Boosting VLSI Design Flow Parameter Tuning with Random Embedding and Multi-objective Trust-region Bayesian Optimization

SU ZHENG, The Chinese University of Hong Kong Shatin, Hong Kong SAR
HAO GENG, ShanghaiTech University, Shanghai, China
CHEN BAI, BEI YU, and MARTIN D. F. WONG, The Chinese University of Hong Kong, Shatin, Hong Kong SAR

Modern very large-scale integration (VLSI) design requires the implementation of integrated circuits using electronic design automation (EDA) tools. Due to the complexity of EDA algorithms, there are numerous tool parameters that have imperative impacts on the chip design quality. Manual selection of parameter values is excessively laborious and constrained by experts' experience. Due to the high complexity and lack of parallelization, most existing parameter tuning methods cannot make sufficient exploration in a large search space. In this article, we boost the efficiency and performance of parameter tuning with random embedding and multi-objective trust-region Bayesian optimization. Random embedding can effectively cut down the number of variables in the search process and thus reduce the runtime of Bayesian optimization. Multi-objective trust-region Bayesian optimization allows the algorithm to explore diverse solutions with excellent parallelism. Due to the ability to do more exploration in limited runtime, the proposed framework can achieve better performance than existing methods in our experiments.

CCS Concepts: • **Hardware → Physical design (EDA)**;

Additional Key Words and Phrases: Physical design, VLSI design flow, parameter tuning, Bayesian optimization

ACM Reference format:
Su Zheng, Hao Geng, Chen Bai, Bei Yu, and Martin D. F. Wong. 2023. Boosting VLSI Design Flow Parameter Tuning with Random Embedding and Multi-objective Trust-region Bayesian Optimization. *ACM Trans. Des. Autom. Electron. Syst.* 28, 5, Article 74 (September 2023), 23 pages.
https://doi.org/10.1145/3597931

This work is supported by AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong, The Innovation and Technology Fund (No. PRP/065/20FX), The Research Grants Council of Hong Kong SAR (Project No. CUHK1409420), and Shanghai Pujiang Program (Project No. 22PJ1410400).
Authors' addresses: S. Zheng, C. Bai, B. Yu (corresponding author), and M. D. F. Wong, Rm905, Ho Sin Hang Engineering Building, The Chinese University of Hong Kong, Shatin, Hong Kong SAR; emails: {szheng22, cbai, byu}@cse.cuhk.edu.hk, mdfwong@cuhk.edu.hk; H. Geng (corresponding author), 3-332, SIST Building, ShanghaiTech University, 393 Middle Huaxia Road, Pudong New District, Shanghai, China; email: genghao@shanghaitech.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1084-4309/2023/09-ART74 $15.00
https://doi.org/10.1145/3597931

1 INTRODUCTION

Electronic design automation (**EDA**) tools play a central role in modern VLSI design flow, which comprises a front-end and a back-end flow. For example, Genus, a Cadence synthesis tool, delivers a solution to the front-end flow. The physical design tool, e.g., Innovus, can be regarded as the implementation of the back-end flow, which contains multiple steps such as placement, clock tree synthesis, routing, and so on. There exist numerous parameters in tool-based flow. For example, partitioning is the process of disassembling designs into more manageable block sizes. Setting the parameter `auto partition` in Genus can enable the partition algorithm for our design. In Innovus, `congestion effort` achieves the trade-off between the global placement runtime cost and the placement quality, which reveals how many areas within a chip floorplan will be difficult to route.

In the industry, the manual selection for tool parameters is widely applied by chip designers. Nevertheless, such a method is extremely laborious and lacks scalability for newly-emerged design if designers can only transfer limited prior knowledge. Chip designers also choose the exhaustive search for the parameter values. Unfortunately, it lacks efficiency since the enormous search space restricts designers from enumerating each parameter value and retrieving the optimal one. Therefore, an efficient parameter tuning methodology is crucial for improving VLSI design automation.

Given the difficulty, many researchers propose various solutions in academia [1, 14, 16, 18, 20, 26, 32, 34, 35]. SynTunSys [35] is a self-evolving system for the parameter tuning of synthesis tools, which is extended in [34]. LAMDA [26] is an auto-tuning algorithm for FPGA design closure utilizing the XGBoost algorithm [7] and design-specific features extracted from the design flow. Reference [18] adopts the tensor decomposition idea from recommender systems and suggests parameter values based on a neural network. Reference [1] optimizes placement parameters with a deep reinforcement learning framework based on handcrafted features and graph neural networks [15]. FIST [32] leverages the proposed feature importance sampling method and XGBoost regressor to optimize the parameters. FlowTuner [19] incorporates **ant colony optimization** (**ACO**) algorithms to build a cooperative co-evolutionary framework for parameter tuning. The AutoTuner platform [16] integrates several optimization algorithms, such as evolutionary algorithm and tree-structured Parzen estimator [6]. Reference [20] utilizes **Bayesian optimization** (**BO**) to tune the tool parameters efficiently. PTPT [14] uses multi-objective BO to optimize VLSI design flow parameters, which can model the correlations among multiple objectives and obtain a Pareto-optimal set of parameter values.

In general, existing methods for VLSI design flow parameter tuning can be classified into two categories, i.e., heuristic search and model-based search. Heuristic search methods usually search parameter values by successively applying minor changes according to pre-defined rules. Such rules, e.g., the mutation criterion in the genetic algorithm, are applied based on a pre-determined threshold and current results quality, and so on. Most heuristic search methods require more iterations to find the global optimum with a certain probability. Yet such convergence is still not guaranteed. Regarding solution quality, model-based methods often outperform the heuristic search methods. Model-based search methods are capable of efficient global search by balancing exploration and exploitation based on a manually designed process [32] or the merits of a robust black-box model, e.g., the **Gaussian process** (**GP**) model [14, 20]. BO, deep reinforcement learning, and so on fall into this category. The black-box model utilized by the methodology can guide how to explore the search space and focus on regions that are likely to have excellent results. However, two limitations restrict current model-based search methods from improving performance and efficiency further. Firstly, more parallelism is failed to exploit. Secondly, the high runtime cost is often consumed in training the black-box models. Notwithstanding that some arts have already

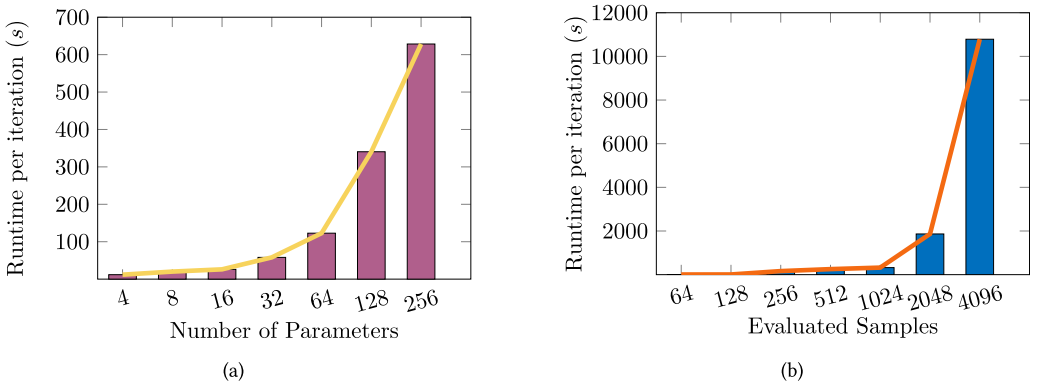

Fig. 1. Runtimes per iteration of MOTPE [21], an efficient multi-objective Bayesian optimization method. (a) shows the runtimes of MOTPE with different numbers of parameters. The number of evaluated samples is 1,024. (b) presents the runtimes of MOTPE with different numbers of evaluated samples. We test the algorithm with 134 parameters, the same as we use in the experiments in Section 4. Table 1 presents information about the parameters. As the number of parameters or evaluated samples increases, the runtime for an iteration rises rapidly.

extended the sequential optimization to the parallel flow [8, 9] to solve the first limitation, the runtime cost may still become unacceptable for the VLSI design flow. The reason lies in two folds. On the one hand, there exist many parameters in the search space, e.g., at least, more than a hundred parameters that can affect the **quality of result** (**QoR**). Moreover, the search space may contain uncountable candidate solutions due to non-discrete parameter values in the VLSI design flow. Hence, it requires more optimization iterations to obtain a satisfactory result. On the other hand, as the algorithm continues to optimize, the black-box model is trained on a larger searched data size, which incurs high training runtime costs. Methods based on BO face the problem since training becomes more expensive as the optimization continues.

VLSI design flow takes a long time to obtain the QoRs. The relation between the tool inputs and the QoRs is so complicated that we can not model it with simple analytic functions. It fits the black-box optimization settings of BO. BO can make a trade-off between exploration and exploitation, enabling an efficient global search. Methods based on BO have shown better performance than other methods such as genetic algorithm in existing research [14, 16]. Therefore, we propose a framework based on BO that can solve the limitations mentioned earlier.

To improve the efficiency of BO for VLSI design flow parameter tuning, we argue that the time required for calibrating the surrogate model and computing acquisition function must be reckoned with when navigating the huge searching space generated by a plethora of tool parameters. Figure 1(a) and Figure 1(b) present the runtimes of an efficient multi-objective BO method MOTPE [21] on a problem with three objectives. Each bar in Figure 1 denotes the algorithm's runtime per iteration. The method uses the evaluated samples as training data to train the surrogate model and select new promising data from the search space based on the surrogate model. In Figure 1(a), the number of evaluated samples is 1,024. It shows that the number of parameters strongly influences the runtime, which inspires us to reduce the runtime of the parameter tuning algorithm via dimensionality reduction. In Figure 1(b), the number of parameters is 134. As the number of evaluated samples increases, the runtime rises rapidly, which can become unaffordable. For example, MOTPE with 4,096 evaluated samples costs 10,786 seconds to sample new promising data. Thus, keeping the number of evaluated samples for BO in an acceptable range is crucial to prevent the parameter tuning process from slowing down significantly.

