LearnCNM2Predict: Transfer Learning-based Performance Model for CNM Systems

Anderson Faustino da Silva[1,2][0000−0002−8588−8197], Hamid Farzaneh[1][0000−0002−1780−6217], João Paulo C. de Lima[1,3][0000−0001−9295−3519], Asif Ali Khan[1][0000−0002−5130−9855], and Jeronimo Castrillon[1,3][0000−0002−5007−445X]

[1] TU Dresden, Dresden, Germany
{hamid.farzaneh,joao.lima,asif_ali.khan, jeronimo.castrillon}@tu-dresden.de
[2] State University of Maringa, Maringa, Brazil
afsilva@uem.br
[3] ScaDS.AI, Dresden, Germany

Abstract. Compute-near-memory (CNM) systems have emerged as a promising solution to address the von Neumann bottleneck by moving computation closer to memory and utilizing dedicated logic near memory arrays or banks. Despite their early stage of development, these architectures have demonstrated significant performance improvements over traditional CPU and GPU systems in various application domains. CNM architectures tend to excel in memory-bound workloads that exhibit high levels of data-level parallelism. However, identifying which kernels can take advantage of CNM execution poses a considerable challenge for software developers. This paper introduces a transfer learning approach for predicting performance on CNM systems. Our method harnesses knowledge from previously analyzed applications to enhance prediction accuracy for new, unseen applications, thereby reducing the necessity for extensive training data for each application. We have developed a feature extraction framework that captures CNM-specific computation and memory access patterns, which are crucial for determining performance. Experimental results demonstrate that our transfer learning model achieves high prediction accuracy across diverse application domains, showcasing robust generalization even in scenarios with limited training data.

Keywords: Compute near memory, processing in memory, memory wall, von Neumann bottleneck, cost model, performance model.

1 Introduction

In conventional CPU/GPU systems, frequent data movement between compute and memory modules—known as the von Neumann bottleneck—significantly hampers performance and energy efficiency, especially in data-intensive workloads where off-chip communication is 10 to 100 times more costly compared to

the computation itself [15]. Compute-near-memory (CNM) systems address this by moving computation closer to data [13, 21]. These architectures have particularly gained traction with the rise of data-heavy applications, such as machine learning and bioinformatics. Notable CNM systems, such as UPMEM [23], Samsung's FIMDRAM [14], Alibaba's PNM Engine [18], and SK hynix's AiM [10], have demonstrated substantial improvements in energy efficiency, performance, and memory bandwidth compared to conventional CPU/GPU systems. However, the extent of these performance gains is highly dependent on an application's memory-to-compute ratio and its computational patterns. In some cases, certain workloads may even exhibit worse performance on CNM systems than on traditional architectures [5]. Consequently, precise performance models are crucial for making informed decisions regarding workload offloading. Furthermore, while compiler optimizations can significantly impact performance [12], their efficacy is intrinsically linked to the availability of robust performance modeling.

High-level analytical performance models exist for CNM architectures; however, they are often too abstract and fail to capture key design complexities, programming model constraints, and memory management overheads associated with these systems [2, 11]. Performance models based on machine learning (ML) better capture and approximate these intricate relationships [17]. Nonetheless, they have not been explored for CNM systems. More importantly, traditional ML-based models often require extensive training from scratch to achieve satisfactory accuracy, demanding large amounts of training data and considerable time for hundreds of epochs. To this end, we propose *LearnCNM2Predict* (L2P), a transfer learning-based performance model targeted explicitly at CNM systems.

Figure 1 compares the L2P's time against the training time of a non-transfer learning model (non-TL) and actual hardware execution for generating a training set in the non-TL case. Although non-TL training time is not significantly high, the bottleneck is the training set generation for an unseen application. L2P enables rapid adaptation of pre-trained performance models to new applications, requiring less than 1% of the training data while maintaining accuracy. As shown, L2P's significantly lower fine-tuning and prediction times make it a practical choice for design space exploration and compiler optimization for CNM systems, where the search space is typically prohibitively large.

The L2P's pipeline takes applications written in a high-level language such as C/C++[1], preprocesses them to extract meaningful high-level embeddings that capture architectural properties and computational characteristics and then uses a Multi-layer Perceptron (MLP) network to predict performance.

The model is trained on 17 different variants of the input applications (generated using different compiler pass pipelines) and various system configurations (by varying the number of parallel compute units). This rigorous training enables the model to generalize effectively to previously unseen applications. To evaluate the effectiveness of L2P, we validate its results against a high-end UPMEM

[1] We use benchmark suites of C/C++ applications in this paper. However, L2P is not restricted to these and can be used with other languages and representations.

