

Bridging Evolutionary Multiobjective Optimization and GPU Acceleration via Tensorization

Zhenyu Liang, Hao Li, Naiwei Yu, Kebin Sun, and Ran Cheng

Abstract—Evolutionary multiobjective optimization (EMO) has made significant strides over the past two decades. However, as problem scales and complexities increase, traditional EMO algorithms face substantial performance limitations due to insufficient parallelism and scalability. While most work has focused on algorithm design to address these challenges, little attention has been given to hardware acceleration, thereby leaving a clear gap between EMO algorithms and advanced computing devices, such as GPUs. To bridge the gap, we propose to parallelize EMO algorithms on GPUs via the *tensorization* methodology. By employing tensorization, the data structures and operations of EMO algorithms are transformed into concise tensor representations, which seamlessly enables automatic utilization of GPU computing. We demonstrate the effectiveness of our approach by applying it to three representative EMO algorithms: NSGA-III, MOEA/D, and HypE. To comprehensively assess our methodology, we introduce a multiobjective robot control benchmark using a GPU-accelerated physics engine. Our experiments show that the tensorized EMO algorithms achieve speedups of up to 1113× compared to their CPU-based counterparts, while maintaining solution quality and effectively scaling population sizes to hundreds of thousands. Furthermore, the tensorized EMO algorithms efficiently tackle complex multiobjective robot control tasks, producing high-quality solutions with diverse behaviors. Source codes are available at <https://github.com/EMI-Group/evomo>.

Index Terms—Evolutionary multiobjective optimization (EMO), GPU acceleration, robot control, tensorization.

I. INTRODUCTION

In many real-world optimization problems (e.g., material design [1], [2], energy management [3], [4], network optimization [5], and portfolio optimization [6]), decision-makers must consider multiple (and often conflicting) objectives simultaneously. Without loss of generality, such multiobjective optimization problems (MOPs) can be defined as:

$$\underset{\mathbf{x}}{\text{minimize}} \quad \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})), \quad (1)$$

where $\mathbf{x} = (x_1, x_2, \dots, x_d) \in \mathcal{X} \subseteq \mathbb{R}^d$ is the decision vector and d is the number of decision variables. $\mathbf{f} : \mathcal{X} \rightarrow \mathbb{R}^m$ maps the decision vector to an m -dimensional objective space. Each $f_i : \mathcal{X} \rightarrow \mathbb{R}$ for $i = 1, \dots, m$ represents an objective function that needs to be minimized (or maximized). The key challenge in solving MOPs is to identify a set of trade-off solutions known as the Pareto set (PS), where no single

Zhenyu Liang, Hao Li, Naiwei Yu, and Kebin Sun are with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mails: zhenyuliang97@gmail.com; li7526a@gmail.com; yunaiweiyu@gmail.com; sunkebin.cn@gmail.com).

Ran Cheng is with the Department of Data Science and Artificial Intelligence and the Department of Computing, The Hong Kong Polytechnic University, Hong Kong SAR, China. e-mail: ranchengcn@gmail.com. (*Corresponding author: Ran Cheng*)

solution can optimize all objectives simultaneously. This set contains all Pareto-optimal or nondominated solutions, and their corresponding points in the objective space collectively form the Pareto front (PF).

Over the past two decades, the field of evolutionary multiobjective optimization (EMO) [7], [8] has seen rapid advancements, resulting in the development of various effective algorithms for solving MOPs. Broadly, these algorithms can be categorized into three main methods: 1) dominance-based; 2) decomposition-based; and 3) indicator-based. Dominance-based algorithms, such as NSGA-II [9] and NSGA-III [10], select solutions based on dominance relations between individuals. Decomposition-based algorithms, like MOEA/D [11], break down an MOP into multiple simpler subproblems, which are optimized collaboratively. Indicator-based algorithms, such as HypE [12], focus on optimizing specific performance indicators, like hypervolume (HV) [13].

While EMO algorithms have proven effective in solving various MOPs, their performance is significantly constrained by limitations in *computing power*. First, since the majority of existing EMO algorithms still rely on CPUs for execution, their computational efficiency is inherently limited, particularly when addressing large-scale MOPs (LSMOPs) [14]. Second, the inconsistent implementation of EMO algorithms across different methods has led to fragmentation, making it difficult to standardize solutions and apply them across diverse domains. Without a unified framework, efforts to generalize these algorithms and enhance computational efficiency are impeded. Third, much of the current research remains focused on relatively simpler numerical optimization tasks, often neglecting computationally intensive real-world applications. This narrow focus further limits the practical use of EMO algorithms in scenarios where real-time performance and scalability are crucial.

To address these limitations, one promising method is to incorporate modern computational accelerators such as GPUs. With their powerful parallel processing capabilities, GPUs have demonstrated significant performance improvements in fields like deep learning [15]. However, to fully leverage the potential of GPUs for EMO algorithms, a systematic method of parallelization is necessary, yet little effort has been made in this direction so far.

Given the high concurrency enabled by the large number of *tensor cores* [16], GPUs are particularly well-suited for efficient handling and acceleration of large-scale data processing. Correspondingly, one promising method for parallelization on GPUs is *tensorization*, i.e., representing data structures and operations as tensors. Building upon this concept, we introduce

a concise and general tensorization methodology for accelerating EMO algorithms on GPUs. By leveraging tensor operations and the inherent parallelism of GPUs, this methodology systematically explains how to transform EMO algorithms into concise tensor representations. Using this methodology, we implement tensorized versions of three representative EMO algorithms from each category: 1) the dominance-based NSGA-III [10]; 2) the decomposition-based MOEA/D [11], and 3) the indicator-based HypE [12]. Moreover, to evaluate the performance of tensorized EMO algorithms in GPU computing environments, we develop a multiobjective robot control benchmark using Brax [17], a GPU-accelerated physics engine. The main contributions of this research are as follows:

- 1) We introduce a general tensorization methodology for EMO algorithms that transforms key data structures (i.e., candidate solutions and objective values) and operations (i.e., crossover, mutation, and selection) into tensor representations. This approach establishes concise yet versatile mathematical models for enabling efficient GPU acceleration of EMO algorithms.
- 2) We apply the proposed tensorization methodology to three representative EMO algorithms: NSGA-III, MOEA/D, and HypE. The tensorized algorithms achieve up to $1113\times$ speedup compared to their CPU-based counterparts, while maintaining solution quality and effectively scaling population sizes to hundreds of thousands.
- 3) We develop a multiobjective robot control benchmark called MoRobtrol. This benchmark represents a computationally intensive scenario with complex black-box properties. It demonstrates the ability of the tensorized EMO algorithms to efficiently generate high-quality solutions with diverse behaviors in such computationally expensive environments.

The structure of this article is as follows: Section II reviews the background and related work. Section III introduces the tensorization methodology for GPU acceleration. Section IV details the implementations of core operations in three representative EMO algorithms. Section V introduces the multiobjective robot control benchmark. Section VI outlines the experimental setup and results. Section VII summarizes the findings and discusses future work.

II. BACKGROUND

A. Taxonomy of EMO Algorithms

Traditional EMO algorithms can generally be classified into three main categories based on their selection mechanisms: dominance-based, decomposition-based, and indicator-based [7].

Dominance-based EMO algorithms are pivotal in addressing complex optimization tasks through Pareto dominance. As the pioneering algorithm in this category, NSGA-II [9] introduced a fast nondominated sorting approach, which has since become a foundation for many subsequent algorithms. SPEA2 [18] introduces a fine-grained fitness assignment strategy, density estimation, and enhanced archive truncation to improve performance. GrEA [19] enhances convergence and diversity balance

by using grid dominance. NSGA-III [10] further advances diversity management in higher-dimensional spaces with reference points. Recent algorithms build on these foundations with innovative strategies. BiGE [20] focuses on proximity and diversity through bi-goal optimization, while VaEA [21] balances these using vector-angle-based principles. RSEA [22] improves performance by projecting solutions into a radial space, and NSGA-II/SDR [23] introduces a novel dominance relation with adaptive niching techniques. MSEA [24] divides the optimization process into stages to enhance diversity preservation. PMEA [25] eliminates dominance resistance solutions using an interquartile range method.

Decomposition-based EMO algorithms address MOPs by decomposing them into simpler subproblems [26]. These algorithms can be further categorized into two types: 1) weighted aggregation-based and 2) reference set-based methods. MOEA/D [11] is the most representative weighted aggregation-based method, which aggregates objectives using weight vectors and has inspired variants like MOEA/D-DRA [27] and EAG-MOEA/D [28]. Recent developments include MOEA/D-AAWNs [29], which adapts weight vectors and neighborhoods to enhance diversity. MOEA/D-GLCM [30] employs bidirectional global search and adaptive neighborhood strategies to improve population distribution. Reference set-based methods, on the other hand, divide the objective space using reference points or vectors, guiding the search toward underexplored regions while maintaining diversity. Representative algorithms of this type include MOEA/D-M2M [31], RVEA [32], and θ -DEA [33]. A recent work, ECRA-DEA [34], adaptively allocates resources across subspaces using fitness contribution and improvement rates.

Indicator-based EMO algorithms rely on performance indicators for selection [35]. As a notable example, IBEA [36] uses binary ϵ^+ indicators for decision-making. Following this, SMS-EMOA [37] and HypE [12] are representative HV-based algorithms, with HypE using Monte Carlo sampling to approximate HV. BCE-IBEA [38] integrates bi-criterion evolution with the IBEA framework, while SRA [39] combines multiple indicators with stochastic ranking based environmental selection. MOMBI-II [40] uses the R2 metric, and AR-MOEA [41] is guided by the enhanced inverted generational distance (IGD) metric. MaOEA/IGD [42] employs the IGD metric. Recently, R2HCA-EMOA [43], HVCTR [44], and IMOEA-ARP [45] extended the SMS-EMOA framework with innovations in HV approximation, reference point management, and diversity handling, respectively.

B. GPU Acceleration in EMO Algorithms

Most previous efforts in GPU acceleration for EMO have focused on specific algorithms and implementations of certain algorithmic components. An early contribution by Wong [46] introduced GPU-accelerated nondominated sorting in NSGA-II. Building on this, Sharma and Collet [47] proposed G-ASREA, a GPU-accelerated variant of NSGA-II that incorporates an external archive to sort nondominated solutions on the GPU. Arca *et al.* [48] applied GPU acceleration to the evaluation phase of NSGA-II, specifically within the context

of fuel treatment optimization. Aguilar-Rivera [49] further developed a fully vectorized NSGA-II, employing stochastic non-domination sorting and grid-crowding techniques.

In addition to dominance-based algorithms, Souza and Pozo [50] proposed the GPU-accelerated MOEA/D-ACO algorithm, where solution construction and pheromone matrix updates benefit from data-parallel processing. Lopez *et al.* [51] leveraged GPUs to accelerate HV contribution calculations, improving performance in the SMS-EMOA algorithm. Furthermore, Hussain and Fujimoto [52] introduced a fast CUDA-based implementation of MOPSO on GPUs.

More recently, frameworks based on Google's JAX [53] have opened new possibilities for GPU acceleration in EMO. Examples include EvoJAX [54], evosax [55], and EvoX [56], all of which provide open-source platforms for GPU-accelerated evolutionary algorithms. While EvoJAX and evosax focus on accelerating evolutionary strategies, EvoX is designed as a distributed GPU-accelerated framework that supports general evolutionary computation. Building upon EvoX, a recent effort has been made to tensorize the RVEA [32] for GPU acceleration [57].

Despite these advances, GPU-accelerated EMO algorithms remain in their infancy. Current research has primarily focused on specific implementations that provide isolated performance improvements. Moreover, many of these implementations rely heavily on CUDA programming [58] and are not open-source, thus making them less accessible, particularly for beginners.

III. TENSORIZATION METHODOLOGY

In this section, we present how to adopt the general tensorization methodology in EMO algorithms. Specifically, we begin by defining the notation and preliminary concepts used throughout this article, including the basic definitions of *tensor* and *tensorization*. Next, we demonstrate how to transform atomic operations such as basic operations and control flow operations into tensors. Furthermore, we discuss why and how tensorization matters for GPU acceleration.

A. Preliminaries

A *tensor* is a multidimensional array that generalizes scalars, vectors, and matrices to higher dimensions [59]. Formally, a k -th order tensor is an element of the tensor product of k vector spaces: $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}$, where d_i represents the dimension along the i -th mode (axis) of the tensor. Scalars are zero-order tensors, vectors are first-order tensors, and matrices are second-order tensors. Correspondingly, *tensorization* refers to the process of transforming algorithmic data structures and operations into tensor representations. This transformation facilitates efficient parallel computation, particularly on massively parallel hardware such as GPUs, by leveraging the inherent parallelism in tensor operations to improve computational performance and scalability.

In this article, scalars (0-order tensors) are denoted by lowercase letters (e.g., a), vectors (1st-order tensors) are denoted by italicized bold lowercase letters (e.g., \mathbf{a}), matrices (2nd-order tensors) are denoted by italicized bold uppercase letters (e.g., \mathbf{A}), and higher-order tensors are denoted by

TABLE I
TENSOR VARIABLES IN EMO ALGORITHMS

Notation	Description
n	Population size
m	Number of objectives
d	Problem dimension
\mathbf{X}	Solution tensor
\mathbf{F}	Objective tensor
\mathbf{R}, \mathbf{W}	Reference and weight tensors
\mathbf{U}, \mathbf{L}	Upper and lower bound tensors

calligraphic letters (e.g., \mathcal{T}). The tensor variables related to EMO algorithms are summarized in Table I. Correspondingly, a tensorized MOP can be formulated as:

$$\underset{\mathbf{X}}{\text{minimize}} \quad \mathbf{F}(\mathbf{X}) = (\mathbf{f}_1(\mathbf{X}), \mathbf{f}_2(\mathbf{X}), \dots, \mathbf{f}_m(\mathbf{X})), \quad (2)$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the solution tensor for n individuals and d dimensions, and $\mathbf{F} \in \mathbb{R}^{n \times m}$ is the corresponding objective tensor.

TABLE II
BASIC TENSOR OPERATIONS

Operation	Description
$\mathbf{A} \cdot \mathbf{B}$	Tensor multiplication: Product of two tensors \mathbf{A} and \mathbf{B} .
$\mathbf{A} \odot \mathbf{B}$	Hadamard product: Element-wise multiplication of \mathbf{A} and \mathbf{B} .
$H(\mathbf{A})$	Heaviside step function: Returns 1 if $\mathbf{A}_{ij} \geq 0$, 0 otherwise.
$\mathbf{1}_{\mathbf{A}}$	Indicator function: Returns 1 if \mathbf{A}_{ij} is true, 0 otherwise.
sort	Arranges elements in ascending order.
argsort	Returns indices of sorted elements.
min, max	Returns the smallest or largest element along the specified axis: column-wise if axis = 0, row-wise if axis = 1.
argmin	Returns the indices of the smallest elements.
vmap	Vectorization map: Applies a function across an array axis.

B. Tensorization of Data Structures in EMO Algorithms

In EMO algorithms, the candidate solutions and their corresponding objective values are two critical data structures, as expressed in (2). They can be encoded as tensors \mathbf{X} and \mathbf{F} , respectively. Moreover, decomposition-based algorithms use additional structures such as reference vectors or weights, which are similarly encoded as tensors \mathbf{R} and \mathbf{W} , respectively. Here, $\mathbf{R} \in \mathbb{R}^{r \times m}$ typically represents r reference vectors, each corresponding to a different point in the objective space, and $\mathbf{W} \in \mathbb{R}^{n \times m}$ denotes a weight tensor with n different weights. These tensor representations allow for the parallel and batch processing of operations along the population dimension, capitalizing on the independence of individuals within the population. Consequently, the EMO algorithm can efficiently process one entire population at a time.

C. Tensorization of Operations in EMO Algorithms

After establishing an effective tensorized data structure, the next crucial step is to consider tensorization for core operations in EMO algorithms, such as crossover, mutation, and selection mechanisms (e.g., environmental selection). These operations

consist of numerous atomic operations, including both basic tensor operations and control flow operations.

1) *Basic Tensor Operations*: Basic tensor operations serve as the foundation for transforming more complex operations in EMO algorithms. Table II summarizes these operations, including tensor multiplication, Hadamard product, Heaviside step function, and indicator function, as well as advanced functions like sort, argsort, min, max, and argmin. These functions are essential for implementing selection and ranking strategies, enabling GPU-accelerated nondominated sorting, and diversity maintenance in EMO algorithms.

2) *Control Flow Operations*: The tensorization of control flow operations, including *loops* and *branches*, is a key challenge in transforming traditional EMO algorithms into their tensor-based counterparts. Control flow operations are commonly used to implement iterative procedures and define selection rules based on specific conditions. However, traditional control flow operations such as loops (`for` and `while`) and `if-else` branches introduce sequential dependencies that hinder parallel execution and reduce the efficiency of GPU computations. Tensorizing these operations requires replacing them with tensor-based operations that can execute in parallel.

Loop operations are often used in EMO algorithms for tasks like calculating distances or updating solutions. However, with a large population size, sequential processing becomes inefficient. These loops can be replaced by either vectorized mapping functions (e.g., `vmap`), or by using broadcasting combined with basic operations. The `vmap` function automatically applies a specified function across all elements in a given tensor dimension, eliminating explicit loops. Mathematically, `vmap` can be expressed as:

$$\text{vmap}(f)(\mathbf{A}) = [f(\mathbf{A}_1), f(\mathbf{A}_2), \dots, f(\mathbf{A}_n)]^\top, \quad (3)$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$ is the input tensor and f is the function applied to each individual \mathbf{A}_i . The function f operates independently on each element, with `vmap` managing parallel computation and concatenation of results. Alternatively, broadcasting can be used to perform the same operations without any loops. Broadcasting works by expanding the dimensions of tensors to align them, enabling element-wise operations to be performed simultaneously across the entire population.

Branch, such as `if-else` operations, are another type of control flow that poses challenges for tensorization. In traditional implementations, `if-else` operations introduce branching and disrupt parallel execution. To address this issue, tensorization replaces branch with element-wise masking operations such as `where`. For example, a traditional branch assigning values to a population matrix based on a threshold can be written as:

$$\mathbf{Y}_{ij} = \begin{cases} \mathbf{A}_{ij}, & \text{if } M_{ij} > \tau \\ \mathbf{B}_{ij}, & \text{otherwise} \end{cases}, \quad (4)$$

where \mathbf{A} , \mathbf{B} are input tensors, τ is a threshold, and M is the mask tensor. It can be replaced by a tensorized masking operation:

$$\mathbf{Y} = \mathbf{1}_{M>\tau} \odot \mathbf{A} + (1 - \mathbf{1}_{M>\tau}) \odot \mathbf{B}, \quad (5)$$

which can be implemented as `where(M > \tau, A, B)`.

```

1 import torch
2
3 # Conventional implementation
4 def dominance_detection_conventional(P):
5     n = P.size(0)
6     dom = torch.zeros(n, n, dtype=torch.bool)
7     for i in range(n):
8         for j in range(n):
9             if i != j:
10                 if (P[i] <= P[j]).all() and
11                     (P[i] < P[j]).any():
12                         dom[i, j] = True
13
14 return dom
15
16 # Tensor-based implementation
17 @torch.compile
18 def dominance_detection_tensor(P):
19     P1 = P.unsqueeze(1)
20     P2 = P.unsqueeze(0)
21     return (P1 <= P2).all(dim=2) & \
22            (P1 < P2).any(dim=2)

```

Listing 1: Comparison between conventional and tensor-based implementations of Pareto dominance detection.

3) *Advantages over Conventional Operations*: Tensorization offers several key advantages over conventional EMO implementations. First, it provides greater flexibility by handling multi-dimensional data, whereas conventional matrix operations are often limited to two dimensions. Second, tensorization enhances computational efficiency by enabling parallel processing and removing the need for explicit loops and conditional branches. Finally, tensorization simplifies the code, making it more concise and easier to maintain. For instance, as shown in Listing 1, conventional Pareto dominance detection relies on nested `for` loops and `if-else` statements to compare individuals. In contrast, the tensorized version uses element-wise operations, broadcasting, and masking to perform these comparisons in parallel, significantly reducing code complexity and boosting performance.

