f-EVOVAQ: A GPU-based Framework for Evolutionary Training of Variational Quantum Algorithms

Giovanni Acampora, Angela Chiatto and Autilia Vitiello
Department of Physics "Ettore Pancini", *University of Naples Federico II*, Naples, Italy
{giovanni.acampora, angela.chiatto, autilia.vitiello}@unina.it

***Abstract*—In the field of quantum computing, Variational Quantum Algorithms (VQAs) combine quantum circuits with classical optimization to perform several tasks including classification and regression. Although gradient-based methods are commonly used for their training, these methods suffer from barren plateaus, noise sensitivity, and high computational costs. In this scenario, Evolutionary Algorithms (EAs) provide a gradient-free alternative for training VQAs enabling the discovery of high-quality solutions while requiring limited hardware resources. Unfortunately, the number of software tools aimed to the application of EAs for training of VQAs is limited and, moreover, they rarely exploits GPU acceleration. In order to bridge this gap, this work presents f-EVOVAQ, a Python package that extends the EVOVAQ framework by implementing vectorized evolutionary operators on GPU using CuPy arrays. The package enables efficient population-level optimization while remaining user-friendly for non-experts in evolutionary computation. Experiments on quantum classification tasks demonstrate significant reductions in optimization time and improved scalability, with limitations arising from GPU–CPU data transfers.**

***Index Terms*—Evolutionary Algorithms, GPU, Python Package, Quantum Computing, Software, Variational Quantum Algorithms**

I. Introduction

In recent years, progress in quantum hardware has brought the quantum computing field closer to practical realization. The present and upcoming era of quantum computers is identified as Noisy Intermediate-Scale Quantum (NISQ) [1], due to constraints such as the limited number of physical qubits available on a quantum processor and the presence of noise that affects the reliability of long sequences of quantum logical operations. In this scenario, Variational Quantum Algorithms (VQAs) have become popular as a way to think about quantum algorithms in the NISQ era. VQAs are hybrid quantum-classical approaches in which a classical optimizer iteratively updates the parameters of a quantum circuit to perform a given task. In detail, a task is encoded into a parameterized cost function that is evaluated using a quantum computer, and whose parameters are trained by a classical computer. The versatility of this approach allows us to solve problems in many areas, from physics to chemistry [2], from optimization to machine learning [3].

The majority of existing VQA training approaches rely on gradient-based optimization techniques. Consequently, software frameworks such as Qiskit [4], PennyLane [5], and TorchQuantum [6] only provide extensive support for differentiable programming of quantum circuits. Qiskit enables gradient-based optimization through quantum-specific gradient evaluation methods, including the parameter-shift rule and adjoint differentiation, and supports execution on both simulators and real quantum backends. PennyLane similarly provides analytic gradient computation based on quantum measurement rules, such as the parameter-shift, which can be executed on both quantum simulators and real quantum devices. In addition, PennyLane offers differentiable quantum simulators that integrate directly with classical automatic differentiation frameworks (e.g., NumPy, PyTorch, and JAX), enabling end-to-end gradient backpropagation entirely in simulation. Finally, TorchQuantum follows a PyTorch-style design in which parameterized quantum circuits are embedded within neural network modules and trained using PyTorch's reverse-mode automatic differentiation when operating on differentiable simulators. However, TorchQuantum is primarily specialized for efficient simulation and integration with classical deep learning workflows.

Despite their popular use, in practice, gradient-based methods for training of VQAs are affected by several limitations. The most well-known issues are the barren plateau phenomenon, where gradients vanish exponentially with system size, the high sensitivity to hardware noise, and the substantial quantum resource requirements for gradient estimation. These challenges hinder scalability and reduce the effectiveness of gradient-based optimization on the near-term quantum hardware. In this context, Evolutionary Algorithms (EAs) have recently emerged as a promising alternative for training VQAs across a variety of tasks [7]–[11]. Their gradient-free nature and robustness to noise make them suitable for practical deployment on the current NISQ devices. However, software support for evolutionary training of VQAs remains limited, particularly that capable of exploiting modern hardware accelerations such as GPUs.