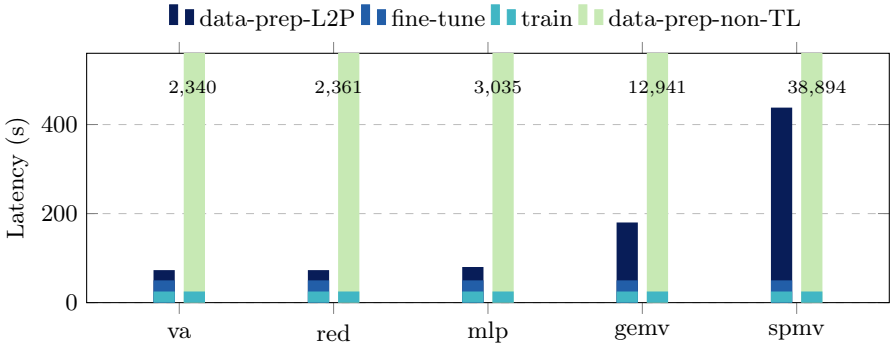


Fig. 1: Time required for generating a training set of 12,000 program variants (UPMEM-machine), non-TL model training and L2P fine-tuning for the following benchmarks: vector addition (`va`), reduction (`red`), matrix-vector multiply (`gemv`), sparse matrix-vector multiply (`spmv`) from [7].

CNM system. Our performance model accurately predicts execution times and speedups for the UPMEM, with a MAPE within ±17%.

2 Background and Related Work

This section provides background on transfer learning, the UPMEM CNM system, and state-of-the-art performance prediction for CNM systems.

2.1 Transfer Learning

In transfer learning (TL), a pre-trained model is adapted to a new task through fine-tuning requires less training than training a model from scratch while preserving many features of the original model. In the context of performance models, TL refers to reusing performance knowledge or architectural features from pre-trained models to accelerate the development of performance predictors for new workloads or hardware targets. This approach reduces data collection and training costs by transferring learned patterns from source domains (e.g., tensor programs) to target domains (e.g., graph processing programs). Often, only the last layers of a model are allowed to adapt to the new task, while the remaining layers, responsible for feature extraction, are frozen. Similarities between pre-trained and target models determine the transfer effectiveness. TL addresses further compiler optimization challenges by reusing performance models concerning optimization passes across applications.

TL-based cost models have been effectively used in the context of compiler optimizations. ATFormer [1] employs attention mechanisms to model scheduling dependencies, enabling cross-device adaptability. This method extracts transferable patterns from code and hardware, facilitating rapid adaptation to tensor-optimized architectures but remains limited to tensor programs. Sasaki et al. [19]

develop a TL cost model that reduces training samples by 60% for compiler optimizations, using feature alignment between source/target platforms. Transfer-Tuning [6] reuses auto-schedules across tensor operators for faster tuning than TVM's autoscheduler. Trümper et al. [22] map code segments to performance embeddings, enabling similarity-based optimization transfers with 28% better heuristic selection than genetic algorithms. However, none of the prior works apply TL in the cost model for commercial CNM architectures.

2.2 UPMEM System

Compute-near-memory (CNM) is a data-centric paradigm that brings compute units closer to the data and integrates them in the peripheral circuitry or directly on the memory chip to reduce data movement. While many CNM systems exist in the literature, including those from the tech giants Samsung and SK-hynix, the only commercially available CNM system to date is UPMEM. It integrates *data processing units* (DPUs) within the memory package but outside the memory arrays [23]. As shown in Fig. 2, each DIMM consists of two ranks, each containing eight memory chips, each hosting eight DPUs. Each DPU features exclusive access to 64 MB of DRAM bank (MRAM) for bulk storage and 64 kB scratchpad (WRAM) for low-latency operations, and 24 kB instruction memory (IRAM). An integrated DMA engine handles data transfers between MRAM and WRAM.

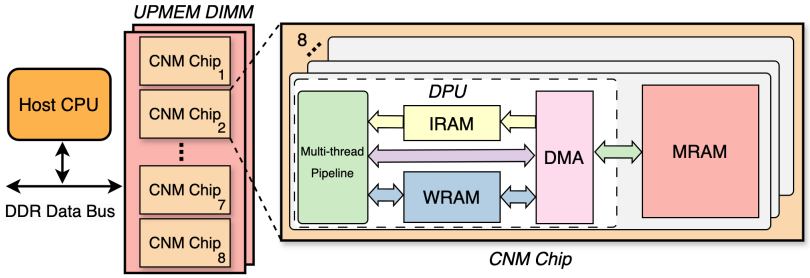


Fig. 2: UPMEM system overview

DPUs are simple RISC processors with a 12-deep pipeline, providing native support for 32-bit integer operations. Other data types are supported via software emulation. To mitigate DMA latency, each DPU employs 24 hardware threads, known as tasklets, using an interleaved multi-threading (IMT) design, where only one tasklet executes per cycle. Tasklets in a DPU can share data using WRAM and MRAM. Still, DPUs can not directly communicate and access each other's memories, requiring host arbitration for every cross-DPU data movement.