D. Discussion

1) *Why Is Tensorization Crucial for GPU Acceleration*: With thousands of cores designed to manage multiple tasks simultaneously, GPUs are intrinsically tailored for parallel computing. The architectures of GPUs, such as the SIMD (Single Instruction, Multiple Threads) model, enable efficient execution of tensor operations. Moreover, the specialized features like NVIDIA's Tensor Cores further enhance performance by accelerating matrix multiplication and accumulation tasks. This makes the tensorization methodology ideally suited for GPUs, which inherently involve large-scale parallel computations.

2) *What Algorithms Are Suitable for Tensorization*: Algorithms with independent computations and minimal branching are ideal for tensorization, as they can be easily parallelized. In contrast, algorithms that depend on sequential processes, frequent branching, or recursion pose challenges for parallelization. For example, in traditional MOEA/D, the aggregate function computation relies on results from previous iterations, thus making direct tensorization challenging. Nonetheless, by

restructuring and decoupling such algorithms, the tensorization methodology can still be effectively applied.

IV. TENSORIZATION IMPLEMENTATION IN REPRESENTATIVE EMO ALGORITHMS

In this section, we present the application of tensorization methodology in three representative EMO algorithms: NSGA-III, MOEA/D, and HypE. The genetic operators, including mating selection, crossover, and mutation, are common across most EMO algorithms and follow similar tensorization procedures, which are elaborated in Section S.I of the Supplementary Document. Here, we focus on the tensorized implementation of the environmental selection operators specific to each algorithm.

It is important to note that both the environmental selection in NSGA-III and the Monte Carlo-based selection in HypE are inherently highly parallelizable, which allows for straightforward tensorization. In contrast, the MOEA/D algorithm presents a unique challenge due to its fundamentally sequential nature. As shown in Algorithm 1, each subproblem in MOEA/D involves four interdependent steps that must be executed in sequence. This sequential dependency prevents direct tensorization and requires a reconfiguration of the entire process to enable parallel computation, which will be elaborated in Section IV-B.

A. Tensorized Environmental Selection in NSGA-III

The key components of environmental selection in NSGA-III include *nondominated sorting*, *normalization*, *association*, *niche count calculation*, and *niche selection*. The tensorization process of each component is elaborated as follows. The pseudocode of both the original and tensorized algorithms is provided in Section S.II of the Supplementary Document.

1) *Nondominated Sorting*: Given the combined objective tensor $\mathbf{F} \in \mathbb{R}^{2n \times m}$, representing the objective tensor of both parent and offspring populations, the nondominated rank is computed iteratively. The primary goal is to assign a nondominated rank to each individual, where lower ranks correspond to better solutions.

First, the dominance relation tensor $\mathbf{D} \in \{0, 1\}^{2n \times 2n}$ is computed using vmap or broadcasting for parallel processing. Each element D_{ij} indicates whether the solution \mathbf{F}_i dominates \mathbf{F}_j :

$$D_{ij} = \mathbf{F}_i \prec \mathbf{F}_j, \quad i, j = 1, 2, \dots, 2n. \quad (6)$$

Next, we calculate the dominance count tensor $\mathbf{c} \in \mathbb{Z}^{2n}$, which indicates how many individuals each individual dominates:

$$\mathbf{c} = \sum_{j=1}^{2n} D_{ij}. \quad (7)$$

The rank tensor $\mathbf{r} \in \mathbb{Z}^{2n}$ is initialized to zeros, and the rank counter k is set to zero. The boolean tensor $\mathbf{p} \in \{0, 1\}^{2n}$, the set of all nondominated solutions at the current rank, is obtained by $\mathbf{p} = \mathbf{1}_{c=0}$.

In each iteration, individuals sharing the same dominance rank are identified and processed collectively. This method ensures that even with a large population size, the number

of iterations remains relatively low. Additionally, the while function is optimized for accelerated computation. The rank tensor \mathbf{r} is updated as follows:

$$\mathbf{r} = H(\mathbf{p}) \cdot k + H(1 - \mathbf{p}) \odot \mathbf{r}. \quad (8)$$

After rank assignment, the dominance count \mathbf{c} is updated by:

$$\mathbf{c} = \mathbf{c} - \sum_{i=1}^{2n} \mathbf{p}_i \cdot \mathbf{D}_{ij} - \mathbf{p}. \quad (9)$$

The process repeats until all individuals are ranked, which means that all elements in \mathbf{p} are zero. Once all ranks are assigned, the tensor \mathbf{r} is sorted to determine the rank l of the n -th individual.

2) *Normalization*: After performing nondominated sorting, the objective tensor \mathbf{F} undergoes a normalization process, similar to that in the original NSGA-III algorithm. This normalization ensures that the objectives are comparable by mapping them onto a hyperplane, enabling the algorithm to maintain diversity across generations.

3) *Association*: In this step, each individual in the population is associated with the closest reference point. The distance between the normalized objective tensor \mathbf{F}' and the reference tensor \mathbf{R} is computed using the perpendicular distance:

$$\mathbf{D} = \|\mathbf{F}'\| \cdot \sqrt{1 - (\mathbf{F}' \cdot \mathbf{R}^\top / (\|\mathbf{F}'\| \cdot \|\mathbf{R}\|))^2}. \quad (10)$$

Based on the distance tensor \mathbf{D} , the index of the closest reference point π is the index of the minimum value in each row of \mathbf{D} , and the corresponding distance d represents the minimum value in each row.

4) *Niche Count Calculation*: For each reference point, the niche count is computed, which indicates how many individuals are associated with that reference point. The niche count tensor ρ is calculated as:

$$\rho_j = \sum_{i=1}^{2n} H(l - \mathbf{r}_i) \cdot \mathbf{1}_{\pi_i=j}, \quad j = 1, \dots, n_r, \quad (11)$$

where n_r is the number of reference points. The tensor ρ_l represents the niche count for the last front (i.e., the niche count corresponding to the front when $\mathbf{r}_i = l$):

$$\rho_{l,j} = \sum_{i=1}^{2n} \mathbf{1}_{\mathbf{r}_i=l} \cdot \mathbf{1}_{\pi_i=j}, \quad j = 1, \dots, n_r. \quad (12)$$

The total number of selected individuals n_s is then updated as $n_s = \sum \rho$.

5) *Niche Selection*: In the niche selection process of NSGA-III, the distance tensor \mathbf{D}' represents \mathbf{D} adjusted by niche counts for each reference tensor. This tensor is used to identify individuals closest to underpopulated niches. The index of the selected individual q is determined by minimizing the distance in each row of \mathbf{D}' , i.e., $q = \arg \min_j (\mathbf{D}')$. After selecting the individual, the rank tensor \mathbf{r} is updated by setting $\mathbf{r}[q] = l - 1$, with further updates in subsequent iterations reflecting the inclusion of individuals in the current front.

Once the selection is complete, the indices of the top n individuals for the next generation are determined by:

$$i_{\text{next}} = \text{sort}(H(l - \mathbf{r}) \odot \mathbf{a} + \infty \cdot (1 - H(l - \mathbf{r})))[: n], \quad (13)$$

Algorithm Environmental Selection of TensorMOEA/D
Input: Population tensor \mathbf{X} , Objective tensor \mathbf{F}_1 , Offspring tensor \mathbf{O} , the objective of offspring \mathbf{F}_2 , the population size n , the ideal points \mathbf{z} , the weights \mathbf{W} , the neighbors indices \mathbf{I}_{nb} , and the PBI function f_{PBI} .
Output: Next population tensor \mathbf{X}_{next} and next objective tensor \mathbf{F}_{next} :
1: $z_{\min} \leftarrow \min_i(\mathbf{z} \cup \mathbf{F}_2)$;
2: $I_{\text{sub}} \leftarrow [i \mid i \in \mathbb{N}, 0 \leq i < n]$;
3: $M \leftarrow \mathbf{0}^n$;
4: **Function** $f_{\text{op1}}(i_{\text{nb}}, f_2)$
5: $g_{\text{old}} \leftarrow f_{\text{PBI}}(\mathbf{F}_1[i_{\text{nb}}], \mathbf{w}[i_{\text{nb}}], z_{\min})$;
6: $g_{\text{new}} \leftarrow f_{\text{PBI}}(\mathbf{f}_2, \mathbf{w}[i_{\text{nb}}], z_{\min})$;
7: $M[i_{\text{nb}}] \leftarrow H(g_{\text{old}} - g_{\text{new}})$;
8: $I_{\text{sub}} \leftarrow M \odot (-1) + (1 - M) \odot I_{\text{sub}}$;
9: **return** I_{sub}
10: $I_{\text{new}} \leftarrow \text{vmap}(f_{\text{op1}})(\mathbf{I}_{\text{nb}}, \mathbf{F}_2)$;
11: **Function** $f_{\text{op2}}(i_{\text{new}}, \mathbf{x}, \mathbf{f}_1, \mathbf{w})$
12: $f \leftarrow \mathbf{1}_{i_{\text{new}}=-1} \odot \mathbf{F}_2 + (1 - \mathbf{1}_{i_{\text{new}}=-1}) \odot \mathbf{f}_1$;
13: $x \leftarrow \mathbf{1}_{i_{\text{new}}=-1} \odot \mathbf{O} + (1 - \mathbf{1}_{i_{\text{new}}=-1}) \odot \mathbf{x}$;
14: $i \leftarrow \text{argmin}(f_{\text{PBI}}(f, \mathbf{w}, z_{\min}))$;
15: **return** $\mathbf{x}[i], f[i]$
16: $\mathbf{X}_{\text{next}}, \mathbf{F}_{\text{next}} \leftarrow \text{vmap}(f_{\text{op2}})(I_{\text{new}}^\top, \mathbf{X}, \mathbf{F}_1, \mathbf{w})$;
17: $z \leftarrow z_{\min}$;

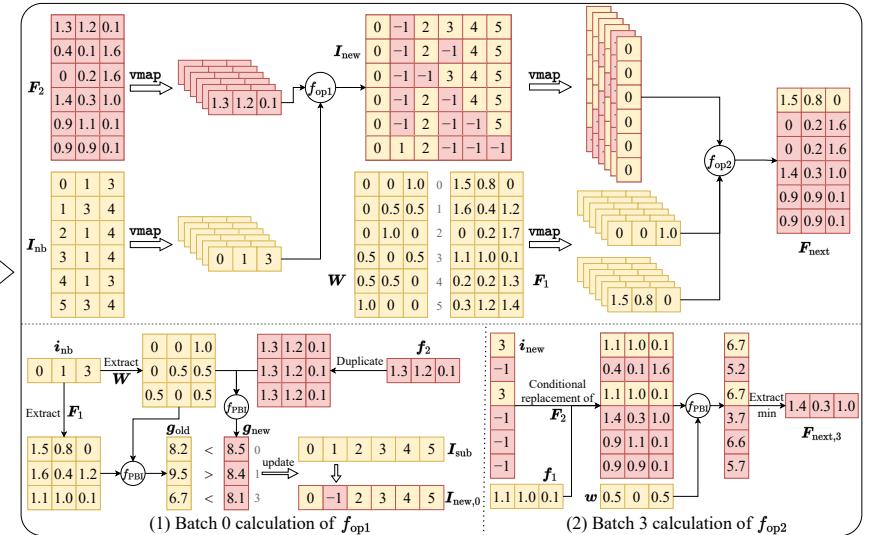


Fig. 1: Overview of the environmental selection in the TensorMOEA/D algorithm. **Left:** Pseudocode of the algorithm. **Right:** Tensor dataflow of module (1) and module (2). The upper part of the right figure shows the overall tensor dataflow for modules (1) and (2), while the lower part presents the batch calculation tensor dataflow, with module (1) on the left and module (2) on the right.

where $a = [0, 1, \dots, n - 1]$. The final solution tensors \mathbf{X}_{next} and \mathbf{F}_{next} are formed by selecting individuals using the sorted indices: $\mathbf{X}_{\text{next}} = \mathbf{X}[i_{\text{next}}]$ and $\mathbf{F}_{\text{next}} = \mathbf{F}[i_{\text{next}}]$.

Algorithm 1 Main Framework of Original MOEA/D

Input: The maximal number of generations t_{\max} ; n weight vectors; the neighborhood size T of each weight vector;

Output: Final population;

- 1: **Initialization;**
- 2: **for** $t = 1$ to t_{\max} **do**
- 3: **for** $i = 1$ to n **do**
- 4: Reproduction;
- 5: Fitness Evaluation;
- 6: Ideal Point Update;
- 7: Neighborhood Update;
- 8: **end for**
- 9: **end for**

B. Tensorized Environmental Selection in MOEA/D

In the original MOEA/D algorithm, as shown in Algorithm 1, the reproduction, fitness evaluation, ideal point update, and neighborhood update are executed *sequentially* within a single loop. This method requires processing individuals one by one in a specific order, which can significantly impede the execution speed of the algorithm. To address this limitation, we apply tensorization methodology to decouple these four steps in the inner loop (i.e., environmental selection), treating them as independent operations. This adjustment enables parallel processing of all individuals in the tensorized version, referred to as TensorMOEA/D.

In TensorMOEA/D, reproduction generates n individuals simultaneously based on the neighborhood, contrasting with the original MOEA/D, which produces one individual at a

time. The environmental selection process is further divided into two main steps: *comparison and population update*, and *elite selection*. These two steps are primarily implemented using two *vmap* operations, with the tensorization process detailed as follows. The pseudocode of both the original and tensorized algorithms is provided in Section S.III of the Supplementary Document.

1) *Comparison and Population Update:* The primary objective of this step is to determine the indices for the updated population by comparing the aggregated function values of the old and new populations. This process ultimately generates an updated index tensor \mathbf{I}_{new} , where each row corresponds to a subpopulation that mirrors the structure of the original population, containing indices of n individuals. Positions that require updates are indicated by -1 .

Given the solution tensor \mathbf{X} , objective tensor \mathbf{F}_1 , offspring tensor \mathbf{O} , the objective of offspring \mathbf{F}_2 , the ideal points \mathbf{z} , the weights \mathbf{W} , and the neighbors indices \mathbf{I}_{nb} , the process begins by calculating the minimal objective values to update the reference points, z_{\min} , by finding the minimum objective values between the current population and the offspring.

Next, the subpopulation indices \mathbf{I}_{sub} are created to track population updates. The *vmap* function is utilized to calculate the update indices \mathbf{I}_{new} :

$$\mathbf{I}_{\text{new}} = \text{vmap}(f_{\text{op1}})(\mathbf{I}_{\text{nb}}, \mathbf{F}_2), \quad (14)$$

where $f_{\text{op1}}(i_{\text{nb}}, f_2) = \mathbf{I}_{\text{sub}}$. This process is conducted in parallel by batching the rows of \mathbf{I}_{nb} and \mathbf{F}_2 , with i_{nb} and f_2 representing a single batch of these tensors. Each batch undergoes the f_{op1} operation, and all batches are processed simultaneously to yield results for all updates. A visual representation of this operation can be found in Fig. 1, with an example for the first batch illustrated in the lower left corner of the tensor dataflow on the right.

In the f_{op1} function, i_{nb} is used as an index to extract the corresponding rows from \mathbf{F}_1 and \mathbf{W} , and the entries of f_2 are replicated to obtain the inputs needed for the aggregation function. The aggregation function employs the penalty-based boundary intersection (PBI) function:

$$f_{\text{PBI}}(\mathbf{f}, \mathbf{w}, \mathbf{z}) = d_1 + \theta \cdot d_2, \quad (15)$$

where $d_1 = \frac{\|(\mathbf{f} - \mathbf{z})^\top \cdot \mathbf{w}\|}{\|\mathbf{w}\|}$, $d_2 = \|(\mathbf{f} - \mathbf{z}) - (d_1 \cdot \mathbf{w})\|$, and θ is a preset penalty parameter. The old and new aggregated values \mathbf{g}_{old} and \mathbf{g}_{new} are computed by applying the PBI function. The mask \mathbf{M} stores the comparison results and is initially a zero tensor, updated as follows:

$$\mathbf{M}[i_{\text{nb}}] = H(\mathbf{g}_{\text{old}} - \mathbf{g}_{\text{new}}). \quad (16)$$

Finally, the subpopulation indices \mathbf{I}_{sub} are updated based on \mathbf{M} :

$$\mathbf{I}_{\text{sub}} = \mathbf{M} \odot (-1) + (1 - \mathbf{M}) \odot \mathbf{I}_{\text{sub}}, \quad (17)$$

where positions that need updates are assigned a value of -1 .

2) Elite Selection: The purpose of this step is to select the best individuals along n distinct weighted directions based on the aggregated function values. Each direction yields a single elite individual, resulting in a total of n individuals that form the population of next generation.

To efficiently update the solution and objective tensor for the entire population, the function `vmap` is applied to parallelize the computation over all rows of the input:

$$\mathbf{X}_{\text{next}}, \mathbf{F}_{\text{next}} = \text{vmap}(f_{\text{op2}})(\mathbf{I}_{\text{new}}^\top, \mathbf{X}, \mathbf{F}_1, \mathbf{W}), \quad (18)$$

where `vmap` maps the function $f_{\text{op2}}(i_{\text{new}}, \mathbf{x}, \mathbf{f}_1, \mathbf{w})$ to each row of the provided inputs in parallel. Fig. 1 shows this process in the bottom-right corner, highlighting the computation for batch 3 (i.e., the 4th row).

The function f_{op2} is defined to update the population and objective tensor based on the new indices i_{new} , \mathbf{f}_1 , \mathbf{x} , and \mathbf{w} . Specifically, the objective values \mathbf{f} are updated as $\mathbf{f} = \mathbb{1}_{i_{\text{new}}=-1} \odot \mathbf{F}_2 + (1 - \mathbb{1}_{i_{\text{new}}=-1}) \odot \mathbf{f}_1$. And the individual is updated as $\mathbf{x} = \mathbb{1}_{i_{\text{new}}=-1} \odot \mathbf{O} + (1 - \mathbb{1}_{i_{\text{new}}=-1}) \odot \mathbf{x}$. The index $i = \text{argmin}(f_{\text{PBI}}(\mathbf{f}, \mathbf{w}, z_{\text{min}}))$ is used to select the best solution in \mathbf{x} . Finally, the ideal points \mathbf{z} are updated to the minimum objective values z_{min} for the next iteration.

C. Tensorized Environmental Selection in HypE

The environmental selection in HypE primarily involves nondominated sorting and HV calculation. In this article, we utilize the Monte Carlo estimation method for HV calculation that is well-suited for tensorization. This Monte Carlo-based HV calculation consists of four key steps: *sampling bound determination*, *sampling weights calculation*, *dominance score and distance update*, and *hypervolume calculation*. The implementation details for these steps are as follows. The pseudocode of both the original and tensorized algorithms are provided in Section S.IV of the Supplementary Document.

1) Sampling Bound Determination: The lower and upper bounds for sampling are determined by calculating the minimum objective values, $\mathbf{f}_l = \min_i(\mathbf{F})$, and using the reference point, $\mathbf{f}_u = \mathbf{v}_{\text{ref}}$. These bounds define the hyperrectangle for sampling. Next, uniformly sample points from the hyperrectangle defined by \mathbf{f}_l and \mathbf{f}_u to generate a sample tensor \mathbf{S} of dimension $s \times m$. The initial distance score \mathbf{v}_{ds} is then initialized to zeros with dimensions $1 \times s$.

2) Sampling Weights Calculation: The sampling weights $\boldsymbol{\alpha}$ are calculated as follows:

$$\alpha_j = \prod_{i=1}^j \lambda_i / j, \quad j = 1, 2, \dots, k, \quad (19)$$

where $\boldsymbol{\lambda}$ is defined as $\boldsymbol{\lambda} = [1, (k-l)/(n_1-l)]$. Here, $l = [i \mid i \in \mathbb{N}, 1 \leq i < n_1]$, and n_1 is the number of rows in \mathbf{F} .

