2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00082

# Attention, Distillation, and Tabularization: Towards Practical Neural Network-Based Prefetching

Pengmiao Zhang*
*University of Southern California*
Los Angeles, USA
pengmiao@usc.edu

Neelesh Gupta*
*University of Southern California*
Los Angeles, USA
neeleshg@usc.edu

Rajgopal Kannan
*DEVCOM Army Research Lab*
Los Angeles, USA
rajgopal.kannan.civ@army.mil

Viktor K. Prasanna
*University of Southern California*
Los Angeles, USA
prasanna@usc.edu

***Abstract*—Attention-based Neural Networks (NN) have demonstrated their effectiveness in accurate memory access prediction, an essential step in data prefetching. However, the substantial computational overheads associated with these models result in high inference latency, limiting their feasibility as practical prefetchers. To close the gap, we propose a new approach based on *tabularization* that significantly reduces model complexity and inference latency without sacrificing prediction accuracy. Our novel tabularization methodology takes input as a distilled, yet highly accurate attention-based model for memory access prediction and efficiently converts its expensive matrix multiplications into a hierarchy of fast table lookups. As an exemplar of the above approach, we develop DART, a prefetcher comprised of a simple hierarchy of tables. With a modest 0.09 drop in F1-score, DART reduces 99.99% of arithmetic operations from the original attention-based model and 91.83% from the distilled model. DART accelerates the large model inference by 170× and the distilled model by 9.4×. DART has comparable latency and storage costs as state-of-the-art rule-based prefetcher BO but surpasses it by 6.1% in IPC improvement. DART outperforms state-of-the-art NN-based prefetchers TransFetch by 33.1% and Voyager by 37.2% in terms of IPC improvement, primarily due to its low prefetching latency.**

***Index Terms*—memory access prediction, attention, neural network, knowledge distillation, tabularization, prefetching**

## I. Introduction

Data prefetching is a fundamental technique used in modern computing systems to bridge the latency gap between CPU cores and memory subsystems, thereby reducing overall program execution time [1], [2]. A data prefetcher works by predicting which data will be needed in the near future and fetching it from memory in advance. This helps the processor avoid waiting for the data to be fetched from memory, significantly improving performance [3].

Existing table-based prefetchers have offered practical solutions using simple and fast table look-ups, relying on heuristic rules, such as spatial and temporal locality, to anticipate future memory accesses [4], [5]. Best-Offset prefetcher (BO) [6] employs a recent request table to record memory accesses and triggers prefetch requests. It updates scores of a list of offsets within a spatial range and predicts the offset with the highest score. Irregular Stream Buffer (ISB) [7] devises table-based structures to track the most recent addresses and their corresponding program counters (PCs). Nonetheless, hampered by their reliance on heuristic and pre-established rules, these prefetchers exhibit limited adaptability and generalizability. They struggle to effectively discern intricate and latent patterns [8]–[10].

On the other hand, machine learning algorithms, especially Neural Networks (NN), have showcased remarkable accuracy in predicting memory accesses. LSTM (Long Short-Term Memory) [11] is widely-used for memory access prediction [12]–[17] due to its proficiency in sequence modeling, but its recurrent architecture poses challenges for parallelization, resulting in high inference latency and low practicality. Attention-based models [18] have demonstrated success in memory access prediction, achieving state-of-the-art performance and showing potential for practical hardware prefetcher deployment due to their high parallelism [9], [10], [19].

The central challenge in deploying NN-based prefetchers lies in their significant computational demands. Firstly, the quantity of parameters in well-performing NN models impose tremendous system resource requirements, potentially demanding more resources than the application that it was set out to prefetch for. Secondly, these models' complex nature leads to slow inference, causing untimely data prefetching. Lastly, the sheer volume of arithmetic operations inherent to NNs consumes both substantial resources and energy. Though existing works use pruning [20], quantization [21], and hardware parallel implementations [22] to accelerate NN inference, these techniques still require a large number of matrix multiplications during inference.

In light of this challenge, motivated by the success of table-based prefetchers, we propose a novel approach that transfers knowledge from a large attention-based neural network to a compact hierarchy of tables for more practical NN-based data prefetching, eliminating all matrix multiplications in model

* These authors contributed equally.

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00082

inference. The key steps of the proposed approach include: 1) training of a large attention-based model for memory access prediction, 2) reducing model complexity via knowledge distillation, and 3) implementing *tabularization* to transform the distilled model into tables.

In developing the above approach, we tackle several key sub-problems that emerge. 1) How to map layers of neural networks with various operations to table lookups? We design kernels that not only store the precomputed matrix multiplications, but also manage challenges including bias incorporation, table size constraints, and the integration of activation functions between operations. 2) How to reduce the critical path of a model to satisfy a given latency constraint? We quantitatively analyze the table-based model's inference latency and compress the NN model using a novel knowledge distillation approach for multi-label classification. 3) How to address the error accumulation when mapping multiple NN layers to tables? We propose a novel fine-tuning approach that trains the table to imitate the NN layer output rather than merely approximating dot products between layer inputs and weights. In this way, we mitigate performance degradation due to the approximation as the number of layers increases.

We develop a prefetcher DART (Distilling Attention-based neuRal network to Tables) as an exemplar of this approach. The predictor in DART is simple and elegant: a hierarchy of tables acquired from the above steps. DART couples the practicality inherent in table-based prefetchers with the precision offered by NN predictors. We summarize our main contributions as follows:

- We propose a novel approach to transfer knowledge from an attention-based NN to a hierarchy of tables for the practical implementation of an NN-based prefetcher.
- We design tabularization kernels for mapping the key operations in an attention-based NN to table look-ups. Using the kernels, we can convert an arbitrary attention-based NN to a set of hierarchical tables.
- We develop a prefetcher DART to exemplify our approach. We develop a table configurator for DART to meet prefetcher design constraints.
- We propose a novel layer fine-tuning algorithm to mitigate the error accumulation problem when mapping more layers to tables.
- We evaluate the prediction performance of DART. With a 0.09 drop of F1-score, DART reduces 99.99% of arithmetic operations from the large attention-based model and 91.83% from the distilled model, accelerating the two models by 170× and 9.4×, respectively.
- We evaluate the prefetching performance of DART on a variety of workloads from SPEC 2006 and SPEC 2017. DART achieves a 37.6% IPC improvement, outperforming state-of-the-art rule-based prefetcher BO by 6.1% with comparable latency and storage costs. It also surpasses state-of-the-art NN-based prefetchers TransFetch and Voyager by 33.1% and 37.2%, respectively.

## II. Background

### A. Attention

The attention mechanism excels in memory access prediction due to its high accuracy, adaptability, and parallelizability [9], [18]. An attention-based model comprises two main components: a feed-forward network based on linear operations and multi-head self-attention based on dot-product attention. By crafting kernels to convert the fundamental linear and attention operations into table look-ups, we can tabularize any given attention-based model using the kernels.

**Feed-Forward Network (FFN).** An FFN is a stack of two linear operations along with a non-linear activation function:

$$\text{Linear}\left(\boldsymbol{X}\right) = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{B} \tag{1}$$

$$\text{FFN}(\boldsymbol{X}) = \text{Linear}_{\text{O}}\left(\max\left(0, \text{Linear}_{\text{H}}\left(\boldsymbol{X}\right)\right)\right) \tag{2}$$

where input $\mathbf{X} \in \mathbb{R}^{D_I \times T}$, weight $\boldsymbol{W} \in \mathbb{R}^{D_O \times D_I}$, and bias $\boldsymbol{B} \in \mathbb{R}^{D_O \times T}$. $T$, $D_I$, and $D_O$ are the input sequence length, input dimension, and output dimension, respectively. $\text{Linear}_{\text{H}}$ and $\text{Linear}_{\text{O}}$ are the hidden and output linear layers.

**Multi-Headed Self-Attention (MSA).** Given three distinct matrices: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{T \times D_k}$ the scaled dot-product attention is defined as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{D_k}}\right)\boldsymbol{V} \tag{3}$$

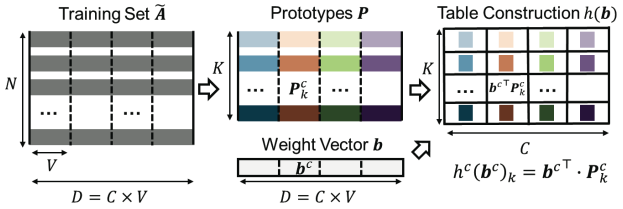
where $D_k$ is the dimension of $\boldsymbol{K}$. Consider the Attention input matrices are projected from the same input matrix though projection matrices $\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K, \boldsymbol{W}_i^V \in \mathbb{R}^{D \times D_h}$, $h$ attention operations are concatenated as multiple heads, MSA is:

$$\begin{aligned} \text{MSA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{Concat}\left(\text{head}_1, \ldots, \text{head}_{\text{h}}\right)\boldsymbol{W}^O \\ \text{head}_{\text{i}} &= \text{Attention}\left(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V\right) \end{aligned} \tag{4}$$

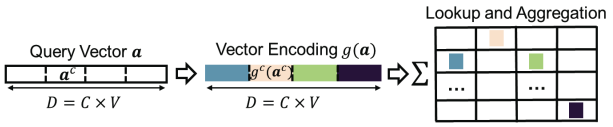
where $\boldsymbol{W}^O \in \mathbb{R}^{hD_v \times D}$ is the output layer weights that combines the results from each head, $i$ is the index for each head, $D$ is the hidden dimension, $D_v$ is the dimension of $\boldsymbol{V}$, and $D_h = D/h$ is the dimension of each head.

