Crop: An Analytical Cost Model for Cross-Platform Performance Prediction of Tensor Programs

Xinyu Sun Yu Zhang* Shuo Liu Yi Zhai
University of Science and Technology of China
yuzhang@ustc.edu.cn

ABSTRACT

Learn-based cost models used for tensor compiler auto-tuning often suffer from poor performance when trained on one hardware platform and applied to another. This issue necessitates collecting performance data for each potential platform during model deployment, incurring significant overhead.

We propose Crop, a comprehensive and universal analytical cost model designed for cross-platform performance prediction of tensor programs. Crop decouples program features and hardware features. It gathers hardware-independent program features on one platform and predicts their performance based on parametric hardware features for a given platform. Crop achieves comparable levels of prediction accuracy to that of a learn-based cost model while ensuring portability.

1 INTRODUCTION

High-performance tensor programs are essential for the efficient execution of machine learning models. In recent years, there has been a growing trend among researchers and practitioners to utilize tensor compilers [2] for searching tensor programs (i.e., optimal low-level implementations). Key to the search [11] is a cost model for evaluating the performance of candidate tensor programs.

Deep learning-based cost models achieve good prediction accuracy by learning directly from the tensor program performance data, but they require large amounts (millions) of performance data for training. Regrettably, the execution time of a tensor program can significantly vary across different hardware platforms owing to distinctions in hardware architecture. Consequently, a cost model trained on one hardware platform usually performs poorly on another, giving rise to the issue known as cross-hardware unavailability. This issue necessitates collecting performance data for each potential platform during model deployment, incurring significant overhead.

In response, TenSet [12] pre-trains cost models on source platforms and fine-tunes them on target platforms. Moses [10] utilizes model distillation to iteratively update the pre-trained cost model during the auto-tuning process. TLP [8] treats the source and target platforms as multiple tasks in multi-task learning and employs a multi-head neural network as a cost model. While these approaches have proven effective, they still demand a considerable amount (tens of thousands) of data collection on target platforms.

In this paper, we articulate a fundamental insight: the pivotal solution to the challenge of cross-hardware unavailability lies in the decoupling of **program features** and **hardware features**. By combining the hardware-independent program features from the tensor program with the hardware-dependent hardware features from the target platform specifications, the feasibility of performance prediction for tensor programs on these target platforms is realized without the substantial overhead associated with measurements.

We propose **Crop**, a comprehensive and universal analytical cost model designed for **Cro**ss-**p**latform **p**erformance **p**rediction of tensor programs. Crop decouples program features and hardware features, gathering hardware-independent program features on one platform and predicting their counterpart performance for given platforms based on parametric hardware features.

The most notable advantage of Crop lies in the ability to conduct all program data collection on a single source platform, eliminating the requirement for any tensor program measurements on potential target platforms. This attribute is especially advantageous in scenarios where the target platform demonstrates lower performance (e.g. mobile, embedded devices) or when deploying tensor programs across diverse target platforms, effectively mitigating the inefficiencies associated with measurement overhead.

On the source platform, Crop conducts static and dynamic analysis on LLVM IR [6] to extract program features for tensor programs. These features encompass control flow graphs with effective basic block counts, task graphlets representing data dependencies, and reuse profiles that characterize memory access patterns. As tensor compilers commonly leverage LLVM for code generation on CPUs, we have ensured the universality of Crop, making it applicable across various tuning frameworks rather than being specific to any particular one.

On given target platforms, Crop considers the characteristics of modern processors, including multi-threading parallelism, instruction-level parallelism, cache hierarchy, SIMD instructions, and branch prediction. By combining features of the program and the hardware, Crop predicts the performance of the tensor program for given target platforms.

We evaluated the performance of Crop using the popular tensor compiler TVM [2] on the state-of-the-art tensor program dataset, TenSet. Crop achieves comparable levels of prediction accuracy to that of a learn-based cost model while avoiding the overhead of measurement on target platforms.