3) Dominance Score and Distance Updating: Calculate the dominance scores for each sample i using the function f_{pds} :

$$f_{\text{pds}}(\mathbf{f}) = \mathbb{1}_{\sum_{j=1}^m H(\mathbf{S}_{ij} - \mathbf{f}) = m}, \quad (20)$$

where $i = 1, 2, \dots, s$. The dominance scores \mathbf{T}_{pds} are then computed by applying `vmap` in parallel to the function f_{pds} across all rows of the objective tensor \mathbf{F} .

The distance score \mathbf{v}_{ds} is then updated based on a temporary matrix \mathbf{T}_{temp} , which is computed as \mathbf{T}_{pds} combined with \mathbf{v}_{ds} and a tensor of ones $\mathbb{1}_{n \times 1}$:

$$\mathbf{v}_{\text{ds}} = \text{maximum} \left(\sum_{i=1}^{n_1} (\mathbf{T}_{\text{temp}})_i - 1, 0 \right), \quad (21)$$

where \mathbf{T}_{temp} is calculated as $\mathbf{T}_{\text{pds}} \odot (\mathbb{1}_{n \times 1} \cdot \mathbf{v}_{\text{ds}} + 1) + (1 - \mathbf{T}_{\text{pds}}) \odot (\mathbb{1}_{n \times 1} \cdot \mathbf{v}_{\text{ds}})$, and $\text{maximum}(\cdot, 0)$ is an element-wise operation that compares each element with 0, returning the element itself if it is greater than or equal to 0, and 0 otherwise.

4) Hypervolume Calculation: The HV contributions are calculated using the function f_{hv} , which sums the contributions from each sample:

$$f_{\text{hv}}(\mathbf{t}_{\text{pds}}) = \sum_{i=1}^s (\boldsymbol{\alpha}[\delta] \odot \mathbb{1}_{t_{\text{pds}} \neq -1})_i, \quad (22)$$

where $\delta = \mathbf{t}_{\text{pds}} \odot \mathbf{v}_{\text{ds}} - (1 - \mathbf{t}_{\text{pds}})$. This function is then applied in parallel to each row of the point dominance score tensor \mathbf{T}_{pds} using `vmap`, which efficiently computes the HV contributions across all samples.

Finally, the total HV v_{hv} is obtained by aggregating these contributions and normalizing:

$$v_{\text{hv}} = v_{\text{hv}} \cdot \prod_{i=1}^m (v_{\text{ref},i} - f_{l,i}) / s. \quad (23)$$

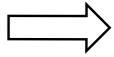
For other indicator-based EMO algorithms, the indicator is typically calculated using mathematical expressions, which facilitates straightforward tensorization. In the case of more complex indices, such as HV, Monte Carlo sampling can be employed to enhance algorithm efficiency by approximating these values. Additionally, to improve efficiency and scalability, the niche selection phase can use a one-shot method [12] for parallel selection.

Algorithm Tensorized Non-dominated Sorting

Input: Objective tensor $\mathbf{F} \in \mathbb{R}^{2n \times m}$ and population size n .

Output: Nondomination rank tensor \mathbf{r} and the last rank $l \in \mathbb{N}$.

- 1: $\mathbf{D} \leftarrow [\mathbf{F}_i \prec \mathbf{F}_j]_{i,j}, \quad i, j = 1, 2, \dots, 2n;$
- 2: $\mathbf{c} \leftarrow \sum_{j=1}^{2n} \mathbf{D}_{ij}, \quad i = 1, 2, \dots, 2n;$
- 3: $\mathbf{r} \leftarrow \mathbf{0}_{2n \times 1};$
- 4: $\mathbf{p} \leftarrow \mathbf{1}_{c=0};$
- 5: $k \leftarrow 0;$
- 6: **while** $\text{any}(\mathbf{p})$ **do**
- 7: $\mathbf{r} \leftarrow H(\mathbf{p}) \cdot k + H(1 - \mathbf{p}) \odot \mathbf{r};$
- 8: $\mathbf{d}_j \leftarrow \sum_{i=1}^{2n} (\mathbf{p}_i \cdot \mathbf{D}_{ij}), \quad j = 1, 2, \dots, 2n;$
- 9: $\mathbf{c} \leftarrow \mathbf{c} - \mathbf{d} - \mathbf{p};$
- 10: $k \leftarrow k + 1;$
- 11: $\mathbf{p} \leftarrow \mathbf{1}_{c=0};$
- 12: **end while**
- 13: $l \leftarrow \text{sort}(\mathbf{r})[n];$



```

1 import torch
2
3 def dominate_relation(X, Y):
4     X = X.unsqueeze(1)
5     Y = Y.unsqueeze(0)
6     return (X <= Y).all(dim=2) & (X < Y).any(dim=2)
7
8
9 def non_dominated_sort(F, n):
10    D = dominate_relation(F, F)
11    c = torch.sum(D, dim=0)
12    r = torch.zeros(F.size(0), dtype=torch.int32)
13    p = c == 0
14    k = 0
15
16    while p.any():
17        r = torch.where(p, c, r)
18        d = torch.sum(p.unsqueeze(1) * D, dim=0)
19        c = c - d - p
20        k += 1
21        p = c == 0
22
23    l = torch.sort(r)[n]
24
return r, l

```

Fig. 2: The seamless transformation of the tensorized nondominated sorting from pseudocode (Left) to Python code (Right).

D. Discussion

The tensorization methodology offers unique transformative benefits in EMO algorithm design and implementation. One of the key advantages is the seamless transformation of EMO algorithms from mathematical formulations into efficient code implementations. This bridge between algorithm design and programming is particularly valuable in GPU computing, where tensor operations can be leveraged for significant performance gains. For instance, Fig. 2 illustrates how pseudocode for tensorized nondominated sorting can be directly translated into Python code. This straightforward translation reduces the gap between high-level algorithmic design and practical implementation, thereby enabling practitioners to focus more on the theoretical aspects without the burden of intricate code optimization.

Tensorization also provides a high level of conciseness, significantly reducing code complexity compared to traditional iterative pseudocode. By representing population-wide operations as single tensor expressions, tensorization minimizes the need for loops and conditional statements, making the codebase more compact and readable. This conciseness not only eases code maintenance but also reduces the risk of programming errors by limiting procedural complexity.

Moreover, tensorization facilitates reproducibility in the field of EMO. As tensorized code relies on structured mathematical expressions, it becomes easier for researchers and developers to replicate results and benchmark different methods. This standardization paves the way for creating robust and high-performance libraries that can be shared and reused across various applications, thereby ultimately advancing research and industrial applications in EMO.

V. MULTI OBJECTIVE ROBOT CONTROL BENCHMARK

Traditional EMO benchmarks, such as ZDT [60], DTLZ [61], WFG [62], LSMOP [63], and MaF [64], primarily focus on numerical optimization problems. While these benchmarks are effective for evaluating the basic mechanisms

of EMO algorithms, they are limited in their ability to leverage hardware acceleration, thereby reducing their relevance in GPU computing environments.

In contrast, multiobjective robot control tasks present more realistic and computationally intensive challenges that better reflect real-world applications, which is particularly significant in the emerging area of *embodied artificial intelligence* (Embodied AI) [65]. These tasks provide complex and dynamic environments where multiple objectives must be balanced, such as energy efficiency and stability, making them well-suited for testing the adaptability and robustness of EMO algorithms. However, these tasks have been underexplored in the EMO community due to a lack of suitable benchmarks and the significant computational cost of running these environments.

To address this gap, we introduce the multiobjective robot control benchmark test suite, dubbed *MoRobtrol*, which reformulates nine tasks from the Brax environment [17] into MOPs. As a GPU-accelerated physics simulation engine, Brax provides a substantial performance improvement over CPU-based platforms such as OpenAI Gym [66] and Mo-Gymnasium [67]. By leveraging Brax's GPU computing capabilities, MoRobtrol enables scalable and rapid evaluations, making it an ideal benchmark for testing EMO algorithms in computationally demanding settings in practice.

As illustrated in Fig. S.1, the MoRobtrol benchmark includes nine robot control tasks: MoHalfcheetah, MoHopper, MoSwimmer, MoInvertedDoublePendulum (MoIDP), MoWalker2d, MoPusher, MoReacher, MoHumanoid, and MoHumanoidStandup (MoHumanoid-s). These tasks involve optimizing multiple conflicting objectives, such as speed, energy consumption, and distance to target, reflecting trade-offs commonly encountered in robotics applications. Specifically, in MoRobtrol, the parameters being optimized are the weights of a multilayer perceptron (MLP), a common design in control policy modeling for evolutionary reinforcement learning (EvoRL) [68]. These parameters are optimized by EMO algorithms to enable agents to maximize performance

TABLE III
OVERVIEW OF MULTIOBJECTIVE ROBOT CONTROL PROBLEMS IN THE PROPOSED MOROBTROL BENCHMARK TEST SUITE

Problem	MLP architecture [†]	d	m
MoHalfcheetah	$17 \times 16 \times 6$	390	f_v, f_c
MoHopper	$11 \times 16 \times 3$	243	f_v, f_h, f_c
MoSwimmer	$8 \times 16 \times 2$	178	f_v, f_c
MoIDP	$8 \times 16 \times 1$	161	f_{dp}, f_{sp}
MoWalker2d	$17 \times 16 \times 6$	390	f_v, f_c
MoPusher	$23 \times 16 \times 7$	503	f_n, f_d, f_c
MoReacher	$11 \times 16 \times 2$	226	f_d, f_c
MoHumanoid	$244 \times 16 \times 17$	4209	f_v, f_c
MoHumanoid-s	$244 \times 16 \times 17$	4209	f_v, f_c

[†] All MLP networks use the \tanh activation function.

across conflicting objectives.

Table III provides an overview of the nine tasks, including the MLP structure, number of parameters (d), and specific objectives for each task. The key objectives across tasks include forward reward (f_v), control cost (f_c), height (f_h), distance penalty (f_{dp}), speed penalty (f_{sp}), distance reward (f_d), and near reward (f_n). The number of objectives m varies depending on the task, allowing for detailed evaluations of algorithmic performance in diverse, real-world-inspired control scenarios. Detailed mathematical definitions are provided in Section S.VI of the Supplementary Document.

VI. EXPERIMENTAL STUDY

In this section, we conduct experiments to evaluate the performance of the tensorized EMO algorithms, including TensorNSGA-III, TensorMOEA/D, and TensorHypE. The experiments are categorized into three main aspects: acceleration performance, benchmarking on standard EMO test problems, and evaluation in multiobjective robot control tasks. All experiments are conducted on an RTX 4090 GPU server with AMD EPYC 7543 CPUs using the EvoX [56] framework.

A. Acceleration Performance

To verify the acceleration performance of the three proposed algorithms, we have conducted two sub-experiments. The first sub-experiment doubles the population size and observes the average runtime per generation. The second sub-experiment doubles the problem dimension and observes the average runtime per generation. We compare the performance of NSGA-III, MOEA/D, and HypE before and after tensorization on both CPU and GPU platforms using the DTLZ1 [61] problem.

Additionally, further experiments have been conducted to compare the performance of the tensorized algorithms in comparison with CUDA-accelerated algorithms in EvoTorch [69], as well as to investigate the impact of different types of GPUs on performance. Detailed results of these experiments can be found in Section S.VII-B and Section S.VII-C of Supplementary Document, respectively.

1) *Experimental Settings:* In the two sub-experiments, the tensorized and non-tensorized¹ algorithms are independently repeated 10 times on both CPU and GPU devices, with each algorithm evolving for 100 generations. The average runtime

¹The non-tensorized algorithms are partially tensorized for efficiency, as the original versions are time-consuming for large populations.

per generation is then calculated. In the first sub-experiment, the DTLZ1 problem has a dimension $d = 500$, with $m = 3$, and the population size n is doubled incrementally from 128 to 32768. In the second sub-experiment, the DTLZ1 problem is configured with $m = 3$, $n = 100$, and d is incrementally doubled from 1024 to 1048576.

2) *Comparison Results:* Fig. 3 shows that TensorNSGA-III, TensorMOEA/D, and TensorHypE consistently achieve faster runtimes on GPU compared to their non-tensorized versions on CPUs. When the population size n reaches 32768, TensorNSGA-III, TensorMOEA/D, and TensorHypE attain speedups of approximately 191 \times , 1113 \times , and 186 \times , respectively, compared to their CPU-based counterparts. As the problem dimension increases to 1048576, these speedups rise to 304 \times , 228 \times , and 263 \times . Although the runtime of tensorized algorithms may increase with n , they consistently outperform the non-tensorized versions. Additionally, as problem dimensions scale up, the runtime of tensorized algorithms remains relatively stable, maintaining a significant performance advantage over the original algorithms.

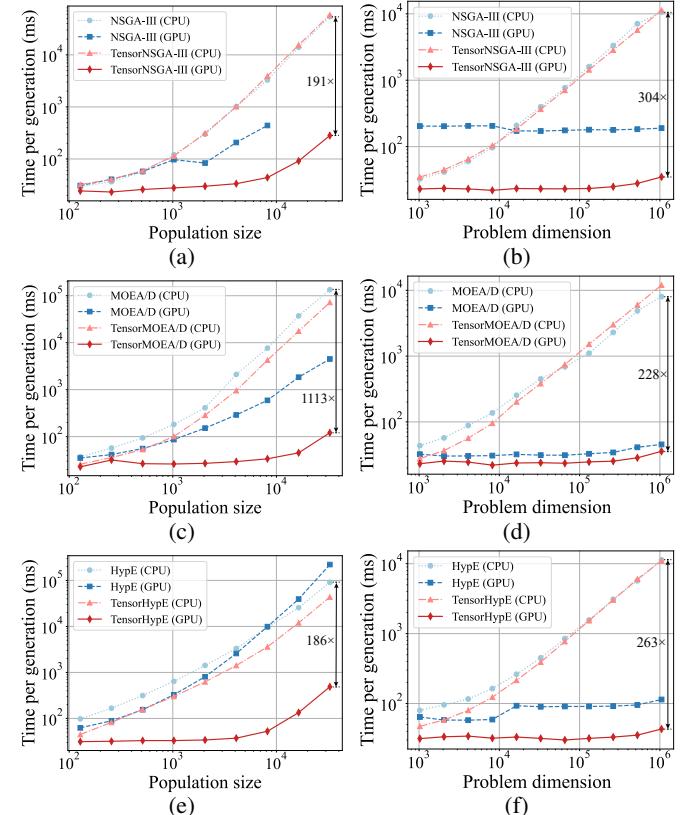


Fig. 3: Comparative acceleration performance of NSGA-III, MOEA/D, and HypE with their tensorized counterparts on CPU and GPU platforms. (a) and (b) illustrate performance for NSGA-III, (c) and (d) for MOEA/D, and (e) and (f) for HypE, across varying population sizes and problem dimensions.

Notably, when n exceeds 16384, the runtime for NSGA-III on GPU surpasses the preset threshold of 5 hours, resulting in missing data points for larger populations. For the HypE algorithm, once n exceeds 1024, its average runtime per generation on GPU begins to exceed that of TensorHypE on

TABLE IV

STATISTICAL RESULTS (MEAN AND STANDARD DEVIATION) OF THE IGD AND RUNTIME (S) FOR NON-TENSORIZED AND TENSORIZED EMO ALGORITHMS IN LSMOP1–LSMOP9. ALL EXPERIMENTS ARE ON AN RTX 4090 GPU AND THE BEST RESULTS ARE HIGHLIGHTED.

Algorithm	Problem	IGD (Non-Tensorized)	IGD (Tensorized)	Time (Non-Tensorized)	Time (Tensorized)
NSGA-III	LSMOP1	8.2378e-01 (5.2173e-03)	1.0399e+00 (1.2327e-01)	3.5048e+03 (2.7933e+00)	7.8573e+00 (2.0971e-01)
	LSMOP2	3.5883e-01 (5.9506e-06)	3.5883e-01 (2.0165e-05)	6.0457e+01 (1.2916e+00)	1.6127e+00 (1.8569e-02)
	LSMOP3	1.4106e+00 (7.5580e-02)	7.3244e+00 (5.3558e-01)	6.1996e+01 (1.6457e+00)	2.4987e+00 (3.4992e-02)
	LSMOP4	5.9798e-01 (1.5968e-04)	5.9815e-01 (2.0699e-04)	6.5350e+01 (1.8958e+00)	1.7237e+00 (1.5717e-02)
	LSMOP5	5.6862e-01 (2.3841e-03)	6.4269e-01 (6.6591e-03)	6.0026e+01 (1.0193e+00)	7.0027e+00 (8.2602e-02)
	LSMOP6	2.8277e+00 (5.9075e-01)	2.3737e+01 (2.9479e+00)	6.3764e+01 (1.8220e+00)	3.5406e+00 (5.6013e-02)
	LSMOP7	1.8425e+00 (7.5222e-03)	1.8474e+00 (3.7155e-03)	6.2175e+01 (1.2735e+00)	5.0992e+00 (1.3959e-01)
	LSMOP8	3.0317e-01 (2.9719e-02)	3.5186e-01 (1.3261e-02)	6.6262e+01 (1.4221e+00)	5.6878e+00 (4.9742e-02)
	LSMOP9	7.4488e-01 (9.4755e-03)	7.6837e-01 (2.9024e-03)	6.1194e+01 (1.6353e+00)	3.4327e+00 (2.2911e-02)
MOEA/D	LSMOP1	7.3475e-01 (3.8276e-04)	7.4365e-01 (2.1169e-03)	9.4239e+01 (3.0883e+00)	5.4412e+00 (2.8179e-01)
	LSMOP2	3.5885e-01 (3.4131e-05)	3.5888e-01 (5.9703e-06)	9.9946e+01 (2.7744e+00)	5.7516e+00 (6.0279e-01)
	LSMOP3	1.0815e+00 (1.0631e-01)	8.2722e-01 (5.3689e-02)	1.0148e+02 (2.6072e+00)	7.0391e+00 (4.7843e-01)
	LSMOP4	5.9570e-01 (1.5407e-03)	5.9420e-01 (1.1813e-03)	1.0919e+02 (2.2606e+00)	7.1559e+00 (3.8251e-01)
	LSMOP5	4.0554e-01 (8.8890e-03)	3.7065e-01 (8.5792e-03)	9.7131e+01 (2.9509e+00)	7.1893e+00 (4.6098e-01)
	LSMOP6	1.6366e+00 (7.7256e-02)	1.8622e+00 (1.2612e-01)	9.6220e+01 (2.9977e+00)	6.8919e+00 (2.4831e-01)
	LSMOP7	1.5534e+00 (2.6469e-01)	1.7345e+00 (3.6746e-03)	1.0456e+02 (2.4456e+00)	7.0900e+00 (4.0993e-01)
	LSMOP8	2.1207e-01 (6.4136e-03)	2.0107e-01 (1.8164e-02)	1.0154e+02 (2.0132e+00)	7.1273e+00 (4.2536e-01)
	LSMOP9	5.1964e-01 (1.7163e-02)	5.0269e-01 (1.0548e-02)	1.0443e+02 (2.9910e+00)	7.2184e+00 (7.7100e-01)
HypE	LSMOP1	8.2757e-01 (1.2610e-03)	8.2747e-01 (1.0132e-03)	6.3918e+01 (6.1344e-01)	1.8814e+00 (9.9671e-02)
	LSMOP2	3.5912e-01 (5.5267e+00)	3.5912e-01 (4.9532e-05)	6.4140e+01 (4.4810e-01)	1.5211e+00 (2.8512e-02)
	LSMOP3	4.6906e+00 (3.2754e-01)	4.6906e+00 (3.2754e-01)	6.4090e+01 (4.1599e-01)	1.7588e+00 (4.3342e-02)
	LSMOP4	5.9841e-01 (2.3223e-04)	5.9847e-01 (2.5082e-04)	6.3900e+01 (2.4586e-01)	1.6785e+00 (4.7110e-02)
	LSMOP5	5.2160e-01 (1.3777e-03)	5.2153e-01 (1.6406e-03)	6.4216e+01 (2.9358e-01)	2.0728e+00 (1.6839e-02)
	LSMOP6	3.0407e+00 (1.2805e-01)	3.0267e+00 (1.1215e-01)	6.5536e+01 (3.2336e-01)	3.4693e+00 (7.3326e-02)
	LSMOP7	1.8223e+00 (2.5992e-03)	1.8220e+00 (2.5472e-03)	6.4099e+01 (2.5939e-01)	1.7594e+00 (2.9685e-02)
	LSMOP8	3.6062e-01 (9.5863e-03)	3.6060e-01 (9.5862e-03)	6.4537e+01 (3.3889e-01)	2.3856e+00 (2.8679e-02)
	LSMOP9	6.5873e-01 (3.0162e-03)	6.5797e-01 (3.4461e-03)	6.3643e+01 (2.6093e-01)	1.5741e+00 (1.2765e-02)

CPU, and for $n > 8192$, the GPU version of HypE becomes even slower than the original CPU-based implementation.