Our first improvement on the parameter tuning method is dimensionality reduction. Although there are a large number of parameters in EDA tools that influence the QoR, some of them are more important than others for a given circuit. For example, the congestion effort parameter is critical for a circuit with potential congestion issues in the placement and routing processes. At the same time, the wire length optimization level may have less effect. In other words, the QoRs is strongly affected by a latent variable vector with lower dimensionality than the parameter space. This fact motivates us to apply dimension reduction to the search space, i.e., utilizing low-dimension features to represent the original high-dimension parameters. Although, we can implement dimensionality reduction by selecting important parameters and ignoring others, it requires many additional data to estimate the importance of each parameter in a huge parameter space, which can significantly slow down the parameter tuning process.

We reduce the dimensionality of the search space with the random embedding method [28]. The dimensionality of a latent space that determines the results is named effective dimensionality d_e. Assuming the D-dimensional variable vector $\boldsymbol{x}$ has an effective dimensionality of d_e, it is proved in [28] that the optimal point of $\boldsymbol{x}$ can be found by optimizing the d_e-dimensional variable vector $\boldsymbol{x}'$, which is mapped to the D-dimensional space with a random embedding matrix. The theory inspires algorithm designers to cut down the number of variables by searching in a space with lower dimensionality, which can effectively reduce the runtime of optimization algorithms.

To enhance the efficiency of parameter tuning, we adopt an optimized multi-objective BO method as our second improvement. Sequential algorithms suffer from the unsatisfactory efficiency. Parallel BO [8, 9] conquers this challenge by sampling a batch of points at each iteration. By sampling more points in one iteration, parallel BO can explore the search space better than sequential methods. However, as shown in Figure 1(b), having many evaluated samples significantly slows down the optimization process, which prohibits the parameter tuning method from having a large batch size. Trust-region BO [10, 12] enhances the parallelism by performing BO within multiple decoupled **trust regions** (**TRs**) simultaneously. Each trust region only uses a portion of the evaluated samples to train the models, which can effectively reduce the runtime of BO and achieve larger-scale parallelism. The adaptive sampling range of trust-region BO can avoid the probable deterioration of search quality caused by the decrease in training data.

The third improvement is an early-stopping mechanism that utilizes the feature of the EDA flow. We also consider the multiple consecutive optimization stages in the de facto VLSI design flow, i.e., the results from the front-end flow strongly influence the QoRs obtained from the back-end flow. Besides, the back-end flow consists of more stages and complex algorithms than the front-end, which incurs higher runtime costs than the front-end flow. This fact motivates us to select parameter values that are more likely to have good QoRs based on the front-end results. We first evaluate multiple points with the front-end flow and select a portion of them that potentially have high QoRs to run the back-end flow. As a result, some samples that fail at the early-stage can also be abandoned, utilizing the finding that early-stage failure costs less time mentioned in [16]. Previous methodologies neglect the interaction between the front-end and back-end flow. Unlike them, our algorithm is aware of the interaction accordingly to solve the problem.

In accordance with the aforementioned arguments and observations, we propose REMOTune, a parameter tuning framework for VLSI design flows with **R**andom **E**mbedding and **M**ulti-**O**bjective trust-region BO. First, we reduce the number of variables for the optimization algorithm by generating the parameter values in a space with lower dimensionality. Second, we overcome the weakness of sequential optimization by utilizing parallel BO. The high efficiency of trust-region BO [10, 12] contributes to the superiority of REMOTune. Finally, the non-dominated sorting [11] of the results from the early stage of the EDA flow can select the samples that are likely to surpass others and reduce the number of evaluations in the time-consuming back-end flow.

The main contributions of this article can be summarized as follows.

- —We propose an efficient algorithm for VLSI design flow parameter tuning. To overcome the challenge of the enormous parameter space, we effectively reduce the number of variables with random embedding, which significantly saves the time spent on the optimization method.
- —We enable parallel optimization in parameter tuning with an improved multi-objective trust-region BO method. Optimizing in several decoupled regions can not only enable parallel evaluations but also reduce the runtime for the parameter tuning method. In addition, we introduce a clustering step in the initialization of the method to ensure the diversity of the TRs.
- —In the early stage of the EDA flow, we apply a non-dominated sorting mechanism to select the samples that are likely to surpass others and reduce the number of evaluations in the time-consuming back-end flow.
- —Experimental results show that the proposed framework can achieve significant improvement compared with existing EDA flow parameter tuning methods.

The rest of our article is organized as follows. Section 2 introduces some prior knowledge about random embedding and multi-objective optimization. Section 3 discusses the developed parameter tuning flow. Section 4 describes our experiments on VLSI design parameter tuning. The conclusion and future directions are discussed in Section 5.

2 PRELIMINARIES

2.1 Bayesian Optimization

BO is an efficient framework for solving black-box optimization problems, where the relations between the inputs and outputs are too complex to model with some analytical functions. For a problem $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}\in X} f(\boldsymbol{x})$, BO utilizes a surrogate model to simulate the objective function $y = f(\boldsymbol{x})$. Typically, the surrogate model is a GP: :

$$p(y|\boldsymbol{x}) = \mathcal{N}(\mu(\boldsymbol{x}), \Sigma(\boldsymbol{x})), \tag{1}$$

where $\mathcal{N}(\mu(\boldsymbol{x}), \Sigma(\boldsymbol{x}))$ denotes a Gaussian distribution with a mean function $\mu(\boldsymbol{x})$ and a covariance matrix $\Sigma(\boldsymbol{x})$. The GP can predict the mean and variance of the points, which can be used to make a trade-off between exploitation and exploration.

GP usually utilizes a covariance function $k(\boldsymbol{x}_1, \boldsymbol{x}_2)$ to achieve non-linear modeling. For example, the widely-used squared exponential function is defined as:

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \lambda^2 \exp\left(-\frac{1}{2}(\boldsymbol{x}_1 - \boldsymbol{x}_2)^\top \boldsymbol{\Lambda}(\boldsymbol{x}_1 - \boldsymbol{x}_2)\right), \tag{2}$$

where $\boldsymbol{\Lambda} = \text{diag}(\lambda_1^{-2}, \lambda_2^{-2}, \ldots, \lambda_D^{-2})$ is the diagonal length scale matrix and λ^2 is a scaling factor.

Given observations $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]$, $\boldsymbol{y} = [y_1, y_2, \ldots, y_N]$, and a new point $\boldsymbol{x}_t$, the distribution of y_t [30] is estimated by:

$$p(y_t|\boldsymbol{x}_t, \boldsymbol{X}, \boldsymbol{y}) = \mathcal{N}(\boldsymbol{k}^\top(\boldsymbol{X}, \boldsymbol{x}_t)\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})^{-1}\boldsymbol{y}, k(\boldsymbol{x}_t, \boldsymbol{x}_t) - \boldsymbol{k}^\top(\boldsymbol{X}, \boldsymbol{x}_t)\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})^{-1}\boldsymbol{k}(\boldsymbol{X}, \boldsymbol{x}_t)), \tag{3}$$

where $\boldsymbol{k}(\boldsymbol{X}, \boldsymbol{x}_t)$ denotes the vector formed by the covariance values between $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]$ and $\boldsymbol{x}_t$. $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})$ is the intra-covariance matrix among $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]$, whose element at the ith row and jth column is $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Equation (3) can explain why the runtime of BO rises rapidly as the number of observations increases, as shown in Figure 1(b). As the number of data increases, the time to get $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X})^{-1}$ grows polynomially, which significantly slows down the computation of GP in BO. Therefore, limiting the number of observations involved in the GPs can effectively reduce the runtime of a parameter tuning method based on BO.

BO uses an acquisition function to determine which datum to evaluate in the next step. Under a single-objective optimization setting, the **expected improvement** (**EI**) is a popular acquisition function defined as:

$$EI(\boldsymbol{x}) = \int_{-\infty}^{+\infty} \max(y^* - y, 0) p(y|\boldsymbol{x}) dy. \tag{4}$$

$EI(\boldsymbol{x})$ prefers low expected value or high uncertainty, which implements the trade-off between exploitation and exploration. Maximizing the acquisition function usually requires a numerical optimization process, which costs more and more time as the number of variables increases. Thus, limiting the number of variables can reduce the runtime of BO because less time is spent maximizing the acquisition function.

After selecting the test point, the evaluation result is used to train the surrogate model. The best evaluated result is regarded as the optimal value $y^* = f(\boldsymbol{x}^*)$. BO does not assume any particular form for $f(\boldsymbol{x})$, which makes it suitable for problems where the evaluation process is complicated or the design of analytic models is difficult.

2.2 Multi-objective Optimization

In EDA flows, there are multiple QoR metrics, such as **performance, power, and area** (**PPA**). Thus, the parameter tuning for EDA flows is intrinsically a multi-objective problem. In multi-objective optimization, we usually find the Pareto-optimal set rather than a single optimal result. For a problem with M objectives, we denote the objectives of a test point $\boldsymbol{x}$ as a vector $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. An objective vector $\boldsymbol{y}_1 = [y_1^{(1)}, y_1^{(2)}, \ldots, y_1^{(M)}]$ is said to **dominate** $\boldsymbol{y}_2$ if

$$\forall i \in \{1, 2, \ldots, M\}, y_1^{(i)} \leq y_2^{(i)} \text{ and } \exists j \in \{1, 2, \ldots, M\}, y_1^{(j)} < y_2^{(j)}. \tag{5}$$

A point x is **Pareto-optimal** if no evaluated point dominates it. The Pareto-optimal points are called the Pareto-optimal set.

To compare results under a multi-objective setting, the **hypervolume (HV) indicator** is commonly used to assess the quality of Pareto-optimal sets. Given a Pareto-optimal set $\boldsymbol{P}$ and a reference point $\boldsymbol{r}$ that typically denotes the worst case, the HV indicator can be defined as:

$$HV(\boldsymbol{P}, \boldsymbol{r}) = \Lambda\left(\bigcup_{\boldsymbol{x}_i \in \boldsymbol{P}} [f(\boldsymbol{x}_i)^{(1)}, r^{(1)}] \times [f(\boldsymbol{x}_i)^{(2)}, r^{(2)}] \times \cdots \times [f(\boldsymbol{x}_i)^{(M)}, r^{(M)}]\right), \tag{6}$$

where $\boldsymbol{x}_i$ is the ith point in the Pareto-optimal set. $[f(\boldsymbol{x}_i)^{(j)}, r^{(j)}]$ denotes the range between the jth elements of $\boldsymbol{f}(\boldsymbol{x}_i)$ and $\boldsymbol{r}$. $\Lambda(\cdot)$ refers to the Lebesgue measure. Figure 2(a) illustrates the HV indicator in a 2-D space. It computes the area of a polygon that is the union of multiple rectangles. Each rectangle is determined by a Pareto-optimal point and the reference point. In Figure 2(b), a new point enlarges the Pareto-optimal set and thus increases the HV. In simple terms, the HV indicator computes the size of the space covered by the Pareto-optimal set.