### B. Product Quantization

Our tabularization approach is based on Product Quantization (PQ) [23], a classic algorithm for approximating and accelerating vectors inner products through quantization and precomputation. Given a vector $\boldsymbol{a} \in \mathbb{R}^D$ that is drawn from a training set $\tilde{\boldsymbol{A}} \in \mathbb{R}^{N \times D}$ with $N$ samples, and a fixed weight vector $\boldsymbol{b} \in \mathbb{R}^D$, PQ generates a quantized approximation $\hat{\boldsymbol{a}}$ such that $\hat{\boldsymbol{a}}^\top \boldsymbol{b} \approx \boldsymbol{a}^\top \boldsymbol{b}$. To quantize $\boldsymbol{a}$, the $D$-dimension vector is split to $C$ subspaces, each of dimension $V$. Within each subspace, $K$ prototypes are learned as the quantized subvectors for the subspace. Since $\hat{\boldsymbol{a}}$ is quantized and $\boldsymbol{b}$ is fixed, $\hat{\boldsymbol{a}}^\top \boldsymbol{b}$ can be precomputed and reused in a query. Figure 1 shows an overview of PQ and the detailed process is introduced below.




(a) For product quantization training, prototypes within each subspace are learned, and dot products are precomputed to store in a table.




(b) For product quantization query, the input vector is encoded to find its nearest prototype index, retrieving it from the constructed table to aggregate and yield the final result.

Fig. 1: Training and query of product quantization.

*1) Training:* PQ training includes two main steps: learning the prototypes for each subsection and constructing the table which stores the precomputed results.

**Prototype Learning ($p$).** Let $\tilde{\boldsymbol{A}}^c \in \mathbb{R}^{N \times V}$ be subvectors in the $c$-th subspace from the training set $\tilde{\boldsymbol{A}}$, the goal of this step is to learn $K$ prototypes $\boldsymbol{P}_k^c$ by minimizing the distance between each sub-vector $\tilde{\boldsymbol{A}}^c$ and its nearest corresponding prototype $\boldsymbol{P}_k^c$ where $k$ signifies the indices of the prototypes within subspace $c$. The process is formulated in Equation 5.

$$p^c(\tilde{\boldsymbol{A}}) \triangleq \arg\min_P \sum_c \sum_i \left\| \tilde{\boldsymbol{A}}_i^c - \boldsymbol{P}_k^c \right\|^2 \tag{5}$$

**Table Construction ($h$).** Subsequently, a table is generated by computing and storing the dot product of each subspace's prototypes $\boldsymbol{P}_k^c$ and the weight vector $\boldsymbol{b}^c$ (the $c$-th subspace of $\boldsymbol{b}$). As detailed in Equation 6, the function $h^c(\boldsymbol{b})_k$ describes the entry that resides at index $(c, k)$ in the table.

$$h^c(\boldsymbol{b})_k \triangleq \boldsymbol{b}^{c\top} \cdot \boldsymbol{P}_k^c \tag{6}$$

*2) Query:* The query process avoids the multiplication operations in dot product by encoding the query vector to the closest prototype, looking up the precomputed results from the table, and perform aggregation.

**Vector Encoding ($g$).** Given the query vector $\boldsymbol{a}$, $g^c(\boldsymbol{a})$ identifies the closest prototype $\boldsymbol{P}_k^c$ in each subspace $c$ by finding index $k$ minimizing distance to $\boldsymbol{a}^c$, as in Equation 7. The outcome is a set of indices that represent the encoding of $\boldsymbol{a}$ using prototypes.

$$g^c(\boldsymbol{a}) \triangleq \arg\min_k \|\boldsymbol{a}^c - \boldsymbol{P}_k^c\|^2 \tag{7}$$

**Table Lookup and Aggregation.** The dot products $\boldsymbol{a}^\top \boldsymbol{b}$ can be approximated by looking up the precomputed values through the encoded indices, and aggregating the subspaces through an aggregation function $f(\cdot)$, as shown in Equation 8.

$$f(\boldsymbol{a}, \boldsymbol{b}) = \sum_c h^c(\boldsymbol{b})_k, k = g^c(\boldsymbol{a}) \tag{8}$$

The query avoids the dot product operation in $\boldsymbol{a}^\top \boldsymbol{b}$ by approximating the result through table look-ups. We use locality sensitive hashing [24] for encoding and use parallel summation for aggregation, the complexity is significantly lower than the dot product, especially for large dimension vectors.

## III. Related Works

### A. ML for Data Prefetching

Although hardware prefetching is well-researched, applying Machine Learning (ML) has seen a surge due to stagnation in existing approaches [25], [26]. Prior works have exhausted Long Short-Term Memory (LSTM) models as initial gauges to assess the potential of ML-based prefetching improvements [12], [13], [17]. Recent ML-based prefetchers exploit attention mechanisms to emphasize salient patterns, setting unparalleled benchmarks in prediction performance [9], [16], [27], [28]. Other ML-based prefetching optimizations have tested logistic regression, decision trees, random forests, and reinforcement learning for both cache-level memory access prediction and the selection of existing hardware prefetchers [8], [29]–[32]. Despite high performance in memory access prediction, previous approaches neglect the practicalities essential for real-world deployment of ML-based prefetchers, most notably latency and storage considerations [33].

### B. Neural Network Acceleration and Approximation

Various techniques have been explored to reduce Neural Network (NN) complexity and accelerate inference. Model compression techniques, such as parameter pruning [20], [34], quantization [21], [35], low-rank factorization [36], and knowledge distillation [37]–[39], target redundancy and model size to boost performance. In hardware, TPUs with systolic arrays allow for tensor processing demands at scale [40], and FPGAs offer parallel acceleration tailored for CNNs and GNNs [41], [42]. While these methods reduce model complexity or accelerate computation, they still require matrix multiplications during inference. Approximate Matrix Multiplication techniques replace matrix multiplications using hashing and averaging [24], [43], while shift-and-add strategies aim to render NNs multiplication-free, optimizing power efficiency [44], [45]. Current methods either concentrate on the final NN layer or simply alter matrix multiplication. In contrast, our paper introduces a novel approach that transforms the entire NN into a customizable table hierarchy while maintaining performance. To the best of our knowledge, our work represents the first instance of a tabularized NN-based prefetcher with a comprehensive design methodology and training scheme.

## IV. Overview

Our objective is to bridge the gap between traditional table-based prefetchers and NN-based prefetchers by reconfiguring the neural network inference process into a rapid and efficient table look-up mechanism, eliminating the need for matrix multiplications during model inference.

### A. Problem Definition

Given the sequences of $T$ historical physical block addresses $\boldsymbol{A}_t = \{a_1, a_2, ..., a_T\}$ and historical program counters $\boldsymbol{PC}_t = \{pc_1, pc_2, ..., pc_T\}$ at time $t$, the input for the prediction models is denoted as $\boldsymbol{X}_t = \{(a_1, pc_1), (a_2, pc_2)..., (a_T, pc_T)\}$. The future $K$ block addresses are represented by $\boldsymbol{Y}_t = \{y_1, y_2, ..., y_K\}$. Prefetcher design constraints include a latency constraint $\tau$ and a storage constraint $s$. Our objective is to construct a tabular model $\mathcal{T}(\boldsymbol{X}_t; \boldsymbol{\theta})$ that learns the mapping from $\boldsymbol{X}_t$ to $\boldsymbol{Y}_t$, such that:

$$\mathcal{L}(\mathcal{T}) < \tau, \mathcal{S}(\mathcal{T}) < s \tag{9}$$

where $\mathcal{L}(\cdot)$ represents the model inference latency and $\mathcal{S}(\cdot)$ represents the model storage cost.

### B. Overall Approach

We develop a three-fold approach to solve this problem as illustrated in Figure 2. The workflow consists of a preprocessing step and three key training steps: 1) **Attention**: Training a high-performance prediction model, 2) **Distillation**: Reducing model complexity to meet the prefetcher design constraints, and 3) **Tabularization**: Converting the model to a hierarchy of tables to achieve fast model inference.

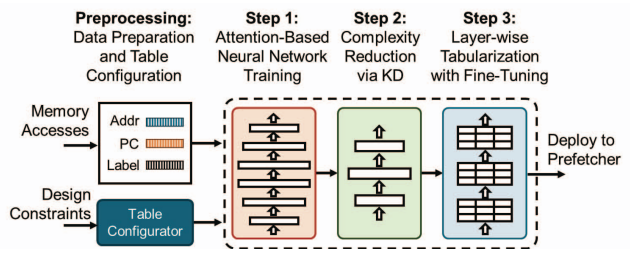



Fig. 2: Approach to constructing compact table-based memory access predictor by distilling knowledge from a trained attention-based neural network.