These performance drops can be attributed to two main factors. First, the original implementations of NSGA-III and HypE do not fully leverage the multicore parallel processing capabilities of GPUs, leading to underutilization of GPU cores. Second, the data transfer overhead between CPU and GPU further reduces efficiency, particularly for large population sizes where the NSGA-III and HypE algorithms incur higher computational costs on GPU than on CPU.

Additionally, the acceleration performance varies across tensorized algorithms. MOEA/D and HypE, due to their simpler operations and fewer conditional branches, achieve greater acceleration after tensorization compared to their original versions. Conversely, NSGA-III shows more limited acceleration, as its more complex operations involve intricate loops and branches, which are less amenable to GPU parallelization.

B. Performance in Numerical Optimization

To verify the precision before and after tensorization, the three proposed algorithms and their original versions are comprehensively tested on the LSMOP [63] and DTLZ [61] test suites. The detailed results for the DTLZ are provided in Section S.VII-D of Supplementary Document.

1) *Experimental Settings:* In this experiment, all algorithms are independently repeated 31 times on 9 LSMOP problems and 7 DTLZ problems. Each algorithm is run for 100 generations with a population size of 10000. Each problem has a dimension of 5000. Performance is measured using the average IGD [70] and average runtime over 31 runs. A Wilcoxon rank-sum test is used to compare tensorized and non-tensorized

algorithms. If the test shows no significant difference (i.e., $p > 0.05$), both performance indicators are highlighted in bold.

2) *Comparison Results:* Table IV demonstrates that the tensorized algorithms maintain comparable precision to their non-tensorized counterparts while achieving significantly faster runtimes. The average IGD between the tensorized and non-tensorized algorithms consistently remain within the same order of magnitude, indicating similar levels of solution quality. In some instances, the tensorized algorithms even exhibit superior indicator performance. Additionally, the average runtime for tensorized algorithms is consistently lower than that for non-tensorized algorithms, underscoring the efficiency gains enabled by tensorization. Notably, when comparing performance within equivalent time frames, tensorized algorithms consistently achieve better indicator performance, highlighting their effectiveness in optimizing both time and solution quality metrics.

However, some performance degradation is observed in certain cases. In TensorNSGA-III, batch random operations simulate the niche selection process of the original algorithm, which differs from the original method's precise operation on individual solutions. This discrepancy can lead to performance degradation, as the batch approach may not capture niche selection as effectively. Similarly, TensorMOEA/D adopts batch offspring generation and updates, which accelerate computation per generation but may result in lower performance for the same number of generations. Despite this, TensorMOEA/D consistently achieves or sometimes surpasses the performance of the original algorithm when comparing performance over equivalent time periods. As for HypE, since its method of calculating the HV is largely similar to that of the original

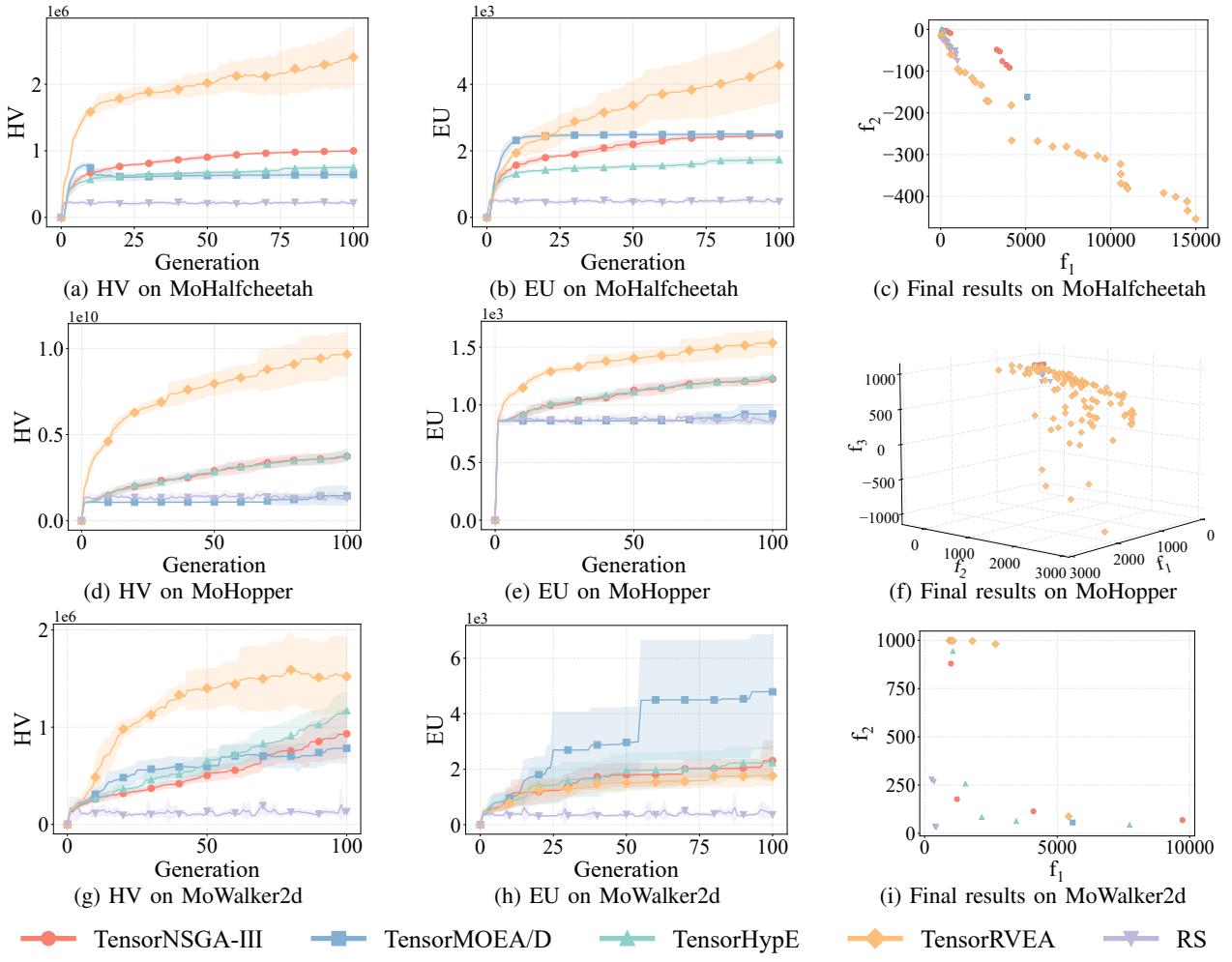


Fig. 4: Comparative performance (HV, EU, and visualization of final results) of TensorNSGA-III, TensorMOEA/D, TensorHypE, TensorRVEA, and random search (RS) across varying problems: MoHalfcheetah (390D), MoHopper (243D), and MoWalker2d (390D). Note: Higher values for all metrics indicate better performance.

algorithm, the primary difference lies in computation speed.

C. Performance in Multiobjective Robot Control Benchmark

In this experiment, we evaluate the advantages of tensorization and the effectiveness of the three proposed tensorized algorithms in solving the multiobjective robot control tasks using the proposed MoRobtrol benchmark test suite. Following the paradigm of EvoRL, the EMO algorithms evolve a population of MLP neural networks, with each MLP serving as a policy model within the simulation environment of each task.

1) *Experimental Settings:* In this experiment, we apply TensorNSGA-III, TensorMOEA/D, TensorHypE, TensorRVEA [57], and random search (RS) algorithms to solve 9 multiobjective robot control problems in MoRobtrol. Each algorithm is repeated 10 times with a population size of 10000. Performance is evaluated using HV [13], expected utility (EU) [71], and visualization of the final nondominated solutions. These indicators are calculated based on the nondominated solutions of each generation. Details on the TensorRVEA algorithm and the reference points for HV

calculation are provided in Sections S.V and S.VII-E of the Supplementary Document, respectively.

2) *Comparison Results:* As shown in Fig. 4, TensorRVEA achieves the highest HV on the MoHalfcheetah, MoHopper, and MoWalker2d problems, followed closely by TensorHypE and TensorNSGA-III, which show similar performance. TensorHypE outperforms TensorNSGA-III on MoWalker2d, while TensorNSGA-III performs better on MoHalfcheetah. For EU and final results, TensorRVEA demonstrates high EU scores and strong diversity in MoHalfcheetah and MoHopper. On MoWalker2d, TensorRVEA achieves better HV, while TensorMOEA/D scores higher in EU, indicating a better preference under uniform weights. TensorNSGA-III performs best on MoWalker2d for the first objective, achieving very high speeds. Although TensorMOEA/D does not perform as well as the other algorithms, it still significantly surpasses random search.

As shown in Fig. 5 and Fig. 6, TensorRVEA achieves higher HV on MoPusher and MoReacher, although its EU is lower than that of the other algorithms. On MoReacher, solutions from TensorNSGA-III, TensorMOEA/D, and TensorHypE

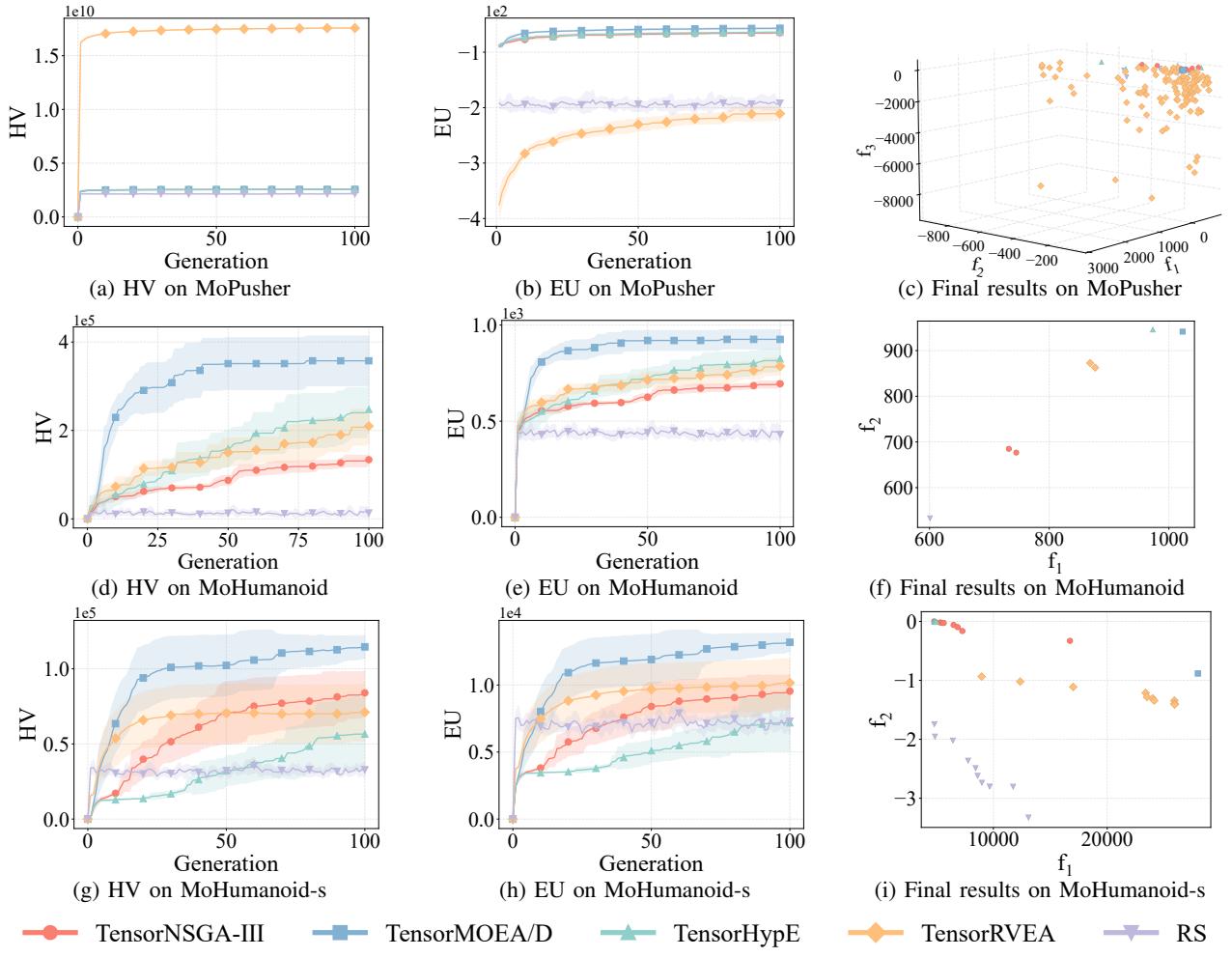


Fig. 5: Comparative performance (HV, EU, and visualization of final results) of TensorNSGA-III, TensorMOEA/D, TensorHypE, TensorRVEA, and random search (RS) across varying problems: MoPusher (503D), MoHumanoid (4209D), and MoHumanoid-s (4209D). Note: Higher values for all metrics indicate better performance.

dominate parts of TensorRVEA's solutions, though TensorRVEA still maintains superior HV. TensorMOEA/D performs significantly better on the MoHumanoid and MoHumanoid-s problems, highlighting its effectiveness in handling large-scale problems.

Another important observation is that TensorRVEA and TensorMOEA/D can exhibit an initial rise in HV, followed by a decline before stabilizing, as seen in problems like MoSwimmer in Fig. 6. This phenomenon can be attributed to the weight or reference tensors guiding the population to optimize in specific directions early in the process, leading to premature convergence and a subsequent decline in HV as diversity decreases. By contrast, on the MoIDP problem, TensorRVEA achieves the highest HV, although its EU and visualization results remain comparable to those of the other algorithms.

Overall, decomposition-based algorithms such as TensorRVEA and TensorMOEA/D exhibit superior performance in large-scale multiobjective robot control tasks, particularly in handling high-dimensional decision spaces and maintaining solution diversity.

VII. CONCLUSION

This article introduces a tensorization approach to address the computational limitations of traditional CPU-based EMO algorithms, enhancing both speed and scalability. We applied this approach across three representative EMO algorithm classes: 1) dominance-based (NSGA-III), 2) decomposition-based (MOEA/D), and 3) indicator-based (HypE), which demonstrated substantial performance improvements on GPU platforms. Our results confirm that tensorized algorithms can significantly accelerate computations while maintaining solution quality comparable to their original CPU-based counterparts.

To demonstrate the applicability of tensorized EMO algorithms in GPU computing environments, we also developed MoRobtrol, a comprehensive benchmark test suite that reformulates complex multiobjective robot control tasks from the physics simulation environments into MOPs. MoRobtrol underscores the potential of tensorized EMO algorithms to efficiently address the high computational demands of Embodied AI, illustrating their relevance to dynamic real-world applications.

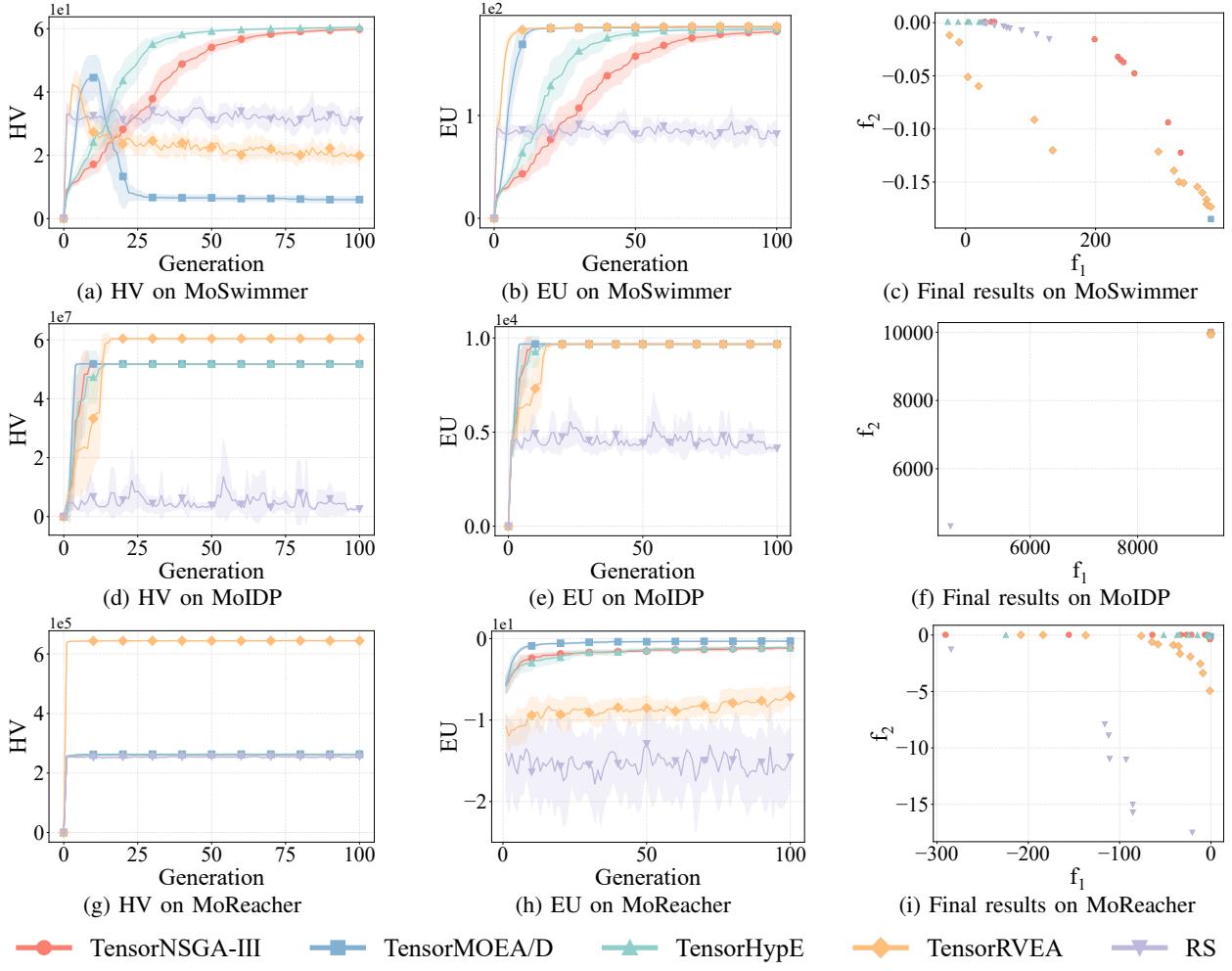


Fig. 6: Comparative performance (HV, EU, and visualization of final results) of TensorNSGA-III, TensorMOEA/D, TensorHypE, TensorRVEA, and random search (RS) across varying problems: MoSwimmer (178D), MoIDP (161D), and MoReacher (226D). Note: Higher values for all metrics indicate better performance.