In multi-objective BO, the acquisition functions are required to consider the Pareto-optimal set. For example, **expected hypervolume improvement** (**EHVI**) [8] extends the idea of EI and acts as a multi-objective acquisition function. **Max-value entropy search** (**MES**) [29] considers the optimization of the Pareto-optimal set from an information-theoretic perspective.

3 PROPOSED METHOD

3.1 Overview

Figure 3 shows the overview of the proposed framework. To obtain layout designs from the EDA flow, we need to prepare the technology library, **hardware description language** (**HDL**) codes, and constraint files. The range of each parameter should also be specified. Given these files and

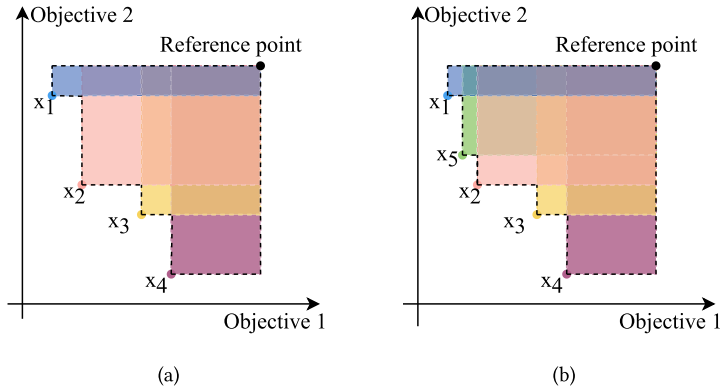

Fig. 2. Illustration of the hypervolume indicator in a 2-D space. The hypervolume indicator computes the area of a polygon that is the union of multiple rectangles. Each rectangle is determined by a Pareto-optimal point and the reference point. (a) shows the 2-D hypervolume of four Pareto-optimal points. (b) is the 2-D hypervolume after adding the fifth point. The difference between (a) and (b) is the hypervolume contribution of the fifth point.

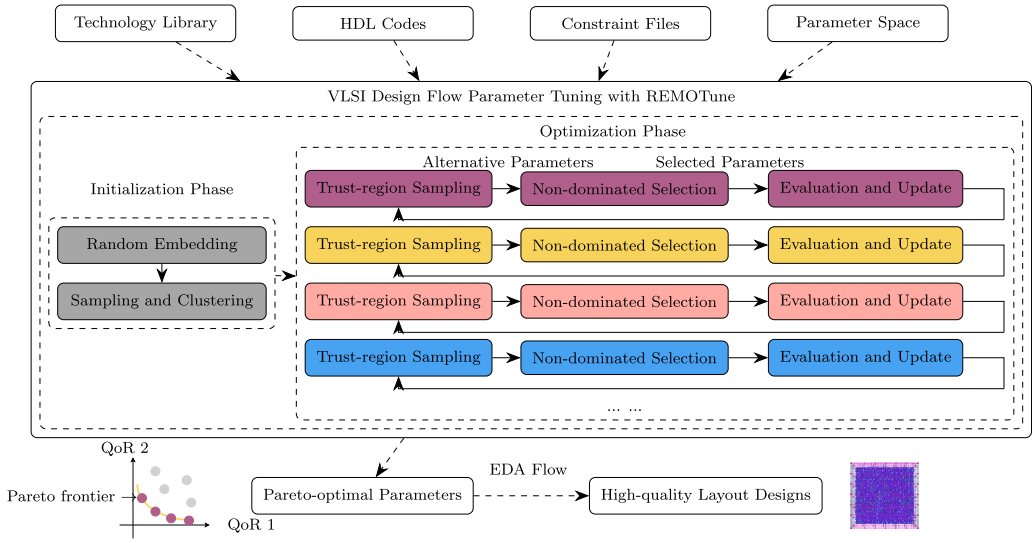

Fig. 3. Overview of REMOTune, which consists of the initialization phase and the optimization phase.

the parameter space, REMOTune can optimize the parameters for the EDA flow and finally output the Pareto-optimal parameter values.

REMOTune consists of two phases, the initialization phase and the optimization phase. The initialization phase consists of two steps. First, we generate a random embedding matrix that can map a variable vector in a low-dimensional space $\mathcal{R}^d$ to the original parameter space $\mathcal{R}^D$. Second, we sample and evaluate multiple points, then utilize the K-Means++ clustering algorithm [27] to partition the samples into T subsets, each of which is used to initialize a trust region.

REMOTune optimizes the parameters with multiple decoupled TRs, which can run in parallel. At each iteration, we first train the GPs for each trust region. After training the surrogate models, we sample a batch of points with Thompson sampling, considering the contribution to the HV. Second,

the sampled points are evaluated with the front-end flow, such as the synthesis tool Genus. Half of them are selected by non-dominated sorting, which can select the points that are more likely to improve the Pareto-optimal set. In the non-dominated sorting, we select the points according to Pareto-optimality and the crowding distance. Section 3.3.2 presents the details about this process. Finally, the selected points and their QoR results from the back-end flow (e.g., Innovus) are used to update the Pareto-optimal set and added to the training data for the next iteration.

3.2 Initialization Phase

In the initialization phase, we first generate a random embedding matrix for dimensionality reduction and second initialize the TRs with clustered points.

3.2.1 Random Embedding. There are a large number of parameters in well-known EDA tools such as Genus and Innovus. The huge parameter space poses a great challenge to the parameter tuning methods. As discussed in Sections 1 and 2.1, a large number of variables can increase the runtime of BO. Therefore, we try to optimize the parameters in a space with lower dimensionality. In this article, we utilize random embedding to reduce the dimensionality of the search space. Assuming that the parameter vector $\boldsymbol{x}$ is D-dimensional and a d_e-dimensional latent variable vector $\boldsymbol{x'}$ dominates the QoR results, it is proved in [28] that optimizing $\boldsymbol{x'}$ is equivalent to optimizing $\boldsymbol{x}$ if $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x'}$ and each element of $\boldsymbol{A}$ is sampled from a standard Gaussian distribution. The theory inspires us to reduce the dimensionality of the search space via random embedding.

We normalize each parameter value in $[-1, 1]$ for simplicity. Discrete parameters are also represented by real numbers in $[-1, 1]$. Thus, the parameter vector $\boldsymbol{x}$ has a box bound $\boldsymbol{B}_D = [-1, 1]^D$. The latent variable vector $\boldsymbol{x'}$ is also limited in a pre-defined box $\boldsymbol{B}_e = [-b_e, b_e]^{d_e}$. In practice, the parameter vector generated by $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x'}$ is likely to fall outside $[-1, 1]^D$ and needs to be clipped. To encourage the searching inside $\boldsymbol{B}_D$, we generate the random embedding matrix $\boldsymbol{A}$ by optimizing the following problem with the simulated annealing algorithm.

$$\boldsymbol{A} = \arg\min_{\boldsymbol{A}}(1 - p(\boldsymbol{A}\boldsymbol{x'} \in \boldsymbol{B}_D)) + \gamma \times D_{KL}(p_{\boldsymbol{A}} \| \phi), \tag{7}$$

where $p(\boldsymbol{A}\boldsymbol{x'} \in \boldsymbol{B}_D)$ represents the probability that $\boldsymbol{A}\boldsymbol{x'}$ falls into $\boldsymbol{B}_D$, $D_{KL}(p\|q) = \sum p(x) \log \frac{p(x)}{q(x)}$ is the KL divergence, $p_{\boldsymbol{A}}$ is the probability distribution of the elements of $\boldsymbol{A}$, ϕ denotes the standard Gaussian distribution, and γ is a weight parameter that makes a trade-off between the two terms. The probability $p(\boldsymbol{A}\boldsymbol{x'} \in \boldsymbol{B}_D)$ is evaluated by random sampling. Minimizing $D_{KL}(p_{\boldsymbol{A}}\|\phi)$ ensures that the distribution of the elements of $\boldsymbol{A}$ is standard Gaussian. At each iteration, one row of the embedding matrix $\boldsymbol{A}$ is re-generated with a Gaussian random number generator, and the simulated annealing mechanism decides whether to accept the change or not. In this article, we do simulated annealing to improve the embedding matrix for 4096 iterations. At each iteration, we randomly generate 4,096 points and count the number of points that fall into $\boldsymbol{B}_D$ to estimate the probability $p(\boldsymbol{A}\boldsymbol{x'} \in \boldsymbol{B}_D)$. The pre-defined box $\boldsymbol{B}_e = [-b_e, b_e]^{d_e}$ is set to be $[-0.25, 0.25]^{d_e}$. Since the random generation of points are not time-consuming, the whole process of simulated annealing only costs several minutes.

3.2.2 Sampling and Clustering. After the generation of $\boldsymbol{A}$, we randomly sample multiple points in $\boldsymbol{B}_e$, map them to the original parameter space $\boldsymbol{B}_D$, and evaluate them in the complete VLSI design flow. To ensure the diversity of the TRs, we utilize the K-Means++ clustering algorithm to partition the random samples into T subsets, where T is the number of TRs. Each cluster contains the initial training data of the corresponding trust region. The details of TRs are discussed in Section 3.3. In the absence of clustering, the search directions of the TRs do not have significant differences, which may lead to redundant searches in the optimization process.

ALGORITHM 1: Initialization Phase

Input: Number of initial points N_{init}, number of trust regions T
Output: Trust regions $\{\boldsymbol{TR}_1, \boldsymbol{TR}_2, \ldots, \boldsymbol{TR}_T\}$

1: **function** `Init`$((N_{init}, T))$
// 1. Generating the random embedding matrix
2: Initialize $\boldsymbol{A}$, each element $A_{i,j}$ is sampled from $\mathcal{N}(0, 1)$;
3: Optimize (7) with the simulated annealing algorithm;
// 2. Sampling and Clustering
4: Draw N_{init} quasi-random points from a Sobol sequence;
5: Partition the N_{init} points into groups $\{\boldsymbol{C}_1, \boldsymbol{C}_2, \ldots, \boldsymbol{C}_T\}$ with the K-Means++ clustering algorithm;
6: **for** i in $\{1, 2, \ldots, T\}$ **do**
7: Map the points in $\boldsymbol{C}_i$ to the original parameter space with $\boldsymbol{A}$, then evaluate them with the complete EDA flow;
8: Add the points in $\boldsymbol{C}_i$ and their QoR results to the training data of $\boldsymbol{TR}_i$;
9: **end for**
10: **return** $\{\boldsymbol{TR}_1, \boldsymbol{TR}_2, \ldots, \boldsymbol{TR}_T\}$;
11: **end function**

The initialization phase is summarized in Algorithm 1, which consists of two steps, generating the random embedding matrix (lines 2 and 3) and initializing the Trs with the clustered initial points (lines 4–9).

