EvoX: A JAX-based EC Library

Beichen Huang, Luming Yang, and Jiachun Li

***Abstract*—During the past decades, evolutionary computation (EC) has demonstrated promising potential in solving various complex optimization problems of relatively small scales. However, as modern EC research is facing problems of a much larger scale, computing power begins to struggle. To meet this grave challenge, to present `EvoX`, a distributed GPU-accelerated EC library. In this report, we will show how `EvoX` is suitable for all EC algorithms while extending to many real-life benchmark problems. We will also show that `EvoX` can also greatly accelerate EC workflow.**

I. Introduction

A. Background

Over the years, Evolutionary Computation (EC) algorithms are mainly written for CPUs. Although running the EC algorithm on the CPU is simple and direct, it is unable to accelerate EC algorithms at the hardware level. But with the increase in population size or dimensions of the problem, evolutionary algorithms will consume much more computing resources, and as a result, it will computing resources will become one of the main constraints of evolutionary computing.

GPU acceleration is a technology that utilizes GPU to do computing instead of CPU [1]. Compared to CPUs, GPUs usually have more cores, which makes them better at processing data in parallel. In the past decade, GPU acceleration has been one of the driving forces of deep learning. Libraries like TensorFlow [2], Pytorch [3], and JAX [4] all have the ability to accelerate computing using GPUs. EC on the other hand, although parallel in nature, few works were purposed to accelerate EC algorithm on GPUs. Among many GPU-accelerated libraries in Python, JAX is one of the more general-purpose ones. JAX provides various composable function transformations that enable vectorized and hardware-accelerated execution. Unlike deep learning tasks, where the most time-consuming operations are simply matrix multiplication, EC operators usually involve a series of relatively cheap operations, causing memory read and write (I/O). However, since memory bandwidth is typically limited, having too much I/O will make GPU acceleration not as effective as it could be. Thankfully, JAX can also use XLA [5] to compile and optimize NumPy [6] programs, and one of the optimizations is done by fusing multiple operations into a single GPU kernel. This makes JAX well-suited for EC workflow.

B. Difficulty Analysis

1) JAX: JAX is a library that is designed with functional programming in mind. In JAX, arrays are immutable, meaning arrays cannot be changed after created. The state of evolutionary algorithm, however, needs to change after every iteration. This makes it extremely tricky to implement a framework in JAX that can support both immutable arrays and stateful computation. What's worse EC algorithms are usually developed in a modular way, for example, an algorithm can consist of a series of selection, mutation, and crossover operations, and each operation can have its own state.

2) Gym: Our library needs to access the Gym platform, but Gym is a library originally made for the RL community, thus the API of Gym was made to allow easy integration with RL algorithms, not EC ones. Another problem is that the Gym environment is single-threaded.

II. Related Work

A. Libraries on EC

1) DEAP: DEAP is a novel evolutionary computation framework for rapid prototyping and testing of ideas. It seeks to make algorithms explicit and data structures transparent. It works in perfect harmony with parallelization mechanisms such as multiprocessing and SCOOP. [7]

2) LEAP: LEAP is a general purpose Evolutionary Computation package that combines readable and easy-to-use syntax for search and optimization algorithms with powerful distribution and visualization features. LEAP's signature is its operator pipeline, which uses a simple list of functional operators to concisely express a metaheuristic algorithm's configuration as high-level code. Adding metrics, visualization, or special features (like distribution, coevolution, or island migrations) is often as simple as adding operators into the pipeline. [8]

3) pymoo: pymoo (Multi-objective Optimization in Python) offers state-of-the-art single-objective and multi-objective algorithms and many more features related to multi-objective optimization such as visualization and decision-making. [9]

B. Libraries on JAX

1) EvoJAX: EvoJAX is a scalable, general-purpose, hardware-accelerated neuroevolution toolkit. Built on top of the JAX library, this toolkit enables neuroevolution algorithms to work with neural networks running in parallel across multiple TPU/GPUs. EvoJAX achieves very high performance by implementing the evolution algorithm, neural network, and task all in NumPy, which is compiled just in time to run on accelerators. [10] However, EvoJAX is mainly based on RL, not EC.