To overcome these limitations, this work introduces a new Python package that extends and improves the capabilities of an existing framework for evolutionary training of VQAs, namely EVOVAQ [12], which currently represents a reference implementation for this workflow. The proposed package, named *f-EVOVAQ* (*fast-EVOVAQ*), incorporates GPU acceleration through CuPy arrays to support efficient population-level

operations, while preserving a high-level and user-friendly programming interface. In particular, the design of f-EVOVAQ aims to remain easily accessible to users who are not experts in evolutionary computation, allowing evolutionary training workflows to be specified with minimal configuration and without requiring detailed knowledge of the underlying optimization mechanisms. The main contributions of f-EVOVAQ are as follows:

- a vectorized representation of evolutionary operators that enables data-parallel computation across entire populations of candidate solutions;
- integration of a GPU-accelerated backend using CuPy to reduce classical optimization time in evolutionary training for VQAs;
- a unified API that supports multiple evolutionary algorithms and interfaces on both simulators and real quantum backend.

The experimental session demonstrates that f-EVOVAQ achieves performance improvements over CPU-only implementations in quantum classification tasks, although it exposes current limitations arising from the restricted interoperability between state-of-the-art quantum simulators and CuPy arrays.

II. Background

This section introduces the fundamental concepts of quantum computing and subsequently describes the principles of VQAs.

A. Essentials of Quantum Computing

Unlike classical bits, which can exist only in one of two definite states, 0 or 1, qubits possess the unique feature that their state can be 0, 1, or a superposition of both simultaneously. Accordingly, the most general state of a qubit is represented in bra-ket notation, or Dirac notation, as follows:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \tag{1}$$

where α_0 and α_1 are complex numbers, known as probability amplitudes; and the logic states $\{|0\rangle, |1\rangle\}$ are orthonormal vectors in the Hilbert space $\mathcal{H}$, forming the computational basis. Upon measurement, the qubit collapses into either $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha_0|^2$ and $|\alpha_1|^2$, respectively, satisfying the normalization condition $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

A quantum computer operates on a register of n qubits, whose general state is represented in bra-ket notation as

$$|\psi_n\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle , \tag{2}$$

where each coefficient α_x denotes the probability amplitude associated with the n-bit string x, therefore $\sum_{x \in \{0,1\}^n} |\alpha_x|^2 = 1$.

In the circuit model of quantum computation, logical operations on a quantum register are executed using quantum gates. Single-qubit gates include the Pauli operators $(\mathrm{X}, \mathrm{Y}, \mathrm{Z})$, as well as parameterized rotation gates such as $\mathrm{R_X}(\theta)$, $\mathrm{R_Y}(\theta)$ and $\mathrm{R_Z}(\theta)$. Among multi-qubit gates, the controlled-NOT (CNOT) gate is widely used to generate entanglement between qubits by applying an X operation to the target qubit conditional on the control qubit being in the state $|1\rangle$. A quantum algorithm is therefore defined as a finite sequence of quantum gates arranged into a quantum circuit, followed by a measurement process that extracts classical information from the final quantum state.

B. Variational Quantum Algorithms

VQAs are a class of hybrid quantum–classical methods designed to operate within the constraints of current NISQ devices. They combine a parameterized quantum circuit with a classical optimization routine, enabling the joint use of quantum and classical computational resources to address complex problems.

The general workflow of a VQA consists of three main components:

- the design of a parameterized quantum circuit, known as *ansatz*, typically composed of rotation and CNOT gates;
- the definition of a task-dependent cost function;
- the selection of a classical optimization algorithm to update the trainable parameters.

The structure of the ansatz can leverage prior knowledge of the problem, giving rise to problem-inspired ansatzes that encode domain-specific properties. Alternatively, problem-independent ansatzes adopt generic circuit architectures that are suitable when such information is not available. In practice, the choice of ansatz requires a balance between expressiveness, circuit depth, and hardware limitations. The cost function maps the circuit parameters to real values using probabilities or expectation values obtained from quantum measurements. It defines a multidimensional surface, known as the cost landscape, over which the classical optimizer searches for an optimal solution.