2.3 Performance Prediction for CNM Systems

Not many performance models exist for CNM systems. In isolated cases, performance models based on analytical approaches have been proposed that capture

key system properties, such as instruction-level parallelism, thread count, and memory access patterns. However, these models are highly abstract and lack the details needed to model the CNM architectures accurately. For instance, in our experiments with the UPMEM system, execution time prediction using [4] deviates by an order of magnitude for certain benchmarks.

For conventional computing systems, machine learning (ML)-based cost models have been effectively used for performance prediction [16, 19]. These models analyze performance data from extensive benchmark runs on target systems, identifying relationships between optimization parameters and execution metrics. However, despite their advantages, existing ML-based cost models face adaptability challenges. Generating training datasets for new domains and applications is time-consuming, as the cost of dataset creation grows with the number of program variants – each defined by its source code and optimization pass sequences – that must be evaluated on the target system. For instance, specialized models for UPMEM systems, such as [4], require profiling memory-bound workloads across thousands of CNM cores to quantify bandwidth benefits. When new optimization passes are introduced, a new training dataset must be generated, and the entire model must be retrained. This is extremely inefficient and results in longer performance prediction times, as shown in Fig. 1. L2P addresses this issue through TL. Instead of requiring full retraining, it can achieve acceptable accuracy across different domains and applications with only inexpensive fine-tuning, as we demonstrate for CNM systems in this paper.

3 Transfer Learning-based Performance Model

Our TL approach for performance prediction builds upon the distinct features of CNM's architecture while tackling the challenges of cross-application generalization. This section details the architecture of our ML model, highlighting how it leverages knowledge from source applications to improve prediction accuracy for target applications.

The foundation of effective TL lies in developing representations that simultaneously capture architecture-specific performance patterns and application-agnostic computational characteristics. Our model achieves this dual objective through a carefully designed network architecture that isolates application-invariant features while preserving the critical aspects of CNM performance behavior consistent across different workloads.

Figure 3 shows the flow of the L2P model, including key stages: encoding, training, fine-tuning, and prediction. The input application is compiled using the *Device compiler* tool-chain with different pass pipelines (compiler flags) and target specifications to generate the training and fine-tuning data. From the compiled output, the encoding module extracts the histogram of operations and uses it together with the target specifications to generate embeddings. During the training phase, these embeddings, along with the training dataset, i.e., the execution time of the training data on the *CNM Hardware*, are used as input to a multi-layer neural network. The ML Model continuously monitors performance

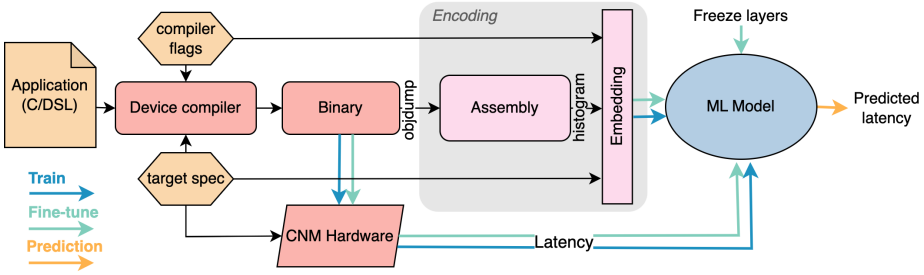


Fig. 3: A high-level overview of the L2P flow.

metrics to avoid overfitting and ensure convergence toward minimal prediction errors. This rigorous training establishes the foundation for accurate and robust cross-application performance prediction. Building on this foundation, the inference phase, composed of fine-tuning and prediction, shows how the trained model processes input features from unseen target applications (extracted the same way as in training) to predict key latency metrics such as runtime and speedup. The model architecture includes configurable frozen layers optimized for TL, enabling it to adapt knowledge gained from source applications to structurally and contextually different target applications. This phase highlights the model's ability to generalize, effectively addressing the challenges associated with variability (transformations) and structural changes (different implementations) in the input application, thereby validating its versatility in diverse application domains.

3.1 LearnCNM2Predict Model

The L2P model utilizes an MLP architecture to estimate program runtime and the potential speedup over a fixed system configuration; this indicates that L2P is a multi-objective model. The network consists of an input layer encoding program features, 16 hidden layers for extracting complex performance patterns, and an output layer directly predicting execution metrics. Each hidden layer maintains consistent dimensionality, enabling residual connections throughout the network to preserve gradient flow – particularly valuable for capturing the non-linear relationships between code characteristics and performance outcomes.

The model incorporates batch normalization after each linear transformation to normalize activation distributions, stabilize training, and improve convergence when learning from heterogeneous performance data. ReLU activation functions introduce the necessary non-linearity to model complex performance behaviors across different hardware configurations. To prevent overfitting to specific performance patterns, dropout is implemented with a configurable rate, ensuring the model generalizes well across diverse inputs and execution environments.

