Exploring Compiler Optimization: A Survey of ML, DL and RL Techniques

C Mithul, D.Mohammad Abdulla, M Hari Virinchi, M Sathvik, Meena Belwal,
Department of Computer Science and Engineering, Amrita School of Computing,
Bengaluru, Amrita Vishwa Vidyapeetham, India.
mithulc15@gmail.com, mohammadabdulla20march@gmail.com,
virinchi90521@gmail.com, maddisathvik9@gmail.com, b_meena@blr.amrita.edu.

***Abstract*— The past few years, traditional compiler optimization methods have been found to be further enhanced by machine learning (ML), deep learning (DL) and reinforcement learning (RL). These differ from classical techniques that often use rule of thumb based decision making. Rather, ML/DL/RL based approaches provide a means for learning from data thus improving performance in different dimensions such as code generation, resource allocation and runtime. In this paper we give an overview of current research and methodologies utilizing ML, DL and RL for compiler optimization purposes. We analyze the major models in terms of their employed learning strategies and desired optimizations within a compiler framework. Moreover, we highlight some of the difficulties faced when these compilers are embedded with these learning models such as adaptability, generalization and overhead trade-offs. Additionally, our survey presents case studies demonstrating Quantitative improvements on well-known benchmarks mainly focusing on models' adaptability to different architectures and their role in supporting the decision-making process of compilers. We conclude outlining open research questions as well as possible future directions for further investigations into this emerging interdisciplinary field.**

***Keywords*— Compiler optimization, machine learning, Bayesian optimization, performance tuning, code generation, auto-tuning, neural networks, decision trees.**

I. INTRODUCTION

The compiler optimization phase is a critical part of software development lifecycle, greatly impacting on performance and the use of resources as well as scalability of programs. Usually compilers apply rule-based methods and heuristics for optimization which often produce poor quality codes that necessitate extensive manual intervention [1-2]. Moreover, machine learning (ML), deep learning (DL) and reinforcement learning (RL) techniques have transformed the field of compiler optimization through enabling systems to be data driven with ability to adapt to different hardware architectures and software specifications in an automatic manner [3-5].

For big data analysis, ML, DL, and RL are currently major foci in academia and industry because they help identify complex patterns that ultimately improve decision accuracy during all stages of optimization process [6-8]. The most relevant optimization techniques such as algorithms used for specific hardware architectures can be identified by ML tools using comprehensive code data, execution traces and profiling sources for certain workloads or performance goals [9-11].

ML, DL, and RL in compiler optimization are surveyed so as to have a wide understanding of their application. Some optimization tasks include code generation [12], resource allocation [13-14], loop transformations [15] and runtime adaptation [16-17]. In addition, the analysis shows the pros and cons of different learning models with regard to optimization factors such as memory usage, run-time, power consumption and code size [18-21].

Furthermore, we discuss some of the issues/challenges faced while integrating ML models into compilers like; adaptability to different software/hardware environments, generality across varied workloads , model inference and training overheads [22-24]. We also present several case studies that report on benchmark improvements quantified over ML-based compilers that especially focus on cross-architecture optimization and decision-making processes supported by machine learning.[25-28]

Also, the survey delves into how supervised learning assist large datasets in optimization problem solving and also how unsupervised learning and reinforcement learning improve optimization strategies [29]. These developments are enhanced by deep learning (DL), which involves the construction of deep neural networks using such techniques as network quantization [6]. DL works, making use of intricate data patterns, can resolve complicated optimization problems through architectures like CNN, RNN, GNN [7], [9], [17].

In addition to this RL has shown promise in auto-tuning compiler pass orders through methods such as coresets and normalized value prediction improving optimization strategies [13],[15-16]. For example, Q-learning, policy gradient methods as well as deep reinforcement learning make some of these approaches that are employed while employing RL-based compiler optimizations [14-16].

