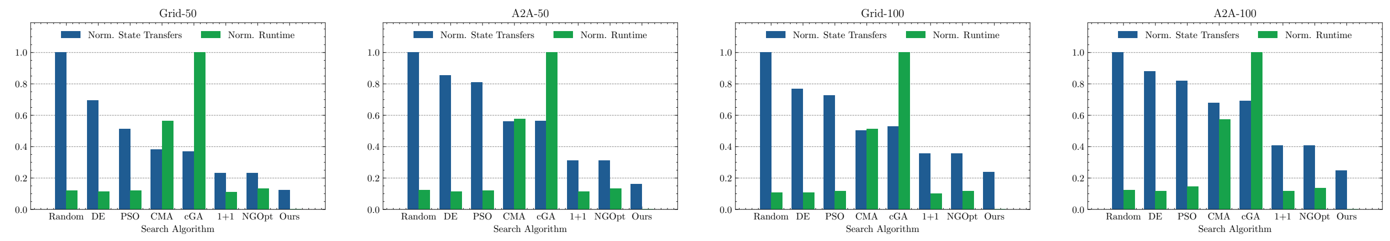

Fig. 7: Inter-core communication count and runtime comparison between the trained policy and black-box iterative optimization approaches on *Grid* and *A2A* architectures.

TABLE I: Trained policy performance for the $2 \times 5$ grid multi-core topology on benchmark circuits.

| Benchmark | Num. Qubits | Num. Slices | Dual Gate Count | Inter-core Comm. Count |
|---|---|---|---|---|
| QFT | 50 | 99 | 1225 | 1361 |
| | 100 | 197 | 4950 | 8918 |
| Quantum Volume | 50 | 50 | 1226 | 1892 |
| | 100 | 100 | 4961 | 11438 |
| Graph State | 50 | 83 | 596 | 827 |
| | 100 | 166 | 2449 | 5382 |
| Draper Adder | 50 | 120 | 925 | 1049 |
| | 100 | 245 | 3725 | 7406 |
| Cuccaro Adder | 50 | 290 | 336 | 250 |
| | 100 | 590 | 686 | 1782 |
| QNN | 50 | 195 | 2498 | 1627 |
| | 100 | 395 | 9998 | 10438 |

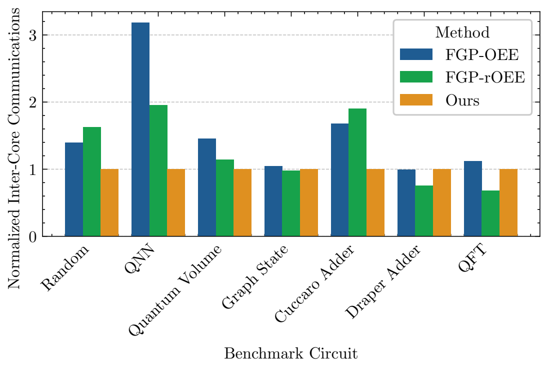

Fig. 8: Number of inter-core communication in *A2A-100* architecture for different benchmark circuits normalized with respect to the output of the proposed technique.

30 minutes to over 4 hours per circuit, while the trained policy returned solutions in seconds. Additionally, the proposed method achieved inter-core communication savings of 33.5% to 48.5% compared to the best baseline.

## C. Comparison with heuristic

In this section we analyze the performance of the trained policies on benchmark quantum circuits. We evaluate the performance for the *Grid* and *A2A* case. For the latter we compare the results with an heuristic technique for multi-core qubit allocation named Fine Grained Partitioning Overall Extreme Exchange (FGP-OEE) and the relaxed version (FGP-rOEE) [3]. These two techniques support fully connected (*A2A*) core topologies only. Table I shows the inter-core communications for benchmark circuits compiled for the *Grid* architecture (sparse topology) using the trained policies. QFT is the Quantum Fourier Transform without final qubit reordering swaps. Graph State refers to an encoder circuit for a graph with nodes equal to the number of qubits and a random adjacency matrix with 0.5 density. The Drapper and Cuccaro Adders are fixed precision adders for quantum state registers, and QNN represents a Quantum Neural Network circuit. All circuits were optimized and decomposed into two-qubit gates using Qiskit [38], with QNN algorithm from the MQT Benchmark Library [39]. Fig. 8 compares the proposed technique with FGP-OEE [3], using the same circuits from Table I but compiled for the *A2A* multi-core architecture. For a random circuit with 200 slices, the proposed method reduced inter-core communications by 28.26% compared to the baseline. Reductions of 48.82% and 40.53% were measured for QNN and Cuccaro Adders, respectively, while modest improvements were seen for Quantum Volume and Graph State circuits. However, there was a degradation of 32.44% to 47.08% for highly structured circuits like the Draper Adder and QFT, highlighting the challenge of training on random circuits alone. Synthetic circuit dataset generation may help address this [1].