Finally, a classical optimization algorithm is employed to iteratively update the parameters in order to minimize the cost function, thus generating a hybrid quantum-classical loop. The choice of this optimizer is not trivial, as it can impact both optimization performance and quantum resource requirements.

III. Our proposal

This work presents f-EVOVAQ[1], a Python library specifically designed to efficiently run EAs on GPUs for the training of VQAs. In order to install it, it is possible to simply run the following shell command:

```
pip install fevovaq
```

The original version of EVOVAQ [12] is not suitable for this purpose, as it relies on CPU-bound NumPy arrays and a predominantly loop-based implementation that does not account for vectorized computations. Such an approach severely limits the exploitation of GPU architectures, whose performance advantages emerge primarily from massive data parallelism rather than sequential data processing. An in-depth description of software architecture and functionalities is given in the following sections.

[1] https://github.com/Quasar-UniNA/f-EVOVAQ/tree/main

A. Software Architecture

The main components of the f-EVOVAQ architecture are inherited from EVOVAQ and are summarized below.

- The `fevovaq.problem.Problem` class formalizes the optimization task in evolutionary terms. It requires the number of gate parameters to be optimized, the corresponding parameter bounds, the initialization range for sampling candidate solutions, and the cost function to be minimized. All parameters are assumed to be floats, consistently with the real-valued parameters used in parameterized quantum circuits.
- Several EA classes can be instantiated as classical optimizers for VQAs. The algorithms currently supported are:
  - Genetic Algorithms (GAs),
  - Differential Evolution (DE) algorithm,
  - Memetic Algorithms (MAs),
  - Big Bang–Big Crunch (BBBC) algorithm,
  - Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation (CHC) algorithm,
  - Particle Swarm Optimization (PSO) algorithm,
  - Hill Climbing (HC) algorithm, a local search technique commonly used in MAs.
- The `support` module contains the classes used as support facilities for each EA.
  - `BestIndividualTracker` class keeps track of the best solution ever found during the evolution.
  - `Logbook` class stores the statistics of fitness values in each generation.
  - `FinalResult` class collects the relevant information to return at the end of the optimization.

Users can find in-depth descriptions of each component along with helpful tutorials in the documentation[2].

Despite the architectural similarity, f-EVOVAQ introduces a substantial redesign of the internal data structures and evolutionary operators. In particular, NumPy arrays are systematically replaced with CuPy arrays [13], performing numerical operations locally on GPUs. While NumPy is restricted to CPU-based computation, CuPy provides an interface that transparently maps array operations to GPU kernels. For a Python library focusing on evolutionary computation for training VQAs, where optimization is performed through population-based search rather than gradient computation, CuPy is particularly well suited because it provides GPU acceleration with a NumPy-compatible API, enabling massive parallel computation on the entire population with minimal changes to existing NumPy-style evolutionary operators. Since no automatic differentiation is required and the computational bottleneck lies in population-level numerical transformations, CuPy's focus on high-performance array computing and direct execution on the GPU provides a simple and natural abstraction. This makes CuPy a pragmatic and lightweight choice for accelerating the classical evolutionary loop in hybrid quantum–classical optimization workflows.

Beyond the choice of the numerical backend, all evolutionary operators in f-EVOVAQ are implemented using a fully vectorized formulation. The population of candidate solutions is represented as a two-dimensional array, where each row corresponds to an individual and each column to a circuit parameter. Evolutionary operations are applied simultaneously to the entire population through array-level computations, thereby avoiding explicit Python loops and enabling efficient parallel execution. In the case of GAs, stochastic decisions such as mating and mutation are expressed through boolean mask arrays that identify the subsets of individuals or genes to which operators are applied. These masks are generated in a fully vectorized manner and used to combine or perturb candidate solutions via element-wise array operations. An analogous strategy is adopted for the other implemented EAs, including DE and PSO, where mutation, recombination, and velocity updates are reformulated as batched operations acting on the full population.