Additionally in the past, it was found that modeling CPU performance can be conducted by examining primitive operations [30] as well as the implementation of specific algorithms such as turbo decoders on coarse-grained reconfigurable architectures [31]. These developments emphasize the practical advantages of incorporating ML and RL into compiler frameworks, resulting in real-world gains in performance and efficiency.

This survey is a crucial reference for compiler developers, researchers and optimization practitioners who want to exploit ML techniques for state-of-the-art optimizations as ML continues to grow interdisciplinary field. We also point out open research questions and future directions for further study towards the integration of compiler and ML domains [13], [19], [21], [28].

Section II of this works discusses the objectives. Section III classifies the approaches into ML, DL, RL and Hybrid techniques. Section IV discusses the various compiler techniques and comparison among them is presented in Section V. Finally, Section VI provides a detailed conclusion of the work.

II. OBJECTIVES

The objective of this survey is to compare the various techniques of compiler optimization those apply ML, DL or RL techniques. The survey performed in this work compares the various techniques based on the objectives a particular technique has been developed. The objectives undertaken by the techniques under study are:

1. Improve Compiler Optimization Efficiency: Compare ML/DL/RL technologies with the ways they can outperform the compiler heuristics with compiler pass orders, resource allocation and runtime adaptation [1], [6], [16].

2.Enhance Compiler Optimization Performance: Combine ML/DL/RL techniques with compiler optimizations problems, that are related to neural networks compilation and outperformance of the conventional compiler optimization methods [4], [5], [9], [12].

3. Adapt to Changing Hardware and Software Environments: Create an FPGA/CPLD-based system that can be reconfigured for changing hardware and software environments such as new processor architectures or new programming languages [2], [10], [14].

4. Ensure Explainability and Transparency: Develop the models that are explainable and transparent, therefore for the developers the decisions taken by the compiler are understandable [8], [11], [18].

5. Reduce Compilation Time and Improve Code Quality: Provide instructions for optimizing the compile speed of ML/DL/RL methods and also ways to enhance code quality having a high-speed application development [3], [7], [13], [15].

III. CLASSIFICATION

We classified the various compiler optimization techniques into four categories as shown in Figure 1.

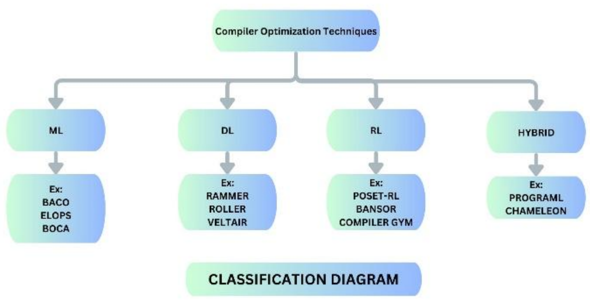

Fig.1.Classification of Compiler Optimization Techniques

IV. RELATED WORKS

In this section the various techniques are discussed as per the classification provided in the previous section.

A. Machine Learning

The use of ML optimization techniques has shown appreciable enhancement in numerous applications. One of the strategies used to undertake this research involves utilizing compiler optimization for code clone detection through the application of the code2vec model in analyzing vis decompiling of binaries with varying optimization levels, including O1, O2, as well as O3 levels. This method achieves 84. 61% accuracy and 84. It was 96% F1-score at the O2 level while degrades when applying to other optimization levels [1].

Another Compiler Optimization Parameter Selection Method (ELOPS) predicts optimal compiler parameters with the aid of ensemble learning. There is exceptional performance shown by ELOPS that presents the prediction accuracy of 0. 71 and 0. 72 and several performance improvements (1. 29x and 1. 26x), although often with high training time and intricate feature engineering [2]. Likewise, the performance Bayesian Optimization framework BaCO specializes in high-level autotuning and design space search, operating billions of configurations and showing the characteristics of previous autotuners [3].