While tensorization has substantially improved algorithmic efficiency, opportunities remain to further optimize speed and memory use. Future work will focus on refining key operators such as nondominated sorting and exploring new tensorized operators optimized for multi-GPU environments to maximize performance. Additionally, leveraging large population data to strengthen search strategies and integrating deep learning techniques may further extend the capabilities of EMO algorithms in tackling large-scale challenges.

REFERENCES

- [1] P. R. Wiecha, A. Arbouet, C. Girard, A. Lecestre, G. Larrieu, and V. Paillard, "Evolutionary multi-objective optimization of colour pixels based on dielectric nanoantennas," *Nature Nanotechnology*, vol. 12, no. 2, pp. 163–169, 2017.
- [2] B. Peng, Y. Wei, Y. Qin, J. Dai, Y. Li, A. Liu, Y. Tian, L. Han, Y. Zheng, and P. Wen, "Machine learning-enabled constrained multi-objective design of architected materials," *Nature Communications*, vol. 14, no. 1, p. 6630, 2023.
- [3] Y. Cui, Z. Geng, Q. Zhu, and Y. Han, "Multi-objective optimization methods and application in energy saving," *Energy*, vol. 125, pp. 681–704, 2017.
- [4] X. Wang, X. Mao, and H. Khodaei, "A multi-objective home energy management system based on internet of things and optimization algorithms," *Journal of Building Engineering*, vol. 33, p. 101603, 2021.
- [5] Z. Fei, B. Li, S. Yang, C. Xing, H. Chen, and L. Hanzo, "A survey of multi-objective optimization in wireless sensor networks: Metrics, algorithms, and open problems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 550–586, 2016.
- [6] A. Ponsich, A. L. Jaimes, and C. A. C. Coello, "A survey on multiobjective evolutionary algorithms for the solution of the portfolio optimization problem and other finance and economics applications," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 3, pp. 321–344, 2012.
- [7] K. Deb, K. Sindhya, and J. Hakanen, "Multi-objective optimization," in *Decision Sciences*. CRC Press, 2016, pp. 161–200.
- [8] B. Li, J. Li, K. Tang, and X. Yao, "Many-objective evolutionary algorithms: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, sep 2015.
- [9] 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, 2002.
- [10] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based non-dominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.
- [11] Qingfu Zhang and Hui Li, "MOEA/D: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, Dec. 2007.
- [12] J. Bader and E. Zitzler, "HypE: An algorithm for fast hypervolume-based many-objective optimization," *Evolutionary Computation*, vol. 19, no. 1, pp. 45–76, 03 2011.
- [13] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A

- comparative case study and the strength pareto approach,” *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
- [14] Y. Tian, L. Si, X. Zhang, R. Cheng, C. He, K. C. Tan, and Y. Jin, “Evolutionary large-scale multi-objective optimization: A survey,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, oct 2021.
- [15] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [16] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA Tensor Core programmability, performance & precision,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531.
- [17] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, “Brax - a differentiable physics engine for large scale rigid body simulation,” 2021. [Online]. Available: <http://github.com/google/brax>
- [18] E. Zitzler, M. Laumanns, and L. Thiele, “SPEA2: Improving the strength pareto evolutionary algorithm,” Computer Engineering and Networks Laboratory, ETH Zurich, Zürich, Switzerland, Tech. Rep. TIK-103, 2001.
- [19] S. Yang, M. Li, X. Liu, and J. Zheng, “A grid-based evolutionary algorithm for many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 5, pp. 721–736, 2013.
- [20] M. Li, S. Yang, and X. Liu, “Bi-goal evolution for many-objective optimization problems,” *Artificial Intelligence*, vol. 228, pp. 45–65, 2015.
- [21] Y. Xiang, Y. Zhou, M. Li, and Z. Chen, “A vector angle-based evolutionary algorithm for unconstrained many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 1, pp. 131–152, 2016.
- [22] C. He, Y. Tian, Y. Jin, X. Zhang, and L. Pan, “A radial space division based evolutionary algorithm for many-objective optimization,” *Applied Soft Computing*, vol. 61, pp. 603–621, 2017.
- [23] Y. Tian, R. Cheng, X. Zhang, Y. Su, and Y. Jin, “A strengthened dominance relation considering convergence and diversity for evolutionary many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 2, pp. 331–345, 2018.
- [24] Y. Tian, C. He, R. Cheng, and X. Zhang, “A multistage evolutionary algorithm for better diversity preservation in multiobjective optimization,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 9, pp. 5880–5894, 2019.
- [25] Y. Liu, N. Zhu, and M. Li, “Solving many-objective optimization problems by a Pareto-based evolutionary algorithm with preprocessing and a penalty mechanism,” *IEEE Transactions on Cybernetics*, vol. 51, no. 11, pp. 5585–5594, 2021.
- [26] K. Li, “A survey of multi-objective evolutionary algorithm based on decomposition: Past and future,” *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2024.
- [27] Q. Zhang, W. Liu, and H. Li, “The performance of a new version of MOEA/D on CEC09 unconstrained MOP test instances,” in *2009 IEEE Congress on Evolutionary Computation*, 2009, pp. 203–208.
- [28] X. Cai, Y. Li, Z. Fan, and Q. Zhang, “An external archive guided multi-objective evolutionary algorithm based on decomposition for combinatorial optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 4, pp. 508–523, 2015.
- [29] Q. Zhao, Y. Guo, X. Yao, and D. Gong, “Decomposition-based multiobjective optimization algorithms with adaptively adjusting weight vectors and neighborhoods,” *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 5, pp. 1485–1497, 2023.
- [30] Q. Wang, Q. Gu, L. Chen, Y. Guo, and N. Xiong, “A MOEA/D with global and local cooperative optimization for complicated bi-objective optimization problems,” *Applied Soft Computing*, vol. 137, p. 110162, 2023.
- [31] H.-L. Liu, F. Gu, and Q. Zhang, “Decomposition of a multiobjective optimization problem into a number of simple multiobjective subproblems,” *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 3, pp. 450–455, 2014.
- [32] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, “A reference vector guided evolutionary algorithm for many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 773–791, 2016.
- [33] Y. Yuan, H. Xu, B. Wang, and X. Yao, “A new dominance relation-based evolutionary algorithm for many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 16–37, 2016.
- [34] J. Zhou, L. Gao, and X. Li, “Ensemble of dynamic resource allocation strategies for decomposition-based multiobjective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 4, pp. 710–723, 2021.
- [35] J. G. Falcón-Cardona and C. A. C. Coello, “Indicator-based multi-objective evolutionary algorithms: A comprehensive survey,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, Mar. 2020.
- [36] E. Zitzler and S. Künzli, “Indicator-based selection in multiobjective search,” in *Parallel Problem Solving from Nature - PPSN VIII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 832–842.
- [37] M. Emmerich, N. Beume, and B. Naujoks, “An EMO algorithm using the hypervolume measure as selection criterion,” in *Evolutionary Multi-Criterion Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 62–76.
- [38] M. Li, S. Yang, and X. Liu, “Pareto or non-Pareto: Bi-criterion evolution in multiobjective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 645–665, 2016.
- [39] B. Li, K. Tang, J. Li, and X. Yao, “Stochastic ranking algorithm for many-objective optimization based on multiple indicators,” *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 6, pp. 924–938, 2016.
- [40] R. Hernández Gómez and C. A. Coello Coello, “Improved metaheuristic based on the R2 indicator for many-objective optimization,” in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO ’15. ACM, 2015, p. 679–686.
- [41] Y. Tian, R. Cheng, X. Zhang, F. Cheng, and Y. Jin, “An indicator-based multiobjective evolutionary algorithm with reference point adaptation for better versatility,” *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 4, pp. 609–622, 2018.
- [42] Y. Sun, G. Y. Yen, and Z. Yi, “IGD indicator-based evolutionary algorithm for many-objective optimization problems,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 2, pp. 173–187, 2019.
- [43] K. Shang and H. Ishibuchi, “A new hypervolume-based evolutionary algorithm for many-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 5, pp. 839–852, 2020.
- [44] L. M. Pang, H. Ishibuchi, L. He, K. Shang, and L. Chen, “Hypervolume-based cooperative coevolution with two reference points for multiobjective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 28, no. 4, pp. 1054–1068, 2024.
- [45] Z. Wang, K. Lin, G. Li, and W. Gao, “Multi-objective optimization problem with hardly dominated boundaries: Benchmark, analysis, and indicator-based algorithm,” *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2024.
- [46] M. L. Wong, “Parallel multi-objective evolutionary algorithms on graphics processing units,” in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, ser. GECCO ’09. ACM, 2009, pp. 2515–2522.
- [47] D. Sharma and P. Collet, “GPGPU-compatible archive based stochastic ranking evolutionary algorithm (G-ASREA) for multi-objective optimization,” in *Parallel Problem Solving from Nature, PPSN XI*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–120.
- [48] B. Arca, T. Ghisu, and G. A. Trunfio, “GPU-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard,” *Journal of Computational Science*, vol. 11, pp. 258–268, 2015.
- [49] A. Aguilar-Rivera, “A GPU fully vectorized approach to accelerate performance of NSGA-2 based on stochastic non-domination sorting and grid-crowding,” *Applied Soft Computing*, vol. 88, p. 106047, 2020.
- [50] M. Z. De Souza and A. T. R. Pozo, “A GPU implementation of MOEA/D-ACO for the multiobjective traveling salesman problem,” in *2014 Brazilian conference on intelligent systems*. IEEE, 2014, pp. 324–329.
- [51] E. M. Lopez, L. M. Antonio, and C. A. Coello Coello, “A GPU-based algorithm for a faster hypervolume contribution computation,” in *Evolutionary Multi-Criterion Optimization*. Cham: Springer International Publishing, 2015, pp. 80–94.
- [52] M. M. Hussain and N. Fujimoto, “GPU-based parallel multi-objective particle swarm optimization for large swarms and high dimensional problems,” *Parallel Computing*, vol. 92, p. 102589, 2020.
- [53] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” 2018. [Online]. Available: <http://github.com/google/jax>
- [54] Y. Tang, Y. Tian, and D. Ha, “EvoJAX: Hardware-accelerated neuroevolution,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO ’22. ACM, 2022, p. 308–311.
- [55] R. T. Lange, “evojax: JAX-based evolution strategies,” in *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, ser. GECCO ’23 Companion. ACM, 2023, p. 659–662.

- [56] B. Huang, R. Cheng, Z. Li, Y. Jin, and K. C. Tan, "EvoX: A distributed GPU-accelerated framework for scalable evolutionary computation," *IEEE Transactions on Evolutionary Computation*, 2024.
- [57] Z. Liang, T. Jiang, K. Sun, and R. Cheng, "GPU-accelerated evolutionary multiobjective optimization using tensorized RVEA," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '24, 2024, pp. 566–575.
- [58] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," ser. ISMM '07. ACM, 2007, p. 103–104.
- [59] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.
- [60] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation*, vol. 8, no. 2, pp. 173–195, 2000.
- [61] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable test problems for evolutionary multiobjective optimization," in *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications*. London: Springer London, 2005, pp. 105–145.
- [62] S. Huband, P. Hingston, L. Barone, and L. While, "A review of multiobjective test problems and a scalable test problem toolkit," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 5, pp. 477–506, 2006.
- [63] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, "Test problems for large-scale multiobjective and many-objective optimization," *IEEE Transactions on Cybernetics*, vol. 47, no. 12, pp. 4108–4121, 2017.
- [64] R. Cheng, M. Li, Y. Tian, X. Zhang, S. Yang, Y. Jin, and X. Yao, "A benchmark test suite for evolutionary many-objective optimization," *Complex & Intelligent Systems*, vol. 3, no. 1, pp. 67–81, Mar. 2017.
- [65] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh, and D. Batra, "Habitat: A platform for embodied AI research," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.
- [66] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.
- [67] F. Felten, L. N. Alegre, A. Nowé, A. L. C. Bazzan, E. G. Talbi, G. Danoy, and B. C. da. Silva, "A toolkit for reliable benchmarking and research in multi-objective reinforcement learning," in *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023)*, 2023.
- [68] H. Bai, R. Cheng, and Y. Jin, "Evolutionary reinforcement learning: A survey," *Intelligent Computing*, vol. 2, p. 0025, 2023.
- [69] N. E. Toklu, T. Atkinson, V. Micka, P. Liskowski, and R. K. Srivastava, "EvoTorch: Scalable evolutionary computation in Python," *arXiv preprint*, 2023, <https://arxiv.org/abs/2302.12600>.
- [70] C. A. C. Coello and N. C. Cortés, "Solving multiobjective optimization problems using an artificial immune system," *Genetic Programming and Evolvable Machines*, vol. 6, pp. 163–190, 2005.
- [71] L. M. Zintgraf, T. V. Kanters, D. M. Ruijters, F. Oliehoek, and P. Beau, "Quality assessment of MORL algorithms: A utility-based approach," in *Benelearn 2015: Proceedings of the 24th Annual Machine Learning Conference of Belgium and the Netherlands*, 2015.
- [72] Z.-H. Zhan, J. Zhang, Y. Lin, J.-Y. Li, T. Huang, X.-Q. Guo, F.-F. Wei, S. Kwong, X.-Y. Zhang, and R. You, "Matrix-based evolutionary computation," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 6, no. 2, pp. 315–328, 2022.
- [73] Y. Tian, R. Cheng, X. Zhang, and Y. Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Computational Intelligence Magazine*, vol. 12, no. 4, pp. 73–87, 2017.
- [74] P. Rajpoot and P. Dwivedi, "Matrix method for non-dominated sorting and population selection for next generation in multi-objective problem solution," in *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 2018, pp. 670–676.

Bridging Evolutionary Multiobjective Optimization and GPU Acceleration via Tensorization (Supplementary Document)

S.I. GENETIC OPERATORS IN EMO ALGORITHMS

This section provides a comprehensive overview of the genetic operators used in EMO algorithms, including mating selection, original crossover and mutation, and the tensorization of these crossover and mutation operations.

A. Mating Selection

Mating selection is a critical component of EMO algorithms, whereby individuals are selected from the existing population to act as parents for the offspring generation. Classical methods, including roulette wheel selection, tournament selection, and random selection, are commonly utilized in this context. These methods, which leverage probability and statistical principles to varying degrees for solution selection, are well-suited for tensorization.

B. Original Crossover and Mutation

In EMO algorithms, genetic operators play a pivotal role in both the exploration and exploitation of the search space. Among the most commonly used operators in genetic algorithms tailored for real-coded variables are simulated binary crossover (SBX) and polynomial mutation. These operators are specifically engineered to maintain a balance between exploration and exploitation, proving particularly effective for problems characterized by continuous decision variables. The conventional implementation is as follows.

1) *Simulated Binary Crossover:* SBX operates under the principle of creating offspring that are close to the parent solutions, analogous to the single-point crossover used in binary-coded genetic algorithms. Given two parent solutions $\mathbf{x}_1 = (x_1^1, x_1^2, \dots, x_1^d)$ and $\mathbf{x}_2 = (x_2^1, x_2^2, \dots, x_2^d)$, SBX generates two offspring \mathbf{c}_1 and \mathbf{c}_2 using the following equations:

$$\mathbf{c}_1^i = 0.5 [(1 + \beta)\mathbf{x}_1^i + (1 - \beta)\mathbf{x}_2^i], \quad (\text{S.1})$$

$$\mathbf{c}_2^i = 0.5 [(1 - \beta)\mathbf{x}_1^i + (1 + \beta)\mathbf{x}_2^i], \quad (\text{S.2})$$

where i is the index of each decision variable, β is the spread factor, calculated based on a probability distribution and a user-defined parameter η_c , typically known as the distribution index. The spread factor β is calculated as follows:

$$\beta = \begin{cases} (2\mu)^{\frac{1}{\eta_c+1}}, & \mu \leq 0.5 \\ \left(\frac{1}{2(1-\mu)}\right)^{\frac{1}{\eta_c+1}}, & \mu > 0.5 \end{cases}, \quad (\text{S.3})$$

where μ is a random number with values between 0 and 1.

2) *Polynomial Mutation:* Polynomial Mutation is designed to introduce minor perturbations in the offspring, promoting diversity in the population. For a given parent solution \mathbf{x} , the mutated solution \mathbf{y} is generated as follows:

$$\mathbf{y}^i = \mathbf{x}^i + \bar{\delta}(\mathbf{u}^i - \mathbf{l}^i), \quad (\text{S.4})$$

where \mathbf{u}^i and \mathbf{l}^i are the upper and lower bounds of the i -the decision variable, respectively, and $\bar{\delta}$ is the mutation step size. The value of $\bar{\delta}$ is determined by:

$$\bar{\delta} = \begin{cases} [2\mu + (1 - 2\mu)(1 - \delta_1)^\eta]^{\frac{1}{\eta}} - 1, & \mu \leq 0.5 \\ 1 - [2 - 2\mu + (2\mu - 1)(1 - \delta_2)^\eta]^{\frac{1}{\eta}}, & \mu > 0.5 \end{cases}, \quad (\text{S.5})$$

where $\eta = \eta_m + 1$, with η_m being the distribution index for mutation. The term μ represents a uniformly distributed random number within the range $[0,1]$. Additionally, $\delta_1 = \frac{\mathbf{x}^i - \mathbf{l}^i}{\mathbf{u}^i - \mathbf{l}^i}$ and $\delta_2 = \frac{\mathbf{u}^i - \mathbf{x}^i}{\mathbf{u}^i - \mathbf{l}^i}$ define the normalized differences relevant to the mutation process.

C. Tensorization of Crossover and Mutation

Inspired by the matrix-based evolutionary computation framework [72], which represents the entire population as a matrix to enable efficient parallel execution of evolutionary operators, we adopt a similar tensorization strategy to accelerate crossover and mutation in EMO. The tensorized methods implemented in TensorRVEA [57] and PlatEMO [73] facilitate the concurrent processing of genetic information across a population, significantly enhancing the speed and scalability of genetic operations within EMO frameworks. This demonstrates the efficacy of tensorization in contemporary computational environments. The tensorized implementation of crossover and mutation operators is as follows.

1) *Simulated Binary Crossover:* In a tensorized form, the crossover operation can be performed in parallel across the population. For instance, in the SBX, the tensorized population $\mathbf{X} \in \mathbb{R}^{n \times d}$ is divided into two parent tensors, \mathbf{X}_1 and \mathbf{X}_2 , each with dimension $\lfloor \frac{n}{2} \rfloor \times d$. The tensorized formulations for SBX are as follows:

$$\mathbf{X}_c = \begin{bmatrix} [(1 + \mathbf{B}) \odot \mathbf{X}_1 + (1 - \mathbf{B}) \odot \mathbf{X}_2]/2 \\ [(1 - \mathbf{B}) \odot \mathbf{X}_1 + (1 + \mathbf{B}) \odot \mathbf{X}_2]/2 \end{bmatrix}, \quad (\text{S.6})$$

where \mathbf{B} is a tensor of spread factors computed as:

$$\mathbf{B} = (2\mathbf{M})^{\frac{1}{\eta_c+1}} \odot H(0.5 - \mathbf{M}) + (1/(2 - 2\mathbf{M}))^{\frac{1}{\eta_c+1}} \odot (1 - H(0.5 - \mathbf{M})), \quad (\text{S.7})$$

where η_c is the distribution parameter of SBX, \mathbf{M} is a tensor with the same dimension as \mathbf{X}_1 and \mathbf{X}_2 , containing uniformly distributed random numbers in [0,1]. The \odot operator denotes the Hadamard product, which performs element-wise multiplication. The $H(\cdot)$ function is the Heaviside step function, returning 1 for non-negative inputs and 0 otherwise.