This design enables f-EVOVAQ to scale efficiently with both the population size and the dimensionality of the search space. Moreover, the vectorized formulation provides a unified computational model that allows for easy switching between CPU- and GPU-based execution, depending on the available hardware and problem requirements. When a GPU is unavailable, f-EVOVAQ falls back to NumPy-based CPU execution, still benefiting from vectorized operations.

B. Software Functionalities

Fig. 1 shows the workflow of EAs-based VQAs implemented in f-EVOVAQ across the different computational backends. While the population is entirely manipulated on the GPU through vectorized operations, the fitness evaluation step supports execution on real quantum processors (QPUs) as well as CPU- or GPU-based simulators.

Regardless of the specific task, the user is required to follow a small set of steps to implement this workflow.

1) After defining the parameterized quantum circuit for the target task, the user specifies a cost function to be passed to the `Problem` class. This function executes the quantum circuit by binding the gate parameters produced during the evolutionary process. To enable GPU-based optimization, the argument `backend` of the `Problem` class must be set to `"gpu"`. Depending on the version of the CUDA Toolkit available to the user, a specific version of CuPy must be installed, as described in the CuPy Installation Guide [14].
2) An EA can then be instantiated as a classical optimizer by specifying only the hyperparameters supported by the chosen algorithm class. The user can rely on the provided API documentation to guide this configuration.
3) The training process shown in Fig. 1 is initiated by calling the `optimize` method of the selected optimizer. The required arguments include the instantiated `Problem` object, the population size, and either the

[2]https://f-evovaq.readthedocs.io/en/latest/index.html

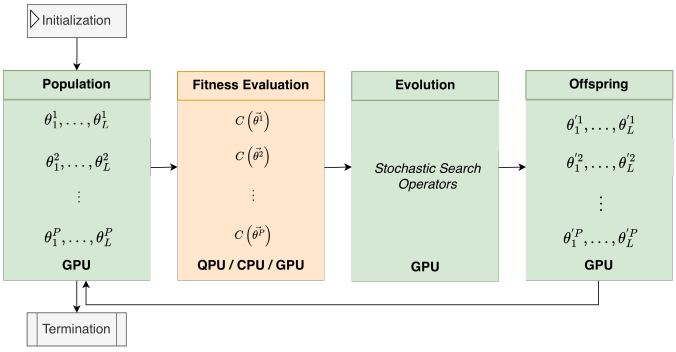

Fig. 1. Graphical representation of the workflow and workload of f-EVOVAQ on the different backends.

maximum number of generations or the maximum number of fitness evaluations.

4) Once the optimization is complete, the framework returns the best solution found, the total number of fitness evaluations performed, the number of completed generations, and a logbook collecting the evolutionary statistics.

This user-friendly workflow allows for an easy implementation of EAs for training VQAs in various tasks, including optimization and machine learning. Furthermore, users can easily customize their own evolutionary operators to include in their framework.

IV. Experiments and Results

This section evaluates the proposed Python package on a quantum classification task using two benchmark datasets. To highlight the advantages of the proposed GPU-enabled and vectorized implementation, a comparative analysis with the original EVOVAQ package is carried out. Details on the experimental setup are first provided, followed by an analysis of the results in terms of performance and computational time.

A. Experimental Setup

The experiments have been conducted on the well-known Iris and Wine dataset from the UCI Machine Learning Database Repository. As the primary goal is to study the performance of the library, only the fist two classes of each dataset are considered for simplicity. The datasets are split into training and test sets using 70/30 ratio. The quantum circuit employed as classification model is shown in Fig. 2. Each feature of the dataset is encoded into the rotation angle (f) of $\mathrm{R_Y}$ gates. The trainable part of the model consists of alternating layers of parameterized rotations ($\mathrm{U}\,(\vec{\theta})$), followed by CNOT gates to introduce entanglement among qubits. More complex models can be designed by increasing the number of layers (ℓ). For simplicity, only the first qubit is measured to produce the output. Specifically, the probability of measuring the state $|1\rangle$ is extracted and associated with class 1, and the loss function is computed using the cross-entropy criterion. Since the proposed library is designed to support scenarios involving a large number of parameters and large populations, thus requiring the simulation of large and deep quantum circuits, an efficient simulator is employed. In particular, the `AerStatevector` simulator from Qiskit Aer is used, as it enables full quantum circuit simulation on a GPU.