**Preprocessing: Data Preparation and Table Configurator.** To train the model, we first preprocess the data and set the configurations for both the neural network and table-based models based on design constraints $\tau$ and $s$. These constraints directly impact the table-based predictor's latency and storage, contingent on the NN's layers and dimension size. Using these parameters, a table configurator formulates the optimal attention-based and table-based model structures.

**Step 1: Attention-based Neural Network Training.** With the prepossessed data and labels, we train a large and robust attention-based multi-label memory access prediction model $\hat{\boldsymbol{Y}}_t = \mathcal{M}(\boldsymbol{X}_t; \boldsymbol{\theta})$. The aim of this step is to pursue exceptional prediction performance, regardless of any limitations imposed by design constraints or model complexity.

**Step 2: Complexity Reduction via Knowledge Distillation.** The well-trained model $\mathcal{M}$ is large and typically cannot meet the prefetcher design constraints. Therefore, we need to reduce the model complexity by reducing the number of layers and dimensions. The adaptability of Knowledge Distillation (KD) [37] allows us to tailor the target model to meet specific constraints. Using the well-trained large model $\mathcal{M}$ as the teacher network, we employ KD to train a shallower and more compact attention-based student network $\mathcal{M}_s$. Then, we adopt the predictor configuration generated by the table configurator to ensure the network can map to a valid table-based predictor that meets the prefetcher design constraints.

**Step 3: Layer-wise Tabularization with Fine-Tuning.** To make our NN-based prefetcher practical, we transform the KD compressed model $\mathcal{M}_s$ into a table-based structure $\mathcal{T}$ for quick look-up-based inferences. This is dubbed *Tabularization*. We have crafted specialized kernels for linear and attention functions to facilitate this process for an arbitrary attention-based NN. With these kernels, we systematically convert our distilled model into organized tables, all while considering predefined design constraints. However, as the process progresses, cumulative errors between subsequent layers can worsen. To counteract this, we employ a layer fine-tuning method to adjust weights in light of the preceding tabular layer outcomes.

### C. DART

As an exemplar of our proposed approach, we introduce DART, a prefetcher equipped with a table-based predictor derived through distilling knowledge from an attention-based memory access prediction model, as is shown in Figure 3. DART is integrated into the last level cache (LLC), uses LLC memory accesses as input, predicts future memory accesses, and request prefetch to LLC. The prediction process is mainly table look-ups layer by layer with a minimal number of arithmetic operations. The model inference is fast and the model implementation is simple. The training of DART follows the proposed approach, detailed in Section VI.

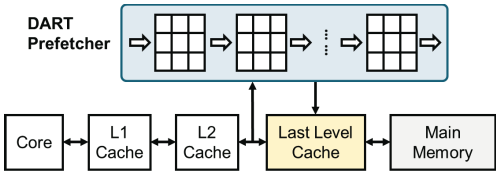



Fig. 3: Using the proposed approach, we present our LLC prefetcher DART with a table-based predictor distilled from an attention-based memory access prediction model.

## V. Tabularization Kernels

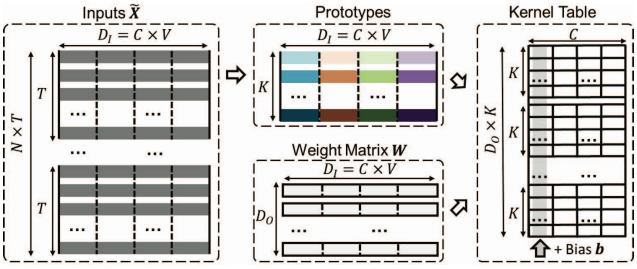
We design tabularization kernels to convert the key operations in an attention-based NN into table look-ups. These operations consist of linear operations and attention operations (Section II-A). By designing these kernels, the inference of an arbitrary attention-based NN can be mapped to table look-ups.

The kernels are based on the technique of product quantization (PQ) in Section II-B, eliminating expensive matrix multiplication operations. Furthermore, we propose three optimizations for attention-based NN tabularization. First, to accommodate the $T$-length input sequence, we extend the PQ

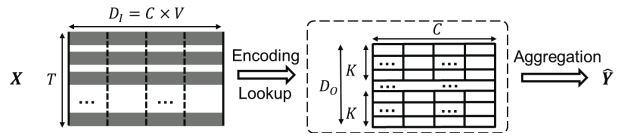
approach from two vectors to the product of matrices. Second, to further condense NN operations, we merge the bias addition operation and activation functions to the table construction and avoid these steps in the query. Third, to tabularize the attention operation without a fixed weight matrix, we generalize the approach to accommodate two variable matrix multiplications. We outline the training and querying processes for the kernels.

### A. Linear Kernel

A linear operation in a NN is shown as in Equation 1. Assume the training set is $\tilde{\boldsymbol{X}} \in \mathbb{R}^{N \times T \times D_I}$ with $N$ number of samples. Bias $\boldsymbol{B}$ in Equation 1 is a $T$ repeat of bias vector $\boldsymbol{b}$ adding to all time-step (sequence length) outputs. We define the number of subspaces as $C$ and number of prototypes within a subspace as $K$ for tabularization. Figure 4 illustrates the linear kernel training and query process.




(a) For linear kernel training, prototypes are learned from row vectors across samples and sequences, adding bias as a column to trim query operations.




(b) For linear kernel query, the input vector is encoded and looked up in parallel across T dimensions.

Fig. 4: Training and query of the linear kernel.

*1) Training:* Figure 4a illustrates the kernel training process. We reshape $\tilde{\boldsymbol{X}}$ from $\mathbb{R}^{N \times T \times D_I}$ to $\tilde{\boldsymbol{X}}_r \in \mathbb{R}^{NT \times D_I}$ to learn prototypes of dimension $D_I$. For each prototype $\boldsymbol{P}_k^c$ in subspace $c$, a dot product between linear layer weights $\boldsymbol{W}_o^c$ is computed and stored for reuse. The outcome populates table element $h_o^c(\boldsymbol{W})_k$. To merge bias addition, we reshape the bias to match the constructed table $\boldsymbol{b}_r \in \mathbb{R}^{D_O \times K \times C}$, repeating values in each $Do$ dimension $K$ times for the first $C$ subspace column and filling the rest with zeroes. $\boldsymbol{b}_r$ is then added to the learned table, equivalent to integrating the bias $\boldsymbol{b}$ into a single subspace dimension. Thus, during query aggregation, the bias will be automatically added to the final result. The formal expression for the table construction is:

$$h_o^c(\boldsymbol{W})_k = \boldsymbol{W}_o^{c\top} \cdot p^c(\tilde{\boldsymbol{X}}_r)_k + \boldsymbol{b}_r \tag{10}$$

*2) Query:* The query of a single row of $\boldsymbol{X}$ is similar to the basic PQ query. All $T$ row vectors in $\boldsymbol{X}$ are encoded and generate results for all $D_O$ output dimensions. These encoding, table look-ups, and aggregations are independent so they are embarrassingly parallel. By aggregating the results over $C$ subspaces, the output $\hat{\boldsymbol{Y}} \in \mathbb{R}^{T \times D_O}$ is:

$$\hat{\boldsymbol{Y}}_{t,o} = \sum_c h_o^c(\boldsymbol{W})_k, k = g^c(\boldsymbol{X}_t) \tag{11}$$

### B. Attention Kernel

In Equation 3, we show the attention mechanism. Compared to linear operations, tabularizing attention processes proves more challenging. Firstly, the absence of a fixed weight matrix renders the precompute processing in linear kernels impossible. To counter this hurdle, we tabularize the pair-wise dot products. Secondly, attention operations requires three matrix multiplications, which can potentially balloon the table depth to $K^3$ for $K$ prototypes, leading to high storage consumption. Our solution involves a secondary quantization of the intermediate result to trim the table depth down to $2K^2$. Lastly, beyond mere matrix multiplication, operations such as scaling and Softmax activation introduce added steps and latency. We have streamlined this process by integrating the operations into prototypes in training, removing need for such operations during a query.