The residual connections are especially beneficial for performance modeling as they allow the network to effectively combine low-level features (instruction

count and memory access patterns) and higher-level abstractions (algorithmic complexity factors) when making predictions. The linear output layer produces numerical estimates of runtime and speedup without activation, preserving the regression scale. This architecture balances predictive power with training stability, making it particularly effective for capturing the intricate relationships between program characteristics and their performance on the target hardware.

The extraction of program embeddings, represented as the *Encoding* in Figure 3, relies purely on static analysis. Our approach to generating training and evaluation sets involves analyzing opcode histograms of CNM programs derived from various applications while varying the CNM configuration (e.g., DPU and tasklet counts), compiler configuration (e.g., sequence of optimization passes), and workload size. Despite their simplicity, opcode histograms have proven surprisingly effective in capturing key program characteristics [3, 20]. Static opcode histograms are extracted from compiled binaries without execution, capturing the raw instruction mix representative of each program variant. These histograms serve as embedding functions, mapping program variants onto a vector space.

Opcode histograms, while computationally simple and effective embeddings, lack sensitivity to input data characteristics. To address this, we augment our feature encoding with dataset size and workload-specific parameters. This allows the model to learn how input scaling and workload variations affect runtime and speedup. By embedding these attributes alongside opcode histograms and configuration codes, we improve the representation's fidelity, capturing both program structure and input-driven behavior for more accurate and robust performance prediction across diverse CNM workloads.

3.2 Methodology for Hyperparameter Optimization and Precision Evaluation

We established a comprehensive methodology to identify optimal hyperparameters for the L2P model.

Setup The experimental setup utilizes the UPMEM CNM system for data generation, equipped with 16 DIMMs containing 128 DPUs each (2,048 DPUs total). The host machine featured an Intel Xeon Silver 4216 Processor with 264 GB of RAM running Ubuntu Linux v22.04. All training, testing, and inference tasks were conducted on a separate system with an AMD Ryzen Threadripper 3960X 24-Core Processor, 64 GB of RAM, and an NVIDIA GeForce RTX 3090 GPU running Ubuntu Linux v20.04.

Benchmarks We leveraged the publicly available PrIM [8] and PIM-ML [9] benchmark suites, representing diverse application domains, along with UPMEM codes generated by the Cinnamon compiler [12]. We excluded three applications from PrIM (SEL, UNI, and NW) due to implementation errors. We selectively evaluated four specific benchmarks from the PIM-ML suite: Linear Regression (LiR-) and Logistic Regression (LoR-), both with and without quantization.

Dataset Generation The optimization and evaluation processes rely on a robust dataset generated by running benchmarks on UPMEM with varying numbers of DPUs (1-16) and tasklets (1-16, >= DPUs), creating 15 distinct hardware configurations. We enhanced this dataset through data augmentation using 1,004 compiler configurations (1,000 unique optimization sequences plus 4 standard compiler optimization levels), initially yielding 15,060 samples per benchmark. After filtering invalid entries, approximately 12,000 samples remained.

L2P Configuration After systematic evaluation for hyperparameters tuning, the optimal L2P configuration was determined to be a model with 12 hidden layers, with the first 6 layers frozen during fine-tuning. Each layer consists of 256 neurons with a dropout rate of 0.5. The model performs best with a batch size of 16 and a learning rate of 0.0001. Initial training utilized up to 1000 epochs with 20% patience, while fine-tuning employed 1% of inference data with a maximum of 100 epochs and 20% patience.

Metrics Throughout the optimization and evaluation processes, we evaluated L2P model precision using MAPE (for prediction accuracy) and R^2 (for goodness-of-fit).

Availability L2P is open-sourced and can be accessed and tested at `https://github.com/ComputerSystemsLaboratory/LearnCNM2Predict`.

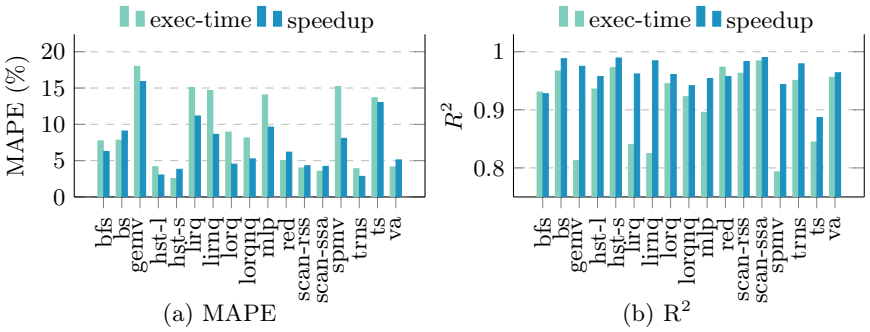
3.3 LearnCNM2Predict Precision

This section evaluates the L2P model's accuracy in predicting application performance by analyzing error metrics and comparing its forecasts to observed values. This assessment will utilize our benchmark dataset, partitioned in a 3:1:1 ratio for training, validation, and testing, respectively.