BOCA (Bayesian Optimization Compiler Autotuning) applies a Random Forest model and a novel selection strategy to deliver better runtime speedups on GCC and LLVM compilers with cBench and PolyBench benchmarks. Nevertheless, the generalization of the results on more recent compilers and different types of programs is still required [4].

Such methods emphasize a considerable growth of ML optimization and show the potential research directions with improvements in performance.

B. Deep learning

Deep learning techniques have been tried out in various compiler optimization tasks. A convolutional neural network (CNN) and a Long Short-Term Memory (LSTM) model to detect characteristics in binary files and estimate the compiler and optimization level from the byte sequences is introduced in [5]. Code representation using intermediate representations like ASTs and CDFGs and also employing Graph Neural Networks (GNNs) to learn representations for problems like predicting optimal CPU/GPU mapping is shown in [7]. NNSmith, which generates diverse and valid test cases for deep learning compilers by ensuring topological and attribute diversity in computation graphs, as well as improving numeric validity through gradient guidance is proposed in [9]. The cost model is trained on runtime measurement data, and the explorer uses the predictions to select promising configurations for real measurements [10].

Other deep learning approaches proposed for specific optimization tasks are RAMMER, which abstracts hardware accelerators as virtualized parallel devices and implements rOperators for DNN computation, compiling models into a

DFG of rOperators [6]. ROLLER, which identifies aligned tile shapes (rTiles) and constructs an efficient tile processing pipeline to improve end-to-end throughput is discussed in [11]. SparseTIR, which defines composable abstractions and schedules for sparse compilation, evaluated on real-world GNN datasets [24]. Another describes VELTAIR, an adaptive compilation and scheduling system for multi-tenant deep learning services, using a single-pass multi-version search algorithm and dynamic scheduling based on interference levels [8].

C. Reinforcement Learning

RL techniques have produced significant gains in enhancing compiler efficiency. CompilerGym [12] is a flexible tool for compiler optimization tasks that can be easily extended with common RL baselines. It has been applied to various RL methods such PPO, Greedy Search, and Random Search using problems such as LLVM phase ordering and CUDA loop nest generation.

Autophase V2 [13] looks at function-level phase ordering with random search optimizing the initial model, as well as deep RL with PPO which at maximum improves code size by 9%. SuperSonic applied multi-armed bandits and deep RL to automatically tune its hyper-parameters for the purposes of code optimisation and surpassed existing tools such as Stoke and OpenTuner. AutoPhase uses deep RL to select ordered compiler phases in the process of HLS utilizing corresponding programs that are 16% faster in comparison with -O3.

Several research studies have incorporated RL in differently compiled optimization tasks. Haj-Ali et al. [15] implemented RL for vectorization for Matrix Vector, Autophase got 28% of speed up than -O3 on HLS, CORL achieved 1. 93% devise faster on unseen programs, and CompilerGym demonstrated optimistic outcomes over -O3 in code quantity. Shahzad et al. [16] introduced new RL strategies and measures for implementation with the most efficient of them performing up to 0023x quicker training and assuring up to 4 times higher performance. POSET-RL [17] leveraged DQN to identify the sequence that would yield the best optimization passes which resulted in up to 22% better performance compared to the standard sequences. Thus, the size of the flow graph has been reduced by 94%, and its runtime has been decreased by 46%. Based on the bandit-based RL, Bansor [18] improved tensor program auto-scheduling, and even surpassed Ansor with notable resultant improvement.

D. Hybrid Techniques

Hybrid Techniques are mix of any two or more of the ML, DL & RL techniques. A hybrid technique of ML & DL is implemented in three of the works compared in the Comparision table [19-21]. Another kind of hybrid techniques are used in work by VenkataKeerthy [22]. An other named CHAMELEON introduces a hybrid method using DL and RL it leverages the strengths of deep learning to extract high-level features and reinforcement learning to make decisions based on those features, optimizing tasks in a dynamic and efficient manner [23]. Another kind of hybrid approach is shown by de Souza Xavier [25]. Hybrid approaches are used in many research works where it is shown that they perform better than normal approaches.