In summary, this paper makes the following contributions:

- An analytical modeling approach for cross-platform performance prediction that requires no tensor program measurements on potential target platforms.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658249

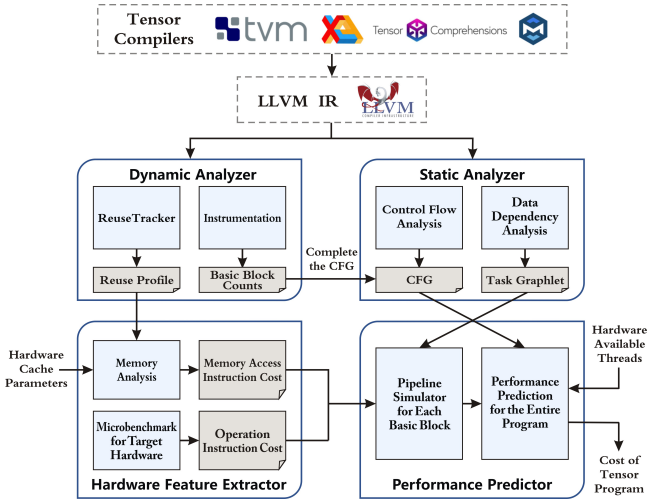


Figure 1: Framework Overview of Crop

- A feature extraction method to separately acquire hardware-independent features from tensor programs and hardware features from target platforms.
- An implementation and evaluation of the proposed method. To our knowledge, Crop is the first analytical modeling approach that effectively addresses the challenge of cross-platform unavailability.

2 PERFORMANCE MODELING

This section introduces the foundational concepts underpinning Crop's analytical modeling of tensor programs. We delve into the methodology for predicting the performance of tensor programs, leveraging hardware-independent program features and the hardware characteristics specific to target platforms.

2.1 Total Execution Cost of Tensor Programs

To comprehensively analyze the performance of various tensor programs, the basic idea of Crop is to partition the complete tensor program into a set of basic blocks and predict the execution performance of each basic block. Then, based on the frequency of occurrence of each basic block during program execution, the overall execution performance of the complete tensor program is calculated.

The total execution cost of tensor program P can be calculated using the following Equation 1:

$$Cost(P) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \Big(Cost(BB_{ij}) \times ECount(BB_{ij}) \Big) \tag{1}$$

where n is the number of functions in the program P, and m_i is the number of basic blocks in the i-th function. $Cost(BB_{ij})$ and $ECount(BB_{ij})$ represent the execution cost and effective execution count of the j-th basic block in the i-th function, respectively.

2.2 Execution Cost of Basic Blocks

The execution cost of a given basic block BB_{ij} can be represented by the following Equation 2:

$$Cost(BB_{ij}) = Pipeline\Big(Tg(BB_{ij}), Cost(Insts) \Big) \tag{2}$$

where $Tg(BB_{ij})$ represents the task graphlet of BB_{ij}, which characterizes the instructions within the basic block and their data dependency relationships. As a hardware-independent program feature, we extensively discuss it in subsection 3.1. $Cost(Insts)$ represents the execution cost of various instructions on a specific platform. In subsection 3.3, we provide a detailed description of these hardware features for target hardware platforms. *Pipeline* is a simulation-based pipeline analyzer that calculates the execution cost for a given basic block based on the input task graphlet and instruction costs, which we introduce in subsection 3.4.

2.3 Effective Execution Counts of Basic Blocks

We refer to the count of basic blocks that impact the overall execution time of the program as the effective basic block count. To achieve high-performance implementations, tensor programs leverage the multi-threading features of multi-core processors, distributing parallel tasks as evenly as possible among the threads. As a result, tensor programs typically consist of both serial and parallel sections, which need to be discussed separately. The effective count of a given basic block BB_{ij} is defined by the following Equation 3:

$$ECount(BB_{ij}) = \begin{cases} Count(F_i) \times Count(BB_{ij}) & \text{if } F_i \text{ serial} \\ \left\lceil \dfrac{Count(F_i)}{threads} \right\rceil \times Count(BB_{ij}) & \text{if } F_i \text{ parallel} \end{cases} \tag{3}$$