The model exhibits varying degrees of predictive accuracy across benchmarks, with higher accuracy in predicting speedups than runtimes for several benchmarks, as illustrated in Figure 4. Although our baseline uses a UPMEM system configuration with one DPU, one tasklet, and the compiler optimization level -O0, our framework is designed to be compiler-agnostic and can model performance with different optimization levels. Indeed, our goal is not to find configurations that outperform specific compiler optimizations, but rather to accurately predict performance across a variety of scenarios.

On average, the model's predictions show similar precision for speedup and runtime. The runtime predictions have an average error rate of approximately 8.04%, slightly higher than the 7.11% average error rate for speedup. Runtime errors range from a best-case scenario of 2.53% (HST-S) to a worst-case scenario of 17.97% (GEMV), while speedup errors range from a best-case of 2.83% (TRNS) to a worst-case of 15.89% (GEMV). These results indicate that while the model performs comparably well overall, runtime predictions exhibit higher average errors and more significant variability, suggesting it is more challenging to model runtime dynamics than accurate speedup. Specific benchmarks,

such as SpMV, exhibit notable differences in the error between speedup (8.07%) and runtime predictions (15.31%), suggesting the model performs better when predicting one metric over the other, depending on the characteristics of the benchmark.



(a) MAPE (b) R^2

Fig. 4: Precision of the L2P model.

On the other hand, the R^2 results demonstrate consistently high performance across nearly all benchmarks for both speedup and runtime, with values exceeding 0.80 in all cases. This suggests that, while the model achieves a good overall fit between actual and predicted values, the apparent differences between MAPE values indicate that smaller deviations in absolute terms may still yield relatively higher relative errors for certain benchmarks. For example, benchmarks (e.g., GEMV) with inherently small runtime durations may exaggerate the relative error in MAPE, even if R^2 suggests a good fit overall.

In summary, L2P demonstrates strong predictive capabilities, as evidenced by the high R^2 values. However, challenges remain in achieving consistently low relative error rates across all benchmarks, as highlighted by the variability in MAPE. Notably, these results were achieved with a relatively small dataset of just 12K samples. Depending on the application, targeted model improvements are necessary to address specific benchmarks where predictive accuracy falls short, particularly for runtime predictions.

4 Cross-Application Precision Evaluation

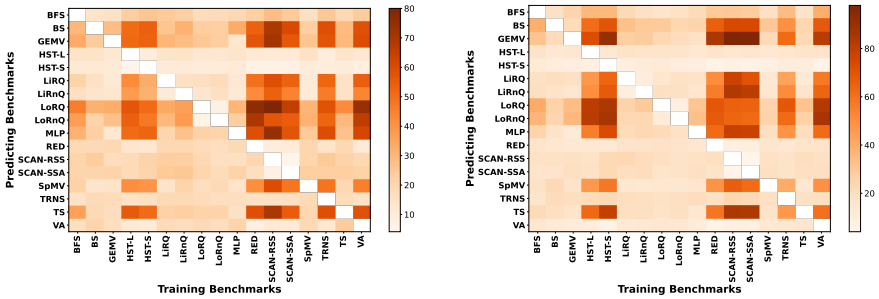
In this section, we present the experimental design used to assess the accuracy of the L2P model. We thoroughly evaluate its precision and robustness across various scenarios.

4.1 Accuracy Analysis

With a cross-application accuracy analysis, we demonstrate L2P's generalization capabilities when trained on a single source application, fine-tuning the model and predicting target metrics for structurally different target applications. These

structurally different applications can be variants of the same or completely different applications. In our experiments, this refers to the latter case.

Figure 5 illustrates MAPE values across various combinations of training and predicting benchmarks, highlighting the model's prediction accuracy in different scenarios. The results indicate that the benchmarks LiR-, LoR-, and MLP are particularly effective as training sources, showing robust performance when used to train models predicting the target metrics of various inference benchmarks. This suggests that they encapsulate a wider range of operational characteristics, allowing them to capture patterns applicable across different contexts effectively.



(a) Speedup (b) Time

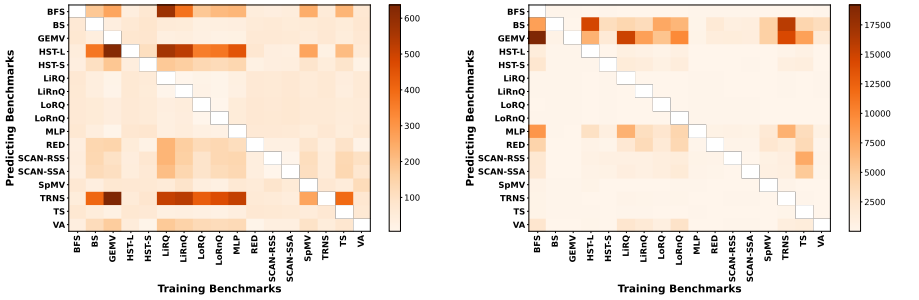
Fig. 5: Cross-application precision of the L2P model evaluated using MAPE.