3.3 Optimization Phase

As discussed in Sections 1 and 2.1, the number of observations has a huge impact on the runtime of BO. As the number of data increases, the runtime for GP inference grows rapidly, which significantly slows down the computation of BO. Although this article aims to utilize parallel BO, the runtime of the optimization algorithm can become unaffordable as the number of evaluated points grows, as shown in Figure 1(b). Therefore, to reduce the runtime of REMOTune, we limit the number of observations involved in the GPs by performing BO within multiple TRs simultaneously.

Trust-region Bayesian optimization (TuRBO) [10, 12] is a robust framework for high-dimensional black-box optimization that can avoid over-exploration by performing optimization in adaptive TRs. TuRBO is designed for the parallel evaluation setting with large batch sizes, which can bring high throughput and thus minimize the end-to-end optimization time. It can provide trust-region-level and batch-level parallelism based on the **sequential model-based optimization (SMBO)** scheme. The TRs are decoupled and optimized in parallel. Within a trust region, a batch of points are sampled and evaluated in parallel at each iteration. In this article, we design an extended multi-objective trust-region BO algorithm that supports random embedding with improved initialization and sampling methods.

Figure 4 illustrates TuRBO in a 2-D space, where the circles with the same color represent the evaluated points belonging to the same trust region. TuRBO explores the search space with multiple decoupled TRs. Each trust region has an adaptive edge-length L and a center point $\boldsymbol{x}_{center}$ that is selected from its evaluated points. A trust region uses its evaluated points to train the GPs and samples one or multiple points in the rectangle determined by the center point and the edge-length. Note that some evaluated points in a trust region can be outside the rectangle because the center point and edge-length can be changed at any iteration. The edge-length is increased and decreased to encourage exploration and exploitation, respectively. Changing the center point enables the trust region to travel around the search space and make better exploration.

Optimizing in decoupled TRs has two significant advantages. First, it effectively reduces the number of training data and the size of the search space for BO. For a typical parallel BO algorithm,

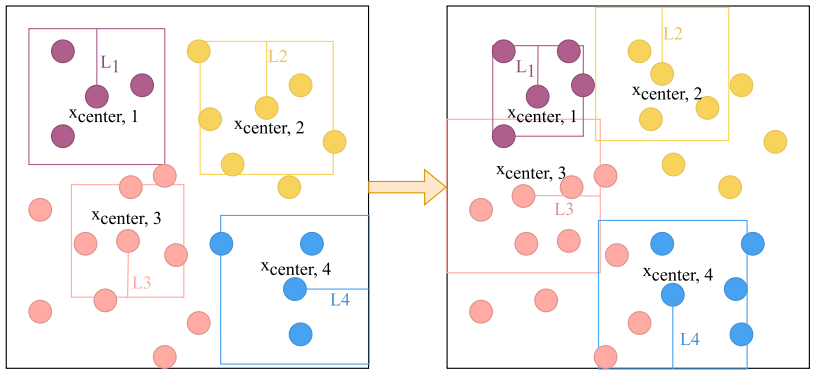

Fig. 4. Overview of TuRBO. Each trust region has an adaptive edge-length L and a center point $\boldsymbol{x}_{center}$ selected from its evaluated points. In each iteration, it samples one or multiple points in the rectangle determined by the center point and the edge-length. The edge-length can be increased and decreased to encourage exploration and exploitation, respectively.

all evaluated points are used as training data, and the algorithm needs to optimize the acquisition function in the whole space. Figure 1(b) has shown that the runtime rises rapidly as the number of training data increases. Nevertheless, in trust-region BO, a trust region uses only a portion of data. It maximizes the acquisition function in a limited space determined by the center point and edge-length. Second, trust-region BO focuses on regions that potentially improve the QoR, avoiding over-exploration in the immense space. This feature of TuRBO enables fast convergence and reduces the number of iterations. Due to the time-consuming EDA flow, it is of great importance for the parameter tuning method to limit the number of iterations. Otherwise, the runtime of the whole parameter tuning process can be unacceptable.

The key points of the optimization phase are as follows.

3.3.1 Trust-region Sampling. Our algorithm carries out BO in several decoupled TRs. A TR samples new points in a hypercube with a center point $\boldsymbol{x}_{center}$ and an edge-length $L \in [L_{\min}, L_{\max}]$. At each iteration, it utilizes GPs to model the relation between the input variables and the output results, and sample one or several points within the trust region using Thompson sampling [17], which is an efficient substitute for acquisition functions.

We select the center point $\boldsymbol{x}_{center,i}$ for each trust region $\boldsymbol{TR}_i$. The center is the evaluated point in $\boldsymbol{TR}_i$ that has the highest **hypervolume contribution** (**HVC**), which is defined as the reduction in HV if that point were to be removed. The HVC of point $\boldsymbol{x}$ is represented by $HVC(\boldsymbol{x}, \boldsymbol{TR}_i)$. The edge-length L_i is initialized with an empirical value and evolves according to the rules described in Section 3.3.3. We initialize the edge-length with 0.25. The basic idea of edge-length evolution is that if the TR succeeds in enlarging the Pareto-optimal set, its edge-length is increased to encourage exploration. Otherwise, the edge-length is decreased to facilitate exploitation.

For the ith trust region whose center point is $\boldsymbol{x}_{center,i}$ and edge-length is L_i, the training data consist of the evaluated points in this trust region and the points from other TRs that are in the hypercube with the same center and an edge-length of $2L$. This data-sharing mechanism can increase the data diversity and improve the modeling ability of the models near the trust region with little overhead.

To generate a batch of data including b points, we use Thompson sampling to draw posterior samples from the GPs. The $2b$ points with the maximum **hypervolume improvement** (**HVI**) are selected to run the front-end flow. Denoting the QoR results of evaluated points with $Y_{tested,i}$,

HVI of a sampled point $\boldsymbol{x}_{sampled}$ is evaluated by:

$$HVI(\boldsymbol{x}_{sampled}) = HV(\text{Pareto}(\boldsymbol{Y}_{tested,i} + \hat{f}(\boldsymbol{x}_{sampled})), \boldsymbol{r}) - HV(\text{Pareto}(\boldsymbol{Y}_{tested,i}), \boldsymbol{r}), \quad (8)$$

where $\hat{f}(\boldsymbol{x}_{sampled})$ is the estimated QoR results obtained in the sampling process and Pareto$(\cdot)$ computes the Pareto-optimal set.

3.3.2 Non-dominated Selection. After the sampling, we evaluate the $2b$ points with the front-end flow and get the early-stage result $\boldsymbol{f}_f(\boldsymbol{x}_i)$ of each point $\boldsymbol{x}_i$. Then we run a non-dominated selection process inspired by NSGA-II [11] to find the better half of the samples, denoted by $\boldsymbol{X}_{selected}$. The selection process iteratively executes the following steps until b samples are selected.

(1) Get the Pareto-optimal set of the samples $\boldsymbol{X}_{sampled}$ and denote them with $\boldsymbol{P}_{sampled}$. Remove the points in $\boldsymbol{P}_{sampled}$ from $\boldsymbol{X}_{sampled}$.
(2) Sort $\boldsymbol{P}_{sampled}$ according to the crowding distance of each point $CD_i = \sum_{j=1}^{M} CD_{i,j}$. To compute $CD_{i,j}$, the early-stage results $\boldsymbol{f}_f(\boldsymbol{x}_i)$ are sorted by the jth dimension. We denote the two points near $\boldsymbol{f}_f(\boldsymbol{x}_i)$ with $\boldsymbol{f}_f(\boldsymbol{x}_i^{\uparrow})$ and $\boldsymbol{f}_f(\boldsymbol{x}_i^{\downarrow})$ such that $f_f(\boldsymbol{x}_i^{\uparrow})^{(j)} \geq f_f(\boldsymbol{x}_i)^{(j)} \geq f_f(\boldsymbol{x}_i^{\downarrow})^{(j)}$. The value $CD_{i,j}$ is computed by:

$$CD_{i,j} = \frac{f_f(\boldsymbol{x}_i^{\uparrow})^{(j)} - f_f(\boldsymbol{x}_i^{\downarrow})^{(j)}}{f_f(\boldsymbol{x}^{\uparrow,j})^{(j)} - f_f(\boldsymbol{x}^{\downarrow,j})^{(j)}}, \quad (9)$$

where $\boldsymbol{x}^{\uparrow,j}$ and $\boldsymbol{x}^{\downarrow,j}$ represent the points that have the maximum and minimum QoR results in the jth dimension, respectively. The $CD_{i,j}$ values of $\boldsymbol{x}^{\uparrow,j}$ and $\boldsymbol{x}^{\downarrow,j}$ are set to $+\infty$.
(3) If $|\boldsymbol{X}_{selected}| \leq b - |\boldsymbol{P}_{sampled}|$, add all points in $\boldsymbol{P}_{sampled}$ to $\boldsymbol{X}_{selected}$. Otherwise, select the points in $\boldsymbol{P}_{sampled}$ according to the non-dominated sorting. The points with larger crowding distances are selected first.

The non-dominated selection can be regarded as an early-stopping mechanism that enables us to explore more points while keeping the overhead at an acceptable level. The front-end flow usually costs less time than the back-end, but it has a strong influence on the final QoRs. Non-dominated selection enables us to explore more points with a little overhead and use the refined samples to evaluate the results. We use half of the candidates empirically to make a trade-off between effectiveness and efficiency. Using fewer candidates has little gain on exploration while using more candidates leads to a high runtime overhead. In our experiments, it costs around 1/4 additional runtime than the method without non-dominated selection but can explore much more samples.