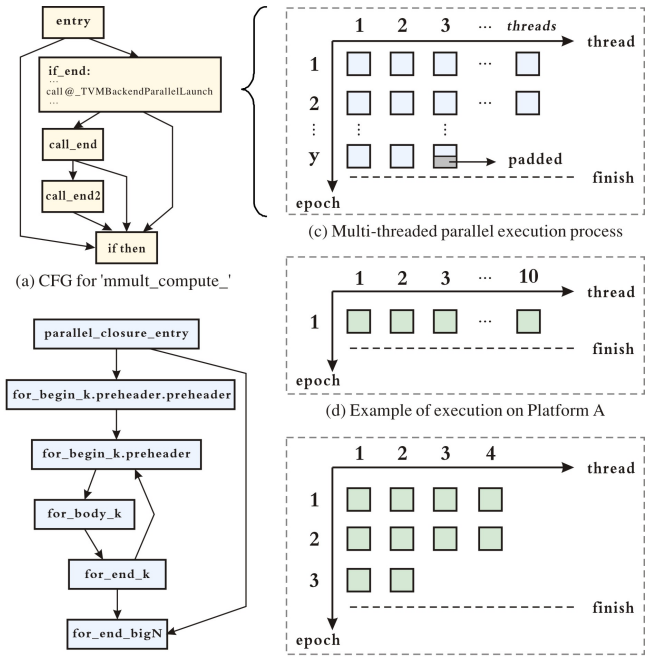
where $Count(F_i)$ represents the number of executions of function F_i, while $Count(BB_{ij})$ represents the number of executions of basic block BB_{ij} per execution of function F_i. $threads$ represents the number of threads used if F_i is executed in parallel.

For the case when F_i is executed serially, each execution of F_i impacts the overall execution time of the program. On the contrary, if F_i is executed across multiple threads, the effective execution count is determined by the thread with the maximum number of executions, which can be calculated with $Count(F_i)$ and $threads$.

$Count(F_i)$ and $Count(BB_{ij})$ are determined by the program itself and its schedule, so for a compiled tensor program, they are fixed and independent of the hardware. In subsection 3.2, we describe how to obtain these program features through dynamic analysis.

At compile time, the tensor compiler declares which code will be executed concurrently, typically encapsulated within a function without specifying the specific number of parallel threads. At runtime, the tensor program dynamically determines the number of available threads for parallel execution based on the size of the thread pool on the target platform. Therefore, the parallel behavior of the same tensor program may vary on different platforms due to the difference in the hardware parameter $threads$.

Figure 2(a) describes a control flow graph (CFG) of function `mmult_compute_`, from a matrix multiplication program that has been optimized with Parallelization by TVM. Function `mmult_compute_` itself is executed serially, but in the `if_end` basic block, the



(a) CFG for 'mmult_compute_'

(b) CFG for '_tvm_parallel_lambda'

(c) Multi-threaded parallel execution process

(d) Example of execution on Platform A

(e) Example of execution on Platform B

Figure 2: Examples for CFGs and multi-thread parallelism

function `_TVMBackendParallelLaunch` is called, which leads to the creation of a parallel program fragment.

The execution process of the multi-thread parallel program fragment is depicted in Figure 2(c). The $Count(F_i)$ instances of F_i are evenly distributed across $threads$ number of threads for parallel execution, completing within y epochs. Each F_i sent to a thread can be analyzed as an independent serial program. In this example, F_i refers to the `_tvm_parallel_lambda` function. Its control flow graph is depicted in Figure 2(b).

In Figure 2(d) and (e), we provide a set of specific examples to illustrate how different available threads affect the parallel execution process. For $Count(F_i) = 10$ instances that need to be parallelized, on Platform A with $threads \geq 10$ available threads, all tasks can be completed in $y = 1$ epoch. However, on Platform B with $threads = 4$ available threads, it requires $y = 3$ epochs to complete all tasks. According to Equation 3, the calculation of $ECount(BB_{ij})$ for a basic block in F_i yields $1 \times Count(BB_{ij})$ for Platform A and $3 \times Count(BB_{ij})$ for Platform B, respectively.