2) evosax: evosax allows people to leverage JAX, XLA compilation, and auto-vectorization/parallelization to scale ES to some accelerators. The API is based on the classical ask, evaluate, tell cycle of ES. Both ask and tell calls are compatible with `jit`, `vmap/pmap`, and `lax.scan`. It includes a vast set of both classics (e.g. CMA-ES, Differential Evolution, etc.) and modern neuroevolution (e.g. OpenAI-ES, Augmented RS, etc.) strategies. [11] However, Evosax doesn't have state management, which leads to some confusion in the logic of

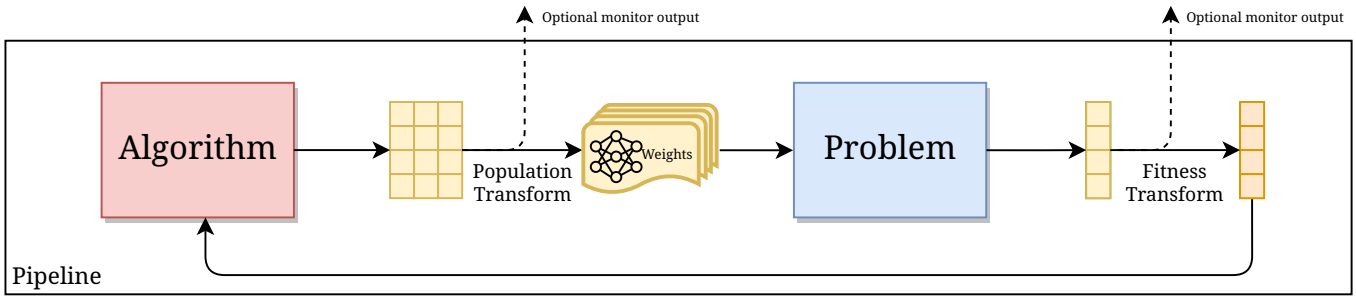


Fig. 1: Standard pipeline of EvoX.

TABLE I: Summary of notations

Notation	Description
$\mathcal{A}_\theta$	The algorithm parameterized by θ
$\mathcal{P}_\mathcal{D}$	The problem parameterized by $\mathcal{D}$
$S_t^{\mathcal{A}_\theta}$	The state of $\mathcal{A}_\theta$
$S_t^{\mathcal{P}_\mathcal{D}}$	The state of $\mathcal{P}_\mathcal{D}$
f	The evaluation function of the problem
h	The decoder function
g^{ask}	The ask part of the algorithm at generation t
g^{tell}	The tell part of the algorithm at generation t
$\mathbf{X}_t$	The candidate solutions at generation t
$\mathbf{y}_t$	The candidate solutions at generation t

the code. And it is not very compatible with codes other than JAX, nor does it support multi-targeting, CC, and distributed pipelines.

III. ENGINEERING DESIGN

A. Overall Design

Specifically, given θ and $\mathcal{D}$ are the hyperparameters for defining the problem and the algorithm respectively, the algorithm can be characterized by $\mathcal{A}_\theta = \langle \theta, g^{\text{ask}}, g^{\text{tell}} \rangle$, problem $\mathcal{P}_\mathcal{D} = \langle \mathcal{D}, f \rangle$, where g^{ask} and g^{tell} are `ask` and `tell` actions for generating a new population and updating the algorithm state. A simple iteration on generation t is:

$$\mathbf{X}_t, S_{t+1}^{\mathcal{A}_\theta} = g_\theta^{\text{ask}}(S_t^{\mathcal{A}_\theta}), \tag{1}$$

$$\mathbf{y}_t, S_{t+1}^{\mathcal{P}_\mathcal{D}} = f_\mathcal{D}(S_t^{\mathcal{P}_\mathcal{D}}, h(\mathbf{X}_t)), \tag{2}$$

$$S_{t+1}^{\mathcal{A}_\theta} = g_\theta^{\text{tell}}(S_{t+1}^{\mathcal{A}_\theta}, \mathbf{y}_t), \tag{3}$$

where P_t, F_t denote the population of candidate solutions and the corresponding fitness values at generation respectively; $S^{\mathcal{A}_\theta}$, $S^{\mathcal{P}_\mathcal{D}}$ are the state of the algorithm and problem respectively; h is the optional decoder function. A summary of notations is given in I.

