Compiler Optimization: A Deep Learning and Transformer-based Approach

[1]Devansh Handa, [2]Kindi Krishna Nikhil, [3]S Duvarakanath, [4]Kanaganandini Kanagaraj, [5]Meena Belwal

[1,2,3,4,5] Department of Computer Science and Engineering, Amrita School of Computing, Amrita Nagar Amrita Vishwa Vidyapeetham Bangalore-560035, Karnataka, India.

[1]devanshhandaji@gmail.com,[2]nikhilkindi1704@gmail.com,[3]duvarak05@gmail.com, [4]nandinikanagaraj03@gmail.com,[5]b_meena@blr.amrita.edu

Abstract

For the case of compiler design the performance optimization never gives up its struggle. Traditional techniques mostly utilize rules and human efforts to adjust that can underutilize the intricacies of complex modern hardware architectures. This This research puts forward a new framework that applies deep Learning techniques to performance analysis and prediction as part of optimization compiler design. Relying on the unification of DL methods with infrastructural features of the compiler, seek to mechanize the procedure of generating optimal code sequencing. It enables the identification of the most appropriate optimization strategies directly from the feedback on how the code is being used. Through this study, this paper delves into the critical aspects of the design process such as design considerations and implementation challenges as well as highlighting the experimental outcomes showing the effectiveness of the approach. From empirical evaluation, the work demonstrates improvements in code performance found on a variety of benchmarks which range from 0.7% up to 99.8% and an average of 27.8% improvement.

Keywords: Deep learning, Transformers, Optimizations, Code generation, Classification, Transfer Learning, BERT, Model Fine-Tuning.

1. Introduction

Compiler optimization now a days a significant role in software development which the goal attain the efficiency is of the essence. Translating high-level programming languages into executable machine code is performed by compilers, thus connecting the empirical gap between human-comprehensible code and the architecture of the underlying hardware. The quality of a compiler's compiling (translating computer code) has a bearing on the runtime performance, resource usage, and power consumption of the final software after the compiler has run. Despite the difficulty of creating the right programming optimizers for a wide range of architectures that keep changing all the time, along with the increasing software complexity.

In contemporary compilers, optimization techniques are mainly performed based on static analysis, heuristic rules, and manual overriding power. On the one hand, these techniques work in many cases, but more often than not they fail to take advantage of the fine-tuned inner structure of modern hardware architectures, whose complex multipurpose units specialize in massive parallel computing and are controlled by several levels of the memory hierarchy. As the software world keeps on developing, the problem of designing better optimization strategies, that could

automatically change in accordance with conditions as workload profile variation or hardware reuse rearrangement, becomes more urgent.

Transformers are a foundational deep learning architecture described by Vaswani et al. (2017), which replaced RNN and CNN based models by masking and parallelizing all inputs and outputs. They employ self-attention methods to identify contextual interactions of the words and not necessarily their order in the series. It employs the encoder-decoder structure and is very useful when it comes to such tasks as translation, text production, as well as classification. To a large extent these are now replacing previous models such as RNNs and LSTMs because of their efficiency and effectiveness to name but a few; BERT, GPT, T5Tokenizer, and many others.

Over the last few years, there has been a rise in utilizing DL techniques to provide an analysis and forecast of performance for compiler design. One way researchers try to reach this goal is by combining runtime learning with the compiler infrastructure. This will enable autonomous agents to quickly learn and adapt optimization mechanisms in real-time, by taking into account information received from application execution. This method of optimization brings about a shift in the paradigm that compiler optimization was used towards removing the static and rule-based method and move to a dynamic and data-driven approach.

The article focuses on the challenges and the prospects of the research on the domain, showing how the field of compiler optimization can be enhanced by introducing the DL into the compiler optimization pipeline. Further, go after the issues on the design specification, implementation strategies, and experimentation methods needed for DL and transformer based optimization and compiler framework. The shown DL practice achieves better performance of the code on different benchmarks and configurations. The key contributions of this work are:

- A compiler optimization framework that is able to identify the most suitable optimization technique for a given input code.
- An approach for code generation after identification of the optimization technique, this analysis the input code and generated the optimized code based on the predicted optimization technique.