In this section, we provide the formulas for the cost of tensor programs and explain the meaning of each term in the formulas. In the next section, we delve into the implementation details of Crop.

3 DESIGN AND IMPLEMENTATION

In this section, we present the implementation details of Crop. The overall framework of Crop is illustrated in Figure 1. Crop consists of four main components: static program analyzer, dynamic program analyzer, hardware feature extractor, and performance predictor.

3.1 Static Program Analyzer

The static program analyzer directly extracts program features from the LLVM IR without the need for program execution. It extracts the control flow graph of each function in the input program and the task graphlet of each basic block.

A basic block is a contiguous and indivisible sequence of straight-line code with a unique entry and exit point. There are no intermediate branches within a basic block, except for possible branching at the exit. We provide an example of several basic blocks within a loop in Figure 3(a). The basic blocks and the instructions within them are both deterministic when given the LLVM IR of a program.

3.1.1 Achievement of Control Flow Graphs. The control flow graph (CFG) is a directed graph composed of nodes and edges. Nodes represent basic blocks in the program, while edges represent the transition relationships between the basic blocks. In Figure 2(a) and (b), we provide the CFGs of two functions as examples. Through control flow analysis, we can obtain the successor and predecessor for each basic block.

The static program analyzer utilizes an LLVM Pass to perform analysis and achieve CFGs for inputs.

3.1.2 Extraction of Task Graphlets. The static program analyzer parses through each basic block to generate the task graphlets, which represent the data dependency relationships between instructions.

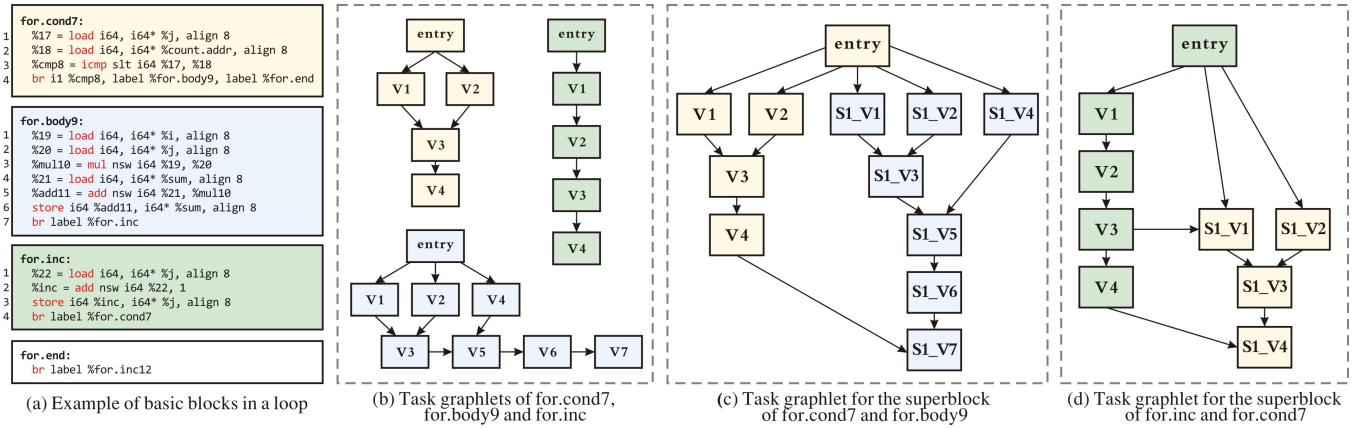
While instructions within a basic block are logically executed sequentially, not every instruction depends on the execution of all its preceding instructions. Modern compilers and hardware architectures aim to exploit these opportunities for instruction-level parallelism to fill their instruction pipelines and achieve better performance. Therefore, extracting task graphlets forms the basis of analyzing the execution cost of basic blocks.