V. COMPARISION OF COMPILER OPTIMIZATION TECHNIQUES

This section shows and compares the compiler optimization techniques discussed in related works section. The comparison is summarized in two tables namely table 1 and table 2 the fields which are as follows:

A. Approach: This column shows the general strategy or the analytic method used in each paper such as machine learning, Deep learning. reinforcement learning or a hybrid model.

B. Algorithm/Technique Description: This column provides a description of the specific algorithms or techniques used in the respective papers, such as PPO, A2C, GNNs, NNs.

C. Training Procedure: The information about the dataset split, episode length, optimization algorithm use for training, and other training specifics are provided in this column for each paper.

D. Validation Method: This column describes how those papers are validated including cross-validation, testing of the solution on held-out datasets, and comparison with baselines or state-of-the-art methods, and other validation methods that were considered in order to measure how effective the put forth solutions were.

TABLE 1. COMPARISON OF COMPILER OPTIMIZATION TECHNIQUE'S PART 1

Technique	Approach	Algorithm/Technique Description	Training Procedure	Validation Method
[1]	ML	Compile source with varied GCC optimizations, classify with DNN	Compile with different optimizations, train DNN for classification	Assess DNN accuracy across optimizations, conduct cross-validation.
ELOPS [2]	ML	Statistical model within the compiler framework for parameter prediction.	Data collection, model construction with multi-objective PSO algorithm, and feature extraction.	Model validated through dataset splitting. Performance compared with SVM and KNN.
BaCO [3]	ML	Gaussian Processes, Modified Expected Improvement, Chain-of-Trees for	Recommendation loop, random init, refined by Bayesian Optimization	Evaluated on 15 kernels from ML, signal processing, linear algebra

		constraints		
BOCA [4]	ML	Iterative RF model, selection balancing exploitation and exploration	Iteratively selects and evaluates sequences based on RF model.	Empirical evaluations on GCC, LLVM, cBench, PolyBench benchmarks
[5]	DL	LSTM & CNN	Split – 50, 25, 25, 40 epochs with batch size 256	Tested on Test set
RAMMER [6]	DL	Operators & Tasks for DNN computation	converts DNN into DFG of Operators, optimizes	Heuristics to find the granularity of Tasks.
[7]	DL	GNNs to learn code from compiler IR like ASTs and CDFGs for prediction tasks	Split into k folds, Train GNN on k-1, validate last fold, repeat for k folds	k-fold cross-validation, training / testing on disjoint benchmark suites.
VELTAIR [8]	DL	Adaptive Compilation, Scheduling	Not Discussed	ResNet-50, Performance Comparison
[9]	DL	Diverse Graphs, Binning-based Attributes	Synthesizing Models, Random Inputs	Differential Testing, Fuzzing
[10]	DL	Schedule Optimizer, Cost Model	Runtime Data, Experimental Trials	ML Predictions, Real Measurements
ROLLER [11]	DL	Tensor Shapes, Tiles, Recursive Algorithm	Not Applicable	Micro-Performance Model, Hardware Abstraction
CompilerGym [12]	RL	A2C, APEX, IMPALA PPO	Trained on Csmith to generate both training and validation sets	Serialized state validation, post-processing scripts
Autophase V2 [13]	RL	- Random search: Evaluate random sequences - Deep RL:Use PPO reward	50K/100K episodes with episode length 45 using PPO	RL. Compared with -Oz flag.
[14]	RL	Meta-optimizer, A3C	Parallel Population-Based Training	Cross-Validation on New Benchmarks
AutoPhase [15]	RL	Policy Gradient (PG), (DQN) algorithms	56 static features extracted from LLVM IR of benchmark	12 HLS benchmarks from CHStone and LegUp
[16]	RL	Base: PPO, Pass ordering, Action tuples, Episode sizing, -O3 backend	PPO, 300 iterations, episode length 45	Test on held-out functions, compare with -O3
POSET-RL [17]	RL	DQN predicts optimal sequences, IR2Vec and ODG dependencies between passes.	DQN with llvm-test-suite benchmark files to approximate optimal Q-function state-action pairs	MiBench, SPEC CPU 2006 and 2017 benchmarks.
Bansor [18]	RL	UCB for selecting sketches (loop structures) and tasks during tensor	Online training of cost model during search process	Measure on actual hardware
PROGRAML [19]	Hybrid ML & DL	Graph Representation, MPNNs	Construct Graphs, Iterative Message Passing	Holdout Sets, K-fold Cross-validation
[20]	Hybrid ML & DL	CNN for feature extraction from LLVM-IR code,	CNN trained for 40 epochs, ML algorithms trained using CNN's output	10-fold cross-validation, 5x avg accuracy, leave-one-out cross-validation
[21]	Hybrid ML and NN	NN predicts & skips passes, clustering & models tailor pass pipelines	Tracks LLVM IR feature changes, trains NN & predictive models	Compares predicted passes with actual, ensuring faster compilation.
[22]	Hybrid ML & RL	"ML-Compiler-Bridge" inter-process (gRPC, named pipes) and in-process (ONNX, TensorFlow AOT) runners.	Uses inter-process runners for training, supports multi-worker training.	Evaluated four ML-enabled LLVM optimizations: compile time, training time, round-trip time.
CHAMELEON [23]	Hybrid DL &RL	Adaptive Exploration, Adaptive Sampling	PPO, Actor-Critic	Real Hardware, GPU Titan Xp