The rest of the article is organized as: Section 2 studies the related work in detail analysing and understanding the recent work in the field. Section 3 discusses in detail the background of the used dataset, the steps and tools involved in making the compiler optimization platform and the logic behind the work, and finally Section 4 displays the results and evaluation pattern for this framework and compares its efficiency across all optimization techniques, this section also concludes the findings and gives reason for its performance and also discusses the future advancements.

2. Related works

Syed et al. [1] had looked into the application of Machine Learning and Data Mining to optimize High Level Synthesis without exhaustively searching the state space. From OLTP systems, they utilize account profiles for profile-based optimizations to minimize memory stalls, enhance the code's speed, and control errors and layout for web deployment. The techniques of compiler optimization proposed by them focus on data and improve efficiency, effectiveness, and quality.

Verma et al. [2] had discussed TensorRT (TF-TRT) as well as TensorFlow Lite (TFLite) only in terms of throughputs, latency and power consumption in the context of edge computing. Selecting various deep learning models for reference, the authors determined that the TF-TRT is unsurpassed in terms of the floating-point operations and TFLite is best among the lightweight models. Their work generates a new benchmarking technique and offers directions that future studies of edge computing can use.

Shewale et al. [3] outlined a three-step approach that utilizes the IHGS to boost compiler optimization predictions. They manually collected C-compiled files and apply on various models (HSSSM, PRO, CMBO, ARCHOA, DO, GOA) to show the better performance of IHGS getting 0. 90 and 0. 94 at 80 and 90 learning phases, it is significantly higher than that of CMBO and HSSSM. They describe how IHGS can help in increasing the accuracy of the compiler optimizations by a large measure.

Cummins et al. [4] introduced the Compiler Gym Python library for tasks like LLVM phase ordering, GCC flag selection, and CUDA loop nest construction. Their framework leverages data-driven autotuning and reinforcement learning, offering a research environment with benchmarks and tools for analyzing computational complexity and reinforcement learning in compiler optimization.

Chen et al. [5] proposed BOCA, an optimization method combining Bayesian optimization and Random Forests to enhance compiler sequences. BOCA achieves performance improvements of 42% to 90% by using machine learning approaches, significantly benefiting compiler autotuning with real-world performance gains and improved noise tolerance.

Pizzolotto et al. [6] applied CNN and LSTM methods for resume learning to identify players and optimize binaries. They compared DNN bin opcodes and raw bytes, with CNN achieving 94.05% accuracy and LSTM 84% using binary samples of 125 and 2048 bytes. Binary classification reached 98% accuracy. The study, using the GCC dataset with 48% accuracy, highlights opportunities to enhance compiler optimization detection across different compilers.

Hussain et al. [7] introduced a method using Recurrent Neural Networks (RNNs) to optimize power, performance, and area (PPA) for core and memory IP blocks. By leveraging weight sharing from previous models, their approach reduces required data features while maintaining 96-98% accuracy. This method enhances memory compiler efficiency and explores design adjustments for core optimization.

Colucci et al. [8] developed a Phase Sequence Selector (PSS) and Performance Estimator (PE) for compiler optimization, targeting energy use, execution time, and code complexity. Their model, trained on diverse applications, achieved less than 2% misclassification, improving program performance and reducing costs with adaptive phase sequence selection.

Haj-Ali et al. [9] propose using deep reinforcement learning (RL) to optimize phase ordering in HLS compiler design. Their model learns sequences of compiler optimizations, achieving a 16% performance improvement over traditional -O3 flags and outperforming genetic algorithms on 12 benchmarks from CHstone and LegUp. The research highlights the effectiveness of deep RL in

HLS and suggests future work to refine these techniques and explore other compiler-related challenges.

Meena and T.K. Ramesh [10] use an objective function to optimize HLS application objectives (area, latency, power) and determine Pareto solutions. They apply the N-PIR scheme, based on Random Forest learning, to refine the Pareto front. N-PIR outperforms heuristic clustering and lattice navigation, improving system design by addressing conflicting objectives and combining model refinement with feature selection in UML design.

Belwal, Meena, and T. K. Ramesh [11] introduced Quantile-based Pareto Iterative Refinement (Q-PIR) for optimizing HLS. Using Average Distance from Reference Set (ADRS) for quality assessment, Q-PIR enhances design efficiency, quality, and cost. Compared to IRF-random and IRF-TED, Q-PIR shows superior results on benchmarks like Transposed Filter and ADPCM Encode/Decode, achieving better design quality and reduced evaluation costs. It provides more accurate and efficient Pareto-optimal solutions than existing methods.