In terms of the code, the most important component of EvoX is `State` and `StatefulModule`, `State` is the representation of state itself, and `StatefulModule` is the class that everything that needs stateful computation must inherit from. `StatefulModule` provides the ability to merge the states of all sub-modules into one single state.

The overall design is illustrated in Fig. 1.

B. Algorithm Design

All algorithms are a subclass of Algorithms and must implement two methods, ask and tell. EvoX adopts the ask-and-tell interface to decouple algorithms and problems so that algorithms don't call problems internally. As a result, this will give more freedom to the implementation of the problem and the optimization loop.

1) Single-objective:

1) PSO: Particle swarm optimization is an optimization algorithm. Particle swarm can find the optimal destination through collective information sharing. In the algorithm process, each particle searches along the direction determined by itself, and records the position where it has found the largest fitness in the search process. At the same time, all particles share their fitness in each iteration, so that the particle swarm knows which position has the largest fitness currently. [12] PSO has the advantages of fast convergence, few parameters and simple algorithm.
Pseudo code shows how the PSO algorithm is used in EvoX architecture. We use the generic Ask Tell structure. Pseudo code is part of the Tell function. We store the key in state S_t to ensure the determinism of the random process.

Algorithm 1: Particle Swarm Optimization - TELL

Data: State S_t, fitness $\mathbf{y}_t$
Result: State S_{t+1}

$Key, Key_{rg}, Key_{rp} \leftarrow SplitKey(S_t.key)$
$R_g \leftarrow randomUniform(Key_{rg})$;
$R_p \leftarrow randomUniform(Key_{rp})$;
$LB_i \leftarrow$ Local Best Location$(S_t.position_i, \mathbf{y}_t)$;
$GB \leftarrow$ Global Best Location$(S_t, \mathbf{y}_t)$;
for *Particle i in population* **do**
  $\Delta V_{i,inertia} \leftarrow W \cdot S_t.velocity_i$;
  $\Delta V_{i,recog} \leftarrow \phi_p \cdot R_p \cdot (LB_i - S_t.position_i)$;
  $\Delta V_{i,society} \leftarrow \phi_g \cdot R_g \cdot (GB - S_t.position_i)$;
for *Particle i in population* **do**
  $\Delta V_i \leftarrow \Delta V_{i,inertia} + \Delta V_{i,recog} + \Delta V_{i,society}$;
  $S_{t+1}.position_i \leftarrow S_t.position_i + \Delta V_i$;
return update(S_{t+1})

2) CMA-ES: The CMA-ES is a stochastic, or randomized, method for real-parameter (continuous domain) optimization of non-linear, non-convex functions, where CMA stands for Covariance Matrix Adaptation. [13]

2) Multi-objective:

1) NSGA-II: Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a non-dominated sorting-based MOEA [14]. NSGA-II purposed a faster non-dominated sort and combined it with crowding distance, and achieved great success in solving multi-objective problems.
2) MOEA/D: MOEA/D is a multi-objective evolutionary algorithm based on decomposition. Each sub problem is optimized by only using information from its several neighboring sub problems, which makes MOEA/D have lower computational complexity at each generation than MOGLS and NSGA-II [15]. MOEA/D is a classic decomposition based multi-objective algorithm.

C. Problem Design

Problem class provides a unified API for evaluation, to capture the noisy nature of some problems, EvoX also makes Problem a StatefulModule, so one can include a random number generator key inside the state. Since algorithms are fully decoupled from problems, problems can have quite different implementations, for example, problems can be written in JAX and GPU accelerated, but problems can be also written in libraries other than JAX.