## V. Conclusion

In this work, we tackled the challenges of scalability in quantum computing. The proposed reinforcement learning (RL) approach addresses the complexity of mapping quantum circuits to physical qubits, focusing on reducing inter-core communication and time-to-solution. Experimental results demonstrated the method's effectiveness in outperforming baselines, making DRL a promising approach for qubit allocation in modular quantum architectures. Future work could explore advanced masking with edge capacities, improved training via proximal policy optimization, and techniques like relative positional encoding [43] for symmetry. Training on structured algorithm circuits, not random ones, may also boost generalization.

## Acknowledgement

Authors gratefully acknowledge funding from the European Commission through HORIZON-EIC-2022-PATHFINDEROPEN01-101099697 (QUADRATURE).

## References

[1] B. Apak, M. Bandic, A. Sarkar, and S. Feld, "Ketgpt–dataset augmentation of quantum circuits using transformers," *arXiv preprint arXiv:2402.13352*, 2024.

[2] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[3] J. M. Baker, C. Duckering, A. Hoover, and F. T. Chong, "Time-sliced quantum circuit partitioning for modular architectures," in *ACM International Conference on Computing Frontiers*, 2020, pp. 98–107.

[4] M. Bandic *et al.*, "Mapping quantum circuits to modular architectures with qubo," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1. IEEE, 2023, pp. 790–801.

[5] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[6] F. Berto *et al.*, "RL4CO: a unified reinforcement learning for combinatorial optimization library," in *NeurIPS 2023 Workshop: New Frontiers in Graph Learning*, 2023, https://github.com/ai4co/rl4co.

[7] A. Botea, A. Kishimoto, and R. Marinescu, "On the complexity of quantum circuit compilation," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 9, no. 1, 2018, pp. 138–142.

[8] J. Chang, J. Wang, B. Li, Y. Zhao, and D. Li, "Attention-based deep reinforcement learning for edge user allocation," *IEEE Transactions on Network and Service Management*, 2023.

[9] E. Charbon, M. Babaie, A. Vladimirescu, and F. Sebastiano, "Cryogenic cmos circuits and systems: Challenges and opportunities in designing the electronic interface for quantum processors," *IEEE Microwave Magazine*, vol. 22, no. 1, pp. 60–78, 2020.

[10] L. Chen *et al.*, "Decision transformer: Reinforcement learning via sequence modeling," *Advances in neural information processing systems*, vol. 34, pp. 15 084–15 097, 2021.

[11] R. Chen, W. Li, and H. Yang, "A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1322–1331, 2022.

[12] D. Cuomo *et al.*, "Optimized compiler for distributed quantum computing," *ACM Transactions on Quantum Computing*, vol. 4, no. 2, pp. 1–29, 2023.

[13] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+ 1) evolutionary algorithm," *Theoretical Computer Science*, vol. 276, no. 1-2, pp. 51–81, 2002.

[14] P. Escofet, A. Ovide, C. G. Almudever, E. Alarcón, and S. Abadal, "Hungarian qubit assignment for optimized mapping of quantum circuits on multi-core architectures," *IEEE Computer Architecture Letters*, 2023.

[15] P. Escofet *et al.*, "Revisiting the mapping of quantum circuits: Entering the multi-core era," *ACM Transactions on Quantum Computing*, 2024.