3.3.3 Evaluation and Update. Finally, the selected points and their QoR results from the back-end flow are used to update the training data and the Pareto-optimal set. The edge-length L is also updated after the evaluation. It is initialized with L_{init} in the initialization phase and changed according to the following rules.

(1) If the best point or the Pareto-optimal set of one trust region is improved at the current iteration, the success count of this trust region is increased. Otherwise, the failure count is incremented.
(2) When the success count of one trust region exceeds the success threshold τ_s, the edge-length is increased to $\min\{2L, L_{\max}\}$, encouraging the exploration in a larger space. The success count is set to zero after changing the edge-length.

(3) When the failure count of one trust region exceeds the failure threshold τ_f, the edge-length is set to $L/2$, encouraging more fine-grained searches. The failure count is set to zero after changing the edge-length.
(4) If a trust region has $L < L_{min}$, it is terminated and a new trust region is started.

Algorithm 2 summarizes an optimization iteration of a trust region. Line 2 selects the center point of the trust region. Lines 3 and 4 show the TR sampling processes, respectively. Lines 5–10 present the non-dominated selection step. Lines 12 and 13 show the evaluation of selected samples and the update of the trust region, respectively. In the optimization phase, we execute multiple instances of Algorithm 2 in parallel to explore multiple regions concurrently.

ALGORITHM 2: Optimization Phase

Input: Trust region $\boldsymbol{TR}_i$, batch size b
Output: Updated trust region $\boldsymbol{TR}_i$
1: **function** `Optimize`($(\boldsymbol{TR}_i, b)$)
// 1. Trust-region Sampling
2: $x_{center,i} = \arg\max_x HVC(x, \boldsymbol{TR}_i)$;
3: Train the Gaussian processes with the evaluated points in $\boldsymbol{TR}_i$ and the shared data;
4: Sample $2b$ points according to $HVI(x)$;
// 2. Non-dominated selection
5: Map samples $\boldsymbol{X}_{sampled}$ to the original parameter space, evaluate them with the front-end flow, get results $\boldsymbol{Y}_{f,sampled}$;
6: $\boldsymbol{X}_{selected} = \emptyset$;
7: **while** $|\boldsymbol{X}_{selected}| < b$ **do**
8: $\boldsymbol{P}_{sampled} = \text{Pareto}(\boldsymbol{Y}_{f,sampled})$;
9: Sort $\boldsymbol{P}_{sampled}$ by the crowding distance, from largest to smallest;
10: Add points from $\boldsymbol{P}_{sampled}$ to $\boldsymbol{X}_{selected}$, and remove the points from $\boldsymbol{X}_{sampled}$;
11: **end while**
// 3. Evaluate and update
12: Map samples $\boldsymbol{X}_{selected}$ to the original parameter space, evaluate them with the back-end flow, get results $\boldsymbol{Y}_{selected}$;
13: Update the evaluated points and the edge-length with $\boldsymbol{X}_{selected}, \boldsymbol{Y}_{selected}$;
14: **return** $\boldsymbol{TR}_i$;
15: **end function**

3.4 The Complete Flow

According to the discussion above, the proposed framework consists of two phases, the initialization phase and the optimization phase. Algorithm 3 presents the outline of our parameter tuning framework. Algorithm 1 is called in Line 2 to initialize the TRs. Lines 4 and 5 concurrently invoke Algorithm 2 to carry out the optimization phase.

4 EXPERIMENTS

4.1 Experimental Settings

'We test REMOTune with a VLSI design flow consisting of Cadence Genus and Innovus 17.1. We optimize 134 parameters demonstrated in Table 1. Every parameter corresponds to an option of a command in Genus or Innovus. For synthesis, we consider the root attributes of Genus for design partition, boundary optimization, control logic optimization, datapath optimization, multibit instances, etc. The parameters for the backend are inspired by AutoTuner [16] and PTPT [14], which have remarkable performance in academia or industry. For floorplan, we focus on the basic parameters of the floorplan command. In terms of placement, we tune the awareness of timing,

Table 1. Examples of the Flow Parameters

Stage	Total	Parameter Examples	Range
Synthesis	105	`auto partition`	false/true
		`control logic optimization`	basic/advanced/none
		`synthesis general effort`	medium/low/high/express/none
		`synthesis map effort`	high/low/medium/express/none
Floorplan	7	`aspect ratio`	0.5–2.0
		`density target`	0.5–1.0
Global placement	10	`congestion effort`	low/medium/high
		`timing effort`	medium/high
		`power effort`	none/standard/high
Detailed placement	3	`wire length optimization`	none/medium/high
		`IR drop aware`	none/low/medium/high
Routing	9	`timing driven`	false/true
		`lithography driven`	false/true
		`Si driven`	false/true
		`via optimization`	false/true

ALGORITHM 3: The proposed REMOTune framework

Input: Number of initial points N_{init}, number of trust regions T, batch size b, number of iterations $MaxIter$
Output: Pareto-optimal set $\boldsymbol{P}$

function PT((N_{init}, T, b, $MaxIter$))
  $\{\boldsymbol{TR}_1, \boldsymbol{TR}_2, \ldots, \boldsymbol{TR}_T\}$ = `Init`(N_{init}, T); ▷ Algorithm 1
  for iteration in $\{1, 2, \ldots, MaxIter\}$ **do**
   // Note that the following loop can be run in parallel;
    for $\boldsymbol{TR}_i$ in $\{\boldsymbol{TR}_1, \boldsymbol{TR}_2, \ldots, \boldsymbol{TR}_T\}$ **do**
      $\boldsymbol{TR}_i$ = `Optimize`($\boldsymbol{TR}_i$, b); ▷ Algorithm 2
    end for
  end for
  Update the Pareto-optimal set and the shared data;
end function

power, congestion, wire length, etc. As for routing, we consider the optimization on via, wire, timing, and so on.

REMOTune is implemented with Python3. The BoTorch [5] toolbox is adopted to provide acceleration for BO. We compare the proposed framework with reproduced state-of-the-art works BO [20], Recommender [18], FIST [32], PTPT [14], and AutoTuner [16]. We reproduce BO and Recommender with BoTorch and PyTorch [22] toolboxes, respectively. FIST and PTPT are implemented with scikit-learn [23]. Moreover, we use the MOTPE [21] algorithm and the toolbox Optuna [3] for AutoTuner, which shows better final scores in [16]. BO, FIST, PTPT, and AutoTuner are **sequential model-based optimization** (**SMBO**) methods. In the initialization of BO, PTPT, and AutoTuner, they randomly sample a certain number of points and used the points to train the surrogate model. To ensure the diversity of initial points, FIST requires the random samples to cover various parameter values. In the optimization phase, BO and PTPT use GP surrogate models. FIST uses XGBoost as the surrogate model while AutoTuner uses a mixture-of-Gaussian model. To score the candidates, BO uses EI as the acquisition function. PTPT maximizes the entropy while FIST randomly samples some points and randomly select a point in the Pareto-frontier. AutoTuner

uses the EI of HV as the acquisition function. For Recommender, we randomly sample and evaluate the initial points, then use them as the training set to train a neural network. After training, we randomly sample thousands of candidates, predict their QoRs with the neural network, and select the samples with the highest sums of the estimated PPAs to evaluate the QoRs.

We apply REMOTune to the parameter tuning for RISC-V processors, `RISCV32I` [25], `Ibex` [13], and `Rocket` [4]. The technology node is TSMC 65nm. The benchmarks `RISCV32I`, `Ibex`, and `Rocket` have 7.6k, 8.1k, and 14.2k cells, respectively. We set the target clock periods to be 7.6ns, 7.4ns, and 5.6ns, respectively. In our experiments, BO, FIST, PTPT, and AutoTuner have 64 samples for initialization and 256 iterations for optimization. Following the settings in the papers, these methods submit one trial at each iteration because they are designed to optimize the parameters sequentially. Recommender is not a sequential optimization algorithm. Thus, we can pre-train the model with 4,096 random samples and use 256 points in the tuning process. The training data can be evaluated in parallel. However, due to the lack of efficient sampling method, Recommender encounters many inferior points that have unsatisfactory QoRs. Only a small portion of samples can achieve acceptable results. As for REMOTune, we use 64 samples for initialization and 128 iterations for optimization. REMOTune uses less iterations to prevent it from using more total runtime than others, since the EDA tools dominate the total runtime and the non-dominated sorting mechanism spends more time on the synthesis flow. We use an embedding size of 16, a batch size of 8, and 4 TRs for `RISCV32I` and `Ibex`. For `Rocket`, we use an embedding size of 32, a batch size of 8, and 8 TRs. REMOTune uses up to 256 threads. Each instance of EDA flow uses four threads. We observe that it is harder to improve the QoRs on `Rocket` than on other benchmarks. Thus, we use a large embedding size and more TRs on `Rocket` to enable a more fine-grained and thorough search. One possible reason of the difficulty may be that `Rocket` contains much low-level verilog code generated automatically, which limits the optimization space of the synthesis flow. In addition, we use only 64 iterations on `Rocket` to reduce the runtime.

To evaluate the scalability of REMOTune, we also test the methods on `BlackParrot` [24] and `Boom` [33] processors, which have 43.2k and 3.68m cells, respectively. Since large benchmarks cost much more time to run the EDA flow, we reduce the numbers of initialization points and optimization iterations. On `BlackParrot`, we use 1/8 initialization points and optimization iterations compared to `RISCV32I`. On `Boom`, we further limit the number of iterations to be 4 for REMOTune and 8 for others.

The baseline result is obtained with the default parameter values determined by the tools. To evaluate the parameters, we compute the percentages of minimum clock period, total power, and total area of the standard cells to the baseline. Following AutoTuner, we get these results by parsing the reports from the EDA tools. The following metrics are used to compare the parameter tuning methods: HV, **maximum performance improvement** (**MPI1**), **maximum power improvement** (**MPI2**), **maximum area improvement** (**MAI**), **maximum performance-power improvement** (**MPPI**), and **maximum performance-area improvement** (**MPAI**). To compute the HV, we use [150.0, 150.0, 150.0] as the reference point. MPI1, MPI2, and MAI are the maximum improvement of minimum clock period, power, and area, respectively. MPPI is the maximum improvement of the product of the minimum clock period and the minimum power. MPAI is the maximum improvement of the product of the minimum clock period and the minimum area.