2) *Polynomial Mutation:* Mutation operations introduce small random changes to solutions. In a tensorized form, mutation can be performed in parallel. For polynomial mutation:

$$\mathbf{X}_m = \mathbf{X}_c + \overline{\Delta} \odot (\mathbf{U} - \mathbf{L}), \quad (\text{S.8})$$

where the tensors \mathbf{U} and \mathbf{L} represent the upper and lower bounds of the population, respectively. \mathbf{X}_c is the population after crossover. $\overline{\Delta}$ is a size tensor of mutation steps, computed as:

$$\overline{\Delta}_1 = [2\mathbf{M} + (1 - 2\mathbf{M}) \odot (1 - \Delta_1)^\eta]^{\frac{1}{\eta}}, \quad (\text{S.9})$$

$$\overline{\Delta}_2 = 1 - [2 - 2\mathbf{M} + (2\mathbf{M} - 1) \odot (1 - \Delta_2)^\eta]^{\frac{1}{\eta}}, \quad (\text{S.10})$$

$$\overline{\Delta} = \overline{\Delta}_1 \odot H(0.5 - \mathbf{M}) + \overline{\Delta}_2 \odot (1 - H(0.5 - \mathbf{M})), \quad (\text{S.11})$$

where $\eta = \eta_m + 1$, and η_m is the distribution index, \mathbf{M} consists of uniformly distributed random numbers within [0,1], and H is the Heaviside step function. Additionally, $\Delta_1 = (\mathbf{X}_c - \mathbf{L}) / (\mathbf{U} - \mathbf{L})$ and $\Delta_2 = (\mathbf{U} - \mathbf{X}_c) / (\mathbf{U} - \mathbf{L})$ are transformation tensors for normalizing crossover transformations, similar to mutation processes.

S.II. TENSORIZATION OF NSGA-III

This section compares the original NSGA-III algorithm with its tensorized version, focusing on the environmental selection process. The pseudocode for both implementations is provided to demonstrate the key differences and the efficiency gains achieved through tensorization methodology.

A. Original Environmental Selection

In the original NSGA-III algorithm, environmental selection consists of several key steps. The process begins with nondominated sorting, which classifies the population into different Pareto fronts by comparing individuals. After sorting, the population undergoes a normalization step to scale the objective values. Following this, an association operation assigns individuals to predefined reference points. Finally, a niche-preservation method is applied to maintain diversity among the selected individuals by filling underrepresented niches. The nondominated sorting and subsequent operations, particularly the iterative loops, can become computationally expensive as the population size increases. The pseudocode for the original environmental selection process, including nondominated sorting, normalization, association, and niche-preservation, is provided in Algorithm S.1 and Algorithm S.2.

B. Tensorized Environmental Selection

The tensorized version of NSGA-III leverages GPU parallelism to accelerate the entire environmental selection process, which includes nondominated sorting, normalization, association, niche count calculation, and niche selection. By transforming these operations into tensor form, batch processing

Algorithm S.1 Original Environmental Selection in NSGA-III

Input: Combined population P_t with size $2n$, population size n , set of reference points Z .

Output: Next generation population P_{t+1} .

- 1: Perform Nondominated Sorting on P_t to identify the fronts F_1, F_2, \dots ;
- 2: Set $P_{t+1} = \emptyset$;
- 3: $i \leftarrow 1$;
- 4: **while** $|P_{t+1}| + |F_i| \leq n$ **do**
- 5: $P_{t+1} \leftarrow P_{t+1} \cup F_i$;
- 6: $i \leftarrow i + 1$;
- 7: **end while**
- 8: $S_t \leftarrow P_{t+1} \cup F_i$;
- 9: **if** $|P_{t+1}| < n$ **then**
- 10: Normalize the objective values of solutions in S_t ;
- 11: Calculate the distance between S_t and Z ;
- 12: Associate all solutions in S_t to a reference point in Z based on the minimum distance;
- 13: Compute the niche counts for each reference point in P_{t+1} and S_t ;
- 14: **while** $|P_{t+1}| < n$ **do**
- 15: Identify the least crowded reference point $j \in Z$;
- 16: **if** There is no solution associated with j in F_i **then**
- 17: $Z \leftarrow Z \setminus \{j\}$;
- 18: **else**
- 19: **if** There is no solution associated with j in P_{t+1} **then**
- 20: Select the solution $a \in F_i$ with the minimum distance to j ;
- 21: **else**
- 22: Randomly select a solution $a \in F_i$ that is associated with j ;
- 23: **end if**
- 24: $P_{t+1} \leftarrow P_{t+1} \cup \{a\}$;
- 25: Update the niche count for j in both P_{t+1} and S_t ;
- 26: **end if**
- 27: **end while**
- 28: **end if**
- 29: **return** P_{t+1}

can be performed across the population, significantly reducing computational complexity and execution time.

A related matrix-based method [74] has been proposed to accelerate nondominated sorting and population selection using simple and efficient matrix operations. This idea inspires our tensorization method, which extends such matrix-level parallelism to GPU-based tensor computations. In the tensorized process, nondominated sorting is conducted in parallel, efficiently classifying individuals into Pareto fronts. This is followed by a normalization step to scale objective values, an association step to assign individuals to reference points, niche count calculation to track the distribution of individuals across niches, and finally, niche selection to maintain diversity. All these operations are performed in parallel, taking full advantage of the tensorization methodology. The pseudocode for the tensorized environmental selection, including all these

Algorithm S.2 Nondominated Sorting in Original NSGA-III

Input: Population P of size n , with each individual \mathbf{p}_i having objective values $\mathbf{f}(\mathbf{p}_i)$

Output: Set of nondominated fronts F_1, F_2, \dots

- 1: Initialize an empty list of fronts: $F = \emptyset$;
- 2: Initialize an empty list of domination counts: $n_i = 0$ for all $\mathbf{p}_i \in P$;
- 3: Initialize an empty list of dominated sets: $S_i = \emptyset$ for all $\mathbf{p}_i \in P$;
- 4: **for** \mathbf{p}_i in P **do**
- 5: **for** \mathbf{p}_j in P **do**
- 6: **if** $j \neq i$ **then**
- 7: **if** \mathbf{p}_i dominates \mathbf{p}_j **then**
- 8: $S_j \leftarrow S_j \cup \{\mathbf{p}_i\}$;
- 9: **else if** \mathbf{p}_j dominates \mathbf{p}_i **then**
- 10: $n_i \leftarrow n_i + 1$;
- 11: **end if**
- 12: **end if**
- 13: **end for**
- 14: **if** $n_i = 0$ **then**
- 15: $F_i \leftarrow F_i \cup \{\mathbf{p}_i\}$;
- 16: **end if**
- 17: **end for**
- 18: $k \leftarrow 1$;
- 19: **while** $F_k \neq \emptyset$ **do**
- 20: $F_{k+1} = \emptyset$;
- 21: **for** \mathbf{p}_i in F_k **do**
- 22: **for** \mathbf{p}_j in S_i **do**
- 23: $n_j \leftarrow n_j - 1$;
- 24: **if** $n_j = 0$ **then**
- 25: $F_{k+1} \leftarrow F_{k+1} \cup \{\mathbf{p}_j\}$;
- 26: **end if**
- 27: **end for**
- 28: **end for**
- 29: $k \leftarrow k + 1$;
- 30: **end while**

Algorithm S.3 Tensorized Nondominated Sorting

Input: Objective tensor $\mathbf{F} \in \mathbb{R}^{2n \times m}$ and population size n .

Output: Nondomination rank tensor \mathbf{r} and the last rank $l \in \mathbb{N}$.

- 1: $\mathbf{D} \leftarrow [\mathbf{F}_i \prec \mathbf{F}_j]_{i,j}, \quad i, j = 1, 2, \dots, 2n$;
- 2: $\mathbf{c} \leftarrow \sum_{j=1}^{2n} \mathbf{D}_{ij}, \quad i = 1, 2, \dots, 2n$;
- 3: $\mathbf{r} \leftarrow \mathbf{0}_{2n \times 1}$;
- 4: $k \leftarrow 0$;
- 5: $\mathbf{p} \leftarrow \mathbf{1}_{c=0}$;
- 6: **while** any(\mathbf{p}) **do**
- 7: $\mathbf{r} \leftarrow H(\mathbf{p}) \cdot k + H(1 - \mathbf{p}) \odot \mathbf{r}$;
- 8: $\mathbf{d}_j \leftarrow \sum_{i=1}^{2n} (\mathbf{p}_i \cdot \mathbf{D}_{ij}), \quad j = 1, 2, \dots, 2n$;
- 9: $\mathbf{c} \leftarrow \mathbf{c} - \mathbf{d} - \mathbf{p}$;
- 10: $k \leftarrow k + 1$;
- 11: $\mathbf{p} \leftarrow \mathbf{1}_{c=0}$;
- 12: **end while**
- 13: $l \leftarrow \text{sort}(\mathbf{r})[n]$;

steps, is provided in Algorithm S.4.

Algorithm S.4 Environmental Selection in TensorNSGA-III

Input: Shuffled solution tensor \mathbf{X} and corresponding objective tensor \mathbf{F} , reference tensors \mathbf{R} with n_r vectors, and population size n .

Output: Next solution tensor \mathbf{X}_{next} and corresponding objective tensor \mathbf{F}_{next} .

- 1: $\mathbf{r}, l \leftarrow \text{GPU-accelerated Nondominated Sorting}(\mathbf{F}, n)$;
- 2: $\mathbf{F} \leftarrow \mathbf{F} + \text{NaN} \odot H(\mathbf{r} - l)$;
- 3: $\mathbf{F}' \leftarrow \text{Normalize}(\mathbf{F})$; // refer to Algorithm S.5
- 4: // Association:
- 5: $\mathbf{D} \leftarrow \|\mathbf{F}'\| \cdot \sqrt{1 - \left(\frac{\mathbf{F}' \cdot \mathbf{R}^\top}{\|\mathbf{F}'\| \cdot \|\mathbf{R}\|} \right)^2}$;
- 6: $\pi \leftarrow \arg \min_j(\mathbf{D})$;
- 7: $\mathbf{d} \leftarrow \min_j(\mathbf{D})$;
- 8: // Niche count calculation of reference tensor:
- 9: $\rho_j \leftarrow \sum_{i=1}^{2n} H(l - \mathbf{r}_i) \cdot \mathbf{1}_{\pi_i=j}, \quad j = 1, \dots, n_r$;
- 10: $\rho_{l,j} \leftarrow \sum_{i=1}^{2n} \mathbf{1}_{\mathbf{r}_i=l} \cdot \mathbf{1}_{\pi_i=j}, \quad j = 1, \dots, n_r$;
- 11: $n_s \leftarrow \sum \rho$;
- 12: // Niche selection:
- 13: $\rho \leftarrow \infty \cdot \mathbf{1}_{\rho_l=0}$;
- 14: $\pi \leftarrow \pi + \infty \cdot \mathbf{1}_{\mathbf{r} \neq l}$;
- 15: $i \leftarrow [1, 2, \dots, n_r]$;
- 16: $\rho_s \leftarrow i \odot \mathbf{1}_{\rho_l=0} + \infty \cdot \mathbf{1}_{\rho \neq 0}$;
- 17: $M_{ij} \leftarrow \mathbf{1}_{\rho_{s,i}=\pi_j}$;
- 18: $\mathbf{D}' \leftarrow \mathbf{d} \odot \mathbf{M} + \infty \cdot (1 - \mathbf{M})$;
- 19: $\mathbf{q} \leftarrow \arg \min_j(\mathbf{D}')$;
- 20: $\mathbf{q} \leftarrow \infty \cdot \mathbf{1}_{\rho_{\text{selected}}=\infty} + \mathbf{q} \odot (1 - \mathbf{1}_{\rho_{\text{selected}}=\infty})$;
- 21: $\mathbf{q} \leftarrow \text{Update}(\mathbf{q})$;
- 22: $\mathbf{r}[\mathbf{q}] \leftarrow l - 1$;
- 23: $n_s \leftarrow n_s + \sum \mathbf{1}_{\rho=0}$;
- 24: $n_{\text{dif}} \leftarrow n - n_s$;
- 25: $\mathbf{r} \leftarrow \text{Update-Rank}(\mathbf{r}, \mathbf{q}, n_{\text{dif}}, l)$; // refer to Algorithm S.6
- 26: $i_{\text{next}} \leftarrow \text{sort}(H(l - \mathbf{r}) \odot \mathbf{a} + \infty \cdot (1 - H(l - \mathbf{r})))[:n], \mathbf{a} = [0, 1, \dots, n - 1]$;
- 27: $\mathbf{X}_{\text{next}}, \mathbf{F}_{\text{next}} \leftarrow \mathbf{X}[i_{\text{next}}], \mathbf{F}[i_{\text{next}}]$;

S.III. TENSORIZATION OF MOEA/D

This section contrasts the original MOEA/D algorithm with its tensorized version, highlighting how tensorization methodology enhances computational efficiency. The provided pseudocode offers insights into the original and tensorized implementations of environmental selection.

A. Original MOEA/D

MOEA/D decomposes a MOP into several single-objective subproblems. Each subproblem is optimized individually, and the solutions are combined to approximate the Pareto front. The original MOEA/D uses weight vectors to guide the search process, and solutions are iteratively updated based on their performance relative to these vectors.

The decomposition process in the original MOEA/D involves calculating the aggregation function for each subproblem and updating the solutions sequentially. This step

Algorithm S.5 Normalization in TensorNSGA-III

Input: Objective Tensor \mathbf{F} ;
Output: Normalized objective tensor \mathbf{F}' ;

- 1: $\mathbf{z} \leftarrow \min_i(\mathbf{F})$;
- 2: $\mathbf{F} \leftarrow \mathbf{F} - \mathbf{z}$;
- 3: $\mathbf{W} \leftarrow \mathbf{I}_m$;
- 4: **Function** $f(\mathbf{w})$
- 5: $e \leftarrow \text{argmin}(\max_j(\mathbf{F}/\mathbf{w}))$;
- 6: **return** e
- 7: $\mathbf{e} \leftarrow \text{vmap}(f)(\mathbf{W})$;
- 8: $\mathbf{E} \leftarrow \mathbf{F}[e]$;
- 9: **if** $\text{rank}(\mathbf{E}) = m$ **then**
- 10: $\mathbf{a} \leftarrow \text{compute intercepts based on } \mathbf{E}$;
- 11: **else**
- 12: $\mathbf{a} \leftarrow \max_i(\mathbf{F})$;
- 13: **end if**
- 14: $\mathbf{F}' \leftarrow \mathbf{F}/\mathbf{a}$;

Algorithm S.6 Update-Rank in TensorNSGA-III

Input: Rank tensor \mathbf{r} , selected indices s_{sel} , difference n_{dif} , last rank l .
Output: Updated rank tensor \mathbf{r} .

- 1: $\mathbf{s} \leftarrow \{i \mid r_i = l\}$;
- 2: $\mathbf{s} \leftarrow \text{sort}(\mathbf{s})$;
- 3: $\mathbf{s} \leftarrow \mathbf{s} \odot H(n_{\text{dif}} - i) + s_{\text{sel}}[0] \cdot H(i - n_{\text{dif}}), i = [1, 2, \dots, \text{len}(\mathbf{s})]$;
- 4: $\boldsymbol{\mu} \leftarrow \infty \cdot \mathbb{1}_{s_{\text{sel}}=s_{\text{sel}}[0]} + s_{\text{sel}} \odot (1 - \mathbb{1}_{s_{\text{sel}}=s_{\text{sel}}[0]})$;
- 5: $i_{\text{drop}} \leftarrow \text{sort}(\boldsymbol{\mu})[0]$;
- 6: $\boldsymbol{\mu}_{\text{adj}} \leftarrow \boldsymbol{\mu} \odot H(-n_{\text{dif}} - i) + i_{\text{drop}} \cdot H(i + n_{\text{dif}}), i = [1, 2, \dots, \text{len}(\boldsymbol{\mu})]$;
- 7: $\mathbf{r}_{\text{add}}, \mathbf{r}_{\text{cut}} \leftarrow \mathbf{r}$;
- 8: $\mathbf{r}_{\text{add}}[s] \leftarrow l - 1$;
- 9: $\mathbf{r}_{\text{cut}}[\boldsymbol{\mu}_{\text{adj}}] \leftarrow l$;
- 10: $\mathbf{r} \leftarrow H(n_{\text{dif}}) \odot \mathbf{r}_{\text{add}} + H(-n_{\text{dif}}) \odot \mathbf{r}_{\text{cut}}$;

is computationally intensive, especially as the number of subproblems and the population size increase. The pseudocode for the original MOEA/D is provided in Algorithm S.7.

B. Tensorized MOEA/D

The implementation of TensorMOEA/D retains the core principles of the original algorithm but leverages the parallel processing capabilities of GPUs. The pseudocode for the environmental selection of TensorMOEA/D is shown in Algorithm S.8.

S.IV. TENSORIZATION OF HYPE

In this section, we discuss the original Monte Carlo HV estimation method employed in HypE and present its tensorized adaptation, which enhances computational efficiency by utilizing GPU parallelization. The tensorized version, TensorHypE, is specifically designed to handle larger populations and more complex optimization tasks by performing HV contribution calculations in parallel, significantly speeding up the environmental selection process.

Algorithm S.7 Original MOEA/D Algorithm

Input: The maximal number of generations t_{max} ; n weight vectors $\lambda_1, \dots, \lambda_n$; the number of the weight vectors in the neighborhood of each weight vector; maximum number of iterations t_{max} ;
Output: EP (efficient set of solutions);

- 1: $\text{EP} \leftarrow \emptyset$;
- 2: **for** each weight vector λ_i **do**
- 3: Compute Euclidean distances to all other weight vectors;
- 4: Determine the T closest weight vectors $B(i) = \{i_1, \dots, i_T\}$;
- 5: **end for**
- 6: Generate initial population $\mathbf{x}_1, \dots, \mathbf{x}_n$:
- 7: **for** $i = 1$ to n **do**
- 8: $\mathbf{f}_i \leftarrow \mathbf{f}(\mathbf{x}_i)$;
- 9: **end for**
- 10: Initialize $\mathbf{z} = (z_1, \dots, z_m)^{\top}$;
- 11: **for** $t = 1$ to t_{max} **do**
- 12: **for** $i = 1$ to n **do**
- 13: Randomly select two indices p, q from $B(i)$;
- 14: Generate a new solution \mathbf{y} from \mathbf{x}_p and \mathbf{x}_q using genetic operators;
- 15: Apply problem-specific improvement heuristic on \mathbf{y} to produce \mathbf{y}' ;
- 16: **for** $j = 1$ to m **do**
- 17: **if** $f_j(\mathbf{y}') < z_j$ **then**
- 18: $z_j \leftarrow f_j(\mathbf{y}')$;
- 19: **end if**
- 20: **end for**
- 21: **for** each $j \in B(i)$ **do**
- 22: **if** $g(\mathbf{y}'|\lambda^j, \mathbf{z}) \leq g(\mathbf{x}_j|\lambda^j, \mathbf{z})$ **then**
- 23: $\mathbf{x}_j \leftarrow \mathbf{y}'$;
- 24: $\mathbf{f}_j \leftarrow \mathbf{f}(\mathbf{y}')$;
- 25: **end if**
- 26: **end for**
- 27: Remove from EP all vectors dominated by $\mathbf{f}(\mathbf{y}')$;
- 28: **if** no vectors in EP dominate $\mathbf{f}(\mathbf{y}')$ **then**
- 29: Add $\mathbf{f}(\mathbf{y}')$ to EP;
- 30: **end if**
- 31: **end for**
- 32: **end for**

A. Original Environmental Selection

The original HypE uses Monte Carlo sampling to estimate the HV contributions of solutions in a population. The algorithm ranks solutions based on their contributions to the overall HV, favoring those that contribute more to the Pareto front's volume. The original implementation of HypE involves sequentially sampling points within a defined HV region and updating fitness values iteratively.

Algorithm S.9 illustrates the pseudocode for the original Monte Carlo HV estimation method in HypE. This method can be computationally expensive when applied to large populations and high-dimensional objective spaces, as it relies heavily on sequential processing.