E. Dataset: This column indicates whether the respective studies used benchmark datasets custom generated data or other relevant datasets used in the research.
F. Scalabile Characteristics: This column shows scalability of approaches that are implemented in respective paper's to convey an understanding of how effectively it operates while processing large volumes of data or how flexibly it can be operated to manage alterations to the hardware and software of the system. This is an important factor as it can show how useful an approach is in real-world scenarios.
G. Result: This highlights the outcome of the specific methods when implementing the above-discussed methods; the outcome may include the quantitative outcome of the methods, performance enhancement, or any other appropriate outcome.
H. Limitations: The following column looks into the limitation or drawback the approaches met in their various studies and this way it enables us to know the possible limitations of these techniques and also point to areas of further improvement.

The purpose of these fields will be to gain detailed information on all the papers and also, know what and where, a certain approach should be used for.

TABLE 2. COMPARISON OF COMPILER OPTIMIZATION TECHNIQUES PART 2

Technique	Dataset	Scalabile Characteristics	Result	Limitations
[1]	Google Code Jam dataset, offering diverse problems and solutions.	Adaptable to diverse optimizations without source code.	Achieves up to 85% accuracy, O2 optimization performs best with 84.61% accuracy and 84.96% F1-score	Cross-optimization detection sub-optimal. Model efficacy.
ELOPS [2]	Uses SPEC2006, NPB, and scientific computing programs for training and evaluation.	Handles offline training and online prediction efficiently, suitable for real-time applications.	Speedup compared to default optimization (-O3).1.29x and 1.26x speedup on two different platforms	Focuses on parameter prediction rather than phase selection.
BaCO [3]	Tensors from SuiteSparse, Facebook, FROSTT, synthetic data.	Handles search spaces up to billions of configurations.	Outperforms random sampling, previous auto tuners on 15 kernels. Handles billions of configure.	May face difficulty handling hidden constraints and complex search spaces,
BOCA [4]	GCC and LLVM compilers, cBench, and PolyBench benchmarks	effectively handles high-dimensional and large optimization spaces	Outperforms existing methods in efficiency.	May struggle with large spaces or resource demands
[5]	Generated dataset of 7700+ files for x86-64 Linux	Good scalability, inference time of 230µs for CNN	Accuracy 98% - binary, 95% - multiclass, 125 bytes input	x86-64 architecture gcc and clang compilers O0 and O2 optimization levels
RAMMER [6]	DNN models in TensorFlow frozen graph, TorchScript, or ONNX	Design is not limited to CUDA and NVIDIA GPUs	52k lines of C++ code, 3k lines to the core compiler and scheduling function	dividing complex DNN operators into independent homogeneous rTasks
[7]	OpenCL kernels from benchmark like AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia, SHOC	faster than prior sequence models	outperformed models on CPU /GPU mapping GNN-AST was 12% more accurate	models not outperformed state-of-the-art on thread coarsening task