*1) Training:* Figure 5a shows the kernel training process to construct tables for attention operation. First, We train $K$ prototypes of the training data set $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{N \times T \times D_k}$ and $\tilde{\boldsymbol{K}} \in \mathbb{R}^{N \times T \times D_k}$, where $N$ is the number of samples, $T$ is the sequence length, and $D_k$ is the dimension size. Within each subspace of $C_k$, we compute and store the pair-wise dot product of the Q prototypes and K prototypes, generating a $K^2$ depth table with $C_k$ width, denoted as QK table. Second, based on the training set $\tilde{\boldsymbol{Q}}$ and $\tilde{\boldsymbol{K}}$ and the trained QK table, we can generate the corresponding $\boldsymbol{Q}\tilde{\boldsymbol{K}}^\top \in \mathbb{R}^{N \times T \times T}$. We perform a second quantization process to $\boldsymbol{Q}\tilde{\boldsymbol{K}}^\top$ by training its $K$ prototypes. Third, we directly process the scaling of $\sqrt{D_k}$ and Softmax activation on the prototypes, then use this processed prototype to perform pair-wise dot product with the other input of attention $\tilde{\boldsymbol{V}} \in \mathbb{R}^{N \times T \times D_k}$, generating a table at depth $K^2$ with width $C_t$ (the number of subspaces for $T$ dimension), denoted as QKV table. In summary, through two quantization steps and two pair-wise dot produce, the tabularization output is two tables: QK table at size $K^2 \times C_k$ and a QKV table at size $K^2 \times C_t$. The tabularization process of QK table is:

$$h^c(\tilde{\boldsymbol{Q}}, \tilde{\boldsymbol{K}})_{i,j} = p^c(\tilde{\boldsymbol{Q}}_r)_i \cdot p^c(\tilde{\boldsymbol{K}}_r)_j \tag{12}$$

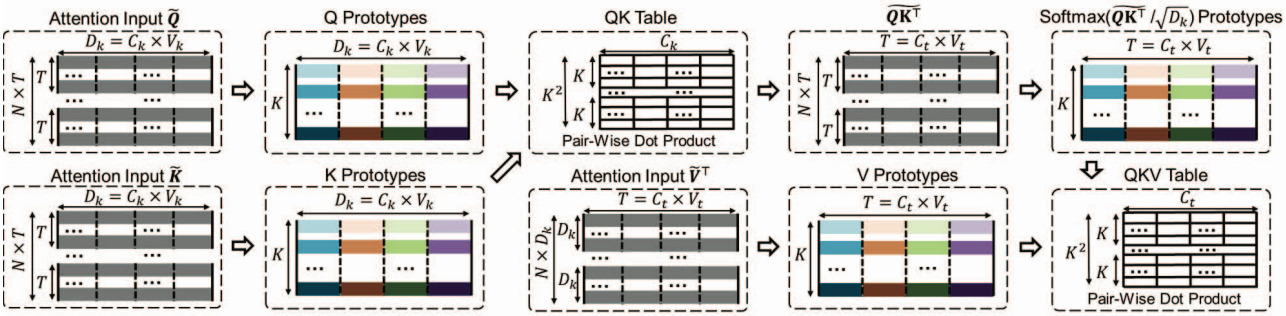
where $\tilde{\boldsymbol{Q}}_r$ and $\tilde{\boldsymbol{K}}_r$ are reshaped input matrix with size $\mathbb{R}^{NT \times D_k}$, $i$ and $j$ are the index of prototypes. The output of the QK table through aggregation is as below:

$$\boldsymbol{Q}\tilde{\boldsymbol{K}}^\top{}_t = \sum_c h^c(\tilde{\boldsymbol{Q}}_t, \tilde{\boldsymbol{K}}_t)_{i,j}, i = g^c(\tilde{\boldsymbol{Q}}_t), j = g^c(\tilde{\boldsymbol{K}}_t) \tag{13}$$

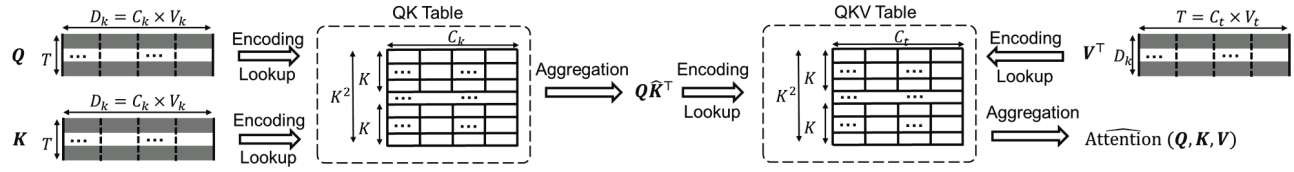
The generated results from QK table along with the reshaped and transposed the attention input $\tilde{\boldsymbol{V}}^\top \in_r \mathbb{R}^{ND_k \times T}$ performs pair-wise dot product and construct the table for lookup:

$$h^c(\boldsymbol{Q}\tilde{\boldsymbol{K}}^\top, \tilde{\boldsymbol{V}})_{i,j} = \text{Sigmoid}(p^c(\boldsymbol{Q}\tilde{\boldsymbol{K}}^\top)_i / \sqrt{D_k}) \cdot p^c(\tilde{\boldsymbol{V}}_r)_j \tag{14}$$

The two constructed tables are stored for query while the learned prototypes are not.




(a) For attention kernel training, pairwise products of learned prototypes are stored for query reuse and doubly quantized to limit table size, integrating Softmax and scaling directly into the QKV table result.




(b) For attention kernel query, matrix multiplication, scaling, and activation are replaced by two lookups: from $\boldsymbol{Q}$, $\boldsymbol{K}$ to $\boldsymbol{Q}\hat{\boldsymbol{K}}^\top$, and $\boldsymbol{Q}\hat{\boldsymbol{K}}^\top$, $\boldsymbol{V}$ to the result.

Fig. 5: Training and query of the attention kernel.

*2) Query:* The query process of attention kernel is shown in Figure 5b. The query process consists of two critical steps of table look-ups. First, the input matrices $\boldsymbol{Q}$ and $\boldsymbol{K}$ are encoded and looked up for the dot product results in the QK table as in Equation 13 and get estimated by $\boldsymbol{Q}\hat{\boldsymbol{K}}^\top$. Then $\boldsymbol{Q}\hat{\boldsymbol{K}}^\top$ is encoded again and along with the encoded input $\boldsymbol{V}$ is looked up from the QKV table and aggregated for the final estimation of attention operation as below:

$$\hat{\boldsymbol{Y}}_t = \sum_c h^c(\boldsymbol{Q}\hat{\boldsymbol{K}}^\top, \boldsymbol{V})_{i,j}, i = g^c(\boldsymbol{Q}\hat{\boldsymbol{K}}^\top{}_t), j = g^c(\boldsymbol{V}_t) \tag{15}$$

The query of attention kernel avoids all matrix multiplications, scaling calculations, and activation operations.

### C. Kernel Complexity

We analyze the kernel complexity by thoroughly examining the latency, storage, and the number of arithmetic operations associated with the proposed kernels. These factors will be instrumental in shaping the overall model design to align with prefetcher constraints.

*1) Latency ($\mathcal{L}$):* We assume fully parallel implementation. We use the locality sensitive hashing in [24] as the encoding function $g$, the latency is $\log(K)$ for $K$ prototypes.

**Linear kernel latency.** The linear kernel latency consists of the latency for encoding subvectors $g$, the latency for table lookup $h$ and the latency for subspace aggregation $f$:

$$\mathcal{L}_l(K, C) = \mathcal{L}_g + \mathcal{L}_f + \mathcal{L}_h = \log(K) + \log(C) + 1 \tag{16}$$

**Attention kernel latency.** The attention kernel latency consists of the latency of the input encodings $g$ for $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$, the aggregation for QK Table, the encoding $g_{qk}$ for the approximated $\boldsymbol{Q}\boldsymbol{K}^\top$, and the final aggregation $f$:

$$\begin{aligned}\mathcal{L}_a(K, C) &= \mathcal{L}_{g_i, g_{qk}} + \mathcal{L}_{f_{qk}, f_{qkv}} + \mathcal{L}_{h_{qk}, h_{qkv}} \\ &= 2\log(K) + \log(C_k) + \log(C_t) + 2 \\ &\Rightarrow 2(\log(K) + \log(C) + 1), \text{ if } C = C_k = C_t\end{aligned} \tag{17}$$

*2) Storage ($\mathcal{S}$):* The storage cost of a kernel consists of the table entries and the vector encoding results. One index of the encoded prototypes takes storage cost at $log(K)$ bits. For the precomputed table entries, we denote the data bit-length as $d$. The prototypes do not need to be stored since we use the encoded indices to look-up from the table directly.

**Linear kernel storage.** Encoding of the input subvectors $g$ assigns them to $TC$ indices of prototypes. The total number of table entry is $D_O KC$ as shown in Figure 4 The linear kernel storage in bit is:

$$\mathcal{S}_l^d(T, D_O, K, C) = \mathcal{S}_g + \mathcal{S}_h^d = TC\log(K) + D_O KCd \tag{18}$$

**Attention kernel storage.** As shown in Figure 5, there are four encoding operations and two tables, the storage cost is:

$$\begin{aligned}\mathcal{S}_a^d(T, D_k, K, C) &= \mathcal{S}_{g_q, g_k, g_{qk}, g_v} + \mathcal{S}^d_{h_{qk}, h_{qkv}} \\ &= (2TC_k + TC_t + D_k C_t)\log(K) + K^2(C_k + C_t)d \\ &\Rightarrow (3T + D_k)C\log(K) + 2K^2 Cd, \text{ if } C = C_k = C_t\end{aligned} \tag{19}$$

where $g_q, g_k, g_v, g_{qk}$ is for the encoding of attention input matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ and intermediate result $\boldsymbol{Q}\boldsymbol{K}^\top$. $h_{qk}$ and $h_{qkv}$ are the QK table and the QKV table in Figure 5.

*3) Arithmetic Operations ($\mathcal{A}$):* We analyze the arithmetic operations for the kernels besides table look-ups.