Algorithm S.8 Environmental Selection of TensorMOEA/D

Input: Solution tensor \mathbf{X} , Objective tensor \mathbf{F}_1 , Offspring tensor \mathbf{O} , the objective of offspring \mathbf{F}_2 , the population size n , the ideal points \mathbf{z} , the weights \mathbf{W} , the neighbors indices \mathbf{I}_{nb} , and the PBI function f_{PBI} ;

Output: Next solution tensor \mathbf{X}_{next} and next objective tensor \mathbf{F}_{next} ;

- 1: $\mathbf{z}_{\min} \leftarrow \min_i(\mathbf{z} \cup \mathbf{F}_2)$;
- 2: $\mathbf{I}_{\text{sub}} \leftarrow [i \mid i \in \mathbb{N}, 0 \leq i < n]$;
- 3: $\mathbf{M} \leftarrow \mathbf{0}_n$;
- 4: **Function** $f_{\text{op1}}(\mathbf{i}_{\text{nb}}, \mathbf{f}_2)$
- 5: $\mathbf{g}_{\text{old}} \leftarrow f_{\text{PBI}}(\mathbf{F}_1[\mathbf{i}_{\text{nb}}], \mathbf{w}[\mathbf{i}_{\text{nb}}], \mathbf{z}_{\min})$;
- 6: $\mathbf{g}_{\text{new}} \leftarrow f_{\text{PBI}}(\mathbf{f}_2, \mathbf{w}[\mathbf{i}_{\text{nb}}], \mathbf{z}_{\min})$;
- 7: $\mathbf{M}[\mathbf{i}_{\text{nb}}] \leftarrow H(\mathbf{g}_{\text{old}} - \mathbf{g}_{\text{new}})$;
- 8: $\mathbf{I}_{\text{sub}} \leftarrow \mathbf{M} \odot (-1) + (1 - \mathbf{M}) \odot \mathbf{I}_{\text{sub}}$;
- 9: **return** \mathbf{I}_{sub}
- 10: // Comparison and Population Update:
- 11: $\mathbf{I}_{\text{new}} \leftarrow \text{vmap}(f_{\text{op1}})(\mathbf{I}_{\text{nb}}, \mathbf{F}_2)$;
- 12: **Function** $f_{\text{op2}}(\mathbf{i}_{\text{new}}, \mathbf{x}, \mathbf{f}_1, \mathbf{w})$
- 13: $\mathbf{f} \leftarrow \mathbb{1}_{\mathbf{i}_{\text{new}} = -1} \odot \mathbf{F}_2 + (1 - \mathbb{1}_{\mathbf{i}_{\text{new}} = -1}) \odot \mathbf{f}_1$;
- 14: $\mathbf{x} \leftarrow \mathbb{1}_{\mathbf{i}_{\text{new}} = -1} \odot \mathbf{O} + (1 - \mathbb{1}_{\mathbf{i}_{\text{new}} = -1}) \odot \mathbf{x}$;
- 15: $i \leftarrow \text{argmin}(f_{\text{PBI}}(\mathbf{f}, \mathbf{w}, \mathbf{z}_{\min}))$;
- 16: **return** $\mathbf{x}[i], \mathbf{f}[i]$
- 17: // Elite Selection:
- 18: $\mathbf{X}_{\text{next}}, \mathbf{F}_{\text{next}} \leftarrow \text{vmap}(f_{\text{op2}})(\mathbf{I}_{\text{new}}^\top, \mathbf{X}, \mathbf{F}_1, \mathbf{w})$;
- 19: // Update the ideal point:
- 20: $\mathbf{z} \leftarrow \mathbf{z}_{\min}$;

B. Tensorized Environmental Selection

While preserving the core principles of HypE, crucial steps such as nondominated sorting, HV estimation, and selection are accelerated using GPUs. Algorithm S.10 illustrates the tensorized environmental selection process, and Algorithm S.11 provides details on the GPU-accelerated HV estimation via Monte Carlo sampling.

S.V. INTRODUCTION TO TENSORRVEA

TensorRVEA [57] is a tensorized extension of the original RVEA [32] designed to leverage the computational advantages of modern hardware, such as GPUs. By representing key data structures and operations in tensor form, TensorRVEA efficiently processes large populations and high-dimensional objectives, making it particularly suitable for solving complex MOPs.

The key contribution in TensorRVEA lies in its ability to parallelize the selection process using tensor operations. The selection operation, a crucial component of the algorithm, is responsible for maintaining a diverse set of solutions that are well-distributed along the Pareto front. This operation uses angular distances between objective tensors and reference tensors to guide the search towards unexplored regions of the objective space.

The TensorRVEA environmental selection operation illustrated in Algorithm S.12 demonstrates the efficiency of tensorization. By performing operations on entire populations

Algorithm S.9 Original Monte Carlo HV Estimation

Input: Population \mathbf{P} , reference set \mathbf{R} , fitness parameter $k \in \mathbb{N}$, number of sampling points $M \in \mathbb{N}$

Output: Estimated fitness values \mathbf{F}

- 1: // determine sampling box \mathbf{S}
- 2: **for** $i \leftarrow 1$ to m **do**
- 3: $l_i \leftarrow \min_{\mathbf{a} \in \mathbf{P}} f_i(\mathbf{a})$;
- 4: $u_i \leftarrow \max_{(r_1, \dots, r_m) \in \mathbf{R}} r_i$;
- 5: **end for**
- 6: $\mathbf{S} \leftarrow [l_1, u_1] \times \dots \times [l_m, u_m]$;
- 7: $\mathbf{V} \leftarrow \prod_{i=1}^m \max\{0, (u_i - l_i)\}$;
- 8: // reset fitness assignment
- 9: $\mathbf{F} \leftarrow \bigcup_{\mathbf{a} \in \mathbf{P}} \{(\mathbf{a}, 0)\}$;
- 10: // sampling
- 11: **for** $j \leftarrow 1$ to M **do**
- 12: choose $\mathbf{s} \in \mathbf{S}$ uniformly at random;
- 13: **if** $\exists \mathbf{r} \in \mathbf{R} : \mathbf{s} \leq \mathbf{r}$ **then**
- 14: $\mathbf{U}_{\mathbf{P}} \leftarrow \bigcup_{\mathbf{a} \in \mathbf{P}, \mathbf{f}(\mathbf{a}) \leq \mathbf{s}} \{\mathbf{f}(\mathbf{a})\}$;
- 15: **if** $|\mathbf{U}_{\mathbf{P}}| \leq k$ **then**
- 16: // hit in a relevant partition
- 17: $\alpha \leftarrow \prod_{l=1}^{|\mathbf{U}_{\mathbf{P}}|-1} \frac{k-l}{|\mathbf{P}|-l}$;
- 18: // update HV estimates
- 19: $\mathbf{F}' \leftarrow \emptyset$;
- 20: **for all** $(\mathbf{a}, \mathbf{v}) \in \mathbf{F}$ **do**
- 21: **if** $\mathbf{f}(\mathbf{a}) \leq \mathbf{s}$ **then**
- 22: $\mathbf{F}' \leftarrow \mathbf{F}' \cup \{(\mathbf{a}, \mathbf{v} + \frac{\alpha}{|\mathbf{U}_{\mathbf{P}}|} \cdot \frac{\mathbf{V}}{M})\}$;
- 23: **else**
- 24: $\mathbf{F}' \leftarrow \mathbf{F}' \cup \{(\mathbf{a}, \mathbf{v})\}$;
- 25: **end if**
- 26: **end for**
- 27: $\mathbf{F} \leftarrow \mathbf{F}'$;
- 28: **end if**
- 29: **end if**
- 30: **end for**

Algorithm S.10 Environmental Selection of TensorHypE

Input: Merged solution tensor \mathbf{X} , merged objective tensor \mathbf{F} , reference point \mathbf{v}_{ref} , the population size n , and the number of sample points s

Output: Next solution tensor \mathbf{X}_{next} and next objective tensor \mathbf{F}_{next} ;

- 1: $\mathbf{r}, l \leftarrow \text{GPU-based Nondominated Sorting}(\mathbf{F})$;
- 2: $\mathbf{r}_{\text{mask}} \leftarrow H(l - \mathbf{r})$;
- 3: $k \leftarrow \sum_{i=1}^{\text{len}(\mathbf{r}_{\text{mask}})} \mathbf{r}_{\text{mask},i} - n$;
- 4: $\mathbf{v}_{\text{hv}} \leftarrow \text{Hypervolume Estimation}(\mathbf{F}, \mathbf{v}_{\text{ref}}, k, s)$;
- 5: $\mathbf{d} \leftarrow \mathbf{r}_{\text{mask}} \odot \mathbf{v}_{\text{hv}} + (1 - \mathbf{r}_{\text{mask}}) \odot (-\infty)$;
- 6: $\mathbf{I} \leftarrow \text{lexsort}(\mathbf{r}, -\mathbf{d})[:n]$;
- 7: $\mathbf{X}_{\text{next}} \leftarrow \mathbf{X}[\mathbf{I}]$;
- 8: $\mathbf{F}_{\text{next}} \leftarrow \mathbf{F}[\mathbf{I}]$;

Algorithm S.11 HV Estimation via Monte Carlo Sampling

Input: Objective tensor \mathbf{F} , reference point \mathbf{v}_{ref} , the parameter k and the number of sample points s ;

Output: HV values \mathbf{v}_{hv} ;

- 1: $n_1, m \leftarrow \text{shape}(\mathbf{F})$;
- 2: $\mathbf{l} \leftarrow [i \mid i \in \mathbb{N}, 1 \leq i < n_1]$;
- 3: $\boldsymbol{\lambda} \leftarrow [1, (k - l)/(n_1 - l)]$;
- 4: $\boldsymbol{\alpha}_j \leftarrow \prod_{i=1}^j \lambda_i / j, j = 1, 2, \dots, k$;
- 5: $f_l \leftarrow \min_i(\mathbf{F})$;
- 6: $f_u \leftarrow \mathbf{v}_{\text{ref}}$;
- 7: $\mathbf{S} \leftarrow \text{Uniform sampling from } [f_l, f_u] \text{ in dimension } s \times m$;
- 8: $\mathbf{v}_{\text{ds}} \leftarrow \mathbf{0}_{1 \times s}$;
- 9: **Function** $f_{\text{pds}}(\mathbf{f})$
- 10: $t_{\text{pds}} \leftarrow \mathbb{1}_{\sum_{j=1}^m H(\mathbf{S}_{ij} - \mathbf{f}) = m}$, where $i = 1, \dots, s$;
- 11: **return** t_{pds} ;
- 12: **Function** $f_{\text{hv}}(\mathbf{t}_{\text{pds}})$
- 13: $\mathbf{v}_{\text{temp}} \leftarrow t_{\text{pds}} \odot \mathbf{v}_{\text{ds}} - (1 - t_{\text{pds}})$;
- 14: $\mathbf{v}_{\text{hv},i} \leftarrow \sum_{i=1}^s (\boldsymbol{\alpha}[\mathbf{v}_{\text{temp}}] \odot \mathbb{1}_{\mathbf{v}_{\text{temp}} \neq -1})_i$;
- 15: **return** $\mathbf{v}_{\text{hv},i}$;
- 16: // Compute point dominance scores:
- 17: $\mathbf{T}_{\text{pds}} \leftarrow \text{vmap}(f_{\text{pds}})(\mathbf{F})$;
- 18: $\mathbf{T}_{\text{temp}} \leftarrow \mathbf{T}_{\text{pds}} \odot (\mathbb{1}_{n \times 1} \cdot \mathbf{v}_{\text{ds}} + 1) + (1 - \mathbf{T}_{\text{pds}}) \odot (\mathbb{1}_{n \times 1} \cdot \mathbf{v}_{\text{ds}})$;
- 19: $\mathbf{v}_{\text{ds}} \leftarrow \text{maximum}(\sum_{i=1}^{n_1} \mathbf{T}_{\text{temp},i} - 1, 0)$;
- 20: // Compute HV values:
- 21: $\mathbf{v}_{\text{hv}} \leftarrow \text{vmap}(f_{\text{hv}})(\mathbf{T}_{\text{pds}})$;
- 22: $\mathbf{v}_{\text{hv}} \leftarrow \mathbf{v}_{\text{hv}} \cdot \prod_{i=1}^m (\mathbf{v}_{\text{ref},i} - f_{l,i}) / s$;

Algorithm S.12 Environmental Selection of TensorRVEA

Input: Solution tensor \mathbf{X} with n individual, objective tensor \mathbf{F} , reference tensors \mathbf{V} with r vectors, maximum number of generations t_{\max} , current generation t , and rate of change of penalty α ;

Output: Elite solution tensor $\mathbf{X}_{\text{elite}}$;

- 1: $\mathbf{z}_{\min} \leftarrow \min_i(\mathbf{F})$;
- 2: $\mathbf{F}' \leftarrow \mathbf{F} - \mathbf{z}_{\min}$;
- 3: $\boldsymbol{\Theta} \leftarrow \arccos(\mathbf{F}' \cdot \mathbf{V}^T / (\|\mathbf{F}'\| \cdot \|\mathbf{V}^T\|))$;
- 4: $\mathbf{A} \leftarrow \text{repeat}(\text{argmin}_j(\boldsymbol{\Theta}), r)$;
- 5: $\mathbf{T}_{\text{part}} \leftarrow \text{repeat}([0, 1, \dots, n - 1]^T, r)$;
- 6: $\mathbf{I} \leftarrow \text{repeat}([0, 1, \dots, r - 1], n)$;
- 7: $\mathbf{T}_{\text{part}} \leftarrow (1 - |\text{sgn}(\mathbf{A} - \mathbf{I})|) \odot \mathbf{T}_{\text{part}} - |\text{sgn}(\mathbf{A} - \mathbf{I})|$;
- 8: $\boldsymbol{\Gamma} \leftarrow \text{argmin}_j \left(\arccos(\mathbf{V} \cdot \mathbf{V}^T / (\|\mathbf{V}\| \cdot \|\mathbf{V}^T\|)) \right)$;
- 9: **Function** $f_{\text{APD}}(\mathbf{t}_{\text{part}}, \boldsymbol{\gamma}, \boldsymbol{\theta})$
- 10: $\mathbf{t}_{\text{APD}} = \left(1 + m \cdot \left(\frac{t}{t_{\max}} \right)^{\alpha} \cdot \frac{\boldsymbol{\theta}[\mathbf{t}_{\text{part}}]}{\boldsymbol{\gamma}} \right) \odot \|\mathbf{F}'[\mathbf{t}_{\text{part}}]\|$;
- 11: **return** \mathbf{t}_{APD} ;
- 12: $\mathbf{T}_{\text{APD}} \leftarrow \text{vmap}(f_{\text{APD}})(\mathbf{T}_{\text{part}}, \boldsymbol{\Gamma}, \boldsymbol{\Theta})$;
- 13: Replace elements in \mathbf{T}_{APD} with \inf where $\mathbf{T}_{\text{part}} = -1$;
- 14: $\mathbf{I}_{\text{next}} \leftarrow \text{argmin}_i(\mathbf{T}_{\text{APD}})$;
- 15: $\mathbf{X}_{\text{elite}} \leftarrow \mathbf{X}[\mathbf{I}_{\text{next}}]$;

simultaneously, TensorRVEA significantly accelerates the selection process compared to its traditional counterpart. This enhancement makes it particularly effective for large-scale multiobjective optimization tasks, where maintaining a diverse set of solutions along the Pareto front is critical.

S.VI. MULTIOBJECTIVE ROBOT CONTROL BENCHMARK

In this section, we introduce the multiobjective robot control problems (MoRobtrol) designed for this study. These problems are specifically crafted to evaluate the performance of the proposed tensorized EMO algorithms in realistic and challenging scenarios. The MoRobtrol consists of 9 distinct multiobjective problems: MoHalfcheetah, MoHopper, MoSwimmer, MoInvertedDoublePendulum (MoIDP), MoWalker2d, MoPusher, MoReacher, MoHumanoid, and MoHumanoidstandup (MoHumanoid-s). Each problem is a variant of a classic single-objective control task, reformulated to include multiple objectives that often conflict with one another, increasing both complexity and relevance to real-world applications. The specific mathematical definitions of these problems are detailed below, offering a precise description of the objectives and constraints associated with each task.

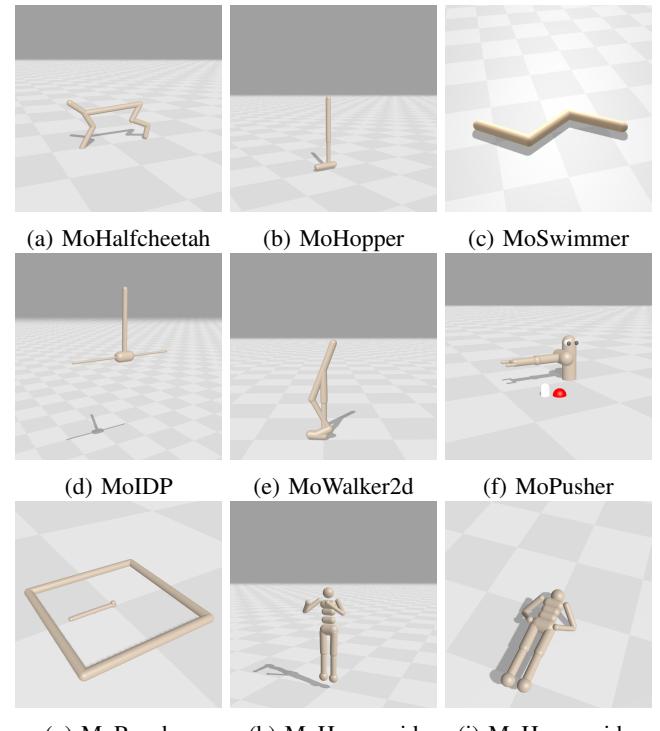


Fig. S.1: Visual illustration of the robot control tasks in the proposed MoRobtrol benchmark.

MoHalfcheetah: The observation space is represented as \mathbb{R}^{17} , indicating a 17-dimensional vector for each observation. Similarly, the action space is defined as \mathbb{R}^6 , indicating a 6-dimensional vector for each action. Each episode is constrained to a maximum of 1000 time steps. The first objective, termed the forward reward, is defined as:

$$f_v = w_1 \cdot v_x, \quad (\text{S.12})$$

where v_x denotes the velocity in the x -direction, and w_1 is the weight associated with this velocity component. The second objective, known as the control cost, is expressed as:

$$f_c = -w_2 \cdot \sum_i a_i^2, \quad (\text{S.13})$$

where a_i represents the action taken by the i -th actuator, and w_2 is the weight applied to the control cost.

MoHopper: The observation space is represented as \mathbb{R}^{11} , indicating that each observation is an 11-dimensional vector. The action space is defined as \mathbb{R}^3 , meaning that each action is a 3-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, termed the forward reward, is given by:

$$f_v = w_1 \cdot v_x + C, \quad (\text{S.14})$$

where v_x denotes the velocity in the x -direction, w_1 is the weight associated with this velocity component, and $C = 1$ is a constant survival reward indicating that the agent remains active. The second objective, referred to as height, is expressed as:

$$f_h = 10 \cdot (h_{\text{curr}} - h_{\text{init}}) + C, \quad (\text{S.15})$$

where h_{curr} represents the current height of the hopper, and h_{init} denotes the initial height. The third objective, known as the control cost, is defined as:

$$f_c = -\sum_i a_i^2 + C, \quad (\text{S.16})$$

where a_i represents the action taken by the i -th actuator.

MoSwimmer: The observation space is represented as \mathbb{R}^8 , indicating that each observation is an 8-dimensional vector. The action space is defined as \mathbb{R}^2 , meaning that each action is a 2-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the forward reward, is defined as:

$$f_v = w_1 \cdot v_x, \quad (\text{S.17})$$

where v_x denotes the velocity in the x -direction, and w_1 is the weight associated with this velocity component. The second objective, known as the control cost, is expressed as:

$$f_c = -w_2 \cdot \sum_i a_i^2, \quad (\text{S.18})$$

where a_i represents the action taken by the i -th actuator, and w_2 is the weight associated with the control cost.