In a task graphlet, vertices symbolize instructions, and directed edges depict data dependency relationships. The entry vertex signifies the starting point of the current basic block. Essentially, the task graphlet succinctly delineates the instructions that must be executed before a specific instruction to satisfy its necessary data dependencies.

As an example, we illustrate the task graphlet of the basic block `for.body9` from Figure 3(a) in the lower half of Figure 3(b). In this basic block, vertices V1 and V2 represent `load` instructions, respectively. Vertex V3 is a `mul` instruction that depends on the results of the previous two instructions. Vertex V5 is an `add` instruction that depends on the computation result of vertex V3 and the result of the `load` instruction vertex V4. Vertex V6 is a `store` instruction used to save the computation result of vertex V5. Finally, the vertex V7 representing the `br` instruction satisfies its data dependency when all other instructions have been completed. Similarly, we further provide the task graphlets of `for.cond7` and `for.inc` in Figure 3(b).

In the scenario of accurate branch prediction, the current basic block and its successor basic block can be treated as a continuously executed superblock. To analyze the influence of branch prediction on pipeline performance, we merge each basic block with its respective successor basic block separately for analysis, representing the data dependency relationships within the merged superblock.

In Figure 3(c) and (d), we provide examples of task graphlets for the superblocks. To distinguish vertices in the current basic block from those in the successor basic block, we prefix the names of vertices in the successor basic block with *S1* to indicate their origin from the first successor basic block. For the superblock merged by



(a) Example of basic blocks in a loop

(b) Task graphlets of for.cond7, for.body9 and for.inc

(c) Task graphlet for the superblock of for.cond7 and for.body9

(d) Task graphlet for the superblock of for.inc and for.cond7

Figure 3: Examples for Basic Blocks and Task Graphlets

`for.cond7` and `for.body9`, there is no data dependency between the instructions of the two basic blocks, resulting in a large space for instruction-level parallelism. However, for the superblock merged by `for.inc` and `for.cond7`, vertex V3 stores the variable `%j`, while vertex S1_V1 loads the same variable `%j`, creating a read-after-write dependency that will impact the execution of this superblock.

The static program analyzer utilizes a custom LLVM Pass to extract task graphlets for each basic block and potential superblock.

3.2 Dynamic Program Analyzer

The dynamic program analyzer executes tensor programs on a source platform and extracts hardware-independent program features, including the effective execution counts of each basic block and reuse profile that characterize memory access patterns.

3.2.1 Achievement of Effective Execution Counts. In Section 2, we discuss how to calculate effective execution counts for both the serial and parallel portions separately. Leveraging the infrastructure of LLVM, we use an instrumentation pass to insert counters into each basic block in the LLVM IR, collecting counts at runtime.

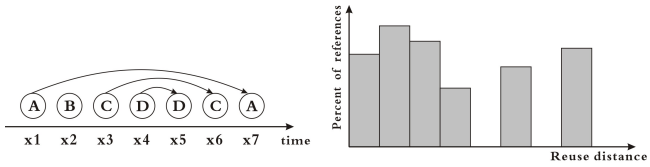
For each function F_i in the tensor program, we obtain its execution count $Count(F_i)$. For each basic block BB_{ij} within F_i, we collect their execution counts $Count(BB_{ij})$ during one execution of F_i. These counts are hardware-independent program features, allowing us to obtain them on a source platform and generalize them to target platforms confidently. For a given target platform, we calculate the effective execution counts for each basic block using Equation 3.

3.2.2 Characterizing Memory Access Patterns. Due to the critical impact of the memory access behavior of tensor programs on performance, characterizing them is a key step in performance modeling. The usage of the reuse profile, also known as the reuse distance distribution, allows us to characterize the memory access behavior in tensor programs.