**Linear kernel arithmetic operations.** The operations consists of two parts, the encoding $g$ to get table indexes and the aggregation $f$ after table look-up to get the output results.

$$\mathcal{A}_l = \mathcal{A}(g) + \mathcal{A}(f) = TC\log(K) + TD_O\log(C) \quad (20)$$

**Attention kernel arithmetic operations.** There are four encoding process $g$ and two aggregation $f$ processes.

$$\begin{aligned}\mathcal{A}_a &= \mathcal{A}(g_q, g_k, g_{qk}, g_v) + \mathcal{A}(f_{qk}, f_{qkv}) \\ &= (2TC_k + TC_t + D_kC_t)\log(K) \\ &\quad + T^2\log(C_k) + D_k^2\log(C_t) \\ &\Rightarrow (3T + D_k)C\log(K) + (T^2 + D_k^2)\log(C), \\ &\quad \text{if } C = C_k = C_t\end{aligned} \quad (21)$$

Using the designed kernels, we are able to accelerate the inference of a given attention-based neural network by converting its key operations to table look-ups.

## VI. Training of DART

In this section, we present the training scheme of DART, an exemplar of our approach to distill knowledge from an attention-based neural network to a hierarchy of tables towards practical prefetching. The key steps include data preparation (Section VI-A), training of an attention-based model (Section VI-B), configuring table structure based on prefetcher constraints (Section VI-C), reducing model complexity through knowledge distillation (Section VI-D), and layer-wise tabularization using the designed kernels (Section VI-E).

### A. Data Preparation

We extract memory access trace from the last level cache. We process the extracted trace following TransFetch [9] to get the input sequences and labels for training the memory access prediction models and the tables.

**Segmented address input.** We dissect a block address into $S = \lceil \frac{p}{c} \rceil + 1$ segments for $p$-bit page address and $c$-bit block index, each housing $c$ bits. This process maps a memory address to an $s$-dimensional vector, transforming a $T$-length sequence to a $T \times S$ matrix for attention-based model input.

**Delta bitmap labels.** We use a bitmap that each position indicates a delta between upcoming and current memory addresses. If a delta that appears in a near future (within a look-forward window) and falls in a range, the corresponding bit in the bitmap is set to 1. This approach enables the model to issue multiple predictions simultaneously.

### B. Attention-Based Neural Network Training

We train a large attention-based neural network for memory access prediction. This model is to pursue high prediction performance without taking the prefetcher design constraints into consideration.

**Model architecture.** We use the commonly used architecture, visualized in Figure 6, employing segmented addresses as inputs into the Transformer encoder layers, where each encoder layer consists of MSA mechanisms and FFNs. Following the encoder layers, the processed data is forwarded to a classification head that generates a delta bitmap, indicating multiple predicted address deltas to the current address.

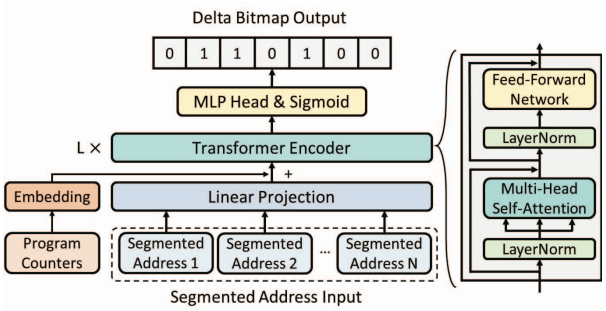



Fig. 6: Attention-based network for memory access prediction.

**Loss function.** We train the model as multi-label classification using Binary Cross-Entropy (BCE) loss.

### C. Table Configuration

Given design constraints on latency $\tau$ and storage $s$, we devise a table configurator to determine the structure of the final hierarchy of tables. We analyze the entire model's latency and storage, then present the table configurator mechanism.

TABLE I: Notations for model structure configuration

| Configuration | Value | Parameter | Value |
|---|---|---|---|
| Input history address | $T_I$ | Feed forward dimension | $D_F$ |
| Transformer input patches | $T_T$ | Output delta bitmap size | $D_O$ |
| Input address dimension | $D_I$ | Transformer heads | $H$ |
| Attention dimension | $D_A$ | Transformer layer | $L$ |

TABLE II: Notations for the table configuration shown in the format of $\langle$prototypes $K$, subspaces $C\rangle$

| Layer | Configuration | Layer | Configuration |
|---|---|---|---|
| Input Linear | $\langle K_I, C_I\rangle$ | Attention | $\langle K_A, C_A\rangle$ |
| Feed Forward | $\langle K_F, C_F\rangle$ | Output Linear | $\langle K_O, C_O\rangle$ |

*1) Entire Model Latency and Storage Cost:* For a network architecture in Figure 6, we use notations for the model configuration as Table I and notations for the table configuration in II. The tabularized entire model latency is shown in Equation 22 and the tabularized entire model storage is shown in Equation 23, where $\mathcal{L}_{ln}$ and $\mathcal{S}_{ln}$ are the latency and storage for a Layer Normalization operation, $\mathcal{L}_\sigma$ and $\mathcal{S}_\sigma$ are for output Sigmoid activation function latency and storage cost.

$$\begin{aligned}\mathcal{L} = &\underbrace{\mathcal{L}_l(K_I, C_I)}_{\text{Input linear}} + \underbrace{\mathcal{L}_{ln} + \mathcal{L}_l(K_O, C_O) + \mathcal{L}_\sigma}_{\text{Output linear}} \\ &+ \underbrace{L\Big[\underbrace{2\mathcal{L}_{ln} + 2\mathcal{L}_l(K_A, C_A) + \mathcal{L}_a(K_A, C_A)}_{\text{Multi-head self-attention}} + \underbrace{2\mathcal{L}_l(K_F, C_F)}_{\text{Feed forward}}\Big]}_{\text{Transformer encoder layers}}\end{aligned} \quad (22)$$

$$\mathcal{S} = \underbrace{2\mathcal{S}_l(T_I, D_A, K_I, C_I)}_{\text{Input linear}} + \underbrace{\mathcal{S}_{ln} + \mathcal{S}_l(T_T, D_O, K_O, C_O) + \mathcal{S}_\sigma}_{\text{Output linear}} + L[2\mathcal{S}_{ln} + \mathcal{S}_l(T_T, 3HD_A, K_A, C_A) + \underbrace{\mathcal{S}_a(T_T, D_A, K_A, C_A) + \mathcal{S}_l(T_T, D_A, K_A, C_A)}_{\text{Multi-head self-attention}} + \underbrace{\mathcal{S}_{ln} + \mathcal{S}_l(T_T, D_F, K_F, C_F) + \mathcal{S}_l(T_T, D_A, K_F, C_F)}_{\text{Feed forward}}]_{\text{Transformer encoder layers}} \tag{23}$$

*2) Table Configurator:* We pre-define a list of designs for the model configuration variables $(D_I, D_A, D_F, D_O, H, L)$ and table configuration variables $(K, C)$. Based on Equation 22 and Equation 23, we generate a configuration dictionary that maps a specific configuration to its latency and storage cost. Based on the dictionary, we design a table configurator that uses a latency-major greedy approach to provide a valid configuration. It first finds configurations with the highest latency smaller than $\tau$. Under this latency, it finds the configuration with maximum storage smaller than $s$. If no such configuration exists, it moves to the lower latency configurations and continues the process, until the configuration meets both the latency and storage constraints.

### D. Complexity Reduction via Knowledge Distillation

The table configurator identifies valid model configurations, adhering to design constraints, which often result in smaller models than our trained large ones (Section VI-B). We use Knowledge Distillation (KD) [37] to transfer knowledge from the larger teacher model to the compact student model. Given that our task is multi-label classification, we utilize BCE loss and introduce a T-Sigmoid function (Equation 24)—a softened variant inspired by T-Softmax [37]—to refine probability distributions over multiple class outputs.

$$z_i = p(y_i)_{t=T} = \sigma\left(\frac{y_i}{T}\right) = \frac{1}{1 + e^{-y_i/T}} \tag{24}$$

The complete loss function ($Loss$) encompasses both the BCE loss ($Loss_{BCE}$) and the soft KD loss ($Loss_{KD}$), as defined in Equation 25:

$$\begin{aligned} Loss_{\text{KD}} &= \sum_{k=1}^{q} \text{KL}\left(\left[z_i^{tch}, 1 - z_i^{tch}\right] \parallel \left[z_i^{stu}, 1 - z_i^{stu}\right]\right) \\ Loss &= \lambda Loss_{\text{KD}} + (1 - \lambda) Loss_{\text{BCE}} \end{aligned} \tag{25}$$

where $\text{KL}(\cdot\|\cdot)$ is the Kullback-Leibler divergence [46], $\lambda$ is a hyper-parameter tuning the weights of the two losses, $z_i^{tch}$ and $z_i^{stu}$ are T-Sigmoid output of teacher and student models.

### E. Layer-Wise Tabularization with Fine-Tuning

*1) Layer-Wise Tabularization:* We convert the compressed model from KD to a hierarchy of tables layer by layer. The table configuration is given by the table configurator. The tabularization process is shown in Algorithm 1. For each neural network layer, we check the operation and use the corresponding tabularization kernel introduced in Section V, including linear kernel (line 10) and attention kernel (line 13).