VELTAIR [8]	Not specified	56 CPU Cores, Outperforms Layer-wise/Model-wise	Higher Performance, Adaptive Strategy	Limited Model Evaluation
[9]	No Specific Dataset	Scalable Generation, Diverse Topologies	Detects Bugs, Diverse Graph Structures	Graph Pattern Diversity, Attribute Exploration
[10]	Not Specified	Automated Optimization, Billions of Configs	Outperforms Libraries, Tensor Operators	Model Speed, Relative Runtime Prediction
ROLLER [11]	Not Applicable	Multi-Core Parallelism, Scale-Out	10x Improvement, rTile Configurations	Small Operators, Device Compiler Dependency
CompilerGym [12]	Benchmark datasets like cBench, Csmith, LLVM-stress	Isolated compiler services, fault tolerance, scalable concurrency	Trained on Csmith programs 0.804-1.023× code size reduction 0.659-0.987× code size reduction	High variance, Unstable convergence, High sample complexity
Autophase V2 [13]	cBench, CHStone, AnghaBench	Not discussed explicitly	- Random search: 2.3% better - RL: Up to 9% better than -Oz	High variance in rewards
[14]	CBench suite, LLVM test suite benchmarks	A3C algorithm for Efficient Search	SuperSonic Outperforms Other Methods	Multi-task learning
AutoPhase [15]	CHStone	Runs 3x faster than genetic algorithms	16% better performance than -O3 compiler flag	Requires significant training data resources
[16]	CHStone, LegUp's benchmarks	Not Discussed	Up to 23x faster learning, reduced fluctuations	Not discussed
POSET-RL [17]	LLVM-test-suite for training, MiBench and SPEC CPU for evaluation	Handles large search space using RL, scales to 90 passes in LLVM.	Reductions in code size (up to 22.94%) and enhancements in runtime (up to 46%) on SPEC 2017	Only considers code size and runtime
Bansor [18]	ResNet, BERT, DCGAN	yes	Better schedules found faster compared to Ansor	Not Mentioned
PROGRAML [19]	250k LLVM-IR Files, Diverse Sources	Linear Scaling, High Inference Throughput	Outperforms SOTA, High Accuracy	Fixed Iterations, Vocabulary Limitations
[20]	LLVM-IR (DeepTune), POJ-104, OpenCL kernels	Combines DL feature extraction with classical ML to handle small datasets	Outperforms state-of-the-art in all tasks: 91.6% (device mapping), 95.48% (algorithm classification), 1.05x speedup (thread coarsening)	Simple code representation, might miss complex code intricacies
[21]	Utilizes LLVM IR features post each pass.	Adapts to diverse program types, enhancing scalability.	Faster compilation, tailored pass sequences for performance gains	Pass dependencies and generalization challenges
[22]	SPEC CPU 2006/2017, TSVC, and LLVM Test Suite datasets.	Scalable framework, supports new additions and compiler compatibility	Achieves up to 22.4x speedup in compile time, deepens ML model integration.	Some runners not universally compatible, not fully supported for C runtime.
CHAMELEON [23]	ImageNet	AlexNet, VGG-16, ResNet-18	4.45x Speedup, 6.4% Improvement	Hyperparameter Tuning, Convolution Focus

Challenges Identified:

Incorporating ML/DL/RL in compilers also comes with the following challenges; ensure models to work across many hardware and software platforms, dealing with overhead that comes with model inference and training at the same time as ensuring that the interpretability and transparency of decisions made by the compilers are not affected.

