Work-in-Progress: ACPO: An AI-Enabled Compiler Framework

Amir H. Ashouri Muhammad Asif Manzoor Minh Vu Raymond Zhang Ziwen Wang
Angel Zhang Bryan Chan Tomasz S. Czajkowski Yaoqing Gao
Huawei Technologies, Heterogeneous Compiler Lab
Toronto, Canada

***Abstract*—This paper presents ACPO: An AI-Enabled Compiler Framework; a novel framework that provides LLVM with simple and comprehensive tools to enable employing ML models for different optimization passes. We showcase a couple of use cases of ACPO by ML-enabling the Loop Unroll (LU) and Function Inlining (FI) passes and experimental results reveal that by including both models, ACPO can provide a combined speedup of 2.4% on Cbench when compared with LLVM's O3.**

***Index Terms*—Compilers, ML, AI, LLVM, Optimizations**

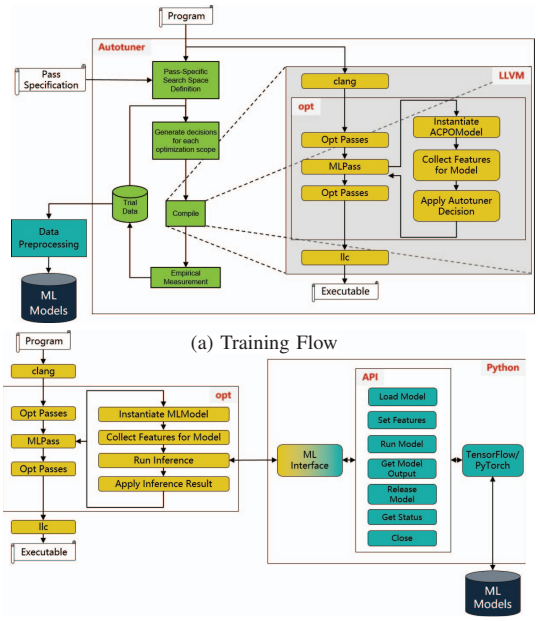
I. Introduction

Recently, a number of works [1], [2] leverage Machine Learning (ML) in an ML-guided optimization (MLGO) approach for which the infrastructure is baked into a single or multiple passes to streamline the optimization process readily. Trofin et al. [1] propose MLGO that uses ML to advise the Inliner pass as to whether or not a call site should be inlined to minimize the size of the generated code. Contrary to MLGO, we focus on the hard problem of `ML performance` optimization, where the key challenge is the development of a model architecture and the features used to capture key performance use cases. ACPO proposes the following contributions: 1) A framework to provide a comprehensive set of predefined handcrafted features, libraries, and algorithmic methods by enabling compiler engineers with a user-friendly interface to instantiate `ACPOModel` classes to replace LLVM's existing hard-coded heuristics. 2) ML APIs and the compiler are seamlessly connected, but at the same time, they are not interdependent. 3) Showcasing the benefits of ACPO, we demonstrate two different scenarios with LLVM: a) Loop Unroll Pass — building both the interface and the ML model. b) Function Inlining Pass — building the interface and leveraging an existing ML model [2].

II. Proposed Methodology & Results

Figure 1 presents the high-level design of our infrastructure.The autotuner is a wrapper around the compiler, and its hooks in the compiler are used to generate model training data. The inference flow begins when an ML-enabled pass is invoked and creates an instance of `ACPOModel` which then aggregates appropriate features and specifies the kind of output required from the ML model. The object then transfers the features and information about the model being invoked to our ML framework via a set of remote procedure call APIs. The APIs provide the ability to send requests, including loading an appropriate model into the ML framework, executing an inference call, and returning the result of the inference back to the compiler. The compiler then applies transformations to the input program as prescribed in the inference output it receives. Table I showcases that we gain 4.1 % against MLGO work when we use FI model, and 2.1%, 1%, and 2.4% compared with LLVM's O3 when we use FI, LU, and both models, respectively. Future works will focus on extending ACPO to optimize other LLVM transformation passes.

*Full version of this work is available at: https://arxiv.org/abs/2312.09982



(a) Training Flow

(b) Inference Flow

Fig. 1: ACPO Flow

TABLE I: Cbench Results (Sz%: Size bloat, Sp: Speedup. cols 1, 3-4 are wrt. O3 and 2, is ACPO-FI against MLGO [1])

Benchmark	ACPO-FI		wrt. MLGO		ACPO-LU		Combined	
	Sz%	Sp	Sz%	Sp	Sz%	Sp	Sz%	Sp
bitcount	1	0.999	1	1.005	1.001	1.003	1	1.001
qsort1	1	0.994	1	1.003	1	0.982	1.91	0.989
bzip2d	1.441	0.991	1.239	1.021	2.224	1.01	3.541	1.014
bzip2e	1.441	1.007	1.189	1.041	2.35	1.001	3.546	1.015
jpeg_c	1.288	1.017	1.2	1.005	5.385	0.986	5.086	1.027
jpeg_d	1.289	0.923	1.19	1.335	5.4	1.073	4.807	1.026
lame	1.437	1.027	1.137	1.007	4.501	1.03	4.729	1.034
mad	1.217	1.071	1.16	1	4.321	0.99	3.193	0.968
dijkstra	1	0.993	1	0.996	1	0.99	1	1.037
patricia	1	0.98	1.052	0.99	1	0.974	1	1.004
g-script	1.275	1	1.16	1.04	1	1	1.12	1.001
ispell	0.996	1	1.17	1.02	1	1	1	1
rsynth	1	1	1.12	1.04	2.024	1.01	2.024	1.011
strings	1	1.006	1	1	1	1	1	1
pgp_d	1.861	0.993	1.12	1	1	0.991	13.144	1
pgp_e	1.861	1.017	1.12	1.042	1	1.006	13.144	1.006
rij_d	1	0.99	1.39	1.039	3.823	0.966	1	0.966
rij_e	1	1.001	1.09	1.035	3.823	0.974	1	0.995
sha	0.999	1.27	1.009	1.132	1	1.174	0.999	1.501
pcm_c	1	1	0.996	1.001	1	1.008	1	1.038
pcm_d	1	0.997	1	1	1	1.132	1	1.138
CRC32	1	1.094	1	1.066	1	1.042	1	1.112
gsm	1.81	1.076	1.05	1.04	1.815	1.15	1.81	1.117
Geomean	**1.152**	**1.021**	**1.161**	**1.041**	**1.818**	**1.01**	**2.442**	**1.024**

References

[1] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[2] A. H. Ashouri, M. Elhoushi, Y. Hua, X. Wang, M. A. Manzoor, B. Chan, and Y. Gao, "Mlgoperf: An ml guided inliner to optimize performance," *arXiv preprint arXiv:2207.08389*, 2022.

2643-1726/24/$31.00 ©2024 IEEE
DOI 10.1109/CASES60062.2024.00011