**Algorithm 1** Layer-Wise Tabularization with Fine-Tuning

1: **Input:** Trained $N$-layer model $\mathcal{M}$, Training input data $\mathcal{D}$
2: **Initialize:** Model layer $i$ output $L[i] \leftarrow \mathcal{M}[0:i](\mathcal{D})$
3: **Initialize:** Table hierarchy $\mathcal{T}$, fine-tune epoch $E$
4: **Initialize:** Configuration lists prototypes $K$, subspace $C$
5: **for** $i$ **in** 0 to $N-1$ **do** ▷ Layer-wise tabularization
6: **if** $\mathcal{M}[i]$ is a linear layer **then**
7: **if** $i > 0$ **then**: ▷ Layer fine-tuning
8: $\mathcal{M}'[i] \leftarrow \text{FINETUNE}(\mathcal{M}[i], \mathcal{T}(\mathcal{D}), L[i], E)$
9: **end if**
10: $\mathcal{T}_{\text{linear}} \leftarrow \text{LINEARKERNEL}(\mathcal{M}'[i], K[i], C[i])$
11: Push $\mathcal{T}_{\text{linear}}$ to $\mathcal{T}$
12: **else if** $\mathcal{M}[i]$ is an attention layer **then**
13: $\mathcal{T}_{\text{attn}} \leftarrow \text{ATTENTIONKERNEL}(\mathcal{M}[i], K[i], C[i])$
14: Push $\mathcal{T}_{\text{attn}}$ to $\mathcal{T}$
15: **else if** $\mathcal{M}[i]$ is a Sigmoid **then**
16: Push Sigmoid lookup table approximation to $\mathcal{T}$
17: **else if** $\mathcal{M}[i]$ is a LayerNorm **then**
18: Push LayerNorm parameters and operations to $\mathcal{T}$
19: **end if**
20: **end for**
21: **Output:** $\mathcal{T}$

The output Sigmoid activation function can be approximated by a fixed lookup table [47]. Layer Normalization process is dimension-wise simple arithmetic operation without matrix multiplication, so we directly use the original operation. We initialize the output hierarchy of tables as an empty sequence $\mathcal{T}$ and push each converted layer to $\mathcal{T}$.

*2) Fine-Tuning:* For a hidden linear layer $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{B}$ (Equation 1) of model $\mathcal{M}$, denoted as $\mathcal{M}[i]$, where $i > 0$ is the layer index, we fine-tune the layer weights to achieve higher approximation performance (Algorithm 1 line 8). Assuming we have tabluarized layers prior to $\mathcal{M}[i]$, the input of $\mathcal{M}[i]$, which is the output of $\mathcal{M}[i-1]$, is now an approximation $\hat{\boldsymbol{X}} = \boldsymbol{X} + \epsilon_x$ with the error $\epsilon_x$ introduced by tabularization. This leads to the error in the tabularized layer output $\boldsymbol{W}\hat{\boldsymbol{X}} + \boldsymbol{b} = \boldsymbol{Y} + \epsilon_y$. With an increasing number of layers being tarbularized, the errors accumulate, resulting in low approximation performance. To ameliorate the error accumulation problem, we fine-tune the layer weight $\boldsymbol{W}$ and bias $\boldsymbol{b}$. We initialize $\boldsymbol{W}$ and $\boldsymbol{b}$ as in the trained model, use the tabularization approximated $\hat{\boldsymbol{X}}$ as input and the original layer output $\boldsymbol{Y}$ as target, and performs $E$ epochs of layer training. We use the Mean Squared Error (MSE) loss function to learn the updated layer $\mathcal{M}'[i]$ with new weights $\boldsymbol{W'}$ and $\boldsymbol{b'}$:

$$(\boldsymbol{W'}, \boldsymbol{b'}) = \underset{\boldsymbol{W}, \boldsymbol{b}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d} \left(\boldsymbol{Y}_{ij} - (\boldsymbol{W}\hat{\boldsymbol{X}}_{ij} + \boldsymbol{b})\right)^2 \tag{26}$$

After fine-tuning, we perform tabularization based on the updated layer $\mathcal{M}'[i]$. In this way, we train a table that imitates the output of a NN layer, going beyond mere approximation of the model weights. This novel approach draws inspiration

from KD, framing the process as distilling knowledge from a NN layer into a table.

## VII. Evaluation

### A. Experimental Setup

*1) Simulator:* Consistent with existing literature, we use ChampSim [48] to generate traces and assess our approach. ChampSim simulates a modern heterogeneous multi-core system with an arbitrary memory hierarchy. The simulation parameters are detailed in Table III, with prefetchers simulated at the last-level cache (LLC).

TABLE III: Simulation parameters

| Parameter | Value |
|---|---|
| CPU | 4 GHz, 4 cores, 4-wide OoO,<br>256-entry ROB, 64-entry LSQ |
| L1 I-cache | 64 KB, 8-way, 8-entry MSHR, 4-cycle |
| L1 D-cache | 64 KB, 12-way, 16-entry MSHR, 5-cycle |
| L2 Cache | 1 MB, 8-way, 32-entry MSHR, 10-cycle |
| LL Cache | 8 MB, 16-way, 64-entry MSHR, 20-cycle |
| DRAM | $t_{RP} = t_{RCD} = t_{CAS} = 12.5$ ns<br>2 channels, 8 ranks, 8 banks<br>32K rows, 8GB/s bandwidth per core |

*2) Benchmark:* We evaluate DART on widely-used benchmarks: *SPEC CPU 2006* [49] and *SPEC CPU 2017* [50]. Table IV details the workloads we use and the LLC trace statistics extracted using ChampSim. The workloads consist of a diverse number of deltas and pages, which are representative to various scenarios.

TABLE IV: Benchmark application memory trace statistics

| Benchmark | Application | # Address | # Page | # Delta |
|---|---|---|---|---|
| SPEC 2006 | 410.bwaves | 236.5K | 3.7K | 14.4K |
| | 433.milc | 170.7K | 19.8K | 15.8K |
| | 437.leslie3d | 104.3K | 1.7K | 3.6K |
| | 462.libquantum | 347.8K | 5.4K | 0.5K |
| SPEC 2017 | 602.gcc | 195.8K | 3.4K | 4.9K |
| | 605.mcf | 176.0K | 3.7K | 207.7K |
| | 619.lbm | 121.8K | 1.9K | 1.2K |
| | 621.wrf | 188.5K | 3.3K | 13.7K |

TABLE V: Configurations of models

| | NN Config | | | Table Config | | Complexity | | |
|---|---|---|---|---|---|---|---|---|
| | $L$ | $D$ | $H$ | $K$ | $C$ | ($\mathcal{L}$/cycle) | ($\mathcal{S}$/B) | ($\mathcal{A}$) |
| Teacher | 4 | 256 | 8 | - | - | 16.5K | 86.2M | 98.3M |
| Student | 1 | 32 | 2 | - | - | 908 | 827.4K | 134.7K |
| DART | 1 | 32 | 2 | 128 | 2 | 97 | 864.4K | 11.0K |

*3) Model Configurations:* Table V outlines configurations for the large Teacher model, compressed Student model, and an exemplar table-based predictor for DART. All models are implemented under full parallelism, the complexity of DART is based on Section V-C, whereas the NN-based models Teacher and Student is examined under systolic array implementation for matrix multiplications [51]. We tailor the NN's layers ($L$), hidden dimensions ($D$), and heads ($H$). For tabularized models, consistent number of prototypes ($K$) and subspaces ($C$) are set across operations.

From our configuration, we measure latency $\mathcal{L}$, storage $\mathcal{S}$, and operations $\mathcal{A}$. Against the Teacher, DART achieves $170\times$ acceleration, $102\times$ compression, and a $99.99\%$ reduction in operations. Compared to the Student of similar size, DART offers $9.36\times$ acceleration, cutting operations by $91.83\%$.

*4) Metrics:* We evaluate multi-label memory access prediction using F1-score [52]. We evaluate the prefetching performance through simulation on ChampSim. In the simulation runs, we record the cache performance and report the prefetch accuracy, coverage, and IPC improvement [53]. The running time of the workloads on ChampSim cannot indicate the direct application performance. Instead, the IPC performance is reported based on the number of instructions and the cycles of simulation, which accurately indicates the prefetcher performance.

### B. Memory Access Patterns

Figure 7 shows the diverse memory access patterns of the benchmark applications in three scaled dimensions: instruction ID, page address, and block delta of consecutive accesses. The performance of the original large attention-based memory access prediction model is shown in Table VI as teacher models. The distribution of memory access patterns impacts the model prediction. We observe that applications showing fewer deltas are easier to predict compared with applications with more deltas under a similar number of pages, e.g., 602.gcc vs 605.mcf. We also observe that under a comparable number of deltas, applications with a smaller number of pages are easier to predict, e.g., 410.bwaves vs 433.milc.