TABLE I
List of EAs and their hyperparameter values in the experimental session.

Algorithm	Hyperparameters
DE	Variant: best/1/bin; Probability of mating two solutions: 0.9; Differential weight: 0.8.
PSO	Velocity bound: (-0.5, 0.5); Inertia weight: 0.7298; Coefficient determining influence of personal best solution: 1.49618; Coefficient determining influence of global best solution: 1.49618.
GA	Selection Operator: Tournament Selection; Crossover Operator: Uniform Crossover; Mutation Operator: Gaussian Mutation; Probability of mating two solutions: 0.9; Probability of mutating a parameter in a solution: 0.25.

TABLE II
Benchmark datasets and number of trainable parameters for different circuit depths ℓ.

Dataset	# Features	# Instances	# Parameters		
			$\ell = 5$	$\ell = 10$	$\ell = 15$
Iris	4	100	60	120	180
Wine	13	130	195	390	585

Table I reports the EAs and hyperparameters considered in the experiments. To compare the performance of the two libraries, the optimization process was executed for 50 generations using increasing population sizes, specifically 50, 100, and 150 individuals. To further increase the number of trainable parameters, the number of circuit layers was set to 5, 10, and 15. Table II summarizes the characteristics of each dataset and the corresponding number of parameters for the tested circuit depths. To obtain statistically meaningful results, five independent runs are performed for each configuration using different random seeds. At the end of each optimization run, the execution time and a performance metric in terms of test accuracy are recorded. The reported results correspond to the mean values over the five runs.

B. Results and Discussion

Firstly, Tables III-IV report the test accuracy values and the training time differences (Δ) between EVOVAQ (CPU) and f-EVOVAQ (GPU) for both datasets across all configurations. The results indicate that the accuracy values obtained using

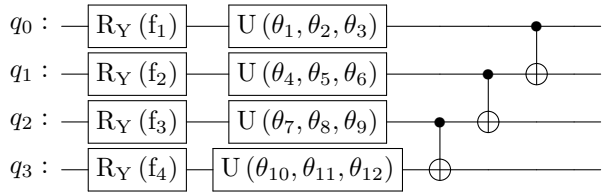

Fig. 2. Graphical representation of the parameterized quantum circuit used to perform classification task on a 4-qubit register with the number of layers $\ell = 1$.

TABLE III
TEST ACCURACY AND EXECUTION TIME DIFFERENCE BETWEEN CPU- AND GPU-BASED FRAMEWORKS ON THE IRIS DATASET, REPORTED FOR ALL EXPERIMENTAL CONFIGURATIONS.

Optimizer	ℓ	P	**Acc. (CPU)**	**Acc. (GPU)**	Δ **(s)**
DE	5	50	**0.9333**	**0.9333**	104.0546
DE	5	100	0.9133	**0.9200**	205.3655
DE	5	150	0.8933	**0.9333**	308.4594
DE	10	50	**0.9067**	0.8933	153.1045
DE	10	100	0.9400	**0.9667**	311.7227
DE	10	150	0.9333	**0.9400**	460.9121
DE	15	50	0.9467	**0.9533**	171.9246
DE	15	100	0.9333	**0.9533**	346.3788
DE	15	150	0.9067	**0.9333**	515.5286
PSO	5	50	0.9133	**0.9800**	103.7295
PSO	5	100	0.9600	**0.9667**	206.9339
PSO	5	150	**0.9733**	**0.9733**	313.1484
PSO	10	50	0.4733	**0.9867**	155.6549
PSO	10	100	0.6733	**0.9867**	310.4635
PSO	10	150	0.7733	**0.9667**	461.5642
PSO	15	50	0.4400	**0.9733**	170.5326
PSO	15	100	0.6733	**0.9867**	339.4523
PSO	15	150	0.4933	**0.9800**	514.4304
GA	5	50	**0.9267**	0.9200	103.8098
GA	5	100	0.9133	**0.9200**	206.5371
GA	5	150	0.9067	**0.9333**	310.6436
GA	10	50	0.8800	**0.9267**	154.5910
GA	10	100	0.9067	**0.9600**	310.0124
GA	10	150	0.9133	**0.9200**	461.8784
GA	15	50	**0.9533**	0.9267	169.7105
GA	15	100	**0.9533**	0.9133	339.6362
GA	15	150	**0.9733**	0.9467	510.4244