Reuse distance serves as a hardware-independent metric, defined by the number of unique memory locations accessed between two consecutive accesses to a specific memory location. For instance, consider the sequence of accessed memory locations illustrated in Figure 4(a). In this example, the reuse distance for A is 3, signifying three unique memory locations (B, C, and D) between two consecutive accesses to A.

Small reuse distances typically indicate good data locality, while large reuse distances may lead to cache misses. By collecting the reuse distances for all memory addresses in the program, individual reuse distances form a histogram known as the reuse profile, as shown in Figure 4(b).

We utilize ReuseTracker [7], the state-of-the-art open-source sampling-based reuse analyzer, to obtain the reuse profile of tensor programs. ReuseTracker can accurately characterize the reuse profile on private and shared caches in parallel applications.



(a) Example of reuse distance

(b) Example of reuse profile

Figure 4: Examples for Reuse Distance

3.3 Hardware Features Extractor

We characterize the hardware of each target platform with parametric features, as enumerated in Table 1. The Parallelism and Memory parameters can be directly obtained by consulting the processor's manual.

3.3.1 Memory Access Instruction Costs. Based on the reuse profile and specific hardware memory parameters, we calculate the cache hit rate specific to the target platform [3] [5] [1]. Equation 4 calculates $P(h|D)$ given reuse distance D:

$$P(h|D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{(D-a)} \tag{4}$$

where D denotes the reuse distance, A denotes cache associativity, and B denotes cache size in terms of the number of blocks, which is the cache size divided by the cache line size. Then we calculate

the approximated unconditional probability of cache hit $P(h)$ for the entire program as follow Equation 5:

$$P(h) = \sum_{i=0}^{N} P(D_i) \times P(h|D_i) \tag{5}$$

where $P(D_i)$ is the probability of the ith reuse distance D in the reuse profile. The average cost of memory access instruction δ_{avg} depend on the cache hit rates at different cache levels, be calculated as follow Equation 6:

$$\delta_{avg} = P_{L1}(h) \times \delta_{L1} + (1 - P_{L1}(h)) \Big[P_{L2}(h) \times \delta_{L2} + (1 - P_{L2}(h)) \times [P_{L3}(h) \times \delta_{L3} + (1 - P_{L3}(h)) \times \delta_{RAM}] \Big] \tag{6}$$

where δ_{L1}, δ_{L2}, δ_{L3} and δ_{RAM} are latencies of L1, L2, L3 caches and RAM. $P_{L1}(h)$, $P_{L2}(h)$ and $P_{L3}(h)$ are the probabilities of cache hit for L1, L2 and L3 caches.

3.3.2 Operation Instruction Costs. We automatically acquire costs of various operation instructions by running a set of custom microbenchmarks on the target platform [9]. Each microbenchmark includes thousands of instructions of a specific category, excluding any other instruction types. By capturing the execution time of each microbenchmark, we calculate the average execution time for each type of IR instruction. By incorporating vector instructions and fused multiply-add instructions in our microbenchmarks, we analyse the impact of SIMD instructions on program performance.

The construction of microbenchmarks are based on the LLVM compiler, enabling an automated process without the need for manual intervention. The measurement of instruction costs for various types can be completed in seconds, presenting a stark contrast to methods requiring extensive tensor program measurements on the target platform.

Table 1: Target Hardware Parameters with Examples

Category	Parameter	Examples (Intel Xeon Gold 5220R)
Parallelism	threads	48
Memory	Cache Size	1536 KB, 48 MB, 71.5 MB
	Cache Line Size	64 bytes
	Cache Associativity	8, 16, 11
	Cache Latency	4 cycles, 14 cycles, 59.5 cycles
	RAM Latency	108 cycles
Instruction	Memory Access Instruction Cost	Calculated by formulas
	Operation Instruction Cost	Obtained from microbenchmark

3.4 Performance Predictor

To consider instruction-level parallelism in modern processors, we utilize a pipeline simulator to compute the execution time of each basic block, achieving the computation outlined in Equation 2. The pipeline simulator is built upon the Simian [4] Parallel Discrete Simulation Engine.