4.2 Comparisons with SOTA Methods

Figure 5 shows the comparison of REMOTune with existing methods on the `RISCV32I` benchmark. The models are initialized before Iteration #0 and the results at Iteration #0 are computed according to the initial points and the first batch of points sampled by the trained models. Although most methods have the same number of initial points, REMOTune can achieve a better exploration of the

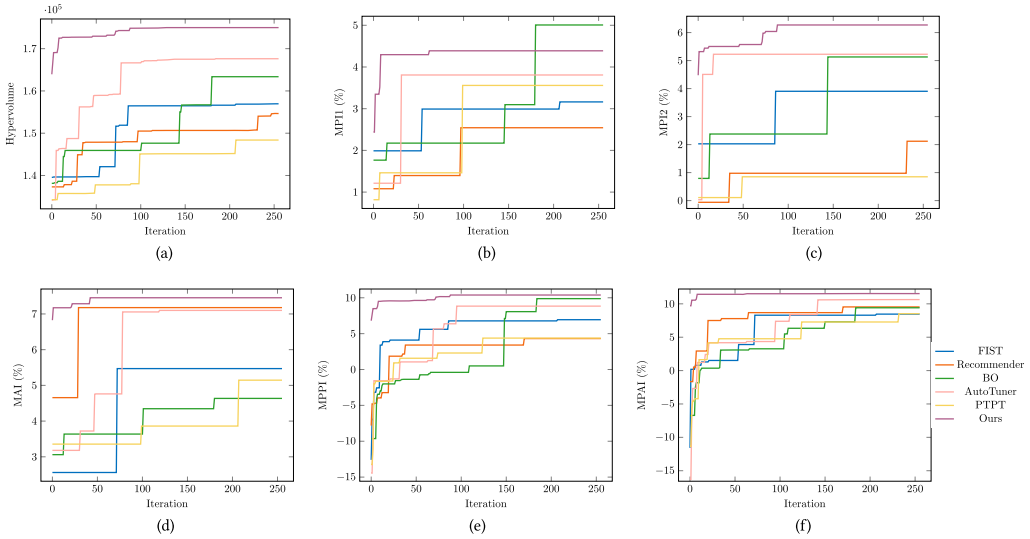

Fig. 5. Comparison of different methods on the RISCV32I benchmark. The metrics are (a) hypervolume; (b) MPI1; (c) MPI2; (d) MAI; (e)MPPI. (f)MPAI. Random embedding enables REMOTune to have a better initialization with limited initial samples. REMOTune achieves a better HV than other methods, which indicates that the proposed method can find a better Pareto-optimal set. REMOTune also has the best MPI2, MAI, MPPI, and MPAI among the methods.

search space due to the effective dimensionality reduction. It reduces the dimensionality to 16 and thus it can roughly characterize the space with 64 initial samples. Therefore, REMOTune shows better performance than others after initialization. On the contrary, a limited number of initial samples is not enough for the 134-dimensional parameter space, and most methods do not perform well at the beginning. REMOTune can enable an efficient exploration of the parameter space with faster convergence and parallelism. Since the search space is smaller and the TRs support parallel exploration, it can achieve better results within fewer iterations.

As shown in Figure 5, the proposed method achieves obviously better HV than others, which indicates that it can obtain a better Pareto-optimal set. Compared to AutoTuner, the proposed framework has a 4.4% improvement on HV, a 17.6% improvement on MPPI, and an 8.5% improvement on MPAI. BO achieves a good MPI1, but does not perform well on other metrics. Figure 6 shows the curves of HV and MPPI with respect to the EDA flow budget (parallel runs × iterations). The most significant gains in HV and MPPI come from the first few hundred samples, corresponding to the beginning iterations.

Tables 2, 3, and 4 present the comparison of parameter tuning methods on RISCV32I, Ibex, and Rocket benchmarks, respectively. $HV_{0,1}$, $HV_{0,2}$, and $HV_{1,2}$ denote the performance-power HV, performance-area HV, and power-area HV, respectively. These metrics can evaluate the ability to explore the objective subspaces. On all benchmarks, REMOTune achieves better HVs than existing methods, which indicates that the proposed method can find better Pareto-optimal sets. REMOTune acquires 4.17%, 4.94%, and 7.33% higher HVs than AutoTuner on RISCV32I, Ibex, and Rocket benchmarks, respectively. In terms of $HV_{0,1}$, $HV_{0,2}$, and $HV_{1,2}$, REMOTune also achieves the best results, which indicates that the proposed method can explore the objective subspaces better than others. Although some methods can obtain better results in individual directions, their limited ability to explore multiple QoR objectives simultaneously results in lower HVs than the proposed method.

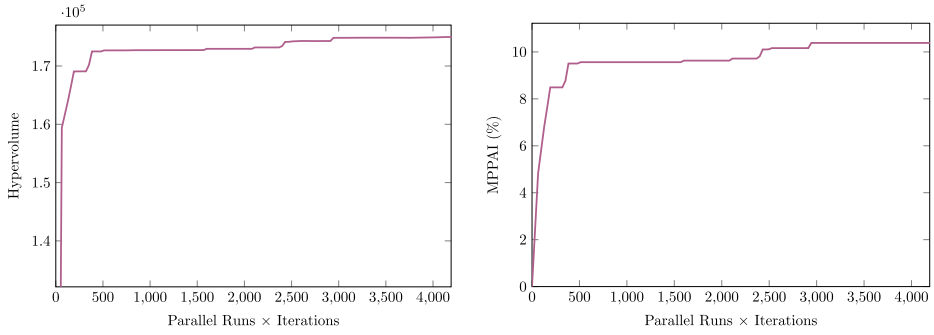

Fig. 6. The curves of (a) hypervolume (b) MPPI with respect to the EDA flow budget (parallel runs × iterations).

Table 2. Comparison of Parameter Tuning Methods on `RISCV32I` Benchmark

Method	FIST [32]	Recommender [18]	BO [20]	AutoTuner [16]	PTPT [14]	Ours
HV(10^5)	1.57	1.55	1.63	1.68	1.48	**1.75**
$HV_{0,1}(10^3)$	2.85	2.72	3.00	2.95	2.70	**3.05**
$HV_{0,2}(10^3)$	2.94	2.99	3.00	3.07	2.95	**3.12**
$HV_{1,2}(10^3)$	2.97	2.97	3.00	3.14	2.79	**3.23**
MPI1(%)	3.16	2.54	**5.00**	3.81	3.56	4.38
MPI2(%)	3.90	2.12	5.12	5.23	0.85	**6.27**
MAI(%)	5.47	7.18	4.64	7.10	5.15	**7.45**
MPPI(%)	6.94	4.51	9.88	8.83	4.37	**10.38**
MPAI(%)	8.46	9.53	9.41	10.63	8.52	**11.53**

Table 3. Comparison of Parameter Tuning Methods on `Ibex` Benchmark

Method	FIST [32]	Recommender [18]	BO [20]	AutoTuner [16]	PTPT [14]	Ours
HV(10^5)	1.54	1.42	1.52	1.62	1.53	**1.70**
$HV_{0,1}(10^3)$	3.24	3.13	3.19	3.44	3.23	**3.54**
$HV_{0,2}(10^3)$	2.86	2.64	2.83	2.92	2.81	**3.07**
$HV_{1,2}(10^3)$	2.61	2.47	2.59	2.65	2.59	**2.68**
MPI1(%)	10.26	8.30	9.47	11.89	10.29	**14.03**
MPI2(%)	4.19	4.20	3.84	**5.74**	4.49	5.35
MAI(%)	−1.83	−4.36	−1.74	−2.42	−2.33	**−1.62**
MPPI(%)	14.02	12.15	12.84	16.95	14.32	**18.63**
MPAI(%)	8.62	4.30	7.89	9.76	8.20	**12.84**

Figure 7 shows the results of the methods on `BlackParrot` and `Boom`, where REMOTune achieves the highest HVs on both large benchmarks. Due to the limited number of iterations, most methods do not perform well on these large testcases. On `BlackParrot`, REMOTune, and Recommender have outstanding results. A common feature of these two methods is parallelism, which can explore more points than others within a few iterations and contribute significantly to the performance. However, since the number of samples is further reduced on `Boom`, Recommender cannot maintain the high performance. With the dimension reduction mechanism, REMOTune can

Table 4. Comparison of Parameter Tuning Methods on Rocket Benchmark

Method	FIST [32]	Recommender [18]	BO [20]	AutoTuner [16]	PTPT [14]	Ours
HV(10^5)	1.47	1.19	1.35	1.50	1.31	**1.61**
$HV_{0,1}$(10^3)	3.03	2.79	2.93	3.16	2.85	**3.35**
$HV_{0,2}$(10^3)	3.02	2.75	2.94	3.16	2.84	**3.18**
$HV_{1,2}$(10^3)	2.42	1.85	2.19	2.23	2.20	**2.51**
MPI1(%)	12.38	14.50	13.44	**16.72**	11.97	16.11
MPI2(%)	−0.51	−6.70	−2.99	−2.42	−2.83	**1.57**
MAI(%)	**−1.01**	−7.25	−3.32	−2.55	−3.39	−1.31
MPPI(%)	11.93	8.77	10.85	14.70	9.48	**17.43**
MPAI(%)	11.50	8.30	10.67	14.60	8.99	**15.01**

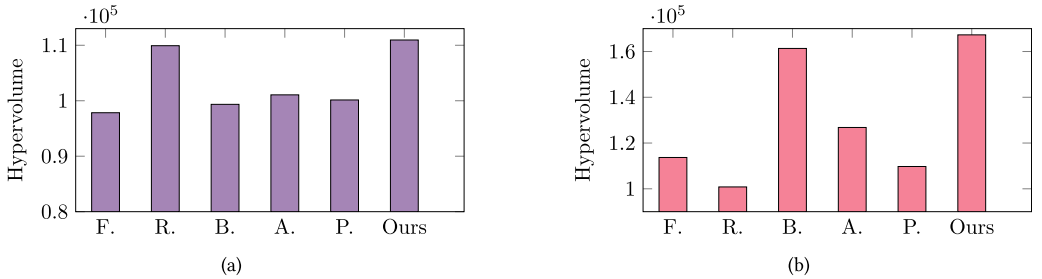

Fig. 7. Comparison of the methods on (a) BlackParrot; (b) Boom. We use the abbreviations F., R., B., A., and P. to represent FIST, Recommender, BO, AutoTuner, and PTPT, respectively. Due to the limited number of iterations, most methods do not perform well on these large testcases.

better model the search space with a very limited number of samples. Thus, it can keep a high performance on Boom. The experiments on large benchmarks show the effectiveness of REMOTune and highlight the importance of dimensionality reduction and parallelism.