[16] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[17] M. Gen, R. Cheng, and D. Wang, "Genetic algorithms for solving shortest path problems," in *IEEE International Conference on Evolutionary Computation*. IEEE, 1997, pp. 401–406.

[18] L. Giannelli *et al.*, "A tutorial on optimal control and reinforcement learning methods for quantum technologies," *Physics Letters A*, vol. 434, p. 128054, 2022.

[19] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *ACM symposium on Theory of computing*, 1996, pp. 212–219.

[20] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," *IEEE transactions on evolutionary computation*, vol. 3, no. 4, pp. 287–297, 1999.

[21] C.-Y. Huang, C.-H. Lien, and W.-K. Mak, "Reinforcement learning and dear framework for solving the qubit mapping problem," in *41st IEEE/ACM international conference on computer-aided design*, 2022, pp. 1–9.

[22] H. Jnane, B. Undseth, Z. Cai, S. C. Benjamin, and B. Koczor, "Multicore quantum computing," *Physical Review Applied*, vol. 18, no. 4, 2022.

[23] H. Kim, M. Kim, F. Berto, J. Kim, and J. Park, "Devformer: A symmetric transformer for context-aware device placement," in *International Conference on Machine Learning*. PMLR, 2023, pp. 16 541–16 566.

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[25] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[26] D. Kremer *et al.*, "Practical and efficient quantum circuit synthesis and transpiling with reinforcement learning," *arXiv preprint arXiv:2405.13196*, 2024.

[27] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for nisq-era quantum devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1001–1014.

[28] W. Li *et al.*, "A survey on transformers in reinforcement learning," *arXiv preprint arXiv:2301.03044*, 2023.

[29] W.-H. Lin, J. Kimko, B. Tan, N. Bjørner, and J. Cong, "Scalable optimal layout synthesis for nisq quantum processors," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[30] A. Montanaro, "Quantum algorithms: an overview," *npj Quantum Information*, vol. 2, no. 1, pp. 1–8, 2016.

[31] 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, pp. 1–31, 2022.

[32] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[33] R. J. Nowling and H. Mauch, "Priority encoding scheme for solving permutation and constraint problems with genetic algorithms and simulated annealing," in *International Conference on Information Technology: New Generations*. IEEE, 2011, pp. 810–815.

[34] A. Ovide *et al.*, "Mapping quantum algorithms to multi-core quantum computing architectures," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5.

[35] A. Pastor *et al.*, "Circuit partitioning for multi-core quantum architectures with deep reinforcement learning," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2024, pp. 1–5.

[36] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[37] P. Promponas *et al.*, "Compiler for distributed quantum computing: a reinforcement learning approach," *arXiv preprint arXiv:2404.17077*, 2024.

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

[39] N. Quetschlich, L. Burgholzer, and R. Wille, "Mqt bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, vol. 7, p. 1062, 2023.

[40] J. Rapin and O. Teytaud, "Nevergrad - A gradient-free optimization platform," https://github.com/facebookresearch/nevergrad, 2018.

[41] S. Rodrigo, S. Abadal, C. G. Almudéver, and E. Alarcón, "Modelling short-range quantum teleportation for scalable multi-core quantum computing architectures," in *ACM International Conference on Nanoscale Computing and Communication*, 2021, pp. 1–7.

[42] M. Sarovar *et al.*, "Detecting crosstalk errors in quantum information processors," *Quantum*, vol. 4, p. 321, 2020.

[43] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *arXiv preprint arXiv:1803.02155*, 2018.

[44] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[45] M. Y. Siraichi, V. F. d. Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation," in *International Symposium on Code Generation and Optimization*, 2018, pp. 113–125.

[46] K. N. Smith, G. S. Ravi, J. M. Baker, and F. T. Chong, "Scaling superconducting quantum computers with chiplet architectures," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1092–1109.

[47] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[48] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[49] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," *arXiv preprint arXiv:1511.06391*, 2015.

[50] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[51] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, pp. 229–256, 1992.

[52] H. Zhang *et al.*, "Mech: Multi-entry communication highway for superconducting quantum chiplets," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.