1) OpenAI Gym: The gym is a standard API for reinforcement learning and a diverse collection of reference environments. The gym provides us with a variety of small games. These games have different goals and tasks. We let the relevant algorithms under the EvoX run in the gym to complete these target tasks to test the speed and completion degree and give good results. The gym itself has a set of interfaces for RL, which include State and Action, but EC is generally to run the entire environment completely and then iterate, so we need to encapsulate a layer of interfaces. And since Gym is single-threaded, we need to encapsulate many threads.

2) Neural network training: TorchVision [16] is an open-source library for Pytorch, that can provide data sets for deep learning tasks. It is possible to integrate EvoX with TorchVision to provide easy access to neuroevolution tasks, for example, training a neural network on the CIFAR-10 data set. We made a simple wrapper for Torchvision to make it like a native JAX library. To bring machine learning workflow to EC, we also model the problem as stateful. As illustrated in Fig. 2, the problem has the ability to fetch the correct batch by looking into the state $S_t^{\mathcal{P}_\mathcal{D}}$.

IV. Experiments

To demonstrate the effectiveness of GPU acceleration, we use CMA-ES on the sphere function for 1000 iterations with a population size set to 100. We scale the dimension of the sphere from 10 to 10,000 and then the dimension to 100 and scale the population size from 10 to 100,000. The run-time

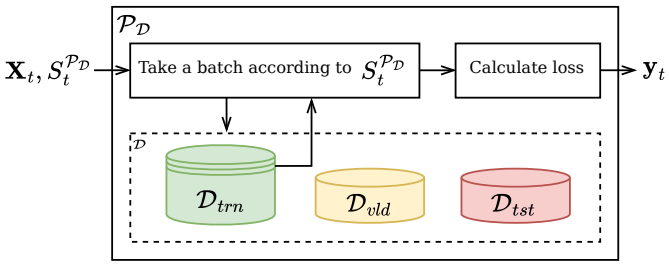
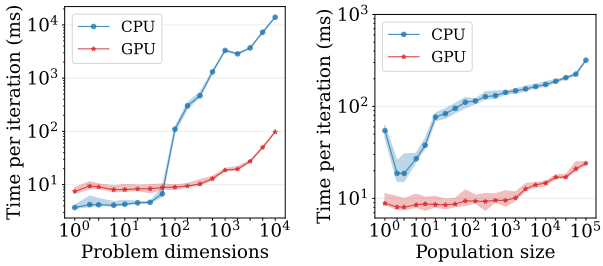


Fig. 2: A demonstration of how one problem can handle different evaluation modes. In this case, $S_t^{\mathcal{P}}$ indicates that the second batch from the training dataset should be used.



(a) Scale dimensions (b) Scale population size

Fig. 3: CMA-ES scaling experiment on EvoX

was measured twice, once with GPU enabled, and another with GPU disabled.

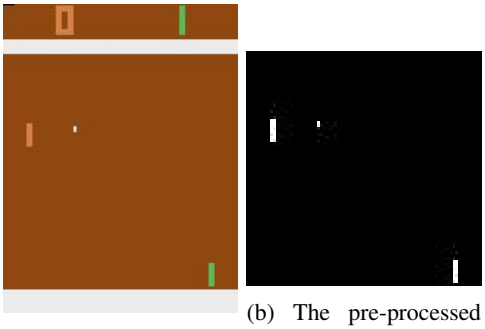
The result is given in Fig. 3. When running with 10 dimensions, the CPU and GPU behaved similar performance, but when the dimension scaled to one million, the program was about 2 times faster on GPU than on the CPU.