Figure 5 illustrates the analysis process of a task graphlet in the pipeline simulator. The simulation execution commences from the top of the task graphlet. If the current vertex already satisfies data dependencies, it is incorporated into the pipeline simulation. If data dependencies are not met, the current vertex waits for its predecessor vertices to complete execution. The time each vertex spends in the pipeline is determined by its execution cost. This iterative process continues until all vertices in the current basic block are completed, ultimately providing the simulated execution time for the basic block.

Finally, we integrate the results obtained from the calculations in Equation 2 and Equation 3, utilizing Equation 1 to compute the total execution cost of the entire tensor program.

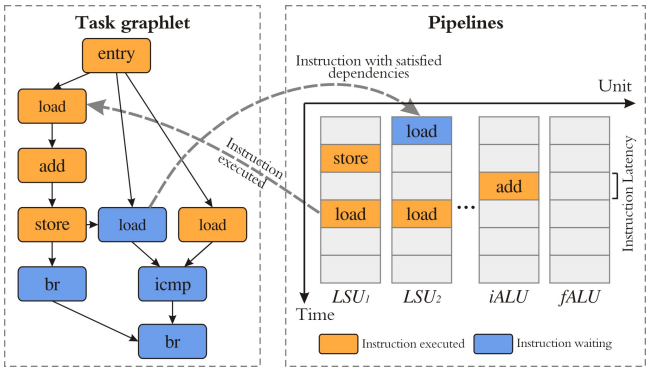


Figure 5: Analysis Process of the Pipeline Simulator

4 EVALUATION

In this section, we assess the accuracy of Crop in cross-platform performance prediction for tensor programs.

Tenset represents the state-of-the-art tensor program dataset, comprising a multitude of tensor programs derived from popular deep learning models. After dedicating several weeks to remeasuring Tenset on the Intel Xeon Gold 5220R, designated as the target platform, we acquired a substantial dataset, including 8,100k performance data.

From this dataset, we hold out a test set within 100k performance data, consisting of five networks: ResNet-50, MobileNet-V2, ResNext-50, BERT-tiny, and BERT-base. The remaining data is allocated for the training set to facilitate the utilization of learn-based cost models. To show Crop's cross-platform capabilities, we gather the hardware-independent features of these 100k tensor programs on the AMD EPYC 7763, which serves as the source platform.

4.1 Intuitive Showcase of Crop's Accuracy

We present the precision of Crop's predictions in an intuitive manner. Figure 6 illustrates the prediction curves of Crop alongside the actual measurement curves, demonstrating a closely aligned and highly similar trend. In these experiments, each group of tensor programs shares remarkably similar schedules, with the only distinction being the tiling size. The results indicate that Crop can effectively discern these subtle differences, conduct quantitative analysis, and identify the optimal tiling size — a crucial factor in our tuning process.

Table 2: The top-k scores of different cost models on test set with average scores and data demand

Name	Metric	ResNet-50	MobileNet-V2	ResNext-50	BERT-tiny	BERT-base	Average	Data demand
MLP	Top-1 Score	0.8951	0.7081	0.6847	0.8098	0.8634	0.7922	8,000k
	Top-5 Score	0.9348	0.8865	0.9153	0.8825	0.9302	0.9098	
XGBoost	Top-1 Score	0.8547	0.7259	0.8504	0.6620	0.7582	0.7702	8,000k
	Top-5 Score	0.9430	0.9173	0.8937	0.8506	0.9442	0.9097	
TLP	Top-1 Score	0.9255	0.8117	0.8880	0.8495	0.8967	0.8742	500k
	Top-5 Score	0.9642	0.9304	0.9491	0.9201	0.9525	0.9432	
Crop	Top-1 Score	0.8132	0.7266	0.7083	0.7861	0.7381	0.7545	N/A
	Top-5 Score	0.8581	0.8824	0.8337	0.8608	0.8897	0.8650	

4.2 Comparison between Crop and Learn-based cost models

MLP and XGBoost, serving as naive cost models, are trained using the entire 8,000k training set from the target platform. TLP is the state-of-the-art cost model designed to address the challenge of cross-platform unavailability and employs a multitask approach for transfer learning. Due to its multi-head design, it simultaneously utilizes 500k training data from the target platform and 8,000k available data from Tenset, sourced from Intel Platinum 8272.