Cummins et al. [12] propose a machine learning approach using Large Language Models (LLMs) for LLVM-IR code optimization. These models, trained on extensive datasets, generate optimized code with a 90.5% success rate. Performance is measured using BLEU score, exact match frequency, and code metrics. The study shows that LLMs significantly improve code optimization accuracy and efficiency, outperforming traditional methods in producing optimizations and reducing compilation errors

Xiangzhe Xu et al. [13] introduced Transformer models pre-trained with BinaryCorp-3M, BinKit, and HowSolve datasets for improved binary code similarity comparison. They developed the DiEmph method, focusing on semantics-driven instruction deemphasis, which enhances performance by integrating deep learning and program analysis. Their method showed up to a 14.4% improvement in Precision at 1 (PR@1).

Mingzhen Li et al. [14] surveyed deep learning compilers, analyzing TVM, nGraph, TC, Glow, and XLA across 19 ONNX models. TVM outperformed others in end-to-end and layer-by-layer comparisons, highlighting the effectiveness of these optimizations.

Rishab Sharma et al. [15] analyzed code attention in BERT using the CodeSearchNet dataset for code clone detection. They found that specific code representations improved BERT's performance, with F1-score gains of 605% in lower layers and 4% in upper layers, and CodeBERT showed a 21-24% improvement using identifier embeddings. Their study emphasized attention distribution across BERT layers

Kuiliang Lin et al. [16] applied priority-guided differential testing to find bugs in DL compilers, focusing on TVM using the DeepDiffer framework. They used mutation strategies and IR-Pass controls, identifying 13 bugs in TVM and uncovering 9 fundamental root causes, with evaluations covering API inconsistency, exception handling, and buffer overflow.

Sanket Tavarageri et al. [17] proposed an Automated-Compiler for optimizing GEMM operations in deep learning scripts. High-level optimizations used polyhedral compilation and deep learning, while low-level methods included target-specific code generation and reinforcement learning.

Their approach achieved performance comparable to Intel oneDNN and outperformed AutoTVM, with 7.6X and 8.2X speed-ups over the baseline for sequential and parallel cases, respectively. This study highlights how compiler strategies can enhance numerical computations essential for deep learning

3. Background and Methodology

Proposed methodology can be observed in Figure 1 where one can find the different sections of processing which includes data pre-processing, code classification, code generation and finally evaluation.