To demonstrate the extendibility of our library, we run a novel RL task, ATARI game Pong with PGPE algorithm. In this environment, the agent controls a paddle on the right side of the screen and uses it to hit a ball away from its own goal and into the opponent's goal. We directly use the raw frame from the game, which is a 210×160 colored image as illustrated in Fig. 4a. The image is then pre-processed by first converting the image to gray-scale, then cropping out unnecessary parts (e.g. the scoreboard), downsampled by a factor of two, and finally erasing the background. The final outcome is an 80×80 gray-scale image as illustrated in Fig. 4b. The pre-processed image is then mapped to the probability of available actions by a neural network. The architecture of this network is given in II. We use the PGPE algorithm to directly optimize the parameters of this neural network. The initial parameter of this neural network is only initialized once, and kept the same across different experiments, but the seed for the algorithm is changed. The experiment is repeated 11 times.

As is shown in Fig. 5, it is totally possible to use the EC algorithm to directly train an RL agent.

References

[1] S. Jia, Z. Tian, Y. Ma, C. Sun, Y. Zhang, and Y. Zhang, "A Survey of GPGPU Parallel Processing Architecture Performance Optimization," in *2021 IEEE/ACIS 20th International Fall Conference on Computer and Information Science (ICIS Fall)*, Oct. 2021, pp. 75–82.



(a) Raw image (b) The pre-processed image

Fig. 4: The ATARI game Pong

TABLE II: The architecture of the neural network used the ATARI game PONG.

Input Shape	Layer	Filter Shape
6400	Fully Connected	6400×64
64	Fully Connected	64×64
64	Fully Connected	64×10

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," May 2016, arXiv:1605.08695 [cs]. [Online]. Available: http://arxiv.org/abs/1605.08695

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Dec. 2019, arXiv:1912.01703 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1912.01703

[4] "JAX: Autograd and XLA," Oct. 2022, original-date: 2018-10-25T21:25:02Z. [Online]. Available: https://github.com/google/jax

[5] "XLA: Optimizing Compiler for Machine Learning." [Online]. Available: https://www.tensorflow.org/xla

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, number: 7825 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2

[7] "deap: Distributed evolutionary algorithms in python." [Online]. Available: https://github.com/DEAP/deap

[8] J. Bassett, "LEAP: A general purpose library for evolutionary algorithms in python." [Online]. Available: https://github.com/aureumchaos/leap

[9] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.

[10] Y. Tang, Y. Tian, and D. Ha, "Evojax: Hardware-accelerated neuroevolution," *arXiv preprint arXiv:2202.05008*, 2022.

[11] R. T. Lange, "evosax: Jax-based evolution strategies," 2022. [Online]. Available: http://github.com/RobertTLange/evosax

[12] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, Nov. 1995, pp. 1942–1948 vol.4.

[13] N. Hansen, "The CMA Evolution Strategy: A Tutorial," Apr. 2016, arXiv:1604.00772 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1604.00772

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002, conference Name: IEEE Transactions on Evolutionary Computation.

[15] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, Dec. 2007, conference Name: IEEE Transactions on Evolutionary Computation.

[16] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *Proceedings of the 18th ACM international conference on Multimedia*, ser. MM '10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 1485–1488. [Online]. Available: https://doi.org/10.1145/1873951.1874254

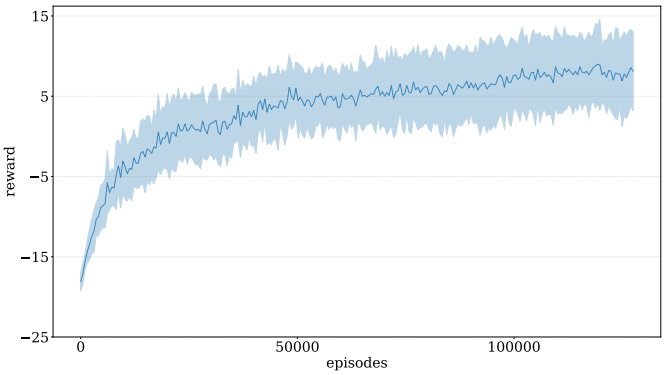


Fig. 5: Reward of ATARI game Pong across different seeds. The dark line is the average reward at that number of episode and the area is bounded by the standard deviation.