Conversely, simple benchmarks like histogram (HST), scan (SCAN), and vector addition (VA) illustrate a different dynamic in their role as training models.

Despite being accurately predictable when acting as inference benchmarks, they do not demonstrate the same effectiveness when used for training purposes. This specificity indicates that while they can serve as reliable targets for predictions, their narrower performance spectrum limits their utility in training scenarios meant for generalization. Therefore, while models trained on diverse ML benchmarks can make accurate predictions for these simple workloads, the inverse is generally untrue; relying on these benchmarks as training sources may restrict model robustness and adaptability.

TL enables the model to generalize effectively across diverse applications. This technique allows the L2P model to leverage knowledge gained from one application to improve precision accuracy on different applications, reducing the need for an extensive task-specific training phase. Our experimental results demonstrate that TL is beneficial and essential for practical model deployment. Figure 6 illustrates that the L2P model without TL techniques severely diminishes precision. Specifically, MAPE increases by at least an order of magnitude when fine-tuning is omitted, rendering the model ineffective for real-world applications.

The significance of this precision gap underscores a fundamental principle in contemporary machine learning: pre-trained models require domain adaptation to achieve optimal performance in specialized contexts. This observation is con-



(a) Speedup (b) Time

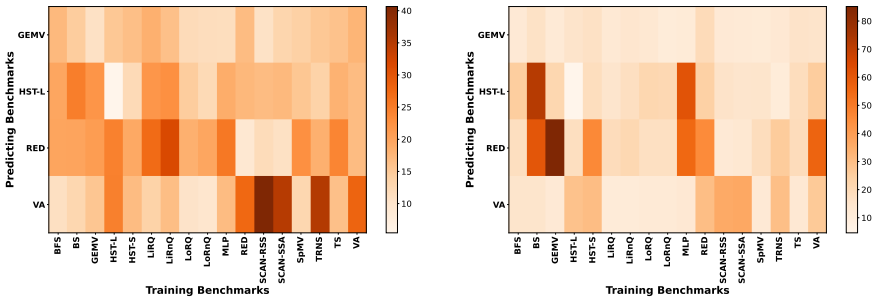
Fig. 6: Cross-application precision of the L2P model evaluated using MAPE, without TL.

sistent with findings across the broader field [19], where TL has become a standard practice for balancing computational efficiency and task-specific accuracy. The TL approach employed in our model enables efficient resource utilization (see Section 4.3), reducing the computational burden of training complex models from scratch for each application domain. Even jointly training our model on several benchmarks indicates that, without TL, model accuracy decreases by order of magnitudes. Indeed, although expanding the training set increases the model's generality, it does not necessarily lead to improved accuracy.

4.2 Robustness Analysis

In real-world environments, application variations arise from being written by different programmers, represented in different high-level programming languages, or compiled using distinct toolchains. Thus, evaluating the L2P model's predictive performance across applications that exhibit diverse structural modifications is important. To this end, we assess a set of code variations generated by the Cinnamon framework [12]. Figure 7 shows the L2P model's precision on automatically generated inference benchmarks.

The experimental results align with our findings in Section 4.1, reinforcing key observations regarding the L2P model's predictive performance characteristics. Runtime remains challenging and more significant than the ability to predict relative speedup across all benchmark categories. A noteworthy aspect is that the L2P model can leverage application structural variations to enhance its predictive accuracy. This phenomenon is particularly pronounced in the GEMV benchmark, where differences in code structure provide supplementary patterns that enrich the model's learning capabilities. Rather than hindering prediction, these structural differences serve as valuable training signals, ultimately enhancing the model's capacity for generalization. These results underscore L2P's proficiency in harnessing cross-application knowledge transfer, effectively utilizing features drawn from structurally diverse implementations of similar algorithms. This adaptability indicates that L2P accommodates application diversity and



(a) Speedup (b) Time

Fig. 7: Cross-application precision of the L2P model evaluated using MAPE on automatically generated benchmarks.

actively capitalizes on it to develop more robust internal representations of predictive performance characteristics.

4.3 Scalability

Scalability, vital for real-world applicability, is assessed by examining the L2P model's computational performance. We analyze training, fine-tuning, and inference times across varying dataset sizes to understand the computational efficiency and complexity of the L2P model. Figure 8 presents the wall clock time required for training across different benchmarks.

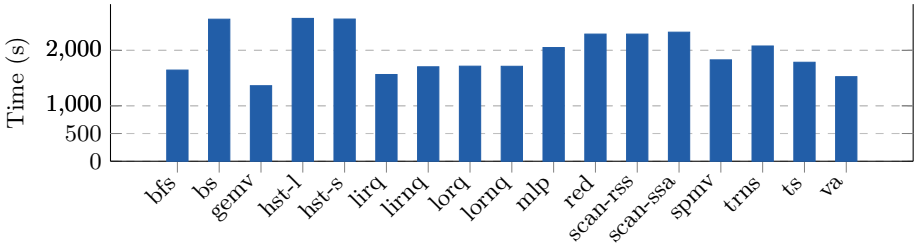


Fig. 8: Training time (wall clock) of the L2P Model.