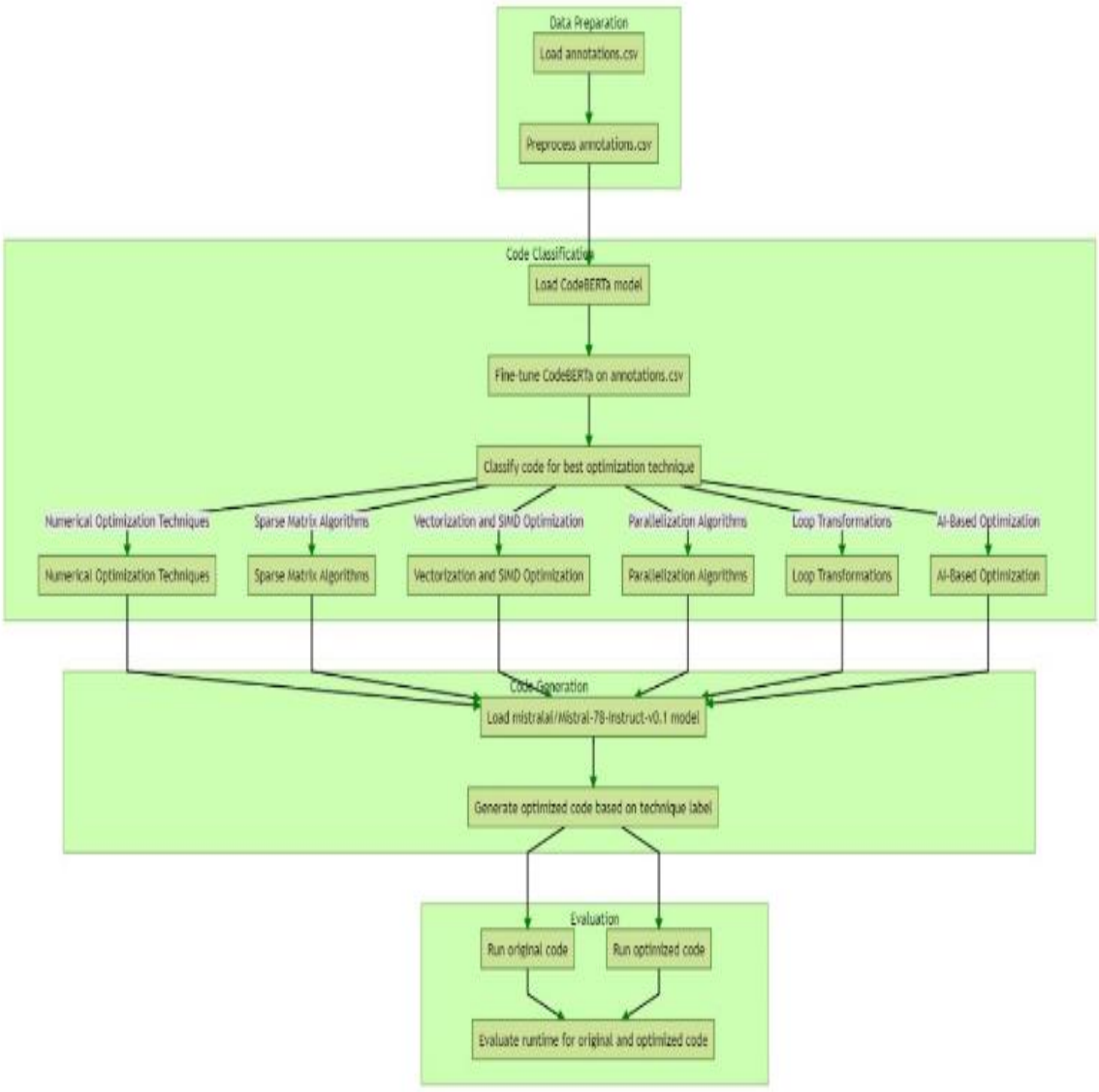

Figure 1. Proposed methodology

3.1 Data Preparation

3.1.1 Dataset

The CodeSearchNet dataset includes Python code, typically functions or methods with comments or docstrings, categorized by the MuSE data model and supplemented with metadata like file path and repository info. It contains millions of code-documentation pairs, reflecting diverse Python coding styles, useful for various research and development challenges.

The original dataset includes codes of different Languages as seen in Figure 2, amongst them the instances with Python codes are extracted and stored in a separately for further use.