4.3 Parameter Importance Analysis

To understand the source of the QoR improvements from REMOTune, we compare the actual parameter values found by different methods. Due to a large number of parameters, we need to focus on a few important parameters. We estimate the importance of the parameters with **automatic relevance determination** (**ARD**) [31]. The evaluated parameters of REMOTune and the QoR results are used as the inputs and outputs of ARD, respectively. Figure 8(a) presents the standard deviations of the parameters, which are the average values from 100 ARD models. A larger deviation indicates higher importance. From the ARD analysis, we can observe that the timing effort, synthesis generic effort, and synthesis mapping effort are critical parameters for the RISCV32I testcase. It is consistent with our intuition. We can also find that the default value for x state has a significant impact on the QoRs, which is not an obvious conclusion we can gain from intuition. In addition, the congestion effort and wire length optimization may not have large influences on the results. Figuring out the important parameters helps us analyze the parameter values and gain experience in parameter tuning.

Figure 8(b) shows the Pareto-optimal results of the tested methods on RISCV32I. REMOTune gets multiple outstanding results, located in the lower left corner. Comparing the outstanding QoRs of REMOTune with the others on the important parameters, we can understand the improvements from REMOTune. Table 5 shows the parameter values of a few representative points

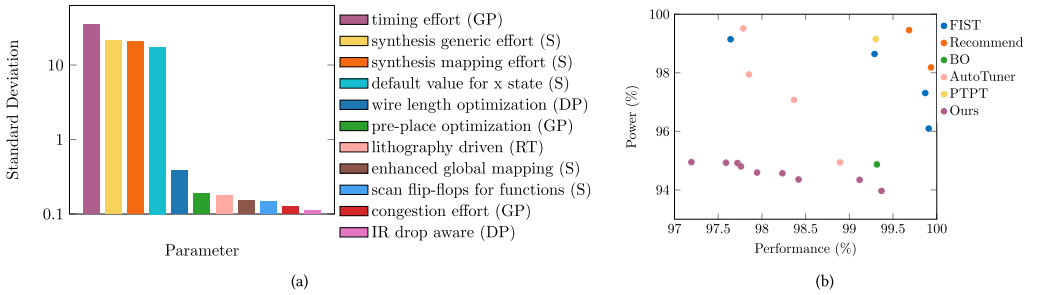

Fig. 8. Analysis of the results on RISCV32I. (a) Standard deviations of some parameters, obtained via automatic relevance determination. We use abbreviations S, GP, DP, and RT for synthesis, global placement, detailed placement, and routing, respectively. (b) Ratios of the Pareto-optimal results of the tested methods to the baseline on performance and power. REMOTune gets multiple outstanding results, located in the lower left corner.

Table 5. Comparison of Parameter Values Found by Different Methods on Rocket Benchmark

Point ID	PI1 (%)	PI2 (%)	AI (%)	timing effort	generic effort	mapping effort	default x	wire length opt.
REMOTune1	**2.8**	5.1	2.1	medium	high	medium	1	medium
REMOTune2*	0.7	**6.1**	2.2	medium	high	medium	1	medium
REMOTune3	0.2	4.7	**6.2**	medium	high	express	0	medium
PTPT	1.4	−1.5	2.1	high	medium	high	1	medium
AutoTuner1	2.2	0.5	4.9	high	high	high	0	medium
AutoTuner2	0.6	2.4	3.2	high	high	high	1	medium
BO	1.0	−2.7	−0.3	medium	medium	medium	1	medium
Recommender	0.1	1.9	1.5	medium	high	express	1	medium
FIST	0.8	1.4	0.6	medium	high	express	0	high

*: Apart from the mentioned parameters, REMOTune1 and REMOTune2 have many different parameter values. For example, REMOTune2 uses pre-place optimization while REMOTune1 does not. Thus, they have different QoRs.

found by different methods on RISCV32I, where PI1, PI2, and AI refer to performance improvement, power improvement, and area improvement, respectively. The parameters 'timing effort', 'generic effort', 'mapping effort', 'default x', and 'wire length opt' correspond to the five most important parameters shown in Figure 8(a). Note that the 'express' option provides a fast but not thorough optimization. According to Table 5, AutoTuner prefers to use high timing, generic, and mapping efforts, which can produce satisfactory results at the expense of runtime. As a result, AutoTuner can surpass PTPT, BO, and Recommender. REMOTune usually uses a high generic effort with lower timing and mapping efforts, which can also lead to outstanding results. The massive parallelism helps REMOTune to explore more possibilities under this setting and utilize the less important parameters, such as the 'enhanced global mapping', to make further improvement.

Table 6 shows the top-5 important parameters on RISCV32I, Ibex, and Rocket benchmarks. We use abbreviations S, F, GP, DP, and RT for synthesis, floorplan, global placement, detailed placement, and routing, respectively. According to the table, the generic effort in synthesis is a common important parameter among the three benchmarks. This finding highlights the importance of synthesis generic effort in the EDA flow. Density in floorplan is an important parameter in larger designs Ibex and Rocket, as shown in Table 6. On RISCV32I and Ibex, we find some important parameters in the placement and routing processes, such as the timing effort in global placement, the wire length optimization in detailed placement, and the timing-driven level in

Table 6. The Five Most Important Parameters on RISCV32I, Ibex, and Rocket Benchmarks

Benchmark	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Parameter 5
RISCV32I	timing effort (GP)	generic effort (S)	mapping effort (S)	default x (S)	wire length opt (DP)
Ibex	density (F)	generic effort (S)	force constant removal (S)	aspect (F)	timing driven (RT)
Rocket	density (F)	generic effort (S)	opt. constant 1 flops (S)	preserve sync. logic (S)	opt. constant hpins (S)

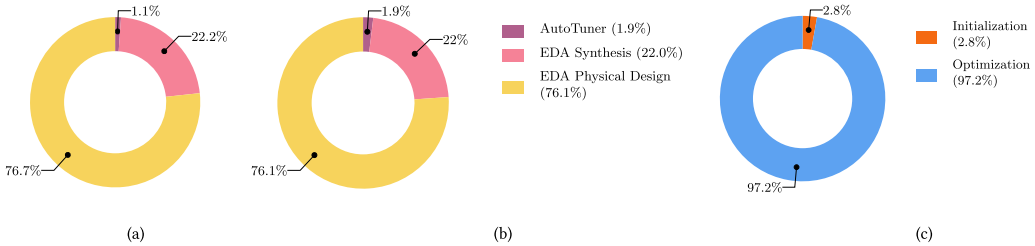

Fig. 9. Runtime breakdowns of (a) REMOTune, (b) AutoTuner on RISCV32I benchmark in the optimization phase. (c) shows the ratios of runtime spent on the REMOTune's initialization and optimization phases. The total runtime is composed of the optimization, synthesis flow, and physical design flow runtime. For the DSE methods, the bulk of the runtime is spent on the EDA flow. It takes less than 2% runtime of REMOTune and AutoTuner to execute the optimization. The initialization phase of REMOTune takes less than 3% of the runtime.

routing. These parameters can significantly affect the physical design flow and thus contribute a lot to the QoRs on these two benchmarks. On Rocket, the majority of important parameters are from the synthesis flow. Since the Verilog code of Rocket is generated automatically from the hardware construction language at a higher level, there may be some redundancy in the code, which needs to be optimized with synthesis options such as allowing constant 1 propagation through flip-flops (opt. constant 1 flops), merging the synchronous control logic near the flip-flops (preserve sync. logic), and allowing constant propagation through this hierarchical boundary pins (opt. constant hpins).

4.4 Runtime Analysis

Figure 9(a) and Figure 9(b) present the runtime breakdowns of REMOTune and AutoTuner on RISCV32I benchmark. The total runtime includes the optimization, synthesis flow, and physical design flow runtime. According to the figures, most methods spend the bulk of their runtime on the EDA flow. As an example, it takes less than 2% runtime of REMOTune to execute the optimization process. Figure 9(c) shows the ratios of runtime spent on the REMOTune's initialization and optimization phases. The initialization phase takes only 2.8% of the runtime. Most of the runtime is consumed by the optimization phase.

Figure 10 shows the runtime comparison of the optimization methods. Recommender takes the shortest runtime due to the simple sampling mechanism, parallel evaluation, and the fast training of the neural network model. Due to the simple surrogate models and acquisition functions, FIST and BO are also fast within a limited number of iterations. Although REMOTune is not the fastest method, it is able to explore significantly more points and achieve much better performance.

4.5 Ablation Studies

REMOTune achieves better performance than others with the help of random embedding and multi-objective trust-region BO. Random embedding limits the size of the search space, which

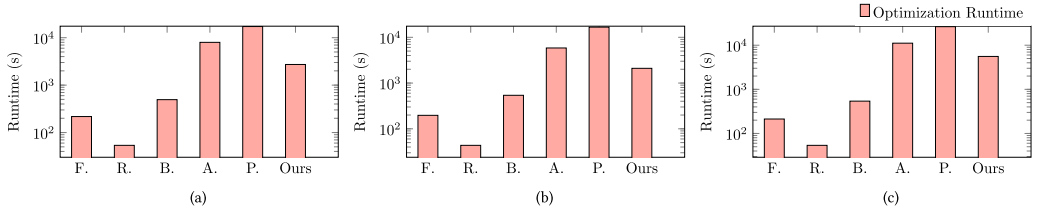

Fig. 10. Runtimes of the tested algorithms on (a) RISCV32I, (b) Ibex, and (c) ROCKET benchmarks. We use the abbreviations F., R., B., A., and P. to represent FIST, Recommender, BO, AutoTuner, and PTPT, respectively. Due to the simple models, FIST, Recommender, and BO achieve small optimization runtimes. REMOTune has a moderate runtime, which is still shorter compared to the EDA flow runtime.