However, there are many opportunities on using compilers with the help of ML/DL/RL, as the benchmarks and various use case types suggest when mentioning the improvements that were possible due to the introduction of such optimizations. Therefore, there will be several open issues, questions, challenges, and potential future works as follows, Going forward, several open research questions, challenges and future directions will remain. These include:

- Manufacturing many more extendable and effective algorithms in solving complex large optimization spaces efficiently.
- Ensuring that the models, which are mainly deployed machine learning compilers, are more understandable and transparent to warrant high credibility in relation to the models involved in a particular decision-making domain.
- Decisions of how important model accuracy is versus how much time is can be spared training versus how much time is required to make certain inferences which must be studied and planned out as to how to create and optimize these factors are.
- The need to find out if any other domains across the tools would possibly call for such techniques like language translation or static analysis as well as dynamic optimization with the help of ML/DL/RL.

In order to accelerate development in this domain important tasks should be addressed in cooperation with other researchers belonging to the ML/DL/RL communities.

VI. Conclusion

A research field which has arisen recently is using ML, DL and RL methods in the compiler optimization process. Any such an area is full of promise since it can enhance the capability, efficiency, resource utilization as well as flexibility of software systems. In this regard, a brief review of the various methodologies and algorithms used in this multi-disciplinary field was done. Further, the various machine learning techniques can be classified into two broad groups: A classification of learning models as supervised and unsupervised learning models. The supervised is about code generation and resource allocation while the unsupervised is about pattern identification. Another category called Reinforcement Learning can also be used for autotuning or as a decision-making tool. These methods have proved beneficial in some aspects of compilers including; scheduling of instructions, loop transformations or runtime adaptive.

This ever expanding field of ML/DL/RL employed compiler optimization has the potential of revolutionizing the way software systems are optimized so as to make optimum utilization of resources, give better performance, and adapt to evolving software and hardware platforms easily

References

[1] Singh, Shirish Kumar, Harshit Singhal, and Bharavi Mishra. "Leveraging Compiler Optimization for Code Clone Detection." In *SEKE*, pp. 516-521. 2021.

[2] Liu, Hui, Jinlong Xu, Sen Chen, and Te Guo. "Compiler Optimization Parameter Selection Method Based on Ensemble Learning." *Electronics* 11, no. 15 (2022): 2452.

[3] Hellsten, Erik Orm, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. "Baco: A fast and portable Bayesian compiler optimization framework." In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pp. 19-42. 2023.

[4] Chen, Junjie, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. "Efficient compiler autotuning via bayesian optimization." In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1198-1209. IEEE, 2021.

[5] Pizzolotto, Davide, and Katsuro Inoue. "Identifying compiler and optimization options from binary code using deep learning approaches." In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 232-242. IEEE, 2020.

[6] Ma, Lingxiao, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. "Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}." In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881-897. 2020.

[7] Brauckmann, Alexander, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. "Compiler-based graph representations for deep learning models of code." In *Proceedings of the 29th International Conference on Compiler Construction*, pp. 201-211. 2020.

[8] Liu, Zihan, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. "VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling." In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 388-401. 2022.

[9] Liu, Jiawei, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. "Nnsmith: Generating diverse and valid test cases for deep learning compilers." In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 530-543. 2023.

[10] Chen, Tianqi, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578-594. 2018.

[11] Zhu, Hongyu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue et al. "{ROLLER}: Fast and efficient tensor compilation for deep learning." In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233-248. 2022.

[12] Cummins, Chris, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. "Programl: Graph-based deep learning for program optimization and analysis." *arXiv preprint arXiv:2003.10536* (2020).

[13] Almakki, Mohammed, Ayman Izzeldin, Qijing Huang, Ameer Haj Ali, and Chris Cummins. "Autophase V2: Towards Function Level Phase Ordering Optimization." In *Machine Learning for Computer Architecture and Systems 2022*. 2022.