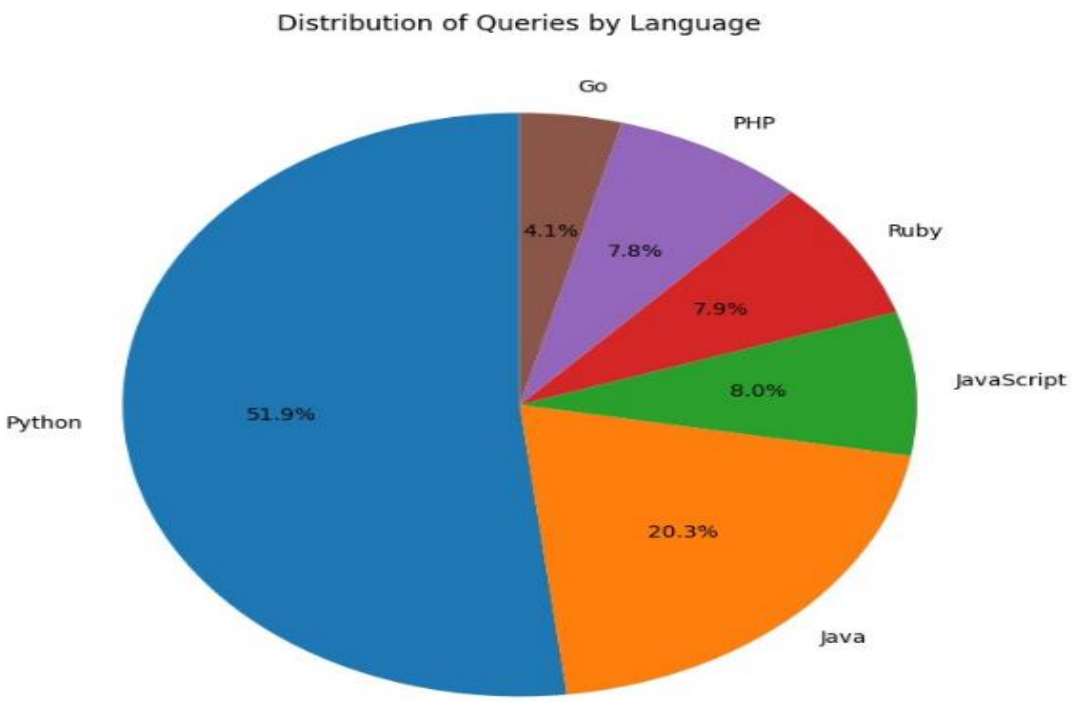

Figure 2. Data distribution of the dataset

3.2 Code Classification

3.1.2 CodeBERTa

CodeBERTa is actually an extension of BERT which has to do with programming languages. It is an open-source machine learning model that is part of the RoBERTa family for code and fine-tuned models for code understanding, Generation, and translation. CodeBERTa is initially trained on massive source code from GitHub and can thus accustom to syntactic and semantic features in different programming languages. Some of its functions include code analysis and prediction, report generation and summarization, and detection of bugs in code, hence being a valuable tool

for software engineers in code-intensive projects. As a result of using the transformer architecture, CodeBERTa is indeed able to accurately capture the inherent nature and usage patterns of programming languages, which is beneficial for numerous software engineering and development tasks. In this project CodeBERTa is used for finding the suitable optimizing techniques for given code.

This model is essentially fine-tuned on the data which allows it to classify an input code among the given 6 classes for code optimization.

3.3 Code Optimization Techniques

- Loop Transformations: To enhance the usage of cache and parallelism, it is possible to redesign loops as loop unrolling, fusion or tiling.
- Sparse Matrix Algorithms: The following recommendations are made about computations involving matrices with predominantly zero elements: Efficient storage formats and computations must be employed to reduce the memory consumption and the time needed for computations.
- Vectorization and SIMD Optimization: Optimize by switching to vector operations, using Single Instruction, Multiple Data (SIMD) instructions to pass operations to as many values at once as possible.
- Parallelization Algorithms: Divide a particular task into different sub-tasks that can run concurrently on different cores/machines in order to decrease the amount of time it takes to run a particular task or enhance the system's scalability.
- Numerical Optimization Techniques: Solve mathematical issues with programs like gradient descent, Newton's method or a genetic algorithm for minimization/maximization of objective.
- AI-Based Optimization: Implement machine learning as well as neural networks in the enhancement of techniques used in the understanding and efficient functioning of large systems using data.

4. Optimized Code Generator

4.1 Mistral-7B-instruct-v0.1

Mistral-7B-Instruct-v0.1 is an Open-Domain language model, with focus on instructions which disambiguates about 7 billion parameters for providing precise and context relevant output responses to a number of tasks and questions.

This model is also responsible for code generation, for doing so, firstly a different approach was tested where the initial idea was to train and load the model, this would lead to a harder time in deploying the proposed methodology. To overcome this, a fine-tuned version of the model is called from HuggingFace Hub via inference API which is making this process faster and efficient.

Above method is used to generate optimized code for the given input. Each model is evaluated using 4 different codes and the model is executed for 6 categories (code optimization techniques) and the results are noted

5. Results and Evaluation

The methodology proposed makes use of the powers of Deep Neural Networks in order to create a code optimizing system that can predict which code optimization technique.

Table 1. Execution time comparison between original and optimized code

Input	Predicted Class						Gain percentage (%)
	Numerical Optimization Techniques		Sparse Matrix Algorithms		Vectorization and SIMD optimization		
	Original code(sec)	Optimized code (sec)	Original code	Optimized code	Original code	Optimized code	
Code 1	1.64E-05	9.60E-06	-	-	-	-	41.5
Code 2	0.0010326	0.0009985	-	-	-	-	3.3
Code 3	5.20E-06	4.60E-06	-	-	-	-	11.5
Code 4	1.60E-06	1.40E-06	-	-	-	-	12.5
Code 5	-	-	0.0001117	0.0001006	-	-	9.9
Code 6	-	-	2.02E-05	1.85E-05	-	-	8.4
Code 7	-	-	-	-	4.50E-06	4.12E-06	8.5
Code 8	-	-	-	-	9.66E-05	4.30E-05	55.5