TABLE IV
TEST ACCURACY AND EXECUTION TIME DIFFERENCE BETWEEN CPU- AND GPU-BASED FRAMEWORKS ON THE WINE DATASET, REPORTED FOR ALL EXPERIMENTAL CONFIGURATIONS.

Optimizer	ℓ	P	**Acc. (CPU)**	**Acc. (GPU)**	Δ **(s)**
DE	5	50	**0.8974**	**0.8974**	229.8809
DE	5	100	**0.8974**	**0.8974**	460.5161
DE	5	150	**0.8974**	**0.8974**	688.0744
DE	10	50	**0.8974**	**0.8974**	191.4092
DE	10	100	**0.8974**	**0.8974**	379.9933
DE	10	150	**0.8974**	**0.8974**	564.2327
DE	15	50	**0.8974**	**0.8974**	-73.9387
DE	15	100	**0.8974**	**0.8974**	-118.6761
DE	15	150	**0.8974**	**0.8974**	-213.1347
PSO	5	50	**0.8974**	**0.8974**	225.1866
PSO	5	100	**0.8974**	**0.8974**	451.1776
PSO	5	150	**0.8974**	**0.8974**	678.2558
PSO	10	50	0.8923	**0.8974**	188.5526
PSO	10	100	**0.9077**	0.8974	376.7214
PSO	10	150	**0.9026**	0.8974	588.5484
PSO	15	50	**0.9231**	0.9026	-76.3967
PSO	15	100	**0.9231**	0.8974	-112.0123
PSO	15	150	**0.9128**	0.8974	-224.2164
GA	5	50	**0.9026**	**0.9026**	221.9380
GA	5	100	**0.8974**	**0.8974**	443.5342
GA	5	150	**0.8974**	**0.8974**	664.4474
GA	10	50	0.9077	**0.9128**	180.0602
GA	10	100	0.8769	**0.9179**	359.5983
GA	10	150	**0.9077**	0.9026	539.4643
GA	15	50	**0.8974**	0.8718	-112.8361
GA	15	100	0.8923	**0.9128**	-259.8213
GA	15	150	**0.9128**	0.8923	-411.7465

the GPU backend are generally comparable, and in some cases slightly higher, than those obtained with the CPU implementation. Given the stochastic nature of EAs, these differences can be attributed to the fact that NumPy and CuPy generate random seeds differently, which can lead to different optimization trajectories and final solutions.

After demonstrating that f-EVOVAQ (GPU) achieves suitable performance in terms of solution quality, we shift our focus to the core contribution of this paper that is to improve the computational efficiency. In detail, Figs. 3-4 show the mean training times obtained over five independent runs for EVOVAQ (CPU) and f-EVOVAQ (GPU) on both datasets and for all considered configurations. As observed, f-EVOVAQ consistently outperforms EVOVAQ in terms of execution time, and the performance gap increases with the number of circuit layers and, consequently, with the number of trainable parameters. Furthermore, the reported time differences highlight that the performance gap increases also with the population size. The only exception occurs for the Wine dataset with 15 layers, where f-EVOVAQ performs slightly worse than EVOVAQ. This is caused by the fact that AerStatevector accepts only NumPy arrays as input. Consequently, at each fitness evaluation step, CuPy arrays must be converted into NumPy arrays. In the Wine dataset with 15 layers, corresponding to the configuration with the largest number of parameters, the overhead introduced by this data transfer offsets the computational advantage provided by GPU-based classical processing. To conclude, although the time saved per run is limited, it accumulates into a significant advantage over multiple executions, reducing the overall computational cost.