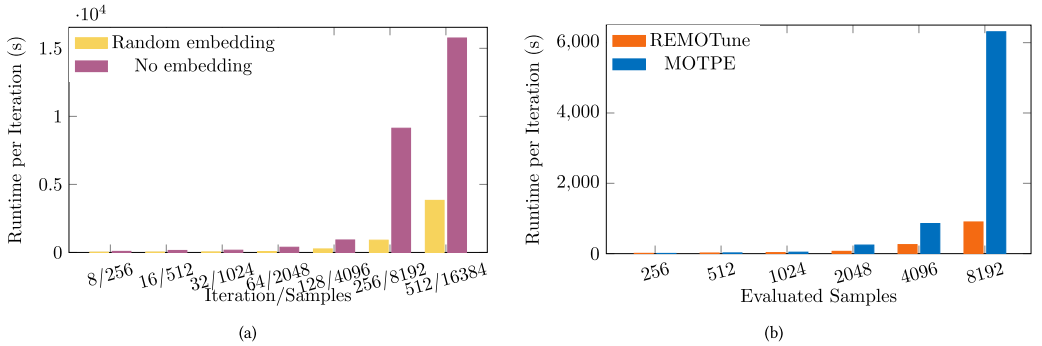

Fig. 11. Ablation study results. (a) compares REMOTune with and without random embedding. REMOTune with random embedding has significantly less runtimes than REMOTune without dimensionality reduction. (b) compares REMOTune and MOTPE with 16-dimensional inputs given the same number of evaluated samples. After dimensionality reduction, MOTPE is still slower than REMOTune, which shows the high efficiency of trust-region BO.

effectively reduces the runtime of the optimization algorithm. With a limited number of iterations, it is easier to make a thorough exploration in a smaller space. Multi-objective trust-region BO enables parallel optimization, which can explore multiple points at each iteration. By optimizing in decoupled TRs, the runtime of multi-objective trust-region BO can be limited to an acceptable range.

Figure 11(a) compares the runtimes of an optimization iteration with and without random embedding. According to the results, random embedding can significantly save the runtime of the optimization algorithm. Moreover, due to the excellent parallelism of trust-region BO, we can explore many points in limited iterations of optimization. Figure 12 presents the performance comparison. With a limited number of data points, random embedding helps REMOTune to make a thorough exploration, achieving significantly better performance than the algorithm without random embedding.

In our experiments, REMOTune can explore 4,096 points in 128 iterations. The runtime of a sequential optimization algorithm can become unaffordable for so many data. Although random embedding is applicable to other multi-objective BO methods, their runtimes may be unsatisfactory without trust-region BO. In Figure 12(b), we compare the runtimes per iteration of REMOTune and MOTPE. In this experiment, both methods have 16-dimensional inputs. REMOTune exhibits lower runtimes than MOTPE, which shows the high efficiency of trust-region BO.

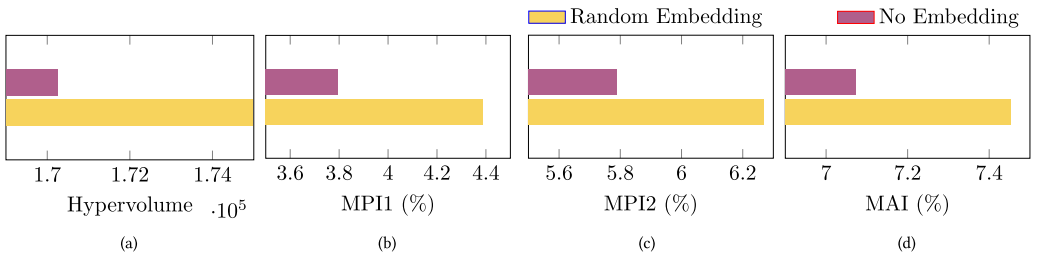

Fig. 12. Ablation study that compares REMOTune with and without random embedding on (a) Hypervolume; (b) MPI1; (c) MPI2; (d) MAI. With random embedding, REMOTune can make a more efficient exploration in the search space, resulting in significantly better performance.

5 CONCLUSION

To enable parameter tuning in an enormous space, we have proposed REMOTune, a parameter tuning framework for VLSI design flows, which reduces the number of variables for the optimization algorithm via random embedding and explores Pareto-optimal tool parameter configurations in parallel by multi-objective trust-region BO. In our experiments, the proposed framework achieves better performance than existing methods. The extraordinary parallelism and efficient optimization of REMOTune enable the exploration of more points in limited runtime, resulting in better performance than existing methods. It also has other possible advantages such as the robustness against the non-determinism in EDA tools. REMOTune requires less iterations than existing methods to get a satisfactory result because it not only has better initialization under the same number of initial samples but also achieves a faster convergence. Thus, it is also suitable for larger designs.

In the article, we have highlighted the importance of dimensionality reduction and parallelism in VLSI design flow parameter tuning. Moreover, noticing the domain-specific feature that the front-end flow usually costs less time than the back-end flow, we have designed a pruning rule to take advantage of this property. In future works, we expect to explore the methods to map the parameters to a space with low dimensionality. Techniques like **singular value decomposition (SVD)** and sparse coding can be adopted. Besides, a better BO algorithm that incorporates domain-specific knowledge can be utilized to further improve the efficiency of parameter tuning. To enhance the scalability of parameter tuning algorithms, we can embrace asynchronous parallel optimization to enable large-scale optimization on distributed computing systems. The optimization about technology nodes is done by the EDA tools and exposed to our algorithm as tool parameters. Since our method is process-agnostic, it can be appied to more advanced processes by adding the process-specific parameters.

The code of REMOTune is available at https://github.com/shelljane/REMOTune. We plan to support the open-source platform OpenROAD [2] soon.

REFERENCES

[1] Anthony Agnesina, Kyungwook Chang, and Sung Kyu Lim. 2020. VLSI placement parameter optimization using deep reinforcement learning. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Article 144, 9 pages.

[2] T. Ajayi, D. Blaauw, T. B. Chan, C. K. Cheng, V. A. Chhabria, D. K. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, et al. 2019. OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain. *Proceedings of Government Microcircuit Applications and Critical Technology Conference*. 1105–1110.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*.

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[5] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. In *Advances in Neural Information Processing Systems*.

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Annual Conference on Neural Information Processing Systems (NIPS)*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.), Vol. 24.

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.

[8] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. 2020. Differentiable expected hypervolume improvement for parallel multi-objective Bayesian optimization. In *34th International Conference on Neural Information Processing Systems*. (2020).

[9] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. 2021. Parallel Bayesian optimization of multiple noisy objectives with expected hypervolume improvement. *Advances in Neural Information Processing Systems*.

[10] Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. 2022. Multi-objective Bayesian optimization over high-dimensional search spaces. *Uncertainty in Artificial Intelligence*, 507–517.

[11] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[12] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D. Turner, and Matthias Poloczek. 2019. Scalable global optimization via local Bayesian optimization. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, Vol. 32.

[13] Markus Wegmann et al. 2017. Ibex RISC-V Core. (2017). Retrieved July 25, 2022 from https://github.com/lowRISC/ibex.

[14] Hao Geng, Tinghuan Chen, Yuzhe Ma, Binwu Zhu, and Bei Yu. 2022. PTPT: Physical design tool parameter tuning via multi-objective Bayesian optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 42, 1 (2022), 178–189.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30.

[16] Jinwook Jung, Andrew B. Kahng, Seungwon Kim, and Ravi Varadarajan. 2021. METRICS2.1 and flow tuning in the IEEE CEDA robust design flow and OpenROAD. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[17] Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabas Poczos. 2018. Parallelised Bayesian optimisation via Thompson sampling. In *International Conference on Artificial Intelligence and Statistics*, Amos Storkey and Fernando Perez-Cruz (Eds.), Vol. 84. 133–142.

[18] Jihye Kwon, Matthew M. Ziegler, and Luca P. Carloni. 2019. A learning-based recommender system for autotuning design flows of industrial high-performance processors. In *ACM/IEEE Design Automation Conference (DAC)*.

[19] Rongjian Liang, Jinwook Jung, Hua Xiang, Lakshmi Reddy, Alexey Lvov, Jiang Hu, and Gi-Joon Nam. 2021. FlowTuner: A multi-stage EDA flow tuner exploiting parameter knowledge transfer. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[20] Yuzhe Ma, Ziyang Yu, and Bei Yu. 2019. CAD tool design space exploration via Bayesian optimization. In *ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*.

[21] Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, Masahiro Nomura, and Masaki Onishi. 2022. Multiobjective tree-structured Parzen estimator. *Journal of Artificial Intelligence Research* 73, 1 (2022), 1209–1250.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Annual Conference on Neural Information Processing Systems (NIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). 8024–8035.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)* 12, 1 (2011), 2825–2830.

[24] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. 2020. BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. DOI: https://doi.org/10.1109/MM.2020.2996145

[25] James E. Stine, Ryan Ridley, and Teodor-Dumitru Ene. 2021. OSU Datapath/Control RV32 Single-Cycle and Pipelined Architecture in SV. (2021). Retrieved July 25, 2022 from https://github.com/stineje/osu-riscv.

[26] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. 2019. LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 74–77.

[27] Sergei Vassilvitskii and David Arthur. 2006. k-means++: The advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*. 1027–1035.

[28] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando De Feitas. 2016. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research* 55, 1 (2016), 361–387.

[29] Zi Wang and Stefanie Jegelka. 2017. Max-value entropy search for efficient Bayesian optimization. In *International Conference on Machine Learning*, Vol. 70. 3627–3635.

[30] Kilian Weinberger. 2018. Gaussian Processes. (2018). Retrieved July 25, 2022 from https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote15.html.

[31] David Wipf and Srikantan Nagarajan. 2007. A new view of automatic relevance determination. *Advances in Neural Information Processing Systems* 20 (2007).

[32] Zhiyao Xie, Guan-Qi Fang, Yu-Hung Huang, Haoxing Ren, Yanqing Zhang, Brucek Khailany, Shao-Yun Fang, Jiang Hu, Yiran Chen, and Erick Carvajal Barboza. 2020. FIST: A feature-importance sampling and tree-based method for automatic design flow parameter tuning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*.

[33] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd generation Berkeley out-of-order machine. (May 2020).

[34] Matthew M. Ziegler, Hung-Yi Liu, and Luca P. Carloni. 2016. Scalable auto-tuning of synthesis parameters for optimizing high-performance processors. In *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*. 180–185.

[35] Matthew M. Ziegler, Hung-Yi Liu, George Gristede, Bruce Owens, Ricardo Nigaglioni, and Luca P. Carloni. 2016. A synthesis-parameter tuning system for autonomous design-space exploration. In *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*. 1148–1151.

Received 11 August 2022; revised 24 February 2023; accepted 25 March 2023