Input	Predicted Class						Gain percentage (%)
	Parallelization Algorithms		Loop transformation		AI based Optimization		
	Original code	Optimized code	Original code	Optimized code	Original code	Optimized code	
Code 9	1.98E-05	1.00E-06	-	-	-	-	9.5
Code 10	2.90E-06	2.20E-06	-	-	-	-	2.4
Code 11	-	-	49.318029	38.171392	-	-	22.6
Code 12	-	-	0.212131	0.210573	-	-	0.7
Code 13	-	-	-	-	0.413266	0.001002	99.8
Code 14	-	-	-	-	0.125023	0.000001	99.9

to apply and generate the optimized code, finally this code is then compared and evaluated based in the its runtime (Execution time).

The proposed framework was thoroughly tested with various code samples. In this section the results obtained for 14 test cases are depicted in Table 1. Each row of Table 1 represents the instance where a single code is given as input to the system.

The Gain percentage in Table 1 represents the percentage in time that the optimized code is faster than the original code. To further analyze the gain, it can be observed to be in the range 0.7% - 99.9%. The highest gain was achieved for code 13 (Matrix Multiplication (AI-Based Optimization)) and code 14 (Fibonacci Sequence (AI-Based Optimization) both under AI-Based Optimization. This is used to validate the functioning and accuracy of the proposed system.

The specific code used in Table 1 and their corresponding operations can be observed in Table 2, all the codes are taken from the dataset CodeSearchNet. For representative purpose Matrix Multiplication is referred as MM in the upcoming sections.

Table 2. Code and its Operation

Code	Operation
Code 1	Factorial
Code 2	Fibonacci Sequence (Numerical Optimization)
Code 3	Exponentiation (Power)
Code 4	Greatest Common Divisor - GCD
Code 5	Multiplies Multiplication (Sparse Matrix)
Code 6	Matrix Addition
Code 7	Vector Sum
Code 8	Matrix Multiplication (Vectorization)
Code 9	Generate Primes
Code 10	Sum of Squares of the elements
Code 11	Matrix Multiplication (Loop Transformation)
Code 12	Array Initialization
Code 13	Matrix Multiplication (AI-Based Optimization)
Code 14	Fibonacci Sequence (AI-Based Optimization)

The following section presents the codes as test cases given which were divided among the 6 classes of optimization Techniques.

5.1 Test Cases

- Numerical Optimization Techniques: The case where the input code, python function origc1 represents the input original code and the corresponding generated optimized code is the function optic1.It can also be observed that the functions have the same objective of finding the factorial of a given variable n. This optimization technique worked best due to the nature of the code written - as by giving it early condition to stop, it was able to decrease the recursion depth.

- Sparse Matrix Algorithms: Function origc2 represents the input original code and the corresponding generated optimized code is the function optic2. It can also be observed that the functions have the same objective of finding the result for multiplication of two matrices A and B. This optimization technique worked best due to the nature of input matrices being sparse.
- Vectorization and SIMD Optimization: Function origc3 represents the input original code and the corresponding generated optimized code is the function optic3. It can also be observed that the functions have the objective of finding the result for multiplication of two matrices A and B, but the difference here from the previous case is the existence of dense matrix here, and a few differences in the method of execution, this is the reason vectorization worked best in this case.
- Parallelization Algorithms: Function origc4 represents the input original code and the corresponding generated optimized code is the function optic4. The observed codes have the objective of finding the first n primes. This optimization worked best in this case due to the application of dividing the task into chunks which helps in faster calculation as calculation of each prime is independent.
- Loop Transformations: Function origc5 represents the input original code and the corresponding generated optimized code is the function optic5. The observed codes have the objective of finding the multiplication of two matrices, once again here the difference is the input size, and a few other lines of code. Since it is the basic matrix multiplication code, Loop optimization worked best in this case.
- AI-Based Optimization: Function origc6 represents the input original code and the corresponding generated optimized code is the function optic6. The observed codes have the objective of finding the position n - Fibonacci series member. This optimization worked best in this case due to elimination of calculation, where the optimized code would directly predict the Fibonacci sequence as it is always fixed for a given input.

The test cases present above indicate the gain in time that was achieved by the system for various given input codes across all classes. Through this evaluation it can be concluded that AI-based optimization resulted in gain due to its ability to predict output and eliminate calculation.