V. CONCLUSIONS

This work introduced f-EVOVAQ, a GPU-accelerated extension of the EVOVAQ framework for evolutionary training of VQAs. By combining vectorized evolutionary operators with CuPy-based GPU computation, the proposed package enables more efficient population-level optimization while maintaining a simple and user-friendly interface. Experimental results show that f-EVOVAQ reduces classical optimization overhead and scales well with increasing population sizes and parameter dimensionality, providing an effective solution for accelerating evolutionary optimization in VQAs. Current limitations mainly arise from data transfer between GPU and CPU when interfacing with existing quantum simulators, which may limit the achievable speedup. Nevertheless, results indicate that for circuits up to 13 qubits (e.g., the Wine dataset) and up to 390 parameters, GPU-based computation still offers a significant performance advantage despite this overhead.

ACKNOWLEDGMENT

This work was supported by Ministero dell'Università e della Ricerca in the FIS3 program (grant number E53C25002450001, QARECA).

REFERENCES

[1] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[2] Y. Zhang, L. Cincio, C. F. Negre, P. Czarnik, P. J. Coles, P. M. Anisimov, S. M. Mniszewski, S. Tretiak, and P. A. Dub, "Variational quantum eigensolver with reduced circuit complexity," *npj Quantum Information*, vol. 8, no. 1, p. 96, 2022.

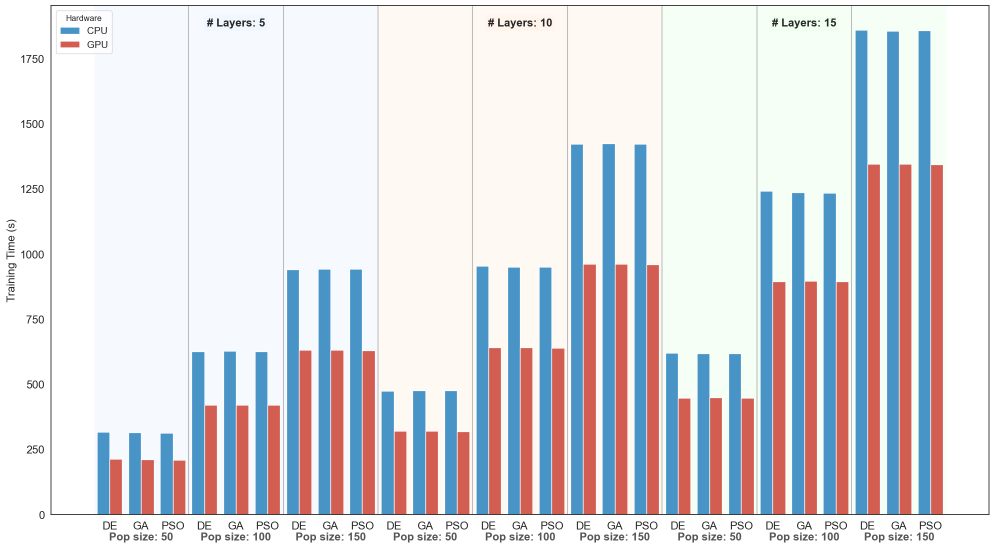

Fig. 3. Bar plot showing training times for the Iris dataset in the various configurations tested.