MoInvertedDoublePendulum (MoIDP): The observation space is represented as \mathbb{R}^{11} , indicating that each observation is an 11-dimensional vector. The action space is defined as \mathbb{R}^1 , meaning that each action is a scalar. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the distance penalty, is defined as:

$$f_{dp} = -w_1 \cdot (x - x_0)^2 - w_2 \cdot (y - y_0)^2 + C, \quad (\text{S.19})$$

where x and y are the current positions of the model, $x_0 = 0$ and $y_0 = 2$ are the initial positions, w_1 and w_2 are the weights for the distance penalty in the x and y directions, respectively, and $C = 10$ is a constant survival reward indicating that the

agent remains operational. The second objective, known as the speed penalty, is expressed as:

$$f_{sp} = -w_3 \cdot v_x^2 - w_4 \cdot v_y^2 + C, \quad (\text{S.20})$$

where v_x and v_y are the velocities in the x and y directions, respectively, and w_3 and w_4 are the weights associated with these velocity components.

MoWalker2d: The observation space is represented as \mathbb{R}^{17} , indicating that each observation is a 17-dimensional vector. The action space is defined as \mathbb{R}^6 , meaning that each action is a 6-dimensional vector. Each episode is limited to a maximum of 1000 steps. The first objective, referred to as the forward reward, is defined as:

$$f_v = w_1 \cdot v_x + C, \quad (\text{S.21})$$

where v_x denotes the velocity in the x -direction, w_1 is the weight associated with this velocity component, and $C = 1$ represents a constant survival reward, indicating that the agent remains active. The second objective, known as the control cost, is expressed as:

$$f_c = -w_2 \cdot \sum_i a_i^2 + C, \quad (\text{S.22})$$

where a_i represents the action taken by the i -th actuator, and w_2 is the weight associated with the control cost.

MoPusher: The observation space is represented as \mathbb{R}^{23} , indicating that each observation is a 23-dimensional vector. The action space is defined as \mathbb{R}^7 , meaning that each action is a 7-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the near reward, is defined as:

$$f_n = -\sqrt{(x_{\text{fin}} - x_{\text{obj}})^2 + (y_{\text{fin}} - y_{\text{obj}})^2}, \quad (\text{S.23})$$

where x_{fin} and y_{fin} denote the coordinates of the fingertip, while x_{obj} and y_{obj} denote the coordinates of the object. The second objective, known as the distance reward, is expressed as:

$$f_d = -\sqrt{(x_{\text{obj}} - x_{\text{tar}})^2 + (y_{\text{obj}} - y_{\text{tar}})^2}, \quad (\text{S.24})$$

where x_{obj} and y_{obj} are the coordinates of the object, and x_{tar} and y_{tar} are the coordinates of the target. The third objective, termed the control cost, is defined as:

$$f_c = -\sum_i a_i^2, \quad (\text{S.25})$$

where a_i represents the action taken by the i -th actuator.

MoReacher: The observation space is represented as \mathbb{R}^{11} , indicating that each observation is an 11-dimensional vector. The action space is defined as \mathbb{R}^2 , meaning that each action is a 2-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the distance reward, is defined as:

$$f_d = -\sqrt{(x_{\text{fin}} - x_{\text{tar}})^2 + (y_{\text{fin}} - y_{\text{tar}})^2}, \quad (\text{S.26})$$

where x_{fin} and y_{fin} denote the coordinates of the fingertip, and x_{tar} and y_{tar} denote the coordinates of the target. The second objective, known as the control cost, is expressed as:

$$f_c = -\sum_i a_i^2, \quad (\text{S.27})$$

where a_i represents the action taken by the i -th actuator.

MoHumanoid: The observation space is represented as \mathbb{R}^{244} , indicating that each observation is a 244-dimensional vector. The action space is defined as \mathbb{R}^{17} , meaning that each action is a 17-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the forward reward, is defined as:

$$f_v = w_1 \cdot v_x + C, \quad (\text{S.28})$$

where v_x denotes the velocity in the x -direction, w_1 is the weight associated with this velocity component, and $C = 5$ represents a constant survival reward, indicating that the agent remains operational. The second objective, known as the control cost, is given by:

$$f_c = -w_2 \cdot \sum_i a_i^2 + C, \quad (\text{S.29})$$

where a_i represents the action taken by the i -th actuator, and w_2 is the weight applied to the control cost.

MoHumanoidstandup (MoHumanoid-s): The observation space is represented as \mathbb{R}^{244} , indicating that each observation is a 244-dimensional vector. The action space is defined as \mathbb{R}^{17} , meaning that each action is a 17-dimensional vector. Each episode is constrained to a maximum of 1000 time steps. The first objective, referred to as the forward reward, is defined as:

$$f_v = w_1 \cdot v_y + C, \quad (\text{S.30})$$

where v_y denotes the velocity in the y -direction, w_1 is the weight associated with this velocity component, and $C = 5$ represents a constant survival reward, indicating that the agent remains operational. The second objective, known as the control cost, is given by:

$$f_c = -w_2 \cdot \sum_i a_i^2 + C, \quad (\text{S.31})$$

where a_i represents the action taken by the i -th actuator, and w_2 is the weight applied to the control cost.

S.VII. EXPERIMENTS

This section presents the performance indicators and reference points employed in the MoRobtrol benchmark, along with supplementary experiments covering comparisons with CUDA-based acceleration algorithms, analysis of GPU performance impact, and detailed evaluations on the DTLZ test suite.

A. Performance Indicators

For a comprehensive analysis of the proposed tensorized EMO algorithms, we employ three key performance indicators: inverted generational distance (IGD), hypervolume (HV), and expected utility (EU) metric.

1) **IGD**: The IGD measures how closely the set of final solutions \mathbf{F} approximates the reference PF \mathbf{F}^* . It is defined as:

$$\text{IGD}(\mathbf{F}, \mathbf{F}^*) = \frac{\sum_{\mathbf{f}^* \in \mathbf{F}^*} \min_{\mathbf{f} \in \mathbf{F}} \|\mathbf{f} - \mathbf{f}^*\|}{|\mathbf{F}^*|}, \quad (\text{S.32})$$

where $\|\cdot\|$ denotes the Euclidean distance. A lower IGD value indicates better approximation to the true Pareto Front.

2) **HV**: The HV metric evaluates the volume of the objective space dominated by the obtained solutions and bounded by a reference point \mathbf{v}_{ref} . The HV is calculated as:

$$\text{HV}(\mathbf{F}, \mathbf{v}_{\text{ref}}) = \bigcup_{\mathbf{f} \in \mathbf{F}} \text{volume}(\mathbf{v}_{\text{ref}}, \mathbf{f}), \quad (\text{S.33})$$

where $\text{volume}(\mathbf{v}_{\text{ref}}, \mathbf{f})$ represents the volume of the hypercube defined by the vector \mathbf{f} and the reference point \mathbf{v}_{ref} . The reference point \mathbf{v}_{ref} is predetermined as a vector of ones, and \mathbf{f} represents the set of normalized objective values. A higher HV value indicates better spread and convergence of the solutions.

3) **EU**: The EU for a set of objectives $\mathbf{F} \in \mathbb{R}^{m \times n}$ is calculated as:

$$\text{EU}(\mathbf{F}, \mathbf{W}) = \frac{1}{n} \sum_{j=1}^n \max_{i \in \{1, \dots, m\}} (\mathbf{W}^\top \cdot \mathbf{f}_i(\mathbf{F}_j)), \quad (\text{S.34})$$

where $\mathbf{W} \in \mathbb{R}^m$ is the weight vector, \mathbf{F}_j is the j -th solution in the objective space, and $\mathbf{f}_i(\mathbf{F}_j)$ represents the utility function applied to the i -th objective of the j -th solution. The expression $\mathbf{W}^\top \cdot \mathbf{f}_i(\mathbf{F}_j)$ calculates the weighted utility of the objectives, and the max operation selects the maximum utility across all objectives for each solution. The mean value across all solutions provides the expected utility.

B. Comparison with CUDA-based Acceleration Algorithms

In this experiment, we compare the TensorNSGA-II algorithm with the NSGA-II implementation in EvoTorch [69]. EvoTorch is a PyTorch-based evolutionary algorithm library that enables CUDA acceleration. Since EvoTorch only provides the NSGA-II algorithm, we conduct a comparison between TensorNSGA-II and the EvoTorch NSGA-II on the DTLZ test suite.

1) **Experimental Settings**: The population size is set to 10000, with a dimensionality of 5000 and 3 objectives. Both algorithms are independently run 31 times, with 100 iterations per run. The comparison is based on the IGD and computation time. We employ the Wilcoxon rank-sum test to assess whether there are significant differences between the two algorithms, with the performance metrics of the superior algorithm highlighted.

2) **Comparison Results**: As shown in Table S.I, TensorNSGA-II outperforms the NSGA-II implementation in EvoTorch across all DTLZ problems, both in terms of IGD and computation time. Furthermore, on the RTX 4090 GPU, the speedup ranged from a maximum of $5.61\times$ to a minimum of $1.59\times$, demonstrating the advantages of the tensorized algorithm.

C. Impact of Different GPUs on Performance

In this experiment, we compare the performance of three proposed algorithms: TensorNSGA-III, TensorMOEA/D, and TensorHypE. The comparison is conducted across multiple hardware configurations, including a CPU and the GPUs RTX 2080Ti, RTX 3090, and RTX 4090. The evaluation is conducted using the DTLZ1 problem. The goal is to analyze

TABLE S.I

STATISTICAL RESULTS (MEAN AND STANDARD DEVIATION) OF THE IGD AND RUNTIME (S) OBTAINED BY NSGA-II AND TENSORNSGA-II IN DTLZ1–DTLZ7. THE BEST RESULTS ARE HIGHLIGHTED.

Indicator	Problem	NSGA-II	TensorNSGA-II
IGD	DTLZ1	1.4861e+05 (4.9045e+02)	1.3866e+05 (5.4954e+02)
	DTLZ2	3.7692e+02 (2.7958e+00)	2.3238e+02 (8.9165e-01)
	DTLZ3	5.0293e+05 (6.4328e+02)	4.6302e+05 (1.2591e+03)
	DTLZ4	3.6834e+02 (6.4508e+00)	2.3839e+02 (1.0780e+00)
	DTLZ5	3.7780e+02 (2.7668e+00)	2.4280e+02 (1.0162e+00)
	DTLZ6	4.4523e+03 (2.9406e+00)	4.3366e+03 (3.7426e+00)
	DTLZ7	9.2249e+00 (3.1569e-01)	7.1779e+00 (4.0208e-02)
Time	DTLZ1	9.2198e+00 (2.4736e-02)	1.8127e+00 (4.0681e-02)
	DTLZ2	9.0389e+00 (2.4060e-02)	2.1148e+00 (1.4466e-02)
	DTLZ3	9.3291e+00 (1.9043e-02)	1.9310e+00 (1.5188e-02)
	DTLZ4	1.3494e+01 (2.1555e+00)	8.5085e+00 (2.0471e-01)
	DTLZ5	9.4792e+00 (6.2154e-02)	3.3992e+00 (5.1943e-02)
	DTLZ6	8.8182e+00 (9.3220e-03)	1.5714e+00 (1.1792e-02)
	DTLZ7	9.7167e+00 (2.7945e-02)	2.3176e+00 (1.1198e-02)

how different hardware configurations affect the performance of the algorithms, with a focus on speedup achieved on GPUs relative to the CPU.

1) *Experimental Settings:* For each algorithm, the population size is set to 10000, with a dimensionality of 1000 and 3 objectives. Each algorithm is independently executed 10 times on each device, with 100 iterations per run. The average execution time across the 10 runs is calculated for each configuration, and the speedup achieved on the GPU relative to the CPU is determined, where speedup is defined as the ratio of execution time on the CPU to that on the GPU.

TABLE S.II

STATISTICAL RESULTS (MEAN AND STANDARD DEVIATION) OF THE RUNTIME (S) AND SPEEDUP FOR EMO ALGORITHMS ON DIFFERENT DEVICES IN DTLZ1

Device	Algorithm	Time	Speedup [†]
CPU	NSGA-III	8.7775e+02 (8.3293e+01)	1.0
	MOEA/D	9.6668e+02 (4.9265e+01)	1.0
	HypE	4.7882e+02 (2.2464e+01)	1.0
RTX 2080Ti	TensorNSGA-III	3.8679e+00 (2.5282e-02)	226.9
	TensorMOEA/D	1.5599e+00 (1.2737e-02)	619.7
	TensorHypE	5.6312e+00 (1.2946e-01)	85.0
RTX 3090	TensorNSGA-III	2.7511e+00 (1.7527e-02)	319.1
	TensorMOEA/D	8.6320e-01 (5.6514e-03)	1119.9
	TensorHypE	2.7511e+00 (1.7530e-02)	174.0
RTX 4090	TensorNSGA-III	1.6648e+00 (1.0636e-02)	527.2
	TensorMOEA/D	6.0490e-01 (5.0061e-03)	1598.1
	TensorHypE	2.4712e+00 (6.0123e-02)	193.8

[†] Speedup = CPU Time / GPU Time.

2) *Comparison Results:* As shown in Table S.II, the type of GPU indeed affects the execution time. However, compared to the CPU, even the less powerful RTX 2080Ti GPU achieves a minimum speedup of 85×, illustrating the scalability of tensorized algorithms across different hardware configurations. The maximum speedup reaches 226.9×. These results highlight the efficiency of tensorized algorithms in utilizing GPU resources, even on relatively low-end consumer hardware. The acceleration effect is more pronounced with higher-end GPUs, reaching a maximum speedup of 1598.1× on the RTX 4090 GPU.

D. Detailed Results for the DTLZ Test Suite

Table S.III presents the results for tensorized and non-tensorized algorithms on the DTLZ test suite. Tensorized algorithms achieve faster runtimes while maintaining similar IGD values. The performance differences are small and consistent across most problems. Some variations are due to the batch operations in tensorized algorithms, which simplify computations compared to the original methods.

E. Experiment in Multiobjective Robot Control Benchmark

For the HV calculation in the multiobjective robot control benchmark experiments, reference points were established based on the minimum values of nondominated solutions obtained by all algorithms for each problem. For objectives related to forward movement and height, if the minimum value was less than 0, the reference value was set to 0. In the MoPusher and MoReacher problems, the reference points were derived by taking the minimum values across all algorithms for each objective. Table S.IV provides the reference points used for the 9 multiobjective robot control problems.

TABLE S.IV
THE REFERENCE POINTS OF 9 MOBOTROL PROBLEMS FOR HV CALCULATION

Problem	Reference Point
MoHalfcheetah	(0, -530.97)
MoHopper	(283.52, -994.39, -1498.70)
MoSwimmer	(0, -0.20)
MoIDP	(3246.31, 153.32)
MoWalker2d	(0, 11.99)
MoPusher	(-1505.66, -984.00, -13645.28)
MoReacher	(-389.79, -1668.30)
MoHumanoid	(340.74, 321.41)
MoHumanoid-s	(4756.48, -6.06)

TABLE S.III

STATISTICAL RESULTS (MEAN AND STANDARD DEVIATION) OF THE IGD AND RUNTIME (S) FOR NON-TENSORIZED AND TENSORIZED EMO ALGORITHMS IN DTLZ1–DTLZ7. ALL EXPERIMENTS ARE CONDUCTED ON AN RTX 4090 GPU AND THE BEST RESULTS ARE HIGHLIGHTED.

Algorithm	Problem	IGD (Non-Tensorized)	IGD (Tensorized)	Time (Non-Tensorized)	Time (Tensorized)
NSGA-III	DTLZ1	1.3619e+05 (3.6592e+02)	1.3693e+05 (4.3756e+02)	7.9940e+01 (2.0346e+00)	1.8314e+00 (1.5300e−02)
	DTLZ2	2.3069e+02 (1.2189e+00)	2.2882e+02 (9.4040e−01)	4.2566e+01 (1.0087e+00)	2.3500e+00 (1.3600e−02)
	DTLZ3	4.6228e+05 (1.1636e+03)	4.6225e+05 (1.2913e+03)	4.7783e+01 (6.8060e−01)	2.1542e+00 (1.2700e−02)
	DTLZ4	2.3785e+02 (1.0874e+00)	2.3572e+02 (8.5302e−01)	5.9497e+01 (9.8333e−01)	8.5898e+00 (2.4899e−01)
	DTLZ5	2.3463e+02 (1.1257e+00)	2.3864e+02 (1.1237e+00)	5.0183e+01 (1.0445e+00)	4.1632e+00 (5.5200e−02)
	DTLZ6	4.2845e+03 (3.4493e+00)	4.3192e+03 (2.4402e+00)	8.3887e+01 (9.5350e−01)	1.8032e+00 (1.2700e−02)
	DTLZ7	6.8165e+00 (4.9090e−02)	7.0588e+00 (4.4870e−02)	6.5831e+01 (9.2720e−01)	2.6303e+00 (5.1900e−02)
MOEA/D	DTLZ1	1.2951e+05 (1.9198e+03)	1.3819e+05 (5.6393e+02)	8.9395e+01 (2.8132e+00)	1.0660e+00 (6.6339e−03)
	DTLZ2	1.2359e+02 (2.0404e+01)	2.0785e+02 (1.3811e+00)	8.8740e+01 (2.6398e+00)	1.0694e+00 (5.3202e−03)
	DTLZ3	4.3746e+05 (4.3569e+03)	4.6991e+05 (1.7973e+03)	9.0152e+01 (3.0706e+00)	1.0701e+00 (3.3552e−03)
	DTLZ4	3.4160e+02 (1.8663e+01)	2.1645e+02 (2.2032e+00)	7.2891e+01 (2.9190e+00)	1.4081e+00 (4.2637e−03)
	DTLZ5	1.2483e+02 (2.4523e+01)	2.0857e+02 (1.7378e+00)	8.6675e+01 (2.9418e+00)	1.0661e+00 (3.9010e−03)
	DTLZ6	4.3760e+03 (5.5493e+00)	4.3915e+03 (3.0552e+00)	8.7844e+01 (3.0132e+00)	1.0704e+00 (2.9017e−03)
	DTLZ7	3.0036e+00 (1.2811e−01)	5.0944e+00 (1.9928e−01)	8.6198e+01 (2.9972e+00)	1.0743e+00 (3.8539e−03)
HypE	DTLZ1	1.3705e+05 (5.2939e+02)	1.3705e+05 (5.2939e+02)	5.7589e+01 (1.5513e+00)	2.3500e+00 (3.9000e−02)
	DTLZ2	1.9759e+02 (1.0991e+00)	1.9759e+02 (1.0991e+00)	5.8689e+01 (1.4897e+00)	3.8541e+00 (4.2200e−02)
	DTLZ3	4.5667e+05 (8.3459e+02)	4.5667e+05 (8.3459e+02)	5.7680e+01 (1.5285e+00)	2.8743e+00 (2.8500e−02)
	DTLZ4	2.0196e+02 (1.0117e+00)	2.0196e+02 (1.0712e+00)	6.7169e+01 (1.4286e+00)	1.1103e+01 (3.8963e−01)
	DTLZ5	2.1053e+02 (1.1076e+00)	2.1053e+02 (1.1076e+00)	6.5312e+01 (1.4286e+00)	1.0254e+01 (1.0990e−01)
	DTLZ6	4.3122e+03 (3.8700e+00)	4.3122e+03 (3.8700e+00)	5.7556e+01 (1.5598e+00)	3.0027e+00 (5.2700e−02)
	DTLZ7	6.1372e+00 (2.9802e−02)	6.1372e+00 (2.9802e−02)	5.6269e+01 (1.7775e+00)	2.9777e+00 (1.2100e−02)