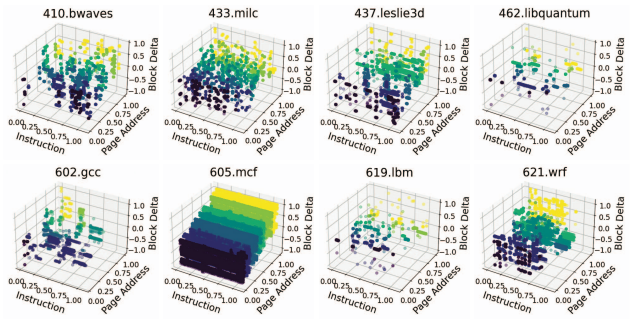



Fig. 7: Visualization of memory access patterns in the benchmark applications. The patterns show diversity in instructions, deltas, and pages, representing various scenarios.

### C. Evaluation of Multi-Label Knowledge Distillation

Table VI shows the F1-score of the teacher model, the student model trained without knowledge distillation (Stu w/o KD), and the student model trained using the proposed multi-label knowledge distillation with T-Sigmoid activation function (Student). Results show that the compressed models trained using KD raise the mean F1-score from 0.751 to 0.783, which is only 0.005 lower than the $107\times$ larger teacher model.

TABLE VI: F1-score of the teacher model and the student models trained with and without knowledge distillation

| Models | Applications | | | | | | | | Mean |
|---|---|---|---|---|---|---|---|---|---|
| | 410. bwav | 433. milc | 437. lesl | 462. libq | 602 gcc | 605. mcf | 619. lbm | 621. wrf | |
| Teacher | 0.969 | 0.863 | 0.599 | 0.992 | 0.952 | 0.551 | 0.742 | 0.638 | 0.788 |
| Stu w/o KD | 0.923 | 0.715 | 0.545 | 0.991 | 0.946 | 0.545 | 0.679 | 0.660 | 0.751 |
| Student | **0.923** | **0.789** | **0.552** | **0.991** | **0.947** | **0.655** | **0.751** | **0.660** | **0.783** |

### D. Evaluation of Tabularization

We assess layer-wise tabularization performance under varying table configurations, such as the number of prototypes ($K$) and subspaces ($C$), without fine-tuning. Our model structure remains fixed as the DART model (see Table V), with the only variation being either $K$ or $C$. Figures 8 and 9 illustrate how adjusting $K$ and $C$ impacts DART's F1-score. The results reveal that increasing $K$ significantly boosts prediction performance when $K$ exceeds 128, with $K = 1024$ achieving a 10.9% higher F1 score compared to $K = 16$. In contrast, the impact of higher $C$ on F1-score is less pronounced, with $C = 8$ outperforming $C = 1$ by 6.6%. While higher values of both $K$ and $C$ generally enhance tabularization performance, it's important to note that this comes at a substantial increase in latency and storage costs. Figure 10 demonstrates a linear relationship between model latency and $\log(K)$ as well as $\log(C)$, while storage costs exhibit exponential growth.

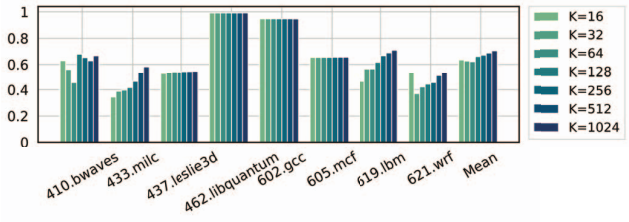



Fig. 8: F1-scores with varying DART prototypes $K$.

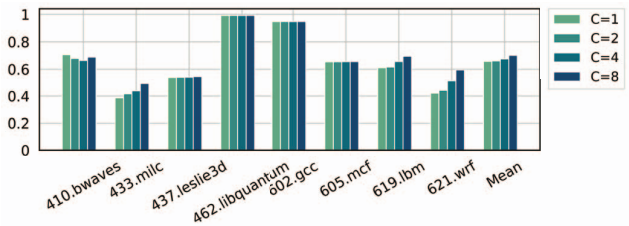



Fig. 9: F1-scores with varying DART subspaces $C$.

### E. Evaluation of Fine-Tuning

We compare cosine similarity between Student NN and its tabularized models: DART with fine-tuning vs. DART without (DART w/o FT). Figure 11 shows that fine-tuning effectively raises the layer cosine similarity, especially for the layers close to the output. Table VII shows the F1-score of the above models. On average, DART achieves an F1-score of 0.699, representing a 5.75% gain over DART w/o FT but 0.084 lower than the Student model.

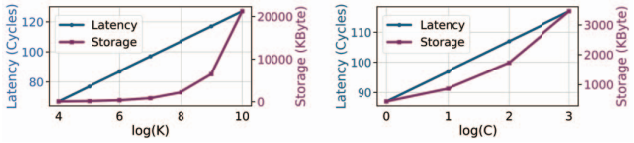



Fig. 10: Latency and storage cost under various $K$ and $C$. Model latency scales linearly with $\log(K)$ and $\log(C)$, whereas storage experiences exponential growth.

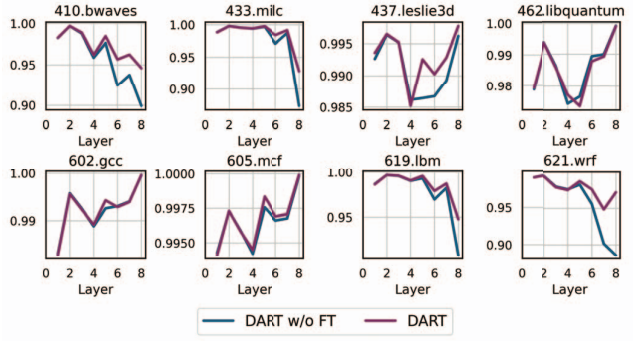



Fig. 11: Layer-wise cosine similarity comparison between DART without fine-tuning (DART w/o FT) and DART.

TABLE VII: F1-score of DART for memory access prediction

| Models | Applications | | | | | | | | Mean |
|---|---|---|---|---|---|---|---|---|---|
| | 410. bwav | 433. milc | 437. lesl | 462. libq | 602 gcc | 605. mcf | 619. lbm | 621. wrf | |
| DART w/o FT | 0.679 | 0.416 | 0.541 | 0.991 | 0.946 | 0.655 | 0.617 | 0.443 | 0.661 |
| DART | **0.790** | **0.480** | **0.544** | **0.991** | **0.947** | **0.655** | **0.638** | **0.543** | **0.699** |

### F. Evaluation of Prefetching

*1) Prefetcher Configurations:* As shown in Table VIII, we generate two more valid configurations using our table configurator given varied design constraints. We evaluate the prefetching performance of DART, along with small and large variants: DART-S and DART-L.

TABLE VIII: Configurations of DART under various prefetcher design constraints

| Prefetcher | Constraints ($\tau$/cycle, $s$/B) | Configuration $(L, D, H, K, C)$ | Latency ($\mathcal{L}$/cycle) | Storage ($\mathcal{S}$/B) | Ops ($\mathcal{A}$) |
|---|---|---|---|---|---|
| DART-S | 60, 30K | 1, 16, 2, 16, 1 | 57 | 29.9K | 1.6K |
| DART | 100, 1M | 1, 32, 2, 128, 2 | 97 | 864.4K | 11.0K |
| DART-L | 200, 4M | 2, 32, 2, 256, 2 | 191 | 3.75M | 17.5K |

*2) Baseline Prefetchers:* Table IX show all the prefetchers we implemented for evaluation. There are three types of baseline prefetchers: 1) practical rule-based prefetchers, including BO [6] and ISB [7], 2) neural network based prefetchers, including TransFetch [9] and Voyager [16], and 3) idealized versions of TransFetch and Voyager, which disregard practical

hardware implementation constraints and assume zero latency in ChampSim simulation.

TABLE IX: Configurations of prefetchers for evaluation

| Prefetcher | Storage | Latency | Table | ML | Mechanism |
|---|---|---|---|---|---|
| BO [6] | 4KB | ≈ 60 | ✓ | | Spatial locality |
| ISB [7] | 8KB | ≈ 30 | ✓ | | Temporal locality |
| TransFetch [9] | 13.8MB | 4.5K | | ✓ | Attention |
| Voyager [16] | 14.9MB | 27.7K | | ✓ | LSTM |
| TransFetch-I | - | 0 | | ✓ | Attention (Ideal) |
| Voyager-I | - | 0 | | ✓ | LSTM (Ideal) |
| DART | 29.9K-3.75M | 57-191 | ✓ | ✓ | Attention |

*3) Prefetching Performance:* Figure 12 shows the prefetch accuracy of DART and the baselines. DART-S, DART, and DART-L achieve prefetch accuracy at 80.6%, 80.7%, and 82.5%, respectively. While TransFetch-I and Voyager-I achieve the two highest accuracy values at 89.6% and 95.1%, their performance drops significantly with latency introduced in simulation to 78.6% and 49.9%. BO as a rule-based prefetcher shows high prefetch accuracy at 89.4%.