[14] Wang, Huanting, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. "Automating reinforcement learning architecture design for code optimization." In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pp. 129-143. 2022.

[15] Haj-Ali, Amir, Qijing Huang, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. *AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning*. Technical Report. http://arxiv. org/abs, 1901.

[16] Shahzad, Hafsah, Ahmed Sanaullah, Sanjay Arora, Robert Munafo, Xiteng Yao, Ulrich Drepper, and Martin Herbordt. "Reinforcement learning strategies for compiler optimization in high level synthesis." In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 13-22. IEEE, 2022.

[17] Jain, Shalini, Yashas Andaluri, S. VenkataKeerthy, and Ramakrishna Upadrasta. "Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning." In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 121-131. IEEE, 2022.

[18] Gao, Chao, Tong Mo, Taylor Zowtuk, Tanvir Sajed, Laiyuan Gong, Hanxuan Chen, Shangling Jui, and Wei Lu. "Bansor: Improving Tensor Program Auto-Scheduling with Bandit Based Reinforcement Learning." In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 273-278. IEEE, 2021.

[19] Cummins, Chris, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. "Programl: Graph-based deep learning for program optimization and analysis." *arXiv preprint arXiv:2003.10536* (2020).

[20] Hakimi, Yacine, Riyadh Baghdadi, and Yacine Challal. "A Hybrid Machine Learning Model for Code Optimization." *International Journal of Parallel Programming* 51, no. 6 (2023): 309-331.

[21] Jayatilaka, Tarindu, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. "Towards compile-time-reducing compiler optimization selection via machine learning." In *50th International Conference on Parallel Processing Workshop*, pp. 1-6. 2021.

[22] VenkataKeerthy, S., Siddharth Jain, Umesh Kalvakuntla, Pranav Sai Gorantla, Rajiv Shailesh Chitale, Eugene Brevdo, Albert Cohen, Mircea Trofin, and Ramakrishna Upadrasta. "The Next 700 ML-Enabled Compiler Optimizations." In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, pp. 238-249. 2024.

[23] Ahn, Byung Hoon, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. "Chameleon: Adaptive code optimization for expedited deep neural network compilation." *arXiv preprint arXiv:2001.08743* (2020).

[24] Ye, Zihao, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. "Sparsetir: Composable abstractions for sparse compilation in deep learning." In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660-678. 2023.

[25] de Souza Xavier, Tiago Cariolano, and Anderson Faustino da Silva. "Exploration of Compiler Optimization Sequences Using a Hybrid Approach." Computing & Informatics 37, no. 1 (2018).

[26] Devkota, Sabin, Pascal Aschwanden, Adam Kunen, Matthew Legendre, and Katherine E. Isaacs. "CcNav: Understanding compiler optimizations in binary code." IEEE transactions on visualization and computer graphics 27, no. 2 (2020): 667-677.

[27] Midkiff, Samuel. "Automatic parallelization: an overview of fundamental compiler techniques." (2022).

[28] Belwal, Meena, and T. K. Ramesh. "Q-PIR: a quantile based Pareto iterative refinement approach for high-level synthesis." *Engineering Science and Technology, an International Journal* 34 (2022): 101078.

[29] Jaswanth, Kunisetty, S. Sruthi, Pranav Ramachandrula, and Meena Belwal. "Data Encryption: A Compiler Based Approach." In *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pp. 1-6. IEEE, 2023.

[30] Vanishree, K., and Madhura Purnaprajna. "CPU Performance Modeling through Analysis of Primitive Operations." In *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 441-446. IEEE, 2020.

[31] Sekhar, Swathi, Yadukrishnan Gopinathan, B. Yamuna, and Karthi Balasubramanian. "Realization of turbo decoder on coarse grained reconfigurable architectures." In *2022 IEEE 19th India Council International Conference (INDICON)*, pp. 1-6. IEEE, 2022.