The training wall clock times for the benchmarks exhibit notable variations, reflecting the differences in the computational demands of the datasets and the early stopping mechanism employed. The highest training time is observed for HST-L, followed closely by HST-S, while the lowest training time is recorded for GEMV. These discrepancies are attributed to the varying complexities and characteristics of the datasets in each benchmark. Benchmarks with more complex datasets require more iterations for convergence, leading to longer training durations, while more straightforward datasets converge faster.

It is crucial to note that the model uses an early stopping strategy to prevent overfitting, allowing training to terminate when no improvement is observed

over 200 epochs. This mechanism ensures the efficiency of the training process but also results in different stopping epochs for each benchmark, depending on the dataset's characteristics. For instance, benchmarks with smoother or less complex data may reach their optimal point sooner and stop after fewer epochs (e.g., BFS, GEMV, and TS), reducing the overall training time. Conversely, benchmarks with noise require more epochs before early stopping is triggered, increasing the training duration (e.g., BS, HST-L, and RED). In fact, the model's maximum limit of 1000 epochs was not reached for any experiments due to this early stopping policy.

Furthermore, the reported training times are for scenarios where only one benchmark is used. When combining datasets from multiple benchmarks, the training time scales linearly as the system adds the workload from each additional dataset. This scalability directly aligns with the proportional increase in the dataset size. Lastly, the observed times highlight the importance of considering dataset-specific characteristics when comparing benchmarks. Benchmarks such as MLP and TRNS fall within a mid-range compared to the extremes, suggesting balanced computational demands. However, benchmarks like SCAN-SSA show a higher demand, likely due to more complex interactions within the data. Such differences underline the need for careful workload balancing when extending training to broader datasets or combining benchmarks for comprehensive analysis.

Figure 9 shows the inference time, with two bars displayed per benchmark: the left bar represents the worst case, while the right bar represents the best case. Each bar is further divided into fine-tuning time and prediction time.

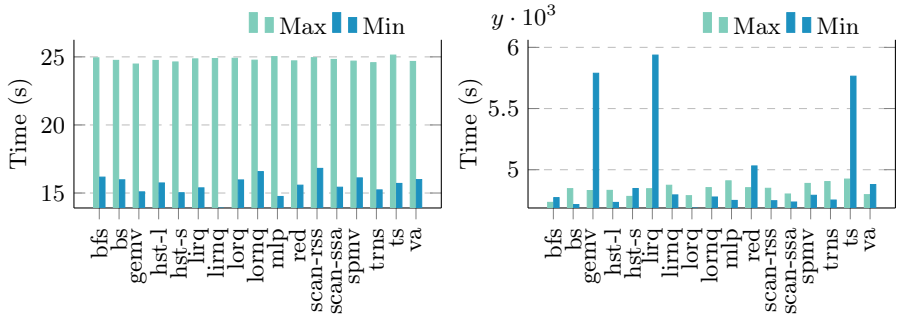


Fig. 9: Fine tuning (left) and prediction time (right) of the L2P Model.

A notable observation is that prediction times are significantly shorter than fine-tuning times. This trend is consistent across all benchmarks, revealing that the fine-tuning phase constitutes the bulk of the overall computational cost. While fine-tuning can take approximately 16 seconds to over 25 seconds, prediction times typically remain below 0.01 seconds. This stark contrast highlights the lightweight and efficient nature of predictions, in contrast to the time required for fine-tuning. Furthermore, even though fine-tuning durations may vary based on

specific configurations, prediction times remain remarkably stable and largely unaffected by such changes. The training time for a single benchmark ranges from 25.42 minutes to a maximum of 42.92 minutes. These durations underline the significant computational investment required for training, which would be compulsory in a non-TL model, as they far exceed any individual fine-tuning or prediction time.

5 Conclusions

We introduced the L2P model, a framework based on TL for predicting the performance on CNM systems. This model facilitates accurate predictions for new, previously unseen applications by leveraging knowledge from already analyzed workloads, thereby minimizing the need for extensive retraining. The L2P framework enhances its predictive capabilities by extracting static features from application binaries and supplementing them with hardware and compiler configuration data. Our research demonstrates strong predictive power for both runtime and speedup. Experimental analysis indicates that using more complex benchmarks during training results in broader coverage of code behaviors. This allows the model to adapt more effectively to new applications. Our results confirm that incorporating TL into performance models enables efficient, scalable, and accurate predictions for emerging CNM architectures. While we believe the model could be adopted for different CNM systems, the lack of reliable open-source evaluation infrastructures prevents its testing on additional architectures. We leave this exploration for future work.

Acknowledgment

This work is partially funded by the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D), and by the German Research Council (DFG) through the HetCIM project (502388442), and the CRC/TRR 404-Active 3D (528378584).