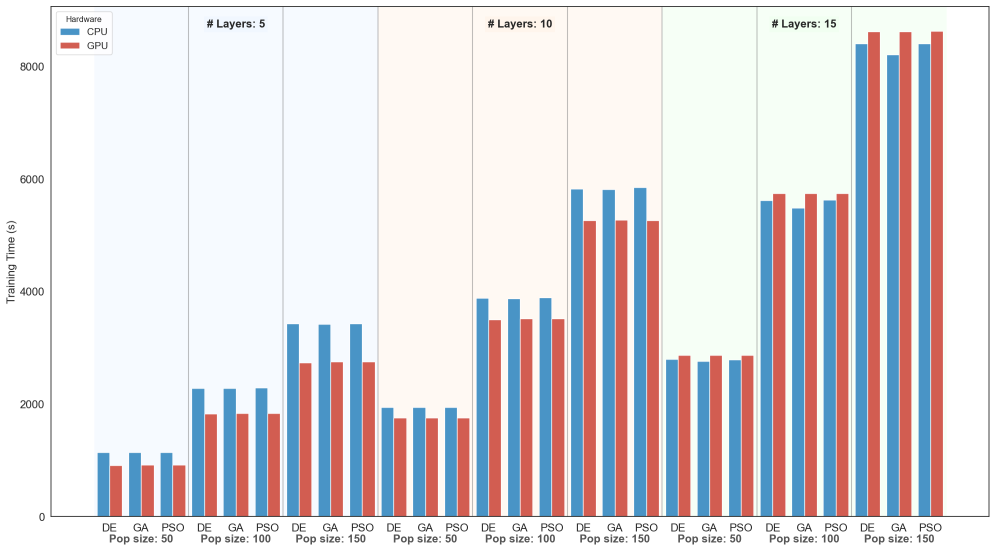

Fig. 4. Bar plot showing the training times for the Wine dataset in the various configurations tested.

[3] G. Acampora, A. Chiatto, R. Schiattarella, and A. Vitiello, "Quantum artificial intelligence: A survey," *Computer Science Review*, vol. 59, p. 100807, 2026.

[4] "Gradient framework," https://qiskit-community.github.io/qiskit-algorithms/tutorials/12_gradients_framework.html, accessed in January 2026.

[5] "Gradients and training in pennylane documentation," https://docs.pennylane.ai/en/stable/introduction/interfaces.html, accessed in January 2026.

[6] "A pytorch library for quantum simulation and quantum machine learning," https://torchquantum.readthedocs.io/en/latest/index.html, accessed in January 2026.

[7] G. Acampora, A. Chiatto, and A. Vitiello, "Training variational quantum circuits through genetic algorithms," in *2022 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2022, pp. 1–8.

[8] A. Massa, G. Nunziata, F. Polverino, L. Campanile, M. Castaldo, L. Coraggio, S. Crisci, G. De Gregorio, W. I. Ibnsalih, N. Itaco *et al.*, "A quantum machine learning algorithm for hazelnut variety recognition," in *2025 IEEE International Conference on Quantum Artificial Intelligence (QAI)*. IEEE, 2025, pp. 67–72.

[9] G. Acampora, A. Chiatto, and A. Vitiello, "Genetic algorithms as classical optimizer for the quantum approximate optimization algorithm," *Applied Soft Computing*, vol. 142, p. 110296, 2023.

[10] S. Y.-C. Chen, C.-M. Huang, C.-W. Hsing, H.-S. Goan, and Y.-J. Kao, "Variational quantum reinforcement learning via evolutionary optimization," *Machine Learning: Science and Technology*, vol. 3, no. 1, p. 015025, 2022.

[11] G. Acampora, A. Chiatto, and A. Vitiello, "Training circuit-based quantum classifiers through memetic algorithms," *Pattern Recognition Letters*, vol. 170, pp. 32–38, 2023.

[12] G. Acampora, C. C. Gutierrez, A. Chiatto, J. M. S. Hidalgo, and A. Vitiello, "Evovaq: Evolutionary algorithms-based toolbox for variational quantum circuits," *SoftwareX*, vol. 26, p. 101756, 2024.

[13] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," *31st confernce on neural information processing systems*, vol. 151, no. 7, 2017.

[14] "Cupy installation," https://docs.cupy.dev/en/stable/install.html, accessed in January 2026.