By increasing the fine-tuning process to finally include more instances and running the model on a higher configuration of GPU were the only limitations faced, increasing these could possibly make the system full proof for all the algorithm classes, and can possible be scaled to include more classes of optimization techniques in python.

6. Conclusion

Deep learning and transformers are incorporated into compiler optimization, which has proved an efficiency enhancement proposal. Thus, using models such as CodeBERTa and Mistral-7B-

Instruct, the system adapts proper optimization methods which could reach up to 99%. Being specific, there was a definite improvement in the execution time that indicated a positive change punctually. This approach suggests how AI can revolutionize the classical compiler optimization, so it is more suitable for practical usage.

Further research will include use of a larger training set, retraining models with larger sample sizes and the use of other forms of optimization. The framework must be further assessed on other applications of a higher complexity level as well as on larger scales since robustness is achieved by its scalability. Moreover, creation of an interface that can fit most of the development environments known at the moment and become an inalienable part of the developer's routine is imperative for a wider dissemination of the system.

References

1. Syed, Aamir, Ashwin Harish, and Sini Anna Alex Keerthana Purushotham. "A Survey of Adaptive Compiler Optimization Heuristics."
2. Verma, Gaurav, et al. "Performance evaluation of deep learning compilers for edge inference." 2021 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2021.
3. Shewale, Chaitali, et al. "Compiler optimization prediction with new self-improved optimization model." International Journal of Advanced Computer Science and Applications 14.2 (2023).
4. Cummins, Chris, et al. "Compilergym: Robust, performant compiler optimization environments for ai research." 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2022.
5. Chen, Junjie, et al. "Efficient compiler autotuning via bayesian optimization." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.
6. Pizzolotto, Davide, and Katsuro Inoue. "Identifying compiler and optimization options from binary code using deep learning approaches." 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020.
7. Hussain, Sabir, Ma Raheem, and Afaq Ahmed. "Memory Compiler Performance Prediction using Recurrent Neural Network." 2023 IEEE Devices for Integrated Circuit (DevIC). IEEE, 2023.
8. Colucci, Alessio, et al. "MLComp: A methodology for machine learning-based performance estimation and adaptive selection of Pareto-optimal compiler optimization sequences." 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2021.
9. Haj-Ali, Ameer, et al. "AutoPhase: Compiler Phase-Ordering for High Level Synthesis with Deep Reinforcement Learning." arXiv preprint arXiv:1901.04615 (2019).
10. Belwal, Meena, and T. K. Ramesh. "N-pir: a neighborhood-based pareto iterative refinement approach for high-level synthesis." Arabian Journal for Science and Engineering 48.2 (2023): 2155-2171.
11. 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.

12. Cummins, Chris, et al. "Large language models for compiler optimization." arXiv preprint arXiv:2309.07062 (2023).
13. Xu, Xiangzhe, et al. "Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis." Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023.
14. M. Li et al., "The Deep Learning Compiler: A Comprehensive Survey," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 708-727, 1 March 2021, doi: 10.1109/TPDS.2020.3030548.
15. Sharma, Rishab, et al. "An exploratory study on code attention in bert." Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. 2022.
16. K. Lin, X. Song, Y. Zeng and S. Guo, "DeepDiffer: Find Deep Learning Compiler Bugs via Priority-guided Differential Fuzzing," 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), Chiang Mai, Thailand, 2023, pp. 616-627, doi: 10.1109/QRS60937.2023.00066.
17. Tavarageri, Sanket, et al. "AI Powered Compiler Techniques for DL Code Optimization." arXiv preprint arXiv:2104.05573 (2021).
18. Bikku, Thulasi, Jyothi Jarugula, Lavanya Kongala, Navya Deepthi Tummala, and Naga Vardhani Donthiboina. "Exploring the effectiveness of BERT for sentiment analysis on large-scale social media data." In 2023 3rd International Conference on Intelligent Technologies (CONIT), pp. 1-4. IEEE, 2023.
19. S. K. Vasudevan, Prashant R. Nair, Abhishek, S. N., Kumar, V., and Aswin, T. S., “An Innovative Application for Code Generation of Mathematical Equations and Problem Solving”, Journal of Intelligent and Fuzzy Systems, vol. 36, no. 3, pp. 2107-2116, 2019.