In Table 2, we present the Top-k accuracy of Crop and various learn-based cost models. The Top-k accuracy is a metric used to characterize the ability of a cost model to select the best-performing program from a set of candidate programs.

The experimental results demonstrate that Crop achieves prediction accuracy comparable to learn-based cost models trained with sufficient training data.

Without utilizing any tensor performance data from the target platform, Crop exhibits a performance gap of only 2.5% to 4.4% compared to naive cost models, and 8.5% to 13.7% compared to the state-of-the-art TLP. In the absence of performance data from the target platform, our proposed Crop model has a significant opportunity to surpass all existing approaches, emerging as the optimal choice.

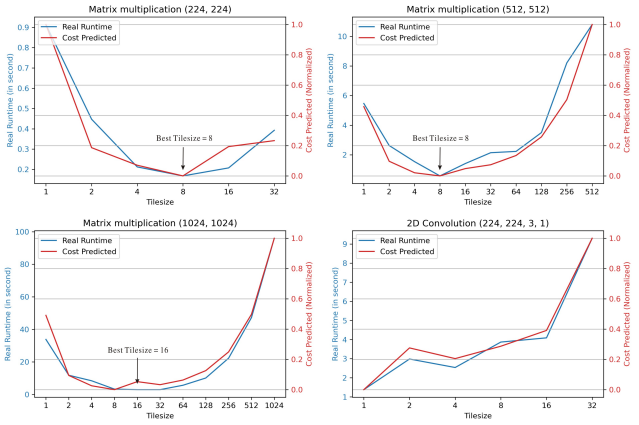


Figure 6: Examples for Prediction Curves of Crop

5 CONCLUSION

In this paper, we propose and extensively describe Crop, a comprehensive and universal analytical cost model designed for cross-platform performance prediction of tensor programs. Crop decouples program features and hardware characteristics by collecting hardware-independent program features on one platform and predicting their performance based on parametric hardware features for given platforms. Crop achieves prediction accuracy comparable to that of a learn-based cost model while entirely circumventing the overhead of collecting performance data on the target platform.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science foundation of China (No. 62272434) .

REFERENCES

[1] Atanu Barai, Yehia Arafa, Abdel-Hameed Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2022. PPT-Multicore: performance prediction of OpenMP applications using reuse profiles and analytical modeling. *The Journal of Supercomputing* (2022), 1–32.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 20 (2018).

[3] Gopinath Chennupati, Nandakishore Santhi, Robert Bird, Sunil Thulasidasan, Abdel-Hameed A Badawy, Satyajayant Misra, and Stephan Eidenbenz. 2018. A scalable analytical memory model for CPU performance prediction. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8*. Springer, 114–135.

[4] Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Scalable performance prediction of codes with memory hierarchy and pipelines. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 13–24.

[5] Gopinath Chennupati, Nandakishore Santhi, Phill Romero, and Stephan Eidenbenz. 2021. Machine Learning–enabled Scalable Performance Prediction of Scientific Codes. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 31, 2 (2021), 1–28.

[6] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[7] Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri Marzijarani, and Didem Unat. 2021. ReuseTracker: fast yet accurate multicore reuse distance analyzer. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–25.

[8] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.

[9] Weizhe Zhang, Meng Hao, and Marc Snir. 2017. Predicting HPC parallel program performance based on LLVM compiler. *Cluster Computing* 20 (2017), 1179–1192.

[10] Zhihe Zhao, Xian Shuai, Neiwen Ling, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2023. Moses: Exploiting Cross-Device Transferable Features for on-Device Tensor Program Optimization. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. 22–28.

[11] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[12] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.