References

[1] Bai, Y., Zhao, W., Yin, S., Wang, Z., Yu, B.: Atformer: A learned performance model with transfer learning across devices for deep learning tensor programs. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. pp. 4102–4116 (2023)

[2] Boroumand, A., Ghose, S., et al.: Conda: Efficient cache coherence support for near-data accelerators. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). pp. 629–642 (2019)

[3] Damásio, T., Canesche, M., Pacheco, V., Botacin, M., Faustino da Silva, A., Quintão Pereira, F.M.: A game-based framework to compare program classifiers and evaders. In: Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. p. 108–121. CGO '23, Association for Computing Machinery, New York, NY, USA (2023)

[4] Devic, A., et al.: To pim or not for emerging general purpose processing in ddr memory systems. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 231–244. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022)
[5] Friesel, B., Lütke Dreimann, M., Spinczyk, O.: A full-system perspective on upmem performance. In: Proceedings of the 1st Workshop on Disruptive Memory Systems. pp. 1–7 (2023)
[6] Gibson, P., Cano, J.: Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. p. 28–39. PACT '22, Association for Computing Machinery, New York, NY, USA (2023)
[7] Gómez-Luna, J., El Hajj, I., Fernandez, I., Giannoula, C., Oliveira, G.F., Mutlu, O.: Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware. In: 2021 12th International Green and Sustainable Computing Conference (IGSC). pp. 1–7. IEEE (2021)
[8] Gómez-Luna, J., Hajj, I.E., Fernandez, et al.: Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. IEEE Access **10**, 52565–52608 (2022)
[9] Gómez-Luna, J., Guo, Y., Brocard, S., Legriel, J., Cimadomo, R., Oliveira, G.F., Singh, G., Mutlu, O.: Evaluating machine learning workloads on memory-centric computing systems. In: 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 35–49 (2023)
[10] He, M., et al.: Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 372–385. IEEE (2020)
[11] Hsieh, K., et al.: Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. ACM SIGARCH Computer Architecture News **44**(3), 204–216 (2016)
[12] Khan, A.A., Farzaneh, H., Friebel, K.F.A., Fournier, C., Chelini, L., Castrillon, J.: Cinm (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25). ASPLOS '25, Association for Computing Machinery (Mar 2025)
[13] Khan, A.A., Lima, J.P.C.D., Farzaneh, H., Castrillon, J.: The landscape of compute-near-memory and compute-in-memory: A research and commercial overview. arXiv (Jan 2024), https://arxiv.org/abs/2401.14428
[14] Lee, S., Kang, S.h., et al.: Hardware architecture and software stack for pim based on commercial dram technology: Industrial product. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). pp. 43–56 (2021)
[15] Li, S., Glova, A.O., Hu, X., Gu, P., Niu, D., Malladi, K.T., Zheng, H., Brennan, B., Xie, Y.: Scope: A stochastic computing engine for dram-based

in-situ accelerator. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 696–709. IEEE (2018)
[16] Liu, C., Baghdadi, R.: Data-efficient performance modeling via pre-training. In: Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction. pp. 48–59 (2025)
[17] Mishra, A., Chheda, S., Soto, C., Malik, A.M., Lin, M., Chapman, B.: Compoff: A compiler cost model using machine learning to predict the cost of openmp offloading. In: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 391–400 (2022)
[18] Niu, D., Li, S., Wang, Y., Han, W., Zhang, Z., Guan, Y., Guan, T., Sun, F., Xue, F., Duan, L., et al.: 184qps/w 64mb/mm 2 3d logic-to-dram hybrid bonding with process-near-memory engine for recommendation system. In: 2022 IEEE International Solid-State Circuits Conference (ISSCC). vol. 65, pp. 1–3. IEEE (2022)
[19] Sasaki, Y., Takahashi, K., Shimomura, Y., Takizawa, H.: A cost model for compilers based on transfer learning. In: Proccedings of the International Parallel and Distributed Processing Symposium Workshops. pp. 942–951 (2022)
[20] Faustino da Silva, A., Castrillon, J., Pereira, F.M.Q.a.: A comparative study on the accuracy and the speed of static and dynamic program classifiers. In: Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction. p. 13–24. CC '25, Association for Computing Machinery, New York, NY, USA (2025)
[21] Singh, G., Chelini, L., et al.: A review of near-memory computing architectures: Opportunities and challenges. In: 2018 21st Euromicro Conference on Digital System Design (DSD). pp. 608–617. IEEE (2018)
[22] Trümper, L., Ben-Nun, T., Schaad, P., Calotoiu, A., Hoefler, T.: Performance embeddings: A similarity-based transfer tuning approach to performance optimization. In: Proceedings of the International Conference on Supercomputing. p. 50–62. Association for Computing Machinery, New York, NY, USA (2023)
[23] Upmem: Upmem processing in-memory (pim): Ultra-efficient acceleration for data-intensive applications. In: 2022 UPMEM PIM Tech paper v2.7. pp. 1–22 (2022)