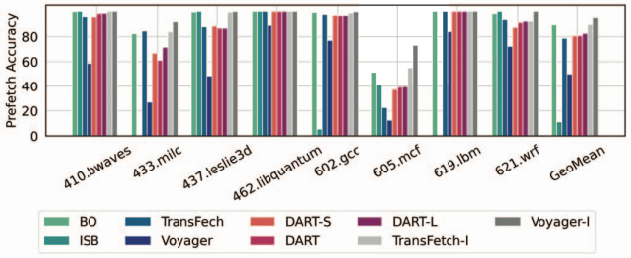



Fig. 12: Prefetch accuracy of DART and the baselines.

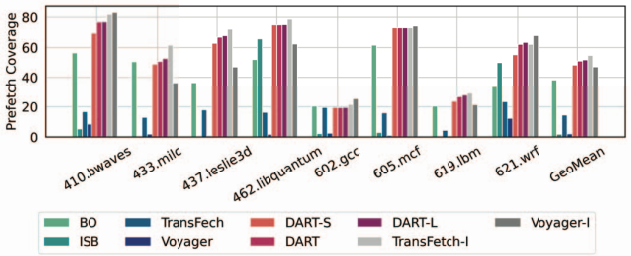



Fig. 13: Prefetch coverage of DART and the baselines.

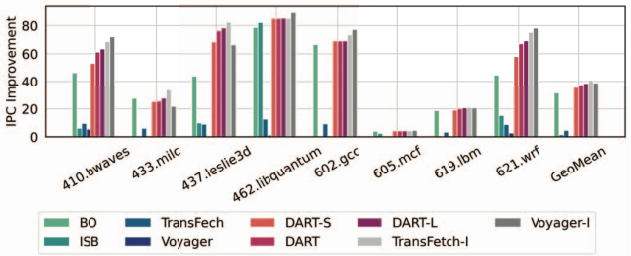



Fig. 14: IPC Improvement using DART and the baselines.

Figure 13 shows the prefetch coverage of DART and the baselines. DART-S, DART, and DART-L achieve coverage at 48.3%, 51.0%, and 51.8%, respectively. TransFetch-I and Voyager-I show coverage at 54.7% and 47.0%, but the values drop significantly to 14.4% and 2.1% with latency introduced.

Figure 14 shows the overall IPC improvement when applying a prefetcher. DART-S, DART, and DART-L achieve IPC improvement at 35.4%, 37.6%, and 38.5%, respectively. All DART variants outperform the rule-based prefetchers BO (31.5%) and ISB (1.6%) and outperform the NN-based prefetchers TransFetch (4.5%) and Voyager (0.38%). The basic configuration DART achieves 37.6% IPC improvement, outperforming BO by 6.1% and outperforming TransFetch by 33.1%, only 3.3% lower compared to the highest ideal and impractical prefetcher TransFetch-I at 40.9%. Even the smallest variant DART-S achieves 35.4% IPC improvement under storage and latency comparable to rule-based prefetchers, outperforming all other baseline prefetchers besides the ideal NN-based prefetchers TransFetch-I and Voyager-I, which it underperforms by 5.5% and 3.4% respectively.

## VIII. Conclusion

We presented DART, a prefetcher exemplifying our innovative approach, which distills knowledge from an attention-based neural network into a hierarchy of tables for practical NN-based data prefetching. The keys to our approach include training an attention-based NN for memory access prediction, distilling this model into a compact form to meet design constraints, and implementing tabularization to create a hierarchy of tables. DART maintains the prediction performance while significantly reducing the NN-based predictor's arithmetic operations and inference latency. DART outperforms state-of-the-art table-based prefetchers under comparable storage and latency cost while achieving close performance to the impractical ML-based prefetchers w.r.t. IPC improvement. In future work, we aim to optimize the tabularized model by reducing encoding and aggregation overheads. Additionally, we plan to explore converting multiple layers into a single table to further reduce latency, storage, and operations, enhancing practicality towards real-world ML-based prefetching.

## Acknowledgment

This work has been supported by the U.S. National Science Foundation (NSF) under grant CNS-2009057 and SaTC-2104264, as well as the DEVCOM Army Research Lab (ARL) under grant W911NF2220159.

**Distribution Statement A**: Approved for public release. Distribution is unlimited.

## References

[1] S. P. Vander Wiel and D. J. Lilja, "When caches aren't enough: Data prefetching techniques," *Computer*, vol. 30, no. 7, pp. 23–30, 1997.

[2] C. Carvalho, "The gap between processor and memory speeds," in *Proc. of IEEE International Conference on Control and Automation*, 2002.

[3] M. Dubois, M. Annavaram, and P. Stenström, *Parallel computer organization and design*. cambridge university press, 2012.

[4] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 1998, pp. 357–368.

[5] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–35, 2016.

[6] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

[7] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 247–259.

[8] P. Zhang, R. Kannan, A. Srivastava, A. V. Nori, and V. K. Prasanna, "Resemble: Reinforced ensemble framework for data prefetching," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 282–292.

[9] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 103–112.

[10] P. Zhang, R. Kannan, and V. K. Prasanna, "Phases, modalities, spatial and temporal locality: Domain specific ml prefetcher for accelerating graph analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: the road to compact ml-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 461–470.

[13] A. Srivastava, T.-Y. Wang, P. Zhang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "Memmap: Compact and generalizable meta-lstm models for memory access prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2020, pp. 57–68.

[14] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "Raop: Recurrent neural network augmented offset prefetcher," in *The International Symposium on Memory Systems*, 2020, pp. 352–362.

[15] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. De Rose, R. Kannan, and V. K. Prasanna, "C-memmap: clustering-driven compact, adaptable, and generalizable meta-lstm models for memory access prediction," *International Journal of Data Science and Analytics*, pp. 1–14, 2021.

[16] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 861–873.

[17] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[19] P. Zhang, R. Kannan, X. Tong, A. V. Nori, and V. K. Prasanna, "Sharp: Software hint-assisted memory access prediction for graph analytics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–8.

[20] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[21] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[22] K. Marino, P. Zhang, and V. K. Prasanna, "Me-vit: A single-load memory-efficient fpga accelerator for vision transformers," 2023.

[23] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[24] D. Blalock and J. Guttag, "Multiplying matrices without multiplying," in *International Conference on Machine Learning*. PMLR, 2021, pp. 992–1004.

[25] M. Islam, S. Banerjee, M. Meswani, and K. Kavi, "Prefetching as a potentially effective technique for hybrid memory optimization," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 220–231. [Online]. Available: https://doi.org/10.1145/2989081.2989129

[26] H. Choi and S. Park, "A survey of machine learning-based system performance optimization techniques," *Applied Sciences*, vol. 11, no. 7, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/7/3235

[27] P. Zhang, R. Kannan, A. V. Nori, and V. K. Prasanna, "A2p: Attention-based memory access prediction for graph analytics," 2022.

[28] C. Yang, X. Man, and J. Shao, "G&l: An attention-based model for improving prefetching in solid-state drives," in *2023 International Joint Conference on Neural Networks (IJCNN)*, 2023, pp. 1–8.

[29] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," in *2015 IEEE 17th International Conference on High Performance Computing and Communications*, Aug. 2015, pp. 383–389.

[30] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

[31] F. Eris, M. S. Louis, K. Eris, J. L. Abellan, and A. Joshi, "Puppeteer: A random forest-based manager for hardware prefetchers across the memory hierarchy," 2022.

[32] E. S. Alcorta, M. Madhav, S. Tetrick, N. J. Yadwadkar, and A. Gerstlauer, "Lightweight ml-based runtime prefetcher selection on many-core platforms," 2023.

[33] S. Mohapatra and B. Panda, "Drishyam: An image is worth a data prefetcher," in *32nd ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '23)*, ACM. ACM New York, NY, USA, October 2023. [Online]. Available: https://doi.org/10.1145/3528416.3530224

[34] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," 2019.

[35] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[36] T. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," 10 2013, pp. 6655–6659.

[37] G. Hinton, O. Vinyals, J. Dean *et al.*, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, vol. 2, no. 7, 2015.

[38] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[39] N. Gupta, P. Zhang, R. Kannan, and V. Prasanna, "Packd: Pattern-clustered knowledge distillation for compressing memory access prediction models," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–7.

[40] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[41] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," 2018.

[42] S. Abi-Karam, Y. He, R. Sarkar, L. Sathidevi, Z. Qiao, and C. Hao, "Gengnn: A generic fpga framework for graph neural network acceleration," 2022.

[43] D. P. Francis and K. Raimond, "A practical streaming approximate matrix multiplication algorithm," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 1, p. 1455–1465, jan 2022. [Online]. Available: https://doi.org/10.1016/j.jksuci.2018.09.010

[44] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin, "Shiftaddnet: A hardware-inspired deep network," 2020.

[45] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "Deepshift: Towards multiplication-less neural networks," 2021.

[46] J. M. Joyce, "Kullback-leibler divergence," in *International encyclopedia of statistical science*. Springer, 2011, pp. 720–722.

[47] P. K. Meher, "An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*. IEEE, 2010, pp. 91–95.

[48] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Ar-

chitectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.
[49] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.
[50] S. CPU2017", "The standard performance evaluation corporation," https://www.spec.org/cpu2017/, 2017.
[51] H. T. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979, pp. 256–282.
[52] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *European conference on information retrieval*. Springer, 2005, pp. 345–359.
[53] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 126–140, 2004.